

Veriopt

July 23, 2021

Abstract

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

Contents

| | | |
|----------|---|-----------|
| 1 | Runtime Values and Arithmetic | 3 |
| 2 | Nodes | 8 |
| 2.1 | Types of Nodes | 8 |
| 2.2 | Hierarchy of Nodes | 15 |
| 3 | Stamp Typing | 22 |
| 4 | Graph Representation | 25 |
| 4.0.1 | Example Graphs | 30 |
| 5 | Data-flow Semantics | 30 |
| 5.1 | Data-flow Tree Representation | 32 |
| 5.2 | Data-flow Tree Evaluation | 40 |
| 5.3 | Data-flow Tree Refinement | 42 |
| 6 | Data-flow Expression-Tree Theorems | 42 |
| 6.1 | Extraction and Evaluation of Expression Trees is Deterministic. | 43 |
| 6.2 | Example Data-flow Optimisations | 49 |
| 6.3 | Monotonicity of Expression Optimization | 50 |
| 7 | Control-flow Semantics | 51 |
| 7.1 | Heap | 51 |
| 7.2 | Intraprocedural Semantics | 51 |
| 7.3 | Interprocedural Semantics | 53 |
| 7.4 | Big-step Execution | 55 |
| 7.4.1 | Heap Testing | 56 |
| 8 | Canonicalization Phase | 56 |
| 9 | Canonicalization Phase | 67 |

1 Runtime Values and Arithmetic

```

theory Values2
  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 new type of runtime values. Our evaluation semantics are defined in terms of these 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 eventually arrays.

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.

Java supports 64, 32, 16, 8 signed ints, plus 1 bit (boolean) ints. Our Value type models this by keeping the value as an infinite precision signed int, but also carrying along the number of bits allowed.

So each $(\text{IntVal } b \ v)$ should satisfy the invariants:

$b \in \{1::'a, 8::'a, 16::'a, 32::'a, 64::'a\}$
 $1 < b \implies v \equiv \text{scast } (\text{signed-take-bit } b \ v)$

```

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

```

```

type-synonym objref = nat option

```

```

datatype Value =
  UndefVal |
  IntVal32 int32 |
  IntVal64 int64 |
  FloatVal float |
  ObjRef objref |
  ObjStr string

```

We define integer values to be well-formed when their bit size is valid and their integer value is able to fit within the bit size. This is defined using the *wf-value* function.

— Check that a signed int value does not overflow b bits.

```

fun fits-into-n :: nat  $\Rightarrow$  int  $\Rightarrow$  bool where
  fits-into-n b val = (( $-(2^{b-1}) \leq \text{val}$ )  $\wedge$  ( $\text{val} < 2^{b-1}$ )))

```

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

```
fun val-to-bool :: Value  $\Rightarrow$  bool where
  val-to-bool (IntVal32 v) = (v = 1) |
  val-to-bool - = False
```

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

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

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.

```
fun intval-add32 :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-add32 (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1+v2)) |
  intval-add32 - - = UndefVal
```

```
fun intval-add64 :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-add64 (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1+v2)) |
  intval-add64 - - = UndefVal
```

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

```
instantiation Value :: plus
begin
```

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

```
instance proof qed
end
```

```

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

```

```

instantiation Value :: minus
begin

```

```

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

```

```

instance proof qed
end

```

```

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

```

```

instantiation Value :: times
begin

```

```

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

```

```

instance proof qed
end

```

```

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

```

```

instantiation Value :: divide
begin

```

```

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

```

```

instance proof qed
end

```

```

fun intval-mod :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where

```

```

    intval-mod (IntVal32 v1) (IntVal32 v2) = (IntVal32 (word-of-int((sint v1) smod
(sint v2)))) |
    intval-mod (IntVal64 v1) (IntVal64 v2) = (IntVal64 (word-of-int((sint v1) smod
(sint v2)))) |
    intval-mod - - = UndefVal

```

instantiation *Value* :: *modulo*
begin

definition *modulo-Value* :: *Value* \Rightarrow *Value* \Rightarrow *Value* **where**
modulo-Value = *intval-mod*

instance **proof** **qed**
end

fun *intval-and* :: *Value* \Rightarrow *Value* \Rightarrow *Value* (**infix** &&* 64) **where**
intval-and (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1 AND v2)) |
intval-and (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1 AND v2)) |
intval-and - - = UndefVal

fun *intval-or* :: *Value* \Rightarrow *Value* \Rightarrow *Value* (**infix** ||* 59) **where**
intval-or (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1 OR v2)) |
intval-or (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1 OR v2)) |
intval-or - - = UndefVal

fun *intval-xor* :: *Value* \Rightarrow *Value* \Rightarrow *Value* (**infix** ^* 59) **where**
intval-xor (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1 XOR v2)) |
intval-xor (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1 XOR v2)) |
intval-xor - - = UndefVal

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

fun *intval-less-than* :: *Value* \Rightarrow *Value* \Rightarrow *Value* **where**
intval-less-than (IntVal32 v1) (IntVal32 v2) = *bool-to-val* (v1 <_s v2) |
intval-less-than (IntVal64 v1) (IntVal64 v2) = *bool-to-val* (v1 <_s v2) |
intval-less-than - - = UndefVal

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

fun *intval-not* :: *Value* \Rightarrow *Value* **where**

```

intval-not (IntVal32 v) = (IntVal32 (NOT v)) |
intval-not (IntVal64 v) = (IntVal64 (NOT v)) |
intval-not - = UndefVal

```

```

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

```

```

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

```

```

lemma word-add-sym:
  shows word-of-int v1 + word-of-int v2 = word-of-int v2 + word-of-int v1
  by simp

```

```

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

```

```

lemma word-add-assoc:
  shows (word-of-int v1 + word-of-int v2) + word-of-int v3
    = word-of-int v1 + (word-of-int v2 + word-of-int v3)
  by simp

```

```

lemma intval-bad1 [simp]: intval-add (IntVal32 x) (IntVal64 y) = UndefVal
  by auto

```

```

lemma intval-bad2 [simp]: intval-add (IntVal64 x) (IntVal32 y) = UndefVal
  by auto

```

```

lemma intval-assoc: intval-add32 (intval-add32 x y) z = intval-add32 x (intval-add32
y z)
  apply (induction x)
    apply auto
    apply (induction y)
    apply auto
    apply (induction z)
  by auto

```

```

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

```

```

lemma intval-add (IntVal32 ( $2^{31}-1$ )) (IntVal32 ( $2^{31}-1$ )) = IntVal32 ( $-2$ )
  by eval
lemma intval-add (IntVal64 ( $2^{31}-1$ )) (IntVal64 ( $2^{31}-1$ )) = IntVal64 4294967294
  by eval

end

```

2 Nodes

2.1 Types of Nodes

```

theory IRNodes2
  imports
    Values2
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)
  | 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)
  | 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)
| ShortCircuitOrNode (ir-x: INPUT-COND) (ir-y: INPUT-COND)
| 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)
| 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)
| 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-LoadFieldNode:
inputs-of (*LoadFieldNode* *nid0* *field* *object* *next*) = (*opt-to-list* *object*) |
inputs-of-LogicNegationNode:
inputs-of (*LogicNegationNode* *value*) = [*value*] |
inputs-of-LoopBeginNode:
inputs-of (*LoopBeginNode* *ends* *overflowGuard* *stateAfter* *next*) = *ends* @ (*opt-to-list*
overflowGuard) @ (*opt-to-list* *stateAfter*) |
inputs-of-LoopEndNode:
inputs-of (*LoopEndNode* *loopBegin*) = [*loopBegin*] |
inputs-of-LoopExitNode:

inputs-of (*LoopExitNode* *loopBegin* *stateAfter* *next*) = *loopBegin* # (*opt-to-list* *stateAfter*) |
inputs-of-MergeNode:
inputs-of (*MergeNode* *ends* *stateAfter* *next*) = *ends* @ (*opt-to-list* *stateAfter*) |
inputs-of-MethodCallTargetNode:
inputs-of (*MethodCallTargetNode* *targetMethod* *arguments*) = *arguments* |
inputs-of-MulNode:
inputs-of (*MulNode* *x* *y*) = [*x*, *y*] |
inputs-of-NegateNode:
inputs-of (*NegateNode* *value*) = [*value*] |
inputs-of-NewArrayNode:
inputs-of (*NewArrayNode* *length0* *stateBefore* *next*) = *length0* # (*opt-to-list* *stateBefore*) |
inputs-of-NewInstanceNode:
inputs-of (*NewInstanceNode* *nid0* *instanceClass* *stateBefore* *next*) = (*opt-to-list* *stateBefore*) |
inputs-of-NotNode:
inputs-of (*NotNode* *value*) = [*value*] |
inputs-of-OrNode:
inputs-of (*OrNode* *x* *y*) = [*x*, *y*] |
inputs-of-ParameterNode:
inputs-of (*ParameterNode* *index*) = [] |
inputs-of-PiNode:
inputs-of (*PiNode* *object* *guard*) = *object* # (*opt-to-list* *guard*) |
inputs-of-ReturnNode:
inputs-of (*ReturnNode* *result* *memoryMap*) = (*opt-to-list* *result*) @ (*opt-to-list* *memoryMap*) |
inputs-of-ShortCircuitOrNode:
inputs-of (*ShortCircuitOrNode* *x* *y*) = [*x*, *y*] |
inputs-of-SignedDivNode:
inputs-of (*SignedDivNode* *nid0* *x* *y* *zeroCheck* *stateBefore* *next*) = [*x*, *y*] @ (*opt-to-list* *zeroCheck*) @ (*opt-to-list* *stateBefore*) |
inputs-of-SignedRemNode:
inputs-of (*SignedRemNode* *nid0* *x* *y* *zeroCheck* *stateBefore* *next*) = [*x*, *y*] @ (*opt-to-list* *zeroCheck*) @ (*opt-to-list* *stateBefore*) |
inputs-of-StartNode:
inputs-of (*StartNode* *stateAfter* *next*) = (*opt-to-list* *stateAfter*) |
inputs-of-StoreFieldNode:
inputs-of (*StoreFieldNode* *nid0* *field* *value* *stateAfter* *object* *next*) = *value* # (*opt-to-list* *stateAfter*) @ (*opt-to-list* *object*) |
inputs-of-SubNode:
inputs-of (*SubNode* *x* *y*) = [*x*, *y*] |
inputs-of-UnwindNode:
inputs-of (*UnwindNode* *exception*) = [*exception*] |
inputs-of-ValuePhiNode:
inputs-of (*ValuePhiNode* *nid* *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-NoNode: *inputs-of* (*NoNode*) = [] |

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

fun *successors-of* :: *IRNode* \Rightarrow *ID list* **where**
successors-of-AbsNode:
successors-of (*AbsNode* *value*) = [] |
successors-of-AddNode:
successors-of (*AddNode* *x y*) = [] |
successors-of-AndNode:
successors-of (*AndNode* *x y*) = [] |
successors-of-BeginNode:
successors-of (*BeginNode* *next*) = [*next*] |
successors-of-BytecodeExceptionNode:
successors-of (*BytecodeExceptionNode* *arguments stateAfter next*) = [*next*] |
successors-of-ConditionalNode:
successors-of (*ConditionalNode* *condition trueValue falseValue*) = [] |
successors-of-ConstantNode:
successors-of (*ConstantNode* *const*) = [] |
successors-of-DynamicNewArrayNode:
successors-of (*DynamicNewArrayNode* *elementType length0 voidClass stateBefore*
next) = [*next*] |
successors-of-EndNode:
successors-of (*EndNode*) = [] |
successors-of-ExceptionObjectNode:
successors-of (*ExceptionObjectNode* *stateAfter next*) = [*next*] |
successors-of-FrameState:
successors-of (*FrameState* *monitorIds outerFrameState values virtualObjectMap-*
pings) = [] |
successors-of-IfNode:
successors-of (*IfNode* *condition trueSuccessor falseSuccessor*) = [*trueSuccessor*,
falseSuccessor] |
successors-of-IntegerBelowNode:
successors-of (*IntegerBelowNode* *x y*) = [] |
successors-of-IntegerEqualsNode:
successors-of (*IntegerEqualsNode* *x y*) = [] |
successors-of-IntegerLessThanNode:
successors-of (*IntegerLessThanNode* *x y*) = [] |
successors-of-InvokeNode:
successors-of (*InvokeNode* *nid0 callTarget classInit stateDuring stateAfter next*)
= [*next*] |
successors-of-InvokeWithExceptionNode:
successors-of (*InvokeWithExceptionNode* *nid0 callTarget classInit stateDuring*
stateAfter next exceptionEdge) = [*next*, *exceptionEdge*] |
successors-of-IsNullNode:
successors-of (*IsNullNode* *value*) = [] |

successors-of-KillingBeginNode:
successors-of (KillingBeginNode next) = [next] |
successors-of-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-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-ShortCircuitOrNode:
successors-of (ShortCircuitOrNode x y) = [] |
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-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-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

