Veriopt

July 3, 2021

Abstract

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

Contents

1 Runtime Values and Arithmetic

theory *Values2*
imports
HOL−Library.Word
HOL−Library.Signed-Division
HOL−Library.Float
HOL−Library.LaTeXsugar
begin

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

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

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

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

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

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

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

type-synonym *objref* = *nat option*

datatype *Value* =
UndefVal |
IntVal32 int32 |
IntVal64 int64 |
FloatVal float |
ObjRef objref |
ObjStr string

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

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

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

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

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

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

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

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

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

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

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

fun *intval-add* :: *Value* ⇒ *Value* ⇒ *Value* **where**
  intval-add (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1*+*v2*)) |
  intval-add (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1*+*v2*)) |
  intval-add - - = *UndefVal*

instantiation *Value* :: *plus*
begin

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

instance proof qed
end

fun *intval-sub* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-sub (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1*−*v2*)) |
intval-sub (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1*−*v2*)) |
intval-sub - - = *UndefVal*

instantiation *Value* :: *minus*
begin

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

instance proof qed
end

fun *intval-mul* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-mul (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1*∗*v2*)) |
intval-mul (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1*∗*v2*)) |
intval-mul - - = *UndefVal*

instantiation *Value* :: *times*
begin

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

instance proof qed
end

fun *intval-div* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-div (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*word-of-int*((*sint v1*) *sdiv* (*sint v2*)))) |
intval-div (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*word-of-int*((*sint v1*) *sdiv* (*sint v2*)))) |
intval-div - - = *UndefVal*

instantiation *Value* :: *divide*
begin

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

instance proof qed
end

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

intval-mod (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*word-of-int*((*sint v1*) *smod* (*sint v2*)))) |
intval-mod (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*word-of-int*((*sint v1*) *smod* (*sint v2*)))) |
intval-mod - - = *UndefVal*

instantiation *Value* :: *modulo*
begin

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

instance proof qed
end

fun *intval-and* :: *Value* ⇒ *Value* ⇒ *Value* (**infix** &&∗ *64*) **where**
intval-and (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1 AND v2*)) |
intval-and (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1 AND v2*)) |
intval-and - - = *UndefVal*

fun *intval-or* :: *Value* ⇒ *Value* ⇒ *Value* (**infix** ||∗ *59*) **where**
intval-or (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1 OR v2*)) |
intval-or (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1 OR v2*)) |
intval-or - - = *UndefVal*

fun *intval-xor* :: *Value* ⇒ *Value* ⇒ *Value* (**infix** ^∗ *59*) **where**
intval-xor (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1 XOR v2*)) |
intval-xor (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1 XOR v2*)) |
intval-xor - - = *UndefVal*

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

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

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

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

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-sym*:
  shows *intval-add a b* = *intval-add b a*
  by (*induction a*; *induction b*; *auto*)

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

lemma *intval-bad1* [*simp*]: *intval-add* (*IntVal32 x*) (*IntVal64 y*) = *UndefVal*
  by *auto*
lemma *intval-bad2* [*simp*]: *intval-add* (*IntVal64 x*) (*IntVal32 y*) = *UndefVal*
  by *auto*

lemma *intval-assoc*: *intval-add32* (*intval-add32 x y*) *z* = *intval-add32 x* (*intval-add32 y z*)
  apply (*induction x*)
      apply *auto*
   apply (*induction y*)
      apply *auto*
    apply (*induction z*)
  by *auto*

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

lemma *intval-add* (*IntVal32* ($2^{31}-1$)) (*IntVal32* ($2^{31}-1$)) = *IntVal32* (-2)
  by *eval*

lemma *intval-add* (*IntVal64* ($2\hat{}31-1$)) (*IntVal64* ($2\hat{}31-1$)) = *IntVal64* *4294967294*
by *eval*

end

2 Nodes

2.1 Types of Nodes

theory *IRNodes2*
imports
Values2
begin

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

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

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

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

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

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

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

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

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

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

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

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

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

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

fun *inputs-of* :: *IRNode* ⇒ *ID list* **where**
inputs-of-AbsNode:
inputs-of (*AbsNode value*) = [*value*] |
inputs-of-AddNode:
inputs-of (*AddNode x y*) = [*x*, *y*] |
inputs-of-AndNode:
inputs-of (*AndNode x y*) = [*x*, *y*] |
inputs-of-BeginNode:
inputs-of (*BeginNode next*) = [] |
inputs-of-BytecodeExceptionNode:
inputs-of (*BytecodeExceptionNode arguments stateAfter next*) = *arguments* @ (*opt-to-list stateAfter*) |
inputs-of-ConditionalNode:
inputs-of (*ConditionalNode condition trueValue falseValue*) = [*condition*, *trueValue*, *falseValue*] |
inputs-of-ConstantNode:
inputs-of (*ConstantNode const*) = [] |
inputs-of-DynamicNewArrayNode:
inputs-of (*DynamicNewArrayNode elementType length0 voidClass stateBefore next*) = [*elementType*, *length0*] @ (*opt-to-list voidClass*) @ (*opt-to-list stateBefore*) |

inputs-of-EndNode:
inputs-of (*EndNode*) = [] |
inputs-of-ExceptionObjectNode:
inputs-of (*ExceptionObjectNode stateAfter next*) = (*opt-to-list stateAfter*) |
inputs-of-FrameState:
inputs-of (*FrameState monitorIds outerFrameState values virtualObjectMappings*) = *monitorIds* @ (*opt-to-list outerFrameState*) @ (*opt-list-to-list values*) @ (*opt-list-to-list virtualObjectMappings*) |
inputs-of-IfNode:
inputs-of (*IfNode condition trueSuccessor falseSuccessor*) = [*condition*] |
inputs-of-IntegerEqualsNode:
inputs-of (*IntegerEqualsNode x y*) = [*x*, *y*] |
inputs-of-IntegerLessThanNode:
inputs-of (*IntegerLessThanNode x y*) = [*x*, *y*] |
inputs-of-InvokeNode:
inputs-of (*InvokeNode nid0 callTarget classInit stateDuring stateAfter next*) = *callTarget* # (*opt-to-list classInit*) @ (*opt-to-list stateDuring*) @ (*opt-to-list stateAfter*) |
inputs-of-InvokeWithExceptionNode:
inputs-of (*InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter next exceptionEdge*) = *callTarget* # (*opt-to-list classInit*) @ (*opt-to-list stateDuring*) @ (*opt-to-list stateAfter*) |
inputs-of-IsNullNode:
inputs-of (*IsNullNode value*) = [*value*] |
inputs-of-KillingBeginNode:
inputs-of (*KillingBeginNode next*) = [] |
inputs-of-LoadFieldNode:
inputs-of (*LoadFieldNode nid0 field object next*) = (*opt-to-list object*) |
inputs-of-LogicNegationNode:
inputs-of (*LogicNegationNode value*) = [*value*] |
inputs-of-LoopBeginNode:
inputs-of (*LoopBeginNode ends overflowGuard stateAfter next*) = *ends* @ (*opt-to-list overflowGuard*) @ (*opt-to-list stateAfter*) |
inputs-of-LoopEndNode:
inputs-of (*LoopEndNode loopBegin*) = [*loopBegin*] |
inputs-of-LoopExitNode:
inputs-of (*LoopExitNode loopBegin stateAfter next*) = *loopBegin* # (*opt-to-list stateAfter*) |
inputs-of-MergeNode:
inputs-of (*MergeNode ends stateAfter next*) = *ends* @ (*opt-to-list stateAfter*) |
inputs-of-MethodCallTargetNode:
inputs-of (*MethodCallTargetNode targetMethod arguments*) = *arguments* |
inputs-of-MulNode:
inputs-of (*MulNode x y*) = [*x*, *y*] |
inputs-of-NegateNode:
inputs-of (*NegateNode value*) = [*value*] |
inputs-of-NewArrayNode:
inputs-of (*NewArrayNode length0 stateBefore next*) = *length0* # (*opt-to-list stateBefore*) |

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

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

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

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

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

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

lemma *inputs-of* (*FrameState x* (*Some y*) (*Some z*) *None*) = *x* @ [*y*] @ *z*
unfolding *inputs-of-FrameState* **by** *simp*
lemma *successors-of* (*FrameState x* (*Some y*) (*Some z*) *None*) = []

unfolding *inputs-of-FrameState* **by** *simp*

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

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

end

2.2 Hierarchy of Nodes

theory *IRNodeHierarchy*
imports *IRNodes2*
begin

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

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

These functions have been automatically generated from the compiler.

