Veriopt

July 3, 2021

Abstract

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

Contents

1 Runtime Values and Arithmetic **3**

2 Nodes **9**
2.1 Types of Nodes 9
2.2 Hierarchy of Nodes 16

3 Stamp Typing **22**

4 Graph Representation **26**
4.0.1 Example Graphs 31

5 Data-flow Semantics **31**

6 Control-flow Semantics **37**
6.1 Heap 37
6.2 Intraprocedural Semantics 38
6.3 Interprocedural Semantics 44
6.4 Big-step Execution 45
6.4.1 Heap Testing 46

7 Proof Infrastructure **47**
7.1 Bisimulation 47
7.2 Formedness Properties 48
7.3 Dynamic Frames 49
7.4 Graph Rewriting 61
7.5 Stuttering 65

8 Canonicalization Phase **66**

1 Runtime Values and Arithmetic

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

In order to properly implement the IR semantics we first introduce a new type of runtime values. Our evaluation semantics are defined in terms of these runtime values. These runtime values represent the full range of primitive types currently allowed by our semantics, ranging from basic integer types to object references and eventually arrays.

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

type-synonym *objref* = *nat option*

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

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

So each (IntVal b v) should satisfy the invariants:

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

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

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

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

— Check that a signed int value does not overflow b bits.
fun *fits-into-n* :: $nat \Rightarrow int \Rightarrow bool$ **where**
  $fits\text{-}into\text{-}n\ b\ val = ((-(2\widehat{\ }(b-1)) \leq val) \wedge (val < (2\widehat{\ }(b-1))))$

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

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

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

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

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

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

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

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

instantiation *Value* :: *plus*
begin

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

instance proof qed
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 qed
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 qed
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 qed
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 qed
end

