Veriopt

July 3, 2021

Abstract

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

Contents

1 Runtime Values and Arithmetic

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.

type-synonym *objref* = *nat option*

datatype *Value* =
  UndefVal |
  IntVal (*v-bits*: *int*) (*v-int*: *int*) |
  FloatVal (*v-bits*: *int*) (*v-float*: *float*) |
  ObjRef objref |
  ObjStr string

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

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**
  $fits\text{-}into\text{-}n\ b\ val = ((-(2\hat{\ }(b-1)) \leq val) \wedge (val < (2\hat{\ }(b-1))))$

definition *int-bits-allowed* :: *int set* **where**
int-bits-allowed = {*32*}

fun *wf-value* :: *Value* ⇒ *bool* **where**
wf-value (*IntVal b v*) =
(*b* ∈ *int-bits-allowed* ∧
(*nat b* = *1* ⟶ (*v* = *0* ∨ *v* = *1*)) ∧
(*nat b* > *1* ⟶ *fits-into-n* (*nat b*) *v*)) |
wf-value - = *True*

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

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

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

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

The following collection of intval functions correspond to the JVM arithmetic operations.

fun *intval-add* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-add (*IntVal b1 v1*) (*IntVal b2 v2*) =
(*if b1* ≤ *32* ∧ *b2* ≤ *32*
then (*IntVal 32* (*sint*((*word-of-int v1* :: *int32*) + (*word-of-int v2* :: *int32*))))
else (*IntVal 64* (*sint*((*word-of-int v1* :: *int64*) + (*word-of-int v2* :: *int64*))))) |
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 (*IntVal b1 v1*) (*IntVal b2 v2*) =
(*if b1* ≤ *32* ∧ *b2* ≤ *32*
then (*IntVal 32* (*sint*((*word-of-int v1* :: *int32*) − (*word-of-int v2* :: *int32*))))
else (*IntVal 64* (*sint*((*word-of-int v1* :: *int64*) − (*word-of-int v2* :: *int64*))))) |

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 (*IntVal b1 v1*) (*IntVal b2 v2*) =
(*if b1* ≤ *32* ∧ *b2* ≤ *32*
then (*IntVal 32* (*sint*((*word-of-int v1* :: *int32*) ∗ (*word-of-int v2* :: *int32*))))
else (*IntVal 64* (*sint*((*word-of-int v1* :: *int64*) ∗ (*word-of-int v2* :: *int64*))))) |
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 (*IntVal b1 v1*) (*IntVal b2 v2*) =
(*if b1* ≤ *32* ∧ *b2* ≤ *32*
then (*IntVal 32* (*sint*((*word-of-int*(*v1 sdiv v2*) :: *int32*))))
else (*IntVal 64* (*sint*((*word-of-int*(*v1 sdiv v2*) :: *int64*))))) |
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 (*IntVal b1 v1*) (*IntVal b2 v2*) =
(*if b1* ≤ *32* ∧ *b2* ≤ *32*

then (*IntVal 32* (*sint*((*word-of-int*(*v1 smod v2*) :: *int32*))))
else (*IntVal 64* (*sint*((*word-of-int*(*v1 smod v2*) :: *int64*))))) |
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 (*IntVal b1 v1*) (*IntVal b2 v2*) =
(*if b1* ≤ *32* ∧ *b2* ≤ *32*
then (*IntVal 32* (*sint*((*word-of-int v1* :: *int32*) *AND* (*word-of-int v2* :: *int32*))))
else (*IntVal 64* (*sint*((*word-of-int v1* :: *int64*) *AND* (*word-of-int v2* :: *int64*)))))
|
intval-and - - = *UndefVal*

fun *intval-or* :: *Value* ⇒ *Value* ⇒ *Value* (**infix** ||∗ *59*) **where**
intval-or (*IntVal b1 v1*) (*IntVal b2 v2*) =
(*if b1* ≤ *32* ∧ *b2* ≤ *32*
then (*IntVal 32* (*sint*((*word-of-int v1* :: *int32*) *OR* (*word-of-int v2* :: *int32*))))
else (*IntVal 64* (*sint*((*word-of-int v1* :: *int64*) *OR* (*word-of-int v2* :: *int64*)))))
|
intval-or - - = *UndefVal*

fun *intval-xor* :: *Value* ⇒ *Value* ⇒ *Value* (**infix** ^∗ *59*) **where**
intval-xor (*IntVal b1 v1*) (*IntVal b2 v2*) =
(*if b1* ≤ *32* ∧ *b2* ≤ *32*
then (*IntVal 32* (*sint*((*word-of-int v1* :: *int32*) *XOR* (*word-of-int v2* :: *int32*))))
else (*IntVal 64* (*sint*((*word-of-int v1* :: *int64*) *XOR* (*word-of-int v2* :: *int64*)))))
|
intval-xor - - = *UndefVal*

fun *intval-not* :: *Value* ⇒ *Value* **where**
intval-not (*IntVal b v*) =
(*if b* ≤ *32*
then (*IntVal 32* (*sint*(*NOT* (*word-of-int v* :: *int32*))))
else (*IntVal 64* (*sint*(*NOT* (*word-of-int v* :: *int64*))))) |
intval-not - = *UndefVal*

lemma *intval-add-bits*:
  assumes *b*: *IntVal b res* = *intval-add x y*
  shows *b* = *32* ∨ *b* = *64*
⟨*proof*⟩

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-sym1*:
  shows *intval-add* (*IntVal b1 v1*) (*IntVal b2 v2*) = *intval-add* (*IntVal b2 v2*) (*IntVal b1 v1*)
  ⟨*proof*⟩

lemma *intval-add-sym*:
  shows *intval-add x y* = *intval-add y x*
  ⟨*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 *wf-int32*:
  assumes *wf*: *wf-value* (*IntVal b v*)
  shows *b* = *32*
⟨*proof*⟩

lemma *wf-int* [*simp*]:
  assumes *wf*: *wf-value* (*IntVal w n*)
  assumes *notbool*: *w* = *32*
  shows *sint*((*word-of-int n*) :: *int32*) = *n*
  ⟨*proof*⟩

lemma *add32-0*:
  assumes *z*:*wf-value* (*IntVal 32 0*)
  assumes *b*:*wf-value* (*IntVal 32 b*)
  shows *intval-add* (*IntVal 32 0*) (*IntVal 32 b*) = (*IntVal 32* (*b*))

$\langle proof \rangle$

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

lemma *intval-add* (*IntVal 32* (2^$31-1$)) (*IntVal 32* (2^$31-1$)) = *IntVal 32* (-2)
$\langle proof \rangle$
lemma *intval-add* (*IntVal 64* (2^$31-1$)) (*IntVal 32* (2^$31-1$)) = *IntVal 64 4294967294*
$\langle proof \rangle$

end

2 Nodes

2.1 Types of Nodes

theory *IRNodes*
imports
Values2
begin

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

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

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

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

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

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

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

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

option)
| *ShortCircuitOrNode* (*ir-x*: *INPUT-COND*) (*ir-y*: *INPUT-COND*)
| *SignedDivNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *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*)
| *UnwindNode* (*ir-exception*: *INPUT*)
| *ValuePhiNode* (*ir-nid*: *ID*) (*ir-values*: *INPUT list*) (*ir-merge*: *INPUT-ASSOC*)
| *ValueProxyNode* (*ir-value*: *INPUT*) (*ir-loopExit*: *INPUT-ASSOC*)
| *XorNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *NoNode*

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

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

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

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

fun *inputs-of* :: *IRNode* ⇒ *ID list* **where**
inputs-of-AbsNode:
inputs-of (*AbsNode value*) = [*value*] |
inputs-of-AddNode:
inputs-of (*AddNode x y*) = [*x*, *y*] |
inputs-of-AndNode:
inputs-of (*AndNode x y*) = [*x*, *y*] |
inputs-of-BeginNode:
inputs-of (*BeginNode next*) = [] |
inputs-of-BytecodeExceptionNode:
inputs-of (*BytecodeExceptionNode arguments stateAfter next*) = *arguments* @ (*opt-to-list stateAfter*) |
inputs-of-ConditionalNode:
inputs-of (*ConditionalNode condition trueValue falseValue*) = [*condition*, *true-*

Value, *falseValue*] |
inputs-of-ConstantNode:
inputs-of (*ConstantNode const*) = [] |
inputs-of-DynamicNewArrayNode:
inputs-of (*DynamicNewArrayNode elementType length0 voidClass stateBefore next*) = [*elementType*, *length0*] @ (*opt-to-list voidClass*) @ (*opt-to-list stateBefore*) |
inputs-of-EndNode:
inputs-of (*EndNode*) = [] |
inputs-of-ExceptionObjectNode:
inputs-of (*ExceptionObjectNode stateAfter next*) = (*opt-to-list stateAfter*) |
inputs-of-FrameState:
inputs-of (*FrameState monitorIds outerFrameState values virtualObjectMappings*) = *monitorIds* @ (*opt-to-list outerFrameState*) @ (*opt-list-to-list values*) @ (*opt-list-to-list virtualObjectMappings*) |
inputs-of-IfNode:
inputs-of (*IfNode condition trueSuccessor falseSuccessor*) = [*condition*] |
inputs-of-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-LoadFieldNode:
inputs-of (*LoadFieldNode nid0 field object next*) = (*opt-to-list object*) |
inputs-of-LogicNegationNode:
inputs-of (*LogicNegationNode value*) = [*value*] |
inputs-of-LoopBeginNode:
inputs-of (*LoopBeginNode ends overflowGuard stateAfter next*) = *ends* @ (*opt-to-list overflowGuard*) @ (*opt-to-list stateAfter*) |
inputs-of-LoopEndNode:
inputs-of (*LoopEndNode loopBegin*) = [*loopBegin*] |
inputs-of-LoopExitNode:
inputs-of (*LoopExitNode loopBegin stateAfter next*) = *loopBegin* # (*opt-to-list stateAfter*) |
inputs-of-MergeNode:
inputs-of (*MergeNode ends stateAfter next*) = *ends* @ (*opt-to-list stateAfter*) |
inputs-of-MethodCallTargetNode:
inputs-of (*MethodCallTargetNode targetMethod arguments*) = *arguments* |

