

Verifying term graph optimizations using Isabelle/HOL

Isabelle/HOL Theories

December 9, 2022

Abstract

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

Contents

| | | |
|----------|--|-----------|
| 1 | Operator Semantics | 3 |
| 1.1 | Arithmetic Operators | 6 |
| 1.2 | Bitwise Operators | 7 |
| 1.3 | Comparison Operators | 7 |
| 1.4 | Narrowing and Widening Operators | 8 |
| 1.5 | Bit-Shifting Operators | 9 |
| 1.5.1 | Examples of Narrowing / Widening Functions | 10 |
| 1.6 | Fixed-width Word Theories | 12 |
| 1.6.1 | Support Lemmas for Upper/Lower Bounds | 12 |
| 1.6.2 | Support lemmas for take bit and signed take bit. | 16 |
| 2 | Stamp Typing | 17 |
| 3 | Graph Representation | 22 |
| 3.1 | IR Graph Nodes | 22 |
| 3.2 | IR Graph Node Hierarchy | 30 |
| 3.3 | IR Graph Type | 37 |
| 3.3.1 | Example Graphs | 41 |
| 4 | java.lang.Long | 42 |
| 4.1 | Long.numberOfLeadingZeros | 43 |
| 4.2 | Long.numberOfTrailingZeros | 43 |
| 4.3 | Long.bitCount | 44 |
| 4.4 | Long.zeroCount | 44 |
| 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 | 54 |
| 5.6 | Data-flow Tree Theorems | 55 |
| 5.6.1 | Deterministic Data-flow Evaluation | 55 |
| 5.6.2 | Typing Properties for Integer Evaluation Functions | 56 |
| 5.6.3 | Evaluation Results are Valid | 58 |
| 5.6.4 | Example Data-flow Optimisations | 60 |
| 5.6.5 | Monotonicity of Expression Refinement | 60 |
| 5.7 | Unfolding rules for evaltree quadruples down to bin-eval level | 61 |
| 5.8 | Lemmas about <i>new_int</i> and integer eval results. | 62 |

| | | |
|----------|--|------------|
| 6 | Tree to Graph | 67 |
| 6.1 | Subgraph to Data-flow Tree | 67 |
| 6.2 | Data-flow Tree to Subgraph | 71 |
| 6.3 | Lift Data-flow Tree Semantics | 76 |
| 6.4 | Graph Refinement | 76 |
| 6.5 | Maximal Sharing | 76 |
| 6.6 | Formedness Properties | 76 |
| 6.7 | Dynamic Frames | 78 |
| 6.8 | Tree to Graph Theorems | 90 |
| 6.8.1 | Extraction and Evaluation of Expression Trees is De- terministic. | 91 |
| 6.8.2 | Monotonicity of Graph Refinement | 97 |
| 6.8.3 | Lift Data-flow Tree Refinement to Graph Refinement . | 100 |
| 6.8.4 | Term Graph Reconstruction | 116 |
| 6.8.5 | Data-flow Tree to Subgraph Preserves Maximal Sharing | 124 |
| 7 | Control-flow Semantics | 138 |
| 7.1 | Object Heap | 138 |
| 7.2 | Intraprocedural Semantics | 139 |
| 7.3 | Interprocedural Semantics | 141 |
| 7.4 | Big-step Execution | 143 |
| 7.4.1 | Heap Testing | 144 |
| 7.5 | Control-flow Semantics Theorems | 144 |
| 7.5.1 | Control-flow Step is Deterministic | 145 |
| 7.6 | Evaluation Stamp Theorems | 149 |
| 7.6.1 | Support Lemmas for Integer Stamps and Associated IntVal values | 151 |
| 7.6.2 | Validity of all Unary Operators | 153 |
| 7.6.3 | Support Lemmas for Binary Operators | 155 |
| 7.6.4 | Validity of Stamp Meet and Join Operators | 158 |
| 7.6.5 | Validity of conditional expressions | 159 |
| 7.6.6 | Validity of Whole Expression Tree Evaluation | 159 |
| 8 | Optimization DSL | 161 |
| 8.1 | Markup | 161 |
| 8.1.1 | Expression Markup | 162 |
| 8.1.2 | Value Markup | 163 |
| 8.1.3 | Word Markup | 164 |
| 8.2 | Optimization Phases | 165 |
| 8.3 | Canonicalization DSL | 165 |
| 8.3.1 | Semantic Preservation Obligation | 168 |
| 8.3.2 | Termination Obligation | 168 |
| 8.3.3 | Standard Termination Measure | 168 |
| 8.3.4 | Automated Tactics | 169 |

| | | |
|-----------|--|------------|
| 9 | Canonicalization Optimizations | 169 |
| 9.1 | AddNode Phase | 171 |
| 9.2 | AndNode Phase | 174 |
| 9.3 | Experimental AndNode Phase | 179 |
| 9.4 | ConditionalNode Phase | 190 |
| 9.5 | MulNode Phase | 194 |
| 9.6 | NotNode Phase | 204 |
| 9.7 | OrNode Phase | 205 |
| 9.8 | SubNode Phase | 208 |
| 9.9 | XorNode Phase | 213 |
| 10 | Verifying term graph optimizations using Isabelle/HOL | 215 |
| 10.1 | Markup syntax for common operations | 216 |
| 10.2 | Representing canonicalization optimizations | 216 |
| 10.3 | Representing terms | 219 |
| 10.4 | Term semantics | 220 |

1 Operator Semantics

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

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

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

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

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

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

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

```
experiment begin
```

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

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

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

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

```
experiment begin
```

Option 3: explicit type stored with each integer value.

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

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

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

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

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

end

Option 4: int64 with the number of significant bits.

type-synonym *iwidth* = *nat*

type-synonym *objref* = *nat option*

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

IntVal *iwidth* *int64* |

ObjRef *objref* |
ObjStr *string*

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

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

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

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

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

bit-bounds *bits* = (((2 ^{*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*)
by *simp*

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

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

1.1 Arithmetic Operators

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

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

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

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

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

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

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

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

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



```

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

```

```

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

```

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

```

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

```

1.2 Bitwise Operators

```

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

```

```

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

```

```

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

```

```

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

```

1.3 Comparison Operators

```

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

```

```

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

```

```

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

```

```

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

```

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

1.4 Narrowing and Widening Operators

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

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

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

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

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

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

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

lemma *intval-narrow-ok*:

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

lemma *intval-sign-extend-ok*:

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

```

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

```

```

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

```

1.5 Bit-Shifting Operators

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

value (128 :: 8 word) >> 2

```

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

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

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

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

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

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

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

1.5.1 Examples of Narrowing / Widening Functions

experiment begin

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

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

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

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

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

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

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

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

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

end

experiment begin

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

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

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

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

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

experiment begin

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

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

experiment begin

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

lemma *intval-add-sym*:

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

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

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

```

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

end

```

1.6 Fixed-width Word Theories

```

theory ValueThms
  imports Values
begin

```

1.6.1 Support Lemmas for Upper/Lower Bounds

```

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

```

```

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

```

```

lemma lower-bounds-equiv:
  assumes 0 < N
  shows -(((2::int) ^ (N-1))) = (2::int) ^ N div 2 * - 1
  by (simp add: assms int-power-div-base)

```

```

lemma upper-bounds-equiv:
  assumes 0 < N
  shows (2::int) ^ (N-1) = (2::int) ^ N div 2
  by (simp add: assms int-power-div-base)

```

Some min/max bounds for 64-bit words

```

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

```

```

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

```

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

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

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

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

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

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

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

A `bit_bounds` version of the above lemma.

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

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

```

assumes 0 < n
assumes val = sint(signed-take-bit (n - 1) ival)
shows fst (bit-bounds n) ≤ val ∧ val ≤ snd (bit-bounds n)
using assms signed-take-bit-bounds
by (metis size64 word-size)

```

```

lemma int-signed-value-bounds:
  assumes b1 ≤ 64
  assumes 0 < b1
  shows fst (bit-bounds b1) ≤ int-signed-value b1 v2 ∧
    int-signed-value b1 v2 ≤ snd (bit-bounds b1)
  using assms int-signed-value.simps signed-take-bit-bounds64 by blast

```

```

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

```

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

```

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

```

```

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

```

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

```

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

```


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

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

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

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

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

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

lemma *scast-bigger-min-bound*:
assumes $(\text{result} :: 'b :: \text{len word}) = \text{scast}(v :: 'a :: \text{len word})$

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

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

Results about *new_int*.

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

1.6.2 Support lemmas for take bit and signed take bit.

Lemmas for removing redundant take_bit wrappers.

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

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

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

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

```

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

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

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

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

end

```

2 Stamp Typing

```

theory Stamp
  imports Values
begin

```

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

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

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

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

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

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

  is-stamp-empty x = False
```

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

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

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

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

— A stamp which includes the full range of the type

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

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

```
fun is-stamp-unrestricted :: Stamp  $\Rightarrow$  bool where
  is-stamp-unrestricted s = (s = unrestricted-stamp s)
```

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

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

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

— Calculate the meet stamp of two stamps

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

```

meet (KlassPointerStamp nn1 an1) (KlassPointerStamp nn2 an2) = (
  KlassPointerStamp (nn1  $\wedge$  nn2) (an1  $\wedge$  an2)
) |
meet (MethodCountersPointerStamp nn1 an1) (MethodCountersPointerStamp
nn2 an2) = (
  MethodCountersPointerStamp (nn1  $\wedge$  nn2) (an1  $\wedge$  an2)
) |
meet (MethodPointersStamp nn1 an1) (MethodPointersStamp nn2 an2) = (
  MethodPointersStamp (nn1  $\wedge$  nn2) (an1  $\wedge$  an2)
) |
meet s1 s2 = IllegalStamp

```

— Calculate the join stamp of two stamps

```

fun join :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  Stamp where
  join VoidStamp VoidStamp = VoidStamp |
  join (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1  $\neq$  b2 then IllegalStamp else
    (IntegerStamp b1 (max l1 l2) (min u1 u2))
  ) |
  join (KlassPointerStamp nn1 an1) (KlassPointerStamp nn2 an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (KlassPointerStamp nn1 an1))
    else (KlassPointerStamp (nn1  $\vee$  nn2) (an1  $\vee$  an2))
  ) |
  join (MethodCountersPointerStamp nn1 an1) (MethodCountersPointerStamp nn2
an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (MethodCountersPointerStamp nn1 an1))
    else (MethodCountersPointerStamp (nn1  $\vee$  nn2) (an1  $\vee$  an2))
  ) |
  join (MethodPointersStamp nn1 an1) (MethodPointersStamp nn2 an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (MethodPointersStamp nn1 an1))
    else (MethodPointersStamp (nn1  $\vee$  nn2) (an1  $\vee$  an2))
  ) |
  join s1 s2 = IllegalStamp

```

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

```

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

```

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

```

fun alwaysDistinct :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  bool where

```

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

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

fun *neverDistinct* :: *Stamp* \Rightarrow *Stamp* \Rightarrow *bool* **where**
neverDistinct stamp1 stamp2 = (*asConstant stamp1* = *asConstant stamp2* \wedge
asConstant stamp1 \neq *UndefVal*)

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

constantAsStamp - = IllegalStamp

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

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

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

definition *wf-value* :: *Value* \Rightarrow *bool* **where**

wf-value v = valid-value v (constantAsStamp v)

lemma *unfold-wf-value[simp]*:

wf-value v \Longrightarrow valid-value v (constantAsStamp v)

using *wf-value-def* **by** *auto*

fun *compatible* :: *Stamp* \Rightarrow *Stamp* \Rightarrow *bool* **where**

compatible (IntegerStamp b1 lo1 hi1) (IntegerStamp b2 lo2 hi2) =
 (*b1 = b2 \wedge valid-stamp (IntegerStamp b1 lo1 hi1) \wedge valid-stamp (IntegerStamp b2 lo2 hi2)*) |
compatible (VoidStamp) (VoidStamp) = True |
compatible - - = False

fun *stamp-under* :: *Stamp* \Rightarrow *Stamp* \Rightarrow *bool* **where**

stamp-under (IntegerStamp b1 lo1 hi1) (IntegerStamp b2 lo2 hi2) = (*b1 = b2 \wedge hi1 < lo2*) |
stamp-under - - = False

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

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

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

3 Graph Representation

3.1 IR Graph Nodes

theory *IRNodes*
imports
Values
begin

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

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

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

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

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

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

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

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

```

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

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

| StartNode (ir-stateAfter-opt: INPUT-STATE option) (ir-next: SUCC)
| StoreFieldNode (ir-nid: ID) (ir-field: string) (ir-value: INPUT) (ir-stateAfter-opt:
INPUT-STATE option) (ir-object-opt: INPUT option) (ir-next: SUCC)
| SubNode (ir-x: INPUT) (ir-y: INPUT)
| UnsignedRightShiftNode (ir-x: INPUT) (ir-y: INPUT)
| UnwindNode (ir-exception: INPUT)
| ValuePhiNode (ir-nid: ID) (ir-values: INPUT list) (ir-merge: INPUT-ASSOC)

| ValueProxyNode (ir-value: INPUT) (ir-loopExit: INPUT-ASSOC)
| XorNode (ir-x: INPUT) (ir-y: INPUT)
| ZeroExtendNode (ir-inputBits: nat) (ir-resultBits: nat) (ir-value: INPUT)
| NoNode

| RefNode (ir-ref:ID)

```

```

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

```

```

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

```

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

```

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

```

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

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

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

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

fun *successors-of* :: *IRNode* ⇒ *ID list* **where**

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

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

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

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

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

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

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

```

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

end

```

3.2 IR Graph Node Hierarchy

```

theory IRNodeHierarchy
imports IRNodes
begin

```

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

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

These functions have been automatically generated from the compiler.

```

fun is-EndNode :: IRNode  $\Rightarrow$  bool where
  is-EndNode EndNode = True |
  is-EndNode - = False

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

is-ValueNode *n* = ((*is-CallTargetNode* *n*) ∨ (*is-FixedNode* *n*) ∨ (*is-FloatingNode* *n*))

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

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

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

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

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

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

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

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

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

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

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

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

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

```

fun is-GuardingNode :: IRNode  $\Rightarrow$  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))  $\vee$ 
```

$((is-AddNode\ n1) \wedge (is-AddNode\ n2)) \vee$
 $((is-AndNode\ n1) \wedge (is-AndNode\ n2)) \vee$
 $((is-BEGINNode\ n1) \wedge (is-BEGINNode\ n2)) \vee$
 $((is-BytecodeExceptionNode\ n1) \wedge (is-BytecodeExceptionNode\ n2)) \vee$
 $((is-ConditionalNode\ n1) \wedge (is-ConditionalNode\ n2)) \vee$
 $((is-ConstantNode\ n1) \wedge (is-ConstantNode\ n2)) \vee$
 $((is-DynamicNewArrayNode\ n1) \wedge (is-DynamicNewArrayNode\ n2)) \vee$
 $((is-EndNode\ n1) \wedge (is-EndNode\ n2)) \vee$
 $((is-ExceptionObjectNode\ n1) \wedge (is-ExceptionObjectNode\ n2)) \vee$
 $((is-FrameState\ n1) \wedge (is-FrameState\ n2)) \vee$
 $((is-IfNode\ n1) \wedge (is-IfNode\ n2)) \vee$
 $((is-IntegerBelowNode\ n1) \wedge (is-IntegerBelowNode\ n2)) \vee$
 $((is-IntegerEqualsNode\ n1) \wedge (is-IntegerEqualsNode\ n2)) \vee$
 $((is-IntegerLessThanNode\ n1) \wedge (is-IntegerLessThanNode\ n2)) \vee$
 $((is-InvokeNode\ n1) \wedge (is-InvokeNode\ n2)) \vee$
 $((is-InvokeWithExceptionNode\ n1) \wedge (is-InvokeWithExceptionNode\ n2)) \vee$
 $((is-IsNullNode\ n1) \wedge (is-IsNullNode\ n2)) \vee$
 $((is-KillingBeginNode\ n1) \wedge (is-KillingBeginNode\ n2)) \vee$
 $((is-LeftShiftNode\ n1) \wedge (is-LeftShiftNode\ n2)) \vee$
 $((is-LoadFieldNode\ n1) \wedge (is-LoadFieldNode\ n2)) \vee$
 $((is-LogicNegationNode\ n1) \wedge (is-LogicNegationNode\ n2)) \vee$
 $((is-LoopBeginNode\ n1) \wedge (is-LoopBeginNode\ n2)) \vee$
 $((is-LoopEndNode\ n1) \wedge (is-LoopEndNode\ n2)) \vee$
 $((is-LoopExitNode\ n1) \wedge (is-LoopExitNode\ n2)) \vee$
 $((is-MergeNode\ n1) \wedge (is-MergeNode\ n2)) \vee$
 $((is-MethodCallTargetNode\ n1) \wedge (is-MethodCallTargetNode\ n2)) \vee$
 $((is-MulNode\ n1) \wedge (is-MulNode\ n2)) \vee$
 $((is-NarrowNode\ n1) \wedge (is-NarrowNode\ n2)) \vee$
 $((is-NegateNode\ n1) \wedge (is-NegateNode\ n2)) \vee$
 $((is-NewArrayNode\ n1) \wedge (is-NewArrayNode\ n2)) \vee$
 $((is-NewInstanceNode\ n1) \wedge (is-NewInstanceNode\ n2)) \vee$
 $((is-NotNode\ n1) \wedge (is-NotNode\ n2)) \vee$
 $((is-OrNode\ n1) \wedge (is-OrNode\ n2)) \vee$
 $((is-ParameterNode\ n1) \wedge (is-ParameterNode\ n2)) \vee$
 $((is-PiNode\ n1) \wedge (is-PiNode\ n2)) \vee$
 $((is-ReturnNode\ n1) \wedge (is-ReturnNode\ n2)) \vee$
 $((is-RightShiftNode\ n1) \wedge (is-RightShiftNode\ n2)) \vee$
 $((is-ShortCircuitOrNode\ n1) \wedge (is-ShortCircuitOrNode\ n2)) \vee$
 $((is-SignedDivNode\ n1) \wedge (is-SignedDivNode\ n2)) \vee$
 $((is-SignedRemNode\ n1) \wedge (is-SignedRemNode\ n2)) \vee$
 $((is-SignExtendNode\ n1) \wedge (is-SignExtendNode\ n2)) \vee$
 $((is-StartNode\ n1) \wedge (is-StartNode\ n2)) \vee$
 $((is-StoreFieldNode\ n1) \wedge (is-StoreFieldNode\ n2)) \vee$
 $((is-SubNode\ n1) \wedge (is-SubNode\ n2)) \vee$
 $((is-UnsignedRightShiftNode\ n1) \wedge (is-UnsignedRightShiftNode\ n2)) \vee$
 $((is-UnwindNode\ n1) \wedge (is-UnwindNode\ n2)) \vee$
 $((is-ValuePhiNode\ n1) \wedge (is-ValuePhiNode\ n2)) \vee$
 $((is-ValueProxyNode\ n1) \wedge (is-ValueProxyNode\ n2)) \vee$
 $((is-XorNode\ n1) \wedge (is-XorNode\ n2)) \vee$

((*is-ZeroExtendNode* *n1*) \wedge (*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  $\rightarrow$  (IRNode  $\times$  Stamp) . finite (dom g)}
proof –
  have finite(dom(Map.empty))  $\wedge$  ran Map.empty = {} by auto
  then show ?thesis
    by fastforce
qed
```

setup-lifting *type-definition-IRGraph*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

code-datatype *irgraph*

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

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

lemma *dom-eq*:
assumes $\forall x \in \text{set } xs. \text{fst } (\text{snd } x) \neq \text{NoNode}$
shows *filter-none* (*map-of* *xs*) = *dom* (*map-of* *xs*)
unfolding *filter-none.simps* **using** *assms map-of-SomeD*
by *fastforce*

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

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

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

— Get the inputs set of a given node ID

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

— Get the successor set of a given node ID

fun *succ* :: *IRGraph* \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 = ($\bigcup i \in \text{ids } g. \{(i,j) | j \in (\text{inputs } g i)\}$)

— 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 = $\{i. i \in \text{ids } g \wedge \text{nid} \in \text{inputs } g i\}$

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

fun *predecessors* :: *IRGraph* \Rightarrow *ID* \Rightarrow *ID set* **where**
predecessors g nid = $\{i. i \in \text{ids } g \wedge \text{nid} \in \text{succ } g i\}$

fun *nodes-of* :: *IRGraph* \Rightarrow (*IRNode* \Rightarrow *bool*) \Rightarrow *ID set* **where**
nodes-of g sel = $\{\text{nid} \in \text{ids } g. \text{sel } (\text{kind } g \text{ nid})\}$

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 \circ (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 \circ (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 = $\{n \in (\text{usages } g \text{ nid}). f (\text{kind } g n)\}$

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]*: $x \in \text{ids } g \longleftrightarrow \text{kind } g x \neq \text{NoNode}$

proof —

have *that*: $x \in \text{ids } g \longrightarrow \text{kind } g x \neq \text{NoNode}$

using *ids.rep-eq kind.rep-eq* **by** *force*

have $\text{kind } g x \neq \text{NoNode} \longrightarrow x \in \text{ids } g$

unfolding *with-default.simps kind-def ids-def*

by (*cases Rep-IRGraph g x = None; auto*)

from *this that* **show** *?thesis* **by** *auto*

qed

```

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

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

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

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

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

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

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

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

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

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

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

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

```

```

next
  case False
  then show ?thesis unfolding irgraph-def replace-node-def no-node.simps
    by (smt (verit, best) Rep-IRGraph comp-apply eq-onp-same-args filter.simps(2)
      id-def irgraph.rep-eq map-fun-apply map-of-upd mem-Collect-eq no-node.elims re-
      place-node.abs-eq replace-node-def snd-eqD)
qed

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

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

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

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

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

```

3.3.1 Example Graphs

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

definition *start-end-graph*:: IRGraph **where**

start-end-graph = irgraph [(0, StartNode None 1, VoidStamp), (1, ReturnNode None None, VoidStamp)]

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

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

definition *eg2-sq* :: *IRGraph* **where**

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

value *input-edges* *eg2-sq*

value *usages* *eg2-sq* 1

end

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*:

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

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

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

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

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

MinOrHighest *s* *m* = (if *s* = {} then *m* 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* $\implies n < \text{size } v$

by (*simp* *add*: *bit-imp-le-length* *size-word.rep-eq*)

lemma *max-set-bit*: *MaxOrNeg* {*n* . *bit* (*v*::('a::len) *word*) *n*} < *Nat.size* *v*

using *max-bit* **unfolding** *MaxOrNeg-def*

by *force*

4.1 Long.numberOfLeadingZeros

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

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

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

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

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

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

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

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

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

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

4.2 Long.numberOfTrailingZeros

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

lemma *lowestOneBit-bot*: *lowestOneBit* (0::64 word) = 64

unfolding *lowestOneBit-def MinOrHighest-def*
by (*simp add: size64*)

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

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

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

4.3 Long.bitCount

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

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

4.4 Long.zeroCount

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

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

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

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

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

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

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

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

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

lemma *card-of-range-bound*:
fixes *x y* :: nat
assumes *x* > *y*
shows *x* - *y* = *card* {*n* . *y* < *n* ∧ *n* ≤ *x*}
proof -
have *finite*: *finite* {*n* . *y* ≤ *n* ∧ *n* < *x*}
by *auto*
have *nonempty*: {*n* . *y* ≤ *n* ∧ *n* < *x*} ≠ {}
using *assms* **by** *blast*
have *simprep*: {*n* . *y* < *n* ∧ *n* ≤ *x*} = {*y*<..*x*}
by *auto*
have *x* - *y* = *card* {*y*<..*x*}
by *auto*
then show ?*thesis*
unfolding *simprep* **by** *blast*
qed

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

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

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

lemma *zerosBelowLowestOne*:
assumes *n* < *lowestOneBit* *a*
shows ¬(*bit* *a* *n*)
proof (*cases* {*i*. *bit* *a* *i*} = {})

```

    case True
    then show ?thesis by simp
next
    case False
    have  $n < \text{Min } (\text{Collect } (\text{bit } a)) \implies \neg \text{bit } a \ n$ 
    using False by auto
    then show ?thesis
    by (metis False MinOrHighest-def assms lowestOneBit-def)
qed

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

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

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

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

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

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

```



```

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

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

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

hide-fact intersect-bitCount-helper

end

```

5 Data-flow Semantics

```

theory IRTreeEval
  imports
    Graph.Stamp
  begin

```

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

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

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

The method state also stores a mapping of node ids to values. The contents of this mapping is 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 }
using is-ground.simps(6) by blast

```

5.2 Functions for re-calculating stamps

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

abbreviation *binary-fixed-32-ops* :: *IRBinaryOp* *set* **where**
binary-fixed-32-ops \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 out-
Bits v |
unary-eval (UnaryZeroExtend inBits outBits) v = intval-zero-extend inBits out-
Bits 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* ($\llbracket -, - \rrbracket \vdash - \mapsto -$ 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*)
 $[\text{show-steps}, \text{show-mode-inference}, \text{show-intermediate-results}]$
 $\text{evaltree} \ .$

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*)
 $\text{evaltrees} \ .$

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

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

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

```

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

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

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

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

definition
le-expr-def [simp]:
 $(e2 \leq e1) \longleftrightarrow (\forall m p v. (([m,p] \vdash e1 \mapsto v) \longrightarrow ([m,p] \vdash e2 \mapsto v)))$

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

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

end

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

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
by simp
```

```
lemma not-may-implies-false:
  [m, p] ⊢ e ↦ IntVal b v ⇒ ¬(bit (↑e) n) ⇒ bit v n = False
using up-spec
using bit-and-iff bit-eq-iff bit-not-iff bit-unsigned-iff down-spec
by (smt (verit, best) bit.double-compl)
```

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

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

```
lemma must-implies-may:
  [m, p] ⊢ e ↦ IntVal b v ⇒ n < 32 ⇒ bit (↓e) n ⇒ bit (↑e) n
by (meson must-implies-true not-may-implies-false)
```

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


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

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

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

end

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

IRExpr-up *e* = *not 0*

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

IRExpr-down *e* = *0*

lemma *ucast-zero*: (*ucast* (*0::int64*)::*int32*) = *0*

by *simp*

lemma *ucast-minus-one*: (*ucast* (*-1::int64*)::*int32*) = *-1*

apply *transfer by auto*

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

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

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

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

apply *unfold-locales*

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

end

5.6 Data-flow Tree Theorems

theory *IRTreeEvalThms*

imports

Graph.ValueThms

IRTreeEval

begin

5.6.1 Deterministic Data-flow Evaluation

lemma *evalDet*:

```

[m,p] ⊢ e ↦ v1 ⇒
[m,p] ⊢ e ↦ v2 ⇒
v1 = v2
apply (induction arbitrary: v2 rule: evaltree.induct)
by (elim EvalTreeE; auto)+

```

```

lemma evalAllDet:
[m,p] ⊢ e ↦L v1 ⇒
[m,p] ⊢ e ↦L v2 ⇒
v1 = v2
apply (induction arbitrary: v2 rule: evaltrees.induct)
apply (elim EvalTreeE; auto)
using evalDet by force

```

5.6.2 Typing Properties for Integer Evaluation Functions

We use three simple typing properties on integer values: *isIntVal32*, *isIntVal64* and the more general *isIntVal*.

```

lemma unary-eval-not-obj-ref:
shows unary-eval op x ≠ ObjRef v
by (cases op; cases x; auto)

```

```

lemma unary-eval-not-obj-str:
shows unary-eval op x ≠ ObjStr v
by (cases op; cases x; auto)

```

```

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

```

```

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

```

```

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

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

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

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

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

```

```

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

```

```

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

```

5.6.3 Evaluation Results are Valid

A valid value cannot be *UndefVal*.

