

Veriopt Theories

April 17, 2024

Contents

1	Conditional Elimination Phase	1
1.1	Implication Rules	1
1.1.1	Structural Implication	2
1.1.2	Type Implication	3
1.2	Lift rules	6
1.3	Control-flow Graph Traversal	7

1 Conditional Elimination Phase

This theory presents the specification of the `ConditionalElimination` phase within the GraalVM compiler. The `ConditionalElimination` phase simplifies any condition of an `if` statement that can be implied by the conditions that dominate it. Such that if condition A implies that condition B *must* be true, the condition B is simplified to `true`.

```
if (A) {
    if (B) {
        ...
    }
}
```

We begin by defining the individual implication rules used by the phase in 1.1. These rules are then lifted to the rewriting of a condition within an `if` statement in ???. The traversal algorithm used by the compiler is specified in ???.

```
theory ConditionalElimination
imports
  Semantics.IRTreeEvalThms
  Proofs.Rewrites
  Proofs.Bisimulation
  OptimizationDSL.Markup
begin
```

```
declare [[show-types=false]]
```

1.1 Implication Rules

The set of rules used for determining whether a condition, q_1 , implies another condition, q_2 , must be true or false.

1.1.1 Structural Implication

The first method for determining if a condition can be implied by another condition, is structural implication. That is, by looking at the structure of the conditions, we can determine the truth value. For instance, $x \equiv y$ implies that $x < y$ cannot be true.

inductive

```
impliesx :: IRExpr ⇒ IRExpr ⇒ bool (- ⇒ -) and
impliesnot :: IRExpr ⇒ IRExpr ⇒ bool (- ⇒¬ -) where
  same:      q ⇒ q |
  eq-not-less: exp[x eq y] ⇒¬ exp[x < y] |
  eq-not-less': exp[x eq y] ⇒¬ exp[y < x] |
  less-not-less: exp[x < y] ⇒¬ exp[y < x] |
  less-not-eq:  exp[x < y] ⇒¬ exp[x eq y] |
  less-not-eq': exp[x < y] ⇒¬ exp[y eq x] |
  negate-true:  [x ⇒¬ y] ==> x ⇒ exp[!y] |
  negate-false: [x ⇒ y] ==> x ⇒¬ exp[!y]
```

inductive implies-complete :: IRExpr ⇒ IRExpr ⇒ bool option ⇒ bool where

```
implies:
  x ⇒ y ==> implies-complete x y (Some True) |
  impliesnot:
  x ⇒¬ y ==> implies-complete x y (Some False) |
  fail:
  ¬((x ⇒ y) ∨ (x ⇒¬ y)) ==> implies-complete x y None
```

The relation $q_1 \Rightarrow q_2$ requires that the implication $q_1 \rightarrow q_2$ is known true (i.e. universally valid). The relation $q_1 \Rightarrow¬ q_2$ requires that the implication $q_1 \rightarrow q_2$ is known false (i.e. $q_1 \rightarrow \neg q_2$ is universally valid). If neither $q_1 \Rightarrow q_2$ nor $q_1 \Rightarrow¬ q_2$ then the status is unknown and the condition cannot be simplified.

fun implies-valid :: IRExpr ⇒ IRExpr ⇒ bool (infix ↪ 50) where

```
implies-valid q1 q2 =
  (forall m p v1 v2. ([m, p] ⊢ q1 ↪ v1) ∧ ([m, p] ⊢ q2 ↪ v2) →
    (val-to-bool v1 → val-to-bool v2))
```

fun impliesnot-valid :: IRExpr ⇒ IRExpr ⇒ bool (infix ↪ 50) where

```
impliesnot-valid q1 q2 =
  (forall m p v1 v2. ([m, p] ⊢ q1 ↪ v1) ∧ ([m, p] ⊢ q2 ↪ v2) →
```

$$(val\text{-}to\text{-}bool v1 \longrightarrow \neg val\text{-}to\text{-}bool v2))$$

The relation $q_1 \rightarrow q_2$ means $q_1 \longrightarrow q_2$ is universally valid, and the relation $q_1 \leftrightarrow q_2$ means $q_1 \longrightarrow \neg q_2$ is universally valid.

lemma *eq-not-less-val*:

$$\begin{aligned} & val\text{-}to\text{-}bool(val[v1 \text{ eq } v2]) \longrightarrow \neg val\text{-}to\text{-}bool(val[v1 < v2]) \\ & \langle proof \rangle \end{aligned}$$

lemma *eq-not-less'-val*:

$$\begin{aligned} & val\text{-}to\text{-}bool(val[v1 \text{ eq } v2]) \longrightarrow \neg val\text{-}to\text{-}bool(val[v2 < v1]) \\ & \langle proof \rangle \end{aligned}$$

lemma *less-not-less-val*:

$$\begin{aligned} & val\text{-}to\text{-}bool(val[v1 < v2]) \longrightarrow \neg val\text{-}to\text{-}bool(val[v2 < v1]) \\ & \langle proof \rangle \end{aligned}$$

lemma *less-not-eq-val*:

$$\begin{aligned} & val\text{-}to\text{-}bool(val[v1 < v2]) \longrightarrow \neg val\text{-}to\text{-}bool(val[v1 \text{ eq } v2]) \\ & \langle proof \rangle \end{aligned}$$

lemma *logic-negate-type*:

$$\begin{aligned} & \text{assumes } [m, p] \vdash \text{UnaryExpr UnaryLogicNegation } x \mapsto v \\ & \text{shows } \exists b v2. [m, p] \vdash x \mapsto \text{IntVal } b v2 \\ & \langle proof \rangle \end{aligned}$$

lemma *intval-logic-negation-inverse*:

$$\begin{aligned} & \text{assumes } b > 0 \\ & \text{assumes } x = \text{IntVal } b v \\ & \text{shows } val\text{-}to\text{-}bool (\text{intval-logic-negation } x) \longleftrightarrow \neg (val\text{-}to\text{-}bool x) \\ & \langle proof \rangle \end{aligned}$$

lemma *logic-negation-relation-tree*:

$$\begin{aligned} & \text{assumes } [m, p] \vdash y \mapsto val \\ & \text{assumes } [m, p] \vdash \text{UnaryExpr UnaryLogicNegation } y \mapsto invval \\ & \text{shows } val\text{-}to\text{-}bool val \longleftrightarrow \neg (val\text{-}to\text{-}bool invval) \\ & \langle proof \rangle \end{aligned}$$

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

theorem *implies-impliesnot-valid*:

$$\begin{aligned} & \text{shows } ((q1 \Rightarrow q2) \longrightarrow (q1 \rightarrow q2)) \wedge \\ & \quad ((q1 \Rightarrow \neg q2) \longrightarrow (q1 \leftrightarrow q2)) \\ & \quad (\text{is } (?imp \longrightarrow ?val) \wedge (?notimp \longrightarrow ?notval)) \\ & \langle proof \rangle \end{aligned}$$

1.1.2 Type Implication

The second mechanism to determine whether a condition implies another is to use the type information of the relevant nodes. For instance, $x < (4::'a)$

implies $x < (10::'a)$. We can show this by strengthening the type, stamp, of the node x such that the upper bound is $4::'a$. Then we the second condition is reached, we know that the condition must be true by the upperbound.