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

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

fun *successors-of* :: *IRNode* ⇒ *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-IntegerEqualsNode:
successors-of (*IntegerEqualsNode x y*) = [] |
successors-of-IntegerLessThanNode:
successors-of (*IntegerLessThanNode x y*) = [] |
successors-of-InvokeNode:
successors-of (*InvokeNode nid0 callTarget classInit stateDuring stateAfter next*) = [*next*] |
successors-of-InvokeWithExceptionNode:
successors-of (*InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter next exceptionEdge*) = [*next*, *exceptionEdge*] |
successors-of-IsNullNode:
successors-of (*IsNullNode value*) = [] |
successors-of-KillingBeginNode:
successors-of (*KillingBeginNode next*) = [*next*] |
successors-of-LoadFieldNode:
successors-of (*LoadFieldNode nid0 field object next*) = [*next*] |
successors-of-LogicNegationNode:
successors-of (*LogicNegationNode value*) = [] |
successors-of-LoopBeginNode:
successors-of (*LoopBeginNode ends overflowGuard stateAfter next*) = [*next*] |

successors-of-LoopEndNode:
successors-of (*LoopEndNode loopBegin*) = [] |
successors-of-LoopExitNode:
successors-of (*LoopExitNode loopBegin stateAfter next*) = [*next*] |
successors-of-MergeNode:
successors-of (*MergeNode ends stateAfter next*) = [*next*] |
successors-of-MethodCallTargetNode:
successors-of (*MethodCallTargetNode targetMethod arguments*) = [] |
successors-of-MulNode:
successors-of (*MulNode x y*) = [] |
successors-of-NegateNode:
successors-of (*NegateNode value*) = [] |
successors-of-NewArrayNode:
successors-of (*NewArrayNode length0 stateBefore next*) = [*next*] |
successors-of-NewInstanceNode:
successors-of (*NewInstanceNode nid0 instanceClass stateBefore next*) = [*next*] |
successors-of-NotNode:
successors-of (*NotNode value*) = [] |
successors-of-OrNode:
successors-of (*OrNode x y*) = [] |
successors-of-ParameterNode:
successors-of (*ParameterNode index*) = [] |
successors-of-PiNode:
successors-of (*PiNode object guard*) = [] |
successors-of-ReturnNode:
successors-of (*ReturnNode result memoryMap*) = [] |
successors-of-ShortCircuitOrNode:
successors-of (*ShortCircuitOrNode x y*) = [] |
successors-of-SignedDivNode:
successors-of (*SignedDivNode nid0 x y zeroCheck stateBefore next*) = [*next*] |
successors-of-SignedRemNode:
successors-of (*SignedRemNode nid0 x y zeroCheck stateBefore next*) = [*next*] |
successors-of-StartNode:
successors-of (*StartNode stateAfter next*) = [*next*] |
successors-of-StoreFieldNode:
successors-of (*StoreFieldNode nid0 field value stateAfter object next*) = [*next*] |
successors-of-SubNode:
successors-of (*SubNode x y*) = [] |
successors-of-UnwindNode:
successors-of (*UnwindNode exception*) = [] |
successors-of-ValuePhiNode:
successors-of (*ValuePhiNode nid0 values merge*) = [] |
successors-of-ValueProxyNode:
successors-of (*ValueProxyNode value loopExit*) = [] |
successors-of-XorNode:
successors-of (*XorNode x y*) = [] |
successors-of-NoNode: *successors-of* (*NoNode*) = [] |

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

lemma *inputs-of* (*FrameState x* (*Some y*) (*Some z*) *None*) = *x* @ [*y*] @ *z*
⟨*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 *IRNodes2*
begin

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

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

These functions have been automatically generated from the compiler.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\textit{is-CallTargetNode } n = ((\textit{is-MethodCallTargetNode } n))$

fun *is-ValueNode* :: *IRNode* ⇒ *bool* **where**
$\textit{is-ValueNode } n = ((\textit{is-CallTargetNode } n) \lor (\textit{is-FixedNode } n) \lor (\textit{is-FloatingNode } n))$

fun *is-VirtualState* :: *IRNode* ⇒ *bool* **where**
$\textit{is-VirtualState } n = ((\textit{is-FrameState } n))$

fun *is-Node* :: *IRNode* ⇒ *bool* **where**
$\textit{is-Node } n = ((\textit{is-ValueNode } n) \lor (\textit{is-VirtualState } n))$

fun *is-MemoryKill* :: *IRNode* ⇒ *bool* **where**
$\textit{is-MemoryKill } n = ((\textit{is-AbstractMemoryCheckpoint } n))$

fun *is-NarrowableArithmeticNode* :: *IRNode* ⇒ *bool* **where**
$\textit{is-NarrowableArithmeticNode } n = ((\textit{is-AbsNode } n) \lor (\textit{is-AddNode } n) \lor (\textit{is-AndNode } n) \lor (\textit{is-MulNode } n) \lor (\textit{is-NegateNode } n) \lor (\textit{is-NotNode } n) \lor (\textit{is-OrNode } n) \lor (\textit{is-SubNode } n) \lor (\textit{is-XorNode } n))$

fun *is-AnchoringNode* :: *IRNode* ⇒ *bool* **where**
$\textit{is-AnchoringNode } n = ((\textit{is-AbstractBeginNode } n))$

fun *is-DeoptBefore* :: *IRNode* ⇒ *bool* **where**
$\textit{is-DeoptBefore } n = ((\textit{is-DeoptimizingFixedWithNextNode } n))$

fun *is-IndirectCanonicalization* :: *IRNode* ⇒ *bool* **where**
$\textit{is-IndirectCanonicalization } n = ((\textit{is-LogicNode } n))$

fun *is-IterableNodeType* :: *IRNode* ⇒ *bool* **where**
$\textit{is-IterableNodeType } n = ((\textit{is-AbstractBeginNode } n) \lor (\textit{is-AbstractMergeNode } n) \lor (\textit{is-FrameState } n) \lor (\textit{is-IfNode } n) \lor (\textit{is-IntegerDivRemNode } n) \lor (\textit{is-InvokeWithExceptionNode } n) \lor (\textit{is-LoopBeginNode } n) \lor (\textit{is-LoopExitNode } n) \lor (\textit{is-MethodCallTargetNode } n) \lor (\textit{is-ParameterNode } n) \lor (\textit{is-ReturnNode } n) \lor (\textit{is-ShortCircuitOrNode } n))$

fun *is-Invoke* :: *IRNode* ⇒ *bool* **where**
$\textit{is-Invoke } n = ((\textit{is-InvokeNode } n) \lor (\textit{is-InvokeWithExceptionNode } n))$

fun *is-Proxy* :: *IRNode* ⇒ *bool* **where**
$\textit{is-Proxy } n = ((\textit{is-ProxyNode } n))$

fun *is-ValueProxy* :: *IRNode* ⇒ *bool* **where**
$\textit{is-ValueProxy } n = ((\textit{is-PiNode } n) \lor (\textit{is-ValueProxyNode } n))$

fun *is-ValueNodeInterface* :: *IRNode* ⇒ *bool* **where**
$\textit{is-ValueNodeInterface } n = ((\textit{is-ValueNode } n))$

fun *is-ArrayLengthProvider* :: *IRNode* ⇒ *bool* **where**
$\textit{is-ArrayLengthProvider } n = ((\textit{is-AbstractNewArrayNode } n) \lor (\textit{is-ConstantNode}$

n))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

fun *is-same-ir-node-type* :: *IRNode* ⇒ *IRNode* ⇒ *bool* **where**
is-same-ir-node-type n1 n2 = (

$((is\text{-}AbsNode\ n1) \wedge (is\text{-}AbsNode\ n2)) \vee$
$((is\text{-}AddNode\ n1) \wedge (is\text{-}AddNode\ n2)) \vee$
$((is\text{-}AndNode\ n1) \wedge (is\text{-}AndNode\ n2)) \vee$
$((is\text{-}BeginNode\ n1) \wedge (is\text{-}BeginNode\ n2)) \vee$
$((is\text{-}BytecodeExceptionNode\ n1) \wedge (is\text{-}BytecodeExceptionNode\ n2)) \vee$
$((is\text{-}ConditionalNode\ n1) \wedge (is\text{-}ConditionalNode\ n2)) \vee$
$((is\text{-}ConstantNode\ n1) \wedge (is\text{-}ConstantNode\ n2)) \vee$
$((is\text{-}DynamicNewArrayNode\ n1) \wedge (is\text{-}DynamicNewArrayNode\ n2)) \vee$
$((is\text{-}EndNode\ n1) \wedge (is\text{-}EndNode\ n2)) \vee$
$((is\text{-}ExceptionObjectNode\ n1) \wedge (is\text{-}ExceptionObjectNode\ n2)) \vee$
$((is\text{-}FrameState\ n1) \wedge (is\text{-}FrameState\ n2)) \vee$
$((is\text{-}IfNode\ n1) \wedge (is\text{-}IfNode\ n2)) \vee$
$((is\text{-}IntegerEqualsNode\ n1) \wedge (is\text{-}IntegerEqualsNode\ n2)) \vee$
$((is\text{-}IntegerLessThanNode\ n1) \wedge (is\text{-}IntegerLessThanNode\ n2)) \vee$
$((is\text{-}InvokeNode\ n1) \wedge (is\text{-}InvokeNode\ n2)) \vee$
$((is\text{-}InvokeWithExceptionNode\ n1) \wedge (is\text{-}InvokeWithExceptionNode\ n2)) \vee$
$((is\text{-}IsNullNode\ n1) \wedge (is\text{-}IsNullNode\ n2)) \vee$
$((is\text{-}KillingBeginNode\ n1) \wedge (is\text{-}KillingBeginNode\ n2)) \vee$
$((is\text{-}LoadFieldNode\ n1) \wedge (is\text{-}LoadFieldNode\ n2)) \vee$
$((is\text{-}LogicNegationNode\ n1) \wedge (is\text{-}LogicNegationNode\ n2)) \vee$
$((is\text{-}LoopBeginNode\ n1) \wedge (is\text{-}LoopBeginNode\ n2)) \vee$
$((is\text{-}LoopEndNode\ n1) \wedge (is\text{-}LoopEndNode\ n2)) \vee$
$((is\text{-}LoopExitNode\ n1) \wedge (is\text{-}LoopExitNode\ n2)) \vee$
$((is\text{-}MergeNode\ n1) \wedge (is\text{-}MergeNode\ n2)) \vee$
$((is\text{-}MethodCallTargetNode\ n1) \wedge (is\text{-}MethodCallTargetNode\ n2)) \vee$
$((is\text{-}MulNode\ n1) \wedge (is\text{-}MulNode\ n2)) \vee$
$((is\text{-}NegateNode\ n1) \wedge (is\text{-}NegateNode\ n2)) \vee$
$((is\text{-}NewArrayNode\ n1) \wedge (is\text{-}NewArrayNode\ n2)) \vee$
$((is\text{-}NewInstanceNode\ n1) \wedge (is\text{-}NewInstanceNode\ n2)) \vee$
$((is\text{-}NotNode\ n1) \wedge (is\text{-}NotNode\ n2)) \vee$
$((is\text{-}OrNode\ n1) \wedge (is\text{-}OrNode\ n2)) \vee$
$((is\text{-}ParameterNode\ n1) \wedge (is\text{-}ParameterNode\ n2)) \vee$
$((is\text{-}PiNode\ n1) \wedge (is\text{-}PiNode\ n2)) \vee$
$((is\text{-}ReturnNode\ n1) \wedge (is\text{-}ReturnNode\ n2)) \vee$
$((is\text{-}ShortCircuitOrNode\ n1) \wedge (is\text{-}ShortCircuitOrNode\ n2)) \vee$
$((is\text{-}SignedDivNode\ n1) \wedge (is\text{-}SignedDivNode\ n2)) \vee$
$((is\text{-}StartNode\ n1) \wedge (is\text{-}StartNode\ n2)) \vee$
$((is\text{-}StoreFieldNode\ n1) \wedge (is\text{-}StoreFieldNode\ n2)) \vee$
$((is\text{-}SubNode\ n1) \wedge (is\text{-}SubNode\ n2)) \vee$
$((is\text{-}UnwindNode\ n1) \wedge (is\text{-}UnwindNode\ n2)) \vee$
$((is\text{-}ValuePhiNode\ n1) \wedge (is\text{-}ValuePhiNode\ n2)) \vee$
$((is\text{-}ValueProxyNode\ n1) \wedge (is\text{-}ValueProxyNode\ n2)) \vee$
$((is\text{-}XorNode\ n1) \wedge (is\text{-}XorNode\ n2)))$

end

3 Stamp Typing

theory *Stamp*

imports *Values2*
begin

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

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

datatype *Stamp* =
VoidStamp
| *IntegerStamp* (*stp-bits*: *nat*) (*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-bounds bits = (((2 ^ *bits*) *div* 2) $*$ -1, ((2 ^ *bits*) *div* 2) $-$ 1)

