

Veriopt Theories

February 4, 2022

Contents

1	Conditional Elimination Phase	1
1.1	Individual Elimination Rules	1
1.2	Control-flow Graph Traversal	12

1 Conditional Elimination Phase

theory *ConditionalElimination*

imports

Proofs.IRGraphFrames

Proofs.Stuttering

Proofs.Form

Proofs.Rewrites

Proofs.Bisimulation

begin

1.1 Individual Elimination Rules

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

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

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

inductive *implies* :: *IRGraph* \Rightarrow *IRNode* \Rightarrow *IRNode* \Rightarrow *TriState* \Rightarrow *bool*

(- \vdash - & - \hookrightarrow -) **for** *g* **where**

eq-imp-less:

$g \vdash (\text{IntegerEqualsNode } x \ y) \ \& \ (\text{IntegerLessThanNode } x \ y) \hookrightarrow \text{KnownFalse} \mid$

eq-imp-less-rev:

$g \vdash (\text{IntegerEqualsNode } x \ y) \ \& \ (\text{IntegerLessThanNode } y \ x) \hookrightarrow \text{KnownFalse} \mid$

less-imp-rev-less:

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

x-imp-x:
 $g \vdash x \ \& \ x \hookrightarrow \text{KnownTrue} \mid$

negate-false:
 $\llbracket g \vdash x \ \& \ (\text{kind } g \ y) \hookrightarrow \text{KnownTrue} \rrbracket \implies g \vdash x \ \& \ (\text{LogicNegationNode } y) \hookrightarrow \text{KnownFalse} \mid$
negate-true:
 $\llbracket g \vdash x \ \& \ (\text{kind } g \ y) \hookrightarrow \text{KnownFalse} \rrbracket \implies g \vdash x \ \& \ (\text{LogicNegationNode } y) \hookrightarrow \text{KnownTrue}$

Total relation over partial implies relation

inductive *condition-implies* :: *IRGraph* \Rightarrow *IRNode* \Rightarrow *IRNode* \Rightarrow *TriState* \Rightarrow *bool*
 $(- \vdash - \ \& \ - \rightarrow -)$ **for** *g* **where**
 $\llbracket \neg(g \vdash a \ \& \ b \hookrightarrow \text{imp}) \rrbracket \implies (g \vdash a \ \& \ b \rightarrow \text{Unknown}) \mid$
 $\llbracket (g \vdash a \ \& \ b \hookrightarrow \text{imp}) \rrbracket \implies (g \vdash a \ \& \ b \rightarrow \text{imp})$

inductive *implies-tree* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* \Rightarrow *bool*
 $(- \ \& \ - \hookrightarrow -)$ **where**
eq-imp-less:
 $(\text{BinaryExpr } \text{BinIntegerEquals } x \ y) \ \& \ (\text{BinaryExpr } \text{BinIntegerLessThan } x \ y) \hookrightarrow \text{False} \mid$
eq-imp-less-rev:
 $(\text{BinaryExpr } \text{BinIntegerEquals } x \ y) \ \& \ (\text{BinaryExpr } \text{BinIntegerLessThan } y \ x) \hookrightarrow \text{False} \mid$
less-imp-rev-less:
 $(\text{BinaryExpr } \text{BinIntegerLessThan } x \ y) \ \& \ (\text{BinaryExpr } \text{BinIntegerLessThan } y \ x) \hookrightarrow \text{False} \mid$
less-imp-not-eq:
 $(\text{BinaryExpr } \text{BinIntegerLessThan } x \ y) \ \& \ (\text{BinaryExpr } \text{BinIntegerEquals } x \ y) \hookrightarrow \text{False} \mid$
less-imp-not-eq-rev:
 $(\text{BinaryExpr } \text{BinIntegerLessThan } x \ y) \ \& \ (\text{BinaryExpr } \text{BinIntegerEquals } y \ x) \hookrightarrow \text{False} \mid$

x-imp-x:
 $x \ \& \ x \hookrightarrow \text{True} \mid$

negate-false:
 $\llbracket x \ \& \ y \hookrightarrow \text{True} \rrbracket \implies x \ \& \ (\text{UnaryExpr } \text{UnaryLogicNegation } y) \hookrightarrow \text{False} \mid$
negate-true:
 $\llbracket x \ \& \ y \hookrightarrow \text{False} \rrbracket \implies x \ \& \ (\text{UnaryExpr } \text{UnaryLogicNegation } y) \hookrightarrow \text{True}$

