Veriopt

August 31, 2022

Abstract

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

Contents

1 Runtime Values and Arithmetic

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

lemma $-((x::float)-y) = (y-x)$
  by *simp*

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*}

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

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

setup-lifting *type-definition-IntWidth*

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

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*

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

instantiation *Value* :: *minus*
begin

definition *minus-Value* :: *Value* ⇒ *Value* ⇒ *Value* **where**
  minus-Value = intval-sub

instance proof qed
end

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

instantiation *Value* :: *times*
begin

definition *times-Value* :: *Value* ⇒ *Value* ⇒ *Value* **where**
  times-Value = intval-mul

instance proof qed

end

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*

instantiation *Value* :: *divide*
begin

definition *divide-Value* :: *Value* ⇒ *Value* ⇒ *Value* **where**
divide-Value = *intval-div*

instance proof qed
end

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*

instantiation *Value* :: *modulo*
begin

definition *modulo-Value* :: *Value* ⇒ *Value* ⇒ *Value* **where**
modulo-Value = *intval-mod*

instance proof qed
end

1.2 Bitwise Operators and Comparisons

context
includes *bit-operations-syntax*
begin

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

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

intval-or - - $=$ *UndefVal*

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

fun *intval-short-circuit-or* :: *Value* $\Rightarrow$ *Value* $\Rightarrow$ *Value* **where**
intval-short-circuit-or (*IntVal b1 v1*) (*IntVal b2 v2*) $=$ *bool-to-val-bin b1 b2* ((($v1 \neq 0$) $\vee$ ($v2 \neq 0$))) |
intval-short-circuit-or - - $=$ *UndefVal*

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

fun *intval-not* :: *Value* $\Rightarrow$ *Value* **where**
intval-not (*IntVal t v*) $=$ *new-int t* (*NOT v*) |
intval-not - $=$ *UndefVal*

fun *intval-negate* :: *Value* $\Rightarrow$ *Value* **where**
intval-negate (*IntVal t v*) $=$ *new-int t* ($-$ *v*) |
intval-negate - $=$ *UndefVal*

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

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

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

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

1.3 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 *intval*(*intval-narrow 16 8* (*IntVal32* (*512* − *2*)))

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

1.4 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 *signed-shiftr* :: '*a* :: *len word* ⇒ *nat* ⇒ '*a* :: *len word* (**infix** >> *75*)
where
  signed-shiftr 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* (*val AND* (*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 = *mask b1 AND* (*NOT* (*mask* (*b1* − *shift*) :: *int64*)) *in*
     (*if int-signed-value b1 v1* < *0*
      then new-int b1 (*ones OR* (*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*

end

2 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

3 Nodes

3.1 Types of 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*, *true-*

Value, *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 Hierarchy of Nodes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

fun $\mathit{is\text{-}DeoptimizingFixedWithNextNode} :: \mathit{IRNode} \Rightarrow \mathit{bool}$ **where**

is-DeoptimizingFixedWithNextNode n = ((*is-AbstractNewObjectNode* n) $\vee$ (*is-FixedBinaryNode* n))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

is-Node n = ((*is-ValueNode* n) ∨ (*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**

$is\text{-}SingleMemoryKill\ n = ((is\text{-}BytecodeExceptionNode\ n) \vee (is\text{-}ExceptionObjectNode\ n) \vee (is\text{-}InvokeNode\ n) \vee (is\text{-}InvokeWithExceptionNode\ n) \vee (is\text{-}KillingBeginNode\ n) \vee (is\text{-}StartNode\ n))$

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

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

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

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

fun $is\text{-}VirtualizableAllocation :: IRNode \Rightarrow bool$ **where**
$is\text{-}VirtualizableAllocation\ n = ((is\text{-}NewArrayNode\ n) \vee (is\text{-}NewInstanceNode\ n))$

fun $is\text{-}Unary :: IRNode \Rightarrow bool$ **where**
$is\text{-}Unary\ n = ((is\text{-}LoadFieldNode\ n) \vee (is\text{-}LogicNegationNode\ n) \vee (is\text{-}UnaryNode\ n) \vee (is\text{-}UnaryOpLogicNode\ n))$

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

fun $is\text{-}BinaryCommutative :: IRNode \Rightarrow bool$ **where**
$is\text{-}BinaryCommutative\ n = ((is\text{-}AddNode\ n) \vee (is\text{-}AndNode\ n) \vee (is\text{-}IntegerEqualsNode\ n) \vee (is\text{-}MulNode\ n) \vee (is\text{-}OrNode\ n) \vee (is\text{-}XorNode\ n))$

fun $is\text{-}Canonicalizable :: IRNode \Rightarrow bool$ **where**
$is\text{-}Canonicalizable\ n = ((is\text{-}BytecodeExceptionNode\ n) \vee (is\text{-}ConditionalNode\ n) \vee (is\text{-}DynamicNewArrayNode\ n) \vee (is\text{-}PhiNode\ n) \vee (is\text{-}PiNode\ n) \vee (is\text{-}ProxyNode\ n) \vee (is\text{-}StoreFieldNode\ n) \vee (is\text{-}ValueProxyNode\ n))$

fun $is\text{-}UncheckedInterfaceProvider :: IRNode \Rightarrow bool$ **where**
$is\text{-}UncheckedInterfaceProvider\ n = ((is\text{-}InvokeNode\ n) \vee (is\text{-}InvokeWithExceptionNode\ n) \vee (is\text{-}LoadFieldNode\ n) \vee (is\text{-}ParameterNode\ n))$

fun $is\text{-}Binary :: IRNode \Rightarrow bool$ **where**
$is\text{-}Binary\ n = ((is\text{-}BinaryArithmeticNode\ n) \vee (is\text{-}BinaryNode\ n) \vee (is\text{-}BinaryOpLogicNode\ n) \vee (is\text{-}CompareNode\ n) \vee (is\text{-}FixedBinaryNode\ n) \vee (is\text{-}ShortCircuitOrNode\ n))$

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*)) ∨

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

end

4 Stamp Typing

theory *Stamp*
imports *Values*
begin

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

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

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

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

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*

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 x y = ((*stpi-upper x*) < (*stpi-lower y*))

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

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

5 Graph Representation

theory *IRGraph*
  imports
    IRNodeHierarchy
    Stamp
    HOL−Library.FSet
    HOL.Relation
begin

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

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

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

setup-lifting *type-definition-IRGraph*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

code-datatype *irgraph*

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

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

lemma *dom-eq*:
assumes $\forall\, x \in$ *set xs*. *fst* (*snd x*) $\neq$ *NoNode*
shows *filter-none* (*map-of xs*) = *dom* (*map-of xs*)
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* $\notin$ *ids g*
shows *kind g nid* = *NoNode*
using *assms ids-some* **by** *blast*

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

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

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

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

lemma [*simp*]: *finite* (*dom g*) $\longrightarrow$ *kind* (*Abs-IRGraph g*) = (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* $\longrightarrow$
(*if k* $\neq$ *NoNode then kind gup nid* = *k* $\wedge$ *stamp gup nid* = *s else kind gup nid* = *kind g nid*)
proof (*cases k* = *NoNode*)
case *True*
then show *?thesis*
by (*simp add*: *add-node.rep-eq kind.rep-eq*)
next
case *False*
then show *?thesis*
by (*simp add*: *kind.rep-eq add-node.rep-eq stamp.rep-eq*)
qed

lemma *remove-node-lookup*:
gup = *remove-node nid g* $\longrightarrow$ *kind gup nid* = *NoNode* $\wedge$ *stamp gup nid* = *IllegalStamp*
by (*simp add*: *kind.rep-eq remove-node.rep-eq stamp.rep-eq*)

lemma *replace-node-lookup*[*simp*]:
gup = *replace-node nid* (*k*, *s*) *g* $\wedge$ *k* $\neq$ *NoNode* $\longrightarrow$ *kind gup nid* = *k* $\wedge$ *stamp gup nid* = *s*
by (*simp add*: *replace-node.rep-eq kind.rep-eq stamp.rep-eq*)

lemma *replace-node-unchanged*:
gup = *replace-node nid* (*k*, *s*) *g* $\longrightarrow$ ($\forall$ *n* $\in$ (*ids g* $-$ {*nid*}) . *n* $\in$ *ids g* $\wedge$ *n* $\in$ *ids gup* $\wedge$ *kind g n* = *kind gup n*)
by (*simp add*: *kind.rep-eq replace-node.rep-eq*)

5.0.1 Example Graphs

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

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

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

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

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

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

end

5.1 Control-flow Graph Traversal

theory
Traversal
imports
IRGraph
begin

type-synonym *Seen* = *ID set*

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

fun *nextEdge* :: *Seen* ⇒ *ID* ⇒ *IRGraph* ⇒ *ID option* **where**
nextEdge seen nid g =
(*let nids* = (*filter* (λ*nid'*. *nid'* ∉ *seen*) (*successors-of* (*kind g nid*))) *in*
(*if length nids* > *0 then Some* (*hd nids*) *else None*))

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

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

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

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

$Some\ (hd\ (sorted\text{-}list\text{-}of\text{-}set\ (IRGraph.predecessors\ g\ nid)))$
)
)

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

type-synonym $'a\ TraversalState = (ID \times Seen \times 'a)$

inductive *Step*
$:: ('a\ TraversalState \Rightarrow 'a) \Rightarrow IRGraph \Rightarrow 'a\ TraversalState \Rightarrow 'a\ TraversalState\ option \Rightarrow bool$
for *sa g* **where**
— Hit a BeginNode with an IfNode predecessor which represents the start of a basic block for the IfNode. 1. nid' will be the successor of the begin node. 2. Find the first and only predecessor. 3. Extract condition from the preceding IfNode. 4. Negate condition if the begin node is second branch (we've taken the else branch of the condition) 5. Add the condition or the negated condition to stack 6. Perform any stamp updates based on the condition using the registerNewCondition function and place them on the top of the stack of stamp information
$[\![kind\ g\ nid = BeginNode\ nid';$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

code-pred $(modes\colon i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool)\ Step$ **.**

end

5.2 Structural Graph Comparison

theory
Comparison
imports
IRGraph
begin

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

fun *find-ref-nodes* :: $IRGraph \Rightarrow (ID \rightharpoonup ID)$ **where**
$find\text{-}ref\text{-}nodes\ g = map\text{-}of$
$(map\ (\lambda n.\ (n,\ ir\text{-}ref\ (kind\ g\ n)))\ (filter\ (\lambda id.\ is\text{-}RefNode\ (kind\ g\ id))\ (sorted\text{-}list\text{-}of\text{-}set\ (ids\ g))))$

fun *replace-ref-nodes* :: $IRGraph \Rightarrow (ID \rightharpoonup ID) \Rightarrow ID\ list \Rightarrow ID\ list$ **where**
$replace\text{-}ref\text{-}nodes\ g\ m\ xs = map\ (\lambda id.\ (case\ (m\ id)\ of\ Some\ other \Rightarrow other \mid None \Rightarrow id))\ xs$

fun *find-next* :: $ID\ list \Rightarrow ID\ set \Rightarrow ID\ option$ **where**
$find\text{-}next\ to\text{-}see\ seen = (let\ l = (filter\ (\lambda nid.\ nid \notin seen)\ to\text{-}see)$
$in\ (case\ l\ of\ [] \Rightarrow None \mid xs \Rightarrow Some\ (hd\ xs)))$

inductive *reachables* :: $IRGraph \Rightarrow ID\ list \Rightarrow ID\ set \Rightarrow ID\ set \Rightarrow bool$ **where**
$reachables\ g\ []\ \{\}\ \{\}$ |
⟦$None = find\text{-}next\ to\text{-}see\ seen$⟧ $\Longrightarrow reachables\ g\ to\text{-}see\ seen\ seen$ |

$[\![$*Some n = find-next to-see seen*;
node = kind g n;
new = (inputs-of node) @ (successors-of node);
reachables g (to-see @ new) ({n} ∪ seen) seen' $]\!] \Longrightarrow$ *reachables g to-see seen seen'*

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

reachables **.**

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

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

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

fun *diffNodesInfo* :: *IRGraph* ⇒ *IRGraph* ⇒ (*ID* × *IRNode* × *IRNode*) *set* **where**
diffNodesInfo g1 g2 = {(*nid*, *kind g1 nid*, *kind g2 nid*) | *nid . nid* ∈ *diffNodesGraph g1 g2*}

fun *eqGraph* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
eqGraph isabelle-graph graal-graph = ((*diffNodesGraph isabelle-graph graal-graph*) = {})

end

6 Data-flow Semantics

theory *IRTreeEval*
imports
Graph.Stamp
begin

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

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

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

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

As a concrete example, as the *SignedDivNode* 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 = (λ*x*. *UndefVal*)

6.1 Data-flow Tree Representation

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

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*

6.2 Functions for re-calculating stamps

Note: in Java all integer calculations are done as 32 or 64 bit calculations. However, here we generalise the operators to allow any size calculations. Most operators have the same output bits as their inputs. But the following $fixed_32$ binary operators always output 32 bits. And the unary operators that are not $normal_unary$ are narrowing or widening operators, so the result bits is specified by the operator.

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

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* (*b1* ≠ *b2*) *then IllegalStamp else*
(*if op* ∈ *fixed-32*
then unrestricted-stamp (*IntegerStamp 32 lo1 hi1*)
else unrestricted-stamp (*IntegerStamp b1 lo1 hi1*))) |

stamp-binary op - - = IllegalStamp

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

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

6.3 Data-flow Tree Evaluation

fun *unary-eval* :: *IRUnaryOp* ⇒ *Value* ⇒ *Value* **where**
unary-eval UnaryAbs v = intval-abs v |
unary-eval UnaryNeg v = intval-negate v |
unary-eval UnaryNot v = intval-not v |
unary-eval UnaryLogicNegation v = intval-logic-negation v |
unary-eval (*UnaryNarrow inBits outBits*) *v = intval-narrow inBits outBits v* |
unary-eval (*UnarySignExtend inBits outBits*) *v = intval-sign-extend inBits outBits v* |
unary-eval (*UnaryZeroExtend inBits outBits*) *v = intval-zero-extend inBits outBits v*

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:
⟦*valid-value c* (*constantAsStamp 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* ↦ *v*;
v ≠ *UndefVal*⟧
⟹ [*m*,*p*] ⊢ (*ConditionalExpr ce te fe*) ↦ *v* |

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

BinaryExpr:
⟦[*m*,*p*] ⊢ *xe* ↦ *x*;
[*m*,*p*] ⊢ *ye* ↦ *y*;

$result = (bin\text{-}eval\ op\ x\ y);$
$result \neq UndefVal]\!]$
$\Longrightarrow [m,p] \vdash (BinaryExpr\ op\ xe\ ye) \mapsto result\ |$

LeafExpr:
$[\![val = m\ n;$
$valid\text{-}value\ val\ s]\!]$
$\Longrightarrow [m,p] \vdash LeafExpr\ n\ s \mapsto val$

code-pred ($modes$: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool\ as\ evalT$)
[*show-steps*,*show-mode-inference*,*show-intermediate-results*]
evaltree **.**

inductive
$evaltrees :: MapState \Rightarrow Params \Rightarrow IRExpr\ list \Rightarrow Value\ list \Rightarrow bool$ ([-,-] $\vdash$ - $\mapsto_L$ - *55*)
for *m p* **where**

EvalNil:
$[m,p] \vdash [] \mapsto_L []\ |$

EvalCons:
$[\![[m,p] \vdash x \mapsto xval;$
$[m,p] \vdash yy \mapsto_L yyval]\!]$
$\Longrightarrow [m,p] \vdash (x\#yy) \mapsto_L (xval\#yyval)$

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

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

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

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

6.4 Data-flow Tree Refinement

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

definition $equiv\text{-}exprs :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ (- $\doteq$ - *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** $\sqsubseteq$ *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*]:
    $(e_1 < e_2) \longleftrightarrow (e_1 \leq e_2 \land \neg\ (e_1 \doteq e_2))$

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

end

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

end

6.5 Data-flow Tree Theorems

theory *IRTreeEvalThms*
  imports
    Graph.ValueThms
    IRTreeEval
begin

6.5.1 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*

6.5.2 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* $\neq$ *ObjRef v*
by (*cases op*; *cases x*; *auto*)

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

lemma *unary-eval-int*:
assumes *def*: *unary-eval op x* $\neq$ *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* $\neq$ *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*:
($IntVal\ 32\ 1$) = ($new\text{-}int\ 32\ 1$)
unfolding *new-int.simps*
by *auto*

lemma *bin-eval-new-int*:
assumes *def*: $bin\text{-}eval\ op\ x\ y \neq UndefVal$
shows $\exists\, b\ v.\ (bin\text{-}eval\ op\ x\ y) = new\text{-}int\ b\ v \land$
$b = (if\ op \in fixed\text{-}32\ then\ 32\ else\ intval\text{-}bits\ x)$
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 $0 < b \land b \leq 64$
shows *valid-stamp* (*constantAsStamp v*)
proof −
have *bnds*: $fst\ (bit\text{-}bounds\ b) \leq int\text{-}signed\text{-}value\ b\ ival \land int\text{-}signed\text{-}value\ b\ ival$
$\leq snd\ (bit\text{-}bounds\ b)$
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

6.5.3 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*) $\Longrightarrow$
    ($\exists$ *v*. *val* = *IntVal b v*)
  using *valid-value.elims*(*2*) **by** *fastforce*

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

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

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

proof –
  have *valid-value val* (*IntegerStamp b lo hi*)
    using *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 $\exists$ *v*. (*val* = *IntVal b v* $\wedge$
        lo $\leq$ *int-signed-value b v* $\wedge$
        int-signed-value b v $\leq$ *hi*)
  using *assms valid-int*
  by (*metis valid-value.simps*(*1*))

6.5.4 Example Data-flow Optimisations

6.5.5 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 (*smt* (*z3*) *compatible.elims*(*2*) *compatible.simps*(*1*))

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

6.6 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 $([m,p] \vdash ConstantExpr\ c \mapsto v) = (valid\text{-}value\ v\ (constantAsStamp\ c) \land v = c)$
by *blast*

lemma *unfold-binary*:
shows $([m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto val) = (\exists\ x\ y.$
$(([m,p] \vdash xe \mapsto x) \land$
$([m,p] \vdash ye \mapsto y) \land$
$(val = bin\text{-}eval\ op\ x\ y) \land$
$(val \neq UndefVal)$
$))$ (**is** *?L = ?R*)
proof (*intro iffI*)
assume *3*: *?L*
show *?R* **by** (*rule evaltree.cases*[*OF 3*]; *blast+*)
next
assume *?R*
then obtain *x y* **where** $[m,p] \vdash xe \mapsto x$
and $[m,p] \vdash ye \mapsto y$
and $val = bin\text{-}eval\ op\ x\ y$
and $val \neq UndefVal$
by *auto*
then show *?L*
by (*rule BinaryExpr*)
qed

lemma *unfold-unary*:
shows $([m,p] \vdash UnaryExpr\ op\ xe \mapsto val)$
$= (\exists\ x.$
$(([m,p] \vdash xe \mapsto x) \land$
$(val = unary\text{-}eval\ op\ x) \land$
$(val \neq UndefVal)$

)) (**is** *?L* = *?R*)
by *auto*

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

6.7 Lemmas about *new_int* and integer eval results.

lemma *unary-eval-new-int*:
assumes *def*: *unary-eval op x* $\neq$ *UndefVal*
shows $\exists$ *b v*. *unary-eval op x* = *new-int b v* $\wedge$
b = (*if op* $\in$ *normal-unary then intval-bits x else ir-resultBits op*)
proof (*cases op* $\in$ *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*:
$[m,p] \vdash xe \mapsto (IntVal\ b\ ix) \Longrightarrow take\text{-}bit\ b\ ix = ix$
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*
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 intval-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* ∈ *fixed-32 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*
   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

