Veriopt

April 24, 2021

Abstract

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

Contents

1 Runtime Values and Arithmetic

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

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

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

type-synonym *objref* = *nat option*

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

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

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

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

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

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

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

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

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

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

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



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 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
    Values
begin

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

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

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

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

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

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

| *FrameState* (*ir-monitorIds*: *INPUT-ASSOC list*) (*ir-outerFrameState-opt*: *INPUT-STATE option*) (*ir-values-opt*: *INPUT list option*) (*ir-virtualObjectMappings-opt*: *INPUT-STATE list option*)
| *IfNode* (*ir-condition*: *INPUT-COND*) (*ir-trueSuccessor*: *SUCC*) (*ir-falseSuccessor*: *SUCC*)
| *IntegerEqualsNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *IntegerLessThanNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *InvokeNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *IN-*

PUT-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 *IRNodes*
begin

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

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

These functions have been automatically generated from the compiler.

fun *is-EndNode* :: *IRNode* $\Rightarrow$ *bool* **where**
is-EndNode EndNode = *True* |
is-EndNode - = *False*

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

fun *is-AbstractMergeNode* :: *IRNode* $\Rightarrow$ *bool* **where**
is-AbstractMergeNode n = ((*is-LoopBeginNode n*) $\vee$ (*is-MergeNode n*))

fun *is-BeginStateSplitNode* :: *IRNode* $\Rightarrow$ *bool* **where**
is-BeginStateSplitNode n = ((*is-AbstractMergeNode n*) $\vee$ (*is-ExceptionObjectNode n*) $\vee$ (*is-LoopExitNode n*) $\vee$ (*is-StartNode n*))

fun *is-AbstractBeginNode* :: *IRNode* $\Rightarrow$ *bool* **where**
is-AbstractBeginNode n = ((*is-BeginNode n*) $\vee$ (*is-BeginStateSplitNode n*) $\vee$ (*is-KillingBeginNode n*))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$n) \vee (\textit{is-UnaryOpLogicNode } n) \vee (\textit{is-UnwindNode } n))$

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

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

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

fun *is-VirtualizableAllocation* :: *IRNode* ⇒ *bool* **where**
$\textit{is-VirtualizableAllocation } n = ((\textit{is-NewArrayNode } n) \vee (\textit{is-NewInstanceNode } n))$

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

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

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

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

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

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

fun *is-ArithmeticOperation* :: *IRNode* ⇒ *bool* **where**
$\textit{is-ArithmeticOperation } n = ((\textit{is-BinaryArithmeticNode } n) \vee (\textit{is-UnaryArithmeticNode } n))$

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

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

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*)) ∨

$((is\text{-}LogicNegationNode\ n1) \wedge (is\text{-}LogicNegationNode\ n2)) \vee$
$((is\text{-}LoopBeginNode\ n1) \wedge (is\text{-}LoopBeginNode\ n2)) \vee$
$((is\text{-}LoopEndNode\ n1) \wedge (is\text{-}LoopEndNode\ n2)) \vee$
$((is\text{-}LoopExitNode\ n1) \wedge (is\text{-}LoopExitNode\ n2)) \vee$
$((is\text{-}MergeNode\ n1) \wedge (is\text{-}MergeNode\ n2)) \vee$
$((is\text{-}MethodCallTargetNode\ n1) \wedge (is\text{-}MethodCallTargetNode\ n2)) \vee$
$((is\text{-}MulNode\ n1) \wedge (is\text{-}MulNode\ n2)) \vee$
$((is\text{-}NegateNode\ n1) \wedge (is\text{-}NegateNode\ n2)) \vee$
$((is\text{-}NewArrayNode\ n1) \wedge (is\text{-}NewArrayNode\ n2)) \vee$
$((is\text{-}NewInstanceNode\ n1) \wedge (is\text{-}NewInstanceNode\ n2)) \vee$
$((is\text{-}NotNode\ n1) \wedge (is\text{-}NotNode\ n2)) \vee$
$((is\text{-}OrNode\ n1) \wedge (is\text{-}OrNode\ n2)) \vee$
$((is\text{-}ParameterNode\ n1) \wedge (is\text{-}ParameterNode\ n2)) \vee$
$((is\text{-}PiNode\ n1) \wedge (is\text{-}PiNode\ n2)) \vee$
$((is\text{-}ReturnNode\ n1) \wedge (is\text{-}ReturnNode\ n2)) \vee$
$((is\text{-}ShortCircuitOrNode\ n1) \wedge (is\text{-}ShortCircuitOrNode\ n2)) \vee$
$((is\text{-}SignedDivNode\ n1) \wedge (is\text{-}SignedDivNode\ n2)) \vee$
$((is\text{-}StartNode\ n1) \wedge (is\text{-}StartNode\ n2)) \vee$
$((is\text{-}StoreFieldNode\ n1) \wedge (is\text{-}StoreFieldNode\ n2)) \vee$
$((is\text{-}SubNode\ n1) \wedge (is\text{-}SubNode\ n2)) \vee$
$((is\text{-}UnwindNode\ n1) \wedge (is\text{-}UnwindNode\ n2)) \vee$
$((is\text{-}ValuePhiNode\ n1) \wedge (is\text{-}ValuePhiNode\ n2)) \vee$
$((is\text{-}ValueProxyNode\ n1) \wedge (is\text{-}ValueProxyNode\ n2)) \vee$
$((is\text{-}XorNode\ n1) \wedge (is\text{-}XorNode\ n2)))$

end

3 Stamp Typing

theory *Stamp*
 imports *Values*
begin

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

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

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

 | *KlassPointerStamp* (*stp-nonNull*: *bool*) (*stp-alwaysNull*: *bool*)
 | *MethodCountersPointerStamp* (*stp-nonNull*: *bool*) (*stp-alwaysNull*: *bool*)
 | *MethodPointersStamp* (*stp-nonNull*: *bool*) (*stp-alwaysNull*: *bool*)

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

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

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

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

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

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

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

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

is-stamp-empty x = *False*

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

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

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

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

— In certain circumstances a stamp provides enough information to evaluate a

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

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

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

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

constantAsStamp - = *IllegalStamp*

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

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

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

lemma *int-valid-range*:
assumes *stamp* = *IntegerStamp bits lower upper*
shows {*x* . *valid-value stamp x*} = {(*IntVal bits val*) | *val* . *val* ∈ {*lower..upper*}}
using *assms valid-value.simps* **apply** *auto*
using *valid-value.elims*(*2*) **by** *blast*

lemma *disjoint-empty*:
assumes *joined* = (*join x-stamp y-stamp*)
assumes *is-stamp-empty joined*
shows {*x* . *valid-value x-stamp x*} ∩ {*y* . *valid-value y-stamp y*} = {}
using *assms int-valid-range*
by (*induction x-stamp*; *induction y-stamp*; *auto*)

lemma *join-unequal*:
assumes *joined* = (*join x-stamp y-stamp*)
assumes *is-stamp-empty joined*
shows $\nexists$ *x y* . *x* = *y* ∧ *valid-value x-stamp x* ∧ *valid-value y-stamp y*
using *assms disjoint-empty* **by** *auto*

lemma *neverDistinctEqual*:
assumes *neverDistinct x-stamp y-stamp*
shows $\nexists$ *x y* . *x* ≠ *y* ∧ *valid-value x-stamp x* ∧ *valid-value y-stamp y*
using *assms*
by (*smt* (*verit*, *best*) *asConstant.simps*(*1*) *asConstant.simps*(*2*) *asConstant.simps*(*3*) *neverDistinct.elims*(*2*) *valid-value.elims*(*2*))

lemma *boundsNoOverlapNoEqual*:
assumes *stpi-upper x-stamp* < *stpi-lower y-stamp*
assumes *is-IntegerStamp x-stamp* ∧ *is-IntegerStamp y-stamp*
shows $\nexists$ *x y* . *x* = *y* ∧ *valid-value x-stamp x* ∧ *valid-value y-stamp y*
using *assms* **apply** (*cases x-stamp*; *auto*)
using *int-valid-range*
by (*smt* (*verit*, *ccfv-threshold*) *Stamp.collapse*(*1*) *mem-Collect-eq valid-value.simps*(*1*))

lemma *boundsNoOverlap*:
assumes *stpi-upper x-stamp* < *stpi-lower y-stamp*
assumes *x* = *IntVal b1 xval*
assumes *y* = *IntVal b2 yval*
assumes *is-IntegerStamp x-stamp* ∧ *is-IntegerStamp y-stamp*
assumes *valid-value x-stamp x* ∧ *valid-value y-stamp y*
shows *xval* < *yval*
using *assms is-IntegerStamp-def* **by** *force*

lemma *boundsAlwaysOverlap*:
assumes *stpi-lower x-stamp* ≥ *stpi-upper y-stamp*
assumes *x* = *IntVal b1 xval*
assumes *y* = *IntVal b2 yval*
assumes *is-IntegerStamp x-stamp* ∧ *is-IntegerStamp y-stamp*
assumes *valid-value x-stamp x* ∧ *valid-value y-stamp y*
shows ¬(*xval* < *yval*)
using *assms is-IntegerStamp-def*
by *fastforce*

lemma *intstamp-bits-eq-meet*:
assumes (*meet* (*IntegerStamp b1 l1 u1*) (*IntegerStamp b2 l2 u2*)) = (*IntegerStamp b3 l3 u3*)
shows *b1* = *b3* ∧ *b2* = *b3*
by (*metis Stamp.distinct*(*25*) *assms meet.simps*(*2*))

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

b3 l3 u3)
  shows *b1* = *b3* ∧ *b2* = *b3*
  by (*metis Stamp.distinct*(*25*) *assms join.simps*(*2*))

lemma *intstamp-bites-eq-unrestricted*:
  assumes (*unrestricted-stamp* (*IntegerStamp b1 l1 u1*)) = (*IntegerStamp b2 l2 u2*)
  shows *b1* = *b2*
  using *assms* **by** *auto*

lemma *intstamp-bits-eq-empty*:
  assumes (*empty-stamp* (*IntegerStamp b1 l1 u1*)) = (*IntegerStamp b2 l2 u2*)
  shows *b1* = *b2*
  using *assms* **by** *auto*

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
Stamp
HOL−Library.FSet
HOL.Relation
begin

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

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

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

setup-lifting *type-definition-IRGraph*

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

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

lift-definition $kind :: IRGraph \Rightarrow (ID \Rightarrow IRNode)$
is $with\text{-}default\ NoNode\ fst$ **.**

lift-definition $stamp :: IRGraph \Rightarrow ID \Rightarrow Stamp$
is $with\text{-}default\ IllegalStamp\ snd$ **.**

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

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

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

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

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

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

code-datatype *irgraph*

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

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

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

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

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

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

— Get the inputs set of a given node ID
fun *inputs* :: *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *ID set* **where**
inputs g nid = *set* (*inputs-of* (*kind g nid*))
— Get the successor set of a given node ID
fun *succ* :: *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *ID set* **where**
succ g nid = *set* (*successors-of* (*kind g nid*))
— Gives a relation between node IDs - between a node and its input nodes
fun *input-edges* :: *IRGraph* $\Rightarrow$ *ID rel* **where**
input-edges g = $(\bigcup\ i \in ids\ g.\ \{(i,j)|j.\ j \in (inputs\ g\ i)\})$
— Find all the nodes in the graph that have nid as an input - the usages of nid
fun *usages* :: *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *ID set* **where**
usages g nid = $\{j.\ j \in ids\ g \wedge (j,nid) \in input\text{-}edges\ g\}$
fun *successor-edges* :: *IRGraph* $\Rightarrow$ *ID rel* **where**
successor-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* ∈ *ids g* ∧ (*j*,*nid*) ∈ *successor-edges g*}
fun *nodes-of* :: *IRGraph* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID set* **where**
nodes-of g sel = {*nid* ∈ *ids g* . *sel* (*kind g nid*)}
fun *edge* :: (*IRNode* ⇒ *'a*) ⇒ *ID* ⇒ *IRGraph* ⇒ *'a* **where**
edge sel nid g = *sel* (*kind g nid*)

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

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

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

lemma *ids-some*[*simp*]: *x* ∈ *ids g* ⟷ *kind g x* ≠ *NoNode*
proof −
have *that*: *x* ∈ *ids g* ⟶ *kind g x* ≠ *NoNode*
using *ids.rep-eq kind.rep-eq* **by** *force*
have *kind g x* ≠ *NoNode* ⟶ *x* ∈ *ids g*
unfolding *with-default.simps kind-def ids-def*
by (*cases Rep-IRGraph g x* = *None*; *auto*)
from *this that* **show** *?thesis* **by** *auto*
qed

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

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

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

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

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

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

lemma [*simp*]: *finite* (*dom g*) $\longrightarrow$ *stamp* (*Abs-IRGraph g*) = (λx . (*case g x of None* $\Rightarrow$ *IllegalStamp* | *Some n* $\Rightarrow$ *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* $\Rightarrow$ *NoNode* | *Some n* $\Rightarrow$ *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*:
  $gup = remove\text{-}node\ nid\ g \longrightarrow kind\ gup\ nid = NoNode \wedge stamp\ gup\ nid = IllegalStamp$
  by (*simp add*: *kind.rep-eq remove-node.rep-eq stamp.rep-eq*)

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

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

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

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

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

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

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

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

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

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

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

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

fun *phi-list* :: *IRGraph* ⇒ *ID* ⇒ *ID list* **where**
  phi-list g nid =
    (*filter* (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-set nid v m*)) |
  set-phis [] (*v* # *vs*) *m* = *m* |
  set-phis (*x* # *xs*) [] *m* = *m*

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

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

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

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

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

— Unary arithmetic operators

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

NegateNode:
$[\![g\ m \vdash (kind\ g\ x) \mapsto v]\!]$
$\Longrightarrow g\ m \vdash (NegateNode\ x) \mapsto (IntVal\ (v\text{-}bits\ v)\ 0) - v\ |$

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

— Binary arithmetic operators

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

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

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

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

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

— Binary logical bitwise operators

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

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

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

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

— Comparison operators

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

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

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

— Other nodes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

fun *is-floating-node* :: *IRNode* $\Rightarrow$ *bool* **where**
is-floating-node n = ($\neg$(*has-control-flow n*))

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

lemma *is-floating-node n* $\longleftrightarrow$ $\neg$(*has-control-flow n*)
by *simp*

lemma $n \in$ *control-nodes* $\longleftrightarrow$ $n \notin$ *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* $\vdash$ (*AddNode x y*) $\mapsto$ *val* $\Longrightarrow$
($\exists$ *v1*. (*g m* $\vdash$ (*kind g x*) $\mapsto$ *v1*) $\land$
($\exists$ *v2*. (*g m* $\vdash$ (*kind g y*) $\mapsto$ *v2*) $\land$
val = *intval-add v1 v2*))
using *AddNodeE plus-Value-def* **by** *metis*

lemma *not-floating*: ($\exists$ *y ys*. (*successors-of n*) = *y* # *ys*) $\longrightarrow$ $\neg$(*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* $\vdash$ *node* $\mapsto$ *val1*) $\Longrightarrow$
($\forall$ *val2*. ((*g m* $\vdash$ *node* $\mapsto$ *val2*) $\longrightarrow$ *val1* = *val2*))
apply (*induction rule*: *eval.induct*)
by (*rule allI*; *rule impI*; *elim EvalE*; *auto*)+

theorem *evalAllDet*:
  $(g\ m \vdash nodes \longmapsto vals1) \Longrightarrow$
  $(\forall\ vals2.\ ((g\ m \vdash nodes \longmapsto vals2) \longrightarrow vals1 = vals2))$
  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**
  $h\text{-}load\text{-}field\ f\ r\ (h,\ n) = h\ f\ r$

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

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

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

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

6.2 Intraprocedural Semantics

Intraprocedural semantics are given as a small-step semantics.