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

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

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

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

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

fun *is-AbstractNewArrayNode* :: *IRNode* ⇒ *bool* **where**

$is\text{-}AbstractNewArrayNode\ n = ((is\text{-}DynamicNewArrayNode\ n) \vee (is\text{-}NewArrayNode\ n))$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

n))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

fun *is-GuardingNode* :: *IRNode* ⇒ *bool* **where**

is-GuardingNode n = ((*is-AbstractBeginNode* n))

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

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

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

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

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

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

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

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

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

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

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

fun *is-Binary* :: *IRNode* ⇒ *bool* **where**

$is\text{-}Binary\ n = ((is\text{-}BinaryArithmeticNode\ n) \lor (is\text{-}BinaryNode\ n) \lor (is\text{-}BinaryOpLogicNode\ n) \lor (is\text{-}CompareNode\ n) \lor (is\text{-}FixedBinaryNode\ n) \lor (is\text{-}ShortCircuitOrNode\ n))$

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

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

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

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

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

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

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*)) ∨

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

end

3 Stamp Typing

theory *Stamp2*
imports *Values2*
begin

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

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

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

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

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

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

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

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

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

  empty-stamp (*KlassPointerStamp nonNull alwaysNull*) = (*KlassPointerStamp*

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

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

is-stamp-empty x = *False*

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

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

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

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

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

— In certain circumstances a stamp provides enough information to evaluate a value as a stamp, the asConstant function converts the stamp to a value where one can be inferred.

fun *asConstant* :: *Stamp* ⇒ *Value* **where**
asConstant (*IntegerStamp b l h*) = (*if l* = *h then IntVal64* (*word-of-int l*) *else UndefVal*) |
asConstant - = *UndefVal*

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

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

fun *constantAsStamp* :: *Value* ⇒ *Stamp* **where**
constantAsStamp (*IntVal32 v*) = (*IntegerStamp* (*nat 32*) (*sint v*) (*sint v*)) |
constantAsStamp (*IntVal64 v*) = (*IntegerStamp* (*nat 64*) (*sint v*) (*sint v*)) |

constantAsStamp - = *IllegalStamp*

— Define when a runtime value is valid for a stamp
fun *valid-value* :: *Stamp* ⇒ *Value* ⇒ *bool* **where**
valid-value (*IntegerStamp b l h*) (*IntVal32 v*) = (*b*=*32* ∧ (*sint v* ≥ *l*) ∧ (*sint v* ≤ *h*)) |
valid-value (*IntegerStamp b l h*) (*IntVal64 v*) = (*b*=*64* ∧ (*sint v* ≥ *l*) ∧ (*sint v* ≤ *h*)) |

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

— The most common type of stamp within the compiler (apart from the Void-

Stamp) is a 32 bit integer stamp with an unrestricted range. We use *default-stamp* as it is a frequently used stamp.

definition *default-stamp* :: *Stamp* **where**
  default-stamp = (*unrestricted-stamp* (*IntegerStamp 32 0 0*))

end

4 Graph Representation

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

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

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

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

setup-lifting *type-definition-IRGraph*

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

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

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

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

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

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

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

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

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

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

code-datatype *irgraph*

fun *filter-none* **where**
  filter-none g = {*nid* ∈ *dom g* . ∄ *s. g nid* = (*Some* (*NoNode*, *s*))}

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

lemma *dom-eq*:
  assumes ∀ *x* ∈ *set xs. fst* (*snd x*) ≠ *NoNode*
  shows *filter-none* (*map-of xs*) = *dom* (*map-of xs*)
  unfolding *filter-none.simps* **using** *assms map-of-SomeD*
  by *fastforce*

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

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

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

— Get the inputs set of a given node ID
fun *inputs* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
  inputs g nid = *set* (*inputs-of* (*kind g nid*))
— Get the successor set of a given node ID

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

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

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

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

lemma *ids-some*[*simp*]: *x* ∈ *ids g* ⟷ *kind g x* ≠ *NoNode*
proof −
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*) $\longrightarrow$ *ids* (*Abs-IRGraph g*) = {*nid* $\in$ *dom g* . $\nexists$ *s*. *g nid* = *Some* (*NoNode*, *s*)}
  using *ids.rep-eq* **by** *simp*

lemma [*simp*]: *finite* (*dom g*) $\longrightarrow$ *kind* (*Abs-IRGraph g*) = (λ*x* . (*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*) = (λ*x* . (*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-node nid g* $\longrightarrow$ *kind gup nid* = *NoNode* $\wedge$ *stamp gup nid* = *IllegalStamp*
  by (*simp add*: *kind.rep-eq remove-node.rep-eq stamp.rep-eq*)

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

lemma *replace-node-unchanged*:
  gup = *replace-node nid* (*k*, *s*) *g* $\longrightarrow$ ($\forall$ *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 *IRTreeEval*
  imports
    Graph.IRGraph
begin

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

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

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

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

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

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

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

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

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

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

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

fun *input-index* :: *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *ID* $\Rightarrow$ *nat* **where**
input-index g n n' = *find-index n'* (*inputs-of* (*kind g n*))

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

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

fun *find-node-and-stamp* :: *IRGraph* $\Rightarrow$ (*IRNode* $\times$ *Stamp*) $\Rightarrow$ *ID option* **where**
find-node-and-stamp g (*n*,*s*) =
find (λi. *kind g i* = *n* $\wedge$ *stamp g i* = *s*) (*sorted-list-of-set*(*ids g*))

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

5.1 Data-flow Tree Representation

datatype *IRUnaryOp* =
UnaryAbs
| *UnaryNeg*
| *UnaryNot*
| *UnaryLogicNegation*

datatype *IRBinaryOp* =
BinAdd
| *BinMul*
| *BinSub*
| *BinAnd*
| *BinOr*
| *BinXor*
| *BinIntegerEquals*
| *BinIntegerLessThan*

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

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

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

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

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

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

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

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

AbsNode:
⟦*kind g n* = *AbsNode x*;

$g \vdash x \triangleright xe\rrbracket$
$\Longrightarrow g \vdash n \triangleright (UnaryExpr\ UnaryAbs\ xe)$ |

NotNode:
$\llbracket kind\ g\ n = NotNode\ x;$
$g \vdash x \triangleright xe\rrbracket$
$\Longrightarrow g \vdash n \triangleright (UnaryExpr\ UnaryNot\ xe)$ |

NegateNode:
$\llbracket kind\ g\ n = NegateNode\ x;$
$g \vdash x \triangleright xe\rrbracket$
$\Longrightarrow g \vdash n \triangleright (UnaryExpr\ UnaryNeg\ xe)$ |

LogicNegationNode:
$\llbracket kind\ g\ n = LogicNegationNode\ x;$
$g \vdash x \triangleright xe\rrbracket$
$\Longrightarrow g \vdash n \triangleright (UnaryExpr\ UnaryLogicNegation\ xe)$ |

AddNode:
$\llbracket kind\ g\ n = AddNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye\rrbracket$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinAdd\ xe\ ye)$ |

