Unspecified Veriopt Theory

April 23, 2021

Contents

1 Canonicalization Phase **1**

2 Conditional Elimination Phase **23**
2.1 Individual Elimination Rules 23
2.2 Control-flow Graph Traversal 32

3 Graph Construction Phase **41**

1 Canonicalization Phase

theory *Canonicalization*
imports
Proofs.IRGraphFrames
Proofs.Stuttering
Proofs.Bisimulation
Proofs.Form

Graph.Traversal
begin

inductive *CanonicalizeConditional* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
where
negate-condition:
⟦*kind g cond* = *LogicNegationNode flip*⟧
⟹ *CanonicalizeConditional g* (*ConditionalNode cond tb fb*) (*ConditionalNode flip fb tb*) |

const-true:
⟦*kind g cond* = *ConstantNode val*;
val-to-bool val⟧
⟹ *CanonicalizeConditional g* (*ConditionalNode cond tb fb*) (*RefNode tb*) |

const-false:
⟦*kind g cond* = *ConstantNode val*;

$\neg(\textit{val-to-bool val})]\!]$
$\Longrightarrow \textit{CanonicalizeConditional } g \ (\textit{ConditionalNode cond tb fb}) \ (\textit{RefNode fb}) \mid$

eq-branches:
$[\![tb = fb]\!]$
$\Longrightarrow \textit{CanonicalizeConditional } g \ (\textit{ConditionalNode cond tb fb}) \ (\textit{RefNode tb}) \mid$

cond-eq:
$[\![\textit{kind } g \textit{ cond} = \textit{IntegerEqualsNode tb fb}]\!]$
$\Longrightarrow \textit{CanonicalizeConditional } g \ (\textit{ConditionalNode cond tb fb}) \ (\textit{RefNode fb}) \mid$

condition-bounds-x:
$[\![\textit{kind } g \textit{ cond} = \textit{IntegerLessThanNode tb fb};$
$\textit{stpi-upper } (\textit{stamp } g \textit{ tb}) \leq \textit{stpi-lower } (\textit{stamp } g \textit{ fb})]\!]$
$\Longrightarrow \textit{CanonicalizeConditional } g \ (\textit{ConditionalNode cond tb fb}) \ (\textit{RefNode tb}) \mid$

condition-bounds-y:
$[\![\textit{kind } g \textit{ cond} = \textit{IntegerLessThanNode fb tb};$
$\textit{stpi-upper } (\textit{stamp } g \textit{ fb}) \leq \textit{stpi-lower } (\textit{stamp } g \textit{ tb})]\!]$
$\Longrightarrow \textit{CanonicalizeConditional } g \ (\textit{ConditionalNode cond tb fb}) \ (\textit{RefNode tb})$

inductive *CanonicalizeAdd* :: $\textit{IRGraph} \Rightarrow \textit{IRNode} \Rightarrow \textit{IRNode} \Rightarrow \textit{bool}$
for *g* **where**
add-both-const:
$[\![\textit{kind } g \ x = \textit{ConstantNode c-1};$
$\textit{kind } g \ y = \textit{ConstantNode c-2};$
$\textit{val} = \textit{intval-add c-1 c-2}]\!]$
$\Longrightarrow \textit{CanonicalizeAdd } g \ (\textit{AddNode } x \ y) \ (\textit{ConstantNode val}) \mid$

add-xzero:
$[\![\textit{kind } g \ x = \textit{ConstantNode c-1};$
$\neg(\textit{is-ConstantNode } (\textit{kind } g \ y));$
$\textit{c-1} = (\textit{IntVal 32 0})]\!]$
$\Longrightarrow \textit{CanonicalizeAdd } g \ (\textit{AddNode } x \ y) \ (\textit{RefNode } y) \mid$

add-yzero:
$[\![\neg(\textit{is-ConstantNode } (\textit{kind } g \ x));$
$\textit{kind } g \ y = \textit{ConstantNode c-2};$
$\textit{c-2} = (\textit{IntVal 32 0})]\!]$
$\Longrightarrow \textit{CanonicalizeAdd } g \ (\textit{AddNode } x \ y) \ (\textit{RefNode } x) \mid$

add-xsub:

$[\![kind\ g\ x = SubNode\ a\ y\ ]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ y)\ (RefNode\ a)$ |

add-ysub:

$[\![kind\ g\ y = SubNode\ a\ x\ ]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ y)\ (RefNode\ a)$ |

add-xnegate:

$[\![kind\ g\ nx = NegateNode\ x\ ]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ nx\ y)\ (SubNode\ y\ x)$ |

add-ynegate:

$[\![kind\ g\ ny = NegateNode\ y\ ]\!]$
$\Longrightarrow CanonicalizeAdd\ g\ (AddNode\ x\ ny)\ (SubNode\ x\ y)$

inductive *CanonicalizeIf* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
trueConst:
$[\![kind\ g\ cond = ConstantNode\ condv;$
$val\text{-}to\text{-}bool\ condv]\!]$
$\Longrightarrow CanonicalizeIf\ g\ (IfNode\ cond\ tb\ fb)\ (RefNode\ tb)$ |

falseConst:
$[\![kind\ g\ cond = ConstantNode\ condv;$
$\neg(val\text{-}to\text{-}bool\ condv)]\!]$
$\Longrightarrow CanonicalizeIf\ g\ (IfNode\ cond\ tb\ fb)\ (RefNode\ fb)$ |

eqBranch:
$[\![\neg(is\text{-}ConstantNode\ (kind\ g\ cond));$
$tb = fb]\!]$
$\Longrightarrow CanonicalizeIf\ g\ (IfNode\ cond\ tb\ fb)\ (RefNode\ tb)$ |

eqCondition:
$[\![kind\ g\ cond = IntegerEqualsNode\ x\ x]\!]$
$\Longrightarrow CanonicalizeIf\ g\ (IfNode\ cond\ tb\ fb)\ (RefNode\ tb)$

inductive *CanonicalizeBinaryArithmeticNode* :: *ID* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**

add-const-fold:
⟦op = *kind g op-id*;
is-AddNode op;
kind g (*ir-x op*) = *ConditionalNode cond tb fb*;
kind g tb = *ConstantNode c-1*;
kind g fb = *ConstantNode c-2*;
kind g (*ir-y op*) = *ConstantNode c-3*;
tv = *intval-add c-1 c-3*;
fv = *intval-add c-2 c-3*;
g' = *replace-node tb* ((*ConstantNode tv*), *constantAsStamp tv*) *g*;
g'' = *replace-node fb* ((*ConstantNode fv*), *constantAsStamp fv*) g';
g''' = *replace-node op-id* (*kind g* (*ir-x op*), *meet* (*constantAsStamp tv*) (*constantAsStamp fv*)) g'' ⟧
$\Longrightarrow$ *CanonicalizeBinaryArithmeticNode op-id g* g'''

inductive *CanonicalizeCommutativeBinaryArithmeticNode* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**

add-ids-ordered:
⟦¬(*is-ConstantNode* (*kind g y*));
((*is-ConstantNode* (*kind g x*)) ∨ ($x > y$))⟧
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AddNode x y*) (*AddNode y x*) |

and-ids-ordered:
⟦¬(*is-ConstantNode* (*kind g y*));
((*is-ConstantNode* (*kind g x*)) ∨ ($x > y$))⟧
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AndNode x y*) (*AndNode y x*) |

int-equals-ids-ordered:
⟦¬(*is-ConstantNode* (*kind g y*));
((*is-ConstantNode* (*kind g x*)) ∨ ($x > y$))⟧
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*IntegerEqualsNode x y*) (*IntegerEqualsNode y x*) |

mul-ids-ordered:
⟦¬(*is-ConstantNode* (*kind g y*));
((*is-ConstantNode* (*kind g x*)) ∨ ($x > y$))⟧
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*MulNode x y*) (*MulNode y x*) |

or-ids-ordered:
⟦¬(*is-ConstantNode* (*kind g y*));
((*is-ConstantNode* (*kind g x*)) ∨ ($x > y$))⟧

$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*OrNode x y*) (*OrNode y x*) |

xor-ids-ordered:
$[\![\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y));$
$((\textit{is-ConstantNode}\ (\textit{kind}\ g\ x)) \vee (x > y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*XorNode x y*) (*XorNode y x*) |

add-swap-const-first:
$[\![\textit{is-ConstantNode}\ (\textit{kind}\ g\ x);$
$\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AddNode x y*) (*AddNode y x*) |

and-swap-const-first:
$[\![\textit{is-ConstantNode}\ (\textit{kind}\ g\ x);$
$\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*AndNode x y*) (*AndNode y x*) |

int-equals-swap-const-first:
$[\![\textit{is-ConstantNode}\ (\textit{kind}\ g\ x);$
$\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*IntegerEqualsNode x y*) (*IntegerEqualsNode y x*) |

mul-swap-const-first:
$[\![\textit{is-ConstantNode}\ (\textit{kind}\ g\ x);$
$\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*MulNode x y*) (*MulNode y x*) |

or-swap-const-first:
$[\![\textit{is-ConstantNode}\ (\textit{kind}\ g\ x);$
$\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*OrNode x y*) (*OrNode y x*) |

xor-swap-const-first:
$[\![\textit{is-ConstantNode}\ (\textit{kind}\ g\ x);$
$\neg(\textit{is-ConstantNode}\ (\textit{kind}\ g\ y))]\!]$
$\Longrightarrow$ *CanonicalizeCommutativeBinaryArithmeticNode g* (*XorNode x y*) (*XorNode y x*)

inductive *CanonicalizeSub* :: *IRGraph* $\Rightarrow$ *IRNode* $\Rightarrow$ *IRNode* $\Rightarrow$ *bool*
for *g* **where**

sub-same:
$[\![x = y;$
$\quad stamp\ g\ x = (IntegerStamp\ b\ l\ h)]\!]$
$\quad \Longrightarrow CanonicalizeSub\ g\ (SubNode\ x\ y)\ (ConstantNode\ (IntVal\ b\ 0))$ |

sub-both-const:
$[\![kind\ g\ x = ConstantNode\ c\text{-}1;$
$\quad kind\ g\ y = ConstantNode\ c\text{-}2;$
$\quad val = intval\text{-}sub\ c\text{-}1\ c\text{-}2]\!]$
$\quad \Longrightarrow CanonicalizeSub\ g\ (SubNode\ x\ y)\ (ConstantNode\ val)$ |

sub-left-add1:

$[\![kind\ g\ left = AddNode\ a\ b]\!]$
$\quad \Longrightarrow CanonicalizeSub\ g\ (SubNode\ left\ b)\ (RefNode\ a)$ |

sub-left-add2:

$[\![kind\ g\ left = AddNode\ a\ b]\!]$
$\quad \Longrightarrow CanonicalizeSub\ g\ (SubNode\ left\ a)\ (RefNode\ b)$ |

sub-left-sub:

$[\![kind\ g\ left = SubNode\ a\ b]\!]$
$\quad \Longrightarrow CanonicalizeSub\ g\ (SubNode\ left\ a)\ (NegateNode\ b)$ |

sub-right-add1:

$[\![kind\ g\ right = AddNode\ a\ b]\!]$
$\quad \Longrightarrow CanonicalizeSub\ g\ (SubNode\ a\ right)\ (NegateNode\ b)$ |

sub-right-add2:

$[\![kind\ g\ right = AddNode\ a\ b]\!]$
$\quad \Longrightarrow CanonicalizeSub\ g\ (SubNode\ b\ right)\ (NegateNode\ a)$ |

sub-right-sub:

$[\![kind\ g\ right = AddNode\ a\ b]\!]$
$\quad \Longrightarrow CanonicalizeSub\ g\ (SubNode\ a\ right)\ (RefNode\ a)$ |

sub-yzero:

$[\![kind\ g\ y = ConstantNode\ (IntVal\ \text{-}\ 0)]\!]$
$\quad \Longrightarrow CanonicalizeSub\ g\ (SubNode\ x\ y)\ (RefNode\ x)$ |

sub-xzero:

$[\![kind\ g\ x = ConstantNode\ (IntVal\ \text{-}\ 0)]\!]$

$\Longrightarrow$ *CanonicalizeSub g* (*SubNode x y*) (*NegateNode y*) |

sub-y-negate:

$[\![$*kind g nb* = *NegateNode b*$]\!]$
$\Longrightarrow$ *CanonicalizeSub g* (*SubNode a nb*) (*AddNode a b*)

inductive *CanonicalizeMul* :: *IRGraph* $\Rightarrow$ *IRNode* $\Rightarrow$ *IRNode* $\Rightarrow$ *bool*
for *g* **where**
mul-both-const:
$[\![$*kind g x* = *ConstantNode c-1*;
kind g y = *ConstantNode c-2*;
val = *intval-mul c-1 c-2*$]\!]$
$\Longrightarrow$ *CanonicalizeMul g* (*MulNode x y*) (*ConstantNode val*) |

mul-xzero:
$[\![$*kind g x* = *ConstantNode c-1*;
$\neg$(*is-ConstantNode* (*kind g y*));
c-1 = (*IntVal b 0*)$]\!]$
$\Longrightarrow$ *CanonicalizeMul g* (*MulNode x y*) (*ConstantNode c-1*) |

mul-yzero:
$[\![$*kind g y* = *ConstantNode c-1*;
$\neg$(*is-ConstantNode* (*kind g x*));
c-1 = (*IntVal b 0*)$]\!]$
$\Longrightarrow$ *CanonicalizeMul g* (*MulNode x y*) (*ConstantNode c-1*) |

mul-xone:
$[\![$*kind g x* = *ConstantNode c-1*;
$\neg$(*is-ConstantNode* (*kind g y*));
c-1 = (*IntVal b 1*)$]\!]$
$\Longrightarrow$ *CanonicalizeMul g* (*MulNode x y*) (*RefNode y*) |

mul-yone:
$[\![$*kind g y* = *ConstantNode c-1*;
$\neg$(*is-ConstantNode* (*kind g x*));
c-1 = (*IntVal b 1*)$]\!]$
$\Longrightarrow$ *CanonicalizeMul g* (*MulNode x y*) (*RefNode x*) |

mul-xnegate:
$[\![$*kind g x* = *ConstantNode c-1*;
$\neg$(*is-ConstantNode* (*kind g y*));
c-1 = (*IntVal b* ($-$*1*))$]\!]$
$\Longrightarrow$ *CanonicalizeMul g* (*MulNode x y*) (*NegateNode y*) |

mul-ynegate:
$[\![$*kind g y* = *ConstantNode c-1*;

