

Veriopt

August 24, 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	3
1.1	Arithmetic Operators	4
1.2	Bitwise Operators and Comparisons	6
1.3	Narrowing and Widening Operators	8
1.4	Bit-Shifting Operators	11
2	Examples of Narrowing / Widening Functions	13
3	Nodes	14
3.1	Types of Nodes	14
3.2	Hierarchy of Nodes	22
4	Stamp Typing	29
5	Graph Representation	32
5.0.1	Example Graphs	37
5.1	Control-flow Graph Traversal	38
5.2	Structural Graph Comparison	40
6	Data-flow Semantics	41
6.1	Data-flow Tree Representation	42
6.2	Functions for re-calculating stamps	43
6.3	Data-flow Tree Evaluation	44
6.4	Data-flow Tree Refinement	46
6.5	Data-flow Tree Theorems	47
6.5.1	Deterministic Data-flow Evaluation	47
6.5.2	Typing Properties for Integer Evaluation Functions	47
6.5.3	Evaluation Results are Valid	49
6.5.4	Example Data-flow Optimisations	51
6.5.5	Monotonicity of Expression Refinement	52
6.6	Unfolding rules for evaltree quadruples down to bin-eval level	53
7	Tree to Graph	54
7.1	Subgraph to Data-flow Tree	55
7.2	Data-flow Tree to Subgraph	59
7.3	Lift Data-flow Tree Semantics	63
7.4	Graph Refinement	63
7.5	Maximal Sharing	63
7.6	Formedness Properties	63
7.7	Dynamic Frames	65
7.8	Tree to Graph Theorems	77
7.8.1	Extraction and Evaluation of Expression Trees is Deterministic.	77

7.8.2	Monotonicity of Graph Refinement	84
7.8.3	Lift Data-flow Tree Refinement to Graph Refinement	87
7.8.4	Term Graph Reconstruction	103
7.8.5	Data-flow Tree to Subgraph Preserves Maximal Sharing	111
8	Control-flow Semantics	141
8.1	Object Heap	141
8.2	Intraprocedural Semantics	142
8.3	Interprocedural Semantics	144
8.4	Big-step Execution	145
8.4.1	Heap Testing	146
8.5	Control-flow Semantics Theorems	147
8.5.1	Control-flow Step is Deterministic	147
9	Proof Infrastructure	152
9.1	Bisimulation	152
9.2	Graph Rewriting	153
9.3	Stuttering	157
9.4	Evaluation Stamp Theorems	158
9.4.1	Support Lemmas for Stamps and Upper/Lower Bounds	158
9.4.2	Support Lemmas for integer input/output size of unary and binary operators	163
9.4.3	Validity of UnaryAbs	164
9.4.4	Validity of UnaryNeg	166
9.4.5	Validity of UnaryNot	166
9.4.6	Validity of UnaryLogicNegation	167
9.4.7	Validity of UnaryNarrow	167
9.4.8	Validity of UnarySignExtend	170
9.4.9	Validity of all Unary Operators	172
9.4.10	Support Lemmas for Binary Operators	173
9.4.11	Validity of Stamp Meet and Join Operators	174
9.4.12	Validity of Whole Expression Tree Evaluation	176
10	Optization DSLs	182
10.1	Canonicalization DSL	185
11	Canonicalization Phase	188
11.1	Conditional Expression	189
12	Conditional Elimination Phase	192
12.1	Individual Elimination Rules	192
12.2	Control-flow Graph Traversal	203

1 Runtime Values and Arithmetic

```

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

```

```

lemma  $-\left((x::\text{float})-y\right) = (y-x)$ 
  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, but during calculations the smaller sizes are sign-extended to 32 bits, so here we model just 32 and 64 bit values.

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  $\equiv \{1, 8, 16, 32, 64\}$ 

```

```

type-synonym objref = nat option

```

```

datatype (discs-sels) Value =
  UndefVal |
  IntVal32 32 word |
  IntVal64 64 word |

  ObjRef objref |
  ObjStr string

```

Characterise integer values, covering both 32 and 64 bit. If a node has a stamp smaller than 32 bits (16, 8, or 1 bit), then the value will be sign-extended to 32 bits. This is necessary to match what the stamps specify 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.

definition *logic-negate* :: ('a::len) word \Rightarrow 'a word **where**
logic-negate x = (if x = 0 then 1 else 0)

definition *is-IntVal* :: Value \Rightarrow bool **where**
is-IntVal v = (*is-IntVal32* v \vee *is-IntVal64* v)

Extract signed integer values from both 32 and 64 bit.

fun *intval* :: Value \Rightarrow int **where**
intval (IntVal32 v) = sint v |
intval (IntVal64 v) = sint v

fun *wf-bool* :: Value \Rightarrow bool **where**
wf-bool (IntVal32 v) = (v = 0 \vee v = 1) |
wf-bool - = False

fun *val-to-bool* :: Value \Rightarrow bool **where**
val-to-bool (IntVal32 val) = (if val = 0 then False else True) |
val-to-bool (IntVal64 val) = (if val = 0 then False else True) |
val-to-bool v = False

fun *bool-to-val* :: bool \Rightarrow Value **where**
bool-to-val True = (IntVal32 1) |
bool-to-val False = (IntVal32 0)

value *sint*(word-of-int (1) :: int1)

fun *is-int-val* :: Value \Rightarrow bool **where**
is-int-val (IntVal32 v) = True |
is-int-val (IntVal64 v) = True |
is-int-val - = False

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 know to 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 \Rightarrow Value \Rightarrow Value **where**
intval-add (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1+v2)) |
intval-add (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1+v2)) |

```

    intval-add - - = UndefVal

instantiation Value :: ab-semigroup-add
begin

definition plus-Value :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
    plus-Value = intval-add

print-locale! ab-semigroup-add

instance proof
    fix a b c :: Value
    show a + b + c = a + (b + c)
        apply (simp add: plus-Value-def)
        apply (induction a; induction b; induction c; auto)
        done
    show a + b = b + a
        apply (simp add: plus-Value-def)
        apply (induction a; induction b; auto)
        done
qed
end

fun intval-sub :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
    intval-sub (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1-v2)) |
    intval-sub (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1-v2)) |
    intval-sub - - = UndefVal

instantiation Value :: minus
begin

definition minus-Value :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
    minus-Value = intval-sub

instance proof qed
end

fun intval-mul :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
    intval-mul (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1*v2)) |
    intval-mul (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1*v2)) |
    intval-mul - - = UndefVal

instantiation Value :: times
begin

definition times-Value :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
    times-Value = intval-mul

```

```
instance proof qed
end
```

```
fun intval-div :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-div (IntVal32 v1) (IntVal32 v2) = (IntVal32 (word-of-int((sint v1) sdiv
(sint v2)))) |
  intval-div (IntVal64 v1) (IntVal64 v2) = (IntVal64 (word-of-int((sint v1) sdiv
(sint v2)))) |
  intval-div - - =.UndefVal
```

```
instantiation Value :: divide
begin
```

```
definition divide-Value :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  divide-Value = intval-div
```

```
instance proof qed
end
```

```
fun intval-mod :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-mod (IntVal32 v1) (IntVal32 v2) = (IntVal32 (word-of-int((sint v1) smod
(sint v2)))) |
  intval-mod (IntVal64 v1) (IntVal64 v2) = (IntVal64 (word-of-int((sint v1) smod
(sint v2)))) |
  intval-mod - - =.UndefVal
```

```
instantiation Value :: modulo
begin
```

```
definition modulo-Value :: Value  $\Rightarrow$  Value  $\Rightarrow$  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  $\Rightarrow$  Value  $\Rightarrow$  Value where
```

```

    intval-and (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1 AND v2)) |
    intval-and (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1 AND v2)) |
    intval-and - - = UndefVal

fun intval-or :: Value ⇒ Value ⇒ Value where
    intval-or (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1 OR v2)) |
    intval-or (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1 OR v2)) |
    intval-or - - = UndefVal

fun intval-xor :: Value ⇒ Value ⇒ Value where
    intval-xor (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1 XOR v2)) |
    intval-xor (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1 XOR v2)) |
    intval-xor - - = UndefVal

fun intval-short-circuit-or :: Value ⇒ Value ⇒ Value where
    intval-short-circuit-or (IntVal32 v1) (IntVal32 v2) = bool-to-val ((v1 ≠ 0) ∨ (v2
    ≠ 0)) |
    intval-short-circuit-or (IntVal64 v1) (IntVal64 v2) = bool-to-val ((v1 ≠ 0) ∨ (v2
    ≠ 0)) |
    intval-short-circuit-or - - = UndefVal

fun intval-equals :: Value ⇒ Value ⇒ Value where
    intval-equals (IntVal32 v1) (IntVal32 v2) = bool-to-val (v1 = v2) |
    intval-equals (IntVal64 v1) (IntVal64 v2) = bool-to-val (v1 = v2) |
    intval-equals - - = UndefVal

fun intval-less-than :: Value ⇒ Value ⇒ Value where
    intval-less-than (IntVal32 v1) (IntVal32 v2) = bool-to-val (v1 <s v2) |
    intval-less-than (IntVal64 v1) (IntVal64 v2) = bool-to-val (v1 <s v2) |
    intval-less-than - - = UndefVal

fun intval-below :: Value ⇒ Value ⇒ Value where
    intval-below (IntVal32 v1) (IntVal32 v2) = bool-to-val (v1 < v2) |
    intval-below (IntVal64 v1) (IntVal64 v2) = bool-to-val (v1 < v2) |
    intval-below - - = UndefVal

fun intval-not :: Value ⇒ Value where
    intval-not (IntVal32 v) = (IntVal32 (NOT v)) |
    intval-not (IntVal64 v) = (IntVal64 (NOT v)) |
    intval-not - = UndefVal

fun intval-negate :: Value ⇒ Value where
    intval-negate (IntVal32 v) = IntVal32 (− v) |
    intval-negate (IntVal64 v) = IntVal64 (− v) |
    intval-negate - = UndefVal

fun intval-abs :: Value ⇒ Value where
    intval-abs (IntVal32 v) = (if (v) <s 0 then (IntVal32 (− v)) else (IntVal32 v)) |
    intval-abs (IntVal64 v) = (if (v) <s 0 then (IntVal64 (− v)) else (IntVal64 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*)

fun *intval-logic-negation* :: *Value* \Rightarrow *Value* **where**
intval-logic-negation (*IntVal32* *v*) = (*IntVal32* (*logic-negate* *v*)) |
intval-logic-negation (*IntVal64* *v*) = (*IntVal64* (*logic-negate* *v*)) |
intval-logic-negation - = *UndefVal*

lemma *intval-eq32*:

assumes *intval-equals* (*IntVal32* *v1*) *v2* \neq *UndefVal*
shows *is-IntVal32* *v2*
by (metis *Value.exhaust-disc* *assms* *intval-equals.simps*(10) *intval-equals.simps*(12)
intval-equals.simps(15) *intval-equals.simps*(16) *is-IntVal64-def* *is-ObjRef-def* *is-ObjStr-def*)

lemma *intval-eq32-simp*:

assumes *intval-equals* (*IntVal32* *v1*) *v2* \neq *UndefVal*
shows *intval-equals* (*IntVal32* *v1*) *v2* = *bool-to-val* (*v1* = *un-IntVal32* *v2*)
by (metis *Value.collapse*(1) *assms* *intval-eq32* *intval-equals.simps*(1))

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.

When narrowing to less than 32 bits, we sign extend back to 32 bits, because we always represent integer values as either 32 or 64 bits.

fun *narrow-helper* :: *nat* \Rightarrow *nat* \Rightarrow *int32* \Rightarrow *Value* **where**
narrow-helper *inBits* *outBits* *val* =
 (if *outBits* \leq *inBits* \wedge *outBits* \leq 32 \wedge
 outBits \in *valid-int-widths* \wedge
 inBits \in *valid-int-widths*
 then *IntVal32* (*signed-take-bit* (*outBits* - 1) *val*)
 else *UndefVal*)

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

fun *intval-narrow* :: *nat* \Rightarrow *nat* \Rightarrow *Value* \Rightarrow *Value* **where**
intval-narrow *inBits* *outBits* (*IntVal32* *v*) =
 (if *inBits* = 64
 then *UndefVal*
 else *narrow-helper* *inBits* *outBits* *v*) |
intval-narrow *inBits* *outBits* (*IntVal64* *v*) =
 (if *inBits* = 64
 then (if *outBits* = 64
 then *IntVal64* *v*
 else *narrow-helper* *inBits* *outBits* (*scast* *v*))

```

    else UndefVal) |
  intval-narrow - - - = UndefVal

value intval(intval-narrow 16 8 (IntVal32 (512 - 2)))

fun choose-32-64 :: nat ⇒ int64 ⇒ Value where
  choose-32-64 outBits v = (if outBits = 64 then (IntVal64 v) else (IntVal32 (scast
v)))

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

fun sign-extend-helper :: nat ⇒ nat ⇒ int32 ⇒ Value where
  sign-extend-helper inBits outBits val =
    (if inBits ≤ outBits ∧ inBits ≤ 32 ∧
      outBits ∈ valid-int-widths ∧
      inBits ∈ valid-int-widths
    then
      (if outBits = 64
        then IntVal64 (scast (signed-take-bit (inBits - 1) val))
        else IntVal32 (signed-take-bit (inBits - 1) val))
      else UndefVal)

fun intval-sign-extend :: nat ⇒ nat ⇒ Value ⇒ Value where
  intval-sign-extend inBits outBits (IntVal32 v) =
    sign-extend-helper inBits outBits v |
  intval-sign-extend inBits outBits (IntVal64 v) =
    (if inBits=64 ∧ outBits=64 then IntVal64 v else UndefVal) |
  intval-sign-extend - - - = UndefVal

fun zero-extend-helper :: nat ⇒ nat ⇒ int32 ⇒ Value where
  zero-extend-helper inBits outBits val =
    (if inBits ≤ outBits ∧ inBits ≤ 32 ∧
      outBits ∈ valid-int-widths ∧
      inBits ∈ valid-int-widths
    then
      (if outBits = 64
        then IntVal64 (ucast (take-bit inBits val))
        else IntVal32 (take-bit inBits val))
      else UndefVal)

fun intval-zero-extend :: nat ⇒ nat ⇒ Value ⇒ Value where
  intval-zero-extend inBits outBits (IntVal32 v) =
    zero-extend-helper inBits outBits v |
  intval-zero-extend inBits outBits (IntVal64 v) =
    (if inBits=64 ∧ outBits=64 then IntVal64 v else UndefVal) |
  intval-zero-extend - - - = UndefVal

```

Some well-formedness results to help reasoning about narrowing and widening operators

lemma *narrow-helper-ok*:

assumes *narrow-helper inBits outBits val* \neq *UndefVal*
shows $0 < \text{outBits} \wedge \text{outBits} \leq 32 \wedge$
 $\text{outBits} \leq \text{inBits} \wedge$
 $\text{outBits} \in \text{valid-int-widths} \wedge$
 $\text{inBits} \in \text{valid-int-widths}$
using *assms narrow-helper.simps neq0-conv* **by** *fastforce*

lemma *intval-narrow-ok*:

assumes *intval-narrow inBits outBits val* \neq *UndefVal*
shows $0 < \text{outBits} \wedge$
 $\text{outBits} \leq \text{inBits} \wedge$
 $\text{outBits} \in \text{valid-int-widths} \wedge$
 $\text{inBits} \in \text{valid-int-widths}$
using *assms narrow-helper-ok intval-narrow.simps neq0-conv*
by (*smt (verit, best) insertCI intval-sign-extend.elims order-le-less zero-neq-numeral*)

lemma *narrow-takes-64*:

assumes *result = intval-narrow inBits outBits value*
assumes *result* \neq *UndefVal*
shows *is-IntVal64 value* $= (\text{inBits} = 64)$
using *assms* **by** (*cases value; simp; presburger*)

lemma *narrow-gives-64*:

assumes *result = intval-narrow inBits outBits value*
assumes *result* \neq *UndefVal*
shows *is-IntVal64 result* $= (\text{outBits} = 64)$
using *assms*
by (*smt (verit, best) Value.case-eq-if Value.discI(1) Value.discI(2) Value.disc-eq-case(3)*
add-diff-cancel-left' diff-is-0-eq intval-narrow.elims narrow-helper.simps numeral-Bit0
zero-neq-numeral)

lemma *sign-extend-helper-ok*:

assumes *sign-extend-helper inBits outBits val* \neq *UndefVal*
shows $0 < \text{inBits} \wedge \text{inBits} \leq 32 \wedge$
 $\text{inBits} \leq \text{outBits} \wedge$
 $\text{outBits} \in \text{valid-int-widths} \wedge$
 $\text{inBits} \in \text{valid-int-widths}$
using *assms sign-extend-helper.simps neq0-conv* **by** *fastforce*

lemma *intval-sign-extend-ok*:

assumes *intval-sign-extend inBits outBits val* \neq *UndefVal*
shows $0 < \text{inBits} \wedge$
 $\text{inBits} \leq \text{outBits} \wedge$
 $\text{outBits} \in \text{valid-int-widths} \wedge$

$inBits \in \text{valid-int-widths}$
using *assms sign-extend-helper-ok intval-sign-extend.simps neq0-conv*
by (*smt (verit, best) insertCI intval-sign-extend.elims order-le-less zero-neq-numeral*)

lemma *zero-extend-helper-ok*:
assumes *zero-extend-helper inBits outBits val \neq UndefVal*
shows $0 < inBits \wedge inBits \leq 32 \wedge$
 $inBits \leq outBits \wedge$
 $outBits \in \text{valid-int-widths} \wedge$
 $inBits \in \text{valid-int-widths}$
using *assms zero-extend-helper.simps neq0-conv* **by** *fastforce*

lemma *intval-zero-extend-ok*:
assumes *intval-zero-extend inBits outBits val \neq UndefVal*
shows $0 < inBits \wedge$
 $inBits \leq outBits \wedge$
 $outBits \in \text{valid-int-widths} \wedge$
 $inBits \in \text{valid-int-widths}$
using *assms zero-extend-helper-ok intval-zero-extend.simps neq0-conv*
by (*smt (verit, best) insertCI intval-zero-extend.elims order-le-less zero-neq-numeral*)

1.4 Bit-Shifting Operators

definition *shiffl* (**infix** $<<$ 75) **where**
 $shiffl\ w\ n = (\text{push-bit}\ n)\ w$

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

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

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

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

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

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

value $(255 :: 8\ \text{word}) >>> (2 :: \text{nat})$

definition *signed-shiftr* :: 'a :: len word \Rightarrow nat \Rightarrow '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 *intval-left-shift* :: Value \Rightarrow Value \Rightarrow Value **where**
intval-left-shift (IntVal32 v1) (IntVal32 v2) = IntVal32 (v1 << unat (v2 AND 0x1f)) |
intval-left-shift (IntVal32 v1) (IntVal64 v2) = IntVal32 (v1 << unat (v2 AND 0x1f)) |
intval-left-shift (IntVal64 v1) (IntVal32 v2) = IntVal64 (v1 << unat (v2 AND 0x3f)) |
intval-left-shift (IntVal64 v1) (IntVal64 v2) = IntVal64 (v1 << unat (v2 AND 0x3f)) |
intval-left-shift - - = UndefVal

fun *intval-right-shift* :: Value \Rightarrow Value \Rightarrow Value **where**
intval-right-shift (IntVal32 v1) (IntVal32 v2) = IntVal32 (v1 >> unat (v2 AND 0x1f)) |
intval-right-shift (IntVal32 v1) (IntVal64 v2) = IntVal32 (v1 >> unat (v2 AND 0x1f)) |
intval-right-shift (IntVal64 v1) (IntVal32 v2) = IntVal64 (v1 >> unat (v2 AND 0x3f)) |
intval-right-shift (IntVal64 v1) (IntVal64 v2) = IntVal64 (v1 >> unat (v2 AND 0x3f)) |
intval-right-shift - - = UndefVal

fun *intval-uright-shift* :: Value \Rightarrow Value \Rightarrow Value **where**
intval-uright-shift (IntVal32 v1) (IntVal32 v2) = IntVal32 (v1 >>> unat (v2 AND 0x1f)) |
intval-uright-shift (IntVal32 v1) (IntVal64 v2) = IntVal32 (v1 >>> unat (v2 AND 0x1f)) |
intval-uright-shift (IntVal64 v1) (IntVal32 v2) = IntVal64 (v1 >>> unat (v2 AND 0x3f)) |
intval-uright-shift (IntVal64 v1) (IntVal64 v2) = IntVal64 (v1 >>> unat (v2 AND 0x3f)) |
intval-uright-shift - - = UndefVal

end

2 Examples of Narrowing / Widening Functions

experiment begin

corollary *intval-narrow* 32 8 (*IntVal32* (256 + 128)) = *IntVal32* (-128) **by** *simp*
corollary *intval-narrow* 32 8 (*IntVal32* (-2)) = *IntVal32* (-2) **by** *simp*
corollary *intval-narrow* 32 1 (*IntVal32* (-2)) = *IntVal32* 0 **by** *simp*
corollary *intval-narrow* 32 1 (*IntVal32* (-3)) = *IntVal32* (-1) **by** *simp*

corollary *intval-narrow* 32 8 (*IntVal64* (-2)) = *UndefVal* **by** *simp*
corollary *intval-narrow* 64 8 (*IntVal32* (-2)) = *UndefVal* **by** *simp*
corollary *intval-narrow* 64 8 (*IntVal64* (-2)) = *IntVal32* (-2) **by** *simp*
corollary *intval-narrow* 64 8 (*IntVal64* (256+127)) = *IntVal32* 127 **by** *simp*
corollary *intval-narrow* 64 32 (*IntVal64* (-2)) = *IntVal32* (-2) **by** *simp*
corollary *intval-narrow* 64 64 (*IntVal64* (-2)) = *IntVal64* (-2) **by** *simp*
end

experiment begin

corollary *intval-sign-extend* 8 32 (*IntVal32* (256 + 128)) = *IntVal32* (-128) **by** *simp*
corollary *intval-sign-extend* 8 32 (*IntVal32* (-2)) = *IntVal32* (-2) **by** *simp*
corollary *intval-sign-extend* 1 32 (*IntVal32* (-2)) = *IntVal32* 0 **by** *simp*
corollary *intval-sign-extend* 1 32 (*IntVal32* (-3)) = *IntVal32* (-1) **by** *simp*

corollary *intval-sign-extend* 8 32 (*IntVal64* (-2)) = *UndefVal* **by** *simp*
corollary *intval-sign-extend* 8 64 (*IntVal64* (-2)) = *UndefVal* **by** *simp*
corollary *intval-sign-extend* 8 64 (*IntVal32* (-2)) = *IntVal64* (-2) **by** *simp*
corollary *intval-sign-extend* 32 64 (*IntVal32* (-2)) = *IntVal64* (-2) **by** *simp*
corollary *intval-sign-extend* 64 64 (*IntVal64* (-2)) = *IntVal64* (-2) **by** *simp*
end

experiment begin

corollary *intval-zero-extend* 8 32 (*IntVal32* (256 + 128)) = *IntVal32* 128 **by** *simp*
corollary *intval-zero-extend* 8 32 (*IntVal32* (-2)) = *IntVal32* 254 **by** *simp*
corollary *intval-zero-extend* 1 32 (*IntVal32* (-1)) = *IntVal32* 1 **by** *simp*
corollary *intval-zero-extend* 1 32 (*IntVal32* (-2)) = *IntVal32* 0 **by** *simp*

corollary *intval-zero-extend* 8 32 (*IntVal64* (-2)) = *UndefVal* **by** *simp*
corollary *intval-zero-extend* 8 64 (*IntVal64* (-2)) = *UndefVal* **by** *simp*
corollary *intval-zero-extend* 8 64 (*IntVal32* (-2)) = *IntVal64* 254 **by** *simp*
corollary *intval-zero-extend* 32 64 (*IntVal32* (-2)) = *IntVal64* 4294967294 **by** *simp*
end

```

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

```

```

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

```

```

lemma intval-add (IntVal32 (2^31-1)) (IntVal32 (2^31-1)) = IntVal32 (-2)
  by eval
lemma intval-add (IntVal64 (2^31-1)) (IntVal64 (2^31-1)) = IntVal64 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: IN-
PUT-STATE option) (ir-next: SUCC)
| NotNode (ir-value: INPUT)
| OrNode (ir-x: INPUT) (ir-y: INPUT)
| ParameterNode (ir-index: nat)
| PiNode (ir-object: INPUT) (ir-guard-opt: INPUT-GUARD option)
| ReturnNode (ir-result-opt: INPUT option) (ir-memoryMap-opt: INPUT-EXT
option)
| RightShiftNode (ir-x: INPUT) (ir-y: INPUT)
| ShortCircuitOrNode (ir-x: INPUT-COND) (ir-y: INPUT-COND)
| SignExtendNode (ir-inputBits: nat) (ir-resultBits: nat) (ir-value: INPUT)
| SignedDivNode (ir-nid: ID) (ir-x: INPUT) (ir-y: INPUT) (ir-zeroCheck-opt: IN-
PUT-GUARD option) (ir-stateBefore-opt: INPUT-STATE option) (ir-next: SUCC)

| 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  $\Rightarrow$  'a list where
  opt-to-list None = [] |
  opt-to-list (Some v) = [v]

```

```

fun opt-list-to-list :: 'a list option  $\Rightarrow$  '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 stateDur-
ing) @ (opt-to-list stateAfter) |
  inputs-of-IsNullNode:
    inputs-of (IsNullNode value) = [value] |
  inputs-of-KillingBeginNode:
    inputs-of (KillingBeginNode next) = [] |

```

inputs-of-LeftShiftNode:
inputs-of (LeftShiftNode x y) = [x, y] |
inputs-of-LoadFieldNode:
inputs-of (LoadFieldNode nid0 field object next) = (opt-to-list object) |
inputs-of-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 virtualObjectMap-
pings) = [] |
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) = []  $\wedge$  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  $\Rightarrow$  bool where
  is-EndNode EndNode = True |
  is-EndNode - = False

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

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

fun is-ShiftNode :: IRNode  $\Rightarrow$  bool where
  is-ShiftNode n = ((is-LeftShiftNode n)  $\vee$  (is-RightShiftNode n)  $\vee$  (is-UnsignedRightShiftNode
n))

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

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

```

```

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

fun is-UnaryArithmeticNode :: IRNode  $\Rightarrow$  bool where
  is-UnaryArithmeticNode n = ((is-AbsNode n)  $\vee$  (is-NegateNode n)  $\vee$  (is-NotNode
n))

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

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

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

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

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

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

fun is-BinaryOpLogicNode :: IRNode  $\Rightarrow$  bool where
  is-BinaryOpLogicNode n = ((is-CompareNode n))

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

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

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

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

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

```



```

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

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

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

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

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)  $\vee$  (is-VirtualState n))

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

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

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

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

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

fun is-IterableNodeType :: IRNode  $\Rightarrow$  bool where
  is-IterableNodeType n = ((is-AbstractBeginNode n)  $\vee$  (is-AbstractMergeNode n)  $\vee$ 
(is-FrameState n)  $\vee$  (is-IfNode n)  $\vee$  (is-IntegerDivRemNode n)  $\vee$  (is-InvokeWithExceptionNode
n)  $\vee$  (is-LoopBeginNode n)  $\vee$  (is-LoopExitNode n)  $\vee$  (is-MethodCallTargetNode n)
 $\vee$  (is-ParameterNode n)  $\vee$  (is-ReturnNode n)  $\vee$  (is-ShortCircuitOrNode n))

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

```

is-sequential-node - = *False*

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

fun *is-same-ir-node-type* :: *IRNode* \Rightarrow *IRNode* \Rightarrow *bool* **where**

is-same-ir-node-type *n1* *n2* = (
 ((*is-AbsNode* *n1*) \wedge (*is-AbsNode* *n2*)) \vee
 ((*is-AddNode* *n1*) \wedge (*is-AddNode* *n2*)) \vee
 ((*is-AndNode* *n1*) \wedge (*is-AndNode* *n2*)) \vee
 ((*is-BeginNode* *n1*) \wedge (*is-BeginNode* *n2*)) \vee
 ((*is-BytecodeExceptionNode* *n1*) \wedge (*is-BytecodeExceptionNode* *n2*)) \vee
 ((*is-ConditionalNode* *n1*) \wedge (*is-ConditionalNode* *n2*)) \vee
 ((*is-ConstantNode* *n1*) \wedge (*is-ConstantNode* *n2*)) \vee
 ((*is-DynamicNewArrayNode* *n1*) \wedge (*is-DynamicNewArrayNode* *n2*)) \vee
 ((*is-EndNode* *n1*) \wedge (*is-EndNode* *n2*)) \vee
 ((*is-ExceptionObjectNode* *n1*) \wedge (*is-ExceptionObjectNode* *n2*)) \vee
 ((*is-FrameState* *n1*) \wedge (*is-FrameState* *n2*)) \vee
 ((*is-IfNode* *n1*) \wedge (*is-IfNode* *n2*)) \vee
 ((*is-IntegerBelowNode* *n1*) \wedge (*is-IntegerBelowNode* *n2*)) \vee
 ((*is-IntegerEqualsNode* *n1*) \wedge (*is-IntegerEqualsNode* *n2*)) \vee
 ((*is-IntegerLessThanNode* *n1*) \wedge (*is-IntegerLessThanNode* *n2*)) \vee
 ((*is-InvokeNode* *n1*) \wedge (*is-InvokeNode* *n2*)) \vee
 ((*is-InvokeWithExceptionNode* *n1*) \wedge (*is-InvokeWithExceptionNode* *n2*)) \vee
 ((*is-IsNullNode* *n1*) \wedge (*is-IsNullNode* *n2*)) \vee
 ((*is-KillingBeginNode* *n1*) \wedge (*is-KillingBeginNode* *n2*)) \vee
 ((*is-LoadFieldNode* *n1*) \wedge (*is-LoadFieldNode* *n2*)) \vee
 ((*is-LogicNegationNode* *n1*) \wedge (*is-LogicNegationNode* *n2*)) \vee
 ((*is-LoopBeginNode* *n1*) \wedge (*is-LoopBeginNode* *n2*)) \vee
 ((*is-LoopEndNode* *n1*) \wedge (*is-LoopEndNode* *n2*)) \vee
 ((*is-LoopExitNode* *n1*) \wedge (*is-LoopExitNode* *n2*)) \vee
 ((*is-MergeNode* *n1*) \wedge (*is-MergeNode* *n2*)) \vee
 ((*is-MethodCallTargetNode* *n1*) \wedge (*is-MethodCallTargetNode* *n2*)) \vee
 ((*is-MulNode* *n1*) \wedge (*is-MulNode* *n2*)) \vee
 ((*is-NegateNode* *n1*) \wedge (*is-NegateNode* *n2*)) \vee
 ((*is-NewArrayNode* *n1*) \wedge (*is-NewArrayNode* *n2*)) \vee
 ((*is-NewInstanceNode* *n1*) \wedge (*is-NewInstanceNode* *n2*)) \vee
 ((*is-NotNode* *n1*) \wedge (*is-NotNode* *n2*)) \vee
 ((*is-OrNode* *n1*) \wedge (*is-OrNode* *n2*)) \vee
 ((*is-ParameterNode* *n1*) \wedge (*is-ParameterNode* *n2*)) \vee
 ((*is-PiNode* *n1*) \wedge (*is-PiNode* *n2*)) \vee
 ((*is-ReturnNode* *n1*) \wedge (*is-ReturnNode* *n2*)) \vee
 ((*is-ShortCircuitOrNode* *n1*) \wedge (*is-ShortCircuitOrNode* *n2*)) \vee
 ((*is-SignedDivNode* *n1*) \wedge (*is-SignedDivNode* *n2*)) \vee
 ((*is-StartNode* *n1*) \wedge (*is-StartNode* *n2*)) \vee
 ((*is-StoreFieldNode* *n1*) \wedge (*is-StoreFieldNode* *n2*)) \vee
 ((*is-SubNode* *n1*) \wedge (*is-SubNode* *n2*)) \vee
 ((*is-UnwindNode* *n1*) \wedge (*is-UnwindNode* *n2*)) \vee
 ((*is-ValuePhiNode* *n1*) \wedge (*is-ValuePhiNode* *n2*)) \vee

```
((is-ValueProxyNode n1) ∧ (is-ValueProxyNode n2)) ∨
((is-XorNode n1) ∧ (is-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 bit-bounds :: nat ⇒ (int × int) where
  bit-bounds bits = (((2 ^ bits) div 2) * -1, ((2 ^ bits) div 2) - 1)
```

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

```

```

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

```

```

    is-stamp-empty x = False

```

— 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  $\wedge$  nn2) (an1  $\wedge$  an2)
) |
meet s1 s2 = IllegalStamp

```

— Calculate the join stamp of two stamps

```

fun join :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  Stamp where
  join VoidStamp VoidStamp = VoidStamp |
  join (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1  $\neq$  b2 then IllegalStamp else
    (IntegerStamp b1 (max l1 l2) (min u1 u2))
  ) |
  join (KlassPointerStamp nn1 an1) (KlassPointerStamp nn2 an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (KlassPointerStamp nn1 an1))
    else (KlassPointerStamp (nn1  $\vee$  nn2) (an1  $\vee$  an2))
  ) |
  join (MethodCountersPointerStamp nn1 an1) (MethodCountersPointerStamp nn2
an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (MethodCountersPointerStamp nn1 an1))
    else (MethodCountersPointerStamp (nn1  $\vee$  nn2) (an1  $\vee$  an2))
  ) |
  join (MethodPointersStamp nn1 an1) (MethodPointersStamp nn2 an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (MethodPointersStamp nn1 an1))
    else (MethodPointersStamp (nn1  $\vee$  nn2) (an1  $\vee$  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  $\Rightarrow$  Value where
  asConstant (IntegerStamp b l h) = (if l = h then IntVal64 (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  $\Rightarrow$  Stamp  $\Rightarrow$  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  $\Rightarrow$  Stamp  $\Rightarrow$  bool where
  neverDistinct stamp1 stamp2 = (asConstant stamp1 = asConstant stamp2  $\wedge$ 
asConstant stamp1  $\neq$  UndefVal)

```



```

fun constantAsStamp :: Value  $\Rightarrow$  Stamp where
  constantAsStamp (IntVal32 v) = (IntegerStamp (nat 32) (sint v) (sint v)) |
  constantAsStamp (IntVal64 v) = (IntegerStamp (nat 64) (sint v) (sint v)) |

  constantAsStamp - = IllegalStamp

— Define when a runtime value is valid for a stamp
fun valid-value :: Value  $\Rightarrow$  Stamp  $\Rightarrow$  bool where
  valid-value (IntVal32 v) (IntegerStamp b l h) = ((b=32  $\vee$  b=16  $\vee$  b=8  $\vee$  b=1)  $\wedge$ 
    (sint v  $\geq$  l)  $\wedge$  (sint v  $\leq$  h)) |
  valid-value (IntVal64 v) (IntegerStamp b l h) = (b=64  $\wedge$  (sint v  $\geq$  l)  $\wedge$  (sint v  $\leq$ 
    h)) |

  valid-value (ObjRef ref) (ObjectStamp klass exact nonNull alwaysNull) =
    ((alwaysNull  $\longrightarrow$  ref = None)  $\wedge$  (ref=None  $\longrightarrow$   $\neg$  nonNull)) |
  valid-value stamp val = False

fun compatible :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  bool where
  compatible (IntegerStamp b1 - -) (IntegerStamp b2 - -) = (b1 = b2) |
  compatible (VoidStamp) (VoidStamp) = True |
  compatible - - = False

fun stamp-under :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  bool where
  stamp-under x y = ((stpi-upper x) < (stpi-lower y))

— The most common type of stamp within the compiler (apart from the Void-
Stamp) 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))

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

definition as-set :: IRGraph  $\Rightarrow$  (ID  $\times$  (IRNode  $\times$  Stamp)) set where
  as-set g = {(n, kind g n, stamp g n) | 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 ≤ 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 = $\{nid \in \text{dom } g . \nexists s. g \text{ nid} = (\text{Some } (\text{NoNode}, s))\}$

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

lemma *dom-eq*:
assumes $\forall x \in \text{set } xs. \text{fst } (\text{snd } x) \neq \text{NoNode}$
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]*: $\text{ids } (\text{irgraph } m) = \text{set } (\text{map fst } (\text{no-node } m))$
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]*: $\text{Rep-IRGraph } (\text{irgraph } m) = \text{map-of } (\text{no-node } m)$
using *Abs-IRGraph-inverse*
by (simp add: *irgraph.rep-eq*)

— Get the inputs set of a given node ID

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

— Get the successor set of a given node ID

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

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

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

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

fun *usages* :: *IRGraph* \Rightarrow *ID* \Rightarrow *ID* set **where**

```

  usages g nid = {i. i ∈ ids g ∧ nid ∈ inputs g i}
fun successor-edges :: IRGraph ⇒ ID rel where
  successor-edges g = (⋃ i ∈ ids g. {(i,j)|j . j ∈ (succ g i)})
fun predecessors :: IRGraph ⇒ ID ⇒ ID set where
  predecessors g nid = {i. i ∈ ids g ∧ nid ∈ succ g i}
fun nodes-of :: IRGraph ⇒ (IRNode ⇒ bool) ⇒ ID set where
  nodes-of g sel = {nid ∈ ids g . sel (kind g nid)}
fun edge :: (IRNode ⇒ 'a) ⇒ ID ⇒ IRGraph ⇒ 'a where
  edge sel nid g = sel (kind g nid)

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

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

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

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

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

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

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

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

lemma [simp]: finite (dom g) ⟶ ids (Abs-IRGraph g) = {nid ∈ dom g . ∄ s. g

```

```

nid = Some (NoNode, s)}
using ids.rep-eq by simp

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

lemma [simp]: finite (dom g) → stamp (Abs-IRGraph g) = (λx . (case g x of
None ⇒ IllegalStamp | Some n ⇒ 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) = (λnid. (case (map-of (no-node g)) nid of None
⇒ NoNode | Some n ⇒ fst n))
using irgraph.rep-eq kind.transfer kind.rep-eq by auto

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

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

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

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

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

```

```

    case True
    then show ?thesis
      by (simp add: add-node.rep-eq kind.rep-eq)
next
    case False
    then show ?thesis
      by (simp add: kind.rep-eq add-node.rep-eq stamp.rep-eq)
qed

```

lemma *remove-node-lookup*:

