

Veriopt Theories

October 10, 2022

Contents

1	Conditional Elimination Phase	1
1.1	Individual Elimination Rules	1
1.2	Control-flow Graph Traversal	11

1 Conditional Elimination Phase

```
theory ConditionalElimination
  imports
    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 ⇒ 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:

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

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 \vdash $\&$ \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.

```

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

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

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

lemma logic-negation-relation:
  assumes  $[g, m, p] \vdash y \mapsto \text{val}$ 
  assumes  $\text{kind } g \ \text{neg} = \text{LogicNegationNode } y$ 
  assumes  $[g, m, p] \vdash \text{neg} \mapsto \text{invval}$ 
  assumes  $\text{invval} \neq \text{UndefVal}$ 
  shows  $\text{val-to-bool } \text{val} \longleftrightarrow \neg(\text{val-to-bool } \text{invval})$ 
  using assms
  by (metis LogicNegationNode encodeeval-def logic-negation-relation-tree repDet)

lemma implies-valid:
  assumes  $x \ \& \ y \hookrightarrow \text{imp}$ 
  assumes  $[m, p] \vdash x \mapsto v1$ 
  assumes  $[m, p] \vdash y \mapsto v2$ 
  shows  $(\text{imp} \longrightarrow (\text{val-to-bool } v1 \longrightarrow \text{val-to-bool } v2)) \wedge$ 
   $(\neg \text{imp} \longrightarrow (\text{val-to-bool } v1 \longrightarrow \neg(\text{val-to-bool } v2)))$ 
   $(\text{is } (?TP \longrightarrow ?TC) \wedge (?FP \longrightarrow ?FC))$ 
  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 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.prem(3)
    by blast
    then obtain yval where yval: [m, p] ⊢ y ↦ yval
    using eq-imp-less.prem(3)
    using eq-imp-less.prem(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(11) eq-imp-less.prem(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.prem(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
    using egeval lesseval
    by (metis eq-imp-less.prem(1) eq-imp-less.prem(2) evalDet)
next
    case (eq-imp-less-rev x y)
    obtain xval where xval: [m, p] ⊢ x ↦ xval
    using eq-imp-less-rev.prem(3)
    using eq-imp-less-rev.prem(2) by blast
    obtain yval where yval: [m, p] ⊢ y ↦ yval

```

```

    using eq-imp-less-rev.premis(3)
    using eq-imp-less-rev.premis(2) by blast
    have egeval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerEquals } x \ y) \mapsto \text{intval-equals } xval$ 
    yval
    using xval yval evaltree.BinaryExpr
    by (metis BinaryExprE bin-eval.simps(11) eq-imp-less-rev.premis(1) evalDet)
    have lesseval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerLessThan } y \ x) \mapsto \text{intval-less-than}$ 
    yval xval
    using xval yval evaltree.BinaryExpr
    by (metis BinaryExprE bin-eval.simps(12) eq-imp-less-rev.premis(2) evalDet)
    have val-to-bool (intval-equals xval yval)  $\longrightarrow \neg(\text{val-to-bool } (\text{intval-less-than } yval$ 
    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 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:  $[m, p] \vdash x \mapsto xval$ 
  using less-imp-rev-less.premis(3)
  using less-imp-rev-less.premis(2) by blast
  obtain yval where yval:  $[m, p] \vdash y \mapsto yval$ 
  using less-imp-rev-less.premis(3)
  using less-imp-rev-less.premis(2) by blast
  have lesseval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerLessThan } x \ y) \mapsto \text{intval-less-than}$ 
  xval yval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(12) evalDet less-imp-rev-less.premis(1))
  have revlesseval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerLessThan } y \ x) \mapsto \text{intval-less-than}$ 
  yval xval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(12) evalDet less-imp-rev-less.premis(2))
  have val-to-bool (intval-less-than xval yval)  $\longrightarrow \neg(\text{val-to-bool } (\text{intval-less-than}$ 
  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.premis(1) less-imp-rev-less.premis(2) lesse-
  val revlesseval)
  next
  case (less-imp-not-eq x y)
  obtain xval where xval:  $[m, p] \vdash x \mapsto xval$ 
  using less-imp-not-eq.premis(3)
  using less-imp-not-eq.premis(1) by blast
  obtain yval where yval:  $[m, p] \vdash y \mapsto yval$ 
  using less-imp-not-eq.premis(3)
  using less-imp-not-eq.premis(1) by blast
  have egeval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerEquals } x \ y) \mapsto \text{intval-equals } xval$ 

```