MulNode:
$\llbracket kind\ g\ n = MulNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye\rrbracket$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinMul\ xe\ ye)$ |

SubNode:
$\llbracket kind\ g\ n = SubNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye\rrbracket$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinSub\ xe\ ye)$ |

AndNode:
$\llbracket kind\ g\ n = AndNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye\rrbracket$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinAnd\ xe\ ye)$ |

OrNode:
$\llbracket kind\ g\ n = OrNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye\rrbracket$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinOr\ xe\ ye)$ |

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

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

$IntegerLessThanNode$:
$[\![kind\ g\ n = IntegerLessThanNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye]\!]$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinIntegerLessThan\ xe\ ye)\ |$

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

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

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

$RepNil$:
$g \vdash [] \triangleright_L []\ |$

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

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

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

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

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

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

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

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

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

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

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

stamp-unary op - = *IllegalStamp*

fun *stamp-binary* :: *IRBinaryOp* ⇒ *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**
stamp-binary op (*IntegerStamp b1 lo1 hi1*) (*IntegerStamp b2 lo2 hi2*) =
(*if* (*b1* = *b2*) *then unrestricted-stamp* (*IntegerStamp b1 lo1 hi1*) *else IllegalStamp*) |

stamp-binary op - - = *IllegalStamp*

fun *stamp-expr* :: *IRExpr* ⇒ *Stamp* **where**
stamp-expr (*UnaryExpr op x*) = *stamp-unary op* (*stamp-expr x*) |
stamp-expr (*BinaryExpr bop x y*) = *stamp-binary bop* (*stamp-expr x*) (*stamp-expr y*) |
stamp-expr (*ConstantExpr val*) = *constantAsStamp val* |
stamp-expr (*LeafExpr i s*) = *s* |
stamp-expr (*ParameterExpr i s*) = *s* |
stamp-expr (*ConditionalExpr c t f*) = *meet* (*stamp-expr t*) (*stamp-expr f*)

export-code *stamp-unary stamp-binary stamp-expr*

fun *unary-node* :: *IRUnaryOp* ⇒ *ID* ⇒ *IRNode* **where**
unary-node UnaryAbs v = *AbsNode v* |
unary-node UnaryNot v = *NotNode v* |
unary-node UnaryNeg v = *NegateNode v* |
unary-node UnaryLogicNegation v = *LogicNegationNode v*

fun *bin-node* :: *IRBinaryOp* ⇒ *ID* ⇒ *ID* ⇒ *IRNode* **where**
bin-node BinAdd x y = *AddNode x y* |

bin-node BinMul x y = MulNode x y |
bin-node BinSub x y = SubNode x y |
bin-node BinAnd x y = AndNode x y |
bin-node BinOr x y = OrNode x y |
bin-node BinXor x y = XorNode x y |
bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
bin-node BinIntegerLessThan x y = IntegerLessThanNode x y

fun *unary-eval* :: *IRUnaryOp* ⇒ *Value* ⇒ *Value* **where**
unary-eval UnaryAbs (*IntVal32 v1*) = *IntVal32* ((*if sint*(*v1*) < *0 then* − *v1 else v1*)) |
unary-eval UnaryAbs (*IntVal64 v1*) = *IntVal64* ((*if sint*(*v1*) < *0 then* − *v1 else v1*)) |

unary-eval UnaryNot (*IntVal32 v1*) = *IntVal32* (*NOT v1*) |
unary-eval UnaryNot (*IntVal64 v1*) = *IntVal64* (*NOT v1*) |

unary-eval UnaryLogicNegation (*IntVal32 v1*) = (*if v1* = *0 then* (*IntVal32 1*) *else* (*IntVal32 0*)) |

unary-eval UnaryNeg v = *intval-negate v* |

unary-eval op v1 = *UndefVal*

fun *bin-eval* :: *IRBinaryOp* ⇒ *Value* ⇒ *Value* ⇒ *Value* **where**
bin-eval BinAdd v1 v2 = *intval-add v1 v2* |
bin-eval BinMul v1 v2 = *intval-mul v1 v2* |
bin-eval BinSub v1 v2 = *intval-sub v1 v2* |
bin-eval BinAnd v1 v2 = *intval-and v1 v2* |
bin-eval BinOr v1 v2 = *intval-or v1 v2* |
bin-eval BinXor v1 v2 = *intval-xor v1 v2* |
bin-eval BinIntegerEquals v1 v2 = *intval-equals v1 v2* |
bin-eval BinIntegerLessThan v1 v2 = *intval-less-than v1 v2*

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

code-pred *fresh-id* **.**

fun *get-fresh-id* :: *IRGraph* ⇒ *ID* **where**

get-fresh-id g = *last*(*sorted-list-of-set*(*ids g*)) + *1*

export-code *get-fresh-id*

value *get-fresh-id eg2-sq*

value *get-fresh-id* (*add-node 6* (*ParameterNode 2*, *default-stamp*) *eg2-sq*)

inductive
unrep :: *IRGraph* $\Rightarrow$ *IRExpr* $\Rightarrow$ (*IRGraph* $\times$ *ID*) $\Rightarrow$ *bool* (- $\triangleleft$ - $\rightsquigarrow$ - *55*)
and
unrepList :: *IRGraph* $\Rightarrow$ *IRExpr list* $\Rightarrow$ (*IRGraph* $\times$ *ID list*) $\Rightarrow$ *bool* (- $\triangleleft_L$ - $\rightsquigarrow$ - *55*)
where

ConstantNodeSame:
$\llbracket$*find-node-and-stamp g* (*ConstantNode c*, *constantAsStamp c*) = *Some nid*$\rrbracket$
$\Longrightarrow$ *g* $\triangleleft$ (*ConstantExpr c*) $\rightsquigarrow$ (*g*, *nid*) |

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

ParameterNodeSame:
$\llbracket$*find-node-and-stamp g* (*ParameterNode i*, *s*) = *Some nid*$\rrbracket$
$\Longrightarrow$ *g* $\triangleleft$ (*ParameterExpr i s*) $\rightsquigarrow$ (*g*, *nid*) |

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

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

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

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

$UnaryNodeNew$:
$[\![g \triangleleft xe \rightsquigarrow (g2,\ x);$
$s' = stamp\text{-}unary\ op\ (stamp\ g2\ x);$
$find\text{-}node\text{-}and\text{-}stamp\ g2\ (unary\text{-}node\ op\ x,\ s') = None;$
$nid = get\text{-}fresh\text{-}id\ g2;$
$g' = add\text{-}node\ nid\ (unary\text{-}node\ op\ x,\ s')\ g2]\!]$
$\Longrightarrow g \triangleleft (UnaryExpr\ op\ xe) \rightsquigarrow (g',\ nid)\ |$

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

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

$AllLeafNodes$:
$stamp\ g\ nid = s$
$\Longrightarrow g \triangleleft (LeafExpr\ nid\ s) \rightsquigarrow (g,\ nid)\ |$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

values $\{(\textit{nid},\ g)\ .\ (\textit{eg2-sq} \triangleleft \textit{sq-param0} \rightsquigarrow (g,\ \textit{nid}))\}$

5.2 Data-flow Tree Evaluation

inductive

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

ConstantExpr:
$[\![c \neq \mathit{UndefVal}]\!]$
$\Longrightarrow [m,p] \vdash (\mathit{ConstantExpr}\ c) \mapsto c$ |

ParameterExpr:
$[\![\mathit{valid\text{-}value}\ s\ (p!i)]\!]$
$\Longrightarrow [m,p] \vdash (\mathit{ParameterExpr}\ i\ s) \mapsto p!i$ |