2.2 Hierarchy of Nodes

```

theory IRNodeHierarchy
imports IRNodes2
begin

```

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

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

These functions have been automatically generated from the compiler.

```

fun is-EndNode :: IRNode ⇒ bool where
  is-EndNode EndNode = True |
  is-EndNode - = False

```

```

fun is-ControlSinkNode :: IRNode ⇒ bool where

```

```

is-ControlSinkNode n = ((is-ReturnNode n) ∨ (is-UnwindNode n))

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

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

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

fun is-AbstractNewArrayNode :: IRNode ⇒ bool where
  is-AbstractNewArrayNode n = ((is-DynamicNewArrayNode n) ∨ (is-NewArrayNode
n))

fun is-AbstractNewObjectNode :: IRNode ⇒ bool where
  is-AbstractNewObjectNode n = ((is-AbstractNewArrayNode n) ∨ (is-NewInstanceNode
n))

fun is-IntegerDivRemNode :: IRNode ⇒ bool where
  is-IntegerDivRemNode n = ((is-SignedDivNode n) ∨ (is-SignedRemNode n))

fun is-FixedBinaryNode :: IRNode ⇒ bool where
  is-FixedBinaryNode n = ((is-IntegerDivRemNode n))

fun is-DeoptimizingFixedWithNextNode :: IRNode ⇒ bool where
  is-DeoptimizingFixedWithNextNode n = ((is-AbstractNewObjectNode n) ∨ (is-FixedBinaryNode
n))

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

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

fun is-AccessFieldNode :: IRNode ⇒ bool where
  is-AccessFieldNode n = ((is-LoadFieldNode n) ∨ (is-StoreFieldNode n))

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

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

fun is-ControlSplitNode :: IRNode ⇒ bool where

```



```

    is-ControlSplitNode n = ((is-IfNode n) ∨ (is-WithExceptionNode n))

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

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

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

fun is-UnaryArithmeticNode :: IRNode ⇒ bool where
    is-UnaryArithmeticNode n = ((is-AbsNode n) ∨ (is-NegateNode n) ∨ (is-NotNode n))

fun is-UnaryNode :: IRNode ⇒ bool where
    is-UnaryNode n = ((is-UnaryArithmeticNode n))

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

fun is-BinaryNode :: IRNode ⇒ bool where
    is-BinaryNode n = ((is-BinaryArithmeticNode n))

fun is-PhiNode :: IRNode ⇒ bool where
    is-PhiNode n = ((is-ValuePhiNode n))

fun is-IntegerLowerThanNode :: IRNode ⇒ bool where
    is-IntegerLowerThanNode n = ((is-IntegerLessThanNode n) ∨ (is-IntegerBelowNode n))

fun is-CompareNode :: IRNode ⇒ bool where
    is-CompareNode n = ((is-IntegerEqualsNode n) ∨ (is-IntegerLowerThanNode n))

fun is-BinaryOpLogicNode :: IRNode ⇒ bool where
    is-BinaryOpLogicNode n = ((is-CompareNode n))

fun is-UnaryOpLogicNode :: IRNode ⇒ bool where
    is-UnaryOpLogicNode n = ((is-IsNullNode n))

fun is-LogicNode :: IRNode ⇒ bool where
    is-LogicNode n = ((is-BinaryOpLogicNode n) ∨ (is-LogicNegationNode n) ∨ (is-ShortCircuitOrNode n) ∨ (is-UnaryOpLogicNode n))

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

```

```

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

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

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

fun is-Node :: IRNode  $\Rightarrow$  bool where
  is-Node n = ((is-ValueNode n)  $\vee$  (is-VirtualState n))

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

fun is-ArithmeticLIRLowerable :: IRNode  $\Rightarrow$  bool where
  is-ArithmeticLIRLowerable n = ((is-AbsNode n)  $\vee$  (is-BinaryArithmeticNode n)  $\vee$  (is-NotNode 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 ⇒ bool where
  is-BinaryCommutative n = ((is-AddNode n) ∨ (is-AndNode n) ∨ (is-IntegerEqualsNode
n) ∨ (is-MulNode n) ∨ (is-OrNode n) ∨ (is-XorNode n))

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

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

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

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

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

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

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

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

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

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

```

```

is-sequential-node (MergeNode - -) = True |
is-sequential-node (RefNode -) = True |
is-sequential-node - = False

```

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

```

fun is-same-ir-node-type :: IRNode ⇒ IRNode ⇒ bool where
is-same-ir-node-type n1 n2 = (
  ((is-AbsNode n1) ∧ (is-AbsNode n2)) ∨
  ((is-AddNode n1) ∧ (is-AddNode n2)) ∨
  ((is-AndNode n1) ∧ (is-AndNode n2)) ∨
  ((is-BEGINNode n1) ∧ (is-BEGINNode n2)) ∨
  ((is-BytecodeExceptionNode n1) ∧ (is-BytecodeExceptionNode n2)) ∨
  ((is-ConditionalNode n1) ∧ (is-ConditionalNode n2)) ∨
  ((is-ConstantNode n1) ∧ (is-ConstantNode n2)) ∨
  ((is-DynamicNewArrayNode n1) ∧ (is-DynamicNewArrayNode n2)) ∨
  ((is-EndNode n1) ∧ (is-EndNode n2)) ∨
  ((is-ExceptionObjectNode n1) ∧ (is-ExceptionObjectNode n2)) ∨
  ((is-FrameState n1) ∧ (is-FrameState n2)) ∨
  ((is-IfNode n1) ∧ (is-IfNode n2)) ∨
  ((is-IntegerBelowNode n1) ∧ (is-IntegerBelowNode n2)) ∨
  ((is-IntegerEqualsNode n1) ∧ (is-IntegerEqualsNode n2)) ∨
  ((is-IntegerLessThanNode n1) ∧ (is-IntegerLessThanNode n2)) ∨
  ((is-InvokeNode n1) ∧ (is-InvokeNode n2)) ∨
  ((is-InvokeWithExceptionNode n1) ∧ (is-InvokeWithExceptionNode n2)) ∨
  ((is-IsNullNode n1) ∧ (is-IsNullNode n2)) ∨
  ((is-Killing-BEGINNode n1) ∧ (is-Killing-BEGINNode n2)) ∨
  ((is-LoadFieldNode n1) ∧ (is-LoadFieldNode n2)) ∨
  ((is-LogicNegationNode n1) ∧ (is-LogicNegationNode n2)) ∨
  ((is-Loop-BEGINNode n1) ∧ (is-Loop-BEGINNode n2)) ∨
  ((is-Loop-ENDNode n1) ∧ (is-Loop-ENDNode n2)) ∨
  ((is-Loop-ExitNode n1) ∧ (is-Loop-ExitNode n2)) ∨
  ((is-MergeNode n1) ∧ (is-MergeNode n2)) ∨
  ((is-MethodCallTargetNode n1) ∧ (is-MethodCallTargetNode n2)) ∨
  ((is-MulNode n1) ∧ (is-MulNode n2)) ∨
  ((is-NegateNode n1) ∧ (is-NegateNode n2)) ∨
  ((is-NewArrayNode n1) ∧ (is-NewArrayNode n2)) ∨
  ((is-NewInstanceNode n1) ∧ (is-NewInstanceNode n2)) ∨
  ((is-NotNode n1) ∧ (is-NotNode n2)) ∨
  ((is-OrNode n1) ∧ (is-OrNode n2)) ∨
  ((is-ParameterNode n1) ∧ (is-ParameterNode n2)) ∨
  ((is-PiNode n1) ∧ (is-PiNode n2)) ∨
  ((is-ReturnNode n1) ∧ (is-ReturnNode n2)) ∨
  ((is-ShortCircuitOrNode n1) ∧ (is-ShortCircuitOrNode n2)) ∨
  ((is-SignedDivNode n1) ∧ (is-SignedDivNode n2)) ∨
  ((is-StartNode n1) ∧ (is-StartNode n2)) ∨
  ((is-StoreFieldNode n1) ∧ (is-StoreFieldNode n2)) ∨
  ((is-SubNode n1) ∧ (is-SubNode n2)) ∨

```

```

((is-UnwindNode n1) ∧ (is-UnwindNode n2)) ∨
((is-ValuePhiNode n1) ∧ (is-ValuePhiNode n2)) ∨
((is-ValueProxyNode n1) ∧ (is-ValueProxyNode n2)) ∨
((is-XorNode n1) ∧ (is-XorNode n2)))

```

end

3 Stamp Typing

```

theory Stamp2
  imports Values2
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) (stp-lower: int) (stp-upper: int)

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

```

```

fun bit-bounds :: nat ⇒ (int × int) where
  bit-bounds bits = (((2 ^ bits) div 2) * -1, ((2 ^ bits) div 2) - 1)

```

— A stamp which includes the full range of the type

```

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

```

```

  unrestricted-stamp (KlassPointerStamp nonNull alwaysNull) = (KlassPointerStamp
False False) |
  unrestricted-stamp (MethodCountersPointerStamp nonNull alwaysNull) = (MethodCountersPointerStamp
False False) |

```

```

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

```

```

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

```

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

```

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

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

```

```

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

    is-stamp-empty x = False

```

— Calculate the meet stamp of two stamps

```

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

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

```

— Calculate the join stamp of two stamps

```
fun join :: Stamp ⇒ Stamp ⇒ Stamp where
  join VoidStamp VoidStamp = VoidStamp |
  join (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1 ≠ b2 then IllegalStamp else
    (IntegerStamp b1 (max l1 l2) (min u1 u2))
  ) |
  join (KlassPointerStamp nn1 an1) (KlassPointerStamp nn2 an2) = (
    if ((nn1 ∨ nn2) ∧ (an1 ∨ an2))
    then (empty-stamp (KlassPointerStamp nn1 an1))
    else (KlassPointerStamp (nn1 ∨ nn2) (an1 ∨ an2))
  ) |
  join (MethodCountersPointerStamp nn1 an1) (MethodCountersPointerStamp nn2
an2) = (
    if ((nn1 ∨ nn2) ∧ (an1 ∨ an2))
    then (empty-stamp (MethodCountersPointerStamp nn1 an1))
    else (MethodCountersPointerStamp (nn1 ∨ nn2) (an1 ∨ an2))
  ) |
  join (MethodPointersStamp nn1 an1) (MethodPointersStamp nn2 an2) = (
    if ((nn1 ∨ nn2) ∧ (an1 ∨ an2))
    then (empty-stamp (MethodPointersStamp nn1 an1))
    else (MethodPointersStamp (nn1 ∨ nn2) (an1 ∨ an2))
  ) |
  join s1 s2 = IllegalStamp
```

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

```
fun asConstant :: Stamp ⇒ Value where
  asConstant (IntegerStamp b l h) = (if l = h then IntVal64 (word-of-int l) else
UndefVal) |
  asConstant - = UndefVal
```

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

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

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

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

```
fun constantAsStamp :: Value ⇒ Stamp where
```

```
  constantAsStamp (IntVal32 v) = (IntegerStamp (nat 32) (sint v) (sint v)) |
  constantAsStamp (IntVal64 v) = (IntegerStamp (nat 64) (sint v) (sint v)) |
```


constantAsStamp - = *IllegalStamp*

— Define when a runtime value is valid for a stamp

```
fun valid-value :: Stamp  $\Rightarrow$  Value  $\Rightarrow$  bool where
  valid-value (IntegerStamp b l h) (IntVal32 v) = (b=32  $\wedge$  (sint v  $\geq$  l)  $\wedge$  (sint v  $\leq$ 
h)) |
  valid-value (IntegerStamp b l h) (IntVal64 v) = (b=64  $\wedge$  (sint v  $\geq$  l)  $\wedge$  (sint v  $\leq$ 
h)) |

  valid-value (VoidStamp) (UndefVal) = True |
  valid-value (ObjectStamp klass exact nonNull alwaysNull) (ObjRef ref) =
    (if nonNull then ref $\neq$ None else True) |
  valid-value stamp val = 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))
```

end

4 Graph Representation

```
theory IRGraph
imports
  IRNodeHierarchy
  Stamp2
  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 = (λm 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 λnid 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 λnid g. g(nid := None) **by** simp

lift-definition *replace-node* :: ID \Rightarrow (IRNode \times Stamp) \Rightarrow IRGraph \Rightarrow IRGraph
is λnid 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 λg . map (λk . (k, the (g k))) (sorted-list-of-set (dom g)) .

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

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

code-datatype *irgraph*

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

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

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

lemma *fil-eq*:
 $filter-none\ (map-of\ (no-node\ xs)) = set\ (map\ fst\ (no-node\ xs))$
using *no-node-clears*

by (metis dom-eq dom-map-of-conv-image-fst list.set-map)

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

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

— Get the inputs set of a given node ID
fun *inputs* :: *IRGraph* \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* = $\{j. j \in \text{ids } g \wedge (j, \text{nid}) \in \text{input-edges } g\}$

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* = $\{j. j \in \text{ids } g \wedge (j, \text{nid}) \in \text{successor-edges } g\}$

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 kind g x ≠ NoNode  $\longrightarrow$  x ∈ ids g
    unfolding with-default.simps kind-def ids-def
    by (cases Rep-IRGraph g x = None; auto)
  from this that show ?thesis by auto
qed

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma [simp]: finite (dom g) ⟶ ids (Abs-IRGraph g) = {nid ∈ dom g . ∄ s. g
nid = Some (NoNode, s)}
  using ids.rep-eq by simp

