Unspecified Veriopt Theory

April 24, 2021

Contents

0.1 Stuttering

theory *Stuttering*
  imports
    Semantics.IRStepObj
begin

inductive *stutter*:: *IRGraph* $\Rightarrow$ *MapState* $\Rightarrow$ *FieldRefHeap* $\Rightarrow$ *ID* $\Rightarrow$ *ID* $\Rightarrow$ *bool* (-
- - $\vdash$ - $\rightsquigarrow$ - *55*)
  for g m h **where**

  StutterStep:
  $[\![g \vdash (nid,m,h) \rightarrow (nid',m,h)]\!]$
  $\Longrightarrow g\ m\ h \vdash nid \rightsquigarrow nid'$ |

  Transitive:
  $[\![g \vdash (nid,m,h) \rightarrow (nid'',m,h);$
    $g\ m\ h \vdash nid'' \rightsquigarrow nid']\!]$
  $\Longrightarrow g\ m\ h \vdash nid \rightsquigarrow nid'$

lemma *stuttering-successor*:
  assumes $(g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h))$
  shows $\{P'.\ (g\ m\ h \vdash nid \rightsquigarrow P')\} = \{nid'\} \cup \{nid''.\ (g\ m\ h \vdash nid' \rightsquigarrow nid'')\}$
proof –
  have *nextin*: $nid' \in \{P'.\ (g\ m\ h \vdash nid \rightsquigarrow P')\}$
    using *assms StutterStep* **by** *blast*
  have *nextsubset*: $\{nid''.\ (g\ m\ h \vdash nid' \rightsquigarrow nid'')\} \subseteq \{P'.\ (g\ m\ h \vdash nid \rightsquigarrow P')\}$

by (*metis Collect-mono assms stutter.Transitive*)
have $\forall n \in \{P'.\ (g\ m\ h \vdash nid \rightsquigarrow P')\}\ .\ n = nid' \vee n \in \{nid''.\ (g\ m\ h \vdash nid' \rightsquigarrow nid'')\}$
using *stepDet*
by (*metis* (*no-types*, *lifting*) *Pair-inject assms mem-Collect-eq stutter.simps*)
then show *?thesis*
using *insert-absorb mk-disjoint-insert nextin nextsubset* **by** *auto*
qed

end

1 Proof Infrastructure

1.1 Bisimulation

theory *Bisimulation*
imports
Stuttering
begin

inductive *weak-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
(- . - ∼ -) **for** *nid* **where**
$[\![\forall P'.\ (g\ m\ h \vdash nid \rightsquigarrow P') \longrightarrow (\exists Q'\ .\ (g'\ m\ h \vdash nid \rightsquigarrow Q') \wedge P' = Q');$
$\forall Q'.\ (g'\ m\ h \vdash nid \rightsquigarrow Q') \longrightarrow (\exists P'\ .\ (g\ m\ h \vdash nid \rightsquigarrow P') \wedge P' = Q')]\!]$
$\Longrightarrow nid\ .\ g \sim g'$

A strong bisimilution between no-op transitions

inductive *strong-noop-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
(- | - ∼ -) **for** *nid* **where**
$[\![\forall P'.\ (g \vdash (nid,\ m,\ h) \rightarrow P') \longrightarrow (\exists Q'\ .\ (g' \vdash (nid,\ m,\ h) \rightarrow Q') \wedge P' = Q');$
$\forall Q'.\ (g' \vdash (nid,\ m,\ h) \rightarrow Q') \longrightarrow (\exists P'\ .\ (g \vdash (nid,\ m,\ h) \rightarrow P') \wedge P' = Q')]\!]$
$\Longrightarrow nid \mid g \sim g'$

lemma *lockstep-strong-bisimilulation*:
assumes $g' = $ *replace-node nid node g*
assumes $g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)$
assumes $g' \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)$
shows $nid \mid g \sim g'$
using *assms*(*2*) *assms*(*3*) *stepDet strong-noop-bisimilar.simps* **by** *blast*

lemma *no-step-bisimulation*:
assumes $\forall m\ h\ nid'\ m'\ h'.\ \neg(g \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h'))$
assumes $\forall m\ h\ nid'\ m'\ h'.\ \neg(g' \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h'))$
shows $nid \mid g \sim g'$
using *assms*
by (*simp add*: *assms*(*1*) *assms*(*2*) *strong-noop-bisimilar.intros*)

end

1.2 Formedness Properties

theory *Form*
imports
  Semantics.IREval
begin

definition *wf-start* **where**
  wf-start g = (*0* ∈ *ids g* ∧
    is-StartNode (*kind g 0*))

definition *wf-closed* **where**
  wf-closed g =
    (∀ *n* ∈ *ids g* .
      inputs g n ⊆ *ids g* ∧
      succ g n ⊆ *ids g* ∧
      kind g n ≠ *NoNode*)

definition *wf-phis* **where**
  wf-phis g =
    (∀ *n* ∈ *ids g*.
      is-PhiNode (*kind g n*) ⟶
      length (*ir-values* (*kind g n*))
       = *length* (*ir-ends*
          (*kind g* (*ir-merge* (*kind g n*)))))

definition *wf-ends* **where**
  wf-ends g =
    (∀ *n* ∈ *ids g* .
      is-AbstractEndNode (*kind g n*) ⟶
      card (*usages g n*) > *0*)

fun *wf-graph* :: *IRGraph* ⇒ *bool* **where**
  wf-graph g = (*wf-start g* ∧ *wf-closed g* ∧ *wf-phis g* ∧ *wf-ends g*)

lemmas *wf-folds* =
  wf-graph.simps
  wf-start-def
  wf-closed-def
  wf-phis-def
  wf-ends-def

