Veriopt

April 24, 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:

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

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

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

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*

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*

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 *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*))
⟨*proof*⟩

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

lemma *intval-add* (*IntVal 32* ($2^{31}-1$)) (*IntVal 32* ($2^{31}-1$)) = *IntVal 32* (-2)
⟨*proof*⟩
lemma *intval-add* (*IntVal 64* ($2^{31}-1$)) (*IntVal 32* ($2^{31}-1$)) = *IntVal 64 4294967294*
⟨*proof*⟩

end

2 Nodes

2.1 Types of Nodes

theory *IRNodes*
imports
Values
begin

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

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

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

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

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

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

| *FrameState* (*ir-monitorIds*: *INPUT-ASSOC list*) (*ir-outerFrameState-opt*: *INPUT-STATE option*) (*ir-values-opt*: *INPUT list option*) (*ir-virtualObjectMappings-opt*: *INPUT-STATE list option*)
| *IfNode* (*ir-condition*: *INPUT-COND*) (*ir-trueSuccessor*: *SUCC*) (*ir-falseSuccessor*: *SUCC*)
| *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*, *trueValue*, *falseValue*] |
inputs-of-ConstantNode:
inputs-of (*ConstantNode const*) = [] |
inputs-of-DynamicNewArrayNode:
inputs-of (*DynamicNewArrayNode elementType length0 voidClass stateBefore next*) = [*elementType*, *length0*] @ (*opt-to-list voidClass*) @ (*opt-to-list stateBefore*) |
inputs-of-EndNode:
inputs-of (*EndNode*) = [] |
inputs-of-ExceptionObjectNode:
inputs-of (*ExceptionObjectNode stateAfter next*) = (*opt-to-list stateAfter*) |
inputs-of-FrameState:

inputs-of (*FrameState monitorIds outerFrameState values virtualObjectMappings*) = *monitorIds* @ (*opt-to-list outerFrameState*) @ (*opt-list-to-list values*) @ (*opt-list-to-list virtualObjectMappings*) |
inputs-of-IfNode:
inputs-of (*IfNode condition trueSuccessor falseSuccessor*) = [*condition*] |
inputs-of-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 *IRNodes*
begin

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

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

These functions have been automatically generated from the compiler.

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

fun *is-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 $\mathit{is\text{-}IntegerDivRemNode}$ :: $\mathit{IRNode} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}IntegerDivRemNode}\ n = ((\mathit{is\text{-}SignedDivNode}\ n) \vee (\mathit{is\text{-}SignedRemNode}\ n))$

fun $\mathit{is\text{-}FixedBinaryNode}$ :: $\mathit{IRNode} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}FixedBinaryNode}\ n = ((\mathit{is\text{-}IntegerDivRemNode}\ n))$

fun $\mathit{is\text{-}DeoptimizingFixedWithNextNode}$ :: $\mathit{IRNode} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}DeoptimizingFixedWithNextNode}\ n = ((\mathit{is\text{-}AbstractNewObjectNode}\ n) \vee (\mathit{is\text{-}FixedBinaryNode}\ n))$

fun $\mathit{is\text{-}AbstractMemoryCheckpoint}$ :: $\mathit{IRNode} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}AbstractMemoryCheckpoint}\ n = ((\mathit{is\text{-}BytecodeExceptionNode}\ n) \vee (\mathit{is\text{-}InvokeNode}\ n))$

fun $\mathit{is\text{-}AbstractStateSplit}$ :: $\mathit{IRNode} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}AbstractStateSplit}\ n = ((\mathit{is\text{-}AbstractMemoryCheckpoint}\ n))$

fun $\mathit{is\text{-}AccessFieldNode}$ :: $\mathit{IRNode} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}AccessFieldNode}\ n = ((\mathit{is\text{-}LoadFieldNode}\ n) \vee (\mathit{is\text{-}StoreFieldNode}\ n))$

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

fun $\mathit{is\text{-}WithExceptionNode}$ :: $\mathit{IRNode} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}WithExceptionNode}\ n = ((\mathit{is\text{-}InvokeWithExceptionNode}\ n))$

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

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

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

fun $\mathit{is\text{-}FloatingGuardedNode}$ :: $\mathit{IRNode} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}FloatingGuardedNode}\ n = ((\mathit{is\text{-}PiNode}\ n))$

fun $\mathit{is\text{-}UnaryArithmeticNode}$ :: $\mathit{IRNode} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}UnaryArithmeticNode}\ n = ((\mathit{is\text{-}AbsNode}\ n) \vee (\mathit{is\text{-}NegateNode}\ n) \vee (\mathit{is\text{-}NotNode}\ n))$

fun $\mathit{is\text{-}UnaryNode}$ :: $\mathit{IRNode} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}UnaryNode}\ n = ((\mathit{is\text{-}UnaryArithmeticNode}\ n))$

fun $\mathit{is\text{-}BinaryArithmeticNode}$ :: $\mathit{IRNode} \Rightarrow \mathit{bool}$ **where**

is-BinaryArithmeticNode n = ((*is-AddNode* n) $\vee$ (*is-AndNode* n) $\vee$ (*is-MulNode* n) $\vee$ (*is-OrNode* n) $\vee$ (*is-SubNode* n) $\vee$ (*is-XorNode* n))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

fun *is-LIRLowerable* :: *IRNode* $\Rightarrow$ *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* $\Rightarrow$ *bool* **where**
is-UncheckedInterfaceProvider n = ((*is-InvokeNode n*) $\vee$ (*is-InvokeWithExceptionNode n*) $\vee$ (*is-LoadFieldNode n*) $\vee$ (*is-ParameterNode n*))

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

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

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

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

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

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

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

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

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

fun *is-same-ir-node-type* :: *IRNode* ⇒ *IRNode* ⇒ *bool* **where**
is-same-ir-node-type n1 n2 = (
((*is-AbsNode n1*) ∧ (*is-AbsNode n2*)) ∨
((*is-AddNode n1*) ∧ (*is-AddNode n2*)) ∨
((*is-AndNode n1*) ∧ (*is-AndNode n2*)) ∨
((*is-BeginNode n1*) ∧ (*is-BeginNode n2*)) ∨
((*is-BytecodeExceptionNode n1*) ∧ (*is-BytecodeExceptionNode n2*)) ∨
((*is-ConditionalNode n1*) ∧ (*is-ConditionalNode n2*)) ∨
((*is-ConstantNode n1*) ∧ (*is-ConstantNode n2*)) ∨
((*is-DynamicNewArrayNode n1*) ∧ (*is-DynamicNewArrayNode n2*)) ∨
((*is-EndNode n1*) ∧ (*is-EndNode n2*)) ∨
((*is-ExceptionObjectNode n1*) ∧ (*is-ExceptionObjectNode n2*)) ∨
((*is-FrameState n1*) ∧ (*is-FrameState n2*)) ∨
((*is-IfNode n1*) ∧ (*is-IfNode n2*)) ∨

((*is-IntegerEqualsNode n1*) ∧ (*is-IntegerEqualsNode n2*)) ∨
((*is-IntegerLessThanNode n1*) ∧ (*is-IntegerLessThanNode n2*)) ∨
((*is-InvokeNode n1*) ∧ (*is-InvokeNode n2*)) ∨
((*is-InvokeWithExceptionNode n1*) ∧ (*is-InvokeWithExceptionNode n2*)) ∨
((*is-IsNullNode n1*) ∧ (*is-IsNullNode n2*)) ∨
((*is-KillingBeginNode n1*) ∧ (*is-KillingBeginNode n2*)) ∨
((*is-LoadFieldNode n1*) ∧ (*is-LoadFieldNode n2*)) ∨
((*is-LogicNegationNode n1*) ∧ (*is-LogicNegationNode n2*)) ∨
((*is-LoopBeginNode n1*) ∧ (*is-LoopBeginNode n2*)) ∨
((*is-LoopEndNode n1*) ∧ (*is-LoopEndNode n2*)) ∨
((*is-LoopExitNode n1*) ∧ (*is-LoopExitNode n2*)) ∨
((*is-MergeNode n1*) ∧ (*is-MergeNode n2*)) ∨
((*is-MethodCallTargetNode n1*) ∧ (*is-MethodCallTargetNode n2*)) ∨
((*is-MulNode n1*) ∧ (*is-MulNode n2*)) ∨
((*is-NegateNode n1*) ∧ (*is-NegateNode n2*)) ∨
((*is-NewArrayNode n1*) ∧ (*is-NewArrayNode n2*)) ∨
((*is-NewInstanceNode n1*) ∧ (*is-NewInstanceNode n2*)) ∨
((*is-NotNode n1*) ∧ (*is-NotNode n2*)) ∨
((*is-OrNode n1*) ∧ (*is-OrNode n2*)) ∨
((*is-ParameterNode n1*) ∧ (*is-ParameterNode n2*)) ∨
((*is-PiNode n1*) ∧ (*is-PiNode n2*)) ∨
((*is-ReturnNode n1*) ∧ (*is-ReturnNode n2*)) ∨
((*is-ShortCircuitOrNode n1*) ∧ (*is-ShortCircuitOrNode n2*)) ∨
((*is-SignedDivNode n1*) ∧ (*is-SignedDivNode n2*)) ∨
((*is-StartNode n1*) ∧ (*is-StartNode n2*)) ∨
((*is-StoreFieldNode n1*) ∧ (*is-StoreFieldNode n2*)) ∨
((*is-SubNode n1*) ∧ (*is-SubNode n2*)) ∨
((*is-UnwindNode n1*) ∧ (*is-UnwindNode n2*)) ∨
((*is-ValuePhiNode n1*) ∧ (*is-ValuePhiNode n2*)) ∨
((*is-ValueProxyNode n1*) ∧ (*is-ValueProxyNode n2*)) ∨
((*is-XorNode n1*) ∧ (*is-XorNode n2*)))

end

3 Stamp Typing

theory *Stamp*
imports *Values*
begin

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

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

datatype *Stamp* =

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

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

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

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

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

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

— A stamp which provides type information but has an empty range of values
fun *empty-stamp* :: *Stamp* ⇒ *Stamp* **where**
empty-stamp VoidStamp = *VoidStamp* |
empty-stamp (*IntegerStamp bits lower upper*) = (*IntegerStamp bits* (*snd* (*bit-bounds bits*)) (*fst* (*bit-bounds bits*))) |

empty-stamp (*KlassPointerStamp nonNull alwaysNull*) = (*KlassPointerStamp nonNull alwaysNull*) |
empty-stamp (*MethodCountersPointerStamp nonNull alwaysNull*) = (*MethodCountersPointerStamp nonNull alwaysNull*) |
empty-stamp (*MethodPointersStamp nonNull alwaysNull*) = (*MethodPointersStamp nonNull alwaysNull*) |
empty-stamp (*ObjectStamp type exactType nonNull alwaysNull*) = (*ObjectStamp*