ConditionalExpr:
$[\![[m,p] \vdash ce \mapsto cond;$
$branch = (\mathit{if}\ \mathit{val\text{-}to\text{-}bool}\ cond\ \mathit{then}\ te\ \mathit{else}\ fe);$
$[m,p] \vdash branch \mapsto v]\!]$
$\Longrightarrow [m,p] \vdash (\mathit{ConditionalExpr}\ ce\ te\ fe) \mapsto v$ |

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

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

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

$$\frac{c \neq \mathit{UndefVal}}{[m,p] \vdash \mathit{ConstantExpr}\ c \mapsto c}$$

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

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

$$\frac{[m,p] \vdash xe \mapsto v}{[m,p] \vdash \mathit{UnaryExpr}\ op\ xe \mapsto \mathit{unary\text{-}eval}\ op\ v}$$

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

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

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

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

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

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

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

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

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

5.3 Data-flow Tree Refinement

We define the induced semantic equivalence relation between expressions. Note that syntactic equality implies semantic equivalence, but not vice versa.

definition *equiv-exprs* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* (- ≐ - *55*) **where**
$(e1 \doteq e2) = (\forall\ m\ p\ v.\ (([m,p] \vdash e1 \mapsto v) \longleftrightarrow ([m,p] \vdash e2 \mapsto v)))$

We also prove that this is a total equivalence relation (*equivp equiv-exprs*) (HOL.Equiv_Relations), so that we can reuse standard results about equivalence relations.

lemma *equivp equiv-exprs*
apply (*auto simp add*: *equivp-def equiv-exprs-def*)
by (*metis equiv-exprs-def*)+

We define a refinement ordering over IRExpr and show that it is a preorder. Note that it is asymmetric because e2 may refer to fewer variables than e1.

instantiation *IRExpr* :: *preorder* **begin**

definition
le-expr-def [*simp*]: $(e1 \leq e2) \longleftrightarrow (\forall\ m\ p\ v.\ (([m,p] \vdash e1 \mapsto v) \longrightarrow ([m,p] \vdash e2 \mapsto v)))$

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

instance proof
fix $x\ y\ z$ :: *IRExpr*
show $x < y \longleftrightarrow x \leq y \wedge \neg\ (y \leq x)$ **by** (*simp add*: *equiv-exprs-def*; *auto*)
show $x \leq x$ **by** *simp*
show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$ **by** *simp*
qed
end

end

6 Data-flow Expression-Tree Theorems

theory *IRTreeEvalThms*
imports
Semantics.IRTreeEval
begin

6.1 Extraction and Evaluation of Expression Trees is Deterministic.

First, we prove some extra rules that relate each type of IRNode to the corresponding IRExpr type that 'rep' will produce. These are very helpful for proving that 'rep' is deterministic.

lemma *rep-constant*:
$g \vdash n \triangleright e \Longrightarrow$
kind $g\ n$ = *ConstantNode* $c \Longrightarrow$
e = *ConstantExpr* c
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-parameter*:
$g \vdash n \triangleright e \Longrightarrow$
kind $g\ n$ = *ParameterNode* $i \Longrightarrow$
$(\exists\, s.\ e$ = *ParameterExpr* $i\ s)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-conditional*:

$g \vdash n \triangleright e \Longrightarrow$
kind g n = *ConditionalNode* c t f $\Longrightarrow$
($\exists$ *ce te fe*. e = *ConditionalExpr ce te fe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-abs*:
$g \vdash n \triangleright e \Longrightarrow$
kind g n = *AbsNode* x $\Longrightarrow$
($\exists$ *xe*. e = *UnaryExpr UnaryAbs xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-not*:
$g \vdash n \triangleright e \Longrightarrow$
kind g n = *NotNode* x $\Longrightarrow$
($\exists$ *xe*. e = *UnaryExpr UnaryNot xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-negate*:
$g \vdash n \triangleright e \Longrightarrow$
kind g n = *NegateNode* x $\Longrightarrow$
($\exists$ *xe*. e = *UnaryExpr UnaryNeg xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-logicnegation*:
$g \vdash n \triangleright e \Longrightarrow$
kind g n = *LogicNegationNode* x $\Longrightarrow$
($\exists$ *xe*. e = *UnaryExpr UnaryLogicNegation xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-add*:
$g \vdash n \triangleright e \Longrightarrow$
kind g n = *AddNode* x y $\Longrightarrow$
($\exists$ *xe ye*. e = *BinaryExpr BinAdd xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-sub*:
$g \vdash n \triangleright e \Longrightarrow$
kind g n = *SubNode* x y $\Longrightarrow$
($\exists$ *xe ye*. e = *BinaryExpr BinSub xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-mul*:
$g \vdash n \triangleright e \Longrightarrow$
kind g n = *MulNode* x y $\Longrightarrow$
($\exists$ *xe ye*. e = *BinaryExpr BinMul xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-and*:
$g \vdash n \triangleright e \Longrightarrow$

kind g n = AndNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinAnd xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-or*:
g $\vdash$ *n* $\triangleright$ *e* $\Longrightarrow$
kind g n = OrNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinOr xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-xor*:
g $\vdash$ *n* $\triangleright$ *e* $\Longrightarrow$
kind g n = XorNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinXor xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-integer-equals*:
g $\vdash$ *n* $\triangleright$ *e* $\Longrightarrow$
kind g n = IntegerEqualsNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinIntegerEquals xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-integer-less-than*:
g $\vdash$ *n* $\triangleright$ *e* $\Longrightarrow$
kind g n = IntegerLessThanNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinIntegerLessThan xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-load-field*:
g $\vdash$ *n* $\triangleright$ *e* $\Longrightarrow$
is-preevaluated (*kind g n*) $\Longrightarrow$
($\exists$ *s. e = LeafExpr n s*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *repDet*:
shows (*g* $\vdash$ *n* $\triangleright$ *e1*) $\Longrightarrow$ (*g* $\vdash$ *n* $\triangleright$ *e2*) $\Longrightarrow$ *e1 = e2*
proof (*induction arbitrary*: *e2 rule*: *rep.induct*)
case (*ConstantNode n c*)
then show *?case* **using** *rep-constant* **by** *auto*
next
case (*ParameterNode n i s*)
then show *?case* **using** *rep-parameter* **by** *auto*
next
case (*ConditionalNode n c t f ce te fe*)
then show *?case*
by (*metis rep-conditional ConditionalNodeE IRNode.inject*(*6*))

next
case (*AbsNode n x xe*)
then show *?case*
by (*metis rep-abs AbsNodeE IRNode.inject*(*1*))
next
case (*NotNode n x xe*)
then show *?case*
by (*metis rep-not NotNodeE IRNode.inject*(*29*))
next
case (*NegateNode n x xe*)
then show *?case*
by (*metis IRNode.inject*(*26*) *NegateNodeE rep-negate*)
next
case (*LogicNegationNode n x xe*)
then show *?case*
by (*metis IRNode.inject*(*19*) *LogicNegationNodeE rep-logicnegation*)
next
case (*AddNode n x y xe ye*)
then show *?case*
by (*metis AddNodeE IRNode.inject*(*2*) *rep-add*)
next
case (*MulNode n x y xe ye*)
then show *?case*
by (*metis IRNode.inject*(*25*) *MulNodeE rep-mul*)
next
case (*SubNode n x y xe ye*)
then show *?case*
by (*metis IRNode.inject*(*39*) *SubNodeE rep-sub*)
next
case (*AndNode n x y xe ye*)
then show *?case*
by (*metis AndNodeE IRNode.inject*(*3*) *rep-and*)
next
case (*OrNode n x y xe ye*)
then show *?case*
by (*metis IRNode.inject*(*30*) *OrNodeE rep-or*)
next
case (*XorNode n x y xe ye*)
then show *?case*
by (*metis IRNode.inject*(*43*) *XorNodeE rep-xor*)
next
case (*IntegerEqualsNode n x y xe ye*)
then show *?case*
by (*metis IRNode.inject*(*12*) *IntegerEqualsNodeE rep-integer-equals*)
next
case (*IntegerLessThanNode n x y xe ye*)
then show *?case*
by (*metis IRNode.inject*(*13*) *IntegerLessThanNodeE rep-integer-less-than*)
next

case (*LeafNode n s*)
then show *?case* **using** *rep-load-field LeafNodeE* **by** *blast*
qed

lemma *evalDet*:
$[m,p] \vdash e \mapsto v1 \Longrightarrow$
$[m,p] \vdash e \mapsto v2 \Longrightarrow$
$v1 = v2$
apply (*induction arbitrary*: *v2 rule*: *evaltree.induct*)
by (*elim EvalTreeE*; *auto*)+

lemma *evalAllDet*:
$[m,p] \vdash e \mapsto_L v1 \Longrightarrow$
$[m,p] \vdash e \mapsto_L v2 \Longrightarrow$
$v1 = v2$
apply (*induction arbitrary*: *v2 rule*: *evaltrees.induct*)
apply (*elim EvalTreeE*; *auto*)
using *evalDet* **by** *force*