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

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

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

IfNode:
⟦*kind g nid* = (*IfNode cond tb fb*);
g m ⊢ (*kind g cond*) ↦ *val*;
nid′ = (*if val-to-bool val then tb else fb*)⟧
⟹ *g* ⊢ (*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 ⊢ *inps* ⟼ *vs*;

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

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

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

SignedDivNode:
⟦*kind g nid* = (*SignedDivNode nid x y zero sb nxt*);
g m ⊢ (*kind g x*) ↦ *v1*;
g m ⊢ (*kind g y*) ↦ *v2*;
v = (*intval-div v1 v2*);
m′ = *m-set nid v m*⟧

$\Longrightarrow g \vdash (nid,\ m,\ h) \rightarrow (nxt,\ m',\ h)$ |

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

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

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

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

code-pred (*modes*: $i \Rightarrow i * i * i \Rightarrow o * o * o \Rightarrow bool$) *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*:
$(g \vdash (nid,m,h) \rightarrow next) \Longrightarrow$
$(\forall\ next'.\ ((g \vdash (nid,m,h) \rightarrow next') \longrightarrow next = next'))$
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*) *IRNode.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 IRNode.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*(*1367*) *IRNode.distinct*(*893*) *IRNode.inject*(*18*) *Pair-inject Value.inject*(*3*) *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*(*2027*) *IRNode.distinct*(*933*) *IRNode.distinct*(*1315*) *IRNode.distinct*(*1725*) *IRNode.distinct*(*1937*) *IRNode.distinct*(*909*) *IRNode.inject*(*38*) *Pair-inject Value.inject*(*3*) *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′*⟧ ⟹ *g* ⊢ (*nid*,*m*,*h*) → (*nid′*,*m*,*h*)
by (*simp add*: *SequentialNode*)

lemma *IfNodeStepCases*:
assumes *kind g nid* = *IfNode cond tb fb*
assumes *g m* ⊢ *kind g cond* ↦ *v*
assumes *g* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*)

shows $nid' \in \{tb,\ fb\}$
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 $g \vdash (nid,\ m,\ h) \to (nid',\ m,\ h)$
shows $\exists\ v.\ (g\ m \vdash kind\ g\ cond \mapsto v)$
using *assms*(*2*,*1*) **by** (*induct* (*nid*,*m*,*h*) (*nid'*,*m*,*h*) *rule*: *step.induct*; *auto*)

lemma *step-in-ids*:
assumes $g \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')$
shows $nid \in ids\ g$
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*) *list* × *FieldRefHeap*
⇒ (*IRGraph* × *ID* × *MapState*) *list* × *FieldRefHeap* ⇒ *bool*
(- ⊢ - ⟶ - *55*)
for *p* **where**

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

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

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

$g\ m \vdash arguments \longmapsto vs$;
$m' = set\text{-}params\ m\ vs]\!]$

$\Longrightarrow p \vdash ((g,nid,m)\#stk,\ h) \longrightarrow ((targetGraph,0,m')\#(g,nid,m)\#stk,\ h)$ |

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

$c\text{-}m' = m\text{-}set\ c\text{-}nid\ v\ c\text{-}m;$
$c\text{-}nid' = (successors\text{-}of\ (kind\ c\text{-}g\ c\text{-}nid))!0]\!]$
$\Longrightarrow p \vdash ((g,nid,m)\#(c\text{-}g,c\text{-}nid,c\text{-}m)\#stk,\ h) \longrightarrow ((c\text{-}g,c\text{-}nid',c\text{-}m')\#stk,\ h)$ |

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

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

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

$g\ m \vdash (kind\ g\ exception) \mapsto e;$

$kind\ c\text{-}g\ c\text{-}nid = (InvokeWithExceptionNode\ \text{- - - - - -}\ exEdge);$

$c\text{-}m' = m\text{-}set\ c\text{-}nid\ e\ c\text{-}m]\!]$
$\Longrightarrow p \vdash ((g,nid,m)\#(c\text{-}g,c\text{-}nid,c\text{-}m)\#stk,\ h) \longrightarrow ((c\text{-}g,exEdge,c\text{-}m')\#stk,\ h)$

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

6.4 Big-step Execution

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

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

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

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

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

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

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

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow$ *bool as Exec*) *exec* **.**

inductive *exec-debug* :: *Program*
$\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState*) *list* $\times$ *FieldRefHeap*
$\Rightarrow$ *nat*
$\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState*) *list* $\times$ *FieldRefHeap*
$\Rightarrow$ *bool*
(-⊢-→∗-∗ -)
where
⟦$n > 0$;
$p \vdash s \longrightarrow s'$;
exec-debug p s' $(n - 1)$ s''⟧
$\Longrightarrow$ *exec-debug p s n* s'' |

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

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

6.4.1 Heap Testing

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

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

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

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

]

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

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

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

7 Proof Infrastructure

7.1 Bisimulation

theory *Bisimulation*
imports
Stuttering
begin

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

A strong bisimilution between no-op transitions

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

lemma *lockstep-strong-bisimilulation*:

assumes $g' = $ *replace-node nid node g*
assumes $g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)$
assumes $g' \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)$
shows $nid \mid g \sim g'$
using *assms(2) assms(3) stepDet strong-noop-bisimilar.simps* **by** *blast*

lemma *no-step-bisimulation*:
assumes $\forall\, m\ h\ nid'\ m'\ h'.\ \neg(g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h'))$
assumes $\forall\, m\ h\ nid'\ m'\ h'.\ \neg(g' \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h'))$
shows $nid \mid g \sim g'$
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 $= (0 \in$ *ids g* $\wedge$
is-StartNode (*kind g 0*))

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

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

definition *wf-ends* **where**
wf-ends g $=$
$(\forall\ n \in$ *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 $g = (\forall\ n \in ids\ g\ .$
$(\forall\ v\ m\ .\ (g\ m \vdash (kind\ g\ n) \mapsto v) \longrightarrow valid\text{-}value\ (stamp\ g\ n)\ v))$

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

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

lemma *wf-value-range*:
$b > 1 \land b \in int\text{-}bits\text{-}allowed \longrightarrow \{v.\ wf\text{-}value\ (IntVal\ b\ v)\} = \{v.\ ((-(2\hat{}(b-1))$
$\leq v) \land (v < (2\hat{}(b-1))))\}$
unfolding *wf-value.simps*
by *auto*

lemma *wf-value-bit-range*:
$b = 1 \longrightarrow \{v.\ wf\text{-}value\ (IntVal\ b\ v)\} = \{\}$
unfolding *wf-value.simps*
by (*simp add*: *int-bits-allowed-def*)

end

7.3 Dynamic Frames

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

theory *IRGraphFrames*

imports
Form
Semantics.IREval
begin

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

fun *changeonly* :: *ID set* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
changeonly ns g1 g2 = (∀ *n* . *n* ∈ *ids g1* ∧ *n* ∉ *ns* ⟶
(*n* ∈ *ids g1* ∧ *n* ∈ *ids g2* ∧ *kind g1 n* = *kind g2 n*))

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

lemma *other-node-unchanged*:
assumes *changeonly ns g1 g2*
assumes *nid* ∈ *ids g1*
assumes *nid* ∉ *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* ⇒ *ID* ⇒ *ID* ⇒ *bool*
for *g* **where**

use0: *nid* ∈ *ids g*
⟹ *eval-uses g nid nid* |

use-inp: *nid′* ∈ *inputs g n*
⟹ *eval-uses g nid nid′* |

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

fun *eval-usages* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
eval-usages g nid = {*n* ∈ *ids g* . *eval-uses g nid n*}

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

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* $\neq$ *NoNode*
assumes *List.member* (*inputs-of n*) *child*
shows *child* $\in$ *inputs g nid*
unfolding *inputs.simps* **using** *assms*
by (*metis in-set-member*)

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

lemma *inp-in-g*:
assumes *n* $\in$ *inputs g nid*
shows *nid* $\in$ *ids g*
proof −
have *inputs g nid* $\neq$ {}
using *assms*
by (*metis empty-iff empty-set*)
then have *kind g nid* $\neq$ *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* $\in$ *inputs g nid*
shows *n* $\in$ *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* ∈ *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-usages g nid*
  assumes *nid′* ∈ *ids g*
  shows *eval-uses g nid nid′* ⟷ *nid′* ∈ *us* (**is** *?P* ⟷ *?Q*)
  using *assms eval-usages.simps*
  by (*simp add*: *ids.rep-eq*)

lemma *inputs-are-uses*:
  assumes *nid′* ∈ *inputs g nid*
  shows *eval-uses g nid nid′*
  by (*metis assms use-inp*)

lemma *inputs-are-usages*:
  assumes *nid′* ∈ *inputs g nid*
  assumes *nid′* ∈ *ids g*
  shows *nid′* ∈ *eval-usages g nid*
  using *assms*(*1*) *assms*(*2*) *eval-usages inputs-are-uses* **by** *blast*

lemma *usage-includes-inputs*:
  assumes *us* = *eval-usages g nid*
  assumes *ls* = *inputs g nid*
  assumes *ls* ⊆ *ids g*
  shows *ls* ⊆ *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* ≠ *NoNode*
  assumes *child* ∈ *set* (*inputs-of k*)

shows *child* ∈ *inputs g nid*
using *assms* **by** *auto*

lemma *eval-in-ids*:
assumes *g m* ⊢ (*kind g nid*) ↦ *v*
shows *nid* ∈ *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* ⊢ (*kind g1 nid*) ↦ *v1*
assumes *wf*: *wf-graph g1*
shows *g2 m* ⊢ (*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 m* (*kind g1 nid*) *v1 arbitrary*: *nid rule*: *eval.induct*)
print-cases
case *const*: (*ConstantNode m 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 m i*)
show *?case*
by (*metis eval.ParameterNode kind-unchanged param.hyps*(*1*) *param.prems*(*1*) *param.prems*(*2*))
next
case (*ValuePhiNode val nida ux uy*)
then have *kind*: (*kind g2 nid*) = *ValuePhiNode nida ux uy*
using *kind-unchanged* **by** *metis*
then show *?case*
using *eval.ValuePhiNode kind ValuePhiNode.hyps*(*1*) **by** *metis*
next
case (*ValueProxyNode m 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* ⊢ (*kind g2 child*) ↦ *val*
using *ValueProxyNode.hyps*(*2*) *cin*
by *blast*
then show *?case*
by (*metis ValueProxyNode.hyps*(*3*) *ValueProxyNode.prems*(*1*) *ValueProxyNode.prems*(*2*) *eval.ValueProxyNode kind-unchanged*)
next
case (*AbsNode m x b 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* ⊢ (*kind g2 x*) ↦ *IntVal b 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 m 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* ⊢ (*kind g1 x*) ↦ *v*
using *Node.hyps*(*1*) *Node.hyps*(*2*)
by *blast*
have *x2*: *g2 m* ⊢ (*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 m 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* ⊢ (*kind g1 x*) ↦ *v1*
using *node.hyps*(*1*) **by** *blast*
have *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis IRNodes.inputs-of-AddNode child-member-in child-unchanged member-rec*(*1*) *node.hyps*(*5*) *node.prems*(*1*) *node.prems*(*2*))
have *y*: *g1 m* ⊢ (*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 m 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* ⊢ (*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* ⊢ (*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 m 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* ⊢ (*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* ⊢ (*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 m 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* ⊢ (*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* ⊢ (*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 m 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* ⊢ (*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* ⊢ (*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 m 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* ⊢ (*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* ⊢ (*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 m x b 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* ⊢ (*kind g1 x*) ↦ *IntVal b 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* ⊢ (*kind g1 y*) ↦ *IntVal b 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 m x b 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* ⊢ (*kind g1 x*) ↦ *IntVal b 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* ⊢ (*kind g1 y*) ↦ *IntVal b 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 subsetD wf wf-folds*(*1,3*))
next
case *node*: (*ShortCircuitOrNode m x b 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* ⊢ (*kind g1 x*) ↦ *IntVal b 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* ⊢ (*kind g1 y*) ↦ *IntVal b v2*
using *node.hyps*(*3*) **by** *blast*
have *x2*: *g2 m* ⊢ (*kind g2 x*) ↦ *IntVal b 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* ⊢ (*kind g2 y*) ↦ *IntVal b 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 m 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* ⊢ (*kind g2 x*) ↦ *IntVal 1 v1*
by (*metis inputs.simps inp-in-g-wf inputs-of-LogicNegationNode list.set-intros*(*1*) *node.hyps*(*2*) *node.hyps*(*4*) *wf*)
then show *?case*
by (*metis LogicNegationNode kind-unchanged node.hyps*(*3*) *node.hyps*(*4*) *node.prems*(*1*) *node.prems*(*2*))
next
case *node*:(*ConditionalNode m condition cond trueExp b trueVal falseExp falseVal val*)
have *c*: *condition* ∈ *inputs g1 nid*
by (*metis IRNodes.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* ⊢ (*kind g2 condition*) ↦ *IntVal 1 cond*

using *node c inp-in-g-wf wf* **by** *blast*

have *t*: *trueExp* ∈ *inputs g1 nid*
by (*metis IRNodes.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* ⊢ (*kind g2 trueExp*) ↦ *IntVal b* (*trueVal*)
using *node.hyps node t inp-in-g-wf wf* **by** *blast*

have *f*: *falseExp* ∈ *inputs g1 nid*
by (*metis IRNodes.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* ⊢ (*kind g2 falseExp*) ↦ *IntVal b* (*falseVal*)
using *node.hyps node f inp-in-g-wf wf* **by** *blast*

