Veriopt

January 8, 2022

Abstract

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

Contents

1 Runtime Values and Arithmetic **3**

2 Nodes **9**
- 2.1 Types of Nodes . . . 9
- 2.2 Hierarchy of Nodes . . . 17

3 Stamp Typing **24**

4 Graph Representation **27**
- 4.0.1 Example Graphs . . . 31

5 Data-flow Semantics **32**
- 5.1 Data-flow Tree Representation . . . 32
- 5.2 Data-flow Tree Evaluation . . . 34
- 5.3 Data-flow Tree Refinement . . . 36

6 Data-flow Expression-Tree Theorems **36**
- 6.1 Extraction and Evaluation of Expression Trees is Deterministic. 37
- 6.2 Example Data-flow Optimisations . . . 42
- 6.3 Monotonicity of Expression Optimization . . . 42

7 Tree to Graph **43**

8 Control-flow Semantics **56**
- 8.1 Heap . . . 56
- 8.2 Intraprocedural Semantics . . . 56
- 8.3 Interprocedural Semantics . . . 59
- 8.4 Big-step Execution . . . 60
  - 8.4.1 Heap Testing . . . 61

9 Properties of Control-flow Semantics **62**

10 Proof Infrastructure **63**
- 10.1 Bisimulation . . . 63
- 10.2 Formedness Properties . . . 64
- 10.3 Dynamic Frames . . . 65
- 10.4 Graph Rewriting . . . 70
- 10.5 Stuttering . . . 72

11 Canonicalization Phase **73**

12 Canonicalization Phase **84**

1 Runtime Values and Arithmetic

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

In order to properly implement the IR semantics we first introduce a 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 (IntVal b v) should satisfy the invariants:

$b \in \{1::'a,\ 8::'a,\ 16::'a,\ 32::'a,\ 64::'a\}$

$1 < b \Longrightarrow v \equiv scast\ (signed\text{-}take\text{-}bit\ b\ v)$

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 |

  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* ⇒ *int* ⇒ *bool* **where**
  $fits\text{-}into\text{-}n\ b\ val = ((-(2\widehat{\ }(b-1)) \leq val) \wedge (val < (2\widehat{\ }(b-1))))$

fun *wf-bool* :: *Value* ⇒ *bool* **where**
wf-bool (*IntVal32 v*) = (*v* = *0* ∨ *v* = *1*) |
wf-bool - = *False*

fun *val-to-bool* :: *Value* ⇒ *bool* **where**
val-to-bool (*IntVal32 v*) = (*v* = *1*) |
val-to-bool - = *False*

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

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

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

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* ⇒ *Value* ⇒ *Value* **where**
intval-add32 (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1*+*v2*)) |
intval-add32 - - = *UndefVal*

fun *intval-add64* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-add64 (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1*+*v2*)) |
intval-add64 - - = *UndefVal*

fun *intval-add* :: *Value* ⇒ *Value* ⇒ *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* ⇒ *Value* ⇒ *Value* **where**
plus-Value = *intval-add*

instance ⟨*proof*⟩
end

fun *intval-sub* :: *Value* ⇒ *Value* ⇒ *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* ⇒ *Value* ⇒ *Value* **where**
minus-Value = *intval-sub*

instance ⟨*proof*⟩
end

fun *intval-mul* :: *Value* ⇒ *Value* ⇒ *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* ⇒ *Value* ⇒ *Value* **where**
times-Value = *intval-mul*

instance ⟨*proof*⟩
end

fun *intval-div* :: *Value* ⇒ *Value* ⇒ *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* ⇒ *Value* ⇒ *Value* **where**
divide-Value = *intval-div*

instance ⟨*proof*⟩
end

fun *intval-mod* :: *Value* ⇒ *Value* ⇒ *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* ⇒ *Value* ⇒ *Value* **where**
modulo-Value = *intval-mod*

instance ⟨*proof*⟩
end

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

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

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

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

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

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

lemma [*code*]: *shiftl1 n* = *n* ∗ *2*
  ⟨*proof*⟩

lemma [*code*]: *shiftr1 n* = *n div 2*
  ⟨*proof*⟩

lemma [*code*]: *sshiftr1 n* = *word-of-int* (*sint n div 2*)
  ⟨*proof*⟩

definition *shiftl* (**infix** << *75*) **where**
  shiftl w n = (*shiftl1* ^^ *n*) *w*

lemma *shiftl-power*[*simp*]: (*x*::(′*a*::*len*) *word*) ∗ (*2* ^ *j*) = *x* << *j*
  ⟨*proof*⟩

lemma (*x*::(′*a*::*len*) *word*) ∗ ((*2* ^ *j*) + *1*) = *x* << *j* + *x*
  ⟨*proof*⟩

lemma (*x*::(′*a*::*len*) *word*) ∗ ((*2* ^ *j*) − *1*) = *x* << *j* − *x*
  ⟨*proof*⟩

lemma (*x*::(′*a*::*len*) *word*) ∗ ((*2*^*j*) + (*2*^*k*)) = *x* << *j* + *x* << *k*
  ⟨*proof*⟩

lemma (*x*::(′*a*::*len*) *word*) ∗ ((*2*^*j*) − (*2*^*k*)) = *x* << *j* − *x* << *k*
  ⟨*proof*⟩

definition *signed-shiftr* (**infix** >> *75*) **where**

signed-shiftr w n = (*sshiftr1* ^^ *n*) *w*

definition *shiftr* (**infix** >>> *75*) **where**
shiftr w n = (*shiftr1* ^^ *n*) *w*

lemma *shiftr-power*[*simp*]: (*x*::(*'a*::*len*) *word*) *div* (*2* ^ *j*) = *x* >>> *j*
⟨*proof*⟩

fun *intval-left-shift* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-left-shift (*IntVal32 v1*) (*IntVal32 v2*) = *IntVal32* (*v1* << *unat* (*v2 AND 0x1f*)) |
intval-left-shift (*IntVal64 v1*) (*IntVal64 v2*) = *IntVal64* (*v1* << *unat* (*v2 AND 0x3f*)) |
intval-left-shift - - = *UndefVal*

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

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

lemma *word-add-sym*:
shows *word-of-int v1* + *word-of-int v2* = *word-of-int v2* + *word-of-int v1*
⟨*proof*⟩

lemma *intval-add-sym*:
shows *intval-add a b* = *intval-add b a*
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *intval-bad1* [*simp*]: *intval-add* (*IntVal32 x*) (*IntVal64 y*) = *UndefVal*
⟨*proof*⟩
lemma *intval-bad2* [*simp*]: *intval-add* (*IntVal64 x*) (*IntVal32 y*) = *UndefVal*
⟨*proof*⟩

lemma *intval-assoc*: *intval-add32* (*intval-add32 x y*) *z* = *intval-add32 x* (*intval-add32 y z*)
⟨*proof*⟩

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

lemma *intval-add* (*IntVal32* (*2^31−1*)) (*IntVal32* (*2^31−1*)) = *IntVal32* (−*2*)
⟨*proof*⟩
lemma *intval-add* (*IntVal64* (*2^31−1*)) (*IntVal64* (*2^31−1*)) = *IntVal64 4294967294*
⟨*proof*⟩

end

2 Nodes

2.1 Types of Nodes

theory *IRNodes*
imports
Values
begin

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

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

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

To identify each Node, we use a simple natural number index. Zero is always the start node in a graph. For human readability, within nodes we write

INPUT (or special case thereof) instead of ID for input edges, and SUCC instead of ID for control-flow successor edges. Optional edges are handled as "INPUT option" etc.

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

datatype (*discs-sels*) *IRNode* =
AbsNode (*ir-value*: *INPUT*)
| *AddNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *AndNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *BeginNode* (*ir-next*: *SUCC*)
| *BytecodeExceptionNode* (*ir-arguments*: *INPUT list*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *ConditionalNode* (*ir-condition*: *INPUT-COND*) (*ir-trueValue*: *INPUT*) (*ir-falseValue*: *INPUT*)
| *ConstantNode* (*ir-const*: *Value*)
| *DynamicNewArrayNode* (*ir-elementType*: *INPUT*) (*ir-length*: *INPUT*) (*ir-voidClass-opt*: *INPUT option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *EndNode*
| *ExceptionObjectNode* (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *FrameState* (*ir-monitorIds*: *INPUT-ASSOC list*) (*ir-outerFrameState-opt*: *INPUT-STATE option*) (*ir-values-opt*: *INPUT list option*) (*ir-virtualObjectMappings-opt*: *INPUT-STATE list option*)
| *IfNode* (*ir-condition*: *INPUT-COND*) (*ir-trueSuccessor*: *SUCC*) (*ir-falseSuccessor*: *SUCC*)
| *IntegerBelowNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *IntegerEqualsNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *IntegerLessThanNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *InvokeNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *InvokeWithExceptionNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*) (*ir-exceptionEdge*: *SUCC*)
| *IsNullNode* (*ir-value*: *INPUT*)
| *KillingBeginNode* (*ir-next*: *SUCC*)
| *LeftShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *LoadFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)
| *LogicNegationNode* (*ir-value*: *INPUT-COND*)
| *LoopBeginNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-overflowGuard-opt*: *INPUT-GUARD*

option) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *LoopEndNode* (*ir-loopBegin*: *INPUT-ASSOC*)
| *LoopExitNode* (*ir-loopBegin*: *INPUT-ASSOC*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *MergeNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *MethodCallTargetNode* (*ir-targetMethod*: *string*) (*ir-arguments*: *INPUT list*)
| *MulNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *NarrowNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
| *NegateNode* (*ir-value*: *INPUT*)
| *NewArrayNode* (*ir-length*: *INPUT*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *NewInstanceNode* (*ir-nid*: *ID*) (*ir-instanceClass*: *string*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *NotNode* (*ir-value*: *INPUT*)
| *OrNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *ParameterNode* (*ir-index*: *nat*)
| *PiNode* (*ir-object*: *INPUT*) (*ir-guard-opt*: *INPUT-GUARD option*)
| *ReturnNode* (*ir-result-opt*: *INPUT option*) (*ir-memoryMap-opt*: *INPUT-EXT option*)
| *RightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *ShortCircuitOrNode* (*ir-x*: *INPUT-COND*) (*ir-y*: *INPUT-COND*)
| *SignExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
| *SignedDivNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *INPUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

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

| *StartNode* (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *StoreFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-value*: *INPUT*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)
| *SubNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *UnsignedRightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *UnwindNode* (*ir-exception*: *INPUT*)
| *ValuePhiNode* (*ir-nid*: *ID*) (*ir-values*: *INPUT list*) (*ir-merge*: *INPUT-ASSOC*)
| *ValueProxyNode* (*ir-value*: *INPUT*) (*ir-loopExit*: *INPUT-ASSOC*)
| *XorNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *ZeroExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
| *NoNode*

| *RefNode* (*ir-ref*:*ID*)

fun *opt-to-list* :: $'a$ *option* $\Rightarrow$ $'a$ *list* **where**
opt-to-list None = [] |

opt-to-list (*Some* *v*) = [*v*]

fun *opt-list-to-list* :: *'a list option* ⇒ *'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*, *trueValue*, *falseValue*] |
inputs-of-ConstantNode:
inputs-of (*ConstantNode const*) = [] |
inputs-of-DynamicNewArrayNode:
inputs-of (*DynamicNewArrayNode elementType length0 voidClass stateBefore next*) = [*elementType*, *length0*] @ (*opt-to-list voidClass*) @ (*opt-to-list stateBefore*) |
inputs-of-EndNode:
inputs-of (*EndNode*) = [] |
inputs-of-ExceptionObjectNode:
inputs-of (*ExceptionObjectNode stateAfter next*) = (*opt-to-list stateAfter*) |
inputs-of-FrameState:
inputs-of (*FrameState monitorIds outerFrameState values virtualObjectMappings*) = *monitorIds* @ (*opt-to-list outerFrameState*) @ (*opt-list-to-list values*) @ (*opt-list-to-list virtualObjectMappings*) |
inputs-of-IfNode:
inputs-of (*IfNode condition trueSuccessor falseSuccessor*) = [*condition*] |
inputs-of-IntegerBelowNode:
inputs-of (*IntegerBelowNode x y*) = [*x*, *y*] |
inputs-of-IntegerEqualsNode:
inputs-of (*IntegerEqualsNode x y*) = [*x*, *y*] |
inputs-of-IntegerLessThanNode:
inputs-of (*IntegerLessThanNode x y*) = [*x*, *y*] |
inputs-of-InvokeNode:
inputs-of (*InvokeNode nid0 callTarget classInit stateDuring stateAfter next*) = *callTarget* # (*opt-to-list classInit*) @ (*opt-to-list stateDuring*) @ (*opt-to-list*

stateAfter) |
inputs-of-InvokeWithExceptionNode:
inputs-of (*InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter next exceptionEdge*) = *callTarget* # (*opt-to-list classInit*) @ (*opt-to-list stateDuring*) @ (*opt-to-list stateAfter*) |
inputs-of-IsNullNode:
inputs-of (*IsNullNode value*) = [*value*] |
inputs-of-KillingBeginNode:
inputs-of (*KillingBeginNode next*) = [] |
inputs-of-LeftShiftNode:
inputs-of (*LeftShiftNode x y*) = [*x*, *y*] |
inputs-of-LoadFieldNode:
inputs-of (*LoadFieldNode nid0 field object next*) = (*opt-to-list object*) |
inputs-of-LogicNegationNode:
inputs-of (*LogicNegationNode value*) = [*value*] |
inputs-of-LoopBeginNode:
inputs-of (*LoopBeginNode ends overflowGuard stateAfter next*) = *ends* @ (*opt-to-list overflowGuard*) @ (*opt-to-list stateAfter*) |
inputs-of-LoopEndNode:
inputs-of (*LoopEndNode loopBegin*) = [*loopBegin*] |
inputs-of-LoopExitNode:
inputs-of (*LoopExitNode loopBegin stateAfter next*) = *loopBegin* # (*opt-to-list stateAfter*) |
inputs-of-MergeNode:
inputs-of (*MergeNode ends stateAfter next*) = *ends* @ (*opt-to-list stateAfter*) |
inputs-of-MethodCallTargetNode:
inputs-of (*MethodCallTargetNode targetMethod arguments*) = *arguments* |
inputs-of-MulNode:
inputs-of (*MulNode x y*) = [*x*, *y*] |
inputs-of-NarrowNode:
inputs-of (*NarrowNode inputBits resultBits value*) = [*value*] |
inputs-of-NegateNode:
inputs-of (*NegateNode value*) = [*value*] |
inputs-of-NewArrayNode:
inputs-of (*NewArrayNode length0 stateBefore next*) = *length0* # (*opt-to-list stateBefore*) |
inputs-of-NewInstanceNode:
inputs-of (*NewInstanceNode nid0 instanceClass stateBefore next*) = (*opt-to-list stateBefore*) |
inputs-of-NotNode:
inputs-of (*NotNode value*) = [*value*] |
inputs-of-OrNode:
inputs-of (*OrNode x y*) = [*x*, *y*] |
inputs-of-ParameterNode:
inputs-of (*ParameterNode index*) = [] |
inputs-of-PiNode:
inputs-of (*PiNode object guard*) = *object* # (*opt-to-list guard*) |
inputs-of-ReturnNode:
inputs-of (*ReturnNode result memoryMap*) = (*opt-to-list result*) @ (*opt-to-list*

memoryMap) |
inputs-of-RightShiftNode:
inputs-of (*RightShiftNode x y*) = [*x*, *y*] |
inputs-of-ShortCircuitOrNode:
inputs-of (*ShortCircuitOrNode x y*) = [*x*, *y*] |
inputs-of-SignExtendNode:
inputs-of (*SignExtendNode inputBits resultBits value*) = [*value*] |
inputs-of-SignedDivNode:
inputs-of (*SignedDivNode nid0 x y zeroCheck stateBefore next*) = [*x*, *y*] @ (*opt-to-list zeroCheck*) @ (*opt-to-list stateBefore*) |
inputs-of-SignedRemNode:
inputs-of (*SignedRemNode nid0 x y zeroCheck stateBefore next*) = [*x*, *y*] @ (*opt-to-list zeroCheck*) @ (*opt-to-list stateBefore*) |
inputs-of-StartNode:
inputs-of (*StartNode stateAfter next*) = (*opt-to-list stateAfter*) |
inputs-of-StoreFieldNode:
inputs-of (*StoreFieldNode nid0 field value stateAfter object next*) = *value* # (*opt-to-list stateAfter*) @ (*opt-to-list object*) |
inputs-of-SubNode:
inputs-of (*SubNode x y*) = [*x*, *y*] |
inputs-of-UnsignedRightShiftNode:
inputs-of (*UnsignedRightShiftNode x y*) = [*x*, *y*] |
inputs-of-UnwindNode:
inputs-of (*UnwindNode exception*) = [*exception*] |
inputs-of-ValuePhiNode:
inputs-of (*ValuePhiNode nid0 values merge*) = *merge* # *values* |
inputs-of-ValueProxyNode:
inputs-of (*ValueProxyNode value loopExit*) = [*value*, *loopExit*] |
inputs-of-XorNode:
inputs-of (*XorNode x y*) = [*x*, *y*] |
inputs-of-ZeroExtendNode:
inputs-of (*ZeroExtendNode inputBits resultBits value*) = [*value*] |
inputs-of-NoNode: *inputs-of* (*NoNode*) = [] |