fun *wf-stamps* :: *IRGraph* ⇒ *bool* **where**
  wf-stamps g = (∀ *n* ∈ *ids g* .
    (∀ *v m* . (*g m* ⊢ (*kind g n*) ↦ *v*) ⟶ *valid-value* (*stamp g n*) *v*))

fun *wf-stamp* :: *IRGraph* ⇒ (*ID* ⇒ *Stamp*) ⇒ *bool* **where**

wf-stamp g s = (∀ *n* ∈ *ids g* .
(∀ *v m* . (*g m* ⊢ (*kind g n*) ↦ *v*) ⟶ *valid-value* (*s n*) *v*))

lemma *wf-empty*: *wf-graph start-end-graph*
unfolding *start-end-graph-def wf-folds* **by** *simp*

lemma *wf-eg2-sq*: *wf-graph eg2-sq*
unfolding *eg2-sq-def wf-folds* **by** *simp*

fun *wf-values* :: *IRGraph* ⇒ *bool* **where**
wf-values g = (∀ *n* ∈ *ids g* .
(∀ *v m* . (*g m* ⊢ *kind g n* ↦ *v*) ⟶ *wf-value v*))

lemma *wf-value-range*:
b > *1* ∧ *b* ∈ *int-bits-allowed* ⟶ {*v*. *wf-value* (*IntVal b v*)} = {*v*. ((−(*2*^(*b*−*1*)) ≤ *v*) ∧ (*v* < (*2*^(*b*−*1*))))}
unfolding *wf-value.simps*
by *auto*

lemma *wf-value-bit-range*:
b = *1* ⟶ {*v*. *wf-value* (*IntVal b v*)} = {}
unfolding *wf-value.simps*
by (*simp add*: *int-bits-allowed-def*)

end

1.3 Dynamic Frames

This theory defines two operators, 'unchanged' and 'changeonly', that are useful for specifying which nodes in an IRGraph can change. The dynamic framing idea originates from 'Dynamic Frames' in software verification, started by Ioannis T. Kassios in "Dynamic frames: Support for framing, dependencies and sharing without restrictions", In FM 2006.

theory *IRGraphFrames*
imports
Form
Semantics.IREval
begin

fun *unchanged* :: *ID set* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
unchanged ns g1 g2 = (∀ *n* . *n* ∈ *ns* ⟶
(*n* ∈ *ids g1* ∧ *n* ∈ *ids g2* ∧ *kind g1 n* = *kind g2 n*))

fun *changeonly* :: *ID set* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
changeonly ns g1 g2 = (∀ *n* . *n* ∈ *ids g1* ∧ *n* ∉ *ns* ⟶
(*n* ∈ *ids g1* ∧ *n* ∈ *ids g2* ∧ *kind g1 n* = *kind g2 n*))

lemma *node-unchanged*:
assumes *unchanged ns g1 g2*
assumes $nid \in ns$
shows $kind\ g1\ nid = kind\ g2\ nid$
using *assms* **by** *auto*

lemma *other-node-unchanged*:
assumes *changeonly ns g1 g2*
assumes $nid \in ids\ g1$
assumes $nid \notin ns$
shows $kind\ g1\ nid = kind\ g2\ nid$
using *assms*
using *changeonly.simps* **by** *blast*

Some notation for input nodes used

inductive *eval-uses*:: $IRGraph \Rightarrow ID \Rightarrow ID \Rightarrow bool$
for *g* **where**

use0: $nid \in ids\ g$
$\Longrightarrow$ *eval-uses g nid nid* |

use-inp: $nid' \in inputs\ g\ n$
$\Longrightarrow$ *eval-uses g nid nid'* |

use-trans: ⟦*eval-uses g nid nid'*;
eval-uses g nid' nid''⟧
$\Longrightarrow$ *eval-uses g nid nid''*

fun *eval-usages* :: $IRGraph \Rightarrow ID \Rightarrow ID\ set$ **where**
eval-usages g nid = $\{n \in ids\ g\ .\ eval\text{-}uses\ g\ nid\ n\}$

lemma *eval-usages-self*:
assumes $nid \in ids\ g$
shows $nid \in eval\text{-}usages\ g\ nid$
using *assms eval-usages.simps eval-uses.intros*(*1*)
by (*simp add*: *ids.rep-eq*)

lemma *not-in-g-inputs*:
assumes $nid \notin ids\ g$
shows $inputs\ g\ nid = \{\}$
proof −
have *k*: $kind\ g\ nid = NoNode$ **using** *assms not-in-g* **by** *blast*
then show *?thesis* **by** (*simp add*: *k*)
qed

lemma *child-member*:
assumes $n = kind\ g\ nid$

assumes $n \neq NoNode$
assumes *List.member* (*inputs-of n*) *child*
shows $child \in inputs\ g\ nid$
unfolding *inputs.simps* **using** *assms*
by (*metis in-set-member*)

lemma *child-member-in*:
assumes $nid \in ids\ g$
assumes *List.member* (*inputs-of* (*kind g nid*)) *child*
shows $child \in inputs\ g\ nid$
unfolding *inputs.simps* **using** *assms*
by (*metis child-member ids-some inputs.elims*)

lemma *inp-in-g*:
assumes $n \in inputs\ g\ nid$
shows $nid \in ids\ g$
proof –
have $inputs\ g\ nid \neq \{\}$
using *assms*
by (*metis empty-iff empty-set*)
then have $kind\ g\ nid \neq NoNode$
using *not-in-g-inputs*
using *ids-some* **by** *blast*
then show *?thesis*
using *not-in-g*
by *metis*
qed

lemma *inp-in-g-wf*:
assumes *wf-graph g*
assumes $n \in inputs\ g\ nid$
shows $n \in ids\ g$
using *assms* **unfolding** *wf-folds*
using *inp-in-g* **by** *blast*