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

experiment begin

lemma *logic-negate-type*:

assumes $[m, p] \vdash \text{UnaryExpr } \text{UnaryLogicNegation } x \mapsto v$

assumes $v \neq \text{UndefVal}$

shows $\exists v2. [m, p] \vdash x \mapsto \text{IntVal32 } v2$

proof –

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

using *assms(1)* **by** *blast*

then have $[m, p] \vdash \text{UnaryExpr } \text{UnaryLogicNegation } x \mapsto \text{unary-eval } \text{UnaryLogicNegation } ve$

by (*metis* *UnaryExprE* *assms(1)* *evalDet*)

then show *?thesis* **using** *assms* *unary-eval.elims* *evalDet* *ve* *IRUnaryOp.distinct*

sorry

qed

lemma *logic-negation-relation-tree*:

assumes $[m, p] \vdash y \mapsto val$

assumes $[m, p] \vdash \text{UnaryExpr } \text{UnaryLogicNegation } y \mapsto \text{invval}$

assumes $\text{intval} \neq \text{UndefVal}$

shows $\text{val-to-bool } val \longleftrightarrow \neg(\text{val-to-bool } \text{invval})$

proof –

obtain v **where** $\text{invval} = \text{unary-eval } \text{UnaryLogicNegation } v$

using *assms(2)* **by** *blast*

then have $[m, p] \vdash y \mapsto v$ **using** *UnaryExprE* *assms(1,2)* **sorry**

then show *?thesis* **sorry**

qed

lemma *logic-negation-relation*:

assumes $[g, m, p] \vdash y \mapsto val$

assumes $\text{kind } g \text{ neg} = \text{LogicNegationNode } y$

assumes $[g, m, p] \vdash \text{neg} \mapsto \text{invval}$

assumes $\text{intval} \neq \text{UndefVal}$

shows $\text{val-to-bool } val \longleftrightarrow \neg(\text{val-to-bool } \text{invval})$

proof –

obtain $y\text{encode}$ **where** $g \vdash y \simeq y\text{encode}$

using *assms(1)* *encodeeval-def* **by** *auto*

then have $g \vdash \text{neg} \simeq \text{UnaryExpr } \text{UnaryLogicNegation } y\text{encode}$

using *rep.intros(7)* *assms(2)* **by** *simp*

then have $[m, p] \vdash \text{UnaryExpr } \text{UnaryLogicNegation } y\text{encode} \mapsto \text{invval}$

using *assms(3)* *encodeeval-def*

by (*metis* *repDet*)

obtain $v1$ **where** $[g, m, p] \vdash y \mapsto \text{IntVal32 } v1$

using *assms(1,2,3,4)* **using** *logic-negate-type* **sorry**

have $\text{invval} = \text{bool-to-val } (\neg(\text{val-to-bool } val))$

using *assms(1,2,3)* *evalDet* *unary-eval.simps(4)*

by (*smt* (*verit*, *ccfv-SIG*) *UnaryExprE* $\langle [m, p] \vdash \text{UnaryExpr } \text{UnaryLogicNegation}$

```

yencode  $\mapsto$  invval  $\langle g \vdash y \simeq yencode \rangle$  bool-to-val.simps(1) bool-to-val.simps(2) en-
codeeval-defintval-logic-negation.simps(1) logic-negate-type repDet val-to-bool.simps(1))
  have val-to-bool invval  $\longleftrightarrow \neg(\text{val-to-bool val})$ 
  using  $\langle \text{invval} = \text{bool-to-val } (\neg \text{val-to-bool val}) \rangle$  by force
  then show ?thesis
  by simp
qed
end