```

yval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(11) evalDet less-imp-not-eq.prem(2))
  have lesseval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerLessThan } x \ y) \mapsto \text{intval-less-than}$ 
xval yval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(12) evalDet less-imp-not-eq.prem(1))
  have val-to-bool (intval-less-than xval yval)  $\longrightarrow \neg(\text{val-to-bool (intval-equals xval yval)})$ 
  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 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:  $[m, p] \vdash x \mapsto \text{xval}$ 
  using less-imp-not-eq-rev.prem(3)
  using less-imp-not-eq-rev.prem(1) by blast
  obtain yval where yval:  $[m, p] \vdash y \mapsto \text{yval}$ 
  using less-imp-not-eq-rev.prem(3)
  using less-imp-not-eq-rev.prem(1) by blast
  have egeval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerEquals } y \ x) \mapsto \text{intval-equals yval xval}$ 
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(11) evalDet less-imp-not-eq-rev.prem(2))
  have lesseval:  $[m, p] \vdash (\text{BinaryExpr BinIntegerLessThan } x \ y) \mapsto \text{intval-less-than}$ 
xval yval
  using xval yval evaltree.BinaryExpr
  by (metis BinaryExprE bin-eval.simps(12) evalDet less-imp-not-eq-rev.prem(1))
  have val-to-bool (intval-less-than xval yval)  $\longrightarrow \neg(\text{val-to-bool (intval-equals yval xval)})$ 
  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 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  $x \ \& \ y \hookrightarrow \text{imp}$ 
  assumes imp
  assumes  $[m, p] \vdash x \mapsto v1$ 
  assumes  $[m, p] \vdash y \mapsto v2$ 
  shows  $\text{val-to-bool } v1 \longrightarrow \text{val-to-bool } v2$ 
  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  $[m, p] \vdash y \mapsto v2$ 
  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  $([g, m, p] \vdash x \mapsto xval) \wedge ([g, m, p] \vdash y \mapsto yval)$ 
  shows  $\text{val-to-bool } (\text{intval-equals } xval \ yval) \longleftrightarrow v = \text{IntVal } 32 \ 1$ 
proof –
  have  $v = \text{intval-equals } xval \ yval$ 
    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 encodeeval-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  $[g, m, p] \vdash nid \mapsto v$ 
  assumes  $stpi\text{-}lower(stamps\ x) \geq stpi\text{-}upper(stamps\ y)$ 
  shows  $v = IntVal\ 32\ 0$ 
  proof -
    have stamp-type: is-IntegerStamp (stamps x)
      using assms
    sorry
  obtain xval where  $xval: [g, m, p] \vdash x \mapsto xval$ 
    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  $\neg(val\text{-}to\text{-}bool\ (intval\text{-}less\text{-}than\ xval\ yval))$ 
    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
      by force
  next
  case (2 stamps x y)
    then show ?case using tryFoldIntegerEqualsAlwaysDistinct assms
      by (smt (verit, best) one-neq-zero tryFold.cases tryFoldIntegerEqualsNeverDis-
tinguish 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 tryFoldIntegerEqualsNeverDis-
tinguish 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*:

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

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:

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

impliesFalse:

$$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$$

```

g ⊢ cid ≃ cond;
∃ ce ∈ conds . (ce & cond ⇔ False);
g' = constantCondition False ifcond (kind g ifcond) g
|| ⇒ ConditionalEliminationStep conds stamps g ifcond g' |

