

Veriopt

February 6, 2023

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	11
1.6.1	Support Lemmas for Upper/Lower Bounds	12
1.6.2	Support lemmas for take bit and signed take bit.	15
2	Stamp Typing	16
3	Graph Representation	20
3.1	IR Graph Nodes	20
3.2	IR Graph Node Hierarchy	28
3.3	IR Graph Type	35
3.3.1	Example Graphs	39
3.4	Control-flow Graph Traversal	40
3.5	Structural Graph Comparison	42
4	java.lang.Long	43
4.1	Long.numberOfLeadingZeros	44
4.2	Long.numberOfTrailingZeros	45
4.3	Long.bitCount	45
4.4	Long.zeroCount	45
5	Data-flow Semantics	47
5.1	Data-flow Tree Representation	48
5.2	Functions for re-calculating stamps	49
5.3	Data-flow Tree Evaluation	50
5.4	Data-flow Tree Refinement	53
5.5	Stamp Masks	53
5.6	Data-flow Tree Theorems	55
5.6.1	Deterministic Data-flow Evaluation	55
5.6.2	Typing Properties for Integer Evaluation Functions	55
5.6.3	Evaluation Results are Valid	57
5.6.4	Example Data-flow Optimisations	58
5.6.5	Monotonicity of Expression Refinement	58
5.7	Unfolding rules for evaltree quadruples down to bin-eval level	59
5.8	Lemmas about <i>new_int</i> and integer eval results.	60

6	Tree to Graph	61
6.1	Subgraph to Data-flow Tree	61
6.2	Data-flow Tree to Subgraph	65
6.3	Lift Data-flow Tree Semantics	70
6.4	Graph Refinement	70
6.5	Maximal Sharing	70
6.6	Formedness Properties	70
6.7	Dynamic Frames	72
6.8	Tree to Graph Theorems	76
6.8.1	Extraction and Evaluation of Expression Trees is De- terministic.	76
6.8.2	Monotonicity of Graph Refinement	81
6.8.3	Lift Data-flow Tree Refinement to Graph Refinement .	83
6.8.4	Term Graph Reconstruction	84
6.8.5	Data-flow Tree to Subgraph Preserves Maximal Sharing	87
7	Control-flow Semantics	91
7.1	Object Heap	91
7.2	Intraprocedural Semantics	91
7.3	Interprocedural Semantics	94
7.4	Big-step Execution	95
7.4.1	Heap Testing	96
7.5	Control-flow Semantics Theorems	97
7.5.1	Control-flow Step is Deterministic	97
8	Proof Infrastructure	98
8.1	Bisimulation	98
8.2	Graph Rewriting	99
8.3	Stuttering	101
8.4	Evaluation Stamp Theorems	102
8.4.1	Support Lemmas for Integer Stamps and Associated IntVal values	102
8.4.2	Validity of all Unary Operators	104
8.4.3	Support Lemmas for Binary Operators	105
8.4.4	Validity of Stamp Meet and Join Operators	106
8.4.5	Validity of conditional expressions	107
8.4.6	Validity of Whole Expression Tree Evaluation	107
9	Optization DSL	108
9.1	Markup	108
9.1.1	Expression Markup	108
9.1.2	Value Markup	109
9.1.3	Word Markup	110
9.2	Optimization Phases	111

9.3	Canonicalization DSL	111
9.3.1	Semantic Preservation Obligation	114
9.3.2	Termination Obligation	114
9.3.3	Standard Termination Measure	115
9.3.4	Automated Tactics	115
10	Canonicalization Optimizations	116
10.1	AbsNode Phase	117
10.2	AddNode Phase	120
10.3	AndNode Phase	123
10.4	BinaryNode Phase	125
10.5	ConditionalNode Phase	126
10.6	MulNode Phase	128
10.7	Experimental AndNode Phase	133
10.8	NotNode Phase	139
10.9	OrNode Phase	140
10.10	ShiftNode Phase	142
10.11	SignedDivNode Phase	143
10.12	SignedRemNode Phase	143
10.13	SubNode Phase	144
10.14	XorNode Phase	147
11	Conditional Elimination Phase	149
11.1	Individual Elimination Rules	149
11.2	Control-flow Graph Traversal	155

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 }
  <proof>
```

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

```
lift-definition IntWidthBits :: IntWidth  $\Rightarrow$  nat
  is  $\lambda w. w$  <proof>
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^{bits}) *div* 2) * -1, ((2^{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*)
 <proof>

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

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  $\Rightarrow$  Value where
```



```

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

```

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

```

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

```

1.2 Bitwise Operators

```

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

```

```

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

```

```

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

```

```

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

```

1.3 Comparison Operators

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

1.4 Narrowing and Widening Operators

Note: we allow these operators to have $\text{inBits} = \text{outBits}$, 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
<proof>
```

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  $\wedge$ 
  intval-bits val = inBits
<proof>
```

lemma *intval-zero-extend-ok*:

```
assumes intval-zero-extend inBits outBits val  $\neq$  UndefVal
```

shows $0 < inBits \wedge$
 $inBits \leq outBits \wedge outBits \leq 64 \wedge$
 $is-IntVal\ val \wedge$
 $intval-bits\ val = inBits$
 $\langle proof \rangle$

1.5 Bit-Shifting Operators

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

lemma *shiffl-power[simp]*: $(x :: ('a :: len)\ word) * (2^j) = x << j$
 $\langle proof \rangle$

lemma $(x :: ('a :: len)\ word) * ((2^j) + 1) = x << j + x$
 $\langle proof \rangle$

lemma $(x :: ('a :: len)\ word) * ((2^j) - 1) = x << j - x$
 $\langle proof \rangle$

lemma $(x :: ('a :: len)\ word) * ((2^j) + (2^k)) = x << j + x << k$
 $\langle proof \rangle$

lemma $(x :: ('a :: len)\ word) * ((2^j) - (2^k)) = x << j - x << k$
 $\langle proof \rangle$

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

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

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

value $(128 :: 8\ word) >> 2$

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

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

fun *intval-left-shift* :: $Value \Rightarrow Value \Rightarrow Value$ **where**
 $intval-left-shift\ (IntVal\ b1\ v1)\ (IntVal\ b2\ v2) = new-int\ b1\ (v1 << shift-amount\ b1\ v2) \mid$
 $intval-left-shift\ -\ - = UndefVal$

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  $\Rightarrow$  Value  $\Rightarrow$  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  $\Rightarrow$  Value  $\Rightarrow$  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 <proof>
corollary intval-narrow 32 8 (IntVal 32 (-2)) = IntVal 8 254 <proof>
corollary intval-narrow 32 1 (IntVal 32 (-2)) = IntVal 1 0 <proof>
corollary intval-narrow 32 1 (IntVal 32 (-3)) = IntVal 1 1 <proof>
```

```
corollary intval-narrow 32 8 (IntVal 64 (-2)) = UndefVal <proof>
corollary intval-narrow 64 8 (IntVal 32 (-2)) = UndefVal <proof>
corollary intval-narrow 64 8 (IntVal 64 254) = IntVal 8 254 <proof>
corollary intval-narrow 64 8 (IntVal 64 (256+127)) = IntVal 8 127 <proof>
corollary intval-narrow 64 64 (IntVal 64 (-2)) = IntVal 64 (-2) <proof>
end
```

experiment begin

```
corollary intval-sign-extend 8 32 (IntVal 8 (256 + 128)) = IntVal 32 (232 -
128) <proof>
corollary intval-sign-extend 8 32 (IntVal 8 (-2)) = IntVal 32 (232 - 2) <proof>
corollary intval-sign-extend 1 32 (IntVal 1 (-2)) = IntVal 32 0 <proof>
corollary intval-sign-extend 1 32 (IntVal 1 (-3)) = IntVal 32 (mask 32) <proof>
```

```
corollary intval-sign-extend 8 32 (IntVal 64 254) = UndefVal <proof>
corollary intval-sign-extend 8 64 (IntVal 32 254) = UndefVal <proof>
corollary intval-sign-extend 8 64 (IntVal 8 254) = IntVal 64 (-2) <proof>
corollary intval-sign-extend 32 64 (IntVal 32 (232 - 2)) = IntVal 64 (-2) <proof>
corollary intval-sign-extend 64 64 (IntVal 64 (-2)) = IntVal 64 (-2) <proof>
end
```

experiment begin

```

corollary intval-zero-extend 8 32 (IntVal 8 (256 + 128)) = IntVal 32 128 <proof>
corollary intval-zero-extend 8 32 (IntVal 8 (-2)) = IntVal 32 254 <proof>
corollary intval-zero-extend 1 32 (IntVal 1 (-1)) = IntVal 32 1 <proof>
corollary intval-zero-extend 1 32 (IntVal 1 (-2)) = IntVal 32 0 <proof>

```

```

corollary intval-zero-extend 8 32 (IntVal 64 (-2)) = UndefVal <proof>
corollary intval-zero-extend 8 64 (IntVal 64 (-2)) = UndefVal <proof>
corollary intval-zero-extend 8 64 (IntVal 8 254) = IntVal 64 254 <proof>
corollary intval-zero-extend 32 64 (IntVal 32 (232 - 2)) = IntVal 64 (232 - 2) <proof>
corollary intval-zero-extend 64 64 (IntVal 64 (-2)) = IntVal 64 (-2) <proof>
end

```

experiment begin

```

corollary intval-right-shift (IntVal 8 128) (IntVal 8 0) = IntVal 8 128 <proof>
corollary intval-right-shift (IntVal 8 128) (IntVal 8 1) = IntVal 8 192 <proof>
corollary intval-right-shift (IntVal 8 128) (IntVal 8 2) = IntVal 8 224 <proof>
corollary intval-right-shift (IntVal 8 128) (IntVal 8 8) = IntVal 8 255 <proof>
corollary intval-right-shift (IntVal 8 128) (IntVal 8 31) = IntVal 8 255 <proof>
end

```

```

lemma intval-add-sym:
  shows intval-add a b = intval-add b a
  <proof>

```

```

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

```

```

lemma intval-add (IntVal 32 (231-1)) (IntVal 32 (231-1)) = IntVal 32 (232 - 2)
  <proof>
lemma intval-add (IntVal 64 (231-1)) (IntVal 64 (231-1)) = IntVal 64 4294967294
  <proof>

```

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** $v :: 32 \text{ word}$
 $\langle \text{proof} \rangle$

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

lemma *lower-bounds-equiv*:
assumes $0 < N$
shows $-(((2::\text{int}) \wedge (N-1))) = (2::\text{int}) \wedge N \text{ div } 2 * - 1$
 $\langle \text{proof} \rangle$

lemma *upper-bounds-equiv*:
assumes $0 < N$
shows $(2::\text{int}) \wedge (N-1) = (2::\text{int}) \wedge N \text{ div } 2$
 $\langle \text{proof} \rangle$

Some min/max bounds for 64-bit words

lemma *bit-bounds-min64*: $((\text{fst } (\text{bit-bounds } 64))) \leq (\text{sint } (v::\text{int64}))$
 $\langle \text{proof} \rangle$

lemma *bit-bounds-max64*: $((\text{snd } (\text{bit-bounds } 64))) \geq (\text{sint } (v::\text{int64}))$
 $\langle \text{proof} \rangle$

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 $\langle @\{ \text{thm } \text{signed-take-bit-decr-length-iff} \} \rangle$
declare $[[\text{show-types}=\text{true}]]$
ML-val $\langle @\{ \text{thm } \text{signed-take-bit-int-less-exp} \} \rangle$

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

lemma *signed-take-bit-int-greater-eq-minus-exp-word*:
fixes $\text{ival} :: 'a :: \text{len word}$
assumes $n < \text{LENGTH}('a)$

shows $-(2 \wedge n) \leq \text{sint}(\text{signed-take-bit } n \text{ ival})$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-range*:
fixes $\text{ival} :: 'a :: \text{len word}$
assumes $n < \text{LENGTH}('a)$
assumes $\text{val} = \text{sint}(\text{signed-take-bit } n \text{ ival})$
shows $-(2 \wedge n) \leq \text{val} \wedge \text{val} < 2 \wedge n$
 $\langle \text{proof} \rangle$

A *bit_bounds* version of the above lemma.

lemma *signed-take-bit-bounds*:
fixes $\text{ival} :: 'a :: \text{len word}$
assumes $n \leq \text{LENGTH}('a)$
assumes $0 < n$
assumes $\text{val} = \text{sint}(\text{signed-take-bit } (n - 1) \text{ ival})$
shows $\text{fst } (\text{bit-bounds } n) \leq \text{val} \wedge \text{val} \leq \text{snd } (\text{bit-bounds } n)$
 $\langle \text{proof} \rangle$

lemma *signed-take-bit-bounds64*:
fixes $\text{ival} :: \text{int64}$
assumes $n \leq 64$
assumes $0 < n$
assumes $\text{val} = \text{sint}(\text{signed-take-bit } (n - 1) \text{ ival})$
shows $\text{fst } (\text{bit-bounds } n) \leq \text{val} \wedge \text{val} \leq \text{snd } (\text{bit-bounds } n)$
 $\langle \text{proof} \rangle$

lemma *int-signed-value-bounds*:
assumes $b1 \leq 64$
assumes $0 < b1$
shows $\text{fst } (\text{bit-bounds } b1) \leq \text{int-signed-value } b1 \text{ v2} \wedge$
 $\text{int-signed-value } b1 \text{ v2} \leq \text{snd } (\text{bit-bounds } b1)$
 $\langle \text{proof} \rangle$

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

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

lemma *take-bit-smaller-range*:
fixes $\text{ival} :: 'a :: \text{len word}$
assumes $n < \text{LENGTH}('a)$
assumes $\text{val} = \text{sint}(\text{take-bit } n \text{ ival})$
shows $0 \leq \text{val} \wedge \text{val} < (2::\text{int}) \wedge n$
 $\langle \text{proof} \rangle$

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

fixes *ival* :: 'a :: len word
assumes $n = \text{LENGTH}('a)$
shows $\text{ival} = \text{take-bit } n \text{ ival}$
 $\langle \text{proof} \rangle$

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

lemma *take-bit-redundant[simp]*:

fixes *ival* :: 'a :: len word
assumes $0 < n$
assumes $n < \text{LENGTH}('a)$
shows $\text{signed-take-bit } (n - 1) (\text{take-bit } n \text{ ival}) = \text{signed-take-bit } (n - 1) \text{ ival}$
 $\langle \text{proof} \rangle$

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 $-(2^n \text{ div } 2) \leq \text{sint ival2} \wedge \text{sint ival2} < 2^n \text{ div } 2$
 $\langle \text{proof} \rangle$

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 $\text{fst } (\text{bit-bounds } n) \leq \text{sint ival2} \wedge \text{sint ival2} \leq \text{snd } (\text{bit-bounds } n)$
 $\langle \text{proof} \rangle$

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 $\text{sint } ((\text{scast } v) :: 'b :: \text{len word}) < M$
 $\langle \text{proof} \rangle$

lemma *scast-min-bound*:

assumes $M \leq \text{sint } (v :: 'a :: \text{len word})$
assumes $\text{LENGTH}('a) < \text{LENGTH}('b)$
shows $M \leq \text{sint } ((\text{scast } v) :: 'b :: \text{len word})$
 $\langle \text{proof} \rangle$

lemma *scast-bigger-max-bound*:

assumes $(\text{result} :: 'b :: \text{len word}) = \text{scast } (v :: 'a :: \text{len word})$
shows $\text{sint result} < 2^n \text{ div } 2$
 $\langle \text{proof} \rangle$

lemma *scast-bigger-min-bound*:

assumes $(result :: 'b :: len\ word) = scast\ (v :: 'a :: len\ word)$
shows $-(2 \wedge LENGTH('a) \div 2) \leq sint\ result$
 $\langle proof \rangle$

lemma *scast-bigger-bit-bounds*:

assumes $(result :: 'b :: len\ word) = scast\ (v :: 'a :: len\ word)$
shows $fst\ (bit-bounds\ (LENGTH('a))) \leq sint\ result \wedge sint\ result \leq snd\ (bit-bounds\ (LENGTH('a)))$
 $\langle proof \rangle$

Results about *new_int*.

lemma *new-int-take-bits*:

assumes $IntVal\ b\ val = new-int\ b\ ival$
shows $take-bit\ b\ val = val$
 $\langle proof \rangle$

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 $take-bit\ b\ (take-bit\ b\ x + y) = take-bit\ b\ (x + y)$
 $\langle proof \rangle$

lemma *take-bit-dist-addR[simp]*:

fixes $x :: 'a :: len\ word$
shows $take-bit\ b\ (x + take-bit\ b\ y) = take-bit\ b\ (x + y)$
 $\langle proof \rangle$

lemma *take-bit-dist-subL[simp]*:

fixes $x :: 'a :: len\ word$
shows $take-bit\ b\ (take-bit\ b\ x - y) = take-bit\ b\ (x - y)$
 $\langle proof \rangle$

lemma *take-bit-dist-subR[simp]*:

fixes $x :: 'a :: len\ word$
shows $take-bit\ b\ (x - take-bit\ b\ y) = take-bit\ b\ (x - y)$
 $\langle proof \rangle$

lemma *take-bit-dist-neg[simp]*:

fixes $ix :: 'a :: len\ word$
shows $take-bit\ b\ (-\ take-bit\ b\ (ix)) = take-bit\ b\ (-\ ix)$
 $\langle proof \rangle$

```

lemma signed-take-take-bit[simp]:
  fixes  $x :: 'a :: \text{len word}$ 
  assumes  $0 < b$ 
  shows  $\text{signed-take-bit } (b - 1) (\text{take-bit } b \ x) = \text{signed-take-bit } (b - 1) \ x$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma mod-larger-ignore:
  fixes  $a :: \text{int}$ 
  fixes  $m \ n :: \text{nat}$ 
  assumes  $n < m$ 
  shows  $(a \bmod 2^m) \bmod 2^n = a \bmod 2^n$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma mod-dist-over-add:
  fixes  $a \ b \ c :: \text{int64}$ 
  fixes  $n :: \text{nat}$ 
  assumes  $1: 0 < n$ 
  assumes  $2: n < 64$ 
  shows  $(a \bmod 2^n + b) \bmod 2^n = (a + b) \bmod 2^n$ 
   $\langle \text{proof} \rangle$ 

```

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) \langle proof \rangle

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
    "" False False False) |
  unrestricted-stamp - = IllegalStamp

```

```

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

```

— A stamp which provides type information but has an empty range of values

```

fun empty-stamp :: Stamp ⇒ Stamp where
  empty-stamp VoidStamp = VoidStamp |
  empty-stamp (IntegerStamp bits lower upper) = (IntegerStamp bits (snd (bit-bounds
    bits)) (fst (bit-bounds bits))) |

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

```

— Calculate the meet stamp of two stamps

```

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

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

```

— Calculate the join stamp of two stamps

```

fun join :: Stamp ⇒ Stamp ⇒ Stamp where
  join VoidStamp VoidStamp = VoidStamp |
  join (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (

```

```

    if b1 ≠ b2 then IllegalStamp else
    (IntegerStamp b1 (max l1 l2) (min u1 u2))
  ) |

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

```

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

```

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

```

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

```

fun alwaysDistinct :: Stamp ⇒ Stamp ⇒ bool where
  alwaysDistinct stamp1 stamp2 = is-stamp-empty (join stamp1 stamp2)

```

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

```

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

```

```

fun constantAsStamp :: Value ⇒ Stamp where

```

```

  constantAsStamp (IntVal b v) = (IntegerStamp b (int-signed-value b v) (int-signed-value
b v)) |

```

```

  constantAsStamp - = IllegalStamp

```

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

```

fun valid-value :: Value ⇒ Stamp ⇒ bool where

```

```

valid-value (IntVal b1 val) (IntegerStamp b l h) =
  (if b1 = b then
    valid-stamp (IntegerStamp b l h) ∧
    take-bit b val = val ∧
    l ≤ int-signed-value b val ∧ int-signed-value b val ≤ h
  else False) |

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

```

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

lemma *unfold-wf-value[simp]*:
wf-value v ⇒ valid-value v (constantAsStamp v)
 <proof>

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

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

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

definition *default-stamp* :: Stamp **where**
default-stamp = (unrestricted-stamp (IntegerStamp 32 0 0))

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

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: IN-
PUT-STATE option) (ir-values-opt: INPUT list option) (ir-virtualObjectMappings-opt:
INPUT-STATE list option)
  | IfNode (ir-condition: INPUT-COND) (ir-trueSuccessor: SUCC) (ir-falseSuccessor:
SUCC)
  | IntegerBelowNode (ir-x: INPUT) (ir-y: INPUT)
  | IntegerEqualsNode (ir-x: INPUT) (ir-y: INPUT)
```

- | *IntegerLessThanNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
- | *InvokeNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
- | *InvokeWithExceptionNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*) (*ir-exceptionEdge*: *SUCC*)
- | *IsNullNode* (*ir-value*: *INPUT*)
- | *KillingBeginNode* (*ir-next*: *SUCC*)
- | *LeftShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
- | *LoadFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)
- | *LogicNegationNode* (*ir-value*: *INPUT-COND*)
- | *LoopBeginNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-overflowGuard-opt*: *INPUT-GUARD option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
- | *LoopEndNode* (*ir-loopBegin*: *INPUT-ASSOC*)
- | *LoopExitNode* (*ir-loopBegin*: *INPUT-ASSOC*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
- | *MergeNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
- | *MethodCallTargetNode* (*ir-targetMethod*: *string*) (*ir-arguments*: *INPUT list*)
- | *MulNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
- | *NarrowNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
- | *NegateNode* (*ir-value*: *INPUT*)
- | *NewArrayNode* (*ir-length*: *INPUT*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
- | *NewInstanceNode* (*ir-nid*: *ID*) (*ir-instanceClass*: *string*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
- | *NotNode* (*ir-value*: *INPUT*)
- | *OrNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
- | *ParameterNode* (*ir-index*: *nat*)
- | *PiNode* (*ir-object*: *INPUT*) (*ir-guard-opt*: *INPUT-GUARD option*)
- | *ReturnNode* (*ir-result-opt*: *INPUT option*) (*ir-memoryMap-opt*: *INPUT-EXT option*)
- | *RightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
- | *ShortCircuitOrNode* (*ir-x*: *INPUT-COND*) (*ir-y*: *INPUT-COND*)
- | *SignExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
- | *SignedDivNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *INPUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
- | *SignedRemNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *INPUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
- | *StartNode* (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
- | *StoreFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-value*: *INPUT*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)
- | *SubNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
- | *UnsignedRightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
- | *UnwindNode* (*ir-exception*: *INPUT*)


```

| ValuePhiNode (ir-nid: ID) (ir-values: INPUT list) (ir-merge: INPUT-ASSOC)
| ValueProxyNode (ir-value: INPUT) (ir-loopExit: INPUT-ASSOC)
| XorNode (ir-x: INPUT) (ir-y: INPUT)
| ZeroExtendNode (ir-inputBits: nat) (ir-resultBits: nat) (ir-value: INPUT)
| NoNode

```

```

| RefNode (ir-ref:ID)

```

```

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

```

```

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

```

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

```

fun inputs-of :: IRNode  $\Rightarrow$  ID list where
  inputs-of-AbsNode:
  inputs-of (AbsNode value) = [value] |
  inputs-of-AddNode:
  inputs-of (AddNode x y) = [x, y] |
  inputs-of-AndNode:
  inputs-of (AndNode x y) = [x, y] |
  inputs-of-BEGINNode:
  inputs-of (BeginNode next) = [] |
  inputs-of-BytecodeExceptionNode:
  inputs-of (BytecodeExceptionNode arguments stateAfter next) = arguments @
  (opt-to-list stateAfter) |
  inputs-of-ConditionalNode:
  inputs-of (ConditionalNode condition trueValue falseValue) = [condition, true-
  Value, falseValue] |
  inputs-of-ConstantNode:
  inputs-of (ConstantNode const) = [] |
  inputs-of-DynamicNewArrayNode:
  inputs-of (DynamicNewArrayNode elementType length0 voidClass stateBefore
  next) = [elementType, length0] @ (opt-to-list voidClass) @ (opt-to-list stateBefore)
  |
  inputs-of-EndNode:
  inputs-of (EndNode) = [] |
  inputs-of-ExceptionObjectNode:
  inputs-of (ExceptionObjectNode stateAfter next) = (opt-to-list stateAfter) |
  inputs-of-FrameState:

```

inputs-of (*FrameState* *monitorIds* *outerFrameState* *values* *virtualObjectMappings*)
 = *monitorIds* @ (*opt-to-list* *outerFrameState*) @ (*opt-list-to-list* *values*) @ (*opt-list-to-list* *virtualObjectMappings*) |
inputs-of-IfNode:
inputs-of (*IfNode* *condition* *trueSuccessor* *falseSuccessor*) = [*condition*] |
inputs-of-IntegerBelowNode:
inputs-of (*IntegerBelowNode* *x* *y*) = [*x*, *y*] |
inputs-of-IntegerEqualsNode:
inputs-of (*IntegerEqualsNode* *x* *y*) = [*x*, *y*] |
inputs-of-IntegerLessThanNode:
inputs-of (*IntegerLessThanNode* *x* *y*) = [*x*, *y*] |
inputs-of-InvokeNode:
inputs-of (*InvokeNode* *nid0* *callTarget* *classInit* *stateDuring* *stateAfter* *next*)
 = *callTarget* # (*opt-to-list* *classInit*) @ (*opt-to-list* *stateDuring*) @ (*opt-to-list* *stateAfter*) |
inputs-of-InvokeWithExceptionNode:
inputs-of (*InvokeWithExceptionNode* *nid0* *callTarget* *classInit* *stateDuring* *stateAfter* *next* *exceptionEdge*) = *callTarget* # (*opt-to-list* *classInit*) @ (*opt-to-list* *stateDuring*) @ (*opt-to-list* *stateAfter*) |
inputs-of-IsNullNode:
inputs-of (*IsNullNode* *value*) = [*value*] |
inputs-of-KillingBeginNode:
inputs-of (*KillingBeginNode* *next*) = [] |
inputs-of-LeftShiftNode:
inputs-of (*LeftShiftNode* *x* *y*) = [*x*, *y*] |
inputs-of-LoadFieldNode:
inputs-of (*LoadFieldNode* *nid0* *field* *object* *next*) = (*opt-to-list* *object*) |
inputs-of-LogicNegationNode:
inputs-of (*LogicNegationNode* *value*) = [*value*] |
inputs-of-LoopBeginNode:
inputs-of (*LoopBeginNode* *ends* *overflowGuard* *stateAfter* *next*) = *ends* @ (*opt-to-list* *overflowGuard*) @ (*opt-to-list* *stateAfter*) |
inputs-of-LoopEndNode:
inputs-of (*LoopEndNode* *loopBegin*) = [*loopBegin*] |
inputs-of-LoopExitNode:
inputs-of (*LoopExitNode* *loopBegin* *stateAfter* *next*) = *loopBegin* # (*opt-to-list* *stateAfter*) |
inputs-of-MergeNode:
inputs-of (*MergeNode* *ends* *stateAfter* *next*) = *ends* @ (*opt-to-list* *stateAfter*) |
inputs-of-MethodCallTargetNode:
inputs-of (*MethodCallTargetNode* *targetMethod* *arguments*) = *arguments* |
inputs-of-MulNode:
inputs-of (*MulNode* *x* *y*) = [*x*, *y*] |
inputs-of-NarrowNode:
inputs-of (*NarrowNode* *inputBits* *resultBits* *value*) = [*value*] |
inputs-of-NegateNode:
inputs-of (*NegateNode* *value*) = [*value*] |
inputs-of-NewArrayNode:
inputs-of (*NewArrayNode* *length0* *stateBefore* *next*) = *length0* # (*opt-to-list* *state-*

Before) |
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
  <proof>

```

```

lemma successors-of (FrameState x (Some y) (Some z) None) = []
  <proof>

```

```

lemma inputs-of (IfNode c t f) = [c]
  <proof>

```

```

lemma successors-of (IfNode c t f) = [t, f]
  <proof>

```

```

lemma inputs-of (EndNode) = [] ∧ successors-of (EndNode) = []
  <proof>

```

```

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

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

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

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

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

```


$n) \vee (is-LoopExitNode\ n) \vee (is-StartNode\ n))$

fun *is-AbstractBeginNode* :: *IRNode* \Rightarrow *bool* **where**

is-AbstractBeginNode *n* = ((*is-BEGINNode* *n*) \vee (*is-BEGINStateSplitNode* *n*) \vee (*is-KillingBeginNode* *n*))

fun *is-FixedWithNextNode* :: *IRNode* \Rightarrow *bool* **where**

is-FixedWithNextNode *n* = ((*is-AbstractBeginNode* *n*) \vee (*is-AbstractStateSplit* *n*) \vee (*is-AccessFieldNode* *n*) \vee (*is-DeoptimizingFixedWithNextNode* *n*))

fun *is-WithExceptionNode* :: *IRNode* \Rightarrow *bool* **where**

is-WithExceptionNode *n* = ((*is-InvokeWithExceptionNode* *n*))

fun *is-ControlSplitNode* :: *IRNode* \Rightarrow *bool* **where**

is-ControlSplitNode *n* = ((*is-IfNode* *n*) \vee (*is-WithExceptionNode* *n*))

fun *is-ControlSinkNode* :: *IRNode* \Rightarrow *bool* **where**

is-ControlSinkNode *n* = ((*is-ReturnNode* *n*) \vee (*is-UnwindNode* *n*))

fun *is-AbstractEndNode* :: *IRNode* \Rightarrow *bool* **where**

is-AbstractEndNode *n* = ((*is-EndNode* *n*) \vee (*is-LoopEndNode* *n*))

fun *is-FixedNode* :: *IRNode* \Rightarrow *bool* **where**

is-FixedNode *n* = ((*is-AbstractEndNode* *n*) \vee (*is-ControlSinkNode* *n*) \vee (*is-ControlSplitNode* *n*) \vee (*is-FixedWithNextNode* *n*))

fun *is-CallTargetNode* :: *IRNode* \Rightarrow *bool* **where**

is-CallTargetNode *n* = ((*is-MethodCallTargetNode* *n*))

fun *is-ValueNode* :: *IRNode* \Rightarrow *bool* **where**

is-ValueNode *n* = ((*is-CallTargetNode* *n*) \vee (*is-FixedNode* *n*) \vee (*is-FloatingNode* *n*))

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

is-Node *n* = ((*is-ValueNode* *n*) \vee (*is-VirtualState* *n*))

fun *is-MemoryKill* :: *IRNode* \Rightarrow *bool* **where**

is-MemoryKill *n* = ((*is-AbstractMemoryCheckpoint* *n*))

fun *is-NarrowableArithmeticNode* :: *IRNode* \Rightarrow *bool* **where**

is-NarrowableArithmeticNode *n* = ((*is-AbsNode* *n*) \vee (*is-AddNode* *n*) \vee (*is-AndNode* *n*) \vee (*is-MulNode* *n*) \vee (*is-NegateNode* *n*) \vee (*is-NotNode* *n*) \vee (*is-OrNode* *n*) \vee (*is-ShiftNode* *n*) \vee (*is-SubNode* *n*) \vee (*is-XorNode* *n*))

fun *is-AnchoringNode* :: *IRNode* \Rightarrow *bool* **where**

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

fun *is-DeoptBefore* :: *IRNode* \Rightarrow *bool* **where**

is-DeoptBefore *n* = ((*is-DeoptimizingFixedWithNextNode* *n*))

```

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

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

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

fun is-UncheckedInterfaceProvider :: IRNode  $\Rightarrow$  bool where
  is-UncheckedInterfaceProvider n = ((is-InvokeNode n)  $\vee$  (is-InvokeWithExceptionNode
n)  $\vee$  (is-LoadFieldNode n)  $\vee$  (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*) \wedge (*is-AndNode* *n2*)) \vee
 ((*is-BeginNode* *n1*) \wedge (*is-BeginNode* *n2*)) \vee
 ((*is-BytecodeExceptionNode* *n1*) \wedge (*is-BytecodeExceptionNode* *n2*)) \vee
 ((*is-ConditionalNode* *n1*) \wedge (*is-ConditionalNode* *n2*)) \vee
 ((*is-ConstantNode* *n1*) \wedge (*is-ConstantNode* *n2*)) \vee
 ((*is-DynamicNewArrayNode* *n1*) \wedge (*is-DynamicNewArrayNode* *n2*)) \vee
 ((*is-EndNode* *n1*) \wedge (*is-EndNode* *n2*)) \vee
 ((*is-ExceptionObjectNode* *n1*) \wedge (*is-ExceptionObjectNode* *n2*)) \vee
 ((*is-FrameState* *n1*) \wedge (*is-FrameState* *n2*)) \vee
 ((*is-IfNode* *n1*) \wedge (*is-IfNode* *n2*)) \vee
 ((*is-IntegerBelowNode* *n1*) \wedge (*is-IntegerBelowNode* *n2*)) \vee
 ((*is-IntegerEqualsNode* *n1*) \wedge (*is-IntegerEqualsNode* *n2*)) \vee
 ((*is-IntegerLessThanNode* *n1*) \wedge (*is-IntegerLessThanNode* *n2*)) \vee
 ((*is-InvokeNode* *n1*) \wedge (*is-InvokeNode* *n2*)) \vee
 ((*is-InvokeWithExceptionNode* *n1*) \wedge (*is-InvokeWithExceptionNode* *n2*)) \vee
 ((*is-IsNullNode* *n1*) \wedge (*is-IsNullNode* *n2*)) \vee
 ((*is-KillingBeginNode* *n1*) \wedge (*is-KillingBeginNode* *n2*)) \vee
 ((*is-LeftShiftNode* *n1*) \wedge (*is-LeftShiftNode* *n2*)) \vee

```

((is-LoadFieldNode n1) ∧ (is-LoadFieldNode n2)) ∨
((is-LogicNegationNode n1) ∧ (is-LogicNegationNode n2)) ∨
((is-LoopBeginNode n1) ∧ (is-LoopBeginNode n2)) ∨
((is-LoopEndNode n1) ∧ (is-LoopEndNode n2)) ∨
((is-LoopExitNode n1) ∧ (is-LoopExitNode n2)) ∨
((is-MergeNode n1) ∧ (is-MergeNode n2)) ∨
((is-MethodCallTargetNode n1) ∧ (is-MethodCallTargetNode n2)) ∨
((is-MulNode n1) ∧ (is-MulNode n2)) ∨
((is-NarrowNode n1) ∧ (is-NarrowNode n2)) ∨
((is-NegateNode n1) ∧ (is-NegateNode n2)) ∨
((is-NewArrayNode n1) ∧ (is-NewArrayNode n2)) ∨
((is-NewInstanceNode n1) ∧ (is-NewInstanceNode n2)) ∨
((is-NotNode n1) ∧ (is-NotNode n2)) ∨
((is-OrNode n1) ∧ (is-OrNode n2)) ∨
((is-ParameterNode n1) ∧ (is-ParameterNode n2)) ∨
((is-PiNode n1) ∧ (is-PiNode n2)) ∨
((is-ReturnNode n1) ∧ (is-ReturnNode n2)) ∨
((is-RightShiftNode n1) ∧ (is-RightShiftNode n2)) ∨
((is-ShortCircuitOrNode n1) ∧ (is-ShortCircuitOrNode n2)) ∨
((is-SignedDivNode n1) ∧ (is-SignedDivNode n2)) ∨
((is-SignedRemNode n1) ∧ (is-SignedRemNode n2)) ∨
((is-SignExtendNode n1) ∧ (is-SignExtendNode n2)) ∨
((is-StartNode n1) ∧ (is-StartNode n2)) ∨
((is-StoreFieldNode n1) ∧ (is-StoreFieldNode n2)) ∨
((is-SubNode n1) ∧ (is-SubNode n2)) ∨
((is-UnsignedRightShiftNode n1) ∧ (is-UnsignedRightShiftNode 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)) ∨
((is-ZeroExtendNode n1) ∧ (is-ZeroExtendNode n2)))

```

end

3.3 IR Graph Type

```

theory IRGraph
  imports
    IRNodeHierarchy
    Stamp
    HOL-Library.FSet
    HOL.Relation
  begin

```

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

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

```

typedef IRGraph = {g :: ID ⇒ (IRNode × Stamp) . finite (dom g)}

```

$\langle proof \rangle$

setup-lifting *type-definition-IRGraph*

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

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

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

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

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*) $\langle proof \rangle$

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

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*) $\langle proof \rangle$

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

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

lift-definition *irgraph* :: (*ID* \times (*IRNode* \times *Stamp*)) *list* \Rightarrow *IRGraph*
is *map-of* \circ *no-node*
 $\langle proof \rangle$

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

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

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

notation (*latex*)
domain-subtraction ($\cdot \Leftarrow \cdot$)

code-datatype *irgraph*

fun *filter-none* **where**

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

lemma *no-node-clears*:

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

<proof>

lemma *dom-eq*:

assumes $\forall x \in \text{set } \text{xs}. \text{fst } (\text{snd } x) \neq \text{NoNode}$

shows *filter-none* (*map-of xs*) = *dom* (*map-of xs*)

<proof>

lemma *fil-eq*:

filter-none (*map-of* (*no-node xs*)) = *set* (*map fst* (*no-node xs*))

<proof>

lemma *irgraph[code]*: *ids* (*irgraph m*) = *set* (*map fst* (*no-node m*))

<proof>

lemma *[code]*: *Rep-IRGraph* (*irgraph m*) = *map-of* (*no-node m*)

<proof>

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

inputs g nid = *set* (*inputs-of* (*kind g nid*))

— Get the successor set of a given node ID

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

succ g nid = *set* (*successors-of* (*kind g nid*))

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

fun *input-edges* :: *IRGraph* \Rightarrow *ID rel* **where**

input-edges g = ($\bigcup i \in \text{ids } g. \{(i,j) | j. j \in (\text{inputs } g i)\}$)

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

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

usages g nid = {*i*. *i* ∈ *ids g* ∧ *nid* ∈ *inputs g i*}

fun *successor-edges* :: *IRGraph* \Rightarrow *ID rel* **where**

successor-edges g = ($\bigcup i \in \text{ids } g. \{(i,j) | j. j \in (\text{succ } g i)\}$)

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

predecessors g nid = {*i*. *i* ∈ *ids g* ∧ *nid* ∈ *succ g i*}

fun *nodes-of* :: *IRGraph* \Rightarrow (*IRNode* \Rightarrow *bool*) \Rightarrow *ID set* **where**

nodes-of g sel = {*nid* ∈ *ids g* . *sel* (*kind g nid*)}

fun *edge* :: (*IRNode* \Rightarrow 'a) \Rightarrow *ID* \Rightarrow *IRGraph* \Rightarrow 'a **where**

edge sel nid g = *sel* (*kind g nid*)

fun *filtered-inputs* :: *IRGraph* \Rightarrow *ID* \Rightarrow (*IRNode* \Rightarrow *bool*) \Rightarrow *ID list* **where**

filtered-inputs g nid f = *filter* (*f* ∘ (*kind g*)) (*inputs-of* (*kind g nid*))

fun *filtered-successors* :: *IRGraph* \Rightarrow *ID* \Rightarrow (*IRNode* \Rightarrow *bool*) \Rightarrow *ID list* **where**

filtered-successors g nid f = *filter* (*f* ∘ (*kind g*)) (*successors-of* (*kind g nid*))

fun *filtered-usages* :: *IRGraph* \Rightarrow *ID* \Rightarrow (*IRNode* \Rightarrow *bool*) \Rightarrow *ID* *set* **where**
filtered-usages *g* *nid* *f* = { *n* \in (*usages* *g* *nid*). *f* (*kind* *g* *n*) }

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

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

lemma *ids-some*[*simp*]: *x* \in *ids* *g* \longleftrightarrow *kind* *g* *x* \neq *NoNode*
 \langle *proof* \rangle

lemma *not-in-g*:
assumes *nid* \notin *ids* *g*
shows *kind* *g* *nid* = *NoNode*
 \langle *proof* \rangle

lemma *valid-creation*[*simp*]:
finite (*dom* *g*) \longleftrightarrow *Rep-IRGraph* (*Abs-IRGraph* *g*) = *g*
 \langle *proof* \rangle

lemma [*simp*]: *finite* (*ids* *g*)
 \langle *proof* \rangle

lemma [*simp*]: *finite* (*ids* (*irgraph* *g*))
 \langle *proof* \rangle

lemma [*simp*]: *finite* (*dom* *g*) \longrightarrow *ids* (*Abs-IRGraph* *g*) = { *nid* \in *dom* *g* . \nexists *s*. *g* *nid* = *Some* (*NoNode*, *s*) }
 \langle *proof* \rangle

lemma [*simp*]: *finite* (*dom* *g*) \longrightarrow *kind* (*Abs-IRGraph* *g*) = (λx . (*case* *g* *x* of *None* \Rightarrow *NoNode* | *Some* *n* \Rightarrow *fst* *n*))
 \langle *proof* \rangle

lemma [*simp*]: *finite* (*dom* *g*) \longrightarrow *stamp* (*Abs-IRGraph* *g*) = (λx . (*case* *g* *x* of *None* \Rightarrow *IllegalStamp* | *Some* *n* \Rightarrow *snd* *n*))
 \langle *proof* \rangle

lemma [*simp*]: *ids* (*irgraph* *g*) = *set* (*map* *fst* (*no-node* *g*))
 \langle *proof* \rangle

lemma [*simp*]: *kind* (*irgraph* *g*) = (λnid . (*case* (*map-of* (*no-node* *g*)) *nid* of *None* \Rightarrow *NoNode* | *Some* *n* \Rightarrow *fst* *n*))
 \langle *proof* \rangle

lemma [*simp*]: *stamp* (*irgraph* *g*) = (λnid . (*case* (*map-of* (*no-node* *g*)) *nid* of *None* \Rightarrow *IllegalStamp* | *Some* *n* \Rightarrow *snd* *n*))
 \langle *proof* \rangle

lemma *map-of-upd*: $(\text{map-of } g)(k \mapsto v) = (\text{map-of } ((k, v) \# g))$
 $\langle \text{proof} \rangle$

lemma *[code]: replace-node* $\text{id } k (\text{irgraph } g) = (\text{irgraph } ((\text{id}, k) \# g))$
 $\langle \text{proof} \rangle$

lemma *[code]: add-node* $\text{id } k (\text{irgraph } g) = (\text{irgraph } (((\text{id}, k) \# g)))$
 $\langle \text{proof} \rangle$

lemma *add-node-lookup*:
 $\text{gup} = \text{add-node } \text{id } (k, s) g \longrightarrow$
 $(\text{if } k \neq \text{NoNode} \text{ then } \text{kind } \text{gup } \text{id} = k \wedge \text{stamp } \text{gup } \text{id} = s \text{ else } \text{kind } \text{gup } \text{id} = \text{kind } g \text{ id})$
 $\langle \text{proof} \rangle$

lemma *remove-node-lookup*:
 $\text{gup} = \text{remove-node } \text{id } g \longrightarrow \text{kind } \text{gup } \text{id} = \text{NoNode} \wedge \text{stamp } \text{gup } \text{id} = \text{IllegalStamp}$
 $\langle \text{proof} \rangle$

lemma *replace-node-lookup[simp]*:
 $\text{gup} = \text{replace-node } \text{id } (k, s) g \wedge k \neq \text{NoNode} \longrightarrow \text{kind } \text{gup } \text{id} = k \wedge \text{stamp } \text{gup } \text{id} = s$
 $\langle \text{proof} \rangle$

lemma *replace-node-unchanged*:
 $\text{gup} = \text{replace-node } \text{id } (k, s) g \longrightarrow (\forall n \in (\text{ids } g - \{\text{id}\}) . n \in \text{ids } g \wedge n \in \text{ids } \text{gup} \wedge \text{kind } g n = \text{kind } \text{gup } n)$
 $\langle \text{proof} \rangle$

3.3.1 Example Graphs

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

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

Example 2: public static int sq(int x) return x * x;
 $[1 \text{ P}(0)] \text{ / } [0 \text{ Start}] [4 *] \mid \text{ / } \text{V} \text{ / } [5 \text{ Return}]$

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

```

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 ⟨proof⟩

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 $\langle \text{proof} \rangle$

inductive *nodeEq* :: ($ID \rightarrow ID$) \Rightarrow $IRGraph \Rightarrow ID \Rightarrow IRGraph \Rightarrow ID \Rightarrow \text{bool}$
where

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

code-pred [*show-modes*] *nodeEq* $\langle \text{proof} \rangle$

fun *diffNodesGraph* :: $IRGraph \Rightarrow IRGraph \Rightarrow ID \text{ set}$ **where**

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

fun *diffNodesInfo* :: $IRGraph \Rightarrow IRGraph \Rightarrow (ID \times IRNode \times IRNode) \text{ set}$ (**infix** $\cap_s \ 20$)
where

diffNodesInfo $g1 \ g2 = \{(nid, \text{kind } g1 \ nid, \text{kind } g2 \ nid) \mid nid . nid \in \text{diffNodesGraph}$
 $g1 \ g2\}$

fun *eqGraph* :: $IRGraph \Rightarrow IRGraph \Rightarrow \text{bool}$ (**infix** $\approx_s \ 20$)

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

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*:

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

definition *MaxOrNeg* :: $\text{nat set} \Rightarrow \text{int}$ **where**

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

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 $max-bit$: $bit\ (v::('a::len)\ word)\ n \implies n < size\ v$
 $\langle proof \rangle$

lemma $max-set-bit$: $MaxOrNeg\ \{n . bit\ (v::('a::len)\ word)\ n\} < Nat.size\ v$
 $\langle proof \rangle$

4.1 Long.numberOfLeadingZeros

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

lemma $MaxOrNeg-neg$: $MaxOrNeg\ \{\} = -1$
 $\langle proof \rangle$

lemma $MaxOrNeg-max$: $s \neq \{\} \implies MaxOrNeg\ s = Max\ s$
 $\langle proof \rangle$

lemma $zero-no-bits$:
 $\{n . bit\ 0\ n\} = \{\}$
 $\langle proof \rangle$

lemma $highestOneBit\ (0::64\ word) = -1$
 $\langle proof \rangle$

lemma $numberOfLeadingZeros\ (0::64\ word) = 64$
 $\langle proof \rangle$

lemma $highestOneBit-top$: $Max\ \{highestOneBit\ (v::64\ word)\} < 64$
 $\langle proof \rangle$

lemma $numberOfLeadingZeros-top$: $Max\ \{numberOfLeadingZeros\ (v::64\ word)\} \leq 64$
 $\langle proof \rangle$

lemma $numberOfLeadingZeros-range$: $0 \leq numberOfLeadingZeros\ a \wedge numberOfLeadingZeros\ a \leq Nat.size\ a$
 $\langle proof \rangle$

lemma *leadingZerosAddHighestOne*: $\text{numberOfLeadingZeros } v + \text{highestOneBit } v = \text{Nat.size } v - 1$
 ⟨proof⟩

4.2 Long.numberOfTrailingZeros

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

lemma *lowestOneBit-bot*: $\text{lowestOneBit } (0::64 \text{ word}) = 64$
 ⟨proof⟩

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

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

lemma *numberOfTrailingZeros* $(0::64 \text{ word}) = 64$
 ⟨proof⟩

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* $0 = 0$
 ⟨proof⟩

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*: $\text{finite } \{n . n < \text{Nat.size } v \wedge \neg(\text{bit } v \ n)\}$
 ⟨proof⟩

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

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

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

lemma *card-of-range*:

$x = \text{card } \{n . 0 \leq n \wedge n < x\}$

$\langle \text{proof} \rangle$

lemma *range-of-nat*:

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

$\langle \text{proof} \rangle$

lemma *finite-range*:

$\text{finite } \{n::\text{nat} . n < x\}$

$\langle \text{proof} \rangle$

lemma *range-eq*:

fixes $x y :: \text{nat}$

shows $\text{card } \{y..<x\} = \text{card } \{y<..x\}$

$\langle \text{proof} \rangle$

lemma *card-of-range-bound*:

fixes $x y :: \text{nat}$

assumes $x > y$

shows $x - y = \text{card } \{n . y < n \wedge n \leq x\}$

$\langle \text{proof} \rangle$

lemma *bitCount* $(-1::('a::\text{len}) \text{ word}) = \text{LENGTH}('a)$

$\langle \text{proof} \rangle$

lemma *bitCount-range*:

fixes $n :: ('a::\text{len}) \text{ word}$

shows $0 \leq \text{bitCount } n \wedge \text{bitCount } n \leq \text{Nat.size } n$

$\langle \text{proof} \rangle$

lemma *zerosAboveHighestOne*:

$n > \text{highestOneBit } a \implies \neg(\text{bit } a \ n)$

$\langle \text{proof} \rangle$

lemma *zerosBelowLowestOne*:

assumes $n < \text{lowestOneBit } a$

shows $\neg(\text{bit } a \ n)$

$\langle \text{proof} \rangle$

lemma *union-bit-sets*:

fixes $a :: ('a::\text{len}) \text{ word}$

shows $\{n . n < \text{Nat.size } a \wedge \text{bit } a \ n\} \cup \{n . n < \text{Nat.size } a \wedge \neg(\text{bit } a \ n)\} = \{n . n < \text{Nat.size } a\}$

$\langle \text{proof} \rangle$

lemma *disjoint-bit-sets*:

fixes $a :: ('a::\text{len}) \text{ word}$


```

shows  $\{n . n < \text{Nat.size } a \wedge \text{bit } a \ n\} \cap \{n . n < \text{Nat.size } a \wedge \neg(\text{bit } a \ n)\} = \{\}$ 
 $\langle \text{proof} \rangle$ 

lemma qualified-bitCount:
   $\text{bitCount } v = \text{card } \{n . n < \text{Nat.size } v \wedge \text{bit } v \ n\}$ 
 $\langle \text{proof} \rangle$ 

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  $\text{card } z - \text{card } y = \text{card } x$ 
 $\langle \text{proof} \rangle$ 

lemma card-add:
  assumes  $\text{finite } x \wedge \text{finite } y \wedge \text{finite } z$ 
  assumes  $x \cup y = z$ 
  assumes  $y \cap x = \{\}$ 
  shows  $\text{card } x + \text{card } y = \text{card } z$ 
 $\langle \text{proof} \rangle$ 

lemma card-add-inverses:
  assumes  $\text{finite } \{n . Q \ n \wedge \neg(P \ n)\} \wedge \text{finite } \{n . Q \ n \wedge P \ n\} \wedge \text{finite } \{n . Q \ n\}$ 
  shows  $\text{card } \{n . Q \ n \wedge P \ n\} + \text{card } \{n . Q \ n \wedge \neg(P \ n)\} = \text{card } \{n . Q \ n\}$ 
 $\langle \text{proof} \rangle$ 

lemma ones-zero-sum-to-width:
   $\text{bitCount } a + \text{zeroCount } a = \text{Nat.size } a$ 
 $\langle \text{proof} \rangle$ 

lemma intersect-bitCount-helper:
   $\text{card } \{n . n < \text{Nat.size } a\} - \text{bitCount } a = \text{card } \{n . n < \text{Nat.size } a \wedge \neg(\text{bit } a \ n)\}$ 
 $\langle \text{proof} \rangle$ 

lemma intersect-bitCount:
   $\text{Nat.size } a - \text{bitCount } a = \text{card } \{n . n < \text{Nat.size } a \wedge \neg(\text{bit } a \ n)\}$ 
 $\langle \text{proof} \rangle$ 

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 calculated during the traversal of the control flow graph.