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

We track the refined stamps by mapping nodes to Stamps, the second parameter to `tryFold`.

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

code-pred (*modes: i ⇒ i ⇒ i ⇒ bool*) `tryFold` $\langle \text{proof} \rangle$

Prove that, when the stamp map is valid, the `tryFold` relation correctly predicts the output value with respect to our evaluation semantics.

inductive-cases *StepE*:

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

```
lemma is-stamp-empty-valid:
assumes is-stamp-empty s
shows  $\neg(\exists \text{ val. valid-value val } s)$ 
 $\langle \text{proof} \rangle$ 
```

lemma join-valid:

```
assumes is-IntegerStamp s1  $\wedge$  is-IntegerStamp s2
assumes valid-stamp s1  $\wedge$  valid-stamp s2
shows (valid-value v s1  $\wedge$  valid-value v s2) = valid-value v (join s1 s2) (is ?lhs
= ?rhs)
 $\langle \text{proof} \rangle$ 
```

lemma alwaysDistinct-evaluate:

```
assumes wf-stamp g stamps
assumes alwaysDistinct (stamps x) (stamps y)
```

```

assumes is-IntegerStamp (stamps x) ∧ is-IntegerStamp (stamps y) ∧ valid-stamp
(stamps x) ∧ valid-stamp (stamps y)
shows ¬(∃ val . ([g, m, p] ⊢ x ↦ val) ∧ ([g, m, p] ⊢ y ↦ val))
⟨proof⟩

lemma alwaysDistinct-valid:
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 ¬(val-to-bool v)
⟨proof⟩
thm-oracles alwaysDistinct-valid

lemma unwrap-valid:
assumes 0 < b ∧ b ≤ 64
assumes take-bit (b::nat) (vv::64 word) = vv
shows (vv::64 word) = take-bit b (word-of-int (int-signed-value (b::nat) (vv::64
word)))
⟨proof⟩

lemma asConstant-valid:
assumes asConstant s = val
assumes val ≠ UndefVal
assumes valid-value v s
shows v = val
⟨proof⟩

lemma neverDistinct-valid:
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 val-to-bool v
⟨proof⟩

lemma stampUnder-valid:
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 val-to-bool v
⟨proof⟩

lemma stampOver-valid:
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  $\neg(\text{val-to-bool } v)$ 
 $\langle \text{proof} \rangle$ 

theorem tryFoldTrue-valid:
  assumes wf-stamp g stamps
  assumes tryFold (kind g nid) stamps True
  assumes  $[g, m, p] \vdash \text{nid} \mapsto v$ 
  shows val-to-bool v
   $\langle \text{proof} \rangle$ 

theorem tryFoldFalse-valid:
  assumes wf-stamp g stamps
  assumes tryFold (kind g nid) stamps False
  assumes  $[g, m, p] \vdash \text{nid} \mapsto v$ 
  shows  $\neg(\text{val-to-bool } v)$ 
   $\langle \text{proof} \rangle$ 

```

1.2 Lift rules

```

inductive condset-implies :: IRExpr set  $\Rightarrow$  IRExpr  $\Rightarrow$  bool  $\Rightarrow$  bool where
  impliesTrue:
     $(\exists ce \in \text{conds} . (ce \Rightarrow cond)) \implies \text{condset-implies } \text{conds } cond \text{ True}$  |
  impliesFalse:
     $(\exists ce \in \text{conds} . (ce \Rightarrow \neg cond)) \implies \text{condset-implies } \text{conds } cond \text{ False}$ 

code-pred (modes:  $i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$ ) condset-implies  $\langle \text{proof} \rangle$ 

```

The *cond-implies* function lifts the structural and type implication rules to the one relation.

```

fun conds-implies :: IRExpr set  $\Rightarrow$   $(ID \Rightarrow \text{Stamp}) \Rightarrow \text{IRNode} \Rightarrow \text{IRExpr} \Rightarrow \text{bool}$ 
option where
  conds-implies conds stamps condNode cond =
    (if condset-implies conds cond True  $\vee$  tryFold condNode stamps True
      then Some True
    else if condset-implies conds cond False  $\vee$  tryFold condNode stamps False
      then Some False
    else None)

```

Perform conditional elimination rewrites on the graph for a particular node by lifting the individual implication rules to a relation that rewrites the condition of *if* statements to constant values.

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.

```

inductive ConditionalEliminationStep :: 
  IRExpr set  $\Rightarrow$   $(ID \Rightarrow \text{Stamp}) \Rightarrow ID \Rightarrow \text{IRGraph} \Rightarrow \text{IRGraph} \Rightarrow \text{bool}$ 

```

where

impliesTrue:

$$\llbracket \text{kind } g \text{ ifcond} = (\text{IfNode } cid \ t \ f); \\ g \vdash cid \simeq cond; \\ condNode = \text{kind } g \text{ cid}; \\ \text{conds-implies } \text{conds stamps condNode cond} = (\text{Some True}); \\ g' = \text{constantCondition True ifcond} (\text{kind } g \text{ ifcond}) \ g \\ \rrbracket \implies \text{ConditionalEliminationStep} \text{ conds stamps ifcond } g \ g' \mid$$

impliesFalse:

$$\llbracket \text{kind } g \text{ ifcond} = (\text{IfNode } cid \ t \ f); \\ g \vdash cid \simeq cond; \\ condNode = \text{kind } g \text{ cid}; \\ \text{conds-implies } \text{conds stamps condNode cond} = (\text{Some False}); \\ g' = \text{constantCondition False ifcond} (\text{kind } g \text{ ifcond}) \ g \\ \rrbracket \implies \text{ConditionalEliminationStep} \text{ conds stamps ifcond } g \ g' \mid$$

unknown:

$$\llbracket \text{kind } g \text{ ifcond} = (\text{IfNode } cid \ t \ f); \\ g \vdash cid \simeq cond; \\ condNode = \text{kind } g \text{ cid}; \\ \text{conds-implies } \text{conds stamps condNode cond} = \text{None} \\ \rrbracket \implies \text{ConditionalEliminationStep} \text{ conds stamps ifcond } g \ g' \mid$$

notIfNode:

$$\neg(\text{is-IfNode} (\text{kind } g \text{ ifcond})) \implies \text{ConditionalEliminationStep} \text{ conds stamps ifcond } g \ g$$

code-pred (*modes: i* \Rightarrow *i* \Rightarrow *i* \Rightarrow *i* \Rightarrow *o* \Rightarrow *bool*) *ConditionalEliminationStep* $\langle proof \rangle$

