Veriopt

January 8, 2022

Abstract

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

Contents

1 Runtime Values and Arithmetic

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

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

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

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 |

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

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*)

fun *is-int-val* :: *Value* ⇒ *bool* **where**
  is-int-val (*IntVal32 v*) = *True* |
  is-int-val (*IntVal64 v*) = *True* |
  is-int-val - = *False*

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-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-below* :: *Value* ⇒ *Value* ⇒ *Value* **where**
  intval-below (*IntVal32 v1*) (*IntVal32 v2*) = *bool-to-val* (*v1* < *v2*) |
  intval-below (*IntVal64 v1*) (*IntVal64 v2*) = *bool-to-val* (*v1* < *v2*) |
  intval-below - - = *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-negate* :: *Value* ⇒ *Value* **where**
  intval-negate (*IntVal32 v*) = *IntVal32* (− *v*) |
  intval-negate (*IntVal64 v*) = *IntVal64* (− *v*) |
  intval-negate - = *UndefVal*

fun *intval-abs* :: *Value* ⇒ *Value* **where**
  intval-abs (*IntVal32 v*) = (*if* (*v*) <*s 0 then* (*IntVal32* (− *v*)) *else* (*IntVal32 v*)) |
  intval-abs (*IntVal64 v*) = (*if* (*v*) <*s 0 then* (*IntVal64* (− *v*)) *else* (*IntVal64 v*)) |
  intval-abs - = *UndefVal*

lemma [*code*]: *shiftl1 n* = *n* ∗ *2*
  by (*simp add*: *shiftl1-eq-mult-2*)

lemma [*code*]: *shiftr1 n* = *n div 2*
  by (*simp add*: *shiftr1-eq-div-2*)

lemma [*code*]: *sshiftr1 n* = *word-of-int* (*sint n div 2*)
  using *sshiftr1-eq* **by** *blast*

definition *shiftl* (**infix** << *75*) **where**
  shiftl w n = (*shiftl1* ^^ *n*) *w*

lemma *shiftl-power*[*simp*]: (*x*::(′*a*::*len*) *word*) ∗ (*2* ^ *j*) = *x* << *j*
  unfolding *shiftl-def* **apply** (*induction j*)
  apply *simp* **unfolding** *funpow-Suc-right*
  by (*metis* (*no-types*, *lifting*) *comp-def funpow-swap1 mult.left-commute power-Suc shiftl1-eq-mult-2*)

lemma (*x*::(′*a*::*len*) *word*) ∗ ((*2* ^ *j*) + *1*) = *x* << *j* + *x*
  by (*simp add*: *distrib-left*)

lemma (*x*::(′*a*::*len*) *word*) ∗ ((*2* ^ *j*) − *1*) = *x* << *j* − *x*
  by (*simp add*: *right-diff-distrib*)

lemma (*x*::(′*a*::*len*) *word*) ∗ ((*2*^*j*) + (*2*^*k*)) = *x* << *j* + *x* << *k*
  by (*simp add*: *distrib-left*)

lemma (*x*::(′*a*::*len*) *word*) ∗ ((*2*^*j*) − (*2*^*k*)) = *x* << *j* − *x* << *k*
  by (*simp add*: *right-diff-distrib*)

definition *signed-shiftr* (**infix** >> *75*) **where**
signed-shiftr w n = (*sshiftr1* ^^ *n*) *w*

definition *shiftr* (**infix** >>> *75*) **where**
shiftr w n = (*shiftr1* ^^ *n*) *w*

lemma *shiftr-power*[*simp*]: (*x*::(′*a*::*len*) *word*) *div* (*2* ^ *j*) = *x* >>> *j*
unfolding *shiftr-def* **apply** (*induction j*)
apply *simp* **unfolding** *funpow-Suc-right*
by (*metis* (*no-types*, *lifting*) *comp-apply div-exp-eq funpow-swap1 power-Suc2 power-add power-one-right shiftr1-eq-div-2*)

fun *intval-left-shift* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-left-shift (*IntVal32 v1*) (*IntVal32 v2*) = *IntVal32* (*v1* << *unat* (*v2 AND 0x1f*)) |
intval-left-shift (*IntVal64 v1*) (*IntVal64 v2*) = *IntVal64* (*v1* << *unat* (*v2 AND 0x3f*)) |
intval-left-shift - - = *UndefVal*

fun *intval-right-shift* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-right-shift (*IntVal32 v1*) (*IntVal32 v2*) = *IntVal32* (*v1* >> *unat* (*v2 AND 0x1f*)) |
intval-right-shift (*IntVal64 v1*) (*IntVal64 v2*) = *IntVal64* (*v1* >> *unat* (*v2 AND 0x3f*)) |
intval-right-shift - - = *UndefVal*

fun *intval-uright-shift* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-uright-shift (*IntVal32 v1*) (*IntVal32 v2*) = *IntVal32* (*v1* >>> *unat* (*v2 AND 0x1f*)) |
intval-uright-shift (*IntVal64 v1*) (*IntVal64 v2*) = *IntVal64* (*v1* >>> *unat* (*v2 AND 0x3f*)) |
intval-uright-shift - - = *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* (*2^31−1*)) (*IntVal64* (*2^31−1*)) = *IntVal64 4294967294*
by *eval*

end

2 Nodes

2.1 Types of Nodes

theory *IRNodes*
imports
Values
begin

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

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

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

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

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

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

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

PUT-STATE option) (*ir-next*: *SUCC*)
| *InvokeWithExceptionNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*) (*ir-exceptionEdge*: *SUCC*)
| *IsNullNode* (*ir-value*: *INPUT*)
| *KillingBeginNode* (*ir-next*: *SUCC*)
| *LeftShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *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*)
| *NarrowNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *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*)
| *RightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *ShortCircuitOrNode* (*ir-x*: *INPUT-COND*) (*ir-y*: *INPUT-COND*)
| *SignExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
| *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*)
| *UnsignedRightShiftNode* (*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*)

| *ZeroExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *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-IntegerBelowNode:
inputs-of (*IntegerBelowNode x y*) = [*x*, *y*] |
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-LeftShiftNode:
inputs-of (*LeftShiftNode x y*) = [*x*, *y*] |
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-NarrowNode:
inputs-of (*NarrowNode inputBits resultBits value*) = [*value*] |
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-RightShiftNode:
inputs-of (*RightShiftNode x y*) = [*x*, *y*] |
inputs-of-ShortCircuitOrNode:
inputs-of (*ShortCircuitOrNode x y*) = [*x*, *y*] |
inputs-of-SignExtendNode:
inputs-of (*SignExtendNode inputBits resultBits value*) = [*value*] |
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-UnsignedRightShiftNode:
inputs-of (*UnsignedRightShiftNode x y*) = [*x*, *y*] |
inputs-of-UnwindNode:
inputs-of (*UnwindNode exception*) = [*exception*] |
inputs-of-ValuePhiNode:
inputs-of (*ValuePhiNode nid0 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-ZeroExtendNode:
inputs-of (*ZeroExtendNode inputBits resultBits value*) = [*value*] |
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-IntegerBelowNode:
successors-of (*IntegerBelowNode x y*) = [] |
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-LeftShiftNode:
successors-of (*LeftShiftNode x y*) = [] |
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-NarrowNode:
successors-of (*NarrowNode inputBits resultBits value*) = [] |
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-RightShiftNode:
successors-of (*RightShiftNode x y*) = [] |
successors-of-ShortCircuitOrNode:
successors-of (*ShortCircuitOrNode x y*) = [] |
successors-of-SignExtendNode:
successors-of (*SignExtendNode inputBits resultBits value*) = [] |
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-UnsignedRightShiftNode:
successors-of (*UnsignedRightShiftNode 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-ZeroExtendNode:
successors-of (*ZeroExtendNode inputBits resultBits value*) = [] |
successors-of-NoNode: *successors-of* (*NoNode*) = [] |

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

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

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

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

end

2.2 Hierarchy of Nodes

theory *IRNodeHierarchy*
imports *IRNodes*
begin

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

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

These functions have been automatically generated from the compiler.

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

$is\text{-}EndNode\ EndNode = True\ |$
$is\text{-}EndNode\ \text{-} = False$

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

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

fun $is\text{-}ShiftNode :: IRNode \Rightarrow bool$ **where**
$is\text{-}ShiftNode\ n = ((is\text{-}LeftShiftNode\ n) \vee (is\text{-}RightShiftNode\ n) \vee (is\text{-}UnsignedRightShiftNode\ n))$

fun $is\text{-}BinaryNode :: IRNode \Rightarrow bool$ **where**
$is\text{-}BinaryNode\ n = ((is\text{-}BinaryArithmeticNode\ n) \vee (is\text{-}ShiftNode\ n))$

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

fun $is\text{-}IntegerConvertNode :: IRNode \Rightarrow bool$ **where**
$is\text{-}IntegerConvertNode\ n = ((is\text{-}NarrowNode\ n) \vee (is\text{-}SignExtendNode\ n) \vee (is\text{-}ZeroExtendNode\ n))$

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

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

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

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

fun $is\text{-}IntegerLowerThanNode :: IRNode \Rightarrow bool$ **where**
$is\text{-}IntegerLowerThanNode\ n = ((is\text{-}IntegerBelowNode\ n) \vee (is\text{-}IntegerLessThanNode\ n))$

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

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

$is\text{-}BinaryOpLogicNode\ n = ((is\text{-}CompareNode\ n))$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

fun *is-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-ShiftNode* 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-IntegerConvertNode* 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-IntegerConvertNode n*) ∨ (*is-NotNode n*) ∨ (*is-ShiftNode n*) ∨ (*is-UnaryArithmeticNode n*))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

fun *is-ConvertNode* :: *IRNode* ⇒ *bool* **where**
is-ConvertNode n = ((*is-IntegerConvertNode n*))

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

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

fun *is-same-ir-node-type* :: *IRNode* ⇒ *IRNode* ⇒ *bool* **where**
is-same-ir-node-type n1 n2 = (
((*is-AbsNode n1*) ∧ (*is-AbsNode n2*)) ∨
((*is-AddNode n1*) ∧ (*is-AddNode n2*)) ∨
((*is-AndNode n1*) ∧ (*is-AndNode n2*)) ∨
((*is-BeginNode n1*) ∧ (*is-BeginNode n2*)) ∨
((*is-BytecodeExceptionNode n1*) ∧ (*is-BytecodeExceptionNode n2*)) ∨
((*is-ConditionalNode n1*) ∧ (*is-ConditionalNode n2*)) ∨
((*is-ConstantNode n1*) ∧ (*is-ConstantNode n2*)) ∨
((*is-DynamicNewArrayNode n1*) ∧ (*is-DynamicNewArrayNode n2*)) ∨
((*is-EndNode n1*) ∧ (*is-EndNode n2*)) ∨
((*is-ExceptionObjectNode n1*) ∧ (*is-ExceptionObjectNode n2*)) ∨
((*is-FrameState n1*) ∧ (*is-FrameState n2*)) ∨
((*is-IfNode n1*) ∧ (*is-IfNode n2*)) ∨
((*is-IntegerBelowNode n1*) ∧ (*is-IntegerBelowNode n2*)) ∨
((*is-IntegerEqualsNode n1*) ∧ (*is-IntegerEqualsNode n2*)) ∨
((*is-IntegerLessThanNode n1*) ∧ (*is-IntegerLessThanNode n2*)) ∨
((*is-InvokeNode n1*) ∧ (*is-InvokeNode n2*)) ∨
((*is-InvokeWithExceptionNode n1*) ∧ (*is-InvokeWithExceptionNode n2*)) ∨
((*is-IsNullNode n1*) ∧ (*is-IsNullNode n2*)) ∨
((*is-KillingBeginNode n1*) ∧ (*is-KillingBeginNode n2*)) ∨
((*is-LoadFieldNode n1*) ∧ (*is-LoadFieldNode n2*)) ∨
((*is-LogicNegationNode n1*) ∧ (*is-LogicNegationNode n2*)) ∨
((*is-LoopBeginNode n1*) ∧ (*is-LoopBeginNode n2*)) ∨

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

end

3 Stamp Typing

theory *Stamp*
imports *Values*
begin

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

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

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

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

| *IllegalStamp*

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

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

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

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

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

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

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

is-stamp-empty x = *False*

— Calculate the meet stamp of two stamps

fun *meet* :: *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**
meet VoidStamp VoidStamp = *VoidStamp* |
meet (*IntegerStamp b1 l1 u1*) (*IntegerStamp b2 l2 u2*) = (
if b1 ≠ *b2 then IllegalStamp else*
(*IntegerStamp b1* (*min l1 l2*) (*max u1 u2*))
) |

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

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

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

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

fun *asConstant* :: *Stamp* ⇒ *Value* **where**
asConstant (*IntegerStamp b l h*) = (*if l* = *h then 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 VoidStamp) is a 32 bit integer stamp with an unrestricted range. We use *default-stamp* as it is a frequently used stamp.
definition *default-stamp* :: *Stamp* **where**
default-stamp = (*unrestricted-stamp* (*IntegerStamp 32 0 0*))

end

4 Graph Representation

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

begin

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

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

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

setup-lifting *type-definition-IRGraph*

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

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

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

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

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

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

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

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

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

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

definition *as-set* :: $IRGraph \Rightarrow (ID \times (IRNode \times Stamp))\ set$ **where**

as-set g = {(*n*, *kind g n*, *stamp g n*) | *n* . *n* ∈ *ids g*}

definition *domain-subtraction* :: *'a set* ⇒ (*'a* × *'b*) *set* ⇒ (*'a* × *'b*) *set*
(**infix** ⊴ *30*) **where**
domain-subtraction s r = {(*x*, *y*) . (*x*, *y*) ∈ *r* ∧ *x* ∉ *s*}

notation (*latex*)
domain-subtraction (- ⩤ -)

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 = (⋃ *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 = $\{j.\ j \in ids\ g \wedge (j,nid) \in input\text{-}edges\ g\}$

fun *successor-edges* :: *IRGraph* ⇒ *ID rel* **where**
  successor-edges g = $(\bigcup\ i \in ids\ g.\ \{(i,j)|j\ .\ j \in (succ\ \ g\ i)\})$

fun *predecessors* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
  predecessors g nid = $\{j.\ j \in ids\ g \wedge (j,nid) \in successor\text{-}edges\ g\}$

fun *nodes-of* :: *IRGraph* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID set* **where**
  nodes-of g sel = $\{nid \in ids\ g\ .\ sel\ (kind\ g\ nid)\}$

fun *edge* :: (*IRNode* ⇒ *'a*) ⇒ *ID* ⇒ *IRGraph* ⇒ *'a* **where**
  edge sel nid g = *sel* (*kind g nid*)

fun *filtered-inputs* :: *IRGraph* ⇒ *ID* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID list* **where**
  filtered-inputs g nid f = *filter* (*f* ∘ (*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 = $\{n \in (usages\ g\ nid).\ f\ (kind\ g\ n)\}$

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

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

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

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

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

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

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

lemma [*simp*]: *finite* (*dom* *g*) $\longrightarrow$ *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.Values
    Graph.Stamp
    HOL−Library.Word
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 *ID* = *nat*
type-synonym *MapState* = *ID* ⇒ *Value*
type-synonym *Params* = *Value list*

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

5.1 Data-flow Tree Representation

datatype *IRUnaryOp* =

UnaryAbs
| *UnaryNeg*
| *UnaryNot*
| *UnaryLogicNegation*
| *UnaryNarrow* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*)
| *UnarySignExtend* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*)
| *UnaryZeroExtend* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*)

datatype *IRBinaryOp* =
BinAdd
| *BinMul*
| *BinSub*
| *BinAnd*
| *BinOr*
| *BinXor*
| *BinLeftShift*
| *BinRightShift*
| *BinURightShift*
| *BinIntegerEquals*
| *BinIntegerLessThan*
| *BinIntegerBelow*

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*)

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

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

| *ConstantExpr* (*ir-const*: *Value*)
| *ConstantVar* (*ir-name*: *string*)
| *VariableExpr* (*ir-name*: *string*) (*ir-stamp*: *Stamp*)

fun *is-ground* :: *IRExpr* ⇒ *bool* **where**
is-ground (*UnaryExpr op e*) = *is-ground e* |
is-ground (*BinaryExpr op e1 e2*) = (*is-ground e1* ∧ *is-ground e2*) |
is-ground (*ConditionalExpr b e1 e2*) = (*is-ground b* ∧ *is-ground e1* ∧ *is-ground e2*) |
is-ground (*ParameterExpr i s*) = *True* |
is-ground (*LeafExpr n s*) = *True* |
is-ground (*ConstantExpr v*) = *True* |
is-ground (*ConstantVar name*) = *False* |
is-ground (*VariableExpr name s*) = *False*

typedef *GroundExpr* = { *e* :: *IRExpr* . *is-ground e* }

using *is-ground.simps*(*6*) **by** *blast*

5.2 Data-flow Tree Evaluation

fun *unary-eval* :: *IRUnaryOp* ⇒ *Value* ⇒ *Value* **where**
unary-eval UnaryAbs v = *intval-abs v* |
unary-eval UnaryNeg v = *intval-negate v* |
unary-eval UnaryNot v = *intval-not v* |
unary-eval UnaryLogicNegation (*IntVal32 v1*) = (*if v1* = *0 then* (*IntVal32 1*) *else* (*IntVal32 0*)) |
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 BinLeftShift v1 v2 = *intval-left-shift v1 v2* |
bin-eval BinRightShift v1 v2 = *intval-right-shift v1 v2* |
bin-eval BinURightShift v1 v2 = *intval-uright-shift v1 v2* |
bin-eval BinIntegerEquals v1 v2 = *intval-equals v1 v2* |
bin-eval BinIntegerLessThan v1 v2 = *intval-less-than v1 v2* |
bin-eval BinIntegerBelow v1 v2 = *intval-below v1 v2*

inductive *not-undef-or-fail* :: *Value* ⇒ *Value* ⇒ *bool* **where**
⟦*value* ≠ *UndefVal*⟧ ⟹ *not-undef-or-fail value value*

notation (*latex* **output**)
not-undef-or-fail (- = -)

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

ConstantExpr:
⟦*valid-value* (*constantAsStamp c*) *c*⟧
⟹ [*m*,*p*] ⊢ (*ConstantExpr c*) ↦ *c* |

ParameterExpr:
⟦*i* < *length p*; *valid-value s* (*p*!*i*)⟧
⟹ [*m*,*p*] ⊢ (*ParameterExpr i s*) ↦ *p*!*i* |

ConditionalExpr:
⟦[*m*,*p*] ⊢ *ce* ↦ *cond*;
branch = (*if val-to-bool cond then te else fe*);
[*m*,*p*] ⊢ *branch* ↦ *v*;

$v \neq UndefVal]\!]$
$\Longrightarrow [m,p] \vdash (ConditionalExpr\ ce\ te\ fe) \mapsto v \mid$

UnaryExpr:
$[\![[m,p] \vdash xe \mapsto v;$
$result = (unary\text{-}eval\ op\ v);$
$result \neq UndefVal]\!]$
$\Longrightarrow [m,p] \vdash (UnaryExpr\ op\ xe) \mapsto result \mid$

BinaryExpr:
$[\![[m,p] \vdash xe \mapsto x;$
$[m,p] \vdash ye \mapsto y;$
$result = (bin\text{-}eval\ op\ x\ y);$
$result \neq UndefVal]\!]$
$\Longrightarrow [m,p] \vdash (BinaryExpr\ op\ xe\ ye) \mapsto result \mid$

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

$$\frac{valid\text{-}value\ (constantAsStamp\ c)\ c}{[m,p] \vdash ConstantExpr\ c \mapsto c}$$

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

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

$$\frac{[m,p] \vdash xe \mapsto v \qquad result = unary\text{-}eval\ op\ v \qquad result \neq UndefVal}{[m,p] \vdash UnaryExpr\ op\ xe \mapsto result}$$

$$\frac{\begin{array}{c}[m,p] \vdash xe \mapsto x\\ [m,p] \vdash ye \mapsto y \qquad result = bin\text{-}eval\ op\ x\ y \qquad result \neq UndefVal\end{array}}{[m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto result}$$

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

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 :: $MapState \Rightarrow Params \Rightarrow IRExpr\ list \Rightarrow Value\ list \Rightarrow bool$ ([-,-] ⊢ - $\mapsto_L$

- *55*)
  for *m p* **where**

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

  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 **.**

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* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* (- $\doteq$ - *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*]: $(e2 \le e1) \longleftrightarrow (\forall\ m\ p\ v.\ (([m,p] \vdash e1 \mapsto v) \longrightarrow ([m,p] \vdash e2 \mapsto v)))$

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

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

end

end

6 Data-flow Expression-Tree Theorems

theory *IRTreeEvalThms*
  imports
    TreeToGraph
    HOL−Eisbach.Eisbach
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.

named-theorems *rep*

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *rep-integer-below* [*rep*]:

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

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

lemma *rep-narrow* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *NarrowNode ib rb x* $\Longrightarrow$
($\exists$ *x*. *e* = *UnaryExpr* (*UnaryNarrow ib rb*) *x*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-sign-extend* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *SignExtendNode ib rb x* $\Longrightarrow$
($\exists$ *x*. *e* = *UnaryExpr* (*UnarySignExtend ib rb*) *x*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-zero-extend* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *ZeroExtendNode ib rb x* $\Longrightarrow$
($\exists$ *x*. *e* = *UnaryExpr* (*UnaryZeroExtend ib rb*) *x*)
by (*induction rule*: *rep.induct*; *auto*)

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

method *solve-det* **uses** *node* =
(*match node* **in** *kind* - - = *node* - **for** *node* ⇒
⟨*match rep in r*: - $\Longrightarrow$ - = *node* - $\Longrightarrow$ - ⇒
⟨*match IRNode.inject in i*: (*node* - = *node* -) = - ⇒
⟨*match RepE in e*: - $\Longrightarrow$ ($\bigwedge$*x*. - = *node x* $\Longrightarrow$ -) $\Longrightarrow$ - ⇒
⟨*metis i e r*⟩⟩⟩ |
match node **in** *kind* - - = *node* - - **for** *node* ⇒