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

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

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

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

```
lemma intval-add-bits:
  assumes b: IntVal b res = intval-add x y
  shows b = 32 ∨ b = 64
proof –
  have def: intval-add x y ≠ UndefVal
    using b by auto
  obtain b1 v1 where x: x = IntVal b1 v1
    by (metis Value.exhaust-sel def intval-add.simps(2,3,4,5))
  obtain b2 v2 where y: y = IntVal b2 v2
    by (metis Value.exhaust-sel def intval-add.simps(6,7,8,9))
  have
     ax: 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)))))
      (is ?L = (if ?C then (IntVal 32 ?A) else (IntVal 64 ?B)))
    by simp
  then have l: IntVal b res = ?L using b x y by simp
  have (b1 ≤ 32 ∧ b2 ≤ 32) ∨ ¬(b1 ≤ 32 ∧ b2 ≤ 32) by auto
  then show ?thesis
  proof
    assume (b1 ≤ 32 ∧ b2 ≤ 32)
    then have r32: ?L = (IntVal 32 ?A) using ax by auto
    then have b = 32 using r32 l b by auto
    then show ?thesis by simp
  next
    assume ¬(b1 ≤ 32 ∧ b2 ≤ 32)
    then have r64: ?L = (IntVal 64 ?B) using ax by auto
    then have b = 64 using r64 l b by auto
    then show ?thesis by simp
  qed
qed


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


lemma intval-add-sym1:
 shows intval-add (IntVal b1 v1) (IntVal b2 v2) = intval-add (IntVal b2 v2) (IntVal
b1 v1)
  by (simp add: word-add-sym)

lemma intval-add-sym:
  shows intval-add x y = intval-add y x
  using intval-add-sym1 apply simp
  apply (induction x)
```

```
    apply auto
  apply (induction y)
    apply auto
  done

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

lemma wf-int32:
  assumes wf: wf-value (IntVal b v)
  shows b = 32
proof −
  have b ∈ int-bits-allowed
    using wf wf-value.simps(1) by blast
  then show ?thesis
    by (simp add: int-bits-allowed-def)
qed

lemma wf-int [simp]:
  assumes wf: wf-value (IntVal w n)
  assumes notbool: w = 32
  shows sint((word-of-int n) :: int32) = n
  apply (simp only: int-word-sint)
  using wf notbool apply simp
  done

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))
  apply (simp only: intval-add.simps word-of-int-0)
  apply (simp only: order-class.order.refl conj-absorb if-True)
  apply (simp only: word-add-def uint-0-eq add-0)
  apply (simp only: word-of-int-uint int-word-sint)
  using b apply simp
  done

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

lemma *intval-add* (*IntVal 32* (*2^31−1*)) (*IntVal 32* (*2^31−1*)) = *IntVal 32* (*−2*)
by *eval*
lemma *intval-add* (*IntVal 64* (*2^31−1*)) (*IntVal 32* (*2^31−1*)) = *IntVal 64 4294967294*
by *eval*

end

2 Nodes

2.1 Types of Nodes

theory *IRNodes*
imports
Values2
begin

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

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

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

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

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

datatype (*discs-sels*) *IRNode* =
AbsNode (*ir-value*: *INPUT*)
| *AddNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *AndNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *BeginNode* (*ir-next*: *SUCC*)
| *BytecodeExceptionNode* (*ir-arguments*: *INPUT list*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *ConditionalNode* (*ir-condition*: *INPUT-COND*) (*ir-trueValue*: *INPUT*) (*ir-falseValue*: *INPUT*)

| *ConstantNode* (*ir-const*: *Value*)

| *DynamicNewArrayNode* (*ir-elementType*: *INPUT*) (*ir-length*: *INPUT*) (*ir-voidClass-opt*: *INPUT option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *EndNode*

| *ExceptionObjectNode* (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *FrameState* (*ir-monitorIds*: *INPUT-ASSOC list*) (*ir-outerFrameState-opt*: *INPUT-STATE option*) (*ir-values-opt*: *INPUT list option*) (*ir-virtualObjectMappings-opt*: *INPUT-STATE list option*)

| *IfNode* (*ir-condition*: *INPUT-COND*) (*ir-trueSuccessor*: *SUCC*) (*ir-falseSuccessor*: *SUCC*)

| *IntegerEqualsNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)

| *IntegerLessThanNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)

| *InvokeNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *InvokeWithExceptionNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*) (*ir-exceptionEdge*: *SUCC*)

| *IsNullNode* (*ir-value*: *INPUT*)

| *KillingBeginNode* (*ir-next*: *SUCC*)

| *LoadFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)

| *LogicNegationNode* (*ir-value*: *INPUT-COND*)

| *LoopBeginNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-overflowGuard-opt*: *INPUT-GUARD option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *LoopEndNode* (*ir-loopBegin*: *INPUT-ASSOC*)

| *LoopExitNode* (*ir-loopBegin*: *INPUT-ASSOC*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *MergeNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *MethodCallTargetNode* (*ir-targetMethod*: *string*) (*ir-arguments*: *INPUT list*)

| *MulNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)

| *NegateNode* (*ir-value*: *INPUT*)

| *NewArrayNode* (*ir-length*: *INPUT*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *NewInstanceNode* (*ir-nid*: *ID*) (*ir-instanceClass*: *string*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *NotNode* (*ir-value*: *INPUT*)

| *OrNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)

| *ParameterNode* (*ir-index*: *nat*)

| *PiNode* (*ir-object*: *INPUT*) (*ir-guard-opt*: *INPUT-GUARD option*)

| *ReturnNode* (*ir-result-opt*: *INPUT option*) (*ir-memoryMap-opt*: *INPUT-EXT option*)

| *ShortCircuitOrNode* (*ir-x*: *INPUT-COND*) (*ir-y*: *INPUT-COND*)

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

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

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

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

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

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

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

fun *inputs-of* :: *IRNode* ⇒ *ID list* **where**
inputs-of-AbsNode:
inputs-of (*AbsNode value*) = [*value*] |
inputs-of-AddNode:
inputs-of (*AddNode x y*) = [*x*, *y*] |
inputs-of-AndNode:
inputs-of (*AndNode x y*) = [*x*, *y*] |
inputs-of-BeginNode:
inputs-of (*BeginNode next*) = [] |
inputs-of-BytecodeExceptionNode:
inputs-of (*BytecodeExceptionNode arguments stateAfter next*) = *arguments* @ (*opt-to-list stateAfter*) |
inputs-of-ConditionalNode:
inputs-of (*ConditionalNode condition trueValue falseValue*) = [*condition*, *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
unfolding *inputs-of-FrameState* **by** *simp*
lemma *successors-of* (*FrameState x* (*Some y*) (*Some z*) *None*) = []
unfolding *inputs-of-FrameState* **by** *simp*

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

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

end

2.2 Hierarchy of Nodes

theory *IRNodeHierarchy*
imports *IRNodes2*
begin

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

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

These functions have been automatically generated from the compiler.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

fun *is-FloatingNode* :: *IRNode* ⇒ *bool* **where**
is-FloatingNode n = ((*is-AbstractLocalNode n*) ∨ (*is-BinaryNode n*) ∨ (*is-ConditionalNode n*) ∨ (*is-ConstantNode n*) ∨ (*is-FloatingGuardedNode n*) ∨ (*is-LogicNode n*) ∨ (*is-PhiNode n*) ∨ (*is-ProxyNode n*) ∨ (*is-UnaryNode n*))

fun *is-CallTargetNode* :: *IRNode* ⇒ *bool* **where**
is-CallTargetNode n = ((*is-MethodCallTargetNode n*))

fun *is-ValueNode* :: *IRNode* ⇒ *bool* **where**
is-ValueNode n = ((*is-CallTargetNode n*) ∨ (*is-FixedNode n*) ∨ (*is-FloatingNode*

n))

fun *is-VirtualState* :: *IRNode* ⇒ *bool* **where**
is-VirtualState n = ((*is-FrameState* n))

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

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

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

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

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

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

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

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

fun *is-FixedNodeInterface* :: *IRNode* ⇒ *bool* **where**
is-FixedNodeInterface n = ((*is-FixedNode n*))

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

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

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

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

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

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

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

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

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

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

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

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

fun *is-same-ir-node-type* :: *IRNode* ⇒ *IRNode* ⇒ *bool* **where**
is-same-ir-node-type n1 n2 = (
((*is-AbsNode n1*) ∧ (*is-AbsNode n2*)) ∨
((*is-AddNode n1*) ∧ (*is-AddNode n2*)) ∨
((*is-AndNode n1*) ∧ (*is-AndNode n2*)) ∨
((*is-BeginNode n1*) ∧ (*is-BeginNode n2*)) ∨

((*is-BytecodeExceptionNode n1*) ∧ (*is-BytecodeExceptionNode n2*)) ∨
((*is-ConditionalNode n1*) ∧ (*is-ConditionalNode n2*)) ∨
((*is-ConstantNode n1*) ∧ (*is-ConstantNode n2*)) ∨
((*is-DynamicNewArrayNode n1*) ∧ (*is-DynamicNewArrayNode n2*)) ∨
((*is-EndNode n1*) ∧ (*is-EndNode n2*)) ∨
((*is-ExceptionObjectNode n1*) ∧ (*is-ExceptionObjectNode n2*)) ∨
((*is-FrameState n1*) ∧ (*is-FrameState n2*)) ∨
((*is-IfNode n1*) ∧ (*is-IfNode n2*)) ∨
((*is-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 *Values2*
begin

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

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

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

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

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

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

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

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

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

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

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

is-stamp-empty x = *False*

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

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

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

join (*KlassPointerStamp nn1 an1*) (*KlassPointerStamp nn2 an2*) = (
if ((*nn1* $\vee$ *nn2*) $\wedge$ (*an1* $\vee$ *an2*))
then (*empty-stamp* (*KlassPointerStamp nn1 an1*))
else (*KlassPointerStamp* (*nn1* $\vee$ *nn2*) (*an1* $\vee$ *an2*))
) |

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

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

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

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

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

constantAsStamp - = *IllegalStamp*

— Define when a runtime value is valid for a stamp
fun *valid-value* :: *Stamp* ⇒ *Value* ⇒ *bool* **where**
valid-value (*IntegerStamp b1 l h*) (*IntVal32 v*) = ((*sint v* ≥ *l*) ∧ (*sint 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*))

notepad
begin
have *unrestricted-stamp* (*IntegerStamp 8 0 10*) = (*IntegerStamp 8* (− *128*) *127*)
by *auto*
have *unrestricted-stamp* (*IntegerStamp 16 0 10*) = (*IntegerStamp 16* (− *32768*) *32767*)
by *auto*
have *unrestricted-stamp* (*IntegerStamp 32 0 10*) = (*IntegerStamp 32* (− *2147483648*) *2147483647*)
by *auto*
have *empty-stamp* (*IntegerStamp 8 0 10*) = (*IntegerStamp 8 127* (− *128*))
by *auto*
have *empty-stamp* (*IntegerStamp 16 0 10*) = (*IntegerStamp 16 32767* (− *32768*))
by *auto*
have *empty-stamp* (*IntegerStamp 32 0 10*) = (*IntegerStamp 32 2147483647* (− *2147483648*))
by *auto*
have *join* (*IntegerStamp 32 0 20*) (*IntegerStamp 32* (−*100*) *10*) = (*IntegerStamp 32 0 10*)
by *auto*
have *meet* (*IntegerStamp 32 0 20*) (*IntegerStamp 32* (−*100*) *10*) = (*IntegerStamp 32* (− *100*) *20*)
by *auto*
end

end

4 Graph Representation

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

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

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

typedef *IRGraph* = $\{g :: ID \rightharpoonup (IRNode \times Stamp) \;.\; finite\ (dom\ g)\}$
proof −
have $finite(dom(Map.empty)) \land ran\ Map.empty = \{\}$ **by** *auto*
then show *?thesis*

by *fastforce*
qed

setup-lifting *type-definition-IRGraph*

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

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

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

lift-definition *stamp* :: *IRGraph* ⇒ *ID* ⇒ *Stamp*
is *with-default IllegalStamp snd* **.**

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

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

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

lift-definition *as-list* :: *IRGraph* ⇒ (*ID* × *IRNode* × *Stamp*) *list*
is $\lambda g.$ *map* ($\lambda k.$ (*k*, *the* (*g k*))) (*sorted-list-of-set* (*dom g*)) **.**

fun *no-node* :: (*ID* × (*IRNode* × *Stamp*)) *list* ⇒ (*ID* × (*IRNode* × *Stamp*)) *list*
where
no-node g = *filter* ($\lambda n.$ *fst* (*snd n*) ≠ *NoNode*) *g*

lift-definition *irgraph* :: (*ID* × (*IRNode* × *Stamp*)) *list* ⇒ *IRGraph*
is *map-of* ∘ *no-node*
by (*simp add*: *finite-dom-map-of*)

code-datatype *irgraph*

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

lemma *no-node-clears*:
res = *no-node xs* ⟶ (∀ *x* ∈ *set res*. *fst* (*snd x*) ≠ *NoNode*)
by *simp*

lemma *dom-eq*:

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

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

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

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

— Get the inputs set of a given node ID
fun *inputs* :: *IRGraph* ⇒ *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 = $(\bigcup\ i \in ids\ g.\ \{(i,j)|j.\ j \in (inputs\ g\ i)\})$
— Find all the nodes in the graph that have nid as an input - the usages of nid
fun *usages* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
usages g nid = $\{j.\ j \in ids\ g \wedge (j,nid) \in input\text{-}edges\ g\}$
fun *successor-edges* :: *IRGraph* ⇒ *ID rel* **where**
successor-edges g = $(\bigcup\ i \in ids\ g.\ \{(i,j)|j\ .\ j \in (succ\ \ g\ i)\})$
fun *predecessors* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
predecessors g nid = $\{j.\ j \in ids\ g \wedge (j,nid) \in successor\text{-}edges\ g\}$
fun *nodes-of* :: *IRGraph* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID set* **where**
nodes-of g sel = $\{nid \in ids\ g\ .\ sel\ (kind\ g\ nid)\}$
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* ($f \circ$ (*kind g*)) (*inputs-of* (*kind g nid*))
fun *filtered-successors* :: *IRGraph* ⇒ *ID* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID list* **where**
filtered-successors g nid f = *filter* ($f \circ$ (*kind g*)) (*successors-of* (*kind g nid*))
fun *filtered-usages* :: *IRGraph* ⇒ *ID* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID set* **where**
filtered-usages g nid f = $\{n \in (usages\ g\ nid).\ f\ (kind\ g\ n)\}$

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*]: $x \in ids\ g \longleftrightarrow kind\ g\ x \neq NoNode$
proof −
have *that*: $x \in ids\ g \longrightarrow kind\ g\ x \neq NoNode$
using *ids.rep-eq kind.rep-eq* **by** *force*
have $kind\ g\ x \neq NoNode \longrightarrow x \in ids\ g$
unfolding *with-default.simps kind-def ids-def*
by (*cases Rep-IRGraph g x* = *None*; *auto*)
from *this that* **show** *?thesis* **by** *auto*
qed

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *replace-node-unchanged*:
gup = *replace-node nid* (*k*, *s*) *g* ⟶ (∀ *n* ∈ (*ids g* − {*nid*}) . *n* ∈ *ids g* ∧ *n* ∈ *ids gup* ∧ *kind g n* = *kind gup n*)
by (*simp add*: *kind.rep-eq replace-node.rep-eq*)

4.0.1 Example Graphs

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

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

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

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

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

value *input-edges eg2-sq*
value *usages eg2-sq 1*

end

5 Data-flow Semantics

theory *IREval*
imports
Graph.IRGraph
begin

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

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

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

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

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

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

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

fun *find-index* :: ′*a* ⇒ ′*a list* ⇒ *nat* **where**
  find-index - [] = *0* |
  find-index v (*x* # *xs*) = (*if* (*x*=*v*) *then 0 else find-index v xs* + *1*)

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

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

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

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

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

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

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

  ValuePhiNode:

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

ValueProxyNode:
$[\![[g,\ m,\ p] \vdash (\mathit{kind}\ g\ c) \mapsto \mathit{val}]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (\mathit{ValueProxyNode}\ c\ \text{-}) \mapsto \mathit{val}\ |$

— Unary arithmetic operators

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

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

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

— Binary arithmetic operators

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

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

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

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

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

— Binary logical bitwise operators

AndNode:
$[\![[g,\ m,\ p] \vdash (\mathit{kind}\ g\ x) \mapsto v1;$

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

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

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

— Comparison operators

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

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

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

— Other nodes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

inductive *eval-graph* :: *IRGraph* ⇒ *ID* ⇒ *Value list* ⇒ *Value* ⇒ *bool*
  where
  ⟦[*g*, *new-map-state*, *ps*] ⊢ (*kind g nid*) ↦ *val*⟧
    ⟹ *eval-graph g nid ps val*

code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*) *eval-graph* **.**

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

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

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

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

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

lemma *is-floating-node n* ⟷ ¬(*has-control-flow n*)
  by *simp*

lemma *n* ∈ *control-nodes* ⟷ *n* ∉ *floating-nodes*
  by (*simp add*: *control-nodes-def floating-nodes-def*)

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

lemma *evalAddNode* : [*g*, *m*, *p*] ⊢ (*AddNode x y*) ↦ *val* ⟹
  (∃ *v1*. ([*g*, *m*, *p*] ⊢ (*kind g x*) ↦ *v1*) ∧
    (∃ *v2*. ([*g*, *m*, *p*] ⊢ (*kind g y*) ↦ *v2*) ∧
      val = *intval-add v1 v2*))
  using *AddNodeE plus-Value-def* **by** *metis*

lemma *not-floating*: (∃ *y ys*. (*successors-of n*) = *y* # *ys*) ⟶ ¬(*is-floating-node n*)
  unfolding *is-floating-node.simps*
  by (*induct n*; *simp add*: *neq-Nil-conv*)

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

theorem *evalDet*:

$([g, m, p] \vdash node \mapsto val1) \Longrightarrow$
$(\forall\ val2.\ (([g, m, p] \vdash node \mapsto val2) \longrightarrow val1 = val2))$
apply (*induction rule*: *eval.induct*)
by (*rule allI*; *rule impI*; *elim EvalE*; *auto*)+

theorem *evalAllDet*:
$([g, m, p] \vdash nodes \longmapsto vals1) \Longrightarrow$
$(\forall\ vals2.\ (([g, m, p] \vdash nodes \longmapsto vals2) \longrightarrow vals1 = vals2))$
apply (*induction rule*: *eval-all.induct*)
using *eval-all.cases* **apply** *blast*
by (*metis evalDet eval-all.cases list.discI list.inject*)

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 r f $(h, n) = h\ r\ f$

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

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 *RefFieldHeap* $=$ (*objref*, *string*) *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* ⇒ *Params* ⇒ (*ID* × *MapState* × *RefFieldHeap*) ⇒ (*ID* × *MapState* × *RefFieldHeap*) ⇒ *bool*
(-, - ⊢ - → - *55*) **for** *g p* **where**

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

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

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

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

m′ = *set-phis phis vs m*⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*merge*, *m′*, *h*) |

NewInstanceNode:
⟦*kind g nid* = (*NewInstanceNode nid f obj nid′*);
(*h′*, *ref*) = *h-new-inst h*;
m′ = *m*(*nid* := *ref*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m′*, *h′*) |

LoadFieldNode:
⟦*kind g nid* = (*LoadFieldNode nid f* (*Some obj*) *nid′*);
[*g*, *m*, *p*] ⊢ (*kind g obj*) ↦ *ObjRef ref*;
h-load-field ref f h = *v*;
m′ = *m*(*nid* := *v*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m′*, *h*) |

SignedDivNode:
⟦*kind g nid* = (*SignedDivNode nid x y zero sb nxt*);
[*g*, *m*, *p*] ⊢ (*kind g x*) ↦ *v1*;

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

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

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

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

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

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

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

theorem *stepDet*:
$(g,\ p \vdash (\mathit{nid},m,h) \rightarrow \mathit{next}) \Longrightarrow$
$(\forall\ \mathit{next}'.\ ((g,\ p \vdash (\mathit{nid},m,h) \rightarrow \mathit{next}') \longrightarrow \mathit{next} = \mathit{next}'))$
proof (*induction rule*: *step.induct*)
  case (*SequentialNode nid next m h*)
  have *notif*: ¬(*is-IfNode* (*kind g nid*))
    using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
    by (*metis is-IfNode-def*)
  have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
    using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*