```

  gup = remove-node nid g  $\longrightarrow$  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* \Rightarrow *ID* \Rightarrow *IRGraph* \Rightarrow *ID option* **where**

nextEdge seen nid g =
 (let *nids* = (filter (λ *nid'*. *nid'* \notin *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* \Rightarrow *ID* \Rightarrow *ID option* **where**

pred g nid = (case *kind g nid* of
 (MergeNode *ends* -) \Rightarrow Some (hd *ends*) |
 - \Rightarrow
 (if *IRGraph.predecessors g nid* = {}
 then None else
 Some (hd (sorted-list-of-set (*IRGraph.predecessors g nid*)))
)
)

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

type-synonym *'a TraversalState* = (*ID* \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

$\llbracket \text{kind } g \text{ nid} = \text{BeginNode } \text{nid}' ;$

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

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

$\text{analysis}' = \text{sa } (\text{nid}, \text{seen}, \text{analysis}) \rrbracket$
 $\implies \text{Step } \text{sa } g (\text{nid}, \text{seen}, \text{analysis}) (\text{Some } (\text{nid}', \text{seen}', \text{analysis}')) \mid$

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

$\llbracket \text{kind } g \text{ nid} = \text{EndNode};$

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

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

$\text{analysis}' = \text{sa } (\text{nid}, \text{seen}, \text{analysis}) \rrbracket$
 $\implies \text{Step } \text{sa } g (\text{nid}, \text{seen}, \text{analysis}) (\text{Some } (\text{nid}', \text{seen}', \text{analysis}')) \mid$

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

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

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

$\text{Some } \text{nid}' = \text{nextEdge } \text{seen}' \text{ nid } g;$

$\text{analysis}' = \text{sa } (\text{nid}, \text{seen}, \text{analysis}) \rrbracket$
 $\implies \text{Step } \text{sa } g (\text{nid}, \text{seen}, \text{analysis}) (\text{Some } (\text{nid}', \text{seen}', \text{analysis}')) \mid$

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

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

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

$\text{None} = \text{nextEdge } \text{seen}' \text{ nid } g \rrbracket$
 $\implies \text{Step } \text{sa } g (\text{nid}, \text{seen}, \text{analysis}) \text{None} \mid$

— We've already seen this node, give back None
 $\llbracket nid \in seen \rrbracket \implies Step\ sa\ g\ (nid, seen, analysis)\ None$

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *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* \rightarrow *ID*) **where**
find-ref-nodes *g* = *map-of*
 (*map* ($\lambda n. (n, ir-ref\ (kind\ g\ n))$) (*filter* ($\lambda id. is-RefNode\ (kind\ g\ id)$) (*sorted-list-of-set*
 (*ids* *g*))))

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

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

inductive *reachables* :: *IRGraph* \Rightarrow *ID list* \Rightarrow *ID set* \Rightarrow *ID set* \Rightarrow *bool* **where**
reachables *g* [] {} {} |
 $\llbracket None = find-next\ to-see\ seen \rrbracket \implies reachables\ g\ to-see\ seen\ seen\ |$
 $\llbracket Some\ n = find-next\ to-see\ seen;$
 $node = kind\ g\ n;$
 $new = (inputs-of\ node)\ @\ (successors-of\ node);$
 $reachables\ g\ (to-see\ @\ new)\ (\{n\} \cup seen)\ seen' \rrbracket \implies reachables\ g\ to-see\ seen$
 $seen'$

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

reachables .

inductive *nodeEq* :: (*ID* \rightarrow *ID*) \Rightarrow *IRGraph* \Rightarrow *ID* \Rightarrow *IRGraph* \Rightarrow *ID* \Rightarrow *bool*
where
 $\llbracket kind\ g1\ n1 = RefNode\ ref; nodeEq\ m\ g1\ ref\ g2\ n2 \rrbracket \implies nodeEq\ m\ g1\ n1\ g2\ n2\ |$
 $\llbracket x = kind\ g1\ n1;$
 $y = kind\ g2\ n2;$

```

    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 ]
     $\Rightarrow$  nodeEq m g1 n1 g2 n2

code-pred [show-modes] nodeEq .

fun diffNodesGraph :: IRGraph  $\Rightarrow$  IRGraph  $\Rightarrow$  ID set where
diffNodesGraph g1 g2 = (let refNodes = find-ref-nodes g1 in
  { n . n  $\in$  Predicate.the (reachables-i-i-i-o g1 [0] {})  $\wedge$  (case refNodes n of Some
-  $\Rightarrow$  False | -  $\Rightarrow$  True)  $\wedge$   $\neg$ (nodeEq refNodes g1 n g2 n)})

fun diffNodesInfo :: IRGraph  $\Rightarrow$  IRGraph  $\Rightarrow$  (ID  $\times$  IRNode  $\times$  IRNode) set where
diffNodesInfo g1 g2 = {(nid, kind g1 nid, kind g2 nid) | nid . nid  $\in$  diffNodesGraph
g1 g2}

fun eqGraph :: IRGraph  $\Rightarrow$  IRGraph  $\Rightarrow$  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 calculated 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  $\Rightarrow$  Value
type-synonym Params = Value list

```

```

definition new-map-state :: MapState where
  new-map-state = ( $\lambda x$ ..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  $\Rightarrow$  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: all integer calculations are done as 32 or 64 bit calculations. Most operators have the same output bits as their inputs. But the following *fixed₃₂* 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) =
  (if op ∈ normal-unary
   then unrestricted-stamp (IntegerStamp (if b=64 then 64 else 32) lo hi)
   else unrestricted-stamp (IntegerStamp (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 ∧ b1=64
    then unrestricted-stamp (IntegerStamp 64 lo1 hi1)
    else unrestricted-stamp (IntegerStamp 32 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)

```

```

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

```

$\llbracket \text{value} \neq \text{UndefVal} \rrbracket \implies \text{not-undef-or-fail value value}$

notation (*latex output*)
not-undef-or-fail ($- = -$)

inductive

evaltree :: *MapState* \Rightarrow *Params* \Rightarrow *IRExpr* \Rightarrow *Value* \Rightarrow *bool* ($[-,-] \vdash - \mapsto -$ 55)

for *m p* **where**

ConstantExpr:

$\llbracket \text{valid-value } c \text{ (constantAsStamp } c) \rrbracket$
 $\implies [m,p] \vdash (\text{ConstantExpr } c) \mapsto c \mid$

ParameterExpr:

$\llbracket i < \text{length } p; \text{valid-value } (p!i) \text{ } s \rrbracket$
 $\implies [m,p] \vdash (\text{ParameterExpr } i \text{ } s) \mapsto p!i \mid$

ConditionalExpr:

$\llbracket [m,p] \vdash ce \mapsto \text{cond};$
 $\text{branch} = (\text{if val-to-bool cond then } te \text{ else } fe);$
 $[m,p] \vdash \text{branch} \mapsto v;$
 $v \neq \text{UndefVal} \rrbracket$
 $\implies [m,p] \vdash (\text{ConditionalExpr } ce \text{ } te \text{ } fe) \mapsto v \mid$

UnaryExpr:

$\llbracket [m,p] \vdash xe \mapsto v;$
 $\text{result} = (\text{unary-eval op } v);$
 $\text{result} \neq \text{UndefVal} \rrbracket$
 $\implies [m,p] \vdash (\text{UnaryExpr op } xe) \mapsto \text{result} \mid$

BinaryExpr:

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

LeafExpr:

$\llbracket \text{val} = m \text{ } n;$
 $\text{valid-value val } s \rrbracket$
 $\implies [m,p] \vdash \text{LeafExpr } n \text{ } s \mapsto \text{val}$

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ 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] ⊢ [] ↦L [] |

    EvalCons:
    [[m,p] ⊢ x ↦ xval;
     [m,p] ⊢ yy ↦L yyval]
    ⇒ [m,p] ⊢ (x#yy) ↦L (xval#yyval)

code-pred (modes: i ⇒ i ⇒ i ⇒ o ⇒ bool as evalTs)
  evaltrees .

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

values {v. evaltree new-map-state [IntVal32 5] sq-param0 v}

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

```

6.4 Data-flow Tree Refinement

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

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

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

lemma *equivp equiv-exprs*
apply (auto simp add: equivp-def equiv-exprs-def)
by (metis equiv-exprs-def)+

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

instantiation IRExp :: preorder **begin**

notation *less-eq* (**infix** ⊑ 65)

definition

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

definition

lt-expr-def [simp]:
 $(e_1 < e_2) \longleftrightarrow (e_1 \leq e_2 \wedge \neg (e_1 \dot{=} e_2))$

instance proof

fix $x y z :: IRExpr$
show $x < y \longleftrightarrow x \leq y \wedge \neg (y \leq x)$ **by** (*simp add: equiv-exprs-def; auto*)
show $x \leq x$ **by** *simp*
show $x \leq y \implies y \leq z \implies x \leq z$ **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

IRTreeEval

begin

6.5.1 Deterministic Data-flow Evaluation

lemma *evalDet*:

$[m,p] \vdash e \mapsto v_1 \implies$
 $[m,p] \vdash e \mapsto v_2 \implies$
 $v_1 = v_2$
apply (*induction arbitrary: v2 rule: evaltree.induct*)
by (*elim EvalTreeE; auto*)+

lemma *evalAllDet*:

$[m,p] \vdash e \mapsto_L v1 \implies$
 $[m,p] \vdash e \mapsto_L v2 \implies$
 $v1 = v2$
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: *isIntVal32*, *isIntVal64* and the more general *isIntVal*.


```

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
  by (metis (full-types) bool-to-val.simps is-IntVal32-def)+

lemma int-stamp32:
  assumes i: is-IntVal32 v
  shows is-IntegerStamp (constantAsStamp v)
  using i unfolding is-IntegerStamp-def is-IntVal32-def by auto

lemma int-stamp64:
  assumes i: is-IntVal64 v
  shows is-IntegerStamp (constantAsStamp v)
  using i unfolding is-IntegerStamp-def is-IntVal64-def by auto

lemma int-stamp-both:
  assumes i: is-IntVal v
  shows is-IntegerStamp (constantAsStamp v)
  using i unfolding is-IntVal-def is-IntegerStamp-def
  using int-stamp32 int-stamp64 is-IntegerStamp-def by auto

lemma validDefIntConst:
  assumes v  $\neq$  UndefVal
  assumes is-IntegerStamp (constantAsStamp v)
  shows valid-value v (constantAsStamp v)
  using assms by (cases v; auto)

lemma validIntConst:
  assumes i: is-IntVal v
  shows valid-value v (constantAsStamp v)

```

using *i int-stamp-both is-IntVal-def validDefIntConst* **by** *auto*

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-int1[elim]*:
shows *valid-value val (IntegerStamp 1 lo hi) ⇒*
(∃ v. val = IntVal32 v)
apply (*rule val-to-bool.cases[of val]*)
using *Value.distinct* **by** *simp+*

lemma *valid-int8[elim]*:
shows *valid-value val (IntegerStamp 8 l h) ⇒*
(∃ v. val = IntVal32 v)
apply (*rule val-to-bool.cases[of val]*)
using *Value.distinct* **by** *simp+*

lemma *valid-int16[elim]*:
shows *valid-value val (IntegerStamp 16 l h) ⇒*
(∃ v. val = IntVal32 v)
apply (*rule val-to-bool.cases[of val]*)
using *Value.distinct* **by** *simp+*

lemma *valid-int32[elim]*:
shows *valid-value val (IntegerStamp 32 l h) ⇒*
(∃ v. val = IntVal32 v)
apply (*rule val-to-bool.cases[of val]*)
using *Value.distinct* **by** *simp+*

lemma *valid-int64[elim]*:
shows *valid-value val (IntegerStamp 64 l h) ⇒*

```

    (∃ v. val = IntVal64 v)
  apply (rule val-to-bool.cases[of val])
  using Value.distinct by simp+

lemmas valid-value-elim =
  valid-VoidStamp
  valid-ObjStamp
  valid-int1
  valid-int8
  valid-int16
  valid-int32
  valid-int64

lemma evaltree-not-undef:
  fixes m p e v
  shows ([m,p] ⊢ e ↦ v) ⇒ v ≠ UndefVal
  apply (induction rule: evaltree.induct)
  using valid-not-undef by auto

lemma leafint32:
  assumes ev: [m,p] ⊢ LeafExpr i (IntegerStamp 32 lo hi) ↦ val
  shows ∃ v. val = (IntVal32 v)

proof -
  have valid-value val (IntegerStamp 32 lo hi)
    using ev by (rule LeafExprE; simp)
  then show ?thesis by auto
qed

lemma leafint64:
  assumes ev: [m,p] ⊢ LeafExpr i (IntegerStamp 64 lo hi) ↦ val
  shows ∃ v. val = (IntVal64 v)

proof -
  have valid-value val (IntegerStamp 64 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 valid32 [simp]:
  assumes valid-value val (IntegerStamp 32 lo hi)
  shows ∃ v. (val = (IntVal32 v) ∧ lo ≤ sint v ∧ sint v ≤ hi)

```

using *assms valid-int32* **by** *force*

lemma *valid64* [*simp*]:

assumes *valid-value val* (*IntegerStamp* 64 *lo hi*)

shows $\exists v. (val = (IntVal64\ v) \wedge lo \leq sint\ v \wedge sint\ v \leq hi)$

using *assms valid-int64* **by** *force*

lemma *valid32or64*:

assumes *valid-value x* (*IntegerStamp* *b lo hi*)

shows $(\exists v1. (x = IntVal32\ v1)) \vee (\exists v2. (x = IntVal64\ v2))$

using *valid32 valid64 assms valid-value.elims(2)* **by** *blast*

lemma *valid32or64-both*:

assumes *valid-value x* (*IntegerStamp* *b lox hix*)

and *valid-value y* (*IntegerStamp* *b loy hiy*)

shows $(\exists v1\ v2. x = IntVal32\ v1 \wedge y = IntVal32\ v2) \vee (\exists v3\ v4. x = IntVal64\ v3 \wedge y = IntVal64\ v4)$

using *assms valid32or64 valid32* **by** (*metis valid-int64 valid-value.simps(2)*)

6.5.4 Example Data-flow Optimisations

lemma *a0a-helper* [*simp*]:

assumes *a: valid-value v* (*IntegerStamp* 32 *lo hi*)

shows *intval-add v* (*IntVal32* 0) = *v*

proof –

obtain *v32* :: *int32* **where** *v* = (*IntVal32* *v32*) **using** *a valid32* **by** *blast*

then show *?thesis* **by** *simp*

qed

lemma *a0a*: (*BinaryExpr* *BinAdd* (*LeafExpr* 1 *default-stamp*) (*ConstantExpr* (*IntVal32* 0)))

\geq (*LeafExpr* 1 *default-stamp*)

by (*auto simp add: evaltree.LeafExpr*)

lemma *xyx-y-helper* [*simp*]:

assumes *valid-value x* (*IntegerStamp* 32 *lox hix*)

assumes *valid-value y* (*IntegerStamp* 32 *loy hiy*)

shows *intval-add x* (*intval-sub y x*) = *y*

proof –

obtain *x32* :: *int32* **where** *x* = (*IntVal32* *x32*) **using** *assms valid32* **by** *blast*

obtain *y32* :: *int32* **where** *y* = (*IntVal32* *y32*) **using** *assms valid32* **by** *blast*

show *?thesis* **using** *x y* **by** *simp*

qed

lemma *xyx-y*:

(*BinaryExpr* *BinAdd*

(*LeafExpr* *x* (*IntegerStamp* 32 *lox hix*))

```

(BinaryExpr BinSub
  (LeafExpr y (IntegerStamp 32 loy hiy))
  (LeafExpr x (IntegerStamp 32 lox hix)))
≥ (LeafExpr y (IntegerStamp 32 loy hiy))
by (auto simp add: LeafExpr)

```

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) ≥ (UnaryExpr op e')
using UnaryExpr assms by auto

```

lemma *mono-binary*:

```

assumes  $x \geq x'$ 
assumes  $y \geq y'$ 
shows (BinaryExpr op x y) ≥ (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 ≠ VoidStamp
using valid-value.simps
using assms(2) by force

```

lemma *compatible-trans*:

```

compatible x y ∧ compatible y z ⇒ compatible x z
by (smt (verit, best) compatible.elims(2) compatible.simps(1))

```

lemma *compatible-refl*:

```

compatible x y ⇒ 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  $b: branch = (if\ val\text{-}to\text{-}bool\ cond\ then\ te\ else\ fe)$ 
  define  $branch'$  where  $b': branch' = (if\ val\text{-}to\text{-}bool\ cond\ then\ te'\ else\ fe')$ 
  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_eeval* / *unary_eeval* level, simply by saying *unfoldingunfold_eevaltree*.

lemma *unfold-valid32* [*simp*]:
 $valid\text{-}value\ y\ (constantAsStamp\ (IntVal32\ v)) = (y = IntVal32\ v)$
by (*induction y; auto dest: signed-word-eqI*)

lemma *unfold-valid64* [*simp*]:
 $valid\text{-}value\ y\ (constantAsStamp\ (IntVal64\ v)) = (y = IntVal64\ v)$
by (*induction y; auto dest: signed-word-eqI*)

lemma *unfold-const*:
shows $([m,p] \vdash ConstantExpr\ c \mapsto v) = (valid\text{-}value\ v\ (constantAsStamp\ c) \wedge v = c)$
by *blast*

corollary *unfold-const32*:
shows $([m,p] \vdash ConstantExpr\ (IntVal32\ c) \mapsto v) = (v = IntVal32\ c)$
using *unfold-valid32* **by** *blast*

corollary *unfold-const64*:
shows $([m,p] \vdash ConstantExpr\ (IntVal64\ c) \mapsto v) = (v = IntVal64\ c)$
using *unfold-valid64* **by** *blast*

```

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

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

```

```

lemmas unfold-evaltree =
  unfold-binary
  unfold-unary
  unfold-const32
  unfold-const64
  unfold-valid32
  unfold-valid64

```

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* \Rightarrow (*IRNode* \times *Stamp*) \Rightarrow *ID option* **where**
find-node-and-stamp *g* (*n,s*) =
find ($\lambda i.$ *kind g i* = *n* \wedge *stamp g i* = *s*) (*sorted-list-of-set*(*ids g*))
export-code *find-node-and-stamp*

fun *is-preevaluated* :: *IRNode* \Rightarrow *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* \Rightarrow *ID* \Rightarrow *IRExpr* \Rightarrow *bool* (- \vdash - \simeq - 55)
for *g* **where**

ConstantNode:
 $\llbracket \text{kind } g \text{ } n = \text{ConstantNode } c \rrbracket$
 $\implies g \vdash n \simeq (\text{ConstantExpr } c) \mid$

ParameterNode:
 $\llbracket \text{kind } g \text{ } n = \text{ParameterNode } i;$
 $\text{stamp } g \text{ } n = s \rrbracket$
 $\implies g \vdash n \simeq (\text{ParameterExpr } i \text{ } s) \mid$

ConditionalNode:
 $\llbracket \text{kind } g \text{ } n = \text{ConditionalNode } c \text{ } t \text{ } f;$
 $g \vdash c \simeq ce;$
 $g \vdash t \simeq te;$
 $g \vdash f \simeq fe \rrbracket$
 $\implies g \vdash n \simeq (\text{ConditionalExpr } ce \text{ } te \text{ } fe) \mid$

AbsNode:
 $\llbracket \text{kind } g \text{ } n = \text{AbsNode } x;$
 $g \vdash x \simeq xe \rrbracket$
 $\implies g \vdash n \simeq (\text{UnaryExpr } \text{UnaryAbs } xe) \mid$

NotNode:

$\llbracket \text{kind } g \ n = \text{NotNode } x;$
 $g \vdash x \simeq xe \rrbracket$
 $\implies g \vdash n \simeq (\text{UnaryExpr } \text{UnaryNot } xe) \mid$

NegateNode:

$\llbracket \text{kind } g \ n = \text{NegateNode } x;$
 $g \vdash x \simeq xe \rrbracket$
 $\implies g \vdash n \simeq (\text{UnaryExpr } \text{UnaryNeg } xe) \mid$

LogicNegationNode:

$\llbracket \text{kind } g \ n = \text{LogicNegationNode } x;$
 $g \vdash x \simeq xe \rrbracket$
 $\implies g \vdash n \simeq (\text{UnaryExpr } \text{UnaryLogicNegation } xe) \mid$

AddNode:

$\llbracket \text{kind } g \ n = \text{AddNode } x \ y;$
 $g \vdash x \simeq xe;$
 $g \vdash y \simeq ye \rrbracket$
 $\implies g \vdash n \simeq (\text{BinaryExpr } \text{BinAdd } xe \ ye) \mid$

MulNode:

$\llbracket \text{kind } g \ n = \text{MulNode } x \ y;$
 $g \vdash x \simeq xe;$
 $g \vdash y \simeq ye \rrbracket$
 $\implies g \vdash n \simeq (\text{BinaryExpr } \text{BinMul } xe \ ye) \mid$

SubNode:

$\llbracket \text{kind } g \ n = \text{SubNode } x \ y;$
 $g \vdash x \simeq xe;$
 $g \vdash y \simeq ye \rrbracket$
 $\implies g \vdash n \simeq (\text{BinaryExpr } \text{BinSub } xe \ ye) \mid$

AndNode:

$\llbracket \text{kind } g \ n = \text{AndNode } x \ y;$
 $g \vdash x \simeq xe;$
 $g \vdash y \simeq ye \rrbracket$
 $\implies g \vdash n \simeq (\text{BinaryExpr } \text{BinAnd } xe \ ye) \mid$

OrNode:

$\llbracket \text{kind } g \ n = \text{OrNode } x \ y;$
 $g \vdash x \simeq xe;$
 $g \vdash y \simeq ye \rrbracket$
 $\implies g \vdash n \simeq (\text{BinaryExpr } \text{BinOr } xe \ ye) \mid$

XorNode:

$\llbracket \text{kind } g \ n = \text{XorNode } x \ y;$
 $g \vdash x \simeq xe;$

$$g \vdash y \simeq ye] \\ \implies g \vdash n \simeq (\text{BinaryExpr BinXor } xe \ ye) \mid$$

ShortCircuitOrNode:

$$[kind \ g \ n = \text{ShortCircuitOrNode } x \ y; \\ g \vdash x \simeq xe; \\ g \vdash y \simeq ye] \\ \implies g \vdash n \simeq (\text{BinaryExpr BinShortCircuitOr } xe \ ye) \mid$$

LeftShiftNode:

$$[kind \ g \ n = \text{LeftShiftNode } x \ y; \\ g \vdash x \simeq xe; \\ g \vdash y \simeq ye] \\ \implies g \vdash n \simeq (\text{BinaryExpr BinLeftShift } xe \ ye) \mid$$

RightShiftNode:

$$[kind \ g \ n = \text{RightShiftNode } x \ y; \\ g \vdash x \simeq xe; \\ g \vdash y \simeq ye] \\ \implies g \vdash n \simeq (\text{BinaryExpr BinRightShift } xe \ ye) \mid$$

UnsignedRightShiftNode:

$$[kind \ g \ n = \text{UnsignedRightShiftNode } x \ y; \\ g \vdash x \simeq xe; \\ g \vdash y \simeq ye] \\ \implies g \vdash n \simeq (\text{BinaryExpr BinURightShift } xe \ ye) \mid$$

IntegerBelowNode:

$$[kind \ g \ n = \text{IntegerBelowNode } x \ y; \\ g \vdash x \simeq xe; \\ g \vdash y \simeq ye] \\ \implies g \vdash n \simeq (\text{BinaryExpr BinIntegerBelow } xe \ ye) \mid$$

IntegerEqualsNode:

$$[kind \ g \ n = \text{IntegerEqualsNode } x \ y; \\ g \vdash x \simeq xe; \\ g \vdash y \simeq ye] \\ \implies g \vdash n \simeq (\text{BinaryExpr BinIntegerEquals } xe \ ye) \mid$$

IntegerLessThanNode:

$$[kind \ g \ n = \text{IntegerLessThanNode } x \ y; \\ g \vdash x \simeq xe; \\ g \vdash y \simeq ye] \\ \implies g \vdash n \simeq (\text{BinaryExpr BinIntegerLessThan } xe \ ye) \mid$$

NarrowNode:

$$[kind \ g \ n = \text{NarrowNode } inputBits \ resultBits \ x; \\ g \vdash x \simeq xe]$$

$\Rightarrow g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryNarrow } \text{inputBits } \text{resultBits}) \text{ xe}) \mid$

SignExtendNode:

$\llbracket \text{kind } g \text{ } n = \text{SignExtendNode } \text{inputBits } \text{resultBits } x; \\ g \vdash x \simeq xe \rrbracket \\ \Rightarrow g \vdash n \simeq (\text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \text{ xe}) \mid$

ZeroExtendNode:

$\llbracket \text{kind } g \text{ } n = \text{ZeroExtendNode } \text{inputBits } \text{resultBits } x; \\ g \vdash x \simeq xe \rrbracket \\ \Rightarrow g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits}) \text{ xe}) \mid$

LeafNode:

$\llbracket \text{is-preevaluated } (\text{kind } g \text{ } n); \\ \text{stamp } g \text{ } n = s \rrbracket \\ \Rightarrow g \vdash n \simeq (\text{LeafExpr } n \text{ } s) \mid$

RefNode:

$\llbracket \text{kind } g \text{ } n = \text{RefNode } n'; \\ g \vdash n' \simeq e \rrbracket \\ \Rightarrow g \vdash n \simeq e$

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ 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:

$\llbracket g \vdash x \simeq xe; \\ g \vdash xs \simeq_L xse \rrbracket \\ \Rightarrow g \vdash x \# xs \simeq_L xe \# xse$

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ 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 $\{t. \text{eg2-sq} \vdash 4 \simeq t\}$

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* \Rightarrow *ID* \Rightarrow *ID* \Rightarrow *IRNode* **where**

```

bin-node BinAdd x y = AddNode x y |
bin-node BinMul x y = MulNode x y |
bin-node BinSub x y = SubNode x y |
bin-node BinAnd x y = AndNode x y |
bin-node BinOr x y = OrNode x y |
bin-node BinXor x y = XorNode x y |
bin-node BinShortCircuitOr x y = ShortCircuitOrNode x y |
bin-node BinLeftShift x y = LeftShiftNode x y |
bin-node BinRightShift x y = RightShiftNode x y |
bin-node BinURightShift x y = UnsignedRightShiftNode x y |
bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
bin-node BinIntegerLessThan x y = IntegerLessThanNode x y |
bin-node BinIntegerBelow x y = IntegerBelowNode x y

```

fun choose-32-64 :: *int* \Rightarrow *int64* \Rightarrow *Value* **where**

```

choose-32-64 bits val =
  (if bits = 32
   then (IntVal32 (ucast val))
   else (IntVal64 (val)))

```

inductive fresh-id :: *IRGraph* \Rightarrow *ID* \Rightarrow *bool* **where**

```

n  $\notin$  ids g  $\implies$  fresh-id g n

```

code-pred fresh-id .

fun get-fresh-id :: *IRGraph* \Rightarrow *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* \Rightarrow *IRExpr* \Rightarrow (*IRGraph* \times *ID*) \Rightarrow *bool* (- \oplus - \rightsquigarrow - 55)
where

ConstantNodeSame:

$\llbracket \text{find-node-and-stamp } g \text{ (ConstantNode } c, \text{ constantAsStamp } c) = \text{Some } n \rrbracket$
 $\implies g \oplus (\text{ConstantExpr } c) \rightsquigarrow (g, n) \mid$

ConstantNodeNew:

$\llbracket \text{find-node-and-stamp } g \text{ (ConstantNode } c, \text{ constantAsStamp } c) = \text{None};$
 $n = \text{get-fresh-id } g;$
 $g' = \text{add-node } n \text{ (ConstantNode } c, \text{ constantAsStamp } c) \text{ } g \rrbracket$
 $\implies g \oplus (\text{ConstantExpr } c) \rightsquigarrow (g', n) \mid$

ParameterNodeSame:

$\llbracket \text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{Some } n \rrbracket$
 $\implies g \oplus (\text{ParameterExpr } i \text{ } s) \rightsquigarrow (g, n) \mid$

ParameterNodeNew:

$\llbracket \text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{None};$
 $n = \text{get-fresh-id } g;$
 $g' = \text{add-node } n \text{ (ParameterNode } i, s) \text{ } g \rrbracket$
 $\implies g \oplus (\text{ParameterExpr } i \text{ } s) \rightsquigarrow (g', n) \mid$

ConditionalNodeSame:

$\llbracket g \oplus ce \rightsquigarrow (g2, c);$
 $g2 \oplus te \rightsquigarrow (g3, t);$
 $g3 \oplus fe \rightsquigarrow (g4, f);$
 $s' = \text{meet (stamp } g4 \text{ } t) \text{ (stamp } g4 \text{ } f);$
 $\text{find-node-and-stamp } g4 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{Some } n \rrbracket$
 $\implies g \oplus (\text{ConditionalExpr } ce \text{ } te \text{ } fe) \rightsquigarrow (g4, n) \mid$

ConditionalNodeNew:

$\llbracket g \oplus ce \rightsquigarrow (g2, c);$
 $g2 \oplus te \rightsquigarrow (g3, t);$
 $g3 \oplus fe \rightsquigarrow (g4, f);$
 $s' = \text{meet (stamp } g4 \text{ } t) \text{ (stamp } g4 \text{ } f);$
 $\text{find-node-and-stamp } g4 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{None};$
 $n = \text{get-fresh-id } g4;$
 $g' = \text{add-node } n \text{ (ConditionalNode } c \text{ } t \text{ } f, s') \text{ } g4 \rrbracket$
 $\implies g \oplus (\text{ConditionalExpr } ce \text{ } te \text{ } fe) \rightsquigarrow (g', n) \mid$

UnaryNodeSame:

$\llbracket g \oplus xe \rightsquigarrow (g2, x);$
 $s' = \text{stamp-unary op (stamp } g2 \text{ } x);$
 $\text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{Some } n \rrbracket$

$$\implies g \oplus (\text{UnaryExpr } op \ xe) \rightsquigarrow (g2, n) \mid$$

UnaryNodeNew:

$$\begin{aligned} & \llbracket g \oplus xe \rightsquigarrow (g2, x); \\ & \quad s' = \text{stamp-unary } op \ (\text{stamp } g2 \ x); \\ & \quad \text{find-node-and-stamp } g2 \ (\text{unary-node } op \ x, s') = \text{None}; \\ & \quad n = \text{get-fresh-id } g2; \\ & \quad g' = \text{add-node } n \ (\text{unary-node } op \ x, s') \ g2 \rrbracket \\ & \implies g \oplus (\text{UnaryExpr } op \ xe) \rightsquigarrow (g', n) \mid \end{aligned}$$

BinaryNodeSame:

$$\begin{aligned} & \llbracket g \oplus xe \rightsquigarrow (g2, x); \\ & \quad g2 \oplus ye \rightsquigarrow (g3, y); \\ & \quad s' = \text{stamp-binary } op \ (\text{stamp } g3 \ x) \ (\text{stamp } g3 \ y); \\ & \quad \text{find-node-and-stamp } g3 \ (\text{bin-node } op \ x \ y, s') = \text{Some } n \rrbracket \\ & \implies g \oplus (\text{BinaryExpr } op \ xe \ ye) \rightsquigarrow (g3, n) \mid \end{aligned}$$

BinaryNodeNew:

$$\begin{aligned} & \llbracket g \oplus xe \rightsquigarrow (g2, x); \\ & \quad g2 \oplus ye \rightsquigarrow (g3, y); \\ & \quad s' = \text{stamp-binary } op \ (\text{stamp } g3 \ x) \ (\text{stamp } g3 \ y); \\ & \quad \text{find-node-and-stamp } g3 \ (\text{bin-node } op \ x \ y, s') = \text{None}; \\ & \quad n = \text{get-fresh-id } g3; \\ & \quad g' = \text{add-node } n \ (\text{bin-node } op \ x \ y, s') \ g3 \rrbracket \\ & \implies g \oplus (\text{BinaryExpr } op \ xe \ ye) \rightsquigarrow (g', n) \mid \end{aligned}$$

AllLeafNodes:

$$\begin{aligned} & \llbracket \text{stamp } g \ n = s; \\ & \quad \text{is-preevaluated } (\text{kind } g \ n) \rrbracket \\ & \implies g \oplus (\text{LeafExpr } n \ s) \rightsquigarrow (g, n) \end{aligned}$$

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

$$\begin{array}{c}
\frac{\text{find-node-and-stamp } g \text{ (ConstantNode } c, \text{ constantAsStamp } c) = \text{Some } n}{g \oplus \text{ConstantExpr } c \rightsquigarrow (g, n)} \\
\\
\frac{\begin{array}{c} \text{find-node-and-stamp } g \text{ (ConstantNode } c, \text{ constantAsStamp } c) = \text{None} \\ n = \text{get-fresh-id } g \\ g' = \text{add-node } n \text{ (ConstantNode } c, \text{ constantAsStamp } c) \text{ } g \end{array}}{g \oplus \text{ConstantExpr } c \rightsquigarrow (g', n)} \\
\\
\frac{\text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{Some } n}{g \oplus \text{ParameterExpr } i \text{ } s \rightsquigarrow (g, n)} \\
\\
\frac{\begin{array}{c} \text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{None} \\ n = \text{get-fresh-id } g \quad g' = \text{add-node } n \text{ (ParameterNode } i, s) \text{ } g \end{array}}{g \oplus \text{ParameterExpr } i \text{ } s \rightsquigarrow (g', n)} \\
\\
\frac{\begin{array}{c} g \oplus ce \rightsquigarrow (g2, c) \quad g2 \oplus te \rightsquigarrow (g3, t) \\ g3 \oplus fe \rightsquigarrow (g4, f) \quad s' = \text{meet (stamp } g4 \text{ } t) \text{ (stamp } g4 \text{ } f) \\ \text{find-node-and-stamp } g4 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{Some } n \end{array}}{g \oplus \text{ConditionalExpr } ce \text{ } te \text{ } fe \rightsquigarrow (g4, n)} \\
\\
\frac{\begin{array}{c} g \oplus ce \rightsquigarrow (g2, c) \quad g2 \oplus te \rightsquigarrow (g3, t) \\ g3 \oplus fe \rightsquigarrow (g4, f) \quad s' = \text{meet (stamp } g4 \text{ } t) \text{ (stamp } g4 \text{ } f) \\ \text{find-node-and-stamp } g4 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{None} \\ n = \text{get-fresh-id } g4 \quad g' = \text{add-node } n \text{ (ConditionalNode } c \text{ } t \text{ } f, s') \text{ } g4 \end{array}}{g \oplus \text{ConditionalExpr } ce \text{ } te \text{ } fe \rightsquigarrow (g', n)} \\
\\
\frac{\begin{array}{c} g \oplus xe \rightsquigarrow (g2, x) \\ g2 \oplus ye \rightsquigarrow (g3, y) \quad s' = \text{stamp-binary op (stamp } g3 \text{ } x) \text{ (stamp } g3 \text{ } y) \\ \text{find-node-and-stamp } g3 \text{ (bin-node op } x \text{ } y, s') = \text{Some } n \end{array}}{g \oplus \text{BinaryExpr op } xe \text{ } ye \rightsquigarrow (g3, n)} \\
\\
\frac{\begin{array}{c} g \oplus xe \rightsquigarrow (g2, x) \\ g2 \oplus ye \rightsquigarrow (g3, y) \quad s' = \text{stamp-binary op (stamp } g3 \text{ } x) \text{ (stamp } g3 \text{ } y) \\ \text{find-node-and-stamp } g3 \text{ (bin-node op } x \text{ } y, s') = \text{None} \\ n = \text{get-fresh-id } g3 \quad g' = \text{add-node } n \text{ (bin-node op } x \text{ } y, s') \text{ } g3 \end{array}}{g \oplus \text{BinaryExpr op } xe \text{ } ye \rightsquigarrow (g', n)} \\
\\
\frac{\begin{array}{c} g \oplus xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op (stamp } g2 \text{ } x) \\ \text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{Some } n \end{array}}{g \oplus \text{UnaryExpr op } xe \rightsquigarrow (g2, n)} \\
\\
\frac{\begin{array}{c} g \oplus xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op (stamp } g2 \text{ } x) \\ \text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{None} \\ n = \text{get-fresh-id } g2 \quad g' = \text{add-node } n \text{ (unary-node op } x, s') \text{ } g2 \end{array}}{g \oplus \text{UnaryExpr op } xe \rightsquigarrow (g', n)} \\
\\
\frac{\text{stamp } g \text{ } n = s \quad \text{is-preevaluated (kind } g \text{ } n)}{g \oplus \text{LeafExpr } n \text{ } s \rightsquigarrow (g, n)}
\end{array}$$

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

7.3 Lift Data-flow Tree Semantics

definition *encodeeval* :: *IRGraph* \Rightarrow *MapState* \Rightarrow *Params* \Rightarrow *ID* \Rightarrow *Value* \Rightarrow *bool*
 $([\cdot, \cdot, \cdot] \vdash \cdot \mapsto \cdot \ 50)$
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* \Rightarrow *ID* \Rightarrow *IRExpr* \Rightarrow *bool*
 $(\vdash \cdot \preceq \cdot \ 50)$
where
 $(g \vdash n \preceq e) = (\exists e'. (g \vdash n \simeq e') \wedge (e' \leq e))$

definition *graph-refinement* :: *IRGraph* \Rightarrow *IRGraph* \Rightarrow *bool* **where**
graph-refinement *g1 g2* =
 $((ids\ g_1 \subseteq ids\ g_2) \wedge$
 $(\forall n. n \in ids\ g_1 \longrightarrow (\forall e. (g_1 \vdash n \simeq e) \longrightarrow (g_2 \vdash n \preceq e))))$

lemma *graph-refinement*:

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

7.5 Maximal Sharing

definition *maximal-sharing*:

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

end

7.6 Formedness Properties

theory *Form*

imports

Semantics.TreeToGraph

begin

definition *wf-start* **where**

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

definition *wf-closed* **where**

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

$$\begin{aligned} &inputs\ g\ n \subseteq ids\ g \wedge \\ &succ\ g\ n \subseteq ids\ g \wedge \\ &kind\ g\ n \neq NoNode) \end{aligned}$$

definition *wf-phs* **where**

$$\begin{aligned} wf-phs\ g = & \\ &(\forall\ n \in ids\ g. \\ &\quad is-PhiNode\ (kind\ g\ n) \longrightarrow \\ &\quad length\ (ir-values\ (kind\ g\ n)) \\ &= length\ (ir-ends \\ &\quad (kind\ g\ (ir-merge\ (kind\ g\ n)))))) \end{aligned}$$

definition *wf-ends* **where**

$$\begin{aligned} wf-ends\ g = & \\ &(\forall\ n \in ids\ g . \\ &\quad is-AbstractEndNode\ (kind\ g\ n) \longrightarrow \\ &\quad card\ (usages\ g\ n) > 0) \end{aligned}$$

fun *wf-graph* :: *IRGraph* \Rightarrow *bool* **where**

$$wf-graph\ g = (wf-start\ g \wedge wf-closed\ g \wedge wf-phs\ g \wedge wf-ends\ g)$$

lemmas *wf-folds* =

$$\begin{aligned} &wf-graph.simps \\ &wf-start-def \\ &wf-closed-def \\ &wf-phs-def \\ &wf-ends-def \end{aligned}$$

fun *wf-stamps* :: *IRGraph* \Rightarrow *bool* **where**

$$\begin{aligned} wf-stamps\ g = &(\forall\ n \in ids\ g . \\ &(\forall\ v\ m\ p\ e . (g \vdash n \simeq e) \wedge ([m, p] \vdash e \mapsto v) \longrightarrow valid-value\ v\ (stamp-expr\ e))) \end{aligned}$$

fun *wf-stamp* :: *IRGraph* \Rightarrow (*ID* \Rightarrow *Stamp*) \Rightarrow *bool* **where**

$$\begin{aligned} wf-stamp\ g\ s = &(\forall\ n \in ids\ g . \\ &(\forall\ v\ m\ p\ e . (g \vdash n \simeq e) \wedge ([m, p] \vdash e \mapsto v) \longrightarrow valid-value\ v\ (s\ n))) \end{aligned}$$

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* \Rightarrow *ID* \Rightarrow *bool* **where**

$$\begin{aligned} wf-logic-node-inputs\ g\ n = & \\ &(\forall\ inp \in set\ (inputs-of\ (kind\ g\ n)) . (\forall\ v\ m\ p . ([g, m, p] \vdash inp \mapsto v) \longrightarrow wf-bool \\ &v)) \end{aligned}$$

fun *wf-values* :: *IRGraph* \Rightarrow *bool* **where**

$$wf-values\ g = (\forall\ n \in ids\ g .$$

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

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* \Rightarrow *IRGraph* \Rightarrow *IRGraph* \Rightarrow *bool* **where**

unchanged ns g1 g2 = $(\forall n . n \in ns \longrightarrow$
 $(n \in ids \ g1 \wedge n \in ids \ g2 \wedge kind \ g1 \ n = kind \ g2 \ n \wedge stamp \ g1 \ n = stamp \ g2 \ n))$

fun *changeonly* :: *ID set* \Rightarrow *IRGraph* \Rightarrow *IRGraph* \Rightarrow *bool* **where**

changeonly ns g1 g2 = $(\forall n . n \in ids \ g1 \wedge n \notin ns \longrightarrow$
 $(n \in ids \ g1 \wedge n \in ids \ g2 \wedge kind \ g1 \ n = kind \ g2 \ n \wedge stamp \ g1 \ n = stamp \ g2 \ n))$

lemma *node-unchanged*:

assumes *unchanged ns g1 g2*

assumes *nid* \in *ns*

shows *kind g1 nid* = *kind g2 nid*

using *assms* **by** *auto*

lemma *other-node-unchanged*:

assumes *changeonly ns g1 g2*

assumes *nid* \in *ids g1*

assumes *nid* \notin *ns*

shows *kind g1 nid* = *kind g2 nid*

using *assms*

using *changeonly.simps* **by** *blast*

Some notation for input nodes used

inductive *eval-uses*:: *IRGraph* \Rightarrow *ID* \Rightarrow *ID* \Rightarrow *bool*

for *g* **where**

use0: *nid* \in *ids g*

$\implies eval-uses \ g \ nid \ nid \mid$