thm *ConditionalEliminationStep.equation*

1.3 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*
type-synonym *ToVisit* = *ID 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 wherein 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 preds :: IRGraph ⇒ ID ⇒ ID list where
  preds g nid = (case kind g nid of
    (MergeNode ends - -) ⇒ ends |
    - ⇒
      sorted-list-of-set (IRGraph.predecessors g nid)
  )  
  
fun pred :: IRGraph ⇒ ID ⇒ ID option where
  pred g nid = (case preds g nid of [] ⇒ None | x # xs ⇒ Some x)
```

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 `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 =
    (if c < h then (IntegerStamp b l c) else (IntegerStamp b l h)) |
  clip-upper s c = s  
  
fun clip-lower :: Stamp ⇒ int ⇒ Stamp where
  clip-lower (IntegerStamp b l h) c =
    (if l < c then (IntegerStamp b c h) else (IntegerStamp b l c)) |
  clip-lower s c = s  
  
fun max-lower :: Stamp ⇒ Stamp ⇒ Stamp where
  max-lower (IntegerStamp b1 xl xh) (IntegerStamp b2 yl yh) =
    (IntegerStamp b1 (max xl yl) xh) |
  max-lower xs ys = xs  
  
fun min-higher :: Stamp ⇒ Stamp ⇒ Stamp where
  min-higher (IntegerStamp b1 xl xh) (IntegerStamp b2 yl yh) =
    (IntegerStamp b1 yl (min xh yh)) |
  min-higher xs ys = ys  
  
fun registerNewCondition :: IRGraph ⇒ IRNode ⇒ (ID ⇒ Stamp) ⇒ (ID ⇒ Stamp) where
  — constrain equality by joining the stamps
  registerNewCondition g (IntegerEqualsNode x y) stamps =
    (stamps
```

```

(x := join (stamps x) (stamps y)))
(y := join (stamps x) (stamps y)) |
— constrain less than by removing overlapping stamps
registerNewCondition g (IntegerLessThanNode x y) stamps =
(stamps
(x := clip-upper (stamps x) ((stpi-lower (stamps y)) - 1)))
(y := clip-lower (stamps y) ((stpi-upper (stamps x)) + 1)) |
registerNewCondition g (LogicNegationNode c) stamps =
(case (kind g c) of
(IntegerLessThanNode x y) =>
(stamps
(x := max-lower (stamps x) (stamps y)))
(y := min-higher (stamps x) (stamps y))
|-> stamps) |
registerNewCondition g - stamps = stamps

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

```

type-synonym *DominatorCache* = (*ID*, *ID set*) map

inductive

dominators-all :: *IRGraph* \Rightarrow *DominatorCache* \Rightarrow *ID* \Rightarrow *ID set set* \Rightarrow *ID list* \Rightarrow *DominatorCache* \Rightarrow *ID set set* \Rightarrow *ID list* \Rightarrow *bool* **and**
dominators :: *IRGraph* \Rightarrow *DominatorCache* \Rightarrow *ID* \Rightarrow (*ID set* \times *DominatorCache*)
 \Rightarrow *bool* **where**

```

[pre = []]
    ==> dominators-all g c nid doms pre c doms pre |

[pre = pr # xs;
 dominators g c pr (doms', c');
 dominators-all g c' pr (doms  $\cup$  {doms'}) xs c'' doms'' pre']
    ==> dominators-all g c nid doms pre c'' doms'' pre' |

[preds g nid = []]
    ==> dominators g c nid ({nid}, c) |

[c nid = None;
 preds g nid = x # xs;
 dominators-all g c nid {} (preds g nid) c' doms pre';
 c'' = c'(nid  $\mapsto$  ({nid}  $\cup$  (doms)))]
    ==> dominators g c nid (((nid)  $\cup$  (doms)), c'') |

[c nid = Some doms]
    ==> dominators g c nid (doms, c)

```

— Trying to simplify by removing the 3rd case won't work. A base case for root nodes is required as $\bigcap \emptyset = \text{coset } []$ which swallows anything unioned with it.

value $\bigcap(\{\}::\text{nat set set})$

value $-\bigcap(\{\}::\text{nat set set})$

value $\bigcap(\{\{\}, \{0\}\}::\text{nat set set})$

value $\{0::\text{nat}\} \cup (\bigcap\{\})$

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$) *dominators-all*
 $\langle \text{proof} \rangle$

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

```
definition ConditionalEliminationTest13-testSnippet2-initial :: IRGraph where
  ConditionalEliminationTest13-testSnippet2-initial = irgraph [
    (0, (StartNode (Some 2) 8), VoidStamp),
    (1, (ParameterNode 0), IntegerStamp 32 (-2147483648) (2147483647)),
    (2, (FrameState [] None None None), IllegalStamp),
    (3, (ConstantNode (new-int 32 (0))), IntegerStamp 32 (0) (0)),
    (4, (ConstantNode (new-int 32 (1))), IntegerStamp 32 (1) (1)),
    (5, (IntegerLessThanNode 1 4), VoidStamp),
    (6, (BeginNode 13), VoidStamp),
    (7, (BeginNode 23), VoidStamp),
    (8, (IfNode 5 7 6), VoidStamp),
    (9, (ConstantNode (new-int 32 (-1))), IntegerStamp 32 (-1) (-1)),
    (10, (IntegerEqualsNode 1 9), VoidStamp),
    (11, (BeginNode 17), VoidStamp),
    (12, (BeginNode 15), VoidStamp),
    (13, (IfNode 10 12 11), VoidStamp),
    (14, (ConstantNode (new-int 32 (-2))), IntegerStamp 32 (-2) (-2)),
    (15, (StoreFieldNode 15 "org.graalvm.compiler.core.test.ConditionalEliminationTestBase::sink2"),
    (14 (Some 16) None 19), VoidStamp),
    (16, (FrameState [] None None None), IllegalStamp),
    (17, (EndNode), VoidStamp),
    (18, (MergeNode [17, 19] (Some 20) 21), VoidStamp),
    (19, (EndNode), VoidStamp),
    (20, (FrameState [] None None None), IllegalStamp),
    (21, (StoreFieldNode 21 "org.graalvm.compiler.core.test.ConditionalEliminationTestBase::sink1"),
    (3 (Some 22) None 25), VoidStamp),
    (22, (FrameState [] None None None), IllegalStamp),
    (23, (EndNode), VoidStamp),
    (24, (MergeNode [23, 25] (Some 26) 27), VoidStamp),
    (25, (EndNode), VoidStamp),
    (26, (FrameState [] None None None), IllegalStamp),
    (27, (StoreFieldNode 27 "org.graalvm.compiler.core.test.ConditionalEliminationTestBase::sink0"),
    (9 (Some 28) None 29), VoidStamp),
    (28, (FrameState [] None None None), IllegalStamp),
    (29, (ReturnNode None None), VoidStamp)
  ]
```

```
values {(snd x) 13| x. dominators ConditionalEliminationTest13-testSnippet2-initial
Map.empty 25 x}
```