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

fun *successors-of* :: *IRNode* ⇒ *ID list* **where**
successors-of-AbsNode:
successors-of (*AbsNode value*) = [] |
successors-of-AddNode:
successors-of (*AddNode x y*) = [] |
successors-of-AndNode:
successors-of (*AndNode x y*) = [] |
successors-of-BeginNode:
successors-of (*BeginNode next*) = [*next*] |
successors-of-BytecodeExceptionNode:
successors-of (*BytecodeExceptionNode arguments stateAfter next*) = [*next*] |

successors-of-ConditionalNode:
successors-of (*ConditionalNode condition trueValue falseValue*) = [] |
successors-of-ConstantNode:
successors-of (*ConstantNode const*) = [] |
successors-of-DynamicNewArrayNode:
successors-of (*DynamicNewArrayNode elementType length0 voidClass stateBefore next*) = [*next*] |
successors-of-EndNode:
successors-of (*EndNode*) = [] |
successors-of-ExceptionObjectNode:
successors-of (*ExceptionObjectNode stateAfter next*) = [*next*] |
successors-of-FrameState:
successors-of (*FrameState monitorIds outerFrameState values virtualObjectMappings*) = [] |
successors-of-IfNode:
successors-of (*IfNode condition trueSuccessor falseSuccessor*) = [*trueSuccessor*, *falseSuccessor*] |
successors-of-IntegerBelowNode:
successors-of (*IntegerBelowNode x y*) = [] |
successors-of-IntegerEqualsNode:
successors-of (*IntegerEqualsNode x y*) = [] |
successors-of-IntegerLessThanNode:
successors-of (*IntegerLessThanNode x y*) = [] |
successors-of-InvokeNode:
successors-of (*InvokeNode nid0 callTarget classInit stateDuring stateAfter next*) = [*next*] |
successors-of-InvokeWithExceptionNode:
successors-of (*InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter next exceptionEdge*) = [*next*, *exceptionEdge*] |
successors-of-IsNullNode:
successors-of (*IsNullNode value*) = [] |
successors-of-KillingBeginNode:
successors-of (*KillingBeginNode next*) = [*next*] |
successors-of-LeftShiftNode:
successors-of (*LeftShiftNode x y*) = [] |
successors-of-LoadFieldNode:
successors-of (*LoadFieldNode nid0 field object next*) = [*next*] |
successors-of-LogicNegationNode:
successors-of (*LogicNegationNode value*) = [] |
successors-of-LoopBeginNode:
successors-of (*LoopBeginNode ends overflowGuard stateAfter next*) = [*next*] |
successors-of-LoopEndNode:
successors-of (*LoopEndNode loopBegin*) = [] |
successors-of-LoopExitNode:
successors-of (*LoopExitNode loopBegin stateAfter next*) = [*next*] |
successors-of-MergeNode:
successors-of (*MergeNode ends stateAfter next*) = [*next*] |
successors-of-MethodCallTargetNode:
successors-of (*MethodCallTargetNode targetMethod arguments*) = [] |

successors-of-MulNode:
successors-of (*MulNode x y*) = [] |
successors-of-NarrowNode:
successors-of (*NarrowNode inputBits resultBits value*) = [] |
successors-of-NegateNode:
successors-of (*NegateNode value*) = [] |
successors-of-NewArrayNode:
successors-of (*NewArrayNode length0 stateBefore next*) = [*next*] |
successors-of-NewInstanceNode:
successors-of (*NewInstanceNode nid0 instanceClass stateBefore next*) = [*next*] |
successors-of-NotNode:
successors-of (*NotNode value*) = [] |
successors-of-OrNode:
successors-of (*OrNode x y*) = [] |
successors-of-ParameterNode:
successors-of (*ParameterNode index*) = [] |
successors-of-PiNode:
successors-of (*PiNode object guard*) = [] |
successors-of-ReturnNode:
successors-of (*ReturnNode result memoryMap*) = [] |
successors-of-RightShiftNode:
successors-of (*RightShiftNode x y*) = [] |
successors-of-ShortCircuitOrNode:
successors-of (*ShortCircuitOrNode x y*) = [] |
successors-of-SignExtendNode:
successors-of (*SignExtendNode inputBits resultBits value*) = [] |
successors-of-SignedDivNode:
successors-of (*SignedDivNode nid0 x y zeroCheck stateBefore next*) = [*next*] |
successors-of-SignedRemNode:
successors-of (*SignedRemNode nid0 x y zeroCheck stateBefore next*) = [*next*] |
successors-of-StartNode:
successors-of (*StartNode stateAfter next*) = [*next*] |
successors-of-StoreFieldNode:
successors-of (*StoreFieldNode nid0 field value stateAfter object next*) = [*next*] |
successors-of-SubNode:
successors-of (*SubNode x y*) = [] |
successors-of-UnsignedRightShiftNode:
successors-of (*UnsignedRightShiftNode x y*) = [] |
successors-of-UnwindNode:
successors-of (*UnwindNode exception*) = [] |
successors-of-ValuePhiNode:
successors-of (*ValuePhiNode nid0 values merge*) = [] |
successors-of-ValueProxyNode:
successors-of (*ValueProxyNode value loopExit*) = [] |
successors-of-XorNode:
successors-of (*XorNode x y*) = [] |
successors-of-ZeroExtendNode:
successors-of (*ZeroExtendNode inputBits resultBits value*) = [] |
successors-of-NoNode: *successors-of* (*NoNode*) = [] |

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

lemma *inputs-of* (*FrameState x* (*Some y*) (*Some z*) *None*) = *x* @ [*y*] @ *z*
⟨*proof*⟩
lemma *successors-of* (*FrameState x* (*Some y*) (*Some z*) *None*) = []
⟨*proof*⟩

lemma *inputs-of* (*IfNode c t f*) = [*c*]
⟨*proof*⟩
lemma *successors-of* (*IfNode c t f*) = [*t*, *f*]
⟨*proof*⟩

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

end

2.2 Hierarchy of Nodes

theory *IRNodeHierarchy*
imports *IRNodes*
begin

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

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

These functions have been automatically generated from the compiler.

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

fun *is-VirtualState* :: *IRNode* ⇒ *bool* **where**
is-VirtualState n = ((*is-FrameState 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-ShiftNode* :: *IRNode* ⇒ *bool* **where**
is-ShiftNode n = ((*is-LeftShiftNode* n) ∨ (*is-RightShiftNode* n) ∨ (*is-UnsignedRightShiftNode* n))

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

fun *is-AbstractLocalNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractLocalNode n = ((*is-ParameterNode* n))

fun *is-IntegerConvertNode* :: *IRNode* ⇒ *bool* **where**
is-IntegerConvertNode n = ((*is-NarrowNode* n) ∨ (*is-SignExtendNode* n) ∨ (*is-ZeroExtendNode* 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-IntegerConvertNode* n) ∨ (*is-UnaryArithmeticNode* n))

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

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

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

fun *is-IntegerLowerThanNode* :: *IRNode* ⇒ *bool* **where**
is-IntegerLowerThanNode n = ((*is-IntegerBelowNode* n) ∨ (*is-IntegerLessThanNode* 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-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-FloatingNode* :: *IRNode* ⇒ *bool* **where**
is-FloatingNode n = ((*is-AbstractLocalNode* n) ∨ (*is-BinaryNode* n) ∨ (*is-ConditionalNode*

n) ∨ (is-ConstantNode n) ∨ (is-FloatingGuardedNode n) ∨ (is-LogicNode n) ∨ (is-PhiNode n) ∨ (is-ProxyNode n) ∨ (is-UnaryNode n))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$n) \lor (\textit{is-UnaryOpLogicNode}\ n))$

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

fun *is-BinaryCommutative* :: *IRNode* $\Rightarrow$ *bool* **where**
$\textit{is-BinaryCommutative}\ n = ((\textit{is-AddNode}\ n) \lor (\textit{is-AndNode}\ n) \lor (\textit{is-IntegerEqualsNode}\ n) \lor (\textit{is-MulNode}\ n) \lor (\textit{is-OrNode}\ n) \lor (\textit{is-XorNode}\ n))$

fun *is-Canonicalizable* :: *IRNode* $\Rightarrow$ *bool* **where**
$\textit{is-Canonicalizable}\ n = ((\textit{is-BytecodeExceptionNode}\ n) \lor (\textit{is-ConditionalNode}\ n) \lor (\textit{is-DynamicNewArrayNode}\ n) \lor (\textit{is-PhiNode}\ n) \lor (\textit{is-PiNode}\ n) \lor (\textit{is-ProxyNode}\ n) \lor (\textit{is-StoreFieldNode}\ n) \lor (\textit{is-ValueProxyNode}\ n))$

fun *is-UncheckedInterfaceProvider* :: *IRNode* $\Rightarrow$ *bool* **where**
$\textit{is-UncheckedInterfaceProvider}\ n = ((\textit{is-InvokeNode}\ n) \lor (\textit{is-InvokeWithExceptionNode}\ n) \lor (\textit{is-LoadFieldNode}\ n) \lor (\textit{is-ParameterNode}\ n))$

fun *is-Binary* :: *IRNode* $\Rightarrow$ *bool* **where**
$\textit{is-Binary}\ n = ((\textit{is-BinaryArithmeticNode}\ n) \lor (\textit{is-BinaryNode}\ n) \lor (\textit{is-BinaryOpLogicNode}\ n) \lor (\textit{is-CompareNode}\ n) \lor (\textit{is-FixedBinaryNode}\ n) \lor (\textit{is-ShortCircuitOrNode}\ n))$

fun *is-ArithmeticOperation* :: *IRNode* $\Rightarrow$ *bool* **where**
$\textit{is-ArithmeticOperation}\ n = ((\textit{is-BinaryArithmeticNode}\ n) \lor (\textit{is-IntegerConvertNode}\ n) \lor (\textit{is-ShiftNode}\ n) \lor (\textit{is-UnaryArithmeticNode}\ n))$

fun *is-ValueNumberable* :: *IRNode* $\Rightarrow$ *bool* **where**
$\textit{is-ValueNumberable}\ n = ((\textit{is-FloatingNode}\ n) \lor (\textit{is-ProxyNode}\ n))$

fun *is-Lowerable* :: *IRNode* $\Rightarrow$ *bool* **where**
$\textit{is-Lowerable}\ n = ((\textit{is-AbstractNewObjectNode}\ n) \lor (\textit{is-AccessFieldNode}\ n) \lor (\textit{is-BytecodeExceptionNode}\ n) \lor (\textit{is-ExceptionObjectNode}\ n) \lor (\textit{is-IntegerDivRemNode}\ n) \lor (\textit{is-UnwindNode}\ n))$

fun *is-Virtualizable* :: *IRNode* $\Rightarrow$ *bool* **where**
$\textit{is-Virtualizable}\ n = ((\textit{is-IsNullNode}\ n) \lor (\textit{is-LoadFieldNode}\ n) \lor (\textit{is-PiNode}\ n) \lor (\textit{is-StoreFieldNode}\ n) \lor (\textit{is-ValueProxyNode}\ n))$

fun *is-Simplifiable* :: *IRNode* $\Rightarrow$ *bool* **where**
$\textit{is-Simplifiable}\ n = ((\textit{is-AbstractMergeNode}\ n) \lor (\textit{is-BeginNode}\ n) \lor (\textit{is-IfNode}\ n) \lor (\textit{is-LoopExitNode}\ n) \lor (\textit{is-MethodCallTargetNode}\ n) \lor (\textit{is-NewArrayNode}\ n))$

fun *is-StateSplit* :: *IRNode* $\Rightarrow$ *bool* **where**
$\textit{is-StateSplit}\ n = ((\textit{is-AbstractStateSplit}\ n) \lor (\textit{is-BeginStateSplitNode}\ n) \lor (\textit{is-StoreFieldNode}\ n))$

fun *is-ConvertNode* :: *IRNode* $\Rightarrow$ *bool* **where**
$\textit{is-ConvertNode}\ n = ((\textit{is-IntegerConvertNode}\ n))$

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

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

fun *is-same-ir-node-type* :: *IRNode* ⇒ *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-KillingBeginNode n1*) ∧ (*is-KillingBeginNode n2*)) ∨
((*is-LoadFieldNode n1*) ∧ (*is-LoadFieldNode n2*)) ∨
((*is-LogicNegationNode n1*) ∧ (*is-LogicNegationNode n2*)) ∨
((*is-LoopBeginNode n1*) ∧ (*is-LoopBeginNode n2*)) ∨
((*is-LoopEndNode n1*) ∧ (*is-LoopEndNode n2*)) ∨
((*is-LoopExitNode n1*) ∧ (*is-LoopExitNode n2*)) ∨
((*is-MergeNode n1*) ∧ (*is-MergeNode n2*)) ∨
((*is-MethodCallTargetNode n1*) ∧ (*is-MethodCallTargetNode n2*)) ∨
((*is-MulNode n1*) ∧ (*is-MulNode n2*)) ∨
((*is-NegateNode n1*) ∧ (*is-NegateNode n2*)) ∨
((*is-NewArrayNode n1*) ∧ (*is-NewArrayNode n2*)) ∨
((*is-NewInstanceNode n1*) ∧ (*is-NewInstanceNode n2*)) ∨
((*is-NotNode n1*) ∧ (*is-NotNode n2*)) ∨
((*is-OrNode n1*) ∧ (*is-OrNode n2*)) ∨
((*is-ParameterNode n1*) ∧ (*is-ParameterNode n2*)) ∨

$((is\text{-}PiNode\ n1) \wedge (is\text{-}PiNode\ n2)) \vee$
$((is\text{-}ReturnNode\ n1) \wedge (is\text{-}ReturnNode\ n2)) \vee$
$((is\text{-}ShortCircuitOrNode\ n1) \wedge (is\text{-}ShortCircuitOrNode\ n2)) \vee$
$((is\text{-}SignedDivNode\ n1) \wedge (is\text{-}SignedDivNode\ n2)) \vee$
$((is\text{-}StartNode\ n1) \wedge (is\text{-}StartNode\ n2)) \vee$
$((is\text{-}StoreFieldNode\ n1) \wedge (is\text{-}StoreFieldNode\ n2)) \vee$
$((is\text{-}SubNode\ n1) \wedge (is\text{-}SubNode\ n2)) \vee$
$((is\text{-}UnwindNode\ n1) \wedge (is\text{-}UnwindNode\ n2)) \vee$
$((is\text{-}ValuePhiNode\ n1) \wedge (is\text{-}ValuePhiNode\ n2)) \vee$
$((is\text{-}ValueProxyNode\ n1) \wedge (is\text{-}ValueProxyNode\ n2)) \vee$
$((is\text{-}XorNode\ n1) \wedge (is\text{-}XorNode\ n2)))$

end

3 Stamp Typing

theory *Stamp*
imports *Values*
begin

The GraalVM compiler uses the Stamp class to store range and type information for a given node in the IR graph. We model the Stamp class as a datatype, Stamp, and provide a number of functions on the datatype which correspond to the class methods within the compiler.

