

Veriopt

September 28, 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	Operator Semantics	3
1.1	Arithmetic Operators	6
1.2	Bitwise Operators	7
1.3	Comparison Operators	7
1.4	Narrowing and Widening Operators	8
1.5	Bit-Shifting Operators	9
1.5.1	Examples of Narrowing / Widening Functions	10
1.6	Fixed-width Word Theories	12
1.6.1	Support Lemmas for Upper/Lower Bounds	12
1.6.2	Support lemmas for take bit and signed take bit.	16
2	Stamp Typing	17
3	Graph Representation	22
3.1	IR Graph Nodes	22
3.2	IR Graph Node Hierarchy	30
3.3	IR Graph Type	36
3.3.1	Example Graphs	41
3.4	Control-flow Graph Traversal	42
3.5	Structural Graph Comparison	44
4	java.lang.Long	45
4.1	Long.numberOfLeadingZeros	46
4.2	Long.numberOfTrailingZeros	47
4.3	Long.bitCount	47
4.4	Long.zeroCount	47
5	Data-flow Semantics	51
5.1	Data-flow Tree Representation	51
5.2	Functions for re-calculating stamps	53
5.3	Data-flow Tree Evaluation	54
5.4	Data-flow Tree Refinement	56
5.5	Stamp Masks	57
5.6	Data-flow Tree Theorems	59
5.6.1	Deterministic Data-flow Evaluation	59
5.6.2	Typing Properties for Integer Evaluation Functions	59
5.6.3	Evaluation Results are Valid	61
5.6.4	Example Data-flow Optimisations	63
5.6.5	Monotonicity of Expression Refinement	63
5.7	Unfolding rules for evaltree quadruples down to bin-eval level	64
5.8	Lemmas about <i>new_int</i> and integer eval results.	65

6	Tree to Graph	70
6.1	Subgraph to Data-flow Tree	70
6.2	Data-flow Tree to Subgraph	74
6.3	Lift Data-flow Tree Semantics	79
6.4	Graph Refinement	79
6.5	Maximal Sharing	79
6.6	Formedness Properties	79
6.7	Dynamic Frames	81
6.8	Tree to Graph Theorems	93
6.8.1	Extraction and Evaluation of Expression Trees is De- terministic.	93
6.8.2	Monotonicity of Graph Refinement	100
6.8.3	Lift Data-flow Tree Refinement to Graph Refinement .	103
6.8.4	Term Graph Reconstruction	119
6.8.5	Data-flow Tree to Subgraph Preserves Maximal Sharing	127
7	Control-flow Semantics	141
7.1	Object Heap	141
7.2	Intraprocedural Semantics	142
7.3	Interprocedural Semantics	144
7.4	Big-step Execution	146
7.4.1	Heap Testing	147
7.5	Control-flow Semantics Theorems	147
7.5.1	Control-flow Step is Deterministic	148
8	Proof Infrastructure	152
8.1	Bisimulation	152
8.2	Graph Rewriting	153
8.3	Stuttering	158
8.4	Evaluation Stamp Theorems	159
8.4.1	Support Lemmas for Integer Stamps and Associated IntVal values	160
8.4.2	Validity of all Unary Operators	162
8.4.3	Support Lemmas for Binary Operators	164
8.4.4	Validity of Stamp Meet and Join Operators	167
8.4.5	Validity of conditional expressions	168
8.4.6	Validity of Whole Expression Tree Evaluation	168
9	Optization DSL	170
9.1	Markup	170
9.1.1	Expression Markup	171
9.1.2	Value Markup	172
9.1.3	Word Markup	173
9.2	Optimization Phases	174

9.3	Canonicalization DSL	174
9.3.1	Semantic Preservation Obligation	177
9.3.2	Termination Obligation	177
9.3.3	Standard Termination Measure	177
9.3.4	Automated Tactics	178
10	Canonicalization Optimizations	178
10.1	AbsNode Phase	180
10.2	AddNode Phase	185
10.3	AndNode Phase	188
10.4	BinaryNode Phase	193
10.5	ConditionalNode Phase	194
10.6	MulNode Phase	198
10.7	Experimental AndNode Phase	206
10.8	NotNode Phase	217
10.9	OrNode Phase	218
10.10	ShiftNode Phase	221
10.11	SignedDivNode Phase	223
10.12	SignedRemNode Phase	223
10.13	SubNode Phase	224
10.14	XorNode Phase	228
11	Conditional Elimination Phase	230
11.1	Individual Elimination Rules	231
11.2	Control-flow Graph Traversal	241

1 Operator Semantics

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

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

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

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

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

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

```
abbreviation valid-int-widths :: nat set where
  valid-int-widths  $\equiv$  {1, 8, 16, 32, 64}
```

```
experiment begin
```

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

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

```
setup-lifting type-definition-IntWidth
```

```
lift-definition IntWidthBits :: IntWidth  $\Rightarrow$  nat
is  $\lambda w. w$  .
end
```

```
experiment begin
```

Option 3: explicit type stored with each integer value.

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

fun *int-bits* :: *IntType* \Rightarrow *nat* **where**

int-bits *ILong* = 64 |
int-bits *IInt* = 32 |
int-bits *IShort* = 16 |
int-bits *IChar* = 16 |
int-bits *IByte* = 8 |
int-bits *IBoolean* = 1

fun *int-signed* :: *IntType* \Rightarrow *bool* **where**

int-signed *ILong* = *True* |
int-signed *IInt* = *True* |
int-signed *IShort* = *True* |
int-signed *IChar* = *False* |
int-signed *IByte* = *True* |
int-signed *IBoolean* = *True*

end

Option 4: int64 with the number of significant bits.

type-synonym *iwidth* = *nat*

type-synonym *objref* = *nat option*

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

IntVal *iwidth int64* |

ObjRef *objref* |
ObjStr *string*

fun *intval-bits* :: *Value* \Rightarrow *nat* **where**

intval-bits (*IntVal* *b v*) = *b*

fun *intval-word* :: *Value* \Rightarrow *int64* **where**

intval-word (*IntVal* *b v*) = *v*

fun *bit-bounds* :: *nat* \Rightarrow (*int* \times *int*) **where**

bit-bounds *bits* = (((2 \wedge *bits*) *div* 2) * -1, ((2 \wedge *bits*) *div* 2) - 1)

definition *logic-negate* :: ('*a*::*len*) *word* \Rightarrow '*a word* **where**

logic-negate *x* = (if *x* = 0 then 1 else 0)

fun *int-signed-value* :: *iwidth* \Rightarrow *int64* \Rightarrow *int* **where**
int-signed-value *b v* = *sint* (*signed-take-bit* (*b* - 1) *v*)

fun *int-unsigned-value* :: *iwidth* \Rightarrow *int64* \Rightarrow *int* **where**
int-unsigned-value *b v* = *uint* *v*

Converts an integer word into a Java value.

fun *new-int* :: *iwidth* \Rightarrow *int64* \Rightarrow *Value* **where**
new-int *b w* = *IntVal* *b* (*take-bit* *b w*)

Converts an integer word into a Java value, iff the two types are equal.

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

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

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

fun *bool-to-val* :: *bool* \Rightarrow *Value* **where**
bool-to-val *True* = (*IntVal* 32 1) |
bool-to-val *False* = (*IntVal* 32 0)

Converts an Isabelle bool into a Java value, iff the two types are equal.

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

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

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

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

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

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

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

1.1 Arithmetic Operators

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

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

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

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

```
fun intval-sub :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-sub (IntVal b1 v1) (IntVal b2 v2) = new-int-bin b1 b2 (v1-v2) |
  intval-sub - - =.UndefVal
```

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

```
fun intval-div :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-div (IntVal b1 v1) (IntVal b2 v2) =
    new-int-bin b1 b2 (word-of-int
      ((int-signed-value b1 v1) sdiv (int-signed-value b2 v2))) |
  intval-div - - =.UndefVal
```

```
fun intval-mod :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-mod (IntVal b1 v1) (IntVal b2 v2) =
    new-int-bin b1 b2 (word-of-int
      ((int-signed-value b1 v1) smod (int-signed-value b2 v2))) |
  intval-mod - - =.UndefVal
```

```
fun intval-negate :: Value  $\Rightarrow$  Value where
```



```

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

```

```

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

```

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

```

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

```

1.2 Bitwise Operators

```

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

```

```

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

```

```

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

```

```

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

```

1.3 Comparison Operators

```

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

```

```

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

```

```

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

```

```

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

```

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

1.4 Narrowing and Widening Operators

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

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

```
fun intval-narrow :: nat  $\Rightarrow$  nat  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-narrow inBits outBits (IntVal b v) =
    (if inBits = b  $\wedge$  0 < outBits  $\wedge$  outBits  $\leq$  inBits  $\wedge$  inBits  $\leq$  64
     then new-int outBits v
     else UndefVal) |
  intval-narrow - - - = UndefVal
```

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

```
fun intval-sign-extend :: nat  $\Rightarrow$  nat  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-sign-extend inBits outBits (IntVal b v) =
    (if inBits = b  $\wedge$  0 < inBits  $\wedge$  inBits  $\leq$  outBits  $\wedge$  outBits  $\leq$  64
     then new-int outBits (signed-take-bit (inBits - 1) v)
     else UndefVal) |
  intval-sign-extend - - - = UndefVal
```

```
fun intval-zero-extend :: nat  $\Rightarrow$  nat  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-zero-extend inBits outBits (IntVal b v) =
    (if inBits = b  $\wedge$  0 < inBits  $\wedge$  inBits  $\leq$  outBits  $\wedge$  outBits  $\leq$  64
     then new-int outBits (take-bit inBits v)
     else UndefVal) |
  intval-zero-extend - - - = UndefVal
```

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

lemma *intval-narrow-ok*:

```
assumes intval-narrow inBits outBits val  $\neq$  UndefVal
shows 0 < outBits  $\wedge$  outBits  $\leq$  inBits  $\wedge$  inBits  $\leq$  64  $\wedge$  outBits  $\leq$  64  $\wedge$ 
  is-IntVal val  $\wedge$ 
  intval-bits val = inBits
using assms intval-narrow.simps neq0-conv intval-bits.simps
by (metis Value.disc(2) intval-narrow.elims le-trans)
```

lemma *intval-sign-extend-ok*:

```
assumes intval-sign-extend inBits outBits val  $\neq$  UndefVal
shows 0 < inBits  $\wedge$ 
  inBits  $\leq$  outBits  $\wedge$  outBits  $\leq$  64  $\wedge$ 
```

```

    is-IntVal val ∧
    intval-bits val = inBits
  using assms intval-sign-extend.simps neq0-conv
  by (metis intval-bits.simps intval-sign-extend.elims is-IntVal-def)

```

```

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

```

1.5 Bit-Shifting Operators

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

value (128 :: 8 word) >> 2

```

Note that Java shift operators use unary numeric promotion, unlike other binary operators, which use binary numeric promotion (see the Java lan-

guage reference manual). This means that the left-hand input determines the output size, while the right-hand input can be any size.

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

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

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

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

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

1.5.1 Examples of Narrowing / Widening Functions

experiment begin

corollary intval-narrow 32 8 (IntVal 32 (256 + 128)) = IntVal 8 128 **by simp**

corollary intval-narrow 32 8 (IntVal 32 (-2)) = IntVal 8 254 **by simp**

corollary intval-narrow 32 1 (IntVal 32 (-2)) = IntVal 1 0 **by simp**

corollary intval-narrow 32 1 (IntVal 32 (-3)) = IntVal 1 1 **by simp**

corollary intval-narrow 32 8 (IntVal 64 (-2)) = UndefVal **by simp**

corollary intval-narrow 64 8 (IntVal 32 (-2)) = UndefVal **by simp**

corollary intval-narrow 64 8 (IntVal 64 254) = IntVal 8 254 **by simp**

corollary intval-narrow 64 8 (IntVal 64 (256+127)) = IntVal 8 127 **by simp**

corollary intval-narrow 64 64 (IntVal 64 (-2)) = IntVal 64 (-2) **by simp**

end

experiment begin

corollary intval-sign-extend 8 32 (IntVal 8 (256 + 128)) = IntVal 32 (2³² - 128) **by simp**

corollary intval-sign-extend 8 32 (IntVal 8 (-2)) = IntVal 32 (2³² - 2) **by simp**

corollary intval-sign-extend 1 32 (IntVal 1 (-2)) = IntVal 32 0 **by simp**

corollary intval-sign-extend 1 32 (IntVal 1 (-3)) = IntVal 32 (mask 32) **by simp**

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

experiment begin

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

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

experiment begin

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

lemma *intval-add-sym*:

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

code-deps *intval-add*

code-thms *intval-add*

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

```

lemma intval-add (IntVal 64 ( $2^{31}-1$ )) (IntVal 64 ( $2^{31}-1$ )) = IntVal 64 4294967294
  by eval

```

```

end

```

1.6 Fixed-width Word Theories

```

theory ValueThms
  imports Values
begin

```

1.6.1 Support Lemmas for Upper/Lower Bounds

```

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

```

```

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

```

```

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

```

```

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

```

Some min/max bounds for 64-bit words

```

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

```

```

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

```

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

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

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

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

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

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

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

A `bit_bounds` version of the above lemma.

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

```
lemma signed-take-bit-bounds64:
  fixes ival :: int64
  assumes n ≤ 64
```

```

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

```

```

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

```

```

lemma int-signed-value-range:
fixes  $ival :: \text{int64}$ 
assumes  $val = \text{int-signed-value } n \text{ ival}$ 
shows  $-(2^{(n-1)}) \leq val \wedge val < 2^n$ 
using signed-take-bit-range assms
by (smt (verit, ccfv-SIG) One-nat-def diff-less int-signed-value.elims len-gt-0
 $\text{len-num1 power-less-imp-less-exp power-strict-increasing sint-greater-eq sint-less}$ )

```

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

```

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

```

```

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

```

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

```

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

```


lemma *take-bit-same-size-range*:

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

lemma *take-bit-same-bounds*:

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

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

lemma *scast-max-bound*:

assumes $\text{sint}(v :: 'a :: \text{len word}) < M$
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 $(\text{result} :: 'b :: \text{len word}) = \text{scast}(v :: 'a :: \text{len word})$
shows $\text{sint result} < 2 \wedge \text{LENGTH}('a) \text{ div } 2$
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 $(\text{result} :: 'b :: \text{len word}) = \text{scast}(v :: 'a :: \text{len word})$

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

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

Results about *new_int*.

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

1.6.2 Support lemmas for take bit and signed take bit.

Lemmas for removing redundant take_bit wrappers.

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

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

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

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

```

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

lemma signed-take-take-bit[simp]:
  fixes x :: 'a :: len word
  assumes  $0 < b$ 
  shows signed-take-bit (b - 1) (take-bit b x) = signed-take-bit (b - 1) x
  by (smt (verit, best) Suc-diff-1 assms lessI linorder-not-less signed-take-bit-take-bit)

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

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

end

```

2 Stamp Typing

```

theory Stamp
  imports Values
begin

```

The GraalVM compiler uses the Stamp class to store range and type information for a given node in the IR graph. We model the Stamp class as a

datatype, Stamp, and provide a number of functions on the datatype which correspond to the class methods within the compiler.

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

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

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

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

  is-stamp-empty x = False
```

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

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

```
fun valid-stamp :: Stamp  $\Rightarrow$  bool where
  valid-stamp (IntegerStamp bits lo hi) =
    (0 < bits  $\wedge$  bits  $\leq$  64  $\wedge$ 
     fst (bit-bounds bits)  $\leq$  lo  $\wedge$  lo  $\leq$  snd (bit-bounds bits)  $\wedge$ 
     fst (bit-bounds bits)  $\leq$  hi  $\wedge$  hi  $\leq$  snd (bit-bounds bits)) |
  valid-stamp s = True
```

experiment begin

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

— A stamp which includes the full range of the type

```
fun unrestricted-stamp :: Stamp  $\Rightarrow$  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
True True True) |
  unrestricted-stamp - = IllegalStamp
```

```
fun is-stamp-unrestricted :: Stamp  $\Rightarrow$  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  $\Rightarrow$  Stamp where
  empty-stamp VoidStamp = VoidStamp |
  empty-stamp (IntegerStamp bits lower upper) = (IntegerStamp bits (snd (bit-bounds
bits)) (fst (bit-bounds bits))) |

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

— Calculate the meet stamp of two stamps

```
fun meet :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  Stamp where
  meet VoidStamp VoidStamp = VoidStamp |
  meet (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1  $\neq$  b2 then IllegalStamp else
    (IntegerStamp b1 (min l1 l2) (max u1 u2))
  ) |
```

```

meet (KlassPointerStamp nn1 an1) (KlassPointerStamp nn2 an2) = (
  KlassPointerStamp (nn1  $\wedge$  nn2) (an1  $\wedge$  an2)
) |
meet (MethodCountersPointerStamp nn1 an1) (MethodCountersPointerStamp
nn2 an2) = (
  MethodCountersPointerStamp (nn1  $\wedge$  nn2) (an1  $\wedge$  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 IntVal b (word-of-int l) else
UndefVal) |
  asConstant - = UndefVal

```

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

```

fun alwaysDistinct :: Stamp  $\Rightarrow$  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 (IntVal b v) = (IntegerStamp b (int-signed-value b v) (int-signed-value
b v)) |

constantAsStamp - = IllegalStamp

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

fun *valid-value* :: *Value* \Rightarrow *Stamp* \Rightarrow *bool* **where**
valid-value (IntVal b1 val) (IntegerStamp b l h) =
(if b1 = b then
valid-stamp (IntegerStamp b l h) \wedge
take-bit b val = val \wedge
l \leq int-signed-value b val \wedge int-signed-value b val \leq h
else False) |

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

definition *wf-value* :: *Value* \Rightarrow *bool* **where**
wf-value v = valid-value v (constantAsStamp v)

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

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

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

— 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))

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

3 Graph Representation

3.1 IR Graph Nodes

theory *IRNodes*
imports
Values
begin

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

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

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

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

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

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

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

```

| ReturnNode (ir-result-opt: INPUT option) (ir-memoryMap-opt: INPUT-EXT
option)
| RightShiftNode (ir-x: INPUT) (ir-y: INPUT)
| ShortCircuitOrNode (ir-x: INPUT-COND) (ir-y: INPUT-COND)
| SignExtendNode (ir-inputBits: nat) (ir-resultBits: nat) (ir-value: INPUT)
| SignedDivNode (ir-nid: ID) (ir-x: INPUT) (ir-y: INPUT) (ir-zeroCheck-opt: 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 ⇒ 'a list where
  opt-to-list None = [] |
  opt-to-list (Some v) = [v]

```

```

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

```

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

```

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

```

inputs-of-BytecodeExceptionNode:
inputs-of (BytecodeExceptionNode arguments stateAfter next) = arguments @
(opt-to-list stateAfter) |
inputs-of-ConditionalNode:
inputs-of (ConditionalNode condition trueValue falseValue) = [condition, true-
Value, falseValue] |
inputs-of-ConstantNode:
inputs-of (ConstantNode const) = [] |
inputs-of-DynamicNewArrayNode:
inputs-of (DynamicNewArrayNode elementType length0 voidClass stateBefore
next) = [elementType, length0] @ (opt-to-list voidClass) @ (opt-to-list stateBefore)
|
inputs-of-EndNode:
inputs-of (EndNode) = [] |
inputs-of-ExceptionObjectNode:
inputs-of (ExceptionObjectNode stateAfter next) = (opt-to-list stateAfter) |
inputs-of-FrameState:
inputs-of (FrameState monitorIds outerFrameState values virtualObjectMappings)
= monitorIds @ (opt-to-list outerFrameState) @ (opt-list-to-list values) @ (opt-list-to-list
virtualObjectMappings) |
inputs-of-IfNode:
inputs-of (IfNode condition trueSuccessor falseSuccessor) = [condition] |
inputs-of-IntegerBelowNode:
inputs-of (IntegerBelowNode x y) = [x, y] |
inputs-of-IntegerEqualsNode:
inputs-of (IntegerEqualsNode x y) = [x, y] |
inputs-of-IntegerLessThanNode:
inputs-of (IntegerLessThanNode x y) = [x, y] |
inputs-of-InvokeNode:
inputs-of (InvokeNode nid0 callTarget classInit stateDuring stateAfter next)
= callTarget # (opt-to-list classInit) @ (opt-to-list stateDuring) @ (opt-to-list
stateAfter) |
inputs-of-InvokeWithExceptionNode:
inputs-of (InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter
next exceptionEdge) = callTarget # (opt-to-list classInit) @ (opt-to-list 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* \Rightarrow *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) = [] ∧ successors-of (EndNode) = []*

unfolding *inputs-of-EndNode successors-of-EndNode by simp*

end

3.2 IR Graph Node Hierarchy

```
theory IRNodeHierarchy  
imports IRNodes  
begin
```

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

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

These functions have been automatically generated from the compiler.

```
fun is-EndNode :: IRNode  $\Rightarrow$  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 ⇒ bool where
  is-DeoptimizingFixedWithNextNode n = ((is-AbstractNewObjectNode n) ∨ (is-FixedBinaryNode
n))

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

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

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

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

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

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

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

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

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

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

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

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

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

```

n))

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

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

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

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

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

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

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

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

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

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

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

```

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

```
fun is-ArithmeticOperation :: IRNode  $\Rightarrow$  bool where
  is-ArithmeticOperation n = ((is-BinaryArithmeticNode n)  $\vee$  (is-IntegerConvertNode
n)  $\vee$  (is-ShiftNode n)  $\vee$  (is-UnaryArithmeticNode n))
```

```
fun is-ValueNumberable :: IRNode  $\Rightarrow$  bool where
  is-ValueNumberable n = ((is-FloatingNode n)  $\vee$  (is-ProxyNode n))
```

```
fun is-Lowerable :: IRNode  $\Rightarrow$  bool where
  is-Lowerable n = ((is-AbstractNewObjectNode n)  $\vee$  (is-AccessFieldNode n)  $\vee$ 
(is-BytecodeExceptionNode n)  $\vee$  (is-ExceptionObjectNode n)  $\vee$  (is-IntegerDivRemNode
n)  $\vee$  (is-UnwindNode n))
```

```
fun is-Virtualizable :: IRNode  $\Rightarrow$  bool where
  is-Virtualizable n = ((is-IsNullNode n)  $\vee$  (is-LoadFieldNode n)  $\vee$  (is-PiNode n)
 $\vee$  (is-StoreFieldNode n)  $\vee$  (is-ValueProxyNode n))
```

```
fun is-Simplifiable :: IRNode  $\Rightarrow$  bool where
  is-Simplifiable n = ((is-AbstractMergeNode n)  $\vee$  (is-BEGINNode n)  $\vee$  (is-IfNode
n)  $\vee$  (is-LoopExitNode n)  $\vee$  (is-MethodCallTargetNode n)  $\vee$  (is-NewArrayNode n))
```

```
fun is-StateSplit :: IRNode  $\Rightarrow$  bool where
  is-StateSplit n = ((is-AbstractStateSplit n)  $\vee$  (is-BEGINStateSplitNode n)  $\vee$  (is-StoreFieldNode
n))
```

```
fun is-ConvertNode :: IRNode  $\Rightarrow$  bool where
  is-ConvertNode n = ((is-IntegerConvertNode n))
```

```
fun is-sequential-node :: IRNode  $\Rightarrow$  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) ∧ (is-AndNode n2)) ∨
((is-BEGINNode n1) ∧ (is-BEGINNode n2)) ∨
((is-BytecodeExceptionNode n1) ∧ (is-BytecodeExceptionNode n2)) ∨
((is-ConditionalNode n1) ∧ (is-ConditionalNode n2)) ∨
((is-ConstantNode n1) ∧ (is-ConstantNode n2)) ∨
((is-DynamicNewArrayNode n1) ∧ (is-DynamicNewArrayNode n2)) ∨
((is-EndNode n1) ∧ (is-EndNode n2)) ∨
((is-ExceptionObjectNode n1) ∧ (is-ExceptionObjectNode n2)) ∨
((is-FrameState n1) ∧ (is-FrameState n2)) ∨
((is-IfNode n1) ∧ (is-IfNode n2)) ∨
((is-IntegerBelowNode n1) ∧ (is-IntegerBelowNode n2)) ∨
((is-IntegerEqualsNode n1) ∧ (is-IntegerEqualsNode n2)) ∨
((is-IntegerLessThanNode n1) ∧ (is-IntegerLessThanNode n2)) ∨
((is-InvokeNode n1) ∧ (is-InvokeNode n2)) ∨
((is-InvokeWithExceptionNode n1) ∧ (is-InvokeWithExceptionNode n2)) ∨
((is-IsNullNode n1) ∧ (is-IsNullNode n2)) ∨
((is-KillingBeginNode n1) ∧ (is-KillingBeginNode n2)) ∨
((is-LoadFieldNode n1) ∧ (is-LoadFieldNode n2)) ∨
((is-LogicNegationNode n1) ∧ (is-LogicNegationNode n2)) ∨
((is-LoopBeginNode n1) ∧ (is-LoopBeginNode n2)) ∨
((is-LoopEndNode n1) ∧ (is-LoopEndNode n2)) ∨
((is-LoopExitNode n1) ∧ (is-LoopExitNode n2)) ∨
((is-MergeNode n1) ∧ (is-MergeNode n2)) ∨
((is-MethodCallTargetNode n1) ∧ (is-MethodCallTargetNode n2)) ∨
((is-MulNode n1) ∧ (is-MulNode n2)) ∨
((is-NegateNode n1) ∧ (is-NegateNode n2)) ∨
((is-NewArrayNode n1) ∧ (is-NewArrayNode n2)) ∨
((is-NewInstanceNode n1) ∧ (is-NewInstanceNode n2)) ∨
((is-NotNode n1) ∧ (is-NotNode n2)) ∨
((is-OrNode n1) ∧ (is-OrNode n2)) ∨
((is-ParameterNode n1) ∧ (is-ParameterNode n2)) ∨
((is-PiNode n1) ∧ (is-PiNode n2)) ∨
((is-ReturnNode n1) ∧ (is-ReturnNode n2)) ∨
((is-ShortCircuitOrNode n1) ∧ (is-ShortCircuitOrNode n2)) ∨
((is-SignedDivNode n1) ∧ (is-SignedDivNode n2)) ∨
((is-StartNode n1) ∧ (is-StartNode n2)) ∨
((is-StoreFieldNode n1) ∧ (is-StoreFieldNode n2)) ∨
((is-SubNode n1) ∧ (is-SubNode n2)) ∨
((is-UnwindNode n1) ∧ (is-UnwindNode n2)) ∨
((is-ValuePhiNode n1) ∧ (is-ValuePhiNode n2)) ∨
((is-ValueProxyNode n1) ∧ (is-ValueProxyNode n2)) ∨
((is-XorNode n1) ∧ (is-XorNode n2))

```

end

3.3 IR Graph Type

```

theory IRGraph
  imports

```

```

    IRNodeHierarchy
    Stamp
    HOL-Library.FSet
    HOL.Relation
begin

```

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

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

```

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

```

setup-lifting type-definition-IRGraph

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lift-definition as-list :: IRGraph  $\Rightarrow$  (ID  $\times$  IRNode  $\times$  Stamp) list
is  $\lambda g.$  map ( $\lambda k. (k, \text{the } (g k))$ ) (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 ( $\lambda n. \text{fst } (\text{snd } n) \neq \text{NoNode}$ ) 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* = $\{(n, \text{kind } g \ n, \text{stamp } g \ n) \mid n . n \in \text{ids } g\}$

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

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

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

code-datatype *irgraph*

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.1 Example Graphs

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

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

Example 2: public static int sq(int x) return x * x;
 [1 P(0)] / [0 Start] [4 *] | / V / [5 Return]

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

]

```

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

end

```

3.4 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 ( $\lambda$ 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))))
  )

```

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

type-synonym 'a TraversalState = (ID × Seen × 'a)

inductive Step

∴ ('a TraversalState ⇒ 'a) ⇒ IRGraph ⇒ 'a TraversalState ⇒ 'a TraversalState
option ⇒ bool

for sa g **where**

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

[[kind g nid = BeginNode nid';

nid ∉ seen;
seen' = {nid} ∪ seen;

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

analysis' = sa (nid, seen, analysis)]
⇒ Step sa g (nid, seen, analysis) (Some (nid', seen', analysis')) |

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

[[kind g nid = EndNode;

nid ∉ seen;
seen' = {nid} ∪ seen;

nid' = any-usage g nid;

analysis' = sa (nid, seen, analysis)]
⇒ Step sa g (nid, seen, analysis) (Some (nid', seen', analysis')) |

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

[[¬(is-EndNode (kind g nid));
¬(is-BeginNode (kind g nid));

nid ∉ seen;
seen' = {nid} ∪ seen;

Some nid' = nextEdge seen' nid g;

analysis' = sa (nid, seen, analysis)]
⇒ Step sa g (nid, seen, analysis) (Some (nid', seen', analysis')) |

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

```

[[¬(is-EndNode (kind g nid));
  ¬(is-BeginNode (kind g nid));

  nid ∉ seen;
  seen' = {nid} ∪ seen;

  None = nextEdge seen' nid g]]
⇒ Step sa g (nid, seen, analysis) None |

— We've already seen this node, give back None
[[nid ∈ seen]] ⇒ Step sa g (nid, seen, analysis) None

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

end

```

3.5 Structural Graph Comparison

```

theory
  Comparison
imports
  IRGraph
begin

```

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

```

fun find-ref-nodes :: IRGraph ⇒ (ID → ID) where
find-ref-nodes g = map-of
  (map (λn. (n, ir-ref (kind g n))) (filter (λid. is-RefNode (kind g id)) (sorted-list-of-set
    (ids g))))