⟨*match rep in r*: - $\Longrightarrow$ - = *node* - - $\Longrightarrow$ - ⇒
⟨*match IRNode.inject in i*: (*node* - - = *node* - -) = - ⇒
⟨*match RepE in e*: - $\Longrightarrow$ ($\bigwedge$*x y*. - = *node x y* $\Longrightarrow$ -) $\Longrightarrow$ - ⇒
⟨*metis i e r*⟩⟩⟩⟩ |
match node **in** *kind* - - = *node* - - - **for** *node* ⇒
⟨*match rep in r*: - $\Longrightarrow$ - = *node* - - - $\Longrightarrow$ - ⇒
⟨*match IRNode.inject in i*: (*node* - - - = *node* - - -) = - ⇒
⟨*match RepE in e*: - $\Longrightarrow$ ($\bigwedge$*x y z*. - = *node x y z* $\Longrightarrow$ -) $\Longrightarrow$ - ⇒
⟨*metis i e r*⟩⟩⟩⟩ |
match node **in** *kind* - - = *node* - - - **for** *node* ⇒
⟨*match rep in r*: - $\Longrightarrow$ - = *node* - - - $\Longrightarrow$ - ⇒
⟨*match IRNode.inject in i*: (*node* - - - = *node* - - -) = - ⇒
⟨*match RepE in e*: - $\Longrightarrow$ ($\bigwedge$*x*. - = *node* - - *x* $\Longrightarrow$ -) $\Longrightarrow$ - ⇒
⟨*metis i e r*⟩⟩⟩⟩)

Now we can prove that 'rep' and 'eval', and their list versions, are deterministic.

lemma *repDet*:
shows ($g \vdash n \simeq e1$) $\Longrightarrow$ ($g \vdash n \simeq e2$) $\Longrightarrow$ $e1 = e2$
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 (*solve-det node*: *ConditionalNode*)
next
case (*AbsNode n x xe*)
then show *?case*
by (*solve-det node*: *AbsNode*)
next
case (*NotNode n x xe*)
then show *?case*
by (*solve-det node*: *NotNode*)
next
case (*NegateNode n x xe*)
then show *?case*
by (*solve-det node*: *NegateNode*)
next
case (*LogicNegationNode n x xe*)
then show *?case*
by (*solve-det node*: *LogicNegationNode*)
next
case (*AddNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *AddNode*)

```
next
  case (MulNode n x y xe ye)
  then show ?case
    by (solve-det node: MulNode)
next
  case (SubNode n x y xe ye)
  then show ?case
    by (solve-det node: SubNode)
next
  case (AndNode n x y xe ye)
  then show ?case
    by (solve-det node: AndNode)
next
  case (OrNode n x y xe ye)
  then show ?case
    by (solve-det node: OrNode)
next
  case (XorNode n x y xe ye)
  then show ?case
    by (solve-det node: XorNode)
next
  case (IntegerBelowNode n x y xe ye)
  then show ?case
    by (solve-det node: IntegerBelowNode)
next
  case (IntegerEqualsNode n x y xe ye)
  then show ?case
    by (solve-det node: IntegerEqualsNode)
next
  case (IntegerLessThanNode n x y xe ye)
  then show ?case
    by (solve-det node: IntegerLessThanNode)
next
  case (NarrowNode n x xe)
  then show ?case
    by (metis IRNode.inject(28) NarrowNodeE rep-narrow)
next
  case (SignExtendNode n x xe)
  then show ?case
    using SignExtendNodeE rep-sign-extend IRNode.inject(39)
    by (metis IRNode.inject(39) SignExtendNodeE rep-sign-extend)
next
  case (ZeroExtendNode n x xe)
  then show ?case
    by (metis IRNode.inject(50) ZeroExtendNodeE rep-zero-extend)
next
  case (LeafNode n s)
  then show ?case using rep-load-field LeafNodeE by blast
qed
```

lemma *repAllDet*:
  $g \vdash xs \simeq_L e1 \Longrightarrow$
   $g \vdash xs \simeq_L e2 \Longrightarrow$
   $e1 = e2$
proof (*induction arbitrary*: *e2 rule*: *replist.induct*)
  case *RepNil*
  then show *?case*
    using *replist.cases* **by** *auto*
next
  case (*RepCons x xe xs xse*)
  then show *?case*
    by (*metis list.distinct*(*1*) *list.sel*(*1*) *list.sel*(*3*) *repDet replist.cases*)
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*

lemma *encodeEvalDet*:
  $[g,m,p] \vdash e \mapsto v1 \Longrightarrow$
   $[g,m,p] \vdash e \mapsto v2 \Longrightarrow$
   $v1 = v2$
  by (*metis encodeeval-def evalDet repDet*)

lemma *graphDet*: $([g,m,p] \vdash nid \mapsto v1) \wedge ([g,m,p] \vdash nid \mapsto v2) \Longrightarrow v1 = v2$
  using *encodeEvalDet* **by** *blast*

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*

experiment begin
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
end

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*

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*)
  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 $e \geq e'$
  shows (*UnaryExpr op e*) ≥ (*UnaryExpr op e'*)

using *UnaryExpr assms* **by** *auto*

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

lemma *mono-conditional*:
  assumes $ce \geq ce'$
  assumes $te \geq te'$
  assumes $fe \geq fe'$
  shows $(ConditionalExpr\ ce\ te\ fe) \geq (ConditionalExpr\ ce'\ te'\ fe')$
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-to-bool cond then te else fe*)
  define *branch'* **where** *b'*: *branch'* = (*if val-to-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'*
    using *a* **by** *blast*
qed

end

7 Tree to Graph

theory *TreeToGraph*
  imports
    Semantics.IRTreeEval
    Graph.IRGraph
begin

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

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

fun *is-preevaluated* :: *IRNode* ⇒ *bool* **where**
  is-preevaluated (*InvokeNode n - - - - -*) = *True* |
  is-preevaluated (*InvokeWithExceptionNode n - - - - - -*) = *True* |

$is\text{-}preevaluated\ (NewInstanceNode\ n\ \text{-}\ \text{-}\ \text{-}) = True\ |$
$is\text{-}preevaluated\ (LoadFieldNode\ n\ \text{-}\ \text{-}\ \text{-}) = True\ |$
$is\text{-}preevaluated\ (SignedDivNode\ n\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-}) = True\ |$
$is\text{-}preevaluated\ (SignedRemNode\ n\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-}) = True\ |$
$is\text{-}preevaluated\ (ValuePhiNode\ n\ \text{-}\ \text{-}) = True\ |$
$is\text{-}preevaluated\ \text{-} = False$

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

$ConstantNode$:
$[\![kind\ g\ n = ConstantNode\ c]\!]$
$\Longrightarrow g \vdash n \simeq (ConstantExpr\ c)\ |$

$ParameterNode$:
$[\![kind\ g\ n = ParameterNode\ i;$
$stamp\ g\ n = s]\!]$
$\Longrightarrow g\ \vdash n \simeq (ParameterExpr\ i\ s)\ |$

$ConditionalNode$:
$[\![kind\ g\ n = ConditionalNode\ c\ t\ f;$
$g \vdash c \simeq ce;$
$g \vdash t \simeq te;$
$g \vdash f \simeq fe]\!]$
$\Longrightarrow g \vdash n \simeq (ConditionalExpr\ ce\ te\ fe)\ |$

$AbsNode$:
$[\![kind\ g\ n = AbsNode\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryAbs\ xe)\ |$

$NotNode$:
$[\![kind\ g\ n = NotNode\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryNot\ xe)\ |$

$NegateNode$:
$[\![kind\ g\ n = NegateNode\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryNeg\ xe)\ |$

$LogicNegationNode$:
$[\![kind\ g\ n = LogicNegationNode\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryLogicNegation\ xe)\ |$

$AddNode$:
$[\![kind\ g\ n = AddNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinAdd\ xe\ ye)\ |$

$MulNode$:
$[\![kind\ g\ n = MulNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinMul\ xe\ ye)\ |$

$SubNode$:
$[\![kind\ g\ n = SubNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinSub\ xe\ ye)\ |$

$AndNode$:
$[\![kind\ g\ n = AndNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinAnd\ xe\ ye)\ |$

$OrNode$:
$[\![kind\ g\ n = OrNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinOr\ xe\ ye)\ |$

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

$IntegerBelowNode$:
$[\![kind\ g\ n = IntegerBelowNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerBelow\ xe\ ye)\ |$

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

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

NarrowNode:
$[\![$*kind g n* = *NarrowNode inputBits resultBits x*;
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq$ (*UnaryExpr* (*UnaryNarrow inputBits resultBits*) *xe*) |

SignExtendNode:
$[\![$*kind g n* = *SignExtendNode inputBits resultBits x*;
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq$ (*UnaryExpr* (*UnarySignExtend inputBits resultBits*) *xe*) |

ZeroExtendNode:
$[\![$*kind g n* = *ZeroExtendNode inputBits resultBits x*;
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq$ (*UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *xe*) |

LeafNode:
$[\![$*is-preevaluated* (*kind g n*);
stamp g n = $s]\!]$
$\Longrightarrow g \vdash n \simeq$ (*LeafExpr n s*)

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

inductive
replist :: *IRGraph* ⇒ *ID list* ⇒ *IRExpr list* ⇒ *bool* (- ⊢ - $\simeq_L$ - *55*)
for *g* **where**

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

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

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

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

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

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

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

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

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

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

values $\{t.\ \textit{eg2-sq} \vdash 4 \simeq t\}$

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

stamp-unary op - = *IllegalStamp*

definition *fixed-32* :: *IRBinaryOp set* **where**
fixed-32 = {*BinIntegerEquals*, *BinIntegerLessThan*, *BinIntegerBelow*}

fun *stamp-binary* :: *IRBinaryOp* ⇒ *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**
stamp-binary op (*IntegerStamp b1 lo1 hi1*) (*IntegerStamp b2 lo2 hi2*) =
(*case op* ∈ *fixed-32 of True* ⇒ *unrestricted-stamp* (*IntegerStamp 32 lo1 hi1*) |
False ⇒
(*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 |
  unary-node (*UnaryNarrow ib rb*) *v = NarrowNode ib rb v* |
  unary-node (*UnarySignExtend ib rb*) *v = SignExtendNode ib rb v* |
  unary-node (*UnaryZeroExtend ib rb*) *v = ZeroExtendNode ib rb 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 BinLeftShift x y = LeftShiftNode x y |
  bin-node BinRightShift x y = RightShiftNode x y |
  bin-node BinURightShift x y = UnsignedRightShiftNode x y |
  bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
  bin-node BinIntegerLessThan x y = IntegerLessThanNode x y |
  bin-node BinIntegerBelow x y = IntegerBelowNode x y

fun *choose-32-64* :: *int* ⇒ *int64* ⇒ *Value* **where**
  choose-32-64 bits val =
    (*if bits = 32*
     then (*IntVal32* (*ucast val*))
     else (*IntVal64* (*val*)))

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

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* ⇒ *IRExpr* ⇒ (*IRGraph* × *ID*) ⇒ *bool* ($- \triangleleft - \rightsquigarrow -$ *55*)
and
unrepList :: *IRGraph* ⇒ *IRExpr list* ⇒ (*IRGraph* × *ID list*) ⇒ *bool* ($- \triangleleft_L - \rightsquigarrow -$ *55*)
where

ConstantNodeSame:
$[\![$*find-node-and-stamp g* (*ConstantNode c*, *constantAsStamp c*) = *Some n*$]\!]$
$\Longrightarrow g \triangleleft$ (*ConstantExpr c*) $\rightsquigarrow$ (*g*, *n*) |

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

ParameterNodeSame:
$[\![$*find-node-and-stamp g* (*ParameterNode i*, *s*) = *Some n*$]\!]$
$\Longrightarrow g \triangleleft$ (*ParameterExpr i s*) $\rightsquigarrow$ (*g*, *n*) |

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

ConditionalNodeSame:
$[\![g \triangleleft_L$ [*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 n*$]\!]$
$\Longrightarrow g \triangleleft$ (*ConditionalExpr ce te fe*) $\rightsquigarrow$ (*g2*, *n*) |

ConditionalNodeNew:
$[\![g \triangleleft_L$ [*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*;
n = *get-fresh-id g2*;
g' = *add-node n* (*ConditionalNode c t f*, s') *g2*$]\!]$
$\Longrightarrow g \triangleleft$ (*ConditionalExpr ce te fe*) $\rightsquigarrow$ (g', *n*) |

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

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

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

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

AllLeafNodes:
$\textit{stamp g n} = s$
$\ \ \Longrightarrow g \triangleleft (\textit{LeafExpr n s}) \leadsto (g,\ n)\ |$

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

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

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* **.**

$$\frac{\textit{find-node-and-stamp g } (\textit{ConstantNode c},\ \textit{constantAsStamp c}) = \textit{Some } n}{g \triangleleft \textit{ConstantExpr c} \leadsto (g,\ n)}$$

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

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

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

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

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

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

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

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

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

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

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

values {(*n*, *g*) . (*eg2-sq* ◁ *sq-param0* ⇝ (*g*, *n*))}

definition *encodeeval* :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *ID* ⇒ *Value* ⇒ *bool*
  ([-,-,-] ⊢ - ↦ - *50*)
  where

encodeeval g m p n v = ($\exists$ *e*. (*g* $\vdash$ *n* $\simeq$ *e*) $\wedge$ ([*m*,*p*] $\vdash$ *e* $\mapsto$ *v*))

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

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

definition *graph-refinement* :: *IRGraph* $\Rightarrow$ *IRGraph* $\Rightarrow$ *bool* **where**
graph-refinement g1 g2 =
($\forall$ *n* . *n* $\in$ *ids g1* $\longrightarrow$ ($\forall$ *e1*. (*g1* $\vdash$ *n* $\simeq$ *e1*) $\longrightarrow$ ($\exists$ *e2*. (*g2* $\vdash$ *n* $\simeq$ *e2*) $\wedge$ *e1* $\geq$ *e2*)))

lemma *graph-refinement*:
graph-refinement g1 g2 $\Longrightarrow$ ($\forall$ *n m p v*. *n* $\in$ *ids g1* $\longrightarrow$ ([*g1*, *m*, *p*] $\vdash$ *n* $\mapsto$ *v*) $\longrightarrow$ ([*g2*, *m*, *p*] $\vdash$ *n* $\mapsto$ *v*))
by (*meson encodeeval-def graph-refinement-def le-expr-def*)

definition *graph-represents-expression* :: *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool*
(- $\vdash$ - $\trianglelefteq$ - *50*)
where
graph-represents-expression g n e = ($\forall$ *m p v* . ([*m*,*p*] $\vdash$ *e* $\mapsto$ *v*) $\longrightarrow$ ([*g*,*m*,*p*] $\vdash$ *n* $\mapsto$ *v*))

end
theory *TreeToGraphThms*
imports
TreeToGraph
IRTreeEvalThms
HOL−Eisbach.Eisbach
begin

Lift refinement monotonicity to graph level. Hopefully these shouldn't really be required.

lemma *mono-abs*:
assumes *kind g1 n* = *AbsNode x* $\wedge$ *kind g2 n* = *AbsNode x*
assumes (*g1* $\vdash$ *x* $\simeq$ *xe1*) $\wedge$ (*g2* $\vdash$ *x* $\simeq$ *xe2*)
assumes *xe1* $\geq$ *xe2*
assumes (*g1* $\vdash$ *n* $\simeq$ *e1*) $\wedge$ (*g2* $\vdash$ *n* $\simeq$ *e2*)
shows *e1* $\geq$ *e2*
by (*metis AbsNode assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *mono-unary repDet*)

lemma *mono-not*:

assumes *kind g1 n = NotNode x* ∧ *kind g2 n = NotNode x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis NotNode assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *mono-unary repDet*)

lemma *mono-negate*:
assumes *kind g1 n = NegateNode x* ∧ *kind g2 n = NegateNode x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis NegateNode assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *mono-unary repDet*)

lemma *mono-logic-negation*:
assumes *kind g1 n = LogicNegationNode x* ∧ *kind g2 n = LogicNegationNode x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis LogicNegationNode assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *mono-unary repDet*)

lemma *mono-narrow*:
assumes *kind g1 n = NarrowNode ib rb x* ∧ *kind g2 n = NarrowNode ib rb x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
using *assms mono-unary repDet NarrowNode*
by *metis*

lemma *mono-sign-extend*:
assumes *kind g1 n = SignExtendNode ib rb x* ∧ *kind g2 n = SignExtendNode ib rb x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis SignExtendNode assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *mono-unary repDet*)

lemma *mono-zero-extend*:
assumes *kind g1 n = ZeroExtendNode ib rb x* ∧ *kind g2 n = ZeroExtendNode ib rb x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)

shows $e1 \geq e2$
using *assms mono-unary repDet ZeroExtendNode*
by *metis*

lemma *mono-conditional-graph*:
assumes $kind\ g1\ n = ConditionalNode\ c\ t\ f \wedge kind\ g2\ n = ConditionalNode\ c\ t\ f$
assumes $(g1 \vdash c \simeq ce1) \wedge (g2 \vdash c \simeq ce2)$
assumes $(g1 \vdash t \simeq te1) \wedge (g2 \vdash t \simeq te2)$
assumes $(g1 \vdash f \simeq fe1) \wedge (g2 \vdash f \simeq fe2)$
assumes $ce1 \geq ce2 \wedge te1 \geq te2 \wedge fe1 \geq fe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
by (*metis ConditionalNodeE IRNode.inject*(*6*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *assms*(*6*) *mono-conditional repDet rep-conditional*)

lemma *mono-add*:
assumes $kind\ g1\ n = AddNode\ x\ y \wedge kind\ g2\ n = AddNode\ x\ y$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$
assumes $xe1 \geq xe2 \wedge ye1 \geq ye2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
using *mono-binary assms*
by (*metis AddNodeE IRNode.inject*(*2*) *repDet rep-add*)

lemma *mono-mul*:
assumes $kind\ g1\ n = MulNode\ x\ y \wedge kind\ g2\ n = MulNode\ x\ y$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$
assumes $xe1 \geq xe2 \wedge ye1 \geq ye2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
using *mono-binary assms*
by (*metis IRNode.inject*(*27*) *MulNodeE repDet rep-mul*)

lemma *encodes-contains*:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n \neq NoNode$
apply (*induction rule*: *rep.induct*)
apply (*match IRNode.distinct* **in** *e*: $?n \neq NoNode \Rightarrow$
⟨*presburger add*: *e*⟩)+
by *fastforce*

lemma *no-encoding*:
assumes $n \notin ids\ g$
shows $\neg(g \vdash n \simeq e)$
using *assms* **apply** *simp* **apply** (*rule notI*) **by** (*induction e*; *simp add*: *encodes-contains*)

lemma *not-excluded-keep-type*:
  assumes $n \in ids\ g1$
  assumes $n \notin excluded$
  assumes $(excluded \trianglelefteq as\text{-}set\ g1) \subseteq as\text{-}set\ g2$
  shows $kind\ g1\ n = kind\ g2\ n \wedge stamp\ g1\ n = stamp\ g2\ n$
  using *assms* **unfolding** *as-set-def domain-subtraction-def* **by** *blast*

method *metis-node-eq-unary* **for** $node :: 'a \Rightarrow IRNode =$
  $($*match IRNode.inject* **in** i: $(node\ \text{-} = node\ \text{-}) = \text{-} \Rightarrow$
    ‹*metis i*›$)$
method *metis-node-eq-binary* **for** $node :: 'a \Rightarrow 'a \Rightarrow IRNode =$
  $($*match IRNode.inject* **in** i: $(node\ \text{-}\ \text{-} = node\ \text{-}\ \text{-}) = \text{-} \Rightarrow$
    ‹*metis i*›$)$
method *metis-node-eq-ternary* **for** $node :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow IRNode =$
  $($*match IRNode.inject* **in** i: $(node\ \text{-}\ \text{-}\ \text{-} = node\ \text{-}\ \text{-}\ \text{-}) = \text{-} \Rightarrow$
    ‹*metis i*›$)$

lemma *graph-semantics-preservation*:
  assumes a: $e1' \geq e2'$
  assumes b: $(\{n'\} \trianglelefteq as\text{-}set\ g1) \subseteq as\text{-}set\ g2$
  assumes c: $g1 \vdash n' \simeq e1'$
  assumes d: $g2 \vdash n' \simeq e2'$
  shows *graph-refinement g1 g2*
  unfolding *graph-refinement-def*
  apply (*rule allI*) **apply** (*rule impI*) **apply** (*rule allI*) **apply** (*rule impI*)
proof −
  fix *n e1*
  assume e: $n \in ids\ g1$
  assume f: $(g1 \vdash n \simeq e1)$

  show $\exists\ e2.\ (g2 \vdash n \simeq e2) \wedge e1 \geq e2$
  proof $(cases\ n = n')$
    case *True*
    have g: $e1 = e1'$ **using** *c f True repDet* **by** *simp*
    have h: $(g2 \vdash n \simeq e2') \wedge e1' \geq e2'$
      using *True a d* **by** *blast*
    then show *?thesis*
      using *g* **by** *blast*
  next
    case *False*
    have $n \notin \{n'\}$
      using *False* **by** *simp*
    then have i: $kind\ g1\ n = kind\ g2\ n \wedge stamp\ g1\ n = stamp\ g2\ n$
      using *not-excluded-keep-type*
      using *b e* **by** *presburger*
    show *?thesis* **using** *f i*
    proof (*induction e1*)
      case (*ConstantNode n c*)