inductive

```
condition-of :: IRGraph ⇒ ID ⇒ (IRExpr × IRNode) option ⇒ bool where
[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)]]
⇒ condition-of g nid (Some (ce', c)) |
```

```
[pred g nid = None] ⇒ condition-of g nid None |
[pred g nid = Some nid';
¬(is-IfNode (kind g nid'))] ⇒ condition-of g nid None
```

```
code-pred (modes: i ⇒ i ⇒ o ⇒ bool) condition-of ⟨proof⟩
```

```
fun conditions-of-dominators :: IRGraph ⇒ ID list ⇒ Conditions ⇒ Conditions
where
```

```
conditions-of-dominators g [] cds = cds |
conditions-of-dominators g (nid # nids) cds =
(case (Predicate.the (condition-of-i-i-o g nid)) of
None ⇒ conditions-of-dominators g nids cds |
Some (expr, -) ⇒ conditions-of-dominators g nids (expr # cds))
```

```
fun stamps-of-dominators :: IRGraph ⇒ ID list ⇒ StampFlow ⇒ StampFlow
where
```

```
stamps-of-dominators g [] stamps = stamps |
stamps-of-dominators g (nid # nids) stamps =
(case (Predicate.the (condition-of-i-i-o g nid)) of
None ⇒ stamps-of-dominators g nids stamps |
Some (-, node) ⇒ stamps-of-dominators g nids
((registerNewCondition g node (hd stamps)) # stamps))
```

```

inductive
  analyse :: IRGraph  $\Rightarrow$  DominatorCache  $\Rightarrow$  ID  $\Rightarrow$  (Conditions  $\times$  StampFlow  $\times$ 
  DominatorCache)  $\Rightarrow$  bool where
     $\llbracket \text{dominators } g c \text{ nid } (\text{doms}, c') ;$ 
     $\text{conditions-of-dominators } g (\text{sorted-list-of-set doms}) [] = \text{conds};$ 
     $\text{stamps-of-dominators } g (\text{sorted-list-of-set doms}) [\text{stamp } g] = \text{stamps} \rrbracket$ 
     $\implies \text{analyse } g c \text{ nid } (\text{conds}, \text{stamps}, c')$ 

code-pred (modes:  $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ ) analyse  $\langle \text{proof} \rangle$ 

values { $x$ . dominators ConditionalEliminationTest13-testSnippet2-initial Map.empty
13  $x$ }
values {( $\text{conds}$ ,  $\text{stamps}$ ,  $c$ )}.
analyse ConditionalEliminationTest13-testSnippet2-initial Map.empty 13 ( $\text{conds}$ ,
 $\text{stamps}$ ,  $c$ )
values {( $\text{hd stamps}$ ) 1 |  $\text{conds stamps } c$  .}
analyse ConditionalEliminationTest13-testSnippet2-initial Map.empty 13 ( $\text{conds}$ ,
 $\text{stamps}$ ,  $c$ )
values {( $\text{hd stamps}$ ) 1 |  $\text{conds stamps } c$  .}
analyse ConditionalEliminationTest13-testSnippet2-initial Map.empty 27 ( $\text{conds}$ ,
 $\text{stamps}$ ,  $c$ )

fun next-nid :: IRGraph  $\Rightarrow$  ID set  $\Rightarrow$  ID  $\Rightarrow$  ID option where
  next-nid  $g$  seen nid = (case (kind  $g$  nid) of
    (EndNode)  $\Rightarrow$  Some (any-usage  $g$  nid) |
    -  $\Rightarrow$  nextEdge seen nid  $g$ )

inductive Step
  :: IRGraph  $\Rightarrow$  (ID  $\times$  Seen)  $\Rightarrow$  (ID  $\times$  Seen) option  $\Rightarrow$  bool
  for  $g$  where
  — We can find a successor edge that is not in seen, go there
   $\llbracket \text{seen}' = \{\text{nid}\} \cup \text{seen};$ 
  Some nid' = next-nid  $g$  seen' nid;
  nid'  $\notin$  seen  $\rrbracket$ 
   $\implies \text{Step } g (\text{nid}, \text{seen}) (\text{Some } (\text{nid}', \text{seen}')) |$ 

  — We can cannot find a successor edge that is not in seen, give back None
   $\llbracket \text{seen}' = \{\text{nid}\} \cup \text{seen};$ 
  None = next-nid  $g$  seen' nid
   $\implies \text{Step } g (\text{nid}, \text{seen}) \text{ None} |$ 

  — We've already seen this node, give back None
   $\llbracket \text{seen}' = \{\text{nid}\} \cup \text{seen};$ 
  Some nid' = next-nid  $g$  seen' nid;
  nid'  $\in$  seen  $\rrbracket$ 
   $\implies \text{Step } g (\text{nid}, \text{seen}) \text{ None}$ 

```

```

code-pred (modes:  $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ )  $\text{Step} \langle \text{proof} \rangle$ 

fun  $\text{nextNode} :: \text{IRGraph} \Rightarrow \text{Seen} \Rightarrow (\text{ID} \times \text{Seen}) \text{ option}$  where
   $\text{nextNode } g \text{ seen} =$ 
     $(\text{let } \text{toSee} = \text{sorted-list-of-set } \{n \in \text{ids } g. n \notin \text{seen}\} \text{ in}$ 
       $\text{case } \text{toSee} \text{ of } [] \Rightarrow \text{None} \mid (x \# xs) \Rightarrow \text{Some } (x, \text{seen} \cup \{x\}))$ 

values  $\{x. \text{Step ConditionalEliminationTest13-testSnippet2-initial} (17, \{17, 11, 25, 21, 18, 19, 15, 12, 13, 6, 29, 27, x\})$ 

```

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  $\text{ConditionalEliminationPhase}$ 
   $:: (\text{Seen} \times \text{DominatorCache}) \Rightarrow \text{IRGraph} \Rightarrow \text{IRGraph} \Rightarrow \text{bool}$ 
  where

```

- Can do a step and optimise for the current node

 $\llbracket \text{nextNode } g \text{ seen} = \text{Some } (nid, \text{seen}') ;$
 $\text{analyse } g \text{ c } nid \text{ (conds, flow, c');}$
 $\text{ConditionalEliminationStep (set conds) (hd flow) } nid \text{ } g \text{ } g' ;$
 $\text{ConditionalEliminationPhase (seen', c') } g' \text{ } g' \rrbracket$
 $\implies \text{ConditionalEliminationPhase (seen, c) } g \text{ } g'' |$
 $\llbracket \text{nextNode } g \text{ seen} = \text{None} \rrbracket$
 $\implies \text{ConditionalEliminationPhase (seen, c) } g \text{ } g$

```
code-pred (modes:  $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ )  $\text{ConditionalEliminationPhase} \langle \text{proof} \rangle$ 
```