lemma *kind-unchanged*:
assumes $nid \in ids\ g1$
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows $kind\ g1\ nid = kind\ g2\ nid$
proof –
show *?thesis*
using *assms eval-usages-self*
using *unchanged.simps* **by** *blast*
qed

lemma *child-unchanged*:

assumes *child* $\in$ *inputs g1 nid*
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows *unchanged* (*eval-usages g1 child*) *g1 g2*
by (*smt assms*(*1*) *assms*(*2*) *eval-usages.simps mem-Collect-eq*
unchanged.simps use-inp use-trans)

lemma *eval-usages*:
assumes *us* = *eval-usages g nid*
assumes *nid'* $\in$ *ids g*
shows *eval-uses g nid nid'* $\longleftrightarrow$ *nid'* $\in$ *us* (**is** *?P* $\longleftrightarrow$ *?Q*)
using *assms eval-usages.simps*
by (*simp add*: *ids.rep-eq*)

lemma *inputs-are-uses*:
assumes *nid'* $\in$ *inputs g nid*
shows *eval-uses g nid nid'*
by (*metis assms use-inp*)

lemma *inputs-are-usages*:
assumes *nid'* $\in$ *inputs g nid*
assumes *nid'* $\in$ *ids g*
shows *nid'* $\in$ *eval-usages g nid*
using *assms*(*1*) *assms*(*2*) *eval-usages inputs-are-uses* **by** *blast*

lemma *usage-includes-inputs*:
assumes *us* = *eval-usages g nid*
assumes *ls* = *inputs g nid*
assumes *ls* $\subseteq$ *ids g*
shows *ls* $\subseteq$ *us*
using *inputs-are-usages eval-usages*
using *assms*(*1*) *assms*(*2*) *assms*(*3*) **by** *blast*

lemma *elim-inp-set*:
assumes *k* = *kind g nid*
assumes *k* $\neq$ *NoNode*
assumes *child* $\in$ *set* (*inputs-of k*)
shows *child* $\in$ *inputs g nid*
using *assms* **by** *auto*

lemma *eval-in-ids*:
assumes *g m* $\vdash$ (*kind g nid*) $\mapsto$ *v*
shows *nid* $\in$ *ids g*
using *assms* **by** (*cases kind g nid* = *NoNode*; *auto*)

theorem *stay-same*:
assumes *nc*: *unchanged* (*eval-usages g1 nid*) *g1 g2*
assumes *g1*: *g1 m* $\vdash$ (*kind g1 nid*) $\mapsto$ *v1*
assumes *wf*: *wf-graph g1*

shows *g2 m* ⊢ (*kind g2 nid*) ↦ *v1*
proof –
have *nid*: *nid* ∈ *ids g1*
using *g1 eval-in-ids* **by** *simp*
then have *nid* ∈ *eval-usages g1 nid*
using *eval-usages-self* **by** *blast*
then have *kind-same*: *kind g1 nid* = *kind g2 nid*
using *nc node-unchanged* **by** *blast*
show *?thesis* **using** *g1 nid nc*
proof (*induct m* (*kind g1 nid*) *v1 arbitrary*: *nid rule*: *eval.induct*)
print-cases
case *const*: (*ConstantNode m c*)
then have (*kind g2 nid*) = *ConstantNode c*
using *kind-unchanged* **by** *metis*
then show *?case* **using** *eval.ConstantNode const.hyps*(*1*) **by** *metis*
next
case *param*: (*ParameterNode val m i*)
show *?case*
by (*metis eval.ParameterNode kind-unchanged param.hyps*(*1*) *param.prems*(*1*) *param.prems*(*2*))
next
case (*ValuePhiNode val nida ux uy*)
then have *kind*: (*kind g2 nid*) = *ValuePhiNode nida ux uy*
using *kind-unchanged* **by** *metis*
then show *?case*
using *eval.ValuePhiNode kind ValuePhiNode.hyps*(*1*) **by** *metis*
next
case (*ValueProxyNode m child val - nid*)
from *ValueProxyNode.prems*(*1*) *ValueProxyNode.hyps*(*3*)
have *inp-in*: *child* ∈ *inputs g1 nid*
using *child-member-in inputs-of-ValueProxyNode*
by (*metis member-rec*(*1*))
then have *cin*: *child* ∈ *ids g1*
using *wf inp-in-g-wf* **by** *blast*
from *inp-in* **have** *unc*: *unchanged* (*eval-usages g1 child*) *g1 g2*
using *child-unchanged ValueProxyNode.prems*(*2*) **by** *metis*
then have *g2 m* ⊢ (*kind g2 child*) ↦ *val*
using *ValueProxyNode.hyps*(*2*) *cin*
by *blast*
then show *?case*
by (*metis ValueProxyNode.hyps*(*3*) *ValueProxyNode.prems*(*1*) *ValueProxyNode.prems*(*2*) *eval.ValueProxyNode kind-unchanged*)
next
case (*AbsNode m x b v -*)
then have *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-unchanged elim-inp-set ids-some inputs-of.simps*(*1*) *list.set-intros*(*1*))
then have *g2 m* ⊢ (*kind g2 x*) ↦ *IntVal b v*
using *AbsNode.hyps*(*1*) *AbsNode.hyps*(*2*) *not-in-g*
by (*metis AbsNode.hyps*(*3*) *AbsNode.prems*(*1*) *elim-inp-set ids-some inp-in-g-wf*