```

```

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

```

```

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

```

```

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

```

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

$\llbracket \text{kind } g1 \text{ } n1 = \text{RefNode } ref; \text{nodeEq } m \text{ } g1 \text{ } ref \text{ } g2 \text{ } n2 \rrbracket \implies \text{nodeEq } m \text{ } g1 \text{ } n1 \text{ } g2 \text{ } n2 \mid$
 $\llbracket x = \text{kind } g1 \text{ } n1;$
 $y = \text{kind } g2 \text{ } n2;$
 $\text{is-same-ir-node-type } x \text{ } y;$
 $\text{replace-ref-nodes } g1 \text{ } m \text{ } (\text{successors-of } x) = \text{successors-of } y;$
 $\text{replace-ref-nodes } g1 \text{ } m \text{ } (\text{inputs-of } x) = \text{inputs-of } y \rrbracket$
 $\implies \text{nodeEq } m \text{ } g1 \text{ } n1 \text{ } g2 \text{ } n2$

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

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

diffNodesGraph *g1* *g2* = (let *refNodes* = *find-ref-nodes* *g1* in
 $\{ n . n \in \text{Predicate.the } (\text{reachables-i-i-i-o } g1 \text{ } [0] \text{ } \{\}) \} \wedge (\text{case } \text{refNodes } n \text{ of } \text{Some}$
 $- \Rightarrow \text{False} \mid - \Rightarrow \text{True} \wedge \neg(\text{nodeEq } \text{refNodes } g1 \text{ } n \text{ } g2 \text{ } n)) \}$)

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

fun *eqGraph* :: *IRGraph* \Rightarrow *IRGraph* \Rightarrow *bool* **where**

eqGraph *isabelle-graph* *graal-graph* = ((*diffNodesGraph* *isabelle-graph* *graal-graph*)
= $\{\}$)

end

4 java.lang.Long

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

theory *Long*

imports *ValueThms*

begin

lemma *negative-all-set-32*:

$n < 32 \implies \text{bit } (-1::\text{int32}) \text{ } n$

apply *transfer* **by** *auto*

definition *MaxOrNeg* :: *nat set* \Rightarrow *int* **where**

MaxOrNeg *s* = (if *s* = $\{\}$ then -1 else *Max* *s*)

definition *MinOrHighest* :: *nat set* \Rightarrow *nat* \Rightarrow *nat* **where**

$MinOrHighest\ s\ m = (if\ s = \{\} \text{ then } m \text{ else } Min\ s)$

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

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

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

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

4.1 Long.numberOfLeadingZeros

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

lemma $MaxOrNeg-neg$: $MaxOrNeg\ \{\} = -1$
by ($simp\ add$: $MaxOrNeg-def$)

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

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

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

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

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

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

lemma *numberOfLeadingZeros-range*: $0 \leq \text{numberOfLeadingZeros } a \wedge \text{numberOfLeadingZeros } a \leq \text{Nat.size } a$
unfolding *numberOfLeadingZeros-def*
using *MaxOrNeg-def highestOneBit-def nat-le-iff*
by (*smt (verit) bot-nat-0.extremum int-eq-iff*)

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

4.2 Long.numberOfTrailingZeros

definition *numberOfTrailingZeros* :: $('a::\text{len}) \text{ word} \Rightarrow \text{nat}$ **where**
numberOfTrailingZeros $v = \text{lowestOneBit } v$

lemma *lowestOneBit-bot*: $\text{lowestOneBit } (0::64 \text{ word}) = 64$
unfolding *lowestOneBit-def MinOrHighest-def*
by (*simp add: size64*)

lemma *bit-zero-set-in-top*: $\text{bit } (-1::'a::\text{len} \text{ word}) \ 0$
by *auto*

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

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

4.3 Long.bitCount

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

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

4.4 Long.zeroCount

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

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

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

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

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

lemma *card-of-range*:
 $x = \text{card } \{n . 0 \leq n \wedge n < x\}$
by *simp*

lemma *range-of-nat*:
 $\{(n::\text{nat}) . 0 \leq n \wedge n < x\} = \{n . n < x\}$
by *simp*

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

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

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

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

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

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

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

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

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

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

lemma *card-eq*:
assumes $\text{finite } x \wedge \text{finite } y \wedge \text{finite } z$
assumes $x \cup y = z$
assumes $y \cap x = \{\}$
shows $\text{card } z - \text{card } y = \text{card } x$
using *assms add-diff-cancel-right' card-Un-disjoint*
by (*metis inf commute*)

lemma *card-add*:

assumes *finite* $x \wedge$ *finite* $y \wedge$ *finite* z
assumes $x \cup y = z$
assumes $y \cap x = \{\}$
shows $\text{card } x + \text{card } y = \text{card } z$
using *assms card-Un-disjoint*
by (*metis inf.commute*)

lemma *card-add-inverses*:

assumes *finite* $\{n. Q\ n \wedge \neg(P\ n)\} \wedge$ *finite* $\{n. Q\ n \wedge P\ n\} \wedge$ *finite* $\{n. Q\ n\}$
shows $\text{card } \{n. Q\ n \wedge P\ n\} + \text{card } \{n. Q\ n \wedge \neg(P\ n)\} = \text{card } \{n. Q\ n\}$
apply (*rule card-add*)
using *assms* **apply** *simp*
apply *auto*[1]
by *auto*

lemma *ones-zero-sum-to-width*:

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

lemma *intersect-bitCount-helper*:

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

```

    using disjoint by simp
qed

```

```

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

```

```

hide-fact intersect-bitCount-helper

```

```

end

```

5 Data-flow Semantics

```

theory IRTreeEval
  imports
    Graph.Stamp
  begin

```

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

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

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

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

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

```

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

```

```

definition new-map-state :: MapState where
  new-map-state = (λx..UndefVal)

```

5.1 Data-flow Tree Representation

```

datatype IRUnaryOp =
  UnaryAbs
  | UnaryNeg

```

```

| UnaryNot
| UnaryLogicNegation
| UnaryNarrow (ir-inputBits: nat) (ir-resultBits: nat)
| UnarySignExtend (ir-inputBits: nat) (ir-resultBits: nat)
| UnaryZeroExtend (ir-inputBits: nat) (ir-resultBits: nat)

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

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

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

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

| ConstantExpr (ir-const: Value)
| ConstantVar (ir-name: string)
| VariableExpr (ir-name: string) (ir-stamp: Stamp)

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

typedef GroundExpr = { e :: IRExpr . is-ground e }
using is-ground.simps(6) by blast

```

5.2 Functions for re-calculating stamps

Note: in Java all integer calculations are done as 32 or 64 bit calculations. However, here we generalise the operators to allow any size calculations. Many operators have the same output bits as their inputs. However, the unary integer operators that are not *normal_unary* are narrowing or widening operators, so the result bits is specified by the operator. The binary integer operators are divided into three groups: (1) *binary_fixed_32* operators always output 32 bits, (2) *binary_shift_ops* operators output size is determined by their left argument, and (3) other operators output the same number of bits as both their inputs.

abbreviation *binary-fixed-32-ops* :: *IRBinaryOp* set **where**

binary-fixed-32-ops ≡ {*BinShortCircuitOr*, *BinIntegerEquals*, *BinIntegerLessThan*, *BinIntegerBelow*}

abbreviation *binary-shift-ops* :: *IRBinaryOp* set **where**

binary-shift-ops ≡ {*BinLeftShift*, *BinRightShift*, *BinURightShift*}

abbreviation *normal-unary* :: *IRUnaryOp* set **where**

normal-unary ≡ {*UnaryAbs*, *UnaryNeg*, *UnaryNot*, *UnaryLogicNegation*}

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

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

stamp-unary *op* - = *IllegalStamp*

fun *stamp-binary* :: *IRBinaryOp* ⇒ *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**

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

stamp-binary *op* - - = *IllegalStamp*

fun *stamp-expr* :: *IRExpr* ⇒ *Stamp* **where**

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

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

5.3 Data-flow Tree Evaluation

fun *unary-eval* :: *IRUnaryOp* \Rightarrow *Value* \Rightarrow *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* \Rightarrow *Value* \Rightarrow *Value* \Rightarrow *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* \Rightarrow *Value* \Rightarrow *bool* **where**
 $\llbracket \text{value} \neq \text{UndefVal} \rrbracket \Longrightarrow \text{not-undef-or-fail value value}$

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:

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

ParameterExpr:

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

ConditionalExpr:

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

UnaryExpr:

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

BinaryExpr:

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

LeafExpr:

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

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

inductive

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

EvalNil:

$[m,p] \vdash [] \mapsto_L [] \mid$

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

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

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

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

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

5.4 Data-flow Tree Refinement

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

definition *equiv-exprs* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* (*- \doteq -* 55) **where**
(e1 \doteq e2) = ($\forall m p v. (([m,p] \vdash e1 \mapsto v) \longleftrightarrow ([m,p] \vdash e2 \mapsto v))$)

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

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

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

instantiation *IRExpr* :: *preorder* **begin**

notation *less-eq* (**infix** \sqsubseteq 65)

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

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

```

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

end

abbreviation (output) Refines :: IRExpr  $\Rightarrow$  IRExpr  $\Rightarrow$  bool (infix  $\sqsupseteq$  64)
  where e1  $\sqsupseteq$  e2  $\equiv$  (e2  $\leq$  e1)

```

5.5 Stamp Masks

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

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

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

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

```

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

```

```

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

```

```

lemma not-may-implies-false:
  [m, p]  $\vdash$  e  $\mapsto$  IntVal b v  $\implies$   $\neg$ (bit ( $\uparrow$ e) n)  $\implies$  bit v n = False
  using up-spec
  using bit-and-iff bit-eq-iff bit-not-iff bit-unsigned-iff down-spec
  by (smt (verit, best) bit.double-compl)

```

```

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

```

lemma *not-must-implies-either*:

$[m, p] \vdash e \mapsto \text{IntVal } b \ v \implies \neg(\text{bit } (\downarrow e) \ n) \implies \text{bit } v \ n = \text{False} \vee \text{bit } v \ n = \text{True}$
by *simp*

lemma *must-implies-may*:

$[m, p] \vdash e \mapsto \text{IntVal } b \ v \implies n < 32 \implies \text{bit } (\downarrow e) \ n \implies \text{bit } (\uparrow e) \ n$
by (*meson must-implies-true not-may-implies-false*)

lemma *up-mask-and-zero-implies-zero*:

assumes *and* $(\uparrow x) (\uparrow y) = 0$
assumes $[m, p] \vdash x \mapsto \text{IntVal } b \ xv$
assumes $[m, p] \vdash y \mapsto \text{IntVal } b \ yv$
shows *and* $xv \ yv = 0$
using *assms*
by (*smt (z3) and.commute and.right-neutral and-zero-eq bit.compl-zero bit.conj-cancel-right bit.conj-disj-distrib(1) ucast-id up-spec word-bw-assocs(1) word-not-dist(2)*)

lemma *not-down-up-mask-and-zero-implies-zero*:

assumes *and* $(\text{not } (\downarrow x)) (\uparrow y) = 0$
assumes $[m, p] \vdash x \mapsto \text{IntVal } b \ xv$
assumes $[m, p] \vdash y \mapsto \text{IntVal } b \ yv$
shows *and* $xv \ yv = yv$
using *assms*
by (*smt (z3) and-zero-eq bit.conj-cancel-left bit.conj-disj-distrib(1) bit.conj-disj-distrib(2) bit.de-Morgan-disj down-spec or-eq-not-not-and ucast-id up-spec word-ao-absorbs(2) word-ao-absorbs(8) word-bw-lcs(1) word-not-dist(2)*)

end

definition *IRExpr-up* :: *IRExpr* \Rightarrow *int64* **where**

IRExpr-up *e* = *not 0*

definition *IRExpr-down* :: *IRExpr* \Rightarrow *int64* **where**

IRExpr-down *e* = *0*

lemma *ucast-zero*: $(\text{ucast } (0::\text{int64})::\text{int32}) = 0$

by *simp*

lemma *ucast-minus-one*: $(\text{ucast } (-1::\text{int64})::\text{int32}) = -1$

apply *transfer by auto*

interpretation *simple-mask*: *stamp-mask*

IRExpr-up :: *IRExpr* \Rightarrow *int64*

IRExpr-down :: *IRExpr* \Rightarrow *int64*

unfolding *IRExpr-up-def IRExpr-down-def*

apply *unfold-locales*

by (*simp add: ucast-minus-one*)**+**

end

5.6 Data-flow Tree Theorems

```
theory IRTreeEvalThms
  imports
    Graph.ValueThms
    IRTreeEval
begin
```

5.6.1 Deterministic Data-flow Evaluation

```
lemma evalDet:
   $[m,p] \vdash e \mapsto v_1 \implies$ 
   $[m,p] \vdash e \mapsto v_2 \implies$ 
   $v_1 = v_2$ 
  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
```

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

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

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

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

```

by meson

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

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

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

5.6.3 Evaluation Results are Valid

A valid value cannot be *UndefVal*.

```
lemma valid-not-undef:
```

```

assumes a1: valid-value val s
assumes a2: s ≠ VoidStamp
shows val ≠ UndefVal
apply (rule valid-value.elims(1)[of val s True])
using a1 a2 by auto

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

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

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

lemmas valid-value-elim =
  valid-VoidStamp
  valid-ObjStamp
  valid-int

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

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

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

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

```



```

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

```

5.6.4 Example Data-flow Optimisations

5.6.5 Monotonicity of Expression Refinement

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

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

```

lemma mono-unary:
  assumes  $e \geq e'$ 
  shows  $(UnaryExpr\ op\ e) \geq (UnaryExpr\ op\ e')$ 
  using UnaryExpr assms by auto

```

```

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

```

```

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

```

```

lemma compatible-trans:
  compatible x y  $\wedge$  compatible y z  $\implies$  compatible x z
  by (cases x; cases y; cases z; simp del: valid-stamp.simps)

```

```

lemma compatible-refl:

```

compatible $x\ y \implies \text{compatible}\ y\ x$
using *compatible.elims*(2) **by** *fastforce*

lemma *mono-conditional*:

assumes $ce \geq ce'$
assumes $te \geq te'$
assumes $fe \geq fe'$
shows $(\text{ConditionalExpr}\ ce\ te\ fe) \geq (\text{ConditionalExpr}\ ce'\ te'\ fe')$
proof (*simp only*: *le-expr-def*; (*rule allI*) $+$; *rule impI*)
fix $m\ p\ v$
assume a : $[m, p] \vdash \text{ConditionalExpr}\ ce\ te\ fe \mapsto v$
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 : $branch = (\text{if val-to-bool}\ cond\ \text{then}\ te\ \text{else}\ fe)$
define $branch'$ **where** b' : $branch' = (\text{if val-to-bool}\ cond\ \text{then}\ te'\ \text{else}\ fe')$
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 $[m, p] \vdash \text{ConditionalExpr}\ ce'\ te'\ fe' \mapsto v$
using *ConditionalExpr\ ce'\ b'*
using a **by** *blast*
qed

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

These rewrite rules can be useful when proving optimizations. They support top-down rewriting of each level of the tree into the lower-level *bin_eeval* / *unary_eeval* level, simply by saying *unfoldingunfold_eevaltree*.

lemma *unfold-const*:

shows $([m, p] \vdash \text{ConstantExpr}\ c \mapsto v) = (\text{wf-value}\ v \wedge v = c)$
by *blast*

lemma *unfold-binary*:

shows $([m, p] \vdash \text{BinaryExpr}\ op\ xe\ ye \mapsto val) = (\exists\ x\ y.$
 $(([m, p] \vdash xe \mapsto x) \wedge$
 $([m, p] \vdash ye \mapsto y) \wedge$
 $(val = \text{bin-eval}\ op\ x\ y) \wedge$
 $(val \neq \text{UndefVal})$
 $))\ (\text{is}\ ?L = ?R)$
proof (*intro iffI*)
assume \exists : $?L$
show $?R$ **by** (*rule evaltree.cases*[*OF* \exists]; *blast* $+$)

```

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

```

```

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

```

```

lemmas unfold-evaltree =
  unfold-binary
  unfold-unary

```

5.8 Lemmas about *new_int* and integer eval results.

```

lemma unary-eval-new-int:
  assumes def: unary-eval op x ≠ UndefVal
  shows ∃ b v. unary-eval op x = new-int b v ∧
    b = (if op ∈ normal-unary then intval-bits x else ir-resultBits op)
proof (cases op ∈ normal-unary)
  case True
  then show ?thesis
    by (metis def empty-iff insert-iff intval-abs.elims intval-bits.simps intval-logic-negation.elims
      intval-negate.elims intval-not.elims unary-eval.simps(1) unary-eval.simps(2) unary-eval.simps(3)
      unary-eval.simps(4))
  next
  case False
  consider ib ob where op = UnaryNarrow ib ob |
    ib ob where op = UnaryZeroExtend ib ob |
    ib ob where op = UnarySignExtend ib ob
  by (metis False IRUnaryOp.exhaust insert-iff)
  then show ?thesis
proof (cases)
  case 1
  then show ?thesis
    by (metis False IRUnaryOp.sel(4) def intval-narrow.elims unary-eval.simps(5))

```

```

next
  case 2
  then show ?thesis
  by (metis False IRUnaryOp.sel(6) def intval-zero-extend.elims unary-eval.simps(7))
next
  case 3
  then show ?thesis
  by (metis False IRUnaryOp.sel(5) def intval-sign-extend.elims unary-eval.simps(6))
qed
qed

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

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

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

lemma eval-unused-bits-zero:
  [m,p] ⊢ xe ↦ (IntVal b ix) ⇒ take-bit b ix = ix
proof (induction xe)
  case (UnaryExpr x1 xe)
  then show ?case
  using unary-eval-unused-bits-zero by force
next
  case (BinaryExpr x1 xe1 xe2)
  then show ?case
  using bin-eval-unused-bits-zero by force
next
  case (ConditionalExpr xe1 xe2 xe3)
  then show ?case
  by (metis (full-types) EvalTreeE(3))
next
  case (ParameterExpr i s)
  then have valid-value (p!i) s
  by fastforce
  then show ?case
  by (metis ParameterExprE Value.distinct(7) intval-bits.simps intval-word.simps
  local.ParameterExpr valid-value.elims(2))

```

```

next
  case (LeafExpr x1 x2)
  then show ?case
  by (smt (z3) EvalTreeE(6) Value.simps(11) valid-value.elims(1) valid-value.simps(1))

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

```

```

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

```

```

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

```

```

lemma unary-eval-bitsize:

```

```

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

```

```

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

```

```

lemma eval-bits-1-64:
  [m,p] ⊢ xe ↦ (IntVal b ix) ⇒ 0 < b ∧ b ≤ 64
proof (induction xe arbitrary: b ix)
  case (UnaryExpr op x2)
  then obtain xv where
    xv: ([m,p] ⊢ x2 ↦ xv) ∧
      IntVal b ix = unary-eval op xv
    using unfold-binary by auto
  then have b = (if op ∈ normal-unary then intval-bits xv else ir-resultBits op)
    using unary-eval-new-int
    by (metis Value.disc(1) Value.discI(1) Value.sel(1) new-int.simps)

```

```

    then show ?case
      by (metis xv UnaryExpr.IH unary-normal-bitsize unary-not-normal-bitsize)
next
  case (BinaryExpr op x y)
  then obtain xv yv where
    xy: ([m,p] ⊢ x ↦ xv) ∧
        ([m,p] ⊢ y ↦ yv) ∧
        IntVal b ix = bin-eval op xv yv
    using unfold-binary by auto
  then have def: bin-eval op xv yv ≠ UndefVal and xv: xv ≠ UndefVal and yv ≠
  UndefVal
    using evaltree-not-undef xy by (force, blast, blast)
  then have b = (if op ∈ binary-fixed-32-ops then 32 else intval-bits xv)
    by (metis xy intval-bits.simps new-int.simps bin-eval-new-int)
  then show ?case
    by (metis BinaryExpr.IH(1) Value.distinct(7) Value.distinct(9) xv bin-eval-inputs-are-ints
    intval-bits.elims le-add-same-cancel1 less-or-eq-imp-le numeral-Bit0 xy zero-less-numeral)
next
  case (ConditionalExpr xe1 xe2 xe3)
  then show ?case
    by (metis (full-types) EvalTreeE(3))
next
  case (ParameterExpr x1 x2)
  then show ?case
    using ParameterExprE intval-bits.simps valid-stamp.simps(1) valid-value.elims(2)
    valid-value.simps(17)
    by (metis (no-types, lifting))
next
  case (LeafExpr x1 x2)
  then show ?case
    by (smt (z3) EvalTreeE(6) Value.distinct(7) Value.inject(1) valid-stamp.simps(1)
    valid-value.elims(1))
next
  case (ConstantExpr x)
  then show ?case using wf-value-def
    by (metis EvalTreeE(1) constantAsStamp.simps(1) valid-stamp.simps(1) valid-value.simps(1))
next
  case (ConstantVar x)
  then show ?case
    by blast
next
  case (VariableExpr x1 x2)
  then show ?case
    by blast
qed

```

lemma *unfold-binary-width:*
 assumes $op \notin \text{binary-fixed-32-ops} \wedge op \notin \text{binary-shift-ops}$

```

shows ( $[m,p] \vdash \text{BinaryExpr } op \ xe \ ye \mapsto \text{IntVal } b \ val$ ) = ( $\exists \ x \ y.$ 
  ( $[m,p] \vdash xe \mapsto \text{IntVal } b \ x$ )  $\wedge$ 
  ( $[m,p] \vdash ye \mapsto \text{IntVal } b \ y$ )  $\wedge$ 
  ( $\text{IntVal } b \ val = \text{bin-eval } op \ (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)$ )  $\wedge$ 
  ( $\text{IntVal } b \ val \neq \text{UndefVal}$ )
  )) (is ?L = ?R)
proof (intro iffI)
  assume  $\mathcal{B}$ : ?L
  show ?R apply (rule evaltree.cases[OF  $\mathcal{B}$ ])
    apply force+ apply auto[1]
  using assms apply (cases op; auto)
    apply (smt (verit) intval-add.elims Value.inject(1))
  using intval-mul.elims Value.inject(1)
    apply (smt (verit) new-int.simps new-int-bin.simps)
  using intval-sub.elims Value.inject(1)
    apply (smt (verit) new-int.simps new-int-bin.simps)
  using intval-and.elims Value.inject(1)
    apply (smt (verit) new-int.simps new-int-bin.simps take-bit-and)
  using intval-or.elims Value.inject(1)
    apply (smt (verit) new-int.simps new-int-bin.simps take-bit-or)
  using intval-xor.elims Value.inject(1)
    apply (smt (verit) new-int.simps new-int-bin.simps take-bit-xor)
  by blast

next
  assume  $R$ : ?R
  then obtain  $x \ y$  where  $[m,p] \vdash xe \mapsto \text{IntVal } b \ x$ 
    and  $[m,p] \vdash ye \mapsto \text{IntVal } b \ y$ 
    and  $\text{new-int } b \ val = \text{bin-eval } op \ (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)$ 
    and  $\text{new-int } b \ val \neq \text{UndefVal}$ 
  using bin-eval-unused-bits-zero by force
  then show ?L
    using  $R$  by blast
qed

end

```

6 Tree to Graph

```

theory TreeToGraph
  imports
    Semantics.IRTreeEval
    Graph.IRGraph
begin

```

6.1 Subgraph to Data-flow Tree

```

fun find-node-and-stamp :: IRGraph  $\Rightarrow$  (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:

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

ParameterNode:

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

ConditionalNode:

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

AbsNode:

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

NotNode:

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

NegateNode:

$\llbracket kind\ g\ n = NegateNode\ x; \rrbracket$

$$g \vdash x \simeq xe \parallel \\ \implies g \vdash n \simeq (\text{UnaryExpr } \text{UnaryNeg } xe) \mid$$

LogicNegationNode:

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

AddNode:

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

MulNode:

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

SubNode:

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

AndNode:

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

OrNode:

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

XorNode:

$$\llbracket \text{kind } g \ n = \text{XorNode } x \ y; \\ g \vdash x \simeq xe; \\ g \vdash y \simeq ye \parallel \\ \implies g \vdash n \simeq (\text{BinaryExpr } \text{BinXor } xe \ ye) \mid$$

ShortCircuitOrNode:

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

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

LeftShiftNode:

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{LeftShiftNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr BinLeftShift } xe \ ye) \mid \end{aligned}$$

RightShiftNode:

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{RightShiftNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr BinRightShift } xe \ ye) \mid \end{aligned}$$

UnsignedRightShiftNode:

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{UnsignedRightShiftNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr BinURightShift } xe \ ye) \mid \end{aligned}$$

IntegerBelowNode:

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{IntegerBelowNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr BinIntegerBelow } xe \ ye) \mid \end{aligned}$$

IntegerEqualsNode:

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{IntegerEqualsNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr BinIntegerEquals } xe \ ye) \mid \end{aligned}$$

IntegerLessThanNode:

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{IntegerLessThanNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr BinIntegerLessThan } xe \ ye) \mid \end{aligned}$$

NarrowNode:

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{NarrowNode } \text{inputBits } \text{resultBits } x; \\ & \quad g \vdash x \simeq xe \rrbracket \\ & \implies g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryNarrow } \text{inputBits } \text{resultBits}) \ xe) \mid \end{aligned}$$

SignExtendNode:

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{SignExtendNode } \text{inputBits } \text{resultBits } x; \\ & \quad g \vdash x \simeq xe \rrbracket \\ & \implies g \vdash n \simeq (\text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ xe) \mid \end{aligned}$$

ZeroExtendNode:

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

LeafNode:

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

RefNode:

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

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:

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

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. eg2-sq* \vdash 4 \simeq *t*}

6.2 Data-flow Tree to Subgraph

fun *unary-node* :: *IRUnaryOp* \Rightarrow *ID* \Rightarrow *IRNode* **where**

unary-node *UnaryAbs* *v* = *AbsNode* *v* \mid
unary-node *UnaryNot* *v* = *NotNode* *v* \mid
unary-node *UnaryNeg* *v* = *NegateNode* *v* \mid
unary-node *UnaryLogicNegation* *v* = *LogicNegationNode* *v* \mid
unary-node (*UnaryNarrow* *ib rb*) *v* = *NarrowNode* *ib rb v* \mid
unary-node (*UnarySignExtend* *ib rb*) *v* = *SignExtendNode* *ib rb v* \mid

unary-node (*UnaryZeroExtend ib rb*) *v* = *ZeroExtendNode ib rb v*

fun *bin-node* :: *IRBinaryOp* \Rightarrow *ID* \Rightarrow *ID* \Rightarrow *IRNode* **where**
bin-node BinAdd *x y* = *AddNode x y* |
bin-node BinMul *x y* = *MulNode x y* |
bin-node BinSub *x y* = *SubNode x y* |
bin-node BinAnd *x y* = *AndNode x y* |
bin-node BinOr *x y* = *OrNode x y* |
bin-node BinXor *x y* = *XorNode x y* |
bin-node BinShortCircuitOr *x y* = *ShortCircuitOrNode x y* |
bin-node BinLeftShift *x y* = *LeftShiftNode x y* |
bin-node BinRightShift *x y* = *RightShiftNode x y* |
bin-node BinURightShift *x y* = *UnsignedRightShiftNode x y* |
bin-node BinIntegerEquals *x y* = *IntegerEqualsNode x y* |
bin-node BinIntegerLessThan *x y* = *IntegerLessThanNode x y* |
bin-node BinIntegerBelow *x y* = *IntegerBelowNode x y*

inductive *fresh-id* :: *IRGraph* \Rightarrow *ID* \Rightarrow *bool* **where**
n \notin *ids g* \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:

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

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:

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

ParameterNodeNew:

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

ConditionalNodeSame:

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

ConditionalNodeNew:

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

UnaryNodeSame:

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

UnaryNodeNew:

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

BinaryNodeSame:

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

$\implies g \oplus (\text{BinaryExpr } op \ x \ ye) \rightsquigarrow (g3, n) \mid$

BinaryNodeNew:

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

AllLeafNodes:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.3 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* = $(\exists e. (g \vdash n \simeq e) \wedge ([m, p] \vdash e \mapsto v))$

6.4 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*)

6.5 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

6.6 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

6.7 Dynamic Frames

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

theory *IRGraphFrames*

imports

Form

begin

fun *unchanged* :: *ID set* \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
   $\langle$ kind g2 n = ConditionalNode c t f $\rangle$  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:  $[g1, m, p] \vdash \text{id} \mapsto v1$ 
  assumes wf: wf-graph g1
  shows  $[g2, m, p] \vdash \text{id} \mapsto v1$ 
proof –
  have nid:  $\text{id} \in \text{ids } g1$ 
    using g1 eval-in-ids by simp
  then have  $\text{id} \in \text{eval-usages } g1 \text{ nid}$ 
    using eval-usages-self by blast
  then have kind-same:  $\text{kind } g1 \text{ nid} = \text{kind } g2 \text{ nid}$ 
    using nc node-unchanged by blast
  obtain e where  $e: (g1 \vdash \text{id} \simeq e) \wedge ([m, p] \vdash e \mapsto v1)$ 
    using encodeeval-def g1
    by auto
  then have val:  $[m, p] \vdash e \mapsto v1$ 
    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  $g2 \vdash \text{id} \simeq \text{ParameterExpr } i \text{ s}$ 
      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  $g2 \vdash \text{id} \simeq \text{ConditionalExpr } ce \text{ te } fe$ 
      using ConditionalExpr.prem1 ConditionalExpr.prem3 local.wf stay-same-encoding
      by presburger
    then show ?case
      by (meson ConditionalExpr.prem1 ConditionalExpr.prem3 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

```

6.8 Tree to Graph Theorems

```

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

```

6.8.1 Extraction and Evaluation of Expression Trees is Deterministic.

First, we prove some extra rules that relate each type of IRNode to the corresponding 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*]:
 $g \vdash n \simeq e \implies$
 $kind\ g\ n = SubNode\ x\ y \implies$
 $(\exists\ xe\ ye. e = BinaryExpr\ BinSub\ xe\ ye)$
by (*induction rule: rep.induct; auto*)

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

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

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

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

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

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

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

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

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

```

6.8.2 Monotonicity of Graph Refinement

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

```

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