```

```

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

```

```

lemma [simp]: finite (dom g) ⟶ stamp (Abs-IRGraph g) = (λx . (case g x of
None ⇒ IllegalStamp | Some n ⇒ snd n))
  using stamp.abs-eq stamp.rep-eq by auto

```

```

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

```

```

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

```

```

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

```

```

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

```

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

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

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

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

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

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

4.0.1 Example Graphs

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

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

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

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

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

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

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

value *input-edges* *eg2-sq*

value *usages* *eg2-sq* 1

end

5 Data-flow Semantics

theory *IRTreeEval*

imports

Graph.IRGraph

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 *MapState* = *ID* \Rightarrow *Value*
type-synonym *Params* = *Value list*

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

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

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

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 nid* =
 (filter (λx . (*is-PhiNode* (*kind g x*)))
 (*sorted-list-of-set* (*usages g nid*)))

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* (λn . (*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 (*nid # xs*) (*v # vs*) *m* = (*set-phis xs vs* (*m*(*nid* := *v*))) |
set-phis [] (*v # vs*) *m* = *m* |
set-phis (*x # xs*) [] *m* = *m*

fun *find-node-and-stamp* :: *IRGraph* \Rightarrow (*IRNode* \times *Stamp*) \Rightarrow *ID option* **where**
find-node-and-stamp *g* (*n,s*) =
find (λi . *kind g i* = *n* \wedge *stamp g i* = *s*) (*sorted-list-of-set*(*ids g*))

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

5.1 Data-flow Tree Representation

datatype *IRUnaryOp* =

UnaryAbs
 | *UnaryNeg*
 | *UnaryNot*
 | *UnaryLogicNegation*

datatype *IRBinaryOp* =

BinAdd
 | *BinMul*
 | *BinSub*
 | *BinAnd*
 | *BinOr*
 | *BinXor*
 | *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)
 | *ConstantExpr* (*ir-const*: *Value*)

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

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

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

is-preevaluated (*InvokeNode* *nid* - - - -) = *True* |
is-preevaluated (*InvokeWithExceptionNode* *nid* - - - - -) = *True* |
is-preevaluated (*NewInstanceNode* *nid* - - -) = *True* |
is-preevaluated (*LoadFieldNode* *nid* - - -) = *True* |
is-preevaluated (*SignedDivNode* *nid* - - - - -) = *True* |
is-preevaluated (*SignedRemNode* *nid* - - - - -) = *True* |
is-preevaluated (*ValuePhiNode* *nid* - -) = *True* |
is-preevaluated - = *False*

inductive

rep :: *IRGraph* \Rightarrow *ID* \Rightarrow *IRExpr* \Rightarrow *bool* (- \vdash - \triangleright - 55)
for *g* **where**

ConstantNode:

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

ParameterNode:

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

ConditionalNode:

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

AbsNode:

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

NotNode:

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

NegateNode:

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

LogicNegationNode:

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

AddNode:

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

MulNode:

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

SubNode:

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

AndNode:

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

OrNode:

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

XorNode:

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

IntegerBelowNode:

$\llbracket \text{kind } g \ n = \text{IntegerBelowNode } x \ y;$
 $g \vdash x \triangleright xe;$
 $g \vdash y \triangleright ye \rrbracket$
 $\implies g \vdash n \triangleright (\text{BinaryExpr } \text{BinIntegerBelow } xe \ ye) \mid$

IntegerEqualsNode:

$\llbracket \text{kind } g \ n = \text{IntegerEqualsNode } x \ y;$
 $g \vdash x \triangleright xe;$
 $g \vdash y \triangleright ye \rrbracket$
 $\implies g \vdash n \triangleright (\text{BinaryExpr } \text{BinIntegerEquals } xe \ ye) \mid$

IntegerLessThanNode:

$\llbracket \text{kind } g \ n = \text{IntegerLessThanNode } x \ y;$
 $g \vdash x \triangleright xe;$
 $g \vdash y \triangleright ye \rrbracket$
 $\implies g \vdash n \triangleright (\text{BinaryExpr } \text{BinIntegerLessThan } xe \ ye) \mid$

LeafNode:

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

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 *-* \triangleright_L - 55)
for *g* **where**

RepNil:

$g \vdash [] \triangleright_L [] \mid$

RepCons:

$\llbracket g \vdash x \triangleright xe;$
 $g \vdash xs \triangleright_L xse \rrbracket$
 $\implies g \vdash x \# xs \triangleright_L xe \# xse$

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

$$\frac{\text{kind } g \ n = \text{ConstantNode } c}{g \vdash n \triangleright \text{ConstantExpr } c}$$

$$\frac{\text{kind } g \ n = \text{ParameterNode } i \quad \text{stamp } g \ n = s}{g \vdash n \triangleright \text{ParameterExpr } i \ s}$$

$$\frac{\text{kind } g \ n = \text{AbsNode } x \quad g \vdash x \triangleright xe}{g \vdash n \triangleright \text{UnaryExpr } \text{UnaryAbs } xe}$$

$$\frac{\text{kind } g \ n = \text{AddNode } x \ y \quad g \vdash x \triangleright xe \quad g \vdash y \triangleright ye}{g \vdash n \triangleright \text{BinaryExpr } \text{BinAdd } xe \ ye}$$

$$\frac{\text{kind } g \ n = \text{MulNode } x \ y \quad g \vdash x \triangleright xe \quad g \vdash y \triangleright ye}{g \vdash n \triangleright \text{BinaryExpr } \text{BinMul } xe \ ye}$$

$$\frac{\text{kind } g \ n = \text{SubNode } x \ y \quad g \vdash x \triangleright xe \quad g \vdash y \triangleright ye}{g \vdash n \triangleright \text{BinaryExpr } \text{BinSub } xe \ ye}$$

$$\frac{\text{is-preevaluated } (\text{kind } g \ n) \quad \text{stamp } g \ n = s}{g \vdash n \triangleright \text{LeafExpr } n \ s}$$

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

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

stamp-unary op (*IntegerStamp* *b lo hi*) = *unrestricted-stamp* (*IntegerStamp* *b 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 (b1 = b2) then unrestricted-stamp (IntegerStamp b1 lo1 hi1) else IllegalStamp)
|

stamp-binary op - - = IllegalStamp

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

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

fun unary-node :: IRUnaryOp ⇒ ID ⇒ 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

fun bin-node :: IRBinaryOp ⇒ ID ⇒ ID ⇒ IRNode where
  bin-node BinAdd x y = AddNode x y |
  bin-node BinMul x y = MulNode x y |
  bin-node BinSub x y = SubNode x y |
  bin-node BinAnd x y = AndNode x y |
  bin-node BinOr x y = OrNode x y |
  bin-node BinXor x y = XorNode x y |
  bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
  bin-node BinIntegerLessThan x y = IntegerLessThanNode x y |
  bin-node BinIntegerBelow x y = IntegerBelowNode x y

fun unary-eval :: IRUnaryOp ⇒ Value ⇒ Value where
  unary-eval UnaryAbs v = intval-abs v |
  unary-eval UnaryNeg v = intval-negate v |
  unary-eval UnaryNot v = intval-not v |
  unary-eval UnaryLogicNegation (IntVal32 v1) = (if v1 = 0 then (IntVal32 1) else
(IntVal32 0)) |
  unary-eval op v1 = UndefVal

fun bin-eval :: IRBinaryOp ⇒ Value ⇒ Value ⇒ Value where
  bin-eval BinAdd v1 v2 = intval-add v1 v2 |
  bin-eval BinMul v1 v2 = intval-mul v1 v2 |

```

```

bin-eval BinSub v1 v2 = intval-sub v1 v2 |
bin-eval BinAnd v1 v2 = intval-and v1 v2 |
bin-eval BinOr v1 v2 = intval-or v1 v2 |
bin-eval BinXor v1 v2 = intval-xor v1 v2 |
bin-eval BinIntegerEquals v1 v2 = intval-equals v1 v2 |
bin-eval BinIntegerLessThan v1 v2 = intval-less-than v1 v2 |
bin-eval BinIntegerBelow v1 v2 = intval-below v1 v2

```

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

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* (- \triangleleft - \rightsquigarrow - 55)

and

unrepList :: *IRGraph* \Rightarrow *IRExpr* *list* \Rightarrow (*IRGraph* \times *ID* *list*) \Rightarrow *bool* (- \triangleleft_L - \rightsquigarrow - 55)

where

ConstantNodeSame:

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

ConstantNodeNew:

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

ParameterNodeSame:

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

ParameterNodeNew:

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

$g' = \text{add-node } \text{nid} \ (\text{ParameterNode } i, s) \ g\rrbracket$
 $\implies g \triangleleft (\text{ParameterExpr } i \ s) \rightsquigarrow (g', \text{nid}) \mid$

ConditionalNodeSame:

$\llbracket g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f]);$
 $s' = \text{meet } (\text{stamp } g2 \ t) \ (\text{stamp } g2 \ f);$
 $\text{find-node-and-stamp } g2 \ (\text{ConditionalNode } c \ t \ f, s') = \text{Some } \text{nid}\rrbracket$
 $\implies g \triangleleft (\text{ConditionalExpr } ce \ te \ fe) \rightsquigarrow (g2, \text{nid}) \mid$

ConditionalNodeNew:

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

UnaryNodeSame:

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

UnaryNodeNew:

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

BinaryNodeSame:

$\llbracket g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]);$
 $s' = \text{stamp-binary } op \ (\text{stamp } g2 \ x) \ (\text{stamp } g2 \ y);$
 $\text{find-node-and-stamp } g2 \ (\text{bin-node } op \ x \ y, s') = \text{Some } \text{nid}\rrbracket$
 $\implies g \triangleleft (\text{BinaryExpr } op \ xe \ ye) \rightsquigarrow (g2, \text{nid}) \mid$

BinaryNodeNew:

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

AllLeafNodes:

$\text{stamp } g \ \text{nid} = s$
 $\implies g \triangleleft (\text{LeafExpr } \text{nid} \ s) \rightsquigarrow (g, \text{nid}) \mid$

UnrepNil:
 $g \triangleleft_L [] \rightsquigarrow (g, []) \mid$

UnrepCons:
 $\llbracket g \triangleleft xe \rightsquigarrow (g2, x);$
 $g2 \triangleleft_L xes \rightsquigarrow (g3, xs) \rrbracket$
 $\implies g \triangleleft_L (xe \# xes) \rightsquigarrow (g3, x \# xs)$

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

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

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

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

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

$$\frac{\begin{array}{c} g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f]) \quad s' = \text{meet } (\text{stamp } g2 \ t) \ (\text{stamp } g2 \ f) \\ \text{find-node-and-stamp } g2 \text{ (ConditionalNode } c \ t \ f, s') = \text{Some } nid \end{array}}{g \triangleleft \text{ConditionalExpr } ce \ te \ fe \rightsquigarrow (g2, nid)}$$

$$\frac{\begin{array}{c} g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f]) \quad s' = \text{meet } (\text{stamp } g2 \ t) \ (\text{stamp } g2 \ f) \\ \text{find-node-and-stamp } g2 \text{ (ConditionalNode } c \ t \ f, s') = \text{None} \\ nid = \text{get-fresh-id } g2 \quad g' = \text{add-node } nid \text{ (ConditionalNode } c \ t \ f, s') \end{array}}{g \triangleleft \text{ConditionalExpr } ce \ te \ fe \rightsquigarrow (g', nid)}$$

$$\frac{\begin{array}{c} g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]) \quad s' = \text{stamp-binary } op \ (\text{stamp } g2 \ x) \ (\text{stamp } g2 \ y) \\ \text{find-node-and-stamp } g2 \text{ (bin-node } op \ x \ y, s') = \text{Some } nid \end{array}}{g \triangleleft \text{BinaryExpr } op \ xe \ ye \rightsquigarrow (g2, nid)}$$

$$\frac{\begin{array}{c} g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]) \quad s' = \text{stamp-binary } op \ (\text{stamp } g2 \ x) \ (\text{stamp } g2 \ y) \\ \text{find-node-and-stamp } g2 \text{ (bin-node } op \ x \ y, s') = \text{None} \\ nid = \text{get-fresh-id } g2 \quad g' = \text{add-node } nid \text{ (bin-node } op \ x \ y, s') \end{array}}{g \triangleleft \text{BinaryExpr } op \ xe \ ye \rightsquigarrow (g', nid)}$$

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

definition *sq-param0* :: *IRExpr* **where**

sq-param0 = *BinaryExpr BinMul*
(ParameterExpr 0 (IntegerStamp 32 (- 2147483648) 2147483647))
(ParameterExpr 0 (IntegerStamp 32 (- 2147483648) 2147483647))

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

5.2 Data-flow Tree Evaluation

inductive

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

ConstantExpr:

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

ParameterExpr:

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

ConditionalExpr:

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

UnaryExpr:

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

BinaryExpr:

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

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

$$\begin{array}{c}
\frac{c \neq \text{UndefVal}}{[m,p] \vdash \text{ConstantExpr } c \mapsto c} \\
\\
\frac{\text{valid-value } s \text{ } p_{[i]}}{[m,p] \vdash \text{ParameterExpr } i \text{ } s \mapsto p_{[i]}} \\
\\
\frac{[m,p] \vdash ce \mapsto \text{cond} \quad \text{branch} = (\text{if } \text{IRTreeEval.val-to-bool cond then te else fe}) \quad [m,p] \vdash \text{branch} \mapsto v}{[m,p] \vdash \text{ConditionalExpr ce te fe} \mapsto v} \\
\\
\frac{[m,p] \vdash xe \mapsto v}{[m,p] \vdash \text{UnaryExpr op xe} \mapsto \text{unary-eval op } v} \\
\\
\frac{[m,p] \vdash xe \mapsto x \quad [m,p] \vdash ye \mapsto y}{[m,p] \vdash \text{BinaryExpr op xe ye} \mapsto \text{bin-eval op } x \text{ } y} \\
\\
\frac{\text{val} = m \text{ nid} \quad \text{valid-value } s \text{ val}}{[m,p] \vdash \text{LeafExpr nid } s \mapsto val}
\end{array}$$

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

inductive

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

for *m p* **where**

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

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

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

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

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

5.3 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**

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

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

end

6 Data-flow Expression-Tree Theorems

theory *IRTreeEvalThms*