inputs-of.simps(*1*) *list.set-intros*(*1*) *wf*)
then show *?case*
by (*metis AbsNode.hyps*(*3*) *AbsNode.prems*(*1*) *AbsNode.prems*(*2*) *eval.AbsNode kind-unchanged*)
next
case *Node*: (*NegateNode m x v -*)
from *inputs-of-NegateNode Node.hyps*(*3*) *Node.prems*(*1*)
have *xinp*: $x \in$ *inputs g1 nid*
using *child-member-in* **by** (*metis member-rec*(*1*))
then have *xin*: $x \in$ *ids g1*
using *wf inp-in-g-wf* **by** *blast*
from *xinp child-unchanged Node.prems*(*2*)
have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2* **by** *blast*
have *x1*:*g1 m* $\vdash$ (*kind g1 x*) $\mapsto$ *v*
using *Node.hyps*(*1*) *Node.hyps*(*2*)
by *blast*
have *x2*: *g2 m* $\vdash$ (*kind g2 x*) $\mapsto$ *v*
using *kind-unchanged ux xin Node.hyps*
by *blast*
then show *?case*
using *kind-same Node.hyps*(*1,3*) *eval.NegateNode*
by (*metis Node.prems*(*1*) *Node.prems*(*2*) *kind-unchanged ux xin*)
next
case *node*:(*AddNode m x v1 y v2*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-unchanged inputs.simps inputs-of-AddNode list.set-intros*(*1*))
then have *x*: *g1 m* $\vdash$ (*kind g1 x*) $\mapsto$ *v1*
using *node.hyps*(*1*) **by** *blast*
have *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis IRNodes.inputs-of-AddNode child-member-in child-unchanged member-rec*(*1*) *node.hyps*(*5*) *node.prems*(*1*) *node.prems*(*2*))
have *y*: *g1 m* $\vdash$ (*kind g1 y*) $\mapsto$ *v2*
using *node.hyps*(*3*) **by** *blast*
show *?case*
using *node.hyps node.prems ux x uy y*
by (*metis AddNode inputs.simps inp-in-g-wf inputs-of-AddNode kind-unchanged list.set-intros*(*1*) *set-subset-Cons subset-iff wf*)
next
case *node*:(*SubNode m x v1 y v2*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-SubNode member-rec*(*1*))
then have *x*: *g1 m* $\vdash$ (*kind g1 x*) $\mapsto$ *v1*
using *node.hyps*(*1*) **by** *blast*
from *node* **have** *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-SubNode member-rec*(*1*))
have *y*: *g1 m* $\vdash$ (*kind g1 y*) $\mapsto$ *v2*
using *node.hyps*(*3*) **by** *blast*
show *?case*
using *node.hyps node.prems ux x uy y*

```
    by (metis SubNode inputs.simps inputs-of-SubNode kind-unchanged list.set-intros(1)
set-subset-Cons subsetD wf wf-folds(1,3))
  next
    case node:(MulNode m x v1 y v2)
    then have ux: unchanged (eval-usages g1 x) g1 g2
     by (metis child-member-in child-unchanged inputs-of-MulNode member-rec(1))
    then have x: g1 m ⊢ (kind g1 x) ↦ v1
      using node.hyps(1) by blast
    from node have uy: unchanged (eval-usages g1 y) g1 g2
     by (metis child-member-in child-unchanged inputs-of-MulNode member-rec(1))
    have y: g1 m ⊢ (kind g1 y) ↦ v2
      using node.hyps(3) by blast
    show ?case
      using node.hyps node.prems ux x uy y
    by (metis MulNode inputs.simps inputs-of-MulNode kind-unchanged list.set-intros(1)
set-subset-Cons subsetD wf wf-folds(1,3))
  next
    case node:(AndNode m x v1 y v2)
    then have ux: unchanged (eval-usages g1 x) g1 g2
     by (metis child-member-in child-unchanged inputs-of-AndNode member-rec(1))
    then have x: g1 m ⊢ (kind g1 x) ↦ v1
      using node.hyps(1) by blast
    from node have uy: unchanged (eval-usages g1 y) g1 g2
     by (metis child-member-in child-unchanged inputs-of-AndNode member-rec(1))
    have y: g1 m ⊢ (kind g1 y) ↦ v2
      using node.hyps(3) by blast
    show ?case
      using node.hyps node.prems ux x uy y
    by (metis AndNode inputs.simps inputs-of-AndNode kind-unchanged list.set-intros(1)
set-subset-Cons subsetD wf wf-folds(1,3))
  next
    case node: (OrNode m x v1 y v2)
    then have ux: unchanged (eval-usages g1 x) g1 g2
      by (metis child-member-in child-unchanged inputs-of-OrNode member-rec(1))
    then have x: g1 m ⊢ (kind g1 x) ↦ v1
      using node.hyps(1) by blast
    from node have uy: unchanged (eval-usages g1 y) g1 g2
      by (metis child-member-in child-unchanged inputs-of-OrNode member-rec(1))
    have y: g1 m ⊢ (kind g1 y) ↦ v2
      using node.hyps(3) by blast
    show ?case
      using node.hyps node.prems ux x uy y
    by (metis OrNode inputs.simps inputs-of-OrNode kind-unchanged list.set-intros(1)
set-subset-Cons subsetD wf wf-folds(1,3))
  next
    case node: (XorNode m x v1 y v2)
    then have ux: unchanged (eval-usages g1 x) g1 g2
     by (metis child-member-in child-unchanged inputs-of-XorNode member-rec(1))
    then have x: g1 m ⊢ (kind g1 x) ↦ v1
```