A valid value cannot be $UndefVal$.

lemma *valid-not-undef*:
assumes *a1*: *valid-value s val*
assumes *a2*: $s \neq$ *VoidStamp*
shows *val* $\neq$ *UndefVal*
apply (*rule valid-value.elims*(*1*)[*of s val True*])
using *a1 a2* **by** *auto*

lemma *valid-VoidStamp*[*elim*]:
shows *valid-value VoidStamp val* $\Longrightarrow$
val = *UndefVal*
using *valid-value.simps* **by** (*metis IRTreeEval.val-to-bool.cases*)

lemma *valid-ObjStamp*[*elim*]:
shows *valid-value* (*ObjectStamp klass exact nonNull alwaysNull*) *val* $\Longrightarrow$
($\exists$ *v*. *val* = *ObjRef v*)
using *valid-value.simps* **by** (*metis IRTreeEval.val-to-bool.cases*)

lemma *valid-int32*[*elim*]:
shows *valid-value* (*IntegerStamp 32 l h*) *val* $\Longrightarrow$
($\exists$ *v*. *val* = *IntVal32 v*)
apply (*rule IRTreeEval.val-to-bool.cases*[*of val*])
using *Value.distinct* **by** *simp*+

lemma *valid-int64*[*elim*]:
shows *valid-value* (*IntegerStamp 64 l h*) *val* $\Longrightarrow$
($\exists$ *v*. *val* = *IntVal64 v*)
apply (*rule IRTreeEval.val-to-bool.cases*[*of val*])

using *Value.distinct* **by** *simp+*

TODO: could we prove that expression evaluation never returns $UndefVal$? But this might require restricting unary and binary operators to be total...

lemma *leafint32*:
assumes *ev*: $[m,p] \vdash$ *LeafExpr i* (*IntegerStamp 32 lo hi*) $\mapsto$ *val*
shows $\exists$ *v*. *val* = (*IntVal32 v*)

proof –
have *valid-value* (*IntegerStamp 32 lo hi*) *val*
using *ev* **by** (*rule LeafExprE*; *simp*)
then show *?thesis* **by** *auto*
qed

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

proof –
have *valid-value* (*IntegerStamp 64 lo hi*) *val*
using *ev* **by** (*rule LeafExprE*; *simp*)
then show *?thesis* **by** *auto*
qed

lemma *default-stamp* [*simp*]: *default-stamp* = *IntegerStamp 32* (−*2147483648*) *2147483647*
using *default-stamp-def* **by** *auto*

lemma *valid32* [*simp*]:
assumes *valid-value* (*IntegerStamp 32 lo hi*) *val*
shows $\exists$ *v*. (*val* = (*IntVal32 v*) $\wedge$ *lo* $\leq$ *sint v* $\wedge$ *sint v* $\leq$ *hi*)
using *assms valid-int32* **by** *force*

lemma *valid64* [*simp*]:
assumes *valid-value* (*IntegerStamp 64 lo hi*) *val*
shows $\exists$ *v*. (*val* = (*IntVal64 v*) $\wedge$ *lo* $\leq$ *sint v* $\wedge$ *sint v* $\leq$ *hi*)
using *assms valid-int64* **by** *force*

lemma *int-stamp-implies-valid-value*:
$[m,p] \vdash$ *expr* $\mapsto$ *val* $\Longrightarrow$
valid-value (*stamp-expr expr*) *val*
proof (*induction rule*: *evaltree.induct*)
case (*ConstantExpr c*)
then show *?case* **sorry**
next
case (*ParameterExpr s i*)
then show *?case* **sorry**
next

case (*ConditionalExpr ce cond branch te fe v*)
then show *?case* **sorry**
next
case (*UnaryExpr xe v op*)
then show *?case* **sorry**
next
case (*BinaryExpr xe x ye y op*)
then show *?case* **sorry**
next
case (*LeafExpr val nid s*)
then show *?case* **sorry**
qed

6.2 Example Data-flow Optimisations

lemma *a0a-helper* [*simp*]:
assumes *a*: *valid-value* (*IntegerStamp 32 lo hi*) *v*
shows *intval-add v* (*IntVal32 0*) = *v*
proof −
obtain *v32* :: *int32* **where** *v* = (*IntVal32 v32*) **using** *a valid32* **by** *blast*
then show *?thesis* **by** *simp*
qed

lemma *a0a*: (*BinaryExpr BinAdd* (*LeafExpr 1 default-stamp*) (*ConstantExpr* (*IntVal32 0*)))
≤ (*LeafExpr 1 default-stamp*) (**is** *?L* ≤ *?R*)
by (*auto simp add*: *evaltree.LeafExpr*)

lemma *xyx-y-helper* [*simp*]:
assumes *valid-value* (*IntegerStamp 32 lox hix*) *x*
assumes *valid-value* (*IntegerStamp 32 loy hiy*) *y*
shows *intval-add x* (*intval-sub y x*) = *y*
proof −
obtain *x32* :: *int32* **where** *x*: *x* = (*IntVal32 x32*) **using** *assms valid32* **by** *blast*
obtain *y32* :: *int32* **where** *y*: *y* = (*IntVal32 y32*) **using** *assms valid32* **by** *blast*
show *?thesis* **using** *x y* **by** *simp*
qed

lemma *xyx-y*:
(*BinaryExpr BinAdd*
(*LeafExpr x* (*IntegerStamp 32 lox hix*))
(*BinaryExpr BinSub*
(*LeafExpr y* (*IntegerStamp 32 loy hiy*))
(*LeafExpr x* (*IntegerStamp 32 lox hix*))))
≤ (*LeafExpr y* (*IntegerStamp 32 loy hiy*))
by (*auto simp add*: *LeafExpr*)

6.3 Monotonicity of Expression Optimization

We prove that each subexpression position is monotonic. That is, optimizing a subexpression anywhere deep inside a top-level expression also optimizes that top-level expression.