```

```

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

```

```

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

```

```

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

```

shows $e1 \geq e2$
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⟩)

```

6.8.3 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:  $\text{kind } g1 \ n = \text{NegateNode } xn$ 
    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:  $\text{kind } g1 \ n = \text{LogicNegationNode } xn$ 
    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.prem1 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  $mx$ :  $g1 \vdash xn \simeq xe1$ 
  using  $\text{RightShiftNode.hyps}(1)\ \text{RightShiftNode.hyps}(2)$  by fastforce
from  $l$  have  $my$ :  $g1 \vdash yn \simeq ye1$ 
  using  $\text{RightShiftNode.hyps}(1)\ \text{RightShiftNode.hyps}(3)$  by fastforce
then show ?case
proof –
  have  $g1 \vdash xn \simeq xe1$  using  $mx$  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  $\text{RightShiftNode}$ 
    using  $a\ b\ c\ d\ l\ \text{no-encoding not-excluded-keep-type repDet singletonD}$ 
    by  $(\text{metis-node-eq-binary } \text{RightShiftNode})$ 
  have  $\exists\ ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
    using  $\text{RightShiftNode } a\ b\ c\ d\ l\ \text{no-encoding not-excluded-keep-type repDet}$ 
    singletonD
    by  $(\text{metis-node-eq-binary } \text{RightShiftNode})$ 
  then have  $\exists\ xe2\ ye2. (g2 \vdash n \simeq \text{BinaryExpr BinRightShift } xe2\ ye2) \wedge$ 
BinaryExpr BinRightShift  $xe1\ ye1 \geq \text{BinaryExpr BinRightShift } xe2\ ye2$ 
    by  $(\text{metis } \text{RightShiftNode.premis } l\ \text{mono-binary rep.RightShiftNode } xer)$ 
  then show ?thesis
    by meson
  qed
next
  case  $(\text{UnsignedRightShiftNode } n\ x\ y\ xe1\ ye1)$ 
  have  $k$ :  $g1 \vdash n \simeq \text{BinaryExpr BinURightShift } xe1\ ye1$  using  $f\ \text{UnsignedRightShiftNode}$ 
    by  $(\text{simp add: UnsignedRightShiftNode.hyps}(2)\ \text{rep.UnsignedRightShiftNode})$ 
  obtain  $xn\ yn$  where  $l$ :  $\text{kind } g1\ n = \text{UnsignedRightShiftNode } xn\ yn$ 
    using  $\text{UnsignedRightShiftNode.hyps}(1)$  by blast
  then have  $mx$ :  $g1 \vdash xn \simeq xe1$ 
    using  $\text{UnsignedRightShiftNode.hyps}(1)\ \text{UnsignedRightShiftNode.hyps}(2)$  by
fastforce
  from  $l$  have  $my$ :  $g1 \vdash yn \simeq ye1$ 
    using  $\text{UnsignedRightShiftNode.hyps}(1)\ \text{UnsignedRightShiftNode.hyps}(3)$  by
fastforce
  then show ?case
  proof –
    have  $g1 \vdash xn \simeq xe1$  using  $mx$  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  $\text{UnsignedRightShiftNode}$ 
      using  $a\ b\ c\ d\ l\ \text{no-encoding not-excluded-keep-type repDet singletonD}$ 
      by  $(\text{metis-node-eq-binary } \text{UnsignedRightShiftNode})$ 
    have  $\exists\ ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
      using  $\text{UnsignedRightShiftNode } a\ b\ c\ d\ l\ \text{no-encoding not-excluded-keep-type}$ 
      repDet singletonD
      by  $(\text{metis-node-eq-binary } \text{UnsignedRightShiftNode})$ 

```

```

      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  $\exists xe2\ ye2. (g2 \vdash n \simeq \text{BinaryExpr BinIntegerLessThan } xe2\ ye2) \wedge$ 
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 = 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 = 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 $[[\text{simp-trace}]]$

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)

```

6.8.4 Term Graph Reconstruction

```

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

```

```

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

```

```

lemma find-new-kind:

```

```

assumes  $g' = \text{add-node } \text{nid} \ (\text{node}, s) \ g$ 
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)

```

6.8.5 Data-flow Tree to Subgraph Preserves Maximal Sharing

```

lemma same-kind-stamp-encodes-equal:
  assumes kind g n = kind g n'
  assumes stamp g n = stamp g n'
  assumes  $\neg(\text{is-preevaluated } (\text{kind } g \ n))$ 
  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  $n = \text{get-fresh-id } g$ 
  assumes  $g' = \text{add-node } n \text{ (node, s) } g$ 
  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  $g = \{n \in \text{ids } g. \neg(\text{is-RefNode } (\text{kind } g \ n)) \wedge ((\text{kind } g \ n) \neq \text{NoNode})\}$ 
  unfolding true-ids-def ids-def
  using ids-def is-RefNode-def by fastforce

```

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

```

  assumes  $k \neq \text{NoNode}$ 
  assumes  $g' = \text{add-node } \text{nid} \ (k, s) \ g$ 
  shows  $g' = \text{Abs-IRGraph } ((\text{Rep-IRGraph } g)(\text{nid} \mapsto (k, s)))$ 
  using assms
  by (metis Rep-IRGraph-inverse add-node.rep-eq fst-conv)

```

lemma *ids-add-update-v1*:

```

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

```

lemma *ids-add-update-v2*:

```

  assumes  $g' = \text{add-node } \text{nid} \ (k, s) \ g$ 
  assumes  $k \neq \text{NoNode}$ 
  shows  $\text{nid} \in \text{ids } g'$ 
  using assms
  using find-new-kind ids-some by presburger

```

lemma *add-node-ids-subset*:

```

  assumes  $n \in \text{ids } g$ 
  assumes  $g' = \text{add-node } n \ \text{node } g$ 
  shows  $\text{ids } g' = \text{ids } g \cup \{n\}$ 
  using assms unfolding add-node-def
  apply (cases fst node = NoNode)
  using ids.rep-eq replace-node.rep-eq replace-node-def apply auto[1]

```

unfolding *ids-def*
by (*smt* (*verit*, *best*) *Collect-cong Un-insert-right dom-fun-upd fst-conv fun-upd-apply*
ids.rep-eq ids-def insert-absorb mem-Collect-eq option.inject option.simps(3) 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  $x \ x_e$ )
  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 = (*IntVal* 32 0)

definition *bad-graph* **where**

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

end

7 Control-flow Semantics

theory *IRStepObj*

imports

TreeToGraph

begin

7.1 Object Heap

The heap model we introduce maps field references to object instances to runtime values. We use the $H[f][p]$ heap representation. See \cite{heap-reps-2011}. We also introduce the *DynamicHeap* type which allocates new object references sequentially storing the next free object reference as 'Free'.

heapdef

```

type-synonym ('a, 'b) Heap = 'a  $\Rightarrow$  '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. UndefVal), 0)

```

7.2 Intraprocedural Semantics

```

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

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

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

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

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

```

Intraprocedural semantics are given as a small-step semantics.

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

```

inductive step :: IRGraph  $\Rightarrow$  Params  $\Rightarrow$  (ID  $\times$  MapState  $\times$  FieldRefHeap)  $\Rightarrow$  (ID

```


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

SequentialNode:

$\llbracket \text{is-sequential-node } (kind \ g \ nid);$
 $\quad nid' = (\text{successors-of } (kind \ g \ nid))!0 \rrbracket$
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \mid$

IfNode:

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

EndNodes:

$\llbracket \text{is-AbstractEndNode } (kind \ g \ nid);$
 $\quad merge = \text{any-usage } g \ nid;$
 $\quad \text{is-AbstractMergeNode } (kind \ g \ merge);$

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

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

NewInstanceNode:

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

LoadFieldNode:

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

SignedDivNode:

$\llbracket kind \ g \ nid = (\text{SignedDivNode } nid \ x \ y \ zero \ sb \ nxt);$
 $\quad g \vdash x \simeq xe;$
 $\quad g \vdash y \simeq ye;$
 $\quad [m, p] \vdash xe \mapsto v1;$

$$\begin{aligned}
& [m, p] \vdash ye \mapsto v2; \\
& v = (\text{intval-div } v1 \ v2); \\
& m' = m(\text{nid} := v) \\
\implies & g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nxt}, m', h) \mid
\end{aligned}$$

SignedRemNode:

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

StaticLoadFieldNode:

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

StoreFieldNode:

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

StaticStoreFieldNode:

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

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

7.3 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:

$\llbracket g, p \vdash (nid, m, h) \rightarrow (nid', m', h') \rrbracket$
 $\implies P \vdash ((g, nid, m, p) \# stk, h) \longrightarrow ((g, nid', m', p) \# stk, h') \mid$

InvokeNodeStep:

$\llbracket is-Invoke \ (kind \ g \ nid);$

$callTarget = ir-callTarget \ (kind \ g \ nid);$

$kind \ g \ callTarget = (MethodCallTargetNode \ targetMethod \ arguments);$

$Some \ targetGraph = P \ targetMethod;$

$m' = new-map-state;$

$g \vdash arguments \simeq_L argsE;$

$[m, p] \vdash argsE \mapsto_L p \rrbracket$

$\implies P \vdash ((g, nid, m, p) \# stk, h) \longrightarrow ((targetGraph, 0, m', p') \# (g, nid, m, p) \# stk, h)$

\mid

ReturnNode:

$\llbracket kind \ g \ nid = (ReturnNode \ (Some \ expr) \ -);$

$g \vdash expr \simeq e;$

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

$cm' = cm(cnid := v);$

$cnid' = (successors-of \ (kind \ cg \ cnid))!0 \rrbracket$

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

ReturnNodeVoid:

$\llbracket kind \ g \ nid = (ReturnNode \ None \ -);$

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

$cnid' = (successors-of \ (kind \ cg \ cnid))!0 \rrbracket$

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

UnwindNode:

$\llbracket kind \ g \ nid = (UnwindNode \ exception);$

$g \vdash exception \simeq exceptionE;$

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

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

$cm' = cm(cnid := e) \rrbracket$

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

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

7.4 Big-step Execution

type-synonym $Trace = (IRGraph \times ID \times MapState \times Params) \text{ list}$

fun $has\text{-}return :: MapState \Rightarrow bool$ **where**
 $has\text{-}return\ m = (m\ 0 \neq Undefined)$

inductive $exec :: Program$
 $\Rightarrow (IRGraph \times ID \times MapState \times Params) \text{ list} \times FieldRefHeap$
 $\Rightarrow Trace$
 $\Rightarrow (IRGraph \times ID \times MapState \times Params) \text{ list} \times FieldRefHeap$
 $\Rightarrow Trace$
 $\Rightarrow bool$
 $(- \vdash - \mid - \longrightarrow * - \mid -)$
for P
where
 $\llbracket P \vdash (((g, nid, m, p) \# xs), h) \longrightarrow (((g', nid', m', p') \# ys), h') ;$
 $\neg(has\text{-}return\ m') ;$
 $l' = (l @ [(g, nid, m, p)]) ;$
 $exec\ P\ (((g', nid', m', p') \# ys), h')\ l'\ next\text{-}state\ l'' \rrbracket$
 $\implies exec\ P\ (((g, nid, m, p) \# xs), h)\ l\ next\text{-}state\ l''$
 \mid
 $\llbracket P \vdash (((g, nid, m, p) \# xs), h) \longrightarrow (((g', nid', m', p') \# ys), h') ;$
 $has\text{-}return\ m' ;$
 $l' = (l @ [(g, nid, m, p)]) \rrbracket$
 $\implies exec\ P\ (((g, nid, m, p) \# xs), h)\ l\ (((g', nid', m', p') \# ys), h')\ l'$
code-pred $(modes: i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool \text{ as } Exec)\ exec .$

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

7.4.1 Heap Testing

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

$p3 = [IntVal\ 32\ 3]$

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

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

$field-sq = "sq"$

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

$eg3-sq = irgraph\ [$
 $(0, StartNode\ None\ 4, VoidStamp),$
 $(1, ParameterNode\ 0, default-stamp),$
 $(3, MulNode\ 1\ 1, default-stamp),$
 $(4, StoreFieldNode\ 4\ field-sq\ 3\ None\ None\ 5, VoidStamp),$
 $(5, ReturnNode\ (Some\ 3)\ None, default-stamp)$
 $]$

values $\{h-load-field\ field-sq\ None\ (prod.snd\ res)$
 $\mid res. (\lambda x. Some\ eg3-sq) \vdash [(eg3-sq, 0, new-map-state, p3), (eg3-sq, 0,$
 $new-map-state, p3)], new-heap) \rightarrow^* 3^* res\}$

definition $eg4-sq :: IRGraph$ **where**

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

values $\{h-load-field\ field-sq\ (Some\ 0)\ (prod.snd\ res) \mid res.$
 $(\lambda x. Some\ eg4-sq) \vdash [(eg4-sq, 0, new-map-state, p3), (eg4-sq, 0,$
 $new-map-state, p3)], new-heap) \rightarrow^* 3^* res\}$

end

7.5 Control-flow Semantics Theorems

theory $IRStepThms$

imports

$IRStepObj$

begin

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

7.5.1 Control-flow Step is Deterministic

theorem *stepDet*:

$(g, p \vdash (nid, m, h) \rightarrow next) \implies$
 $(\forall next'. ((g, p \vdash (nid, m, h) \rightarrow next') \longrightarrow next = next'))$

proof (*induction rule: step.induct*)

case (*SequentialNode* *nid* *next* *m* *h*)

have *notif*: $\neg(is_IfNode\ (kind\ g\ nid))$

using *SequentialNode.hyps*(1) *is-sequential-node.simps*

by (*metis is-IfNode-def*)

have *notend*: $\neg(is_AbstractEndNode\ (kind\ g\ nid))$

using *SequentialNode.hyps*(1) *is-sequential-node.simps*

by (*metis is-AbstractEndNode.simps is-EndNode.elims*(2) *is-LoopEndNode-def*)

have *notnew*: $\neg(is_NewInstanceNode\ (kind\ g\ nid))$

using *SequentialNode.hyps*(1) *is-sequential-node.simps*

by (*metis is-NewInstanceNode-def*)

have *notload*: $\neg(is_LoadFieldNode\ (kind\ g\ nid))$

using *SequentialNode.hyps*(1) *is-sequential-node.simps*

by (*metis is-LoadFieldNode-def*)

have *notstore*: $\neg(is_StoreFieldNode\ (kind\ g\ nid))$

using *SequentialNode.hyps*(1) *is-sequential-node.simps*

by (*metis is-StoreFieldNode-def*)

have *notdivrem*: $\neg(is_IntegerDivRemNode\ (kind\ g\ nid))$

using *SequentialNode.hyps*(1) *is-sequential-node.simps is-SignedDivNode-def*

is-SignedRemNode-def

by (*metis is-IntegerDivRemNode.simps*)

from *notif notend notnew notload notstore notdivrem*

show ?*case* **using** *SequentialNode step.cases*

by (*smt* (z3) *IRNode.disc*(1028) *IRNode.disc*(2270) *IRNode.discI*(31) *Pair-inject*

is-sequential-node.simps(18) *is-sequential-node.simps*(43) *is-sequential-node.simps*(44))

next

case (*IfNode* *nid* *cond* *tb* *fb* *m* *val* *next* *h*)

then have *notseq*: $\neg(is_sequential-node\ (kind\ g\ nid))$

using *is-sequential-node.simps is-AbstractMergeNode.simps*

by (*simp add: IfNode.hyps*(1))

have *notend*: $\neg(is_AbstractEndNode\ (kind\ g\ nid))$

using *is-AbstractEndNode.simps*

by (*simp add: IfNode.hyps*(1))

have *notdivrem*: $\neg(is_IntegerDivRemNode\ (kind\ g\ nid))$

using *is-AbstractEndNode.simps*

by (*simp add: IfNode.hyps*(1))

from *notseq notend notdivrem* **show** ?*case* **using** *IfNode repDet evalDet IRN-*

```

ode.distinct IRNode.inject(11) Pair-inject step.simps
  by (smt (z3) IRNode.distinct IRNode.inject(12) Pair-inject step.simps)
next
case (EndNodes nid merge i phis inputs m vs m' h)
have notseq: ¬(is-sequential-node (kind g nid))
  using EndNodes.hyps(1) is-AbstractEndNode.simps is-sequential-node.simps
  by (metis is-EndNode.elims(2) is-LoopEndNode-def)
have notif: ¬(is-IfNode (kind g nid))
  using EndNodes.hyps(1) is-IfNode-def is-AbstractEndNode.elims
  by (metis IRNode.distinct-disc(1058) is-EndNode.simps(12))
have notref: ¬(is-RefNode (kind g nid))
  using EndNodes.hyps(1) is-sequential-node.simps
  using IRNode.disc(1899) IRNode.distinct(1473) is-AbstractEndNode.simps
is-EndNode.elims(2) is-LoopEndNode-def is-RefNode-def
  by metis
have notnew: ¬(is-NewInstanceNode (kind g nid))
  using EndNodes.hyps(1) is-AbstractEndNode.simps
  using IRNode.distinct-disc(1442) is-EndNode.simps(29) is-NewInstanceNode-def
  by (metis IRNode.distinct-disc(1901) is-EndNode.simps(32))
have notload: ¬(is-LoadFieldNode (kind g nid))
  using EndNodes.hyps(1) is-AbstractEndNode.simps
  using is-LoadFieldNode-def
  by (metis IRNode.distinct-disc(1706) is-EndNode.simps(21))
have notstore: ¬(is-StoreFieldNode (kind g nid))
  using EndNodes.hyps(1) is-AbstractEndNode.simps is-StoreFieldNode-def
  by (metis IRNode.distinct-disc(1926) is-EndNode.simps(44))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  using EndNodes.hyps(1) is-AbstractEndNode.simps is-SignedDivNode-def is-SignedRemNode-def
  using IRNode.distinct-disc(1498) IRNode.distinct-disc(1500) is-IntegerDivRemNode.simps
is-EndNode.simps(36) is-EndNode.simps(37)
  by auto
from notseq notif notref notnew notload notstore notdivrem
show ?case using EndNodes repAllDet evalAllDet
  by (smt (z3) is-IfNode-def is-LoadFieldNode-def is-NewInstanceNode-def is-RefNode-def
is-StoreFieldNode-def is-SignedDivNode-def is-SignedRemNode-def Pair-inject is-IntegerDivRemNode.elims(3)
step.cases)
next
case (NewInstanceNode nid f obj nxt h' ref h m' m)
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notif: ¬(is-IfNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notref: ¬(is-RefNode (kind g nid))
  using is-AbstractMergeNode.simps

```

```

    by (simp add: NewInstanceNode.hyps(1))
  have notload:  $\neg(\text{is-LoadFieldNode } (\text{kind } g \text{ nid}))$ 
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notstore:  $\neg(\text{is-StoreFieldNode } (\text{kind } g \text{ nid}))$ 
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notdivrem:  $\neg(\text{is-IntegerDivRemNode } (\text{kind } g \text{ nid}))$ 
    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 nrt m ref h v m')
  then have notseq:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  have notend:  $\neg(\text{is-AbstractEndNode } (\text{kind } g \text{ nid}))$ 
    using is-AbstractEndNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  have notdivrem:  $\neg(\text{is-IntegerDivRemNode } (\text{kind } g \text{ nid}))$ 
    using is-AbstractEndNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using LoadFieldNode.step.cases repDet evalDet
    by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject Value.inject(2)
option.distinct(1) option.inject)
next
  case (StaticLoadFieldNode nid f nrt h v m' m)
  then have notseq:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notend:  $\neg(\text{is-AbstractEndNode } (\text{kind } g \text{ nid}))$ 
    using is-AbstractEndNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notdivrem:  $\neg(\text{is-IntegerDivRemNode } (\text{kind } g \text{ nid}))$ 
    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 nrt m val ref h' h m')
  then have notseq:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StoreFieldNode.hyps(1))

```



```

have notend:  $\neg(\text{is-AbstractEndNode } (\text{kind } g \text{ nid}))$ 
  using is-AbstractEndNode.simps
  by (simp add: StoreFieldNode.hyps(1))
have notdivrem:  $\neg(\text{is-IntegerDivRemNode } (\text{kind } g \text{ nid}))$ 
  by (simp add: StoreFieldNode.hyps(1))
from notseq notend notdivrem
show ?case using StoreFieldNode.step.cases repDet evalDet
  by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Value.inject(2)
option.distinct(1) option.inject)
next
  case (StaticStoreFieldNode nid f newval uv nxt m val h' h m')
  then have notseq:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
  have notend:  $\neg(\text{is-AbstractEndNode } (\text{kind } g \text{ nid}))$ 
    using is-AbstractEndNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
  have notdivrem:  $\neg(\text{is-IntegerDivRemNode } (\text{kind } g \text{ nid}))$ 
    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 $g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m, h)$

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 $g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m', h')$

shows $\text{nid} \in \text{ids } g$

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

8 Proof Infrastructure

8.1 Bisimulation

theory *Bisimulation*

imports

Stuttering
begin

inductive *weak-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
 (· · · ~ ·) **for** *nid* **where**
 $\llbracket \forall P'. (g \ m \ p \ h \vdash \text{nid} \rightsquigarrow P') \longrightarrow (\exists Q'. (g' \ m \ p \ h \vdash \text{nid} \rightsquigarrow Q') \wedge P' = Q');$
 $\forall Q'. (g' \ m \ p \ h \vdash \text{nid} \rightsquigarrow Q') \longrightarrow (\exists P'. (g \ m \ p \ h \vdash \text{nid} \rightsquigarrow P') \wedge P' = Q') \rrbracket$
 $\implies \text{nid} \cdot g \sim g'$

A strong bisimilution between no-op transitions

inductive *strong-noop-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
 (· | · ~ ·) **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-bisimilulation*:
assumes $g' = \text{replace-node } \text{nid} \ \text{node } g$
assumes $g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m, h)$
assumes $g', p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m, h)$
shows $\text{nid} \mid g \sim g'$
using *assms(2) assms(3) stepDet strong-noop-bisimilar.simps* **by** *metis*

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

end

8.2 Graph Rewriting

theory
Rewrites
imports
Stuttering
begin

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

lemma *replace-usages-effect*:
assumes $g' = \text{replace-usages } \text{nid} \ \text{nid}' \ g$

```

shows kind  $g'$  nid = RefNode nid'
using assms replace-node-lookup replace-usages.simps
by (metis IRNode.distinct(2755))

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

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

fun nextNid :: IRGraph  $\Rightarrow$  ID where
  nextNid g = (Max (ids g)) + 1

lemma max-plus-one:
  fixes c :: ID set
  shows  $\llbracket \text{finite } c; c \neq \{\} \rrbracket \Longrightarrow (\text{Max } c) + 1 \notin c$ 
  by (meson Max-gr-iff less-add-one less-irrefl)

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

lemma nextNidNotIn:
  ids g  $\neq \{\} \longrightarrow \text{nextNid } g \notin \text{ids } g$ 
  unfolding nextNid.simps
  using ids-finite max-plus-one by blast

fun constantCondition :: bool  $\Rightarrow$  ID  $\Rightarrow$  IRNode  $\Rightarrow$  IRGraph  $\Rightarrow$  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  $g' = \text{constantCondition } \text{True} \ \text{ifcond} \ (\text{kind } g \ \text{ifcond}) \ g$ 
  shows  $g', p \vdash (\text{ifcond}, m, h) \rightarrow (t, m, h)$ 
proof -

```

have *ifn*: $\bigwedge c\ t\ f. \text{IfNode } c\ t\ f \neq \text{NoNode}$
by *simp*
then have *if'*: $\text{kind } g'\ \text{ifcond} = \text{IfNode } (\text{nextNid } g)\ t\ f$
using *assms(1) assms(2) constantCondition.simps(1) replace-node-lookup*
by *presburger*
have *truedef*: $\text{bool-to-val True} = (\text{IntVal } 32\ 1)$
by *auto*
from *ifn* **have** $\text{ifcond} \neq (\text{nextNid } g)$
by (*metis assms(1) emptyE ids-some nextNidNotIn*)
moreover have $\bigwedge c. \text{ConstantNode } c \neq \text{NoNode}$ **by** *simp*
ultimately have $\text{kind } g' (\text{nextNid } g) = \text{ConstantNode } (\text{bool-to-val True})$
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'*: $\text{kind } g' (\text{nextNid } g) = \text{ConstantNode } (\text{IntVal } 32\ 1)$
using *truedef* **by** *simp*
have *valid-value* $(\text{IntVal } 32\ 1) (\text{constantAsStamp } (\text{IntVal } 32\ 1))$
unfolding *constantAsStamp.simps valid-value.simps*
using *nat-numeral* **by** *force*
then have $[g', m, p] \vdash \text{nextNid } g \mapsto \text{IntVal } 32\ 1$
using *ConstantExpr ConstantNode Value.distinct(1) $\langle \text{kind } g' (\text{nextNid } g) = \text{ConstantNode } (\text{bool-to-val True}) \rangle$ encodeeval-def truedef*
by (*metis wf-value-def*)
from *if' c' show ?thesis* **using** *IfNode*
by (*metis (no-types, opaque-lifting) val-to-bool.simps(1) $\langle [g', m, p] \vdash \text{nextNid } g \mapsto \text{IntVal } 32\ 1 \rangle$ encodeeval-def zero-neq-one*)
qed

lemma *constantConditionFalse*:

assumes $\text{kind } g\ \text{ifcond} = \text{IfNode } \text{cond } t\ f$
assumes $g' = \text{constantCondition False ifcond } (\text{kind } g\ \text{ifcond})\ g$
shows $g', p \vdash (\text{ifcond}, m, h) \rightarrow (f, m, h)$
proof –
have *ifn*: $\bigwedge c\ t\ f. \text{IfNode } c\ t\ f \neq \text{NoNode}$
by *simp*
then have *if'*: $\text{kind } g'\ \text{ifcond} = \text{IfNode } (\text{nextNid } g)\ t\ f$
by (*metis assms(1) assms(2) constantCondition.simps(1) replace-node-lookup*)
have *falsedef*: $\text{bool-to-val False} = (\text{IntVal } 32\ 0)$
by *auto*
from *ifn* **have** $\text{ifcond} \neq (\text{nextNid } g)$
by (*metis assms(1) equals0D ids-some nextNidNotIn*)
moreover have $\bigwedge c. \text{ConstantNode } c \neq \text{NoNode}$ **by** *simp*
ultimately have $\text{kind } g' (\text{nextNid } g) = \text{ConstantNode } (\text{bool-to-val False})$
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'*: $\text{kind } g' (\text{nextNid } g) = \text{ConstantNode } (\text{IntVal } 32\ 0)$
using *falsedef* **by** *simp*
have *valid-value* $(\text{IntVal } 32\ 0) (\text{constantAsStamp } (\text{IntVal } 32\ 0))$

```

unfolding constantAsStamp.simps valid-value.simps
using nat-numeral by force
then have  $[g', m, p] \vdash \text{nextNid } g \mapsto \text{IntVal } 32 \ 0$ 
  by (metis wf-value-def ConstantExpr ConstantNode  $\langle \text{kind } g' \ (\text{nextNid } g) =$ 
ConstantNode (bool-to-val False) \rangle encodeeval-def falsedef)
  from if' c' show ?thesis using IfNode
  by (metis (no-types, opaque-lifting) val-to-bool.simps(1)  $\langle [g', m, p] \vdash \text{nextNid } g$ 
 $\mapsto \text{IntVal } 32 \ 0 \rangle \text{ encodeeval-def}$ )
qed

```

```

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

```

```

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

```

```

lemma constantConditionNoEffect:
  assumes  $\neg(\text{is-IfNode } (\text{kind } g \ nid))$ 
  shows  $g = \text{constantCondition } b \ nid \ (\text{kind } g \ nid) \ g$ 
  using assms apply (cases kind g nid)
  using constantCondition.simps
  apply presburger+
  apply (metis is-IfNode-def)
  using constantCondition.simps
  by presburger+

```

```

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

```

```

lemma constantCondition-changeonly:
  assumes  $nid \in \text{ids } g$ 

```

```

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

```

```

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

```

```

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

```

```

    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

```

8.3 Stuttering

```

theory Stuttering
  imports
    Semantics.IRStepThms
begin

```

```

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

```

```

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

```

```

  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

```


8.4 Evaluation Stamp Theorems

```

theory StampEvalThms
  imports Graph.ValueThms
           Semantics.IRTreeEvalThms
begin

lemma
  assumes take-bit b v = v
  shows signed-take-bit b v = v
  using assms
  by (metis(full-types) eq-imp-le signed-take-bit-take-bit)

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

lemma unrestricted-new-int-always-valid [simp]:
  assumes 0 < b ∧ b ≤ 64
  shows valid-value (new-int b v) (unrestricted-stamp (IntegerStamp b lo hi))
  unfolding unrestricted-stamp.simps new-int.simps valid-value.simps
  by (simp; metis One-nat-def assms int-power-div-base int-signed-value.simps
    int-signed-value-range linorder-not-le not-exp-less-eq-0-int zero-less-numeral)

lemma unary-undef: val = UndefVal ⇒ unary-eval op val = UndefVal
  by (cases op; auto)

lemma unary-obj: val = ObjRef x ⇒ unary-eval op val = UndefVal
  by (cases op; auto)

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

lemma unrestricted-stamp-valid-value [simp]:
  assumes 1: result = IntVal b ival
  assumes take-bit b ival = ival
  assumes 0 < b ∧ b ≤ 64
  shows valid-value result (unrestricted-stamp (IntegerStamp b lo hi))
proof –
  have valid-stamp (unrestricted-stamp (IntegerStamp b lo hi))

```

```

    using assms unrestricted-stamp-valid by blast
  then show ?thesis
    unfolding 1 unrestricted-stamp.simps valid-value.simps
    using assms int-signed-value-bounds by presburger
qed

```

8.4.1 Support Lemmas for Integer Stamps and Associated IntVal values

Valid int implies some useful facts.

lemma *valid-int-gives:*

```

  assumes valid-value (IntVal b val) stamp
  obtains lo hi where stamp = IntegerStamp b lo hi ∧
    valid-stamp (IntegerStamp b lo hi) ∧
    take-bit b val = val ∧
    lo ≤ int-signed-value b val ∧ int-signed-value b val ≤ hi
  using assms
  by (smt (z3) Value.distinct(7) Value.inject(1) valid-value.elims(1))

```

And the corresponding lemma where we know the stamp rather than the value.

lemma *valid-int-stamp-gives:*

```

  assumes valid-value val (IntegerStamp b lo hi)
  obtains ival where val = IntVal b ival ∧
    valid-stamp (IntegerStamp b lo hi) ∧
    take-bit b ival = ival ∧
    lo ≤ int-signed-value b ival ∧ int-signed-value b ival ≤ hi
  by (metis assms valid-int valid-value.simps(1))

```

A valid int must have the expected number of bits.

lemma *valid-int-same-bits:*

```

  assumes valid-value (IntVal b val) (IntegerStamp bits lo hi)
  shows b = bits
  by (meson assms valid-value.simps(1))

```

A valid value means a valid stamp.

lemma *valid-int-valid-stamp:*

```

  assumes valid-value (IntVal b val) (IntegerStamp bits lo hi)
  shows valid-stamp (IntegerStamp bits lo hi)
  by (metis assms valid-value.simps(1))

```

A valid int means a valid non-empty stamp.

lemma *valid-int-not-empty:*

```

  assumes valid-value (IntVal b val) (IntegerStamp bits lo hi)
  shows lo ≤ hi
  by (metis assms order.trans valid-value.simps(1))

```

A valid int fits into the given number of bits (and other bits are zero).

lemma *valid-int-fits*:

assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows *take-bit* *bits val* = *val*
by (*metis* *assms* *valid-value.simps*(1))

lemma *valid-int-is-zero-masked*:

assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows *and* *val* (*not* (*mask* *bits*)) = 0
by (*metis* (*no-types*, *lifting*) *assms* *bit.conj-cancel-right* *take-bit-eq-mask* *valid-int-fits*

word-bw-assocs(1) *word-log-esimps*(1))

Unsigned ints have bounds 0 up to 2^{bits} .

lemma *valid-int-unsigned-bounds*:

assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows *uint* *val* < 2^{bits}
by (*metis* *assms*(1) *mask-eq-iff* *take-bit-eq-mask* *valid-value.simps*(1))

Signed ints have the usual two-complement bounds.

lemma *valid-int-signed-upper-bound*:

assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows *int-signed-value* *bits val* < $2^{(\text{bits} - 1)}$
by (*metis* (*mono-tags*, *opaque-lifting*) *diff-le-mono* *int-signed-value.simps* *less-imp-diff-less*

linorder-not-le *one-le-numeral* *order-less-le-trans* *power-increasing* *signed-take-bit-int-less-exp-word* *sint-lt*)

lemma *valid-int-signed-lower-bound*:

assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows $-(2^{(\text{bits} - 1)}) \leq \text{int-signed-value } \text{bits } \text{val}$
by (*smt* (*verit*) *diff-le-self* *int-signed-value.simps* *linorder-not-less* *power-increasing-iff* *signed-take-bit-int-greater-eq-minus-exp-word* *sint-greater-eq*)

and *bit_bounds* versions of the above bounds.

lemma *valid-int-signed-upper-bit-bound*:

assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows *int-signed-value* *bits val* ≤ *snd* (*bit-bounds* *bits*)

proof –

have *b* = *bits* **using** *assms* *valid-int-same-bits* **by** *blast*
then show *?thesis*

using *assms* **by** *force*

qed

lemma *valid-int-signed-lower-bit-bound*:

assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows *fst* (*bit-bounds* *bits*) ≤ *int-signed-value* *bits val*

proof –

have *b* = *bits* **using** *assms* *valid-int-same-bits* **by** *blast*

```

then show ?thesis
  using assms by force
qed

```

Valid values satisfy their stamp bounds.