```

lemma *implies-valid*:

```

assumes  $x \ \& \ y \hookrightarrow \text{imp}$ 
assumes  $[m, p] \vdash x \mapsto v1$ 
assumes  $[m, p] \vdash y \mapsto v2$ 
assumes  $v1 \neq \text{UndefVal} \wedge v2 \neq \text{UndefVal}$ 
shows  $(\text{imp} \longrightarrow (\text{val-to-bool } v1 \longrightarrow \text{val-to-bool } v2)) \wedge$ 
 $(\neg \text{imp} \longrightarrow (\text{val-to-bool } v1 \longrightarrow \neg(\text{val-to-bool } v2)))$ 
(is  $(?TP \longrightarrow ?TC) \wedge (?FP \longrightarrow ?FC)$ )
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)
    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 y)
    then show ?case
    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 egeval: [m, p] ⊢ (BinaryExpr BinIntegerEquals x y) ↦ intval-equals xval
yval
      using xval yval evaltree.BinaryExpr
      by (metis BinaryExprE bin-eval.simps(10) 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(11) eq-imp-less.prems(2) evalDet)
      have val-to-bool (intval-equals xval yval) ⟶ ¬(val-to-bool (intval-less-than xval
yval))
      using assms(4) apply (cases xval; cases yval; auto)
      apply (metis (full-types) val-to-bool.simps(1) Values.bool-to-val.simps(2)
signed.less-irrefl)
      by (metis (mono-tags) val-to-bool.simps(1) Values.bool-to-val.elims signed.order.strict-implies-not-eq)
      then show ?case
        using egeval 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] ⊢ x ↦ xval
      using eq-imp-less-rev.prems(3)
      using eq-imp-less-rev.prems(2) by blast
      obtain yval where yval: [m, p] ⊢ y ↦ yval
      using eq-imp-less-rev.prems(3)
      using eq-imp-less-rev.prems(2) by blast
      have egeval: [m, p] ⊢ (BinaryExpr BinIntegerEquals x y) ↦ intval-equals xval
yval
      using xval yval evaltree.BinaryExpr
      by (metis BinaryExprE bin-eval.simps(10) eq-imp-less-rev.prems(1) evalDet)
      have lesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan y x) ↦ intval-less-than
yval xval
      using xval yval evaltree.BinaryExpr
      by (metis BinaryExprE bin-eval.simps(11) eq-imp-less-rev.prems(2) evalDet)
      have val-to-bool (intval-equals xval yval) ⟶ ¬(val-to-bool (intval-less-than yval
xval))
      using assms(4) apply (cases xval; cases yval; auto)
      apply (metis (full-types) val-to-bool.simps(1) Values.bool-to-val.simps(2)
signed.less-irrefl)
      by (metis (full-types) val-to-bool.simps(1) Values.bool-to-val.elims signed.order.strict-implies-not-eq)

```

```

then show ?case
  using egeval lesseval
  by (metis eq-imp-less-rev.premis(1) eq-imp-less-rev.premis(2) evalDet)
next
case (less-imp-rev-less x y)
obtain xval where xval: [m, p] ⊢ x ↦ xval
  using less-imp-rev-less.premis(3)
  using less-imp-rev-less.premis(2) by blast
obtain yval where yval: [m, p] ⊢ y ↦ yval
  using less-imp-rev-less.premis(3)
  using less-imp-rev-less.premis(2) by blast
have lesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ intval-less-than
xval yval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(11) evalDet less-imp-rev-less.premis(1))
  have revlesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan y x) ↦ int-
val-less-than yval xval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(11) evalDet less-imp-rev-less.premis(2))
  have val-to-bool (intval-less-than xval yval) ⟶ ¬(val-to-bool (intval-less-than
yval xval))
  using assms(4) apply (cases xval; cases yval; auto)
  apply (metis val-to-bool.simps(1) Values.bool-to-val.elims signed.not-less-iff-gr-or-eq)
  by (metis val-to-bool.simps(1) Values.bool-to-val.elims signed.less-asym')
then show ?case
  by (metis evalDet less-imp-rev-less.premis(1) less-imp-rev-less.premis(2) lesseval
revlesseval)
next
case (less-imp-not-eq x y)
obtain xval where xval: [m, p] ⊢ x ↦ xval
  using less-imp-not-eq.premis(3)
  using less-imp-not-eq.premis(1) by blast
obtain yval where yval: [m, p] ⊢ y ↦ yval
  using less-imp-not-eq.premis(3)
  using less-imp-not-eq.premis(1) by blast
have egeval: [m, p] ⊢ (BinaryExpr BinIntegerEquals x y) ↦ intval-equals xval
yval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(10) evalDet less-imp-not-eq.premis(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(11) evalDet less-imp-not-eq.premis(1))
  have val-to-bool (intval-less-than xval yval) ⟶ ¬(val-to-bool (intval-equals xval
yval))
  using assms(4) apply (cases xval; cases yval; auto)
  apply (metis (full-types) bool-to-val.simps(2) signed.less-imp-not-eq val-to-bool.simps(1))
  by (metis (full-types) bool-to-val.simps(2) signed.less-imp-not-eq2 val-to-bool.simps(1))
then show ?case