```

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

```

```

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

```

```

lemma valid-ObjStamp[elim]:
  shows valid-value val (ObjectStamp klass exact nonNull alwaysNull) ⇒

```

```

    (∃ v. val = ObjRef v)
  using valid-value.simps by (metis val-to-bool.cases)

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

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

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

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

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

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

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

```

5.6.4 Example Data-flow Optimisations

5.6.5 Monotonicity of Expression Refinement

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

Note that we might also be able to do this via reusing Isabelle's *mono* operator (HOL.Orderings theory), proving instantiations like *mono(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 $(UnaryExpr\ op\ x) \geq (UnaryExpr\ op\ x')$
using *UnaryExpr assms* **by** *auto*

lemma *mono-binary*:

assumes $x \geq x'$
assumes $y \geq y'$
shows $(BinaryExpr\ op\ x\ y) \geq (BinaryExpr\ op\ x'\ y')$
using *BinaryExpr assms* **by** *auto*

lemma *never-void*:

assumes $[m, p] \vdash x \mapsto xv$
assumes *valid-value xv (stamp-expr xe)*
shows *stamp-expr xe* \neq *VoidStamp*
using *valid-value.simps*
using *assms(2)* **by** *force*

lemma *compatible-trans*:

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

lemma *compatible-refl*:

compatible x y \implies *compatible y x*
using *compatible.elims(2)* **by** *fastforce*

lemma *mono-conditional*:

assumes $c \geq c'$
assumes $t \geq t'$
assumes $f \geq f'$
shows $(ConditionalExpr\ c\ t\ f) \geq (ConditionalExpr\ c'\ t'\ f')$

```

proof (simp only: le-expr-def; (rule allI)+; rule impI)
  fix m p v
  assume a: [m,p] ⊢ ConditionalExpr c t f ↦ v
  then obtain cond where c: [m,p] ⊢ c ↦ cond by auto
  then have c': [m,p] ⊢ c' ↦ cond using assms by auto

  define branch where b: branch = (if val-to-bool cond then t else f)
  define branch' where b': branch' = (if val-to-bool cond then t' else f')
  then have beval: [m,p] ⊢ branch ↦ v using a b c evalDet by blast

  from beval have [m,p] ⊢ branch' ↦ v using assms b b' by auto
  then show [m,p] ⊢ ConditionalExpr c' t' f' ↦ v
    using ConditionalExpr c' b'
    by (simp add: evaltree-not-undef)
qed

```

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

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

lemma *unfold-const*:

```

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

```

lemma *unfold-binary*:

```

shows ([m,p] ⊢ BinaryExpr op xe ye ↦ val) = (∃ x y.
  ([m,p] ⊢ xe ↦ x) ∧
  ([m,p] ⊢ ye ↦ y) ∧
  (val = bin-eval op x y) ∧
  (val ≠ UndefVal)
  )) (is ?L = ?R)

```

proof (intro iffI)

assume \mathcal{I} : ?L

show ?R **by** (rule evaltree.cases[OF \mathcal{I}]; blast+)

next

assume ?R

then obtain x y **where** [m,p] ⊢ xe ↦ x

and [m,p] ⊢ ye ↦ y

and val = bin-eval op x y

and val ≠ UndefVal

by auto

then show ?L

by (rule BinaryExpr)

qed

```

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

```

```

lemmas unfold-evaltree =
  unfold-binary
  unfold-unary

```

5.8 Lemmas about *new_int* and integer eval results.

```

lemma unary-eval-new-int:
  assumes def: unary-eval op x ≠ UndefVal
  shows  $\exists \ b \ v. \text{unary-eval } op \ x = \text{new-int } b \ v \wedge$ 
     $b = (\text{if } op \in \text{normal-unary then intval-bits } x \text{ else ir-resultBits } op)$ 
proof (cases op ∈ normal-unary)
  case True
    then show ?thesis
    by (metis def empty-iff insert-iff intval-abs.elims intval-bits.simps intval-logic-negation.elims
      intval-negate.elims intval-not.elims unary-eval.simps(1) unary-eval.simps(2) unary-eval.simps(3)
      unary-eval.simps(4))
  next
    case False
    consider ib ob where op = UnaryNarrow ib ob |
      ib ob where op = UnaryZeroExtend ib ob |
      ib ob where op = UnarySignExtend ib ob
    by (metis False IRUnaryOp.exhaust insert-iff)
    then show ?thesis
    proof (cases)
      case 1
        then show ?thesis
        by (metis False IRUnaryOp.sel(4) def intval-narrow.elims unary-eval.simps(5))
      next
        case 2
        then show ?thesis
        by (metis False IRUnaryOp.sel(6) def intval-zero-extend.elims unary-eval.simps(7))
      next
        case 3
        then show ?thesis
        by (metis False IRUnaryOp.sel(5) def intval-sign-extend.elims unary-eval.simps(6))
    qed
qed

```



```

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

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

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

lemma eval-unused-bits-zero:
  [m,p] ⊢ xe ↦ (IntVal b ix) ⇒ take-bit b ix = ix
proof (induction xe)
  case (UnaryExpr x1 xe)
  then show ?case
    using unary-eval-unused-bits-zero by force
next
  case (BinaryExpr x1 xe1 xe2)
  then show ?case
    using bin-eval-unused-bits-zero by force
next
  case (ConditionalExpr xe1 xe2 xe3)
  then show ?case
    by (metis (full-types) EvalTreeE(3))
next
  case (ParameterExpr i s)
  then have valid-value (p!i) s
    by fastforce
  then show ?case
    by (metis ParameterExprE Value.distinct(7) intval-bits.simps intval-word.simps
local.ParameterExpr valid-value.elims(2))
next
  case (LeafExpr x1 x2)
  then show ?case
    by (smt (z3) EvalTreeE(6) Value.simps(11) valid-value.elims(1) valid-value.simps(1))

next
  case (ConstantExpr x)
  then show ?case using wf-value-def
    by (metis EvalTreeE(1) constantAsStamp.simps(1) valid-value.simps(1))
next

```

```

    case (ConstantVar x)
    then show ?case
      by fastforce
next
    case (VariableExpr x1 x2)
    then show ?case
      by fastforce
qed

```

```

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

```

```

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

```

```

lemma unary-eval-bitsize:
  assumes unary-eval op x = IntVal b ival
  assumes 2: x = IntVal bx ix
  assumes 0 < bx ∧ bx ≤ 64
  shows 0 < b ∧ b ≤ 64
proof (cases op ∈ normal-unary)
  case True
  then obtain tmp where unary-eval op x = new-int bx tmp
    by (cases op; simp; auto simp: 2)
  then show ?thesis
    using assms by simp

```

```

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

```

```

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

```

```

lemma eval-bits-1-64:
   $[m, p] \vdash xe \mapsto (IntVal b ix) \implies 0 < b \wedge b \leq 64$ 
proof (induction xe arbitrary: b ix)
  case (UnaryExpr op x2)
  then obtain xv where
    xv:  $[m, p] \vdash x2 \mapsto xv$   $\wedge$ 
    IntVal b ix = unary-eval op xv
  using unfold-binary by auto
  then have b = (if op  $\in$  normal-unary then intval-bits xv else ir-resultBits op)
  using unary-eval-new-int
  by (metis Value.disc(1) Value.discI(1) Value.sel(1) new-int.simps)
  then show ?case
    by (metis xv UnaryExpr.IH unary-normal-bitsize unary-not-normal-bitsize)
next
  case (BinaryExpr op x y)
  then obtain xv yv where
    xy:  $[m, p] \vdash x \mapsto xv$   $\wedge$ 
     $[m, p] \vdash y \mapsto yv$   $\wedge$ 
    IntVal b ix = bin-eval op xv yv
  using unfold-binary by auto
  then have def: bin-eval op xv yv  $\neq$  UndefVal and xv: xv  $\neq$  UndefVal and yv  $\neq$ 

```

```

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

```

lemma *unfold-binary-width:*

```

  assumes op ∉ binary-fixed-32-ops ∧ op ∉ binary-shift-ops
  shows ([m,p] ⊢ BinaryExpr op xe ye ↦ IntVal b val) = (∃ x y.
    ([m,p] ⊢ xe ↦ IntVal b x) ∧
    ([m,p] ⊢ ye ↦ IntVal b y) ∧
    (IntVal b val = bin-eval op (IntVal b x) (IntVal b y)) ∧
    (IntVal b val ≠ UndefVal)
  ) (is ?L = ?R)
proof (intro iffI)
  assume 3: ?L
  show ?R apply (rule evaltree.cases[OF 3])
  apply force+ apply auto[1]

```

```

    using assms apply (cases op; auto)
      apply (smt (verit) intval-add.elims Value.inject(1))
    using intval-mul.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps)
    using intval-sub.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps)
    using intval-and.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps take-bit-and)
    using intval-or.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps take-bit-or)
    using intval-xor.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps take-bit-xor)
  by blast

next
  assume R: ?R
  then obtain x y where [m,p] ⊢ xe ↦ IntVal b x
    and [m,p] ⊢ ye ↦ IntVal b y
    and new-int b val = bin-eval op (IntVal b x) (IntVal b y)
    and new-int b val ≠ UndefVal
  using bin-eval-unused-bits-zero by force
  then show ?L
    using R by blast
qed

end

```

6 Tree to Graph

```

theory TreeToGraph
  imports
    Semantics.IRTreeEval
    Graph.IRGraph
begin

```

6.1 Subgraph to Data-flow Tree

```

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

export-code find-node-and-stamp

```

```

fun is-preevaluated :: IRNode ⇒ bool where
  is-preevaluated (InvokeNode n - - - -) = True |
  is-preevaluated (InvokeWithExceptionNode n - - - -) = True |
  is-preevaluated (NewInstanceNode n - - -) = True |
  is-preevaluated (LoadFieldNode n - - -) = True |

```

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

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:

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

SignExtendNode:

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

ZeroExtendNode:

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

LeafNode:

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

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* .

inductive

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

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

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

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

definition *wf-term-graph* :: *MapState* \Rightarrow *Params* \Rightarrow *IRGraph* \Rightarrow *ID* \Rightarrow *bool* **where**
wf-term-graph *m p g n* = ($\exists e. (g \vdash n \simeq e) \wedge (\exists v. ([m, p] \vdash e \mapsto v))$)

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

6.2 Data-flow Tree to Subgraph

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

unary-node *UnaryAbs* *v* = *AbsNode* *v* |
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* .

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 .

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *graph-refinement*:

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

6.5 Maximal Sharing

definition *maximal-sharing*:

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

end

6.6 Formedness Properties

theory *Form*

imports

Semantics.TreeToGraph

begin

definition *wf-start* **where**

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

definition *wf-closed* **where**

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

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

definition *wf-phs* **where**

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

definition *wf-ends* **where**

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

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

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

lemmas *wf-folds* =

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

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

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

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

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

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

unfolding *start-end-graph-def wf-folds by simp*

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

unfolding *eg2-sq-def wf-folds by simp*

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

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

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

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

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

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*

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

lemma *other-node-unchanged:*

assumes *changeonly ns g1 g2*

assumes *nid* \in *ids g1*

assumes *nid* \notin *ns*

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

using *assms*

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

Some notation for input nodes used

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

for *g* **where**

use0: *nid* \in *ids g*


```

     $\implies \text{eval-uses } g \text{ nid nid} \mid$ 

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

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

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

lemma not-in-g-inputs:
    assumes nid  $\notin$  ids g
    shows inputs g nid = {}
proof –
    have k: kind g nid = NoNode using assms not-in-g by blast
    then show ?thesis by (simp add: k)
qed

lemma child-member:
    assumes n = kind g nid
    assumes n  $\neq$  NoNode
    assumes List.member (inputs-of n) child
    shows child  $\in$  inputs g nid
    unfolding inputs.simps using assms
    by (metis in-set-member)

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

lemma inp-in-g:
    assumes n  $\in$  inputs g nid
    shows nid  $\in$  ids g
proof –
    have inputs g nid  $\neq$  {}

```

```

    using assms
    by (metis empty-iff empty-set)
  then have kind g nid  $\neq$  NoNode
    using not-in-g-inputs
    using ids-some by blast
  then show ?thesis
    using not-in-g
    by metis
qed

```

```

lemma inp-in-g-wf:
  assumes wf-graph g
  assumes  $n \in \text{inputs } g \text{ nid}$ 
  shows  $n \in \text{ids } g$ 
  using assms unfolding wf-folds
  using inp-in-g by blast

```

```

lemma kind-unchanged:
  assumes  $\text{nid} \in \text{ids } g1$ 
  assumes unchanged (eval-usages g1 nid) g1 g2
  shows  $\text{kind } g1 \text{ nid} = \text{kind } g2 \text{ nid}$ 
proof -
  show ?thesis
    using assms eval-usages-self
    using unchanged.simps by blast
qed

```

```

lemma stamp-unchanged:
  assumes  $\text{nid} \in \text{ids } g1$ 
  assumes unchanged (eval-usages g1 nid) g1 g2
  shows  $\text{stamp } g1 \text{ nid} = \text{stamp } g2 \text{ nid}$ 
  by (meson assms(1) assms(2) eval-usages-self unchanged.elims(2))

```

```

lemma child-unchanged:
  assumes  $\text{child} \in \text{inputs } g1 \text{ nid}$ 
  assumes unchanged (eval-usages g1 nid) g1 g2
  shows unchanged (eval-usages g1 child) g1 g2
  by (smt assms(1) assms(2) eval-usages.simps mem-Collect-eq
    unchanged.simps use-inp use-trans)

```

```

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

```

```

lemma inputs-are-uses:
  assumes  $nid' \in \text{inputs } g \text{ } nid$ 
  shows  $\text{eval-uses } g \text{ } nid \text{ } nid'$ 
  by (metis assms use-inp)

lemma inputs-are-usages:
  assumes  $nid' \in \text{inputs } g \text{ } nid$ 
  assumes  $nid' \in \text{ids } g$ 
  shows  $nid' \in \text{eval-usages } g \text{ } nid$ 
  using assms(1) assms(2) eval-usages inputs-are-uses by blast

lemma inputs-of-are-usages:
  assumes  $\text{List.member } (\text{inputs-of } (\text{kind } g \text{ } nid)) \text{ } nid'$ 
  assumes  $nid' \in \text{ids } g$ 
  shows  $nid' \in \text{eval-usages } g \text{ } nid$ 
  by (metis assms(1) assms(2) in-set-member inputs.elims inputs-are-usages)

lemma usage-includes-inputs:
  assumes  $us = \text{eval-usages } g \text{ } nid$ 
  assumes  $ls = \text{inputs } g \text{ } nid$ 
  assumes  $ls \subseteq \text{ids } g$ 
  shows  $ls \subseteq us$ 
  using inputs-are-usages eval-usages
  using assms(1) assms(2) assms(3) by blast

lemma elim-inp-set:
  assumes  $k = \text{kind } g \text{ } nid$ 
  assumes  $k \neq \text{NoNode}$ 
  assumes  $\text{child} \in \text{set } (\text{inputs-of } k)$ 
  shows  $\text{child} \in \text{inputs } g \text{ } nid$ 
  using assms by auto

lemma encode-in-ids:
  assumes  $g \vdash nid \simeq e$ 
  shows  $nid \in \text{ids } g$ 
  using assms
  apply (induction rule: rep.induct)
  apply simp+
  by fastforce+

lemma eval-in-ids:
  assumes  $[g, m, p] \vdash nid \mapsto v$ 
  shows  $nid \in \text{ids } g$ 
  using assms using encodeeval-def encode-in-ids
  by auto

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

```

```

using assms
by (meson unchanged.elims(1))

theorem stay-same-encoding:
  assumes nc: unchanged (eval-usages g1 nid) g1 g2
  assumes g1: g1 ⊢ nid ≃ e
  assumes wf: wf-graph g1
  shows g2 ⊢ nid ≃ e
proof -
  have dom: nid ∈ ids g1
    using g1 encode-in-ids by simp
  show ?thesis
using g1 nc wf dom proof (induction e rule: rep.induct)
  case (ConstantNode n c)
  then have kind g2 n = ConstantNode c
    using dom nc kind-unchanged
    by metis
  then show ?case using rep.ConstantNode
    by presburger
next
  case (ParameterNode n i s)
  then have kind g2 n = ParameterNode i
    by (metis kind-unchanged)
  then show ?case
    by (metis ParameterNode.hyps(2) ParameterNode.prem(1) ParameterNode.prem(3)
rep.ParameterNode stamp-unchanged)
next
  case (ConditionalNode n c t f ce te fe)
  then have kind g2 n = ConditionalNode c t f
    by (metis kind-unchanged)
  have c ∈ eval-usages g1 n ∧ t ∈ eval-usages g1 n ∧ f ∈ eval-usages g1 n
    using inputs-of-ConditionalNode
    by (metis ConditionalNode.hyps(1) ConditionalNode.hyps(2) ConditionalNode.hyps(3) ConditionalNode.hyps(4) encode-in-ids inputs.simps inputs-are-usages
list.set-intros(1) set-subset-Cons subset-code(1))
  then show ?case using transitive-kind-same
    by (metis ConditionalNode.hyps(1) ConditionalNode.prem(1) IRNodes.inputs-of-ConditionalNode
⟨kind g2 n = ConditionalNode c t f⟩ child-unchanged inputs.simps list.set-intros(1)
local.ConditionalNode(5) local.ConditionalNode(6) local.ConditionalNode(7) local.ConditionalNode(9)
rep.ConditionalNode set-subset-Cons subset-code(1) unchanged.elims(2))
next
  case (AbsNode n x xe)
  then have kind g2 n = AbsNode x
    using kind-unchanged
    by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-AbsNode
    by (metis AbsNode.hyps(1) AbsNode.hyps(2) encode-in-ids inputs.simps inputs-are-usages
list.set-intros(1))

```

```

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

```

```

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

```

```

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

```

```

case (UnsignedRightShiftNode n x y xe ye)
  then have kind g2 n = UnsignedRightShiftNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-UnsignedRightShiftNode inputs-of-are-usages
    by (metis UnsignedRightShiftNode.hyps(1) UnsignedRightShiftNode.hyps(2) Un-
signedRightShiftNode.hyps(3) IRNodes.inputs-of-UnsignedRightShiftNode encode-in-ids
in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using UnsignedRightShiftNode inputs-of-UnsignedRightShiftNode
    by (metis ⟨kind g2 n = UnsignedRightShiftNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.UnsignedRightShiftNode)
next
  case (IntegerBelowNode n x y xe ye)
  then have kind g2 n = IntegerBelowNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-IntegerBelowNode inputs-of-are-usages
    by (metis IntegerBelowNode.hyps(1) IntegerBelowNode.hyps(2) IntegerBelowN-
ode.hyps(3) IRNodes.inputs-of-IntegerBelowNode encode-in-ids in-mono inputs.simps
inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using IntegerBelowNode inputs-of-IntegerBelowNode
    by (metis ⟨kind g2 n = IntegerBelowNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.IntegerBelowNode)
next
  case (IntegerEqualsNode n x y xe ye)
  then have kind g2 n = IntegerEqualsNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-IntegerEqualsNode inputs-of-are-usages
    by (metis IntegerEqualsNode.hyps(1) IntegerEqualsNode.hyps(2) IntegerEqual-
sNode.hyps(3) IRNodes.inputs-of-IntegerEqualsNode encode-in-ids in-mono inputs.simps
inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using IntegerEqualsNode inputs-of-IntegerEqualsNode
    by (metis ⟨kind g2 n = IntegerEqualsNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.IntegerEqualsNode)
next
  case (IntegerLessThanNode n x y xe ye)
  then have kind g2 n = IntegerLessThanNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-IntegerLessThanNode inputs-of-are-usages
    by (metis IntegerLessThanNode.hyps(1) IntegerLessThanNode.hyps(2) Inte-
gerLessThanNode.hyps(3) IRNodes.inputs-of-IntegerLessThanNode encode-in-ids in-mono
inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using IntegerLessThanNode inputs-of-IntegerLessThanNode
    by (metis ⟨kind g2 n = IntegerLessThanNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.IntegerLessThanNode)
next
  case (NarrowNode n ib rb x xe)

```



```

then have kind g2 n = NarrowNode ib rb x
  using kind-unchanged by metis
then have x ∈ eval-usages g1 n
  using inputs-of-NarrowNode inputs-of-are-usages
  by (metis NarrowNode.hyps(1) NarrowNode.hyps(2) IRNodes.inputs-of-NarrowNode
    encode-in-ids inputs.simps inputs-are-usages list.set-intros(1))
  then show ?case using NarrowNode inputs-of-NarrowNode
    by (metis ⟨kind g2 n = NarrowNode ib rb x⟩ child-unchanged inputs.elims
      list.set-intros(1) rep.NarrowNode unchanged.simps)
next
  case (SignExtendNode n ib rb x xe)
  then have kind g2 n = SignExtendNode ib rb x
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-SignExtendNode inputs-of-are-usages
    by (metis SignExtendNode.hyps(1) SignExtendNode.hyps(2) encode-in-ids in-
      puts.simps inputs-are-usages list.set-intros(1))
    then show ?case using SignExtendNode inputs-of-SignExtendNode
      by (metis ⟨kind g2 n = SignExtendNode ib rb x⟩ child-member-in child-unchanged
        in-set-member list.set-intros(1) rep.SignExtendNode unchanged.elims(2))
  next
    case (ZeroExtendNode n ib rb x xe)
    then have kind g2 n = ZeroExtendNode ib rb x
      using kind-unchanged by metis
    then have x ∈ eval-usages g1 n
      using inputs-of-ZeroExtendNode inputs-of-are-usages
      by (metis ZeroExtendNode.hyps(1) ZeroExtendNode.hyps(2) IRNodes.inputs-of-ZeroExtendNode
        encode-in-ids inputs.simps inputs-are-usages list.set-intros(1))
      then show ?case using ZeroExtendNode inputs-of-ZeroExtendNode
        by (metis ⟨kind g2 n = ZeroExtendNode ib rb x⟩ child-member-in child-unchanged
          member-rec(1) rep.ZeroExtendNode unchanged.simps)
    next
      case (LeafNode n s)
      then show ?case
        by (metis kind-unchanged rep.LeafNode stamp-unchanged)
    next
      case (RefNode n n')
      then have kind g2 n = RefNode n'
        using kind-unchanged by metis
      then have n' ∈ eval-usages g1 n
        by (metis IRNodes.inputs-of-RefNode RefNode.hyps(1) RefNode.hyps(2) en-
          code-in-ids inputs.elims inputs-are-usages list.set-intros(1))
      then show ?case
        by (metis IRNodes.inputs-of-RefNode RefNode.IH RefNode.hyps(1) RefNode.hyps(2)
          RefNode.premis(1) ⟨kind g2 n = RefNode n'⟩ child-unchanged encode-in-ids in-
          puts.elims list.set-intros(1) local.wf rep.RefNode)
qed
qed

```

```

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

```

```

next
  case (LeafExpr val nid s)
  then show ?case
    by (metis local.wf stay-same-encoding)
qed
qed

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

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

lemma add-node-unchanged:
  assumes new  $\notin$  ids g
  assumes nid  $\in$  ids g
  assumes gup = add-node new k g
  assumes wf-graph g
  shows unchanged (eval-usages g nid) g gup
proof -
  have new  $\notin$  (eval-usages g nid) using assms
    using eval-usages.simps by blast
  then have changeonly {new} g gup
    using assms add-changed by blast
  then show ?thesis using assms add-node-def disjoint-change
    using Diff-insert-absorb by auto
qed

lemma eval-uses-imp:
  ((nid'  $\in$  ids g  $\wedge$  nid = nid')
   $\vee$  nid'  $\in$  inputs g nid
   $\vee$  ( $\exists$  nid'' . eval-uses g nid nid''  $\wedge$  eval-uses g nid'' nid'))
 $\longleftrightarrow$  eval-uses g nid nid'
  using use0 use-inp use-trans
  by (meson eval-uses.simps)

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

```

```

    using assms(3)
  proof (induction rule: eval-uses.induct)
    case use0
    then show ?case by simp
  next
    case use-inp
    then show ?case
      using assms(1) inp-in-g-wf by blast
  next
    case use-trans
    then show ?case by blast
  qed

lemma no-external-use:
  assumes wf-graph g
  assumes nid'  $\notin$  ids g
  assumes nid  $\in$  ids g
  shows  $\neg$ (eval-uses g nid nid')
proof -
  have 0: nid  $\neq$  nid'
  using assms by blast
  have inp: nid'  $\notin$  inputs g nid
  using assms
  using inp-in-g-wf by blast
  have rec-0:  $\nexists n . n \in$  ids g  $\wedge$  n = nid'
  using assms by blast
  have rec-inp:  $\nexists n . n \in$  ids g  $\wedge$  n  $\in$  inputs g nid'
  using assms(2) inp-in-g by blast
  have rec:  $\nexists$  nid'' . eval-uses g nid nid''  $\wedge$  eval-uses g nid'' nid'
  using wf-use-ids assms(1) assms(2) assms(3) by blast
  from inp 0 rec show ?thesis
  using eval-uses-imp by blast
qed

end

```

6.8 Tree to Graph Theorems

```

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

```

6.8.1 Extraction and Evaluation of Expression Trees is Deterministic.

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

named-theorems *rep*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *rep-narrow* [rep]:

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

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

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

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

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

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

```

g ⊢ n ≃ e ⇒
is-preevaluated (kind g n) ⇒
(∃ s. e = LeafExpr n s)
by (induction rule: rep.induct; auto)

```

lemma *rep-ref* [*rep*]:

```

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

```

method *solve-det* **uses** *node* =

```

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

```

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

lemma *repDet*:

shows $(g ⊢ n ≃ e_1) ⇒ (g ⊢ n ≃ e_2) ⇒ e_1 = e_2$

proof (*induction arbitrary: e₂ rule: rep.induct*)

case (*ConstantNode n c*)

then show ?*case* **using** *rep-constant* **by** *auto*

next

case (*ParameterNode n i s*)

then show ?*case*

by (*metis IRNode.disc(2685) ParameterNodeE is-RefNode-def rep-parameter*)


```

next
  case (ConditionalNode n c t f ce te fe)
  then show ?case
    using IRNode.distinct(593)
    using IRNode.inject(6) ConditionalNodeE rep-conditional
    by metis
next
  case (AbsNode n x xe)
  then show ?case
    by (solve-det node: AbsNode)
next
  case (NotNode n x xe)
  then show ?case
    by (solve-det node: NotNode)
next
  case (NegateNode n x xe)
  then show ?case
    by (solve-det node: NegateNode)
next
  case (LogicNegationNode n x xe)
  then show ?case
    by (solve-det node: LogicNegationNode)
next
  case (AddNode n x y xe ye)
  then show ?case
    by (solve-det node: AddNode)
next
  case (MulNode n x y xe ye)
  then show ?case
    by (solve-det node: MulNode)
next
  case (SubNode n x y xe ye)
  then show ?case
    by (solve-det node: SubNode)
next
  case (AndNode n x y xe ye)
  then show ?case
    by (solve-det node: AndNode)
next
  case (OrNode n x y xe ye)
  then show ?case
    by (solve-det node: OrNode)
next
  case (XorNode n x y xe ye)
  then show ?case
    by (solve-det node: XorNode)
next
  case (ShortCircuitOrNode n x y xe ye)
  then show ?case

```

```

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

```

lemma repAllDet:
 $g \vdash xs \simeq_L e1 \implies$

```

  g ⊢ xs ≃L e2 ⇒
  e1 = e2
proof (induction arbitrary: e2 rule: replist.induct)
  case RepNil
  then show ?case
    using replist.cases by auto
next
  case (RepCons x xe xs xse)
  then show ?case
    by (metis list.distinct(1) list.sel(1) list.sel(3) repDet replist.cases)
qed

```

```

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

```

```

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

```

6.8.2 Monotonicity of Graph Refinement

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

```

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

```

```

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

```

```

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

```

lemma *mono-logic-negation*:

assumes $\text{kind } g1 \ n = \text{LogicNegationNode } x \wedge \text{kind } g2 \ n = \text{LogicNegationNode } x$
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 $e1 \geq e2$
by (*metis* *LogicNegationNode* *assms(1)* *assms(2)* *assms(3)* *assms(4)* *mono-unary* *repDet*)

lemma *mono-narrow*:

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

lemma *mono-sign-extend*:

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

lemma *mono-zero-extend*:

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

lemma *mono-conditional-graph*:

assumes $\text{kind } g1 \ n = \text{ConditionalNode } c \ t \ f \wedge \text{kind } g2 \ n = \text{ConditionalNode } c \ t \ f$
assumes $(g1 \vdash c \simeq ce1) \wedge (g2 \vdash c \simeq ce2)$
assumes $(g1 \vdash t \simeq te1) \wedge (g2 \vdash t \simeq te2)$
assumes $(g1 \vdash f \simeq fe1) \wedge (g2 \vdash f \simeq fe2)$
assumes $ce1 \geq ce2 \wedge te1 \geq te2 \wedge fe1 \geq fe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
using *ConditionalNodeE* *IRNode.inject(6)* *assms(1)* *assms(2)* *assms(3)* *assms(4)* *assms(5)* *assms(6)* *mono-conditional* *repDet* *rep-conditional*

by (*smt* (*verit*, *best*) *ConditionalNode*)

lemma *mono-add*:

assumes *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*
using *mono-binary assms AddNodeE IRNode.inject(2) repDet rep-add*
by (*metis IRNode.distinct(205)*)

lemma *mono-mul*:

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

lemma *term-graph-evaluation*:

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

lemma *encodes-contains*:

g \vdash *n* \simeq *e* \implies
kind g n \neq *NoNode*
apply (*induction rule: rep.induct*)
apply (*match IRNode.distinct in e: ?n* \neq *NoNode* \implies
 \langle *presburger add: e* \rangle $+$
apply *force*
by *fastforce*

lemma *no-encoding*:

assumes *n* \notin *ids g*
shows \neg (*g* \vdash *n* \simeq *e*)
using *assms* **apply** *simp* **apply** (*rule notI*) **by** (*induction e; simp add: encodes-contains*)

lemma *not-excluded-keep-type*:

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

