Veriopt Theories

January 6, 2023

Contents

1   Conditional Elimination Phase

theory *ConditionalElimination*
  imports
    Semantics.IRTreeEvalThms
    Proofs.Rewrites
    Proofs.Bisimulation
begin

1.1   Individual Elimination Rules

The set of rules used for determining whether a condition $q1::'a$ implies another condition $q2::'a$ or its negation. These rules are used for conditional elimination.

inductive *impliesx* :: *IRExpr* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* (- $\Rightarrow$ -) **and**
    impliesnot :: *IRExpr* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* (- $\Rightarrow\neg$ -) **where**
  q-imp-q:
  $q \Rightarrow q$ |
  eq-impliesnot-less:
  (*BinaryExpr BinIntegerEquals x y*) $\Rightarrow\neg$ (*BinaryExpr BinIntegerLessThan x y*) |
  eq-impliesnot-less-rev:
  (*BinaryExpr BinIntegerEquals x y*) $\Rightarrow\neg$ (*BinaryExpr BinIntegerLessThan y x*) |
  less-impliesnot-rev-less:
  (*BinaryExpr BinIntegerLessThan x y*) $\Rightarrow\neg$ (*BinaryExpr BinIntegerLessThan y x*)
|
  less-impliesnot-eq:
  (*BinaryExpr BinIntegerLessThan x y*) $\Rightarrow\neg$ (*BinaryExpr BinIntegerEquals x y*) |
  less-impliesnot-eq-rev:
  (*BinaryExpr BinIntegerLessThan x y*) $\Rightarrow\neg$ (*BinaryExpr BinIntegerEquals y x*) |
  negate-true:

$[\![x \Rightarrow\neg\ y]\!] \Longrightarrow x \Rightarrow (\textit{UnaryExpr UnaryLogicNegation y})\ |$
negate-false:
$[\![x \Rightarrow y]\!] \Longrightarrow x \Rightarrow\neg\ (\textit{UnaryExpr UnaryLogicNegation y})$

The relation *q1*::*IRExpr* $\Rightarrow$ *q2*::*IRExpr* indicates that the implication (*q1*::*bool*) $\longrightarrow$ (*q2*::*bool*) is known true (i.e. universally valid), and the relation *q1*::*IRExpr* $\Rightarrow\neg$ *q2*::*IRExpr* indicates that the implication (*q1*::*bool*) $\longrightarrow$ (*q2*::*bool*) is known false (i.e. (*q1*::*bool*) $\longrightarrow \neg$ (*q2*::*bool*) is universally valid. If neither *q1*::*IRExpr* $\Rightarrow$ *q2*::*IRExpr* nor *q1*::*IRExpr* $\Rightarrow\neg$ *q2*::*IRExpr* then the status is unknown. Only the known true and known false cases can be used for conditional elimination.

fun *implies-valid* :: *IRExpr* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* (**infix** $\rightarrowtail$ *50*) **where**
  implies-valid q1 q2 =
    ($\forall$ *m p v1 v2*. ([*m, p*] $\vdash$ *q1* $\mapsto$ *v1*) $\wedge$ ([*m,p*] $\vdash$ *q2* $\mapsto$ *v2*) $\longrightarrow$
      (*val-to-bool v1* $\longrightarrow$ *val-to-bool v2*))

fun *impliesnot-valid* :: *IRExpr* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* (**infix** $\rightarrowtail$ *50*) **where**
  impliesnot-valid q1 q2 =
    ($\forall$ *m p v1 v2*. ([*m, p*] $\vdash$ *q1* $\mapsto$ *v1*) $\wedge$ ([*m,p*] $\vdash$ *q2* $\mapsto$ *v2*) $\longrightarrow$
      (*val-to-bool v1* $\longrightarrow \neg$*val-to-bool v2*))

The relation (*q1*::*IRExpr*) $\rightarrowtail$ (*q2*::*IRExpr*) means (*q1*::*bool*) $\longrightarrow$ (*q2*::*bool*) is universally valid, and the relation (*q1*::*IRExpr*) $\rightarrowtail$ (*q2*::*IRExpr*) means (*q1*::*bool*) $\longrightarrow \neg$ (*q2*::*bool*) is universally valid.