```

use-inp:  $nid' \in inputs\ g\ n$ 
 $\implies eval\text{-}uses\ g\ nid\ nid' \mid$ 

use-trans:  $\llbracket eval\text{-}uses\ g\ nid\ nid';$ 
 $eval\text{-}uses\ g\ nid'\ nid'' \rrbracket$ 
 $\implies eval\text{-}uses\ g\ nid\ nid''$ 

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

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

lemma not-in-g-inputs:
  assumes nid  $\notin$  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  $n \in \text{inputs } g \text{ nid}$ 
  shows  $n \in \text{ids } g$ 
  using assms unfolding wf-folds
  using inp-in-g by blast

```

```

lemma kind-unchanged:
  assumes nid  $\in \text{ids } g1$ 
  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  $\in \text{ids } g1$ 
  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  $\in \text{inputs } g1 \text{ nid}$ 
  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'  $\in \text{ids } g$ 
  shows eval-uses g nid nid'  $\longleftrightarrow$  nid'  $\in \text{us}$  (is ?P  $\longleftrightarrow$  ?Q)
  using assms eval-usages.simps
  by (simp add: ids.rep-eq)

```

```

lemma inputs-are-uses:
  assumes nid'  $\in \text{inputs } g \text{ nid}$ 

```

```

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  $\forall \text{nid}' \in (\text{eval-usages } g1 \text{ nid}) . \text{kind } g1 \text{ nid}' = \text{kind } g2 \text{ nid}'$ 
  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.premis(1) ParameterNode.premis(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.premis(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.premis(1) AbsNode.premis(3))

```

```

IRNodes.inputs-of-AbsNode ⟨kind g2 n = AbsNode x⟩ child-member-in child-unchanged
local.wf member-rec(1) rep.AbsNode unchanged.simps)
next
  case (NotNode n x xe)
  then have kind g2 n = NotNode x
    using kind-unchanged
    by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-NotNode
    by (metis NotNode.hyps(1) NotNode.hyps(2) encode-in-ids inputs.simps in-
      puts-are-usages list.set-intros(1))
  then show ?case
    by (metis NotNode.IH NotNode.hyps(1) NotNode.prem(1) NotNode.prem(3)
      IRNodes.inputs-of-NotNode ⟨kind g2 n = NotNode x⟩ child-member-in child-unchanged
      local.wf member-rec(1) rep.NotNode unchanged.simps)
next
  case (NegateNode n x xe)
  then have kind g2 n = NegateNode x
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-NegateNode
    by (metis NegateNode.hyps(1) NegateNode.hyps(2) encode-in-ids inputs.simps
      inputs-are-usages list.set-intros(1))
  then show ?case
    by (metis IRNodes.inputs-of-NegateNode NegateNode.IH NegateNode.hyps(1)
      NegateNode.prem(1) NegateNode.prem(3) ⟨kind g2 n = NegateNode x⟩ child-member-in
      child-unchanged local.wf member-rec(1) rep.NegateNode unchanged.elims(1))
next
  case (LogicNegationNode n x xe)
  then have kind g2 n = LogicNegationNode x
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
    by (metis LogicNegationNode.hyps(1) LogicNegationNode.hyps(2) encode-in-ids
      member-rec(1))
  then show ?case
    by (metis IRNodes.inputs-of-LogicNegationNode LogicNegationNode.IH Logic-
      NegationNode.hyps(1) LogicNegationNode.hyps(2) LogicNegationNode.prem(1) ⟨kind
      g2 n = LogicNegationNode x⟩ child-unchanged encode-in-ids inputs.simps list.set-intros(1)
      local.wf rep.LogicNegationNode)
next
  case (AddNode n x y xe ye)
  then have kind g2 n = AddNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
    by (metis AddNode.hyps(1) AddNode.hyps(2) AddNode.hyps(3) IRNodes.inputs-of-AddNode
      encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case

```

```

    by (metis AddNode.IH(1) AddNode.IH(2) AddNode.hyps(1) AddNode.hyps(2)
AddNode.hyps(3) AddNode.premis(1) IRNodes.inputs-of-AddNode ⟨kind g2 n = AddNode
x y⟩ child-unchanged encode-in-ids in-set-member inputs.simps local.wf member-rec(1)
rep.AddNode)
next
  case (MulNode n x y xe ye)
  then have kind g2 n = MulNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
  by (metis MulNode.hyps(1) MulNode.hyps(2) MulNode.hyps(3) IRNodes.inputs-of-MulNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using MulNode inputs-of-MulNode
  by (metis ⟨kind g2 n = MulNode x y⟩ child-unchanged inputs.simps list.set-intros(1)
rep.MulNode set-subset-Cons subset-iff unchanged.elims(2))
next
  case (SubNode n x y xe ye)
  then have kind g2 n = SubNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
  by (metis SubNode.hyps(1) SubNode.hyps(2) SubNode.hyps(3) IRNodes.inputs-of-SubNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using SubNode inputs-of-SubNode
  by (metis ⟨kind g2 n = SubNode x y⟩ child-member child-unchanged encode-in-ids
ids-some member-rec(1) rep.SubNode)
next
  case (AndNode n x y xe ye)
  then have kind g2 n = AndNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
  by (metis AndNode.hyps(1) AndNode.hyps(2) AndNode.hyps(3) IRNodes.inputs-of-AndNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using AndNode inputs-of-AndNode
  by (metis ⟨kind g2 n = AndNode x y⟩ child-unchanged inputs.simps list.set-intros(1)
rep.AndNode set-subset-Cons subset-iff unchanged.elims(2))
next
  case (OrNode n x y xe ye)
  then have kind g2 n = OrNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-OrNode inputs-of-are-usages
  by (metis OrNode.hyps(1) OrNode.hyps(2) OrNode.hyps(3) IRNodes.inputs-of-OrNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using OrNode inputs-of-OrNode
  by (metis ⟨kind g2 n = OrNode x y⟩ child-member child-unchanged encode-in-ids
ids-some member-rec(1) rep.OrNode)
next

```



```

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

```

```

then have  $x \in \text{eval-usages } g1 \ n$ 
  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  $\langle \text{kind } g2 \ n = \text{NarrowNode } ib \ rb \ x \rangle$  child-unchanged inputs.elims
list.set-intros(1) rep.NarrowNode unchanged.simps)
next
  case (SignExtendNode n ib rb x xe)
  then have  $\text{kind } g2 \ n = \text{SignExtendNode } ib \ rb \ x$ 
    using kind-unchanged by metis
  then have  $x \in \text{eval-usages } g1 \ n$ 
    using inputs-of-SignExtendNode inputs-of-are-usages
    by (metis SignExtendNode.hyps(1) SignExtendNode.hyps(2) encode-in-ids in-
puts.simps inputs-are-usages list.set-intros(1))
    then show ?case using SignExtendNode inputs-of-SignExtendNode
      by (metis  $\langle \text{kind } g2 \ n = \text{SignExtendNode } ib \ rb \ x \rangle$  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  $\text{kind } g2 \ n = \text{ZeroExtendNode } ib \ rb \ x$ 
    using kind-unchanged by metis
  then have  $x \in \text{eval-usages } g1 \ n$ 
    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  $\langle \text{kind } g2 \ n = \text{ZeroExtendNode } ib \ rb \ x \rangle$  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  $\text{kind } g2 \ n = \text{RefNode } n'$ 
    using kind-unchanged by metis
  then have  $n' \in \text{eval-usages } g1 \ n$ 
    by (metis IRNodes.inputs-of-RefNode RefNode.hyps(1) RefNode.hyps(2) en-
code-in-ids inputs.elims inputs-are-usages list.set-intros(1))
    then show ?case
      by (metis IRNodes.inputs-of-RefNode RefNode.IH RefNode.hyps(1) RefNode.hyps(2)
RefNode.prem(1)  $\langle \text{kind } g2 \ n = \text{RefNode } n' \rangle$  child-unchanged encode-in-ids in-
puts.elims list.set-intros(1) local.wf rep.RefNode)
qed
qed

```

```

theorem stay-same:
  assumes nc: unchanged (eval-usages g1 nid) g1 g2
  assumes g1: [g1, m, p] ⊢ nid ↦ v1
  assumes wf: wf-graph g1
  shows [g2, m, p] ⊢ nid ↦ v1
proof –
  have nid: nid ∈ ids g1
    using g1 eval-in-ids by simp
  then have nid ∈ eval-usages g1 nid
    using eval-usages-self by blast
  then have kind-same: kind g1 nid = kind g2 nid
    using nc node-unchanged by blast
  obtain e where e: (g1 ⊢ nid ≃ e) ∧ ([m,p] ⊢ e ↦ v1)
    using encodeeval-def g1
    by auto
  then have val: [m,p] ⊢ e ↦ v1
    using g1 encodeeval-def
    by simp
  then show ?thesis using e nid nc
    unfolding encodeeval-def
  proof (induct e v1 arbitrary: nid rule: evaltree.induct)
    case (ConstantExpr c)
      then show ?case
        by (meson local.wf stay-same-encoding)
    next
      case (ParameterExpr i s)
        have g2 ⊢ nid ≃ ParameterExpr i s
          using stay-same-encoding ParameterExpr
          by (meson local.wf)
        then show ?case using evaltree.ParameterExpr
          by (meson ParameterExpr.hyps)
    next
      case (ConditionalExpr ce cond branch te fe v)
        then have g2 ⊢ nid ≃ ConditionalExpr ce te fe
          using ConditionalExpr.prems(1) ConditionalExpr.prems(3) local.wf stay-same-encoding
          by presburger
        then show ?case
          by (meson ConditionalExpr.prems(1) ConditionalExpr.prems(3) local.wf
stay-same-encoding)
    next
      case (UnaryExpr xe v op)
        then show ?case
          using local.wf stay-same-encoding by blast
    next
      case (BinaryExpr xe x ye y op)
        then show ?case
          using local.wf stay-same-encoding by blast
    next
      case (LeafExpr val nid s)

```

```

    then show ?case
    by (metis local.wf stay-same-encoding)
qed
qed

```

```

lemma add-changed:
  assumes gup = add-node new k g
  shows changeonly {new} g gup
  using assms unfolding add-node-def changeonly.simps
  using add-node.rep-eq add-node-def kind.rep-eq stamp.rep-eq by simp

```

```

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

```

```

lemma add-node-unchanged:
  assumes new  $\notin$  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'  $\notin$  ids g
  assumes nid  $\in$  ids g
  shows  $\neg$ (eval-uses g nid nid')
proof -
  have 0: nid  $\neq$  nid'
    using assms by blast
  have inp: nid'  $\notin$  inputs g nid
    using assms
    using inp-in-g-wf by blast
  have rec-0:  $\nexists n . n \in$  ids g  $\wedge$  n = nid'
    using assms by blast
  have rec-inp:  $\nexists n . n \in$  ids g  $\wedge$  n  $\in$  inputs g nid'
    using assms(2) inp-in-g by blast
  have rec:  $\nexists$  nid'' . eval-uses g nid nid''  $\wedge$  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 IRExp type that 'rep' will produce. These are very helpful

for proving that 'rep' is deterministic.

named-theorems *rep*

lemma *rep-constant* [*rep*]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = ConstantNode\ c \implies$
 $e = ConstantExpr\ c$
by (*induction rule: rep.induct; auto*)

lemma *rep-parameter* [*rep*]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = ParameterNode\ i \implies$
 $(\exists\ s.\ e = ParameterExpr\ i\ s)$
by (*induction rule: rep.induct; auto*)

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

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

lemma *rep-abs* [*rep*]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = AbsNode\ x \implies$
 $(\exists\ xe.\ e = UnaryExpr\ UnaryAbs\ xe)$
by (*induction rule: rep.induct; auto*)

lemma *rep-not* [*rep*]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = NotNode\ x \implies$
 $(\exists\ xe.\ e = UnaryExpr\ UnaryNot\ xe)$
by (*induction rule: rep.induct; auto*)

lemma *rep-negate* [*rep*]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = NegateNode\ x \implies$
 $(\exists\ xe.\ e = UnaryExpr\ UnaryNeg\ xe)$
by (*induction rule: rep.induct; auto*)

lemma *rep-logicnegation* [*rep*]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = LogicNegationNode\ x \implies$
 $(\exists\ xe.\ e = UnaryExpr\ UnaryLogicNegation\ xe)$
by (*induction rule: rep.induct; auto*)

lemma *rep-add* [*rep*]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = AddNode\ x\ y \implies$
 $(\exists\ xe\ ye.\ e = BinaryExpr\ BinAdd\ xe\ ye)$

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

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

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

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

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

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

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

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

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

lemma *rep-unsigned-right-shift* [rep]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = UnsignedRightShiftNode\ x\ y \implies$
 $(\exists\ xe\ ye. e = BinaryExpr\ BinURightShift\ xe\ ye)$
by (induction rule: *rep.induct*; auto)

lemma *rep-integer-below* [rep]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = IntegerBelowNode\ x\ y \implies$
 $(\exists\ xe\ ye. e = BinaryExpr\ BinIntegerBelow\ xe\ ye)$
by (induction rule: *rep.induct*; auto)

lemma *rep-integer-equals* [rep]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = IntegerEqualsNode\ x\ y \implies$
 $(\exists\ xe\ ye. e = BinaryExpr\ BinIntegerEquals\ xe\ ye)$
by (induction rule: *rep.induct*; auto)

lemma *rep-integer-less-than* [rep]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = IntegerLessThanNode\ x\ y \implies$
 $(\exists\ xe\ ye. e = BinaryExpr\ BinIntegerLessThan\ xe\ ye)$
by (induction rule: *rep.induct*; auto)

lemma *rep-narrow* [rep]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = NarrowNode\ ib\ rb\ x \implies$
 $(\exists\ x. e = UnaryExpr\ (UnaryNarrow\ ib\ rb)\ x)$
by (induction rule: *rep.induct*; auto)

lemma *rep-sign-extend* [rep]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = SignExtendNode\ ib\ rb\ x \implies$
 $(\exists\ x. e = UnaryExpr\ (UnarySignExtend\ ib\ rb)\ x)$
by (induction rule: *rep.induct*; auto)

lemma *rep-zero-extend* [rep]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = ZeroExtendNode\ ib\ rb\ x \implies$
 $(\exists\ x. e = UnaryExpr\ (UnaryZeroExtend\ ib\ rb)\ x)$
by (induction rule: *rep.induct*; auto)

lemma *rep-load-field* [rep]:

$g \vdash n \simeq e \implies$
 $is\ preevaluated\ (kind\ g\ n) \implies$
 $(\exists\ s. e = LeafExpr\ n\ s)$
by (induction rule: *rep.induct*; auto)

```

lemma rep-ref [rep]:
  g ⊢ n ≃ e ⇒
    kind g n = RefNode n' ⇒
      g ⊢ n' ≃ e
  by (induction rule: rep.induct; auto)

```

```

method solve-det uses node =
  (match node in kind - - = node - for node ⇒
    ⟨match rep in r: - ⇒ - = node - ⇒ - ⇒
      ⟨match IRNode.inject in i: (node - = node -) = - ⇒
        ⟨match RepE in e: - ⇒ (∧x. - = node x ⇒ -) ⇒ - ⇒
          ⟨match IRNode.distinct in d: node - ≠ RefNode - ⇒
            ⟨metis i e r d⟩⟩⟩⟩ |
    match node in kind - - = node - - for node ⇒
      ⟨match rep in r: - ⇒ - = node - - ⇒ - ⇒
        ⟨match IRNode.inject in i: (node - - = node - -) = - ⇒
          ⟨match RepE in e: - ⇒ (∧x y. - = node x y ⇒ -) ⇒ - ⇒
            ⟨match IRNode.distinct in d: node - - ≠ RefNode - ⇒
              ⟨metis i e r d⟩⟩⟩⟩ |
    match node in kind - - = node - - - for node ⇒
      ⟨match rep in r: - ⇒ - = node - - - ⇒ - ⇒
        ⟨match IRNode.inject in i: (node - - - = node - - -) = - ⇒
          ⟨match RepE in e: - ⇒ (∧x y z. - = node x y z ⇒ -) ⇒ - ⇒
            ⟨match IRNode.distinct in d: node - - - ≠ RefNode - ⇒
              ⟨metis i e r d⟩⟩⟩⟩ |
    match node in kind - - = node - - - for node ⇒
      ⟨match rep in r: - ⇒ - = node - - - ⇒ - ⇒
        ⟨match IRNode.inject in i: (node - - - = node - - -) = - ⇒
          ⟨match RepE in e: - ⇒ (∧x. - = node - - x ⇒ -) ⇒ - ⇒
            ⟨match IRNode.distinct in d: node - - - ≠ RefNode - ⇒
              ⟨metis i e r d⟩⟩⟩⟩)

```

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

```

lemma repDet:
  shows (g ⊢ n ≃ e1) ⇒ (g ⊢ n ≃ e2) ⇒ e1 = e2
proof (induction arbitrary: e2 rule: rep.induct)
  case (ConstantNode n c)
  then show ?case using rep-constant by auto
next
  case (ParameterNode n i s)
  then show ?case
    by (metis IRNode.disc(2685) ParameterNodeE is-RefNode-def rep-parameter)
next
  case (ConditionalNode n c t f ce te fe)
  then show ?case
    using IRNode.distinct(593)
    using IRNode.inject(6) ConditionalNodeE rep-conditional

```

```

      by metis
next
  case (AbsNode n x xe)
  then show ?case
    by (solve-det node: AbsNode)
next
  case (NotNode n x xe)
  then show ?case
    by (solve-det node: NotNode)
next
  case (NegateNode n x xe)
  then show ?case
    by (solve-det node: NegateNode)
next
  case (LogicNegationNode n x xe)
  then show ?case
    by (solve-det node: LogicNegationNode)
next
  case (AddNode n x y xe ye)
  then show ?case
    by (solve-det node: AddNode)
next
  case (MulNode n x y xe ye)
  then show ?case
    by (solve-det node: MulNode)
next
  case (SubNode n x y xe ye)
  then show ?case
    by (solve-det node: SubNode)
next
  case (AndNode n x y xe ye)
  then show ?case
    by (solve-det node: AndNode)
next
  case (OrNode n x y xe ye)
  then show ?case
    by (solve-det node: OrNode)
next
  case (XorNode n x y xe ye)
  then show ?case
    by (solve-det node: XorNode)
next
  case (ShortCircuitOrNode n x y xe ye)
  then show ?case
    by (solve-det node: ShortCircuitOrNode)
next
  case (LeftShiftNode n x y xe ye)
  then show ?case
    by (solve-det node: LeftShiftNode)

```

```

next
  case (RightShiftNode n x y xe ye)
  then show ?case
    by (solve-det node: RightShiftNode)
next
  case (UnsignedRightShiftNode n x y xe ye)
  then show ?case
    by (solve-det node: UnsignedRightShiftNode)
next
  case (IntegerBelowNode n x y xe ye)
  then show ?case
    by (solve-det node: IntegerBelowNode)
next
  case (IntegerEqualsNode n x y xe ye)
  then show ?case
    by (solve-det node: IntegerEqualsNode)
next
  case (IntegerLessThanNode n x y xe ye)
  then show ?case
    by (solve-det node: IntegerLessThanNode)
next
  case (NarrowNode n x xe)
  then show ?case
    by (metis IRNode.distinct(2203) IRNode.inject(28) NarrowNodeE rep-narrow)
next
  case (SignExtendNode n x xe)
  then show ?case
    by (metis IRNode.distinct(2599) IRNode.inject(39) SignExtendNodeE rep-sign-extend)
next
  case (ZeroExtendNode n x xe)
  then show ?case
    by (metis IRNode.distinct(2753) IRNode.inject(50) ZeroExtendNodeE rep-zero-extend)
next
  case (LeafNode n s)
  then show ?case using rep-load-field LeafNodeE
    by (metis is-preevaluated.simps(53))
next
  case (RefNode n')
  then show ?case
    using rep-ref by blast
qed

lemma repAllDet:
   $g \vdash xs \simeq_L e1 \implies$ 
   $g \vdash xs \simeq_L e2 \implies$ 
   $e1 = e2$ 
proof (induction arbitrary: e2 rule: replist.induct)
  case RepNil
  then show ?case

```

```

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

```

```

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

```

```

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

```

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

```

```

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

```

```

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

```

```

lemma mono-logic-negation:
  assumes kind g1 n = LogicNegationNode x ∧ kind g2 n = LogicNegationNode x
  assumes (g1 ⊢ x ≃ xe1) ∧ (g2 ⊢ x ≃ xe2)
  assumes xe1 ≥ xe2
  assumes (g1 ⊢ n ≃ e1) ∧ (g2 ⊢ n ≃ e2)

```

shows $e1 \geq e2$
by (*metis* *LogicNegationNode* *assms*(1) *assms*(2) *assms*(3) *assms*(4) *mono-unary* *repDet*)

lemma *mono-narrow*:

assumes $\text{kind } g1 \ n = \text{NarrowNode } ib \ rb \ x \wedge \text{kind } g2 \ n = \text{NarrowNode } ib \ rb \ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
using *assms* *mono-unary* *repDet* *NarrowNode*
by *metis*

lemma *mono-sign-extend*:

assumes $\text{kind } g1 \ n = \text{SignExtendNode } ib \ rb \ x \wedge \text{kind } g2 \ n = \text{SignExtendNode } ib \ rb \ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
by (*metis* *SignExtendNode* *assms*(1) *assms*(2) *assms*(3) *assms*(4) *mono-unary* *repDet*)

lemma *mono-zero-extend*:

assumes $\text{kind } g1 \ n = \text{ZeroExtendNode } ib \ rb \ x \wedge \text{kind } g2 \ n = \text{ZeroExtendNode } ib \ rb \ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
using *assms* *mono-unary* *repDet* *ZeroExtendNode*
by *metis*

lemma *mono-conditional-graph*:

assumes $\text{kind } g1 \ n = \text{ConditionalNode } c \ t \ f \wedge \text{kind } g2 \ n = \text{ConditionalNode } c \ t \ f$
assumes $(g1 \vdash c \simeq ce1) \wedge (g2 \vdash c \simeq ce2)$
assumes $(g1 \vdash t \simeq te1) \wedge (g2 \vdash t \simeq te2)$
assumes $(g1 \vdash f \simeq fe1) \wedge (g2 \vdash f \simeq fe2)$
assumes $ce1 \geq ce2 \wedge te1 \geq te2 \wedge fe1 \geq fe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
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 $\text{kind } g1 \ n = \text{AddNode } x \ y \wedge \text{kind } g2 \ n = \text{AddNode } x \ y$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$

```

assumes  $xe1 \geq xe2 \wedge ye1 \geq ye2$ 
assumes  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$ 
shows  $e1 \geq e2$ 
using mono-binary assms AddNodeE IRNode.inject(2) repDet rep-add
by (metis IRNode.distinct(205))

```

lemma *mono-mul*:

```

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

```

lemma *term-graph-evaluation*:

```

 $(g \vdash n \sqsubseteq e) \implies (\forall\ m\ p\ v.\ ([m,p] \vdash e \mapsto v) \longrightarrow ([g,m,p] \vdash n \mapsto v))$ 
unfolding graph-represents-expression-def apply auto
by (meson encodeeval-def)

```

lemma *encodes-contains*:

```

 $g \vdash n \simeq e \implies$ 
 $kind\ g\ n \neq NoNode$ 
apply (induction rule: rep.induct)
apply (match IRNode.distinct in e: ?n ≠ NoNode ⇒
 $\langle presburger\ add: e \rangle +$ 
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: en-
codes-contains)

```

lemma *not-excluded-keep-type*:

```

assumes  $n \in ids\ g1$ 
assumes  $n \notin excluded$ 
assumes  $(excluded \sqsubseteq as-set\ g1) \subseteq as-set\ g2$ 
shows  $kind\ g1\ n = kind\ g2\ n \wedge stamp\ g1\ n = stamp\ g2\ n$ 
using assms unfolding as-set-def domain-subtraction-def by blast

```

method *metis-node-eq-unary* **for** $node :: 'a \Rightarrow IRNode =$

```

(match IRNode.inject in i: (node - = node -) = - ⇒
 $\langle metis\ i \rangle$ )

```

method *metis-node-eq-binary* **for** $node :: 'a \Rightarrow 'a \Rightarrow IRNode =$

```

(match IRNode.inject in i: (node - - = node - -) = - ⇒)

```

```

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

```

7.8.3 Lift Data-flow Tree Refinement to Graph Refinement

```

theorem graph-semantic-preservation:
  assumes a:  $e1' \geq e2'$ 
  assumes b:  $(\{n'\} \trianglelefteq \text{as-set } g1) \subseteq \text{as-set } g2$ 
  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 sub-
    setI)
  apply (rule allI) apply (rule impI) apply (rule allI) apply (rule impI)
  unfolding graph-represents-expression-def
proof –
  fix n e1
  assume e:  $n \in \text{ids } g1$ 
  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:  $\text{kind } g1 \ n = \text{kind } g2 \ n \wedge \text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
    using not-excluded-keep-type
    using b e by presburger
  show ?thesis using f i
proof (induction e1)
  case (ConstantNode n c)
  then show ?case
    by (metis eq-refl rep.ConstantNode)
next
  case (ParameterNode n i s)
  then show ?case
    by (metis eq-refl rep.ParameterNode)
next
  case (ConditionalNode n c t f ce1 te1 fe1)

```



```

have k: g1 ⊢ n ≃ ConditionalExpr ce1 te1 fe1 using f ConditionalNode
  by (simp add: ConditionalNode.hyps(2) rep.ConditionalNode)
obtain cn tn fn where l: kind g1 n = ConditionalNode cn tn fn
  using ConditionalNode.hyps(1) by blast
then have mc: g1 ⊢ cn ≃ ce1
  using ConditionalNode.hyps(1) ConditionalNode.hyps(2) by fastforce
from l have mt: g1 ⊢ tn ≃ te1
  using ConditionalNode.hyps(1) ConditionalNode.hyps(3) by fastforce
from l have mf: g1 ⊢ fn ≃ fe1
  using ConditionalNode.hyps(1) ConditionalNode.hyps(4) by fastforce
then show ?case
proof -
  have g1 ⊢ cn ≃ ce1 using mc by simp
  have g1 ⊢ tn ≃ te1 using mt by simp
  have g1 ⊢ fn ≃ fe1 using mf by simp
  have cer: ∃ ce2. (g2 ⊢ cn ≃ ce2) ∧ ce1 ≥ ce2
    using ConditionalNode
    using a b c d l no-encoding not-excluded-keep-type repDet singletonD
    by (metis-node-eq-ternary ConditionalNode)
  have ter: ∃ te2. (g2 ⊢ tn ≃ te2) ∧ te1 ≥ te2
    using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
    singletonD
    by (metis-node-eq-ternary ConditionalNode)
  have ∃ fe2. (g2 ⊢ fn ≃ fe2) ∧ fe1 ≥ fe2
    using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
    singletonD
    by (metis-node-eq-ternary ConditionalNode)
  then have ∃ ce2 te2 fe2. (g2 ⊢ n ≃ ConditionalExpr ce2 te2 fe2) ∧
    ConditionalExpr ce1 te1 fe1 ≥ ConditionalExpr ce2 te2 fe2
    using ConditionalNode.premis l rep.ConditionalNode cer ter
    by (smt (verit) mono-conditional)
  then show ?thesis
    by meson
qed
next
case (AbsNode n x xe1)
have k: g1 ⊢ n ≃ UnaryExpr UnaryAbs xe1 using f AbsNode
  by (simp add: AbsNode.hyps(2) rep.AbsNode)
obtain xn where l: kind g1 n = AbsNode xn
  using AbsNode.hyps(1) by blast
then have m: g1 ⊢ xn ≃ xe1
  using AbsNode.hyps(1) AbsNode.hyps(2) by fastforce
then show ?case
proof (cases xn = n')
case True
  then have n: xe1 = e1' using c m repDet by simp
  then have ev: g2 ⊢ n ≃ UnaryExpr UnaryAbs e2' using AbsNode.hyps(1)
    l m n
    using AbsNode.premis True d rep.AbsNode by simp

```

```

    then have r: UnaryExpr UnaryAbs e1' ≥ UnaryExpr UnaryAbs e2'
      by (meson a mono-unary)
    then show ?thesis using ev r
      by (metis n)
  next
    case False
    have g1 ⊢ xn ≃ xe1 using m by simp
    have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
      using AbsNode
    using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
      by (metis-node-eq-unary AbsNode)
    then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryAbs xe2) ∧ UnaryExpr
UnaryAbs xe1 ≥ UnaryExpr UnaryAbs xe2
      by (metis AbsNode.premis l mono-unary rep.AbsNode)
    then show ?thesis
      by meson
  qed
next
  case (NotNode n x xe1)
  have k: g1 ⊢ n ≃ UnaryExpr UnaryNot xe1 using f NotNode
    by (simp add: NotNode.hyps(2) rep.NotNode)
  obtain xn where l: kind g1 n = NotNode xn
    using NotNode.hyps(1) by blast
  then have m: g1 ⊢ xn ≃ xe1
    using NotNode.hyps(1) NotNode.hyps(2) by fastforce
  then show ?case
  proof (cases xn = n')
    case True
    then have n: xe1 = e1' using c m repDet by simp
    then have ev: g2 ⊢ n ≃ UnaryExpr UnaryNot e2' using NotNode.hyps(1)
l m n
      using NotNode.premis True d rep.NotNode by simp
    then have r: UnaryExpr UnaryNot e1' ≥ UnaryExpr UnaryNot e2'
      by (meson a mono-unary)
    then show ?thesis using ev r
      by (metis n)
  next
    case False
    have g1 ⊢ xn ≃ xe1 using m by simp
    have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
      using NotNode
    using False i b l not-excluded-keep-type singletonD no-encoding
      by (metis-node-eq-unary NotNode)
    then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryNot xe2) ∧ UnaryExpr
UnaryNot xe1 ≥ UnaryExpr UnaryNot xe2
      by (metis NotNode.premis l mono-unary rep.NotNode)
    then show ?thesis
      by meson
  qed

```

```

next
  case (NegateNode n x xe1)
  have k:  $g1 \vdash n \simeq \text{UnaryExpr UnaryNeg xe1}$  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:  $g2 \vdash n \simeq \text{UnaryExpr UnaryNeg e2'}$  using NegateNode.hyps(1)
  l m n
    using NegateNode.premis True d rep.NegateNode by simp
  then have r:  $\text{UnaryExpr UnaryNeg e1'} \geq \text{UnaryExpr UnaryNeg e2'}$ 
    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  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr UnaryNeg xe2}) \wedge \text{UnaryExpr}$ 
  UnaryNeg xe1  $\geq \text{UnaryExpr UnaryNeg xe2}$ 
    by (metis NegateNode.premis l mono-unary rep.NegateNode)
  then show ?thesis
    by meson
  qed
next
case (LogicNegationNode n x xe1)
  have k:  $g1 \vdash n \simeq \text{UnaryExpr UnaryLogicNegation xe1}$  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:  $g2 \vdash n \simeq \text{UnaryExpr UnaryLogicNegation e2'}$  using
  LogicNegationNode.hyps(1) l m n
    using LogicNegationNode.premis True d rep.LogicNegationNode by simp
  then have r:  $\text{UnaryExpr UnaryLogicNegation e1'} \geq \text{UnaryExpr UnaryLog-}$ 

```

```

icNegation e2'
  by (meson a mono-unary)
  then show ?thesis using ev r
  by (metis n)
next
case False
have g1 ⊢ xn ≃ xe1 using m by simp
have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
  using LogicNegationNode
  using False i b l not-excluded-keep-type singletonD no-encoding
  by (metis-node-eq-unary LogicNegationNode)
  then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryLogicNegation xe2) ∧
UnaryExpr UnaryLogicNegation xe1 ≥ UnaryExpr UnaryLogicNegation xe2
  by (metis LogicNegationNode.prem1 mono-unary rep.LogicNegationNode)
  then show ?thesis
  by meson
qed
next
case (AddNode n x y xe1 ye1)
have k: g1 ⊢ n ≃ BinaryExpr BinAdd xe1 ye1 using f AddNode
  by (simp add: AddNode.hyps(2) rep.AddNode)
obtain xn yn where l: kind g1 n = AddNode xn yn
  using AddNode.hyps(1) by blast
then have mx: g1 ⊢ xn ≃ xe1
  using AddNode.hyps(1) AddNode.hyps(2) by fastforce
from l have my: g1 ⊢ yn ≃ ye1
  using AddNode.hyps(1) AddNode.hyps(3) by fastforce
then show ?case
proof -
  have g1 ⊢ xn ≃ xe1 using mx by simp
  have g1 ⊢ yn ≃ ye1 using my by simp
  have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
    using AddNode
    using a b c d l no-encoding not-excluded-keep-type repDet singletonD
    by (metis-node-eq-binary AddNode)
  have ∃ ye2. (g2 ⊢ yn ≃ ye2) ∧ ye1 ≥ ye2
    using AddNode
    using a b c d l no-encoding not-excluded-keep-type repDet singletonD
    by (metis-node-eq-binary AddNode)
  then have ∃ xe2 ye2. (g2 ⊢ n ≃ BinaryExpr BinAdd xe2 ye2) ∧ BinaryExpr
BinAdd xe1 ye1 ≥ BinaryExpr BinAdd xe2 ye2
  by (metis AddNode.prem1 mono-binary rep.AddNode xer)
  then show ?thesis
  by meson
qed
next
case (MulNode n x y xe1 ye1)
have k: g1 ⊢ n ≃ BinaryExpr BinMul xe1 ye1 using f MulNode
  by (simp add: MulNode.hyps(2) rep.MulNode)

```