```

method metis-node-eq-unary for node :: 'a  $\Rightarrow$  IRNode =
  (match IRNode.inject in i: (node - = node -) = -  $\Rightarrow$ 
     $\langle$ metis i $\rangle$ )
method metis-node-eq-binary for node :: 'a  $\Rightarrow$  'a  $\Rightarrow$  IRNode =
  (match IRNode.inject in i: (node - - = node - -) = -  $\Rightarrow$ 
     $\langle$ metis i $\rangle$ )
method metis-node-eq-ternary for node :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  IRNode =
  (match IRNode.inject in i: (node - - - = node - - -) = -  $\Rightarrow$ 
     $\langle$ metis i $\rangle$ )

```

6.8.3 Lift Data-flow Tree Refinement to Graph Refinement

```

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

  show  $\exists e2. (g2 \vdash n \simeq e2) \wedge e1 \geq e2$ 
  proof (cases  $n = n'$ )
    case True
    have g:  $e1 = e1'$  using c f True repDet by simp
    have h:  $(g2 \vdash n \simeq e2') \wedge e1' \geq e2'$ 
      using True a d by blast
    then show ?thesis
      using g by blast
    next
    case False
    have  $n \notin \{n'\}$ 
      using False by simp
    then have i:  $\text{kind } g1 \ n = \text{kind } g2 \ n \wedge \text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
      using not-excluded-keep-type
      using b e by presburger
    show ?thesis using f i
  proof (induction e1)
    case (ConstantNode n c)
    then show ?case
      by (metis eq-refl rep.ConstantNode)

```

```

next
  case (ParameterNode n i s)
  then show ?case
    by (metis eq-refl rep.ParameterNode)
next
  case (ConditionalNode n c t f ce1 te1 fe1)
  have k:  $g1 \vdash n \simeq \text{ConditionalExpr } ce1 \text{ te1 } fe1$  using f ConditionalNode
    by (simp add: ConditionalNode.hyps(2) rep.ConditionalNode)
  obtain cn tn fn where l:  $\text{kind } g1 \text{ } n = \text{ConditionalNode } cn \text{ tn } fn$ 
    using ConditionalNode.hyps(1) by blast
  then have mc:  $g1 \vdash cn \simeq ce1$ 
    using ConditionalNode.hyps(1) ConditionalNode.hyps(2) by fastforce
  from l have mt:  $g1 \vdash tn \simeq te1$ 
    using ConditionalNode.hyps(1) ConditionalNode.hyps(3) by fastforce
  from l have mf:  $g1 \vdash fn \simeq fe1$ 
    using ConditionalNode.hyps(1) ConditionalNode.hyps(4) by fastforce
  then show ?case
  proof -
    have  $g1 \vdash cn \simeq ce1$  using mc by simp
    have  $g1 \vdash tn \simeq te1$  using mt by simp
    have  $g1 \vdash fn \simeq fe1$  using mf by simp
    have cer:  $\exists ce2. (g2 \vdash cn \simeq ce2) \wedge ce1 \geq ce2$ 
      using ConditionalNode
      using a b c d l no-encoding not-excluded-keep-type repDet singletonD
      by (metis-node-eq-ternary ConditionalNode)
    have ter:  $\exists te2. (g2 \vdash tn \simeq te2) \wedge te1 \geq te2$ 
      using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
      singletonD
      by (metis-node-eq-ternary ConditionalNode)
    have  $\exists fe2. (g2 \vdash fn \simeq fe2) \wedge fe1 \geq fe2$ 
      using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
      singletonD
      by (metis-node-eq-ternary ConditionalNode)
    then have  $\exists ce2 \text{ te2 } fe2. (g2 \vdash n \simeq \text{ConditionalExpr } ce2 \text{ te2 } fe2) \wedge$ 
       $\text{ConditionalExpr } ce1 \text{ te1 } fe1 \geq \text{ConditionalExpr } ce2 \text{ te2 } fe2$ 
      using ConditionalNode.premis l rep.ConditionalNode cer ter
      by (smt (verit) mono-conditional)
    then show ?thesis
      by meson
  qed
next
  case (AbsNode n x xe1)
  have k:  $g1 \vdash n \simeq \text{UnaryExpr } \text{UnaryAbs } xe1$  using f AbsNode
    by (simp add: AbsNode.hyps(2) rep.AbsNode)
  obtain xn where l:  $\text{kind } g1 \text{ } n = \text{AbsNode } xn$ 
    using AbsNode.hyps(1) by blast
  then have m:  $g1 \vdash xn \simeq xe1$ 
    using AbsNode.hyps(1) AbsNode.hyps(2) by fastforce
  then show ?case

```

```

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

```



```

      then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr UnaryNot } xe2) \wedge \text{UnaryExpr}$ 
         $\text{UnaryNot } xe1 \geq \text{UnaryExpr UnaryNot } xe2$ 
      by (metis NotNode.premis l mono-unary rep.NotNode)
    then show ?thesis
    by meson
  qed
next
case (NegateNode n x xe1)
have k:  $g1 \vdash n \simeq \text{UnaryExpr UnaryNeg } xe1$  using f NegateNode
by (simp add: NegateNode.hyps(2) rep.NegateNode)
obtain xn where l: kind g1 n = NegateNode xn
using NegateNode.hyps(1) by blast
then have m:  $g1 \vdash xn \simeq xe1$ 
using NegateNode.hyps(1) NegateNode.hyps(2) by fastforce
then show ?case
proof (cases xn = n')
case True
then have n:  $xe1 = e1'$  using c m repDet by simp
then have ev:  $g2 \vdash n \simeq \text{UnaryExpr UnaryNeg } e2'$  using NegateNode.hyps(1)
l m n
using NegateNode.premis True d rep.NegateNode by simp
then have r:  $\text{UnaryExpr UnaryNeg } e1' \geq \text{UnaryExpr UnaryNeg } e2'$ 
by (meson a mono-unary)
then show ?thesis using ev r
by (metis n)
next
case False
have  $g1 \vdash xn \simeq xe1$  using m by simp
have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
using NegateNode
using False i b l not-excluded-keep-type singletonD no-encoding
by (metis-node-eq-unary NegateNode)
then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr UnaryNeg } xe2) \wedge \text{UnaryExpr}$ 
 $\text{UnaryNeg } xe1 \geq \text{UnaryExpr UnaryNeg } xe2$ 
by (metis NegateNode.premis l mono-unary rep.NegateNode)
then show ?thesis
by meson
qed
next
case (LogicNegationNode n x xe1)
have k:  $g1 \vdash n \simeq \text{UnaryExpr UnaryLogicNegation } xe1$  using f LogicNegationNode
by (simp add: LogicNegationNode.hyps(2) rep.LogicNegationNode)
obtain xn where l: kind g1 n = LogicNegationNode xn
using LogicNegationNode.hyps(1) by blast
then have m:  $g1 \vdash xn \simeq xe1$ 
using LogicNegationNode.hyps(1) LogicNegationNode.hyps(2) by fastforce
then show ?case
proof (cases xn = n')

```

```

    case True
    then have n:  $xe1 = e1'$  using c m repDet by simp
    then have ev:  $g2 \vdash n \simeq \text{UnaryExpr UnaryLogicNegation } e2'$  using LogicNegationNode.hyps(1) l m n
    using LogicNegationNode.premis True d rep.LogicNegationNode by simp
    then have r:  $\text{UnaryExpr UnaryLogicNegation } e1' \geq \text{UnaryExpr UnaryLogicNegation } e2'$ 
    by (meson a mono-unary)
    then show ?thesis using ev r
    by (metis n)
  next
  case False
  have  $g1 \vdash xn \simeq xe1$  using m by simp
  have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
  using LogicNegationNode
  using False i b l not-excluded-keep-type singletonD no-encoding
  by (metis-node-eq-unary LogicNegationNode)
  then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr UnaryLogicNegation } xe2) \wedge$ 
 $\text{UnaryExpr UnaryLogicNegation } xe1 \geq \text{UnaryExpr UnaryLogicNegation } xe2$ 
  by (metis LogicNegationNode.premis l mono-unary rep.LogicNegationNode)
  then show ?thesis
  by meson
qed
next
case (AddNode n x y xe1 ye1)
have k:  $g1 \vdash n \simeq \text{BinaryExpr BinAdd } xe1 ye1$  using f AddNode
by (simp add: AddNode.hyps(2) rep.AddNode)
obtain  $xn yn$  where l: kind  $g1 n = \text{AddNode } xn yn$ 
using AddNode.hyps(1) by blast
then have mx:  $g1 \vdash xn \simeq xe1$ 
using AddNode.hyps(1) AddNode.hyps(2) by fastforce
from l have my:  $g1 \vdash yn \simeq ye1$ 
using AddNode.hyps(1) AddNode.hyps(3) by fastforce
then show ?case
proof -
  have  $g1 \vdash xn \simeq xe1$  using mx by simp
  have  $g1 \vdash yn \simeq ye1$  using my by simp
  have xer:  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
  using AddNode
  using a b c d l no-encoding not-excluded-keep-type repDet singletonD
  by (metis-node-eq-binary AddNode)
  have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
  using AddNode
  using a b c d l no-encoding not-excluded-keep-type repDet singletonD
  by (metis-node-eq-binary AddNode)
  then have  $\exists xe2 ye2. (g2 \vdash n \simeq \text{BinaryExpr BinAdd } xe2 ye2) \wedge \text{BinaryExpr BinAdd } xe1 ye1 \geq \text{BinaryExpr BinAdd } xe2 ye2$ 
  by (metis AddNode.premis l mono-binary rep.AddNode xer)
  then show ?thesis

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

      then have  $\exists xe2 ye2. (g2 \vdash n \simeq BinaryExpr\ BinIntegerLessThan\ xe2\ ye2)$ 
 $\wedge BinaryExpr\ BinIntegerLessThan\ xe1\ ye1 \geq BinaryExpr\ BinIntegerLessThan\ xe2\ ye2$ 
    by (metis IntegerLessThanNode.prem1 mono-binary rep.IntegerLessThanNode
xer)
      then show ?thesis
        by meson
      qed
    next
      case (NarrowNode n inputBits resultBits x xe1)
      have k:  $g1 \vdash n \simeq UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)\ xe1$  using
f NarrowNode
        by (simp add: NarrowNode.hyps(2) rep.NarrowNode)
      obtain xn where l: kind g1 n = NarrowNode inputBits resultBits xn
        using NarrowNode.hyps(1) by blast
      then have m:  $g1 \vdash xn \simeq xe1$ 
        using NarrowNode.hyps(1) NarrowNode.hyps(2)
        by auto
      then show ?case
      proof (cases xn = n')
        case True
          then have n:  $xe1 = e1'$  using c m repDet by simp
          then have ev:  $g2 \vdash n \simeq UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)$ 
e2' using NarrowNode.hyps(1) l m n
            using NarrowNode.prem1 True d rep.NarrowNode by simp
          then have r:  $UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)\ e1' \geq Unary-$ 
Expr (UnaryNarrow inputBits resultBits) e2'
            by (meson a mono-unary)
          then show ?thesis using ev r
            by (metis n)
        case False
          have  $g1 \vdash xn \simeq xe1$  using m by simp
          have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
            using NarrowNode
            using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
            by (metis node-eq-ternary NarrowNode)
          then have  $\exists xe2. (g2 \vdash n \simeq UnaryExpr\ (UnaryNarrow\ inputBits\ result-$ 
Bits) xe2)  $\wedge UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)\ xe1 \geq UnaryExpr$ 
(UnaryNarrow inputBits resultBits) xe2
            by (metis NarrowNode.prem1 mono-unary rep.NarrowNode)
          then show ?thesis
            by meson
          qed
        next
          case (SignExtendNode n inputBits resultBits x xe1)
          have k:  $g1 \vdash n \simeq UnaryExpr\ (UnarySignExtend\ inputBits\ resultBits)\ xe1$ 
using f SignExtendNode
            by (simp add: SignExtendNode.hyps(2) rep.SignExtendNode)

```

```

obtain  $xn$  where  $l$ :  $\text{kind } g1 \ n = \text{SignExtendNode } \text{inputBits } \text{resultBits } xn$ 
  using  $\text{SignExtendNode.hyps}(1)$  by blast
then have  $m$ :  $g1 \vdash xn \simeq xe1$ 
  using  $\text{SignExtendNode.hyps}(1)$   $\text{SignExtendNode.hyps}(2)$ 
  by auto
then show  $?case$ 
proof ( $\text{cases } xn = n'$ )
  case True
    then have  $n$ :  $xe1 = e1'$  using  $c \ m \ \text{repDet}$  by simp
    then have  $ev$ :  $g2 \vdash n \simeq \text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits})$ 
   $e2'$  using  $\text{SignExtendNode.hyps}(1)$   $l \ m \ n$ 
    using  $\text{SignExtendNode.premis } \text{True } d \ \text{rep.SignExtendNode}$  by simp
    then have  $r$ :  $\text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ e1' \geq$ 
 $\text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ e2'$ 
    by (meson a mono-unary)
    then show  $?thesis$  using  $ev \ r$ 
    by (metis n)
  next
  case False
    have  $g1 \vdash xn \simeq xe1$  using  $m$  by simp
    have  $\exists \ xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
    using  $\text{SignExtendNode}$ 
    using  $\text{False } b \ \text{encodes-contains } l \ \text{not-excluded-keep-type not-in-g singleton-iff}$ 
    by (metis-node-eq-ternary SignExtendNode)
    then have  $\exists \ xe2. (g2 \vdash n \simeq \text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{result-}$ 
 $\text{Bits}) \ xe2) \wedge \text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ xe1 \geq \text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ xe2$ 
    by (metis SignExtendNode.premis l mono-unary rep.SignExtendNode)
    then show  $?thesis$ 
    by meson
  qed
next
  case ( $\text{ZeroExtendNode } n \ \text{inputBits } \text{resultBits } x \ xe1$ )
    have  $k$ :  $g1 \vdash n \simeq \text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits}) \ xe1$ 
using  $f \ \text{ZeroExtendNode}$ 
    by (simp add: ZeroExtendNode.hyps(2) rep.ZeroExtendNode)
    obtain  $xn$  where  $l$ :  $\text{kind } g1 \ n = \text{ZeroExtendNode } \text{inputBits } \text{resultBits } xn$ 
    using  $\text{ZeroExtendNode.hyps}(1)$  by blast
    then have  $m$ :  $g1 \vdash xn \simeq xe1$ 
    using  $\text{ZeroExtendNode.hyps}(1)$   $\text{ZeroExtendNode.hyps}(2)$ 
    by auto
    then show  $?case$ 
    proof ( $\text{cases } xn = n'$ )
      case True
        then have  $n$ :  $xe1 = e1'$  using  $c \ m \ \text{repDet}$  by simp
        then have  $ev$ :  $g2 \vdash n \simeq \text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits})$ 
       $e2'$  using  $\text{ZeroExtendNode.hyps}(1)$   $l \ m \ n$ 
        using  $\text{ZeroExtendNode.premis } \text{True } d \ \text{rep.ZeroExtendNode}$  by simp
        then have  $r$ :  $\text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits}) \ e1' \geq$ 

```

```

UnaryExpr (UnaryZeroExtend inputBits resultBits) e2'
  by (meson a mono-unary)
  then show ?thesis using ev r
  by (metis n)
next
case False
have g1 ⊢ xn ≃ xe1 using m by simp
have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
  using ZeroExtendNode
  using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
  by (metis-node-eq-ternary ZeroExtendNode)
then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits result-
Bits) xe2) ∧ UnaryExpr (UnaryZeroExtend inputBits resultBits) xe1 ≥ UnaryExpr
(UnaryZeroExtend inputBits resultBits) xe2
  by (metis ZeroExtendNode.premis l mono-unary rep.ZeroExtendNode)
  then show ?thesis
  by meson
qed
next
case (LeafNode n s)
then show ?case
  by (metis eq-refl rep.LeafNode)
next
case (RefNode n')
then show ?case
  by (metis a b c d no-encoding not-excluded-keep-type rep.RefNode repDet
singletonD)
qed
qed
qed

```

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

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

```

declare [[simp-trace]]
lemma equal-refines:
  fixes e1 e2 :: IRExpr
  assumes e1 = e2
  shows e1 ≥ e2
  using assms
  by simp
declare [[simp-trace=false]]

lemma eval-contains-id[simp]: g1 ⊢ n ≃ e ⇒ n ∈ ids g1
  using no-encoding by blast

lemma subset-kind[simp]: as-set g1 ⊆ as-set g2 ⇒ g1 ⊢ n ≃ e ⇒ kind g1 n =
kind g2 n
  using eval-contains-id unfolding as-set-def
  by blast

lemma subset-stamp[simp]: as-set g1 ⊆ as-set g2 ⇒ g1 ⊢ n ≃ e ⇒ stamp g1
n = stamp g2 n
  using eval-contains-id unfolding as-set-def
  by blast

method solve-subset-eval uses as-set eval =
  (metis eval as-set subset-kind subset-stamp |
   metis eval as-set subset-kind)

lemma subset-implies-evals:
  assumes as-set g1 ⊆ as-set g2
  assumes (g1 ⊢ n ≃ e)
  shows (g2 ⊢ n ≃ e)
  using assms(2)
  apply (induction e)
    apply (solve-subset-eval as-set: assms(1) eval: ConstantNode)
    apply (solve-subset-eval as-set: assms(1) eval: ParameterNode)
    apply (solve-subset-eval as-set: assms(1) eval: ConditionalNode)
    apply (solve-subset-eval as-set: assms(1) eval: AbsNode)
    apply (solve-subset-eval as-set: assms(1) eval: NotNode)
    apply (solve-subset-eval as-set: assms(1) eval: NegateNode)
    apply (solve-subset-eval as-set: assms(1) eval: LogicNegationNode)
    apply (solve-subset-eval as-set: assms(1) eval: AddNode)
    apply (solve-subset-eval as-set: assms(1) eval: MulNode)
    apply (solve-subset-eval as-set: assms(1) eval: SubNode)
    apply (solve-subset-eval as-set: assms(1) eval: AndNode)
    apply (solve-subset-eval as-set: assms(1) eval: OrNode)
    apply (solve-subset-eval as-set: assms(1) eval: XorNode)
    apply (solve-subset-eval as-set: assms(1) eval: ShortCircuitOrNode)

```

```

    apply (solve-subset-eval as-set: assms(1) eval: LeftShiftNode)
    apply (solve-subset-eval as-set: assms(1) eval: RightShiftNode)
    apply (solve-subset-eval as-set: assms(1) eval: UnsignedRightShiftNode)
    apply (solve-subset-eval as-set: assms(1) eval: IntegerBelowNode)
    apply (solve-subset-eval as-set: assms(1) eval: IntegerEqualsNode)
    apply (solve-subset-eval as-set: assms(1) eval: IntegerLessThanNode)
    apply (solve-subset-eval as-set: assms(1) eval: NarrowNode)
    apply (solve-subset-eval as-set: assms(1) eval: SignExtendNode)
    apply (solve-subset-eval as-set: assms(1) eval: ZeroExtendNode)
    apply (solve-subset-eval as-set: assms(1) eval: LeafNode)
  by (solve-subset-eval as-set: assms(1) eval: RefNode)

```

lemma *subset-refines*:

assumes $as\text{-}set\ g1 \subseteq as\text{-}set\ g2$

shows *graph-refinement* $g1\ g2$

proof –

have $ids\ g1 \subseteq ids\ g2$ **using** *assms* **unfolding** *as-set-def*

by *blast*

then show *?thesis* **unfolding** *graph-refinement-def* **apply** *rule*

apply (*rule allI*) **apply** (*rule impI*) **apply** (*rule allI*) **apply** (*rule impI*)

unfolding *graph-represents-expression-def*

proof –

fix $n\ e1$

assume $1:n \in ids\ g1$

assume $2:g1 \vdash n \simeq e1$

show $\exists e2. (g2 \vdash n \simeq e2) \wedge e1 \geq e2$

using *assms* $1\ 2$ **using** *subset-implies-evals*

by (*meson equal-refines*)

qed

qed

lemma *graph-construction*:

$e_1 \geq e_2$

$\wedge as\text{-}set\ g1 \subseteq as\text{-}set\ g2$

$\wedge (g2 \vdash n \simeq e2)$

$\implies (g2 \vdash n \trianglelefteq e1) \wedge graph\text{-}refinement\ g1\ g2$

using *subset-refines*

by (*meson encodeeval-def graph-represents-expression-def le-expr-def*)

6.8.4 Term Graph Reconstruction

lemma *find-exists-kind*:

assumes *find-node-and-stamp* $g\ (node, s) = Some\ nid$

shows *kind* $g\ nid = node$

using *assms* **unfolding** *find-node-and-stamp.simps*

by (*metis* (*mono-tags*, *lifting*) *find-Some-iff*)

lemma *find-exists-stamp*:

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma fresh: finite  $xs \implies last(sorted-list-of-set(xs::nat\ set)) + 1 \notin xs$ 
  using sorted-bottom
  using not-le by auto

```

```

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

```

```

lemma graph-unchanged-rep-unchanged:
  assumes  $\forall n \in ids\ g. kind\ g\ n = kind\ g'\ n$ 
  assumes  $\forall n \in ids\ g. stamp\ g\ n = stamp\ g'\ n$ 
  shows  $(g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$ 
  apply (rule impI) subgoal premises  $e$  using  $e$  assms
    apply (induction  $n\ e$ )
      apply (metis no-encoding rep.ConstantNode)

```

```

    apply (metis no-encoding rep.ParameterNode)
    apply (metis no-encoding rep.ConditionalNode)
    apply (metis no-encoding rep.AbsNode)
    apply (metis no-encoding rep.NotNode)
    apply (metis no-encoding rep.NegateNode)
    apply (metis no-encoding rep.LogicNegationNode)
    apply (metis no-encoding rep.AddNode)
    apply (metis no-encoding rep.MulNode)
    apply (metis no-encoding rep.SubNode)
    apply (metis no-encoding rep.AndNode)
    apply (metis no-encoding rep.OrNode)
    apply (metis no-encoding rep.XorNode)
    apply (metis no-encoding rep.ShortCircuitOrNode)
    apply (metis no-encoding rep.LeftShiftNode)
    apply (metis no-encoding rep.RightShiftNode)
    apply (metis no-encoding rep.UnsignedRightShiftNode)
    apply (metis no-encoding rep.IntegerBelowNode)
    apply (metis no-encoding rep.IntegerEqualsNode)
    apply (metis no-encoding rep.IntegerLessThanNode)
    apply (metis no-encoding rep.NarrowNode)
    apply (metis no-encoding rep.SignExtendNode)
    apply (metis no-encoding rep.ZeroExtendNode)
    apply (metis no-encoding rep.LeafNode)
    by (metis no-encoding rep.RefNode)
done

```

lemma *fresh-node-subset*:

```

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

```

lemma *unrep-subset*:

```

  assumes  $(g \oplus e \rightsquigarrow (g', n))$ 
  shows  $\text{as-set } g \subseteq \text{as-set } g'$ 
  using assms proof (induction  $g \ e \ (g', n)$  arbitrary:  $g' \ n$ )
  case (ConstantNodeSame  $g \ c \ n$ )
  then show ?case by blast
next
  case (ConstantNodeNew  $g \ c \ n \ g'$ )
  then show ?case using fresh-ids fresh-node-subset
    by presburger
next
  case (ParameterNodeSame  $g \ i \ s \ n$ )
  then show ?case by blast
next
  case (ParameterNodeNew  $g \ i \ s \ n \ g'$ )

```



```

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

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

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

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

lemma unrep-unchanged:

```

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

theorem *term-graph-reconstruction:*

$g \oplus e \rightsquigarrow (g', n) \implies (g' \vdash n \simeq e) \wedge \text{as-set } g \subseteq \text{as-set } g'$
subgoal premises e **apply** (*rule conjI*) **defer**
 using e *unrep-subset* **apply** *blast* **using** e
proof (*induction g e (g', n) arbitrary: g' n*)
 case (*ConstantNodeSame g' c n*)
 then have $\text{kind } g' n = \text{ConstantNode } c$
 using *find-exists-kind local.ConstantNodeSame* **by** *blast*
 then show $?case$ **using** *ConstantNode* **by** *blast*
next
 case (*ConstantNodeNew g c*)
 then show $?case$
 using *ConstantNode IRNode.distinct(683) add-node-lookup* **by** *presburger*
next
 case (*ParameterNodeSame i s*)
 then show $?case$
 by (*metis ParameterNode find-exists-kind find-exists-stamp*)
next
 case (*ParameterNodeNew g i s*)
 then show $?case$
 by (*metis IRNode.distinct(2447) ParameterNode add-node-lookup*)
next
 case (*ConditionalNodeSame g4 c t f s' n g ce g2 te g3 fe*)
 then have k : $\text{kind } g4 n = \text{ConditionalNode } c t f$
 using *find-exists-kind* **by** *blast*
 have c : $g4 \vdash c \simeq ce$ **using** *local.ConditionalNodeSame unrep-unchanged*
 using *no-encoding* **by** *blast*
 have t : $g4 \vdash t \simeq te$ **using** *local.ConditionalNodeSame unrep-unchanged*
 using *no-encoding* **by** *blast*
 have f : $g4 \vdash f \simeq fe$ **using** *local.ConditionalNodeSame unrep-unchanged*
 using *no-encoding* **by** *blast*
 then show $?case$ **using** $c t f$
 using *ConditionalNode k* **by** *blast*
next
 case (*ConditionalNodeNew g4 c t f s' g ce g2 te g3 fe n g'*)
 moreover have *ConditionalNode c t f* $\neq \text{NoNode}$
 using *unary-node.elims* **by** *blast*
 ultimately have k : $\text{kind } g' n = \text{ConditionalNode } c t f$
 using *find-new-kind local.ConditionalNodeNew*
 by *presburger*
 then have c : $g' \vdash c \simeq ce$ **using** *local.ConditionalNodeNew unrep-unchanged*
 using *no-encoding*
 by (*metis ConditionalNodeNew.hyps(9) fresh-node-preserves-other-nodes*)
 then have t : $g' \vdash t \simeq te$ **using** *local.ConditionalNodeNew unrep-unchanged*

```

    using no-encoding fresh-node-preserves-other-nodes
    by metis
  then have  $f: g' \vdash f \simeq fe$  using local.ConditionalNodeNew unrep-unchanged
    using no-encoding fresh-node-preserves-other-nodes
    by metis
  then show  $?case$  using  $c \ t \ f$ 
    using ConditionalNode k by blast
next
case (UnaryNodeSame  $g' \ op \ x \ s' \ n \ g \ xe$ )
then have  $k: kind \ g' \ n = unary-node \ op \ x$ 
  using find-exists-kind local.UnaryNodeSame by blast
then have  $g' \vdash x \simeq xe$  using local.UnaryNodeSame by blast
then show  $?case$  using  $k$ 
  apply (cases op)
  using AbsNode unary-node.simps(1) apply presburger
  using NegateNode unary-node.simps(3) apply presburger
  using NotNode unary-node.simps(2) apply presburger
  using LogicNegationNode unary-node.simps(4) apply presburger
  using NarrowNode unary-node.simps(5) apply presburger
  using SignExtendNode unary-node.simps(6) apply presburger
  using ZeroExtendNode unary-node.simps(7) by presburger
next
case (UnaryNodeNew  $g2 \ op \ x \ s' \ g \ xe \ n \ g'$ )
moreover have  $unary-node \ op \ x \neq NoNode$ 
  using unary-node.elims by blast
ultimately have  $k: kind \ g' \ n = unary-node \ op \ x$ 
  using find-new-kind local.UnaryNodeNew
  by presburger
have  $x \in ids \ g2$  using local.UnaryNodeNew
  using eval-contains-id by blast
then have  $x \neq n$  using local.UnaryNodeNew(5) fresh-ids by blast
have  $g' \vdash x \simeq xe$  using local.UnaryNodeNew fresh-node-preserves-other-nodes
  using  $\langle x \in ids \ g2 \rangle$  by blast
then show  $?case$  using  $k$ 
  apply (cases op)
  using AbsNode unary-node.simps(1) apply presburger
  using NegateNode unary-node.simps(3) apply presburger
  using NotNode unary-node.simps(2) apply presburger
  using LogicNegationNode unary-node.simps(4) apply presburger
  using NarrowNode unary-node.simps(5) apply presburger
  using SignExtendNode unary-node.simps(6) apply presburger
  using ZeroExtendNode unary-node.simps(7) by presburger
next
case (BinaryNodeSame  $g3 \ op \ x \ y \ s' \ n \ g \ xe \ g2 \ ye$ )
then have  $k: kind \ g3 \ n = bin-node \ op \ x \ y$ 
  using find-exists-kind by blast
have  $x: g3 \vdash x \simeq xe$  using local.BinaryNodeSame unrep-unchanged
  using no-encoding by blast
have  $y: g3 \vdash y \simeq ye$  using local.BinaryNodeSame unrep-unchanged

```

```

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

```

lemma *ref-refinement*:

assumes $g \vdash n \simeq e_1$
assumes $\text{kind } g \ n' = \text{RefNode } n$
shows $g \vdash n' \sqsubseteq e_1$
using *assms RefNode*
by (*meson equal-refines graph-represents-expression-def*)

lemma *unrep-refines*:

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

lemma *add-new-node-refines*:

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

lemma *add-node-as-set*:

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

theorem *refined-insert*:

assumes $e_1 \geq e_2$
assumes $g_1 \oplus e_2 \rightsquigarrow (g_2, n')$
shows $(g_2 \vdash n' \sqsubseteq e_1) \wedge \text{graph-refinement } g_1 \ g_2$
using *assms*
using *graph-construction term-graph-reconstruction* **by** *blast*

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

using *Rep-IRGraph ids.rep-eq* **by** *simp*

lemma *unwrap-sorted*: *set* (*sorted-list-of-set* (*ids* g)) = *ids* g

using *Rep-IRGraph set-sorted-list-of-set ids-finite*

by *blast*

lemma *find-none*:

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

proof –

have $(\nexists n. n \in \text{ids } g \wedge (\text{kind } g \ n = k \wedge \text{stamp } g \ n = s))$

using *assms unfolding find-node-and-stamp.simps* **using** *find-None-iff un-*

```

wrap-sorted
  by (metis (mono-tags, lifting))
then show ?thesis
  by blast
qed

```

```

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

```

6.8.5 Data-flow Tree to Subgraph Preserves Maximal Sharing

```

lemma same-kind-stamp-encodes-equal:
  assumes kind g n = kind g n'
  assumes stamp g n = stamp g n'
  assumes  $\neg(\text{is-preevaluated } (\text{kind } g \ n))$ 
  shows  $\forall e. (g \vdash n \simeq e) \longrightarrow (g \vdash n' \simeq e)$ 
  apply (rule allI)
  subgoal for e
    apply (rule impI)
    subgoal premises eval using eval assms
      apply (induction e)
    using ConstantNode apply presburger
    using ParameterNode apply presburger
      apply (metis ConditionalNode)
      apply (metis AbsNode)
      apply (metis NotNode)
      apply (metis NegateNode)
      apply (metis LogicNegationNode)
      apply (metis AddNode)
      apply (metis MulNode)
      apply (metis SubNode)
      apply (metis AndNode)
      apply (metis OrNode)
      apply (metis XorNode)
      apply (metis ShortCircuitOrNode)
      apply (metis LeftShiftNode)

```

```

      apply (metis RightShiftNode)
      apply (metis UnsignedRightShiftNode)
      apply (metis IntegerBelowNode)
      apply (metis IntegerEqualsNode)
      apply (metis IntegerLessThanNode)
      apply (metis NarrowNode)
      apply (metis SignExtendNode)
      apply (metis ZeroExtendNode)
    defer
      apply (metis RefNode)
    by blast
  done
done

```

lemma *new-node-not-present*:

```

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

```

lemma *true-ids-def*:

```

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

```

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

```

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

```

lemma *ids-add-update-v1*:

```

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

```

lemma *ids-add-update-v2*:

```

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

```

lemma *add-node-ids-subset*:

assumes $n \in \text{ids } g$
assumes $g' = \text{add-node } n \text{ node } g$
shows $\text{ids } g' = \text{ids } g \cup \{n\}$
using *assms* **unfolding** *add-node-def*
apply (*cases fst node = NoNode*)
using *ids.rep-eq* *replace-node.rep-eq* *replace-node-def* **apply** *auto[1]*
unfolding *ids-def*
by (*smt (verit, best) Collect-cong Un-insert-right dom-fun-upd fst-conv fun-upd-apply*
ids.rep-eq ids-def insert-absorb mem-Collect-eq option.inject option.simps(3) re-
place-node.rep-eq replace-node-def sup-bot.right-neutral)

lemma *convert-maximal:*

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

lemma *add-node-set-eq:*

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

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

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

lemma *true-ids:*

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

lemma *as-set-ids:*

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

lemma *ids-add-update:*

assumes $k \neq \text{NoNode}$
assumes $n \notin \text{ids } g$


```

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

```

lemma *true-ids-add-update*:

```

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

```

lemma *new-def*:

```

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

```

lemma *add-preserves-rep*:

```

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

```

```

show ?thesis using assms(4) kind-eq stamp-eq False
proof (induction n e rule: rep.induct)
  case (ConstantNode n c)
  then show ?case
    using rep.ConstantNode kind-eq by presburger
next
  case (ParameterNode n i s)
  then show ?case
    using rep.ParameterNode
    by (metis no-encoding)
next
  case (ConditionalNode n c t f ce te fe)
  have kind: kind g n = ConditionalNode c t f
    using ConditionalNode.hyps(1) ConditionalNode.prem(3) kind-eq by pres-
    burger
  then have isin:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{c, t, f\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps using inputs-of-ConditionalNode by simp
  have  $c \in \text{ids } g \wedge t \in \text{ids } g \wedge f \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $c \notin \text{new} \wedge t \notin \text{new} \wedge f \notin \text{new}$ 
    using new-def unchanged by blast
  then show ?case using ConditionalNode apply simp
    using rep.ConditionalNode by presburger
next
  case (AbsNode n x xe)
  then have kind: kind g n = AbsNode x
    by simp
  then have isin:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps by simp
  have  $x \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $x \notin \text{new}$ 
    using new-def unchanged by blast
  then show ?case
    using AbsNode
    using rep.AbsNode by presburger
next
  case (NotNode n x xe)
  then have kind: kind g n = NotNode x
    by simp
  then have isin:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x\} = \text{inputs } g \ n$ 

```

```

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

```

```

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

```

```

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

```

```

then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using LeftShiftNode
  using rep.LeftShiftNode by presburger
next
case (RightShiftNode  $n \ x \ y \ x_e \ y_e$ )
then have kind:  $\text{kind } g \ n = \text{RightShiftNode } x \ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using RightShiftNode
  using rep.RightShiftNode by presburger
next
case (UnsignedRightShiftNode  $n \ x \ y \ x_e \ y_e$ )
then have kind:  $\text{kind } g \ n = \text{UnsignedRightShiftNode } x \ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using UnsignedRightShiftNode
  using rep.UnsignedRightShiftNode by presburger
next
case (IntegerBelowNode  $n \ x \ y \ x_e \ y_e$ )
then have kind:  $\text{kind } g \ n = \text{IntegerBelowNode } x \ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp

```

```

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

```

```

    using new-def unchanged by blast
  then show ?case using NarrowNode
    using rep.NarrowNode by presburger
next
case (SignExtendNode n inputBits resultBits x xe)
then have kind: kind g n = SignExtendNode inputBits resultBits x
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using SignExtendNode
  using rep.SignExtendNode by presburger
next
case (ZeroExtendNode n inputBits resultBits x xe)
then have kind: kind g n = ZeroExtendNode inputBits resultBits x
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using ZeroExtendNode
  using rep.ZeroExtendNode by presburger
next
case (LeafNode n s)
then show ?case
  by (metis no-encoding rep.LeanNode)
next
case (RefNode n n' e)
then have kind: kind g n = RefNode n'
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{n'\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $n' \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $n' \notin \text{new}$ 

```



```

      using new-def unchanged by blast
    then show ?case
      using RefNode
      using rep.RefNode by presburger
  qed
qed

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma unrep-preserves-closure:
  assumes wf-closed g
  assumes  $g \oplus e \rightsquigarrow (g', n)$ 

```

```

shows wf-closed g'
using assms(2,1) unfolding wf-closed-def
proof (induction g e (g', n) arbitrary: g' n)
  case (ConstantNodeSame g c n)
  then show ?case
    by blast
next
  case (ConstantNodeNew g c n g')
  then have dom:  $ids\ g' = ids\ g \cup \{n\}$ 
    by (meson IRNode.distinct(683) add-node-ids-subset ids-add-update)
  have k:  $kind\ g'\ n = ConstantNode\ c$ 
    using ConstantNodeNew add-node-lookup by simp
  then have inp:  $\{\} = inputs\ g'\ n$ 
    unfolding inputs.simps by simp
  from k have suc:  $\{\} = succ\ g'\ n$ 
    unfolding succ.simps by simp
  have  $inputs\ g'\ n \subseteq ids\ g' \wedge succ\ g'\ n \subseteq ids\ g' \wedge kind\ g'\ n \neq NoNode$ 
    using inp suc k by simp
  then show ?case
    by (smt (verit) ConstantNodeNew.hyps(3) ConstantNodeNew.prem Un-insert-right
    add-changed changeonly.elims(2) dom inputs.simps insert-iff singleton-iff subset-insertI
    subset-trans succ.simps sup-bot-right)
next
  case (ParameterNodeSame g i s n)
  then show ?case by blast
next
  case (ParameterNodeNew g i s n g')
  then have dom:  $ids\ g' = ids\ g \cup \{n\}$ 
    using IRNode.distinct(2447) fresh-ids ids-add-update by presburger
  have k:  $kind\ g'\ n = ParameterNode\ i$ 
    using ParameterNodeNew add-node-lookup by simp
  then have inp:  $\{\} = inputs\ g'\ n$ 
    unfolding inputs.simps by simp
  from k have suc:  $\{\} = succ\ g'\ n$ 
    unfolding succ.simps by simp
  have  $inputs\ g'\ n \subseteq ids\ g' \wedge succ\ g'\ n \subseteq ids\ g' \wedge kind\ g'\ n \neq NoNode$ 
    using k inp suc by simp
  then show ?case
    by (smt (verit) ParameterNodeNew.hyps(3) ParameterNodeNew.prem Un-insert-right
    add-node-as-set dom inputs.elims insertE not-excluded-keep-type order-trans single-
    tonD subset-insertI succ.elims sup-bot-right)
next
  case (ConditionalNodeSame g ce g2 c te g3 t fe g4 f s' n)
  then show ?case by blast
next
  case (ConditionalNodeNew g4 c t f s' g ce g2 te g3 fe n g')
  then have dom:  $ids\ g' = ids\ g4 \cup \{n\}$ 
    by (meson IRNode.distinct(591) add-node-ids-subset ids-add-update)
  have k:  $kind\ g'\ n = ConditionalNode\ c\ t\ f$ 

```

```

    using ConditionalNodeNew add-node-lookup by simp
  then have inp:  $\{c, t, f\} = \text{inputs } g' n$ 
    unfolding inputs.simps by simp
  from k have suc:  $\{\} = \text{succ } g' n$ 
    unfolding succ.simps by simp
  have inputs  $g' n \subseteq \text{ids } g' \wedge \text{succ } g' n \subseteq \text{ids } g' \wedge \text{kind } g' n \neq \text{NoNode}$ 
    using k inp suc unrep-contains unrep-preserves-contains
    using ConditionalNodeNew
    by (smt (verit) IRNode.simps(643) Un-insert-right bot.extremum dom insert-absorb insert-subset subset-insertI sup-bot-right)
  then show ?case using dom
    by (smt (z3) ConditionalNodeNew.hyps ConditionalNodeNew.premis Diff-eq-empty-iff
    Diff-iff Un-insert-right Un-upper1 add-node-def inputs.simps insertE replace-node-def
    replace-node-unchanged subset-trans succ.simps sup-bot-right)
  next
    case (UnaryNodeSame g xe g2 x s' op n)
    then show ?case by blast
  next
    case (UnaryNodeNew g2 op x s' g xe n g')
    then have dom:  $\text{ids } g' = \text{ids } g2 \cup \{n\}$ 
      by (metis add-node-ids-subset add-node-lookup ids-add-update ids-some unrep.UnaryNodeNew unrep-contains)
    have k:  $\text{kind } g' n = \text{unary-node op } x$ 
      using UnaryNodeNew add-node-lookup
      by (metis fresh-ids ids-some)
    then have inp:  $\{x\} = \text{inputs } g' n$ 
      using unary-inputs by simp
    from k have suc:  $\{\} = \text{succ } g' n$ 
      using unary-succ by simp
    have inputs  $g' n \subseteq \text{ids } g' \wedge \text{succ } g' n \subseteq \text{ids } g' \wedge \text{kind } g' n \neq \text{NoNode}$ 
      using k inp suc unrep-contains unrep-preserves-contains
      using UnaryNodeNew
      by (metis Un-upper1 dom empty-subsetI ids-some insert-not-empty insert-subsetI not-in-g-inputs subset-iff)
    then show ?case
      by (smt (verit) Un-insert-right UnaryNodeNew.hyps UnaryNodeNew.premis add-changed changeonly.elims(2) dom inputs.simps insert-iff singleton-iff subset-insertI subset-trans succ.simps sup-bot-right)
  next
    case (BinaryNodeSame g xe g2 x ye g3 y s' op n)
    then show ?case by blast
  next
    case (BinaryNodeNew g3 op x y s' g xe g2 ye n g')
    then have dom:  $\text{ids } g' = \text{ids } g3 \cup \{n\}$ 
      by (metis binary-inputs fresh-ids ids-add-update ids-some insert-not-empty not-in-g-inputs)
    have k:  $\text{kind } g' n = \text{bin-node op } x y$ 
      using BinaryNodeNew add-node-lookup
      by (metis fresh-ids ids-some)

```

```

then have inp:  $\{x, y\} = \text{inputs } g' \ n$ 
  using binary-inputs by simp
from k have suc:  $\{\} = \text{succ } g' \ n$ 
  using binary-succ by simp
have  $\text{inputs } g' \ n \subseteq \text{ids } g' \wedge \text{succ } g' \ n \subseteq \text{ids } g' \wedge \text{kind } g' \ n \neq \text{NoNode}$ 
  using k inp suc unrep-contains unrep-preserves-contains
  using BinaryNodeNew
  by (metis Un-upper1 dom empty-subsetI ids-some insert-not-empty insert-subsetI
not-in-g-inputs subset-iff)
  then show ?case using dom BinaryNodeNew
    by (smt (verit, del-ists) Diff-eq-empty-iff Diff-iff Un-insert-right Un-upper1
add-node-def inputs.simps insertE replace-node-def replace-node-unchanged subset-trans
succ.simps sup-bot-right)
  next
    case (AllLeafNodes g n s)
    then show ?case
      by blast
qed

```

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

definition *constant-value* **where**

constant-value = (*IntVal* 32 0)

definition *bad-graph* **where**

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

end

7 Control-flow Semantics

theory *IRStepObj*

imports

TreeToGraph

begin

7.1 Object Heap

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

heapdef

```

type-synonym ('a, 'b) Heap = 'a  $\Rightarrow$  'b  $\Rightarrow$  Value
type-synonym Free = nat
type-synonym ('a, 'b) DynamicHeap = ('a, 'b) Heap  $\times$  Free

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

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

fun h-new-inst :: ('a, 'b) DynamicHeap  $\Rightarrow$  ('a, 'b) DynamicHeap  $\times$  Value
where
  h-new-inst (h, n) = ((h, n+1), (ObjRef (Some n)))

type-synonym FieldRefHeap = (string, objref) DynamicHeap

```

```

definition new-heap :: ('a, 'b) DynamicHeap where
  new-heap = (( $\lambda$ f.  $\lambda$ p. UndefVal), 0)

```

7.2 Intraprocedural Semantics

```

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

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

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

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

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

```

Intraprocedural semantics are given as a small-step semantics.

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

```

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

```

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

SequentialNode:

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

IfNode:

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

EndNodes:

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

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

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

NewInstanceNode:

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

LoadFieldNode:

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

SignedDivNode:

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

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

SignedRemNode:

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

StaticLoadFieldNode:

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

StoreFieldNode:

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

StaticStoreFieldNode:

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

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

7.3 Interprocedural Semantics

type-synonym *Signature* = *string*

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

inductive *step-top* :: *Program* \Rightarrow (*IRGraph* \times *ID* \times *MapState* \times *Params*) *list* \times *FieldRefHeap* \Rightarrow (*IRGraph* \times *ID* \times *MapState* \times *Params*) *list* \times *FieldRefHeap* \Rightarrow

bool

$(- \vdash - \longrightarrow - \ 55)$

for P where

Lift:

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

InvokeNodeStep:

$\llbracket is_Invoke \ (kind \ g \ nid);$

$callTarget = ir_callTarget \ (kind \ g \ nid);$

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

$Some \ targetGraph = P \ targetMethod;$

$m' = new_map_state;$

$g \vdash arguments \simeq_L argsE;$

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

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

|

ReturnNode:

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

$g \vdash expr \simeq e;$

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

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

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

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

|

ReturnNodeVoid:

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

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

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

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

|

UnwindNode:

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

$g \vdash exception \simeq exceptionE;$

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

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

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

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

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

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

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

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

7.4.1 Heap Testing

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

p3 = [*IntVal* 32 3]

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

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

field-sq = "sq"

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

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

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

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

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

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

end

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*:

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

proof (*induction rule: step.induct*)

```

case (SequentialNode nid next m h)
have notif:  $\neg(is\_IfNode\ (kind\ g\ nid))$ 
  using SequentialNode.hyps(1) is-sequential-node.simps
  by (metis is-IfNode-def)
have notend:  $\neg(is\_AbstractEndNode\ (kind\ g\ nid))$ 
  using SequentialNode.hyps(1) is-sequential-node.simps
  by (metis is-AbstractEndNode.simps is-EndNode.elims(2) is-LoopEndNode-def)
have notnew:  $\neg(is\_NewInstanceNode\ (kind\ g\ nid))$ 
  using SequentialNode.hyps(1) is-sequential-node.simps
  by (metis is-NewInstanceNode-def)
have notload:  $\neg(is\_LoadFieldNode\ (kind\ g\ nid))$ 
  using SequentialNode.hyps(1) is-sequential-node.simps
  by (metis is-LoadFieldNode-def)
have notstore:  $\neg(is\_StoreFieldNode\ (kind\ g\ nid))$ 
  using SequentialNode.hyps(1) is-sequential-node.simps
  by (metis is-StoreFieldNode-def)
have notdivrem:  $\neg(is\_IntegerDivRemNode\ (kind\ g\ nid))$ 
  using SequentialNode.hyps(1) is-sequential-node.simps is-SignedDivNode-def
  is-SignedRemNode-def
  by (metis is-IntegerDivRemNode.simps)
from notif notend notnew notload notstore notdivrem
show ?case using SequentialNode step.cases
  by (smt (z3) IRNode.disc(1028) IRNode.disc(2270) IRNode.discI(31) Pair-inject
  is-sequential-node.simps(18) is-sequential-node.simps(43) is-sequential-node.simps(44))
next
case (IfNode nid cond tb fb m val next h)
then have notseq:  $\neg(is\_sequential-node\ (kind\ g\ nid))$ 
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: IfNode.hyps(1))
have notend:  $\neg(is\_AbstractEndNode\ (kind\ g\ nid))$ 
  using is-AbstractEndNode.simps
  by (simp add: IfNode.hyps(1))
have notdivrem:  $\neg(is\_IntegerDivRemNode\ (kind\ g\ nid))$ 
  using is-AbstractEndNode.simps

```

```

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

```

```

have notref: ¬(is-RefNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notload: ¬(is-LoadFieldNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notstore: ¬(is-StoreFieldNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
from notseq notend notif notref notload notstore notdivrem
show ?case using NewInstanceNode.step.cases
  by (smt (z3) IRNode.disc(1028) IRNode.disc(2270) IRNode.discI(11) IRN-
ode.distinct(2311) IRNode.distinct(2313) IRNode.inject(31) Pair-inject)
next
case (LoadFieldNode nid f obj nrt m ref h v m')
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: LoadFieldNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: LoadFieldNode.hyps(1))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: LoadFieldNode.hyps(1))
from notseq notend notdivrem
show ?case using LoadFieldNode.step.cases repDet evalDet
  by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject Value.inject(2)
option.distinct(1) option.inject)
next
case (StaticLoadFieldNode nid f nrt h v m' m)
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: StaticLoadFieldNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: StaticLoadFieldNode.hyps(1))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  by (simp add: StaticLoadFieldNode.hyps(1))
from notseq notend notdivrem
show ?case using StaticLoadFieldNode.step.cases
  by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject option.distinct(1))
next
case (StoreFieldNode nid f newval uu obj nrt m val ref h' h m')
then have notseq: ¬(is-sequential-node (kind g nid))

```

```

    using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: StoreFieldNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: StoreFieldNode.hyps(1))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  by (simp add: StoreFieldNode.hyps(1))
from notseq notend notdivrem
show ?case using StoreFieldNode step.cases repDet evalDet
  by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Value.inject(2)
option.distinct(1) option.inject)
next
case (StaticStoreFieldNode nid f newval uv nxt m val h' h m')
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: StaticStoreFieldNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: StaticStoreFieldNode.hyps(1))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  by (simp add: StaticStoreFieldNode.hyps(1))
from notseq notend notdivrem
show ?case using StoreFieldNode step.cases repDet evalDet
  by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Stat-
icStoreFieldNode.hyps(1) StaticStoreFieldNode.hyps(2) StaticStoreFieldNode.hyps(3)
StaticStoreFieldNode.hyps(4) StaticStoreFieldNode.hyps(5) option.distinct(1))
next
case (SignedDivNode nid x y zero sb nxt m v1 v2 v m' h)
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: SignedDivNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: SignedDivNode.hyps(1))
from notseq notend
show ?case using SignedDivNode step.cases repDet evalDet
  by (smt (z3) IRNode.distinct(1091) IRNode.distinct(1739) IRNode.distinct(2311)
IRNode.distinct(2601) IRNode.distinct(2605) IRNode.inject(40) Pair-inject)
next
case (SignedRemNode nid x y zero sb nxt m v1 v2 v m' h)
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: SignedRemNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: SignedRemNode.hyps(1))
from notseq notend

```

```

show ?case using SignedRemNode step.cases repDet evalDet
by (smt (z3) IRNode.distinct(1093) IRNode.distinct(1741) IRNode.distinct(2313)
IRNode.distinct(2601) IRNode.distinct(2627) IRNode.inject(41) Pair-inject)
qed

```

```

lemma stepRefNode:
   $\llbracket \text{kind } g \text{ nid} = \text{RefNode nid}' \rrbracket \implies g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m, h)$ 
using SequentialNode
by (metis IRNodes.successors-of-RefNode is-sequential-node.simps(7) nth-Cons-0)

```

```

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

```

```

lemma IfNodeSeq:
  shows kind g nid = IfNode cond tb fb  $\longrightarrow \neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
unfolding is-sequential-node.simps
using is-sequential-node.simps(18) by presburger

```

```

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

```

```

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

```

end

7.6 Evaluation Stamp Theorems

```

theory StampEvalThms
imports Graph.ValueThms

```

begin

lemma

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

lemma *unwrap-signed-take-bit*:

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

lemma *unrestricted-new-int-always-valid* [*simp*]:

assumes $0 < b \wedge b \leq 64$
shows *valid-value* (*new-int* *b v*) (*unrestricted-stamp* (*IntegerStamp* *b lo hi*))
unfolding *unrestricted-stamp.simps* *new-int.simps* *valid-value.simps*
by (*simp*; *metis* *One-nat-def* *assms* *int-power-div-base* *int-signed-value.simps* *int-signed-value-range* *linorder-not-le* *not-exp-less-eq-0-int* *zero-less-numeral*)

lemma *unary-undef*: *val = UndefVal* \implies *unary-eval op val = UndefVal*

by (*cases op*; *auto*)

lemma *unary-obj*: *val = ObjRef x* \implies *unary-eval op val = UndefVal*

by (*cases op*; *auto*)

lemma *unrestricted-stamp-valid*:

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

lemma *unrestricted-stamp-valid-value* [*simp*]:

assumes 1: *result = IntVal* *b ival*
assumes *take-bit* *b ival = ival*
assumes $0 < b \wedge b \leq 64$
shows *valid-value* *result* (*unrestricted-stamp* (*IntegerStamp* *b lo hi*))
proof –
have *valid-stamp* (*unrestricted-stamp* (*IntegerStamp* *b lo hi*))
using *assms* *unrestricted-stamp-valid* **by** *blast*
then show ?thesis
unfolding 1 *unrestricted-stamp.simps* *valid-value.simps*

using *assms int-signed-value-bounds* by *presburger*
 qed

7.6.1 Support Lemmas for Integer Stamps and Associated IntVal values

Valid int implies some useful facts.

lemma *valid-int-gives*:
 assumes *valid-value* (*IntVal b val*) *stamp*
 obtains *lo hi* where *stamp* = *IntegerStamp b lo hi* \wedge
 valid-stamp (*IntegerStamp b lo hi*) \wedge
 take-bit b val = *val* \wedge
 $lo \leq \text{int-signed-value } b \text{ val} \wedge \text{int-signed-value } b \text{ val} \leq hi$
 using *assms*
 by (*smt (z3) Value.distinct(7) Value.inject(1) valid-value.elims(1)*)

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

lemma *valid-int-stamp-gives*:
 assumes *valid-value val* (*IntegerStamp b lo hi*)
 obtains *ival* where *val* = *IntVal b ival* \wedge
 valid-stamp (*IntegerStamp b lo hi*) \wedge
 take-bit b ival = *ival* \wedge
 $lo \leq \text{int-signed-value } b \text{ ival} \wedge \text{int-signed-value } b \text{ ival} \leq hi$
 by (*metis assms valid-int valid-value.simps(1)*)

A valid int must have the expected number of bits.

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

A valid value means a valid stamp.

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

A valid int means a valid non-empty stamp.

lemma *valid-int-not-empty*:
 assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)
 shows $lo \leq hi$
 by (*metis assms order.trans valid-value.simps(1)*)

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

lemma *valid-int-fits*:
 assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)

shows *take-bit bits val = val*
by (*metis assms valid-value.simps(1)*)

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

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

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

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

Signed ints have the usual two-complement bounds.

lemma *valid-int-signed-upper-bound*:
assumes *valid-value (IntVal b val) (IntegerStamp bits lo hi)*
shows *int-signed-value bits val < 2^(bits - 1)*
by (*metis (mono-tags, opaque-lifting) diff-le-mono int-signed-value.simps less-imp-diff-less*)

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

lemma *valid-int-signed-lower-bound*:
assumes *valid-value (IntVal b val) (IntegerStamp bits lo hi)*
shows *-(2^(bits - 1)) ≤ int-signed-value bits val*
by (*smt (verit) diff-le-self int-signed-value.simps linorder-not-less power-increasing-iff signed-take-bit-int-greater-eq-minus-exp-word sint-greater-eq*)

and *bit_bounds* versions of the above bounds.