lemma *eq-impliesnot-less-helper*:
  v1 = *v2* $\longrightarrow \neg$(*int-signed-value b v1* $<$ *int-signed-value b v2*)
  by *force*

lemma *eq-impliesnot-less-val*:
  val-to-bool(*intval-equals v1 v2*) $\longrightarrow \neg$*val-to-bool*(*intval-less-than v1 v2*)
  using *eq-impliesnot-less-helper bool-to-val.simps bool-to-val-bin.simps intval-equals.simps*
    intval-less-than.elims val-to-bool.elims val-to-bool.simps
  by (*smt* (*verit*))

lemma *eq-impliesnot-less-rev-val*:
  val-to-bool(*intval-equals v1 v2*) $\longrightarrow \neg$*val-to-bool*(*intval-less-than v2 v1*)
proof $-$
  have *a*: *intval-equals v1 v2* = *intval-equals v2 v1*
    using *bool-to-val-bin.simps intval-equals.simps intval-equals.elims*
    by (*smt* (*verit*))
  show *?thesis* **using** *a eq-impliesnot-less-val* **by** *presburger*
qed

lemma *less-impliesnot-rev-less-val*:
  val-to-bool(*intval-less-than v1 v2*) $\longrightarrow \neg$*val-to-bool*(*intval-less-than v2 v1*)
  by (*smt* (*verit, del-insts*) *Value.exhaust Value.inject*(*1*) *bool-to-val.simps*(*2*)
    bool-to-val-bin.simps intval-less-than.simps(*1*) *intval-less-than.simps*(*5*)
    intval-less-than.simps(*6*) *intval-less-than.simps*(*7*) *val-to-bool.elims*(*2*))

lemma *less-impliesnot-eq-val*:
  val-to-bool(*intval-less-than v1 v2*) $\longrightarrow \neg$*val-to-bool*(*intval-equals v1 v2*)
  using *eq-impliesnot-less-val* **by** *blast*

lemma *logic-negate-type*:
  assumes [*m*, *p*] $\vdash$ *UnaryExpr UnaryLogicNegation x* $\mapsto$ *v*
  shows $\exists$ *b v2*. [*m*, *p*] $\vdash$ *x* $\mapsto$ *IntVal b v2*
  using *assms*
  by (*metis UnaryExprE intval-logic-negation.elims unary-eval.simps(4)*)

lemma *intval-logic-negation-inverse*:
  assumes *b* > *0*
  assumes *x* = *IntVal b v*
  shows *val-to-bool* (*intval-logic-negation x*) $\longleftrightarrow \neg$(*val-to-bool x*)
  using *assms* **by** (*cases x*; *auto simp*: *logic-negate-def*)

lemma *logic-negation-relation-tree*:
  assumes [*m*, *p*] $\vdash$ *y* $\mapsto$ *val*
  assumes [*m*, *p*] $\vdash$ *UnaryExpr UnaryLogicNegation y* $\mapsto$ *invval*
  shows *val-to-bool val* $\longleftrightarrow \neg$(*val-to-bool invval*)
  using *assms* **using** *intval-logic-negation-inverse*
 by (*metis UnaryExprE evalDet eval-bits-1-64 logic-negate-type unary-eval.simps(4)*)

The following theorem shows that the known true/false rules are valid.

theorem *implies-impliesnot-valid*:
  shows ((*q1* $\Rrightarrow$ *q2*) $\longrightarrow$ (*q1* $\rightarrowtail$ *q2*)) $\wedge$
        ((*q1* $\Rrightarrow\neg$ *q2*) $\longrightarrow$ (*q1* $\rightarrowtail\!\!\rightarrow$ *q2*))
        (**is** (*?imp* $\longrightarrow$ *?val*) $\wedge$ (*?notimp* $\longrightarrow$ *?notval*))
proof (*induct q1 q2 rule*: *impliesx-impliesnot.induct*)
  case (*q-imp-q q*)
  then show *?case*
    using *evalDet* **by** *fastforce*
next
  case (*eq-impliesnot-less x y*)
  then show *?case* **apply** *auto* **using** *eq-impliesnot-less-val evalDet* **by** *blast*
next
  case (*eq-impliesnot-less-rev x y*)
  then show *?case* **apply** *auto* **using** *eq-impliesnot-less-rev-val evalDet* **by** *blast*
next
  case (*less-impliesnot-rev-less x y*)
  then show *?case* **apply** *auto* **using** *less-impliesnot-rev-less-val evalDet* **by** *blast*
next
  case (*less-impliesnot-eq x y*)
  then show *?case* **apply** *auto* **using** *less-impliesnot-eq-val evalDet* **by** *blast*
next
  case (*less-impliesnot-eq-rev x y*)
  then show *?case* **apply** *auto* **using** *eq-impliesnot-less-rev-val evalDet* **by** *metis*
next

case (*negate-true x y*)
then show *?case* **apply** *auto*
  by (*metis logic-negation-relation-tree unary-eval.simps(4) unfold-unary*)
next
case (*negate-false x y*)
then show *?case* **apply** *auto*
  by (*metis UnaryExpr logic-negation-relation-tree unary-eval.simps(4)*)
qed