end

7 Tree to Graph

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

7.1 Subgraph to Data-flow Tree

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

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

```
fun is-preevaluated :: IRNode ⇒ bool where
  is-preevaluated (InvokeNode n - - - - -) = True |
  is-preevaluated (InvokeWithExceptionNode n - - - - - -) = True |
  is-preevaluated (NewInstanceNode n - - -) = True |
  is-preevaluated (LoadFieldNode n - - -) = True |
  is-preevaluated (SignedDivNode n - - - - -) = True |
  is-preevaluated (SignedRemNode n - - - - -) = True |
  is-preevaluated (ValuePhiNode n - -) = True |
  is-preevaluated - = 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⟧
    ⟹ g ⊢ n ≃ (ParameterExpr i s) |

  ConditionalNode:
  ⟦kind g n = ConditionalNode c t f;
    g ⊢ c ≃ ce;
    g ⊢ t ≃ te;
    g ⊢ f ≃ fe⟧
    ⟹ g ⊢ n ≃ (ConditionalExpr ce te fe) |


  AbsNode:
  ⟦kind g n = AbsNode x;
    g ⊢ x ≃ xe⟧
    ⟹ g ⊢ n ≃ (UnaryExpr UnaryAbs xe) |

  NotNode:
  ⟦kind g n = NotNode x;
    g ⊢ x ≃ xe⟧
    ⟹ g ⊢ n ≃ (UnaryExpr UnaryNot xe) |

  NegateNode:
  ⟦kind g n = NegateNode x;
    g ⊢ x ≃ xe⟧
    ⟹ g ⊢ n ≃ (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:

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

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

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

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

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

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

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

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

$ZeroExtendNode$:
$[\![kind\ g\ n = ZeroExtendNode\ inputBits\ resultBits\ x;$
$\ \ g \vdash x \simeq xe]\!]$

$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnaryZeroExtend\ inputBits\ resultBits)\ xe) \mid$

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

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

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:
$g \vdash [] \simeq_L [] \mid$

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

7.2 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* ⇒ *ID* ⇒ *ID* ⇒ *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* ⇒ *ID* ⇒ *bool* **where**
n ∉ *ids g* ⟹ *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* $n]\!]$
$\Longrightarrow g \oplus$ (*ParameterExpr* i s) $\leadsto (g,\ n)$ |

ParameterNodeNew:
$[\![$*find-node-and-stamp* g (*ParameterNode* i, s) $=$ *None*;
$n =$ *get-fresh-id* g;
$g' =$ *add-node* n (*ParameterNode* i, s) $g]\!]$
$\Longrightarrow g \oplus$ (*ParameterExpr* i s) $\leadsto (g',\ n)$ |

ConditionalNodeSame:
$[\![g \oplus ce \leadsto (g2,\ c)$;
$g2 \oplus te \leadsto (g3,\ t)$;
$g3 \oplus fe \leadsto (g4,\ f)$;
$s' =$ *meet* (*stamp* $g4$ t) (*stamp* $g4$ f);
find-node-and-stamp $g4$ (*ConditionalNode* c t f, s') $=$ *Some* $n]\!]$
$\Longrightarrow g \oplus$ (*ConditionalExpr* ce te fe) $\leadsto (g4,\ n)$ |

ConditionalNodeNew:
$[\![g \oplus ce \leadsto (g2,\ c)$;
$g2 \oplus te \leadsto (g3,\ t)$;
$g3 \oplus fe \leadsto (g4,\ f)$;
$s' =$ *meet* (*stamp* $g4$ t) (*stamp* $g4$ f);
find-node-and-stamp $g4$ (*ConditionalNode* c t f, s') $=$ *None*;
$n =$ *get-fresh-id* $g4$;
$g' =$ *add-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-unary* op (*stamp* $g2$ x);
find-node-and-stamp $g2$ (*unary-node* op x, s') $=$ *Some* $n]\!]$
$\Longrightarrow g \oplus$ (*UnaryExpr* op xe) $\leadsto (g2,\ n)$ |

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

BinaryNodeNew:

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

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

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

unrepRules

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

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

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

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

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

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

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

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

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

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

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

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

7.3 Lift Data-flow Tree Semantics

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

7.4 Graph Refinement

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

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

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

7.5 Maximal Sharing

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

end

7.6 Formedness Properties

theory *Form*
imports
Semantics.TreeToGraph
begin

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

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

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* .

$(\forall\ v\ m\ p\ .\ ([g,\ m,\ p] \vdash n \mapsto v) \longrightarrow$
$(\textit{is-LogicNode}\ (\textit{kind}\ g\ n) \longrightarrow$
$\textit{wf-bool}\ v \wedge \textit{wf-logic-node-inputs}\ g\ n)))$

end

7.7 Dynamic Frames

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

theory *IRGraphFrames*
imports
Form
begin

fun *unchanged* :: *ID set* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
$\textit{unchanged ns g1 g2} = (\forall\ n\ .\ n \in ns \longrightarrow$
$(n \in \textit{ids g1} \wedge n \in \textit{ids g2} \wedge \textit{kind g1 n} = \textit{kind g2 n} \wedge \textit{stamp g1 n} = \textit{stamp g2 n}))$

fun *changeonly* :: *ID set* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
$\textit{changeonly ns g1 g2} = (\forall\ n\ .\ n \in \textit{ids g1} \wedge n \notin ns \longrightarrow$
$(n \in \textit{ids g1} \wedge n \in \textit{ids g2} \wedge \textit{kind g1 n} = \textit{kind g2 n} \wedge \textit{stamp g1 n} = \textit{stamp g2 n}))$

lemma *node-unchanged*:
assumes *unchanged ns g1 g2*
assumes $\textit{nid} \in \textit{ns}$
shows $\textit{kind g1 nid} = \textit{kind g2 nid}$
using *assms* **by** *auto*

lemma *other-node-unchanged*:
assumes *changeonly ns g1 g2*
assumes $\textit{nid} \in \textit{ids g1}$
assumes $\textit{nid} \notin \textit{ns}$
shows $\textit{kind g1 nid} = \textit{kind g2 nid}$
using *assms*
using *changeonly.simps* **by** *blast*

Some notation for input nodes used

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

use0: $\textit{nid} \in \textit{ids g}$
$\Longrightarrow \textit{eval-uses g nid nid}\ |$

use-inp: $nid' \in inputs\ g\ n$
  $\Longrightarrow$ *eval-uses g nid nid'* |

use-trans: $[\![$*eval-uses g nid nid'*;
  eval-uses g nid' nid''$]\!]$
  $\Longrightarrow$ *eval-uses g nid nid''*

fun *eval-usages* :: *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *ID set* **where**
  eval-usages g nid = $\{n \in ids\ g\ .\ eval\text{-}uses\ g\ nid\ n\}$

lemma *eval-usages-self*:
  assumes $nid \in ids\ g$
  shows $nid \in eval\text{-}usages\ g\ nid$
  using *assms eval-usages.simps eval-uses.intros*(*1*)
  by (*simp add*: *ids.rep-eq*)

lemma *not-in-g-inputs*:
  assumes $nid \notin ids\ g$
  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 $n \neq NoNode$
  assumes *List.member* (*inputs-of n*) *child*
  shows $child \in inputs\ g\ nid$
  unfolding *inputs.simps* **using** *assms*
  by (*metis in-set-member*)

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

lemma *inp-in-g*:
  assumes $n \in inputs\ g\ nid$
  shows $nid \in ids\ g$
proof –
  have *inputs g nid* $\neq$ {}
    using *assms*
    by (*metis empty-iff empty-set*)

then have *kind g nid* $\neq$ *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* $\in$ *inputs g nid*
shows *n* $\in$ *ids g*
using *assms* **unfolding** *wf-folds*
using *inp-in-g* **by** *blast*

lemma *kind-unchanged*:
assumes *nid* $\in$ *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* $\in$ *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* $\in$ *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'* $\in$ *ids g*
shows *eval-uses g nid nid'* $\longleftrightarrow$ *nid'* $\in$ *us* (**is** *?P* $\longleftrightarrow$ *?Q*)
using *assms eval-usages.simps*
by (*simp add*: *ids.rep-eq*)

lemma *inputs-are-uses*:
assumes *nid'* $\in$ *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* $\vdash$ *nid* $\simeq$ *e*
  assumes *wf*: *wf-graph g1*
  shows *g2* $\vdash$ *nid* $\simeq$ *e*
proof −
  have *dom*: *nid* $\in$ *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* $\in$ *eval-usages g1 n* $\wedge$ *t* $\in$ *eval-usages g1 n* $\wedge$ *f* $\in$ *eval-usages g1 n*
    using *inputs-of-ConditionalNode*
      by (*metis ConditionalNode.hyps*(*1*) *ConditionalNode.hyps*(*2*) *ConditionalNode.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* $\in$ *eval-usages g1 n*
    using *inputs-of-AbsNode*
      by (*metis AbsNode.hyps*(*1*) *AbsNode.hyps*(*2*) *encode-in-ids inputs.simps inputs-are-usages list.set-intros*(*1*))
  then show *?case*
    by (*metis AbsNode.IH AbsNode.hyps*(*1*) *AbsNode.prems*(*1*) *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 $x \in$ *eval-usages g1 n*
using *inputs-of-NotNode*
by (*metis NotNode.hyps*(*1*) *NotNode.hyps*(*2*) *encode-in-ids inputs.simps inputs-are-usages list.set-intros*(*1*))
then show *?case*
by (*metis NotNode.IH NotNode.hyps*(*1*) *NotNode.prems*(*1*) *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 $x \in$ *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 $x \in$ *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 LogicNegationNode.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 $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *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 $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *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 $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *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 $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *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 $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *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*) *ShortCircuitOrNode.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 $x \in$ *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 $x \in$ *eval-usages g1 n*
using *inputs-of-SignExtendNode inputs-of-are-usages*
by (*metis SignExtendNode.hyps*(*1*) *SignExtendNode.hyps*(*2*) *encode-in-ids inputs.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 $x \in$ *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 $n' \in$ *eval-usages g1 n*
by (*metis IRNodes.inputs-of-RefNode RefNode.hyps*(*1*) *RefNode.hyps*(*2*) *encode-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 inputs.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* $\notin$ *ids g*
assumes *nid* $\in$ *ids g*
assumes *gup* = *add-node new k g*
assumes *wf-graph g*
shows *unchanged* (*eval-usages g nid*) *g gup*
proof −
have *new* $\notin$ (*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'* $\in$ *ids g* $\wedge$ *nid* = *nid'*)
$\vee$ *nid'* $\in$ *inputs g nid*
$\vee$ ($\exists$ *nid''* . *eval-uses g nid nid''* $\wedge$ *eval-uses g nid'' nid'*))
$\longleftrightarrow$ *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* $\in$ *ids g*
assumes *eval-uses g nid nid'*
shows *nid'* $\in$ *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
```

7.8 Tree to Graph Theorems

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

7.8.1 Extraction and Evaluation of Expression Trees is Deterministic.

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

named-theorems *rep*

lemma *rep-constant* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = ConstantNode\ c \Longrightarrow$
$e = ConstantExpr\ c$
by (*induction rule*: *rep.induct*; *auto*)

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

lemma *rep-conditional* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = ConditionalNode\ c\ t\ f \Longrightarrow$
$(\exists\ ce\ te\ fe.\ e = ConditionalExpr\ ce\ te\ fe)$
by (*induction rule*: *rep.induct*; *auto*)

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

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

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

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

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

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

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

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

lemma *rep-right-shift* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$\mathit{kind}\ g\ n = \mathit{RightShiftNode}\ x\ y \Longrightarrow$
$(\exists\, xe\ ye.\ e = \mathit{BinaryExpr}\ \mathit{BinRightShift}\ xe\ ye)$
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*]:
  $g \vdash n \simeq e \Longrightarrow$
   kind g n = *RefNode* $n' \Longrightarrow$
   $g \vdash n' \simeq e$
  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 $(g \vdash n \simeq e_1) \Longrightarrow (g \vdash n \simeq e_2) \Longrightarrow e_1 = e_2$
proof (*induction arbitrary*: e_2 *rule*: *rep.induct*)
  case (*ConstantNode n c*)
  then show *?case* **using** *rep-constant* **by** *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*:
  $g \vdash xs \simeq_L e1 \Longrightarrow$
   $g \vdash xs \simeq_L e2 \Longrightarrow$
   $e1 = e2$
proof (*induction arbitrary*: *e2 rule*: *replist.induct*)
  case *RepNil*
  then show *?case*

using *replist.cases* **by** *auto*
next
case (*RepCons x xe xs xse*)
then show *?case*
by (*metis list.distinct*(*1*) *list.sel*(*1*) *list.sel*(*3*) *repDet replist.cases*)
qed

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

lemma *graphDet*: $([g,m,p] \vdash n \mapsto v_1) \land ([g,m,p] \vdash n \mapsto v_2) \Longrightarrow v_1 = v_2$
using *encodeEvalDet* **by** *blast*

7.8.2 Monotonicity of Graph Refinement

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

lemma *mono-abs*:
assumes *kind g1 n = AbsNode x* $\land$ *kind g2 n = AbsNode x*
assumes $(g1 \vdash x \simeq xe1) \land (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \land (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
by (*metis AbsNode assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *mono-unary repDet*)

lemma *mono-not*:
assumes *kind g1 n = NotNode x* $\land$ *kind g2 n = NotNode x*
assumes $(g1 \vdash x \simeq xe1) \land (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \land (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
by (*metis NotNode assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *mono-unary repDet*)

lemma *mono-negate*:
assumes *kind g1 n = NegateNode x* $\land$ *kind g2 n = NegateNode x*
assumes $(g1 \vdash x \simeq xe1) \land (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \land (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
by (*metis NegateNode assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *mono-unary repDet*)

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

shows $e1 \geq e2$
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 ($g1 \vdash x \simeq xe1$) ∧ ($g2 \vdash x \simeq xe2$)
assumes $xe1 \geq xe2$
assumes ($g1 \vdash n \simeq e1$) ∧ ($g2 \vdash n \simeq e2$)
shows $e1 \geq e2$
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 ($g1 \vdash x \simeq xe1$) ∧ ($g2 \vdash x \simeq xe2$)
assumes $xe1 \geq xe2$
assumes ($g1 \vdash n \simeq e1$) ∧ ($g2 \vdash n \simeq e2$)
shows $e1 \geq e2$
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 ($g1 \vdash x \simeq xe1$) ∧ ($g2 \vdash x \simeq xe2$)
assumes $xe1 \geq xe2$
assumes ($g1 \vdash n \simeq e1$) ∧ ($g2 \vdash n \simeq e2$)
shows $e1 \geq e2$
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 ($g1 \vdash c \simeq ce1$) ∧ ($g2 \vdash c \simeq ce2$)
assumes ($g1 \vdash t \simeq te1$) ∧ ($g2 \vdash t \simeq te2$)
assumes ($g1 \vdash f \simeq fe1$) ∧ ($g2 \vdash f \simeq fe2$)
assumes $ce1 \geq ce2 \wedge te1 \geq te2 \wedge fe1 \geq fe2$
assumes ($g1 \vdash n \simeq e1$) ∧ ($g2 \vdash n \simeq e2$)
shows $e1 \geq e2$
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 ($g1 \vdash x \simeq xe1$) ∧ ($g2 \vdash x \simeq xe2$)
assumes ($g1 \vdash y \simeq ye1$) ∧ ($g2 \vdash y \simeq ye2$)

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 AddNodeE IRNode.inject*(*2*) *repDet rep-add*
by (*metis IRNode.distinct*(*205*))

lemma *mono-mul*:
assumes *kind g1 n* = *MulNode x y* ∧ *kind g2 n* = *MulNode x y*
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$
assumes $xe1 \geq xe2 \wedge ye1 \geq ye2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
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 ≠ *NoNode*
apply (*induction rule*: *rep.induct*)
apply (*match IRNode.distinct* **in** *e*: *?n* ≠ *NoNode* ⇒
‹*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* ⊴ *as-set g1*) ⊆ *as-set g2*
shows *kind g1 n* = *kind g2 n* ∧ *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* ⇒ *IRNode* =
(*match IRNode.inject* **in** *i*: (*node* - = *node* -) = - ⇒
‹*metis i*›)
method *metis-node-eq-binary* **for** *node* :: *'a* ⇒ *'a* ⇒ *IRNode* =
(*match IRNode.inject* **in** *i*: (*node* - - = *node* - -) = - ⇒

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

7.8.3 Lift Data-flow Tree Refinement to Graph Refinement

theorem *graph-semantics-preservation*:
assumes *a*: $e1' \geq e2'$
assumes *b*: $(\{n'\} \trianglelefteq$ *as-set g1*$) \subseteq$ *as-set g2*
assumes *c*: $g1 \vdash n' \simeq e1'$
assumes *d*: $g2 \vdash n' \simeq e2'$
shows *graph-refinement g1 g2*
unfolding *graph-refinement-def* **apply** *rule*
apply (*metis b d ids-some no-encoding not-excluded-keep-type singleton-iff subsetI*)
apply (*rule allI*) **apply** (*rule impI*) **apply** (*rule allI*) **apply** (*rule impI*)
unfolding *graph-represents-expression-def*
proof −
fix *n e1*
assume *e*: $n \in$ *ids g1*
assume *f*: $(g1 \vdash n \simeq e1)$

show $\exists\ e2.\ (g2 \vdash n \simeq e2) \wedge e1 \geq e2$
proof (*cases* $n = n'$)
case *True*
have *g*: $e1 = e1'$ **using** *c f True repDet* **by** *simp*
have *h*: $(g2 \vdash n \simeq e2') \wedge e1' \geq e2'$
using *True a d* **by** *blast*
then show *?thesis*
using *g* **by** *blast*
next
case *False*
have $n \notin \{n'\}$
using *False* **by** *simp*
then have *i*: *kind g1 n* = *kind g2 n* $\wedge$ *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*: $g1 \vdash n \simeq$ *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*: $g1 \vdash cn \simeq ce1$
using *ConditionalNode.hyps*(*1*) *ConditionalNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *mt*: $g1 \vdash tn \simeq te1$
using *ConditionalNode.hyps*(*1*) *ConditionalNode.hyps*(*3*) **by** *fastforce*
from *l* **have** *mf*: $g1 \vdash fn \simeq fe1$
using *ConditionalNode.hyps*(*1*) *ConditionalNode.hyps*(*4*) **by** *fastforce*
then show *?case*
proof −
have $g1 \vdash cn \simeq ce1$ **using** *mc* **by** *simp*
have $g1 \vdash tn \simeq te1$ **using** *mt* **by** *simp*
have $g1 \vdash fn \simeq fe1$ **using** *mf* **by** *simp*
have *cer*: $\exists$ *ce2*. ($g2 \vdash cn \simeq ce2$) $\wedge$ *ce1* $\geq$ *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*: $\exists$ *te2*. ($g2 \vdash tn \simeq te2$) $\wedge$ *te1* $\geq$ *te2*
using *ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-ternary ConditionalNode*)
have $\exists$ *fe2*. ($g2 \vdash fn \simeq fe2$) $\wedge$ *fe1* $\geq$ *fe2*
using *ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-ternary ConditionalNode*)
then have $\exists$ *ce2 te2 fe2*. ($g2 \vdash n \simeq$ *ConditionalExpr ce2 te2 fe2*) $\wedge$ *ConditionalExpr ce1 te1 fe1* $\geq$ *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*: $g1 \vdash n \simeq$ *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*: $g1 \vdash xn \simeq xe1$
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*: $g2 \vdash n \simeq$ *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'* $\geq$ *UnaryExpr UnaryAbs e2'*
by (*meson a mono-unary*)
then show *?thesis* **using** *ev r*
by (*metis n*)
next
case *False*
have *g1* $\vdash$ *xn* $\simeq$ *xe1* **using** *m* **by** *simp*
have $\exists$ *xe2*. (*g2* $\vdash$ *xn* $\simeq$ *xe2*) $\wedge$ *xe1* $\geq$ *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 $\exists$ *xe2*. (*g2* $\vdash$ *n* $\simeq$ *UnaryExpr UnaryAbs xe2*) $\wedge$ *UnaryExpr UnaryAbs xe1* $\geq$ *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* $\vdash$ *n* $\simeq$ *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* $\vdash$ *xn* $\simeq$ *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* $\vdash$ *n* $\simeq$ *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'* $\geq$ *UnaryExpr UnaryNot e2'*
by (*meson a mono-unary*)
then show *?thesis* **using** *ev r*
by (*metis n*)
next
case *False*
have *g1* $\vdash$ *xn* $\simeq$ *xe1* **using** *m* **by** *simp*
have $\exists$ *xe2*. (*g2* $\vdash$ *xn* $\simeq$ *xe2*) $\wedge$ *xe1* $\geq$ *xe2*
using *NotNode*
using *False i b l not-excluded-keep-type singletonD no-encoding*
by (*metis-node-eq-unary NotNode*)
then have $\exists$ *xe2*. (*g2* $\vdash$ *n* $\simeq$ *UnaryExpr UnaryNot xe2*) $\wedge$ *UnaryExpr UnaryNot xe1* $\geq$ *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*: $g1 \vdash n \simeq$ *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*: $g1 \vdash xn \simeq xe1$
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*: $g2 \vdash n \simeq$ *UnaryExpr UnaryNeg* $e2'$ **using** *NegateNode.hyps*(*1*) *l m n*
using *NegateNode.prems True d rep.NegateNode* **by** *simp*
then have *r*: *UnaryExpr UnaryNeg* $e1' \geq$ *UnaryExpr UnaryNeg* $e2'$
by (*meson a mono-unary*)
then show *?thesis* **using** *ev r*
by (*metis n*)
next
case *False*
have $g1 \vdash xn \simeq xe1$ **using** *m* **by** *simp*
have $\exists$ *xe2*. ($g2 \vdash xn \simeq xe2$) $\wedge$ $xe1 \geq xe2$
using *NegateNode*
using *False i b l not-excluded-keep-type singletonD no-encoding*
by (*metis-node-eq-unary NegateNode*)
then have $\exists$ *xe2*. ($g2 \vdash n \simeq$ *UnaryExpr UnaryNeg xe2*) $\wedge$ *UnaryExpr UnaryNeg xe1* $\geq$ *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*: $g1 \vdash n \simeq$ *UnaryExpr UnaryLogicNegation xe1* **using** *f LogicNegationNode*
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*: $g1 \vdash xn \simeq xe1$
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*: $g2 \vdash n \simeq$ *UnaryExpr UnaryLogicNegation* $e2'$ **using** *LogicNegationNode.hyps*(*1*) *l m n*
using *LogicNegationNode.prems True d rep.LogicNegationNode* **by** *simp*
then have *r*: *UnaryExpr UnaryLogicNegation* $e1' \geq$ *UnaryExpr UnaryLog-*

icNegation e2'
    by (*meson a mono-unary*)
  then show *?thesis* **using** *ev r*
    by (*metis n*)
next
  case *False*
  have $g1 \vdash xn \simeq xe1$ **using** *m* **by** *simp*
  have $\exists$ *xe2*. $(g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$
    using *LogicNegationNode*
    using *False i b l not-excluded-keep-type singletonD no-encoding*
    by (*metis-node-eq-unary LogicNegationNode*)
  then have $\exists$ *xe2*. $(g2 \vdash n \simeq$ *UnaryExpr UnaryLogicNegation xe2*$) \wedge$ *UnaryExpr UnaryLogicNegation xe1* $\geq$ *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*: $g1 \vdash n \simeq$ *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*: $g1 \vdash xn \simeq xe1$
    using *AddNode.hyps*(*1*) *AddNode.hyps*(*2*) **by** *fastforce*
  from *l* **have** *my*: $g1 \vdash yn \simeq ye1$
    using *AddNode.hyps*(*1*) *AddNode.hyps*(*3*) **by** *fastforce*
  then show *?case*
  proof –
    have $g1 \vdash xn \simeq xe1$ **using** *mx* **by** *simp*
    have $g1 \vdash yn \simeq ye1$ **using** *my* **by** *simp*
    have *xer*: $\exists$ *xe2*. $(g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$
      using *AddNode*
      using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
      by (*metis-node-eq-binary AddNode*)
    have $\exists$ *ye2*. $(g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$
      using *AddNode*
      using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
      by (*metis-node-eq-binary AddNode*)
    then have $\exists$ *xe2 ye2*. $(g2 \vdash n \simeq$ *BinaryExpr BinAdd xe2 ye2*$) \wedge$ *BinaryExpr BinAdd xe1 ye1* $\geq$ *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*: $g1 \vdash n \simeq$ *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*: $g1 \vdash n \simeq$ *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*: $g1 \vdash xn \simeq xe1$
using *AndNode.hyps*(*1*) *AndNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: $g1 \vdash yn \simeq ye1$
using *AndNode.hyps*(*1*) *AndNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof −
have $g1 \vdash xn \simeq xe1$ **using** *mx* **by** *simp*
have $g1 \vdash yn \simeq ye1$ **using** *my* **by** *simp*
have *xer*: $\exists$ *xe2*. ($g2 \vdash xn \simeq xe2$) $\wedge$ *xe1* $\geq$ *xe2*
using *AndNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary AndNode*)
have $\exists$ *ye2*. ($g2 \vdash yn \simeq ye2$) $\wedge$ *ye1* $\geq$ *ye2*
using *AndNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary AndNode*)
then have $\exists$ *xe2 ye2*. ($g2 \vdash n \simeq$ *BinaryExpr BinAnd xe2 ye2*) $\wedge$ *BinaryExpr BinAnd xe1 ye1* $\geq$ *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*: $g1 \vdash n \simeq$ *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*: $g1 \vdash xn \simeq xe1$
using *OrNode.hyps*(*1*) *OrNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: $g1 \vdash yn \simeq ye1$
using *OrNode.hyps*(*1*) *OrNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof −
have $g1 \vdash xn \simeq xe1$ **using** *mx* **by** *simp*
have $g1 \vdash yn \simeq ye1$ **using** *my* **by** *simp*
have *xer*: $\exists$ *xe2*. ($g2 \vdash xn \simeq xe2$) $\wedge$ *xe1* $\geq$ *xe2*
using *OrNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary OrNode*)
have $\exists$ *ye2*. ($g2 \vdash yn \simeq ye2$) $\wedge$ *ye1* $\geq$ *ye2*