```

```

tryFoldTrue:
[[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
 || ⇒ ConditionalEliminationStep conds stamps g ifcond g' |

```

```

tryFoldFalse:
[[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 = IRExpr
type-synonym Conditions = Condition list
type-synonym StampFlow = (ID ⇒ Stamp) list

```

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

```

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

```

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

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

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

```

fun pred :: IRGraph ⇒ ID ⇒ ID option where

```

```

pred g nid = (case kind g nid of
  (MergeNode ends -) => Some (hd ends) |
  - =>
    (if IRGraph.predecessors g nid = {}
     then None else
     Some (hd (sorted-list-of-set (IRGraph.predecessors g nid))))
)

```

When the basic block of an if statement is entered, we know that the condition of the preceding if statement must be true. As in the GraalVM compiler, we introduce the registerNewCondition 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 => int => Stamp where
  clip-upper (IntegerStamp b l h) c = (IntegerStamp b l c) |
  clip-upper s c = s
fun clip-lower :: Stamp => int => Stamp where
  clip-lower (IntegerStamp b l h) c = (IntegerStamp b c h) |
  clip-lower s c = s

```

```

fun registerNewCondition :: IRGraph => IRNode => (ID => Stamp) => (ID => Stamp) where

```

```

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

```

```

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

```

```

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

```

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

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

```

inductive Step
  :: IRGraph => (ID × Seen × Conditions × StampFlow) => (ID × Seen × Con-
    ditions × 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

$\llbracket kind\ g\ nid = BeginNode\ nid';$

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

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

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

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

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

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

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

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

$conds' = tl\ conds;$
 $flow' = tl\ flow$

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

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

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

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

$Some\ nid' = nextEdge\ seen'\ nid\ g$
 $\implies Step\ g\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds, flow)) \mid$

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

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

$\neg(\text{is-BEGINNode } (\text{kind } g \text{ nid}));$

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

$\text{None} = \text{nextEdge } \text{seen}' \text{ nid } g$
 $\implies \text{Step } g \text{ (nid, seen, conds, flow) None} \mid$

— We've already seen this node, give back None
 $\llbracket \text{nid} \in \text{seen} \rrbracket \implies \text{Step } g \text{ (nid, seen, conds, flow) None}$

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

$:: \text{IRGraph} \Rightarrow (\text{ID} \times \text{Seen} \times \text{Conditions} \times \text{StampFlow}) \Rightarrow \text{IRGraph} \Rightarrow \text{bool}$

where

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

$\text{ConditionalEliminationPhase } g' \text{ (nid', seen', conds', flow')} g' \rrbracket$
 $\implies \text{ConditionalEliminationPhase } g \text{ (nid, seen, conds, flow) } g'' \mid$

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

$\text{ConditionalEliminationPhase } g \text{ (nid', seen', conds', flow')} g' \rrbracket$
 $\implies \text{ConditionalEliminationPhase } g \text{ (nid, seen, conds, flow) } g' \mid$

— Can't do a step but there is a predecessor we can backtrace to
 $\llbracket \text{Step } g \text{ (nid, seen, conds, flow) None} \rrbracket;$
 $\text{Some nid}' = \text{pred } g \text{ nid};$
 $\text{seen}' = \{\text{nid}\} \cup \text{seen};$
 $\text{ConditionalEliminationPhase } g \text{ (nid', seen', conds, flow) } g' \rrbracket$
 $\implies \text{ConditionalEliminationPhase } g \text{ (nid, seen, conds, flow) } g' \mid$

— Can't do a step and have no predecessors so terminate
 $\llbracket \text{Step } g \text{ (nid, seen, conds, flow) None} \rrbracket;$
 $\text{None} = \text{pred } g \text{ nid} \rrbracket$
 $\implies \text{ConditionalEliminationPhase } g \text{ (nid, seen, conds, flow) } g$

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

```

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

end

```