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

```
type-synonym ID = nat
type-synonym MapState = ID  $\Rightarrow$  Value
type-synonym Params = Value list
```

```
definition new-map-state :: MapState where
  new-map-state = ( $\lambda x$ ..UndefVal)
```

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  $\Rightarrow$  bool where
  is-ground (UnaryExpr op e) = is-ground e |
  is-ground (BinaryExpr op e1 e2) = (is-ground e1  $\wedge$  is-ground e2) |
  is-ground (ConditionalExpr b e1 e2) = (is-ground b  $\wedge$  is-ground e1  $\wedge$  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 }
  (proof)

```

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 \equiv { *BinShortCircuitOr*, *BinIntegerEquals*, *BinIntegerLessThan*, *BinIntegerBelow* }

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

binary-shift-ops $\equiv \{BinLeftShift, BinRightShift, BinURightShift\}$

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

fun *stamp-unary* :: *IRUnaryOp* \Rightarrow *Stamp* \Rightarrow *Stamp* **where**

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

stamp-unary op - = IllegalStamp

fun *stamp-binary* :: *IRBinaryOp* \Rightarrow *Stamp* \Rightarrow *Stamp* \Rightarrow *Stamp* **where**
stamp-binary op (IntegerStamp b1 lo1 hi1) (IntegerStamp b2 lo2 hi2) =
(if op \in binary-shift-ops then unrestricted-stamp (IntegerStamp b1 lo1 hi1)
else if b1 \neq b2 then IllegalStamp else
(if op \in 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* \Rightarrow *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**

$\text{bin-eval BinAdd } v1 \ v2 = \text{intval-add } v1 \ v2 \mid$
 $\text{bin-eval BinMul } v1 \ v2 = \text{intval-mul } v1 \ v2 \mid$
 $\text{bin-eval BinSub } v1 \ v2 = \text{intval-sub } v1 \ v2 \mid$
 $\text{bin-eval BinAnd } v1 \ v2 = \text{intval-and } v1 \ v2 \mid$
 $\text{bin-eval BinOr } v1 \ v2 = \text{intval-or } v1 \ v2 \mid$
 $\text{bin-eval BinXor } v1 \ v2 = \text{intval-xor } v1 \ v2 \mid$
 $\text{bin-eval BinShortCircuitOr } v1 \ v2 = \text{intval-short-circuit-or } v1 \ v2 \mid$
 $\text{bin-eval BinLeftShift } v1 \ v2 = \text{intval-left-shift } v1 \ v2 \mid$
 $\text{bin-eval BinRightShift } v1 \ v2 = \text{intval-right-shift } v1 \ v2 \mid$
 $\text{bin-eval BinURightShift } v1 \ v2 = \text{intval-uright-shift } v1 \ v2 \mid$
 $\text{bin-eval BinIntegerEquals } v1 \ v2 = \text{intval-equals } v1 \ v2 \mid$
 $\text{bin-eval BinIntegerLessThan } v1 \ v2 = \text{intval-less-than } v1 \ v2 \mid$
 $\text{bin-eval BinIntegerBelow } v1 \ v2 = \text{intval-below } v1 \ v2$

lemmas *eval-thms* =

intval-abs.simps $\text{intval-negate.simps}$ intval-not.simps
 $\text{intval-logic-negation.simps}$ $\text{intval-narrow.simps}$
 $\text{intval-sign-extend.simps}$ $\text{intval-zero-extend.simps}$
 intval-add.simps intval-mul.simps intval-sub.simps
 intval-and.simps intval-or.simps intval-xor.simps
 $\text{intval-left-shift.simps}$ $\text{intval-right-shift.simps}$
 $\text{intval-uright-shift.simps}$ $\text{intval-equals.simps}$
 $\text{intval-less-than.simps}$ $\text{intval-below.simps}$

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:

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

ParameterExpr:

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

ConditionalExpr:

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

$result \neq \text{UndefVal}$
 $\implies [m,p] \vdash (\text{ConditionalExpr } ce \text{ te } fe) \mapsto result \mid$

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

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

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

code-pred (*modes: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ as evalT*)
 $[\text{show-steps}, \text{show-mode-inference}, \text{show-intermediate-results}]$
 $\text{evaltree } \langle \text{proof} \rangle$

inductive

$\text{evaltrees} :: \text{MapState} \Rightarrow \text{Params} \Rightarrow \text{IRExpr list} \Rightarrow \text{Value list} \Rightarrow \text{bool } ([-,] \vdash - \mapsto_L$
- 55)

for $m \ p$ where

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

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

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

definition $\text{sq-param0} :: \text{IRExpr}$ **where**

$\text{sq-param0} = \text{BinaryExpr } \text{BinMul}$
 $(\text{ParameterExpr } 0 \ (\text{IntegerStamp } 32 \ (- \ 2147483648) \ 2147483647))$
 $(\text{ParameterExpr } 0 \ (\text{IntegerStamp } 32 \ (- \ 2147483648) \ 2147483647))$

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

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*
 $\langle proof \rangle$

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*]:
 $(e2 \leq e1) \longleftrightarrow (\forall m p v. (([m,p] \vdash e1 \mapsto v) \longrightarrow ([m,p] \vdash e2 \mapsto v)))$

definition

lt-expr-def [*simp*]:
 $(e1 < e2) \longleftrightarrow (e1 \leq e2 \wedge \neg (e1 \doteq e2))$

instance $\langle proof \rangle$

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 :: IRExp ⇒ int64 (↑)
  fixes down :: IRExp ⇒ int64 (↓)
  assumes up-spec: [m, p] ⊢ e ↦ IntVal b v ⇒ (and v (not ((ucast (↑e))))) = 0
  and down-spec: [m, p] ⊢ e ↦ IntVal b v ⇒ (and (not v) (ucast (↓e))) = 0
begin

```

```

lemma may-implies-either:
  [m, p] ⊢ e ↦ IntVal b v ⇒ bit (↑e) n ⇒ bit v n = False ∨ bit v n = True
  ⟨proof⟩

```

```

lemma not-may-implies-false:
  [m, p] ⊢ e ↦ IntVal b v ⇒ ¬(bit (↑e) n) ⇒ bit v n = False
  ⟨proof⟩

```

```

lemma must-implies-true:
  [m, p] ⊢ e ↦ IntVal b v ⇒ bit (↓e) n ⇒ bit v n = True
  ⟨proof⟩

```

```

lemma not-must-implies-either:
  [m, p] ⊢ e ↦ IntVal b v ⇒ ¬(bit (↓e) n) ⇒ bit v n = False ∨ bit v n = True
  ⟨proof⟩

```

```

lemma must-implies-may:
  [m, p] ⊢ e ↦ IntVal b v ⇒ n < 32 ⇒ bit (↓e) n ⇒ bit (↑e) n
  ⟨proof⟩

```

```

lemma up-mask-and-zero-implies-zero:
  assumes and (↑x) (↑y) = 0
  assumes [m, p] ⊢ x ↦ IntVal b xv
  assumes [m, p] ⊢ y ↦ IntVal b yv
  shows and xv yv = 0
  ⟨proof⟩

```

```

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

```

```

end

```


definition $IRExpr\text{-}up :: IExpr \Rightarrow int64$ **where**
 $IRExpr\text{-}up\ e = not\ 0$

definition $IRExpr\text{-}down :: IExpr \Rightarrow int64$ **where**
 $IRExpr\text{-}down\ e = 0$

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

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

interpretation $simple\text{-}mask: stamp\text{-}mask$
 $IRExpr\text{-}up :: IExpr \Rightarrow int64$
 $IRExpr\text{-}down :: IExpr \Rightarrow int64$
 $\langle proof \rangle$

end

5.6 Data-flow Tree Theorems

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

5.6.1 Deterministic Data-flow Evaluation

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

lemma $evalAllDet:$
 $[m,p] \vdash e \mapsto_L v1 \implies$
 $[m,p] \vdash e \mapsto_L v2 \implies$
 $v1 = v2$
 $\langle proof \rangle$

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 $unary\text{-}eval\text{-}not\text{-}obj\text{-}ref:$
shows $unary\text{-}eval\ op\ x \neq ObjRef\ v$
 $\langle proof \rangle$

lemma *unary-eval-not-obj-str*:
shows *unary-eval op x \neq ObjStr v*
<proof>

lemma *unary-eval-int*:
assumes *def: unary-eval op x \neq UndefVal*
shows *is-IntVal (unary-eval op x)*
<proof>

lemma *bin-eval-int*:
assumes *def: bin-eval op x y \neq UndefVal*
shows *is-IntVal (bin-eval op x y)*
<proof>

lemma *IntVal0*:
(IntVal 32 0) = (new-int 32 0)
<proof>

lemma *IntVal1*:
(IntVal 32 1) = (new-int 32 1)
<proof>

lemma *bin-eval-new-int*:
assumes *def: bin-eval op x y \neq UndefVal*
shows $\exists b v. (bin-eval op x y) = new-int b v \wedge$
 $b = (if\ op \in binary-fixed-32-ops\ then\ 32\ else\ intval-bits\ x)$
<proof>

lemma *int-stamp*:
assumes *i: is-IntVal v*
shows *is-IntegerStamp (constantAsStamp v)*
<proof>

lemma *validStampIntConst*:
assumes *v = IntVal b ival*
assumes *0 < b \wedge b \leq 64*
shows *valid-stamp (constantAsStamp v)*
<proof>

lemma *validDefIntConst*:
assumes $v: v = \text{IntVal } b \text{ ival}$
assumes $0 < b \wedge b \leq 64$
assumes $\text{take-bit } b \text{ ival} = \text{ival}$
shows $\text{valid-value } v \text{ (constantAsStamp } v)$
 $\langle \text{proof} \rangle$

5.6.3 Evaluation Results are Valid

A valid value cannot be *UndefVal*.

lemma *valid-not-undef*:
assumes $a1: \text{valid-value } val \text{ } s$
assumes $a2: s \neq \text{VoidStamp}$
shows $val \neq \text{UndefVal}$
 $\langle \text{proof} \rangle$

lemma *valid-VoidStamp[elim]*:
shows $\text{valid-value } val \text{ VoidStamp} \implies$
 $val = \text{UndefVal}$
 $\langle \text{proof} \rangle$

lemma *valid-ObjStamp[elim]*:
shows $\text{valid-value } val \text{ (ObjectStamp klass exact nonNull alwaysNull)} \implies$
 $(\exists v. val = \text{ObjRef } v)$
 $\langle \text{proof} \rangle$

lemma *valid-int[elim]*:
shows $\text{valid-value } val \text{ (IntegerStamp b lo hi)} \implies$
 $(\exists v. val = \text{IntVal } b \text{ } v)$
 $\langle \text{proof} \rangle$

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

lemma *evaltree-not-undef*:
fixes $m \text{ } p \text{ } e \text{ } v$
shows $([m, p] \vdash e \mapsto v) \implies v \neq \text{UndefVal}$
 $\langle \text{proof} \rangle$

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

lemma *default-stamp* [simp]: *default-stamp* = *IntegerStamp* 32 (-2147483648)
 2147483647
 <proof>

lemma *valid-value-signed-int-range* [simp]:
 assumes *valid-value* *val* (*IntegerStamp* *b* *lo* *hi*)
 assumes *lo* < 0
 shows $\exists v. (val = \text{IntVal } b \ v \wedge$
 $lo \leq \text{int-signed-value } b \ v \wedge$
 $\text{int-signed-value } b \ v \leq hi)$
 <proof>

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 $x \geq x'$
 shows $(\text{UnaryExpr } op \ x) \geq (\text{UnaryExpr } op \ x')$
 <proof>

lemma *mono-binary*:
 assumes $x \geq x'$
 assumes $y \geq y'$
 shows $(\text{BinaryExpr } op \ x \ y) \geq (\text{BinaryExpr } op \ x' \ y')$
 <proof>

lemma *never-void*:
 assumes $[m, p] \vdash x \mapsto xv$
 assumes *valid-value* *xv* (*stamp-expr* *xe*)
 shows *stamp-expr* *xe* \neq *VoidStamp*
 <proof>

lemma *compatible-trans*:
 compatible *x* *y* \wedge compatible *y* *z* \implies compatible *x* *z*

$\langle proof \rangle$

lemma *compatible-refl*:

compatible $x\ y \implies \text{compatible } y\ x$

$\langle proof \rangle$

lemma *mono-conditional*:

assumes $c \geq c'$

assumes $t \geq t'$

assumes $f \geq f'$

shows $(\text{ConditionalExpr } c\ t\ f) \geq (\text{ConditionalExpr } c'\ t'\ f')$

$\langle proof \rangle$

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

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

lemma *unfold-const*:

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

$\langle proof \rangle$

lemma *unfold-binary*:

shows $([m,p] \vdash \text{BinaryExpr } op\ xe\ ye \mapsto val) = (\exists\ x\ y.$

$(([m,p] \vdash xe \mapsto x) \wedge$

$([m,p] \vdash ye \mapsto y) \wedge$

$(val = \text{bin-eval } op\ x\ y) \wedge$

$(val \neq \text{UndefVal})$

$))\ (\text{is } ?L = ?R)$

$\langle proof \rangle$

lemma *unfold-unary*:

shows $([m,p] \vdash \text{UnaryExpr } op\ xe \mapsto val)$

$= (\exists\ x.$

$(([m,p] \vdash xe \mapsto x) \wedge$

$(val = \text{unary-eval } op\ x) \wedge$

$(val \neq \text{UndefVal})$

$))\ (\text{is } ?L = ?R)$

$\langle proof \rangle$

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 $\exists b v. \text{unary-eval } op \ x = \text{new-int } b \ v \wedge$

$b = (\text{if } op \in \text{normal-unary} \text{ then } \text{intval-bits } x \text{ else } \text{ir-resultBits } op)$

<proof>

lemma *new-int-unused-bits-zero*:

assumes *IntVal b ival = new-int b ival0*

shows *take-bit b ival = ival*

<proof>

lemma *unary-eval-unused-bits-zero*:

assumes *unary-eval op x = IntVal b ival*

shows *take-bit b ival = ival*

<proof>

lemma *bin-eval-unused-bits-zero*:

assumes *bin-eval op x y = (IntVal b ival)*

shows *take-bit b ival = ival*

<proof>

lemma *eval-unused-bits-zero*:

$[m,p] \vdash xe \mapsto (\text{IntVal } b \ ix) \implies \text{take-bit } b \ ix = ix$

<proof>

lemma *unary-normal-bitsize*:

assumes *unary-eval op x = IntVal b ival*

assumes *op ∈ normal-unary*

shows $\exists ix. x = \text{IntVal } b \ ix$

<proof>

lemma *unary-not-normal-bitsize*:

assumes *unary-eval op x = IntVal b ival*

assumes *op ∉ normal-unary*

shows $b = \text{ir-resultBits } op \wedge 0 < b \wedge b \leq 64$

<proof>

lemma *unary-eval-bitsize*:

assumes *unary-eval op x = IntVal b ival*

assumes $2: x = \text{IntVal } bx \ ix$

assumes $0 < bx \wedge bx \leq 64$

shows $0 < b \wedge b \leq 64$

⟨proof⟩

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⟩

lemma *eval-bits-1-64*:

$[m,p] \vdash xe \mapsto (IntVal\ b\ ix) \implies 0 < b \wedge b \leq 64$

⟨proof⟩

lemma *unfold-binary-width*:

assumes *op ∉ binary-fixed-32-ops ∧ op ∉ binary-shift-ops*

shows $([m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto IntVal\ b\ val) = (\exists\ x\ y.$

$([m,p] \vdash xe \mapsto IntVal\ b\ x) \wedge$

$([m,p] \vdash ye \mapsto IntVal\ b\ y) \wedge$

$(IntVal\ b\ val = bin-eval\ op\ (IntVal\ b\ x)\ (IntVal\ b\ y)) \wedge$

$(IntVal\ b\ val \neq UndefinedVal)$

$) \rangle$ **(is ?L = ?R)**

⟨proof⟩

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* |

$is_preevaluated \ (SignedDivNode \ n \ - \ - \ - \ -) = True \mid$
 $is_preevaluated \ (SignedRemNode \ n \ - \ - \ - \ -) = True \mid$
 $is_preevaluated \ (ValuePhiNode \ n \ -) = True \mid$
 $is_preevaluated \ - = False$

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:

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

LogicNegationNode:

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

AddNode:

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

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

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

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

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

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

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

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

RightShiftNode:
 $\llbracket \text{kind } g \ n = \text{RightShiftNode } x \ y;$

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

UnsignedRightShiftNode:

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

IntegerBelowNode:

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

IntegerEqualsNode:

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

IntegerLessThanNode:

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

NarrowNode:

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

SignExtendNode:

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

ZeroExtendNode:

$$\begin{aligned}
&[[\text{kind } g \ n = \text{ZeroExtendNode } \text{inputBits } \text{resultBits } x; \\
&\quad g \vdash x \simeq xe]] \\
\implies &g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits}) \ xe) \mid
\end{aligned}$$

LeafNode:

$$\begin{aligned}
&[[\text{is-preevaluated } (\text{kind } g \ n); \\
&\quad \text{stamp } g \ n = s]] \\
\implies &g \vdash n \simeq (\text{LeafExpr } n \ s) \mid
\end{aligned}$$

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* $\langle \text{proof} \rangle$

inductive

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

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

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

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ as *exprListE*) *replist* $\langle \text{proof} \rangle$

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* |
unary-node *UnaryNot* *v* = *NotNode* *v* |
unary-node *UnaryNeg* *v* = *NegateNode* *v* |
unary-node *UnaryLogicNegation* *v* = *LogicNegationNode* *v* |
unary-node (*UnaryNarrow* *ib rb*) *v* = *NarrowNode* *ib rb v* |
unary-node (*UnarySignExtend* *ib rb*) *v* = *SignExtendNode* *ib rb v* |
unary-node (*UnaryZeroExtend* *ib rb*) *v* = *ZeroExtendNode* *ib rb v*

fun *bin-node* :: *IRBinaryOp* \Rightarrow *ID* \Rightarrow *ID* \Rightarrow *IRNode* **where**

bin-node *BinAdd* *x y* = *AddNode* *x y* |
bin-node *BinMul* *x y* = *MulNode* *x y* |
bin-node *BinSub* *x y* = *SubNode* *x y* |
bin-node *BinAnd* *x y* = *AndNode* *x y* |
bin-node *BinOr* *x y* = *OrNode* *x y* |

```

bin-node BinXor x y = XorNode x y |
bin-node BinShortCircuitOr x y = ShortCircuitOrNode x y |
bin-node BinLeftShift x y = LeftShiftNode x y |
bin-node BinRightShift x y = RightShiftNode x y |
bin-node BinURightShift x y = UnsignedRightShiftNode x y |
bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
bin-node BinIntegerLessThan x y = IntegerLessThanNode x y |
bin-node BinIntegerBelow x y = IntegerBelowNode x y

```

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

code-pred *fresh-id* \langle *proof* \rangle

fun *get-fresh-id* :: *IRGraph* \Rightarrow *ID* **where**

get-fresh-id g = *last(sorted-list-of-set(ids g)) + 1*

export-code *get-fresh-id*

value *get-fresh-id eg2-sq*

value *get-fresh-id* (*add-node 6 (ParameterNode 2, default-stamp) eg2-sq*)

inductive

unrep :: *IRGraph* \Rightarrow *IRExpr* \Rightarrow (*IRGraph* \times *ID*) \Rightarrow *bool* (*-* \oplus *-* \rightsquigarrow *-* 55)
where

ConstantNodeSame:

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

ConstantNodeNew:

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

ParameterNodeSame:

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

ParameterNodeNew:

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

$$\implies g \oplus (\text{ParameterExpr } i \ s) \rightsquigarrow (g', n) \mid$$

ConditionalNodeSame:

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

ConditionalNodeNew:

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

UnaryNodeSame:

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

UnaryNodeNew:

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

BinaryNodeSame:

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

BinaryNodeNew:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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*
 $(\vdash \cdot \leq \cdot \ 50)$
where
 $(g \vdash n \leq e) = (\exists \ e'. (g \vdash n \simeq e') \wedge (e' \leq e))$

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

lemma *graph-refinement*:

graph-refinement *g1 g2* $\implies (\forall \ n \ m \ p \ v. n \in ids \ g1 \longrightarrow ([g1, m, p] \vdash n \mapsto v) \longrightarrow$
 $([g2, m, p] \vdash n \mapsto v))$
 $\langle proof \rangle$

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* =
 $(\forall \ n \in ids \ g .$
 $inputs \ g \ n \subseteq ids \ g \wedge$

$$\text{succ } g \ n \subseteq \text{ids } g \wedge \\ \text{kind } g \ n \neq \text{NoNode})$$

definition *wf-phs* **where**

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

definition *wf-ends* **where**

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

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

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

lemmas *wf-folds* =

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

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

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

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

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

lemma *wf-empty*: *wf-graph start-end-graph*

<proof>

lemma *wf-eg2-sq*: *wf-graph eg2-sq*

<proof>

fun *wf-logic-node-inputs* :: *IRGraph* \Rightarrow *ID* \Rightarrow *bool* **where**

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

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

$$\begin{aligned} \text{wf-values } g = & (\forall \ n \in \text{ids } g . \\ & (\forall \ v \ m \ p . ([g, m, p] \vdash n \mapsto v) \longrightarrow \end{aligned}$$

```

(is-LogicNode (kind g n)  $\longrightarrow$ 
  wf-bool v  $\wedge$  wf-logic-node-inputs g n)))

```

end

6.7 Dynamic Frames

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

theory IRGraphFrames

imports

Form

begin

fun unchanged :: ID set \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

\langle proof \rangle

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

\langle proof \rangle

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 |

use-inp: nid' \in inputs g n

\implies eval-uses g nid nid' |

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

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

lemma *eval-usages-self*:
assumes *nid* \in *ids g*
shows *nid* \in *eval-usages g nid*
 $\langle \text{proof} \rangle$

lemma *not-in-g-inputs*:
assumes *nid* \notin *ids g*
shows *inputs g nid* = {}
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *child-member-in*:
assumes *nid* \in *ids g*
assumes *List.member (inputs-of (kind g nid)) child*
shows *child* \in *inputs g nid*
 $\langle \text{proof} \rangle$

lemma *inp-in-g*:
assumes *n* \in *inputs g nid*
shows *nid* \in *ids g*
 $\langle \text{proof} \rangle$

lemma *inp-in-g-wf*:
assumes *wf-graph g*
assumes *n* \in *inputs g nid*
shows *n* \in *ids g*
 $\langle \text{proof} \rangle$

lemma *kind-unchanged*:
assumes *nid* \in *ids g1*
assumes *unchanged (eval-usages g1 nid) g1 g2*

shows $kind\ g1\ nid = kind\ g2\ nid$
 $\langle proof \rangle$

lemma *stamp-unchanged*:
assumes $nid \in ids\ g1$
assumes $unchanged\ (eval-usages\ g1\ nid)\ g1\ g2$
shows $stamp\ g1\ nid = stamp\ g2\ nid$
 $\langle proof \rangle$

lemma *child-unchanged*:
assumes $child \in inputs\ g1\ nid$
assumes $unchanged\ (eval-usages\ g1\ nid)\ g1\ g2$
shows $unchanged\ (eval-usages\ g1\ child)\ g1\ g2$
 $\langle proof \rangle$

lemma *eval-usages*:
assumes $us = eval-usages\ g\ nid$
assumes $nid' \in ids\ g$
shows $eval-uses\ g\ nid\ nid' \longleftrightarrow nid' \in us\ (\text{is } ?P \longleftrightarrow ?Q)$
 $\langle proof \rangle$

lemma *inputs-are-uses*:
assumes $nid' \in inputs\ g\ nid$
shows $eval-uses\ g\ nid\ nid'$
 $\langle proof \rangle$

lemma *inputs-are-usages*:
assumes $nid' \in inputs\ g\ nid$
assumes $nid' \in ids\ g$
shows $nid' \in eval-usages\ g\ nid$
 $\langle proof \rangle$

lemma *inputs-of-are-usages*:
assumes $List.member\ (inputs-of\ (kind\ g\ nid))\ nid'$
assumes $nid' \in ids\ g$
shows $nid' \in eval-usages\ g\ nid$
 $\langle proof \rangle$

lemma *usage-includes-inputs*:
assumes $us = eval-usages\ g\ nid$
assumes $ls = inputs\ g\ nid$
assumes $ls \subseteq ids\ g$
shows $ls \subseteq us$
 $\langle proof \rangle$

lemma *elim-inp-set*:
assumes $k = kind\ g\ nid$
assumes $k \neq NoNode$

assumes $child \in set \ (inputs\text{-}of \ k)$
shows $child \in inputs \ g \ nid$
 $\langle proof \rangle$

lemma *encode-in-ids*:
assumes $g \vdash nid \simeq e$
shows $nid \in ids \ g$
 $\langle proof \rangle$

lemma *eval-in-ids*:
assumes $[g, m, p] \vdash nid \mapsto v$
shows $nid \in ids \ g$
 $\langle proof \rangle$

lemma *transitive-kind-same*:
assumes $unchanged \ (eval\text{-}usages \ g1 \ nid) \ g1 \ g2$
shows $\forall \ nid' \in (eval\text{-}usages \ g1 \ nid) . kind \ g1 \ nid' = kind \ g2 \ nid'$
 $\langle proof \rangle$

theorem *stay-same-encoding*:
assumes $nc: unchanged \ (eval\text{-}usages \ g1 \ nid) \ g1 \ g2$
assumes $g1: g1 \vdash nid \simeq e$
assumes $wf: wf\text{-}graph \ g1$
shows $g2 \vdash nid \simeq e$
 $\langle proof \rangle$

theorem *stay-same*:
assumes $nc: unchanged \ (eval\text{-}usages \ g1 \ nid) \ g1 \ g2$
assumes $g1: [g1, m, p] \vdash nid \mapsto v1$
assumes $wf: wf\text{-}graph \ g1$
shows $[g2, m, p] \vdash nid \mapsto v1$
 $\langle proof \rangle$

lemma *add-changed*:
assumes $gup = add\text{-}node \ new \ k \ g$
shows $changeonly \ \{new\} \ g \ gup$
 $\langle proof \rangle$

lemma *disjoint-change*:
assumes $changeonly \ change \ g \ gup$
assumes $nochange = ids \ g - change$
shows $unchanged \ nochange \ g \ gup$
 $\langle proof \rangle$