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

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

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

fun *bit-bounds* :: $nat \Rightarrow (int \times int)$ **where**
$bit\text{-}bounds\ bits = (((2\ \hat{}\ bits)\ div\ 2) * -1, ((2\ \hat{}\ bits)\ div\ 2) - 1)$

— A stamp which includes the full range of the type
fun *unrestricted-stamp* :: $Stamp \Rightarrow Stamp$ **where**
$unrestricted\text{-}stamp\ VoidStamp = VoidStamp\ |$

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* ⇒ *Value* ⇒ *bool* **where**
valid-value (*IntegerStamp b l h*) (*IntVal32 v*) = (*b*=*32* ∧ (*sint v* ≥ *l*) ∧ (*sint v* ≤ *h*)) |
valid-value (*IntegerStamp b l h*) (*IntVal64 v*) = (*b*=*64* ∧ (*sint v* ≥ *l*) ∧ (*sint v* ≤ *h*)) |

valid-value (*VoidStamp*) (*UndefVal*) = *True* |
valid-value (*ObjectStamp klass exact nonNull alwaysNull*) (*ObjRef ref*) =
(*if nonNull then ref*≠*None else True*) |
valid-value stamp val = *False*

— The most common type of stamp within the compiler (apart from the VoidStamp) is a 32 bit integer stamp with an unrestricted range. We use *default-stamp* as it is a frequently used stamp.
definition *default-stamp* :: *Stamp* **where**
default-stamp = (*unrestricted-stamp* (*IntegerStamp 32 0 0*))

end

4 Graph Representation

theory *IRGraph*
imports
IRNodeHierarchy
Stamp
HOL−Library.FSet
HOL.Relation
begin

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

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

typedef *IRGraph* = {*g* :: *ID* ⇀ (*IRNode* × *Stamp*) . *finite* (*dom g*)}
⟨*proof*⟩

setup-lifting *type-definition-IRGraph*

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

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

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

lift-definition $\mathit{stamp} :: \mathit{IRGraph} \Rightarrow \mathit{ID} \Rightarrow \mathit{Stamp}$
is $\mathit{with\text{-}default}\ \mathit{IllegalStamp}\ \mathit{snd}$ $\langle \mathit{proof} \rangle$

lift-definition $\mathit{add\text{-}node} :: \mathit{ID} \Rightarrow (\mathit{IRNode} \times \mathit{Stamp}) \Rightarrow \mathit{IRGraph} \Rightarrow \mathit{IRGraph}$
is $\lambda \mathit{nid}\ k\ g.\ \mathit{if}\ \mathit{fst}\ k = \mathit{NoNode}\ \mathit{then}\ g\ \mathit{else}\ g(\mathit{nid} \mapsto k)$ $\langle \mathit{proof} \rangle$

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

lift-definition $\mathit{replace\text{-}node} :: \mathit{ID} \Rightarrow (\mathit{IRNode} \times \mathit{Stamp}) \Rightarrow \mathit{IRGraph} \Rightarrow \mathit{IRGraph}$
is $\lambda \mathit{nid}\ k\ g.\ \mathit{if}\ \mathit{fst}\ k = \mathit{NoNode}\ \mathit{then}\ g\ \mathit{else}\ g(\mathit{nid} \mapsto k)$ $\langle \mathit{proof} \rangle$

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

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

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

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

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

notation (*latex*)
$\mathit{domain\text{-}subtraction}$ (- $\lessdot$ -)

code-datatype *irgraph*

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

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

lemma *dom-eq*:
  assumes $\forall\, x \in set\ xs.\ fst\ (snd\ x) \neq NoNode$
  shows $filter\text{-}none\ (map\text{-}of\ xs) = dom\ (map\text{-}of\ xs)$
  ⟨*proof*⟩

lemma *fil-eq*:
  $filter\text{-}none\ (map\text{-}of\ (no\text{-}node\ xs)) = set\ (map\ fst\ (no\text{-}node\ xs))$
  ⟨*proof*⟩

lemma *irgraph*[*code*]: $ids\ (irgraph\ m) = set\ (map\ fst\ (no\text{-}node\ m))$
  ⟨*proof*⟩

lemma [*code*]: $Rep\text{-}IRGraph\ (irgraph\ m) = map\text{-}of\ (no\text{-}node\ m)$
  ⟨*proof*⟩
fun *inputs* :: $IRGraph \Rightarrow ID \Rightarrow ID\ set$ **where**
  $inputs\ g\ nid = set\ (inputs\text{-}of\ (kind\ g\ nid))$
— Get the successor set of a given node ID
fun *succ* :: $IRGraph \Rightarrow ID \Rightarrow ID\ set$ **where**
  $succ\ g\ nid = set\ (successors\text{-}of\ (kind\ g\ nid))$
— Gives a relation between node IDs - between a node and its input nodes
fun *input-edges* :: $IRGraph \Rightarrow ID\ rel$ **where**
  $input\text{-}edges\ g = (\bigcup\ i \in ids\ g.\ \{(i,j)|j.\ j \in (inputs\ g\ i)\})$
— 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**
  $usages\ g\ nid = \{j.\ j \in ids\ g \wedge (j,nid) \in input\text{-}edges\ g\}$
fun *successor-edges* :: $IRGraph \Rightarrow ID\ rel$ **where**
  $successor\text{-}edges\ g = (\bigcup\ i \in ids\ g.\ \{(i,j)|j\ .\ j \in (succ\ \ g\ i)\})$
fun *predecessors* :: $IRGraph \Rightarrow ID \Rightarrow ID\ set$ **where**
  $predecessors\ g\ nid = \{j.\ j \in ids\ g \wedge (j,nid) \in successor\text{-}edges\ g\}$
fun *nodes-of* :: $IRGraph \Rightarrow (IRNode \Rightarrow bool) \Rightarrow ID\ set$ **where**
  $nodes\text{-}of\ g\ sel = \{nid \in ids\ g\ .\ sel\ (kind\ g\ nid)\}$
fun *edge* :: $(IRNode \Rightarrow 'a) \Rightarrow ID \Rightarrow IRGraph \Rightarrow 'a$ **where**
  $edge\ sel\ nid\ g = sel\ (kind\ g\ nid)$

fun *filtered-inputs* :: $IRGraph \Rightarrow ID \Rightarrow (IRNode \Rightarrow bool) \Rightarrow ID\ list$ **where**
  $filtered\text{-}inputs\ g\ nid\ f = filter\ (f \circ (kind\ g))\ (inputs\text{-}of\ (kind\ g\ nid))$
fun *filtered-successors* :: $IRGraph \Rightarrow ID \Rightarrow (IRNode \Rightarrow bool) \Rightarrow ID\ list$ **where**
  $filtered\text{-}successors\ g\ nid\ f = filter\ (f \circ (kind\ g))\ (successors\text{-}of\ (kind\ g\ nid))$
fun *filtered-usages* :: $IRGraph \Rightarrow ID \Rightarrow (IRNode \Rightarrow bool) \Rightarrow ID\ set$ **where**
  $filtered\text{-}usages\ g\ nid\ f = \{n \in (usages\ g\ nid).\ f\ (kind\ g\ n)\}$

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

fun *any-usage* :: $IRGraph \Rightarrow ID \Rightarrow ID$ **where**

$any\text{-}usage\ g\ nid = hd\ (sorted\text{-}list\text{-}of\text{-}set\ (usages\ g\ nid))$

lemma *ids-some*[*simp*]: $x \in ids\ g \longleftrightarrow kind\ g\ x \neq NoNode$
⟨*proof*⟩

lemma *not-in-g*:
assumes $nid \notin ids\ g$
shows $kind\ g\ nid = NoNode$
⟨*proof*⟩

lemma *valid-creation*[*simp*]:
$finite\ (dom\ g) \longleftrightarrow Rep\text{-}IRGraph\ (Abs\text{-}IRGraph\ g) = g$
⟨*proof*⟩

lemma [*simp*]: $finite\ (ids\ g)$
⟨*proof*⟩

lemma [*simp*]: $finite\ (ids\ (irgraph\ g))$
⟨*proof*⟩

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

lemma [*simp*]: $finite\ (dom\ g) \longrightarrow kind\ (Abs\text{-}IRGraph\ g) = (\lambda x\ .\ (case\ g\ x\ of\ None \Rightarrow NoNode \mid Some\ n \Rightarrow fst\ n))$
⟨*proof*⟩

lemma [*simp*]: $finite\ (dom\ g) \longrightarrow stamp\ (Abs\text{-}IRGraph\ g) = (\lambda x\ .\ (case\ g\ x\ of\ None \Rightarrow IllegalStamp \mid Some\ n \Rightarrow snd\ n))$
⟨*proof*⟩

lemma [*simp*]: $ids\ (irgraph\ g) = set\ (map\ fst\ (no\text{-}node\ g))$
⟨*proof*⟩

lemma [*simp*]: $kind\ (irgraph\ g) = (\lambda nid.\ (case\ (map\text{-}of\ (no\text{-}node\ g))\ nid\ of\ None \Rightarrow NoNode \mid Some\ n \Rightarrow fst\ n))$
⟨*proof*⟩

lemma [*simp*]: $stamp\ (irgraph\ g) = (\lambda nid.\ (case\ (map\text{-}of\ (no\text{-}node\ g))\ nid\ of\ None \Rightarrow IllegalStamp \mid Some\ n \Rightarrow snd\ n))$
⟨*proof*⟩

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

lemma [*code*]: $replace\text{-}node\ nid\ k\ (irgraph\ g) = (irgraph\ (\ ((nid,\ k)\ \#\ g)))$
⟨*proof*⟩

lemma [*code*]: *add-node nid k* (*irgraph g*) = (*irgraph* (((*nid*, *k*) # *g*)))
⟨*proof*⟩

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

lemma *remove-node-lookup*:
gup = *remove-node nid g* ⟶ *kind gup nid* = *NoNode* ∧ *stamp gup nid* = *IllegalStamp*
⟨*proof*⟩

lemma *replace-node-lookup*[*simp*]:
gup = *replace-node nid* (*k*, *s*) *g* ∧ *k* ≠ *NoNode* ⟶ *kind gup nid* = *k* ∧ *stamp gup nid* = *s*
⟨*proof*⟩

lemma *replace-node-unchanged*:
gup = *replace-node nid* (*k*, *s*) *g* ⟶ (∀ *n* ∈ (*ids g* − {*nid*}) . *n* ∈ *ids g* ∧ *n* ∈ *ids gup* ∧ *kind g n* = *kind gup n*)
⟨*proof*⟩

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.Values
    Graph.Stamp
    HOL−Library.Word
begin

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

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

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

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

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

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

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

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

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

5.1 Data-flow Tree Representation

datatype *IRUnaryOp* =
    UnaryAbs

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

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

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

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

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

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

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

typedef *GroundExpr* = { *e* :: *IRExpr* . *is-ground e* }
⟨*proof*⟩

5.2 Data-flow Tree Evaluation

fun *unary-eval* :: *IRUnaryOp* ⇒ *Value* ⇒ *Value* **where**
unary-eval UnaryAbs v = *intval-abs v* |
unary-eval UnaryNeg v = *intval-negate v* |
unary-eval UnaryNot v = *intval-not v* |
unary-eval UnaryLogicNegation (*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 BinLeftShift v1 v2 = *intval-left-shift v1 v2* |
bin-eval BinRightShift v1 v2 = *intval-right-shift v1 v2* |
bin-eval BinURightShift v1 v2 = *intval-uright-shift v1 v2* |
bin-eval BinIntegerEquals v1 v2 = *intval-equals v1 v2* |
bin-eval BinIntegerLessThan v1 v2 = *intval-less-than v1 v2* |
bin-eval BinIntegerBelow v1 v2 = *intval-below v1 v2*

inductive *not-undef-or-fail* :: *Value* ⇒ *Value* ⇒ *bool* **where**
⟦*value* ≠ *UndefVal*⟧ ⟹ *not-undef-or-fail value value*

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

inductive
evaltree :: *MapState* ⇒ *Params* ⇒ *IRExpr* ⇒ *Value* ⇒ *bool* ([-,-] ⊢ - ↦ - *55*)
for *m p* **where**

ConstantExpr:
⟦*valid-value* (*constantAsStamp c*) *c*⟧
⟹ [*m*,*p*] ⊢ (*ConstantExpr c*) ↦ *c* |

ParameterExpr:
⟦*i* < *length p*; *valid-value s* (*p*!*i*)⟧
⟹ [*m*,*p*] ⊢ (*ParameterExpr i s*) ↦ *p*!*i* |

ConditionalExpr:
⟦[*m*,*p*] ⊢ *ce* ↦ *cond*;
branch = (*if val-to-bool cond then te else fe*);
[*m*,*p*] ⊢ *branch* ↦ *v*;
v ≠ *UndefVal*⟧
⟹ [*m*,*p*] ⊢ (*ConditionalExpr ce te fe*) ↦ *v* |

UnaryExpr:
$[\![[m,p] \vdash xe \mapsto v;$
$result = (unary\text{-}eval\ op\ v);$
$result \neq UndefVal]\!]$
$\Longrightarrow [m,p] \vdash (UnaryExpr\ op\ xe) \mapsto result$ |

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

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

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

$$\frac{i < |p| \qquad valid\text{-}value\ s\ p_{[i]}}{[m,p] \vdash ParameterExpr\ i\ s \mapsto p_{[i]}}$$

$$\frac{[m,p] \vdash ce \mapsto cond \qquad branch = (\textsf{if}\ IRTreeEval.val\text{-}to\text{-}bool\ cond\ \textsf{then}\ te\ \textsf{else}\ fe) \qquad [m,p] \vdash branch \mapsto v \qquad v \neq UndefVal}{[m,p] \vdash ConditionalExpr\ ce\ te\ fe \mapsto v}$$

$$\frac{[m,p] \vdash xe \mapsto v \qquad result = unary\text{-}eval\ op\ v \qquad result \neq UndefVal}{[m,p] \vdash UnaryExpr\ op\ xe \mapsto result}$$

$$\frac{[m,p] \vdash xe \mapsto x \qquad [m,p] \vdash ye \mapsto y \qquad result = bin\text{-}eval\ op\ x\ y \qquad result \neq UndefVal}{[m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto result}$$

$$\frac{val = m\ n \qquad valid\text{-}value\ s\ val}{[m,p] \vdash LeafExpr\ n\ s \mapsto val}$$

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

inductive
evaltrees :: *MapState* ⇒ *Params* ⇒ *IRExpr list* ⇒ *Value list* ⇒ *bool* ($[\text{-},\text{-}] \vdash \text{-} \mapsto_L \text{-}$ *55*)
for *m p* **where**

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

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

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$ *as evalTs*)
evaltrees ⟨*proof*⟩

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* ⇒ *IRExpr* ⇒ *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*
⟨*proof*⟩

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

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

instance ⟨*proof*⟩

end

end

6 Data-flow Expression-Tree Theorems

theory *IRTreeEvalThms*
imports
TreeToGraph
HOL−Eisbach.Eisbach
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.

named-theorems *rep*

lemma *rep-constant* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = ConstantNode\ c \Longrightarrow$
$e = ConstantExpr\ c$
$\langle proof \rangle$

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

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

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

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

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

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

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

lemma *rep-sub* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = SubNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinSub\ xe\ ye)$
$\langle proof \rangle$

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

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

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

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

lemma *rep-integer-below* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IntegerBelowNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinIntegerBelow\ xe\ ye)$
$\langle proof \rangle$

lemma *rep-integer-equals* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IntegerEqualsNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinIntegerEquals\ xe\ ye)$
$\langle proof \rangle$

lemma *rep-integer-less-than* [*rep*]:
  $g \vdash n \simeq e \Longrightarrow$
   kind g n = *IntegerLessThanNode x y* $\Longrightarrow$
   ($\exists$ *xe ye*. *e* = *BinaryExpr BinIntegerLessThan xe ye*)
  ⟨*proof*⟩

lemma *rep-narrow* [*rep*]:
  $g \vdash n \simeq e \Longrightarrow$
   kind g n = *NarrowNode ib rb x* $\Longrightarrow$
   ($\exists$ *x*. *e* = *UnaryExpr* (*UnaryNarrow ib rb*) *x*)
  ⟨*proof*⟩

lemma *rep-sign-extend* [*rep*]:
  $g \vdash n \simeq e \Longrightarrow$
   kind g n = *SignExtendNode ib rb x* $\Longrightarrow$
   ($\exists$ *x*. *e* = *UnaryExpr* (*UnarySignExtend ib rb*) *x*)
  ⟨*proof*⟩

lemma *rep-zero-extend* [*rep*]:
  $g \vdash n \simeq e \Longrightarrow$
   kind g n = *ZeroExtendNode ib rb x* $\Longrightarrow$
   ($\exists$ *x*. *e* = *UnaryExpr* (*UnaryZeroExtend ib rb*) *x*)
  ⟨*proof*⟩

lemma *rep-load-field* [*rep*]:
  $g \vdash n \simeq e \Longrightarrow$
   is-preevaluated (*kind g n*) $\Longrightarrow$
   ($\exists$ *s*. *e* = *LeafExpr n s*)
  ⟨*proof*⟩

method *solve-det* **uses** *node* =
  (*match node* **in** *kind* - - = *node* - **for** *node* ⇒
    ‹*match rep in r*: - $\Longrightarrow$ - = *node* - $\Longrightarrow$ - ⇒
      ‹*match IRNode.inject in i*: (*node* - = *node* -) = - ⇒
        ‹*match RepE in e*: - $\Longrightarrow$ ($\bigwedge$*x*. - = *node x* $\Longrightarrow$ -) $\Longrightarrow$ - ⇒
          ‹*metis i e r*›››› |
   match node **in** *kind* - - = *node* - - **for** *node* ⇒
    ‹*match rep in r*: - $\Longrightarrow$ - = *node* - - $\Longrightarrow$ - ⇒
      ‹*match IRNode.inject in i*: (*node* - - = *node* - -) = - ⇒
        ‹*match RepE in e*: - $\Longrightarrow$ ($\bigwedge$*x y*. - = *node x y* $\Longrightarrow$ -) $\Longrightarrow$ - ⇒
          ‹*metis i e r*›››› |
   match node **in** *kind* - - = *node* - - - **for** *node* ⇒
    ‹*match rep in r*: - $\Longrightarrow$ - = *node* - - - $\Longrightarrow$ - ⇒
      ‹*match IRNode.inject in i*: (*node* - - - = *node* - - -) = - ⇒
        ‹*match RepE in e*: - $\Longrightarrow$ ($\bigwedge$*x y z*. - = *node x y z* $\Longrightarrow$ -) $\Longrightarrow$ - ⇒
          ‹*metis i e r*›››› |
  match node **in** *kind* - - = *node* - - - **for** *node* ⇒
    ‹*match rep in r*: - $\Longrightarrow$ - = *node* - - - $\Longrightarrow$ - ⇒

