Unspecified Veriopt Theory

July 3, 2021

Contents

1 Data-flow Semantics **1**

2 Control-flow Semantics **7**
- 2.1 Heap 7
- 2.2 Intraprocedural Semantics 7
- 2.3 Interprocedural Semantics 14
- 2.4 Big-step Execution 15
  - 2.4.1 Heap Testing 16

1 Data-flow Semantics

theory *IREval*
  imports
    Graph.IRGraph
begin

We define the semantics of data-flow nodes as big-step operational semantics. Data-flow nodes are evaluated in the context of the *IRGraph* and a method state (currently called MapState in the theories for historical reasons).

The method state consists of the values for each method parameter, references to method parameters use an index of the parameter wihtin the parameter list, as such we store a list of parameter values which are looked up at parameter references.

The method state also stores a mapping of node ids to values. The contents of this mapping is calculates during the traversal of the control flow graph.

As a concrete example, as the *SignedDivNode* can have side-effects (during division by zero), it is treated part of the control-flow as the data-flow is specified to be side-effect free. As a result, the control-flow semantics for *SignedDivNode* calculates the value of a node and maps the node identifier to the value within the method state. The data-flow semantics then just reads the value stored in the method state for the node.

type-synonym *MapState* = *ID* ⇒ *Value*
type-synonym *Params* = *Value list*

definition *new-map-state* :: *MapState* **where**
  new-map-state = (λ*x*. *UndefVal*)

fun *find-index* :: ′*a* ⇒ ′*a list* ⇒ *nat* **where**
  find-index - [] = *0* |
  find-index v (*x* # *xs*) = (*if* (*x*=*v*) *then 0 else find-index v xs* + *1*)

fun *phi-list* :: *IRGraph* ⇒ *ID* ⇒ *ID list* **where**
  phi-list g nid =
    (*filter* (λ*x*.(*is-PhiNode* (*kind g x*)))
      (*sorted-list-of-set* (*usages g nid*)))

fun *input-index* :: *IRGraph* ⇒ *ID* ⇒ *ID* ⇒ *nat* **where**
  input-index g n n′ = *find-index n*′ (*inputs-of* (*kind g n*))

fun *phi-inputs* :: *IRGraph* ⇒ *nat* ⇒ *ID list* ⇒ *ID list* **where**
  phi-inputs g i nodes = (*map* (λ*n*. (*inputs-of* (*kind g n*))!(*i* + *1*)) *nodes*)

fun *set-phis* :: *ID list* ⇒ *Value list* ⇒ *MapState* ⇒ *MapState* **where**
  set-phis [] [] *m* = *m* |
  set-phis (*nid* # *xs*) (*v* # *vs*) *m* = (*set-phis xs vs* (*m*(*nid* := *v*))) |
  set-phis [] (*v* # *vs*) *m* = *m* |
  set-phis (*x* # *xs*) [] *m* = *m*

inductive
  eval :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *IRNode* ⇒ *Value* ⇒ *bool* ([-, -, -] ⊢
- ↦ - *55*)
  for *g m p* **where**

  ConstantNode:
  [*g*, *m*, *p*] ⊢ (*ConstantNode c*) ↦ *c* |

  ParameterNode:
  [*g*, *m*, *p*] ⊢ (*ParameterNode i*) ↦ *p*!*i* |

  ValuePhiNode:
  [*g*, *m*, *p*] ⊢ (*ValuePhiNode nid* - -) ↦ *m nid* |

  ValueProxyNode:
  ⟦[*g*, *m*, *p*] ⊢ (*kind g c*) ↦ *val*⟧
    ⟹ [*g*, *m*, *p*] ⊢ (*ValueProxyNode c* -) ↦ *val* |

  — Unary arithmetic operators

AbsNode:
$[[g, m, p] \vdash (kind\ g\ x) \mapsto IntVal32\ v]\!]$
$\Longrightarrow [g, m, p] \vdash (AbsNode\ x) \mapsto if\ v < 0\ then\ (intval\text{-}sub\ (IntVal32\ 0)\ (IntVal32\ v))\ else\ (IntVal32\ v)\ |$

NegateNode:
$[[g, m, p] \vdash (kind\ g\ x) \mapsto v]\!]$
$\Longrightarrow [g, m, p] \vdash (NegateNode\ x) \mapsto (IntVal32\ 0) - v\ |$

NotNode:
$[[g, m, p] \vdash (kind\ g\ x) \mapsto v;$
$nv = intval\text{-}not\ v]\!]$
$\Longrightarrow [g, m, p] \vdash (NotNode\ x) \mapsto nv\ |$

— Binary arithmetic operators

AddNode:
$[[g, m, p] \vdash (kind\ g\ x) \mapsto v1;$
$[g, m, p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g, m, p] \vdash (AddNode\ x\ y) \mapsto v1 + v2\ |$

SubNode:
$[[g, m, p] \vdash (kind\ g\ x) \mapsto v1;$
$[g, m, p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g, m, p] \vdash (SubNode\ x\ y) \mapsto v1 - v2\ |$