have *g2 m* ⊢ (*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 m x val nid*)
have *x*: *x* ∈ *inputs g1 nid*
by (*metis IRNodes.inputs-of-RefNode RefNode.hyps*(*3*) *RefNode.prems*(*1*) *child-member-in member-rec*(*1*))
then have *ref*: *g2 m* ⊢ (*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 m - callTarget classInit stateDuring stateAfter nex*)
then show *?case*
by (*metis eval.InvokeNodeEval kind-unchanged*)
next
case (*SignedDivNode m x v1 y v2 zeroCheck frameState nex*)
then show *?case*
by (*metis eval.SignedDivNode kind-unchanged*)
next
case (*SignedRemNode m x v1 y v2 zeroCheck frameState nex*)
then show *?case*
by (*metis eval.SignedRemNode kind-unchanged*)
next

case (*InvokeWithExceptionNodeEval val m - callTarget classInit stateDuring stateAfter nex exceptionEdge*)
then show *?case*
by (*metis eval.InvokeWithExceptionNodeEval kind-unchanged*)
next
case (*NewInstanceNode m nid clazz stateBefore nex*)
then show *?case*
by (*metis eval.NewInstanceNode kind-unchanged*)
next
case (*IsNullNode m obj ref val*)
have *obj*: *obj* ∈ *inputs g1 nid*
by (*metis IRNodes.inputs-of-IsNullNode IsNullNode.hyps*(*4*) *inputs.simps list.set-intros*(*1*))
then have *ref*: *g2 m* ⊢ (*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 m object val*)
have *object*: *object* ∈ *inputs g1 nid*
using *inputs-of-PiNode inputs.simps*
by (*metis PiNode.hyps*(*3*) *append-Cons list.set-intros*(*1*))
then have *ref*: *g2 m* ⊢ (*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 m 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* ⊢ (*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 $gup = add\text{-}node\ new\ k\ g$
  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 $gup = add\text{-}node\ new\ k\ g$
  assumes *wf-graph g*
  shows *unchanged* (*eval-usages g nid*) *g gup*
proof −
  have $new \notin (eval\text{-}usages\ g\ nid)$ **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*:
  $((nid' \in ids\ g \wedge nid = nid')$
    $\vee\ nid' \in inputs\ g\ nid$
    $\vee\ (\exists\, nid''\ .\ eval\text{-}uses\ g\ nid\ nid'' \wedge eval\text{-}uses\ g\ nid''\ nid'))$
    $\longleftrightarrow eval\text{-}uses\ g\ nid\ nid'$
  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 $nid' \notin ids\ g$
  assumes $nid \in ids\ g$
  shows $\neg(eval\text{-}uses\ g\ nid\ nid')$
proof –
  have *0*: $nid \neq nid'$
    using *assms* **by** *blast*
  have *inp*: $nid' \notin inputs\ g\ nid$
    using *assms*
    using *inp-in-g-wf* **by** *blast*
  have *rec-0*: $\nexists n\ .\ n \in ids\ g \wedge n = nid'$
    using *assms* **by** *blast*
  have *rec-inp*: $\nexists n\ .\ n \in ids\ g \wedge n \in inputs\ g\ nid'$
    using *assms*(*2*) *inp-in-g* **by** *blast*
  have *rec*: $\nexists nid''\ .\ eval\text{-}uses\ g\ nid\ nid'' \wedge eval\text{-}uses\ g\ nid''\ nid'$
    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* :: $ID \Rightarrow ID \Rightarrow IRGraph \Rightarrow IRGraph$ **where**
  $replace\text{-}usages\ nid\ nid'\ g = replace\text{-}node\ nid\ (RefNode\ nid',\ stamp\ g\ nid')\ g$

lemma *replace-usages-effect*:
  assumes $g' = replace\text{-}usages\ nid\ nid'\ g$
  shows $kind\ g'\ nid = RefNode\ nid'$
  using *assms replace-node-lookup replace-usages.simps IRNode.distinct*(*2069*)
  by (*metis*)

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

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* $\in$ *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* $\Rightarrow$ *ID* **where**
nextNid g = (*Max* (*ids g*)) + *1*

lemma *max-plus-one*:
fixes *c* :: *ID set*
shows $[\![$*finite c*; *c* $\neq$ $\{\}]\!]$ $\Longrightarrow$ (*Max c*) + *1* $\notin$ *c*
by (*meson Max-gr-iff less-add-one less-irrefl*)

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

lemma *nextNidNotIn*:
ids g $\neq$ $\{\}$ $\longrightarrow$ *nextNid g* $\notin$ *ids g*
unfolding *nextNid.simps*
using *ids-finite max-plus-one* **by** *blast*

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

lemma *constantConditionTrue*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes *g'* = *constantCondition True ifcond* (*kind g ifcond*) *g*
shows *g'* $\vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*t*, *m*, *h*)
proof $-$
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* = (*IntVal 1 1*)
by *auto*
have *ifcond* $\neq$ (*nextNid g*)
by (*metis IRNode.simps*(*989*) *assms*(*1*) *emptyE ids-some nextNidNotIn*)
then have *c'*: *kind g'* (*nextNid g*) = *ConstantNode* (*IntVal 1 1*)

using *assms*(*2*) *replace-node-unchanged*
by (*metis DiffI IRNode.distinct*(*585*) ‹*bool-to-val True* = *IntVal 1 1*› *add-node-lookup assms*(*1*) *constantCondition.simps*(*1*) *emptyE insertE not-in-g*)
from *if*′ *c*′ **show** *?thesis* **using** *IfNode*
by (*smt* (*z3*) *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*′ ⊢ (*ifcond*, *m*, *h*) → (*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* = (*IntVal 1 0*)
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* (*IntVal 1 0*)
using *assms*(*2*) *replace-node-unchanged*
by (*metis DiffI IRNode.distinct*(*585*) ‹*bool-to-val False* = *IntVal 1 0*› *add-node-lookup assms*(*1*) *constantCondition.simps*(*1*) *emptyE insertE not-in-g*)
from *if*′ *c*′ **show** *?thesis* **using** *IfNode*
by (*smt* (*z3*) *ConstantNode val-to-bool.simps*(*1*))
qed

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

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 ¬(*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*)), *default-stamp*) *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 $\exists\, nid'\ .(g\ m\ h \vdash ifcond \leadsto nid') \longleftrightarrow (g'\ m\ h \vdash ifcond \leadsto nid')$
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 $g\ m \vdash kind\ g\ cond \mapsto v$
assumes $const = val\text{-}to\text{-}bool\ v$
assumes $g' = constantCondition\ const\ ifcond\ (kind\ g\ ifcond)\ g$
shows $\exists\, nid'\ .(g\ m\ h \vdash ifcond \rightsquigarrow nid') \longleftrightarrow (g'\ m\ h \vdash ifcond \rightsquigarrow nid')$
proof (*cases const*)
case *True*
have *ifstep*: $g \vdash (ifcond,\ m,\ h) \rightarrow (t,\ m,\ h)$
by (*meson IfNode True assms*(*1*) *assms*(*2*) *assms*(*3*))
have *ifstep'*: $g' \vdash (ifcond,\ m,\ h) \rightarrow (t,\ m,\ h)$
using *constantConditionTrue*
using *True assms*(*1*) *assms*(*4*) **by** *presburger*
from *ifstep ifstep'* **show** *?thesis*
using *StutterStep* **by** *blast*
next
case *False*
have *ifstep*: $g \vdash (ifcond,\ m,\ h) \rightarrow (f,\ m,\ h)$
by (*meson IfNode False assms*(*1*) *assms*(*2*) *assms*(*3*))
have *ifstep'*: $g' \vdash (ifcond,\ m,\ h) \rightarrow (f,\ m,\ h)$
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* ⇒ *MapState* ⇒ *FieldRefHeap* ⇒ *ID* ⇒ *ID* ⇒ *bool* (-
- - ⊢ - ⇝ - *55*)
for *g m h* **where**

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

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

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

proof −
  have *nextin*: $nid' \in \{P'. (g\ m\ h \vdash nid \rightsquigarrow P')\}$
    using *assms StutterStep* **by** *blast*
  have *nextsubset*: $\{nid''. (g\ m\ h \vdash nid' \rightsquigarrow nid'')\} \subseteq \{P'. (g\ m\ h \vdash nid \rightsquigarrow P')\}$
    by (*metis Collect-mono assms stutter.Transitive*)
  have $\forall\, n \in \{P'. (g\ m\ h \vdash nid \rightsquigarrow P')\}\ .\ n = nid' \lor n \in \{nid''. (g\ m\ h \vdash nid' \rightsquigarrow nid'')\}$
    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* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
add-both-const:
$[\![kind\ g\ x = ConstantNode\ c\text{-}1;$
$kind\ g\ y = ConstantNode\ c\text{-}2;$
$val = intval\text{-}add\ c\text{-}1\ c\text{-}2]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ y)\ (ConstantNode\ val)\ |$

add-xzero:
$[\![kind\ g\ x = ConstantNode\ c\text{-}1;$
$\neg(is\text{-}ConstantNode\ (kind\ g\ y));$
$c\text{-}1 = (IntVal\ 32\ 0)]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ y)\ (RefNode\ y)\ |$

add-yzero:
$[\![\neg(is\text{-}ConstantNode\ (kind\ g\ x));$
$kind\ g\ y = ConstantNode\ c\text{-}2;$
$c\text{-}2 = (IntVal\ 32\ 0)]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ y)\ (RefNode\ x)\ |$

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*;

$fv = \textit{intval-add c-2 c-3}$;
$g' = \textit{replace-node tb}$ ((*ConstantNode tv*), *constantAsStamp tv*) g;
$g'' = \textit{replace-node fb}$ ((*ConstantNode fv*), *constantAsStamp fv*) g';
$g''' = \textit{replace-node op-id}$ (*kind g* (*ir-x op*), *meet* (*constantAsStamp tv*) (*constantAsStamp fv*)) g'' ⟧
$\Longrightarrow$ *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*)) ∨ ($x > y$))⟧
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AddNode x y*) (*AddNode y x*) |

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

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

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

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

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

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

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

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

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

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

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

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

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

sub-xzero:

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

sub-y-negate:

$⟦kind\ g\ nb = NegateNode\ b⟧$
$\Longrightarrow CanonicalizeSub\ g\ (SubNode\ a\ nb)\ (AddNode\ a\ b)$

```
inductive CanonicalizeMul :: IRGraph ⇒ IRNode ⇒ IRNode ⇒ bool
  for g where
  mul-both-const:
  ⟦kind g x = ConstantNode c-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 = (IntVal b 0)⟧
    ⟹ CanonicalizeMul g (MulNode x y) (ConstantNode c-1) |

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

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

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

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

  mul-ynegate:
  ⟦kind g y = ConstantNode c-1;
    ¬(is-ConstantNode (kind g x));
    c-1 = (IntVal b (−1))⟧
    ⟹ 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 = (*IntVal b v*);
neg-val = *intval-sub* (*IntVal b 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 = *bool-to-val* (¬(*val-to-bool val*)) ⟧
⟹ *CanonicalizeNot g* (*NotNode nx*) (*ConstantNode neg-val*) |

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

inductive *CanonicalizeAnd* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
and-same:
⟦*x* = *y*⟧
⟹ *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:
$[\![kind\ g\ x = ConstantNode\ val;$
$\neg(val\text{-}to\text{-}bool\ val)]\!]$
$\Longrightarrow CanonicalizeOr\ g\ (OrNode\ x\ y)\ (RefNode\ y)\ |$

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

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

de-morgan-or-to-and:
⟦*kind g nid* = *OrNode nx ny*;
kind g nx = *NotNode x*;
kind g ny = *NotNode y*;
new-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* (*IntVal 1 1*)) |

int-equals-distinct:
⟦*alwaysDistinct* (*stamp g x*) (*stamp g y*)⟧
⟹ *CanonicalizeIntegerEquals g* (*IntegerEqualsNode x y*) (*ConstantNode* (*IntVal 1 0*)) |

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

⟦*kind g left* = *AddNode x y*;
kind g right = *AddNode x z*⟧
⟹ *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*⟧
⟹ *CanonicalizeIntegerEquals g* (*IntegerEqualsNode left right*) (*IntegerEqualsNode*

y z) |

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

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

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

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

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

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

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

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

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

int-equals-left-contains-right1:
$[\![$*kind g nid* = *IntegerEqualsNode left x*;
kind g left = *AddNode x y*;
const-id = *nextNid g*;
g' = *add-node const-id* ((*ConstantNode* (*IntVal 1 0*)), *constantAsStamp* (*IntVal 1 0*)) *g*;
g'' = *replace-node const-id* ((*IntegerEqualsNode y const-id*), *stamp g nid*) *g'*$]\!]$

$\Longrightarrow$ *CanonicalizeIntegerEqualsGraph nid g g''* |

int-equals-left-contains-right2:
$[\![$*kind g nid = IntegerEqualsNode left y*;
kind g left = AddNode x y;
const-id = nextNid g;
g' = add-node const-id ((ConstantNode (IntVal 1 0)), constantAsStamp (IntVal 1 0)) g;
g'' = replace-node const-id ((IntegerEqualsNode x const-id), stamp g nid) g'$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEqualsGraph nid g g''* |

int-equals-right-contains-left1:
$[\![$*kind g nid = IntegerEqualsNode x right*;
kind g right = AddNode x y;
const-id = nextNid g;
g' = add-node const-id ((ConstantNode (IntVal 1 0)), constantAsStamp (IntVal 1 0)) g;
g'' = replace-node const-id ((IntegerEqualsNode y const-id), stamp g nid) g'$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEqualsGraph nid g g''* |

int-equals-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 (IntVal 1 0)), constantAsStamp (IntVal 1 0)) g;
g'' = replace-node const-id ((IntegerEqualsNode x const-id), stamp g nid) g'$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEqualsGraph nid g g''* |

int-equals-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 (IntVal 1 0)), constantAsStamp (IntVal 1 0)) g;
g'' = replace-node const-id ((IntegerEqualsNode y const-id), stamp g nid) g'$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEqualsGraph nid g g''* |

int-equals-right-contains-left3:

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

inductive $CanonicalizationStep :: IRGraph \Rightarrow IRNode \Rightarrow IRNode \Rightarrow bool$
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'*

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*) *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*;

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

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

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

⟦$Step\ analyse\ g\ (nid,\ seen,\ i)\ None$;
$Some\ nid' = pred\ g\ nid$;
$seen' = \{nid\} \cup seen$;
$CanonicalizationPhase\ g\ (nid',\ seen',\ i)\ g'$⟧
$\Longrightarrow CanonicalizationPhase\ g\ (nid,\ seen,\ i)\ g'$ |

⟦$Step\ analyse\ g\ (nid,\ seen,\ i)\ None$;
$None = pred\ g\ nid$⟧
$\Longrightarrow CanonicalizationPhase\ g\ (nid,\ seen,\ i)\ g$

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

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

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

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

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

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

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

⟦$Step\ analyse\ g\ (nid,\ seen,\ i)\ None$;
$Some\ nid' = pred\ g\ nid$;

$seen' = \{nid\} \cup seen;$
$CanonicalizationPhaseWithTrace\ g\ (nid', seen', i)\ g'\ (kind\ g\ nid\ \#\ t)\ t'\ ]\!]$
$\Longrightarrow CanonicalizationPhaseWithTrace\ g\ (nid, seen, i)\ g'\ t\ t'\ |$

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

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

end

9 Conditional Elimination Phase

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

9.1 Individual Elimination Rules

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

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

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

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

less-imp-not-eq-rev:
$g \vdash (IntegerLessThanNode\ x\ y)\ \&\ (IntegerEqualsNode\ y\ x) \hookrightarrow KnownFalse$ |

x-imp-x:
$g \vdash x\ \&\ x \hookrightarrow KnownTrue$ |

negate-false:
$[\![g \vdash x\ \&\ (kind\ g\ y) \hookrightarrow KnownTrue]\!] \Longrightarrow g \vdash x\ \&\ (LogicNegationNode\ y) \hookrightarrow KnownFalse$ |
negate-true:
$[\![g \vdash x\ \&\ (kind\ g\ y) \hookrightarrow KnownFalse]\!] \Longrightarrow g \vdash x\ \&\ (LogicNegationNode\ y) \hookrightarrow KnownTrue$

Total relation over partial implies relation

inductive *condition-implies* :: $IRGraph \Rightarrow IRNode \Rightarrow IRNode \Rightarrow TriState \Rightarrow bool$
(- $\vdash$ - & - $\rightharpoonup$ -) **for** *g* **where**
$[\![\neg(g \vdash a\ \&\ b \hookrightarrow imp)]\!] \Longrightarrow (g \vdash a\ \&\ b \rightharpoonup Unknown)$ |
$[\![(g \vdash a\ \&\ b \hookrightarrow imp)]\!] \Longrightarrow (g \vdash a\ \&\ b \rightharpoonup imp)$

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

lemma *logic-negation-relation*:
assumes *wf-values g*
assumes $g\ m \vdash kind\ g\ y \mapsto val$
assumes $kind\ g\ neg = LogicNegationNode\ y$
assumes $g\ m \vdash kind\ g\ neg \mapsto invval$
shows $val\text{-}to\text{-}bool\ val \longleftrightarrow \neg(val\text{-}to\text{-}bool\ inval)$
proof –
have *wf-value val*
using *assms*(*1*) *assms*(*2*) *eval-in-ids wf-values.elims*(*2*)
by *meson*
have *wf-value invval*
using *assms*(*1*,*4*) *eval-in-ids wf-values.simps* **by** *blast*
then show *?thesis*
using *assms eval.LogicNegationNode*
by *fastforce*
qed

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