then show *?case*
by (*metis eq-refl rep.ConstantNode*)
next
case (*ParameterNode n i s*)
then show *?case*
by (*metis eq-refl rep.ParameterNode*)
next
case (*ConditionalNode n c t f ce1 te1 fe1*)
have *k*: $g1 \vdash n \simeq$ *ConditionalExpr ce1 te1 fe1* **using** *f ConditionalNode*
by (*simp add*: *ConditionalNode.hyps*(*2*) *rep.ConditionalNode*)
obtain *cn tn fn* **where** *l*: *kind g1 n* = *ConditionalNode cn tn fn*
using *ConditionalNode.hyps*(*1*) **by** *blast*
then have *mc*: $g1 \vdash cn \simeq ce1$
using *ConditionalNode.hyps*(*1*) *ConditionalNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *mt*: $g1 \vdash tn \simeq te1$
using *ConditionalNode.hyps*(*1*) *ConditionalNode.hyps*(*3*) **by** *fastforce*
from *l* **have** *mf*: $g1 \vdash fn \simeq fe1$
using *ConditionalNode.hyps*(*1*) *ConditionalNode.hyps*(*4*) **by** *fastforce*
then show *?case*
proof −
have $g1 \vdash cn \simeq ce1$ **using** *mc* **by** *simp*
have $g1 \vdash tn \simeq te1$ **using** *mt* **by** *simp*
have $g1 \vdash fn \simeq fe1$ **using** *mf* **by** *simp*
have *cer*: $\exists$ *ce2*. ($g2 \vdash cn \simeq ce2$) $\wedge$ *ce1* $\geq$ *ce2*
using *ConditionalNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-ternary ConditionalNode*)
have *ter*: $\exists$ *te2*. ($g2 \vdash tn \simeq te2$) $\wedge$ *te1* $\geq$ *te2*
using *ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-ternary ConditionalNode*)
have $\exists$ *fe2*. ($g2 \vdash fn \simeq fe2$) $\wedge$ *fe1* $\geq$ *fe2*
using *ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-ternary ConditionalNode*)
then have $\exists$ *ce2 te2 fe2*. ($g2 \vdash n \simeq$ *ConditionalExpr ce2 te2 fe2*) $\wedge$ *ConditionalExpr ce1 te1 fe1* $\geq$ *ConditionalExpr ce2 te2 fe2*
using *ConditionalNode.prems l mono-conditional rep.ConditionalNode cer ter*
by (*smt* (*verit*) *IRTreeEvalThms.mono-conditional*)
then show *?thesis*
by *meson*
qed
next
case (*AbsNode n x xe1*)
have *k*: $g1 \vdash n \simeq$ *UnaryExpr UnaryAbs xe1* **using** *f AbsNode*
by (*simp add*: *AbsNode.hyps*(*2*) *rep.AbsNode*)
obtain *xn* **where** *l*: *kind g1 n* = *AbsNode xn*
using *AbsNode.hyps*(*1*) **by** *blast*

```
      then have m: g1 ⊢ xn ≃ xe1
        using AbsNode.hyps(1) AbsNode.hyps(2) by fastforce
      then show ?case
      proof (cases xn = n′)
        case True
        then have n: xe1 = e1′ using c m repDet by simp
        then have ev: g2 ⊢ n ≃ UnaryExpr UnaryAbs e2′ using AbsNode.hyps(1)
l m n
          using AbsNode.prems True d rep.AbsNode by simp
        then have r: UnaryExpr UnaryAbs e1′ ≥ UnaryExpr UnaryAbs e2′
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using AbsNode
        using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
          by (metis-node-eq-unary AbsNode)
          then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryAbs xe2) ∧ UnaryExpr
UnaryAbs xe1 ≥ UnaryExpr UnaryAbs xe2
          by (metis AbsNode.prems l mono-unary rep.AbsNode)
        then show ?thesis
          by meson
      qed
    next
      case (NotNode n x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr UnaryNot xe1 using f NotNode
        by (simp add: NotNode.hyps(2) rep.NotNode)
      obtain xn where l: kind g1 n = NotNode xn
        using NotNode.hyps(1) by blast
      then have m: g1 ⊢ xn ≃ xe1
        using NotNode.hyps(1) NotNode.hyps(2) by fastforce
      then show ?case
      proof (cases xn = n′)
        case True
        then have n: xe1 = e1′ using c m repDet by simp
        then have ev: g2 ⊢ n ≃ UnaryExpr UnaryNot e2′ using NotNode.hyps(1)
l m n
          using NotNode.prems True d rep.NotNode by simp
        then have r: UnaryExpr UnaryNot e1′ ≥ UnaryExpr UnaryNot e2′
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
```

using *NotNode*
using *False i b l not-excluded-keep-type singletonD no-encoding*
by (*metis-node-eq-unary NotNode*)
then have $\exists$ *xe2*. (*g2* $\vdash$ *n* $\simeq$ *UnaryExpr UnaryNot xe2*) $\wedge$ *UnaryExpr UnaryNot xe1* $\geq$ *UnaryExpr UnaryNot xe2*
by (*metis NotNode.prems l mono-unary rep.NotNode*)
then show *?thesis*
by *meson*
qed
next
case (*NegateNode n x xe1*)
have *k*: *g1* $\vdash$ *n* $\simeq$ *UnaryExpr UnaryNeg xe1* **using** *f NegateNode*
by (*simp add*: *NegateNode.hyps*(*2*) *rep.NegateNode*)
obtain *xn* **where** *l*: *kind g1 n* = *NegateNode xn*
using *NegateNode.hyps*(*1*) **by** *blast*
then have *m*: *g1* $\vdash$ *xn* $\simeq$ *xe1*
using *NegateNode.hyps*(*1*) *NegateNode.hyps*(*2*) **by** *fastforce*
then show *?case*
proof (*cases xn* = *n'*)
case *True*
then have *n*: *xe1* = *e1'* **using** *c m repDet* **by** *simp*
then have *ev*: *g2* $\vdash$ *n* $\simeq$ *UnaryExpr UnaryNeg e2'* **using** *NegateNode.hyps*(*1*) *l m n*
using *NegateNode.prems True d rep.NegateNode* **by** *simp*
then have *r*: *UnaryExpr UnaryNeg e1'* $\geq$ *UnaryExpr UnaryNeg e2'*
by (*meson a mono-unary*)
then show *?thesis* **using** *ev r*
by (*metis n*)
next
case *False*
have *g1* $\vdash$ *xn* $\simeq$ *xe1* **using** *m* **by** *simp*
have $\exists$ *xe2*. (*g2* $\vdash$ *xn* $\simeq$ *xe2*) $\wedge$ *xe1* $\geq$ *xe2*
using *NegateNode*
using *False i b l not-excluded-keep-type singletonD no-encoding*
by (*metis-node-eq-unary NegateNode*)
then have $\exists$ *xe2*. (*g2* $\vdash$ *n* $\simeq$ *UnaryExpr UnaryNeg xe2*) $\wedge$ *UnaryExpr UnaryNeg xe1* $\geq$ *UnaryExpr UnaryNeg xe2*
by (*metis NegateNode.prems l mono-unary rep.NegateNode*)
then show *?thesis*
by *meson*
qed
next
case (*LogicNegationNode n x xe1*)
have *k*: *g1* $\vdash$ *n* $\simeq$ *UnaryExpr UnaryLogicNegation xe1* **using** *f LogicNegationNode*
by (*simp add*: *LogicNegationNode.hyps*(*2*) *rep.LogicNegationNode*)
obtain *xn* **where** *l*: *kind g1 n* = *LogicNegationNode xn*
using *LogicNegationNode.hyps*(*1*) **by** *blast*
then have *m*: *g1* $\vdash$ *xn* $\simeq$ *xe1*

using *LogicNegationNode.hyps*(*1*) *LogicNegationNode.hyps*(*2*) **by** *fastforce*
then show *?case*
proof (*cases* $xn = n'$)
case *True*
then have *n*: $xe1 = e1'$ **using** *c m repDet* **by** *simp*
then have *ev*: $g2 \vdash n \simeq$ *UnaryExpr UnaryLogicNegation* $e2'$ **using** *LogicNegationNode.hyps*(*1*) *l m n*
using *LogicNegationNode.prems True d rep.LogicNegationNode* **by** *simp*
then have *r*: *UnaryExpr UnaryLogicNegation* $e1' \geq$ *UnaryExpr UnaryLogicNegation* $e2'$
by (*meson a mono-unary*)
then show *?thesis* **using** *ev r*
by (*metis n*)
next
case *False*
have $g1 \vdash xn \simeq xe1$ **using** *m* **by** *simp*
have $\exists$ *xe2*. ($g2 \vdash xn \simeq xe2$) $\wedge$ $xe1 \geq xe2$
using *LogicNegationNode*
using *False i b l not-excluded-keep-type singletonD no-encoding*
by (*metis-node-eq-unary LogicNegationNode*)
then have $\exists$ *xe2*. ($g2 \vdash n \simeq$ *UnaryExpr UnaryLogicNegation xe2*) $\wedge$ *UnaryExpr UnaryLogicNegation xe1* $\geq$ *UnaryExpr UnaryLogicNegation xe2*
by (*metis LogicNegationNode.prems l mono-unary rep.LogicNegationNode*)
then show *?thesis*
by *meson*
qed
next
case (*AddNode n x y xe1 ye1*)
have *k*: $g1 \vdash n \simeq$ *BinaryExpr BinAdd xe1 ye1* **using** *f AddNode*
by (*simp add*: *AddNode.hyps*(*2*) *rep.AddNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *AddNode xn yn*
using *AddNode.hyps*(*1*) **by** *blast*
then have *mx*: $g1 \vdash xn \simeq xe1$
using *AddNode.hyps*(*1*) *AddNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: $g1 \vdash yn \simeq ye1$
using *AddNode.hyps*(*1*) *AddNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof −
have $g1 \vdash xn \simeq xe1$ **using** *mx* **by** *simp*
have $g1 \vdash yn \simeq ye1$ **using** *my* **by** *simp*
have *xer*: $\exists$ *xe2*. ($g2 \vdash xn \simeq xe2$) $\wedge$ $xe1 \geq xe2$
using *AddNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary AddNode*)
have $\exists$ *ye2*. ($g2 \vdash yn \simeq ye2$) $\wedge$ $ye1 \geq ye2$
using *AddNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary AddNode*)
then have $\exists$ *xe2 ye2*. ($g2 \vdash n \simeq$ *BinaryExpr BinAdd xe2 ye2*) $\wedge$ *BinaryExpr*

BinAdd xe1 ye1 $\geq$ *BinaryExpr BinAdd xe2 ye2*
by (*metis AddNode.prems l mono-binary rep.AddNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*MulNode n x y xe1 ye1*)
have *k*: *g1* $\vdash$ *n* $\simeq$ *BinaryExpr BinMul xe1 ye1* **using** *f MulNode*
by (*simp add*: *MulNode.hyps*(*2*) *rep.MulNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *MulNode xn yn*
using *MulNode.hyps*(*1*) **by** *blast*
then have *mx*: *g1* $\vdash$ *xn* $\simeq$ *xe1*
using *MulNode.hyps*(*1*) *MulNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* $\vdash$ *yn* $\simeq$ *ye1*
using *MulNode.hyps*(*1*) *MulNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof –
have *g1* $\vdash$ *xn* $\simeq$ *xe1* **using** *mx* **by** *simp*
have *g1* $\vdash$ *yn* $\simeq$ *ye1* **using** *my* **by** *simp*
have *xer*: $\exists$ *xe2*. (*g2* $\vdash$ *xn* $\simeq$ *xe2*) $\wedge$ *xe1* $\geq$ *xe2*
using *MulNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary MulNode*)
have $\exists$ *ye2*. (*g2* $\vdash$ *yn* $\simeq$ *ye2*) $\wedge$ *ye1* $\geq$ *ye2*
using *MulNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary MulNode*)
then have $\exists$ *xe2 ye2*. (*g2* $\vdash$ *n* $\simeq$ *BinaryExpr BinMul xe2 ye2*) $\wedge$ *BinaryExpr BinMul xe1 ye1* $\geq$ *BinaryExpr BinMul xe2 ye2*
by (*metis MulNode.prems l mono-binary rep.MulNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*SubNode n x y xe1 ye1*)
have *k*: *g1* $\vdash$ *n* $\simeq$ *BinaryExpr BinSub xe1 ye1* **using** *f SubNode*
by (*simp add*: *SubNode.hyps*(*2*) *rep.SubNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *SubNode xn yn*
using *SubNode.hyps*(*1*) **by** *blast*
then have *mx*: *g1* $\vdash$ *xn* $\simeq$ *xe1*
using *SubNode.hyps*(*1*) *SubNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* $\vdash$ *yn* $\simeq$ *ye1*
using *SubNode.hyps*(*1*) *SubNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof –
have *g1* $\vdash$ *xn* $\simeq$ *xe1* **using** *mx* **by** *simp*
have *g1* $\vdash$ *yn* $\simeq$ *ye1* **using** *my* **by** *simp*
have *xer*: $\exists$ *xe2*. (*g2* $\vdash$ *xn* $\simeq$ *xe2*) $\wedge$ *xe1* $\geq$ *xe2*
using *SubNode*

using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary SubNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *SubNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary SubNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinSub xe2 ye2*) ∧ *BinaryExpr BinSub xe1 ye1* ≥ *BinaryExpr BinSub xe2 ye2*
by (*metis SubNode.prems l mono-binary rep.SubNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*AndNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinAnd xe1 ye1* **using** *f AndNode*
by (*simp add*: *AndNode.hyps*(*2*) *rep.AndNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *AndNode xn yn*
using *AndNode.hyps*(*1*) **by** *blast*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *AndNode.hyps*(*1*) *AndNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *AndNode.hyps*(*1*) *AndNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1* **using** *mx* **by** *simp*
have *g1* ⊢ *yn* ≃ *ye1* **using** *my* **by** *simp*
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *AndNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary AndNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *AndNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary AndNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinAnd xe2 ye2*) ∧ *BinaryExpr BinAnd xe1 ye1* ≥ *BinaryExpr BinAnd xe2 ye2*
by (*metis AndNode.prems l mono-binary rep.AndNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*OrNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinOr xe1 ye1* **using** *f OrNode*
by (*simp add*: *OrNode.hyps*(*2*) *rep.OrNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *OrNode xn yn*
using *OrNode.hyps*(*1*) **by** *blast*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *OrNode.hyps*(*1*) *OrNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *OrNode.hyps*(*1*) *OrNode.hyps*(*3*) **by** *fastforce*

then show *?case*
proof −
have $g1 \vdash xn \simeq xe1$ **using** *mx* **by** *simp*
have $g1 \vdash yn \simeq ye1$ **using** *my* **by** *simp*
have *xer*: $\exists\ xe2.\ (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$
using *OrNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary OrNode*)
have $\exists\ ye2.\ (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$
using *OrNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary OrNode*)
then have $\exists\ xe2\ ye2.\ (g2 \vdash n \simeq BinaryExpr\ BinOr\ xe2\ ye2) \wedge BinaryExpr\ BinOr\ xe1\ ye1 \geq BinaryExpr\ BinOr\ xe2\ ye2$
by (*metis OrNode.prems l mono-binary rep.OrNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*XorNode n x y xe1 ye1*)
have *k*: $g1 \vdash n \simeq BinaryExpr\ BinXor\ xe1\ ye1$ **using** *f XorNode*
by (*simp add*: *XorNode.hyps*(*2*) *rep.XorNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *XorNode xn yn*
using *XorNode.hyps*(*1*) **by** *blast*
then have *mx*: $g1 \vdash xn \simeq xe1$
using *XorNode.hyps*(*1*) *XorNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: $g1 \vdash yn \simeq ye1$
using *XorNode.hyps*(*1*) *XorNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof −
have $g1 \vdash xn \simeq xe1$ **using** *mx* **by** *simp*
have $g1 \vdash yn \simeq ye1$ **using** *my* **by** *simp*
have *xer*: $\exists\ xe2.\ (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$
using *XorNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary XorNode*)
have $\exists\ ye2.\ (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$
using *XorNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary XorNode*)
then have $\exists\ xe2\ ye2.\ (g2 \vdash n \simeq BinaryExpr\ BinXor\ xe2\ ye2) \wedge BinaryExpr\ BinXor\ xe1\ ye1 \geq BinaryExpr\ BinXor\ xe2\ ye2$
by (*metis XorNode.prems l mono-binary rep.XorNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*IntegerBelowNode n x y xe1 ye1*)
have *k*: $g1 \vdash n \simeq BinaryExpr\ BinIntegerBelow\ xe1\ ye1$ **using** *f IntegerBelowNode*

by (*simp add*: *IntegerBelowNode.hyps*(*2*) *rep.IntegerBelowNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *IntegerBelowNode xn yn*
using *IntegerBelowNode.hyps*(*1*) **by** *blast*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *IntegerBelowNode.hyps*(*1*) *IntegerBelowNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *IntegerBelowNode.hyps*(*1*) *IntegerBelowNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1* **using** *mx* **by** *simp*
have *g1* ⊢ *yn* ≃ *ye1* **using** *my* **by** *simp*
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *IntegerBelowNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary IntegerBelowNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *IntegerBelowNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary IntegerBelowNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinIntegerBelow xe2 ye2*) ∧ *BinaryExpr BinIntegerBelow xe1 ye1* ≥ *BinaryExpr BinIntegerBelow xe2 ye2*
by (*metis IntegerBelowNode.prems l mono-binary rep.IntegerBelowNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*IntegerEqualsNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinIntegerEquals xe1 ye1* **using** *f IntegerEqualsNode*
by (*simp add*: *IntegerEqualsNode.hyps*(*2*) *rep.IntegerEqualsNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *IntegerEqualsNode xn yn*
using *IntegerEqualsNode.hyps*(*1*) **by** *blast*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *IntegerEqualsNode.hyps*(*1*) *IntegerEqualsNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *IntegerEqualsNode.hyps*(*1*) *IntegerEqualsNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1* **using** *mx* **by** *simp*
have *g1* ⊢ *yn* ≃ *ye1* **using** *my* **by** *simp*
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *IntegerEqualsNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary IntegerEqualsNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *IntegerEqualsNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary IntegerEqualsNode*)

then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinIntegerEquals xe2 ye2*) ∧ *BinaryExpr BinIntegerEquals xe1 ye1* ≥ *BinaryExpr BinIntegerEquals xe2 ye2*
by (*metis IntegerEqualsNode.prems l mono-binary rep.IntegerEqualsNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*IntegerLessThanNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinIntegerLessThan xe1 ye1* **using** *f IntegerLessThanNode*
by (*simp add*: *IntegerLessThanNode.hyps*(*2*) *rep.IntegerLessThanNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *IntegerLessThanNode xn yn*
using *IntegerLessThanNode.hyps*(*1*) **by** *blast*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *IntegerLessThanNode.hyps*(*1*) *IntegerLessThanNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *IntegerLessThanNode.hyps*(*1*) *IntegerLessThanNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1* **using** *mx* **by** *simp*
have *g1* ⊢ *yn* ≃ *ye1* **using** *my* **by** *simp*
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *IntegerLessThanNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary IntegerLessThanNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *IntegerLessThanNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary IntegerLessThanNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinIntegerLessThan xe2 ye2*) ∧ *BinaryExpr BinIntegerLessThan xe1 ye1* ≥ *BinaryExpr BinIntegerLessThan xe2 ye2*
by (*metis IntegerLessThanNode.prems l mono-binary rep.IntegerLessThanNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*NarrowNode n inputBits resultBits x xe1*)
have *k*: *g1* ⊢ *n* ≃ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) *xe1* **using** *f NarrowNode*
by (*simp add*: *NarrowNode.hyps*(*2*) *rep.NarrowNode*)
obtain *xn* **where** *l*: *kind g1 n* = *NarrowNode inputBits resultBits xn*
using *NarrowNode.hyps*(*1*) **by** *blast*
then have *m*: *g1* ⊢ *xn* ≃ *xe1*
using *NarrowNode.hyps*(*1*) *NarrowNode.hyps*(*2*)

by *auto*
then show *?case*
proof (*cases* $xn = n'$)
case *True*
then have n: $xe1 = e1'$ **using** *c m repDet* **by** *simp*
then have *ev*: $g2 \vdash n \simeq$ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) $e2'$
using *NarrowNode.hyps*(*1*) *l m n*
using *NarrowNode.prems True d rep.NarrowNode* **by** *simp*
then have *r*: *UnaryExpr* (*UnaryNarrow inputBits resultBits*) $e1' \geq$ *UnaryExpr*
(*UnaryNarrow inputBits resultBits*) $e2'$
by (*meson a mono-unary*)
then show *?thesis* **using** *ev r*
by (*metis n*)
next
case *False*
have $g1 \vdash xn \simeq xe1$ **using** *m* **by** *simp*
have $\exists$ *xe2*. ($g2 \vdash xn \simeq xe2$) $\wedge$ $xe1 \geq xe2$
using *NarrowNode*
using *False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff*
by (*metis-node-eq-ternary NarrowNode*)
then have $\exists$ *xe2*. ($g2 \vdash n \simeq$ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) *xe2*) $\wedge$ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) $xe1 \geq$ *UnaryExpr*
(*UnaryNarrow inputBits resultBits*) *xe2*
by (*metis NarrowNode.prems l mono-unary rep.NarrowNode*)
then show *?thesis*
by *meson*
qed
next
case (*SignExtendNode n inputBits resultBits x xe1*)
have *k*: $g1 \vdash n \simeq$ *UnaryExpr* (*UnarySignExtend inputBits resultBits*) *xe1*
using *f SignExtendNode*
by (*simp add*: *SignExtendNode.hyps*(*2*) *rep.SignExtendNode*)
obtain *xn* **where** *l*: *kind g1 n* = *SignExtendNode inputBits resultBits xn*
using *SignExtendNode.hyps*(*1*) **by** *blast*
then have *m*: $g1 \vdash xn \simeq xe1$
using *SignExtendNode.hyps*(*1*) *SignExtendNode.hyps*(*2*)
by *auto*
then show *?case*
proof (*cases* $xn = n'$)
case *True*
then have n: $xe1 = e1'$ **using** *c m repDet* **by** *simp*
then have *ev*: $g2 \vdash n \simeq$ *UnaryExpr* (*UnarySignExtend inputBits resultBits*)
$e2'$ **using** *SignExtendNode.hyps*(*1*) *l m n*
using *SignExtendNode.prems True d rep.SignExtendNode* **by** *simp*
then have *r*: *UnaryExpr* (*UnarySignExtend inputBits resultBits*) $e1' \geq$
UnaryExpr (*UnarySignExtend inputBits resultBits*) $e2'$
by (*meson a mono-unary*)
then show *?thesis* **using** *ev r*
by (*metis n*)

```
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using SignExtendNode
        using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
          by (metis-node-eq-ternary SignExtendNode)
       then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr (UnarySignExtend inputBits result-
Bits) xe2) ∧ UnaryExpr (UnarySignExtend inputBits resultBits) xe1 ≥ UnaryExpr
(UnarySignExtend inputBits resultBits) xe2
          by (metis SignExtendNode.prems l mono-unary rep.SignExtendNode)
        then show ?thesis
          by meson
      qed
    next
      case (ZeroExtendNode n inputBits resultBits x xe1)
       have k: g1 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits resultBits) xe1
using f ZeroExtendNode
        by (simp add: ZeroExtendNode.hyps(2) rep.ZeroExtendNode)
      obtain xn where l: kind g1 n = ZeroExtendNode inputBits resultBits xn
        using ZeroExtendNode.hyps(1) by blast
      then have m: g1 ⊢ xn ≃ xe1
        using ZeroExtendNode.hyps(1) ZeroExtendNode.hyps(2)
        by auto
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1' using c m repDet by simp
       then have ev: g2 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits resultBits)
e2' using ZeroExtendNode.hyps(1) l m n
          using ZeroExtendNode.prems True d rep.ZeroExtendNode by simp
          then have r: UnaryExpr (UnaryZeroExtend inputBits resultBits) e1' ≥
UnaryExpr (UnaryZeroExtend inputBits resultBits) e2'
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using ZeroExtendNode
        using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
          by (metis-node-eq-ternary ZeroExtendNode)
       then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits result-
Bits) xe2) ∧ UnaryExpr (UnaryZeroExtend inputBits resultBits) xe1 ≥ UnaryExpr
(UnaryZeroExtend inputBits resultBits) xe2
          by (metis ZeroExtendNode.prems l mono-unary rep.ZeroExtendNode)
        then show ?thesis
          by meson
```