```

imports
  Semantics.IRTreeEval
begin

```

6.1 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.

lemma *rep-constant*:

```

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

```

lemma *rep-parameter*:

```

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

```

lemma *rep-conditional*:

```

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

```

lemma *rep-abs*:

```

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

```

lemma *rep-not*:

```

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

```

lemma *rep-negate*:

```

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

```

lemma *rep-logicnegation*:

```

 $g \vdash n \triangleright e \implies$ 

```

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

lemma rep-add:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = AddNode\ x\ y \implies$
 $(\exists xe\ ye. e = BinaryExpr\ BinAdd\ xe\ ye)$
by (induction rule: rep.induct; auto)

lemma rep-sub:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = SubNode\ x\ y \implies$
 $(\exists xe\ ye. e = BinaryExpr\ BinSub\ xe\ ye)$
by (induction rule: rep.induct; auto)

lemma rep-mul:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = MulNode\ x\ y \implies$
 $(\exists xe\ ye. e = BinaryExpr\ BinMul\ xe\ ye)$
by (induction rule: rep.induct; auto)

lemma rep-and:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = AndNode\ x\ y \implies$
 $(\exists xe\ ye. e = BinaryExpr\ BinAnd\ xe\ ye)$
by (induction rule: rep.induct; auto)

lemma rep-or:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = OrNode\ x\ y \implies$
 $(\exists xe\ ye. e = BinaryExpr\ BinOr\ xe\ ye)$
by (induction rule: rep.induct; auto)

lemma rep-xor:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = XorNode\ x\ y \implies$
 $(\exists xe\ ye. e = BinaryExpr\ BinXor\ xe\ ye)$
by (induction rule: rep.induct; auto)

lemma rep-integer-below:

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

lemma rep-integer-equals:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = IntegerEqualsNode\ x\ y \implies$

($\exists xe ye. e = \text{BinaryExpr BinIntegerEquals } xe ye$)
by (*induction rule: rep.induct; auto*)

lemma *rep-integer-less-than*:

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

lemma *rep-load-field*:

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

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

lemma *repDet*:

shows ($g \vdash n \triangleright e1 \implies g \vdash n \triangleright e2 \implies e1 = e2$)
proof (*induction arbitrary: e2 rule: rep.induct*)
case (*ConstantNode* $n \ c$)
then show $?case$ **using** *rep-constant* **by** *auto*
next
case (*ParameterNode* $n \ i \ s$)
then show $?case$ **using** *rep-parameter* **by** *auto*
next
case (*ConditionalNode* $n \ c \ t \ f \ ce \ te \ fe$)
then show $?case$
by (*metis rep-conditional ConditionalNodeE IRNode.inject(6)*)
next
case (*AbsNode* $n \ x \ xe$)
then show $?case$
by (*metis rep-abs AbsNodeE IRNode.inject(1)*)
next
case (*NotNode* $n \ x \ xe$)
then show $?case$
by (*metis IRNode.inject(30) NotNodeE rep-not*)
next
case (*NegateNode* $n \ x \ xe$)
then show $?case$
by (*metis IRNode.inject(27) NegateNodeE rep-negate*)
next
case (*LogicNegationNode* $n \ x \ xe$)
then show $?case$
by (*metis IRNode.inject(20) LogicNegationNodeE rep-logicnegation*)
next
case (*AddNode* $n \ x \ y \ xe \ ye$)
then show $?case$
by (*metis AddNodeE IRNode.inject(2) rep-add*)

```

next
case (MulNode n x y xe ye)
  then show ?case
  by (metis IRNode.inject(26) MulNodeE rep-mul)
next
case (SubNode n x y xe ye)
  then show ?case
  by (metis IRNode.inject(40) SubNodeE rep-sub)
next
case (AndNode n x y xe ye)
  then show ?case
  by (metis AndNodeE IRNode.inject(3) rep-and)
next
case (OrNode n x y xe ye)
  then show ?case
  by (metis IRNode.inject(31) OrNodeE rep-or)
next
case (XorNode n x y xe ye)
  then show ?case
  by (metis IRNode.inject(44) XorNodeE rep-xor)
next
case (IntegerBelowNode n x y xe ye)
  then show ?case
  by (metis IRNode.inject(12) IntegerBelowNodeE rep-integer-below)
next
case (IntegerEqualsNode n x y xe ye)
  then show ?case
  by (metis IRNode.inject(13) IntegerEqualsNodeE rep-integer-equals)
next
case (IntegerLessThanNode n x y xe ye)
  then show ?case
  by (metis IRNode.inject(14) IntegerLessThanNodeE rep-integer-less-than)
next
case (LeafNode n s)
  then show ?case using rep-load-field LeafNodeE by blast
qed

```

lemma *repAllDet*:

$$\begin{aligned}
&g \vdash xs \triangleright_L e1 \implies \\
&g \vdash xs \triangleright_L e2 \implies \\
&e1 = e2
\end{aligned}$$

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 *evalDet*:

$[m, p] \vdash e \mapsto v1 \implies$

$[m, p] \vdash e \mapsto v2 \implies$

$v1 = v2$

apply (*induction arbitrary*; *v2 rule: evaltree.induct*)

by (*elim EvalTreeE*; *auto*)⁺

lemma *evalAllDet*:

$[m, p] \vdash e \mapsto_L v1 \implies$

$[m, p] \vdash e \mapsto_L v2 \implies$

$v1 = v2$

apply (*induction arbitrary*; *v2 rule: evaltrees.induct*)

apply (*elim EvalTreeE*; *auto*)

using *evalDet* **by** *force*

A valid value cannot be *UndefVal*.

lemma *valid-not-undef*:

assumes *a1*: *valid-value s val*

assumes *a2*: $s \neq \text{VoidStamp}$

shows $val \neq \text{UndefVal}$

apply (*rule valid-value.elims*(1)[*of s val True*])

using *a1 a2* **by** *auto*

lemma *valid-VoidStamp*[*elim*]:

shows *valid-value VoidStamp val* \implies

$val = \text{UndefVal}$

using *valid-value.simps* **by** (*metis IRTreeEval.val-to-bool.cases*)

lemma *valid-ObjStamp*[*elim*]:

shows *valid-value (ObjectStamp klass exact nonNull alwaysNull) val* \implies

$(\exists v. val = \text{ObjRef } v)$

using *valid-value.simps* **by** (*metis IRTreeEval.val-to-bool.cases*)

lemma *valid-int32*[*elim*]:

shows *valid-value (IntegerStamp 32 l h) val* \implies

$(\exists v. val = \text{IntVal32 } v)$

apply (*rule IRTreeEval.val-to-bool.cases*[*of val*])

using *Value.distinct* **by** *simp*⁺

lemma *valid-int64*[*elim*]:

shows *valid-value (IntegerStamp 64 l h) val* \implies

$(\exists v. val = \text{IntVal64 } v)$

apply (*rule IRTreeEval.val-to-bool.cases*[*of val*])

using *Value.distinct* **by** *simp*⁺

TODO: could we prove that expression evaluation never returns *UndefVal*?

But this might require restricting unary and binary operators to be total...

lemma *leafint32*:

assumes *ev*: $[m,p] \vdash \text{LeafExpr } i \text{ (IntegerStamp 32 lo hi)} \mapsto \text{val}$
shows $\exists v. \text{val} = (\text{IntVal32 } v)$

proof –

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

qed

lemma *leafint64*:

assumes *ev*: $[m,p] \vdash \text{LeafExpr } i \text{ (IntegerStamp 64 lo hi)} \mapsto \text{val}$
shows $\exists v. \text{val} = (\text{IntVal64 } v)$

proof –

have *valid-value* (IntegerStamp 64 lo hi) *val*
using *ev* **by** (rule *LeafExprE*; *simp*)
then show ?thesis **by** *auto*

qed

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

using *default-stamp-def* **by** *auto*

lemma *valid32* [*simp*]:

assumes *valid-value* (IntegerStamp 32 lo hi) *val*
shows $\exists v. (\text{val} = (\text{IntVal32 } v) \wedge \text{lo} \leq \text{sint } v \wedge \text{sint } v \leq \text{hi})$
using *assms valid-int32* **by** *force*

lemma *valid64* [*simp*]:

assumes *valid-value* (IntegerStamp 64 lo hi) *val*
shows $\exists v. (\text{val} = (\text{IntVal64 } v) \wedge \text{lo} \leq \text{sint } v \wedge \text{sint } v \leq \text{hi})$
using *assms valid-int64* **by** *force*

lemma *int-stamp-implies-valid-value*:

$[m,p] \vdash \text{expr} \mapsto \text{val} \implies$
valid-value (stamp-expr *expr*) *val*

proof (*induction rule: evaltree.induct*)

case (*ConstantExpr c*)

then show ?case **sorry**

next

case (*ParameterExpr s i*)

then show ?case **sorry**

next

case (*ConditionalExpr ce cond branch te fe v*)

then show ?case **sorry**

next


```

    case (UnaryExpr xe v op)
    then show ?case sorry
next
    case (BinaryExpr xe x ye y op)
    then show ?case sorry
next
    case (LeafExpr val nid s)
    then show ?case sorry
qed

```

```

lemma valid32or64:
  assumes valid-value (IntegerStamp b lo hi) x
  shows ( $\exists v1. (x = \text{IntVal32 } v1)$ )  $\vee$  ( $\exists v2. (x = \text{IntVal64 } v2)$ )
  using valid32 valid64 assms valid-value.elims(2) by blast

```

```

lemma valid32or64-both:
  assumes valid-value (IntegerStamp b lox hix) x
  and valid-value (IntegerStamp b loy hiy) y
  shows ( $\exists v1 v2. x = \text{IntVal32 } v1 \wedge y = \text{IntVal32 } v2$ )  $\vee$  ( $\exists v3 v4. x = \text{IntVal64 } v3 \wedge y = \text{IntVal64 } v4$ )
  using assms valid32or64 valid32 valid-value.elims(2) valid-value.simps(1) by metis

```

6.2 Example Data-flow Optimisations

```

lemma a0a-helper [simp]:
  assumes a: valid-value (IntegerStamp 32 lo hi) v
  shows intval-add v (IntVal32 0) = v
proof -
  obtain v32 :: int32 where v = (IntVal32 v32) using a valid32 by blast
  then show ?thesis by simp
qed

```

```

lemma a0a: (BinaryExpr BinAdd (LeafExpr 1 default-stamp) (ConstantExpr (IntVal32 0)))
   $\leq$  (LeafExpr 1 default-stamp) (is ?L  $\leq$  ?R)
  by (auto simp add: evaltree.LeafExpr)

```

```

lemma xyx-y-helper [simp]:
  assumes valid-value (IntegerStamp 32 lox hix) x
  assumes valid-value (IntegerStamp 32 loy hiy) y
  shows intval-add x (intval-sub y x) = y
proof -
  obtain x32 :: int32 where x = (IntVal32 x32) using assms valid32 by blast
  obtain y32 :: int32 where y = (IntVal32 y32) using assms valid32 by blast
  show ?thesis using x y by simp
qed

```

