

Backing up Slicing: Verifying the interprocedural two-phase Horwitz-Reps-Binkley Slicer

Daniel Wasserrab

May 31, 2023

Abstract

Slicing is a widely-used technique with applications in e.g. compiler technology and software security. Thus verification of algorithms in these areas is often based on the correctness of slicing, which should ideally be proven independent of concrete programming languages and with the help of well-known verifying techniques such as proof assistants.

After verifying static intraprocedural and dynamic slicing [3], we focus now on the sophisticated interprocedural two-phase Horwitz-Reps-Binkley slicer [1], including summary edges which were added in [2].

Again, abstracting from concrete syntax we base our work on a graph representation of the program fulfilling certain structural and well-formedness properties. The framework is instantiated with a simple While language with procedures, showing its validity.

0.1 Auxiliary lemmas

theory *AuxLemmas* **imports** *Main* **begin**

Lemma concerning maps and @

lemma *map-append-append-maps*:

assumes $map:map\ f\ xs = ys@zs$

obtains $xs'\ xs''$ **where** $map\ f\ xs' = ys$ **and** $map\ f\ xs'' = zs$ **and** $xs = xs'@xs''$

<proof>

Lemma concerning splitting of lists

lemma *path-split-general*:

assumes $all:\forall\ zs.\ xs \neq ys@zs$

obtains $j\ zs$ **where** $xs = (take\ j\ ys)@zs$ **and** $j < length\ ys$

and $\forall\ k > j.\ \forall\ zs'. xs \neq (take\ k\ ys)@zs'$

<proof>

end

Chapter 1

The Framework

theory *BasicDefs* **imports** *AuxLemmas* **begin**

As slicing is a program analysis that can be completely based on the information given in the CFG, we want to provide a framework which allows us to formalize and prove properties of slicing regardless of the actual programming language. So the starting point for the formalization is the definition of an abstract CFG, i.e. without considering features specific for certain languages. By doing so we ensure that our framework is as generic as possible since all proofs hold for every language whose CFG conforms to this abstract CFG.

Static Slicing analyses a CFG prior to execution. Whereas dynamic slicing can provide better results for certain inputs (i.e. trace and initial state), static slicing is more conservative but provides results independent of inputs. Correctness for static slicing is defined using a weak simulation between nodes and states when traversing the original and the sliced graph. The weak simulation property demands that if a (node,state) tuples (n_1, s_1) simulates (n_2, s_2) and making an observable move in the original graph leads from (n_1, s_1) to (n'_1, s'_1) , this tuple simulates a tuple (n_2, s_2) which is the result of making an observable move in the sliced graph beginning in (n'_2, s'_2) .

1.1 Basic Definitions

fun *fun-upds* :: ('a \Rightarrow 'b) \Rightarrow 'a list \Rightarrow 'b list \Rightarrow ('a \Rightarrow 'b)
where *fun-upds* f [] ys = f
| *fun-upds* f xs [] = f
| *fun-upds* f (x#xs) (y#ys) = (*fun-upds* f xs ys)(x := y)

notation *fun-upds* (-'(- /[:=]/ -')

lemma *fun-upds-nth*:

$\llbracket i < \text{length } xs; \text{length } xs = \text{length } ys; \text{distinct } xs \rrbracket$

$\implies f(xs [:=] ys)(xs!i) = (ys!i)$
 ⟨proof⟩

lemma *fun-upds-eq*:

assumes $V \in \text{set } xs$ **and** $\text{length } xs = \text{length } ys$ **and** *distinct* xs

shows $f(xs [:=] ys) V = f'(xs [:=] ys) V$

⟨proof⟩

lemma *fun-upds-notin*: $x \notin \text{set } xs \implies f(xs [:=] ys) x = f x$

⟨proof⟩

1.1.1 *distinct-fst*

definition *distinct-fst* :: $('a \times 'b) \text{ list} \Rightarrow \text{bool}$ **where**

$\text{distinct-fst} \equiv \text{distinct} \circ \text{map } \text{fst}$

lemma *distinct-fst-Nil* [*simp*]:

$\text{distinct-fst } []$

⟨proof⟩

lemma *distinct-fst-Cons* [*simp*]:

$\text{distinct-fst } ((k,x)\#kxs) = (\text{distinct-fst } kxs \wedge (\forall y. (k,y) \notin \text{set } kxs))$

⟨proof⟩

lemma *distinct-fst-isin-same-fst*:

$\llbracket (x,y) \in \text{set } xs; (x,y') \in \text{set } xs; \text{distinct-fst } xs \rrbracket$

$\implies y = y'$

⟨proof⟩

1.1.2 **Edge kinds**

Every procedure has a unique name, e.g. in object oriented languages *pname* refers to class + procedure.

A state is a call stack of tuples, which consists of:

1. data information, i.e. a mapping from the local variables in the call frame to their values, and
2. control flow information, e.g. which node called the current procedure.

Update and predicate edges check and manipulate only the data information of the top call stack element. Call and return edges however may use the data and control flow information present in the top stack element to state if this edge is traversable. The call edge additionally has a list of functions to determine what values the parameters have in a certain call frame and

control flow information for the return. The return edge is concerned with passing the values of the return parameter values to the underlying stack frame. See the funtions *transfer* and *pred* in locale *CFG*.

```
datatype (dead 'var, dead 'val, dead 'ret, dead 'pname) edge-kind =
  UpdateEdge ('var  $\rightarrow$  'val)  $\Rightarrow$  ('var  $\rightarrow$  'val)           ( $\uparrow$ -)
| PredicateEdge ('var  $\rightarrow$  'val)  $\Rightarrow$  bool                ('(-) $\surd$ )
| CallEdge ('var  $\rightarrow$  'val)  $\times$  'ret  $\Rightarrow$  bool 'ret 'pname
    (('var  $\rightarrow$  'val)  $\rightarrow$  'val) list                    (-: $\leftrightarrow$ - 70)
| ReturnEdge ('var  $\rightarrow$  'val)  $\times$  'ret  $\Rightarrow$  bool 'pname
    ('var  $\rightarrow$  'val)  $\Rightarrow$  ('var  $\rightarrow$  'val)  $\Rightarrow$  ('var  $\rightarrow$  'val) (- $\leftrightarrow$ - 70)
```

definition *intra-kind* :: ('var,'val,'ret,'pname) edge-kind \Rightarrow bool
where *intra-kind* et \equiv ($\exists f$. et = $\uparrow f$) \vee ($\exists Q$. et = (Q) \surd)

lemma *edge-kind-cases* [case-names *Intra Call Return*]:
 \llbracket intra-kind et \Rightarrow P; $\bigwedge Q$ r p fs. et = Q:r \leftrightarrow pfs \Rightarrow P;
 $\bigwedge Q$ p f. et = Q \leftrightarrow pf \Rightarrow P $\rrbracket \Rightarrow$ P
 <proof>

end

1.2 CFG

theory *CFG* imports *BasicDefs* begin

1.2.1 The abstract CFG

Locale fixes and assumptions

```
locale CFG =
  fixes sourcenode :: 'edge  $\Rightarrow$  'node
  fixes targetnode :: 'edge  $\Rightarrow$  'node
  fixes kind :: 'edge  $\Rightarrow$  ('var,'val,'ret,'pname) edge-kind
  fixes valid-edge :: 'edge  $\Rightarrow$  bool
  fixes Entry::'node ('(-Entry'-))
  fixes get-proc::'node  $\Rightarrow$  'pname
  fixes get-return-edges::'edge  $\Rightarrow$  'edge set
  fixes procs::('pname  $\times$  'var list  $\times$  'var list) list
  fixes Main::'pname
  assumes Entry-target [dest]:  $\llbracket$ valid-edge a; targetnode a = (-Entry-) $\rrbracket \Rightarrow$  False
  and get-proc-Entry:get-proc (-Entry-) = Main
  and Entry-no-call-source:
     $\llbracket$ valid-edge a; kind a = Q:r $\leftrightarrow$ pfs; sourcenode a = (-Entry-) $\rrbracket \Rightarrow$  False
  and edge-det:
     $\llbracket$ valid-edge a; valid-edge a'; sourcenode a = sourcenode a';
    targetnode a = targetnode a' $\rrbracket \Rightarrow$  a = a'
```

and *Main-no-call-target*: $\llbracket \text{valid-edge } a; \text{ kind } a = Q:r \hookrightarrow \text{Mainf} \rrbracket \implies \text{False}$
and *Main-no-return-source*: $\llbracket \text{valid-edge } a; \text{ kind } a = Q' \leftarrow \text{Mainf} \rrbracket \implies \text{False}$
and *callee-in-procs*:
 $\llbracket \text{valid-edge } a; \text{ kind } a = Q:r \hookrightarrow pfs \rrbracket \implies \exists \text{ ins outs. } (p, \text{ins}, \text{outs}) \in \text{set procs}$
and *get-proc-intra*: $\llbracket \text{valid-edge } a; \text{ intra-kind}(\text{kind } a) \rrbracket$
 $\implies \text{get-proc}(\text{sourcenode } a) = \text{get-proc}(\text{targetnode } a)$
and *get-proc-call*:
 $\llbracket \text{valid-edge } a; \text{ kind } a = Q:r \hookrightarrow pfs \rrbracket \implies \text{get-proc}(\text{targetnode } a) = p$
and *get-proc-return*:
 $\llbracket \text{valid-edge } a; \text{ kind } a = Q' \leftarrow pf \rrbracket \implies \text{get-proc}(\text{sourcenode } a) = p$
and *call-edges-only*: $\llbracket \text{valid-edge } a; \text{ kind } a = Q:r \hookrightarrow pfs \rrbracket$
 $\implies \forall a'. \text{valid-edge } a' \wedge \text{targetnode } a' = \text{targetnode } a \longrightarrow$
 $(\exists Qx \text{ rx fsx. kind } a' = Qx:rx \hookrightarrow pfsx)$
and *return-edges-only*: $\llbracket \text{valid-edge } a; \text{ kind } a = Q' \leftarrow pf \rrbracket$
 $\implies \forall a'. \text{valid-edge } a' \wedge \text{sourcenode } a' = \text{sourcenode } a \longrightarrow$
 $(\exists Qx \text{ fx. kind } a' = Qx \leftarrow pfx)$
and *get-return-edge-call*:
 $\llbracket \text{valid-edge } a; \text{ kind } a = Q:r \hookrightarrow pfs \rrbracket \implies \text{get-return-edges } a \neq \{\}$
and *get-return-edges-valid*:
 $\llbracket \text{valid-edge } a; a' \in \text{get-return-edges } a \rrbracket \implies \text{valid-edge } a'$
and *only-call-get-return-edges*:
 $\llbracket \text{valid-edge } a; a' \in \text{get-return-edges } a \rrbracket \implies \exists Q \text{ r p fs. kind } a = Q:r \hookrightarrow pfs$
and *call-return-edges*:
 $\llbracket \text{valid-edge } a; \text{ kind } a = Q:r \hookrightarrow pfs; a' \in \text{get-return-edges } a \rrbracket$
 $\implies \exists Q' \text{ f'}. \text{kind } a' = Q' \leftarrow pf'$
and *return-needs-call*: $\llbracket \text{valid-edge } a; \text{ kind } a = Q' \leftarrow pf \rrbracket$
 $\implies \exists !a'. \text{valid-edge } a' \wedge (\exists Q \text{ r fs. kind } a' = Q:r \hookrightarrow pfs) \wedge a \in \text{get-return-edges}$
 a'
and *intra-proc-additional-edge*:
 $\llbracket \text{valid-edge } a; a' \in \text{get-return-edges } a \rrbracket$
 $\implies \exists a''. \text{valid-edge } a'' \wedge \text{sourcenode } a'' = \text{targetnode } a \wedge$
 $\text{targetnode } a'' = \text{sourcenode } a' \wedge \text{kind } a'' = (\lambda cf. \text{False})_{\checkmark}$
and *call-return-node-edge*:
 $\llbracket \text{valid-edge } a; a' \in \text{get-return-edges } a \rrbracket$
 $\implies \exists a''. \text{valid-edge } a'' \wedge \text{sourcenode } a'' = \text{sourcenode } a \wedge$
 $\text{targetnode } a'' = \text{targetnode } a' \wedge \text{kind } a'' = (\lambda cf. \text{False})_{\checkmark}$
and *call-only-one-intra-edge*:
 $\llbracket \text{valid-edge } a; \text{ kind } a = Q:r \hookrightarrow pfs \rrbracket$
 $\implies \exists !a'. \text{valid-edge } a' \wedge \text{sourcenode } a' = \text{sourcenode } a \wedge \text{intra-kind}(\text{kind } a')$
and *return-only-one-intra-edge*:
 $\llbracket \text{valid-edge } a; \text{ kind } a = Q' \leftarrow pf \rrbracket$
 $\implies \exists !a'. \text{valid-edge } a' \wedge \text{targetnode } a' = \text{targetnode } a \wedge \text{intra-kind}(\text{kind } a')$
and *same-proc-call-unique-target*:
 $\llbracket \text{valid-edge } a; \text{ valid-edge } a'; \text{ kind } a = Q_1:r_1 \hookrightarrow pfs_1; \text{ kind } a' = Q_2:r_2 \hookrightarrow pfs_2 \rrbracket$
 $\implies \text{targetnode } a = \text{targetnode } a'$
and *unique-callers*: *distinct-fst procs*
and *distinct-formal-ins*: $(p, \text{ins}, \text{outs}) \in \text{set procs} \implies \text{distinct ins}$
and *distinct-formal-outs*: $(p, \text{ins}, \text{outs}) \in \text{set procs} \implies \text{distinct outs}$

begin

lemma *get-proc-get-return-edge*:

assumes *valid-edge a* **and** $a' \in \text{get-return-edges } a$

shows $\text{get-proc } (\text{sourcenode } a) = \text{get-proc } (\text{targetnode } a')$

$\langle \text{proof} \rangle$

lemma *call-intra-edge-False*:

assumes *valid-edge a* **and** $\text{kind } a = Q:r \hookrightarrow_p fs$ **and** *valid-edge a'*

and $\text{sourcenode } a = \text{sourcenode } a'$ **and** $\text{intra-kind}(\text{kind } a')$

shows $\text{kind } a' = (\lambda cf. \text{False})_{\checkmark}$

$\langle \text{proof} \rangle$

lemma *formal-in-THE*:

$\llbracket \text{valid-edge } a; \text{kind } a = Q:r \hookrightarrow_p fs; (p, \text{ins}, \text{outs}) \in \text{set procs} \rrbracket$

$\implies (\text{THE } \text{ins}. \exists \text{outs}. (p, \text{ins}, \text{outs}) \in \text{set procs}) = \text{ins}$

$\langle \text{proof} \rangle$

lemma *formal-out-THE*:

$\llbracket \text{valid-edge } a; \text{kind } a = Q \leftarrow_p f; (p, \text{ins}, \text{outs}) \in \text{set procs} \rrbracket$

$\implies (\text{THE } \text{outs}. \exists \text{ins}. (p, \text{ins}, \text{outs}) \in \text{set procs}) = \text{outs}$

$\langle \text{proof} \rangle$

Transfer and predicate functions

fun *params* :: $((\text{'var} \rightarrow \text{'val}) \rightarrow \text{'val}) \text{ list} \Rightarrow (\text{'var} \rightarrow \text{'val}) \Rightarrow \text{'val option list}$

where *params* [] *cf* = []

| *params* (*f*#*fs*) *cf* = (*f cf*)#*params fs cf*

lemma *params-nth*:

$i < \text{length } fs \implies (\text{params } fs \text{ cf})!i = (fs!i) \text{ cf}$

$\langle \text{proof} \rangle$

lemma [*simp*]: $\text{length } (\text{params } fs \text{ cf}) = \text{length } fs$

$\langle \text{proof} \rangle$

fun *transfer* :: $(\text{'var}, \text{'val}, \text{'ret}, \text{'pname}) \text{ edge-kind} \Rightarrow ((\text{'var} \rightarrow \text{'val}) \times \text{'ret}) \text{ list} \Rightarrow$
 $((\text{'var} \rightarrow \text{'val}) \times \text{'ret}) \text{ list}$

where *transfer* ($\uparrow f$) (*cf*#*cfs*) = (*f (fst cf), snd cf*)#*cfs*

| *transfer* (Q) _{\checkmark} (*cf*#*cfs*) = (*cf*#*cfs*)

| *transfer* ($Q:r \hookrightarrow_p fs$) (*cf*#*cfs*) =

(*let ins* = *THE ins. \exists outs. (p, ins, outs) \in set procs in*

(*Map.empty*(*ins* [:=] *params fs (fst cf)*), *r*)#*cf*#*cfs*)

| $transfer (Q \leftarrow pf) (cf \# cfs) = (case\ cfs\ of\ [] \Rightarrow []$
| $cf' \# cfs' \Rightarrow (f\ (fst\ cf)\ (fst\ cf'),\ snd\ cf') \# cfs')$
| $transfer\ et\ [] = []$

fun $transfers :: ('var, 'val, 'ret, 'pname)\ edge\ kind\ list \Rightarrow (('var \rightarrow 'val) \times 'ret)\ list$
 \Rightarrow
 $(('var \rightarrow 'val) \times 'ret)\ list$

where $transfers\ []\ s = s$
| $transfers\ (et \# ets)\ s = transfers\ ets\ (transfer\ et\ s)$

fun $pred :: ('var, 'val, 'ret, 'pname)\ edge\ kind \Rightarrow (('var \rightarrow 'val) \times 'ret)\ list \Rightarrow bool$
where $pred\ (\uparrow f)\ (cf \# cfs) = True$
| $pred\ (Q)_{\surd}\ (cf \# cfs) = Q\ (fst\ cf)$
| $pred\ (Q: r \leftarrow pf)\ (cf \# cfs) = Q\ (fst\ cf, r)$
| $pred\ (Q \leftarrow pf)\ (cf \# cfs) = (Q\ cf \wedge cfs \neq [])$
| $pred\ et\ [] = False$

fun $preds :: ('var, 'val, 'ret, 'pname)\ edge\ kind\ list \Rightarrow (('var \rightarrow 'val) \times 'ret)\ list \Rightarrow bool$

where $preds\ []\ s = True$
| $preds\ (et \# ets)\ s = (pred\ et\ s \wedge preds\ ets\ (transfer\ et\ s))$

lemma $transfers\ split:$

$(transfers\ (ets @ ets')\ s) = (transfers\ ets'\ (transfers\ ets\ s))$
 $\langle proof \rangle$

lemma $preds\ split:$

$(preds\ (ets @ ets')\ s) = (preds\ ets\ s \wedge preds\ ets'\ (transfers\ ets\ s))$
 $\langle proof \rangle$

abbreviation $state\ val :: (('var \rightarrow 'val) \times 'ret)\ list \Rightarrow 'var \rightarrow 'val$
where $state\ val\ s\ V \equiv (fst\ (hd\ s))\ V$

valid-node

definition $valid\ node :: 'node \Rightarrow bool$

where $valid\ node\ n \equiv$
 $(\exists a.\ valid\ edge\ a \wedge (n = sourcenode\ a \vee n = targetnode\ a))$

lemma $[simp]: valid\ edge\ a \Longrightarrow valid\ node\ (sourcenode\ a)$
 $\langle proof \rangle$

lemma $[simp]: valid\ edge\ a \Longrightarrow valid\ node\ (targetnode\ a)$
 $\langle proof \rangle$

1.2.2 CFG paths

inductive *path* :: 'node \Rightarrow 'edge list \Rightarrow 'node \Rightarrow bool

(- \dashrightarrow^* - [51,0,0] 80)

where

empty-path:valid-node $n \implies n - [] \rightarrow^* n$

| *Cons-path*:

$\llbracket n'' - as \rightarrow^* n'; \text{valid-edge } a; \text{sourcenode } a = n; \text{targetnode } a = n' \rrbracket$

$\implies n - a \# as \rightarrow^* n'$

lemma *path-valid-node*:

assumes $n - as \rightarrow^* n'$ **shows** *valid-node* n **and** *valid-node* n'

<proof>

lemma *empty-path-nodes* [*dest*]: $n - [] \rightarrow^* n' \implies n = n'$

<proof>

lemma *path-valid-edges*: $n - as \rightarrow^* n' \implies \forall a \in \text{set } as. \text{valid-edge } a$

<proof>

lemma *path-edge:valid-edge* $a \implies \text{sourcenode } a - [a] \rightarrow^* \text{targetnode } a$

<proof>

lemma *path-Append*: $\llbracket n - as \rightarrow^* n''; n'' - as' \rightarrow^* n' \rrbracket$

$\implies n - as @ as' \rightarrow^* n'$

<proof>

lemma *path-split*:

assumes $n - as @ a \# as' \rightarrow^* n'$

shows $n - as \rightarrow^* \text{sourcenode } a$ **and** *valid-edge* a **and** $\text{targetnode } a - as' \rightarrow^* n'$

<proof>

lemma *path-split-Cons*:

assumes $n - as \rightarrow^* n'$ **and** $as \neq []$

obtains $a' as'$ **where** $as = a' \# as'$ **and** $n = \text{sourcenode } a'$

and *valid-edge* a' **and** $\text{targetnode } a' - as' \rightarrow^* n'$

<proof>

lemma *path-split-snoc*:

assumes $n - as \rightarrow^* n'$ **and** $as \neq []$

obtains $a' as'$ **where** $as = as' @ [a']$ **and** $n - as' \rightarrow^* \text{sourcenode } a'$

and *valid-edge* a' **and** $n' = \text{targetnode } a'$

<proof>

lemma *path-split-second*:

assumes $n -as@a\#as'\rightarrow* n'$ **shows** *sourcenode* $a -a\#as'\rightarrow* n'$
<proof>

lemma *path-Entry-Cons*:

assumes $(-Entry-) -as\rightarrow* n'$ **and** $n' \neq (-Entry-)$
obtains n **where** *sourcenode* $a = (-Entry-)$ **and** *targetnode* $a = n$
and $n -tl as\rightarrow* n'$ **and** *valid-edge* a **and** $a = hd as$
<proof>

lemma *path-det*:

$\llbracket n -as\rightarrow* n'; n -as\rightarrow* n'' \rrbracket \implies n' = n''$
<proof>

definition

sourcenodes :: 'edge list \Rightarrow 'node list
where *sourcenodes* $xs \equiv map$ *sourcenode* xs

definition

kinds :: 'edge list \Rightarrow ('var,'val,'ret,'pname) edge-kind list
where *kinds* $xs \equiv map$ *kind* xs

definition

targetnodes :: 'edge list \Rightarrow 'node list
where *targetnodes* $xs \equiv map$ *targetnode* xs

lemma *path-sourcenode*:

$\llbracket n -as\rightarrow* n'; as \neq [] \rrbracket \implies hd (sourcenodes as) = n$
<proof>

lemma *path-targetnode*:

$\llbracket n -as\rightarrow* n'; as \neq [] \rrbracket \implies last (targetnodes as) = n'$
<proof>

lemma *sourcenodes-is-n-Cons-butlast-targetnodes*:

$\llbracket n -as\rightarrow* n'; as \neq [] \rrbracket \implies$
 $sourcenodes as = n\#(butlast (targetnodes as))$
<proof>

lemma *targetnodes-is-tl-sourcenodes-App-n'*:
 $\llbracket n - as \rightarrow_* n'; as \neq [] \rrbracket \implies$
 $targetnodes\ as = (tl\ (sourcenodes\ as))@[n']$
<proof>

Intraprocedural paths

definition *intra-path* :: 'node \Rightarrow 'edge list \Rightarrow 'node \Rightarrow bool
 $(- \dashrightarrow_i^* - [51,0,0] 80)$
where $n - as \rightarrow_i^* n' \equiv n - as \rightarrow_* n' \wedge (\forall a \in set\ as.\ intra\ kind(kind\ a))$

lemma *intra-path-get-procs*:
assumes $n - as \rightarrow_i^* n'$ **shows** $get\ proc\ n = get\ proc\ n'$
<proof>

lemma *intra-path-Append*:
 $\llbracket n - as \rightarrow_i^* n''; n'' - as' \rightarrow_i^* n' \rrbracket \implies n - as @ as' \rightarrow_i^* n'$
<proof>

lemma *get-proc-get-return-edges*:
assumes *valid-edge a* **and** $a' \in get\ return\ edges\ a$
shows $get\ proc(targetnode\ a) = get\ proc(sourcenode\ a')$
<proof>

Valid paths

declare *conj-cong*[*fundef-cong*]

fun *valid-path-aux* :: 'edge list \Rightarrow 'edge list \Rightarrow bool
where *valid-path-aux* $cs [] \longleftrightarrow True$
| *valid-path-aux* $cs (a\#\ as) \longleftrightarrow$
 $(case\ (kind\ a)\ of\ Q:r \hookrightarrow pfs \Rightarrow valid\ path\ aux\ (a\#\ cs)\ as$
 $\quad | Q \hookrightarrow pf \Rightarrow case\ cs\ of\ [] \Rightarrow valid\ path\ aux\ []\ as$
 $\quad \quad | c'\#\ cs' \Rightarrow a \in get\ return\ edges\ c' \wedge$
 $\quad \quad \quad valid\ path\ aux\ cs'\ as$
| $- \Rightarrow valid\ path\ aux\ cs\ as)$

lemma *vpa-induct* [*consumes 1, case-names vpa-empty vpa-intra vpa-Call vpa-ReturnEmpty vpa-ReturnCons*]:
assumes *major*: *valid-path-aux xs ys*
and *rules*: $\bigwedge cs.\ P\ cs\ []$
 $\bigwedge cs\ a\ as.\ \llbracket intra\ kind(kind\ a); valid\ path\ aux\ cs\ as; P\ cs\ as \rrbracket \implies P\ cs\ (a\#\ as)$
 $\bigwedge cs\ a\ as\ Q\ r\ p\ fs.\ \llbracket kind\ a = Q:r \hookrightarrow pfs; valid\ path\ aux\ (a\#\ cs)\ as; P\ (a\#\ cs)\ as \rrbracket$
 $\implies P\ cs\ (a\#\ as)$

$\wedge cs\ a\ as\ Q\ p\ f. \llbracket kind\ a = Q \leftrightarrow pf; cs = []; valid\text{-}path\text{-}aux\ []\ as; P\ []\ as \rrbracket$
 $\implies P\ cs\ (a\#as)$
 $\wedge cs\ a\ as\ Q\ p\ f\ c'\ cs'. \llbracket kind\ a = Q \leftrightarrow pf; cs = c'\#cs'; valid\text{-}path\text{-}aux\ cs'\ as;$
 $a \in get\text{-}return\text{-}edges\ c'; P\ cs'\ as \rrbracket$
 $\implies P\ cs\ (a\#as)$
shows $P\ xs\ ys$
 $\langle proof \rangle$

lemma *valid-path-aux-intra-path*:
 $\forall a \in set\ as. intra\text{-}kind(kind\ a) \implies valid\text{-}path\text{-}aux\ cs\ as$
 $\langle proof \rangle$

lemma *valid-path-aux-callstack-prefix*:
 $valid\text{-}path\text{-}aux\ (cs@cs')\ as \implies valid\text{-}path\text{-}aux\ cs\ as$
 $\langle proof \rangle$

fun *upd-cs* :: 'edge list \Rightarrow 'edge list \Rightarrow 'edge list
where $upd\text{-}cs\ cs\ [] = cs$
 $| upd\text{-}cs\ cs\ (a\#as) =$
 $(case\ (kind\ a)\ of\ Q:r \leftrightarrow pfs \Rightarrow upd\text{-}cs\ (a\#cs)\ as$
 $| Q \leftrightarrow pf \Rightarrow case\ cs\ of\ [] \Rightarrow upd\text{-}cs\ cs\ as$
 $| c'\#cs' \Rightarrow upd\text{-}cs\ cs'\ as$
 $| - \Rightarrow upd\text{-}cs\ cs\ as)$

lemma *upd-cs-empty* [*dest*]:
 $upd\text{-}cs\ cs\ [] = [] \implies cs = []$
 $\langle proof \rangle$

lemma *upd-cs-intra-path*:
 $\forall a \in set\ as. intra\text{-}kind(kind\ a) \implies upd\text{-}cs\ cs\ as = cs$
 $\langle proof \rangle$

lemma *upd-cs-Append*:
 $\llbracket upd\text{-}cs\ cs\ as = cs'; upd\text{-}cs\ cs'\ as' = cs'' \rrbracket \implies upd\text{-}cs\ cs\ (as@as') = cs''$
 $\langle proof \rangle$

lemma *upd-cs-empty-split*:
assumes $upd\text{-}cs\ cs\ as = []$ **and** $cs \neq []$ **and** $as \neq []$
obtains $xs\ ys$ **where** $as = xs@ys$ **and** $xs \neq []$ **and** $upd\text{-}cs\ cs\ xs = []$
and $\forall xs'\ ys'. xs = xs'@ys' \wedge ys' \neq [] \longrightarrow upd\text{-}cs\ cs\ xs' \neq []$
and $upd\text{-}cs\ []\ ys = []$
 $\langle proof \rangle$

lemma *upd-cs-snoc-Return-Cons*:
assumes $kind\ a = Q \leftrightarrow pf$
shows $upd-cs\ cs\ as = c' \# cs' \implies upd-cs\ cs\ (as@[a]) = cs'$
 $\langle proof \rangle$

lemma *upd-cs-snoc-Call*:
assumes $kind\ a = Q:r \hookrightarrow pfs$
shows $upd-cs\ cs\ (as@[a]) = a \# (upd-cs\ cs\ as)$
 $\langle proof \rangle$

lemma *valid-path-aux-split*:
assumes $valid-path-aux\ cs\ (as@as')$
shows $valid-path-aux\ cs\ as$ **and** $valid-path-aux\ (upd-cs\ cs\ as)\ as'$
 $\langle proof \rangle$

lemma *valid-path-aux-Append*:
 $\llbracket valid-path-aux\ cs\ as; valid-path-aux\ (upd-cs\ cs\ as)\ as' \rrbracket$
 $\implies valid-path-aux\ cs\ (as@as')$
 $\langle proof \rangle$

lemma *vpa-snoc-Call*:
assumes $kind\ a = Q:r \hookrightarrow pfs$
shows $valid-path-aux\ cs\ as \implies valid-path-aux\ cs\ (as@[a])$
 $\langle proof \rangle$

definition *valid-path* :: $'edge\ list \Rightarrow bool$
where $valid-path\ as \equiv valid-path-aux\ []\ as$

lemma *valid-path-aux-valid-path*:
 $valid-path-aux\ cs\ as \implies valid-path\ as$
 $\langle proof \rangle$

lemma *valid-path-split*:
assumes $valid-path\ (as@as')$ **shows** $valid-path\ as$ **and** $valid-path\ as'$
 $\langle proof \rangle$

definition *valid-path'* :: 'node \Rightarrow 'edge list \Rightarrow 'node \Rightarrow bool
 (- $\dashrightarrow_{\sqrt{*}}$ - [51,0,0] 80)
where *vp-def*: $n -as \rightarrow_{\sqrt{*}} n' \equiv n -as \rightarrow_* n' \wedge \text{valid-path } as$

lemma *intra-path-vp*:
assumes $n -as \rightarrow_{\iota_*} n'$ **shows** $n -as \rightarrow_{\sqrt{*}} n'$
 $\langle \text{proof} \rangle$

lemma *vp-split-Cons*:
assumes $n -as \rightarrow_{\sqrt{*}} n'$ **and** $as \neq []$
obtains $a' as'$ **where** $as = a' \# as'$ **and** $n = \text{sourcenode } a'$
and *valid-edge* a' **and** *targetnode* $a' -as' \rightarrow_{\sqrt{*}} n'$
 $\langle \text{proof} \rangle$

lemma *vp-split-snoc*:
assumes $n -as \rightarrow_{\sqrt{*}} n'$ **and** $as \neq []$
obtains $a' as'$ **where** $as = as'@[a']$ **and** $n -as' \rightarrow_{\sqrt{*}} \text{sourcenode } a'$
and *valid-edge* a' **and** $n' = \text{targetnode } a'$
 $\langle \text{proof} \rangle$

lemma *vp-split*:
assumes $n -as@a\#as' \rightarrow_{\sqrt{*}} n'$
shows $n -as \rightarrow_{\sqrt{*}} \text{sourcenode } a$ **and** *valid-edge* a **and** *targetnode* $a -as' \rightarrow_{\sqrt{*}} n'$
 $\langle \text{proof} \rangle$

lemma *vp-split-second*:
assumes $n -as@a\#as' \rightarrow_{\sqrt{*}} n'$ **shows** $\text{sourcenode } a -a\#as' \rightarrow_{\sqrt{*}} n'$
 $\langle \text{proof} \rangle$

function *valid-path-rev-aux* :: 'edge list \Rightarrow 'edge list \Rightarrow bool
where *valid-path-rev-aux* $cs [] \longleftrightarrow \text{True}$
 | *valid-path-rev-aux* $cs (as@[a]) \longleftrightarrow$
 (case (kind a) of $Q \leftrightarrow pf \Rightarrow \text{valid-path-rev-aux } (a\#cs) as$
 | $Q:r \leftrightarrow pfs \Rightarrow \text{case } cs \text{ of } [] \Rightarrow \text{valid-path-rev-aux } [] as$
 | $c' \# cs' \Rightarrow c' \in \text{get-return-edges } a \wedge$
 valid-path-rev-aux $cs' as$
 | - $\Rightarrow \text{valid-path-rev-aux } cs as$)
 $\langle \text{proof} \rangle$

termination $\langle \text{proof} \rangle$

lemma *vpra-induct* [consumes 1, case-names *vpra-empty vpra-intra vpra-Return*

vpra-CallEmpty vpra-CallCons]:
assumes *major*: *valid-path-rev-aux xs ys*
and *rules*: $\bigwedge cs. P cs []$
 $\bigwedge cs a as. \llbracket \text{intra-kind}(\text{kind } a); \text{valid-path-rev-aux } cs as; P cs as \rrbracket$
 $\implies P cs (as@[a])$
 $\bigwedge cs a as Q p f. \llbracket \text{kind } a = Q \leftrightarrow pf; \text{valid-path-rev-aux } (a\#cs) as; P (a\#cs) as \rrbracket$
 $\implies P cs (as@[a])$
 $\bigwedge cs a as Q r p fs. \llbracket \text{kind } a = Q:r \hookrightarrow pfs; cs = []; \text{valid-path-rev-aux } [] as;$
 $P [] as \rrbracket \implies P cs (as@[a])$
 $\bigwedge cs a as Q r p fs c' cs'. \llbracket \text{kind } a = Q:r \hookrightarrow pfs; cs = c'\#cs';$
 $\text{valid-path-rev-aux } cs' as; c' \in \text{get-return-edges } a; P cs' as \rrbracket$
 $\implies P cs (as@[a])$
shows $P xs ys$
 $\langle \text{proof} \rangle$

lemma *vpra-callstack-prefix*:
 $\text{valid-path-rev-aux } (cs@[cs']) as \implies \text{valid-path-rev-aux } cs as$
 $\langle \text{proof} \rangle$

function *upd-rev-cs* :: *'edge list* \Rightarrow *'edge list* \Rightarrow *'edge list*
where *upd-rev-cs* $cs [] = cs$
 $| \text{upd-rev-cs } cs (as@[a]) =$
 $(\text{case } (\text{kind } a) \text{ of } Q \leftrightarrow pf \Rightarrow \text{upd-rev-cs } (a\#cs) as$
 $| Q:r \hookrightarrow pfs \Rightarrow \text{case } cs \text{ of } [] \Rightarrow \text{upd-rev-cs } cs as$
 $| c'\#cs' \Rightarrow \text{upd-rev-cs } cs' as$
 $| - \Rightarrow \text{upd-rev-cs } cs as)$
 $\langle \text{proof} \rangle$
termination $\langle \text{proof} \rangle$

lemma *upd-rev-cs-empty* [*dest*]:
 $\text{upd-rev-cs } cs [] = [] \implies cs = []$
 $\langle \text{proof} \rangle$

lemma *valid-path-rev-aux-split*:
assumes $\text{valid-path-rev-aux } cs (as@[as'])$
shows $\text{valid-path-rev-aux } cs as'$ **and** $\text{valid-path-rev-aux } (\text{upd-rev-cs } cs as')$ as
 $\langle \text{proof} \rangle$

lemma *valid-path-rev-aux-Append*:
 $\llbracket \text{valid-path-rev-aux } cs as'; \text{valid-path-rev-aux } (\text{upd-rev-cs } cs as') as \rrbracket$
 $\implies \text{valid-path-rev-aux } cs (as@[as'])$
 $\langle \text{proof} \rangle$

lemma *vpra-Cons-intra*:
assumes *intra-kind*(*kind a*)
shows *valid-path-rev-aux cs as* \implies *valid-path-rev-aux cs (a#as)*
 \langle *proof* \rangle

lemma *vpra-Cons-Return*:
assumes *kind a = Q* \leftrightarrow *pf*
shows *valid-path-rev-aux cs as* \implies *valid-path-rev-aux cs (a#as)*
 \langle *proof* \rangle

lemma *upd-rev-cs-Cons-intra*:
assumes *intra-kind*(*kind a*) **shows** *upd-rev-cs cs (a#as) = upd-rev-cs cs as*
 \langle *proof* \rangle

lemma *upd-rev-cs-Cons-Return*:
assumes *kind a = Q* \leftrightarrow *pf* **shows** *upd-rev-cs cs (a#as) = a#(upd-rev-cs cs as)*
 \langle *proof* \rangle

lemma *upd-rev-cs-Cons-Call-Cons*:
assumes *kind a = Q:r* \leftrightarrow *pf*
shows *upd-rev-cs cs as = c'#cs'* \implies *upd-rev-cs cs (a#as) = cs'*
 \langle *proof* \rangle

lemma *upd-rev-cs-Cons-Call-Cons-Empty*:
assumes *kind a = Q:r* \leftrightarrow *pf*
shows *upd-rev-cs cs as = []* \implies *upd-rev-cs cs (a#as) = []*
 \langle *proof* \rangle

definition *valid-call-list* :: *'edge list* \Rightarrow *'node* \Rightarrow *bool*
where *valid-call-list cs n* \equiv
 $\forall cs' c cs''. cs = cs'@c\#cs'' \longrightarrow (valid-edge c \wedge (\exists Q r p fs. (kind c = Q:r\leftrightarrow pfs)$
 \wedge
 $p = get-proc (case cs' of [] \Rightarrow n \mid - \Rightarrow last (sourcenodes cs'))))$

definition *valid-return-list* :: *'edge list* \Rightarrow *'node* \Rightarrow *bool*
where *valid-return-list cs n* \equiv
 $\forall cs' c cs''. cs = cs'@c\#cs'' \longrightarrow (valid-edge c \wedge (\exists Q p f. (kind c = Q\leftrightarrow pf) \wedge$
 $p = get-proc (case cs' of [] \Rightarrow n \mid - \Rightarrow last (targetnodes cs'))))$

lemma *valid-call-list-valid-edges*:
assumes *valid-call-list cs n* **shows** $\forall c \in set\ cs. valid-edge\ c$
 \langle *proof* \rangle

lemma *valid-return-list-valid-edges*:

assumes *valid-return-list* *rs n* **shows** $\forall r \in \text{set } rs. \text{valid-edge } r$
 $\langle \text{proof} \rangle$

lemma *vpra-empty-valid-call-list-rev*:

valid-call-list cs n \implies *valid-path-rev-aux* [] (*rev cs*)
 $\langle \text{proof} \rangle$

lemma *vpa-upd-cs-cases*:

$\llbracket \text{valid-path-aux } cs \text{ as}; \text{valid-call-list } cs \text{ n}; n -as \rightarrow^* n' \rrbracket$
 \implies *case* (*upd-cs cs as*) of [] \Rightarrow ($\forall c \in \text{set } cs. \exists a \in \text{set } as. a \in \text{get-return-edges } c$)
 $\quad \quad \quad | \text{cx}\#csx \Rightarrow \text{valid-call-list } (\text{cx}\#csx) \text{ n}'$
 $\langle \text{proof} \rangle$

lemma *vpa-valid-call-list-valid-return-list-vpra*:

$\llbracket \text{valid-path-aux } cs \text{ cs}'; \text{valid-call-list } cs \text{ n}; \text{valid-return-list } cs' \text{ n}' \rrbracket$
 \implies *valid-path-rev-aux* *cs'* (*rev cs*)
 $\langle \text{proof} \rangle$

lemma *vpa-to-vpra*:

$\llbracket \text{valid-path-aux } cs \text{ as}; \text{valid-path-aux } (\text{upd-cs } cs \text{ as}) \text{ cs}';$
 $n -as \rightarrow^* n'; \text{valid-call-list } cs \text{ n}; \text{valid-return-list } cs' \text{ n}' \rrbracket$
 \implies *valid-path-rev-aux* *cs'* *as* \wedge *valid-path-rev-aux* (*upd-rev-cs cs' as*) (*rev cs*)
 $\langle \text{proof} \rangle$

lemma *vp-to-vpra*:

$n -as \rightarrow \surd^* n' \implies \text{valid-path-rev-aux} [] \text{ as}$
 $\langle \text{proof} \rangle$

Same level paths

fun *same-level-path-aux* :: 'edge list \Rightarrow 'edge list \Rightarrow bool

where *same-level-path-aux* *cs* [] \longleftrightarrow True

| *same-level-path-aux* *cs* (*a*#*as*) \longleftrightarrow

(*case* (*kind a*) of *Q*:*r* \leftrightarrow *pfs* \Rightarrow *same-level-path-aux* (*a*#*cs*) *as*

| *Q* \leftrightarrow *pf* \Rightarrow *case cs* of [] \Rightarrow False

| *c*'#*cs'* $\Rightarrow a \in \text{get-return-edges } c' \wedge$
same-level-path-aux *cs'* *as*

| - \Rightarrow *same-level-path-aux* *cs as*)

lemma *slpa-induct* [*consumes 1, case-names slpa-empty slpa-intra slpa-Call slpa-Return*]:
assumes *major: same-level-path-aux xs ys*
and rules: $\bigwedge cs. P cs \square$
 $\bigwedge cs a as. \llbracket \text{intra-kind}(\text{kind } a); \text{same-level-path-aux } cs \text{ as}; P cs as \rrbracket$
 $\implies P cs (a\#as)$
 $\bigwedge cs a as Q r p fs. \llbracket \text{kind } a = Q:r \hookrightarrow pf; \text{same-level-path-aux } (a\#cs) \text{ as}; P (a\#cs) as \rrbracket$
 $\implies P cs (a\#as)$
 $\bigwedge cs a as Q p f c' cs'. \llbracket \text{kind } a = Q \leftrightarrow pf; cs = c'\#cs'; \text{same-level-path-aux } cs' as; a \in \text{get-return-edges } c'; P cs' as \rrbracket$
 $\implies P cs (a\#as)$
shows $P xs ys$
 $\langle \text{proof} \rangle$

lemma *slpa-cases* [*consumes 4, case-names intra-path return-intra-path*]:
assumes *same-level-path-aux cs as and upd-cs cs as = []*
and $\forall c \in \text{set } cs. \text{valid-edge } c$ **and** $\forall a \in \text{set } as. \text{valid-edge } a$
obtains $\forall a \in \text{set } as. \text{intra-kind}(\text{kind } a)$
 $| asx a asx' Q p f c' cs' \textbf{ where } as = asx@a\#asx' \textbf{ and } \text{same-level-path-aux } cs \text{ asx}$
 $\textbf{ and } \text{kind } a = Q \leftrightarrow pf \textbf{ and } \text{upd-cs } cs \text{ asx} = c'\#cs' \textbf{ and } \text{upd-cs } cs (asx@[a]) =$
 \square
and $a \in \text{get-return-edges } c' \textbf{ and } \text{valid-edge } c'$
and $\forall a \in \text{set } asx'. \text{intra-kind}(\text{kind } a)$
 $\langle \text{proof} \rangle$

lemma *same-level-path-aux-valid-path-aux*:
 $\text{same-level-path-aux } cs \text{ as} \implies \text{valid-path-aux } cs \text{ as}$
 $\langle \text{proof} \rangle$

lemma *same-level-path-aux-Append*:
 $\llbracket \text{same-level-path-aux } cs \text{ as}; \text{same-level-path-aux } (\text{upd-cs } cs \text{ as}) \text{ as}' \rrbracket$
 $\implies \text{same-level-path-aux } cs (as@as')$
 $\langle \text{proof} \rangle$

lemma *same-level-path-aux-callstack-Append*:
 $\text{same-level-path-aux } cs \text{ as} \implies \text{same-level-path-aux } (cs@cs') \text{ as}$
 $\langle \text{proof} \rangle$

lemma *same-level-path-upd-cs-callstack-Append*:
 $\llbracket \text{same-level-path-aux } cs \text{ as}; \text{upd-cs } cs \text{ as} = cs' \rrbracket$
 $\implies \text{upd-cs } (cs@cs') \text{ as} = (cs'@cs')$
 $\langle \text{proof} \rangle$

lemma *slpa-split*:

assumes *same-level-path-aux cs as* **and** $as = xs@ys$ **and** $upd\text{-}cs\ cs\ xs = []$
shows *same-level-path-aux cs xs* **and** *same-level-path-aux [] ys*
 ⟨*proof*⟩

lemma *slpa-number-Calls-eq-number>Returns*:

$\llbracket same\text{-}level\text{-}path\text{-}aux\ cs\ as; upd\text{-}cs\ cs\ as = [];$
 $\forall a \in set\ as.\ valid\text{-}edge\ a; \forall c \in set\ cs.\ valid\text{-}edge\ c \rrbracket$
 $\implies length\ [a \leftarrow as@cs.\ \exists Q\ r\ p\ fs.\ kind\ a = Q:r \leftrightarrow pfs] =$
 $length\ [a \leftarrow as.\ \exists Q\ p\ f.\ kind\ a = Q \leftrightarrow pf]$
 ⟨*proof*⟩

lemma *slpa-get-proc*:

$\llbracket same\text{-}level\text{-}path\text{-}aux\ cs\ as; upd\text{-}cs\ cs\ as = []; n -as \rightarrow * n';$
 $\forall c \in set\ cs.\ valid\text{-}edge\ c \rrbracket$
 $\implies (if\ cs = []\ then\ get\text{-}proc\ n\ else\ get\text{-}proc(last(sourcenodes\ cs))) = get\text{-}proc\ n'$
 ⟨*proof*⟩

lemma *slpa-get-return-edges*:

$\llbracket same\text{-}level\text{-}path\text{-}aux\ cs\ as; cs \neq []; upd\text{-}cs\ cs\ as = [];$
 $\forall xs\ ys.\ as = xs@ys \wedge ys \neq [] \longrightarrow upd\text{-}cs\ cs\ xs \neq [] \rrbracket$
 $\implies last\ as \in get\text{-}return\text{-}edges\ (last\ cs)$
 ⟨*proof*⟩

lemma *slpa-callstack-length*:

assumes *same-level-path-aux cs as* **and** $length\ cs = length\ cfsx$
obtains $cfx\ cfsx'$ **where** *transfers (kinds as) (cfsx@cf#cfs) = cfsx'@cfx#cfs*
and *transfers (kinds as) (cfsx@cf#cfs') = cfsx'@cfx#cfs'*
and $length\ cfsx' = length\ (upd\text{-}cs\ cs\ as)$
 ⟨*proof*⟩

lemma *slpa-snoc-intra*:

$\llbracket same\text{-}level\text{-}path\text{-}aux\ cs\ as; intra\text{-}kind\ (kind\ a) \rrbracket$
 $\implies same\text{-}level\text{-}path\text{-}aux\ cs\ (as@[a])$
 ⟨*proof*⟩

lemma *slpa-snoc-Call*:

$\llbracket same\text{-}level\text{-}path\text{-}aux\ cs\ as; kind\ a = Q:r \leftrightarrow pfs \rrbracket$
 $\implies same\text{-}level\text{-}path\text{-}aux\ cs\ (as@[a])$
 ⟨*proof*⟩

lemma *vpa-Main-slpa*:

[[*valid-path-aux cs as; m -as→* m'; as ≠ []*;
valid-call-list cs m; get-proc m' = Main;
get-proc (case cs of [] ⇒ m | - ⇒ sourcenode (last cs)) = Main]]
 \implies *same-level-path-aux cs as ∧ upd-cs cs as = []*
 ⟨*proof*⟩

definition *same-level-path* :: *'edge list ⇒ bool*

where *same-level-path as ≡ same-level-path-aux [] as ∧ upd-cs [] as = []*

lemma *same-level-path-valid-path*:

same-level-path as ⇒ valid-path as
 ⟨*proof*⟩

lemma *same-level-path-Append*:

[[*same-level-path as; same-level-path as'*]] \implies *same-level-path (as@as')*
 ⟨*proof*⟩

lemma *same-level-path-number-Calls-eq-number>Returns*:

[[*same-level-path as; ∀ a ∈ set as. valid-edge a*]] \implies
length [a←as. ∃ Q r p fs. kind a = Q:r↔pfs] = length [a←as. ∃ Q p f. kind a =
Q↔pf]
 ⟨*proof*⟩

lemma *same-level-path-valid-path-Append*:

[[*same-level-path as; valid-path as'*]] \implies *valid-path (as@as')*
 ⟨*proof*⟩

lemma *valid-path-same-level-path-Append*:

[[*valid-path as; same-level-path as'*]] \implies *valid-path (as@as')*
 ⟨*proof*⟩

lemma *intra-same-level-path*:

assumes $\forall a \in \text{set } as. \text{intra-kind}(\text{kind } a)$ **shows** *same-level-path as*
 ⟨*proof*⟩

definition *same-level-path'* :: *'node ⇒ 'edge list ⇒ 'node ⇒ bool*

(- ---_{sl^*} - [51,0,0] 80)

where *slp-def:n -as→_{sl*} n' ≡ n -as→* n' ∧ same-level-path as*

lemma *slp-vp*: *n -as→_{sl*} n' ⇒ n -as→_{√*} n'*

$\langle proof \rangle$

lemma *intra-path-slp*: $n - as \rightarrow_{\iota}^* n' \implies n - as \rightarrow_{sl}^* n'$
 $\langle proof \rangle$

lemma *slp-Append*:

$\llbracket n - as \rightarrow_{sl}^* n''; n'' - as' \rightarrow_{sl}^* n' \rrbracket \implies n - as @ as' \rightarrow_{sl}^* n'$
 $\langle proof \rangle$

lemma *slp-vp-Append*:

$\llbracket n - as \rightarrow_{sl}^* n''; n'' - as' \rightarrow_{\sqrt{}}^* n' \rrbracket \implies n - as @ as' \rightarrow_{\sqrt{}}^* n'$
 $\langle proof \rangle$

lemma *vp-slp-Append*:

$\llbracket n - as \rightarrow_{\sqrt{}}^* n''; n'' - as' \rightarrow_{sl}^* n' \rrbracket \implies n - as @ as' \rightarrow_{\sqrt{}}^* n'$
 $\langle proof \rangle$

lemma *slp-get-proc*:

$n - as \rightarrow_{sl}^* n' \implies get\text{-proc } n = get\text{-proc } n'$
 $\langle proof \rangle$

lemma *same-level-path-inner-path*:

assumes $n - as \rightarrow_{sl}^* n'$
obtains as' **where** $n - as' \rightarrow_{\iota}^* n'$ **and** $set(\text{sourcenodes } as') \subseteq set(\text{sourcenodes } as)$
 $\langle proof \rangle$

lemma *slp-callstack-length-equal*:

assumes $n - as \rightarrow_{sl}^* n'$ **obtains** cf' **where** *transfers* (*kinds* as) ($cf \# cfs$) = $cf' \# cfs$
and *transfers* (*kinds* as) ($cf \# cfs'$) = $cf' \# cfs'$
 $\langle proof \rangle$

lemma *slp-cases* [*consumes 1, case-names intra-path return-intra-path*]:

assumes $m - as \rightarrow_{sl}^* m'$
obtains $m - as \rightarrow_{\iota}^* m'$
 $| as' a as'' Q p f$ **where** $as = as' @ a \# as''$ **and** $kind a = Q \leftrightarrow pf$
and $m - as' @ [a] \rightarrow_{sl}^* \text{targetnode } a$ **and** $\text{targetnode } a - as'' \rightarrow_{\iota}^* m'$
 $\langle proof \rangle$

function *same-level-path-rev-aux* :: 'edge list \Rightarrow 'edge list \Rightarrow bool
where *same-level-path-rev-aux* cs [] \longleftrightarrow True
| *same-level-path-rev-aux* cs (as@[a]) \longleftrightarrow
 (case (kind a) of Q \leftrightarrow pf \Rightarrow *same-level-path-rev-aux* (a#cs) as
 | Q:r \hookrightarrow pf \Rightarrow case cs of [] \Rightarrow False
 | c'#cs' \Rightarrow c' \in get-return-edges a \wedge
 same-level-path-rev-aux cs' as
 | - \Rightarrow *same-level-path-rev-aux* cs as)
<proof>
termination <proof>

lemma *slpra-induct* [consumes 1, case-names *slpra-empty slpra-intra slpra-Return slpra-Call*]:
assumes major: *same-level-path-rev-aux* xs ys
and rules: \bigwedge cs. P cs []
 \bigwedge cs a as. [[*intra-kind*(kind a); *same-level-path-rev-aux* cs as; P cs as]]
 \Rightarrow P cs (as@[a])
 \bigwedge cs a as Q p f. [[kind a = Q \leftrightarrow pf; *same-level-path-rev-aux* (a#cs) as; P (a#cs)
as]]
 \Rightarrow P cs (as@[a])
 \bigwedge cs a as Q r p fs c' cs'. [[kind a = Q:r \hookrightarrow pf; cs = c'#cs';
 same-level-path-rev-aux cs' as; c' \in get-return-edges a; P cs' as]]
 \Rightarrow P cs (as@[a])
shows P xs ys
<proof>

lemma *same-level-path-rev-aux-Append*:
[[*same-level-path-rev-aux* cs as'; *same-level-path-rev-aux* (upd-rev-cs cs as') as]]
 \Rightarrow *same-level-path-rev-aux* cs (as@as')
<proof>

lemma *slpra-to-slpa*:
[[*same-level-path-rev-aux* cs as; upd-rev-cs cs as = []; n -as \rightarrow^* n';
valid-return-list cs n]]
 \Rightarrow *same-level-path-aux* [] as \wedge *same-level-path-aux* (upd-cs [] as) cs \wedge
 upd-cs (upd-cs [] as) cs = []
<proof>

Lemmas on paths with (-Entry-)

lemma *path-Entry-target* [dest]:
assumes n -as \rightarrow^* (-Entry-)
shows n = (-Entry-) **and** as = []
<proof>

lemma *Entry-sourcenode-hd*:

assumes $n -as \rightarrow^* n'$ **and** $(-Entry-) \in set(sourcenodes\ as)$
shows $n = (-Entry-)$ **and** $(-Entry-) \notin set(sourcenodes\ (tl\ as))$
 $\langle proof \rangle$

lemma *Entry-no-inner-return-path*:

assumes $(-Entry-) -as@[a] \rightarrow^* n$ **and** $\forall a \in set\ as.\ intra-kind(kind\ a)$
and $kind\ a = Q \leftrightarrow pf$
shows *False*
 $\langle proof \rangle$

lemma *vpra-no-spra*:

$\llbracket valid-path-rev-aux\ cs\ as;\ n -as \rightarrow^* n';\ valid-return-list\ cs\ n';\ cs \neq [];$
 $\forall xs\ ys.\ as = xs@ys \rightarrow (\neg same-level-path-rev-aux\ cs\ ys \vee upd-rev-cs\ cs\ ys \neq$
 $[]) \rrbracket$
 $\implies \exists a\ Q\ f.\ valid-edge\ a \wedge kind\ a = Q \leftrightarrow get-proc\ nf$
 $\langle proof \rangle$

lemma *valid-Entry-path-cases*:

assumes $(-Entry-) -as \rightarrow_{\sqrt{}}^* n$ **and** $as \neq []$
shows $(\exists a'\ as'. as = as'@[a'] \wedge intra-kind(kind\ a')) \vee$
 $(\exists a'\ as'\ Q\ r\ p\ fs.\ as = as'@[a'] \wedge kind\ a' = Q:r \hookrightarrow pfs) \vee$
 $(\exists as'\ as''\ n'. as = as'@as'' \wedge as'' \neq [] \wedge n' -as'' \rightarrow_{sl}^* n)$
 $\langle proof \rangle$

lemma *valid-Entry-path-ascending-path*:

assumes $(-Entry-) -as \rightarrow_{\sqrt{}}^* n$
obtains as' **where** $(-Entry-) -as' \rightarrow_{\sqrt{}}^* n$
and $set(sourcenodes\ as') \subseteq set(sourcenodes\ as)$
and $\forall a' \in set\ as'. intra-kind(kind\ a') \vee (\exists Q\ r\ p\ fs.\ kind\ a' = Q:r \hookrightarrow pfs)$
 $\langle proof \rangle$

end

end

theory *CFGExit* **imports** *CFG* **begin**

1.2.3 Adds an exit node to the abstract CFG

locale *CFGExit* = *CFG* *sourcenode* *targetnode* *kind* *valid-edge* *Entry*
get-proc *get-return-edges* *procs* *Main*

for *sourcenode* :: 'edge \Rightarrow 'node **and** *targetnode* :: 'edge \Rightarrow 'node
and *kind* :: 'edge \Rightarrow ('var,'val,'ret,'pname) edge-kind
and *valid-edge* :: 'edge \Rightarrow bool
and *Entry* :: 'node ('('Entry'-')) **and** *get-proc* :: 'node \Rightarrow 'pname
and *get-return-edges* :: 'edge \Rightarrow 'edge set
and *procs* :: ('pname \times 'var list \times 'var list) list **and** *Main* :: 'pname +
fixes *Exit*::'node ('('Exit'-'))
assumes *Exit-source* [*dest*]: $\llbracket \text{valid-edge } a; \text{sourcenode } a = (-\text{Exit-}) \rrbracket \Longrightarrow \text{False}$
and *get-proc-Exit*:*get-proc* (-Exit-) = *Main*
and *Exit-no-return-target*:
 $\llbracket \text{valid-edge } a; \text{kind } a = Q \leftrightarrow_{pf}; \text{targetnode } a = (-\text{Exit-}) \rrbracket \Longrightarrow \text{False}$
and *Entry-Exit-edge*: $\exists a. \text{valid-edge } a \wedge \text{sourcenode } a = (-\text{Entry-}) \wedge$
 $\text{targetnode } a = (-\text{Exit-}) \wedge \text{kind } a = (\lambda s. \text{False})_{\checkmark}$

begin

lemma *Entry-noteq-Exit* [*dest*]:
assumes *eq*:(-Entry-) = (-Exit-) **shows** *False*
<proof>

lemma *Exit-noteq-Entry* [*dest*]:(-Exit-) = (-Entry-) \Longrightarrow *False*
<proof>

lemma [*simp*]: *valid-node* (-Entry-)
<proof>

lemma [*simp*]: *valid-node* (-Exit-)
<proof>

Definition of method-exit

definition *method-exit* :: 'node \Rightarrow bool
where *method-exit* *n* \equiv *n* = (-Exit-) \vee
 $(\exists a Q p f. n = \text{sourcenode } a \wedge \text{valid-edge } a \wedge \text{kind } a = Q \leftrightarrow_{pf})$

lemma *method-exit-cases*:
 $\llbracket \text{method-exit } n; n = (-\text{Exit-}) \rrbracket \Longrightarrow P;$
 $\wedge a Q p f. \llbracket n = \text{sourcenode } a; \text{valid-edge } a; \text{kind } a = Q \leftrightarrow_{pf} \rrbracket \Longrightarrow P \Longrightarrow P$
<proof>

lemma *method-exit-inner-path*:
assumes *method-exit* *n* **and** *n* -*as*- \rightarrow_i^* *n'* **shows** *as* = []
<proof>

Definition of *inner-node*

definition *inner-node* :: 'node \Rightarrow bool

where *inner-node-def*:

inner-node $n \equiv$ *valid-node* $n \wedge n \neq (-\text{Entry-}) \wedge n \neq (-\text{Exit-})$

lemma *inner-is-valid*:

inner-node $n \implies$ *valid-node* n

\langle *proof* \rangle

lemma [*dest*]:

inner-node $(-\text{Entry-}) \implies$ *False*

\langle *proof* \rangle

lemma [*dest*]:

inner-node $(-\text{Exit-}) \implies$ *False*

\langle *proof* \rangle

lemma [*simp*]: \llbracket *valid-edge* a ; *targetnode* $a \neq (-\text{Exit-})$ \rrbracket

\implies *inner-node* (*targetnode* a)

\langle *proof* \rangle

lemma [*simp*]: \llbracket *valid-edge* a ; *sourcenode* $a \neq (-\text{Entry-})$ \rrbracket

\implies *inner-node* (*sourcenode* a)

\langle *proof* \rangle

lemma *valid-node-cases* [*consumes 1, case-names Entry Exit inner*]:

\llbracket *valid-node* n ; $n = (-\text{Entry-}) \implies Q$; $n = (-\text{Exit-}) \implies Q$;

inner-node $n \implies Q$ $\rrbracket \implies Q$

\langle *proof* \rangle

Lemmas on paths with $(-\text{Exit-})$

lemma *path-Exit-source*:

$\llbracket n -as \rightarrow^* n'; n = (-\text{Exit-}) \rrbracket \implies n' = (-\text{Exit-}) \wedge as = []$

\langle *proof* \rangle

lemma [*dest*]: $(-\text{Exit-}) -as \rightarrow^* n' \implies n' = (-\text{Exit-}) \wedge as = []$

\langle *proof* \rangle

lemma *Exit-no-sourcenode*[*dest*]:

assumes *isin*: $(-\text{Exit-}) \in$ *set* (*sourcenodes* as) **and** *path*: $n -as \rightarrow^* n'$

shows *False*

\langle *proof* \rangle

lemma *vpa-no-slpa*:

\llbracket *valid-path-aux* cs as ; $n -as \rightarrow^* n'$; *valid-call-list* cs n ; $cs \neq []$ \rrbracket

$\forall xs\ ys.\ as = xs@ys \longrightarrow (\neg \text{same-level-path-aux}\ cs\ xs \vee \text{upd-cs}\ cs\ xs \neq [])$
 $\implies \exists a\ Q\ r\ fs.\ \text{valid-edge}\ a \wedge \text{kind}\ a = Q:r \leftrightarrow_{\text{get-proc}} n'fs$
 <proof>

lemma *valid-Exit-path-cases*:

assumes $n - as \rightarrow_{\sqrt{*}} (-Exit-)$ **and** $as \neq []$
shows $(\exists a'\ as'. as = a'\#as' \wedge \text{intra-kind}(\text{kind}\ a')) \vee$
 $(\exists a'\ as'\ Q\ p\ f.\ as = a'\#as' \wedge \text{kind}\ a' = Q \leftrightarrow_{pf}) \vee$
 $(\exists as'\ as''\ n'. as = as'@as'' \wedge as' \neq [] \wedge n - as' \rightarrow_{st^*} n')$
 <proof>

lemma *valid-Exit-path-descending-path*:

assumes $n - as \rightarrow_{\sqrt{*}} (-Exit-)$
obtains as' **where** $n - as' \rightarrow_{\sqrt{*}} (-Exit-)$
and $\text{set}(\text{sourcenodes}\ as') \subseteq \text{set}(\text{sourcenodes}\ as)$
and $\forall a' \in \text{set}\ as'. \text{intra-kind}(\text{kind}\ a') \vee (\exists Q\ f\ p.\ \text{kind}\ a' = Q \leftrightarrow_{pf})$
 <proof>

lemma *valid-Exit-path-intra-path*:

assumes $n - as \rightarrow_{\sqrt{*}} (-Exit-)$
obtains as' **per** x **where** $n - as' \rightarrow_{t^*} x$ **and** *method-exit* x
and $\text{set}(\text{sourcenodes}\ as') \subseteq \text{set}(\text{sourcenodes}\ as)$
 <proof>

end

end

1.3 CFG well-formedness

theory *CFG-wf* **imports** *CFG* **begin**

locale *CFG-wf* = *CFG* *sourcenode* *targetnode* *kind* *valid-edge* *Entry*
get-proc *get-return-edges* *procs* *Main*
for *sourcenode* :: 'edge \Rightarrow 'node **and** *targetnode* :: 'edge \Rightarrow 'node
and *kind* :: 'edge \Rightarrow ('var,'val,'ret,'pname) *edge-kind*
and *valid-edge* :: 'edge \Rightarrow bool
and *Entry* :: 'node ('('Entry'-')) **and** *get-proc* :: 'node \Rightarrow 'pname
and *get-return-edges* :: 'edge \Rightarrow 'edge set
and *procs* :: ('pname \times 'var list \times 'var list) list **and** *Main* :: 'pname +
fixes *Def*::'node \Rightarrow 'var set
fixes *Use*::'node \Rightarrow 'var set
fixes *ParamDefs*::'node \Rightarrow 'var list
fixes *ParamUses*::'node \Rightarrow 'var set list

assumes $Entry\text{-}empty:Def (-Entry) = \{\} \wedge Use (-Entry) = \{\}$
and $ParamUses\text{-}call\text{-}source\text{-}length:$
 $\llbracket valid\text{-}edge\ a; kind\ a = Q:r\hookrightarrow_pfs; (p,ins,outs) \in set\ procs \rrbracket$
 $\implies length(ParamUses\ (sourcename\ a)) = length\ ins$
and $distinct\text{-}ParamDefs:valid\text{-}edge\ a \implies distinct\ (ParamDefs\ (targetnode\ a))$
and $ParamDefs\text{-}return\text{-}target\text{-}length:$
 $\llbracket valid\text{-}edge\ a; kind\ a = Q'\hookleftarrow_pf'; (p,ins,outs) \in set\ procs \rrbracket$
 $\implies length(ParamDefs\ (targetnode\ a)) = length\ outs$
and $ParamDefs\text{-}in\text{-}Def:$
 $\llbracket valid\text{-}node\ n; V \in set\ (ParamDefs\ n) \rrbracket \implies V \in Def\ n$
and $ins\text{-}in\text{-}Def:$
 $\llbracket valid\text{-}edge\ a; kind\ a = Q:r\hookrightarrow_pfs; (p,ins,outs) \in set\ procs; V \in set\ ins \rrbracket$
 $\implies V \in Def\ (targetnode\ a)$
and $call\text{-}source\text{-}Def\text{-}empty:$
 $\llbracket valid\text{-}edge\ a; kind\ a = Q:r\hookrightarrow_pfs \rrbracket \implies Def\ (sourcename\ a) = \{\}$
and $ParamUses\text{-}in\text{-}Use:$
 $\llbracket valid\text{-}node\ n; V \in Union\ (set\ (ParamUses\ n)) \rrbracket \implies V \in Use\ n$
and $outs\text{-}in\text{-}Use:$
 $\llbracket valid\text{-}edge\ a; kind\ a = Q\hookleftarrow_pf; (p,ins,outs) \in set\ procs; V \in set\ outs \rrbracket$
 $\implies V \in Use\ (sourcename\ a)$
and $CFG\text{-}intra\text{-}edge\text{-}no\text{-}Def\text{-}equal:$
 $\llbracket valid\text{-}edge\ a; V \notin Def\ (sourcename\ a); intra\text{-}kind\ (kind\ a); pred\ (kind\ a)\ s \rrbracket$
 $\implies state\text{-}val\ (transfer\ (kind\ a)\ s)\ V = state\text{-}val\ s\ V$
and $CFG\text{-}intra\text{-}edge\text{-}transfer\text{-}uses\text{-}only\text{-}Use:$
 $\llbracket valid\text{-}edge\ a; \forall V \in Use\ (sourcename\ a). state\text{-}val\ s\ V = state\text{-}val\ s'\ V;$
 $intra\text{-}kind\ (kind\ a); pred\ (kind\ a)\ s; pred\ (kind\ a)\ s' \rrbracket$
 $\implies \forall V \in Def\ (sourcename\ a). state\text{-}val\ (transfer\ (kind\ a)\ s)\ V =$
 $state\text{-}val\ (transfer\ (kind\ a)\ s')\ V$
and $CFG\text{-}edge\text{-}Uses\text{-}pred\text{-}equal:$
 $\llbracket valid\text{-}edge\ a; pred\ (kind\ a)\ s; snd\ (hd\ s) = snd\ (hd\ s');$
 $\forall V \in Use\ (sourcename\ a). state\text{-}val\ s\ V = state\text{-}val\ s'\ V; length\ s = length\ s' \rrbracket$
 $\implies pred\ (kind\ a)\ s'$
and $CFG\text{-}call\text{-}edge\text{-}length:$
 $\llbracket valid\text{-}edge\ a; kind\ a = Q:r\hookrightarrow_pfs; (p,ins,outs) \in set\ procs \rrbracket$
 $\implies length\ fs = length\ ins$
and $CFG\text{-}call\text{-}determ:$
 $\llbracket valid\text{-}edge\ a; kind\ a = Q:r\hookrightarrow_pfs; valid\text{-}edge\ a'; kind\ a' = Q':r'\hookrightarrow_{p'}fs';$
 $sourcename\ a = sourcename\ a'; pred\ (kind\ a)\ s; pred\ (kind\ a')\ s \rrbracket$
 $\implies a = a'$
and $CFG\text{-}call\text{-}edge\text{-}params:$
 $\llbracket valid\text{-}edge\ a; kind\ a = Q:r\hookrightarrow_pfs; i < length\ ins;$
 $(p,ins,outs) \in set\ procs; pred\ (kind\ a)\ s; pred\ (kind\ a)\ s';$
 $\forall V \in (ParamUses\ (sourcename\ a))!i. state\text{-}val\ s\ V = state\text{-}val\ s'\ V \rrbracket$
 $\implies (params\ fs\ (fst\ (hd\ s)))!i = (params\ fs\ (fst\ (hd\ s')))!i$
and $CFG\text{-}return\text{-}edge\text{-}fun:$
 $\llbracket valid\text{-}edge\ a; kind\ a = Q'\hookleftarrow_pf'; (p,ins,outs) \in set\ procs \rrbracket$
 $\implies f'\ vmap\ vmap' = vmap'(ParamDefs\ (targetnode\ a)\ [:=]\ map\ vmap\ outs)$
and $deterministic:\llbracket valid\text{-}edge\ a; valid\text{-}edge\ a'; sourcename\ a = sourcename\ a';$
 $targetnode\ a \neq targetnode\ a'; intra\text{-}kind\ (kind\ a); intra\text{-}kind\ (kind\ a') \rrbracket$

$$\implies \exists Q Q'. \text{kind } a = (Q)_{\surd} \wedge \text{kind } a' = (Q')_{\surd} \wedge \\ (\forall s. (Q s \longrightarrow \neg Q' s) \wedge (Q' s \longrightarrow \neg Q s))$$

begin

lemma *CFG-equal-Use-equal-call*:

assumes *valid-edge* a **and** $\text{kind } a = Q:r \hookrightarrow_p fs$ **and** *valid-edge* a'
and $\text{kind } a' = Q':r' \hookrightarrow_{p'} fs'$ **and** $\text{sourcenode } a = \text{sourcenode } a'$
and $\text{pred } (\text{kind } a) s$ **and** $\text{pred } (\text{kind } a') s'$
and $\text{snd } (\text{hd } s) = \text{snd } (\text{hd } s')$ **and** $\text{length } s = \text{length } s'$
and $\forall V \in \text{Use } (\text{sourcenode } a). \text{state-val } s V = \text{state-val } s' V$
shows $a = a'$

<proof>

lemma *CFG-call-edge-param-in*:

assumes *valid-edge* a **and** $\text{kind } a = Q:r \hookrightarrow_p fs$ **and** $i < \text{length } ins$
and $(p, ins, outs) \in \text{set procs}$ **and** $\text{pred } (\text{kind } a) s$ **and** $\text{pred } (\text{kind } a) s'$
and $\forall V \in (\text{ParamUses } (\text{sourcenode } a))!i. \text{state-val } s V = \text{state-val } s' V$
shows $\text{state-val } (\text{transfer } (\text{kind } a) s) (ins!i) =$
 $\text{state-val } (\text{transfer } (\text{kind } a) s') (ins!i)$

<proof>

lemma *CFG-call-edge-no-param*:

assumes *valid-edge* a **and** $\text{kind } a = Q:r \hookrightarrow_p fs$ **and** $V \notin \text{set } ins$
and $(p, ins, outs) \in \text{set procs}$ **and** $\text{pred } (\text{kind } a) s$
shows $\text{state-val } (\text{transfer } (\text{kind } a) s) V = \text{None}$

<proof>

lemma *CFG-return-edge-param-out*:

assumes *valid-edge* a **and** $\text{kind } a = Q \hookrightarrow_p f$ **and** $i < \text{length } outs$
and $(p, ins, outs) \in \text{set procs}$ **and** $\text{state-val } s (outs!i) = \text{state-val } s' (outs!i)$
and $s = cf \# cf_x \# cfs$ **and** $s' = cf' \# cf'_x \# cfs'$
shows $\text{state-val } (\text{transfer } (\text{kind } a) s) ((\text{ParamDefs } (\text{targetnode } a))!i) =$
 $\text{state-val } (\text{transfer } (\text{kind } a) s') ((\text{ParamDefs } (\text{targetnode } a))!i)$

<proof>

lemma *CFG-slp-no-Def-equal*:

assumes $n - as \rightarrow_{sl} n'$ **and** *valid-edge* a **and** $a' \in \text{get-return-edges } a$
and $V \notin \text{set } (\text{ParamDefs } (\text{targetnode } a'))$ **and** $\text{preds } (\text{kinds } (a \# as @ [a'])) s$
shows $\text{state-val } (\text{transfers } (\text{kinds } (a \# as @ [a'])) s) V = \text{state-val } s V$

<proof>

lemma [dest!]: $V \in Use (-Entry-) \implies False$
 <proof>

lemma [dest!]: $V \in Def (-Entry-) \implies False$
 <proof>

lemma *CFG-intra-path-no-Def-equal*:
assumes $n -as \rightarrow_{\iota}^* n'$ **and** $\forall n \in set (sourcenodes\ as)$. $V \notin Def\ n$
and $preds\ (kinds\ as)\ s$
shows $state-val\ (transfers\ (kinds\ as)\ s)\ V = state-val\ s\ V$
 <proof>

lemma *slpa-preds*:
 [[*same-level-path-aux* $cs\ as$; $s = cfsx@cf\ \#cfs$; $s' = cfsx@cf\ \#cfs'$;
 $length\ cfs = length\ cfs'$; $\forall a \in set\ as$. *valid-edge* a ; $length\ cs = length\ cfsx$;
 $preds\ (kinds\ as)\ s$]]
 $\implies preds\ (kinds\ as)\ s'$
 <proof>

lemma *slp-preds*:
assumes $n -as \rightarrow_{sl}^* n'$ **and** $preds\ (kinds\ as)\ (cf\ \#cfs)$
and $length\ cfs = length\ cfs'$
shows $preds\ (kinds\ as)\ (cf\ \#cfs')$
 <proof>
end

end
theory *CFGExit-wf* **imports** *CFGExit CFG-wf* **begin**

1.3.1 New well-formedness lemmas using (-Exit-)

locale *CFGExit-wf* = *CFGExit sourcenode targetnode kind valid-edge Entry*
get-proc get-return-edges procs Main Exit +
CFG-wf sourcenode targetnode kind valid-edge Entry
get-proc get-return-edges procs Main Def Use ParamDefs ParamUses
for *sourcenode* :: 'edge \Rightarrow 'node **and** *targetnode* :: 'edge \Rightarrow 'node
and *kind* :: 'edge \Rightarrow ('var,'val,'ret,'pname) edge-kind
and *valid-edge* :: 'edge \Rightarrow bool
and *Entry* :: 'node (('(-Entry'-)) **and** *get-proc* :: 'node \Rightarrow 'pname
and *get-return-edges* :: 'edge \Rightarrow 'edge set
and *procs* :: ('pname \times 'var list \times 'var list) list **and** *Main* :: 'pname
and *Exit*::'node (('(-Exit'-))
and *Def* :: 'node \Rightarrow 'var set **and** *Use* :: 'node \Rightarrow 'var set
and *ParamDefs* :: 'node \Rightarrow 'var list

```

and ParamUses :: 'node  $\Rightarrow$  'var set list +
assumes Exit-empty:Def (-Exit-) = {}  $\wedge$  Use (-Exit-) = {}

begin

lemma Exit-Use-empty [dest!]:  $V \in Use (-Exit-) \Rightarrow False$ 
<proof>

lemma Exit-Def-empty [dest!]:  $V \in Def (-Exit-) \Rightarrow False$ 
<proof>

end

end

```

1.4 CFG and semantics conform

```

theory SemanticsCFG imports CFG begin

```

```

locale CFG-semantics-wf = CFG sourcenode targetnode kind valid-edge Entry
  get-proc get-return-edges procs Main
for sourcenode :: 'edge  $\Rightarrow$  'node and targetnode :: 'edge  $\Rightarrow$  'node
and kind :: 'edge  $\Rightarrow$  ('var,'val,'ret,'pname) edge-kind
and valid-edge :: 'edge  $\Rightarrow$  bool
and Entry :: 'node ('('Entry'-')) and get-proc :: 'node  $\Rightarrow$  'pname
and get-return-edges :: 'edge  $\Rightarrow$  'edge set
and procs :: ('pname  $\times$  'var list  $\times$  'var list) list and Main :: 'pname +
fixes sem::'com  $\Rightarrow$  ('var  $\rightarrow$  'val) list  $\Rightarrow$  'com  $\Rightarrow$  ('var  $\rightarrow$  'val) list  $\Rightarrow$  bool
  (((1<-,->)  $\Rightarrow$  / (1<-,->)) [0,0,0,0] 81)
fixes identifies::'node  $\Rightarrow$  'com  $\Rightarrow$  bool (-  $\triangleq$  - [51,0] 80)
assumes fundamental-property:
  [[n  $\triangleq$  c; <c,[cf]>  $\Rightarrow$  <c',s^>]]  $\Rightarrow$ 
   $\exists n' \text{ as. } n - \text{as} \rightarrow_{\sqrt{*}} n' \wedge n' \triangleq c' \wedge \text{preds } (\text{kinds as}) [(cf, \text{undefined})] \wedge$ 
  transfers (kinds as) [(cf, \text{undefined})] = cfs'  $\wedge$  map fst cfs' = s'

```

```

end

```

1.5 Return and their corresponding call nodes

```

theory ReturnAndCallNodes imports CFG begin

```

```

context CFG begin

```

1.5.1 Defining return-node

```

definition return-node :: 'node  $\Rightarrow$  bool
  where return-node n  $\equiv \exists a a'. \text{valid-edge } a \wedge n = \text{targetnode } a \wedge$ 

```

$valid-edge\ a' \wedge a \in get-return-edges\ a'$

lemma *return-node-determines-call-node*:

assumes *return-node* n

shows $\exists!n'. \exists a\ a'. valid-edge\ a \wedge n' = sourcenode\ a \wedge valid-edge\ a' \wedge a' \in get-return-edges\ a \wedge n = targetnode\ a'$

<proof>

lemma *return-node-THE-call-node*:

$\llbracket return-node\ n; valid-edge\ a; valid-edge\ a'; a' \in get-return-edges\ a; n = targetnode\ a' \rrbracket$

$\implies (THE\ n'. \exists a\ a'. valid-edge\ a \wedge n' = sourcenode\ a \wedge valid-edge\ a' \wedge a' \in get-return-edges\ a \wedge n = targetnode\ a') = sourcenode\ a$

<proof>

1.5.2 Defining call nodes belonging to a certain *return-node*

definition *call-of-return-node* :: $'node \Rightarrow 'node \Rightarrow bool$

where *call-of-return-node* $n\ n' \equiv \exists a\ a'. return-node\ n \wedge$

$valid-edge\ a \wedge n' = sourcenode\ a \wedge valid-edge\ a' \wedge$

$a' \in get-return-edges\ a \wedge n = targetnode\ a'$

lemma *return-node-call-of-return-node*:

$return-node\ n \implies \exists!n'. call-of-return-node\ n\ n'$

<proof>

lemma *call-of-return-nodes-det* [*dest*]:

assumes *call-of-return-node* $n\ n'$ **and** *call-of-return-node* $n\ n''$

shows $n' = n''$

<proof>

lemma *get-return-edges-call-of-return-nodes*:

$\llbracket valid-call-list\ cs\ m; valid-return-list\ rs\ m; \forall i < length\ rs. rs!i \in get-return-edges\ (cs!i); length\ rs = length\ cs \rrbracket$

$\implies \forall i < length\ cs. call-of-return-node\ (targetnodes\ rs!i)\ (sourcenode\ (cs!i))$

<proof>

end

end

1.6 Observable Sets of Nodes

theory *Observable* **imports** *ReturnAndCallNodes* **begin**

context *CFG* **begin**

1.6.1 Intraprocedural observable sets

inductive-set *obs-intra* :: 'node \Rightarrow 'node set \Rightarrow 'node set

for $n::$ 'node **and** $S::$ 'node set

where *obs-intra-elem*:

$\llbracket n -as \rightarrow_{\iota}^* n'; \forall nx \in \text{set}(\text{sourcenodes } as). nx \notin S; n' \in S \rrbracket \implies n' \in \text{obs-intra } n S$

lemma *obs-intraE*:

assumes $n' \in \text{obs-intra } n S$

obtains as **where** $n -as \rightarrow_{\iota}^* n'$ **and** $\forall nx \in \text{set}(\text{sourcenodes } as). nx \notin S$ **and** $n' \in S$

$\langle \text{proof} \rangle$

lemma *n-in-obs-intra*:

assumes *valid-node* n **and** $n \in S$ **shows** $\text{obs-intra } n S = \{n\}$

$\langle \text{proof} \rangle$

lemma *in-obs-intra-valid*:

assumes $n' \in \text{obs-intra } n S$ **shows** *valid-node* n **and** *valid-node* n'

$\langle \text{proof} \rangle$

lemma *edge-obs-intra-subset*:

assumes *valid-edge* a **and** *intra-kind* (*kind* a) **and** *sourcenode* $a \notin S$

shows $\text{obs-intra } (\text{targetnode } a) S \subseteq \text{obs-intra } (\text{sourcenode } a) S$

$\langle \text{proof} \rangle$

lemma *path-obs-intra-subset*:

assumes $n -as \rightarrow_{\iota}^* n'$ **and** $\forall n' \in \text{set}(\text{sourcenodes } as). n' \notin S$

shows $\text{obs-intra } n' S \subseteq \text{obs-intra } n S$

$\langle \text{proof} \rangle$

lemma *path-ex-obs-intra*:

assumes $n -as \rightarrow_{\iota}^* n'$ **and** $n' \in S$

obtains m **where** $m \in \text{obs-intra } n S$

$\langle \text{proof} \rangle$

1.6.2 Interprocedural observable sets restricted to the slice

fun *obs* :: 'node list \Rightarrow 'node set \Rightarrow 'node list set
where *obs* [] S = {}
| *obs* ($n\#ns$) S = (let $S' = \text{obs-intra } n \ S$ in
(if ($S' = \{\}$) \vee ($\exists n' \in \text{set } ns. \exists nx. \text{call-of-return-node } n' \ nx \wedge nx \notin S$))
then *obs* $ns \ S$ else ($\lambda nx. nx\#ns$) ' S')

lemma *obsI*:

assumes $n' \in \text{obs-intra } n \ S$
and $\forall nx \in \text{set } nsx'. \exists nx'. \text{call-of-return-node } nx \ nx' \wedge nx' \in S$
shows [$ns = nsx@n\#nsx'$; $\forall xs \ x \ xs'. nsx = xs@x\#xs' \wedge \text{obs-intra } x \ S \neq \{\}$
 $\rightarrow (\exists x'' \in \text{set } (xs'@[n]). \exists nx. \text{call-of-return-node } x'' \ nx \wedge nx \notin S)$]
 $\Rightarrow n'\#nsx' \in \text{obs } ns \ S$
<proof>

lemma *obsE* [*consumes* 2]:

assumes $ns' \in \text{obs } ns \ S$ **and** $\forall n \in \text{set } (tl \ ns). \text{return-node } n$
obtains $nsx \ n \ nsx' \ n'$ **where** $ns = nsx@n\#nsx'$ **and** $ns' = n'\#nsx'$
and $n' \in \text{obs-intra } n \ S$
and $\forall nx \in \text{set } nsx'. \exists nx'. \text{call-of-return-node } nx \ nx' \wedge nx' \in S$
and $\forall xs \ x \ xs'. nsx = xs@x\#xs' \wedge \text{obs-intra } x \ S \neq \{\}$
 $\rightarrow (\exists x'' \in \text{set } (xs'@[n]). \exists nx. \text{call-of-return-node } x'' \ nx \wedge nx \notin S)$
<proof>

lemma *obs-split-det*:

assumes $xs@x\#xs' = ys@y\#ys'$
and $\text{obs-intra } x \ S \neq \{\}$
and $\forall x' \in \text{set } xs'. \exists x''. \text{call-of-return-node } x' \ x'' \wedge x'' \in S$
and $\forall zs \ z \ zs'. xs = zs@z\#zs' \wedge \text{obs-intra } z \ S \neq \{\}$
 $\rightarrow (\exists z'' \in \text{set } (zs'@[x]). \exists nx. \text{call-of-return-node } z'' \ nx \wedge nx \notin S)$
and $\text{obs-intra } y \ S \neq \{\}$
and $\forall y' \in \text{set } ys'. \exists y''. \text{call-of-return-node } y' \ y'' \wedge y'' \in S$
and $\forall zs \ z \ zs'. ys = zs@z\#zs' \wedge \text{obs-intra } z \ S \neq \{\}$
 $\rightarrow (\exists z'' \in \text{set } (zs'@[y]). \exists ny. \text{call-of-return-node } z'' \ ny \wedge ny \notin S)$
shows $xs = ys \wedge x = y \wedge xs' = ys'$
<proof>

lemma *in-obs-valid*:

assumes $ns' \in \text{obs } ns \ S$ **and** $\forall n \in \text{set } ns. \text{valid-node } n$
shows $\forall n \in \text{set } ns'. \text{valid-node } n$
<proof>

end

end

1.7 Postdomination

theory *Postdomination* **imports** *CFGExit* **begin**

For static interprocedural slicing, we only consider standard control dependence, hence we only need standard postdomination.

locale *Postdomination* = *CFGExit* *sourcenode* *targetnode* *kind* *valid-edge* *Entry*

get-proc *get-return-edges* *procs* *Main* *Exit*

for *sourcenode* :: 'edge \Rightarrow 'node **and** *targetnode* :: 'edge \Rightarrow 'node

and *kind* :: 'edge \Rightarrow ('var,'val,'ret,'pname) *edge-kind*

and *valid-edge* :: 'edge \Rightarrow bool

and *Entry* :: 'node ('('Entry-')) **and** *get-proc* :: 'node \Rightarrow 'pname

and *get-return-edges* :: 'edge \Rightarrow 'edge set

and *procs* :: ('pname \times 'var list \times 'var list) list **and** *Main* :: 'pname

and *Exit*::'node ('('Exit-')) +

assumes *Entry-path:valid-node* $n \implies \exists as. (-Entry-) -as \rightarrow_{\sqrt{*}} n$

and *Exit-path:valid-node* $n \implies \exists as. n -as \rightarrow_{\sqrt{*}} (-Exit-)$

and *method-exit-unique*:

$\llbracket \text{method-exit } n; \text{method-exit } n'; \text{get-proc } n = \text{get-proc } n' \rrbracket \implies n = n'$

begin

lemma *get-return-edges-unique*:

assumes *valid-edge* a **and** $a' \in \text{get-return-edges } a$ **and** $a'' \in \text{get-return-edges } a$

shows $a' = a''$

<proof>

definition *postdominate* :: 'node \Rightarrow 'node \Rightarrow bool (- *postdominates* - [51,0])

where *postdominate-def*: n' *postdominates* $n \equiv$

$(\text{valid-node } n \wedge \text{valid-node } n' \wedge$

$(\forall as \text{ pex. } (n -as \rightarrow_{\iota^*} \text{pex} \wedge \text{method-exit } \text{pex}) \longrightarrow n' \in \text{set } (\text{sourcenodes } as)))$

lemma *postdominate-implies-inner-path*:

assumes n' *postdominates* n

obtains as **where** $n -as \rightarrow_{\iota^*} n'$ **and** $n' \notin \text{set } (\text{sourcenodes } as)$

<proof>

lemma *postdominate-variant*:

assumes n' *postdominates* n

shows $\forall as. n -as \rightarrow_{\sqrt{*}} (-Exit-) \longrightarrow n' \in \text{set } (\text{sourcenodes } as)$

<proof>

lemma *postdominate-refl*:

assumes *valid-node n* **and** \neg *method-exit n* **shows** *n* *postdominates n*
<proof>

lemma *postdominate-trans*:

assumes *n'' postdominates n* **and** *n' postdominates n''*
shows *n' postdominates n*
<proof>

lemma *postdominate-antisym*:

assumes *n' postdominates n* **and** *n postdominates n'*
shows $n = n'$
<proof>

lemma *postdominate-path-branch*:

assumes $n -as \rightarrow^* n''$ **and** *n' postdominates n''* **and** \neg *n' postdominates n*
obtains *a as' as''* **where** $as = as' @ a \# as''$ **and** *valid-edge a*
and \neg *n' postdominates (sourcenode a)* **and** *n' postdominates (targetnode a)*
<proof>

lemma *Exit-no-postdominator*:

assumes $(-Exit-) postdominates n$ **shows** *False*
<proof>

lemma *postdominate-inner-path-targetnode*:

assumes *n' postdominates n* **and** $n -as \rightarrow_i^* n''$ **and** $n' \notin \text{set}(\text{sourcenodes } as)$
shows *n' postdominates n''*
<proof>

lemma *not-postdominate-source-not-postdominate-target*:

assumes \neg *n postdominates (sourcenode a)*
and *valid-node n* **and** *valid-edge a* **and** *intra-kind (kind a)*
obtains *ax* **where** $\text{sourcenode } a = \text{sourcenode } ax$ **and** *valid-edge ax*
and \neg *n postdominates targetnode ax*
<proof>

lemma *inner-node-Exit-edge*:

assumes *inner-node n*
obtains *a* **where** *valid-edge a* **and** *intra-kind (kind a)*

and *inner-node* (*sourcenode* *a*) **and** *targetnode* *a* = (-Exit-)
 ⟨*proof*⟩

lemma *inner-node-Entry-edge*:
assumes *inner-node* *n*
obtains *a* **where** *valid-edge* *a* **and** *intra-kind* (*kind* *a*)
and *inner-node* (*targetnode* *a*) **and** *sourcenode* *a* = (-Entry-)
 ⟨*proof*⟩

lemma *intra-path-to-matching-method-exit*:
assumes *method-exit* *n'* **and** *get-proc* *n* = *get-proc* *n'* **and** *valid-node* *n*
obtains *as* **where** *n* -*as*->_{t,*} *n'*
 ⟨*proof*⟩

end

end

1.8 SDG

theory *SDG* **imports** *CFGExit-wf* *Postdomination* **begin**

1.8.1 The nodes of the SDG

datatype *'node* *SDG-node* =
 CFG-node *'node*
 | *Formal-in* *'node* × *nat*
 | *Formal-out* *'node* × *nat*
 | *Actual-in* *'node* × *nat*
 | *Actual-out* *'node* × *nat*

fun *parent-node* :: *'node* *SDG-node* ⇒ *'node*
where *parent-node* (*CFG-node* *n*) = *n*
 | *parent-node* (*Formal-in* (*m*,*x*)) = *m*
 | *parent-node* (*Formal-out* (*m*,*x*)) = *m*
 | *parent-node* (*Actual-in* (*m*,*x*)) = *m*
 | *parent-node* (*Actual-out* (*m*,*x*)) = *m*

locale *SDG* = *CFGExit-wf* *sourcenode* *targetnode* *kind* *valid-edge* *Entry*
 get-proc *get-return-edges* *procs* *Main* *Exit* *Def* *Use* *ParamDefs* *ParamUses* +
 Postdomination *sourcenode* *targetnode* *kind* *valid-edge* *Entry*
 get-proc *get-return-edges* *procs* *Main* *Exit*
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ (*'var*,*'val*,*'ret*,*'pname*) *edge-kind*
and *valid-edge* :: *'edge* ⇒ *bool*

```

and Entry :: 'node ('(-Entry'-)) and get-proc :: 'node  $\Rightarrow$  'pname
and get-return-edges :: 'edge  $\Rightarrow$  'edge set
and procs :: ('pname  $\times$  'var list  $\times$  'var list) list and Main :: 'pname
and Exit::'node ('(-Exit'-))
and Def :: 'node  $\Rightarrow$  'var set and Use :: 'node  $\Rightarrow$  'var set
and ParamDefs :: 'node  $\Rightarrow$  'var list and ParamUses :: 'node  $\Rightarrow$  'var set list

```

begin

```

fun valid-SDG-node :: 'node SDG-node  $\Rightarrow$  bool
  where valid-SDG-node (CFG-node n)  $\longleftrightarrow$  valid-node n
    | valid-SDG-node (Formal-in (m,x))  $\longleftrightarrow$ 
      ( $\exists a Q r p fs ins outs.$  valid-edge a  $\wedge$  (kind a = Q:r $\hookrightarrow$ pfs)  $\wedge$  targetnode a = m  $\wedge$ 
        (p,ins,outs)  $\in$  set procs  $\wedge$  x < length ins)
    | valid-SDG-node (Formal-out (m,x))  $\longleftrightarrow$ 
      ( $\exists a Q p f ins outs.$  valid-edge a  $\wedge$  (kind a = Q $\hookleftarrow$ pf)  $\wedge$  sourcenode a = m  $\wedge$ 
        (p,ins,outs)  $\in$  set procs  $\wedge$  x < length outs)
    | valid-SDG-node (Actual-in (m,x))  $\longleftrightarrow$ 
      ( $\exists a Q r p fs ins outs.$  valid-edge a  $\wedge$  (kind a = Q:r $\hookrightarrow$ pfs)  $\wedge$  sourcenode a = m
       $\wedge$ 
        (p,ins,outs)  $\in$  set procs  $\wedge$  x < length ins)
    | valid-SDG-node (Actual-out (m,x))  $\longleftrightarrow$ 
      ( $\exists a Q p f ins outs.$  valid-edge a  $\wedge$  (kind a = Q $\hookleftarrow$ pf)  $\wedge$  targetnode a = m  $\wedge$ 
        (p,ins,outs)  $\in$  set procs  $\wedge$  x < length outs)

```

lemma valid-SDG-CFG-node:
 valid-SDG-node n \implies valid-node (parent-node n)
 <proof>

lemma Formal-in-parent-det:
assumes valid-SDG-node (Formal-in (m,x)) **and** valid-SDG-node (Formal-in (m',x'))
and get-proc m = get-proc m'
shows m = m'
 <proof>

lemma valid-SDG-node-parent-Entry:
assumes valid-SDG-node n **and** parent-node n = (-Entry-)
shows n = CFG-node (-Entry-)
 <proof>

lemma valid-SDG-node-parent-Exit:
assumes valid-SDG-node n **and** parent-node n = (-Exit-)
shows n = CFG-node (-Exit-)

$\langle proof \rangle$

1.8.2 Data dependence

inductive $SDG\text{-}Use :: 'var \Rightarrow 'node\ SDG\text{-}node \Rightarrow bool\ (- \in Use_{SDG}\ -)$

where $CFG\text{-}Use\text{-}SDG\text{-}Use$:

$\llbracket valid\text{-}node\ m; V \in Use\ m; n = CFG\text{-}node\ m \rrbracket \Longrightarrow V \in Use_{SDG}\ n$

| $Actual\text{-}in\text{-}SDG\text{-}Use$:

$\llbracket valid\text{-}SDG\text{-}node\ n; n = Actual\text{-}in\ (m,x); V \in (ParamUses\ m)!x \rrbracket \Longrightarrow V \in Use_{SDG}\ n$

| $Formal\text{-}out\text{-}SDG\text{-}Use$:

$\llbracket valid\text{-}SDG\text{-}node\ n; n = Formal\text{-}out\ (m,x); get\text{-}proc\ m = p; (p,ins,outs) \in set\ procs;$

$V = outs!x \rrbracket \Longrightarrow V \in Use_{SDG}\ n$

abbreviation $notin\text{-}SDG\text{-}Use :: 'var \Rightarrow 'node\ SDG\text{-}node \Rightarrow bool\ (- \notin Use_{SDG}\ -)$

where $V \notin Use_{SDG}\ n \equiv \neg V \in Use_{SDG}\ n$

lemma $in\text{-}Use\text{-}valid\text{-}SDG\text{-}node$:

$V \in Use_{SDG}\ n \Longrightarrow valid\text{-}SDG\text{-}node\ n$

$\langle proof \rangle$

lemma $SDG\text{-}Use\text{-}parent\text{-}Use$:

$V \in Use_{SDG}\ n \Longrightarrow V \in Use\ (parent\text{-}node\ n)$

$\langle proof \rangle$

inductive $SDG\text{-}Def :: 'var \Rightarrow 'node\ SDG\text{-}node \Rightarrow bool\ (- \in Def_{SDG}\ -)$

where $CFG\text{-}Def\text{-}SDG\text{-}Def$:

$\llbracket valid\text{-}node\ m; V \in Def\ m; n = CFG\text{-}node\ m \rrbracket \Longrightarrow V \in Def_{SDG}\ n$

| $Formal\text{-}in\text{-}SDG\text{-}Def$:

$\llbracket valid\text{-}SDG\text{-}node\ n; n = Formal\text{-}in\ (m,x); get\text{-}proc\ m = p; (p,ins,outs) \in set\ procs;$

$V = ins!x \rrbracket \Longrightarrow V \in Def_{SDG}\ n$

| $Actual\text{-}out\text{-}SDG\text{-}Def$:

$\llbracket valid\text{-}SDG\text{-}node\ n; n = Actual\text{-}out\ (m,x); V = (ParamDefs\ m)!x \rrbracket \Longrightarrow V \in Def_{SDG}\ n$

abbreviation $notin\text{-}SDG\text{-}Def :: 'var \Rightarrow 'node\ SDG\text{-}node \Rightarrow bool\ (- \notin Def_{SDG}\ -)$

where $V \notin Def_{SDG}\ n \equiv \neg V \in Def_{SDG}\ n$

lemma $in\text{-}Def\text{-}valid\text{-}SDG\text{-}node$:

$V \in Def_{SDG}\ n \Longrightarrow valid\text{-}SDG\text{-}node\ n$

$\langle proof \rangle$

lemma *SDG-Def-parent-Def*:

$V \in Def_{SDG} n \implies V \in Def$ (parent-node n)
 ⟨proof⟩

definition *data-dependence* :: 'node *SDG-node* \Rightarrow 'var \Rightarrow 'node *SDG-node* \Rightarrow bool

(- influences - in - [51,0,0])
where n influences V in $n' \equiv \exists as. (V \in Def_{SDG} n) \wedge (V \in Use_{SDG} n') \wedge$
 (parent-node $n - as \rightarrow_i^* \text{parent-node } n')$ \wedge
 ($\forall n''. \text{valid-SDG-node } n'' \wedge \text{parent-node } n'' \in \text{set}(\text{sourcenodes}(tl\ as))$
 $\longrightarrow V \notin Def_{SDG} n''$)

1.8.3 Control dependence

definition *control-dependence* :: 'node \Rightarrow 'node \Rightarrow bool

(- controls - [51,0])
where n controls $n' \equiv \exists a\ a'\ as. n - a\#as \rightarrow_i^* n' \wedge n' \notin \text{set}(\text{sourcenodes}(a\#as))$
 \wedge
 intra-kind(kind a) $\wedge n'$ postdominates (targetnode a) \wedge
 valid-edge $a' \wedge$ intra-kind(kind a') \wedge sourcenode $a' = n \wedge$
 $\neg n'$ postdominates (targetnode a')

lemma *control-dependence-path*:

assumes n controls n' **obtains** as **where** $n - as \rightarrow_i^* n'$ **and** $as \neq []$
 ⟨proof⟩

lemma *Exit-does-not-control* [*dest*]:

assumes (\neg Exit-) controls n' **shows** False
 ⟨proof⟩

lemma *Exit-not-control-dependent*:

assumes n controls n' **shows** $n' \neq (\neg$ Exit-)
 ⟨proof⟩

lemma *which-node-intra-standard-control-dependence-source*:

assumes $nx - as@a\#as' \rightarrow_i^* n$ **and** sourcenode $a = n'$ **and** sourcenode $a' = n'$
and $n \notin \text{set}(\text{sourcenodes}(a\#as'))$ **and** valid-edge a' **and** intra-kind(kind a')
and inner-node n **and** \neg method-exit n **and** $\neg n$ postdominates (targetnode a')
and last: $\forall ax\ ax'. ax \in \text{set } as' \wedge \text{sourcenode } ax = \text{sourcenode } ax' \wedge$
 valid-edge $ax' \wedge$ intra-kind(kind ax') $\longrightarrow n$ postdominates targetnode ax'
shows n' controls n

<proof>

1.8.4 SDG without summary edges

inductive *cdep-edge* :: 'node SDG-node ⇒ 'node SDG-node ⇒ bool
 (- →_{cd} - [51,0] 80)
and *ddep-edge* :: 'node SDG-node ⇒ 'var ⇒ 'node SDG-node ⇒ bool
 (- →_{dd} - [51,0,0] 80)
and *call-edge* :: 'node SDG-node ⇒ 'pname ⇒ 'node SDG-node ⇒ bool
 (- →_{call} - [51,0,0] 80)
and *return-edge* :: 'node SDG-node ⇒ 'pname ⇒ 'node SDG-node ⇒ bool
 (- →_{ret} - [51,0,0] 80)
and *param-in-edge* :: 'node SDG-node ⇒ 'pname ⇒ 'var ⇒ 'node SDG-node ⇒ bool
 (- →_{in} - [51,0,0,0] 80)
and *param-out-edge* :: 'node SDG-node ⇒ 'pname ⇒ 'var ⇒ 'node SDG-node ⇒ bool
 (- →_{out} - [51,0,0,0] 80)
and *SDG-edge* :: 'node SDG-node ⇒ 'var option ⇒ ('pname × bool) option ⇒ 'node SDG-node ⇒ bool

where

$n \rightarrow_{cd} n' == SDG-edge\ n\ None\ None\ n'$
 $| n - V \rightarrow_{dd} n' == SDG-edge\ n\ (Some\ V)\ None\ n'$
 $| n - p \rightarrow_{call} n' == SDG-edge\ n\ None\ (Some(p, True))\ n'$
 $| n - p \rightarrow_{ret} n' == SDG-edge\ n\ None\ (Some(p, False))\ n'$
 $| n - p : V \rightarrow_{in} n' == SDG-edge\ n\ (Some\ V)\ (Some(p, True))\ n'$
 $| n - p : V \rightarrow_{out} n' == SDG-edge\ n\ (Some\ V)\ (Some(p, False))\ n'$

$| SDG-cdep-edge:$
 $\llbracket n = CFG-node\ m; n' = CFG-node\ m'; m\ controls\ m' \rrbracket \implies n \rightarrow_{cd} n'$

$| SDG-proc-entry-exit-cdep:$
 $\llbracket valid-edge\ a; kind\ a = Q:r \leftarrow pfs; n = CFG-node\ (targetnode\ a); a' \in get-return-edges\ a; n' = CFG-node\ (sourcnode\ a') \rrbracket \implies n \rightarrow_{cd} n'$

$| SDG-parent-cdep-edge:$
 $\llbracket valid-SDG-node\ n'; m = parent-node\ n'; n = CFG-node\ m; n \neq n' \rrbracket \implies n \rightarrow_{cd} n'$

$| SDG-ddep-edge:n\ influences\ V\ in\ n' \implies n - V \rightarrow_{dd} n'$

$| SDG-call-edge:$
 $\llbracket valid-edge\ a; kind\ a = Q:r \leftarrow pfs; n = CFG-node\ (sourcnode\ a); n' = CFG-node\ (targetnode\ a) \rrbracket \implies n - p \rightarrow_{call} n'$

$| SDG-return-edge:$
 $\llbracket valid-edge\ a; kind\ a = Q \leftarrow pf; n = CFG-node\ (sourcnode\ a); n' = CFG-node\ (targetnode\ a) \rrbracket \implies n - p \rightarrow_{ret} n'$

$| SDG-param-in-edge:$
 $\llbracket valid-edge\ a; kind\ a = Q:r \leftarrow pfs; (p, ins, outs) \in set\ procs; V = ins!x; x < length\ ins; n = Actual-in\ (sourcnode\ a, x); n' = Formal-in\ (targetnode$

$a, x)$
 $\implies n - p: V \rightarrow_{in} n'$
 | *SDG-param-out-edge*:
 $\llbracket \text{valid-edge } a; \text{ kind } a = Q \leftrightarrow pf; (p, ins, outs) \in \text{set procs}; V = outs!x;$
 $x < \text{length } outs; n = \text{Formal-out (sourcenode } a, x);$
 $n' = \text{Actual-out (targetnode } a, x) \rrbracket$
 $\implies n - p: V \rightarrow_{out} n'$

lemma *cdep-edge-cases*:

$\llbracket n \rightarrow_{cd} n'; (\text{parent-node } n) \text{ controls } (\text{parent-node } n') \implies P;$
 $\wedge a \ Q \ r \ p \ fs \ a'. \llbracket \text{valid-edge } a; \text{ kind } a = Q: r \hookrightarrow pfs; a' \in \text{get-return-edges } a;$
 $\text{parent-node } n = \text{targetnode } a; \text{parent-node } n' = \text{sourcenode } a \rrbracket \implies$
 $P;$
 $\wedge m. \llbracket n = \text{CFG-node } m; m = \text{parent-node } n'; n \neq n' \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *SDG-edge-valid-SDG-node*:

assumes *SDG-edge* $n \ Vopt \ popt \ n'$
shows *valid-SDG-node* n **and** *valid-SDG-node* n'
 $\langle \text{proof} \rangle$

lemma *valid-SDG-node-cases*:

assumes *valid-SDG-node* n
shows $n = \text{CFG-node (parent-node } n) \vee \text{CFG-node (parent-node } n) \rightarrow_{cd} n$
 $\langle \text{proof} \rangle$

lemma *SDG-cdep-edge-CFG-node*: $n \rightarrow_{cd} n' \implies \exists m. n = \text{CFG-node } m$
 $\langle \text{proof} \rangle$

lemma *SDG-call-edge-CFG-node*: $n - p \rightarrow_{call} n' \implies \exists m. n = \text{CFG-node } m$
 $\langle \text{proof} \rangle$

lemma *SDG-return-edge-CFG-node*: $n - p \rightarrow_{ret} n' \implies \exists m. n = \text{CFG-node } m$
 $\langle \text{proof} \rangle$

lemma *SDG-call-or-param-in-edge-unique-CFG-call-edge*:

$\text{SDG-edge } n \ Vopt \ (\text{Some}(p, \text{True})) \ n'$
 $\implies \exists ! a. \text{valid-edge } a \wedge \text{sourcenode } a = \text{parent-node } n \wedge$
 $\text{targetnode } a = \text{parent-node } n' \wedge (\exists Q \ r \ fs. \text{kind } a = Q: r \hookrightarrow pfs)$
 $\langle \text{proof} \rangle$

lemma *SDG-return-or-param-out-edge-unique-CFG-return-edge*:

$SDG\text{-edge } n \text{ Vopt } (Some(p, False)) \ n'$
 $\implies \exists ! a. \text{ valid-edge } a \wedge \text{ sourcenode } a = \text{ parent-node } n \wedge$
 $\text{ targetnode } a = \text{ parent-node } n' \wedge (\exists Q f. \text{ kind } a = Q \leftrightarrow pf)$
 <proof>

lemma *Exit-no-SDG-edge-source:*
 $SDG\text{-edge } (CFG\text{-node } (-Exit-)) \ \text{Vopt } \text{popt } n' \implies False$
 <proof>

1.8.5 Intraprocedural paths in the SDG

inductive *intra-SDG-path* ::
 $'node \ SDG\text{-node} \Rightarrow 'node \ SDG\text{-node} \ list \Rightarrow 'node \ SDG\text{-node} \Rightarrow bool$
 $(- \ i \dashrightarrow_d^* \ - \ [51, 0, 0] \ 80)$

where *iSp-Nil:*
 $\text{valid-SDG-node } n \implies n \ i \dashrightarrow_d^* \ n$

| *iSp-Append-cdep:*
 $\llbracket n \ i \text{-ns} \rightarrow_d^* \ n''; n'' \dashrightarrow_{cd} \ n' \rrbracket \implies n \ i \text{-ns} @ [n''] \rightarrow_d^* \ n'$

| *iSp-Append-ddep:*
 $\llbracket n \ i \text{-ns} \rightarrow_d^* \ n''; n'' \dashrightarrow_{dd} \ n'; n'' \neq n' \rrbracket \implies n \ i \text{-ns} @ [n''] \rightarrow_d^* \ n'$

lemma *intra-SDG-path-Append:*
 $\llbracket n'' \ i \text{-ns}' \rightarrow_d^* \ n'; n \ i \text{-ns} \rightarrow_d^* \ n'' \rrbracket \implies n \ i \text{-ns} @ ns' \rightarrow_d^* \ n'$
 <proof>

lemma *intra-SDG-path-valid-SDG-node:*
assumes $n \ i \text{-ns} \rightarrow_d^* \ n'$ **shows** *valid-SDG-node* n **and** *valid-SDG-node* n'
 <proof>

lemma *intra-SDG-path-intra-CFG-path:*
assumes $n \ i \text{-ns} \rightarrow_d^* \ n'$
obtains *as* **where** *parent-node* $n \dashrightarrow_{i,*} \ \text{parent-node } n'$
 <proof>

1.8.6 Control dependence paths in the SDG

inductive *cddep-SDG-path* ::
 $'node \ SDG\text{-node} \Rightarrow 'node \ SDG\text{-node} \ list \Rightarrow 'node \ SDG\text{-node} \Rightarrow bool$
 $(- \ cd \dashrightarrow_d^* \ - \ [51, 0, 0] \ 80)$

where *cdSp-Nil:*
 $\text{valid-SDG-node } n \implies n \ cd \dashrightarrow_d^* \ n$

| *cdSp-Append-cdep*:
 $\llbracket n \text{ cd-ns} \rightarrow_d^* n''; n'' \rightarrow_{cd} n \rrbracket \implies n \text{ cd-ns} @ [n''] \rightarrow_d^* n'$

lemma *cdep-SDG-path-intra-SDG-path*:
 $n \text{ cd-ns} \rightarrow_d^* n' \implies n \text{ i-ns} \rightarrow_d^* n'$
 ⟨*proof*⟩

lemma *Entry-cdep-SDG-path*:
 assumes $(\text{-Entry-}) \text{ -as} \rightarrow_i^* n'$ and *inner-node* n' and \neg *method-exit* n'
 obtains *ns* where *CFG-node* $(\text{-Entry-}) \text{ cd-ns} \rightarrow_d^* \text{CFG-node } n'$
 and $ns \neq []$ and $\forall n'' \in \text{set } ns. \text{parent-node } n'' \in \text{set}(\text{sourcenodes } as)$
 ⟨*proof*⟩

lemma *in-proc-cdep-SDG-path*:
 assumes $n \text{ -as} \rightarrow_i^* n'$ and $n \neq n'$ and $n' \neq (\text{-Exit-})$ and *valid-edge* a
 and *kind* $a = Q:r \hookrightarrow_p fs$ and *targetnode* $a = n$
 obtains *ns* where *CFG-node* $n \text{ cd-ns} \rightarrow_d^* \text{CFG-node } n'$
 and $ns \neq []$ and $\forall n'' \in \text{set } ns. \text{parent-node } n'' \in \text{set}(\text{sourcenodes } as)$
 ⟨*proof*⟩

1.8.7 Paths consisting of calls and control dependences

inductive *call-cdep-SDG-path* ::
 $'node \text{ SDG-node} \Rightarrow 'node \text{ SDG-node list} \Rightarrow 'node \text{ SDG-node} \Rightarrow \text{bool}$
 $(\text{- cc-} \rightarrow_d^* \text{-} [51,0,0] \ 80)$
where *ccSp-Nil*:
 $\text{valid-SDG-node } n \implies n \text{ cc-} [] \rightarrow_d^* n$

| *ccSp-Append-cdep*:
 $\llbracket n \text{ cc-ns} \rightarrow_d^* n''; n'' \rightarrow_{cd} n \rrbracket \implies n \text{ cc-ns} @ [n''] \rightarrow_d^* n'$

| *ccSp-Append-call*:
 $\llbracket n \text{ cc-ns} \rightarrow_d^* n''; n'' \text{ -p} \rightarrow_{call} n \rrbracket \implies n \text{ cc-ns} @ [n''] \rightarrow_d^* n'$

lemma *cc-SDG-path-Append*:
 $\llbracket n'' \text{ cc-ns}' \rightarrow_d^* n'; n \text{ cc-ns} \rightarrow_d^* n'' \rrbracket \implies n \text{ cc-ns} @ ns' \rightarrow_d^* n'$
 ⟨*proof*⟩

lemma *cdep-SDG-path-cc-SDG-path*:
 $n \text{ cd-ns} \rightarrow_d^* n' \implies n \text{ cc-ns} \rightarrow_d^* n'$
 ⟨*proof*⟩

lemma *Entry-cc-SDG-path-to-inner-node*:

assumes *valid-SDG-node* n **and** *parent-node* $n \neq (-Exit-)$
obtains ns **where** *CFG-node* $(-Entry-)$ $cc-ns \rightarrow_d^* n$
 $\langle proof \rangle$

1.8.8 Same level paths in the SDG

inductive *matched* :: *'node SDG-node* \Rightarrow *'node SDG-node list* \Rightarrow *'node SDG-node*
 \Rightarrow *bool*

where *matched-Nil*:

valid-SDG-node $n \Longrightarrow$ *matched* $n [] n$

| *matched-Append-intra-SDG-path*:

\llbracket *matched* $n ns n''$; $n'' i-ns' \rightarrow_d^* n'$ $\rrbracket \Longrightarrow$ *matched* $n (ns@ns') n'$

| *matched-bracket-call*:

\llbracket *matched* $n_0 ns n_1$; $n_1 -p \rightarrow_{call} n_2$; *matched* $n_2 ns' n_3$;
 $(n_3 -p \rightarrow_{ret} n_4 \vee n_3 -p:V \rightarrow_{out} n_4)$; *valid-edge* a ; $a' \in$ *get-return-edges* a ;
sourcenode $a =$ *parent-node* n_1 ; *targetnode* $a =$ *parent-node* n_2 ;
sourcenode $a' =$ *parent-node* n_3 ; *targetnode* $a' =$ *parent-node* n_4 \rrbracket
 \Longrightarrow *matched* $n_0 (ns@n_1 \# ns'@[n_3]) n_4$

| *matched-bracket-param*:

\llbracket *matched* $n_0 ns n_1$; $n_1 -p:V \rightarrow_{in} n_2$; *matched* $n_2 ns' n_3$;
 $n_3 -p:V' \rightarrow_{out} n_4$; *valid-edge* a ; $a' \in$ *get-return-edges* a ;
sourcenode $a =$ *parent-node* n_1 ; *targetnode* $a =$ *parent-node* n_2 ;
sourcenode $a' =$ *parent-node* n_3 ; *targetnode* $a' =$ *parent-node* n_4 \rrbracket
 \Longrightarrow *matched* $n_0 (ns@n_1 \# ns'@[n_3]) n_4$

lemma *matched-Append*:

\llbracket *matched* $n'' ns' n'$; *matched* $n ns n''$ $\rrbracket \Longrightarrow$ *matched* $n (ns@ns') n'$

$\langle proof \rangle$

lemma *intra-SDG-path-matched*:

assumes $n i-ns \rightarrow_d^* n'$ **shows** *matched* $n ns n'$

$\langle proof \rangle$

lemma *intra-proc-matched*:

assumes *valid-edge* a **and** *kind* $a = Q:r \hookrightarrow_p fs$ **and** $a' \in$ *get-return-edges* a
shows *matched* (*CFG-node* (*targetnode* a)) [*CFG-node* (*targetnode* a)]
(*CFG-node* (*sourcenode* a'))

$\langle proof \rangle$

lemma *matched-intra-CFG-path*:

assumes *matched* $n ns n'$

obtains as **where** *parent-node* $n -as \rightarrow_{i,*}$ *parent-node* n'

$\langle proof \rangle$

lemma *matched-same-level-CFG-path*:
assumes *matched* n ns n'
obtains *as* **where** *parent-node* $n -as \rightarrow_{sl^*}$ *parent-node* n'
 $\langle proof \rangle$

1.8.9 Realizable paths in the SDG

inductive *realizable* ::
 $'node$ *SDG-node* $\Rightarrow 'node$ *SDG-node list* $\Rightarrow 'node$ *SDG-node* $\Rightarrow bool$
where *realizable-matched:matched* n ns $n' \Longrightarrow realizable$ n ns n'
| *realizable-call*:
 $\llbracket realizable$ n_0 ns $n_1; n_1 -p \rightarrow_{call}$ $n_2 \vee n_1 -p:V \rightarrow_{in}$ $n_2; matched$ n_2 ns' $n_3 \rrbracket$
 $\Longrightarrow realizable$ n_0 $(ns@n_1\#ns')$ n_3

lemma *realizable-Append-matched*:
 $\llbracket realizable$ n ns $n''; matched$ n'' ns' $n' \rrbracket \Longrightarrow realizable$ n $(ns@ns')$ n'
 $\langle proof \rangle$

lemma *realizable-valid-CFG-path*:
assumes *realizable* n ns n'
obtains *as* **where** *parent-node* $n -as \rightarrow_{\sqrt{*}}$ *parent-node* n'
 $\langle proof \rangle$

lemma *cdep-SDG-path-realizable*:
 n *cc-ns* \rightarrow_{d^*} $n' \Longrightarrow realizable$ n ns n'
 $\langle proof \rangle$

1.8.10 SDG with summary edges

inductive *sum-cdep-edge* :: $'node$ *SDG-node* $\Rightarrow 'node$ *SDG-node* $\Rightarrow bool$
 $(- s \rightarrow_{cd} - [51,0] 80)$
and *sum-ddep-edge* :: $'node$ *SDG-node* $\Rightarrow 'var \Rightarrow 'node$ *SDG-node* $\Rightarrow bool$
 $(- s \dashrightarrow_{dd} - [51,0,0] 80)$
and *sum-call-edge* :: $'node$ *SDG-node* $\Rightarrow 'pname \Rightarrow 'node$ *SDG-node* $\Rightarrow bool$
 $(- s \dashrightarrow_{call} - [51,0,0] 80)$
and *sum-return-edge* :: $'node$ *SDG-node* $\Rightarrow 'pname \Rightarrow 'node$ *SDG-node* $\Rightarrow bool$
 $(- s \dashrightarrow_{ret} - [51,0,0] 80)$
and *sum-param-in-edge* :: $'node$ *SDG-node* $\Rightarrow 'pname \Rightarrow 'var \Rightarrow 'node$ *SDG-node*
 $\Rightarrow bool$
 $(- s \dashrightarrow_{in} - [51,0,0,0] 80)$
and *sum-param-out-edge* :: $'node$ *SDG-node* $\Rightarrow 'pname \Rightarrow 'var \Rightarrow 'node$ *SDG-node*
 $\Rightarrow bool$
 $(- s \dashrightarrow_{out} - [51,0,0,0] 80)$
and *sum-summary-edge* :: $'node$ *SDG-node* $\Rightarrow 'pname \Rightarrow 'node$ *SDG-node* $\Rightarrow bool$