qed
next
case (*LeafNode n s*)
then show *?case*
by (*metis eq-refl rep.LeafNode*)
qed
qed
qed

definition *maximal-sharing*:
maximal-sharing g = ($\forall$ *n1 n2* . *n1* $\in$ *ids g* $\wedge$ *n2* $\in$ *ids g* $\longrightarrow$
($\forall$ *e*. (*g* $\vdash$ *n1* $\simeq$ *e*) $\wedge$ (*g* $\vdash$ *n2* $\simeq$ *e*) $\longrightarrow$ *n1* = *n2*))

lemma *tree-to-graph-rewriting*:
e1 $\geq$ *e2*
$\wedge$ (*g1* $\vdash$ *n* $\simeq$ *e1*) $\wedge$ *maximal-sharing g1*
$\wedge$ ({*n*} $\trianglelefteq$ *as-set g1*) $\subseteq$ *as-set g2*
$\wedge$ (*g2* $\vdash$ *n* $\simeq$ *e2*) $\wedge$ *maximal-sharing g2*
$\Longrightarrow$ *graph-refinement g1 g2*
using *graph-semantics-preservation*
by *auto*

declare [[*simp-trace*]]
lemma *equal-refines*:
fixes *e1 e2* :: *IRExpr*
assumes *e1* = *e2*
shows *e1* $\geq$ *e2*
using *assms*
by *simp*
declare [[*simp-trace=false*]]

lemma *subset-implies-evals*:
assumes *as-set g1* $\subseteq$ *as-set g2*
shows (*g1* $\vdash$ *n* $\simeq$ *e*) $\Longrightarrow$ (*g2* $\vdash$ *n* $\simeq$ *e*)
proof (*induction e arbitrary*: *n*)
case (*UnaryExpr op e*)
then have *n* $\in$ *ids g1*
using *no-encoding* **by** *force*
then have *kind g1 n* = *kind g2 n*
using *assms* **unfolding** *as-set-def*
by *blast*
then show *?case* **using** *UnaryExpr UnaryRepE*
by (*smt* (*verit*, *ccfv-threshold*) *AbsNode LogicNegationNode NarrowNode NegateNode NotNode SignExtendNode ZeroExtendNode*)

```
next
  case (BinaryExpr op e1 e2)
  then have n ∈ ids g1
    using no-encoding by force
  then have kind g1 n = kind g2 n
    using assms unfolding as-set-def
    by blast
  then show ?case using BinaryExpr BinaryRepE
     by (smt (verit, ccfv-threshold) AddNode MulNode SubNode AndNode OrNode
XorNode IntegerBelowNode IntegerEqualsNode IntegerLessThanNode)
next
  case (ConditionalExpr e1 e2 e3)
  then have n ∈ ids g1
    using no-encoding by force
  then have kind g1 n = kind g2 n
    using assms unfolding as-set-def
    by blast
  then show ?case using ConditionalExpr ConditionalExprE
    by (smt (verit, best) ConditionalNode ConditionalNodeE)
next
  case (ConstantExpr x)
  then have n ∈ ids g1
    using no-encoding by force
  then have kind g1 n = kind g2 n
    using assms unfolding as-set-def
    by blast
  then show ?case using ConstantExpr ConstantExprE
    by (metis ConstantNode ConstantNodeE)
next
  case (ParameterExpr x1 x2)
  then have in-g1: n ∈ ids g1
    using no-encoding by force
  then have kinds: kind g1 n = kind g2 n
    using assms unfolding as-set-def
    by blast
  from in-g1 have stamps: stamp g1 n = stamp g2 n
    using assms unfolding as-set-def
    by blast
  from kinds stamps show ?case using ParameterExpr ParameterExprE
    by (metis ParameterNode ParameterNodeE)
next
  case (LeafExpr nid s)
  then have in-g1: n ∈ ids g1
    using no-encoding by force
  then have kinds: kind g1 n = kind g2 n
    using assms unfolding as-set-def
    by blast
  from in-g1 have stamps: stamp g1 n = stamp g2 n
    using assms unfolding as-set-def
```

```
    by blast
  from kinds stamps show ?case using LeafExpr LeafExprE LeafNode
   by (smt (z3) IRExpr.distinct(29) IRExpr.simps(16) IRExpr.simps(28) rep.simps)
next
  case (ConstantVar x)
  then have in-g1: n ∈ ids g1
    using no-encoding by force
  then have kinds: kind g1 n = kind g2 n
    using assms unfolding as-set-def
    by blast
  from in-g1 have stamps: stamp g1 n = stamp g2 n
    using assms unfolding as-set-def
    by blast
  from kinds stamps show ?case using ConstantVar
    using rep.simps by blast
next
  case (VariableExpr x s)
  then have in-g1: n ∈ ids g1
    using no-encoding by force
  then have kinds: kind g1 n = kind g2 n
    using assms unfolding as-set-def
    by blast
  from in-g1 have stamps: stamp g1 n = stamp g2 n
    using assms unfolding as-set-def
    by blast
  from kinds stamps show ?case using VariableExpr
    using rep.simps by blast
qed


lemma subset-refines:
  assumes as-set g1 ⊆ as-set g2
  shows graph-refinement g1 g2
proof −
  have ids g1 ⊆ ids g2 using assms unfolding as-set-def
    by blast
  show ?thesis unfolding graph-refinement-def
    apply (rule allI) apply (rule impI) apply (rule allI) apply (rule impI)
    proof −
      fix n e1
      assume 1:n ∈ ids g1
      assume 2:g1 ⊢ n ≃ e1

      show ∃ e2. (g2 ⊢ n ≃ e2) ∧ e1 ≥ e2
        using assms 1 2 using subset-implies-evals
        by (meson equal-refines)
    qed
  qed
```

lemma *graph-construction*:
  $e1 \geq e2$
  $\wedge$ *as-set g1* $\subseteq$ *as-set g2* $\wedge$ *maximal-sharing g1*
  $\wedge$ $(g2 \vdash n \simeq e2)$ $\wedge$ *maximal-sharing g2*
  $\Longrightarrow$ $(g2 \vdash n \trianglelefteq e1)$ $\wedge$ *graph-refinement g1 g2*
  using *subset-refines*
  by (*meson encodeeval-def graph-represents-expression-def le-expr-def*)

end

8 Control-flow Semantics

theory *IRStepObj*
  imports
    TreeToGraph
begin

8.1 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*$),\ 0)$

8.2 Intraprocedural Semantics

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* ⇒ *ID* ⇒ *ID list* **where**
  phi-list g n =
    (filter (λx.(is-PhiNode (kind g x)))
      (sorted-list-of-set (usages g n)))

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

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

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

Intraprocedural semantics are given as a small-step semantics.

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

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

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

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

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

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

$m' = \textit{set-phis phis vs m}\rrbracket$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (merge,\ m',\ h)\ |$

NewInstanceNode:
$\llbracket \textit{kind g nid} = (\textit{NewInstanceNode nid f obj nid}');$
$(h',\ ref) = \textit{h-new-inst h};$
$m' = m(nid := ref)\rrbracket$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')\ |$

LoadFieldNode:
$\llbracket \textit{kind g nid} = (\textit{LoadFieldNode nid f}\ (\textit{Some obj})\ nid');$
$g \vdash obj \simeq objE;$
$[m,\ p] \vdash objE \mapsto \textit{ObjRef ref};$
$\textit{h-load-field f ref h} = v;$
$m' = m(nid := v)\rrbracket$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h)\ |$

SignedDivNode:
$\llbracket \textit{kind g nid} = (\textit{SignedDivNode nid x y zero sb nxt});$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye;$
$[m,\ p] \vdash xe \mapsto v1;$
$[m,\ p] \vdash ye \mapsto v2;$
$v = (\textit{intval-div v1 v2});$
$m' = \ m(nid := v)\rrbracket$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (nxt,\ m',\ h)\ |$

SignedRemNode:
$\llbracket \textit{kind g nid} = (\textit{SignedRemNode nid x y zero sb nxt});$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye;$
$[m,\ p] \vdash xe \mapsto v1;$
$[m,\ p] \vdash ye \mapsto v2;$
$v = (\textit{intval-mod v1 v2});$
$m' = \ m(nid := v)\rrbracket$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (nxt,\ m',\ h)\ |$

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

StoreFieldNode:
$\llbracket \textit{kind g nid} = (\textit{StoreFieldNode nid f newval}\ \text{-}\ (\textit{Some obj})\ nid');$
$g \vdash newval \simeq newvalE;$
$g \vdash obj \simeq objE;$
$[m,\ p] \vdash newvalE \mapsto val;$

$[m, p] \vdash objE \mapsto ObjRef\ ref$;
$h' = h\text{-}store\text{-}field\ f\ ref\ val\ h$;
$m' = m(nid := val)]\!]$
$\Longrightarrow g, p \vdash (nid, m, h) \to (nid', m', h')$ |

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

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

8.3 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:
$[\![g, p \vdash (nid, m, h) \to (nid', m', h')]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#stk, h) \longrightarrow ((g,nid',m',p)\#stk, h')$ |

InvokeNodeStep:
$[\![is\text{-}Invoke\ (kind\ g\ nid)$;

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

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

$cm' = cm(cnid := v)$;

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

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

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

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

$g \vdash exception \simeq exceptionE;$
$[m,\ p] \vdash exceptionE \mapsto e;$

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

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

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

8.4 Big-step Execution

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

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

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

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

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

|
$\llbracket P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h');$

has-return m′;

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

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

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

8.4.1 Heap Testing

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

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

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

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

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

end

9 Properties of Control-flow Semantics

theory *IRStepThms*
imports
IRStepObj
IRTreeEvalThms
begin

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

theorem *stepDet*:
(*g*, *p* ⊢ (*nid*,*m*,*h*) → *next*) ⟹
(∀ *next*′. ((*g*, *p* ⊢ (*nid*,*m*,*h*) → *next*′) ⟶ *next* = *next*′))

proof (*induction rule*: *step.induct*)
case (*SequentialNode nid next m h*)
have *notif*: ¬(*is-IfNode* (*kind g nid*))
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-IfNode-def*)
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-AbstractEndNode.simps is-EndNode.elims*(*2*) *is-LoopEndNode-def*)
have *notnew*: ¬(*is-NewInstanceNode* (*kind g nid*))
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-NewInstanceNode-def*)
have *notload*: ¬(*is-LoadFieldNode* (*kind g nid*))

using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-LoadFieldNode-def*)
have *notstore*: ¬(*is-StoreFieldNode* (*kind g nid*))
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-StoreFieldNode-def*)
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps is-SignedDivNode-def is-SignedRemNode-def*
by (*metis is-IntegerDivRemNode.simps*)
from *notif notend notnew notload notstore notdivrem*
show *?case* **using** *SequentialNode step.cases*
by (*smt* (*z3*) *IRNode.disc*(*1028*) *IRNode.disc*(*2270*) *IRNode.discI*(*31*) *Pair-inject is-sequential-node.simps*(*18*) *is-sequential-node.simps*(*43*) *is-sequential-node.simps*(*44*))
next
case (*IfNode nid cond tb fb m val next h*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *IfNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *IfNode.hyps*(*1*))
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *IfNode.hyps*(*1*))
from *notseq notend notdivrem* **show** *?case* **using** *IfNode repDet evalDet IRNode.distinct IRNode.inject*(*11*) *Pair-inject step.simps*
by (*smt* (*z3*) *IRNode.distinct IRNode.inject*(*12*) *Pair-inject step.simps*)
next
case (*EndNodes nid merge i phis inputs m vs m′ h*)
have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-AbstractEndNode.simps is-sequential-node.simps*
by (*metis is-EndNode.elims*(*2*) *is-LoopEndNode-def*)
have *notif*: ¬(*is-IfNode* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-IfNode-def is-AbstractEndNode.elims*
by (*metis IRNode.distinct-disc*(*1058*) *is-EndNode.simps*(*12*))
have *notref*: ¬(*is-RefNode* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-sequential-node.simps*
using *IRNode.disc*(*1899*) *IRNode.distinct*(*1473*) *is-AbstractEndNode.simps is-EndNode.elims*(*2*) *is-LoopEndNode-def is-RefNode-def*
by *metis*
have *notnew*: ¬(*is-NewInstanceNode* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-AbstractEndNode.simps*
using *IRNode.distinct-disc*(*1442*) *is-EndNode.simps*(*29*) *is-NewInstanceNode-def*
by (*metis IRNode.distinct-disc*(*1901*) *is-EndNode.simps*(*32*))
have *notload*: ¬(*is-LoadFieldNode* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-AbstractEndNode.simps*
using *is-LoadFieldNode-def*
by (*metis IRNode.distinct-disc*(*1706*) *is-EndNode.simps*(*21*))
have *notstore*: ¬(*is-StoreFieldNode* (*kind g nid*))

using *EndNodes.hyps*(*1*) *is-AbstractEndNode.simps is-StoreFieldNode-def*
by (*metis IRNode.distinct-disc*(*1926*) *is-EndNode.simps*(*44*))
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-AbstractEndNode.simps is-SignedDivNode-def is-SignedRemNode-def*
using *IRNode.distinct-disc*(*1498*) *IRNode.distinct-disc*(*1500*) *is-IntegerDivRemNode.simps is-EndNode.simps*(*36*) *is-EndNode.simps*(*37*)
by *auto*
from *notseq notif notref notnew notload notstore notdivrem*
show *?case* **using** *EndNodes repAllDet evalAllDet*
by (*smt* (*z3*) *is-IfNode-def is-LoadFieldNode-def is-NewInstanceNode-def is-RefNode-def is-StoreFieldNode-def is-SignedDivNode-def is-SignedRemNode-def Pair-inject is-IntegerDivRemNode.elims*(*3*) *step.cases*)
next
case (*NewInstanceNode nid f obj nxt h′ ref h m′ m*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notif*: ¬(*is-IfNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notref*: ¬(*is-RefNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notload*: ¬(*is-LoadFieldNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notstore*: ¬(*is-StoreFieldNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
using *is-AbstractMergeNode.simps*
by (*simp add*: *NewInstanceNode.hyps*(*1*))
from *notseq notend notif notref notload notstore notdivrem*
show *?case* **using** *NewInstanceNode step.cases*
by (*smt* (*z3*) *IRNode.disc*(*1028*) *IRNode.disc*(*2270*) *IRNode.discI*(*11*) *IRNode.distinct*(*2311*) *IRNode.distinct*(*2313*) *IRNode.inject*(*31*) *Pair-inject*)
next
case (*LoadFieldNode nid f obj nxt m ref h v m′*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *LoadFieldNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *LoadFieldNode.hyps*(*1*))
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
using *is-AbstractEndNode.simps*

```
    by (simp add: LoadFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using LoadFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject Value.inject(3)
option.distinct(1) option.inject)
next
  case (StaticLoadFieldNode nid f nxt h v m' m)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StaticLoadFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StaticLoadFieldNode step.cases
   by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject option.distinct(1))
next
  case (StoreFieldNode nid f newval uu obj nxt m val ref h' h m')
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StoreFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StoreFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StoreFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StoreFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Value.inject(3)
option.distinct(1) option.inject)
next
  case (StaticStoreFieldNode nid f newval uv nxt m val h' h m')
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StaticStoreFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StoreFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Static-
```

StoreFieldNode.hyps(*1*) *StaticStoreFieldNode.hyps*(*2*) *StaticStoreFieldNode.hyps*(*3*) *StaticStoreFieldNode.hyps*(*4*) *StaticStoreFieldNode.hyps*(*5*) *option.distinct*(*1*))
next
  case (*SignedDivNode nid x y zero sb nxt m v1 v2 v m′ h*)
  then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
    using *is-sequential-node.simps is-AbstractMergeNode.simps*
    by (*simp add*: *SignedDivNode.hyps*(*1*))
  have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
    using *is-AbstractEndNode.simps*
    by (*simp add*: *SignedDivNode.hyps*(*1*))
  from *notseq notend*
  show *?case* **using** *SignedDivNode step.cases repDet evalDet*
   by (*smt* (*z3*) *IRNode.distinct*(*1091*) *IRNode.distinct*(*1739*) *IRNode.distinct*(*2311*) *IRNode.distinct*(*2601*) *IRNode.distinct*(*2605*) *IRNode.inject*(*40*) *Pair-inject*)
next
  case (*SignedRemNode nid x y zero sb nxt m v1 v2 v m′ h*)
  then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
    using *is-sequential-node.simps is-AbstractMergeNode.simps*
    by (*simp add*: *SignedRemNode.hyps*(*1*))
  have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
    using *is-AbstractEndNode.simps*
    by (*simp add*: *SignedRemNode.hyps*(*1*))
  from *notseq notend*
  show *?case* **using** *SignedRemNode step.cases repDet evalDet*
   by (*smt* (*z3*) *IRNode.distinct*(*1093*) *IRNode.distinct*(*1741*) *IRNode.distinct*(*2313*) *IRNode.distinct*(*2601*) *IRNode.distinct*(*2627*) *IRNode.inject*(*41*) *Pair-inject*)
qed

lemma *stepRefNode*:
  ⟦*kind g nid* = *RefNode nid′*⟧ ⟹ *g*, *p* ⊢ (*nid*,*m*,*h*) → (*nid′*,*m*,*h*)
  by (*simp add*: *SequentialNode*)

lemma *IfNodeStepCases*:
  assumes *kind g nid* = *IfNode cond tb fb*
  assumes *g* ⊢ *cond* ≃ *condE*
  assumes [*m*, *p*] ⊢ *condE* ↦ *v*
  assumes *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*)
  shows *nid′* ∈ {*tb*, *fb*}
  using *step.IfNode repDet stepDet assms*
  by (*metis insert-iff old.prod.inject*)

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

lemma *IfNodeCond*:
  assumes *kind g nid* = *IfNode cond tb fb*
  assumes *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*)
  shows ∃ *condE v*. ((*g* ⊢ *cond* ≃ *condE*) ∧ ([*m*, *p*] ⊢ *condE* ↦ *v*))

using *assms*(*2*,*1*) **by** (*induct* (*nid*,*m*,*h*) (*nid′*,*m*,*h*) *rule*: *step.induct*; *auto*)

lemma *step-in-ids*:
assumes g, $p \vdash (nid, m, h) \to (nid', m', h')$
shows $nid \in ids\ g$
using *assms* **apply** (*induct* (*nid*, *m*, *h*) (*nid′*, *m′*, *h′*) *rule*: *step.induct*)
using *is-sequential-node.simps*(*45*) *not-in-g*
apply *simp*
apply (*metis is-sequential-node.simps*(*53*))
using *ids-some*
using *IRNode.distinct*(*1113*) **apply** *presburger*
using *EndNodes*(*1*) *is-AbstractEndNode.simps is-EndNode.simps*(*45*) *ids-some*
apply (*metis IRNode.disc*(*1218*) *is-EndNode.simps*(*52*))
by *simp*+

end

10 Proof Infrastructure

10.1 Bisimulation

theory *Bisimulation*
imports
Stuttering
begin

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

A strong bisimilution between no-op transitions

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

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

using *assms*(*2*) *assms*(*3*) *stepDet strong-noop-bisimilar.simps* **by** *metis*

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

end

10.2 Formedness Properties

theory *Form*
imports
Semantics.TreeToGraph
begin

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

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

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

definition *wf-ends* **where**
wf-ends g =
(∀ *n* ∈ *ids g* .
is-AbstractEndNode (*kind g n*) ⟶
card (*usages g n*) > *0*)

fun *wf-graph* :: *IRGraph* ⇒ *bool* **where**
wf-graph g = (*wf-start g* ∧ *wf-closed g* ∧ *wf-phis g* ∧ *wf-ends g*)

lemmas *wf-folds* =
wf-graph.simps
wf-start-def

wf-closed-def
wf-phis-def
wf-ends-def

fun *wf-stamps* :: *IRGraph* ⇒ *bool* **where**
wf-stamps g = (∀ *n* ∈ *ids g* .
(∀ *v m p e* . (*g* ⊢ *n* ≃ *e*) ∧ ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ *valid-value* (*stamp-expr e*) *v*))

fun *wf-stamp* :: *IRGraph* ⇒ (*ID* ⇒ *Stamp*) ⇒ *bool* **where**
wf-stamp g s = (∀ *n* ∈ *ids g* .
(∀ *v m p e* . (*g* ⊢ *n* ≃ *e*) ∧ ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ *valid-value* (*s n*) *v*))

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

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

fun *wf-logic-node-inputs* :: *IRGraph* ⇒ *ID* ⇒ *bool* **where**
wf-logic-node-inputs g n =
(∀ *inp* ∈ *set* (*inputs-of* (*kind g n*)) . (∀ *v m p* . ([*g*, *m*, *p*] ⊢ *inp* ↦ *v*) ⟶ *wf-bool* *v*))

fun *wf-values* :: *IRGraph* ⇒ *bool* **where**
wf-values g = (∀ *n* ∈ *ids g* .
(∀ *v m p* . ([*g*, *m*, *p*] ⊢ *n* ↦ *v*) ⟶
(*is-LogicNode* (*kind g n*) ⟶
wf-bool v ∧ *wf-logic-node-inputs g n*)))

end

10.3 Dynamic Frames

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

theory *IRGraphFrames*
imports
Form
Semantics.IRTreeEval
begin

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

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

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

lemma *other-node-unchanged*:
assumes *changeonly ns g1 g2*
assumes *nid* ∈ *ids g1*
assumes *nid* ∉ *ns*
shows *kind g1 nid* = *kind g2 nid*
using *assms*
using *changeonly.simps* **by** *blast*