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  $unchanged\ (eval-usages\ g\ nid)\ g\ gup$ 
 $\langle proof \rangle$ 

lemma  $eval-uses-imp$ :
   $((nid' \in ids\ g \wedge nid = nid')$ 
     $\vee nid' \in inputs\ g\ nid$ 
     $\vee (\exists nid'' . eval-uses\ g\ nid\ nid'' \wedge eval-uses\ g\ nid''\ nid'))$ 
     $\longleftrightarrow eval-uses\ g\ nid\ nid'$ 
   $\langle proof \rangle$ 

lemma  $wf-use-ids$ :
  assumes  $wf-graph\ g$ 
  assumes  $nid \in ids\ g$ 
  assumes  $eval-uses\ g\ nid\ nid'$ 
  shows  $nid' \in ids\ g$ 
   $\langle proof \rangle$ 

lemma  $no-external-use$ :
  assumes  $wf-graph\ g$ 
  assumes  $nid' \notin ids\ g$ 
  assumes  $nid \in ids\ g$ 
  shows  $\neg(eval-uses\ g\ nid\ nid')$ 
   $\langle proof \rangle$ 

end

```

6.8 Tree to Graph Theorems

```

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

```

6.8.1 Extraction and Evaluation of Expression Trees is Deterministic.

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

named-theorems rep

```

lemma  $rep-constant\ [rep]$ :
   $g \vdash n \simeq e \implies$ 

```

$kind\ g\ n = ConstantNode\ c \implies$
 $e = ConstantExpr\ c$
 $\langle proof \rangle$

lemma *rep-parameter* [rep]:
 $g \vdash n \simeq e \implies$
 $kind\ g\ n = ParameterNode\ i \implies$
 $(\exists s. e = ParameterExpr\ i\ s)$
 $\langle proof \rangle$

lemma *rep-conditional* [rep]:
 $g \vdash n \simeq e \implies$
 $kind\ g\ n = ConditionalNode\ c\ t\ f \implies$
 $(\exists ce\ te\ fe. e = ConditionalExpr\ ce\ te\ fe)$
 $\langle proof \rangle$

lemma *rep-abs* [rep]:
 $g \vdash n \simeq e \implies$
 $kind\ g\ n = AbsNode\ x \implies$
 $(\exists xe. e = UnaryExpr\ UnaryAbs\ xe)$
 $\langle proof \rangle$

lemma *rep-not* [rep]:
 $g \vdash n \simeq e \implies$
 $kind\ g\ n = NotNode\ x \implies$
 $(\exists xe. e = UnaryExpr\ UnaryNot\ xe)$
 $\langle proof \rangle$

lemma *rep-negate* [rep]:
 $g \vdash n \simeq e \implies$
 $kind\ g\ n = NegateNode\ x \implies$
 $(\exists xe. e = UnaryExpr\ UnaryNeg\ xe)$
 $\langle proof \rangle$

lemma *rep-logicnegation* [rep]:
 $g \vdash n \simeq e \implies$
 $kind\ g\ n = LogicNegationNode\ x \implies$
 $(\exists xe. e = UnaryExpr\ UnaryLogicNegation\ xe)$
 $\langle proof \rangle$

lemma *rep-add* [rep]:
 $g \vdash n \simeq e \implies$
 $kind\ g\ n = AddNode\ x\ y \implies$
 $(\exists xe\ ye. e = BinaryExpr\ BinAdd\ xe\ ye)$
 $\langle proof \rangle$

lemma *rep-sub* [rep]:
 $g \vdash n \simeq e \implies$
 $kind\ g\ n = SubNode\ x\ y \implies$

$(\exists xe ye. e = \text{BinaryExpr BinSub } xe ye)$
 $\langle \text{proof} \rangle$

lemma *rep-mul* [*rep*]:
 $g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{MulNode } x \ y \implies$
 $(\exists xe ye. e = \text{BinaryExpr BinMul } xe ye)$
 $\langle \text{proof} \rangle$

lemma *rep-and* [*rep*]:
 $g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{AndNode } x \ y \implies$
 $(\exists xe ye. e = \text{BinaryExpr BinAnd } xe ye)$
 $\langle \text{proof} \rangle$

lemma *rep-or* [*rep*]:
 $g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{OrNode } x \ y \implies$
 $(\exists xe ye. e = \text{BinaryExpr BinOr } xe ye)$
 $\langle \text{proof} \rangle$

lemma *rep-xor* [*rep*]:
 $g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{XorNode } x \ y \implies$
 $(\exists xe ye. e = \text{BinaryExpr BinXor } xe ye)$
 $\langle \text{proof} \rangle$

lemma *rep-short-circuit-or* [*rep*]:
 $g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{ShortCircuitOrNode } x \ y \implies$
 $(\exists xe ye. e = \text{BinaryExpr BinShortCircuitOr } xe ye)$
 $\langle \text{proof} \rangle$

lemma *rep-left-shift* [*rep*]:
 $g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{LeftShiftNode } x \ y \implies$
 $(\exists xe ye. e = \text{BinaryExpr BinLeftShift } xe ye)$
 $\langle \text{proof} \rangle$

lemma *rep-right-shift* [*rep*]:
 $g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{RightShiftNode } x \ y \implies$
 $(\exists xe ye. e = \text{BinaryExpr BinRightShift } xe ye)$
 $\langle \text{proof} \rangle$

lemma *rep-unsigned-right-shift* [*rep*]:
 $g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{UnsignedRightShiftNode } x \ y \implies$
 $(\exists xe ye. e = \text{BinaryExpr BinURightShift } xe ye)$

$\langle \text{proof} \rangle$

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

$g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{IntegerBelowNode } x \ y \implies$
 $(\exists xe \ ye. \ e = \text{BinaryExpr } \text{BinIntegerBelow } xe \ ye)$
 $\langle \text{proof} \rangle$

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

$g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{IntegerEqualsNode } x \ y \implies$
 $(\exists xe \ ye. \ e = \text{BinaryExpr } \text{BinIntegerEquals } xe \ ye)$
 $\langle \text{proof} \rangle$

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

$g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{IntegerLessThanNode } x \ y \implies$
 $(\exists xe \ ye. \ e = \text{BinaryExpr } \text{BinIntegerLessThan } xe \ ye)$
 $\langle \text{proof} \rangle$

lemma *rep-narrow* [rep]:

$g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{NarrowNode } ib \ rb \ x \implies$
 $(\exists x. \ e = \text{UnaryExpr } (\text{UnaryNarrow } ib \ rb) \ x)$
 $\langle \text{proof} \rangle$

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

$g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{SignExtendNode } ib \ rb \ x \implies$
 $(\exists x. \ e = \text{UnaryExpr } (\text{UnarySignExtend } ib \ rb) \ x)$
 $\langle \text{proof} \rangle$

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

$g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{ZeroExtendNode } ib \ rb \ x \implies$
 $(\exists x. \ e = \text{UnaryExpr } (\text{UnaryZeroExtend } ib \ rb) \ x)$
 $\langle \text{proof} \rangle$

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

$g \vdash n \simeq e \implies$
 $\text{is-preevaluated } (\text{kind } g \ n) \implies$
 $(\exists s. \ e = \text{LeafExpr } n \ s)$
 $\langle \text{proof} \rangle$

lemma *rep-ref* [rep]:

$g \vdash n \simeq e \implies$
 $\text{kind } g \ n = \text{RefNode } n' \implies$
 $g \vdash n' \simeq e$
 $\langle \text{proof} \rangle$

```

method solve-det uses node =
  (match node in kind - - = node - for node  $\Rightarrow$ 
    <match rep in r: -  $\Rightarrow$  - = node -  $\Rightarrow$  -  $\Rightarrow$ 
      <match IRNode.inject in i: (node - = node -) = -  $\Rightarrow$ 
        <match RepE in e: -  $\Rightarrow$  ( $\bigwedge x$ . - = node x  $\Rightarrow$  -)  $\Rightarrow$  -  $\Rightarrow$ 
          <match IRNode.distinct in d: node -  $\neq$  RefNode -  $\Rightarrow$ 
            <metis i e r d>>>> |
      match node in kind - - = node - - for node  $\Rightarrow$ 
        <match rep in r: -  $\Rightarrow$  - = node - -  $\Rightarrow$  -  $\Rightarrow$ 
          <match IRNode.inject in i: (node - - = node - -) = -  $\Rightarrow$ 
            <match RepE in e: -  $\Rightarrow$  ( $\bigwedge x y$ . - = node x y  $\Rightarrow$  -)  $\Rightarrow$  -  $\Rightarrow$ 
              <match IRNode.distinct in d: node - -  $\neq$  RefNode -  $\Rightarrow$ 
                <metis i e r d>>>> |
          match node in kind - - = node - - - for node  $\Rightarrow$ 
            <match rep in r: -  $\Rightarrow$  - = node - - -  $\Rightarrow$  -  $\Rightarrow$ 
              <match IRNode.inject in i: (node - - - = node - - -) = -  $\Rightarrow$ 
                <match RepE in e: -  $\Rightarrow$  ( $\bigwedge x y z$ . - = node x y z  $\Rightarrow$  -)  $\Rightarrow$  -  $\Rightarrow$ 
                  <match IRNode.distinct in d: node - - -  $\neq$  RefNode -  $\Rightarrow$ 
                    <metis i e r d>>>> |
              match node in kind - - = node - - - for node  $\Rightarrow$ 
                <match rep in r: -  $\Rightarrow$  - = node - - -  $\Rightarrow$  -  $\Rightarrow$ 
                  <match IRNode.inject in i: (node - - - = node - - -) = -  $\Rightarrow$ 
                    <match RepE in e: -  $\Rightarrow$  ( $\bigwedge x$ . - = node - - x  $\Rightarrow$  -)  $\Rightarrow$  -  $\Rightarrow$ 
                      <match IRNode.distinct in d: node - - -  $\neq$  RefNode -  $\Rightarrow$ 
                        <metis i e r d>>>>)
  )

```

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

lemma repDet:
shows $(g \vdash n \simeq e_1) \Rightarrow (g \vdash n \simeq e_2) \Rightarrow e_1 = e_2$
 <proof>

lemma repAllDet:
 $g \vdash xs \simeq_L e1 \Rightarrow$
 $g \vdash xs \simeq_L e2 \Rightarrow$
 $e1 = e2$
 <proof>

lemma encodeEvalDet:
 $[g, m, p] \vdash e \mapsto v1 \Rightarrow$
 $[g, m, p] \vdash e \mapsto v2 \Rightarrow$
 $v1 = v2$
 <proof>

lemma graphDet: $([g, m, p] \vdash n \mapsto v_1) \wedge ([g, m, p] \vdash n \mapsto v_2) \Rightarrow v_1 = v_2$
 <proof>

6.8.2 Monotonicity of Graph Refinement

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

lemma *mono-abs*:

assumes $\text{kind } g1 \ n = \text{AbsNode } x \wedge \text{kind } g2 \ n = \text{AbsNode } x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
 $\langle \text{proof} \rangle$

lemma *mono-not*:

assumes $\text{kind } g1 \ n = \text{NotNode } x \wedge \text{kind } g2 \ n = \text{NotNode } x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
 $\langle \text{proof} \rangle$

lemma *mono-negate*:

assumes $\text{kind } g1 \ n = \text{NegateNode } x \wedge \text{kind } g2 \ n = \text{NegateNode } x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
 $\langle \text{proof} \rangle$

lemma *mono-logic-negation*:

assumes $\text{kind } g1 \ n = \text{LogicNegationNode } x \wedge \text{kind } g2 \ n = \text{LogicNegationNode } x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
 $\langle \text{proof} \rangle$

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 $e1 \geq e2$
 $\langle \text{proof} \rangle$

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 $e1 \geq e2$
 $\langle proof \rangle$

lemma *mono-zero-extend*:

assumes $kind\ g1\ n = ZeroExtendNode\ ib\ rb\ x \wedge kind\ g2\ n = ZeroExtendNode\ ib\ rb\ x$
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 $e1 \geq e2$
 $\langle proof \rangle$

lemma *mono-conditional-graph*:

assumes $kind\ g1\ n = ConditionalNode\ c\ t\ f \wedge kind\ g2\ n = ConditionalNode\ c\ t\ f$
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 $e1 \geq e2$
 $\langle proof \rangle$

lemma *mono-add*:

assumes $kind\ g1\ n = AddNode\ x\ y \wedge kind\ g2\ n = AddNode\ x\ y$
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 $e1 \geq e2$
 $\langle proof \rangle$

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 $e1 \geq e2$
 $\langle proof \rangle$

lemma *term-graph-evaluation*:

$(g \vdash n \sqsubseteq e) \implies (\forall\ m\ p\ v . ([m,p] \vdash e \mapsto v) \longrightarrow ([g,m,p] \vdash n \mapsto v))$
 $\langle proof \rangle$

lemma *encodes-contains*:

$g \vdash n \simeq e \implies$
 $kind\ g\ n \neq NoNode$

$\langle \text{proof} \rangle$

lemma *no-encoding*:

assumes $n \notin \text{ids } g$
shows $\neg(g \vdash n \simeq e)$
 $\langle \text{proof} \rangle$

lemma *not-excluded-keep-type*:

assumes $n \in \text{ids } g1$
assumes $n \notin \text{excluded}$
assumes $(\text{excluded} \trianglelefteq \text{as-set } g1) \subseteq \text{as-set } g2$
shows $\text{kind } g1 \ n = \text{kind } g2 \ n \wedge \text{stamp } g1 \ n = \text{stamp } g2 \ n$
 $\langle \text{proof} \rangle$

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

$(\text{match } \text{IRNode.inject} \text{ in } i: (\text{node } - = \text{node } -) = - \Rightarrow$
 $\langle \text{metis } i \rangle)$

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

$(\text{match } \text{IRNode.inject} \text{ in } i: (\text{node } - - = \text{node } - -) = - \Rightarrow$
 $\langle \text{metis } i \rangle)$

method *metis-node-eq-ternary* **for** $\text{node} :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{IRNode} =$

$(\text{match } \text{IRNode.inject} \text{ in } i: (\text{node } - - - = \text{node } - - -) = - \Rightarrow$
 $\langle \text{metis } i \rangle)$

6.8.3 Lift Data-flow Tree Refinement to Graph Refinement

theorem *graph-semantics-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 $\text{graph-refinement } g1 \ g2$
 $\langle \text{proof} \rangle$

lemma *graph-semantics-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 $\text{graph-refinement } g_1 \ g_2$
 $\langle \text{proof} \rangle$

lemma *tree-to-graph-rewriting*:

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

$\langle \text{proof} \rangle$
declare $[[\text{simp-trace}]]$
lemma *equal-refines*:
 fixes $e1\ e2 :: \text{IRExpr}$
 assumes $e1 = e2$
 shows $e1 \geq e2$
 $\langle \text{proof} \rangle$
declare $[[\text{simp-trace}=\text{false}]]$

lemma *eval-contains-id* $[\text{simp}]$: $g1 \vdash n \simeq e \implies n \in \text{ids } g1$
 $\langle \text{proof} \rangle$

lemma *subset-kind* $[\text{simp}]$: $\text{as-set } g1 \subseteq \text{as-set } g2 \implies g1 \vdash n \simeq e \implies \text{kind } g1\ n = \text{kind } g2\ n$
 $\langle \text{proof} \rangle$

lemma *subset-stamp* $[\text{simp}]$: $\text{as-set } g1 \subseteq \text{as-set } g2 \implies g1 \vdash n \simeq e \implies \text{stamp } g1\ n = \text{stamp } g2\ n$
 $\langle \text{proof} \rangle$

method *solve-subset-eval* **uses** $\text{as-set eval} =$
 $(\text{metis eval as-set subset-kind subset-stamp} \mid$
 $\text{metis eval as-set subset-kind})$

lemma *subset-implies-evals*:
 assumes $\text{as-set } g1 \subseteq \text{as-set } g2$
 assumes $(g1 \vdash n \simeq e)$
 shows $(g2 \vdash n \simeq e)$
 $\langle \text{proof} \rangle$

lemma *subset-refines*:
 assumes $\text{as-set } g1 \subseteq \text{as-set } g2$
 shows $\text{graph-refinement } g1\ g2$
 $\langle \text{proof} \rangle$

lemma *graph-construction*:
 $e1 \geq e2$
 $\wedge \text{as-set } g1 \subseteq \text{as-set } g2$
 $\wedge (g2 \vdash n \simeq e2)$
 $\implies (g2 \vdash n \sqsubseteq e1) \wedge \text{graph-refinement } g1\ g2$
 $\langle \text{proof} \rangle$

6.8.4 Term Graph Reconstruction

lemma *find-exists-kind*:

assumes $\text{find-node-and-stamp } g \text{ (node, s) = Some nid}$
shows $\text{kind } g \text{ nid} = \text{node}$
 $\langle \text{proof} \rangle$

lemma *find-exists-stamp*:
assumes $\text{find-node-and-stamp } g \text{ (node, s) = Some nid}$
shows $\text{stamp } g \text{ nid} = s$
 $\langle \text{proof} \rangle$

lemma *find-new-kind*:
assumes $g' = \text{add-node nid (node, s) } g$
assumes $\text{node} \neq \text{NoNode}$
shows $\text{kind } g' \text{ nid} = \text{node}$
 $\langle \text{proof} \rangle$

lemma *find-new-stamp*:
assumes $g' = \text{add-node nid (node, s) } g$
assumes $\text{node} \neq \text{NoNode}$
shows $\text{stamp } g' \text{ nid} = s$
 $\langle \text{proof} \rangle$

lemma *sorted-bottom*:
assumes $\text{finite } xs$
assumes $x \in xs$
shows $x \leq \text{last}(\text{sorted-list-of-set}(xs::\text{nat set}))$
 $\langle \text{proof} \rangle$

lemma *fresh*: $\text{finite } xs \implies \text{last}(\text{sorted-list-of-set}(xs::\text{nat set})) + 1 \notin xs$
 $\langle \text{proof} \rangle$

lemma *fresh-ids*:
assumes $n = \text{get-fresh-id } g$
shows $n \notin \text{ids } g$
 $\langle \text{proof} \rangle$

lemma *graph-unchanged-rep-unchanged*:
assumes $\forall n \in \text{ids } g. \text{kind } g \text{ } n = \text{kind } g' \text{ } n$
assumes $\forall n \in \text{ids } g. \text{stamp } g \text{ } n = \text{stamp } g' \text{ } n$
shows $(g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
 $\langle \text{proof} \rangle$

lemma *fresh-node-subset*:
assumes $n \notin \text{ids } g$
assumes $g' = \text{add-node } n \text{ (k, s) } g$
shows $\text{as-set } g \subseteq \text{as-set } g'$
 $\langle \text{proof} \rangle$

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

shows $as\text{-}set\ g \subseteq as\text{-}set\ g'$
 $\langle proof \rangle$

lemma *fresh-node-preserves-other-nodes*:
assumes $n' = get\text{-}fresh\text{-}id\ g$
assumes $g' = add\text{-}node\ n'\ (k, s)\ g$
shows $\forall\ n \in ids\ g. (g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
 $\langle proof \rangle$

lemma *found-node-preserves-other-nodes*:
assumes $find\text{-}node\text{-}and\text{-}stamp\ g\ (k, s) = Some\ n$
shows $\forall\ n \in ids\ g. (g \vdash n \simeq e) \longleftrightarrow (g \vdash n \simeq e)$
 $\langle proof \rangle$

lemma *unrep-ids-subset[simp]*:
assumes $g \oplus e \rightsquigarrow (g', n)$
shows $ids\ g \subseteq ids\ g'$
 $\langle proof \rangle$

lemma *unrep-unchanged*:
assumes $g \oplus e \rightsquigarrow (g', n)$
shows $\forall\ n \in ids\ g. \forall\ e. (g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
 $\langle proof \rangle$

theorem *term-graph-reconstruction*:
 $g \oplus e \rightsquigarrow (g', n) \implies (g' \vdash n \simeq e) \wedge as\text{-}set\ g \subseteq as\text{-}set\ g'$
 $\langle proof \rangle$

lemma *ref-refinement*:
assumes $g \vdash n \simeq e_1$
assumes $kind\ g\ n' = RefNode\ n$
shows $g \vdash n' \trianglelefteq e_1$
 $\langle proof \rangle$

lemma *unrep-refines*:
assumes $g \oplus e \rightsquigarrow (g', n)$
shows $graph\text{-}refinement\ g\ g'$
 $\langle proof \rangle$

lemma *add-new-node-refines*:
assumes $n \notin ids\ g$
assumes $g' = add\text{-}node\ n\ (k, s)\ g$
shows $graph\text{-}refinement\ g\ g'$
 $\langle proof \rangle$

lemma *add-node-as-set*:
assumes $g' = add\text{-}node\ n\ (k, s)\ g$
shows $(\{n\} \trianglelefteq as\text{-}set\ g) \subseteq as\text{-}set\ g'$
 $\langle proof \rangle$

theorem *refined-insert*:

assumes $e_1 \geq e_2$
assumes $g_1 \oplus e_2 \rightsquigarrow (g_2, n')$
shows $(g_2 \vdash n' \leq e_1) \wedge \text{graph-refinement } g_1 \ g_2$
 $\langle \text{proof} \rangle$

lemma *ids-finite*: $\text{finite } (\text{ids } g)$

$\langle \text{proof} \rangle$

lemma *unwrap-sorted*: $\text{set } (\text{sorted-list-of-set } (\text{ids } g)) = \text{ids } g$

$\langle \text{proof} \rangle$

lemma *find-none*:

assumes $\text{find-node-and-stamp } g \ (k, s) = \text{None}$
shows $\forall n \in \text{ids } g. \text{kind } g \ n \neq k \vee \text{stamp } g \ n \neq s$
 $\langle \text{proof} \rangle$

method *ref-represents* **uses** $\text{node} =$

$(\text{metis IRNode.distinct}(2755) \text{RefNode.dual-order.refl find-new-kind fresh-node-subset}$
 $\text{node subset-implies-evals})$

6.8.5 Data-flow Tree to Subgraph Preserves Maximal Sharing

lemma *same-kind-stamp-encodes-equal*:

assumes $\text{kind } g \ n = \text{kind } g \ n'$
assumes $\text{stamp } g \ n = \text{stamp } g \ n'$
assumes $\neg(\text{is-preevaluated } (\text{kind } g \ n))$
shows $\forall e. (g \vdash n \simeq e) \longrightarrow (g \vdash n' \simeq e)$
 $\langle \text{proof} \rangle$

lemma *new-node-not-present*:

assumes $\text{find-node-and-stamp } g \ (\text{node}, s) = \text{None}$
assumes $n = \text{get-fresh-id } g$
assumes $g' = \text{add-node } n \ (\text{node}, s) \ g$
shows $\forall n' \in \text{true-ids } g. (\forall e. ((g \vdash n \simeq e) \wedge (g \vdash n' \simeq e)) \longrightarrow n = n')$

$\langle \text{proof} \rangle$

lemma *true-ids-def*:

$\text{true-ids } g = \{n \in \text{ids } g. \neg(\text{is-RefNode } (\text{kind } g \ n)) \wedge ((\text{kind } g \ n) \neq \text{NoNode})\}$

$\langle \text{proof} \rangle$

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

assumes $k \neq \text{NoNode}$

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

shows $g' = \text{Abs-IRGraph } ((\text{Rep-IRGraph } g)(nid \mapsto (k, s)))$

$\langle \text{proof} \rangle$

lemma *ids-add-update-v1*:

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

assumes $k \neq \text{NoNode}$

shows $\text{dom } (\text{Rep-IRGraph } g') = \text{dom } (\text{Rep-IRGraph } g) \cup \{nid\}$

$\langle \text{proof} \rangle$

lemma *ids-add-update-v2*:

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

assumes $k \neq \text{NoNode}$

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

$\langle \text{proof} \rangle$

lemma *add-node-ids-subset*:

assumes $n \in \text{ids } g$

assumes $g' = \text{add-node } n \ \text{node } g$

shows $\text{ids } g' = \text{ids } g \cup \{n\}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *add-node-set-eq*:

assumes $k \neq \text{NoNode}$

assumes $n \notin \text{ids } g$

shows $\text{as-set } (\text{add-node } n \ (k, s) \ g) = \text{as-set } g \cup \{(n, (k, s))\}$

$\langle \text{proof} \rangle$

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

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

assumes $n \notin \text{ids } g$

shows $(\{n\} \trianglelefteq \text{as-set } g') = \text{as-set } g$

$\langle \text{proof} \rangle$

lemma *true-ids*:

$true-ids\ g = ids\ g - \{n \in ids\ g. is-RefNode\ (kind\ g\ n)\}$
 $\langle proof \rangle$

lemma *as-set-ids*:
assumes $as-set\ g = as-set\ g'$
shows $ids\ g = ids\ g'$
 $\langle proof \rangle$

lemma *ids-add-update*:
assumes $k \neq NoNode$
assumes $n \notin ids\ g$
assumes $g' = add-node\ n\ (k, s)\ g$
shows $ids\ g' = ids\ g \cup \{n\}$
 $\langle proof \rangle$

lemma *true-ids-add-update*:
assumes $k \neq NoNode$
assumes $n \notin ids\ g$
assumes $g' = add-node\ n\ (k, s)\ g$
assumes $\neg(is-RefNode\ k)$
shows $true-ids\ g' = true-ids\ g \cup \{n\}$
 $\langle proof \rangle$

lemma *new-def*:
assumes $(new \sqsubseteq as-set\ g') = as-set\ g$
shows $n \in ids\ g \longrightarrow n \notin new$
 $\langle proof \rangle$

lemma *add-preserves-rep*:
assumes $unchanged: (new \sqsubseteq as-set\ g') = as-set\ g$
assumes $closed: wf-closed\ g$
assumes $existed: n \in ids\ g$
assumes $g' \vdash n \simeq e$
shows $g \vdash n \simeq e$
 $\langle proof \rangle$

lemma *not-in-no-rep*:
 $n \notin ids\ g \implies \forall e. \neg(g \vdash n \simeq e)$
 $\langle proof \rangle$

lemma *unary-inputs*:
assumes $kind\ g\ n = unary-node\ op\ x$
shows $inputs\ g\ n = \{x\}$
 $\langle proof \rangle$

lemma *unary-succ*:

assumes $\text{kind } g \ n = \text{unary-node } op \ x$

shows $\text{succ } g \ n = \{\}$

$\langle \text{proof} \rangle$

lemma *binary-inputs*:

assumes $\text{kind } g \ n = \text{bin-node } op \ x \ y$

shows $\text{inputs } g \ n = \{x, y\}$

$\langle \text{proof} \rangle$

lemma *binary-succ*:

assumes $\text{kind } g \ n = \text{bin-node } op \ x \ y$

shows $\text{succ } g \ n = \{\}$

$\langle \text{proof} \rangle$

lemma *unrep-contains*:

assumes $g \oplus e \rightsquigarrow (g', n)$

shows $n \in \text{ids } g'$

$\langle \text{proof} \rangle$

lemma *unrep-preserves-contains*:

assumes $n \in \text{ids } g$

assumes $g \oplus e \rightsquigarrow (g', n')$

shows $n \in \text{ids } g'$

$\langle \text{proof} \rangle$

lemma *unrep-preserves-closure*:

assumes $\text{wf-closed } g$

assumes $g \oplus e \rightsquigarrow (g', n)$

shows $\text{wf-closed } g'$

$\langle \text{proof} \rangle$

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

definition *constant-value* **where**

$\text{constant-value} = (\text{IntVal } 32 \ 0)$

definition *bad-graph* **where**

$\text{bad-graph} = \text{irgraph } [$

$(0, \text{AbsNode } 1, \text{constantAsStamp } \text{constant-value}),$

$(1, \text{RefNode } 2, \text{constantAsStamp } \text{constant-value}),$

$(2, \text{ConstantNode } \text{constant-value}, \text{constantAsStamp } \text{constant-value})$

$]$

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
 $\times$  MapState  $\times$  FieldRefHeap)  $\Rightarrow$  bool
  ( $\neg$ ,  $- \vdash - \rightarrow -$  55) for g p where