```
    by (metis is-AbstractEndNode.simps is-EndNode.elims(2) is-LoopEndNode-def)
  have notnew: ¬(is-NewInstanceNode (kind g nid))
    using SequentialNode.hyps(1) is-sequential-node.simps
    by (metis is-NewInstanceNode-def)
  have notload: ¬(is-LoadFieldNode (kind g nid))
    using SequentialNode.hyps(1) is-sequential-node.simps
    by (metis is-LoadFieldNode-def)
  have notstore: ¬(is-StoreFieldNode (kind g nid))
    using SequentialNode.hyps(1) is-sequential-node.simps
    by (metis is-StoreFieldNode-def)
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    using SequentialNode.hyps(1) is-sequential-node.simps is-SignedDivNode-def
is-SignedRemNode-def
    by (metis is-IntegerDivRemNode.simps)
  from notif notend notnew notload notstore notdivrem
  show ?case using SequentialNode step.cases
   by (smt (verit) IRNode.discI(18) is-IfNode-def is-NewInstanceNode-def is-StoreFieldNode-def
is-sequential-node.simps(38) is-sequential-node.simps(39) old.prod.inject)
next
  case (IfNode nid cond tb fb m val next h)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: IfNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: IfNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: IfNode.hyps(1))
  from notseq notend notdivrem show ?case using IfNode evalDet
    using IRNode.distinct(871) IRNode.distinct(891) IRNode.distinct(909) IRN-
ode.distinct(923)
    by (smt (z3) IRNode.distinct(893) IRNode.distinct(913) IRNode.distinct(927)
IRNode.distinct(929) IRNode.distinct(933) IRNode.distinct(947) IRNode.inject(11)
Pair-inject step.simps)
next
  case (EndNodes nid merge i phis inputs m vs m' h)
  have notseq: ¬(is-sequential-node (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps is-sequential-node.simps
    by (metis is-EndNode.elims(2) is-LoopEndNode-def)
  have notif: ¬(is-IfNode (kind g nid))
    using EndNodes.hyps(1)
   by (metis is-AbstractEndNode.elims(1) is-EndNode.simps(12) is-IfNode-def IRN-
ode.distinct-disc(900))
  have notref: ¬(is-RefNode (kind g nid))
    using EndNodes.hyps(1) is-sequential-node.simps
    using IRNode.disc(1899) IRNode.distinct(1473) is-AbstractEndNode.simps
is-EndNode.elims(2) is-LoopEndNode-def is-RefNode-def
    by (metis IRNode.distinct(737) IRNode.distinct-disc(1518))
```

have *notnew*: ¬(*is-NewInstanceNode* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-AbstractEndNode.simps*
using *IRNode.distinct-disc*(*1442*) *is-EndNode.simps*(*29*) *is-NewInstanceNode-def*
by (*metis IRNode.distinct-disc*(*1483*))
have *notload*: ¬(*is-LoadFieldNode* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-AbstractEndNode.simps*
by (*metis IRNode.disc*(*939*) *is-EndNode.simps*(*19*) *is-LoadFieldNode-def*)
have *notstore*: ¬(*is-StoreFieldNode* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-AbstractEndNode.simps*
using *IRNode.distinct-disc*(*1504*) *is-EndNode.simps*(*39*) *is-StoreFieldNode-def*
by *fastforce*
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-AbstractEndNode.simps is-SignedDivNode-def is-SignedRemNode-def*
using *IRNode.distinct-disc*(*1498*) *IRNode.distinct-disc*(*1500*) *is-IntegerDivRemNode.simps*
is-EndNode.simps(*36*) *is-EndNode.simps*(*37*)
by *auto*
from *notseq notif notref notnew notload notstore notdivrem*
show *?case* **using** *EndNodes evalAllDet*
by (*smt* (*z3*) *is-IfNode-def is-LoadFieldNode-def is-NewInstanceNode-def is-RefNode-def*
is-StoreFieldNode-def is-SignedDivNode-def is-SignedRemNode-def Pair-inject is-IntegerDivRemNode.elims(*3*)
step.cases)
next
case (*NewInstanceNode nid f obj nxt h′ ref h m′ m*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notif*: ¬(*is-IfNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notref*: ¬(*is-RefNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notload*: ¬(*is-LoadFieldNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notstore*: ¬(*is-StoreFieldNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
from *notseq notend notif notref notload notstore notdivrem*
show *?case* **using** *NewInstanceNode step.cases*
by (*smt* (*z3*) *IRNode.discI*(*11*) *IRNode.discI*(*18*) *IRNode.discI*(*38*) *IRNode.distinct*(*1777*)
IRNode.distinct(*1779*) *IRNode.distinct*(*1797*) *IRNode.inject*(*28*) *Pair-inject*)
next

case (*LoadFieldNode nid f obj nxt m ref h v m'*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *LoadFieldNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *LoadFieldNode.hyps*(*1*))
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *LoadFieldNode.hyps*(*1*))
from *notseq notend notdivrem*
show *?case* **using** *LoadFieldNode step.cases*
by (*smt* (*z3*) *IRNode.distinct*(*1333*) *IRNode.distinct*(*1347*) *IRNode.distinct*(*1349*) *IRNode.distinct*(*1353*) *IRNode.distinct*(*893*) *IRNode.inject*(*18*) *Pair-inject Value.inject*(*4*) *evalDet option.distinct*(*1*) *option.inject*)
next
case (*StaticLoadFieldNode nid f nxt h v m' m*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *StaticLoadFieldNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *StaticLoadFieldNode.hyps*(*1*))
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
by (*simp add*: *StaticLoadFieldNode.hyps*(*1*))
from *notseq notend notdivrem*
show *?case* **using** *StaticLoadFieldNode step.cases*
by (*smt* (*z3*) *IRNode.distinct*(*1333*) *IRNode.distinct*(*1347*) *IRNode.distinct*(*1349*) *IRNode.distinct*(*1353*) *IRNode.distinct*(*1367*) *IRNode.distinct*(*893*) *IRNode.distinct*(*1297*) *IRNode.distinct*(*1315*) *IRNode.distinct*(*1329*) *IRNode.distinct*(*871*) *IRNode.inject*(*18*) *Pair-inject option.discI*)
next
case (*StoreFieldNode nid f newval uu obj nxt m val ref h' h m'*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *StoreFieldNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *StoreFieldNode.hyps*(*1*))
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
by (*simp add*: *StoreFieldNode.hyps*(*1*))
from *notseq notend notdivrem*
show *?case* **using** *StoreFieldNode step.cases*
by (*smt* (*z3*) *IRNode.distinct*(*1353*) *IRNode.distinct*(*1783*) *IRNode.distinct*(*1965*) *IRNode.distinct*(*1983*) *IRNode.distinct*(*933*) *IRNode.inject*(*38*) *Pair-inject Value.inject*(*4*) *evalDet option.distinct*(*1*) *option.inject*)
next
case (*StaticStoreFieldNode nid f newval uv nxt m val h' h m'*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))

using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *StaticStoreFieldNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *StaticStoreFieldNode.hyps*(*1*))
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
by (*simp add*: *StaticStoreFieldNode.hyps*(*1*))
from *notseq notend notdivrem*
show *?case* **using** *StoreFieldNode step.cases*
by (*smt* (*z3*) *IRNode.distinct*(*1315*) *IRNode.distinct*(*1353*) *IRNode.distinct*(*1783*) *IRNode.distinct*(*1965*)
IRNode.distinct(*1983*) *IRNode.distinct*(*2027*) *IRNode.distinct*(*933*) *IRNode.inject*(*38*) *IRNode.distinct*(*1725*) *Pair-inject StaticStoreFieldNode.hyps*(*1*) *StaticStoreFieldNode.hyps*(*2*) *StaticStoreFieldNode.hyps*(*3*) *StaticStoreFieldNode.hyps*(*4*) *evalDet option.discI*)
next
case (*SignedDivNode nid x y zero sb nxt m v1 v2 v m′ h*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *SignedDivNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *SignedDivNode.hyps*(*1*))
from *notseq notend*
show *?case* **using** *SignedDivNode step.cases*
by (*smt* (*z3*) *IRNode.distinct*(*1347*) *IRNode.distinct*(*1777*) *IRNode.distinct*(*1961*) *IRNode.distinct*(*1965*) *IRNode.distinct*(*1979*) *IRNode.distinct*(*927*) *IRNode.inject*(*35*) *Pair-inject evalDet*)
next
case (*SignedRemNode nid x y zero sb nxt m v1 v2 v m′ h*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *SignedRemNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *SignedRemNode.hyps*(*1*))
from *notseq notend*
show *?case* **using** *SignedRemNode step.cases*
by (*smt* (*z3*) *IRNode.distinct*(*1349*) *IRNode.distinct*(*1779*) *IRNode.distinct*(*1961*) *IRNode.distinct*(*1983*) *IRNode.distinct*(*1997*) *IRNode.distinct*(*929*) *IRNode.inject*(*36*) *Pair-inject evalDet*)
qed

lemma *stepRefNode*:
$[\![$*kind g nid* = *RefNode nid′*$]\!] \Longrightarrow g,\ p \vdash (nid,m,h) \rightarrow (nid',m,h)$
by (*simp add*: *SequentialNode*)

lemma *IfNodeStepCases*:
assumes *kind g nid* = *IfNode cond tb fb*