```

lemma xyx-y:
  (BinaryExpr BinAdd
    (LeafExpr x (IntegerStamp 32 lox hix))
    (BinaryExpr BinSub
      (LeafExpr y (IntegerStamp 32 loy hiy))
      (LeafExpr x (IntegerStamp 32 lox hix))))
  ≤ (LeafExpr y (IntegerStamp 32 loy hiy))
by (auto simp add: LeafExpr)

```

6.3 Monotonicity of Expression Optimization

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

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

```

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

```

```

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

```

```

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

```

end

7 Control-flow Semantics

```
theory IRStepObj
  imports
    IRTreeEval
begin
```

7.1 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'.

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

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

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

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

type-synonym FieldRefHeap = (string, objref) DynamicHeap

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

7.2 Intraprocedural Semantics

Intraprocedural semantics are given as a small-step semantics. Within the context of a graph, the configuration triple, (ID, MethodState, Heap), is related to the subsequent configuration.

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

SequentialNode:

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

IfNode:

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

EndNodes:

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

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

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

NewInstanceNode:

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

LoadFieldNode:

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

SignedDivNode:

$\llbracket kind\ g\ nid = (SignedDivNode\ nid\ x\ y\ zero\ sb\ nxt);$
 $g \vdash x \triangleright xe;$
 $g \vdash y \triangleright ye;$
 $[m, p] \vdash xe \mapsto v1;$
 $[m, p] \vdash ye \mapsto v2;$
 $v = (intval\text{-div}\ v1\ v2);$

$$m' = m(nid := v) \\ \Longrightarrow g, p \vdash (nid, m, h) \rightarrow (nxt, m', h) \mid$$

SignedRemNode:

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

StaticLoadFieldNode:

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

StoreFieldNode:

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

StaticStoreFieldNode:

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

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

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

InvokeNodeStep:

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

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

ReturnNode:

$$\begin{aligned} & \llbracket kind\ g\ nid = (ReturnNode\ (Some\ expr)\ -) \rrbracket \\ & g \vdash expr \triangleright e; \\ & [m, p] \vdash e \mapsto v; \end{aligned}$$

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

ReturnNodeVoid:

$$\begin{aligned} & \llbracket kind\ g\ nid = (ReturnNode\ None\ -) \rrbracket \\ & cm' = cm(cnid := (ObjRef (Some (2048)))); \end{aligned}$$

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

UnwindNode:

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

$$\begin{aligned} & g \vdash exception \triangleright exceptionE; \\ & [m, p] \vdash exceptionE \mapsto e; \end{aligned}$$

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

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

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

7.4 Big-step Execution

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

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

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

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

7.4.1 Heap Testing

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

$p3 = [IntVal32\ 3]$

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

definition $field\text{-}sq :: string$ **where**

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

definition $eg3\text{-}sq :: IRGraph$ **where**

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

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

definition $eg4\text{-}sq :: IRGraph$ **where**

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

values $\{h\text{-}load\text{-}field\ field\text{-}sq\ (Some\ 0)\ (prod.snd\ res) \mid res.$
 $(\lambda x. Some\ eg4\text{-}sq) \vdash [(eg4\text{-}sq, 0, new\text{-}map\text{-}state, p3), (eg4\text{-}sq, 0,$
 $new\text{-}map\text{-}state, p3)], new\text{-}heap) \rightarrow^* 4^* res\}$

end

8 Canonicalization Phase

theory $CanonicalizationTree$

imports

Semantics.IRTreeEval
begin

fun *is-neutral* :: *IRBinaryOp* \Rightarrow *Value* \Rightarrow *bool* **where**

is-neutral *BinMul* (*IntVal32* *x*) = (*x* = 1) |
is-neutral *BinMul* (*IntVal64* *x*) = (*x* = 1) |

is-neutral *BinAdd* (*IntVal32* *x*) = (*x* = 0) |
is-neutral *BinAdd* (*IntVal64* *x*) = (*x* = 0) |

is-neutral *BinXor* (*IntVal32* *x*) = (*x* = 0) |
is-neutral *BinXor* (*IntVal64* *x*) = (*x* = 0) |

is-neutral *BinSub* (*IntVal32* *x*) = (*x* = 0) |
is-neutral *BinSub* (*IntVal64* *x*) = (*x* = 0) |

is-neutral - - = *False*

fun *is-zero* :: *IRBinaryOp* \Rightarrow *Value* \Rightarrow *bool* **where**

is-zero *BinMul* (*IntVal32* *x*) = (*x* = 0) |
is-zero *BinMul* (*IntVal64* *x*) = (*x* = 0) |

is-zero - - = *False*

fun *int-to-value* :: *Value* \Rightarrow *int* \Rightarrow *Value* **where**

int-to-value (*IntVal32* -) *y* = (*IntVal32* (*word-of-int* *y*)) |
int-to-value (*IntVal64* -) *y* = (*IntVal64* (*word-of-int* *y*)) |

int-to-value - - = *UndefVal*

inductive *CanonicalizeBinaryOp* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**

binary-const-fold:

$\llbracket x = (\text{ConstantExpr } \text{val1});$
 $y = (\text{ConstantExpr } \text{val2});$
 $\text{val} = \text{bin-eval } \text{op } \text{val1 } \text{val2};$
 $\text{val} \neq \text{UndefVal} \rrbracket$
 $\implies \text{CanonicalizeBinaryOp } (\text{BinaryExpr } \text{op } x \ y) (\text{ConstantExpr } \text{val}) \mid$

binary-fold-yneutral:

$\llbracket y = (\text{ConstantExpr } c);$
 $\text{is-neutral } \text{op } c;$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{stp-bits } \text{stampx} = \text{stp-bits } \text{stampy};$
 $\text{is-IntegerStamp } \text{stampx} \wedge \text{is-IntegerStamp } \text{stampy} \rrbracket$

$\implies \text{CanonicalizeBinaryOp } (\text{BinaryExpr op } x \ y) \ x \mid$

binary-fold-yzero32:

$\llbracket y = \text{ConstantExpr } c;$
 $\text{is-zero op } c;$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{stp-bits stampx} = \text{stp-bits stampy};$
 $\text{stp-bits stampx} = 32;$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy} \rrbracket$
 $\implies \text{CanonicalizeBinaryOp } (\text{BinaryExpr op } x \ y) \ (\text{ConstantExpr } (\text{IntVal32 } 0)) \mid$

binary-fold-yzero64:

$\llbracket y = \text{ConstantExpr } c;$
 $\text{is-zero op } c;$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{stp-bits stampx} = \text{stp-bits stampy};$
 $\text{stp-bits stampx} = 64;$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy} \rrbracket$
 $\implies \text{CanonicalizeBinaryOp } (\text{BinaryExpr op } x \ y) \ (\text{ConstantExpr } (\text{IntVal64 } 0))$

inductive *CanonicalizeUnaryOp* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**

unary-const-fold:

$\llbracket \text{val}' = \text{unary-eval op val};$
 $\text{val}' \neq \text{UndefVal} \rrbracket$
 $\implies \text{CanonicalizeUnaryOp } (\text{UnaryExpr op } (\text{ConstantExpr val})) \ (\text{ConstantExpr val'})$

inductive *CanonicalizeMul* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**

mul-negate32:

$\llbracket y = \text{ConstantExpr } (\text{IntVal32 } (-1));$
 $\text{stamp-expr } x = \text{IntegerStamp } 32 \text{ lo hi} \rrbracket$
 $\implies \text{CanonicalizeMul } (\text{BinaryExpr BinMul } x \ y) \ (\text{UnaryExpr UnaryNeg } x) \mid$

mul-negate64:

$\llbracket y = \text{ConstantExpr } (\text{IntVal64 } (-1));$
 $\text{stamp-expr } x = \text{IntegerStamp } 64 \text{ lo hi} \rrbracket$
 $\implies \text{CanonicalizeMul } (\text{BinaryExpr BinMul } x \ y) \ (\text{UnaryExpr UnaryNeg } x)$

inductive *CanonicalizeAdd* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**

add-xsub:

$\llbracket x = (\text{BinaryExpr BinSub } a \ y);$
 $\text{stampa} = \text{stamp-expr } a;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{is-IntegerStamp stampa} \wedge \text{is-IntegerStamp stampy};$
 $\text{stp-bits stampa} = \text{stp-bits stampy} \rrbracket$

$\implies \text{CanonicalizeAdd } (\text{BinaryExpr BinAdd } x \ y) \ a \mid$

add-ysub:

$\llbracket y = (\text{BinaryExpr BinSub } a \ x);$
 $\text{stampa} = \text{stamp-expr } a;$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{is-IntegerStamp stampa} \wedge \text{is-IntegerStamp stampx};$
 $\text{stp-bits stampa} = \text{stp-bits stampx} \rrbracket$
 $\implies \text{CanonicalizeAdd } (\text{BinaryExpr BinAdd } x \ y) \ a \mid$

add-xnegate:

$\llbracket nx = (\text{UnaryExpr UnaryNeg } x);$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy};$
 $\text{stp-bits stampx} = \text{stp-bits stampy} \rrbracket$
 $\implies \text{CanonicalizeAdd } (\text{BinaryExpr BinAdd } nx \ y) \ (\text{BinaryExpr BinSub } y \ x) \mid$

add-ynegate:

$\llbracket ny = (\text{UnaryExpr UnaryNeg } y);$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy};$
 $\text{stp-bits stampx} = \text{stp-bits stampy} \rrbracket$
 $\implies \text{CanonicalizeAdd } (\text{BinaryExpr BinAdd } x \ ny) \ (\text{BinaryExpr BinSub } x \ y)$

inductive *CanonicalizeSub* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**

sub-same32:

$\llbracket \text{stampx} = \text{stamp-expr } x;$
 $\text{stampx} = \text{IntegerStamp } 32 \ \text{lo } \text{hi} \rrbracket$
 $\implies \text{CanonicalizeSub } (\text{BinaryExpr BinSub } x \ x) \ (\text{ConstantExpr } (\text{IntVal32 } 0)) \mid$
sub-same64:

$\llbracket \text{stampx} = \text{stamp-expr } x;$
 $\text{stampx} = \text{IntegerStamp } 64 \ \text{lo } \text{hi} \rrbracket$
 $\implies \text{CanonicalizeSub } (\text{BinaryExpr BinSub } x \ x) \ (\text{ConstantExpr } (\text{IntVal64 } 0)) \mid$

sub-left-add1:

$\llbracket x = (\text{BinaryExpr BinAdd } a \ b);$
 $\text{stampa} = \text{stamp-expr } a;$
 $\text{stampb} = \text{stamp-expr } b;$

$is-IntegerStamp\ stampa \wedge is-IntegerStamp\ stampb;$
 $stp-bits\ stampa = stp-bits\ stampb]$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ x\ b)\ a\ |$

sub-left-add2:

$\llbracket x = (BinaryExpr\ BinAdd\ a\ b);$
 $stamp_a = stamp_expr\ a;$
 $stamp_b = stamp_expr\ b;$
 $is-IntegerStamp\ stamp_a \wedge is-IntegerStamp\ stamp_b;$
 $stp-bits\ stamp_a = stp-bits\ stamp_b]$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ x\ a)\ b\ |$

sub-left-sub:

$\llbracket x = (BinaryExpr\ BinSub\ a\ b);$
 $stamp_a = stamp_expr\ a;$
 $stamp_b = stamp_expr\ b;$
 $is-IntegerStamp\ stamp_a \wedge is-IntegerStamp\ stamp_b;$
 $stp-bits\ stamp_a = stp-bits\ stamp_b]$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ x\ a)\ (UnaryExpr\ UnaryNeg\ b)\ |$

sub-right-add1:

$\llbracket y = (BinaryExpr\ BinAdd\ a\ b);$
 $stamp_a = stamp_expr\ a;$
 $stamp_b = stamp_expr\ b;$
 $is-IntegerStamp\ stamp_a \wedge is-IntegerStamp\ stamp_b;$
 $stp-bits\ stamp_a = stp-bits\ stamp_b]$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ a\ y)\ (UnaryExpr\ UnaryNeg\ b)\ |$

sub-right-add2:

$\llbracket y = (BinaryExpr\ BinAdd\ a\ b);$
 $stamp_a = stamp_expr\ a;$
 $stamp_b = stamp_expr\ b;$
 $is-IntegerStamp\ stamp_a \wedge is-IntegerStamp\ stamp_b;$
 $stp-bits\ stamp_a = stp-bits\ stamp_b]$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ b\ y)\ (UnaryExpr\ UnaryNeg\ a)\ |$

sub-right-sub:

$\llbracket y = (BinaryExpr\ BinSub\ a\ b);$
 $stamp_a = stamp_expr\ a;$
 $stamp_b = stamp_expr\ b;$
 $is-IntegerStamp\ stamp_a \wedge is-IntegerStamp\ stamp_b;$
 $stp-bits\ stamp_a = stp-bits\ stamp_b]$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ a\ y)\ b\ |$

sub-xzero32:
 $\llbracket \text{stamp}x = \text{stamp-expr } x;$
 $\text{stamp}x = \text{IntegerStamp } 32 \text{ lo } hi \rrbracket$
 $\implies \text{CanonicalizeSub } (\text{BinaryExpr } \text{BinSub } (\text{ConstantExpr } (\text{IntVal32 } 0)) \text{ } x)$