```

obtain  $xn\ yn$  where  $l$ :  $kind\ g1\ n = MulNode\ xn\ yn$ 
  using  $MulNode.hyps(1)$  by blast
then have  $mx$ :  $g1 \vdash xn \simeq xe1$ 
  using  $MulNode.hyps(1)\ MulNode.hyps(2)$  by fastforce
from  $l$  have  $my$ :  $g1 \vdash yn \simeq ye1$ 
  using  $MulNode.hyps(1)\ MulNode.hyps(3)$  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  $MulNode$ 
    using  $a\ b\ c\ d\ l\ no-encoding\ not-excluded-keep-type\ repDet\ singletonD$ 
    by (metis-node-eq-binary  $MulNode$ )
  have  $\exists\ ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
    using  $MulNode$ 
    using  $a\ b\ c\ d\ l\ no-encoding\ not-excluded-keep-type\ repDet\ singletonD$ 
    by (metis-node-eq-binary  $MulNode$ )
  then have  $\exists\ xe2\ ye2. (g2 \vdash n \simeq BinaryExpr\ BinMul\ xe2\ ye2) \wedge BinaryExpr$ 
     $BinMul\ xe1\ ye1 \geq BinaryExpr\ BinMul\ xe2\ ye2$ 
    by (metis  $MulNode.premis\ l\ mono-binary\ rep.MulNode\ xer$ )
  then show ?thesis
    by meson
qed
next
case ( $SubNode\ n\ x\ y\ xe1\ ye1$ )
have  $k$ :  $g1 \vdash n \simeq BinaryExpr\ BinSub\ xe1\ ye1$  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  $mx$ :  $g1 \vdash xn \simeq xe1$ 
  using  $SubNode.hyps(1)\ SubNode.hyps(2)$  by fastforce
from  $l$  have  $my$ :  $g1 \vdash yn \simeq ye1$ 
  using  $SubNode.hyps(1)\ SubNode.hyps(3)$  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  $SubNode$ 
    using  $a\ b\ c\ d\ l\ no-encoding\ not-excluded-keep-type\ repDet\ singletonD$ 
    by (metis-node-eq-binary  $SubNode$ )
  have  $\exists\ ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
using  $SubNode\ a\ b\ c\ d\ l\ no-encoding\ not-excluded-keep-type\ repDet\ singletonD$ 
  by (metis-node-eq-binary  $SubNode$ )
  then have  $\exists\ xe2\ ye2. (g2 \vdash n \simeq BinaryExpr\ BinSub\ xe2\ ye2) \wedge BinaryExpr$ 
     $BinSub\ xe1\ ye1 \geq BinaryExpr\ BinSub\ xe2\ ye2$ 
    by (metis  $SubNode.premis\ l\ mono-binary\ rep.SubNode\ xer$ )
  then show ?thesis

```

```

      by meson
    qed
  next
    case (AndNode n x y xe1 ye1)
    have k:  $g1 \vdash n \simeq \text{BinaryExpr BinAnd } xe1 \text{ } ye1$  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  $\exists xe2 ye2. (g2 \vdash n \simeq \text{BinaryExpr BinAnd } xe2 \text{ } ye2) \wedge \text{BinaryExpr BinAnd } xe1 \text{ } ye1 \geq \text{BinaryExpr BinAnd } xe2 \text{ } ye2$ 
        by (metis AndNode.premis l mono-binary rep.AndNode xer)
      then show ?thesis
        by meson
    qed
  next
    case (OrNode n x y xe1 ye1)
    have k:  $g1 \vdash n \simeq \text{BinaryExpr BinOr } xe1 \text{ } ye1$  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.premis 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.premis 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 ShortCir-
cuitOrNode
by (simp add: ShortCircuitOrNode.hyps(2) rep.ShortCircuitOrNode)
obtain xn yn where l: kind g1 n = ShortCircuitOrNode xn yn
using ShortCircuitOrNode.hyps(1) by blast
then have mx:  $g1 \vdash xn \simeq xe1$ 
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  $g1 \vdash xn \simeq xe1$  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 ShortCircuitOrNode
      using  $a\ b\ c\ d\ l$  no-encoding not-excluded-keep-type repDet singletonD
      by (metis-node-eq-binary ShortCircuitOrNode)
    have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
      using ShortCircuitOrNode a b c d l no-encoding not-excluded-keep-type
repDet singletonD
      by (metis-node-eq-binary ShortCircuitOrNode)
    then have  $\exists xe2\ ye2. (g2 \vdash n \simeq BinaryExpr\ BinShortCircuitOr\ xe2\ ye2) \wedge$ 
BinaryExpr BinShortCircuitOr xe1 ye1  $\geq BinaryExpr BinShortCircuitOr xe2 ye2$ 
      by (metis ShortCircuitOrNode.premis l mono-binary rep.ShortCircuitOrNode
xer)
    then show ?thesis
      by meson
  qed
next
case (LeftShiftNode n x y xe1 ye1)
have  $k: g1 \vdash n \simeq BinaryExpr\ BinLeftShift\ xe1\ ye1$  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  $mx: g1 \vdash xn \simeq xe1$ 
  using LeftShiftNode.hyps(1) LeftShiftNode.hyps(2) by fastforce
from  $l$  have  $my: g1 \vdash yn \simeq ye1$ 
  using LeftShiftNode.hyps(1) LeftShiftNode.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 LeftShiftNode
    using  $a\ b\ c\ d\ l$  no-encoding not-excluded-keep-type repDet singletonD
    by (metis-node-eq-binary LeftShiftNode)
  have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
    using LeftShiftNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
    by (metis-node-eq-binary LeftShiftNode)
  then have  $\exists xe2\ ye2. (g2 \vdash n \simeq BinaryExpr\ BinLeftShift\ xe2\ ye2) \wedge$ 
BinaryExpr BinLeftShift xe1 ye1  $\geq BinaryExpr BinLeftShift xe2 ye2$ 
    by (metis LeftShiftNode.premis l mono-binary rep.LeftShiftNode xer)
  then show ?thesis
    by meson
  qed
next
case (RightShiftNode n x y xe1 ye1)
have  $k: g1 \vdash n \simeq BinaryExpr\ BinRightShift\ xe1\ ye1$  using  $f$  RightShiftNode
  by (simp add: RightShiftNode.hyps(2) rep.RightShiftNode)

```



```

obtain  $xn\ yn$  where  $l$ :  $\text{kind } g1\ n = \text{RightShiftNode } xn\ yn$ 
  using  $\text{RightShiftNode.hyps}(1)$  by blast
then have  $m\!x$ :  $g1 \vdash xn \simeq xe1$ 
  using  $\text{RightShiftNode.hyps}(1)\ \text{RightShiftNode.hyps}(2)$  by fastforce
from  $l$  have  $m\!y$ :  $g1 \vdash yn \simeq ye1$ 
  using  $\text{RightShiftNode.hyps}(1)\ \text{RightShiftNode.hyps}(3)$  by fastforce
then show ?case
proof –
  have  $g1 \vdash xn \simeq xe1$  using  $m\!x$  by simp
  have  $g1 \vdash yn \simeq ye1$  using  $m\!y$  by simp
  have  $x\!e\!r$ :  $\exists\ xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
    using  $\text{RightShiftNode}$ 
    using  $a\ b\ c\ d\ l$  no-encoding not-excluded-keep-type repDet singletonD
    by (metis-node-eq-binary RightShiftNode)
  have  $\exists\ ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
    using  $\text{RightShiftNode } a\ b\ c\ d\ l$  no-encoding not-excluded-keep-type repDet
singletonD
    by (metis-node-eq-binary RightShiftNode)
  then have  $\exists\ xe2\ ye2. (g2 \vdash n \simeq \text{BinaryExpr } \text{BinRightShift } xe2\ ye2) \wedge$ 
BinaryExpr BinRightShift xe1 ye1  $\geq \text{BinaryExpr } \text{BinRightShift } xe2\ ye2$ 
    by (metis RightShiftNode.premis l mono-binary rep.RightShiftNode xer)
  then show ?thesis
    by meson
qed
next
  case ( $\text{UnsignedRightShiftNode } n\ x\ y\ xe1\ ye1$ )
  have  $k$ :  $g1 \vdash n \simeq \text{BinaryExpr } \text{BinURightShift } xe1\ ye1$  using  $f\ \text{UnsignedRightShiftNode}$ 
    by (simp add: UnsignedRightShiftNode.hyps(2) rep.UnsignedRightShiftNode)
  obtain  $xn\ yn$  where  $l$ :  $\text{kind } g1\ n = \text{UnsignedRightShiftNode } xn\ yn$ 
    using  $\text{UnsignedRightShiftNode.hyps}(1)$  by blast
  then have  $m\!x$ :  $g1 \vdash xn \simeq xe1$ 
    using  $\text{UnsignedRightShiftNode.hyps}(1)\ \text{UnsignedRightShiftNode.hyps}(2)$  by
fastforce
  from  $l$  have  $m\!y$ :  $g1 \vdash yn \simeq ye1$ 
    using  $\text{UnsignedRightShiftNode.hyps}(1)\ \text{UnsignedRightShiftNode.hyps}(3)$  by
fastforce
  then show ?case
  proof –
    have  $g1 \vdash xn \simeq xe1$  using  $m\!x$  by simp
    have  $g1 \vdash yn \simeq ye1$  using  $m\!y$  by simp
    have  $x\!e\!r$ :  $\exists\ xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
      using  $\text{UnsignedRightShiftNode}$ 
      using  $a\ b\ c\ d\ l$  no-encoding not-excluded-keep-type repDet singletonD
      by (metis-node-eq-binary UnsignedRightShiftNode)
    have  $\exists\ ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
      using  $\text{UnsignedRightShiftNode } a\ b\ c\ d\ l$  no-encoding not-excluded-keep-type
repDet singletonD
      by (metis-node-eq-binary UnsignedRightShiftNode)

```

```

      then have  $\exists xe2 ye2. (g2 \vdash n \simeq BinaryExpr BinURightShift xe2 ye2) \wedge$ 
BinaryExpr BinURightShift xe1 ye1  $\geq BinaryExpr BinURightShift xe2 ye2$ 
      by (metis UnsignedRightShiftNode.premis l mono-binary rep.UnsignedRightShiftNode
xer)
      then show ?thesis
      by meson
    qed
  next
    case (IntegerBelowNode n x y xe1 ye1)
    have k:  $g1 \vdash n \simeq BinaryExpr BinIntegerBelow xe1 ye1$  using f IntegerBe-
lowNode
    by (simp add: IntegerBelowNode.hyps(2) rep.IntegerBelowNode)
    obtain xn yn where l: kind g1 n = IntegerBelowNode xn yn
    using IntegerBelowNode.hyps(1) by blast
    then have mx:  $g1 \vdash xn \simeq xe1$ 
    using IntegerBelowNode.hyps(1) IntegerBelowNode.hyps(2) by fastforce
    from l have my:  $g1 \vdash yn \simeq ye1$ 
    using IntegerBelowNode.hyps(1) IntegerBelowNode.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 IntegerBelowNode
      using a b c d l no-encoding not-excluded-keep-type repDet singletonD
      by (metis-node-eq-binary IntegerBelowNode)
      have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
      using IntegerBelowNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
      by (metis-node-eq-binary IntegerBelowNode)
      then have  $\exists xe2 ye2. (g2 \vdash n \simeq BinaryExpr BinIntegerBelow xe2 ye2) \wedge$ 
BinaryExpr BinIntegerBelow xe1 ye1  $\geq BinaryExpr BinIntegerBelow xe2 ye2$ 
      by (metis IntegerBelowNode.premis l mono-binary rep.IntegerBelowNode
xer)
      then show ?thesis
      by meson
    qed
  next
    case (IntegerEqualsNode n x y xe1 ye1)
    have k:  $g1 \vdash n \simeq BinaryExpr BinIntegerEquals xe1 ye1$  using f IntegerEqual-
sNode
    by (simp add: IntegerEqualsNode.hyps(2) rep.IntegerEqualsNode)
    obtain xn yn where l: kind g1 n = IntegerEqualsNode xn yn
    using IntegerEqualsNode.hyps(1) by blast
    then have mx:  $g1 \vdash xn \simeq xe1$ 
    using IntegerEqualsNode.hyps(1) IntegerEqualsNode.hyps(2) by fastforce
    from l have my:  $g1 \vdash yn \simeq ye1$ 
    using IntegerEqualsNode.hyps(1) IntegerEqualsNode.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 IntegerEqualsNode
    using  $a\ b\ c\ d\ l$  no-encoding not-excluded-keep-type repDet singletonD
    by (metis-node-eq-binary IntegerEqualsNode)
  have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
    using IntegerEqualsNode  $a\ b\ c\ d\ l$  no-encoding not-excluded-keep-type
repDet singletonD
    by (metis-node-eq-binary IntegerEqualsNode)
  then have  $\exists xe2\ ye2. (g2 \vdash n \simeq \text{BinaryExpr BinIntegerEquals } xe2\ ye2) \wedge$ 
BinaryExpr BinIntegerEquals  $xe1\ ye1 \geq \text{BinaryExpr BinIntegerEquals } xe2\ ye2$ 
    by (metis IntegerEqualsNode.premis l mono-binary rep.IntegerEqualsNode
xer)
    then show ?thesis
      by meson
qed
next
  case (IntegerLessThanNode  $n\ x\ y\ xe1\ ye1$ )
    have  $k: g1 \vdash n \simeq \text{BinaryExpr BinIntegerLessThan } xe1\ ye1$  using  $f$  IntegerLessThanNode
    by (simp add: IntegerLessThanNode.hyps(2) rep.IntegerLessThanNode)
    obtain  $xn\ yn$  where  $l: \text{kind } g1\ n = \text{IntegerLessThanNode } xn\ yn$ 
    using IntegerLessThanNode.hyps(1) by blast
    then have  $mx: g1 \vdash xn \simeq xe1$ 
    using IntegerLessThanNode.hyps(1) IntegerLessThanNode.hyps(2) by fast-force
    from  $l$  have  $my: g1 \vdash yn \simeq ye1$ 
    using IntegerLessThanNode.hyps(1) IntegerLessThanNode.hyps(3) by fast-force
    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 IntegerLessThanNode
          using  $a\ b\ c\ d\ l$  no-encoding not-excluded-keep-type repDet singletonD
          by (metis-node-eq-binary IntegerLessThanNode)
        have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
          using IntegerLessThanNode  $a\ b\ c\ d\ l$  no-encoding not-excluded-keep-type
repDet singletonD
          by (metis-node-eq-binary IntegerLessThanNode)
        then have  $\exists xe2\ ye2. (g2 \vdash n \simeq \text{BinaryExpr BinIntegerLessThan } xe2\ ye2)$ 
BinaryExpr BinIntegerLessThan  $xe1\ ye1 \geq \text{BinaryExpr BinIntegerLessThan } xe2\ ye2$ 
          by (metis IntegerLessThanNode.premis l mono-binary rep.IntegerLessThanNode
xer)
        then show ?thesis

```

```

      by meson
    qed
  next
    case (NarrowNode n inputBits resultBits x xe1)
    have k:  $g1 \vdash n \simeq \text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) xe1$  using
    f NarrowNode
      by (simp add: NarrowNode.hyps(2) rep.NarrowNode)
    obtain xn where l: kind  $g1$   $n = \text{NarrowNode inputBits resultBits } xn$ 
      using NarrowNode.hyps(1) by blast
    then have m:  $g1 \vdash xn \simeq xe1$ 
      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:  $g2 \vdash n \simeq \text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) e2'$ 
      using NarrowNode.hyps(1) l m n
        using NarrowNode.premis True d rep.NarrowNode by simp
        then have r:  $\text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) e1' \geq \text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) e2'$ 
          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  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
            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  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) xe2) \wedge \text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) xe1 \geq \text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) xe2$ 
            by (metis NarrowNode.premis l mono-unary rep.NarrowNode)
          then show ?thesis
            by meson
        qed
      next
        case (SignExtendNode n inputBits resultBits x xe1)
        have k:  $g1 \vdash n \simeq \text{UnaryExpr } (\text{UnarySignExtend inputBits resultBits}) xe1$ 
      using f SignExtendNode
        by (simp add: SignExtendNode.hyps(2) rep.SignExtendNode)
      obtain xn where l: kind  $g1$   $n = \text{SignExtendNode inputBits resultBits } xn$ 
        using SignExtendNode.hyps(1) by blast
      then have m:  $g1 \vdash xn \simeq xe1$ 
        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  $ev: g2 \vdash n \simeq UnaryExpr\ (UnarySignExtend\ inputBits\ resultBits)$ 
     $e2'$  using  $SignExtendNode.hyps(1)\ l\ m\ n$ 
      using  $SignExtendNode.premis\ True\ d\ rep.SignExtendNode$  by simp
      then have  $r: UnaryExpr\ (UnarySignExtend\ inputBits\ resultBits)\ e1' \geq$ 
 $UnaryExpr\ (UnarySignExtend\ inputBits\ resultBits)\ e2'$ 
        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  $\exists\ xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
        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  $\exists\ xe2. (g2 \vdash n \simeq UnaryExpr\ (UnarySignExtend\ inputBits\ result-$ 
 $Bits)\ xe2) \wedge UnaryExpr\ (UnarySignExtend\ inputBits\ resultBits)\ xe1 \geq UnaryExpr$ 
 $(UnarySignExtend\ inputBits\ resultBits)\ xe2$ 
        by (metis SignExtendNode.premis l mono-unary rep.SignExtendNode)
      then show ?thesis
        by meson
      qed
    next
      case ( $ZeroExtendNode\ n\ inputBits\ resultBits\ x\ xe1$ )
      have  $k: g1 \vdash n \simeq UnaryExpr\ (UnaryZeroExtend\ inputBits\ resultBits)\ xe1$ 
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  $m: g1 \vdash xn \simeq xe1$ 
      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  $ev: g2 \vdash n \simeq UnaryExpr\ (UnaryZeroExtend\ inputBits\ resultBits)$ 
           $e2'$  using  $ZeroExtendNode.hyps(1)\ l\ m\ n$ 
            using  $ZeroExtendNode.premis\ True\ d\ rep.ZeroExtendNode$  by simp
            then have  $r: UnaryExpr\ (UnaryZeroExtend\ inputBits\ resultBits)\ e1' \geq$ 
 $UnaryExpr\ (UnaryZeroExtend\ inputBits\ resultBits)\ e2'$ 
              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  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
      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  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr } (\text{UnaryZeroExtend inputBits resultBits}) xe2) \wedge \text{UnaryExpr } (\text{UnaryZeroExtend inputBits resultBits}) xe1 \geq \text{UnaryExpr } (\text{UnaryZeroExtend inputBits resultBits}) xe2$ 
      by (metis ZeroExtendNode.premis 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-antics-preservation-subscript*:

```

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

```

lemma *tree-to-graph-rewriting*:

```

   $e_1 \geq e_2$ 
   $\wedge (g_1 \vdash n \simeq e_1) \wedge \text{maximal-sharing } g_1$ 
   $\wedge (\{n\} \trianglelefteq \text{as-set } g_1) \subseteq \text{as-set } g_2$ 
   $\wedge (g_2 \vdash n \simeq e_2) \wedge \text{maximal-sharing } g_2$ 
 $\implies \text{graph-refinement } g_1 g_2$ 
  using graph-antics-preservation
  by auto

```

```

declare [[simp-trace]]
lemma equal-refines:
  fixes  $e1 e2 :: \text{IRExpr}$ 
  assumes  $e1 = e2$ 
  shows  $e1 \geq e2$ 
  using assms

```

```

  by simp
declare [[simp-trace=false]]

```

```

lemma eval-contains-id[simp]:  $g1 \vdash n \simeq e \implies n \in \text{ids } g1$ 
  using no-encoding by blast

```

```

lemma subset-kind[simp]:  $\text{as-set } g1 \subseteq \text{as-set } g2 \implies g1 \vdash n \simeq e \implies \text{kind } g1 \ n = \text{kind } g2 \ n$ 
  using eval-contains-id unfolding as-set-def
  by blast

```

```

lemma subset-stamp[simp]:  $\text{as-set } g1 \subseteq \text{as-set } g2 \implies g1 \vdash n \simeq e \implies \text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
  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  $\text{as-set } g1 \subseteq \text{as-set } g2$ 
  assumes  $(g1 \vdash n \simeq e)$ 
  shows  $(g2 \vdash n \simeq e)$ 
  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  $\subseteq$  as-set g2
  shows graph-refinement g1 g2
proof -
  have ids g1  $\subseteq$  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  $\in$  ids g1
    assume 2:g1  $\vdash$  n  $\simeq$  e1

    show  $\exists e2. (g2 \vdash n \simeq e2) \wedge e1 \geq e2$ 
    using assms 1 2 using subset-implies-evals
    by (meson equal-refines)
  qed
qed

```

```

lemma graph-construction:
  e1  $\geq$  e2
   $\wedge$  as-set g1  $\subseteq$  as-set g2
   $\wedge$  (g2  $\vdash$  n  $\simeq$  e2)
   $\implies$  (g2  $\vdash$  n  $\trianglelefteq$  e1)  $\wedge$  graph-refinement g1 g2
  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  $g' = \text{add-node } \text{nid} \ (\text{node}, s) \ g$ 
assumes  $\text{node} \neq \text{NoNode}$ 
shows  $\text{kind } g' \ \text{nid} = \text{node}$ 
using assms
using add-node-lookup by presburger

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

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

lemma fresh-ids:
assumes  $n = \text{get-fresh-id } g$ 
shows  $n \notin \text{ids } g$ 
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  $\forall n \in \text{ids } g. \text{kind } g \ n = \text{kind } g' \ n$ 
assumes  $\forall n \in \text{ids } g. \text{stamp } g \ n = \text{stamp } g' \ n$ 
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  $n \notin \text{ids } g$ 
  assumes  $g' = \text{add-node } n (k, s) g$ 
  shows  $\text{as-set } g \subseteq \text{as-set } g'$ 
  using assms
  by (smt (verit, del-insts) Collect-mono-iff Diff-idemp Diff-insert-absorb add-changed
    as-set-def disjoint-change unchanged.simps)

```

lemma *unrep-subset*:

```

  assumes  $(g \oplus e \rightsquigarrow (g', n))$ 
  shows  $\text{as-set } g \subseteq \text{as-set } g'$ 
  using assms proof (induction  $g \ e \ (g', n)$  arbitrary:  $g' \ n$ )
  case (ConstantNodeSame  $g \ c \ n$ )
  then show ?case by blast
next
  case (ConstantNodeNew  $g \ c \ n \ g'$ )
  then show ?case using fresh-ids fresh-node-subset
    by presburger
next
  case (ParameterNodeSame  $g \ i \ s \ n$ )
  then show ?case by blast
next
  case (ParameterNodeNew  $g \ i \ s \ n \ g'$ )
  then show ?case using fresh-ids fresh-node-subset
    by presburger
next
  case (ConditionalNodeSame  $g \ ce \ g2 \ c \ te \ g3 \ t \ fe \ g4 \ f \ s' \ n$ )
  then show ?case by blast
next

```

```

    case (ConditionalNodeNew g ce g2 c te g3 t fe g4 f s' n g')
    then show ?case using fresh-ids fresh-node-subset
      by (meson subset-trans)
next
    case (UnaryNodeSame g xe g2 x s' op n)
    then show ?case by blast
next
    case (UnaryNodeNew g xe g2 x s' op n g')
    then show ?case using fresh-ids fresh-node-subset
      by (meson subset-trans)
next
    case (BinaryNodeSame g xe g2 x ye g3 y s' op n)
    then show ?case by blast
next
    case (BinaryNodeNew g xe g2 x ye g3 y s' op n g')
    then show ?case using fresh-ids fresh-node-subset
      by (meson subset-trans)
next
    case (AllLeafNodes g n s)
    then show ?case by blast
qed

```

lemma *fresh-node-preserves-other-nodes*:

```

  assumes n' = get-fresh-id g
  assumes g' = add-node n' (k, s) g
  shows  $\forall n \in \text{ids } g. (g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$ 
  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 find-node-and-stamp g (k, s) = Some n
  shows  $\forall n \in \text{ids } g. (g \vdash n \simeq e) \longleftrightarrow (g \vdash n \simeq e)$ 
  using assms
  by blast

```

lemma *unrep-ids-subset[simp]*:

```

  assumes  $g \oplus e \rightsquigarrow (g', n)$ 
  shows  $\text{ids } g \subseteq \text{ids } g'$ 
  using assms unrep-subset
  by (meson graph-refinement-def subset-refines)

```

lemma *unrep-unchanged*:

```

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

```

theorem *term-graph-reconstruction*:

$g \oplus e \rightsquigarrow (g', n) \implies (g' \vdash n \simeq e) \wedge \text{as-set } g \subseteq \text{as-set } g'$
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 $\text{kind } g' n = \text{ConstantNode } c$
 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 $k: \text{kind } g4 n = \text{ConditionalNode } c t f$
 using *find-exists-kind* **by** *blast*
 have $c: g4 \vdash c \simeq ce$ **using** *local.ConditionalNodeSame unrep-unchanged*
 using *no-encoding* **by** *blast*
 have $t: g4 \vdash t \simeq te$ **using** *local.ConditionalNodeSame unrep-unchanged*
 using *no-encoding* **by** *blast*
 have $f: g4 \vdash f \simeq fe$ **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* $\neq \text{NoNode}$
 using *unary-node.elims* **by** *blast*
 ultimately have $k: \text{kind } g' n = \text{ConditionalNode } c t f$
 using *find-new-kind local.ConditionalNodeNew*
 by *presburger*
 then have $c: g' \vdash c \simeq ce$ **using** *local.ConditionalNodeNew unrep-unchanged*
 using *no-encoding*
 by (*metis ConditionalNodeNew.hyps(9) fresh-node-preserves-other-nodes*)
 then have $t: g' \vdash t \simeq te$ **using** *local.ConditionalNodeNew unrep-unchanged*
 using *no-encoding fresh-node-preserves-other-nodes*
 by *metis*
 then have $f: g' \vdash f \simeq fe$ **using** *local.ConditionalNodeNew unrep-unchanged*
 using *no-encoding fresh-node-preserves-other-nodes*
 by *metis*
 then show ?*case* **using** $c t f$

```

    using ConditionalNode k by blast
next
case (UnaryNodeSame g xe g' x s' op n)
then have k: kind g' n = unary-node op x
    using find-exists-kind local.UnaryNodeSame by blast
then have g' ⊢ x ≃ xe using local.UnaryNodeSame by blast
then show ?case using k
    apply (cases op)
    using AbsNode unary-node.simps(1) apply presburger
    using NegateNode unary-node.simps(3) apply presburger
    using NotNode unary-node.simps(2) apply presburger
    using LogicNegationNode unary-node.simps(4) apply presburger
    using NarrowNode unary-node.simps(5) apply presburger
    using SignExtendNode unary-node.simps(6) apply presburger
    using ZeroExtendNode unary-node.simps(7) by presburger
next
case (UnaryNodeNew g xe g2 x s' op n g')
moreover have unary-node op x ≠ NoNode
    using unary-node.elims by blast
ultimately have k: kind g' n = unary-node op x
    using find-new-kind local.UnaryNodeNew
    by presburger
have x ∈ ids g2 using local.UnaryNodeNew
    using eval-contains-id by blast
then have x ≠ n using local.UnaryNodeNew(5) fresh-ids by blast
have g' ⊢ x ≃ xe using local.UnaryNodeNew fresh-node-preserves-other-nodes
    using ⟨x ∈ ids g2⟩ by blast
then show ?case using k
    apply (cases op)
    using AbsNode unary-node.simps(1) apply presburger
    using NegateNode unary-node.simps(3) apply presburger
    using NotNode unary-node.simps(2) apply presburger
    using LogicNegationNode unary-node.simps(4) apply presburger
    using NarrowNode unary-node.simps(5) apply presburger
    using SignExtendNode unary-node.simps(6) apply presburger
    using ZeroExtendNode unary-node.simps(7) by presburger
next
case (BinaryNodeSame g xe g2 x ye g3 y s' op n)
then have k: kind g3 n = bin-node op x y
    using find-exists-kind by blast
have x: g3 ⊢ x ≃ xe using local.BinaryNodeSame unrep-unchanged
    using no-encoding by blast
have y: g3 ⊢ y ≃ ye using local.BinaryNodeSame unrep-unchanged
    using no-encoding by blast
then show ?case using x y k apply (cases op)
    using AddNode bin-node.simps(1) apply presburger
    using MulNode bin-node.simps(2) apply presburger
    using SubNode bin-node.simps(3) apply presburger
    using AndNode bin-node.simps(4) apply presburger

```

```

    using OrNode bin-node.simps(5) apply presburger
    using XorNode bin-node.simps(6) apply presburger
    using ShortCircuitOrNode bin-node.simps(7) apply presburger
    using LeftShiftNode bin-node.simps(8) apply presburger
    using RightShiftNode bin-node.simps(9) apply presburger
    using UnsignedRightShiftNode bin-node.simps(10) apply presburger
    using IntegerEqualsNode bin-node.simps(11) apply presburger
    using IntegerLessThanNode bin-node.simps(12) apply presburger
    using IntegerBelowNode bin-node.simps(13) by presburger
next
case (BinaryNodeNew g xe g2 x ye g3 y s' op n g')
moreover have bin-node op x y  $\neq$  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 e₁
 assumes kind g n' = RefNode n
 shows g \vdash n' \trianglelefteq e₁
 using assms RefNode

```

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

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

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

lemma add-node-as-set:
  assumes  $g' = \text{add-node } n \ (k, s) \ g$ 
  shows  $(\{n\} \sqsubseteq \text{as-set } g) \subseteq \text{as-set } g'$ 
  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  $(g_2 \vdash n' \sqsubseteq e_1) \wedge \text{graph-refinement } g_1 \ g_2$ 
  using assms
  using graph-construction term-graph-reconstruction by blast

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

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

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

```

```

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

```

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  $\neg(\text{is-preevaluated } (\text{kind } g \ n))$ 
  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  $\forall n' \in \text{true-ids } g. (\forall e. ((g \vdash n \simeq e) \wedge (g \vdash n' \simeq e)) \longrightarrow n = n')$ 
using assms
using encode-in-ids fresh-ids by blast

```

lemma *true-ids-def*:

```

true-ids g = {n ∈ ids g. ¬(is-RefNode (kind g n)) ∧ ((kind g n) ≠ NoNode)}
unfolding true-ids-def ids-def
using ids-def is-RefNode-def by fastforce

```

lemma *add-node-some-node-def*:

```

assumes k ≠ NoNode
assumes g' = add-node nid (k, s) g
shows g' = Abs-IRGraph ((Rep-IRGraph g)(nid ↦ (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 ≠ NoNode
shows dom (Rep-IRGraph g') = dom (Rep-IRGraph g) ∪ {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 ≠ NoNode
shows nid ∈ ids g'
using assms
using find-new-kind ids-some by presburger

```

lemma *add-node-ids-subset*:

```

assumes n ∈ ids g
assumes g' = add-node n node g
shows ids g' = ids g ∪ {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) re-
place-node.rep-eq replace-node-def sup-bot.right-neutral)

lemma *convert-maximal:*

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

lemma *add-node-set-eq:*

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

lemma *add-node-as-set-eq:*

assumes $g' = \text{add-node } n (k, s) g$
assumes $n \notin \text{ids } g$
shows $\{n\} \sqsubseteq \text{as-set } g' = \text{as-set } g$
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:*

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

lemma *as-set-ids:*

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

lemma *ids-add-update:*

assumes $k \neq \text{NoNode}$
assumes $n \notin \text{ids } g$
assumes $g' = \text{add-node } n (k, s) g$
shows $\text{ids } g' = \text{ids } g \cup \{n\}$
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 $k \neq \text{NoNode}$
assumes $n \notin \text{ids } g$
assumes $g' = \text{add-node } n \ (k, s) \ g$
assumes $\neg(\text{is-RefNode } k)$
shows $\text{true-ids } g' = \text{true-ids } g \cup \{n\}$
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 $(\text{new} \sqsubseteq \text{as-set } g') = \text{as-set } g$
shows $n \in \text{ids } g \longrightarrow n \notin \text{new}$
using *assms*
by (*smt* (*z3*) *as-set-def case-prodD domain-subtraction-def mem-Collect-eq*)

lemma *add-preserves-rep*:

assumes *unchanged*: $(\text{new} \sqsubseteq \text{as-set } g') = \text{as-set } g$
assumes *closed*: *wf-closed* *g*
assumes *existed*: $n \in \text{ids } g$
assumes $g' \vdash n \simeq e$
shows $g \vdash n \simeq e$
proof (*cases* $n \in \text{new}$)
case *True*
have $n \notin \text{ids } g$
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*: $\forall n'. n' \notin \text{new} \longrightarrow \text{kind } g \ n' = \text{kind } g' \ n'$
— 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*: $\forall n' \in \text{ids } g'. n' \notin \text{new} \longrightarrow \text{stamp } g \ n' = \text{stamp } g' \ n'$
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.prem(3) kind-eq by pres-
    burger
    then have isin: n ∈ ids g
      by simp
    have inputs: {c, t, f} = inputs g n
      using kind unfolding inputs.simps using inputs-of-ConditionalNode by simp
    have c ∈ ids g ∧ t ∈ ids g ∧ f ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have c ∉ new ∧ t ∉ new ∧ f ∉ new
      using new-def unchanged by blast
    then show ?case using ConditionalNode apply simp
      using rep.ConditionalNode by presburger
  next
    case (AbsNode n x xe)
    then have kind: kind g n = AbsNode x
      by simp
    then have isin: n ∈ ids g
      by simp
    have inputs: {x} = inputs g n
      using kind unfolding inputs.simps by simp
    have x ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have x ∉ new
      using new-def unchanged by blast
    then show ?case
      using AbsNode
      using rep.AbsNode by presburger
  next
    case (NotNode n x xe)
    then have kind: kind g n = NotNode x
      by simp
    then have isin: n ∈ ids g
      by simp
    have inputs: {x} = inputs g n
      using kind unfolding inputs.simps by simp
    have x ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have x ∉ new
      using new-def unchanged by blast

```

```

then show ?case using NotNode
  using rep.NotNode by presburger
next
case (NegateNode n x xe)
then have kind: kind g n = NegateNode x
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new}$ 
  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:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new}$ 
  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:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using AddNode
  using rep.AddNode by presburger
next
case (MulNode n x y xe ye)

```

```

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

```

```

have inputs: {x, y} = inputs g n
  using kind unfolding inputs.simps by simp
have x ∈ ids g ∧ y ∈ ids g
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have x ∉ new ∧ y ∉ new
  using new-def unchanged by blast
then show ?case using OrNode
  using rep.OrNode by presburger
next
case (XorNode n x y xe ye)
then have kind: kind g n = XorNode x y
  by simp
then have isin: n ∈ ids g
  by simp
have inputs: {x, y} = inputs g n
  using kind unfolding inputs.simps by simp
have x ∈ ids g ∧ y ∈ ids g
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have x ∉ new ∧ y ∉ new
  using new-def unchanged by blast
then show ?case using XorNode
  using rep.XorNode by presburger
next
case (ShortCircuitOrNode n x y xe ye)
then have kind: kind g n = ShortCircuitOrNode x y
  by simp
then have isin: n ∈ ids g
  by simp
have inputs: {x, y} = inputs g n
  using kind unfolding inputs.simps by simp
have x ∈ ids g ∧ y ∈ ids g
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have x ∉ new ∧ y ∉ new
  using new-def unchanged by blast
then show ?case using ShortCircuitOrNode
  using rep.ShortCircuitOrNode by presburger
next
case (LeftShiftNode n x y xe ye)
then have kind: kind g n = LeftShiftNode x y
  by simp
then have isin: n ∈ ids g
  by simp
have inputs: {x, y} = inputs g n
  using kind unfolding inputs.simps by simp
have x ∈ ids g ∧ y ∈ ids g
  using closed unfolding wf-closed-def

```

```

    using isin inputs by blast
  then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
    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:  $\text{kind } g\ n = \text{RightShiftNode } x\ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g\ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  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:  $\text{kind } g\ n = \text{UnsignedRightShiftNode } x\ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g\ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  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:  $\text{kind } g\ n = \text{IntegerBelowNode } x\ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g\ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  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:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps by simp
  have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
    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:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps by simp
  have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
    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:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps by simp
  have  $x \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $x \notin \text{new}$ 
    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:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps by simp
  have  $x \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $x \notin \text{new}$ 
    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:  $\text{kind } g \ n = \text{ZeroExtendNode inputBits resultBits } x$ 
    by simp
  then have isin:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps by simp
  have  $x \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $x \notin \text{new}$ 
    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:  $\text{kind } g \ n = \text{RefNode } n'$ 
    by simp
  then have isin:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{n'\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps by simp
  have  $n' \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $n' \notin \text{new}$ 
    using new-def unchanged by blast
  then show ?case
    using RefNode
    using rep.RefNode by presburger
qed
qed

```

```

lemma not-in-no-rep:
   $n \notin \text{ids } g \implies \forall e. \neg(g \vdash n \simeq e)$ 
  using eval-contains-id by blast

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

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

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

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

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

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

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

```

```

next
  case (ConstantNodeNew g c n g')
  then have dom:  $ids\ g' = ids\ g \cup \{n\}$ 
    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.prem 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.prem Un-insert-right
      add-node-as-set dom inputs.elims insertE not-excluded-keep-type order-trans single-
      tonD subset-insertI succ.elims sup-bot-right)
    next
      case (ConditionalNodeSame g ce g2 c te g3 t fe g4 f s' n)
      then show ?case by blast
    next
      case (ConditionalNodeNew g ce g2 c te g3 t fe g4 f s' n g')
      then have dom:  $ids\ g' = ids\ g4 \cup \{n\}$ 
        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 in-
sert-absorb insert-subset subset-insertI sup-bot-right)
    then show ?case using dom
    by (smt (z3) ConditionalNodeNew.hyps(10) ConditionalNodeNew.hyps(2) Con-
ditionalNodeNew.hyps(4) ConditionalNodeNew.hyps(6) ConditionalNodeNew.prem
Diff-eq-empty-iff Diff-iff Un-insert-right Un-upper1 add-node-def inputs.simps in-
sertE replace-node-def replace-node-unchanged subset-trans succ.simps sup-bot-right)
  next
    case (UnaryNodeSame g xe g2 x s' op n)
    then show ?case by blast
  next
    case (UnaryNodeNew g xe g2 x s' op n g')
    then have dom:  $ids\ g' = ids\ g2 \cup \{n\}$ 
    by (metis add-node-ids-subset add-node-lookup ids-add-update ids-some un-
rep.UnaryNodeNew unrep-contains)
    have k:  $kind\ g'\ n = unary-node\ op\ x$ 
    using UnaryNodeNew add-node-lookup
    by (metis fresh-ids ids-some)
    then have inp:  $\{x\} = inputs\ g'\ n$ 
    using unary-inputs by simp
    from k have suc:  $\{\} = succ\ g'\ n$ 
    using unary-succ by simp
    have  $inputs\ g'\ n \subseteq ids\ g' \wedge succ\ g'\ n \subseteq ids\ g' \wedge kind\ g'\ n \neq NoNode$ 
    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 in-
sert-subsetI not-in-g-inputs subset-iff)
    then show ?case
    by (smt (verit) Un-insert-right UnaryNodeNew.hyps(2) UnaryNodeNew.hyps(6)
UnaryNodeNew.prem
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:  $kind\ g'\ n = bin-node\ op\ x\ y$ 
    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 in-
sert-subsetI not-in-g-inputs subset-iff)
    then show ?case using dom BinaryNodeNew
    by (smt (verit, del-Insts) Diff-eq-empty-iff Diff-iff Un-insert-right Un-upper1
add-node-def inputs.simps insertE replace-node-def replace-node-unchanged subset-trans
succ.simps sup-bot-right)
  next
  case (AllLeafNodes g n s)
  then show ?case
  by blast
qed