$'''' True\ True\ False)$ |
  $empty\text{-}stamp\ stamp = IllegalStamp$

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

  $is\text{-}stamp\text{-}empty\ x = False$

— Calculate the meet stamp of two stamps
fun $meet :: Stamp \Rightarrow Stamp \Rightarrow Stamp$ **where**
  $meet\ VoidStamp\ VoidStamp = VoidStamp$ |
  $meet\ (IntegerStamp\ b1\ l1\ u1)\ (IntegerStamp\ b2\ l2\ u2) = ($
    $if\ b1 \neq b2\ then\ IllegalStamp\ else$
    $(IntegerStamp\ b1\ (min\ l1\ l2)\ (max\ u1\ u2))$
  $)$ |

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

— Calculate the join stamp of two stamps
fun $join :: Stamp \Rightarrow Stamp \Rightarrow Stamp$ **where**
  $join\ VoidStamp\ VoidStamp = VoidStamp$ |
  $join\ (IntegerStamp\ b1\ l1\ u1)\ (IntegerStamp\ b2\ l2\ u2) = ($
    $if\ b1 \neq b2\ then\ IllegalStamp\ else$
    $(IntegerStamp\ b1\ (max\ l1\ l2)\ (min\ u1\ u2))$
  $)$ |

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

then (*empty-stamp* (*MethodPointersStamp nn1 an1*))
else (*MethodPointersStamp* (*nn1* ∨ *nn2*) (*an1* ∨ *an2*))
) |
join s1 s2 = *IllegalStamp*

— In certain circumstances a stamp provides enough information to evaluate a value as a stamp, the asConstant function converts the stamp to a value where one can be inferred.
fun *asConstant* :: *Stamp* ⇒ *Value* **where**
asConstant (*IntegerStamp b l h*) = (*if l* = *h then IntVal b l else UndefVal*) |
asConstant - = *UndefVal*

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

— Determine if two stamps must always be the same value i.e. two equal constants
fun *neverDistinct* :: *Stamp* ⇒ *Stamp* ⇒ *bool* **where**
neverDistinct stamp1 stamp2 = (*asConstant stamp1* = *asConstant stamp2* ∧ *asConstant stamp1* ≠ *UndefVal*)

fun *constantAsStamp* :: *Value* ⇒ *Stamp* **where**
constantAsStamp (*IntVal b v*) = (*IntegerStamp* (*nat b*) *v v*) |

constantAsStamp - = *IllegalStamp*

— Define when a runtime value is valid for a stamp
fun *valid-value* :: *Stamp* ⇒ *Value* ⇒ *bool* **where**
valid-value (*IntegerStamp b1 l h*) (*IntVal b2 v*) = ((*b1* = *b2*) ∧ (*v* ≥ *l*) ∧ (*v* ≤ *h*)) |

valid-value (*VoidStamp*) (*UndefVal*) = *True* |
valid-value stamp val = *False*

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

lemma *int-valid-range*:
assumes *stamp* = *IntegerStamp bits lower upper*
shows {*x* . *valid-value stamp x*} = {(*IntVal bits val*) | *val* . *val* ∈ {*lower..upper*}}
⟨*proof*⟩

lemma *disjoint-empty*:

assumes *joined* = (*join x-stamp y-stamp*)
assumes *is-stamp-empty joined*
shows $\{x\ .\ \textit{valid-value x-stamp}\ x\} \cap \{y\ .\ \textit{valid-value y-stamp}\ y\} = \{\}$
⟨*proof*⟩

lemma *join-unequal*:
assumes *joined* = (*join x-stamp y-stamp*)
assumes *is-stamp-empty joined*
shows $\nexists\ x\ y\ .\ x = y \wedge \textit{valid-value x-stamp}\ x \wedge \textit{valid-value y-stamp}\ y$
⟨*proof*⟩

lemma *neverDistinctEqual*:
assumes *neverDistinct x-stamp y-stamp*
shows $\nexists\ x\ y\ .\ x \neq y \wedge \textit{valid-value x-stamp}\ x \wedge \textit{valid-value y-stamp}\ y$
⟨*proof*⟩

lemma *boundsNoOverlapNoEqual*:
assumes *stpi-upper x-stamp* < *stpi-lower y-stamp*
assumes *is-IntegerStamp x-stamp* ∧ *is-IntegerStamp y-stamp*
shows $\nexists\ x\ y\ .\ x = y \wedge \textit{valid-value x-stamp}\ x \wedge \textit{valid-value y-stamp}\ y$
⟨*proof*⟩

lemma *boundsNoOverlap*:
assumes *stpi-upper x-stamp* < *stpi-lower y-stamp*
assumes *x* = *IntVal b1 xval*
assumes *y* = *IntVal b2 yval*
assumes *is-IntegerStamp x-stamp* ∧ *is-IntegerStamp y-stamp*
assumes *valid-value x-stamp x* ∧ *valid-value y-stamp y*
shows *xval* < *yval*
⟨*proof*⟩

lemma *boundsAlwaysOverlap*:
assumes *stpi-lower x-stamp* ≥ *stpi-upper y-stamp*
assumes *x* = *IntVal b1 xval*
assumes *y* = *IntVal b2 yval*
assumes *is-IntegerStamp x-stamp* ∧ *is-IntegerStamp y-stamp*
assumes *valid-value x-stamp x* ∧ *valid-value y-stamp y*
shows ¬(*xval* < *yval*)
⟨*proof*⟩

lemma *intstamp-bits-eq-meet*:
assumes (*meet* (*IntegerStamp b1 l1 u1*) (*IntegerStamp b2 l2 u2*)) = (*IntegerStamp b3 l3 u3*)
shows *b1* = *b3* ∧ *b2* = *b3*
⟨*proof*⟩

lemma *intstamp-bits-eq-join*:
assumes (*join* (*IntegerStamp b1 l1 u1*) (*IntegerStamp b2 l2 u2*)) = (*IntegerStamp b3 l3 u3*)

shows $b1 = b3 \wedge b2 = b3$
⟨*proof*⟩

lemma *intstamp-bites-eq-unrestricted*:
assumes $(unrestricted\text{-}stamp\ (IntegerStamp\ b1\ l1\ u1)) = (IntegerStamp\ b2\ l2\ u2)$
shows $b1 = b2$
⟨*proof*⟩

lemma *intstamp-bits-eq-empty*:
assumes $(empty\text{-}stamp\ (IntegerStamp\ b1\ l1\ u1)) = (IntegerStamp\ b2\ l2\ u2)$
shows $b1 = b2$
⟨*proof*⟩

notepad
begin
⟨*proof*⟩
end

end

4 Graph Representation

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

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

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

typedef $IRGraph = \{g :: ID \rightharpoonup (IRNode \times Stamp)\ .\ finite\ (dom\ g)\}$
⟨*proof*⟩

setup-lifting *type-definition-IRGraph*

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

fun $with\text{-}default :: 'c \Rightarrow ('b \Rightarrow 'c) \Rightarrow (('a \rightharpoonup 'b) \Rightarrow 'a \Rightarrow 'c)$ **where**
$with\text{-}default\ def\ conv = (\lambda m\ k.$

$(case\ m\ k\ of\ None \Rightarrow def \mid Some\ v \Rightarrow conv\ v))$

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

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

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

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

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

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

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 $map\text{-}of \circ no\text{-}node$
⟨*proof*⟩

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

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

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

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

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

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

— Get the successor set of a given node ID

fun *succ* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
succ g nid = *set* (*successors-of* (*kind g nid*))

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

fun *input-edges* :: *IRGraph* ⇒ *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*) $\longrightarrow$ *ids* (*Abs-IRGraph g*) = {*nid* $\in$ *dom g* . $\nexists$ *s*. *g nid* = *Some* (*NoNode*, *s*)}
⟨*proof*⟩

lemma [*simp*]: *finite* (*dom g*) $\longrightarrow$ *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*) = (λ*x* . (*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*]:
$gup = replace\text{-}node\ nid\ (k,\ s)\ g \wedge k \neq NoNode \longrightarrow kind\ gup\ nid = k \wedge stamp\ gup\ nid = s$
⟨*proof*⟩

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

datatype *MapState* =
  MapState
    (*m-values*: *ID* ⇒ *Value*)
    (*m-params*: *Value list*)

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

fun *m-val* :: *MapState* ⇒ *ID* ⇒ *Value* **where**
  m-val m nid = (*m-values m*) *nid*

fun *m-set* :: *ID* ⇒ *Value* ⇒ *MapState* ⇒ *MapState* **where**
  m-set nid v (*MapState m p*) = *MapState* (*m*(*nid* := *v*)) *p*

fun *m-param* :: *IRGraph* ⇒ *MapState* ⇒ *ID* ⇒ *Value* **where**
  m-param g m nid = (*case* (*kind g nid*) *of*
    (*ParameterNode i*) ⇒ (*m-params m*)!*i* |
    - ⇒ *UndefVal*)

fun *set-params* :: *MapState* ⇒ *Value list* ⇒ *MapState* **where**
  set-params (*MapState m* -) *vs* = *MapState m vs*

fun *new-map* :: *Value list* ⇒ *MapState* **where**
  new-map ps = *set-params new-map-state ps*

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

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

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

inductive
eval :: *IRGraph ⇒ MapState ⇒ IRNode ⇒ Value ⇒ bool* (- - ⊢ - ↦ - *55*)
for *g* **where**

ConstantNode:
g m ⊢ (*ConstantNode c*) ↦ *c* |

ParameterNode:
g m ⊢ (*ParameterNode i*) ↦ (*m-params m*)!*i* |

ValuePhiNode:
g m ⊢ (*ValuePhiNode nid - -*) ↦ *m-val m nid* |

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

— Unary arithmetic operators

AbsNode:
⟦*g m* ⊢ (*kind g x*) ↦ *IntVal b v*⟧
⟹ *g m* ⊢ (*AbsNode x*) ↦ *if v < 0 then (intval-sub (IntVal b 0) (IntVal b v)) else (IntVal b v)* |

NegateNode:
⟦*g m* ⊢ (*kind g x*) ↦ *v*⟧
⟹ *g m* ⊢ (*NegateNode x*) ↦ (*IntVal* (*v-bits v*) *0*) − *v* |

NotNode:
$[\![g\ m \vdash (kind\ g\ x) \mapsto val;$
$\ \ not\text{-}val = (\neg(val\text{-}to\text{-}bool\ val))\ ]\!]$
$\ \ \Longrightarrow g\ m \vdash (NotNode\ x) \mapsto bool\text{-}to\text{-}val\ not\text{-}val\ |$

— Binary arithmetic operators

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

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

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

SignedDivNode:
$g\ m \vdash (SignedDivNode\ nid\ \text{- - - - -}) \mapsto m\text{-}val\ m\ nid\ |$