($- s \dashrightarrow_{sum} - [51,0] 80$)
and $sum\text{-SDG-edge} :: 'node\ SDG\text{-node} \Rightarrow 'var\ option \Rightarrow$
 $('pname \times bool)\ option \Rightarrow bool \Rightarrow 'node\ SDG\text{-node} \Rightarrow bool$

where

$n s \longrightarrow_{cd} n' == sum\text{-SDG-edge } n\ None\ None\ False\ n'$
 $| n s - V \rightarrow_{dd} n' == sum\text{-SDG-edge } n\ (Some\ V)\ None\ False\ n'$
 $| n s - p \rightarrow_{call} n' == sum\text{-SDG-edge } n\ None\ (Some(p, True))\ False\ n'$
 $| n s - p \rightarrow_{ret} n' == sum\text{-SDG-edge } n\ None\ (Some(p, False))\ False\ n'$
 $| n s - p: V \rightarrow_{in} n' == sum\text{-SDG-edge } n\ (Some\ V)\ (Some(p, True))\ False\ n'$
 $| n s - p: V \rightarrow_{out} n' == sum\text{-SDG-edge } n\ (Some\ V)\ (Some(p, False))\ False\ n'$
 $| n s - p \rightarrow_{sum} n' == sum\text{-SDG-edge } n\ None\ (Some(p, True))\ True\ n'$

$| sum\text{-SDG-cdep-edge}:$
 $\llbracket n = CFG\text{-node } m; n' = CFG\text{-node } m'; m\ controls\ m' \rrbracket \Longrightarrow n s \longrightarrow_{cd} n'$

$| sum\text{-SDG-proc-entry-exit-cdep}:$
 $\llbracket valid\text{-edge } a; kind\ a = Q:r \hookrightarrow pfs; n = CFG\text{-node } (targetnode\ a);$
 $a' \in get\text{-return-edges } a; n' = CFG\text{-node } (sourcenode\ a') \rrbracket \Longrightarrow n s \longrightarrow_{cd} n'$

$| sum\text{-SDG-parent-cdep-edge}:$
 $\llbracket valid\text{-SDG-node } n'; m = parent\text{-node } n'; n = CFG\text{-node } m; n \neq n' \rrbracket$
 $\Longrightarrow n s \longrightarrow_{cd} n'$

$| sum\text{-SDG-ddep-edge}: n\ influences\ V\ in\ n' \Longrightarrow n s - V \rightarrow_{dd} n'$

$| sum\text{-SDG-call-edge}:$
 $\llbracket valid\text{-edge } a; kind\ a = Q:r \hookrightarrow pfs; n = CFG\text{-node } (sourcenode\ a);$
 $n' = CFG\text{-node } (targetnode\ a) \rrbracket \Longrightarrow n s - p \rightarrow_{call} n'$

$| sum\text{-SDG-return-edge}:$
 $\llbracket valid\text{-edge } a; kind\ a = Q \leftarrow pfs; n = CFG\text{-node } (sourcenode\ a);$
 $n' = CFG\text{-node } (targetnode\ a) \rrbracket \Longrightarrow n s - p \rightarrow_{ret} n'$

$| sum\text{-SDG-param-in-edge}:$
 $\llbracket valid\text{-edge } a; kind\ a = Q:r \hookrightarrow pfs; (p, ins, outs) \in set\ procs; V = ins!x;$
 $x < length\ ins; n = Actual\text{-in } (sourcenode\ a, x); n' = Formal\text{-in } (targetnode$
 $a, x) \rrbracket$
 $\Longrightarrow n s - p: V \rightarrow_{in} n'$

$| sum\text{-SDG-param-out-edge}:$
 $\llbracket valid\text{-edge } a; kind\ a = Q \leftarrow pf; (p, ins, outs) \in set\ procs; V = outs!x;$
 $x < length\ outs; n = Formal\text{-out } (sourcenode\ a, x);$
 $n' = Actual\text{-out } (targetnode\ a, x) \rrbracket$
 $\Longrightarrow n s - p: V \rightarrow_{out} n'$

$| sum\text{-SDG-call-summary-edge}:$
 $\llbracket valid\text{-edge } a; kind\ a = Q:r \hookrightarrow pfs; a' \in get\text{-return-edges } a;$
 $n = CFG\text{-node } (sourcenode\ a); n' = CFG\text{-node } (targetnode\ a') \rrbracket$
 $\Longrightarrow n s - p \rightarrow_{sum} n'$

$| sum\text{-SDG-param-summary-edge}:$
 $\llbracket valid\text{-edge } a; kind\ a = Q:r \hookrightarrow pfs; a' \in get\text{-return-edges } a;$
 $matched\ (Formal\text{-in } (targetnode\ a, x))\ ns\ (Formal\text{-out } (sourcenode\ a', x'));$
 $n = Actual\text{-in } (sourcenode\ a, x); n' = Actual\text{-out } (targetnode\ a', x');$
 $(p, ins, outs) \in set\ procs; x < length\ ins; x' < length\ outs \rrbracket$

$\implies n \text{ s-p} \rightarrow \text{sum } n'$

lemma *sum-edge-cases*:

$\llbracket n \text{ s-p} \rightarrow \text{sum } n';$
 $\bigwedge a \ Q \ r \ fs \ a'. \llbracket \text{valid-edge } a; \text{ kind } a = Q:r \hookrightarrow pfs; a' \in \text{get-return-edges } a;$
 $n = \text{CFG-node (sourcenode } a); n' = \text{CFG-node (targetnode } a') \rrbracket \implies$
 $P;$
 $\bigwedge a \ Q \ p \ r \ fs \ a' \ ns \ x \ x' \ ins \ outs.$
 $\llbracket \text{valid-edge } a; \text{ kind } a = Q:r \hookrightarrow pfs; a' \in \text{get-return-edges } a;$
 $\text{matched (Formal-in (targetnode } a,x)) \ ns \ (\text{Formal-out (sourcenode } a',x'));$
 $n = \text{Actual-in (sourcenode } a,x); n' = \text{Actual-out (targetnode } a',x');$
 $(p,ins,outs) \in \text{set procs}; x < \text{length } ins; x' < \text{length } outs \rrbracket \implies P \rrbracket$
 $\implies P$
 $\langle \text{proof} \rangle$

lemma *SDG-edge-sum-SDG-edge*:

$\text{SDG-edge } n \ \text{Vopt } \text{popt } n' \implies \text{sum-SDG-edge } n \ \text{Vopt } \text{popt } \text{False } n'$
 $\langle \text{proof} \rangle$

lemma *sum-SDG-edge-SDG-edge*:

$\text{sum-SDG-edge } n \ \text{Vopt } \text{popt } \text{False } n' \implies \text{SDG-edge } n \ \text{Vopt } \text{popt } n'$
 $\langle \text{proof} \rangle$

lemma *sum-SDG-edge-valid-SDG-node*:

assumes $\text{sum-SDG-edge } n \ \text{Vopt } \text{popt } b \ n'$
shows $\text{valid-SDG-node } n$ **and** $\text{valid-SDG-node } n'$
 $\langle \text{proof} \rangle$

lemma *Exit-no-sum-SDG-edge-source*:

assumes $\text{sum-SDG-edge (CFG-node (-Exit-)) Vopt } \text{popt } b \ n'$ **shows** False
 $\langle \text{proof} \rangle$

lemma *Exit-no-sum-SDG-edge-target*:

$\text{sum-SDG-edge } n \ \text{Vopt } \text{popt } b \ (\text{CFG-node (-Exit-)}) \implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *sum-SDG-summary-edge-matched*:

assumes $n \text{ s-p} \rightarrow \text{sum } n'$
obtains ns **where** $\text{matched } n \ ns \ n'$ **and** $n \in \text{set } ns$

and $get\text{-}proc\ (parent\text{-}node(last\ ns)) = p$
 $\langle proof \rangle$

lemma *return-edge-determines-call-and-sum-edge*:

assumes *valid-edge* a **and** $kind\ a = Q \leftarrow pf$
obtains $a' Q' r' fs'$ **where** $a \in get\text{-}return\text{-}edges\ a'$ **and** *valid-edge* a'
and $kind\ a' = Q' : r' \leftarrow pfs'$
and *CFG-node* (*sourcenode* a') $s\text{-}p \rightarrow sum$ *CFG-node* (*targetnode* a)
 $\langle proof \rangle$

1.8.11 Paths consisting of intraprocedural and summary edges in the SDG

inductive *intra-sum-SDG-path* ::

$'node\ SDG\text{-}node \Rightarrow 'node\ SDG\text{-}node\ list \Rightarrow 'node\ SDG\text{-}node \Rightarrow bool$
 $(- is\ \text{---}\ \rightarrow_d^* - [51, 0, 0] 80)$

where *isSp-Nil*:

$valid\text{-}SDG\text{-}node\ n \Longrightarrow n\ is\ \text{---}\ \rightarrow_d^* n$

| *isSp-Append-cdep*:

$\llbracket n\ is\ ns \rightarrow_d^* n''; n''\ s \rightarrow_{cd}\ n' \rrbracket \Longrightarrow n\ is\ ns @ [n'] \rightarrow_d^* n'$

| *isSp-Append-ddep*:

$\llbracket n\ is\ ns \rightarrow_d^* n''; n''\ s\ V \rightarrow_{dd}\ n'; n'' \neq n' \rrbracket \Longrightarrow n\ is\ ns @ [n'] \rightarrow_d^* n'$

| *isSp-Append-sum*:

$\llbracket n\ is\ ns \rightarrow_d^* n''; n''\ s\ p \rightarrow_{sum}\ n' \rrbracket \Longrightarrow n\ is\ ns @ [n'] \rightarrow_d^* n'$

lemma *is-SDG-path-Append*:

$\llbracket n''\ is\ ns' \rightarrow_d^* n'; n\ is\ ns \rightarrow_d^* n'' \rrbracket \Longrightarrow n\ is\ ns @ ns' \rightarrow_d^* n'$
 $\langle proof \rangle$

lemma *is-SDG-path-valid-SDG-node*:

assumes $n\ is\ ns \rightarrow_d^* n'$ **shows** *valid-SDG-node* n **and** *valid-SDG-node* n'
 $\langle proof \rangle$

lemma *intra-SDG-path-is-SDG-path*:

$n\ i\ ns \rightarrow_d^* n' \Longrightarrow n\ is\ ns \rightarrow_d^* n'$
 $\langle proof \rangle$

lemma *is-SDG-path-hd*: $\llbracket n\ is\ ns \rightarrow_d^* n'; ns \neq [] \rrbracket \Longrightarrow hd\ ns = n$

$\langle proof \rangle$

lemma *intra-sum-SDG-path-rev-induct* [*consumes 1, case-names isSp-Nil isSp-Cons-cdep isSp-Cons-ddep isSp-Cons-sum*]:
assumes $n \text{ is-ns} \rightarrow_{d^*} n'$
and *refl*: $\bigwedge n. \text{ valid-SDG-node } n \implies P \ n \ [] \ n$
and *step-cdep*: $\bigwedge n \ ns \ n' \ n''. \llbracket n \ s \rightarrow_{cd} \ n''; \ n'' \ \text{is-ns} \rightarrow_{d^*} \ n'; \ P \ n'' \ ns \ n' \rrbracket$
 $\implies P \ n \ (n\#ns) \ n'$
and *step-ddep*: $\bigwedge n \ ns \ n' \ V \ n''. \llbracket n \ s \ - \ V \rightarrow_{dd} \ n''; \ n \neq \ n''; \ n'' \ \text{is-ns} \rightarrow_{d^*} \ n';$
 $P \ n'' \ ns \ n' \rrbracket \implies P \ n \ (n\#ns) \ n'$
and *step-sum*: $\bigwedge n \ ns \ n' \ p \ n''. \llbracket n \ s \ - \ p \rightarrow_{sum} \ n''; \ n'' \ \text{is-ns} \rightarrow_{d^*} \ n'; \ P \ n'' \ ns \ n' \rrbracket$
 $\implies P \ n \ (n\#ns) \ n'$
shows $P \ n \ ns \ n'$
 $\langle \text{proof} \rangle$

lemma *is-SDG-path-CFG-path*:
assumes $n \text{ is-ns} \rightarrow_{d^*} n'$
obtains as where *parent-node* $n \ - \ \text{as} \rightarrow_{\iota^*} \ \text{parent-node } n'$
 $\langle \text{proof} \rangle$

lemma *matched-is-SDG-path*:
assumes *matched* $n \ ns \ n'$ **obtains** ns' **where** $n \ \text{is-ns}' \rightarrow_{d^*} \ n'$
 $\langle \text{proof} \rangle$

lemma *is-SDG-path-matched*:
assumes $n \ \text{is-ns} \rightarrow_{d^*} \ n'$ **obtains** ns' **where** *matched* $n \ ns' \ n'$ **and** $\text{set } ns \subseteq \text{set } ns'$
 $\langle \text{proof} \rangle$

lemma *is-SDG-path-intra-CFG-path*:
assumes $n \ \text{is-ns} \rightarrow_{d^*} \ n'$
obtains as where *parent-node* $n \ - \ \text{as} \rightarrow_{\iota^*} \ \text{parent-node } n'$
 $\langle \text{proof} \rangle$

SDG paths without return edges

inductive *intra-call-sum-SDG-path* ::
 $'node \ \text{SDG-node} \Rightarrow 'node \ \text{SDG-node list} \Rightarrow 'node \ \text{SDG-node} \Rightarrow \text{bool}$
 $(- \ \text{ics} \ - \rightarrow_{d^*} \ - \ [51,0,0] \ 80)$
where *icsSp-Nil*:
 $\text{valid-SDG-node } n \implies n \ \text{ics-} [] \rightarrow_{d^*} \ n$

| *icsSp-Append-cdep*:
 $\llbracket n \ \text{ics-ns} \rightarrow_{d^*} \ n''; \ n'' \ s \rightarrow_{cd} \ n' \rrbracket \implies n \ \text{ics-ns}@[n''] \rightarrow_{d^*} \ n'$

| *icsSp-Append-ddep*:
 $\llbracket n \ \text{ics-ns} \rightarrow_{d^*} \ n''; \ n'' \ s \ - \ V \rightarrow_{dd} \ n'; \ n'' \neq \ n' \rrbracket \implies n \ \text{ics-ns}@[n''] \rightarrow_{d^*} \ n'$

| *icsSp-Append-sum*:
 $\llbracket n \text{ ics-ns} \rightarrow_d^* n''; n'' \text{ s-p} \rightarrow_{\text{sum}} n' \rrbracket \Longrightarrow n \text{ ics-ns} @ [n''] \rightarrow_d^* n'$

| *icsSp-Append-call*:
 $\llbracket n \text{ ics-ns} \rightarrow_d^* n''; n'' \text{ s-p} \rightarrow_{\text{call}} n' \rrbracket \Longrightarrow n \text{ ics-ns} @ [n''] \rightarrow_d^* n'$

| *icsSp-Append-param-in*:
 $\llbracket n \text{ ics-ns} \rightarrow_d^* n''; n'' \text{ s-p: } V \rightarrow_{\text{in}} n' \rrbracket \Longrightarrow n \text{ ics-ns} @ [n''] \rightarrow_d^* n'$

lemma *ics-SDG-path-valid-SDG-node*:

assumes $n \text{ ics-ns} \rightarrow_d^* n'$ **shows** *valid-SDG-node* n **and** *valid-SDG-node* n'
 $\langle \text{proof} \rangle$

lemma *ics-SDG-path-Append*:

$\llbracket n'' \text{ ics-ns}' \rightarrow_d^* n'; n \text{ ics-ns} \rightarrow_d^* n'' \rrbracket \Longrightarrow n \text{ ics-ns} @ \text{ns}' \rightarrow_d^* n'$
 $\langle \text{proof} \rangle$

lemma *is-SDG-path-ics-SDG-path*:

$n \text{ is-ns} \rightarrow_d^* n' \Longrightarrow n \text{ ics-ns} \rightarrow_d^* n'$
 $\langle \text{proof} \rangle$

lemma *cc-SDG-path-ics-SDG-path*:

$n \text{ cc-ns} \rightarrow_d^* n' \Longrightarrow n \text{ ics-ns} \rightarrow_d^* n'$
 $\langle \text{proof} \rangle$

lemma *ics-SDG-path-split*:

assumes $n \text{ ics-ns} \rightarrow_d^* n'$ **and** $n'' \in \text{set ns}$
obtains $\text{ns}' \text{ ns}''$ **where** $\text{ns} = \text{ns}' @ \text{ns}''$ **and** $n \text{ ics-ns}' \rightarrow_d^* n''$
and $n'' \text{ ics-ns}'' \rightarrow_d^* n'$
 $\langle \text{proof} \rangle$

lemma *realizable-ics-SDG-path*:

assumes *realizable* $n \text{ ns}' n'$ **obtains** ns' **where** $n \text{ ics-ns}' \rightarrow_d^* n'$
 $\langle \text{proof} \rangle$

lemma *ics-SDG-path-realizable*:

assumes $n \text{ ics-ns} \rightarrow_d^* n'$
obtains ns' **where** *realizable* $n \text{ ns}' n'$ **and** $\text{set ns} \subseteq \text{set ns}'$
 $\langle \text{proof} \rangle$

lemma *realizable-Append-ics-SDG-path*:
assumes *realizable* n ns n'' **and** $n'' \text{ ics-ns}' \rightarrow_{d^*} n'$
obtains ns'' **where** *realizable* n $(ns@ns')$ n'
 $\langle \text{proof} \rangle$

1.8.12 SDG paths without call edges

inductive *intra-return-sum-SDG-path* ::
 $'node \text{ SDG-node} \Rightarrow 'node \text{ SDG-node list} \Rightarrow 'node \text{ SDG-node} \Rightarrow \text{bool}$
 $(- \text{ irs} \dashrightarrow_{d^*} - [51,0,0] \ 80)$
where *irsSp-Nil*:
 $\text{valid-SDG-node } n \Longrightarrow n \text{ irs-} [] \rightarrow_{d^*} n$

| *irsSp-Cons-cdep*:
 $\llbracket n'' \text{ irs-ns} \rightarrow_{d^*} n'; n \text{ s} \rightarrow_{cd} n'' \rrbracket \Longrightarrow n \text{ irs-} n \# ns \rightarrow_{d^*} n'$

| *irsSp-Cons-ddep*:
 $\llbracket n'' \text{ irs-ns} \rightarrow_{d^*} n'; n \text{ s} \rightarrow_{V} n''; n \neq n'' \rrbracket \Longrightarrow n \text{ irs-} n \# ns \rightarrow_{d^*} n'$

| *irsSp-Cons-sum*:
 $\llbracket n'' \text{ irs-ns} \rightarrow_{d^*} n'; n \text{ s-p} \rightarrow_{sum} n'' \rrbracket \Longrightarrow n \text{ irs-} n \# ns \rightarrow_{d^*} n'$

| *irsSp-Cons-return*:
 $\llbracket n'' \text{ irs-ns} \rightarrow_{d^*} n'; n \text{ s-p} \rightarrow_{ret} n'' \rrbracket \Longrightarrow n \text{ irs-} n \# ns \rightarrow_{d^*} n'$

| *irsSp-Cons-param-out*:
 $\llbracket n'' \text{ irs-ns} \rightarrow_{d^*} n'; n \text{ s-p:V} \rightarrow_{out} n'' \rrbracket \Longrightarrow n \text{ irs-} n \# ns \rightarrow_{d^*} n'$

lemma *irs-SDG-path-Append*:
 $\llbracket n \text{ irs-ns} \rightarrow_{d^*} n''; n'' \text{ irs-ns}' \rightarrow_{d^*} n' \rrbracket \Longrightarrow n \text{ irs-ns}@ns' \rightarrow_{d^*} n'$
 $\langle \text{proof} \rangle$

lemma *is-SDG-path-irs-SDG-path*:
 $n \text{ is-ns} \rightarrow_{d^*} n' \Longrightarrow n \text{ irs-ns} \rightarrow_{d^*} n'$
 $\langle \text{proof} \rangle$

lemma *irs-SDG-path-split*:
assumes $n \text{ irs-ns} \rightarrow_{d^*} n'$
obtains $n \text{ is-ns} \rightarrow_{d^*} n'$
| $nsx \ nsx' \ nx \ nx' \ p$ **where** $ns = nsx@nx \# nsx'$ **and** $n \text{ irs-nsx} \rightarrow_{d^*} nx$
and $nx \text{ s-p} \rightarrow_{ret} nx' \vee (\exists V. nx \text{ s-p:V} \rightarrow_{out} nx')$ **and** $nx' \text{ is-nsx}' \rightarrow_{d^*} n'$
 $\langle \text{proof} \rangle$

lemma *irs-SDG-path-matched*:

assumes $n \text{ irs-ns} \rightarrow_d^* n''$ **and** $n'' \text{ s-p} \rightarrow_{ret} n' \vee n'' \text{ s-p: } V \rightarrow_{out} n'$
obtains $nx \text{ nsx}$ **where** $matched \text{ nx nsx } n'$ **and** $n \in \text{set nsx}$
and $nx \text{ s-p} \rightarrow_{sum} \text{CFG-node (parent-node } n')$
 ⟨proof⟩

lemma *irs-SDG-path-realizable*:

assumes $n \text{ irs-ns} \rightarrow_d^* n'$ **and** $n \neq n'$
obtains ns' **where** *realizable (CFG-node (-Entry-)) ns' n'* **and** $n \in \text{set ns'}$
 ⟨proof⟩

end

end

1.9 Horwitz-Reps-Binkley Slice

theory *HRBSlice* **imports** *SDG* **begin**

context *SDG* **begin**

1.9.1 Set describing phase 1 of the two-phase slicer

inductive-set *sum-SDG-slice1* :: *'node SDG-node* \Rightarrow *'node SDG-node set*
for $n::'node \text{ SDG-node}$
where *refl-slice1:valid-SDG-node* $n \Longrightarrow n \in \text{sum-SDG-slice1 } n$
 | *cdep-slice1*:
 $\llbracket n'' \text{ s} \rightarrow_{cd} n'; n' \in \text{sum-SDG-slice1 } n \rrbracket \Longrightarrow n'' \in \text{sum-SDG-slice1 } n$
 | *ddep-slice1*:
 $\llbracket n'' \text{ s-} V \rightarrow_{dd} n'; n' \in \text{sum-SDG-slice1 } n \rrbracket \Longrightarrow n'' \in \text{sum-SDG-slice1 } n$
 | *call-slice1*:
 $\llbracket n'' \text{ s-p} \rightarrow_{call} n'; n' \in \text{sum-SDG-slice1 } n \rrbracket \Longrightarrow n'' \in \text{sum-SDG-slice1 } n$
 | *param-in-slice1*:
 $\llbracket n'' \text{ s-p: } V \rightarrow_{in} n'; n' \in \text{sum-SDG-slice1 } n \rrbracket \Longrightarrow n'' \in \text{sum-SDG-slice1 } n$
 | *sum-slice1*:
 $\llbracket n'' \text{ s-p} \rightarrow_{sum} n'; n' \in \text{sum-SDG-slice1 } n \rrbracket \Longrightarrow n'' \in \text{sum-SDG-slice1 } n$

lemma *slice1-cdep-slice1*:

$\llbracket nx \in \text{sum-SDG-slice1 } n; n \text{ s} \rightarrow_{cd} n' \rrbracket \Longrightarrow nx \in \text{sum-SDG-slice1 } n'$
 ⟨proof⟩

lemma *slice1-ddep-slice1*:

$\llbracket nx \in \text{sum-SDG-slice1 } n; n \text{ s-} V \rightarrow_{dd} n' \rrbracket \Longrightarrow nx \in \text{sum-SDG-slice1 } n'$
 ⟨proof⟩

lemma *slice1-sum-slice1*:

$\llbracket nx \in \text{sum-SDG-slice1 } n; n \text{ s-p} \rightarrow_{sum} n' \rrbracket \Longrightarrow nx \in \text{sum-SDG-slice1 } n'$
 ⟨proof⟩

lemma *slice1-call-slice1*:

$\llbracket nx \in \text{sum-SDG-slice1 } n; n \text{ s-p} \rightarrow_{\text{call}} n' \rrbracket \implies nx \in \text{sum-SDG-slice1 } n'$
 ⟨proof⟩

lemma *slice1-param-in-slice1*:

$\llbracket nx \in \text{sum-SDG-slice1 } n; n \text{ s-p: } V \rightarrow_{\text{in}} n' \rrbracket \implies nx \in \text{sum-SDG-slice1 } n'$
 ⟨proof⟩

lemma *is-SDG-path-slice1*:

$\llbracket n \text{ is-ns} \rightarrow_{d^*} n'; n' \in \text{sum-SDG-slice1 } n' \rrbracket \implies n \in \text{sum-SDG-slice1 } n'$
 ⟨proof⟩

1.9.2 Set describing phase 2 of the two-phase slicer

inductive-set *sum-SDG-slice2* :: 'node SDG-node \Rightarrow 'node SDG-node set
for $n::$ 'node SDG-node

where *refl-slice2*: *valid-SDG-node* $n \implies n \in \text{sum-SDG-slice2 } n$

| *cdep-slice2*:

$\llbracket n'' \text{ s} \rightarrow_{\text{cd}} n'; n' \in \text{sum-SDG-slice2 } n \rrbracket \implies n'' \in \text{sum-SDG-slice2 } n$

| *ddep-slice2*:

$\llbracket n'' \text{ s-} V \rightarrow_{\text{dd}} n'; n' \in \text{sum-SDG-slice2 } n \rrbracket \implies n'' \in \text{sum-SDG-slice2 } n$

| *return-slice2*:

$\llbracket n'' \text{ s-p} \rightarrow_{\text{ret}} n'; n' \in \text{sum-SDG-slice2 } n \rrbracket \implies n'' \in \text{sum-SDG-slice2 } n$

| *param-out-slice2*:

$\llbracket n'' \text{ s-p: } V \rightarrow_{\text{out}} n'; n' \in \text{sum-SDG-slice2 } n \rrbracket \implies n'' \in \text{sum-SDG-slice2 } n$

| *sum-slice2*:

$\llbracket n'' \text{ s-p} \rightarrow_{\text{sum}} n'; n' \in \text{sum-SDG-slice2 } n \rrbracket \implies n'' \in \text{sum-SDG-slice2 } n$

lemma *slice2-cdep-slice2*:

$\llbracket nx \in \text{sum-SDG-slice2 } n; n \text{ s} \rightarrow_{\text{cd}} n' \rrbracket \implies nx \in \text{sum-SDG-slice2 } n'$
 ⟨proof⟩

lemma *slice2-ddep-slice2*:

$\llbracket nx \in \text{sum-SDG-slice2 } n; n \text{ s-} V \rightarrow_{\text{dd}} n' \rrbracket \implies nx \in \text{sum-SDG-slice2 } n'$
 ⟨proof⟩

lemma *slice2-sum-slice2*:

$\llbracket nx \in \text{sum-SDG-slice2 } n; n \text{ s-p} \rightarrow_{\text{sum}} n' \rrbracket \implies nx \in \text{sum-SDG-slice2 } n'$
 ⟨proof⟩

lemma *slice2-ret-slice2*:

$\llbracket nx \in \text{sum-SDG-slice2 } n; n \text{ s-p} \rightarrow_{\text{ret}} n' \rrbracket \implies nx \in \text{sum-SDG-slice2 } n'$
 ⟨proof⟩

lemma *slice2-param-out-slice2*:

$\llbracket nx \in \text{sum-SDG-slice2 } n; n \text{ s-p: } V \rightarrow_{\text{out}} n' \rrbracket \implies nx \in \text{sum-SDG-slice2 } n'$

<proof>

lemma *is-SDG-path-slice2*:

$\llbracket n \text{ is-ns} \rightarrow_d^* n'; n' \in \text{sum-SDG-slice2 } n' \rrbracket \implies n \in \text{sum-SDG-slice2 } n'$
<proof>

lemma *slice2-is-SDG-path-slice2*:

$\llbracket n \text{ is-ns} \rightarrow_d^* n'; n'' \in \text{sum-SDG-slice2 } n \rrbracket \implies n'' \in \text{sum-SDG-slice2 } n'$
<proof>

1.9.3 The backward slice using the Horwitz-Reps-Binkley slicer

Note: our slicing criterion is a set of nodes, not a unique node.

inductive-set *combine-SDG-slices* :: 'node SDG-node set \Rightarrow 'node SDG-node set
for *S*::'node SDG-node set

where *combSlice-refl*: $n \in S \implies n \in \text{combine-SDG-slices } S$

| *combSlice-Return-parent-node*:

$\llbracket n' \in S; n'' \text{ s-p} \rightarrow_{\text{ret}} \text{CFG-node (parent-node } n') \rrbracket; n \in \text{sum-SDG-slice2 } n'$
 $\implies n \in \text{combine-SDG-slices } S$

definition *HRB-slice* :: 'node SDG-node set \Rightarrow 'node SDG-node set

where *HRB-slice* $S \equiv \{n'. \exists n \in S. n' \in \text{combine-SDG-slices (sum-SDG-slice1 } n)\}$

lemma *HRB-slice-cases*[*consumes 1, case-names phase1 phase2*]:

$\llbracket x \in \text{HRB-slice } S; \bigwedge n \text{ nx. } \llbracket n \in \text{sum-SDG-slice1 } nx; nx \in S \rrbracket \implies P n;$
 $\bigwedge nx \text{ n' n'' p n. } \llbracket n' \in \text{sum-SDG-slice1 } nx; n'' \text{ s-p} \rightarrow_{\text{ret}} \text{CFG-node (parent-node } n') \rrbracket;$

$n \in \text{sum-SDG-slice2 } n'; nx \in S \rrbracket \implies P n$

$\implies P x$

<proof>

lemma *HRB-slice-refl*:

assumes *valid-node m and CFG-node m \in S shows* *CFG-node m \in HRB-slice S*

<proof>

lemma *HRB-slice-valid-node*: $n \in \text{HRB-slice } S \implies \text{valid-SDG-node } n$

<proof>

lemma *valid-SDG-node-in-slice-parent-node-in-slice:*

assumes $n \in \text{HRB-slice } S$ **shows** $\text{CFG-node (parent-node } n) \in \text{HRB-slice } S$
 $\langle \text{proof} \rangle$

lemma *HRB-slice-is-SDG-path-HRB-slice:*

$\llbracket n \text{ is-ns} \rightarrow_d^* n'; n'' \in \text{HRB-slice } \{n\}; n' \in S \rrbracket \implies n'' \in \text{HRB-slice } S$
 $\langle \text{proof} \rangle$

lemma *call-return-nodes-in-slice:*

assumes *valid-edge* a **and** $\text{kind } a = Q \leftrightarrow_p f$
and *valid-edge* a' **and** $\text{kind } a' = Q':r' \hookrightarrow_p f s'$ **and** $a \in \text{get-return-edges } a'$
and $\text{CFG-node (targetnode } a) \in \text{HRB-slice } S$
shows $\text{CFG-node (sourcnode } a) \in \text{HRB-slice } S$
and $\text{CFG-node (sourcnode } a') \in \text{HRB-slice } S$
and $\text{CFG-node (targetnode } a') \in \text{HRB-slice } S$
 $\langle \text{proof} \rangle$

1.9.4 Proof of Precision

lemma *in-intra-SDG-path-in-slice2:*

$\llbracket n \text{ i-ns} \rightarrow_d^* n'; n'' \in \text{set ns} \rrbracket \implies n'' \in \text{sum-SDG-slice2 } n'$
 $\langle \text{proof} \rangle$

lemma *in-intra-SDG-path-in-HRB-slice:*

$\llbracket n \text{ i-ns} \rightarrow_d^* n'; n'' \in \text{set ns}; n' \in S \rrbracket \implies n'' \in \text{HRB-slice } S$
 $\langle \text{proof} \rangle$

lemma *in-matched-in-slice2:*

$\llbracket \text{matched } n \text{ ns } n'; n'' \in \text{set ns} \rrbracket \implies n'' \in \text{sum-SDG-slice2 } n'$
 $\langle \text{proof} \rangle$

lemma *in-matched-in-HRB-slice:*

$\llbracket \text{matched } n \text{ ns } n'; n'' \in \text{set ns}; n' \in S \rrbracket \implies n'' \in \text{HRB-slice } S$
 $\langle \text{proof} \rangle$

theorem *in-realizable-in-HRB-slice:*

$\llbracket \text{realizable } n \text{ ns } n'; n'' \in \text{set ns}; n' \in S \rrbracket \implies n'' \in \text{HRB-slice } S$
 $\langle \text{proof} \rangle$

lemma *slice1-ics-SDG-path:*

assumes $n \in \text{sum-SDG-slice1 } n'$ **and** $n \neq n'$

obtains ns **where** *CFG-node* (-Entry-) $ics-ns \rightarrow_d^* n'$ **and** $n \in set\ ns$
 ⟨*proof*⟩

lemma *slice2-irs-SDG-path*:
assumes $n \in sum-SDG-slice2\ n'$ **and** *valid-SDG-node* n'
obtains ns **where** $n\ irs-ns \rightarrow_d^* n'$
 ⟨*proof*⟩

theorem *HRB-slice-realizable*:
assumes $n \in HRB-slice\ S$ **and** $\forall n' \in S.$ *valid-SDG-node* n' **and** $n \notin S$
obtains n' ns **where** $n' \in S$ **and** *realizable* (*CFG-node* (-Entry-)) $ns\ n'$
and $n \in set\ ns$
 ⟨*proof*⟩

theorem *HRB-slice-precise*:
 $\llbracket \forall n' \in S. \text{valid-SDG-node } n'; n \notin S \rrbracket \implies$
 $n \in HRB-slice\ S =$
 $(\exists n' ns. n' \in S \wedge \text{realizable } (CFG-node\ (-Entry-))\ ns\ n' \wedge n \in set\ ns)$
 ⟨*proof*⟩

end

end

1.10 Observable sets w.r.t. standard control dependence

theory *SCDObservable* **imports** *Observable HRBSlice* **begin**

context *SDG* **begin**

lemma *matched-bracket-assms-variant*:
assumes $n_1 -p \rightarrow_{call} n_2 \vee n_1 -p: V' \rightarrow_{in} n_2$ **and** *matched* $n_2\ ns'\ n_3$
and $n_3 -p \rightarrow_{ret} n_4 \vee n_3 -p: V \rightarrow_{out} n_4$
and *call-of-return-node* (*parent-node* n_4) (*parent-node* n_1)
obtains $a\ a'$ **where** *valid-edge* a **and** $a' \in get-return-edges\ a$
and *sourcenode* $a = parent-node\ n_1$ **and** *targetnode* $a = parent-node\ n_2$
and *sourcenode* $a' = parent-node\ n_3$ **and** *targetnode* $a' = parent-node\ n_4$
 ⟨*proof*⟩

1.10.1 Observable set of standard control dependence is at most a singleton

definition *SDG-to-CFG-set* :: $'node\ SDG-node\ set \Rightarrow 'node\ set\ ([_]_{CFG})$
where $[S]_{CFG} \equiv \{m. \text{CFG-node } m \in S\}$

lemma [intro]: $\forall n \in S. \text{valid-SDG-node } n \implies \forall n \in [S]_{CFG}. \text{valid-node } n$
 ⟨proof⟩

lemma *Exit-HRB-Slice*:

assumes $n \in [HRB\text{-slice } \{CFG\text{-node } (-Exit-)\}]_{CFG}$ **shows** $n = (-Exit-)$
 ⟨proof⟩

lemma *Exit-in-obs-intra-slice-node*:

assumes $(-Exit-) \in \text{obs-intra } n' [HRB\text{-slice } S]_{CFG}$
shows $CFG\text{-node } (-Exit-) \in S$
 ⟨proof⟩

lemma *obs-intra-postdominate*:

assumes $n \in \text{obs-intra } n' [HRB\text{-slice } S]_{CFG}$ **and** $\neg \text{method-exit } n$
shows n *postdominates* n'
 ⟨proof⟩

lemma *obs-intra-singleton-disj*:

assumes *valid-node* n
shows $(\exists m. \text{obs-intra } n [HRB\text{-slice } S]_{CFG} = \{m\}) \vee$
 $\text{obs-intra } n [HRB\text{-slice } S]_{CFG} = \{\}$
 ⟨proof⟩

lemma *obs-intra-finite:valid-node* $n \implies \text{finite } (\text{obs-intra } n [HRB\text{-slice } S]_{CFG})$
 ⟨proof⟩

lemma *obs-intra-singleton:valid-node* $n \implies \text{card } (\text{obs-intra } n [HRB\text{-slice } S]_{CFG})$
 ≤ 1
 ⟨proof⟩

lemma *obs-intra-singleton-element*:

$m \in \text{obs-intra } n [HRB\text{-slice } S]_{CFG} \implies \text{obs-intra } n [HRB\text{-slice } S]_{CFG} = \{m\}$
 ⟨proof⟩

lemma *obs-intra-the-element*:

$m \in \text{obs-intra } n [HRB\text{-slice } S]_{CFG} \implies (\text{THE } m. m \in \text{obs-intra } n [HRB\text{-slice } S]_{CFG}) = m$
 ⟨proof⟩

lemma *obs-singleton-element*:

assumes $ms \in \text{obs } ns \lfloor \text{HRB-slice } S \rfloor_{CFG}$ **and** $\forall n \in \text{set } (tl \ ns)$. *return-node* n
shows $\text{obs } ns \lfloor \text{HRB-slice } S \rfloor_{CFG} = \{ms\}$
(*proof*)

lemma *obs-finite*: $\forall n \in \text{set } (tl \ ns)$. *return-node* n

$\implies \text{finite } (\text{obs } ns \lfloor \text{HRB-slice } S \rfloor_{CFG})$
(*proof*)

lemma *obs-singleton*: $\forall n \in \text{set } (tl \ ns)$. *return-node* n

$\implies \text{card } (\text{obs } ns \lfloor \text{HRB-slice } S \rfloor_{CFG}) \leq 1$
(*proof*)

lemma *obs-the-element*:

$\llbracket ms \in \text{obs } ns \lfloor \text{HRB-slice } S \rfloor_{CFG}; \forall n \in \text{set } (tl \ ns)$. *return-node* $n \rrbracket$
 $\implies (\text{THE } ms. ms \in \text{obs } ns \lfloor \text{HRB-slice } S \rfloor_{CFG}) = ms$
(*proof*)

end

end

1.11 Distance of Paths

theory *Distance* **imports** *CFG* **begin**

context *CFG* **begin**

inductive *distance* :: $'node \Rightarrow 'node \Rightarrow nat \Rightarrow bool$

where *distanceI*:

$\llbracket n - as \rightarrow_i^* n'; \text{length } as = x; \forall as'. n - as' \rightarrow_i^* n' \longrightarrow x \leq \text{length } as \rrbracket$
 $\implies \text{distance } n \ n' \ x$

lemma *every-path-distance*:

assumes $n - as \rightarrow_i^* n'$
obtains x **where** $\text{distance } n \ n' \ x$ **and** $x \leq \text{length } as$
(*proof*)

lemma *distance-det*:

$\llbracket \text{distance } n \ n' \ x; \text{distance } n \ n' \ x' \rrbracket \implies x = x'$
(*proof*)

lemma *only-one-SOME-dist-edge*:

assumes *valid-edge a* **and** *intra-kind(kind a)* **and** *distance (targetnode a) n' x*
shows $\exists! a'. \text{ sourcenode } a = \text{ sourcenode } a' \wedge \text{ distance } (\text{targetnode } a') n' x \wedge$
valid-edge a' \wedge intra-kind(kind a') \wedge
targetnode a' = (SOME nx. $\exists a'. \text{ sourcenode } a = \text{ sourcenode } a' \wedge$
distance (targetnode a') n' x \wedge
valid-edge a' \wedge intra-kind(kind a') \wedge
targetnode a' = nx)

<proof>

lemma *distance-successor-distance*:

assumes *distance n n' x* **and** $x \neq 0$
obtains a where *valid-edge a* **and** $n = \text{ sourcenode } a$ **and** *intra-kind(kind a)*
and *distance (targetnode a) n' (x - 1)*
and *targetnode a = (SOME nx. $\exists a'. \text{ sourcenode } a = \text{ sourcenode } a' \wedge$*
distance (targetnode a') n' (x - 1) \wedge
valid-edge a' \wedge intra-kind(kind a') \wedge
targetnode a' = nx)