Some notation for input nodes used

inductive *eval-uses*:: *IRGraph* ⇒ *ID* ⇒ *ID* ⇒ *bool*
for *g* **where**

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

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

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

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

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

lemma *not-in-g-inputs*:
assumes *nid* ∉ *ids g*
shows *inputs g nid* = {}
proof −
have *k*: *kind g nid* = *NoNode* **using** *assms not-in-g* **by** *blast*
then show *?thesis* **by** (*simp add*: *k*)

qed

lemma *child-member*:
  assumes $n = kind\ g\ nid$
  assumes $n \neq NoNode$
  assumes $List.member\ (inputs\text{-}of\ n)\ child$
  shows $child \in inputs\ g\ nid$
  unfolding *inputs.simps* **using** *assms*
  by (*metis in-set-member*)

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

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

lemma *inp-in-g-wf*:
  assumes $wf\text{-}graph\ g$
  assumes $n \in inputs\ g\ nid$
  shows $n \in ids\ g$
  using *assms* **unfolding** *wf-folds*
  using *inp-in-g* **by** *blast*

lemma *kind-unchanged*:
  assumes $nid \in ids\ g1$
  assumes $unchanged\ (eval\text{-}usages\ g1\ nid)\ g1\ g2$
  shows $kind\ g1\ nid = kind\ g2\ nid$
proof –
  show *?thesis*
    using *assms eval-usages-self*

using *unchanged.simps* **by** *blast*
qed

lemma *stamp-unchanged*:
assumes *nid* ∈ *ids g1*
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows *stamp g1 nid* = *stamp g2 nid*
by (*meson assms*(*1*) *assms*(*2*) *eval-usages-self unchanged.elims*(*2*))

lemma *child-unchanged*:
assumes *child* ∈ *inputs g1 nid*
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows *unchanged* (*eval-usages g1 child*) *g1 g2*
by (*smt assms*(*1*) *assms*(*2*) *eval-usages.simps mem-Collect-eq unchanged.simps use-inp use-trans*)

lemma *eval-usages*:
assumes *us* = *eval-usages g nid*
assumes *nid′* ∈ *ids g*
shows *eval-uses g nid nid′* ⟷ *nid′* ∈ *us* (**is** *?P* ⟷ *?Q*)
using *assms eval-usages.simps*
by (*simp add*: *ids.rep-eq*)

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

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

lemma *inputs-of-are-usages*:
assumes *List.member* (*inputs-of* (*kind g nid*)) *nid′*
assumes *nid′* ∈ *ids g*
shows *nid′* ∈ *eval-usages g nid*
by (*metis assms*(*1*) *assms*(*2*) *in-set-member inputs.elims inputs-are-usages*)

lemma *usage-includes-inputs*:
assumes *us* = *eval-usages g nid*
assumes *ls* = *inputs g nid*
assumes *ls* ⊆ *ids g*
shows *ls* ⊆ *us*
using *inputs-are-usages eval-usages*
using *assms*(*1*) *assms*(*2*) *assms*(*3*) **by** *blast*

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

lemma *encode-in-ids*:
assumes $g \vdash nid \simeq e$
shows $nid \in ids\ g$
using *assms*
apply (*induction rule*: *rep.induct*)
apply *simp+*
by *fastforce*

lemma *eval-in-ids*:
assumes $[g, m, p] \vdash nid \mapsto v$
shows $nid \in ids\ g$
using *assms* **using** *encodeeval-def encode-in-ids*
by *auto*

lemma *transitive-kind-same*:
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows $\forall\ nid' \in (eval\text{-}usages\ g1\ nid)\ .\ kind\ g1\ nid' = kind\ g2\ nid'$
using *assms*
by (*meson unchanged.elims*(*1*))

theorem *stay-same-encoding*:
assumes *nc*: *unchanged* (*eval-usages g1 nid*) *g1 g2*
assumes *g1*: $g1 \vdash nid \simeq e$
assumes *wf*: *wf-graph g1*
shows $g2 \vdash nid \simeq e$
proof −
have *dom*: $nid \in ids\ g1$
using *g1 encode-in-ids* **by** *simp*
show *?thesis*
using *g1 nc wf dom* **proof** (*induction e rule*: *rep.induct*)
case (*ConstantNode n c*)
then have $kind\ g2\ n = ConstantNode\ c$
using *dom nc kind-unchanged*
by *metis*
then show *?case* **using** *rep.ConstantNode*
by *presburger*
next
case (*ParameterNode n i s*)
then have $kind\ g2\ n = ParameterNode\ i$
by (*metis kind-unchanged*)
then show *?case*
by (*metis ParameterNode.hyps*(*2*) *ParameterNode.prems*(*1*) *ParameterNode.prems*(*3*)

rep.ParameterNode stamp-unchanged)
next
case (*ConditionalNode n c t f ce te fe*)
then have *kind g2 n* = *ConditionalNode c t f*
by (*metis kind-unchanged*)
have $c \in$ *eval-usages g1 n* $\wedge$ $t \in$ *eval-usages g1 n* $\wedge$ $f \in$ *eval-usages g1 n*
using *inputs-of-ConditionalNode*
by (*metis ConditionalNode.hyps*(*1*) *ConditionalNode.hyps*(*2*) *ConditionalNode.hyps*(*3*) *ConditionalNode.hyps*(*4*) *encode-in-ids inputs.simps inputs-are-usages list.set-intros*(*1*) *set-subset-Cons subset-code*(*1*))
then show *?case* **using** *transitive-kind-same*
by (*metis ConditionalNode.hyps*(*1*) *ConditionalNode.prems*(*1*) *IRNodes.inputs-of-ConditionalNode* ‹*kind g2 n* = *ConditionalNode c t f*› *child-unchanged inputs.simps list.set-intros*(*1*) *local.ConditionalNode*(*5*) *local.ConditionalNode*(*6*) *local.ConditionalNode*(*7*) *local.ConditionalNode*(*9*) *rep.ConditionalNode set-subset-Cons subset-code*(*1*) *unchanged.elims*(*2*))
next
case (*AbsNode n x xe*)
then have *kind g2 n* = *AbsNode x*
using *kind-unchanged*
by *metis*
then have $x \in$ *eval-usages g1 n*
using *inputs-of-AbsNode*
by (*metis AbsNode.hyps*(*1*) *AbsNode.hyps*(*2*) *encode-in-ids inputs.simps inputs-are-usages list.set-intros*(*1*))
then show *?case*
by (*metis AbsNode.IH AbsNode.hyps*(*1*) *AbsNode.prems*(*1*) *AbsNode.prems*(*3*) *IRNodes.inputs-of-AbsNode* ‹*kind g2 n* = *AbsNode x*› *child-member-in child-unchanged local.wf member-rec*(*1*) *rep.AbsNode unchanged.simps*)
next
case (*NotNode n x xe*)
then have *kind g2 n* = *NotNode x*
using *kind-unchanged*
by *metis*
then have $x \in$ *eval-usages g1 n*
using *inputs-of-NotNode*
by (*metis NotNode.hyps*(*1*) *NotNode.hyps*(*2*) *encode-in-ids inputs.simps inputs-are-usages list.set-intros*(*1*))
then show *?case*
by (*metis NotNode.IH NotNode.hyps*(*1*) *NotNode.prems*(*1*) *NotNode.prems*(*3*) *IRNodes.inputs-of-NotNode* ‹*kind g2 n* = *NotNode x*› *child-member-in child-unchanged local.wf member-rec*(*1*) *rep.NotNode unchanged.simps*)
next
case (*NegateNode n x xe*)
then have *kind g2 n* = *NegateNode x*
using *kind-unchanged* **by** *metis*
then have $x \in$ *eval-usages g1 n*
using *inputs-of-NegateNode*
by (*metis NegateNode.hyps*(*1*) *NegateNode.hyps*(*2*) *encode-in-ids inputs.simps inputs-are-usages list.set-intros*(*1*))

then show *?case*
by (*metis IRNodes.inputs-of-NegateNode NegateNode.IH NegateNode.hyps*(*1*) *NegateNode.prems*(*1*) *NegateNode.prems*(*3*) ‹*kind g2 n* = *NegateNode x*› *child-member-in child-unchanged local.wf member-rec*(*1*) *rep.NegateNode unchanged.elims*(*1*))
next
case (*LogicNegationNode n x xe*)
then have *kind g2 n* = *LogicNegationNode x*
using *kind-unchanged* **by** *metis*
then have *x* ∈ *eval-usages g1 n*
using *inputs-of-LogicNegationNode inputs-of-are-usages*
by (*metis LogicNegationNode.hyps*(*1*) *LogicNegationNode.hyps*(*2*) *encode-in-ids member-rec*(*1*))
then show *?case*
by (*metis IRNodes.inputs-of-LogicNegationNode LogicNegationNode.IH LogicNegationNode.hyps*(*1*) *LogicNegationNode.hyps*(*2*) *LogicNegationNode.prems*(*1*) ‹*kind g2 n* = *LogicNegationNode x*› *child-unchanged encode-in-ids inputs.simps list.set-intros*(*1*) *local.wf rep.LogicNegationNode*)
next
case (*AddNode n x y xe ye*)
then have *kind g2 n* = *AddNode x y*
using *kind-unchanged* **by** *metis*
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
using *inputs-of-LogicNegationNode inputs-of-are-usages*
by (*metis AddNode.hyps*(*1*) *AddNode.hyps*(*2*) *AddNode.hyps*(*3*) *IRNodes.inputs-of-AddNode encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros*(*1*) *set-subset-Cons*)
then show *?case*
by (*metis AddNode.IH*(*1*) *AddNode.IH*(*2*) *AddNode.hyps*(*1*) *AddNode.hyps*(*2*) *AddNode.hyps*(*3*) *AddNode.prems*(*1*) *IRNodes.inputs-of-AddNode* ‹*kind g2 n* = *AddNode x y*› *child-unchanged encode-in-ids in-set-member inputs.simps local.wf member-rec*(*1*) *rep.AddNode*)
next
case (*MulNode n x y xe ye*)
then have *kind g2 n* = *MulNode x y*
using *kind-unchanged* **by** *metis*
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
using *inputs-of-LogicNegationNode inputs-of-are-usages*
by (*metis MulNode.hyps*(*1*) *MulNode.hyps*(*2*) *MulNode.hyps*(*3*) *IRNodes.inputs-of-MulNode encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros*(*1*) *set-subset-Cons*)
then show *?case* **using** *MulNode inputs-of-MulNode*
by (*metis* ‹*kind g2 n* = *MulNode x y*› *child-unchanged inputs.simps list.set-intros*(*1*) *rep.MulNode set-subset-Cons subset-iff unchanged.elims*(*2*))
next
case (*SubNode n x y xe ye*)
then have *kind g2 n* = *SubNode x y*
using *kind-unchanged* **by** *metis*
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
using *inputs-of-LogicNegationNode inputs-of-are-usages*
by (*metis SubNode.hyps*(*1*) *SubNode.hyps*(*2*) *SubNode.hyps*(*3*) *IRNodes.inputs-of-SubNode encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros*(*1*) *set-subset-Cons*)

then show *?case* **using** *SubNode inputs-of-SubNode*
by (*metis* ‹*kind g2 n* = *SubNode x y*› *child-member child-unchanged encode-in-ids ids-some member-rec*(*1*) *rep.SubNode*)
next
case (*AndNode n x y xe ye*)
then have *kind g2 n* = *AndNode x y*
using *kind-unchanged* **by** *metis*
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
using *inputs-of-LogicNegationNode inputs-of-are-usages*
by (*metis AndNode.hyps*(*1*) *AndNode.hyps*(*2*) *AndNode.hyps*(*3*) *IRNodes.inputs-of-AndNode encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros*(*1*) *set-subset-Cons*)
then show *?case* **using** *AndNode inputs-of-AndNode*
by (*metis* ‹*kind g2 n* = *AndNode x y*› *child-unchanged inputs.simps list.set-intros*(*1*) *rep.AndNode set-subset-Cons subset-iff unchanged.elims*(*2*))
next
case (*OrNode n x y xe ye*)
then have *kind g2 n* = *OrNode x y*
using *kind-unchanged* **by** *metis*
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
using *inputs-of-OrNode inputs-of-are-usages*
by (*metis OrNode.hyps*(*1*) *OrNode.hyps*(*2*) *OrNode.hyps*(*3*) *IRNodes.inputs-of-OrNode encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros*(*1*) *set-subset-Cons*)
then show *?case* **using** *OrNode inputs-of-OrNode*
by (*metis* ‹*kind g2 n* = *OrNode x y*› *child-member child-unchanged encode-in-ids ids-some member-rec*(*1*) *rep.OrNode*)
next
case (*XorNode n x y xe ye*)
then have *kind g2 n* = *XorNode x y*
using *kind-unchanged* **by** *metis*
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
using *inputs-of-XorNode inputs-of-are-usages*
by (*metis XorNode.hyps*(*1*) *XorNode.hyps*(*2*) *XorNode.hyps*(*3*) *IRNodes.inputs-of-XorNode encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros*(*1*) *set-subset-Cons*)
then show *?case* **using** *XorNode inputs-of-XorNode*
by (*metis* ‹*kind g2 n* = *XorNode x y*› *child-member child-unchanged encode-in-ids ids-some member-rec*(*1*) *rep.XorNode*)
next
case (*IntegerBelowNode n x y xe ye*)
then have *kind g2 n* = *IntegerBelowNode x y*
using *kind-unchanged* **by** *metis*
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
using *inputs-of-IntegerBelowNode inputs-of-are-usages*
by (*metis IntegerBelowNode.hyps*(*1*) *IntegerBelowNode.hyps*(*2*) *IntegerBelowNode.hyps*(*3*) *IRNodes.inputs-of-IntegerBelowNode encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros*(*1*) *set-subset-Cons*)
then show *?case* **using** *IntegerBelowNode inputs-of-IntegerBelowNode*
by (*metis* ‹*kind g2 n* = *IntegerBelowNode x y*› *child-member child-unchanged encode-in-ids ids-some member-rec*(*1*) *rep.IntegerBelowNode*)
next

case (*IntegerEqualsNode n x y xe ye*)
then have *kind g2 n* = *IntegerEqualsNode x y*
using *kind-unchanged* **by** *metis*
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
using *inputs-of-IntegerEqualsNode inputs-of-are-usages*
by (*metis IntegerEqualsNode.hyps*(*1*) *IntegerEqualsNode.hyps*(*2*) *IntegerEqualsNode.hyps*(*3*) *IRNodes.inputs-of-IntegerEqualsNode encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros*(*1*) *set-subset-Cons*)
then show *?case* **using** *IntegerEqualsNode inputs-of-IntegerEqualsNode*
by (*metis* ‹*kind g2 n* = *IntegerEqualsNode x y*› *child-member child-unchanged encode-in-ids ids-some member-rec*(*1*) *rep.IntegerEqualsNode*)
next
case (*IntegerLessThanNode n x y xe ye*)
then have *kind g2 n* = *IntegerLessThanNode x y*
using *kind-unchanged* **by** *metis*
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
using *inputs-of-IntegerLessThanNode inputs-of-are-usages*
by (*metis IntegerLessThanNode.hyps*(*1*) *IntegerLessThanNode.hyps*(*2*) *IntegerLessThanNode.hyps*(*3*) *IRNodes.inputs-of-IntegerLessThanNode encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros*(*1*) *set-subset-Cons*)
then show *?case* **using** *IntegerLessThanNode inputs-of-IntegerLessThanNode*
by (*metis* ‹*kind g2 n* = *IntegerLessThanNode x y*› *child-member child-unchanged encode-in-ids ids-some member-rec*(*1*) *rep.IntegerLessThanNode*)
next
case (*NarrowNode n ib rb x xe*)
then have *kind g2 n* = *NarrowNode ib rb x*
using *kind-unchanged* **by** *metis*
then have $x \in$ *eval-usages g1 n*
using *inputs-of-NarrowNode inputs-of-are-usages*
by (*metis NarrowNode.hyps*(*1*) *NarrowNode.hyps*(*2*) *IRNodes.inputs-of-NarrowNode encode-in-ids inputs.simps inputs-are-usages list.set-intros*(*1*))
then show *?case* **using** *NarrowNode inputs-of-NarrowNode*
by (*metis* ‹*kind g2 n* = *NarrowNode ib rb x*› *child-unchanged inputs.elims list.set-intros*(*1*) *rep.NarrowNode unchanged.simps*)
next
case (*SignExtendNode n ib rb x xe*)
then have *kind g2 n* = *SignExtendNode ib rb x*
using *kind-unchanged* **by** *metis*
then have $x \in$ *eval-usages g1 n*
using *inputs-of-SignExtendNode inputs-of-are-usages*
by (*metis SignExtendNode.hyps*(*1*) *SignExtendNode.hyps*(*2*) *encode-in-ids inputs.simps inputs-are-usages list.set-intros*(*1*))
then show *?case* **using** *SignExtendNode inputs-of-SignExtendNode*
by (*metis* ‹*kind g2 n* = *SignExtendNode ib rb x*› *child-member-in child-unchanged in-set-member list.set-intros*(*1*) *rep.SignExtendNode unchanged.elims*(*2*))
next
case (*ZeroExtendNode n ib rb x xe*)
then have *kind g2 n* = *ZeroExtendNode ib rb x*
using *kind-unchanged* **by** *metis*

then have *x* ∈ *eval-usages g1 n*
using *inputs-of-ZeroExtendNode inputs-of-are-usages*
by (*metis ZeroExtendNode.hyps*(*1*) *ZeroExtendNode.hyps*(*2*) *IRNodes.inputs-of-ZeroExtendNode encode-in-ids inputs.simps inputs-are-usages list.set-intros*(*1*))
then show *?case* **using** *ZeroExtendNode inputs-of-ZeroExtendNode*
by (*metis* ‹*kind g2 n* = *ZeroExtendNode ib rb x*› *child-member-in child-unchanged member-rec*(*1*) *rep.ZeroExtendNode unchanged.simps*)
next
case (*LeafNode n s*)
then show *?case*
by (*metis kind-unchanged rep.LeafNode stamp-unchanged*)
qed
qed

theorem *stay-same*:
assumes *nc*: *unchanged* (*eval-usages g1 nid*) *g1 g2*
assumes *g1*: [*g1*, *m*, *p*] ⊢ *nid* ↦ *v1*
assumes *wf*: *wf-graph g1*
shows [*g2*, *m*, *p*] ⊢ *nid* ↦ *v1*
proof −
have *nid*: *nid* ∈ *ids g1*
using *g1 eval-in-ids* **by** *simp*
then have *nid* ∈ *eval-usages g1 nid*
using *eval-usages-self* **by** *blast*
then have *kind-same*: *kind g1 nid* = *kind g2 nid*
using *nc node-unchanged* **by** *blast*
obtain *e* **where** *e*: (*g1* ⊢ *nid* ≃ *e*) ∧ ([*m*,*p*] ⊢ *e* ↦ *v1*)
using *encodeeval-def g1*
by *auto*
then have *val*: [*m*,*p*] ⊢ *e* ↦ *v1*
using *g1 encodeeval-def*
by *simp*
then show *?thesis* **using** *e nid nc*
unfolding *encodeeval-def*
proof (*induct e v1 arbitrary*: *nid rule*: *evaltree.induct*)
case (*ConstantExpr c*)
then show *?case*
by (*metis ConstantNode ConstantNodeE kind-unchanged*)
next
case (*ParameterExpr i s*)
have *g2* ⊢ *nid* ≃ *ParameterExpr i s*
using *stay-same-encoding ParameterExpr*
by (*meson local.wf*)
then show *?case* **using** *evaltree.ParameterExpr*
by (*meson ParameterExpr.hyps*)
next
case (*ConditionalExpr ce cond branch te fe v*)

then have *g2* ⊢ *nid* ≃ *ConditionalExpr ce te fe*
using *ConditionalExpr.prems*(*1*) *ConditionalExpr.prems*(*3*) *local.wf stay-same-encoding*
by *presburger*
then show *?case*
by (*metis ConditionalExpr.prems*(*1*))
next
case (*UnaryExpr xe v op*)
then show *?case*
using *local.wf stay-same-encoding* **by** *blast*
next
case (*BinaryExpr xe x ye y op*)
then show *?case*
using *local.wf stay-same-encoding* **by** *blast*
next
case (*LeafExpr val nid s*)
then show *?case*
by (*metis local.wf stay-same-encoding*)
qed
qed

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

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

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

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

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

lemma *no-external-use*:
  assumes *wf-graph g*
  assumes $nid' \notin ids\ g$
  assumes $nid \in ids\ g$
  shows $\neg(eval\text{-}uses\ g\ nid\ nid')$
proof –
  have *0*: $nid \neq nid'$
    using *assms* **by** *blast*
  have *inp*: $nid' \notin inputs\ g\ nid$
    using *assms*
    using *inp-in-g-wf* **by** *blast*
  have *rec-0*: $\nexists n\ .\ n \in ids\ g \wedge n = nid'$
    using *assms* **by** *blast*
  have *rec-inp*: $\nexists n\ .\ n \in ids\ g \wedge n \in inputs\ g\ nid'$
    using *assms*(*2*) *inp-in-g* **by** *blast*
  have *rec*: $\nexists nid''\ .\ eval\text{-}uses\ g\ nid\ nid'' \wedge eval\text{-}uses\ g\ nid''\ nid'$
    using *wf-use-ids assms*(*1*) *assms*(*2*) *assms*(*3*) **by** *blast*
  from *inp 0 rec* **show** *?thesis*
    using *eval-uses-imp* **by** *blast*
qed

end

10.4 Graph Rewriting

theory
  Rewrites
imports
  IRGraphFrames
  Stuttering
begin

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

lemma *replace-usages-effect*:
  assumes *g′* = *replace-usages nid nid′ g*
  shows *kind g′ nid* = *RefNode nid′*
  using *assms replace-node-lookup replace-usages.simps*
  by (*metis IRNode.distinct*(*2755*))

lemma *replace-usages-changeonly*:
  assumes *nid* ∈ *ids g*
  assumes *g′* = *replace-usages nid nid′ g*
  shows *changeonly* {*nid*} *g g′*
  using *assms* **unfolding** *replace-usages.simps*
  by (*metis add-changed add-node-def replace-node-def*)

lemma *replace-usages-unchanged*:
  assumes *nid* ∈ *ids g*
  assumes *g′* = *replace-usages nid nid′ g*
  shows *unchanged* (*ids g* − {*nid*}) *g g′*
  using *assms* **unfolding** *replace-usages.simps*
  using *assms*(*2*) *disjoint-change replace-usages-changeonly* **by** *presburger*