MulNode:
$[[g, m, p] \vdash (kind\ g\ x) \mapsto v1;$
$[g, m, p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g, m, p] \vdash (MulNode\ x\ y) \mapsto v1 * v2\ |$

SignedDivNode:
$[g, m, p] \vdash (SignedDivNode\ nid\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-}) \mapsto m\ nid\ |$

SignedRemNode:
$[g, m, p] \vdash (SignedRemNode\ nid\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-}) \mapsto m\ nid\ |$

— Binary logical bitwise operators

AndNode:
$[[g, m, p] \vdash (kind\ g\ x) \mapsto v1;$
$[g, m, p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g, m, p] \vdash (AndNode\ x\ y) \mapsto intval\text{-}and\ \ v1\ v2\ |$

OrNode:
$[[g, m, p] \vdash (kind\ g\ x) \mapsto v1;$
$[g, m, p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g, m, p] \vdash (OrNode\ x\ y) \mapsto intval\text{-}or\ v1\ v2\ |$

XorNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto v1;$
$[g,\ m,\ p] \vdash (kind\ g\ y) \mapsto v2]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (XorNode\ x\ y) \mapsto intval\text{-}xor\ v1\ v2\ |$

— Comparison operators

IntegerEqualsNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto IntVal32\ v1;$
$[g,\ m,\ p] \vdash (kind\ g\ y) \mapsto IntVal32\ v2;$
$val = bool\text{-}to\text{-}val(v1 = v2)]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (IntegerEqualsNode\ x\ y) \mapsto val\ |$

IntegerLessThanNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto IntVal32\ v1;$
$[g,\ m,\ p] \vdash (kind\ g\ y) \mapsto IntVal32\ v2;$
$val = bool\text{-}to\text{-}val(v1 < v2)]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (IntegerLessThanNode\ x\ y) \mapsto val\ |$

IsNullNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ obj) \mapsto ObjRef\ ref;$
$val = bool\text{-}to\text{-}val(ref = None)]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (IsNullNode\ obj) \mapsto val\ |$

— Other nodes

ConditionalNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ condition) \mapsto IntVal32\ cond;$
$[g,\ m,\ p] \vdash (kind\ g\ trueExp) \mapsto IntVal32\ trueVal;$
$[g,\ m,\ p] \vdash (kind\ g\ falseExp) \mapsto IntVal32\ falseVal;$
$val = IntVal32\ (if\ (val\text{-}to\text{-}bool\ (IntVal32\ cond))\ then\ trueVal\ else\ falseVal)]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (ConditionalNode\ condition\ trueExp\ falseExp) \mapsto val\ |$

ShortCircuitOrNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto IntVal32\ v1;$
$[g,\ m,\ p] \vdash (kind\ g\ y) \mapsto IntVal32\ v2;$
$val = IntVal32\ (if\ v1 \neq 0\ then\ v1\ else\ v2)]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (ShortCircuitOrNode\ x\ y) \mapsto val\ |$

LogicNegationNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto IntVal32\ v1;$
$neg\text{-}v1 = (\neg(val\text{-}to\text{-}bool\ (IntVal32\ v1)));$
$val = bool\text{-}to\text{-}val\ neg\text{-}v1]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (LogicNegationNode\ x) \mapsto val\ |$

InvokeNodeEval:
$[g,\ m,\ p] \vdash (InvokeNode\ nid\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-}) \mapsto m\ nid$ |

InvokeWithExceptionNodeEval:
$[g,\ m,\ p] \vdash (InvokeWithExceptionNode\ nid\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-}) \mapsto m\ nid$ |

NewInstanceNode:
$[g,\ m,\ p] \vdash (NewInstanceNode\ nid\ \text{-}\ \text{-}\ \text{-}) \mapsto m\ nid$ |

LoadFieldNode:
$[g,\ m,\ p] \vdash (LoadFieldNode\ nid\ \text{-}\ \text{-}\ \text{-}) \mapsto m\ nid$ |

PiNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ object) \mapsto val]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (PiNode\ object\ guard) \mapsto val$ |

RefNode:
$[\![[g,\ m,\ p] \vdash (kind\ g\ x) \mapsto val]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (RefNode\ x) \mapsto val$

code-pred $(modes\colon i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool\ as\ evalE)\ eval$ **.**

The step semantics for phi nodes requires all the input nodes of the phi node to be evaluated to a value at the same time.