— A stamp which includes the full range of the type
fun *unrestricted-stamp* :: *Stamp* $\Rightarrow$ *Stamp* **where**
unrestricted-stamp VoidStamp = *VoidStamp* |
unrestricted-stamp (*IntegerStamp bits lower upper*) = (*IntegerStamp bits* (*fst* (*bit-bounds bits*)) (*snd* (*bit-bounds bits*))) |

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

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

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

fun *empty-stamp* :: *Stamp* ⇒ *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*) = (

$if\ ((nn1 \vee nn2) \wedge (an1 \vee an2))$
$then\ (empty\text{-}stamp\ (KlassPointerStamp\ nn1\ an1))$
$else\ (KlassPointerStamp\ (nn1 \vee nn2)\ (an1 \vee an2))$
$)\ |$
$join\ (MethodCountersPointerStamp\ nn1\ an1)\ (MethodCountersPointerStamp\ nn2\ an2) = ($
$if\ ((nn1 \vee nn2) \wedge (an1 \vee an2))$
$then\ (empty\text{-}stamp\ (MethodCountersPointerStamp\ nn1\ an1))$
$else\ (MethodCountersPointerStamp\ (nn1 \vee nn2)\ (an1 \vee an2))$
$)\ |$
$join\ (MethodPointersStamp\ nn1\ an1)\ (MethodPointersStamp\ nn2\ an2) = ($
$if\ ((nn1 \vee nn2) \wedge (an1 \vee an2))$
$then\ (empty\text{-}stamp\ (MethodPointersStamp\ nn1\ an1))$
$else\ (MethodPointersStamp\ (nn1 \vee nn2)\ (an1 \vee an2))$
$)\ |$
$join\ s1\ s2 = IllegalStamp$

— In certain circumstances a stamp provides enough information to evaluate a value as a stamp, the asConstant function converts the stamp to a value where one can be inferred.
fun $asConstant :: Stamp \Rightarrow Value$ **where**
$asConstant\ (IntegerStamp\ b\ l\ h) = (if\ l = h\ then\ IntVal32\ (word\text{-}of\text{-}int\ l)\ else\ UndefVal)\ |$
$asConstant\ \text{-} = UndefVal$

— Determine if two stamps never have value overlaps i.e. their join is empty
fun $alwaysDistinct :: Stamp \Rightarrow Stamp \Rightarrow bool$ **where**
$alwaysDistinct\ stamp1\ stamp2 = is\text{-}stamp\text{-}empty\ (join\ stamp1\ stamp2)$

— Determine if two stamps must always be the same value i.e. two equal constants
fun $neverDistinct :: Stamp \Rightarrow Stamp \Rightarrow bool$ **where**
$neverDistinct\ stamp1\ stamp2 = (asConstant\ stamp1 = asConstant\ stamp2 \wedge asConstant\ stamp1 \neq UndefVal)$

fun $constantAsStamp :: Value \Rightarrow Stamp$ **where**
$constantAsStamp\ (IntVal32\ v) = (IntegerStamp\ 32\ (sint\ v)\ (sint\ v))\ |$

$constantAsStamp\ \text{-} = IllegalStamp$

— Define when a runtime value is valid for a stamp
fun $valid\text{-}value :: Stamp \Rightarrow Value \Rightarrow bool$ **where**
$valid\text{-}value\ (IntegerStamp\ b1\ l\ h)\ (IntVal32\ v) = ((sint\ v \geq l) \wedge (sint\ v \leq h))\ |$

$valid\text{-}value\ (VoidStamp)\ (UndefVal) = True\ |$
$valid\text{-}value\ stamp\ val = False$

— 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*))

notepad
begin
  ⟨*proof*⟩
end

end

4 Graph Representation

theory *IRGraph*
  imports
    IRNodeHierarchy
    Stamp2
    HOL−Library.FSet
    HOL.Relation
begin

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

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

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

setup-lifting *type-definition-IRGraph*

lift-definition *ids* :: *IRGraph* ⇒ *ID set*
  is λ*g*. {*nid* ∈ *dom g* . ∄ *s*. *g nid* = (*Some* (*NoNode*, *s*))} ⟨*proof*⟩

fun *with-default* :: ′*c* ⇒ (′*b* ⇒ ′*c*) ⇒ ((′*a* ⇀ ′*b*) ⇒ ′*a* ⇒ ′*c*) **where**
  with-default def conv = (λ*m k*.
    (*case m k of None* ⇒ *def* | *Some v* ⇒ *conv v*))

lift-definition *kind* :: *IRGraph* ⇒ (*ID* ⇒ *IRNode*)
  is *with-default NoNode fst* ⟨*proof*⟩

lift-definition *stamp* :: *IRGraph* ⇒ *ID* ⇒ *Stamp*
  is *with-default IllegalStamp snd* ⟨*proof*⟩

lift-definition *add-node* :: *ID* ⇒ (*IRNode* × *Stamp*) ⇒ *IRGraph* ⇒ *IRGraph*
  is λ*nid k g*. *if fst k* = *NoNode then g else g*(*nid* ↦ *k*) ⟨*proof*⟩

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

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

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

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

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

code-datatype *irgraph*

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

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

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

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

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

lemma [*code*]: $Rep\text{-}IRGraph\ (irgraph\ m) = map\text{-}of\ (no\text{-}node\ m)$
  $\langle proof \rangle$
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-edges g = ($\bigcup$ *i* ∈ *ids g*. {(*i*,*j*)|*j*. *j* ∈ (*inputs g i*)})
— Find all the nodes in the graph that have nid as an input - the usages of nid
fun *usages* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
usages g nid = {*j*. *j* ∈ *ids g* ∧ (*j*,*nid*) ∈ *input-edges g*}
fun *successor-edges* :: *IRGraph* ⇒ *ID rel* **where**
successor-edges g = ($\bigcup$ *i* ∈ *ids g*. {(*i*,*j*)|*j* . *j* ∈ (*succ g i*)})
fun *predecessors* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
predecessors g nid = {*j*. *j* ∈ *ids g* ∧ (*j*,*nid*) ∈ *successor-edges g*}
fun *nodes-of* :: *IRGraph* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID set* **where**
nodes-of g sel = {*nid* ∈ *ids g* . *sel* (*kind g nid*)}
fun *edge* :: (*IRNode* ⇒ *'a*) ⇒ *ID* ⇒ *IRGraph* ⇒ *'a* **where**
edge sel nid g = *sel* (*kind g nid*)

fun *filtered-inputs* :: *IRGraph* ⇒ *ID* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID list* **where**
filtered-inputs g nid f = *filter* (*f* ∘ (*kind g*)) (*inputs-of* (*kind g nid*))
fun *filtered-successors* :: *IRGraph* ⇒ *ID* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID list* **where**
filtered-successors g nid f = *filter* (*f* ∘ (*kind g*)) (*successors-of* (*kind g nid*))
fun *filtered-usages* :: *IRGraph* ⇒ *ID* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID set* **where**
filtered-usages g nid f = {*n* ∈ (*usages g nid*). *f* (*kind g n*)}

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

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

lemma *ids-some*[*simp*]: *x* ∈ *ids g* ⟷ *kind g x* ≠ *NoNode*
⟨*proof*⟩

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

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

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

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

lemma [*simp*]: *finite* (*dom g*) ⟶ *ids* (*Abs-IRGraph g*) = {*nid* ∈ *dom g* . ∄ *s*. *g nid* = *Some* (*NoNode*, *s*)}
⟨*proof*⟩

lemma [*simp*]: *finite* (*dom g*) ⟶ *kind* (*Abs-IRGraph g*) = (λ*x* . (*case g x of None*

$\Rightarrow$ *NoNode* | *Some n* $\Rightarrow$ *fst n*))
⟨*proof*⟩

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

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

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

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

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

lemma [*code*]: *replace-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* $\longrightarrow$
(*if k* $\neq$ *NoNode then kind gup nid* = *k* $\wedge$ *stamp gup nid* = *s else kind gup nid* = *kind g nid*)
⟨*proof*⟩

lemma *remove-node-lookup*:
gup = *remove-node nid g* $\longrightarrow$ *kind gup nid* = *NoNode* $\wedge$ *stamp gup nid* = *IllegalStamp*
⟨*proof*⟩

lemma *replace-node-lookup*[*simp*]:
gup = *replace-node nid* (*k*, *s*) *g* $\wedge$ *k* $\neq$ *NoNode* $\longrightarrow$ *kind gup nid* = *k* $\wedge$ *stamp gup nid* = *s*
⟨*proof*⟩

lemma *replace-node-unchanged*:
gup = *replace-node nid* (*k*, *s*) *g* $\longrightarrow$ ($\forall$ *n* $\in$ (*ids g* $-$ {*nid*}) . *n* $\in$ *ids g* $\wedge$ *n* $\in$ *ids gup* $\wedge$ *kind g n* = *kind gup n*)
⟨*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 *IREval*
imports
Graph.IRGraph
begin

We define the semantics of data-flow nodes as big-step operational semantics.

Data-flow nodes are evaluated in the context of the *IRGraph* and 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 wihtin 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 part of the control-flow as the data-flow is specified to be side-effect free. As a result, the control-flow semantics for *SignedDivNode* calculates the value of a node and maps the node identifier to the value within the method state. The data-flow semantics then just reads the value stored in the method state for the node.