```
definition  $\text{runConditionalElimination} :: \text{IRGraph} \Rightarrow \text{IRGraph}$  where
```

 $\text{runConditionalElimination } g =$
 $(\text{Predicate.the } (\text{ConditionalEliminationPhase-i-i-o} (\{\}, \text{Map.empty}) \text{ } g))$

```
values  $\{(doms, c')| doms \text{ } c'\}$ 
```

dominators ConditionalEliminationTest13-testSnippet2-initial Map.empty 6 (doms, c')

```
values  $\{(conds, stamps, c)| conds \text{ } stamps \text{ } c\}$ 
```

analyse ConditionalEliminationTest13-testSnippet2-initial Map.empty 6 (conds, stamps, c)

```
value
```

 $(\text{nextNode}$

ConditionalEliminationTest13-testSnippet2-initial {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1028, 1029, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1059, 1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1059, 1060, 1061, 1062, 1063, 1064, 1065, 1066, 1067, 1068, 1069, 1060, 1061, 1062, 1063, 1064, 1065, 1066, 1067, 1068, 1069, 1070, 1071, 1072, 1073, 1074, 1075, 1076, 1077, 1078, 1079, 1070, 1071, 1072, 1073, 1074, 1075, 1076, 1077, 1078, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1087, 1088, 1089, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1087, 1088, 1089, 1090, 1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1090, 1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1100, 1101, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1100, 1101, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114, 1115, 1116, 1117, 1118, 1119, 1110, 1111, 1112, 1113, 1114, 1115, 1116, 1117, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128, 1129, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128, 1129, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1140, 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1150, 1151, 1152, 1153, 1154, 1155, 1156, 1157, 1158, 1159, 1150, 1151, 1152, 1153, 1154, 1155, 1156, 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164, 1165, 1166, 1167, 1168, 1169, 1160, 1161, 1162, 1163, 1164, 1165, 1166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1177, 1178, 1179, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1177, 1178, 1179, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187, 1188, 1189, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187, 1188, 1189, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1200, 1201, 1202, 1203, 1204, 1205, 1206, 1207, 1208, 1209, 1200, 1201, 1202, 1203, 1204, 1205, 1206, 1207, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1238, 1239, 1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1238, 1239, 1240, 1241, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1249, 1240, 1241, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1249, 1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1260, 1261, 1262, 1263, 1264, 1265, 1266, 1267, 1268, 1269, 1260, 1261, 1262, 1263, 1264, 1265, 1266, 1267, 1268, 1269, 1270, 1271, 1272, 1273, 1274, 1275, 1276, 1277, 1278, 1279, 1270, 1271, 1272, 1273, 1274, 1275, 1276, 1277, 1278, 1279, 1280, 1281, 1282, 1283, 1284, 1285, 1286, 1287, 1288, 1289, 1280, 1281, 1282, 1283, 1284, 1285, 1286, 1287, 1288, 1289, 1290, 1291, 1

```

lemma IfNodeStepE:  $g, p \vdash (nid, m, h) \rightarrow (nid', m', h) \implies$ 
 $(\wedge cond tb fb val.$ 
 $kind g nid = IfNode cond tb fb \implies$ 
 $nid' = (if val\text{-}to\text{-}bool val then tb else fb) \implies$ 
 $[g, m, p] \vdash cond \mapsto val \implies m' = m)$ 
 $\langle proof \rangle$ 

lemma ifNodeHasCondEvalStutter:
assumes  $(g m p h \vdash nid \rightsquigarrow nid')$ 
assumes  $kind g nid = IfNode cond t f$ 
shows  $\exists v. ([g, m, p] \vdash cond \mapsto v)$ 
 $\langle proof \rangle$ 

lemma ifNodeHasCondEval:
assumes  $(g, p \vdash (nid, m, h) \rightarrow (nid', m', h'))$ 
assumes  $kind g nid = IfNode cond t f$ 
shows  $\exists v. ([g, m, p] \vdash cond \mapsto v)$ 
 $\langle proof \rangle$ 

lemma replace-if-t:
assumes  $kind g nid = IfNode cond tb fb$ 
assumes  $[g, m, p] \vdash cond \mapsto \text{bool}$ 
assumes  $\text{val-to-bool bool}$ 
assumes  $g': g' = \text{replace-usages } nid \text{ tb } g$ 
shows  $\exists nid'. (g m p h \vdash nid \rightsquigarrow nid') \longleftrightarrow (g' m p h \vdash nid \rightsquigarrow nid')$ 
 $\langle proof \rangle$ 

lemma replace-if-t-imp:
assumes  $kind g nid = IfNode cond tb fb$ 
assumes  $[g, m, p] \vdash cond \mapsto \text{bool}$ 
assumes  $\text{val-to-bool bool}$ 
assumes  $g': g' = \text{replace-usages } nid \text{ tb } g$ 
shows  $\exists nid'. (g m p h \vdash nid \rightsquigarrow nid') \longrightarrow (g' m p h \vdash nid \rightsquigarrow nid')$ 
 $\langle proof \rangle$ 

lemma replace-if-f:
assumes  $kind g nid = IfNode cond tb fb$ 
assumes  $[g, m, p] \vdash cond \mapsto \text{bool}$ 
assumes  $\neg(\text{val-to-bool bool})$ 
assumes  $g': g' = \text{replace-usages } nid \text{ fb } g$ 
shows  $\exists nid'. (g m p h \vdash nid \rightsquigarrow nid') \longleftrightarrow (g' m p h \vdash nid \rightsquigarrow nid')$ 
 $\langle proof \rangle$ 

```

Prove that the individual conditional elimination rules are correct with respect to preservation of stuttering steps.

lemma ConditionalEliminationStepProof:

```

assumes wg: wf-graph g
assumes ws: wf-stamps g
assumes wv: wf-values g
assumes nid: nid ∈ ids g
assumes conds-valid: ∀ c ∈ conds . ∃ v. ([m, p] ⊢ c ↦ v) ∧ val-to-bool v
assumes ce: ConditionalEliminationStep conds stamps nid g g'

shows ∃ nid' .(g m p h ⊢ nid ~> nid') —→ (g' m p h ⊢ nid ~> nid')
⟨proof⟩

```

Prove that the individual conditional elimination rules are correct with respect to finding a bisimulation between the unoptimized and optimized graphs.