```

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

definition *constant-value* **where**

constant-value = (*IntVal32* 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))  $\rightsquigarrow$  (g', n) (is g  $\oplus$  ?e  $\rightsquigarrow$ 
(g', n))
  shows  $\neg(\text{maximal-sharing } g')$ 
  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  $\rightsquigarrow$  (g, y'')
  using assms(5) ConstUnrepE sorry
  then show ?thesis sorry
qed

```

end

lemma *conditional-rep-kind*:

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

lemma *unary-rep-kind*:

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

lemma *binary-rep-kind*:

assumes $g \vdash n \simeq \text{BinaryExpr } op \ xe \ ye$
assumes $g \vdash x \simeq xe$
assumes $g \vdash y \simeq ye$
assumes $\neg(\exists n'. \text{kind } g \ n = \text{RefNode } n')$
shows $\text{kind } g \ n = \text{bin-node } op \ x \ y$
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 $\text{kind } g' \ n = \text{ConstantNode } c$
 using *find-new-kind* **by** *blast*
 then have *repn*: $g' \vdash n \simeq \text{ConstantExpr } c$
 using *rep.ConstantNode* **by** *simp*
 from *ConstantNodeNew* **have** *real-node*: $\neg(\text{is-RefNode } (\text{ConstantNode } c)) \wedge$
ConstantNode $c \neq \text{NoNode}$
 by *simp*
 then have *dom*: $\text{true-ids } g' = \text{true-ids } g \cup \{n\}$
 using *ConstantNodeNew.hyps*(2) *ConstantNodeNew.hyps*(3) *fresh-ids*
 by (*meson* *true-ids-add-update*)
 have *new*: $n \notin \text{ids } g$

```

    using fresh-ids
    using ConstantNodeNew.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  $\leq$  as-set g') = as-set g
    using ConstantNodeNew(3) new add-node-as-set-eq
    by presburger
    then have kind-eq:  $\forall n'. n' \notin \text{new} \longrightarrow \text{kind } g \ n' = \text{kind } g' \ n'$ 
    by (metis ConstantNodeNew.hyps(3)  $\langle \text{new} = \{n\} \rangle$  add-node-as-set dual-order.eq-iff
    not-excluded-keep-type not-in-g)
    from unchanged have stamp-eq:  $\forall n' \in \text{ids } g. n' \notin \text{new} \longrightarrow \text{stamp } g \ n' =$ 
    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 n1 n2 e
    assume 1:  $\forall n_1 \ n_2.$ 
     $n_1 \in \text{true-ids } g \wedge n_2 \in \text{true-ids } g \longrightarrow$ 
     $(\exists e. (g \vdash n_1 \simeq e) \wedge (g \vdash n_2 \simeq e) \wedge \text{stamp } g \ n_1 = \text{stamp } g \ n_2) \longrightarrow n_1 = n_2$ 
    assume n1  $\in \text{true-ids } g'$ 
    assume n2  $\in \text{true-ids } g'$ 
    show  $g' \vdash n_1 \simeq e \implies g' \vdash n_2 \simeq e \implies \text{stamp } g' \ n_1 = \text{stamp } g' \ n_2 \implies n_1 =$ 
    n2
    proof (cases n1  $\in \text{true-ids } g$ )
    case n1: True
    then show  $g' \vdash n_1 \simeq e \implies g' \vdash n_2 \simeq e \implies \text{stamp } g' \ n_1 = \text{stamp } g' \ n_2$ 
 $\implies n_1 = n_2$ 
    proof (cases n2  $\in \text{true-ids } g$ )
    case n2: True
    assume n1rep':  $g' \vdash n_1 \simeq e$ 
    assume n2rep':  $g' \vdash n_2 \simeq e$ 
    assume stmp: stamp g' n1 = stamp g' n2
    have n1rep:  $g \vdash n_1 \simeq e$ 
    using n1rep' kind-eq stamp-eq new-def add-preserves-rep
    using ConstantNodeNew.prem(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.prem(1) IRGraph.true-ids-def n2 unchanged
    by auto
    have stamp g n1 = stamp g n2
    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:  $\text{stamp } g' \ n_1 = \text{stamp } g' \ n_2$ 
have n2-def:  $n_2 = n$ 
  using  $\langle n_2 \in \text{true-ids } g' \rangle \text{ dom } n2$  by auto
have n1rep:  $g \vdash n_1 \simeq \text{ConstantExpr } c$ 
  by (metis (no-types, lifting) ConstantNodeNew.prem(1) DiffE IR-
    Graph.true-ids-def add-preserves-rep n1 n1rep' n2-def n2rep' repDet repn unchanged)
then have n1in:  $n_1 \in \text{ids } g$ 
  using no-encoding by metis
have k:  $\text{kind } g \ n_1 = \text{ConstantNode } c$ 
  using TreeToGraphThms.true-ids-def n1 n1rep by force
have s:  $\text{stamp } g \ n_1 = \text{constantAsStamp } c$ 
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  $g' \vdash n_1 \simeq e \implies g' \vdash n_2 \simeq e \implies \text{stamp } g' \ n_1 = \text{stamp } g' \ n_2$ 
 $\implies n_1 = n_2$ 
proof (cases  $n_2 \in \text{true-ids } g$ )
case n2: True
assume n1rep':  $g' \vdash n_1 \simeq e$ 
assume n2rep':  $g' \vdash n_2 \simeq e$ 
assume stmp:  $\text{stamp } g' \ n_1 = \text{stamp } g' \ n_2$ 
have n1-def:  $n_1 = n$ 
  using  $\langle n_1 \in \text{true-ids } g' \rangle \text{ dom } n1$  by auto
have n2in:  $n_2 \in \text{ids } g$ 
  using IRGraph.true-ids-def n2 by auto
have k:  $\text{kind } g \ n_2 = \text{ConstantNode } c$ 
  by (metis (mono-tags, lifting) ConstantNodeE ConstantNodeNew.prem(1)
    DiffE IRGraph.true-ids-def add-preserves-rep mem-Collect-eq n1-def n1rep' n2 n2rep'
    repDet repn unchanged)
have s:  $\text{stamp } g \ n_2 = \text{constantAsStamp } c$ 
  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 stmp:  $\text{stamp } g' \ n_1 = \text{stamp } g' \ n_2$ 

```

```

      have  $n_1 = n \wedge n_2 = n$ 
      using  $\langle n_1 \in \text{true-ids } g' \rangle \text{ dom } n1$ 
      using  $\langle n_2 \in \text{true-ids } g' \rangle n2$  by blast
      then show ?thesis
      by simp
    qed
  qed
next
case (ParameterNodeSame  $g \ i \ s \ n$ )
then show ?case by blast
next
case (ParameterNodeNew  $g \ i \ s \ n \ g'$ )
then have  $k: \text{kind } g' \ n = \text{ParameterNode } i$ 
  using find-new-kind by blast
have  $\text{stamp } g' \ n = s$ 
  using ParameterNodeNew.hyps(3) find-new-stamp by blast
then have  $\text{repn}: g' \vdash n \simeq \text{ParameterExpr } i \ s$ 
  using rep.ParameterNode  $k$  by simp
from ConstantNodeNew have  $\neg(\text{is-RefNode } (\text{ParameterNode } i)) \wedge \text{ParameterNode } i \neq \text{NoNode}$ 
  by simp
then have  $\text{dom}: \text{true-ids } g' = \text{true-ids } g \cup \{n\}$ 
  using ParameterNodeNew.hyps(2) ParameterNodeNew.hyps(3) fresh-ids
  by (meson true-ids-add-update)
have  $\text{new}: n \notin \text{ids } g$ 
  using fresh-ids
  using ParameterNodeNew.hyps(2) by blast
obtain new where  $\text{new} = \text{true-ids } g' - \text{true-ids } g$ 
  by simp
then have new-def:  $\text{new} = \{n\}$ 
  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:  $(\text{new} \sqsubseteq \text{as-set } g') = \text{as-set } g$ 
  using ParameterNodeNew(3) new add-node-as-set-eq
  by presburger
then have kind-eq:  $\forall n'. n' \notin \text{new} \longrightarrow \text{kind } g \ n' = \text{kind } g' \ n'$ 
  by (metis ParameterNodeNew.hyps(3)  $\langle \text{new} = \{n\} \rangle \text{add-node-as-set dual-order.eq-iff}$ 
    not-excluded-keep-type not-in-g)
from unchanged have stamp-eq:  $\forall n' \in \text{ids } g. n' \notin \text{new} \longrightarrow \text{stamp } g \ n' = \text{stamp } g' \ n'$ 
  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:  $\forall n_1 \ n_2. n_1 \in \text{true-ids } g \wedge n_2 \in \text{true-ids } g \longrightarrow$ 

```

```

      (∃ e. (g ⊢ n₁ ≃ e) ∧ (g ⊢ n₂ ≃ e) ∧ stamp g n₁ = stamp g n₂) ⟶ n₁ = n₂
    assume n₁ ∈ true-ids g'
    assume n₂ ∈ true-ids g'
    show g' ⊢ n₁ ≃ e ⟹ g' ⊢ n₂ ≃ e ⟹ stamp g' n₁ = stamp g' n₂ ⟹ n₁ =
n₂
    proof (cases n₁ ∈ true-ids g)
      case n1: True
        then show g' ⊢ n₁ ≃ e ⟹ g' ⊢ n₂ ≃ e ⟹ stamp g' n₁ = stamp g' n₂
    ⟹ n₁ = n₂
      proof (cases n₂ ∈ true-ids g)
        case n2: True
          assume n1rep': g' ⊢ n₁ ≃ e
          assume n2rep': g' ⊢ n₂ ≃ e
          assume stamp g' n₁ = stamp g' n₂
          have n1rep: g ⊢ n₁ ≃ e
            using n1rep' kind-eq stamp-eq new-def add-preserves-rep
            using ParameterNodeNew.premis(1) IRGraph.true-ids-def n1 unchanged
        by auto
          have n2rep: g ⊢ n₂ ≃ e
            using n2rep' kind-eq stamp-eq new-def add-preserves-rep
            using ParameterNodeNew.premis(1) IRGraph.true-ids-def n2 unchanged
        by auto
          have stamp g n₁ = stamp g n₂
            by (metis ParameterNodeNew.hyps(3) ⟨stamp g' n₁ = stamp g' n₂⟩
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' ⊢ n₁ ≃ e
            assume n2rep': g' ⊢ n₂ ≃ e
            assume stamp g' n₁ = stamp g' n₂
            have n₂ = n
              using ⟨n₂ ∈ true-ids g'⟩ dom n2 by auto
            then have ne: n₂ ∉ ids g
              using new n2 by blast
            have n1rep: g ⊢ n₁ ≃ e
              using n1rep' kind-eq stamp-eq new-def add-preserves-rep
              using ParameterNodeNew.premis(1) IRGraph.true-ids-def n1 unchanged
        by auto
          have n2rep: g ⊢ n₂ ≃ e
            using n2rep' kind-eq stamp-eq new-def add-preserves-rep
            using ParameterNodeNew.premis(1) IRGraph.true-ids-def unchanged
            by (metis (no-types, lifting) IRNode.disc(2703) ParameterNodeE
ParameterNodeNew.hyps(1) TreeToGraphThms.true-ids-def ⟨n₂ = 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  $g' \vdash n_1 \simeq e \implies g' \vdash n_2 \simeq e \implies \text{stamp } g' n_1 = \text{stamp } g' n_2$ 
 $\implies n_1 = n_2$ 
    proof (cases  $n_2 \in \text{true-ids } g$ )
      case n2: True
      assume n1rep':  $g' \vdash n_1 \simeq e$ 
      assume n2rep':  $g' \vdash n_2 \simeq e$ 
      assume  $\text{stamp } g' n_1 = \text{stamp } g' n_2$ 
      have  $n_1 = n$ 
      using  $\langle n_1 \in \text{true-ids } g' \rangle \text{ dom } n1$  by auto
      then have ne:  $n_1 \notin \text{ids } g$ 
      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.premis(1) IRGraph.true-ids-def n1 unchanged
      by (metis (no-types, lifting) IRNode.disc(2703) ParameterNodeE
        ParameterNodeNew.hyps(1) TreeToGraphThms.true-ids-def  $\langle n_1 = n \rangle$  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  $\text{stamp } g' n_1 = \text{stamp } g' n_2$ 
      have  $n_1 = n \wedge n_2 = n$ 
      using  $\langle n_1 \in \text{true-ids } g' \rangle \text{ dom } n1$ 
      using  $\langle n_2 \in \text{true-ids } g' \rangle n2$  by blast
      then show ?thesis
      by simp
    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  $g' n = \text{ConditionalNode } c t f$ 
  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:  $g' \vdash n \simeq \text{ConditionalExpr } ce te fe$ 
  using rep.ConditionalNode k
  by (metis ConditionalNodeNew.hyps(1) ConditionalNodeNew.hyps(10) Condi-

```

$\text{ConditionalNodeNew.hyps}(3) \text{ ConditionalNodeNew.hyps}(5) \text{ ConditionalNodeNew.hyps}(9)$
 $\text{fresh-ids fresh-node-subset subset-implies-evals term-graph-reconstruction})$
from ConstantNodeNew **have** $\neg(\text{is-RefNode } (\text{ConditionalNode } c \ t \ f)) \wedge \text{ConditionalNode } c \ t \ f \neq \text{NoNode}$
by *simp*
then have $\text{dom: true-ids } g' = \text{true-ids } g_4 \cup \{n\}$
using $\text{ConditionalNodeNew.hyps}(10) \text{ ConditionalNodeNew.hyps}(9) \text{ fresh-ids}$
 $\text{true-ids-add-update}$ **by** *presburger*
have $\text{new: } n \notin \text{ids } g$
using *fresh-ids*
by ($\text{meson } \text{ConditionalNodeNew.hyps}(1) \text{ ConditionalNodeNew.hyps}(3) \text{ ConditionalNodeNew.hyps}(5) \text{ ConditionalNodeNew.hyps}(9) \text{ unrep-preserves-contains})$
obtain new **where** $\text{new} = \text{true-ids } g' - \text{true-ids } g_4$
by *simp*
then have $\text{new-def: new} = \{n\}$
using *dom*
by ($\text{metis } \text{ConditionalNodeNew.hyps}(9) \text{ DiffD1 DiffI Diff-cancel Diff-insert}$
 $\text{Un-insert-right boolean-algebra.disj-zero-right fresh-ids insertCI insert-Diff true-ids})$
then have $\text{unchanged: } (\text{new} \triangleleft \text{as-set } g^\wedge) = \text{as-set } g_4$
using $\text{new add-node-as-set-eq}$
using $\text{ConditionalNodeNew.hyps}(10) \text{ ConditionalNodeNew.hyps}(9) \text{ fresh-ids}$
by *presburger*
then have $\text{kind-eq: } \forall n'. n' \notin \text{new} \longrightarrow \text{kind } g_4 \ n' = \text{kind } g' \ n'$
by ($\text{metis } \text{ConditionalNodeNew.hyps}(10) \text{ add-node-as-set equalityE local.new-def}$
 $\text{not-excluded-keep-type not-in-g})$
from unchanged **have** $\text{stamp-eq: } \forall n' \in \text{ids } g. n' \notin \text{new} \longrightarrow \text{stamp } g_4 \ n' = \text{stamp } g' \ n'$
using $\text{not-excluded-keep-type new-def new}$
by ($\text{metis } \text{ConditionalNodeNew.hyps}(1) \text{ ConditionalNodeNew.hyps}(10) \text{ ConditionalNodeNew.hyps}(3) \text{ ConditionalNodeNew.hyps}(5) \text{ add-node-as-set unrep-preserves-contains})$
have max-g_4 : $\text{maximal-sharing } g_4$
using $\text{ConditionalNodeNew.hyps}(1) \text{ ConditionalNodeNew.hyps}(2) \text{ ConditionalNodeNew.hyps}(3) \text{ ConditionalNodeNew.hyps}(4) \text{ ConditionalNodeNew.hyps}(6) \text{ ConditionalNodeNew.premis}(1) \text{ ConditionalNodeNew.premis}(2) \text{ unrep-preserves-closure}$
by *blast*
show *?case unfolding maximal-sharing apply (rule allI; rule allI; rule impI)*
using $\text{max-g}_4 \text{ unfolding maximal-sharing apply auto}$
proof –
fix $n_1 \ n_2 \ e$
assume $1: \forall n_1 \ n_2.$
 $n_1 \in \text{true-ids } g_4 \wedge n_2 \in \text{true-ids } g_4 \longrightarrow$
 $(\exists e. (g_4 \vdash n_1 \simeq e) \wedge (g_4 \vdash n_2 \simeq e) \wedge \text{stamp } g_4 \ n_1 = \text{stamp } g_4 \ n_2) \longrightarrow$
 $n_1 = n_2$
assume $n_1 \in \text{true-ids } g'$
assume $n_2 \in \text{true-ids } g'$
show $g' \vdash n_1 \simeq e \implies g' \vdash n_2 \simeq e \implies \text{stamp } g' \ n_1 = \text{stamp } g' \ n_2 \implies n_1 =$
 n_2
proof ($\text{cases } n_1 \in \text{true-ids } g_4$)
case $n1$: *True*

```

then show  $g' \vdash n_1 \simeq e \implies g' \vdash n_2 \simeq e \implies \text{stamp } g' n_1 = \text{stamp } g' n_2$ 
 $\implies n_1 = n_2$ 
proof (cases  $n_2 \in \text{true-ids } g_4$ )
  case  $n_2$ : True
    assume  $n1rep'$ :  $g' \vdash n_1 \simeq e$ 
    assume  $n2rep'$ :  $g' \vdash n_2 \simeq e$ 
    assume  $\text{stamp } g' n_1 = \text{stamp } g' n_2$ 
    have  $n1rep$ :  $g_4 \vdash n_1 \simeq e$ 
      using  $n1rep'$  kind-eq stamp-eq new-def add-preserves-rep
    using ConditionalNodeNew.premis(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  $n2rep$ :  $g_4 \vdash n_2 \simeq e$ 
      using  $n2rep'$  kind-eq stamp-eq new-def add-preserves-rep
    using ConditionalNodeNew.premis(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  $\text{stamp } g_4 n_1 = \text{stamp } g_4 n_2$ 
      by (metis ConditionalNodeNew.hyps(10) ConditionalNodeNew.hyps(9)
         $\langle \text{stamp } g' n_1 = \text{stamp } g' n_2 \rangle$  fresh-ids fresh-node-subset n1rep n2rep subset-stamp)
    then show ?thesis using 1
      using  $n1 n2$ 
      using  $n1rep n2rep$  by blast
  next
    case  $n_2$ : False
      assume  $n1rep'$ :  $g' \vdash n_1 \simeq e$ 
      assume  $n2rep'$ :  $g' \vdash n_2 \simeq e$ 
      assume  $\text{stamp}$ :  $\text{stamp } g' n_1 = \text{stamp } g' n_2$ 
      have  $n2\text{-def}$ :  $n_2 = n$ 
        using  $\langle n_2 \in \text{true-ids } g' \rangle$  dom n2 by auto
      have  $n1rep$ :  $g_4 \vdash n_1 \simeq \text{ConditionalExpr } ce \text{ te } fe$ 
      by (metis (no-types, lifting) ConditionalNodeNew.hyps(1) ConditionalNodeNew.hyps(3) ConditionalNodeNew.hyps(5) ConditionalNodeNew.premis(1) Diff-iff IRGraph.true-ids-def add-preserves-rep n1 n1rep' n2-def n2rep' repDet repn unchanged unrep-preserves-closure)
      then have  $n1in$ :  $n_1 \in \text{ids } g_4$ 
        using no-encoding by metis

      have  $\text{rep}$ :  $(g_4 \vdash c \simeq ce) \wedge (g_4 \vdash t \simeq te) \wedge (g_4 \vdash f \simeq fe)$ 
        by (meson ConditionalNodeNew.hyps(1) ConditionalNodeNew.hyps(3) ConditionalNodeNew.hyps(5) subset-implies-evals term-graph-reconstruction)
      have not-ref:  $\neg(\exists n'. \text{kind } g_4 n_1 = \text{RefNode } n')$ 
        using TreeToGraphThms.true-ids-def n1 by fastforce
      then have  $\text{kind } g_4 n_1 = \text{ConditionalNode } c \text{ t } f$ 
        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))

```

```

  ⟨stamp g' n = s'⟩ fresh-ids fresh-node-subset n1rep n2-def stmp subset-stamp)
  qed
next
  case n1: False
  then show g' ⊢ n1 ≃ e ⇒ g' ⊢ n2 ≃ e ⇒ stamp g' n1 = stamp g' n2
⇒ n1 = n2
  proof (cases n2 ∈ true-ids g4)
  case n2: True
  assume n1rep': g' ⊢ n1 ≃ e
  assume n2rep': g' ⊢ n2 ≃ e
  assume stamp g' n1 = stamp g' n2
  have new-n1: n1 = n
  using ⟨n1 ∈ true-ids g'⟩ dom n1 by auto
  then have ne: n1 ∉ ids g4
  using new n1
  using ConditionalNodeNew.hyps(9) fresh-ids by blast
  have unrep-cond: g4 ⊢ n2 ≃ ConditionalExpr ce te fe
  using n1rep' kind-eq stamp-eq new-def add-preserves-rep
  using ConditionalNodeNew.prem(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 ⊢ c ≃ ce) ∧ (g4 ⊢ t ≃ te) ∧ (g4 ⊢ f ≃ fe)
  by (meson ConditionalNodeNew.hyps(1) ConditionalNodeNew.hyps(3) ConditionalNodeNew.hyps(5) subset-implies-evals term-graph-reconstruction)
  have not-ref: ¬(∃ n'. kind g4 n2 = RefNode n')
  using TreeToGraphThms.true-ids-def n2 by fastforce
  then have kind g4 n2 = 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) ⟨stamp g' n = s'⟩ ⟨stamp g' n1 = stamp g' n2⟩ encodes-contains fresh-node-subset ne new-n1 not-in-g subset-stamp unrep-cond)
  next
  case n2: False
  assume n1rep': g' ⊢ n1 ≃ e
  assume n2rep': g' ⊢ n2 ≃ e
  assume stamp g' n1 = stamp g' n2
  have n1 = n ∧ n2 = n
  using ⟨n1 ∈ true-ids g'⟩ dom n1
  using ⟨n2 ∈ true-ids g'⟩ 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' ⊢ n ≃ 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.prem(1) UnaryNode-
    New.prem(2))
  show ?case unfolding maximal-sharing apply (rule allI; rule allI; rule impI)
  using max-g2 unfolding maximal-sharing apply auto
  proof −
  fix n1 n2 e
  assume 1: ∀ n1 n2.

```


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

```

then have kind  $g2 \ n_1 = \text{unary-node } op \ x$ 
  using unrep-un unary-rep-kind rep by simp

then show ?thesis using find-none UnaryNodeNew.hyps(4)
  by (metis UnaryNodeNew.hyps(6) ⟨stamp  $g' \ n = s' \rangle \langle stamp \ g' \ n_1 =$ 
stamp  $g' \ n_2 \rangle \text{fresh-node-subset } ne \ \text{new-n2} \ \text{no-encoding subset-stamp unrep-un}$ )
qed
next
case n1: False
  then show  $g' \vdash n_1 \simeq e \implies g' \vdash n_2 \simeq e \implies \text{stamp } g' \ n_1 = \text{stamp } g' \ n_2$ 
 $\implies n_1 = n_2$ 
  proof (cases  $n_2 \in \text{true-ids } g2$ )
    case n2: True
      assume n1rep':  $g' \vdash n_1 \simeq e$ 
      assume n2rep':  $g' \vdash n_2 \simeq e$ 
      assume stamp  $g' \ n_1 = \text{stamp } g' \ n_2$ 
      have new-n1:  $n_1 = n$ 
        using ⟨ $n_1 \in \text{true-ids } g' \rangle \text{dom } n1$  by auto
      then have ne:  $n_1 \notin \text{ids } g2$ 
        using new n1
        using UnaryNodeNew.hyps(5) fresh-ids by blast
      have unrep-un:  $g2 \vdash n_2 \simeq \text{UnaryExpr } op \ xe$ 
        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.prem(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:  $\neg(\exists n'. \text{kind } g2 \ n_2 = \text{RefNode } n')$ 
        using TreeToGraphThms.true-ids-def n2 by fastforce
      then have kind  $g2 \ n_2 = \text{unary-node } op \ x$ 
        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  $g' \ n = s' \rangle \langle stamp \ g' \ n_1 =$ 
stamp  $g' \ n_2 \rangle \text{fresh-node-subset } ne \ \text{new-n1} \ \text{no-encoding subset-stamp unrep-un}$ )
      next
        case n2: False
          assume n1rep':  $g' \vdash n_1 \simeq e$ 
          assume n2rep':  $g' \vdash n_2 \simeq e$ 
          assume stamp  $g' \ n_1 = \text{stamp } g' \ n_2$ 
          have  $n_1 = n \wedge n_2 = n$ 
            using ⟨ $n_1 \in \text{true-ids } g' \rangle \text{dom } n1$ 
            using ⟨ $n_2 \in \text{true-ids } g' \rangle \text{dom } n2$ 
            by simp
          then show ?thesis
            by simp
          qed
        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' ⊢ n ≃ 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.premis(1) BinaryNodeNew.premis(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 n1 n2 e
assume 1: ∀ n1 n2.
  n1 ∈ true-ids g3 ∧ n2 ∈ true-ids g3 →
  (∃ e. (g3 ⊢ n1 ≃ e) ∧ (g3 ⊢ n2 ≃ e) ∧ stamp g3 n1 = stamp g3 n2) →
n1 = n2
  assume n1 ∈ true-ids g'
  assume n2 ∈ true-ids g'
  show g' ⊢ n1 ≃ e ⇒ g' ⊢ n2 ≃ e ⇒ stamp g' n1 = stamp g' n2 ⇒ n1 =
n2
  proof (cases n1 ∈ true-ids g3)
    case n1: True
      then show g' ⊢ n1 ≃ e ⇒ g' ⊢ n2 ≃ e ⇒ stamp g' n1 = stamp g' n2
⇒ n1 = n2
      proof (cases n2 ∈ true-ids g3)
        case n2: True
          assume n1rep': g' ⊢ n1 ≃ e
          assume n2rep': g' ⊢ n2 ≃ e
          assume stamp g' n1 = stamp g' n2
          have n1rep: g3 ⊢ n1 ≃ e
            using n1rep' kind-eq stamp-eq new-def add-preserves-rep
            by (metis (no-types, lifting) BinaryNodeNew.hyps(1) BinaryNode-
New.hyps(3) BinaryNodeNew.prem(1) Diff-iff IRGraph.true-ids-def n1 unchanged
unrep-preserves-closure)
          have n2rep: g3 ⊢ n2 ≃ e
            using n2rep' kind-eq stamp-eq new-def add-preserves-rep
            by (metis BinaryNodeNew.hyps(1) BinaryNodeNew.hyps(3) BinaryNode-
New.prem(1) DiffE n2 true-ids unchanged unrep-preserves-closure)
          have stamp g3 n1 = stamp g3 n2
            by (metis BinaryNodeNew.hyps(7) BinaryNodeNew.hyps(8) ⟨stamp g' n1
= stamp g' n2⟩ 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' ⊢ n1 ≃ e
            assume n2rep': g' ⊢ n2 ≃ e
            assume stamp g' n1 = stamp g' n2
            have new-n2: n2 = n
              using ⟨n2 ∈ true-ids g'⟩ dom n2 by auto
            then have ne: n2 ∉ ids g3
              using new n2
              using BinaryNodeNew.hyps(7) fresh-ids by presburger
            have unrep-bin: g3 ⊢ n1 ≃ 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.prem(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 n1 = RefNode n')
    using TreeToGraphThms.true-ids-def n1 by force
    then have kind g3 n1 = 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' n1 =
stamp g' n2⟩ fresh-node-subset ne new-n2 no-encoding subset-stamp unrep-bin)
    qed
  next
  case n1: False
  then show g' ⊢ n1 ≃ e ⇒ g' ⊢ n2 ≃ e ⇒ stamp g' n1 = stamp g' n2
⇒ n1 = n2
  proof (cases n2 ∈ true-ids g3)
    case n2: True
    assume n1rep': g' ⊢ n1 ≃ e
    assume n2rep': g' ⊢ n2 ≃ e
    assume stamp g' n1 = stamp g' n2
    have new-n1: n1 = n
    using ⟨n1 ∈ true-ids g'⟩ dom n1 by auto
    then have ne: n1 ∉ ids g3
    using new n1
    using BinaryNodeNew.hyps(7) fresh-ids by blast
    have unrep-bin: g3 ⊢ n2 ≃ BinaryExpr op xe ye
    using n1rep' kind-eq stamp-eq new-def add-preserves-rep
    by (metis (mono-tags, lifting) BinaryNodeNew.hyps(1) BinaryN-
odeNew.hyps(3) BinaryNodeNew.prem(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 n2 = RefNode n')
    using TreeToGraphThms.true-ids-def n2 by fastforce
    then have kind g3 n2 = 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' n1 =
stamp g' n2⟩ fresh-node-subset ne new-n1 no-encoding subset-stamp unrep-bin)
  next
  case n2: False
  assume n1rep': g' ⊢ n1 ≃ e
  assume n2rep': g' ⊢ n2 ≃ e
  assume stamp g' n1 = stamp g' n2
  have n1 = n ∧ n2 = n
  using ⟨n1 ∈ true-ids g'⟩ dom n1

```

```

      using ⟨ $n_2 \in \text{true-ids } g'$ ⟩  $n_2$ 
      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  $\Rightarrow$  'b  $\Rightarrow$  Value
type-synonym Free = nat
type-synonym ('a, 'b) DynamicHeap = ('a, 'b) Heap  $\times$  Free

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

fun h-store-field :: 'a  $\Rightarrow$  'b  $\Rightarrow$  Value  $\Rightarrow$  ('a, 'b) DynamicHeap  $\Rightarrow$  ('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  $\Rightarrow$  ('a, 'b) DynamicHeap  $\times$  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

```

$new_heap = ((\lambda f. \lambda p. \text{UndefVal}), 0)$

8.2 Intraprocedural Semantics

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

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

fun $input_index :: \text{IRGraph} \Rightarrow \text{ID} \Rightarrow \text{ID} \Rightarrow \text{nat}$ **where**
 $input_index\ g\ n\ n' = find_index\ n' (inputs_of\ (kind\ g\ n))$

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

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

Intraprocedural semantics are given as a small-step semantics.

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

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

SequentialNode:

$\llbracket is_sequential_node\ (kind\ g\ nid);$
 $nid' = (successors_of\ (kind\ g\ nid))!0 \rrbracket$
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \mid$

IfNode:

$\llbracket kind\ g\ nid = (IfNode\ cond\ tb\ fb);$
 $g \vdash cond \simeq condE;$
 $[m, p] \vdash condE \mapsto val;$
 $nid' = (\text{if } val_to_bool\ val \text{ then } tb \text{ else } fb) \rrbracket$
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \mid$

EndNodes:

$\llbracket is_AbstractEndNode\ (kind\ g\ nid);$
 $merge = any_usage\ g\ nid;$
 $is_AbstractMergeNode\ (kind\ g\ merge);$

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

$m' = \text{set-phis } phis \text{ vs } m$
 $\implies g, p \vdash (nid, m, h) \rightarrow (\text{merge}, m', h) \mid$

NewInstanceNode:

$\llbracket \text{kind } g \text{ } nid = (\text{NewInstanceNode } nid \text{ } f \text{ } obj \text{ } nid') \rrbracket;$
 $(h', \text{ref}) = h\text{-new-inst } h;$
 $m' = m(nid := \text{ref})$
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m', h') \mid$

LoadFieldNode:

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

SignedDivNode:

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

SignedRemNode:

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

StaticLoadFieldNode:

$\llbracket \text{kind } g \text{ } nid = (\text{LoadFieldNode } nid \text{ } f \text{ } \text{None} \text{ } nid') \rrbracket;$
 $h\text{-load-field } f \text{ } \text{None} \text{ } h = v;$
 $m' = m(nid := v)$
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m', h) \mid$

StoreFieldNode:

$$\begin{aligned} & \llbracket \text{kind } g \text{ nid} = (\text{StoreFieldNode } \text{nid } f \text{ newval} - (\text{Some } \text{obj}) \text{ nid}') \rrbracket; \\ & g \vdash \text{newval} \simeq \text{newval}E; \\ & g \vdash \text{obj} \simeq \text{obj}E; \\ & [m, p] \vdash \text{newval}E \mapsto \text{val}; \\ & [m, p] \vdash \text{obj}E \mapsto \text{ObjRef } \text{ref}; \\ & h' = h\text{-store-field } f \text{ ref val } h; \\ & m' = m(\text{nid} := \text{val}) \\ \implies & g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m', h') \mid \end{aligned}$$

StaticStoreFieldNode:

$$\begin{aligned} & \llbracket \text{kind } g \text{ nid} = (\text{StoreFieldNode } \text{nid } f \text{ newval} - \text{None } \text{nid}') \rrbracket; \\ & g \vdash \text{newval} \simeq \text{newval}E; \\ & [m, p] \vdash \text{newval}E \mapsto \text{val}; \\ & h' = h\text{-store-field } f \text{ None val } h; \\ & m' = m(\text{nid} := \text{val}) \\ \implies & g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m', h') \end{aligned}$$

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

8.3 Interprocedural Semantics

type-synonym *Signature* = *string*

type-synonym *Program* = *Signature* \rightarrow *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:

$$\begin{aligned} & \llbracket g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m', h') \rrbracket \\ \implies & P \vdash ((g, \text{nid}, m, p) \# \text{stk}, h) \longrightarrow ((g, \text{nid}', m', p) \# \text{stk}, h') \mid \end{aligned}$$

InvokeNodeStep:

$\llbracket \text{is-Invoke } (\text{kind } g \text{ nid}) \rrbracket;$

$$\begin{aligned} & \text{callTarget} = \text{ir-callTarget } (\text{kind } g \text{ nid}); \\ & \text{kind } g \text{ callTarget} = (\text{MethodCallTargetNode } \text{targetMethod } \text{arguments}); \\ & \text{Some } \text{targetGraph} = P \text{ targetMethod}; \\ & m' = \text{new-map-state}; \\ & g \vdash \text{arguments} \simeq_L \text{args}E; \\ & [m, p] \vdash \text{args}E \mapsto_L p \\ \implies & P \vdash ((g, \text{nid}, m, p) \# \text{stk}, h) \longrightarrow ((\text{targetGraph}, 0, m', p') \# (g, \text{nid}, m, p) \# \text{stk}, h) \end{aligned}$$

|

ReturnNode:

$\llbracket \text{kind } g \text{ nid} = (\text{ReturnNode } (\text{Some } \text{expr}) \text{ -}) \rrbracket$;

$g \vdash \text{expr} \simeq e$;

$[m, p] \vdash e \mapsto v$;

$cm' = cm(\text{cnid} := v)$;

$\text{cnid}' = (\text{successors-of } (\text{kind } cg \text{ cnid}))!0$

$\implies P \vdash ((g, \text{nid}, m, p) \# (cg, \text{cnid}, cm, cp) \# \text{stk}, h) \longrightarrow ((cg, \text{cnid}', cm', cp) \# \text{stk}, h) \mid$

ReturnNodeVoid:

$\llbracket \text{kind } g \text{ nid} = (\text{ReturnNode } \text{None} \text{ -}) \rrbracket$;

$cm' = cm(\text{cnid} := (\text{ObjRef } (\text{Some } (2048))))$;

$\text{cnid}' = (\text{successors-of } (\text{kind } cg \text{ cnid}))!0$

$\implies P \vdash ((g, \text{nid}, m, p) \# (cg, \text{cnid}, cm, cp) \# \text{stk}, h) \longrightarrow ((cg, \text{cnid}', cm', cp) \# \text{stk}, h) \mid$

UnwindNode:

$\llbracket \text{kind } g \text{ nid} = (\text{UnwindNode } \text{exception}) \rrbracket$;

$g \vdash \text{exception} \simeq \text{exceptionE}$;

$[m, p] \vdash \text{exceptionE} \mapsto e$;

$\text{kind } cg \text{ cnid} = (\text{InvokeWithExceptionNode} \text{ - - - - - } \text{exEdge})$;

$cm' = cm(\text{cnid} := e)$

$\implies P \vdash ((g, \text{nid}, m, p) \# (cg, \text{cnid}, cm, cp) \# \text{stk}, h) \longrightarrow ((cg, \text{exEdge}, cm', cp) \# \text{stk}, h)$

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

8.4 Big-step Execution

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

fun *has-return* :: *MapState* \Rightarrow *bool* **where**

has-return *m* = (*m* 0 \neq *UndefVal*)

inductive *exec* :: *Program*

\Rightarrow (*IRGraph* \times *ID* \times *MapState* \times *Params*) *list* \times *FieldRefHeap*

\Rightarrow *Trace*

\Rightarrow (*IRGraph* \times *ID* \times *MapState* \times *Params*) *list* \times *FieldRefHeap*

\Rightarrow *Trace*

\Rightarrow *bool*

($- \vdash - \mid - \longrightarrow * - \mid -$)

for *P*

where

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

$\neg(\text{has-return } m')$;

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

$$\begin{aligned} & \text{exec } P \ ((g', \text{nid}', m', p') \# \text{ys}), h') \ l' \ \text{next-state } l'' \rceil \\ & \implies \text{exec } P \ ((g, \text{nid}, m, p) \# \text{xs}), h) \ l \ \text{next-state } l'' \\ & | \\ & \llbracket P \vdash ((g, \text{nid}, m, p) \# \text{xs}), h) \longrightarrow ((g', \text{nid}', m', p') \# \text{ys}), h'); \\ & \quad \text{has-return } m'; \\ & l' = (l \ @ \ [(g, \text{nid}, m, p)]) \\ & \implies \text{exec } P \ ((g, \text{nid}, m, p) \# \text{xs}), h) \ l \ ((g', \text{nid}', m', p') \# \text{ys}), h') \ l' \\ \text{code-pred } & (\text{modes: } i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow \text{bool as Exec}) \ \text{exec} . \end{aligned}$$

inductive *exec-debug* :: *Program*

$$\begin{aligned} & \Rightarrow (IRGraph \times ID \times MapState \times Params) \ \text{list} \times FieldRefHeap \\ & \Rightarrow \text{nat} \\ & \Rightarrow (IRGraph \times ID \times MapState \times Params) \ \text{list} \times FieldRefHeap \\ & \Rightarrow \text{bool} \\ & (-\vdash \longrightarrow * - * -) \\ \text{where} & \\ & \llbracket n > 0; \\ & \quad p \vdash s \longrightarrow s'; \\ & \quad \text{exec-debug } p \ s' \ (n - 1) \ s' \rceil \\ & \implies \text{exec-debug } p \ s \ n \ s'' \mid \\ & \llbracket n = 0 \rceil \\ & \implies \text{exec-debug } p \ s \ n \ s \\ \text{code-pred } & (\text{modes: } i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}) \ \text{exec-debug} . \end{aligned}$$

8.4.1 Heap Testing

definition *p3* :: *Params* **where**

$$p3 = [IntVal32 \ 3]$$

values $\{(prod.fst(prod.snd \ (prod.snd \ (hd \ (prod.fst \ res)))) \ 0$

$$\mid res. (\lambda x. \text{Some } eg2\text{-sq}) \vdash ((eg2\text{-sq}, 0, \text{new-map-state}, p3), (eg2\text{-sq}, 0, \text{new-map-state}, p3)),$$

$$\text{new-heap}) \rightarrow * 2 * \ res\}$$

definition *field-sq* :: *string* **where**

$$\text{field-sq} = "sq"$$

definition *eg3-sq* :: *IRGraph* **where**

$$\begin{aligned} eg3\text{-sq} = \text{irgraph } [\\ & (0, \text{StartNode } \text{None } 4, \text{VoidStamp}), \\ & (1, \text{ParameterNode } 0, \text{default-stamp}), \\ & (3, \text{MulNode } 1 \ 1, \text{default-stamp}), \\ & (4, \text{StoreFieldNode } 4 \ \text{field-sq } 3 \ \text{None } \text{None } 5, \text{VoidStamp}), \\ & (5, \text{ReturnNode } (\text{Some } 3) \ \text{None}, \text{default-stamp}) \end{aligned}$$