‹match IRNode.inject in i: (node - - - = node - - -) = - ⇒
‹match RepE in e: - $\Longrightarrow$ ($\bigwedge$x. - = node - - x $\Longrightarrow$ -) $\Longrightarrow$ - ⇒
‹metis i e r››››)

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

lemma *repDet*:
shows $(g \vdash n \simeq e1) \Longrightarrow (g \vdash n \simeq e2) \Longrightarrow e1 = e2$
⟨*proof*⟩

lemma *repAllDet*:
$g \vdash xs \simeq_L e1 \Longrightarrow$
$g \vdash xs \simeq_L e2 \Longrightarrow$
$e1 = e2$
⟨*proof*⟩

lemma *evalDet*:
$[m,p] \vdash e \mapsto v1 \Longrightarrow$
$[m,p] \vdash e \mapsto v2 \Longrightarrow$
$v1 = v2$
⟨*proof*⟩

lemma *evalAllDet*:
$[m,p] \vdash e \mapsto_L v1 \Longrightarrow$
$[m,p] \vdash e \mapsto_L v2 \Longrightarrow$
$v1 = v2$
⟨*proof*⟩

lemma *encodeEvalDet*:
$[g,m,p] \vdash e \mapsto v1 \Longrightarrow$
$[g,m,p] \vdash e \mapsto v2 \Longrightarrow$
$v1 = v2$
⟨*proof*⟩

lemma *graphDet*: $([g,m,p] \vdash nid \mapsto v1) \wedge ([g,m,p] \vdash nid \mapsto v2) \Longrightarrow v1 = v2$
⟨*proof*⟩

A valid value cannot be $UndefVal$.

lemma *valid-not-undef*:
assumes *a1*: *valid-value s val*
assumes *a2*: $s \neq$ *VoidStamp*
shows *val* $\neq$ *UndefVal*
⟨*proof*⟩

lemma *valid-VoidStamp*[*elim*]:
shows *valid-value VoidStamp val* $\Longrightarrow$
val = *UndefVal*

⟨*proof*⟩

lemma *valid-ObjStamp*[*elim*]:
shows *valid-value* (*ObjectStamp klass exact nonNull alwaysNull*) *val* $\Longrightarrow$
($\exists$ *v*. *val* = *ObjRef v*)
⟨*proof*⟩

lemma *valid-int32*[*elim*]:
shows *valid-value* (*IntegerStamp 32 l h*) *val* $\Longrightarrow$
($\exists$ *v*. *val* = *IntVal32 v*)
⟨*proof*⟩

lemma *valid-int64*[*elim*]:
shows *valid-value* (*IntegerStamp 64 l h*) *val* $\Longrightarrow$
($\exists$ *v*. *val* = *IntVal64 v*)
⟨*proof*⟩

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*] $\vdash$ *LeafExpr i* (*IntegerStamp 32 lo hi*) $\mapsto$ *val*
shows $\exists$ *v*. *val* = (*IntVal32 v*)

⟨*proof*⟩

lemma *leafint64*:
assumes *ev*: [*m*,*p*] $\vdash$ *LeafExpr i* (*IntegerStamp 64 lo hi*) $\mapsto$ *val*
shows $\exists$ *v*. *val* = (*IntVal64 v*)

⟨*proof*⟩

lemma *default-stamp* [*simp*]: *default-stamp* = *IntegerStamp 32* (−*2147483648*) *2147483647*
⟨*proof*⟩

lemma *valid32* [*simp*]:
assumes *valid-value* (*IntegerStamp 32 lo hi*) *val*
shows $\exists$ *v*. (*val* = (*IntVal32 v*) $\wedge$ *lo* $\leq$ *sint v* $\wedge$ *sint v* $\leq$ *hi*)
⟨*proof*⟩

lemma *valid64* [*simp*]:
assumes *valid-value* (*IntegerStamp 64 lo hi*) *val*
shows $\exists$ *v*. (*val* = (*IntVal64 v*) $\wedge$ *lo* $\leq$ *sint v* $\wedge$ *sint v* $\leq$ *hi*)
⟨*proof*⟩

experiment begin
lemma *int-stamp-implies-valid-value*:
[*m*,*p*] $\vdash$ *expr* $\mapsto$ *val* $\Longrightarrow$

valid-value (*stamp-expr expr*) *val*
⟨*proof*⟩
end

lemma *valid32or64*:
assumes *valid-value* (*IntegerStamp b lo hi*) *x*
shows $(\exists\ v1.\ (x = IntVal32\ v1)) \lor (\exists\ v2.\ (x = IntVal64\ v2))$
⟨*proof*⟩

lemma *valid32or64-both*:
assumes *valid-value* (*IntegerStamp b lox hix*) *x*
and *valid-value* (*IntegerStamp b loy hiy*) *y*
shows $(\exists\ v1\ v2.\ x = IntVal32\ v1 \land y = IntVal32\ v2) \lor (\exists\ v3\ v4.\ x = IntVal64\ v3 \land y = IntVal64\ v4)$
⟨*proof*⟩

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*⟩

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

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*⟩

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*))
⟨*proof*⟩

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 $e \geq e'$
  shows $(UnaryExpr\ op\ e) \geq (UnaryExpr\ op\ e')$
  ⟨*proof*⟩

lemma *mono-binary*:
  assumes $x \geq x'$
  assumes $y \geq y'$
  shows $(BinaryExpr\ op\ x\ y) \geq (BinaryExpr\ op\ x'\ y')$
  ⟨*proof*⟩

lemma *mono-conditional*:
  assumes $ce \geq ce'$
  assumes $te \geq te'$
  assumes $fe \geq fe'$
  shows $(ConditionalExpr\ ce\ te\ fe) \geq (ConditionalExpr\ ce'\ te'\ fe')$
⟨*proof*⟩

end

7 Tree to Graph

theory *TreeToGraph*
  imports
    Semantics.IRTreeEval
    Graph.IRGraph
begin

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

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

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

is-preevaluated (*SignedRemNode n* - - - - -) = *True* |
is-preevaluated (*ValuePhiNode n* - -) = *True* |
is-preevaluated - = *False*

inductive
rep :: *IRGraph* ⇒ *ID* ⇒ *IRExpr* ⇒ *bool* (- ⊢ - ≃ - *55*)
for *g* **where**

ConstantNode:
⟦*kind g n* = *ConstantNode c*⟧
⟹ *g* ⊢ *n* ≃ (*ConstantExpr c*) |

ParameterNode:
⟦*kind g n* = *ParameterNode i*;
stamp g n = *s*⟧
⟹ *g* ⊢ *n* ≃ (*ParameterExpr i s*) |

ConditionalNode:
⟦*kind g n* = *ConditionalNode c t f*;
g ⊢ *c* ≃ *ce*;
g ⊢ *t* ≃ *te*;
g ⊢ *f* ≃ *fe*⟧
⟹ *g* ⊢ *n* ≃ (*ConditionalExpr ce te fe*) |

AbsNode:
⟦*kind g n* = *AbsNode x*;
g ⊢ *x* ≃ *xe*⟧
⟹ *g* ⊢ *n* ≃ (*UnaryExpr UnaryAbs xe*) |

NotNode:
⟦*kind g n* = *NotNode x*;
g ⊢ *x* ≃ *xe*⟧
⟹ *g* ⊢ *n* ≃ (*UnaryExpr UnaryNot xe*) |

NegateNode:
⟦*kind g n* = *NegateNode x*;
g ⊢ *x* ≃ *xe*⟧
⟹ *g* ⊢ *n* ≃ (*UnaryExpr UnaryNeg xe*) |

LogicNegationNode:
⟦*kind g n* = *LogicNegationNode x*;
g ⊢ *x* ≃ *xe*⟧
⟹ *g* ⊢ *n* ≃ (*UnaryExpr UnaryLogicNegation xe*) |

AddNode:
⟦*kind g n* = *AddNode x y*;

$g \vdash x \simeq xe$;
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinAdd\ xe\ ye)$ |

MulNode:
$[\![kind\ g\ n = MulNode\ x\ y$;
$g \vdash x \simeq xe$;
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinMul\ xe\ ye)$ |

SubNode:
$[\![kind\ g\ n = SubNode\ x\ y$;
$g \vdash x \simeq xe$;
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinSub\ xe\ ye)$ |

AndNode:
$[\![kind\ g\ n = AndNode\ x\ y$;
$g \vdash x \simeq xe$;
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinAnd\ xe\ ye)$ |

OrNode:
$[\![kind\ g\ n = OrNode\ x\ y$;
$g \vdash x \simeq xe$;
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinOr\ xe\ ye)$ |

XorNode:
$[\![kind\ g\ n = XorNode\ x\ y$;
$g \vdash x \simeq xe$;
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinXor\ xe\ ye)$ |

IntegerBelowNode:
$[\![kind\ g\ n = IntegerBelowNode\ x\ y$;
$g \vdash x \simeq xe$;
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerBelow\ xe\ ye)$ |

IntegerEqualsNode:
$[\![kind\ g\ n = IntegerEqualsNode\ x\ y$;
$g \vdash x \simeq xe$;
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerEquals\ xe\ ye)$ |

IntegerLessThanNode:
$[\![kind\ g\ n = IntegerLessThanNode\ x\ y$;
$g \vdash x \simeq xe$;

$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerLessThan\ xe\ ye) \mid$

NarrowNode:
$[\![kind\ g\ n = NarrowNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)\ xe) \mid$

SignExtendNode:
$[\![kind\ g\ n = SignExtendNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnarySignExtend\ inputBits\ resultBits)\ xe) \mid$

ZeroExtendNode:
$[\![kind\ g\ n = ZeroExtendNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnaryZeroExtend\ inputBits\ resultBits)\ xe) \mid$

LeafNode:
$[\![is\text{-}preevaluated\ (kind\ g\ n);$
$stamp\ g\ n = s]\!]$
$\Longrightarrow g \vdash n \simeq (LeafExpr\ n\ s)$

code-pred $(modes\colon i \Rightarrow i \Rightarrow o \Rightarrow bool\ as\ exprE)\ rep\ \langle proof \rangle$

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

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

RepCons:
$[\![g \vdash x \simeq xe;$
$g \vdash xs \simeq_L xse]\!]$
$\Longrightarrow g \vdash x\#xs \simeq_L xe\#xse$

code-pred $(modes\colon i \Rightarrow i \Rightarrow o \Rightarrow bool\ as\ exprListE)\ replist\ \langle proof \rangle$

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

$$\frac{kind\ g\ n = ParameterNode\ i \qquad stamp\ g\ n = s}{g \vdash n \simeq ParameterExpr\ i\ s}$$

$$\frac{kind\ g\ n = AbsNode\ x \qquad g \vdash x \simeq xe}{g \vdash n \simeq UnaryExpr\ UnaryAbs\ xe}$$

$$\frac{kind\ g\ n = AddNode\ x\ y \qquad g \vdash x \simeq xe \qquad g \vdash y \simeq ye}{g \vdash n \simeq BinaryExpr\ BinAdd\ xe\ ye}$$

$$\frac{kind\ g\ n = MulNode\ x\ y \qquad g \vdash x \simeq xe \qquad g \vdash y \simeq ye}{g \vdash n \simeq BinaryExpr\ BinMul\ xe\ ye}$$

$$\frac{kind\ g\ n = SubNode\ x\ y \qquad g \vdash x \simeq xe \qquad g \vdash y \simeq ye}{g \vdash n \simeq BinaryExpr\ BinSub\ xe\ ye}$$

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

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

fun *stamp-unary* :: *IRUnaryOp* ⇒ *Stamp* ⇒ *Stamp* **where**
stamp-unary op (*IntegerStamp b lo hi*) = *unrestricted-stamp* (*IntegerStamp b lo hi*) |

stamp-unary op - = *IllegalStamp*

definition *fixed-32* :: *IRBinaryOp set* **where**
fixed-32 = {*BinIntegerEquals*, *BinIntegerLessThan*, *BinIntegerBelow*}

fun *stamp-binary* :: *IRBinaryOp* ⇒ *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**
stamp-binary op (*IntegerStamp b1 lo1 hi1*) (*IntegerStamp b2 lo2 hi2*) =
(*case op* ∈ *fixed-32 of True* ⇒ *unrestricted-stamp* (*IntegerStamp 32 lo1 hi1*) |
False ⇒
(*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 |
unary-node (*UnaryNarrow ib rb*) *v = NarrowNode ib rb v |*
unary-node (*UnarySignExtend ib rb*) *v = SignExtendNode ib rb v |*
unary-node (*UnaryZeroExtend ib rb*) *v = ZeroExtendNode ib rb v*

fun *bin-node* :: *IRBinaryOp* ⇒ *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 BinLeftShift x y = LeftShiftNode x y |
bin-node BinRightShift x y = RightShiftNode x y |
bin-node BinURightShift x y = UnsignedRightShiftNode x y |
bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
bin-node BinIntegerLessThan x y = IntegerLessThanNode x y |
bin-node BinIntegerBelow x y = IntegerBelowNode x y

fun *choose-32-64* :: *int* ⇒ *int64* ⇒ *Value* **where**
choose-32-64 bits val =
(*if bits = 32*
then (*IntVal32* (*ucast val*))
else (*IntVal64* (*val*)))

inductive *fresh-id* :: *IRGraph* ⇒ *ID* ⇒ *bool* **where**
n ∉ *ids g* ⟹ *fresh-id g n*

code-pred *fresh-id* ⟨*proof*⟩

fun *get-fresh-id* :: *IRGraph* ⇒ *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:
$[\![$*find-node-and-stamp* g (*ConstantNode* c, *constantAsStamp* c) = *Some* $n]\!]$
$\Longrightarrow g \triangleleft$ (*ConstantExpr* c) $\rightsquigarrow (g, n)$ |

ConstantNodeNew:
$[\![$*find-node-and-stamp* g (*ConstantNode* c, *constantAsStamp* c) = *None*;
n = *get-fresh-id* g;
g' = *add-node* n (*ConstantNode* c, *constantAsStamp* c) g $]\!]$
$\Longrightarrow g \triangleleft$ (*ConstantExpr* c) $\rightsquigarrow (g', n)$ |

ParameterNodeSame:
$[\![$*find-node-and-stamp* g (*ParameterNode* i, s) = *Some* $n]\!]$
$\Longrightarrow g \triangleleft$ (*ParameterExpr* i s) $\rightsquigarrow (g, n)$ |

ParameterNodeNew:
$[\![$*find-node-and-stamp* g (*ParameterNode* i, s) = *None*;
n = *get-fresh-id* g;
g' = *add-node* n (*ParameterNode* i, s) $g]\!]$
$\Longrightarrow g \triangleleft$ (*ParameterExpr* i s) $\rightsquigarrow (g', n)$ |

ConditionalNodeSame:
$[\![g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f])$;
s' = *meet* (*stamp* $g2$ t) (*stamp* $g2$ f);
find-node-and-stamp $g2$ (*ConditionalNode* c t f, s') = *Some* $n]\!]$
$\Longrightarrow g \triangleleft$ (*ConditionalExpr* ce te fe) $\rightsquigarrow (g2, n)$ |

ConditionalNodeNew:
$[\![g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f])$;
s' = *meet* (*stamp* $g2$ t) (*stamp* $g2$ f);
find-node-and-stamp $g2$ (*ConditionalNode* c t f, s') = *None*;
n = *get-fresh-id* $g2$;
g' = *add-node* n (*ConditionalNode* c t f, s') $g2]\!]$
$\Longrightarrow g \triangleleft$ (*ConditionalExpr* ce te fe) $\rightsquigarrow (g', n)$ |

UnaryNodeSame:
$[\![g \triangleleft xe \rightsquigarrow (g2, x)$;
s' = *stamp-unary* op (*stamp* $g2$ x);
find-node-and-stamp $g2$ (*unary-node* op x, s') = *Some* $n]\!]$
$\Longrightarrow g \triangleleft$ (*UnaryExpr* op xe) $\rightsquigarrow (g2, n)$ |

UnaryNodeNew:
$[\![g \triangleleft xe \rightsquigarrow (g2, x)$;
s' = *stamp-unary* op (*stamp* $g2$ x);

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

BinaryNodeSame:
$\llbracket g \triangleleft_L [xe,\ ye] \rightsquigarrow (g2,\ [x,\ y])$;
$s' = \mathit{stamp\text{-}binary}\ op\ (\mathit{stamp}\ g2\ x)\ (\mathit{stamp}\ g2\ y)$;
$\mathit{find\text{-}node\text{-}and\text{-}stamp}\ g2\ (\mathit{bin\text{-}node}\ op\ x\ y,\ s') = \mathit{Some}\ n\rrbracket$
$\Longrightarrow g \triangleleft (\mathit{BinaryExpr}\ op\ xe\ ye) \rightsquigarrow (g2,\ n)$ |