<proof>

end

end

1.12 Static backward slice

theory *Slice* **imports** *SCDObservable Distance* **begin**

context *SDG* **begin**

1.12.1 Preliminary definitions on the parameter nodes for defining sliced call and return edges

fun *csppa* :: *'node \Rightarrow 'node SDG-node set \Rightarrow nat \Rightarrow*
((('var \rightarrow 'val) \Rightarrow 'val option) list) \Rightarrow ((('var \rightarrow 'val) \Rightarrow 'val option) list)
where *csppa m S x [] = []*
| csppa m S x (f#fs) =
(if Formal-in(m,x) \notin S then Map.empty else f)#csppa m S (Suc x) fs

definition *cspp* :: *'node \Rightarrow 'node SDG-node set \Rightarrow*
((('var \rightarrow 'val) \Rightarrow 'val option) list) \Rightarrow ((('var \rightarrow 'val) \Rightarrow 'val option) list)
where *cspp m S fs \equiv csppa m S 0 fs*

lemma [*simp*]: *length (csppa m S x fs) = length fs*
<proof>

lemma [*simp*]: *length (cspp m S fs) = length fs*
<proof>

lemma *csppa-Formal-in-notin-slice*:

$\llbracket x < \text{length } fs; \text{Formal-in}(m, x + i) \notin S \rrbracket$
 $\implies (\text{csppa } m \ S \ i \ fs)!x = \text{Map.empty}$
 $\langle \text{proof} \rangle$

lemma *csppa-Formal-in-in-slice*:

$\llbracket x < \text{length } fs; \text{Formal-in}(m, x + i) \in S \rrbracket$
 $\implies (\text{csppa } m \ S \ i \ fs)!x = fs!x$
 $\langle \text{proof} \rangle$

definition *map-merge* :: ('var \rightarrow 'val) \Rightarrow ('var \rightarrow 'val) \Rightarrow (nat \Rightarrow bool) \Rightarrow
'var list \Rightarrow ('var \rightarrow 'val)

where *map-merge* $f \ g \ Q \ xs \equiv (\lambda V. \text{if } (\exists i. i < \text{length } xs \wedge xs!i = V \wedge Q \ i) \text{ then}$
 $g \ V$
 $\text{else } f \ V)$

definition *rspp* :: 'node \Rightarrow 'node SDG-node set \Rightarrow 'var list \Rightarrow
('var \rightarrow 'val) \Rightarrow ('var \rightarrow 'val) \Rightarrow ('var \rightarrow 'val)

where *rspp* $m \ S \ xs \ f \ g \equiv \text{map-merge } f \ (\text{Map.empty}(\text{ParamDefs } m \ [:=] \ \text{map } g \ xs))$
 $(\lambda i. \text{Actual-out}(m, i) \in S) \ (\text{ParamDefs } m)$

lemma *rspp-Actual-out-in-slice*:

assumes $x < \text{length } (\text{ParamDefs } (\text{targetnode } a))$ **and** *valid-edge* a
and $\text{length } (\text{ParamDefs } (\text{targetnode } a)) = \text{length } xs$
and $\text{Actual-out } (\text{targetnode } a, x) \in S$
shows $(\text{rspp } (\text{targetnode } a) \ S \ xs \ f \ g) ((\text{ParamDefs } (\text{targetnode } a))!x) = g(xs!x)$
 $\langle \text{proof} \rangle$

lemma *rspp-Actual-out-notin-slice*:

assumes $x < \text{length } (\text{ParamDefs } (\text{targetnode } a))$ **and** *valid-edge* a
and $\text{length } (\text{ParamDefs } (\text{targetnode } a)) = \text{length } xs$
and $\text{Actual-out}((\text{targetnode } a), x) \notin S$
shows $(\text{rspp } (\text{targetnode } a) \ S \ xs \ f \ g) ((\text{ParamDefs } (\text{targetnode } a))!x) =$
 $f((\text{ParamDefs } (\text{targetnode } a))!x)$
 $\langle \text{proof} \rangle$

1.12.2 Defining the sliced edge kinds

primrec *slice-kind-aux* :: 'node \Rightarrow 'node \Rightarrow 'node SDG-node set \Rightarrow

('var, 'val, 'ret, 'pname) edge-kind \Rightarrow ('var, 'val, 'ret, 'pname) edge-kind

where *slice-kind-aux* $m \ m' \ S \ \uparrow f = (\text{if } m \in \llbracket S \rrbracket_{CFG} \text{ then } \uparrow f \ \text{else } \uparrow id)$

$\mid \text{slice-kind-aux } m \ m' \ S \ (Q)_{\surd} = (\text{if } m \in \llbracket S \rrbracket_{CFG} \text{ then } (Q)_{\surd} \ \text{else}$

$(\text{if } \text{obs-intra } m \ \llbracket S \rrbracket_{CFG} = \{\} \text{ then}$

$(\text{let } mex = (\text{THE } mex. \text{method-exit } mex \wedge \text{get-proc } m = \text{get-proc } mex) \text{ in}$

$(\text{if } (\exists x. \text{distance } m' \ mex \ x \wedge \text{distance } m \ mex \ (x + 1) \wedge$

$$\begin{aligned}
& (m' = (\text{SOME } mx'. \exists a'. m = \text{sourcenode } a' \wedge \\
& \quad \text{distance } (\text{targetnode } a') \text{ } mex \ x \wedge \\
& \quad \text{valid-edge } a' \wedge \text{intra-kind}(\text{kind } a') \wedge \\
& \quad \text{targetnode } a' = mx')) \\
& \quad \text{then } (\lambda cf. \text{True})_{\surd} \text{ else } (\lambda cf. \text{False})_{\surd}) \\
& \text{else } (\text{let } mx = \text{THE } mx. mx \in \text{obs-intra } m \ [S]_{CFG} \text{ in} \\
& \quad (\text{if } (\exists x. \text{distance } m' \ mx \ x \wedge \text{distance } m \ mx \ (x + 1) \wedge \\
& \quad \quad (m' = (\text{SOME } mx'. \exists a'. m = \text{sourcenode } a' \wedge \\
& \quad \quad \quad \text{distance } (\text{targetnode } a') \ mx \ x \wedge \\
& \quad \quad \quad \text{valid-edge } a' \wedge \text{intra-kind}(\text{kind } a') \wedge \\
& \quad \quad \quad \text{targetnode } a' = mx')) \\
& \quad \quad \text{then } (\lambda cf. \text{True})_{\surd} \text{ else } (\lambda cf. \text{False})_{\surd}))) \\
& | \text{slice-kind-aux } m \ m' \ S \ (Q:r \leftrightarrow_p fs) = (\text{if } m \in [S]_{CFG} \text{ then } (Q:r \leftrightarrow_p (cspp \ m' \ S \\
& fs)) \\
& \quad \quad \text{else } ((\lambda cf. \text{False}):r \leftrightarrow_p fs)) \\
& | \text{slice-kind-aux } m \ m' \ S \ (Q \leftrightarrow_p f) = (\text{if } m \in [S]_{CFG} \text{ then} \\
& \quad (\text{let } \text{outs} = \text{THE } \text{outs}. \exists \text{ins}. (p, \text{ins}, \text{outs}) \in \text{set procs in} \\
& \quad \quad (Q \leftrightarrow_p (\lambda cf \ cf'. rspp \ m' \ S \ \text{outs} \ cf' \ cf))) \\
& \quad \text{else } ((\lambda cf. \text{True}) \leftrightarrow_p (\lambda cf \ cf'. cf')))
\end{aligned}$$

definition *slice-kind* :: 'node SDG-node set \Rightarrow 'edge \Rightarrow
('var,'val,'ret,'pname) edge-kind
where *slice-kind* $S \ a \equiv$
slice-kind-aux (sourcenode a) (targetnode a) (HRB-slice S) (kind a)

definition *slice-kinds* :: 'node SDG-node set \Rightarrow 'edge list \Rightarrow
('var,'val,'ret,'pname) edge-kind list
where *slice-kinds* $S \ as \equiv \text{map } (\text{slice-kind } S) \ as$

lemma *slice-intra-kind-in-slice*:
 $\llbracket \text{sourcenode } a \in [HRB\text{-slice } S]_{CFG}; \text{intra-kind } (\text{kind } a) \rrbracket$
 $\implies \text{slice-kind } S \ a = \text{kind } a$
<proof>

lemma *slice-kind-Upd*:
 $\llbracket \text{sourcenode } a \notin [HRB\text{-slice } S]_{CFG}; \text{kind } a = \uparrow f \rrbracket \implies \text{slice-kind } S \ a = \uparrow id$
<proof>

lemma *slice-kind-Pred-empty-obs-nearer-SOME*:
assumes $\text{sourcenode } a \notin [HRB\text{-slice } S]_{CFG}$ **and** $\text{kind } a = (Q)_{\surd}$
and $\text{obs-intra } (\text{sourcenode } a) \ [HRB\text{-slice } S]_{CFG} = \{\}$
and $\text{method-exit } mex$ **and** $\text{get-proc } (\text{sourcenode } a) = \text{get-proc } mex$
and $\text{distance } (\text{targetnode } a) \ mex \ x$ **and** $\text{distance } (\text{sourcenode } a) \ mex \ (x + 1)$
and $\text{targetnode } a = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\quad \text{distance } (\text{targetnode } a') \ mex \ x \wedge$

$valid-edge\ a' \wedge intra-kind(kind\ a') \wedge$
 $targetnode\ a' = n'$

shows $slice-kind\ S\ a = (\lambda s. True)_{\checkmark}$
 $\langle proof \rangle$

lemma *slice-kind-Pred-empty-obs-nearer-not-SOME*:

assumes $sourcenode\ a \notin [HRB-slice\ S]_{CFG}$ **and** $kind\ a = (Q)_{\checkmark}$
and $obs-intra\ (sourcenode\ a)\ [HRB-slice\ S]_{CFG} = \{\}$
and $method-exit\ mex$ **and** $get-proc\ (sourcenode\ a) = get-proc\ mex$
and $distance\ (targetnode\ a)\ mex\ x$ **and** $distance\ (sourcenode\ a)\ mex\ (x + 1)$
and $targetnode\ a \neq (SOME\ n'. \exists a'. sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ mex\ x \wedge$
 $valid-edge\ a' \wedge intra-kind(kind\ a') \wedge$
 $targetnode\ a' = n')$

shows $slice-kind\ S\ a = (\lambda s. False)_{\checkmark}$
 $\langle proof \rangle$

lemma *slice-kind-Pred-empty-obs-not-nearer*:

assumes $sourcenode\ a \notin [HRB-slice\ S]_{CFG}$ **and** $kind\ a = (Q)_{\checkmark}$
and $obs-intra\ (sourcenode\ a)\ [HRB-slice\ S]_{CFG} = \{\}$
and $method-exit\ mex$ **and** $get-proc\ (sourcenode\ a) = get-proc\ mex$
and $dist:distance\ (sourcenode\ a)\ mex\ (x + 1) \neg distance\ (targetnode\ a)\ mex\ x$
shows $slice-kind\ S\ a = (\lambda s. False)_{\checkmark}$

$\langle proof \rangle$

lemma *slice-kind-Pred-obs-nearer-SOME*:

assumes $sourcenode\ a \notin [HRB-slice\ S]_{CFG}$ **and** $kind\ a = (Q)_{\checkmark}$
and $m \in obs-intra\ (sourcenode\ a)\ [HRB-slice\ S]_{CFG}$
and $distance\ (targetnode\ a)\ m\ x$ $distance\ (sourcenode\ a)\ m\ (x + 1)$
and $targetnode\ a = (SOME\ n'. \exists a'. sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ m\ x \wedge$
 $valid-edge\ a' \wedge intra-kind(kind\ a') \wedge$
 $targetnode\ a' = n')$

shows $slice-kind\ S\ a = (\lambda s. True)_{\checkmark}$
 $\langle proof \rangle$

lemma *slice-kind-Pred-obs-nearer-not-SOME*:

assumes $sourcenode\ a \notin [HRB-slice\ S]_{CFG}$ **and** $kind\ a = (Q)_{\checkmark}$
and $m \in obs-intra\ (sourcenode\ a)\ [HRB-slice\ S]_{CFG}$
and $distance\ (targetnode\ a)\ m\ x$ $distance\ (sourcenode\ a)\ m\ (x + 1)$
and $targetnode\ a \neq (SOME\ nx'. \exists a'. sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ m\ x \wedge$
 $valid-edge\ a' \wedge intra-kind(kind\ a') \wedge$
 $targetnode\ a' = nx')$

shows $slice-kind\ S\ a = (\lambda s. False)_{\checkmark}$

$\langle proof \rangle$

lemma *slice-kind-Pred-obs-not-nearer:*

assumes $sourcenode\ a \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **and** $kind\ a = (Q)_{\checkmark}$

and $in\text{-}obs:m \in obs\text{-}intra\ (sourcenode\ a)\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$

and $dist:distance\ (sourcenode\ a)\ m\ (x + 1)$

$\neg\ distance\ (targetnode\ a)\ m\ x$

shows $slice\text{-}kind\ S\ a = (\lambda s. False)_{\checkmark}$

$\langle proof \rangle$

lemma *kind-Predicate-notin-slice-slice-kind-Predicate:*

assumes $sourcenode\ a \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **and** $valid\text{-}edge\ a$ **and** $kind\ a = (Q)_{\checkmark}$

obtains Q' **where** $slice\text{-}kind\ S\ a = (Q')_{\checkmark}$ **and** $Q' = (\lambda s. False) \vee Q' = (\lambda s. True)$

$\langle proof \rangle$

lemma *slice-kind-Call:*

$\llbracket sourcenode\ a \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG};\ kind\ a = Q:r \hookrightarrow_p fs \rrbracket$

$\implies slice\text{-}kind\ S\ a = (\lambda cf. False):r \hookrightarrow_p fs$

$\langle proof \rangle$

lemma *slice-kind-Call-in-slice:*

$\llbracket sourcenode\ a \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG};\ kind\ a = Q:r \hookrightarrow_p fs \rrbracket$

$\implies slice\text{-}kind\ S\ a = Q:r \hookrightarrow_p (cspp\ (targetnode\ a)\ (HRB\text{-}slice\ S)\ fs)$

$\langle proof \rangle$

lemma *slice-kind-Call-in-slice-Formal-in-not:*

assumes $sourcenode\ a \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **and** $kind\ a = Q:r \hookrightarrow_p fs$

and $\forall x < length\ fs. Formal\text{-}in(targetnode\ a, x) \notin HRB\text{-}slice\ S$

shows $slice\text{-}kind\ S\ a = Q:r \hookrightarrow_p replicate\ (length\ fs)\ Map.empty$

$\langle proof \rangle$

lemma *slice-kind-Call-in-slice-Formal-in-also:*

assumes $sourcenode\ a \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **and** $kind\ a = Q:r \hookrightarrow_p fs$

and $\forall x < length\ fs. Formal\text{-}in(targetnode\ a, x) \in HRB\text{-}slice\ S$

shows $slice\text{-}kind\ S\ a = Q:r \hookrightarrow_p fs$

$\langle proof \rangle$

lemma *slice-kind-Call-intra-notin-slice:*

assumes $sourcenode\ a \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **and** $valid\text{-}edge\ a$

and $intra\text{-}kind\ (kind\ a)$ **and** $valid\text{-}edge\ a'$ **and** $kind\ a' = Q:r \hookrightarrow_p fs$

and *sourcenode* $a' = \text{sourcenode } a$
shows *slice-kind* $S a = (\lambda s. \text{True})_{\checkmark}$
 $\langle \text{proof} \rangle$

lemma *slice-kind-Return*:
 $\llbracket \text{sourcenode } a \notin \llbracket \text{HRB-slice } S \rrbracket_{CFG}; \text{kind } a = Q \leftrightarrow pf \rrbracket$
 $\implies \text{slice-kind } S a = (\lambda cf. \text{True}) \leftrightarrow_p (\lambda cf \ cf'. \ cf')$
 $\langle \text{proof} \rangle$

lemma *slice-kind-Return-in-slice*:
 $\llbracket \text{sourcenode } a \in \llbracket \text{HRB-slice } S \rrbracket_{CFG}; \text{valid-edge } a; \text{kind } a = Q \leftrightarrow pf;$
 $(p, \text{ins}, \text{outs}) \in \text{set procs} \rrbracket$
 $\implies \text{slice-kind } S a = Q \leftrightarrow_p (\lambda cf \ cf'. \ \text{rspp } (\text{targetnode } a) (\text{HRB-slice } S) \ \text{outs } cf'$
 $cf)$
 $\langle \text{proof} \rangle$

lemma *length-transfer-kind-slice-kind*:
assumes *valid-edge* a **and** *length* $s_1 = \text{length } s_2$
and *transfer* $(\text{kind } a) \ s_1 = s_1'$ **and** *transfer* $(\text{slice-kind } S a) \ s_2 = s_2'$
shows *length* $s_1' = \text{length } s_2'$
 $\langle \text{proof} \rangle$

1.12.3 The sliced graph of a deterministic CFG is still deterministic

lemma *only-one-SOME-edge*:
assumes *valid-edge* a **and** *intra-kind* $(\text{kind } a)$ **and** *distance* $(\text{targetnode } a) \ \text{mex } x$
shows $\exists! a'. \ \text{sourcenode } a = \text{sourcenode } a' \wedge \text{distance } (\text{targetnode } a') \ \text{mex } x \wedge$
 $\text{valid-edge } a' \wedge \text{intra-kind } (\text{kind } a') \wedge$
 $\text{targetnode } a' = (\text{SOME } n'. \ \exists a'. \ \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{distance } (\text{targetnode } a') \ \text{mex } x \wedge$
 $\text{valid-edge } a' \wedge \text{intra-kind } (\text{kind } a') \wedge$
 $\text{targetnode } a' = n')$
 $\langle \text{proof} \rangle$

lemma *slice-kind-only-one-True-edge*:
assumes *sourcenode* $a = \text{sourcenode } a'$ **and** *targetnode* $a \neq \text{targetnode } a'$
and *valid-edge* a **and** *valid-edge* a' **and** *intra-kind* $(\text{kind } a)$
and *intra-kind* $(\text{kind } a')$ **and** *slice-kind* $S a = (\lambda s. \text{True})_{\checkmark}$
shows *slice-kind* $S a' = (\lambda s. \text{False})_{\checkmark}$
 $\langle \text{proof} \rangle$

lemma *slice-deterministic*:
assumes *valid-edge* a **and** *valid-edge* a'

and *intra-kind* (*kind a*) **and** *intra-kind* (*kind a'*)
and *sourcenode a = sourcenode a'* **and** *targetnode a ≠ targetnode a'*
obtains *Q Q'* **where** *slice-kind S a = (Q)_✓* **and** *slice-kind S a' = (Q')_✓*
and $\forall s. (Q\ s \longrightarrow \neg Q'\ s) \wedge (Q'\ s \longrightarrow \neg Q\ s)$
 $\langle proof \rangle$
end
end

1.13 The weak simulation

theory *WeakSimulation* **imports** *Slice* **begin**

context *SDG* **begin**

lemma *call-node-notin-slice-return-node-neither*:
assumes *call-of-return-node n n'* **and** $n' \notin [HRB\text{-}slice\ S]_{CFG}$
shows $n \notin [HRB\text{-}slice\ S]_{CFG}$
 $\langle proof \rangle$

lemma *edge-obs-intra-slice-eq*:
assumes *valid-edge a* **and** *intra-kind (kind a)* **and** $sourcenode\ a \notin [HRB\text{-}slice\ S]_{CFG}$
shows $obs\text{-}intra\ (targetnode\ a)\ [HRB\text{-}slice\ S]_{CFG} =$
 $obs\text{-}intra\ (sourcenode\ a)\ [HRB\text{-}slice\ S]_{CFG}$
 $\langle proof \rangle$

lemma *intra-edge-obs-slice*:
assumes $ms \neq []$ **and** $ms'' \in obs\ ms' [HRB\text{-}slice\ S]_{CFG}$ **and** *valid-edge a*
and *intra-kind (kind a)*
and $disj:(\exists m \in set\ (tl\ ms). \exists m'. call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge$
 $m' \notin [HRB\text{-}slice\ S]_{CFG}) \vee hd\ ms \notin [HRB\text{-}slice\ S]_{CFG}$
and $hd\ ms = sourcenode\ a$ **and** $ms' = targetnode\ a \# tl\ ms$
and $\forall n \in set\ (tl\ ms^{\wedge}). return\text{-}node\ n$
shows $ms'' \in obs\ ms [HRB\text{-}slice\ S]_{CFG}$
 $\langle proof \rangle$

1.13.1 Silent moves

inductive *silent-move* ::
 $'node\ SDG\text{-}node\ set \Rightarrow ('edge \Rightarrow ('var, 'val, 'ret, 'pname)\ edge\text{-}kind) \Rightarrow 'node\ list$
 \Rightarrow
 $((('var \rightarrow 'val) \times 'ret)\ list \Rightarrow 'edge \Rightarrow 'node\ list \Rightarrow ((('var \rightarrow 'val) \times 'ret)\ list \Rightarrow$
 $bool$
 $(-, - \vdash '(-, -) \dashrightarrow_{\tau} '(-, -) [51, 50, 0, 0, 50, 0, 0] 51)$

where *silent-move-intra*:

$\llbracket \text{pred } (f a) s; \text{transfer } (f a) s = s'; \text{valid-edge } a; \text{intra-kind}(kind a);$
 $(\exists m \in \text{set } (tl ms). \exists m'. \text{call-of-return-node } m m' \wedge m' \notin \llbracket \text{HRB-slice } S \rrbracket_{CFG})$

\vee

$hd ms \notin \llbracket \text{HRB-slice } S \rrbracket_{CFG}; \forall m \in \text{set } (tl ms). \text{return-node } m;$
 $\text{length } s' = \text{length } s; \text{length } ms = \text{length } s;$
 $hd ms = \text{sourcenode } a; ms' = (\text{targetnode } a) \# tl ms$
 $\implies S, f \vdash (ms, s) -a \rightarrow_{\tau} (ms', s')$

| *silent-move-call*:

$\llbracket \text{pred } (f a) s; \text{transfer } (f a) s = s'; \text{valid-edge } a; \text{kind } a = Q:r \hookrightarrow pfs;$
 $\text{valid-edge } a'; a' \in \text{get-return-edges } a;$
 $(\exists m \in \text{set } (tl ms). \exists m'. \text{call-of-return-node } m m' \wedge m' \notin \llbracket \text{HRB-slice } S \rrbracket_{CFG})$

\vee

$hd ms \notin \llbracket \text{HRB-slice } S \rrbracket_{CFG}; \forall m \in \text{set } (tl ms). \text{return-node } m;$
 $\text{length } ms = \text{length } s; \text{length } s' = \text{Suc}(\text{length } s);$
 $hd ms = \text{sourcenode } a; ms' = (\text{targetnode } a) \# (\text{targetnode } a') \# tl ms$
 $\implies S, f \vdash (ms, s) -a \rightarrow_{\tau} (ms', s')$

| *silent-move-return*:

$\llbracket \text{pred } (f a) s; \text{transfer } (f a) s = s'; \text{valid-edge } a; \text{kind } a = Q \leftrightarrow pf';$
 $\exists m \in \text{set } (tl ms). \exists m'. \text{call-of-return-node } m m' \wedge m' \notin \llbracket \text{HRB-slice } S \rrbracket_{CFG};$
 $\forall m \in \text{set } (tl ms). \text{return-node } m; \text{length } ms = \text{length } s; \text{length } s = \text{Suc}(\text{length } s');$
 $s' \neq []; hd ms = \text{sourcenode } a; hd(tl ms) = \text{targetnode } a; ms' = tl ms$
 $\implies S, f \vdash (ms, s) -a \rightarrow_{\tau} (ms', s')$

lemma *silent-move-valid-nodes*:

$\llbracket S, f \vdash (ms, s) -a \rightarrow_{\tau} (ms', s'); \forall m \in \text{set } ms'. \text{valid-node } m \rrbracket$
 $\implies \forall m \in \text{set } ms. \text{valid-node } m$

<proof>

lemma *silent-move-return-node*:

$S, f \vdash (ms, s) -a \rightarrow_{\tau} (ms', s') \implies \forall m \in \text{set } (tl ms'). \text{return-node } m$
<proof>

lemma *silent-move-equal-length*:

assumes $S, f \vdash (ms, s) -a \rightarrow_{\tau} (ms', s')$
shows $\text{length } ms = \text{length } s$ **and** $\text{length } ms' = \text{length } s'$

<proof>

lemma *silent-move-obs-slice*:

$\llbracket S, kind \vdash (ms, s) -a \rightarrow_{\tau} (ms', s'); msx \in \text{obs } ms' \llbracket \text{HRB-slice } S \rrbracket_{CFG};$
 $\forall n \in \text{set } (tl ms'). \text{return-node } n \rrbracket$
 $\implies msx \in \text{obs } ms \llbracket \text{HRB-slice } S \rrbracket_{CFG}$

$\langle proof \rangle$

lemma *silent-move-empty-obs-slice*:

assumes $S, f \vdash (ms, s) -a \rightarrow_{\tau} (ms', s')$ **and** $obs\ ms' \ [HRB\text{-}slice\ S]_{CFG} = \{\}$

shows $obs\ ms \ [HRB\text{-}slice\ S]_{CFG} = \{\}$

$\langle proof \rangle$

inductive *silent-moves* ::

$'node\ SDG\text{-}node\ set \Rightarrow ('edge \Rightarrow ('var, 'val, 'ret, 'pname)\ edge\text{-}kind) \Rightarrow 'node\ list$
 \Rightarrow

$(('var \rightarrow 'val) \times 'ret)\ list \Rightarrow 'edge\ list \Rightarrow 'node\ list \Rightarrow (('var \rightarrow 'val) \times 'ret)\ list$
 $\Rightarrow bool$

$(-, \vdash '(-, -) \Rightarrow_{\tau} '(-, -) \ [51, 50, 0, 0, 50, 0, 0] \ 51)$

where *silent-moves-Nil*: $length\ ms = length\ s \Longrightarrow S, f \vdash (ms, s) = [] \Rightarrow_{\tau} (ms, s)$

| *silent-moves-Cons*:

$\llbracket S, f \vdash (ms, s) -a \rightarrow_{\tau} (ms', s'); S, f \vdash (ms', s') = as \Rightarrow_{\tau} (ms'', s'') \rrbracket$

$\Longrightarrow S, f \vdash (ms, s) = a \# as \Rightarrow_{\tau} (ms'', s'')$

lemma *silent-moves-equal-length*:

assumes $S, f \vdash (ms, s) = as \Rightarrow_{\tau} (ms', s')$

shows $length\ ms = length\ s$ **and** $length\ ms' = length\ s'$

$\langle proof \rangle$

lemma *silent-moves-Append*:

$\llbracket S, f \vdash (ms, s) = as \Rightarrow_{\tau} (ms'', s''); S, f \vdash (ms'', s'') = as' \Rightarrow_{\tau} (ms', s') \rrbracket$

$\Longrightarrow S, f \vdash (ms, s) = as @ as' \Rightarrow_{\tau} (ms', s')$

$\langle proof \rangle$

lemma *silent-moves-split*:

assumes $S, f \vdash (ms, s) = as @ as' \Rightarrow_{\tau} (ms', s')$

obtains $ms''\ s''$ **where** $S, f \vdash (ms, s) = as \Rightarrow_{\tau} (ms'', s'')$

and $S, f \vdash (ms'', s'') = as' \Rightarrow_{\tau} (ms', s')$

$\langle proof \rangle$

lemma *valid-nodes-silent-moves*:

$\llbracket S, f \vdash (ms, s) = as' \Rightarrow_{\tau} (ms', s'); \forall m \in set\ ms.\ valid\text{-}node\ m \rrbracket$

$\Longrightarrow \forall m \in set\ ms'.\ valid\text{-}node\ m$

$\langle proof \rangle$

lemma *return-nodes-silent-moves*:

$\llbracket S, f \vdash (ms, s) = as \Rightarrow_{\tau} (ms', s'); \forall m \in \text{set } (tl \ ms). \text{ return-node } m \rrbracket$
 $\implies \forall m \in \text{set } (tl \ ms'). \text{ return-node } m$
 ⟨proof⟩

lemma *silent-moves-intra-path*:

$\llbracket S, f \vdash (m \# ms, s) = as \Rightarrow_{\tau} (m' \# ms', s'); \forall a \in \text{set } as. \text{ intra-kind}(kind \ a) \rrbracket$
 $\implies ms = ms' \wedge \text{get-proc } m = \text{get-proc } m'$
 ⟨proof⟩

lemma *silent-moves-nodestack-notempty*:

$\llbracket S, f \vdash (ms, s) = as \Rightarrow_{\tau} (ms', s'); ms \neq [] \rrbracket \implies ms' \neq []$
 ⟨proof⟩

lemma *silent-moves-obs-slice*:

$\llbracket S, kind \vdash (ms, s) = as \Rightarrow_{\tau} (ms', s'); mx \in \text{obs } ms' \llbracket HRB\text{-slice } S \rrbracket_{CFG};$
 $\forall n \in \text{set } (tl \ ms'). \text{ return-node } n \rrbracket$
 $\implies mx \in \text{obs } ms \llbracket HRB\text{-slice } S \rrbracket_{CFG} \wedge (\forall n \in \text{set } (tl \ ms). \text{ return-node } n)$
 ⟨proof⟩

lemma *silent-moves-empty-obs-slice*:

$\llbracket S, f \vdash (ms, s) = as \Rightarrow_{\tau} (ms', s'); \text{obs } ms' \llbracket HRB\text{-slice } S \rrbracket_{CFG} = \{\} \rrbracket$
 $\implies \text{obs } ms \llbracket HRB\text{-slice } S \rrbracket_{CFG} = \{\}$
 ⟨proof⟩

lemma *silent-moves-preds-transfers*:

assumes $S, f \vdash (ms, s) = as \Rightarrow_{\tau} (ms', s')$
shows *preds* $(\text{map } f \ as) \ s$ **and** *transfers* $(\text{map } f \ as) \ s = s'$
 ⟨proof⟩

lemma *silent-moves-intra-path-obs*:

assumes $m' \in \text{obs-intra } m \llbracket HRB\text{-slice } S \rrbracket_{CFG}$ **and** $\text{length } s = \text{length } (m \# msx')$
and $\forall m \in \text{set } msx'. \text{ return-node } m$
obtains as' **where** $S, \text{slice-kind } S \vdash (m \# msx', s) = as' \Rightarrow_{\tau} (m' \# msx', s)$
 ⟨proof⟩

lemma *silent-moves-intra-path-no-obs*:

assumes $\text{obs-intra } m \llbracket HRB\text{-slice } S \rrbracket_{CFG} = \{\}$ **and** *method-exit* m'
and $\text{get-proc } m = \text{get-proc } m'$ **and** *valid-node* m **and** $\text{length } s = \text{length } (m \# msx')$
and $\forall m \in \text{set } msx'. \text{ return-node } m$

obtains *as* **where** $S, \text{slice-kind } S \vdash (m \# ms', s) = as \Rightarrow_{\tau} (m' \# ms', s)$
 $\langle \text{proof} \rangle$

lemma *silent-moves-vpa-path*:

assumes $S, f \vdash (m \# ms, s) = as \Rightarrow_{\tau} (m' \# ms', s')$ **and** *valid-node* m
and $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i)$
and $ms = \text{targetnodes } rs$ **and** *valid-return-list* rs m
and $\text{length } rs = \text{length } cs$
shows $m -as \rightarrow^* m'$ **and** *valid-path-aux* cs as
 $\langle \text{proof} \rangle$

1.13.2 Observable moves

inductive *observable-move* ::

$'node \text{ SDG-node set} \Rightarrow ('edge \Rightarrow ('var, 'val, 'ret, 'pname) \text{ edge-kind}) \Rightarrow 'node \text{ list}$
 \Rightarrow
 $(('var \rightarrow 'val) \times 'ret) \text{ list} \Rightarrow 'edge \Rightarrow 'node \text{ list} \Rightarrow (('var \rightarrow 'val) \times 'ret) \text{ list} \Rightarrow$
 bool
 $(-, - \vdash '(-, -) \dashrightarrow '(-, -) [51, 50, 0, 0, 50, 0, 0] 51)$

where *observable-move-intra*:

$\llbracket \text{pred } (f \ a) \ s; \text{transfer } (f \ a) \ s = s'; \text{valid-edge } a; \text{intra-kind}(kind \ a);$
 $\forall m \in \text{set } (tl \ ms). \exists m'. \text{call-of-return-node } m \ m' \wedge m' \in \llbracket \text{HRB-slice } S \rrbracket_{CFG};$
 $hd \ ms \in \llbracket \text{HRB-slice } S \rrbracket_{CFG}; \text{length } s' = \text{length } s; \text{length } ms = \text{length } s;$
 $hd \ ms = \text{sourcenode } a; ms' = (\text{targetnode } a) \# tl \ ms \rrbracket$
 $\Rightarrow S, f \vdash (ms, s) -a \rightarrow (ms', s')$

| *observable-move-call*:

$\llbracket \text{pred } (f \ a) \ s; \text{transfer } (f \ a) \ s = s'; \text{valid-edge } a; \text{kind } a = Q: r \leftrightarrow_p f';$
 $\text{valid-edge } a'; a' \in \text{get-return-edges } a;$
 $\forall m \in \text{set } (tl \ ms). \exists m'. \text{call-of-return-node } m \ m' \wedge m' \in \llbracket \text{HRB-slice } S \rrbracket_{CFG};$
 $hd \ ms \in \llbracket \text{HRB-slice } S \rrbracket_{CFG}; \text{length } ms = \text{length } s; \text{length } s' = \text{Suc}(\text{length } s);$
 $hd \ ms = \text{sourcenode } a; ms' = (\text{targetnode } a) \# (\text{targetnode } a') \# tl \ ms \rrbracket$
 $\Rightarrow S, f \vdash (ms, s) -a \rightarrow (ms', s')$

| *observable-move-return*:

$\llbracket \text{pred } (f \ a) \ s; \text{transfer } (f \ a) \ s = s'; \text{valid-edge } a; \text{kind } a = Q \leftarrow_p f';$
 $\forall m \in \text{set } (tl \ ms). \exists m'. \text{call-of-return-node } m \ m' \wedge m' \in \llbracket \text{HRB-slice } S \rrbracket_{CFG};$
 $\text{length } ms = \text{length } s; \text{length } s = \text{Suc}(\text{length } s'); s' \neq [];$
 $hd \ ms = \text{sourcenode } a; hd(tl \ ms) = \text{targetnode } a; ms' = tl \ ms \rrbracket$
 $\Rightarrow S, f \vdash (ms, s) -a \rightarrow (ms', s')$

inductive *observable-moves* ::

$'node \text{ SDG-node set} \Rightarrow ('edge \Rightarrow ('var, 'val, 'ret, 'pname) \text{ edge-kind}) \Rightarrow 'node \text{ list}$
 \Rightarrow
 $(('var \rightarrow 'val) \times 'ret) \text{ list} \Rightarrow 'edge \text{ list} \Rightarrow 'node \text{ list} \Rightarrow (('var \rightarrow 'val) \times 'ret)$

list \Rightarrow *bool*

$(-, - \vdash '(-, -) \Rightarrow '(-, -) [51, 50, 0, 0, 50, 0, 0] 51)$

where *observable-moves-snoc*:

$\llbracket S, f \vdash (ms, s) = as \Rightarrow_{\tau} (ms', s'); S, f \vdash (ms', s') - a \rightarrow (ms'', s'') \rrbracket$
 $\Rightarrow S, f \vdash (ms, s) = as @ [a] \Rightarrow (ms'', s'')$

lemma *observable-move-equal-length*:

assumes $S, f \vdash (ms, s) - a \rightarrow (ms', s')$

shows *length* $ms = \text{length } s$ **and** *length* $ms' = \text{length } s'$

<proof>

lemma *observable-moves-equal-length*:

assumes $S, f \vdash (ms, s) = as \Rightarrow (ms', s')$

shows *length* $ms = \text{length } s$ **and** *length* $ms' = \text{length } s'$

<proof>

lemma *observable-move-notempty*:

$\llbracket S, f \vdash (ms, s) = as \Rightarrow (ms', s'); as = [] \rrbracket \Rightarrow \text{False}$

<proof>

lemma *silent-move-observable-moves*:

$\llbracket S, f \vdash (ms'', s'') = as \Rightarrow (ms', s'); S, f \vdash (ms, s) - a \rightarrow_{\tau} (ms'', s'') \rrbracket$
 $\Rightarrow S, f \vdash (ms, s) = a \# as \Rightarrow (ms', s')$

<proof>

lemma *silent-append-observable-moves*:

$\llbracket S, f \vdash (ms, s) = as \Rightarrow_{\tau} (ms'', s''); S, f \vdash (ms'', s'') = as' \Rightarrow (ms', s') \rrbracket$
 $\Rightarrow S, f \vdash (ms, s) = as @ as' \Rightarrow (ms', s')$

<proof>

lemma *observable-moves-preds-transfers*:

assumes $S, f \vdash (ms, s) = as \Rightarrow (ms', s')$

shows *preds* $(\text{map } f \text{ as}) s$ **and** *transfers* $(\text{map } f \text{ as}) s = s'$

<proof>

lemma *observable-move-vpa-path*:

$\llbracket S, f \vdash (m \# ms, s) - a \rightarrow (m' \# ms', s'); \text{valid-node } m;$

$\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i); ms = \text{targetnodes } rs;$

$\text{valid-return-list } rs \text{ } m; \text{length } rs = \text{length } cs \rrbracket \Rightarrow \text{valid-path-aux } cs [a]$

<proof>

1.13.3 Relevant variables

inductive-set *relevant-vars* ::

'node SDG-node set \Rightarrow *'node SDG-node* \Rightarrow *'var set (rv -)*

for *S* :: *'node SDG-node set* **and** *n* :: *'node SDG-node*

where *rvI*:

\llbracket *parent-node n* $-as \rightarrow_i^*$ *parent-node n'*; *n' ∈ HRB-slice S*; *V ∈ Use_{SDG} n'*;

$\forall n''$. *valid-SDG-node n''* \wedge *parent-node n'' ∈ set (sourcenodes as)*

$\longrightarrow V \notin Def_{SDG} n'' \rrbracket$

$\implies V \in rv\ S\ n$

lemma *rvE*:

assumes *rv*: *V ∈ rv S n*

obtains *as n' where* *parent-node n* $-as \rightarrow_i^*$ *parent-node n'*

and *n' ∈ HRB-slice S* **and** *V ∈ Use_{SDG} n'*

and $\forall n''$. *valid-SDG-node n''* \wedge *parent-node n'' ∈ set (sourcenodes as)*

$\longrightarrow V \notin Def_{SDG} n''$

<proof>

lemma *rv-parent-node*:

parent-node n = parent-node n' \implies rv (S::'node SDG-node set) n = rv S n'

<proof>

lemma *obs-intra-empty-rv-empty*:

assumes *obs-intra m* $\llbracket HRB-slice\ S \rrbracket_{CFG} = \{\}$ **shows** *rv S (CFG-node m) = \{\}*

<proof>

lemma *eq-obs-intra-in-rv*:

assumes *obs-eq:obs-intra (parent-node n)* $\llbracket HRB-slice\ S \rrbracket_{CFG} =$

obs-intra (parent-node n') $\llbracket HRB-slice\ S \rrbracket_{CFG}$

and *x ∈ rv S n* **shows** *x ∈ rv S n'*

<proof>

lemma *closed-eq-obs-eq-rvs*:

fixes *S* :: *'node SDG-node set*

assumes *obs-eq:obs-intra (parent-node n)* $\llbracket HRB-slice\ S \rrbracket_{CFG} =$

obs-intra (parent-node n') $\llbracket HRB-slice\ S \rrbracket_{CFG}$

shows *rv S n = rv S n'*

<proof>

lemma *closed-eq-obs-eq-rvs'*:

fixes *S* :: *'node SDG-node set*

assumes $obs\text{-}eq:obs\text{-}intra\ m\ \llbracket HRB\text{-}slice\ S \rrbracket_{CFG} = obs\text{-}intra\ m'\ \llbracket HRB\text{-}slice\ S \rrbracket_{CFG}$
shows $rv\ S\ (CFG\text{-}node\ m) = rv\ S\ (CFG\text{-}node\ m')$
 $\langle proof \rangle$

lemma *rv-branching-edges-slice-kinds-False*:

assumes *valid-edge a* **and** *valid-edge ax*
and $sourcenode\ a = sourcenode\ ax$ **and** $targetnode\ a \neq targetnode\ ax$
and *intra-kind (kind a)* **and** *intra-kind (kind ax)*
and $preds\ (slice\text{-}kinds\ S\ (a\#as))\ s$
and $preds\ (slice\text{-}kinds\ S\ (ax\#asx))\ s'$
and $length\ s = length\ s'$ **and** $snd\ (hd\ s) = snd\ (hd\ s')$
and $\forall V \in rv\ S\ (CFG\text{-}node\ (sourcenode\ a)).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$
shows *False*
 $\langle proof \rangle$

lemma *rv-edge-slice-kinds*:

assumes *valid-edge a* **and** *intra-kind (kind a)*
and $\forall V \in rv\ S\ (CFG\text{-}node\ (sourcenode\ a)).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$
and $preds\ (slice\text{-}kinds\ S\ (a\#as))\ s$ **and** $preds\ (slice\text{-}kinds\ S\ (a\#asx))\ s'$
shows $\forall V \in rv\ S\ (CFG\text{-}node\ (targetnode\ a)).$
 $state\text{-}val\ (transfer\ (slice\text{-}kind\ S\ a)\ s)\ V =$
 $state\text{-}val\ (transfer\ (slice\text{-}kind\ S\ a)\ s')\ V$
 $\langle proof \rangle$