using *node.hyps*(*1*) **by** *blast*
from *node* **have** *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-XorNode member-rec*(*1*))
have *y*: *g1 m* ⊢ (*kind g1 y*) ↦ *v2*
using *node.hyps*(*3*) **by** *blast*
show *?case*
using *node.hyps node.prems ux x uy y*
by (*metis XorNode inputs.simps inputs-of-XorNode kind-unchanged list.set-intros*(*1*) *set-subset-Cons subsetD wf wf-folds*(*1*,*3*))
next
case *node*: (*IntegerEqualsNode m x b v1 y v2 val*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-IntegerEqualsNode member-rec*(*1*))
then have *x*: *g1 m* ⊢ (*kind g1 x*) ↦ *IntVal b v1*
using *node.hyps*(*1*) **by** *blast*
from *node* **have** *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-IntegerEqualsNode member-rec*(*1*))
have *y*: *g1 m* ⊢ (*kind g1 y*) ↦ *IntVal b v2*
using *node.hyps*(*3*) **by** *blast*
show *?case*
using *node.hyps node.prems ux x uy y*
by (*metis* (*full-types*) *IntegerEqualsNode child-member-in in-set-member inputs-of-IntegerEqualsNode kind-unchanged list.set-intros*(*1*) *set-subset-Cons subsetD wf wf-folds*(*1*,*3*))
next
case *node*: (*IntegerLessThanNode m x b v1 y v2 val*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-IntegerLessThanNode member-rec*(*1*))
then have *x*: *g1 m* ⊢ (*kind g1 x*) ↦ *IntVal b v1*
using *node.hyps*(*1*) **by** *blast*
from *node* **have** *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-IntegerLessThanNode member-rec*(*1*))
have *y*: *g1 m* ⊢ (*kind g1 y*) ↦ *IntVal b v2*
using *node.hyps*(*3*) **by** *blast*
show *?case*
using *node.hyps node.prems ux x uy y*
by (*metis* (*full-types*) *IntegerLessThanNode child-member-in in-set-member inputs-of-IntegerLessThanNode kind-unchanged list.set-intros*(*1*) *set-subset-Cons subsetD wf wf-folds*(*1*,*3*))
next
case *node*: (*ShortCircuitOrNode m x b v1 y v2 val*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-ShortCircuitOrNode member-rec*(*1*))
then have *x*: *g1 m* ⊢ (*kind g1 x*) ↦ *IntVal b v1*

using *node.hyps*(*1*) **by** *blast*
from *node* **have** *uy*: *unchanged* (*eval-usages g1 y*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-ShortCircuitOrNode member-rec*(*1*))
have *y*: *g1 m* ⊢ (*kind g1 y*) ↦ *IntVal b v2*
using *node.hyps*(*3*) **by** *blast*
have *x2*: *g2 m* ⊢ (*kind g2 x*) ↦ *IntVal b v1*
by (*metis inputs.simps inputs-of-ShortCircuitOrNode list.set-intros*(*1*) *node.hyps*(*2*) *node.hyps*(*6*) *node.prems*(*1*) *subsetD ux wf wf-folds*(*1*,*3*))
have *y2*: *g2 m* ⊢ (*kind g2 y*) ↦ *IntVal b v2*
by (*metis basic-trans-rules*(*31*) *inputs.simps inputs-of-ShortCircuitOrNode list.set-intros*(*1*) *node.hyps*(*4*) *node.hyps*(*6*) *node.prems*(*1*) *set-subset-Cons uy wf wf-folds*(*1*,*3*))
show *?case*
using *node.hyps node.prems ux x uy y x2 y2*
by (*metis ShortCircuitOrNode kind-unchanged*)
next
case *node*: (*LogicNegationNode m x v1 val nida*)
then have *ux*: *unchanged* (*eval-usages g1 x*) *g1 g2*
by (*metis child-member-in child-unchanged inputs-of-LogicNegationNode member-rec*(*1*))
then have *x*:*g2 m* ⊢ (*kind g2 x*) ↦ *IntVal 1 v1*
by (*metis inputs.simps inp-in-g-wf inputs-of-LogicNegationNode list.set-intros*(*1*) *node.hyps*(*2*) *node.hyps*(*4*) *wf*)
then show *?case*
by (*metis LogicNegationNode kind-unchanged node.hyps*(*3*) *node.hyps*(*4*) *node.prems*(*1*) *node.prems*(*2*))
next
case *node*:(*ConditionalNode m condition cond trueExp b trueVal falseExp falseVal val*)
have *c*: *condition* ∈ *inputs g1 nid*
by (*metis IRNodes.inputs-of-ConditionalNode child-member-in member-rec*(*1*) *node.hyps*(*8*) *node.prems*(*1*))
then have *unchanged* (*eval-usages g1 condition*) *g1 g2*
using *child-unchanged node.prems*(*2*) **by** *blast*
then have *cond*: *g2 m* ⊢ (*kind g2 condition*) ↦ *IntVal 1 cond*
using *node c inp-in-g-wf wf* **by** *blast*

have *t*: *trueExp* ∈ *inputs g1 nid*
by (*metis IRNodes.inputs-of-ConditionalNode child-member-in member-rec*(*1*) *node.hyps*(*8*) *node.prems*(*1*))
then have *utrue*: *unchanged* (*eval-usages g1 trueExp*) *g1 g2*
using *node.prems*(*2*) *child-unchanged* **by** *blast*
then have *trueVal*: *g2 m* ⊢ (*kind g2 trueExp*) ↦ *IntVal b* (*trueVal*)
using *node.hyps node t inp-in-g-wf wf* **by** *blast*

have *f*: *falseExp* ∈ *inputs g1 nid*
by (*metis IRNodes.inputs-of-ConditionalNode child-member-in member-rec*(*1*) *node.hyps*(*8*) *node.prems*(*1*))

then have *ufalse*: *unchanged* (*eval-usages g1 falseExp*) *g1 g2*
using *node.prems*(*2*) *child-unchanged* **by** *blast*
then have *falseVal*: *g2 m* ⊢ (*kind g2 falseExp*) ↦ *IntVal b* (*falseVal*)
using *node.hyps node f inp-in-g-wf wf* **by** *blast*