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

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

fun *find-index* :: *'a* ⇒ *'a list* ⇒ *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 nid =
    (*filter* (λ*x*.(*is-PhiNode* (*kind g x*)))
      (*sorted-list-of-set* (*usages g nid*)))

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

inductive
  eval :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *IRNode* ⇒ *Value* ⇒ *bool* ([-, -, -] ⊢ - ↦ - *55*)
  for *g m p* **where**

  ConstantNode:
  [*g*, *m*, *p*] ⊢ (*ConstantNode c*) ↦ *c* |

  ParameterNode:
  [*g*, *m*, *p*] ⊢ (*ParameterNode i*) ↦ *p*!*i* |

  ValuePhiNode:
  [*g*, *m*, *p*] ⊢ (*ValuePhiNode nid* - -) ↦ *m nid* |

  ValueProxyNode:
  ⟦[*g*, *m*, *p*] ⊢ (*kind g c*) ↦ *val*⟧
    ⟹ [*g*, *m*, *p*] ⊢ (*ValueProxyNode c* -) ↦ *val* |

— Unary arithmetic operators

AbsNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto IntVal32\ v]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (AbsNode\ x) \mapsto if\ v < 0\ then\ (intval\text{-}sub\ (IntVal32\ 0)\ (IntVal32\ v))\ else\ (IntVal32\ v)\ |$

NegateNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto v]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (NegateNode\ x) \mapsto (IntVal32\ 0) - v\ |$

NotNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto v;$
$nv = intval\text{-}not\ v]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (NotNode\ x) \mapsto nv\ |$

— Binary arithmetic operators

AddNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto v1;$
$[g,\ m,\ p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (AddNode\ x\ y) \mapsto v1 + v2\ |$

SubNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto v1;$
$[g,\ m,\ p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (SubNode\ x\ y) \mapsto v1 - v2\ |$

MulNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto v1;$
$[g,\ m,\ p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (MulNode\ x\ y) \mapsto v1 * v2\ |$

SignedDivNode:
$[g,\ m,\ p] \vdash (SignedDivNode\ nid\ \text{- - - - -}) \mapsto m\ nid\ |$

SignedRemNode:
$[g,\ m,\ p] \vdash (SignedRemNode\ nid\ \text{- - - - -}) \mapsto m\ nid\ |$

— Binary logical bitwise operators

AndNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto v1;$
$[g,\ m,\ p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (AndNode\ x\ y) \mapsto intval\text{-}and\ \ v1\ v2\ |$

OrNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto v1;$

$[g, m, p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g, m, p] \vdash (OrNode\ x\ y) \mapsto intval\text{-}or\ v1\ v2 \mid$

XorNode:
$[\![[g, m, p] \vdash (kind\ g\ x) \mapsto v1;$
$[g, m, p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g, m, p] \vdash (XorNode\ x\ y) \mapsto intval\text{-}xor\ v1\ v2 \mid$

— Comparison operators

IntegerEqualsNode:
$[\![[g, m, p] \vdash (kind\ g\ x) \mapsto IntVal32\ v1;$
$[g, m, p] \vdash (kind\ g\ y) \mapsto IntVal32\ v2;$
$val = bool\text{-}to\text{-}val(v1 = v2)]\!]$
$\Longrightarrow [g, m, p] \vdash (IntegerEqualsNode\ x\ y) \mapsto val \mid$

IntegerLessThanNode:
$[\![[g, m, p] \vdash (kind\ g\ x) \mapsto IntVal32\ v1;$
$[g, m, p] \vdash (kind\ g\ y) \mapsto IntVal32\ v2;$
$val = bool\text{-}to\text{-}val(v1 < v2)]\!]$
$\Longrightarrow [g, m, p] \vdash (IntegerLessThanNode\ x\ y) \mapsto val \mid$

IsNullNode:
$[\![[g, m, p] \vdash (kind\ g\ obj) \mapsto ObjRef\ ref;$
$val = bool\text{-}to\text{-}val(ref = None)]\!]$
$\Longrightarrow [g, m, p] \vdash (IsNullNode\ obj) \mapsto val \mid$

— Other nodes

ConditionalNode:
$[\![[g, m, p] \vdash (kind\ g\ condition) \mapsto IntVal32\ cond;$
$[g, m, p] \vdash (kind\ g\ trueExp) \mapsto IntVal32\ trueVal;$
$[g, m, p] \vdash (kind\ g\ falseExp) \mapsto IntVal32\ falseVal;$
$val = IntVal32\ (if\ (val\text{-}to\text{-}bool\ (IntVal32\ cond))\ then\ trueVal\ else\ falseVal)]\!]$
$\Longrightarrow [g, m, p] \vdash (ConditionalNode\ condition\ trueExp\ falseExp) \mapsto val \mid$

ShortCircuitOrNode:
$[\![[g, m, p] \vdash (kind\ g\ x) \mapsto IntVal32\ v1;$
$[g, m, p] \vdash (kind\ g\ y) \mapsto IntVal32\ v2;$
$val = IntVal32\ (if\ v1 \neq 0\ then\ v1\ else\ v2)]\!]$
$\Longrightarrow [g, m, p] \vdash (ShortCircuitOrNode\ x\ y) \mapsto val \mid$

LogicNegationNode:
$[\![[g, m, p] \vdash (kind\ g\ x) \mapsto IntVal32\ v1;$
$neg\text{-}v1 = (\neg(val\text{-}to\text{-}bool\ (IntVal32\ v1)));$
$val = bool\text{-}to\text{-}val\ neg\text{-}v1]\!]$

$\Longrightarrow [g,\ m,\ p] \vdash (LogicNegationNode\ x) \mapsto val\ |$

$InvokeNodeEval$:
$[g,\ m,\ p] \vdash (InvokeNode\ nid\ \text{- - - - -}) \mapsto m\ nid\ |$

$InvokeWithExceptionNodeEval$:
$[g,\ m,\ p] \vdash (InvokeWithExceptionNode\ nid\ \text{- - - - - -}) \mapsto m\ nid\ |$

$NewInstanceNode$:
$[g,\ m,\ p] \vdash (NewInstanceNode\ nid\ \text{- - -}) \mapsto m\ nid\ |$

$LoadFieldNode$:
$[g,\ m,\ p] \vdash (LoadFieldNode\ nid\ \text{- - -}) \mapsto m\ nid\ |$

$PiNode$:
$[\![[g,\ m,\ p] \vdash (kind\ g\ object) \mapsto val]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (PiNode\ object\ guard) \mapsto val\ |$

$RefNode$:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto val]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (RefNode\ x) \mapsto val$

code-pred $(modes: i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool\ as\ evalE)\ eval\ \langle proof \rangle$

The step semantics for phi nodes requires all the input nodes of the phi node to be evaluated to a value at the same time.

We introduce the *eval-all* relation to handle the evaluation of a list of node identifiers in parallel. As the evaluation semantics are side-effect free this is trivial.

inductive
$eval\text{-}all :: IRGraph \Rightarrow MapState \Rightarrow Params \Rightarrow ID\ list \Rightarrow Value\ list \Rightarrow bool$
$([\text{-},\ \text{-},\ \text{-}] \vdash \text{-} \longmapsto \text{-}\ 55)$
for $g\ m\ p$ **where**
$Base$:
$[g,\ m,\ p] \vdash [] \longmapsto []\ |$

$Transitive$:
$[\![[g,\ m,\ p] \vdash (kind\ g\ nid) \mapsto v;$
$[g,\ m,\ p] \vdash xs \longmapsto vs]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (nid\ \#\ xs) \longmapsto (v\ \#\ vs)$

code-pred $(modes: i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool\ as\ eval\text{-}allE)\ eval\text{-}all\ \langle proof \rangle$

inductive $eval\text{-}graph :: IRGraph \Rightarrow ID \Rightarrow Value\ list \Rightarrow Value \Rightarrow bool$
where
$[\![[g,\ new\text{-}map\text{-}state,\ ps] \vdash (kind\ g\ nid) \mapsto val]\!]$

$\Longrightarrow$ *eval-graph g nid ps val*

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

values {*v. eval-graph eg2-sq 4* [*IntVal32 5*] *v*}

fun *has-control-flow* :: *IRNode* ⇒ *bool* **where**
has-control-flow n = (*is-AbstractEndNode n*
∨ (*length* (*successors-of n*) > *0*))

definition *control-nodes* :: *IRNode set* **where**
control-nodes = {*n* . *has-control-flow n*}

fun *is-floating-node* :: *IRNode* ⇒ *bool* **where**
is-floating-node n = (¬(*has-control-flow n*))

definition *floating-nodes* :: *IRNode set* **where**
floating-nodes = {*n* . *is-floating-node n*}

lemma *is-floating-node n* ⟷ ¬(*has-control-flow n*)
⟨*proof*⟩

lemma *n* ∈ *control-nodes* ⟷ *n* ∉ *floating-nodes*
⟨*proof*⟩

Here we show that using the elimination rules for eval we can prove 'inverted rule' properties

lemma *evalAddNode* : [*g*, *m*, *p*] ⊢ (*AddNode x y*) ↦ *val* ⟹
(∃ *v1*. ([*g*, *m*, *p*] ⊢ (*kind g x*) ↦ *v1*) ∧
(∃ *v2*. ([*g*, *m*, *p*] ⊢ (*kind g y*) ↦ *v2*) ∧
val = *intval-add v1 v2*))
⟨*proof*⟩

lemma *not-floating*: (∃ *y ys*. (*successors-of n*) = *y* # *ys*) ⟶ ¬(*is-floating-node n*)
⟨*proof*⟩

We show that within the context of a graph and method state, the same node will always evaluate to the same value and the semantics is therefore deterministic.

theorem *evalDet*:
([*g*, *m*, *p*] ⊢ *node* ↦ *val1*) ⟹
(∀ *val2*. (([*g*, *m*, *p*] ⊢ *node* ↦ *val2*) ⟶ *val1* = *val2*))
⟨*proof*⟩

theorem *evalAllDet*:
([*g*, *m*, *p*] ⊢ *nodes* ⟼ *vals1*) ⟹

$(\forall\ vals2.\ (([g,\ m,\ p] \vdash nodes \longmapsto vals2) \longrightarrow vals1 = vals2))$
⟨*proof*⟩

end

6 Control-flow Semantics

theory *IRStepObj*
imports
IREval
begin

6.1 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 $h\text{-}load\text{-}field :: 'a \Rightarrow 'b \Rightarrow ('a,\ 'b)\ DynamicHeap \Rightarrow Value$ **where**
$h\text{-}load\text{-}field\ r\ f\ (h,\ n) = h\ r\ f$

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

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

type-synonym $RefFieldHeap = (objref,\ string)\ DynamicHeap$

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

6.2 Intraprocedural Semantics

Intraprocedural semantics are given as a small-step semantics.