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

lemma *max-plus-one*:
  fixes *c* :: *ID set*
  shows ⟦*finite c*; *c* ≠ {}⟧ ⟹ (*Max c*) + *1* ∉ *c*
  by (*meson Max-gr-iff less-add-one less-irrefl*)

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

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

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

lemma *constantConditionTrue*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes *g′* = *constantCondition True ifcond* (*kind g ifcond*) *g*
shows *g′*, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
proof –
have *ifn*: ⋀ *c t f*. *IfNode c t f* ≠ *NoNode*
by *simp*
then have *if′*: *kind g′ ifcond* = *IfNode* (*nextNid g*) *t f*
using *assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *replace-node-lookup*
by *presburger*
have *truedef*: *bool-to-val True* = (*IntVal32 1*)
by *auto*
from *ifn* **have** *ifcond* ≠ (*nextNid g*)
by (*metis assms*(*1*) *emptyE ids-some nextNidNotIn*)
moreover have ⋀ *c*. *ConstantNode c* ≠ *NoNode* **by** *simp*
ultimately have *kind g′* (*nextNid g*) = *ConstantNode* (*IRTreeEval.bool-to-val True*)
using *add-changed add-node-def assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *not-in-g other-node-unchanged replace-node-def replace-node-lookup singletonD*
by (*smt* (*z3*) *DiffI add-node-lookup replace-node-unchanged*)
then have *c′*: *kind g′* (*nextNid g*) = *ConstantNode* (*IntVal32 1*)
using *truedef* **by** *simp*
have *valid-value* (*constantAsStamp* (*IntVal32 1*)) (*IntVal32 1*)
unfolding *constantAsStamp.simps valid-value.simps*
using *nat-numeral* **by** *blast*
then have [*g′*, *m*, *p*] ⊢ *nextNid g* ↦ *IntVal32 1*
using *ConstantExpr ConstantNode Value.distinct*(*1*) ‹*kind g′* (*nextNid g*) = *ConstantNode* (*IRTreeEval.bool-to-val True*)› *encodeeval-def truedef*
by *metis*
from *if′ c′* **show** *?thesis* **using** *IfNode*
by (*metis* (*no-types*, *hide-lams*) *IRTreeEval.val-to-bool.simps*(*1*) ‹[*g′*,*m*,*p*] ⊢ *nextNid g* ↦ *IntVal32 1*› *encodeeval-def zero-neq-one*)
qed

lemma *constantConditionFalse*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes *g′* = *constantCondition False ifcond* (*kind g ifcond*) *g*
shows *g′*, *p* ⊢ (*ifcond*, *m*, *h*) → (*f*, *m*, *h*)
proof –
have *ifn*: ⋀ *c t f*. *IfNode c t f* ≠ *NoNode*
by *simp*

then have *if'*: *kind g' ifcond* = *IfNode* (*nextNid g*) *t f*
by (*metis assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *replace-node-lookup*)
have *falsedef*: *bool-to-val False* = (*IntVal32 0*)
by *auto*
from *ifn* **have** *ifcond* ≠ (*nextNid g*)
by (*metis assms*(*1*) *equals0D ids-some nextNidNotIn*)
moreover have ⋀ *c*. *ConstantNode c* ≠ *NoNode* **by** *simp*
ultimately have *kind g'* (*nextNid g*) = *ConstantNode* (*IRTreeEval.bool-to-val False*)
by (*smt* (*z3*) *add-changed add-node-def assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *not-in-g other-node-unchanged replace-node-def replace-node-lookup singletonD*)
then have *c'*: *kind g'* (*nextNid g*) = *ConstantNode* (*IntVal32 0*)
using *falsedef* **by** *simp*
have *valid-value* (*constantAsStamp* (*IntVal32 0*)) (*IntVal32 0*)
unfolding *constantAsStamp.simps valid-value.simps*
using *nat-numeral* **by** *blast*
then have [*g'*, *m*, *p*] ⊢ *nextNid g* ↦ *IntVal32 0*
by (*metis ConstantExpr ConstantNode* ‹*kind g'* (*nextNid g*) = *ConstantNode* (*IRTreeEval.bool-to-val False*)› *encodeeval-def falsedef*)
from *if' c'* **show** *?thesis* **using** *IfNode*
by (*metis* (*no-types*, *hide-lams*) *IRTreeEval.val-to-bool.simps*(*1*) ‹[*g'*,*m*,*p*] ⊢ *nextNid g* ↦ *IntVal32 0*› *encodeeval-def*)
qed

lemma *diff-forall*:
assumes ∀ *n*∈*ids g* − {*nid*}. *cond n*
shows ∀ *n*. *n* ∈ *ids g* ∧ *n* ∉ {*nid*} ⟶ *cond n*
by (*meson Diff-iff assms*)

lemma *replace-node-changeonly*:
assumes *g'* = *replace-node nid node g*
shows *changeonly* {*nid*} *g g'*
using *assms replace-node-unchanged*
unfolding *changeonly.simps* **using** *diff-forall*
by (*metis add-changed add-node-def changeonly.simps replace-node-def*)

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

lemma *constantConditionNoEffect*:
assumes ¬(*is-IfNode* (*kind g nid*))
shows *g* = *constantCondition b nid* (*kind g nid*) *g*
using *assms* **apply** (*cases kind g nid*)
using *constantCondition.simps*
apply *presburger+*

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

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

lemma *constantCondition-changeonly*:
assumes $nid \in ids\ g$
assumes $g' = constantCondition\ b\ nid\ (kind\ g\ nid)\ g$
shows $changeonly\ \{nid\}\ g\ g'$
proof (*cases is-IfNode (kind g nid)*)
case *True*
have $nextNid\ g \notin ids\ g$
using *nextNidNotIn* **by** (*metis emptyE*)
then show *?thesis* **using** *assms*
using *replace-node-changeonly add-node-changeonly* **unfolding** *changeonly.simps*
using *True constantCondition.simps(1) is-IfNode-def*
by (*metis (no-types, lifting) insert-iff*)
next
case *False*
have $g = g'$
using *constantConditionNoEffect*
using *False assms(2)* **by** *blast*
then show *?thesis* **by** *simp*
qed

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

lemma *constantConditionValid*:
assumes *kind g ifcond = IfNode cond t f*
assumes $[g, m, p] \vdash cond \mapsto v$

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

end

10.5 Stuttering

theory *Stuttering*
imports
Semantics.IRStepThms
begin

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

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

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

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

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

end

11 Canonicalization Phase

theory *CanonicalizationTree*
imports
Semantics.TreeToGraph
Semantics.IRTreeEval
begin

fun *is-idempotent-binary* :: *IRBinaryOp* ⇒ *bool* **where**
is-idempotent-binary BinAnd = *True* |
is-idempotent-binary BinOr = *True* |
is-idempotent-binary - = *False*

fun *is-idempotent-unary* :: *IRUnaryOp* ⇒ *bool* **where**
is-idempotent-unary UnaryAbs = *True* |
is-idempotent-unary - = *False*

fun *is-self-inverse* :: *IRUnaryOp* ⇒ *bool* **where**
is-self-inverse UnaryNeg = *True* |
is-self-inverse UnaryNot = *True* |
is-self-inverse UnaryLogicNegation = *True* |
is-self-inverse - = *False*

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

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

is-neutral BinSub (*IntVal32 x*) = ($x = 0$) |

is-neutral BinSub $(IntVal64\ x) = (x = 0)$ |

is-neutral BinMul $(IntVal32\ x) = (x = 1)$ |
is-neutral BinMul $(IntVal64\ x) = (x = 1)$ |

is-neutral BinAnd $(IntVal32\ x) = (x = 1)$ |
is-neutral BinAnd $(IntVal64\ x) = (x = 1)$ |

is-neutral BinOr $(IntVal32\ x) = (x = 0)$ |
is-neutral BinOr $(IntVal64\ x) = (x = 0)$ |

is-neutral BinXor $(IntVal32\ x) = (x = 0)$ |
is-neutral BinXor $(IntVal64\ x) = (x = 0)$ |

is-neutral - - = *False*

fun *is-annihilator* :: *IRBinaryOp* $\Rightarrow$ *Value* $\Rightarrow$ *bool* **where**

is-annihilator BinMul $(IntVal32\ x) = (x = 0)$ |
is-annihilator BinMul $(IntVal64\ x) = (x = 0)$ |

is-annihilator BinAnd $(IntVal32\ x) = (x = 0)$ |
is-annihilator BinAnd $(IntVal64\ x) = (x = 0)$ |

is-annihilator BinOr $(IntVal32\ x) = (x = 1)$ |
is-annihilator BinOr $(IntVal64\ x) = (x = 1)$ |

is-annihilator - - = *False*

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

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

binary-fold-yneutral:
$[\![y = (ConstantExpr\ c);$
is-neutral op c;
stampx = *stamp-expr x*;
stampy = *stamp-expr y*;

$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy]\!]$
$\Longrightarrow CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ x\ |$

$binary\text{-}fold\text{-}yzero32:$
$[\![y = ConstantExpr\ c;$
$is\text{-}annihilator\ op\ c;$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
$stp\text{-}bits\ stampx = 32;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy]\!]$
$\Longrightarrow CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ (ConstantExpr\ c)\ |$

$binary\text{-}fold\text{-}yzero64:$
$[\![y = ConstantExpr\ c;$
$is\text{-}annihilator\ op\ c;$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
$stp\text{-}bits\ stampx = 64;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy]\!]$
$\Longrightarrow CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ (ConstantExpr\ c)\ |$

$binary\text{-}idempotent:$
$[\![is\text{-}idempotent\text{-}binary\ op]\!]$
$\Longrightarrow CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ x)\ x$

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

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

$mul\text{-}negate32:$
$[\![y = ConstantExpr\ (IntVal32\ (-1));$
$stamp\text{-}expr\ x = IntegerStamp\ 32\ lo\ hi]\!]$
$\Longrightarrow CanonicalizeMul\ (BinaryExpr\ BinMul\ x\ y)\ (UnaryExpr\ UnaryNeg\ x)\ |$
$mul\text{-}negate64:$
$[\![y = ConstantExpr\ (IntVal64\ (-1));$
$stamp\text{-}expr\ x = IntegerStamp\ 64\ lo\ hi]\!]$
$\Longrightarrow CanonicalizeMul\ (BinaryExpr\ BinMul\ x\ y)\ (UnaryExpr\ UnaryNeg\ x)$

inductive $CanonicalizeAdd :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ **where**
$add\text{-}xsub:$

$[\![x = (BinaryExpr\ BinSub\ a\ y);$
$stampa = stamp\text{-}expr\ a;$
$stampy = stamp\text{-}expr\ y;$
$is\text{-}IntegerStamp\ stampa \wedge is\text{-}IntegerStamp\ stampy;$
$stp\text{-}bits\ stampa = stp\text{-}bits\ stampy]\!]$
$\Longrightarrow CanonicalizeAdd\ (BinaryExpr\ BinAdd\ x\ y)\ a \mid$

add-ysub:

$[\![y = (BinaryExpr\ BinSub\ a\ x);$
$stampa = stamp\text{-}expr\ a;$
$stampx = stamp\text{-}expr\ x;$
$is\text{-}IntegerStamp\ stampa \wedge is\text{-}IntegerStamp\ stampx;$
$stp\text{-}bits\ stampa = stp\text{-}bits\ stampx]\!]$
$\Longrightarrow CanonicalizeAdd\ (BinaryExpr\ BinAdd\ x\ y)\ a \mid$

add-xnegate:

$[\![nx = (UnaryExpr\ UnaryNeg\ x);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy]\!]$
$\Longrightarrow CanonicalizeAdd\ (BinaryExpr\ BinAdd\ nx\ y)\ (BinaryExpr\ BinSub\ y\ x) \mid$

add-ynegate:

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

inductive *CanonicalizeSub* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**

sub-same32:

$[\![stampx = stamp\text{-}expr\ x;$
$stampx = IntegerStamp\ 32\ lo\ hi]\!]$
$\Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ x\ x)\ (ConstantExpr\ (IntVal32\ 0)) \mid$
sub-same64:

$[\![stampx = stamp\text{-}expr\ x;$
$stampx = IntegerStamp\ 64\ lo\ hi]\!]$
$\Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ x\ x)\ (ConstantExpr\ (IntVal64\ 0)) \mid$

sub-left-add1:

$[\![x = (BinaryExpr\ BinAdd\ a\ b);$
$\ \ stampa = stamp\text{-}expr\ a;$
$\ \ stampb = stamp\text{-}expr\ b;$
$\ \ is\text{-}IntegerStamp\ stampa \wedge is\text{-}IntegerStamp\ stampb;$
$\ \ stp\text{-}bits\ stampa = stp\text{-}bits\ stampb]\!]$
$\ \ \Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ x\ b)\ a\ |$

sub-left-add2:

$[\![x = (BinaryExpr\ BinAdd\ a\ b);$
$\ \ stampa = stamp\text{-}expr\ a;$
$\ \ stampb = stamp\text{-}expr\ b;$
$\ \ is\text{-}IntegerStamp\ stampa \wedge is\text{-}IntegerStamp\ stampb;$
$\ \ stp\text{-}bits\ stampa = stp\text{-}bits\ stampb]\!]$
$\ \ \Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ x\ a)\ b\ |$

sub-left-sub:

$[\![x = (BinaryExpr\ BinSub\ a\ b);$
$\ \ stampa = stamp\text{-}expr\ a;$
$\ \ stampb = stamp\text{-}expr\ b;$
$\ \ is\text{-}IntegerStamp\ stampa \wedge is\text{-}IntegerStamp\ stampb;$
$\ \ stp\text{-}bits\ stampa = stp\text{-}bits\ stampb]\!]$
$\ \ \Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ x\ a)\ (UnaryExpr\ UnaryNeg\ b)\ |$

sub-right-add1:

$[\![y = (BinaryExpr\ BinAdd\ a\ b);$
$\ \ stampa = stamp\text{-}expr\ a;$
$\ \ stampb = stamp\text{-}expr\ b;$
$\ \ is\text{-}IntegerStamp\ stampa \wedge is\text{-}IntegerStamp\ stampb;$
$\ \ stp\text{-}bits\ stampa = stp\text{-}bits\ stampb]\!]$
$\ \ \Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ a\ y)\ (UnaryExpr\ UnaryNeg\ b)\ |$

sub-right-add2:

$[\![y = (BinaryExpr\ BinAdd\ a\ b);$
$\ \ stampa = stamp\text{-}expr\ a;$
$\ \ stampb = stamp\text{-}expr\ b;$
$\ \ is\text{-}IntegerStamp\ stampa \wedge is\text{-}IntegerStamp\ stampb;$
$\ \ stp\text{-}bits\ stampa = stp\text{-}bits\ stampb]\!]$
$\ \ \Longrightarrow CanonicalizeSub\ (BinaryExpr\ BinSub\ b\ y)\ (UnaryExpr\ UnaryNeg\ a)\ |$

sub-right-sub:

$[\![y = (BinaryExpr\ BinSub\ a\ b);$

stampa = *stamp-expr a*;
stampb = *stamp-expr b*;
is-IntegerStamp stampa ∧ *is-IntegerStamp stampb*;
stp-bits stampa = *stp-bits stampb*⟧
⟹ *CanonicalizeSub* (*BinaryExpr BinSub a y*) *b* |

sub-xzero32:
⟦*stampx* = *stamp-expr x*;
stampx = *IntegerStamp 32 lo hi*⟧
⟹ *CanonicalizeSub* (*BinaryExpr BinSub* (*ConstantExpr* (*IntVal32 0*)) *x*) (*UnaryExpr UnaryNeg x*) |
sub-xzero64:
⟦*stampx* = *stamp-expr x*;
stampx = *IntegerStamp 64 lo hi*⟧
⟹ *CanonicalizeSub* (*BinaryExpr BinSub* (*ConstantExpr* (*IntVal64 0*)) *x*) (*UnaryExpr UnaryNeg x*) |

sub-y-negate:

⟦*nb* = (*UnaryExpr UnaryNeg b*);
stampa = *stamp-expr a*;
stampb = *stamp-expr b*;
is-IntegerStamp stampa ∧ *is-IntegerStamp stampb*;
stp-bits stampa = *stp-bits stampb*⟧
⟹ *CanonicalizeSub* (*BinaryExpr BinSub a nb*) (*BinaryExpr BinAdd a b*)

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

⟦*nx* = (*UnaryExpr UnaryNeg x*);
is-IntegerStamp (*stamp-expr x*)⟧
⟹ *CanonicalizeNegate* (*UnaryExpr UnaryNeg nx*) *x* |

negate-sub:

⟦*e* = (*BinaryExpr BinSub x y*);
stampx = *stamp-expr x*;
stampy = *stamp-expr y*;
is-IntegerStamp stampx ∧ *is-IntegerStamp stampy*;
stp-bits stampx = *stp-bits stampy*⟧
⟹ *CanonicalizeNegate* (*UnaryExpr UnaryNeg e*) (*BinaryExpr BinSub y x*)

inductive *CanonicalizeAbs* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**

abs-abs:

$[\![ax = (UnaryExpr\ UnaryAbs\ x);$
$is\text{-}IntegerStamp\ (stamp\text{-}expr\ x)]\!]$
$\Longrightarrow CanonicalizeAbs\ (UnaryExpr\ UnaryAbs\ ax)\ ax\ |$

abs-neg:

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

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

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

inductive *CanonicalizeAnd* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**
and-same:

$[\![is\text{-}IntegerStamp\ (stamp\text{-}expr\ x)]\!]$
$\Longrightarrow CanonicalizeAnd\ (BinaryExpr\ BinAnd\ x\ x)\ x\ |$

and-demorgans:

$[\![nx = (UnaryExpr\ UnaryNot\ x);$
$ny = (UnaryExpr\ UnaryNot\ y);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy]\!]$
$\Longrightarrow CanonicalizeAnd\ (BinaryExpr\ BinAnd\ nx\ ny)\ (UnaryExpr\ UnaryNot\ (BinaryExpr\ BinOr\ x\ y))$

inductive *CanonicalizeOr* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**
or-same:

$[\![is\text{-}IntegerStamp\ (stamp\text{-}expr\ x)]\!]$
$\Longrightarrow CanonicalizeOr\ (BinaryExpr\ BinOr\ x\ x)\ x\ |$

or-demorgans:

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

$ny = (UnaryExpr\ UnaryNot\ y)$;
$stampx = stamp\text{-}expr\ x$;
$stampy = stamp\text{-}expr\ y$;
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy$;
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy$⟧
$\Longrightarrow$ *CanonicalizeOr* (*BinaryExpr BinOr nx ny*) (*UnaryExpr UnaryNot* (*BinaryExpr BinAnd x y*))

inductive *CanonicalizeIntegerEquals* :: *IRExpr* ⇒ *IRExpr* ⇒ *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*)$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left right*) (*BinaryExpr BinIntegerEquals y z*) |

int-equals-left-contains-right1:

$[\![$*left* = (*BinaryExpr BinAdd x y*);
zero = (*ConstantExpr* (*IntVal32 0*))$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left x*) (*BinaryExpr BinIntegerEquals y zero*) |

int-equals-left-contains-right2:

$[\![$*left* = (*BinaryExpr BinAdd x y*);
zero = (*ConstantExpr* (*IntVal32 0*))$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals left y*) (*BinaryExpr BinIntegerEquals x zero*) |

int-equals-right-contains-left1:

$[\![$*right* = (*BinaryExpr BinAdd x y*);
zero = (*ConstantExpr* (*IntVal32 0*))$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals x right*) (*BinaryExpr BinIntegerEquals y zero*) |

int-equals-right-contains-left2:

$[\![$*right* = (*BinaryExpr BinAdd x y*);
zero = (*ConstantExpr* (*IntVal32 0*))$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals* (*BinaryExpr BinIntegerEquals y right*) (*BinaryExpr BinIntegerEquals x zero*) |

int-equals-left-contains-right3:

$[\![left = (BinaryExpr\ BinSub\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ left\ x)\ (BinaryExpr\ BinIntegerEquals\ y\ zero)\ |$

int-equals-right-contains-left3:

$[\![right = (BinaryExpr\ BinSub\ x\ y);$
$zero = (ConstantExpr\ (IntVal32\ 0))]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ (BinaryExpr\ BinIntegerEquals\ x\ right)\ (BinaryExpr\ BinIntegerEquals\ y\ zero)$

inductive *CanonicalizeConditional* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* **where**
eq-branches:

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

cond-eq:

$[\![c = (BinaryExpr\ BinIntegerEquals\ x\ y);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy]\!]$
$\Longrightarrow CanonicalizeConditional\ (ConditionalExpr\ c\ x\ y)\ y\ |$

condition-bounds-x:

$[\![c = (BinaryExpr\ BinIntegerLessThan\ x\ y);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$stpi\text{-}upper\ stampx \leq stpi\text{-}lower\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy]\!]$
$\Longrightarrow CanonicalizeConditional\ (ConditionalExpr\ c\ x\ y)\ x\ |$

condition-bounds-y:

$[\![c = (BinaryExpr\ BinIntegerLessThan\ x\ y);$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$stpi\text{-}upper\ stampx \leq stpi\text{-}lower\ stampy;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy]\!]$
$\Longrightarrow CanonicalizeConditional\ (ConditionalExpr\ c\ y\ x)\ y\ |$

negate-condition:

$[\![nc = (UnaryExpr\ UnaryLogicNegation\ c);$
$stampc = stamp\text{-}expr\ c;$
$stampc = IntegerStamp\ 32\ lo\ hi;$
$stampx = stamp\text{-}expr\ x;$
$stampy = stamp\text{-}expr\ y;$
$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
$is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy]\!]$
$\Longrightarrow CanonicalizeConditional\ (ConditionalExpr\ nc\ x\ y)\ (ConditionalExpr\ c\ y\ x)$
|

const-true:

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

const-false:

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

inductive *CanonicalizationStep* :: *IRExpr* ⇒ *IRExpr* ⇒ *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

12 Canonicalization Phase

theory *CanonicalizationTreeProofs*
imports
CanonicalizationTree
Semantics.TreeToGraph