SignedRemNode:
$g\ m \vdash (SignedRemNode\ nid\ \text{- - - - -}) \mapsto m\text{-}val\ m\ nid\ |$

— Binary logical bitwise operators

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

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

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

— Comparison operators

IntegerEqualsNode:
$[\![g\ m \vdash (kind\ g\ x) \mapsto IntVal\ b\ v1;$

$g\ m \vdash (kind\ g\ y) \mapsto IntVal\ b\ v2;$
$val = bool\text{-}to\text{-}val(v1 = v2)\rrbracket$
$\Longrightarrow g\ m \vdash (IntegerEqualsNode\ x\ y) \mapsto val\ |$

IntegerLessThanNode:
$\llbracket g\ m \vdash (kind\ g\ x) \mapsto IntVal\ b\ v1;$
$g\ m \vdash (kind\ g\ y) \mapsto IntVal\ b\ v2;$
$val = bool\text{-}to\text{-}val(v1 < v2)\rrbracket$
$\Longrightarrow g\ m \vdash (IntegerLessThanNode\ x\ y) \mapsto val\ |$

IsNullNode:
$\llbracket g\ m \vdash (kind\ g\ obj) \mapsto ObjRef\ ref;$
$val = bool\text{-}to\text{-}val(ref = None)\rrbracket$
$\Longrightarrow g\ m \vdash (IsNullNode\ obj) \mapsto val\ |$

— Other nodes

ConditionalNode:
$\llbracket g\ m \vdash (kind\ g\ condition) \mapsto IntVal\ 1\ cond;$
$g\ m \vdash (kind\ g\ trueExp) \mapsto IntVal\ b\ trueVal;$
$g\ m \vdash (kind\ g\ falseExp) \mapsto IntVal\ b\ falseVal;$
$val = IntVal\ b\ (if\ cond \neq 0\ then\ trueVal\ else\ falseVal)\rrbracket$
$\Longrightarrow g\ m \vdash (ConditionalNode\ condition\ trueExp\ falseExp) \mapsto val\ |$

ShortCircuitOrNode:
$\llbracket g\ m \vdash (kind\ g\ x) \mapsto IntVal\ b\ v1;$
$g\ m \vdash (kind\ g\ y) \mapsto IntVal\ b\ v2;$
$val = IntVal\ b\ (if\ v1 \neq 0\ then\ v1\ else\ v2)\rrbracket$
$\Longrightarrow g\ m \vdash (ShortCircuitOrNode\ x\ y) \mapsto val\ |$

LogicNegationNode:
$\llbracket g\ m \vdash (kind\ g\ x) \mapsto IntVal\ 1\ v1;$
$val = IntVal\ 1\ (NOT\ v1)\rrbracket$
$\Longrightarrow g\ m \vdash (LogicNegationNode\ x) \mapsto val\ |$

InvokeNodeEval:
$g\ m \vdash (InvokeNode\ nid\ \text{- - - - -}) \mapsto m\text{-}val\ m\ nid\ |$

InvokeWithExceptionNodeEval:
$g\ m \vdash (InvokeWithExceptionNode\ nid\ \text{- - - - - -}) \mapsto m\text{-}val\ m\ nid\ |$

NewInstanceNode:
$g\ m \vdash (NewInstanceNode\ nid\ \text{- - -}) \mapsto m\text{-}val\ m\ nid\ |$

LoadFieldNode:

$g\ m \vdash (LoadFieldNode\ nid\ \text{-}\ \text{-}\ \text{-}) \mapsto m\text{-}val\ m\ nid\ |$

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

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

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$ *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-all :: $IRGraph \Rightarrow MapState \Rightarrow ID\ list \Rightarrow Value\ list \Rightarrow bool$
$(\text{-}\ \text{-} \vdash \text{-} \longmapsto \text{-}\ 55)$
for g **where**
Base:
$g\ m \vdash [] \longmapsto []\ |$

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

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

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

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

values $\{v.\ eval\text{-}graph\ eg2\text{-}sq\ 4\ [IntVal\ 32\ 5]\ v\}$

fun *has-control-flow* :: $IRNode \Rightarrow bool$ **where**
$has\text{-}control\text{-}flow\ n = (is\text{-}AbstractEndNode\ n$
$\vee (length\ (successors\text{-}of\ n) > 0))$

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* ⊢ (*AddNode x y*) ↦ *val* ⟹
(∃ *v1*. (*g m* ⊢ (*kind g x*) ↦ *v1*) ∧
(∃ *v2*. (*g m* ⊢ (*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* ⊢ *node* ↦ *val1*) ⟹
(∀ *val2*. ((*g m* ⊢ *node* ↦ *val2*) ⟶ *val1* = *val2*))
⟨*proof*⟩

theorem *evalAllDet*:
(*g m* ⊢ *nodes* ⟼ *vals1*) ⟹
(∀ *vals2*. ((*g m* ⊢ *nodes* ⟼ *vals2*) ⟶ *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-load-field* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b)$ *DynamicHeap* $\Rightarrow$ *Value* **where**
h-load-field f r $(h, n) = h\ f\ r$

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

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

type-synonym *FieldRefHeap* $=$ (*string*, *objref*) *DynamicHeap*

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

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* $\Rightarrow$ (*ID* $\times$ *MapState* $\times$ *FieldRefHeap*) $\Rightarrow$ (*ID* $\times$ *MapState* $\times$ *FieldRefHeap*) $\Rightarrow$ *bool*
(- $\vdash$ - $\rightarrow$ - *55*) **for** *g* **where**

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

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

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

i = *find-index nid* (*inputs-of* (*kind g merge*));
phis = (*phi-list g merge*);
inps = (*phi-inputs g i phis*);
$g\ m \vdash inps \longmapsto vs$;

$m' = $ *set-phis phis vs m*$]\!]$
$\Longrightarrow g \vdash (nid,\ m,\ h) \rightarrow (merge,\ m',\ h)$ |

NewInstanceNode:
$[\![$*kind g nid* = (*NewInstanceNode nid f obj nid'*);
(h', *ref*) = *h-new-inst h*;
m' = *m-set nid ref m*$]\!]$
$\Longrightarrow g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h')$ |

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

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

SignedRemNode:
$[\![$*kind g nid* = (*SignedRemNode nid x y zero sb nxt*);
$g\ m \vdash$ (*kind g x*) $\mapsto$ *v1*;
$g\ m \vdash$ (*kind g y*) $\mapsto$ *v2*;
v = (*intval-mod v1 v2*);
m' = *m-set nid v m*$]\!]$
$\Longrightarrow g \vdash (nid,\ m,\ h) \rightarrow (nxt,\ m',\ h)$ |

StaticLoadFieldNode:
$[\![$*kind g nid* = (*LoadFieldNode nid f None nid'*);
h-load-field f None h = v;
m' = *m-set nid v m*$]\!]$
$\Longrightarrow g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h)$ |

StoreFieldNode:

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

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

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

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*:
$(g \vdash (nid,m,h) \rightarrow next) \Longrightarrow$
$(\forall\ next'.\ ((g \vdash (nid,m,h) \rightarrow next') \longrightarrow next = next'))$
$\langle proof\rangle$

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

lemma *IfNodeStepCases*:
assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
assumes $g\ m \vdash kind\ g\ cond \mapsto v$
assumes $g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)$
shows $nid' \in \{tb,\ fb\}$
$\langle proof\rangle$

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

lemma *IfNodeCond*:
assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
assumes $g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)$
shows $\exists\ v.\ (g\ m \vdash kind\ g\ cond \mapsto v)$
$\langle proof\rangle$

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

6.3 Interprocedural Semantics

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

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

Lift:
⟦*g* ⊢ (*nid*, *m*, *h*) → (*nid'*, *m'*, *h'*)⟧
⟹ *p* ⊢ ((*g*,*nid*,*m*)#*stk*, *h*) ⟶ ((*g*,*nid'*,*m'*)#*stk*, *h'*) |

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

callTarget = *ir-callTarget* (*kind g nid*);
kind g callTarget = (*MethodCallTargetNode targetMethod arguments*);
Some targetGraph = *p targetMethod*;

g m ⊢ *arguments* ⟼ *vs*;
m' = *set-params m vs*⟧
⟹ *p* ⊢ ((*g*,*nid*,*m*)#*stk*, *h*) ⟶ ((*targetGraph*,*0*,*m'*)#(*g*,*nid*,*m*)#*stk*, *h*) |

ReturnNode:
⟦*kind g nid* = (*ReturnNode* (*Some expr*) -);
g m ⊢ (*kind g expr*) ↦ *v*;

c-m' = *m-set c-nid v c-m*;
c-nid' = (*successors-of* (*kind c-g c-nid*))!*0*⟧
⟹ *p* ⊢ ((*g*,*nid*,*m*)#(*c-g*,*c-nid*,*c-m*)#*stk*, *h*) ⟶ ((*c-g*,*c-nid'*,*c-m'*)#*stk*, *h*) |

ReturnNodeVoid:
⟦*kind g nid* = (*ReturnNode None* -);
c-m' = *m-set c-nid* (*ObjRef* (*Some* (*2048*))) *c-m*;

c-nid' = (*successors-of* (*kind c-g c-nid*))!*0*⟧
⟹ *p* ⊢ ((*g*,*nid*,*m*)#(*c-g*,*c-nid*,*c-m*)#*stk*, *h*) ⟶ ((*c-g*,*c-nid'*,*c-m'*)#*stk*, *h*) |

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

g m ⊢ (*kind g exception*) ↦ *e*;

kind c-g c-nid = (*InvokeWithExceptionNode* - - - - - - *exEdge*);

c-m' = *m-set c-nid e c-m*⟧
⟹ *p* ⊢ ((*g*,*nid*,*m*)#(*c-g*,*c-nid*,*c-m*)#*stk*, *h*) ⟶ ((*c-g*,*exEdge*,*c-m'*)#*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*) *list*

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

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

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

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

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

  $l' = (l \ @\ [(g,nid,m)])$⟧
  ⟹ *exec* $p\ (((g,nid,m)\#xs),h)\ l\ (((g',nid',m')\#ys),h')\ l'$

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*) *list* × *FieldRefHeap*
  ⇒ *nat*
  ⇒ (*IRGraph* × *ID* × *MapState*) *list* × *FieldRefHeap*
  ⇒ *bool*
  (-⊢-⟶*-* -)
  where
  ⟦$n > 0$;
  $p \vdash s \longrightarrow s'$;
  exec-debug $p\ s'\ (n - 1)\ s''$⟧
  ⟹ *exec-debug* $p\ s\ n\ s''$ |

  ⟦$n = 0$⟧
  ⟹ *exec-debug* $p\ s\ n\ s$

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

6.4.1 Heap Testing

definition *p3*:: *MapState* **where**
p3 = *set-params new-map-state* [*IntVal 32 3*]