$\neg(is\text{-}ConstantNode\ (kind\ g\ x));$
$c\text{-}1 = (IntVal\ b\ (-1))]\!]$
$\Longrightarrow CanonicalizeMul\ g\ (MulNode\ x\ y)\ (NegateNode\ x)$

inductive *CanonicalizeAbs* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
abs-abs:
$[\![kind\ g\ x = (AbsNode\ y)]\!]$
$\Longrightarrow CanonicalizeAbs\ g\ (AbsNode\ x)\ (AbsNode\ y)$ |

abs-negate:
$[\![kind\ g\ nx = (NegateNode\ x)]\!]$
$\Longrightarrow CanonicalizeAbs\ g\ (AbsNode\ nx)\ (AbsNode\ x)$

inductive *CanonicalizeNegate* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
negate-const:
$[\![kind\ g\ nx = (ConstantNode\ val);$
$val = (IntVal\ b\ v);$
$neg\text{-}val = intval\text{-}sub\ (IntVal\ b\ 0)\ val\ ]\!]$
$\Longrightarrow CanonicalizeNegate\ g\ (NegateNode\ nx)\ (ConstantNode\ neg\text{-}val)$ |

negate-negate:
$[\![kind\ g\ nx = (NegateNode\ x)]\!]$
$\Longrightarrow CanonicalizeNegate\ g\ (NegateNode\ nx)\ (RefNode\ x)$ |

negate-sub:
$[\![kind\ g\ sub = (SubNode\ x\ y);$
$stamp\ g\ sub = (IntegerStamp\ \text{-}\ \text{-}\ \text{-})]\!]$
$\Longrightarrow CanonicalizeNegate\ g\ (NegateNode\ sub)\ (SubNode\ y\ x)$

inductive *CanonicalizeNot* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
not-const:
$[\![kind\ g\ nx = (ConstantNode\ val);$
$neg\text{-}val = bool\text{-}to\text{-}val\ (\neg(val\text{-}to\text{-}bool\ val))\ ]\!]$
$\Longrightarrow CanonicalizeNot\ g\ (NotNode\ nx)\ (ConstantNode\ neg\text{-}val)$ |

not-not:
$[\![kind\ g\ nx = (NotNode\ x)]\!]$
$\Longrightarrow CanonicalizeNot\ g\ (NotNode\ nx)\ (RefNode\ x)$

inductive $CanonicalizeAnd :: IRGraph \Rightarrow IRNode \Rightarrow IRNode \Rightarrow bool$
for g **where**
and-same:
$[\![x = y]\!]$
$\Longrightarrow CanonicalizeAnd\ g\ (AndNode\ x\ y)\ (RefNode\ x)\ |$

and-xtrue:
$[\![kind\ g\ x = ConstantNode\ val;$
$val\text{-}to\text{-}bool\ val]\!]$
$\Longrightarrow CanonicalizeAnd\ g\ (AndNode\ x\ y)\ (RefNode\ y)\ |$

and-ytrue:
$[\![kind\ g\ y = ConstantNode\ val;$
$val\text{-}to\text{-}bool\ val]\!]$
$\Longrightarrow CanonicalizeAnd\ g\ (AndNode\ x\ y)\ (RefNode\ x)\ |$

and-xfalse:
$[\![kind\ g\ x = ConstantNode\ val;$
$\neg(val\text{-}to\text{-}bool\ val)]\!]$
$\Longrightarrow CanonicalizeAnd\ g\ (AndNode\ x\ y)\ (ConstantNode\ val)\ |$

and-yfalse:
$[\![kind\ g\ y = ConstantNode\ val;$
$\neg(val\text{-}to\text{-}bool\ val)]\!]$
$\Longrightarrow CanonicalizeAnd\ g\ (AndNode\ x\ y)\ (ConstantNode\ val)$

inductive $CanonicalizeOr :: IRGraph \Rightarrow IRNode \Rightarrow IRNode \Rightarrow bool$
for g **where**
or-same:
$[\![x = y]\!]$
$\Longrightarrow CanonicalizeOr\ g\ (OrNode\ x\ y)\ (RefNode\ x)\ |$

or-xtrue:
$[\![kind\ g\ x = ConstantNode\ val;$
$val\text{-}to\text{-}bool\ val]\!]$
$\Longrightarrow CanonicalizeOr\ g\ (OrNode\ x\ y)\ (ConstantNode\ val)\ |$

or-ytrue:
$[\![kind\ g\ y = ConstantNode\ val;$
$val\text{-}to\text{-}bool\ val]\!]$
$\Longrightarrow CanonicalizeOr\ g\ (OrNode\ x\ y)\ (ConstantNode\ val)\ |$

or-xfalse:
$[\![kind\ g\ x = ConstantNode\ val;$
$\neg(val\text{-}to\text{-}bool\ val)]\!]$
$\Longrightarrow CanonicalizeOr\ g\ (OrNode\ x\ y)\ (RefNode\ y)\ |$

or-yfalse:
$[\![kind\ g\ y = ConstantNode\ val;$
$\neg(val\text{-}to\text{-}bool\ val)]\!]$
$\Longrightarrow CanonicalizeOr\ g\ (OrNode\ x\ y)\ (RefNode\ x)$

inductive *CanonicalizeDeMorgansLaw* :: *ID* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool*
where

de-morgan-or-to-and:
$[\![kind\ g\ nid = OrNode\ nx\ ny;$
$kind\ g\ nx = NotNode\ x;$
$kind\ g\ ny = NotNode\ y;$
$new\text{-}add\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ new\text{-}add\text{-}id\ ((AddNode\ x\ y),\ (IntegerStamp\ 1\ 0\ 1))\ g;$
$g'' = replace\text{-}node\ nid\ ((NotNode\ new\text{-}add\text{-}id),\ (IntegerStamp\ 1\ 0\ 1))\ g']\!]$
$\Longrightarrow CanonicalizeDeMorgansLaw\ nid\ g\ g''$ |

de-morgan-and-to-or:
$[\![kind\ g\ nid = AndNode\ nx\ ny;$
$kind\ g\ nx = NotNode\ x;$
$kind\ g\ ny = NotNode\ y;$
$new\text{-}add\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ new\text{-}add\text{-}id\ ((OrNode\ x\ y),\ (IntegerStamp\ 1\ 0\ 1))\ g;$
$g'' = replace\text{-}node\ nid\ ((NotNode\ new\text{-}add\text{-}id),\ (IntegerStamp\ 1\ 0\ 1))\ g']\!]$
$\Longrightarrow CanonicalizeDeMorgansLaw\ nid\ g\ g''$

inductive *CanonicalizeIntegerEquals* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
int-equals-same-node:
$[\![x = y]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ g\ (IntegerEqualsNode\ x\ y)\ (ConstantNode\ (IntVal\ 1\ 1))$ |

int-equals-distinct:
$[\![alwaysDistinct\ (stamp\ g\ x)\ (stamp\ g\ y)]\!]$
$\Longrightarrow CanonicalizeIntegerEquals\ g\ (IntegerEqualsNode\ x\ y)\ (ConstantNode\ (IntVal\ 1\ 0))$ |

int-equals-add-first-both-same:

$[\![kind\ g\ left = AddNode\ x\ y;$
$kind\ g\ right = AddNode\ x\ z]\!]$

$\Longrightarrow$ *CanonicalizeIntegerEquals g* (*IntegerEqualsNode left right*) (*IntegerEqualsNode y z*) |

int-equals-add-first-second-same:

$[\![$*kind g left* = *AddNode x y*;
kind g right = *AddNode z x*$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals g* (*IntegerEqualsNode left right*) (*IntegerEqualsNode y z*) |

int-equals-add-second-first-same:

$[\![$*kind g left* = *AddNode y x*;
kind g right = *AddNode x z*$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals g* (*IntegerEqualsNode left right*) (*IntegerEqualsNode y z*) |

int-equals-add-second-both--same:

$[\![$*kind g left* = *AddNode y x*;
kind g right = *AddNode z x*$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals g* (*IntegerEqualsNode left right*) (*IntegerEqualsNode y z*) |

int-equals-sub-first-both-same:

$[\![$*kind g left* = *SubNode x y*;
kind g right = *SubNode x z*$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals g* (*IntegerEqualsNode left right*) (*IntegerEqualsNode y z*) |

int-equals-sub-second-both-same:

$[\![$*kind g left* = *SubNode y x*;
kind g right = *SubNode z x*$]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEquals g* (*IntegerEqualsNode left right*) (*IntegerEqualsNode y z*)

inductive *CanonicalizeIntegerEqualsGraph* :: *ID* $\Rightarrow$ *IRGraph* $\Rightarrow$ *IRGraph* $\Rightarrow$ *bool*
where
int-equals-rewrite:
$[\![$*CanonicalizeIntegerEquals g node node'*;
node = *kind g nid*;
g' = *replace-node nid* (*node'*, *stamp g nid*) *g* $]\!]$
$\Longrightarrow$ *CanonicalizeIntegerEqualsGraph nid g g'* |

int-equals-left-contains-right1:
$[\![kind\ g\ nid = IntegerEqualsNode\ left\ x;$
$kind\ g\ left = AddNode\ x\ y;$
$const\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ const\text{-}id\ ((ConstantNode\ (IntVal\ 1\ 0)),\ constantAsStamp\ (IntVal\ 1\ 0))\ g;$
$g'' = replace\text{-}node\ const\text{-}id\ ((IntegerEqualsNode\ y\ const\text{-}id),\ stamp\ g\ nid)\ g']\!]$
$\Longrightarrow CanonicalizeIntegerEqualsGraph\ nid\ g\ g''\ |$

int-equals-left-contains-right2:
$[\![kind\ g\ nid = IntegerEqualsNode\ left\ y;$
$kind\ g\ left = AddNode\ x\ y;$
$const\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ const\text{-}id\ ((ConstantNode\ (IntVal\ 1\ 0)),\ constantAsStamp\ (IntVal\ 1\ 0))\ g;$
$g'' = replace\text{-}node\ const\text{-}id\ ((IntegerEqualsNode\ x\ const\text{-}id),\ stamp\ g\ nid)\ g']\!]$
$\Longrightarrow CanonicalizeIntegerEqualsGraph\ nid\ g\ g''\ |$

int-equals-right-contains-left1:
$[\![kind\ g\ nid = IntegerEqualsNode\ x\ right;$
$kind\ g\ right = AddNode\ x\ y;$
$const\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ const\text{-}id\ ((ConstantNode\ (IntVal\ 1\ 0)),\ constantAsStamp\ (IntVal\ 1\ 0))\ g;$
$g'' = replace\text{-}node\ const\text{-}id\ ((IntegerEqualsNode\ y\ const\text{-}id),\ stamp\ g\ nid)\ g']\!]$
$\Longrightarrow CanonicalizeIntegerEqualsGraph\ nid\ g\ g''\ |$

int-equals-right-contains-left2:
$[\![kind\ g\ nid = IntegerEqualsNode\ y\ right;$
$kind\ g\ right = AddNode\ x\ y;$
$const\text{-}id = nextNid\ g;$
$g' = add\text{-}node\ const\text{-}id\ ((ConstantNode\ (IntVal\ 1\ 0)),\ constantAsStamp\ (IntVal\ 1\ 0))\ g;$
$g'' = replace\text{-}node\ const\text{-}id\ ((IntegerEqualsNode\ x\ const\text{-}id),\ stamp\ g\ nid)\ g']\!]$
$\Longrightarrow CanonicalizeIntegerEqualsGraph\ nid\ g\ g''\ |$

int-equals-left-contains-right3:
$[\![kind\ g\ nid = IntegerEqualsNode\ left\ x;$
$kind\ g\ left = SubNode\ x\ y;$
$const\text{-}id = nextNid\ g;$

$g' = \textit{add-node const-id } ((\textit{ConstantNode } (\textit{IntVal } 1\ 0)), \textit{constantAsStamp } (\textit{IntVal } 1\ 0))\ g;$
$g'' = \textit{replace-node const-id } ((\textit{IntegerEqualsNode } y \textit{ const-id}), \textit{stamp } g \textit{ nid})\ g'\rrbracket$
$\Longrightarrow \textit{CanonicalizeIntegerEqualsGraph nid } g\ g''\ |$

int-equals-right-contains-left3:
$\llbracket \textit{kind } g \textit{ nid} = \textit{IntegerEqualsNode } x \textit{ right};$
$\textit{kind } g \textit{ right} = \textit{SubNode } x\ y;$
$\textit{const-id} = \textit{nextNid } g;$
$g' = \textit{add-node const-id } ((\textit{ConstantNode } (\textit{IntVal } 1\ 0)), \textit{constantAsStamp } (\textit{IntVal } 1\ 0))\ g;$
$g'' = \textit{replace-node const-id } ((\textit{IntegerEqualsNode } y \textit{ const-id}), \textit{stamp } g \textit{ nid})\ g'\rrbracket$
$\Longrightarrow \textit{CanonicalizeIntegerEqualsGraph nid } g\ g''$

inductive *CanonicalizationStep* :: *IRGraph* ⇒ *IRNode* ⇒ *IRNode* ⇒ *bool*
for *g* **where**
ConditionalNode:
$\llbracket \textit{CanonicalizeConditional } g \textit{ node node}'\rrbracket$
$\Longrightarrow \textit{CanonicalizationStep } g \textit{ node node}'\ |$

AddNode:
$\llbracket \textit{CanonicalizeAdd } g \textit{ node node}'\rrbracket$
$\Longrightarrow \textit{CanonicalizationStep } g \textit{ node node}'\ |$

IfNode:
$\llbracket \textit{CanonicalizeIf } g \textit{ node node}'\rrbracket$
$\Longrightarrow \textit{CanonicalizationStep } g \textit{ node node}'\ |$

SubNode:
$[\![CanonicalizeSub\ g\ node\ node']\!]$
$\Longrightarrow CanonicalizationStep\ g\ node\ node'$ |

MulNode:
$[\![CanonicalizeMul\ g\ node\ node']\!]$
$\Longrightarrow CanonicalizationStep\ g\ node\ node'$ |

AndNode:
$[\![CanonicalizeAnd\ g\ node\ node']\!]$
$\Longrightarrow CanonicalizationStep\ g\ node\ node'$ |

OrNode:
$[\![CanonicalizeOr\ g\ node\ node']\!]$
$\Longrightarrow CanonicalizationStep\ g\ node\ node'$ |

AbsNode:
$[\![CanonicalizeAbs\ g\ node\ node']\!]$
$\Longrightarrow CanonicalizationStep\ g\ node\ node'$ |

NotNode:
$[\![CanonicalizeNot\ g\ node\ node']\!]$
$\Longrightarrow CanonicalizationStep\ g\ node\ node'$ |

Negatenode:
$[\![CanonicalizeNegate\ g\ node\ node']\!]$
$\Longrightarrow CanonicalizationStep\ g\ node\ node'$

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *CanonicalizeConditional* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *CanonicalizeAdd* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *CanonicalizeIf* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *CanonicalizeSub* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *CanonicalizeMul* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *CanonicalizeAnd* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *CanonicalizeOr* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *CanonicalizeAbs* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *CanonicalizeNot* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *CanonicalizeNegate* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *CanonicalizationStep* **.**

type-synonym *CanonicalizationAnalysis* = *bool option*

fun *analyse* :: $(ID \times Seen \times CanonicalizationAnalysis) \Rightarrow CanonicalizationAnalysis$
where
analyse i = *None*

inductive *CanonicalizationPhase*
:: $IRGraph \Rightarrow (ID \times Seen \times CanonicalizationAnalysis) \Rightarrow IRGraph \Rightarrow bool$ **where**

— Can do a step and optimise for the current node
$[\![Step\ analyse\ g\ (nid,\ seen,\ i)\ (Some\ (nid',\ seen',\ i'));$
$CanonicalizationStep\ g\ (kind\ g\ nid)\ node;$

$g' = replace\text{-}node\ nid\ (node,\ stamp\ g\ nid)\ g;$

$CanonicalizationPhase\ g'\ (nid',\ seen',\ i')\ g'']\!]$
$\Longrightarrow CanonicalizationPhase\ g\ (nid,\ seen,\ i)\ g''$ |

— Can do a step, matches whether optimised or not causing non-determinism We need to find a way to negate ConditionalEliminationStep
$[\![Step\ analyse\ g\ (nid,\ seen,\ i)\ (Some\ (nid',\ seen',\ i'));$

$CanonicalizationPhase\ g\ (nid',\ seen',\ i')\ g']\!]$
$\Longrightarrow CanonicalizationPhase\ g\ (nid,\ seen,\ i)\ g'$ |

$[\![Step\ analyse\ g\ (nid,\ seen,\ i)\ None;$
$Some\ nid' = pred\ g\ nid;$
$seen' = \{nid\} \cup seen;$
$CanonicalizationPhase\ g\ (nid',\ seen',\ i)\ g']\!]$
$\Longrightarrow CanonicalizationPhase\ g\ (nid,\ seen,\ i)\ g'$ |

$[\![Step\ analyse\ g\ (nid,\ seen,\ i)\ None;$
$None = pred\ g\ nid]\!]$
$\Longrightarrow CanonicalizationPhase\ g\ (nid,\ seen,\ i)\ g$

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

type-synonym $Trace = IRNode\ list$
inductive *CanonicalizationPhaseWithTrace*
:: $IRGraph \Rightarrow (ID \times Seen \times CanonicalizationAnalysis) \Rightarrow IRGraph \Rightarrow Trace \Rightarrow Trace \Rightarrow bool$ **where**

— Can do a step and optimise for the current node
$[\![Step\ analyse\ g\ (nid,\ seen,\ i)\ (Some\ (nid',\ seen',\ i'));$
$CanonicalizationStep\ g\ (kind\ g\ nid)\ node;$

$g' = replace\text{-}node\ nid\ (node,\ stamp\ g\ nid)\ g;$

$CanonicalizationPhaseWithTrace\ g'\ (nid',\ seen',\ i')\ g''\ (kind\ g\ nid\ \#\ t)\ t'\ ]\!]$
$\Longrightarrow CanonicalizationPhaseWithTrace\ g\ (nid,\ seen,\ i)\ g''\ t\ t'$ |

— Can do a step, matches whether optimised or not causing non-determinism We need to find a way to negate ConditionalEliminationStep

⟦*Step analyse g* (*nid*, *seen*, *i*) (*Some* (*nid'*, *seen'*, *i'*));

CanonicalizationPhaseWithTrace g (*nid'*, *seen'*, *i'*) *g'* (*kind g nid* # *t*) *t'* ⟧
⟹ *CanonicalizationPhaseWithTrace g* (*nid*, *seen*, *i*) *g' t t'* |

⟦*Step analyse g* (*nid*, *seen*, *i*) *None*;
Some nid' = *pred g nid*;
seen' = {*nid*} ∪ *seen*;
CanonicalizationPhaseWithTrace g (*nid'*, *seen'*, *i*) *g'* (*kind g nid* # *t*) *t'* ⟧
⟹ *CanonicalizationPhaseWithTrace g* (*nid*, *seen*, *i*) *g' t t'* |