Semantics.IRTreeEvalThms
begin

lemma *neutral-rewrite-helper*:
shows *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-mul x* (*IntVal32* (*1*)) = *x*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-mul x* (*IntVal64* (*1*)) = *x*

and *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-add x* (*IntVal32* (*0*)) = *x*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-add x* (*IntVal64* (*0*)) = *x*

and *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-sub x* (*IntVal32* (*0*)) = *x*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-sub x* (*IntVal64* (*0*)) = *x*

and *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-xor x* (*IntVal32* (*0*)) = *x*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-xor x* (*IntVal64* (*0*)) = *x*

and *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-or x* (*IntVal32* (*0*)) = *x*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-or x* (*IntVal64* (*0*)) = *x*
using *valid32or64-both* **by** *fastforce+*

lemma *annihilator-rewrite-helper*:
shows *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-mul x* (*IntVal32 0*) = *IntVal32 0*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-mul x* (*IntVal64 0*) = *IntVal64 0*

and *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-and x* (*IntVal32 0*) = *IntVal32 0*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-and x* (*IntVal64 0*) = *IntVal64 0*

and *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-or x* (*IntVal32* ($-$*1*)) = *IntVal32* ($-$*1*)
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-or x* (*IntVal64* ($-$*1*)) = *IntVal64* ($-$*1*)
using *valid32or64-both*
apply *auto*
apply (*metis intval-mul.simps*(*1*) *mult-zero-right valid32*)
by *fastforce+*

lemma *idempotent-rewrite-helper*:
shows *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-and x x* = *x*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-and x x* = *x*

and *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-or x x* = *x*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-or x x* = *x*
using *valid32or64-both*
apply *auto*
by *fastforce+*

value *size* (*v*::*32 word*)

lemma *signed-int-bottom32*: $-(((2::int)\ \widehat{}\ 31)) \leq sint\ (v::int32)$
proof –
  have *size v* = *32* **apply** (*cases v*; *auto*) **sorry**
  then show *?thesis*
    using *sint-range-size* **sorry**
qed

lemma *signed-int-top32*: $(2\ \widehat{}\ 31) - 1 \geq sint\ (v::int32)$
proof –
  have *size v* = *32* **sorry**
  then show *?thesis*
    using *sint-range-size* **sorry**
qed

lemma *lower-bounds-equiv32*: $-(((2::int)\ \widehat{}\ 31)) = (2::int)\ \widehat{}\ 32\ div\ 2 * - 1$
  by *fastforce*

lemma *upper-bounds-equiv32*: $(2::int)\ \widehat{}\ 31 = (2::int)\ \widehat{}\ 32\ div\ 2$
  by *simp*

lemma *bit-bounds-min32*: $((fst\ (bit\text{-}bounds\ 32))) \leq (sint\ (v::int32))$
 unfolding *bit-bounds.simps fst-def* **using** *signed-int-bottom32 lower-bounds-equiv32*
  by *auto*

lemma *bit-bounds-max32*: $((snd\ (bit\text{-}bounds\ 32))) \geq (sint\ (v::int32))$
  unfolding *bit-bounds.simps fst-def* **using** *signed-int-top32 upper-bounds-equiv32*
  by *auto*

value *size* (*v*::*64 word*)

lemma *signed-int-bottom64*: $-(((2::int)\ \widehat{}\ 63)) \leq sint\ (v::int64)$
proof –
  have *size v* = *64* **apply** (*cases v*; *auto*) **sorry**
  then show *?thesis*
    using *sint-range-size* **sorry**
qed

lemma *signed-int-top64*: $(2\ \widehat{}\ 63) - 1 \geq sint\ (v::int64)$
proof –
  have *size v* = *32* **sorry**
  then show *?thesis*
    using *sint-range-size* **sorry**
qed

lemma *lower-bounds-equiv64*: $-(((2::int)\ \widehat{}\ 63)) = (2::int)\ \widehat{}\ 64\ div\ 2 * - 1$
  by *fastforce*

```
lemma upper-bounds-equiv64: (2::int) ^ 63 = (2::int) ^ 64 div 2
  by simp

lemma bit-bounds-min64: ((fst (bit-bounds 64))) ≤ (sint (v::int64))
  unfolding bit-bounds.simps fst-def using signed-int-bottom64 lower-bounds-equiv64
  by auto

lemma bit-bounds-max64: ((snd (bit-bounds 64))) ≥ (sint (v::int64))
  unfolding bit-bounds.simps fst-def using signed-int-top64 upper-bounds-equiv64
  by auto

lemma unrestricted-32bit-always-valid:
  valid-value (unrestricted-stamp (IntegerStamp 32 lo hi)) (IntVal32 v)
  using valid-value.simps(1) bit-bounds-min32 bit-bounds-max32
  using unrestricted-stamp.simps(2) by presburger

lemma unrestricted-64bit-always-valid:
  valid-value (unrestricted-stamp (IntegerStamp 64 lo hi)) (IntVal64 v)
  using valid-value.simps(2) bit-bounds-min64 bit-bounds-max64
  using unrestricted-stamp.simps(2) by presburger

lemma unary-undef: val = UndefVal ⟹ unary-eval op val = UndefVal
  by (cases op; auto)

lemma unary-obj: val = ObjRef x ⟹ unary-eval op val = UndefVal
  by (cases op; auto)

lemma unary-eval-implies-valud-value:
  assumes [m,p] ⊢ expr ↦ val
  assumes result = unary-eval op val
  assumes result ≠ UndefVal
  assumes valid-value (stamp-expr expr) val
  shows valid-value (stamp-expr (UnaryExpr op expr)) result
proof –
  have is-IntVal: ∃ x y. result = IntVal32 x ∨ result = IntVal64 y
    using assms(2,3) apply (cases op; auto; cases val; auto)
    by metis
  then have is-IntegerStamp (stamp-expr expr)
    using assms(2,3,4) apply (cases (stamp-expr expr); auto)
    using valid-VoidStamp unary-undef apply simp
    using valid-VoidStamp unary-undef apply simp
    using valid-ObjStamp unary-obj apply fastforce
    using valid-ObjStamp unary-obj by fastforce
  then obtain b lo hi where stamp-expr-def: stamp-expr expr = (IntegerStamp b
lo hi)
    using is-IntegerStamp-def by auto
  then have stamp-expr (UnaryExpr op expr) = unrestricted-stamp (IntegerStamp
b lo hi)
```

using *stamp-expr.simps*(*1*) *stamp-unary.simps*(*1*) **by** *presburger*
from *stamp-expr-def* **have** *bit32*: $b = 32 \Longrightarrow \exists\ x.\ result = IntVal32\ x$
using *assms*(*2*,*3*,*4*) **by** (*cases op*; *auto*; *cases val*; *auto*)
from *stamp-expr-def* **have** *bit64*: $b = 64 \Longrightarrow \exists\ x.\ result = IntVal64\ x$
using *assms*(*2*,*3*,*4*) **by** (*cases op*; *auto*; *cases val*; *auto*)

show *?thesis* **using** *valid-value.simps*(*1*,*2*)
unrestricted-32bit-always-valid unrestricted-64bit-always-valid stamp-expr-def bit32 bit64
by (*metis* ‹*stamp-expr* (*UnaryExpr op expr*) = *unrestricted-stamp* (*IntegerStamp b lo hi*)› *assms*(*4*) *valid32or64-both*)
qed

lemma *binary-undef*: $v1 = UndefVal \lor v2 = UndefVal \Longrightarrow bin\text{-}eval\ op\ v1\ v2 = UndefVal$
by (*cases op*; *auto*)

lemma *binary-obj*: $v1 = ObjRef\ x \lor v2 = ObjRef\ y \Longrightarrow bin\text{-}eval\ op\ v1\ v2 = UndefVal$
by (*cases op*; *auto*)

lemma *binary-eval-bits-equal*:
assumes $result = bin\text{-}eval\ op\ val1\ val2$
assumes $result \neq UndefVal$
assumes *valid-value* (*IntegerStamp b1 lo1 hi1*) *val1*
assumes *valid-value* (*IntegerStamp b2 lo2 hi2*) *val2*
shows $b1 = b2$
using *assms*
by (*cases op*; *cases val1*; *cases val2*; *auto*)

lemma *binary-eval-values*:
assumes $\exists x\ y.\ result = IntVal32\ x \lor result = IntVal64\ y$
assumes $result = bin\text{-}eval\ op\ val1\ val2$
shows $\exists x32\ x64\ y32\ y64.\ val1 = IntVal32\ x32 \land val2 = IntVal32\ y32 \lor val1 = IntVal64\ x64 \land val2 = IntVal64\ y64$
using *assms* **apply** (*cases result*)
apply *simp* **apply** (*cases op*; *cases val1*; *cases val2*; *auto*)
apply (*cases op*; *cases val1*; *cases val2*; *auto*) **by** *auto*+

lemma *binary-eval-implies-valud-value*:
assumes $[m,p] \vdash expr1 \mapsto val1$
assumes $[m,p] \vdash expr2 \mapsto val2$
assumes $result = bin\text{-}eval\ op\ val1\ val2$
assumes $result \neq UndefVal$
assumes *valid-value* (*stamp-expr expr1*) *val1*
assumes *valid-value* (*stamp-expr expr2*) *val2*
shows *valid-value* (*stamp-expr* (*BinaryExpr op expr1 expr2*)) *result*
proof –
have *is-IntVal*: $\exists\ x\ y.\ result = IntVal32\ x \lor result = IntVal64\ y$

using *assms*(*1*,*2*,*3*,*4*) **apply** (*cases op*; *auto*; *cases val1*; *auto*; *cases val2*; *auto*)
by (*meson Values.bool-to-val.elims*)+
then have *expr1-intstamp*: *is-IntegerStamp* (*stamp-expr expr1*)
using *assms*(*1*,*3*,*4*,*5*) **apply** (*cases* (*stamp-expr expr1*); *auto simp*: *valid-VoidStamp binary-undef*)
using *valid-ObjStamp binary-obj* **apply** (*metis assms*(*4*))
using *valid-ObjStamp binary-obj* **by** (*metis assms*(*4*))
from *is-IntVal* **have** *expr2-intstamp*: *is-IntegerStamp* (*stamp-expr expr2*)
using *assms*(*2*,*3*,*4*,*6*) **apply** (*cases* (*stamp-expr expr2*); *auto simp*: *valid-VoidStamp binary-undef*)
using *valid-ObjStamp binary-obj* **apply** (*metis assms*(*4*))
using *valid-ObjStamp binary-obj* **by** (*metis assms*(*4*))
from *expr1-intstamp* **obtain** *b1 lo1 hi1* **where** *stamp-expr1-def*: *stamp-expr expr1* = (*IntegerStamp b1 lo1 hi1*)
using *is-IntegerStamp-def* **by** *auto*
from *expr2-intstamp* **obtain** *b2 lo2 hi2* **where** *stamp-expr2-def*: *stamp-expr expr2* = (*IntegerStamp b2 lo2 hi2*)
using *is-IntegerStamp-def* **by** *auto*

have ∃ *x32 x64 y32 y64* . (*val1* = *IntVal32 x32* ∧ *val2* = *IntVal32 y32*) ∨ (*val1* = *IntVal64 x64* ∧ *val2* = *IntVal64 y64*)
using *is-IntVal assms*(*3*) *binary-eval-values*
by *presburger*

have *b1* = *b2*
using *assms*(*3*,*4*,*5*,*6*) *stamp-expr1-def stamp-expr2-def*
using *binary-eval-bits-equal*
by *auto*
then have *stamp-def*: *stamp-expr* (*BinaryExpr op expr1 expr2*) =
(*case op* ∈ *fixed-32 of True* ⇒ *unrestricted-stamp* (*IntegerStamp 32 lo1 hi1*)| *False* ⇒ *unrestricted-stamp* (*IntegerStamp b1 lo1 hi1*))
using *stamp-expr.simps*(*2*) *stamp-binary.simps*(*1*)
using *stamp-expr1-def stamp-expr2-def* **by** *presburger*
from *stamp-expr1-def* **have** *bit32*: *b1* = *32* ⟹ ∃ *x*. *result* = *IntVal32 x*
using *assms* **apply** (*cases op*; *cases val1*; *cases val2*; *auto*)
by (*meson Values.bool-to-val.elims*)+
from *stamp-expr1-def* **have** *bit64*: *b1* = *64* ∧ *op* ∉ *fixed-32* ⟹ ∃ *x y*. *result* = *IntVal64 x*
using *assms* **apply** (*cases op*; *cases val1*; *cases val2*; *simp*)
using *fixed-32-def* **by** *auto*+
from *stamp-expr1-def* **have** *fixed*: *op* ∈ *fixed-32* ⟹ ∃ *x y*. *result* = *IntVal32 x*
using *assms* **unfolding** *fixed-32-def* **apply** (*cases op*; *auto*)
apply (*cases val1*; *cases val2*; *auto*)
using *bit32* **apply** *fastforce*
apply (*meson Values.bool-to-val.elims*)
apply (*cases val1*; *cases val2*; *auto*)
using *bit32* **apply** *fastforce*
apply (*meson Values.bool-to-val.elims*)
apply (*cases val1*; *cases val2*; *auto*)

using *bit32* **apply** *fastforce*
by (*meson Values.bool-to-val.elims*)

show *?thesis* **apply** (*cases op* ∈ *fixed-32*) **defer using** *valid-value.simps*(*1,2*)
unrestricted-32bit-always-valid unrestricted-64bit-always-valid stamp-expr1-def
bit32 bit64 stamp-def **apply** *auto*
using ‹∃ *x32 x64 y32 y64*. *val1* = *IntVal32 x32* ∧ *val2* = *IntVal32 y32* ∨ *val1* = *IntVal64 x64* ∧ *val2* = *IntVal64 y64*› *assms*(*5*) **apply** *auto*[*1*]
using *fixed* **by** *force*
qed

lemma *stamp-meet-is-valid*:
assumes *valid-value stamp1 val* ∨ *valid-value stamp2 val*
assumes *meet stamp1 stamp2* ≠ *IllegalStamp*
shows *valid-value* (*meet stamp1 stamp2*) *val*
using *assms* **proof** (*cases stamp1*)
case *VoidStamp*
then show *?thesis*
by (*metis Stamp.exhaust assms*(*1*) *assms*(*2*) *meet.simps*(*1*) *meet.simps*(*37*) *meet.simps*(*44*) *meet.simps*(*51*) *meet.simps*(*58*) *meet.simps*(*65*) *meet.simps*(*66*) *meet.simps*(*67*))
next
case (*IntegerStamp b lo hi*)
obtain *b2 lo2 hi2* **where** *stamp2-def*: *stamp2* = *IntegerStamp b2 lo2 hi2*
by (*metis IntegerStamp assms*(*2*) *meet.simps*(*45*) *meet.simps*(*52*) *meet.simps*(*59*) *meet.simps*(*6*) *meet.simps*(*65*) *meet.simps*(*66*) *meet.simps*(*67*) *unrestricted-stamp.cases*)
then have *b* = *b2* **using** *meet.simps*(*2*) *assms*(*2*)
by (*metis IntegerStamp*)
then have *meet-def*: *meet stamp1 stamp2* = (*IntegerStamp b* (*min lo lo2*) (*max hi hi2*))
by (*simp add*: *IntegerStamp stamp2-def*)
then show *?thesis* **proof** (*cases b* = *32*)
case *True*
then obtain *x* **where** *val-def*: *val* = *IntVal32 x*
using *IntegerStamp assms*(*1*) *valid32*
using ‹*b* = *b2*› *stamp2-def* **by** *blast*
have *min*: *sint x* ≥ *min lo lo2*
using *val-def*
using *IntegerStamp assms*(*1*)
using *stamp2-def* **by** *force*
have *max*: *sint x* ≤ *max hi hi2*
using *val-def*
using *IntegerStamp assms*(*1*)
using *stamp2-def* **by** *force*
from *min max* **show** *?thesis*
by (*simp add*: *True meet-def val-def*)
next
case *False*
then have *bit64*: *b* = *64*
using *assms*(*1*) *IntegerStamp valid-value.simps*

```
      valid32or64-both
      by (metis ‹b = b2› stamp2-def)
    then obtain x where val-def: val = IntVal64 x
      using IntegerStamp assms(1) valid64
      using ‹b = b2› stamp2-def by blast
    have min: sint x ≥ min lo lo2
      using val-def
      using IntegerStamp assms(1)
      using stamp2-def by force
    have max: sint x ≤ max hi hi2
      using val-def
      using IntegerStamp assms(1)
      using stamp2-def by force
    from min max show ?thesis
      by (simp add: bit64 meet-def val-def)
  qed
next
  case (KlassPointerStamp x31 x32)
  then show ?thesis using assms
   by (metis meet.simps(13) meet.simps(14) meet.simps(65) meet.simps(67) unrestricted-stamp.cases valid-value.simps(10) valid-value.simps(11) valid-value.simps(16) valid-value.simps(9))
next
  case (MethodCountersPointerStamp x41 x42)
  then show ?thesis using assms
   by (metis meet.simps(20) meet.simps(21) meet.simps(24) meet.simps(67) unrestricted-stamp.cases valid-value.simps(10) valid-value.simps(11) valid-value.simps(16) valid-value.simps(9))
next
  case (MethodPointersStamp x51 x52)
then show ?thesis using assms
 by (smt (z3) is-stamp-empty.elims(1) meet.simps(27) meet.simps(28) meet.simps(65) meet.simps(67) valid-value.simps(10) valid-value.simps(11) valid-value.simps(16) valid-value.simps(9))
next
  case (ObjectStamp x61 x62 x63 x64)
  then show ?thesis using assms
    using meet.simps(34) by blast
next
  case (RawPointerStamp x71 x72)
  then show ?thesis using assms
    using meet.simps(35) by blast
next
  case IllegalStamp
  then show ?thesis using assms
    using meet.simps(36) by blast
qed
```

lemma *conditional-eval-implies-valud-value*:
  assumes $[m,p] \vdash cond \mapsto condv$
  assumes $expr = (if\ IRTreeEval.val\text{-}to\text{-}bool\ condv\ then\ expr1\ else\ expr2)$
  assumes $[m,p] \vdash expr \mapsto val$
  assumes $val \neq UndefVal$
  assumes *valid-value* (*stamp-expr cond*) *condv*
  assumes *valid-value* (*stamp-expr expr*) *val*
  shows *valid-value* (*stamp-expr* (*ConditionalExpr cond expr1 expr2*)) *val*
proof –
  have *meet* (*stamp-expr expr1*) (*stamp-expr expr2*) $\neq$ *IllegalStamp*
    using *assms* **apply** (*cases stamp-expr expr*; *auto*)
    using *valid-VoidStamp* **apply** *blast* **sorry**
  then show *?thesis* **using** *stamp-meet-is-valid* **using** *stamp-expr.simps*(*6*)
    using *assms*(*2*) *assms*(*6*) **by** *presburger*
qed

lemma *stamp-implies-valid-value*:
  assumes $[m,p] \vdash expr \mapsto val$
  shows *valid-value* (*stamp-expr expr*) *val*
  using *assms* **proof** (*induction expr val*)
case (*UnaryExpr expr val result op*)
  then show *?case* **using** *unary-eval-implies-valud-value* **by** *simp*
next
  case (*BinaryExpr expr1 val1 expr2 val2 result op*)
  then show *?case* **using** *binary-eval-implies-valud-value* **by** *simp*
next
  case (*ConditionalExpr cond condv expr expr1 expr2 val*)
  then show *?case* **using** *conditional-eval-implies-valud-value* **by** *simp*
next
  case (*ParameterExpr x1 x2*)
  then show *?case* **by** *auto*
next
  case (*LeafExpr x1 x2*)
  then show *?case* **by** *auto*
next
  case (*ConstantExpr x*)
  then show *?case* **by** *auto*
qed

lemma *CanonicalizeBinaryProof*:
  assumes *CanonicalizeBinaryOp before after*
  assumes $[m, p] \vdash before \mapsto res$
  assumes $[m, p] \vdash after \mapsto res'$
  shows $res = res'$
  using *assms*
proof (*induct rule*: *CanonicalizeBinaryOp.induct*)
  case (*binary-const-fold x val1 y val2 val op*)
  then show *?case* **by** *auto*
next

case (*binary-fold-yneutral y c op stampx x stampy*)
obtain *xval* **where** *x-eval*: $[m, p] \vdash x \mapsto xval$
using *binary-fold-yneutral.prems(2)* **by** *auto*
then have *bin-eval op xval c = xval*
using *neutral-rewrite-helper binary-fold-yneutral.hyps(2−3,6−) stamp-implies-valid-value is-IntegerStamp-def*
sorry
then show *?case*
by (*metis binary-fold-yneutral.hyps(1) binary-fold-yneutral.prems(1) binary-fold-yneutral.prems(2) x-eval*
BinaryExprE ConstantExprE evalDet)
next
case (*binary-fold-yzero32 y c op stampx x stampy*)
obtain *xval* **where** *x-eval*: $[m, p] \vdash x \mapsto xval$
using *binary-fold-yzero32.prems(1)* **by** *auto*
then have *bin-eval op xval c = c*
using *annihilator-rewrite-helper binary-fold-yzero32.hyps stamp-implies-valid-value is-IntegerStamp-def*
sorry
then show *?case*
by (*metis BinaryExprE ConstantExprE binary-fold-yzero32.hyps(1) binary-fold-yzero32.prems(1) binary-fold-yzero32.prems(2) evalDet x-eval*)

next
case (*binary-fold-yzero64 y c op stampx x stampy*)
obtain *xval* **where** *x-eval*: $[m, p] \vdash x \mapsto xval$
using *binary-fold-yzero64.prems(1)* **by** *auto*
then have *bin-eval op xval c = c*
using *annihilator-rewrite-helper*
sorry
then show *?case*
by (*metis BinaryExprE ConstantExprE binary-fold-yzero64.hyps(1)*
binary-fold-yzero64.prems(1) binary-fold-yzero64.prems(2) evalDet x-eval)

next
case (*binary-idempotent op x*)
obtain *xval* **where** *x-eval*: $[m, p] \vdash x \mapsto xval$
using *binary-idempotent.prems(1)* **by** *auto*
then have *bin-eval op xval xval = xval*
using *idempotent-rewrite-helper binary-idempotent.hyps*
sorry
then show *?case*
by (*metis BinaryExprE binary-idempotent.prems(1) binary-idempotent.prems(2) evalDet x-eval*)

qed

lemma *CanonicalizeUnaryProof*:
assumes *CanonicalizeUnaryOp before after*

assumes $[m, p] \vdash$ *before* $\mapsto$ *res*
assumes $[m, p] \vdash$ *after* $\mapsto$ *res'*
shows *res* = *res'*
using *assms*
proof (*induct rule*: *CanonicalizeUnaryOp.induct*)
case (*unary-const-fold val' op val*)
then show *?case* **by** *auto*
qed