We introduce a type *TriState*::$'a$ (as in the GraalVM compiler) to represent when static analysis can tell us information about the value of a Boolean expression. If *Unknown*::$'a$ then no information can be inferred and if *Known-True*::$'a$/*KnownFalse*::$'a$ one can infer the expression is always true/false.

datatype *TriState = Unknown | KnownTrue | KnownFalse*

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

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

x-imp-x:
g ⊢ x & x ↪ KnownTrue |

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

Total relation over partial implies relation

inductive *condition-implies* :: *IRGraph ⇒ IRNode ⇒ IRNode ⇒ TriState ⇒ bool*
  (*- ⊢ - & - ⇀ -*) **for** *g* **where**
⟦¬(*g ⊢ a & b ↪ imp*)⟧ ⟹ (*g ⊢ a & b ⇀ Unknown*) |
⟦(*g ⊢ a & b ↪ imp*)⟧ ⟹ (*g ⊢ a & b ⇀ imp*)

4

inductive *implies-tree* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* ⇒ *bool*
  (- & - ↪ -) **where**
  eq-imp-less:
  (*BinaryExpr BinIntegerEquals x y*) & (*BinaryExpr BinIntegerLessThan x y*) ↪
False |
  eq-imp-less-rev:
  (*BinaryExpr BinIntegerEquals x y*) & (*BinaryExpr BinIntegerLessThan y x*) ↪
False |
  less-imp-rev-less:
  (*BinaryExpr BinIntegerLessThan x y*) & (*BinaryExpr BinIntegerLessThan y x*)
↪ *False* |
  less-imp-not-eq:
  (*BinaryExpr BinIntegerLessThan x y*) & (*BinaryExpr BinIntegerEquals x y*) ↪
False |
  less-imp-not-eq-rev:
  (*BinaryExpr BinIntegerLessThan x y*) & (*BinaryExpr BinIntegerEquals y x*) ↪
False |
  x-imp-x:
  x & *x* ↪ *True* |
  negate-false:
  ⟦*x* & *y* ↪ *True*⟧ ⟹ *x* & (*UnaryExpr UnaryLogicNegation y*) ↪ *False* |
  negate-true:
  ⟦*x* & *y* ↪ *False*⟧ ⟹ *x* & (*UnaryExpr UnaryLogicNegation y*) ↪ *True*

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

lemma *logic-negation-relation*:
  assumes [*g*, *m*, *p*] ⊢ *y* ↦ *val*
  assumes *kind g neg = LogicNegationNode y*
  assumes [*g*, *m*, *p*] ⊢ *neg* ↦ *invval*
  assumes *invval* ≠ *UndefVal*
  shows *val-to-bool val* ⟷ ¬(*val-to-bool invval*)
  using *assms*
  by (*metis LogicNegationNode encodeeval-def logic-negation-relation-tree repDet*)

lemma *implies-valid*:
  assumes *x* & *y* ↪ *imp*
  assumes [*m*, *p*] ⊢ *x* ↦ *v1*
  assumes [*m*, *p*] ⊢ *y* ↦ *v2*
  shows (*imp* ⟶ (*val-to-bool v1* ⟶ *val-to-bool v2*)) ∧
      (¬*imp* ⟶ (*val-to-bool v1* ⟶ ¬(*val-to-bool v2*)))
   (**is** (*?TP* ⟶ *?TC*) ∧ (*?FP* ⟶ *?FC*))
  apply (*intro conjI*; *rule impI*)
proof −
  assume *KnownTrue*: *?TP*
  show *?TC*
  using *assms*(*1*) *KnownTrue assms*(*2*−) **proof** (*induct x y imp rule*: *implies-tree.induct*)
    case (*eq-imp-less x y*)