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

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

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

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

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

values {*h-load-field field-sq* (*Some 0*) (*prod.snd res*)
| *res*. (λx. *Some eg4-sq*) ⊢ ([(*eg4-sq*, *0*, *p3*), (*eg4-sq*, *0*, *p3*)], *new-heap*) →∗*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**
  $[\![\forall P'. (g\ m\ h \vdash nid \rightsquigarrow P') \longrightarrow (\exists Q'. (g'\ m\ h \vdash nid \rightsquigarrow Q') \wedge P' = Q');$
    $\forall Q'. (g'\ m\ h \vdash nid \rightsquigarrow Q') \longrightarrow (\exists P'. (g\ m\ h \vdash nid \rightsquigarrow P') \wedge P' = Q')]\!]$
  $\Longrightarrow nid\ .\ g \sim g'$

A strong bisimilution between no-op transitions

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

lemma *lockstep-strong-bisimilulation*:
  assumes $g' = replace\text{-}node\ nid\ node\ g$
  assumes $g \vdash (nid, m, h) \rightarrow (nid', m, h)$
  assumes $g' \vdash (nid, m, h) \rightarrow (nid', m, h)$
  shows $nid \mid g \sim g'$
  $\langle proof \rangle$

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

end

7.2 Formedness Properties

theory *Form*
imports
  Semantics.IREval
begin

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

definition *wf-closed* **where**
  $\mathit{wf\text{-}closed}\ g =$
    $(\forall\ n \in \mathit{ids}\ g\ .$
      $\mathit{inputs}\ g\ n \subseteq \mathit{ids}\ g\ \wedge$
      $\mathit{succ}\ g\ n \subseteq \mathit{ids}\ g\ \wedge$
      $\mathit{kind}\ g\ n \neq \mathit{NoNode})$

definition *wf-phis* **where**
  $\mathit{wf\text{-}phis}\ g =$
    $(\forall\ n \in \mathit{ids}\ g.$
      $\mathit{is\text{-}PhiNode}\ (\mathit{kind}\ g\ n) \longrightarrow$
      $\mathit{length}\ (\mathit{ir\text{-}values}\ (\mathit{kind}\ g\ n))$
       $= \mathit{length}\ (\mathit{ir\text{-}ends}$
          $(\mathit{kind}\ g\ (\mathit{ir\text{-}merge}\ (\mathit{kind}\ g\ n)))))$

definition *wf-ends* **where**
  $\mathit{wf\text{-}ends}\ g =$
    $(\forall\ n \in \mathit{ids}\ g\ .$
      $\mathit{is\text{-}AbstractEndNode}\ (\mathit{kind}\ g\ n) \longrightarrow$
      $\mathit{card}\ (\mathit{usages}\ g\ n) > 0)$

fun *wf-graph* :: *IRGraph* ⇒ *bool* **where**
  $\mathit{wf\text{-}graph}\ g = (\mathit{wf\text{-}start}\ g \wedge \mathit{wf\text{-}closed}\ g \wedge \mathit{wf\text{-}phis}\ g \wedge \mathit{wf\text{-}ends}\ g)$

lemmas *wf-folds* =
  wf-graph.simps
  wf-start-def
  wf-closed-def
  wf-phis-def
  wf-ends-def

fun *wf-stamps* :: *IRGraph* ⇒ *bool* **where**
  $\mathit{wf\text{-}stamps}\ g = (\forall\ n \in \mathit{ids}\ g\ .$
    $(\forall\ v\ m\ .\ (g\ m \vdash (\mathit{kind}\ g\ n) \mapsto v) \longrightarrow \mathit{valid\text{-}value}\ (\mathit{stamp}\ g\ n)\ v))$

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

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

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

fun *wf-values* :: *IRGraph* ⇒ *bool* **where**
  $\mathit{wf\text{-}values}\ g = (\forall\ n \in \mathit{ids}\ g\ .$
    $(\forall\ v\ m\ .\ (g\ m \vdash \mathit{kind}\ g\ n \mapsto v) \longrightarrow \mathit{wf\text{-}value}\ v))$

lemma *wf-value-range*:
$b > 1 \wedge b \in \textit{int-bits-allowed} \longrightarrow \{v.\ \textit{wf-value}\ (\textit{IntVal}\ b\ v)\} = \{v.\ ((-(2\widehat{\ }(b-1)) \leq v) \wedge (v < (2\widehat{\ }(b-1))))\}$
⟨*proof*⟩

lemma *wf-value-bit-range*:
$b = 1 \longrightarrow \{v.\ \textit{wf-value}\ (\textit{IntVal}\ b\ v)\} = \{\}$
⟨*proof*⟩

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* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
$\textit{unchanged ns g1 g2} = (\forall\ n\ .\ n \in ns \longrightarrow$
$(n \in \textit{ids g1} \wedge n \in \textit{ids g2} \wedge \textit{kind g1 n} = \textit{kind g2 n}))$

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

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 $nid \in \textit{ids g1}$
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* ∈ *ids g*
    ⟹ *eval-uses g nid nid* |

  use-inp: *nid′* ∈ *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* ∈ *ids g* . *eval-uses g nid n*}

lemma *eval-usages-self*:
  assumes *nid* ∈ *ids g*
  shows *nid* ∈ *eval-usages g nid*
  ⟨*proof*⟩

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

lemma *child-member*:
  assumes *n* = *kind g nid*
  assumes *n* ≠ *NoNode*
  assumes *List.member* (*inputs-of n*) *child*
  shows *child* ∈ *inputs g nid*
  ⟨*proof*⟩

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

lemma *inp-in-g*:
  assumes *n* ∈ *inputs g nid*
  shows *nid* ∈ *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-of k*)
shows *child* $\in$ *inputs g nid*
⟨*proof*⟩

lemma *eval-in-ids*:

assumes $g\ m \vdash (kind\ g\ nid) \mapsto v$
shows $nid \in ids\ g$
$\langle proof \rangle$

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

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

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

lemma *no-external-use*:
assumes $wf\text{-}graph\ g$
assumes $nid' \notin ids\ g$

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

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

fun *nextNid* :: *IRGraph* ⇒ *ID* **where**
nextNid g = (*Max* (*ids g*)) + *1*

lemma *max-plus-one*:
fixes *c* :: *ID set*
shows ⟦*finite c*; *c* ≠ {}⟧ ⟹ (*Max c*) + *1* ∉ *c*
⟨*proof*⟩

lemma *ids-finite*:
finite (*ids g*)
⟨*proof*⟩

lemma *nextNidNotIn*:
$ids\ g \neq \{\} \longrightarrow nextNid\ g \notin ids\ g$
⟨*proof*⟩

fun *constantCondition* :: $bool \Rightarrow ID \Rightarrow IRNode \Rightarrow IRGraph \Rightarrow IRGraph$ **where**
constantCondition val nid (*IfNode cond t f*) *g* =
replace-node nid (*IfNode* (*nextNid g*) *t f*, *stamp g nid*)
(*add-node* (*nextNid g*) ((*ConstantNode* (*bool-to-val val*)), *default-stamp*) *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 $g' \vdash (ifcond,\ m,\ h) \rightarrow (t,\ m,\ h)$
⟨*proof*⟩

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

lemma *diff-forall*:
assumes $\forall\, n{\in}ids\ g - \{nid\}.\ cond\ n$
shows $\forall\, n.\ n \in ids\ g \wedge n \notin \{nid\} \longrightarrow cond\ n$
⟨*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*)), *default-stamp*) *g*)
⟨*proof*⟩

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

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

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

end

7.5 Stuttering

theory *Stuttering*
  imports
    Semantics.IRStepObj
begin

inductive *stutter*:: $IRGraph \Rightarrow MapState \Rightarrow FieldRefHeap \Rightarrow ID \Rightarrow ID \Rightarrow bool$ (-
- - ⊢ - ⇝ - 55)
  for $g\ m\ h$ **where**

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

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

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

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-upper (*stamp g tb*) ≤ *stpi-lower* (*stamp g fb*)⟧
  ⟹ *CanonicalizeConditional g* (*ConditionalNode cond tb fb*) (*RefNode tb*) |

  condition-bounds-y:
  ⟦*kind g cond* = *IntegerLessThanNode fb tb*;
    stpi-upper (*stamp g fb*) ≤ *stpi-lower* (*stamp g tb*)⟧
  ⟹ *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-1*;
kind g y = *ConstantNode c-2*;
val = *intval-add c-1 c-2*⟧
⟹ *CanonicalizeAdd g* (*AddNode x y*) (*ConstantNode val*) |

add-xzero:
⟦*kind g x* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g y*));
c-1 = (*IntVal 32 0*)⟧
⟹ *CanonicalizeAdd g* (*AddNode x y*) (*RefNode y*) |

add-yzero:
⟦¬(*is-ConstantNode* (*kind g x*));
kind g y = *ConstantNode c-2*;
c-2 = (*IntVal 32 0*)⟧
⟹ *CanonicalizeAdd g* (*AddNode x y*) (*RefNode x*) |

add-xsub:

⟦*kind g x* = *SubNode a y* ⟧
⟹ *CanonicalizeAdd g* (*AddNode x y*) (*RefNode a*) |

add-ysub:

⟦*kind g y* = *SubNode a x* ⟧
⟹ *CanonicalizeAdd g* (*AddNode x y*) (*RefNode a*) |

add-xnegate:

⟦*kind g nx* = *NegateNode x* ⟧
⟹ *CanonicalizeAdd g* (*AddNode nx y*) (*SubNode y x*) |

add-ynegate:

⟦*kind g ny* = *NegateNode y* ⟧
⟹ *CanonicalizeAdd g* (*AddNode x ny*) (*SubNode x y*)

inductive *CanonicalizeIf* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
trueConst:
⟦*kind g cond* = *ConstantNode condv*;
val-to-bool condv⟧
⟹ *CanonicalizeIf g* (*IfNode cond tb fb*) (*RefNode tb*) |

falseConst:
⟦*kind g cond* = *ConstantNode condv*;
¬(*val-to-bool condv*)⟧
⟹ *CanonicalizeIf g* (*IfNode cond tb fb*) (*RefNode fb*) |

eqBranch:
⟦¬(*is-ConstantNode* (*kind g cond*));
tb = *fb*⟧
⟹ *CanonicalizeIf g* (*IfNode cond tb fb*) (*RefNode tb*) |

eqCondition:
⟦*kind g cond* = *IntegerEqualsNode x x*⟧
⟹ *CanonicalizeIf g* (*IfNode cond tb fb*) (*RefNode tb*)

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

inductive *CanonicalizeCommutativeBinaryArithmeticNode* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**

add-ids-ordered:

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

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

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

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

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

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

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

and-swap-const-first:
$[\![\textit{is-ConstantNode}\ (\textit{kind}\ g\ x);$
$\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y))]\!]$
$\Longrightarrow \textit{CanonicalizeCommutativeBinaryArithmeticNode}\ g\ (\textit{AndNode}\ x\ y)\ (\textit{AndNode}\ y\ x)$ |

int-equals-swap-const-first:

$[\![is\text{-}ConstantNode\ (kind\ g\ x);$
$\neg(is\text{-}ConstantNode\ (kind\ g\ y))]\!]$
$\Longrightarrow CanonicalizeCommutativeBinaryArithmeticNode\ g\ (IntegerEqualsNode\ x\ y)\ (IntegerEqualsNode\ y\ x)$ |

mul-swap-const-first:
$[\![is\text{-}ConstantNode\ (kind\ g\ x);$
$\neg(is\text{-}ConstantNode\ (kind\ g\ y))]\!]$
$\Longrightarrow CanonicalizeCommutativeBinaryArithmeticNode\ g\ (MulNode\ x\ y)\ (MulNode\ y\ x)$ |

or-swap-const-first:
$[\![is\text{-}ConstantNode\ (kind\ g\ x);$
$\neg(is\text{-}ConstantNode\ (kind\ g\ y))]\!]$
$\Longrightarrow CanonicalizeCommutativeBinaryArithmeticNode\ g\ (OrNode\ x\ y)\ (OrNode\ y\ x)$ |

xor-swap-const-first:
$[\![is\text{-}ConstantNode\ (kind\ g\ x);$
$\neg(is\text{-}ConstantNode\ (kind\ g\ y))]\!]$
$\Longrightarrow CanonicalizeCommutativeBinaryArithmeticNode\ g\ (XorNode\ x\ y)\ (XorNode\ y\ x)$

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

sub-both-const:
$[\![kind\ g\ x = ConstantNode\ c\text{-}1;$
$kind\ g\ y = ConstantNode\ c\text{-}2;$
$val = intval\text{-}sub\ c\text{-}1\ c\text{-}2]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ x\ y)\ (ConstantNode\ val)$ |

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:

$[\![kind\ g\ y = ConstantNode\ (IntVal\ \text{-}\ 0)]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ x\ y)\ (RefNode\ x)$ |

sub-xzero:

$[\![kind\ g\ x = ConstantNode\ (IntVal\ \text{-}\ 0)]\!]$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ x\ y)\ (NegateNode\ y)$ |

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

mul-yzero:
$[\![$*kind g y* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g x*));
c-1 = (*IntVal b 0*)$]\!]$
$\Longrightarrow$ *CanonicalizeMul g* (*MulNode x y*) (*ConstantNode c-1*) |

mul-xone:
$[\![$*kind g x* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g y*));
c-1 = (*IntVal b 1*)$]\!]$
$\Longrightarrow$ *CanonicalizeMul g* (*MulNode x y*) (*RefNode y*) |

mul-yone:
$[\![$*kind g y* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g x*));
c-1 = (*IntVal b 1*)$]\!]$
$\Longrightarrow$ *CanonicalizeMul g* (*MulNode x y*) (*RefNode x*) |

mul-xnegate:
$[\![$*kind g x* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g y*));
c-1 = (*IntVal b* (−*1*))$]\!]$
$\Longrightarrow$ *CanonicalizeMul g* (*MulNode x y*) (*NegateNode y*) |

mul-ynegate:
$[\![$*kind g y* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g x*));
c-1 = (*IntVal b* (−*1*))$]\!]$
$\Longrightarrow$ *CanonicalizeMul g* (*MulNode x y*) (*NegateNode x*)

inductive *CanonicalizeAbs* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *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*);

$val = (IntVal\ b\ v)$;
$neg\text{-}val = intval\text{-}sub\ (IntVal\ b\ 0)\ val$ ⟧
$\Longrightarrow$ *CanonicalizeNegate g (NegateNode nx) (ConstantNode neg-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 - - -)⟧
$\Longrightarrow$ *CanonicalizeNegate g (NegateNode sub) (SubNode y x)*

inductive *CanonicalizeNot* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
not-const:
⟦*kind g nx = (ConstantNode val)*;
neg-val = bool-to-val (¬(*val-to-bool val*)) ⟧
$\Longrightarrow$ *CanonicalizeNot g (NotNode nx) (ConstantNode neg-val)* |

not-not:
⟦*kind g nx = (NotNode x)*⟧
$\Longrightarrow$ *CanonicalizeNot g (NotNode 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*;
¬(*val-to-bool val*)⟧
$\Longrightarrow$ *CanonicalizeAnd g (AndNode x y) (ConstantNode val)* |

and-yfalse:
⟦*kind g y = ConstantNode val*;

$\neg(\textit{val-to-bool val})]\!]$
$\Longrightarrow$ *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-to-bool val*)$]\!]$
$\Longrightarrow$ *CanonicalizeOr g* (*OrNode x y*) (*RefNode x*)

inductive *CanonicalizeDeMorgansLaw* :: *ID* $\Rightarrow$ *IRGraph* $\Rightarrow$ *IRGraph* $\Rightarrow$ *bool*
where

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

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

int-equals-distinct:
$[\![$*alwaysDistinct* (*stamp g x*) (*stamp g y*)$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals g* (*IntegerEqualsNode x y*) (*ConstantNode* (*IntVal 1 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-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* (*IntVal 1 0*)), *constantAsStamp* (*IntVal 1 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* (*IntVal 1 0*)), *constantAsStamp* (*IntVal 1 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:
$[\![kind\ g\ nid = IntegerEqualsNode\ x\ right;$
$kind\ g\ right = AddNode\ x\ y;$
$const\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ const\text{-}id\ ((ConstantNode\ (IntVal\ 1\ 0)),\ constantAsStamp\ (IntVal\ 1\ 0))\ g;$
$g'' = replace\text{-}node\ const\text{-}id\ ((IntegerEqualsNode\ y\ const\text{-}id),\ stamp\ g\ nid)\ g']\!]$
$\Longrightarrow CanonicalizeIntegerEqualsGraph\ nid\ g\ g''$ |

int-equals-right-contains-left2:
$[\![kind\ g\ nid = IntegerEqualsNode\ y\ right;$
$kind\ g\ right = AddNode\ x\ y;$
$const\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ const\text{-}id\ ((ConstantNode\ (IntVal\ 1\ 0)),\ constantAsStamp\ (IntVal\ 1\ 0))\ g;$
$g'' = replace\text{-}node\ const\text{-}id\ ((IntegerEqualsNode\ x\ const\text{-}id),\ stamp\ g\ nid)\ g']\!]$
$\Longrightarrow CanonicalizeIntegerEqualsGraph\ nid\ g\ g''$ |

int-equals-left-contains-right3:
$[\![kind\ g\ nid = IntegerEqualsNode\ left\ x;$
$kind\ g\ left = SubNode\ x\ y;$
$const\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ const\text{-}id\ ((ConstantNode\ (IntVal\ 1\ 0)),\ constantAsStamp\ (IntVal\ 1\ 0))\ g;$
$g'' = replace\text{-}node\ const\text{-}id\ ((IntegerEqualsNode\ y\ const\text{-}id),\ stamp\ g\ nid)\ g']\!]$
$\Longrightarrow CanonicalizeIntegerEqualsGraph\ nid\ g\ g''$ |

int-equals-right-contains-left3:
$[\![kind\ g\ nid = IntegerEqualsNode\ x\ right;$
$kind\ g\ right = SubNode\ x\ y;$
$const\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ const\text{-}id\ ((ConstantNode\ (IntVal\ 1\ 0)),\ constantAsStamp\ (IntVal\ 1\ 0))\ g;$
$g'' = replace\text{-}node\ const\text{-}id\ ((IntegerEqualsNode\ y\ const\text{-}id),\ stamp\ g\ nid)\ g']\!]$
$\Longrightarrow CanonicalizeIntegerEqualsGraph\ nid\ g\ g''$

inductive *CanonicalizationStep* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
ConditionalNode:
⟦*CanonicalizeConditional g node node'*⟧
⟹ *CanonicalizationStep g node node'* |

AddNode:
⟦*CanonicalizeAdd g node node'*⟧
⟹ *CanonicalizationStep g node node'* |

IfNode:
⟦*CanonicalizeIf g node node'*⟧
⟹ *CanonicalizationStep g node node'* |

SubNode:
⟦*CanonicalizeSub g node node'*⟧
⟹ *CanonicalizationStep g node node'* |

MulNode:
⟦*CanonicalizeMul g node node'*⟧
⟹ *CanonicalizationStep g node node'* |

AndNode:
⟦*CanonicalizeAnd g node node'*⟧
⟹ *CanonicalizationStep g node node'* |

OrNode:
⟦*CanonicalizeOr g node node'*⟧
⟹ *CanonicalizationStep g node node'* |

AbsNode:
⟦*CanonicalizeAbs g node node'*⟧
⟹ *CanonicalizationStep g node node'* |

NotNode:
⟦*CanonicalizeNot g node node'*⟧

$\Longrightarrow$ *CanonicalizationStep g node node'* |

Negatenode:
$[\![$*CanonicalizeNegate 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*) *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''*$]\!]$
$\Longrightarrow$ *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'*$]\!]$
$\Longrightarrow$ *CanonicalizationPhase g* (*nid*, *seen*, *i*) *g'* |

$[\![$*Step analyse g* (*nid*, *seen*, *i*) *None*;
Some nid' = *pred g nid*;
seen' = {*nid*} ∪ *seen*;

$CanonicalizationPhase\ g\ (nid', seen', i)\ g']\!]$
$\Longrightarrow CanonicalizationPhase\ g\ (nid, seen, i)\ g'\ |$

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

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

type-synonym $Trace = IRNode\ list$
inductive $CanonicalizationPhaseWithTrace$
$:: IRGraph \Rightarrow (ID \times Seen \times CanonicalizationAnalysis) \Rightarrow IRGraph \Rightarrow Trace \Rightarrow Trace \Rightarrow bool$ **where**

— Can do a step and optimise for the current node
$[\![Step\ analyse\ g\ (nid, seen, i)\ (Some\ (nid', seen', i'));$
$CanonicalizationStep\ g\ (kind\ g\ nid)\ node;$

$g' = replace\text{-}node\ nid\ (node, stamp\ g\ nid)\ g;$

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

— 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 $(modes: i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow bool)\ CanonicalizationPhaseWithTrace\ \langle proof\rangle$

end

9 Conditional Elimination Phase

theory *ConditionalElimination*
imports
Proofs.IRGraphFrames
Proofs.Stuttering
Proofs.Form
Proofs.Rewrites
Proofs.Bisimulation
begin

9.1 Individual Elimination Rules

We introduce a TriState as in the Graal compiler to represent when static analysis can tell us information about the value of a boolean expression. Unknown = No information can be inferred KnownTrue/KnownFalse = We can infer the expression will always be true or false.