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

AllLeafNodes:
$\mathit{stamp}\ g\ n = s$
$\Longrightarrow g \triangleleft (\mathit{LeafExpr}\ n\ s) \rightsquigarrow (g,\ n)$ |

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

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

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool as unrepE*)
unrep ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool as unrepListE*) *unrepList* ⟨*proof*⟩

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}g \triangleleft xe \rightsquigarrow (g2,\ x) \qquad s' = \textit{stamp-unary } op\ (\textit{stamp } g2\ x)\\ \textit{find-node-and-stamp } g2\ (\textit{unary-node } op\ x,\ s') = \textit{Some } n\end{array}}{g \triangleleft \textit{UnaryExpr } op\ xe \rightsquigarrow (g2,\ n)}$$

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

$$\frac{\textit{stamp } g\ n = s}{g \triangleleft \textit{LeafExpr } n\ s \rightsquigarrow (g,\ n)}$$

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

values {(*n*, *g*) . (*eg2-sq* ◁ *sq-param0* ⇝ (*g*, *n*))}

definition *encodeeval* :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *ID* ⇒ *Value* ⇒ *bool*
([-,-,-] ⊢ - ↦ - *50*)
where

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

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

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

definition *graph-refinement* :: *IRGraph* $\Rightarrow$ *IRGraph* $\Rightarrow$ *bool* **where**
graph-refinement g1 g2 =
($\forall$ *n* . *n* $\in$ *ids g1* $\longrightarrow$ ($\forall$ *e1*. (*g1* $\vdash$ *n* $\simeq$ *e1*) $\longrightarrow$ ($\exists$ *e2*. (*g2* $\vdash$ *n* $\simeq$ *e2*) $\wedge$ *e1* $\geq$ *e2*)))

lemma *graph-refinement*:
graph-refinement g1 g2 $\Longrightarrow$ ($\forall$ *n m p v*. *n* $\in$ *ids g1* $\longrightarrow$ ([*g1*, *m*, *p*] $\vdash$ *n* $\mapsto$ *v*) $\longrightarrow$ ([*g2*, *m*, *p*] $\vdash$ *n* $\mapsto$ *v*))
⟨*proof*⟩

definition *graph-represents-expression* :: *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool*
(- $\vdash$ - $\trianglelefteq$ - *50*)
where
graph-represents-expression g n e = ($\forall$ *m p v* . ([*m*,*p*] $\vdash$ *e* $\mapsto$ *v*) $\longrightarrow$ ([*g*,*m*,*p*] $\vdash$ *n* $\mapsto$ *v*))

end
theory *TreeToGraphThms*
imports
TreeToGraph
IRTreeEvalThms
HOL−Eisbach.Eisbach
begin

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

lemma *mono-abs*:
assumes *kind g1 n* = *AbsNode x* $\wedge$ *kind g2 n* = *AbsNode x*
assumes (*g1* $\vdash$ *x* $\simeq$ *xe1*) $\wedge$ (*g2* $\vdash$ *x* $\simeq$ *xe2*)
assumes *xe1* $\geq$ *xe2*
assumes (*g1* $\vdash$ *n* $\simeq$ *e1*) $\wedge$ (*g2* $\vdash$ *n* $\simeq$ *e2*)
shows *e1* $\geq$ *e2*
⟨*proof*⟩

lemma *mono-not*:

assumes $kind\ g1\ n = NotNode\ x \wedge kind\ g2\ n = NotNode\ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-negate*:
assumes $kind\ g1\ n = NegateNode\ x \wedge kind\ g2\ n = NegateNode\ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-logic-negation*:
assumes $kind\ g1\ n = LogicNegationNode\ x \wedge kind\ g2\ n = LogicNegationNode\ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-narrow*:
assumes $kind\ g1\ n = NarrowNode\ ib\ rb\ x \wedge kind\ g2\ n = NarrowNode\ ib\ rb\ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-sign-extend*:
assumes $kind\ g1\ n = SignExtendNode\ ib\ rb\ x \wedge kind\ g2\ n = SignExtendNode\ ib\ rb\ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-zero-extend*:
assumes $kind\ g1\ n = ZeroExtendNode\ ib\ rb\ x \wedge kind\ g2\ n = ZeroExtendNode\ ib\ rb\ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-conditional-graph*:
assumes $kind\ g1\ n = ConditionalNode\ c\ t\ f \wedge kind\ g2\ n = ConditionalNode\ c\ t\ f$
assumes $(g1 \vdash c \simeq ce1) \wedge (g2 \vdash c \simeq ce2)$
assumes $(g1 \vdash t \simeq te1) \wedge (g2 \vdash t \simeq te2)$
assumes $(g1 \vdash f \simeq fe1) \wedge (g2 \vdash f \simeq fe2)$
assumes $ce1 \geq ce2 \wedge te1 \geq te2 \wedge fe1 \geq fe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-add*:
assumes $kind\ g1\ n = AddNode\ x\ y \wedge kind\ g2\ n = AddNode\ x\ y$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$
assumes $xe1 \geq xe2 \wedge ye1 \geq ye2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-mul*:
assumes $kind\ g1\ n = MulNode\ x\ y \wedge kind\ g2\ n = MulNode\ x\ y$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$
assumes $xe1 \geq xe2 \wedge ye1 \geq ye2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *encodes-contains*:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n \neq NoNode$
⟨*proof*⟩

lemma *no-encoding*:
assumes $n \notin ids\ g$
shows $\neg(g \vdash n \simeq e)$
⟨*proof*⟩

lemma *not-excluded-keep-type*:
assumes $n \in ids\ g1$
assumes $n \notin excluded$
assumes $(excluded \trianglelefteq as\text{-}set\ g1) \subseteq as\text{-}set\ g2$
shows $kind\ g1\ n = kind\ g2\ n \wedge stamp\ g1\ n = stamp\ g2\ n$
⟨*proof*⟩

method *metis-node-eq-unary* **for** *node* :: $'a \Rightarrow IRNode$ =
(*match IRNode.inject* **in** *i*: (*node* - = *node* -) = - ⇒
‹*metis i*›)

method *metis-node-eq-binary* **for** *node* :: $'a \Rightarrow 'a \Rightarrow IRNode$ =
  (*match IRNode.inject* **in** *i*: (*node* - - = *node* - -) = - ⇒
    ‹*metis i*›)
method *metis-node-eq-ternary* **for** *node* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow IRNode$ =
  (*match IRNode.inject* **in** *i*: (*node* - - - = *node* - - -) = - ⇒
    ‹*metis i*›)

lemma *graph-semantics-preservation*:
  assumes *a*: $e1' \geq e2'$
  assumes *b*: $(\{n'\} \trianglelefteq as\text{-}set\ g1) \subseteq as\text{-}set\ g2$
  assumes *c*: $g1 \vdash n' \simeq e1'$
  assumes *d*: $g2 \vdash n' \simeq e2'$
  shows *graph-refinement g1 g2*
  ⟨*proof*⟩

definition *maximal-sharing*:
  $maximal\text{-}sharing\ g = (\forall\ n1\ n2\ .\ n1 \in ids\ g \wedge n2 \in ids\ g \longrightarrow$
    $(\forall\ e.\ (g \vdash n1 \simeq e) \wedge (g \vdash n2 \simeq e) \longrightarrow n1 = n2))$

lemma *tree-to-graph-rewriting*:
  $e1 \geq e2$
  $\wedge\ (g1 \vdash n \simeq e1) \wedge maximal\text{-}sharing\ g1$
  $\wedge\ (\{n\} \trianglelefteq as\text{-}set\ g1) \subseteq as\text{-}set\ g2$
  $\wedge\ (g2 \vdash n \simeq e2) \wedge maximal\text{-}sharing\ g2$
  $\Longrightarrow graph\text{-}refinement\ g1\ g2$
  ⟨*proof*⟩

declare [[*simp-trace*]]
lemma *equal-refines*:
  fixes *e1 e2* :: *IRExpr*
  assumes $e1 = e2$
  shows $e1 \geq e2$
  ⟨*proof*⟩
declare [[*simp-trace=false*]]

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

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

$\langle proof \rangle$

lemma *graph-construction*:
  $e1 \geq e2$
  $\wedge$ *as-set g1* $\subseteq$ *as-set g2* $\wedge$ *maximal-sharing g1*
  $\wedge$ $(g2 \vdash n \simeq e2)$ $\wedge$ *maximal-sharing g2*
  $\Longrightarrow (g2 \vdash n \trianglelefteq e1)$ $\wedge$ *graph-refinement g1 g2*
  $\langle proof \rangle$

end

8 Control-flow Semantics

theory *IRStepObj*
  imports
    TreeToGraph
begin

8.1 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 = $((\lambda f.\ \lambda p.\ UndefVal), 0)$

8.2 Intraprocedural Semantics

fun *find-index* :: $'a \Rightarrow 'a$ *list* $\Rightarrow$ *nat* **where**

find-index - [] = *0* |
find-index v (*x* # *xs*) = (*if* (*x*=*v*) *then 0 else find-index v xs* + *1*)

fun *phi-list* :: *IRGraph* ⇒ *ID* ⇒ *ID list* **where**
phi-list g n =
(*filter* (λ*x*.(*is-PhiNode* (*kind g x*)))
(*sorted-list-of-set* (*usages g n*)))

fun *input-index* :: *IRGraph* ⇒ *ID* ⇒ *ID* ⇒ *nat* **where**
input-index g n n′ = *find-index n′* (*inputs-of* (*kind g n*))

fun *phi-inputs* :: *IRGraph* ⇒ *nat* ⇒ *ID list* ⇒ *ID list* **where**
phi-inputs g i nodes = (*map* (λ*n*. (*inputs-of* (*kind g n*))!(*i* + *1*)) *nodes*)

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

Intraprocedural semantics are given as a small-step semantics.

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

inductive *step* :: *IRGraph* ⇒ *Params* ⇒ (*ID* × *MapState* × *FieldRefHeap*) ⇒ (*ID* × *MapState* × *FieldRefHeap*) ⇒ *bool*
(-, - ⊢ - → - *55*) **for** *g p* **where**

SequentialNode:
⟦*is-sequential-node* (*kind g nid*);
nid′ = (*successors-of* (*kind g nid*))!*0*⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*) |

IfNode:
⟦*kind g nid* = (*IfNode cond tb fb*);
g ⊢ *cond* ≃ *condE*;
[*m*, *p*] ⊢ *condE* ↦ *val*;
nid′ = (*if val-to-bool val then tb else fb*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*) |

EndNodes:
⟦*is-AbstractEndNode* (*kind g nid*);
merge = *any-usage g nid*;
is-AbstractMergeNode (*kind g merge*);

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

$m' = set\text{-}phis\ phis\ vs\ m]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (merge,\ m',\ h)\ |$

NewInstanceNode:
$[\![kind\ g\ nid = (NewInstanceNode\ nid\ f\ obj\ nid');$
$(h',\ ref) = h\text{-}new\text{-}inst\ h;$
$m' = m(nid := ref)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')\ |$

LoadFieldNode:
$[\![kind\ g\ nid = (LoadFieldNode\ nid\ f\ (Some\ obj)\ nid');$
$g \vdash obj \simeq objE;$
$[m,\ p] \vdash objE \mapsto ObjRef\ ref;$
$h\text{-}load\text{-}field\ f\ ref\ h = v;$
$m' = m(nid := v)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h)\ |$

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

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

StaticLoadFieldNode:
$[\![kind\ g\ nid = (LoadFieldNode\ nid\ f\ None\ nid');$
$h\text{-}load\text{-}field\ f\ None\ h = v;$
$m' = \ m(nid := v)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h)\ |$

StoreFieldNode:
$[\![kind\ g\ nid = (StoreFieldNode\ nid\ f\ newval\ \text{-}\ (Some\ obj)\ nid');$
$g \vdash newval \simeq newvalE;$
$g \vdash obj \simeq objE;$

$[m,\ p] \vdash newvalE \mapsto val;$
$[m,\ p] \vdash objE \mapsto ObjRef\ ref;$
$h' = h\text{-}store\text{-}field\ f\ ref\ val\ h;$
$m' = \ m(nid := val)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')\ |$

StaticStoreFieldNode:
$[\![kind\ g\ nid = (StoreFieldNode\ nid\ f\ newval\ \text{-}\ None\ nid');$
$g \vdash newval \simeq newvalE;$
$[m,\ p] \vdash newvalE \mapsto val;$
$h' = h\text{-}store\text{-}field\ f\ None\ val\ h;$
$m' = \ m(nid := val)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')$

code-pred $(modes\colon i \Rightarrow i \Rightarrow i * i * i \Rightarrow o * o * o \Rightarrow bool)\ step\ \langle proof\rangle$

8.3 Interprocedural Semantics

type-synonym $Signature = string$
type-synonym $Program = Signature \rightharpoonup IRGraph$

inductive $step\text{-}top :: Program \Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap \Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap \Rightarrow bool$
$(\text{-} \vdash \text{-} \longrightarrow \text{-}\ 55)$
for P **where**

Lift:
$[\![g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#stk,\ h) \longrightarrow ((g,nid',m',p)\#stk,\ h')\ |$

InvokeNodeStep:
$[\![is\text{-}Invoke\ (kind\ g\ nid);$

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

ReturnNode:
$[\![kind\ g\ nid = (ReturnNode\ (Some\ expr)\ \text{-});$
$g \vdash expr \simeq e;$
$[m,\ p] \vdash e \mapsto v;$

$cm' = cm(cnid := v);$
$cnid' = (successors\text{-}of\ (kind\ cg\ cnid))!0]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,cnid',cm',cp)\#stk,\ h)$ |

ReturnNodeVoid:
$[\![kind\ g\ nid = (ReturnNode\ None\ \text{-});$
$cm' = cm(cnid := (ObjRef\ (Some\ (2048))));$

$cnid' = (successors\text{-}of\ (kind\ cg\ cnid))!0]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,cnid',cm',cp)\#stk,\ h)$ |

UnwindNode:
$[\![kind\ g\ nid = (UnwindNode\ exception);$

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

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

$cm' = cm(cnid := e)]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,exEdge,cm',cp)\#stk,\ h)$

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

8.4 Big-step Execution

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

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

inductive *exec* :: *Program*
$\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap$
$\Rightarrow Trace$
$\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap$
$\Rightarrow Trace$
$\Rightarrow bool$
$(\text{-} \vdash \text{-} \mid \text{-} \longrightarrow\!\ast\ \text{-} \mid \text{-})$
for *P*
where
$[\![P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h');$
$\neg(has\text{-}return\ m');$

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

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

|

$[\![P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h');$
  has-return m';

  $l' = (l\ @\ [(g,nid,m,p)])]\!]$
  $\Longrightarrow$ *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 as Exec*) *exec* ⟨*proof*⟩

inductive *exec-debug* :: *Program*
  ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap*
  ⇒ *nat*
  ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap*
  ⇒ *bool*
 (-⊢-→∗-∗ -)
 where
 $[\![n > 0;$
  $p \vdash s \longrightarrow s'$;
  exec-debug $p\ s'\ (n - 1)\ s'']\!]$
  $\Longrightarrow$ *exec-debug* $p\ s\ n\ s''$ |

 $[\![n = 0]\!]$
  $\Longrightarrow$ *exec-debug* $p\ s\ n\ s$
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *exec-debug* ⟨*proof*⟩

8.4.1 Heap Testing

definition *p3*:: *Params* **where**
 p3 = [*IntVal32 3*]

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

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

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

values {*h-load-field field-sq None* (*prod.snd res*)
  | *res*. (λ*x*. *Some eg3-sq*) ⊢ ([(*eg3-sq*, *0*, *new-map-state*, *p3*), (*eg3-sq*, *0*,

new-map-state, *p3*)], *new-heap*) →∗*3*∗ *res*}

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

values {*h-load-field field-sq* (*Some 0*) (*prod.snd res*) | *res*.
(*λx. Some eg4-sq*) ⊢ ([(*eg4-sq*, *0*, *new-map-state*, *p3*), (*eg4-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) →∗*4*∗ *res*}

end

9 Properties of Control-flow Semantics

theory *IRStepThms*
imports
IRStepObj
IRTreeEvalThms
begin

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

theorem *stepDet*:
$(g,\ p \vdash (nid,m,h) \rightarrow next) \Longrightarrow$
$(\forall\ next'.\ ((g,\ p \vdash (nid,m,h) \rightarrow next') \longrightarrow next = next'))$

⟨*proof*⟩

lemma *stepRefNode*:
$[\![kind\ g\ nid = RefNode\ nid']\!] \Longrightarrow g,\ p \vdash (nid,m,h) \rightarrow (nid',m,h)$
⟨*proof*⟩

lemma *IfNodeStepCases*:
assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
assumes $g \vdash cond \simeq condE$
assumes $[m,\ p] \vdash condE \mapsto v$
assumes $g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)$

shows $nid' \in \{tb, fb\}$
⟨*proof*⟩

lemma *IfNodeSeq*:
shows $kind\ g\ nid = IfNode\ cond\ tb\ fb \longrightarrow \neg(is\text{-}sequential\text{-}node\ (kind\ g\ nid))$
⟨*proof*⟩

lemma *IfNodeCond*:
assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
assumes $g, p \vdash (nid, m, h) \rightarrow (nid', m, h)$
shows $\exists\ condE\ v.\ ((g \vdash cond \simeq condE) \wedge ([m, p] \vdash condE \mapsto v))$
⟨*proof*⟩