```
  show ?TC proof -
  have s: g ⊢ x & y ↪ imp
    using KnownTrue assms(2) condition-implies.cases by blast
  then show ?thesis
  using KnownTrue assms proof (induct x y imp rule: implies.induct)
    case (eq-imp-less x y)
    then show ?case by simp
  next
    case (eq-imp-less-rev x y)
    then show ?case by simp
  next
    case (less-imp-rev-less x y)
    then show ?case by simp
  next
    case (less-imp-not-eq x y)
    then show ?case by simp
  next
    case (less-imp-not-eq-rev x y)
    then show ?case by simp
  next
    case (x-imp-x x1)
    then show ?case using evalDet
      using assms(2,3) by blast
  next
    case (negate-false x1)
    then show ?case using evalDet
      using assms(2,3) by blast
  next
    case (negate-true x y)
    then show ?case using logic-negation-relation
      by fastforce
  qed
  qed
next
  assume KnownFalse: ?FP
  show ?FC proof -
    have g ⊢ x & y ↪ imp
    using KnownFalse assms(2) condition-implies.cases by blast
  then show ?thesis
  using assms KnownFalse proof (induct x y imp rule: implies.induct)
    case (eq-imp-less x y)
    obtain b xval where xval: g m ⊢ (kind g x) ↦ IntVal b xval
      using eq-imp-less.prems(3) by blast
    then obtain yval where yval: g m ⊢ (kind g y) ↦ IntVal b yval
      using eq-imp-less.prems(3)
      using evalDet by blast
    have eqeval: g m ⊢ (IntegerEqualsNode x y) ↦ bool-to-val(xval = yval)
      using eval.IntegerEqualsNode
      using xval yval by blast
```

have *lesseval*: $g\ m \vdash (IntegerLessThanNode\ x\ y) \mapsto bool\text{-}to\text{-}val(xval < yval)$
using *eval.IntegerLessThanNode*
using *xval yval* **by** *blast*
have $xval = yval \longrightarrow \neg(xval < yval)$
by *blast*
then show *?case*
using *eqeval lesseval*
by (*metis* (*full-types*) *eq-imp-less.prems*(*3*) *eq-imp-less.prems*(*4*) *bool-to-val.simps*(*2*) *evalDet val-to-bool.simps*(*1*))
next
case (*eq-imp-less-rev x y*)
obtain *b xval* **where** *xval*: $g\ m \vdash (kind\ g\ x) \mapsto IntVal\ b\ xval$
using *eq-imp-less-rev.prems*(*3*) **by** *blast*
then obtain *yval* **where** *yval*: $g\ m \vdash (kind\ g\ y) \mapsto IntVal\ b\ yval$
using *eq-imp-less-rev.prems*(*3*)
using *evalDet* **by** *blast*
have *eqeval*: $g\ m \vdash (IntegerEqualsNode\ x\ y) \mapsto bool\text{-}to\text{-}val(xval = yval)$
using *eval.IntegerEqualsNode*
using *xval yval* **by** *blast*
have *lesseval*: $g\ m \vdash (IntegerLessThanNode\ y\ x) \mapsto bool\text{-}to\text{-}val(yval < xval)$
using *eval.IntegerLessThanNode*
using *xval yval* **by** *blast*
have $xval = yval \longrightarrow \neg(yval < xval)$
by *blast*
then show *?case*
using *eqeval lesseval*
by (*metis* (*full-types*) *eq-imp-less-rev.prems*(*3*) *eq-imp-less-rev.prems*(*4*) *bool-to-val.simps*(*2*) *evalDet val-to-bool.simps*(*1*))
next
case (*less-imp-rev-less x y*)
obtain *b xval* **where** *xval*: $g\ m \vdash (kind\ g\ x) \mapsto IntVal\ b\ xval$
using *less-imp-rev-less.prems*(*3*) **by** *blast*
then obtain *yval* **where** *yval*: $g\ m \vdash (kind\ g\ y) \mapsto IntVal\ b\ yval$
using *less-imp-rev-less.prems*(*3*)
using *evalDet* **by** *blast*
have *lesseval*: $g\ m \vdash (IntegerLessThanNode\ x\ y) \mapsto bool\text{-}to\text{-}val(xval < yval)$
using *eval.IntegerLessThanNode*
using *xval yval* **by** *blast*
have *revlesseval*: $g\ m \vdash (IntegerLessThanNode\ y\ x) \mapsto bool\text{-}to\text{-}val(yval < xval)$
using *eval.IntegerLessThanNode*
using *xval yval* **by** *blast*
have $xval < yval \longrightarrow \neg(yval < xval)$
by *simp*
then show *?case*
by (*metis* (*full-types*) *bool-to-val.simps*(*2*) *evalDet less-imp-rev-less.prems*(*3*,*4*) *less-imp-rev-less.prems*(*3*) *lesseval revlesseval val-to-bool.simps*(*1*))
next
case (*less-imp-not-eq x y*)
obtain *b xval* **where** *xval*: $g\ m \vdash (kind\ g\ x) \mapsto IntVal\ b\ xval$

```
      using less-imp-not-eq.prems(3) by blast
    then obtain yval where yval: g m ⊢ (kind g y) ↦ IntVal b yval
      using less-imp-not-eq.prems(3)
      using evalDet by blast
    have eqeval: g m ⊢ (IntegerEqualsNode x y) ↦ bool-to-val(xval = yval)
      using eval.IntegerEqualsNode
      using xval yval by blast
    have lesseval: g m ⊢ (IntegerLessThanNode x y) ↦ bool-to-val(xval < yval)
      using eval.IntegerLessThanNode
      using xval yval by blast
    have xval < yval ⟶ ¬(xval = yval)
      by simp
    then show ?case
    by (metis (full-types) bool-to-val.simps(2) eqeval evalDet less-imp-not-eq.prems(3,4)
less-imp-not-eq.prems(3) lesseval val-to-bool.simps(1))
  next
    case (less-imp-not-eq-rev x y)
    obtain b xval where xval: g m ⊢ (kind g x) ↦ IntVal b xval
      using less-imp-not-eq-rev.prems(3) by blast
    then obtain yval where yval: g m ⊢ (kind g y) ↦ IntVal b yval
      using less-imp-not-eq-rev.prems(3)
      using evalDet by blast
    have eqeval: g m ⊢ (IntegerEqualsNode y x) ↦ bool-to-val(yval = xval)
      using eval.IntegerEqualsNode
      using xval yval by blast
    have lesseval: g m ⊢ (IntegerLessThanNode x y) ↦ bool-to-val(xval < yval)
      using eval.IntegerLessThanNode
      using xval yval by blast
    have xval < yval ⟶ ¬(yval = xval)
      by simp
    then show ?case
    by (metis (full-types) bool-to-val.simps(2) eqeval evalDet less-imp-not-eq-rev.prems(3,4)
less-imp-not-eq-rev.prems(3) lesseval val-to-bool.simps(1))
  next
    case (x-imp-x x1)
    then show ?case by simp
  next
    case (negate-false x y)
    then show ?case using logic-negation-relation sorry
  next
    case (negate-true x1)
    then show ?case by simp
  qed
  qed
qed

lemma implies-true-valid:
  assumes wf-graph g ∧ wf-values g
  assumes g ⊢ x & y ⇀ imp
```

assumes *imp* = *KnownTrue*
assumes $g\ m \vdash x \mapsto v1$
assumes $g\ m \vdash y \mapsto v2$
shows *val-to-bool v1* $\longrightarrow$ *val-to-bool v2*
using *assms implies-valid* **by** *blast*

lemma *implies-false-valid*:
assumes *wf-graph g* $\wedge$ *wf-values g*
assumes $g \vdash x\ \&\ y \rightharpoonup imp$
assumes *imp* = *KnownFalse*
assumes $g\ m \vdash x \mapsto v1$
assumes $g\ m \vdash y \mapsto v2$
shows *val-to-bool v1* $\longrightarrow \neg$(*val-to-bool v2*)
using *assms implies-valid* **by** *blast*

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

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

inductive *tryFold* :: *IRNode* $\Rightarrow$ (*ID* $\Rightarrow$ *Stamp*) $\Rightarrow$ *TriState* $\Rightarrow$ *bool*
where
$[\![$*alwaysDistinct* (*stamps x*) (*stamps y*)$]\!]$
$\Longrightarrow$ *tryFold* (*IntegerEqualsNode x y*) *stamps KnownFalse* |
$[\![$*neverDistinct* (*stamps x*) (*stamps y*)$]\!]$
$\Longrightarrow$ *tryFold* (*IntegerEqualsNode x y*) *stamps KnownTrue* |
$[\![$*is-IntegerStamp* (*stamps x*);
is-IntegerStamp (*stamps y*);
stpi-upper (*stamps x*) < *stpi-lower* (*stamps y*)$]\!]$
$\Longrightarrow$ *tryFold* (*IntegerLessThanNode x y*) *stamps KnownTrue* |
$[\![$*is-IntegerStamp* (*stamps x*);
is-IntegerStamp (*stamps y*);
stpi-lower (*stamps x*) $\geq$ *stpi-upper* (*stamps y*)$]\!]$
$\Longrightarrow$ *tryFold* (*IntegerLessThanNode x y*) *stamps KnownFalse*

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

lemma *tryFoldIntegerEqualsAlwaysDistinct*:
assumes *wf-stamp g stamps*
assumes *kind g nid* = (*IntegerEqualsNode x y*)
assumes $g\ m \vdash (kind\ g\ nid) \mapsto v$
assumes *alwaysDistinct* (*stamps x*) (*stamps y*)
shows *v* = *IntVal 1 0*
using *assms eval.IntegerEqualsNode join-unequal alwaysDistinct.simps*
by (*smt* (*verit*, *best*) *IntegerEqualsNodeE bool-to-val.simps*(*2*) *eval-in-ids wf-stamp.elims*(*2*))

lemma *tryFoldIntegerEqualsNeverDistinct*:

```
  assumes wf-stamp g stamps
  assumes kind g nid = (IntegerEqualsNode x y)
  assumes g m ⊢ (kind g nid) ↦ v
  assumes neverDistinct (stamps x) (stamps y)
  shows v = IntVal 1 1
  using assms neverDistinctEqual IntegerEqualsNodeE
   by (smt (verit, ccfv-threshold) Value.inject(1) bool-to-val.simps(1) eval-in-ids
wf-stamp.simps)

lemma tryFoldIntegerLessThanTrue:
  assumes wf-stamp g stamps
  assumes kind g nid = (IntegerLessThanNode x y)
  assumes g m ⊢ (kind g nid) ↦ v
  assumes stpi-upper (stamps x) < stpi-lower (stamps y)
  shows v = IntVal 1 1
proof -
  have stamp-type: is-IntegerStamp (stamps x)
    using assms
     by (metis IntegerLessThanNodeE Stamp.disc(2) Value.distinct(1) eval-in-ids
valid-value.elims(2) wf-stamp.elims(2))
  obtain xval b where xval: g m ⊢ kind g x ↦ IntVal b xval
    using assms(2,3) eval.IntegerLessThanNode by auto
  obtain yval b where yval: g m ⊢ kind g y ↦ IntVal b yval
    using assms(2,3) eval.IntegerLessThanNode by auto
  have is-IntegerStamp (stamps x) ∧ is-IntegerStamp (stamps y)
    using assms(4)
      by (metis stamp-type Stamp.disc(2) Value.distinct(1) assms(1) eval-in-ids
valid-value.elims(2) wf-stamp.simps yval)
  then have xval < yval
    using boundsNoOverlap xval yval assms(1,4)
    using eval-in-ids wf-stamp.elims(2)
    by metis
  then show ?thesis
    by (metis (full-types) IntegerLessThanNodeE Value.sel(3) assms(2) assms(3)
bool-to-val.simps(1) evalDet xval yval)
qed

lemma tryFoldIntegerLessThanFalse:
  assumes wf-stamp g stamps
  assumes kind g nid = (IntegerLessThanNode x y)
  assumes g m ⊢ (kind g nid) ↦ v
  assumes stpi-lower (stamps x) ≥ stpi-upper (stamps y)
  shows v = IntVal 1 0
  proof -
  have stamp-type: is-IntegerStamp (stamps x)
    using assms
     by (metis IntegerLessThanNodeE Stamp.disc(2) Value.distinct(1) eval-in-ids
valid-value.elims(2) wf-stamp.elims(2))
  obtain xval b where xval: g m ⊢ kind g x ↦ IntVal b xval
```

using *assms*(*2*,*3*) *eval.IntegerLessThanNode* **by** *auto*
obtain *yval b* **where** *yval*: *g m* ⊢ *kind g y* ↦ *IntVal b yval*
using *assms*(*2*,*3*) *eval.IntegerLessThanNode* **by** *auto*
have *is-IntegerStamp* (*stamps x*) ∧ *is-IntegerStamp* (*stamps y*)
using *assms*(*4*)
by (*metis stamp-type Stamp.disc*(*2*) *Value.distinct*(*1*) *assms*(*1*) *eval-in-ids valid-value.elims*(*2*) *wf-stamp.simps yval*)
then have ¬(*xval* < *yval*)
using *boundsAlwaysOverlap xval yval assms*(*1*,*4*)
using *eval-in-ids wf-stamp.elims*(*2*)
by *metis*
then show *?thesis*
by (*smt* (*verit*, *best*) *IntegerLessThanNodeE Value.inject*(*1*) *assms*(*2*) *assms*(*3*) *bool-to-val.simps*(*2*) *evalDet xval yval*)
qed

theorem *tryFoldProofTrue*:
assumes *wf-stamp g stamps*
assumes *tryFold* (*kind g nid*) *stamps tristate*
assumes *tristate* = *KnownTrue*
assumes *g m* ⊢ *kind g nid* ↦ *v*
shows *val-to-bool v*
using *assms*(*2*) **proof** (*induction kind g nid stamps tristate rule*: *tryFold.induct*)
case (*1 stamps x y*)
then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms*
by (*smt* (*verit*, *best*) *IRNode.distinct*(*949*) *TriState.distinct*(*5*) *tryFold.cases tryFoldIntegerEqualsNeverDistinct val-to-bool.simps*(*1*))
next
case (*2 stamps x y*)
then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms*
by (*smt* (*verit*) *IRNode.distinct*(*949*) *TriState.distinct*(*5*) *tryFold.cases tryFold-IntegerEqualsNeverDistinct val-to-bool.simps*(*1*))
next
case (*3 stamps x y*)
then show *?case* **using** *tryFoldIntegerLessThanTrue assms*
by (*smt* (*verit*, *best*) *IRNode.simps*(*994*) *TriState.simps*(*6*) *tryFold.cases val-to-bool.simps*(*1*))
next
case (*4 stamps x y*)
then show *?case* **using** *tryFoldIntegerLessThanFalse assms*
by (*smt* (*verit*, *best*) *IRNode.simps*(*994*) *TriState.simps*(*6*) *tryFold.simps try-FoldIntegerLessThanTrue val-to-bool.simps*(*1*))
qed

theorem *tryFoldProofFalse*:
assumes *wf-stamp g stamps*
assumes *tryFold* (*kind g nid*) *stamps tristate*
assumes *tristate* = *KnownFalse*
assumes *g m* ⊢ (*kind g nid*) ↦ *v*
shows ¬(*val-to-bool v*)

using *assms*(*2*) **proof** (*induction kind g nid stamps tristate rule*: *tryFold.induct*)
case (*1 stamps x y*)
  then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms*
    by (*smt* (*verit*, *best*) *IRNode.distinct*(*949*) *TriState.distinct*(*5*) *Value.inject*(*1*) *tryFold.cases val-to-bool.elims*(*2*))
next
case (*2 stamps x y*)
  then show *?case* **using** *tryFoldIntegerEqualsNeverDistinct assms*
    by (*smt* (*verit*, *best*) *IRNode.distinct*(*949*) *TriState.distinct*(*5*) *Value.inject*(*1*) *tryFold.cases tryFoldIntegerEqualsAlwaysDistinct val-to-bool.elims*(*2*))
next
  case (*3 stamps x y*)
  then show *?case* **using** *tryFoldIntegerLessThanTrue assms*
    by (*smt* (*verit*, *best*) *TriState.distinct*(*5*) *tryFold.cases tryFoldIntegerEqualsAlwaysDistinct tryFoldIntegerLessThanFalse val-to-bool.simps*(*1*))
next
  case (*4 stamps x y*)
  then show *?case* **using** *tryFoldIntegerLessThanFalse assms*
    by (*smt* (*verit*, *best*) *TriState.distinct*(*5*) *tryFold.cases tryFoldIntegerEqualsAlwaysDistinct val-to-bool.simps*(*1*))
qed

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

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

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

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

inductive *ConditionalEliminationStep* ::
  IRNode set ⇒ (*ID* ⇒ *Stamp*) ⇒ *IRGraph* ⇒ *ID* ⇒ *IRGraph* ⇒ *bool* **where**
  impliesTrue:
  ⟦*kind g ifcond* = (*IfNode cid t f*);
    cond = *kind g cid*;
    ∃ *c* ∈ *conds* . (*g* ⊢ *c* & *cond* ↪ *KnownTrue*);
    g′ = *constantCondition True ifcond* (*kind g ifcond*) *g*
    ⟧ ⟹ *ConditionalEliminationStep conds stamps g ifcond g′* |

  impliesFalse:
  ⟦*kind g ifcond* = (*IfNode cid t f*);
    cond = *kind g cid*;
    ∃ *c* ∈ *conds* . (*g* ⊢ *c* & *cond* ↪ *KnownFalse*);

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

tryFoldTrue:
$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
$cond = kind\ g\ cid;$
$tryFold\ (kind\ g\ cid)\ stamps\ KnownTrue;$
$g' = constantCondition\ True\ ifcond\ (kind\ g\ ifcond)\ g$
$\rrbracket \Longrightarrow ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'$ |

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

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

thm *ConditionalEliminationStep.equation*

9.2 Control-flow Graph Traversal

type-synonym *Seen* = *ID set*
type-synonym *Conditions* = *IRNode list*
type-synonym *StampFlow* = (*ID* ⇒ *Stamp*) *list*

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

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

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

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

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

fun *pred* :: *IRGraph* ⇒ *ID* ⇒ *ID option* **where**
pred g nid = (*case kind g nid of*
(*MergeNode ends - -*) ⇒ *Some* (*hd ends*) |
- ⇒

```
    (if IRGraph.predecessors g nid = {}
      then None else
      Some (hd (sorted-list-of-set (IRGraph.predecessors g nid)))
    )
  )
```

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

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

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

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

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

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

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

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

```
inductive Step
  :: IRGraph ⇒ (ID × Seen × Conditions × StampFlow) ⇒ (ID × Seen ×
Conditions × StampFlow) option ⇒ bool
  for g where
```

— Hit a BeginNode with an IfNode predecessor which represents the start of a basic block for the IfNode. 1. nid' will be the successor of the begin node. 2. Find the first and only predecessor. 3. Extract condition from the preceding IfNode. 4. Negate condition if the begin node is second branch (we've taken the else branch of the condition) 5. Add the condition or the negated condition to stack 6. Perform any stamp updates based on the condition using the registerNewCondition function and place them on the top of the stack of stamp information

$[\![kind\ g\ nid = BeginNode\ nid';$

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

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

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

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

— Hit an EndNode 1. nid' will be the usage of EndNode 2. pop the conditions and stamp stack

$[\![kind\ g\ nid = EndNode;$

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

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

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

— We can find a successor edge that is not in seen, go there

$[\![\neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\neg(is\text{-}BeginNode\ (kind\ g\ nid));$

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

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

— We can cannot find a successor edge that is not in seen, give back None

$[\![\neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\neg(is\text{-}BeginNode\ (kind\ g\ nid));$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *IfNodeStepE*: $g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h) \Longrightarrow$

$(\bigwedge cond\ tb\ fb\ val.$
$\quad kind\ g\ nid = IfNode\ cond\ tb\ fb \Longrightarrow$
$\quad nid' = (if\ val\text{-}to\text{-}bool\ val\ then\ tb\ else\ fb) \Longrightarrow$
$\quad g\ m \vdash kind\ g\ cond \mapsto val \Longrightarrow m' = m)$
using *StepE*
by (*smt* (*verit*, *best*) *IfNode Pair-inject stepDet*)

lemma *ifNodeHasCondEvalStutter*:
assumes $(g\ m\ h \vdash nid \leadsto nid')$
assumes $kind\ g\ nid = IfNode\ cond\ t\ f$
shows $\exists\ v.\ (g\ m \vdash kind\ g\ cond \mapsto v)$
using *IfNodeStepE assms*(*1*) *assms*(*2*) *stutter.cases*
by (*meson IfNodeCond*)

lemma *ifNodeHasCondEval*:
assumes $(g \vdash (nid, m, h) \to (nid', m', h'))$
assumes $kind\ g\ nid = IfNode\ cond\ t\ f$
shows $\exists\ v.\ (g\ m \vdash kind\ g\ cond \mapsto v)$
using *IfNodeStepE assms*(*1*) *assms*(*2*)
by (*smt* (*z3*) *IRNode.disc*(*932*) *IRNode.simps*(*938*) *IRNode.simps*(*958*) *IRNode.simps*(*972*) *IRNode.simps*(*974*) *IRNode.simps*(*978*) *Pair-inject StutterStep ifNodeHasCondEvalStutter is-AbstractEndNode.simps is-EndNode.simps*(*12*) *step.cases*)

lemma *replace-if-t*:
assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
assumes $g\ m \vdash kind\ g\ cond \mapsto bool$
assumes *val-to-bool bool*
assumes g': $g' = replace\text{-}usages\ nid\ tb\ g$
shows $\exists nid'.(g\ m\ h \vdash nid \leadsto nid') \longleftrightarrow (g'\ m\ h \vdash nid \leadsto nid')$
proof −
have *g1step*: $g \vdash (nid, m, h) \to (tb, m, h)$
by (*meson IfNode assms*(*1*) *assms*(*2*) *assms*(*3*))
have *g2step*: $g' \vdash (nid, m, h) \to (tb, m, h)$
using g' **unfolding** *replace-usages.simps*
by (*simp add*: *stepRefNode*)
from *g1step g2step* **show** *?thesis*
using *StutterStep* **by** *blast*
qed

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

lemma *replace-if-f*:

assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
assumes $g\ m \vdash kind\ g\ cond \mapsto bool$
assumes $\neg(val\text{-}to\text{-}bool\ bool)$
assumes g': $g' = replace\text{-}usages\ nid\ fb\ g$
shows $\exists nid'\ .(g\ m\ h \vdash nid \leadsto nid') \longleftrightarrow (g'\ m\ h \vdash nid \leadsto nid')$
proof −
have *g1step*: $g \vdash (nid,\ m,\ h) \to (fb,\ m,\ h)$
by (*meson IfNode assms*(*1*) *assms*(*2*) *assms*(*3*))
have *g2step*: $g' \vdash (nid,\ m,\ h) \to (fb,\ m,\ h)$
using g' **unfolding** *replace-usages.simps*
by (*simp add*: *stepRefNode*)
from *g1step g2step* **show** *?thesis*
using *StutterStep* **by** *blast*
qed

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

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

shows $\exists nid'\ .(g\ m\ h \vdash nid \leadsto nid') \longrightarrow (g'\ m\ h \vdash nid \leadsto nid')$
using *ce* **using** *assms*
proof (*induct g nid g' rule*: *ConditionalEliminationStep.induct*)
case (*impliesTrue g ifcond cid t f cond conds g'*)
show *?case* **proof** (*cases* ($g\ m\ h \vdash ifcond \leadsto nid'$))
case *True*
obtain *condv* **where** *condv*: $g\ m \vdash kind\ g\ cid \mapsto condv$
using *implies.simps impliesTrue.hyps*(*3*) *impliesTrue.prems*(*4*)
using *impliesTrue.hyps*(*2*) *True*
by (*metis ifNodeHasCondEvalStutter impliesTrue.hyps*(*1*))
have *condvTrue*: *val-to-bool condv*
by (*metis condition-implies.intros*(*2*) *condv impliesTrue.hyps*(*2*) *impliesTrue.hyps*(*3*) *impliesTrue.prems*(*1*) *impliesTrue.prems*(*3*) *impliesTrue.prems*(*5*) *implies-true-valid*)
then show *?thesis*
using *constantConditionValid*
using *impliesTrue.hyps*(*1*) *condv impliesTrue.hyps*(*4*)
by *blast*
next
case *False*
then show *?thesis* **by** *auto*
qed
next
case (*impliesFalse g ifcond cid t f cond conds g'*)
then show *?case*

proof (*cases* (*g m h* ⊢ *ifcond* ⇝ *nid′*))
case *True*
obtain *condv* **where** *condv*: *g m* ⊢ *kind g cid* ↦ *condv*
using *ifNodeHasCondEvalStutter impliesFalse.hyps*(*1*)
using *True* **by** *blast*
have *condvFalse*: *False* = *val-to-bool condv*
by (*metis condition-implies.intros*(*2*) *condv impliesFalse.hyps*(*2*) *impliesFalse.hyps*(*3*) *impliesFalse.prems*(*1*) *impliesFalse.prems*(*3*) *impliesFalse.prems*(*5*) *implies-false-valid*)
then show *?thesis*
using *constantConditionValid*
using *impliesFalse.hyps*(*1*) *condv impliesFalse.hyps*(*4*)
by *blast*
next
case *False*
then show *?thesis*
by *auto*
qed
next
case (*tryFoldTrue g ifcond cid t f cond g′ conds*)
then show *?case* **using** *constantConditionValid tryFoldProofTrue*
using *StutterStep constantConditionTrue* **by** *metis*
next
case (*tryFoldFalse g ifcond cid t f cond g′ conds*)
then show *?case* **using** *constantConditionValid tryFoldProofFalse*
using *StutterStep constantConditionFalse* **by** *metis*
qed

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

lemma *ConditionalEliminationStepProofBisimulation*:
assumes *wf*: *wf-graph g* ∧ *wf-stamp g stamps* ∧ *wf-values g*
assumes *nid*: *nid* ∈ *ids g*
assumes *conds-valid*: ∀ *c* ∈ *conds* . ∃ *v*. (*g m* ⊢ *c* ↦ *v*) ∧ *val-to-bool v*
assumes *ce*: *ConditionalEliminationStep conds stamps g nid g′*
assumes *gstep*: ∃ *h nid′*. (*g* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*))

shows *nid* | *g* ∼ *g′*
using *ce gstep* **using** *assms*
proof (*induct g nid g′ rule*: *ConditionalEliminationStep.induct*)
case (*impliesTrue g ifcond cid t f cond conds g′ stamps*)
from *impliesTrue*(*5*) **obtain** *h* **where** *gstep*: *g* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
by (*metis IfNode StutterStep condition-implies.intros*(*2*) *ifNodeHasCondEvalStutter impliesTrue.hyps*(*1*) *impliesTrue.hyps*(*2*) *impliesTrue.hyps*(*3*) *impliesTrue.prems*(*2*) *impliesTrue.prems*(*4*) *implies-true-valid*)
have *g′* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
using *constantConditionTrue impliesTrue.hyps*(*1*) *impliesTrue.hyps*(*4*) **by** *blast*
then show *?case* **using** *gstep*

by (*metis stepDet strong-noop-bisimilar.intros*)
next
case (*impliesFalse g ifcond cid t f cond conds g′ stamps*)
from *impliesFalse*(*5*) **obtain** *h* **where** *gstep*: $g \vdash (ifcond,\ m,\ h) \rightarrow (f,\ m,\ h)$
by (*metis IfNode condition-implies.intros*(*2*) *ifNodeHasCondEval impliesFalse.hyps*(*1*) *impliesFalse.hyps*(*2*) *impliesFalse.hyps*(*3*) *impliesFalse.prems*(*2*) *impliesFalse.prems*(*4*) *implies-false-valid*)
have $g' \vdash (ifcond,\ m,\ h) \rightarrow (f,\ m,\ h)$
using *constantConditionFalse impliesFalse.hyps*(*1*) *impliesFalse.hyps*(*4*) **by** *blast*
then show *?case* **using** *gstep*
by (*metis stepDet strong-noop-bisimilar.intros*)
next
case (*tryFoldTrue g ifcond cid t f cond stamps g′ conds*)
from *tryFoldTrue*(*5*) **obtain** *val* **where** $g\ m \vdash kind\ g\ cid \mapsto val$
using *ifNodeHasCondEval tryFoldTrue.hyps*(*1*) **by** *blast*
then have *val-to-bool val*
using *tryFoldProofTrue tryFoldTrue.prems*(*2*) *tryFoldTrue*(*3*)
by *blast*
then obtain *h* **where** *gstep*: $g \vdash (ifcond,\ m,\ h) \rightarrow (t,\ m,\ h)$
using *tryFoldTrue*(*5*)
by (*meson IfNode* ‹$g\ m \vdash kind\ g\ cid \mapsto val$› *tryFoldTrue.hyps*(*1*))
have $g' \vdash (ifcond,\ m,\ h) \rightarrow (t,\ m,\ h)$
using *constantConditionTrue tryFoldTrue.hyps*(*1*) *tryFoldTrue.hyps*(*4*) **by** *presburger*
then show *?case* **using** *gstep*
by (*metis stepDet strong-noop-bisimilar.intros*)
next
case (*tryFoldFalse g ifcond cid t f cond stamps g′ conds*)
from *tryFoldFalse*(*5*) **obtain** *h* **where** *gstep*: $g \vdash (ifcond,\ m,\ h) \rightarrow (f,\ m,\ h)$
by (*meson IfNode ifNodeHasCondEval tryFoldFalse.hyps*(*1*) *tryFoldFalse.hyps*(*3*) *tryFoldFalse.prems*(*2*) *tryFoldProofFalse*)
have $g' \vdash (ifcond,\ m,\ h) \rightarrow (f,\ m,\ h)$
using *constantConditionFalse tryFoldFalse.hyps*(*1*) *tryFoldFalse.hyps*(*4*) **by** *blast*
then show *?case* **using** *gstep*
by (*metis stepDet strong-noop-bisimilar.intros*)
qed

Mostly experimental proofs from here on out.

lemma *if-step*:
assumes $nid \in ids\ g$
assumes $(kind\ g\ nid) \in control\text{-}nodes$
shows $(g\ m\ h \vdash nid \leadsto nid')$
using *assms* **apply** (*cases kind g nid*) **sorry**

lemma *StepConditionsValid*:
assumes $\forall\ cond \in set\ conds.\ (g\ m \vdash cond \mapsto v) \wedge val\text{-}to\text{-}bool\ v$
assumes *Step g* (*nid*, *seen*, *conds*, *flow*) (*Some* (*nid′*, *seen′*, *conds′*, *flow′*))
shows $\forall\ cond \in set\ conds'.\ (g\ m \vdash cond \mapsto v) \wedge val\text{-}to\text{-}bool\ v$
using *assms*(*2*)

proof (*induction* (*nid*, *seen*, *conds*, *flow*) *Some* (*nid'*, *seen'*, *conds'*, *flow'*) *rule*: *Step.induct*)
  case (*1 ifcond cond t f i c*)
  obtain *cv* **where** *cv*: $g\ m \vdash c \mapsto cv$
    sorry
  have *cvt*: *val-to-bool cv*
    sorry
  have $set\ conds' = \{c\} \cup set\ conds$
    using *1.hyps*(*8*) **by** *auto*
  then show *?case* **using** *cv cvt assms*(*1*) **sorry**
next
  case (*2*)
  from *2*(*5*) **have** $set\ conds' \subseteq set\ conds$
    by (*metis list.sel*(*2*) *list.set-sel*(*2*) *subsetI*)
  then show *?case* **using** *assms*(*1*)
    by *blast*
next
case (*3*)
  then show *?case*
    using *assms*(*1*) **by** *force*
qed

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

  shows $\exists\, nid'\ .(g\ m\ h \vdash 0 \leadsto nid') \longrightarrow (g'\ m\ h \vdash 0 \leadsto nid')$
proof –
  have $0 \in ids\ g$
    using *assms*(*1*) *wf-folds* **by** *blast*
  show *?thesis*
using *assms*(*3*) *assms* **proof** (*induct rule*: *ConditionalEliminationPhase.induct*)
case (*1 g nid g' succs nid' g''*)
  then show *?case* **sorry**
next
  case (*2 succs g nid nid' g''*)
  then show *?case* **sorry**
next
  case (*3 succs g nid*)
  then show *?case*
    by *simp*
next
  case (*4*)
  then show *?case* **sorry**
qed
qed

end

10 Graph Construction Phase

