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 ⇛ q_2 requires that the implication $q_1 \longrightarrow q_2$ is known true (i.e. universally valid). The relation q_1 ⇛¬ 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 ⇛ q_2 nor q_1 ⇛¬ 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*) ⟶

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

The relation $q_1 \rightarrowtail q_2$ means $q_1 \longrightarrow q_2$ is universally valid, and the relation $q_1 \not\rightarrowtail q_2$ means $q_1 \longrightarrow \neg\ q_2$ is universally valid.

```
lemma eq-not-less-val:
  val-to-bool(val[v1 eq v2]) ⟶ ¬val-to-bool(val[v1 < v2])
  proof –
  have unfoldEqualDefined: (intval-equals v1 v2 ≠ UndefVal) ⟹
      (val-to-bool(intval-equals v1 v2) ⟶ (¬(val-to-bool(intval-less-than v1 v2))))
    subgoal premises p
  proof –
    obtain v1b v1v where v1v: v1 = IntVal v1b v1v
      by (metis array-length.cases intval-equals.simps(2,3,4,5) p)
    obtain v2b v2v where v2v: v2 = IntVal v2b v2v
      by (metis Value.exhaust-sel intval-equals.simps(6,7,8,9) p)
    have sameWidth: v1b=v2b
      by (metis bool-to-val-bin.simps intval-equals.simps(1) p v1v v2v)
    have unfoldEqual: intval-equals v1 v2 = (bool-to-val (v1v=v2v))
      by (simp add: sameWidth v1v v2v)
   have unfoldLessThan: intval-less-than v1 v2 = (bool-to-val (int-signed-value v1b
v1v < int-signed-value v2b v2v))
      by (simp add: sameWidth v1v v2v)
    have val: ((v1v=v2v)) ⟶ (¬((int-signed-value v1b v1v < int-signed-value v2b
v2v)))
      using sameWidth by auto
    have doubleCast0: val-to-bool (bool-to-val ((v1v = v2v))) = (v1v = v2v)
      using bool-to-val.elims val-to-bool.simps(1) by fastforce
  have doubleCast1: val-to-bool (bool-to-val ((int-signed-value v1b v1v < int-signed-value
v2b v2v))) =
                                                (int-signed-value v1b v1v < int-signed-value
v2b v2v)
      using bool-to-val.elims val-to-bool.simps(1) by fastforce
    then show ?thesis
      using p val unfolding unfoldEqual unfoldLessThan doubleCast0 doubleCast1
by blast
  qed done
  show ?thesis
    by (metis Value.distinct(1) val-to-bool.elims(2) unfoldEqualDefined)
qed

lemma eq-not-less'-val:
  val-to-bool(val[v1 eq v2]) ⟶ ¬val-to-bool(val[v2 < v1])
proof –
  have a: intval-equals v1 v2 = intval-equals v2 v1
    apply (cases intval-equals v1 v2 = UndefVal)
    apply (smt (z3) bool-to-val-bin.simps intval-equals.elims intval-equals.simps)
    subgoal premises p
    proof –
      obtain v1b v1v where v1v: v1 = IntVal v1b v1v
```

by (*metis Value.exhaust-sel intval-equals.simps*(*2*,*3*,*4*,*5*) *p*)
obtain *v2b v2v* **where** *v2v*: *v2* = *IntVal v2b v2v*
by (*metis Value.exhaust-sel intval-equals.simps*(*6*,*7*,*8*,*9*) *p*)
then show *?thesis*
by (*smt* (*verit*) *bool-to-val-bin.simps intval-equals.simps*(*1*) *v1v*)
qed done
show *?thesis*
using *a eq-not-less-val* **by** *presburger*
qed

lemma *less-not-less-val*:
val-to-bool(*val*[*v1* < *v2*]) ⟶ ¬*val-to-bool*(*val*[*v2* < *v1*])
apply (*rule impI*)
subgoal premises *p*
proof −
obtain *v1b v1v* **where** *v1v*: *v1* = *IntVal v1b v1v*
by (*metis Value.exhaust-sel intval-less-than.simps*(*2*,*3*,*4*,*5*) *p val-to-bool.simps*(*2*))
obtain *v2b v2v* **where** *v2v*: *v2* = *IntVal v2b v2v*
by (*metis Value.exhaust-sel intval-less-than.simps*(*6*,*7*,*8*,*9*) *p val-to-bool.simps*(*2*))
then have *unfoldLessThanRHS*: *intval-less-than v2 v1* =
(*bool-to-val* (*int-signed-value v2b v2v* < *int-signed-value*
v1b v1v))
using *p v1v* **by** *force*
then have *unfoldLessThanLHS*: *intval-less-than v1 v2* =
(*bool-to-val* (*int-signed-value v1b v1v* < *int-signed-value*
v2b v2v))
using *bool-to-val-bin.simps intval-less-than.simps*(*1*) *p v1v v2v val-to-bool.simps*(*2*)
by *auto*
then have *symmetry*: (*int-signed-value v2b v2v* < *int-signed-value v1b v1v*) ⟶
(¬(*int-signed-value v1b v1v* < *int-signed-value v2b v2v*))
by *simp*
then show *?thesis*
using *p unfoldLessThanLHS unfoldLessThanRHS* **by** *fastforce*
qed done

lemma *less-not-eq-val*:
val-to-bool(*val*[*v1* < *v2*]) ⟶ ¬*val-to-bool*(*val*[*v1 eq v2*])
using *eq-not-less-val* **by** *blast*