We introduce the *eval-all* relation to handle the evaluation of a list of node identifiers in parallel. As the evaluation semantics are side-effect free this is trivial.

inductive
eval-all $::$ $IRGraph \Rightarrow MapState \Rightarrow Params \Rightarrow ID\ list \Rightarrow Value\ list \Rightarrow bool$
$([\text{-},\ \text{-},\ \text{-}] \vdash \text{-} \longmapsto \text{-}\ 55)$
for $g\ m\ p$ **where**
Base:
$[g,\ m,\ p] \vdash [] \longmapsto []$ |

Transitive:
$[\![[g,\ m,\ p] \vdash (kind\ g\ nid) \mapsto v;$
$[g,\ m,\ p] \vdash xs \longmapsto vs]\!]$
$\Longrightarrow [g,\ m,\ p] \vdash (nid\ \#\ xs) \longmapsto (v\ \#\ vs)$

code-pred $(modes\colon i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool\ as\ eval\text{-}allE)\ eval\text{-}all$ **.**

inductive *eval-graph* $::$ $IRGraph \Rightarrow ID \Rightarrow Value\ list \Rightarrow Value \Rightarrow bool$
where
$[\![[g,\ new\text{-}map\text{-}state,\ ps] \vdash (kind\ g\ nid) \mapsto val]\!]$
$\Longrightarrow eval\text{-}graph\ g\ nid\ ps\ val$

code-pred ($modes$: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) $eval\text{-}graph$ **.**

values $\{v.\ eval\text{-}graph\ eg2\text{-}sq\ 4\ [IntVal32\ 5]\ v\}$

fun $has\text{-}control\text{-}flow :: IRNode \Rightarrow bool$ **where**
$has\text{-}control\text{-}flow\ n = (is\text{-}AbstractEndNode\ n$
$\vee (length\ (successors\text{-}of\ n) > 0))$

definition $control\text{-}nodes :: IRNode\ set$ **where**
$control\text{-}nodes = \{n\ .\ has\text{-}control\text{-}flow\ n\}$

fun $is\text{-}floating\text{-}node :: IRNode \Rightarrow bool$ **where**
$is\text{-}floating\text{-}node\ n = (\neg(has\text{-}control\text{-}flow\ n))$

definition $floating\text{-}nodes :: IRNode\ set$ **where**
$floating\text{-}nodes = \{n\ .\ is\text{-}floating\text{-}node\ n\}$

lemma $is\text{-}floating\text{-}node\ n \longleftrightarrow \neg(has\text{-}control\text{-}flow\ n)$
by $simp$

lemma $n \in control\text{-}nodes \longleftrightarrow n \notin floating\text{-}nodes$
by ($simp\ add$: $control\text{-}nodes\text{-}def\ floating\text{-}nodes\text{-}def$)

Here we show that using the elimination rules for eval we can prove 'inverted rule' properties

lemma $evalAddNode : [g,\ m,\ p] \vdash (AddNode\ x\ y) \mapsto val \Longrightarrow$
$(\exists\ v1.\ ([g,\ m,\ p] \vdash (kind\ g\ x) \mapsto v1) \wedge$
$(\exists\ v2.\ ([g,\ m,\ p] \vdash (kind\ g\ y) \mapsto v2) \wedge$
$val = intval\text{-}add\ v1\ v2))$
using $AddNodeE\ plus\text{-}Value\text{-}def$ **by** $metis$

lemma $not\text{-}floating$: $(\exists\, y\ ys.\ (successors\text{-}of\ n) = y\ \#\ ys) \longrightarrow \neg(is\text{-}floating\text{-}node\ n)$
unfolding $is\text{-}floating\text{-}node.simps$
by ($induct\ n$; $simp\ add$: $neq\text{-}Nil\text{-}conv$)

We show that within the context of a graph and method state, the same node will always evaluate to the same value and the semantics is therefore deterministic.

theorem $evalDet$:
$([g,\ m,\ p] \vdash node \mapsto val1) \Longrightarrow$
$(\forall\ val2.\ (([g,\ m,\ p] \vdash node \mapsto val2) \longrightarrow val1 = val2))$
apply ($induction\ rule$: $eval.induct$)
by ($rule\ allI$; $rule\ impI$; $elim\ EvalE$; $auto$)+

theorem $evalAllDet$:
$([g,\ m,\ p] \vdash nodes \longmapsto vals1) \Longrightarrow$

($\forall$ *vals2*. (([g, m, p] $\vdash$ *nodes* $\longmapsto$ *vals2*) $\longrightarrow$ *vals1* = *vals2*))
apply (*induction rule*: *eval-all.induct*)
using *eval-all.cases* **apply** *blast*
by (*metis evalDet eval-all.cases list.discI list.inject*)

end

2 Control-flow Semantics

theory *IRStepObj*
imports
IREval
begin

2.1 Heap

The heap model we introduce maps field references to object instances to runtime values. We use the H[f][p] heap representation. See *cite*{*heap−reps−2011*}.

We also introduce the DynamicHeap type which allocates new object references sequentially storing the next free object reference as 'Free'.

type-synonym ($'a$, $'b$) *Heap* = $'a \Rightarrow 'b \Rightarrow$ *Value*
type-synonym *Free* = *nat*
type-synonym ($'a$, $'b$) *DynamicHeap* = ($'a$, $'b$) *Heap* $\times$ *Free*

fun *h-load-field* :: $'a \Rightarrow 'b \Rightarrow$ ($'a$, $'b$) *DynamicHeap* $\Rightarrow$ *Value* **where**
h-load-field r f (h, n) = h r f

fun *h-store-field* :: $'a \Rightarrow 'b \Rightarrow$ *Value* $\Rightarrow$ ($'a$, $'b$) *DynamicHeap* $\Rightarrow$ ($'a$, $'b$) *DynamicHeap* **where**
h-store-field r f v (h, n) = (h(r := ((h r)(f := v))), n)

fun *h-new-inst* :: ($'a$, $'b$) *DynamicHeap* $\Rightarrow$ ($'a$, $'b$) *DynamicHeap* $\times$ *Value* **where**
h-new-inst (h, n) = ((h,n+*1*), (*ObjRef* (*Some* n)))

type-synonym *RefFieldHeap* = (*objref*, *string*) *DynamicHeap*

definition *new-heap* :: ($'a$, $'b$) *DynamicHeap* **where**
new-heap = ((λf. λp. *UndefVal*), *0*)