5

then show *?case* **by** *simp*
 next
   case (*eq-imp-less-rev x y*)
   then show *?case* **by** *simp*
 next
   case (*less-imp-rev-less x y*)
   then show *?case* **by** *simp*
 next
   case (*less-imp-not-eq x y*)
   then show *?case* **by** *simp*
 next
   case (*less-imp-not-eq-rev x y*)
   then show *?case* **by** *simp*
 next
   case (*x-imp-x*)
   then show *?case*
     by (*metis evalDet*)
 next
   case (*negate-false x1*)
   then show *?case* **using** *evalDet*
     using *assms(2,3)* **by** *blast*
 next
   case (*negate-true x y*)
   then show *?case*
     using *logic-negation-relation-tree* **sorry**
 qed
next
 assume *KnownFalse*: *?FP*
 show *?FC* **using** *assms KnownFalse* **proof** (*induct x y imp rule*: *implies-tree.induct*)
   case (*eq-imp-less x y*)
   obtain *xval* **where** *xval*: [*m, p*] ⊢ *x* ↦ *xval*
     using *eq-imp-less(1) eq-imp-less.prems(3)*
     by *blast*
   then obtain *yval* **where** *yval*: [*m, p*] ⊢ *y* ↦ *yval*
     using *eq-imp-less.prems(3)*
     using *eq-imp-less.prems(2)* **by** *blast*
   have *eqeval*: [*m, p*] ⊢ (*BinaryExpr BinIntegerEquals x y*) ↦ *intval-equals xval yval*
     using *xval yval evaltree.BinaryExpr*
     by (*metis BinaryExprE bin-eval.simps(11) eq-imp-less.prems(1) evalDet*)
   have *lesseval*: [*m, p*] ⊢ (*BinaryExpr BinIntegerLessThan x y*) ↦ *intval-less-than xval yval*
     using *xval yval evaltree.BinaryExpr*
     by (*metis BinaryExprE bin-eval.simps(12) eq-imp-less.prems(2) evalDet*)
   have *val-to-bool* (*intval-equals xval yval*) ⟶ ¬(*val-to-bool* (*intval-less-than xval yval*))
     apply (*cases xval*; *cases yval*; *auto*)
     by (*smt* (*verit, best*) *bool-to-val.simps(2) val-to-bool.simps(1)*)
   then show *?case*

6

       using *eqeval lesseval*

       by (*metis eq-imp-less.prems(1) eq-imp-less.prems(2) evalDet*)

    next

     case (*eq-imp-less-rev x y*)

     obtain *xval* **where** *xval*: $[m, p] \vdash x \mapsto xval$

      using *eq-imp-less-rev.prems(3)*

      using *eq-imp-less-rev.prems(2)* **by** *blast*

     obtain *yval* **where** *yval*: $[m, p] \vdash y \mapsto yval$

      using *eq-imp-less-rev.prems(3)*

      using *eq-imp-less-rev.prems(2)* **by** *blast*

      have *eqeval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerEquals x y*) $\mapsto$ *intval-equals xval yval*

      using *xval yval evaltree.BinaryExpr*

      by (*metis BinaryExprE bin-eval.simps(11) eq-imp-less-rev.prems(1) evalDet*)

      have *lesseval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerLessThan y x*) $\mapsto$ *intval-less-than yval xval*

      using *xval yval evaltree.BinaryExpr*

      by (*metis BinaryExprE bin-eval.simps(12) eq-imp-less-rev.prems(2) evalDet*)

      have *val-to-bool* (*intval-equals xval yval*) $\longrightarrow \neg$(*val-to-bool* (*intval-less-than yval xval*))

      apply (*cases xval*; *cases yval*; *auto*)

      by (*metis* (*full-types*) *bool-to-val.simps(2) less-irrefl val-to-bool.simps(1)*)

     then show *?case*

      using *eqeval lesseval*

      by (*metis eq-imp-less-rev.prems(1) eq-imp-less-rev.prems(2) evalDet*)

    next

     case (*less-imp-rev-less x y*)

     obtain *xval* **where** *xval*: $[m, p] \vdash x \mapsto xval$

      using *less-imp-rev-less.prems(3)*

      using *less-imp-rev-less.prems(2)* **by** *blast*

     obtain *yval* **where** *yval*: $[m, p] \vdash y \mapsto yval$

      using *less-imp-rev-less.prems(3)*

      using *less-imp-rev-less.prems(2)* **by** *blast*

      have *lesseval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerLessThan x y*) $\mapsto$ *intval-less-than xval yval*

      using *xval yval evaltree.BinaryExpr*

      by (*metis BinaryExprE bin-eval.simps(12) evalDet less-imp-rev-less.prems(1)*)