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

lemma *intval-logic-negation-inverse*:
assumes *b* > *0*
assumes *x* = *IntVal b v*
shows *val-to-bool* (*intval-logic-negation x*) ⟷ ¬(*val-to-bool x*)

using *assms* **by** (*cases x*; *auto simp*: *logic-negate-def*)

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

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

theorem *implies-impliesnot-valid*:
shows $((q1 \Rrightarrow q2) \longrightarrow (q1 \rightarrowtail q2)) \wedge$
$((q1 \Rrightarrow\neg\ q2) \longrightarrow (q1 \nrightarrow q2))$
(**is** $(?imp \longrightarrow ?val) \wedge (?notimp \longrightarrow ?notval)$)
proof (*induct q1 q2 rule*: *impliesx-impliesnot.induct*)
case (*same q*)
then show *?case*
using *evalDet* **by** *fastforce*
next
case (*eq-not-less x y*)
then show *?case* **apply** *auto*[*1*] **using** *eq-not-less-val evalDet* **by** *blast*
next
case (*eq-not-less′ x y*)
then show *?case* **apply** *auto*[*1*] **using** *eq-not-less′-val evalDet* **by** *blast*
next
case (*less-not-less x y*)
then show *?case* **apply** *auto*[*1*] **using** *less-not-less-val evalDet* **by** *blast*
next
case (*less-not-eq x y*)
then show *?case* **apply** *auto*[*1*] **using** *less-not-eq-val evalDet* **by** *blast*
next
case (*less-not-eq′ x y*)
then show *?case* **apply** *auto*[*1*] **using** *eq-not-less′-val evalDet* **by** *metis*
next
case (*negate-true x y*)
then show *?case* **apply** *auto*[*1*]
by (*metis logic-negation-relation-tree unary-eval.simps*(*4*) *unfold-unary*)
next
case (*negate-false x y*)
then show *?case* **apply** *auto*[*1*]
by (*metis UnaryExpr logic-negation-relation-tree unary-eval.simps*(*4*))
qed

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* **.**

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*)
  using *assms is-stamp-empty.simps* **apply** (*cases s*; *auto*)
  by (*metis linorder-not-le not-less-iff-gr-or-eq order.strict-trans valid-value.elims*(*2*) *valid-value.simps*(*1*) *valid-value.simps*(*5*))

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
  assume *?lhs*
  then show *?rhs*
   using *assms*(*1*) **apply** (*cases s1*; *cases s2*; *auto*)
   apply (*metis Value.inject*(*1*) *valid-int*)
  by (*smt* (*z3*) *valid-int valid-stamp.simps*(*1*) *valid-value.simps*(*1*))

next
assume *?rhs*
then show *?lhs*
using *assms* **apply** (*cases s1*; *cases s2*; *simp*)
by (*smt* (*verit*, *best*) *assms*(*2*) *valid-int valid-value.simps*(*1*) *valid-value.simps*(*22*))
qed

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 −
obtain *stampx stampy* **where** *stampdef*: *stampx* = *stamps x* ∧ *stampy* = *stamps y*
by *simp*
then have *xv*: ∀ *xv* . ([*g*, *m*, *p*] ⊢ *x* ↦ *xv*) ⟶ *valid-value xv stampx*
by (*meson assms*(*1*) *encodeeval.simps eval-in-ids wf-stamp.elims*(*2*))
from *stampdef* **have** *yv*: ∀ *yv* . ([*g*, *m*, *p*] ⊢ *y* ↦ *yv*) ⟶ *valid-value yv stampy*
by (*meson assms*(*1*) *encodeeval.simps eval-in-ids wf-stamp.elims*(*2*))
have ∀ *v*. *valid-value v* (*join stampx stampy*) = (*valid-value v stampx* ∧ *valid-value v stampy*)
using *assms*(*3*)
by (*simp add*: *join-valid stampdef*)
then show *?thesis*
using *assms* **unfolding** *alwaysDistinct.simps*
using *is-stamp-empty-valid stampdef xv yv* **by** *blast*
qed

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 −
have *no-valid*: ∀ *val*. ¬(*valid-value val* (*join* (*stamps x*) (*stamps y*)))
by (*smt* (*verit*, *best*) *is-stamp-empty.elims*(*2*) *valid-int valid-value.simps*(*1*) *assms*(*1*,*4*)
alwaysDistinct.simps)
obtain *xe ye* **where** *repr*: *rep g nid* (*BinaryExpr BinIntegerEquals xe ye*)
by (*metis assms*(*2*) *assms*(*3*) *encodeeval.simps rep-integer-equals*)
moreover have *evale*: [*m*, *p*] ⊢ (*BinaryExpr BinIntegerEquals xe ye*) ↦ *v*
by (*metis assms*(*3*) *calculation encodeeval.simps repDet*)
moreover have *repsub*: *rep g x xe* ∧ *rep g y ye*
by (*metis IRNode.distinct*(*1955*) *IRNode.distinct*(*1997*) *IRNode.inject*(*17*) *IntegerEqualsNodeE assms*(*2*) *calculation*)
ultimately obtain *xv yv* **where** *evalsub*: [*g*, *m*, *p*] ⊢ *x* ↦ *xv* ∧ [*g*, *m*, *p*] ⊢ *y* ↦