```

lemma valid-int-signed-range:
  assumes valid-value (IntVal b val) (IntegerStamp bits lo hi)
  shows  $lo \leq \text{int-signed-value } bits \text{ } val \wedge \text{int-signed-value } bits \text{ } val \leq hi$ 
  by (metis assms valid-value.simps(1))

```

8.4.2 Validity of all Unary Operators

We split the validity proof for unary operators into two lemmas, one for normal unary operators whose output bits equals their input bits, and the other case for the widen and narrow operators.

```

lemma eval-normal-unary-implies-valid-value:
  assumes  $[m, p] \vdash \text{expr} \mapsto \text{val}$ 
  assumes  $\text{result} = \text{unary-eval } op \text{ } val$ 
  assumes  $op: op \in \text{normal-unary}$ 
  assumes  $\text{result} \neq \text{UndefVal}$ 
  assumes valid-value val (stamp-expr expr)
  shows valid-value result (stamp-expr (UnaryExpr op expr))
proof –
  obtain b1 v1 where  $v1: val = \text{IntVal } b1 \text{ } v1$ 
  by (metis Value.exhaust assms(1) assms(2) assms(4) assms(5) evaltree-not-undef
unary-obj valid-value.simps(11))
  then obtain b2 v2 where  $v2: \text{result} = \text{IntVal } b2 \text{ } v2$ 
  using assms(2) assms(4) is-IntVal-def unary-eval-int by presburger
  then have  $\text{result} = \text{unary-eval } op \text{ } (\text{IntVal } b1 \text{ } v1)$ 
  using assms(2) v1 by blast
  then obtain vtmp where  $\text{result} = \text{new-int } b2 \text{ } vtmp$ 
  using assms(3) v2 by auto
  obtain b' lo' hi' where  $\text{stamp-expr } \text{expr} = \text{IntegerStamp } b' \text{ } lo' \text{ } hi'$ 
  by (metis assms(5) v1 valid-int-gives)
  then have  $\text{stamp-unary } op \text{ } (\text{stamp-expr } \text{expr}) =$ 
    unrestricted-stamp
    (IntegerStamp (if  $op \in \text{normal-unary}$  then b' else ir-resultBits op) lo' hi')
  using stamp-unary.simps(1) by presburger
  then obtain lo2 hi2 where  $s: (\text{stamp-expr } (\text{UnaryExpr } op \text{ } \text{expr})) = \text{unrestricted-stamp } (\text{IntegerStamp } b2 \text{ } lo2 \text{ } hi2)$ 
  unfolding stamp-expr.simps
  using vtmp op
  by (smt (verit, best) Value.inject(1)  $\langle (\text{result}::\text{Value}) = \text{unary-eval } (op::\text{IRUnaryOp})$ 
(IntVal (b1::nat) (v1::64 word))  $\rangle \langle \text{stamp-expr } (\text{expr}::\text{IRExpr}) = \text{IntegerStamp } (b'::\text{nat})$ 
(lo'::int) (hi'::int)  $\rangle$  assms(5) insertE intval-abs.simps(1) intval-logic-negation.simps(1)
intval-negate.simps(1) intval-not.simps(1) new-int.elims singleton-iff unary-eval.simps(1)
unary-eval.simps(2) unary-eval.simps(3) unary-eval.simps(4) v1 valid-int-same-bits)
  then have  $0 < b1 \wedge b1 \leq 64$ 

```

```

    using valid-int-gives
    by (metis assms(5) v1 valid-stamp.simps(1))
  then have fst (bit-bounds b2) ≤ int-signed-value b2 v2 ∧
    int-signed-value b2 v2 ≤ snd (bit-bounds b2)
    by (smt (verit, del-insts) Stamp.inject(1) assms(3) assms(5) int-signed-value-bounds
    s stamp-expr.simps(1) stamp-unary.simps(1) unrestricted-stamp.simps(2) v1 valid-int-gives)
  then show ?thesis
    unfolding s v2 unrestricted-stamp.simps valid-value.simps
    by (smt (z3) assms(3) assms(5) is-stamp-empty.simps(1) new-int-take-bits s
    stamp-expr.simps(1) stamp-unary.simps(1) unrestricted-stamp.simps(2) v1 v2 valid-int-gives
    valid-stamp.simps(1) vtmp)
qed

```

lemma narrow-widen-output-bits:

```

  assumes unary-eval op val ≠ UndefVal
  assumes op ∉ normal-unary
  shows 0 < (ir-resultBits op) ∧ (ir-resultBits op) ≤ 64
proof -
  consider ib ob where op = UnaryNarrow ib ob
    | ib ob where op = UnarySignExtend ib ob
    | ib ob where op = UnaryZeroExtend ib ob
  using IRUnaryOp.exhaust-sel assms(2) by blast
  then show ?thesis
  proof (cases)
    case 1
    then show ?thesis using assms intval-narrow-ok by force
  next
    case 2
    then show ?thesis using assms intval-sign-extend-ok by force
  next
    case 3
    then show ?thesis using assms intval-zero-extend-ok by force
  qed
qed

```

lemma eval-widen-narrow-unary-implies-valid-value:

```

  assumes [m,p] ⊢ expr ↦ val
  assumes result = unary-eval op val
  assumes op: op ∉ normal-unary
  assumes result ≠ UndefVal
  assumes valid-value val (stamp-expr expr)
  shows valid-value result (stamp-expr (UnaryExpr op expr))
proof -
  obtain b1 v1 where v1: val = IntVal b1 v1
  by (metis Value.exhaust assms(1) assms(2) assms(4) assms(5) evaltree-not-undef
  unary-obj valid-value.simps(11))
  then have result = unary-eval op (IntVal b1 v1)
    using assms(2) v1 by blast

```

```

then obtain v2 where v2: result = new-int (ir-resultBits op) v2
  using assms by (cases op; simp; (meson new-int.simps)+)
then obtain v3 where v3: result = IntVal (ir-resultBits op) v3
  using assms by (cases op; simp; (meson new-int.simps)+)
then obtain lo2 hi2 where s: (stamp-expr (UnaryExpr op expr)) = unre-
stricted-stamp (IntegerStamp (ir-resultBits op) lo2 hi2)
  unfolding stamp-expr.simps stamp-unary.simps
  using assms(3) assms(5) v1 valid-int-gives by fastforce
then have outBits: 0 < (ir-resultBits op) ∧ (ir-resultBits op) ≤ 64
  using assms narrow-widen-output-bits
  by blast
then have fst (bit-bounds (ir-resultBits op)) ≤ int-signed-value (ir-resultBits op)
v3 ∧
  int-signed-value (ir-resultBits op) v3 ≤ snd (bit-bounds (ir-resultBits op))
  using int-signed-value-bounds
  by (smt (verit, del-insts) Stamp.inject(1) assms(3) assms(5) int-signed-value-bounds
s stamp-expr.simps(1) stamp-unary.simps(1) unrestricted-stamp.simps(2) v1 valid-int-gives)
then show ?thesis
  unfolding s v3 unrestricted-stamp.simps valid-value.simps
  using outBits v2 v3 by auto
qed

```

```

lemma eval-unary-implies-valid-value:
  assumes [m,p] ⊢ expr ↦ val
  assumes result = unary-eval op val
  assumes result ≠ UndefVal
  assumes valid-value val (stamp-expr expr)
  shows valid-value result (stamp-expr (UnaryExpr op expr))
  proof (cases op ∈ normal-unary)
    case True
      then show ?thesis by (metis assms eval-normal-unary-implies-valid-value)
    next
      case False
      then show ?thesis by (metis assms eval-widen-narrow-unary-implies-valid-value)
  qed

```

8.4.3 Support Lemmas for Binary Operators

```

lemma binary-undef: v1 = UndefVal ∨ v2 = UndefVal ⇒ bin-eval op v1 v2 =
UndefVal
  by (cases op; auto)

```

```

lemma binary-obj: v1 = ObjRef x ∨ v2 = ObjRef y ⇒ bin-eval op v1 v2 =
UndefVal
  by (cases op; auto)

```

Some lemmas about the three different output sizes for binary operators.

```

lemma bin-eval-bits-binary-shift-ops:
  assumes result = bin-eval op (IntVal b1 v1) (IntVal b2 v2)

```

```

assumes result  $\neq$  UndefVal
assumes op  $\in$  binary-shift-ops
shows  $\exists v. \text{result} = \text{new-int } b1 \ v$ 
using assms
by (cases op; simp; smt (verit, best) new-int.simps) +

lemma bin-eval-bits-fixed-32-ops:
assumes result = bin-eval op (IntVal b1 v1) (IntVal b2 v2)
assumes result  $\neq$  UndefVal
assumes op  $\in$  binary-fixed-32-ops
shows  $\exists v. \text{result} = \text{new-int } 32 \ v$ 
using assms
apply (cases op; simp)
using assms bool-to-val.simps bin-eval-new-int new-int.simps bin-eval-unused-bits-zero
by metis +

lemma bin-eval-bits-normal-ops:
assumes result = bin-eval op (IntVal b1 v1) (IntVal b2 v2)
assumes result  $\neq$  UndefVal
assumes op  $\notin$  binary-shift-ops
assumes op  $\notin$  binary-fixed-32-ops
shows  $\exists v. \text{result} = \text{new-int } b1 \ v$ 
using assms apply (cases op; simp)
using assms apply (metis (mono-tags)) +
using take-bit-and apply metis
using take-bit-or apply metis
using take-bit-xor by metis

lemma bin-eval-input-bits-equal:
assumes result = bin-eval op (IntVal b1 v1) (IntVal b2 v2)
assumes result  $\neq$  UndefVal
assumes op  $\notin$  binary-shift-ops
shows b1 = b2
using assms apply (cases op; simp)
by presburger +

lemma bin-eval-implies-valid-value:
assumes [m,p]  $\vdash$  expr1  $\mapsto$  val1
assumes [m,p]  $\vdash$  expr2  $\mapsto$  val2
assumes result = bin-eval op val1 val2
assumes result  $\neq$  UndefVal
assumes valid-value val1 (stamp-expr expr1)
assumes valid-value val2 (stamp-expr expr2)
shows valid-value result (stamp-expr (BinaryExpr op expr1 expr2))
proof –
obtain b1 v1 where v1: val1 = IntVal b1 v1
by (metis Value.collapse(1) assms(3) assms(4) bin-eval-inputs-are-ints bin-eval-int)
obtain b2 v2 where v2: val2 = IntVal b2 v2

```

```

    by (metis Value.collapse(1) assms(3) assms(4) bin-eval-inputs-are-ints bin-eval-int)
  then obtain lo1 hi1 where s1: stamp-expr expr1 = IntegerStamp b1 lo1 hi1
    by (metis assms(5) v1 valid-int-gives)
  then obtain lo2 hi2 where s2: stamp-expr expr2 = IntegerStamp b2 lo2 hi2
    by (metis assms(6) v2 valid-int-gives)
  then have r: result = bin-eval op (IntVal b1 v1) (IntVal b2 v2)
    using assms(3) v1 v2 by blast
  then obtain bres vtmp where vtmp: result = new-int bres vtmp
    using assms bin-eval-bits-binary-shift-ops
    by (meson bin-eval-new-int)
  then obtain vres where vres: result = IntVal bres vres
    by force

  then have sres: stamp-expr (BinaryExpr op expr1 expr2) =
    unrestricted-stamp (IntegerStamp bres lo1 hi1)
     $\wedge 0 < bres \wedge bres \leq 64$ 
  proof (cases op  $\in$  binary-shift-ops)
    case True
    then show ?thesis
      unfolding s1 s2 stamp-binary.simps stamp-expr.simps
      using assms bin-eval-bits-binary-shift-ops
      by (metis Value.inject(1) eval-bits-1-64 new-int.simps r v1 vres)
  next
    case False
    then have op  $\notin$  binary-shift-ops
      by simp
    then have beq: b1 = b2
      using v1 v2 assms bin-eval-input-bits-equal by simp
    then show ?thesis
      proof (cases op  $\in$  binary-fixed-32-ops)
        case True
        then show ?thesis
          unfolding s1 s2 stamp-binary.simps stamp-expr.simps
          using assms bin-eval-bits-fixed-32-ops
          by (metis False Value.inject(1) beq bin-eval-new-int le-add-same-cancel1
            new-int.simps numeral-Bit0 vres zero-le-numeral zero-less-numeral)
        next
          case False
          then show ?thesis
            unfolding s1 s2 stamp-binary.simps stamp-expr.simps
            using assms
            by (metis beq bin-eval-new-int eval-bits-1-64 intval-bits.simps unrestricted-new-int-always-valid
              unrestricted-stamp.simps(2) v1 valid-int-same-bits vres)
      qed
    qed
  then show ?thesis
    unfolding vres
    using unrestricted-new-int-always-valid vres vtmp by presburger
  qed

```


8.4.4 Validity of Stamp Meet and Join Operators

```

lemma stamp-meet-integer-is-valid-stamp:
  assumes valid-stamp stamp1
  assumes valid-stamp stamp2
  assumes is-IntegerStamp stamp1
  assumes is-IntegerStamp stamp2
  shows valid-stamp (meet stamp1 stamp2)
  using assms unfolding is-IntegerStamp-def valid-stamp.simps meet.simps
  by (smt (verit, del-insts) meet.simps(2) valid-stamp.simps(1) valid-stamp.simps(8))

```

```

lemma stamp-meet-is-valid-stamp:
  assumes 1: valid-stamp stamp1
  assumes 2: valid-stamp stamp2
  shows valid-stamp (meet stamp1 stamp2)
  by (cases stamp1; cases stamp2; insert stamp-meet-integer-is-valid-stamp[OF 1
2]; auto)

```

```

lemma stamp-meet-commutes: meet stamp1 stamp2 = meet stamp2 stamp1
  by (cases stamp1; cases stamp2; auto)

```

```

lemma stamp-meet-is-valid-value1:
  assumes valid-value val stamp1
  assumes valid-stamp stamp2
  assumes stamp1 = IntegerStamp b1 lo1 hi1
  assumes stamp2 = IntegerStamp b2 lo2 hi2
  assumes meet stamp1 stamp2 ≠ IllegalStamp
  shows valid-value val (meet stamp1 stamp2)
proof –
  have m: meet stamp1 stamp2 = IntegerStamp b1 (min lo1 lo2) (max hi1 hi2)
    using assms by (metis meet.simps(2))
  obtain ival where val: val = IntVal b1 ival
    using assms valid-int by blast
  then have v: valid-stamp (IntegerStamp b1 lo1 hi1) ∧
    take-bit b1 ival = ival ∧
    lo1 ≤ int-signed-value b1 ival ∧ int-signed-value b1 ival ≤ hi1
    using assms by (metis valid-value.simps(1))
  then have mm: min lo1 lo2 ≤ int-signed-value b1 ival ∧ int-signed-value b1 ival
≤ max hi1 hi2
    by linarith
  then have valid-stamp (IntegerStamp b1 (min lo1 lo2) (max hi1 hi2))
    using assms v stamp-meet-is-valid-stamp
    by (metis meet.simps(2))
  then show ?thesis
    unfolding m val valid-value.simps
    using mm v by presburger
qed

```

and the symmetric lemma follows by the commutativity of meet.

```

lemma stamp-meet-is-valid-value:
  assumes valid-value val stamp2
  assumes valid-stamp stamp1
  assumes stamp1 = IntegerStamp b1 lo1 hi1
  assumes stamp2 = IntegerStamp b2 lo2 hi2
  assumes meet stamp1 stamp2  $\neq$  IllegalStamp
  shows valid-value val (meet stamp1 stamp2)
  using assms stamp-meet-commutes stamp-meet-is-valid-value1
  by metis

```

8.4.5 Validity of conditional expressions

```

lemma conditional-eval-implies-valid-value:
  assumes [m,p]  $\vdash$  cond  $\mapsto$  condv
  assumes expr = (if val-to-bool condv then expr1 else expr2)
  assumes [m,p]  $\vdash$  expr  $\mapsto$  val
  assumes val  $\neq$  UndefinedVal
  assumes valid-value condv (stamp-expr cond)
  assumes valid-value val (stamp-expr expr)
  assumes compatible (stamp-expr expr1) (stamp-expr expr2)
  shows valid-value val (stamp-expr (ConditionalExpr cond expr1 expr2))
proof –
  have def: meet (stamp-expr expr1) (stamp-expr expr2)  $\neq$  IllegalStamp
    using assms
  by (metis Stamp.distinct(13) Stamp.distinct(25) compatible.elims(2) meet.simps(1)
    meet.simps(2))
  then have valid-stamp (meet (stamp-expr expr1) (stamp-expr expr2))
    using assms
  by (smt (verit, best) compatible.elims(2) stamp-meet-is-valid-stamp valid-stamp.simps(2))

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

qed

```

8.4.6 Validity of Whole Expression Tree Evaluation

TODO: find a way to encode that conditional expressions must have compatible (and valid) stamps? One approach would be for all the stamp_expr operators to require that all input stamps are valid.

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

```

lemma stamp-under-defn:
  assumes stamp-under (stamp-expr x) (stamp-expr y)
  assumes wf-stamp x  $\wedge$  wf-stamp y
  assumes ([m, p]  $\vdash$  x  $\mapsto$  xv)  $\wedge$  ([m, p]  $\vdash$  y  $\mapsto$  yv)
  shows val-to-bool (bin-eval BinIntegerLessThan xv yv)

```

```

proof –
  have yval: valid-value yv (stamp-expr y)
    using assms wf-stamp-def by blast
  obtain b lx hi where xstamp: stamp-expr x = IntegerStamp b lx hi
    using assms(1)
    by (metis stamp-under.elims(2))
  then obtain lo hy where ystamp: stamp-expr y = IntegerStamp b lo hy
    using assms(1)
    by (metis Stamp.sel(1) stamp-under.elims(2))
  obtain xvv where xvv: xv = IntVal b xvv
    by (metis assms(2) assms(3) valid-int wf-stamp-def xstamp)
  then have xval: valid-value (IntVal b xvv) (stamp-expr x)
    using assms(2) assms(3) wf-stamp-def by blast
  obtain yvv where yvv: yv = IntVal b yvv
    by (metis valid-int ystamp yval)
  then have xval: valid-value (IntVal b yvv) (stamp-expr y)
    using yval by auto
  have xunder: int-signed-value b xvv ≤ hi
    using xvv xval valid-value.simps
    by (metis assms(2) assms(3) wf-stamp-def xstamp)
  have yunder: lo ≤ int-signed-value b yvv
    using yvv yval valid-value.simps
    by (metis ystamp)
  have unwrap:  $\forall \text{cond. } \text{bool-to-val-bin } b \text{ } \text{cond} = \text{bool-to-val cond}$ 
    by simp
  from xunder yunder have int-signed-value b xvv < int-signed-value b yvv
    using assms(1) xstamp ystamp by auto
  then have (intval-less-than xv yv) = IntVal 32 1
    using xvv yvv
    using intval-less-than.simps(1) unwrap
    using bool-to-val.simps(1) by presburger
  then show ?thesis
    by simp
qed

```

```

lemma stamp-under-defn-inverse:
  assumes stamp-under (stamp-expr y) (stamp-expr x)
  assumes wf-stamp x ∧ wf-stamp y
  assumes ( $[m, p] \vdash x \mapsto xv$ )  $\wedge$  ( $[m, p] \vdash y \mapsto yv$ )
  shows  $\neg(\text{val-to-bool } (\text{bin-eval BinIntegerLessThan } xv yv))$ 
proof –
  have yval: valid-value yv (stamp-expr y)
    using assms wf-stamp-def by blast
  obtain b lo hx where xstamp: stamp-expr x = IntegerStamp b lo hx
    using assms(1)
    by (metis stamp-under.elims(2))
  then obtain ly hi where ystamp: stamp-expr y = IntegerStamp b ly hi
    using assms(1)
    by (metis Stamp.sel(1) stamp-under.elims(2))

```

```

obtain xvv where xvv: xv = IntVal b xv
  by (metis assms(2) assms(3) valid-int wf-stamp-def xstamp)
then have xval: valid-value (IntVal b xv) (stamp-expr x)
  using assms(2) assms(3) wf-stamp-def by blast
obtain yvv where yvv: yv = IntVal b yv
  by (metis valid-int ystamp yval)
then have xval: valid-value (IntVal b yvv) (stamp-expr y)
  using yval by auto
have yunder: int-signed-value b yvv ≤ hi
  using yvv yval valid-value.simps
  by (metis ystamp)
have xover: lo ≤ int-signed-value b xv
  using xvv xval valid-value.simps
  by (metis assms(2) assms(3) wf-stamp-def xstamp)
have unwrap: ∀ cond. bool-to-val-bin b b cond = bool-to-val cond
  by simp
from xover yunder have int-signed-value b yvv < int-signed-value b xv
  using assms(1) xstamp ystamp by auto
then have (intval-less-than xv yv) = IntVal 32 0
  using xvv yvv
  using intval-less-than.simps(1) unwrap
  by force
then show ?thesis
  by simp
qed

end

```

9 Optimization DSL

9.1 Markup

```

theory Markup
  imports Semantics.IRTreeEval Snippets.Snipping
begin

```

```

datatype 'a Rewrite =
  Transform 'a 'a (- ⟶ - 10) |
  Conditional 'a 'a bool (- ⟶ - when - 11) |
  Sequential 'a Rewrite 'a Rewrite |
  Transitive 'a Rewrite

```

```

datatype 'a ExtraNotation =
  ConditionalNotation 'a 'a 'a (- ? - : - 50) |
  EqualsNotation 'a 'a (- eq -) |
  ConstantNotation 'a (const - 120) |
  TrueNotation (true) |
  FalseNotation (false) |
  ExclusiveOr 'a 'a (- ⊕ -) |

```

```
LogicNegationNotation 'a (!-) |
ShortCircuitOr 'a 'a (- || -)
```

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

ML-file <markup.ML>

9.1.1 Expression Markup

```
ML <
structure IRExprTranslator : DSL-TRANSLATION =
struct
fun markup DSL-Tokens.Add = @{term BinaryExpr} $ @{term BinAdd}
  | markup DSL-Tokens.Sub = @{term BinaryExpr} $ @{term BinSub}
  | markup DSL-Tokens.Mul = @{term BinaryExpr} $ @{term BinMul}
  | markup DSL-Tokens.And = @{term BinaryExpr} $ @{term BinAnd}
  | markup DSL-Tokens.Or = @{term BinaryExpr} $ @{term BinOr}
  | markup DSL-Tokens.Xor = @{term BinaryExpr} $ @{term BinXor}
  | markup DSL-Tokens.ShortCircuitOr = @{term BinaryExpr} $ @{term Bin-
ShortCircuitOr}
  | markup DSL-Tokens.Abs = @{term UnaryExpr} $ @{term UnaryAbs}
  | markup DSL-Tokens.Less = @{term BinaryExpr} $ @{term BinIntegerLessThan}
  | markup DSL-Tokens.Equals = @{term BinaryExpr} $ @{term BinIntegerEquals}
  | markup DSL-Tokens.Not = @{term UnaryExpr} $ @{term UnaryNot}
  | markup DSL-Tokens.Negate = @{term UnaryExpr} $ @{term UnaryNeg}
  | markup DSL-Tokens.LogicNegate = @{term UnaryExpr} $ @{term UnaryLog-
icNegation}
  | markup DSL-Tokens.LeftShift = @{term BinaryExpr} $ @{term BinLeftShift}
  | markup DSL-Tokens.RightShift = @{term BinaryExpr} $ @{term BinRightShift}
  | markup DSL-Tokens.UnsignedRightShift = @{term BinaryExpr} $ @{term Bin-
URightShift}
  | markup DSL-Tokens.Conditional = @{term ConditionalExpr}
  | markup DSL-Tokens.Constant = @{term ConstantExpr}
  | markup DSL-Tokens.TrueConstant = @{term ConstantExpr (IntVal 32 1)}
  | markup DSL-Tokens.FalseConstant = @{term ConstantExpr (IntVal 32 0)}
end
structure IRExprMarkup = DSL-Markup(IRExprTranslator);
>
```

ir expression translation

```
syntax -expandExpr :: term  $\Rightarrow$  term (exp[-])
parse-translation < [( @{syntax-const -expandExpr} , IREx-
prMarkup.markup-expr []) ] >
```

ir expression example

value $\text{exp}[(e_1 < e_2) ? e_1 : e_2]$

ConditionalExpr (BinaryExpr BinIntegerLessThan e_1 e_2) e_1 e_2

9.1.2 Value Markup

ML <

structure IntValTranslator : DSL-TRANSLATION =
struct

fun markup DSL-Tokens.Add = @{term intval-add}
| *markup DSL-Tokens.Sub = @{term intval-sub}*
| *markup DSL-Tokens.Mul = @{term intval-mul}*
| *markup DSL-Tokens.And = @{term intval-and}*
| *markup DSL-Tokens.Or = @{term intval-or}*
| *markup DSL-Tokens.ShortCircuitOr = @{term intval-short-circuit-or}*
| *markup DSL-Tokens.Xor = @{term intval-xor}*
| *markup DSL-Tokens.Abs = @{term intval-abs}*
| *markup DSL-Tokens.Less = @{term intval-less-than}*
| *markup DSL-Tokens.Equals = @{term intval-equals}*
| *markup DSL-Tokens.Not = @{term intval-not}*
| *markup DSL-Tokens.Negate = @{term intval-negate}*
| *markup DSL-Tokens.LogicNegate = @{term intval-logic-negation}*
| *markup DSL-Tokens.LeftShift = @{term intval-left-shift}*
| *markup DSL-Tokens.RightShift = @{term intval-right-shift}*
| *markup DSL-Tokens.UnsignedRightShift = @{term intval-uright-shift}*
| *markup DSL-Tokens.Conditional = @{term intval-conditional}*
| *markup DSL-Tokens.Constant = @{term IntVal 32}*
| *markup DSL-Tokens.TrueConstant = @{term IntVal 32 1}*
| *markup DSL-Tokens.FalseConstant = @{term IntVal 32 0}*

end

structure IntValMarkup = DSL-Markup(IntValTranslator);

>

value expression translation

syntax *-expandIntVal :: term \Rightarrow term (val[-])*

parse-translation < [(@{syntax-const -expandIntVal} , IntVal-
Markup.markup-expr [])] >

value expression example

value $\text{val}[(e_1 < e_2) ? e_1 : e_2]$

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

9.1.3 Word Markup