1.13.4 The weak simulation relational set WS

inductive-set $WS :: 'node\ SDG\text{-}node\ set \Rightarrow (('node\ list \times (('var \rightarrow 'val) \times 'ret)\ list) \times$

$('node\ list \times (('var \rightarrow 'val) \times 'ret)\ list))\ set$

for $S :: 'node\ SDG\text{-}node\ set$

where $WSI: \llbracket \forall m \in set\ ms.\ valid\text{-}node\ m; \forall m' \in set\ ms'.\ valid\text{-}node\ m';$
 $length\ ms = length\ s; length\ ms' = length\ s'; s \neq []; s' \neq []; ms = msx @ mx \# tl\ ms';$

$get\text{-}proc\ mx = get\text{-}proc\ (hd\ ms');$

$\forall m \in set\ (tl\ ms').\ \exists m'.\ call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge m' \in \llbracket HRB\text{-}slice\ S \rrbracket_{CFG};$

$msx \neq [] \longrightarrow (\exists mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \notin \llbracket HRB\text{-}slice\ S \rrbracket_{CFG});$

$\forall i < length\ ms'.\ snd\ (s!(length\ msx + i)) = snd\ (s'!i);$

$\forall m \in set\ (tl\ ms).\ return\text{-}node\ m;$

$\forall i < length\ ms'.\ \forall V \in rv\ S\ (CFG\text{-}node\ ((mx \# tl\ ms')!i)).$

$(fst\ (s!(length\ msx + i)))\ V = (fst\ (s'!i))\ V;$

$obs\ ms\ \llbracket HRB\text{-}slice\ S \rrbracket_{CFG} = obs\ ms'\ \llbracket HRB\text{-}slice\ S \rrbracket_{CFG}$

$\implies ((ms,s),(ms',s')) \in WS\ S$

lemma *WS-silent-move*:

assumes $S, kind \vdash (ms_1, s_1) \text{-}a \rightarrow_\tau (ms_1', s_1')$ **and** $((ms_1, s_1), (ms_2, s_2)) \in WS\ S$

shows $((ms_1', s_1'), (ms_2, s_2)) \in WS\ S$

<proof>

lemma *WS-silent-moves*:

$\llbracket S, kind \vdash (ms_1, s_1) = as \Rightarrow_{\tau} (ms_1', s_1'); ((ms_1, s_1), (ms_2, s_2)) \in WS S \rrbracket$
 $\implies ((ms_1', s_1'), (ms_2, s_2)) \in WS S$

<proof>

lemma *WS-observable-move*:

assumes $((ms_1, s_1), (ms_2, s_2)) \in WS S$
and $S, kind \vdash (ms_1, s_1) -a \rightarrow (ms_1', s_1')$ **and** $s_1' \neq []$
obtains *as* **where** $((ms_1', s_1'), (ms_1', transfer (slice-kind S a) s_2)) \in WS S$
and $S, slice-kind S \vdash (ms_2, s_2) = as@[a] \Rightarrow (ms_1', transfer (slice-kind S a) s_2)$

<proof>

1.13.5 The weak simulation

definition *is-weak-sim* ::

$((node\ list \times ((var \rightarrow val) \times ret)\ list) \times$
 $(node\ list \times ((var \rightarrow val) \times ret)\ list))\ set \Rightarrow node\ SDG\ node\ set \Rightarrow bool$

where *is-weak-sim* $R\ S \equiv$

$\forall ms_1\ s_1\ ms_2\ s_2\ ms_1'\ s_1'\ as.$
 $((ms_1, s_1), (ms_2, s_2)) \in R \wedge S, kind \vdash (ms_1, s_1) = as \Rightarrow (ms_1', s_1') \wedge s_1' \neq []$
 $\longrightarrow (\exists ms_2'\ s_2'\ as'. ((ms_1', s_1'), (ms_2', s_2')) \in R \wedge$
 $S, slice-kind S \vdash (ms_2, s_2) = as' \Rightarrow (ms_2', s_2'))$

lemma *WS-weak-sim*:

assumes $((ms_1, s_1), (ms_2, s_2)) \in WS S$
and $S, kind \vdash (ms_1, s_1) = as \Rightarrow (ms_1', s_1')$ **and** $s_1' \neq []$
obtains *as'* **where** $((ms_1', s_1'), (ms_1', transfer (slice-kind S (last as)) s_2)) \in WS S$
and $S, slice-kind S \vdash (ms_2, s_2) = as'@[last\ as] \Rightarrow$
 $(ms_1', transfer (slice-kind S (last as)) s_2)$

<proof>

The following lemma states the correctness of static intraprocedural slicing:
the simulation *WS S* is a desired weak simulation

theorem *WS-is-weak-sim:is-weak-sim* (*WS S*) *S*

<proof>

end

end

1.14 The fundamental property of slicing

theory *FundamentalProperty* **imports** *WeakSimulation SemanticsCFG* **begin**

context *SDG* begin

1.14.1 Auxiliary lemmas for moves in the graph

lemma *observable-set-stack-in-slice*:

$S, f \vdash (ms, s) -a \rightarrow (ms', s')$
 $\implies \forall mx \in \text{set } (tl \ ms'). \exists mx'. \text{call-of-return-node } mx \ mx' \wedge mx' \in \lfloor \text{HRB-slice}$
 $S \rfloor_{CFG}$
(*proof*)

lemma *silent-move-preserves-stacks*:

assumes $S, f \vdash (m \# ms, s) -a \rightarrow_{\tau} (m' \# ms', s')$ **and** *valid-call-list* $cs \ m$
and $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i)$ **and** *valid-return-list* $rs \ m$
and $\text{length } rs = \text{length } cs$ **and** $ms = \text{targetnodes } rs$
obtains $cs' \ rs'$ **where** *valid-node* m' **and** *valid-call-list* $cs' \ m'$
and $\forall i < \text{length } rs'. rs'!i \in \text{get-return-edges } (cs'!i)$
and *valid-return-list* $rs' \ m'$ **and** $\text{length } rs' = \text{length } cs'$
and $ms' = \text{targetnodes } rs'$ **and** $\text{upd-cs } cs \ [a] = cs'$
(*proof*)

lemma *silent-moves-preserves-stacks*:

assumes $S, f \vdash (m \# ms, s) =as \Rightarrow_{\tau} (m' \# ms', s')$
and *valid-node* m **and** *valid-call-list* $cs \ m$
and $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i)$ **and** *valid-return-list* $rs \ m$
and $\text{length } rs = \text{length } cs$ **and** $ms = \text{targetnodes } rs$
obtains $cs' \ rs'$ **where** *valid-node* m' **and** *valid-call-list* $cs' \ m'$
and $\forall i < \text{length } rs'. rs'!i \in \text{get-return-edges } (cs'!i)$
and *valid-return-list* $rs' \ m'$ **and** $\text{length } rs' = \text{length } cs'$
and $ms' = \text{targetnodes } rs'$ **and** $\text{upd-cs } cs \ as = cs'$
(*proof*)

lemma *observable-move-preserves-stacks*:

assumes $S, f \vdash (m \# ms, s) -a \rightarrow (m' \# ms', s')$ **and** *valid-call-list* $cs \ m$
and $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i)$ **and** *valid-return-list* $rs \ m$
and $\text{length } rs = \text{length } cs$ **and** $ms = \text{targetnodes } rs$
obtains $cs' \ rs'$ **where** *valid-node* m' **and** *valid-call-list* $cs' \ m'$
and $\forall i < \text{length } rs'. rs'!i \in \text{get-return-edges } (cs'!i)$
and *valid-return-list* $rs' \ m'$ **and** $\text{length } rs' = \text{length } cs'$
and $ms' = \text{targetnodes } rs'$ **and** $\text{upd-cs } cs \ [a] = cs'$
(*proof*)

lemma *observable-moves-preserves-stack*:

assumes $S, f \vdash (m \# ms, s) =as \Rightarrow (m' \# ms', s')$
and *valid-node* m **and** *valid-call-list* $cs \ m$

and $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i)$ **and** *valid-return-list* rs m
and $\text{length } rs = \text{length } cs$ **and** $ms = \text{targetnodes } rs$
obtains $cs' rs'$ **where** *valid-node* m' **and** *valid-call-list* $cs' m'$
and $\forall i < \text{length } rs'. rs'!i \in \text{get-return-edges } (cs'!i)$
and *valid-return-list* $rs' m'$ **and** $\text{length } rs' = \text{length } cs'$
and $ms' = \text{targetnodes } rs'$ **and** $\text{upd-cs } cs \text{ as} = cs'$
 ⟨*proof*⟩

lemma *silent-moves-slp-path*:

$\llbracket S, f \vdash (m \# ms'' @ ms, s) = as \Rightarrow_{\tau} (m' \# ms', s'); \text{valid-node } m; \text{valid-call-list } cs \ m;$
 $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i); \text{valid-return-list } rs \ m;$
 $\text{length } rs = \text{length } cs; ms'' = \text{targetnodes } rs;$
 $\forall mx \in \text{set } ms. \exists mx'. \text{call-of-return-node } mx \ mx' \wedge mx' \in \llbracket \text{HRB-slice } S \rrbracket_{CFG};$
 $ms'' \neq [] \longrightarrow (\exists mx'. \text{call-of-return-node } (\text{last } ms'') \ mx' \wedge mx' \notin \llbracket \text{HRB-slice } S \rrbracket_{CFG});$
 $\forall mx \in \text{set } ms'. \exists mx'. \text{call-of-return-node } mx \ mx' \wedge mx' \in \llbracket \text{HRB-slice } S \rrbracket_{CFG}$
 $\implies \text{same-level-path-aux } cs \text{ as} \wedge \text{upd-cs } cs \text{ as} = [] \wedge m \text{ -as} \rightarrow^* m' \wedge ms = ms'$
 ⟨*proof*⟩

lemma *silent-moves-slp*:

$\llbracket S, f \vdash (m \# ms, s) = as \Rightarrow_{\tau} (m' \# ms', s'); \text{valid-node } m;$
 $\forall mx \in \text{set } ms. \exists mx'. \text{call-of-return-node } mx \ mx' \wedge mx' \in \llbracket \text{HRB-slice } S \rrbracket_{CFG};$
 $\forall mx \in \text{set } ms'. \exists mx'. \text{call-of-return-node } mx \ mx' \wedge mx' \in \llbracket \text{HRB-slice } S \rrbracket_{CFG}$
 $\implies m \text{ -as} \rightarrow_{sl}^* m' \wedge ms = ms'$
 ⟨*proof*⟩

lemma *slpa-silent-moves-callstacks-eq*:

$\llbracket \text{same-level-path-aux } cs \text{ as}; S, f \vdash (m \# msx @ ms, s) = as \Rightarrow_{\tau} (m' \# ms', s');$
 $\text{length } ms = \text{length } ms'; \text{valid-call-list } cs \ m;$
 $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i); \text{valid-return-list } rs \ m;$
 $\text{length } rs = \text{length } cs; msx = \text{targetnodes } rs$
 $\implies ms = ms'$
 ⟨*proof*⟩

lemma *silent-moves-same-level-path*:

assumes $S, kind \vdash (m \# ms, s) = as \Rightarrow_{\tau} (m' \# ms', s')$ **and** $m \text{ -as} \rightarrow_{sl}^* m'$ **shows**
 $ms = ms'$
 ⟨*proof*⟩

lemma *silent-moves-call-edge*:

assumes $S, kind \vdash (m \# ms, s) = as \Rightarrow_{\tau} (m' \# ms', s')$ **and** *valid-node* m
and *callstack*: $\forall mx \in \text{set } ms. \exists mx'. \text{call-of-return-node } mx \ mx' \wedge$
 $mx' \in \llbracket \text{HRB-slice } S \rrbracket_{CFG}$
and *rest*: $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i)$

$ms = \text{targetnodes } rs \text{ valid-return-list } rs \ m \ \text{length } rs = \text{length } cs$
obtains $as' \ a \ as''$ **where** $as = as'@a\#as''$ **and** $\exists Q \ r \ p \ fs. \text{kind } a = Q:r\hookrightarrow pfs$
and $\text{call-of-return-node } (hd \ ms')$ ($\text{sourcenode } a$)
and $\text{targetnode } a \ -as'' \rightarrow_{sl^*} m'$
 $| \ ms' = ms$
 $\langle \text{proof} \rangle$

lemma *silent-moves-called-node-in-slice1-hd-nodestack-in-slice1*:
assumes $S, \text{kind} \vdash (m\#ms, s) = as \Rightarrow_{\tau} (m'\#ms', s')$ **and** $\text{valid-node } m$
and $\text{CFG-node } m' \in \text{sum-SDG-slice1 } nx$
and $\forall mx \in \text{set } ms. \exists mx'. \text{call-of-return-node } mx \ mx' \wedge$
 $mx' \in \lfloor \text{HRB-slice } S \rfloor_{\text{CFG}}$
and $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i)$ **and** $ms = \text{targetnodes } rs$
and $\text{valid-return-list } rs \ m$ **and** $\text{length } rs = \text{length } cs$
obtains $as' \ a \ as''$ **where** $as = as'@a\#as''$ **and** $\exists Q \ r \ p \ fs. \text{kind } a = Q:r\hookrightarrow pfs$
and $\text{call-of-return-node } (hd \ ms')$ ($\text{sourcenode } a$)
and $\text{targetnode } a \ -as'' \rightarrow_{sl^*} m'$ **and** $\text{CFG-node } (\text{sourcenode } a) \in \text{sum-SDG-slice1}$
 nx
 $| \ ms' = ms$
 $\langle \text{proof} \rangle$

lemma *silent-moves-called-node-in-slice1-nodestack-in-slice1*:
 $\llbracket S, \text{kind} \vdash (m\#ms, s) = as \Rightarrow_{\tau} (m'\#ms', s'); \text{valid-node } m;$
 $\text{CFG-node } m' \in \text{sum-SDG-slice1 } nx; nx \in S;$
 $\forall mx \in \text{set } ms. \exists mx'. \text{call-of-return-node } mx \ mx' \wedge mx' \in \lfloor \text{HRB-slice } S \rfloor_{\text{CFG}};$
 $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i); ms = \text{targetnodes } rs;$
 $\text{valid-return-list } rs \ m; \text{length } rs = \text{length } cs \rrbracket$
 $\implies \forall mx \in \text{set } ms'. \exists mx'. \text{call-of-return-node } mx \ mx' \wedge mx' \in \lfloor \text{HRB-slice } S \rfloor_{\text{CFG}}$
 $\langle \text{proof} \rangle$

lemma *silent-moves-slice-intra-path*:
assumes $S, \text{slice-kind } S \vdash (m\#ms, s) = as \Rightarrow_{\tau} (m'\#ms', s')$
and $\forall mx \in \text{set } ms. \exists mx'. \text{call-of-return-node } mx \ mx' \wedge mx' \in \lfloor \text{HRB-slice } S \rfloor_{\text{CFG}}$
shows $\forall a \in \text{set } as. \text{intra-kind } (\text{kind } a)$
 $\langle \text{proof} \rangle$

lemma *silent-moves-slice-keeps-state*:
assumes $S, \text{slice-kind } S \vdash (m\#ms, s) = as \Rightarrow_{\tau} (m'\#ms', s')$
and $\forall mx \in \text{set } ms. \exists mx'. \text{call-of-return-node } mx \ mx' \wedge mx' \in \lfloor \text{HRB-slice } S \rfloor_{\text{CFG}}$
shows $s = s'$
 $\langle \text{proof} \rangle$

1.14.2 Definition of slice-edges

definition $\text{slice-edge} :: 'node \ \text{SDG-node set} \Rightarrow 'edge \ \text{list} \Rightarrow 'edge \Rightarrow \text{bool}$

where $\text{slice-edge } S \text{ cs } a \equiv (\forall c \in \text{set cs. sourcenode } c \in \llbracket \text{HRB-slice } S \rrbracket_{CFG}) \wedge$
 $(\text{case } (\text{kind } a) \text{ of } Q \leftrightarrow pf \Rightarrow \text{True} \mid - \Rightarrow \text{sourcenode } a \in \llbracket \text{HRB-slice } S \rrbracket_{CFG})$

lemma *silent-move-no-slice-edge*:

$\llbracket S, f \vdash (ms, s) -a \rightarrow_{\tau} (ms', s'); \text{tl } ms = \text{targetnodes } rs; \text{length } rs = \text{length } cs;$
 $\forall i < \text{length } cs. \text{call-of-return-node } (\text{tl } ms!i) (\text{sourcenode } (cs!i)) \rrbracket$
 $\implies \neg \text{slice-edge } S \text{ cs } a$
 $\langle \text{proof} \rangle$

lemma *observable-move-slice-edge*:

$\llbracket S, f \vdash (ms, s) -a \rightarrow (ms', s'); \text{tl } ms = \text{targetnodes } rs; \text{length } rs = \text{length } cs;$
 $\forall i < \text{length } cs. \text{call-of-return-node } (\text{tl } ms!i) (\text{sourcenode } (cs!i)) \rrbracket$
 $\implies \text{slice-edge } S \text{ cs } a$
 $\langle \text{proof} \rangle$

function $\text{slice-edges} :: 'node \text{SDG-node set} \Rightarrow 'edge \text{list} \Rightarrow 'edge \text{list} \Rightarrow 'edge \text{list}$

where $\text{slice-edges } S \text{ cs } [] = []$

$\mid \text{slice-edge } S \text{ cs } a \implies$

$\text{slice-edges } S \text{ cs } (a \# as) = a \# \text{slice-edges } S (\text{upd-cs } cs [a]) as$

$\mid \neg \text{slice-edge } S \text{ cs } a \implies$

$\text{slice-edges } S \text{ cs } (a \# as) = \text{slice-edges } S (\text{upd-cs } cs [a]) as$

$\langle \text{proof} \rangle$

termination $\langle \text{proof} \rangle$

lemma *slice-edges-Append*:

$\llbracket \text{slice-edges } S \text{ cs } as = as'; \text{slice-edges } S (\text{upd-cs } cs as) asx = asx' \rrbracket$
 $\implies \text{slice-edges } S \text{ cs } (as @ asx) = as' @ asx'$
 $\langle \text{proof} \rangle$

lemma *slice-edges-Nil-split*:

$\text{slice-edges } S \text{ cs } (as @ as') = []$
 $\implies \text{slice-edges } S \text{ cs } as = [] \wedge \text{slice-edges } S (\text{upd-cs } cs as) as' = []$
 $\langle \text{proof} \rangle$

lemma *slice-intra-edges-no-nodes-in-slice*:

$\llbracket \text{slice-edges } S \text{ cs } as = []; \forall a \in \text{set } as. \text{intra-kind } (\text{kind } a);$
 $\forall c \in \text{set } cs. \text{sourcenode } c \in \llbracket \text{HRB-slice } S \rrbracket_{CFG} \rrbracket$
 $\implies \forall nx \in \text{set}(\text{sourcenodes } as). nx \notin \llbracket \text{HRB-slice } S \rrbracket_{CFG}$
 $\langle \text{proof} \rangle$

lemma *silent-moves-no-slice-edges*:

$\llbracket S, f \vdash (ms, s) = as \Rightarrow_{\tau} (ms', s'); tl\ ms = targetnodes\ rs; length\ rs = length\ cs;$
 $\forall i < length\ cs. call-of-return-node\ (tl\ ms!i)\ (sourcenode\ (cs!i)) \rrbracket$
 $\implies slice-edges\ S\ cs\ as = [] \wedge (\exists rs'. tl\ ms' = targetnodes\ rs' \wedge$
 $length\ rs' = length\ (upd-cs\ cs\ as) \wedge (\forall i < length\ (upd-cs\ cs\ as).$
 $call-of-return-node\ (tl\ ms'!i)\ (sourcenode\ ((upd-cs\ cs\ as)!i))))$
 <proof>

lemma *observable-moves-singular-slice-edge:*

$\llbracket S, f \vdash (ms, s) = as \Rightarrow (ms', s'); tl\ ms = targetnodes\ rs; length\ rs = length\ cs;$
 $\forall i < length\ cs. call-of-return-node\ (tl\ ms!i)\ (sourcenode\ (cs!i)) \rrbracket$
 $\implies slice-edges\ S\ cs\ as = [last\ as]$
 <proof>

lemma *silent-moves-nonempty-nodestack-False:*

assumes $S, kind \vdash ([m], [cf]) = as \Rightarrow_{\tau} (m' \# ms', s')$ **and** *valid-node* m
and $ms' \neq []$ **and** *CFG-node* $m' \in sum-SDG-slice1\ nx$ **and** $nx \in S$
shows *False*
 <proof>

lemma *transfers-intra-slice-kinds-slice-edges:*

$\llbracket \forall a \in set\ as. intra-kind\ (kind\ a); \forall c \in set\ cs. sourcenode\ c \in [HRB-slice\ S]_{CFG} \rrbracket$
 $\implies transfers\ (slice-kinds\ S\ (slice-edges\ S\ cs\ as))\ s =$
 $transfers\ (slice-kinds\ S\ as)\ s$
 <proof>

lemma *exists-sliced-intra-path-preds:*

assumes $m - as \rightarrow_{\iota} * m'$ **and** $slice-edges\ S\ cs\ as = []$
and $m' \in [HRB-slice\ S]_{CFG}$ **and** $\forall c \in set\ cs. sourcenode\ c \in [HRB-slice\ S]_{CFG}$
obtains as' **where** $m - as' \rightarrow_{\iota} * m'$ **and** $preds\ (slice-kinds\ S\ as')\ (cf \# cfs)$
and $slice-edges\ S\ cs\ as' = []$
 <proof>

lemma *slp-to-intra-path-with-slice-edges:*

assumes $n - as \rightarrow_{sl} * n'$ **and** $slice-edges\ S\ cs\ as = []$
obtains as' **where** $n - as' \rightarrow_{\iota} * n'$ **and** $slice-edges\ S\ cs\ as' = []$
 <proof>

1.14.3 $S, f \vdash (ms, s) = as \Rightarrow^* (ms', s')$: the reflexive transitive closure of $S, f \vdash (ms, s) = as \Rightarrow (ms', s')$

inductive *trans-observable-moves* ::

$'node \text{ SDG-node set} \Rightarrow ('edge \Rightarrow ('var, 'val, 'ret, 'pname) \text{ edge-kind}) \Rightarrow 'node \text{ list}$
 \Rightarrow
 $(('var \rightarrow 'val) \times 'ret) \text{ list} \Rightarrow 'edge \text{ list} \Rightarrow 'node \text{ list} \Rightarrow$
 $(('var \rightarrow 'val) \times 'ret) \text{ list} \Rightarrow \text{bool}$
 $(-, - \vdash '(-, -) \Rightarrow^* '(-, -) [51, 50, 0, 0, 50, 0, 0] 51)$

where *tom-Nil*:

$\text{length } ms = \text{length } s \Rightarrow S, f \vdash (ms, s) = [] \Rightarrow^* (ms, s)$

| *tom-Cons*:

$\llbracket S, f \vdash (ms, s) = as \Rightarrow (ms', s'); S, f \vdash (ms', s') = as' \Rightarrow^* (ms'', s'') \rrbracket$
 $\Rightarrow S, f \vdash (ms, s) = (last \ as) \# as' \Rightarrow^* (ms'', s'')$

lemma *tom-split-snoc*:

assumes $S, f \vdash (ms, s) = as \Rightarrow^* (ms', s')$ **and** $as \neq []$
obtains $asx \ asx' \ ms'' \ s''$ **where** $as = asx @ [last \ asx']$
and $S, f \vdash (ms, s) = asx \Rightarrow^* (ms'', s'')$ **and** $S, f \vdash (ms'', s'') = asx' \Rightarrow (ms', s')$
 $\langle \text{proof} \rangle$

lemma *tom-preserves-stacks*:

assumes $S, f \vdash (m \# ms, s) = as \Rightarrow^* (m' \# ms', s')$ **and** *valid-node* m
and *valid-call-list* $cs \ m$ **and** $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i)$
and *valid-return-list* $rs \ m$ **and** $\text{length } rs = \text{length } cs$ **and** $ms = \text{targetnodes } rs$
obtains $cs' \ rs'$ **where** *valid-node* m' **and** *valid-call-list* $cs' \ m'$
and $\forall i < \text{length } rs'. rs'!i \in \text{get-return-edges } (cs'!i)$
and *valid-return-list* $rs' \ m'$ **and** $\text{length } rs' = \text{length } cs'$
and $ms' = \text{targetnodes } rs'$
 $\langle \text{proof} \rangle$

lemma *vpa-trans-observable-moves*:

assumes *valid-path-aux* $cs \ as$ **and** $m - as \rightarrow^* m'$ **and** *preds* $(\text{kinds } as) \ s$
and *transfers* $(\text{kinds } as) \ s = s'$ **and** *valid-call-list* $cs \ m$
and $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i)$
and *valid-return-list* $rs \ m$
and $\text{length } rs = \text{length } cs$ **and** $s = \text{Suc } (\text{length } cs)$
obtains $ms \ ms'' \ s'' \ ms' \ as' \ as''$
where $S, kind \vdash (m \# ms, s) = \text{slice-edges } S \ cs \ as \Rightarrow^* (ms'', s'')$
and $S, kind \vdash (ms'', s'') = as' \Rightarrow_{\tau} (m' \# ms', s')$
and $ms = \text{targetnodes } rs$ **and** $\text{length } ms = \text{length } cs$
and $\forall i < \text{length } cs. \text{call-of-return-node } (ms!i) (\text{sourcenode } (cs!i))$
and $\text{slice-edges } S \ cs \ as = \text{slice-edges } S \ cs \ as''$
and $m - as'' @ as' \rightarrow^* m'$ **and** *valid-path-aux* $cs \ (as'' @ as')$
 $\langle \text{proof} \rangle$

lemma *valid-path-trans-observable-moves*:

assumes $m -as \rightarrow_{\sqrt{*}} m'$ **and** $\text{preds } (\text{kinds } as) [cf]$
and $\text{transfers } (\text{kinds } as) [cf] = s'$
obtains $ms'' s'' ms' as' as''$
where $S, \text{kind} \vdash ([m], [cf]) = \text{slice-edges } S \square as \Rightarrow * (ms'', s'')$
and $S, \text{kind} \vdash (ms'', s'') = as' \Rightarrow_{\tau} (m' \# ms', s')$
and $\text{slice-edges } S \square as = \text{slice-edges } S \square as''$
and $m -as'' @ as' \rightarrow_{\sqrt{*}} m'$
 $\langle \text{proof} \rangle$

lemma *WS-weak-sim-trans*:

assumes $((ms_1, s_1), (ms_2, s_2)) \in WS S$
and $S, \text{kind} \vdash (ms_1, s_1) = as \Rightarrow * (ms_1', s_1')$ **and** $as \neq \square$
shows $((ms_1', s_1'), (ms_1', \text{transfers } (\text{slice-kinds } S as) s_2)) \in WS S \wedge$
 $S, \text{slice-kind } S \vdash (ms_2, s_2) = as \Rightarrow * (ms_1', \text{transfers } (\text{slice-kinds } S as) s_2)$
 $\langle \text{proof} \rangle$

lemma *stacks-rewrite*:

assumes *valid-call-list* $cs m$ **and** *valid-return-list* $rs m$
and $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i)$
and $\text{length } rs = \text{length } cs$ **and** $ms = \text{targetnodes } rs$
shows $\forall i < \text{length } cs. \text{call-of-return-node } (ms!i) (\text{sourcenode } (cs!i))$
 $\langle \text{proof} \rangle$

lemma *slice-tom-preds-vp*:

assumes $S, \text{slice-kind } S \vdash (m \# ms, s) = as \Rightarrow * (m' \# ms', s')$ **and** *valid-node* m
and *valid-call-list* $cs m$ **and** $\forall i < \text{length } rs. rs!i \in \text{get-return-edges } (cs!i)$
and *valid-return-list* $rs m$ **and** $\text{length } rs = \text{length } cs$ **and** $ms = \text{targetnodes } rs$
and $\forall mx \in \text{set } ms. \exists mx'. \text{call-of-return-node } mx mx' \wedge mx' \in [HRB\text{-slice } S]_{CFG}$
obtains $as' cs' rs'$ **where** $\text{preds } (\text{slice-kinds } S as') s$
and $\text{slice-edges } S cs as' = as$ **and** $m -as' \rightarrow_{\sqrt{*}} m'$ **and** *valid-path-aux* $cs as'$
and $\text{upd-cs } cs as' = cs'$ **and** *valid-node* m' **and** *valid-call-list* $cs' m'$
and $\forall i < \text{length } rs'. rs'!i \in \text{get-return-edges } (cs'!i)$
and *valid-return-list* $rs' m'$ **and** $\text{length } rs' = \text{length } cs'$
and $ms' = \text{targetnodes } rs'$ **and** $\text{transfers } (\text{slice-kinds } S as') s \neq \square$
and $\text{transfers } (\text{slice-kinds } S (\text{slice-edges } S cs as')) s =$
 $\text{transfers } (\text{slice-kinds } S as') s$
 $\langle \text{proof} \rangle$

1.14.4 The fundamental property of static interprocedural slicing

theorem *fundamental-property-of-static-slicing*:

assumes $m -as \rightarrow_{\sqrt{*}} m'$ **and** $\text{preds } (\text{kinds } as) [cf]$ **and** *CFG-node* $m' \in S$

```

obtains  $as'$  where  $preds$  ( $slice-kinds$   $S$   $as'$ ) [ $cf$ ]
and  $\forall V \in Use$   $m'$ .  $state-val$  ( $transfers$  ( $slice-kinds$   $S$   $as'$ ) [ $cf$ ])  $V =$ 
     $state-val$  ( $transfers$  ( $kinds$   $as$ ) [ $cf$ ])  $V$ 
and  $slice-edges$   $S$  []  $as = slice-edges$   $S$  []  $as'$ 
and  $transfers$  ( $kinds$   $as$ ) [ $cf$ ]  $\neq$  [] and  $m -as' \rightarrow_{\surd}^* m'$ 
 $\langle proof \rangle$ 

end

```

1.14.5 The fundamental property of static interprocedural slicing related to the semantics

```

locale  $SemanticsProperty = SDG$   $sourcenode$   $targetnode$   $kind$   $valid-edge$   $Entry$ 
     $get-proc$   $get-return-edges$   $procs$   $Main$   $Exit$   $Def$   $Use$   $ParamDefs$   $ParamUses$  +
     $CFG-semantics-wf$   $sourcenode$   $targetnode$   $kind$   $valid-edge$   $Entry$ 
     $get-proc$   $get-return-edges$   $procs$   $Main$   $sem$   $identifies$ 
for  $sourcenode$  ::  $'edge \Rightarrow 'node$  and  $targetnode$  ::  $'edge \Rightarrow 'node$ 
and  $kind$  ::  $'edge \Rightarrow ('var, 'val, 'ret, 'pname)$   $edge-kind$ 
and  $valid-edge$  ::  $'edge \Rightarrow bool$ 
and  $Entry$  ::  $'node$  ( $'(-Entry'-)$ ) and  $get-proc$  ::  $'node \Rightarrow 'pname$ 
and  $get-return-edges$  ::  $'edge \Rightarrow 'edge$   $set$ 
and  $procs$  ::  $('pname \times 'var$   $list \times 'var$   $list)$   $list$  and  $Main$  ::  $'pname$ 
and  $Exit$ :: $'node$  ( $'(-Exit'-)$ )
and  $Def$  ::  $'node \Rightarrow 'var$   $set$  and  $Use$  ::  $'node \Rightarrow 'var$   $set$ 
and  $ParamDefs$  ::  $'node \Rightarrow 'var$   $list$  and  $ParamUses$  ::  $'node \Rightarrow 'var$   $set$   $list$ 
and  $sem$  ::  $'com \Rightarrow ('var \rightarrow 'val)$   $list \Rightarrow 'com \Rightarrow ('var \rightarrow 'val)$   $list \Rightarrow bool$ 
    ( $((1 \langle -,/- \rangle) \Rightarrow / (1 \langle -,/- \rangle))$  [ $0,0,0,0$ ] 81)
and  $identifies$  ::  $'node \Rightarrow 'com \Rightarrow bool$  ( $- \triangleq -$  [ $51,0$ ] 80)
begin

```

```

theorem  $fundamental-property-of-path-slicing-semantically:$ 
    assumes  $m \triangleq c$  and  $\langle c, [cf] \rangle \Rightarrow \langle c', s' \rangle$ 
    obtains  $m'$   $as$   $cfs'$  where  $m -as \rightarrow_{\surd}^* m'$  and  $m' \triangleq c'$ 
    and  $preds$  ( $slice-kinds$   $\{CFG-node$   $m'\}$   $as$ ) [ $cf, undefined$ ]
    and  $\forall V \in Use$   $m'$ .
         $state-val$  ( $transfers$  ( $slice-kinds$   $\{CFG-node$   $m'\}$   $as$ ) [ $cf, undefined$ ])  $V =$ 
         $state-val$   $cfs'$   $V$  and  $map$   $fst$   $cfs' = s'$ 
     $\langle proof \rangle$ 

```

end

end

Chapter 2

Instantiating the Framework with a simple While-Language using procedures

2.1 Commands

```
theory Com imports ../StaticInter/BasicDefs begin
```

2.1.1 Variables and Values

```
type-synonym vname = string — names for variables
```

```
type-synonym pname = string — names for procedures
```

```
datatype val  
  = Bool bool      — Boolean value  
  | Intg int       — integer value
```

```
abbreviation true == Bool True
```

```
abbreviation false == Bool False
```

2.1.2 Expressions

```
datatype bop = Eq | And | Less | Add | Sub — names of binary operations
```

```
datatype expr  
  = Val val          — value  
  | Var vname       — local variable  
  | BinOp expr bop expr  (- «-» - [80,0,81] 80) — binary operation
```

```
fun binop :: bop ⇒ val ⇒ val ⇒ val option
```

```

where binop Eq v1 v2 = Some(Bool(v1 = v2))
| binop And (Bool b1) (Bool b2) = Some(Bool(b1 ∧ b2))
| binop Less (Intg i1) (Intg i2) = Some(Bool(i1 < i2))
| binop Add (Intg i1) (Intg i2) = Some(Intg(i1 + i2))
| binop Sub (Intg i1) (Intg i2) = Some(Intg(i1 - i2))
| binop bop v1 v2 = None

```

2.1.3 Commands

```

datatype cmd
= Skip
| LAss vname expr (· := - [70,70] 70) — local assignment
| Seq cmd cmd (· ;; - [60,61] 60)
| Cond expr cmd cmd (if '(-) - / else - [80,79,79] 70)
| While expr cmd (while '(-) - [80,79] 70)
| Call pname expr list vname list
— Call needs procedure, actual parameters and variables for return values

```

```

fun num-inner-nodes :: cmd ⇒ nat (#:-)
where #:Skip = 1
| #:(V:=e) = 2
| #:(c1;;c2) = #:c1 + #:c2
| #:(if (b) c1 else c2) = #:c1 + #:c2 + 1
| #:(while (b) c) = #:c + 2
| #:(Call p es rets) = 2

```

```

lemma num-inner-nodes-gr-0 [simp]:#:c > 0
⟨proof⟩

```

```

lemma [dest]:#:c = 0 ⇒ False
⟨proof⟩

```

end

2.2 The state

```

theory ProcState imports Com begin

```

```

fun interpret :: expr ⇒ (vname → val) ⇒ val option
where Val: interpret (Val v) cf = Some v
| Var: interpret (Var V) cf = cf V
| BinOp: interpret (e1«bop»e2) cf =
(case interpret e1 cf of None ⇒ None
| Some v1 ⇒ (case interpret e2 cf of None ⇒ None
| Some v2 ⇒ (

```

case binop bop v1 v2 of None => None | Some v => Some v)))

abbreviation *update* :: (*vname* \rightarrow *val*) \Rightarrow *vname* \Rightarrow *expr* \Rightarrow (*vname* \rightarrow *val*)
where *update cf V e* \equiv *cf*(*V:=*(*interpret e cf*))

abbreviation *state-check* :: (*vname* \rightarrow *val*) \Rightarrow *expr* \Rightarrow *val option* \Rightarrow *bool*
where *state-check cf b v* \equiv (*interpret b cf = v*)

end

2.3 Definition of the CFG

theory *PCFG* **imports** *ProcState* **begin**

definition *Main* :: *pname*
where *Main* = "Main"

datatype *label* = *Label nat* | *Entry* | *Exit*

2.3.1 The CFG for every procedure

Definition of \oplus

fun *label-incr* :: *label* \Rightarrow *nat* \Rightarrow *label* (*-* \oplus *-* 60)
where (*Label l*) \oplus *i* = *Label (l + i)*
| *Entry* \oplus *i* = *Entry*
| *Exit* \oplus *i* = *Exit*

lemma *Exit-label-incr* [*dest*]: *Exit* = *n* \oplus *i* \Longrightarrow *n* = *Exit*
 \langle *proof* \rangle

lemma *label-incr-Exit* [*dest*]: *n* \oplus *i* = *Exit* \Longrightarrow *n* = *Exit*
 \langle *proof* \rangle

lemma *Entry-label-incr* [*dest*]: *Entry* = *n* \oplus *i* \Longrightarrow *n* = *Entry*
 \langle *proof* \rangle

lemma *label-incr-Entry* [*dest*]: *n* \oplus *i* = *Entry* \Longrightarrow *n* = *Entry*
 \langle *proof* \rangle

lemma *label-incr-inj*:
n \oplus *c* = *n'* \oplus *c* \Longrightarrow *n* = *n'*
 \langle *proof* \rangle

lemma *label-incr-simp*: *n* \oplus *i* = *m* \oplus (*i* + *j*) \Longrightarrow *n* = *m* \oplus *j*
 \langle *proof* \rangle

lemma *label-incr-simp-rev*: $m \oplus (j + i) = n \oplus i \implies m \oplus j = n$
 ⟨proof⟩

lemma *label-incr-start-Node-smaller*:
 $Label\ l = n \oplus i \implies n = Label\ (l - i)$
 ⟨proof⟩

lemma *label-incr-start-Node-smaller-rev*:
 $n \oplus i = Label\ l \implies n = Label\ (l - i)$
 ⟨proof⟩

lemma *label-incr-ge*: $Label\ l = n \oplus i \implies l \geq i$
 ⟨proof⟩

lemma *label-incr-0* [dest]:
 $\llbracket Label\ 0 = n \oplus i; i > 0 \rrbracket \implies False$
 ⟨proof⟩

lemma *label-incr-0-rev* [dest]:
 $\llbracket n \oplus i = Label\ 0; i > 0 \rrbracket \implies False$
 ⟨proof⟩

The edges of the procedure CFG

Control flow information in this language is the node, to which we return after the called procedure is finished.