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

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

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

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

qed

Valid values satisfy their stamp bounds.

lemma *valid-int-signed-range*:

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

7.6.2 Validity of all Unary Operators

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

lemma *eval-normal-unary-implies-valid-value*:

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

```

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

```

```

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

```

```

lemma eval-widen-narrow-unary-implies-valid-value:
  assumes [m,p] ⊢ expr ↦ val
  assumes result = unary-eval op val
  assumes op: op ∉ normal-unary
  assumes result ≠ UndefVal
  assumes valid-value val (stamp-expr expr)
  shows valid-value result (stamp-expr (UnaryExpr op expr))
proof –
  obtain b1 v1 where v1: val = IntVal b1 v1
  by (metis Value.exhaust assms(1) assms(2) assms(4) assms(5) evaltree-not-undef
unary-obj valid-value.simps(11))
  then have result = unary-eval op (IntVal b1 v1)
  using assms(2) v1 by blast
  then obtain v2 where v2: result = new-int (ir-resultBits op) v2
  using assms by (cases op; simp; (meson new-int.simps)+)

```

```

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

```

```

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

```

7.6.3 Support Lemmas for Binary Operators

```

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

```

```

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

```

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

```

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

```

```

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

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

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

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

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

```

```

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

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

```

7.6.4 Validity of Stamp Meet and Join Operators

```

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

```

```

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

```

```

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

```

```

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

```

and the symmetric lemma follows by the commutativity of meet.


```

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

```

7.6.5 Validity of conditional expressions

```

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

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

qed

```

7.6.6 Validity of Whole Expression Tree Evaluation

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

```

definition wf-stamp :: IRExpr ⇒ bool where
  wf-stamp e = (∀ m p v. ([m, p] ⊢ e ↦ v) ⟶ valid-value v (stamp-expr e))

```

```

lemma stamp-under-defn:
  assumes stamp-under (stamp-expr x) (stamp-expr y)
  assumes wf-stamp x ∧ wf-stamp y
  assumes ([m, p] ⊢ x ↦ xv) ∧ ([m, p] ⊢ y ↦ yv)
  shows val-to-bool (bin-eval BinIntegerLessThan xv yv)

```

```

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

```

```

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

```

```

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

end

```

8 Optimization DSL

8.1 Markup

```

theory Markup
  imports Semantics.IRTreeEval Snippets.Snipping
begin

```

```

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

```

```

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

```

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

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

ML-file <markup.ML>

8.1.1 Expression Markup

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

ir expression translation

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

ir expression example

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

ConditionalExpr (BinaryExpr BinIntegerLessThan e_1 e_2) e_1 e_2

8.1.2 Value Markup

ML <

structure IntValTranslator : DSL-TRANSLATION =

struct

```

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

```

end

structure IntValMarkup = DSL-Markup(IntValTranslator);

>

value expression translation

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

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

value expression example

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

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

8.1.3 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 shiftl}
  | markup DSL-Tokens.RightShift = @{term signed-shiftr}
  | markup DSL-Tokens.UnsignedRightShift = @{term shiftr}
  | markup DSL-Tokens.Constant = @{term word}
  | markup DSL-Tokens.TrueConstant = @{term 1}
  | markup DSL-Tokens.FalseConstant = @{term 0}
end
structure WordMarkup = DSL-Markup(WordTranslator);
 $\rangle$ 
```

word expression translation

syntax $\text{-expandWord} :: \text{term} \Rightarrow \text{term} \ (\text{bin}[-])$
parse-translation $\langle \ [(\ @\{\text{syntax-const} \ \text{-expandWord}\} \ , \ \text{Word-Markup.markup-expr} \ [])] \rangle$

word expression example

value $\text{bin}[x \ \& \ y \ | \ z]$
intval-conditional $(\text{intval-less-than } e_1 \ e_2) \ e_1 \ e_2$

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

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

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

```

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

value  $\sim x$ 

end

```

8.2 Optimization Phases

```

theory Phase
  imports Main
begin

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

end

```

8.3 Canonicalization DSL

```

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

print-methods

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

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

```

```

}

structure RewriteRule : Rule =
struct
type T = rewrite;

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

fun pretty-thm ctxt thm =
  (Proof-Context.pretty-fact ctxt (, [thm]))

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

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

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

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

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

```



```

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

structure RewritePhase = DSL-Phase(RewriteRule);

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

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

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

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

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

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

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

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

```

```

end

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

```

ML-file *rewrites.ML*

8.3.1 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

```

8.3.2 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

```

8.3.3 Standard Termination Measure

```

fun size :: IRExp  $\Rightarrow$  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

```

8.3.4 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

9 Canonicalization Optimizations

```

theory Common
imports

```

OptimizationDSL.Canonicalization
Semantics.IRTreeEvalThms

begin

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

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

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

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

lemma *size-flip-binary[size-simps]*:
 $\neg(\text{is-ConstantExpr } y) \longrightarrow \text{size } (\text{BinaryExpr op } (\text{ConstantExpr } x) \ y) > \text{size } (\text{BinaryExpr op } y \ (\text{ConstantExpr } x))$
by (*metis add-Suc not-less-eq order-less-asym plus-1-eq-Suc size.simps(11) size.simps(2)*
size-non-add)

lemma *size-binary-lhs-a[size-simps]*:
 $\text{size } (\text{BinaryExpr op } (\text{BinaryExpr op}' a \ b) \ c) > \text{size } a$
by (*metis add-lessD1 less-add-same-cancel1 pos2 size-binary-const size-non-add*)

lemma *size-binary-lhs-b[size-simps]*:
 $\text{size } (\text{BinaryExpr op } (\text{BinaryExpr op}' a \ b) \ c) > \text{size } b$
by (*metis IRExp.disc(42) One-nat-def add.left-commute add.right-neutral is-ConstantExpr-def*
less-add-Suc2 numeral-2-eq-2 plus-1-eq-Suc size.simps(11) size-binary-const size-non-add
size-non-const trans-less-add1)

lemma *size-binary-lhs-c[size-simps]*:
 $\text{size } (\text{BinaryExpr op } (\text{BinaryExpr op}' a \ b) \ c) > \text{size } c$
by (*metis IRExp.disc(42) add.left-commute add.right-neutral is-ConstantExpr-def*
less-Suc-eq numeral-2-eq-2 plus-1-eq-Suc size.simps(11) size-non-add size-non-const
trans-less-add2)

lemma *size-binary-rhs-a[size-simps]*:
 $\text{size } (\text{BinaryExpr op } c \ (\text{BinaryExpr op}' a \ b)) > \text{size } a$
by (*smt (verit, best) less-Suc-eq less-add-Suc2 less-add-same-cancel1 linorder-neqE-nat*
not-add-less1 order-less-trans pos2 size.simps(4) size-binary-const size-non-add)

```

lemma size-binary-rhs-b[size-simps]:
  size (BinaryExpr op c (BinaryExpr op' a b)) > size b
  by (metis add.left-commute add.right-neutral is-ConstantExpr-def lessI numeral-2-eq-2
plus-1-eq-Suc size.simps(11) size.simps(4) size-non-add trans-less-add2)

lemma size-binary-rhs-c[size-simps]:
  size (BinaryExpr op c (BinaryExpr op' a b)) > size c
  by simp

lemma size-binary-lhs[size-simps]:
  size (BinaryExpr op x y) > size x
  by (metis One-nat-def Suc-eq-plus1 add-Suc-right less-add-Suc1 numeral-2-eq-2
size-binary-const size-non-add)

lemma size-binary-rhs[size-simps]:
  size (BinaryExpr op x y) > size y
  by (metis IRExpr.disc(42) add-strict-increasing is-ConstantExpr-def linorder-not-le
not-add-less1 size.simps(11) size-non-add size-non-const size-pos)

lemmas arith[size-simps] = Suc-leI add-strict-increasing order-less-trans trans-less-add2

definition well-formed-equal :: Value  $\Rightarrow$  Value  $\Rightarrow$  bool
  (infix  $\approx$  50) where
    well-formed-equal  $v_1$   $v_2 = (v_1 \neq \text{UndefVal} \longrightarrow v_1 = v_2)$ 

lemma well-formed-equal-defn [simp]:
  well-formed-equal  $v_1$   $v_2 = (v_1 \neq \text{UndefVal} \longrightarrow v_1 = v_2)$ 
  unfolding well-formed-equal-def by simp

end

## 9.1 AddNode Phase

theory AddPhase
  imports
    Common
  begin

  phase AddNode
    terminating size
  begin

  lemma binadd-commute:
    assumes bin-eval BinAdd  $x$   $y \neq \text{UndefVal}$ 
    shows bin-eval BinAdd  $x$   $y = \text{bin-eval BinAdd } y$   $x$ 
    using assms intval-add-sym by simp

```

optimization *AddShiftConstantRight*: $((\text{const } v) + y) \mapsto y + (\text{const } v)$ when $\neg(\text{is-ConstantExpr } y)$
using *size-non-const*
apply (*metis add-2-eq-Suc' less-Suc-eq plus-1-eq-Suc size.simps(11) size-non-add*)
unfolding *le-expr-def*
apply (*rule impI*)
subgoal premises 1
apply (*rule allI impI*)
done
subgoal premises 2 for m p va
apply (*rule BinaryExprE[OF 2]*)
subgoal premises 3 for x ya
apply (*rule BinaryExpr*)
using 3 apply simp
using 3 apply simp
using 3 binadd-commute apply auto
done
done
done
done

optimization *AddShiftConstantRight2*: $((\text{const } v) + y) \mapsto y + (\text{const } v)$ when $\neg(\text{is-ConstantExpr } y)$
unfolding *le-expr-def*
apply (*auto simp: intval-add-sym*)
using size-non-const
by (*metis add-2-eq-Suc' lessI plus-1-eq-Suc size.simps(11) size-non-add*)

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

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

ML-val $\langle @\{term \ \langle x = y \rangle\} \rangle$

lemma *NeutralLeftSubVal*:
assumes $e1 = \text{new-int } b \text{ ival}$
shows $\text{val}[(e1 - e2) + e2] \approx e1$
apply *simp using assms by (cases e1; cases e2; auto)*

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

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

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

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

lemma *AddToSubHelperLowLevel*:
shows $\text{intval-add } (\text{intval-negate } e) y = \text{intval-sub } y e \text{ (is } ?x = ?y)$
by (*induction y; induction e; auto*)

print-phases

lemma *val-redundant-add-sub*:
assumes $a = \text{new-int } bb \text{ ival}$

```

assumes  $val[b + a] \neq \text{UndefVal}$ 
shows  $val[(b + a) - b] = a$ 
using assms apply (cases a; cases b; auto)
by presburger

```

```

lemma val-add-right-negate-to-sub:
assumes  $val[x + e] \neq \text{UndefVal}$ 
shows  $val[x + (-e)] = val[x - e]$ 
using assms by (cases x; cases e; auto)

```

```

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

```

Optimisations

```

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

```

```

optimization AddRightNegateToSub:  $x + -e \mapsto x - e$ 
apply (metis Nat.add-0-right add-2-eq-Suc' add-less-mono1 add-mono-thms-linordered-field(2)
less-SucI not-less-less-Suc-eq size-binary-const size-non-add size-pos)
using AddToSubHelperLowLevel intval-add-sym by auto

```

```

optimization AddLeftNegateToSub:  $-e + y \mapsto y - e$ 
apply (smt (verit, best) One-nat-def add commute add-Suc-right is-ConstantExpr-def
less-add-Suc2
numeral-2-eq-2 plus-1-eq-Suc size.simps(1) size.simps(11) size-binary-const
size-non-add)
using exp-add-left-negate-to-sub by blast

```

end

end

9.2 AndNode Phase

```

theory AndPhase
imports
  Common

```


Proofs.StampEvalThms
begin

context *stamp-mask*
begin

lemma *AndRightFallthrough*: $((\text{and } (\text{not } (\downarrow x)) (\uparrow y)) = 0) \longrightarrow \text{exp}[x \ \& \ y] \geq \text{exp}[y]$
apply *simp* **apply** (*rule impI*; (*rule allI*)+)
apply (*rule impI*)
subgoal *premises p* **for** *m p v*
proof –
 obtain *xv* **where** *xv*: $[m, p] \vdash x \mapsto xv$
 using *p(2)* **by** *blast*
 obtain *yv* **where** *yv*: $[m, p] \vdash y \mapsto yv$
 using *p(2)* **by** *blast*
 have $v = \text{val}[xv \ \& \ yv]$
 using *p(2)* *xv yv*
 by (*metis BinaryExprE bin-eval.simps(4) evalDet*)
 then **have** $v = yv$
 using *p(1)* *not-down-up-mask-and-zero-implies-zero*
 by (*smt (verit) eval-unused-bits-zero intval-and.elims new-int.elims new-int-bin.elims*
p(2)
 unfold-binary xv yv)
 then **show** *?thesis* **using** *yv* **by** *simp*
qed
done

lemma *AndLeftFallthrough*: $((\text{and } (\text{not } (\downarrow y)) (\uparrow x)) = 0) \longrightarrow \text{exp}[x \ \& \ y] \geq \text{exp}[x]$
apply *simp* **apply** (*rule impI*; (*rule allI*)+)
apply (*rule impI*)
subgoal *premises p* **for** *m p v*
proof –
 obtain *xv* **where** *xv*: $[m, p] \vdash x \mapsto xv$
 using *p(2)* **by** *blast*
 obtain *yv* **where** *yv*: $[m, p] \vdash y \mapsto yv$
 using *p(2)* **by** *blast*
 have $v = \text{val}[xv \ \& \ yv]$
 using *p(2)* *xv yv*
 by (*metis BinaryExprE bin-eval.simps(4) evalDet*)
 then **have** $v = xv$
 using *p(1)* *not-down-up-mask-and-zero-implies-zero*
 by (*smt (verit) and.commute eval-unused-bits-zero intval-and.elims new-int.simps*
 new-int-bin.simps p(2) unfold-binary xv yv)
 then **show** *?thesis* **using** *xv* **by** *simp*
qed
done

end

phase *AndNode*
 terminating *size*
begin

lemma *bin-and-nots*:
 $(\sim x \ \& \ \sim y) = (\sim (x \mid y))$
 by *simp*

lemma *bin-and-neutral*:
 $(x \ \& \ \sim \text{False}) = x$
 by *simp*

lemma *val-and-equal*:
 assumes $x = \text{new-int } b \ v$
 and $\text{val}[x \ \& \ x] \neq \text{UndefVal}$
 shows $\text{val}[x \ \& \ x] = x$
 using *assms* **by** (*cases* x ; *auto*)

lemma *val-and-nots*:
 $\text{val}[\sim x \ \& \ \sim y] = \text{val}[\sim (x \mid y)]$
 apply (*cases* x ; *cases* y ; *auto*) **by** (*simp add: take-bit-not-take-bit*)

lemma *val-and-neutral*:
 assumes $x = \text{new-int } b \ v$
 and $\text{val}[x \ \& \ \sim (\text{new-int } b' \ 0)] \neq \text{UndefVal}$
 shows $\text{val}[x \ \& \ \sim (\text{new-int } b' \ 0)] = x$
 using *assms* **apply** (*cases* x ; *auto*) **apply** (*simp add: take-bit-eq-mask*)
 by *presburger*

lemma *val-and-zero*:
 assumes $x = \text{new-int } b \ v$
 shows $\text{val}[x \ \& \ (\text{IntVal } b \ 0)] = \text{IntVal } b \ 0$
 using *assms* **by** (*cases* x ; *auto*)

lemma *exp-and-equal*:
 $\text{exp}[x \ \& \ x] \geq \text{exp}[x]$
 apply *auto*
 by (*smt (verit) evalDet intval-and.elims new-int.elims val-and-equal eval-unused-bits-zero*)

lemma *exp-and-nots*:

```

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

lemma exp-sign-extend:
assumes e = (1 << In) - 1
shows BinaryExpr BinAnd (UnaryExpr (UnarySignExtend In Out) x)
      (ConstantExpr (new-int b e))
      ≥ (UnaryExpr (UnaryZeroExtend In Out) x)

apply auto
subgoal premises p for m p va
proof -
  obtain va where va: [m,p] ⊢ x ↦ va
  using p(2) by auto
  then have va ≠ UndefVal
    by (simp add: evaltree-not-undef)
  then have 1: intval-and (intval-sign-extend In Out va) (IntVal b (take-bit b
e)) ≠ UndefVal
    using evalDet p(1) p(2) va by blast
  then have 2: intval-sign-extend In Out va ≠ UndefVal
    by auto
  then have 21: (0::nat) < b
    using eval-bits-1-64 p(4) by blast
  then have 3: b ⊆ (64::nat)
    using eval-bits-1-64 p(4) by blast
  then have 4: - ((2::int) ^ b div (2::int)) ⊆ sint (signed-take-bit (b - Suc
(0::nat)) (take-bit b e))
    by (simp add: 21 int-power-div-base signed-take-bit-int-greater-eq-minus-exp-word)
  then have 5: sint (signed-take-bit (b - Suc (0::nat)) (take-bit b e)) < (2::int)
^ b div (2::int)
    by (simp add: 21 3 Suc-le-lessD int-power-div-base signed-take-bit-int-less-exp-word)
  then have 6: [m,p] ⊢ UnaryExpr (UnaryZeroExtend In Out)
    x ↦ intval-and (intval-sign-extend In Out va) (IntVal b (take-bit b e))
    apply (cases va; simp)
  apply (simp add: ⟨(va::Value) ≠ UndefVal⟩) defer
  subgoal premises p for x3
  proof -
    have va = ObjRef x3
    using p(1) by auto
    then have sint (signed-take-bit (b - Suc (0::nat)) (take-bit b e)) <
(2::int) ^ b div (2::int)
      by (simp add: 5)
    then show ?thesis
      using 2 intval-sign-extend.simps(3) p(1) by blast
  qed

subgoal premises p for x4
proof -
  have sg1: va = ObjStr x4

```

```

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

```

```

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

```

```

lemma val-and-commute[simp]:
  val[x & y] = val[y & x]
  apply (cases x; cases y; auto)
  by (simp add: word-bw-comms(1))

```

Optimisations

```

optimization AndEqual: x & x ⟶ x
  using exp-and-equal by blast

```

```

optimization AndShiftConstantRight: ((const x) & y) ⟶ y & (const x)
  when ¬(is-ConstantExpr y)
  using size-flip-binary by auto

```

```

optimization AndNots: (~x) & (~y) ⟶ ~(x | y)
  apply (metis add-2-eq-Suc' less-SucI less-add-Suc1 not-less-eq size-binary-const
size-non-add)
  using exp-and-nots by presburger

```

optimization *AndSignExtend*: *BinaryExpr BinAnd (UnaryExpr (UnarySignExtend In Out) (x))*

$$\begin{aligned} & (\text{const } (\text{new-int } b \ e)) \\ \mapsto & (\text{UnaryExpr } (\text{UnaryZeroExtend In Out}) (x)) \\ & \text{when } (e = (1 \ll \text{In}) - 1) \end{aligned}$$

using *exp-sign-extend* **by** *simp*

optimization *AndNeutral*: $(x \ \& \sim(\text{const } (\text{IntVal } b \ 0))) \mapsto x$
when (*wf-stamp* *x* \wedge *stamp-expr* *x* = *IntegerStamp* *b lo hi*)

apply *auto*

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

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

optimization *AndRightFallThrough*: $(x \ \& \ y) \mapsto y$

when (((*and* (*not* (*IRExpr-down* *x*)) (*IRExpr-up* *y*)) = 0))

by (*simp add: IRExpr-down-def IRExpr-up-def*)

optimization *AndLeftFallThrough*: $(x \ \& \ y) \mapsto x$

when (((*and* (*not* (*IRExpr-down* *y*)) (*IRExpr-up* *x*)) = 0))

by (*simp add: IRExpr-down-def IRExpr-up-def*)

end

end

9.3 Experimental AndNode Phase

theory *NewAnd*

imports

Common

Graph.Long

begin

lemma *bin-distribute-and-over-or*:

$\text{bin}[z \ \& \ (x \mid y)] = \text{bin}[(z \ \& \ x) \mid (z \ \& \ y)]$

by (*smt* (*verit*, *best*) *bit-and-iff bit-eqI bit-or-iff*)

lemma *intval-distribute-and-over-or*:

$\text{val}[z \ \& \ (x \mid y)] = \text{val}[(z \ \& \ x) \mid (z \ \& \ y)]$

apply (*cases* *x*; *cases* *y*; *cases* *z*; *auto*)

using *bin-distribute-and-over-or* **by** *blast+*

lemma *exp-distribute-and-over-or*:

$\text{exp}[z \ \& \ (x \mid y)] \geq \text{exp}[(z \ \& \ x) \mid (z \ \& \ y)]$

apply *simp using intval-distribute-and-over-or*

using *BinaryExpr bin-eval.simps*(4,5)

using *intval-or.simps(1)* **unfolding** *new-int-bin.simps new-int.simps* **apply** *auto*
by (*metis bin-eval.simps(4) bin-eval.simps(5) intval-or.simps(2) intval-or.simps(5)*)

lemma *intval-and-commute*:
 $val[x \ \& \ y] = val[y \ \& \ x]$
by (*cases x; cases y; auto simp: and.commute*)

lemma *intval-or-commute*:
 $val[x \ | \ y] = val[y \ | \ x]$
by (*cases x; cases y; auto simp: or.commute*)

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

```

```

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

```

```

context stamp-mask
begin

```

```

lemma intval-up-and-zero-implies-zero:
  assumes and  $(\uparrow x) (\uparrow y) = 0$ 
  assumes  $[m, p] \vdash x \mapsto xv$ 
  assumes  $[m, p] \vdash y \mapsto yv$ 
  assumes  $\text{val}[xv \ \& \ yv] \neq \text{UndefVal}$ 
  shows  $\exists b. \text{val}[xv \ \& \ yv] = \text{new-int } b \ 0$ 
  using assms apply (cases xv; cases yv; auto)
  using up-mask-and-zero-implies-zero
  apply (smt (verit, best) take-bit-and take-bit-of-0)
  by presburger

```

```

lemma exp-eliminate-y:
  and  $(\uparrow y) (\uparrow z) = 0 \longrightarrow \text{BinaryExpr BinAnd } (\text{BinaryExpr BinOr } x \ y) \ z \geq \text{BinaryExpr BinAnd } x \ z$ 
  apply simp apply (rule impI; rule allI; rule allI; rule allI)
  subgoal premises p for m p v apply (rule impI) subgoal premises e
  proof –
    obtain xv where xv:  $[m, p] \vdash x \mapsto xv$ 
    using e by auto
    obtain yv where yv:  $[m, p] \vdash y \mapsto yv$ 
    using e by auto
    obtain zv where zv:  $[m, p] \vdash z \mapsto zv$ 
    using e by auto
    have lhs:  $v = \text{val}[(xv \mid yv) \ \& \ zv]$ 
    using xv yv zv
    by (smt (verit, best) BinaryExprE bin-eval.simps(4) bin-eval.simps(5) e evalDet)
    then have  $v = \text{val}[(xv \ \& \ zv) \mid (yv \ \& \ zv)]$ 
    by (simp add: intval-and-commute intval-distribute-and-over-or)
    also have  $\exists b. \text{val}[yv \ \& \ zv] = \text{new-int } b \ 0$ 
    using intval-up-and-zero-implies-zero

```



```

    by (metis calculation e intval-or.simps(5) p unfold-binary yv zv)
  ultimately have rhs:  $v = \text{val}[xv \ \& \ zv]$ 
  using intval-eliminate-y lhs by force
  from lhs rhs show ?thesis
  by (metis BinaryExpr BinaryExprE bin-eval.simps(4) e xv zv)
qed
done
done

```

lemma *leadingZeroBounds*:

```

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

```

lemma *above-nth-not-set*:

```

  fixes  $x :: \text{int64}$ 
  assumes  $n = 64 - \text{numberOfLeadingZeros } x$ 
  shows  $j > n \longrightarrow \neg(\text{bit } x \ j)$ 
  using assms unfolding numberOfLeadingZeros-def
  by (smt (verit, ccfv-SIG) highestOneBit-def int-nat-eq int-ops(6) less-imp-of-nat-less
    max-set-bit size64 zerosAboveHighestOne)

```

no-notation *LogicNegationNotation* (!-)

lemma *zero-horner*:

```

  horner-sum of-bool 2 (map ( $\lambda x. \text{False}$ ) xs) = 0
  apply (induction xs) apply simp
  by force

```

lemma *zero-map*:

```

  assumes  $j \leq n$ 
  assumes  $\forall i. j \leq i \longrightarrow \neg(f \ i)$ 
  shows  $\text{map } f \ [0..<n] = \text{map } f \ [0..<j] @ \text{map } (\lambda x. \text{False}) \ [j..<n]$ 
  apply (insert assms)
  by (smt (verit, del-insts) add-diff-inverse-nat atLeastLessThan-iff bot-nat-0.extremum
    leD map-append map-eq-conv set-upt upt-add-eq-append)

```

lemma *map-join-horner*:

```

  assumes  $\text{map } f \ [0..<n] = \text{map } f \ [0..<j] @ \text{map } (\lambda x. \text{False}) \ [j..<n]$ 
  shows  $\text{horner-sum of-bool } (2::'a::\text{len word}) \ (\text{map } f \ [0..<n]) = \text{horner-sum of-bool } 2 \ (\text{map } f \ [0..<j])$ 
  proof -
    have  $\text{horner-sum of-bool } (2::'a::\text{len word}) \ (\text{map } f \ [0..<n]) = \text{horner-sum of-bool } 2 \ (\text{map } f \ [0..<j]) + 2 \wedge \text{length } [0..<j] * \text{horner-sum of-bool } 2 \ (\text{map } f \ [j..<n])$ 
    using horner-sum-append
    by (smt (verit) assms diff-le-self diff-zero le-add-same-cancel2 length-append
      length-map length-upt map-append upt-add-eq-append)

```

```

also have ... = horner-sum of-bool 2 (map f [0..<j]) + 2 ^ length [0..<j] *
horner-sum of-bool 2 (map (λx. False) [j..<n])
  using assms
  by (metis calculation horner-sum-append length-map)
also have ... = horner-sum of-bool 2 (map f [0..<j])
  using zero-horner
  using mult-not-zero by auto
finally show ?thesis by simp
qed

```

```

lemma split-horner:
  assumes  $j \leq n$ 
  assumes  $\forall i. j \leq i \longrightarrow \neg(f\ i)$ 
  shows horner-sum of-bool (2::'a::len word) (map f [0..<n]) = horner-sum of-bool
2 (map f [0..<j])
  apply (rule map-join-horner)
  apply (rule zero-map)
  using assms by auto

```

```

lemma transfer-map:
  assumes  $\forall i. i < n \longrightarrow f\ i = f'\ i$ 
  shows (map f [0..<n]) = (map f' [0..<n])
  using assms by simp

```

```

lemma transfer-horner:
  assumes  $\forall i. i < n \longrightarrow f\ i = f'\ i$ 
  shows horner-sum of-bool (2::'a::len word) (map f [0..<n]) = horner-sum of-bool
2 (map f' [0..<n])
  using assms using transfer-map
  by (smt (verit, best))

```

```

lemma L1:
  assumes  $n = 64 - \text{numberOfLeadingZeros } (\uparrow z)$ 
  assumes  $[m, p] \vdash z \mapsto \text{IntVal } b\ zv$ 
  shows  $\text{and } v\ zv = \text{and } (v \bmod 2^n)\ zv$ 
proof –
  have  $n \leq 64$ 
  using assms
  using diff-le-self by blast
  also have  $\text{and } v\ zv = \text{horner-sum of-bool } 2\ (\text{map } (\text{bit } (\text{and } v\ zv))\ [0..<64])$ 
  using horner-sum-bit-eq-take-bit size64
  by (metis size-word.rep-eq take-bit-length-eq)
  also have ... = horner-sum of-bool 2 (map (λi. bit (and v zv) i) [0..<64])
  by blast
  also have ... = horner-sum of-bool 2 (map (λi. ((bit v i) ∧ (bit zv i))) [0..<64])
  using bit-and-iff by metis
  also have ... = horner-sum of-bool 2 (map (λi. ((bit v i) ∧ (bit zv i))) [0..<n])
  proof –
  have  $\forall i. i \geq n \longrightarrow \neg(\text{bit } zv\ i)$ 

```

```

    using above-nth-not-set assms(1)
    using assms(2) not-may-implies-false
    by (smt (verit, ccfv-SIG) One-nat-def diff-less int-ops(6) leadingZerosAddHighestOne linorder-not-le nat-int-comparison(2) not-numeral-le-zero size64 zero-less-Suc zerosAboveHighestOne)
  then have  $\forall i. i \geq n \longrightarrow \neg((\text{bit } v \ i) \wedge (\text{bit } zv \ i))$ 
    by auto
  then show ?thesis using nle split-horner
    by (metis (no-types, lifting))
qed
also have ... = horner-sum of-bool 2 (map ( $\lambda i. ((\text{bit } (v \bmod 2^n) \ i) \wedge (\text{bit } zv \ i))$ ) [0..\forall i. i < n \longrightarrow \text{bit } (v \bmod 2^n) \ i = \text{bit } v \ i
    by (metis bit-take-bit-iff take-bit-eq-mod)
  then have  $\forall i. i < n \longrightarrow ((\text{bit } v \ i) \wedge (\text{bit } zv \ i)) = ((\text{bit } (v \bmod 2^n) \ i) \wedge (\text{bit } zv \ i))$ 
    by force
  then show ?thesis
    by (rule transfer-horner)
qed
also have ... = horner-sum of-bool 2 (map ( $\lambda i. ((\text{bit } (v \bmod 2^n) \ i) \wedge (\text{bit } zv \ i))$ ) [0..<64])
proof -
  have  $\forall i. i \geq n \longrightarrow \neg(\text{bit } zv \ i)$ 
    using above-nth-not-set assms(1)
    using assms(2) not-may-implies-false
    by (smt (verit, ccfv-SIG) One-nat-def diff-less int-ops(6) leadingZerosAddHighestOne linorder-not-le nat-int-comparison(2) not-numeral-le-zero size64 zero-less-Suc zerosAboveHighestOne)
  then show ?thesis
    by (metis (no-types, lifting) assms(1) diff-le-self split-horner)
qed
also have ... = horner-sum of-bool 2 (map (bit (and (v mod 2^n) zv)) [0..<64])
  by (meson bit-and-iff)
also have ... = and (v mod 2^n) zv
  using horner-sum-bit-eq-take-bit size64
  by (metis size-word.rep-eq take-bit-length-eq)
finally show ?thesis
  using ⟨and (v::64 word) (zv::64 word) = horner-sum of-bool (2::64 word) (map (bit (and v zv)) [0::nat..<64::nat])⟩
  ⟨horner-sum of-bool (2::64 word) (map ( $\lambda i::\text{nat}. \text{bit } ((v::64 \text{ word}) \bmod (2::64 \text{ word}) \wedge (n::\text{nat})) \ i \wedge \text{bit } (zv::64 \text{ word}) \ i$ ) [0::nat..<64::nat]) = horner-sum of-bool (2::64 word) (map (bit (and (v mod (2::64 word) ^ n) zv)) [0::nat..<64::nat])⟩
  ⟨horner-sum of-bool (2::64 word) (map ( $\lambda i::\text{nat}. \text{bit } ((v::64 \text{ word}) \bmod (2::64 \text{ word}) \wedge (n::\text{nat})) \ i \wedge \text{bit } (zv::64 \text{ word}) \ i$ ) [0::nat..\lambda i::\text{nat}. \text{bit } (v \bmod (2::64 \text{ word}) \wedge n) \ i \wedge \text{bit } zv \ i) [0::nat..<64::nat])⟩
  ⟨horner-sum of-bool (2::64 word) (map ( $\lambda i::\text{nat}. \text{bit } (v::64 \text{ word}) \ i \wedge \text{bit } (zv::64 \text{ word}) \ i$ ) [0::nat..<64::nat]) = horner-sum of-bool (2::64 word) (map ( $\lambda i::\text{nat}. \text{bit } v \ i \wedge \text{bit } zv \ i$ ) [0::nat..