2.2 Intraprocedural Semantics

Intraprocedural semantics are given as a small-step semantics.

Within the context of a graph, the configuration triple, (ID, MethodState, Heap), is related to the subsequent configuration.

inductive *step* :: *IRGraph* $\Rightarrow$ *Params* $\Rightarrow$ (*ID* $\times$ *MapState* $\times$ *RefFieldHeap*) $\Rightarrow$ (*ID* $\times$ *MapState* $\times$ *RefFieldHeap*) $\Rightarrow$ *bool*
(-, - $\vdash$ - $\rightarrow$ - *55*) **for** *g p* **where**

SequentialNode:
$[\![$*is-sequential-node* (*kind g nid*);
nid$'$ = (*successors-of* (*kind g nid*))!*0*$]\!]$
$\Longrightarrow$ *g*, *p* $\vdash$ (*nid*, *m*, *h*) $\rightarrow$ (*nid*$'$, *m*, *h*) |

IfNode:
$[\![$*kind g nid* = (*IfNode cond tb fb*);
[*g*, *m*, *p*] $\vdash$ (*kind g cond*) $\mapsto$ *val*;
nid$'$ = (*if val-to-bool val then tb else fb*)$]\!]$
$\Longrightarrow$ *g*, *p* $\vdash$ (*nid*, *m*, *h*) $\rightarrow$ (*nid*$'$, *m*, *h*) |

EndNodes:
$[\![$*is-AbstractEndNode* (*kind g nid*);
merge = *any-usage g nid*;
is-AbstractMergeNode (*kind g merge*);

i = *find-index nid* (*inputs-of* (*kind g merge*));
phis = (*phi-list g merge*);
inps = (*phi-inputs g i phis*);
[*g*, *m*, *p*] $\vdash$ *inps* $\longmapsto$ *vs*;

m$'$ = *set-phis phis vs m*$]\!]$
$\Longrightarrow$ *g*, *p* $\vdash$ (*nid*, *m*, *h*) $\rightarrow$ (*merge*, *m*$'$, *h*) |

NewInstanceNode:
$[\![$*kind g nid* = (*NewInstanceNode nid f obj nid*$'$);
(*h*$'$, *ref*) = *h-new-inst h*;
m$'$ = *m*(*nid* := *ref*)$]\!]$
$\Longrightarrow$ *g*, *p* $\vdash$ (*nid*, *m*, *h*) $\rightarrow$ (*nid*$'$, *m*$'$, *h*$'$) |

LoadFieldNode:
$[\![$*kind g nid* = (*LoadFieldNode nid f* (*Some obj*) *nid*$'$);
[*g*, *m*, *p*] $\vdash$ (*kind g obj*) $\mapsto$ *ObjRef ref*;
h-load-field ref f h = *v*;
m$'$ = *m*(*nid* := *v*)$]\!]$
$\Longrightarrow$ *g*, *p* $\vdash$ (*nid*, *m*, *h*) $\rightarrow$ (*nid*$'$, *m*$'$, *h*) |

SignedDivNode:
$[\![$*kind g nid* = (*SignedDivNode nid x y zero sb nxt*);
[*g*, *m*, *p*] $\vdash$ (*kind g x*) $\mapsto$ *v1*;
[*g*, *m*, *p*] $\vdash$ (*kind g y*) $\mapsto$ *v2*;
v = (*intval-div v1 v2*);
m$'$ = *m*(*nid* := *v*)$]\!]$
$\Longrightarrow$ *g*, *p* $\vdash$ (*nid*, *m*, *h*) $\rightarrow$ (*nxt*, *m*$'$, *h*) |

SignedRemNode:
$[\![\mathit{kind}\ g\ \mathit{nid} = (\mathit{SignedRemNode}\ \mathit{nid}\ x\ y\ \mathit{zero}\ \mathit{sb}\ \mathit{nxt});$
$[g,\ m,\ p] \vdash (\mathit{kind}\ g\ x) \mapsto \mathit{v1};$
$[g,\ m,\ p] \vdash (\mathit{kind}\ g\ y) \mapsto \mathit{v2};$
$v = (\mathit{intval\text{-}mod}\ \mathit{v1}\ \mathit{v2});$
$m' = \ m(\mathit{nid} := v)]\!]$
$\Longrightarrow g,\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow (\mathit{nxt},\ m',\ h)\ |$