datatype *TriState* = *Unknown* | *KnownTrue* | *KnownFalse*

The implies relation corresponds to the LogicNode.implies method from the compiler which attempts to infer when one logic nodes value can be inferred from a known logic node.

inductive *implies* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *TriState* ⇒ *bool*
(- ⊢ - & - ↪ -) **for** *g* **where**
eq-imp-less:
$g \vdash$ (*IntegerEqualsNode x y*) & (*IntegerLessThanNode x y*) ↪ *KnownFalse* |
eq-imp-less-rev:
$g \vdash$ (*IntegerEqualsNode x y*) & (*IntegerLessThanNode y x*) ↪ *KnownFalse* |
less-imp-rev-less:
$g \vdash$ (*IntegerLessThanNode x y*) & (*IntegerLessThanNode y x*) ↪ *KnownFalse* |
less-imp-not-eq:
$g \vdash$ (*IntegerLessThanNode x y*) & (*IntegerEqualsNode x y*) ↪ *KnownFalse* |
less-imp-not-eq-rev:
$g \vdash$ (*IntegerLessThanNode x y*) & (*IntegerEqualsNode y x*) ↪ *KnownFalse* |

x-imp-x:
$g \vdash x$ & x ↪ *KnownTrue* |

negate-false:
⟦$g \vdash x$ & (*kind g y*) ↪ *KnownTrue*⟧ ⟹ $g \vdash x$ & (*LogicNegationNode y*) ↪ *KnownFalse* |
negate-true:
⟦$g \vdash x$ & (*kind g y*) ↪ *KnownFalse*⟧ ⟹ $g \vdash x$ & (*LogicNegationNode y*) ↪ *KnownTrue*

Total relation over partial implies relation

inductive *condition-implies* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *TriState* ⇒ *bool*
(- ⊢ - & - ⇀ -) **for** *g* **where**

$[\![\neg(g \vdash a \ \& \ b \hookrightarrow imp)]\!] \Longrightarrow (g \vdash a \ \& \ b \rightharpoonup Unknown) \mid$
$[\![(g \vdash a \ \& \ b \hookrightarrow imp)]\!] \Longrightarrow (g \vdash a \ \& \ b \rightharpoonup imp)$

Proofs that the implies relation is correct with respect to the existing evaluation semantics.

lemma *logic-negation-relation*:
assumes *wf-values g*
assumes $g\ m \vdash kind\ g\ y \mapsto val$
assumes $kind\ g\ neg = LogicNegationNode\ y$
assumes $g\ m \vdash kind\ g\ neg \mapsto invval$
shows $val\text{-}to\text{-}bool\ val \longleftrightarrow \neg(val\text{-}to\text{-}bool\ inval)$
⟨*proof*⟩

lemma *implies-valid*:
assumes $wf\text{-}graph\ g \wedge wf\text{-}values\ g$
assumes $g \vdash x \ \& \ y \rightharpoonup imp$
assumes $g\ m \vdash x \mapsto v1$
assumes $g\ m \vdash y \mapsto v2$
shows $(imp = KnownTrue \longrightarrow (val\text{-}to\text{-}bool\ v1 \longrightarrow val\text{-}to\text{-}bool\ v2)) \wedge$
$(imp = KnownFalse \longrightarrow (val\text{-}to\text{-}bool\ v1 \longrightarrow \neg(val\text{-}to\text{-}bool\ v2)))$
(**is** $(?TP \longrightarrow ?TC) \wedge (?FP \longrightarrow ?FC)$)
⟨*proof*⟩

lemma *implies-true-valid*:
assumes $wf\text{-}graph\ g \wedge wf\text{-}values\ g$
assumes $g \vdash x \ \& \ y \rightharpoonup imp$
assumes $imp = KnownTrue$
assumes $g\ m \vdash x \mapsto v1$
assumes $g\ m \vdash y \mapsto v2$
shows $val\text{-}to\text{-}bool\ v1 \longrightarrow val\text{-}to\text{-}bool\ v2$
⟨*proof*⟩

lemma *implies-false-valid*:
assumes $wf\text{-}graph\ g \wedge wf\text{-}values\ g$
assumes $g \vdash x \ \& \ y \rightharpoonup imp$
assumes $imp = KnownFalse$
assumes $g\ m \vdash x \mapsto v1$
assumes $g\ m \vdash y \mapsto v2$
shows $val\text{-}to\text{-}bool\ v1 \longrightarrow \neg(val\text{-}to\text{-}bool\ v2)$
⟨*proof*⟩

The following relation corresponds to the UnaryOpLogicNode.tryFold and BinaryOpLogicNode.tryFold methods and their associated concrete implementations.

The relation determines if a logic operation can be shown true or false through the stamp typing information.

inductive $tryFold :: IRNode \Rightarrow (ID \Rightarrow Stamp) \Rightarrow TriState \Rightarrow bool$
where

$[\![alwaysDistinct\ (stamps\ x)\ (stamps\ y)]\!]$
$\Longrightarrow tryFold\ (IntegerEqualsNode\ x\ y)\ stamps\ KnownFalse\ |$
$[\![neverDistinct\ (stamps\ x)\ (stamps\ y)]\!]$
$\Longrightarrow tryFold\ (IntegerEqualsNode\ x\ y)\ stamps\ KnownTrue\ |$
$[\![is\text{-}IntegerStamp\ (stamps\ x);$
$is\text{-}IntegerStamp\ (stamps\ y);$
$stpi\text{-}upper\ (stamps\ x) < stpi\text{-}lower\ (stamps\ y)]\!]$
$\Longrightarrow tryFold\ (IntegerLessThanNode\ x\ y)\ stamps\ KnownTrue\ |$
$[\![is\text{-}IntegerStamp\ (stamps\ x);$
$is\text{-}IntegerStamp\ (stamps\ y);$
$stpi\text{-}lower\ (stamps\ x) \geq stpi\text{-}upper\ (stamps\ y)]\!]$
$\Longrightarrow tryFold\ (IntegerLessThanNode\ x\ y)\ stamps\ KnownFalse$

Proofs that show that when the stamp lookup function is well-formed, the tryFold relation correctly predicts the output value with respect to our evaluation semantics.

lemma *tryFoldIntegerEqualsAlwaysDistinct*:
assumes *wf-stamp g stamps*
assumes $kind\ g\ nid = (IntegerEqualsNode\ x\ y)$
assumes $g\ m \vdash (kind\ g\ nid) \mapsto v$
assumes *alwaysDistinct* (*stamps x*) (*stamps y*)
shows $v = IntVal\ 1\ 0$
⟨*proof*⟩

lemma *tryFoldIntegerEqualsNeverDistinct*:
assumes *wf-stamp g stamps*
assumes $kind\ g\ nid = (IntegerEqualsNode\ x\ y)$
assumes $g\ m \vdash (kind\ g\ nid) \mapsto v$
assumes *neverDistinct* (*stamps x*) (*stamps y*)
shows $v = IntVal\ 1\ 1$
⟨*proof*⟩

lemma *tryFoldIntegerLessThanTrue*:
assumes *wf-stamp g stamps*
assumes $kind\ g\ nid = (IntegerLessThanNode\ x\ y)$
assumes $g\ m \vdash (kind\ g\ nid) \mapsto v$
assumes $stpi\text{-}upper\ (stamps\ x) < stpi\text{-}lower\ (stamps\ y)$
shows $v = IntVal\ 1\ 1$
⟨*proof*⟩

lemma *tryFoldIntegerLessThanFalse*:
assumes *wf-stamp g stamps*
assumes $kind\ g\ nid = (IntegerLessThanNode\ x\ y)$
assumes $g\ m \vdash (kind\ g\ nid) \mapsto v$
assumes $stpi\text{-}lower\ (stamps\ x) \geq stpi\text{-}upper\ (stamps\ y)$
shows $v = IntVal\ 1\ 0$
⟨*proof*⟩

theorem *tryFoldProofTrue*:

assumes $\textit{wf-stamp } g \textit{ stamps}$
assumes $\textit{tryFold } (\textit{kind } g \textit{ nid}) \textit{ stamps tristate}$
assumes $\textit{tristate} = \textit{KnownTrue}$
assumes $g\ m \vdash \textit{kind } g \textit{ nid} \mapsto v$
shows $\textit{val-to-bool } v$
$\langle \textit{proof} \rangle$

theorem *tryFoldProofFalse*:
assumes $\textit{wf-stamp } g \textit{ stamps}$
assumes $\textit{tryFold } (\textit{kind } g \textit{ nid}) \textit{ stamps tristate}$
assumes $\textit{tristate} = \textit{KnownFalse}$
assumes $g\ m \vdash (\textit{kind } g \textit{ nid}) \mapsto v$
shows $\neg(\textit{val-to-bool } v)$
$\langle \textit{proof} \rangle$

inductive-cases *StepE*:
$g \vdash (\textit{nid},m,h) \rightarrow (\textit{nid}',m',h)$

Perform conditional elimination rewrites on the graph for a particular node.

In order to determine conditional eliminations appropriately the rule needs two data structures produced by static analysis. The first parameter is the set of IRNodes that we know result in a true value when evaluated. The second parameter is a mapping from node identifiers to the flow-sensitive stamp.

The relation transforms the third parameter to the fifth parameter for a node identifier which represents the fourth parameter.

inductive *ConditionalEliminationStep* ::
$\textit{IRNode set} \Rightarrow (\textit{ID} \Rightarrow \textit{Stamp}) \Rightarrow \textit{IRGraph} \Rightarrow \textit{ID} \Rightarrow \textit{IRGraph} \Rightarrow \textit{bool}$ **where**
impliesTrue:
$[\![\textit{kind } g \textit{ ifcond} = (\textit{IfNode cid } t\ f);$
$\textit{cond} = \textit{kind } g \textit{ cid};$
$\exists\ c \in \textit{conds} .\ (g \vdash c\ \&\ \textit{cond} \hookrightarrow \textit{KnownTrue});$
$g' = \textit{constantCondition True ifcond } (\textit{kind } g \textit{ ifcond})\ g$
$]\!] \Longrightarrow \textit{ConditionalEliminationStep conds stamps } g \textit{ ifcond } g'\ |$

impliesFalse:
$[\![\textit{kind } g \textit{ ifcond} = (\textit{IfNode cid } t\ f);$
$\textit{cond} = \textit{kind } g \textit{ cid};$
$\exists\ c \in \textit{conds} .\ (g \vdash c\ \&\ \textit{cond} \hookrightarrow \textit{KnownFalse});$
$g' = \textit{constantCondition False ifcond } (\textit{kind } g \textit{ ifcond})\ g$
$]\!] \Longrightarrow \textit{ConditionalEliminationStep conds stamps } g \textit{ ifcond } g'\ |$

tryFoldTrue:
$[\![\textit{kind } g \textit{ ifcond} = (\textit{IfNode cid } t\ f);$
$\textit{cond} = \textit{kind } g \textit{ cid};$
$\textit{tryFold } (\textit{kind } g \textit{ cid}) \textit{ stamps KnownTrue};$

$g' = constantCondition\ True\ ifcond\ (kind\ g\ ifcond)\ g$
$\rrbracket \Longrightarrow ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'$ |

tryFoldFalse:
$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
$cond = kind\ g\ cid;$
$tryFold\ (kind\ g\ cid)\ stamps\ KnownFalse;$
$g' = constantCondition\ False\ ifcond\ (kind\ g\ ifcond)\ g$
$\rrbracket \Longrightarrow ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'$

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

thm *ConditionalEliminationStep.equation*

9.2 Control-flow Graph Traversal

type-synonym *Seen* = *ID set*
type-synonym *Conditions* = *IRNode list*
type-synonym *StampFlow* = $(ID \Rightarrow Stamp)$ *list*

nextEdge helps determine which node to traverse next by returning the first successor edge that isn't in the set of already visited nodes. If there is not an appropriate successor, None is returned instead.

fun *nextEdge* :: $Seen \Rightarrow ID \Rightarrow IRGraph \Rightarrow ID\ option$ **where**
nextEdge seen nid g =
$(let\ nids = (filter\ (\lambda nid'.\ nid' \notin seen)\ (successors\text{-}of\ (kind\ g\ nid)))\ in$
$(if\ length\ nids > 0\ then\ Some\ (hd\ nids)\ else\ None))$

pred determines which node, if any, acts as the predecessor of another.