```
theory
  Construction
imports
  Proofs.Bisimulation
  Proofs.IRGraphFrames
begin

lemma add-const-nodes:
  assumes xn: kind g x = (ConstantNode (IntVal b xv))
  assumes yn: kind g y = (ConstantNode (IntVal b yv))
  assumes zn: kind g z = (AddNode x y)
  assumes wn: kind g w = (ConstantNode (intval-add (IntVal b xv) (IntVal b yv)))
  assumes val: intval-add (IntVal b xv) (IntVal b yv) = IntVal b v1
  assumes ez: g m ⊢ (kind g z) ↦ (IntVal b v1)
  assumes ew: g m ⊢ (kind g w) ↦ (IntVal b v2)
  shows v1 = v2
proof −
  have zv: g m ⊢ (kind g z) ↦ IntVal b v1
    using eval.AddNode eval.ConstantNode xn yn zn val plus-Value-def by metis
  have wv: g m ⊢ (kind g w) ↦ IntVal b v2
    using eval.ConstantNode wn ew by blast
  show ?thesis using evalDet zv wv ew ez
    using ConstantNode val wn by auto
qed



lemma add-val-xzero:
  shows intval-add (IntVal b 0) (IntVal b yv) = (IntVal b yv)
  unfolding intval-add.simps sorry

lemma add-val-yzero:
  shows intval-add (IntVal b xv) (IntVal b 0) = (IntVal b xv)
  unfolding intval-add.simps sorry

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

- ⇒ *AddNode x y*
)
)

lemma *add-node-create*:
assumes *xv*: $g\ m \vdash (kind\ g\ x) \mapsto IntVal\ b\ xv$
assumes *yv*: $g\ m \vdash (kind\ g\ y) \mapsto IntVal\ b\ yv$
assumes *res*: $res = intval\text{-}add\ (IntVal\ b\ xv)\ (IntVal\ b\ yv)$
shows
$(g\ m \vdash (AddNode\ x\ y) \mapsto res) \wedge$
$(g\ m \vdash (create\text{-}add\ g\ x\ y) \mapsto res)$

proof –
let $?P = (g\ m \vdash (AddNode\ x\ y) \mapsto res)$
let $?Q = (g\ m \vdash (create\text{-}add\ g\ x\ y) \mapsto res)$
have *P*: *?P*
using *xv yv res eval.AddNode plus-Value-def* **by** *metis*
have *Q*: *?Q*
proof (*cases is-ConstantNode* (*kind g x*))
case *xconst*: *True*
then show *?thesis*
proof (*cases is-ConstantNode* (*kind g y*))
case *yconst*: *True*
have *create-add g x y* = *ConstantNode res*
using *xconst yconst*
using *ConstantNodeE is-ConstantNode-def xv yv res* **by** *auto*
then show *?thesis* **using** *eval.ConstantNode* **by** *simp*
next
case *ynotconst*: *False*
have *kind g x* = *ConstantNode* (*IntVal b xv*)
using *ConstantNodeE xconst*
by (*metis is-ConstantNode-def xv*)
then have *add-def*:
create-add g x y = (*if xv* = *0 then RefNode y else AddNode x y*)
using *xconst ynotconst is-ConstantNode-def*
unfolding *create-add.simps*
by (*simp split*: *IRNode.split*)
then show *?thesis*
proof (*cases xv* = *0*)
case *xzero*: *True*
have *ref*: *create-add g x y* = *RefNode y*
using *xzero add-def*
by *meson*
have *refval*: $g\ m \vdash RefNode\ y \mapsto IntVal\ b\ yv$
using *eval.RefNode yv* **by** *simp*
have *res* = *IntVal b yv*
using *res* **unfolding** *xzero add-val-xzero* **by** *simp*

```
        then show ?thesis using xzero ref refval by simp
      next
        case xnotzero: False
        then show ?thesis
          using P add-def by presburger
      qed
    qed
next
  case notxconst: False
  then show ?thesis
    proof (cases is-ConstantNode (kind g y))
      case yconst: True
      have kind g y = ConstantNode (IntVal b yv)
        using ConstantNodeE yconst
        by (metis is-ConstantNode-def yv)
      then have add-def:
        create-add g x y = (if yv = 0 then RefNode x else AddNode x y)
        using notxconst yconst is-ConstantNode-def
        unfolding create-add.simps
        by (simp split: IRNode.split)
      then show ?thesis
      proof (cases yv = 0)
        case yzero: True
        have ref: create-add g x y = RefNode x
          using yzero add-def
          by meson
        have refval: g m ⊢ RefNode x ↦ IntVal b xv
          using eval.RefNode xv by simp
        have res = IntVal b xv
          using res unfolding yzero add-val-yzero by simp
        then show ?thesis using yzero ref refval by simp
      next
        case ynotzero: False
        then show ?thesis
          using P add-def by presburger
      qed

    next
      case notyconst: False
      have create-add g x y = AddNode x y
        using notxconst notyconst is-ConstantNode-def
        create-add.simps by (simp split: IRNode.split)
      then show ?thesis
        using P by presburger
    qed
qed
  from P Q show ?thesis by simp
qed
```

fun *add-node-fake* :: *ID* ⇒ *IRNode* ⇒ *IRGraph* ⇒ *IRGraph* **where**
  add-node-fake nid k g = *add-node nid* (*k*, *VoidStamp*) *g*
lemma *add-node-lookup-fake*:
  assumes *gup* = *add-node-fake nid k g*
  assumes *nid* ∉ *ids g*
  shows *kind gup nid* = *k*
  using *add-node-lookup* **proof** (*cases k* = *NoNode*)
  case *True*
  have *kind g nid* = *NoNode*
    using *assms*(*2*)
    using *not-in-g* **by** *blast*
  then show *?thesis* **using** *assms*
    by (*metis add-node-fake.simps add-node-lookup*)
next
  case *False*
  then show *?thesis*
    by (*simp add*: *add-node-lookup assms*(*1*))
qed
lemma *add-node-unchanged-fake*:
  assumes *new* ∉ *ids g*
  assumes *nid* ∈ *ids g*
  assumes *gup* = *add-node-fake new k g*
  assumes *wf-graph g*
  shows *unchanged* (*eval-usages g nid*) *g gup*
  using *add-node-fake.simps add-node-unchanged assms* **by** *blast*

lemma *dom-add-unchanged*:
  assumes *nid* ∈ *ids g*
  assumes *g′* = *add-node-fake n k g*
  assumes *nid* ≠ *n*
  shows *nid* ∈ *ids g′*
  using *add-changed assms*(*1*) *assms*(*2*) *assms*(*3*) **by** *force*

lemma *preserve-wf*:
  assumes *wf*: *wf-graph g*
  assumes *nid* ∉ *ids g*
  assumes *closed*: *inputs g′ nid* ∪ *succ g′ nid* ⊆ *ids g*
  assumes *g′*: *g′* = *add-node-fake nid k g*
  shows *wf-graph g′*
  using *assms* **unfolding** *wf-folds*
  apply (*intro conjI*)
      apply (*metis dom-add-unchanged*)
     apply (*metis add-node-unchanged-fake assms*(*1*) *kind-unchanged*)
  sorry

lemma *equal-closure-bisimilar*:
  assumes {*P′*. (*g m h* ⊢ *nid* ⇝ *P′*)} = {*P′*. (*g′ m h* ⊢ *nid* ⇝ *P′*)}
  shows *nid* . *g* ∼ *g′*

by (*metis assms weak-bisimilar.simps mem-Collect-eq*)

lemma *wf-size*:
  assumes *nid* ∈ *ids g*
  assumes *wf-graph g*
  assumes *is-AbstractEndNode* (*kind g nid*)
  shows *card* (*usages g nid*) > *0*
  using *assms* **unfolding** *wf-folds*
  by *fastforce*

lemma *sequentials-have-successors*:
  assumes *is-sequential-node n*
  shows *size* (*successors-of n*) > *0*
  using *assms* **by** (*cases n*; *auto*)

