Veriopt Theories

April 17, 2024

Contents

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 $q_1 \Rrightarrow q_2$ requires that the implication $q_1 \longrightarrow q_2$ is known true (i.e. universally valid). The relation $q_1 \Rrightarrow\neg\ q_2$ requires that the implication $q_1 \longrightarrow q_2$ is known false (i.e. *q1* ⟶ ¬ *q2* is universally valid). If neither $q_1 \Rrightarrow q_2$ nor $q_1 \Rrightarrow\neg\ q_2$ then the status is unknown and the condition cannot be simplified.

fun *implies-valid* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* (**infix** ↣ *50*) **where**
implies-valid q1 q2 =
(∀ *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 =
(∀ *m p v1 v2*. ([*m*, *p*] ⊢ *q1* ↦ *v1*) ∧ ([*m*,*p*] ⊢ *q2* ↦ *v2*) ⟶

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

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

lemma *eq-not-less-val*:
$\mathit{val\text{-}to\text{-}bool}(\mathit{val}[v1\ eq\ v2]) \longrightarrow \neg \mathit{val\text{-}to\text{-}bool}(\mathit{val}[v1 < v2])$
⟨*proof*⟩

lemma *eq-not-less′-val*:
$\mathit{val\text{-}to\text{-}bool}(\mathit{val}[v1\ eq\ v2]) \longrightarrow \neg \mathit{val\text{-}to\text{-}bool}(\mathit{val}[v2 < v1])$
⟨*proof*⟩

lemma *less-not-less-val*:
$\mathit{val\text{-}to\text{-}bool}(\mathit{val}[v1 < v2]) \longrightarrow \neg \mathit{val\text{-}to\text{-}bool}(\mathit{val}[v2 < v1])$
⟨*proof*⟩

lemma *less-not-eq-val*:
$\mathit{val\text{-}to\text{-}bool}(\mathit{val}[v1 < v2]) \longrightarrow \neg \mathit{val\text{-}to\text{-}bool}(\mathit{val}[v1\ eq\ v2])$
⟨*proof*⟩

lemma *logic-negate-type*:
assumes $[m, p] \vdash \mathit{UnaryExpr}\ \mathit{UnaryLogicNegation}\ x \mapsto v$
shows $\exists\, b\ v2.\ [m, p] \vdash x \mapsto \mathit{IntVal}\ b\ v2$
⟨*proof*⟩

lemma *intval-logic-negation-inverse*:
assumes $b > 0$
assumes $x = \mathit{IntVal}\ b\ v$
shows $\mathit{val\text{-}to\text{-}bool}\ (\mathit{intval\text{-}logic\text{-}negation}\ x) \longleftrightarrow \neg(\mathit{val\text{-}to\text{-}bool}\ x)$
⟨*proof*⟩

lemma *logic-negation-relation-tree*:
assumes $[m, p] \vdash y \mapsto \mathit{val}$
assumes $[m, p] \vdash \mathit{UnaryExpr}\ \mathit{UnaryLogicNegation}\ y \mapsto \mathit{invval}$
shows $\mathit{val\text{-}to\text{-}bool}\ \mathit{val} \longleftrightarrow \neg(\mathit{val\text{-}to\text{-}bool}\ \mathit{invval})$
⟨*proof*⟩

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

theorem *implies-impliesnot-valid*:
shows $((q1 \Rrightarrow q2) \longrightarrow (q1 \rightarrowtail q2)) \wedge$
$((q1 \Rrightarrow\neg\ q2) \longrightarrow (q1 \nrightarrowtail q2))$
(**is** $(?imp \longrightarrow ?val) \wedge (?notimp \longrightarrow ?notval))$
⟨*proof*⟩