```

```

    by (metis egeval evalDet less-imp-not-eq.prem(1) less-imp-not-eq.prem(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.prem(3)
    using less-imp-not-eq-rev.prem(1) by blast
    obtain yval where yval: [m, p] ⊢ y ↦ yval
    using less-imp-not-eq-rev.prem(3)
    using less-imp-not-eq-rev.prem(1) by blast
    have egeval: [m, p] ⊢ (BinaryExpr BinIntegerEquals y x) ↦ intval-equals yval
xval
    using xval yval evaltree.BinaryExpr
    by (metis BinaryExprE bin-eval.simp(10) evalDet less-imp-not-eq-rev.prem(2))
    have lesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ intval-less-than
xval yval
    using xval yval evaltree.BinaryExpr
    by (metis BinaryExprE bin-eval.simp(11) evalDet less-imp-not-eq-rev.prem(1))
    have val-to-bool (intval-less-than xval yval) ⟶ ¬(val-to-bool (intval-equals yval
xval))
    using assms(4) apply (cases xval; cases yval; auto)
    apply (metis (full-types) bool-to-val.simp(2) signed.less-imp-not-eq2 val-to-bool.simp(1))
    by (metis (full-types, opaque-lifting) val-to-bool.simp(1) Values.bool-to-val.elims
signed.dual-order.strict-implies-not-eq)
    then show ?case
    by (metis egeval evalDet less-imp-not-eq-rev.prem(1) less-imp-not-eq-rev.prem(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
  assumes v1 ≠ UndefVal ∧ v2 ≠ UndefVal
  shows val-to-bool v1 ⟶ val-to-bool v2
  using assms implies-valid
  by blast

```

```

lemma implies-false-valid:
  assumes  $x \ \& \ y \hookrightarrow \text{imp}$ 
  assumes  $\neg \text{imp}$ 
  assumes  $[m, p] \vdash x \mapsto v1$ 
  assumes  $[m, p] \vdash y \mapsto v2$ 
  assumes  $v1 \neq \text{UndefVal} \wedge v2 \neq \text{UndefVal}$ 
  shows  $\text{val-to-bool } v1 \longrightarrow \neg(\text{val-to-bool } v2)$ 
  using assms implies-valid by blast