using *OrNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary OrNode*)
then have $\exists$ *xe2 ye2*. (*g2* $\vdash$ *n* $\simeq$ *BinaryExpr BinOr xe2 ye2*) $\wedge$ *BinaryExpr BinOr xe1 ye1* $\geq$ *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* $\vdash$ *n* $\simeq$ *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* $\vdash$ *xn* $\simeq$ *xe1*
using *XorNode.hyps*(*1*) *XorNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* $\vdash$ *yn* $\simeq$ *ye1*
using *XorNode.hyps*(*1*) *XorNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof –
have *g1* $\vdash$ *xn* $\simeq$ *xe1* **using** *mx* **by** *simp*
have *g1* $\vdash$ *yn* $\simeq$ *ye1* **using** *my* **by** *simp*
have *xer*: $\exists$ *xe2*. (*g2* $\vdash$ *xn* $\simeq$ *xe2*) $\wedge$ *xe1* $\geq$ *xe2*
using *XorNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary XorNode*)
have $\exists$ *ye2*. (*g2* $\vdash$ *yn* $\simeq$ *ye2*) $\wedge$ *ye1* $\geq$ *ye2*
using *XorNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary XorNode*)
then have $\exists$ *xe2 ye2*. (*g2* $\vdash$ *n* $\simeq$ *BinaryExpr BinXor xe2 ye2*) $\wedge$ *BinaryExpr BinXor xe1 ye1* $\geq$ *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* $\vdash$ *n* $\simeq$ *BinaryExpr BinShortCircuitOr xe1 ye1* **using** *f ShortCircuitOrNode*
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* $\vdash$ *xn* $\simeq$ *xe1*
using *ShortCircuitOrNode.hyps*(*1*) *ShortCircuitOrNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* $\vdash$ *yn* $\simeq$ *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 UnsignedRightShiftNode*
by (*simp add*: *UnsignedRightShiftNode.hyps*(*2*) *rep.UnsignedRightShiftNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *UnsignedRightShiftNode xn yn*
using *UnsignedRightShiftNode.hyps*(*1*) **by** *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 IntegerBelowNode*
by (*simp add*: *IntegerBelowNode.hyps*(*2*) *rep.IntegerBelowNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *IntegerBelowNode xn yn*
using *IntegerBelowNode.hyps*(*1*) **by** *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 IntegerEqualsNode*
by (*simp add*: *IntegerEqualsNode.hyps*(*2*) *rep.IntegerEqualsNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *IntegerEqualsNode xn yn*
using *IntegerEqualsNode.hyps*(*1*) **by** *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 resultBits*) *xe2*) ∧ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) *xe1* ≥ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) *xe2*
by (*metis NarrowNode.prems l mono-unary rep.NarrowNode*)
then show *?thesis*
by *meson*
qed
next
case (*SignExtendNode n inputBits resultBits x xe1*)
have *k*: *g1* ⊢ *n* ≃ *UnaryExpr* (*UnarySignExtend inputBits resultBits*) *xe1* **using** *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 $g1 \vdash xn \simeq xe1$ **using** *m* **by** *simp*
have $\exists\ xe2.\ (g2 \vdash xn \simeq xe2) \land xe1 \geq xe2$
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 $\exists\ xe2.\ (g2 \vdash n \simeq$ *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *xe2*) $\land$ *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *xe1* $\geq$ *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*: $(\{n\} \trianglelefteq as\text{-}set\ g_1) \subseteq as\text{-}set\ g_2$
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*:
$e_1 \geq e_2$
$\land (g_1 \vdash n \simeq e_1) \land$ *maximal-sharing* g_1
$\land (\{n\} \trianglelefteq as\text{-}set\ g_1) \subseteq as\text{-}set\ g_2$
$\land (g_2 \vdash n \simeq e_2) \land$ *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* $\geq$ *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*)

7.8.4 Term Graph Reconstruction

lemma *find-exists-kind*:
assumes *find-node-and-stamp g* (*node*, *s*) = *Some nid*
shows *kind g nid* = *node*
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 $g' = add\text{-}node\ nid\ (node,\ s)\ g$
assumes $node \neq NoNode$
shows $kind\ g'\ nid = node$
using *assms*
using *add-node-lookup* **by** *presburger*

lemma *find-new-stamp*:
assumes $g' = add\text{-}node\ nid\ (node,\ s)\ g$
assumes $node \neq NoNode$
shows $stamp\ g'\ nid = s$
using *assms*
using *add-node-lookup* **by** *presburger*

lemma *sorted-bottom*:
assumes *finite xs*
assumes $x \in xs$
shows $x \leq last(sorted\text{-}list\text{-}of\text{-}set(xs::nat\ set))$
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*: $finite\ xs \Longrightarrow last(sorted\text{-}list\text{-}of\text{-}set(xs::nat\ set)) + 1 \notin xs$
using *sorted-bottom*
using *not-le* **by** *auto*

lemma *fresh-ids*:
assumes $n = get\text{-}fresh\text{-}id\ g$
shows $n \notin ids\ g$
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 $\forall\, n \in ids\ g.\ kind\ g\ n = kind\ g'\ n$
assumes $\forall\, n \in ids\ g.\ stamp\ g\ n = stamp\ g'\ n$
shows $(g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
apply (*rule impI*) **subgoal premises** *e* **using** *e assms*
apply (*induction n e*)
apply (*metis no-encoding rep.ConstantNode*)
apply (*metis no-encoding rep.ParameterNode*)
apply (*metis no-encoding rep.ConditionalNode*)
apply (*metis no-encoding rep.AbsNode*)
apply (*metis no-encoding rep.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 $n \notin ids\ g$
assumes $g' = add\text{-}node\ n\ (k, s)\ g$
shows $as\text{-}set\ g \subseteq as\text{-}set\ g'$
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 $(g \oplus e \rightsquigarrow (g', n))$
shows $as\text{-}set\ g \subseteq as\text{-}set\ g'$
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 $g \oplus e \leadsto (g', n)$
shows $\textit{ids } g \subseteq \textit{ids } g'$
using *assms unrep-subset*
by (*meson graph-refinement-def subset-refines*)

lemma *unrep-unchanged*:
assumes $g \oplus e \leadsto (g', n)$
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*:

$g \oplus e \rightsquigarrow (g', n) \Longrightarrow (g' \vdash n \simeq e) \wedge \textit{as-set } g \subseteq \textit{as-set } g'$
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 $g' \vdash x \simeq xe$ **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 $x \in$ *ids g2* **using** *local.UnaryNodeNew*
using *eval-contains-id* **by** *blast*
then have $x \neq n$ **using** *local.UnaryNodeNew*(*5*) *fresh-ids* **by** *blast*
have $g' \vdash x \simeq xe$ **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*: $g3 \vdash x \simeq xe$ **using** *local.BinaryNodeSame unrep-unchanged*
using *no-encoding* **by** *blast*
have *y*: $g3 \vdash y \simeq ye$ **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* $\neq$ *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*: $g' \vdash x \simeq xe$ **using** *local.BinaryNodeNew unrep-unchanged*
using *no-encoding*
by (*meson fresh-node-preserves-other-nodes*)
have *y*: $g' \vdash y \simeq ye$ **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 $g \vdash n \simeq e_1$
assumes *kind g n′* = *RefNode n*
shows $g \vdash n' \trianglelefteq e_1$
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 $(g_2 \vdash n' \trianglelefteq e_1) \wedge graph\text{-}refinement\ g_1\ g_2$
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*: $set\ (sorted\text{-}list\text{-}of\text{-}set\ (ids\ g)) = ids\ g$
using *Rep-IRGraph set-sorted-list-of-set ids-finite*
by *blast*

lemma *find-none*:
assumes $find\text{-}node\text{-}and\text{-}stamp\ g\ (k, s) = None$
shows $\forall\ n \in ids\ g.\ kind\ g\ n \neq k \vee stamp\ g\ n \neq s$
proof –
have $(\nexists n.\ n \in ids\ g \wedge (kind\ g\ n = k \wedge stamp\ g\ n = s))$
using *assms* **unfolding** *find-node-and-stamp.simps* **using** *find-None-iff unwrap-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*)

7.8.5 Data-flow Tree to Subgraph Preserves Maximal Sharing

lemma *same-kind-stamp-encodes-equal*:
assumes $kind\ g\ n = kind\ g\ n'$
assumes $stamp\ g\ n = stamp\ g\ n'$
assumes $\neg(is\text{-}preevaluated\ (kind\ g\ n))$
shows $\forall\ e.\ (g \vdash n \simeq e) \longrightarrow (g \vdash n' \simeq e)$
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** *presburger*
then have *isin*: $n \in$ *ids g*
by *simp*
have *inputs*: $\{c, t, f\}$ = *inputs g n*
using *kind* **unfolding** *inputs.simps* **using** *inputs-of-ConditionalNode* **by** *simp*
have $c \in$ *ids g* $\wedge$ $t \in$ *ids g* $\wedge$ $f \in$ *ids g*
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $c \notin$ *new* $\wedge$ $t \notin$ *new* $\wedge$ $f \notin$ *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*: $n \in$ *ids g*
by *simp*
have *inputs*: $\{x\}$ = *inputs g n*
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in$ *ids g*
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin$ *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*: $n \in$ *ids g*
by *simp*
have *inputs*: $\{x\}$ = *inputs g n*
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in$ *ids g*
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin$ *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*: $n \in$ *ids g*
by *simp*
have *inputs*: $\{x\}$ = *inputs g n*
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in$ *ids g*
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin$ *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*: $n \in$ *ids g*
by *simp*
have *inputs*: $\{x\}$ = *inputs g n*
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in$ *ids g*
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin$ *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*: $n \in$ *ids g*
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in$ *ids g* $\wedge$ $y \in$ *ids g*
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin$ *new* $\wedge$ $y \notin$ *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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\} = inputs\ g\ n$
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g \wedge y \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\} = inputs\ g\ n$
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g \wedge y \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\} = inputs\ g\ n$
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g \wedge y \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
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*: $n \in ids\ g$
by *simp*

have *inputs*: $\{x, y\} = inputs\ g\ n$
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g \wedge y \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\} = inputs\ g\ n$
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g \wedge y \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\} = inputs\ g\ n$
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g \wedge y \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\} = inputs\ g\ n$
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g \wedge y \in ids\ g$
using *closed* **unfolding** *wf-closed-def*

using *isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g \wedge y \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g \wedge y \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g \wedge y \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x\} = inputs\ g\ n$
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin new$
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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x\} = inputs\ g\ n$
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $x \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $x \notin new$
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*: $n \in ids\ g$
by *simp*
have *inputs*: $\{n'\} = inputs\ g\ n$
using *kind* **unfolding** *inputs.simps* **by** *simp*
have $n' \in ids\ g$
using *closed* **unfolding** *wf-closed-def*
using *isin inputs* **by** *blast*
then have $n' \notin new$
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 $g \oplus e \leadsto (g', n)$
  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 $g \oplus e \leadsto (g', n')$
  shows $n \in ids\ g'$
  using *assms*
  by (*meson subsetD unrep-ids-subset*)

lemma *unrep-preserves-closure*:
  assumes *wf-closed g*
  assumes $g \oplus e \leadsto (g', n)$
  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*: $ids\ g' = ids\ g \cup \{n\}$
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 $inputs\ g'\ n \subseteq ids\ g' \wedge succ\ g'\ n \subseteq ids\ g' \wedge kind\ g'\ n \neq NoNode$
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*: $ids\ g' = ids\ g \cup \{n\}$
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 $inputs\ g'\ n \subseteq ids\ g' \wedge succ\ g'\ n \subseteq ids\ g' \wedge kind\ g'\ n \neq NoNode$
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 singletonD 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*: $ids\ g' = ids\ g4 \cup \{n\}$
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 $inputs\ g'\ n \subseteq ids\ g' \wedge succ\ g'\ n \subseteq ids\ g' \wedge kind\ g'\ n \neq NoNode$

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 insert-absorb insert-subset subset-insertI sup-bot-right*)
then show *?case* **using** *dom*
by (*smt* (*z3*) *ConditionalNodeNew.hyps*(*10*) *ConditionalNodeNew.hyps*(*2*) *ConditionalNodeNew.hyps*(*4*) *ConditionalNodeNew.hyps*(*6*) *ConditionalNodeNew.prems 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 (*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*: $ids\ g' = ids\ g2 \cup \{n\}$
by (*metis add-node-ids-subset add-node-lookup ids-add-update ids-some unrep.UnaryNodeNew unrep-contains*)
have *k*: $kind\ g'\ n = unary\text{-}node\ op\ x$
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 $inputs\ g'\ n \subseteq ids\ g' \wedge succ\ g'\ n \subseteq ids\ g' \wedge kind\ g'\ n \neq NoNode$
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 insert-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*: $ids\ g' = ids\ g3 \cup \{n\}$
by (*metis binary-inputs fresh-ids ids-add-update ids-some insert-not-empty not-in-g-inputs*)
have *k*: $kind\ g'\ n = bin\text{-}node\ op\ x\ y$
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 $inputs\ g'\ n \subseteq ids\ g' \wedge succ\ g'\ n \subseteq ids\ g' \wedge kind\ g'\ n \neq NoNode$

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 insert-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*: $g \oplus (ConstantExpr\ x) \leadsto (g', n)$

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*)
]

experiment begin
lemma
assumes *maximal-sharing g*
assumes *wf-closed g*
assumes $kind\ g\ y = AbsNode\ y'$
assumes $kind\ g\ y' = RefNode\ y''$
assumes $kind\ g\ y'' = ConstantNode\ v$
assumes $stamp\ g\ y'' = constantAsStamp\ v$
assumes $g \oplus (UnaryExpr\ UnaryAbs\ (ConstantExpr\ v)) \leadsto (g', n)$ (**is** $g \oplus ?e \leadsto (g', n)$)
shows $\neg(maximal\text{-}sharing\ g')$
using *assms*(*3*,*2*,*1*)
proof –
have $y'' \in ids\ g$
using *assms*(*5*) **by** *simp*
then have *List.member* (*sorted-list-of-set* (*ids g*)) y''
by (*metis member-def unwrap-sorted*)
then have *find* ($\lambda i.\ kind\ g\ i = ConstantNode\ v \wedge stamp\ g\ i = constantAsStamp\ v$) (*sorted-list-of-set* (*ids g*)) = *Some* y''
using *assms*(*5*,*6*) *find-Some-iff* **sorry**
then have $g \oplus ConstantExpr\ v \leadsto (g, y'')$
using *assms*(*5*) *ConstUnrepE* **sorry**
then show *?thesis* **sorry**
qed

end

lemma *conditional-rep-kind*:
  assumes $g \vdash n \simeq$ *ConditionalExpr ce te fe*
  assumes $g \vdash c \simeq ce$
  assumes $g \vdash t \simeq te$
  assumes $g \vdash f \simeq fe$
  assumes $\neg(\exists\, n'.$ *kind* $g\ n =$ *RefNode* $n')$
  shows *kind* $g\ n =$ *ConditionalNode* $c\ t\ f$
  using *assms* **apply** (*induction n ConditionalExpr ce te fe rule*: *rep.induct*) **defer**
  apply *meson* **using** *repDet* **sorry**

lemma *unary-rep-kind*:
  assumes $g \vdash n \simeq$ *UnaryExpr op xe*
  assumes $g \vdash x \simeq xe$
  assumes $\neg(\exists\, n'.$ *kind* $g\ n =$ *RefNode* $n')$
  shows *kind* $g\ n =$ *unary-node op x*
  using *assms* **apply** (*cases op*) **using** *AbsNodeE* **sorry**

lemma *binary-rep-kind*:
  assumes $g \vdash n \simeq$ *BinaryExpr op xe ye*
  assumes $g \vdash x \simeq xe$
  assumes $g \vdash y \simeq ye$
  assumes $\neg(\exists\, n'.$ *kind* $g\ n =$ *RefNode* $n')$
  shows *kind* $g\ n =$ *bin-node op x y*
  using *assms* **sorry**

theorem *unrep-maximal-sharing*:
  assumes *maximal-sharing g*
  assumes *wf-closed g*
  assumes $g \oplus e \rightsquigarrow (g', n)$
  shows *maximal-sharing* g'
  using *assms*(*3*,*2*,*1*)
  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 *kind* $g'\ n =$ *ConstantNode c*
      using *find-new-kind* **by** *blast*
    then have *repn*: $g' \vdash n \simeq$ *ConstantExpr c*
      using *rep.ConstantNode* **by** *simp*
     from *ConstantNodeNew* **have** *real-node*: $\neg$(*is-RefNode* (*ConstantNode c*)) $\wedge$
ConstantNode c $\neq$ *NoNode*
      by *simp*
    then have *dom*: *true-ids* $g' =$ *true-ids* $g \cup \{n\}$
      using *ConstantNodeNew.hyps*(*2*) *ConstantNodeNew.hyps*(*3*) *fresh-ids*
      by (*meson true-ids-add-update*)
    have *new*: $n \notin$ *ids g*

using *fresh-ids*
using *ConstantNodeNew.hyps*(*2*) **by** *blast*
obtain *new* **where** $new = true\text{-}ids\ g' - true\text{-}ids\ g$
by *simp*
then have *new-def*: $new = \{n\}$
by (*metis* (*no-types*, *lifting*) *DiffE Diff-cancel IRGraph.true-ids-def Un-insert-right dom insert-Diff-if new sup-bot-right*)
then have *unchanged*: $(new \trianglelefteq as\text{-}set\ g') = as\text{-}set\ g$
using *ConstantNodeNew*(*3*) *new add-node-as-set-eq*
by *presburger*
then have *kind-eq*: $\forall n' .\ n' \notin new \longrightarrow kind\ g\ n' = kind\ g'\ n'$
by (*metis ConstantNodeNew.hyps*(*3*) ‹$new = \{n\}$› *add-node-as-set dual-order.eq-iff not-excluded-keep-type not-in-g*)
from *unchanged* **have** *stamp-eq*: $\forall n' \in ids\ g\ .\ n' \notin new \longrightarrow stamp\ g\ n' = stamp\ g'\ n'$
using *not-excluded-keep-type new-def new*
by (*metis ConstantNodeNew.hyps*(*3*) *add-node-as-set*)
show *?case* **unfolding** *maximal-sharing* **apply** (*rule allI*; *rule allI*; *rule impI*)
using *ConstantNodeNew*(*5*) **unfolding** *maximal-sharing* **apply** *auto*
proof −
fix $n_1\ n_2\ e$
assume *1*: $\forall n_1\ n_2.$
$n_1 \in true\text{-}ids\ g \wedge n_2 \in true\text{-}ids\ g \longrightarrow$
$(\exists e.\ (g \vdash n_1 \simeq e) \wedge (g \vdash n_2 \simeq e) \wedge stamp\ g\ n_1 = stamp\ g\ n_2) \longrightarrow n_1 = n_2$
assume $n_1 \in true\text{-}ids\ g'$
assume $n_2 \in true\text{-}ids\ g'$
show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow stamp\ g'\ n_1 = stamp\ g'\ n_2 \Longrightarrow n_1 = n_2$
proof (*cases* $n_1 \in true\text{-}ids\ g$)
case *n1*: *True*
then show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow stamp\ g'\ n_1 = stamp\ g'\ n_2 \Longrightarrow n_1 = n_2$
proof (*cases* $n_2 \in true\text{-}ids\ g$)
case *n2*: *True*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stmp*: $stamp\ g'\ n_1 = stamp\ g'\ n_2$
have *n1rep*: $g \vdash n_1 \simeq e$
using *n1rep' kind-eq stamp-eq new-def add-preserves-rep*
using *ConstantNodeNew.prems*(*1*) *IRGraph.true-ids-def n1 unchanged* **by** *auto*
have *n2rep*: $g \vdash n_2 \simeq e$
using *n2rep' kind-eq stamp-eq new-def add-preserves-rep*
using *ConstantNodeNew.prems*(*1*) *IRGraph.true-ids-def n2 unchanged* **by** *auto*
have $stamp\ g\ n_1 = stamp\ g\ n_2$
by (*metis ConstantNodeNew.hyps*(*3*) *stmp fresh-node-subset n1rep n2rep new subset-stamp*)
then show *?thesis* **using** *1*

using *n1 n2*
using *n1rep n2rep* **by** *blast*
next
case *n2*: *False*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stmp*: *stamp* g' n_1 = *stamp* g' n_2
have *n2-def*: $n_2 = n$
using ‹$n_2 \in$ *true-ids* g'› *dom n2* **by** *auto*
have *n1rep*: $g \vdash n_1 \simeq$ *ConstantExpr c*
by (*metis* (*no-types*, *lifting*) *ConstantNodeNew.prems*(*1*) *DiffE IRGraph.true-ids-def add-preserves-rep n1 n1rep' n2-def n2rep' repDet repn unchanged*)
then have *n1in*: $n_1 \in$ *ids g*
using *no-encoding* **by** *metis*
have *k*: *kind g* n_1 = *ConstantNode c*
using *TreeToGraphThms.true-ids-def n1 n1rep* **by** *force*
have *s*: *stamp g* n_1 = *constantAsStamp c*
by (*metis ConstantNodeNew.hyps*(*3*) *real-node n2-def stmp find-new-stamp fresh-node-subset n1rep new subset-stamp*)
from *k s* **show** *?thesis*
using *find-none ConstantNodeNew.hyps*(*1*) *n1in* **by** *blast*
qed
next
case *n1*: *False*
then show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow$ *stamp* g' n_1 = *stamp* g' n_2 $\Longrightarrow n_1 = n_2$
proof (*cases* $n_2 \in$ *true-ids g*)
case *n2*: *True*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stmp*: *stamp* g' n_1 = *stamp* g' n_2
have *n1-def*: $n_1 = n$
using ‹$n_1 \in$ *true-ids* g'› *dom n1* **by** *auto*
have *n2in*: $n_2 \in$ *ids g*
using *IRGraph.true-ids-def n2* **by** *auto*
have *k*: *kind g* n_2 = *ConstantNode c*
by (*metis* (*mono-tags*, *lifting*) *ConstantNodeE ConstantNodeNew.prems*(*1*) *DiffE IRGraph.true-ids-def add-preserves-rep mem-Collect-eq n1-def n1rep' n2 n2rep' repDet repn unchanged*)
have *s*: *stamp g* n_2 = *constantAsStamp c*
by (*metis ConstantNodeNew.hyps*(*3*) *TreeToGraphThms.new-def add-node-lookup n1-def n2in real-node stamp-eq stmp unchanged*)
from *k s* **show** *?thesis*
using *find-none ConstantNodeNew.hyps*(*1*) *n2in* **by** *blast*
next
case *n2*: *False*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stamp* g' n_1 = *stamp* g' n_2