Note that we might also be able to do this via reusing Isabelle's 'mono' operator (HOL.Orderings theory), proving instantiations like 'mono (UnaryExpr op)', but it is not obvious how to do this for both arguments of the binary expressions.

lemma *mono-unary*:
  assumes $e \leq e'$
  shows $(UnaryExpr\ op\ e) \leq (UnaryExpr\ op\ e')$
  using *UnaryExpr assms* **by** *auto*

lemma *mono-binary*:
  assumes $x \leq x'$
  assumes $y \leq y'$
  shows $(BinaryExpr\ op\ x\ y) \leq (BinaryExpr\ op\ x'\ y')$
  using *BinaryExpr assms* **by** *auto*

lemma *mono-conditional*:
  assumes $ce \leq ce'$
  assumes $te \leq te'$
  assumes $fe \leq fe'$
  shows $(ConditionalExpr\ ce\ te\ fe) \leq (ConditionalExpr\ ce'\ te'\ fe')$
proof (*simp only*: *le-expr-def*; (*rule allI*)+; *rule impI*)
  fix *m p v*
  assume *a*: $[m,p] \vdash ConditionalExpr\ ce\ te\ fe \mapsto v$
  then obtain *cond* **where** *ce*: $[m,p] \vdash ce \mapsto cond$ **by** *auto*
  then have *ce'*: $[m,p] \vdash ce' \mapsto cond$ **using** *assms* **by** *auto*
  define *branch* **where** *b*: $branch = (if\ val\text{-}to\text{-}bool\ cond\ then\ te\ else\ fe)$
  define *branch'* **where** *b'*: $branch' = (if\ val\text{-}to\text{-}bool\ cond\ then\ te'\ else\ fe')$
  then have $[m,p] \vdash branch \mapsto v$ **using** *a b ce evalDet* **by** *blast*
  then have $[m,p] \vdash branch' \mapsto v$ **using** *assms b b'* **by** *auto*
  then show $[m,p] \vdash ConditionalExpr\ ce'\ te'\ fe' \mapsto v$
    using *ConditionalExpr ce' b'* **by** *auto*
qed

end

7 Control-flow Semantics

theory *IRStepObj*
  imports
    IRTreeEval
begin

7.1 Heap

The heap model we introduce maps field references to object instances to runtime values. We use the H[f][p] heap representation. See \cite{heap−reps−2011}. We also introduce the DynamicHeap type which allocates new object references sequentially storing the next free object reference as 'Free'.

type-synonym $('a, 'b)$ *Heap* $= 'a \Rightarrow 'b \Rightarrow$ *Value*
type-synonym *Free* = *nat*
type-synonym $('a, 'b)$ *DynamicHeap* $= ('a, 'b)$ *Heap* $\times$ *Free*

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

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

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

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

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

7.2 Intraprocedural Semantics

Intraprocedural semantics are given as a small-step semantics.
Within the context of a graph, the configuration triple, (ID, MethodState, Heap), is related to the subsequent configuration.

inductive *step* :: *IRGraph* $\Rightarrow$ *Params* $\Rightarrow$ (*ID* $\times$ *MapState* $\times$ *FieldRefHeap*) $\Rightarrow$ (*ID* $\times$ *MapState* $\times$ *FieldRefHeap*) $\Rightarrow$ *bool*
(-, - $\vdash$ - $\rightarrow$ - *55*) **for** *g p* **where**

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

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

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

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

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

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

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

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

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

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

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

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

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

7.3 Interprocedural Semantics

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

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

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

  InvokeNodeStep:
  $[\![$*is-Invoke* (*kind g nid*);

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

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

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

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

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

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

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

$g \vdash \textit{exception} \triangleright \textit{exceptionE}$;
$[m,\ p] \vdash \textit{exceptionE} \mapsto e$;

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

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

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

7.4 Big-step Execution

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

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

inductive *exec* :: *Program*
⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap*
⇒ *Trace*
⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap*
⇒ *Trace*
⇒ *bool*
$(\text{-} \vdash \text{-} \mid \text{-} \longrightarrow\!* \text{-} \mid \text{-})$

for P
where
$\llbracket P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h');$
$\neg(\textit{has-return } m');$

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

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

|
$\llbracket P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h');$
$\textit{has-return } m';$

$l' = (l \mathbin{@} [(g,nid,m,p)])\rrbracket$
$\Longrightarrow \textit{exec } P\ (((g,nid,m,p)\#xs),h)\ l\ (((g',nid',m',p')\#ys),h')\ l'$
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$ *as Exec*) *exec* **.**

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

$\llbracket n = 0\rrbracket$
$\Longrightarrow \textit{exec-debug } p\ s\ n\ s$
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *exec-debug* **.**

7.4.1 Heap Testing

definition *p3*:: *Params* **where**
$p3 = [\textit{IntVal32 } 3]$

values $\{(\textit{prod.fst}(\textit{prod.snd}\ (\textit{prod.snd}\ (\textit{hd}\ (\textit{prod.fst res}))))\ 0$
$\mid \textit{res}.\ (\lambda x\ .\ \textit{Some eg2-sq}) \vdash ([(\textit{eg2-sq},0,\textit{new-map-state},p3), (\textit{eg2-sq},0,\textit{new-map-state},p3)],$
$\textit{new-heap}) \rightarrow *2*\ \textit{res}\}$

definition *field-sq* :: *string* **where**
$\textit{field-sq} = ''sq''$

definition *eg3-sq* :: *IRGraph* **where**

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

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

definition *eg4-sq* :: *IRGraph* **where**

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

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

end

8 Canonicalization Phase

theory *CanonicalizationTree*
  imports
    Semantics.IRTreeEval
begin

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

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

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

$is\text{-}neutral\ BinXor\ (IntVal32\ x) = (sint\ (x) = 0)$ |
$is\text{-}neutral\ BinXor\ (IntVal64\ x) = (sint\ (x) = 0)$ |

$is\text{-}neutral\ BinSub\ (IntVal32\ x) = (sint\ (x) = 0)$ |
$is\text{-}neutral\ BinSub\ (IntVal64\ x) = (sint\ (x) = 0)$ |

$is\text{-}neutral$ - - $= False$

fun $is\text{-}zero :: IRBinaryOp \Rightarrow Value \Rightarrow bool$ **where**
$is\text{-}zero\ BinMul\ (IntVal32\ x) = (sint\ (x) = 0)$ |
$is\text{-}zero\ BinMul\ (IntVal64\ x) = (sint\ (x) = 0)$ |
$is\text{-}zero$ - - $= False$

fun $int\text{-}to\text{-}value :: Value \Rightarrow int \Rightarrow Value$ **where**
$int\text{-}to\text{-}value\ (IntVal32\ \text{-})\ y = (IntVal32\ (word\text{-}of\text{-}int\ y))$ |
$int\text{-}to\text{-}value\ (IntVal64\ \text{-})\ y = (IntVal64\ (word\text{-}of\text{-}int\ y))$ |
$int\text{-}to\text{-}value$ - - $= UndefVal$

inductive $CanonicalizeBinaryOp :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ **where**
$binary\text{-}const\text{-}fold$:
$[\![x = (ConstantExpr\ val1);$
$y = (ConstantExpr\ val2);$
$val = bin\text{-}eval\ op\ val1\ val2]\!]$
$\Longrightarrow CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ (ConstantExpr\ val)$ |

$binary\text{-}fold\text{-}yneutral$:
$[\![y = (ConstantExpr\ c);$
$is\text{-}neutral\ op\ c]\!]$
$\Longrightarrow CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ x$ |

$binary\text{-}fold\text{-}yzero$:
$[\![y = ConstantExpr\ c;$
$is\text{-}zero\ op\ c;$
$zero = (int\text{-}to\text{-}value\ c\ (int\ 0))]\!]$
$\Longrightarrow CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ (ConstantExpr\ zero)$

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

inductive $CanonicalizeMul :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ **where**
$mul\text{-}negate$:
$[\![y = ConstantExpr\ c;$
$c = (IntVal32\ (-1)) \lor c = (IntVal64\ (-1))]\!]$
$\Longrightarrow CanonicalizeMul\ (BinaryExpr\ BinMul\ x\ y)\ (UnaryExpr\ UnaryNeg\ x)$

inductive *CanonicalizeAdd* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**
add-xsub:

$[\![x = (\mathit{BinaryExpr}\ \mathit{BinSub}\ a\ y)]\!]$
$\Longrightarrow \mathit{CanonicalizeAdd}\ (\mathit{BinaryExpr}\ \mathit{BinAdd}\ x\ y)\ a \mid$

add-ysub:

$[\![y = (\mathit{BinaryExpr}\ \mathit{BinSub}\ a\ x)]\!]$
$\Longrightarrow \mathit{CanonicalizeAdd}\ (\mathit{BinaryExpr}\ \mathit{BinAdd}\ x\ y)\ a \mid$

add-xnegate:

$[\![nx = (\mathit{UnaryExpr}\ \mathit{UnaryNeg}\ x)]\!]$
$\Longrightarrow \mathit{CanonicalizeAdd}\ (\mathit{BinaryExpr}\ \mathit{BinAdd}\ nx\ y)\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ y\ x) \mid$

add-ynegate:

$[\![ny = (\mathit{UnaryExpr}\ \mathit{UnaryNeg}\ y)]\!]$
$\Longrightarrow \mathit{CanonicalizeAdd}\ (\mathit{BinaryExpr}\ \mathit{BinAdd}\ x\ ny)\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ x\ y)$

inductive *CanonicalizeSub* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**
sub-same:

$[\![x = y;$
$b = \mathit{stp\text{-}bits}\ (\mathit{stamp\text{-}expr}\ x);$
$\mathit{zero} = (\mathit{if}\ b = 32\ \mathit{then}\ (\mathit{IntVal32}\ 0)\ \mathit{else}\ (\mathit{IntVal64}\ 0))]\!]$
$\Longrightarrow \mathit{CanonicalizeSub}\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ x\ y)\ (\mathit{ConstantExpr}\ \mathit{zero}) \mid$

sub-left-add1:

$[\![x = (\mathit{BinaryExpr}\ \mathit{BinAdd}\ a\ b)]\!]$
$\Longrightarrow \mathit{CanonicalizeSub}\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ x\ b)\ a \mid$

sub-left-add2:

$[\![x = (\mathit{BinaryExpr}\ \mathit{BinAdd}\ a\ b)]\!]$
$\Longrightarrow \mathit{CanonicalizeSub}\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ x\ a)\ b \mid$

sub-left-sub:

$[\![x = (\mathit{BinaryExpr}\ \mathit{BinSub}\ a\ b)]\!]$
$\Longrightarrow \mathit{CanonicalizeSub}\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ x\ a)\ (\mathit{UnaryExpr}\ \mathit{UnaryNeg}\ b) \mid$

sub-right-add1:

$[\![y = (BinaryExpr\ BinAdd\ a\ b)]\!]$
$\Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ a\ y)\ (UnaryExpr\ UnaryNeg\ b)\ |$

sub-right-add2:

$[\![y = (BinaryExpr\ BinAdd\ a\ b)]\!]$
$\Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ b\ y)\ (UnaryExpr\ UnaryNeg\ a)\ |$

sub-right-sub:

$[\![y = (BinaryExpr\ BinSub\ a\ b)]\!]$
$\Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ a\ y)\ b\ |$

sub-xzero:

$[\![z = (ConstantExpr\ (IntVal32\ 0)) \lor z = (ConstantExpr\ (IntVal64\ 0))]\!]$
$\Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ z\ x)\ (UnaryExpr\ UnaryNeg\ x)\ |$

sub-y-negate:

$[\![nb = (UnaryExpr\ UnaryNeg\ b)]\!]$
$\Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ a\ nb)\ (BinaryExpr\ BinAdd\ a\ b)$

inductive *CanonicalizeNegate* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**
negate-negate:

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

negate-sub:

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

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

$[\![nx = (UnaryExpr\ UnaryNot\ x)]\!]$

$\Longrightarrow$ *CanonicalizeNot* (*UnaryExpr UnaryNot nx*) *x*

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

$[\![ax = (UnaryExpr\ UnaryAbs\ x)]\!]$
$\Longrightarrow$ *CanonicalizeAbs* (*UnaryExpr UnaryAbs ax*) *ax* |

abs-neg:

$[\![nx = (UnaryExpr\ UnaryNeg\ x)]\!]$
$\Longrightarrow$ *CanonicalizeAbs* (*UnaryExpr UnaryAbs nx*) (*UnaryExpr UnaryAbs x*)

inductive *CanonicalizeAnd* :: *IRExpr* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* **where**
and-same:

$[\![x = y]\!]$
$\Longrightarrow$ *CanonicalizeAnd* (*BinaryExpr BinAnd x y*) *x* |

and-demorgans:

$[\![nx = (UnaryExpr\ UnaryNot\ x);$
$ny = (UnaryExpr\ UnaryNot\ y)]\!]$
$\Longrightarrow$ *CanonicalizeAnd* (*BinaryExpr BinAnd nx ny*) (*UnaryExpr UnaryNot* (*BinaryExpr BinOr x y*))

inductive *CanonicalizeOr* :: *IRExpr* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* **where**
or-same:

$[\![x = y]\!]$
$\Longrightarrow$ *CanonicalizeOr* (*BinaryExpr BinOr x y*) *x* |

or-demorgans:

$[\![nx = (UnaryExpr\ UnaryNot\ x);$
$ny = (UnaryExpr\ UnaryNot\ y)]\!]$
$\Longrightarrow$ *CanonicalizeOr* (*BinaryExpr BinOr nx ny*) (*UnaryExpr UnaryNot* (*BinaryExpr BinAnd x y*))

inductive *CanonicalizeIntegerEquals* :: *IRExpr* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* **where**
int-equals-same:

$[\![x = y]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals x y*) (*ConstantExpr* (*IntVal32 1*)) |

int-equals-distinct:

$[\![$*alwaysDistinct* (*stamp-expr x*) (*stamp-expr y*)$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals x y*) (*ConstantExpr* (*IntVal32 0*)) |

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

$[\![$*left* = (*BinaryExpr BinAdd x y*);
right = (*BinaryExpr BinAdd x z*)$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left right*) (*BinaryExpr BinIntegerEquals y z*) |

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

$[\![$*left* = (*BinaryExpr BinAdd x y*);
right = (*BinaryExpr BinAdd z x*)$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left right*) (*BinaryExpr BinIntegerEquals y z*) |

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

$[\![$*left* = (*BinaryExpr BinAdd y x*);
right = (*BinaryExpr BinAdd x z*)$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left right*) (*BinaryExpr BinIntegerEquals y z*) |

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

$[\![$*left* = (*BinaryExpr BinAdd y x*);
right = (*BinaryExpr BinAdd z x*)$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left right*) (*BinaryExpr BinIntegerEquals y z*) |

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

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

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

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

$right = (BinaryExpr\ BinSub\ z\ x)\rrbracket$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ right)\ (BinaryExpr\ BinIntegerEquals\ y\ z)\ |$

int-equals-left-contains-right1:

$\llbracket left = (BinaryExpr\ BinAdd\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))\rrbracket$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ x)\ (BinaryExpr\ BinIntegerEquals\ y\ zero)\ |$

int-equals-left-contains-right2:

$\llbracket left = (BinaryExpr\ BinAdd\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))\rrbracket$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ y)\ (BinaryExpr\ BinIntegerEquals\ x\ zero)\ |$

int-equals-right-contains-left1:

$\llbracket right = (BinaryExpr\ BinAdd\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))\rrbracket$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ x\ right)\ (BinaryExpr\ BinIntegerEquals\ y\ zero)\ |$

int-equals-right-contains-left2:

$\llbracket right = (BinaryExpr\ BinAdd\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))\rrbracket$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ y\ right)\ (BinaryExpr\ BinIntegerEquals\ x\ zero)\ |$

int-equals-left-contains-right3:

$\llbracket left = (BinaryExpr\ BinSub\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))\rrbracket$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ x)\ (BinaryExpr\ BinIntegerEquals\ y\ zero)\ |$

int-equals-right-contains-left3:

$\llbracket right = (BinaryExpr\ BinSub\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))\rrbracket$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ x\ right)\ (BinaryExpr\ BinIntegerEquals\ y\ zero)$