⟦*Step analyse g* (*nid*, *seen*, *i*) *None*;
None = *pred g nid*⟧
⟹ *CanonicalizationPhaseWithTrace g* (*nid*, *seen*, *i*) *g t t*

code-pred (*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *i* ⇒ *o* ⇒ *bool*) *CanonicalizationPhaseWithTrace*
.

end
theory
CanonicalizationProofs
imports
Canonicalization
begin

lemma *CanonicalizeConditionalProof*:
assumes *CanonicalizeConditional g before after*
assumes *wff-graph g* ∧ *wff-stamps g* ∧ *wff-values g*
assumes *g m* ⊢ *before* ↦ *res*
assumes *g m* ⊢ *after* ↦ *res'*
shows *res* = *res'*
using *assms*(*1*) *assms*
proof (*induct rule*: *CanonicalizeConditional.induct*)
case (*negate-condition g cond flip tb fb*)
obtain *condv* **where** *condv*: *g m* ⊢ *kind g cond* ↦ *IntVal 1 condv*
using *negate-condition.prems*(*3*) **by** *blast*
then obtain *flipv* **where** *flipv*: *g m* ⊢ *kind g flip* ↦ *IntVal 1 flipv*
by (*metis LogicNegationNodeE negate-condition.hyps*)
have *invert*: *condv* = *0* ⟷ (*NOT flipv*) = *0*
using *eval.LogicNegationNode condv flipv*
by (*metis Value.inject*(*1*) *evalDet negate-condition.hyps*)
obtain *tbval* **where** *tbval*: *g m* ⊢ *kind g tb* ↦ *tbval*
using *negate-condition.prems*(*3*) **by** *blast*
obtain *fbval* **where** *fbval*: *g m* ⊢ *kind g fb* ↦ *fbval*
using *negate-condition.prems*(*3*) **by** *blast*
show *?case* **proof** (*cases condv* = *0*)

case *True*
have *flipv* $\neq$ *0*
using *eval.LogicNegationNode condv flipv*
using *True evalDet negate-condition.hyps* **by** *fastforce*
then have *fbval* = *res*
using *eval.ConditionalNode tbval fbval flipv negate-condition*
by (*smt* (*verit*, *del-insts*) *ConditionalNodeE True Value.inject*(*1*) *condv evalDet*)
then show *?thesis*
by (*smt* (*verit*, *best*) *ConditionalNodeE True Value.inject*(*1*) *bit.compl-zero evalDet fbval flipv invert negate-condition.prems*(*4*))
next
case *False*
have *flipv-range*: *flipv* $\in$ {*0*, *1*}
using *assms*(*2*) *flipv wff-value-bit-range* **sorry**
have (*NOT flipv*) $\neq$ *0*
using *False invert* **by** *fastforce*
then have *flipv* $\neq$ *1*
using *not-eq-complement* **sorry**
then have *flipv* = *0*
using *flipv-range* **by** *auto*
then have *tbval* = *res*
using *eval.ConditionalNode tbval fbval flipv negate-condition*
by (*smt* (*verit*, *del-insts*) *ConditionalNodeE False Value.inject*(*1*) *condv evalDet*)
then show *?thesis*
using ‹*flipv* = *0*› *evalDet flipv negate-condition.prems*(*4*) *tbval* **by** *fastforce*
qed
next
case (*const-true g cond val tb fb*)
then show *?case*
using *eval.RefNode evalDet* **by** *force*
next
case (*const-false g cond val tb fb*)
then show *?case*
using *eval.RefNode evalDet* **by** *force*
next
case (*eq-branches tb fb g cond*)
then show *?case*
using *eval.RefNode evalDet* **by** *force*
next
case (*cond-eq g cond tb fb*)
then obtain *condv* **where** *condv*: *g m* $\vdash$ *kind g cond* $\mapsto$ *condv*
by *blast*
obtain *tbval* **where** *tbval*: *g m* $\vdash$ *kind g tb* $\mapsto$ *tbval*
using *cond-eq.prems*(*3*) **by** *blast*
obtain *fbval* **where** *fbval*: *g m* $\vdash$ *kind g fb* $\mapsto$ *fbval*
using *cond-eq.prems*(*3*) **by** *blast*
from *cond-eq* **show** *?case* **proof** (*cases val-to-bool condv*)
case *True*

have *tbval* = *fbval* **using** *IntegerEqualsNodeE condv cond-eq*(*1*)
by (*smt* (*z3*) *True bool-to-val.simps*(*2*) *evalDet fbval tbval val-to-bool.simps*(*1*))
then show *?thesis* **using** *cond-eq*
by (*smt* (*verit*, *ccfv-threshold*) *ConditionalNodeE eval.RefNode evalDet fbval tbval*)
next
case *False*
then show *?thesis*
by (*smt* (*verit*) *ConditionalNodeE cond-eq.prems*(*3*) *cond-eq.prems*(*4*) *condv eval.RefNode evalDet val-to-bool.simps*(*1*))
qed
next
case (*condition-bounds-x g cond tb fb*)
obtain *tbval b* **where** *tbval*: *g m* $\vdash$ *kind g tb* $\mapsto$ *IntVal b tbval*
using *condition-bounds-x.prems*(*3*) **by** *blast*
obtain *fbval b* **where** *fbval*: *g m* $\vdash$ *kind g fb* $\mapsto$ *IntVal b fbval*
using *condition-bounds-x.prems*(*3*) **by** *blast*
have *tbval* $\leq$ *fbval*
using *condition-bounds-x.prems*(*2*) *tbval fbval condition-bounds-x.hyps*(*2*) *int-valid-range*
unfolding *wff-stamps.simps*
by (*smt* (*verit*, *best*) *Stamp.sel*(*2*) *Stamp.sel*(*3*) *Value.inject*(*1*) *eval-in-ids valid-value.elims*(*2*) *valid-value.simps*(*3*))
then have *res* = *IntVal b tbval*
using *ConditionalNodeE tbval fbval*
by (*smt* (*verit*, *del-insts*) *IntegerLessThanNodeE Value.inject*(*1*) *bool-to-val.simps*(*1*) *condition-bounds-x.hyps*(*1*) *condition-bounds-x.prems*(*3*) *evalDet*)
then show *?case*
using *condition-bounds-x.prems*(*3*) *eval.RefNode evalDet tbval*
using *ConditionalNodeE Value.sel*(*1*) *condition-bounds-x.prems*(*4*) **by** *blast*
next
case (*condition-bounds-y g cond fb tb*)
obtain *tbval b* **where** *tbval*: *g m* $\vdash$ *kind g tb* $\mapsto$ *IntVal b tbval*
using *condition-bounds-y.prems*(*3*) **by** *blast*
obtain *fbval b* **where** *fbval*: *g m* $\vdash$ *kind g fb* $\mapsto$ *IntVal b fbval*
using *condition-bounds-y.prems*(*3*) **by** *blast*
have *tbval* $\geq$ *fbval*
using *condition-bounds-y.prems*(*2*) *tbval fbval condition-bounds-y.hyps*(*2*) *int-valid-range*
unfolding *wff-stamps.simps*
by (*smt* (*verit*, *ccfv-SIG*) *Stamp.disc*(*2*) *boundsAlwaysOverlap eval-in-ids valid-value.elims*(*2*) *valid-value.simps*(*3*))
then have *res* = *IntVal b tbval*
using *ConditionalNodeE tbval fbval*
by (*smt* (*verit*) *IntegerLessThanNodeE Value.inject*(*1*) *bool-to-val.simps*(*1*) *condition-bounds-y.hyps*(*1*) *condition-bounds-y.prems*(*3*) *evalDet*)
then show *?case*
using *condition-bounds-y.prems*(*3*) *eval.RefNode evalDet tbval*
using *ConditionalNodeE Value.sel*(*1*) *condition-bounds-y.prems*(*4*) **by** *blast*
qed

lemma *add-zero-32*:
  assumes *wff-value* (*IntVal 32 y*)
  shows (*IntVal 32 0*) $+*$ (*IntVal 32 y*) = (*IntVal 32 y*)
proof −
  have $-(2\hat{}31) \leq y \wedge y < 2\hat{}31$
    using *assms* **unfolding** *wff-value.simps* **by** *simp*
  then show *?thesis* **unfolding** *intval-add.simps* **apply** *auto*
    using ‹$-(2 \hat{} 31) \leq y \wedge y < 2 \hat{} 31$› *signed-take-bit-int-eq-self* **by** *blast*
qed

lemma *add-zero-64*:
  assumes *wff-value* (*IntVal 64 y*)
  shows (*IntVal 64 0*) $+*$ (*IntVal 64 y*) = (*IntVal 64 y*)
proof −
  have $-(2\hat{}63) \leq y \wedge y < 2\hat{}63$
    using *assms* **unfolding** *wff-value.simps* **by** *simp*
  then show *?thesis* **unfolding** *intval-add.simps* **apply** *auto*
    using ‹$-(2 \hat{} 63) \leq y \wedge y < 2 \hat{} 63$› *signed-take-bit-int-eq-self* **by** *blast*
qed

lemma
  assumes *wff-value* (*IntVal bc y*)
  assumes $bc \in \{32,64\}$
  shows (*IntVal bc 0*) $+*$ (*IntVal bc y*) = (*IntVal bc y*)
proof −
  have *bounds*: $-(2\hat{}((nat\ bc)-1)) \leq y \wedge y < 2\hat{}((nat\ bc)-1)$
    using *assms* **unfolding** *wff-value.simps* **by** *auto*
  then show *?thesis* **unfolding** *intval-add.simps* **apply** *auto*
    using *bounds signed-take-bit-int-eq-self assms*
    by *auto*
qed

lemma
  assumes *wff-value* (*IntVal b x*) $\wedge$ *wff-value* (*IntVal b y*)
  shows ((*IntVal b 0*) $-*$ (*IntVal b x*)) $+*$ (*IntVal b y*) = (*IntVal b y*) $-*$ (*IntVal b x*)
  using *assms* **unfolding** *wff-value.simps* **by** *simp*

lemma *CanonicalizeAddProof*:
  assumes *CanonicalizeAdd g before after*
  assumes *wff-graph g* $\wedge$ *wff-stamps g* $\wedge$ *wff-values g*
  assumes $g\ m \vdash before \mapsto IntVal\ b\ res$
  assumes $g\ m \vdash after \mapsto IntVal\ b'\ res'$
  shows $res = res'$
proof −

```
  obtain x y where addkind: before = AddNode x y
    using CanonicalizeAdd.simps assms by auto
  from addkind
  obtain xval where xval: g m ⊢ kind g x ↦ xval
    using assms(3) by blast
  from addkind
  obtain yval where yval: g m ⊢ kind g y ↦ yval
    using assms(3) by blast
  have res: IntVal b res = intval-add xval yval
    using assms(3) eval.AddNode
    using addkind evalDet xval yval by presburger
  show ?thesis
    using assms addkind xval yval res
  proof (induct rule: CanonicalizeAdd.induct)
case (add-both-const x c-1 y c-2 val)
  then show ?case using eval.ConstantNode
    by (metis ConstantNodeE IRNode.inject(2) Value.inject(1))
next
  case (add-xzero x c-1 y)
  have xeval: g m ⊢ kind g x ↦ (IntVal 32 0)
    by (simp add: ConstantNode add-xzero.hyps(1) add-xzero.hyps(3))
  have yeval: g m ⊢ kind g y ↦ yval
    using add-xzero.prems(4) yval by blast
  have ywff: wff-value yval
    using yeval add-xzero.prems(1) eval-in-ids wff-values.simps by blast
  then have y: IntVal b' res' = yval
    by (meson RefNodeE add-xzero.prems(3) evalDet yeval)
  then have bpBits: b' = 32
    using ywff wff-int32 by auto
  then have res-val: IntVal b res = intval-add (IntVal 32 0) yval
    using eval.AddNode eval.ConstantNode add-xzero(1,3,5)
    using evalDet by (metis IRNode.inject(2) add-xzero.prems(4) res xval)
  then have bBits: b = 32
    using ywff intval-add-bits bpBits y by force
  then show ?case
    using eval.RefNode yval res-val ywff add32-0 y
    by (metis Value.inject(1) add-zero-32 bpBits)
next
  case (add-yzero x y c-2)
  have yeval: g m ⊢ kind g y ↦ (IntVal 32 0)
    by (simp add: ConstantNode add-yzero.hyps(2) add-yzero.hyps(3))
  have xeval: g m ⊢ kind g x ↦ xval
    using add-yzero.prems(4) xval by fastforce
  then have xwff: wff-value xval
    using yeval add-yzero.prems(1) eval-in-ids wff-values.simps by blast
  then have y: IntVal b' res' = xval
    by (meson RefNodeE add-yzero.prems(3) evalDet xeval)
  then have bpBits: b' = 32
    using xwff wff-int32 by auto
```

then have *IntVal b res* = *intval-add xval* (*IntVal 32 0*)
using *eval.AddNode eval.ConstantNode add-yzero*(*2*,*3*,*5*)
using *evalDet xeval* **by** *presburger*
then have *res*: *IntVal b res* = *intval-add* (*IntVal 32 0*) *xval*
by (*simp add*: *intval-add-sym*)
then have *b* = *32*
using *xwff intval-add-bits bpBits y* **by** *force*
then show *?case* **using** *eval.RefNode xval wff-int32 intval-add-bits*
by (*metis Value.inject*(*1*) *res add-zero-32 xwff y*)
next
case (*add-xsub x a y*)
then show *?case* **sorry**
next
case (*add-ysub y a x*)
then show *?case* **sorry**
next
case (*add-xnegate nx x y*)
then show *?case* **sorry**
next
case (*add-ynegate ny y x*)
then show *?case* **sorry**
qed
qed

lemma *CanonicalizeSubProof*:
assumes *CanonicalizeSub g before after*
assumes *wff-stamps g*
assumes *g m* ⊢ *before* ↦ *IntVal b1 res*
assumes *g m* ⊢ *after* ↦ *IntVal b2 res′*
shows *res* = *res′*
using *assms* **proof** (*induct rule*: *CanonicalizeSub.induct*)
case (*sub-same x y b l h*)
then show *?case* **sorry**
next
case (*sub-both-const x c-1 y c-2 val*)
then show *?case* **sorry**
next
case (*sub-left-add1 left a b*)
then show *?case* **sorry**
next
case (*sub-left-add2 left a b*)
then show *?case* **sorry**
next
case (*sub-left-sub left a b*)
then show *?case* **sorry**
next
case (*sub-right-add1 right a b*)
then show *?case* **sorry**

next
  case (*sub-right-add2 right a b*)
  then show *?case* **sorry**
next
  case (*sub-right-sub right a b*)
  then show *?case* **sorry**
next
  case (*sub-yzero y uu x*)
  then show *?case* **sorry**
next
  case (*sub-xzero x uv y*)
  then show *?case* **sorry**
next
  case (*sub-y-negate nb b a*)
  then show *?case* **sorry**
qed

lemma *CanonicalizeIfProof*:
  fixes *m*::*MapState* **and** *h*::*FieldRefHeap*
  assumes *kind g nid* = *before*
  assumes *CanonicalizeIf g before after*
  assumes g' = *replace-node nid* (*after*, *s*) *g*
  assumes $g \vdash (nid, m, h) \rightarrow (nid', m, h)$
  shows $nid \mid g \sim g'$
  using *assms*(*2*) *assms*
proof (*induct rule*: *CanonicalizeIf.induct*)
  case (*trueConst cond condv tb fb*)
  have *gstep*: $g \vdash (nid, m, h) \rightarrow (tb, m, h)$
   using *ConstantNode IfNode trueConst.hyps*(*1*) *trueConst.hyps*(*2*) *trueConst.prems*(*1*)
    using *step.IfNode* **by** *presburger*
  have g'*step*: $g' \vdash (nid, m, h) \rightarrow (tb, m, h)$
    using *replace-node-lookup*
    by (*simp add*: *stepRefNode trueConst.prems*(*3*))
  from *gstep* g'*step* **show** *?case*
    using *lockstep-strong-bisimilulation assms*(*3*) **by** *simp*
next
  case (*falseConst cond condv tb fb*)
  have *gstep*: $g \vdash (nid, m, h) \rightarrow (fb, m, h)$
      using *ConstantNode IfNode falseConst.hyps*(*1*) *falseConst.hyps*(*2*) *falseC-onst.prems*(*1*)
    using *step.IfNode* **by** *presburger*
  have g'*step*: $g' \vdash (nid, m, h) \rightarrow (fb, m, h)$
    using *replace-node-lookup*
    by (*simp add*: *falseConst.prems*(*3*) *stepRefNode*)
  from *gstep* g'*step* **show** *?case*
    using *lockstep-strong-bisimilulation assms*(*3*) **by** *simp*
next
  case (*eqBranch cond tb fb*)

```
  have cval: ∃ v. (g m ⊢ kind g cond ↦ v)
    using IfNodeCond
    by (meson eqBranch.prems(1) eqBranch.prems(4))
  then have gstep: g ⊢ (nid, m, h) → (tb, m, h)
    using eqBranch(2,3) assms(4) IfNodeStepCases by blast
  have g'step: g' ⊢ (nid, m, h) → (tb, m, h)
    by (simp add: eqBranch.prems(3) stepRefNode)
  from gstep g'step show ?thesis
    using lockstep-strong-bisimilulation assms(3) by simp
next
  case (eqCondition cond x tb fb)
  have cval: ∃ v. (g m ⊢ kind g cond ↦ v)
    using IfNodeCond
    by (meson eqCondition.prems(1) eqCondition.prems(4))
  have gstep: g ⊢ (nid, m, h) → (tb, m, h)
    using step.IfNode eval.IntegerEqualsNode
     by (smt (z3) IntegerEqualsNodeE bool-to-val.simps(1) cval eqCondition.hyps
eqCondition.prems(1) val-to-bool.simps(1))
  have g'step: g' ⊢ (nid, m, h) → (tb, m, h)
    using replace-node-lookup
    using IRNode.simps(2114) eqCondition.prems(3) stepRefNode by presburger
  from gstep g'step show ?thesis
    using lockstep-strong-bisimilulation assms(3) by simp
qed

end
```