```

lemma ConditionalEliminationStepProofBisimulation:
assumes wf: wf-graph g ∧ wf-stamp g stamps ∧ wf-values g
assumes nid: nid ∈ ids g
assumes conds-valid: ∀ c ∈ conds . ∃ v. ([m, p] ⊢ c ↦ v) ∧ val-to-bool v
assumes ce: ConditionalEliminationStep conds stamps nid g g'
assumes gstep: ∃ h nid'. (g, p ⊢ (nid, m, h) → (nid', m, h))

shows nid | g ~ g'
⟨proof⟩

```

experiment begin

```

lemma inverse-succ:
∀ n' ∈ (succ g n). n ∈ ids g —→ n ∈ (predecessors g n')
⟨proof⟩

```

```

lemma sequential-successors:
assumes is-sequential-node n
shows successors-of n ≠ []
⟨proof⟩

```

```

lemma nid'-succ:
assumes nid ∈ ids g
assumes ¬(is-AbstractEndNode (kind g nid0))
assumes g, p ⊢ (nid0, m0, h0) → (nid, m, h)
shows nid ∈ succ g nid0
⟨proof⟩

```

```

lemma nid'-pred:
assumes nid ∈ ids g
assumes ¬(is-AbstractEndNode (kind g nid0))
assumes g, p ⊢ (nid0, m0, h0) → (nid, m, h)
shows nid0 ∈ predecessors g nid

```

```

⟨proof⟩

definition wf-pred:
  wf-pred g = (forall n in ids g. card (predecessors g n) = 1)

lemma
  assumes not(is-AbstractMergeNode (kind g n'))
  assumes wf-pred g
  shows exists v. predecessors g n = {v} ∧ pred g n' = Some v
  ⟨proof⟩

lemma inverse-succ1:
  assumes not(is-AbstractEndNode (kind g n'))
  assumes wf-pred g
  shows forall n' in (succ g n). n in ids g → Some n = (pred g n')
  ⟨proof⟩

lemma BeginNodeFlow:
  assumes g, p ⊢ (nid0, m0, h0) → (nid, m, h)
  assumes Some ifcond = pred g nid
  assumes kind g ifcond = IfNode cond t f
  assumes i = find-index nid (successors-of (kind g ifcond))
  shows i = 0 ↔ ([g, m, p] ⊢ cond ↦ v) ∧ val-to-bool v
  ⟨proof⟩

end

end
theory CFG
  imports Graph.IRGraph
begin

datatype Block =
  BasicBlock (start-node: ID) (end-node: ID) |
  NoBlock

function findEnd :: IRGraph ⇒ ID ⇒ ID list ⇒ ID where
  findEnd g nid [next] = findEnd g next (successors-of (kind g next)) |
  findEnd g nid succs = nid
  ⟨proof⟩ termination ⟨proof⟩

function findStart :: IRGraph ⇒ ID ⇒ ID list ⇒ ID where
  findStart g nid [pred] =

```

```

(if is-AbstractBeginNode (kind g nid) then
  nid
else
  (findStart g pred (sorted-list-of-set (predecessors g nid)))) | 
findStart g nid preds = nid
⟨proof⟩ termination ⟨proof⟩

fun blockOf :: IRGraph ⇒ ID ⇒ Block where
blockOf g nid = (
  let end = (findEnd g nid (sorted-list-of-set (succ g nid))) in
  let start = (findStart g nid (sorted-list-of-set (predecessors g nid))) in
  if (start = end ∧ start = nid) then NoBlock else
    BasicBlock start end
)

fun succ-from-end :: IRGraph ⇒ ID ⇒ IRNode ⇒ Block set where
succ-from-end g e EndNode = {blockOf g (any-usage g e)} |
succ-from-end g e (IfNode c tb fb) = {blockOf g tb, blockOf g fb} |
succ-from-end g e (LoopEndNode begin) = {blockOf g begin} |
succ-from-end g e - = (if (is-AbstractEndNode (kind g e))
  then (set (map (blockOf g) (successors-of (kind g e))))
  else {})

fun succ :: IRGraph ⇒ Block ⇒ Block set where
succ g (BasicBlock start end) = succ-from-end g end (kind g end) |
succ g - = {}

fun register-by-pred :: IRGraph ⇒ ID ⇒ Block option where
register-by-pred g nid = (
  case kind g (end-node (blockOf g nid)) of
    (IfNode c tb fb) ⇒ Some (blockOf g nid) |
    k ⇒ (if (is-AbstractEndNode k) then Some (blockOf g nid) else None)
)

fun pred-from-start :: IRGraph ⇒ ID ⇒ IRNode ⇒ Block set where
pred-from-start g s (MergeNode ends - -) = set (map (blockOf g) ends) |
pred-from-start g s (LoopBeginNode ends - - -) = set (map (blockOf g) ends) |
pred-from-start g s (LoopEndNode begin) = {blockOf g begin} |
pred-from-start g s - = set (List.map-filter (register-by-pred g) (sorted-list-of-set
  (predecessors g s)))

fun pred :: IRGraph ⇒ Block ⇒ Block set where
pred g (BasicBlock start end) = pred-from-start g start (kind g start) |
pred g - = {}

inductive dominates :: IRGraph ⇒ Block ⇒ Block ⇒ bool (- ⊢ - ≥ ≥ - 20) where
  [(d = n) ∨ ((pred g n ≠ {}) ∧ (∀ p ∈ pred g n . (g ⊢ d ≥ ≥ p)))] ⇒ dominates
  g d n
code-pred [show-modes] dominates ⟨proof⟩

```

```

inductive postdominates :: IRGraph  $\Rightarrow$  Block  $\Rightarrow$  Block  $\Rightarrow$  bool (-  $\vdash$  -  $\leq\leq$  - 20)
where
   $\llbracket (z = n) \vee ((\text{succ } g\ n \neq \{\}) \wedge (\forall s \in \text{succ } g\ n . (g \vdash z \leq\leq s))) \rrbracket \implies \text{postdominates}$ 
   $g\ z\ n$ 
code-pred [show-modes] postdominates ⟨proof⟩

inductive strictly-dominates :: IRGraph  $\Rightarrow$  Block  $\Rightarrow$  Block  $\Rightarrow$  bool (-  $\vdash$  -  $>>$  - 20) where
   $\llbracket (g \vdash d \geq n); (d \neq n) \rrbracket \implies \text{strictly-dominates } g\ d\ n$ 
code-pred [show-modes] strictly-dominates ⟨proof⟩

inductive strictly-postdominates :: IRGraph  $\Rightarrow$  Block  $\Rightarrow$  Block  $\Rightarrow$  bool (-  $\vdash$  - << - 20) where
   $\llbracket (g \vdash d \leq n); (d \neq n) \rrbracket \implies \text{strictly-postdominates } g\ d\ n$ 
code-pred [show-modes] strictly-postdominates ⟨proof⟩

lemma pred g nid = {}  $\longrightarrow \neg(\exists d . (d \neq \text{nid}) \wedge (g \vdash d \geq n))$ 
  ⟨proof⟩

lemma succ g nid = {}  $\longrightarrow \neg(\exists d . (d \neq \text{nid}) \wedge (g \vdash d \leq n))$ 
  ⟨proof⟩

lemma pred g nid = {}  $\longrightarrow \neg(\exists d . (g \vdash d >> \text{nid}))$ 
  ⟨proof⟩

lemma succ g nid = {}  $\longrightarrow \neg(\exists d . (g \vdash d << \text{nid}))$ 
  ⟨proof⟩