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

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

$(\text{-}, \text{-} \vdash \text{-} \rightarrow \text{-}\ 55)$ **for** $g\ p$ **where**

$SequentialNode$:
$[\![is\text{-}sequential\text{-}node\ (kind\ g\ nid);$
$\quad nid' = (successors\text{-}of\ (kind\ g\ nid))!0]\!]$
$\quad \Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)\ |$

$IfNode$:
$[\![kind\ g\ nid = (IfNode\ cond\ tb\ fb);$
$\quad [g,\ m,\ p] \vdash (kind\ g\ cond) \mapsto val;$
$\quad nid' = (if\ val\text{-}to\text{-}bool\ val\ then\ tb\ else\ fb)]\!]$
$\quad \Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)\ |$

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

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

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

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

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

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

$SignedRemNode$:
$\quad [\![kind\ g\ nid = (SignedRemNode\ nid\ x\ y\ zero\ sb\ nxt);$

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

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

StoreFieldNode:
$[\![\textit{kind } g\ \textit{nid} = (\textit{StoreFieldNode nid f newval - } (\textit{Some obj})\ \textit{nid}');$
$[g, m, p] \vdash (\textit{kind } g\ \textit{newval}) \mapsto \textit{val};$
$[g, m, p] \vdash (\textit{kind } g\ \textit{obj}) \mapsto \textit{ObjRef ref};$
$h' = \textit{h-store-field ref f val } h;$
$m' = m(\textit{nid} := \textit{val})]\!]$
$\Longrightarrow g, p \vdash (\textit{nid}, m, h) \to (\textit{nid}', m', h')$ |

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

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

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*:
$(g, p \vdash (\textit{nid},m,h) \to \textit{next}) \Longrightarrow$
$(\forall\ \textit{next}'.\ ((g, p \vdash (\textit{nid},m,h) \to \textit{next}') \longrightarrow \textit{next} = \textit{next}'))$
⟨*proof*⟩

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

lemma *IfNodeStepCases*:
assumes $\textit{kind } g\ \textit{nid} = \textit{IfNode cond tb fb}$
assumes $[g, m, p] \vdash \textit{kind } g\ \textit{cond} \mapsto v$
assumes $g, p \vdash (\textit{nid}, m, h) \to (\textit{nid}', m, h)$
shows $\textit{nid}' \in \{\textit{tb}, \textit{fb}\}$
⟨*proof*⟩

lemma *IfNodeSeq*:

shows *kind g nid* = *IfNode cond tb fb* $\longrightarrow$ ¬(*is-sequential-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 $\exists\ v.\ ([g,\ m,\ p] \vdash kind\ g\ cond \mapsto v)$
⟨*proof*⟩

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

6.3 Interprocedural Semantics

type-synonym *Signature* = *string*
type-synonym *Program* = *Signature* ⇀ *IRGraph*

inductive *step-top* :: *Program* ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *RefFieldHeap* ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *RefFieldHeap* ⇒ *bool*
(- ⊢ - ⟶ - *55*)
for *P* **where**

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

InvokeNodeStep:
⟦*is-Invoke* (*kind g nid*);

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

ReturnNode:
⟦*kind g nid* = (*ReturnNode* (*Some expr*) -);
$[g,\ m,\ p] \vdash (kind\ g\ expr) \mapsto v$;

$cm' = cm(cnid := v)$;
$cnid'$ = (*successors-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 -);
  cm' = cm(cnid := (ObjRef (Some (2048))));

  cnid' = (successors-of (kind cg cnid))!0⟧
 ⟹ P ⊢ ((g,nid,m,p)#(cg,cnid,cm,cp)#stk, h) ⟶ ((cg,cnid',cm',cp)#stk, h) |

UnwindNode:
⟦kind g nid = (UnwindNode exception);

  [g, m, p] ⊢ (kind g exception) ↦ e;

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

  cm' = cm(cnid := e)⟧
⟹ P ⊢ ((g,nid,m,p)#(cg,cnid,cm,cp)#stk, h) ⟶ ((cg,exEdge,cm',cp)#stk, h)
```

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

6.4 Big-step Execution

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

fun *has-return* :: *MapState* ⇒ *bool* **where**
  has-return m = (*m 0* ≠ *UndefVal*)

```
inductive exec :: Program
    ⇒ (IRGraph × ID × MapState × Params) list × RefFieldHeap
    ⇒ Trace
    ⇒ (IRGraph × ID × MapState × Params) list × RefFieldHeap
    ⇒ Trace
    ⇒ bool
  (- ⊢ - | - ⟶* - | -)
  for P
  where
  ⟦P ⊢ (((g,nid,m,p)#xs),h) ⟶ (((g',nid',m',p')#ys),h');
    ¬(has-return m');

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

    exec P (((g',nid',m',p')#ys),h') l' next-state l''⟧
    ⟹ exec P (((g,nid,m,p)#xs),h) l next-state l''

  |
  ⟦P ⊢ (((g,nid,m,p)#xs),h) ⟶ (((g',nid',m',p')#ys),h');
    has-return m';

    l' = (l @ [(g,nid,m,p)])⟧
    ⟹ 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* × *RefFieldHeap*
  ⇒ *nat*
  ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *RefFieldHeap*
  ⇒ *bool*
 (-⊢-→∗-∗ -)
 where
 ⟦*n* > *0*;
  p ⊢ *s* ⟶ *s*′;
  exec-debug p s′ (*n* − *1*) *s*′′⟧
  ⟹ *exec-debug p s n s*′′ |

 ⟦*n* = *0*⟧
  ⟹ *exec-debug p s n s*
code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*) *exec-debug* ⟨*proof*⟩

6.4.1 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 None field-sq* (*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* (*Some 0*) *field-sq* (*prod.snd res*)
| *res*. (λx. *Some eg4-sq*) $\vdash$ ([(*eg4-sq*, *0*, *new-map-state*, *p3*), (*eg4-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) $\rightarrow$*∗3∗ *res*}
end

7 Proof Infrastructure

7.1 Bisimulation

theory *Bisimulation*
imports
Stuttering
begin

inductive *weak-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
(- . - $\sim$ -) **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* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
(- | - $\sim$ -) **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* | $g \sim g'$
⟨*proof*⟩

end

7.2 Formedness Properties

theory *Form*
imports
Semantics.IREval
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* .

$(\forall\ v\ m\ p\ .\ ([g,\ m,\ p] \vdash (kind\ g\ n) \mapsto v) \longrightarrow valid\text{-}value\ (stamp\ g\ n)\ v))$

fun *wf-stamp* :: $IRGraph \Rightarrow (ID \Rightarrow Stamp) \Rightarrow bool$ **where**
$wf\text{-}stamp\ g\ s = (\forall\ n \in ids\ g\ .$
$(\forall\ v\ m\ p\ .\ ([g,\ m,\ p] \vdash (kind\ g\ n) \mapsto v) \longrightarrow valid\text{-}value\ (s\ n)\ v))$

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

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

fun *wf-logic-node-inputs* :: $IRGraph \Rightarrow ID \Rightarrow bool$ **where**
$wf\text{-}logic\text{-}node\text{-}inputs\ g\ n =$
$(\forall\ inp \in set\ (inputs\text{-}of\ (kind\ g\ n))\ .\ (\forall\ v\ m\ p\ .\ ([g,\ m,\ p] \vdash kind\ g\ inp \mapsto v) \longrightarrow wf\text{-}bool\ v))$

end

7.3 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.IREval
begin

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

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

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* ⇒ *ID* ⇒ *ID* ⇒ *bool*
for *g* **where**

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

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

use-trans: ⟦*eval-uses g nid nid′*;
eval-uses g nid′ nid″⟧
⟹ *eval-uses g nid nid″*

fun *eval-usages* :: *IRGraph* ⇒ *ID* ⇒ *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 $nid \in eval\text{-}usages\ g\ nid$
⟨*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-of n*) *child*
shows $child \in inputs\ g\ nid$
⟨*proof*⟩

lemma *child-member-in*:
assumes $nid \in ids\ g$
assumes *List.member* (*inputs-of* (*kind g nid*)) *child*
shows $child \in inputs\ g\ nid$
⟨*proof*⟩

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

lemma *inp-in-g-wf*:
  assumes *wf-graph g*
  assumes $n \in$ *inputs g nid*
  shows $n \in$ *ids g*
  ⟨*proof*⟩

lemma *kind-unchanged*:
  assumes *nid* $\in$ *ids g1*
  assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
  shows *kind g1 nid* = *kind g2 nid*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

lemma *eval-uses-imp*:
$((nid' \in ids\ g \land nid = nid')$
$\lor\ nid' \in inputs\ g\ nid$
$\lor\ (\exists\, nid''\ .\ eval\text{-}uses\ g\ nid\ nid'' \land eval\text{-}uses\ g\ nid''\ nid'))$
$\longleftrightarrow eval\text{-}uses\ g\ nid\ nid'$
$\langle proof \rangle$

lemma *wf-use-ids*:
assumes $wf\text{-}graph\ g$
assumes $nid \in ids\ g$

assumes *eval-uses g nid nid'*
shows *nid'* ∈ *ids g*
⟨*proof*⟩

lemma *no-external-use*:
assumes *wf-graph g*
assumes *nid'* ∉ *ids g*
assumes *nid* ∈ *ids g*
shows ¬(*eval-uses g nid nid'*)
⟨*proof*⟩

end

7.4 Graph Rewriting

theory
Rewrites
imports
IRGraphFrames
Stuttering
begin

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

lemma *replace-usages-effect*:
assumes *g'* = *replace-usages nid nid' g*
shows *kind g' nid* = *RefNode nid'*
⟨*proof*⟩

lemma *replace-usages-changeonly*:
assumes *nid* ∈ *ids g*
assumes *g'* = *replace-usages nid nid' g*
shows *changeonly* {*nid*} *g g'*
⟨*proof*⟩

lemma *replace-usages-unchanged*:
assumes *nid* ∈ *ids g*
assumes *g'* = *replace-usages nid nid' g*
shows *unchanged* (*ids g* − {*nid*}) *g g'*
⟨*proof*⟩

fun *nextNid* :: *IRGraph* ⇒ *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*:
$ids\ g \neq \{\} \longrightarrow nextNid\ g \notin ids\ g$
⟨*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 $g', p \vdash (ifcond, m, h) \rightarrow (t, m, h)$
⟨*proof*⟩