2 Conditional Elimination Phase

```
theory ConditionalElimination
  imports
    Proofs.IRGraphFrames
    Proofs.Stuttering
    Proofs.Form
    Proofs.Rewrites
    Proofs.Bisimulation
begin
```

2.1 Individual Elimination Rules

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

```
datatype TriState = Unknown | KnownTrue | KnownFalse
```

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

inductive *implies* :: *IRGraph* $\Rightarrow$ *IRNode* $\Rightarrow$ *IRNode* $\Rightarrow$ *TriState* $\Rightarrow$ *bool*
  (- $\vdash$ - & - $\hookrightarrow$ -) **for** g **where**
  eq-imp-less:
  $g \vdash$ (*IntegerEqualsNode* x y) & (*IntegerLessThanNode* x y) $\hookrightarrow$ *KnownFalse* |
  eq-imp-less-rev:
  $g \vdash$ (*IntegerEqualsNode* x y) & (*IntegerLessThanNode* y x) $\hookrightarrow$ *KnownFalse* |
  less-imp-rev-less:
  $g \vdash$ (*IntegerLessThanNode* x y) & (*IntegerLessThanNode* y x) $\hookrightarrow$ *KnownFalse* |
  less-imp-not-eq:
  $g \vdash$ (*IntegerLessThanNode* x y) & (*IntegerEqualsNode* x y) $\hookrightarrow$ *KnownFalse* |
  less-imp-not-eq-rev:
  $g \vdash$ (*IntegerLessThanNode* x y) & (*IntegerEqualsNode* y x) $\hookrightarrow$ *KnownFalse* |

  x-imp-x:
  $g \vdash x$ & $x \hookrightarrow$ *KnownTrue* |

  negate-false:
  $[\![g \vdash x$ & (*kind* g y) $\hookrightarrow$ *KnownTrue*$]\!] \Longrightarrow g \vdash x$ & (*LogicNegationNode* y) $\hookrightarrow$ *KnownFalse* |
  negate-true:
  $[\![g \vdash x$ & (*kind* g y) $\hookrightarrow$ *KnownFalse*$]\!] \Longrightarrow g \vdash x$ & (*LogicNegationNode* y) $\hookrightarrow$ *KnownTrue*

Total relation over partial implies relation

inductive *condition-implies* :: *IRGraph* $\Rightarrow$ *IRNode* $\Rightarrow$ *IRNode* $\Rightarrow$ *TriState* $\Rightarrow$ *bool*
  (- $\vdash$ - & - $\rightharpoonup$ -) **for** g **where**
  $[\![\neg(g \vdash a$ & $b \hookrightarrow imp)]\!] \Longrightarrow (g \vdash a$ & $b \rightharpoonup$ *Unknown*) |
  $[\![(g \vdash a$ & $b \hookrightarrow imp)]\!] \Longrightarrow (g \vdash a$ & $b \rightharpoonup imp)$

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

lemma *logic-negation-relation*:
  assumes *wff-values* g
  assumes g $m \vdash$ *kind* g $y \mapsto val$
  assumes *kind* g *neg* = *LogicNegationNode* y
  assumes g $m \vdash$ *kind* g *neg* $\mapsto$ *invval*
  shows *val-to-bool val* $\longleftrightarrow$ $\neg$(*val-to-bool inval*)
proof –
  have *wff-value val*
    using *assms*(*1*) *assms*(*2*) *eval-in-ids wff-values.elims*(*2*)
    by *meson*
  have *wff-value invval*
    using *assms*(*1*,*4*) *eval-in-ids wff-values.simps* **by** *blast*
  then show *?thesis*
    using *assms eval.LogicNegationNode*

```
    by fastforce
qed

lemma implies-valid:
  assumes wff-graph g ∧ wff-values g
  assumes g ⊢ x & y ⇀ imp
  assumes g m ⊢ x ↦ v1
  assumes g m ⊢ y ↦ v2
  shows (imp = KnownTrue ⟶ (val-to-bool v1 ⟶ val-to-bool v2)) ∧
         (imp = KnownFalse ⟶ (val-to-bool v1 ⟶ ¬(val-to-bool v2)))
    (is (?TP ⟶ ?TC) ∧ (?FP ⟶ ?FC))
  apply (intro conjI; rule impI)
proof −
  assume KnownTrue: ?TP
  show ?TC proof −
  have s: g ⊢ x & y ↪ imp
    using KnownTrue assms(2) condition-implies.cases by blast
  then show ?thesis
  using KnownTrue assms proof (induct x y imp rule: implies.induct)
    case (eq-imp-less x y)
    then show ?case by simp
  next
    case (eq-imp-less-rev x y)
    then show ?case by simp
  next
    case (less-imp-rev-less x y)
    then show ?case by simp
  next
    case (less-imp-not-eq x y)
    then show ?case by simp
  next
    case (less-imp-not-eq-rev x y)
    then show ?case by simp
  next
    case (x-imp-x x1)
    then show ?case using evalDet
      using assms(2,3) by blast
  next
    case (negate-false x1)
    then show ?case using evalDet
      using assms(2,3) by blast
  next
    case (negate-true x y)
    then show ?case using logic-negation-relation
      by fastforce
  qed
  qed
next
  assume KnownFalse: ?FP
```

show *?FC* **proof** −
have $g \vdash x \ \& \ y \hookrightarrow imp$
using *KnownFalse assms*(*2*) *condition-implies.cases* **by** *blast*
then show *?thesis*
using *assms KnownFalse* **proof** (*induct x y imp rule*: *implies.induct*)
case (*eq-imp-less x y*)
obtain *b xval* **where** *xval*: $g\ m \vdash (kind\ g\ x) \mapsto IntVal\ b\ xval$
using *eq-imp-less.prems*(*3*) **by** *blast*
then obtain *yval* **where** *yval*: $g\ m \vdash (kind\ g\ y) \mapsto IntVal\ b\ yval$
using *eq-imp-less.prems*(*3*)
using *evalDet* **by** *blast*
have *eqeval*: $g\ m \vdash (IntegerEqualsNode\ x\ y) \mapsto bool\text{-}to\text{-}val(xval = yval)$
using *eval.IntegerEqualsNode*
using *xval yval* **by** *blast*
have *lesseval*: $g\ m \vdash (IntegerLessThanNode\ x\ y) \mapsto bool\text{-}to\text{-}val(xval < yval)$
using *eval.IntegerLessThanNode*
using *xval yval* **by** *blast*
have $xval = yval \longrightarrow \neg(xval < yval)$
by *blast*
then show *?case*
using *eqeval lesseval*
by (*metis* (*full-types*) *eq-imp-less.prems*(*3*) *eq-imp-less.prems*(*4*) *bool-to-val.simps*(*2*) *evalDet val-to-bool.simps*(*1*))
next
case (*eq-imp-less-rev x y*)
obtain *b xval* **where** *xval*: $g\ m \vdash (kind\ g\ x) \mapsto IntVal\ b\ xval$
using *eq-imp-less-rev.prems*(*3*) **by** *blast*
then obtain *yval* **where** *yval*: $g\ m \vdash (kind\ g\ y) \mapsto IntVal\ b\ yval$
using *eq-imp-less-rev.prems*(*3*)
using *evalDet* **by** *blast*
have *eqeval*: $g\ m \vdash (IntegerEqualsNode\ x\ y) \mapsto bool\text{-}to\text{-}val(xval = yval)$
using *eval.IntegerEqualsNode*
using *xval yval* **by** *blast*
have *lesseval*: $g\ m \vdash (IntegerLessThanNode\ y\ x) \mapsto bool\text{-}to\text{-}val(yval < xval)$
using *eval.IntegerLessThanNode*
using *xval yval* **by** *blast*
have $xval = yval \longrightarrow \neg(yval < xval)$
by *blast*
then show *?case*
using *eqeval lesseval*
by (*metis* (*full-types*) *eq-imp-less-rev.prems*(*3*) *eq-imp-less-rev.prems*(*4*) *bool-to-val.simps*(*2*) *evalDet val-to-bool.simps*(*1*))
next
case (*less-imp-rev-less x y*)
obtain *b xval* **where** *xval*: $g\ m \vdash (kind\ g\ x) \mapsto IntVal\ b\ xval$
using *less-imp-rev-less.prems*(*3*) **by** *blast*
then obtain *yval* **where** *yval*: $g\ m \vdash (kind\ g\ y) \mapsto IntVal\ b\ yval$
using *less-imp-rev-less.prems*(*3*)
using *evalDet* **by** *blast*

have *lesseval*: *g m* ⊢ (*IntegerLessThanNode x y*) ↦ *bool-to-val*(*xval* < *yval*)
  using *eval.IntegerLessThanNode*
  using *xval yval* **by** *blast*
have *revlesseval*: *g m* ⊢ (*IntegerLessThanNode y x*) ↦ *bool-to-val*(*yval* < *xval*)
  using *eval.IntegerLessThanNode*
  using *xval yval* **by** *blast*
have *xval* < *yval* ⟶ ¬(*yval* < *xval*)
  by *simp*
then show *?case*
by (*metis* (*full-types*) *bool-to-val.simps*(*2*) *evalDet less-imp-rev-less.prems*(*3*,*4*) *less-imp-rev-less.prems*(*3*) *lesseval revlesseval val-to-bool.simps*(*1*))
next
case (*less-imp-not-eq x y*)
obtain *b xval* **where** *xval*: *g m* ⊢ (*kind g x*) ↦ *IntVal b xval*
  using *less-imp-not-eq.prems*(*3*) **by** *blast*
then obtain *yval* **where** *yval*: *g m* ⊢ (*kind g y*) ↦ *IntVal b yval*
  using *less-imp-not-eq.prems*(*3*)
  using *evalDet* **by** *blast*
have *eqeval*: *g m* ⊢ (*IntegerEqualsNode x y*) ↦ *bool-to-val*(*xval* = *yval*)
  using *eval.IntegerEqualsNode*
  using *xval yval* **by** *blast*
have *lesseval*: *g m* ⊢ (*IntegerLessThanNode x y*) ↦ *bool-to-val*(*xval* < *yval*)
  using *eval.IntegerLessThanNode*
  using *xval yval* **by** *blast*
have *xval* < *yval* ⟶ ¬(*xval* = *yval*)
  by *simp*
then show *?case*
by (*metis* (*full-types*) *bool-to-val.simps*(*2*) *eqeval evalDet less-imp-not-eq.prems*(*3*,*4*) *less-imp-not-eq.prems*(*3*) *lesseval val-to-bool.simps*(*1*))
next
case (*less-imp-not-eq-rev x y*)
obtain *b xval* **where** *xval*: *g m* ⊢ (*kind g x*) ↦ *IntVal b xval*
  using *less-imp-not-eq-rev.prems*(*3*) **by** *blast*
then obtain *yval* **where** *yval*: *g m* ⊢ (*kind g y*) ↦ *IntVal b yval*
  using *less-imp-not-eq-rev.prems*(*3*)
  using *evalDet* **by** *blast*
have *eqeval*: *g m* ⊢ (*IntegerEqualsNode y x*) ↦ *bool-to-val*(*yval* = *xval*)
  using *eval.IntegerEqualsNode*
  using *xval yval* **by** *blast*
have *lesseval*: *g m* ⊢ (*IntegerLessThanNode x y*) ↦ *bool-to-val*(*xval* < *yval*)
  using *eval.IntegerLessThanNode*
  using *xval yval* **by** *blast*
have *xval* < *yval* ⟶ ¬(*yval* = *xval*)
  by *simp*
then show *?case*
by (*metis* (*full-types*) *bool-to-val.simps*(*2*) *eqeval evalDet less-imp-not-eq-rev.prems*(*3*,*4*) *less-imp-not-eq-rev.prems*(*3*) *lesseval val-to-bool.simps*(*1*))
next
case (*x-imp-x x1*)

    then show *?case* **by** *simp*
  next
    case (*negate-false x y*)
    then show *?case* **using** *logic-negation-relation* **sorry**
  next
    case (*negate-true x1*)
    then show *?case* **by** *simp*
  qed
  qed
qed

lemma *implies-true-valid*:
  assumes $wff\text{-}graph\ g \wedge wff\text{-}values\ g$
  assumes $g \vdash x\ \&\ y \rightharpoonup imp$
  assumes $imp = KnownTrue$
  assumes $g\ m \vdash x \mapsto v1$
  assumes $g\ m \vdash y \mapsto v2$
  shows $val\text{-}to\text{-}bool\ v1 \longrightarrow val\text{-}to\text{-}bool\ v2$
  using *assms implies-valid* **by** *blast*

lemma *implies-false-valid*:
  assumes $wff\text{-}graph\ g \wedge wff\text{-}values\ g$
  assumes $g \vdash x\ \&\ y \rightharpoonup imp$
  assumes $imp = KnownFalse$
  assumes $g\ m \vdash x \mapsto v1$
  assumes $g\ m \vdash y \mapsto v2$
  shows $val\text{-}to\text{-}bool\ v1 \longrightarrow \neg(val\text{-}to\text{-}bool\ v2)$
  using *assms implies-valid* **by** *blast*

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

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

inductive $tryFold :: IRNode \Rightarrow (ID \Rightarrow Stamp) \Rightarrow TriState \Rightarrow bool$
  where
  $[\![alwaysDistinct\ (stamps\ x)\ (stamps\ y)]\!]$
    $\Longrightarrow tryFold\ (IntegerEqualsNode\ x\ y)\ stamps\ KnownFalse\ |$
  $[\![neverDistinct\ (stamps\ x)\ (stamps\ y)]\!]$
    $\Longrightarrow tryFold\ (IntegerEqualsNode\ x\ y)\ stamps\ KnownTrue\ |$
  $[\![is\text{-}IntegerStamp\ (stamps\ x);$
    $is\text{-}IntegerStamp\ (stamps\ y);$
    $stpi\text{-}upper\ (stamps\ x) < stpi\text{-}lower\ (stamps\ y)]\!]$
    $\Longrightarrow tryFold\ (IntegerLessThanNode\ x\ y)\ stamps\ KnownTrue\ |$
  $[\![is\text{-}IntegerStamp\ (stamps\ x);$
    $is\text{-}IntegerStamp\ (stamps\ y);$
    $stpi\text{-}lower\ (stamps\ x) \geq stpi\text{-}upper\ (stamps\ y)]\!]$
    $\Longrightarrow tryFold\ (IntegerLessThanNode\ x\ y)\ stamps\ KnownFalse$

Proofs that show that when the stamp lookup function is well-formed, the tryFold relation correctly predicts the output value with respect to our evaluation semantics.