yv
  by (*meson BinaryExprE encodeeval.simps*)
 have *xvalid*: *valid-value xv* (*stamps x*)
  using *assms*(*1*) *encode-in-ids encodeeval.simps evalsub wf-stamp.simps* **by** *blast*
 then have *xint*: *is-IntegerStamp* (*stamps x*)
  using *assms*(*4*) *valid-value.elims*(*2*) **by** *fastforce*
 then have *xstamp*: *valid-stamp* (*stamps x*)
  using *xvalid* **apply** (*cases xv*; *auto*)
  apply (*smt* (*z3*) *valid-stamp.simps*(*6*) *valid-value.elims*(*1*))
  using *is-IntegerStamp-def* **by** *fastforce*
 have *yvalid*: *valid-value yv* (*stamps y*)
  using *assms*(*1*) *encode-in-ids encodeeval.simps evalsub wf-stamp.simps* **by** *blast*
 then have *yint*: *is-IntegerStamp* (*stamps y*)
  using *assms*(*4*) *valid-value.elims*(*2*) **by** *fastforce*
 then have *ystamp*: *valid-stamp* (*stamps y*)
  using *yvalid* **apply** (*cases yv*; *auto*)
  apply (*smt* (*z3*) *valid-stamp.simps*(*6*) *valid-value.elims*(*1*))
  using *is-IntegerStamp-def* **by** *fastforce*
 have *disjoint*: ¬(∃ *val* . ([*g*, *m*, *p*] ⊢ *x* ↦ *val*) ∧ ([*g*, *m*, *p*] ⊢ *y* ↦ *val*))
  using *alwaysDistinct-evaluate*
  using *assms*(*1*) *assms*(*4*) *xint yint xvalid yvalid xstamp ystamp* **by** *simp*
 have *v* = *bin-eval BinIntegerEquals xv yv*
  by (*metis BinaryExprE encodeeval.simps evale evalsub graphDet repsub*)
 also have *v* ≠ *UndefVal*
  using *evale* **by** *auto*
 ultimately have ∃ *b1 b2*. *v* = *bool-to-val-bin b1 b2* (*xv* = *yv*)
  unfolding *bin-eval.simps*
  by (*smt* (*z3*) *Value.inject*(*1*) *bool-to-val-bin.simps intval-equals.elims*)
 then show *?thesis*
  by (*metis* (*mono-tags*, *lifting*) ‹(*v*::*Value*) ≠ *UndefVal*› *bool-to-val.elims bool-to-val-bin.simps disjoint evalsub val-to-bool.simps*(*1*))
qed
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*)))
 using *assms* **apply** *auto*[*1*]
 by (*simp add*: *take-bit-signed-take-bit*)

lemma *asConstant-valid*:
 assumes *asConstant s* = *val*
 assumes *val* ≠ *UndefVal*
 assumes *valid-value v s*
 shows *v* = *val*
proof −
 obtain *b l h* **where** *s*: *s* = *IntegerStamp b l h*

```
    using assms(1,2) by (cases s; auto)
  obtain vv where vdef: v = IntVal b vv
    using assms(3) s valid-int by blast
  have l ≤ int-signed-value b vv ∧ int-signed-value b vv ≤ h
   by (metis ‹(v::Value) = IntVal (b::nat) (vv::64 word)› assms(3) s valid-value.simps(1))
  then have veq: int-signed-value b vv = l
    by (smt (verit) asConstant.simps(1) assms(1) assms(2) s)
  have valdef: val = new-int b (word-of-int l)
    by (metis asConstant.simps(1) assms(1) assms(2) s)
  have take-bit b vv = vv
   by (metis ‹(v::Value) = IntVal (b::nat) (vv::64 word)› assms(3) s valid-value.simps(1))
  then show ?thesis
    using veq vdef valdef
    using assms(3) s unwrap-valid by force
qed

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 -
  obtain val where constx: asConstant (stamps x) = val
    by simp
  moreover have val ≠ UndefVal
    using assms(4) calculation by auto
  then have constx: val = asConstant (stamps y)
    using calculation assms(4) by force
  obtain xe ye where repr: rep g nid (BinaryExpr BinIntegerEquals xe ye)
    by (metis assms(2) assms(3) encodeeval.simps rep-integer-equals)
  moreover have evale: [m, p] ⊢ (BinaryExpr BinIntegerEquals xe ye) ↦ v
    by (metis assms(3) calculation encodeeval.simps repDet)
  moreover have repsub: rep g x xe ∧ rep g y ye
    by (metis IRNode.distinct(1955) IRNode.distinct(1997) IRNode.inject(17) In-
tegerEqualsNodeE assms(2) calculation)
  ultimately obtain xv yv where evalsub: [g, m, p] ⊢ x ↦ xv ∧ [g, m, p] ⊢ y ↦
yv
    by (meson BinaryExprE encodeeval.simps)
  have xvalid: valid-value xv (stamps x)
   using assms(1) encode-in-ids encodeeval.simps evalsub wf-stamp.simps by blast
  then have xint: is-IntegerStamp (stamps x)
    using assms(4) valid-value.elims(2) by fastforce
  have yvalid: valid-value yv (stamps y)
   using assms(1) encode-in-ids encodeeval.simps evalsub wf-stamp.simps by blast
  then have yint: is-IntegerStamp (stamps y)
    using assms(4) valid-value.elims(2) by fastforce
  have eq: ∀ v1 v2. (([g, m, p] ⊢ x ↦ v1) ∧ ([g, m, p] ⊢ y ↦ v2)) ⟶ v1 = v2
   by (metis asConstant-valid assms(4) encodeEvalDet evalsub neverDistinct.elims(1)
```