have $n_1 = n \wedge n_2 = n$
using ‹$n_1 \in$ *true-ids* g'› *dom n1*
using ‹$n_2 \in$ *true-ids* g'› *n2* **by** *blast*
then show *?thesis*
by *simp*
qed
qed
qed
next
case (*ParameterNodeSame g i s n*)
then show *?case* **by** *blast*
next
case (*ParameterNodeNew g i s n g'*)
then have *k*: *kind g' n* = *ParameterNode i*
using *find-new-kind* **by** *blast*
have *stamp g' n* = *s*
using *ParameterNodeNew.hyps*(*3*) *find-new-stamp* **by** *blast*
then have *repn*: $g' \vdash n \simeq$ *ParameterExpr i s*
using *rep.ParameterNode k* **by** *simp*
from *ConstantNodeNew* **have** ¬(*is-RefNode* (*ParameterNode i*)) ∧ *ParameterNode i* ≠ *NoNode*
by *simp*
then have *dom*: *true-ids g'* = *true-ids g* ∪ {*n*}
using *ParameterNodeNew.hyps*(*2*) *ParameterNodeNew.hyps*(*3*) *fresh-ids*
by (*meson true-ids-add-update*)
have *new*: *n* ∉ *ids g*
using *fresh-ids*
using *ParameterNodeNew.hyps*(*2*) **by** *blast*
obtain *new* **where** *new* = *true-ids g'* − *true-ids g*
by *simp*
then have *new-def*: *new* = {*n*}
by (*metis* (*no-types*, *lifting*) *DiffE Diff-cancel IRGraph.true-ids-def Un-insert-right dom insert-Diff-if new sup-bot-right*)
then have *unchanged*: (*new* ⊴ *as-set g'*) = *as-set g*
using *ParameterNodeNew*(*3*) *new add-node-as-set-eq*
by *presburger*
then have *kind-eq*: ∀ *n'* . *n'* ∉ *new* ⟶ *kind g n'* = *kind g' n'*
by (*metis ParameterNodeNew.hyps*(*3*) ‹*new* = {*n*}› *add-node-as-set dual-order.eq-iff not-excluded-keep-type not-in-g*)
from *unchanged* **have** *stamp-eq*: ∀ *n'* ∈ *ids g* . *n'* ∉ *new* ⟶ *stamp g n'* = *stamp g' n'*
using *not-excluded-keep-type new-def new*
by (*metis ParameterNodeNew.hyps*(*3*) *add-node-as-set*)
show *?case* **unfolding** *maximal-sharing* **apply** (*rule allI*; *rule allI*; *rule impI*)
using *ParameterNodeNew*(*5*) **unfolding** *maximal-sharing* **apply** *auto*
proof −
fix n_1 n_2 *e*
assume *1*: ∀ n_1 n_2.
$n_1 \in$ *true-ids* $g \wedge n_2 \in$ *true-ids* $g \longrightarrow$

$(\exists\, e.\ (g \vdash n_1 \simeq e) \wedge (g \vdash n_2 \simeq e) \wedge stamp\ g\ n_1 = stamp\ g\ n_2) \longrightarrow n_1 = n_2$
assume $n_1 \in$ *true-ids* g'
assume $n_2 \in$ *true-ids* g'
show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow stamp\ g'\ n_1 = stamp\ g'\ n_2 \Longrightarrow n_1 = n_2$
proof (*cases* $n_1 \in$ *true-ids* g)
case *n1*: *True*
then show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow stamp\ g'\ n_1 = stamp\ g'\ n_2 \Longrightarrow n_1 = n_2$
proof (*cases* $n_2 \in$ *true-ids* g)
case *n2*: *True*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume $stamp\ g'\ n_1 = stamp\ g'\ n_2$
have *n1rep*: $g \vdash n_1 \simeq e$
using *n1rep' kind-eq stamp-eq new-def add-preserves-rep*
using *ParameterNodeNew.prems*(*1*) *IRGraph.true-ids-def n1 unchanged* **by** *auto*
have *n2rep*: $g \vdash n_2 \simeq e$
using *n2rep' kind-eq stamp-eq new-def add-preserves-rep*
using *ParameterNodeNew.prems*(*1*) *IRGraph.true-ids-def n2 unchanged* **by** *auto*
have $stamp\ g\ n_1 = stamp\ g\ n_2$
by (*metis ParameterNodeNew.hyps*(*3*) ‹$stamp\ g'\ n_1 = stamp\ g'\ n_2$› *fresh-node-subset n1rep n2rep new subset-stamp*)
then show *?thesis* **using** *1*
using *n1 n2*
using *n1rep n2rep* **by** *blast*
next
case *n2*: *False*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume $stamp\ g'\ n_1 = stamp\ g'\ n_2$
have $n_2 = n$
using ‹$n_2 \in$ *true-ids* g'› *dom n2* **by** *auto*
then have *ne*: $n_2 \notin$ *ids* g
using *new n2* **by** *blast*
have *n1rep*: $g \vdash n_1 \simeq e$
using *n1rep' kind-eq stamp-eq new-def add-preserves-rep*
using *ParameterNodeNew.prems*(*1*) *IRGraph.true-ids-def n1 unchanged* **by** *auto*
have *n2rep*: $g \vdash n_2 \simeq e$
using *n2rep' kind-eq stamp-eq new-def add-preserves-rep*
using *ParameterNodeNew.prems*(*1*) *IRGraph.true-ids-def unchanged*
by (*metis* (*no-types*, *lifting*) *IRNode.disc*(*2703*) *ParameterNodeE ParameterNodeNew.hyps*(*1*) *TreeToGraphThms.true-ids-def* ‹$n_2 = n$› *find-none mem-Collect-eq n1 n1rep' repDet repn*)
then show *?thesis*
using *n2rep not-in-no-rep ne* **by** *blast*

qed
next
case *n1*: *False*
then show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow$ *stamp* g' n_1 = *stamp* g' n_2 $\Longrightarrow n_1 = n_2$
proof (*cases* $n_2 \in$ *true-ids* g)
case *n2*: *True*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stamp* g' n_1 = *stamp* g' n_2
have $n_1 = n$
using ‹$n_1 \in$ *true-ids* g'› *dom n1* **by** *auto*
then have *ne*: $n_1 \notin$ *ids* g
using *new n2* **by** *blast*
have *n1rep*: $g \vdash n_1 \simeq e$
using *n1rep' kind-eq stamp-eq new-def add-preserves-rep*
using *ParameterNodeNew.prems*(*1*) *IRGraph.true-ids-def n1 unchanged*
by (*metis* (*no-types*, *lifting*) *IRNode.disc*(*2703*) *ParameterNodeE ParameterNodeNew.hyps*(*1*) *TreeToGraphThms.true-ids-def* ‹$n_1 = n$› *find-none mem-Collect-eq n2 n2rep' repDet repn*)
then show *?thesis*
using *n1rep not-in-no-rep ne* **by** *blast*
next
case *n2*: *False*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stamp* g' n_1 = *stamp* g' n_2
have $n_1 = n \wedge n_2 = n$
using ‹$n_1 \in$ *true-ids* g'› *dom n1*
using ‹$n_2 \in$ *true-ids* g'› *n2* **by** *blast*
then show *?thesis*
by *simp*
qed
qed
qed
next
case (*ConditionalNodeSame g ce g2 c te g3 t fe g4 f s' n*)
then show *?case*
using *unrep-preserves-closure* **by** *blast*
next
case (*ConditionalNodeNew g ce g2 c te g3 t fe g4 f s' n g'*)
then have *k*: *kind* g' n = *ConditionalNode c t f*
using *find-new-kind* **by** *blast*
have *stamp* g' $n = s'$
using *ConditionalNodeNew.hyps*(*10*) *IRNode.distinct*(*591*) *find-new-stamp* **by** *blast*
then have *repn*: $g' \vdash n \simeq$ *ConditionalExpr ce te fe*
using *rep.ConditionalNode k*
by (*metis ConditionalNodeNew.hyps*(*1*) *ConditionalNodeNew.hyps*(*10*) *Condi-*

tionalNodeNew.hyps(*3*) *ConditionalNodeNew.hyps*(*5*) *ConditionalNodeNew.hyps*(*9*) *fresh-ids fresh-node-subset subset-implies-evals term-graph-reconstruction*)

from *ConstantNodeNew* **have** $\neg$(*is-RefNode* (*ConditionalNode c t f*)) $\wedge$ *ConditionalNode c t f* $\neq$ *NoNode*

by *simp*

then have *dom*: *true-ids* g' = *true-ids* $g4 \cup \{n\}$

using *ConditionalNodeNew.hyps*(*10*) *ConditionalNodeNew.hyps*(*9*) *fresh-ids true-ids-add-update* **by** *presburger*

have *new*: $n \notin$ *ids g*

using *fresh-ids*

by (*meson ConditionalNodeNew.hyps*(*1*) *ConditionalNodeNew.hyps*(*3*) *ConditionalNodeNew.hyps*(*5*) *ConditionalNodeNew.hyps*(*9*) *unrep-preserves-contains*)

obtain *new* **where** *new* = *true-ids* g' $-$ *true-ids g4*

by *simp*

then have *new-def*: *new* = $\{n\}$

using *dom*

by (*metis ConditionalNodeNew.hyps*(*9*) *DiffD1 DiffI Diff-cancel Diff-insert Un-insert-right boolean-algebra.disj-zero-right fresh-ids insertCI insert-Diff true-ids*)

then have *unchanged*: (*new* $\trianglelefteq$ *as-set* g') = *as-set g4*

using *new add-node-as-set-eq*

using *ConditionalNodeNew.hyps*(*10*) *ConditionalNodeNew.hyps*(*9*) *fresh-ids* **by** *presburger*

then have *kind-eq*: $\forall n'$. $n' \notin$ *new* $\longrightarrow$ *kind g4* n' = *kind* g' n'

by (*metis ConditionalNodeNew.hyps*(*10*) *add-node-as-set equalityE local.new-def not-excluded-keep-type not-in-g*)

from *unchanged* **have** *stamp-eq*: $\forall n' \in$ *ids g* . $n' \notin$ *new* $\longrightarrow$ *stamp g4* n' = *stamp* g' n'

using *not-excluded-keep-type new-def new*

by (*metis ConditionalNodeNew.hyps*(*1*) *ConditionalNodeNew.hyps*(*10*) *ConditionalNodeNew.hyps*(*3*) *ConditionalNodeNew.hyps*(*5*) *add-node-as-set unrep-preserves-contains*)

have *max-g4*: *maximal-sharing g4*

using *ConditionalNodeNew.hyps*(*1*) *ConditionalNodeNew.hyps*(*2*) *ConditionalNodeNew.hyps*(*3*) *ConditionalNodeNew.hyps*(*4*) *ConditionalNodeNew.hyps*(*6*) *ConditionalNodeNew.prems*(*1*) *ConditionalNodeNew.prems*(*2*) *unrep-preserves-closure* **by** *blast*

show *?case* **unfolding** *maximal-sharing* **apply** (*rule allI*; *rule allI*; *rule impI*)

using *max-g4* **unfolding** *maximal-sharing* **apply** *auto*

proof –

fix n_1 n_2 e

assume *1*: $\forall n_1\ n_2$.

$n_1 \in$ *true-ids g4* $\wedge$ $n_2 \in$ *true-ids g4* $\longrightarrow$

($\exists e$. (*g4* $\vdash n_1 \simeq e$) $\wedge$ (*g4* $\vdash n_2 \simeq e$) $\wedge$ *stamp g4* n_1 = *stamp g4* n_2) $\longrightarrow$ $n_1 = n_2$

assume $n_1 \in$ *true-ids* g'

assume $n_2 \in$ *true-ids* g'

show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow$ *stamp* g' n_1 = *stamp* g' $n_2 \Longrightarrow n_1 = n_2$

proof (*cases* $n_1 \in$ *true-ids g4*)

case *n1*: *True*

then show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow$ *stamp* g' n_1 = *stamp* g' n_2 $\Longrightarrow n_1 = n_2$
proof (*cases* $n_2 \in$ *true-ids g4*)
case *n2*: *True*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stamp* g' n_1 = *stamp* g' n_2
have *n1rep*: *g4* $\vdash n_1 \simeq e$
using *n1rep' kind-eq stamp-eq new-def add-preserves-rep*
using *ConditionalNodeNew.prems*(*1*) *IRGraph.true-ids-def n1 unchanged*
by (*metis* (*mono-tags*, *lifting*) *ConditionalNodeNew.hyps*(*1*) *ConditionalNodeNew.hyps*(*3*) *ConditionalNodeNew.hyps*(*5*) *DiffE unrep-preserves-closure*)
have *n2rep*: *g4* $\vdash n_2 \simeq e$
using *n2rep' kind-eq stamp-eq new-def add-preserves-rep*
using *ConditionalNodeNew.prems*(*1*) *IRGraph.true-ids-def n2 unchanged*
by (*metis* (*no-types*, *lifting*) *ConditionalNodeNew.hyps*(*1*) *ConditionalNodeNew.hyps*(*3*) *ConditionalNodeNew.hyps*(*5*) *DiffE unrep-preserves-closure*)
have *stamp g4* n_1 = *stamp g4* n_2
by (*metis ConditionalNodeNew.hyps*(*10*) *ConditionalNodeNew.hyps*(*9*) ‹*stamp* g' n_1 = *stamp* g' n_2› *fresh-ids fresh-node-subset n1rep n2rep subset-stamp*)
then show *?thesis* **using** *1*
using *n1 n2*
using *n1rep n2rep* **by** *blast*
next
case *n2*: *False*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stmp*: *stamp* g' n_1 = *stamp* g' n_2
have *n2-def*: $n_2 = n$
using ‹$n_2 \in$ *true-ids* g'› *dom n2* **by** *auto*
have *n1rep*: *g4* $\vdash n_1 \simeq$ *ConditionalExpr ce te fe*
by (*metis* (*no-types*, *lifting*) *ConditionalNodeNew.hyps*(*1*) *ConditionalNodeNew.hyps*(*3*) *ConditionalNodeNew.hyps*(*5*) *ConditionalNodeNew.prems*(*1*) *Diff-iff IRGraph.true-ids-def add-preserves-rep n1 n1rep' n2-def n2rep' repDet repn unchanged unrep-preserves-closure*)
then have *n1in*: $n_1 \in$ *ids g4*
using *no-encoding* **by** *metis*

have *rep*: (*g4* $\vdash c \simeq ce$) $\wedge$ (*g4* $\vdash t \simeq te$) $\wedge$ (*g4* $\vdash f \simeq fe$)
by (*meson ConditionalNodeNew.hyps*(*1*) *ConditionalNodeNew.hyps*(*3*) *ConditionalNodeNew.hyps*(*5*) *subset-implies-evals term-graph-reconstruction*)
have *not-ref*: $\neg(\exists n'.$ *kind g4* n_1 = *RefNode* $n')$
using *TreeToGraphThms.true-ids-def n1* **by** *fastforce*
then have *kind g4* n_1 = *ConditionalNode c t f*
using *conditional-rep-kind*
using *local.rep n1rep* **by** *presburger*
then show *?thesis*
using *find-none ConditionalNodeNew.hyps*(*8*) *n1in*
by (*metis ConditionalNodeNew.hyps*(*10*) *ConditionalNodeNew.hyps*(*9*)

$\langle$*stamp* g' n = $s'$$\rangle$ *fresh-ids fresh-node-subset n1rep n2-def stmp subset-stamp*)
    qed
   next
    case *n1*: *False*
    then show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow$ *stamp* g' n_1 = *stamp* g' n_2 $\Longrightarrow n_1 = n_2$
    proof (*cases* $n_2 \in$ *true-ids g4*)
     case *n2*: *True*
     assume *n1rep'*: $g' \vdash n_1 \simeq e$
     assume *n2rep'*: $g' \vdash n_2 \simeq e$
     assume *stamp* g' n_1 = *stamp* g' n_2
     have *new-n1*: $n_1 = n$
      using $\langle n_1 \in$ *true-ids* $g'\rangle$ *dom n1* **by** *auto*
     then have *ne*: $n_1 \notin$ *ids g4*
      using *new n1*
      using *ConditionalNodeNew.hyps*(*9*) *fresh-ids* **by** *blast*
     have *unrep-cond*: *g4* $\vdash n_2 \simeq$ *ConditionalExpr ce te fe*
      using *n1rep' kind-eq stamp-eq new-def add-preserves-rep*
      using *ConditionalNodeNew.prems*(*1*) *IRGraph.true-ids-def n2 unchanged*
      by (*metis* (*no-types*, *lifting*) *ConditionalNodeNew.hyps*(*1*) *ConditionalNodeNew.hyps*(*3*) *ConditionalNodeNew.hyps*(*5*) *DiffD1 n2rep' new-n1 repDet repn unrep-preserves-closure*)
     have *rep*: (*g4* $\vdash c \simeq ce$) $\wedge$ (*g4* $\vdash t \simeq te$) $\wedge$ (*g4* $\vdash f \simeq fe$)
       by (*meson ConditionalNodeNew.hyps*(*1*) *ConditionalNodeNew.hyps*(*3*) *ConditionalNodeNew.hyps*(*5*) *subset-implies-evals term-graph-reconstruction*)
     have *not-ref*: $\neg(\exists\, n'.$ *kind g4* n_2 = *RefNode* n')
      using *TreeToGraphThms.true-ids-def n2* **by** *fastforce*
     then have *kind g4* n_2 = *ConditionalNode c t f*
      using *conditional-rep-kind*
      using *local.rep unrep-cond* **by** *presburger*
     then show *?thesis* **using** *find-none ConditionalNodeNew.hyps*(*8*)
      by (*metis ConditionalNodeNew.hyps*(*10*) $\langle$*stamp* g' n = $s'$$\rangle$ $\langle$*stamp* g' n_1 = *stamp* g' $n_2$$\rangle$ *encodes-contains fresh-node-subset ne new-n1 not-in-g subset-stamp unrep-cond*)
    next
     case *n2*: *False*
     assume *n1rep'*: $g' \vdash n_1 \simeq e$
     assume *n2rep'*: $g' \vdash n_2 \simeq e$
     assume *stamp* g' n_1 = *stamp* g' n_2
     have $n_1 = n \wedge n_2 = n$
      using $\langle n_1 \in$ *true-ids* $g'\rangle$ *dom n1*
      using $\langle n_2 \in$ *true-ids* $g'\rangle$ *n2*
      by *simp*
     then show *?thesis*
      by *simp*
    qed
   qed
  qed
 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 *k*: *kind g′ n = unary-node op x*
using *find-new-kind*
by (*metis add-node-lookup fresh-ids ids-some*)
have *stamp g′ n = s′*
by (*metis UnaryNodeNew.hyps*(*6*) *empty-iff find-new-stamp ids-some insertI1 k not-in-g-inputs unary-inputs*)
then have *repn*: $g' \vdash n \simeq$ *UnaryExpr op xe*
using *k*
using *UnaryNodeNew.hyps*(*1*) *UnaryNodeNew.hyps*(*3*) *UnaryNodeNew.hyps*(*4*) *UnaryNodeNew.hyps*(*5*) *UnaryNodeNew.hyps*(*6*) *term-graph-reconstruction unrep.UnaryNodeNew* **by** *blast*
from *ConstantNodeNew* **have** ¬(*is-RefNode* (*unary-node op x*)) ∧ *unary-node op x* ≠ *NoNode*
by (*cases op*; *auto*)
then have *dom*: *true-ids g′ = true-ids g2* ∪ {*n*}
using *UnaryNodeNew.hyps*(*5*) *UnaryNodeNew.hyps*(*6*) *fresh-ids true-ids-add-update* **by** *presburger*
have *new*: *n* ∉ *ids g*
using *fresh-ids*
by (*meson UnaryNodeNew.hyps*(*1*) *UnaryNodeNew.hyps*(*5*) *unrep-preserves-contains*)
obtain *new* **where** *new = true-ids g′ − true-ids g2*
by *simp*
then have *new-def*: *new* = {*n*}
using *dom*
by (*metis Diff-cancel Diff-iff Un-insert-right UnaryNodeNew.hyps*(*5*) *fresh-ids insert-Diff-if sup-bot.right-neutral true-ids*)
then have *unchanged*: (*new* ⊴ *as-set g′*) = *as-set g2*
using *new add-node-as-set-eq*
using *UnaryNodeNew.hyps*(*5*) *UnaryNodeNew.hyps*(*6*) *fresh-ids* **by** *presburger*
then have *kind-eq*: ∀ *n′* . *n′* ∉ *new* ⟶ *kind g2 n′ = kind g′ n′*
by (*metis UnaryNodeNew.hyps*(*6*) *add-node-as-set equalityD1 local.new-def not-excluded-keep-type not-in-g*)
from *unchanged* **have** *stamp-eq*: ∀ *n′* ∈ *ids g* . *n′* ∉ *new* ⟶ *stamp g2 n′ = stamp g′ n′*
using *not-excluded-keep-type new-def new*
by (*metis UnaryNodeNew.hyps*(*1*) *UnaryNodeNew.hyps*(*6*) *add-node-as-set unrep-preserves-contains*)
have *max-g2*: *maximal-sharing g2*
by (*simp add*: *UnaryNodeNew.hyps*(*2*) *UnaryNodeNew.prems*(*1*) *UnaryNode-New.prems*(*2*))
show *?case* **unfolding** *maximal-sharing* **apply** (*rule allI*; *rule allI*; *rule impI*)
using *max-g2* **unfolding** *maximal-sharing* **apply** *auto*
proof −
fix n_1 n_2 *e*
assume *1*: ∀ n_1 n_2.