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

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

cond-eq:

$[\![c = (BinaryExpr\ BinIntegerEquals\ x\ y)]\!]$
$\Longrightarrow$ *CanonicalizeConditional* (*ConditionalExpr c x y*) *y* |

condition-bounds-x:

$[\![c = (BinaryExpr\ BinIntegerLessThan\ x\ y);$
$stamp\text{-}x = stamp\text{-}expr\ x;$
$stamp\text{-}y = stamp\text{-}expr\ y;$
$stpi\text{-}upper\ stamp\text{-}x \leq stpi\text{-}lower\ stamp\text{-}y]\!]$
$\Longrightarrow$ *CanonicalizeConditional* (*ConditionalExpr c x y*) *x* |

condition-bounds-y:

$[\![c = (BinaryExpr\ BinIntegerLessThan\ x\ y);$
$stamp\text{-}x = stamp\text{-}expr\ x;$
$stamp\text{-}y = stamp\text{-}expr\ y;$
$stpi\text{-}upper\ stamp\text{-}x \leq stpi\text{-}lower\ stamp\text{-}y]\!]$
$\Longrightarrow$ *CanonicalizeConditional* (*ConditionalExpr c y x*) *y* |

negate-condition:

$[\![nc = (UnaryExpr\ UnaryLogicNegation\ c)]\!]$
$\Longrightarrow$ *CanonicalizeConditional* (*ConditionalExpr nc x y*) (*ConditionalExpr c y x*)
|

const-true:

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

const-false:

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

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

$BinaryNode$:
$[\![CanonicalizeBinaryOp\ expr\ expr']\!]$
$\Longrightarrow CanonicalizationStep\ expr\ expr'$ |

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

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

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

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

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

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

$AndNode$:
$[\![CanonicalizeSub\ expr\ expr']\!]$
$\Longrightarrow CanonicalizationStep\ expr\ expr'$ |

$OrNode$:
$[\![CanonicalizeSub\ expr\ expr']\!]$
$\Longrightarrow CanonicalizationStep\ expr\ expr'$ |

$IntegerEqualsNode$:
$[\![CanonicalizeIntegerEquals\ expr\ expr']\!]$
$\Longrightarrow CanonicalizationStep\ expr\ expr'$ |

$ConditionalNode$:
$[\![CanonicalizeConditional\ expr\ expr']\!]$
$\Longrightarrow CanonicalizationStep\ expr\ expr'$

code-pred (*modes*: $i \Rightarrow o \Rightarrow bool$) *CanonicalizeBinaryOp* .
code-pred (*modes*: $i \Rightarrow o \Rightarrow bool$) *CanonicalizeUnaryOp* .
code-pred (*modes*: $i \Rightarrow o \Rightarrow bool$) *CanonicalizeNegate* .
code-pred (*modes*: $i \Rightarrow o \Rightarrow bool$) *CanonicalizeNot* .
code-pred (*modes*: $i \Rightarrow o \Rightarrow bool$) *CanonicalizeAdd* .
code-pred (*modes*: $i \Rightarrow o \Rightarrow bool$) *CanonicalizeSub* .
code-pred (*modes*: $i \Rightarrow o \Rightarrow bool$) *CanonicalizeMul* .
code-pred (*modes*: $i \Rightarrow o \Rightarrow bool$) *CanonicalizeAnd* .
code-pred (*modes*: $i \Rightarrow o \Rightarrow bool$) *CanonicalizeIntegerEquals* .
code-pred (*modes*: $i \Rightarrow o \Rightarrow bool$) *CanonicalizeConditional* .

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

end

9 Canonicalization Phase

theory *CanonicalizationTreeProofs*
imports
CanonicalizationTree
Semantics.IRTreeEvalThms
begin

lemma *valid32or64*:
assumes *valid-value* (*IntegerStamp b lo hi*) *x*
shows ($\exists$ *v1*. (*x* = *IntVal32 v1*)) $\vee$ ($\exists$ *v2*. (*x* = *IntVal64 v2*))
using *valid32 valid64 assms valid-value.elims*(*2*) **by** *blast*

lemma *valid32or64-both*:
assumes *valid-value* (*IntegerStamp b lox hix*) *x*
and *valid-value* (*IntegerStamp b loy hiy*) *y*
shows ($\exists$ *v1 v2*. *x* = *IntVal32 v1* $\wedge$ *y* = *IntVal32 v2*) $\vee$ ($\exists$ *v3 v4*. *x* = *IntVal64 v3* $\wedge$ *y* = *IntVal64 v4*)
using *assms valid32or64 valid32 valid-value.elims*(*2*) *valid-value.simps*(*1*) **by** *metis*

lemma *double-negate-refinement*:
assumes [*m*,*p*] $\vdash$ *expr* $\mapsto$ *val*
assumes *stamp-expr expr* = *IntegerStamp b lo hi*
shows (*UnaryExpr UnaryNeg* (*UnaryExpr UnaryNeg* (*expr*))) $\leq$ *expr*
proof −
have *valid-value* (*IntegerStamp b lo hi*) *val*
by (*metis assms int-stamp-implies-valid-value*)
moreover have *x* = *intval-negate* (*intval-negate x*) **if** *valid-value* (*IntegerStamp b lo hi*) *x* **for** *x*
using *valid32or64 that* **by** *fastforce*
ultimately show *?thesis* **using** *assms* **by** (*auto*; (*metis int-stamp-implies-valid-value*)+)

qed

lemma *negate-xsuby-helper*:
  assumes *valid-value* (*IntegerStamp b lox hix*) *x*
  and *valid-value* (*IntegerStamp b loy hiy*) *y*
  shows *intval-negate* (*intval-sub x y*) = *intval-sub y x*
proof −
  have (∃ *v1 v2*. *x* = *IntVal32 v1* ∧ *y* = *IntVal32 v2*) ∨ (∃ *v3 v4*. *x* = *IntVal64 v3* ∧ *y* = *IntVal64 v4*)
    using *valid32or64-both assms* **by** *auto*
  thus *?thesis* **by** *auto*
qed

lemma *neg-sub-refinement*:
  assumes [*m*,*p*] ⊢ *x* ↦ *xval*
  assumes [*m*,*p*] ⊢ *y* ↦ *yval*
  assumes *stamp-expr x* = *IntegerStamp b lox hix*
  assumes *stamp-expr y* = *IntegerStamp b loy hiy*
  shows (*UnaryExpr UnaryNeg* (*BinaryExpr BinSub x y*)) ≤ (*BinaryExpr BinSub y x*)
  unfolding *le-expr-def*
  by (*smt* (*verit*, *ccfv-threshold*) *assms negate-xsuby-helper int-stamp-implies-valid-value evaltree.simps BinaryExprE UnaryExprE bin-eval.simps*(*3*) *unary-eval.simps*(*6*))

lemma *CanonicalizeNegateProof*:
  assumes *CanonicalizeNegate before after*
  assumes [*m*, *p*] ⊢ *before* ↦ *res*
  assumes [*m*, *p*] ⊢ *after* ↦ *res'*
  shows *res* = *res'*
  using *assms*
proof (*induct rule*: *CanonicalizeNegate.induct*)
  case (*negate-negate nx x*)
  thus *?case* **using** *double-negate-refinement*
   by (*metis evalDet is-IntegerStamp-def le-expr-def negate-negate.hyps*(*1*) *negate-negate.prems*(*1*) *negate-negate.prems*(*2*))
next
  case (*negate-sub e x y stampx stampy*)
  thus *?case*
   using *assms neg-sub-refinement le-expr-def evalDet is-IntegerStamp-def negate-sub.hyps negate-sub.prems*
    by (*smt* (*verit*, *ccfv-SIG*) *BinaryExprE Stamp.collapse*(*1*))
qed

end