```

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

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

```

inductive tryFold :: IRNode  $\Rightarrow$  (ID  $\Rightarrow$  Stamp)  $\Rightarrow$  bool  $\Rightarrow$  bool
  where
     $\llbracket \text{alwaysDistinct } (\text{stamps } x) (\text{stamps } y) \rrbracket$ 
       $\implies \text{tryFold } (\text{IntegerEqualsNode } x \ y) \ \text{stamps } \text{False} \mid$ 
     $\llbracket \text{neverDistinct } (\text{stamps } x) (\text{stamps } y) \rrbracket$ 
       $\implies \text{tryFold } (\text{IntegerEqualsNode } x \ y) \ \text{stamps } \text{True} \mid$ 
     $\llbracket \text{is-IntegerStamp } (\text{stamps } x);$ 
       $\text{is-IntegerStamp } (\text{stamps } y);$ 
       $\text{stpi-upper } (\text{stamps } x) < \text{stpi-lower } (\text{stamps } y) \rrbracket$ 
       $\implies \text{tryFold } (\text{IntegerLessThanNode } x \ y) \ \text{stamps } \text{True} \mid$ 
     $\llbracket \text{is-IntegerStamp } (\text{stamps } x);$ 
       $\text{is-IntegerStamp } (\text{stamps } y);$ 
       $\text{stpi-lower } (\text{stamps } x) \geq \text{stpi-upper } (\text{stamps } y) \rrbracket$ 
       $\implies \text{tryFold } (\text{IntegerLessThanNode } x \ y) \ \text{stamps } \text{False}$ 

```

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  $\text{kind } g \ \text{nid} = \text{IntegerEqualsNode } x \ y$ 
  assumes  $[g, m, p] \vdash \text{nid} \mapsto v$ 
  assumes  $v \neq \text{UndefVal}$ 
  assumes  $([g, m, p] \vdash x \mapsto xval) \wedge ([g, m, p] \vdash y \mapsto yval)$ 
  shows  $\text{val-to-bool } (\text{intval-equals } xval \ yval) \longleftrightarrow v = \text{IntVal32 } 1$ 
proof –
  have  $v = \text{intval-equals } xval \ yval$ 
    using assms(1, 2, 3, 4) BinaryExprE IntegerEqualsNode bin-eval.simps(7)
    by (smt (verit) bin-eval.simps(10) encodeeval-def evalDet repDet)
  then show ?thesis using intval-equals.simps val-to-bool.simps sorry
qed

```

```

lemma tryFoldIntegerEqualsAlwaysDistinct:
  assumes wf-stamp  $g \ \text{stamps}$ 
  assumes  $\text{kind } g \ \text{nid} = (\text{IntegerEqualsNode } x \ y)$ 

```



```

assumes  $[g, m, p] \vdash nid \mapsto v$ 
assumes alwaysDistinct (stamps x) (stamps y)
shows  $v = \text{IntVal32 } 0$ 
proof –
  have  $\forall \text{ val. } \neg(\text{valid-value val } (\text{join } (\text{stamps } x) (\text{stamps } y)))$ 
    using assms(1,4) unfolding alwaysDistinct.simps
  by (metis is-stamp-empty.elims(2) le-less-trans not-less valid32or64 valid-value.simps(1)
valid-value.simps(2))
  have  $\neg(\exists \text{ val. } ([g, m, p] \vdash x \mapsto \text{val}) \wedge ([g, m, p] \vdash y \mapsto \text{val}))$ 
    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  $\text{kind } g \text{ nid} = (\text{IntegerEqualsNode } x \ y)$ 
  assumes  $[g, m, p] \vdash nid \mapsto v$ 
  assumes neverDistinct (stamps x) (stamps y)
  shows  $v = \text{IntVal32 } 1$ 
  using assms IntegerEqualsNodeE sorry

```

```

lemma tryFoldIntegerLessThanTrue:
  assumes wf-stamp g stamps
  assumes  $\text{kind } g \text{ nid} = (\text{IntegerLessThanNode } x \ y)$ 
  assumes  $[g, m, p] \vdash nid \mapsto v$ 
  assumes  $\text{stpi-upper } (\text{stamps } x) < \text{stpi-lower } (\text{stamps } y)$ 
  shows  $v = \text{IntVal32 } 1$ 
proof –
  have stamp-type: is-IntegerStamp (stamps x)
    using assms
  sorry
  obtain xval where xval:  $[g, m, p] \vdash x \mapsto \text{xval}$ 
    using assms(2,3) sorry
  obtain yval where yval:  $[g, m, p] \vdash y \mapsto \text{yval}$ 
    using assms(2,3) sorry
  have is-IntegerStamp (stamps x)  $\wedge$  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  $\text{kind } g \text{ nid} = (\text{IntegerLessThanNode } x \ y)$ 
  assumes  $[g, m, p] \vdash nid \mapsto v$ 

```