 $(\text{UnaryExpr } \text{UnaryNeg } x) \mid$
sub-xzero64:
 $\llbracket \text{stamp}x = \text{stamp-expr } x;$
 $\text{stamp}x = \text{IntegerStamp } 64 \text{ lo } hi \rrbracket$
 $\implies \text{CanonicalizeSub } (\text{BinaryExpr } \text{BinSub } (\text{ConstantExpr } (\text{IntVal64 } 0)) \text{ } x)$
 $(\text{UnaryExpr } \text{UnaryNeg } x) \mid$

sub-y-negate:

$\llbracket nb = (\text{UnaryExpr } \text{UnaryNeg } b);$
 $\text{stamp}a = \text{stamp-expr } a;$
 $\text{stamp}b = \text{stamp-expr } b;$
 $is\text{-IntegerStamp } \text{stamp}a \wedge is\text{-IntegerStamp } \text{stamp}b;$
 $stp\text{-bits } \text{stamp}a = stp\text{-bits } \text{stamp}b \rrbracket$
 $\implies \text{CanonicalizeSub } (\text{BinaryExpr } \text{BinSub } a \text{ } nb) (\text{BinaryExpr } \text{BinAdd } a \text{ } b)$

inductive *CanonicalizeNegate* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
negate-negate:

$\llbracket nx = (\text{UnaryExpr } \text{UnaryNeg } x);$
 $is\text{-IntegerStamp } (\text{stamp-expr } x) \rrbracket$
 $\implies \text{CanonicalizeNegate } (\text{UnaryExpr } \text{UnaryNeg } nx) \text{ } x \mid$

negate-sub:

$\llbracket e = (\text{BinaryExpr } \text{BinSub } x \text{ } y);$
 $\text{stamp}x = \text{stamp-expr } x;$
 $\text{stamp}y = \text{stamp-expr } y;$
 $is\text{-IntegerStamp } \text{stamp}x \wedge is\text{-IntegerStamp } \text{stamp}y;$
 $stp\text{-bits } \text{stamp}x = stp\text{-bits } \text{stamp}y \rrbracket$
 $\implies \text{CanonicalizeNegate } (\text{UnaryExpr } \text{UnaryNeg } e) (\text{BinaryExpr } \text{BinSub } y \text{ } x)$

inductive *CanonicalizeAbs* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
abs-abs:

$\llbracket ax = (\text{UnaryExpr } \text{UnaryAbs } x);$
 $is\text{-IntegerStamp } (\text{stamp-expr } x) \rrbracket$
 $\implies \text{CanonicalizeAbs } (\text{UnaryExpr } \text{UnaryAbs } ax) \text{ } ax \mid$

abs-neg:

$$\begin{aligned} & \llbracket nx = (\text{UnaryExpr } \text{UnaryNeg } x); \\ & \quad \text{is-IntegerStamp } (\text{stamp-expr } x) \rrbracket \\ & \implies \text{CanonicalizeAbs } (\text{UnaryExpr } \text{UnaryAbs } nx) (\text{UnaryExpr } \text{UnaryAbs } x) \end{aligned}$$

inductive *CanonicalizeNot* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
not-not:

$$\begin{aligned} & \llbracket nx = (\text{UnaryExpr } \text{UnaryNot } x); \\ & \quad \text{is-IntegerStamp } (\text{stamp-expr } x) \rrbracket \\ & \implies \text{CanonicalizeNot } (\text{UnaryExpr } \text{UnaryNot } nx) x \end{aligned}$$

inductive *CanonicalizeAnd* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
and-same:

$$\begin{aligned} & \llbracket \text{is-IntegerStamp } (\text{stamp-expr } x) \rrbracket \\ & \implies \text{CanonicalizeAnd } (\text{BinaryExpr } \text{BinAnd } x x) x \mid \end{aligned}$$

and-demorgans:

$$\begin{aligned} & \llbracket nx = (\text{UnaryExpr } \text{UnaryNot } x); \\ & \quad ny = (\text{UnaryExpr } \text{UnaryNot } y); \\ & \quad \text{stampx} = \text{stamp-expr } x; \\ & \quad \text{stampy} = \text{stamp-expr } y; \\ & \quad \text{is-IntegerStamp } \text{stampx} \wedge \text{is-IntegerStamp } \text{stampy}; \\ & \quad \text{stp-bits } \text{stampx} = \text{stp-bits } \text{stampy} \rrbracket \\ & \implies \text{CanonicalizeAnd } (\text{BinaryExpr } \text{BinAnd } nx ny) (\text{UnaryExpr } \text{UnaryNot } \\ & (\text{BinaryExpr } \text{BinOr } x y)) \end{aligned}$$

inductive *CanonicalizeOr* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
or-same:

$$\begin{aligned} & \llbracket \text{is-IntegerStamp } (\text{stamp-expr } x) \rrbracket \\ & \implies \text{CanonicalizeOr } (\text{BinaryExpr } \text{BinOr } x x) x \mid \end{aligned}$$

or-demorgans:

$$\begin{aligned} & \llbracket nx = (\text{UnaryExpr } \text{UnaryNot } x); \\ & \quad ny = (\text{UnaryExpr } \text{UnaryNot } y); \\ & \quad \text{stampx} = \text{stamp-expr } x; \\ & \quad \text{stampy} = \text{stamp-expr } y; \\ & \quad \text{is-IntegerStamp } \text{stampx} \wedge \text{is-IntegerStamp } \text{stampy}; \\ & \quad \text{stp-bits } \text{stampx} = \text{stp-bits } \text{stampy} \rrbracket \\ & \implies \text{CanonicalizeOr } (\text{BinaryExpr } \text{BinOr } nx ny) (\text{UnaryExpr } \text{UnaryNot } (\text{BinaryExpr } \end{aligned}$$

BinAnd x y))

inductive *CanonicalizeIntegerEquals* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
int-equals-same:

$\llbracket x = y \rrbracket$
 \implies *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals* x y) (*ConstantExpr*
(IntVal32 1)) |

int-equals-distinct:
 $\llbracket \text{alwaysDistinct } (\text{stamp-expr } x) (\text{stamp-expr } y) \rrbracket$
 \implies *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals* x y) (*ConstantExpr*
(IntVal32 0)) |

int-equals-add-first-both-same:
 $\llbracket \text{left} = (\text{BinaryExpr BinAdd } x \ y);$
 $\text{right} = (\text{BinaryExpr BinAdd } x \ z) \rrbracket$
 \implies *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals* left right) (*BinaryExpr*
BinIntegerEquals y z) |

int-equals-add-first-second-same:
 $\llbracket \text{left} = (\text{BinaryExpr BinAdd } x \ y);$
 $\text{right} = (\text{BinaryExpr BinAdd } z \ x) \rrbracket$
 \implies *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals* left right) (*BinaryExpr*
BinIntegerEquals y z) |

int-equals-add-second-first-same:
 $\llbracket \text{left} = (\text{BinaryExpr BinAdd } y \ x);$
 $\text{right} = (\text{BinaryExpr BinAdd } x \ z) \rrbracket$
 \implies *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals* left right) (*BinaryExpr*
BinIntegerEquals y z) |

int-equals-add-second-both--same:
 $\llbracket \text{left} = (\text{BinaryExpr BinAdd } y \ x);$
 $\text{right} = (\text{BinaryExpr BinAdd } z \ x) \rrbracket$
 \implies *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals* left right) (*BinaryExpr*
BinIntegerEquals y z) |

int-equals-sub-first-both-same:

$\llbracket \text{left} = (\text{BinaryExpr BinSub } x \ y);$

$right = (BinaryExpr\ BinSub\ x\ z)]$
 $\implies CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ right)\ (BinaryExpr\ BinIntegerEquals\ y\ z)\ |$

int-equals-sub-second-both-same:

$\llbracket left = (BinaryExpr\ BinSub\ y\ x);$
 $right = (BinaryExpr\ BinSub\ z\ x) \rrbracket$
 $\implies CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ right)\ (BinaryExpr\ BinIntegerEquals\ y\ z)\ |$

int-equals-left-contains-right1:

$\llbracket left = (BinaryExpr\ BinAdd\ x\ y);$
 $zero = (ConstantExpr\ (IntVal32\ 0)) \rrbracket$
 $\implies CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ x)\ (BinaryExpr\ BinIntegerEquals\ y\ zero)\ |$

int-equals-left-contains-right2:

$\llbracket left = (BinaryExpr\ BinAdd\ x\ y);$
 $zero = (ConstantExpr\ (IntVal32\ 0)) \rrbracket$
 $\implies CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ y)\ (BinaryExpr\ BinIntegerEquals\ x\ zero)\ |$

int-equals-right-contains-left1:

$\llbracket right = (BinaryExpr\ BinAdd\ x\ y);$
 $zero = (ConstantExpr\ (IntVal32\ 0)) \rrbracket$
 $\implies CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ x\ right)\ (BinaryExpr\ BinIntegerEquals\ y\ zero)\ |$

int-equals-right-contains-left2:

$\llbracket right = (BinaryExpr\ BinAdd\ x\ y);$
 $zero = (ConstantExpr\ (IntVal32\ 0)) \rrbracket$
 $\implies CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ y\ right)\ (BinaryExpr\ BinIntegerEquals\ x\ zero)\ |$

int-equals-left-contains-right3:

$\llbracket left = (BinaryExpr\ BinSub\ x\ y);$
 $zero = (ConstantExpr\ (IntVal32\ 0)) \rrbracket$
 $\implies CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ x)\ (BinaryExpr\ BinIntegerEquals\ y\ zero)\ |$

int-equals-right-contains-left3:

$\llbracket \text{right} = (\text{BinaryExpr BinSub } x \ y);$
 $\text{zero} = (\text{ConstantExpr (IntVal32 0)}) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr BinIntegerEquals } x \ \text{right}) \ (\text{BinaryExpr BinIntegerEquals } y \ \text{zero})$

inductive *CanonicalizeConditional* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
eq-branches:

$\llbracket t = f \rrbracket$
 $\implies \text{CanonicalizeConditional } (\text{ConditionalExpr } c \ t \ f) \ t \mid$

cond-eq:

$\llbracket c = (\text{BinaryExpr BinIntegerEquals } x \ y);$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy};$
 $\text{stp-bits stampx} = \text{stp-bits stampy} \rrbracket$
 $\implies \text{CanonicalizeConditional } (\text{ConditionalExpr } c \ x \ y) \ y \mid$

condition-bounds-x:

$\llbracket c = (\text{BinaryExpr BinIntegerLessThan } x \ y);$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{stpi-upper stampx} \leq \text{stpi-lower stampy};$
 $\text{stp-bits stampx} = \text{stp-bits stampy};$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy} \rrbracket$
 $\implies \text{CanonicalizeConditional } (\text{ConditionalExpr } c \ x \ y) \ x \mid$

condition-bounds-y:

$\llbracket c = (\text{BinaryExpr BinIntegerLessThan } x \ y);$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{stpi-upper stampx} \leq \text{stpi-lower stampy};$
 $\text{stp-bits stampx} = \text{stp-bits stampy};$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy} \rrbracket$
 $\implies \text{CanonicalizeConditional } (\text{ConditionalExpr } c \ y \ x) \ y \mid$

negate-condition:

$\llbracket nc = (\text{UnaryExpr UnaryLogicNegation } c);$
 $\text{stampc} = \text{stamp-expr } c;$
 $\text{stampc} = \text{IntegerStamp } 32 \ \text{lo } \text{hi};$
 $\text{stampx} = \text{stamp-expr } x;$

$stampy = stamp\text{-}expr\ y;$
 $stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
 $is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy$
 $\implies CanonicalizeConditional\ (ConditionalExpr\ nc\ x\ y)\ (ConditionalExpr\ c\ y\ x)$

const-true:

$\llbracket c = ConstantExpr\ val;$
 $val\text{-}to\text{-}bool\ val \rrbracket$
 $\implies CanonicalizeConditional\ (ConditionalExpr\ c\ t\ f)\ t$

const-false:

$\llbracket c = ConstantExpr\ val;$
 $\neg(val\text{-}to\text{-}bool\ val) \rrbracket$
 $\implies CanonicalizeConditional\ (ConditionalExpr\ c\ t\ f)\ f$

inductive *CanonicalizationStep* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**

BinaryNode:
 $\llbracket CanonicalizeBinaryOp\ expr\ expr' \rrbracket$
 $\implies CanonicalizationStep\ expr\ expr'$

UnaryNode:
 $\llbracket CanonicalizeUnaryOp\ expr\ expr' \rrbracket$
 $\implies CanonicalizationStep\ expr\ expr'$

NegateNode:
 $\llbracket CanonicalizeNegate\ expr\ expr' \rrbracket$
 $\implies CanonicalizationStep\ expr\ expr'$

NotNode:
 $\llbracket CanonicalizeNegate\ expr\ expr' \rrbracket$
 $\implies CanonicalizationStep\ expr\ expr'$

AddNode:
 $\llbracket CanonicalizeAdd\ expr\ expr' \rrbracket$
 $\implies CanonicalizationStep\ expr\ expr'$

MulNode:
 $\llbracket CanonicalizeMul\ expr\ expr' \rrbracket$

```

     $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$ 

SubNode:
 $\llbracket \text{CanonicalizeSub } \text{expr } \text{expr}' \rrbracket$ 
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$ 

AndNode:
 $\llbracket \text{CanonicalizeSub } \text{expr } \text{expr}' \rrbracket$ 
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$ 

OrNode:
 $\llbracket \text{CanonicalizeSub } \text{expr } \text{expr}' \rrbracket$ 
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$ 

IntegerEqualsNode:
 $\llbracket \text{CanonicalizeIntegerEquals } \text{expr } \text{expr}' \rrbracket$ 
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$ 

ConditionalNode:
 $\llbracket \text{CanonicalizeConditional } \text{expr } \text{expr}' \rrbracket$ 
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}'$ 

code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeBinaryOp .
code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeUnaryOp .
code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeNegate .
code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeNot .
code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeAdd .
code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeSub .
code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeMul .
code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeAnd .
code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeIntegerEquals .
code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeConditional .

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

end

```

9 Canonicalization Phase

```

theory CanonicalizationTreeProofs
  imports
    CanonicalizationTree
    Semantics.IRTreeEvalThms
begin

```

```

lemma neutral-rewrite-helper:
  shows valid-value (IntegerStamp 32 lo hi)  $x \implies \text{intval-mul } x \text{ (IntVal32 (1))} = x$ 
  and valid-value (IntegerStamp 64 lo hi)  $x \implies \text{intval-mul } x \text{ (IntVal64 (1))} = x$ 

```

```

and   valid-value (IntegerStamp 32 lo hi) x  $\implies$  intval-add x (IntVal32 0) = x
and   valid-value (IntegerStamp 64 lo hi) x  $\implies$  intval-add x (IntVal64 0) = x

and   valid-value (IntegerStamp 32 lo hi) x  $\implies$  intval-sub x (IntVal32 0) = x
and   valid-value (IntegerStamp 64 lo hi) x  $\implies$  intval-sub x (IntVal64 0) = x

and   valid-value (IntegerStamp 32 lo hi) x  $\implies$  intval-xor x (IntVal32 0) = x
and   valid-value (IntegerStamp 64 lo hi) x  $\implies$  intval-xor x (IntVal64 0) = x

using valid32or64-both by fastforce+

lemma CanonicalizeBinaryProof:
  assumes CanonicalizeBinaryOp before after
  assumes [m, p]  $\vdash$  before  $\mapsto$  res
  assumes [m, p]  $\vdash$  after  $\mapsto$  res'
  shows res = res'
  using assms
proof (induct rule: CanonicalizeBinaryOp.induct)
  case (binary-const-fold x val1 y val2 val op)
  then show ?case by auto
next
  case (binary-fold-yneutral y c op stampx x stampy)
  obtain xval where x-eval: [m, p]  $\vdash$  x  $\mapsto$  xval
  using binary-fold-yneutral.prem(2) by auto
  then have bin-eval op xval c = xval
  using neutral-rewrite-helper binary-fold-yneutral.hyps(2-3,6-) int-stamp-implies-valid-value
  is-IntegerStamp-def
  by (metis ConstantExpr Value.distinct(5) valid-value.simps(34))
  then show ?case
  by (metis binary-fold-yneutral.hyps(1) binary-fold-yneutral.prem(1) binary-fold-yneutral.prem(2)
  x-eval
  BinaryExprE ConstantExprE evalDet)
next
  case (binary-fold-yzero32 y c op stampx x stampy)
  obtain xval where x-eval: [m, p]  $\vdash$  x  $\mapsto$  xval
  using binary-fold-yzero32.prem(1) by auto
  then have bin-eval op xval c = (IntVal32 0)
  using ConstantExpr Value.distinct(5) int-stamp-implies-valid-value binary-fold-yzero32.hyps
  is-IntegerStamp-def
  by (metis valid-value.simps(34))
  then show ?case
  by (metis BinaryExpr ConstantExpr ConstantExprE binary-fold-yzero32.hyps(1)
  binary-fold-yzero32.hyps(2)
  binary-fold-yzero32.prem(1) binary-fold-yzero32.prem(2) evalDet is-zero.simps(11)
  x-eval)
next
  case (binary-fold-yzero64 y c op stampx x stampy)

```

```

obtain xval where x-eval:  $[m, p] \vdash x \mapsto xval$ 
using binary-fold-yzero64.prem(1) by auto
then have bin-eval op xval c = (IntVal64 0)
using ConstantExpr Value.distinct(5) int-stamp-implies-valid-value binary-fold-yzero64.hyps
is-IntegerStamp-def
by (metis valid-value.simps(34))
then show ?case
by (metis BinaryExprE ConstantExprE binary-fold-yzero64.hyps(1) binary-fold-yzero64.prem(1)
binary-fold-yzero64.prem(2) evalDet x-eval)
qed

```

```

lemma CanonicalizeUnaryProof:
assumes CanonicalizeUnaryOp before after
assumes  $[m, p] \vdash \textit{before} \mapsto \textit{res}$ 
assumes  $[m, p] \vdash \textit{after} \mapsto \textit{res}'$ 
shows  $\textit{res} = \textit{res}'$ 
using assms
proof (induct rule: CanonicalizeUnaryOp.induct)
case (unary-const-fold val' op val)
then show ?case by auto
qed

```

```

lemma mul-rewrite-helper:
shows valid-value (IntegerStamp 32 lo hi)  $x \implies \textit{intval-mul } x \text{ (IntVal32 } (-1)) =$ 
intval-negate x
and valid-value (IntegerStamp 64 lo hi)  $x \implies \textit{intval-mul } x \text{ (IntVal64 } (-1)) =$ 
intval-negate x
using valid32or64-both by fastforce +

```

```

lemma CanonicalizeMulProof:
assumes CanonicalizeMul before after
assumes  $[m, p] \vdash \textit{before} \mapsto \textit{res}$ 
assumes  $[m, p] \vdash \textit{after} \mapsto \textit{res}'$ 
shows  $\textit{res} = \textit{res}'$ 
using assms
proof (induct rule: CanonicalizeMul.induct)
case (mul-negate32 y x lo hi)
then show ?case
using ConstantExprE BinaryExprE bin-eval.simps evalDet mul-rewrite-helper
int-stamp-implies-valid-value
by (auto; metis)
next
case (mul-negate64 y x lo hi)
then show ?case
using ConstantExprE BinaryExprE bin-eval.simps evalDet mul-rewrite-helper
int-stamp-implies-valid-value
by (auto; metis)
qed

```

```

lemma add-rewrites-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  and    valid-value (IntegerStamp b loy hiy) y

  shows intval-add (intval-sub x y) y = x
  and  intval-add x (intval-sub y x) = y
  and  intval-add (intval-negate x) y = intval-sub y x
  and  intval-add x (intval-negate y) = intval-sub x y
  using valid32or64-both assms by fastforce+
```

```

lemma CanonicalizeAddProof:
  assumes CanonicalizeAdd before after
  assumes [m, p] ⊢ before ↦ res
  assumes [m, p] ⊢ after ↦ res'
  shows res = res'
  using assms
proof (induct rule: CanonicalizeAdd.induct)
  case (add-xsub x a y stampa stampy)
  then show ?case
    by (metis BinaryExprE Stamp.collapse(1) bin-eval.simps(1) bin-eval.simps(3)
      evalDet int-stamp-implies-valid-value intval-add-sym add-rewrites-helper(1))
  next
  case (add-ysub y a x stampa stampx)
  then show ?case
    by (metis is-IntegerStamp-def add-ysub.hyps add-ysub.premis evalDet BinaryExprE Stamp.sel(1)
      bin-eval.simps(1) bin-eval.simps(3) int-stamp-implies-valid-value intval-add-sym
      add-rewrites-helper(2))
  next
  case (add-xnegate nx x stampx stampy)
  then show ?case
    by (smt (verit, del-insts) BinaryExprE Stamp.sel(1) UnaryExprE add-rewrites-helper(4)

      bin-eval.simps(1) bin-eval.simps(3) evalDet int-stamp-implies-valid-value
      intval-add-sym is-IntegerStamp-def unary-eval.simps(2))
  next
  case (add-ynegate ny y stampx stampy)
  then show ?case
    by (smt (verit) BinaryExprE Stamp.sel(1) UnaryExprE add-rewrites-helper(4)
      bin-eval.simps(1)
      bin-eval.simps(3) evalDet int-stamp-implies-valid-value is-IntegerStamp-def
      unary-eval.simps(2))
qed
```

```

lemma sub-rewrites-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  and    valid-value (IntegerStamp b loy hiy) y

  shows intval-sub (intval-add x y) y = x
  and  intval-sub (intval-add x y) x = y
  and  intval-sub (intval-sub x y) x = intval-negate y
  and  intval-sub x (intval-add x y) = intval-negate y
  and  intval-sub y (intval-add x y) = intval-negate x
  and  intval-sub x (intval-sub x y) = y
  and  intval-sub x (intval-negate y) = intval-add x y
  using valid32or64-both assms by fastforce+
```

```

lemma sub-single-rewrites-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  shows  b = 32  $\implies$  intval-sub x x = IntVal32 0
  and    b = 64  $\implies$  intval-sub x x = IntVal64 0
  and    b = 32  $\implies$  intval-sub (IntVal32 0) x = intval-negate x
  and    b = 64  $\implies$  intval-sub (IntVal64 0) x = intval-negate x
  using valid32or64-both assms by fastforce+
```

```

lemma CanonicalizeSubProof:
  assumes CanonicalizeSub before after
  assumes [m, p]  $\vdash$  before  $\mapsto$  res
  assumes [m, p]  $\vdash$  after  $\mapsto$  res'
  shows res = res'
  using assms
proof (induct rule: CanonicalizeSub.induct)
  case (sub-same32 stampx x lo hi)
  show ?case
    using ConstantExprE BinaryExprE bin-eval.simps evalDet sub-same32.prem
    sub-single-rewrites-helper
    int-stamp-implies-valid-value sub-same32.hyps(1) sub-same32.hyps(2)
    by (auto; metis)
  next
  case (sub-same64 stampx x lo hi)
  show ?case
    using ConstantExprE BinaryExprE bin-eval.simps evalDet sub-same64.prem
    sub-single-rewrites-helper
    int-stamp-implies-valid-value sub-same64.hyps(1) sub-same64.hyps(2)
    by (auto; metis)
```

```

next
  case (sub-left-add1 x a b stampa stampb)
  then show ?case
    by (metis BinaryExprE Stamp.collapse(1) bin-eval.simps(1) bin-eval.simps(3)
evalDet
      int-stamp-implies-valid-value sub-rewrites-helper(1))
next
  case (sub-left-add2 x a b stampa stampb)
  then show ?case
    by (metis BinaryExprE Stamp.collapse(1) bin-eval.simps(1) bin-eval.simps(3)
evalDet
      int-stamp-implies-valid-value sub-rewrites-helper(2))
next
  case (sub-left-sub x a b stampa stampb)
  then show ?case
    by (smt (verit) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(3)
evalDet
      int-stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(3)
unary-eval.simps(2))
next
  case (sub-right-add1 y a b stampa stampb)
  then show ?case
    by (smt (verit) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(1)
bin-eval.simps(3) evalDet
      int-stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(4)
unary-eval.simps(2))
next
  case (sub-right-add2 y a b stampa stampb)
  then show ?case
    by (smt (verit) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(1)
bin-eval.simps(3) evalDet
      int-stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(5)
unary-eval.simps(2))
next
  case (sub-right-sub y a b stampa stampb)
  then show ?case
    by (metis BinaryExprE Stamp.sel(1) bin-eval.simps(3) evalDet
      int-stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(6))
next
  case (sub-xzero32 stampx x lo hi)
  then show ?case
    using ConstantExprE BinaryExprE bin-eval.simps evalDet sub-xzero32.premis
sub-single-rewrites-helper
      int-stamp-implies-valid-value sub-xzero32.hyps(1) sub-xzero32.hyps(2)
    by (auto; metis)
next
  case (sub-xzero64 stampx x lo hi)
  then show ?case
    using ConstantExprE BinaryExprE bin-eval.simps evalDet sub-xzero64.premis

```



```

sub-single-rewrites-helper
  int-stamp-implies-valid-value sub-xzero64.hyps(1) sub-xzero64.hyps(2)
  by (auto; metis)
next
  case (sub-y-negate nb b stampa a stampb)
  then show ?case
    by (smt (verit, best) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(1)
bin-eval.simps(3) evalDet
      int-stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(7)
unary-eval.simps(2))
qed

```