datatype *p-edge-kind* =
IEdge (*vname*, *val*, *pname* × *label*, *pname*) *edge-kind*
 | *CEdge* *pname* × *expr list* × *vname list*

type-synonym *p-edge* = (*label* × *p-edge-kind* × *label*)

inductive *Proc-CFG* :: *cmd* ⇒ *label* ⇒ *p-edge-kind* ⇒ *label* ⇒ *bool*
 (- ⊢ - →_p -)

where

Proc-CFG-Entry-Exit:
 $prog \vdash Entry - IEdge\ (\lambda s. False)_{\surd \rightarrow_p} Exit$

| *Proc-CFG-Entry*:
 $prog \vdash Entry - IEdge\ (\lambda s. True)_{\surd \rightarrow_p} Label\ 0$

| *Proc-CFG-Skip*:
 $Skip \vdash Label\ 0 - IEdge\ \uparrow id_{\rightarrow_p} Exit$

| *Proc-CFG-LAss*:
 $V := e \vdash Label\ 0 - IEdge\ \uparrow (\lambda cf. update\ cf\ V\ e)_{\rightarrow_p} Label\ 1$

| *Proc-CFG-LAssSkip*:
 $V := e \vdash \text{Label } 1 \text{ -IEdge } \uparrow \text{id} \rightarrow_p \text{Exit}$

| *Proc-CFG-SeqFirst*:
 $\llbracket c_1 \vdash n \text{ -et} \rightarrow_p n'; n' \neq \text{Exit} \rrbracket \implies c_1;;c_2 \vdash n \text{ -et} \rightarrow_p n'$

| *Proc-CFG-SeqConnect*:
 $\llbracket c_1 \vdash n \text{ -et} \rightarrow_p \text{Exit}; n \neq \text{Entry} \rrbracket \implies c_1;;c_2 \vdash n \text{ -et} \rightarrow_p \text{Label } \#:c_1$

| *Proc-CFG-SeqSecond*:
 $\llbracket c_2 \vdash n \text{ -et} \rightarrow_p n'; n \neq \text{Entry} \rrbracket \implies c_1;;c_2 \vdash n \oplus \#:c_1 \text{ -et} \rightarrow_p n' \oplus \#:c_1$

| *Proc-CFG-CondTrue*:
 $\text{if } (b) \ c_1 \text{ else } c_2 \vdash \text{Label } 0$
 $\text{-IEdge } (\lambda cf. \text{state-check cf } b \text{ (Some true)})_{\checkmark} \rightarrow_p \text{Label } 1$

| *Proc-CFG-CondFalse*:
 $\text{if } (b) \ c_1 \text{ else } c_2 \vdash \text{Label } 0 \text{ -IEdge } (\lambda cf. \text{state-check cf } b \text{ (Some false)})_{\checkmark} \rightarrow_p$
 $\text{Label } (\#:c_1 + 1)$

| *Proc-CFG-CondThen*:
 $\llbracket c_1 \vdash n \text{ -et} \rightarrow_p n'; n \neq \text{Entry} \rrbracket \implies \text{if } (b) \ c_1 \text{ else } c_2 \vdash n \oplus 1 \text{ -et} \rightarrow_p n' \oplus 1$

| *Proc-CFG-CondElse*:
 $\llbracket c_2 \vdash n \text{ -et} \rightarrow_p n'; n \neq \text{Entry} \rrbracket$
 $\implies \text{if } (b) \ c_1 \text{ else } c_2 \vdash n \oplus (\#:c_1 + 1) \text{ -et} \rightarrow_p n' \oplus (\#:c_1 + 1)$

| *Proc-CFG-WhileTrue*:
 $\text{while } (b) \ c' \vdash \text{Label } 0 \text{ -IEdge } (\lambda cf. \text{state-check cf } b \text{ (Some true)})_{\checkmark} \rightarrow_p \text{Label } 2$

| *Proc-CFG-WhileFalse*:
 $\text{while } (b) \ c' \vdash \text{Label } 0 \text{ -IEdge } (\lambda cf. \text{state-check cf } b \text{ (Some false)})_{\checkmark} \rightarrow_p \text{Label } 1$

| *Proc-CFG-WhileFalseSkip*:
 $\text{while } (b) \ c' \vdash \text{Label } 1 \text{ -IEdge } \uparrow \text{id} \rightarrow_p \text{Exit}$

| *Proc-CFG-WhileBody*:
 $\llbracket c' \vdash n \text{ -et} \rightarrow_p n'; n \neq \text{Entry}; n' \neq \text{Exit} \rrbracket$
 $\implies \text{while } (b) \ c' \vdash n \oplus 2 \text{ -et} \rightarrow_p n' \oplus 2$

| *Proc-CFG-WhileBodyExit*:
 $\llbracket c' \vdash n \text{ -et} \rightarrow_p \text{Exit}; n \neq \text{Entry} \rrbracket \implies \text{while } (b) \ c' \vdash n \oplus 2 \text{ -et} \rightarrow_p \text{Label } 0$

| *Proc-CFG-Call*:
 $\text{Call } p \text{ es } \text{rets} \vdash \text{Label } 0 \text{ -CEdge } (p, \text{es}, \text{rets}) \rightarrow_p \text{Label } 1$

| *Proc-CFG-CallSkip*:
 $\text{Call } p \text{ es } \text{rets} \vdash \text{Label } 1 \text{ -IEdge } \uparrow \text{id} \rightarrow_p \text{Exit}$

Some lemmas about the procedure CFG

lemma *Proc-CFG-Exit-no-sourcenode* [dest]:
 $prog \vdash Exit -et \rightarrow_p n' \implies False$
 ⟨proof⟩

lemma *Proc-CFG-Entry-no-targetnode* [dest]:
 $prog \vdash n -et \rightarrow_p Entry \implies False$
 ⟨proof⟩

lemma *Proc-CFG-IEdge-intra-kind*:
 $prog \vdash n -IEdge et \rightarrow_p n' \implies \text{intra-kind } et$
 ⟨proof⟩

lemma [dest]: $prog \vdash n -IEdge (Q:r \rightarrow_p fs) \rightarrow_p n' \implies False$
 ⟨proof⟩

lemma [dest]: $prog \vdash n -IEdge (Q \leftarrow_p f) \rightarrow_p n' \implies False$
 ⟨proof⟩

lemma *Proc-CFG-sourcelabel-less-num-nodes*:
 $prog \vdash Label \ l -et \rightarrow_p n' \implies l < \# : prog$
 ⟨proof⟩

lemma *Proc-CFG-targetlabel-less-num-nodes*:
 $prog \vdash n -et \rightarrow_p Label \ l \implies l < \# : prog$
 ⟨proof⟩

lemma *Proc-CFG-EntryD*:
 $prog \vdash Entry -et \rightarrow_p n'$
 $\implies (n' = Exit \wedge et = IEdge(\lambda s. False)_{\surd}) \vee (n' = Label \ 0 \wedge et = IEdge(\lambda s. True)_{\surd})$
 ⟨proof⟩

lemma *Proc-CFG-Exit-edge*:
obtains $l \ et$ **where** $prog \vdash Label \ l -IEdge et \rightarrow_p Exit$ **and** $l \leq \# : prog$
 ⟨proof⟩

Lots of lemmas for call edges ...

lemma *Proc-CFG-Call-Labels*:
 $prog \vdash n -CEdge (p, es, rets) \rightarrow_p n' \implies \exists l. n = Label \ l \wedge n' = Label (Suc \ l)$
 ⟨proof⟩

lemma *Proc-CFG-Call-target-0:*

$\text{prog} \vdash n - \text{CEdge } (p, es, rets) \rightarrow_p \text{Label } 0 \implies n = \text{Entry}$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-Call-Intra-edge-not-same-source:*

$\llbracket \text{prog} \vdash n - \text{CEdge } (p, es, rets) \rightarrow_p n'; \text{prog} \vdash n - \text{IEdge } et \rightarrow_p n' \rrbracket \implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-Call-Intra-edge-not-same-target:*

$\llbracket \text{prog} \vdash n - \text{CEdge } (p, es, rets) \rightarrow_p n'; \text{prog} \vdash n'' - \text{IEdge } et \rightarrow_p n' \rrbracket \implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-Call-nodes-eq:*

$\llbracket \text{prog} \vdash n - \text{CEdge } (p, es, rets) \rightarrow_p n'; \text{prog} \vdash n - \text{CEdge } (p', es', rets') \rightarrow_p n' \rrbracket$
 $\implies n' = n'' \wedge p = p' \wedge es = es' \wedge rets = rets'$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-Call-nodes-eq':*

$\llbracket \text{prog} \vdash n - \text{CEdge } (p, es, rets) \rightarrow_p n'; \text{prog} \vdash n'' - \text{CEdge } (p', es', rets') \rightarrow_p n' \rrbracket$
 $\implies n = n'' \wedge p = p' \wedge es = es' \wedge rets = rets'$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-Call-targetnode-no-Call-sourcenode:*

$\llbracket \text{prog} \vdash n - \text{CEdge } (p, es, rets) \rightarrow_p n'; \text{prog} \vdash n' - \text{CEdge } (p', es', rets') \rightarrow_p n' \rrbracket$
 $\implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-Call-follows-id-edge:*

$\llbracket \text{prog} \vdash n - \text{CEdge } (p, es, rets) \rightarrow_p n'; \text{prog} \vdash n' - \text{IEdge } et \rightarrow_p n' \rrbracket \implies et = \uparrow id$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-edge-det:*

$\llbracket \text{prog} \vdash n - et \rightarrow_p n'; \text{prog} \vdash n - et' \rightarrow_p n' \rrbracket \implies et = et'$
 $\langle \text{proof} \rangle$

lemma *WCFG-deterministic:*

$\llbracket \text{prog} \vdash n_1 - et_1 \rightarrow_p n_1'; \text{prog} \vdash n_2 - et_2 \rightarrow_p n_2'; n_1 = n_2; n_1' \neq n_2' \rrbracket$
 $\implies \exists Q Q'. et_1 = \text{IEdge } (Q)_{\surd} \wedge et_2 = \text{IEdge } (Q')_{\surd} \wedge$
 $(\forall s. (Q s \longrightarrow \neg Q' s) \wedge (Q' s \longrightarrow \neg Q s))$
 $\langle \text{proof} \rangle$

2.3.2 And now: the interprocedural CFG

Statements containing calls

A procedure is a tuple composed of its name, its input and output variables and its method body

type-synonym $proc = (pname \times vname\ list \times vname\ list \times cmd)$

type-synonym $procs = proc\ list$

$containsCall$ guarantees that a call to procedure p is in a certain statement.

declare $conj-cong[fundef-cong]$

function $containsCall ::$

$procs \Rightarrow cmd \Rightarrow pname\ list \Rightarrow pname \Rightarrow bool$

where $containsCall\ procs\ Skip\ ps\ p \longleftrightarrow False$

| $containsCall\ procs\ (V:=e)\ ps\ p \longleftrightarrow False$

| $containsCall\ procs\ (c_1;;c_2)\ ps\ p \longleftrightarrow$

$containsCall\ procs\ c_1\ ps\ p \vee containsCall\ procs\ c_2\ ps\ p$

| $containsCall\ procs\ (if\ (b)\ c_1\ else\ c_2)\ ps\ p \longleftrightarrow$

$containsCall\ procs\ c_1\ ps\ p \vee containsCall\ procs\ c_2\ ps\ p$

| $containsCall\ procs\ (while\ (b)\ c)\ ps\ p \longleftrightarrow$

$containsCall\ procs\ c\ ps\ p$

| $containsCall\ procs\ (Call\ q\ es'\ rets')\ ps\ p \longleftrightarrow p = q \wedge ps = [] \vee$

$(\exists\ ins\ outs\ c\ ps'.\ ps = q\#\ ps' \wedge (q,ins,outs,c) \in set\ procs \wedge$

$containsCall\ procs\ c\ ps'\ p)$

$\langle proof \rangle$

termination $containsCall$

$\langle proof \rangle$

lemmas $containsCall-induct[case-names\ Skip\ LAss\ Seq\ Cond\ While\ Call] = containsCall.induct$

lemma $containsCallcases:$

$containsCall\ procs\ prog\ ps\ p$

$\implies ps = [] \wedge containsCall\ procs\ prog\ ps\ p \vee$

$(\exists\ q\ ins\ outs\ c\ ps'.\ ps = ps'\@[q] \wedge (q,ins,outs,c) \in set\ procs \wedge$

$containsCall\ procs\ c\ []\ p \wedge containsCall\ procs\ prog\ ps'\ q)$

$\langle proof \rangle$

lemma $containsCallE:$

$\llbracket containsCall\ procs\ prog\ ps\ p;$

$\llbracket ps = []; containsCall\ procs\ prog\ ps\ p \rrbracket \implies P\ procs\ prog\ ps\ p;$

$\wedge q\ ins\ outs\ c\ es'\ rets'\ ps'.\ \llbracket ps = ps'\@[q]; (q,ins,outs,c) \in set\ procs;$

$containsCall\ procs\ c\ []\ p; containsCall\ procs\ prog\ ps'\ q \rrbracket$

$\implies P\ procs\ prog\ ps\ p \rrbracket \implies P\ procs\ prog\ ps\ p$

$\langle \text{proof} \rangle$

lemma *containsCall-in-proc*:

$\llbracket \text{containsCall procs prog qs } q; (q, \text{ins}, \text{outs}, c) \in \text{set procs};$
 $\text{containsCall procs } c \llbracket p \rrbracket$
 $\implies \text{containsCall procs prog } (qs@[q]) p$
 $\langle \text{proof} \rangle$

lemma *containsCall-indirection*:

$\llbracket \text{containsCall procs prog qs } q; \text{containsCall procs } c \text{ ps } p;$
 $(q, \text{ins}, \text{outs}, c) \in \text{set procs} \rrbracket$
 $\implies \text{containsCall procs prog } (qs@q\#ps) p$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-Call-containsCall*:

$\text{prog} \vdash n - \text{CEdge } (p, \text{es}, \text{rets}) \rightarrow_p n' \implies \text{containsCall procs prog} \llbracket p$
 $\langle \text{proof} \rangle$

lemma *containsCall-empty-Proc-CFG-Call-edge*:

assumes $\text{containsCall procs prog} \llbracket p$
obtains $l \text{ es } \text{rets } l'$ **where** $\text{prog} \vdash \text{Label } l - \text{CEdge } (p, \text{es}, \text{rets}) \rightarrow_p \text{Label } l'$
 $\langle \text{proof} \rangle$

The edges of the combined CFG

type-synonym $\text{node} = (\text{pname} \times \text{label})$

type-synonym $\text{edge} = (\text{node} \times (\text{vname}, \text{val}, \text{node}, \text{pname}) \text{ edge-kind} \times \text{node})$

fun $\text{get-proc} :: \text{node} \Rightarrow \text{pname}$
where $\text{get-proc } (p, l) = p$

inductive *PCFG* ::

$\text{cmd} \Rightarrow \text{procs} \Rightarrow \text{node} \Rightarrow (\text{vname}, \text{val}, \text{node}, \text{pname}) \text{ edge-kind} \Rightarrow \text{node} \Rightarrow \text{bool}$
 $(-, - \vdash - \dashrightarrow - [51, 51, 0, 0, 0] 81)$

for $\text{prog} :: \text{cmd}$ **and** $\text{procs} :: \text{procs}$

where

Main:

$\text{prog} \vdash n - \text{IEdge } et \rightarrow_p n' \implies \text{prog}, \text{procs} \vdash (\text{Main}, n) - et \rightarrow (\text{Main}, n')$

| *Proc*:

$\llbracket (p, \text{ins}, \text{outs}, c) \in \text{set procs}; c \vdash n - \text{IEdge } et \rightarrow_p n';$
 $\text{containsCall procs prog ps } p \rrbracket$
 $\implies \text{prog}, \text{procs} \vdash (p, n) - et \rightarrow (p, n')$

| *MainCall*:

$$\begin{aligned} & \llbracket \text{prog} \vdash \text{Label } l - \text{CEdge } (p, \text{es}, \text{rets}) \rightarrow_p n'; (p, \text{ins}, \text{outs}, c) \in \text{set procs} \rrbracket \\ & \implies \text{prog}, \text{procs} \vdash (\text{Main}, \text{Label } l) \\ & \quad -(\lambda s. \text{True}): (\text{Main}, n') \hookrightarrow_p \text{map } (\lambda e \text{ cf}. \text{interpret } e \text{ cf}) \text{ es} \rightarrow (p, \text{Entry}) \end{aligned}$$

| *ProcCall*:

$$\begin{aligned} & \llbracket (p, \text{ins}, \text{outs}, c) \in \text{set procs}; c \vdash \text{Label } l - \text{CEdge } (p', \text{es}', \text{rets}') \rightarrow_p \text{Label } l'; \\ & \quad (p', \text{ins}', \text{outs}', c') \in \text{set procs}; \text{containsCall procs prog ps } p \rrbracket \\ & \implies \text{prog}, \text{procs} \vdash (p, \text{Label } l) \\ & \quad -(\lambda s. \text{True}): (p, \text{Label } l') \hookrightarrow_p \text{map } (\lambda e \text{ cf}. \text{interpret } e \text{ cf}) \text{ es}' \rightarrow (p', \text{Entry}) \end{aligned}$$

| *MainReturn*:

$$\begin{aligned} & \llbracket \text{prog} \vdash \text{Label } l - \text{CEdge } (p, \text{es}, \text{rets}) \rightarrow_p \text{Label } l'; (p, \text{ins}, \text{outs}, c) \in \text{set procs} \rrbracket \\ & \implies \text{prog}, \text{procs} \vdash (p, \text{Exit}) -(\lambda \text{cf}. \text{snd } \text{cf} = (\text{Main}, \text{Label } l')) \hookrightarrow_p \\ & \quad (\lambda \text{cf } \text{cf}'. \text{cf}'(\text{rets} \text{[:]=} \text{map } \text{cf } \text{outs})) \rightarrow (\text{Main}, \text{Label } l') \end{aligned}$$

| *ProcReturn*:

$$\begin{aligned} & \llbracket (p, \text{ins}, \text{outs}, c) \in \text{set procs}; c \vdash \text{Label } l - \text{CEdge } (p', \text{es}', \text{rets}') \rightarrow_p \text{Label } l'; \\ & \quad (p', \text{ins}', \text{outs}', c') \in \text{set procs}; \text{containsCall procs prog ps } p \rrbracket \\ & \implies \text{prog}, \text{procs} \vdash (p', \text{Exit}) -(\lambda \text{cf}. \text{snd } \text{cf} = (p, \text{Label } l')) \hookrightarrow_{p'} \\ & \quad (\lambda \text{cf } \text{cf}'. \text{cf}'(\text{rets}' \text{[:]=} \text{map } \text{cf } \text{outs}')) \rightarrow (p, \text{Label } l') \end{aligned}$$

| *MainCallReturn*:

$$\begin{aligned} & \text{prog} \vdash n - \text{CEdge } (p, \text{es}, \text{rets}) \rightarrow_p n' \\ & \implies \text{prog}, \text{procs} \vdash (\text{Main}, n) -(\lambda s. \text{False})_{\surd} \rightarrow (\text{Main}, n') \end{aligned}$$

| *ProcCallReturn*:

$$\begin{aligned} & \llbracket (p, \text{ins}, \text{outs}, c) \in \text{set procs}; c \vdash n - \text{CEdge } (p', \text{es}', \text{rets}') \rightarrow_p n'; \\ & \quad \text{containsCall procs prog ps } p \rrbracket \\ & \implies \text{prog}, \text{procs} \vdash (p, n) -(\lambda s. \text{False})_{\surd} \rightarrow (p, n') \end{aligned}$$

end

2.4 Well-formedness of programs

theory *WellFormProgs* imports *PCFG* begin

2.4.1 Well-formedness of procedure lists.

definition *wf-proc* :: *proc* \Rightarrow *bool*
where *wf-proc* *x* \equiv *let* (*p*, *ins*, *outs*, *c*) = *x in*
p \neq *Main* \wedge *distinct ins* \wedge *distinct outs*

definition *well-formed* :: *procs* \Rightarrow *bool*
where *well-formed procs* \equiv *distinct-fst procs* \wedge
 $(\forall (p, \text{ins}, \text{outs}, c) \in \text{set procs}. \text{wf-proc } (p, \text{ins}, \text{outs}, c))$

lemma $[dest]: \llbracket \text{well-formed procs}; (Main, ins, outs, c) \in \text{set procs} \rrbracket \implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *well-formed-same-procs* $[dest]:$
 $\llbracket \text{well-formed procs}; (p, ins, outs, c) \in \text{set procs}; (p, ins', outs', c') \in \text{set procs} \rrbracket$
 $\implies ins = ins' \wedge outs = outs' \wedge c = c'$
 $\langle \text{proof} \rangle$

lemma *PCFG-sourcelabel-None-less-num-nodes*:
 $\llbracket \text{prog, procs} \vdash (Main, Label\ l) -et \rightarrow n'; \text{well-formed procs} \rrbracket \implies l < \#: \text{prog}$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-sourcelabel-Some-less-num-nodes*:
 $\llbracket \text{prog, procs} \vdash (p, Label\ l) -et \rightarrow n'; (p, ins, outs, c) \in \text{set procs};$
 $\text{well-formed procs} \rrbracket \implies l < \#: c$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-targetlabel-Main-less-num-nodes*:
 $\llbracket \text{prog, procs} \vdash n -et \rightarrow (Main, Label\ l); \text{well-formed procs} \rrbracket \implies l < \#: \text{prog}$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-targetlabel-Some-less-num-nodes*:
 $\llbracket \text{prog, procs} \vdash n -et \rightarrow (p, Label\ l); (p, ins, outs, c) \in \text{set procs};$
 $\text{well-formed procs} \rrbracket \implies l < \#: c$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-edge-det*:
 $\llbracket \text{prog, procs} \vdash n -et \rightarrow n'; \text{prog, procs} \vdash n -et' \rightarrow n'; \text{well-formed procs} \rrbracket$
 $\implies et = et'$
 $\langle \text{proof} \rangle$

lemma *Proc-CFG-deterministic*:
 $\llbracket \text{prog, procs} \vdash n_1 -et_1 \rightarrow n_1'; \text{prog, procs} \vdash n_2 -et_2 \rightarrow n_2'; n_1 = n_2; n_1' \neq n_2';$
 $\text{intra-kind } et_1; \text{intra-kind } et_2; \text{well-formed procs} \rrbracket$
 $\implies \exists Q\ Q'. et_1 = (Q)_{\surd} \wedge et_2 = (Q')_{\surd} \wedge$
 $(\forall s. (Q\ s \longrightarrow \neg Q'\ s) \wedge (Q'\ s \longrightarrow \neg Q\ s))$
 $\langle \text{proof} \rangle$

2.4.2 Well-formedness of programs in combination with a procedure list.

definition $wf :: \text{cmd} \Rightarrow \text{procs} \Rightarrow \text{bool}$
where $wf\ \text{prog}\ \text{procs} \equiv \text{well-formed procs} \wedge$

$(\forall ps p. \text{containsCall procs prog ps } p \longrightarrow (\exists ins outs c. (p,ins,outs,c) \in \text{set procs}$
 \wedge
 $(\forall c' n n' es \text{rets}. c' \vdash n - \text{CEdge } (p,es,rets) \rightarrow_p n' \longrightarrow$
 $\text{distinct rets} \wedge \text{length rets} = \text{length outs} \wedge \text{length es} = \text{length ins}))$)

lemma *wf-well-formed* [intro]: *wf prog procs* \implies *well-formed procs*
 $\langle \text{proof} \rangle$

lemma *wf-distinct-rets* [intro]:
 $\llbracket \text{wf prog procs}; \text{containsCall procs prog ps } p; (p,ins,outs,c) \in \text{set procs};$
 $c' \vdash n - \text{CEdge } (p,es,rets) \rightarrow_p n' \rrbracket \implies \text{distinct rets}$
 $\langle \text{proof} \rangle$

lemma
assumes *wf prog procs* **and** *containsCall procs prog ps p*
and $(p,ins,outs,c) \in \text{set procs}$ **and** $c' \vdash n - \text{CEdge } (p,es,rets) \rightarrow_p n'$
shows *wf-length-retsI* [intro]: *length rets = length outs*
and *wf-length-esI* [intro]: *length es = length ins*
 $\langle \text{proof} \rangle$

2.4.3 Type of well-formed programs

definition *wf-prog* = $\{(prog,procs). \text{wf prog procs}\}$

typedef *wf-prog* = *wf-prog*
 $\langle \text{proof} \rangle$

lemma *wf-wf-prog:Rep-wf-prog wfp* = $(prog,procs) \implies \text{wf prog procs}$
 $\langle \text{proof} \rangle$

lemma *wfp-Seq1*: **assumes** *Rep-wf-prog wfp* = $(c_1;; c_2, procs)$
obtains *wfp'* **where** *Rep-wf-prog wfp'* = $(c_1, procs)$
 $\langle \text{proof} \rangle$

lemma *wfp-Seq2*: **assumes** *Rep-wf-prog wfp* = $(c_1;; c_2, procs)$
obtains *wfp'* **where** *Rep-wf-prog wfp'* = $(c_2, procs)$
 $\langle \text{proof} \rangle$

lemma *wfp-CondTrue*: **assumes** *Rep-wf-prog wfp* = $(\text{if } (b) c_1 \text{ else } c_2, procs)$
obtains *wfp'* **where** *Rep-wf-prog wfp'* = $(c_1, procs)$
 $\langle \text{proof} \rangle$

lemma *wfp-CondFalse*: **assumes** *Rep-wf-prog wfp* = $(\text{if } (b) c_1 \text{ else } c_2, procs)$
obtains *wfp'* **where** *Rep-wf-prog wfp'* = $(c_2, procs)$
 $\langle \text{proof} \rangle$

lemma *wfp-WhileBody*: **assumes** *Rep-wf-prog wfp = (while (b) c', procs)*
obtains *wfp'* **where** *Rep-wf-prog wfp' = (c', procs)*
 \langle *proof* \rangle

lemma *wfp-Call*: **assumes** *Rep-wf-prog wfp = (prog,procs)*
and $(p,ins,outs,c) \in \text{set } procs$ **and** *containsCall procs prog ps p*
obtains *wfp'* **where** *Rep-wf-prog wfp' = (c,procs)*
 \langle *proof* \rangle

end

2.5 Instantiate CFG locales with Proc CFG

theory *Interpretation* **imports** *WellFormProgs ../StaticInter/CFGExit* **begin**

2.5.1 Lifting of the basic definitions

abbreviation *sourcenode* :: *edge* \Rightarrow *node*
where *sourcenode e* \equiv *fst e*

abbreviation *targetnode* :: *edge* \Rightarrow *node*
where *targetnode e* \equiv *snd(snd e)*

abbreviation *kind* :: *edge* \Rightarrow $(vname, val, node, pname)$ *edge-kind*
where *kind e* \equiv *fst(snd e)*

definition *valid-edge* :: *wf-prog* \Rightarrow *edge* \Rightarrow *bool*
where *valid-edge wfp a* \equiv *let (prog,procs) = Rep-wf-prog wfp in*
prog,procs \vdash *sourcenode a* $-$ *kind a* \rightarrow *targetnode a*

definition *get-return-edges* :: *wf-prog* \Rightarrow *edge* \Rightarrow *edge set*
where *get-return-edges wfp a* \equiv
case kind a of $Q:r \hookrightarrow_p fs \Rightarrow \{a'. \text{valid-edge } wfp \ a' \wedge (\exists Q' f'. \text{kind } a' = Q' \hookrightarrow_p f') \wedge$
 $\text{targetnode } a' = r\}$
 $| - \Rightarrow \{\}$

lemma *get-return-edges-non-call-empty*:
 $\forall Q \ r \ p \ fs. \text{kind } a \neq Q:r \hookrightarrow_p fs \Longrightarrow \text{get-return-edges } wfp \ a = \{\}$
 \langle *proof* \rangle

lemma *call-has-return-edge*:

assumes *valid-edge wfp a* **and** *kind a = Q:r↔pfs*
obtains *a'* **where** *valid-edge wfp a'* **and** $\exists Q' f'. \textit{kind } a' = Q' \leftrightarrow pf'$
and *targetnode a' = r*
 ⟨*proof*⟩

lemma *get-return-edges-call-nonempty*:
 $\llbracket \textit{valid-edge wfp } a; \textit{kind } a = Q:r \leftrightarrow pfs \rrbracket \implies \textit{get-return-edges wfp } a \neq \{\}$
 ⟨*proof*⟩

lemma *only-return-edges-in-get-return-edges*:
 $\llbracket \textit{valid-edge wfp } a; \textit{kind } a = Q:r \leftrightarrow pfs; a' \in \textit{get-return-edges wfp } a \rrbracket$
 $\implies \exists Q' f'. \textit{kind } a' = Q' \leftrightarrow pf'$
 ⟨*proof*⟩

abbreviation *lift-procs* :: *wf-prog* \Rightarrow (*pname* \times *vname list* \times *vname list*) *list*
where *lift-procs wfp* \equiv *let (prog,procs) = Rep-wf-prog wfp in*
map ($\lambda x. (\textit{fst } x, \textit{fst}(\textit{snd } x), \textit{fst}(\textit{snd}(\textit{snd } x)))$) procs

2.5.2 Instatiation of the CFG locale

interpretation *ProcCFG*:
CFG sourcenode targetnode kind valid-edge wfp (Main,Entry)
get-proc get-return-edges wfp lift-procs wfp Main
for *wfp*
 ⟨*proof*⟩

2.5.3 Instatiation of the CFGExit locale

interpretation *ProcCFGExit*:
CFGExit sourcenode targetnode kind valid-edge wfp (Main,Entry)
get-proc get-return-edges wfp lift-procs wfp Main (Main,Exit)
for *wfp*
 ⟨*proof*⟩

end

2.6 Labels

theory *Labels imports Com begin*

Labels describe a mapping from the inner node label to the matching command

inductive *labels* :: *cmd* \Rightarrow *nat* \Rightarrow *cmd* \Rightarrow *bool*
where

Labels-Base:
 $labels\ c\ 0\ c$

| *Labels-LAss:*
 $labels\ (V:=e)\ 1\ Skip$

| *Labels-Seq1:*
 $labels\ c_1\ l\ c \implies labels\ (c_1;;c_2)\ l\ (c;;c_2)$

| *Labels-Seq2:*
 $labels\ c_2\ l\ c \implies labels\ (c_1;;c_2)\ (l + \#:c_1)\ c$

| *Labels-CondTrue:*
 $labels\ c_1\ l\ c \implies labels\ (if\ (b)\ c_1\ else\ c_2)\ (l + 1)\ c$

| *Labels-CondFalse:*
 $labels\ c_2\ l\ c \implies labels\ (if\ (b)\ c_1\ else\ c_2)\ (l + \#:c_1 + 1)\ c$

| *Labels-WhileBody:*
 $labels\ c'\ l\ c \implies labels\ (while(b)\ c')\ (l + 2)\ (c;;while(b)\ c')$

| *Labels-WhileExit:*
 $labels\ (while(b)\ c')\ 1\ Skip$

| *Labels-Call:*
 $labels\ (Call\ p\ es\ rets)\ 1\ Skip$

lemma *label-less-num-inner-nodes:*
 $labels\ c\ l\ c' \implies l < \#:c$
 $\langle proof \rangle$

declare *One-nat-def* [*simp del*]

lemma *less-num-inner-nodes-label:*
assumes $l < \#:c$ **obtains** c' **where** $labels\ c\ l\ c'$
 $\langle proof \rangle$

lemma *labels-det:*
 $labels\ c\ l\ c' \implies (\bigwedge c''. labels\ c\ l\ c'' \implies c' = c'')$
 $\langle proof \rangle$

definition *label* :: $cmd \Rightarrow nat \Rightarrow cmd$
where $label\ c\ n \equiv (THE\ c'. labels\ c\ n\ c')$

lemma *labels-THE*:
 $labels\ c\ l\ c' \implies (THE\ c'.\ labels\ c\ l\ c') = c'$
 <proof>

lemma *labels-label*: $labels\ c\ l\ c' \implies label\ c\ l = c'$
 <proof>

end

2.7 Instantiate well-formedness locales with Proc CFG

theory *WellFormed* imports *Interpretation Labels ../StaticInter/CFGExit-wf* begin

2.7.1 Determining the first atomic command

fun *fst-cmd* :: $cmd \Rightarrow cmd$
where *fst-cmd* ($c_1;;c_2$) = *fst-cmd* c_1
 | *fst-cmd* $c = c$

lemma *Proc-CFG-Call-target-fst-cmd-Skip*:
 $\llbracket labels\ prog\ l'\ c; prog \vdash n - CEdge\ (p, es, rets) \rightarrow_p\ Label\ l \rrbracket$
 $\implies fst-cmd\ c = Skip$
 <proof>

lemma *Proc-CFG-Call-source-fst-cmd-Call*:
 $\llbracket labels\ prog\ l\ c; prog \vdash Label\ l - CEdge\ (p, es, rets) \rightarrow_p\ n \rrbracket$
 $\implies \exists p\ es\ rets.\ fst-cmd\ c = Call\ p\ es\ rets$
 <proof>

2.7.2 Definition of Def and Use sets

ParamDefs

lemma *PCFG-CallEdge-THE-rets*:
 $prog \vdash n - CEdge\ (p, es, rets) \rightarrow_p\ n'$
 $\implies (THE\ rets'.\ \exists p'\ es'\ n'.\ prog \vdash n - CEdge(p', es', rets') \rightarrow_p\ n') = rets$
 <proof>

definition *ParamDefs-proc* :: $cmd \Rightarrow label \Rightarrow vname\ list$
where *ParamDefs-proc* $c\ n \equiv$
 if $(\exists n'\ p'\ es'\ rets'.\ c \vdash n' - CEdge(p', es', rets') \rightarrow_p\ n)$ then
 ($THE\ rets'.\ \exists p'\ es'\ n'.\ c \vdash n' - CEdge(p', es', rets') \rightarrow_p\ n$)

else []

lemma *in-procs-THE-in-procs-cmd*:

[[well-formed procs; (p,ins,outs,c) ∈ set procs]]
⇒ (THE c'. ∃ ins' outs'. (p,ins',outs',c') ∈ set procs) = c
<proof>

definition *ParamDefs* :: wf-prog ⇒ node ⇒ vname list

where *ParamDefs wfp n* ≡ let (prog,procs) = Rep-wf-prog wfp; (p,l) = n in
(if (p = Main) then *ParamDefs-proc prog l*
else (if (∃ ins outs c. (p,ins,outs,c) ∈ set procs)
then *ParamDefs-proc (THE c'. ∃ ins' outs'. (p,ins',outs',c') ∈ set procs) l*
else []))

lemma *ParamDefs-Main-Return-target*:

[[Rep-wf-prog wfp = (prog,procs); prog ⊢ n -CEdge(p',es,rets)→_p n']]
⇒ *ParamDefs wfp (Main,n')* = rets
<proof>

lemma *ParamDefs-Proc-Return-target*:

assumes *Rep-wf-prog wfp = (prog,procs)*
and (p,ins,outs,c) ∈ set procs **and** c ⊢ n -CEdge(p',es,rets)→_p n'
shows *ParamDefs wfp (p,n')* = rets
<proof>

lemma *ParamDefs-Main-IEdge-Nil*:

[[Rep-wf-prog wfp = (prog,procs); prog ⊢ n -IEdge et→_p n']]
⇒ *ParamDefs wfp (Main,n')* = []
<proof>

lemma *ParamDefs-Proc-IEdge-Nil*:

assumes *Rep-wf-prog wfp = (prog,procs)*
and (p,ins,outs,c) ∈ set procs **and** c ⊢ n -IEdge et→_p n'
shows *ParamDefs wfp (p,n')* = []
<proof>

lemma *ParamDefs-Main-CEdge-Nil*:

[[Rep-wf-prog wfp = (prog,procs); prog ⊢ n' -CEdge(p',es,rets)→_p n'']]
⇒ *ParamDefs wfp (Main,n')* = []
<proof>

lemma *ParamDefs-Proc-CEdge-Nil*:

assumes *Rep-wf-prog wfp = (prog,procs)*
and (p,ins,outs,c) ∈ set procs **and** c ⊢ n' -CEdge(p',es,rets)→_p n''
shows *ParamDefs wfp (p,n')* = []
<proof>

lemma *assumes* *valid-edge wfp a* **and** *kind a = Q'↔pf'*
and $(p, ins, outs) \in set (lift-procs wfp)$
shows $ParamDefs-length:length (ParamDefs wfp (targetnode a)) = length outs$
(is ?length)
and $Return-update:f' cf cf' = cf'(ParamDefs wfp (targetnode a) [:=] map cf outs)$
(is ?update)
 $\langle proof \rangle$

ParamUses

fun *fv* :: *expr* \Rightarrow *vname set*

where

$fv (Val v) = \{\}$
 $| fv (Var V) = \{V\}$
 $| fv (e1 \ll bop \gg e2) = (fv e1 \cup fv e2)$

lemma *rhs-interpret-eq*:

$\llbracket state-check cf e v'; \forall V \in fv e. cf V = cf' V \rrbracket$
 $\implies state-check cf' e v'$

$\langle proof \rangle$

lemma *PCFG-CallEdge-THE-es*:

$prog \vdash n - CEdge(p, es, rets) \rightarrow_p n'$
 $\implies (THE es'. \exists p' rets' n'. prog \vdash n - CEdge(p', es', rets') \rightarrow_p n') = es$
 $\langle proof \rangle$

definition *ParamUses-proc* :: *cmd* \Rightarrow *label* \Rightarrow *vname set list*

where *ParamUses-proc* *c* *n* \equiv

if $(\exists n' p' es' rets'. c \vdash n - CEdge(p', es', rets') \rightarrow_p n')$ *then*
 $(map fv (THE es'. \exists p' rets' n'. c \vdash n - CEdge(p', es', rets') \rightarrow_p n'))$
else \square

definition *ParamUses* :: *wf-prog* \Rightarrow *node* \Rightarrow *vname set list*

where *ParamUses* *wfp* *n* $\equiv let (prog, procs) = Rep-wf-prog wfp; (p, l) = n in$

(if $(p = Main)$ *then* *ParamUses-proc* *prog* *l*

else *(if* $(\exists ins outs c. (p, ins, outs, c) \in set procs)$

then *ParamUses-proc* $(THE c'. \exists ins' outs'. (p, ins', outs', c') \in set procs)$ *l*

else \square *)*

lemma *ParamUses-Main-Return-target*:

$\llbracket Rep-wf-prog wfp = (prog, procs); prog \vdash n - CEdge(p', es, rets) \rightarrow_p n' \rrbracket$

$\implies \text{ParamUses wfp (Main,n) = map fv es}$
 ⟨proof⟩

lemma *ParamUses-Proc-Return-target:*
assumes $\text{Rep-wf-prog wfp} = (\text{prog}, \text{procs})$
and $(p, \text{ins}, \text{outs}, c) \in \text{set procs}$ **and** $c \vdash n - \text{CEdge}(p', \text{es}, \text{rets}) \rightarrow_p n'$
shows $\text{ParamUses wfp} (p, n) = \text{map fv es}$
 ⟨proof⟩

lemma *ParamUses-Main-IEdge-Nil:*
 $\llbracket \text{Rep-wf-prog wfp} = (\text{prog}, \text{procs}); \text{prog} \vdash n - \text{IEdge et} \rightarrow_p n' \rrbracket$
 $\implies \text{ParamUses wfp} (\text{Main}, n) = []$
 ⟨proof⟩

lemma *ParamUses-Proc-IEdge-Nil:*
assumes $\text{Rep-wf-prog wfp} = (\text{prog}, \text{procs})$
and $(p, \text{ins}, \text{outs}, c) \in \text{set procs}$ **and** $c \vdash n - \text{IEdge et} \rightarrow_p n'$
shows $\text{ParamUses wfp} (p, n) = []$
 ⟨proof⟩

lemma *ParamUses-Main-CEdge-Nil:*
 $\llbracket \text{Rep-wf-prog wfp} = (\text{prog}, \text{procs}); \text{prog} \vdash n' - \text{CEdge}(p', \text{es}, \text{rets}) \rightarrow_p n \rrbracket$
 $\implies \text{ParamUses wfp} (\text{Main}, n) = []$
 ⟨proof⟩

lemma *ParamUses-Proc-CEdge-Nil:*
assumes $\text{Rep-wf-prog wfp} = (\text{prog}, \text{procs})$
and $(p, \text{ins}, \text{outs}, c) \in \text{set procs}$ **and** $c \vdash n' - \text{CEdge}(p', \text{es}, \text{rets}) \rightarrow_p n$
shows $\text{ParamUses wfp} (p, n) = []$
 ⟨proof⟩