$n_1 \in$ *true-ids* $g2 \wedge n_2 \in$ *true-ids* $g2 \longrightarrow$
$(\exists\, e.\ (g2 \vdash n_1 \simeq e) \wedge (g2 \vdash n_2 \simeq e) \wedge$ *stamp* $g2\ n_1 =$ *stamp* $g2\ n_2) \longrightarrow$
$n_1 = n_2$
assume $n_1 \in$ *true-ids* g'
assume $n_2 \in$ *true-ids* g'
show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow$ *stamp* $g'\ n_1 =$ *stamp* $g'\ n_2 \Longrightarrow n_1 =$
n_2
proof (*cases* $n_1 \in$ *true-ids g2*)
case *n1*: *True*
then show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow$ *stamp* $g'\ n_1 =$ *stamp* $g'\ n_2$
$\Longrightarrow n_1 = n_2$
proof (*cases* $n_2 \in$ *true-ids g2*)
case *n2*: *True*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stamp* $g'\ n_1 =$ *stamp* $g'\ n_2$
have *n1rep*: $g2 \vdash n_1 \simeq e$
using *n1rep' kind-eq stamp-eq new-def add-preserves-rep*
using *Diff-iff IRGraph.true-ids-def UnaryNodeNew.hyps*(*1*) *UnaryNode-New.prems*(*1*) *n1 unchanged unrep-preserves-closure* **by** *auto*
have *n2rep*: $g2 \vdash n_2 \simeq e$
using *n2rep' kind-eq stamp-eq new-def add-preserves-rep*
by (*metis* (*no-types*, *lifting*) *Diff-iff IRGraph.true-ids-def UnaryNode-New.hyps*(*1*) *UnaryNodeNew.prems*(*1*) *n2 unchanged unrep-preserves-closure*)
have *stamp* $g2\ n_1 =$ *stamp* $g2\ n_2$
by (*metis UnaryNodeNew.hyps*(*5*) *UnaryNodeNew.hyps*(*6*) ‹*stamp* $g'\ n_1$ $=$ *stamp* $g'\ n_2$› *fresh-ids fresh-node-subset n1rep n2rep subset-stamp*)
then show *?thesis* **using** *1*
using *n1 n2*
using *n1rep n2rep* **by** *blast*
next
case *n2*: *False*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stamp* $g'\ n_1 =$ *stamp* $g'\ n_2$
have *new-n2*: $n_2 = n$
using ‹$n_2 \in$ *true-ids* g'› *dom n2* **by** *auto*
then have *ne*: $n_2 \notin$ *ids g2*
using *new n2*
using *UnaryNodeNew.hyps*(*5*) *fresh-ids* **by** *blast*
have *unrep-un*: $g2 \vdash n_1 \simeq$ *UnaryExpr op xe*
using *n1rep' kind-eq stamp-eq new-def add-preserves-rep*
by (*metis* (*no-types*, *lifting*) *Diff-iff IRGraph.true-ids-def UnaryNode-New.hyps*(*1*) *UnaryNodeNew.prems*(*1*) *n1 n2rep' new-n2 repDet repn unchanged unrep-preserves-closure*)
have *rep*: $(g2 \vdash x \simeq xe)$
using *UnaryNodeNew.hyps*(*1*) *term-graph-reconstruction* **by** *auto*
have *not-ref*: $\neg(\exists\, n'.$ *kind* $g2\ n_1 =$ *RefNode* $n')$
using *TreeToGraphThms.true-ids-def n1* **by** *force*

then have *kind g2* n_1 = *unary-node op x*
  using *unrep-un unary-rep-kind rep* **by** *simp*

then show *?thesis* **using** *find-none UnaryNodeNew.hyps(4)*
  by (*metis UnaryNodeNew.hyps(6)* ‹*stamp* g' *n* = s'› ‹*stamp* g' n_1 = *stamp* g' n_2› *fresh-node-subset ne new-n2 no-encoding subset-stamp unrep-un*)
qed
next
case *n1*: *False*
then show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow$ *stamp* g' n_1 = *stamp* g' n_2 $\Longrightarrow n_1 = n_2$
proof (*cases* $n_2 \in$ *true-ids g2*)
case *n2*: *True*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stamp* g' n_1 = *stamp* g' n_2
have *new-n1*: $n_1 = n$
  using ‹$n_1 \in$ *true-ids* g'› *dom n1* **by** *auto*
then have *ne*: $n_1 \notin$ *ids g2*
  using *new n1*
  using *UnaryNodeNew.hyps(5) fresh-ids* **by** *blast*
have *unrep-un*: *g2* $\vdash n_2 \simeq$ *UnaryExpr op xe*
  using *n1rep' kind-eq stamp-eq new-def add-preserves-rep*
  by (*metis* (*no-types, lifting*) *Diff-iff IRGraph.true-ids-def UnaryNodeNew.hyps(1) UnaryNodeNew.prems(1) n2 n2rep' new-n1 repDet repn unchanged unrep-preserves-closure*)
have *rep*: (*g2* $\vdash x \simeq xe$)
  using *UnaryNodeNew.hyps(1) term-graph-reconstruction* **by** *presburger*
have *not-ref*: $\neg(\exists\, n'.$ *kind g2* n_2 = *RefNode* $n')$
  using *TreeToGraphThms.true-ids-def n2* **by** *fastforce*
then have *kind g2* n_2 = *unary-node op x*
  using *unary-rep-kind*
  using *local.rep unrep-un* **by** *presburger*
then show *?thesis* **using** *find-none UnaryNodeNew.hyps(4)*
  by (*metis UnaryNodeNew.hyps(6)* ‹*stamp* g' *n* = s'› ‹*stamp* g' n_1 = *stamp* g' n_2› *fresh-node-subset ne new-n1 no-encoding subset-stamp unrep-un*)
next
case *n2*: *False*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stamp* g' n_1 = *stamp* g' n_2
have $n_1 = n \wedge n_2 = n$
  using ‹$n_1 \in$ *true-ids* g'› *dom n1*
  using ‹$n_2 \in$ *true-ids* g'› *n2*
  by *simp*
then show *?thesis*
  by *simp*
qed
qed

qed
next
case (*BinaryNodeSame g xe g2 x ye g3 y s′ op n*)
then show *?case*
using *unrep-preserves-closure* **by** *blast*
next
case (*BinaryNodeNew g xe g2 x ye g3 y s′ op n g′*)
then have *k*: *kind g′ n = bin-node op x y*
using *find-new-kind*
by (*metis add-node-lookup fresh-ids ids-some*)
have *stamp g′ n = s′*
by (*metis BinaryNodeNew.hyps*(*1*) *BinaryNodeNew.hyps*(*3*) *BinaryNode-New.hyps*(*5*) *BinaryNodeNew.hyps*(*6*) *BinaryNodeNew.hyps*(*7*) *BinaryNodeNew.hyps*(*8*) *find-new-stamp ids-some k unrep.BinaryNodeNew unrep-contains*)
then have *repn*: $g' \vdash n \simeq$ *BinaryExpr op xe ye*
using *k*
using *BinaryNodeNew.hyps*(*1*) *BinaryNodeNew.hyps*(*3*) *BinaryNodeNew.hyps*(*5*) *BinaryNodeNew.hyps*(*6*) *BinaryNodeNew.hyps*(*7*) *BinaryNodeNew.hyps*(*8*) *term-graph-reconstruction unrep.BinaryNodeNew* **by** *blast*
from *BinaryNodeNew* **have** ¬(*is-RefNode* (*bin-node op x y*)) ∧ *bin-node op x y* ≠ *NoNode*
by (*cases op*; *auto*)
then have *dom*: *true-ids g′ = true-ids g3* ∪ {*n*}
using *BinaryNodeNew.hyps*(*7*) *BinaryNodeNew.hyps*(*8*) *fresh-ids true-ids-add-update* **by** *presburger*
have *new*: *n* ∉ *ids g*
using *fresh-ids*
by (*meson BinaryNodeNew.hyps*(*1*) *BinaryNodeNew.hyps*(*3*) *BinaryNode-New.hyps*(*7*) *unrep-preserves-contains*)
obtain *new* **where** *new = true-ids g′ − true-ids g3*
by *simp*
then have *new-def*: *new* = {*n*}
using *dom*
by (*metis BinaryNodeNew.hyps*(*7*) *Diff-cancel Diff-iff Un-insert-right fresh-ids insert-Diff-if sup-bot.right-neutral true-ids*)
then have *unchanged*: (*new* ⊴ *as-set g′*) = *as-set g3*
using *new add-node-as-set-eq*
using *BinaryNodeNew.hyps*(*7*) *BinaryNodeNew.hyps*(*8*) *fresh-ids* **by** *presburger*
then have *kind-eq*: ∀ *n′* . *n′* ∉ *new* ⟶ *kind g3 n′ = kind g′ n′*
by (*metis BinaryNodeNew.hyps*(*8*) *add-node-as-set equalityD1 local.new-def not-excluded-keep-type not-in-g*)
from *unchanged* **have** *stamp-eq*: ∀ *n′* ∈ *ids g* . *n′* ∉ *new* ⟶ *stamp g3 n′* = *stamp g′ n′*
using *not-excluded-keep-type new-def new*
by (*metis BinaryNodeNew.hyps*(*1*) *BinaryNodeNew.hyps*(*3*) *BinaryNode-New.hyps*(*8*) *add-node-as-set unrep-preserves-contains*)
have *max-g3*: *maximal-sharing g3*
using *BinaryNodeNew.hyps*(*1*) *BinaryNodeNew.hyps*(*2*) *BinaryNodeNew.hyps*(*4*) *BinaryNodeNew.prems*(*1*) *BinaryNodeNew.prems*(*2*) *unrep-preserves-closure* **by** *blast*

show *?case* **unfolding** *maximal-sharing* **apply** (*rule allI*; *rule allI*; *rule impI*)
using *max-g3* **unfolding** *maximal-sharing* **apply** *auto*
proof −
fix n_1 n_2 *e*
assume *1*: $\forall n_1\ n_2.$
$n_1 \in$ *true-ids g3* $\wedge$ $n_2 \in$ *true-ids g3* $\longrightarrow$
$(\exists e.\ (g3 \vdash n_1 \simeq e) \wedge (g3 \vdash n_2 \simeq e) \wedge stamp\ g3\ n_1 = stamp\ g3\ n_2) \longrightarrow n_1 = n_2$
assume $n_1 \in$ *true-ids* g'
assume $n_2 \in$ *true-ids* g'
show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow stamp\ g'\ n_1 = stamp\ g'\ n_2 \Longrightarrow n_1 = n_2$
proof (*cases* $n_1 \in$ *true-ids g3*)
case *n1*: *True*
then show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow stamp\ g'\ n_1 = stamp\ g'\ n_2 \Longrightarrow n_1 = n_2$
proof (*cases* $n_2 \in$ *true-ids g3*)
case *n2*: *True*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume $stamp\ g'\ n_1 = stamp\ g'\ n_2$
have *n1rep*: $g3 \vdash n_1 \simeq e$
using *n1rep' kind-eq stamp-eq new-def add-preserves-rep*
by (*metis* (*no-types*, *lifting*) *BinaryNodeNew.hyps*(*1*) *BinaryNodeNew.hyps*(*3*) *BinaryNodeNew.prems*(*1*) *Diff-iff IRGraph.true-ids-def n1 unchanged unrep-preserves-closure*)
have *n2rep*: $g3 \vdash n_2 \simeq e$
using *n2rep' kind-eq stamp-eq new-def add-preserves-rep*
by (*metis BinaryNodeNew.hyps*(*1*) *BinaryNodeNew.hyps*(*3*) *BinaryNodeNew.prems*(*1*) *DiffE n2 true-ids unchanged unrep-preserves-closure*)
have $stamp\ g3\ n_1 = stamp\ g3\ n_2$
by (*metis BinaryNodeNew.hyps*(*7*) *BinaryNodeNew.hyps*(*8*) ‹$stamp\ g'\ n_1 = stamp\ g'\ n_2$› *fresh-ids fresh-node-subset n1rep n2rep subset-stamp*)
then show *?thesis* **using** *1*
using *n1 n2*
using *n1rep n2rep* **by** *blast*
next
case *n2*: *False*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume $stamp\ g'\ n_1 = stamp\ g'\ n_2$
have *new-n2*: $n_2 = n$
using ‹$n_2 \in$ *true-ids* g'› *dom n2* **by** *auto*
then have *ne*: $n_2 \notin$ *ids g3*
using *new n2*
using *BinaryNodeNew.hyps*(*7*) *fresh-ids* **by** *presburger*
have *unrep-bin*: $g3 \vdash n_1 \simeq$ *BinaryExpr op xe ye*
using *n1rep' kind-eq stamp-eq new-def add-preserves-rep*
by (*metis BinaryNodeNew.hyps*(*1*) *BinaryNodeNew.hyps*(*3*) *BinaryNode-*

New.prems(*1*) *DiffE* ‹*new* = *true-ids* g' − *true-ids* *g3*› *encodes-contains ids-some n1 n2rep'* *new-n2 repDet repn unchanged unrep-preserves-closure*)
have *rep*: (*g3* ⊢ *x* ≃ *xe*) ∧ (*g3* ⊢ *y* ≃ *ye*)
by (*meson BinaryNodeNew.hyps*(*1*) *BinaryNodeNew.hyps*(*3*) *term-graph-reconstruction unrep-contains unrep-unchanged*)
have *not-ref*: ¬(∃ n'. *kind g3* n_1 = *RefNode* n')
using *TreeToGraphThms.true-ids-def n1* **by** *force*
then have *kind g3* n_1 = *bin-node op x y*
using *unrep-bin binary-rep-kind rep* **by** *simp*
then show *?thesis* **using** *find-none BinaryNodeNew.hyps*(*6*)
by (*metis BinaryNodeNew.hyps*(*8*) ‹*stamp* g' *n* = s'› ‹*stamp* g' n_1 = *stamp* g' n_2› *fresh-node-subset ne new-n2 no-encoding subset-stamp unrep-bin*)
qed
next
case *n1*: *False*
then show $g' \vdash n_1 \simeq e \Longrightarrow g' \vdash n_2 \simeq e \Longrightarrow$ *stamp* g' n_1 = *stamp* g' n_2 $\Longrightarrow n_1 = n_2$
proof (*cases* n_2 ∈ *true-ids g3*)
case *n2*: *True*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stamp* g' n_1 = *stamp* g' n_2
have *new-n1*: $n_1 = n$
using ‹n_1 ∈ *true-ids* g'› *dom n1* **by** *auto*
then have *ne*: $n_1 \notin$ *ids g3*
using *new n1*
using *BinaryNodeNew.hyps*(*7*) *fresh-ids* **by** *blast*
have *unrep-bin*: *g3* ⊢ n_2 ≃ *BinaryExpr op xe ye*
using *n1rep'* *kind-eq stamp-eq new-def add-preserves-rep*
by (*metis* (*mono-tags*, *lifting*) *BinaryNodeNew.hyps*(*1*) *BinaryNodeNew.hyps*(*3*) *BinaryNodeNew.prems*(*1*) *Diff-iff IRGraph.true-ids-def n2 n2rep'* *new-n1 repDet repn unchanged unrep-preserves-closure*)
have *rep*: (*g3* ⊢ *x* ≃ *xe*) ∧ (*g3* ⊢ *y* ≃ *ye*)
using *BinaryNodeNew.hyps*(*1*) *BinaryNodeNew.hyps*(*3*) *term-graph-reconstruction unrep-contains unrep-unchanged* **by** *blast*
have *not-ref*: ¬(∃ n'. *kind g3* n_2 = *RefNode* n')
using *TreeToGraphThms.true-ids-def n2* **by** *fastforce*
then have *kind g3* n_2 = *bin-node op x y*
using *unrep-bin binary-rep-kind rep* **by** *simp*
then show *?thesis* **using** *find-none BinaryNodeNew.hyps*(*6*)
by (*metis BinaryNodeNew.hyps*(*8*) ‹*stamp* g' *n* = s'› ‹*stamp* g' n_1 = *stamp* g' n_2› *fresh-node-subset ne new-n1 no-encoding subset-stamp unrep-bin*)
next
case *n2*: *False*
assume *n1rep'*: $g' \vdash n_1 \simeq e$
assume *n2rep'*: $g' \vdash n_2 \simeq e$
assume *stamp* g' n_1 = *stamp* g' n_2
have $n_1 = n \wedge n_2 = n$
using ‹n_1 ∈ *true-ids* g'› *dom n1*

```
          using ‹n₂ ∈ true-ids g'› n2
          by simp
        then show ?thesis
          by simp
      qed
    qed
  qed
next
  case (AllLeafNodes g n s)
  then show ?case by blast
qed

end
```