```

]

values {h-load-field field-sq None (prod.snd res)
        | res. (λx. Some eg3-sq) ⊢ [(eg3-sq, 0, new-map-state, p3), (eg3-sq, 0,
new-map-state, p3)], new-heap) →*3* res}

definition eg4-sq :: IRGraph where
  eg4-sq = irgraph [
    (0, StartNode None 4, VoidStamp),
    (1, ParameterNode 0, default-stamp),
    (3, MulNode 1 1, default-stamp),
    (4, NewInstanceNode 4 "obj-class" None 5, ObjectStamp "obj-class" True True
True),
    (5, StoreFieldNode 5 field-sq 3 None (Some 4) 6, VoidStamp),
    (6, ReturnNode (Some 3) None, default-stamp)
  ]

values {h-load-field field-sq (Some 0) (prod.snd res) | res.
        (λx. Some eg4-sq) ⊢ [(eg4-sq, 0, new-map-state, p3), (eg4-sq, 0,
new-map-state, p3)], new-heap) →*4* 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 ⊢ (nid,m,h) → next) ⇒
  (∀ next'. ((g, p ⊢ (nid,m,h) → next') ⟶ next = next'))
proof (induction rule: step.induct)
case (SequentialNode nid next m h)
have notif: ¬(is-IfNode (kind g nid))
using SequentialNode.hyps(1) is-sequential-node.simps
by (metis is-IfNode-def)
have notend: ¬(is-AbstractEndNode (kind g nid))
using SequentialNode.hyps(1) is-sequential-node.simps

```

```

    by (metis is-AbstractEndNode.simps is-EndNode.elims(2) is-LoopEndNode-def)
  have notnew:  $\neg(\text{is-NewInstanceNode } (\text{kind } g \text{ nid}))$ 
    using SequentialNode.hyps(1) is-sequential-node.simps
    by (metis is-NewInstanceNode-def)
  have notload:  $\neg(\text{is-LoadFieldNode } (\text{kind } g \text{ nid}))$ 
    using SequentialNode.hyps(1) is-sequential-node.simps
    by (metis is-LoadFieldNode-def)
  have notstore:  $\neg(\text{is-StoreFieldNode } (\text{kind } g \text{ nid}))$ 
    using SequentialNode.hyps(1) is-sequential-node.simps
    by (metis is-StoreFieldNode-def)
  have notdivrem:  $\neg(\text{is-IntegerDivRemNode } (\text{kind } g \text{ nid}))$ 
    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:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: IfNode.hyps(1))
have notend:  $\neg(\text{is-AbstractEndNode } (\text{kind } g \text{ nid}))$ 
  using is-AbstractEndNode.simps
  by (simp add: IfNode.hyps(1))
have notdivrem:  $\neg(\text{is-IntegerDivRemNode } (\text{kind } g \text{ nid}))$ 
  using is-AbstractEndNode.simps
  by (simp add: IfNode.hyps(1))
from notseq notend notdivrem show ?case using IfNode.repDet evalDet IRN-
ode.distinct IRNode.inject(11) Pair-inject step.simps
  by (smt (z3) IRNode.distinct IRNode.inject(12) Pair-inject step.simps)
next
case (EndNodes nid merge i phis inputs m vs m' h)
have notseq:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
  using EndNodes.hyps(1) is-AbstractEndNode.simps is-sequential-node.simps
  by (metis is-EndNode.elims(2) is-LoopEndNode-def)
have notif:  $\neg(\text{is-IfNode } (\text{kind } g \text{ nid}))$ 
  using EndNodes.hyps(1) is-IfNode-def is-AbstractEndNode.elims
  by (metis IRNode.distinct-disc(1058) is-EndNode.simps(12))
have notref:  $\neg(\text{is-RefNode } (\text{kind } g \text{ nid}))$ 
  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:  $\neg(\text{is-NewInstanceNode } (\text{kind } g \text{ nid}))$ 
  using EndNodes.hyps(1) is-AbstractEndNode.simps
  using IRNode.distinct-disc(1442) is-EndNode.simps(29) is-NewInstanceNode-def
  by (metis IRNode.distinct-disc(1901) is-EndNode.simps(32))

```

```

have notload: ¬(is-LoadFieldNode (kind g nid))
  using EndNodes.hyps(1) is-AbstractEndNode.simps
  using is-LoadFieldNode-def
  by (metis IRNode.distinct-disc(1706) is-EndNode.simps(21))
have notstore: ¬(is-StoreFieldNode (kind g nid))
  using EndNodes.hyps(1) is-AbstractEndNode.simps is-StoreFieldNode-def
  by (metis IRNode.distinct-disc(1926) is-EndNode.simps(44))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  using EndNodes.hyps(1) is-AbstractEndNode.simps is-SignedDivNode-def is-SignedRemNode-def
  using IRNode.distinct-disc(1498) IRNode.distinct-disc(1500) is-IntegerDivRemNode.simps
is-EndNode.simps(36) is-EndNode.simps(37)
  by auto
from notseq notif notref notnew notload notstore notdivrem
show ?case using EndNodes.repAllDet evalAllDet
  by (smt (z3) is-IfNode-def is-LoadFieldNode-def is-NewInstanceNode-def is-RefNode-def
is-StoreFieldNode-def is-SignedDivNode-def is-SignedRemNode-def Pair-inject is-IntegerDivRemNode.elims(3)
step.cases)
next
case (NewInstanceNode nid f obj nxt h' ref h m' m)
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notif: ¬(is-IfNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notref: ¬(is-RefNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notload: ¬(is-LoadFieldNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notstore: ¬(is-StoreFieldNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
from notseq notend notif notref notload notstore notdivrem
show ?case using NewInstanceNode.step.cases
  by (smt (z3) IRNode.disc(1028) IRNode.disc(2270) IRNode.discI(11) IRN-
ode.distinct(2311) IRNode.distinct(2313) IRNode.inject(31) Pair-inject)
next
case (LoadFieldNode nid f obj nxt m ref h v m')
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: LoadFieldNode.hyps(1))

```

```

have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: LoadFieldNode.hyps(1))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: LoadFieldNode.hyps(1))
from notseq notend notdivrem
show ?case using LoadFieldNode step.cases repDet evalDet
  by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject Value.inject(3)
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(3)
option.distinct(1) option.inject)
next
case (StaticStoreFieldNode nid f newval uv nxt m val h' h m')
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: StaticStoreFieldNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: StaticStoreFieldNode.hyps(1))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))

```

```

  by (simp add: StaticStoreFieldNode.hyps(1))
from notseq notend notdivrem
show ?case using StoreFieldNode step.cases repDet evalDet
  by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Static-
StoreFieldNode.hyps(1) StaticStoreFieldNode.hyps(2) StaticStoreFieldNode.hyps(3)
StaticStoreFieldNode.hyps(4) StaticStoreFieldNode.hyps(5) option.distinct(1))
next
case (SignedDivNode nid x y zero sb nxt m v1 v2 v m' h)
then have notseq:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: SignedDivNode.hyps(1))
have notend:  $\neg(\text{is-AbstractEndNode } (\text{kind } g \text{ nid}))$ 
  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:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: SignedRemNode.hyps(1))
have notend:  $\neg(\text{is-AbstractEndNode } (\text{kind } g \text{ nid}))$ 
  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*:

```

 $\llbracket \text{kind } g \text{ nid} = \text{RefNode nid}' \rrbracket \implies g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m, h)$ 
  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  $g \vdash \text{cond} \simeq \text{condE}$ 
assumes  $[m, p] \vdash \text{condE} \mapsto v$ 
assumes  $g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m, h)$ 
shows  $\text{nid}' \in \{\text{tb}, \text{fb}\}$ 
using step.IfNode repDet stepDet assms
  by (metis insert-iff old.prod.inject)

```

lemma *IfNodeSeq*:

```

shows kind g nid = IfNode cond tb fb  $\longrightarrow \neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 

```


unfolding *is-sequential-node.simps*
using *is-sequential-node.simps(18)* **by** *presburger*

lemma *IfNodeCond*:

assumes *kind g nid = IfNode cond tb fb*
assumes *g, p ⊢ (nid, m, h) → (nid', m, h)*
shows $\exists \text{ condE } v. ((g \vdash \text{cond} \simeq \text{condE}) \wedge ([m, p] \vdash \text{condE} \mapsto v))$
using *assms(2,1)* **by** (*induct (nid,m,h) (nid',m,h)* *rule: step.induct; auto*)

lemma *step-in-ids*:

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

end

9 Proof Infrastructure

9.1 Bisimulation

theory *Bisimulation*

imports

Stuttering

begin

inductive *weak-bisimilar* :: *ID ⇒ IRGraph ⇒ IRGraph ⇒ bool*

(*- . - ~ -*) **for** *nid* **where**

$\llbracket \forall P'. (g \text{ m } p \text{ h} \vdash \text{nid} \rightsquigarrow P') \longrightarrow (\exists Q'. (g' \text{ m } p \text{ h} \vdash \text{nid} \rightsquigarrow Q') \wedge P' = Q');$
 $\forall Q'. (g' \text{ m } p \text{ h} \vdash \text{nid} \rightsquigarrow Q') \longrightarrow (\exists P'. (g \text{ m } p \text{ h} \vdash \text{nid} \rightsquigarrow P') \wedge P' = Q') \rrbracket$
 $\implies \text{nid} . g \sim g'$

A strong bisimulation between no-op transitions

inductive *strong-noop-bisimilar* :: *ID ⇒ IRGraph ⇒ IRGraph ⇒ bool*

(*- | - ~ -*) **for** *nid* **where**

$\llbracket \forall P'. (g, p \vdash (\text{nid}, m, h) \rightarrow P') \longrightarrow (\exists Q'. (g', p \vdash (\text{nid}, m, h) \rightarrow Q') \wedge P' = Q');$
 $\forall Q'. (g', p \vdash (\text{nid}, m, h) \rightarrow Q') \longrightarrow (\exists P'. (g, p \vdash (\text{nid}, m, h) \rightarrow P') \wedge P' = Q') \rrbracket$
 $\implies \text{nid} \mid g \sim g'$

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

lemma *no-step-bisimulation*:
assumes $\forall m p h \textit{nid}' m' h'. \neg(g, p \vdash (\textit{nid}, m, h) \rightarrow (\textit{nid}', m', h'))$
assumes $\forall m p h \textit{nid}' m' h'. \neg(g', p \vdash (\textit{nid}, m, h) \rightarrow (\textit{nid}', m', h'))$
shows $\textit{nid} \mid g \sim g'$
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-usages $\textit{nid} \textit{nid}' g = \text{replace-node } \textit{nid} \text{ (RefNode } \textit{nid}', \text{ stamp } g \text{ } \textit{nid}') } g$

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

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

lemma *replace-usages-unchanged*:
assumes $\textit{nid} \in \textit{ids } g$
assumes $g' = \text{replace-usages } \textit{nid} \textit{nid}' g$
shows *unchanged* $(\textit{ids } g - \{\textit{nid}\}) g g'$
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  $\llbracket \text{finite } c; c \neq \{\} \rrbracket \implies (\text{Max } c) + 1 \notin c$ 
  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:  $\bigwedge c t f. \text{IfNode } c t f \neq \text{NoNode}$ 
    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 = (IntVal32 1)
    by auto
  from ifn have ifcond ≠ (nextNid g)
    by (metis assms(1) emptyE ids-some nextNidNotIn)
  moreover have  $\bigwedge c. \text{ConstantNode } c \neq \text{NoNode}$  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 (IntVal32 1)
    using truedef by simp
  have valid-value (IntVal32 1) (constantAsStamp (IntVal32 1))
    unfolding constantAsStamp.simps valid-value.simps
    using nat-numeral by blast

```

then have $[g', m, p] \vdash \text{nextNid } g \mapsto \text{IntVal32 } 1$
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] ⊢ nextNid g ↦ IntVal32 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 ⊢ (ifcond, m, h) → (f, 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*
by *(metis assms(1) assms(2) constantCondition.simps(1) replace-node-lookup)*
have *falsedef: bool-to-val False = (IntVal32 0)*
by *auto*
from *ifn have ifcond ≠ (nextNid g)*
by *(metis assms(1) equals0D ids-some nextNidNotIn)*
moreover have *∧ c. ConstantNode c ≠ 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 (IntVal32 0)*
using *falsedef* **by** *simp*
have *valid-value (IntVal32 0) (constantAsStamp (IntVal32 0))*
unfolding *constantAsStamp.simps valid-value.simps*
using *nat-numeral* **by** *blast*
then have $[g', m, p] \vdash \text{nextNid } g \mapsto \text{IntVal32 } 0$
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] ⊢ nextNid g ↦ IntVal32 0> encodeeval-def)*
qed

lemma *diff-forall:*

assumes $\forall n \in \text{ids } g - \{\text{nid}\}. \text{cond } n$
shows $\forall n. n \in \text{ids } g \wedge n \notin \{\text{nid}\} \longrightarrow \text{cond } n$
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  $g' = \text{add-node } \textit{nid} \textit{ node } g$ 
  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  $\neg(\textit{is-IfNode } (\textit{kind } g \textit{ nid}))$ 
  shows  $g = \textit{constantCondition } b \textit{ nid } (\textit{kind } g \textit{ nid}) \ g$ 
  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  $\textit{kind } g \textit{ nid} = \textit{IfNode } \textit{cond } t \ f$ 
  shows  $\textit{constantCondition } \textit{val } \textit{nid } (\textit{kind } g \textit{ nid}) \ g =$ 
     $\textit{replace-node } \textit{nid } (\textit{IfNode } (\textit{nextNid } g) \ t \ f, \textit{stamp } g \textit{ nid})$ 
     $(\textit{add-node } (\textit{nextNid } g) ((\textit{ConstantNode } (\textit{bool-to-val } \textit{val})), \textit{constantAsStamp}$ 
     $(\textit{bool-to-val } \textit{val}))) \ g)$ 
  using constantCondition.simps
  by (simp add: assms)

lemma constantCondition-changeonly:
  assumes  $\textit{nid} \in \textit{ids } g$ 
  assumes  $g' = \textit{constantCondition } b \textit{ nid } (\textit{kind } g \textit{ nid}) \ g$ 
  shows changeonly {nid}  $g \ g'$ 
proof (cases is-IfNode (kind g nid))
  case True
  have  $\textit{nextNid } g \notin \textit{ids } g$ 
  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  $\forall \text{ cond } t \text{ f. kind } g \text{ ifcond} \neq \text{IfNode cond } t \text{ f}$ 
  assumes  $g' = \text{constantCondition val ifcond (kind } g \text{ ifcond) } g$ 
  shows  $\exists \text{ nid' . } (g \text{ m } p \text{ h} \vdash \text{ifcond} \rightsquigarrow \text{nid'}) \longleftrightarrow (g' \text{ m } p \text{ h} \vdash \text{ifcond} \rightsquigarrow \text{nid'})$ 
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  $\text{kind } g \text{ ifcond} = \text{IfNode cond } t \text{ f}$ 
  assumes  $[g, m, p] \vdash \text{cond} \mapsto v$ 
  assumes  $\text{const} = \text{val-to-bool } v$ 
  assumes  $g' = \text{constantCondition const ifcond (kind } g \text{ ifcond) } g$ 
  shows  $\exists \text{ nid' . } (g \text{ m } p \text{ h} \vdash \text{ifcond} \rightsquigarrow \text{nid'}) \longleftrightarrow (g' \text{ m } p \text{ h} \vdash \text{ifcond} \rightsquigarrow \text{nid'})$ 
proof (cases const)
  case True
  have ifstep:  $g, p \vdash (\text{ifcond}, m, h) \rightarrow (t, m, h)$ 
  by (meson IfNode True assms(1) assms(2) assms(3) encodeeval-def)
  have ifstep':  $g', p \vdash (\text{ifcond}, m, h) \rightarrow (t, m, h)$ 
  using constantConditionTrue
  using True assms(1) assms(4) by presburger
  from ifstep ifstep' show ?thesis
  using StutterStep by blast
next
  case False
  have ifstep:  $g, p \vdash (\text{ifcond}, m, h) \rightarrow (f, m, h)$ 
  by (meson IfNode False assms(1) assms(2) assms(3) encodeeval-def)
  have ifstep':  $g', p \vdash (\text{ifcond}, m, h) \rightarrow (f, m, h)$ 
  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 (- - - - -  $\rightsquigarrow$  - 55)

```

for $g \ m \ p \ h$ **where**

StutterStep:

$\llbracket g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \rrbracket$
 $\implies g \ m \ p \ h \vdash nid \rightsquigarrow nid' \mid$

Transitive:

$\llbracket g, p \vdash (nid, m, h) \rightarrow (nid'', m, h);$
 $g \ m \ p \ h \vdash nid'' \rightsquigarrow nid' \rrbracket$
 $\implies g \ m \ p \ h \vdash nid \rightsquigarrow nid'$

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 *Semantics.IRTreeEvalThms*

begin

9.4.1 Support Lemmas for Stamps and Upper/Lower Bounds

lemma *size32*: *size* $v = 32$ **for** $v :: 32 \text{ word}$

using *size-word.rep-eq*

using *One-nat-def add.right-neutral add-Suc-right len-of-numeral-defs(2) len-of-numeral-defs(3)*
mult.right-neutral mult-Suc-right numeral-2-eq-2 numeral-Bit0

by (*smt (verit, del-insts) mult commute*)

lemma *size64*: *size* $v = 64$ **for** $v :: 64 \text{ word}$

using *size-word.rep-eq*

using *One-nat-def add.right-neutral add-Suc-right len-of-numeral-defs(2) len-of-numeral-defs(3)*
mult.right-neutral mult-Suc-right numeral-2-eq-2 numeral-Bit0

by (*smt (verit, del-insts) mult commute*)

declare $\llbracket \text{show-types} = \text{true} \rrbracket$

lemma *signed-int-bottom32*: $-(((2::int) \wedge 31)) \leq \text{sint } (v::int32)$
using *sint-range-size size32*
by (*smt* (*verit*, *ccfv-SIG*) *One-nat-def Suc-pred add-Suc add-Suc-right eval-nat-numeral(3)*)
nat.inject numeral-2-eq-2 numeral-Bit0 numeral-Bit1 zero-less-numeral

lemma *signed-int-top32*: $(2 \wedge 31) - 1 \geq \text{sint } (v::int32)$
using *sint-range-size size32*
by (*smt* (*verit*, *ccfv-SIG*) *One-nat-def Suc-pred add-Suc add-Suc-right eval-nat-numeral(3)*)
nat.inject numeral-2-eq-2 numeral-Bit0 numeral-Bit1 zero-less-numeral

lemma *lower-bounds-equiv32*: $-(((2::int) \wedge 31)) = (2::int) \wedge 32 \text{ div } 2 * - 1$
by *fastforce*

lemma *upper-bounds-equiv32*: $(2::int) \wedge 31 = (2::int) \wedge 32 \text{ div } 2$
by *simp*

lemma *bit-bounds-min32*: $((fst (bit-bounds 32))) \leq (\text{sint } (v::int32))$
unfolding *bit-bounds.simps fst-def* **using** *signed-int-bottom32 lower-bounds-equiv32*
by *auto*

lemma *bit-bounds-max32*: $((snd (bit-bounds 32))) \geq (\text{sint } (v::int32))$
unfolding *bit-bounds.simps fst-def* **using** *signed-int-top32 upper-bounds-equiv32*
by *auto*

lemma *signed-int-bottom64*: $-(((2::int) \wedge 63)) \leq \text{sint } (v::int64)$
using *sint-range-size size64*
by (*smt* (*verit*, *ccfv-SIG*) *One-nat-def Suc-pred add-Suc add-Suc-right eval-nat-numeral(3)*)
nat.inject numeral-2-eq-2 numeral-Bit0 numeral-Bit1 zero-less-numeral

lemma *signed-int-top64*: $(2 \wedge 63) - 1 \geq \text{sint } (v::int64)$
using *sint-range-size size64*
by (*smt* (*verit*, *ccfv-SIG*) *One-nat-def Suc-pred add-Suc add-Suc-right eval-nat-numeral(3)*)
nat.inject numeral-2-eq-2 numeral-Bit0 numeral-Bit1 zero-less-numeral

lemma *lower-bounds-equiv64*: $-(((2::int) \wedge 63)) = (2::int) \wedge 64 \text{ div } 2 * - 1$
by *fastforce*

lemma *upper-bounds-equiv64*: $(2::int) \wedge 63 = (2::int) \wedge 64 \text{ div } 2$
by *simp*

lemma *bit-bounds-min64*: $((fst (bit-bounds 64))) \leq (\text{sint } (v::int64))$
unfolding *bit-bounds.simps fst-def* **using** *signed-int-bottom64 lower-bounds-equiv64*
by *auto*

lemma *bit-bounds-max64*: $((snd (bit-bounds 64))) \geq (\text{sint } (v::int64))$
unfolding *bit-bounds.simps fst-def* **using** *signed-int-top64 upper-bounds-equiv64*
by *auto*

lemma *unrestricted-32bit-always-valid* [simp]:
valid-value (IntVal32 v) (unrestricted-stamp (IntegerStamp 32 lo hi))
using *valid-value.simps*(1) *bit-bounds-min32* *bit-bounds-max32*
using *unrestricted-stamp.simps*(2) **by** *presburger*

lemma *unrestricted-64bit-always-valid* [simp]:
valid-value (IntVal64 v) (unrestricted-stamp (IntegerStamp 64 lo hi))
using *valid-value.simps*(2) *bit-bounds-min64* *bit-bounds-max64*
using *unrestricted-stamp.simps*(2) **by** *presburger*

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

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

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

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

Next we show that casting a word to a wider word preserves any upper/lower bounds.

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

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

sint-lt)

lemma *scast-bigger-max-bound*:

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

lemma *scast-bigger-min-bound*:

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

lemma *scast-bigger-bit-bounds*:

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

lemma *unrestricted-stamp32-always-valid* [*simp*]:

assumes *fst* (*bit-bounds bits*) ≤ *sint ival* ∧ *sint ival* ≤ *snd* (*bit-bounds bits*)
assumes *bits* = 32 ∨ *bits* = 16 ∨ *bits* = 8 ∨ *bits* = 1
assumes *result* = *IntVal32 ival*
shows *valid-value result* (*unrestricted-stamp* (*IntegerStamp bits lo hi*))
using *assms valid-value.simps*(1) *unrestricted-stamp.simps*(2) **by** *presburger*

lemma *larger-stamp32-always-valid* [*simp*]:

assumes *valid-value result* (*unrestricted-stamp* (*IntegerStamp inBits lo hi*))
assumes *result* = *IntVal32 ival*
assumes *outBits* = 32 ∨ *outBits* = 16 ∨ *outBits* = 8 ∨ *outBits* = 1
assumes *inBits* ≤ *outBits*
shows *valid-value result* (*unrestricted-stamp* (*IntegerStamp outBits lo hi*))
using *assms by* (*smt* (*z3*) *bit-bounds.simps diff-le-mono linorder-not-less lower-bounds-equiv not-numeral-le-zero numerals*(1) *power-increasing-iff prod.sel*(1) *prod.sel*(2) *unrestricted-stamp.simps*(2) *valid-value.simps*(1))

Possibly helpful lemmas about *signed_ttake_{bit}*, to help with *UnaryNarrow*.

Note: we could use *signed* to convert between bit-widths, instead of *signed_take_bit*.

But this has to be done separately for each bit-width type.

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

ML-val <@{*thm signed-take-bit-decr-length-iff*}>

declare [[*show-types=true*]]

ML-val <@{*thm signed-take-bit-int-less-exp*}>

lemma *signed-take-bit-int-less-exp-word*:
assumes $n < \text{LENGTH}('a)$
shows $\text{sint}(\text{signed-take-bit } n \ (k :: 'a :: \text{len word})) < (2::\text{int}) \wedge n$
apply *transfer*
by (*smt* (*verit*, *best*) *not-take-bit-negative signed-take-bit-eq-take-bit-shift*
signed-take-bit-int-less-exp take-bit-int-greater-self-iff)

lemma *signed-take-bit-int-greater-eq-minus-exp-word*:
assumes $n < \text{LENGTH}('a)$
shows $-(2 \wedge n) \leq \text{sint}(\text{signed-take-bit } n \ (k :: 'a :: \text{len word}))$
apply *transfer*
by (*smt* (*verit*, *best*) *signed-take-bit-int-greater-eq-minus-exp*
signed-take-bit-int-greater-eq-self-iff signed-take-bit-int-less-exp)

Some important lemmas showing that `sign_extend_helper` produces integer results whose range is determined by the `inBits` parameter.

lemma *sign-extend-helper-output-range64*:
assumes $\text{result} = \text{sign-extend-helper } \text{inBits } \text{outBits } \text{val}$
assumes $\text{result} = \text{IntVal64 } \text{ival}$
shows $\text{outBits} = 64 \wedge -(2 \wedge (\text{inBits} - 1)) \leq \text{sint } \text{ival} \wedge \text{sint } \text{ival} \leq 2 \wedge (\text{inBits} - 1)$

proof –

have *ival*: $\text{ival} = (\text{scast } (\text{signed-take-bit } (\text{inBits} - 1) \ \text{val}))$
using *assms sign-extend-helper.simps*
by (*smt* (*verit*, *ccfv-SIG*) *Value.distinct(3) Value.inject(2) Value.simps(14)*)
then have *lo*: $-(2 \wedge (\text{inBits} - 1)) \leq \text{sint } (\text{signed-take-bit } (\text{inBits} - 1) \ \text{val})$
using *signed-take-bit-int-greater-eq-minus-exp-word*
by (*smt* (*verit*, *best*) *diff-le-self not-less power-increasing-iff sint-below-size*
wsst-TYs(3))
then have *lo2*: $-(2 \wedge (\text{inBits} - 1)) \leq \text{sint } (\text{scast } (\text{signed-take-bit } (\text{inBits} - 1) \ \text{val}))$
by (*smt* (*verit*, *best*) *diff-less len-gt-0 less-Suc-eq power-strict-increasing-iff*
signed-scast-eq signed-take-bit-int-greater-eq-self-iff signed-take-bit-int-less-exp-word
sint-range-size wsst-TYs(3))
have *hi*: $\text{sint } (\text{signed-take-bit } (\text{inBits} - 1) \ \text{val}) < 2 \wedge (\text{inBits} - 1)$
using *signed-take-bit-int-less-exp-word*
by (*metis* *diff-le-mono less-imp-diff-less linorder-not-le one-le-numeral power-increasing*
sint-above-size wsst-TYs(3))
then have *hi2*: $\text{sint } (\text{scast } (\text{signed-take-bit } (\text{inBits} - 1) \ \text{val})) < 2 \wedge (\text{inBits} - 1)$
by (*smt* (*verit*) *One-nat-def lo signed-scast-eq signed-take-bit-int-less-eq-self-iff*
sint-lt)
show *?thesis*
unfolding *bit-bounds.simps fst-def ival*
using *assms lo2 hi2 order-le-less*
by (*smt* (*verit*, *best*) *Value.simps(14) Value.simps(8) sign-extend-helper.simps*)
qed

lemma *sign-extend-helper-output-range32*:

```

assumes result = sign-extend-helper inBits outBits val
assumes result = IntVal32 ival
shows outBits ≤ 32 ∧ ¬(2 ^ (inBits - 1)) ≤ sint ival ∧ sint ival ≤ 2 ^ (inBits
- 1)

proof -
  have ival: ival = (signed-take-bit (inBits - 1) val)
    using assms sign-extend-helper.simps
    by (smt (verit, ccfv-SIG) Value.distinct(1) Value.inject(1) Value.simps(14)
scast-id)
  have def: result ≠ UndefinedVal
    using assms
    by blast
  then have ok: 0 < inBits ∧ inBits ≤ 32 ∧
    inBits ≤ outBits ∧
    outBits ∈ valid-int-widths ∧
    inBits ∈ valid-int-widths
    using assms sign-extend-helper-ok by blast
  then have lo: ¬(2 ^ (inBits - 1)) ≤ sint (signed-take-bit (inBits - 1) val)
    using signed-take-bit-int-greater-eq-minus-exp-word
    by (smt (verit, best) diff-le-self not-less power-increasing-iff sint-below-size
wsst-TYs(3))
  have hi: sint (signed-take-bit (inBits - 1) val) < 2 ^ (inBits - 1)
    using signed-take-bit-int-less-exp-word
    by (metis diff-le-mono less-imp-diff-less linorder-not-le one-le-numeral power-increasing
sint-above-size wsst-TYs(3))
  show ?thesis
    unfolding bit-bounds.simps fst-def ival
    using assms ival ok lo hi order-le-less
    by force
qed

```

9.4.2 Support Lemmas for integer input/output size of unary and binary operators

These help us to deduce integer sizes through expressions. Not used yet.

lemma unary-abs-io32:

```

assumes result = unary-eval UnaryAbs val
assumes result = IntVal32 r32
shows ∃ v32. val = IntVal32 v32
  by (smt (verit, best) Value.distinct(9) Value.simps(6) assms(1) assms(2) int-
val-abs.elims unary-eval.simps(1))

```

lemma unary-abs-io64:

```

assumes result = unary-eval UnaryAbs val
assumes result = IntVal64 r64
shows ∃ v64. val = IntVal64 v64
  by (metis Value.collapse(2) Value.collapse(3) Value.collapse(4) Value.disc(3)
Value.exhaust-disc Value.simps(8) assms(1) assms(2) intval-abs.simps(1) intval-abs.simps(5))

```

is-IntVal32-def unary-eval.simps(1) unary-obj unary-undef)

lemma *unary-neg-io32:*

assumes *result = unary-eval UnaryNeg val*
assumes *result = IntVal32 r32*
shows $\exists v32. val = IntVal32 v32$
by (*metis Value.disc(7) Value.distinct(1) assms(1) assms(2) intval-negate.elims is-IntVal64-def unary-eval.simps(2)*)

lemma *unary-neg-io64:*

assumes *result = unary-eval UnaryNeg val*
assumes *result = IntVal64 r64*
shows $\exists v64. val = IntVal64 v64$
by (*metis Value.disc(3) Value.simps(8) assms(1) assms(2) intval-negate.elims is-IntVal32-def unary-eval.simps(2)*)

9.4.3 Validity of UnaryAbs

A set of lemmas for each evaltree step. Questions: 1. do we need separate 32/64 lemmas? Yes, I think so, because almost every operator behaves differently on each width. And it makes the matching more direct, does not need *is_IntVal_def* etc. 2. is this top-down approach (assume the result node evaluation) best? Maybe. It seems to be the shortest/simplest trigger?

lemma *unary-abs-result64:*

assumes $[m,p] \vdash (UnaryExpr UnaryAbs e) \mapsto IntVal64 v$
obtains *ve where* ($[m, p] \vdash e \mapsto IntVal64 ve$) \wedge
 $v = (if\ ve < s\ 0\ then\ -ve\ else\ ve)$
proof –
obtain *ve where* $[m,p] \vdash e \mapsto IntVal64 ve$
by (*smt (verit, best) assms UnaryExprE Value.distinct evalDet intval-abs.elims unary-eval.simps(1)*)
then show *?thesis*
by (*metis UnaryExprE Value.sel(2) assms evalDet intval-abs.simps(2) that unary-eval.simps(1)*)
qed

lemma *unary-abs-result32:*

assumes $1:[m,p] \vdash (UnaryExpr UnaryAbs e) \mapsto IntVal32 v$
shows $\exists ve. ([m, p] \vdash e \mapsto IntVal32 ve) \wedge$
 $v = (if\ ve < s\ 0\ then\ -ve\ else\ ve)$
proof –
obtain *ve where* $[m,p] \vdash e \mapsto IntVal32 ve$
by (*smt (verit, best) 1 UnaryExprE Value.distinct evalDet intval-abs.elims unary-eval.simps(1)*)
then show *?thesis*
by (*metis UnaryExprE Value.inject(1) assms evalDet intval-abs.simps(1) unary-eval.simps(1)*)
qed

```

lemma unary-abs-implies-valid-value:
  assumes 1:[m,p] ⊢ expr ↦ val
  assumes 2:result = unary-eval UnaryAbs val
  assumes 3:result ≠ UndefVal
  assumes 4:valid-value val (stamp-expr expr)
  shows valid-value result (stamp-expr (UnaryExpr UnaryAbs expr))
proof -
  have 5:[m,p] ⊢ (UnaryExpr UnaryAbs expr) ↦ result
    using assms by blast
  then have 6: is-IntegerStamp (stamp-expr expr)
    using assms valid-value.elims(2) by fastforce
  then consider v32 where result = IntVal32 v32 | v64 where result = IntVal64
v64
  by (metis 2 4 Stamp.collapse(1) intval-abs.simps(1) intval-abs.simps(2) unary-eval.simps(1)
valid32or64)
  then show ?thesis
proof cases
  case 1
  then obtain ve where ve: ([m, p] ⊢ expr ↦ IntVal32 ve) ∧
    result = (if ve <ₛ 0 then IntVal32 (-ve) else IntVal32 ve)
    using 5 unary-abs-result32 by metis
  then have 32: val = IntVal32 ve
    using assms(1) evalDet by presburger

  then obtain b lo hi where se: stamp-expr expr = IntegerStamp b lo hi
    using 6 is-IntegerStamp-def by auto
  then have ((b=32 ∨ b=16 ∨ b=8 ∨ b=1) ∧ (lo ≤ sint ve) ∧ (sint ve ≤ hi))
    using 4 32 se by simp
  then have stamp-expr (UnaryExpr UnaryAbs expr) = unrestricted-stamp
(IntegerStamp 32 lo hi)
    using se by fastforce
  then show ?thesis
    using 1 unrestricted-32bit-always-valid by presburger
next
  case 2
  then obtain ve where ve: ([m, p] ⊢ expr ↦ IntVal64 ve) ∧
    result = (if ve <ₛ 0 then IntVal64 (-ve) else IntVal64 ve)
    using 5 unary-abs-result64 by metis
  then have 64: val = IntVal64 ve
    using assms(1) evalDet by presburger

  then obtain b lo hi where se: stamp-expr expr = IntegerStamp b lo hi
    using 6 is-IntegerStamp-def by auto
  then have range64: b=64 ∧ (lo ≤ sint ve) ∧ (sint ve ≤ hi)
    using 4 64 se by simp
  then have stamp-expr (UnaryExpr UnaryAbs expr) = unrestricted-stamp
(IntegerStamp b lo hi)

```

```

    using se by simp
  then show ?thesis
    by (metis 2 range64 unrestricted-64bit-always-valid)
qed
qed

```

9.4.4 Validity of UnaryNeg

```

lemma unary-neg-implies-valid-value:
  assumes 1:[m,p] ⊢ expr ↦ val
  assumes 2:result = unary-eval UnaryNeg val
  assumes 3:result ≠ UndefVal
  assumes 4:valid-value val (stamp-expr expr)
  shows valid-value result (stamp-expr (UnaryExpr UnaryNeg expr))
proof -
  have 6: result = intval-negate val
    using assms by auto
  then have 7: is-IntegerStamp (stamp-expr expr)
    using assms valid-value.elims(2) by fastforce
  then obtain b lo hi where se: stamp-expr expr = IntegerStamp b lo hi
    using 7 assms valid-value.elims(2) is-IntegerStamp-def by auto
  then have stamp-expr (UnaryExpr UnaryNeg expr) = unrestricted-stamp (IntegerStamp
    (if b=64 then 64 else 32) lo hi)
    using assms by auto
  then show ?thesis
    using assms 6 se
    by (smt (verit, best) intval-negate.simps(1) intval-negate.simps(2) unrestricted-32bit-always-valid
    unrestricted-64bit-always-valid valid32or64 valid-int64 valid-value.simps(2))
qed