```

assumes stpi-lower (stamps x)  $\geq$  stpi-upper (stamps y)
shows v = IntVal32 0
proof –
have stamp-type: is-IntegerStamp (stamps x)
  using assms
  sorry
obtain xval where xval: [g, m, p]  $\vdash$  x  $\mapsto$  xval
  using assms(2,3) sorry
obtain yval where yval: [g, m, p]  $\vdash$  y  $\mapsto$  yval
  using assms(2,3) sorry
have is-IntegerStamp (stamps x)  $\wedge$  is-IntegerStamp (stamps y)
  using assms(4)
  sorry
then have  $\neg$ (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]  $\vdash$  nid  $\mapsto$  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 sorry
next
  case (2 stamps x y)
  then show ?case using tryFoldIntegerEqualsAlwaysDistinct 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

theorem tryFoldProofFalse:
  assumes wf-stamp g stamps
  assumes tryFold (kind g nid) stamps False
  assumes [g, m, p]  $\vdash$  nid  $\mapsto$  v
  shows  $\neg$ (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*:

$g, p \vdash (nid, m, h) \rightarrow (nid', m', h)$

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

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

inductive *ConditionalEliminationStep* ::

$IRExpr \text{ set} \Rightarrow (ID \Rightarrow Stamp) \Rightarrow IRGraph \Rightarrow ID \Rightarrow IRGraph \Rightarrow bool$ **where**
impliesTrue:

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

impliesFalse:

$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
 $g \vdash cid \simeq cond;$
 $\exists\ ce \in conds . (ce \ \&\ cond \hookrightarrow False);$
 $g' = constantCondition\ False\ ifcond\ (kind\ g\ ifcond)\ g$
 $\rrbracket \implies ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g' \mid$

tryFoldTrue:

$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
 $cond = kind\ g\ cid;$
 $tryFold\ (kind\ g\ cid)\ stamps\ True;$
 $g' = constantCondition\ True\ ifcond\ (kind\ g\ ifcond)\ g$
 $\rrbracket \implies ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g' \mid$

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
  ] ==> ConditionalEliminationStep conds stamps g ifcond g'

```

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

thm *ConditionalEliminationStep.equation*

1.2 Control-flow Graph Traversal

```

type-synonym Seen = ID set
type-synonym Condition = IRNode
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` function 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  $\Rightarrow$  Condition  $\Rightarrow$  (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 transitions between individual nodes of the graph.

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

inductive Step
  :: IRGraph  $\Rightarrow$  (ID  $\times$  Seen  $\times$  Conditions  $\times$  StampFlow)  $\Rightarrow$  (ID  $\times$  Seen  $\times$  Conditions  $\times$  StampFlow) option  $\Rightarrow$  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
     $\llbracket$ kind g nid = BeginNode nid';

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

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

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

```

$flow' = registerNewCondition\ g\ c\ (hdOr\ flow\ (stamp\ g))$
 $\implies Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds',\ flow'\ \# \ flow))\ |$

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

$\llbracket kind\ g\ nid = EndNode;$

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

$nid' = any-usage\ g\ nid;$

$conds' = tl\ conds;$

$flow' = tl\ flow$

$\implies Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds',\ flow'))\ |$

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

$\llbracket \neg(is-EndNode\ (kind\ g\ nid));$
 $\neg(is-BeginNode\ (kind\ g\ nid));$

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

$Some\ nid' = nextEdge\ seen'\ nid\ g$

$\implies Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds,\ flow))\ |$

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

$\llbracket \neg(is-EndNode\ (kind\ g\ nid));$
 $\neg(is-BeginNode\ (kind\ g\ nid));$

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

$None = nextEdge\ seen'\ nid\ g$

$\implies Step\ g\ (nid,\ seen,\ conds,\ flow)\ None\ |$

— We've already seen this node, give back None

$\llbracket nid \in seen \rrbracket \implies Step\ g\ (nid,\ seen,\ conds,\ flow)\ None$

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

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

end