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

lemma *diff-forall*:
assumes $\forall n \in ids\ g - \{nid\}.\ cond\ n$
shows $\forall n.\ n \in ids\ g \land n \notin \{nid\} \longrightarrow cond\ n$
⟨*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* ∈ *ids g*
  assumes *g′* = *constantCondition b nid* (*kind g nid*) *g*
  shows *changeonly* {*nid*} *g g′*
⟨*proof*⟩

lemma *constantConditionNoIf*:
  assumes ∀ *cond t f*. *kind g ifcond* ≠ *IfNode cond t f*
  assumes *g′* = *constantCondition val ifcond* (*kind g ifcond*) *g*
  shows ∃ *nid′* .(*g m p h* ⊢ *ifcond* ⇝ *nid′*) ⟷ (*g′ m p h* ⊢ *ifcond* ⇝ *nid′*)
⟨*proof*⟩

lemma *constantConditionValid*:
  assumes *kind g ifcond* = *IfNode cond t f*
  assumes [*g*, *m*, *p*] ⊢ *kind g cond* ↦ *v*
  assumes *const* = *val-to-bool v*
  assumes *g′* = *constantCondition const ifcond* (*kind g ifcond*) *g*
  shows ∃ *nid′* .(*g m p h* ⊢ *ifcond* ⇝ *nid′*) ⟷ (*g′ m p h* ⊢ *ifcond* ⇝ *nid′*)
⟨*proof*⟩

end

7.5 Stuttering

theory *Stuttering*
  imports
    Semantics.IRStepObj
begin

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

  StutterStep:
  ⟦*g*, *p* ⊢ (*nid*,*m*,*h*) → (*nid′*,*m*,*h*)⟧
   ⟹ *g m p h* ⊢ *nid* ⇝ *nid′* |

  Transitive:
  ⟦*g*, *p* ⊢ (*nid*,*m*,*h*) → (*nid″*,*m*,*h*);

$g\ m\ p\ h \vdash nid'' \rightsquigarrow nid'\rrbracket$
$\Longrightarrow g\ m\ p\ h \vdash nid \rightsquigarrow nid'$

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

end

8 Canonicalization Phase

theory *Canonicalization*
imports
Proofs.IRGraphFrames
Proofs.Stuttering
Proofs.Bisimulation
Proofs.Form

Graph.Traversal
begin

inductive *CanonicalizeConditional* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
where
negate-condition:
⟦*kind g cond* = *LogicNegationNode flip*⟧
⟹ *CanonicalizeConditional g* (*ConditionalNode cond tb fb*) (*ConditionalNode flip fb tb*) |

const-true:
⟦*kind g cond* = *ConstantNode val*;
val-to-bool val⟧
⟹ *CanonicalizeConditional g* (*ConditionalNode cond tb fb*) (*RefNode tb*) |

const-false:
⟦*kind g cond* = *ConstantNode val*;
¬(*val-to-bool val*)⟧
⟹ *CanonicalizeConditional g* (*ConditionalNode cond tb fb*) (*RefNode fb*) |

eq-branches:
⟦*tb* = *fb*⟧
⟹ *CanonicalizeConditional g* (*ConditionalNode cond tb fb*) (*RefNode tb*) |

cond-eq:
⟦*kind g cond* = *IntegerEqualsNode tb fb*⟧
⟹ *CanonicalizeConditional g* (*ConditionalNode cond tb fb*) (*RefNode fb*) |

condition-bounds-x:
⟦*kind g cond* = *IntegerLessThanNode tb fb*;

$stpi\text{-}upper\ (stamp\ g\ tb) \leq stpi\text{-}lower\ (stamp\ g\ fb)]\!]$
$\Longrightarrow CanonicalizeConditional\ g\ (ConditionalNode\ cond\ tb\ fb)\ (RefNode\ tb)\ |$

condition-bounds-y:
$[\![kind\ g\ cond = IntegerLessThanNode\ fb\ tb;$
$stpi\text{-}upper\ (stamp\ g\ fb) \leq stpi\text{-}lower\ (stamp\ g\ tb)]\!]$
$\Longrightarrow CanonicalizeConditional\ g\ (ConditionalNode\ cond\ tb\ fb)\ (RefNode\ tb)$

inductive *CanonicalizeAdd* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
add-both-const:
$[\![kind\ g\ x = ConstantNode\ c\text{-}1;$
$kind\ g\ y = ConstantNode\ c\text{-}2;$
$val = intval\text{-}add\ c\text{-}1\ c\text{-}2]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ y)\ (ConstantNode\ val)\ |$

add-xzero:
$[\![kind\ g\ x = ConstantNode\ c\text{-}1;$
$\neg(is\text{-}ConstantNode\ (kind\ g\ y));$
$c\text{-}1 = (IntVal32\ 0)]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ y)\ (RefNode\ y)\ |$

add-yzero:
$[\![\neg(is\text{-}ConstantNode\ (kind\ g\ x));$
$kind\ g\ y = ConstantNode\ c\text{-}2;$
$c\text{-}2 = (IntVal32\ 0)]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ y)\ (RefNode\ x)\ |$

add-xsub:

$[\![kind\ g\ x = SubNode\ a\ y\ ]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ y)\ (RefNode\ a)\ |$

add-ysub:

$[\![kind\ g\ y = SubNode\ a\ x\ ]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ y)\ (RefNode\ a)\ |$

add-xnegate:

$[\![kind\ g\ nx = NegateNode\ x\ ]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ nx\ y)\ (SubNode\ y\ x)\ |$

add-ynegate:

$[\![kind\ g\ ny = NegateNode\ y\ ]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ ny)\ (SubNode\ x\ y)$

inductive *CanonicalizeIf* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
trueConst:
$[\![kind\ g\ cond = ConstantNode\ condv;$
$val\text{-}to\text{-}bool\ condv]\!]$
$\Longrightarrow CanonicalizeIf\ g\ (IfNode\ cond\ tb\ fb)\ (RefNode\ tb)\ |$

falseConst:
$[\![kind\ g\ cond = ConstantNode\ condv;$
$\neg(val\text{-}to\text{-}bool\ condv)]\!]$
$\Longrightarrow CanonicalizeIf\ g\ (IfNode\ cond\ tb\ fb)\ (RefNode\ fb)\ |$

eqBranch:
$[\![\neg(is\text{-}ConstantNode\ (kind\ g\ cond));$
$tb = fb]\!]$
$\Longrightarrow CanonicalizeIf\ g\ (IfNode\ cond\ tb\ fb)\ (RefNode\ tb)\ |$

eqCondition:
$[\![kind\ g\ cond = IntegerEqualsNode\ x\ x]\!]$
$\Longrightarrow CanonicalizeIf\ g\ (IfNode\ cond\ tb\ fb)\ (RefNode\ tb)$

inductive *CanonicalizeBinaryArithmeticNode* :: *ID* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
add-const-fold:
$[\![op = kind\ g\ op\text{-}id;$
$is\text{-}AddNode\ op;$
$kind\ g\ (ir\text{-}x\ op) = ConditionalNode\ cond\ tb\ fb;$
$kind\ g\ tb = ConstantNode\ c\text{-}1;$
$kind\ g\ fb = ConstantNode\ c\text{-}2;$
$kind\ g\ (ir\text{-}y\ op) = ConstantNode\ c\text{-}3;$
$tv = intval\text{-}add\ c\text{-}1\ c\text{-}3;$
$fv = intval\text{-}add\ c\text{-}2\ c\text{-}3;$
$g' = replace\text{-}node\ tb\ ((ConstantNode\ tv),\ constantAsStamp\ tv)\ g;$
$g'' = replace\text{-}node\ fb\ ((ConstantNode\ fv),\ constantAsStamp\ fv)\ g';$
$g''' = replace\text{-}node\ op\text{-}id\ (kind\ g\ (ir\text{-}x\ op),\ meet\ (constantAsStamp\ tv)\ (constantAsStamp\ fv))\ g''\ ]\!]$

$\Longrightarrow$ *CanonicalizeBinaryArithmeticNode op-id g g'''*

inductive *CanonicalizeCommutativeBinaryArithmeticNode* :: *IRGraph* $\Rightarrow$ *IRNode* $\Rightarrow$ *IRNode* $\Rightarrow$ *bool*
for *g* **where**

add-ids-ordered:
$[\![\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y));$
$((\textit{is-ConstantNode}\ (\textit{kind}\ g\ x)) \vee (x > y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AddNode x y*) (*AddNode y x*) |

and-ids-ordered:
$[\![\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y));$
$((\textit{is-ConstantNode}\ (\textit{kind}\ g\ x)) \vee (x > y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AndNode x y*) (*AndNode y x*) |

int-equals-ids-ordered:
$[\![\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y));$
$((\textit{is-ConstantNode}\ (\textit{kind}\ g\ x)) \vee (x > y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*IntegerEqualsNode x y*) (*IntegerEqualsNode y x*) |

mul-ids-ordered:
$[\![\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y));$
$((\textit{is-ConstantNode}\ (\textit{kind}\ g\ x)) \vee (x > y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*MulNode x y*) (*MulNode y x*) |

or-ids-ordered:
$[\![\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y));$
$((\textit{is-ConstantNode}\ (\textit{kind}\ g\ x)) \vee (x > y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*OrNode x y*) (*OrNode y x*) |

xor-ids-ordered:
$[\![\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y));$
$((\textit{is-ConstantNode}\ (\textit{kind}\ g\ x)) \vee (x > y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*XorNode x y*) (*XorNode y x*) |

add-swap-const-first:
$[\![\textit{is-ConstantNode}\ (\textit{kind}\ g\ x);$
$\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y))]\!]$

$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AddNode x y*) (*AddNode y x*) |

and-swap-const-first:
⟦*is-ConstantNode* (*kind g x*);
¬(*is-ConstantNode* (*kind g y*))⟧
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AndNode x y*) (*AndNode y x*) |

int-equals-swap-const-first:
⟦*is-ConstantNode* (*kind g x*);
¬(*is-ConstantNode* (*kind g y*))⟧
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*IntegerEqualsNode x y*) (*IntegerEqualsNode y x*) |

mul-swap-const-first:
⟦*is-ConstantNode* (*kind g x*);
¬(*is-ConstantNode* (*kind g y*))⟧
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*MulNode x y*) (*MulNode y x*) |

or-swap-const-first:
⟦*is-ConstantNode* (*kind g x*);
¬(*is-ConstantNode* (*kind g y*))⟧
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*OrNode x y*) (*OrNode y x*) |

xor-swap-const-first:
⟦*is-ConstantNode* (*kind g x*);
¬(*is-ConstantNode* (*kind g y*))⟧
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*XorNode x y*) (*XorNode y x*)

inductive *CanonicalizeSub* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
sub-same:
⟦*x* = *y*;
stamp g x = (*IntegerStamp b l h*)⟧
$\Longrightarrow$ *CanonicalizeSub g* (*SubNode x y*) (*ConstantNode* (*IntVal32 0*)) |

sub-both-const:
⟦*kind g x* = *ConstantNode c-1*;
kind g y = *ConstantNode c-2*;
val = *intval-sub c-1 c-2*⟧
$\Longrightarrow$ *CanonicalizeSub g* (*SubNode x y*) (*ConstantNode val*) |

sub-left-add1:

$[\![kind\ g\ left = AddNode\ a\ b]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ left\ b)\ (RefNode\ a)\ |$

sub-left-add2:

$[\![kind\ g\ left = AddNode\ a\ b]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ left\ a)\ (RefNode\ b)\ |$

sub-left-sub:

$[\![kind\ g\ left = SubNode\ a\ b]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ left\ a)\ (NegateNode\ b)\ |$

sub-right-add1:

$[\![kind\ g\ right = AddNode\ a\ b]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ a\ right)\ (NegateNode\ b)\ |$

sub-right-add2:

$[\![kind\ g\ right = AddNode\ a\ b]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ b\ right)\ (NegateNode\ a)\ |$

sub-right-sub:

$[\![kind\ g\ right = AddNode\ a\ b]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ a\ right)\ (RefNode\ a)\ |$

sub-yzero:

$[\![kind\ g\ y = ConstantNode\ (IntVal32\ 0)]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ x\ y)\ (RefNode\ x)\ |$

sub-xzero:

$[\![kind\ g\ x = ConstantNode\ (IntVal32\ 0)]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ x\ y)\ (NegateNode\ y)\ |$

sub-y-negate:

$[\![kind\ g\ nb = NegateNode\ b]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ a\ nb)\ (AddNode\ a\ b)$

inductive *CanonicalizeMul* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
mul-both-const:
$[\![kind\ g\ x = ConstantNode\ c\text{-}1;$

$kind\ g\ y = ConstantNode\ c\text{-}2;$
$val = intval\text{-}mul\ c\text{-}1\ c\text{-}2]\!]$
$\Longrightarrow CanonicalizeMul\ g\ (MulNode\ x\ y)\ (ConstantNode\ val)\ |$

mul-xzero:
$[\![kind\ g\ x = ConstantNode\ c\text{-}1;$
$\neg(is\text{-}ConstantNode\ (kind\ g\ y));$
$c\text{-}1 = (IntVal32\ 0)]\!]$
$\Longrightarrow CanonicalizeMul\ g\ (MulNode\ x\ y)\ (ConstantNode\ c\text{-}1)\ |$

mul-yzero:
$[\![kind\ g\ y = ConstantNode\ c\text{-}1;$
$\neg(is\text{-}ConstantNode\ (kind\ g\ x));$
$c\text{-}1 = (IntVal32\ 0)]\!]$
$\Longrightarrow CanonicalizeMul\ g\ (MulNode\ x\ y)\ (ConstantNode\ c\text{-}1)\ |$

mul-xone:
$[\![kind\ g\ x = ConstantNode\ c\text{-}1;$
$\neg(is\text{-}ConstantNode\ (kind\ g\ y));$
$c\text{-}1 = (IntVal32\ 1)]\!]$
$\Longrightarrow CanonicalizeMul\ g\ (MulNode\ x\ y)\ (RefNode\ y)\ |$

mul-yone:
$[\![kind\ g\ y = ConstantNode\ c\text{-}1;$
$\neg(is\text{-}ConstantNode\ (kind\ g\ x));$
$c\text{-}1 = (IntVal32\ 1)]\!]$
$\Longrightarrow CanonicalizeMul\ g\ (MulNode\ x\ y)\ (RefNode\ x)\ |$

mul-xnegate:
$[\![kind\ g\ x = ConstantNode\ c\text{-}1;$
$\neg(is\text{-}ConstantNode\ (kind\ g\ y));$
$c\text{-}1 = (IntVal32\ (-1))]\!]$
$\Longrightarrow CanonicalizeMul\ g\ (MulNode\ x\ y)\ (NegateNode\ y)\ |$

mul-ynegate:
$[\![kind\ g\ y = ConstantNode\ c\text{-}1;$
$\neg(is\text{-}ConstantNode\ (kind\ g\ x));$
$c\text{-}1 = (IntVal32\ (-1))]\!]$
$\Longrightarrow CanonicalizeMul\ g\ (MulNode\ x\ y)\ (NegateNode\ x)$

inductive $CanonicalizeAbs :: IRGraph \Rightarrow IRNode \Rightarrow IRNode \Rightarrow bool$
for g **where**
abs-abs:
$[\![kind\ g\ x = (AbsNode\ y)]\!]$
$\Longrightarrow CanonicalizeAbs\ g\ (AbsNode\ x)\ (AbsNode\ y)\ |$

abs-negate:
$[\![kind\ g\ nx = (NegateNode\ x)]\!]$
$\Longrightarrow CanonicalizeAbs\ g\ (AbsNode\ nx)\ (AbsNode\ x)$

inductive *CanonicalizeNegate* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
negate-const:
$[\![kind\ g\ nx = (ConstantNode\ val);$
$val = (IntVal32\ v);$
$neg\text{-}val = intval\text{-}sub\ (IntVal32\ 0)\ val\ ]\!]$
$\Longrightarrow CanonicalizeNegate\ g\ (NegateNode\ nx)\ (ConstantNode\ neg\text{-}val)$ |

negate-negate:
$[\![kind\ g\ nx = (NegateNode\ x)]\!]$
$\Longrightarrow CanonicalizeNegate\ g\ (NegateNode\ nx)\ (RefNode\ x)$ |

negate-sub:
$[\![kind\ g\ sub = (SubNode\ x\ y);$
$stamp\ g\ sub = (IntegerStamp\ \text{-}\ \text{-}\ \text{-})]\!]$
$\Longrightarrow CanonicalizeNegate\ g\ (NegateNode\ sub)\ (SubNode\ y\ x)$

inductive *CanonicalizeNot* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
not-const:
$[\![kind\ g\ nx = (ConstantNode\ val);$
$neg\text{-}val = intval\text{-}not\ val]\!]$
$\Longrightarrow CanonicalizeNot\ g\ (NotNode\ nx)\ (ConstantNode\ neg\text{-}val)$ |

not-not:
$[\![kind\ g\ nx = (NotNode\ x)]\!]$
$\Longrightarrow CanonicalizeNot\ g\ (NotNode\ nx)\ (RefNode\ x)$

inductive *CanonicalizeLogicNegation* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
logical-not-const:
$[\![kind\ g\ nx = (ConstantNode\ val);$
$neg\text{-}val = bool\text{-}to\text{-}val\ (\neg(val\text{-}to\text{-}bool\ val))]\!]$
$\Longrightarrow CanonicalizeLogicNegation\ g\ (LogicNegationNode\ nx)\ (ConstantNode\ neg\text{-}val)$
|

logical-not-not:
$[\![kind\ g\ nx = (LogicNegationNode\ x)]\!]$
$\Longrightarrow CanonicalizeLogicNegation\ g\ (LogicNegationNode\ nx)\ (RefNode\ x)$

inductive *CanonicalizeAnd* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*

for g **where**
and-same:
$[\![x = y]\!]$
$\Longrightarrow$ *CanonicalizeAnd* g (*AndNode* x y) (*RefNode* x) |

and-xtrue:
$[\![$*kind* g x = *ConstantNode val*;
val-to-bool val$]\!]$
$\Longrightarrow$ *CanonicalizeAnd* g (*AndNode* x y) (*RefNode* y) |

and-ytrue:
$[\![$*kind* g y = *ConstantNode val*;
val-to-bool val$]\!]$
$\Longrightarrow$ *CanonicalizeAnd* g (*AndNode* x y) (*RefNode* x) |

and-xfalse:
$[\![$*kind* g x = *ConstantNode val*;
$\neg$(*val-to-bool val*)$]\!]$
$\Longrightarrow$ *CanonicalizeAnd* g (*AndNode* x y) (*ConstantNode val*) |

and-yfalse:
$[\![$*kind* g y = *ConstantNode val*;
$\neg$(*val-to-bool val*)$]\!]$
$\Longrightarrow$ *CanonicalizeAnd* g (*AndNode* x y) (*ConstantNode val*)

inductive *CanonicalizeOr* :: *IRGraph* $\Rightarrow$ *IRNode* $\Rightarrow$ *IRNode* $\Rightarrow$ *bool*
for g **where**
or-same:
$[\![x = y]\!]$
$\Longrightarrow$ *CanonicalizeOr* g (*OrNode* x y) (*RefNode* x) |

or-xtrue:
$[\![$*kind* g x = *ConstantNode val*;
val-to-bool val$]\!]$
$\Longrightarrow$ *CanonicalizeOr* g (*OrNode* x y) (*ConstantNode val*) |

or-ytrue:
$[\![$*kind* g y = *ConstantNode val*;
val-to-bool val$]\!]$
$\Longrightarrow$ *CanonicalizeOr* g (*OrNode* x y) (*ConstantNode val*) |

or-xfalse:
$[\![$*kind* g x = *ConstantNode val*;
$\neg$(*val-to-bool val*)$]\!]$
$\Longrightarrow$ *CanonicalizeOr* g (*OrNode* x y) (*RefNode* y) |

or-yfalse:
$[\![kind\ g\ y = ConstantNode\ val;$
$\neg(val\text{-}to\text{-}bool\ val)]\!]$
$\Longrightarrow CanonicalizeOr\ g\ (OrNode\ x\ y)\ (RefNode\ x)$

inductive *CanonicalizeDeMorgansLaw* :: *ID* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool*
where

de-morgan-or-to-and:
$[\![kind\ g\ nid = OrNode\ nx\ ny;$
$kind\ g\ nx = NotNode\ x;$
$kind\ g\ ny = NotNode\ y;$
$new\text{-}add\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ new\text{-}add\text{-}id\ ((AddNode\ x\ y),\ (IntegerStamp\ 1\ 0\ 1))\ g;$
$g'' = replace\text{-}node\ nid\ ((NotNode\ new\text{-}add\text{-}id),\ (IntegerStamp\ 1\ 0\ 1))\ g']\!]$
$\Longrightarrow CanonicalizeDeMorgansLaw\ nid\ g\ g''\ |$

de-morgan-and-to-or:
$[\![kind\ g\ nid = AndNode\ nx\ ny;$
$kind\ g\ nx = NotNode\ x;$
$kind\ g\ ny = NotNode\ y;$
$new\text{-}add\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ new\text{-}add\text{-}id\ ((OrNode\ x\ y),\ (IntegerStamp\ 1\ 0\ 1))\ g;$
$g'' = replace\text{-}node\ nid\ ((NotNode\ new\text{-}add\text{-}id),\ (IntegerStamp\ 1\ 0\ 1))\ g']\!]$
$\Longrightarrow CanonicalizeDeMorgansLaw\ nid\ g\ g''$

inductive *CanonicalizeIntegerEquals* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
int-equals-same-node:
$[\![x = y]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ g\ (IntegerEqualsNode\ x\ y)\ (ConstantNode\ (IntVal32\ 1))\ |$

int-equals-distinct:
$[\![alwaysDistinct\ (stamp\ g\ x)\ (stamp\ g\ y)]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ g\ (IntegerEqualsNode\ x\ y)\ (ConstantNode\ (IntVal32\ 0))\ |$