```

$\langle \text{horner-sum of-bool } (2::64 \text{ word}) (\text{map } (\lambda i::\text{nat}. \text{bit } (v::64 \text{ word}) i \wedge \text{bit } (zv::64 \text{ word}) i) [0::\text{nat}..<n::\text{nat}]) = \text{horner-sum of-bool } (2::64 \text{ word}) (\text{map } (\lambda i::\text{nat}. \text{bit } (v \bmod (2::64 \text{ word}) \wedge n) i \wedge \text{bit } zv i) [0::\text{nat}..<n]) \rangle$
 $\langle \text{horner-sum of-bool } (2::64 \text{ word}) (\text{map } (\text{bit } (\text{and } ((v::64 \text{ word}) \bmod (2::64 \text{ word}) \wedge (n::\text{nat})) (zv::64 \text{ word}))) [0::\text{nat}..<64::\text{nat}]) = \text{and } (v \bmod (2::64 \text{ word}) \wedge n) zv \rangle$
 $\langle \text{horner-sum of-bool } (2::64 \text{ word}) (\text{map } (\text{bit } (\text{and } (v::64 \text{ word}) (zv::64 \text{ word}))) [0::\text{nat}..<64::\text{nat}]) = \text{horner-sum of-bool } (2::64 \text{ word}) (\text{map } (\lambda i::\text{nat}. \text{bit } v i \wedge \text{bit } zv i) [0::\text{nat}..<64::\text{nat}]) \rangle$
by *presburger*
qed

lemma *up-mask-upper-bound*:

assumes $[m, p] \vdash x \mapsto \text{IntVal } b \text{ } xv$
shows $xv \leq (\uparrow x)$
using *assms*
by (*metis* (*no-types*, *lifting*) *and.idem and.right-neutral bit.conj-cancel-left bit.conj-disj-distrib(1)* *bit.double-compl ucast-id up-spec word-and-le1 word-not-dist(2)*)

lemma *L2*:

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

thm-oracles *L1 L2*

```

lemma unfold-binary-width-add:
  shows ( $[m,p] \vdash \text{BinaryExpr BinAdd } xe \ ye \mapsto \text{IntVal } b \ \text{val}$ ) = ( $\exists \ x \ y.$ 
    ( $[m,p] \vdash xe \mapsto \text{IntVal } b \ x$ )  $\wedge$ 
    ( $[m,p] \vdash ye \mapsto \text{IntVal } b \ y$ )  $\wedge$ 
    ( $\text{IntVal } b \ \text{val} = \text{bin-eval BinAdd } (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)$ )  $\wedge$ 
    ( $\text{IntVal } b \ \text{val} \neq \text{UndefVal}$ )
  ) (is ?L = ?R)
proof (intro iffI)
  assume  $\mathcal{I}$ : ?L
  show ?R apply (rule evaltree.cases[OF  $\mathcal{I}$ ])
  apply force+ apply auto[1]
  apply (smt (verit) intval-add.elims intval-bits.simps)
  by blast
next
  assume  $R$ : ?R
  then obtain  $x \ y$  where  $[m,p] \vdash xe \mapsto \text{IntVal } b \ x$ 
    and  $[m,p] \vdash ye \mapsto \text{IntVal } b \ y$ 
    and  $\text{new-int } b \ \text{val} = \text{bin-eval BinAdd } (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)$ 
    and  $\text{new-int } b \ \text{val} \neq \text{UndefVal}$ 
  by auto
  then show ?L
  using  $R$  by blast
qed

lemma unfold-binary-width-and:
  shows ( $[m,p] \vdash \text{BinaryExpr BinAnd } xe \ ye \mapsto \text{IntVal } b \ \text{val}$ ) = ( $\exists \ x \ y.$ 
    ( $[m,p] \vdash xe \mapsto \text{IntVal } b \ x$ )  $\wedge$ 
    ( $[m,p] \vdash ye \mapsto \text{IntVal } b \ y$ )  $\wedge$ 
    ( $\text{IntVal } b \ \text{val} = \text{bin-eval BinAnd } (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)$ )  $\wedge$ 
    ( $\text{IntVal } b \ \text{val} \neq \text{UndefVal}$ )
  ) (is ?L = ?R)
proof (intro iffI)
  assume  $\mathcal{I}$ : ?L
  show ?R apply (rule evaltree.cases[OF  $\mathcal{I}$ ])
  apply force+ apply auto[1] using intval-and.elims intval-bits.simps
  apply (smt (verit) new-int.simps new-int-bin.simps take-bit-and)
  by blast
next
  assume  $R$ : ?R
  then obtain  $x \ y$  where  $[m,p] \vdash xe \mapsto \text{IntVal } b \ x$ 
    and  $[m,p] \vdash ye \mapsto \text{IntVal } b \ y$ 
    and  $\text{new-int } b \ \text{val} = \text{bin-eval BinAnd } (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)$ 
    and  $\text{new-int } b \ \text{val} \neq \text{UndefVal}$ 
  by auto
  then show ?L
  using  $R$  by blast
qed

```

lemma *mod-dist-over-add-right*:

fixes *a b c* :: *int64*

fixes *n* :: *nat*

assumes *1*: $0 < n$

assumes *2*: $n < 64$

shows $(a + b \bmod 2^n) \bmod 2^n = (a + b) \bmod 2^n$

using *mod-dist-over-add*

by (*simp add: 1 2 add.commute*)

lemma *numberOfLeadingZeros-range*:

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

unfolding *numberOfLeadingZeros-def highestOneBit-def* **using** *max-set-bit*

by (*simp add: highestOneBit-def leadingZeroBounds numberOfLeadingZeros-def*)

lemma *improved-opt*:

assumes $\text{numberOfLeadingZeros } (\uparrow z) + \text{numberOfTrailingZeros } (\uparrow y) \geq 64$

shows $\text{exp}[(x + y) \& z] \geq \text{exp}[x \& z]$

apply *simp* **apply** (*rule allI*)+; *rule impI*)

subgoal **premises** *eval* **for** *m p v*

proof –

obtain *n* **where** *n*: $n = 64 - \text{numberOfLeadingZeros } (\uparrow z)$

by *simp*

obtain *b val* **where** *val*: $[m, p] \vdash \text{exp}[(x + y) \& z] \mapsto \text{IntVal } b \text{ val}$

by (*metis BinaryExprE bin-eval-new-int eval new-int.simps*)

then obtain *xv yv* **where** *addv*: $[m, p] \vdash \text{exp}[x + y] \mapsto \text{IntVal } b (xv + yv)$

apply (*subst (asm) unfold-binary-width-and*) **by** (*metis add.right-neutral*)

then obtain *yv* **where** *yv*: $[m, p] \vdash y \mapsto \text{IntVal } b \text{ yv}$

apply (*subst (asm) unfold-binary-width-add*) **by** *blast*

from *addv* **obtain** *xv* **where** *xv*: $[m, p] \vdash x \mapsto \text{IntVal } b \text{ xv}$

apply (*subst (asm) unfold-binary-width-add*) **by** *blast*

from *val* **obtain** *zv* **where** *zv*: $[m, p] \vdash z \mapsto \text{IntVal } b \text{ zv}$

apply (*subst (asm) unfold-binary-width-and*) **by** *blast*

have *addv*: $[m, p] \vdash \text{exp}[x + y] \mapsto \text{new-int } b (xv + yv)$

apply (*rule evaltree.BinaryExpr*)

using *xv* **apply** *simp*

using *yv* **apply** *simp*

by *simp+*

have *lhs*: $[m, p] \vdash \text{exp}[(x + y) \& z] \mapsto \text{new-int } b (\text{and } (xv + yv) \text{ zv})$

apply (*rule evaltree.BinaryExpr*)

using *addv* **apply** *simp*

using *zv* **apply** *simp*

using *addv* **apply** *auto*[1]

by *simp*

have *rhs*: $[m, p] \vdash \text{exp}[x \& z] \mapsto \text{new-int } b (\text{and } xv \text{ zv})$

apply (*rule evaltree.BinaryExpr*)

using *xv* **apply** *simp*

using *zv* **apply** *simp*

apply *force*

by *simp*

```

then show ?thesis
proof (cases numberOfLeadingZeros ( $\uparrow z$ ) > 0)
  case True
    have n-bounds:  $0 \leq n \wedge n < 64$ 
      using diff-le-self n numberOfLeadingZeros-range
      by (simp add: True)
    have and (xv + yv) zv = and ((xv + yv) mod  $2^n$ ) zv
      using L1 n zv by blast
    also have ... = and ((xv + (yv mod  $2^n$ )) mod  $2^n$ ) zv
      using mod-dist-over-add-right n-bounds
      by (metis take-bit-0 take-bit-eq-mod zero-less-iff-neq-zero)
    also have ... = and (((xv mod  $2^n$ ) + (yv mod  $2^n$ )) mod  $2^n$ ) zv
      by (metis bits-mod-by-1 mod-dist-over-add n-bounds order-le-imp-less-or-eq
power-0)
    also have ... = and ((xv mod  $2^n$ ) mod  $2^n$ ) zv
      using L2 n zv yv
      using assms by auto
    also have ... = and (xv mod  $2^n$ ) zv
      using mod-mod-trivial
    by (smt (verit, best) and.idem take-bit-eq-mask take-bit-eq-mod word-bw-assocs(1))
    also have ... = and xv zv
      using L1 n zv by metis
    finally show ?thesis
      using eval lhs rhs
      by (metis evalDet)
  next
    case False
    then have numberOfLeadingZeros ( $\uparrow z$ ) = 0
      by simp
    then have numberOfTrailingZeros ( $\uparrow y$ )  $\geq 64$ 
      using assms(1)
      by fastforce
    then have yv = 0
      using yv
      by (metis (no-types, lifting) L1 L2 add-diff-cancel-left' and.comm-neutral
and.idem bit.compl-zero bit.conj-cancel-right bit.conj-disj-distrib(1) bit.double-compl
less-imp-diff-less linorder-not-le word-not-dist(2))
    then show ?thesis
      by (metis add.right-neutral eval evalDet lhs rhs)
  qed
qed
done

thm-oracles improved-opt

```

```

phase NewAnd
  terminating size
begin

```

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

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

end

end

```
theory ConditionalPhase
imports
  Common
  Proofs.StampEvalThms
begin
```

192

lemma *negates*: $\exists v b. e = \text{IntVal } b \ v \wedge b > 0 \implies \text{val-to-bool } (\text{val}[e]) \longleftrightarrow \neg(\text{val-to-bool } (\text{val}[!e]))$
unfolding *intval-logic-negation.simps*
by (*metis* (*mono-tags*, *lifting*) *intval-logic-negation.simps*(1) *logic-negate-def new-int.simps* *of-bool-eq*(2) *one-neq-zero* *take-bit-of-0* *take-bit-of-1* *val-to-bool.simps*(1))

lemma *negation-condition-intval*:
assumes $e = \text{IntVal } b \ ie$
assumes $0 < b$
shows $\text{val}[(!e) \ ? \ x : y] = \text{val}[e \ ? \ y : x]$
using *assms* **by** (*cases* *e*; *auto* *simp*: *negates* *logic-negate-def*)

lemma *negation-preserve-eval*:
assumes $[m, p] \vdash \text{exp}[!e] \mapsto v$
shows $\exists v'. ([m, p] \vdash \text{exp}[e] \mapsto v') \wedge v = \text{val}[!v']$
using *assms* **by** *auto*

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

optimization *NegateConditionFlipBranches*: $((!e) \ ? \ x : y) \mapsto (e \ ? \ y : x)$
apply *simp* **using** *negation-condition-intval* *negation-preserve-eval-intval*
by (*smt* (*verit*, *best*) *ConditionalExpr* *ConditionalExprE* *Value.distinct*(1) *evalDet* *negates* *negation-preserve-eval*)

optimization *DefaultTrueBranch*: $(\text{true} \ ? \ x : y) \mapsto x$.

optimization *DefaultFalseBranch*: $(\text{false} \ ? \ x : y) \mapsto y$.

optimization *ConditionalEqualBranches*: $(e \ ? \ x : x) \mapsto x$.

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

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

lemma *val-optimise-integer-test*:
assumes $\exists v. x = \text{IntVal } 32 \ v$
shows $\text{val}[(x \ \& \ (\text{IntVal } 32 \ 1)) \ \text{eq} \ (\text{IntVal } 32 \ 0)) \ ? \ (\text{IntVal } 32 \ 0) : (\text{IntVal } 32 \ 1)] =$

```

      val[x & IntVal 32 1]
    using assms apply auto
    apply (metis (full-types) bool-to-val.simps(2) val-to-bool.simps(1))
    by (metis (mono-tags, lifting) and-one-eq bool-to-val.simps(1) even-iff-mod-2-eq-zero
    odd-iff-mod-2-eq-one val-to-bool.simps(1))

```

```

optimization ConditionalEliminateKnownLess:  $((x < y) \text{ ? } x : y) \mapsto x$ 
      when (stamp-under (stamp-expr x) (stamp-expr y)
       $\wedge$  wf-stamp x  $\wedge$  wf-stamp y)
    using stamp-under-defn by auto

```

```

optimization ConditionalEqualIsRHS:  $((x \text{ eq } y) \text{ ? } x : y) \mapsto y$ 
    apply auto
    by (smt (verit) Value.inject(1) bool-to-val.simps(2) bool-to-val-bin.simps evalDet

    intval-equals.elims val-to-bool.elims(1))

```

```

optimization normalizeX:  $((x \text{ eq } \text{const } (\text{IntVal } 32 \ 0)) \text{ ? }
      (\text{const } (\text{IntVal } 32 \ 0)) : (\text{const } (\text{IntVal } 32 \ 1))) \mapsto x$ 
      when (IRExpr-up x = 1)  $\wedge$  stamp-expr x = IntegerStamp

```

```

b 0 1
    apply auto
    subgoal premises p for m p v xa
    proof –
      obtain xa where xa: [m,p]  $\vdash x \mapsto xa$ 
      using p by blast
      have 3: [m,p]  $\vdash$  if val-to-bool (intval-equals xa (IntVal (32::nat) (0::64 word)))
      then ConstantExpr (IntVal (32::nat) (0::64 word))
      else ConstantExpr (IntVal (32::nat) (1::64 word))  $\mapsto v$ 
      using evalDet p(3) p(5) xa
      using p(4) p(6) by blast
      then have 4: xa = IntVal 32 0 | xa = IntVal 32 1
      sorry
      then have 6: v = xa
      sorry
      then show ?thesis
      using xa by auto
    qed
  done

```

```

optimization normalizeX2:  $((x \text{ eq } (\text{const } (\text{IntVal } 32 \ 1))) \text{ ? }
      (\text{const } (\text{IntVal } 32 \ 1)) : (\text{const } (\text{IntVal } 32 \ 0))) \mapsto x$ 
      when (x = ConstantExpr (IntVal 32 0) | (x =
      ConstantExpr (IntVal 32 1))) .

```

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

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

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

optimization *OptimiseIntegerTest*:
 $((x \ \& \ (\text{const } (\text{IntVal } 32 \ 1))) \text{ eq } (\text{const } (\text{IntVal } 32 \ 0))) \ ?$
 $(\text{const } (\text{IntVal } 32 \ 0)) : (\text{const } (\text{IntVal } 32 \ 1))) \mapsto$
 $x \ \& \ (\text{const } (\text{IntVal } 32 \ 1))$
 $\text{when } (\text{stamp-expr } x = \text{default-stamp} \wedge \text{wf-stamp } x)$
apply *simp* **apply** $(\text{rule } \text{impI}; (\text{rule } \text{allI})+; \text{rule } \text{impI})$
subgoal **premises** *eval* **for** $m \ p \ v$
proof –
obtain xv **where** $[m, p] \vdash x \mapsto xv$
using *eval* **by** *fast*
then **have** $x32: \exists v. xv = \text{IntVal } 32 \ v$
using *stamp-of-default* *eval* **by** *auto*
obtain lhs **where** $[m, p] \vdash \text{exp}[(((x \ \& \ (\text{const } (\text{IntVal } 32 \ 1))) \text{ eq } (\text{const } (\text{IntVal } 32 \ 0))) \ ?$
 $(\text{const } (\text{IntVal } 32 \ 0)) : (\text{const } (\text{IntVal } 32 \ 1))) \mapsto lhs$
using *eval*(2) **by** *auto*
then **have** $lhsV: lhs = \text{val}[(xv \ \& \ (\text{IntVal } 32 \ 1)) \text{ eq } (\text{IntVal } 32 \ 0)) \ ? (\text{IntVal } 32$
 $0) : (\text{IntVal } 32 \ 1)]$
using $xv \ \text{evaltree.BinaryExpr evaltree.ConstantExpr evaltree.ConditionalExpr}$
by $(\text{smt } (\text{verit}) \ \text{ConditionalExprE ConstantExprE bin-eval.simps}(11) \ \text{bin-eval.simps}(4)$
 $\text{evalDet intval-conditional.simps } \text{unfold-binary})$
obtain rhs **where** $[m, p] \vdash \text{exp}[x \ \& \ (\text{const } (\text{IntVal } 32 \ 1))] \mapsto rhs$
using *eval*(2) **by** *blast*
then **have** $rhsV: rhs = \text{val}[xv \ \& \ \text{IntVal } 32 \ 1]$
by $(\text{metis } \text{BinaryExprE ConstantExprE bin-eval.simps}(4) \ \text{evalDet } xv)$
have $lhs = rhs$ **using** *val-optimize-integer-test* $x32$
using $lhsV \ rhsV$ **by** *presburger*
then **show** *?thesis*
by $(\text{metis } \text{eval}(2) \ \text{evalDet } lhs \ rhs)$

```
qed
done
```

```
optimization opt-optimise-integer-test-2:
  (((x & (const (IntVal 32 1))) eq (const (IntVal 32 0))) ?
    (const (IntVal 32 0)) : (const (IntVal 32 1)))  $\mapsto$ 
     $\begin{matrix} x \\ \text{when } (x = \text{ConstantExpr } (\text{IntVal } 32\ 0) \mid (x = \text{ConstantExpr } (\text{IntVal } \\ 32\ 1))) \end{matrix}$  .
```

```
end
```

```
end
```

9.5 MulNode Phase

```
theory MulPhase
```

```
  imports
```

```
    Common
```

```
    Proofs.StampEvalThms
```

```
begin
```

```
fun mul-size :: IRExpr  $\Rightarrow$  nat where
```

```
  mul-size (UnaryExpr op e) = (mul-size e) + 2 |
```

```
  mul-size (BinaryExpr BinMul x y) = ((mul-size x) + (mul-size y) + 2) * 2 |
```

```
  mul-size (BinaryExpr op x y) = (mul-size x) + (mul-size y) + 2 |
```

```
  mul-size (ConditionalExpr cond t f) = (mul-size cond) + (mul-size t) + (mul-size f) + 2 |
```

```
  mul-size (ConstantExpr c) = 1 |
```

```
  mul-size (ParameterExpr ind s) = 2 |
```

```
  mul-size (LeafExpr nid s) = 2 |
```

```
  mul-size (ConstantVar c) = 2 |
```

```
  mul-size (VariableExpr x s) = 2
```

```
phase MulNode
```

```
  terminating mul-size
```

```
begin
```

```
lemma bin-eliminate-redundant-negative:
```

```
  uminus (x :: 'a::len word) * uminus (y :: 'a::len word) = x * y
```

```
  by simp
```

lemma *bin-multiply-identity*:

($x :: 'a::len\ word$) * 1 = x
by *simp*

lemma *bin-multiply-eliminate*:

($x :: 'a::len\ word$) * 0 = 0
by *simp*

lemma *bin-multiply-negative*:

($x :: 'a::len\ word$) * *uminus* 1 = *uminus* x
by *simp*

lemma *bin-multiply-power-2*:

($x :: 'a::len\ word$) * (2^j) = $x << j$
by *simp*

lemma *take-bit64* [*simp*]:

fixes $w :: int64$
shows *take-bit* 64 $w = w$

proof –

have *Nat.size* $w = 64$
by (*simp add: size64*)

then show ?thesis

by (*metis lt2p-lem mask-eq-iff take-bit-eq-mask verit-comp-simplify1* (2) *wsst-TYs* (3))

qed

lemma *mergeTakeBit*:

fixes $a :: nat$

fixes $b\ c :: 64\ word$

shows *take-bit* a (*take-bit* a (b) * *take-bit* a (c)) =
take-bit a ($b * c$)

by (*smt* (*verit*, *ccfv-SIG*) *take-bit-mult take-bit-of-int unsigned-take-bit-eq word-mult-def*)

lemma *val-eliminate-redundant-negative*:

assumes *val* [$-x * -y$] \neq *UndefVal*

shows *val* [$-x * -y$] = *val* [$x * y$]

using *assms apply* (*cases* x ; *cases* y ; *auto*)

using *mergeTakeBit* **by** *auto*

lemma *val-multiply-neutral*:

assumes $x = new_int\ b\ v$

shows *val* [$x * (IntVal\ b\ 1)$] = *val* [x]

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

```

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

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

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

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

```

```

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  $\text{val}[x * y] = \text{val}[x \ll \text{IntVal } 64 \ i]$ 
  using assms apply (cases  $x$ ; cases  $y$ ; auto)
  subgoal premises  $p$  for  $x2$ 
  proof –
    have  $63 :: \text{int64} = \text{mask } 6$ 
    by eval
    then have  $(2 :: \text{int}) \wedge 6 = 64$ 
    by eval
    then have  $\text{uint } i < (2 :: \text{int}) \wedge 6$ 
    by (metis linorder-not-less lt2p-lem of-int-numeral  $p(4)$  size64 word-2p-lem
word-of-int-2p
      wsst-TYs(3))
    then have and  $i (\text{mask } 6) = i$ 
    using mask-eq-iff by blast
    then show  $x2 \ll \text{unat } i = x2 \ll \text{unat } (\text{and } i \ (63 :: 64 \text{ word}))$ 
    unfolding 63
    by force
  qed
by presburger

```

```

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     $\text{val}[x * y] = \text{val}[(x \ll \text{IntVal } 64 \ i) + x]$ 
using    assms apply (cases x; cases y; auto)
  subgoal premises p for x2
proof -
  have 63:  $(63 :: \text{int64}) = \text{mask } 6$ 
  by eval
  then have  $(2 :: \text{int}) \wedge 6 = 64$ 
  by eval
  then have  $\text{and } i (\text{mask } 6) = i$ 
  using mask-eq-iff by (simp add: less-mask-eq p(6))
  then have  $x2 * ((2 :: 64 \text{ word}) \wedge \text{unat } i + (1 :: 64 \text{ word})) = (x2 * ((2 :: 64 \text{ word})$ 
 $\wedge \text{unat } i)) + x2$ 
  by (simp add: distrib-left)
  then show  $x2 * ((2 :: 64 \text{ word}) \wedge \text{unat } i + (1 :: 64 \text{ word})) = x2 \ll \text{unat } (\text{and } i$ 
 $(63 :: 64 \text{ word})) + x2$ 
  by (simp add: 63 ‹and (i::64 word) (mask (6::nat)) = i›)
qed
using val-to-bool.simps(2) by presburger

```

```

lemma val-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     $\text{val}[x * y] = \text{val}[(x \ll \text{IntVal } 64 \ i) - x]$ 
  using    assms apply (cases x; cases y; auto)
  subgoal premises p for x2
  proof -
    have 63:  $(63 :: \text{int64}) = \text{mask } 6$ 
    by eval
    then have  $(2 :: \text{int}) \wedge 6 = 64$ 
    by eval
    then have  $\text{and } i (\text{mask } 6) = i$ 
    using mask-eq-iff by (simp add: less-mask-eq p(6))
    then have  $x2 * ((2 :: 64 \text{ word}) \wedge \text{unat } i - (1 :: 64 \text{ word})) = (x2 * ((2 :: 64 \text{ word})$ 
 $\wedge \text{unat } i)) - x2$ 
    by (simp add: right-diff-distrib')
    then show  $x2 * ((2 :: 64 \text{ word}) \wedge \text{unat } i - (1 :: 64 \text{ word})) = x2 \ll \text{unat } (\text{and } i$ 
 $(63 :: 64 \text{ word})) - x2$ 
    by (simp add: 63 ‹and (i::64 word) (mask (6::nat)) = i›)
  qed
  using val-to-bool.simps(2) by presburger

```

lemma *val-distribute-multiplication*:

assumes $x = \text{new-int } 64 \text{ } xx \wedge q = \text{new-int } 64 \text{ } qq \wedge a = \text{new-int } 64 \text{ } aa$
shows $\text{val}[x * (q + a)] = \text{val}[(x * q) + (x * a)]$
apply (*cases* x ; *cases* q ; *cases* a ; *auto*) **using** *distrib-left* *assms* **by** *auto*

lemma *val-MulPower2AddPower2*:

fixes $i \ j :: 64 \text{ word}$
assumes $y = \text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) + (2 \wedge \text{unat}(j)))$
and $0 < i$
and $0 < j$
and $i < 64$
and $j < 64$
and $x = \text{new-int } 64 \text{ } xx$
shows $\text{val}[x * y] = \text{val}[(x << \text{IntVal } 64 \ i) + (x << \text{IntVal } 64 \ j)]$
using *assms*
proof –
have $63: (63 :: \text{int}64) = \text{mask } 6$
by *eval*
then have $(2 :: \text{int}) \wedge 6 = 64$
by *eval*
then have $n: \text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) + (2 \wedge \text{unat}(j))) =$
 $\text{val}[(\text{IntVal } 64 \ (2 \wedge \text{unat}(i))) + (\text{IntVal } 64 \ (2 \wedge \text{unat}(j)))]$

using *assms* **by** (*cases* i ; *cases* j ; *auto*)
then have $1: \text{val}[x * ((\text{IntVal } 64 \ (2 \wedge \text{unat}(i))) + (\text{IntVal } 64 \ (2 \wedge \text{unat}(j))))]$
 $=$
 $\text{val}[(x * \text{IntVal } 64 \ (2 \wedge \text{unat}(i))) + (x * \text{IntVal } 64 \ (2 \wedge \text{unat}(j)))]$

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

thm-oracles *val-MulPower2AddPower2*

lemma *exp-multiply-zero-64*:

$\text{exp}[x * (\text{const } (\text{IntVal } 64 \ 0))] \geq \text{ConstantExpr } (\text{IntVal } 64 \ 0)$
using *val-multiply-zero* **apply** *auto*
by (*smt* (*verit*) *Value.inject*(1) *constantAsStamp.simps*(1) *int-signed-value-bounds*)

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

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

thm-oracles *exp-multiply-neutral*

lemma *exp-MulPower2*:
fixes $i :: 64 \text{ word}$
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 $\text{exp}[x * y] \geq \text{exp}[x << \text{ConstantExpr } (\text{IntVal } 64 \ i)]$
using *assms* **apply** *simp*
by (*metis ConstantExprE equiv-exprs-def unfold-binary*)

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 $\text{exp}[x * y] \geq \text{exp}[(x << \text{ConstantExpr } (\text{IntVal } 64 \ i)) + x]$
using *assms* **apply** *simp*
by (*metis (no-types, lifting) ConstantExprE equiv-exprs-def unfold-binary*)

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 $\text{exp}[x > (\text{const } \text{IntVal } b \ 0)]$
and $\text{exp}[y > (\text{const } \text{IntVal } b \ 0)]$
shows $\text{exp}[x * y] \geq \text{exp}[(x << \text{ConstantExpr } (\text{IntVal } 64 \ i)) - x]$
using *assms* **apply** *simp*
by (*metis (no-types, lifting) ConstantExprE equiv-exprs-def unfold-binary*)

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  $\text{exp}[x * y] \geq \text{exp}[(x << \text{ConstantExpr } (\text{IntVal } 64 \ i)) + (x << \text{ConstantExpr } (\text{IntVal } 64 \ j))]$ 
using assms apply simp
by (metis (no-types, lifting) ConstantExprE equiv-exprs-def unfold-binary)

```

```

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  $\text{exp}[y > x]$ 
  apply auto
  sorry

```

```

lemma exp-distribute-multiplication:
  shows  $\text{exp}[(x * q) + (x * a)] \geq \text{exp}[x * (q + a)]$ 
  sorry

```

Optimisations

```

optimization EliminateRedundantNegative:  $-x * -y \mapsto x * y$ 
  using mul-size.simps apply auto
  by (metis BinaryExpr val-eliminate-redundant-negative bin-eval.simps(2))

```

```

optimization MulNeutral:  $x * \text{ConstantExpr } (\text{IntVal } b \ 1) \mapsto x$ 
  using exp-multiply-neutral by blast

```

```

optimization MulEliminator:  $x * \text{ConstantExpr } (\text{IntVal } b \ 0) \mapsto \text{const } (\text{IntVal } b \ 0)$ 
  apply auto
  by (smt (verit) Value.inject(1) constantAsStamp.simps(1) int-signed-value-bounds intval-mul.elims
    mult-zero-right new-int.simps new-int-bin.simps take-bit-of-0 unfold-const
    valid-stamp.simps(1) valid-value.simps(1) val-multiply-zero)

```