```

9.4.5 Validity of UnaryNot

```

lemma unary-not-implies-valid-value:
  assumes 1:[m,p] ⊢ expr ↦ val
  assumes 2:result = unary-eval UnaryNot val
  assumes 3:result ≠ UndefVal
  assumes 4:valid-value val (stamp-expr expr)
  shows valid-value result (stamp-expr (UnaryExpr UnaryNot expr))
proof -
  have 6: result = intval-not val
    using assms by auto
  then have 7: is-IntegerStamp (stamp-expr expr)
    using assms valid-value.elims(2) by fastforce
  then obtain b lo hi where se: stamp-expr expr = IntegerStamp b lo hi
    using 7 assms valid-value.elims(2) is-IntegerStamp-def by auto
  then have stamp-expr (UnaryExpr UnaryNot expr) = unrestricted-stamp (IntegerStamp
    (if b=64 then 64 else 32) lo hi)
    using assms by auto
  then show ?thesis
    using assms 6 se

```

```

    by (smt (verit, best) intval-not.simps(1) intval-not.simps(2) unrestricted-32bit-always-valid
unrestricted-64bit-always-valid valid32or64 valid-int64 valid-value.simps(2))
qed

```

9.4.6 Validity of UnaryLogicNegation

```

lemma unary-logic-negation-implies-valid-value:
  assumes 1:[m,p] ⊢ expr ↦ val
  assumes 2:result = unary-eval UnaryLogicNegation val
  assumes 3:result ≠ UndefVal
  assumes 4:valid-value val (stamp-expr expr)
  shows valid-value result (stamp-expr (UnaryExpr UnaryLogicNegation expr))
proof -
  have 6: result = intval-logic-negation val
    using assms by auto
  then have 7: is-IntegerStamp (stamp-expr expr)
    using assms valid-value.elims(2) by fastforce
  then obtain b lo hi where se: stamp-expr expr = IntegerStamp b lo hi
    using 7 assms valid-value.elims(2) is-IntegerStamp-def by auto
  then have stamp-expr (UnaryExpr UnaryLogicNegation expr) = unrestricted-stamp
(IntegerStamp (if b=64 then 64 else 32) lo hi)
    using assms by auto
  then show ?thesis
    using assms 6 se
    by (smt (verit, best) intval-logic-negation.simps(1) intval-logic-negation.simps(2)
unrestricted-32bit-always-valid unrestricted-64bit-always-valid valid32or64 valid-int64
valid-value.simps(2))
qed

```

9.4.7 Validity of UnaryNarrow

Possibly helpful lemmas about mod - mostly not used now.

```

lemma uint-distr-mod:
  fixes n :: nat
  assumes n < LENGTH('a)
  shows uint ((uval :: 'a :: len word) mod 2^n) = uint uval mod 2^n
  by (metis take-bit-eq-mod unsigned-take-bit-eq)

```

```

lemma sint-mod-not-sign-bit:
  fixes n :: nat
  assumes n < LENGTH('a)
  shows ¬ bit ((uval :: 'a :: len word) mod 2^n) LENGTH('a)
  by simp

```

```

lemma sint-mod-upper-bound:
  fixes n :: nat
  assumes n < LENGTH('a)
  shows sint ((uval :: 'a :: len word) mod 2^n) < 2^n
  by (metis assms(1) signed-take-bit-eq take-bit-eq-mod take-bit-int-less-exp)

```



```

lemma sint-mod-lower-bound:
  fixes  $n :: \text{nat}$ 
  assumes  $n < \text{LENGTH}('a)$ 
  shows  $0 \leq \text{sint } ((\text{uval} :: 'a :: \text{len word}) \bmod 2^n)$ 
  unfolding sint-uint
  by (metis assms signed-take-bit-eq sint-uint take-bit-eq-mod take-bit-nonnegative)

lemma sint-mod-range:
  fixes  $n :: \text{nat}$ 
  assumes  $n < \text{LENGTH}('a)$ 
  assumes smaller =  $((\text{val} :: 'a :: \text{len word}) \bmod 2^n)$ 
  shows  $0 \leq \text{sint } \text{smaller} \wedge \text{sint } \text{smaller} < 2^n$ 
  using assms sint-mod-upper-bound sint-mod-lower-bound
  using le-less by blast

lemma sint-mod-eq-uint:
  fixes  $n :: \text{nat}$ 
  assumes  $n < \text{LENGTH}('a)$ 
  shows  $\text{sint } ((\text{uval} :: 'a :: \text{len word}) \bmod 2^n) = \text{uint } (\text{uval} \bmod 2^n)$ 
  unfolding sint-uint
  by (metis Suc-pred assms le-less len-gt-0 signed-take-bit-eq sint-uint take-bit-eq-mod
    take-bit-signed-take-bit unsigned-take-bit-eq)

lemma unary-narrow-helper32:
  assumes  $[m, p] \vdash \text{expr} \mapsto \text{IntVal32 } i32$ 
  assumes stamp-expr  $\text{expr} = \text{IntegerStamp } b \text{ lo } hi$ 
  assumes  $r32 = \text{signed-take-bit } (\text{outBits} - 1) \ i32$ 
  assumes  $\text{result} = \text{IntVal32 } r32$ 
  assumes  $\text{outBits} = 32 \vee \text{outBits} = 16 \vee \text{outBits} = 8 \vee \text{outBits} = 1$ 
  assumes stamp-expr  $(\text{UnaryExpr } (\text{UnaryNarrow } \text{inBits } \text{outBits}) \text{ expr})$ 
    = unrestricted-stamp  $(\text{IntegerStamp } \text{outBits } \text{lo } hi)$ 
  shows valid-value  $\text{result } (\text{stamp-expr } (\text{UnaryExpr } (\text{UnaryNarrow } \text{inBits } \text{outBits})$ 
     $\text{expr}))$ 
proof –
  have hi:  $\text{sint } r32 < 2^{(\text{outBits}-1)}$ 
    using assms signed-take-bit-int-less-exp-word
    by (metis diff-le-mono less-imp-diff-less linorder-not-le one-le-numeral
      power-increasing sint-above-size word-size)
  then have lo:  $-(2^{(\text{outBits}-1)}) \leq \text{sint } r32$ 
    using assms signed-take-bit-int-greater-eq-minus-exp-word
    by (smt (verit, best) diff-le-self less-le-trans power-less-imp-less-exp sint-ge)
  then show ?thesis
    using assms lo hi apply simp
    by (metis int-power-div-base lessI zero-less-numeral)
qed

```

```

lemma unary-narrow-helper64:
  assumes  $[m,p] \vdash \text{expr} \mapsto \text{IntVal64 } i64$ 
  assumes  $\text{stamp-expr expr} = \text{IntegerStamp } b \text{ lo } hi$ 
  assumes  $r32 = \text{signed-take-bit } (\text{outBits} - 1) (\text{scast } i64)$ 
  assumes  $\text{result} = \text{IntVal32 } r32$ 
  assumes  $\text{outBits}=32 \vee \text{outBits}=16 \vee \text{outBits}=8 \vee \text{outBits}=1$ 
  assumes  $\text{stamp-expr } (\text{UnaryExpr } (\text{UnaryNarrow } \text{inBits } \text{outBits}) \text{ expr})$ 
     $= \text{unrestricted-stamp } (\text{IntegerStamp } \text{outBits } \text{lo } \text{hi})$ 
  shows  $\text{valid-value result } (\text{stamp-expr } (\text{UnaryExpr } (\text{UnaryNarrow } \text{inBits } \text{outBits})$ 
     $\text{expr}))$ 
proof –
  have  $hi: \text{sint } r32 < 2^{(\text{outBits}-1)}$ 
    using  $\text{assms signed-take-bit-int-less-exp-word}$ 
    by ( $\text{metis diff-le-mono less-imp-diff-less linorder-not-le one-le-numeral}$ 
       $\text{power-increasing sint-above-size word-size}$ )
  then have  $lo: -(2^{(\text{outBits}-1)}) \leq \text{sint } r32$ 
    using  $\text{assms signed-take-bit-int-greater-eq-minus-exp-word}$ 
    by ( $\text{smt (verit, best) diff-le-self less-le-trans power-less-imp-less-exp sint-ge}$ )
  then show  $?thesis$ 
    using  $\text{assms lo hi apply simp}$ 
    by ( $\text{metis int-power-div-base lessI zero-less-numeral}$ )
qed

```

```

lemma unary-narrow-implies-valid-value:
  assumes  $[m,p] \vdash \text{expr} \mapsto \text{val}$ 
  assumes  $\text{result} = \text{unary-eval } (\text{UnaryNarrow } \text{inBits } \text{outBits}) \text{ val}$ 
  assumes  $\text{result} \neq \text{UndefVal}$ 
  assumes  $\text{valid-value val } (\text{stamp-expr expr})$ 
  shows  $\text{valid-value result } (\text{stamp-expr } (\text{UnaryExpr } (\text{UnaryNarrow } \text{inBits } \text{outBits})$ 
     $\text{expr}))$ 
proof –

  have  $i: \text{is-IntegerStamp } (\text{stamp-expr expr})$ 
    using  $\text{assms valid-value.elims}(2)$  by  $\text{fastforce}$ 
  then obtain  $b \text{ lo } hi$  where  $se: \text{stamp-expr expr} = \text{IntegerStamp } b \text{ lo } hi$ 
    by ( $\text{auto simp add: assms valid-value.elims}(2) \text{is-IntegerStamp-def}$ )
  then have  $u: \text{stamp-expr } (\text{UnaryExpr } (\text{UnaryNarrow } \text{inBits } \text{outBits}) \text{ expr})$ 
     $= \text{unrestricted-stamp } (\text{IntegerStamp } \text{outBits } \text{lo } \text{hi})$ 
    by  $\text{simp}$ 

  have  $r: \text{result} = \text{intval-narrow } \text{inBits } \text{outBits } \text{val}$ 
    by ( $\text{simp add: assms}(2)$ )
  then have  $ok: 0 < \text{outBits} \wedge \text{outBits} \leq \text{inBits} \wedge$ 
     $\text{outBits} \in \text{valid-int-widths} \wedge \text{inBits} \in \text{valid-int-widths}$ 
    using  $\text{assms intval-narrow-ok}$  by  $\text{simp}$ 
  then consider  $i32$  where  $\text{val} = \text{IntVal32 } i32 \mid i64$  where  $\text{val} = \text{IntVal64 } i64$ 
    using  $\text{assms}$  by ( $\text{metis se valid32or64}$ )
  then show  $?thesis$ 

```

```

proof cases
  case 1
    then have r1: result = narrow-helper inBits outBits i32
      using assms r by (metis intval-narrow.simps(1))
    then have r2: result = (IntVal32 (signed-take-bit (outBits - 1) i32))
      using assms by (metis narrow-helper.simps)
    then obtain r32 where
      r32: result = IntVal32 r32  $\wedge$  r32 = signed-take-bit (outBits - 1) i32
      by simp
    then have outBits=32  $\vee$  outBits=16  $\vee$  outBits=8  $\vee$  outBits=1
      using ok 1 assms by force
    then show ?thesis
      using ok 1 assms u r32 se unary-narrow-helper32 by force
  next
    case 2
      then have in64: inBits = 64
        using assms ok intval-narrow.simps(2) r by presburger
      then show ?thesis
        proof (cases outBits = 64)
          case True
            then show ?thesis
              using 2 in64 r u intval-narrow.simps(2) unrestricted-64bit-always-valid by
presburger
          next
            case False

            then have out32: outBits=32  $\vee$  outBits=16  $\vee$  outBits=8  $\vee$  outBits=1
              using ok assms by force
            then have r1: result = narrow-helper inBits outBits (scast i64)
              using assms 2 False in64 r ok narrow-takes-64 by simp
            then have r2: result = (IntVal32 (signed-take-bit (outBits - 1) (scast i64)))
              using assms by (metis narrow-helper.simps)
            then obtain r32 where
              r32: result = IntVal32 r32  $\wedge$  r32 = signed-take-bit (outBits - 1) (scast
i64)
              by simp
            then show ?thesis
              using assms 2 r32 out32 u se unary-narrow-helper64 by blast
          qed
        qed
      qed

```

9.4.8 Validity of UnarySignExtend

lemma *valid-sign-extend32-or-less*:

```

assumes (result :: int32) = scast (v :: 'a :: len word)
assumes LENGTH('a) = 32  $\vee$  LENGTH('a) = 16  $\vee$  LENGTH('a) = 8  $\vee$ 
LENGTH('a) = 1
shows valid-value (IntVal32 result) (IntegerStamp LENGTH('a))

```

```

      (fst (bit-bounds (LENGTH('a))))
      (snd (bit-bounds (LENGTH('a'))))
unfolding valid-value.simps
using scast-bigger-bit-bounds assms by blast

lemma valid-sign-extend64:
  assumes (result :: int64) = scast (v :: 'a :: len word)
  shows valid-value (IntVal64 result) (IntegerStamp 64
    (fst (bit-bounds (LENGTH('a'))))
    (snd (bit-bounds (LENGTH('a')))))
  unfolding valid-value.simps
  using scast-bigger-bit-bounds
  using assms(1) len-gt-0 by blast

lemma unary-sign-extend-implies-valid-value:
  assumes [m,p] ⊢ expr ↦ val
  assumes result = unary-eval (UnarySignExtend inBits outBits) val
  assumes result ≠ UndefVal
  assumes valid-value val (stamp-expr expr)
  shows valid-value result (stamp-expr (UnaryExpr (UnarySignExtend inBits out-
    Bits) expr))
proof –

  have i: is-IntegerStamp (stamp-expr expr)
    using assms valid-value.elims(2) by fastforce
  then obtain b lo hi where se:stamp-expr expr = IntegerStamp b lo hi
    by (auto simp add: assms valid-value.elims(2) is-IntegerStamp-def)
  then have u: stamp-expr (UnaryExpr (UnarySignExtend inBits outBits) expr)
    = unrestricted-stamp (IntegerStamp outBits lo hi)
    by simp
  then show ?thesis
proof (cases is-IntVal64 val)
  case True
    then show ?thesis
      using assms u unrestricted-64bit-always-valid
      using is-IntVal64-def by fastforce
  next
  case False
    then obtain i32 where i32: result = sign-extend-helper inBits outBits i32
      using assms intval-sign-extend.simps
      by (metis is-IntVal64-def se unary-eval.simps(6) valid32or64)
    then have ok: 0 < inBits ∧ inBits ≤ 32 ∧ inBits ≤ outBits ∧
      outBits ∈ valid-int-widths ∧ inBits ∈ valid-int-widths
      using assms sign-extend-helper-ok by blast
    then show ?thesis
proof (cases outBits = 64)
  case True

```

```

    then obtain r64 where result = IntVal64 r64
    by (metis assms(3) i32 sign-extend-helper.simps)
  then show ?thesis
    using True u unrestricted-64bit-always-valid by presburger
next
case False
then obtain r32 where r32: result = IntVal32 r32
  using ok i32 by force
then have lohi:  $-(2^{inBits-1}) \leq \text{sint } r32 \wedge \text{sint } r32 < 2^{inBits-1}$ 
  using sign-extend-helper-output-range32
  by (smt (verit, ccfv-threshold) False Value.inject(1) assms(3) diff-le-self i32
linorder-not-le power-less-imp-less-exp sign-extend-helper.simps signed-take-bit-int-less-exp-word
sint-lt)
  then have bnds:  $\text{fst } (\text{bit-bounds } inBits) \leq \text{sint } r32 \wedge \text{sint } r32 \leq \text{snd } (\text{bit-bounds } inBits)$ 
    unfolding bit-bounds.simps fst-def
    using ok lower-bounds-equiv upper-bounds-equiv by simp
  then have v: valid-value result (unrestricted-stamp (IntegerStamp inBits lo
hi))
    using ok r32 by force
  then have outBits=1  $\vee$  outBits=8  $\vee$  outBits=16  $\vee$  outBits=32
    using ok False by fastforce
  then show ?thesis
    unfolding u using ok v r32 larger-stamp32-always-valid by presburger
qed
qed
qed

```

9.4.9 Validity of all Unary Operators

lemma unary-eval-implies-valid-value:

```

assumes  $[m, p] \vdash \text{expr} \mapsto \text{val}$ 
assumes result = unary-eval op val
assumes result  $\neq$  UndefVal
assumes valid-value val (stamp-expr expr)
shows valid-value result (stamp-expr (UnaryExpr op expr))

```

proof (cases op)

case UnaryAbs

then show ?thesis using assms unary-abs-implies-valid-value by presburger

next

case UnaryNeg

then show ?thesis using assms unary-neg-implies-valid-value by presburger

next

case UnaryNot

then show ?thesis using assms unary-not-implies-valid-value by presburger

next

case UnaryLogicNegation

then show ?thesis using assms unary-logic-negation-implies-valid-value by presburger

```

next
  case (UnaryNarrow x51 x52)
  then show ?thesis using assms unary-narrow-implies-valid-value by presburger
next
  case (UnarySignExtend x61 x62)
  then show ?thesis using assms unary-sign-extend-implies-valid-value by presburger
next
  case (UnaryZeroExtend x71 x72)
  then show ?thesis sorry
qed

```

9.4.10 Support Lemmas for Binary Operators

lemma *binary-undef*: $v1 = \text{UndefVal} \vee v2 = \text{UndefVal} \implies \text{bin-eval op } v1 \ v2 = \text{UndefVal}$
by (cases op; auto)

lemma *binary-obj*: $v1 = \text{ObjRef } x \vee v2 = \text{ObjRef } y \implies \text{bin-eval op } v1 \ v2 = \text{UndefVal}$
by (cases op; auto)

lemma *binary-eval-implies-valid-value*:
assumes $[m, p] \vdash \text{expr1} \mapsto \text{val1}$
assumes $[m, p] \vdash \text{expr2} \mapsto \text{val2}$
assumes $\text{result} = \text{bin-eval op } \text{val1 } \text{val2}$
assumes $\text{result} \neq \text{UndefVal}$
assumes *valid-value* val1 (stamp-expr expr1)
assumes *valid-value* val2 (stamp-expr expr2)
shows *valid-value* result (stamp-expr (BinaryExpr op expr1 expr2))
proof –
have *is-IntVal*: $\exists x \ y. \text{result} = \text{IntVal32 } x \vee \text{result} = \text{IntVal64 } y$
using assms(1,2,3,4) **apply** (cases op; auto; cases val1; auto; cases val2; auto)
by (meson Values.bool-to-val.elims)+
then have *expr1-intstamp*: *is-IntegerStamp* (stamp-expr expr1)
using assms(1,3,4,5) **apply** (cases (stamp-expr expr1); auto simp: valid-VoidStamp binary-undef)
using *valid-ObjStamp* *binary-obj* **apply** (metis assms(4))
using *valid-ObjStamp* *binary-obj* **by** (metis assms(4))
from *is-IntVal* **have** *expr2-intstamp*: *is-IntegerStamp* (stamp-expr expr2)
using assms(2,3,4,6) **apply** (cases (stamp-expr expr2); auto simp: valid-VoidStamp binary-undef)
using *valid-ObjStamp* *binary-obj* **apply** (metis assms(4))
using *valid-ObjStamp* *binary-obj* **by** (metis assms(4))
from *expr1-intstamp* **obtain** *b1 lo1 hi1* **where** *stamp-expr1-def*: stamp-expr expr1

```

= (IntegerStamp b1 lo1 hi1)
  using is-IntegerStamp-def by auto
  from expr2-intstamp obtain b2 lo2 hi2 where stamp-expr2-def: stamp-expr
expr2 = (IntegerStamp b2 lo2 hi2)
  using is-IntegerStamp-def by auto

```

```

have b1 = b2
  using assms(3,4,5,6) stamp-expr1-def stamp-expr2-def
  sorry
then have stamp-def: stamp-expr (BinaryExpr op expr1 expr2) =
  (if op ∉ fixed-32 ∧ b1=64
   then unrestricted-stamp (IntegerStamp 64 lo1 hi1)
   else unrestricted-stamp (IntegerStamp 32 lo1 hi1))
  using stamp-expr.simps(2) stamp-binary.simps(1)
  using stamp-expr1-def stamp-expr2-def by presburger
show ?thesis
proof (cases op ∉ fixed-32 ∧ b1=64)
  case True
  then obtain x where bit64: result = IntVal64 x
    using stamp-expr1-def assms by (cases op; cases val1; cases val2; simp)
  then show ?thesis
    by (metis True stamp-def unrestricted-64bit-always-valid)
next
  case False
  then obtain x where bit32: result = IntVal32 x
    using assms stamp-expr1-def apply (cases op; cases val1; cases val2; auto)
    by (meson Values.bool-to-val.elims)+
  then show ?thesis
    using False stamp-def unrestricted-32bit-always-valid by presburger
qed
qed

```

9.4.11 Validity of Stamp Meet and Join Operators

lemma stamp-meet-is-valid:

```

  assumes valid-value val stamp1 ∨ valid-value val stamp2
  assumes meet stamp1 stamp2 ≠ IllegalStamp
  shows valid-value val (meet stamp1 stamp2)
  using assms
proof (cases stamp1)
  case VoidStamp
  then show ?thesis
    by (metis Stamp.exhaust assms(1) assms(2) meet.simps(1) meet.simps(37)
meet.simps(44) meet.simps(51) meet.simps(58) meet.simps(65) meet.simps(66) meet.simps(67))
  next
  case (IntegerStamp b lo hi)

```

```

obtain b2 lo2 hi2 where stamp2-def: stamp2 = IntegerStamp b2 lo2 hi2
by (metis IntegerStamp assms(2) meet.simps(45) meet.simps(52) meet.simps(59)
meet.simps(6) meet.simps(65) meet.simps(66) meet.simps(67) unrestricted-stamp.cases)
then have b = b2 using meet.simps(2) assms(2)
by (metis IntegerStamp)
then have meet-def: meet stamp1 stamp2 = (IntegerStamp b (min lo lo2) (max
hi hi2))
by (simp add: IntegerStamp stamp2-def)
then show ?thesis proof (cases b = 64)
  case True
    then obtain x where val-def: val = IntVal64 x
    using IntegerStamp assms(1) valid64
    using ⟨b = b2⟩ stamp2-def by blast
    have min: sint x ≥ min lo lo2
    using val-def
    using IntegerStamp assms(1)
    using stamp2-def by force
    have max: sint x ≤ max hi hi2
    using val-def
    using IntegerStamp assms(1)
    using stamp2-def by force
    from min max show ?thesis
    by (simp add: True meet-def val-def)
  next
    case False
    then have bit32: b = 32 ∨ b = 16 ∨ b = 8 ∨ b = 1
    using assms(1) IntegerStamp valid-value.simps valid32or64-both
    by (metis ⟨b = b2⟩ stamp2-def)
    then obtain x where val-def: val = IntVal32 x
    using IntegerStamp assms(1) valid32 valid-int16 valid-int8 valid-int1
    using ⟨b = b2⟩ stamp2-def by blast
    have min: sint x ≥ min lo lo2
    using val-def
    using IntegerStamp assms(1)
    using stamp2-def by force
    have max: sint x ≤ max hi hi2
    using val-def
    using IntegerStamp assms(1)
    using stamp2-def by force
    from min max show ?thesis
    using bit32 meet-def val-def valid-value.simps(1) by presburger
  qed
next
  case (KlassPointerStamp x31 x32)
  then show ?thesis using assms valid-value.elims(2)
  by fastforce
next
  case (MethodCountersPointerStamp x41 x42)
  then show ?thesis using assms valid-value.elims(2)

```



```

    by fastforce
  next
    case (MethodPointersStamp x51 x52)
    then show ?thesis using assms valid-value.elims(2)
    by fastforce
  next
    case (ObjectStamp x61 x62 x63 x64)
    then show ?thesis using assms
    using meet.simps(34) by blast
  next
    case (RawPointerStamp x71 x72)
    then show ?thesis using assms
    using meet.simps(35) by blast
  next
    case IllegalStamp
    then show ?thesis using assms
    using meet.simps(36) by blast
qed

```

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 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 show ?thesis using stamp-meet-is-valid using stamp-expr.simps(6)
    using assms(2) assms(6) by presburger
qed

```

9.4.12 Validity of Whole Expression Tree Evaluation

experiment begin

lemma *stamp-implies-valid-value:*

```

  assumes [m,p] ⊢ expr ↦ val
  shows valid-value val (stamp-expr expr)
  using assms proof (induction expr val)
  case (UnaryExpr expr val result op)
  then show ?case using unary-eval-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)
    then show ?case using conditional-eval-implies-valid-value sorry
  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  $v \in \{val \mid \text{valid-value } val \text{ (stamp-expr } e)\}$ 
  using assms sorry
end

```

```

lemma upper-bound-32:
  assumes  $val = \text{IntVal32 } v$ 
  assumes  $\exists l h. s = (\text{IntegerStamp } 32 \ l \ h)$ 
  shows  $\text{valid-value } val \ s \implies \text{sint } v \leq (\text{stpi-upper } s)$ 
  using assms by force

```

```

lemma upper-bound-64:
  assumes  $val = \text{IntVal64 } v$ 
  assumes  $\exists l h. s = (\text{IntegerStamp } 64 \ l \ h)$ 
  shows  $\text{valid-value } val \ s \implies \text{sint } v \leq (\text{stpi-upper } s)$ 
  using assms by force

```

```

lemma lower-bound-32:
  assumes  $val = \text{IntVal32 } v$ 
  assumes  $\exists l h. s = (\text{IntegerStamp } 32 \ l \ h)$ 
  shows  $\text{valid-value } val \ s \implies \text{sint } v \geq (\text{stpi-lower } s)$ 
  using assms by force

```

```

lemma lower-bound-64:
  assumes  $val = \text{IntVal64 } v$ 
  assumes  $\exists l h. s = (\text{IntegerStamp } 64 \ l \ h)$ 
  shows  $\text{valid-value } val \ s \implies \text{sint } v \geq (\text{stpi-lower } s)$ 
  using assms
  by force

```

```

lemma stamp-under-semantics:
  assumes stamp-under (stamp-expr x) (stamp-expr y)

```

```

assumes  $[m, p] \vdash (\text{BinaryExpr BinIntegerLessThan } x \ y) \mapsto v$ 
assumes  $xval$ :  $(\forall m \ p \ v. ([m, p] \vdash x \mapsto v) \longrightarrow \text{valid-value } v \ (\text{stamp-expr } x))$ 
assumes  $yval$ :  $(\forall m \ p \ v. ([m, p] \vdash y \mapsto v) \longrightarrow \text{valid-value } v \ (\text{stamp-expr } y))$ 
shows  $val\text{-to-bool } v$ 
proof –
  obtain  $xval$  where  $xval\text{-def}$ :  $[m, p] \vdash x \mapsto xval$ 
    using  $assms(2)$  by blast
  obtain  $yval$  where  $yval\text{-def}$ :  $[m, p] \vdash y \mapsto yval$ 
    using  $assms(2)$  by blast
  have  $is\text{-IntVal32 } xval \vee is\text{-IntVal64 } xval$ 
    by (metis BinaryExprE Value.collapse(3) Value.collapse(4) Value.exhaust-disc
assms(2) binary-obj evalDet evaltree-not-undef valid-value.simps(19) xval-def xvalid)
  have  $is\text{-IntVal32 } yval \vee is\text{-IntVal64 } yval$ 
    by (metis BinaryExprE Value.collapse(3) Value.collapse(4) Value.exhaust-disc
assms(2) binary-obj evalDet evaltree-not-undef valid-value.simps(19) yval-def yvalid)
  have  $is\text{-IntVal32 } xval = is\text{-IntVal32 } yval$ 
    using BinaryExprE Value.collapse(2) <is-IntVal32 xval  $\vee$  is-IntVal64 xval>
<is-IntVal32 yval  $\vee$  is-IntVal64 yval> assms(2) bin-eval.simps(11) evalDet int-
val-less-than.simps(12) intval-less-than.simps(5) is-IntVal32-def xval-def yval-def
    by (smt (verit, ccfv-SIG) bin-eval.simps(12))
  have  $is\text{-IntVal64 } xval = is\text{-IntVal64 } yval$ 
    using <is-IntVal32 xval = is-IntVal32 yval> <is-IntVal32 xval  $\vee$  is-IntVal64 xval>
<is-IntVal32 yval  $\vee$  is-IntVal64 yval> by blast
  have  $(\text{intval-less-than } xval \ yval) \neq \text{UndefVal}$ 
    using  $assms(2)$ 
    by (metis bin-eval.simps(12) evalDet unfold-binary xval-def yval-def)
  have  $is\text{-IntVal32 } xval \implies ((\exists lo \ hi. \text{stamp-expr } x = \text{IntegerStamp } 32 \ lo \ hi) \wedge$ 
 $(\exists lo \ hi. \text{stamp-expr } y = \text{IntegerStamp } 32 \ lo \ hi))$ 

  sorry
  have  $is\text{-IntVal64 } xval \implies ((\exists lo \ hi. \text{stamp-expr } x = \text{IntegerStamp } 64 \ lo \ hi) \wedge$ 
 $(\exists lo \ hi. \text{stamp-expr } y = \text{IntegerStamp } 64 \ lo \ hi))$ 

  sorry
  have  $xvalid$ :  $\text{valid-value } xval \ (\text{stamp-expr } x)$ 
    using  $xvalid \ xval\text{-def}$  by auto
  have  $yvalid$ :  $\text{valid-value } yval \ (\text{stamp-expr } y)$ 
    using  $yvalid \ yval\text{-def}$  by auto
  { assume  $c$ :  $is\text{-IntVal32 } xval$ 
    obtain  $xxval$  where  $x32$ :  $xval = \text{IntVal32 } xxval$ 
      using  $c \ is\text{-IntVal32-def}$  by blast
    obtain  $yyval$  where  $y32$ :  $yval = \text{IntVal32 } yyval$ 
      using <is-IntVal32 xval = is-IntVal32 yval> c is-IntVal32-def by auto
    have  $xs$ :  $\exists lo \ hi. \text{stamp-expr } x = \text{IntegerStamp } 32 \ lo \ hi$ 
      by (simp add: <is-IntVal32 xval  $\implies$  ( $\exists lo \ hi. \text{stamp-expr } x = \text{IntegerStamp } 32$ 
lo hi)  $\wedge$  ( $\exists lo \ hi. \text{stamp-expr } y = \text{IntegerStamp } 32 \ lo \ hi)> c$ ))
    have  $ys$ :  $\exists lo \ hi. \text{stamp-expr } y = \text{IntegerStamp } 32 \ lo \ hi$ 
      using <is-IntVal32 xval  $\implies$  ( $\exists lo \ hi. \text{stamp-expr } x = \text{IntegerStamp } 32 \ lo \ hi)$ 
 $\wedge (\exists lo \ hi. \text{stamp-expr } y = \text{IntegerStamp } 32 \ lo \ hi)> c$  by blast
  }

```

```

have sint xval ≤ stpi-upper (stamp-expr x)
  using upper-bound-32 x32 xs xvalid by presburger
have stpi-lower (stamp-expr y) ≤ sint yyval
  using lower-bound-32 y32 ys yvalid by presburger
have stpi-upper (stamp-expr x) < stpi-lower (stamp-expr y)
  using assms(1) unfolding stamp-under.simps
  by auto
then have xval <ₛ yyval
  using assms(1) unfolding stamp-under.simps
  using ⟨sint xval ⊆ stpi-upper (stamp-expr x)⟩ ⟨stpi-lower (stamp-expr y) ⊆
sint yyval⟩ word-sless-alt by fastforce
  then have (intval-less-than xval yval) = IntVal32 1
    by (simp add: x32 y32)
}
note case32 = this
{ assume c: is-IntVal64 xval
  obtain x64 where x64: xval = IntVal64 x64
    using c is-IntVal64-def by blast
  obtain yyval where y64: yval = IntVal64 yyval
    using ⟨is-IntVal64 xval = is-IntVal64 yyval⟩ c is-IntVal64-def by auto
  have xs: ∃ lo hi. stamp-expr x = IntegerStamp 64 lo hi
    by (simp add: ⟨is-IntVal64 xval ⟹ (∃ lo hi. stamp-expr x = IntegerStamp 64
lo hi) ∧ (∃ lo hi. stamp-expr y = IntegerStamp 64 lo hi)⟩ c)
  have ys: ∃ lo hi. stamp-expr y = IntegerStamp 64 lo hi
    using ⟨is-IntVal64 xval ⟹ (∃ lo hi. stamp-expr x = IntegerStamp 64 lo hi)
∧ (∃ lo hi. stamp-expr y = IntegerStamp 64 lo hi)⟩ c by blast
  have sint xval ≤ stpi-upper (stamp-expr x)
    using upper-bound-64 x64 xs xvalid by presburger
  have stpi-lower (stamp-expr y) ≤ sint yyval
    using lower-bound-64 y64 ys yvalid by presburger
  have stpi-upper (stamp-expr x) < stpi-lower (stamp-expr y)
    using assms(1) unfolding stamp-under.simps
    by auto
  then have xval <ₛ yyval
    using assms(1) unfolding stamp-under.simps
    using ⟨sint xval ⊆ stpi-upper (stamp-expr x)⟩ ⟨stpi-lower (stamp-expr y) ⊆
sint yyval⟩ word-sless-alt by fastforce
  then have (intval-less-than xval yval) = IntVal32 1
    by (simp add: x64 y64)
}
note case64 = this
have (intval-less-than xval yval) = IntVal32 1
  using ⟨is-IntVal32 xval ∨ is-IntVal64 xval⟩ case32 case64 by fastforce
then show ?thesis
  by (metis EvalTreeE(5) assms(2) bin-eval.simps(12) evalDet val-to-bool.simps(1)
xval-def yval-def zero-neq-one)
qed

```

lemma stamp-under-semantics-inversed:

```

assumes stamp-under (stamp-expr y) (stamp-expr x)
assumes [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ v
assumes xvalid: (∀ m p v. ([m, p] ⊢ x ↦ v) ⟶ valid-value v (stamp-expr x))
assumes yvalid: (∀ m p v. ([m, p] ⊢ y ↦ v) ⟶ valid-value v (stamp-expr y))
shows ¬(val-to-bool v)
proof –
  obtain xval where xval-def: [m, p] ⊢ x ↦ xval
    using assms(2) by blast
  obtain yval where yval-def: [m, p] ⊢ y ↦ yval
    using assms(2) by blast
  have is-IntVal32 xval ∨ is-IntVal64 xval
    by (metis BinaryExprE Value.discI(1) Value.discI(2) assms(2) bin-eval.simps(12))
  binary-obj
    constantAsStamp.elims evalDet evaltree-not-undef intval-less-than.simps(9)
  xval-def)
  have is-IntVal32 yval ∨ is-IntVal64 yval
    by (metis BinaryExprE Value.discI(1) Value.discI(2) assms(2) bin-eval.simps(12))
  binary-obj
    constantAsStamp.elims evalDet evaltree-not-undef intval-less-than.simps(16)
  yval-def)
  have is-IntVal32 xval = is-IntVal32 yval
    by (metis BinaryExprE Value.collapse(2) ⟨is-IntVal32 xval ∨ is-IntVal64 xval⟩
      ⟨is-IntVal32 yval ∨ is-IntVal64 yval⟩ assms(2) bin-eval.simps(12) evalDet int-
      val-less-than.simps(12) intval-less-than.simps(5) is-IntVal32-def xval-def yval-def)
  have is-IntVal64 xval = is-IntVal64 yval
    using ⟨is-IntVal32 xval = is-IntVal32 yval⟩ ⟨is-IntVal32 xval ∨ is-IntVal64 xval⟩
    ⟨is-IntVal32 yval ∨ is-IntVal64 yval⟩ by blast
  have (intval-less-than xval yval) ≠ UndefinedVal
    using assms(2)
    by (metis BinaryExprE bin-eval.simps(12) evalDet xval-def yval-def)
  have is-IntVal32 xval ⟹ ((∃ lo hi. stamp-expr x = IntegerStamp 32 lo hi) ∧
    (∃ lo hi. stamp-expr y = IntegerStamp 32 lo hi))

  sorry
  have is-IntVal64 xval ⟹ ((∃ lo hi. stamp-expr x = IntegerStamp 64 lo hi) ∧
    (∃ lo hi. stamp-expr y = IntegerStamp 64 lo hi))

  sorry
  have xvalid: valid-value xval (stamp-expr x)
    using xvalid xval-def by auto
  have yvalid: valid-value yval (stamp-expr y)
    using yvalid yval-def by auto
  { assume c: is-IntVal32 xval
    obtain xxval where x32: xval = IntVal32 xxval
      using c is-IntVal32-def by blast
    obtain yyval where y32: yval = IntVal32 yyval
      using ⟨is-IntVal32 xval = is-IntVal32 yval⟩ c is-IntVal32-def by auto
    have xs: ∃ lo hi. stamp-expr x = IntegerStamp 32 lo hi
      by (simp add: ⟨is-IntVal32 xval ⟹ (∃ lo hi. stamp-expr x = IntegerStamp 32

```