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

end

10 Proof Infrastructure

10.1 Bisimulation

theory *Bisimulation*
imports
Stuttering
begin

inductive *weak-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
(- . - ∼ -) **for** *nid* **where**
$[\![\forall P'.\ (g\ m\ p\ h \vdash nid \rightsquigarrow P') \longrightarrow (\exists\ Q'\ .\ (g'\ m\ p\ h \vdash nid \rightsquigarrow Q') \wedge P' = Q');$
$\forall Q'.\ (g'\ m\ p\ h \vdash nid \rightsquigarrow Q') \longrightarrow (\exists\ P'\ .\ (g\ m\ p\ h \vdash nid \rightsquigarrow P') \wedge P' = Q')]\!]$
$\Longrightarrow nid\ .\ g \sim g'$

A strong bisimilution between no-op transitions

inductive *strong-noop-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
(- | - ∼ -) **for** *nid* **where**
$[\![\forall P'.\ (g, p \vdash (nid, m, h) \rightarrow P') \longrightarrow (\exists\ Q'\ .\ (g', p \vdash (nid, m, h) \rightarrow Q') \wedge P' = Q');$
$\forall Q'.\ (g', p \vdash (nid, m, h) \rightarrow Q') \longrightarrow (\exists\ P'\ .\ (g, p \vdash (nid, m, h) \rightarrow P') \wedge P' = Q')]\!]$
$\Longrightarrow nid \mid g \sim g'$

lemma *lockstep-strong-bisimilulation*:
assumes $g' = replace\text{-}node\ nid\ node\ g$

assumes $g, p \vdash (nid, m, h) \rightarrow (nid', m, h)$
assumes $g', p \vdash (nid, m, h) \rightarrow (nid', m, h)$
shows $nid \mid g \sim g'$
⟨*proof*⟩

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

end

10.2 Formedness Properties

theory *Form*
imports
Semantics.TreeToGraph
begin

definition *wf-start* **where**
wf-start g = (*0* ∈ *ids g* ∧
is-StartNode (*kind g 0*))

definition *wf-closed* **where**
wf-closed g =
(∀ *n* ∈ *ids g* .
inputs g n ⊆ *ids g* ∧
succ g n ⊆ *ids g* ∧
kind g n ≠ *NoNode*)

definition *wf-phis* **where**
wf-phis g =
(∀ *n* ∈ *ids g*.
is-PhiNode (*kind g n*) ⟶
length (*ir-values* (*kind g n*))
= *length* (*ir-ends*
(*kind g* (*ir-merge* (*kind g n*)))))

definition *wf-ends* **where**
wf-ends g =
(∀ *n* ∈ *ids g* .
is-AbstractEndNode (*kind g n*) ⟶
card (*usages g n*) > *0*)

fun *wf-graph* :: *IRGraph* ⇒ *bool* **where**
wf-graph g = (*wf-start g* ∧ *wf-closed g* ∧ *wf-phis g* ∧ *wf-ends g*)

lemmas *wf-folds* =

wf-graph.simps
wf-start-def
wf-closed-def
wf-phis-def
wf-ends-def

fun *wf-stamps* :: *IRGraph* ⇒ *bool* **where**
wf-stamps g = (∀ *n* ∈ *ids g* .
(∀ *v m p e* . (*g* ⊢ *n* ≃ *e*) ∧ ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ *valid-value* (*stamp-expr e*) *v*))

fun *wf-stamp* :: *IRGraph* ⇒ (*ID* ⇒ *Stamp*) ⇒ *bool* **where**
wf-stamp g s = (∀ *n* ∈ *ids g* .
(∀ *v m p e* . (*g* ⊢ *n* ≃ *e*) ∧ ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ *valid-value* (*s n*) *v*))

lemma *wf-empty*: *wf-graph start-end-graph*
⟨*proof*⟩

lemma *wf-eg2-sq*: *wf-graph eg2-sq*
⟨*proof*⟩

fun *wf-logic-node-inputs* :: *IRGraph* ⇒ *ID* ⇒ *bool* **where**
wf-logic-node-inputs g n =
(∀ *inp* ∈ *set* (*inputs-of* (*kind g n*)) . (∀ *v m p* . ([*g*, *m*, *p*] ⊢ *inp* ↦ *v*) ⟶ *wf-bool* *v*))

fun *wf-values* :: *IRGraph* ⇒ *bool* **where**
wf-values g = (∀ *n* ∈ *ids g* .
(∀ *v m p* . ([*g*, *m*, *p*] ⊢ *n* ↦ *v*) ⟶
(*is-LogicNode* (*kind g n*) ⟶
wf-bool v ∧ *wf-logic-node-inputs g n*)))

end

10.3 Dynamic Frames

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

theory *IRGraphFrames*
imports
Form
Semantics.IRTreeEval
begin

fun *unchanged* :: *ID set* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
unchanged ns g1 g2 = (∀ *n* . *n* ∈ *ns* ⟶

$(n \in ids\ g1 \wedge n \in ids\ g2 \wedge kind\ g1\ n = kind\ g2\ n \wedge stamp\ g1\ n = stamp\ g2\ n))$

fun *changeonly* :: *ID set* $\Rightarrow$ *IRGraph* $\Rightarrow$ *IRGraph* $\Rightarrow$ *bool* **where**
changeonly ns g1 g2 $= (\forall\ n\ .\ n \in ids\ g1 \wedge n \notin ns \longrightarrow$
$(n \in ids\ g1 \wedge n \in ids\ g2 \wedge kind\ g1\ n = kind\ g2\ n \wedge stamp\ g1\ n = stamp\ g2\ n))$

lemma *node-unchanged*:
assumes *unchanged ns g1 g2*
assumes *nid* $\in$ *ns*
shows *kind g1 nid* $=$ *kind g2 nid*
⟨*proof*⟩

lemma *other-node-unchanged*:
assumes *changeonly ns g1 g2*
assumes *nid* $\in$ *ids g1*
assumes *nid* $\notin$ *ns*
shows *kind g1 nid* $=$ *kind g2 nid*
⟨*proof*⟩

Some notation for input nodes used

inductive *eval-uses*:: *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *ID* $\Rightarrow$ *bool*
for *g* **where**

use0: *nid* $\in$ *ids g*
$\Longrightarrow$ *eval-uses g nid nid* |

use-inp: *nid'* $\in$ *inputs g n*
$\Longrightarrow$ *eval-uses g nid nid'* |

use-trans: ⟦*eval-uses g nid nid'*;
eval-uses g nid' nid''⟧
$\Longrightarrow$ *eval-uses g nid nid''*

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

lemma *eval-usages-self*:
assumes *nid* $\in$ *ids g*
shows *nid* $\in$ *eval-usages g nid*
⟨*proof*⟩

lemma *not-in-g-inputs*:
assumes *nid* $\notin$ *ids g*
shows *inputs g nid* $= \{\}$
⟨*proof*⟩

lemma *child-member*:

assumes $n = kind\ g\ nid$
assumes $n \neq NoNode$
assumes $List.member\ (inputs\text{-}of\ n)\ child$
shows $child \in inputs\ g\ nid$
$\langle proof \rangle$

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

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

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

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

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

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

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

$\langle proof \rangle$

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

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

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

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

lemma *elim-inp-set*:
assumes $k = kind\ g\ nid$
assumes $k \neq NoNode$
assumes $child \in set\ (inputs\text{-}of\ k)$
shows $child \in inputs\ g\ nid$
$\langle proof \rangle$

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

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

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

theorem *stay-same-encoding*:

assumes *nc*: *unchanged* (*eval-usages g1 nid*) *g1 g2*
assumes *g1*: *g1* $\vdash$ *nid* $\simeq$ *e*
assumes *wf*: *wf-graph g1*
shows *g2* $\vdash$ *nid* $\simeq$ *e*
$\langle$*proof*$\rangle$

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

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

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

lemma *add-node-unchanged*:
assumes *new* $\notin$ *ids g*
assumes *nid* $\in$ *ids g*
assumes *gup* = *add-node new k g*
assumes *wf-graph g*
shows *unchanged* (*eval-usages g nid*) *g gup*
$\langle$*proof*$\rangle$

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

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

lemma *no-external-use*:
  assumes *wf-graph g*
  assumes $nid' \notin ids\ g$
  assumes $nid \in ids\ g$
  shows $\neg(eval\text{-}uses\ g\ nid\ nid')$
⟨*proof*⟩

end

10.4 Graph Rewriting

theory
  Rewrites
imports
  IRGraphFrames
  Stuttering
begin

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

lemma *replace-usages-effect*:
  assumes $g' = replace\text{-}usages\ nid\ nid'\ g$
  shows $kind\ g'\ nid = RefNode\ nid'$
  ⟨*proof*⟩

lemma *replace-usages-changeonly*:
  assumes $nid \in ids\ g$
  assumes $g' = replace\text{-}usages\ nid\ nid'\ g$
  shows $changeonly\ \{nid\}\ g\ g'$
  ⟨*proof*⟩

lemma *replace-usages-unchanged*:
  assumes $nid \in ids\ g$
  assumes $g' = replace\text{-}usages\ nid\ nid'\ g$
  shows $unchanged\ (ids\ g - \{nid\})\ g\ g'$
  ⟨*proof*⟩

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

lemma *max-plus-one*:
  fixes $c :: ID\ set$
  shows $[\![finite\ c;\ c \neq \{\}]\!] \Longrightarrow (Max\ c) + 1 \notin c$
  ⟨*proof*⟩

lemma *ids-finite*:

finite (*ids* *g*)
⟨*proof*⟩

lemma *nextNidNotIn*:
ids *g* ≠ {} ⟶ *nextNid* *g* ∉ *ids* *g*
⟨*proof*⟩

fun *constantCondition* :: *bool* ⇒ *ID* ⇒ *IRNode* ⇒ *IRGraph* ⇒ *IRGraph* **where**
constantCondition *val* *nid* (*IfNode* *cond* *t* *f*) *g* =
replace-node *nid* (*IfNode* (*nextNid* *g*) *t* *f*, *stamp* *g* *nid*)
(*add-node* (*nextNid* *g*) ((*ConstantNode* (*bool-to-val* *val*)), *constantAsStamp* (*bool-to-val* *val*)) *g*) |
constantCondition *cond* *nid* - *g* = *g*

lemma *constantConditionTrue*:
assumes *kind* *g* *ifcond* = *IfNode* *cond* *t* *f*
assumes *g*′ = *constantCondition* *True* *ifcond* (*kind* *g* *ifcond*) *g*
shows *g*′, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
⟨*proof*⟩

lemma *constantConditionFalse*:
assumes *kind* *g* *ifcond* = *IfNode* *cond* *t* *f*
assumes *g*′ = *constantCondition* *False* *ifcond* (*kind* *g* *ifcond*) *g*
shows *g*′, *p* ⊢ (*ifcond*, *m*, *h*) → (*f*, *m*, *h*)
⟨*proof*⟩

lemma *diff-forall*:
assumes ∀ *n*∈*ids* *g* − {*nid*}. *cond* *n*
shows ∀ *n*. *n* ∈ *ids* *g* ∧ *n* ∉ {*nid*} ⟶ *cond* *n*
⟨*proof*⟩

lemma *replace-node-changeonly*:
assumes *g*′ = *replace-node* *nid* *node* *g*
shows *changeonly* {*nid*} *g* *g*′
⟨*proof*⟩

lemma *add-node-changeonly*:
assumes *g*′ = *add-node* *nid* *node* *g*
shows *changeonly* {*nid*} *g* *g*′
⟨*proof*⟩

lemma *constantConditionNoEffect*:
assumes ¬(*is-IfNode* (*kind* *g* *nid*))
shows *g* = *constantCondition* *b* *nid* (*kind* *g* *nid*) *g*
⟨*proof*⟩

lemma *constantConditionIfNode*:
assumes *kind* *g* *nid* = *IfNode* *cond* *t* *f*
shows *constantCondition* *val* *nid* (*kind* *g* *nid*) *g* =

replace-node nid (*IfNode* (*nextNid g*) *t f*, *stamp g nid*)
(*add-node* (*nextNid g*) ((*ConstantNode* (*bool-to-val val*)), *constantAsStamp* (*bool-to-val val*)) *g*)
⟨*proof*⟩

lemma *constantCondition-changeonly*:
assumes $nid \in ids\ g$
assumes $g' = constantCondition\ b\ nid\ (kind\ g\ nid)\ g$
shows $changeonly\ \{nid\}\ g\ g'$
⟨*proof*⟩

lemma *constantConditionNoIf*:
assumes $\forall\, cond\ t\ f.\ kind\ g\ ifcond \neq IfNode\ cond\ t\ f$
assumes $g' = constantCondition\ val\ ifcond\ (kind\ g\ ifcond)\ g$
shows $\exists\, nid'\ .(g\ m\ p\ h \vdash ifcond \leadsto nid') \longleftrightarrow (g'\ m\ p\ h \vdash ifcond \leadsto nid')$
⟨*proof*⟩

lemma *constantConditionValid*:
assumes $kind\ g\ ifcond = IfNode\ cond\ t\ f$
assumes $[g,\ m,\ p] \vdash cond \mapsto v$
assumes $const = val\text{-}to\text{-}bool\ v$
assumes $g' = constantCondition\ const\ ifcond\ (kind\ g\ ifcond)\ g$
shows $\exists\, nid'\ .(g\ m\ p\ h \vdash ifcond \leadsto nid') \longleftrightarrow (g'\ m\ p\ h \vdash ifcond \leadsto nid')$
⟨*proof*⟩

end

10.5 Stuttering

theory *Stuttering*
imports
Semantics.IRStepThms
begin

inductive *stutter*:: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *FieldRefHeap* ⇒ *ID* ⇒ *ID* ⇒ *bool* (- - - - ⊢ - ⇝ - *55*)
for *g m p h* **where**

StutterStep:
$[\![g,\ p \vdash (nid,m,h) \rightarrow (nid',m,h)]\!]$
$\Longrightarrow g\ m\ p\ h \vdash nid \leadsto nid'$ |

Transitive:
$[\![g,\ p \vdash (nid,m,h) \rightarrow (nid'',m,h);$
$g\ m\ p\ h \vdash nid'' \leadsto nid']\!]$
$\Longrightarrow g\ m\ p\ h \vdash nid \leadsto nid'$

lemma *stuttering-successor*:

assumes $(g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m,\ h))$
shows $\{P'.\ (g\ m\ p\ h \vdash nid \leadsto P')\} = \{nid'\} \cup \{nid''.\ (g\ m\ p\ h \vdash nid' \leadsto nid'')\}$
$\langle proof \rangle$

end

11 Canonicalization Phase

theory *CanonicalizationTree*
imports
Semantics.TreeToGraph
Semantics.IRTreeEval
begin

fun *is-idempotent-binary* :: *IRBinaryOp* ⇒ *bool* **where**
is-idempotent-binary BinAnd = *True* |
is-idempotent-binary BinOr = *True* |
is-idempotent-binary - = *False*

fun *is-idempotent-unary* :: *IRUnaryOp* ⇒ *bool* **where**
is-idempotent-unary UnaryAbs = *True* |
is-idempotent-unary - = *False*

fun *is-self-inverse* :: *IRUnaryOp* ⇒ *bool* **where**
is-self-inverse UnaryNeg = *True* |
is-self-inverse UnaryNot = *True* |
is-self-inverse UnaryLogicNegation = *True* |
is-self-inverse - = *False*

fun *is-neutral* :: *IRBinaryOp* ⇒ *Value* ⇒ *bool* **where**

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

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

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

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

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

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

is-neutral - - = *False*

fun *is-annihilator* :: *IRBinaryOp* ⇒ *Value* ⇒ *bool* **where**

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

is-annihilator BinAnd (*IntVal32 x*) = (*x* = *0*) |
is-annihilator BinAnd (*IntVal64 x*) = (*x* = *0*) |

is-annihilator BinOr (*IntVal32 x*) = (*x* = *1*) |
is-annihilator BinOr (*IntVal64 x*) = (*x* = *1*) |

is-annihilator - - = *False*

fun *int-to-value* :: *Value* ⇒ *int* ⇒ *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* ⇒ *IRExpr* ⇒ *bool* **where**
binary-const-fold:
⟦*x* = (*ConstantExpr val1*);
y = (*ConstantExpr val2*);
val = *bin-eval op val1 val2*;
val ≠ *UndefVal*⟧
⟹ *CanonicalizeBinaryOp* (*BinaryExpr op x y*) (*ConstantExpr val*) |

binary-fold-yneutral:
⟦*y* = (*ConstantExpr c*);
is-neutral op c;
stampx = *stamp-expr x*;
stampy = *stamp-expr y*;
stp-bits stampx = *stp-bits stampy*;
is-IntegerStamp stampx ∧ *is-IntegerStamp stampy*⟧
⟹ *CanonicalizeBinaryOp* (*BinaryExpr op x y*) *x* |

binary-fold-yzero32:
⟦*y* = *ConstantExpr c*;
is-annihilator op c;
stampx = *stamp-expr x*;