```
  assumes [g, m, p] ⊢ kind g cond ↦ v
  assumes g, p ⊢ (nid, m, h) → (nid′, m, h)
  shows nid′ ∈ {tb, fb}
  using step.IfNode
  by (metis assms(1) assms(2) assms(3) insert-iff prod.inject stepDet)

lemma IfNodeSeq:
  shows kind g nid = IfNode cond tb fb ⟶ ¬(is-sequential-node (kind g nid))
  unfolding is-sequential-node.simps by simp

lemma IfNodeCond:
  assumes kind g nid = IfNode cond tb fb
  assumes g, p ⊢ (nid, m, h) → (nid′, m, h)
  shows ∃ v. ([g, m, p] ⊢ kind g cond ↦ v)
  using assms(2,1) by (induct (nid,m,h) (nid′,m,h) rule: step.induct; auto)

lemma step-in-ids:
  assumes g, p ⊢ (nid, m, h) → (nid′, m′, h′)
  shows nid ∈ ids g
  using assms apply (induct (nid, m, h) (nid′, m′, h′) rule: step.induct)
  using is-sequential-node.simps(45) not-in-g
  apply simp
  apply (metis is-sequential-node.simps(46))
  using ids-some apply (metis IRNode.simps(990))
  using EndNodes(1) is-AbstractEndNode.simps is-EndNode.simps(45) ids-some
  apply (metis IRNode.disc(965))
  by simp+
```

6.3 Interprocedural Semantics

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

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

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

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

    callTarget = ir-callTarget (kind g nid);
    kind g callTarget = (MethodCallTargetNode targetMethod arguments);
    Some targetGraph = P targetMethod;
```

$m' = \textit{new-map-state}$;
$[g,\ m,\ p] \vdash \textit{arguments} \longmapsto p'\rrbracket$
$\Longrightarrow P \vdash ((g,nid,m,p)\#stk,\ h) \longrightarrow ((targetGraph,0,m',p')\#(g,nid,m,p)\#stk,\ h)$
|

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

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

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

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

UnwindNode:
$\llbracket \textit{kind } g \textit{ nid} = (\textit{UnwindNode exception})$;

$[g,\ m,\ p] \vdash (\textit{kind } g \textit{ exception}) \mapsto e$;

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

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

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

6.4 Big-step Execution

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

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

inductive *exec* :: *Program*
⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *RefFieldHeap*
⇒ *Trace*
⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *RefFieldHeap*
⇒ *Trace*
⇒ *bool*
($\text{-} \vdash \text{-} \mid \text{-} \longrightarrow\!* \text{-} \mid \text{-}$)
for *P*
where
$\llbracket P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h')$;

$\neg$(*has-return m′*);

l′ = (*l* @ [(*g*, *nid*,*m*,*p*)]);

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

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

l′ = (*l* @ [(*g*,*nid*,*m*,*p*)])⟧
⟹ *exec P* (((*g*,*nid*,*m*,*p*)#*xs*),*h*) *l* (((*g′*,*nid′*,*m′*,*p′*)#*ys*),*h′*) *l′*
code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *o* ⇒ *bool as Exec*) *exec* **.**

inductive *exec-debug* :: *Program*
⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *RefFieldHeap*
⇒ *nat*
⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *RefFieldHeap*
⇒ *bool*
(-⊢-⟶∗-∗ -)
where
⟦*n* > *0*;
p ⊢ *s* ⟶ *s′*;
exec-debug p s′ (*n* − *1*) *s′′*⟧
⟹ *exec-debug p s n s′′* |

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

6.4.1 Heap Testing

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

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

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

definition *eg3-sq* :: *IRGraph* **where**
eg3-sq = *irgraph* [
(*0*, *StartNode None 4*, *VoidStamp*),
(*1*, *ParameterNode 0*, *default-stamp*),

(*3*, *MulNode 1 1*, *default-stamp*),
(*4*, *StoreFieldNode 4 field-sq 3 None None 5*, *VoidStamp*),
(*5*, *ReturnNode* (*Some 3*) *None*, *default-stamp*)
]

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

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

values {*h-load-field* (*Some 0*) *field-sq* (*prod.snd res*)
| *res.* (λx. *Some eg4-sq*) ⊢ ([(*eg4-sq*, *0*, *new-map-state*, *p3*), (*eg4-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) →∗*3*∗ *res*}
end

7 Proof Infrastructure

7.1 Bisimulation

theory *Bisimulation*
imports
Stuttering
begin

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

A strong bisimilution between no-op transitions

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

$\forall Q'. (g', p \vdash (nid, m, h) \rightarrow Q') \longrightarrow (\exists P' . (g, p \vdash (nid, m, h) \rightarrow P') \wedge P' = Q')]\!]$
$\Longrightarrow nid \mid g \sim g'$

lemma *lockstep-strong-bisimilulation*:
assumes $g' = replace\text{-}node\ nid\ node\ g$
assumes $g, p \vdash (nid, m, h) \rightarrow (nid', m, h)$
assumes $g', p \vdash (nid, m, h) \rightarrow (nid', m, h)$
shows $nid \mid g \sim g'$
using *assms*(*2*) *assms*(*3*) *stepDet strong-noop-bisimilar.simps* **by** *metis*

lemma *no-step-bisimulation*:
assumes $\forall m\ p\ h\ nid'\ m'\ h'. \neg(g, p \vdash (nid, m, h) \rightarrow (nid', m', h'))$
assumes $\forall m\ p\ h\ nid'\ m'\ h'. \neg(g', p \vdash (nid, m, h) \rightarrow (nid', m', h'))$
shows $nid \mid g \sim g'$
using *assms*
by (*simp add*: *assms*(*1*) *assms*(*2*) *strong-noop-bisimilar.intros*)

end

7.2 Formedness Properties

theory *Form*
imports
Semantics.IREval
begin

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

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

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

definition *wf-ends* **where**
wf-ends g =
(∀ *n* ∈ *ids g* .

is-AbstractEndNode (*kind* *g* *n*) $\longrightarrow$
card (*usages* *g* *n*) > *0*)

fun *wf-graph* :: *IRGraph* $\Rightarrow$ *bool* **where**
wf-graph *g* = (*wf-start* *g* $\wedge$ *wf-closed* *g* $\wedge$ *wf-phis* *g* $\wedge$ *wf-ends* *g*)

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

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

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

lemma *wf-empty*: *wf-graph start-end-graph*
unfolding *start-end-graph-def wf-folds* **by** *simp*

lemma *wf-eg2-sq*: *wf-graph eg2-sq*
unfolding *eg2-sq-def wf-folds* **by** *simp*

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

end

7.3 Dynamic Frames

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

theory *IRGraphFrames*
imports
Form
Semantics.IREval
begin

fun *unchanged* :: *ID set* $\Rightarrow$ *IRGraph* $\Rightarrow$ *IRGraph* $\Rightarrow$ *bool* **where**

unchanged ns g1 g2 = ($\forall$ *n* . *n* $\in$ *ns* $\longrightarrow$
  (*n* $\in$ *ids g1* $\wedge$ *n* $\in$ *ids g2* $\wedge$ *kind g1 n* = *kind g2 n*))

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

lemma *node-unchanged*:
assumes *unchanged ns g1 g2*
assumes *nid* $\in$ *ns*
shows *kind g1 nid* = *kind g2 nid*
using *assms* **by** *auto*

lemma *other-node-unchanged*:
assumes *changeonly ns g1 g2*
assumes *nid* $\in$ *ids g1*
assumes *nid* $\notin$ *ns*
shows *kind g1 nid* = *kind g2 nid*
using *assms*
using *changeonly.simps* **by** *blast*

Some notation for input nodes used

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

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

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

use-trans: $[\![$*eval-uses g nid nid'*;
  eval-uses g nid' nid''$]\!]$
  $\Longrightarrow$ *eval-uses g nid nid''*