StaticLoadFieldNode:
$[\![\mathit{kind}\ g\ \mathit{nid} = (\mathit{LoadFieldNode}\ \mathit{nid}\ f\ \mathit{None}\ \mathit{nid}');$
$\mathit{h\text{-}load\text{-}field}\ \mathit{None}\ f\ h = v;$
$m' = \ m(\mathit{nid} := v)]\!]$
$\Longrightarrow g,\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow (\mathit{nid}',\ m',\ h)\ |$

StoreFieldNode:
$[\![\mathit{kind}\ g\ \mathit{nid} = (\mathit{StoreFieldNode}\ \mathit{nid}\ f\ \mathit{newval}\ \text{-}\ (\mathit{Some}\ \mathit{obj})\ \mathit{nid}');$
$[g,\ m,\ p] \vdash (\mathit{kind}\ g\ \mathit{newval}) \mapsto \mathit{val};$
$[g,\ m,\ p] \vdash (\mathit{kind}\ g\ \mathit{obj}) \mapsto \mathit{ObjRef}\ \mathit{ref};$
$h' = \mathit{h\text{-}store\text{-}field}\ \mathit{ref}\ f\ \mathit{val}\ h;$
$m' = \ m(\mathit{nid} := \mathit{val})]\!]$
$\Longrightarrow g,\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow (\mathit{nid}',\ m',\ h')\ |$

StaticStoreFieldNode:
$[\![\mathit{kind}\ g\ \mathit{nid} = (\mathit{StoreFieldNode}\ \mathit{nid}\ f\ \mathit{newval}\ \text{-}\ \mathit{None}\ \mathit{nid}');$
$[g,\ m,\ p] \vdash (\mathit{kind}\ g\ \mathit{newval}) \mapsto \mathit{val};$
$h' = \mathit{h\text{-}store\text{-}field}\ \mathit{None}\ f\ \mathit{val}\ h;$
$m' = \ m(\mathit{nid} := \mathit{val})]\!]$
$\Longrightarrow g,\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow (\mathit{nid}',\ m',\ h')$

code-pred (*modes*: $i \Rightarrow i \Rightarrow i * i * i \Rightarrow o * o * o \Rightarrow \mathit{bool}$) *step* **.**

We prove that within the same graph, a configuration triple will always transition to the same subsequent configuration. Therefore, our step semantics is deterministic.

theorem *stepDet*:
$(g,\ p \vdash (\mathit{nid},m,h) \rightarrow \mathit{next}) \Longrightarrow$
$(\forall\ \mathit{next}'.\ ((g,\ p \vdash (\mathit{nid},m,h) \rightarrow \mathit{next}') \longrightarrow \mathit{next} = \mathit{next}'))$
proof (*induction rule*: *step.induct*)
case (*SequentialNode nid next m h*)
have *notif*: $\neg(\mathit{is\text{-}IfNode}\ (\mathit{kind}\ g\ \mathit{nid}))$
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-IfNode-def*)
have *notend*: $\neg(\mathit{is\text{-}AbstractEndNode}\ (\mathit{kind}\ g\ \mathit{nid}))$
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-AbstractEndNode.simps is-EndNode.elims*(*2*) *is-LoopEndNode-def*)
have *notnew*: $\neg(\mathit{is\text{-}NewInstanceNode}\ (\mathit{kind}\ g\ \mathit{nid}))$
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-NewInstanceNode-def*)
have *notload*: $\neg(\mathit{is\text{-}LoadFieldNode}\ (\mathit{kind}\ g\ \mathit{nid}))$