```

SequentialNode:

```

[[is-sequential-node (kind g nid);
  nid' = (successors-of (kind g nid))!0]]
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \mid$ 

```

IfNode:

```

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

```

EndNodes:

```

[[is-AbstractEndNode (kind g nid);
  merge = any-usage g nid;
  is-AbstractMergeNode (kind g merge);

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

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

```

NewInstanceNode:

```

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

```

LoadFieldNode:

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

SignedDivNode:

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

SignedRemNode:

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

StaticLoadFieldNode:

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

StoreFieldNode:

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

StaticStoreFieldNode:

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

$$\begin{aligned}
& h' = h\text{-store-field } f \text{ None val } h; \\
& m' = m(nid := val) \\
\implies & g, p \vdash (nid, m, h) \rightarrow (nid', m', h')
\end{aligned}$$

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

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:

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

InvokeNodeStep:

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

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

ReturnNode:

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

$$\begin{aligned}
& g \vdash expr \simeq e; \\
& [m, p] \vdash e \mapsto v;
\end{aligned}$$

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

ReturnNodeVoid:

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

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

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

UnwindNode:

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

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

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

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

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

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

7.4 Big-step Execution

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

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

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

inductive *exec* :: *Program*

$\Rightarrow (\text{IRGraph} \times \text{ID} \times \text{MapState} \times \text{Params}) \text{ list} \times \text{FieldRefHeap}$

$\Rightarrow \text{Trace}$

$\Rightarrow (\text{IRGraph} \times \text{ID} \times \text{MapState} \times \text{Params}) \text{ list} \times \text{FieldRefHeap}$

$\Rightarrow \text{Trace}$

$\Rightarrow \text{bool}$

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

for *P*

where

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

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

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

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

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

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

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

$\implies \text{exec } P (((g, \text{nid}, m, p) \# xs), h) \text{ l } (((g', \text{nid}', m', p') \# ys), h') \text{ l'}$

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$ as *Exec*) *exec* $\langle \text{proof} \rangle$

inductive *exec-debug* :: *Program*

$\Rightarrow (\text{IRGraph} \times \text{ID} \times \text{MapState} \times \text{Params}) \text{ list} \times \text{FieldRefHeap}$

```

    ⇒ nat
    ⇒ (IRGraph × ID × MapState × Params) list × FieldRefHeap
    ⇒ bool
  (⊢ → * -)
  where
    ⌊n > 0;
    p ⊢ s → s';
    exec-debug p s' (n - 1) s'⌋
    ⇒ exec-debug p s n s'' |

    ⌊n = 0⌋
    ⇒ exec-debug p s n s
  code-pred (modes: i ⇒ i ⇒ i ⇒ o ⇒ bool) exec-debug ⟨proof⟩

```

7.4.1 Heap Testing

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

values {(prod.fst(prod.snd (prod.snd (hd (prod.fst res))))) 0
 | res. (λx. Some eg2-sq) ⊢ [(eg2-sq, 0, new-map-state, p3), (eg2-sq, 0, new-map-state, p3)],
 new-heap) →*2* res}

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

definition *eg3-sq* :: IRGraph **where**
eg3-sq = irgraph [
 (0, StartNode None 4, VoidStamp),
 (1, ParameterNode 0, default-stamp),
 (3, MulNode 1 1, default-stamp),
 (4, StoreFieldNode 4 field-sq 3 None None 5, VoidStamp),
 (5, ReturnNode (Some 3) None, default-stamp)
]

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

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

```

    (6, ReturnNode (Some 3) None, default-stamp)
  ]

values {h-load-field field-sq (Some 0) (prod.snd res) | res.
    (λx. Some eg4-sq) ⊢ [(eg4-sq, 0, new-map-state, p3), (eg4-sq, 0,
new-map-state, p3)], new-heap) →*3* res}

end

```

7.5 Control-flow Semantics Theorems

```

theory IRStepThms
imports
  IRStepObj
  TreeToGraphThms
begin

```

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

7.5.1 Control-flow Step is Deterministic

```

theorem stepDet:
  (g, p ⊢ (nid,m,h) → next) ⇒
  (∀ next'. ((g, p ⊢ (nid,m,h) → next') ⟶ next = next'))
  <proof>

```

```

lemma stepRefNode:
  ⟦kind g nid = RefNode nid'⟧ ⟹ g, p ⊢ (nid,m,h) → (nid',m,h)
  <proof>

```

```

lemma IfNodeStepCases:
  assumes kind g nid = IfNode cond tb fb
  assumes g ⊢ cond ≃ condE
  assumes [m, p] ⊢ condE ↦ v
  assumes g, p ⊢ (nid, m, h) → (nid', m, h)
  shows nid' ∈ {tb, fb}
  <proof>

```

```

lemma IfNodeSeq:
  shows kind g nid = IfNode cond tb fb ⟶ ¬(is-sequential-node (kind g nid))
  <proof>

```

```

lemma IfNodeCond:
  assumes kind g nid = IfNode cond tb fb
  assumes g, p ⊢ (nid, m, h) → (nid', m, h)
  shows ∃ condE v. (g ⊢ cond ≃ condE) ∧ ([m, p] ⊢ condE ↦ v)

```

```

    <proof>

lemma step-in-ids:
  assumes  $g, p \vdash (nid, m, h) \rightarrow (nid', m', h')$ 
  shows  $nid \in ids\ g$ 
  <proof>

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

```

A strong bisimulation between no-op transitions

```

inductive strong-noop-bisimilar ::  $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$ 
  (- | - ~ -) for  $nid$  where
     $\llbracket \forall P'. (g, p \vdash (nid, m, h) \rightarrow P') \longrightarrow (\exists Q'. (g', p \vdash (nid, m, h) \rightarrow Q') \wedge P' =$ 
     $Q');$ 
     $\forall Q'. (g', p \vdash (nid, m, h) \rightarrow Q') \longrightarrow (\exists P'. (g, p \vdash (nid, m, h) \rightarrow P') \wedge P' =$ 
     $Q') \rrbracket$ 
     $\implies nid \mid g \sim g'$ 

```

```

lemma lockstep-strong-bisimulation:
  assumes  $g' = replace\_node\ nid\ node\ g$ 
  assumes  $g, p \vdash (nid, m, h) \rightarrow (nid', m, h)$ 
  assumes  $g', p \vdash (nid, m, h) \rightarrow (nid', m, h)$ 
  shows  $nid \mid g \sim g'$ 
  <proof>

```

```

lemma no-step-bisimulation:
  assumes  $\forall m\ p\ h\ nid'\ m'\ h'. \neg(g, p \vdash (nid, m, h) \rightarrow (nid', m', h'))$ 
  assumes  $\forall m\ p\ h\ nid'\ m'\ h'. \neg(g', p \vdash (nid, m, h) \rightarrow (nid', m', h'))$ 
  shows  $nid \mid g \sim g'$ 
  <proof>

```

```

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'* = *replace-usages* *nid* *nid'* *g*

shows *kind* *g'* *nid* = *RefNode* *nid'*

<proof>

lemma *replace-usages-changeonly*:

assumes *nid* \in *ids* *g*

assumes *g'* = *replace-usages* *nid* *nid'* *g*

shows *changeonly* {*nid*} *g* *g'*

<proof>

lemma *replace-usages-unchanged*:

assumes *nid* \in *ids* *g*

assumes *g'* = *replace-usages* *nid* *nid'* *g*

shows *unchanged* (*ids* *g* - {*nid*}) *g* *g'*

<proof>

fun *nextNid* :: *IRGraph* \Rightarrow *ID* **where**

nextNid *g* = (*Max* (*ids* *g*)) + 1

lemma *max-plus-one*:

fixes *c* :: *ID* *set*

shows $\llbracket \text{finite } c; c \neq \{\} \rrbracket \implies (\text{Max } c) + 1 \notin c$

<proof>

lemma *ids-finite*:

finite (*ids* *g*)

<proof>

lemma *nextNidNotIn*:

ids *g* $\neq \{\} \longrightarrow \text{nextNid } g \notin \text{ids } g$

<proof>

fun *bool-to-val-width1* :: *bool* \Rightarrow *Value* **where**

bool-to-val-width1 *True* = (*IntVal* 1 1) |

bool-to-val-width1 *False* = (*IntVal* 1 0)

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-width1* *val*)), *constantA-*
sStamp (*bool-to-val-width1* *val*)) *g*) |
constantCondition *cond nid* - *g* = *g*

lemma *constantConditionTrue*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes *g'* = *constantCondition True ifcond* (*kind g ifcond*) *g*
shows *g', p* \vdash (*ifcond*, *m*, *h*) \rightarrow (*t*, *m*, *h*)
 \langle *proof* \rangle

lemma *constantConditionFalse*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes *g'* = *constantCondition False ifcond* (*kind g ifcond*) *g*
shows *g', p* \vdash (*ifcond*, *m*, *h*) \rightarrow (*f*, *m*, *h*)
 \langle *proof* \rangle

lemma *diff-forall*:
assumes $\forall n \in \text{ids } g - \{nid\}. \text{cond } n$
shows $\forall n. n \in \text{ids } g \wedge n \notin \{nid\} \longrightarrow \text{cond } n$
 \langle *proof* \rangle

lemma *replace-node-changeonly*:
assumes *g'* = *replace-node nid node g*
shows *changeonly* {*nid*} *g g'*
 \langle *proof* \rangle

lemma *add-node-changeonly*:
assumes *g'* = *add-node nid node g*
shows *changeonly* {*nid*} *g g'*
 \langle *proof* \rangle

lemma *constantConditionNoEffect*:
assumes $\neg(\text{is-IfNode } (\text{kind } g \text{ nid}))$
shows *g* = *constantCondition b nid* (*kind g nid*) *g*
 \langle *proof* \rangle

lemma *constantConditionIfNode*:
assumes *kind g nid* = *IfNode cond t f*
shows *constantCondition val nid* (*kind g nid*) *g* =
replace-node *nid* (*IfNode* (*nextNid* *g*) *t f*, *stamp* *g* *nid*)
(*add-node* (*nextNid* *g*) ((*ConstantNode* (*bool-to-val-width1* *val*)), *constantA-*
sStamp (*bool-to-val-width1* *val*)) *g*)
 \langle *proof* \rangle

lemma *constantCondition-changeonly*:
assumes *nid* \in *ids g*

assumes $g' = \text{constantCondition } b \text{ nid } (\text{kind } g \text{ nid}) \ g$
shows $\text{changeonly } \{ \text{nid} \} \ g \ g'$
 $\langle \text{proof} \rangle$

lemma *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 $\exists \text{ nid}' . (g \ m \ p \ h \vdash \text{ifcond} \rightsquigarrow \text{nid}') \longleftrightarrow (g' \ m \ p \ h \vdash \text{ifcond} \rightsquigarrow \text{nid}')$
 $\langle \text{proof} \rangle$

lemma *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 $\exists \text{ nid}' . (g \ m \ p \ h \vdash \text{ifcond} \rightsquigarrow \text{nid}') \longleftrightarrow (g' \ m \ p \ h \vdash \text{ifcond} \rightsquigarrow \text{nid}')$
 $\langle \text{proof} \rangle$

end

8.3 Stuttering

theory *Stuttering*
imports
Semantics.IRStepThms
begin

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

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

Transitive:
 $\llbracket g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}'', m, h);$
 $g \ m \ p \ h \vdash \text{nid}'' \rightsquigarrow \text{nid}' \rrbracket$
 $\implies g \ m \ p \ h \vdash \text{nid} \rightsquigarrow \text{nid}'$

lemma *stuttering-successor*:
assumes $(g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m, h))$
shows $\{P'. (g \ m \ p \ h \vdash \text{nid} \rightsquigarrow P')\} = \{\text{nid}'\} \cup \{\text{nid}'' . (g \ m \ p \ h \vdash \text{nid}' \rightsquigarrow \text{nid}'')\}$
 $\langle \text{proof} \rangle$

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
   $\langle proof \rangle$ 

```

```

lemma unwrap-signed-take-bit:
  fixes v :: int64
  assumes 0 < b  $\wedge$  b  $\leq$  64
  assumes signed-take-bit (b - 1) v = v
  shows signed-take-bit 63 (Word.rep (signed-take-bit (b - Suc 0) v)) = sint v
   $\langle proof \rangle$ 

```

```

lemma unrestricted-new-int-always-valid [simp]:
  assumes 0 < b  $\wedge$  b  $\leq$  64
  shows valid-value (new-int b v) (unrestricted-stamp (IntegerStamp b lo hi))
   $\langle proof \rangle$ 

```

```

lemma unary-undef: val =.UndefVal  $\implies$  unary-eval op val =.UndefVal
   $\langle proof \rangle$ 

```

```

lemma unary-obj: val = ObjRef x  $\implies$  unary-eval op val =.UndefVal
   $\langle proof \rangle$ 

```

```

lemma unrestricted-stamp-valid:
  assumes s = unrestricted-stamp (IntegerStamp b lo hi)
  assumes 0 < b  $\wedge$  b  $\leq$  64
  shows valid-stamp s
   $\langle proof \rangle$ 

```

```

lemma unrestricted-stamp-valid-value [simp]:
  assumes 1: result = IntVal b ival
  assumes take-bit b ival = ival
  assumes 0 < b  $\wedge$  b  $\leq$  64
  shows valid-value result (unrestricted-stamp (IntegerStamp b lo hi))
   $\langle proof \rangle$ 

```

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* \wedge
valid-stamp (*IntegerStamp* *b lo hi*) \wedge
take-bit *b val* = *val* \wedge
 $lo \leq \text{int-signed-value } b \text{ val} \wedge \text{int-signed-value } b \text{ val} \leq hi$
 $\langle \text{proof} \rangle$

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* \wedge
valid-stamp (*IntegerStamp* *b lo hi*) \wedge
take-bit *b ival* = *ival* \wedge
 $lo \leq \text{int-signed-value } b \text{ ival} \wedge \text{int-signed-value } b \text{ ival} \leq hi$
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

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*)
 $\langle \text{proof} \rangle$

A valid int means a valid non-empty stamp.

lemma *valid-int-not-empty:*

assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows $lo \leq hi$
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

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

assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows *and* *val* (*not* (*mask* *bits*)) = 0
 $\langle \text{proof} \rangle$

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}
 $\langle \text{proof} \rangle$

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* < $2^{(\text{bits} - 1)}$
 $\langle \text{proof} \rangle$

lemma *valid-int-signed-lower-bound*:
assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows $-(2^{(\text{bits} - 1)}) \leq \text{int-signed-value bits val}$
 $\langle \text{proof} \rangle$

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* $\leq \text{snd}(\text{bit-bounds bits})$
 $\langle \text{proof} \rangle$

lemma *valid-int-signed-lower-bit-bound*:
assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows *fst* (*bit-bounds bits*) $\leq \text{int-signed-value bits val}$
 $\langle \text{proof} \rangle$

Valid values satisfy their stamp bounds.

lemma *valid-int-signed-range*:
assumes *valid-value* (*IntVal* *b val*) (*IntegerStamp* *bits lo hi*)
shows $\text{lo} \leq \text{int-signed-value bits val} \wedge \text{int-signed-value bits val} \leq \text{hi}$
 $\langle \text{proof} \rangle$

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 *result* = *unary-eval op val*
assumes *op*: *op* \in *normal-unary*
assumes *result* $\neq \text{UndefVal}$
assumes *valid-value val* (*stamp-expr expr*)
shows *valid-value result* (*stamp-expr* (*UnaryExpr op expr*))
 $\langle \text{proof} \rangle$

lemma *narrow-widen-output-bits*:
assumes *unary-eval op val* \neq *UndefVal*
assumes *op* \notin *normal-unary*
shows $0 < (\text{ir-resultBits } op) \wedge (\text{ir-resultBits } op) \leq 64$
 $\langle \text{proof} \rangle$

lemma *eval-widen-narrow-unary-implies-valid-value*:
assumes $[m, p] \vdash \text{expr} \mapsto \text{val}$
assumes *result* = *unary-eval op val*
assumes *op*: *op* \notin *normal-unary*
assumes *result* \neq *UndefVal*
assumes *valid-value val* (*stamp-expr expr*)
shows *valid-value result* (*stamp-expr (UnaryExpr op expr)*)
 $\langle \text{proof} \rangle$

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

8.4.3 Support Lemmas for Binary Operators

lemma *binary-undef*: $v1 = \text{UndefVal} \vee v2 = \text{UndefVal} \implies \text{bin-eval } op \ v1 \ v2 = \text{UndefVal}$
 $\langle \text{proof} \rangle$

lemma *binary-obj*: $v1 = \text{ObjRef } x \vee v2 = \text{ObjRef } y \implies \text{bin-eval } op \ v1 \ v2 = \text{UndefVal}$
 $\langle \text{proof} \rangle$

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 $\exists v. \text{result} = \text{new-int } b1 \ v$
 $\langle \text{proof} \rangle$

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 $\exists v. \text{result} = \text{new-int } 32 \ v$
 $\langle \text{proof} \rangle$

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 $\exists v. result = new_int\ b1\ v$
 $\langle proof \rangle$

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 $b1 = b2$
 $\langle proof \rangle$

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 $valid_value\ result\ (stamp_expr\ (BinaryExpr\ op\ expr1\ expr2))$
 $\langle proof \rangle$

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 $valid_stamp\ (meet\ stamp1\ stamp2)$
 $\langle proof \rangle$

lemma *stamp-meet-is-valid-stamp*:
assumes $1: valid_stamp\ stamp1$
assumes $2: valid_stamp\ stamp2$
shows $valid_stamp\ (meet\ stamp1\ stamp2)$
 $\langle proof \rangle$

lemma *stamp-meet-commutes*: $meet\ stamp1\ stamp2 = meet\ stamp2\ stamp1$
 $\langle proof \rangle$

lemma *stamp-meet-is-valid-value1*:
assumes $valid_value\ val\ stamp1$
assumes $valid_stamp\ stamp2$

```

assumes stamp1 = IntegerStamp b1 lo1 hi1
assumes stamp2 = IntegerStamp b2 lo2 hi2
assumes meet stamp1 stamp2  $\neq$  IllegalStamp
shows valid-value val (meet stamp1 stamp2)
<proof>

```

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)
<proof>

```

8.4.5 Validity of conditional expressions

```

lemma conditional-eval-implies-valid-value:
assumes  $[m, p] \vdash \text{cond} \mapsto \text{condv}$ 
assumes  $\text{expr} = (\text{if } \text{val-to-bool } \text{condv} \text{ then } \text{expr1} \text{ else } \text{expr2})$ 
assumes  $[m, p] \vdash \text{expr} \mapsto \text{val}$ 
assumes  $\text{val} \neq \text{UndefVal}$ 
assumes valid-value condv (stamp-expr cond)
assumes valid-value val (stamp-expr expr)
assumes compatible (stamp-expr expr1) (stamp-expr expr2)
shows valid-value val (stamp-expr (ConditionalExpr cond expr1 expr2))
<proof>

```

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)  $\vee$  (bin-eval BinIntegerLessThan xv yv) = UndefVal
<proof>

```

```

lemma stamp-under-defn-inverse:
assumes stamp-under (stamp-expr y) (stamp-expr x)
assumes wf-stamp x  $\wedge$  wf-stamp y

```

```

assumes ([m, p] ⊢ x ↦ xv) ∧ ([m, p] ⊢ y ↦ yv)
shows ¬(val-to-bool (bin-eval BinIntegerLessThan xv yv)) ∨ (bin-eval BinIntegerLessThan xv yv) = UndefVal
⟨proof⟩

```

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 ⇒ '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 BinShortCircuitOr}

```

```

| markup DSL-Tokens.Abs = @{term UnaryExpr} $ @{term UnaryAbs}
| markup DSL-Tokens.Less = @{term BinaryExpr} $ @{term BinIntegerLessThan}
| markup DSL-Tokens.Equals = @{term BinaryExpr} $ @{term BinIntegerEquals}
| markup DSL-Tokens.Not = @{term UnaryExpr} $ @{term UnaryNot}
| markup DSL-Tokens.Negate = @{term UnaryExpr} $ @{term UnaryNeg}
| markup DSL-Tokens.LogicNegate = @{term UnaryExpr} $ @{term UnaryLogicNegation}
| markup DSL-Tokens.LeftShift = @{term BinaryExpr} $ @{term BinLeftShift}
| markup DSL-Tokens.RightShift = @{term BinaryExpr} $ @{term BinRightShift}
| markup DSL-Tokens.UnsignedRightShift = @{term BinaryExpr} $ @{term BinURightShift}
| markup DSL-Tokens.Conditional = @{term ConditionalExpr}
| markup DSL-Tokens.Constant = @{term ConstantExpr}
| markup DSL-Tokens.TrueConstant = @{term ConstantExpr (IntVal 32 1)}
| markup DSL-Tokens.FalseConstant = @{term ConstantExpr (IntVal 32 0)}
end
structure IRExprMarkup = DSL-Markup(IRExprTranslator);
>

```

ir expression translation

```

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

```

ir expression example

```

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

ConditionalExpr (BinaryExpr BinIntegerLessThan e1 e2) e1 e2

```

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  $\langle$  [( @{syntax-const -expandIntVal} , IntVal-
Markup.markup-expr [])  $\rangle$ 

```

value expression example

```

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

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

```

9.1.3 Word Markup

```

ML  $\langle$ 
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 shiftrl}
| 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 e1 e2) e1 e2
```

```
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[~x]  
value val[~x]  
value exp[~x]
```

```
value ~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, (Bytes.string content));

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

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

val - =
  Outer-Syntax.command command-keyword⟨export-phases⟩
    export information about encoded optimizations
    (Parse.path >>
      (fn name => Toplevel.theory (fn state => export-phases state name)))
,