xvalid yvalid)
  have *v* = *bin-eval BinIntegerEquals xv yv*
    by (*metis BinaryExprE encodeeval.simps evale evalsub graphDet repsub*)
  also have *v* ≠ *UndefVal*
    using *evale* **by** *auto*
  ultimately have ∃ *b1 b2*. *v* = *bool-to-val-bin b1 b2* (*xv* = *yv*)
    unfolding *bin-eval.simps*
    by (*smt* (*z3*) *Value.inject*(*1*) *bool-to-val-bin.simps intval-equals.elims*)
  then show *?thesis*
    using ‹(*v*::*Value*) ≠ *UndefVal*› *eq evalsub* **by** *fastforce*
qed

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 −
  obtain *xe ye* **where** *repr*: *rep g nid* (*BinaryExpr BinIntegerLessThan xe ye*)
    by (*metis assms*(*2*) *assms*(*3*) *encodeeval.simps rep-integer-less-than*)
  moreover have *evale*: [*m*, *p*] ⊢ (*BinaryExpr BinIntegerLessThan xe ye*) ↦ *v*
    by (*metis assms*(*3*) *calculation encodeeval.simps repDet*)
  moreover have *repsub*: *rep g x xe* ∧ *rep g y ye*
    by (*metis IRNode.distinct*(*2047*) *IRNode.distinct*(*2089*) *IRNode.inject*(*18*) *IntegerLessThanNodeE assms*(*2*) *repr*)
  ultimately obtain *xv yv* **where** *evalsub*: [*g*, *m*, *p*] ⊢ *x* ↦ *xv* ∧ [*g*, *m*, *p*] ⊢ *y* ↦ *yv*
    by (*meson BinaryExprE encodeeval.simps*)
  have *vval*: *v* = *intval-less-than xv yv*
    by (*metis BinaryExprE bin-eval.simps*(*14*) *encodeEvalDet encodeeval.simps evale evalsub repsub*)
  then obtain *b xvv* **where** *xv* = *IntVal b xvv*
    by (*metis bin-eval.simps*(*14*) *defined-eval-is-intval evale evaltree-not-undef is-IntVal-def*)
  also have *xvalid*: *valid-value xv* (*stamps x*)
    by (*meson assms*(*1*) *encodeeval.simps eval-in-ids evalsub wf-stamp.elims*(*2*))
  then obtain *xl xh* **where** *xstamp*: *stamps x* = *IntegerStamp b xl xh*
    using *calculation valid-value.simps* **apply** (*cases stamps x*; *auto*)
    by *presburger*
  from *vval* **obtain** *yvv* **where** *yint*: *yv* = *IntVal b yvv*
    by (*metis Value.collapse*(*1*) *bin-eval.simps*(*14*) *bool-to-val-bin.simps calculation defined-eval-is-intval evale evaltree-not-undef intval-less-than.simps*(*1*))
  then have *yvalid*: *valid-value yv* (*stamps y*)
    using *assms*(*1*) *encodeeval.simps evalsub no-encoding wf-stamp.simps* **by** *blast*
  then obtain *yl yh* **where** *ystamp*: *stamps y* = *IntegerStamp b yl yh*
    using *calculation yint valid-value.simps* **apply** (*cases stamps y*; *auto*)
    by *presburger*
  have *int-signed-value b xvv* ≤ *xh*
    using *calculation valid-value.simps*(*1*) *xstamp xvalid* **by** *presburger*

moreover have *yl* ≤ *int-signed-value b yvv*
using *valid-value.simps*(*1*) *yint ystamp yvalid* **by** *presburger*
moreover have *xh* < *yl*
using *assms*(*4*) *xstamp ystamp* **by** *auto*
ultimately have *int-signed-value b xvv* < *int-signed-value b yvv*
by *linarith*
then have *val-to-bool* (*intval-less-than xv yv*)
by (*simp add*: ‹(*xv*::*Value*) = *IntVal* (*b*::*nat*) (*xvv*::*64 word*)› *yint*)
then show *?thesis*
by (*simp add*: *vval*)
qed