using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-LoadFieldNode-def*)
have *notstore*: ¬(*is-StoreFieldNode* (*kind g nid*))
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-StoreFieldNode-def*)
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps is-SignedDivNode-def is-SignedRemNode-def*
by (*metis is-IntegerDivRemNode.simps*)
from *notif notend notnew notload notstore notdivrem*
show *?case* **using** *SequentialNode step.cases*
by (*smt* (*verit*) *IRNode.discI*(*18*) *is-IfNode-def is-NewInstanceNode-def is-StoreFieldNode-def is-sequential-node.simps*(*38*) *is-sequential-node.simps*(*39*) *old.prod.inject*)
next
case (*IfNode nid cond tb fb m val next h*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *IfNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *IfNode.hyps*(*1*))
have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *IfNode.hyps*(*1*))
from *notseq notend notdivrem* **show** *?case* **using** *IfNode evalDet*
using *IRNode.distinct*(*871*) *IRNode.distinct*(*891*) *IRNode.distinct*(*909*) *IRNode.distinct*(*923*)
by (*smt* (*z3*) *IRNode.distinct*(*893*) *IRNode.distinct*(*913*) *IRNode.distinct*(*927*) *IRNode.distinct*(*929*) *IRNode.distinct*(*933*) *IRNode.distinct*(*947*) *IRNode.inject*(*11*) *Pair-inject step.simps*)
next
case (*EndNodes nid merge i phis inputs m vs m′ h*)
have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-AbstractEndNode.simps is-sequential-node.simps*
by (*metis is-EndNode.elims*(*2*) *is-LoopEndNode-def*)
have *notif*: ¬(*is-IfNode* (*kind g nid*))
using *EndNodes.hyps*(*1*)
by (*metis is-AbstractEndNode.elims*(*1*) *is-EndNode.simps*(*12*) *is-IfNode-def IRNode.distinct-disc*(*900*))
have *notref*: ¬(*is-RefNode* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-sequential-node.simps*
using *IRNode.disc*(*1899*) *IRNode.distinct*(*1473*) *is-AbstractEndNode.simps is-EndNode.elims*(*2*) *is-LoopEndNode-def is-RefNode-def*
by (*metis IRNode.distinct*(*737*) *IRNode.distinct-disc*(*1518*))
have *notnew*: ¬(*is-NewInstanceNode* (*kind g nid*))
using *EndNodes.hyps*(*1*) *is-AbstractEndNode.simps*
using *IRNode.distinct-disc*(*1442*) *is-EndNode.simps*(*29*) *is-NewInstanceNode-def*
by (*metis IRNode.distinct-disc*(*1483*))
have *notload*: ¬(*is-LoadFieldNode* (*kind g nid*))

```
    using EndNodes.hyps(1) is-AbstractEndNode.simps
    by (metis IRNode.disc(939) is-EndNode.simps(19) is-LoadFieldNode-def)
  have notstore: ¬(is-StoreFieldNode (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps
    using IRNode.distinct-disc(1504) is-EndNode.simps(39) is-StoreFieldNode-def
    by fastforce
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps is-SignedDivNode-def is-SignedRemNode-def
    using IRNode.distinct-disc(1498) IRNode.distinct-disc(1500) is-IntegerDivRemNode.simps
is-EndNode.simps(36) is-EndNode.simps(37)
    by auto
  from notseq notif notref notnew notload notstore notdivrem
  show ?case using EndNodes evalAllDet
    by (smt (z3) is-IfNode-def is-LoadFieldNode-def is-NewInstanceNode-def is-RefNode-def
is-StoreFieldNode-def is-SignedDivNode-def is-SignedRemNode-def Pair-inject is-IntegerDivRemNode.elims(3)
step.cases)
next
  case (NewInstanceNode nid f obj nxt h' ref h m' m)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notif: ¬(is-IfNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notref: ¬(is-RefNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notload: ¬(is-LoadFieldNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notstore: ¬(is-StoreFieldNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notdivrem:  ¬(is-IntegerDivRemNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  from notseq notend notif notref notload notstore notdivrem
  show ?case using NewInstanceNode step.cases
    by (smt (z3) IRNode.discI(11) IRNode.discI(18) IRNode.discI(38) IRNode.distinct(1777)
IRNode.distinct(1779) IRNode.distinct(1797) IRNode.inject(28) Pair-inject)
next
  case (LoadFieldNode nid f obj nxt m ref h v m')
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
```

```
    using is-AbstractEndNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  have notdivrem:  ¬(is-IntegerDivRemNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using LoadFieldNode step.cases
   by (smt (z3) IRNode.distinct(1333) IRNode.distinct(1347) IRNode.distinct(1349)
IRNode.distinct(1353) IRNode.distinct(893) IRNode.inject(18) Pair-inject Value.inject(4)
evalDet option.distinct(1) option.inject)
next
  case (StaticLoadFieldNode nid f nxt h v m' m)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StaticLoadFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StaticLoadFieldNode step.cases
   by (smt (z3) IRNode.distinct(1333) IRNode.distinct(1347) IRNode.distinct(1349)
IRNode.distinct(1353) IRNode.distinct(1367) IRNode.distinct(893) IRNode.distinct(1297)
IRNode.distinct(1315) IRNode.distinct(1329) IRNode.distinct(871) IRNode.inject(18)
Pair-inject option.discI)
next
  case (StoreFieldNode nid f newval uu obj nxt m val ref h' h m')
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StoreFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StoreFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StoreFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StoreFieldNode step.cases
   by (smt (z3) IRNode.distinct(1353) IRNode.distinct(1783) IRNode.distinct(1965)
IRNode.distinct(1983) IRNode.distinct(933) IRNode.inject(38) Pair-inject Value.inject(4)
evalDet option.distinct(1) option.inject)
next
  case (StaticStoreFieldNode nid f newval uv nxt m val h' h m')
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
```