```

ML <
structure WordTranslator : DSL-TRANSLATION =
struct
fun markup DSL-Tokens.Add = @{term plus}
  | markup DSL-Tokens.Sub = @{term minus}
  | markup DSL-Tokens.Mul = @{term times}
  | markup DSL-Tokens.And = @{term Bit-Operations.semiring-bit-operations-class.and}
  | markup DSL-Tokens.Or = @{term or}
  | markup DSL-Tokens.Xor = @{term xor}
  | markup DSL-Tokens.Abs = @{term abs}
  | markup DSL-Tokens.Less = @{term less}
  | markup DSL-Tokens.Equals = @{term HOL.eq}
  | markup DSL-Tokens.Not = @{term not}
  | markup DSL-Tokens.Negate = @{term uminus}
  | markup DSL-Tokens.LogicNegate = @{term logic-negate}
  | markup DSL-Tokens.LeftShift = @{term shiftl}
  | markup DSL-Tokens.RightShift = @{term signed-shiftr}
  | markup DSL-Tokens.UnsignedRightShift = @{term shiftr}
  | markup DSL-Tokens.Constant = @{term word}
  | markup DSL-Tokens.TrueConstant = @{term 1}
  | markup DSL-Tokens.FalseConstant = @{term 0}
end
structure WordMarkup = DSL-Markup(WordTranslator);
>

```

word expression translation

syntax *-expandWord* :: *term* \Rightarrow *term* (*bin*[-])
parse-translation < [(@{syntax-const *-expandWord*} , Word-
Markup.markup-expr [])] >

word expression example

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

intval-conditional (*intval-less-than* *e*₁ *e*₂) *e*₁ *e*₂

value *bin*[-*x*]
value *val*[-*x*]
value *exp*[-*x*]

value *bin*[!*x*]
value *val*[!*x*]
value *exp*[!*x*]

value *bin*[-*x*]
value *val*[-*x*]
value *exp*[-*x*]

```

value bin[ $\sim x$ ]
value val[ $\sim x$ ]
value exp[ $\sim x$ ]

value  $\sim x$ 

end

```

9.2 Optimization Phases

```

theory Phase
  imports Main
begin

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

end

```

9.3 Canonicalization DSL

```

theory Canonicalization
  imports
    Markup
    Phase
    HOL-Eisbach.Eisbach
  keywords
    phase :: thy-decl and
    terminating :: quasi-command and
    print-phases :: diag and
    export-phases :: thy-decl and
    optimization :: thy-goal-defn
begin

print-methods

ML <
datatype 'a Rewrite =
  Transform of 'a * 'a |
  Conditional of 'a * 'a * term |
  Sequential of 'a Rewrite * 'a Rewrite |
  Transitive of 'a Rewrite

type rewrite = {
  name: binding,
  rewrite: term Rewrite,
  proofs: thm list,
  code: thm list,
  source: term

```



```

}

structure RewriteRule : Rule =
struct
type T = rewrite;

(*
fun pretty-rewrite ctxt (Transform (from, to)) =
  Pretty.block [
    Syntax.pretty-term ctxt from,
    Pretty.str  $\mapsto$  ,
    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-thm ctxt thm =
  (Proof-Context.pretty-fact ctxt (, [thm]))

fun pretty ctxt obligations t =
  let
    val is-skipped = Thm-Deps.has-skip-proof (#proofs t);

    val warning = (if is-skipped
      then [Pretty.str (proof skipped), Pretty.brk 0]
      else []);

    val obligations = (if obligations
      then [Pretty.big-list
        obligations:
        (map (pretty-thm ctxt) (#proofs t)),
        Pretty.brk 0]
      else []);

    fun pretty-bind binding =
      Pretty.markup
        (Position.markup (Binding.pos-of binding) Markup.position)
        [Pretty.str (Binding.name-of binding)];

  in
    Pretty.block ([
      pretty-bind (#name t), Pretty.str : ,

```

```

    Syntax.pretty-term ctxt (#source t), Pretty.fbrk
  ] @ obligations @ warning)
end
end

structure RewritePhase = DSL-Phase(RewriteRule);

val - =
  Outer-Syntax.command command-keyword⟨phase⟩ enter an optimization phase
  (Parse.binding --| Parse.$$$ terminating -- Parse.const --| Parse.begin
   >> (Toplevel.begin-main-target true o RewritePhase.setup));

fun print-phases print-obligations ctxt =
  let
    val thy = Proof-Context.theory-of ctxt;
    fun print phase = RewritePhase.pretty print-obligations phase ctxt
  in
    map print (RewritePhase.phases thy)
  end

fun print-optimizations print-obligations thy =
  print-phases print-obligations thy |> Pretty.writeln-chunks

val - =
  Outer-Syntax.command command-keyword⟨print-phases⟩
  print debug information for optimizations
  (Parse.opt-bang >>
   (fn b => Toplevel.keep ((print-optimizations b) o Toplevel.context-of)));

fun export-phases thy name =
  let
    val state = Toplevel.theory-tolevel thy;
    val ctxt = Toplevel.context-of state;
    val content = Pretty.string-of (Pretty.chunks (print-phases false ctxt));
    val cleaned = YXML.content-of content;

    val filename = Path.explode (name^".rules");
    val directory = Path.explode optimizations;
    val path = Path.binding (
      Path.append directory filename,
      Position.none);
    val thy' = thy |> Generated-Files.add-files (path, content);

    val - = Export.export thy' path [YXML.parse cleaned];

    val - = writeln (Export.message thy' (Path.basic optimizations));
  in
    thy'
  end

```

```

end

val - =
  Outer-Syntax.command command-keyword <export-phases>
    export information about encoded optimizations
    (Parse.text >>
      (fn name => Toplevel.theory (fn state => export-phases state name)))
  >

```

ML-file *rewrites.ML*

9.3.1 Semantic Preservation Obligation

```

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

```

9.3.2 Termination Obligation

```

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

```

9.3.3 Standard Termination Measure

```

fun size :: IRExp  $\Rightarrow$  nat where
  unary-size:
  size (UnaryExpr op e) = (size e) + 2 |

  bin-const-size:
  size (BinaryExpr op x (ConstantExpr cy)) = (size x) + 2 |
  bin-size:
  size (BinaryExpr op x y) = (size x) + (size y) + 2 |
  cond-size:
  size (ConditionalExpr cond t f) = (size cond) + (size t) + (size f) + 2 |
  const-size:
  size (ConstantExpr c) = 1 |
  param-size:

```

```

size (ParameterExpr ind s) = 2 |
leaf-size:
size (LeafExpr nid s) = 2 |
size (ConstantVar c) = 2 |
size (VariableExpr x s) = 2

```

9.3.4 Automated Tactics

named-theorems *size-simps size simplification rules*

```

method unfold-optimization =
  (unfold rewrite-preservation.simps, unfold rewrite-termination.simps,
   unfold intval.simps,
   rule conjE, simp, simp del: le-expr-def, force?)
| (unfold rewrite-preservation.simps, unfold rewrite-termination.simps,
   rule conjE, simp, simp del: le-expr-def, force?)

```

```

method unfold-size =
  (((unfold size.simps, simp add: size-simps del: le-expr-def)?
   ; (simp add: size-simps del: le-expr-def)?
   ; (auto simp: size-simps)?
   ; (unfold size.simps)?)[1])

```

print-methods

```

ML <
  structure System : RewriteSystem =
  struct
    val preservation = @{const rewrite-preservation};
    val termination = @{const rewrite-termination};
    val intval = @{const intval};
  end

  structure DSL = DSL-Rewrites(System);

  val - =
    Outer-Syntax.local-theory-to-proof command-keyword <optimization>
    define an optimization and open proof obligation
    (Parse-Spec.thm-name : -- Parse.term
     >> DSL.rewrite-cmd);
>

```

end

10 Canonicalization Optimizations

```

theory Common
  imports

```

OptimizationDSL.Canonicalization
Semantics.IRTreeEvalThms

begin

lemma *size-pos[size-simps]*: $0 < \text{size } y$
apply (*induction y; auto?*)
by (*smt (z3) add-2-eq-Suc' add-is-0 not-gr0 size.elims size.simps(12) size.simps(13)*
size.simps(14) size.simps(15) zero-neq-numeral zero-neq-one)

lemma *size-non-add[size-simps]*: $\text{size } (\text{BinaryExpr op } a \ b) = \text{size } a + \text{size } b + 2$
 $\longleftrightarrow \neg(\text{is-ConstantExpr } b)$
by (*induction b; induction op; auto simp: is-ConstantExpr-def*)

lemma *size-non-const[size-simps]*:
 $\neg \text{is-ConstantExpr } y \implies 1 < \text{size } y$
using *size-pos* **apply** (*induction y; auto*)
by (*metis Suc-lessI add-is-1 is-ConstantExpr-def le-less linorder-not-le n-not-Suc-n*
numeral-2-eq-2 pos2 size.simps(2) size-non-add)

lemma *size-binary-const[size-simps]*:
 $\text{size } (\text{BinaryExpr op } a \ b) = \text{size } a + 2 \longleftrightarrow (\text{is-ConstantExpr } b)$
by (*induction b; auto simp: is-ConstantExpr-def size-pos*)

lemma *size-flip-binary[size-simps]*:
 $\neg(\text{is-ConstantExpr } y) \longrightarrow \text{size } (\text{BinaryExpr op } (\text{ConstantExpr } x) \ y) > \text{size } (\text{BinaryExpr op } y \ (\text{ConstantExpr } x))$
by (*metis add-Suc not-less-eq order-less-asym plus-1-eq-Suc size.simps(11) size.simps(2)*
size-non-add)

lemma *size-binary-lhs-a[size-simps]*:
 $\text{size } (\text{BinaryExpr op } (\text{BinaryExpr op' } a \ b) \ c) > \text{size } a$
by (*metis add-lessD1 less-add-same-cancel1 pos2 size-binary-const size-non-add*)

lemma *size-binary-lhs-b[size-simps]*:
 $\text{size } (\text{BinaryExpr op } (\text{BinaryExpr op' } a \ b) \ c) > \text{size } b$
by (*metis IRExp.disc(42) One-nat-def add.left-commute add.right-neutral is-ConstantExpr-def*
less-add-Suc2 numeral-2-eq-2 plus-1-eq-Suc size.simps(11) size-binary-const size-non-add
size-non-const trans-less-add1)

lemma *size-binary-lhs-c[size-simps]*:
 $\text{size } (\text{BinaryExpr op } (\text{BinaryExpr op' } a \ b) \ c) > \text{size } c$
by (*metis IRExp.disc(42) add.left-commute add.right-neutral is-ConstantExpr-def*
less-Suc-eq numeral-2-eq-2 plus-1-eq-Suc size.simps(11) size-non-add size-non-const
trans-less-add2)

lemma *size-binary-rhs-a[size-simps]*:
 $\text{size } (\text{BinaryExpr op } c \ (\text{BinaryExpr op' } a \ b)) > \text{size } a$
by (*smt (verit, best) less-Suc-eq less-add-Suc2 less-add-same-cancel1 linorder-neqE-nat*
not-add-less1 order-less-trans pos2 size.simps(4) size-binary-const size-non-add)

lemma *size-binary-rhs-b*[*size-simps*]:
 $\text{size } (\text{BinaryExpr } \text{op } c \ (\text{BinaryExpr } \text{op}' \ a \ b)) > \text{size } b$
by (*metis add.left-commute add.right-neutral is-ConstantExpr-def lessI numeral-2-eq-2 plus-1-eq-Suc size.simps(11) size.simps(4) size-non-add trans-less-add2*)

lemma *size-binary-rhs-c*[*size-simps*]:
 $\text{size } (\text{BinaryExpr } \text{op } c \ (\text{BinaryExpr } \text{op}' \ a \ b)) > \text{size } c$
by *simp*

lemma *size-binary-lhs*[*size-simps*]:
 $\text{size } (\text{BinaryExpr } \text{op } x \ y) > \text{size } x$
by (*metis One-nat-def Suc-eq-plus1 add-Suc-right less-add-Suc1 numeral-2-eq-2 size-binary-const size-non-add*)

lemma *size-binary-rhs*[*size-simps*]:
 $\text{size } (\text{BinaryExpr } \text{op } x \ y) > \text{size } y$
by (*metis IRExpr.disc(42) add-strict-increasing is-ConstantExpr-def linorder-not-le not-add-less1 size.simps(11) size-non-add size-non-const size-pos*)

lemmas *arith*[*size-simps*] = *Suc-leI add-strict-increasing order-less-trans trans-less-add2*

definition *well-formed-equal* :: *Value* \Rightarrow *Value* \Rightarrow *bool*
 (*infix* \approx 50) **where**
 $\text{well-formed-equal } v_1 \ v_2 = (v_1 \neq \text{UndefVal} \longrightarrow v_1 = v_2)$

lemma *well-formed-equal-defn* [*simp*]:
 $\text{well-formed-equal } v_1 \ v_2 = (v_1 \neq \text{UndefVal} \longrightarrow v_1 = v_2)$
unfolding *well-formed-equal-def* **by** *simp*

end

10.1 AbsNode Phase

theory *AbsPhase*
imports
 Common
begin

phase *AbsNode*
 terminating *size*
begin

lemma *abs-pos*:
fixes $v :: ('a :: \text{len } \text{word})$

```

assumes  $0 \leq_s v$ 
shows (if  $v <_s 0$  then  $-v$  else  $v$ ) =  $v$ 
by (simp add: assms signed.leD)

lemma abs-neg:
  fixes  $v :: ('a :: \text{len word})$ 
  assumes  $v <_s 0$ 
  assumes  $-(2^{\wedge}(\text{Nat.size } v - 1)) <_s v$ 
  shows (if  $v <_s 0$  then  $-v$  else  $v$ ) =  $-v \wedge 0 <_s -v$ 
by (smt (verit, ccfv-SIG) assms(1) assms(2) signed-take-bit-int-greater-eq-minus-exp

    signed-take-bit-int-greater-eq-self-iff sint-0 sint-word-ariths(4) word-sless-alt)

lemma abs-max-neg:
  fixes  $v :: ('a :: \text{len word})$ 
  assumes  $v <_s 0$ 
  assumes  $-(2^{\wedge}(\text{Nat.size } v - 1)) = v$ 
  shows  $-v = v$ 
  using assms
  by (metis One-nat-def add.inverse-neutral double-eq-zero-iff mult-minus-right
    size-word.rep-eq)

lemma final-abs:
  fixes  $v :: ('a :: \text{len word})$ 
  assumes take-bit (Nat.size  $v$ )  $v = v$ 
  assumes  $-(2^{\wedge}(\text{Nat.size } v - 1)) \neq v$ 
  shows  $0 \leq_s$  (if  $v <_s 0$  then  $-v$  else  $v$ )

proof (cases  $v <_s 0$ )
  case True
  then show ?thesis
  proof (cases  $v = -(2^{\wedge}(\text{Nat.size } v - 1))$ )
  case True
  then show ?thesis using abs-max-neg
  using assms by presburger
  next
  case False
  then have  $-(2^{\wedge}(\text{Nat.size } v - 1)) <_s v$ 
  unfolding word-sless-def using signed-take-bit-int-greater-self-iff
  by (smt (verit, best) One-nat-def diff-less double-eq-zero-iff len-gt-0 lessI
    less-irrefl
    mult-minus-right neg-equal-0-iff-equal signed.rep-eq signed-of-int
    signed-take-bit-int-greater-eq-self-iff signed-word-eqI sint-0 sint-range-size
    sint-sbintrunc' sint-word-ariths(4) size-word.rep-eq unsigned-0 word-2p-lem

    word-sless.rep-eq word-sless-def)
  then show ?thesis
  using abs-neg abs-pos signed.nless-le by auto
qed

```

```

next
  case False
  then show ?thesis using abs-pos by auto
qed

```

```

lemma wf-abs: is-IntVal x  $\implies$  intval-abs x  $\neq$  UndefVal
  using intval-abs.simps unfolding new-int.simps
  using is-IntVal-def by force

```

```

fun bin-abs :: 'a :: len word  $\Rightarrow$  'a :: len word where
  bin-abs v = (if (v < s 0) then (− v) else v)

```

```

lemma val-abs-zero:
  intval-abs (new-int b 0) = new-int b 0
by simp

```

```

lemma less-eq-zero:
  assumes val-to-bool (val[(IntVal b 0) < (IntVal b v)])
  shows int-signed-value b v > 0
  using assms unfolding intval-less-than.simps(1) apply simp
  by (metis bool-to-val.elims val-to-bool.simps(1))

```

```

lemma val-abs-pos:
  assumes val-to-bool(val[(new-int b 0) < (new-int b v)])
  shows intval-abs (new-int b v) = (new-int b v)
  using assms using less-eq-zero unfolding intval-abs.simps new-int.simps
  by force

```

```

lemma val-abs-neg:
  assumes val-to-bool(val[(new-int b v) < (new-int b 0)])
  shows intval-abs (new-int b v) = intval-negate (new-int b v)
  using assms using less-eq-zero unfolding intval-abs.simps new-int.simps
  by force

```

```

lemma val-bool-unwrap:
  val-to-bool (bool-to-val v) = v
  by (metis bool-to-val.elims one-neq-zero val-to-bool.simps(1))

```

```

lemma take-bit-unwrap:
  b = 64  $\implies$  take-bit b (v1::64 word) = v1
  by (metis size64 size-word.rep-eq take-bit-length-eq)

```

```

lemma bit-less-eq-def:
  fixes v1 v2 :: 64 word
  assumes b  $\leq$  64

```



```

shows sint (signed-take-bit (b - Suc (0::nat)) (take-bit b v1))
  < sint (signed-take-bit (b - Suc (0::nat)) (take-bit b v2))  $\longleftrightarrow$ 
  signed-take-bit (63::nat) (Word.rep v1) < signed-take-bit (63::nat) (Word.rep
v2)
using assms sorry

```

lemma less-eq-def:

```

shows val-to-bool(val[(new-int b v1) < (new-int b v2)])  $\longleftrightarrow$  v1 <s v2
unfolding new-int.simps intval-less-than.simps bool-to-val-bin.simps bool-to-val.simps
int-signed-value.simps apply (simp add: val-bool-unwrap)
apply auto unfolding word-sless-def apply auto
unfolding signed-def apply auto using bit-less-eq-def
apply (metis bot-nat-0.extremum take-bit-0)
by (metis bit-less-eq-def bot-nat-0.extremum take-bit-0)

```

lemma val-abs-always-pos:

```

assumes intval-abs (new-int b v) = (new-int b v')
shows 0 ≤s v'
using assms
proof (cases v = 0)
  case True
  then have v' = 0
  using val-abs-zero assms
  by (smt (verit, ccfv-threshold) Suc-diff-1 bit-less-eq-def bot-nat-0.extremum
diff-is-0-eq len-gt-0 len-of-numeral-defs(2) order-le-less signed-eq-0-iff take-bit-0 take-bit-signed-take-bit
take-bit-unwrap)
  then show ?thesis by simp
next
  case neq0: False
  then show ?thesis
  proof (cases val-to-bool(val[(new-int b 0) < (new-int b v)]))
    case True
    then show ?thesis using less-eq-def
    using assms val-abs-pos
    by (smt (verit, ccfv-SIG) One-nat-def Suc-leI bit.compl-one bit-less-eq-def
cancel-comm-monoid-add-class.diff-cancel diff-zero len-gt-0 len-of-numeral-defs(2)
mask-0 mask-1 one-le-numeral one-neq-zero signed-word-eqI take-bit-dist-subL take-bit-minus-one-eq-mask
take-bit-not-eq-mask-diff take-bit-signed-take-bit zero-le-numeral)
  next
  case False
  then have val-to-bool(val[(new-int b v) < (new-int b 0)])
  using neq0 less-eq-def
  by (metis signed.neqE)
  then show ?thesis using val-abs-neg less-eq-def unfolding new-int.simps
intval-negate.simps
  by (metis signed.nless-le take-bit-0)
qed

```

qed

lemma *intval-abs-elim:*

assumes *intval-abs* $x \neq \text{UndefVal}$
shows $\exists t v . x = \text{IntVal } t v \wedge \text{intval-abs } x = \text{new-int } t \text{ (if int-signed-value } t v < 0 \text{ then } -v \text{ else } v)$
using *assms*
by (*meson intval-abs.elims*)

lemma *wf-abs-new-int:*

assumes *intval-abs* $(\text{IntVal } t v) \neq \text{UndefVal}$
shows *intval-abs* $(\text{IntVal } t v) = \text{new-int } t v \vee \text{intval-abs } (\text{IntVal } t v) = \text{new-int } t (-v)$
using *assms*
using *intval-abs.simps(1)* **by** *presburger*

lemma *mono-undef-abs:*

assumes *intval-abs* $(\text{intval-abs } x) \neq \text{UndefVal}$
shows *intval-abs* $x \neq \text{UndefVal}$
using *assms*
by *force*

lemma *val-abs-idem:*

assumes *intval-abs* $(\text{intval-abs}(x)) \neq \text{UndefVal}$
shows *intval-abs* $(\text{intval-abs}(x)) = \text{intval-abs } x$
using *assms*
proof –
obtain $b v$ **where** *in-def*: *intval-abs* $x = \text{new-int } b v$
using *assms intval-abs-elim mono-undef-abs* **by** *blast*
then show *?thesis*
proof (*cases val-to-bool(val[(new-int b v) < (new-int b 0)])*)
case *True*
then have *nested*: *intval-abs* $(\text{intval-abs } x) = \text{new-int } b (-v)$
using *val-abs-neg intval-negate.simps in-def*
by *simp*
then have $x = \text{new-int } b (-v)$
using *in-def True unfolding new-int.simps*
by (*smt (verit, best) intval-abs.simps(1) less-eq-def less-eq-zero less-numeral-extra(1)*)

mask-1 mask-eq-take-bit-minus-one neg-one.elims neg-one-signed new-int.simps

one-le-numeral one-neq-zero signed.neqE signed.not-less take-bit-of-0
val-abs-always-pos)
then show *?thesis* **using** *val-abs-always-pos*
using *True in-def less-eq-def signed.leD*
using *signed.nless-le* **by** *blast*
next

```

    case False
    then show ?thesis
      using in-def by force
qed
qed

lemma val-abs-negate:
  assumes intval-abs (intval-negate x) ≠ UndefVal
  shows intval-abs (intval-negate x) = intval-abs x
  using assms apply (cases x; auto)
  apply (metis less-eq-def new-int.simps signed.dual-order.strict-iff-not signed.less-linear

      take-bit-0)
  by (smt (verit, ccfv-threshold) add.inverse-neutral intval-abs.simps(1) less-eq-def
less-eq-zero
    less-numeral-extra(1) mask-1 mask-eq-take-bit-minus-one neg-one.elims neg-one-signed

    new-int.simps one-le-numeral one-neg-zero signed.order.order-iff-strict take-bit-of-0

    val-abs-always-pos)

Optimisations

optimization AbsIdempotence: abs(abs(x)) ⟶ abs(x)
  apply auto
  by (metis UnaryExpr unary-eval.simps(1) val-abs-idem)

optimization AbsNegate: (abs(−x)) ⟶ abs(x)
  apply auto using val-abs-negate
  by (metis unary-eval.simps(1) unfold-unary)

end

end

```

10.2 AddNode Phase

```

theory AddPhase
  imports
    Common
begin

phase AddNode
  terminating size
begin

lemma binadd-commute:
  assumes bin-eval BinAdd x y ≠ UndefVal
  shows bin-eval BinAdd x y = bin-eval BinAdd y x
  using assms intval-add-sym by simp

```

optimization *AddShiftConstantRight*: $((\text{const } v) + y) \mapsto y + (\text{const } v)$ when $\neg(\text{is-ConstantExpr } y)$
 using *size-non-const*
 apply (*metis add-2-eq-Suc' less-Suc-eq plus-1-eq-Suc size.simps(11) size-non-add*)
 unfolding *le-expr-def*
 apply (*rule impI*)
 subgoal premises 1
 apply (*rule allI impI*)
 done
 subgoal premises 2 for *m p va*
 apply (*rule BinaryExprE[OF 2]*)
 subgoal premises 3 for *x ya*
 apply (*rule BinaryExpr*)
 using 3 apply *simp*
 using 3 apply *simp*
 using 3 *binadd-commute* apply *auto*
 done
 done
 done
 done

optimization *AddShiftConstantRight2*: $((\text{const } v) + y) \mapsto y + (\text{const } v)$ when $\neg(\text{is-ConstantExpr } y)$
 unfolding *le-expr-def*
 apply (*auto simp: intval-add-sym*)
 using *size-non-const*
 by (*metis add-2-eq-Suc' lessI plus-1-eq-Suc size.simps(11) size-non-add*)

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

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

ML-val $\langle @\{\text{term } \langle x = y \rangle\} \rangle$

```

lemma NeutralLeftSubVal:
  assumes  $e1 = \text{new-int } b \text{ ival}$ 
  shows  $\text{val}[(e1 - e2) + e2] \approx e1$ 
  apply simp using assms by (cases e1; cases e2; auto)

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

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

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

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

lemma AddToSubHelperLowLevel:
  shows  $\text{intval-add } (\text{intval-negate } e) y = \text{intval-sub } y e \text{ (is } ?x = ?y)$ 
  by (induction y; induction e; auto)

print-phases

lemma val-redundant-add-sub:

```

```

assumes  $a = \text{new-int } bb \text{ ival}$ 
assumes  $\text{val}[b + a] \neq \text{UndefVal}$ 
shows  $\text{val}[(b + a) - b] = a$ 
using assms apply (cases  $a$ ; cases  $b$ ; auto)
by presburger

```

```

lemma val-add-right-negate-to-sub:
assumes  $\text{val}[x + e] \neq \text{UndefVal}$ 
shows  $\text{val}[x + (-e)] = \text{val}[x - e]$ 
using assms by (cases  $x$ ; cases  $e$ ; auto)

```

```

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

```

Optimisations

```

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

```

```

optimization AddRightNegateToSub:  $x + -e \mapsto x - e$ 
apply (metis Nat.add-0-right add-2-eq-Suc' add-less-mono1 add-mono-thms-linordered-field(2)
less-SucI not-less-less-Suc-eq size-binary-const size-non-add size-pos)
using AddToSubHelperLowLevel intval-add-sym by auto

```

```

optimization AddLeftNegateToSub:  $-e + y \mapsto y - e$ 
defer
using exp-add-left-negate-to-sub apply blast
by (smt (verit, best) One-nat-def add.commute add-Suc-right is-ConstantExpr-def
less-add-Suc2
numeral-2-eq-2 plus-1-eq-Suc size.simps(1) size.simps(11) size-binary-const
size-non-add)

```

end

end

10.3 AndNode Phase

```

theory AndPhase
imports

```

```

    Common
    Proofs.StampEvalThms
begin

context stamp-mask
begin

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

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

```

```

phase AndNode
  terminating size
begin

```

```

lemma bin-and-nots:
  ( $\sim x \ \& \ \sim y$ ) = ( $\sim(x \mid y)$ )
  by simp

```

```

lemma bin-and-neutral:
  ( $x \ \& \ \sim False$ ) =  $x$ 
  by simp

```

```

lemma val-and-equal:
  assumes  $x = \text{new-int } b \ v$ 
  and      $\text{val}[x \ \& \ x] \neq \text{UndefVal}$ 
  shows    $\text{val}[x \ \& \ x] = x$ 
  using  assms by (cases x; auto)

```

```

lemma val-and-nots:
   $\text{val}[\sim x \ \& \ \sim y] = \text{val}[\sim(x \mid y)]$ 
  apply (cases x; cases y; auto) by (simp add: take-bit-not-take-bit)

```

```

lemma val-and-neutral:
  assumes  $x = \text{new-int } b \ v$ 
  and      $\text{val}[x \ \& \ \sim(\text{new-int } b' \ 0)] \neq \text{UndefVal}$ 
  shows    $\text{val}[x \ \& \ \sim(\text{new-int } b' \ 0)] = x$ 
  using  assms apply (cases x; auto) apply (simp add: take-bit-eq-mask)
  by presburger

```

```

lemma val-and-zero:
  assumes  $x = \text{new-int } b \ v$ 
  shows    $\text{val}[x \ \& \ (\text{IntVal } b \ 0)] = \text{IntVal } b \ 0$ 
  using  assms by (cases x; auto)

```

```

lemma exp-and-equal:
   $\text{exp}[x \ \& \ x] \geq \text{exp}[x]$ 
  apply auto using val-and-equal eval-unused-bits-zero
  by (smt (verit) evalDet intval-and.elims new-int.elims)

```

```

lemma exp-and-nots:
   $\text{exp}[\sim x \ \& \ \sim y] \geq \text{exp}[\sim(x \mid y)]$ 

```