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

lemma *eval-usages-self*:
assumes *nid* $\in$ *ids g*
shows *nid* $\in$ *eval-usages g nid*
using *assms eval-usages.simps eval-uses.intros*(*1*)
by (*simp add*: *ids.rep-eq*)

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

```
proof -
  have k: kind g nid = NoNode using assms not-in-g by blast
  then show ?thesis by (simp add: k)
qed

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

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


lemma inp-in-g:
  assumes n ∈ inputs g nid
  shows nid ∈ ids g
proof -
  have inputs g nid ≠ {}
    using assms
    by (metis empty-iff empty-set)
  then have kind g nid ≠ NoNode
    using not-in-g-inputs
    using ids-some by blast
  then show ?thesis
    using not-in-g
    by metis
qed

lemma inp-in-g-wf:
  assumes wf-graph g
  assumes n ∈ inputs g nid
  shows n ∈ ids g
  using assms unfolding wf-folds
  using inp-in-g by blast


lemma kind-unchanged:
  assumes nid ∈ ids g1
  assumes unchanged (eval-usages g1 nid) g1 g2
  shows kind g1 nid = kind g2 nid
```

proof –
  show *?thesis*
    using *assms eval-usages-self*
    using *unchanged.simps* **by** *blast*
qed

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*
  by (*smt assms*(*1*) *assms*(*2*) *eval-usages.simps mem-Collect-eq*
    unchanged.simps use-inp use-trans)

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

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

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

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

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

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

```
  using assms by (cases kind g nid = NoNode; auto)


theorem stay-same:
  assumes nc: unchanged (eval-usages g1 nid) g1 g2
  assumes g1: [g1, m, p] ⊢ (kind g1 nid) ↦ v1
  assumes wf: wf-graph g1
  shows [g2, m, p] ⊢ (kind g2 nid) ↦ v1
proof –
  have nid: nid ∈ ids g1
    using g1 eval-in-ids by simp
  then have nid ∈ eval-usages g1 nid
    using eval-usages-self by blast
  then have kind-same: kind g1 nid = kind g2 nid
    using nc node-unchanged by blast
  show ?thesis using g1 nid nc
  proof (induct (kind g1 nid) v1 arbitrary: nid rule: eval.induct)
    print-cases
    case const: (ConstantNode c)
    then have (kind g2 nid) = ConstantNode c
      using kind-unchanged by metis
    then show ?case using eval.ConstantNode const.hyps(1) by metis
  next
    case param: (ParameterNode val i)
    show ?case
     by (metis eval.ParameterNode kind-unchanged param.hyps(1) param.prems(1)
param.prems(2))
  next
    case (ValuePhiNode nida vals merges)
    then have kind: (kind g2 nid) = ValuePhiNode nida vals merges
      using kind-unchanged by metis
    then show ?case
      using eval.ValuePhiNode kind ValuePhiNode.hyps(1) by metis
  next
    case (ValueProxyNode child val - nid)
    from ValueProxyNode.prems(1) ValueProxyNode.hyps(3)
    have inp-in: child ∈ inputs g1 nid
      using child-member-in inputs-of-ValueProxyNode
      by (metis member-rec(1))
    then have cin: child ∈ ids g1
      using wf inp-in-g-wf by blast
    from inp-in have unc: unchanged (eval-usages g1 child) g1 g2
      using child-unchanged ValueProxyNode.prems(2) by metis
    then have [g2, m, p] ⊢ (kind g2 child) ↦ val
      using ValueProxyNode.hyps(2) cin
      by blast
    then show ?case
      by (metis ValueProxyNode.hyps(3) ValueProxyNode.prems(1) ValueProxyN-
ode.prems(2) eval.ValueProxyNode kind-unchanged)
```

next
case (*AbsNode x v -*)
then have *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-unchanged elim-inp-set ids-some inputs-of.simps*(*1*) *list.set-intros*(*1*))
then have [*g2*, *m*, *p*] ⊢ (*kind g2 x*) ↦ *IntVal32 v*
using *AbsNode.hyps*(*1*) *AbsNode.hyps*(*2*) *not-in-g*
by (*metis AbsNode.hyps*(*3*) *AbsNode.prems*(*1*) *elim-inp-set ids-some inp-in-g-wf inputs-of.simps*(*1*) *list.set-intros*(*1*) *wf*)
then show *?case*
by (*metis AbsNode.hyps*(*3*) *AbsNode.prems*(*1*) *AbsNode.prems*(*2*) *eval.AbsNode kind-unchanged*)
next
case *Node*: (*NegateNode x v -*)
from *inputs-of-NegateNode Node.hyps*(*3*) *Node.prems*(*1*)
have *xinp*: *x* ∈ *inputs g1 nid*
using *child-member-in* **by** (*metis member-rec*(*1*))
then have *xin*: *x* ∈ *ids g1*
using *wf inp-in-g-wf* **by** *blast*
from *xinp child-unchanged Node.prems*(*2*)
have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2* **by** *blast*
have *x1*:[*g1*, *m*, *p*] ⊢ (*kind g1 x*) ↦ *v*
using *Node.hyps*(*1*) *Node.hyps*(*2*)
by *blast*
have *x2*: [*g2*, *m*, *p*] ⊢ (*kind g2 x*) ↦ *v*
using *kind-unchanged ux xin Node.hyps*
by *blast*
then show *?case*
using *kind-same Node.hyps*(*1,3*) *eval.NegateNode*
by (*metis Node.prems*(*1*) *Node.prems*(*2*) *kind-unchanged ux xin*)
next
case *node*:(*AddNode x v1 y v2*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-unchanged inputs.simps inputs-of-AddNode list.set-intros*(*1*))
then have *x*: [*g1*, *m*, *p*] ⊢ (*kind g1 x*) ↦ *v1*
using *node.hyps*(*1*) **by** *blast*
have *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis IRNodes2.inputs-of-AddNode child-member-in child-unchanged member-rec*(*1*) *node.hyps*(*5*) *node.prems*(*1*) *node.prems*(*2*))
have *y*: [*g1*, *m*, *p*] ⊢ (*kind g1 y*) ↦ *v2*
using *node.hyps*(*3*) **by** *blast*
show *?case*
using *node.hyps node.prems ux x uy y*
by (*metis AddNode inputs.simps inp-in-g-wf inputs-of-AddNode kind-unchanged list.set-intros*(*1*) *set-subset-Cons subset-iff wf*)
next
case *node*:(*SubNode x v1 y v2*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-SubNode member-rec*(*1*))
then have *x*: [*g1*, *m*, *p*] ⊢ (*kind g1 x*) ↦ *v1*

```
      using node.hyps(1) by blast
    from node have uy: unchanged (eval-usages g1 y) g1 g2
     by (metis child-member-in child-unchanged inputs-of-SubNode member-rec(1))
    have y: [g1, m, p] ⊢ (kind g1 y) ↦ v2
      using node.hyps(3) by blast
    show ?case
      using node.hyps node.prems ux x uy y
    by (metis SubNode inputs.simps inputs-of-SubNode kind-unchanged list.set-intros(1)
set-subset-Cons subsetD wf wf-folds(1,3))
  next
    case node:(MulNode x v1 y v2)
    then have ux: unchanged (eval-usages g1 x) g1 g2
     by (metis child-member-in child-unchanged inputs-of-MulNode member-rec(1))
    then have x: [g1, m, p] ⊢ (kind g1 x) ↦ v1
      using node.hyps(1) by blast
    from node have uy: unchanged (eval-usages g1 y) g1 g2
     by (metis child-member-in child-unchanged inputs-of-MulNode member-rec(1))
    have y: [g1, m, p] ⊢ (kind g1 y) ↦ v2
      using node.hyps(3) by blast
    show ?case
      using node.hyps node.prems ux x uy y
    by (metis MulNode inputs.simps inputs-of-MulNode kind-unchanged list.set-intros(1)
set-subset-Cons subsetD wf wf-folds(1,3))
  next
    case node:(AndNode x v1 y v2)
    then have ux: unchanged (eval-usages g1 x) g1 g2
     by (metis child-member-in child-unchanged inputs-of-AndNode member-rec(1))
    then have x: [g1, m, p] ⊢ (kind g1 x) ↦ v1
      using node.hyps(1) by blast
    from node have uy: unchanged (eval-usages g1 y) g1 g2
     by (metis child-member-in child-unchanged inputs-of-AndNode member-rec(1))
    have y: [g1, m, p] ⊢ (kind g1 y) ↦ v2
      using node.hyps(3) by blast
    show ?case
      using node.hyps node.prems ux x uy y
    by (metis AndNode inputs.simps inputs-of-AndNode kind-unchanged list.set-intros(1)
set-subset-Cons subsetD wf wf-folds(1,3))
  next
    case node: (OrNode x v1 y v2)
    then have ux: unchanged (eval-usages g1 x) g1 g2
      by (metis child-member-in child-unchanged inputs-of-OrNode member-rec(1))
    then have x: [g1, m, p] ⊢ (kind g1 x) ↦ v1
      using node.hyps(1) by blast
    from node have uy: unchanged (eval-usages g1 y) g1 g2
      by (metis child-member-in child-unchanged inputs-of-OrNode member-rec(1))
    have y: [g1, m, p] ⊢ (kind g1 y) ↦ v2
      using node.hyps(3) by blast
    show ?case
      using node.hyps node.prems ux x uy y
```