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
  ⟦*alwaysDistinct* (*stamps x*) (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerEqualsNode x y*) *stamps False* |
  ⟦*neverDistinct* (*stamps x*) (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerEqualsNode x y*) *stamps True* |
  ⟦*is-IntegerStamp* (*stamps x*);
    is-IntegerStamp (*stamps y*);
    stpi-upper (*stamps x*) < *stpi-lower* (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerLessThanNode x y*) *stamps True* |
  ⟦*is-IntegerStamp* (*stamps x*);
    is-IntegerStamp (*stamps y*);
    stpi-lower (*stamps x*) ≥ *stpi-upper* (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerLessThanNode x y*) *stamps False*

code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*) *tryFold* ⟨*proof*⟩

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* ⊢ (*nid*,*m*,*h*) → (*nid*′,*m*′,*h*)

lemma *is-stamp-empty-valid*:
  assumes *is-stamp-empty s*
  shows ¬(∃ *val*. *valid-value val s*)
  ⟨*proof*⟩

lemma *join-valid*:
  assumes *is-IntegerStamp s1* ∧ *is-IntegerStamp s2*
  assumes *valid-stamp s1* ∧ *valid-stamp s2*
  shows (*valid-value v s1* ∧ *valid-value v s2*) = *valid-value v* (*join s1 s2*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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 ¬(*val-to-bool v*)
⟨*proof*⟩

theorem *tryFoldTrue-valid*:
assumes *wf-stamp g stamps*
assumes *tryFold* (*kind g nid*) *stamps True*
assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
shows *val-to-bool v*
⟨*proof*⟩

theorem *tryFoldFalse-valid*:
assumes *wf-stamp g stamps*
assumes *tryFold* (*kind g nid*) *stamps False*
assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
shows ¬(*val-to-bool v*)
⟨*proof*⟩

1.2 Lift rules

inductive *condset-implies* :: *IRExpr set* ⇒ *IRExpr* ⇒ *bool* ⇒ *bool* **where**
impliesTrue:
(∃ *ce* ∈ *conds* . (*ce* ⇛ *cond*)) ⟹ *condset-implies conds cond True* |
impliesFalse:
(∃ *ce* ∈ *conds* . (*ce* ⇛¬ *cond*)) ⟹ *condset-implies conds cond False*

code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*) *condset-implies* ⟨*proof*⟩

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

fun *conds-implies* :: *IRExpr set* ⇒ (*ID* ⇒ *Stamp*) ⇒ *IRNode* ⇒ *IRExpr* ⇒ *bool option* **where**
conds-implies conds stamps condNode cond =
(*if condset-implies conds cond True* ∨ *tryFold condNode stamps True*
then Some True
else if condset-implies conds cond False ∨ *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 ⇒ (*ID* ⇒ *Stamp*) ⇒ *ID* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool*

where
impliesTrue:
$[\![$*kind g ifcond* = (*IfNode cid t f*);
  $g \vdash$ *cid* $\simeq$ *cond*;
  condNode = *kind g cid*;
  conds-implies conds stamps condNode cond = (*Some True*);
  g' = *constantCondition True ifcond* (*kind g ifcond*) *g*
  $]\!]$ $\Longrightarrow$ *ConditionalEliminationStep conds stamps ifcond g* g' |

impliesFalse:
$[\![$*kind g ifcond* = (*IfNode cid t f*);
  $g \vdash$ *cid* $\simeq$ *cond*;
  condNode = *kind g cid*;
  conds-implies conds stamps condNode cond = (*Some False*);
  g' = *constantCondition False ifcond* (*kind g ifcond*) *g*
  $]\!]$ $\Longrightarrow$ *ConditionalEliminationStep conds stamps ifcond g* g' |

unknown:
$[\![$*kind g ifcond* = (*IfNode cid t f*);
  $g \vdash$ *cid* $\simeq$ *cond*;
  condNode = *kind g cid*;
  conds-implies conds stamps condNode cond = *None*
  $]\!]$ $\Longrightarrow$ *ConditionalEliminationStep conds stamps ifcond g g* |

notIfNode:
$\neg$(*is-IfNode* (*kind g ifcond*)) $\Longrightarrow$
  ConditionalEliminationStep conds stamps ifcond g g

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

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* (λ*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 ⇒ DominatorCache ⇒ ID ⇒ ID set set ⇒ ID list ⇒ DominatorCache ⇒ ID set set ⇒ ID list ⇒ bool **and**
dominators :: IRGraph ⇒ DominatorCache ⇒ ID ⇒ (ID set × DominatorCache) ⇒ 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 ∪ {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 ↦ ({nid} ∪ (⋂ doms)))⟧
⟹ dominators g c nid (({nid} ∪ (⋂ 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 $\bigcap\ \emptyset = coset\ []$ which swallows anything unioned with it.
value $\bigcap(\{\}::nat\ set\ set)$
value $-\ \bigcap(\{\}::nat\ set\ set)$
value $\bigcap(\{\{\},\ \{0\}\}::nat\ set\ set)$
value $\{0::nat\} \cup (\bigcap\{\})$