$stampy = stamp\text{-}expr\ y$;
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy$;
$stp\text{-}bits\ stampx = 32$;
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy]\!]$
$\Longrightarrow CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ (ConstantExpr\ c)\ |$

$binary\text{-}fold\text{-}yzero64$:
$[\![y = ConstantExpr\ c$;
$is\text{-}annihilator\ op\ c$;
$stampx = stamp\text{-}expr\ x$;
$stampy = stamp\text{-}expr\ y$;
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy$;
$stp\text{-}bits\ stampx = 64$;
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy]\!]$
$\Longrightarrow CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ (ConstantExpr\ c)\ |$

$binary\text{-}idempotent$:
$[\![is\text{-}idempotent\text{-}binary\ op]\!]$
$\Longrightarrow CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ x)\ x$

inductive $CanonicalizeUnaryOp :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ **where**
$unary\text{-}const\text{-}fold$:
$[\![val' = unary\text{-}eval\ op\ val$;
$val' \neq UndefVal]\!]$
$\Longrightarrow CanonicalizeUnaryOp\ (UnaryExpr\ op\ (ConstantExpr\ val))\ (ConstantExpr\ val')$

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

$mul\text{-}negate32$:
$[\![y = ConstantExpr\ (IntVal32\ (-1))$;
$stamp\text{-}expr\ x = IntegerStamp\ 32\ lo\ hi]\!]$
$\Longrightarrow CanonicalizeMul\ (BinaryExpr\ BinMul\ x\ y)\ (UnaryExpr\ UnaryNeg\ x)\ |$
$mul\text{-}negate64$:
$[\![y = ConstantExpr\ (IntVal64\ (-1))$;
$stamp\text{-}expr\ x = IntegerStamp\ 64\ lo\ hi]\!]$
$\Longrightarrow CanonicalizeMul\ (BinaryExpr\ BinMul\ x\ y)\ (UnaryExpr\ UnaryNeg\ x)$

inductive $CanonicalizeAdd :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ **where**
$add\text{-}xsub$:

$[\![x = (BinaryExpr\ BinSub\ a\ y)$;
$stampa = stamp\text{-}expr\ a$;
$stampy = stamp\text{-}expr\ y$;
$is\text{-}IntegerStamp\ stampa \wedge is\text{-}IntegerStamp\ stampy$;
$stp\text{-}bits\ stampa = stp\text{-}bits\ stampy]\!]$
$\Longrightarrow CanonicalizeAdd\ (BinaryExpr\ BinAdd\ x\ y)\ a\ |$

add-ysub:

$[\![y = (BinaryExpr\ BinSub\ a\ x);$
$stampa = stamp\text{-}expr\ a;$
$stampx = stamp\text{-}expr\ x;$
$is\text{-}IntegerStamp\ stampa \wedge is\text{-}IntegerStamp\ stampx;$
$stp\text{-}bits\ stampa = stp\text{-}bits\ stampx]\!]$
$\Longrightarrow CanonicalizeAdd\ (BinaryExpr\ BinAdd\ x\ y)\ a\ |$

add-xnegate:

$[\![nx = (UnaryExpr\ UnaryNeg\ x);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy]\!]$
$\Longrightarrow CanonicalizeAdd\ (BinaryExpr\ BinAdd\ nx\ y)\ (BinaryExpr\ BinSub\ y\ x)\ |$

add-ynegate:

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

inductive *CanonicalizeSub* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**

sub-same32:

$[\![stampx = stamp\text{-}expr\ x;$
$stampx = IntegerStamp\ 32\ lo\ hi]\!]$
$\Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ x\ x)\ (ConstantExpr\ (IntVal32\ 0))\ |$
sub-same64:

$[\![stampx = stamp\text{-}expr\ x;$
$stampx = IntegerStamp\ 64\ lo\ hi]\!]$
$\Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ x\ x)\ (ConstantExpr\ (IntVal64\ 0))\ |$

sub-left-add1:

$[\![x = (BinaryExpr\ BinAdd\ a\ b);$
$stampa = stamp\text{-}expr\ a;$
$stampb = stamp\text{-}expr\ b;$
$is\text{-}IntegerStamp\ stampa \wedge is\text{-}IntegerStamp\ stampb;$
$stp\text{-}bits\ stampa = stp\text{-}bits\ stampb]\!]$

$\Longrightarrow$ *CanonicalizeSub* (*BinaryExpr BinSub x b*) *a* |

sub-left-add2:

$[\![x = (\mathit{BinaryExpr}\ \mathit{BinAdd}\ a\ b);$
$\mathit{stampa} = \mathit{stamp\text{-}expr}\ a;$
$\mathit{stampb} = \mathit{stamp\text{-}expr}\ b;$
$\mathit{is\text{-}IntegerStamp}\ \mathit{stampa} \wedge \mathit{is\text{-}IntegerStamp}\ \mathit{stampb};$
$\mathit{stp\text{-}bits}\ \mathit{stampa} = \mathit{stp\text{-}bits}\ \mathit{stampb}]\!]$
$\Longrightarrow \mathit{CanonicalizeSub}\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ x\ a)\ b\ |$

sub-left-sub:

$[\![x = (\mathit{BinaryExpr}\ \mathit{BinSub}\ a\ b);$
$\mathit{stampa} = \mathit{stamp\text{-}expr}\ a;$
$\mathit{stampb} = \mathit{stamp\text{-}expr}\ b;$
$\mathit{is\text{-}IntegerStamp}\ \mathit{stampa} \wedge \mathit{is\text{-}IntegerStamp}\ \mathit{stampb};$
$\mathit{stp\text{-}bits}\ \mathit{stampa} = \mathit{stp\text{-}bits}\ \mathit{stampb}]\!]$
$\Longrightarrow \mathit{CanonicalizeSub}\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ x\ a)\ (\mathit{UnaryExpr}\ \mathit{UnaryNeg}\ b)\ |$

sub-right-add1:

$[\![y = (\mathit{BinaryExpr}\ \mathit{BinAdd}\ a\ b);$
$\mathit{stampa} = \mathit{stamp\text{-}expr}\ a;$
$\mathit{stampb} = \mathit{stamp\text{-}expr}\ b;$
$\mathit{is\text{-}IntegerStamp}\ \mathit{stampa} \wedge \mathit{is\text{-}IntegerStamp}\ \mathit{stampb};$
$\mathit{stp\text{-}bits}\ \mathit{stampa} = \mathit{stp\text{-}bits}\ \mathit{stampb}]\!]$
$\Longrightarrow \mathit{CanonicalizeSub}\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ a\ y)\ (\mathit{UnaryExpr}\ \mathit{UnaryNeg}\ b)\ |$

sub-right-add2:

$[\![y = (\mathit{BinaryExpr}\ \mathit{BinAdd}\ a\ b);$
$\mathit{stampa} = \mathit{stamp\text{-}expr}\ a;$
$\mathit{stampb} = \mathit{stamp\text{-}expr}\ b;$
$\mathit{is\text{-}IntegerStamp}\ \mathit{stampa} \wedge \mathit{is\text{-}IntegerStamp}\ \mathit{stampb};$
$\mathit{stp\text{-}bits}\ \mathit{stampa} = \mathit{stp\text{-}bits}\ \mathit{stampb}]\!]$
$\Longrightarrow \mathit{CanonicalizeSub}\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ b\ y)\ (\mathit{UnaryExpr}\ \mathit{UnaryNeg}\ a)\ |$

sub-right-sub:

$[\![y = (\mathit{BinaryExpr}\ \mathit{BinSub}\ a\ b);$
$\mathit{stampa} = \mathit{stamp\text{-}expr}\ a;$
$\mathit{stampb} = \mathit{stamp\text{-}expr}\ b;$
$\mathit{is\text{-}IntegerStamp}\ \mathit{stampa} \wedge \mathit{is\text{-}IntegerStamp}\ \mathit{stampb};$
$\mathit{stp\text{-}bits}\ \mathit{stampa} = \mathit{stp\text{-}bits}\ \mathit{stampb}]\!]$
$\Longrightarrow \mathit{CanonicalizeSub}\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ a\ y)\ b\ |$

sub-xzero32:
$[\![stampx = stamp\text{-}expr\ x;$
$stampx = IntegerStamp\ 32\ lo\ hi]\!]$
$\Longrightarrow$ *CanonicalizeSub* (*BinaryExpr BinSub* (*ConstantExpr* (*IntVal32 0*)) *x*) (*UnaryExpr UnaryNeg x*) |
sub-xzero64:
$[\![stampx = stamp\text{-}expr\ x;$
$stampx = IntegerStamp\ 64\ lo\ hi]\!]$
$\Longrightarrow$ *CanonicalizeSub* (*BinaryExpr BinSub* (*ConstantExpr* (*IntVal64 0*)) *x*) (*UnaryExpr UnaryNeg x*) |

sub-y-negate:

$[\![nb = (UnaryExpr\ UnaryNeg\ b);$
$stampa = stamp\text{-}expr\ a;$
$stampb = stamp\text{-}expr\ b;$
$is\text{-}IntegerStamp\ stampa \wedge is\text{-}IntegerStamp\ stampb;$
$stp\text{-}bits\ stampa = stp\text{-}bits\ stampb]\!]$
$\Longrightarrow$ *CanonicalizeSub* (*BinaryExpr BinSub a nb*) (*BinaryExpr BinAdd a b*)

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

$[\![nx = (UnaryExpr\ UnaryNeg\ x);$
$is\text{-}IntegerStamp\ (stamp\text{-}expr\ x)]\!]$
$\Longrightarrow$ *CanonicalizeNegate* (*UnaryExpr UnaryNeg nx*) *x* |

negate-sub:

$[\![e = (BinaryExpr\ BinSub\ x\ y);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy]\!]$
$\Longrightarrow$ *CanonicalizeNegate* (*UnaryExpr UnaryNeg e*) (*BinaryExpr BinSub y x*)

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

$[\![ax = (UnaryExpr\ UnaryAbs\ x);$
$is\text{-}IntegerStamp\ (stamp\text{-}expr\ x)]\!]$
$\Longrightarrow$ *CanonicalizeAbs* (*UnaryExpr UnaryAbs ax*) *ax* |

abs-neg:

$[\![nx = (UnaryExpr\ UnaryNeg\ x);$
$is\text{-}IntegerStamp\ (stamp\text{-}expr\ x)]\!]$
$\Longrightarrow CanonicalizeAbs\ (UnaryExpr\ UnaryAbs\ nx)\ (UnaryExpr\ UnaryAbs\ x)$

inductive *CanonicalizeNot* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**
not-not:

$[\![nx = (UnaryExpr\ UnaryNot\ x);$
$is\text{-}IntegerStamp\ (stamp\text{-}expr\ x)]\!]$
$\Longrightarrow CanonicalizeNot\ (UnaryExpr\ UnaryNot\ nx)\ x$

inductive *CanonicalizeAnd* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**
and-same:

$[\![is\text{-}IntegerStamp\ (stamp\text{-}expr\ x)]\!]$
$\Longrightarrow CanonicalizeAnd\ (BinaryExpr\ BinAnd\ x\ x)\ x\ |$

and-demorgans:

$[\![nx = (UnaryExpr\ UnaryNot\ x);$
$ny = (UnaryExpr\ UnaryNot\ y);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy]\!]$
$\Longrightarrow CanonicalizeAnd\ (BinaryExpr\ BinAnd\ nx\ ny)\ (UnaryExpr\ UnaryNot\ (BinaryExpr\ BinOr\ x\ y))$

inductive *CanonicalizeOr* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**
or-same:

$[\![is\text{-}IntegerStamp\ (stamp\text{-}expr\ x)]\!]$
$\Longrightarrow CanonicalizeOr\ (BinaryExpr\ BinOr\ x\ x)\ x\ |$

or-demorgans:

$[\![nx = (UnaryExpr\ UnaryNot\ x);$
$ny = (UnaryExpr\ UnaryNot\ y);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy]\!]$
$\Longrightarrow CanonicalizeOr\ (BinaryExpr\ BinOr\ nx\ ny)\ (UnaryExpr\ UnaryNot\ (BinaryExpr\ BinAnd\ x\ y))$

inductive *CanonicalizeIntegerEquals* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**
int-equals-same:

⟦$x = y$⟧
⟹ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals x y*) (*ConstantExpr* (*IntVal32 1*)) |

int-equals-distinct:
⟦*alwaysDistinct* (*stamp-expr x*) (*stamp-expr y*)⟧
⟹ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals x y*) (*ConstantExpr* (*IntVal32 0*)) |

int-equals-add-first-both-same:

⟦*left* = (*BinaryExpr BinAdd x y*);
right = (*BinaryExpr BinAdd x z*)⟧
⟹ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left right*) (*BinaryExpr BinIntegerEquals y z*) |

int-equals-add-first-second-same:

⟦*left* = (*BinaryExpr BinAdd x y*);
right = (*BinaryExpr BinAdd z x*)⟧
⟹ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left right*) (*BinaryExpr BinIntegerEquals y z*) |

int-equals-add-second-first-same:

⟦*left* = (*BinaryExpr BinAdd y x*);
right = (*BinaryExpr BinAdd x z*)⟧
⟹ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left right*) (*BinaryExpr BinIntegerEquals y z*) |

int-equals-add-second-both--same:

⟦*left* = (*BinaryExpr BinAdd y x*);
right = (*BinaryExpr BinAdd z x*)⟧
⟹ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left right*) (*BinaryExpr BinIntegerEquals y z*) |

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

⟦*left* = (*BinaryExpr BinSub x y*);
right = (*BinaryExpr BinSub x z*)⟧
⟹ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left right*) (*BinaryExpr

$BinIntegerEquals\ y\ z) \mid$

$int\text{-}equals\text{-}sub\text{-}second\text{-}both\text{-}same$:

$[\![left = (BinaryExpr\ BinSub\ y\ x);$
$right = (BinaryExpr\ BinSub\ z\ x)]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ right)\ (BinaryExpr\ BinIntegerEquals\ y\ z) \mid$

$int\text{-}equals\text{-}left\text{-}contains\text{-}right1$:

$[\![left = (BinaryExpr\ BinAdd\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ x)\ (BinaryExpr\ BinIntegerEquals\ y\ zero) \mid$

$int\text{-}equals\text{-}left\text{-}contains\text{-}right2$:

$[\![left = (BinaryExpr\ BinAdd\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ y)\ (BinaryExpr\ BinIntegerEquals\ x\ zero) \mid$

$int\text{-}equals\text{-}right\text{-}contains\text{-}left1$:

$[\![right = (BinaryExpr\ BinAdd\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ x\ right)\ (BinaryExpr\ BinIntegerEquals\ y\ zero) \mid$

$int\text{-}equals\text{-}right\text{-}contains\text{-}left2$:

$[\![right = (BinaryExpr\ BinAdd\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ y\ right)\ (BinaryExpr\ BinIntegerEquals\ x\ zero) \mid$

$int\text{-}equals\text{-}left\text{-}contains\text{-}right3$:

$[\![left = (BinaryExpr\ BinSub\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ x)\ (BinaryExpr\ BinIntegerEquals\ y\ zero) \mid$

$int\text{-}equals\text{-}right\text{-}contains\text{-}left3$:

$[\![right = (BinaryExpr\ BinSub\ x\ y);$

$zero = (ConstantExpr\ (IntVal32\ 0))]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ x\ right)\ (BinaryExpr\ BinIntegerEquals\ y\ zero)$

inductive $CanonicalizeConditional :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ **where**
$eq\text{-}branches$:

$[\![t = f]\!]$
$\Longrightarrow CanonicalizeConditional\ (ConditionalExpr\ c\ t\ f)\ t\ |$

$cond\text{-}eq$:

$[\![c = (BinaryExpr\ BinIntegerEquals\ x\ y);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy]\!]$
$\Longrightarrow CanonicalizeConditional\ (ConditionalExpr\ c\ x\ y)\ y\ |$

$condition\text{-}bounds\text{-}x$:

$[\![c = (BinaryExpr\ BinIntegerLessThan\ x\ y);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$stpi\text{-}upper\ stampx \leq stpi\text{-}lower\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy]\!]$
$\Longrightarrow CanonicalizeConditional\ (ConditionalExpr\ c\ x\ y)\ x\ |$

$condition\text{-}bounds\text{-}y$:

$[\![c = (BinaryExpr\ BinIntegerLessThan\ x\ y);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$stpi\text{-}upper\ stampx \leq stpi\text{-}lower\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy]\!]$
$\Longrightarrow CanonicalizeConditional\ (ConditionalExpr\ c\ y\ x)\ y\ |$

$negate\text{-}condition$:

$[\![nc = (UnaryExpr\ UnaryLogicNegation\ c);$
$stampc = stamp\text{-}expr\ c;$
$stampc = IntegerStamp\ 32\ lo\ hi;$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$