```

apply (cases x; cases y; auto) using val-and-nots
by fastforce+

lemma exp-sign-extend:
assumes e = (1 << In) - 1
shows BinaryExpr BinAnd (UnaryExpr (UnarySignExtend In Out) x)
      (ConstantExpr (new-int b e))
      ≥ (UnaryExpr (UnaryZeroExtend In Out) x)

apply auto
subgoal premises p for m p va
proof -
  obtain va where va: [m,p] ⊢ x ↦ va
  using p(2) by auto
  then have va ≠ UndefVal
    by (simp add: evaltree-not-undef)
  then have 1: intval-and (intval-sign-extend In Out va) (IntVal b (take-bit b
e)) ≠ UndefVal
    using evalDet p(1) p(2) va by blast
  then have 2: intval-sign-extend In Out va ≠ UndefVal
    by auto
  then have 21: (0::nat) < b
    using eval-bits-1-64 p(4) by blast
  then have 3: b ⊆ (64::nat)
    using eval-bits-1-64 p(4) by blast
  then have 4: - ((2::int) ^ b div (2::int)) ⊆ sint (signed-take-bit (b - Suc
(0::nat)) (take-bit b e))
    by (simp add: 21 int-power-div-base signed-take-bit-int-greater-eq-minus-exp-word)
  then have 5: sint (signed-take-bit (b - Suc (0::nat)) (take-bit b e)) < (2::int)
^ b div (2::int)
    by (simp add: 21 3 Suc-le-lessD int-power-div-base signed-take-bit-int-less-exp-word)
  then have 6: [m,p] ⊢ UnaryExpr (UnaryZeroExtend In Out)
    x ↦ intval-and (intval-sign-extend In Out va) (IntVal b (take-bit b e))
    apply (cases va; simp)
  apply (simp add: ⟨(va::Value) ≠ UndefVal⟩) defer
  subgoal premises p for x3
  proof -
    have va = ObjRef x3
    using p(1) by auto
    then have sint (signed-take-bit (b - Suc (0::nat)) (take-bit b e)) <
(2::int) ^ b div (2::int)
      by (simp add: 5)
    then show ?thesis
      using 2 intval-sign-extend.simps(3) p(1) by blast
  qed

subgoal premises p for x4
proof -
  have sg1: va = ObjStr x4
  using 2 p(1) by auto

```

```

      then have sint (signed-take-bit (b - Suc (0::nat)) (take-bit b e)) <
(2::int) ^ b div (2::int)
      by (simp add: 5)
      then show ?thesis
      using 1 sg1 by auto
    qed

```

```

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

```

```

lemma val-and-commute[simp]:
  val[x & y] = val[y & x]
  apply (cases x; cases y; auto)
  by (simp add: word-bw-comms(1))

```

Optimisations

```

optimization AndEqual: x & x ⟶ x
  using exp-and-equal by blast

```

```

optimization AndShiftConstantRight: ((const x) & y) ⟶ y & (const x)
  when ¬(is-ConstantExpr y)
  using size-flip-binary by auto

```

```

optimization AndNots: (~x) & (~y) ⟶ ~(x | y)
  defer using exp-and-nots
  apply presburger
  by (metis add-2-eq-Suc' less-SucI less-add-Suc1 not-less-eq size-binary-const size-non-add)

```

```

optimization AndSignExtend: BinaryExpr BinAnd (UnaryExpr (UnarySignExtend

```

```

In Out) (x))
      (const (new-int b e))
     $\mapsto$  (UnaryExpr (UnaryZeroExtend In Out) (x))
      when (e = (1 << In) - 1)
  using exp-sign-extend by simp

optimization AndNeutral: (x & ~ (const (IntVal b 0)))  $\mapsto$  x
  when (wf-stamp x  $\wedge$  stamp-expr x = IntegerStamp b lo hi)
  apply auto
by (smt (verit) Value.sel(1) eval-unused-bits-zero intval-and.elims intval-word.simps

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

optimization AndRightFallThrough: (x & y)  $\mapsto$  y
  when (((and (not (IRExpr-down x)) (IRExpr-up y)) = 0))
by (simp add: IRExpr-down-def IRExpr-up-def)

optimization AndLeftFallThrough: (x & y)  $\mapsto$  x
  when (((and (not (IRExpr-down y)) (IRExpr-up x)) = 0))
by (simp add: IRExpr-down-def IRExpr-up-def)

end

end

```

10.4 BinaryNode Phase

```

theory BinaryNode
  imports
    Common
begin

phase BinaryNode
  terminating size
begin

optimization BinaryFoldConstant: BinaryExpr op (const v1) (const v2)  $\mapsto$  ConstantExpr (bin-eval op v1 v2)
  unfolding le-expr-def
  apply (rule allI impI)+
  subgoal premises bin for m p v
  print-facts
  apply (rule BinaryExprE[OF bin])
  subgoal premises prems for x y
  print-facts

```

```

proof –
  have  $x$ :  $x = v1$  using prems by auto
  have  $y$ :  $y = v2$  using prems by auto
  have  $xy$ :  $v = \text{bin-eval } op \ x \ y$  using prems  $x \ y$  by simp
  have  $int$ :  $\exists \ b \ vv . v = \text{new-int } b \ vv$  using bin-eval-new-int prems by fast
  show ?thesis
    unfolding prems  $x \ y \ xy$ 
    apply (rule ConstantExpr)
    using prems  $x \ y \ xy \ int$  sorry
  qed
done
done

```

print-facts

end

end

10.5 ConditionalNode Phase

theory *ConditionalPhase*

imports

Common

Proofs.StampEvalThms

begin

phase *ConditionalNode*

terminating *size*

begin

lemma *negates*: $\exists \ v \ b. \ e = \text{IntVal } b \ v \wedge b > 0 \implies \text{val-to-bool } (\text{val}[e]) \longleftrightarrow \neg(\text{val-to-bool } (\text{val}[\neg e]))$

unfolding *intval-logic-negation.simps*

by (*metis* (*mono-tags*, *lifting*) *intval-logic-negation.simps*(1) *logic-negate-def* *new-int.simps* *of-bool-eq*(2) *one-neq-zero* *take-bit-of-0* *take-bit-of-1* *val-to-bool.simps*(1))

lemma *negation-condition-intval*:

assumes $e = \text{IntVal } b \ ie$

assumes $0 < b$

shows $\text{val}[(\neg e) \ ? \ x : y] = \text{val}[e \ ? \ y : x]$

using *assms* **by** (*cases* e ; *auto* *simp*: *negates* *logic-negate-def*)

lemma *negation-preserve-eval*:

assumes $[m, p] \vdash \text{exp}[\neg e] \mapsto v$

shows $\exists \ v'. ([m, p] \vdash \text{exp}[e] \mapsto v') \wedge v = \text{val}[\neg v']$

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

lemma *negation-preserve-eval-intval*:

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

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

optimization *DefaultTrueBranch*: $(\text{true} \text{ ? } x : y) \mapsto x$.

optimization *DefaultFalseBranch*: $(\text{false} \text{ ? } x : y) \mapsto y$.

optimization *ConditionalEqualBranches*: $(e \text{ ? } x : x) \mapsto x$.

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

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

lemma *val-optimise-integer-test*:
assumes $\exists v. x = \text{IntVal } 32 \text{ } v$
shows $\text{val}[((x \& (\text{IntVal } 32 \text{ } 1)) \text{ eq } (\text{IntVal } 32 \text{ } 0)) \text{ ? } (\text{IntVal } 32 \text{ } 0) : (\text{IntVal } 32 \text{ } 1)]$
 $=$
 $\text{val}[x \& \text{IntVal } 32 \text{ } 1]$
using *assms apply auto*
apply (*metis (full-types) bool-to-val.simps(2) val-to-bool.simps(1)*)
by (*metis (mono-tags, lifting) and-one-eq bool-to-val.simps(1) even-iff-mod-2-eq-zero odd-iff-mod-2-eq-one val-to-bool.simps(1)*)

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

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

optimization *normalizeX*: $((x \text{ eq } \text{const } (\text{IntVal } 32 \ 0)) \ ?$
 $(\text{const } (\text{IntVal } 32 \ 0)) : (\text{const } (\text{IntVal } 32 \ 1))) \mapsto x$
 $\text{when } (\text{IRExpr-up } x = 1) \wedge \text{stamp-expr } x = \text{IntegerStamp}$

b 0 1

apply *auto*

subgoal *premises p for m p v xa*

proof –

obtain *xa* **where** *xa*: $[m, p] \vdash x \mapsto xa$

using *p* **by** *blast*

have β : $[m, p] \vdash \text{if val-to-bool } (\text{intval-equals } xa \ (\text{IntVal } (32::\text{nat}) \ (0::64 \ \text{word})))$
 $\text{then } \text{ConstantExpr } (\text{IntVal } (32::\text{nat}) \ (0::64 \ \text{word}))$
 $\text{else } \text{ConstantExpr } (\text{IntVal } (32::\text{nat}) \ (1::64 \ \text{word})) \mapsto v$

using *evalDet p(β) p(5) xa* **by** *blast*

then **have** δ : $xa = \text{IntVal } 32 \ 0 \mid xa = \text{IntVal } 32 \ 1$

sorry

then **have** δ : $v = xa$

sorry

then **show** *?thesis*

using *xa* **by** *auto*

qed

done

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

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

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

lemma *stamp-of-default*:

assumes *stamp-expr x = default-stamp*

assumes *wf-stamp x*

shows $([m, p] \vdash x \mapsto v) \longrightarrow (\exists vv. v = \text{IntVal } 32 \ vv)$

using *assms*

by $(\text{metis default-stamp valid-value-elim}(3) \text{wf-stamp-def})$

optimization *OptimiseIntegerTest*:

```

  (((x & (const (IntVal 32 1))) eq (const (IntVal 32 0))) ?
   (const (IntVal 32 0)) : (const (IntVal 32 1)))  $\mapsto$ 
   x & (const (IntVal 32 1))
   when (stamp-expr x = default-stamp  $\wedge$  wf-stamp x)
  apply simp apply (rule impI; (rule allI)+; rule impI)
  subgoal premises eval for m p v
proof -
  obtain xv where xv: [m, p]  $\vdash$  x  $\mapsto$  xv
  using eval by fast
  then have x32:  $\exists v. xv = \text{IntVal } 32 \ v$ 
  using stamp-of-default eval by auto
  obtain lhs where lhs: [m, p]  $\vdash$  exp[(((x & (const (IntVal 32 1))) eq (const (IntVal
32 0))) ?
    (const (IntVal 32 0)) : (const (IntVal 32 1))))]  $\mapsto$  lhs
  using eval(2) by auto
  then have lhsV: lhs = val[((xv & (IntVal 32 1)) eq (IntVal 32 0)) ? (IntVal 32
0) : (IntVal 32 1)]
  using xv evaltree.BinaryExpr evaltree.ConstantExpr evaltree.ConditionalExpr
  by (smt (verit) ConditionalExprE ConstantExprE bin-eval.simps(11) bin-eval.simps(4)
evalDet intval-conditional.simps unfold-binary)
  obtain rhs where rhs: [m, p]  $\vdash$  exp[x & (const (IntVal 32 1))]  $\mapsto$  rhs
  using eval(2) by blast
  then have rhsV: rhs = val[xv & IntVal 32 1]
  by (metis BinaryExprE ConstantExprE bin-eval.simps(4) evalDet xv)
  have lhs = rhs using val-optimize-integer-test x32
  using lhsV rhsV by presburger
  then show ?thesis
  by (metis eval(2) evalDet lhs rhs)
qed
done

```

optimization *opt-optimize-integer-test-2*:

```

  (((x & (const (IntVal 32 1))) eq (const (IntVal 32 0))) ?
   (const (IntVal 32 0)) : (const (IntVal 32 1)))  $\mapsto$ 
   x
   when (x = ConstantExpr (IntVal 32 0) | (x = ConstantExpr (IntVal
32 1))) .

```

end

end

10.6 MulNode Phase

theory *MulPhase*

imports

Common

Proofs.StampEvalThms

begin

fun *mul-size* :: *IRExpr* \Rightarrow *nat* **where**

mul-size (*UnaryExpr op e*) = (*mul-size e*) + 2 |

mul-size (*BinaryExpr BinMul x y*) = ((*mul-size x*) + (*mul-size y*) + 2) * 2 |

mul-size (*BinaryExpr op x y*) = (*mul-size x*) + (*mul-size y*) + 2 |

mul-size (*ConditionalExpr cond t f*) = (*mul-size cond*) + (*mul-size t*) + (*mul-size f*) + 2 |

mul-size (*ConstantExpr c*) = 1 |

mul-size (*ParameterExpr ind s*) = 2 |

mul-size (*LeafExpr nid s*) = 2 |

mul-size (*ConstantVar c*) = 2 |

mul-size (*VariableExpr x s*) = 2

phase *MulNode*

terminating *mul-size*

begin

lemma *bin-eliminate-redundant-negative*:

uminus (*x* :: '*a*::len word) * *uminus* (*y* :: '*a*::len word) = *x* * *y*

by *simp*

lemma *bin-multiply-identity*:

(*x* :: '*a*::len word) * 1 = *x*

by *simp*

lemma *bin-multiply-eliminate*:

(*x* :: '*a*::len word) * 0 = 0

by *simp*

lemma *bin-multiply-negative*:

(*x* :: '*a*::len word) * *uminus* 1 = *uminus x*

by *simp*

lemma *bin-multiply-power-2*:

(*x* :: '*a*::len word) * ($2^{\hat{j}}$) = *x* << *j*

by *simp*

lemma *take-bit64*[*simp*]:


```

fixes  $w :: \text{int64}$ 
shows  $\text{take-bit } 64 \ w = w$ 
proof –
  have  $\text{Nat.size } w = 64$ 
  by ( $\text{simp add: size64}$ )
  then show  $?thesis$ 
  by ( $\text{metis lt2p-lem mask-eq-iff take-bit-eq-mask verit-comp-simplify1(2) wsst-TYs(3)}$ )
qed

lemma testt:
  fixes  $a :: \text{nat}$ 
  fixes  $b \ c :: 64 \ \text{word}$ 
  shows  $\text{take-bit } a \ (\text{take-bit } a \ b) * \text{take-bit } a \ c) =$ 
     $\text{take-bit } a \ (b * c)$ 
by ( $\text{smt (verit, ccfv-SIG) take-bit-mult take-bit-of-int unsigned-take-bit-eq word-mult-def}$ )

lemma val-eliminate-redundant-negative:
  assumes  $\text{val}[-x * -y] \neq \text{UndefVal}$ 
  shows  $\text{val}[-x * -y] = \text{val}[x * y]$ 
  using  $\text{assms apply (cases x; cases y; auto)}$ 
  using testt by auto

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

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

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

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

 $\text{verit-minus-simplify(4) zero-neq-one}$ 

```

```

lemma val-MulPower2:
  fixes i :: 64 word
  assumes y = IntVal 64 (2 ^ unat(i))
  and    0 < i
  and    i < 64
  and    val[x * y] ≠ UndefVal
  shows  val[x * y] = val[x << IntVal 64 i]
  using assms apply (cases x; cases y; auto)
  subgoal premises p for x2
  proof –
    have 63: (63 :: int64) = mask 6
      by eval
    then have (2::int) ^ 6 = 64
      by eval
    then have uint i < (2::int) ^ 6
      by (metis linorder-not-less lt2p-lem of-int-numeral p(4) size64 word-2p-lem
word-of-int-2p wsst-TYs(3))
    then have and i (mask 6) = i
      using mask-eq-iff by blast
    then show x2 << unat i = x2 << unat (and i (63::64 word))
      unfolding 63
      by force
  qed
by presburger

```

```

lemma val-MulPower2Add1:
  fixes i :: 64 word
  assumes y = IntVal 64 ((2 ^ unat(i)) + 1)
  and    0 < i
  and    i < 64
  and    val-to-bool(val[IntVal 64 0 < x])
  and    val-to-bool(val[IntVal 64 0 < y])
  shows  val[x * y] = val[(x << IntVal 64 i) + x]
  using assms apply (cases x; cases y; auto)
  subgoal premises p for x2
  proof –
    have 63: (63 :: int64) = mask 6
      by eval
    then have (2::int) ^ 6 = 64
      by eval
    then have and i (mask 6) = i
      using mask-eq-iff by (simp add: less-mask-eq p(6))
    then have x2 * ((2::64 word) ^ unat i + (1::64 word)) = (x2 * ((2::64 word)
^ unat i)) + x2
      by (simp add: distrib-left)
    then show x2 * ((2::64 word) ^ unat i + (1::64 word)) = x2 << unat (and i
(63::64 word)) + x2
      by (simp add: 63 and (i::64 word) (mask (6::nat)) = i)

```

qed
 using *val-to-bool.simps*(2) by *presburger*

lemma *val-MulPower2Sub1*:
 fixes $i :: 64 \text{ word}$
 assumes $y = \text{IntVal } 64 ((2 \wedge \text{unat}(i)) - 1)$
 and $0 < i$
 and $i < 64$
 and $\text{val-to-bool}(\text{val}[\text{IntVal } 64 \ 0 < x])$
 and $\text{val-to-bool}(\text{val}[\text{IntVal } 64 \ 0 < y])$
 shows $\text{val}[x * y] = \text{val}[(x << \text{IntVal } 64 \ i) - x]$
 using *assms* **apply** (*cases* x ; *cases* y ; *auto*)
 subgoal premises p for $x2$
proof –
 have $63: (63 :: \text{int}64) = \text{mask } 6$
 by *eval*
 then have $(2 :: \text{int}) \wedge 6 = 64$
 by *eval*
 then have $\text{and } i (\text{mask } 6) = i$
 using *mask-eq-iff* by (*simp* *add: less-mask-eq* $p(6)$)
 then have $x2 * ((2 :: 64 \text{ word}) \wedge \text{unat } i - (1 :: 64 \text{ word})) = (x2 * ((2 :: 64 \text{ word}) \wedge \text{unat } i)) - x2$
 by (*simp* *add: right-diff-distrib'*)
 then show $x2 * ((2 :: 64 \text{ word}) \wedge \text{unat } i - (1 :: 64 \text{ word})) = x2 << \text{unat } (\text{and } i (63 :: 64 \text{ word})) - x2$
 by (*simp* *add: 63 (and (i :: 64 word) (mask (6 :: nat)) = i)*)
 qed
 using *val-to-bool.simps*(2) by *presburger*

lemma *val-distribute-multiplication*:
 assumes $x = \text{new-int } 64 \ xx \wedge q = \text{new-int } 64 \ qq \wedge a = \text{new-int } 64 \ aa$
 shows $\text{val}[x * (q + a)] = \text{val}[(x * q) + (x * a)]$
apply (*cases* x ; *cases* q ; *cases* a ; *auto*) **using** *distrib-left* *assms* **by** *auto*

lemma *val-MulPower2AddPower2*:
 fixes $i \ j :: 64 \text{ word}$
 assumes $y = \text{IntVal } 64 ((2 \wedge \text{unat}(i)) + (2 \wedge \text{unat}(j)))$
 and $0 < i$
 and $0 < j$
 and $i < 64$
 and $j < 64$
 and $x = \text{new-int } 64 \ xx$
 shows $\text{val}[x * y] = \text{val}[(x << \text{IntVal } 64 \ i) + (x << \text{IntVal } 64 \ j)]$
 using *assms*
proof –

```

have 63: (63 :: int64) = mask 6
  by eval
then have (2::int) ^ 6 = 64
  by eval
then have n: IntVal 64 ((2 ^ unat(i)) + (2 ^ unat(j))) =
  val[(IntVal 64 (2 ^ unat(i))) + (IntVal 64 (2 ^ unat(j)))]

  using assms by (cases i; cases j; auto)
then have 1: val[x * ((IntVal 64 (2 ^ unat(i))) + (IntVal 64 (2 ^ unat(j))))] =
  val[(x * IntVal 64 (2 ^ unat(i))) + (x * IntVal 64 (2 ^ unat(j)))]

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

thm-oracles val-MulPower2AddPower2

lemma exp-multiply-zero-64:
  exp[x * (const (IntVal 64 0))] ≥ ConstantExpr (IntVal 64 0)
  using val-multiply-zero apply auto
  using Value.inject(1) constantAsStamp.simps(1) int-signed-value-bounds intval-mul.elims
    mult-zero-right new-int.simps new-int-bin.simps nle-le numeral-eq-Suc
take-bit-of-0
  unfold-const valid-stamp.simps(1) valid-value.simps(1) zero-less-Suc wf-value-def
  by (smt (verit))

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

thm-oracles exp-multiply-neutral

lemma exp-MulPower2:
  fixes i :: 64 word
  assumes y = ConstantExpr (IntVal 64 (2 ^ unat(i)))
  and 0 < i

```

```

and      i < 64
and      exp[x > (const IntVal b 0)]
and      exp[y > (const IntVal b 0)]
shows exp[x * y] ≥ exp[x << ConstantExpr (IntVal 64 i)]
  using assms apply simp
by (metis ConstantExprE equiv-exprs-def unfold-binary)

lemma exp-MulPower2Add1:
  fixes i :: 64 word
  assumes y = ConstantExpr (IntVal 64 ((2 ^ unat(i)) + 1))
  and      0 < i
  and      i < 64
  and      exp[x > (const IntVal b 0)]
  and      exp[y > (const IntVal b 0)]
shows exp[x * y] ≥ exp[(x << ConstantExpr (IntVal 64 i)) + x]
  using assms apply simp
  by (metis (no-types, lifting) ConstantExprE equiv-exprs-def unfold-binary)

lemma exp-MulPower2Sub1:
  fixes i :: 64 word
  assumes y = ConstantExpr (IntVal 64 ((2 ^ unat(i)) - 1))
  and      0 < i
  and      i < 64
  and      exp[x > (const IntVal b 0)]
  and      exp[y > (const IntVal b 0)]
shows exp[x * y] ≥ exp[(x << ConstantExpr (IntVal 64 i)) - x]
  using assms apply simp
  by (metis (no-types, lifting) ConstantExprE equiv-exprs-def unfold-binary)

lemma exp-MulPower2AddPower2:
  fixes i j :: 64 word
  assumes y = ConstantExpr (IntVal 64 ((2 ^ unat(i)) + (2 ^ unat(j))))
  and      0 < i
  and      0 < j
  and      i < 64
  and      j < 64
  and      exp[x > (const IntVal b 0)]
  and      exp[y > (const IntVal b 0)]
shows exp[x * y] ≥ exp[(x << ConstantExpr (IntVal 64 i)) + (x << ConstantExpr
(IntVal 64 j))]
  using assms apply simp
  by (metis (no-types, lifting) ConstantExprE equiv-exprs-def unfold-binary)

lemma greaterConstant:
  fixes a b :: 64 word
  assumes a > b

```

```

and     $y = \text{ConstantExpr } (\text{IntVal } 64 \ a)$ 
and     $x = \text{ConstantExpr } (\text{IntVal } 64 \ b)$ 
shows  $\text{exp}[y > x]$ 
apply auto
sorry

lemma exp-distribute-multiplication:
shows  $\text{exp}[(x * q) + (x * a)] \geq \text{exp}[x * (q + a)]$ 
sorry

Optimisations

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

optimization MulNeutral:  $x * \text{ConstantExpr } (\text{IntVal } b \ 1) \mapsto x$ 
using exp-multiply-neutral by blast

optimization MulEliminator:  $x * \text{ConstantExpr } (\text{IntVal } b \ 0) \mapsto \text{const } (\text{IntVal } b \ 0)$ 
apply auto
by (smt (verit) Value.inject(1) constantAsStamp.simps(1) int-signed-value-bounds
intval-mul.elims
mult-zero-right new-int.simps new-int-bin.simps take-bit-of-0 unfold-const
valid-stamp.simps(1) valid-value.simps(1) val-multiply-zero)

optimization MulNegate:  $x * -(\text{const } (\text{IntVal } b \ 1)) \mapsto -x$ 
apply auto using val-multiply-negative wf-value-def
by (smt (verit) Value.distinct(1) Value.sel(1) add.inverse-inverse intval-mul.elims
intval-negate.simps(1) mask-eq-take-bit-minus-one new-int.simps new-int-bin.simps
take-bit-dist-neg unary-eval.simps(2) unfold-unary val-multiply-negative
val-eliminate-redundant-negative)

fun isNonZero :: Stamp  $\Rightarrow$  bool where
isNonZero (IntegerStamp b lo hi) = (lo > 0) |
isNonZero - = False

lemma isNonZero-defn:
assumes isNonZero (stamp-expr x)
assumes wf-stamp x
shows ( $[m, p] \vdash x \mapsto v \longrightarrow (\exists vv \ b. (v = \text{IntVal } b \ vv \wedge \text{val-to-bool } \text{val}[(\text{IntVal } b \ 0) < v]))$ )
apply (rule impI) subgoal premises eval
proof -
obtain b lo hi where xstamp: stamp-expr x = IntegerStamp b lo hi
using assms

```

```

    by (meson isNonZero.elims(2))
  then obtain vv where vdef: v = IntVal b vv
    by (metis assms(2) eval valid-int wf-stamp-def)
  have lo > 0
    using assms(1) xstamp by force
  then have signed-above: int-signed-value b vv > 0
    using assms unfolding wf-stamp-def
    using eval vdef xstamp by fastforce
  have take-bit b vv = vv
    using eval eval-unused-bits-zero vdef by auto
  then have vv > 0
    using signed-above
    by (metis bit-take-bit-iff int-signed-value.simps not-less-zero signed-eq-0-iff
signed-take-bit-eq-if-positive take-bit-0 take-bit-of-0 verit-comp-simplify1(1) word-gt-0)
  then show ?thesis
    using vdef using signed-above
    by simp
qed
done

optimization MulPower2:  $x * y \mapsto x << \text{const } (\text{IntVal } 64 \ i)$ 
  when ( $i > 0 \wedge$ 
         $64 > i \wedge$ 
         $y = \text{exp}[\text{const } (\text{IntVal } 64 \ (2 \wedge \text{unat}(i)))]$ )

defer
apply simp apply (rule impI; (rule allI)+; rule impI)
subgoal premises eval for m p v
proof -
  obtain xv where xv:  $[m, p] \vdash x \mapsto xv$ 
    using eval(2) by blast
  then obtain xvv where xvv:  $xv = \text{IntVal } 64 \ xvv$ 
    using eval
    using ConstantExprE bin-eval.simps(2) evalDet intval-bits.simps intval-mul.elims
new-int-bin.simps unfold-binary
    by (smt (verit))
  obtain yv where yv:  $[m, p] \vdash y \mapsto yv$ 
    using eval(1) eval(2) by blast
  then have lhs:  $[m, p] \vdash \text{exp}[x * y] \mapsto \text{val}[xv * yv]$ 
    by (metis bin-eval.simps(2) eval(1) eval(2) evalDet unfold-binary xv)
  have  $[m, p] \vdash \text{exp}[\text{const } (\text{IntVal } 64 \ i)] \mapsto \text{val}[(\text{IntVal } 64 \ i)]$ 
    by (smt (verit, ccfv-SIG) ConstantExpr constantAsStamp.simps(1) eval-bits-1-64
take-bit64 validStampIntConst wf-value-def valid-value.simps(1) xv xvv)
  then have rhs:  $[m, p] \vdash \text{exp}[x << \text{const } (\text{IntVal } 64 \ i)] \mapsto \text{val}[xv << (\text{IntVal } 64 \ i)]$ 
    using xv xvv using evaltree.BinaryExpr
  by (metis Value.simps(5) bin-eval.simps(8) intval-left-shift.simps(1) new-int.simps)
  have  $\text{val}[xv * yv] = \text{val}[xv << (\text{IntVal } 64 \ i)]$ 
    using val-MulPower2
  by (metis ConstantExprE eval(1) evaltree-not-undef lhs yv)

```

```

    then show ?thesis
      by (metis eval(1) eval(2) evalDet lhs rhs)
qed
done

```

```
end
```

```
end
```

10.7 Experimental AndNode Phase

```
theory NewAnd
```

```
  imports
```

```
    Common
```

```
    Graph.Long
```

```
begin
```

```
lemma bin-distribute-and-over-or:
```

```
  bin[z & (x | y)] = bin[(z & x) | (z & y)]
```

```
  by (smt (verit, best) bit-and-iff bit-eqI bit-or-iff)
```

```
lemma intval-distribute-and-over-or:
```

```
  val[z & (x | y)] = val[(z & x) | (z & y)]
```

```
  apply (cases x; cases y; cases z; auto)
```

```
  using bin-distribute-and-over-or by blast+
```

```
lemma exp-distribute-and-over-or:
```

```
  exp[z & (x | y)] ≥ exp[(z & x) | (z & y)]
```

```
  apply simp using intval-distribute-and-over-or
```

```
  using BinaryExpr bin-eval.simps(4,5)
```

```
  using intval-or.simps(1) unfolding new-int-bin.simps new-int.simps apply auto
```

```
  by (metis bin-eval.simps(4) bin-eval.simps(5) intval-or.simps(2) intval-or.simps(5))
```

```
lemma intval-and-commute:
```

```
  val[x & y] = val[y & x]
```

```
  by (cases x; cases y; auto simp: and.commute)
```

```
lemma intval-or-commute:
```

```
  val[x | y] = val[y | x]
```

```
  by (cases x; cases y; auto simp: or.commute)
```

```
lemma intval-xor-commute:
```

```
  val[x ⊕ y] = val[y ⊕ x]
```

```
  by (cases x; cases y; auto simp: xor.commute)
```

```
lemma exp-and-commute:
```


$\text{exp}[x \& z] \geq \text{exp}[z \& x]$
apply *simp* **using** *intval-and-commute* **by** *auto*

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

lemma
  assumes  $y = 0$ 
  shows  $x + y = \text{or } x\ y$ 
  using assms
  by simp

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

```

```

context stamp-mask
begin

```

```

lemma intval-up-and-zero-implies-zero:
  assumes and ( $\uparrow x$ ) ( $\uparrow y$ ) = 0
  assumes  $[m, p] \vdash x \mapsto xv$ 
  assumes  $[m, p] \vdash y \mapsto yv$ 
  assumes  $val[xv \ \& \ yv] \neq \text{UndefVal}$ 
  shows  $\exists b. val[xv \ \& \ yv] = \text{new-int } b \ 0$ 
  using assms apply (cases xv; cases yv; auto)
  using up-mask-and-zero-implies-zero
  apply (smt (verit, best) take-bit-and take-bit-of-0)
  by presburger

lemma exp-eliminate-y:
  and ( $\uparrow y$ ) ( $\uparrow z$ ) = 0  $\longrightarrow$  BinaryExpr BinAnd (BinaryExpr BinOr x y) z  $\geq$  BinaryExpr BinAnd x z
  apply simp apply (rule impI; rule allI; rule allI; rule allI)
  subgoal premises p for m p v apply (rule impI) subgoal premises e
  proof –
    obtain xv where xv:  $[m, p] \vdash x \mapsto xv$ 
    using e by auto
    obtain yv where yv:  $[m, p] \vdash y \mapsto yv$ 
    using e by auto
    obtain zv where zv:  $[m, p] \vdash z \mapsto zv$ 
    using e by auto
    have lhs:  $v = val[(xv \mid yv) \ \& \ zv]$ 
    using xv yv zv
    by (smt (verit, best) BinaryExprE bin-eval.simps(4) bin-eval.simps(5) e evalDet)
    then have  $v = val[(xv \ \& \ zv) \mid (yv \ \& \ zv)]$ 
    by (simp add: intval-and-commute intval-distribute-and-over-or)
    also have  $\exists b. val[yv \ \& \ zv] = \text{new-int } b \ 0$ 
    using intval-up-and-zero-implies-zero
    by (metis calculation e intval-or.simps(5) p unfold-binary yv zv)
    ultimately have rhs:  $v = val[xv \ \& \ zv]$ 
    using intval-eliminate-y lhs by force
    from lhs rhs show ?thesis
    by (metis BinaryExpr BinaryExprE bin-eval.simps(4) e xv zv)
  qed
done
done