```

ML-file *rewrites.ML*

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* :: *IRExpr* \Rightarrow *nat* **where**

unary-size:

size (*UnaryExpr* *op x*) = (*size* *x*) + 2 |

bin-const-size:

size (*BinaryExpr* *op x* (*ConstantExpr* *cy*)) = (*size* *x*) + 2 |

bin-size:

size (*BinaryExpr* *op x y*) = (*size* *x*) + (*size* *y*) + 2 |

cond-size:

size (*ConditionalExpr* *c t f*) = (*size* *c*) + (*size* *t*) + (*size* *f*) + 2 |

const-size:

size (*ConstantExpr* *c*) = 1 |

param-size:

size (*ParameterExpr* *ind s*) = 2 |

leaf-size:

size (*LeafExpr* *nid s*) = 2 |

size (*ConstantVar* *c*) = 2 |

size (*VariableExpr* *x s*) = 2

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 < size y
    <proof>

  lemma size-non-add[size-simps]: size (BinaryExpr op a b) = size a + size b + 2
     $\longleftrightarrow \neg(\text{is-ConstantExpr } b)$ 
    <proof>

  lemma size-non-const[size-simps]:
     $\neg \text{is-ConstantExpr } y \implies 1 < \text{size } y$ 
    <proof>

  lemma size-binary-const[size-simps]:
     $\text{size } (\text{BinaryExpr op } a \ b) = \text{size } a + 2 \longleftrightarrow (\text{is-ConstantExpr } b)$ 
    <proof>

  lemma size-flip-binary[size-simps]:
     $\neg(\text{is-ConstantExpr } y) \longrightarrow \text{size } (\text{BinaryExpr op } (\text{ConstantExpr } x) \ y) > \text{size } (\text{BinaryExpr op } y \ (\text{ConstantExpr } x))$ 
    <proof>

  lemma size-binary-lhs-a[size-simps]:

```

size (BinaryExpr op (BinaryExpr op' a b) c) > *size* a
 ⟨proof⟩

lemma *size-binary-lhs-b*[*size-simps*]:
size (BinaryExpr op (BinaryExpr op' a b) c) > *size* b
 ⟨proof⟩

lemma *size-binary-lhs-c*[*size-simps*]:
size (BinaryExpr op (BinaryExpr op' a b) c) > *size* c
 ⟨proof⟩

lemma *size-binary-rhs-a*[*size-simps*]:
size (BinaryExpr op c (BinaryExpr op' a b)) > *size* a
 ⟨proof⟩

lemma *size-binary-rhs-b*[*size-simps*]:
size (BinaryExpr op c (BinaryExpr op' a b)) > *size* b
 ⟨proof⟩

lemma *size-binary-rhs-c*[*size-simps*]:
size (BinaryExpr op c (BinaryExpr op' a b)) > *size* c
 ⟨proof⟩

lemma *size-binary-lhs*[*size-simps*]:
size (BinaryExpr op x y) > *size* x
 ⟨proof⟩

lemma *size-binary-rhs*[*size-simps*]:
size (BinaryExpr op x y) > *size* y
 ⟨proof⟩

lemmas *arith*[*size-simps*] = *Suc-leI add-strict-increasing order-less-trans trans-less-add2*

definition *well-formed-equal* :: *Value* ⇒ *Value* ⇒ *bool*
 (**infix** ≈ 50) **where**
well-formed-equal v₁ v₂ = (v₁ ≠ *UndefVal* → v₁ = v₂)

lemma *well-formed-equal-defn* [*simp*]:
well-formed-equal v₁ v₂ = (v₁ ≠ *UndefVal* → v₁ = v₂)
 ⟨proof⟩

end

10.1 AbsNode Phase

theory *AbsPhase*
imports
Common

begin

phase *AbsNode*
 terminating *size*
begin

lemma *abs-pos*:
 fixes $v :: ('a :: \text{len word})$
 assumes $0 \leq_s v$
 shows $(\text{if } v <_s 0 \text{ then } -v \text{ else } v) = v$
 $\langle \text{proof} \rangle$

lemma *abs-neg*:
 fixes $v :: ('a :: \text{len word})$
 assumes $v <_s 0$
 assumes $-(2^{\wedge}(\text{Nat.size } v - 1)) <_s v$
 shows $(\text{if } v <_s 0 \text{ then } -v \text{ else } v) = -v \wedge 0 <_s -v$
 $\langle \text{proof} \rangle$

lemma *abs-max-neg*:
 fixes $v :: ('a :: \text{len word})$
 assumes $v <_s 0$
 assumes $-(2^{\wedge}(\text{Nat.size } v - 1)) = v$
 shows $-v = v$
 $\langle \text{proof} \rangle$

lemma *final-abs*:
 fixes $v :: ('a :: \text{len word})$
 assumes $\text{take-bit } (\text{Nat.size } v) \ v = v$
 assumes $-(2^{\wedge}(\text{Nat.size } v - 1)) \neq v$
 shows $0 \leq_s (\text{if } v <_s 0 \text{ then } -v \text{ else } v)$

 $\langle \text{proof} \rangle$

lemma *wf-abs*: $\text{is-IntVal } x \implies \text{intval-abs } x \neq \text{UndefVal}$
 $\langle \text{proof} \rangle$

fun *bin-abs* :: $'a :: \text{len word} \Rightarrow 'a :: \text{len word}$ **where**
 $\text{bin-abs } v = (\text{if } (v <_s 0) \text{ then } (-v) \text{ else } v)$

lemma *val-abs-zero*:
 $\text{intval-abs } (\text{new-int } b \ 0) = \text{new-int } b \ 0$

<proof>

lemma *less-eq-zero*:

assumes *val-to-bool* (*val*[(*IntVal* *b* 0) < (*IntVal* *b* *v*)])

shows *int-signed-value* *b* *v* > 0

<proof>

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*)

<proof>

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*)

<proof>

lemma *val-bool-unwrap*:

val-to-bool (*bool-to-val* *v*) = *v*

<proof>

lemma *take-bit-unwrap*:

b = 64 \implies *take-bit* *b* (*v1*::64 word) = *v1*

<proof>

lemma *bit-less-eq-def*:

fixes *v1* *v2* :: 64 word

assumes *b* ≤ 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*)

<proof>

lemma *less-eq-def*:

shows *val-to-bool*(*val*[(*new-int* *b* *v1*) < (*new-int* *b* *v2*)]) \longleftrightarrow *v1* <_s *v2*

<proof>

lemma *val-abs-always-pos*:

assumes *intval-abs* (*new-int* *b* *v*) = (*new-int* *b* *v'*)

shows 0 ≤_s *v'*

<proof>

lemma *intval-abs-elim*:

assumes *intval-abs* *x* ≠ *UndefVal*

shows $\exists t v . x = \text{IntVal } t v \wedge \text{intval-abs } x = \text{new-int } t \text{ (if int-signed-value } t v < 0 \text{ then } -v \text{ else } v)$

<proof>

lemma *wf-abs-new-int*:

assumes *intval-abs* (*IntVal* *t v*) \neq *UndefVal*

shows *intval-abs* (*IntVal* *t v*) = *new-int* *t v* \vee *intval-abs* (*IntVal* *t v*) = *new-int* *t* ($-v$)

<proof>

lemma *mono-undef-abs*:

assumes *intval-abs* (*intval-abs* *x*) \neq *UndefVal*

shows *intval-abs* *x* \neq *UndefVal*

<proof>

lemma *val-abs-idem*:

assumes *intval-abs*(*intval-abs*(*x*)) \neq *UndefVal*

shows *intval-abs*(*intval-abs*(*x*)) = *intval-abs* *x*

<proof>

lemma *val-abs-negate*:

assumes *intval-abs* (*intval-negate* *x*) \neq *UndefVal*

shows *intval-abs* (*intval-negate* *x*) = *intval-abs* *x*

<proof>

Optimisations

optimization *AbsIdempotence*: *abs*(*abs*(*x*)) \mapsto *abs*(*x*)

<proof>

optimization *AbsNegate*: (*abs*($-x$)) \mapsto *abs*(*x*)

<proof>

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* \neq *UndefVal*

shows *bin-eval* *BinAdd* *x y* = *bin-eval* *BinAdd* *y x*

$\langle proof \rangle$

optimization *AddShiftConstantRight*: $((const\ v) + y) \mapsto y + (const\ v)$ when $\neg(is-ConstantExpr\ y)$
 $\langle proof \rangle$

optimization *AddShiftConstantRight2*: $((const\ v) + y) \mapsto y + (const\ v)$ when $\neg(is-ConstantExpr\ y)$
 $\langle proof \rangle$

lemma *is-neutral-0* [simp]:
 assumes 1: $intval-add\ (IntVal\ b\ x)\ (IntVal\ b\ 0) \neq UndefVal$
 shows $intval-add\ (IntVal\ b\ x)\ (IntVal\ b\ 0) = (new-int\ b\ x)$
 $\langle proof \rangle$

optimization *AddNeutral*: $(e + (const\ (IntVal\ 32\ 0))) \mapsto e$
 $\langle proof \rangle$

ML-val $\langle @\{term\ \langle x = y \rangle\} \rangle$

lemma *NeutralLeftSubVal*:
 assumes $e1 = new-int\ b\ ival$
 shows $val[(e1 - e2) + e2] \approx e1$
 $\langle proof \rangle$

optimization *RedundantSubAdd*: $((e_1 - e_2) + e_2) \mapsto e_1$
 $\langle proof \rangle$

lemma *allE2*: $(\forall x\ y. P\ x\ y) \implies (P\ a\ b \implies R) \implies R$
 $\langle proof \rangle$

lemma *just-goal2*:
 assumes 1: $(\forall\ a\ b. (intval-add\ (intval-sub\ a\ b)\ b \neq UndefVal \wedge a \neq UndefVal \implies$
 $intval-add\ (intval-sub\ a\ b)\ b = a))$
 shows $(BinaryExpr\ BinAdd\ (BinaryExpr\ BinSub\ e_1\ e_2)\ e_2) \geq e_1$
 $\langle proof \rangle$

optimization *RedundantSubAdd2*: $e_2 + (e_1 - e_2) \mapsto e_1$
 $\langle proof \rangle$

lemma *AddToSubHelperLowLevel*:
shows $intval\text{-}add\ (intval\text{-}negate\ e)\ y = intval\text{-}sub\ y\ e$ **(is** $?x = ?y$)
 $\langle proof \rangle$

print-phases

lemma *val-redundant-add-sub*:
assumes $a = new\text{-}int\ bb\ ival$
assumes $val[b + a] \neq UndefVal$
shows $val[(b + a) - b] = a$
 $\langle proof \rangle$

lemma *val-add-right-negate-to-sub*:
assumes $val[x + e] \neq UndefVal$
shows $val[x + (-e)] = val[x - e]$
 $\langle proof \rangle$

lemma *exp-add-left-negate-to-sub*:
 $exp[-e + y] \geq exp[y - e]$
 $\langle proof \rangle$

Optimisations

optimization *RedundantAddSub*: $(b + a) - b \mapsto a$
 $\langle proof \rangle$

optimization *AddRightNegateToSub*: $x + -e \mapsto x - e$
 $\langle proof \rangle$

optimization *AddLeftNegateToSub*: $-e + y \mapsto y - e$
 $\langle proof \rangle$

end

end

10.3 AndNode Phase

theory *AndPhase*

imports

Common

Proofs.StampEvalThms

begin

context *stamp-mask*

begin

lemma *AndRightFallthrough*: $((\text{and } (\text{not } (\downarrow x)) (\uparrow y)) = 0) \longrightarrow \text{exp}[x \ \& \ y] \geq \text{exp}[y]$
<proof>

lemma *AndLeftFallthrough*: $((\text{and } (\text{not } (\downarrow y)) (\uparrow x)) = 0) \longrightarrow \text{exp}[x \ \& \ y] \geq \text{exp}[x]$
<proof>

end

phase *AndNode*

terminating *size*

begin

lemma *bin-and-nots*:

$(\sim x \ \& \ \sim y) = (\sim(x \mid y))$
<proof>

lemma *bin-and-neutral*:

$(x \ \& \ \sim \text{False}) = x$
<proof>

lemma *val-and-equal*:

assumes $x = \text{new-int } b \ v$

and $\text{val}[x \ \& \ x] \neq \text{UndefVal}$

shows $\text{val}[x \ \& \ x] = x$

<proof>

lemma *val-and-nots*:

$\text{val}[\sim x \ \& \ \sim y] = \text{val}[\sim(x \mid y)]$

<proof>

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 $\text{val}[x \ \& \ \sim(\text{new-int } b' \ 0)] = x$
 $\langle \text{proof} \rangle$

lemma *val-and-zero*:
assumes $x = \text{new-int } b \ v$
shows $\text{val}[x \ \& \ (\text{IntVal } b \ 0)] = \text{IntVal } b \ 0$
 $\langle \text{proof} \rangle$

lemma *exp-and-equal*:
 $\text{exp}[x \ \& \ x] \geq \text{exp}[x]$
 $\langle \text{proof} \rangle$

lemma *exp-and-nots*:
 $\text{exp}[\sim x \ \& \ \sim y] \geq \text{exp}[\sim(x \mid y)]$
 $\langle \text{proof} \rangle$

lemma *exp-sign-extend*:
assumes $e = (1 << \text{In}) - 1$
shows $\text{BinaryExpr } \text{BinAnd } (\text{UnaryExpr } (\text{UnarySignExtend } \text{In } \text{Out}) \ x)$
 $\quad (\text{ConstantExpr } (\text{new-int } b \ e))$
 $\quad \geq (\text{UnaryExpr } (\text{UnaryZeroExtend } \text{In } \text{Out}) \ x)$
 $\langle \text{proof} \rangle$

lemma *val-and-commute[simp]*:
 $\text{val}[x \ \& \ y] = \text{val}[y \ \& \ x]$
 $\langle \text{proof} \rangle$

Optimisations

optimization *AndEqual*: $x \ \& \ x \longmapsto x$
 $\langle \text{proof} \rangle$

optimization *AndShiftConstantRight*: $((\text{const } x) \ \& \ y) \longmapsto y \ \& \ (\text{const } x)$
 $\text{when } \neg(\text{is-ConstantExpr } y)$
 $\langle \text{proof} \rangle$

optimization *AndNots*: $(\sim x) \ \& \ (\sim y) \mapsto \sim(x \mid y)$
 $\langle proof \rangle$

optimization *AndSignExtend*: *BinaryExpr* *BinAnd* (*UnaryExpr* (*UnarySignExtend* *In Out*) (*x*))

$(const \ (new-int \ b \ e))$
 $\mapsto (UnaryExpr \ (UnaryZeroExtend \ In \ Out) \ (x))$
 $when \ (e = (1 << In) - 1)$

$\langle proof \rangle$

optimization *AndNeutral*: $(x \ \& \ \sim(const \ (IntVal \ b \ 0))) \mapsto x$
 $when \ (wf-stamp \ x \wedge \ stamp-expr \ x = IntegerStamp \ b \ lo \ hi)$
 $\langle proof \rangle$

optimization *AndRightFallThrough*: $(x \ \& \ y) \mapsto y$
 $when \ (((and \ (not \ (IRExpr-down \ x)) \ (IRExpr-up \ y)) = 0))$
 $\langle proof \rangle$

optimization *AndLeftFallThrough*: $(x \ \& \ y) \mapsto x$
 $when \ (((and \ (not \ (IRExpr-down \ y)) \ (IRExpr-up \ x)) = 0))$
 $\langle proof \rangle$

end

end

10.4 BinaryNode Phase

theory *BinaryNode*

imports

Common

begin

phase *BinaryNode*

terminating *size*

begin

optimization *BinaryFoldConstant*: *BinaryExpr* *op* (*const v1*) (*const v2*) $\mapsto ConstantExpr \ (bin-eval \ op \ v1 \ v2)$
 $\langle proof \rangle$

print-facts

end

end

10.5 ConditionalNode Phase

theory *ConditionalPhase*

imports

Common

Proofs.StampEvalThms

begin

phase *ConditionalNode*

terminating *size*

begin

lemma *negates*: $\exists v\ b. e = \text{IntVal } b\ v \wedge b > 0 \implies \text{val-to-bool } (\text{val}[e]) \longleftrightarrow \neg(\text{val-to-bool } (\text{val}[!e]))$
 $\langle \text{proof} \rangle$

lemma *negation-condition-intval*:

assumes $e = \text{IntVal } b\ ie$

assumes $0 < b$

shows $\text{val}[(!e)\ ?\ x : y] = \text{val}[e\ ?\ y : x]$

$\langle \text{proof} \rangle$

lemma *negation-preserve-eval*:

assumes $[m, p] \vdash \text{exp}[!e] \mapsto v$

shows $\exists v'. ([m, p] \vdash \text{exp}[e] \mapsto v') \wedge v = \text{val}[!v']$

$\langle \text{proof} \rangle$

lemma *negation-preserve-eval-intval*:

assumes $[m, p] \vdash \text{exp}[!e] \mapsto v$

shows $\exists v'\ b\ vv. ([m, p] \vdash \text{exp}[e] \mapsto v') \wedge v' = \text{IntVal } b\ vv \wedge b > 0$

$\langle \text{proof} \rangle$

optimization *NegateConditionFlipBranches*: $((!e)\ ?\ x : y) \mapsto (e\ ?\ y : x)$

$\langle \text{proof} \rangle$

optimization *DefaultTrueBranch*: $(\text{true}\ ?\ x : y) \mapsto x\ \langle \text{proof} \rangle$

optimization *DefaultFalseBranch*: $(\text{false}\ ?\ x : y) \mapsto y\ \langle \text{proof} \rangle$

optimization *ConditionalEqualBranches*: $(e\ ?\ x : x) \mapsto x\ \langle \text{proof} \rangle$

optimization *condition-bounds-x*: $((u < v)\ ?\ x : y) \mapsto x$

when $(\text{stamp-under } (\text{stamp-expr } u) (\text{stamp-expr } v) \wedge \text{wf-stamp } u \wedge \text{wf-stamp } v)$

$\langle \text{proof} \rangle$

optimization *condition-bounds-y*: $((u < v)\ ?\ x : y) \mapsto y$

when $(\text{stamp-under } (\text{stamp-expr } v) (\text{stamp-expr } u) \wedge \text{wf-stamp } u \wedge \text{wf-stamp } v)$

$\langle \text{proof} \rangle$

lemma *val-optimise-integer-test*:

assumes $\exists v. x = \text{IntVal } 32 \ v$

shows $\text{val}[(x \ \& \ (\text{IntVal } 32 \ 1)) \ \text{eq} \ (\text{IntVal } 32 \ 0)) \ ? \ (\text{IntVal } 32 \ 0) : (\text{IntVal } 32 \ 1)]$

=

$\text{val}[x \ \& \ \text{IntVal } 32 \ 1]$

$\langle \text{proof} \rangle$

optimization *ConditionalEliminateKnownLess*: $((x < y) \ ? \ x : y) \mapsto x$

when $(\text{stamp-under} \ (\text{stamp-expr } x) \ (\text{stamp-expr } y))$

$\wedge \text{wf-stamp } x \wedge \text{wf-stamp } y$

$\langle \text{proof} \rangle$

optimization *ConditionalEqualIsRHS*: $((x \ \text{eq} \ y) \ ? \ x : y) \mapsto y$

$\langle \text{proof} \rangle$

optimization *normalizeX*: $((x \ \text{eq} \ \text{const} \ (\text{IntVal } 32 \ 0)) \ ?$

$(\text{const} \ (\text{IntVal } 32 \ 0)) : (\text{const} \ (\text{IntVal } 32 \ 1))) \mapsto x$

when $(\text{IRExpr-up } x = 1) \wedge \text{stamp-expr } x = \text{IntegerStamp}$

$b \ 0 \ 1$

$\langle \text{proof} \rangle$

optimization *normalizeX2*: $((x \ \text{eq} \ (\text{const} \ (\text{IntVal } 32 \ 1))) \ ?$

$(\text{const} \ (\text{IntVal } 32 \ 1)) : (\text{const} \ (\text{IntVal } 32 \ 0))) \mapsto x$

when $(x = \text{ConstantExpr} \ (\text{IntVal } 32 \ 0) \mid (x = \text{ConstantExpr}$

$(\text{IntVal } 32 \ 1))) \ \langle \text{proof} \rangle$

optimization *flipX*: $((x \ \text{eq} \ (\text{const} \ (\text{IntVal } 32 \ 0))) \ ?$

$(\text{const} \ (\text{IntVal } 32 \ 1)) : (\text{const} \ (\text{IntVal } 32 \ 0))) \mapsto$

$x \oplus (\text{const} \ (\text{IntVal } 32 \ 1))$

when $(x = \text{ConstantExpr} \ (\text{IntVal } 32 \ 0) \mid (x = \text{ConstantExpr}$

$(\text{IntVal } 32 \ 1))) \ \langle \text{proof} \rangle$

optimization *flipX2*: $((x \ \text{eq} \ (\text{const} \ (\text{IntVal } 32 \ 1))) \ ?$

$(\text{const} \ (\text{IntVal } 32 \ 0)) : (\text{const} \ (\text{IntVal } 32 \ 1))) \mapsto$

$x \oplus (\text{const} \ (\text{IntVal } 32 \ 1))$

when $(x = \text{ConstantExpr} \ (\text{IntVal } 32 \ 0) \mid (x = \text{ConstantExpr}$

$(\text{IntVal } 32 \ 1))) \ \langle \text{proof} \rangle$

lemma *stamp-of-default*:

assumes $\text{stamp-expr } x = \text{default-stamp}$

assumes $\text{wf-stamp } x$

shows $([m, p] \vdash x \mapsto v) \longrightarrow (\exists vv. v = \text{IntVal } 32 \ vv)$

$\langle \text{proof} \rangle$

optimization *OptimiseIntegerTest*:

$((x \ \& \ (\text{const} \ (\text{IntVal} \ 32 \ 1))) \ \text{eq} \ (\text{const} \ (\text{IntVal} \ 32 \ 0))) \ ?$
 $(\text{const} \ (\text{IntVal} \ 32 \ 0)) : (\text{const} \ (\text{IntVal} \ 32 \ 1))) \mapsto$
 $x \ \& \ (\text{const} \ (\text{IntVal} \ 32 \ 1))$
 $\text{when} \ (\text{stamp-expr} \ x = \text{default-stamp} \wedge \text{wf-stamp} \ x)$
 $\langle \text{proof} \rangle$

optimization *opt-optimise-integer-test-2*:

$((x \ \& \ (\text{const} \ (\text{IntVal} \ 32 \ 1))) \ \text{eq} \ (\text{const} \ (\text{IntVal} \ 32 \ 0))) \ ?$
 $(\text{const} \ (\text{IntVal} \ 32 \ 0)) : (\text{const} \ (\text{IntVal} \ 32 \ 1))) \mapsto$
 $\quad \quad \quad x$
 $\text{when} \ (x = \text{ConstantExpr} \ (\text{IntVal} \ 32 \ 0) \mid (x = \text{ConstantExpr} \ (\text{IntVal} \ 32 \ 1))) \ \langle \text{proof} \rangle$

end

end

10.6 MulNode Phase

theory *MulPhase*

imports

Common

Proofs.StampEvalThms

begin

fun *mul-size* :: *IRExpr* \Rightarrow *nat* **where**

$\text{mul-size} \ (\text{UnaryExpr} \ \text{op} \ e) = (\text{mul-size} \ e) + 2 \mid$
 $\text{mul-size} \ (\text{BinaryExpr} \ \text{BinMul} \ x \ y) = ((\text{mul-size} \ x) + (\text{mul-size} \ y) + 2) * 2 \mid$
 $\text{mul-size} \ (\text{BinaryExpr} \ \text{op} \ x \ y) = (\text{mul-size} \ x) + (\text{mul-size} \ y) + 2 \mid$
 $\text{mul-size} \ (\text{ConditionalExpr} \ \text{cond} \ t \ f) = (\text{mul-size} \ \text{cond}) + (\text{mul-size} \ t) + (\text{mul-size} \ f) + 2 \mid$
 $\text{mul-size} \ (\text{ConstantExpr} \ c) = 1 \mid$
 $\text{mul-size} \ (\text{ParameterExpr} \ \text{ind} \ s) = 2 \mid$
 $\text{mul-size} \ (\text{LeafExpr} \ \text{nid} \ s) = 2 \mid$
 $\text{mul-size} \ (\text{ConstantVar} \ c) = 2 \mid$
 $\text{mul-size} \ (\text{VariableExpr} \ x \ s) = 2$

phase *MulNode*

terminating *mul-size*

begin

lemma *bin-eliminate-redundant-negative:*

uminus ($x :: 'a::len\ word$) * *uminus* ($y :: 'a::len\ word$) = $x * y$
<proof>

lemma *bin-multiply-identity:*

$(x :: 'a::len\ word) * 1 = x$
<proof>

lemma *bin-multiply-eliminate:*

$(x :: 'a::len\ word) * 0 = 0$
<proof>

lemma *bin-multiply-negative:*

$(x :: 'a::len\ word) * \text{uminus } 1 = \text{uminus } x$
<proof>

lemma *bin-multiply-power-2:*

$(x :: 'a::len\ word) * (2^j) = x << j$
<proof>

lemma *take-bit64[simp]:*

fixes $w :: int64$
shows $\text{take-bit } 64\ w = w$
<proof>

lemma *mergeTakeBit:*

fixes $a :: nat$
fixes $b\ c :: 64\ word$
shows $\text{take-bit } a\ (\text{take-bit } a\ (b) * \text{take-bit } a\ (c)) =$
 $\text{take-bit } a\ (b * c)$
<proof>

lemma *val-eliminate-redundant-negative:*

assumes $\text{val}[-x * -y] \neq \text{UndefVal}$
shows $\text{val}[-x * -y] = \text{val}[x * y]$
<proof>

lemma *val-multiply-neutral:*

assumes $x = \text{new-int } b\ v$
shows $\text{val}[x * (\text{IntVal } b\ 1)] = \text{val}[x]$
<proof>

lemma *val-multiply-zero*:

assumes $x = \text{new-int } b \ v$
shows $\text{val}[x * (\text{IntVal } b \ 0)] = \text{IntVal } b \ 0$
<proof>

lemma *val-multiply-negative*:

assumes $x = \text{new-int } b \ v$
shows $\text{val}[x * \text{intval-negate } (\text{IntVal } b \ 1)] = \text{intval-negate } x$
<proof>

lemma *val-MulPower2*:

fixes $i :: 64 \text{ word}$
assumes $y = \text{IntVal } 64 \ (2 \wedge \text{unat}(i))$
and $0 < i$
and $i < 64$
and $\text{val}[x * y] \neq \text{UndefVal}$
shows $\text{val}[x * y] = \text{val}[x << \text{IntVal } 64 \ i]$
<proof>

lemma *val-MulPower2Add1*:

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 $\text{val}[x * y] = \text{val}[(x << \text{IntVal } 64 \ i) + x]$
<proof>

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 $\text{val}[x * y] = \text{val}[(x << \text{IntVal } 64 \ i) - x]$
<proof>

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 $\text{val}[x * (q + a)] = \text{val}[(x * q) + (x * a)]$
<proof>

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 $\text{val}[x * y] = \text{val}[(x << \text{IntVal } 64\ i) + (x << \text{IntVal } 64\ j)]$
 $\langle \text{proof} \rangle$

thm-oracles *val-MulPower2AddPower2*

lemma *exp-multiply-zero-64*:
 $\text{exp}[x * (\text{const } (\text{IntVal } 64\ 0))] \geq \text{ConstantExpr } (\text{IntVal } 64\ 0)$
 $\langle \text{proof} \rangle$

lemma *exp-multiply-neutral*:
 $\text{exp}[x * (\text{const } (\text{IntVal } b\ 1))] \geq x$
 $\langle \text{proof} \rangle$

thm-oracles *exp-multiply-neutral*

lemma *exp-MulPower2*:
fixes $i :: 64\ \text{word}$
assumes $y = \text{ConstantExpr } (\text{IntVal } 64\ (2 \wedge \text{unat}(i)))$
and $0 < i$
and $i < 64$
and $\text{exp}[x > (\text{const } \text{IntVal } b\ 0)]$
and $\text{exp}[y > (\text{const } \text{IntVal } b\ 0)]$
shows $\text{exp}[x * y] \geq \text{exp}[x << \text{ConstantExpr } (\text{IntVal } 64\ i)]$
 $\langle \text{proof} \rangle$

lemma *exp-MulPower2Add1*:
fixes $i :: 64\ \text{word}$
assumes $y = \text{ConstantExpr } (\text{IntVal } 64\ ((2 \wedge \text{unat}(i)) + 1))$
and $0 < i$
and $i < 64$
and $\text{exp}[x > (\text{const } \text{IntVal } b\ 0)]$
and $\text{exp}[y > (\text{const } \text{IntVal } b\ 0)]$
shows $\text{exp}[x * y] \geq \text{exp}[(x << \text{ConstantExpr } (\text{IntVal } 64\ i)) + x]$
 $\langle \text{proof} \rangle$

lemma *exp-MulPower2Sub1*:
fixes $i :: 64\ \text{word}$
assumes $y = \text{ConstantExpr } (\text{IntVal } 64\ ((2 \wedge \text{unat}(i)) - 1))$

and $0 < i$
and $i < 64$
and $\text{exp}[x > (\text{const IntVal } b \ 0)]$
and $\text{exp}[y > (\text{const IntVal } b \ 0)]$
shows $\text{exp}[x * y] \geq \text{exp}[(x << \text{ConstantExpr } (\text{IntVal } 64 \ i)) - x]$
 $\langle \text{proof} \rangle$

lemma *exp-MulPower2AddPower2*:
fixes $i \ j :: 64 \text{ word}$
assumes $y = \text{ConstantExpr } (\text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) + (2 \wedge \text{unat}(j))))$
and $0 < i$
and $0 < j$
and $i < 64$
and $j < 64$
and $\text{exp}[x > (\text{const IntVal } b \ 0)]$
and $\text{exp}[y > (\text{const IntVal } b \ 0)]$
shows $\text{exp}[x * y] \geq \text{exp}[(x << \text{ConstantExpr } (\text{IntVal } 64 \ i)) + (x << \text{ConstantExpr } (\text{IntVal } 64 \ j))]$
 $\langle \text{proof} \rangle$

lemma *greaterConstant*:
fixes $a \ b :: 64 \text{ word}$
assumes $a > b$
and $y = \text{ConstantExpr } (\text{IntVal } 64 \ a)$
and $x = \text{ConstantExpr } (\text{IntVal } 64 \ b)$
shows $\text{exp}[y > x]$
 $\langle \text{proof} \rangle$

lemma *exp-distribute-multiplication*:
shows $\text{exp}[(x * q) + (x * a)] \geq \text{exp}[x * (q + a)]$
 $\langle \text{proof} \rangle$

Optimisations

optimization *EliminateRedundantNegative*: $-x * -y \mapsto x * y$
 $\langle \text{proof} \rangle$

optimization *MulNeutral*: $x * \text{ConstantExpr } (\text{IntVal } b \ 1) \mapsto x$
 $\langle \text{proof} \rangle$

optimization *MulEliminator*: $x * \text{ConstantExpr } (\text{IntVal } b \ 0) \mapsto \text{const } (\text{IntVal } b \ 0)$
 $\langle \text{proof} \rangle$

optimization *MulNegate*: $x * -(\text{const } (\text{IntVal } b \ 1)) \mapsto -x$
 $\langle \text{proof} \rangle$