lemma *mul-rewrite-helper*:
shows *valid-value* (*IntegerStamp 32 lo hi*) *x* $\Longrightarrow$ *intval-mul x* (*IntVal32* (-1)) = *intval-negate x*
and *valid-value* (*IntegerStamp 64 lo hi*) *x* $\Longrightarrow$ *intval-mul x* (*IntVal64* (-1)) = *intval-negate x*
using *valid32or64-both* **by** *fastforce+*

lemma *CanonicalizeMulProof*:
assumes *CanonicalizeMul before after*
assumes $[m, p] \vdash$ *before* $\mapsto$ *res*
assumes $[m, p] \vdash$ *after* $\mapsto$ *res'*
shows *res* = *res'*
using *assms*
proof (*induct rule*: *CanonicalizeMul.induct*)
case (*mul-negate32 y x lo hi*)
then show *?case*
using *ConstantExprE BinaryExprE bin-eval.simps evalDet mul-rewrite-helper stamp-implies-valid-value*
by (*auto*; *metis*)
next
case (*mul-negate64 y x lo hi*)
then show *?case*
using *ConstantExprE BinaryExprE bin-eval.simps evalDet mul-rewrite-helper stamp-implies-valid-value*
by (*auto*; *metis*)
qed

lemma *add-rewrites-helper*:
assumes *valid-value* (*IntegerStamp b lox hix*) *x*
and *valid-value* (*IntegerStamp b loy hiy*) *y*

shows *intval-add* (*intval-sub x y*) *y* = *x*
and *intval-add x* (*intval-sub y x*) = *y*
and *intval-add* (*intval-negate x*) *y* = *intval-sub y x*
and *intval-add x* (*intval-negate y*) = *intval-sub x y*
using *valid32or64-both assms* **by** *fastforce+*

lemma *CanonicalizeAddProof*:
  assumes *CanonicalizeAdd before after*
  assumes $[m, p] \vdash$ *before* $\mapsto$ *res*
  assumes $[m, p] \vdash$ *after* $\mapsto$ *res'*
  shows *res* = *res'*
  using *assms*
proof (*induct rule*: *CanonicalizeAdd.induct*)
  case (*add-xsub x a y stampa stampy*)
  then show *?case*
    by (*metis BinaryExprE Stamp.collapse*(*1*) *bin-eval.simps*(*1*) *bin-eval.simps*(*3*)
        evalDet stamp-implies-valid-value intval-add-sym add-rewrites-helper(*1*))
next
  case (*add-ysub y a x stampa stampx*)
  then show *?case*
    by (*metis is-IntegerStamp-def add-ysub.hyps add-ysub.prems evalDet BinaryExprE Stamp.sel*(*1*)
      bin-eval.simps(*1*) *bin-eval.simps*(*3*) *stamp-implies-valid-value intval-add-sym add-rewrites-helper*(*2*))
next
  case (*add-xnegate nx x stampx stampy y*)
  then show *?case*
   by (*smt* (*verit*, *del-insts*) *BinaryExprE Stamp.sel*(*1*) *UnaryExprE add-rewrites-helper*(*4*)

        bin-eval.simps(*1*) *bin-eval.simps*(*3*) *evalDet stamp-implies-valid-value intval-add-sym is-IntegerStamp-def unary-eval.simps*(*2*))
next
  case (*add-ynegate ny y stampx x stampy*)
  then show *?case*
    by (*smt* (*verit*) *BinaryExprE Stamp.sel*(*1*) *UnaryExprE add-rewrites-helper*(*4*) *bin-eval.simps*(*1*)
        bin-eval.simps(*3*) *evalDet stamp-implies-valid-value is-IntegerStamp-def unary-eval.simps*(*2*))
qed

lemma *sub-rewrites-helper*:
  assumes *valid-value* (*IntegerStamp b lox hix*) *x*
  and *valid-value* (*IntegerStamp b loy hiy*) *y*

  shows *intval-sub* (*intval-add x y*) *y* = *x*
  and *intval-sub* (*intval-add x y*) *x* = *y*
  and *intval-sub* (*intval-sub x y*) *x* = *intval-negate y*

and *intval-sub x* (*intval-add x y*) = *intval-negate y*
and *intval-sub y* (*intval-add x y*) = *intval-negate x*
and *intval-sub x* (*intval-sub x y*) = *y*
and *intval-sub x* (*intval-negate y*) = *intval-add x y*
using *valid32or64-both assms* **by** *fastforce+*

lemma *sub-single-rewrites-helper*:
assumes *valid-value* (*IntegerStamp b lox hix*) *x*
shows $b = 32 \Longrightarrow$ *intval-sub x x* = *IntVal32 0*
and $b = 64 \Longrightarrow$ *intval-sub x x* = *IntVal64 0*
and $b = 32 \Longrightarrow$ *intval-sub* (*IntVal32 0*) *x* = *intval-negate x*
and $b = 64 \Longrightarrow$ *intval-sub* (*IntVal64 0*) *x* = *intval-negate x*
using *valid32or64-both assms* **by** *fastforce+*

lemma *CanonicalizeSubProof*:
assumes *CanonicalizeSub before after*
assumes $[m, p] \vdash$ *before* $\mapsto$ *res*
assumes $[m, p] \vdash$ *after* $\mapsto$ *res'*
shows *res* = *res'*
using *assms*
proof (*induct rule*: *CanonicalizeSub.induct*)
case (*sub-same32 stampx x lo hi*)
show *?case*
using *ConstantExprE BinaryExprE bin-eval.simps evalDet sub-same32.prems sub-single-rewrites-helper*
stamp-implies-valid-value sub-same32.hyps(*1*) *sub-same32.hyps*(*2*)
by (*auto*; *metis*)
next
case (*sub-same64 stampx x lo hi*)
show *?case*
using *ConstantExprE BinaryExprE bin-eval.simps evalDet sub-same64.prems sub-single-rewrites-helper*
stamp-implies-valid-value sub-same64.hyps(*1*) *sub-same64.hyps*(*2*)
by (*auto*; *metis*)
next
case (*sub-left-add1 x a b stampa stampb*)
then show *?case*
by (*metis BinaryExprE Stamp.collapse*(*1*) *bin-eval.simps*(*1*) *bin-eval.simps*(*3*) *evalDet*
stamp-implies-valid-value sub-rewrites-helper(*1*))
next
case (*sub-left-add2 x a b stampa stampb*)
then show *?case*
by (*metis BinaryExprE Stamp.collapse*(*1*) *bin-eval.simps*(*1*) *bin-eval.simps*(*3*) *evalDet*
stamp-implies-valid-value sub-rewrites-helper(*2*))
next

case (*sub-left-sub x a b stampa stampb*)
then show *?case*
by (*smt* (*verit*) *BinaryExprE Stamp.sel*(*1*) *UnaryExprE bin-eval.simps*(*3*) *evalDet*
stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(*3*) *unary-eval.simps*(*2*))
next
case (*sub-right-add1 y a b stampa stampb*)
then show *?case*
by (*smt* (*verit*) *BinaryExprE Stamp.sel*(*1*) *UnaryExprE bin-eval.simps*(*1*) *bin-eval.simps*(*3*) *evalDet*
stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(*4*) *unary-eval.simps*(*2*))
next
case (*sub-right-add2 y a b stampa stampb*)
then show *?case*
by (*smt* (*verit*) *BinaryExprE Stamp.sel*(*1*) *UnaryExprE bin-eval.simps*(*1*) *bin-eval.simps*(*3*) *evalDet*
stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(*5*) *unary-eval.simps*(*2*))
next
case (*sub-right-sub y a b stampa stampb*)
then show *?case*
by (*metis BinaryExprE Stamp.sel*(*1*) *bin-eval.simps*(*3*) *evalDet*
stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(*6*))
next
case (*sub-xzero32 stampx x lo hi*)
then show *?case*
using *ConstantExprE BinaryExprE bin-eval.simps evalDet sub-xzero32.prems sub-single-rewrites-helper*
stamp-implies-valid-value sub-xzero32.hyps(*1*) *sub-xzero32.hyps*(*2*)
by (*auto*; *metis*)
next
case (*sub-xzero64 stampx x lo hi*)
then show *?case*
using *ConstantExprE BinaryExprE bin-eval.simps evalDet sub-xzero64.prems sub-single-rewrites-helper*
stamp-implies-valid-value sub-xzero64.hyps(*1*) *sub-xzero64.hyps*(*2*)
by (*auto*; *metis*)
next
case (*sub-y-negate nb b stampa a stampb*)
then show *?case*
by (*smt* (*verit*, *best*) *BinaryExprE Stamp.sel*(*1*) *UnaryExprE bin-eval.simps*(*1*) *bin-eval.simps*(*3*) *evalDet*
stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(*7*) *unary-eval.simps*(*2*))
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*

using *valid32or64-both assms* **by** *fastforce*

lemma *negate-negate-helper*:
assumes *valid-value* (*IntegerStamp b lox hix*) *x*
shows *intval-negate* (*intval-negate x*) = *x*
using *valid32or64 assms* **by** *fastforce*

lemma *CanonicalizeNegateProof*:
assumes *CanonicalizeNegate before after*
assumes $[m, p] \vdash before \mapsto res$
assumes $[m, p] \vdash after \mapsto res'$
shows $res = res'$
using *assms*
proof (*induct rule*: *CanonicalizeNegate.induct*)
case (*negate-negate nx x*)
thus *?case*
by (*metis UnaryExprE evalDet stamp-implies-valid-value is-IntegerStamp-def negate-negate-helper unary-eval.simps*(*2*))
next
case (*negate-sub e x y stampx stampy*)
thus *?case*
by (*smt* (*verit*) *BinaryExprE Stamp.sel*(*1*) *UnaryExprE bin-eval.simps*(*3*) *evalDet stamp-implies-valid-value*
is-IntegerStamp-def negate-xsuby-helper unary-eval.simps(*2*))
qed

lemma *word-helper*:
shows $\bigwedge x :: 32\ word.\ \neg(- x <s\ 0 \wedge x <s\ 0)$
and $\bigwedge x :: 64\ word.\ \neg(- x <s\ 0 \wedge x <s\ 0)$
and $\bigwedge x :: 32\ word.\ \neg - x <s\ 0 \wedge \neg\ x <s\ 0 \Longrightarrow 2 * x = 0$
and $\bigwedge x :: 64\ word.\ \neg - x <s\ 0 \wedge \neg\ x <s\ 0 \Longrightarrow 2 * x = 0$
apply (*case-tac*[!] *x*)
apply *auto+*
sorry

lemma *abs-abs-is-abs*:
assumes *valid-value* (*IntegerStamp b lox hix*) *x*
shows *intval-abs* (*intval-abs x*) = *intval-abs x*
using *word-helper*
by (*metis assms intval-abs.simps*(*1*) *intval-abs.simps*(*2*) *valid32or64-both*)

lemma *abs-neg-is-neg*:
assumes *valid-value* (*IntegerStamp b lox hix*) *x*
shows *intval-abs* (*intval-negate x*) = *intval-abs x*
apply (*case-tac*[!] *x*)

```
  using word-helper apply auto+
  done

lemma not-rewrite-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  shows intval-not (intval-not x) = x
  using valid32or64 assms by fastforce+

lemma CanonicalizeNotProof:
  assumes CanonicalizeNot before after
  assumes [m, p] ⊢ before ↦ res
  assumes [m, p] ⊢ after ↦ res'
  shows res = res'
  using assms
proof (induct rule: CanonicalizeNot.induct)
  case (not-not nx x)
  then show ?case
    by (metis UnaryExprE evalDet is-IntegerStamp-def not-rewrite-helper
        stamp-implies-valid-value unary-eval.simps(3))
qed

lemma demorgans-rewrites-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  and     valid-value (IntegerStamp b loy hiy) y

  shows intval-and (intval-not x) (intval-not y) = intval-not (intval-or x y)
  and   intval-or (intval-not x) (intval-not y) = intval-not (intval-and x y)
  and   x = y ⟹ intval-and x y = x
  and   x = y ⟹ intval-or x y = x
  using valid32or64-both assms by fastforce+

lemma CanonicalizeAndProof:
  assumes CanonicalizeAnd before after
  assumes [m, p] ⊢ before ↦ res
  assumes [m, p] ⊢ after ↦ res'
  shows res = res'
  using assms
proof (induct rule: CanonicalizeAnd.induct)
  case (and-same x)
  then show ?case
   by (metis BinaryExprE bin-eval.simps(4) demorgans-rewrites-helper(3) evalDet
        stamp-implies-valid-value is-IntegerStamp-def)
next
  case (and-demorgans nx x ny y stampx stampy)
  then show ?case
   by (smt (z3) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(4) bin-eval.simps(5)
```

demorgans-rewrites-helper(*1*) *evalDet stamp-implies-valid-value is-IntegerStamp-def*
unary-eval.simps(*3*))
qed

lemma *CanonicalizeOrProof*:
assumes *CanonicalizeOr before after*
assumes $[m, p] \vdash$ *before* $\mapsto$ *res*
assumes $[m, p] \vdash$ *after* $\mapsto$ *res'*
shows *res* = *res'*
using *assms*
proof (*induct rule*: *CanonicalizeOr.induct*)
case (*or-same x*)
then show *?case*
by (*metis BinaryExprE bin-eval.simps*(*5*) *demorgans-rewrites-helper*(*4*) *evalDet*
stamp-implies-valid-value is-IntegerStamp-def)
next
case (*or-demorgans nx x ny y stampx stampy*)
then show *?case*
by (*smt* (*z3*) *BinaryExprE Stamp.sel*(*1*) *UnaryExprE bin-eval.simps*(*4*) *bin-eval.simps*(*5*)
demorgans-rewrites-helper(*2*)
evalDet stamp-implies-valid-value is-IntegerStamp-def unary-eval.simps(*3*))
qed

lemma *stamps-touch-but-not-less-than-implies-equal*:
⟦*valid-value stampx x*;
valid-value stampy y;
is-IntegerStamp stampx ∧ *is-IntegerStamp stampy*;
stpi-upper stampx = *stpi-lower stampy*;
¬ *val-to-bool* (*intval-less-than x y*)⟧ ⟹ *x* = *y*
using *valid32or64-both intval-equals.simps*(*1*−*2*) *intval-less-than.simps*(*1*−*2*) *val-to-bool.simps*(*1*)
sorry

lemma *disjoint-stamp-implies-less-than*:
⟦*valid-value stampx x*;
valid-value stampy y;
is-IntegerStamp stampx ∧ *is-IntegerStamp stampy*;
stpi-upper stampx < *stpi-lower stampy*⟧
⟹ *val-to-bool*(*intval-less-than x y*)
sorry

lemma *CanonicalizeConditionalProof*:
assumes *CanonicalizeConditional before after*
assumes $[m, p] \vdash$ *before* $\mapsto$ *res*
assumes $[m, p] \vdash$ *after* $\mapsto$ *res'*
shows *res* = *res'*
using *assms*
proof (*induct rule*: *CanonicalizeConditional.induct*)
case (*eq-branches t f c*)

then show *?case* **using** *evalDet* **by** *auto*
next
case (*cond-eq c x y stampx stampy*)
obtain *xval* **where** *xeval*: $[m,p] \vdash x \mapsto xval$
using *cond-eq.hyps*(*1*) *cond-eq.prems*(*1*) **by** *blast*
obtain *yval* **where** *yeval*: $[m,p] \vdash y \mapsto yval$
using *cond-eq.prems*(*2*) **by** *auto*
show *?case* **proof** (*cases xval = yval*)
case *True*
then show *?thesis*
by (*smt* (*verit*, *ccfv-threshold*) *ConditionalExprE cond-eq.prems*(*1*) *cond-eq.prems*(*2*) *evalDet xeval yeval*)
next
case *False*
then have ¬(*val-to-bool*(*intval-equals xval yval*))
using *ConstantExpr Value.distinct*(*9*) *valid-value.simps stamp-implies-valid-value*
apply (*cases intval-equals xval yval*)
using *IRTreeEval.val-to-bool.simps*(*2*) **apply** *presburger* **sorry**
then have *res = yval*
by (*smt* (*verit*, *ccfv-threshold*) *BinaryExprE ConditionalExprE bin-eval.simps*(*10*) *cond-eq.hyps*(*1*) *cond-eq.prems*(*1*) *evalDet xeval yeval*)
then show *?thesis*
using *cond-eq.prems*(*1*) *cond-eq.prems*(*2*) *xeval yeval evalDet* **by** *auto*
qed
next
case (*condition-bounds-x c x y stampx stampy*)
obtain *xval* **where** *xeval*: $[m,p] \vdash x \mapsto xval$
using *condition-bounds-x.prems*(*2*) **by** *auto*
obtain *yval* **where** *yeval*: $[m,p] \vdash y \mapsto yval$
using *condition-bounds-x.hyps*(*1*) *condition-bounds-x.prems*(*1*) **by** *blast*
then show *?case* **proof** (*cases val-to-bool*(*intval-less-than xval yval*))
case *True*
then show *?thesis*
by (*smt* (*verit*, *best*) *BinaryExprE ConditionalExprE bin-eval.simps*(*11*) *condition-bounds-x.hyps*(*1*) *condition-bounds-x.prems*(*1*) *condition-bounds-x.prems*(*2*) *evalDet xeval yeval*)
next
case *False*
then have *stpi-upper stampx = stpi-lower stampy*
by (*metis False condition-bounds-x.hyps*(*4*) *order.not-eq-order-implies-strict disjoint-stamp-implies-less-than condition-bounds-x.hyps*(*2*) *condition-bounds-x.hyps*(*3*) *condition-bounds-x.hyps*(*6*)
stamp-implies-valid-value xeval yeval)
then have (*xval = yval*)
by (*metis False condition-bounds-x.hyps*(*2*−*3*,*6*) *stamp-implies-valid-value stamps-touch-but-not-less-than-implies-equal xeval yeval*)
then have $res = xval \wedge res' = xval$
using *ConditionalExprE condition-bounds-x.prems*(*1*) ‹$[m,p] \vdash x \mapsto res'$› *evalDet xeval yeval*

by *force*
then show *?thesis* **by** *simp*
qed
next
case (*condition-bounds-y c x y stampx stampy*)
obtain *xval* **where** *xeval*: [*m*,*p*] ⊢ *x* ↦ *xval*
using *condition-bounds-y.hyps*(*1*) *condition-bounds-y.prems*(*1*) **by** *auto*
obtain *yval* **where** *yeval*: [*m*,*p*] ⊢ *y* ↦ *yval*
using *condition-bounds-y.hyps*(*1*) *condition-bounds-y.prems*(*1*) **by** *blast*
then show *?case* **proof** (*cases val-to-bool*(*intval-less-than xval yval*))
case *True*
then show *?thesis*
by (*smt* (*verit*, *best*) *BinaryExprE ConditionalExprE bin-eval.simps*(*11*) *condition-bounds-y.hyps*(*1*) *condition-bounds-y.prems*(*1*) *condition-bounds-y.prems*(*2*) *evalDet xeval yeval*)
next
case *False*
have *stpi-upper stampx* = *stpi-lower stampy*
by (*metis False condition-bounds-y.hyps*(*4*) *order.not-eq-order-implies-strict disjoint-stamp-implies-less-than condition-bounds-y.hyps*(*2*) *condition-bounds-y.hyps*(*3*) *condition-bounds-y.hyps*(*6*) *stamp-implies-valid-value xeval yeval*)
then have (*xval* = *yval*)
by (*metis False condition-bounds-y.hyps*(*2*−*3*,*6*) *stamp-implies-valid-value stamps-touch-but-not-less-than-implies-equal xeval yeval*)
then have *res* = *yval* ∧ *res′* = *yval*
using *ConditionalExprE condition-bounds-y.prems*(*1*) ‹[*m*,*p*] ⊢ *y* ↦ *res′*› *evalDet xeval yeval*
by *force*
then show *?thesis* **by** *simp*
qed
next
case (*negate-condition nc c stampc lo hi stampx x stampy y*)
obtain *cval* **where** *ceval*: [*m*,*p*] ⊢ *c* ↦ *cval*
using *negate-condition.prems*(*2*) **by** *auto*
obtain *ncval* **where** *nceval*: [*m*,*p*] ⊢ *nc* ↦ *ncval*
using *negate-condition.prems negate-condition.prems* **by** *blast*
then show *?case* **using** *assms* **proof** (*cases* (*val-to-bool ncval*))
case *True*
obtain *xval* **where** *xeval*: [*m*,*p*] ⊢ *x* ↦ *xval*
by (*metis* (*full-types*) *ConditionalExprE nceval evalDet True negate-condition.prems*(*1*))
then have *res* = *xval*
by (*metis* (*full-types*) *ConditionalExprE True evalDet nceval negate-condition.prems*(*1*))
have *c* ≠ *nc*
by (*simp add*: *negate-condition.hyps*(*1*))
then have ¬(*val-to-bool cval*)
by (*metis IRTreeEval.val-to-bool.elims*(*2*) *IRTreeEval.val-to-bool.simps*(*1*) *True UnaryExprE ceval evalDet nceval negate-condition.hyps*(*1*) *unary-eval.simps*(*4*))
then have *res′* = *xval*

using *nceval ceval True negate-condition*(*1*) *negate-condition*(*9*)
by (*metis* (*full-types*) *ConditionalExprE evalDet xeval*)
then show *?thesis*
by (*simp add*: ‹*res* = *xval*›)
next
case *False*
obtain *yval* where *yeval*: [*m*,*p*] ⊢ *y* ↦ *yval*
by (*metis* (*full-types*) *ConditionalExprE nceval evalDet False negate-condition.prems*(*1*))
then have *res* = *yval*
using *False nceval negate-condition.prems*(*1*) *evaltree.ConditionalExpr yeval evalDet*
by (*metis* (*full-types*) *ConditionalExprE*)
moreover have *val-to-bool*(*cval*)
by (*metis False UnaryExprE ceval nceval negate-condition.hyps*(*1*−*3*) *unary-eval.simps*(*4*) *IRTreeEval.val-to-bool.simps*(*1*) *evalDet IRTreeEval.bool-to-val.simps*(*2*) *stamp-implies-valid-value valid-int32 zero-neq-one*)
moreover have res' = *yval*
using *calculation*(*2*) *ceval negate-condition.prems evaltree.ConditionalExpr yeval evalDet unary-eval.simps*(*4*)
by (*metis* (*full-types*) *ConditionalExprE*)
ultimately show *?thesis* by *simp*
qed
next
case (*const-true c val t f*)
then show *?case* using *evalDet* by *auto*
next
case (*const-false c val t f*)
then show *?case* using *evalDet* by *auto*
qed

end