```

lo hi) ∧ (∃ lo hi. stamp-expr y = IntegerStamp 32 lo hi)⟩ c)
  have ys: ∃ lo hi. stamp-expr y = IntegerStamp 32 lo hi
    using ⟨is-IntVal32 xval ⇒ (∃ lo hi. stamp-expr x = IntegerStamp 32 lo hi)
  ∧ (∃ lo hi. stamp-expr y = IntegerStamp 32 lo hi)⟩ c by blast
  have sint yyval ≤ stpi-upper (stamp-expr y)
    using y32 ys yvalid by force
  have stpi-lower (stamp-expr x) ≤ sint xxval
    using x32 xs xvalid by force
  have stpi-upper (stamp-expr y) < stpi-lower (stamp-expr x)
    using assms(1) unfolding stamp-under.simps
    by auto
  then have yyval <ₛ xxval
    using assms(1) unfolding stamp-under.simps
    using ⟨sint yyval ⊆ stpi-upper (stamp-expr y)⟩ ⟨stpi-lower (stamp-expr x) ⊆
sint xxval⟩ word-sless-alt by fastforce
  then have (intval-less-than xval yval) = IntVal32 0
    using signed.less-not-sym x32 y32 by fastforce
}
note case32 = this
{ assume c: is-IntVal64 xval
  obtain xxval where x64: xval = IntVal64 xxval
    using c is-IntVal64-def by blast
  obtain yyval where y64: yval = IntVal64 yyval
    using ⟨is-IntVal64 xval = is-IntVal64 yval⟩ c is-IntVal64-def by auto
  have xs: ∃ lo hi. stamp-expr x = IntegerStamp 64 lo hi
    by (simp add: ⟨is-IntVal64 xval ⇒ (∃ lo hi. stamp-expr x = IntegerStamp 64
lo hi) ∧ (∃ lo hi. stamp-expr y = IntegerStamp 64 lo hi)⟩ c)
  have ys: ∃ lo hi. stamp-expr y = IntegerStamp 64 lo hi
    using ⟨is-IntVal64 xval ⇒ (∃ lo hi. stamp-expr x = IntegerStamp 64 lo hi)
  ∧ (∃ lo hi. stamp-expr y = IntegerStamp 64 lo hi)⟩ c by blast
  have sint yyval ≤ stpi-upper (stamp-expr y)
    using y64 ys yvalid by force
  have stpi-lower (stamp-expr x) ≤ sint xxval
    using x64 xs xvalid by force
  have stpi-upper (stamp-expr y) < stpi-lower (stamp-expr x)
    using assms(1) unfolding stamp-under.simps
    by auto
  then have yyval <ₛ xxval
    using assms(1) unfolding stamp-under.simps
    using ⟨sint yyval ⊆ stpi-upper (stamp-expr y)⟩ ⟨stpi-lower (stamp-expr x) ⊆
sint xxval⟩ word-sless-alt by fastforce
  then have (intval-less-than xval yval) = IntVal32 0
    using signed.less-imp-triv x64 y64 by fastforce
}
note case64 = this
have (intval-less-than xval yval) = IntVal32 0
  using ⟨is-IntVal32 xval ∨ is-IntVal64 xval⟩ case32 case64 by fastforce
then show ?thesis
  by (metis BinaryExprE assms(2) bin-eval.simps(12) evalDet val-to-bool.simps(1)

```

```
xval-def yval-def)
qed
```

```
end
```

10 Optization DSLs

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

```
datatype 'a Rewrite =
  Transform 'a 'a (-  $\mapsto$  - 10) |
  Conditional 'a 'a bool (-  $\mapsto$  - 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 $\langle markup.ML \rangle$

ML \langle

```
structure IRExprTranslator : DSL-TRANSLATION =
struct
fun markup DSL-Tokens.Add = @{term BinaryExpr} $ @{term BinAdd}
  | markup DSL-Tokens.Sub = @{term BinaryExpr} $ @{term BinSub}
  | markup DSL-Tokens.Mul = @{term BinaryExpr} $ @{term BinMul}
  | markup DSL-Tokens.And = @{term BinaryExpr} $ @{term BinAnd}
  | markup DSL-Tokens.Or = @{term BinaryExpr} $ @{term BinOr}
  | markup DSL-Tokens.Xor = @{term BinaryExpr} $ @{term BinXor}
  | markup DSL-Tokens.ShortCircuitOr = @{term BinaryExpr} $ @{term Bin-
ShortCircuitOr}
  | markup DSL-Tokens.Abs = @{term UnaryExpr} $ @{term UnaryAbs}
  | markup DSL-Tokens.Less = @{term BinaryExpr} $ @{term BinIntegerLessThan}
  | markup DSL-Tokens.Equals = @{term BinaryExpr} $ @{term BinIntegerEquals}
  | markup DSL-Tokens.Not = @{term UnaryExpr} $ @{term UnaryNot}
  | markup DSL-Tokens.Negate = @{term UnaryExpr} $ @{term UnaryNeg}
  | markup DSL-Tokens.LogicNegate = @{term UnaryExpr} $ @{term UnaryLog-
```

```

icNegation}
| markup DSL-Tokens.LeftShift = @{term BinaryExpr} $ @{term BinLeftShift}
| markup DSL-Tokens.RightShift = @{term BinaryExpr} $ @{term BinRightShift}
| markup DSL-Tokens.UnsignedRightShift = @{term BinaryExpr} $ @{term Bin-
URightShift}
| markup DSL-Tokens.Conditional = @{term ConditionalExpr}
| markup DSL-Tokens.Constant = @{term ConstantExpr}
| markup DSL-Tokens.TrueConstant = @{term ConstantExpr (IntVal32 1)}
| markup DSL-Tokens.FalseConstant = @{term ConstantExpr (IntVal32 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 IntVal32}
| markup DSL-Tokens.TrueConstant = @{term IntVal32 1}
| markup DSL-Tokens.FalseConstant = @{term IntVal32 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[(e1 < e2) ? e1 : e2]

ConditionalExpr (BinaryExpr BinIntegerLessThan e1 e2) e1 e2

```

value expression example

```

value val[(e1 < e2) ? e1 : e2]

intval-conditional (intval-less-than e1 e2) e1 e2

```

```

value exp[((e1 - e2) + (const (IntVal32 0)) + e2) ⟶ e1 when True]
value val[((e1 - e2) + (const 0) + e2) ⟶ e1 when True]

```

word expression example

value $\text{bin}[x \ \& \ y \mid z]$

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

value $\text{bin}[\neg x]$

value $\text{val}[\neg x]$

value $\text{exp}[\neg x]$

value $\text{bin}![x]$

value $\text{val}![x]$

value $\text{exp}![x]$

value $\text{bin}[\neg x]$

value $\text{val}[\neg x]$

value $\text{exp}[\neg x]$

value $\text{bin}[\sim x]$

value $\text{val}[\sim x]$

value $\text{exp}[\sim x]$

value $\sim x$

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

$\text{phase} :: \text{thy-decl}$ **and**

$\text{terminating} :: \text{quasi-command}$ **and**

$\text{print-phases} :: \text{diag}$ **and**

$\text{optimization} :: \text{thy-goal-defn}$

begin

ML \langle

```

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: string, rewrite: term Rewrite}

structure RewriteRule : Rule =
struct
type T = rewrite;

fun pretty-rewrite ctxt (Transform (from, to)) =
  Pretty.block [
    Syntax.pretty-term ctxt from,
    Pretty.str  $\mapsto$  ,
    Syntax.pretty-term ctxt to
  ]
| pretty-rewrite ctxt (Conditional (from, to, cond)) =
  Pretty.block [
    Syntax.pretty-term ctxt from,
    Pretty.str  $\mapsto$  ,
    Syntax.pretty-term ctxt to,
    Pretty.str when ,
    Syntax.pretty-term ctxt cond
  ]
| pretty-rewrite - - = Pretty.str not implemented

fun pretty ctxt t =
  Pretty.block [
    Pretty.str ((#name t) ^ ":"),
    pretty-rewrite ctxt (#rewrite t)
  ]
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 ctxt =
  let
    val thy = Proof-Context.theory-of ctxt;
    fun print phase = RewritePhase.pretty phase ctxt
  in
    map print (RewritePhase.phases thy)
  end

```

```

fun print-optimizations thy =
  print-phases thy |> Pretty.writeln-chunks

val - =
  Outer-Syntax.command command-keyword 'print-phases'
  print debug information for optimizations
  (Scan.succeed
    (Toplevel.keep (print-optimizations o Toplevel.context-of)));
  >

```

ML-file *rewrites.ML*

```

fun rewrite-preservation :: IRExp Rewrite  $\Rightarrow$  bool where
  rewrite-preservation (Transform x y) = (y  $\leq$  x) |
  rewrite-preservation (Conditional x y cond) = (cond  $\longrightarrow$  (y  $\leq$  x)) |
  rewrite-preservation (Sequential x y) = (rewrite-preservation x  $\wedge$  rewrite-preservation
y) |
  rewrite-preservation (Transitive x) = rewrite-preservation x

fun rewrite-termination :: IRExp Rewrite  $\Rightarrow$  (IRExp  $\Rightarrow$  nat)  $\Rightarrow$  bool where
  rewrite-termination (Transform x y) trm = (trm x > trm y) |
  rewrite-termination (Conditional x y cond) trm = (cond  $\longrightarrow$  (trm x > trm y)) |
  rewrite-termination (Sequential x y) trm = (rewrite-termination x trm  $\wedge$  rewrite-termination
y trm) |
  rewrite-termination (Transitive x) trm = rewrite-termination x trm

fun intval :: Value Rewrite  $\Rightarrow$  bool where
  intval (Transform x y) = (x  $\neq$  UndefVal  $\wedge$  y  $\neq$  UndefVal  $\longrightarrow$  x = y) |
  intval (Conditional x y cond) = (cond  $\longrightarrow$  (x = y)) |
  intval (Sequential x y) = (intval x  $\wedge$  intval y) |
  intval (Transitive x) = intval x

fun size :: IRExp  $\Rightarrow$  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  $y \implies 1 < \text{size } y$ 
  using size-pos apply (induction  $y$ ; auto)
  subgoal premises  $\text{prems}$  for  $op \ a \ b$ 
    apply (cases  $op = \text{BinAdd}$ )
    using size-non-add size-pos apply auto
    by (simp add: Suc-lessI one-is-add)+
  done

```

```

definition well-formed-equal ::  $\text{Value} \Rightarrow \text{Value} \Rightarrow \text{bool}$ 
  (infix  $\approx 50$ ) where
    well-formed-equal  $v_1 \ v_2 = (v_1 \neq \text{UndefVal} \longrightarrow v_1 = v_2)$ 

```

```

lemma well-formed-equal-defn [simp]:
  well-formed-equal  $v_1 \ v_2 = (v_1 \neq \text{UndefVal} \longrightarrow v_1 = v_2)$ 
  unfolding well-formed-equal-def by simp

```

end

11.1 Conditional Expression

```

theory ConditionalPhase
  imports
    Common
  begin

```

```

  phase Conditional
    terminating size
  begin

```

```

lemma negates:  $\text{is-IntVal32 } e \vee \text{is-IntVal64 } e \implies \text{val-to-bool } (\text{val}[e]) \equiv \neg(\text{val-to-bool } (\text{val}[\neg e]))$ 
  using intval-logic-negation.simps unfolding logic-negate-def
  by (smt (verit, best) Value.collapse(1) is-IntVal64-def val-to-bool.simps(1) val-to-bool.simps(2) zero-neq-one)

```

```

lemma negation-condition-intval:
  assumes  $e \neq \text{UndefVal} \wedge \neg(\text{is-ObjRef } e) \wedge \neg(\text{is-ObjStr } e)$ 
  shows  $\text{val}[(\neg e) \ ? \ x : y] = \text{val}[e \ ? \ y : x]$ 
  using assms by (cases  $e$ ; auto simp: negates logic-negate-def)

```

```

optimization negate-condition:  $((\neg e) \ ? \ x : y) \longmapsto (e \ ? \ y : x)$ 
  apply simp using negation-condition-intval
  by (smt (verit, ccfv-SIG) ConditionalExpr ConditionalExprE Value.collapse(3)
    Value.collapse(4) Value.exhaust-disc evaltree-not-undef intval-logic-negation.simps(4)
    intval-logic-negation.simps(5) negates unary-eval.simps(4) unfold-unary)

```

```

optimization const-true:  $(\text{true} \ ? \ x : y) \longmapsto x$  .

```

optimization *const-false*: $(\text{false} \text{ ? } x : y) \mapsto y$.

optimization *equal-branches*: $(e \text{ ? } x : x) \mapsto x$.

definition *wff-stamps* :: *bool* **where**

wff-stamps = $(\forall m \ p \ \text{expr} \ \text{val} . ([m, p] \vdash \text{expr} \mapsto \text{val}) \longrightarrow \text{valid-value} \ \text{val} \ (\text{stamp-expr} \ \text{expr}))$

definition *wf-stamp* :: *IRExpr* \Rightarrow *bool* **where**

wf-stamp *e* = $(\forall m \ p \ v . ([m, p] \vdash e \mapsto v) \longrightarrow \text{valid-value} \ v \ (\text{stamp-expr} \ e))$

optimization *b[intval]*: $((x \ \text{eq} \ y) \text{ ? } x : y) \mapsto y$
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*: $((x < y) \text{ ? } x : y) \mapsto x$
when (*stamp-under* (*stamp-expr* *x*) (*stamp-expr* *y*)
 \wedge *wf-stamp* *x* \wedge *wf-stamp* *y*)

apply *auto*

using *stamp-under.simps wf-stamp-def val-to-bool.simps*

sorry

optimization *opt-conditional-eq-is-RHS*: $((\text{BinaryExpr} \ \text{BinIntegerEquals} \ x \ y) \text{ ? } x : y) \mapsto y$

apply *simp-all* **apply** *auto* **using** *b*

apply (*metis* (*mono-tags*, *lifting*) *Canonicalization.intval.simps(1) evalDet*
intval-conditional.simps intval-equals.simps(10))

done

optimization *opt-normalize-x*: $((x \text{ eq } \text{const } (\text{IntVal32 } 0)) \text{ ? } \\ (\text{const } (\text{IntVal32 } 0)) : (\text{const } (\text{IntVal32 } 1))) \mapsto x \\ \text{when } (x = \text{ConstantExpr } (\text{IntVal32 } 0) \mid (x = \text{ConstantExpr } (\text{IntVal32 } 1))) \\ \text{done}$

optimization *opt-normalize-x2*: $((x \text{ eq } (\text{const } (\text{IntVal32 } 1))) \text{ ? } \\ (\text{const } (\text{IntVal32 } 1)) : (\text{const } (\text{IntVal32 } 0))) \mapsto x \\ \text{when } (x = \text{ConstantExpr } (\text{IntVal32 } 0) \mid (x = \text{ConstantExpr } (\text{IntVal32 } 1))) \\ \text{done}$

optimization *opt-flip-x*: $((x \text{ eq } (\text{const } (\text{IntVal32 } 0))) \text{ ? } \\ (\text{const } (\text{IntVal32 } 1)) : (\text{const } (\text{IntVal32 } 0))) \mapsto \\ x \oplus (\text{const } (\text{IntVal32 } 1)) \\ \text{when } (x = \text{ConstantExpr } (\text{IntVal32 } 0) \mid (x = \text{ConstantExpr } (\text{IntVal32 } 1))) \\ \text{done}$

optimization *opt-flip-x2*: $((x \text{ eq } (\text{const } (\text{IntVal32 } 1))) \text{ ? } \\ (\text{const } (\text{IntVal32 } 0)) : (\text{const } (\text{IntVal32 } 1))) \mapsto \\ x \oplus (\text{const } (\text{IntVal32 } 1)) \\ \text{when } (x = \text{ConstantExpr } (\text{IntVal32 } 0) \mid (x = \text{ConstantExpr } (\text{IntVal32 } 1))) \\ \text{done}$

optimization *opt-optimise-integer-test*:
 $((((x \& (\text{const } (\text{IntVal32 } 1))) \text{ eq } (\text{const } (\text{IntVal32 } 0))) \text{ ? } \\ (\text{const } (\text{IntVal32 } 0)) : (\text{const } (\text{IntVal32 } 1))) \mapsto \\ x \& (\text{const } (\text{IntVal32 } 1)) \\ \text{when } (\text{stamp-expr } x = \text{default-stamp}) \\ \text{apply } \text{simp-all} \\ \text{apply } \text{auto} \\ \text{using } \text{val-optimise-integer-test } \text{sorry}$

optimization *opt-optimise-integer-test-2*:
 $((((x \& (\text{const } (\text{IntVal32 } 1))) \text{ eq } (\text{const } (\text{IntVal32 } 0))) \text{ ? } \\ (\text{const } (\text{IntVal32 } 0)) : (\text{const } (\text{IntVal32 } 1))) \mapsto \\ x \\ \text{when } (x = \text{ConstantExpr } (\text{IntVal32 } 0) \mid (x = \text{ConstantExpr } (\text{IntVal32 } 1))) \\ \text{done}$


```

optimization opt-conditional-eliminate-known-less: (( $x < y$ ) ?  $x : y$ )  $\mapsto$   $x$ 
  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  $\Rightarrow$  IRNode  $\Rightarrow$  IRNode  $\Rightarrow$  TriState  $\Rightarrow$  bool
  (-  $\vdash$  - & -  $\hookrightarrow$  -) for  $g$  where
    eq-imp-less:
       $g \vdash (\text{IntegerEqualsNode } x \ y) \ \& \ (\text{IntegerLessThanNode } x \ y) \hookrightarrow \text{KnownFalse} \mid$ 
    eq-imp-less-rev:
       $g \vdash (\text{IntegerEqualsNode } x \ y) \ \& \ (\text{IntegerLessThanNode } y \ x) \hookrightarrow \text{KnownFalse} \mid$ 
    less-imp-rev-less:
       $g \vdash (\text{IntegerLessThanNode } x \ y) \ \& \ (\text{IntegerLessThanNode } y \ x) \hookrightarrow \text{KnownFalse} \mid$ 
    less-imp-not-eq:

```

$g \vdash (\text{IntegerLessThanNode } x \ y) \ \& \ (\text{IntegerEqualsNode } x \ y) \hookrightarrow \text{KnownFalse} \mid$
less-imp-not-eq-rev:
 $g \vdash (\text{IntegerLessThanNode } x \ y) \ \& \ (\text{IntegerEqualsNode } y \ x) \hookrightarrow \text{KnownFalse} \mid$

x-imp-x:
 $g \vdash x \ \& \ x \hookrightarrow \text{KnownTrue} \mid$

negate-false:
 $\llbracket g \vdash x \ \& \ (\text{kind } g \ y) \hookrightarrow \text{KnownTrue} \rrbracket \implies g \vdash x \ \& \ (\text{LogicNegationNode } y) \hookrightarrow \text{KnownFalse} \mid$
negate-true:
 $\llbracket g \vdash x \ \& \ (\text{kind } g \ y) \hookrightarrow \text{KnownFalse} \rrbracket \implies g \vdash x \ \& \ (\text{LogicNegationNode } y) \hookrightarrow \text{KnownTrue}$

Total relation over partial implies relation

inductive *condition-implies* :: $\text{IRGraph} \Rightarrow \text{IRNode} \Rightarrow \text{IRNode} \Rightarrow \text{TriState} \Rightarrow \text{bool}$
 $(- \vdash - \ \& \ - \hookrightarrow -)$ **for** g **where**
 $\llbracket \neg(g \vdash a \ \& \ b \hookrightarrow \text{imp}) \rrbracket \implies (g \vdash a \ \& \ b \hookrightarrow \text{Unknown}) \mid$
 $\llbracket (g \vdash a \ \& \ b \hookrightarrow \text{imp}) \rrbracket \implies (g \vdash a \ \& \ b \hookrightarrow \text{imp})$

inductive *implies-tree* :: $\text{IRExpr} \Rightarrow \text{IRExpr} \Rightarrow \text{bool} \Rightarrow \text{bool}$
 $(- \ \& \ - \hookrightarrow -)$ **where**
eq-imp-less:
 $(\text{BinaryExpr } \text{BinIntegerEquals } x \ y) \ \& \ (\text{BinaryExpr } \text{BinIntegerLessThan } x \ y) \hookrightarrow \text{False} \mid$
eq-imp-less-rev:
 $(\text{BinaryExpr } \text{BinIntegerEquals } x \ y) \ \& \ (\text{BinaryExpr } \text{BinIntegerLessThan } y \ x) \hookrightarrow \text{False} \mid$
less-imp-rev-less:
 $(\text{BinaryExpr } \text{BinIntegerLessThan } x \ y) \ \& \ (\text{BinaryExpr } \text{BinIntegerLessThan } y \ x) \hookrightarrow \text{False} \mid$
less-imp-not-eq:
 $(\text{BinaryExpr } \text{BinIntegerLessThan } x \ y) \ \& \ (\text{BinaryExpr } \text{BinIntegerEquals } x \ y) \hookrightarrow \text{False} \mid$
less-imp-not-eq-rev:
 $(\text{BinaryExpr } \text{BinIntegerLessThan } x \ y) \ \& \ (\text{BinaryExpr } \text{BinIntegerEquals } y \ x) \hookrightarrow \text{False} \mid$

x-imp-x:
 $x \ \& \ x \hookrightarrow \text{True} \mid$

negate-false:
 $\llbracket x \ \& \ y \hookrightarrow \text{True} \rrbracket \implies x \ \& \ (\text{UnaryExpr } \text{UnaryLogicNegation } y) \hookrightarrow \text{False} \mid$
negate-true:
 $\llbracket x \ \& \ y \hookrightarrow \text{False} \rrbracket \implies x \ \& \ (\text{UnaryExpr } \text{UnaryLogicNegation } y) \hookrightarrow \text{True}$

Proofs that the implies relation is correct with respect to the existing eval-

uation semantics.

experiment begin

lemma *logic-negate-type*:

assumes $[m, p] \vdash \text{UnaryExpr } \text{UnaryLogicNegation } x \mapsto v$

assumes $v \neq \text{UndefVal}$

shows $\exists v2. [m, p] \vdash x \mapsto \text{IntVal32 } v2$

proof –

obtain ve **where** $ve: [m, p] \vdash x \mapsto ve$

using *assms(1)* **by** *blast*

then have $[m, p] \vdash \text{UnaryExpr } \text{UnaryLogicNegation } x \mapsto \text{unary-eval } \text{UnaryLogicNegation } ve$

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 $[m, p] \vdash y \mapsto val$

assumes $[m, p] \vdash \text{UnaryExpr } \text{UnaryLogicNegation } y \mapsto invval$

assumes $invval \neq \text{UndefVal}$

shows $\text{val-to-bool } val \longleftrightarrow \neg(\text{val-to-bool } invval)$

proof –

obtain v **where** $invval = \text{unary-eval } \text{UnaryLogicNegation } v$

using *assms(2)* **by** *blast*

then have $[m, p] \vdash y \mapsto v$ **using** *UnaryExprE* *assms(1,2)* **sorry**

then show *?thesis* **sorry**

qed

lemma *logic-negation-relation*:

assumes $[g, m, p] \vdash y \mapsto val$

assumes $\text{kind } g \text{ neg} = \text{LogicNegationNode } y$

assumes $[g, m, p] \vdash \text{neg} \mapsto invval$

assumes $invval \neq \text{UndefVal}$

shows $\text{val-to-bool } val \longleftrightarrow \neg(\text{val-to-bool } invval)$

proof –

obtain $yencode$ **where** $g \vdash y \simeq yencode$

using *assms(1)* *encodeeval-def* **by** *auto*

then have $g \vdash \text{neg} \simeq \text{UnaryExpr } \text{UnaryLogicNegation } yencode$

using *rep.intros(7)* *assms(2)* **by** *simp*

then have $[m, p] \vdash \text{UnaryExpr } \text{UnaryLogicNegation } yencode \mapsto invval$

using *assms(3)* *encodeeval-def*

by (*metis* *repDet*)

obtain $v1$ **where** $[g, m, p] \vdash y \mapsto \text{IntVal32 } v1$

using *assms(1,2,3,4)* **using** *logic-negate-type* **sorry**

have $invval = \text{bool-to-val } (\neg(\text{val-to-bool } val))$

using *assms(1,2,3)* *evalDet* *unary-eval.simps(4)*

by (*smt* (*verit*, *ccfv-threshold*) *UnaryExprE* $\langle [g, m, p] \vdash y \mapsto \text{IntVal32 } v1 \rangle$
 $\langle [m, p] \vdash \text{UnaryExpr } \text{UnaryLogicNegation } yencode \mapsto invval \rangle \langle g \vdash y \simeq yencode \rangle$)

```

bool-to-val.simps(1) bool-to-val.simps(2) encodeeval-def graphDet intval-logic-negation.simps(1)
logic-negate-def val-to-bool.simps(1))
  have val-to-bool invval  $\longleftrightarrow \neg(\text{val-to-bool val})$ 
    using  $\langle \text{invval} = \text{bool-to-val } (\neg \text{val-to-bool val}) \rangle$  by force
  then show ?thesis
    by simp
qed
end

```

lemma *implies-valid*:

```

assumes  $x \ \& \ y \hookrightarrow \text{imp}$ 
assumes  $[m, p] \vdash x \mapsto v1$ 
assumes  $[m, p] \vdash y \mapsto v2$ 
assumes  $v1 \neq \text{UndefVal} \wedge v2 \neq \text{UndefVal}$ 
shows  $(\text{imp} \longrightarrow (\text{val-to-bool } v1 \longrightarrow \text{val-to-bool } v2)) \wedge$ 
       $(\neg \text{imp} \longrightarrow (\text{val-to-bool } v1 \longrightarrow \neg(\text{val-to-bool } v2)))$ 
  (is  $(?TP \longrightarrow ?TC) \wedge (?FP \longrightarrow ?FC)$ )
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 egeval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerEquals } x \ y) \mapsto \text{intval-equals } xval$ 
        yval
        using xval yval evaltree.BinaryExpr
        by (metis BinaryExprE bin-eval.simps(11) eq-imp-less.prems(1) evalDet)
      have lesseval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerLessThan } x \ y) \mapsto \text{intval-less-than}$ 
        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)  $\longrightarrow \neg(\text{val-to-bool } (\text{intval-less-than } xval$ 
        yval))
        using assms(4) apply (cases xval; cases yval; auto)
        apply (metis (full-types) val-to-bool.simps(1) Values.bool-to-val.simps(2)
          signed.less-irrefl)
      by (metis (mono-tags) val-to-bool.simps(1) Values.bool-to-val.elims signed.order.strict-implies-not-eq)
      then show ?case
        using egeval 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 egeval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerEquals } x \ y) \mapsto \text{intval-equals } xval$ 
        yval
        using xval yval evaltree.BinaryExpr
        by (metis BinaryExprE bin-eval.simps(11) eq-imp-less-rev.prems(1) evalDet)
      have lesseval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerLessThan } y \ x) \mapsto \text{intval-less-than}$ 
        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)  $\longrightarrow \neg(\text{val-to-bool } (\text{intval-less-than } yval$ 
        xval))
        using assms(4) apply (cases xval; cases yval; auto)
        apply (metis (full-types) val-to-bool.simps(1) Values.bool-to-val.simps(2)
          signed.less-irrefl)
      by (metis (full-types) val-to-bool.simps(1) Values.bool-to-val.elims signed.order.strict-implies-not-eq)

```

```

then show ?case
  using egeval lesseval
  by (metis eq-imp-less-rev.premis(1) eq-imp-less-rev.premis(2) evalDet)
next
case (less-imp-rev-less x y)
obtain xval where xval: [m, p] ⊢ x ↦ xval
  using less-imp-rev-less.premis(3)
  using less-imp-rev-less.premis(2) by blast
obtain yval where yval: [m, p] ⊢ y ↦ yval
  using less-imp-rev-less.premis(3)
  using less-imp-rev-less.premis(2) by blast
have lesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ intval-less-than
xval yval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(12) evalDet less-imp-rev-less.premis(1))
  have revlesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan y x) ↦ int-
val-less-than yval xval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(12) evalDet less-imp-rev-less.premis(2))
  have val-to-bool (intval-less-than xval yval) ⟶ ¬(val-to-bool (intval-less-than
yval xval))
  using assms(4) apply (cases xval; cases yval; auto)
  apply (metis val-to-bool.simps(1) Values.bool-to-val.elims signed.not-less-iff-gr-or-eq)
  by (metis val-to-bool.simps(1) Values.bool-to-val.elims signed.less-asym')
then show ?case
  by (metis evalDet less-imp-rev-less.premis(1) less-imp-rev-less.premis(2) lesseval
revlesseval)
next
case (less-imp-not-eq x y)
obtain xval where xval: [m, p] ⊢ x ↦ xval
  using less-imp-not-eq.premis(3)
  using less-imp-not-eq.premis(1) by blast
obtain yval where yval: [m, p] ⊢ y ↦ yval
  using less-imp-not-eq.premis(3)
  using less-imp-not-eq.premis(1) by blast
have egeval: [m, p] ⊢ (BinaryExpr BinIntegerEquals x y) ↦ intval-equals xval
yval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(11) evalDet less-imp-not-eq.premis(2))
  have lesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ intval-less-than
xval yval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(12) evalDet less-imp-not-eq.premis(1))
  have val-to-bool (intval-less-than xval yval) ⟶ ¬(val-to-bool (intval-equals xval
yval))
  using assms(4) apply (cases xval; cases yval; auto)
  apply (metis (full-types) bool-to-val.simps(2) signed.less-imp-not-eq val-to-bool.simps(1))
  by (metis (full-types) bool-to-val.simps(2) signed.less-imp-not-eq2 val-to-bool.simps(1))
then show ?case

```

```

    by (metis egeval evalDet less-imp-not-eq.prem(1) less-imp-not-eq.prem(2)
lesseval)
  next
    case (less-imp-not-eq-rev x y)
    obtain xval where xval: [m, p] ⊢ x ↦ xval
    using less-imp-not-eq-rev.prem(3)
    using less-imp-not-eq-rev.prem(1) by blast
    obtain yval where yval: [m, p] ⊢ y ↦ yval
    using less-imp-not-eq-rev.prem(3)
    using less-imp-not-eq-rev.prem(1) by blast
    have egeval: [m, p] ⊢ (BinaryExpr BinIntegerEquals y x) ↦ intval-equals yval
xval
    using xval yval evaltree.BinaryExpr
    by (metis BinaryExprE bin-eval.simp(11) evalDet less-imp-not-eq-rev.prem(2))
    have lesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ intval-less-than
xval yval
    using xval yval evaltree.BinaryExpr
    by (metis BinaryExprE bin-eval.simp(12) evalDet less-imp-not-eq-rev.prem(1))
    have val-to-bool (intval-less-than xval yval) ⟶ ¬(val-to-bool (intval-equals yval
xval))
    using assms(4) apply (cases xval; cases yval; auto)
    apply (metis (full-types) bool-to-val.simp(2) signed.less-imp-not-eq2 val-to-bool.simp(1))
    by (metis (full-types, opaque-lifting) val-to-bool.simp(1) Values.bool-to-val.elims
signed.dual-order.strict-implies-not-eq)
    then show ?case
    by (metis egeval evalDet less-imp-not-eq-rev.prem(1) less-imp-not-eq-rev.prem(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 ↦ imp
  assumes imp
  assumes [m, p] ⊢ x ↦ v1
  assumes [m, p] ⊢ y ↦ v2
  assumes v1 ≠ UndefVal ∧ v2 ≠ UndefVal
  shows val-to-bool v1 ⟶ val-to-bool v2
  using assms implies-valid
  by blast

```

```

lemma implies-false-valid:
  assumes  $x \ \& \ y \hookrightarrow \text{imp}$ 
  assumes  $\neg \text{imp}$ 
  assumes  $[m, p] \vdash x \mapsto v1$ 
  assumes  $[m, p] \vdash y \mapsto v2$ 
  assumes  $v1 \neq \text{UndefVal} \wedge v2 \neq \text{UndefVal}$ 
  shows  $\text{val-to-bool } v1 \longrightarrow \neg(\text{val-to-bool } v2)$ 
  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
     $\llbracket \text{alwaysDistinct } (\text{stamps } x) (\text{stamps } y) \rrbracket$ 
       $\implies \text{tryFold } (\text{IntegerEqualsNode } x \ y) \ \text{stamps } \text{False} \mid$ 
     $\llbracket \text{neverDistinct } (\text{stamps } x) (\text{stamps } y) \rrbracket$ 
       $\implies \text{tryFold } (\text{IntegerEqualsNode } x \ y) \ \text{stamps } \text{True} \mid$ 
     $\llbracket \text{is-IntegerStamp } (\text{stamps } x);$ 
       $\text{is-IntegerStamp } (\text{stamps } y);$ 
       $\text{stpi-upper } (\text{stamps } x) < \text{stpi-lower } (\text{stamps } y) \rrbracket$ 
       $\implies \text{tryFold } (\text{IntegerLessThanNode } x \ y) \ \text{stamps } \text{True} \mid$ 
     $\llbracket \text{is-IntegerStamp } (\text{stamps } x);$ 
       $\text{is-IntegerStamp } (\text{stamps } y);$ 
       $\text{stpi-lower } (\text{stamps } x) \geq \text{stpi-upper } (\text{stamps } y) \rrbracket$ 
       $\implies \text{tryFold } (\text{IntegerLessThanNode } x \ y) \ \text{stamps } \text{False}$ 

```

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  $\text{kind } g \ \text{nid} = \text{IntegerEqualsNode } x \ y$ 
  assumes  $[g, m, p] \vdash \text{nid} \mapsto v$ 
  assumes  $v \neq \text{UndefVal}$ 
  assumes  $([g, m, p] \vdash x \mapsto xval) \wedge ([g, m, p] \vdash y \mapsto yval)$ 
  shows  $\text{val-to-bool } (\text{intval-equals } xval \ yval) \longleftrightarrow v = \text{IntVal32 } 1$ 
proof –
  have  $v = \text{intval-equals } xval \ yval$ 
    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  $g \ \text{stamps}$ 
  assumes  $\text{kind } g \ \text{nid} = (\text{IntegerEqualsNode } x \ y)$ 

```



```

assumes  $[g, m, p] \vdash nid \mapsto v$ 
assumes alwaysDistinct (stamps x) (stamps y)
shows  $v = \text{IntVal32 } 0$ 
proof –
  have  $\forall \text{ val. } \neg(\text{valid-value val } (\text{join } (\text{stamps } x) (\text{stamps } y)))$ 
    using assms(1,4) unfolding alwaysDistinct.simps
  by (metis is-stamp-empty.elims(2) le-less-trans not-less valid32or64 valid-value.simps(1)
valid-value.simps(2))
  have  $\neg(\exists \text{ val. } ([g, m, p] \vdash x \mapsto \text{val}) \wedge ([g, m, p] \vdash y \mapsto \text{val}))$ 
    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  $[g, m, p] \vdash nid \mapsto v$ 
assumes neverDistinct (stamps x) (stamps y)
shows  $v = \text{IntVal32 } 1$ 
using assms IntegerEqualsNodeE sorry

```

lemma *tryFoldIntegerLessThanTrue*:

```

assumes wf-stamp g stamps
assumes kind g nid = (IntegerLessThanNode x y)
assumes  $[g, m, p] \vdash nid \mapsto v$ 
assumes stpi-upper (stamps x) < stpi-lower (stamps y)
shows  $v = \text{IntVal32 } 1$ 
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)  $\wedge$  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)  $\geq$  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)  $\wedge$  is-IntegerStamp (stamps y)
  using assms(4)
  sorry
then have  $\neg$ (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* ::

$IRExpr \text{ set} \Rightarrow (ID \Rightarrow Stamp) \Rightarrow IRGraph \Rightarrow ID \Rightarrow IRGraph \Rightarrow bool$ **where**
impliesTrue:

$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
 $g \vdash cid \simeq cond;$
 $\exists\ ce \in conds . (ce \ \&\ cond \hookrightarrow True);$
 $g' = constantCondition\ True\ ifcond\ (kind\ g\ ifcond)\ g$
 $\rrbracket \implies ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g' \mid$

impliesFalse:

$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
 $g \vdash cid \simeq cond;$
 $\exists\ ce \in conds . (ce \ \&\ cond \hookrightarrow False);$
 $g' = constantCondition\ False\ ifcond\ (kind\ g\ ifcond)\ g$
 $\rrbracket \implies ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g' \mid$

tryFoldTrue:

$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
 $cond = kind\ g\ cid;$
 $tryFold\ (kind\ g\ cid)\ stamps\ True;$
 $g' = constantCondition\ True\ ifcond\ (kind\ g\ ifcond)\ g$
 $\rrbracket \implies ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g' \mid$

tryFoldFalse:

$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
 $cond = kind\ g\ cid;$

```

    tryFold (kind g cid) stamps False;
    g' = constantCondition False ifcond (kind g ifcond) g
  ] ==> ConditionalEliminationStep conds stamps g ifcond g'

```

code-pred (modes: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$) *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  $\Rightarrow$  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  $\Rightarrow$  ID  $\Rightarrow$  IRGraph  $\Rightarrow$  ID option where
  nextEdge seen nid g =
    (let nids = (filter ( $\lambda$ nid'. nid'  $\notin$  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  $\Rightarrow$  ID  $\Rightarrow$  ID option where
  pred g nid = (case kind g nid of
    (MergeNode ends -)  $\Rightarrow$  Some (hd ends) |
    -  $\Rightarrow$ 
      (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` function 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  $\Rightarrow$  int  $\Rightarrow$  Stamp where
  clip-upper (IntegerStamp b l h) c = (IntegerStamp b l c) |
  clip-upper s c = s
fun clip-lower :: Stamp  $\Rightarrow$  int  $\Rightarrow$  Stamp where
  clip-lower (IntegerStamp b l h) c = (IntegerStamp b c h) |
  clip-lower s c = s

fun registerNewCondition :: IRGraph  $\Rightarrow$  Condition  $\Rightarrow$  (ID  $\Rightarrow$  Stamp)  $\Rightarrow$  (ID  $\Rightarrow$  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  $\Rightarrow$  'a  $\Rightarrow$  '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  $\Rightarrow$  (ID  $\times$  Seen  $\times$  Conditions  $\times$  StampFlow)  $\Rightarrow$  (ID  $\times$  Seen  $\times$  Conditions  $\times$  StampFlow) option  $\Rightarrow$  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
     $\llbracket$ 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;

```

$flow' = registerNewCondition\ g\ c\ (hdOr\ flow\ (stamp\ g))$
 $\implies Step\ g\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds', flow' \# flow)) \mid$

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

$\llbracket kind\ g\ nid = EndNode;$

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

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

$conds' = tl\ conds;$

$flow' = tl\ flow$

$\implies Step\ g\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds', flow')) \mid$

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

$\llbracket \neg(is-EndNode\ (kind\ g\ nid));$
 $\neg(is-BEGINNode\ (kind\ g\ nid));$

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

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

$\implies Step\ g\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds, flow)) \mid$

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

$\llbracket \neg(is-EndNode\ (kind\ g\ nid));$
 $\neg(is-BEGINNode\ (kind\ g\ nid));$

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

$None = nextEdge\ seen'\ nid\ g$

$\implies Step\ g\ (nid, seen, conds, flow)\ None \mid$

— We've already seen this node, give back None

$\llbracket nid \in seen \rrbracket \implies Step\ g\ (nid, seen, conds, flow)\ None$

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