```

lemma negate-xsuby-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  and valid-value (IntegerStamp b loy hiy) y
  shows intval-negate (intval-sub x y) = intval-sub y x
  using valid32or64-both assms by fastforce

```

```

lemma negate-negate-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  shows intval-negate (intval-negate x) = x
  using valid32or64 assms by fastforce

```

```

lemma CanonicalizeNegateProof:
  assumes CanonicalizeNegate before after
  assumes [m, p] ⊢ before ↦ res
  assumes [m, p] ⊢ after ↦ res'
  shows res = res'
  using assms
proof (induct rule: CanonicalizeNegate.induct)
  case (negate-negate nx x)
  thus ?case
    by (metis UnaryExprE evalDet int-stamp-implies-valid-value is-IntegerStamp-def
negate-negate-helper unary-eval.simps(2))
next
  case (negate-sub e x y stampx stampy)
  thus ?case
    by (smt (verit) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(3)
evalDet int-stamp-implies-valid-value
      is-IntegerStamp-def negate-xsuby-helper unary-eval.simps(2))
qed

```

```

lemma word-helper:
  shows  $\bigwedge x :: 32 \text{ word. } \neg(\neg x <_s 0 \wedge x <_s 0)$ 
  and  $\bigwedge x :: 64 \text{ word. } \neg(\neg x <_s 0 \wedge x <_s 0)$ 
  and  $\bigwedge x :: 32 \text{ word. } \neg \neg x <_s 0 \wedge \neg x <_s 0 \implies 2 * x = 0$ 
  and  $\bigwedge x :: 64 \text{ word. } \neg \neg x <_s 0 \wedge \neg x <_s 0 \implies 2 * x = 0$ 

```

```

apply (case-tac[!] x)
apply auto+
sorry

```

```

lemma abs-rewrite-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  shows intval-abs (intval-negate x) = intval-abs x
  and intval-abs (intval-abs x) = intval-abs x
  apply (case-tac[!] x)
  using word-helper apply auto+
done

```

```

lemma CanonicalizeAbsProof:
  assumes CanonicalizeAbs before after
  assumes [m, p] ⊢ before ↦ res
  assumes [m, p] ⊢ after ↦ res'
  shows res = res'
  using assms
proof (induct rule: CanonicalizeAbs.induct)
  case (abs-abs ax x)
  then show ?case
  by (metis UnaryExprE abs-rewrite-helper(2) evalDet int-stamp-implies-valid-value
    is-IntegerStamp-def
      unary-eval.simps(1))
next
  case (abs-neg nx x)
  then show ?case
  by (metis UnaryExprE abs-rewrite-helper(1) evalDet int-stamp-implies-valid-value
    is-IntegerStamp-def
      unary-eval.simps(1) unary-eval.simps(2))
qed

```

```

lemma not-rewrite-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  shows intval-not (intval-not x) = x
  using valid32or64 assms by fastforce+

```

```

lemma CanonicalizeNotProof:
  assumes CanonicalizeNot before after
  assumes [m, p] ⊢ before ↦ res
  assumes [m, p] ⊢ after ↦ res'
  shows res = res'
  using assms
proof (induct rule: CanonicalizeNot.induct)
  case (not-not nx x)
  then show ?case

```

```

    by (metis UnaryExprE evalDet is-IntegerStamp-def not-rewrite-helper
        int-stamp-implies-valid-value unary-eval.simps(3))
qed

lemma demorgans-rewrites-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  and     valid-value (IntegerStamp b loy hiy) y

  shows intval-and (intval-not x) (intval-not y) = intval-not (intval-or x y)
  and   intval-or (intval-not x) (intval-not y) = intval-not (intval-and x y)
  and   x = y  $\implies$  intval-and x y = x
  and   x = y  $\implies$  intval-or x y = x
  using valid32or64-both assms by fastforce+

lemma CanonicalizeAndProof:
  assumes CanonicalizeAnd before after
  assumes [m, p]  $\vdash$  before  $\mapsto$  res
  assumes [m, p]  $\vdash$  after  $\mapsto$  res'
  shows res = res'
  using assms
proof (induct rule: CanonicalizeAnd.induct)
  case (and-same x)
  then show ?case
    by (metis BinaryExprE bin-eval.simps(4) demorgans-rewrites-helper(3) evalDet
        int-stamp-implies-valid-value is-IntegerStamp-def)
next
  case (and-demorgans nx x ny y stampx stampy)
  then show ?case
    by (smt (z3) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(4) bin-eval.simps(5)

        demorgans-rewrites-helper(1) evalDet int-stamp-implies-valid-value is-IntegerStamp-def
        unary-eval.simps(3))
qed

lemma CanonicalizeOrProof:
  assumes CanonicalizeOr before after
  assumes [m, p]  $\vdash$  before  $\mapsto$  res
  assumes [m, p]  $\vdash$  after  $\mapsto$  res'
  shows res = res'
  using assms
proof (induct rule: CanonicalizeOr.induct)
  case (or-same x)
  then show ?case
    by (metis BinaryExprE bin-eval.simps(5) demorgans-rewrites-helper(4) evalDet
        int-stamp-implies-valid-value is-IntegerStamp-def)
next
  case (or-demorgans nx x ny y stampx stampy)
  then show ?case
    by (smt (z3) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(4) bin-eval.simps(5))

```

```

demorgans-rewrites-helper(2)
  evalDet int-stamp-implies-valid-value is-IntegerStamp-def unary-eval.simps(3))
qed

lemma stamps-touch-but-not-less-than-implies-equal:
  [[valid-value stampx x;
    valid-value stampy y;
    is-IntegerStamp stampx  $\wedge$  is-IntegerStamp stampy;
    stpi-upper stampx = stpi-lower stampy;
     $\neg$  val-to-bool (intval-less-than x y)]]  $\implies$  x = y
using valid32or64-both intval-equals.simps(1-2) intval-less-than.simps(1-2) val-to-bool.simps(1)
sorry

lemma disjoint-stamp-implies-less-than:
  [[valid-value stampx x;
    valid-value stampy y;
    is-IntegerStamp stampx  $\wedge$  is-IntegerStamp stampy;
    stpi-upper stampx < stpi-lower stampy]]
 $\implies$  val-to-bool(intval-less-than x y)
sorry

lemma CanonicalizeConditionalProof:
  assumes CanonicalizeConditional before after
  assumes [m, p]  $\vdash$  before  $\mapsto$  res
  assumes [m, p]  $\vdash$  after  $\mapsto$  res'
  shows res = res'
  using assms
proof (induct rule: CanonicalizeConditional.induct)
  case (eq-branches t f c)
  then show ?case using evalDet by auto
next
  case (cond-eq c x y stampx stampy)
  obtain xval where xeval: [m,p]  $\vdash$  x  $\mapsto$  xval
  using cond-eq.hyps(1) cond-eq.prem(1) by blast
  obtain yval where yeval: [m,p]  $\vdash$  y  $\mapsto$  yval
  using cond-eq.prem(2) by auto
  show ?case proof (cases xval = yval)
    case True
    then show ?thesis
    by (smt (verit, ccv-threshold) ConditionalExprE cond-eq.prem(1) cond-eq.prem(2)
    evalDet xeval yeval)
  next
  case False
  then have  $\neg$ (val-to-bool(intval-equals xval yval))
  using ConstantExpr Value.distinct(9) valid-value.simps(42) int-stamp-implies-valid-value
  by metis
  then have res = yval
  by (smt (verit) ConditionalExprE BinaryExprE xeval yeval evalDet bin-eval.simps(7)
  cond-eq.hyps(1) cond-eq.prem(1))

```

```

    then show ?thesis
      using cond-eq.premis(1) cond-eq.premis(2) xeval yeval evalDet by auto
qed
next
case (condition-bounds-x c x y stampx stampy)
obtain xval where xeval: [m,p] ⊢ x ↦ xval
  using condition-bounds-x.premis(2) by auto
obtain yval where yeval: [m,p] ⊢ y ↦ yval
  using condition-bounds-x.hyps(1) condition-bounds-x.premis(1) by blast
then show ?case proof (cases val-to-bool(intval-less-than xval yval))
  case True
  then show ?thesis
    by (smt (verit, best) xeval yeval evalDet ConditionalExprE BinaryExprE
bin-eval.simps(8)
    condition-bounds-x.hyps(1) condition-bounds-x.premis(1) condition-bounds-x.premis(2))
  next
  case False
  then have stpi-upper stampx = stpi-lower stampy
    by (metis False condition-bounds-x.hyps(4) order.not-eq-order-implies-strict
disjoint-stamp-implies-less-than condition-bounds-x.hyps(2) condition-bounds-x.hyps(3)
condition-bounds-x.hyps(6)
    int-stamp-implies-valid-value xeval yeval)
  then have (xval = yval)
    by (metis False condition-bounds-x.hyps(2-3,6) int-stamp-implies-valid-value
stamps-touch-but-not-less-than-implies-equal xeval yeval)
  then have res = xval ∧ res' = xval
    using ConditionalExprE condition-bounds-x.premis(1) ⟨[m,p] ⊢ x ↦ res'⟩
evalDet xeval yeval
    by force
  then show ?thesis by simp
qed
next
case (condition-bounds-y c x y stampx stampy)
obtain xval where xeval: [m,p] ⊢ x ↦ xval
  using condition-bounds-y.hyps(1) condition-bounds-y.premis(1) by auto
obtain yval where yeval: [m,p] ⊢ y ↦ yval
  using condition-bounds-y.hyps(1) condition-bounds-y.premis(1) by blast
then show ?case proof (cases val-to-bool(intval-less-than xval yval))
  case True
  then show ?thesis
    by (smt (verit, best) xeval yeval evalDet ConditionalExprE BinaryExprE
bin-eval.simps(8)
    condition-bounds-y.hyps(1) condition-bounds-y.premis(1) condition-bounds-y.premis(2))
  next
  case False
  have stpi-upper stampx = stpi-lower stampy
    by (metis False condition-bounds-y.hyps(4) order.not-eq-order-implies-strict
disjoint-stamp-implies-less-than condition-bounds-y.hyps(2) condition-bounds-y.hyps(3)
condition-bounds-y.hyps(6) int-stamp-implies-valid-value xeval yeval)

```

```

    then have (xval = yval)
      by (metis False condition-bounds-y.hyps(2-3,6)
        int-stamp-implies-valid-value stamps-touch-but-not-less-than-implies-equal
        xeval yeval)
    then have res = yval  $\wedge$  res' = yval
      using ConditionalExprE condition-bounds-y.prem(1)  $\langle [m,p] \vdash y \mapsto res' \rangle$ 
    evalDet xeval yeval
      by force
    then show ?thesis by simp
  qed
next
case (negate-condition nc c stampc lo hi stampx x stampy y)
obtain cval where cval:  $[m,p] \vdash c \mapsto cval$ 
  using negate-condition.prem(2) by auto
obtain nceval where nceval:  $[m,p] \vdash nc \mapsto nceval$ 
  using negate-condition.prem negate-condition.prem by blast
then show ?case using assms proof (cases (val-to-bool nceval))
case True
  obtain xval where xeval:  $[m,p] \vdash x \mapsto xval$ 
  by (metis (full-types) ConditionalExprE nceval evalDet True negate-condition.prem(1))
  then have res = xval
    by (metis True nceval negate-condition.prem(1) evaltree.ConditionalExpr
      xeval evalDet)
  moreover have res' = xval
  by (metis (no-types, hide-lams) nceval negate-condition.prem(1) evaltree.ConditionalExpr
    xeval
      evalDet IRTreeEval.val-to-bool.elims(2) True UnaryExpr Value.inject(1)
       $\langle \wedge thesis. (\wedge cval. [m,p] \vdash c \mapsto cval \implies thesis) \implies thesis \rangle$ 
      negate-condition.hyps(1)
      negate-condition.prem(2) unary-eval.simp(4))
  ultimately show ?thesis by simp
next
case False
  obtain yval where yeval:  $[m,p] \vdash y \mapsto yval$ 
  by (metis (full-types) ConditionalExprE nceval evalDet False negate-condition.prem(1))
  then have res = yval
    using False nceval negate-condition.prem(1) evaltree.ConditionalExpr yeval
    evalDet by presburger
  moreover have val-to-bool(cval)
  by (metis False UnaryExprE cval nceval negate-condition.hyps(1-3) unary-eval.simp(4)
    IRTreeEval.val-to-bool.simp(1) evalDet IRTreeEval.bool-to-val.simp(2)
    int-stamp-implies-valid-value valid-int32 zero-neq-one)
  moreover have res' = yval
    using calculation(2) cval negate-condition.prem evaltree.ConditionalExpr
    yeval evalDet unary-eval.simp(4)
    by presburger
  ultimately show ?thesis by simp
qed
next
case (const-true c val t f)

```

```
    then show ?case using evalDet by auto
next
  case (const-false c val t f)
  then show ?case using evalDet by auto
qed

end
```