```

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 ( $[m, p] \vdash x \mapsto v \longrightarrow (\exists vv\ b. (v = \text{IntVal } b\ vv \wedge \text{val-to-bool val}[(\text{IntVal } b\ 0) < v]))$ )
   $\langle \text{proof} \rangle$ 

optimization MulPower2:  $x * y \longmapsto x << \text{const } (\text{IntVal } 64\ i)$ 
  when ( $i > 0 \wedge$ 
     $64 > i \wedge$ 
     $y = \text{exp}[\text{const } (\text{IntVal } 64\ (2 \wedge \text{unat}(i)))]$ )
   $\langle \text{proof} \rangle$ 

optimization MulPower2Add1:  $x * y \longmapsto (x << \text{const } (\text{IntVal } 64\ i)) + x$ 
  when ( $i > 0 \wedge$ 
     $64 > i \wedge$ 
     $y = \text{ConstantExpr } (\text{IntVal } 64\ ((2 \wedge \text{unat}(i)) + 1))$ )
   $\langle \text{proof} \rangle$ 

optimization MulPower2Sub1:  $x * y \longmapsto (x << \text{const } (\text{IntVal } 64\ i)) - x$ 
  when ( $i > 0 \wedge$ 
     $64 > i \wedge$ 
     $y = \text{ConstantExpr } (\text{IntVal } 64\ ((2 \wedge \text{unat}(i)) - 1))$ )
   $\langle \text{proof} \rangle$ 

end

end

```

10.7 Experimental AndNode Phase

```

theory NewAnd
imports
  Common
  Graph.Long
begin

lemma bin-distribute-and-over-or:
   $\text{bin}[z \ \& \ (x \mid y)] = \text{bin}[(z \ \& \ x) \mid (z \ \& \ y)]$ 
   $\langle \text{proof} \rangle$ 

lemma intval-distribute-and-over-or:

```

$val[z \& (x \mid y)] = val[(z \& x) \mid (z \& y)]$
 $\langle proof \rangle$

lemma *exp-distribute-and-over-or*:
 $exp[z \& (x \mid y)] \geq exp[(z \& x) \mid (z \& y)]$
 $\langle proof \rangle$

lemma *intval-and-commute*:
 $val[x \& y] = val[y \& x]$
 $\langle proof \rangle$

lemma *intval-or-commute*:
 $val[x \mid y] = val[y \mid x]$
 $\langle proof \rangle$

lemma *intval-xor-commute*:
 $val[x \oplus y] = val[y \oplus x]$
 $\langle proof \rangle$

lemma *exp-and-commute*:
 $exp[x \& z] \geq exp[z \& x]$
 $\langle proof \rangle$

lemma *exp-or-commute*:
 $exp[x \mid y] \geq exp[y \mid x]$
 $\langle proof \rangle$

lemma *exp-xor-commute*:
 $exp[x \oplus y] \geq exp[y \oplus x]$
 $\langle proof \rangle$

lemma *bin-eliminate-y*:
assumes $bin[y \& z] = 0$
shows $bin[(x \mid y) \& z] = bin[x \& z]$
 $\langle proof \rangle$

lemma *intval-eliminate-y*:
assumes $val[y \& z] = IntVal\ b\ 0$
shows $val[(x \mid y) \& z] = val[x \& z]$
 $\langle proof \rangle$

lemma *intval-and-associative*:
 $val[(x \& y) \& z] = val[x \& (y \& z)]$
 $\langle proof \rangle$

lemma *intval-or-associative*:
 $val[(x \mid y) \mid z] = val[x \mid (y \mid z)]$

$\langle proof \rangle$

lemma *intval-xor-associative:*

$val[(x \oplus y) \oplus z] = val[x \oplus (y \oplus z)]$

$\langle proof \rangle$

lemma *exp-and-associative:*

$exp[(x \& y) \& z] \geq exp[x \& (y \& z)]$

$\langle proof \rangle$

lemma *exp-or-associative:*

$exp[(x \mid y) \mid z] \geq exp[x \mid (y \mid z)]$

$\langle proof \rangle$

lemma *exp-xor-associative:*

$exp[(x \oplus y) \oplus z] \geq exp[x \oplus (y \oplus z)]$

$\langle proof \rangle$

lemma *intval-and-absorb-or:*

assumes $\exists b \ v . \ x = new_int \ b \ v$

assumes $val[x \& (x \mid y)] \neq UndefinedVal$

shows $val[x \& (x \mid y)] = val[x]$

$\langle proof \rangle$

lemma *intval-or-absorb-and:*

assumes $\exists b \ v . \ x = new_int \ b \ v$

assumes $val[x \mid (x \& y)] \neq UndefinedVal$

shows $val[x \mid (x \& y)] = val[x]$

$\langle proof \rangle$

lemma *exp-and-absorb-or:*

$exp[x \& (x \mid y)] \geq exp[x]$

$\langle proof \rangle$

lemma *exp-or-absorb-and:*

$exp[x \mid (x \& y)] \geq exp[x]$

$\langle proof \rangle$

lemma

assumes $y = 0$

shows $x + y = or \ x \ y$

$\langle proof \rangle$

lemma *no-overlap-or:*

assumes $and \ x \ y = 0$

shows $x + y = or \ x \ y$

<proof>

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 $\exists \ b. \ val[xv \ \& \ yv] = \text{new-int } b \ 0$
 <proof>

lemma *exp-eliminate-y*:
 $\text{and } (\uparrow y) (\uparrow z) = 0 \longrightarrow \text{BinaryExpr BinAnd } (\text{BinaryExpr BinOr } x \ y) \ z \geq \text{BinaryExpr BinAnd } x \ z$
 <proof>

lemma *leadingZeroBounds*:
 fixes $x :: 'a::\text{len word}$
 assumes $n = \text{numberOfLeadingZeros } x$
 shows $0 \leq n \wedge n \leq \text{Nat.size } x$
 <proof>

lemma *above-nth-not-set*:
 fixes $x :: \text{int64}$
 assumes $n = 64 - \text{numberOfLeadingZeros } x$
 shows $j > n \longrightarrow \neg(\text{bit } x \ j)$
 <proof>

no-notation *LogicNegationNotation* (!-)

lemma *zero-horner*:
 $\text{horner-sum of-bool } 2 \ (\text{map } (\lambda x. \text{False}) \ xs) = 0$
 <proof>

lemma *zero-map*:
 assumes $j \leq n$
 assumes $\forall i. \ j \leq i \longrightarrow \neg(f \ i)$
 shows $\text{map } f \ [0..<n] = \text{map } f \ [0..<j] \ @ \ \text{map } (\lambda x. \text{False}) \ [j..<n]$
 <proof>

lemma *map-join-horner*:

assumes $\text{map } f \ [0..<n] = \text{map } f \ [0..<j] \ @ \ \text{map } (\lambda x. \text{False}) \ [j..<n]$
shows $\text{horner-sum of-bool } (2::'a::\text{len word}) \ (\text{map } f \ [0..<n]) = \text{horner-sum of-bool}$
 $2 \ (\text{map } f \ [0..<j])$
 $\langle \text{proof} \rangle$

lemma *split-horner*:

assumes $j \leq n$
assumes $\forall i. j \leq i \longrightarrow \neg(f \ i)$
shows $\text{horner-sum of-bool } (2::'a::\text{len word}) \ (\text{map } f \ [0..<n]) = \text{horner-sum of-bool}$
 $2 \ (\text{map } f \ [0..<j])$
 $\langle \text{proof} \rangle$

lemma *transfer-map*:

assumes $\forall i. i < n \longrightarrow f \ i = f' \ i$
shows $(\text{map } f \ [0..<n]) = (\text{map } f' \ [0..<n])$
 $\langle \text{proof} \rangle$

lemma *transfer-horner*:

assumes $\forall i. i < n \longrightarrow f \ i = f' \ i$
shows $\text{horner-sum of-bool } (2::'a::\text{len word}) \ (\text{map } f \ [0..<n]) = \text{horner-sum of-bool}$
 $2 \ (\text{map } f' \ [0..<n])$
 $\langle \text{proof} \rangle$

lemma *L1*:

assumes $n = 64 - \text{numberOfLeadingZeros } (\uparrow z)$
assumes $[m, p] \vdash z \mapsto \text{IntVal } b \ zv$
shows $\text{and } v \ zv = \text{and } (v \bmod 2^{\wedge n}) \ zv$
 $\langle \text{proof} \rangle$

lemma *up-mask-upper-bound*:

assumes $[m, p] \vdash x \mapsto \text{IntVal } b \ xv$
shows $xv \leq (\uparrow x)$
 $\langle \text{proof} \rangle$

lemma *L2*:

assumes $\text{numberOfLeadingZeros } (\uparrow z) + \text{numberOfTrailingZeros } (\uparrow y) \geq 64$
assumes $n = 64 - \text{numberOfLeadingZeros } (\uparrow z)$
assumes $[m, p] \vdash z \mapsto \text{IntVal } b \ zv$
assumes $[m, p] \vdash y \mapsto \text{IntVal } b \ yv$
shows $yv \bmod 2^{\wedge n} = 0$
 $\langle \text{proof} \rangle$

thm-oracles *L1 L2*

lemma *unfold-binary-width-add*:

shows $([m, p] \vdash \text{BinaryExpr BinAdd } xe \ ye \mapsto \text{IntVal } b \ val) = (\exists \ x \ y. \\ ([m, p] \vdash xe \mapsto \text{IntVal } b \ x) \wedge \\ ([m, p] \vdash ye \mapsto \text{IntVal } b \ y) \wedge$

$(IntVal\ b\ val = bin\text{-}eval\ BinAdd\ (IntVal\ b\ x)\ (IntVal\ b\ y)) \wedge$
 $(IntVal\ b\ val \neq UndefVal)$
 $) \rangle \text{ (is } ?L = ?R)$
 $\langle proof \rangle$

lemma *unfold-binary-width-and:*

shows $([m,p] \vdash BinaryExpr\ BinAnd\ xe\ ye \mapsto IntVal\ b\ val) = (\exists\ x\ y.$
 $([m,p] \vdash xe \mapsto IntVal\ b\ x) \wedge$
 $([m,p] \vdash ye \mapsto IntVal\ b\ y) \wedge$
 $(IntVal\ b\ val = bin\text{-}eval\ BinAnd\ (IntVal\ b\ x)\ (IntVal\ b\ y)) \wedge$
 $(IntVal\ b\ val \neq UndefVal)$
 $) \rangle \text{ (is } ?L = ?R)$
 $\langle proof \rangle$

lemma *mod-dist-over-add-right:*

fixes $a\ b\ c :: int64$
fixes $n :: nat$
assumes $1: 0 < n$
assumes $2: n < 64$
shows $(a + b \bmod 2^n) \bmod 2^n = (a + b) \bmod 2^n$
 $\langle proof \rangle$

lemma *numberOfLeadingZeros-range:*

$0 \leq numberOfLeadingZeros\ n \wedge numberOfLeadingZeros\ n \leq Nat.size\ n$
 $\langle proof \rangle$

lemma *improved-opt:*

assumes $numberOfLeadingZeros\ (\uparrow z) + numberOfTrailingZeros\ (\uparrow y) \geq 64$
shows $exp[(x + y) \& z] \geq exp[x \& z]$
 $\langle proof \rangle$

thm-oracles *improved-opt*

end

phase *NewAnd*

terminating *size*

begin

optimization *redundant-lhs-y-or:* $((x \mid y) \& z) \mapsto x \& z$
 $when\ (((and\ (IRExpr\text{-}up\ y)\ (IRExpr\text{-}up\ z)) = 0))$
 $\langle proof \rangle$

optimization *redundant-lhs-x-or:* $((x \mid y) \& z) \mapsto y \& z$

when (((*and* (*IRExpr-up* *x*) (*IRExpr-up* *z*)) = 0))

⟨*proof*⟩

optimization *redundant-rhs-y-or*: (*z* & (*x* | *y*)) \mapsto *z* & *x*

when (((*and* (*IRExpr-up* *y*) (*IRExpr-up* *z*)) = 0))

⟨*proof*⟩

optimization *redundant-rhs-x-or*: (*z* & (*x* | *y*)) \mapsto *z* & *y*

when (((*and* (*IRExpr-up* *x*) (*IRExpr-up* *z*)) = 0))

⟨*proof*⟩

end

end

10.8 NotNode Phase

theory *NotPhase*

imports

Common

begin

phase *NotNode*

terminating *size*

begin

lemma *bin-not-cancel*:

$\text{bin}[\neg(\neg(e))] = \text{bin}[e]$

 ⟨*proof*⟩

lemma *val-not-cancel*:

assumes $\text{val}[\sim(\text{new-int } b \ v)] \neq \text{UndefVal}$

shows $\text{val}[\sim(\sim(\text{new-int } b \ v))] = (\text{new-int } b \ v)$

 ⟨*proof*⟩

lemma *exp-not-cancel*:

$\text{exp}[\sim(\sim a)] \geq \text{exp}[a]$

 ⟨*proof*⟩

Optimisations

optimization *NotCancel*: $\text{exp}[\sim(\sim a)] \mapsto a$

 ⟨*proof*⟩

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 $(\text{and } (\text{not } (\downarrow x)) (\uparrow y)) = 0$

shows $\text{exp}[x \mid y] \geq \text{exp}[x]$

$\langle \text{proof} \rangle$

lemma *OrRightFallthrough*:

assumes $(\text{and } (\text{not } (\downarrow y)) (\uparrow x)) = 0$

shows $\text{exp}[x \mid y] \geq \text{exp}[y]$

$\langle \text{proof} \rangle$

end

phase *OrNode*

terminating *size*

begin

lemma *bin-or-equal*:

$\text{bin}[x \mid x] = \text{bin}[x]$

$\langle \text{proof} \rangle$

lemma *bin-shift-const-right-helper*:

$x \mid y = y \mid x$

$\langle proof \rangle$

lemma *bin-or-not-operands*:

$(\sim x \mid \sim y) = (\sim(x \ \& \ y))$

$\langle proof \rangle$

lemma *val-or-equal*:

assumes $x = \text{new-int } b \ v$

and $(\text{val}[x \mid x] \neq \text{UndefVal})$

shows $\text{val}[x \mid x] = \text{val}[x]$

$\langle proof \rangle$

lemma *val-elim-redundant-false*:

assumes $x = \text{new-int } b \ v$

and $\text{val}[x \mid \text{false}] \neq \text{UndefVal}$

shows $\text{val}[x \mid \text{false}] = \text{val}[x]$

$\langle proof \rangle$

lemma *val-shift-const-right-helper*:

$\text{val}[x \mid y] = \text{val}[y \mid x]$

$\langle proof \rangle$

lemma *val-or-not-operands*:

$\text{val}[\sim x \mid \sim y] = \text{val}[\sim(x \ \& \ y)]$

$\langle proof \rangle$

lemma *exp-or-equal*:

$\text{exp}[x \mid x] \geq \text{exp}[x]$

$\langle proof \rangle$

lemma *exp-elim-redundant-false*:

$\text{exp}[x \mid \text{false}] \geq \text{exp}[x]$

$\langle proof \rangle$

Optimisations

optimization *OrEqual*: $x \mid x \mapsto x$

$\langle proof \rangle$

optimization *OrShiftConstantRight*: $((\text{const } x) \mid y) \mapsto y \mid (\text{const } x) \text{ when } \neg(\text{is-ConstantExpr } y)$

$\langle proof \rangle$

optimization *EliminateRedundantFalse*: $x \mid \text{false} \mapsto x$

$\langle proof \rangle$

optimization *OrNotOperands*: $(\sim x \mid \sim y) \mapsto \sim(x \ \& \ y)$

$\langle proof \rangle$

optimization *OrLeftFallthrough*:

$x \mid y \mapsto x$ when $((\text{and } (\text{not } (\text{IExpr-down } x)) (\text{IExpr-up } y)) = 0)$
 $\langle \text{proof} \rangle$

optimization *OrRightFallthrough*:

$x \mid y \mapsto y$ when $((\text{and } (\text{not } (\text{IExpr-down } y)) (\text{IExpr-up } x)) = 0)$
 $\langle \text{proof} \rangle$

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*

$\langle \text{proof} \rangle$

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*

$\langle \text{proof} \rangle$

optimization *e*:

$x * (\text{const } c) \mapsto x << (\text{const } n)$ when $(n = \text{intval-log2 } c \wedge \text{in-bounds } n \ 0 \ 32)$

$\langle \text{proof} \rangle$

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 $\text{intval-div } x \ (\text{IntVal } 32 \ 1) = x$

$\langle \text{proof} \rangle$

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 $\text{intval-mod } x \ (\text{IntVal } 32 \ 1) = \text{IntVal } 32 \ 0$

$\langle \text{proof} \rangle$

value $\text{word-of-int } (\text{sint } (x2::32 \ \text{word}) \ \text{smod } 1)$

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  $((x :: ('a :: len) \text{ word}) + (y :: ('a :: len) \text{ word})) - y = x$ 
 $\langle \text{proof} \rangle$ 
```

```
lemma sub-self-is-zero:
shows  $(x :: ('a :: len) \text{ word}) - x = 0$ 
 $\langle \text{proof} \rangle$ 
```

```
lemma bin-sub-then-left-add:
shows  $(x :: ('a :: len) \text{ word}) - (x + (y :: ('a :: len) \text{ word})) = -y$ 
 $\langle \text{proof} \rangle$ 
```

```
lemma bin-sub-then-left-sub:
shows  $(x :: ('a :: len) \text{ word}) - (x - (y :: ('a :: len) \text{ word})) = y$ 
 $\langle \text{proof} \rangle$ 
```

```
lemma bin-subtract-zero:
shows  $(x :: 'a :: len \text{ word}) - (0 :: 'a :: len \text{ word}) = x$ 
 $\langle \text{proof} \rangle$ 
```

```
lemma bin-sub-negative-value:
 $(x :: ('a :: len) \text{ word}) - (-(y :: ('a :: len) \text{ word})) = x + y$ 
 $\langle \text{proof} \rangle$ 
```

```
lemma bin-sub-self-is-zero:
 $(x :: ('a :: len) \text{ word}) - x = 0$ 
 $\langle \text{proof} \rangle$ 
```

```
lemma bin-sub-negative-const:
 $(x :: 'a :: len \text{ word}) - (-(y :: 'a :: len \text{ word})) = x + y$ 
 $\langle \text{proof} \rangle$ 
```

```
lemma val-sub-after-right-add-2:
```

assumes $x = \text{new-int } b \ v$
assumes $\text{val}[(x + y) - y] \neq \text{UndefVal}$
shows $\text{val}[(x + y) - y] = \text{val}[x]$
 $\langle \text{proof} \rangle$

lemma *val-sub-after-left-sub*:
assumes $\text{val}[(x - y) - x] \neq \text{UndefVal}$
shows $\text{val}[(x - y) - x] = \text{val}[-y]$
 $\langle \text{proof} \rangle$

lemma *val-sub-then-left-sub*:
assumes $y = \text{new-int } b \ v$
assumes $\text{val}[x - (x - y)] \neq \text{UndefVal}$
shows $\text{val}[x - (x - y)] = \text{val}[y]$
 $\langle \text{proof} \rangle$

lemma *val-subtract-zero*:
assumes $x = \text{new-int } b \ v$
assumes $\text{intval-sub } x \ (\text{IntVal } b \ 0) \neq \text{UndefVal}$
shows $\text{intval-sub } x \ (\text{IntVal } b \ 0) = \text{val}[x]$
 $\langle \text{proof} \rangle$