have *g2 m* ⊢ (*kind g2 nid*) ↦ *val*
using *kind-same trueVal falseVal cond*
by (*metis ConditionalNode kind-unchanged node.hyps*(*7*) *node.hyps*(*8*) *node.prems*(*1*) *node.prems*(*2*))
then show *?case*
by *blast*

next
case (*RefNode m x val nid*)
have *x*: *x* ∈ *inputs g1 nid*
by (*metis IRNodes.inputs-of-RefNode RefNode.hyps*(*3*) *RefNode.prems*(*1*) *child-member-in member-rec*(*1*))
then have *ref*: *g2 m* ⊢ (*kind g2 x*) ↦ *val*
using *RefNode.hyps*(*2*) *RefNode.prems*(*2*) *child-unchanged inp-in-g-wf wf* **by** *blast*
then show *?case*
by (*metis RefNode.hyps*(*3*) *RefNode.prems*(*1*) *RefNode.prems*(*2*) *eval.RefNode kind-unchanged*)
next
case (*InvokeNodeEval val m - callTarget classInit stateDuring stateAfter nex*)
then show *?case*
by (*metis eval.InvokeNodeEval kind-unchanged*)
next
case (*SignedDivNode m x v1 y v2 zeroCheck frameState nex*)
then show *?case*
by (*metis eval.SignedDivNode kind-unchanged*)
next
case (*SignedRemNode m x v1 y v2 zeroCheck frameState nex*)
then show *?case*
by (*metis eval.SignedRemNode kind-unchanged*)
next
case (*InvokeWithExceptionNodeEval val m - callTarget classInit stateDuring stateAfter nex exceptionEdge*)
then show *?case*
by (*metis eval.InvokeWithExceptionNodeEval kind-unchanged*)
next
case (*NewInstanceNode m nid clazz stateBefore nex*)
then show *?case*
by (*metis eval.NewInstanceNode kind-unchanged*)
next
case (*IsNullNode m obj ref val*)
have *obj*: *obj* ∈ *inputs g1 nid*
by (*metis IRNodes.inputs-of-IsNullNode IsNullNode.hyps*(*4*) *inputs.simps list.set-intros*(*1*))

then have *ref*: *g2 m* ⊢ (*kind g2 obj*) ↦ *ObjRef ref*
using *IsNullNode.hyps*(*1*) *IsNullNode.hyps*(*2*) *IsNullNode.prems*(*2*) *child-unchanged eval-in-ids* **by** *blast*
then show *?case*
by (*metis* (*full-types*) *IsNullNode.hyps*(*3*) *IsNullNode.hyps*(*4*) *IsNullNode.prems*(*1*) *IsNullNode.prems*(*2*) *eval.IsNullNode kind-unchanged*)
next
case (*LoadFieldNode*)
then show *?case*
by (*metis eval.LoadFieldNode kind-unchanged*)
next
case (*PiNode m object val*)
have *object*: *object* ∈ *inputs g1 nid*
using *inputs-of-PiNode inputs.simps*
by (*metis PiNode.hyps*(*3*) *append-Cons list.set-intros*(*1*))
then have *ref*: *g2 m* ⊢ (*kind g2 object*) ↦ *val*
using *PiNode.hyps*(*1*) *PiNode.hyps*(*2*) *PiNode.prems*(*2*) *child-unchanged eval-in-ids* **by** *blast*
then show *?case*
by (*metis PiNode.hyps*(*3*) *PiNode.prems*(*1*) *PiNode.prems*(*2*) *eval.PiNode kind-unchanged*)
next
case (*NotNode m x val not-val*)
have *object*: *x* ∈ *inputs g1 nid*
using *inputs-of-NotNode inputs.simps*
by (*metis NotNode.hyps*(*4*) *list.set-intros*(*1*))
then have *ref*: *g2 m* ⊢ (*kind g2 x*) ↦ *val*
using *NotNode.hyps*(*1*) *NotNode.hyps*(*2*) *NotNode.prems*(*2*) *child-unchanged eval-in-ids* **by** *blast*
then show *?case*
by (*metis NotNode.hyps*(*3*) *NotNode.hyps*(*4*) *NotNode.prems*(*1*) *NotNode.prems*(*2*) *eval.NotNode kind-unchanged*)
qed
qed

lemma *add-changed*:
assumes *gup* = *add-node new k g*
shows *changeonly* {*new*} *g gup*
using *assms* **unfolding** *add-node-def changeonly.simps*
using *add-node.rep-eq add-node-def kind.rep-eq* **by** *auto*

lemma *disjoint-change*:
assumes *changeonly change g gup*
assumes *nochange* = *ids g* − *change*
shows *unchanged nochange g gup*
using *assms* **unfolding** *changeonly.simps unchanged.simps*
by *blast*

lemma *add-node-unchanged*:
  assumes $new \notin ids\ g$
  assumes $nid \in ids\ g$
  assumes $gup = add\text{-}node\ new\ k\ g$
  assumes *wf-graph g*
  shows *unchanged* (*eval-usages g nid*) *g gup*
proof −
  have $new \notin (eval\text{-}usages\ g\ nid)$ **using** *assms*
    using *eval-usages.simps* **by** *blast*
  then have *changeonly* $\{new\}$ *g gup*
    using *assms add-changed* **by** *blast*
  then show *?thesis* **using** *assms add-node-def disjoint-change*
    using *Diff-insert-absorb* **by** *auto*
qed

lemma *eval-uses-imp*:
  $((nid' \in ids\ g \wedge nid = nid')$
    $\vee\ nid' \in inputs\ g\ nid$
    $\vee\ (\exists\, nid''\ .\ eval\text{-}uses\ g\ nid\ nid'' \wedge eval\text{-}uses\ g\ nid''\ nid'))$
    $\longleftrightarrow eval\text{-}uses\ g\ nid\ nid'$
  using *use0 use-inp use-trans*
  by (*meson eval-uses.simps*)