lemma *step-reaches-successors-only*:
  assumes (*g* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*))
  assumes *wf*: *wf-graph g*
  shows *nid′* ∈ *succ g nid* ∨ *nid′* ∈ *usages g nid*
  using *assms* **proof** (*induct* (*nid*, *m*, *h*) (*nid′*, *m*, *h*)*rule*: *step.induct*)
  case *SequentialNode*
  then show *?case* **using** *sequentials-have-successors*
    by (*metis nth-mem succ.simps*)
next
  case (*IfNode cond tb fb val*)
  then show *?case* **using** *successors-of-IfNode*
    by (*simp add*: *IfNode.hyps*(*1*))
next
  case (*EndNodes i phis inputs vs*)
  have *nid* ∈ *ids g*
    using *assms*(*1*) *step-in-ids*
    by *blast*
  then have *usage-size*: *card* (*usages g nid*) > *0*
    using *wf EndNodes*(*1*) *wf-size*
    by *blast*
  then have *usage-size*: *size* (*sorted-list-of-set* (*usages g nid*)) > *0*
    by (*metis length-sorted-list-of-set*)
  have *usages g nid* ⊆ *ids g*
    using *wf* **by** *fastforce*
  then have *finite-usage*: *finite* (*usages g nid*)
    by (*metis bot-nat-0.extremum-strict list.size*(*3*) *sorted-list-of-set.infinite usage-size*)
  from *EndNodes*(*2*) **have** *nid′* ∈ *usages g nid*
    unfolding *any-usage.simps*
    using *usage-size finite-usage*
    by (*metis hd-in-set length-greater-0-conv sorted-list-of-set*(*1*))
  then show *?case*
    by *simp*
next

case (*NewInstanceNode f obj ref*)
then show *?case* **using** *successors-of-NewInstanceNode* **by** *simp*
next
case (*LoadFieldNode f obj ref v*)
then show *?case* **by** *simp*
next
case (*SignedDivNode x y zero sb v1 v2 v*)
then show *?case* **by** *simp*
next
case (*SignedRemNode x y zero sb v1 v2 v*)
then show *?case* **by** *simp*
next
case (*StaticLoadFieldNode f v*)
then show *?case* **by** *simp*
next
case (*StoreFieldNode f newval uu obj val ref*)
then show *?case* **by** *simp*
next
case (*StaticStoreFieldNode f newval uv val*)
then show *?case* **by** *simp*
qed

lemma *stutter-closed*:
assumes $g\ m\ h \vdash nid \rightsquigarrow nid'$
assumes *wf-graph g*
shows $\exists\ n \in ids\ g\ .\ nid' \in succ\ g\ n \lor nid' \in usages\ g\ n$
using *assms*
proof (*induct nid nid' rule*: *stutter.induct*)
case (*StutterStep nid nid'*)
have $nid \in ids\ g$
using *StutterStep.hyps step-in-ids* **by** *blast*
then show *?case* **using** *StutterStep step-reaches-successors-only*
by *blast*
next
case (*Transitive nid nid'' nid'*)
then show *?case*
by *blast*
qed

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

shows $g' \vdash (nid, m, h) \rightarrow (nid', m, h)$
using *assms* **proof** (*induct* (nid, m, h) (nid', m, h) *rule*: *step.induct*)

case *SequentialNode*
  then show *?case*
    by (*metis step.SequentialNode*)
next
  case (*IfNode cond tb fb val*)
  then show *?case* **using** *stay-same step.IfNode*
    by (*metis* (*no-types*, *lifting*) *IRNodes.inputs-of-IfNode child-unchanged inputs.elims list.set-intros*(*1*))
next
  case (*EndNodes i phis inputs vs*)
  then show *?case* **sorry**
next
  case (*NewInstanceNode f obj ref*)
  then show *?case* **using** *step.NewInstanceNode*
    by *metis*
next
  case (*LoadFieldNode f obj ref v*)
  have *obj* ∈ *inputs g nid*
    using *LoadFieldNode*(*1*) *inputs-of-LoadFieldNode*
    using *opt-to-list.simps*
    by (*simp add*: *LoadFieldNode.hyps*(*1*))
  then have *unchanged* (*eval-usages g obj*) *g g′*
    using *unchanged*
    using *child-unchanged* **by** *blast*
  then have *g′ m* ⊢ *kind g′ obj* ↦ *ObjRef ref*
    using *unchanged wf stay-same*
    using *LoadFieldNode.hyps*(*2*) **by** *presburger*
  then show *?case* **using** *step.LoadFieldNode*
    by (*metis LoadFieldNode.hyps*(*1*) *LoadFieldNode.hyps*(*3*) *LoadFieldNode.hyps*(*4*) *assms*(*3*))
next
  case (*SignedDivNode x y zero sb v1 v2 v*)
  have *x* ∈ *inputs g nid*
    using *SignedDivNode*(*1*) *inputs-of-SignedDivNode*
    using *opt-to-list.simps*
    by (*simp add*: *SignedDivNode.hyps*(*1*))
  then have *unchanged* (*eval-usages g x*) *g g′*
    using *unchanged*
    using *child-unchanged* **by** *blast*
  then have *g′ m* ⊢ *kind g′ x* ↦ *v1*
    using *unchanged wf stay-same*
    using *SignedDivNode.hyps*(*2*) **by** *presburger*
  have *y* ∈ *inputs g nid*
    using *SignedDivNode*(*1*) *inputs-of-SignedDivNode*
    using *opt-to-list.simps*
    by (*simp add*: *SignedDivNode.hyps*(*1*))
  then have *unchanged* (*eval-usages g y*) *g g′*
    using *unchanged*
    using *child-unchanged* **by** *blast*

then have $g' \ m \vdash kind \ g' \ y \mapsto v2$
  using *unchanged wf stay-same*
  using *SignedDivNode.hyps*(*3*) **by** *presburger*
then show *?case* **using** *step.SignedDivNode*
 by (*metis SignedDivNode.hyps*(*1*) *SignedDivNode.hyps*(*4*) *SignedDivNode.hyps*(*5*)
$\langle g' \ m \vdash kind \ g' \ x \mapsto v1 \rangle$ *kind*)
next
case (*SignedRemNode x y zero sb v1 v2 v*)
have $x \in inputs \ g \ nid$
  using *SignedRemNode*(*1*) *inputs-of-SignedRemNode*
  using *opt-to-list.simps*
  by (*simp add*: *SignedRemNode.hyps*(*1*))
then have *unchanged* (*eval-usages g x*) $g \ g'$
  using *unchanged*
  using *child-unchanged* **by** *blast*
then have $g' \ m \vdash kind \ g' \ x \mapsto v1$
  using *unchanged wf stay-same*
  using *SignedRemNode.hyps*(*2*) **by** *presburger*
have $y \in inputs \ g \ nid$
  using *SignedRemNode*(*1*) *inputs-of-SignedRemNode*
  using *opt-to-list.simps*
  by (*simp add*: *SignedRemNode.hyps*(*1*))
then have *unchanged* (*eval-usages g y*) $g \ g'$
  using *unchanged*
  using *child-unchanged* **by** *blast*
then have $g' \ m \vdash kind \ g' \ y \mapsto v2$
  using *unchanged wf stay-same*
  using *SignedRemNode.hyps*(*3*) **by** *presburger*
then show *?case*
 by (*metis SignedRemNode.hyps*(*1*) *SignedRemNode.hyps*(*4*) *SignedRemNode.hyps*(*5*)
$\langle g' \ m \vdash kind \ g' \ x \mapsto v1 \rangle$ *kind step.SignedRemNode*)
next
case (*StaticLoadFieldNode f v*)
then show *?case* **using** *step.StaticLoadFieldNode*
  by *metis*
next
case (*StoreFieldNode f newval uu obj val ref*)
have $obj \in inputs \ g \ nid$
  using *StoreFieldNode*(*1*) *inputs-of-StoreFieldNode*
  using *opt-to-list.simps*
  by (*simp add*: *StoreFieldNode.hyps*(*1*))
then have *unchanged* (*eval-usages g obj*) $g \ g'$
  using *unchanged*
  using *child-unchanged* **by** *blast*
then have $g' \ m \vdash kind \ g' \ obj \mapsto ObjRef \ ref$
  using *unchanged wf stay-same*
  using *StoreFieldNode.hyps*(*3*) **by** *presburger*
have $newval \in inputs \ g \ nid$
  using *StoreFieldNode*(*1*) *inputs-of-StoreFieldNode*

using *opt-to-list.simps*
by (*simp add*: *StoreFieldNode.hyps*(*1*))
then have *unchanged* (*eval-usages g newval*) *g g'*
using *unchanged*
using *child-unchanged* **by** *blast*
then have g' $m \vdash$ *kind* g' *newval* $\mapsto$ *val*
using *unchanged wf stay-same*
using *StoreFieldNode.hyps*(*2*) **by** *blast*
then show *?case* **using** *step.StoreFieldNode*
by (*metis StoreFieldNode.hyps*(*1*) *StoreFieldNode.hyps*(*4*) *StoreFieldNode.hyps*(*5*) ‹g' $m \vdash$ *kind* g' *obj* $\mapsto$ *ObjRef ref*› *assms*(*3*))
next
case (*StaticStoreFieldNode f newval uv val*)
have *newval* $\in$ *inputs g nid*
using *StoreFieldNode*(*1*) *inputs-of-StoreFieldNode*
using *opt-to-list.simps*
by (*simp add*: *StaticStoreFieldNode.hyps*(*1*))
then have *unchanged* (*eval-usages g newval*) *g g'*
using *unchanged*
using *child-unchanged* **by** *blast*
then have g' $m \vdash$ *kind* g' *newval* $\mapsto$ *val*
using *unchanged wf stay-same*
using *StaticStoreFieldNode.hyps*(*2*) **by** *blast*
then show *?case* **using** *step.StaticStoreFieldNode*
by (*metis StaticStoreFieldNode.hyps*(*1*) *StaticStoreFieldNode.hyps*(*3*) *StaticStoreFieldNode.hyps*(*4*) *kind*)
qed

lemma *unchanged-closure*:
assumes *nid* $\notin$ *ids g*
assumes *wf*: *wf-graph g* $\wedge$ *wf-graph g'*
assumes g': g' = *add-node-fake nid k g*
assumes *nid'* $\in$ *ids g*
shows (g m $h \vdash$ *nid'* $\rightsquigarrow$ *nid''*) $\longleftrightarrow$ (g' m $h \vdash$ *nid'* $\rightsquigarrow$ *nid''*)
(**is** *?P* $\longleftrightarrow$ *?Q*)
proof
assume *P*: *?P*
have *niddiff*: *nid* $\neq$ *nid'*
using *assms*
by *blast*
from *P* **show** *?Q* **using** *assms niddiff*
proof (*induction rule*: *stutter.induct*)
case (*StutterStep start e*)
have *unchanged*: *unchanged* (*eval-usages g start*) *g g'*
using *StutterStep.prems*(*4*) *add-node-unchanged-fake assms*(*1*) g' *wf* **by** *blast*
have *succ-same*: *succ g start* = *succ g' start*
using *StutterStep.prems*(*4*) *kind-unchanged succ.simps unchanged* **by** *presburger*

have *kind g start* = *kind g′ start*
by (*metis StutterStep.prems*(*4*) *add-node-fake.elims add-node-unchanged assms*(*1*) *assms*(*2*) *g′ kind-unchanged*)
then have $g' \vdash (start, m, h) \to (e, m, h)$
using *unchanged-step wf unchanged succ-same*
by (*meson StutterStep.hyps*)
then show *?case*
using *stutter.StutterStep* **by** *blast*
next
case (*Transitive nid nid″ nid′*)
then show *?case*
by (*metis add-node-unchanged-fake kind-unchanged step-in-ids stutter.Transitive stutter.cases succ.simps unchanged-step*)
qed
next
assume *Q*: *?Q*
have *niddiff*: *nid* ≠ *nid′*
using *assms*
by *blast*
from *Q* **show** *?P* **using** *assms niddiff*
proof (*induction rule*: *stutter.induct*)
case (*StutterStep start e*)
have *eval-usages g′ start* ⊆ *eval-usages g start*
using *g′ eval-usages* **sorry**
then have *unchanged*: *unchanged* (*eval-usages g′ start*) *g′ g*
by (*smt* (*verit*, *ccfv-SIG*) *StutterStep.prems*(*4*) *add-node-unchanged-fake assms*(*1*) *g′ subset-iff unchanged.simps wf*)
have *succ-same*: *succ g start* = *succ g′ start*
using *StutterStep.prems*(*4*) *eval-usages-self node-unchanged succ.simps unchanged*
by (*metis* (*no-types*, *lifting*) *StutterStep.hyps step-in-ids*)
have *kind g start* = *kind g′ start*
by (*metis StutterStep.prems*(*4*) *add-node-fake.elims add-node-unchanged assms*(*1*) *assms*(*2*) *g′ kind-unchanged*)
then have $g \vdash (start, m, h) \to (e, m, h)$
using *StutterStep*(*1*) *wf unchanged-step unchanged succ-same*
sorry
then show *?case*
using *stutter.StutterStep* **by** *blast*
next
case (*Transitive nid nid″ nid′*)
then show *?case*
using *add-node-unchanged-fake kind-unchanged step-in-ids stutter.Transitive stutter.cases succ.simps unchanged-step*
sorry
qed
qed

fun *create-if* :: *IRGraph* ⇒ *ID* ⇒ *ID* ⇒ *ID* ⇒ *IRNode*

where
create-if g cond tb fb =
  (*case* (*kind g cond*) *of*
    ConstantNode condv ⇒
      RefNode (*if* (*val-to-bool condv*) *then tb else fb*) |
    - ⇒ (*if tb* = *fb then*
          RefNode tb
        else
          IfNode cond tb fb)
)