by (*metis OrNode inputs.simps inputs-of-OrNode kind-unchanged list.set-intros*(*1*) *set-subset-Cons subsetD wf wf-folds*(*1*,*3*))
next
case *node*: (*XorNode x v1 y v2*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-XorNode member-rec*(*1*))
then have *x*: [*g1*, *m*, *p*] ⊢ (*kind g1 x*) ↦ *v1*
using *node.hyps*(*1*) **by** *blast*
from *node* **have** *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-XorNode member-rec*(*1*))
have *y*: [*g1*, *m*, *p*] ⊢ (*kind g1 y*) ↦ *v2*
using *node.hyps*(*3*) **by** *blast*
show *?case*
using *node.hyps node.prems ux x uy y*
by (*metis XorNode inputs.simps inputs-of-XorNode kind-unchanged list.set-intros*(*1*) *set-subset-Cons subsetD wf wf-folds*(*1*,*3*))
next
case *node*: (*IntegerEqualsNode x v1 y v2 val*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-IntegerEqualsNode member-rec*(*1*))
then have *x*: [*g1*, *m*, *p*] ⊢ (*kind g1 x*) ↦ *IntVal32 v1*
using *node.hyps*(*1*) **by** *blast*
from *node* **have** *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-IntegerEqualsNode member-rec*(*1*))
have *y*: [*g1*, *m*, *p*] ⊢ (*kind g1 y*) ↦ *IntVal32 v2*
using *node.hyps*(*3*) **by** *blast*
show *?case*
using *node.hyps node.prems ux x uy y*
by (*metis* (*full-types*) *IntegerEqualsNode child-member-in in-set-member inputs-of-IntegerEqualsNode kind-unchanged list.set-intros*(*1*) *set-subset-Cons subsetD wf wf-folds*(*1*,*3*))
next
case *node*: (*IntegerLessThanNode x v1 y v2 val*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-IntegerLessThanNode member-rec*(*1*))
then have *x*: [*g1*, *m*, *p*] ⊢ (*kind g1 x*) ↦ *IntVal32 v1*
using *node.hyps*(*1*) **by** *blast*
from *node* **have** *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-IntegerLessThanNode member-rec*(*1*))
have *y*: [*g1*, *m*, *p*] ⊢ (*kind g1 y*) ↦ *IntVal32 v2*
using *node.hyps*(*3*) **by** *blast*
show *?case*
using *node.hyps node.prems ux x uy y*
by (*metis* (*full-types*) *IntegerLessThanNode child-member-in in-set-member inputs-of-IntegerLessThanNode kind-unchanged list.set-intros*(*1*) *set-subset-Cons sub-*

setD wf wf-folds(1,3))
next
case *node*: (*ShortCircuitOrNode x v1 y v2 val*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-ShortCircuitOrNode member-rec(1)*)
then have *x*: [*g1*, *m*, *p*] $\vdash$ (*kind g1 x*) $\mapsto$ *IntVal32 v1*
using *node.hyps(1)* **by** *blast*
from *node* **have** *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-ShortCircuitOrNode member-rec(1)*)
have *y*: [*g1*, *m*, *p*] $\vdash$ (*kind g1 y*) $\mapsto$ *IntVal32 v2*
using *node.hyps(3)* **by** *blast*
have *x2*: [*g2*, *m*, *p*] $\vdash$ (*kind g2 x*) $\mapsto$ *IntVal32 v1*
by (*metis inputs.simps inputs-of-ShortCircuitOrNode list.set-intros(1) node.hyps(2) node.hyps(6) node.prems(1) subsetD ux wf wf-folds(1,3)*)
have *y2*: [*g2*, *m*, *p*] $\vdash$ (*kind g2 y*) $\mapsto$ *IntVal32 v2*
by (*metis basic-trans-rules(31) inputs.simps inputs-of-ShortCircuitOrNode list.set-intros(1) node.hyps(4) node.hyps(6) node.prems(1) set-subset-Cons uy wf wf-folds(1,3)*)
show *?case*
using *node.hyps node.prems ux x uy y x2 y2*
by (*metis ShortCircuitOrNode kind-unchanged*)
next
case *node*: (*LogicNegationNode x v1 val nida*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-LogicNegationNode member-rec(1)*)
then have *x*:[*g2*, *m*, *p*] $\vdash$ (*kind g2 x*) $\mapsto$ *IntVal32 v1*
using *eval-in-ids node.hyps(1) node.hyps(2)* **by** *blast*
then show *?case*
by (*metis LogicNegationNode kind-unchanged node.hyps(3) node.hyps(4) node.hyps(5) node.prems(1) node.prems(2)*)
next
case *node*:(*ConditionalNode condition cond trueExp trueVal falseExp falseVal val*)
have *c*: *condition* $\in$ *inputs g1 nid*
by (*metis IRNodes2.inputs-of-ConditionalNode child-member-in member-rec(1) node.hyps(8) node.prems(1)*)
then have *unchanged* (*eval-usages g1 condition*) *g1 g2*
using *child-unchanged node.prems(2)* **by** *blast*
then have *cond*: [*g2*, *m*, *p*] $\vdash$ (*kind g2 condition*) $\mapsto$ *IntVal32 cond*
using *node c inp-in-g-wf wf* **by** *blast*

have *t*: *trueExp* $\in$ *inputs g1 nid*
by (*metis IRNodes2.inputs-of-ConditionalNode child-member-in member-rec(1) node.hyps(8) node.prems(1)*)
then have *utrue*: *unchanged* (*eval-usages g1 trueExp*) *g1 g2*
using *node.prems(2) child-unchanged* **by** *blast*

then have *trueVal*: [*g2*, *m*, *p*] ⊢ (*kind g2 trueExp*) ↦ *IntVal32* (*trueVal*)
using *node.hyps node t inp-in-g-wf wf* **by** *blast*

have *f*: *falseExp* ∈ *inputs g1 nid*
by (*metis IRNodes2.inputs-of-ConditionalNode child-member-in member-rec*(*1*) *node.hyps*(*8*) *node.prems*(*1*))
then have *ufalse*: *unchanged* (*eval-usages g1 falseExp*) *g1 g2*
using *node.prems*(*2*) *child-unchanged* **by** *blast*
then have *falseVal*: [*g2*, *m*, *p*] ⊢ (*kind g2 falseExp*) ↦ *IntVal32* (*falseVal*)
using *node.hyps node f inp-in-g-wf wf* **by** *blast*

have [*g2*, *m*, *p*] ⊢ (*kind g2 nid*) ↦ *val*
using *kind-same trueVal falseVal cond*
by (*metis ConditionalNode kind-unchanged node.hyps*(*7*) *node.hyps*(*8*) *node.prems*(*1*) *node.prems*(*2*))
then show *?case*
by *blast*

next
case (*RefNode x val nid*)
have *x*: *x* ∈ *inputs g1 nid*
by (*metis IRNodes2.inputs-of-RefNode RefNode.hyps*(*3*) *RefNode.prems*(*1*) *child-member-in member-rec*(*1*))
then have *ref*: [*g2*, *m*, *p*] ⊢ (*kind g2 x*) ↦ *val*
using *RefNode.hyps*(*2*) *RefNode.prems*(*2*) *child-unchanged inp-in-g-wf wf* **by** *blast*
then show *?case*
by (*metis RefNode.hyps*(*3*) *RefNode.prems*(*1*) *RefNode.prems*(*2*) *eval.RefNode kind-unchanged*)
next
case (*InvokeNodeEval val - callTarget classInit stateDuring stateAfter nex*)
then show *?case*
by (*metis eval.InvokeNodeEval kind-unchanged*)
next
case (*SignedDivNode x v1 y v2 zeroCheck frameState nex*)
then show *?case*
by (*metis eval.SignedDivNode kind-unchanged*)
next
case (*SignedRemNode x v1 y v2 zeroCheck frameState nex*)
then show *?case*
by (*metis eval.SignedRemNode kind-unchanged*)
next
case (*InvokeWithExceptionNodeEval val - callTarget classInit stateDuring stateAfter nex exceptionEdge*)
then show *?case*
by (*metis eval.InvokeWithExceptionNodeEval kind-unchanged*)
next
case (*NewInstanceNode nid clazz stateBefore nex*)
then show *?case*

by (*metis eval.NewInstanceNode kind-unchanged*)
next
case (*IsNullNode obj ref val*)
have *obj*: *obj* ∈ *inputs g1 nid*
by (*metis IRNodes2.inputs-of-IsNullNode IsNullNode.hyps*(*4*) *inputs.simps list.set-intros*(*1*))
then have *ref*: [*g2*, *m*, *p*] ⊢ (*kind g2 obj*) ↦ *ObjRef ref*
using *IsNullNode.hyps*(*1*) *IsNullNode.hyps*(*2*) *IsNullNode.prems*(*2*) *child-unchanged eval-in-ids* **by** *blast*
then show *?case*
by (*metis* (*full-types*) *IsNullNode.hyps*(*3*) *IsNullNode.hyps*(*4*) *IsNullNode.prems*(*1*) *IsNullNode.prems*(*2*) *eval.IsNullNode kind-unchanged*)
next
case (*LoadFieldNode*)
then show *?case*
by (*metis eval.LoadFieldNode kind-unchanged*)
next
case (*PiNode object val*)
have *object*: *object* ∈ *inputs g1 nid*
using *inputs-of-PiNode inputs.simps*
by (*metis PiNode.hyps*(*3*) *list.set-intros*(*1*))
then have *ref*: [*g2*, *m*, *p*] ⊢ (*kind g2 object*) ↦ *val*
using *PiNode.hyps*(*1*) *PiNode.hyps*(*2*) *PiNode.prems*(*2*) *child-unchanged eval-in-ids* **by** *blast*
then show *?case*
by (*metis PiNode.hyps*(*3*) *PiNode.prems*(*1*) *PiNode.prems*(*2*) *eval.PiNode kind-unchanged*)
next
case (*NotNode x val not-val*)
have *object*: *x* ∈ *inputs g1 nid*
using *inputs-of-NotNode inputs.simps*
by (*metis NotNode.hyps*(*4*) *list.set-intros*(*1*))
then have *ref*: [*g2*, *m*, *p*] ⊢ (*kind g2 x*) ↦ *val*
using *NotNode.hyps*(*1*) *NotNode.hyps*(*2*) *NotNode.prems*(*2*) *child-unchanged eval-in-ids* **by** *blast*
then show *?case*
by (*metis NotNode.hyps*(*3*) *NotNode.hyps*(*4*) *NotNode.prems*(*1*) *NotNode.prems*(*2*) *eval.NotNode kind-unchanged*)
qed
qed