lemma *wf-use-ids*:
  assumes *wf-graph g*
  assumes $nid \in ids\ g$
  assumes $eval\text{-}uses\ g\ nid\ nid'$
  shows $nid' \in ids\ g$
  using *assms*(*3*)
proof (*induction rule*: *eval-uses.induct*)
  case *use0*
  then show *?case* **by** *simp*
next
  case *use-inp*
  then show *?case*
    using *assms*(*1*) *inp-in-g-wf* **by** *blast*
next
  case *use-trans*
  then show *?case* **by** *blast*
qed

lemma *no-external-use*:
  assumes *wf-graph g*
  assumes $nid' \notin ids\ g$
  assumes $nid \in ids\ g$
  shows $\neg(eval\text{-}uses\ g\ nid\ nid')$
proof −
  have *0*: $nid \neq nid'$
    using *assms* **by** *blast*

have *inp*: $nid' \notin$ *inputs g nid*
  using *assms*
  using *inp-in-g-wf* **by** *blast*
have *rec-0*: $\nexists n\ .\ n \in$ *ids g* $\wedge$ $n = nid'$
  using *assms* **by** *blast*
have *rec-inp*: $\nexists n\ .\ n \in$ *ids g* $\wedge$ $n \in$ *inputs g* nid'
  using *assms*(*2*) *inp-in-g* **by** *blast*
have *rec*: $\nexists nid''$. *eval-uses g nid* nid'' $\wedge$ *eval-uses g* nid'' nid'
  using *wf-use-ids assms*(*1*) *assms*(*2*) *assms*(*3*) **by** *blast*
from *inp 0 rec* **show** *?thesis*
  using *eval-uses-imp* **by** *blast*
qed

end

1.4 Graph Rewriting

theory
  Rewrites
imports
  IRGraphFrames
  Stuttering
begin

fun *replace-usages* :: *ID* $\Rightarrow$ *ID* $\Rightarrow$ *IRGraph* $\Rightarrow$ *IRGraph* **where**
  replace-usages nid nid' *g* = *replace-node nid* (*RefNode* nid', *stamp g* nid') *g*

lemma *replace-usages-effect*:
  assumes g' = *replace-usages nid* nid' *g*
  shows *kind* g' *nid* = *RefNode* nid'
  using *assms replace-node-lookup replace-usages.simps IRNode.distinct*(*2069*)
  by (*metis*)

lemma *replace-usages-changeonly*:
  assumes *nid* $\in$ *ids g*
  assumes g' = *replace-usages nid* nid' *g*
  shows *changeonly* {*nid*} *g* g'
  using *assms* **unfolding** *replace-usages.simps*
  by (*metis DiffI changeonly.elims*(*3*) *ids-some replace-node-unchanged*)

lemma *replace-usages-unchanged*:
  assumes *nid* $\in$ *ids g*
  assumes g' = *replace-usages nid* nid' *g*
  shows *unchanged* (*ids g* $-$ {*nid*}) *g* g'
  using *assms* **unfolding** *replace-usages.simps*
  by (*smt* (*verit, del-insts*) *DiffE ids-some replace-node-unchanged unchanged.simps*)

```
fun nextNid :: IRGraph ⇒ ID where
  nextNid g = (Max (ids g)) + 1

lemma max-plus-one:
  fixes c :: ID set
  shows ⟦finite c; c ≠ {}⟧ ⟹ (Max c) + 1 ∉ c
  by (meson Max-gr-iff less-add-one less-irrefl)

lemma ids-finite:
  finite (ids g)
  by simp

lemma nextNidNotIn:
  ids g ≠ {} ⟶ nextNid g ∉ ids g
  unfolding nextNid.simps
  using ids-finite max-plus-one by blast

fun constantCondition :: bool ⇒ ID ⇒ IRNode ⇒ IRGraph ⇒ IRGraph where
  constantCondition val nid (IfNode cond t f) g =
    replace-node nid (IfNode (nextNid g) t f, stamp g nid)
      (add-node (nextNid g) ((ConstantNode (bool-to-val val)), default-stamp) g) |
  constantCondition cond nid - g = g

lemma constantConditionTrue:
  assumes kind g ifcond = IfNode cond t f
  assumes g' = constantCondition True ifcond (kind g ifcond) g
  shows g' ⊢ (ifcond, m, h) → (t, m, h)
proof -
  have if': kind g' ifcond = IfNode (nextNid g) t f
    by (metis IRNode.simps(989) assms(1) assms(2) constantCondition.simps(1)
replace-node-lookup)
  have bool-to-val True = (IntVal 1 1)
    by auto
  have ifcond ≠ (nextNid g)
    by (metis IRNode.simps(989) assms(1) emptyE ids-some nextNidNotIn)
  then have c': kind g' (nextNid g) = ConstantNode (IntVal 1 1)
    using assms(2) replace-node-unchanged
   by (metis DiffI IRNode.distinct(585) ‹bool-to-val True = IntVal 1 1› add-node-lookup
assms(1) constantCondition.simps(1) emptyE insertE not-in-g)
  from if' c' show ?thesis using IfNode
    by (smt (z3) ConstantNode val-to-bool.simps(1))
qed

lemma constantConditionFalse:
  assumes kind g ifcond = IfNode cond t f
  assumes g' = constantCondition False ifcond (kind g ifcond) g
  shows g' ⊢ (ifcond, m, h) → (f, m, h)
proof -
  have if': kind g' ifcond = IfNode (nextNid g) t f
```