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 $\text{intval-sub } (\text{IntVal } b \ 0) \ x = \text{val}[-x]$
 $\langle \text{proof} \rangle$

lemma *val-sub-then-left-add*:
assumes $\text{val}[x - (x + y)] \neq \text{UndefVal}$
shows $\text{val}[x - (x + y)] = \text{val}[-y]$
 $\langle \text{proof} \rangle$

lemma *val-sub-negative-value*:
assumes $\text{val}[x - (-y)] \neq \text{UndefVal}$
shows $\text{val}[x - (-y)] = \text{val}[x + y]$
 $\langle \text{proof} \rangle$

lemma *val-sub-self-is-zero*:
assumes $x = \text{new-int } b \ v \wedge \text{val}[x - x] \neq \text{UndefVal}$
shows $\text{val}[x - x] = \text{new-int } b \ 0$
 $\langle \text{proof} \rangle$

lemma *val-sub-negative-const*:
assumes $y = \text{new-int } b \ v \wedge \text{val}[x - (-y)] \neq \text{UndefVal}$
shows $\text{val}[x - (-y)] = \text{val}[x + y]$
 $\langle \text{proof} \rangle$

lemma *exp-sub-after-right-add*:
shows $\text{exp}[(x + y) - y] \geq \text{exp}[x]$
 $\langle \text{proof} \rangle$

lemma *exp-sub-after-right-add2*:
shows $\text{exp}[(x + y) - x] \geq \text{exp}[y]$
 $\langle \text{proof} \rangle$

lemma *exp-sub-negative-value*:
 $\text{exp}[x - (-y)] \geq \text{exp}[x + y]$
 $\langle \text{proof} \rangle$

lemma *exp-sub-then-left-sub*:
 $\text{exp}[x - (x - y)] \geq \text{exp}[y]$
 $\langle \text{proof} \rangle$

thm-oracles *exp-sub-then-left-sub*

Optimisations

optimization *SubAfterAddRight*: $((x + y) - y) \mapsto x$
 $\langle \text{proof} \rangle$

optimization *SubAfterAddLeft*: $((x + y) - x) \mapsto y$
 $\langle \text{proof} \rangle$

optimization *SubAfterSubLeft*: $((x - y) - x) \mapsto -y$
 $\langle \text{proof} \rangle$

optimization *SubThenAddLeft*: $(x - (x + y)) \mapsto -y$
 $\langle \text{proof} \rangle$

optimization *SubThenAddRight*: $(y - (x + y)) \mapsto -x$
 $\langle \text{proof} \rangle$

optimization *SubThenSubLeft*: $(x - (x - y)) \mapsto y$
 $\langle \text{proof} \rangle$

optimization *SubtractZero*: $(x - (\text{const IntVal } b \ 0)) \mapsto x$
 $\langle \text{proof} \rangle$

thm-oracles *SubtractZero*

optimization *SubNegativeValue*: $(x - (-y)) \mapsto x + y$
 $\langle \text{proof} \rangle$

thm-oracles *SubNegativeValue*

lemma *negate-idempotent*:

assumes $x = \text{IntVal } b \ v \wedge \text{take-bit } b \ v = v$

shows $x = \text{val}[-(-x)]$

$\langle \text{proof} \rangle$

optimization *ZeroSubtractValue*: $((\text{const IntVal } b \ 0) - x) \mapsto (-x)$

when $(\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \ lo$

$hi \wedge \neg(\text{is-ConstantExpr } x))$

$\langle \text{proof} \rangle$

optimization *SubSelfIsZero*: $(x - x) \mapsto \text{const IntVal } b \ 0$ *when*

$(\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \ lo \ hi)$

$\langle \text{proof} \rangle$

end

end

10.14 XorNode Phase

theory *XorPhase*

imports

Common

Proofs.StampEvalThms

begin

phase *XorNode*

terminating *size*

begin

lemma *bin-xor-self-is-false*:

$\text{bin}[x \oplus x] = 0$

$\langle \text{proof} \rangle$

lemma *bin-xor-commute*:

$\text{bin}[x \oplus y] = \text{bin}[y \oplus x]$

$\langle \text{proof} \rangle$

lemma *bin-eliminate-redundant-false*:

$\text{bin}[x \oplus 0] = \text{bin}[x]$

$\langle \text{proof} \rangle$

lemma *val-xor-self-is-false*:
assumes $\text{val}[x \oplus x] \neq \text{UndefVal}$
shows $\text{val-to-bool } (\text{val}[x \oplus x]) = \text{False}$
 $\langle \text{proof} \rangle$

lemma *val-xor-self-is-false-2*:
assumes $(\text{val}[x \oplus x]) \neq \text{UndefVal}$
and $x = \text{IntVal } 32 \ v$
shows $\text{val}[x \oplus x] = \text{bool-to-val } \text{False}$
 $\langle \text{proof} \rangle$

lemma *val-xor-self-is-false-3*:
assumes $\text{val}[x \oplus x] \neq \text{UndefVal} \wedge x = \text{IntVal } 64 \ v$
shows $\text{val}[x \oplus x] = \text{IntVal } 64 \ 0$
 $\langle \text{proof} \rangle$

lemma *val-xor-commute*:
 $\text{val}[x \oplus y] = \text{val}[y \oplus x]$
 $\langle \text{proof} \rangle$

lemma *val-eliminate-redundant-false*:
assumes $x = \text{new-int } b \ v$
assumes $\text{val}[x \oplus (\text{bool-to-val } \text{False})] \neq \text{UndefVal}$
shows $\text{val}[x \oplus (\text{bool-to-val } \text{False})] = x$
 $\langle \text{proof} \rangle$

lemma *exp-xor-self-is-false*:
assumes $\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{default-stamp}$
shows $\text{exp}[x \oplus x] \geq \text{exp}[\text{false}]$
 $\langle \text{proof} \rangle$

lemma *exp-eliminate-redundant-false*:
shows $\text{exp}[x \oplus \text{false}] \geq \text{exp}[x]$
 $\langle \text{proof} \rangle$

Optimisations

optimization *XorSelfIsFalse*: $(x \oplus x) \mapsto \text{false}$ when
 $(\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{default-stamp})$
 $\langle \text{proof} \rangle$

optimization *XorShiftConstantRight*: $((\text{const } x) \oplus y) \mapsto y \oplus (\text{const } x)$ when
 $\neg(\text{is-ConstantExpr } y)$
 $\langle \text{proof} \rangle$

optimization *EliminateRedundantFalse*: $(x \oplus \text{false}) \mapsto x$

$\langle proof \rangle$

end

end

11 Conditional Elimination Phase

theory *ConditionalElimination*

imports

Semantics.IRTreeEvalThms

Proofs.Rewrites

Proofs.Bisimulation

begin

11.1 Individual Elimination Rules

The set of rules used for determining whether a condition $q1::'a$ implies another condition $q2::'a$ or its negation. These rules are used for conditional elimination.

inductive *impliesx* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* ($- \Rightarrow -$) **and**

impliesnot :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* ($- \Rightarrow \neg -$) **where**

q-imp-q:

$q \Rightarrow q$ |

eq-impliesnot-less:

$(\text{BinaryExpr BinIntegerEquals } x \ y) \Rightarrow \neg (\text{BinaryExpr BinIntegerLessThan } x \ y)$ |

eq-impliesnot-less-rev:

$(\text{BinaryExpr BinIntegerEquals } x \ y) \Rightarrow \neg (\text{BinaryExpr BinIntegerLessThan } y \ x)$ |

less-impliesnot-rev-less:

$(\text{BinaryExpr BinIntegerLessThan } x \ y) \Rightarrow \neg (\text{BinaryExpr BinIntegerLessThan } y \ x)$

|

less-impliesnot-eq:

$(\text{BinaryExpr BinIntegerLessThan } x \ y) \Rightarrow \neg (\text{BinaryExpr BinIntegerEquals } x \ y)$ |

less-impliesnot-eq-rev:

$(\text{BinaryExpr BinIntegerLessThan } x \ y) \Rightarrow \neg (\text{BinaryExpr BinIntegerEquals } y \ x)$ |

negate-true:

$\llbracket x \Rightarrow \neg y \rrbracket \Longrightarrow x \Rightarrow (\text{UnaryExpr UnaryLogicNegation } y)$ |

negate-false:

$\llbracket x \Rightarrow y \rrbracket \Longrightarrow x \Rightarrow \neg (\text{UnaryExpr UnaryLogicNegation } y)$

The relation $q1::IRExpr \Rightarrow q2::IRExpr$ indicates that the implication $(q1::bool) \longrightarrow (q2::bool)$ is known true (i.e. universally valid), and the relation $q1::IRExpr \Rightarrow \neg q2::IRExpr$ indicates that the implication $(q1::bool) \longrightarrow (q2::bool)$ is known false (i.e. $(q1::bool) \longrightarrow \neg (q2::bool)$ is universally valid). If neither

$q1::IRExpr \Rightarrow q2::IRExpr$ nor $q1::IRExpr \Rightarrow \neg q2::IRExpr$ then the status is unknown. Only the known true and known false cases can be used for conditional elimination.

fun *implies-valid* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* (**infix** \rightarrow 50) **where**
implies-valid *q1* *q2* =
 $(\forall m\ p\ v1\ v2. ([m, p] \vdash q1 \mapsto v1) \wedge ([m, p] \vdash q2 \mapsto v2) \longrightarrow$
 $(val\text{-}to\text{-}bool\ v1 \longrightarrow val\text{-}to\text{-}bool\ v2))$

fun *impliesnot-valid* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* (**infix** \nrightarrow 50) **where**
impliesnot-valid *q1* *q2* =
 $(\forall m\ p\ v1\ v2. ([m, p] \vdash q1 \mapsto v1) \wedge ([m, p] \vdash q2 \mapsto v2) \longrightarrow$
 $(val\text{-}to\text{-}bool\ v1 \longrightarrow \neg val\text{-}to\text{-}bool\ v2))$

The relation $(q1::IRExpr) \rightarrow (q2::IRExpr)$ means $(q1::bool) \longrightarrow (q2::bool)$ is universally valid, and the relation $(q1::IRExpr) \nrightarrow (q2::IRExpr)$ means $(q1::bool) \longrightarrow \neg (q2::bool)$ is universally valid.

lemma *eq-impliesnot-less-helper*:
 $v1 = v2 \longrightarrow \neg(int\text{-}signed\text{-}value\ b\ v1 < int\text{-}signed\text{-}value\ b\ v2)$
 $\langle proof \rangle$

lemma *eq-impliesnot-less-val*:
 $val\text{-}to\text{-}bool(intval\text{-}equals\ v1\ v2) \longrightarrow \neg val\text{-}to\text{-}bool(intval\text{-}less\text{-}than\ v1\ v2)$
 $\langle proof \rangle$

lemma *eq-impliesnot-less-rev-val*:
 $val\text{-}to\text{-}bool(intval\text{-}equals\ v1\ v2) \longrightarrow \neg val\text{-}to\text{-}bool(intval\text{-}less\text{-}than\ v2\ v1)$
 $\langle proof \rangle$

lemma *less-impliesnot-rev-less-val*:
 $val\text{-}to\text{-}bool(intval\text{-}less\text{-}than\ v1\ v2) \longrightarrow \neg val\text{-}to\text{-}bool(intval\text{-}less\text{-}than\ v2\ v1)$
 $\langle proof \rangle$

lemma *less-impliesnot-eq-val*:
 $val\text{-}to\text{-}bool(intval\text{-}less\text{-}than\ v1\ v2) \longrightarrow \neg val\text{-}to\text{-}bool(intval\text{-}equals\ v1\ v2)$
 $\langle proof \rangle$

lemma *logic-negate-type*:
assumes $[m, p] \vdash UnaryExpr\ UnaryLogicNegation\ x \mapsto v$
shows $\exists b\ v2. [m, p] \vdash x \mapsto IntVal\ b\ v2$
 $\langle proof \rangle$

lemma *intval-logic-negation-inverse*:
assumes $b > 0$
assumes $x = IntVal\ b\ v$
shows $val\text{-}to\text{-}bool\ (intval\text{-}logic\text{-}negation\ x) \longleftrightarrow \neg(val\text{-}to\text{-}bool\ x)$
 $\langle proof \rangle$

lemma *logic-negation-relation-tree*:


```

assumes  $[m, p] \vdash y \mapsto val$ 
assumes  $[m, p] \vdash \text{UnaryExpr } \text{UnaryLogicNegation } y \mapsto invval$ 
shows  $val\text{-to-bool } val \longleftrightarrow \neg(val\text{-to-bool } invval)$ 
 $\langle proof \rangle$ 

```

The following theorem shows that the known true/false rules are valid.

theorem *implies-impliesnot-valid:*

```

shows  $((q1 \Rightarrow q2) \longrightarrow (q1 \mapsto q2)) \wedge$ 
 $((q1 \Rightarrow \neg q2) \longrightarrow (q1 \mapsto \neg q2))$ 
(is  $(?imp \longrightarrow ?val) \wedge (?notimp \longrightarrow ?notval))$ 
 $\langle proof \rangle$ 

```

We introduce a type *TriState::'a* (as in the GraalVM compiler) to represent when static analysis can tell us information about the value of a Boolean expression. If *Unknown::'a* then no information can be inferred and if *Known-True::'a*/*KnownFalse::'a* one can infer the expression is always true/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:

$\llbracket g \vdash x \ \& \ (\text{kind } g \ y) \hookrightarrow \text{KnownTrue} \rrbracket \Longrightarrow g \vdash x \ \& \ (\text{LogicNegationNode } y) \hookrightarrow \text{KnownFalse} \mid$

negate-true:

$\llbracket g \vdash x \ \& \ (\text{kind } g \ y) \hookrightarrow \text{KnownFalse} \rrbracket \Longrightarrow g \vdash x \ \& \ (\text{LogicNegationNode } y) \hookrightarrow \text{KnownTrue}$

Total relation over partial implies relation

inductive *condition-implies* :: *IRGraph* \Rightarrow *IRNode* \Rightarrow *IRNode* \Rightarrow *TriState* \Rightarrow *bool*

$(- \vdash - \ \& \ - \hookrightarrow -)$ **for** *g* **where**

$\llbracket \neg(g \vdash a \ \& \ b \hookrightarrow \text{imp}) \rrbracket \Longrightarrow (g \vdash a \ \& \ b \hookrightarrow \text{Unknown}) \mid$

$$\llbracket (g \vdash a \ \& \ b \hookrightarrow \text{imp}) \rrbracket \implies (g \vdash a \ \& \ b \multimap \text{imp})$$

inductive *implies-tree* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* \Rightarrow *bool*
 (- & - \hookrightarrow -) **where**
eq-imp-less:
 (*BinaryExpr* *BinIntegerEquals* *x y*) & (*BinaryExpr* *BinIntegerLessThan* *x y*) \hookrightarrow
False |
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:
 $\llbracket x \ \& \ y \hookrightarrow \text{True} \rrbracket \implies x \ \& \ (\text{UnaryExpr } \text{UnaryLogicNegation } y) \hookrightarrow \text{False}$ |
negate-true:
 $\llbracket x \ \& \ y \hookrightarrow \text{False} \rrbracket \implies x \ \& \ (\text{UnaryExpr } \text{UnaryLogicNegation } y) \hookrightarrow \text{True}$

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

lemma *logic-negation-relation*:
assumes $[g, m, p] \vdash y \mapsto \text{val}$
assumes $\text{kind } g \text{ neg} = \text{LogicNegationNode } y$
assumes $[g, m, p] \vdash \text{neg} \mapsto \text{invval}$
assumes $\text{invval} \neq \text{UndefVal}$
shows $\text{val-to-bool } \text{val} \longleftrightarrow \neg(\text{val-to-bool } \text{invval})$
<proof>

lemma *implies-valid*:
assumes $x \ \& \ y \hookrightarrow \text{imp}$
assumes $[m, p] \vdash x \mapsto v1$
assumes $[m, p] \vdash y \mapsto v2$
shows $(\text{imp} \longrightarrow (\text{val-to-bool } v1 \longrightarrow \text{val-to-bool } v2)) \wedge$
 $(\neg \text{imp} \longrightarrow (\text{val-to-bool } v1 \longrightarrow \neg(\text{val-to-bool } v2)))$
(is $(?TP \longrightarrow ?TC) \wedge (?FP \longrightarrow ?FC)$ **)**
<proof>

lemma *implies-true-valid*:
assumes $x \ \& \ y \hookrightarrow \text{imp}$

```

assumes imp
assumes  $[m, p] \vdash x \mapsto v1$ 
assumes  $[m, p] \vdash y \mapsto v2$ 
shows  $val\text{-}to\text{-}bool\ v1 \longrightarrow val\text{-}to\text{-}bool\ v2$ 
 $\langle proof \rangle$ 

```

```

lemma implies-false-valid:
assumes  $x \ \& \ y \hookrightarrow imp$ 
assumes  $\neg imp$ 
assumes  $[m, p] \vdash x \mapsto v1$ 
assumes  $[m, p] \vdash y \mapsto v2$ 
shows  $val\text{-}to\text{-}bool\ v1 \longrightarrow \neg(val\text{-}to\text{-}bool\ v2)$ 
 $\langle proof \rangle$ 

```

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
   $\llbracket alwaysDistinct\ (stamps\ x)\ (stamps\ y) \rrbracket$ 
     $\implies tryFold\ (IntegerEqualsNode\ x\ y)\ stamps\ False \mid$ 
   $\llbracket neverDistinct\ (stamps\ x)\ (stamps\ y) \rrbracket$ 
     $\implies tryFold\ (IntegerEqualsNode\ x\ y)\ stamps\ True \mid$ 
   $\llbracket is\text{-}IntegerStamp\ (stamps\ x);$ 
     $is\text{-}IntegerStamp\ (stamps\ y);$ 
     $stpi\text{-}upper\ (stamps\ x) < stpi\text{-}lower\ (stamps\ y) \rrbracket$ 
     $\implies tryFold\ (IntegerLessThanNode\ x\ y)\ stamps\ True \mid$ 
   $\llbracket is\text{-}IntegerStamp\ (stamps\ x);$ 
     $is\text{-}IntegerStamp\ (stamps\ y);$ 
     $stpi\text{-}lower\ (stamps\ x) \geq stpi\text{-}upper\ (stamps\ y) \rrbracket$ 
     $\implies tryFold\ (IntegerLessThanNode\ x\ y)\ stamps\ False$ 

```

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

```

lemma
assumes  $kind\ g\ nid = IntegerEqualsNode\ x\ y$ 
assumes  $[g, m, p] \vdash nid \mapsto v$ 
assumes  $([g, m, p] \vdash x \mapsto xval) \wedge ([g, m, p] \vdash y \mapsto yval)$ 
shows  $val\text{-}to\text{-}bool\ (intval\text{-}equals\ xval\ yval) \longleftrightarrow v = IntVal\ 32\ 1$ 
 $\langle proof \rangle$ 

```

```

lemma tryFoldIntegerEqualsAlwaysDistinct:
assumes wf-stamp g stamps
assumes  $kind\ g\ nid = (IntegerEqualsNode\ x\ y)$ 
assumes  $[g, m, p] \vdash nid \mapsto v$ 

```

assumes *alwaysDistinct* (*stamps x*) (*stamps y*)
shows $v = \text{IntVal } 32 \ 0$
 <proof>

lemma *tryFoldIntegerEqualsNeverDistinct*:
assumes *wf-stamp g stamps*
assumes $\text{kind } g \ nid = (\text{IntegerEqualsNode } x \ y)$
assumes $[g, m, p] \vdash nid \mapsto v$
assumes *neverDistinct* (*stamps x*) (*stamps y*)
shows $v = \text{IntVal } 32 \ 1$
 <proof>

lemma *tryFoldIntegerLessThanTrue*:
assumes *wf-stamp g stamps*
assumes $\text{kind } g \ nid = (\text{IntegerLessThanNode } x \ y)$
assumes $[g, m, p] \vdash nid \mapsto v$
assumes $\text{stpi-upper } (\text{stamps } x) < \text{stpi-lower } (\text{stamps } y)$
shows $v = \text{IntVal } 32 \ 1$
 <proof>

lemma *tryFoldIntegerLessThanFalse*:
assumes *wf-stamp g stamps*
assumes $\text{kind } g \ nid = (\text{IntegerLessThanNode } x \ y)$
assumes $[g, m, p] \vdash nid \mapsto v$
assumes $\text{stpi-lower } (\text{stamps } x) \geq \text{stpi-upper } (\text{stamps } y)$
shows $v = \text{IntVal } 32 \ 0$
 <proof>

theorem *tryFoldProofTrue*:
assumes *wf-stamp g stamps*
assumes $\text{tryFold } (\text{kind } g \ nid) \ \text{stamps } \text{True}$
assumes $[g, m, p] \vdash nid \mapsto v$
shows *val-to-bool v*
 <proof>

theorem *tryFoldProofFalse*:
assumes *wf-stamp g stamps*
assumes $\text{tryFold } (\text{kind } g \ nid) \ \text{stamps } \text{False}$
assumes $[g, m, p] \vdash nid \mapsto v$
shows $\neg(\text{val-to-bool } v)$
 <proof>

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

Perform conditional elimination rewrites on the graph for a particular node.
 In order to determine conditional eliminations appropriately the rule needs

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

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

inductive *ConditionalEliminationStep* ::

IRExpr set \Rightarrow (*ID* \Rightarrow *Stamp*) \Rightarrow *IRGraph* \Rightarrow *ID* \Rightarrow *IRGraph* \Rightarrow *bool* **where**
impliesTrue:

$\llbracket \text{kind } g \text{ ifcond} = (\text{IfNode } cid \text{ } t \text{ } f);$
 $g \vdash cid \simeq cond;$
 $\exists ce \in conds . (ce \Rightarrow cond);$
 $g' = \text{constantCondition } True \text{ ifcond } (kind \text{ } g \text{ ifcond}) \text{ } g$
 $\rrbracket \Longrightarrow \text{ConditionalEliminationStep } conds \text{ stamps } g \text{ ifcond } g' \mid$

impliesFalse:

$\llbracket \text{kind } g \text{ ifcond} = (\text{IfNode } cid \text{ } t \text{ } f);$
 $g \vdash cid \simeq cond;$
 $\exists ce \in conds . (ce \Rightarrow \neg cond);$
 $g' = \text{constantCondition } False \text{ ifcond } (kind \text{ } g \text{ ifcond}) \text{ } g$
 $\rrbracket \Longrightarrow \text{ConditionalEliminationStep } conds \text{ stamps } g \text{ ifcond } g' \mid$

tryFoldTrue:

$\llbracket \text{kind } g \text{ ifcond} = (\text{IfNode } cid \text{ } t \text{ } f);$
 $cond = kind \text{ } g \text{ } cid;$
 $\text{tryFold } (kind \text{ } g \text{ } cid) \text{ stamps } True;$
 $g' = \text{constantCondition } True \text{ ifcond } (kind \text{ } g \text{ ifcond}) \text{ } g$
 $\rrbracket \Longrightarrow \text{ConditionalEliminationStep } conds \text{ stamps } g \text{ ifcond } g' \mid$

tryFoldFalse:

$\llbracket \text{kind } g \text{ ifcond} = (\text{IfNode } cid \text{ } t \text{ } f);$
 $cond = kind \text{ } g \text{ } cid;$
 $\text{tryFold } (kind \text{ } g \text{ } cid) \text{ stamps } False;$
 $g' = \text{constantCondition } False \text{ ifcond } (kind \text{ } g \text{ ifcond}) \text{ } g$
 $\rrbracket \Longrightarrow \text{ConditionalEliminationStep } conds \text{ stamps } g \text{ ifcond } g' \mid$

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *ConditionalEliminationStep*
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

```

[[kind g nid = BeginNode nid';

```

```

  nid ∉ seen;
  seen' = {nid} ∪ seen;

```

```

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

```

```

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

```

```

  flow' = registerNewCondition g c (hdOr flow (stamp g))]]
⇒ Step g (nid, seen, conds, flow) (Some (nid', seen', conds', flow' # flow)) |

```

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

```

[[kind g nid = EndNode;

```

```

  nid ∉ seen;
  seen' = {nid} ∪ seen;

```

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

$conds' = \text{tl } conds;$

$flow' = \text{tl } flow$

$\implies \text{Step } g \ (nid, \text{seen}, \text{conds}, \text{flow}) \ (\text{Some } (nid', \text{seen}', \text{conds}', \text{flow}')) \mid$

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

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

$\neg(\text{is-BEGINNode } (kind \ g \ nid));$

$nid \notin \text{seen};$

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

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

$\implies \text{Step } g \ (nid, \text{seen}, \text{conds}, \text{flow}) \ (\text{Some } (nid', \text{seen}', \text{conds}, \text{flow})) \mid$

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

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

$\neg(\text{is-BEGINNode } (kind \ g \ nid));$

$nid \notin \text{seen};$

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

$\text{None} = \text{nextEdge } \text{seen}' \ nid \ g$

$\implies \text{Step } g \ (nid, \text{seen}, \text{conds}, \text{flow}) \ \text{None} \mid$

— We've already seen this node, give back None

$\llbracket nid \in \text{seen} \rrbracket \implies \text{Step } g \ (nid, \text{seen}, \text{conds}, \text{flow}) \ \text{None}$

code-pred ($modes: i \Rightarrow i \Rightarrow o \Rightarrow bool$) $\text{Step } \langle \text{proof} \rangle$

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`

$:: \text{IRGraph} \Rightarrow (\text{ID} \times \text{Seen} \times \text{Conditions} \times \text{StampFlow}) \Rightarrow \text{IRGraph} \Rightarrow bool$

where

— Can do a step and optimise for the current node

$\llbracket \text{Step } g \ (nid, \text{seen}, \text{conds}, \text{flow}) \ (\text{Some } (nid', \text{seen}', \text{conds}', \text{flow}'));$

$\text{ConditionalEliminationStep } (\text{set } \text{conds}) \ (\text{hdOr } \text{flow } (\text{stamp } g)) \ g \ nid \ g';$

$\text{ConditionalEliminationPhase } g' \ (nid', \text{seen}', \text{conds}', \text{flow}') \ g' \rrbracket$

$\implies \text{ConditionalEliminationPhase } g \ (nid, \text{seen}, \text{conds}, \text{flow}) \ g'' \mid$

— Can do a step, matches whether optimised or not causing non-determinism We need to find a way to negate `ConditionalEliminationStep`


```

[[Step g (nid, seen, conds, flow) (Some (nid', seen', conds', flow'))];

  ConditionalEliminationPhase g (nid', seen', conds', flow') g]]
⇒ ConditionalEliminationPhase g (nid, seen, conds, flow) g' |

— Can't do a step but there is a predecessor we can backtrace to
[[Step g (nid, seen, conds, flow) None;
  Some nid' = pred g nid;
  seen' = {nid} ∪ seen;
  ConditionalEliminationPhase g (nid', seen', conds, flow) g]]
⇒ ConditionalEliminationPhase g (nid, seen, conds, flow) g' |

— Can't do a step and have no predecessors so terminate
[[Step g (nid, seen, conds, flow) None;
  None = pred g nid]]
⇒ ConditionalEliminationPhase g (nid, seen, conds, flow) g

code-pred (modes: i ⇒ i ⇒ o ⇒ bool) ConditionalEliminationPhase ⟨proof⟩

definition runConditionalElimination :: IRGraph ⇒ IRGraph where
  runConditionalElimination g =
    (Predicate.the (ConditionalEliminationPhase-i-i-o g (0, {}, ([], []))))

end

```