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 −
obtain *xe ye* **where** *repr*: *rep g nid* (*BinaryExpr BinIntegerLessThan xe ye*)
by (*metis assms*(*2*) *assms*(*3*) *encodeeval.simps rep-integer-less-than*)
moreover have *evale*: [*m*, *p*] ⊢ (*BinaryExpr BinIntegerLessThan xe ye*) ↦ *v*
by (*metis assms*(*3*) *calculation encodeeval.simps repDet*)
moreover have *repsub*: *rep g x xe* ∧ *rep g y ye*
by (*metis IRNode.distinct*(*2047*) *IRNode.distinct*(*2089*) *IRNode.inject*(*18*) *IntegerLessThanNodeE assms*(*2*) *repr*)
ultimately obtain *xv yv* **where** *evalsub*: [*g*, *m*, *p*] ⊢ *x* ↦ *xv* ∧ [*g*, *m*, *p*] ⊢ *y* ↦ *yv*
by (*meson BinaryExprE encodeeval.simps*)
have *vval*: *v* = *intval-less-than xv yv*
by (*metis BinaryExprE bin-eval.simps*(*14*) *encodeEvalDet encodeeval.simps evale evalsub repsub*)
then obtain *b xvv* **where** *xv* = *IntVal b xvv*
by (*metis bin-eval.simps*(*14*) *defined-eval-is-intval evale evaltree-not-undef is-IntVal-def*)
also have *xvalid*: *valid-value xv* (*stamps x*)
by (*meson assms*(*1*) *encodeeval.simps eval-in-ids evalsub wf-stamp.elims*(*2*))
then obtain *xl xh* **where** *xstamp*: *stamps x* = *IntegerStamp b xl xh*
using *calculation valid-value.simps* **apply** (*cases stamps x*; *auto*)
by *presburger*
from *vval* **obtain** *yvv* **where** *yint*: *yv* = *IntVal b yvv*
by (*metis Value.collapse*(*1*) *bin-eval.simps*(*14*) *bool-to-val-bin.simps calculation defined-eval-is-intval evale evaltree-not-undef intval-less-than.simps*(*1*))
then have *yvalid*: *valid-value yv* (*stamps y*)
using *assms*(*1*) *encodeeval.simps evalsub no-encoding wf-stamp.simps* **by** *blast*
then obtain *yl yh* **where** *ystamp*: *stamps y* = *IntegerStamp b yl yh*
using *calculation yint valid-value.simps* **apply** (*cases stamps y*; *auto*)
by *presburger*
have *xl* ≤ *int-signed-value b xvv*
using *calculation valid-value.simps*(*1*) *xstamp xvalid* **by** *presburger*

moreover have *int-signed-value b yvv* $\leq$ *yh*
  using *valid-value.simps*(*1*) *yint ystamp yvalid* **by** *presburger*
moreover have *xl* $\geq$ *yh*
  using *assms*(*4*) *xstamp ystamp* **by** *auto*
ultimately have *int-signed-value b xvv* $\geq$ *int-signed-value b yvv*
  by *linarith*
then have ¬(*val-to-bool* (*intval-less-than xv yv*))
  by (*simp add*: ‹(*xv*::*Value*) = *IntVal* (*b*::*nat*) (*xvv*::*64 word*)› *yint*)
then show *?thesis*
  by (*simp add*: *vval*)
qed

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*
  using *assms*(*2*) **proof** (*induction kind g nid stamps True rule*: *tryFold.induct*)
case (*1 stamps x y*)
  then show *?case*
    using *alwaysDistinct-valid assms* **by** *force*
next
  case (*2 stamps x y*)
  then show *?case*
    by (*smt* (*verit*, *best*) *one-neq-zero tryFold.cases neverDistinct-valid assms*
      stampUnder-valid val-to-bool.simps(*1*))
next
  case (*3 stamps x y*)
  then show *?case*
    by (*smt* (*verit*, *best*) *one-neq-zero tryFold.cases neverDistinct-valid assms*
      val-to-bool.simps(*1*) *stampUnder-valid*)
next
case (*4 stamps x y*)
  then show *?case*
    by *force*
qed

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*)
using *assms*(*2*) **proof** (*induction kind g nid stamps False rule*: *tryFold.induct*)
case (*1 stamps x y*)
  then show *?case*
    by (*smt* (*verit*) *stampOver-valid alwaysDistinct-valid tryFold.cases*
      neverDistinct-valid val-to-bool.simps(*1*) *assms*)
next
case (*2 stamps x y*)

then show *?case*
by *blast*
next
case (*3 stamps x y*)
then show *?case*
by *blast*
next
case (*4 stamps x y*)
then show *?case*
by (*smt* (*verit*, *del-insts*) *tryFold.cases alwaysDistinct-valid val-to-bool.simps*(*1*) *stampOver-valid assms*)
qed

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* **.**

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 ⊢ *cid* ≃ *cond*;

$condNode = kind\ g\ cid;$
$conds\text{-}implies\ conds\ stamps\ condNode\ cond = (Some\ True);$
$g' = constantCondition\ True\ ifcond\ (kind\ g\ ifcond)\ g$
$\rrbracket \Longrightarrow ConditionalEliminationStep\ conds\ stamps\ ifcond\ g\ g'\ |$

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

unknown:
$\llbracket kind\ g\ ifcond = (IfNode\ cid\ t\ f);$
$g \vdash cid \simeq cond;$
$condNode = kind\ g\ cid;$
$conds\text{-}implies\ conds\ stamps\ condNode\ cond = None$
$\rrbracket \Longrightarrow ConditionalEliminationStep\ conds\ stamps\ ifcond\ g\ g\ |$

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

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

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

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 ⋂ ∅ = *coset* [] which swallows anything unioned with it.
value ⋂({}::*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* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *dominators* **.**