8 Control-flow Semantics

```
theory IRStepObj
  imports
    TreeToGraph
begin
```

8.1 Object Heap

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

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

heapdef

```
type-synonym ('a, 'b) Heap = 'a ⇒ '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 = $((\lambda f.\ \lambda p.\ UndefVal),\ 0)$

8.2 Intraprocedural Semantics

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

fun *phi-list* :: $IRGraph \Rightarrow ID \Rightarrow ID\ list$ **where**
$phi\text{-}list\ g\ n =$
$(filter\ (\lambda x.(is\text{-}PhiNode\ (kind\ g\ x)))$
$(sorted\text{-}list\text{-}of\text{-}set\ (usages\ g\ n)))$

fun *input-index* :: $IRGraph \Rightarrow ID \Rightarrow ID \Rightarrow nat$ **where**
$input\text{-}index\ g\ n\ n' = find\text{-}index\ n'\ (inputs\text{-}of\ (kind\ g\ n))$

fun *phi-inputs* :: $IRGraph \Rightarrow nat \Rightarrow ID\ list \Rightarrow ID\ list$ **where**
$phi\text{-}inputs\ g\ i\ nodes = (map\ (\lambda n.\ (inputs\text{-}of\ (kind\ g\ n))!(i + 1))\ nodes)$

fun *set-phis* :: $ID\ list \Rightarrow Value\ list \Rightarrow MapState \Rightarrow MapState$ **where**
$set\text{-}phis\ []\ []\ m = m\ |$
$set\text{-}phis\ (n\ \#\ xs)\ (v\ \#\ vs)\ m = (set\text{-}phis\ xs\ vs\ (m(n := v)))\ |$
$set\text{-}phis\ []\ (v\ \#\ vs)\ m = m\ |$
$set\text{-}phis\ (x\ \#\ xs)\ []\ m = m$

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* :: $IRGraph \Rightarrow Params \Rightarrow (ID \times MapState \times FieldRefHeap) \Rightarrow (ID \times MapState \times FieldRefHeap) \Rightarrow bool$
$(\text{-},\ \text{-} \vdash \text{-} \rightarrow \text{-}\ 55)$ **for** $g\ p$ **where**

SequentialNode:
$[\![is\text{-}sequential\text{-}node\ (kind\ g\ nid);$
$nid' = (successors\text{-}of\ (kind\ g\ nid))!0]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)\ |$

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

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

$i = \textit{find-index nid (inputs-of (kind g merge))};$
$\textit{phis} = (\textit{phi-list g merge});$
$\textit{inps} = (\textit{phi-inputs g i phis});$
$g \vdash \textit{inps} \simeq_L \textit{inpsE};$
$[m,\ p] \vdash \textit{inpsE} \mapsto_L \textit{vs};$

$m' = \textit{set-phis phis vs m}]\!]$
$\Longrightarrow g,\ p \vdash (\textit{nid},\ m,\ h) \rightarrow (\textit{merge},\ m',\ h) \mid$

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

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

SignedDivNode:
$[\![\textit{kind g nid} = (\textit{SignedDivNode nid x y zero sb nxt});$
$g \vdash x \simeq \textit{xe};$
$g \vdash y \simeq \textit{ye};$
$[m,\ p] \vdash \textit{xe} \mapsto \textit{v1};$
$[m,\ p] \vdash \textit{ye} \mapsto \textit{v2};$
$v = (\textit{intval-div v1 v2});$
$m' = m(\textit{nid} := v)]\!]$
$\Longrightarrow g,\ p \vdash (\textit{nid},\ m,\ h) \rightarrow (\textit{nxt},\ m',\ h) \mid$

SignedRemNode:
$[\![\textit{kind g nid} = (\textit{SignedRemNode nid x y zero sb nxt});$
$g \vdash x \simeq \textit{xe};$
$g \vdash y \simeq \textit{ye};$
$[m,\ p] \vdash \textit{xe} \mapsto \textit{v1};$
$[m,\ p] \vdash \textit{ye} \mapsto \textit{v2};$
$v = (\textit{intval-mod v1 v2});$
$m' = m(\textit{nid} := v)]\!]$
$\Longrightarrow g,\ p \vdash (\textit{nid},\ m,\ h) \rightarrow (\textit{nxt},\ m',\ h) \mid$

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

StoreFieldNode:
$[\![$*kind g nid* = (*StoreFieldNode nid f newval* - (*Some obj*) *nid'*);
$g \vdash$ *newval* $\simeq$ *newvalE*;
$g \vdash$ *obj* $\simeq$ *objE*;
$[m, p] \vdash$ *newvalE* $\mapsto$ *val*;
$[m, p] \vdash$ *objE* $\mapsto$ *ObjRef ref*;
h' = *h-store-field f ref val h*;
m' = *m*(*nid* := *val*)$]\!]$
$\Longrightarrow g, p \vdash (nid, m, h) \to (nid', m', h')$ |

StaticStoreFieldNode:
$[\![$*kind g nid* = (*StoreFieldNode nid f newval* - *None nid'*);
$g \vdash$ *newval* $\simeq$ *newvalE*;
$[m, p] \vdash$ *newvalE* $\mapsto$ *val*;
h' = *h-store-field f None val h*;
m' = *m*(*nid* := *val*)$]\!]$
$\Longrightarrow g, p \vdash (nid, m, h) \to (nid', m', h')$

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

8.3 Interprocedural Semantics

type-synonym *Signature* = *string*
type-synonym *Program* = *Signature* $\rightharpoonup$ *IRGraph*

inductive *step-top* :: *Program* $\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState* $\times$ *Params*) *list* $\times$ *FieldRefHeap* $\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState* $\times$ *Params*) *list* $\times$ *FieldRefHeap* $\Rightarrow$ *bool*
(- $\vdash$ - $\longrightarrow$ - *55*)
for *P* **where**

Lift:
$[\![g, p \vdash (nid, m, h) \to (nid', m', h')]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#stk,\ h) \longrightarrow ((g,nid',m',p)\#stk,\ h')$ |

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*;
$g \vdash$ *arguments* $\simeq_L$ *argsE*;
$[m, p] \vdash$ *argsE* $\mapsto_L$ $p'$$]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#stk,\ h) \longrightarrow ((targetGraph,0,m',p')\#(g,nid,m,p)\#stk,\ h)$
|

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

  $cm' = cm(cnid := v)$;
  $cnid' = (successors\text{-}of\ (kind\ cg\ cnid))!0]\!]$
  $\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,cnid',cm',cp)\#stk,\ h) \mid$

ReturnNodeVoid:
$[\![$*kind g nid* = (*ReturnNode None* -);
  $cm' = cm(cnid := (ObjRef\ (Some\ (2048))))$;

  $cnid' = (successors\text{-}of\ (kind\ cg\ cnid))!0]\!]$
  $\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,cnid',cm',cp)\#stk,\ h) \mid$

UnwindNode:
$[\![$*kind g nid* = (*UnwindNode exception*);

  $g \vdash exception \simeq exceptionE$;
  $[m,\ p] \vdash exceptionE \mapsto e$;

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

  $cm' = cm(cnid := e)]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,exEdge,cm',cp)\#stk,\ h)$

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

8.4 Big-step Execution

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

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

inductive *exec* :: *Program*
    ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap*
    ⇒ *Trace*
    ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap*
    ⇒ *Trace*
    ⇒ *bool*
  ($- \vdash - \mid - \longrightarrow * - \mid -$)
  for *P*
  where
  $[\![P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h')$;
  $\neg(has\text{-}return\ m')$;

  $l' = (l\ @\ [(g,nid,m,p)])$;

$exec\ P\ (((g',nid',m',p')\#ys),h')\ l'\ next\text{-}state\ l''\rrbracket$
$\Longrightarrow exec\ P\ (((g,nid,m,p)\#xs),h)\ l\ next\text{-}state\ l''$

|
$\llbracket P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h');$
$has\text{-}return\ m';$

$l' = (l\ @\ [(g,nid,m,p)])\rrbracket$
$\Longrightarrow exec\ P\ (((g,nid,m,p)\#xs),h)\ l\ (((g',nid',m',p')\#ys),h')\ l'$
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$ *as Exec*) *exec* **.**

inductive *exec-debug* :: *Program*
$\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap$
$\Rightarrow nat$
$\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap$
$\Rightarrow bool$
(-⊢-→*-* -)
where
$\llbracket n > 0;$
$p \vdash s \longrightarrow s';$
$exec\text{-}debug\ p\ s'\ (n - 1)\ s''\rrbracket$
$\Longrightarrow exec\text{-}debug\ p\ s\ n\ s''$ |

$\llbracket n = 0\rrbracket$
$\Longrightarrow exec\text{-}debug\ p\ s\ n\ s$
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *exec-debug* **.**

8.4.1 Heap Testing

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

values $\{(prod.fst(prod.snd\ (prod.snd\ (hd\ (prod.fst\ res)))))\ 0$
$|\ res.\ (\lambda x\ .\ Some\ eg2\text{-}sq) \vdash ([(eg2\text{-}sq,0,new\text{-}map\text{-}state,p3),\ (eg2\text{-}sq,0,new\text{-}map\text{-}state,p3)],$
$new\text{-}heap) \rightarrow *2*\ res\}$

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*. (λx. *Some eg3-sq*) $\vdash$ ([(*eg3-sq*, *0*, *new-map-state*, *p3*), (*eg3-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) $\rightarrow * 3 *$ *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*.
(λx. *Some eg4-sq*) $\vdash$ ([(*eg4-sq*, *0*, *new-map-state*, *p3*), (*eg4-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) $\rightarrow * 3 *$ *res*}

end

8.5 Control-flow Semantics Theorems

theory *IRStepThms*
imports
IRStepObj
TreeToGraphThms
begin

We prove that within the same graph, a configuration triple will always transition to the same subsequent configuration. Therefore, our step semantics is deterministic.

8.5.1 Control-flow Step is Deterministic

theorem *stepDet*:
(g, $p \vdash$ (*nid*,*m*,*h*) $\rightarrow$ *next*) $\Longrightarrow$
($\forall$ *next'*. ((g, $p \vdash$ (*nid*,*m*,*h*) $\rightarrow$ *next'*) $\longrightarrow$ *next* = *next'*))
proof (*induction rule*: *step.induct*)
case (*SequentialNode nid next m h*)
have *notif*: $\neg$(*is-IfNode* (*kind g nid*))
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-IfNode-def*)
have *notend*: $\neg$(*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 IRNode.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*) *IRNode.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 StaticStoreFieldNode.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 $g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m,\ h)$
shows $\exists\ condE\ v.\ ((g \vdash cond \simeq condE) \wedge ([m,\ p] \vdash condE \mapsto v))$
using *assms*(*2*,*1*) **by** (*induct* (*nid*,*m*,*h*) (*nid'*,*m*,*h*) *rule*: *step.induct*; *auto*)

lemma *step-in-ids*:
assumes $g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')$
shows $nid \in ids\ g$
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

9 Proof Infrastructure

9.1 Bisimulation

theory *Bisimulation*
imports
Stuttering
begin

inductive *weak-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
(- . - ∼ -) **for** *nid* **where**
$[\![\forall P'.\ (g\ m\ p\ h \vdash nid \rightsquigarrow P') \longrightarrow (\exists\ Q'\ .\ (g'\ m\ p\ h \vdash nid \rightsquigarrow Q') \wedge P' = Q');$
$\forall\ Q'.\ (g'\ m\ p\ h \vdash nid \rightsquigarrow Q') \longrightarrow (\exists\ P'\ .\ (g\ m\ p\ h \vdash nid \rightsquigarrow P') \wedge P' = Q')]\!]$
$\Longrightarrow nid\ .\ g \sim g'$

A strong bisimilution between no-op transitions

inductive *strong-noop-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
(- | - ∼ -) **for** *nid* **where**
$[\![\forall P'.\ (g,\ p \vdash (nid,\ m,\ h) \to P') \longrightarrow (\exists\ Q'\ .\ (g',\ p \vdash (nid,\ m,\ h) \to Q') \wedge P' = Q');$
$\forall\ Q'.\ (g',\ p \vdash (nid,\ m,\ h) \to Q') \longrightarrow (\exists\ P'\ .\ (g,\ p \vdash (nid,\ m,\ h) \to P') \wedge P' = Q')]\!]$
$\Longrightarrow nid\ |\ g \sim g'$

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

lemma *no-step-bisimulation*:
  assumes $\forall\, m\ p\ h\ nid'\ m'\ h'.\ \neg(g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h'))$
  assumes $\forall\, m\ p\ h\ nid'\ m'\ h'.\ \neg(g',\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h'))$
  shows $nid \mid g \sim g'$
  using *assms*
  by (*simp add*: *assms*(*1*) *assms*(*2*) *strong-noop-bisimilar.intros*)

end

9.2 Graph Rewriting

theory
  Rewrites
imports
  Stuttering
begin

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

lemma *replace-usages-effect*:
  assumes $g' = replace\text{-}usages\ nid\ nid'\ g$
  shows $kind\ g'\ nid = RefNode\ nid'$
  using *assms replace-node-lookup replace-usages.simps*
  by (*metis IRNode.distinct*(*2755*))

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

lemma *replace-usages-unchanged*:
  assumes $nid \in ids\ g$
  assumes $g' = replace\text{-}usages\ nid\ nid'\ g$
  shows $unchanged\ (ids\ g - \{nid\})\ g\ g'$
  using *assms* **unfolding** *replace-usages.simps*
  using *assms*(*2*) *disjoint-change replace-usages-changeonly* **by** *presburger*

fun *nextNid* :: *IRGraph* ⇒ *ID* **where**
nextNid g = (*Max* (*ids g*)) + *1*

lemma *max-plus-one*:
fixes *c* :: *ID set*
shows ⟦*finite c*; *c* ≠ {}⟧ ⟹ (*Max c*) + *1* ∉ *c*
by (*meson Max-gr-iff less-add-one less-irrefl*)

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

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

fun *constantCondition* :: *bool* ⇒ *ID* ⇒ *IRNode* ⇒ *IRGraph* ⇒ *IRGraph* **where**
constantCondition val nid (*IfNode cond t f*) *g* =
replace-node nid (*IfNode* (*nextNid g*) *t f*, *stamp g nid*)
(*add-node* (*nextNid g*) ((*ConstantNode* (*bool-to-val val*)), *constantAsStamp* (*bool-to-val val*)) *g*) |
constantCondition cond nid - g = *g*

lemma *constantConditionTrue*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes *g′* = *constantCondition True ifcond* (*kind g ifcond*) *g*
shows *g′*, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
proof −
have *ifn*: ⋀ *c t f*. *IfNode c t f* ≠ *NoNode*
by *simp*
then have *if′*: *kind g′ ifcond* = *IfNode* (*nextNid g*) *t f*
using *assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *replace-node-lookup*
by *presburger*
have *truedef*: *bool-to-val True* = (*IntVal 32 1*)
by *auto*
from *ifn* **have** *ifcond* ≠ (*nextNid g*)
by (*metis assms*(*1*) *emptyE ids-some nextNidNotIn*)
moreover have ⋀ *c*. *ConstantNode c* ≠ *NoNode* **by** *simp*
ultimately have *kind g′* (*nextNid g*) = *ConstantNode* (*bool-to-val True*)
using *add-changed add-node-def assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *not-in-g other-node-unchanged replace-node-def replace-node-lookup singletonD*
by (*smt* (*z3*) *DiffI add-node-lookup replace-node-unchanged*)
then have *c′*: *kind g′* (*nextNid g*) = *ConstantNode* (*IntVal 32 1*)
using *truedef* **by** *simp*
have *valid-value* (*IntVal 32 1*) (*constantAsStamp* (*IntVal 32 1*))
unfolding *constantAsStamp.simps valid-value.simps*
using *nat-numeral* **by** *force*

then have $[g', m, p] \vdash$ *nextNid g* $\mapsto$ *IntVal 32 1*
using *ConstantExpr ConstantNode Value.distinct*(*1*) ‹*kind* g' (*nextNid g*) = *ConstantNode* (*bool-to-val True*)› *encodeeval-def truedef*
by *metis*
from *if'* *c'* **show** *?thesis* **using** *IfNode*
by (*metis* (*no-types*, *opaque-lifting*) *val-to-bool.simps*(*1*) ‹$[g',m,p] \vdash$ *nextNid g* $\mapsto$ *IntVal 32 1*› *encodeeval-def zero-neq-one*)
qed

lemma *constantConditionFalse*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes g' = *constantCondition False ifcond* (*kind g ifcond*) *g*
shows g', $p \vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*f*, *m*, *h*)
proof −
have *ifn*: $\bigwedge$ *c t f*. *IfNode c t f* $\neq$ *NoNode*
by *simp*
then have *if'*: *kind* g' *ifcond* = *IfNode* (*nextNid g*) *t f*
by (*metis assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *replace-node-lookup*)
have *falsedef*: *bool-to-val False* = (*IntVal 32 0*)
by *auto*
from *ifn* **have** *ifcond* $\neq$ (*nextNid g*)
by (*metis assms*(*1*) *equals0D ids-some nextNidNotIn*)
moreover have $\bigwedge$ *c*. *ConstantNode c* $\neq$ *NoNode* **by** *simp*
ultimately have *kind* g' (*nextNid g*) = *ConstantNode* (*bool-to-val False*)
by (*smt* (*z3*) *add-changed add-node-def assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *not-in-g other-node-unchanged replace-node-def replace-node-lookup singletonD*)
then have *c'*: *kind* g' (*nextNid g*) = *ConstantNode* (*IntVal 32 0*)
using *falsedef* **by** *simp*
have *valid-value* (*IntVal 32 0*) (*constantAsStamp* (*IntVal 32 0*))
unfolding *constantAsStamp.simps valid-value.simps*
using *nat-numeral* **by** *force*
then have $[g', m, p] \vdash$ *nextNid g* $\mapsto$ *IntVal 32 0*
by (*metis ConstantExpr ConstantNode* ‹*kind* g' (*nextNid g*) = *ConstantNode* (*bool-to-val False*)› *encodeeval-def falsedef*)
from *if'* *c'* **show** *?thesis* **using** *IfNode*
by (*metis* (*no-types*, *opaque-lifting*) *val-to-bool.simps*(*1*) ‹$[g',m,p] \vdash$ *nextNid g* $\mapsto$ *IntVal 32 0*› *encodeeval-def*)
qed

lemma *diff-forall*:
assumes $\forall n \in$ *ids g* − {*nid*}. *cond n*
shows $\forall n.\ n \in$ *ids g* $\wedge$ $n \notin$ {*nid*} $\longrightarrow$ *cond n*
by (*meson Diff-iff assms*)

lemma *replace-node-changeonly*:
assumes g' = *replace-node nid node g*
shows *changeonly* {*nid*} *g* g'
using *assms replace-node-unchanged*

unfolding *changeonly.simps* **using** *diff-forall*
by (*metis add-changed add-node-def changeonly.simps replace-node-def*)

lemma *add-node-changeonly*:
assumes $g' = $ *add-node nid node g*
shows *changeonly* {*nid*} *g* g'
by (*metis Rep-IRGraph-inverse add-node.rep-eq assms replace-node.rep-eq replace-node-changeonly*)

lemma *constantConditionNoEffect*:
assumes ¬(*is-IfNode* (*kind g nid*))
shows *g* = *constantCondition b nid* (*kind g nid*) *g*
using *assms* **apply** (*cases kind g nid*)
using *constantCondition.simps*
apply *presburger+*
apply (*metis is-IfNode-def*)
using *constantCondition.simps*
by *presburger+*

lemma *constantConditionIfNode*:
assumes *kind g nid* = *IfNode cond t f*
shows *constantCondition val nid* (*kind g nid*) *g* =
replace-node nid (*IfNode* (*nextNid g*) *t f*, *stamp g nid*)
(*add-node* (*nextNid g*) ((*ConstantNode* (*bool-to-val val*)), *constantAsStamp* (*bool-to-val val*)) *g*)
using *constantCondition.simps*
by (*simp add*: *assms*)

lemma *constantCondition-changeonly*:
assumes *nid* ∈ *ids g*
assumes $g' = $ *constantCondition b nid* (*kind g nid*) *g*
shows *changeonly* {*nid*} *g* g'
proof (*cases is-IfNode* (*kind g nid*))
case *True*
have *nextNid g* ∉ *ids g*
using *nextNidNotIn* **by** (*metis emptyE*)
then show *?thesis* **using** *assms*
using *replace-node-changeonly add-node-changeonly* **unfolding** *changeonly.simps*
using *True constantCondition.simps*(*1*) *is-IfNode-def*
by (*metis* (*no-types*, *lifting*) *insert-iff*)
next
case *False*
have $g = g'$
using *constantConditionNoEffect*
using *False assms*(*2*) **by** *blast*
then show *?thesis* **by** *simp*
qed

lemma *constantConditionNoIf*:
  assumes $\forall$ *cond t f. kind g ifcond* $\neq$ *IfNode cond t f*
  assumes $g' =$ *constantCondition val ifcond* (*kind g ifcond*) *g*
  shows $\exists$ *nid'* .(*g m p h* $\vdash$ *ifcond* $\rightsquigarrow$ *nid'*) $\longleftrightarrow$ (*g' m p h* $\vdash$ *ifcond* $\rightsquigarrow$ *nid'*)
proof –
  have $g' = g$
    using *assms*(*2*) *assms*(*1*)
    using *constantConditionNoEffect*
    by (*metis IRNode.collapse*(*11*))
  then show *?thesis* **by** *simp*
qed

lemma *constantConditionValid*:
  assumes *kind g ifcond* = *IfNode cond t f*
  assumes [*g*, *m*, *p*] $\vdash$ *cond* $\mapsto$ *v*
  assumes *const* = *val-to-bool v*
  assumes $g' =$ *constantCondition const ifcond* (*kind g ifcond*) *g*
  shows $\exists$ *nid'* .(*g m p h* $\vdash$ *ifcond* $\rightsquigarrow$ *nid'*) $\longleftrightarrow$ (*g' m p h* $\vdash$ *ifcond* $\rightsquigarrow$ *nid'*)
proof (*cases const*)
  case *True*
  have *ifstep*: *g*, *p* $\vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*t*, *m*, *h*)
    by (*meson IfNode True assms*(*1*) *assms*(*2*) *assms*(*3*) *encodeeval-def*)
  have *ifstep'*: *g'*, *p* $\vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*t*, *m*, *h*)
    using *constantConditionTrue*
    using *True assms*(*1*) *assms*(*4*) **by** *presburger*
  from *ifstep ifstep'* **show** *?thesis*
    using *StutterStep* **by** *blast*
next
  case *False*
  have *ifstep*: *g*, *p* $\vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*f*, *m*, *h*)
    by (*meson IfNode False assms*(*1*) *assms*(*2*) *assms*(*3*) *encodeeval-def*)
  have *ifstep'*: *g'*, *p* $\vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*f*, *m*, *h*)
    using *constantConditionFalse*
    using *False assms*(*1*) *assms*(*4*) **by** *presburger*
  from *ifstep ifstep'* **show** *?thesis*
    using *StutterStep* **by** *blast*
qed

end

9.3 Stuttering

theory *Stuttering*
  imports
    Semantics.IRStepThms
begin

inductive *stutter*:: *IRGraph* $\Rightarrow$ *MapState* $\Rightarrow$ *Params* $\Rightarrow$ *FieldRefHeap* $\Rightarrow$ *ID* $\Rightarrow$ *ID* $\Rightarrow$ *bool* (- - - - $\vdash$ - $\rightsquigarrow$ - *55*) for g m p h **where**

StutterStep:
$[\![g,\ p \vdash (nid,m,h) \rightarrow (nid',m,h)]\!]$
$\Longrightarrow g\ m\ p\ h \vdash nid \rightsquigarrow nid'$ |

Transitive:
$[\![g,\ p \vdash (nid,m,h) \rightarrow (nid'',m,h);$
$g\ m\ p\ h \vdash nid'' \rightsquigarrow nid']\!]$
$\Longrightarrow g\ m\ p\ h \vdash nid \rightsquigarrow nid'$

lemma *stuttering-successor*:
assumes $(g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h))$
shows $\{P'.\ (g\ m\ p\ h \vdash nid \rightsquigarrow P')\} = \{nid'\} \cup \{nid''.\ (g\ m\ p\ h \vdash nid' \rightsquigarrow nid'')\}$
proof –
have *nextin*: $nid' \in \{P'.\ (g\ m\ p\ h \vdash nid \rightsquigarrow P')\}$
using *assms StutterStep* **by** *blast*
have *nextsubset*: $\{nid''.\ (g\ m\ p\ h \vdash nid' \rightsquigarrow nid'')\} \subseteq \{P'.\ (g\ m\ p\ h \vdash nid \rightsquigarrow P')\}$
by (*metis Collect-mono assms stutter.Transitive*)
have $\forall\, n \in \{P'.\ (g\ m\ p\ h \vdash nid \rightsquigarrow P')\}\ .\ n = nid' \vee n \in \{nid''.\ (g\ m\ p\ h \vdash nid'$
$\rightsquigarrow nid'')\}$
using *stepDet*
by (*metis* (*no-types*, *lifting*) *Pair-inject assms mem-Collect-eq stutter.simps*)
then show *?thesis*
using *insert-absorb mk-disjoint-insert nextin nextsubset* **by** *auto*
qed

end

9.4 Evaluation Stamp Theorems

theory *StampEvalThms*
imports *Graph.ValueThms*
Semantics.IRTreeEvalThms
begin

lemma *unary-undef*: $val = UndefVal \Longrightarrow unary\text{-}eval\ op\ val = UndefVal$
by (*cases op*; *auto*)

lemma *unary-obj*: $val = ObjRef\ x \Longrightarrow unary\text{-}eval\ op\ val = UndefVal$
by (*cases op*; *auto*)

lemma *unrestricted-stamp-valid*:
assumes $s = unrestricted\text{-}stamp\ (IntegerStamp\ b\ lo\ hi)$

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

9.4.1 Support Lemmas for Integer Stamps and Associated IntVal values

Valid int implies some useful facts.

lemma *valid-int-gives*:
assumes *valid-value* (*IntVal b val*) *stamp*
obtains *lo hi* **where** *stamp* = *IntegerStamp b lo hi* ∧
valid-stamp (*IntegerStamp b lo hi*) ∧
take-bit b val = *val* ∧
lo ≤ *int-signed-value b val* ∧ *int-signed-value b val* ≤ *hi*
using *assms*
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* $\leq$ *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*)) $\leq$ *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*))

9.4.2 Validity of UnaryAbs

A set of lemmas for each evaltree step. Questions: 1. is this top-down approach (assume the result node evaluation) best? Maybe. It seems to be the shortest/simplest trigger?

lemma *unary-abs-implies-valid-value*:
assumes *1*:[*m*,*p*] ⊢ *e1* ↦ *r1*
assumes *2*:*result* = *unary-eval UnaryAbs r1*
assumes *3*:*result* ≠ *UndefVal*
assumes *4*:*valid-value r1* (*stamp-expr e1*)
shows *valid-value result* (*stamp-expr* (*UnaryExpr UnaryAbs e1*))
proof −
have [*m*,*p*] ⊢ (*UnaryExpr UnaryAbs e1*) ↦ *result*
using *assms* **by** *blast*
then obtain *b1 v1* **where** *r1*: *IntVal b1 v1* = *r1*
using *assms* **by** (*metis intval-abs.elims unary-eval.simps*(*1*))
then obtain *lo1 hi1* **where** *s1*: *stamp-expr e1* = *IntegerStamp b1 lo1 hi1*
by (*metis 4 valid-int-gives*)
then obtain *v2* **where** *r2*: *result* = *IntVal b1 v2*
using *assms* **by** (*metis intval-abs.simps*(*1*) *new-int.simps r1 unary-eval.simps*(*1*))
then have *r*: *result* = *intval-abs* (*IntVal b1 v1*)
using *2 r1 unary-eval.simps*(*1*) **by** *presburger*
then have *b1*: *0* < *b1* ∧ *b1* ≤ *64*
by (*metis 4 r1 s1 valid-stamp.simps*(*1*) *valid-value.simps*(*1*))
then have *bnds1*: *fst* (*bit-bounds b1*) ≤ *int-signed-value b1 v2* ∧ *int-signed-value b1 v2* ≤ *snd* (*bit-bounds b1*)

using *int-signed-value-bounds* **by** *blast*
then have *s*: (*stamp-expr* (*UnaryExpr UnaryAbs e1*)) = *unrestricted-stamp* (*IntegerStamp b1 lo1 hi1*)
by (*simp add*: *s1*)
then show *?thesis*
unfolding *s unrestricted-stamp.simps r2 valid-value.simps*
using *4 bnds1 r r1 r2 s1* **by** *auto*
qed

9.4.3 Validity of all Unary 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* $\neq$ *UndefVal*
assumes *op* $\notin$ *normal-unary*
shows $0 <$ (*ir-resultBits op*) $\wedge$ (*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*] $\vdash$ *expr* $\mapsto$ *val*
assumes *result* = *unary-eval op val*
assumes *op*: *op* $\notin$ *normal-unary*
assumes *result* $\neq$ *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*: $0 < (ir\text{-}resultBits\ op) \wedge (ir\text{-}resultBits\ op) \leq 64$
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

9.4.4 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*)

lemma *binary-eval-implies-valid-value*:
assumes [*m*,*p*] ⊢ *expr1* ↦ *val1*
assumes [*m*,*p*] ⊢ *expr2* ↦ *val2*
assumes *result* = *bin-eval op val1 val2*

assumes *result* $\neq$ *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*))
sorry

9.4.5 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* $\neq$ *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*) $\wedge$
take-bit b1 ival = *ival* $\wedge$
lo1 $\leq$ *int-signed-value b1 ival* $\wedge$ *int-signed-value b1 ival* $\leq$ *hi1*
using *assms* **by** (*metis valid-value.simps*(*1*))
then have *mm*: *min lo1 lo2* $\leq$ *int-signed-value b1 ival* $\wedge$ *int-signed-value b1 ival* $\leq$ *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*

9.4.6 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

9.4.7 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.

experiment begin

lemma *stamp-implies-valid-value*:
  assumes $[m,p] \vdash$ *expr* $\mapsto$ *val*
  shows *valid-value val* (*stamp-expr expr*)
  using *assms* **proof** (*induction expr val*)
  case (*UnaryExpr expr val result op*)
  then show *?case* **using** *eval-unary-implies-valid-value* **by** *simp*
  next
    case (*BinaryExpr expr1 val1 expr2 val2 result op*)
    then show *?case* **using** *binary-eval-implies-valid-value* **by** *simp*
  next
    case (*ConditionalExpr cond condv expr expr1 expr2 val*)
    have *compatible* (*stamp-expr expr1*) (*stamp-expr expr2*)
      using *assms* **sorry**
    then show *?case*
      using *assms conditional-eval-implies-valid-value*
     using *ConditionalExpr.IH*(*1*) *ConditionalExpr.IH*(*2*) *ConditionalExpr.hyps*(*1*)
ConditionalExpr.hyps(*2*) *ConditionalExpr.hyps*(*3*) *ConditionalExpr.hyps*(*4*) **by** *blast*
  next
    case (*ParameterExpr x1 x2*)
    then show *?case* **by** *auto*
  next
    case (*LeafExpr x1 x2*)
    then show *?case* **by** *auto*
  next
    case (*ConstantExpr x*)
    then show *?case* **by** *auto*
qed

lemma *value-range*:
  assumes $[m, p] \vdash e \mapsto v$
  shows $v \in \{$*val* . *valid-value val* (*stamp-expr e*)$\}$
  using *assms* **sorry**
end

lemma *stamp-under-semantics*:
  assumes *stamp-under* (*stamp-expr x*) (*stamp-expr y*)
  assumes $[m, p] \vdash$ (*BinaryExpr BinIntegerLessThan x y*) $\mapsto v$
  assumes *xvalid*: ($\forall$ *m p v*. ($[m, p] \vdash x \mapsto v$) $\longrightarrow$ *valid-value v* (*stamp-expr x*))
  assumes *yvalid*: ($\forall$ *m p v*. ($[m, p] \vdash y \mapsto v$) $\longrightarrow$ *valid-value v* (*stamp-expr y*))
  shows *val-to-bool v*
  sorry

lemma *stamp-under-semantics-inversed*:
  assumes *stamp-under* (*stamp-expr y*) (*stamp-expr x*)
  assumes $[m, p] \vdash$ (*BinaryExpr BinIntegerLessThan x y*) $\mapsto v$
  assumes *xvalid*: ($\forall$ *m p v*. ($[m, p] \vdash x \mapsto v$) $\longrightarrow$ *valid-value v* (*stamp-expr x*))
  assumes *yvalid*: ($\forall$ *m p v*. ($[m, p] \vdash y \mapsto v$) $\longrightarrow$ *valid-value v* (*stamp-expr y*))
  shows $\neg$(*val-to-bool v*)

sorry

end

10 Optization DSLs

theory *Markup*
imports *Semantics.IRTreeEval Snippets.Snipping*
begin

datatype *'a Rewrite* =
Transform 'a 'a (- $\longmapsto$ - *10*) |
Conditional 'a 'a bool (- $\longmapsto$ - *when* - *70*) |
Sequential 'a Rewrite 'a Rewrite |
Transitive 'a Rewrite

datatype *'a ExtraNotation* =
ConditionalNotation 'a 'a 'a (- *?* - : -) |
EqualsNotation 'a 'a (- *eq* -) |
ConstantNotation 'a (*const* - *120*) |
TrueNotation (*true*) |
FalseNotation (*false*) |
ExclusiveOr 'a 'a (- $\oplus$ -) |
LogicNegationNotation 'a (!-) |
ShortCircuitOr 'a 'a (- || -)

definition *word* :: (*'a::len*) *word* $\Rightarrow$ *'a word* **where**
word x = *x*

ML-file ‹*markup.ML*›

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 BinShortCircuitOr*}
| *markup DSL-Tokens.Abs* = @{*term UnaryExpr*} $ @{*term UnaryAbs*}
| *markup DSL-Tokens.Less* = @{*term BinaryExpr*} $ @{*term BinIntegerLessThan*}
| *markup DSL-Tokens.Equals* = @{*term BinaryExpr*} $ @{*term BinIntegerEquals*}
| *markup DSL-Tokens.Not* = @{*term UnaryExpr*} $ @{*term UnaryNot*}
| *markup DSL-Tokens.Negate* = @{*term UnaryExpr*} $ @{*term UnaryNeg*}
| *markup DSL-Tokens.LogicNegate* = @{*term UnaryExpr*} $ @{*term UnaryLogicNegation*}

| *markup DSL-Tokens.LeftShift* = @{*term BinaryExpr*} $ @{*term BinLeftShift*}
| *markup DSL-Tokens.RightShift* = @{*term BinaryExpr*} $ @{*term BinRightShift*}
| *markup DSL-Tokens.UnsignedRightShift* = @{*term BinaryExpr*} $ @{*term 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 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 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 IRExprMarkup = *DSL-Markup*(*IRExprTranslator*);
structure IntValMarkup = *DSL-Markup*(*IntValTranslator*);
structure WordMarkup = *DSL-Markup*(*WordTranslator*);
›

ir expression translation

syntax *-expandExpr* :: *term* ⇒ *term* (*exp*[-])
parse-translation ‹ [(@{*syntax-const -expandExpr*} , *IRExprMarkup.markup-expr* [])] ›

value expression translation

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

word expression translation

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

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

value *exp*[((e_1 − e_2) + (*const* (*IntVal 32 0*)) + e_2) ⟼ e_1 *when True*] *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
theory *Phase*
  imports *Main*
begin

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

end

10.1 Canonicalization DSL

theory *Canonicalization*
  imports
    Markup
    Phase
    HOL−Eisbach.Eisbach
  keywords
    phase :: *thy-decl* **and**
    terminating :: *quasi-command* **and**
    print-phases :: *diag* **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)));
›
```