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

lemma *disjoint-change*:

assumes *changeonly change g gup*
assumes *nochange* = *ids g* − *change*
shows *unchanged nochange g gup*
using *assms* **unfolding** *changeonly.simps unchanged.simps*
by *blast*

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

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

lemma *wf-use-ids*:
assumes *wf-graph g*
assumes *nid* $\in$ *ids g*
assumes *eval-uses g nid nid'*
shows *nid'* $\in$ *ids g*
using *assms*(*3*)
proof (*induction rule*: *eval-uses.induct*)
case *use0*
then show *?case* **by** *simp*
next
case *use-inp*
then show *?case*
using *assms*(*1*) *inp-in-g-wf* **by** *blast*
next
case *use-trans*
then show *?case* **by** *blast*
qed

lemma *no-external-use*:
assumes *wf-graph g*

```
  assumes nid′ ∉ ids g
  assumes nid ∈ ids g
  shows ¬(eval-uses g nid nid′)
proof −
  have 0: nid ≠ nid′
    using assms by blast
  have inp: nid′ ∉ inputs g nid
    using assms
    using inp-in-g-wf by blast
  have rec-0: ∄n . n ∈ ids g ∧ n = nid′
    using assms by blast
  have rec-inp: ∄n . n ∈ ids g ∧ n ∈ inputs g nid′
    using assms(2) inp-in-g by blast
  have rec: ∄nid′′ . eval-uses g nid nid′′ ∧ eval-uses g nid′′ nid′
    using wf-use-ids assms(1) assms(2) assms(3) by blast
  from inp 0 rec show ?thesis
    using eval-uses-imp by blast
qed

end
```

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′
  using assms replace-node-lookup replace-usages.simps IRNode.distinct(2069)
  by (metis)

lemma replace-usages-changeonly:
  assumes nid ∈ ids g
  assumes g′ = replace-usages nid nid′ g
  shows changeonly {nid} g g′
  using assms unfolding replace-usages.simps
  by (metis DiffI changeonly.elims(3) ids-some replace-node-unchanged)