$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy]\!]$
$\Longrightarrow CanonicalizeConditional\ (ConditionalExpr\ nc\ x\ y)\ (ConditionalExpr\ c\ y\ x)$
|

const-true:

$[\![c = ConstantExpr\ val;$
$val\text{-}to\text{-}bool\ val]\!]$
$\Longrightarrow CanonicalizeConditional\ (ConditionalExpr\ c\ t\ f)\ t$ |

const-false:

$[\![c = ConstantExpr\ val;$
$\neg(val\text{-}to\text{-}bool\ val)]\!]$
$\Longrightarrow CanonicalizeConditional\ (ConditionalExpr\ c\ t\ f)\ f$

inductive *CanonicalizationStep* :: *IRExpr* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* **where**
BinaryNode:
$[\![CanonicalizeBinaryOp\ expr\ expr']\!]$
$\Longrightarrow CanonicalizationStep\ expr\ expr'$ |

UnaryNode:
$[\![CanonicalizeUnaryOp\ expr\ expr']\!]$
$\Longrightarrow CanonicalizationStep\ expr\ expr'$ |

NegateNode:
$[\![CanonicalizeNegate\ expr\ expr']\!]$
$\Longrightarrow CanonicalizationStep\ expr\ expr'$ |

NotNode:
$[\![CanonicalizeNegate\ expr\ expr']\!]$
$\Longrightarrow CanonicalizationStep\ expr\ expr'$ |

AddNode:
$[\![CanonicalizeAdd\ expr\ expr']\!]$
$\Longrightarrow CanonicalizationStep\ expr\ expr'$ |

MulNode:
$[\![CanonicalizeMul\ expr\ expr']\!]$
$\Longrightarrow CanonicalizationStep\ expr\ expr'$ |

SubNode:
⟦*CanonicalizeSub expr expr'*⟧
$\Longrightarrow$ *CanonicalizationStep expr expr'* |

AndNode:
⟦*CanonicalizeSub expr expr'*⟧
$\Longrightarrow$ *CanonicalizationStep expr expr'* |

OrNode:
⟦*CanonicalizeSub expr expr'*⟧
$\Longrightarrow$ *CanonicalizationStep expr expr'* |

IntegerEqualsNode:
⟦*CanonicalizeIntegerEquals expr expr'*⟧
$\Longrightarrow$ *CanonicalizationStep expr expr'* |

ConditionalNode:
⟦*CanonicalizeConditional expr expr'*⟧
$\Longrightarrow$ *CanonicalizationStep expr expr'*

code-pred (*modes*: *i* ⇒ *o* ⇒ *bool*) *CanonicalizeBinaryOp* ⟨*proof*⟩
code-pred (*modes*: *i* ⇒ *o* ⇒ *bool*) *CanonicalizeUnaryOp* ⟨*proof*⟩
code-pred (*modes*: *i* ⇒ *o* ⇒ *bool*) *CanonicalizeNegate* ⟨*proof*⟩
code-pred (*modes*: *i* ⇒ *o* ⇒ *bool*) *CanonicalizeNot* ⟨*proof*⟩
code-pred (*modes*: *i* ⇒ *o* ⇒ *bool*) *CanonicalizeAdd* ⟨*proof*⟩
code-pred (*modes*: *i* ⇒ *o* ⇒ *bool*) *CanonicalizeSub* ⟨*proof*⟩
code-pred (*modes*: *i* ⇒ *o* ⇒ *bool*) *CanonicalizeMul* ⟨*proof*⟩
code-pred (*modes*: *i* ⇒ *o* ⇒ *bool*) *CanonicalizeAnd* ⟨*proof*⟩
code-pred (*modes*: *i* ⇒ *o* ⇒ *bool*) *CanonicalizeIntegerEquals* ⟨*proof*⟩
code-pred (*modes*: *i* ⇒ *o* ⇒ *bool*) *CanonicalizeConditional* ⟨*proof*⟩

code-pred (*modes*: *i* ⇒ *o* ⇒ *bool*) *CanonicalizationStep* ⟨*proof*⟩

end

12 Canonicalization Phase

theory *CanonicalizationTreeProofs*
imports
CanonicalizationTree
Semantics.TreeToGraph
Semantics.IRTreeEvalThms
begin

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

and *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-add x* (*IntVal32* (*0*)) = *x*

and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-add x* (*IntVal64* (*0*)) = *x*

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

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

and *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-or x* (*IntVal32* (*0*)) = *x*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-or x* (*IntVal64* (*0*)) = *x*
⟨*proof*⟩

lemma *annihilator-rewrite-helper*:
shows *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-mul x* (*IntVal32 0*) = *IntVal32 0*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-mul x* (*IntVal64 0*) = *IntVal64 0*

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

and *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-or x* (*IntVal32* (−*1*)) = *IntVal32* (−*1*)
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-or x* (*IntVal64* (−*1*)) = *IntVal64* (−*1*)
⟨*proof*⟩

lemma *idempotent-rewrite-helper*:
shows *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-and x x* = *x*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-and x x* = *x*

and *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-or x x* = *x*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-or x x* = *x*
⟨*proof*⟩

value *size* (*v*::*32 word*)

lemma *signed-int-bottom32*: −(((*2*::*int*) ^ *31*)) ≤ *sint* (*v*::*int32*)
⟨*proof*⟩

lemma *signed-int-top32*: (*2* ^ *31*) − *1* ≥ *sint* (*v*::*int32*)
⟨*proof*⟩

lemma *lower-bounds-equiv32*: −(((*2*::*int*) ^ *31*)) = (*2*::*int*) ^ *32 div 2* ∗ − *1*
⟨*proof*⟩

lemma *upper-bounds-equiv32*: (*2*::*int*) ^ *31* = (*2*::*int*) ^ *32 div 2*

⟨proof⟩

lemma *bit-bounds-min32*: $((fst\ (bit\text{-}bounds\ 32))) \leq (sint\ (v::int32))$
⟨proof⟩

lemma *bit-bounds-max32*: $((snd\ (bit\text{-}bounds\ 32))) \geq (sint\ (v::int32))$
⟨proof⟩

value *size* $(v::64\ word)$

lemma *signed-int-bottom64*: $-(((2::int)\ \widehat{}\ 63)) \leq sint\ (v::int64)$
⟨proof⟩

lemma *signed-int-top64*: $(2\ \widehat{}\ 63) - 1 \geq sint\ (v::int64)$
⟨proof⟩

lemma *lower-bounds-equiv64*: $-(((2::int)\ \widehat{}\ 63)) = (2::int)\ \widehat{}\ 64\ div\ 2 * - 1$
⟨proof⟩

lemma *upper-bounds-equiv64*: $(2::int)\ \widehat{}\ 63 = (2::int)\ \widehat{}\ 64\ div\ 2$
⟨proof⟩

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

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

lemma *unrestricted-32bit-always-valid*:
valid-value (*unrestricted-stamp* (*IntegerStamp 32 lo hi*)) (*IntVal32 v*)
⟨proof⟩

lemma *unrestricted-64bit-always-valid*:
valid-value (*unrestricted-stamp* (*IntegerStamp 64 lo hi*)) (*IntVal64 v*)
⟨proof⟩

lemma *unary-undef*: *val* = *UndefVal* ⟹ *unary-eval op val* = *UndefVal*
⟨proof⟩

lemma *unary-obj*: *val* = *ObjRef x* ⟹ *unary-eval op val* = *UndefVal*
⟨proof⟩

lemma *unary-eval-implies-valud-value*:
assumes $[m,p] \vdash expr \mapsto val$
assumes *result* = *unary-eval op val*
assumes *result* ≠ *UndefVal*
assumes *valid-value* (*stamp-expr expr*) *val*
shows *valid-value* (*stamp-expr* (*UnaryExpr op expr*)) *result*
⟨proof⟩

lemma *binary-undef*: $v1 = UndefVal \lor v2 = UndefVal \Longrightarrow bin\text{-}eval\ op\ v1\ v2 = UndefVal$
⟨*proof*⟩

lemma *binary-obj*: $v1 = ObjRef\ x \lor v2 = ObjRef\ y \Longrightarrow bin\text{-}eval\ op\ v1\ v2 = UndefVal$
⟨*proof*⟩

lemma *binary-eval-bits-equal*:
assumes $result = bin\text{-}eval\ op\ val1\ val2$
assumes $result \neq UndefVal$
assumes $valid\text{-}value\ (IntegerStamp\ b1\ lo1\ hi1)\ val1$
assumes $valid\text{-}value\ (IntegerStamp\ b2\ lo2\ hi2)\ val2$
shows $b1 = b2$
⟨*proof*⟩

lemma *binary-eval-values*:
assumes $\exists\, x\ y.\ result = IntVal32\ x \lor result = IntVal64\ y$
assumes $result = bin\text{-}eval\ op\ val1\ val2$
shows $\exists\, x32\ x64\ y32\ y64.\ val1 = IntVal32\ x32 \land val2 = IntVal32\ y32 \lor val1 = IntVal64\ x64 \land val2 = IntVal64\ y64$
⟨*proof*⟩

lemma *binary-eval-implies-valud-value*:
assumes $[m,p] \vdash expr1 \mapsto val1$
assumes $[m,p] \vdash expr2 \mapsto val2$
assumes $result = bin\text{-}eval\ op\ val1\ val2$
assumes $result \neq UndefVal$
assumes $valid\text{-}value\ (stamp\text{-}expr\ expr1)\ val1$
assumes $valid\text{-}value\ (stamp\text{-}expr\ expr2)\ val2$
shows $valid\text{-}value\ (stamp\text{-}expr\ (BinaryExpr\ op\ expr1\ expr2))\ result$
⟨*proof*⟩

lemma *stamp-meet-is-valid*:
assumes $valid\text{-}value\ stamp1\ val \lor valid\text{-}value\ stamp2\ val$
assumes $meet\ stamp1\ stamp2 \neq IllegalStamp$
shows $valid\text{-}value\ (meet\ stamp1\ stamp2)\ val$
⟨*proof*⟩

lemma *conditional-eval-implies-valud-value*:
assumes $[m,p] \vdash cond \mapsto condv$
assumes $expr = (if\ IRTreeEval.val\text{-}to\text{-}bool\ condv\ then\ expr1\ else\ expr2)$
assumes $[m,p] \vdash expr \mapsto val$
assumes $val \neq UndefVal$
assumes $valid\text{-}value\ (stamp\text{-}expr\ cond)\ condv$
assumes $valid\text{-}value\ (stamp\text{-}expr\ expr)\ val$
shows $valid\text{-}value\ (stamp\text{-}expr\ (ConditionalExpr\ cond\ expr1\ expr2))\ val$

$\langle proof \rangle$

lemma *stamp-implies-valid-value*:
  assumes $[m,p] \vdash expr \mapsto val$
  shows *valid-value* (*stamp-expr expr*) *val*
  $\langle proof \rangle$

lemma *CanonicalizeBinaryProof*:
  assumes *CanonicalizeBinaryOp before after*
  assumes $[m,\ p] \vdash before \mapsto res$
  assumes $[m,\ p] \vdash after \mapsto res'$
  shows $res = res'$
  $\langle proof \rangle$

lemma *CanonicalizeUnaryProof*:
  assumes *CanonicalizeUnaryOp before after*
  assumes $[m,\ p] \vdash before \mapsto res$
  assumes $[m,\ p] \vdash after \mapsto res'$
  shows $res = res'$
  $\langle proof \rangle$

lemma *mul-rewrite-helper*:
  shows *valid-value* (*IntegerStamp 32 lo hi*) $x \Longrightarrow$ *intval-mul* x (*IntVal32* (-1)) = *intval-negate* x
  and *valid-value* (*IntegerStamp 64 lo hi*) $x \Longrightarrow$ *intval-mul* x (*IntVal64* (-1)) = *intval-negate* x
  $\langle proof \rangle$

lemma *CanonicalizeMulProof*:
  assumes *CanonicalizeMul before after*
  assumes $[m,\ p] \vdash before \mapsto res$
  assumes $[m,\ p] \vdash after \mapsto res'$
  shows $res = res'$
  $\langle proof \rangle$

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*
  $\langle proof \rangle$

lemma *CanonicalizeAddProof*:
  assumes *CanonicalizeAdd before after*
  assumes $[m, p] \vdash before \mapsto res$
  assumes $[m, p] \vdash after \mapsto res'$
  shows $res = res'$
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *sub-single-rewrites-helper*:
  assumes *valid-value* (*IntegerStamp b lox hix*) *x*
  shows $b = 32 \Longrightarrow$ *intval-sub x x* = *IntVal32 0*
  and $b = 64 \Longrightarrow$ *intval-sub x x* = *IntVal64 0*
  and $b = 32 \Longrightarrow$ *intval-sub* (*IntVal32 0*) *x* = *intval-negate x*
  and $b = 64 \Longrightarrow$ *intval-sub* (*IntVal64 0*) *x* = *intval-negate x*
  ⟨*proof*⟩

lemma *CanonicalizeSubProof*:
  assumes *CanonicalizeSub before after*
  assumes $[m, p] \vdash before \mapsto res$
  assumes $[m, p] \vdash after \mapsto res'$
  shows $res = res'$
  ⟨*proof*⟩

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*

⟨*proof*⟩

lemma *negate-negate-helper*:
assumes *valid-value* (*IntegerStamp b lox hix*) *x*
shows *intval-negate* (*intval-negate x*) = *x*
⟨*proof*⟩

lemma *CanonicalizeNegateProof*:
assumes *CanonicalizeNegate before after*
assumes $[m, p] \vdash$ *before* $\mapsto$ *res*
assumes $[m, p] \vdash$ *after* $\mapsto$ *res'*
shows *res* = *res'*
⟨*proof*⟩

lemma *word-helper*:
shows $\bigwedge x :: 32\ word.\ \neg(-\ x <s\ 0 \wedge x <s\ 0)$
and $\bigwedge x :: 64\ word.\ \neg(-\ x <s\ 0 \wedge x <s\ 0)$
and $\bigwedge x :: 32\ word.\ \neg -\ x <s\ 0 \wedge \neg\ x <s\ 0 \Longrightarrow 2 * x = 0$
and $\bigwedge x :: 64\ word.\ \neg -\ x <s\ 0 \wedge \neg\ x <s\ 0 \Longrightarrow 2 * x = 0$
⟨*proof*⟩

lemma *abs-abs-is-abs*:
assumes *valid-value* (*IntegerStamp b lox hix*) *x*
shows *intval-abs* (*intval-abs x*) = *intval-abs x*
⟨*proof*⟩

lemma *abs-neg-is-neg*:
assumes *valid-value* (*IntegerStamp b lox hix*) *x*
shows *intval-abs* (*intval-negate x*) = *intval-abs x*
⟨*proof*⟩

lemma *not-rewrite-helper*:
assumes *valid-value* (*IntegerStamp b lox hix*) *x*
shows *intval-not* (*intval-not x*) = *x*
⟨*proof*⟩

lemma *CanonicalizeNotProof*:
assumes *CanonicalizeNot before after*
assumes $[m, p] \vdash$ *before* $\mapsto$ *res*
assumes $[m, p] \vdash$ *after* $\mapsto$ *res'*
shows *res* = *res'*
⟨*proof*⟩

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 $x = y \Longrightarrow$ *intval-and x y* = *x*
and $x = y \Longrightarrow$ *intval-or x y* = *x*
⟨*proof*⟩

lemma *CanonicalizeAndProof*:
assumes *CanonicalizeAnd before after*
assumes [*m*, *p*] ⊢ *before* ↦ *res*
assumes [*m*, *p*] ⊢ *after* ↦ *res′*
shows *res* = *res′*
⟨*proof*⟩

lemma *CanonicalizeOrProof*:
assumes *CanonicalizeOr before after*
assumes [*m*, *p*] ⊢ *before* ↦ *res*
assumes [*m*, *p*] ⊢ *after* ↦ *res′*
shows *res* = *res′*
⟨*proof*⟩

lemma *stamps-touch-but-not-less-than-implies-equal*:
⟦*valid-value stampx x*;
valid-value stampy y;
is-IntegerStamp stampx ∧ *is-IntegerStamp stampy*;
stpi-upper stampx = *stpi-lower stampy*;
¬ *val-to-bool* (*intval-less-than x y*)⟧ ⟹ *x* = *y*
⟨*proof*⟩

lemma *disjoint-stamp-implies-less-than*:
⟦*valid-value stampx x*;
valid-value stampy y;
is-IntegerStamp stampx ∧ *is-IntegerStamp stampy*;
stpi-upper stampx < *stpi-lower stampy*⟧
⟹ *val-to-bool*(*intval-less-than x y*)
⟨*proof*⟩

lemma *CanonicalizeConditionalProof*:
assumes *CanonicalizeConditional before after*
assumes [*m*, *p*] ⊢ *before* ↦ *res*
assumes [*m*, *p*] ⊢ *after* ↦ *res′*
shows *res* = *res′*
⟨*proof*⟩

end