ML-file *rewrites.ML*

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*

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*

fun *size* :: *IRExpr* ⇒ *nat* **where**
size (*UnaryExpr op e*) = (*size e*) + *1* |
size (*BinaryExpr BinAdd x y*) = (*size x*) + ((*size y*) ∗ *2*) |
size (*BinaryExpr op x y*) = (*size x*) + (*size y*) |
size (*ConditionalExpr cond t f*) = (*size cond*) + (*size t*) + (*size f*) + *2* |
size (*ConstantExpr c*) = *1* |
size (*ParameterExpr ind s*) = *2* |
size (*LeafExpr nid s*) = *2* |
size (*ConstantVar c*) = *2* |
size (*VariableExpr x s*) = *2*

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 del*: *le-expr-def*)*?*
| (*unfold size.simps*)*?*

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

11 Canonicalization Phase

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

lemma *size-pos*[*simp*]: *0* < *size y*
  apply (*induction y*; *auto?*)
  subgoal premises *prems* **for** *op a b*
    using *prems* **by** (*induction op*; *auto*)
  done

lemma *size-non-add*: *op* ≠ *BinAdd* ⟹ *size* (*BinaryExpr op a b*) = *size a* + *size b*
  by (*induction op*; *auto*)

lemma *size-non-const*:
  ¬ *is-ConstantExpr y* ⟹ *1* < *size y*
  using *size-pos* **apply** (*induction y*; *auto*)
  subgoal premises *prems* **for** *op a b*
    apply (*cases op* = *BinAdd*)
    using *size-non-add size-pos* **apply** *auto*
    by (*simp add*: *Suc-lessI one-is-add*)+
  done

definition *well-formed-equal* :: *Value* ⇒ *Value* ⇒ *bool*
  (**infix** ≈ *50*) **where**
  well-formed-equal v_1 v_2 = (v_1 ≠ *UndefVal* ⟶ v_1 = v_2)

lemma *well-formed-equal-defn* [*simp*]:
  well-formed-equal v_1 v_2 = (v_1 ≠ *UndefVal* ⟶ v_1 = v_2)

unfolding *well-formed-equal-def* **by** *simp*

end

11.1 Conditional Expression

theory *ConditionalPhase*
imports
Common
begin

phase *Conditional*
terminating *size*
begin

lemma *negates*: *is-IntVal e* $\Longrightarrow$ *val-to-bool* (*val*[*e*]) $\equiv$ $\neg$(*val-to-bool* (*val*[!*e*]))
using *intval-logic-negation.simps* **unfolding** *logic-negate-def*
sorry

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*)

optimization *negate-condition*: ((!*e*) *?* *x* : *y*) $\longmapsto$ (*e* *?* *y* : *x*)
apply *simp* **using** *negation-condition-intval*
by (*smt* (*verit*, *ccfv-SIG*) *ConditionalExpr ConditionalExprE Value.collapse Value.exhaust-disc evaltree-not-undef intval-logic-negation.simps*(*4*) *intval-logic-negation.simps negates unary-eval.simps*(*4*) *unfold-unary*)

definition *wff-stamps* :: *bool* **where**
wff-stamps = ($\forall$ *m p expr val* . ([*m*,*p*] $\vdash$ *expr* $\mapsto$ *val*) $\longrightarrow$ *valid-value val* (*stamp-expr expr*))

definition *wf-stamp* :: *IRExpr* $\Rightarrow$ *bool* **where**
wf-stamp e = ($\forall$ *m p v*. ([*m*, *p*] $\vdash$ *e* $\mapsto$ *v*) $\longrightarrow$ *valid-value v* (*stamp-expr e*))

optimization *b*[*intval*]: ((*x eq y*) *?* *x* : *y*) $\longmapsto$ *y*

sorry

lemma *val-optimise-integer-test*:
assumes *is-IntVal32 x*
shows *intval-conditional* (*intval-equals val*[(*x* & (*IntVal32 1*))] (*IntVal32 0*))
(*IntVal32 0*) (*IntVal32 1*) =
val[*x* & *IntVal32 1*]
apply *simp-all*
apply *auto*
using *bool-to-val.elims intval-equals.elims val-to-bool.simps*(*1*) *val-to-bool.simps*(*3*)
sorry

optimization *val-conditional-eliminate-known-less*: ((*x* < *y*) *?* *x* : *y*) ⟼ *x*
when (*stamp-under* (*stamp-expr x*) (*stamp-expr y*)
∧ *wf-stamp x* ∧ *wf-stamp y*)
apply *auto*
using *stamp-under.simps wf-stamp-def val-to-bool.simps*
sorry

optimization *opt-conditional-eq-is-RHS*: ((*BinaryExpr BinIntegerEquals x y*) *?* *x* : *y*) ⟼ *y*
apply *simp-all* **apply** *auto* **using** *b Canonicalization.intval.simps*(*1*) *evalDet intval-conditional.simps*
by (*metis* (*mono-tags*, *lifting*) *evaltree-not-undef*)

optimization *opt-normalize-x*: ((*x 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*)))
done

optimization *opt-normalize-x2*: ((*x eq* (*const* (*IntVal 32 1*))) *?*
(*const* (*IntVal 32 1*)) : (*const* (*IntVal 32 0*))) ⟼ *x*
when (*x* = *ConstantExpr* (*IntVal 32 0*) | (*x* = *ConstantExpr* (*IntVal 32 1*)))
done

optimization *opt-flip-x*: ((*x eq* (*const* (*IntVal 32 0*))) *?*
(*const* (*IntVal 32 1*)) : (*const* (*IntVal 32 0*))) ⟼
x ⊕ (*const* (*IntVal 32 1*))
when (*x* = *ConstantExpr* (*IntVal 32 0*) | (*x* = *ConstantExpr*

(*IntVal 32 1*)))
  done

optimization *opt-flip-x2*: ((*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*) | (*x* = *ConstantExpr*
(*IntVal 32 1*)))
  done

optimization *opt-optimise-integer-test*:
    (((*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*)
  apply *simp-all*
  apply *auto*
  using *val-optimise-integer-test* **sorry**

optimization *opt-optimise-integer-test-2*:
    (((*x* & (*const* (*IntVal 32 1*))) *eq* (*const* (*IntVal 32 0*))) *?*
    (*const* (*IntVal 32 0*)) : (*const* (*IntVal 32 1*))) $\longmapsto$
    x
    when (*x* = *ConstantExpr* (*IntVal 32 0*) | (*x* = *ConstantExpr* (*IntVal*
32 1)))
  done

optimization *opt-conditional-eliminate-known-less*: ((*x* < *y*) *?* *x* : *y*) $\longmapsto$ *x*
    when (((*stamp-under* (*stamp-expr x*) (*stamp-expr y*)) |
    ((*stpi-upper* (*stamp-expr x*)) = (*stpi-lower* (*stamp-expr*
y))))
    $\wedge$ *wf-stamp x* $\wedge$ *wf-stamp y*)
  unfolding *le-expr-def* **apply** *auto*
  using *stamp-under.simps wf-stamp-def val-conditional-eliminate-known-less*
  sorry

end

end

12 Conditional Elimination Phase

theory *ConditionalElimination*
  imports
    Proofs.Rewrites
    Proofs.Bisimulation
begin

12.1 Individual Elimination Rules

We introduce a TriState as in the Graal compiler to represent when static analysis can tell us information about the value of a boolean expression. Unknown = No information can be inferred KnownTrue/KnownFalse = We can infer the expression will always be true or false.

datatype *TriState* = *Unknown* | *KnownTrue* | *KnownFalse*

The implies relation corresponds to the LogicNode.implies method from the compiler which attempts to infer when one logic nodes value can be inferred from a known logic node.

inductive *implies* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *TriState* ⇒ *bool*
  (- ⊢ - & - ↪ -) **for** *g* **where**
  eq-imp-less:
  g ⊢ (*IntegerEqualsNode x y*) & (*IntegerLessThanNode x y*) ↪ *KnownFalse* |
  eq-imp-less-rev:
  g ⊢ (*IntegerEqualsNode x y*) & (*IntegerLessThanNode y x*) ↪ *KnownFalse* |
  less-imp-rev-less:
  g ⊢ (*IntegerLessThanNode x y*) & (*IntegerLessThanNode y x*) ↪ *KnownFalse* |
  less-imp-not-eq:
  g ⊢ (*IntegerLessThanNode x y*) & (*IntegerEqualsNode x y*) ↪ *KnownFalse* |
  less-imp-not-eq-rev:
  g ⊢ (*IntegerLessThanNode x y*) & (*IntegerEqualsNode y x*) ↪ *KnownFalse* |

  x-imp-x:
  g ⊢ *x* & *x* ↪ *KnownTrue* |

  negate-false:
  ⟦*g* ⊢ *x* & (*kind g y*) ↪ *KnownTrue*⟧ ⟹ *g* ⊢ *x* & (*LogicNegationNode y*) ↪ *KnownFalse* |
  negate-true:
  ⟦*g* ⊢ *x* & (*kind g y*) ↪ *KnownFalse*⟧ ⟹ *g* ⊢ *x* & (*LogicNegationNode y*) ↪ *KnownTrue*

Total relation over partial implies relation

inductive *condition-implies* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *TriState* ⇒ *bool*
  (- ⊢ - & - ⇀ -) **for** *g* **where**
  ⟦¬(*g* ⊢ *a* & *b* ↪ *imp*)⟧ ⟹ (*g* ⊢ *a* & *b* ⇀ *Unknown*) |
  ⟦(*g* ⊢ *a* & *b* ↪ *imp*)⟧ ⟹ (*g* ⊢ *a* & *b* ⇀ *imp*)

inductive *implies-tree* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* ⇒ *bool*
(- & - ↪ -) **where**
eq-imp-less:
(*BinaryExpr BinIntegerEquals x y*) & (*BinaryExpr BinIntegerLessThan x y*) ↪ *False* |
eq-imp-less-rev:
(*BinaryExpr BinIntegerEquals x y*) & (*BinaryExpr BinIntegerLessThan y x*) ↪ *False* |
less-imp-rev-less:
(*BinaryExpr BinIntegerLessThan x y*) & (*BinaryExpr BinIntegerLessThan y x*) ↪ *False* |
less-imp-not-eq:
(*BinaryExpr BinIntegerLessThan x y*) & (*BinaryExpr BinIntegerEquals x y*) ↪ *False* |
less-imp-not-eq-rev:
(*BinaryExpr BinIntegerLessThan x y*) & (*BinaryExpr BinIntegerEquals y x*) ↪ *False* |

x-imp-x:
x & *x* ↪ *True* |

negate-false:
⟦*x* & *y* ↪ *True*⟧ ⟹ *x* & (*UnaryExpr UnaryLogicNegation y*) ↪ *False* |
negate-true:
⟦*x* & *y* ↪ *False*⟧ ⟹ *x* & (*UnaryExpr UnaryLogicNegation y*) ↪ *True*

Proofs that the implies relation is correct with respect to the existing evaluation semantics.

experiment begin
lemma *logic-negate-type*:
assumes [*m*, *p*] ⊢ *UnaryExpr UnaryLogicNegation x* ↦ *v*
assumes *v* ≠ *UndefVal*
shows ∃ *v2*. [*m*, *p*] ⊢ *x* ↦ *IntVal32 v2*
proof −
obtain *ve* **where** *ve*: [*m*, *p*] ⊢ *x* ↦ *ve*
using *assms*(*1*) **by** *blast*
then have [*m*, *p*] ⊢ *UnaryExpr UnaryLogicNegation x* ↦ *unary-eval UnaryLogicNegation ve*
by (*metis UnaryExprE assms*(*1*) *evalDet*)
then show *?thesis* **using** *assms unary-eval.elims evalDet ve IRUnaryOp.distinct*
sorry
qed

lemma *logic-negation-relation-tree*:
assumes [*m*, *p*] ⊢ *y* ↦ *val*
assumes [*m*, *p*] ⊢ *UnaryExpr UnaryLogicNegation y* ↦ *invval*

```
  assumes invval ≠ UndefVal
  shows val-to-bool val ⟷ ¬(val-to-bool invval)
proof –
  obtain v where invval = unary-eval UnaryLogicNegation v
    using assms(2) by blast
  then have [m, p] ⊢ y ↦ v using UnaryExprE assms(1,2) sorry
  then show ?thesis sorry
  qed

lemma logic-negation-relation:
  assumes [g, m, p] ⊢ y ↦ val
  assumes kind g neg = LogicNegationNode y
  assumes [g, m, p] ⊢ neg ↦ invval
  assumes invval ≠ UndefVal
  shows val-to-bool val ⟷ ¬(val-to-bool invval)
proof –
  obtain yencode where 5: g ⊢ y ≃ yencode
    using assms(1) encodeeval-def by auto
  then have 6: g ⊢ neg ≃ UnaryExpr UnaryLogicNegation yencode
    using rep.intros(7) assms(2) by simp
  then have 7: [m, p] ⊢ UnaryExpr UnaryLogicNegation yencode ↦ invval
    using assms(3) encodeeval-def
    by (metis repDet)
  obtain v1 where v1: [g, m, p] ⊢ y ↦ IntVal 32 v1
    using assms(1,2,3,4) using logic-negate-type sorry
  have invval = bool-to-val (¬(val-to-bool val))
    using assms(1,2,3) evalDet unary-eval.simps(4)
    sorry
  have val-to-bool invval ⟷ ¬(val-to-bool val)
    using ‹invval = bool-to-val (¬ val-to-bool val)› by force
  then show ?thesis
    by simp
qed
end

lemma implies-valid:
  assumes x & y ↪ imp
  assumes [m, p] ⊢ x ↦ v1
  assumes [m, p] ⊢ y ↦ v2
  assumes v1 ≠ UndefVal ∧ v2 ≠ UndefVal
  shows (imp ⟶ (val-to-bool v1 ⟶ val-to-bool v2)) ∧
        (¬imp ⟶ (val-to-bool v1 ⟶ ¬(val-to-bool v2)))
  (is (?TP ⟶ ?TC) ∧ (?FP ⟶ ?FC))
  apply (intro conjI; rule impI)
proof –
  assume KnownTrue: ?TP
  show ?TC
 using assms(1) KnownTrue assms(2–) proof (induct x y imp rule: implies-tree.induct)
    case (eq-imp-less x y)
```