```

```

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

```

```

lemma above-nth-not-set:

```

```

fixes  $x :: \text{int64}$ 
assumes  $n = 64 - \text{numberOfLeadingZeros } x$ 
shows  $j > n \longrightarrow \neg(\text{bit } x \ j)$ 
using assms unfolding numberOfLeadingZeros-def
by (smt (verit, ccfv-SIG) highestOneBit-def int-nat-eq int-ops(6) less-imp-of-nat-less
max-set-bit size64 zerosAboveHighestOne)

```

no-notation *LogicNegationNotation* (!-)

```

lemma zero-horner:
  horner-sum of-bool 2 (map ( $\lambda x. \text{False}$ ) xs) = 0
apply (induction xs) apply simp
by force

```

```

lemma zero-map:
  assumes  $j \leq n$ 
  assumes  $\forall i. j \leq i \longrightarrow \neg(f \ i)$ 
  shows map f [ $0..<n$ ] = map f [ $0..<j$ ] @ map ( $\lambda x. \text{False}$ ) [ $j..<n$ ]
  apply (insert assms)
  by (smt (verit, del-insts) add-diff-inverse-nat atLeastLessThan-iff bot-nat-0.extremum
leD map-append map-eq-conv set-upt upt-add-eq-append)

```

```

lemma map-join-horner:
  assumes map f [ $0..<n$ ] = map f [ $0..<j$ ] @ map ( $\lambda x. \text{False}$ ) [ $j..<n$ ]
  shows horner-sum of-bool (2::a::len word) (map f [ $0..<n$ ]) = horner-sum of-bool
  2 (map f [ $0..<j$ ])
proof -
  have horner-sum of-bool (2::a::len word) (map f [ $0..<n$ ]) = horner-sum of-bool
  2 (map f [ $0..<j$ ]) + 2 ^ length [ $0..<j$ ] * horner-sum of-bool 2 (map f [ $j..<n$ ])
  using horner-sum-append
  by (smt (verit) assms diff-le-self diff-zero le-add-same-cancel2 length-append
length-map length-upt map-append upt-add-eq-append)
  also have ... = horner-sum of-bool 2 (map f [ $0..<j$ ]) + 2 ^ length [ $0..<j$ ] *
  horner-sum of-bool 2 (map ( $\lambda x. \text{False}$ ) [ $j..<n$ ])
  using assms
  by (metis calculation horner-sum-append length-map)
  also have ... = horner-sum of-bool 2 (map f [ $0..<j$ ])
  using zero-horner
  using mult-not-zero by auto
  finally show ?thesis by simp
qed

```

```

lemma split-horner:
  assumes  $j \leq n$ 
  assumes  $\forall i. j \leq i \longrightarrow \neg(f \ i)$ 
  shows horner-sum of-bool (2::a::len word) (map f [ $0..<n$ ]) = horner-sum of-bool
  2 (map f [ $0..<j$ ])
  apply (rule map-join-horner)
  apply (rule zero-map)

```

```

using assms by auto

lemma transfer-map:
  assumes  $\forall i. i < n \longrightarrow f\ i = f'\ i$ 
  shows  $(\text{map } f\ [0..<n]) = (\text{map } f'\ [0..<n])$ 
  using assms by simp

lemma transfer-horner:
  assumes  $\forall i. i < n \longrightarrow f\ i = f'\ i$ 
  shows  $\text{horner-sum of-bool } (2::'a::\text{len word})\ (\text{map } f\ [0..<n]) = \text{horner-sum of-bool}$ 
 $2\ (\text{map } f'\ [0..<n])$ 
  using assms using transfer-map
  by (smt (verit, best))

lemma L1:
  assumes  $n = 64 - \text{numberOfLeadingZeros } (\uparrow z)$ 
  assumes  $[m, p] \vdash z \mapsto \text{IntVal } b\ zv$ 
  shows  $\text{and } v\ zv = \text{and } (v \bmod 2^n)\ zv$ 
proof –
  have  $nle: n \leq 64$ 
  using assms
  using diff-le-self by blast
  also have  $\text{and } v\ zv = \text{horner-sum of-bool } 2\ (\text{map } (\text{bit } (\text{and } v\ zv))\ [0..<64])$ 
  using horner-sum-bit-eq-take-bit size64
  by (metis size-word.rep-eq take-bit-length-eq)
  also have  $\dots = \text{horner-sum of-bool } 2\ (\text{map } (\lambda i. \text{bit } (\text{and } v\ zv)\ i)\ [0..<64])$ 
  by blast
  also have  $\dots = \text{horner-sum of-bool } 2\ (\text{map } (\lambda i. ((\text{bit } v\ i) \wedge (\text{bit } zv\ i)))\ [0..<64])$ 
  using bit-and-iff by metis
  also have  $\dots = \text{horner-sum of-bool } 2\ (\text{map } (\lambda i. ((\text{bit } v\ i) \wedge (\text{bit } zv\ i)))\ [0..<n])$ 
proof –
  have  $\forall i. i \geq n \longrightarrow \neg(\text{bit } zv\ i)$ 
  using above-nth-not-set assms(1)
  using assms(2) not-may-implies-false
  by (smt (verit, ccfv-SIG) One-nat-def diff-less int-ops(6) leadingZerosAddHighestOne linorder-not-le nat-int-comparison(2) not-numeral-le-zero size64 zero-less-Suc zerosAboveHighestOne)
  then have  $\forall i. i \geq n \longrightarrow \neg((\text{bit } v\ i) \wedge (\text{bit } zv\ i))$ 
  by auto
  then show ?thesis using nle split-horner
  by (metis (no-types, lifting))
qed
  also have  $\dots = \text{horner-sum of-bool } 2\ (\text{map } (\lambda i. ((\text{bit } (v \bmod 2^n)\ i) \wedge (\text{bit } zv\ i)))\ [0..<n])$ 
proof –
  have  $\forall i. i < n \longrightarrow \text{bit } (v \bmod 2^n)\ i = \text{bit } v\ i$ 
  by (metis bit-take-bit-iff take-bit-eq-mod)
  then have  $\forall i. i < n \longrightarrow ((\text{bit } v\ i) \wedge (\text{bit } zv\ i)) = ((\text{bit } (v \bmod 2^n)\ i) \wedge (\text{bit } zv\ i))$ 

```

```

    by force
  then show ?thesis
    by (rule transfer-horner)
qed
also have ... = horner-sum of-bool 2 (map (λi. ((bit (v mod 2^n) i) ∧ (bit zv
i))) [0..<64])
proof -
  have ∀ i. i ≥ n ⟶ ¬(bit zv i)
  using above-nth-not-set assms(1)
  using assms(2) not-may-implies-false
  by (smt (verit, ccfv-SIG) One-nat-def diff-less int-ops(6) leadingZerosAddHigh-
estOne linorder-not-le nat-int-comparison(2) not-numeral-le-zero size64 zero-less-Suc
zerosAboveHighestOne)
  then show ?thesis
    by (metis (no-types, lifting) assms(1) diff-le-self split-horner)
qed
also have ... = horner-sum of-bool 2 (map (bit (and (v mod 2^n) zv)) [0..<64])
  by (meson bit-and-iff)
also have ... = and (v mod 2^n) zv
  using horner-sum-bit-eq-take-bit size64
  by (metis size-word.rep-eq take-bit-length-eq)
finally show ?thesis
  using ⟨and (v::64 word) (zv::64 word) = horner-sum of-bool (2::64 word)
(map (bit (and v zv)) [0::nat..<64::nat])⟩ ⟨horner-sum of-bool (2::64 word) (map
(λi::nat. bit ((v::64 word) mod (2::64 word) ^ (n::nat)) i ∧ bit (zv::64 word) i)
[0::nat..<64::nat]) = horner-sum of-bool (2::64 word) (map (bit (and (v mod (2::64
word) ^ n) zv)) [0::nat..<64::nat])⟩ ⟨horner-sum of-bool (2::64 word) (map (λi::nat.
bit ((v::64 word) mod (2::64 word) ^ (n::nat)) i ∧ bit (zv::64 word) i) [0::nat..<n])
= horner-sum of-bool (2::64 word) (map (λi::nat. bit (v mod (2::64 word) ^ n) i
∧ bit zv i) [0::nat..<64::nat])⟩ ⟨horner-sum of-bool (2::64 word) (map (λi::nat. bit
(v::64 word) i ∧ bit (zv::64 word) i) [0::nat..<64::nat]) = horner-sum of-bool (2::64
word) (map (λi::nat. bit v i ∧ bit zv i) [0::nat..<n::nat])⟩ ⟨horner-sum of-bool (2::64
word) (map (λi::nat. bit (v::64 word) i ∧ bit (zv::64 word) i) [0::nat..<n::nat]) =
horner-sum of-bool (2::64 word) (map (λi::nat. bit (v mod (2::64 word) ^ n) i ∧
bit zv i) [0::nat..<n])⟩ ⟨horner-sum of-bool (2::64 word) (map (bit (and ((v::64
word) mod (2::64 word) ^ (n::nat)) (zv::64 word))) [0::nat..<64::nat]) = and (v
mod (2::64 word) ^ n) zv⟩ ⟨horner-sum of-bool (2::64 word) (map (bit (and (v::64
word) (zv::64 word))) [0::nat..<64::nat]) = horner-sum of-bool (2::64 word) (map
(λi::nat. bit v i ∧ bit zv i) [0::nat..<64::nat])⟩ by presburger
qed

```

lemma *up-mask-upper-bound*:

```

  assumes [m, p] ⊢ x ↦ IntVal b xv
  shows xv ≤ (↑x)
  using assms
  by (metis (no-types, lifting) and.idem and.right-neutral bit.conj-cancel-left bit.conj-disj-distrib(1)
bit.double-compl ucast-id up-spec word-and-le1 word-not-dist(2))

```

lemma *L2*:

```

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

thm-oracles L1 L2

lemma unfold-binary-width-add:
shows  $([m,p] \vdash \text{BinaryExpr BinAdd } xe \text{ } ye \mapsto \text{IntVal } b \text{ } val) = (\exists x \ y.$ 
 $(([m,p] \vdash xe \mapsto \text{IntVal } b \text{ } x) \wedge$ 
 $([m,p] \vdash ye \mapsto \text{IntVal } b \text{ } y) \wedge$ 
 $(\text{IntVal } b \text{ } val = \text{bin-eval BinAdd } (\text{IntVal } b \text{ } x) (\text{IntVal } b \text{ } y)) \wedge$ 
 $(\text{IntVal } b \text{ } val \neq \text{UndefVal})$ 
 $)) \text{ (is } ?L = ?R)$ 
proof (intro iffI)
  assume  $?L$ 
  show  $?R$  apply (rule evaltree.cases[OF ?L])
    apply force+ apply auto[1]
    apply (smt (verit) intval-add.elims intval-bits.simps)
    by blast
next
  assume  $R$ :  $?R$ 
  then obtain  $x \ y$  where  $[m,p] \vdash xe \mapsto \text{IntVal } b \text{ } x$ 
    and  $[m,p] \vdash ye \mapsto \text{IntVal } b \text{ } y$ 

```

```

    and new-int b val = bin-eval BinAdd (IntVal b x) (IntVal b y)
    and new-int b val ≠ UndefVal
  by auto
then show ?L
  using R by blast
qed

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

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

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

lemma improved-opt:
  assumes numberOfLeadingZeros (↑z) + numberOfTrailingZeros (↑y) ≥ 64
  shows exp[(x + y) & z] ≥ exp[x & z]
  apply simp apply ((rule allI)+; rule impI)

```



```

subgoal premises eval for m p v
proof -
  obtain n where n: n = 64 - numberOfLeadingZeros ( $\uparrow z$ )
  by simp
  obtain b val where val:  $[m, p] \vdash \text{exp}[(x + y) \ \& \ z] \mapsto \text{IntVal } b \text{ val}$ 
  by (metis BinaryExprE bin-eval-new-int eval new-int.simps)
  then obtain xv yv where addv:  $[m, p] \vdash \text{exp}[x + y] \mapsto \text{IntVal } b \ (xv + yv)$ 
  apply (subst (asm) unfold-binary-width-and) by (metis add.right-neutral)
  then obtain yv where yv:  $[m, p] \vdash y \mapsto \text{IntVal } b \ yv$ 
  apply (subst (asm) unfold-binary-width-add) by blast
  from addv obtain xv where xv:  $[m, p] \vdash x \mapsto \text{IntVal } b \ xv$ 
  apply (subst (asm) unfold-binary-width-add) by blast
  from val obtain zv where zv:  $[m, p] \vdash z \mapsto \text{IntVal } b \ zv$ 
  apply (subst (asm) unfold-binary-width-and) by blast
  have addv:  $[m, p] \vdash \text{exp}[x + y] \mapsto \text{new-int } b \ (xv + yv)$ 
  apply (rule evaltree.BinaryExpr)
  using xv apply simp
  using yv apply simp
  by simp+
  have lhs:  $[m, p] \vdash \text{exp}[(x + y) \ \& \ z] \mapsto \text{new-int } b \ (\text{and } (xv + yv) \ zv)$ 
  apply (rule evaltree.BinaryExpr)
  using addv apply simp
  using zv apply simp
  using addv apply auto[1]
  by simp
  have rhs:  $[m, p] \vdash \text{exp}[x \ \& \ z] \mapsto \text{new-int } b \ (\text{and } xv \ zv)$ 
  apply (rule evaltree.BinaryExpr)
  using xv apply simp
  using zv apply simp
  apply force
  by simp
  then show ?thesis
  proof (cases numberOfLeadingZeros ( $\uparrow z$ ) > 0)
  case True
  have n-bounds:  $0 \leq n \wedge n < 64$ 
  using diff-le-self n numberOfLeadingZeros-range
  by (simp add: True)
  have and  $(xv + yv) \ zv = \text{and } ((xv + yv) \bmod 2^n) \ zv$ 
  using L1 n zv by blast
  also have  $\dots = \text{and } ((xv + (yv \bmod 2^n)) \bmod 2^n) \ zv$ 
  using mod-dist-over-add-right n-bounds
  by (metis take-bit-0 take-bit-eq-mod zero-less-iff-neq-zero)
  also have  $\dots = \text{and } (((xv \bmod 2^n) + (yv \bmod 2^n)) \bmod 2^n) \ zv$ 
  by (metis bits-mod-by-1 mod-dist-over-add n-bounds order-le-imp-less-or-eq
power-0)
  also have  $\dots = \text{and } ((xv \bmod 2^n) \bmod 2^n) \ zv$ 
  using L2 n zv yv
  using assms by auto
  also have  $\dots = \text{and } (xv \bmod 2^n) \ zv$ 

```

```

    using mod-mod-trivial
  by (smt (verit, best) and.idem take-bit-eq-mask take-bit-eq-mod word-bw-assocs(1))
  also have ... = and xv zv
    using L1 n zv by metis
  finally show ?thesis
    using eval lhs rhs
    by (metis evalDet)
next
case False
then have numberOfLeadingZeros ( $\uparrow z$ ) = 0
  by simp
then have numberOfTrailingZeros ( $\uparrow y$ )  $\geq$  64
  using assms(1)
  by fastforce
then have yv = 0
  using yv
  by (metis (no-types, lifting) L1 L2 add-diff-cancel-left' and.comm-neutral
and.idem bit.compl-zero bit.conj-cancel-right bit.conj-disj-distrib(1) bit.double-compl
less-imp-diff-less linorder-not-le word-not-dist(2))
  then show ?thesis
    by (metis add.right-neutral eval evalDet lhs rhs)
qed
qed
done

thm-oracles improved-opt

```

end

```

phase NewAnd
  terminating size
begin

```

```

optimization redundant-lhs-y-or:  $((x \mid y) \& z) \mapsto x \& z$ 
  when (((and (IRExpr-up y) (IRExpr-up z)) = 0))
  apply (simp add: IRExpr-up-def)
  using simple-mask.exp-eliminate-y by blast

```

```

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

```

```

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

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

```

end

end

10.8 NotNode Phase

```

theory NotPhase
  imports
    Common
begin

  phase NotNode
    terminating size
begin

```

```

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

```

```

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

```

```

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

```

Optimisations

optimization *NotCancel*: $\text{exp}[\sim(\sim a)] \mapsto a$
by (*metis exp-not-cancel*)

end

end

10.9 OrNode Phase

theory *OrPhase*

imports

Common

begin

context *stamp-mask*

begin

Taking advantage of the truth table of or operations.

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

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

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

lemma *OrLeftFallthrough*:

assumes (*and* (*not* ($\downarrow x$)) ($\uparrow y$)) = 0

shows $\text{exp}[x \mid y] \geq \text{exp}[x]$

using *assms*

apply *simp* **apply** ((*rule allI*)₊; *rule impI*)

subgoal **premises** *eval* **for** *m p v*

proof –

obtain *b vv* **where** *e*: $[m, p] \vdash \text{exp}[x \mid y] \mapsto \text{IntVal } b \ vv$

using *eval*

by (*metis BinaryExprE bin-eval-new-int new-int.simps*)

from *e* **obtain** *xv* **where** *xv*: $[m, p] \vdash x \mapsto \text{IntVal } b \ xv$

apply (*subst* (*asm*) *unfold-binary-width*)

by *force*₊

from *e* **obtain** *yv* **where** *yv*: $[m, p] \vdash y \mapsto \text{IntVal } b \ yv$

apply (*subst* (*asm*) *unfold-binary-width*)

by *force*₊

have *vdef*: $v = \text{intval-or } (\text{IntVal } b \ xv) \ (\text{IntVal } b \ yv)$

using *e xv yv*

by (*metis bin-eval.simps*(5) *eval*(2) *evalDet* *unfold-binary*)

```

have  $\forall i. (bit\ xv\ i) \mid (bit\ yv\ i) = (bit\ xv\ i)$ 
  by (metis assms bit-and-iff not-down-up-mask-and-zero-implies-zero xv yv)
then have IntVal b xv = intval-or (IntVal b xv) (IntVal b yv)
  by (smt (verit, ccfv-threshold) and.idem assms bit.conj-disj-distrib eval-unused-bits-zero
intval-or.simps(1) new-int.simps new-int-bin.simps not-down-up-mask-and-zero-implies-zero
word-ao-absorbs(3) xv yv)
  then show ?thesis
    using vdef
    using xv by presburger
qed
done

```

lemma *OrRightFallthrough*:

```

assumes (and (not ( $\downarrow y$ )) ( $\uparrow x$ )) = 0
shows exp[x | y]  $\geq$  exp[y]
using assms
apply simp apply ((rule allI) $+$ ; rule impI)
subgoal premises eval for m p v
proof –
  obtain b vv where e: [m, p]  $\vdash$  exp[x | y]  $\mapsto$  IntVal b vv
    using eval
    by (metis BinaryExprE bin-eval-new-int new-int.simps)
  from e obtain xv where xv: [m, p]  $\vdash$  x  $\mapsto$  IntVal b xv
    apply (subst (asm) unfold-binary-width)
    by force $+$ 
  from e obtain yv where yv: [m, p]  $\vdash$  y  $\mapsto$  IntVal b yv
    apply (subst (asm) unfold-binary-width)
    by force $+$ 
  have vdef: v = intval-or (IntVal b xv) (IntVal b yv)
    using e xv yv
    by (metis bin-eval.simps(5) eval(2) evalDet unfold-binary)
  have  $\forall i. (bit\ xv\ i) \mid (bit\ yv\ i) = (bit\ yv\ i)$ 
    by (metis assms bit-and-iff not-down-up-mask-and-zero-implies-zero xv yv)
  then have IntVal b yv = intval-or (IntVal b xv) (IntVal b yv)
    by (metis (no-types, lifting) assms eval-unused-bits-zero intval-or.simps(1)
new-int.elims new-int-bin.elims stamp-mask.not-down-up-mask-and-zero-implies-zero
stamp-mask-axioms word-ao-absorbs(8) xv yv)
  then show ?thesis
    using vdef
    using yv by presburger
qed
done

```

end

phase *OrNode*

terminating *size*

begin

lemma *bin-or-equal*:

$\text{bin}[x \mid x] = \text{bin}[x]$

by *simp*

lemma *bin-shift-const-right-helper*:

$x \mid y = y \mid x$

by *simp*

lemma *bin-or-not-operands*:

$(\sim x \mid \sim y) = (\sim(x \ \& \ y))$

by *simp*

lemma *val-or-equal*:

assumes $x = \text{new-int } b \ v$

and $(\text{val}[x \mid x] \neq \text{UndefVal})$

shows $\text{val}[x \mid x] = \text{val}[x]$

apply (*cases* x ; *auto*) **using** *bin-or-equal* *assms*

by *auto*+

lemma *val-elim-redundant-false*:

assumes $x = \text{new-int } b \ v$

and $\text{val}[x \mid \text{false}] \neq \text{UndefVal}$

shows $\text{val}[x \mid \text{false}] = \text{val}[x]$

using *assms* **apply** (*cases* x ; *auto*) **by** *presburger*

lemma *val-shift-const-right-helper*:

$\text{val}[x \mid y] = \text{val}[y \mid x]$

apply (*cases* x ; *cases* y ; *auto*)

by (*simp* *add: or.commute*)+

lemma *val-or-not-operands*:

$\text{val}[\sim x \mid \sim y] = \text{val}[\sim(x \ \& \ y)]$

apply (*cases* x ; *cases* y ; *auto*)

by (*simp* *add: take-bit-not-take-bit*)

lemma *exp-or-equal*:

$\text{exp}[x \mid x] \geq \text{exp}[x]$

using *val-or-equal* **apply** *auto*

by (*smt* (*verit*, *ccfv-SIG*) *evalDet* *eval-unused-bits-zero* *intval-negate.elims* *int-val-or.simps*(2)

intval-or.simps(6) *intval-or.simps*(7) *new-int.simps* *val-or-equal*)

lemma *exp-elim-redundant-false*:

$\text{exp}[x \mid \text{false}] \geq \text{exp}[x]$

using *val-elim-redundant-false* **apply** *auto*

by (*smt* (*verit*) *Value.sel*(1) *eval-unused-bits-zero* *intval-or.elims* *new-int.simps*

new-int-bin.simps val-elim-redundant-false)

Optimisations

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

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

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

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

optimization *OrLeftFallthrough*:
 $x \mid y \mapsto x$ when $((\text{and } (\text{not } (\text{IExpr-down } x)) (\text{IExpr-up } y))) = 0$
using *simple-mask.OrLeftFallthrough* **by** *blast*

optimization *OrRightFallthrough*:
 $x \mid y \mapsto y$ when $((\text{and } (\text{not } (\text{IExpr-down } y)) (\text{IExpr-up } x))) = 0$
using *simple-mask.OrRightFallthrough* **by** *blast*

end

end

10.10 ShiftNode Phase

theory *ShiftPhase*

imports

Common

begin

phase *ShiftNode*

terminating *size*

begin

fun *intval-log2* :: *Value* \Rightarrow *Value* **where**
intval-log2 (*IntVal* *b v*) = *IntVal* *b* (*word-of-int* (*SOME* *e. v=2^e*)) |
intval-log2 - = *UndefVal*

```

fun in-bounds :: Value  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  bool where
  in-bounds (IntVal b v) l h = (l < sint v  $\wedge$  sint v < h) |
  in-bounds - l h = False

lemma
  assumes in-bounds (intval-log2 val-c) 0 32
  shows intval-left-shift x (intval-log2 val-c) = intval-mul x val-c
  apply (cases val-c; auto) using intval-left-shift.simps(1) intval-mul.simps(1) intval-log2.simps(1)
  sorry

lemma e-intval:
  n = intval-log2 val-c  $\wedge$  in-bounds n 0 32  $\longrightarrow$ 
    intval-left-shift x (intval-log2 val-c) =
      intval-mul x val-c
proof (rule impI)
  assume n = intval-log2 val-c  $\wedge$  in-bounds n 0 32
  show intval-left-shift x (intval-log2 val-c) =
    intval-mul x val-c
  proof (cases  $\exists$  v . val-c = IntVal 32 v)
    case True
      obtain vc where val-c = IntVal 32 vc
      using True by blast
      then have n = IntVal 32 (word-of-int (SOME e. vc=2e))
      using  $\langle$ n = intval-log2 val-c  $\wedge$  in-bounds n 0 32 $\rangle$  intval-log2.simps(1) by
        presburger
      then show ?thesis sorry
    next
      case False
      then have  $\exists$  v . val-c = IntVal 64 v
      sorry
      then obtain vc where val-c = IntVal 64 vc
      by auto
      then have n = IntVal 64 (word-of-int (SOME e. vc=2e))
      using  $\langle$ n = intval-log2 val-c  $\wedge$  in-bounds n 0 32 $\rangle$  intval-log2.simps(1) by
        presburger
      then show ?thesis sorry
  qed
qed

optimization e:
  x * (const c)  $\longmapsto$  x << (const n) when (n = intval-log2 c  $\wedge$  in-bounds n 0 32)
  using e-intval
  using BinaryExprE ConstantExprE bin-eval.simps(2,7) sorry

end

```


end

10.11 SignedDivNode Phase

theory *SignedDivPhase*

imports

Common

begin

phase *SignedDivNode*

terminating *size*

begin

lemma *val-division-by-one-is-self-32*:

assumes $x = \text{new-int } 32 \ v$

shows $\text{intval-div } x \ (\text{IntVal } 32 \ 1) = x$

using *assms* **apply** (*cases* x ; *auto*)

by (*simp add: take-bit-signed-take-bit*)

end

end

10.12 SignedRemNode Phase

theory *SignedRemPhase*

imports

Common

begin

phase *SignedRemNode*

terminating *size*

begin

lemma *val-remainder-one*:

assumes $\text{intval-mod } x \ (\text{IntVal } 32 \ 1) \neq \text{UndefVal}$

shows $\text{intval-mod } x \ (\text{IntVal } 32 \ 1) = \text{IntVal } 32 \ 0$

using *assms* **apply** (*cases* x ; *auto*) **sorry**

value *word-of-int* (*sint* ($x2::32 \ \text{word}$) *smod* 1)

end

end

10.13 SubNode Phase

```
theory SubPhase
  imports
    Common
    Proofs.StampEvalThms
begin

phase SubNode
  terminating size
begin

lemma bin-sub-after-right-add:
  shows  $((x :: ('a :: len) \text{ word}) + (y :: ('a :: len) \text{ word})) - y = x$ 
  by simp

lemma sub-self-is-zero:
  shows  $(x :: ('a :: len) \text{ word}) - x = 0$ 
  by simp

lemma bin-sub-then-left-add:
  shows  $(x :: ('a :: len) \text{ word}) - (x + (y :: ('a :: len) \text{ word})) = -y$ 
  by simp

lemma bin-sub-then-left-sub:
  shows  $(x :: ('a :: len) \text{ word}) - (x - (y :: ('a :: len) \text{ word})) = y$ 
  by simp

lemma bin-subtract-zero:
  shows  $(x :: 'a :: len \text{ word}) - (0 :: 'a :: len \text{ word}) = x$ 
  by simp

lemma bin-sub-negative-value:
   $(x :: ('a :: len) \text{ word}) - (-(y :: ('a :: len) \text{ word})) = x + y$ 
  by simp

lemma bin-sub-self-is-zero:
   $(x :: ('a :: len) \text{ word}) - x = 0$ 
  by simp

lemma bin-sub-negative-const:
   $(x :: 'a :: len \text{ word}) - (-(y :: 'a :: len \text{ word})) = x + y$ 
  by simp

lemma val-sub-after-right-add-2:
  assumes  $x = \text{new-int } b \ v$ 
  assumes  $\text{val}[(x + y) - y] \neq \text{UndefVal}$ 
  shows  $\text{val}[(x + y) - y] = \text{val}[x]$ 
```

```

using bin-sub-after-right-add
using assms apply (cases x; cases y; auto)
by (metis (full-types) intval-sub.simps(2))

lemma val-sub-after-left-sub:
  assumes  $\text{val}[(x - y) - x] \neq \text{UndefVal}$ 
  shows  $\text{val}[(x - y) - x] = \text{val}[-y]$ 
  using assms apply (cases x; cases y; auto)
  using intval-sub.elims by fastforce

lemma val-sub-then-left-sub:
  assumes  $y = \text{new-int } b \ v$ 
  assumes  $\text{val}[x - (x - y)] \neq \text{UndefVal}$ 
  shows  $\text{val}[x - (x - y)] = \text{val}[y]$ 
  using assms apply (cases x; cases y; auto)
  by (metis (mono-tags) intval-sub.simps(5))

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

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

lemma val-sub-then-left-add:
  assumes  $\text{val}[x - (x + y)] \neq \text{UndefVal}$ 
  shows  $\text{val}[x - (x + y)] = \text{val}[-y]$ 
  using assms apply (cases x; cases y; auto)
  by (metis (mono-tags, lifting) intval-sub.simps(5))

lemma val-sub-negative-value:
  assumes  $\text{val}[x - (-y)] \neq \text{UndefVal}$ 
  shows  $\text{val}[x - (-y)] = \text{val}[x + y]$ 
  using assms by (cases x; cases y; auto)

lemma val-sub-self-is-zero:
  assumes  $x = \text{new-int } b \ v \wedge \text{val}[x - x] \neq \text{UndefVal}$ 
  shows  $\text{val}[x - x] = \text{new-int } b \ 0$ 
  using assms by (cases x; auto)

lemma val-sub-negative-const:
  assumes  $y = \text{new-int } b \ v \wedge \text{val}[x - (-y)] \neq \text{UndefVal}$ 
  shows  $\text{val}[x - (-y)] = \text{val}[x + y]$ 
  using assms by (cases x; cases y; auto)