have *notdivrem*: ¬(*is-IntegerDivRemNode* (*kind g nid*))
by (*simp add*: *StaticStoreFieldNode.hyps*(*1*))
from *notseq notend notdivrem*
show *?case* **using** *StoreFieldNode step.cases*
by (*smt* (*z3*) *IRNode.distinct*(*1315*) *IRNode.distinct*(*1353*) *IRNode.distinct*(*1783*) *IRNode.distinct*(*1965*)
IRNode.distinct(*1983*) *IRNode.distinct*(*2027*) *IRNode.distinct*(*933*) *IRNode.inject*(*38*) *IRNode.distinct*(*1725*) *Pair-inject StaticStoreFieldNode.hyps*(*1*) *StaticStoreFieldNode.hyps*(*2*) *StaticStoreFieldNode.hyps*(*3*) *StaticStoreFieldNode.hyps*(*4*) *evalDet option.discI*)
next
case (*SignedDivNode nid x y zero sb nxt m v1 v2 v m′ h*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *SignedDivNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *SignedDivNode.hyps*(*1*))
from *notseq notend*
show *?case* **using** *SignedDivNode step.cases*
by (*smt* (*z3*) *IRNode.distinct*(*1347*) *IRNode.distinct*(*1777*) *IRNode.distinct*(*1961*) *IRNode.distinct*(*1965*) *IRNode.distinct*(*1979*) *IRNode.distinct*(*927*) *IRNode.inject*(*35*) *Pair-inject evalDet*)
next
case (*SignedRemNode nid x y zero sb nxt m v1 v2 v m′ h*)
then have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *SignedRemNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *SignedRemNode.hyps*(*1*))
from *notseq notend*
show *?case* **using** *SignedRemNode step.cases*
by (*smt* (*z3*) *IRNode.distinct*(*1349*) *IRNode.distinct*(*1779*) *IRNode.distinct*(*1961*) *IRNode.distinct*(*1983*) *IRNode.distinct*(*1997*) *IRNode.distinct*(*929*) *IRNode.inject*(*36*) *Pair-inject evalDet*)
qed

lemma *stepRefNode*:
⟦*kind g nid* = *RefNode nid′*⟧ ⟹ *g*, *p* ⊢ (*nid*,*m*,*h*) → (*nid′*,*m*,*h*)
by (*simp add*: *SequentialNode*)

lemma *IfNodeStepCases*:
assumes *kind g nid* = *IfNode cond tb fb*
assumes [*g*, *m*, *p*] ⊢ *kind g cond* ↦ *v*
assumes *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*)
shows *nid′* ∈ {*tb*, *fb*}
using *step.IfNode*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *insert-iff prod.inject stepDet*)

lemma *IfNodeSeq*:
  shows *kind g nid* = *IfNode cond tb fb* ⟶ ¬(*is-sequential-node* (*kind g nid*))
  unfolding *is-sequential-node.simps* **by** *simp*

lemma *IfNodeCond*:
  assumes *kind g nid* = *IfNode cond tb fb*
  assumes g, $p \vdash (nid, m, h) \rightarrow (nid', m, h)$
  shows $\exists\ v.\ ([g, m, p] \vdash kind\ g\ cond \mapsto v)$
  using *assms*(*2*,*1*) **by** (*induct* (*nid*,*m*,*h*) (*nid′*,*m*,*h*) *rule*: *step.induct*; *auto*)

lemma *step-in-ids*:
  assumes g, $p \vdash (nid, m, h) \rightarrow (nid', m', h')$
  shows $nid \in ids\ g$
  using *assms* **apply** (*induct* (*nid*, *m*, *h*) (*nid′*, *m′*, *h′*) *rule*: *step.induct*)
  using *is-sequential-node.simps*(*45*) *not-in-g*
  apply *simp*
  apply (*metis is-sequential-node.simps*(*46*))
  using *ids-some* **apply** (*metis IRNode.simps*(*990*))
  using *EndNodes*(*1*) *is-AbstractEndNode.simps is-EndNode.simps*(*45*) *ids-some*
  apply (*metis IRNode.disc*(*965*))
  by *simp+*

2.3 Interprocedural Semantics

type-synonym *Signature* = *string*
type-synonym *Program* = *Signature* ⇀ *IRGraph*

inductive *step-top* :: *Program* ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *RefFieldHeap* ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *RefFieldHeap* ⇒ *bool*
  (- ⊢ - ⟶ - *55*)
  for *P* **where**

  Lift:
  $[\![g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h')]\!]$
    $\Longrightarrow P \vdash ((g,nid,m,p)\#stk,\ h) \longrightarrow ((g,nid',m',p)\#stk,\ h')$ |

  InvokeNodeStep:
  ⟦*is-Invoke* (*kind g nid*);

    callTarget = *ir-callTarget* (*kind g nid*);
    kind g callTarget = (*MethodCallTargetNode targetMethod arguments*);
    Some targetGraph = *P targetMethod*;
    m′ = *new-map-state*;
    $[g,\ m,\ p] \vdash arguments \longmapsto p'$⟧
    $\Longrightarrow P \vdash ((g,nid,m,p)\#stk,\ h) \longrightarrow ((targetGraph,0,m',p')\#(g,nid,m,p)\#stk,\ h)$
|

ReturnNode:
$[\![kind\ g\ nid = (ReturnNode\ (Some\ expr)\ \text{-});$
$[g,\ m,\ p] \vdash (kind\ g\ expr) \mapsto v;$

$cm' = cm(cnid := v);$
$cnid' = (successors\text{-}of\ (kind\ cg\ cnid))!0]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,cnid',cm',cp)\#stk,\ h)\ |$

ReturnNodeVoid:
$[\![kind\ g\ nid = (ReturnNode\ None\ \text{-});$
$cm' = cm(cnid := (ObjRef\ (Some\ (2048))));$