lemma *tryFoldIntegerEqualsAlwaysDistinct*:
 assumes *wff-stamp g stamps*
 assumes *kind g nid* = (*IntegerEqualsNode x y*)
 assumes *g m* ⊢ (*kind g nid*) ↦ *v*
 assumes *alwaysDistinct* (*stamps x*) (*stamps y*)
 shows *v* = *IntVal 1 0*
 using *assms eval.IntegerEqualsNode join-unequal alwaysDistinct.simps*
 by (*smt* (*verit*, *best*) *IntegerEqualsNodeE bool-to-val.simps*(*2*) *eval-in-ids wff-stamp.elims*(*2*))

lemma *tryFoldIntegerEqualsNeverDistinct*:
 assumes *wff-stamp g stamps*
 assumes *kind g nid* = (*IntegerEqualsNode x y*)
 assumes *g m* ⊢ (*kind g nid*) ↦ *v*
 assumes *neverDistinct* (*stamps x*) (*stamps y*)
 shows *v* = *IntVal 1 1*
 using *assms neverDistinctEqual IntegerEqualsNodeE*
 by (*smt* (*verit*, *ccfv-threshold*) *Value.inject*(*1*) *bool-to-val.simps*(*1*) *eval-in-ids wff-stamp.simps*)

lemma *tryFoldIntegerLessThanTrue*:
 assumes *wff-stamp g stamps*
 assumes *kind g nid* = (*IntegerLessThanNode x y*)
 assumes *g m* ⊢ (*kind g nid*) ↦ *v*
 assumes *stpi-upper* (*stamps x*) < *stpi-lower* (*stamps y*)
 shows *v* = *IntVal 1 1*
proof −
 have *stamp-type*: *is-IntegerStamp* (*stamps x*)
 using *assms*
 by (*metis IntegerLessThanNodeE Stamp.disc*(*2*) *Value.distinct*(*1*) *eval-in-ids valid-value.elims*(*2*) *wff-stamp.elims*(*2*))
 obtain *xval b* **where** *xval*: *g m* ⊢ *kind g x* ↦ *IntVal b xval*
 using *assms*(*2*,*3*) *eval.IntegerLessThanNode* **by** *auto*
 obtain *yval b* **where** *yval*: *g m* ⊢ *kind g y* ↦ *IntVal b yval*
 using *assms*(*2*,*3*) *eval.IntegerLessThanNode* **by** *auto*
 have *is-IntegerStamp* (*stamps x*) ∧ *is-IntegerStamp* (*stamps y*)
 using *assms*(*4*)
 by (*metis stamp-type Stamp.disc*(*2*) *Value.distinct*(*1*) *assms*(*1*) *eval-in-ids valid-value.elims*(*2*) *wff-stamp.simps yval*)
 then have *xval* < *yval*
 using *boundsNoOverlap xval yval assms*(*1*,*4*)
 using *eval-in-ids wff-stamp.elims*(*2*)
 by *metis*
 then show *?thesis*
 by (*metis* (*full-types*) *IntegerLessThanNodeE Value.sel*(*3*) *assms*(*2*) *assms*(*3*) *bool-to-val.simps*(*1*) *evalDet xval yval*)
qed

lemma *tryFoldIntegerLessThanFalse*:
  assumes *wff-stamp g stamps*
  assumes *kind g nid* = (*IntegerLessThanNode x y*)
  assumes *g m* ⊢ (*kind g nid*) ↦ *v*
  assumes *stpi-lower* (*stamps x*) ≥ *stpi-upper* (*stamps y*)
  shows *v* = *IntVal 1 0*
  proof −
  have *stamp-type*: *is-IntegerStamp* (*stamps x*)
    using *assms*
     by (*metis IntegerLessThanNodeE Stamp.disc*(*2*) *Value.distinct*(*1*) *eval-in-ids valid-value.elims*(*2*) *wff-stamp.elims*(*2*))
  obtain *xval b* **where** *xval*: *g m* ⊢ *kind g x* ↦ *IntVal b xval*
    using *assms*(*2*,*3*) *eval.IntegerLessThanNode* **by** *auto*
  obtain *yval b* **where** *yval*: *g m* ⊢ *kind g y* ↦ *IntVal b yval*
    using *assms*(*2*,*3*) *eval.IntegerLessThanNode* **by** *auto*
  have *is-IntegerStamp* (*stamps x*) ∧ *is-IntegerStamp* (*stamps y*)
    using *assms*(*4*)
      by (*metis stamp-type Stamp.disc*(*2*) *Value.distinct*(*1*) *assms*(*1*) *eval-in-ids valid-value.elims*(*2*) *wff-stamp.simps yval*)
  then have ¬(*xval* < *yval*)
    using *boundsAlwaysOverlap xval yval assms*(*1*,*4*)
    using *eval-in-ids wff-stamp.elims*(*2*)
    by *metis*
  then show *?thesis*
    by (*smt* (*verit*, *best*) *IntegerLessThanNodeE Value.inject*(*1*) *assms*(*2*) *assms*(*3*) *bool-to-val.simps*(*2*) *evalDet xval yval*)
qed

theorem *tryFoldProofTrue*:
  assumes *wff-stamp g stamps*
  assumes *tryFold* (*kind g nid*) *stamps tristate*
  assumes *tristate* = *KnownTrue*
  assumes *g m* ⊢ *kind g nid* ↦ *v*
  shows *val-to-bool v*
  using *assms*(*2*) **proof** (*induction kind g nid stamps tristate rule*: *tryFold.induct*)
case (*1 stamps x y*)
  then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms*
     by (*smt* (*verit*, *best*) *IRNode.distinct*(*949*) *TriState.distinct*(*5*) *tryFold.cases tryFoldIntegerEqualsNeverDistinct val-to-bool.simps*(*1*))
next
  case (*2 stamps x y*)
  then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms*
    by (*smt* (*verit*) *IRNode.distinct*(*949*) *TriState.distinct*(*5*) *tryFold.cases tryFold-IntegerEqualsNeverDistinct val-to-bool.simps*(*1*))
next
  case (*3 stamps x y*)
  then show *?case* **using** *tryFoldIntegerLessThanTrue assms*
   by (*smt* (*verit*, *best*) *IRNode.simps*(*994*) *TriState.simps*(*6*) *tryFold.cases val-to-bool.simps*(*1*))

next
case (*4 stamps x y*)
  then show *?case* **using** *tryFoldIntegerLessThanFalse assms*
    by (*smt* (*verit*, *best*) *IRNode.simps*(*994*) *TriState.simps*(*6*) *tryFold.simps tryFoldIntegerLessThanTrue val-to-bool.simps*(*1*))
qed

theorem *tryFoldProofFalse*:
  assumes *wff-stamp g stamps*
  assumes *tryFold* (*kind g nid*) *stamps tristate*
  assumes *tristate* = *KnownFalse*
  assumes $g\ m \vdash (kind\ g\ nid) \mapsto v$
  shows ¬(*val-to-bool v*)
using *assms*(*2*) **proof** (*induction kind g nid stamps tristate rule*: *tryFold.induct*)
case (*1 stamps x y*)
  then show *?case* **using** *tryFoldIntegerEqualsAlwaysDistinct assms*
    by (*smt* (*verit*, *best*) *IRNode.distinct*(*949*) *TriState.distinct*(*5*) *Value.inject*(*1*) *tryFold.cases val-to-bool.elims*(*2*))
next
case (*2 stamps x y*)
  then show *?case* **using** *tryFoldIntegerEqualsNeverDistinct assms*
    by (*smt* (*verit*, *best*) *IRNode.distinct*(*949*) *TriState.distinct*(*5*) *Value.inject*(*1*) *tryFold.cases tryFoldIntegerEqualsAlwaysDistinct val-to-bool.elims*(*2*))
next
  case (*3 stamps x y*)
  then show *?case* **using** *tryFoldIntegerLessThanTrue assms*
    by (*smt* (*verit*, *best*) *TriState.distinct*(*5*) *tryFold.cases tryFoldIntegerEqualsAlwaysDistinct tryFoldIntegerLessThanFalse val-to-bool.simps*(*1*))
next
  case (*4 stamps x y*)
  then show *?case* **using** *tryFoldIntegerLessThanFalse assms*
    by (*smt* (*verit*, *best*) *TriState.distinct*(*5*) *tryFold.cases tryFoldIntegerEqualsAlwaysDistinct val-to-bool.simps*(*1*))
qed

inductive-cases *StepE*:
  $g \vdash (nid,m,h) \rightarrow (nid',m',h)$

Perform conditional elimination rewrites on the graph for a particular node.

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

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

inductive *ConditionalEliminationStep* ::
IRNode set ⇒ (*ID* ⇒ *Stamp*) ⇒ *IRGraph* ⇒ *ID* ⇒ *IRGraph* ⇒ *bool* **where**
impliesTrue:
$[\![$*kind g ifcond* = (*IfNode cid t f*);
cond = *kind g cid*;
∃ *c* ∈ *conds* . (*g* ⊢ *c* & *cond* ↪ *KnownTrue*);
g′ = *constantCondition True ifcond* (*kind g ifcond*) *g*
$]\!]$ ⟹ *ConditionalEliminationStep conds stamps g ifcond g′* |

impliesFalse:
$[\![$*kind g ifcond* = (*IfNode cid t f*);
cond = *kind g cid*;
∃ *c* ∈ *conds* . (*g* ⊢ *c* & *cond* ↪ *KnownFalse*);
g′ = *constantCondition False ifcond* (*kind g ifcond*) *g*
$]\!]$ ⟹ *ConditionalEliminationStep conds stamps g ifcond g′* |

tryFoldTrue:
$[\![$*kind g ifcond* = (*IfNode cid t f*);
cond = *kind g cid*;
tryFold (*kind g cid*) *stamps KnownTrue*;
g′ = *constantCondition True ifcond* (*kind g ifcond*) *g*
$]\!]$ ⟹ *ConditionalEliminationStep conds stamps g ifcond g′* |

tryFoldFalse:
$[\![$*kind g ifcond* = (*IfNode cid t f*);
cond = *kind g cid*;
tryFold (*kind g cid*) *stamps KnownFalse*;
g′ = *constantCondition False ifcond* (*kind g ifcond*) *g*
$]\!]$ ⟹ *ConditionalEliminationStep conds stamps g ifcond g′*

code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*) *ConditionalEliminationStep* **.**

thm *ConditionalEliminationStep.equation*

2.2 Control-flow Graph Traversal

type-synonym *Seen* = *ID set*
type-synonym *Conditions* = *IRNode list*
type-synonym *StampFlow* = (*ID* ⇒ *Stamp*) *list*

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

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

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

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

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

```
fun pred :: IRGraph ⇒ ID ⇒ ID option where
  pred g nid = (case kind g nid of
    (MergeNode ends - -) ⇒ Some (hd ends) |
    - ⇒
      (if IRGraph.predecessors g nid = {}
        then None else
        Some (hd (sorted-list-of-set (IRGraph.predecessors g nid)))
      )
  )
```