```

```

lemma exp-sub-after-right-add:
  shows  $\exp[(x + y) - y] \geq \exp[x]$ 
  apply auto using val-sub-after-right-add-2
  using evalDet eval-unused-bits-zero intval-add.elims new-int.simps
  by (smt (verit))

lemma exp-sub-after-right-add2:
  shows  $\exp[(x + y) - x] \geq \exp[y]$ 
  using exp-sub-after-right-add apply auto
  using bin-eval.simps(1) bin-eval.simps(3) intval-add-sym unfold-binary
  by (smt (z3) Value.inject(1) diff-eq-eq evalDet eval-unused-bits-zero intval-add.elims

    intval-sub.elims new-int.simps new-int-bin.simps take-bit-dist-subL)

lemma exp-sub-negative-value:
   $\exp[x - (-y)] \geq \exp[x + y]$ 
  apply simp using val-sub-negative-value
  by (smt (verit) bin-eval.simps(1) bin-eval.simps(3) evaltree-not-undef
    unary-eval.simps(2) unfold-binary unfold-unary)

lemma exp-sub-then-left-sub:
  shows  $\exp[x - (x - y)] \geq \exp[y]$ 
  using val-sub-then-left-sub apply auto
  subgoal premises p for m p xa xaa ya
  proof–
    obtain xa where xa:  $[m, p] \vdash x \mapsto xa$ 
    using p(2) by blast
    obtain ya where ya:  $[m, p] \vdash y \mapsto ya$ 
    using p(5) by auto
    obtain xaa where xaa:  $[m, p] \vdash x \mapsto xaa$ 
    using p(2) by blast
    have 1:  $\text{val}[xa - (xaa - ya)] \neq \text{UndefVal}$ 
    by (metis evalDet p(2) p(3) p(4) p(5) xa xaa ya)
    then have  $\text{val}[xaa - ya] \neq \text{UndefVal}$ 
    by auto
    then have  $[m, p] \vdash y \mapsto \text{val}[xa - (xaa - ya)]$ 
    by (metis 1 Value.exhaust evalDet eval-unused-bits-zero evaltree-not-undef
      intval-sub.simps(6) intval-sub.simps(7) new-int.simps p(5) val-sub-then-left-sub xa
      xaa ya)
    then show ?thesis
    by (metis evalDet p(2) p(4) p(5) xa xaa ya)
  qed
done

```

thm-oracles *exp-sub-then-left-sub*

Optimisations

optimization *SubAfterAddRight*: $((x + y) - y) \mapsto x$
using *exp-sub-after-right-add* **by** *blast*

optimization *SubAfterAddLeft*: $((x + y) - x) \mapsto y$
using *exp-sub-after-right-add2* **by** *blast*

optimization *SubAfterSubLeft*: $((x - y) - x) \mapsto -y$
apply (*metis Suc-lessI add-2-eq-Suc' add-less-cancel-right less-trans-Suc not-add-less1 size-binary-const size-binary-lhs size-binary-rhs size-non-add*)
apply *auto*
by (*metis evalDet unary-eval.simps(2) unfold-unary val-sub-after-left-sub*)

optimization *SubThenAddLeft*: $(x - (x + y)) \mapsto -y$
apply *auto*
by (*metis evalDet unary-eval.simps(2) unfold-unary val-sub-then-left-add*)

optimization *SubThenAddRight*: $(y - (x + y)) \mapsto -x$
apply *auto*
by (*metis evalDet intval-add-sym unary-eval.simps(2) unfold-unary val-sub-then-left-add*)

optimization *SubThenSubLeft*: $(x - (x - y)) \mapsto y$
using *size-simps* **apply** *simp*
using *exp-sub-then-left-sub* **by** *blast*

optimization *SubtractZero*: $(x - (\text{const IntVal } b \ 0)) \mapsto x$
apply *auto*
by (*smt (verit) add.right-neutral diff-add-cancel eval-unused-bits-zero intval-sub.elims*
intval-word.simps new-int.simps new-int-bin.simps)

thm-oracles *SubtractZero*

optimization *SubNegativeValue*: $(x - (-y)) \mapsto x + y$
apply (*metis add-2-eq-Suc' less-SucI less-add-Suc1 not-less-eq size-binary-const size-non-add*)
using *exp-sub-negative-value* **by** *simp*

thm-oracles *SubNegativeValue*

lemma *negate-idempotent*:
assumes $x = \text{IntVal } b \ v \wedge \text{take-bit } b \ v = v$
shows $x = \text{val}[-(-x)]$
using *assms*
using *is-IntVal-def* **by** *force*

```

optimization ZeroSubtractValue:  $((\text{const IntVal } b \ 0) - x) \mapsto (-x)$ 
                                when  $(\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \ \text{lo}$ 
                                 $hi \wedge \neg(\text{is-ConstantExpr } x))$ 
  defer
  apply auto unfolding wf-stamp-def
  apply (smt (verit) diff-0 intval-negate.simps(1) intval-sub.elims intval-word.simps

                                new-int-bin.simps unary-eval.simps(2) unfold-unary)
  using add-2-eq-Suc' size.simps(2) size-flip-binary by presburger

```

```

optimization SubSelfIsZero:  $(x - x) \mapsto \text{const IntVal } b \ 0$  when
                                 $(\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \ \text{lo } hi)$ 
  apply simp-all
  apply auto
  using IRExpr.disc(42) One-nat-def size-non-const apply presburger
  by (smt (verit, best) wf-value-def ConstantExpr evalDet eval-bits-1-64 eval-unused-bits-zero
  new-int.simps take-bit-of-0 val-sub-self-is-zero validDefIntConst valid-int wf-stamp-def)

end

end

```

10.14 XorNode Phase

```

theory XorPhase
  imports
    Common
    Proofs.StampEvalThms
  begin

  phase XorNode
    terminating size
  begin

```

```

lemma bin-xor-self-is-false:
   $\text{bin}[x \oplus x] = 0$ 
  by simp

```

```

lemma bin-xor-commute:
   $\text{bin}[x \oplus y] = \text{bin}[y \oplus x]$ 
  by (simp add: xor.commute)

```

lemma *bin-eliminate-redundant-false:*

bin[$x \oplus 0$] = *bin*[x]

by *simp*

lemma *val-xor-self-is-false:*

assumes *val*[$x \oplus x$] \neq *UndefVal*

shows *val-to-bool* (*val*[$x \oplus x$]) = *False*

using *assms* **by** (*cases* x ; *auto*)

lemma *val-xor-self-is-false-2:*

assumes (*val*[$x \oplus x$]) \neq *UndefVal*

and $x = \text{IntVal } 32 \ v$

shows *val*[$x \oplus x$] = *bool-to-val False*

using *assms* **by** (*cases* x ; *auto*)

lemma *val-xor-self-is-false-3:*

assumes *val*[$x \oplus x$] \neq *UndefVal* $\wedge x = \text{IntVal } 64 \ v$

shows *val*[$x \oplus x$] = *IntVal 64 0*

using *assms* **by** (*cases* x ; *auto*)

lemma *val-xor-commute:*

val[$x \oplus y$] = *val*[$y \oplus x$]

apply (*cases* x ; *cases* y ; *auto*)

by (*simp add: xor.commute*)⁺

lemma *val-eliminate-redundant-false:*

assumes $x = \text{new-int } b \ v$

assumes *val*[$x \oplus (\text{bool-to-val False})$] \neq *UndefVal*

shows *val*[$x \oplus (\text{bool-to-val False})$] = x

using *assms* **apply** (*cases* x ; *auto*)

by *meson*

lemma *exp-xor-self-is-false:*

assumes *wf-stamp* $x \wedge \text{stamp-expr } x = \text{default-stamp}$

shows *exp*[$x \oplus x$] \geq *exp*[*false*]

using *assms* **apply** *auto* **unfolding** *wf-stamp-def*

using *IntVal0 Value.inject(1) bool-to-val.simps(2) constantAsStamp.simps(1) evalDet*

int-signed-value-bounds new-int.simps unfold-const val-xor-self-is-false-2

valid-int

valid-stamp.simps(1) valid-value.simps(1) wf-value-def

by (*smt (z3) validDefIntConst*)

lemma *exp-eliminate-redundant-false:*

shows *exp*[$x \oplus \text{false}$] \geq *exp*[x]

```

using val-eliminate-redundant-false apply auto
subgoal premises p for m p xa
proof –
  obtain xa where xa:  $[m, p] \vdash x \mapsto xa$ 
  using p(2) by blast
  then have  $val[xa \oplus (IntVal\ 32\ 0)] \neq UndefinedVal$ 
  using evalDet p(2) p(3) by blast
  then have  $[m, p] \vdash x \mapsto val[xa \oplus (IntVal\ 32\ 0)]$ 
  apply (cases xa; auto) using eval-unused-bits-zero xa by auto
  then show ?thesis
  using evalDet p(2) xa by blast
qed
done

```

Optimisations

```

optimization XorSelfIsFalse:  $(x \oplus x) \mapsto false$  when
   $(wf\_stamp\ x \wedge stamp\_expr\ x = default\_stamp)$ 
  using size-non-const apply force
  using exp-xor-self-is-false by auto

optimization XorShiftConstantRight:  $((const\ x) \oplus y) \mapsto y \oplus (const\ x)$  when
 $\neg(is\_ConstantExpr\ y)$ 
  using size-flip-binary apply force
  unfolding le-expr-def using val-xor-commute
  by auto

optimization EliminateRedundantFalse:  $(x \oplus false) \mapsto x$ 
  using exp-eliminate-redundant-false by blast

```

end

end

11 Conditional Elimination Phase

```

theory ConditionalElimination
  imports
    Proofs.Rewrites
    Proofs.Bisimulation
begin

```


11.1 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` :: `IRGraph` \Rightarrow `IRNode` \Rightarrow `IRNode` \Rightarrow `TriState` \Rightarrow `bool`

(- \vdash - & - \rightarrow -) **for** `g` **where**

$\llbracket \neg(g \vdash a \ \& \ b \hookrightarrow \text{imp}) \rrbracket \implies (g \vdash a \ \& \ b \rightarrow \text{Unknown}) \mid$

$\llbracket (g \vdash a \ \& \ b \hookrightarrow \text{imp}) \rrbracket \implies (g \vdash a \ \& \ b \rightarrow \text{imp})$

inductive `implies-tree` :: `IRExpr` \Rightarrow `IRExpr` \Rightarrow `bool` \Rightarrow `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:

```

  (BinaryExpr BinIntegerEquals x y) & (BinaryExpr BinIntegerLessThan y x)  $\hookrightarrow$ 
  False |
  less-imp-rev-less:
  (BinaryExpr BinIntegerLessThan x y) & (BinaryExpr BinIntegerLessThan y x)
 $\hookrightarrow$  False |
  less-imp-not-eq:
  (BinaryExpr BinIntegerLessThan x y) & (BinaryExpr BinIntegerEquals x y)  $\hookrightarrow$ 
  False |
  less-imp-not-eq-rev:
  (BinaryExpr BinIntegerLessThan x y) & (BinaryExpr BinIntegerEquals y x)  $\hookrightarrow$ 
  False |

```

```

x-imp-x:
x & x  $\hookrightarrow$  True |

```

```

negate-false:
[[x & y  $\hookrightarrow$  True]]  $\implies$  x & (UnaryExpr UnaryLogicNegation y)  $\hookrightarrow$  False |
negate-true:
[[x & y  $\hookrightarrow$  False]]  $\implies$  x & (UnaryExpr UnaryLogicNegation y)  $\hookrightarrow$  True

```

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

experiment begin

lemma *logic-negate-type*:

```

  assumes [m, p]  $\vdash$  UnaryExpr UnaryLogicNegation x  $\mapsto$  v
  assumes v  $\neq$  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]  $\vdash$  UnaryExpr UnaryLogicNegation x  $\mapsto$  unary-eval UnaryLogicNegation ve
  by (metis UnaryExprE assms(1) evalDet)
  then show ?thesis using assms unary-eval.elims evalDet ve IRUnaryOp.distinct
  sorry
qed

```

lemma *logic-negation-relation-tree*:

```

  assumes [m, p]  $\vdash$  y  $\mapsto$  val
  assumes [m, p]  $\vdash$  UnaryExpr UnaryLogicNegation y  $\mapsto$  invval
  assumes invval  $\neq$  UndefVal
  shows val-to-bool val  $\longleftrightarrow$   $\neg$ (val-to-bool invval)
proof –
  obtain v where invval = unary-eval 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  $kind\ g\ neg = LogicNegationNode\ y$ 
  assumes  $[g, m, p] \vdash neg \mapsto invval$ 
  assumes  $invval \neq UndefinedVal$ 
  shows  $val\text{-}to\text{-}bool\ val \longleftrightarrow \neg(val\text{-}to\text{-}bool\ invval)$ 
proof -
  obtain yencode where 5:  $g \vdash y \simeq yencode$ 
  using assms(1) encodeeval-def by auto
  then have 6:  $g \vdash neg \simeq UnaryExpr\ UnaryLogicNegation\ yencode$ 
  using rep.intros(7) assms(2) by simp
  then have 7:  $[m, p] \vdash UnaryExpr\ UnaryLogicNegation\ yencode \mapsto invval$ 
  using assms(3) encodeeval-def
  by (metis repDet)
  obtain v1 where v1:  $[g, m, p] \vdash y \mapsto IntVal\ 32\ v1$ 
  using assms(1,2,3,4) using logic-negate-type sorry
  have  $invval = bool\text{-}to\text{-}val\ (\neg(val\text{-}to\text{-}bool\ val))$ 
  using assms(1,2,3) evalDet unary-eval.simps(4)
  sorry
  have  $val\text{-}to\text{-}bool\ invval \longleftrightarrow \neg(val\text{-}to\text{-}bool\ val)$ 
  using  $\langle invval = bool\text{-}to\text{-}val\ (\neg\ val\text{-}to\text{-}bool\ val) \rangle$  by force
  then show ?thesis
  by simp
qed
end

```

```

lemma implies-valid:
  assumes  $x \ \&\ y \hookrightarrow imp$ 
  assumes  $[m, p] \vdash x \mapsto v1$ 
  assumes  $[m, p] \vdash y \mapsto v2$ 
  assumes  $v1 \neq UndefinedVal \wedge v2 \neq UndefinedVal$ 
  shows  $(imp \longrightarrow (val\text{-}to\text{-}bool\ v1 \longrightarrow val\text{-}to\text{-}bool\ v2)) \wedge$ 
     $(\neg imp \longrightarrow (val\text{-}to\text{-}bool\ v1 \longrightarrow \neg(val\text{-}to\text{-}bool\ v2)))$ 
     $(is\ (?TP \longrightarrow ?TC) \wedge (?FP \longrightarrow ?FC))$ 
  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] ⊢ x ↦ xval
    using eq-imp-less(1) eq-imp-less.prem(3)
    by blast
    then obtain yval where yval: [m, p] ⊢ y ↦ yval
    using eq-imp-less.prem(3)
    using eq-imp-less.prem(2) 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) eq-imp-less.prem(1) evalDet)
    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) eq-imp-less.prem(2) evalDet)
    have val-to-bool (intval-equals xval yval) ⟶ ¬(val-to-bool (intval-less-than xval
    yval))
    using assms(4) apply (cases xval; cases yval; auto) sorry

    then show ?case
    using egeval lesseval
    by (metis eq-imp-less.prem(1) eq-imp-less.prem(2) evalDet)
next
    case (eq-imp-less-rev x y)
    obtain xval where xval: [m, p] ⊢ x ↦ xval
    using eq-imp-less-rev.prem(3)
    using eq-imp-less-rev.prem(2) by blast
    obtain yval where yval: [m, p] ⊢ y ↦ yval

```

```

    using eq-imp-less-rev.premis(3)
    using eq-imp-less-rev.premis(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.premis(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.premis(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) sorry

  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:  $[m, p] \vdash x \mapsto xval$ 
    using less-imp-rev-less.premis(3)
    using less-imp-rev-less.premis(2) by blast
  obtain yval where yval:  $[m, p] \vdash y \mapsto yval$ 
    using less-imp-rev-less.premis(3)
    using less-imp-rev-less.premis(2) by blast
  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) evalDet less-imp-rev-less.premis(1))
    have revlesseval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerLessThan } y \ x) \mapsto \text{int-}$ 
  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)  $\longrightarrow \neg(\text{val-to-bool } (\text{intval-less-than}$ 
  yval xval))
    using assms(4) apply (cases xval; cases yval; auto) sorry

  then show ?case
    by (metis evalDet less-imp-rev-less.premis(1) less-imp-rev-less.premis(2) lesseval
  revlesseval)
next
  case (less-imp-not-eq x y)
  obtain xval where xval:  $[m, p] \vdash x \mapsto xval$ 
    using less-imp-not-eq.premis(3)
    using less-imp-not-eq.premis(1) by blast
  obtain yval where yval:  $[m, p] \vdash y \mapsto yval$ 
    using less-imp-not-eq.premis(3)
    using less-imp-not-eq.premis(1) 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) evalDet less-imp-not-eq.prem(2))
  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) evalDet less-imp-not-eq.prem(1))
  have val-to-bool (intval-less-than xval yval)  $\longrightarrow \neg(\text{val-to-bool (intval-equals xval yval)})$ 
  using assms(4) apply (cases xval; cases yval; auto) sorry

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

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  $x \ \& \ y \hookrightarrow \text{imp}$ 
  assumes 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  $\text{val-to-bool } v1 \longrightarrow \text{val-to-bool } v2$ 
  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
     $\llbracket \text{alwaysDistinct } (\text{stamps } x) \ (\text{stamps } y) \rrbracket$ 
       $\Longrightarrow \text{tryFold } (\text{IntegerEqualsNode } x \ y) \ \text{stamps } \text{False} \mid$ 
     $\llbracket \text{neverDistinct } (\text{stamps } x) \ (\text{stamps } y) \rrbracket$ 
       $\Longrightarrow \text{tryFold } (\text{IntegerEqualsNode } x \ y) \ \text{stamps } \text{True} \mid$ 
     $\llbracket \text{is-IntegerStamp } (\text{stamps } x);$ 
       $\text{is-IntegerStamp } (\text{stamps } y);$ 
       $\text{stpi-upper } (\text{stamps } x) < \text{stpi-lower } (\text{stamps } y) \rrbracket$ 
       $\Longrightarrow \text{tryFold } (\text{IntegerLessThanNode } x \ y) \ \text{stamps } \text{True} \mid$ 
     $\llbracket \text{is-IntegerStamp } (\text{stamps } x);$ 
       $\text{is-IntegerStamp } (\text{stamps } y);$ 
       $\text{stpi-lower } (\text{stamps } x) \geq \text{stpi-upper } (\text{stamps } y) \rrbracket$ 
       $\Longrightarrow \text{tryFold } (\text{IntegerLessThanNode } x \ y) \ \text{stamps } \text{False}$ 

```

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  $\text{assms}(1, 2, 3, 4) \ \text{BinaryExprE IntegerEqualsNode bin-eval.simps}(7)$ 
    by  $(\text{smt } (\text{verit}) \ \text{bin-eval.simps}(11) \ \text{encodeeval-def evalDet repDet})$ 
  then show  $?thesis$  using  $\text{intval-equals.simps val-to-bool.simps}$  sorry
qed

lemma  $\text{tryFoldIntegerEqualsAlwaysDistinct}$ :
  assumes  $\text{wf-stamp } g \ \text{stamps}$ 
  assumes  $\text{kind } g \ \text{nid} = (\text{IntegerEqualsNode } x \ y)$ 
  assumes  $[g, m, p] \vdash \text{nid} \mapsto v$ 
  assumes  $\text{alwaysDistinct } (\text{stamps } x) \ (\text{stamps } y)$ 
  shows  $v = \text{IntVal32 } 0$ 
proof –
  have  $\forall \ \text{val}. \neg(\text{valid-value } \text{val} \ (\text{join } (\text{stamps } x) \ (\text{stamps } y)))$ 
    using  $\text{assms}(1,4) \ \text{unfolding alwaysDistinct.simps}$ 
    by  $(\text{smt } (\text{verit}, \text{best}) \ \text{is-stamp-empty.elims}(2) \ \text{valid-int valid-value.simps}(1))$ 
  have  $\neg(\exists \ \text{val} . ([g, m, p] \vdash x \mapsto \text{val}) \wedge ([g, m, p] \vdash y \mapsto \text{val}))$ 
    using  $\text{assms}(1,4) \ \text{unfolding alwaysDistinct.simps wf-stamp.simps encodeeval-def}$  sorry
  then show  $?thesis$  sorry
qed

lemma  $\text{tryFoldIntegerEqualsNeverDistinct}$ :
  assumes  $\text{wf-stamp } g \ \text{stamps}$ 
  assumes  $\text{kind } g \ \text{nid} = (\text{IntegerEqualsNode } x \ y)$ 
  assumes  $[g, m, p] \vdash \text{nid} \mapsto v$ 
  assumes  $\text{neverDistinct } (\text{stamps } x) \ (\text{stamps } y)$ 
  shows  $v = \text{IntVal32 } 1$ 
  using  $\text{assms IntegerEqualsNodeE}$  sorry

lemma  $\text{tryFoldIntegerLessThanTrue}$ :
  assumes  $\text{wf-stamp } g \ \text{stamps}$ 
  assumes  $\text{kind } g \ \text{nid} = (\text{IntegerLessThanNode } x \ y)$ 
  assumes  $[g, m, p] \vdash \text{nid} \mapsto v$ 
  assumes  $\text{stpi-upper } (\text{stamps } x) < \text{stpi-lower } (\text{stamps } y)$ 
  shows  $v = \text{IntVal32 } 1$ 
proof –
  have  $\text{stamp-type: is-IntegerStamp } (\text{stamps } x)$ 
    using  $\text{assms}$ 
    sorry
  obtain  $xval$  where  $xval: [g, m, p] \vdash x \mapsto xval$ 
    using  $\text{assms}(2,3)$  sorry
  obtain  $yval$  where  $yval: [g, m, p] \vdash y \mapsto yval$ 
    using  $\text{assms}(2,3)$  sorry
  have  $\text{is-IntegerStamp } (\text{stamps } x) \wedge \text{is-IntegerStamp } (\text{stamps } y)$ 
    using  $\text{assms}(4)$ 
    sorry
  then have  $\text{val-to-bool } (\text{intval-less-than } xval \ yval)$ 

```



```

    sorry
  then show ?thesis
    sorry
qed

```

```

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

```

```

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

$g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m', h)$

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* ::

$\text{IRExpr set} \Rightarrow (\text{ID} \Rightarrow \text{Stamp}) \Rightarrow \text{IRGraph} \Rightarrow \text{ID} \Rightarrow \text{IRGraph} \Rightarrow \text{bool}$ **where**
impliesTrue:

$\llbracket \text{kind } g \text{ ifcond} = (\text{IfNode } \text{cid } t \text{ } f);$
 $g \vdash \text{cid} \simeq \text{cond};$
 $\exists \text{ ce} \in \text{conds} . (\text{ce} \ \& \ \text{cond} \hookrightarrow \text{True});$
 $g' = \text{constantCondition True ifcond (kind } g \text{ ifcond) } g$
 $\rrbracket \implies \text{ConditionalEliminationStep conds stamps } g \text{ ifcond } g' \mid$

impliesFalse:

$\llbracket \text{kind } g \text{ ifcond} = (\text{IfNode } \text{cid } t \text{ } f);$
 $g \vdash \text{cid} \simeq \text{cond};$
 $\exists \text{ ce} \in \text{conds} . (\text{ce} \ \& \ \text{cond} \hookrightarrow \text{False});$
 $g' = \text{constantCondition False ifcond (kind } g \text{ ifcond) } g$
 $\rrbracket \implies \text{ConditionalEliminationStep conds stamps } g \text{ ifcond } g' \mid$

tryFoldTrue:

```

[[kind g ifcond = (IfNode cid t f);
  cond = kind g cid;
  tryFold (kind g cid) stamps True;
  g' = constantCondition True ifcond (kind g ifcond) g
]] ==> ConditionalEliminationStep conds stamps g ifcond g' |

```

```

tryFoldFalse:
[[kind g ifcond = (IfNode cid t f);
  cond = kind g cid;
  tryFold (kind g cid) stamps False;
  g' = constantCondition False ifcond (kind g ifcond) g
]] ==> 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*

11.2 Control-flow Graph Traversal

```

type-synonym Seen = ID set
type-synonym Condition = IRExpr
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 ( $\lambda \text{nid}'$ . 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 ⇒ int ⇒ Stamp where
  clip-upper (IntegerStamp b l h) c = (IntegerStamp b l c) |
  clip-upper s c = s
fun clip-lower :: Stamp ⇒ int ⇒ Stamp where
  clip-lower (IntegerStamp b l h) c = (IntegerStamp b c h) |
  clip-lower s c = s
```

```
fun registerNewCondition :: IRGraph ⇒ IRNode ⇒ (ID ⇒ Stamp) ⇒ (ID ⇒ Stamp) where
```

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

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

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

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

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

inductive *Step*

```
:: IRGraph ⇒ (ID × Seen × Conditions × StampFlow) ⇒ (ID × Seen × Conditions × StampFlow) option ⇒ bool
```

for *g* **where**

- Hit a `BeginNode` with an `IfNode` predecessor which represents the start of a basic block for the `IfNode`. 1. `nid'` will be the successor of the begin node. 2. Find the first and only predecessor. 3. Extract condition from the preceding `IfNode`. 4. Negate condition if the begin node is second branch (we've taken the else branch of the condition) 5. Add the condition or the negated condition to stack 6. Perform any stamp updates based on the condition using the `registerNewCondition` function

and place them on the top of the stack of stamp information

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

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

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

$i = \text{find-index } \text{nid} \text{ (successors-of } (\text{kind } g \text{ ifcond}));$
 $c = (\text{if } i = 0 \text{ then } \text{kind } g \text{ cond else } \text{LogicNegationNode } \text{cond});$
 $\text{rep } g \text{ cond } ce;$
 $ce' = (\text{if } i = 0 \text{ then } ce \text{ else } \text{UnaryExpr } \text{UnaryLogicNegation } ce);$
 $\text{conds}' = ce' \# \text{conds};$

$\text{flow}' = \text{registerNewCondition } g \text{ } c \text{ (hdOr flow (stamp } g));$
 $\implies \text{Step } g \text{ (nid, seen, conds, flow) (Some (nid', seen', conds', flow' \# flow))} \mid$

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

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

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

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

$\text{conds}' = \text{tl conds};$
 $\text{flow}' = \text{tl flow} \rrbracket$
 $\implies \text{Step } g \text{ (nid, seen, conds, flow) (Some (nid', seen', conds', flow'))} \mid$

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

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

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

$\text{Some } \text{nid}' = \text{nextEdge seen' nid } g \rrbracket$
 $\implies \text{Step } g \text{ (nid, seen, conds, flow) (Some (nid', seen', conds, flow))} \mid$

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

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

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

$\text{None} = \text{nextEdge seen' nid } g \rrbracket$

$\implies \text{Step } g \text{ (nid, seen, conds, flow) None} \mid$

— We've already seen this node, give back None
 $\llbracket \text{nid} \in \text{seen} \rrbracket \implies \text{Step } g \text{ (nid, seen, conds, flow) None}$

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

inductive ConditionalEliminationPhase

$\text{:: IRGraph} \Rightarrow (\text{ID} \times \text{Seen} \times \text{Conditions} \times \text{StampFlow}) \Rightarrow \text{IRGraph} \Rightarrow \text{bool}$

where

— Can do a step and optimise for the current node
 $\llbracket \text{Step } g \text{ (nid, seen, conds, flow) (Some (nid', seen', conds', flow'))} \rrbracket$;
 $\text{ConditionalEliminationStep (set conds) (hdOr flow (stamp g)) } g \text{ nid } g';$

$\text{ConditionalEliminationPhase } g' \text{ (nid', seen', conds', flow') } g' \rrbracket$
 $\implies \text{ConditionalEliminationPhase } g \text{ (nid, seen, conds, flow) } g'' \mid$

— Can do a step, matches whether optimised or not causing non-determinism We need to find a way to negate ConditionalEliminationStep

$\llbracket \text{Step } g \text{ (nid, seen, conds, flow) (Some (nid', seen', conds', flow'))} \rrbracket$;
 $\text{ConditionalEliminationPhase } g \text{ (nid', seen', conds', flow') } g' \rrbracket$
 $\implies \text{ConditionalEliminationPhase } g \text{ (nid, seen, conds, flow) } g' \mid$

— Can't do a step but there is a predecessor we can backtrace to

$\llbracket \text{Step } g \text{ (nid, seen, conds, flow) None} \rrbracket$;
 $\text{Some nid}' = \text{pred } g \text{ nid};$
 $\text{seen}' = \{\text{nid}\} \cup \text{seen};$
 $\text{ConditionalEliminationPhase } g \text{ (nid', seen', conds, flow) } g' \rrbracket$
 $\implies \text{ConditionalEliminationPhase } g \text{ (nid, seen, conds, flow) } g' \mid$

— Can't do a step and have no predecessors so terminate

$\llbracket \text{Step } g \text{ (nid, seen, conds, flow) None} \rrbracket$;
 $\text{None} = \text{pred } g \text{ nid} \rrbracket$
 $\implies \text{ConditionalEliminationPhase } g \text{ (nid, seen, conds, flow) } g$

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

end