then show *?case* **by** *simp*
next
case (*eq-imp-less-rev x y*)
then show *?case* **by** *simp*
next
case (*less-imp-rev-less x y*)
then show *?case* **by** *simp*
next
case (*less-imp-not-eq x y*)
then show *?case* **by** *simp*
next
case (*less-imp-not-eq-rev x y*)
then show *?case* **by** *simp*
next
case (*x-imp-x*)
then show *?case*
by (*metis evalDet*)
next
case (*negate-false x1*)
then show *?case* **using** *evalDet*
using *assms*(*2*,*3*) **by** *blast*
next
case (*negate-true y*)
then show *?case*
sorry
qed
next
assume *KnownFalse*: *?FP*
show *?FC* **using** *assms KnownFalse* **proof** (*induct x y imp rule*: *implies-tree.induct*)
case (*eq-imp-less x y*)
obtain *xval* **where** *xval*: $[m, p] \vdash x \mapsto xval$
using *eq-imp-less*(*1*) *eq-imp-less.prems*(*3*)
by *blast*
then obtain *yval* **where** *yval*: $[m, p] \vdash y \mapsto yval$
using *eq-imp-less.prems*(*3*)
using *eq-imp-less.prems*(*2*) **by** *blast*
have *eqeval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerEquals x y*) $\mapsto$ *intval-equals xval yval*
using *xval yval evaltree.BinaryExpr*
by (*metis BinaryExprE bin-eval.simps*(*11*) *eq-imp-less.prems*(*1*) *evalDet*)
have *lesseval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerLessThan x y*) $\mapsto$ *intval-less-than xval yval*
using *xval yval evaltree.BinaryExpr*
by (*metis BinaryExprE bin-eval.simps*(*12*) *eq-imp-less.prems*(*2*) *evalDet*)
have *val-to-bool* (*intval-equals xval yval*) $\longrightarrow \neg$(*val-to-bool* (*intval-less-than xval yval*))
using *assms*(*4*) **apply** (*cases xval*; *cases yval*; *auto*) **sorry**

then show *?case*

using *eqeval lesseval*
by (*metis eq-imp-less.prems*(*1*) *eq-imp-less.prems*(*2*) *evalDet*)
next
case (*eq-imp-less-rev x y*)
obtain *xval* **where** *xval*: $[m, p] \vdash x \mapsto xval$
using *eq-imp-less-rev.prems*(*3*)
using *eq-imp-less-rev.prems*(*2*) **by** *blast*
obtain *yval* **where** *yval*: $[m, p] \vdash y \mapsto yval$
using *eq-imp-less-rev.prems*(*3*)
using *eq-imp-less-rev.prems*(*2*) **by** *blast*
have *eqeval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerEquals x y*) $\mapsto$ *intval-equals xval yval*
using *xval yval evaltree.BinaryExpr*
by (*metis BinaryExprE bin-eval.simps*(*11*) *eq-imp-less-rev.prems*(*1*) *evalDet*)
have *lesseval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerLessThan y x*) $\mapsto$ *intval-less-than yval xval*
using *xval yval evaltree.BinaryExpr*
by (*metis BinaryExprE bin-eval.simps*(*12*) *eq-imp-less-rev.prems*(*2*) *evalDet*)
have *val-to-bool* (*intval-equals xval yval*) $\longrightarrow \neg$(*val-to-bool* (*intval-less-than yval xval*))
using *assms*(*4*) **apply** (*cases xval*; *cases yval*; *auto*) **sorry**

then show *?case*
using *eqeval lesseval*
by (*metis eq-imp-less-rev.prems*(*1*) *eq-imp-less-rev.prems*(*2*) *evalDet*)
next
case (*less-imp-rev-less x y*)
obtain *xval* **where** *xval*: $[m, p] \vdash x \mapsto xval$
using *less-imp-rev-less.prems*(*3*)
using *less-imp-rev-less.prems*(*2*) **by** *blast*
obtain *yval* **where** *yval*: $[m, p] \vdash y \mapsto yval$
using *less-imp-rev-less.prems*(*3*)
using *less-imp-rev-less.prems*(*2*) **by** *blast*
have *lesseval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerLessThan x y*) $\mapsto$ *intval-less-than xval yval*
using *xval yval evaltree.BinaryExpr*
by (*metis BinaryExprE bin-eval.simps*(*12*) *evalDet less-imp-rev-less.prems*(*1*))
have *revlesseval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerLessThan y x*) $\mapsto$ *intval-less-than yval xval*
using *xval yval evaltree.BinaryExpr*
by (*metis BinaryExprE bin-eval.simps*(*12*) *evalDet less-imp-rev-less.prems*(*2*))
have *val-to-bool* (*intval-less-than xval yval*) $\longrightarrow \neg$(*val-to-bool* (*intval-less-than yval xval*))
using *assms*(*4*) **apply** (*cases xval*; *cases yval*; *auto*) **sorry**

then show *?case*
by (*metis evalDet less-imp-rev-less.prems*(*1*) *less-imp-rev-less.prems*(*2*) *lesseval revlesseval*)
next

case (*less-imp-not-eq x y*)
obtain *xval* **where** *xval*: $[m, p] \vdash x \mapsto xval$
using *less-imp-not-eq.prems*(*3*)
using *less-imp-not-eq.prems*(*1*) **by** *blast*
obtain *yval* **where** *yval*: $[m, p] \vdash y \mapsto yval$
using *less-imp-not-eq.prems*(*3*)
using *less-imp-not-eq.prems*(*1*) **by** *blast*
have *eqeval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerEquals x y*) $\mapsto$ *intval-equals xval yval*
using *xval yval evaltree.BinaryExpr*
by (*metis BinaryExprE bin-eval.simps*(*11*) *evalDet less-imp-not-eq.prems*(*2*))
have *lesseval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerLessThan x y*) $\mapsto$ *intval-less-than xval yval*
using *xval yval evaltree.BinaryExpr*
by (*metis BinaryExprE bin-eval.simps*(*12*) *evalDet less-imp-not-eq.prems*(*1*))
have *val-to-bool* (*intval-less-than xval yval*) $\longrightarrow \neg$(*val-to-bool* (*intval-equals xval yval*))
using *assms*(*4*) **apply** (*cases xval*; *cases yval*; *auto*) **sorry**

then show *?case*
by (*metis eqeval evalDet less-imp-not-eq.prems*(*1*) *less-imp-not-eq.prems*(*2*) *lesseval*)
next
case (*less-imp-not-eq-rev x y*)
obtain *xval* **where** *xval*: $[m, p] \vdash x \mapsto xval$
using *less-imp-not-eq-rev.prems*(*3*)
using *less-imp-not-eq-rev.prems*(*1*) **by** *blast*
obtain *yval* **where** *yval*: $[m, p] \vdash y \mapsto yval$
using *less-imp-not-eq-rev.prems*(*3*)
using *less-imp-not-eq-rev.prems*(*1*) **by** *blast*
have *eqeval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerEquals y x*) $\mapsto$ *intval-equals yval xval*
using *xval yval evaltree.BinaryExpr*
by (*metis BinaryExprE bin-eval.simps*(*11*) *evalDet less-imp-not-eq-rev.prems*(*2*))
have *lesseval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerLessThan x y*) $\mapsto$ *intval-less-than xval yval*
using *xval yval evaltree.BinaryExpr*
by (*metis BinaryExprE bin-eval.simps*(*12*) *evalDet less-imp-not-eq-rev.prems*(*1*))
have *val-to-bool* (*intval-less-than xval yval*) $\longrightarrow \neg$(*val-to-bool* (*intval-equals yval xval*))
using *assms*(*4*) **apply** (*cases xval*; *cases yval*; *auto*) **sorry**

then show *?case*
by (*metis eqeval evalDet less-imp-not-eq-rev.prems*(*1*) *less-imp-not-eq-rev.prems*(*2*) *lesseval*)
next
case (*x-imp-x x1*)
then show *?case* **by** *simp*
next

case (*negate-false x y*)
then show *?case* **sorry**
next
case (*negate-true x1*)
then show *?case* **by** *simp*
qed
qed

lemma *implies-true-valid*:
assumes $x \;\&\; y \hookrightarrow imp$
assumes *imp*
assumes $[m, p] \vdash x \mapsto v1$
assumes $[m, p] \vdash y \mapsto v2$
assumes $v1 \neq UndefVal \wedge v2 \neq UndefVal$
shows $val\text{-}to\text{-}bool\ v1 \longrightarrow val\text{-}to\text{-}bool\ v2$
using *assms implies-valid*
by *blast*

lemma *implies-false-valid*:
assumes $x \;\&\; y \hookrightarrow imp$
assumes $\neg imp$
assumes $[m, p] \vdash x \mapsto v1$
assumes $[m, p] \vdash y \mapsto v2$
assumes $v1 \neq UndefVal \wedge v2 \neq UndefVal$
shows $val\text{-}to\text{-}bool\ v1 \longrightarrow \neg(val\text{-}to\text{-}bool\ v2)$
using *assms implies-valid* **by** *blast*

The following relation corresponds to the UnaryOpLogicNode.tryFold and BinaryOpLogicNode.tryFold methods and their associated concrete implementations.

The relation determines if a logic operation can be shown true or false through the stamp typing information.

inductive *tryFold* :: $IRNode \Rightarrow (ID \Rightarrow Stamp) \Rightarrow bool \Rightarrow bool$
where
$[\![alwaysDistinct\ (stamps\ x)\ (stamps\ y)]\!]$
$\Longrightarrow tryFold\ (IntegerEqualsNode\ x\ y)\ stamps\ False\ |$
$[\![neverDistinct\ (stamps\ x)\ (stamps\ y)]\!]$
$\Longrightarrow tryFold\ (IntegerEqualsNode\ x\ y)\ stamps\ True\ |$
$[\![is\text{-}IntegerStamp\ (stamps\ x);$
$is\text{-}IntegerStamp\ (stamps\ y);$
$stpi\text{-}upper\ (stamps\ x) < stpi\text{-}lower\ (stamps\ y)]\!]$
$\Longrightarrow tryFold\ (IntegerLessThanNode\ x\ y)\ stamps\ True\ |$
$[\![is\text{-}IntegerStamp\ (stamps\ x);$
$is\text{-}IntegerStamp\ (stamps\ y);$
$stpi\text{-}lower\ (stamps\ x) \geq stpi\text{-}upper\ (stamps\ y)]\!]$
$\Longrightarrow tryFold\ (IntegerLessThanNode\ x\ y)\ stamps\ False$

Proofs that show that when the stamp lookup function is well-formed, the tryFold relation correctly predicts the output value with respect to our evaluation semantics.

lemma
  assumes *kind g nid* = *IntegerEqualsNode x y*
  assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
  assumes *v* ≠ *UndefVal*
  assumes ([*g*, *m*, *p*] ⊢ *x* ↦ *xval*) ∧ ([*g*, *m*, *p*] ⊢ *y* ↦ *yval*)
  shows *val-to-bool* (*intval-equals xval yval*) ⟷ *v* = *IntVal32 1*
proof −
  have *v* = *intval-equals xval yval*
    using *assms*(*1*, *2*, *3*, *4*) *BinaryExprE IntegerEqualsNode bin-eval.simps*(*7*)
    by (*smt* (*verit*) *bin-eval.simps*(*11*) *encodeeval-def evalDet repDet*)
  then show *?thesis* **using** *intval-equals.simps val-to-bool.simps* **sorry**
qed

lemma *tryFoldIntegerEqualsAlwaysDistinct*:
  assumes *wf-stamp g stamps*
  assumes *kind g nid* = (*IntegerEqualsNode x y*)
  assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
  assumes *alwaysDistinct* (*stamps x*) (*stamps y*)
  shows *v* = *IntVal32 0*
proof −
  have ∀ *val*. ¬(*valid-value val* (*join* (*stamps x*) (*stamps y*)))
    using *assms*(*1*,*4*) **unfolding** *alwaysDistinct.simps*
    by (*smt* (*verit*, *best*) *is-stamp-empty.elims*(*2*) *valid-int valid-value.simps*(*1*))
  have ¬(∃ *val* . ([*g*, *m*, *p*] ⊢ *x* ↦ *val*) ∧ ([*g*, *m*, *p*] ⊢ *y* ↦ *val*))
    using *assms*(*1*,*4*) **unfolding** *alwaysDistinct.simps wf-stamp.simps encodee-val-def* **sorry**
  then show *?thesis* **sorry**
qed

lemma *tryFoldIntegerEqualsNeverDistinct*:
  assumes *wf-stamp g stamps*
  assumes *kind g nid* = (*IntegerEqualsNode x y*)
  assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
  assumes *neverDistinct* (*stamps x*) (*stamps y*)
  shows *v* = *IntVal32 1*
  using *assms IntegerEqualsNodeE* **sorry**

lemma *tryFoldIntegerLessThanTrue*:
  assumes *wf-stamp g stamps*
  assumes *kind g nid* = (*IntegerLessThanNode x y*)
  assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
  assumes *stpi-upper* (*stamps x*) < *stpi-lower* (*stamps y*)
  shows *v* = *IntVal32 1*
proof −
  have *stamp-type*: *is-IntegerStamp* (*stamps x*)
    using *assms*
    sorry
  obtain *xval* **where** *xval*: [*g*, *m*, *p*] ⊢ *x* ↦ *xval*

using *assms*(*2*,*3*) **sorry**
obtain *yval* **where** *yval*: $[g, m, p] \vdash y \mapsto yval$
using *assms*(*2*,*3*) **sorry**
have *is-IntegerStamp* (*stamps x*) ∧ *is-IntegerStamp* (*stamps y*)
using *assms*(*4*)
sorry
then have *val-to-bool* (*intval-less-than xval yval*)
sorry
then show *?thesis*
sorry
qed

lemma *tryFoldIntegerLessThanFalse*:
assumes *wf-stamp g stamps*
assumes *kind g nid* = (*IntegerLessThanNode x y*)
assumes $[g, m, p] \vdash nid \mapsto v$
assumes *stpi-lower* (*stamps x*) ≥ *stpi-upper* (*stamps y*)
shows *v* = *IntVal32 0*
proof −
have *stamp-type*: *is-IntegerStamp* (*stamps x*)
using *assms*
sorry
obtain *xval* **where** *xval*: $[g, m, p] \vdash x \mapsto xval$
using *assms*(*2*,*3*) **sorry**
obtain *yval* **where** *yval*: $[g, m, p] \vdash y \mapsto yval$
using *assms*(*2*,*3*) **sorry**
have *is-IntegerStamp* (*stamps x*) ∧ *is-IntegerStamp* (*stamps y*)
using *assms*(*4*)
sorry
then have ¬(*val-to-bool* (*intval-less-than xval yval*))
sorry
then show *?thesis*
sorry
qed

theorem *tryFoldProofTrue*:
assumes *wf-stamp g stamps*
assumes *tryFold* (*kind g nid*) *stamps True*
assumes $[g, m, p] \vdash nid \mapsto v$
shows *val-to-bool v*
using *assms*(*2*) **proof** (*induction kind g nid stamps True rule*: *tryFold.induct*)
case (*1 stamps x y*)
then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms* **sorry**
next
case (*2 stamps x y*)
then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms* **sorry**
next
case (*3 stamps x y*)
then show *?case* **using** *tryFoldIntegerLessThanTrue assms* **sorry**

next
case (*4 stamps x y*)
  then show *?case* **using** *tryFoldIntegerLessThanFalse assms* **sorry**
qed

theorem *tryFoldProofFalse*:
  assumes *wf-stamp g stamps*
  assumes *tryFold* (*kind g nid*) *stamps False*
  assumes $[g,\ m,\ p] \vdash nid \mapsto v$
  shows $\neg$(*val-to-bool v*)
using *assms*(*2*) **proof** (*induction kind g nid stamps False rule*: *tryFold.induct*)
case (*1 stamps x y*)
  then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms* **sorry**
next
case (*2 stamps x y*)
  then show *?case* **using** *tryFoldIntegerEqualsNeverDistinct assms* **sorry**
next
  case (*3 stamps x y*)
  then show *?case* **using** *tryFoldIntegerLessThanTrue assms* **sorry**
next
  case (*4 stamps x y*)
  then show *?case* **using** *tryFoldIntegerLessThanFalse assms* **sorry**

qed

inductive-cases *StepE*:
  $g,\ p \vdash (nid,m,h) \rightarrow (nid',m',h)$

Perform conditional elimination rewrites on the graph for a particular node.

In order to determine conditional eliminations appropriately the rule needs two data structures produced by static analysis. The first parameter is the set of IRNodes that we know result in a true value when evaluated. The second parameter is a mapping from node identifiers to the flow-sensitive stamp.

The relation transforms the third parameter to the fifth parameter for a node identifier which represents the fourth parameter.

inductive *ConditionalEliminationStep* ::
  IRExpr set $\Rightarrow$ (*ID* $\Rightarrow$ *Stamp*) $\Rightarrow$ *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *IRGraph* $\Rightarrow$ *bool* **where**
  impliesTrue:
  $[\![$*kind g ifcond* = (*IfNode cid t f*);
    $g \vdash cid \simeq cond$;
    $\exists\ ce \in conds\ .\ (ce\ \&\ cond \hookrightarrow True)$;
    g' = *constantCondition True ifcond* (*kind g ifcond*) *g*
    $]\!] \Longrightarrow$ *ConditionalEliminationStep conds stamps g ifcond* g' |

  impliesFalse:

$[\![kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
$g \vdash cid \simeq cond;$
$\exists\ ce \in conds\ .\ (ce\ \&\ cond \hookrightarrow False);$
$g' = constantCondition\ False\ ifcond\ (kind\ g\ ifcond)\ g$
$]\!] \Longrightarrow ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'\ |$

tryFoldTrue:
$[\![kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
$cond = kind\ g\ cid;$
$tryFold\ (kind\ g\ cid)\ stamps\ True;$
$g' = constantCondition\ True\ ifcond\ (kind\ g\ ifcond)\ g$
$]\!] \Longrightarrow ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'\ |$

tryFoldFalse:
$[\![kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
$cond = kind\ g\ cid;$
$tryFold\ (kind\ g\ cid)\ stamps\ False;$
$g' = constantCondition\ False\ ifcond\ (kind\ g\ ifcond)\ g$
$]\!] \Longrightarrow ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'$

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

thm *ConditionalEliminationStep.equation*

12.2 Control-flow Graph Traversal

type-synonym *Seen* = *ID set*
type-synonym *Condition* = *IRNode*
type-synonym *Conditions* = *Condition list*
type-synonym *StampFlow* = (*ID* ⇒ *Stamp*) *list*

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

fun *nextEdge* :: *Seen* ⇒ *ID* ⇒ *IRGraph* ⇒ *ID option* **where**
$nextEdge\ seen\ nid\ g =$
$(let\ nids = (filter\ (\lambda nid'.\ nid' \notin seen)\ (successors\text{-}of\ (kind\ g\ nid)))\ in$
$(if\ length\ nids > 0\ then\ Some\ (hd\ nids)\ else\ None))$

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

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

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

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

When the basic block of an if statement is entered, we know that the condition of the preceding if statement must be true. As in the GraalVM compiler, we introduce the registerNewCondition funciton which roughly corresponds to the ConditionalEliminationPhase.registerNewCondition. This method updates the flow-sensitive stamp information based on the condition which we know must be true.

fun *clip-upper* :: *Stamp* ⇒ *int* ⇒ *Stamp* **where**
  clip-upper (*IntegerStamp b l h*) *c* = (*IntegerStamp b l c*) |
  clip-upper s c = *s*
fun *clip-lower* :: *Stamp* ⇒ *int* ⇒ *Stamp* **where**
  clip-lower (*IntegerStamp b l h*) *c* = (*IntegerStamp b c h*) |
  clip-lower s c = *s*

fun *registerNewCondition* :: *IRGraph* ⇒ *Condition* ⇒ (*ID* ⇒ *Stamp*) ⇒ (*ID* ⇒ *Stamp*) **where**

  registerNewCondition g (*IntegerEqualsNode x y*) *stamps* =
    (*stamps*(*x* := *join* (*stamps x*) (*stamps y*)))(*y* := *join* (*stamps x*) (*stamps y*)) |

  registerNewCondition g (*IntegerLessThanNode x y*) *stamps* =
    (*stamps*
      (*x* := *clip-upper* (*stamps x*) (*stpi-lower* (*stamps y*))))
      (*y* := *clip-lower* (*stamps y*) (*stpi-upper* (*stamps x*))) |
  registerNewCondition g - stamps = *stamps*

fun *hdOr* :: *'a list* ⇒ *'a* ⇒ *'a* **where**
  hdOr (*x # xs*) *de* = *x* |
  hdOr [] *de* = *de*

The Step relation is a small-step traversal of the graph which handles transitions between individual nodes of the graph.

It relates a pairs of tuple of the current node, the set of seen nodes, the always true stack of IfNode conditions, and the flow-sensitive stamp information.

inductive *Step*
  :: *IRGraph* ⇒ (*ID* × *Seen* × *Conditions* × *StampFlow*) ⇒ (*ID* × *Seen* × *Conditions* × *StampFlow*) *option* ⇒ *bool*
  for *g* **where**
  — Hit a BeginNode with an IfNode predecessor which represents the start of a

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

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

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

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

i $=$ *find-index nid* (*successors-of* (*kind g ifcond*));
c $=$ (*if i* $=$ *0 then kind g cond else LogicNegationNode cond*);
conds' $=$ *c* # *conds*;

flow' $=$ *registerNewCondition g c* (*hdOr flow* (*stamp g*))$]\!]$
$\Longrightarrow$ *Step g* (*nid*, *seen*, *conds*, *flow*) (*Some* (*nid'*, *seen'*, *conds'*, *flow'* # *flow*)) |

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

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

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

nid' $=$ *any-usage g nid*;

conds' $=$ *tl conds*;
flow' $=$ *tl flow*$]\!]$
$\Longrightarrow$ *Step g* (*nid*, *seen*, *conds*, *flow*) (*Some* (*nid'*, *seen'*, *conds'*, *flow'*)) |

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

$[\![\neg$(*is-EndNode* (*kind g nid*));
$\neg$(*is-BeginNode* (*kind g nid*));

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

Some nid' $=$ *nextEdge seen' nid g*$]\!]$
$\Longrightarrow$ *Step g* (*nid*, *seen*, *conds*, *flow*) (*Some* (*nid'*, *seen'*, *conds*, *flow*)) |

— We can cannot find a successor edge that is not in seen, give back None

$[\![\neg$(*is-EndNode* (*kind g nid*));
$\neg$(*is-BeginNode* (*kind g nid*));

nid $\notin$ *seen*;

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

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

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

code-pred $(modes\colon i \Rightarrow i \Rightarrow o \Rightarrow bool)\ Step$ **.**

The ConditionalEliminationPhase relation is responsible for combining the individual traversal steps from the Step relation and the optimizations from the ConditionalEliminationStep relation to perform a transformation of the whole graph.

end