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* **.**

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* **.**

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* ⇒ *i* ⇒ *o* ⇒ *bool*) *Step* **.**

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* ⇒ *i* ⇒ *o* ⇒ *bool*) *ConditionalEliminationPhase* **.**

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*,*2*

lemma *IfNodeStepE*: *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid*′, *m*′, *h*) ⟹
(⋀*cond tb fb val*.

kind g nid = *IfNode cond tb fb* ⟹
nid' = (*if val-to-bool val then tb else fb*) ⟹
[*g*, *m*, *p*] ⊢ *cond* ↦ *val* ⟹ *m'* = *m*)
using *StepE*
by (*smt* (*verit*, *best*) *IfNode Pair-inject stepDet*)

lemma *ifNodeHasCondEvalStutter*:
assumes (*g m p h* ⊢ *nid* ⇝ *nid'*)
assumes *kind g nid* = *IfNode cond t f*
shows ∃ *v*. ([*g*, *m*, *p*] ⊢ *cond* ↦ *v*)
using *IfNodeStepE assms*(*1*) *assms*(*2*) *stutter.cases* **unfolding** *encodeeval.simps*
by (*smt* (*verit*, *ccfv-SIG*) *IfNodeCond*)

lemma *ifNodeHasCondEval*:
assumes (*g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid'*, *m'*, *h'*))
assumes *kind g nid* = *IfNode cond t f*
shows ∃ *v*. ([*g*, *m*, *p*] ⊢ *cond* ↦ *v*)
using *IfNodeStepE assms*(*1*) *assms*(*2*) **apply** *auto*[*1*]
by (*smt* (*verit*) *IRNode.disc*(*1966*) *IRNode.distinct*(*1733*) *IRNode.distinct*(*1735*) *IRNode.distinct*(*1755*) *IRNode.distinct*(*1757*) *IRNode.distinct*(*1777*) *IRNode.distinct*(*1783*) *IRNode.distinct*(*1787*) *IRNode.distinct*(*1789*) *IRNode.distinct*(*401*) *IRNode.distinct*(*755*) *StutterStep fst-conv ifNodeHasCondEvalStutter is-AbstractEndNode.simps is-EndNode.simps*(*16*) *snd-conv step.cases*)

lemma *replace-if-t*:
assumes *kind g nid* = *IfNode cond tb fb*
assumes [*g*, *m*, *p*] ⊢ *cond* ↦ *bool*
assumes *val-to-bool bool*
assumes *g'*: *g'* = *replace-usages nid tb g*
shows ∃*nid'* .(*g m p h* ⊢ *nid* ⇝ *nid'*) ⟷ (*g' m p h* ⊢ *nid* ⇝ *nid'*)
proof −
have *g1step*: *g*, *p* ⊢ (*nid*, *m*, *h*) → (*tb*, *m*, *h*)
by (*meson IfNode assms*(*1*) *assms*(*2*) *assms*(*3*) *encodeeval.simps*)
have *g2step*: *g'*, *p* ⊢ (*nid*, *m*, *h*) → (*tb*, *m*, *h*)
using *g'* **unfolding** *replace-usages.simps*
by (*simp add*: *stepRefNode*)
from *g1step g2step* **show** *?thesis*
using *StutterStep* **by** *blast*
qed

lemma *replace-if-t-imp*:
assumes *kind g nid* = *IfNode cond tb fb*
assumes [*g*, *m*, *p*] ⊢ *cond* ↦ *bool*
assumes *val-to-bool bool*
assumes *g'*: *g'* = *replace-usages nid tb g*
shows ∃*nid'* .(*g m p h* ⊢ *nid* ⇝ *nid'*) ⟶ (*g' m p h* ⊢ *nid* ⇝ *nid'*)
using *replace-if-t assms* **by** *blast*

lemma *replace-if-f*:

assumes *kind g nid* = *IfNode cond tb fb*
assumes $[g, m, p] \vdash cond \mapsto bool$
assumes $\neg$(*val-to-bool bool*)
assumes g': g' = *replace-usages nid fb g*
shows $\exists nid'.(g\ m\ p\ h \vdash nid \leadsto nid') \longleftrightarrow (g'\ m\ p\ h \vdash nid \leadsto nid')$
proof –
have *g1step*: $g, p \vdash (nid, m, h) \rightarrow (fb, m, h)$
by (*meson IfNode assms*(*1*) *assms*(*2*) *assms*(*3*) *encodeeval.simps*)
have *g2step*: $g', p \vdash (nid, m, h) \rightarrow (fb, m, h)$
using g' **unfolding** *replace-usages.simps*
by (*simp add*: *stepRefNode*)
from *g1step g2step* **show** *?thesis*
using *StutterStep* **by** *blast*
qed