```

optimization MulNegate:  $x * -(\text{const } (\text{IntVal } b \ 1)) \mapsto -x$ 
  apply auto
  by (smt (verit) Value.distinct(1) Value.sel(1) add.inverse-inverse intval-mul.elims
    intval-negate.simps(1) mask-eq-take-bit-minus-one new-int.simps new-int-bin.simps)

```

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

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 ($\exists vv\ b. (v = \text{IntVal } b\ vv \wedge \text{val-to-bool val}[(\text{IntVal } b\ 0) < v])$)
apply (*rule impI*) **subgoal** **premises** *eval*
proof –
obtain *b lo hi* **where** *xstamp*: *stamp-expr* *x* = *IntegerStamp* *b lo hi*
by (*meson isNonZero.elims(2) assms*)
then obtain *vv* **where** *vdef*: *v* = *IntVal* *b vv*
by (*metis assms(2) eval valid-int wf-stamp-def*)
have *lo* > 0
using *assms(1) xstamp* **by** *force*
then have *signed-above*: *int-signed-value* *b vv* > 0
using *assms unfolding wf-stamp-def*
using *eval vdef xstamp* **by** *fastforce*
have *take-bit* *b vv* = *vv*
using *eval eval-unused-bits-zero vdef* **by** *auto*
then have *vv* > 0
by (*metis bit-take-bit-iff int-signed-value.simps not-less-zero signed-eq-0-iff
signed-take-bit-eq-if-positive take-bit-0 take-bit-of-0 verit-comp-simplify1(1)
word-gt-0
signed-above*)
then show *?thesis*
using *vdef signed-above*
by *simp*
qed
done

optimization *MulPower2*: $x * y \longmapsto x << \text{const } (\text{IntVal } 64\ i)$
when (*i* > 0 \wedge
64 > *i* \wedge
y = *exp*[*const* (*IntVal* 64 (2 \wedge *unat*(*i*)))]])

defer
apply *simp* **apply** (*rule impI*; (*rule allI*) $+$; *rule impI*)
subgoal **premises** *eval* **for** *m p v*
proof –
obtain *xv* **where** *xv*: $[m, p] \vdash x \mapsto xv$
using *eval(2)* **by** *blast*
then obtain *xvv* **where** *xvv*: *xv* = *IntVal* 64 *xvv*
by (*smt (verit) ConstantExprE bin-eval.simps(2) evalDet intval-bits.simps int-*

```

val-mul.elims
  new-int-bin.simps unfold-binary eval)
obtain yv where yv:  $[m, p] \vdash y \mapsto yv$ 
  using eval(1) eval(2) by blast
then have lhs:  $[m, p] \vdash \text{exp}[x * y] \mapsto \text{val}[xv * yv]$ 
  by (metis bin-eval.simps(2) eval(1) eval(2) evalDet unfold-binary xv)
have  $[m, p] \vdash \text{exp}[\text{const } (\text{IntVal } 64 \ i)] \mapsto \text{val}[(\text{IntVal } 64 \ i)]$ 
  by (smt (verit, ccfv-SIG) ConstantExpr constantAsStamp.simps(1) eval-bits-1-64
take-bit64
  validStampIntConst wf-value-def valid-value.simps(1) xv xvv)
then have rhs:  $[m, p] \vdash \text{exp}[x << \text{const } (\text{IntVal } 64 \ i)] \mapsto \text{val}[xv << (\text{IntVal } 64 \ i)]$ 
using xv xvv using evaltree.BinaryExpr
by (metis Value.simps(5) bin-eval.simps(8) intval-left-shift.simps(1) new-int.simps)
have  $\text{val}[xv * yv] = \text{val}[xv << (\text{IntVal } 64 \ i)]$ 
  by (metis ConstantExprE eval(1) evaltree-not-undef lhs yv val-MulPower2)
then show ?thesis
  by (metis eval(1) eval(2) evalDet lhs rhs)
qed
done

```

```

optimization MulPower2Add1:  $x * y \mapsto (x << \text{const } (\text{IntVal } 64 \ i)) + x$ 
  when ( $i > 0 \wedge$ 
     $64 > i \wedge$ 
     $y = \text{ConstantExpr } (\text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) + 1))$ )
defer
apply simp apply (rule impI; (rule allI)+; rule impI)
subgoal premises p for m p v
proof –
  obtain xv where xv:  $[m, p] \vdash x \mapsto xv$ 
  using p by fast
  then obtain xvv where xvv:  $xv = \text{IntVal } 64 \ xvv$ 
  by (smt (verit) p ConstantExprE bin-eval.simps(2) evalDet intval-bits.simps
intval-mul.elims
  new-int-bin.simps unfold-binary)
obtain yv where yv:  $[m, p] \vdash y \mapsto yv$ 
  using p by blast
have ygezero:  $y > \text{ConstantExpr } (\text{IntVal } 64 \ 0)$ 
  using greaterConstant p wf-value-def by fastforce
then have 1:  $0 < i \wedge$ 
   $i < 64 \wedge$ 
   $y = \text{ConstantExpr } (\text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) + 1))$ 
  using p by blast
then have lhs:  $[m, p] \vdash \text{exp}[x * y] \mapsto \text{val}[xv * yv]$ 
  by (metis bin-eval.simps(2) evalDet p(1) p(2) xv yv unfold-binary)
then have  $[m, p] \vdash \text{exp}[\text{const } (\text{IntVal } 64 \ i)] \mapsto \text{val}[(\text{IntVal } 64 \ i)]$ 
  by (metis wf-value-def verit-comp-simplify1(2) zero-less-numeral ConstantExpr

```

```

      constantAsStamp.simps(1) take-bit64 validStampIntConst valid-value.simps(1))
    then have rhs2: [m, p] ⊢ exp[x << const (IntVal 64 i)] ↦ val[xv << (IntVal
64 i)]
    by (metis Value.simps(5) bin-eval.simps(8) intval-left-shift.simps(1) new-int.simps
xv xvv
      evaltree.BinaryExpr)
    then have rhs: [m, p] ⊢ exp[(x << const (IntVal 64 i)) + x] ↦ val[(xv <<
(IntVal 64 i)) + xv]
    by (metis (no-types, lifting) intval-add.simps(1) rhs2 bin-eval.simps(1)
Value.simps(5)
      evaltree.BinaryExpr intval-left-shift.simps(1) new-int.simps xv xvv)
    then have simple: val[xv * (IntVal 64 (2 ^ unat(i)))] = val[xv << (IntVal 64
i)]
    using val-MulPower2 sorry
    then have val[xv * yv] = val[(xv << (IntVal 64 i)) + xv]
    sorry
    then show ?thesis
    by (metis 1 evalDet lhs p(2) rhs)
qed
done

```

```

optimization MulPower2Sub1:  $x * y \mapsto (x \ll \text{const } (\text{IntVal } 64 \ i)) - x$ 
      when ( $i > 0 \wedge$ 
       $64 > i \wedge$ 
       $y = \text{ConstantExpr } (\text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) - 1))$ )

defer
apply simp apply (rule impI; (rule allI)+; rule impI)
subgoal premises p for m p v
proof –
  obtain xv where xv: [m,p] ⊢ x ↦ xv
  using p by fast
  then obtain xvv where xvv: xv = IntVal 64 xvv
  by (smt (verit) p ConstantExprE bin-eval.simps(2) evalDet intval-bits.simps
intval-mul.elims
      new-int-bin.simps unfold-binary)
  obtain yv where yv: [m,p] ⊢ y ↦ yv
  using p by blast
  have ygezero: y > ConstantExpr (IntVal 64 0)
  by (smt (verit, del-insts) eq-iff-diff-eq-0 mask-0 mask-eq-exp-minus-1 power-inject-exp

      uint-2p unat-eq-zero word-gt-0 zero-neq-one greaterConstant p)
  then have 1:  $0 < i \wedge$ 
       $i < 64 \wedge$ 
       $y = \text{ConstantExpr } (\text{IntVal } 64 \ ((2 \wedge \text{unat}(i)) - 1))$ 
  using p by blast
  then have lhs: [m, p] ⊢ exp[x * y] ↦ val[xv * yv]
  by (metis bin-eval.simps(2) evalDet p(1) p(2) xv yv unfold-binary)
  then have [m, p] ⊢ exp[const (IntVal 64 i)] ↦ val[(IntVal 64 i)]

```

```

    by (metis wf-value-def verit-comp-simplify1 (2) zero-less-numeral ConstantExpr

        constantAsStamp.simps(1) take-bit64 validStampIntConst valid-value.simps(1))
  then have rhs2:  $[m, p] \vdash \text{exp}[x \ll \text{const } (\text{IntVal } 64 \ i)] \mapsto \text{val}[xv \ll (\text{IntVal } 64 \ i)]$ 
  by (metis Value.simps(5) bin-eval.simps(8) intval-left-shift.simps(1) new-int.simps
    xv xvv
        evaltree.BinaryExpr)
  then have rhs:  $[m, p] \vdash \text{exp}[(x \ll \text{const } (\text{IntVal } 64 \ i)) - x] \mapsto \text{val}[(xv \ll (\text{IntVal } 64 \ i)) - xv]$ 
  by (smt (verit, ccfv-threshold) bin-eval.simps(3) new-int-bin.simps intval-sub.simps(1)
    rhs2 bin-eval.simps(1) Value.simps(5) evaltree.BinaryExpr intval-left-shift.simps(1)
    new-int.simps xv xvv )
  then have  $\text{val}[xv * yv] = \text{val}[(xv \ll (\text{IntVal } 64 \ i)) - xv]$ 
  using 1 exp-MulPower2Sub1 ygezero sorry
  then show ?thesis
  by (metis evalDet lhs p(1) p(2) rhs)
qed
done

```

end

end

9.6 NotNode Phase

```

theory NotPhase
  imports
    Common
begin

phase NotNode
  terminating size
begin

```

```

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

```

```

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

```

```

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

      intval-not.simps(2) intval-not.simps(3) intval-not.simps(4))

```

Optimisations

```

optimization NotCancel:  $\text{exp}[\sim(\sim a)] \mapsto a$ 
  by (metis exp-not-cancel)

```

end

end

9.7 OrNode Phase

```

theory OrPhase

```

```

  imports

```

```

    Common

```

```

begin

```

```

context stamp-mask

```

```

begin

```

Taking advantage of the truth table of or operations.

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

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

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

```

lemma OrLeftFallthrough:
  assumes (and (not ( $\downarrow x$ )) ( $\uparrow y$ )) = 0
  shows  $\text{exp}[x \mid y] \geq \text{exp}[x]$ 
  using assms
  apply simp apply ((rule allI)+; rule impI)
  subgoal premises eval for m p v
  proof –
    obtain b vv where e:  $[m, p] \vdash \text{exp}[x \mid y] \mapsto \text{IntVal } b \ vv$ 
      by (metis BinaryExprE bin-eval-new-int new-int.simps eval)
    from e obtain xv where xv:  $[m, p] \vdash x \mapsto \text{IntVal } b \ xv$ 

```

```

    apply (subst (asm) unfold-binary-width)
  by force+
from e obtain yv where yv: [m, p] ⊢ y ↦ IntVal b yv
  apply (subst (asm) unfold-binary-width)
  by force+
have vdef: v = intval-or (IntVal b xv) (IntVal b yv)
  by (metis bin-eval.simps(5) eval(2) evalDet unfold-binary xv yv)
have ∀ i. (bit xv i) | (bit yv i) = (bit xv i)
  by (metis assms bit-and-iff not-down-up-mask-and-zero-implies-zero xv yv)
then have IntVal b xv = intval-or (IntVal b xv) (IntVal b yv)
  by (smt (verit, ccfv-threshold) and.idem assms bit.conj-disj-distrib eval-unused-bits-zero

    intval-or.simps(1) new-int.simps new-int-bin.simps not-down-up-mask-and-zero-implies-zero

    word-ao-absorbs(3) xv yv)
then show ?thesis
  using xv vdef by presburger
qed
done

lemma OrRightFallthrough:
  assumes (and (not (↓y)) (↑x)) = 0
  shows exp[x | y] ≥ exp[y]
  using assms
  apply simp apply ((rule allI)+; rule impI)
  subgoal premises eval for m p v
  proof -
    obtain b vv where e: [m, p] ⊢ exp[x | y] ↦ IntVal b vv
      by (metis BinaryExprE bin-eval-new-int new-int.simps eval)
    from e obtain xv where xv: [m, p] ⊢ x ↦ IntVal b xv
      apply (subst (asm) unfold-binary-width)
      by force+
    from e obtain yv where yv: [m, p] ⊢ y ↦ IntVal b yv
      apply (subst (asm) unfold-binary-width)
      by force+
    have vdef: v = intval-or (IntVal b xv) (IntVal b yv)
      by (metis bin-eval.simps(5) eval(2) evalDet unfold-binary xv yv)
    have ∀ i. (bit xv i) | (bit yv i) = (bit yv i)
      by (metis assms bit-and-iff not-down-up-mask-and-zero-implies-zero xv yv)
    then have IntVal b yv = intval-or (IntVal b xv) (IntVal b yv)
      by (metis (no-types, lifting) assms eval-unused-bits-zero intval-or.simps(1)
new-int.elims
    new-int-bin.elims stamp-mask.not-down-up-mask-and-zero-implies-zero
stamp-mask-axioms
    word-ao-absorbs(8) xv yv)
  then show ?thesis
    using vdef yv by presburger
qed
done

```


end

phase *OrNode*
 terminating *size*
begin

lemma *bin-or-equal*:
 $\text{bin}[x \mid x] = \text{bin}[x]$
 by *simp*

lemma *bin-shift-const-right-helper*:
 $x \mid y = y \mid x$
 by *simp*

lemma *bin-or-not-operands*:
 $(\sim x \mid \sim y) = (\sim (x \& y))$
 by *simp*

lemma *val-or-equal*:
 assumes $x = \text{new-int } b \ v$
 and $(\text{val}[x \mid x] \neq \text{UndefVal})$
 shows $\text{val}[x \mid x] = \text{val}[x]$
 apply (*cases* x ; *auto*) **using** *bin-or-equal* *assms*
 by *auto*+

lemma *val-elim-redundant-false*:
 assumes $x = \text{new-int } b \ v$
 and $\text{val}[x \mid \text{false}] \neq \text{UndefVal}$
 shows $\text{val}[x \mid \text{false}] = \text{val}[x]$
 using *assms* **apply** (*cases* x ; *auto*) **by** *presburger*

lemma *val-shift-const-right-helper*:
 $\text{val}[x \mid y] = \text{val}[y \mid x]$
 apply (*cases* x ; *cases* y ; *auto*)
 by (*simp* *add: or.commute*)+

lemma *val-or-not-operands*:
 $\text{val}[\sim x \mid \sim y] = \text{val}[\sim (x \& y)]$
 apply (*cases* x ; *cases* y ; *auto*)
 by (*simp* *add: take-bit-not-take-bit*)

lemma *exp-or-equal*:
 $\text{exp}[x \mid x] \geq \text{exp}[x]$
 using *val-or-equal* **apply** *auto*
 by (*smt* (*verit*, *ccfv-SIG*) *evalDet* *eval-unused-bits-zero* *intval-negate.elims* *int-*

```

val-or.simps(2)
  intval-or.simps(6) intval-or.simps(7) new-int.simps val-or-equal)

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

    new-int-bin.simps val-elim-redundant-false)

Optimisations

optimization OrEqual:  $x \mid x \mapsto x$ 
  by (meson exp-or-equal)

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

optimization EliminateRedundantFalse:  $x \mid \text{false} \mapsto x$ 
  by (meson exp-elim-redundant-false)

optimization OrNotOperands:  $(\sim x \mid \sim y) \mapsto \sim(x \ \& \ y)$ 
  apply (metis add-2-eq-Suc' less-SucI not-add-less1 not-less-eq size-binary-const
size-non-add)
  apply auto
  by (metis BinaryExpr UnaryExpr bin-eval.simps(4) intval-not.simps(2) unary-eval.simps(3)

    val-or-not-operands)

optimization OrLeftFallthrough:
   $x \mid y \mapsto x$  when  $((\text{and } (\text{not } (\text{IExpr-down } x)) (\text{IExpr-up } y))) = 0$ 
  using simple-mask.OrLeftFallthrough by blast

optimization OrRightFallthrough:
   $x \mid y \mapsto y$  when  $((\text{and } (\text{not } (\text{IExpr-down } y)) (\text{IExpr-up } x))) = 0$ 
  using simple-mask.OrRightFallthrough by blast

end

end

```

9.8 SubNode Phase

```

theory SubPhase
  imports
    Common

```

```

    Proofs.StampEvalThms
begin

phase SubNode
  terminating size
begin

lemma bin-sub-after-right-add:
  shows  $((x::('a::len) \text{ word}) + (y::('a::len) \text{ word})) - y = x$ 
  by simp

lemma sub-self-is-zero:
  shows  $(x::('a::len) \text{ word}) - x = 0$ 
  by simp

lemma bin-sub-then-left-add:
  shows  $(x::('a::len) \text{ word}) - (x + (y::('a::len) \text{ word})) = -y$ 
  by simp

lemma bin-sub-then-left-sub:
  shows  $(x::('a::len) \text{ word}) - (x - (y::('a::len) \text{ word})) = y$ 
  by simp

lemma bin-subtract-zero:
  shows  $(x :: 'a::len \text{ word}) - (0 :: 'a::len \text{ word}) = x$ 
  by simp

lemma bin-sub-negative-value:
  shows  $(x :: ('a::len) \text{ word}) - (-(y :: ('a::len) \text{ word})) = x + y$ 
  by simp

lemma bin-sub-self-is-zero:
  shows  $(x :: ('a::len) \text{ word}) - x = 0$ 
  by simp

lemma bin-sub-negative-const:
  shows  $(x :: 'a::len \text{ word}) - (-(y :: 'a::len \text{ word})) = x + y$ 
  by simp

lemma val-sub-after-right-add-2:
  assumes  $x = \text{new-int } b \ v$ 
  assumes  $\text{val}[(x + y) - y] \neq \text{UndefVal}$ 
  shows  $\text{val}[(x + y) - y] = \text{val}[x]$ 
  using bin-sub-after-right-add
  using assms apply (cases x; cases y; auto)
  by (metis (full-types) intval-sub.simps(2))

```

lemma *val-sub-after-left-sub*:
assumes $\text{val}[(x - y) - x] \neq \text{UndefVal}$
shows $\text{val}[(x - y) - x] = \text{val}[-y]$
using *assms* **apply** (*cases* *x*; *cases* *y*; *auto*)
using *intval-sub.elims* **by** *fastforce*

lemma *val-sub-then-left-sub*:
assumes $y = \text{new-int } b \ v$
assumes $\text{val}[x - (x - y)] \neq \text{UndefVal}$
shows $\text{val}[x - (x - y)] = \text{val}[y]$
using *assms* **apply** (*cases* *x*; *cases* *y*; *auto*)
by (*metis* (*mono-tags*) *intval-sub.simps*(5))

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

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

lemma *val-sub-then-left-add*:
assumes $\text{val}[x - (x + y)] \neq \text{UndefVal}$
shows $\text{val}[x - (x + y)] = \text{val}[-y]$
using *assms* **apply** (*cases* *x*; *cases* *y*; *auto*)
by (*metis* (*mono-tags*, *lifting*) *intval-sub.simps*(5))

lemma *val-sub-negative-value*:
assumes $\text{val}[x - (-y)] \neq \text{UndefVal}$
shows $\text{val}[x - (-y)] = \text{val}[x + y]$
using *assms* **by** (*cases* *x*; *cases* *y*; *auto*)

lemma *val-sub-self-is-zero*:
assumes $x = \text{new-int } b \ v \wedge \text{val}[x - x] \neq \text{UndefVal}$
shows $\text{val}[x - x] = \text{new-int } b \ 0$
using *assms* **by** (*cases* *x*; *auto*)

lemma *val-sub-negative-const*:
assumes $y = \text{new-int } b \ v \wedge \text{val}[x - (-y)] \neq \text{UndefVal}$
shows $\text{val}[x - (-y)] = \text{val}[x + y]$
using *assms* **by** (*cases* *x*; *cases* *y*; *auto*)

lemma *exp-sub-after-right-add*:
shows $\text{exp}[(x + y) - y] \geq \text{exp}[x]$

```

apply auto
by (smt (verit) evalDet eval-unused-bits-zero intval-add.elims new-int.simps
      val-sub-after-right-add-2)

lemma exp-sub-after-right-add2:
  shows  $\exp[(x + y) - x] \geq \exp[y]$ 
  using exp-sub-after-right-add apply auto
  by (smt (z3) Value.inject(1) diff-eq-eq evalDet eval-unused-bits-zero intval-add.elims

      intval-sub.elims new-int.simps new-int-bin.simps take-bit-dist-subL bin-eval.simps(1)

      bin-eval.simps(3) intval-add-sym unfold-binary)

lemma exp-sub-negative-value:
   $\exp[x - (-y)] \geq \exp[x + y]$ 
  apply simp
  by (smt (verit) bin-eval.simps(1) bin-eval.simps(3) evaltree-not-undef unary-eval.simps(2)

      unfold-binary unfold-unary val-sub-negative-value)

lemma exp-sub-then-left-sub:
   $\exp[x - (x - y)] \geq \exp[y]$ 
  using val-sub-then-left-sub apply auto
  subgoal premises  $p$  for  $m$   $p$   $xa$   $xaa$   $ya$ 
  proof –
    obtain  $xa$  where  $xa: [m, p] \vdash x \mapsto xa$ 
    using  $p(2)$  by blast
    obtain  $ya$  where  $ya: [m, p] \vdash y \mapsto ya$ 
    using  $p(5)$  by auto
    obtain  $xaa$  where  $xaa: [m, p] \vdash x \mapsto xaa$ 
    using  $p(2)$  by blast
    have  $1: \text{val}[xa - (xaa - ya)] \neq \text{UndefVal}$ 
    by (metis evalDet  $p(2)$   $p(3)$   $p(4)$   $p(5)$   $xa$   $xaa$   $ya$ )
    then have  $\text{val}[xaa - ya] \neq \text{UndefVal}$ 
    by auto
    then have  $[m, p] \vdash y \mapsto \text{val}[xa - (xaa - ya)]$ 
    by (metis 1 Value.exhaust evalDet eval-unused-bits-zero evaltree-not-undef
      intval-sub.simps(6) intval-sub.simps(7) new-int.simps  $p(5)$  val-sub-then-left-sub
       $xa$   $xaa$ 
       $ya$ )
    then show ?thesis
    by (metis evalDet  $p(2)$   $p(4)$   $p(5)$   $xa$   $xaa$   $ya$ )
  qed
done

thm-oracles exp-sub-then-left-sub

Optimisations

optimization SubAfterAddRight:  $((x + y) - y) \mapsto x$ 

```

```

using exp-sub-after-right-add by blast

optimization SubAfterAddLeft:  $((x + y) - x) \mapsto y$ 
using exp-sub-after-right-add2 by blast

optimization SubAfterSubLeft:  $((x - y) - x) \mapsto -y$ 
apply (metis Suc-lessI add-2-eq-Suc' add-less-cancel-right less-trans-Suc not-add-less1

      size-binary-const size-binary-lhs size-binary-rhs size-non-add)
apply auto
by (metis evalDet unary-eval.simps(2) unfold-unary val-sub-after-left-sub)

optimization SubThenAddLeft:  $(x - (x + y)) \mapsto -y$ 
apply auto
by (metis evalDet unary-eval.simps(2) unfold-unary val-sub-then-left-add)

optimization SubThenAddRight:  $(y - (x + y)) \mapsto -x$ 
apply auto
by (metis evalDet intval-add-sym unary-eval.simps(2) unfold-unary val-sub-then-left-add)

optimization SubThenSubLeft:  $(x - (x - y)) \mapsto y$ 
using size-simps apply simp
using exp-sub-then-left-sub by blast

optimization SubtractZero:  $(x - (\text{const IntVal } b \ 0)) \mapsto x$ 
apply auto
by (smt (verit) add.right-neutral diff-add-cancel eval-unused-bits-zero intval-sub.elims

      intval-word.simps new-int.simps new-int-bin.simps)

thm-oracles SubtractZero

optimization SubNegativeValue:  $(x - (-y)) \mapsto x + y$ 
apply (metis add-2-eq-Suc' less-SucI less-add-Suc1 not-less-eq size-binary-const
      size-non-add)
using exp-sub-negative-value by simp

thm-oracles SubNegativeValue

lemma negate-idempotent:
assumes  $x = \text{IntVal } b \ v \wedge \text{take-bit } b \ v = v$ 
shows  $x = \text{val}[-(-x)]$ 
using assms
using is-IntVal-def by force

```

```

optimization ZeroSubtractValue:  $((\text{const IntVal } b \ 0) - x) \mapsto (-x)$ 
                                when  $(\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \ \text{lo}$ 
                                 $hi \wedge \neg(\text{is-ConstantExpr } x))$ 
  defer
  apply auto unfolding wf-stamp-def
  apply (smt (verit) diff-0 intval-negate.simps(1) intval-sub.elims intval-word.simps

                                new-int-bin.simps unary-eval.simps(2) unfold-unary)
  using add-2-eq-Suc' size.simps(2) size-flip-binary by presburger

```

```

optimization SubSelfIsZero:  $(x - x) \mapsto \text{const IntVal } b \ 0$  when
                                 $(\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{IntegerStamp } b \ \text{lo } hi)$ 
  apply simp-all
  apply auto
  using IRExpr.disc(42) One-nat-def size-non-const apply presburger
  by (smt (verit, best) wf-value-def ConstantExpr evalDet eval-bits-1-64 eval-unused-bits-zero

                                new-int.simps take-bit-of-0 val-sub-self-is-zero validDefIntConst valid-int wf-stamp-def)

```

end

end

9.9 XorNode Phase

```

theory XorPhase
  imports
    Common
    Proofs.StampEvalThms
begin

  phase XorNode
    terminating size
begin

```

```

lemma bin-xor-self-is-false:
   $\text{bin}[x \oplus x] = 0$ 
  by simp

```

```

lemma bin-xor-commute:
   $\text{bin}[x \oplus y] = \text{bin}[y \oplus x]$ 
  by (simp add: xor.commute)

```

lemma *bin-eliminate-redundant-false*:
 $\text{bin}[x \oplus 0] = \text{bin}[x]$
by *simp*

lemma *val-xor-self-is-false*:
assumes $\text{val}[x \oplus x] \neq \text{UndefVal}$
shows $\text{val-to-bool}(\text{val}[x \oplus x]) = \text{False}$
using *assms* **by** (*cases* *x*; *auto*)

lemma *val-xor-self-is-false-2*:
assumes $(\text{val}[x \oplus x]) \neq \text{UndefVal}$
and $x = \text{IntVal } 32 \ v$
shows $\text{val}[x \oplus x] = \text{bool-to-val } \text{False}$
using *assms* **by** (*cases* *x*; *auto*)

lemma *val-xor-self-is-false-3*:
assumes $\text{val}[x \oplus x] \neq \text{UndefVal} \wedge x = \text{IntVal } 64 \ v$
shows $\text{val}[x \oplus x] = \text{IntVal } 64 \ 0$
using *assms* **by** (*cases* *x*; *auto*)

lemma *val-xor-commute*:
 $\text{val}[x \oplus y] = \text{val}[y \oplus x]$
apply (*cases* *x*; *cases* *y*; *auto*)
by (*simp* *add*: *xor.commute*)**+**

lemma *val-eliminate-redundant-false*:
assumes $x = \text{new-int } b \ v$
assumes $\text{val}[x \oplus (\text{bool-to-val } \text{False})] \neq \text{UndefVal}$
shows $\text{val}[x \oplus (\text{bool-to-val } \text{False})] = x$
using *assms* **apply** (*cases* *x*; *auto*)
by *meson*

lemma *exp-xor-self-is-false*:
assumes $\text{wf-stamp } x \wedge \text{stamp-expr } x = \text{default-stamp}$
shows $\text{exp}[x \oplus x] \geq \text{exp}[\text{false}]$
using *assms* **apply** *auto* **unfolding** *wf-stamp-def*
by (*smt* (*z3*) *validDefIntConst* *IntVal0* *Value.inject*(1) *bool-to-val.simps*(2)
constantAsStamp.simps(1) *evalDet int-signed-value-bounds new-int.simps unf-*
fold-const
val-xor-self-is-false-2 *valid-int* *valid-stamp.simps*(1) *valid-value.simps*(1) *wf-value-def*)

lemma *exp-eliminate-redundant-false*:
shows $\text{exp}[x \oplus \text{false}] \geq \text{exp}[x]$
using *val-eliminate-redundant-false* **apply** *auto*
subgoal *premises* *p* **for** *m* *p* *xa*
proof $-$


```

obtain xa where xa:  $[m, p] \vdash x \mapsto xa$ 
  using p(2) by blast
then have  $val[xa \oplus (IntVal\ 32\ 0)] \neq UndefinedVal$ 
  using evalDet p(2) p(3) by blast
then have  $[m, p] \vdash x \mapsto val[xa \oplus (IntVal\ 32\ 0)]$ 
  apply (cases xa; auto) using eval-unused-bits-zero xa by auto
then show ?thesis
  using evalDet p(2) xa by blast
qed
done

```

Optimisations