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

$[\![kind\ g\ left = AddNode\ x\ y;$
$kind\ g\ right = AddNode\ x\ z]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ g\ (IntegerEqualsNode\ left\ right)\ (IntegerEqualsNode$

$y\ z$) |

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

$[\![kind\ g\ left = AddNode\ x\ y;$
$kind\ g\ right = AddNode\ z\ x]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ g\ (IntegerEqualsNode\ left\ right)\ (IntegerEqualsNode\ y\ z)$ |

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

$[\![kind\ g\ left = AddNode\ y\ x;$
$kind\ g\ right = AddNode\ x\ z]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ g\ (IntegerEqualsNode\ left\ right)\ (IntegerEqualsNode\ y\ z)$ |

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

$[\![kind\ g\ left = AddNode\ y\ x;$
$kind\ g\ right = AddNode\ z\ x]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ g\ (IntegerEqualsNode\ left\ right)\ (IntegerEqualsNode\ y\ z)$ |

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

$[\![kind\ g\ left = SubNode\ x\ y;$
$kind\ g\ right = SubNode\ x\ z]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ g\ (IntegerEqualsNode\ left\ right)\ (IntegerEqualsNode\ y\ z)$ |

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

$[\![kind\ g\ left = SubNode\ y\ x;$
$kind\ g\ right = SubNode\ z\ x]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ g\ (IntegerEqualsNode\ left\ right)\ (IntegerEqualsNode\ y\ z)$

inductive *CanonicalizeIntegerEqualsGraph* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
where
int-equals-rewrite:
$[\![CanonicalizeIntegerEquals\ g\ node\ node';$
$node = kind\ g\ nid;$
$g' = replace\text{-}node\ nid\ (node', stamp\ g\ nid)\ g\ ]\!]$
$\Longrightarrow CanonicalizeIntegerEqualsGraph\ nid\ g\ g'$ |

int-equals-left-contains-right1:
$[\![$*kind g nid* $=$ *IntegerEqualsNode left x*;
kind g left $=$ *AddNode x y*;
const-id $=$ *nextNid g*;
g' $=$ *add-node const-id* ((*ConstantNode* (*IntVal32 0*)), *constantAsStamp* (*IntVal32 0*)) *g*;
g'' $=$ *replace-node const-id* ((*IntegerEqualsNode y const-id*), *stamp g nid*) $g'$$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEqualsGraph nid g* g'' |

int-equals-left-contains-right2:
$[\![$*kind g nid* $=$ *IntegerEqualsNode left y*;
kind g left $=$ *AddNode x y*;
const-id $=$ *nextNid g*;
g' $=$ *add-node const-id* ((*ConstantNode* (*IntVal32 0*)), *constantAsStamp* (*IntVal32 0*)) *g*;
g'' $=$ *replace-node const-id* ((*IntegerEqualsNode x const-id*), *stamp g nid*) $g'$$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEqualsGraph nid g* g'' |

int-equals-right-contains-left1:
$[\![$*kind g nid* $=$ *IntegerEqualsNode x right*;
kind g right $=$ *AddNode x y*;
const-id $=$ *nextNid g*;
g' $=$ *add-node const-id* ((*ConstantNode* (*IntVal32 0*)), *constantAsStamp* (*IntVal32 0*)) *g*;
g'' $=$ *replace-node const-id* ((*IntegerEqualsNode y const-id*), *stamp g nid*) $g'$$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEqualsGraph nid g* g'' |

int-equals-right-contains-left2:
$[\![$*kind g nid* $=$ *IntegerEqualsNode y right*;
kind g right $=$ *AddNode x y*;
const-id $=$ *nextNid g*;
g' $=$ *add-node const-id* ((*ConstantNode* (*IntVal32 0*)), *constantAsStamp* (*IntVal32 0*)) *g*;
g'' $=$ *replace-node const-id* ((*IntegerEqualsNode x const-id*), *stamp g nid*) $g'$$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEqualsGraph nid g* g'' |

int-equals-left-contains-right3:
$[\![$*kind g nid* $=$ *IntegerEqualsNode left x*;
kind g left $=$ *SubNode x y*;
const-id $=$ *nextNid g*;
g' $=$ *add-node const-id* ((*ConstantNode* (*IntVal32 0*)), *constantAsStamp* (*IntVal32*

$0)) \ g;$
$g'' = replace\text{-}node\ const\text{-}id\ ((IntegerEqualsNode\ y\ const\text{-}id),\ stamp\ g\ nid)\ g']\!]$
$\Longrightarrow CanonicalizeIntegerEqualsGraph\ nid\ g\ g''\ |$

int-equals-right-contains-left3:
$[\![kind\ g\ nid = IntegerEqualsNode\ x\ right;$
$kind\ g\ right = SubNode\ x\ y;$
$const\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ const\text{-}id\ ((ConstantNode\ (IntVal32\ 0)),\ constantAsStamp\ (IntVal32$
$0))\ g;$
$g'' = replace\text{-}node\ const\text{-}id\ ((IntegerEqualsNode\ y\ const\text{-}id),\ stamp\ g\ nid)\ g']\!]$
$\Longrightarrow CanonicalizeIntegerEqualsGraph\ nid\ g\ g''$

inductive *CanonicalizationStep* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
ConditionalNode:
$[\![CanonicalizeConditional\ g\ node\ node']\!]$
$\Longrightarrow CanonicalizationStep\ g\ node\ node'\ |$

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

IfNode:
$[\![CanonicalizeIf\ g\ node\ node']\!]$
$\Longrightarrow CanonicalizationStep\ g\ node\ node'\ |$

SubNode:
$[\![$*CanonicalizeSub g node node'*$]\!]$
$\Longrightarrow$ *CanonicalizationStep g node node'* |

MulNode:
$[\![$*CanonicalizeMul g node node'*$]\!]$
$\Longrightarrow$ *CanonicalizationStep g node node'* |

AndNode:
$[\![$*CanonicalizeAnd g node node'*$]\!]$
$\Longrightarrow$ *CanonicalizationStep g node node'* |

OrNode:
$[\![$*CanonicalizeOr g node node'*$]\!]$
$\Longrightarrow$ *CanonicalizationStep g node node'* |

AbsNode:
$[\![$*CanonicalizeAbs g node node'*$]\!]$
$\Longrightarrow$ *CanonicalizationStep g node node'* |

NotNode:
$[\![$*CanonicalizeNot g node node'*$]\!]$
$\Longrightarrow$ *CanonicalizationStep g node node'* |

NegateNode:
$[\![$*CanonicalizeNegate g node node'*$]\!]$
$\Longrightarrow$ *CanonicalizationStep g node node'* |

LogicNegationNode:
$[\![$*CanonicalizeLogicNegation g node node'*$]\!]$
$\Longrightarrow$ *CanonicalizationStep g node node'*

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizeConditional* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizeAdd* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizeIf* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizeSub* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizeMul* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizeAnd* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizeOr* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizeAbs* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizeNot* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizeNegate* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizeLogicNegation* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *CanonicalizationStep* ⟨*proof*⟩

type-synonym *CanonicalizationAnalysis* = *bool option*

fun *analyse* :: (*ID* × *Seen* × *CanonicalizationAnalysis*) ⇒ *CanonicalizationAnalysis*
where
  analyse i = *None*

inductive *CanonicalizationPhase*
  :: *IRGraph* ⇒ (*ID* × *Seen* × *CanonicalizationAnalysis*) ⇒ *IRGraph* ⇒ *bool* **where**

  — Can do a step and optimise for the current node
  ⟦*Step analyse g* (*nid*, *seen*, *i*) (*Some* (*nid*′, *seen*′, *i*′));
    CanonicalizationStep g (*kind g nid*) *node*;

    g′ = *replace-node nid* (*node*, *stamp g nid*) *g*;

    CanonicalizationPhase g′ (*nid*′, *seen*′, *i*′) *g*′′⟧
    ⟹ *CanonicalizationPhase g* (*nid*, *seen*, *i*) *g*′′ |

  — Can do a step, matches whether optimised or not causing non-determinism We need to find a way to negate ConditionalEliminationStep
  ⟦*Step analyse g* (*nid*, *seen*, *i*) (*Some* (*nid*′, *seen*′, *i*′));

    CanonicalizationPhase g (*nid*′, *seen*′, *i*′) *g*′⟧
    ⟹ *CanonicalizationPhase g* (*nid*, *seen*, *i*) *g*′ |

  ⟦*Step analyse g* (*nid*, *seen*, *i*) *None*;
    Some nid′ = *pred g nid*;
    seen′ = {*nid*} ∪ *seen*;
    CanonicalizationPhase g (*nid*′, *seen*′, *i*) *g*′⟧
    ⟹ *CanonicalizationPhase g* (*nid*, *seen*, *i*) *g*′ |

  ⟦*Step analyse g* (*nid*, *seen*, *i*) *None*;
    None = *pred g nid*⟧
    ⟹ *CanonicalizationPhase g* (*nid*, *seen*, *i*) *g*

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

type-synonym *Trace* = *IRNode list*
inductive *CanonicalizationPhaseWithTrace*
  :: *IRGraph* ⇒ (*ID* × *Seen* × *CanonicalizationAnalysis*) ⇒ *IRGraph* ⇒ *Trace* ⇒ *Trace* ⇒ *bool* **where**

  — Can do a step and optimise for the current node
  ⟦*Step analyse g* (*nid*, *seen*, *i*) (*Some* (*nid*′, *seen*′, *i*′));
    CanonicalizationStep g (*kind g nid*) *node*;

    g′ = *replace-node nid* (*node*, *stamp g nid*) *g*;

    CanonicalizationPhaseWithTrace g′ (*nid*′, *seen*′, *i*′) *g*′′ (*kind g nid* # *t*) *t*′ ⟧

$\Longrightarrow$ *CanonicalizationPhaseWithTrace g (nid, seen, i) g'' t t'* |

— Can do a step, matches whether optimised or not causing non-determinism We need to find a way to negate ConditionalEliminationStep

$[\![$*Step analyse g (nid, seen, i) (Some (nid', seen', i'))*;

CanonicalizationPhaseWithTrace g (nid', seen', i') g' (kind g nid # t) t' $]\!]$
$\Longrightarrow$ *CanonicalizationPhaseWithTrace g (nid, seen, i) g' t t'* |

$[\![$*Step analyse g (nid, seen, i) None*;
Some nid' = pred g nid;
seen' = {*nid*} $\cup$ *seen*;
CanonicalizationPhaseWithTrace g (nid', seen', i) g' (kind g nid # t) t' $]\!]$
$\Longrightarrow$ *CanonicalizationPhaseWithTrace g (nid, seen, i) g' t t'* |

$[\![$*Step analyse g (nid, seen, i) None*;
None = pred g nid$]\!]$
$\Longrightarrow$ *CanonicalizationPhaseWithTrace g (nid, seen, i) g t t*

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

end