When the basic block of an if statement is entered, we know that the condition of the preceding if statement must be true. As in the GraalVM compiler, we introduce the registerNewCondition funciton which roughly corresponds to the ConditionalEliminationPhase.registerNewCondition. This method updates the flow-sensitive stamp information based on the condition which we know must be true.

```
fun clip-upper :: Stamp ⇒ int ⇒ Stamp where
  clip-upper (IntegerStamp b l h) c = (IntegerStamp b l c) |
  clip-upper s c = s
fun clip-lower :: Stamp ⇒ int ⇒ Stamp where
  clip-lower (IntegerStamp b l h) c = (IntegerStamp b c h) |
  clip-lower s c = s

fun registerNewCondition :: IRGraph ⇒ IRNode ⇒ (ID ⇒ Stamp) ⇒ (ID ⇒
Stamp) where

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

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

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

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

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

inductive *Step*
$:: IRGraph \Rightarrow (ID \times Seen \times Conditions \times StampFlow) \Rightarrow (ID \times Seen \times Conditions \times StampFlow)\ option \Rightarrow bool$
for g **where**
— Hit a BeginNode with an IfNode predecessor which represents the start of a basic block for the IfNode. 1. nid' will be the successor of the begin node. 2. Find the first and only predecessor. 3. Extract condition from the preceding IfNode. 4. Negate condition if the begin node is second branch (we've taken the else branch of the condition) 5. Add the condition or the negated condition to stack 6. Perform any stamp updates based on the condition using the registerNewCondition function and place them on the top of the stack of stamp information

$[\![kind\ g\ nid = BeginNode\ nid';$

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

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

$i = find\text{-}index\ nid\ (successors\text{-}of\ (kind\ g\ ifcond));$
$c = (if\ i = 0\ then\ kind\ g\ cond\ else\ NegateNode\ cond);$
$conds' = c\ \#\ conds;$

$flow' = registerNewCondition\ g\ (kind\ g\ cond)\ (hdOr\ flow\ (stamp\ g))]\!]$
$\Longrightarrow Step\ g\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds', flow'\ \#\ flow))\ |$

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

$[\![kind\ g\ nid = EndNode;$

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

$nid' = any\text{-}usage\ g\ nid;$

$conds' = tl\ conds;$
$flow' = tl\ flow]\!]$
$\Longrightarrow Step\ g\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds', flow'))\ |$

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

$[\![\neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\neg(is\text{-}BeginNode\ (kind\ g\ nid));$

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

$Some\ nid' = nextEdge\ seen'\ nid\ g\rrbracket$
$\Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds,\ flow))\ |$

— We can cannot find a successor edge that is not in seen, give back None
$\llbracket\neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\neg(is\text{-}BeginNode\ (kind\ g\ nid));$

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

$None = nextEdge\ seen'\ nid\ g\rrbracket$
$\Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ None\ |$

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

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

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

inductive *ConditionalEliminationPhase*
$::\ IRGraph \Rightarrow (ID \times Seen \times Conditions \times StampFlow) \Rightarrow IRGraph \Rightarrow bool$
where

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

$ConditionalEliminationPhase\ g'\ (nid',\ seen',\ conds',\ flow')\ g''\rrbracket$
$\Longrightarrow ConditionalEliminationPhase\ g\ (nid,\ seen,\ conds,\ flow)\ g''\ |$

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

$ConditionalEliminationPhase\ g\ (nid',\ seen',\ conds',\ flow')\ g'\rrbracket$
$\Longrightarrow ConditionalEliminationPhase\ g\ (nid,\ seen,\ conds,\ flow)\ g'\ |$

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

— Can't do a step and have no predecessors so terminate

$[\![$*Step g* (*nid*, *seen*, *conds*, *flow*) *None*;
None = *pred g nid*$]\!]$
$\Longrightarrow$ *ConditionalEliminationPhase g* (*nid*, *seen*, *conds*, *flow*) *g*

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

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

lemma *IfNodeStepE*: $g \vdash (nid, m, h) \rightarrow (nid', m', h) \Longrightarrow$
($\bigwedge$*cond tb fb val*.
kind g nid = *IfNode cond tb fb* $\Longrightarrow$
nid' = (*if val-to-bool val then tb else fb*) $\Longrightarrow$
$g\ m \vdash$ *kind g cond* $\mapsto$ *val* $\Longrightarrow m' = m$)
using *StepE*
by (*smt* (*verit*, *best*) *IfNode Pair-inject stepDet*)

lemma *ifNodeHasCondEvalStutter*:
assumes ($g\ m\ h \vdash nid \rightsquigarrow nid'$)
assumes *kind g nid* = *IfNode cond t f*
shows $\exists$ *v*. ($g\ m \vdash$ *kind g cond* $\mapsto v$)
using *IfNodeStepE assms*(*1*) *assms*(*2*) *stutter.cases*
by (*meson IfNodeCond*)

lemma *ifNodeHasCondEval*:
assumes ($g \vdash (nid, m, h) \rightarrow (nid', m', h')$)
assumes *kind g nid* = *IfNode cond t f*
shows $\exists$ *v*. ($g\ m \vdash$ *kind g cond* $\mapsto v$)
using *IfNodeStepE assms*(*1*) *assms*(*2*)
by (*smt* (*z3*) *IRNode.disc*(*932*) *IRNode.simps*(*938*) *IRNode.simps*(*958*) *IRNode.simps*(*972*) *IRNode.simps*(*974*) *IRNode.simps*(*978*) *Pair-inject StutterStep ifNodeHasCondEvalStutter is-AbstractEndNode.simps is-EndNode.simps*(*12*) *step.cases*)

lemma *replace-if-t*:
assumes *kind g nid* = *IfNode cond tb fb*
assumes $g\ m \vdash$ *kind g cond* $\mapsto$ *bool*
assumes *val-to-bool bool*
assumes g': g' = *replace-usages nid tb g*
shows $\exists nid'$.($g\ m\ h \vdash nid \rightsquigarrow nid'$) $\longleftrightarrow$ ($g'\ m\ h \vdash nid \rightsquigarrow nid'$)
proof −
have *g1step*: $g \vdash (nid, m, h) \rightarrow (tb, m, h)$
by (*meson IfNode assms*(*1*) *assms*(*2*) *assms*(*3*))
have *g2step*: $g' \vdash (nid, m, h) \rightarrow (tb, m, h)$
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* $\vdash$ *kind g cond* $\mapsto$ *bool*
assumes *val-to-bool bool*
assumes *g'*: *g'* = *replace-usages nid tb g*
shows $\exists$ *nid'* .(*g m h* $\vdash$ *nid* $\rightsquigarrow$ *nid'*) $\longrightarrow$ (*g' m h* $\vdash$ *nid* $\rightsquigarrow$ *nid'*)
using *replace-if-t assms* **by** *blast*

lemma *replace-if-f*:
assumes *kind g nid* = *IfNode cond tb fb*
assumes *g m* $\vdash$ *kind g cond* $\mapsto$ *bool*
assumes $\neg$(*val-to-bool bool*)
assumes *g'*: *g'* = *replace-usages nid fb g*
shows $\exists$ *nid'* .(*g m h* $\vdash$ *nid* $\rightsquigarrow$ *nid'*) $\longleftrightarrow$ (*g' m h* $\vdash$ *nid* $\rightsquigarrow$ *nid'*)
proof –
have *g1step*: *g* $\vdash$ (*nid*, *m*, *h*) $\rightarrow$ (*fb*, *m*, *h*)
by (*meson IfNode assms*(*1*) *assms*(*2*) *assms*(*3*))
have *g2step*: *g'* $\vdash$ (*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*: *wff-graph g*
assumes *ws*: *wff-stamps g*
assumes *wv*: *wff-values g*
assumes *nid*: *nid* $\in$ *ids g*
assumes *conds-valid*: $\forall$ *c* $\in$ *conds* . $\exists$ *v*. (*g m* $\vdash$ *c* $\mapsto$ *v*) $\wedge$ *val-to-bool v*
assumes *ce*: *ConditionalEliminationStep conds stamps g nid g'*

shows $\exists$ *nid'* .(*g m h* $\vdash$ *nid* $\rightsquigarrow$ *nid'*) $\longrightarrow$ (*g' m h* $\vdash$ *nid* $\rightsquigarrow$ *nid'*)
using *ce* **using** *assms*
proof (*induct g nid g' rule*: *ConditionalEliminationStep.induct*)
case (*impliesTrue g ifcond cid t f cond conds g'*)
show *?case* **proof** (*cases* (*g m h* $\vdash$ *ifcond* $\rightsquigarrow$ *nid'*))
case *True*
obtain *condv* **where** *condv*: *g m* $\vdash$ *kind g cid* $\mapsto$ *condv*
using *implies.simps impliesTrue.hyps*(*3*) *impliesTrue.prems*(*4*)
using *impliesTrue.hyps*(*2*) *True*
by (*metis ifNodeHasCondEvalStutter impliesTrue.hyps*(*1*))
have *condvTrue*: *val-to-bool condv*

by (*metis condition-implies.intros*(*2*) *condv impliesTrue.hyps*(*2*) *impliesTrue.hyps*(*3*) *impliesTrue.prems*(*1*) *impliesTrue.prems*(*3*) *impliesTrue.prems*(*5*) *implies-true-valid*)
then show *?thesis*
using *constantConditionValid*
using *impliesTrue.hyps*(*1*) *condv impliesTrue.hyps*(*4*)
by *blast*
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* ($g\ m\ h \vdash ifcond \rightsquigarrow nid'$))
case *True*
obtain *condv* **where** *condv*: $g\ m \vdash kind\ g\ cid \mapsto condv$
using *ifNodeHasCondEvalStutter impliesFalse.hyps*(*1*)
using *True* **by** *blast*
have *condvFalse*: *False* = *val-to-bool condv*
by (*metis condition-implies.intros*(*2*) *condv impliesFalse.hyps*(*2*) *impliesFalse.hyps*(*3*) *impliesFalse.prems*(*1*) *impliesFalse.prems*(*3*) *impliesFalse.prems*(*5*) *implies-false-valid*)
then show *?thesis*
using *constantConditionValid*
using *impliesFalse.hyps*(*1*) *condv impliesFalse.hyps*(*4*)
by *blast*
next
case *False*
then show *?thesis*
by *auto*
qed
next
case (*tryFoldTrue g ifcond cid t f cond g′ conds*)
then show *?case* **using** *constantConditionValid tryFoldProofTrue*
using *StutterStep constantConditionTrue* **by** *metis*
next
case (*tryFoldFalse g ifcond cid t f cond g′ conds*)
then show *?case* **using** *constantConditionValid tryFoldProofFalse*
using *StutterStep constantConditionFalse* **by** *metis*
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 *wff*: *wff-graph g* ∧ *wff-stamp g stamps* ∧ *wff-values g*
assumes *nid*: $nid \in ids\ g$
assumes *conds-valid*: $\forall\ c \in conds\ .\ \exists\ v.\ (g\ m \vdash c \mapsto v) \wedge val\text{-}to\text{-}bool\ v$
assumes *ce*: *ConditionalEliminationStep conds stamps g nid g′*

assumes *gstep*: $\exists$ *h nid′*. ($g \vdash$ (*nid*, *m*, *h*) $\rightarrow$ (*nid′*, *m*, *h*))

shows *nid* | $g \sim g'$
using *ce gstep* **using** *assms*
proof (*induct g nid g′ rule*: *ConditionalEliminationStep.induct*)
case (*impliesTrue g ifcond cid t f cond conds g′ stamps*)
from *impliesTrue*(*5*) **obtain** *h* **where** *gstep*: $g \vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*t*, *m*, *h*)
by (*metis IfNode StutterStep condition-implies.intros*(*2*) *ifNodeHasCondEval-Stutter impliesTrue.hyps*(*1*) *impliesTrue.hyps*(*2*) *impliesTrue.hyps*(*3*) *impliesTrue.prems*(*2*) *impliesTrue.prems*(*4*) *implies-true-valid*)
have $g' \vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*t*, *m*, *h*)
using *constantConditionTrue impliesTrue.hyps*(*1*) *impliesTrue.hyps*(*4*) **by** *blast*
then show *?case* **using** *gstep*
by (*metis stepDet strong-noop-bisimilar.intros*)
next
case (*impliesFalse g ifcond cid t f cond conds g′ stamps*)
from *impliesFalse*(*5*) **obtain** *h* **where** *gstep*: $g \vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*f*, *m*, *h*)
by (*metis IfNode condition-implies.intros*(*2*) *ifNodeHasCondEval impliesFalse.hyps*(*1*) *impliesFalse.hyps*(*2*) *impliesFalse.hyps*(*3*) *impliesFalse.prems*(*2*) *impliesFalse.prems*(*4*) *implies-false-valid*)
have $g' \vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*f*, *m*, *h*)
using *constantConditionFalse impliesFalse.hyps*(*1*) *impliesFalse.hyps*(*4*) **by** *blast*
then show *?case* **using** *gstep*
by (*metis stepDet strong-noop-bisimilar.intros*)
next
case (*tryFoldTrue g ifcond cid t f cond stamps g′ conds*)
from *tryFoldTrue*(*5*) **obtain** *val* **where** *g m* $\vdash$ *kind g cid* $\mapsto$ *val*
using *ifNodeHasCondEval tryFoldTrue.hyps*(*1*) **by** *blast*
then have *val-to-bool val*
using *tryFoldProofTrue tryFoldTrue.prems*(*2*) *tryFoldTrue*(*3*)
by *blast*
then obtain *h* **where** *gstep*: $g \vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*t*, *m*, *h*)
using *tryFoldTrue*(*5*)
by (*meson IfNode* ‹*g m* $\vdash$ *kind g cid* $\mapsto$ *val*› *tryFoldTrue.hyps*(*1*))
have $g' \vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*t*, *m*, *h*)
using *constantConditionTrue tryFoldTrue.hyps*(*1*) *tryFoldTrue.hyps*(*4*) **by** *presburger*
then show *?case* **using** *gstep*
by (*metis stepDet strong-noop-bisimilar.intros*)
next
case (*tryFoldFalse g ifcond cid t f cond stamps g′ conds*)
from *tryFoldFalse*(*5*) **obtain** *h* **where** *gstep*: $g \vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*f*, *m*, *h*)
by (*meson IfNode ifNodeHasCondEval tryFoldFalse.hyps*(*1*) *tryFoldFalse.hyps*(*3*) *tryFoldFalse.prems*(*2*) *tryFoldProofFalse*)
have $g' \vdash$ (*ifcond*, *m*, *h*) $\rightarrow$ (*f*, *m*, *h*)
using *constantConditionFalse tryFoldFalse.hyps*(*1*) *tryFoldFalse.hyps*(*4*) **by** *blast*
then show *?case* **using** *gstep*
by (*metis stepDet strong-noop-bisimilar.intros*)
qed

Mostly experimental proofs from here on out.

lemma *if-step*:
  assumes $nid \in ids\ g$
  assumes $(kind\ g\ nid) \in control\text{-}nodes$
  shows $(g\ m\ h \vdash nid \leadsto nid')$
  using *assms* **apply** (*cases kind g nid*) **sorry**

lemma *StepConditionsValid*:
  assumes $\forall\ cond \in set\ conds.\ (g\ m \vdash cond \mapsto v) \wedge val\text{-}to\text{-}bool\ v$
  assumes $Step\ g\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds', flow'))$
  shows $\forall\ cond \in set\ conds'.\ (g\ m \vdash cond \mapsto v) \wedge val\text{-}to\text{-}bool\ v$
  using *assms*(*2*)
proof (*induction* (*nid*, *seen*, *conds*, *flow*) *Some* (*nid′*, *seen′*, *conds′*, *flow′*) *rule*: *Step.induct*)
  case (*1 ifcond cond t f i c*)
  obtain *cv* **where** *cv*: $g\ m \vdash c \mapsto cv$
    sorry
  have *cvt*: *val-to-bool cv*
    sorry
  have $set\ conds' = \{c\} \cup set\ conds$
    using *1.hyps*(*8*) **by** *auto*
  then show *?case* **using** *cv cvt assms*(*1*) **sorry**
next
  case (*2*)
  from *2*(*5*) **have** $set\ conds' \subseteq set\ conds$
    by (*metis list.sel*(*2*) *list.set-sel*(*2*) *subsetI*)
  then show *?case* **using** *assms*(*1*)
    by *blast*
next
case (*3*)
  then show *?case*
    using *assms*(*1*) **by** *force*
qed

lemma *ConditionalEliminationPhaseProof*:
  assumes *wff-graph g*
  assumes *wff-stamps g*
  assumes $ConditionalEliminationPhase\ g\ (0, \{\}, [], [])\ g'$

  shows $\exists nid'.(g\ m\ h \vdash 0 \leadsto nid') \longrightarrow (g'\ m\ h \vdash 0 \leadsto nid')$
proof –
  have $0 \in ids\ g$
    using *assms*(*1*) *wff-folds* **by** *blast*
  show *?thesis*
using *assms*(*3*) *assms* **proof** (*induct rule*: *ConditionalEliminationPhase.induct*)
case (*1 g nid g′ succs nid′ g′′*)
  then show *?case* **sorry**
next
  case (*2 succs g nid nid′ g′′*)

```
  then show ?case sorry
next
  case (3 succs g nid)
  then show ?case
    by simp
next
  case (4)
  then show ?case sorry
qed
qed

end
```

3 Graph Construction Phase

```
theory
  Construction
imports
  Proofs.Bisimulation
  Proofs.IRGraphFrames
begin

lemma add-const-nodes:
  assumes xn: kind g x = (ConstantNode (IntVal b xv))
  assumes yn: kind g y = (ConstantNode (IntVal b yv))
  assumes zn: kind g z = (AddNode x y)
  assumes wn: kind g w = (ConstantNode (intval-add (IntVal b xv) (IntVal b yv)))
  assumes val: intval-add (IntVal b xv) (IntVal b yv) = IntVal b v1
  assumes ez: g m ⊢ (kind g z) ↦ (IntVal b v1)
  assumes ew: g m ⊢ (kind g w) ↦ (IntVal b v2)
  shows v1 = v2
proof –
  have zv: g m ⊢ (kind g z) ↦ IntVal b v1
    using eval.AddNode eval.ConstantNode xn yn zn val by metis
  have wv: g m ⊢ (kind g w) ↦ IntVal b v2
    using eval.ConstantNode wn ew by blast
  show ?thesis using evalDet zv wv ew ez
    using ConstantNode val wn by auto
qed



lemma add-val-xzero:
  shows intval-add (IntVal b 0) (IntVal b yv) = (IntVal b yv)
  unfolding intval-add.simps sorry

lemma add-val-yzero:
  shows intval-add (IntVal b xv) (IntVal b 0) = (IntVal b xv)
  unfolding intval-add.simps sorry
```

```
fun create-add :: IRGraph ⇒ ID ⇒ ID ⇒ IRNode where
  create-add g x y =
    (case (kind g x) of
      ConstantNode (IntVal b xv) ⇒
        (case (kind g y) of
          ConstantNode (IntVal b yv) ⇒
            ConstantNode (intval-add (IntVal b xv) (IntVal b yv)) |
          - ⇒ if xv = 0 then RefNode y else AddNode x y
        ) |
      - ⇒ (case (kind g y) of
            ConstantNode (IntVal b yv) ⇒
              if yv = 0 then RefNode x else AddNode x y |
            - ⇒ AddNode x y
          )
    )
```

```
lemma add-node-create:
  assumes xv: g m ⊢ (kind g x) ↦ IntVal b xv
  assumes yv: g m ⊢ (kind g y) ↦ IntVal b yv
  assumes res: res = intval-add (IntVal b xv) (IntVal b yv)
  shows
    (g m ⊢ (AddNode x y) ↦ res) ∧
     (g m ⊢ (create-add g x y) ↦ res)

proof –
  let ?P = (g m ⊢ (AddNode x y) ↦ res)
  let ?Q = (g m ⊢ (create-add g x y) ↦ res)
  have P: ?P
    using xv yv res eval.AddNode by blast
  have Q: ?Q
  proof (cases is-ConstantNode (kind g x))
    case xconst: True
    then show ?thesis
    proof (cases is-ConstantNode (kind g y))
      case yconst: True
      have create-add g x y = ConstantNode res
        using xconst yconst
        using ConstantNodeE is-ConstantNode-def xv yv res by auto
      then show ?thesis using eval.ConstantNode by simp
    next
      case ynotconst: False
      have kind g x = ConstantNode (IntVal b xv)
        using ConstantNodeE xconst
        by (metis is-ConstantNode-def xv)
      then have add-def:
        create-add g x y = (if xv = 0 then RefNode y else AddNode x y)
```