Merge nodes represent a special case where-in the predecessor exists as an input edge of the merge node, to simplify the traversal we treat only the first input end node as the predecessor, ignoring that multiple nodes may act as a successor.

For all other nodes, the predecessor is the first element of the predecessors set. Note that in a well-formed graph there should only be one element in the predecessor set.

fun *pred* :: $IRGraph \Rightarrow ID \Rightarrow ID\ option$ **where**
$pred\ g\ nid = (case\ kind\ g\ nid\ of$
$(MergeNode\ ends\ \text{-}\ \text{-}) \Rightarrow Some\ (hd\ ends)$ |
$\text{-} \Rightarrow$
$(if\ IRGraph.predecessors\ g\ nid = \{\}$
$then\ None\ else$
$Some\ (hd\ (sorted\text{-}list\text{-}of\text{-}set\ (IRGraph.predecessors\ g\ nid)))$
)
)

When the basic block of an if statement is entered, we know that the condition of the preceding if statement must be true. As in the GraalVM compiler, we introduce the registerNewCondition funciton which roughly corresponds to the ConditionalEliminationPhase.registerNewCondition. This method updates the flow-sensitive stamp information based on the condition which we know must be true.

fun *clip-upper* :: *Stamp* ⇒ *int* ⇒ *Stamp* **where**
clip-upper (*IntegerStamp b l h*) *c* = (*IntegerStamp b l c*) |
clip-upper s c = *s*
fun *clip-lower* :: *Stamp* ⇒ *int* ⇒ *Stamp* **where**
clip-lower (*IntegerStamp b l h*) *c* = (*IntegerStamp b c h*) |
clip-lower s c = *s*

fun *registerNewCondition* :: *IRGraph* ⇒ *IRNode* ⇒ (*ID* ⇒ *Stamp*) ⇒ (*ID* ⇒ *Stamp*) **where**

registerNewCondition g (*IntegerEqualsNode x y*) *stamps* =
(*stamps*(*x* := *join* (*stamps x*) (*stamps y*)))(*y* := *join* (*stamps x*) (*stamps y*)) |

registerNewCondition g (*IntegerLessThanNode x y*) *stamps* =
(*stamps*
(*x* := *clip-upper* (*stamps x*) (*stpi-lower* (*stamps y*))))
(*y* := *clip-lower* (*stamps y*) (*stpi-upper* (*stamps x*))) |
registerNewCondition g - stamps = *stamps*

fun *hdOr* :: *'a list* ⇒ *'a* ⇒ *'a* **where**
hdOr (*x* # *xs*) *de* = *x* |
hdOr [] *de* = *de*

The Step relation is a small-step traversal of the graph which handles transitions between individual nodes of the graph.

It relates a pairs of tuple of the current node, the set of seen nodes, the always true stack of IfNode conditions, and the flow-sensitive stamp information.

inductive *Step*
:: *IRGraph* ⇒ (*ID* × *Seen* × *Conditions* × *StampFlow*) ⇒ (*ID* × *Seen* × *Conditions* × *StampFlow*) *option* ⇒ *bool*
for *g* **where**
— Hit a BeginNode with an IfNode predecessor which represents the start of a basic block for the IfNode. 1. nid' will be the successor of the begin node. 2. Find the first and only predecessor. 3. Extract condition from the preceding IfNode. 4. Negate condition if the begin node is second branch (we've taken the else branch of the condition) 5. Add the condition or the negated condition to stack 6. Perform any stamp updates based on the condition using the registerNewCondition function and place them on the top of the stack of stamp information
⟦*kind g nid* = *BeginNode nid'*;

nid ∉ *seen*;
seen' = {*nid*} ∪ *seen*;

$Some\ ifcond = pred\ g\ nid;$
$kind\ g\ ifcond = IfNode\ cond\ t\ f;$

$i = find\text{-}index\ nid\ (successors\text{-}of\ (kind\ g\ ifcond));$
$c = (if\ i = 0\ then\ kind\ g\ cond\ else\ NegateNode\ cond);$
$conds' = c\ \#\ conds;$

$flow' = registerNewCondition\ g\ (kind\ g\ cond)\ (hdOr\ flow\ (stamp\ g))]\!]$
$\Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds',\ flow'\ \#\ flow))\ |$

— Hit an EndNode 1. nid' will be the usage of EndNode 2. pop the conditions and stamp stack
$[\![kind\ g\ nid = EndNode;$

$nid \notin seen;$
$seen' = \{nid\} \cup seen;$

$nid' = any\text{-}usage\ g\ nid;$

$conds' = tl\ conds;$
$flow' = tl\ flow]\!]$
$\Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds',\ flow'))\ |$

— We can find a successor edge that is not in seen, go there
$[\![\neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\neg(is\text{-}BeginNode\ (kind\ g\ nid));$

$nid \notin seen;$
$seen' = \{nid\} \cup seen;$

$Some\ nid' = nextEdge\ seen'\ nid\ g]\!]$
$\Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds,\ flow))\ |$

— We can cannot find a successor edge that is not in seen, give back None
$[\![\neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\neg(is\text{-}BeginNode\ (kind\ g\ nid));$

$nid \notin seen;$
$seen' = \{nid\} \cup seen;$

$None = nextEdge\ seen'\ nid\ g]\!]$
$\Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ None\ |$

— We've already seen this node, give back None
$[\![nid \in seen]\!] \Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ None$

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

The ConditionalEliminationPhase relation is responsible for combining the individual traversal steps from the Step relation and the optimizations from the ConditionalEliminationStep relation to perform a transformation of the whole graph.

inductive *ConditionalEliminationPhase*
:: $IRGraph \Rightarrow (ID \times Seen \times Conditions \times StampFlow) \Rightarrow IRGraph \Rightarrow bool$
where

— Can do a step and optimise for the current node
$[\![Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds',\ flow'));$
$ConditionalEliminationStep\ (set\ conds)\ (hdOr\ flow\ (stamp\ g))\ g\ nid\ g';$

$ConditionalEliminationPhase\ g'\ (nid',\ seen',\ conds',\ flow')\ g'']\!]$
$\Longrightarrow ConditionalEliminationPhase\ g\ (nid,\ seen,\ conds,\ flow)\ g''\ |$

— Can do a step, matches whether optimised or not causing non-determinism We need to find a way to negate ConditionalEliminationStep
$[\![Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds',\ flow'));$

$ConditionalEliminationPhase\ g\ (nid',\ seen',\ conds',\ flow')\ g']\!]$
$\Longrightarrow ConditionalEliminationPhase\ g\ (nid,\ seen,\ conds,\ flow)\ g'\ |$

— Can't do a step but there is a predecessor we can backtrace to
$[\![Step\ g\ (nid,\ seen,\ conds,\ flow)\ None;$
$Some\ nid' = pred\ g\ nid;$
$seen' = \{nid\} \cup seen;$
$ConditionalEliminationPhase\ g\ (nid',\ seen',\ conds,\ flow)\ g']\!]$
$\Longrightarrow ConditionalEliminationPhase\ g\ (nid,\ seen,\ conds,\ flow)\ g'\ |$

— Can't do a step and have no predecessors so terminate
$[\![Step\ g\ (nid,\ seen,\ conds,\ flow)\ None;$
$None = pred\ g\ nid]\!]$
$\Longrightarrow ConditionalEliminationPhase\ g\ (nid,\ seen,\ conds,\ flow)\ g$

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

code-pred $(modes: i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool)$ *ConditionalElimination-PhaseWithTrace* $\langle proof \rangle$

lemma *IfNodeStepE*: $g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h) \Longrightarrow$
$(\bigwedge cond\ tb\ fb\ val.$
$kind\ g\ nid = IfNode\ cond\ tb\ fb \Longrightarrow$
$nid' = (if\ val\text{-}to\text{-}bool\ val\ then\ tb\ else\ fb) \Longrightarrow$
$g\ m \vdash kind\ g\ cond \mapsto val \Longrightarrow m' = m)$
$\langle proof \rangle$

lemma *ifNodeHasCondEvalStutter*:

assumes $(g\ m\ h \vdash nid \leadsto nid')$
assumes $kind\ g\ nid = IfNode\ cond\ t\ f$
shows $\exists\ v.\ (g\ m \vdash kind\ g\ cond \mapsto v)$
$\langle proof \rangle$

lemma *ifNodeHasCondEval*:
assumes $(g \vdash (nid, m, h) \rightarrow (nid', m', h'))$
assumes $kind\ g\ nid = IfNode\ cond\ t\ f$
shows $\exists\ v.\ (g\ m \vdash kind\ g\ cond \mapsto v)$
$\langle proof \rangle$

lemma *replace-if-t*:
assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
assumes $g\ m \vdash kind\ g\ cond \mapsto bool$
assumes $val\text{-}to\text{-}bool\ bool$
assumes g': $g' = replace\text{-}usages\ nid\ tb\ g$
shows $\exists nid'.(g\ m\ h \vdash nid \leadsto nid') \longleftrightarrow (g'\ m\ h \vdash nid \leadsto nid')$
$\langle proof \rangle$

lemma *replace-if-t-imp*:
assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
assumes $g\ m \vdash kind\ g\ cond \mapsto bool$
assumes $val\text{-}to\text{-}bool\ bool$
assumes g': $g' = replace\text{-}usages\ nid\ tb\ g$
shows $\exists nid'.(g\ m\ h \vdash nid \leadsto nid') \longrightarrow (g'\ m\ h \vdash nid \leadsto nid')$
$\langle proof \rangle$

lemma *replace-if-f*:
assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
assumes $g\ m \vdash kind\ g\ cond \mapsto bool$
assumes $\neg(val\text{-}to\text{-}bool\ bool)$
assumes g': $g' = replace\text{-}usages\ nid\ fb\ g$
shows $\exists nid'.(g\ m\ h \vdash nid \leadsto nid') \longleftrightarrow (g'\ m\ h \vdash nid \leadsto nid')$
$\langle proof \rangle$