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

shows $\exists nid'.(g\ m\ p\ h \vdash nid \leadsto nid') \longrightarrow (g'\ m\ p\ h \vdash nid \leadsto nid')$
using *ce* **using** *assms*
proof (*induct nid g* g' *rule*: *ConditionalEliminationStep.induct*)
case (*impliesTrue g ifcond cid t f cond conds* g')
show *?case* **proof** (*cases* $\exists nid'.\ (g\ m\ p\ h \vdash ifcond \leadsto nid')$)
case *True*
show *?thesis*
by (*metis StutterStep constantConditionNoIf constantConditionTrue impliesTrue.hyps*(*5*))
next
case *False*
then show *?thesis* **by** *auto*
qed
next
case (*impliesFalse g ifcond cid t f cond conds* g')
then show *?case*
proof (*cases* $\exists nid'.\ (g\ m\ p\ h \vdash ifcond \leadsto nid')$)
case *True*
then show *?thesis*
by (*metis StutterStep constantConditionFalse constantConditionNoIf impliesFalse.hyps*(*5*))
next
case *False*
then show *?thesis*

by *auto*
qed
next
case (*unknown g ifcond cid t f cond condNode conds stamps*)
then show *?case*
by *blast*
next
case (*notIfNode g ifcond conds stamps*)
then show *?case*
by *blast*
qed

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′*
using *ce gstep* **using** *assms*
proof (*induct nid g g′ rule*: *ConditionalEliminationStep.induct*)
case (*impliesTrue g ifcond cid t f cond condNode conds stamps g′*)
from *impliesTrue*(*5*) **obtain** *h* **where** *gstep*: *g*, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
using *IfNode encodeeval.simps ifNodeHasCondEval impliesTrue.hyps*(*1*) *impliesTrue.hyps*(*2*) *impliesTrue.hyps*(*3*) *impliesTrue.prems*(*4*) *implies-impliesnot-valid implies-valid.simps repDet*
by (*smt* (*verit*) *conds-implies.elims condset-implies.simps impliesTrue.hyps*(*4*) *impliesTrue.prems*(*1*) *impliesTrue.prems*(*2*) *option.distinct*(*1*) *option.inject tryFoldTrue-valid*)
have *g′*, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
using *constantConditionTrue impliesTrue.hyps*(*1*) *impliesTrue.hyps*(*5*) **by** *blast*
then show *?case* **using** *gstep*
by (*metis stepDet strong-noop-bisimilar.intros*)
next
case (*impliesFalse g ifcond cid t f cond condNode conds stamps g′*)
from *impliesFalse*(*5*) **obtain** *h* **where** *gstep*: *g*, *p* ⊢ (*ifcond*, *m*, *h*) → (*f*, *m*, *h*)
using *IfNode encodeeval.simps ifNodeHasCondEval impliesFalse.hyps*(*1*) *impliesFalse.hyps*(*2*) *impliesFalse.hyps*(*3*) *impliesFalse.prems*(*4*) *implies-impliesnot-valid impliesnot-valid.simps repDet*
by (*smt* (*verit*) *conds-implies.elims condset-implies.simps impliesFalse.hyps*(*4*) *impliesFalse.prems*(*1*) *impliesFalse.prems*(*2*) *option.distinct*(*1*) *option.inject tryFoldFalse-valid*)
have *g′*, *p* ⊢ (*ifcond*, *m*, *h*) → (*f*, *m*, *h*)
using *constantConditionFalse impliesFalse.hyps*(*1*) *impliesFalse.hyps*(*5*) **by** *blast*
then show *?case* **using** *gstep*

```
    by (metis stepDet strong-noop-bisimilar.intros)
next
  case (unknown g ifcond cid t f cond condNode conds stamps)
  then show ?case
    using strong-noop-bisimilar.simps by presburger
next
  case (notIfNode g ifcond conds stamps)
  then show ?case
    using strong-noop-bisimilar.simps by presburger
qed


experiment begin



lemma inverse-succ:
  ∀ n′ ∈ (succ g n). n ∈ ids g ⟶ n ∈ (predecessors g n′)
  by simp

lemma sequential-successors:
  assumes is-sequential-node n
  shows successors-of n ≠ []
  using assms by (cases n; auto)

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
  using assms(3) proof (induction (nid0, m0, h0) (nid, m, h) rule: step.induct)
  case SequentialNode
  then show ?case
    by (metis length-greater-0-conv nth-mem sequential-successors succ.simps)
next
  case (FixedGuardNode cond before val)
  then have {nid} = succ g nid0
    using IRNodes.successors-of-FixedGuardNode unfolding succ.simps
    by (metis empty-set list.simps(15))
  then show ?case
    using FixedGuardNode.hyps(5) by blast
next
  case (BytecodeExceptionNode args st exceptionType ref)
  then have {nid} = succ g nid0
    using IRNodes.successors-of-BytecodeExceptionNode unfolding succ.simps
    by (metis empty-set list.simps(15))
  then show ?case
    by blast
next
```