```
        using xconst ynotconst is-ConstantNode-def
        unfolding create-add.simps
        by (simp split: IRNode.split)
      then show ?thesis
      proof (cases xv = 0)
        case xzero: True
        have ref: create-add g x y = RefNode y
          using xzero add-def
          by meson
        have refval: g m ⊢ RefNode y ↦ IntVal b yv
          using eval.RefNode yv by simp
        have res = IntVal b yv
          using res unfolding xzero add-val-xzero by simp
        then show ?thesis using xzero ref refval by simp
      next
        case xnotzero: False
        then show ?thesis
          using P add-def by presburger
      qed
    qed
next
  case notxconst: False
  then show ?thesis
    proof (cases is-ConstantNode (kind g y))
      case yconst: True
      have kind g y = ConstantNode (IntVal b yv)
        using ConstantNodeE yconst
        by (metis is-ConstantNode-def yv)
      then have add-def:
        create-add g x y = (if yv = 0 then RefNode x else AddNode x y)
        using notxconst yconst is-ConstantNode-def
        unfolding create-add.simps
        by (simp split: IRNode.split)
      then show ?thesis
      proof (cases yv = 0)
        case yzero: True
        have ref: create-add g x y = RefNode x
          using yzero add-def
          by meson
        have refval: g m ⊢ RefNode x ↦ IntVal b xv
          using eval.RefNode xv by simp
        have res = IntVal b xv
          using res unfolding yzero add-val-yzero by simp
        then show ?thesis using yzero ref refval by simp
      next
        case ynotzero: False
        then show ?thesis
          using P add-def by presburger
      qed
```

next
case *notyconst*: *False*
have *create-add g x y* = *AddNode x y*
using *notxconst notyconst is-ConstantNode-def*
create-add.simps **by** (*simp split*: *IRNode.split*)
then show *?thesis*
using *P* **by** *presburger*
qed
qed
from *P Q* **show** *?thesis* **by** *simp*
qed

fun *add-node-fake* :: *ID* ⇒ *IRNode* ⇒ *IRGraph* ⇒ *IRGraph* **where**
add-node-fake nid k g = *add-node nid* (*k*, *VoidStamp*) *g*
lemma *add-node-lookup-fake*:
assumes *gup* = *add-node-fake nid k g*
assumes *nid* ∉ *ids g*
shows *kind gup nid* = *k*
using *add-node-lookup* **proof** (*cases k* = *NoNode*)
case *True*
have *kind g nid* = *NoNode*
using *assms*(*2*)
using *not-in-g* **by** *blast*
then show *?thesis* **using** *assms*
by (*metis add-node-fake.simps add-node-lookup*)
next
case *False*
then show *?thesis*
by (*simp add*: *add-node-lookup assms*(*1*))
qed
lemma *add-node-unchanged-fake*:
assumes *new* ∉ *ids g*
assumes *nid* ∈ *ids g*
assumes *gup* = *add-node-fake new k g*
assumes *wff-graph g*
shows *unchanged* (*eval-usages g nid*) *g gup*
using *add-node-fake.simps add-node-unchanged assms* **by** *blast*

lemma *dom-add-unchanged*:
assumes *nid* ∈ *ids g*
assumes *g′* = *add-node-fake n k g*
assumes *nid* ≠ *n*
shows *nid* ∈ *ids g′*
using *add-changed assms*(*1*) *assms*(*2*) *assms*(*3*) **by** *force*

lemma *preserve-wff*:
assumes *wff*: *wff-graph g*

assumes *nid* $\notin$ *ids g*
assumes *closed*: *inputs* g' *nid* $\cup$ *succ* g' *nid* $\subseteq$ *ids g*
assumes g': g' = *add-node-fake nid k g*
shows *wff-graph* g'
using *assms* **unfolding** *wff-folds*
apply (*intro conjI*)
  apply (*metis dom-add-unchanged*)
 apply (*metis add-node-unchanged-fake assms*(*1*) *kind-unchanged*)
sorry

lemma *equal-closure-bisimilar*:
assumes $\{P'. (g\ m\ h \vdash nid \leadsto P')\} = \{P'. (g'\ m\ h \vdash nid \leadsto P')\}$
shows *nid* . $g \sim g'$
by (*metis assms weak-bisimilar.simps mem-Collect-eq*)

lemma *wff-size*:
assumes *nid* $\in$ *ids g*
assumes *wff-graph g*
assumes *is-AbstractEndNode* (*kind g nid*)
shows *card* (*usages g nid*) > *0*
using *assms* **unfolding** *wff-folds*
by *fastforce*

lemma *sequentials-have-successors*:
assumes *is-sequential-node n*
shows *size* (*successors-of n*) > *0*
using *assms* **by** (*cases n*; *auto*)

lemma *step-reaches-successors-only*:
assumes $(g \vdash (nid, m, h) \rightarrow (nid', m, h))$
assumes *wff*: *wff-graph g*
shows $nid' \in$ *succ g nid* $\vee$ $nid' \in$ *usages g nid*
using *assms* **proof** (*induct* (*nid*, *m*, *h*) (*nid'*, *m*, *h*)*rule*: *step.induct*)
case *SequentialNode*
then show *?case* **using** *sequentials-have-successors*
 by (*metis nth-mem succ.simps*)
next
case (*IfNode cond tb fb val*)
then show *?case* **using** *successors-of-IfNode*
 by (*simp add*: *IfNode.hyps*(*1*))
next
case (*EndNodes i phis inputs vs*)
have *nid* $\in$ *ids g*
 using *assms*(*1*) *step-in-ids*
 by *blast*
then have *usage-size*: *card* (*usages g nid*) > *0*
 using *wff EndNodes*(*1*) *wff-size*
 by *blast*
then have *usage-size*: *size* (*sorted-list-of-set* (*usages g nid*)) > *0*

by (*metis length-sorted-list-of-set*)
have *usages g nid* $\subseteq$ *ids g*
using *wff* **by** *fastforce*
then have *finite-usage*: *finite* (*usages g nid*)
by (*metis bot-nat-0.extremum-strict list.size*(*3*) *sorted-list-of-set.infinite usage-size*)
from *EndNodes*(*2*) **have** *nid'* $\in$ *usages g nid*
unfolding *any-usage.simps*
using *usage-size finite-usage*
by (*metis hd-in-set length-greater-0-conv sorted-list-of-set*(*1*))
then show *?case*
by *simp*
next
case (*NewInstanceNode f obj ref*)
then show *?case* **using** *successors-of-NewInstanceNode* **by** *simp*
next
case (*LoadFieldNode f obj ref v*)
then show *?case* **by** *simp*
next
case (*SignedDivNode x y zero sb v1 v2 v*)
then show *?case* **by** *simp*
next
case (*SignedRemNode x y zero sb v1 v2 v*)
then show *?case* **by** *simp*
next
case (*StaticLoadFieldNode f v*)
then show *?case* **by** *simp*
next
case (*StoreFieldNode f newval uu obj val ref*)
then show *?case* **by** *simp*
next
case (*StaticStoreFieldNode f newval uv val*)
then show *?case* **by** *simp*
qed

lemma *stutter-closed*:
assumes *g m h* $\vdash$ *nid* $\rightsquigarrow$ *nid'*
assumes *wff-graph g*
shows $\exists$ *n* $\in$ *ids g* . *nid'* $\in$ *succ g n* $\vee$ *nid'* $\in$ *usages g n*
using *assms*
proof (*induct nid nid' rule*: *stutter.induct*)
case (*StutterStep nid nid'*)
have *nid* $\in$ *ids g*
using *StutterStep.hyps step-in-ids* **by** *blast*
then show *?case* **using** *StutterStep step-reaches-successors-only*
by *blast*
next
case (*Transitive nid nid'' nid'*)
then show *?case*

by *blast*
qed

lemma *unchanged-step*:
assumes $g \vdash (nid, m, h) \rightarrow (nid', m, h)$
assumes *wff*: *wff-graph g*
assumes *kind*: *kind g nid* = *kind g' nid*
assumes *unchanged*: *unchanged* (*eval-usages g nid*) *g g'*
assumes *succ*: *succ g nid* = *succ g' nid*

shows $g' \vdash (nid, m, h) \rightarrow (nid', m, h)$
using *assms* **proof** (*induct* (*nid*, *m*, *h*) (*nid'*, *m*, *h*) *rule*: *step.induct*)
case *SequentialNode*
then show *?case*
by (*metis step.SequentialNode*)
next
case (*IfNode cond tb fb val*)
then show *?case* **using** *stay-same step.IfNode*
by (*metis* (*no-types*, *lifting*) *IRNodes.inputs-of-IfNode child-unchanged inputs.elims list.set-intros*(*1*))
next
case (*EndNodes i phis inputs vs*)
then show *?case* **sorry**
next
case (*NewInstanceNode f obj ref*)
then show *?case* **using** *step.NewInstanceNode*
by *metis*
next
case (*LoadFieldNode f obj ref v*)
have $obj \in$ *inputs g nid*
using *LoadFieldNode*(*1*) *inputs-of-LoadFieldNode*
using *opt-to-list.simps*
by (*simp add*: *LoadFieldNode.hyps*(*1*))
then have *unchanged* (*eval-usages g obj*) *g g'*
using *unchanged*
using *child-unchanged* **by** *blast*
then have $g'\ m \vdash$ *kind g' obj* $\mapsto$ *ObjRef ref*
using *unchanged wff stay-same*
using *LoadFieldNode.hyps*(*2*) **by** *presburger*
then show *?case* **using** *step.LoadFieldNode*
by (*metis LoadFieldNode.hyps*(*1*) *LoadFieldNode.hyps*(*3*) *LoadFieldNode.hyps*(*4*) *assms*(*3*))
next
case (*SignedDivNode x y zero sb v1 v2 v*)
have $x \in$ *inputs g nid*
using *SignedDivNode*(*1*) *inputs-of-SignedDivNode*
using *opt-to-list.simps*
by (*simp add*: *SignedDivNode.hyps*(*1*))

then have *unchanged* (*eval-usages g x*) *g g'*
using *unchanged*
using *child-unchanged* **by** *blast*
then have *g' m* ⊢ *kind g' x* ↦ *v1*
using *unchanged wff stay-same*
using *SignedDivNode.hyps*(*2*) **by** *presburger*
have *y* ∈ *inputs g nid*
using *SignedDivNode*(*1*) *inputs-of-SignedDivNode*
using *opt-to-list.simps*
by (*simp add*: *SignedDivNode.hyps*(*1*))
then have *unchanged* (*eval-usages g y*) *g g'*
using *unchanged*
using *child-unchanged* **by** *blast*
then have *g' m* ⊢ *kind g' y* ↦ *v2*
using *unchanged wff stay-same*
using *SignedDivNode.hyps*(*3*) **by** *presburger*
then show *?case* **using** *step.SignedDivNode*
by (*metis SignedDivNode.hyps*(*1*) *SignedDivNode.hyps*(*4*) *SignedDivNode.hyps*(*5*)
‹*g' m* ⊢ *kind g' x* ↦ *v1*› *kind*)
next
case (*SignedRemNode x y zero sb v1 v2 v*)
have *x* ∈ *inputs g nid*
using *SignedRemNode*(*1*) *inputs-of-SignedRemNode*
using *opt-to-list.simps*
by (*simp add*: *SignedRemNode.hyps*(*1*))
then have *unchanged* (*eval-usages g x*) *g g'*
using *unchanged*
using *child-unchanged* **by** *blast*
then have *g' m* ⊢ *kind g' x* ↦ *v1*
using *unchanged wff stay-same*
using *SignedRemNode.hyps*(*2*) **by** *presburger*
have *y* ∈ *inputs g nid*
using *SignedRemNode*(*1*) *inputs-of-SignedRemNode*
using *opt-to-list.simps*
by (*simp add*: *SignedRemNode.hyps*(*1*))
then have *unchanged* (*eval-usages g y*) *g g'*
using *unchanged*
using *child-unchanged* **by** *blast*
then have *g' m* ⊢ *kind g' y* ↦ *v2*
using *unchanged wff stay-same*
using *SignedRemNode.hyps*(*3*) **by** *presburger*
then show *?case*
by (*metis SignedRemNode.hyps*(*1*) *SignedRemNode.hyps*(*4*) *SignedRemNode.hyps*(*5*)
‹*g' m* ⊢ *kind g' x* ↦ *v1*› *kind step.SignedRemNode*)
next
case (*StaticLoadFieldNode f v*)
then show *?case* **using** *step.StaticLoadFieldNode*
by *metis*
next

case (*StoreFieldNode f newval uu obj val ref*)
have *obj* $\in$ *inputs g nid*
using *StoreFieldNode*(*1*) *inputs-of-StoreFieldNode*
using *opt-to-list.simps*
by (*simp add*: *StoreFieldNode.hyps*(*1*))
then have *unchanged* (*eval-usages g obj*) *g* g'
using *unchanged*
using *child-unchanged* **by** *blast*
then have g' *m* $\vdash$ *kind* g' *obj* $\mapsto$ *ObjRef ref*
using *unchanged wff stay-same*
using *StoreFieldNode.hyps*(*3*) **by** *presburger*
have *newval* $\in$ *inputs g nid*
using *StoreFieldNode*(*1*) *inputs-of-StoreFieldNode*
using *opt-to-list.simps*
by (*simp add*: *StoreFieldNode.hyps*(*1*))
then have *unchanged* (*eval-usages g newval*) *g* g'
using *unchanged*
using *child-unchanged* **by** *blast*
then have g' *m* $\vdash$ *kind* g' *newval* $\mapsto$ *val*
using *unchanged wff stay-same*
using *StoreFieldNode.hyps*(*2*) **by** *blast*
then show *?case* **using** *step.StoreFieldNode*
by (*metis StoreFieldNode.hyps*(*1*) *StoreFieldNode.hyps*(*4*) *StoreFieldNode.hyps*(*5*) $\langle g'$ *m* $\vdash$ *kind* g' *obj* $\mapsto$ *ObjRef ref*$\rangle$ *assms*(*3*))
next
case (*StaticStoreFieldNode f newval uv val*)
have *newval* $\in$ *inputs g nid*
using *StoreFieldNode*(*1*) *inputs-of-StoreFieldNode*
using *opt-to-list.simps*
by (*simp add*: *StaticStoreFieldNode.hyps*(*1*))
then have *unchanged* (*eval-usages g newval*) *g* g'
using *unchanged*
using *child-unchanged* **by** *blast*
then have g' *m* $\vdash$ *kind* g' *newval* $\mapsto$ *val*
using *unchanged wff stay-same*
using *StaticStoreFieldNode.hyps*(*2*) **by** *blast*
then show *?case* **using** *step.StaticStoreFieldNode*
by (*metis StaticStoreFieldNode.hyps*(*1*) *StaticStoreFieldNode.hyps*(*3*) *StaticStoreFieldNode.hyps*(*4*) *kind*)
qed

lemma *unchanged-closure*:
assumes *nid* $\notin$ *ids g*
assumes *wff*: *wff-graph g* $\wedge$ *wff-graph* g'
assumes g': g' = *add-node-fake nid k g*
assumes nid' $\in$ *ids g*
shows (*g m h* $\vdash$ nid' $\rightsquigarrow$ nid'') $\longleftrightarrow$ (g' *m h* $\vdash$ nid' $\rightsquigarrow$ nid'')
(**is** *?P* $\longleftrightarrow$ *?Q*)