inductive wf-cfg :: IRGraph  $\Rightarrow$  bool where
   $\forall nid \in \text{ids } g . (\text{blockOf } g\ nid \neq \text{NoBlock}) \longrightarrow (g \vdash (\text{blockOf } g\ 0) \geq (\text{blockOf } g\ nid))$ 
   $\implies \text{wf-cfg } g$ 
code-pred [show-modes] wf-cfg ⟨proof⟩

inductive immediately-dominates :: IRGraph  $\Rightarrow$  Block  $\Rightarrow$  Block  $\Rightarrow$  bool (-  $\vdash$  - idom - 20) where
   $\llbracket (g \vdash d >> n); (\forall w \in \text{ids } g . (g \vdash (\text{blockOf } g\ w) >> n) \longrightarrow (g \vdash (\text{blockOf } g\ w) \geq d)) \rrbracket \implies \text{immediately-dominates } g\ d\ n$ 
code-pred [show-modes] immediately-dominates ⟨proof⟩

definition simple-if :: IRGraph where
  simple-if = irgraph [
    (0, StartNode None 2, VoidStamp),
    (1, ParameterNode 0, default-stamp),
    (2, IfNode 1 3 4, VoidStamp),
    (3, BeginNode 5, VoidStamp),
    (4, BeginNode 6, VoidStamp),
    (5, EndNode, VoidStamp),
  ]

```

```

(6, EndNode, VoidStamp),
(7, ParameterNode 1, default-stamp),
(8, ParameterNode 2, default-stamp),
(9, AddNode 7 8, default-stamp),
(10, MergeNode [5,6] None 12, VoidStamp),
(11, ValuePhiNode 11 [9,7] 10, default-stamp),
(12, ReturnNode (Some 11) None, default-stamp)
]

```

value wf-cfg simple-if

```

value simple-if  $\vdash$  blockOf simple-if 0  $\geq\geq$  blockOf simple-if 0
value simple-if  $\vdash$  blockOf simple-if 0  $\geq\geq$  blockOf simple-if 3
value simple-if  $\vdash$  blockOf simple-if 0  $\geq\geq$  blockOf simple-if 4
value simple-if  $\vdash$  blockOf simple-if 0  $\geq\geq$  blockOf simple-if 12

```

```

value simple-if  $\vdash$  blockOf simple-if 3  $\geq\geq$  blockOf simple-if 0
value simple-if  $\vdash$  blockOf simple-if 3  $\geq\geq$  blockOf simple-if 3
value simple-if  $\vdash$  blockOf simple-if 3  $\geq\geq$  blockOf simple-if 4
value simple-if  $\vdash$  blockOf simple-if 3  $\geq\geq$  blockOf simple-if 12

```

```

value simple-if  $\vdash$  blockOf simple-if 4  $\geq\geq$  blockOf simple-if 0
value simple-if  $\vdash$  blockOf simple-if 4  $\geq\geq$  blockOf simple-if 3
value simple-if  $\vdash$  blockOf simple-if 4  $\geq\geq$  blockOf simple-if 4
value simple-if  $\vdash$  blockOf simple-if 4  $\geq\geq$  blockOf simple-if 12

```

```

value simple-if  $\vdash$  blockOf simple-if 12  $\geq\geq$  blockOf simple-if 0
value simple-if  $\vdash$  blockOf simple-if 12  $\geq\geq$  blockOf simple-if 3
value simple-if  $\vdash$  blockOf simple-if 12  $\geq\geq$  blockOf simple-if 4
value simple-if  $\vdash$  blockOf simple-if 12  $\geq\geq$  blockOf simple-if 12

```

```

value simple-if  $\vdash$  blockOf simple-if 0  $\leq\leq$  blockOf simple-if 0
value simple-if  $\vdash$  blockOf simple-if 0  $\leq\leq$  blockOf simple-if 3
value simple-if  $\vdash$  blockOf simple-if 0  $\leq\leq$  blockOf simple-if 4
value simple-if  $\vdash$  blockOf simple-if 0  $\leq\leq$  blockOf simple-if 12

```

```

value simple-if  $\vdash$  blockOf simple-if 3  $\leq\leq$  blockOf simple-if 0
value simple-if  $\vdash$  blockOf simple-if 3  $\leq\leq$  blockOf simple-if 3
value simple-if  $\vdash$  blockOf simple-if 3  $\leq\leq$  blockOf simple-if 4

```

```
value simple-if  $\vdash$  blockOf simple-if 3  $\leq\leq$  blockOf simple-if 12
```

```
value simple-if  $\vdash$  blockOf simple-if 4  $\leq\leq$  blockOf simple-if 0  
value simple-if  $\vdash$  blockOf simple-if 4  $\leq\leq$  blockOf simple-if 3  
value simple-if  $\vdash$  blockOf simple-if 4  $\leq\leq$  blockOf simple-if 4  
value simple-if  $\vdash$  blockOf simple-if 4  $\leq\leq$  blockOf simple-if 12
```

```
value simple-if  $\vdash$  blockOf simple-if 12  $\leq\leq$  blockOf simple-if 0  
value simple-if  $\vdash$  blockOf simple-if 12  $\leq\leq$  blockOf simple-if 3  
value simple-if  $\vdash$  blockOf simple-if 12  $\leq\leq$  blockOf simple-if 4  
value simple-if  $\vdash$  blockOf simple-if 12  $\leq\leq$  blockOf simple-if 12
```

```
value blockOf simple-if 0  
value blockOf simple-if 1  
value blockOf simple-if 2  
value blockOf simple-if 3  
value blockOf simple-if 4  
value blockOf simple-if 5  
value blockOf simple-if 6  
value blockOf simple-if 7  
value blockOf simple-if 8  
value blockOf simple-if 9  
value blockOf simple-if 10  
value blockOf simple-if 11  
value blockOf simple-if 12
```

```
value pred simple-if (blockOf simple-if 0)  
value succ simple-if (blockOf simple-if 0)
```

```
value pred simple-if (blockOf simple-if 3)  
value succ simple-if (blockOf simple-if 3)
```

```
value pred simple-if (blockOf simple-if 4)  
value succ simple-if (blockOf simple-if 4)
```

```
value pred simple-if (blockOf simple-if 10)  
value succ simple-if (blockOf simple-if 10)
```

```
definition ConditionalEliminationTest1-test1Snippet-initial :: IRGraph where  
  ConditionalEliminationTest1-test1Snippet-initial = irgraph [  
    (0, (StartNode (Some 2) 7), VoidStamp),  
    (1, (ParameterNode 0), IntegerStamp 32 (-2147483648) (2147483647)),  
    (2, (FrameState [] None None None), IllegalStamp),  
    (3, (ConstantNode (IntVal 32 (0))), IntegerStamp 32 (0) (0)),
```