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$) *dominators-all* ⟨*proof*⟩
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *dominators* ⟨*proof*⟩

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 $\{(snd\ x)\ 13|\ x.\ dominators\ ConditionalEliminationTest13\text{-}testSnippet2\text{-}initial\ Map.empty\ 25\ x\}$

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* ⇒ *DominatorCache* ⇒ *ID* ⇒ (*Conditions* × *StampFlow* × *DominatorCache*) ⇒ *bool* **where**
⟦*dominators g c nid* (*doms*, *c'*);
conditions-of-dominators g (*sorted-list-of-set doms*) [] = *conds*;
stamps-of-dominators g (*sorted-list-of-set doms*) [*stamp g*] = *stamps*⟧
⟹ *analyse g c nid* (*conds*, *stamps*, *c'*)

code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*) *analyse* ⟨*proof*⟩

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

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

inductive *Step*
:: *IRGraph* ⇒ (*ID* × *Seen*) ⇒ (*ID* × *Seen*) *option* ⇒ *bool*
for *g* **where**
— We can find a successor edge that is not in seen, go there
⟦*seen'* = {*nid*} ∪ *seen*;

Some nid' = *next-nid g seen' nid*;
nid' ∉ *seen'*⟧
⟹ *Step g* (*nid*, *seen*) (*Some* (*nid'*, *seen'*)) |

— We can cannot find a successor edge that is not in seen, give back None
⟦*seen'* = {*nid*} ∪ *seen*;

None = *next-nid g seen' nid*⟧
⟹ *Step g* (*nid*, *seen*) *None* |

— We've already seen this node, give back None
⟦*seen'* = {*nid*} ∪ *seen*;

Some nid' = *next-nid g seen' nid*;
nid' ∈ *seen'*⟧ ⟹ *Step g* (*nid*, *seen*) *None*

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *Step* ⟨*proof*⟩

fun *nextNode* :: *IRGraph* ⇒ *Seen* ⇒ (*ID* × *Seen*) *option* **where**
  nextNode g seen =
    (*let toSee* = *sorted-list-of-set* {*n* ∈ *ids g*. *n* ∉ *seen*} *in*
      case toSee of [] ⇒ *None* | (*x* # *xs*) ⇒ *Some* (*x*, *seen* ∪ {*x*}))

values {*x*. *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 *ConditionalEliminationPhase*
  :: (*Seen* × *DominatorCache*) ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool*
  where

  — Can do a step and optimise for the current node
  ⟦*nextNode g seen* = *Some* (*nid*, *seen′*);

    analyse g c nid (*conds*, *flow*, *c′*);
    ConditionalEliminationStep (*set conds*) (*hd flow*) *nid g g′*;

    ConditionalEliminationPhase (*seen′*, *c′*) *g′ g″*⟧
    ⟹ *ConditionalEliminationPhase* (*seen*, *c*) *g g″* |

  ⟦*nextNode g seen* = *None*⟧
    ⟹ *ConditionalEliminationPhase* (*seen*, *c*) *g g*

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *ConditionalEliminationPhase* ⟨*proof*⟩

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

values {(*doms*, *c′*)| *doms c′*.
dominators ConditionalEliminationTest13-testSnippet2-initial Map.empty 6 (*doms*, *c′*)}

values {(*conds*, *stamps*, *c*)| *conds stamps c* .
analyse ConditionalEliminationTest13-testSnippet2-initial Map.empty 6 (*conds*, *stamps*, *c*)}
value
  (*nextNode*
    ConditionalEliminationTest13-testSnippet2-initial {*0*,*1*,*2*,*3*,*4*,*5*,*6*,*7*,*8*,*9*,*10*,*11*,*12*,*13*,*14*,*15*,*16*,*17*,*18*,*19*,