      have *revlesseval*: $[m, p] \vdash$ (*BinaryExpr BinIntegerLessThan y x*) $\mapsto$ *intval-less-than yval xval*

      using *xval yval evaltree.BinaryExpr*

      by (*metis BinaryExprE bin-eval.simps(12) evalDet less-imp-rev-less.prems(2)*)

      have *val-to-bool* (*intval-less-than xval yval*) $\longrightarrow \neg$(*val-to-bool* (*intval-less-than yval xval*))

      apply (*cases xval*; *cases yval*; *auto*)

      by (*smt* (*verit*) *bool-to-val.simps(2) val-to-bool.simps(1)*)

     then show *?case*

      by (*metis evalDet less-imp-rev-less.prems(1) less-imp-rev-less.prems(2) lesseval revlesseval*)

    next

case (*less-imp-not-eq x y*)
obtain *xval* **where** *xval*: [*m*, *p*] ⊢ *x* ↦ *xval*
  using *less-imp-not-eq.prems(3)*
  using *less-imp-not-eq.prems(1)* **by** *blast*
obtain *yval* **where** *yval*: [*m*, *p*] ⊢ *y* ↦ *yval*
  using *less-imp-not-eq.prems(3)*
  using *less-imp-not-eq.prems(1)* **by** *blast*
have *eqeval*: [*m*, *p*] ⊢ (*BinaryExpr BinIntegerEquals x y*) ↦ *intval-equals xval yval*
  using *xval yval evaltree.BinaryExpr*
  by (*metis BinaryExprE bin-eval.simps(11) evalDet less-imp-not-eq.prems(2)*)
have *lesseval*: [*m*, *p*] ⊢ (*BinaryExpr BinIntegerLessThan x y*) ↦ *intval-less-than xval yval*
  using *xval yval evaltree.BinaryExpr*
  by (*metis BinaryExprE bin-eval.simps(12) evalDet less-imp-not-eq.prems(1)*)
have *val-to-bool* (*intval-less-than xval yval*) ⟶ ¬(*val-to-bool* (*intval-equals xval yval*))
  apply (*cases xval*; *cases yval*; *auto*)
  by (*smt* (*verit, best*) *bool-to-val.simps(2) val-to-bool.simps(1)*)
then show *?case*
  by (*metis eqeval evalDet less-imp-not-eq.prems(1) less-imp-not-eq.prems(2) lesseval*)
next
  case (*less-imp-not-eq-rev x y*)
  obtain *xval* **where** *xval*: [*m*, *p*] ⊢ *x* ↦ *xval*
    using *less-imp-not-eq-rev.prems(3)*
    using *less-imp-not-eq-rev.prems(1)* **by** *blast*
  obtain *yval* **where** *yval*: [*m*, *p*] ⊢ *y* ↦ *yval*
    using *less-imp-not-eq-rev.prems(3)*
    using *less-imp-not-eq-rev.prems(1)* **by** *blast*
  have *eqeval*: [*m*, *p*] ⊢ (*BinaryExpr BinIntegerEquals y x*) ↦ *intval-equals yval xval*
    using *xval yval evaltree.BinaryExpr*
    by (*metis BinaryExprE bin-eval.simps(11) evalDet less-imp-not-eq-rev.prems(2)*)
  have *lesseval*: [*m*, *p*] ⊢ (*BinaryExpr BinIntegerLessThan x y*) ↦ *intval-less-than xval yval*
    using *xval yval evaltree.BinaryExpr*
    by (*metis BinaryExprE bin-eval.simps(12) evalDet less-imp-not-eq-rev.prems(1)*)
  have *val-to-bool* (*intval-less-than xval yval*) ⟶ ¬(*val-to-bool* (*intval-equals yval xval*))
    apply (*cases xval*; *cases yval*; *auto*)
    by (*smt* (*verit, best*) *bool-to-val.simps(2) val-to-bool.simps(1)*)
  then show *?case*
  by (*metis eqeval evalDet less-imp-not-eq-rev.prems(1) less-imp-not-eq-rev.prems(2) lesseval*)
next
  case (*x-imp-x x1*)
  then show *?case* **by** *simp*
next

    case (*negate-false x y*)
    then show *?case* **sorry**
  next
    case (*negate-true x1*)
    then show *?case* **by** *simp*
  qed
qed

lemma *implies-true-valid*:
  assumes *x & y ↪ imp*
  assumes *imp*
  assumes *[m, p] ⊢ x ↦ v1*
  assumes *[m, p] ⊢ y ↦ v2*
  shows *val-to-bool v1 ⟶ val-to-bool v2*
  using *assms implies-valid*
  by *blast*

lemma *implies-false-valid*:
  assumes *x & y ↪ imp*
  assumes *¬imp*
  assumes *[m, p] ⊢ x ↦ v1*
  assumes *[m, p] ⊢ y ↦ v2*
  shows *val-to-bool v1 ⟶ ¬(val-to-bool v2)*
  using *assms implies-valid* **by** *blast*