lemma replace-usages-unchanged:
  assumes nid ∈ ids g
  assumes g′ = replace-usages nid nid′ g
```

shows *unchanged* (*ids g* − {*nid*}) *g g′*
using *assms* **unfolding** *replace-usages.simps*
by (*smt* (*verit*, *del-insts*) *DiffE ids-some replace-node-unchanged unchanged.simps*)

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*
by (*meson Max-gr-iff less-add-one less-irrefl*)

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

lemma *nextNidNotIn*:
ids g ≠ {} ⟶ *nextNid g* ∉ *ids g*
unfolding *nextNid.simps*
using *ids-finite max-plus-one* **by** *blast*

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

lemma *constantConditionTrue*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes *g′* = *constantCondition True ifcond* (*kind g ifcond*) *g*
shows *g′*, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
proof −
have *if′*: *kind g′ ifcond* = *IfNode* (*nextNid g*) *t f*
by (*metis IRNode.simps*(*989*) *assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *replace-node-lookup*)
have *bool-to-val True* = (*IntVal32 1*)
by *auto*
have *ifcond* ≠ (*nextNid g*)
by (*metis IRNode.simps*(*989*) *assms*(*1*) *emptyE ids-some nextNidNotIn*)
then have *c′*: *kind g′* (*nextNid g*) = *ConstantNode* (*IntVal32 1*)
using *assms*(*2*) *replace-node-unchanged*
by (*metis DiffI IRNode.distinct*(*585*) ‹*bool-to-val True* = *IntVal32 1*› *add-node-lookup assms*(*1*) *constantCondition.simps*(*1*) *emptyE insertE not-in-g*)
from *if′ c′* **show** *?thesis* **using** *IfNode*
by (*metis* (*no-types*, *hide-lams*) *ConstantNode val-to-bool.simps*(*1*))
qed

lemma *constantConditionFalse*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes g' = *constantCondition False ifcond* (*kind g ifcond*) *g*
shows g', $p \vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*f*, *m*, *h*)
proof –
have *if'*: *kind* g' *ifcond* = *IfNode* (*nextNid g*) *t f*
by (*metis IRNode.simps*(*989*) *assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *replace-node-lookup*)
have *bool-to-val False* = (*IntVal32 0*)
by *auto*
have *ifcond* $\neq$ (*nextNid g*)
by (*metis IRNode.simps*(*989*) *assms*(*1*) *emptyE ids-some nextNidNotIn*)
then have *kind* g' (*nextNid g*) = *ConstantNode* (*IntVal32 0*)
using *assms*(*2*) *replace-node-unchanged*
by (*metis DiffI IRNode.distinct*(*585*) ‹*bool-to-val False* = *IntVal32 0*› *add-node-lookup assms*(*1*) *constantCondition.simps*(*1*) *emptyE insertE not-in-g*)
then have *c'*: [g', *m*, *p*] $\vdash$ *kind* g' (*nextNid g*) $\mapsto$ *IntVal32 0*
using *ConstantNode* **by** *presburger*
have $\neg$(*val-to-bool* (*IntVal32 0*))
by *simp*
from *if' c'* **show** *?thesis* **using** *IfNode*
using ‹$\neg$ *val-to-bool* (*IntVal32 0*)› **by** *presburger*
qed

lemma *diff-forall*:
assumes $\forall n \in ids\ g - \{nid\}.\ cond\ n$
shows $\forall n.\ n \in ids\ g \wedge n \notin \{nid\} \longrightarrow cond\ n$
by (*meson Diff-iff assms*)

lemma *replace-node-changeonly*:
assumes g' = *replace-node nid node g*
shows *changeonly* {*nid*} *g* g'
using *assms replace-node-unchanged*
unfolding *changeonly.simps* **using** *diff-forall*
by (*metis Rep-IRGraph-inverse add-changed add-node.rep-eq ids-some other-node-unchanged replace-node.rep-eq*)

lemma *add-node-changeonly*:
assumes g' = *add-node nid node g*
shows *changeonly* {*nid*} *g* g'
by (*metis Rep-IRGraph-inverse add-node.rep-eq assms replace-node.rep-eq replace-node-changeonly*)

lemma *constantConditionNoEffect*:
assumes $\neg$(*is-IfNode* (*kind g nid*))
shows *g* = *constantCondition b nid* (*kind g nid*) *g*
using *assms* **apply** (*cases kind g nid*)
using *constantCondition.simps*

apply *presburger+*
apply (*metis is-IfNode-def*)
using *constantCondition.simps*
by *presburger+*

lemma *constantConditionIfNode*:
assumes *kind g nid = IfNode cond t f*
shows *constantCondition val nid* (*kind g nid*) *g* =
replace-node nid (*IfNode* (*nextNid g*) *t f*, *stamp g nid*)
(*add-node* (*nextNid g*) ((*ConstantNode* (*bool-to-val val*)), *constantAsStamp* (*bool-to-val val*)) *g*)
using *constantCondition.simps*
by (*simp add*: *assms*)

lemma *constantCondition-changeonly*:
assumes $nid \in ids\ g$
assumes $g' =$ *constantCondition b nid* (*kind g nid*) *g*
shows *changeonly* $\{nid\}$ $g\ g'$
proof (*cases is-IfNode* (*kind g nid*))
case *True*
have $nextNid\ g \notin ids\ g$
using *nextNidNotIn* **by** (*metis emptyE*)
then show *?thesis* **using** *assms*
using *replace-node-changeonly add-node-changeonly* **unfolding** *changeonly.simps*
using *True constantCondition.simps*(*1*) *is-IfNode-def*
by (*metis* (*full-types*) *DiffD2 Diff-insert-absorb*)
next
case *False*
have $g = g'$
using *constantConditionNoEffect*
using *False assms*(*2*) **by** *blast*
then show *?thesis* **by** *simp*
qed

lemma *constantConditionNoIf*:
assumes $\forall$ *cond t f. kind g ifcond* $\neq$ *IfNode cond t f*
assumes $g' =$ *constantCondition val ifcond* (*kind g ifcond*) *g*
shows $\exists\, nid'\ .(g\ m\ p\ h \vdash ifcond \rightsquigarrow nid') \longleftrightarrow (g'\ m\ p\ h \vdash ifcond \rightsquigarrow nid')$
proof −
have $g' = g$
using *assms*(*2*) *assms*(*1*)
using *constantConditionNoEffect*
by (*metis IRNode.collapse*(*11*))
then show *?thesis* **by** *simp*
qed

lemma *constantConditionValid*:
assumes *kind g ifcond = IfNode cond t f*

assumes $[g, m, p] \vdash$ *kind g cond* $\mapsto v$
assumes *const* = *val-to-bool v*
assumes g' = *constantCondition const ifcond* (*kind g ifcond*) *g*
shows $\exists$ *nid'* .(*g m p h* $\vdash$ *ifcond* $\leadsto$ *nid'*) $\longleftrightarrow$ (g' *m p h* $\vdash$ *ifcond* $\leadsto$ *nid'*)
proof (*cases const*)
case *True*
have *ifstep*: *g*, *p* $\vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*t*, *m*, *h*)
by (*meson IfNode True assms*(*1*) *assms*(*2*) *assms*(*3*))
have *ifstep'*: *g'*, *p* $\vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*t*, *m*, *h*)
using *constantConditionTrue*
using *True assms*(*1*) *assms*(*4*) **by** *presburger*
from *ifstep ifstep'* **show** *?thesis*
using *StutterStep* **by** *blast*
next
case *False*
have *ifstep*: *g*, *p* $\vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*f*, *m*, *h*)
by (*meson IfNode False assms*(*1*) *assms*(*2*) *assms*(*3*))
have *ifstep'*: *g'*, *p* $\vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*f*, *m*, *h*)
using *constantConditionFalse*
using *False assms*(*1*) *assms*(*4*) **by** *presburger*
from *ifstep ifstep'* **show** *?thesis*
using *StutterStep* **by** *blast*
qed

end

7.5 Stuttering

theory *Stuttering*
imports
Semantics.IRStepObj
begin

inductive *stutter*:: *IRGraph* $\Rightarrow$ *MapState* $\Rightarrow$ *Params* $\Rightarrow$ *RefFieldHeap* $\Rightarrow$ *ID* $\Rightarrow$ *ID* $\Rightarrow$ *bool* (- - - - $\vdash$ - $\leadsto$ - *55*)
for *g m p h* **where**

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

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

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

proof −
have *nextin*: $nid' \in \{P'. (g\ m\ p\ h \vdash nid \leadsto P')\}$
using *assms StutterStep* **by** *blast*
have *nextsubset*: $\{nid''. (g\ m\ p\ h \vdash nid' \leadsto nid'')\} \subseteq \{P'. (g\ m\ p\ h \vdash nid \leadsto P')\}$
by (*metis Collect-mono assms stutter.Transitive*)
have $\forall\, n \in \{P'. (g\ m\ p\ h \vdash nid \leadsto P')\}\ .\ n = nid' \lor n \in \{nid''. (g\ m\ p\ h \vdash nid' \leadsto nid'')\}$
using *stepDet*
by (*metis* (*no-types*, *lifting*) *Pair-inject assms mem-Collect-eq stutter.simps*)
then show *?thesis*
using *insert-absorb mk-disjoint-insert nextin nextsubset* **by** *auto*
qed

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:

$[\![kind\ g\ cond = IntegerEqualsNode\ tb\ fb]\!]$
$\Longrightarrow CanonicalizeConditional\ g\ (ConditionalNode\ cond\ tb\ fb)\ (RefNode\ fb)\ |$

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

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

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

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

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

add-xsub:

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

add-ysub:

$[\![kind\ g\ y = SubNode\ a\ x\ ]\!]$

$\Longrightarrow$ *CanonicalizeAdd g* (*AddNode x y*) (*RefNode a*) |

add-xnegate:

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

add-ynegate:

⟦*kind g ny* = *NegateNode y* ⟧
$\Longrightarrow$ *CanonicalizeAdd g* (*AddNode x ny*) (*SubNode x y*)

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

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

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

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

inductive *CanonicalizeBinaryArithmeticNode* :: *ID* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
add-const-fold:
⟦*op* = *kind g op-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:
⟦¬(*is-ConstantNode* (*kind g y*));
((*is-ConstantNode* (*kind g x*)) ∨ ($x > y$))⟧
⟹ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AddNode x y*) (*AddNode y x*) |

and-ids-ordered:
⟦¬(*is-ConstantNode* (*kind g y*));
((*is-ConstantNode* (*kind g x*)) ∨ ($x > y$))⟧
⟹ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AndNode x y*) (*AndNode y x*) |

int-equals-ids-ordered:
⟦¬(*is-ConstantNode* (*kind g y*));
((*is-ConstantNode* (*kind g x*)) ∨ ($x > y$))⟧
⟹ *CanonicalizeCommutativeBinaryArithmeticNode g* (*IntegerEqualsNode x y*) (*IntegerEqualsNode y x*) |

mul-ids-ordered:
⟦¬(*is-ConstantNode* (*kind g y*));
((*is-ConstantNode* (*kind g x*)) ∨ ($x > y$))⟧
⟹ *CanonicalizeCommutativeBinaryArithmeticNode g* (*MulNode x y*) (*MulNode y x*) |

or-ids-ordered:
⟦¬(*is-ConstantNode* (*kind g y*));
((*is-ConstantNode* (*kind g x*)) ∨ ($x > y$))⟧
⟹ *CanonicalizeCommutativeBinaryArithmeticNode g* (*OrNode x y*) (*OrNode y x*) |

xor-ids-ordered:
⟦¬(*is-ConstantNode* (*kind g y*));
((*is-ConstantNode* (*kind g x*)) ∨ ($x > y$))⟧
⟹ *CanonicalizeCommutativeBinaryArithmeticNode g* (*XorNode x y*) (*XorNode y x*) |

add-swap-const-first:
⟦*is-ConstantNode* (*kind g x*);
¬(*is-ConstantNode* (*kind g y*))⟧
⟹ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AddNode x y*) (*AddNode y x*) |

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

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

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

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

xor-swap-const-first:
⟦*is-ConstantNode* (*kind g x*);
¬(*is-ConstantNode* (*kind g y*))⟧
⟹ *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*)⟧
⟹ *CanonicalizeSub g* (*SubNode x y*) (*ConstantNode* (*IntVal32 0*)) |

sub-both-const:
⟦*kind g x* = *ConstantNode c-1*;
kind g y = *ConstantNode c-2*;

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

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:

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

sub-xzero:

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

sub-y-negate:

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

inductive *CanonicalizeMul* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
mul-both-const:
⟦*kind g x* = *ConstantNode c-1*;
kind g y = *ConstantNode c-2*;
val = *intval-mul c-1 c-2*⟧
⟹ *CanonicalizeMul g* (*MulNode x y*) (*ConstantNode val*) |

mul-xzero:
⟦*kind g x* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g y*));
c-1 = (*IntVal32 0*)⟧
⟹ *CanonicalizeMul g* (*MulNode x y*) (*ConstantNode c-1*) |

mul-yzero:
⟦*kind g y* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g x*));
c-1 = (*IntVal32 0*)⟧
⟹ *CanonicalizeMul g* (*MulNode x y*) (*ConstantNode c-1*) |

mul-xone:
⟦*kind g x* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g y*));
c-1 = (*IntVal32 1*)⟧
⟹ *CanonicalizeMul g* (*MulNode x y*) (*RefNode y*) |

mul-yone:
⟦*kind g y* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g x*));
c-1 = (*IntVal32 1*)⟧
⟹ *CanonicalizeMul g* (*MulNode x y*) (*RefNode x*) |

mul-xnegate:
⟦*kind g x* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g y*));
c-1 = (*IntVal32* (−*1*))⟧
⟹ *CanonicalizeMul g* (*MulNode x y*) (*NegateNode y*) |

mul-ynegate:
⟦*kind g y* = *ConstantNode c-1*;
¬(*is-ConstantNode* (*kind g x*));
c-1 = (*IntVal32* (−*1*))⟧
⟹ *CanonicalizeMul g* (*MulNode x y*) (*NegateNode x*)

inductive *CanonicalizeAbs* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
abs-abs:
⟦*kind g x* = (*AbsNode y*)⟧
⟹ *CanonicalizeAbs g* (*AbsNode x*) (*AbsNode y*) |

abs-negate:
⟦*kind g nx* = (*NegateNode x*)⟧
⟹ *CanonicalizeAbs g* (*AbsNode nx*) (*AbsNode x*)

inductive *CanonicalizeNegate* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
negate-const:
⟦*kind g nx* = (*ConstantNode val*);
val = (*IntVal32 v*);
neg-val = *intval-sub* (*IntVal32 0*) *val* ⟧
⟹ *CanonicalizeNegate g* (*NegateNode nx*) (*ConstantNode neg-val*) |

negate-negate:
⟦*kind g nx* = (*NegateNode x*)⟧
⟹ *CanonicalizeNegate g* (*NegateNode nx*) (*RefNode x*) |

negate-sub:
⟦*kind g sub* = (*SubNode x y*);
stamp g sub = (*IntegerStamp - - -*)⟧
⟹ *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 = *intval-not val*⟧
⟹ *CanonicalizeNot g* (*NotNode nx*) (*ConstantNode neg-val*) |

not-not:
⟦*kind g nx* = (*NotNode x*)⟧
⟹ *CanonicalizeNot g* (*NotNode nx*) (*RefNode x*)

inductive *CanonicalizeLogicNegation* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
logical-not-const:
⟦*kind g nx* = (*ConstantNode val*);
neg-val = *bool-to-val* (¬(*val-to-bool val*))⟧
⟹ *CanonicalizeLogicNegation g* (*LogicNegationNode nx*) (*ConstantNode neg-val*)
|

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

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

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

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

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

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

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

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

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

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

or-yfalse:
$[\![$*kind g y* = *ConstantNode val*;
¬(*val-to-bool val*)$]\!]$
⟹ *CanonicalizeOr g* (*OrNode x y*) (*RefNode x*)

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

de-morgan-or-to-and:
$[\![$*kind g nid* = *OrNode nx ny*;
kind g nx = *NotNode x*;
kind g ny = *NotNode y*;
new-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′*$]\!]$
⟹ *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′*$]\!]$
⟹ *CanonicalizeDeMorgansLaw nid g g″*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

inductive *CanonicalizeIntegerEqualsGraph* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
where
int-equals-rewrite:
$[\![CanonicalizeIntegerEquals\ g\ node\ node';$
$node = kind\ g\ nid;$

$g' = replace\text{-}node\ nid\ (node', stamp\ g\ nid)\ g$ ⟧
$\Longrightarrow$ *CanonicalizeIntegerEqualsGraph nid g g'* |

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

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

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

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

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

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

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

IfNode:

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

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

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

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

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

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

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

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

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

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

type-synonym *CanonicalizationAnalysis* = *bool option*

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

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

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

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

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

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

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

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

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

code-pred (*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *bool*) *CanonicalizationPhase* **.**

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

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

$CanonicalizationStep\ g\ (kind\ g\ nid)\ node;$

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

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

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

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

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

$\llbracket Step\ analyse\ g\ (nid,\ seen,\ i)\ None;$
$None = pred\ g\ nid \rrbracket$
$\Longrightarrow CanonicalizationPhaseWithTrace\ g\ (nid,\ seen,\ i)\ g\ t\ t$

code-pred $(modes\colon i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow bool)\ CanonicalizationPhaseWithTrace$
.

end