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

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

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

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

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

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

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*: $nid \in ids\ g$
assumes *conds-valid*: $\forall\ c \in conds\ .\ \exists\ v.\ ([m, p] \vdash c \mapsto v) \wedge val\text{-}to\text{-}bool\ v$
assumes *ce*: *ConditionalEliminationStep conds stamps nid g g′*

shows $\exists nid'\ .(g\ m\ p\ h \vdash nid \leadsto nid') \longrightarrow (g'\ m\ p\ h \vdash nid \leadsto nid')$
⟨*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*: $wf\text{-}graph\ g \wedge wf\text{-}stamp\ g\ stamps \wedge wf\text{-}values\ g$
assumes *nid*: $nid \in ids\ g$
assumes *conds-valid*: $\forall\ c \in conds\ .\ \exists\ v.\ ([m, p] \vdash c \mapsto v) \wedge val\text{-}to\text{-}bool\ v$
assumes *ce*: *ConditionalEliminationStep conds stamps nid g g′*
assumes *gstep*: $\exists\ h\ nid'.\ (g, p \vdash (nid, m, h) \rightarrow (nid', m, h))$

shows $nid \mid g \sim g'$
⟨*proof*⟩

experiment begin

lemma *inverse-succ*:
$\forall n' \in (succ\ g\ n).\ n \in ids\ g \longrightarrow n \in (predecessors\ g\ n')$
⟨*proof*⟩

lemma *sequential-successors*:
assumes *is-sequential-node n*
shows $successors\text{-}of\ n \neq []$
⟨*proof*⟩

lemma *nid′-succ*:
assumes $nid \in ids\ g$
assumes $\neg(is\text{-}AbstractEndNode\ (kind\ g\ nid0))$
assumes $g, p \vdash (nid0, m0, h0) \rightarrow (nid, m, h)$
shows $nid \in succ\ g\ nid0$
⟨*proof*⟩

lemma *nid′-pred*:
assumes $nid \in ids\ g$
assumes $\neg(is\text{-}AbstractEndNode\ (kind\ g\ nid0))$
assumes $g, p \vdash (nid0, m0, h0) \rightarrow (nid, m, h)$
shows $nid0 \in predecessors\ g\ nid$

⟨*proof*⟩

definition *wf-pred*:
wf-pred g = (∀ *n* ∈ *ids g*. *card* (*predecessors g n*) = *1*)

lemma
assumes ¬(*is-AbstractMergeNode* (*kind g n′*))
assumes *wf-pred g*
shows ∃ *v*. *predecessors g n* = {*v*} ∧ *pred g n′* = *Some v*
⟨*proof*⟩

lemma *inverse-succ1*:
assumes ¬(*is-AbstractEndNode* (*kind g n′*))
assumes *wf-pred g*
shows ∀ *n′* ∈ (*succ g n*). *n* ∈ *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* ⇒ *Block* ⇒ *Block* ⇒ *bool* (- ⊢ - ≤≤ - *20*) **where**
⟦(*z* = *n*) ∨ ((*succ g n* ≠ {}) ∧ (∀ *s* ∈ *succ g n* . (*g* ⊢ *z* ≤≤ *s*)))⟧ ⟹ *postdominates g z n*
code-pred [*show-modes*] *postdominates* ⟨*proof*⟩

inductive *strictly-dominates* :: *IRGraph* ⇒ *Block* ⇒ *Block* ⇒ *bool* (- ⊢ - >> - *20*) **where**
⟦(*g* ⊢ *d* ≥≥ *n*); (*d* ≠ *n*)⟧ ⟹ *strictly-dominates g d n*
code-pred [*show-modes*] *strictly-dominates* ⟨*proof*⟩

inductive *strictly-postdominates* :: *IRGraph* ⇒ *Block* ⇒ *Block* ⇒ *bool* (- ⊢ - << - *20*) **where**
⟦(*g* ⊢ *d* ≤≤ *n*); (*d* ≠ *n*)⟧ ⟹ *strictly-postdominates g d n*
code-pred [*show-modes*] *strictly-postdominates* ⟨*proof*⟩

lemma *pred g nid* = {} ⟶ ¬(∃ *d* . (*d* ≠ *nid*) ∧ (*g* ⊢ *d* ≥≥ *nid*))
⟨*proof*⟩

lemma *succ g nid* = {} ⟶ ¬(∃ *d* . (*d* ≠ *nid*) ∧ (*g* ⊢ *d* ≤≤ *nid*))
⟨*proof*⟩