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

  shows *nid* . *gif* ∼ *gcreate*

proof –
  have *indep*: ¬(*eval-uses g cond nid*)
    using *cv eval-in-ids fresh no-external-use wf* **by** *blast*
  have *kind gif nid* = *IfNode cond tb fb*
    using *gif add-node-lookup* **by** *simp*
  then have {*cond*, *tb*, *fb*} = *inputs gif nid* ∪ *succ gif nid*
    using *inputs-of-IfNode successors-of-IfNode*
    by (*metis empty-set inputs.simps insert-is-Un list.simps*(*15*) *succ.simps*)
  then have *wf-gif*: *wf-graph gif*
    using *closed wf preserve-wf*
    using *fresh gif* **by** *presburger*
  have *create-if g cond tb fb* = *IfNode cond tb fb* ∨
        create-if g cond tb fb = *RefNode tb* ∨
        create-if g cond tb fb = *RefNode fb*
    by (*cases kind g cond*; *auto*)
  then have *kind gcreate nid* = *IfNode cond tb fb* ∨
        kind gcreate nid = *RefNode tb* ∨
        kind gcreate nid = *RefNode fb*
    using *gcreate add-node-lookup*
    using *add-node-lookup-fake fresh* **by** *presburger*
  then have *inputs gcreate nid* ∪ *succ gcreate nid* ⊆ {*cond*, *tb*, *fb*}
   using *inputs-of-IfNode successors-of-IfNode inputs-of-RefNode successors-of-RefNode*
    by *force*
  then have *wf-gcreate*: *wf-graph gcreate*
    using *closed wf preserve-wf fresh gcreate*
    by (*metis subset-trans*)
  have *tb-unchanged*: {*nid*′. (*gif m h* ⊢ *tb* ⇝ *nid*′)} = {*nid*′. (*gcreate m h* ⊢ *tb* ⇝

nid′)}
proof −
have ¬(∃ *n* ∈ *ids g*. *nid* ∈ *succ g n* ∨ *nid* ∈ *usages g n*)
using *wf*
by (*metis* (*no-types*, *lifting*) *fresh mem-Collect-eq subsetD usages.simps wf-folds*(*1*,*3*))
then have *nid* ∉ {*nid′*. (*g m h* ⊢ *tb* ⇝ *nid′*)}
using *wf stutter-closed*
by (*metis mem-Collect-eq*)
have *gif-set*: {*nid′*. (*gif m h* ⊢ *tb* ⇝ *nid′*)} = {*nid′*. (*g m h* ⊢ *tb* ⇝ *nid′*)}
using *unchanged-closure fresh wf gif closed wf-gif*
by *blast*
have *gcreate-set*: {*nid′*. (*gcreate m h* ⊢ *tb* ⇝ *nid′*)} = {*nid′*. (*g m h* ⊢ *tb* ⇝ *nid′*)}
using *unchanged-closure fresh wf gcreate closed wf-gcreate*
by *blast*
from *gif-set gcreate-set* **show** *?thesis* **by** *simp*
qed
have *fb-unchanged*: {*nid′*. (*gif m h* ⊢ *fb* ⇝ *nid′*)} = {*nid′*. (*gcreate m h* ⊢ *fb* ⇝ *nid′*)}
proof −
have ¬(∃ *n* ∈ *ids g*. *nid* ∈ *succ g n* ∨ *nid* ∈ *usages g n*)
using *wf*
by (*metis* (*no-types*, *lifting*) *fresh mem-Collect-eq subsetD usages.simps wf-folds*(*1*,*3*))
then have *nid* ∉ {*nid′*. (*g m h* ⊢ *fb* ⇝ *nid′*)}
using *wf stutter-closed*
by (*metis mem-Collect-eq*)
have *gif-set*: {*nid′*. (*gif m h* ⊢ *fb* ⇝ *nid′*)} = {*nid′*. (*g m h* ⊢ *fb* ⇝ *nid′*)}
using *unchanged-closure fresh wf gif closed wf-gif*
by *blast*
have *gcreate-set*: {*nid′*. (*gcreate m h* ⊢ *fb* ⇝ *nid′*)} = {*nid′*. (*g m h* ⊢ *fb* ⇝ *nid′*)}
using *unchanged-closure fresh wf gcreate closed wf-gcreate*
by *blast*
from *gif-set gcreate-set* **show** *?thesis* **by** *simp*
qed
show *?thesis*
proof (*cases* ∃ *val* . (*kind g cond*) = *ConstantNode val*)
let *?gif-closure* = {*P′*. (*gif m h* ⊢ *nid* ⇝ *P′*)}
let *?gcreate-closure* = {*P′*. (*gcreate m h* ⊢ *nid* ⇝ *P′*)}
case *constantCond*: *True*
obtain *val* **where** *val*: (*kind g cond*) = *ConstantNode val*
using *constantCond* **by** *blast*
then show *?thesis*
proof (*cases val-to-bool val*)
case *constantTrue*: *True*
have *if-kind*: *kind gif nid* = (*IfNode cond tb fb*)
using *gif add-node-lookup* **by** *simp*

have *if-cv*: *gif m* ⊢ (*kind gif cond*) ↦ *val*
by (*metis ConstantNodeE add-node-unchanged-fake cv eval-in-ids fresh gif stay-same val wf*)
have (*gif* ⊢ (*nid*, *m*, *h*) → (*tb*, *m*, *h*))
using *step.IfNode if-kind if-cv*
using *constantTrue* **by** *presburger*
then have *gif-closure*: *?gif-closure* = {*tb*} ∪ {*nid'*. (*gif m h* ⊢ *tb* ⇝ *nid'*)}
using *stuttering-successor* **by** *presburger*
have *ref-kind*: *kind gcreate nid* = (*RefNode tb*)
using *gcreate add-node-lookup constantTrue constantCond* **unfolding** *create-if.simps*
by (*simp add*: *val*)
have (*gcreate* ⊢ (*nid*, *m*, *h*) → (*tb*, *m*, *h*))
using *stepRefNode ref-kind* **by** *simp*
then have *gcreate-closure*: *?gcreate-closure* = {*tb*} ∪ {*nid'*. (*gcreate m h* ⊢ *tb* ⇝ *nid'*)}
using *stuttering-successor*
by *auto*
from *gif-closure gcreate-closure* **have** *?gif-closure* = *?gcreate-closure*
using *tb-unchanged* **by** *simp*
then show *?thesis*
using *equal-closure-bisimilar* **by** *simp*
next
case *constantFalse*: *False*
have *if-kind*: *kind gif nid* = (*IfNode cond tb fb*)
using *gif add-node-lookup* **by** *simp*
have *if-cv*: *gif m* ⊢ (*kind gif cond*) ↦ *val*
by (*metis ConstantNodeE add-node-unchanged-fake cv eval-in-ids fresh gif stay-same val wf*)
have (*gif* ⊢ (*nid*, *m*, *h*) → (*fb*, *m*, *h*))
using *step.IfNode if-kind if-cv*
using *constantFalse* **by** *presburger*
then have *gif-closure*: *?gif-closure* = {*fb*} ∪ {*nid'*. (*gif m h* ⊢ *fb* ⇝ *nid'*)}
using *stuttering-successor* **by** *presburger*
have *ref-kind*: *kind gcreate nid* = *RefNode fb*
using *add-node-lookup-fake constantFalse fresh gcreate val* **by** *force*
then have (*gcreate* ⊢ (*nid*, *m*, *h*) → (*fb*, *m*, *h*))
using *stepRefNode* **by** *presburger*
then have *gcreate-closure*: *?gcreate-closure* = {*fb*} ∪ {*nid'*. (*gcreate m h* ⊢ *fb* ⇝ *nid'*)}
using *stuttering-successor* **by** *presburger*
from *gif-closure gcreate-closure* **have** *?gif-closure* = *?gcreate-closure*
using *fb-unchanged* **by** *simp*
then show *?thesis*
using *equal-closure-bisimilar* **by** *simp*
qed
next
let *?gif-closure* = {*P'*. (*gif m h* ⊢ *nid* ⇝ *P'*)}
let *?gcreate-closure* = {*P'*. (*gcreate m h* ⊢ *nid* ⇝ *P'*)}

case *notConstantCond*: *False*
then show *?thesis*
proof (*cases tb* = *fb*)
case *equalBranches*: *True*
have *if-kind*: *kind gif nid* = (*IfNode cond tb fb*)
using *gif add-node-lookup* **by** *simp*
have (*gif* ⊢ (*nid*, *m*, *h*) → (*tb*, *m*, *h*)) ∨ (*gif* ⊢ (*nid*, *m*, *h*) → (*fb*, *m*, *h*))
using *step.IfNode if-kind cv* **apply** (*cases val-to-bool cv*)
apply (*metis add-node-fake.simps add-node-unchanged eval-in-ids fresh gif stay-same wf*)
by (*metis add-node-unchanged-fake eval-in-ids fresh gif stay-same wf*)
then have *gif-closure*: *?gif-closure* = {*tb*} ∪ {*nid′*. (*gif m h* ⊢ *tb* ⇝ *nid′*)}
using *equalBranches*
using *stuttering-successor* **by** *presburger*
have *iref-kind*: *kind gcreate nid* = (*RefNode tb*)
using *gcreate add-node-lookup notConstantCond equalBranches*
unfolding *create-if.simps*
by (*cases* (*kind g cond*); *auto*)
then have (*gcreate* ⊢ (*nid*, *m*, *h*) → (*tb*, *m*, *h*))
using *stepRefNode* **by** *simp*
then have *gcreate-closure*: *?gcreate-closure* = {*tb*} ∪ {*nid′*. (*gcreate m h* ⊢ *tb* ⇝ *nid′*)}
using *stuttering-successor* **by** *presburger*
from *gif-closure gcreate-closure* **have** *?gif-closure* = *?gcreate-closure*
using *tb-unchanged* **by** *simp*
then show *?thesis*
using *equal-closure-bisimilar* **by** *simp*
next
case *uniqueBranches*: *False*
let *?tb-closure* = {*tb*} ∪ {*nid′*. (*gif m h* ⊢ *tb* ⇝ *nid′*)}
let *?fb-closure* = {*fb*} ∪ {*nid′*. (*gif m h* ⊢ *fb* ⇝ *nid′*)}
have *if-kind*: *kind gif nid* = (*IfNode cond tb fb*)
using *gif add-node-lookup* **by** *simp*
have *if-step*: (*gif* ⊢ (*nid*, *m*, *h*) → (*tb*, *m*, *h*)) ∨ (*gif* ⊢ (*nid*, *m*, *h*) → (*fb*, *m*, *h*))
using *step.IfNode if-kind cv* **apply** (*cases val-to-bool cv*)
apply (*metis add-node-fake.simps add-node-unchanged eval-in-ids fresh gif stay-same wf*)
by (*metis add-node-unchanged-fake eval-in-ids fresh gif stay-same wf*)
then have *gif-closure*: *?gif-closure* = *?tb-closure* ∨ *?gif-closure* = *?fb-closure*
using *stuttering-successor* **by** *presburger*
have *gc-kind*: *kind gcreate nid* = (*IfNode cond tb fb*)
using *gcreate add-node-lookup notConstantCond uniqueBranches*
unfolding *create-if.simps*
by (*cases* (*kind g cond*); *auto*)
then have (*gcreate* ⊢ (*nid*, *m*, *h*) → (*tb*, *m*, *h*)) ∨ (*gcreate* ⊢ (*nid*, *m*, *h*) → (*fb*, *m*, *h*))
by (*metis add-node-lookup-fake fresh gcreate gif if-step*)
then have *gcreate-closure*: *?gcreate-closure* = *?tb-closure* ∨ *?gcreate-closure* =

?fb-closure
    by (*metis add-node-lookup-fake fresh gc-kind gcreate gif gif-closure*)
  from *gif-closure gcreate-closure* **have** *?gif-closure* = *?gcreate-closure*
    using *tb-unchanged fb-unchanged*
    by (*metis add-node-lookup-fake fresh gc-kind gcreate gif*)
  then show *?thesis*
    using *equal-closure-bisimilar* **by** *simp*
 qed
qed
qed

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

proof (*cases* ∃ *val* . (*kind g cond*) = *ConstantNode val*)
 case *True*
 show *?thesis*
 proof −
  obtain *val* **where** *val*: (*kind g cond*) = *ConstantNode val*
    using *True* **by** *blast*
  have *cond-exists*: $cond \in ids\ g$
    using *cv eval-in-ids* **by** *auto*
  have *if-kind*: *kind gif nid* = (*IfNode cond tb fb*)
    using *gif add-node-lookup* **by** *simp*
  have *if-cv*: $gif\ m \vdash (kind\ gif\ cond) \mapsto val$
    using *step.IfNode if-kind*
    using *True eval.ConstantNode gif fresh*
    using *stay-same cond-exists*
    using *val*
    using *add-node.rep-eq kind.rep-eq* **by** *auto*
  have *if-step*: $gif \vdash (nid,m,h) \rightarrow$ (*if val-to-bool val then tb else fb*,*m*,*h*)
  proof −
    show *?thesis* **using** *step.IfNode if-kind if-cv*
      by (*simp*)
  qed
  have *create-step*: $gcreate \vdash (nid,m,h) \rightarrow$ (*if val-to-bool val then tb else fb*,*m*,*h*)
  proof −
    have *create-kind*: *kind gcreate nid* = (*create-if g cond tb fb*)
      using *gcreate add-node-lookup-fake*
      using *fresh* **by** *blast*
     have *create-fun*: *create-if g cond tb fb* = *RefNode* (*if val-to-bool val then tb else fb*)
      using *True create-kind val* **by** *simp*
    show *?thesis* **using** *stepRefNode create-kind create-fun if-cv*

```
        by (simp)
      qed
      then show ?thesis using StutterStep create-step if-step
        by blast
    qed
  next
    case not-const: False
    obtain nid′ where nid′ = (if val-to-bool cv then tb else fb)
      by blast
   have nid-eq: (gif ⊢ (nid,m,h) → (nid′,m,h)) ∧ (gcreate ⊢ (nid,m,h) → (nid′,m,h))
    proof −
      have indep: ¬(eval-uses g cond nid)
        using no-external-use
        using cv eval-in-ids fresh wf by blast
      have nid′: nid′ = (if val-to-bool cv then tb else fb)
        by (simp add: ‹nid′ = (if val-to-bool cv then tb else fb)›)
      have gif-kind: kind gif nid = (IfNode cond tb fb)
        using add-node-lookup-fake gif
        using fresh by blast
      then have nid ≠ cond
        using cv fresh indep
        using eval-in-ids by blast
      have unchanged (eval-usages g cond) g gif
        using gif add-node-unchanged-fake
        using cv eval-in-ids fresh wf by blast
      then obtain cv2 where cv2: gif m ⊢ (kind gif cond) ↦ cv2
        using cv gif wf stay-same by blast
      then have cv = cv2
        using indep gif cv
        using ‹nid ≠ cond›
        using fresh
        using ‹unchanged (eval-usages g cond) g gif› evalDet stay-same wf by blast
      then have eval-gif: (gif ⊢ (nid,m,h) → (nid′,m,h))
        using step.IfNode gif-kind nid′ cv2
        by auto
      have gcreate-kind: kind gcreate nid = (create-if g cond tb fb)
        using gcreate add-node-lookup-fake
        using fresh by blast
      have eval-gcreate: gcreate ⊢ (nid,m,h) → (nid′,m,h)
      proof (cases tb = fb)
        case True
        have create-if g cond tb fb = RefNode tb
          using not-const True by (cases (kind g cond); auto)
        then show ?thesis
          using True gcreate-kind nid′ stepRefNode
          by (simp)
      next
        case False
        have create-if g cond tb fb = IfNode cond tb fb
```

using *not-const False* **by** (*cases* (*kind g cond*); *auto*)
then show *?thesis*
using *eval-gif gcreate gif*
using *IfNode* ‹*cv* = *cv2*› *cv2 gif-kind nid'* **by** *auto*
qed
show *?thesis*
using *eval-gcreate eval-gif StutterStep* **by** *blast*
qed
show *?thesis* **using** *nid-eq StutterStep* **by** *meson*
qed

end