by (*metis IRNode.simps*(*989*) *assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *replace-node-lookup*)
have *bool-to-val False* = (*IntVal 1 0*)
by *auto*
have *ifcond* ≠ (*nextNid g*)
by (*metis IRNode.simps*(*989*) *assms*(*1*) *emptyE ids-some nextNidNotIn*)
then have *c'*: *kind g'* (*nextNid g*) = *ConstantNode* (*IntVal 1 0*)
using *assms*(*2*) *replace-node-unchanged*
by (*metis DiffI IRNode.distinct*(*585*) ‹*bool-to-val False* = *IntVal 1 0*› *add-node-lookup assms*(*1*) *constantCondition.simps*(*1*) *emptyE insertE not-in-g*)
from *if' c'* **show** *?thesis* **using** *IfNode*
by (*smt* (*z3*) *ConstantNode val-to-bool.simps*(*1*))
qed

lemma *diff-forall*:
assumes $\forall n \in ids\ g - \{nid\}.\ cond\ n$
shows $\forall n.\ n \in ids\ g \wedge n \notin \{nid\} \longrightarrow cond\ n$
by (*meson Diff-iff assms*)

lemma *replace-node-changeonly*:
assumes *g'* = *replace-node nid node g*
shows *changeonly* {*nid*} *g g'*
using *assms replace-node-unchanged*
unfolding *changeonly.simps* **using** *diff-forall*
sorry

lemma *add-node-changeonly*:
assumes *g'* = *add-node nid node g*
shows *changeonly* {*nid*} *g g'*
by (*metis Rep-IRGraph-inverse add-node.rep-eq assms replace-node.rep-eq replace-node-changeonly*)

lemma *constantConditionNoEffect*:
assumes ¬(*is-IfNode* (*kind g nid*))
shows *g* = *constantCondition b nid* (*kind g nid*) *g*
using *assms* **apply** (*cases kind g nid*)
using *constantCondition.simps*
apply *presburger+*
apply (*metis is-IfNode-def*)
using *constantCondition.simps*
by *presburger+*

lemma *constantConditionIfNode*:
assumes *kind g nid* = *IfNode cond t f*
shows *constantCondition val nid* (*kind g nid*) *g* =
replace-node nid (*IfNode* (*nextNid g*) *t f*, *stamp g nid*)
(*add-node* (*nextNid g*) ((*ConstantNode* (*bool-to-val val*)), *default-stamp*) *g*)
using *constantCondition.simps*
by (*simp add*: *assms*)

lemma *constantCondition-changeonly*:
  assumes $nid \in ids\ g$
  assumes $g' = constantCondition\ b\ nid\ (kind\ g\ nid)\ g$
  shows $changeonly\ \{nid\}\ g\ g'$
proof (*cases is-IfNode* (*kind g nid*))
  case *True*
  have $nextNid\ g \notin ids\ g$
    using *nextNidNotIn* **by** (*metis emptyE*)
  then show *?thesis* **using** *assms*
   using *replace-node-changeonly add-node-changeonly* **unfolding** *changeonly.simps*
    using *True constantCondition.simps*(*1*) *is-IfNode-def*
    by (*metis* (*full-types*) *DiffD2 Diff-insert-absorb*)
next
  case *False*
  have $g = g'$
    using *constantConditionNoEffect*
    using *False assms*(*2*) **by** *blast*
  then show *?thesis* **by** *simp*
qed

lemma *constantConditionNoIf*:
  assumes $\forall\, cond\ t\ f.\ kind\ g\ ifcond \neq IfNode\ cond\ t\ f$
  assumes $g' = constantCondition\ val\ ifcond\ (kind\ g\ ifcond)\ g$
  shows $\exists\, nid'\ .(g\ m\ h \vdash ifcond \leadsto nid') \longleftrightarrow (g'\ m\ h \vdash ifcond \leadsto nid')$
proof −
  have $g' = g$
    using *assms*(*2*) *assms*(*1*)
    using *constantConditionNoEffect*
    by (*metis IRNode.collapse*(*11*))
  then show *?thesis* **by** *simp*
qed

lemma *constantConditionValid*:
  assumes $kind\ g\ ifcond = IfNode\ cond\ t\ f$
  assumes $g\ m \vdash kind\ g\ cond \mapsto v$
  assumes $const = val\text{-}to\text{-}bool\ v$
  assumes $g' = constantCondition\ const\ ifcond\ (kind\ g\ ifcond)\ g$
  shows $\exists\, nid'\ .(g\ m\ h \vdash ifcond \leadsto nid') \longleftrightarrow (g'\ m\ h \vdash ifcond \leadsto nid')$
proof (*cases const*)
  case *True*
  have *ifstep*: $g \vdash (ifcond,\ m,\ h) \rightarrow (t,\ m,\ h)$
    by (*meson IfNode True assms*(*1*) *assms*(*2*) *assms*(*3*))
  have *ifstep'*: $g' \vdash (ifcond,\ m,\ h) \rightarrow (t,\ m,\ h)$
    using *constantConditionTrue*
    using *True assms*(*1*) *assms*(*4*) **by** *presburger*
  from *ifstep ifstep'* **show** *?thesis*
    using *StutterStep* **by** *blast*

next
  case *False*
  have *ifstep*: $g \vdash (ifcond,\ m,\ h) \rightarrow (f,\ m,\ h)$
    by (*meson IfNode False assms*(*1*) *assms*(*2*) *assms*(*3*))
  have *ifstep'*: $g' \vdash (ifcond,\ m,\ h) \rightarrow (f,\ m,\ h)$
    using *constantConditionFalse*
    using *False assms*(*1*) *assms*(*4*) **by** *presburger*
  from *ifstep ifstep'* **show** *?thesis*
    using *StutterStep* **by** *blast*
qed

end