case (*IfNode cond tb fb val*)
then have {*tb*, *fb*} = *succ g nid0*
using *IRNodes.successors-of-IfNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by (*metis IfNode.hyps*(*3*) *insert-iff*)
next
case (*EndNodes i phis inps vs*)
then show *?case* **using** *assms*(*2*) **by** *blast*
next
case (*NewArrayNode len st length′ arrayType h′ ref refNo*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-NewArrayNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*ArrayLengthNode x ref arrayVal length′*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-ArrayLengthNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*LoadIndexedNode index guard array indexVal ref arrayVal loaded*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-LoadIndexedNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*StoreIndexedNode check val st index guard array indexVal ref value arrayVal updated*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-StoreIndexedNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*NewInstanceNode cname obj ref*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-NewInstanceNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*LoadFieldNode f obj ref*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-LoadFieldNode* **unfolding** *succ.simps*

```
    by (metis empty-set list.simps(15))
  then show ?case
    by blast
next
  case (SignedDivNode x y zero sb v1 v2)
  then have {nid} = succ g nid0
    using IRNodes.successors-of-SignedDivNode unfolding succ.simps
    by (metis empty-set list.simps(15))
  then show ?case
    by blast
next
  case (SignedRemNode x y zero sb v1 v2)
  then have {nid} = succ g nid0
    using IRNodes.successors-of-SignedRemNode unfolding succ.simps
    by (metis empty-set list.simps(15))
  then show ?case
    by blast
next
  case (StaticLoadFieldNode f)
  then have {nid} = succ g nid0
    using IRNodes.successors-of-LoadFieldNode unfolding succ.simps
    by (metis empty-set list.simps(15))
  then show ?case
    by blast
next
  case (StoreFieldNode - - - - - -)
  then have {nid} = succ g nid0
    using IRNodes.successors-of-StoreFieldNode unfolding succ.simps
    by (metis empty-set list.simps(15))
  then show ?case
    by blast
next
  case (StaticStoreFieldNode - - - -)
  then have {nid} = succ g nid0
    using IRNodes.successors-of-StoreFieldNode unfolding succ.simps
    by (metis empty-set list.simps(15))
  then show ?case
    by blast
qed

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
  using assms
  by (meson inverse-succ nid'-succ step-in-ids)

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
  using assms unfolding pred.simps sorry

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′)
  using assms sorry

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 −
  obtain tb fb where [tb, fb] = successors-of (kind g ifcond)
    by (simp add: assms(3))
  have nid0 = ifcond
    using assms step.IfNode sorry
  show ?thesis sorry
qed



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
  sorry termination sorry
```

```
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
  sorry termination sorry

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
```

$[\![(d = n) \vee ((pred\ g\ n \neq \{\}) \wedge (\forall p \in pred\ g\ n\ .\ (g \vdash d \geq\geq p)))]\!] \Longrightarrow dominates\ g\ d\ n$
code-pred [*show-modes*] *dominates* **.**

inductive *postdominates* :: *IRGraph* ⇒ *Block* ⇒ *Block* ⇒ *bool* (- ⊢ - ≤≤ - *20*) **where**
$[\![(z = n) \vee ((succ\ g\ n \neq \{\}) \wedge (\forall s \in succ\ g\ n\ .\ (g \vdash z \leq\leq s)))]\!] \Longrightarrow postdominates\ g\ z\ n$
code-pred [*show-modes*] *postdominates* **.**

inductive *strictly-dominates* :: *IRGraph* ⇒ *Block* ⇒ *Block* ⇒ *bool* (- ⊢ - >> - *20*) **where**
$[\![(g \vdash d \geq\geq n); (d \neq n)]\!] \Longrightarrow strictly\text{-}dominates\ g\ d\ n$
code-pred [*show-modes*] *strictly-dominates* **.**

inductive *strictly-postdominates* :: *IRGraph* ⇒ *Block* ⇒ *Block* ⇒ *bool* (- ⊢ - << - *20*) **where**
$[\![(g \vdash d \leq\leq n); (d \neq n)]\!] \Longrightarrow strictly\text{-}postdominates\ g\ d\ n$
code-pred [*show-modes*] *strictly-postdominates* **.**

lemma $pred\ g\ nid = \{\} \longrightarrow \neg(\exists\ d\ .\ (d \neq nid) \wedge (g \vdash d \geq\geq nid))$
using *dominates.cases* **by** *blast*

lemma $succ\ g\ nid = \{\} \longrightarrow \neg(\exists\ d\ .\ (d \neq nid) \wedge (g \vdash d \leq\leq nid))$
using *postdominates.cases* **by** *blast*

lemma $pred\ g\ nid = \{\} \longrightarrow \neg(\exists\ d\ .\ (g \vdash d >> nid))$
using *dominates.simps strictly-dominates.simps* **by** *presburger*

lemma $succ\ g\ nid = \{\} \longrightarrow \neg(\exists\ d\ .\ (g \vdash d << nid))$
using *postdominates.simps strictly-postdominates.simps* **by** *presburger*

inductive *wf-cfg* :: *IRGraph* ⇒ *bool* **where**
$[\![\forall\ nid \in ids\ g\ .\ (blockOf\ g\ nid \neq NoBlock) \longrightarrow (g \vdash (blockOf\ g\ 0) \geq\geq (blockOf\ g\ nid))]\!]$
$\Longrightarrow wf\text{-}cfg\ g$
code-pred [*show-modes*] *wf-cfg* **.**

inductive *immediately-dominates* :: *IRGraph* ⇒ *Block* ⇒ *Block* ⇒ *bool* (- ⊢ - *idom* - *20*) **where**
$[\![(g \vdash d >> n); (\forall\ w \in ids\ g\ .\ (g \vdash (blockOf\ g\ w) >> n) \longrightarrow (g \vdash (blockOf\ g\ w) \geq\geq d))]\!] \Longrightarrow immediately\text{-}dominates\ g\ d\ n$
code-pred [*show-modes*] *immediately-dominates* **.**

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