Prove that the individual conditional elimination rules are correct with respect to preservation of stuttering steps.

lemma *ConditionalEliminationStepProof*:
assumes *wg*: *wf-graph g*
assumes *ws*: *wf-stamps g*
assumes *wv*: *wf-values g*
assumes *nid*: $nid \in ids\ g$
assumes *cond-valid*: $\forall\ c \in conds\ .\ \exists\ v.\ (g\ m \vdash c \mapsto v) \land val\text{-}to\text{-}bool\ v$
assumes *ce*: $ConditionalEliminationStep\ conds\ stamps\ g\ nid\ g'$

shows $\exists nid'.(g\ m\ h \vdash nid \leadsto nid') \longrightarrow (g'\ m\ h \vdash nid \leadsto nid')$
$\langle proof \rangle$

Prove that the individual conditional elimination rules are correct with respect to finding a bisimulation between the unoptimized and optimized graphs.

lemma *ConditionalEliminationStepProofBisimulation*:
assumes *wf*: $wf\text{-}graph\ g \wedge wf\text{-}stamp\ g\ stamps \wedge wf\text{-}values\ g$
assumes *nid*: $nid \in ids\ g$
assumes *conds-valid*: $\forall\ c \in conds\ .\ \exists\ v.\ (g\ m \vdash c \mapsto v) \wedge val\text{-}to\text{-}bool\ v$
assumes *ce*: $ConditionalEliminationStep\ conds\ stamps\ g\ nid\ g'$
assumes *gstep*: $\exists\ h\ nid'.\ (g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h))$

shows $nid \mid g \sim g'$
⟨*proof*⟩

Mostly experimental proofs from here on out.

lemma *if-step*:
assumes $nid \in ids\ g$
assumes $(kind\ g\ nid) \in control\text{-}nodes$
shows $(g\ m\ h \vdash nid \rightsquigarrow nid')$
⟨*proof*⟩

lemma *StepConditionsValid*:
assumes $\forall\ cond \in set\ conds.\ (g\ m \vdash cond \mapsto v) \wedge val\text{-}to\text{-}bool\ v$
assumes $Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds',\ flow'))$
shows $\forall\ cond \in set\ conds'.\ (g\ m \vdash cond \mapsto v) \wedge val\text{-}to\text{-}bool\ v$
⟨*proof*⟩

lemma *ConditionalEliminationPhaseProof*:
assumes $wf\text{-}graph\ g$
assumes $wf\text{-}stamps\ g$
assumes $ConditionalEliminationPhase\ g\ (0,\ \{\},\ [],\ [])\ g'$

shows $\exists nid'\ .(g\ m\ h \vdash 0 \rightsquigarrow nid') \longrightarrow (g'\ m\ h \vdash 0 \rightsquigarrow nid')$
⟨*proof*⟩

end

10 Graph Construction Phase

theory
Construction
imports
Proofs.Bisimulation
Proofs.IRGraphFrames
begin

lemma *add-const-nodes*:
assumes *xn*: $kind\ g\ x = (ConstantNode\ (IntVal\ b\ xv))$
assumes *yn*: $kind\ g\ y = (ConstantNode\ (IntVal\ b\ yv))$

assumes *zn*: *kind g z* = (*AddNode x y*)
assumes *wn*: *kind g w* = (*ConstantNode* (*intval-add* (*IntVal b xv*) (*IntVal b yv*)))
assumes *val*: *intval-add* (*IntVal b xv*) (*IntVal b yv*) = *IntVal b v1*
assumes *ez*: *g m* ⊢ (*kind g z*) ↦ (*IntVal b v1*)
assumes *ew*: *g m* ⊢ (*kind g w*) ↦ (*IntVal b v2*)
shows *v1* = *v2*
⟨*proof*⟩

lemma *add-val-xzero*:
shows *intval-add* (*IntVal b 0*) (*IntVal b yv*) = (*IntVal b yv*)
⟨*proof*⟩

lemma *add-val-yzero*:
shows *intval-add* (*IntVal b xv*) (*IntVal b 0*) = (*IntVal b xv*)
⟨*proof*⟩

fun *create-add* :: *IRGraph* ⇒ *ID* ⇒ *ID* ⇒ *IRNode* **where**
create-add g x y =
(*case* (*kind g x*) *of*
ConstantNode (*IntVal b xv*) ⇒
(*case* (*kind g y*) *of*
ConstantNode (*IntVal b yv*) ⇒
ConstantNode (*intval-add* (*IntVal b xv*) (*IntVal b yv*)) |
- ⇒ *if xv* = *0 then RefNode y else AddNode x y*
) |
- ⇒ (*case* (*kind g y*) *of*
ConstantNode (*IntVal b yv*) ⇒
if yv = *0 then RefNode x else AddNode x y* |
- ⇒ *AddNode x y*
)
)

lemma *add-node-create*:
assumes *xv*: *g m* ⊢ (*kind g x*) ↦ *IntVal b xv*
assumes *yv*: *g m* ⊢ (*kind g y*) ↦ *IntVal b yv*
assumes *res*: *res* = *intval-add* (*IntVal b xv*) (*IntVal b yv*)
shows
(*g m* ⊢ (*AddNode x y*) ↦ *res*) ∧
(*g m* ⊢ (*create-add g x y*) ↦ *res*)

⟨*proof*⟩

fun *add-node-fake* :: *ID* ⇒ *IRNode* ⇒ *IRGraph* ⇒ *IRGraph* **where**
add-node-fake nid k g = *add-node nid* (*k*, *VoidStamp*) *g*

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

lemma *dom-add-unchanged*:
assumes $nid \in ids\ g$
assumes $g' = add\text{-}node\text{-}fake\ n\ k\ g$
assumes $nid \neq n$
shows $nid \in ids\ g'$
⟨*proof*⟩

lemma *preserve-wf*:
assumes *wf*: $wf\text{-}graph\ g$
assumes $nid \notin ids\ g$
assumes *closed*: $inputs\ g'\ nid \cup succ\ g'\ nid \subseteq ids\ g$
assumes g': $g' = add\text{-}node\text{-}fake\ nid\ k\ g$
shows $wf\text{-}graph\ g'$
⟨*proof*⟩

lemma *equal-closure-bisimilar*:
assumes $\{P'.\ (g\ m\ h \vdash nid \rightsquigarrow P')\} = \{P'.\ (g'\ m\ h \vdash nid \rightsquigarrow P')\}$
shows $nid\ .\ g \sim g'$
⟨*proof*⟩

lemma *wf-size*:
assumes $nid \in ids\ g$
assumes $wf\text{-}graph\ g$
assumes $is\text{-}AbstractEndNode\ (kind\ g\ nid)$
shows $card\ (usages\ g\ nid) > 0$
⟨*proof*⟩

lemma *sequentials-have-successors*:
assumes $is\text{-}sequential\text{-}node\ n$
shows $size\ (successors\text{-}of\ n) > 0$
⟨*proof*⟩

lemma *step-reaches-successors-only*:
assumes $(g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h))$
assumes *wf*: $wf\text{-}graph\ g$
shows $nid' \in succ\ g\ nid \lor nid' \in usages\ g\ nid$

⟨*proof*⟩

lemma *stutter-closed*:
assumes $g\ m\ h \vdash nid \leadsto nid'$
assumes *wf-graph g*
shows $\exists\ n \in ids\ g\ .\ nid' \in succ\ g\ n \vee nid' \in usages\ g\ n$
⟨*proof*⟩

lemma *unchanged-step*:
assumes $g \vdash (nid, m, h) \to (nid', m, h)$
assumes *wf*: *wf-graph g*
assumes *kind*: $kind\ g\ nid = kind\ g'\ nid$
assumes *unchanged*: $unchanged\ (eval\text{-}usages\ g\ nid)\ g\ g'$
assumes *succ*: $succ\ g\ nid = succ\ g'\ nid$

shows $g' \vdash (nid, m, h) \to (nid', m, h)$
⟨*proof*⟩

lemma *unchanged-closure*:
assumes $nid \notin ids\ g$
assumes *wf*: $wf\text{-}graph\ g \wedge wf\text{-}graph\ g'$
assumes g': $g' = add\text{-}node\text{-}fake\ nid\ k\ g$
assumes $nid' \in ids\ g$
shows $(g\ m\ h \vdash nid' \leadsto nid'') \longleftrightarrow (g'\ m\ h \vdash nid' \leadsto nid'')$
(**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

fun *create-if* :: $IRGraph \Rightarrow ID \Rightarrow ID \Rightarrow ID \Rightarrow IRNode$
where
create-if g cond tb fb =
(*case* (*kind g cond*) *of*
ConstantNode condv ⇒
RefNode (*if* (*val-to-bool condv*) *then tb else fb*) |
\- ⇒ (*if tb* = *fb then*
RefNode tb
else
IfNode cond tb fb)
)

lemma *if-node-create-bisimulation*:
fixes *h* :: *FieldRefHeap*
assumes *wf*: *wf-graph g*
assumes *cv*: $g\ m \vdash (kind\ g\ cond) \mapsto cv$
assumes *fresh*: $nid \notin ids\ g$
assumes *closed*: $\{cond, tb, fb\} \subseteq ids\ g$
assumes *gif*: $gif = add\text{-}node\text{-}fake\ nid\ (IfNode\ cond\ tb\ fb)\ g$
assumes *gcreate*: $gcreate = add\text{-}node\text{-}fake\ nid\ (create\text{-}if\ g\ cond\ tb\ fb)\ g$

shows nid . $gif \sim gcreate$

$\langle proof \rangle$

lemma *if-node-create*:
assumes *wf*: *wf-graph g*
assumes *cv*: $g\ m \vdash (kind\ g\ cond) \mapsto cv$
assumes *fresh*: $nid \notin ids\ g$
assumes *gif*: $gif = add\text{-}node\text{-}fake\ nid\ (IfNode\ cond\ tb\ fb)\ g$
assumes *gcreate*: $gcreate = add\text{-}node\text{-}fake\ nid\ (create\text{-}if\ g\ cond\ tb\ fb)\ g$
shows $\exists\, nid'.\ (gif\ m\ h \vdash nid \leadsto nid') \wedge (gcreate\ m\ h \vdash nid \leadsto nid')$

$\langle proof \rangle$

end