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

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

inductive *tryFold :: IRNode ⇒ (ID ⇒ Stamp) ⇒ bool ⇒ bool*
  where
  ⟦*alwaysDistinct* (*stamps x*) (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerEqualsNode x y*) *stamps False* |
  ⟦*neverDistinct* (*stamps x*) (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerEqualsNode x y*) *stamps True* |
  ⟦*is-IntegerStamp* (*stamps x*);
   is-IntegerStamp (*stamps y*);
   stpi-upper (*stamps x*) < *stpi-lower* (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerLessThanNode x y*) *stamps True* |
  ⟦*is-IntegerStamp* (*stamps x*);
   is-IntegerStamp (*stamps y*);
   stpi-lower (*stamps x*) ≥ *stpi-upper* (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerLessThanNode x y*) *stamps False*

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

lemma
  assumes *kind g nid = IntegerEqualsNode x y*
  assumes *[g, m, p] ⊢ nid ↦ v*
  assumes *([g, m, p] ⊢ x ↦ xval) ∧ ([g, m, p] ⊢ y ↦ yval)*
  shows *val-to-bool (intval-equals xval yval) ⟷ v = IntVal 32 1*
proof −
  have *v = intval-equals xval yval*
    using *assms(1, 2, 3) BinaryExprE IntegerEqualsNode bin-eval.simps(7)*
    by *(smt (verit) bin-eval.simps(11) encodeeval-def evalDet repDet)*
  then show *?thesis* **using** *intval-equals.simps val-to-bool.simps*
    by *(smt (verit) bool-to-val.simps(1) bool-to-val.simps(2) bool-to-val-bin.simps*
      intval-equals.elims one-neq-zero)
qed

lemma *tryFoldIntegerEqualsAlwaysDistinct*:
  assumes *wf-stamp g stamps*
  assumes *kind g nid = (IntegerEqualsNode x y)*
  assumes *[g, m, p] ⊢ nid ↦ v*
  assumes *alwaysDistinct (stamps x) (stamps y)*
  shows *v = IntVal 32 0*
proof −
  have *∀ val. ¬(valid-value val (join (stamps x) (stamps y)))*
    using *assms(1,4)* **unfolding** *alwaysDistinct.simps*
    by *(smt (verit, best) is-stamp-empty.elims(2) valid-int valid-value.simps(1))*
  obtain *xv* **where** *[g, m, p] ⊢ x ↦ xv*
    using *assms* **using** *assms(2,3)* **unfolding** *encodeeval-def* **sorry**
  have *¬(∃ val . ([g, m, p] ⊢ x ↦ val) ∧ ([g, m, p] ⊢ y ↦ val))*
     using *assms(1,4)* **unfolding** *alwaysDistinct.simps wf-stamp.simps encodee-*
val-def **sorry**
  then show *?thesis* **sorry**
qed

lemma *tryFoldIntegerEqualsNeverDistinct*:
  assumes *wf-stamp g stamps*
  assumes *kind g nid = (IntegerEqualsNode x y)*
  assumes *[g, m, p] ⊢ nid ↦ v*
  assumes *neverDistinct (stamps x) (stamps y)*
  shows *v = IntVal 32 1*
  using *assms IntegerEqualsNodeE* **sorry**

lemma *tryFoldIntegerLessThanTrue*:
  assumes *wf-stamp g stamps*
  assumes *kind g nid = (IntegerLessThanNode x y)*
  assumes *[g, m, p] ⊢ nid ↦ v*
  assumes *stpi-upper (stamps x) < stpi-lower (stamps y)*
  shows *v = IntVal 32 1*
proof −
  have *stamp-type*: *is-IntegerStamp (stamps x)*
    using *assms*