lemma *pred g nid* = {} ⟶ ¬(∃ *d* . (*g* ⊢ *d* >> *nid*))
⟨*proof*⟩

lemma *succ g nid* = {} ⟶ ¬(∃ *d* . (*g* ⊢ *d* << *nid*))
⟨*proof*⟩

inductive *wf-cfg* :: *IRGraph* ⇒ *bool* **where**
⟦∀ *nid* ∈ *ids g* . (*blockOf g nid* ≠ *NoBlock*) ⟶ (*g* ⊢ (*blockOf g 0*) ≥≥ (*blockOf g nid*))⟧
⟹ *wf-cfg g*
code-pred [*show-modes*] *wf-cfg* ⟨*proof*⟩

inductive *immediately-dominates* :: *IRGraph* ⇒ *Block* ⇒ *Block* ⇒ *bool* (- ⊢ - *idom* - *20*) **where**
⟦(*g* ⊢ *d* >> *n*); (∀ *w* ∈ *ids g* . (*g* ⊢ (*blockOf g w*) >> *n*) ⟶ (*g* ⊢ (*blockOf g w*) ≥≥ *d*))⟧ ⟹ *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* ⊢ *blockOf simple-if 0* ≥≥ *blockOf simple-if 0*
value *simple-if* ⊢ *blockOf simple-if 0* ≥≥ *blockOf simple-if 3*
value *simple-if* ⊢ *blockOf simple-if 0* ≥≥ *blockOf simple-if 4*
value *simple-if* ⊢ *blockOf simple-if 0* ≥≥ *blockOf simple-if 12*

value *simple-if* ⊢ *blockOf simple-if 3* ≥≥ *blockOf simple-if 0*
value *simple-if* ⊢ *blockOf simple-if 3* ≥≥ *blockOf simple-if 3*
value *simple-if* ⊢ *blockOf simple-if 3* ≥≥ *blockOf simple-if 4*
value *simple-if* ⊢ *blockOf simple-if 3* ≥≥ *blockOf simple-if 12*

value *simple-if* ⊢ *blockOf simple-if 4* ≥≥ *blockOf simple-if 0*
value *simple-if* ⊢ *blockOf simple-if 4* ≥≥ *blockOf simple-if 3*
value *simple-if* ⊢ *blockOf simple-if 4* ≥≥ *blockOf simple-if 4*
value *simple-if* ⊢ *blockOf simple-if 4* ≥≥ *blockOf simple-if 12*

value *simple-if* ⊢ *blockOf simple-if 12* ≥≥ *blockOf simple-if 0*
value *simple-if* ⊢ *blockOf simple-if 12* ≥≥ *blockOf simple-if 3*
value *simple-if* ⊢ *blockOf simple-if 12* ≥≥ *blockOf simple-if 4*
value *simple-if* ⊢ *blockOf simple-if 12* ≥≥ *blockOf simple-if 12*

value *simple-if* ⊢ *blockOf simple-if 0* ≤≤ *blockOf simple-if 0*
value *simple-if* ⊢ *blockOf simple-if 0* ≤≤ *blockOf simple-if 3*
value *simple-if* ⊢ *blockOf simple-if 0* ≤≤ *blockOf simple-if 4*
value *simple-if* ⊢ *blockOf simple-if 0* ≤≤ *blockOf simple-if 12*

value *simple-if* ⊢ *blockOf simple-if 3* ≤≤ *blockOf simple-if 0*
value *simple-if* ⊢ *blockOf simple-if 3* ≤≤ *blockOf simple-if 3*
value *simple-if* ⊢ *blockOf simple-if 3* ≤≤ *blockOf simple-if 4*

value *simple-if* ⊢ *blockOf simple-if 3* ≤≤ *blockOf simple-if 12*

value *simple-if* ⊢ *blockOf simple-if 4* ≤≤ *blockOf simple-if 0*
value *simple-if* ⊢ *blockOf simple-if 4* ≤≤ *blockOf simple-if 3*
value *simple-if* ⊢ *blockOf simple-if 4* ≤≤ *blockOf simple-if 4*
value *simple-if* ⊢ *blockOf simple-if 4* ≤≤ *blockOf simple-if 12*

value *simple-if* ⊢ *blockOf simple-if 12* ≤≤ *blockOf simple-if 0*
value *simple-if* ⊢ *blockOf simple-if 12* ≤≤ *blockOf simple-if 3*
value *simple-if* ⊢ *blockOf simple-if 12* ≤≤ *blockOf simple-if 4*
value *simple-if* ⊢ *blockOf simple-if 12* ≤≤ *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