```

optimization XorSelfIsFalse:  $(x \oplus x) \mapsto false$  when
   $(wf\_stamp\ x \wedge stamp\_expr\ x = default\_stamp)$ 
  using size-non-const apply force
  using exp-xor-self-is-false by auto

optimization XorShiftConstantRight:  $((const\ x) \oplus y) \mapsto y \oplus (const\ x)$  when
 $\neg(is\_ConstantExpr\ y)$ 
  using size-flip-binary apply force
  unfolding le-expr-def using val-xor-commute
  by auto

optimization EliminateRedundantFalse:  $(x \oplus false) \mapsto x$ 
  using exp-eliminate-redundant-false by blast

```

end

end

10 Verifying term graph optimizations using Isabelle/HOL

```

theory TreeSnippets
imports
  Canonicalizations.BinaryNode
  Canonicalizations.ConditionalPhase
  Canonicalizations.AddPhase
  Semantics.TreeToGraphThms
  Snippets.Snipping
  HOL-Library.OptionalSugar
begin

```

— First, we disable undesirable markup.

declare $[[show-types=false, show-sorts=false]]$
no-notation *ConditionalExpr* (- ? - : -)

— We want to disable and reduce how aggressive automated tactics are as obligations are generated in the paper

method *unfold-size* = –
method *unfold-optimization* =
 (*unfold* *rewrite-preservation.simps*, *unfold* *rewrite-termination.simps*,
rule *conjE*, *simp*, *simp* *del*: *le-expr-def*)

10.1 Markup syntax for common operations

notation (*latex*)
kind (-⟨-⟩)

notation (*latex*)
valid-value (- ∈ -)

notation (*latex*)
val-to-bool (*bool-of* -)

notation (*latex*)
constantAsStamp (*stamp-from-value* -)

notation (*latex*)
find-node-and-stamp (*find-matching* -)

notation (*latex*)
add-node (*insert* -)

notation (*latex*)
get-fresh-id (*fresh-id* -)

notation (*latex*)
size (*trm*(-))

10.2 Representing canonicalization optimizations

We wish to provide an example of the semantics layers at which optimizations can be expressed.

lemma *diff-self*:
fixes $x :: int$
shows $x - x = 0$
by *simp*
lemma *diff-diff-cancel*:
fixes $x y :: int$
shows $x - (x - y) = y$

```

by simp
thm diff-self
thm diff-diff-cancel

```

algebraic-laws

$$x - x = 0 \quad (1)$$

$$x - (x - y) = y \quad (2)$$

lemma *diff-self-value*: $\forall x :: 'a :: \text{len word}. x - x = 0$

by *simp*

lemma *diff-diff-cancel-value*:

$\forall x y :: 'a :: \text{len word}. x - (x - y) = y$

by *simp*

algebraic-laws-values

$$\forall x :: 'a \text{ word}. x - x = (0 :: 'a \text{ word}) \quad (3)$$

$$\forall (x :: 'a \text{ word}) y :: 'a \text{ word}. x - (x - y) = y \quad (4)$$

translations

$n \leq \text{CONST ConstantExpr } (\text{CONST IntVal } b \ n)$

$x - y \leq \text{CONST BinaryExpr } (\text{CONST BinSub}) \ x \ y$

notation (*ExprRule* **output**)

Refines ($- \mapsto -$)

lemma *diff-self-expr*:

assumes $\forall m \ p \ v. [m, p] \vdash \text{exp}[x - x] \mapsto \text{IntVal } b \ v$

shows $\text{exp}[x - x] \geq \text{exp}[\text{const } (\text{IntVal } b \ 0)]$

using *assms* **apply** *simp*

by (*metis*(*full-types*) *evalDet val-to-bool.simps*(1) *zero-neq-one*)

method *open-eval* = (*simp*; (*rule impI*)?; (*rule allI*)⁺; *rule impI*)

lemma *diff-diff-cancel-expr*:

shows $\text{exp}[x - (x - y)] \geq \text{exp}[y]$

apply *open-eval*

subgoal **premises** *eval* **for** $m \ p \ v$

proof –

obtain *vx* **where** $vx: [m, p] \vdash x \mapsto vx$

using *eval* **by** *blast*

obtain *vy* **where** $vy: [m, p] \vdash y \mapsto vy$

using *eval* **by** *blast*

then **have** $e: [m, p] \vdash \text{exp}[x - (x - y)] \mapsto \text{val}[vx - (vx - vy)]$

using *vx vy eval*

by (*smt* (*verit*, *ccfv-SIG*) *bin-eval.simps*(3) *evalDet unfold-binary*)

then **have** *notUn*: $\text{val}[vx - (vx - vy)] \neq \text{UndefVal}$

```

    using evaltree-not-undef by auto
  then have val[ $vx - (vx - vy)$ ] =  $vy$ 
    apply (cases  $vx$ ; cases  $vy$ ; auto simp: notUn)
    using eval-unused-bits-zero  $vy$  apply blast
    by (metis (full-types) intval-sub.simps(5))
  then show ?thesis
    by (metis  $e$  eval evalDet  $vy$ )
qed
done

```

thm-oracles *diff-diff-cancel-expr*

algebraic-laws-expressions

$$x - x \mapsto 0 \quad (5)$$

$$x - (x - y) \mapsto y \quad (6)$$

no-translations

```

 $n$  <= CONST ConstantExpr (CONST IntVal  $b$   $n$ )
 $x - y$  <= CONST BinaryExpr (CONST BinSub)  $x$   $y$ 

```

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 *wf-stamp-eval*:

```

assumes wf-stamp  $e$ 
assumes stamp-expr  $e = \text{IntegerStamp } b\ lo\ hi$ 
shows  $\forall m\ p\ v. ([m, p] \vdash e \mapsto v) \longrightarrow (\exists vv. v = \text{IntVal } b\ vv)$ 
using assms unfolding wf-stamp-def
using valid-int-same-bits valid-int
by metis

```

phase *SnipPhase*

terminating *size*

begin

lemma *sub-same-val*:

```

assumes  $\text{val}[x - x] = \text{IntVal } b\ v$ 
shows  $\text{val}[x - x] = \text{val}[\text{IntVal } b\ 0]$ 
using assms by (cases  $x$ ; auto)

```

sub-same-32

optimization *SubIdentity*:

```

 $x - x \mapsto \text{ConstantExpr } (\text{IntVal } b\ 0)$ 
  when  $((\text{stamp-expr } \text{exp}[x - x] = \text{IntegerStamp } b\ lo\ hi) \wedge \text{wf-stamp } \text{exp}[x - x])$ 

```

using *IRExpr.disc*(42) *size.simps*(4) *size-non-const*

```

    apply simp
    apply (rule impI) apply simp
  proof -
    assume assms: stamp-binary BinSub (stamp-expr x) (stamp-expr x) = IntegerStamp b lo hi ∧ wf-stamp exp[x - x]
    have ∀ m p v . ([m, p] ⊢ exp[x - x] ↦ v) ⟶ (∃ vv. v = IntVal b vv)
      using assms wf-stamp-eval
      by (metis stamp-expr.simps(2))
    then show ∀ m p v . ([m, p] ⊢ BinaryExpr BinSub x x ↦ v) ⟶ ([m, p] ⊢ ConstantExpr (IntVal b 0) ↦ v)
      using wf-value-def
      by (smt (verit, best) BinaryExprE TreeSnippets.wf-stamp-def assms bin-eval.simps(3)
        constantAsStamp.simps(1) evalDet stamp-expr.simps(2) sub-same-val unfold-const
        valid-stamp.simps(1) valid-value.simps(1))
    qed
  thm-oracles SubIdentity



RedundantSubtract



optimization RedundantSubtract:


$$x - (x - y) \mapsto y$$



    using size-simps apply simp
    using diff-diff-cancel-expr by presburger
  end

```

10.3 Representing terms

We wish to show a simple example of expressions represented as terms.

ast-example

```

BinaryExpr BinAdd
  (BinaryExpr BinMul x x)
  (BinaryExpr BinMul x x)

```

Then we need to show the datatypes that compose the example expression.

abstract-syntax-tree

```
datatype IExpr =  
  UnaryExpr IRUnaryOp IExpr  
| BinaryExpr IRBinaryOp IExpr IExpr  
| ConditionalExpr IExpr IExpr IExpr  
| ParameterExpr nat Stamp  
| LeafExpr nat Stamp  
| ConstantExpr Value  
| ConstantVar (char list)  
| VariableExpr (char list) Stamp
```

value

```
datatype Value = UndefVal  
| IntVal nat (64 word)  
| ObjRef (nat option)  
| ObjStr (char list)
```

10.4 Term semantics

The core expression evaluation functions need to be introduced.

eval

```
unary-eval :: IRUnaryOp ⇒ Value ⇒ Value  
bin-eval  :: IRBinaryOp ⇒ Value ⇒ Value ⇒ Value
```

We then provide the full semantics of IR expressions.

no-translations

$(prop) P \wedge Q \implies R \leq (prop) P \implies Q \implies R$

translations

$(prop) P \implies Q \implies R \leq (prop) P \wedge Q \implies R$

tree-semantics

$$\begin{array}{c}
\frac{[m,p] \vdash xe \mapsto x}{\text{result} = \text{unary-eval } op \ x \quad \text{result} \neq \text{UndefVal}} \\
\frac{[m,p] \vdash \text{UnaryExpr } op \ xe \mapsto \text{result}}{[m,p] \vdash xe \mapsto x \quad [m,p] \vdash ye \mapsto y} \\
\frac{\text{result} = \text{bin-eval } op \ x \ y \quad \text{result} \neq \text{UndefVal}}{[m,p] \vdash \text{BinaryExpr } op \ xe \ ye \mapsto \text{result}} \\
\frac{[m,p] \vdash ce \mapsto \text{cond} \quad \text{cond} \neq \text{UndefVal} \quad \text{branch} = (\text{if bool-of cond then te else fe})}{[m,p] \vdash \text{ConditionalExpr } ce \ te \ fe \mapsto \text{result}} \\
\frac{\text{wf-value } c}{[m,p] \vdash \text{ConstantExpr } c \mapsto c} \quad \frac{i < |p| \quad p[i] \in s}{[m,p] \vdash \text{ParameterExpr } i \ s \mapsto p[i]} \\
\frac{val = m \ n \quad val \in s}{[m,p] \vdash \text{LeafExpr } n \ s \mapsto val}
\end{array}$$

no-translations

$$(prop) \ P \implies Q \implies R \leq (prop) \ P \wedge Q \implies R$$

translations

$$(prop) \ P \wedge Q \implies R \leq (prop) \ P \implies Q \implies R$$

And show that expression evaluation is deterministic.

tree-evaluation-deterministic

$$[m,p] \vdash e \mapsto v_1 \wedge [m,p] \vdash e \mapsto v_2 \implies v_1 = v_2$$

We then want to start demonstrating the obligations for optimizations. For this we define refinement over terms.

expression-refinement

$$e_1 \sqsupseteq e_2 = (\forall m \ p \ v. [m,p] \vdash e_1 \mapsto v \longrightarrow [m,p] \vdash e_2 \mapsto v)$$

To motivate this definition we show the obligations generated by optimization definitions.

phase *SnipPhase*
terminating *size*
begin

InverseLeftSub

optimization *InverseLeftSub*:

$$(x - y) + y \mapsto x$$

InverseLeftSubObligation

1. $\text{trm}(x) < \text{trm}(\text{BinaryExpr BinAdd } (\text{BinaryExpr BinSub } x \ y) \ y)$
2. $\text{BinaryExpr BinAdd } (\text{BinaryExpr BinSub } x \ y) \ y \sqsupseteq x$

using *RedundantSubAdd* **by** *auto*

InverseRightSub

optimization *InverseRightSub*: $y + (x - y) \mapsto x$

InverseRightSubObligation

1. $\text{trm}(x) < \text{trm}(\text{BinaryExpr BinAdd } y \ (\text{BinaryExpr BinSub } x \ y))$
2. $\text{BinaryExpr BinAdd } y \ (\text{BinaryExpr BinSub } x \ y) \sqsupseteq x$

using *RedundantSubAdd2*(2) *rewrite-termination.simps*(1) **apply** *blast*

using *RedundantSubAdd2*(1) *rewrite-preservation.simps*(1) **by** *blast*

end

expression-refinement-monotone

$$x \sqsupseteq x' \implies \text{UnaryExpr op } x \sqsupseteq \text{UnaryExpr op } x'$$

$$x \sqsupseteq x' \wedge y \sqsupseteq y' \implies \text{BinaryExpr op } x \ y \sqsupseteq \text{BinaryExpr op } x' \ y'$$

$$c \sqsupseteq c' \wedge t \sqsupseteq t' \wedge f \sqsupseteq f' \implies \\ \text{ConditionalExpr } c \ t \ f \sqsupseteq \text{ConditionalExpr } c' \ t' \ f'$$

phase *SnipPhase*

terminating *size*

begin

BinaryFoldConstant

optimization *BinaryFoldConstant*: $\text{BinaryExpr op } (\text{const } v1) \ (\text{const } v2) \mapsto \text{ConstantExpr } (\text{bin-eval op } v1 \ v2)$

BinaryFoldConstantObligation

1. $trm(ConstantExpr\ (bin\text{-}eval\ op\ v1\ v2))$
 $< trm(BinaryExpr\ op\ (ConstantExpr\ v1)\ (ConstantExpr\ v2))$
2. $BinaryExpr\ op\ (ConstantExpr\ v1)\ (ConstantExpr\ v2) \sqsubseteq$
 $ConstantExpr\ (bin\text{-}eval\ op\ v1\ v2)$

using *BinaryFoldConstant(1)* **by** *auto*

AddCommuteConstantRight

optimization *AddCommuteConstantRight*:
 $(const\ v) + y \mapsto y + (const\ v)$ when $\neg(is\text{-}ConstantExpr\ y)$

AddCommuteConstantRightObligation

1. $\neg is\text{-}ConstantExpr\ y \longrightarrow$
 $trm(BinaryExpr\ BinAdd\ y\ (ConstantExpr\ v))$
 $< trm(BinaryExpr\ BinAdd\ (ConstantExpr\ v)\ y)$
2. $\neg is\text{-}ConstantExpr\ y \longrightarrow$
 $BinaryExpr\ BinAdd\ (ConstantExpr\ v)\ y \sqsubseteq$
 $BinaryExpr\ BinAdd\ y\ (ConstantExpr\ v)$

using *AddShiftConstantRight* **by** *auto*

AddNeutral

optimization *AddNeutral*: $x + (const\ (IntVal\ 32\ 0)) \mapsto x$

AddNeutralObligation

1. $trm(x) < trm(BinaryExpr\ BinAdd\ x\ (ConstantExpr\ (IntVal\ 32\ 0)))$
2. $BinaryExpr\ BinAdd\ x\ (ConstantExpr\ (IntVal\ 32\ 0)) \sqsubseteq x$

apply *auto*

using *AddNeutral(1)* *rewrite-preservation.simps(1)* **by** *force*

AddToSub

optimization *AddToSub*: $-x + y \mapsto y - x$

AddToSubObligation

1. $\text{trm}(\text{BinaryExpr BinSub } y \ x) < \text{trm}(\text{BinaryExpr BinAdd } (\text{UnaryExpr UnaryNeg } x) \ y)$
2. $\text{BinaryExpr BinAdd } (\text{UnaryExpr UnaryNeg } x) \ y \sqsubseteq \text{BinaryExpr BinSub } y \ x$

using *AddLeftNegateToSub* **by** *auto*

end

definition *trm* **where** *trm* = *size*

lemma *trm-defn*[*size-simps*]:

trm *x* = *size* *x*

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

phase

phase *AddCanonicalizations*

terminating *trm*

begin...**end**

hide-const (**open**) *Form.wf-stamp*

phase-example

phase *Conditional*

terminating *trm*

begin

phase-example-1

optimization *NegateCond*: $((!c) \ ? \ t : f) \mapsto (c \ ? \ f : t)$

apply (*simp* *add*: *size-simps*)

using *ConditionalPhase.NegateConditionFlipBranches*(1) **by** *simp*

phase-example-2

optimization *TrueCond*: $(\text{true} \ ? \ t : f) \mapsto t$

by (*auto* *simp*: *trm-def*)

phase-example-3

optimization *FalseCond*: $(\text{false} \ ? \ t : f) \mapsto f$

by (*auto* *simp*: *trm-def*)

phase-example-4

optimization *BranchEqual*: $(c \text{ ? } x : x) \mapsto x$

by (auto simp: trm-def)

phase-example-5

optimization *LessCond*: $((u < v) \text{ ? } t : f) \mapsto t$
when (stamp-under (stamp-expr u) (stamp-expr v)
 \wedge wf-stamp u \wedge wf-stamp v)

apply (auto simp: trm-def)

using ConditionalPhase.condition-bounds-x(1)

by (metis(full-types) StampEvalThms.wf-stamp-def TreeSnippets.wf-stamp-def bin-eval.simps(12)
stamp-under-defn)

phase-example-6

optimization *condition-bounds-y*: $((x < y) \text{ ? } x : y) \mapsto y$
when (stamp-under (stamp-expr y) (stamp-expr x) \wedge wf-stamp
 $x \wedge$ wf-stamp y)

apply (auto simp: trm-def)

using ConditionalPhase.condition-bounds-y(1)

by (metis(full-types) StampEvalThms.wf-stamp-def TreeSnippets.wf-stamp-def bin-eval.simps(12)
stamp-under-defn-inverse)

phase-example-7

end

lemma *simplified-binary*: $\neg(\text{is-ConstantExpr } b) \implies \text{size } (\text{BinaryExpr op } a \text{ } b) =$
 $\text{size } a + \text{size } b + 2$

by (induction b; induction op; auto simp: is-ConstantExpr-def)

thm *bin-size*

thm *bin-const-size*

thm *unary-size*

thm *size-non-add*

termination

$$\text{trm}(\text{UnaryExpr } op \ x) = \text{trm}(x) + 2$$

$$\text{trm}(\text{BinaryExpr } op \ x \ (\text{ConstantExpr } cy)) = \text{trm}(x) + 2$$

$$\text{trm}(\text{BinaryExpr } op \ a \ b) = \text{trm}(a) + \text{trm}(b) + 2$$

$$\text{trm}(\text{ConditionalExpr } c \ t \ f) = \text{trm}(c) + \text{trm}(t) + \text{trm}(f) + 2$$

$$\text{trm}(\text{ConstantExpr } c) = 1$$

$$\text{trm}(\text{ParameterExpr } ind \ s) = 2$$

$$\text{trm}(\text{LeafExpr } nid \ s) = 2$$

graph-representation

typedef IRGraph =

$$\{g :: ID \rightarrow (IRNode \times Stamp) \cdot \text{finite } (\text{dom } g)\}$$

no-translations

$$(\text{prop}) \ P \wedge Q \implies R \leq (\text{prop}) \ P \implies Q \implies R$$

translations

$$(\text{prop}) \ P \implies Q \implies R \leq (\text{prop}) \ P \wedge Q \implies R$$

graph2tree

$$\frac{g\langle n \rangle = \text{ConstantNode } c \quad g\langle n \rangle = \text{ParameterNode } i \quad \text{stamp } g \ n = s}{g \vdash n \simeq \text{ConstantExpr } c \quad g \vdash n \simeq \text{ParameterExpr } i \ s}$$

$$\frac{g\langle n \rangle = \text{ConditionalNode } c \ t \ f}{g \vdash c \simeq ce \quad g \vdash t \simeq te \quad g \vdash f \simeq fe}$$

$$\frac{g \vdash c \simeq ce \quad g \vdash t \simeq te \quad g \vdash f \simeq fe}{g \vdash n \simeq \text{ConditionalExpr } ce \ te \ fe}$$

$$\frac{g\langle n \rangle = \text{AbsNode } x \quad g \vdash x \simeq xe}{g \vdash n \simeq \text{UnaryExpr } \text{UnaryAbs } xe}$$

$$\frac{g\langle n \rangle = \text{SignExtendNode } inputBits \ resultBits \ x \quad g \vdash x \simeq xe}{g \vdash n \simeq \text{UnaryExpr } \text{UnarySignExtend } inputBits \ resultBits \ xe}$$

$$\frac{g\langle n \rangle = \text{AddNode } x \ y \quad g \vdash x \simeq xe \quad g \vdash y \simeq ye}{g \vdash n \simeq \text{BinaryExpr } \text{BinAdd } xe \ ye}$$

$$\frac{g\langle n \rangle = \text{RefNode } n' \quad g \vdash n' \simeq e}{g \vdash n \simeq e}$$

$$\frac{is\text{-preevaluated } g\langle n \rangle \quad \text{stamp } g \ n = s}{g \vdash n \simeq \text{LeafExpr } n \ s}$$

tree2graph

$$\begin{array}{c}
\frac{\text{find-matching } g2 \text{ (unary-node op } x, s') = \text{Some } n}{g \oplus xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op (stamp } g2 \text{ } x)} \\
\hline
g \oplus \text{UnaryExpr op } xe \rightsquigarrow (g2, n) \\
\text{find-matching } g \text{ (ConstantNode } c, \text{ stamp-from-value } c) = \text{None} \\
n = \text{fresh-id } g \\
\frac{g' = \text{insert } n \text{ (ConstantNode } c, \text{ stamp-from-value } c) \text{ } g}{g \oplus \text{ConstantExpr } c \rightsquigarrow (g', n)} \\
\text{find-matching } g \text{ (ParameterNode } i, s) = \text{None} \\
n = \text{fresh-id } g \quad g' = \text{insert } n \text{ (ParameterNode } i, s) \text{ } g \\
\hline
g \oplus \text{ParameterExpr } i \text{ } s \rightsquigarrow (g', n) \\
\text{find-matching } g2 \text{ (unary-node op } x, s') = \text{None} \\
\frac{g \oplus xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op (stamp } g2 \text{ } x)}{n = \text{fresh-id } g2 \quad g' = \text{insert } n \text{ (unary-node op } x, s') \text{ } g2} \\
\hline
g \oplus \text{UnaryExpr op } xe \rightsquigarrow (g', n) \\
\text{find-matching } g3 \text{ (bin-node op } x \text{ } y, s') = \text{None} \\
\frac{g \oplus xe \rightsquigarrow (g2, x) \quad g2 \oplus ye \rightsquigarrow (g3, y) \quad s' = \text{stamp-binary op (stamp } g3 \text{ } x) \text{ (stamp } g3 \text{ } y)}{n = \text{fresh-id } g3 \quad g' = \text{insert } n \text{ (bin-node op } x \text{ } y, s') \text{ } g3} \\
\hline
g \oplus \text{BinaryExpr op } xe \text{ } ye \rightsquigarrow (g', n) \\
\frac{\text{stamp } g \text{ } n = s \quad \text{is-preevaluated } g \llbracket n \rrbracket}{g \oplus \text{LeafExpr } n \text{ } s \rightsquigarrow (g, n)}
\end{array}$$

no-translations

$$(\text{prop}) P \implies Q \implies R \leq (\text{prop}) P \wedge Q \implies R$$

translations

$$(\text{prop}) P \wedge Q \implies R \leq (\text{prop}) P \implies Q \implies R$$

preeval

is-preevaluated (*InvokeNode* *n uu uv uw ux uy*) = *True*
is-preevaluated (*InvokeWithExceptionNode* *n uz va vb vc vd ve*) = *True*
is-preevaluated (*NewInstanceNode* *n vf vg vh*) = *True*
is-preevaluated (*LoadFieldNode* *n vi vj vk*) = *True*
is-preevaluated (*SignedDivNode* *n vl vm vn vo vp*) = *True*
is-preevaluated (*SignedRemNode* *n vq vr vs vt vu*) = *True*
is-preevaluated (*ValuePhiNode* *n vv vw*) = *True*
is-preevaluated (*AbsNode* *v*) = *False*
is-preevaluated (*AddNode* *v va*) = *False*
is-preevaluated (*AndNode* *v va*) = *False*
is-preevaluated (*BeginNode* *v*) = *False*
is-preevaluated (*BytecodeExceptionNode* *v va vb*) = *False*
is-preevaluated (*ConditionalNode* *v va vb*) = *False*
is-preevaluated (*ConstantNode* *v*) = *False*
is-preevaluated (*DynamicNewArrayNode* *v va vb vc vd*) = *False*
is-preevaluated *EndNode* = *False*
is-preevaluated (*ExceptionObjectNode* *v va*) = *False*
is-preevaluated (*FrameState* *v va vb vc*) = *False*
is-preevaluated (*IfNode* *v va vb*) = *False*
is-preevaluated (*IntegerBelowNode* *v va*) = *False*
is-preevaluated (*IntegerEqualsNode* *v va*) = *False*
is-preevaluated (*IntegerLessThanNode* *v va*) = *False*
is-preevaluated (*IsNullNode* *v*) = *False*
is-preevaluated (*KillingBeginNode* *v*) = *False*
is-preevaluated (*LeftShiftNode* *v va*) = *False*
is-preevaluated (*LogicNegationNode* *v*) = *False*
is-preevaluated (*LoopBeginNode* *v va vb vc*) = *False*
is-preevaluated (*LoopEndNode* *v*) = *False*
is-preevaluated (*LoopExitNode* *v va vb*) = *False*
is-preevaluated (*MergeNode* *v va vb*) = *False*
is-preevaluated (*MethodCallTargetNode* *v va*) = *False*
is-preevaluated (*MulNode* *v va*) = *False*
is-preevaluated (*NarrowNode* *v va vb*) = *False*
is-preevaluated (*NegateNode* *v*) = *False*
is-preevaluated (*NewArrayNode* *v va vb*) = *False*
is-preevaluated (*NotNode* *v*) = *False*
is-preevaluated (*OrNode* *v va*) = *False*
is-preevaluated (*ParameterNode* *v*) = *False*
is-preevaluated (*PiNode* *v va*) = *False*
is-preevaluated (*ReturnNode* *v va*) = *False*
is-preevaluated (*RightShiftNode* *v va*) = *False*
is-preevaluated (*ShortCircuitOrNode* *v va*) = *False*
is-preevaluated (*SignExtendNode* *v va vb*) = *False*

deterministic-representation

$$g \vdash n \simeq e_1 \wedge g \vdash n \simeq e_2 \implies e_1 = e_2$$

thm-oracles *repDet*

well-formed-term-graph

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

graph-semantics

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

graph-semantics-deterministic

$$[g,m,p] \vdash n \mapsto v_1 \wedge [g,m,p] \vdash n \mapsto v_2 \implies v_1 = v_2$$

thm-oracles *graphDet*

notation (*latex*)

graph-refinement (*term-graph-refinement -*)

graph-refinement

$$\begin{aligned} & \text{term-graph-refinement } g_1 \ g_2 = \\ & (\text{ids } g_1 \subseteq \text{ids } g_2 \wedge \\ & (\forall n. n \in \text{ids } g_1 \longrightarrow (\forall e. g_1 \vdash n \simeq e \longrightarrow g_2 \vdash n \sqsubseteq e))) \end{aligned}$$

translations

$n \leq \text{CONST as-set } n$

graph-semantics-preservation

$$\begin{aligned} & e_1' \sqsupseteq e_2' \wedge \\ & \{n\} \triangleleft g_1 \subseteq g_2 \wedge \\ & g_1 \vdash n \simeq e_1' \wedge g_2 \vdash n \simeq e_2' \implies \\ & \text{term-graph-refinement } g_1 \ g_2 \end{aligned}$$

thm-oracles *graph-semantics-preservation-subscript*

maximal-sharing

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

tree-to-graph-rewriting

$e_1 \sqsupseteq e_2 \wedge$
 $g_1 \vdash n \simeq e_1 \wedge$
 $\text{maximal-sharing } g_1 \wedge$
 $\{n\} \triangleleft g_1 \subseteq g_2 \wedge$
 $g_2 \vdash n \simeq e_2 \wedge$
 $\text{maximal-sharing } g_2 \implies$
 $\text{term-graph-refinement } g_1 \ g_2$

thm-oracles *tree-to-graph-rewriting*

term-graph-refines-term

$(g \vdash n \trianglelefteq e) = (\exists e'. g \vdash n \simeq e' \wedge e \sqsupseteq e')$

term-graph-evaluation

$g \vdash n \trianglelefteq e \implies \forall m \ p \ v. [m, p] \vdash e \mapsto v \longrightarrow [g, m, p] \vdash n \mapsto v$

graph-construction

$e_1 \sqsupseteq e_2 \wedge g_1 \subseteq g_2 \wedge g_2 \vdash n \simeq e_2 \implies$
 $g_2 \vdash n \trianglelefteq e_1 \wedge \text{term-graph-refinement } g_1 \ g_2$

thm-oracles *graph-construction*

term-graph-reconstruction

$g \oplus e \rightsquigarrow (g', n) \implies g' \vdash n \simeq e \wedge g \subseteq g'$

refined-insert

$$\begin{array}{l} e_1 \sqsupseteq e_2 \wedge g_1 \oplus e_2 \rightsquigarrow (g_2, n') \implies \\ g_2 \vdash n' \sqsubseteq e_1 \wedge \textit{term-graph-refinement } g_1 \ g_2 \end{array}$$

end