    sorry
  obtain *xval* **where** *xval*: [*g*, *m*, *p*] ⊢ *x* ↦ *xval*
    using *assms(2,3)* **sorry**
  obtain *yval* **where** *yval*: [*g*, *m*, *p*] ⊢ *y* ↦ *yval*
    using *assms(2,3)* **sorry**
  have *is-IntegerStamp* (*stamps x*) ∧ *is-IntegerStamp* (*stamps y*)
    using *assms(4)*
    sorry
  then have *val-to-bool* (*intval-less-than xval yval*)
    sorry
  then show *?thesis*
    sorry
qed

lemma *tryFoldIntegerLessThanFalse*:
  assumes *wf-stamp g stamps*
  assumes *kind g nid* = (*IntegerLessThanNode x y*)
  assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
  assumes *stpi-lower* (*stamps x*) ≥ *stpi-upper* (*stamps y*)
  shows *v* = *IntVal 32 0*
  proof −
  have *stamp-type*: *is-IntegerStamp* (*stamps x*)
    using *assms*
    sorry
  obtain *xval* **where** *xval*: [*g*, *m*, *p*] ⊢ *x* ↦ *xval*
    using *assms(2,3)* **sorry**
  obtain *yval* **where** *yval*: [*g*, *m*, *p*] ⊢ *y* ↦ *yval*
    using *assms(2,3)* **sorry**
  have *is-IntegerStamp* (*stamps x*) ∧ *is-IntegerStamp* (*stamps y*)
    using *assms(4)*
    sorry
  then have ¬(*val-to-bool* (*intval-less-than xval yval*))
    sorry
  then show *?thesis*
    sorry
qed

theorem *tryFoldProofTrue*:
  assumes *wf-stamp g stamps*
  assumes *tryFold* (*kind g nid*) *stamps True*
  assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
  shows *val-to-bool v*
  using *assms(2)* **proof** (*induction kind g nid stamps True rule*: *tryFold.induct*)
case (*1 stamps x y*)
  then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms*
    by *force*
next
  case (*2 stamps x y*)
  then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms*

by (*smt* (*verit, best*) *one-neq-zero tryFold.cases tryFoldIntegerEqualsNeverDistinct tryFoldIntegerLessThanTrue val-to-bool.simps*(*1*))
next
   case (*3 stamps x y*)
   then show *?case* **using** *tryFoldIntegerLessThanTrue assms*
      by (*smt* (*verit, best*) *one-neq-zero tryFold.cases tryFoldIntegerEqualsNeverDistinct val-to-bool.simps*(*1*))
next
case (*4 stamps x y*)
   then show *?case* **using** *tryFoldIntegerLessThanFalse assms* **sorry**
qed

theorem *tryFoldProofFalse*:
   assumes *wf-stamp g stamps*
   assumes *tryFold* (*kind g nid*) *stamps False*
   assumes [*g, m, p*] ⊢ *nid* ↦ *v*
   shows ¬(*val-to-bool v*)
using *assms*(*2*) **proof** (*induction kind g nid stamps False rule*: *tryFold.induct*)
case (*1 stamps x y*)
   then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms* **sorry**
next
case (*2 stamps x y*)
   then show *?case* **using** *tryFoldIntegerEqualsNeverDistinct assms* **sorry**
next
   case (*3 stamps x y*)
   then show *?case* **using** *tryFoldIntegerLessThanTrue assms* **sorry**
next
   case (*4 stamps x y*)
   then show *?case* **using** *tryFoldIntegerLessThanFalse assms* **sorry**

qed

inductive-cases *StepE*:
   g, p ⊢ (*nid,m,h*) → (*nid′,m′,h*)

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

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

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

inductive *ConditionalEliminationStep* ::
   IRExpr set ⇒ (*ID* ⇒ *Stamp*) ⇒ *IRGraph* ⇒ *ID* ⇒ *IRGraph* ⇒ *bool* **where**
   impliesTrue:

$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
  $g \vdash cid \simeq cond;$
  $\exists\ ce \in conds\ .\ (ce \Rightarrow cond);$
  $g' = constantCondition\ True\ ifcond\ (kind\ g\ ifcond)\ g$
  $\rrbracket \Longrightarrow ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'\ |$

  impliesFalse:
$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
  $g \vdash cid \simeq cond;$
  $\exists\ ce \in conds\ .\ (ce \Rightarrow\neg\ cond);$
  $g' = constantCondition\ False\ ifcond\ (kind\ g\ ifcond)\ g$
  $\rrbracket \Longrightarrow ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'\ |$

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

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

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

thm *ConditionalEliminationStep.equation*

1.2 Control-flow Graph Traversal

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

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

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

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

Merge nodes represent a special case where-in the predecessor exists as an

input edge of the merge node, to simplify the traversal we treat only the first input end node as the predecessor, ignoring that multiple nodes may act as a successor.