```
proof
  assume P: ?P
  have niddiff: nid ≠ nid′
    using assms
    by blast
  from P show ?Q using assms niddiff
  proof (induction rule: stutter.induct)
    case (StutterStep start e)
    have unchanged: unchanged (eval-usages g start) g g′
     using StutterStep.prems(4) add-node-unchanged-fake assms(1) g′ wff by blast
    have succ-same: succ g start = succ g′ start
       using StutterStep.prems(4) kind-unchanged succ.simps unchanged by pres-
burger
    have kind g start = kind g′ start
        by (metis StutterStep.prems(4) add-node-fake.elims add-node-unchanged
assms(1) assms(2) g′ kind-unchanged)
    then have g′ ⊢ (start, m, h) → (e, m, h)
      using unchanged-step wff unchanged succ-same
      by (meson StutterStep.hyps)
    then show ?case
      using stutter.StutterStep by blast
  next
    case (Transitive nid nid′′ nid′)
    then show ?case
    by (metis add-node-unchanged-fake kind-unchanged step-in-ids stutter.Transitive
stutter.cases succ.simps unchanged-step)
  qed
next
  assume Q: ?Q
  have niddiff: nid ≠ nid′
    using assms
    by blast
  from Q show ?P using assms niddiff
  proof (induction rule: stutter.induct)
    case (StutterStep start e)
    have eval-usages g′ start ⊆ eval-usages g start
      using g′ eval-usages sorry
    then have unchanged: unchanged (eval-usages g′ start) g′ g
        by (smt (verit, ccfv-SIG) StutterStep.prems(4) add-node-unchanged-fake
assms(1) g′ subset-iff unchanged.simps wff)
    have succ-same: succ g start = succ g′ start
       using StutterStep.prems(4) eval-usages-self node-unchanged succ.simps un-
changed
      by (metis (no-types, lifting) StutterStep.hyps step-in-ids)
    have kind g start = kind g′ start
        by (metis StutterStep.prems(4) add-node-fake.elims add-node-unchanged
assms(1) assms(2) g′ kind-unchanged)
    then have g ⊢ (start, m, h) → (e, m, h)
      using StutterStep(1) wff unchanged-step unchanged succ-same
```

    sorry
  then show *?case*
    using *stutter.StutterStep* **by** *blast*
next
  case (*Transitive nid nid″ nid′*)
  then show *?case*
    using *add-node-unchanged-fake kind-unchanged step-in-ids stutter.Transitive stutter.cases succ.simps unchanged-step*
    sorry
qed
qed

fun *create-if* :: *IRGraph* ⇒ *ID* ⇒ *ID* ⇒ *ID* ⇒ *IRNode*
  where
  create-if g cond tb fb =
    (*case* (*kind g cond*) *of*
      ConstantNode condv ⇒
        RefNode (*if* (*val-to-bool condv*) *then tb else fb*) |
      - ⇒ (*if tb* = *fb then*
            RefNode tb
          else
            IfNode cond tb fb)
)

lemma *if-node-create-bisimulation*:
  fixes *h* :: *FieldRefHeap*
  assumes *wff*: *wff-graph g*
  assumes *cv*: *g m* ⊢ (*kind g cond*) ↦ *cv*
  assumes *fresh*: *nid* ∉ *ids g*
  assumes *closed*: {*cond*, *tb*, *fb*} ⊆ *ids g*
  assumes *gif*: *gif* = *add-node-fake nid* (*IfNode cond tb fb*) *g*
  assumes *gcreate*: *gcreate* = *add-node-fake nid* (*create-if g cond tb fb*) *g*

  shows *nid* . *gif* ∼ *gcreate*

proof −
  have *indep*: ¬(*eval-uses g cond nid*)
    using *cv eval-in-ids fresh no-external-use wff* **by** *blast*
  have *kind gif nid* = *IfNode cond tb fb*
    using *gif add-node-lookup* **by** *simp*
  then have {*cond*, *tb*, *fb*} = *inputs gif nid* ∪ *succ gif nid*
    using *inputs-of-IfNode successors-of-IfNode*
    by (*metis empty-set inputs.simps insert-is-Un list.simps*(*15*) *succ.simps*)
  then have *wff-gif*: *wff-graph gif*
    using *closed wff preserve-wff*
    using *fresh gif* **by** *presburger*
  have *create-if g cond tb fb* = *IfNode cond tb fb* ∨
        create-if g cond tb fb = *RefNode tb* ∨
        create-if g cond tb fb = *RefNode fb*

by (*cases kind g cond*; *auto*)
then have *kind gcreate nid* = *IfNode cond tb fb* ∨
kind gcreate nid = *RefNode tb* ∨
kind gcreate nid = *RefNode fb*
using *gcreate add-node-lookup*
using *add-node-lookup-fake fresh* **by** *presburger*
then have *inputs gcreate nid* ∪ *succ gcreate nid* ⊆ {*cond*, *tb*, *fb*}
using *inputs-of-IfNode successors-of-IfNode inputs-of-RefNode successors-of-RefNode*
by *force*
then have *wff-gcreate*: *wff-graph gcreate*
using *closed wff preserve-wff fresh gcreate*
by (*metis subset-trans*)
have *tb-unchanged*: {*nid'*. (*gif m h* ⊢ *tb* ⇝ *nid'*)} = {*nid'*. (*gcreate m h* ⊢ *tb* ⇝ *nid'*)}
proof −
have ¬(∃ *n* ∈ *ids g*. *nid* ∈ *succ g n* ∨ *nid* ∈ *usages g n*)
using *wff*
by (*metis* (*no-types*, *lifting*) *fresh mem-Collect-eq subsetD usages.simps wff-folds*(*1*,*3*))
then have *nid* ∉ {*nid'*. (*g m h* ⊢ *tb* ⇝ *nid'*)}
using *wff stutter-closed*
by (*metis mem-Collect-eq*)
have *gif-set*: {*nid'*. (*gif m h* ⊢ *tb* ⇝ *nid'*)} = {*nid'*. (*g m h* ⊢ *tb* ⇝ *nid'*)}
using *unchanged-closure fresh wff gif closed wff-gif*
by *blast*
have *gcreate-set*: {*nid'*. (*gcreate m h* ⊢ *tb* ⇝ *nid'*)} = {*nid'*. (*g m h* ⊢ *tb* ⇝ *nid'*)}
using *unchanged-closure fresh wff gcreate closed wff-gcreate*
by *blast*
from *gif-set gcreate-set* **show** *?thesis* **by** *simp*
qed
have *fb-unchanged*: {*nid'*. (*gif m h* ⊢ *fb* ⇝ *nid'*)} = {*nid'*. (*gcreate m h* ⊢ *fb* ⇝ *nid'*)}
proof −
have ¬(∃ *n* ∈ *ids g*. *nid* ∈ *succ g n* ∨ *nid* ∈ *usages g n*)
using *wff*
by (*metis* (*no-types*, *lifting*) *fresh mem-Collect-eq subsetD usages.simps wff-folds*(*1*,*3*))
then have *nid* ∉ {*nid'*. (*g m h* ⊢ *fb* ⇝ *nid'*)}
using *wff stutter-closed*
by (*metis mem-Collect-eq*)
have *gif-set*: {*nid'*. (*gif m h* ⊢ *fb* ⇝ *nid'*)} = {*nid'*. (*g m h* ⊢ *fb* ⇝ *nid'*)}
using *unchanged-closure fresh wff gif closed wff-gif*
by *blast*
have *gcreate-set*: {*nid'*. (*gcreate m h* ⊢ *fb* ⇝ *nid'*)} = {*nid'*. (*g m h* ⊢ *fb* ⇝ *nid'*)}
using *unchanged-closure fresh wff gcreate closed wff-gcreate*
by *blast*
from *gif-set gcreate-set* **show** *?thesis* **by** *simp*

qed
show *?thesis*
proof (*cases* $\exists$ *val* . (*kind g cond*) = *ConstantNode val*)
let *?gif-closure* = $\{P'$. (*gif m h* $\vdash$ *nid* $\leadsto$ P')$\}$
let *?gcreate-closure* = $\{P'$. (*gcreate m h* $\vdash$ *nid* $\leadsto$ P')$\}$
case *constantCond*: *True*
obtain *val* **where** *val*: (*kind g cond*) = *ConstantNode val*
using *constantCond* **by** *blast*
then show *?thesis*
proof (*cases val-to-bool val*)
case *constantTrue*: *True*
have *if-kind*: *kind gif nid* = (*IfNode cond tb fb*)
using *gif add-node-lookup* **by** *simp*
have *if-cv*: *gif m* $\vdash$ (*kind gif cond*) $\mapsto$ *val*
by (*metis ConstantNodeE add-node-unchanged-fake cv eval-in-ids fresh gif stay-same val wff*)
have (*gif* $\vdash$ (*nid*, *m*, *h*) $\rightarrow$ (*tb*, *m*, *h*))
using *step.IfNode if-kind if-cv*
using *constantTrue* **by** *presburger*
then have *gif-closure*: *?gif-closure* = $\{tb\} \cup \{nid'$. (*gif m h* $\vdash$ *tb* $\leadsto$ *nid'*)$\}$
using *stuttering-successor* **by** *presburger*
have *ref-kind*: *kind gcreate nid* = (*RefNode tb*)
using *gcreate add-node-lookup constantTrue constantCond* **unfolding** *create-if.simps*
by (*simp add*: *val*)
have (*gcreate* $\vdash$ (*nid*, *m*, *h*) $\rightarrow$ (*tb*, *m*, *h*))
using *stepRefNode ref-kind* **by** *simp*
then have *gcreate-closure*: *?gcreate-closure* = $\{tb\} \cup \{nid'$. (*gcreate m h* $\vdash$ *tb* $\leadsto$ *nid'*)$\}$
using *stuttering-successor*
by *auto*
from *gif-closure gcreate-closure* **have** *?gif-closure* = *?gcreate-closure*
using *tb-unchanged* **by** *simp*
then show *?thesis*
using *equal-closure-bisimilar* **by** *simp*
next
case *constantFalse*: *False*
have *if-kind*: *kind gif nid* = (*IfNode cond tb fb*)
using *gif add-node-lookup* **by** *simp*
have *if-cv*: *gif m* $\vdash$ (*kind gif cond*) $\mapsto$ *val*
by (*metis ConstantNodeE add-node-unchanged-fake cv eval-in-ids fresh gif stay-same val wff*)
have (*gif* $\vdash$ (*nid*, *m*, *h*) $\rightarrow$ (*fb*, *m*, *h*))
using *step.IfNode if-kind if-cv*
using *constantFalse* **by** *presburger*
then have *gif-closure*: *?gif-closure* = $\{fb\} \cup \{nid'$. (*gif m h* $\vdash$ *fb* $\leadsto$ *nid'*)$\}$
using *stuttering-successor* **by** *presburger*
have *ref-kind*: *kind gcreate nid* = *RefNode fb*
using *add-node-lookup-fake constantFalse fresh gcreate val* **by** *force*

then have ($gcreate \vdash (nid, m, h) \rightarrow (fb, m, h)$)
using *stepRefNode* **by** *presburger*
then have *gcreate-closure*: $?gcreate\text{-}closure = \{fb\} \cup \{nid'. (gcreate\ m\ h \vdash fb \rightsquigarrow nid')\}$
using *stuttering-successor* **by** *presburger*
from *gif-closure gcreate-closure* **have** $?gif\text{-}closure = ?gcreate\text{-}closure$
using *fb-unchanged* **by** *simp*
then show *?thesis*
using *equal-closure-bisimilar* **by** *simp*
qed
next
let $?gif\text{-}closure = \{P'. (gif\ m\ h \vdash nid \rightsquigarrow P')\}$
let $?gcreate\text{-}closure = \{P'. (gcreate\ m\ h \vdash nid \rightsquigarrow P')\}$
case *notConstantCond*: *False*
then show *?thesis*
proof (*cases* $tb = fb$)
case *equalBranches*: *True*
have *if-kind*: $kind\ gif\ nid = (IfNode\ cond\ tb\ fb)$
using *gif add-node-lookup* **by** *simp*
have ($gif \vdash (nid, m, h) \rightarrow (tb, m, h)) \vee (gif \vdash (nid, m, h) \rightarrow (fb, m, h)$)
using *step.IfNode if-kind cv* **apply** (*cases val-to-bool cv*)
apply (*metis add-node-fake.simps add-node-unchanged eval-in-ids fresh gif stay-same wff*)
by (*metis add-node-unchanged-fake eval-in-ids fresh gif stay-same wff*)
then have *gif-closure*: $?gif\text{-}closure = \{tb\} \cup \{nid'. (gif\ m\ h \vdash tb \rightsquigarrow nid')\}$
using *equalBranches*
using *stuttering-successor* **by** *presburger*
have *iref-kind*: $kind\ gcreate\ nid = (RefNode\ tb)$
using *gcreate add-node-lookup notConstantCond equalBranches*
unfolding *create-if.simps*
by (*cases* (*kind g cond*); *auto*)
then have ($gcreate \vdash (nid, m, h) \rightarrow (tb, m, h)$)
using *stepRefNode* **by** *simp*
then have *gcreate-closure*: $?gcreate\text{-}closure = \{tb\} \cup \{nid'. (gcreate\ m\ h \vdash tb \rightsquigarrow nid')\}$
using *stuttering-successor* **by** *presburger*
from *gif-closure gcreate-closure* **have** $?gif\text{-}closure = ?gcreate\text{-}closure$
using *tb-unchanged* **by** *simp*
then show *?thesis*
using *equal-closure-bisimilar* **by** *simp*
next
case *uniqueBranches*: *False*
let $?tb\text{-}closure = \{tb\} \cup \{nid'. (gif\ m\ h \vdash tb \rightsquigarrow nid')\}$
let $?fb\text{-}closure = \{fb\} \cup \{nid'. (gif\ m\ h \vdash fb \rightsquigarrow nid')\}$
have *if-kind*: $kind\ gif\ nid = (IfNode\ cond\ tb\ fb)$
using *gif add-node-lookup* **by** *simp*
have *if-step*: ($gif \vdash (nid, m, h) \rightarrow (tb, m, h)) \vee (gif \vdash (nid, m, h) \rightarrow (fb, m, h)$)
using *step.IfNode if-kind cv* **apply** (*cases val-to-bool cv*)

apply (*metis add-node-fake.simps add-node-unchanged eval-in-ids fresh gif stay-same wff*)
by (*metis add-node-unchanged-fake eval-in-ids fresh gif stay-same wff*)
then have *gif-closure*: *?gif-closure* = *?tb-closure* ∨ *?gif-closure* = *?fb-closure*
using *stuttering-successor* **by** *presburger*
have *gc-kind*: *kind gcreate nid* = (*IfNode cond tb fb*)
using *gcreate add-node-lookup notConstantCond uniqueBranches*
unfolding *create-if.simps*
by (*cases* (*kind g cond*); *auto*)
then have (*gcreate* ⊢ (*nid*, *m*, *h*) → (*tb*, *m*, *h*)) ∨ (*gcreate* ⊢ (*nid*, *m*, *h*) → (*fb*, *m*, *h*))
by (*metis add-node-lookup-fake fresh gcreate gif if-step*)
then have *gcreate-closure*: *?gcreate-closure* = *?tb-closure* ∨ *?gcreate-closure* = *?fb-closure*
by (*metis add-node-lookup-fake fresh gc-kind gcreate gif gif-closure*)
from *gif-closure gcreate-closure* **have** *?gif-closure* = *?gcreate-closure*
using *tb-unchanged fb-unchanged*
by (*metis add-node-lookup-fake fresh gc-kind gcreate gif*)
then show *?thesis*
using *equal-closure-bisimilar* **by** *simp*
qed
qed
qed

lemma *if-node-create*:
assumes *wff*: *wff-graph g*
assumes *cv*: *g m* ⊢ (*kind g cond*) ↦ *cv*
assumes *fresh*: *nid* ∉ *ids g*
assumes *gif*: *gif* = *add-node-fake nid* (*IfNode cond tb fb*) *g*
assumes *gcreate*: *gcreate* = *add-node-fake nid* (*create-if g cond tb fb*) *g*
shows ∃ *nid'*. (*gif m h* ⊢ *nid* ⇝ *nid'*) ∧ (*gcreate m h* ⊢ *nid* ⇝ *nid'*)

proof (*cases* ∃ *val* . (*kind g cond*) = *ConstantNode val*)
case *True*
show *?thesis*
proof −
obtain *val* **where** *val*: (*kind g cond*) = *ConstantNode val*
using *True* **by** *blast*
have *cond-exists*: *cond* ∈ *ids g*
using *cv eval-in-ids* **by** *auto*
have *if-kind*: *kind gif nid* = (*IfNode cond tb fb*)
using *gif add-node-lookup* **by** *simp*
have *if-cv*: *gif m* ⊢ (*kind gif cond*) ↦ *val*
using *step.IfNode if-kind*
using *True eval.ConstantNode gif fresh*
using *stay-same cond-exists*
using *val*
using *add-node.rep-eq kind.rep-eq* **by** *auto*
have *if-step*: *gif* ⊢ (*nid*,*m*,*h*) → (*if val-to-bool val then tb else fb*,*m*,*h*)

proof −
show *?thesis* **using** *step.IfNode if-kind if-cv*
by (*simp*)
qed
have *create-step*: *gcreate* ⊢ (*nid*,*m*,*h*) → (*if val-to-bool val then tb else fb*,*m*,*h*)
proof −
have *create-kind*: *kind gcreate nid* = (*create-if g cond tb fb*)
using *gcreate add-node-lookup-fake*
using *fresh* **by** *blast*
have *create-fun*: *create-if g cond tb fb* = *RefNode* (*if val-to-bool val then tb else fb*)
using *True create-kind val* **by** *simp*
show *?thesis* **using** *stepRefNode create-kind create-fun if-cv*
by (*simp*)
qed
then show *?thesis* **using** *StutterStep create-step if-step*
by *blast*
qed
next
case *not-const*: *False*
obtain *nid′* **where** *nid′* = (*if val-to-bool cv then tb else fb*)
by *blast*
have *nid-eq*: (*gif* ⊢ (*nid*,*m*,*h*) → (*nid′*,*m*,*h*)) ∧ (*gcreate* ⊢ (*nid*,*m*,*h*) → (*nid′*,*m*,*h*))
proof −
have *indep*: ¬(*eval-uses g cond nid*)
using *no-external-use*
using *cv eval-in-ids fresh wff* **by** *blast*
have *nid′*: *nid′* = (*if val-to-bool cv then tb else fb*)
by (*simp add*: ‹*nid′* = (*if val-to-bool cv then tb else fb*)›)
have *gif-kind*: *kind gif nid* = (*IfNode cond tb fb*)
using *add-node-lookup-fake gif*
using *fresh* **by** *blast*
then have *nid* ≠ *cond*
using *cv fresh indep*
using *eval-in-ids* **by** *blast*
have *unchanged* (*eval-usages g cond*) *g gif*
using *gif add-node-unchanged-fake*
using *cv eval-in-ids fresh wff* **by** *blast*
then obtain *cv2* **where** *cv2*: *gif m* ⊢ (*kind gif cond*) ↦ *cv2*
using *cv gif wff stay-same* **by** *blast*
then have *cv* = *cv2*
using *indep gif cv*
using ‹*nid* ≠ *cond*›
using *fresh*
using ‹*unchanged* (*eval-usages g cond*) *g gif*› *evalDet stay-same wff* **by** *blast*
then have *eval-gif*: (*gif* ⊢ (*nid*,*m*,*h*) → (*nid′*,*m*,*h*))
using *step.IfNode gif-kind nid′ cv2*
by *auto*
have *gcreate-kind*: *kind gcreate nid* = (*create-if g cond tb fb*)

```
      using gcreate add-node-lookup-fake
      using fresh by blast
    have eval-gcreate: gcreate ⊢ (nid,m,h) → (nid′,m,h)
    proof (cases tb = fb)
      case True
      have create-if g cond tb fb = RefNode tb
        using not-const True by (cases (kind g cond); auto)
      then show ?thesis
        using True gcreate-kind nid′ stepRefNode
        by (simp)
    next
      case False
      have create-if g cond tb fb = IfNode cond tb fb
        using not-const False by (cases (kind g cond); auto)
      then show ?thesis
        using eval-gif gcreate gif
        using IfNode ‹cv = cv2› cv2 gif-kind nid′ by auto
    qed
    show ?thesis
      using eval-gcreate eval-gif StutterStep by blast
  qed
  show ?thesis using nid-eq StutterStep by meson
qed

end
```