$cnid' = (successors\text{-}of\ (kind\ cg\ cnid))!0]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,cnid',cm',cp)\#stk,\ h)\ |$

UnwindNode:
$[\![kind\ g\ nid = (UnwindNode\ exception);$

$[g,\ m,\ p] \vdash (kind\ g\ exception) \mapsto e;$

$kind\ cg\ cnid = (InvokeWithExceptionNode\ \text{- - - - - -}\ exEdge);$

$cm' = cm(cnid := e)]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,exEdge,cm',cp)\#stk,\ h)$

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

2.4 Big-step Execution

type-synonym $Trace = (IRGraph \times ID \times MapState \times Params)\ list$

fun *has-return* :: $MapState \Rightarrow bool$ **where**
$has\text{-}return\ m = (m\ 0 \neq UndefVal)$

inductive *exec* :: *Program*
$\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times RefFieldHeap$
$\Rightarrow Trace$
$\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times RefFieldHeap$
$\Rightarrow Trace$
$\Rightarrow bool$
$(\text{-} \vdash \text{-} \mid \text{-} \longrightarrow * \text{-} \mid \text{-})$
for P
where
$[\![P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h');$
$\neg(has\text{-}return\ m');$

$l' = (l\ @\ [(g,\ nid,m,p)]);$

$exec\ P\ (((g',nid',m',p')\#ys),h')\ l'\ next\text{-}state\ l'']\!]$

$\Longrightarrow$ *exec P (((g,nid,m,p)#xs),h) l next-state l''*

|
$[\![$*P* $\vdash$ *(((g,nid,m,p)#xs),h)* $\longrightarrow$ *(((g',nid',m',p')#ys),h')*;
has-return m';

l' = (l @ [(g,nid,m,p)])$]\!]$
$\Longrightarrow$ *exec P (((g,nid,m,p)#xs),h) l (((g',nid',m',p')#ys),h') l'*

code-pred (*modes*: *i* $\Rightarrow$ *i* $\Rightarrow$ *i* $\Rightarrow$ *o* $\Rightarrow$ *o* $\Rightarrow$ *bool as Exec*) *exec* **.**

inductive *exec-debug* :: *Program*
$\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState* $\times$ *Params*) *list* $\times$ *RefFieldHeap*
$\Rightarrow$ *nat*
$\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState* $\times$ *Params*) *list* $\times$ *RefFieldHeap*
$\Rightarrow$ *bool*
(-$\vdash$-$\longrightarrow$*-* -)
where
$[\![$*n > 0*;
p $\vdash$ *s* $\longrightarrow$ *s'*;
exec-debug p s' (n − 1) s''$]\!]$
$\Longrightarrow$ *exec-debug p s n s''* |

$[\![$*n = 0*$]\!]$
$\Longrightarrow$ *exec-debug p s n s*

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

2.4.1 Heap Testing

definition *p3*:: *Params* **where**
p3 = [IntVal32 3]

values {(*prod.fst*(*prod.snd* (*prod.snd* (*hd* (*prod.fst res*))))) *0*
| *res*. (λ*x* . *Some eg2-sq*) $\vdash$ ([(*eg2-sq*,*0*,*new-map-state*,*p3*), (*eg2-sq*,*0*,*new-map-state*,*p3*)], *new-heap*) $\rightarrow$**2** *res*}

definition *field-sq* :: *string* **where**
field-sq = ''sq''

definition *eg3-sq* :: *IRGraph* **where**
eg3-sq = irgraph [
(*0*, *StartNode None 4*, *VoidStamp*),
(*1*, *ParameterNode 0*, *default-stamp*),
(*3*, *MulNode 1 1*, *default-stamp*),
(*4*, *StoreFieldNode 4 field-sq 3 None None 5*, *VoidStamp*),
(*5*, *ReturnNode* (*Some 3*) *None*, *default-stamp*)
]

values {*h-load-field None field-sq* (*prod.snd res*)
| *res.* (λx. *Some eg3-sq*) $\vdash$ ([(*eg3-sq*, *0*, *new-map-state*, *p3*), (*eg3-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) $\to\!*3*$ *res*}

definition *eg4-sq* :: *IRGraph* **where**
eg4-sq = *irgraph* [
(*0*, *StartNode None 4*, *VoidStamp*),
(*1*, *ParameterNode 0*, *default-stamp*),
(*3*, *MulNode 1 1*, *default-stamp*),
(*4*, *NewInstanceNode 4 ''obj-class'' None 5*, *ObjectStamp ''obj-class'' True True True*),
(*5*, *StoreFieldNode 5 field-sq 3 None* (*Some 4*) *6*, *VoidStamp*),
(*6*, *ReturnNode* (*Some 3*) *None*, *default-stamp*)
]

values {*h-load-field* (*Some 0*) *field-sq* (*prod.snd res*)
| *res.* (λx. *Some eg4-sq*) $\vdash$ ([(*eg4-sq*, *0*, *new-map-state*, *p3*), (*eg4-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) $\to\!*3*$ *res*}
end