```

(4, (IntegerEqualsNode 1 3), VoidStamp),
(5, (BeginNode 39), VoidStamp),
(6, (BeginNode 12), VoidStamp),
(7, (IfNode 4 6 5), VoidStamp),
(8, (ConstantNode (IntVal 32 (5))), IntegerStamp 32 (5) (5)),
(9, (IntegerEqualsNode 1 8), VoidStamp),
(10, (BeginNode 16), VoidStamp),
(11, (BeginNode 14), VoidStamp),
(12, (IfNode 9 11 10), VoidStamp),
(13, (ConstantNode (IntVal 32 (100))), IntegerStamp 32 (100) (100)),
(14, (StoreFieldNode 14 "org.graalvm.compiler.core.test.ConditionalEliminationTestBase::sink2"))
13 (Some 15) None 18), VoidStamp),
(15, (FrameState [] None None None), IllegalStamp),
(16, (EndNode), VoidStamp),
(17, (MergeNode [16, 18] (Some 19) 24), VoidStamp),
(18, (EndNode), VoidStamp),
(19, (FrameState [] None None None), IllegalStamp),
(20, (ConstantNode (IntVal 32 (101))), IntegerStamp 32 (101) (101)),
(21, (IntegerLessThanNode 1 20), VoidStamp),
(22, (BeginNode 30), VoidStamp),
(23, (BeginNode 25), VoidStamp),
(24, (IfNode 21 23 22), VoidStamp),
(25, (EndNode), VoidStamp),
(26, (MergeNode [25, 27, 34] (Some 35) 43), VoidStamp),
(27, (EndNode), VoidStamp),
(28, (BeginNode 32), VoidStamp),
(29, (BeginNode 27), VoidStamp),
(30, (IfNode 4 28 29), VoidStamp),
(31, (ConstantNode (IntVal 32 (200))), IntegerStamp 32 (200) (200)),
(32, (StoreFieldNode 32 "org.graalvm.compiler.core.test.ConditionalEliminationTest1::sink3"))
31 (Some 33) None 34), VoidStamp),
(33, (FrameState [] None None None), IllegalStamp),
(34, (EndNode), VoidStamp),
(35, (FrameState [] None None None), IllegalStamp),
(36, (ConstantNode (IntVal 32 (2))), IntegerStamp 32 (2) (2)),
(37, (IntegerEqualsNode 1 36), VoidStamp),
(38, (BeginNode 45), VoidStamp),
(39, (EndNode), VoidStamp),
(40, (MergeNode [39, 41, 47] (Some 48) 49), VoidStamp),
(41, (EndNode), VoidStamp),
(42, (BeginNode 41), VoidStamp),
(43, (IfNode 37 42 38), VoidStamp),
(44, (ConstantNode (IntVal 32 (1))), IntegerStamp 32 (1) (1)),
(45, (StoreFieldNode 45 "org.graalvm.compiler.core.test.ConditionalEliminationTestBase::sink1"))
44 (Some 46) None 47), VoidStamp),
(46, (FrameState [] None None None), IllegalStamp),
(47, (EndNode), VoidStamp),
(48, (FrameState [] None None None), IllegalStamp),
(49, (StoreFieldNode 49 "org.graalvm.compiler.core.test.ConditionalEliminationTestBase::sink0"))

```

```

3 (Some 50) None 51), VoidStamp),
(50, (FrameState [] None None None), IllegalStamp),
(51, (ReturnNode None None), VoidStamp)
]

value blockOf ConditionalEliminationTest1-test1Snippet-initial 0
value blockOf ConditionalEliminationTest1-test1Snippet-initial 7

value blockOf ConditionalEliminationTest1-test1Snippet-initial 6
value blockOf ConditionalEliminationTest1-test1Snippet-initial 12

value blockOf ConditionalEliminationTest1-test1Snippet-initial 11
value blockOf ConditionalEliminationTest1-test1Snippet-initial 14
value blockOf ConditionalEliminationTest1-test1Snippet-initial 18

value blockOf ConditionalEliminationTest1-test1Snippet-initial 10
value blockOf ConditionalEliminationTest1-test1Snippet-initial 16

value blockOf ConditionalEliminationTest1-test1Snippet-initial 17
value blockOf ConditionalEliminationTest1-test1Snippet-initial 24

value blockOf ConditionalEliminationTest1-test1Snippet-initial 23
value blockOf ConditionalEliminationTest1-test1Snippet-initial 25

value blockOf ConditionalEliminationTest1-test1Snippet-initial 22
value blockOf ConditionalEliminationTest1-test1Snippet-initial 30

value blockOf ConditionalEliminationTest1-test1Snippet-initial 28
value blockOf ConditionalEliminationTest1-test1Snippet-initial 32
value blockOf ConditionalEliminationTest1-test1Snippet-initial 34

value blockOf ConditionalEliminationTest1-test1Snippet-initial 29
value blockOf ConditionalEliminationTest1-test1Snippet-initial 27

value blockOf ConditionalEliminationTest1-test1Snippet-initial 26
value blockOf ConditionalEliminationTest1-test1Snippet-initial 43

value blockOf ConditionalEliminationTest1-test1Snippet-initial 42
value blockOf ConditionalEliminationTest1-test1Snippet-initial 41

value blockOf ConditionalEliminationTest1-test1Snippet-initial 38
value blockOf ConditionalEliminationTest1-test1Snippet-initial 45
value blockOf ConditionalEliminationTest1-test1Snippet-initial 47

value blockOf ConditionalEliminationTest1-test1Snippet-initial 5
value blockOf ConditionalEliminationTest1-test1Snippet-initial 39

value blockOf ConditionalEliminationTest1-test1Snippet-initial 40
value blockOf ConditionalEliminationTest1-test1Snippet-initial 49

```

value *blockOf ConditionalEliminationTest1-test1Snippet-initial* 51

```
value pred ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 0)  
value succ ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 0)
```

```
value pred ConditionalEliminationTest1-test1Snippet-initial  
        (blockOf ConditionalEliminationTest1-test1Snippet-initial 6)  
value succ ConditionalEliminationTest1-test1Snippet-initial  
        (blockOf ConditionalEliminationTest1-test1Snippet-initial 6)
```

```
value pred ConditionalEliminationTest1-test1Snippet-initial  
      (blockOf ConditionalEliminationTest1-test1Snippet-initial 14)  
value succ ConditionalEliminationTest1-test1Snippet-initial  
      (blockOf ConditionalEliminationTest1-test1Snippet-initial 14)
```

```
value pred ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 10)  
value succ ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 10)
```

```
value pred ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 24)  
value succ ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 24)
```

```
value pred ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 23)  
value succ ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 23)
```

```
value pred ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 22)  
value succ ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 22)
```

```
value pred ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 32)  
value succ ConditionalEliminationTest1-test1Snippet-initial  
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 32)
```

```

value pred ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 29)
value succ ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 29)

value pred ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 43)
value succ ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 43)

value pred ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 42)
value succ ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 42)

value pred ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 45)
value succ ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 45)

value pred ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 5)
value succ ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 5)

value pred ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 49)
value succ ConditionalEliminationTest1-test1Snippet-initial
  (blockOf ConditionalEliminationTest1-test1Snippet-initial 49)

```

end