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

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

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

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

fun *registerNewCondition* :: *IRGraph* $\Rightarrow$ *IRNode* $\Rightarrow$ (*ID* $\Rightarrow$ *Stamp*) $\Rightarrow$ (*ID* $\Rightarrow$ *Stamp*) **where**

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

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

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

The Step relation is a small-step traversal of the graph which handles tran-

sitions between individual nodes of the graph.

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

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

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

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

 i = find-index nid (successors-of (kind g ifcond));
 c = (if i = 0 then kind g cond else LogicNegationNode cond);
 rep g cond ce;
 ce' = (if i = 0 then ce else UnaryExpr UnaryLogicNegation ce);
 conds' = ce' # *conds*;

 flow' = registerNewCondition g c (hdOr flow (stamp g))⟧
 ⟹ *Step g (nid, seen, conds, flow) (Some (nid', seen', conds', flow'* # *flow))* |

 — Hit an EndNode 1. nid' will be the usage of EndNode 2. pop the conditions and stamp stack
 ⟦*kind g nid = EndNode*;

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

 nid' = any-usage g nid;

 conds' = tl conds;
 flow' = tl flow⟧
 ⟹ *Step g (nid, seen, conds, flow) (Some (nid', seen', conds', flow'))* |

 — We can find a successor edge that is not in seen, go there
 ⟦¬(*is-EndNode (kind g nid)*);
  ¬(*is-BeginNode (kind g nid)*);

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

Some nid′ = *nextEdge seen′ nid g*⟧
⟹ *Step g* (*nid*, *seen*, *conds*, *flow*) (*Some* (*nid′*, *seen′*, *conds*, *flow*)) |

— We can cannot find a successor edge that is not in seen, give back None
⟦¬(*is-EndNode* (*kind g nid*));
  ¬(*is-BeginNode* (*kind g nid*));

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

None = *nextEdge seen′ nid g*⟧
⟹ *Step g* (*nid*, *seen*, *conds*, *flow*) *None* |

— We've already seen this node, give back None
⟦*nid* ∈ *seen*⟧ ⟹ *Step g* (*nid*, *seen*, *conds*, *flow*) *None*

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

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

inductive *ConditionalEliminationPhase*
  :: *IRGraph* ⇒ (*ID* × *Seen* × *Conditions* × *StampFlow*) ⇒ *IRGraph* ⇒ *bool*
where

— Can do a step and optimise for the current node
⟦*Step g* (*nid*, *seen*, *conds*, *flow*) (*Some* (*nid′*, *seen′*, *conds′*, *flow′*));
  ConditionalEliminationStep (*set conds*) (*hdOr flow* (*stamp g*)) *g nid g′*;

ConditionalEliminationPhase g′ (*nid′*, *seen′*, *conds′*, *flow′*) *g′′*⟧
⟹ *ConditionalEliminationPhase g* (*nid*, *seen*, *conds*, *flow*) *g′′* |

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

ConditionalEliminationPhase g (*nid′*, *seen′*, *conds′*, *flow′*) *g′*⟧
⟹ *ConditionalEliminationPhase g* (*nid*, *seen*, *conds*, *flow*) *g′* |

— Can't do a step but there is a predecessor we can backtrace to
⟦*Step g* (*nid*, *seen*, *conds*, *flow*) *None*;
  Some nid′ = *pred g nid*;
  seen′ = {*nid*} ∪ *seen*;
  ConditionalEliminationPhase g (*nid′*, *seen′*, *conds*, *flow*) *g′*⟧
⟹ *ConditionalEliminationPhase g* (*nid*, *seen*, *conds*, *flow*) *g′* |

— Can't do a step and have no predecessors so terminate
⟦*Step g* (*nid*, *seen*, *conds*, *flow*) *None*;
    None = *pred g nid*⟧
    ⟹ *ConditionalEliminationPhase g* (*nid*, *seen*, *conds*, *flow*) *g*

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

definition *runConditionalElimination* :: *IRGraph* ⇒ *IRGraph* **where**
  runConditionalElimination g =
    (*Predicate.the* (*ConditionalEliminationPhase-i-i-o g* (*0*, {}, ([], []))))

end