Def

fun $\text{lhs} :: \text{cmd} \Rightarrow \text{vname set}$

where

$\text{lhs Skip} = \{\}$
 $| \text{lhs } (V := e) = \{V\}$
 $| \text{lhs } (c_1 ;; c_2) = \text{lhs } c_1$
 $| \text{lhs } (\text{if } (b) c_1 \text{ else } c_2) = \{\}$
 $| \text{lhs } (\text{while } (b) c) = \{\}$
 $| \text{lhs } (\text{Call } p \text{ es } \text{rets}) = \{\}$

lemma $\text{lhs-fst-cmd:lhs (fst-cmd } c) = \text{lhs } c$ ⟨proof⟩

lemma *Proc-CFG-Call-source-empty-lhs:*
assumes $\text{prog} \vdash \text{Label } l - \text{CEdge} (p, \text{es}, \text{rets}) \rightarrow_p n'$
shows $\text{lhs } (\text{label prog } l) = \{\}$
 ⟨proof⟩

lemma *in-procs-THE-in-procs-ins*:

$\llbracket \text{well-formed procs}; (p, \text{ins}, \text{outs}, c) \in \text{set procs} \rrbracket$
 $\implies (\text{THE } \text{ins}'. \exists c' \text{ outs}'. (p, \text{ins}', \text{outs}', c') \in \text{set procs}) = \text{ins}$
 $\langle \text{proof} \rangle$

definition *Def* :: *wf-prog* \Rightarrow *node* \Rightarrow *vname set*

where *Def wfp n* \equiv (let (*prog,procs*) = *Rep-wf-prog wfp*; (*p,l*) = *n* in
(case *l* of *Label lx* \Rightarrow
(if *p* = *Main* then *lhs* (label *prog lx*)
else (if ($\exists \text{ins outs } c. (p, \text{ins}, \text{outs}, c) \in \text{set procs}$)
then
lhs (label (THE *c'*. $\exists \text{ins}' \text{ outs}' . (p, \text{ins}', \text{outs}', c') \in \text{set procs}$) *lx*)
else { })))
| *Entry* \Rightarrow if ($\exists \text{ins outs } c. (p, \text{ins}, \text{outs}, c) \in \text{set procs}$)
then (set
(THE *ins'*. $\exists c' \text{ outs}' . (p, \text{ins}', \text{outs}', c') \in \text{set procs}$)) else { }
| *Exit* \Rightarrow { })))
 \cup set (*ParamDefs wfp n*)

lemma *Entry-Def-empty:Def wfp (Main, Entry) = { }*
 $\langle \text{proof} \rangle$

lemma *Exit-Def-empty:Def wfp (Main, Exit) = { }*
 $\langle \text{proof} \rangle$

Use

fun *rhs* :: *cmd* \Rightarrow *vname set*

where

rhs Skip = { }
| *rhs (V:=e)* = *fv e*
| *rhs (c₁;;c₂)* = *rhs c₁*
| *rhs (if (b) c₁ else c₂)* = *fv b*
| *rhs (while (b) c)* = *fv b*
| *rhs (Call p es rets)* = { }

lemma *rhs-fst-cmd:rhs (fst-cmd c) = rhs c* $\langle \text{proof} \rangle$

lemma *Proc-CFG-Call-target-empty-rhs*:

assumes *prog* \vdash *n* - *CEdge (p,es,rets)* \rightarrow_p *Label l'*

shows *rhs* (label *prog l'*) = { }

$\langle \text{proof} \rangle$

lemma *in-procs-THE-in-procs-outs*:

$\llbracket \text{well-formed procs}; (p, \text{ins}, \text{outs}, c) \in \text{set procs} \rrbracket$
 $\implies (\text{THE } \text{outs}' . \exists c' \text{ ins}' . (p, \text{ins}', \text{outs}', c') \in \text{set procs}) = \text{outs}$
 $\langle \text{proof} \rangle$

definition $\text{Use} :: \text{wf-prog} \Rightarrow \text{node} \Rightarrow \text{vname set}$
where $\text{Use wfp } n \equiv (\text{let } (\text{prog}, \text{procs}) = \text{Rep-wf-prog wfp}; (p, l) = n \text{ in}$
 $(\text{case } l \text{ of Label } lx \Rightarrow$
 $(\text{if } p = \text{Main} \text{ then rhs (label prog } lx)$
 $\text{else (if } (\exists \text{ins outs } c . (p, \text{ins}, \text{outs}, c) \in \text{set procs})$
 then
 $\text{rhs (label (THE } c' . \exists \text{ins}' \text{ outs}' . (p, \text{ins}', \text{outs}', c') \in \text{set procs}) } lx)$
 $\text{else } \{\})$)
 $| \text{Exit} \Rightarrow \text{if } (\exists \text{ins outs } c . (p, \text{ins}, \text{outs}, c) \in \text{set procs})$
 $\text{then (set (THE } \text{outs}' . \exists c' \text{ ins}' . (p, \text{ins}', \text{outs}', c') \in \text{set procs}))$
 $\text{else } \{\}$
 $| \text{Entry} \Rightarrow \text{if } (\exists \text{ins outs } c . (p, \text{ins}, \text{outs}, c) \in \text{set procs})$
 $\text{then (set (THE } \text{ins}' . \exists c' \text{ outs}' . (p, \text{ins}', \text{outs}', c') \in \text{set procs}))$
 $\text{else } \{\})$)
 $\cup \text{Union (set (ParamUses wfp } n)) \cup \text{set (ParamDefs wfp } n)$

lemma $\text{Entry-Use-empty:Use wfp (Main, Entry)} = \{\}$
 $\langle \text{proof} \rangle$

lemma $\text{Exit-Use-empty:Use wfp (Main, Exit)} = \{\}$
 $\langle \text{proof} \rangle$

2.7.3 Lemmas about edges and call frames

lemmas $\text{transfers-simps} = \text{ProcCFG.transfer.simps}[\text{simplified}]$
declare $\text{transfers-simps} [\text{simp}]$

abbreviation $\text{state-val} :: ('var \rightarrow 'val) \times 'ret \text{ list} \Rightarrow 'var \rightarrow 'val$
where $\text{state-val } s \ V \equiv (\text{fst (hd } s)) \ V$

lemma $\text{Proc-CFG-edge-no-lhs-equal}$:
assumes $\text{prog} \vdash \text{Label } l \text{ -IEdge } et \rightarrow_p n'$ **and** $V \notin \text{lhs (label prog } l)$
shows $\text{state-val (CFG.transfer (lift-procs wfp) et (cf\#cfs)) } V = \text{fst } cf \ V$
 $\langle \text{proof} \rangle$

lemma $\text{Proc-CFG-edge-uses-only-rhs}$:
assumes $\text{prog} \vdash \text{Label } l \text{ -IEdge } et \rightarrow_p n'$ **and** $\text{CFG.pred } et \ s$
and $\text{CFG.pred } et \ s'$ **and** $\forall V \in \text{rhs (label prog } l). \text{state-val } s \ V = \text{state-val } s' \ V$
shows $\forall V \in \text{lhs (label prog } l).$
 $\text{state-val (CFG.transfer (lift-procs wfp) et } s) \ V =$
 $\text{state-val (CFG.transfer (lift-procs wfp) et } s') \ V$

<proof>

lemma *Proc-CFG-edge-rhs-pred-eq*:

assumes *prog* \vdash *Label l* $-IEdge et \rightarrow_p n'$ **and** *CFG.pred et s*
and $\forall V \in rhs (label\ prog\ l). state-val\ s\ V = state-val\ s'\ V$
and *length s = length s'*
shows *CFG.pred et s'*

<proof>

2.7.4 Instantiating the *CFG-wf* locale

interpretation *ProcCFG-wf*:

CFG-wf sourcenode targetnode kind valid-edge wfp (Main,Entry)
get-proc get-return-edges wfp lift-procs wfp Main
Def wfp Use wfp ParamDefs wfp ParamUses wfp
for *wfp*

<proof>

2.7.5 Instantiating the *CFGExit-wf* locale

interpretation *ProcCFGExit-wf*:

CFGExit-wf sourcenode targetnode kind valid-edge wfp (Main,Entry)
get-proc get-return-edges wfp lift-procs wfp Main (Main,Exit)
Def wfp Use wfp ParamDefs wfp ParamUses wfp
for *wfp*

<proof>

end

2.8 Lemmas concerning paths to instantiate locale Postdomination

theory *ValidPaths* **imports** *WellFormed ../StaticInter/Postdomination* **begin**

2.8.1 Intraprocedural paths from method entry and to method exit

abbreviation *path* $:: wf-prog \Rightarrow node \Rightarrow edge\ list \Rightarrow node \Rightarrow bool (- \vdash - \dashrightarrow^* -)$
where *wfp* $\vdash n -as \rightarrow^* n' \equiv CFG.path\ sourcenode\ targetnode (valid-edge\ wfp)\ n$
as n'

definition *label-incrs* $:: edge\ list \Rightarrow nat \Rightarrow edge\ list (- \oplus s - 60)$

where *as* $\oplus s\ i \equiv map (\lambda((p,n),et,(p',n')). ((p,n \oplus i),et,(p',n' \oplus i)))\ as$

declare *One-nat-def* [*simp del*]

From $prog$ to $prog;;c_2$

lemma *Proc-CFG-edge-SeqFirst-nodes-Label*:

$prog \vdash Label\ l -et\rightarrow_p\ Label\ l' \implies prog;;c_2 \vdash Label\ l -et\rightarrow_p\ Label\ l'$
 $\langle proof \rangle$

lemma *Proc-CFG-edge-SeqFirst-source-Label*:

assumes $prog \vdash Label\ l -et\rightarrow_p\ n'$
obtains nx **where** $prog;;c_2 \vdash Label\ l -et\rightarrow_p\ nx$
 $\langle proof \rangle$

lemma *Proc-CFG-edge-SeqFirst-target-Label*:

$\llbracket prog \vdash n -et\rightarrow_p\ n'; Label\ l' = n' \rrbracket \implies prog;;c_2 \vdash n -et\rightarrow_p\ Label\ l'$
 $\langle proof \rangle$

lemma *PCFG-edge-SeqFirst-source-Label*:

assumes $prog,procs \vdash (p,Label\ l) -et\rightarrow (p',n')$
obtains nx **where** $prog;;c_2,procs \vdash (p,Label\ l) -et\rightarrow (p',nx)$
 $\langle proof \rangle$

lemma *PCFG-edge-SeqFirst-target-Label*:

$prog,procs \vdash (p,n) -et\rightarrow (p',Label\ l')$
 $\implies prog;;c_2,procs \vdash (p,n) -et\rightarrow (p',Label\ l')$
 $\langle proof \rangle$

lemma *path-SeqFirst*:

assumes $Rep-wf-prog\ wfp = (prog,procs)$ **and** $Rep-wf-prog\ wfp' = (prog;;c_2,procs)$
shows $\llbracket wfp \vdash (p,n) -as\rightarrow^* (p,Label\ l); \forall a \in set\ as.\ intra-kind\ (kind\ a) \rrbracket$
 $\implies wfp' \vdash (p,n) -as\rightarrow^* (p,Label\ l)$
 $\langle proof \rangle$

From $prog$ to $c_1;;prog$

lemma *Proc-CFG-edge-SeqSecond-source-not-Entry*:

$\llbracket prog \vdash n -et\rightarrow_p\ n'; n \neq Entry \rrbracket \implies c_1;;prog \vdash n \oplus \#:c_1 -et\rightarrow_p\ n' \oplus \#:c_1$
 $\langle proof \rangle$

lemma *PCFG-Main-edge-SeqSecond-source-not-Entry*:

$\llbracket prog,procs \vdash (Main,n) -et\rightarrow (p',n'); n \neq Entry; intra-kind\ et; well-formed\ procs \rrbracket$
 $\implies c_1;;prog,procs \vdash (Main,n \oplus \#:c_1) -et\rightarrow (p',n' \oplus \#:c_1)$
 $\langle proof \rangle$

lemma *valid-node-Main-SeqSecond*:

assumes $CFG.valid\text{-}node\ sourcenode\ targetnode\ (valid\text{-}edge\ wfp)\ (Main, n)$
and $n \neq Entry$ **and** $Rep\text{-}wf\text{-}prog\ wfp = (prog, procs)$
and $Rep\text{-}wf\text{-}prog\ wfp' = (c_1;; prog, procs)$
shows $CFG.valid\text{-}node\ sourcenode\ targetnode\ (valid\text{-}edge\ wfp')\ (Main, n \oplus \#:c_1)$
 $\langle proof \rangle$

lemma *path-Main-SeqSecond*:

assumes $Rep\text{-}wf\text{-}prog\ wfp = (prog, procs)$ **and** $Rep\text{-}wf\text{-}prog\ wfp' = (c_1;; prog, procs)$
shows $\llbracket wfp \vdash (Main, n) -as \rightarrow^* (p', n'); \forall a \in set\ as.\ intra\text{-}kind\ (kind\ a); n \neq Entry \rrbracket$
 $\implies wfp' \vdash (Main, n \oplus \#:c_1) -as \oplus s \#:c_1 \rightarrow^* (p', n' \oplus \#:c_1)$
 $\langle proof \rangle$

From prog to if (b) prog else c2

lemma *Proc-CFG-edge-CondTrue-source-not-Entry*:

$\llbracket prog \vdash n -et \rightarrow_p n'; n \neq Entry \rrbracket \implies if\ (b)\ prog\ else\ c_2 \vdash n \oplus 1 -et \rightarrow_p n' \oplus 1$
 $\langle proof \rangle$

lemma *PCFG-Main-edge-CondTrue-source-not-Entry*:

$\llbracket prog, procs \vdash (Main, n) -et \rightarrow (p', n'); n \neq Entry; intra\text{-}kind\ et; well\text{-}formed\ procs \rrbracket$
 $\implies if\ (b)\ prog\ else\ c_2, procs \vdash (Main, n \oplus 1) -et \rightarrow (p', n' \oplus 1)$
 $\langle proof \rangle$

lemma *valid-node-Main-CondTrue*:

assumes $CFG.valid\text{-}node\ sourcenode\ targetnode\ (valid\text{-}edge\ wfp)\ (Main, n)$
and $n \neq Entry$ **and** $Rep\text{-}wf\text{-}prog\ wfp = (prog, procs)$
and $Rep\text{-}wf\text{-}prog\ wfp' = (if\ (b)\ prog\ else\ c_2, procs)$
shows $CFG.valid\text{-}node\ sourcenode\ targetnode\ (valid\text{-}edge\ wfp')\ (Main, n \oplus 1)$
 $\langle proof \rangle$

lemma *path-Main-CondTrue*:

assumes $Rep\text{-}wf\text{-}prog\ wfp = (prog, procs)$
and $Rep\text{-}wf\text{-}prog\ wfp' = (if\ (b)\ prog\ else\ c_2, procs)$
shows $\llbracket wfp \vdash (Main, n) -as \rightarrow^* (p', n'); \forall a \in set\ as.\ intra\text{-}kind\ (kind\ a); n \neq Entry \rrbracket$
 $\implies wfp' \vdash (Main, n \oplus 1) -as \oplus s 1 \rightarrow^* (p', n' \oplus 1)$
 $\langle proof \rangle$

From prog to if (b) c1 else prog

lemma *Proc-CFG-edge-CondFalse-source-not-Entry*:

$\llbracket prog \vdash n -et \rightarrow_p n'; n \neq Entry \rrbracket$
 $\implies if\ (b)\ c_1\ else\ prog \vdash n \oplus (\#:c_1 + 1) -et \rightarrow_p n' \oplus (\#:c_1 + 1)$
 $\langle proof \rangle$

lemma *PCFG-Main-edge-CondFalse-source-not-Entry*:

$\llbracket \text{prog,procs} \vdash (\text{Main},n) -et \rightarrow (p',n'); n \neq \text{Entry}; \text{intra-kind } et; \text{well-formed procs} \rrbracket$
 $\implies \text{if } (b) \ c_1 \ \text{else } \text{prog,procs} \vdash (\text{Main},n \oplus (\# : c_1 + 1)) -et \rightarrow (p',n' \oplus (\# : c_1 + 1))$
 $\langle \text{proof} \rangle$

lemma *valid-node-Main-CondFalse*:

assumes *CFG.valid-node sourcenode targetnode (valid-edge wfp) (Main,n)*
and $n \neq \text{Entry}$ **and** $\text{Rep-wf-prog wfp} = (\text{prog,procs})$
and $\text{Rep-wf-prog wfp}' = (\text{if } (b) \ c_1 \ \text{else } \text{prog,procs})$
shows *CFG.valid-node sourcenode targetnode (valid-edge wfp')*
 $(\text{Main}, n \oplus (\# : c_1 + 1))$
 $\langle \text{proof} \rangle$

lemma *path-Main-CondFalse*:

assumes $\text{Rep-wf-prog wfp} = (\text{prog,procs})$
and $\text{Rep-wf-prog wfp}' = (\text{if } (b) \ c_1 \ \text{else } \text{prog,procs})$
shows $\llbracket \text{wfp} \vdash (\text{Main},n) -as \rightarrow^* (p',n'); \forall a \in \text{set } as. \text{intra-kind } (kind \ a); n \neq \text{Entry} \rrbracket$
 $\implies \text{wfp}' \vdash (\text{Main},n \oplus (\# : c_1 + 1)) -as \oplus s (\# : c_1 + 1) \rightarrow^* (p',n' \oplus (\# : c_1 + 1))$
 $\langle \text{proof} \rangle$

From prog to while (b) prog

lemma *Proc-CFG-edge-WhileBody-source-not-Entry*:

$\llbracket \text{prog} \vdash n -et \rightarrow_p n'; n \neq \text{Entry}; n' \neq \text{Exit} \rrbracket$
 $\implies \text{while } (b) \ \text{prog} \vdash n \oplus 2 -et \rightarrow_p n' \oplus 2$
 $\langle \text{proof} \rangle$

lemma *PCFG-Main-edge-WhileBody-source-not-Entry*:

$\llbracket \text{prog,procs} \vdash (\text{Main},n) -et \rightarrow (p',n'); n \neq \text{Entry}; n' \neq \text{Exit}; \text{intra-kind } et; \text{well-formed procs} \rrbracket \implies \text{while } (b) \ \text{prog,procs} \vdash (\text{Main},n \oplus 2) -et \rightarrow (p',n' \oplus 2)$
 $\langle \text{proof} \rangle$

lemma *valid-node-Main-WhileBody*:

assumes *CFG.valid-node sourcenode targetnode (valid-edge wfp) (Main,n)*
and $n \neq \text{Entry}$ **and** $\text{Rep-wf-prog wfp} = (\text{prog,procs})$
and $\text{Rep-wf-prog wfp}' = (\text{while } (b) \ \text{prog,procs})$
shows *CFG.valid-node sourcenode targetnode (valid-edge wfp')*
 $(\text{Main}, n \oplus 2)$
 $\langle \text{proof} \rangle$

lemma *path-Main-WhileBody*:

assumes $\text{Rep-wf-prog wfp} = (\text{prog,procs})$

and $\text{Rep-wf-prog wfp}' = (\text{while } (b) \text{ prog,procs})$
shows $\llbracket \text{wfp} \vdash (\text{Main},n) -\text{as}\rightarrow^* (p',n'); \forall a \in \text{set as. intra-kind } (kind\ a);$
 $n \neq \text{Entry}; n' \neq \text{Exit} \rrbracket \implies \text{wfp}' \vdash (\text{Main},n \oplus 2) -\text{as} \oplus s\ 2 \rightarrow^* (p',n' \oplus 2)$
 $\langle \text{proof} \rangle$

Existence of intraprocedural paths

lemma *Label-Proc-CFG-Entry-Exit-path-Main*:
assumes $\text{Rep-wf-prog wfp} = (\text{prog,procs})$ **and** $l < \#:\text{prog}$
obtains $\text{as as}'$ **where** $\text{wfp} \vdash (\text{Main},\text{Label } l) -\text{as}\rightarrow^* (\text{Main},\text{Exit})$
and $\forall a \in \text{set as. intra-kind } (kind\ a)$
and $\text{wfp} \vdash (\text{Main},\text{Entry}) -\text{as}'\rightarrow^* (\text{Main},\text{Label } l)$
and $\forall a \in \text{set as}'. intra-kind } (kind\ a)$
 $\langle \text{proof} \rangle$

2.8.2 Lifting from edges in procedure Main to arbitrary procedures

lemma *lift-edge-Main-Main*:
 $\llbracket c,\text{procs} \vdash (\text{Main}, n) -\text{et}\rightarrow (\text{Main}, n'); (p,\text{ins},\text{outs},c) \in \text{set procs};$
 $\text{containsCall procs prog ps } p; \text{well-formed procs} \rrbracket$
 $\implies \text{prog,procs} \vdash (p, n) -\text{et}\rightarrow (p, n')$
 $\langle \text{proof} \rangle$

lemma *lift-edge-Main-Proc*:
 $\llbracket c,\text{procs} \vdash (\text{Main}, n) -\text{et}\rightarrow (q, n'); q \neq \text{Main}; (p,\text{ins},\text{outs},c) \in \text{set procs};$
 $\text{containsCall procs prog ps } p; \text{well-formed procs} \rrbracket$
 $\implies \exists \text{et}'. \text{prog,procs} \vdash (p, n) -\text{et}'\rightarrow (q, n')$
 $\langle \text{proof} \rangle$

lemma *lift-edge-Proc-Main*:
 $\llbracket c,\text{procs} \vdash (q, n) -\text{et}\rightarrow (\text{Main}, n'); q \neq \text{Main}; (p,\text{ins},\text{outs},c) \in \text{set procs};$
 $\text{containsCall procs prog ps } p; \text{well-formed procs} \rrbracket$
 $\implies \exists \text{et}'. \text{prog,procs} \vdash (q, n) -\text{et}'\rightarrow (p, n')$
 $\langle \text{proof} \rangle$

fun *lift-edge* :: $\text{edge} \Rightarrow \text{pname} \Rightarrow \text{edge}$
where $\text{lift-edge } a\ p = ((p,\text{snd}(\text{sourcenode } a)),\text{kind } a,(p,\text{snd}(\text{targetnode } a)))$

fun *lift-path* :: $\text{edge list} \Rightarrow \text{pname} \Rightarrow \text{edge list}$
where $\text{lift-path } \text{as } p = \text{map } (\lambda a. \text{lift-edge } a\ p) \text{ as}$

lemma *lift-path-Proc*:
assumes $\text{Rep-wf-prog wfp}' = (c,\text{procs})$ **and** $\text{Rep-wf-prog wfp} = (\text{prog,procs})$
and $(p,\text{ins},\text{outs},c) \in \text{set procs}$ **and** $\text{containsCall procs prog ps } p$
shows $\llbracket \text{wfp}' \vdash (\text{Main},n) -\text{as}\rightarrow^* (\text{Main},n'); \forall a \in \text{set as. intra-kind } (kind\ a) \rrbracket$
 $\implies \text{wfp} \vdash (p,n) -\text{lift-path } \text{as } p \rightarrow^* (p,n')$
 $\langle \text{proof} \rangle$

2.8.3 Existence of paths from Entry and to Exit

lemma *Label-Proc-CFG-Entry-Exit-path-Proc:*

assumes *Rep-wf-prog* $wfp = (prog, procs)$ **and** $l < \# : c$
and $(p, ins, outs, c) \in set\ procs$ **and** *containsCall* $procs\ prog\ ps\ p$
obtains $as\ as'$ **where** $wfp \vdash (p, Label\ l) -as \rightarrow^* (p, Exit)$
and $\forall a \in set\ as.$ *intra-kind* $(kind\ a)$
and $wfp \vdash (p, Entry) -as' \rightarrow^* (p, Label\ l)$
and $\forall a \in set\ as'.$ *intra-kind* $(kind\ a)$

<proof>

lemma *Entry-to-Entry-and-Exit-to-Exit:*

assumes *Rep-wf-prog* $wfp = (prog, procs)$
and *containsCall* $procs\ prog\ ps\ p$ **and** $(p, ins, outs, c) \in set\ procs$
obtains $as\ as'$ **where** *CFG.valid-path'* $sourcenode\ targetnode\ kind$
 $(valid-edge\ wfp)\ (get-return-edges\ wfp)\ (Main, Entry)\ as\ (p, Entry)$
and *CFG.valid-path'* $sourcenode\ targetnode\ kind$
 $(valid-edge\ wfp)\ (get-return-edges\ wfp)\ (p, Exit)\ as'\ (Main, Exit)$

<proof>

lemma *edge-valid-paths:*

assumes $prog, procs \vdash sourcenode\ a -kind\ a \rightarrow targetnode\ a$
and $disj:(p, n) = sourcenode\ a \vee (p, n) = targetnode\ a$
and $[simp]: Rep-wf-prog\ wfp = (prog, procs)$
shows $\exists as\ as'.$ *CFG.valid-path'* $sourcenode\ targetnode\ kind\ (valid-edge\ wfp)$
 $(get-return-edges\ wfp)\ (Main, Entry)\ as\ (p, n) \wedge$
 $CFG.valid-path'\ sourcenode\ targetnode\ kind\ (valid-edge\ wfp)$
 $(get-return-edges\ wfp)\ (p, n)\ as'\ (Main, Exit)$

<proof>

2.8.4 Instantiating the Postdomination locale

interpretation *ProcPostdomination:*

Postdomination $sourcenode\ targetnode\ kind\ valid-edge\ wfp\ (Main, Entry)$
 $get-proc\ get-return-edges\ wfp\ lift-procs\ wfp\ Main\ (Main, Exit)$
for wfp

<proof>

end

2.9 Instantiation of the SDG locale

theory *ProcSDG* **imports** *ValidPaths* *../StaticInter/SDG* **begin**

interpretation *Proc-SDG:*

SDG $sourcenode\ targetnode\ kind\ valid-edge\ wfp\ (Main, Entry)$

```
get-proc get-return-edges wfp lift-procs wfp Main (Main,Exit)  
Def wfp Use wfp ParamDefs wfp ParamUses wfp  
for wfp <proof>  
end
```

Chapter 3

A Control Flow Graph for Jinja Byte Code

3.1 Formalizing the CFG

```
theory JVMCFG imports ../StaticInter/BasicDefs Jinja.BVExample begin
```

```
declare lesub-list-impl-same-size [simp del]
```

```
declare nlistsE-length [simp del]
```

3.1.1 Type definitions

Wellformed Programs

```
definition wf-jvmprog = {(P, Phi). wf-jvm-progPhi P}
```

```
typedef wf-jvmprog = wf-jvmprog  
⟨proof⟩
```

```
hide-const Phi E
```

```
abbreviation PROG :: wf-jvmprog ⇒ jvm-prog  
where PROG P ≡ fst(Rep-wf-jvmprog(P))
```

```
abbreviation TYPING :: wf-jvmprog ⇒ tyP  
where TYPING P ≡ snd(Rep-wf-jvmprog(P))
```

```
lemma wf-jvmprog-is-wf-ty: wf-jvm-progTYPING P (PROG P)  
⟨proof⟩
```

```
lemma wf-jvmprog-is-wf: wf-jvm-prog (PROG P)  
⟨proof⟩
```

Interprocedural CFG

type-synonym *jvm-method* = *wf-jvmprog* × *cname* × *mname*

datatype *var* = *Heap* | *Local nat* | *Stack nat* | *Exception*

datatype *val* = *Hp heap* | *Value Value.val*

type-synonym *state* = *var* → *val*

definition *valid-state* :: *state* ⇒ *bool*

where *valid-state* *s* ≡ (∀ *val*. *s Heap* ≠ *Some (Value val)*)

∧ (*s Exception* = *None* ∨ (∃ *addr*. *s Exception* = *Some (Value (Addr addr))*))

∧ (∀ *var*. *var* ≠ *Heap* ∧ *var* ≠ *Exception* → (∀ *h*. *s var* ≠ *Some (Hp h)*))

fun *the-Heap* :: *val* ⇒ *heap*

where *the-Heap* (*Hp h*) = *h*

fun *the-Value* :: *val* ⇒ *Value.val*

where *the-Value* (*Value v*) = *v*

abbreviation *heap-of* :: *state* ⇒ *heap*

where *heap-of* *s* ≡ *the-Heap (the (s Heap))*

abbreviation *exc-flag* :: *state* ⇒ *addr option*

where *exc-flag* *s* ≡ *case (s Exception) of None* ⇒ *None*

| *Some v* ⇒ *Some (THE a. v = Value (Addr a))*

abbreviation *stkAt* :: *state* ⇒ *nat* ⇒ *Value.val*

where *stkAt* *s n* ≡ *the-Value (the (s (Stack n)))*

abbreviation *locAt* :: *state* ⇒ *nat* ⇒ *Value.val*

where *locAt* *s n* ≡ *the-Value (the (s (Local n)))*

datatype *nodeType* = *Enter* | *Normal* | *Return* | *Exceptional pc option nodeType*

type-synonym *cfg-node* = *cname* × *mname* × *pc option* × *nodeType*

type-synonym

cfg-edge = *cfg-node* × (*var*, *val*, *cname* × *mname* × *pc*, *cname* × *mname*)
edge-kind × *cfg-node*

definition *ClassMain* :: *wf-jvmprog* ⇒ *cname*

where *ClassMain* *P* ≡ *SOME Name. ¬ is-class (PROG P) Name*

definition *MethodMain* :: *wf-jvmprog* ⇒ *mname*

where *MethodMain* *P* ≡ *SOME Name.*

∀ *C D fs ms*. *class (PROG P) C* = [(*D*, *fs*, *ms*)] → (∀ *m* ∈ *set ms*. *Name* ≠ *fst m*)

definition *stkLength* :: *jvm-method* ⇒ *pc* ⇒ *nat*

where

stkLength *m pc* ≡ *let (P, C, M) = m in* (

if (C = ClassMain P) then 1 else (
 length (fst(the(((TYPING P) C M) ! pc)))
))

definition locLength :: jvm-method \Rightarrow pc \Rightarrow nat

where
 locLength m pc \equiv let (P, C, M) = m in (
 if (C = ClassMain P) then 1 else (
 length (snd(the(((TYPING P) C M) ! pc)))
))

lemma ex-new-class-name: $\exists C. \neg$ is-class P C
 <proof>

lemma ClassMain-unique-in-P:
assumes is-class (PROG P) C
shows ClassMain P \neq C
 <proof>

lemma map-of-fstD: \llbracket map-of xs a = [b]; $\forall x \in$ set xs. fst x \neq a $\rrbracket \Longrightarrow$ False
 <proof>

lemma map-of-fstE: \llbracket map-of xs a = [b]; $\exists x \in$ set xs. fst x = a $\rrbracket \Longrightarrow$ thesis
 <proof>

lemma ex-unique-method-name:
 \exists Name. $\forall C D fs ms.$ class (PROG P) C = [(D, fs, ms)] \longrightarrow ($\forall m \in$ set ms. Name
 \neq fst m)
 <proof>

lemma MethodMain-unique-in-P:
assumes PROG P \vdash D sees M:Ts \rightarrow T = mb in C
shows MethodMain P \neq M
 <proof>

lemma ClassMain-is-no-class [dest!]: is-class (PROG P) (ClassMain P) \Longrightarrow False
 <proof>

lemma MethodMain-not-seen [dest!]: PROG P \vdash C sees (MethodMain P):Ts \rightarrow T
 = mb in D \Longrightarrow False
 <proof>

lemma no-Call-from-ClassMain [dest!]: PROG P \vdash ClassMain P sees M:Ts \rightarrow T
 = mb in C \Longrightarrow False
 <proof>

lemma no-Call-in-ClassMain [dest!]: PROG P \vdash C sees M:Ts \rightarrow T = mb in Class-
 Main P \Longrightarrow False

$\langle \text{proof} \rangle$

inductive $JVMCFG :: \text{jvm-method} \Rightarrow \text{cfg-node} \Rightarrow (\text{var}, \text{val}, \text{cname} \times \text{mname} \times \text{pc}, \text{cname} \times \text{mname}) \text{ edge-kind} \Rightarrow \text{cfg-node} \Rightarrow \text{bool} \ (- \vdash - \dashrightarrow -)$
and $\text{reachable} :: \text{jvm-method} \Rightarrow \text{cfg-node} \Rightarrow \text{bool} \ (- \vdash \Rightarrow -)$
where
 Entry-reachable: $(P, C0, \text{Main}) \vdash \Rightarrow (\text{ClassMain } P, \text{MethodMain } P, \text{None}, \text{Enter})$
 | reachable-step: $\llbracket P \vdash \Rightarrow n; P \vdash n \text{ -(e)} \rightarrow n' \rrbracket \Longrightarrow P \vdash \Rightarrow n'$
 | Main-to-Call: $(P, C0, \text{Main}) \vdash \Rightarrow (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Enter})$
 $\Longrightarrow (P, C0, \text{Main}) \vdash (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Enter}) \text{ --}\uparrow \text{id} \rightarrow (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Normal})$
 | Main-Call-LFalse: $(P, C0, \text{Main}) \vdash \Rightarrow (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Normal})$
 $\Longrightarrow (P, C0, \text{Main}) \vdash (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Normal}) \text{ --}(\lambda s. \text{False})_{\surd} \rightarrow (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Return})$
 | Main-Call: $\llbracket (P, C0, \text{Main}) \vdash \Rightarrow (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Normal});$
 $\text{PROG } P \vdash C0 \text{ sees Main:} \llbracket \rightarrow T = (\text{mxs}, \text{mxl}_0, \text{is}, \text{xt}) \text{ in } D;$
 $\text{initParams} = \llbracket (\lambda s. s \text{ Heap}), (\lambda s. \lfloor \text{Value Null} \rfloor) \rrbracket;$
 $\text{ek} = (\lambda (s, \text{ret}). \text{True}): (\text{ClassMain } P, \text{MethodMain } P, 0) \hookrightarrow (D, \text{Main}) \text{ initParams}$
 \rrbracket
 $\Longrightarrow (P, C0, \text{Main}) \vdash (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Normal}) \text{ --}(\text{ek}) \rightarrow (D, \text{Main}, \text{None}, \text{Enter})$
 | Main-Return-to-Exit: $(P, C0, \text{Main}) \vdash \Rightarrow (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Return})$
 $\Longrightarrow (P, C0, \text{Main}) \vdash (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Return}) \text{ --}(\uparrow \text{id}) \rightarrow (\text{ClassMain } P, \text{MethodMain } P, \text{None}, \text{Return})$
 | Method-LFalse: $(P, C0, \text{Main}) \vdash \Rightarrow (C, M, \text{None}, \text{Enter})$
 $\Longrightarrow (P, C0, \text{Main}) \vdash (C, M, \text{None}, \text{Enter}) \text{ --}(\lambda s. \text{False})_{\surd} \rightarrow (C, M, \text{None}, \text{Return})$
 | Method-LTrue: $(P, C0, \text{Main}) \vdash \Rightarrow (C, M, \text{None}, \text{Enter})$
 $\Longrightarrow (P, C0, \text{Main}) \vdash (C, M, \text{None}, \text{Enter}) \text{ --}(\lambda s. \text{True})_{\surd} \rightarrow (C, M, \lfloor 0 \rfloor, \text{Enter})$
 | CFG-Load: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor \text{pc} \rfloor, \text{Enter});$
 $\text{instrs-of } (\text{PROG } P) C M ! \text{pc} = \text{Load } n;$
 $\text{ek} = \uparrow (\lambda s. s(\text{Stack } (\text{stkLength } (P, C, M) \text{pc}) := s(\text{Local } n))) \rrbracket$
 $\Longrightarrow (P, C0, \text{Main}) \vdash (C, M, \lfloor \text{pc} \rfloor, \text{Enter}) \text{ --}(\text{ek}) \rightarrow (C, M, \lfloor \text{Suc } \text{pc} \rfloor, \text{Enter})$
 | CFG-Store: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor \text{pc} \rfloor, \text{Enter});$
 $\text{instrs-of } (\text{PROG } P) C M ! \text{pc} = \text{Store } n;$
 $\text{ek} = \uparrow (\lambda s. s(\text{Local } n := s(\text{Stack } (\text{stkLength } (P, C, M) \text{pc} - 1)))) \rrbracket$
 $\Longrightarrow (P, C0, \text{Main}) \vdash (C, M, \lfloor \text{pc} \rfloor, \text{Enter}) \text{ --}(\text{ek}) \rightarrow (C, M, \lfloor \text{Suc } \text{pc} \rfloor, \text{Enter})$
 | CFG-Push: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor \text{pc} \rfloor, \text{Enter});$
 $\text{instrs-of } (\text{PROG } P) C M ! \text{pc} = \text{Push } v;$
 $\text{ek} = \uparrow (\lambda s. s(\text{Stack } (\text{stkLength } (P, C, M) \text{pc}) \mapsto \text{Value } v)) \rrbracket$
 $\Longrightarrow (P, C0, \text{Main}) \vdash (C, M, \lfloor \text{pc} \rfloor, \text{Enter}) \text{ --}(\text{ek}) \rightarrow (C, M, \lfloor \text{Suc } \text{pc} \rfloor, \text{Enter})$
 | CFG-Pop: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor \text{pc} \rfloor, \text{Enter}); \text{instrs-of}$
 $(\text{PROG } P) C M ! \text{pc} = \text{Pop};$
 $\text{ek} = \uparrow \text{id} \rrbracket$
 $\Longrightarrow (P, C0, \text{Main}) \vdash (C, M, \lfloor \text{pc} \rfloor, \text{Enter}) \text{ --}(\text{ek}) \rightarrow (C, M, \lfloor \text{Suc } \text{pc} \rfloor, \text{Enter})$
 | CFG-IAdd: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor \text{pc} \rfloor, \text{Enter});$

$instrs\text{-of } (PROG P) C M ! pc = IAdd;$
 $ek = \uparrow(\lambda s. let i1 = the\text{-Intg } (stkAt s (stkLength (P, C, M) pc - 1));$
 $i2 = the\text{-Intg } (stkAt s (stkLength (P, C, M) pc - 2))$
 $in s(Stack (stkLength (P, C, M) pc - 2) \mapsto Value (Intg (i1 + i2))))$
 \Downarrow
 $\implies (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter) \text{---}(ek) \rightarrow (C, M, \lfloor Suc pc \rfloor, Enter)$
 $| CFG\text{-Goto: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Enter);$
 $instrs\text{-of } (PROG P) C M ! pc = Goto i \rrbracket$
 $\implies (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter) \text{---}((\lambda s. True)_{\checkmark}) \rightarrow (C, M, \lfloor nat (int$
 $pc + i) \rfloor, Enter)$
 $| CFG\text{-CmpEq: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Enter);$
 $instrs\text{-of } (PROG P) C M ! pc = CmpEq;$
 $ek = \uparrow(\lambda s. let e1 = stkAt s (stkLength (P, C, M) pc - 1);$
 $e2 = stkAt s (stkLength (P, C, M) pc - 2)$
 $in s(Stack (stkLength (P, C, M) pc - 2) \mapsto Value (Bool (e1 = e2))))$
 \Downarrow
 $\implies (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter) \text{---}(ek) \rightarrow (C, M, \lfloor Suc pc \rfloor, Enter)$
 $| CFG\text{-IfFalse-False: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Enter);$
 $instrs\text{-of } (PROG P) C M ! pc = IfFalse i;$
 $i \neq 1;$
 $ek = (\lambda s. stkAt s (stkLength(P, C, M) pc - 1) = Bool False)_{\checkmark}$
 $\implies (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter) \text{---}(ek) \rightarrow (C, M, \lfloor nat (int pc + i) \rfloor,$
 $Enter)$
 $| CFG\text{-IfFalse-True: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Enter);$
 $instrs\text{-of } (PROG P) C M ! pc = IfFalse i;$
 $ek = (\lambda s. stkAt s (stkLength(P, C, M) pc - 1) \neq Bool False \vee i = 1)_{\checkmark}$
 $\implies (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter) \text{---}(ek) \rightarrow (C, M, \lfloor Suc pc \rfloor, Enter)$
 $| CFG\text{-New-Check-Normal: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor,$
 $Enter);$
 $instrs\text{-of } (PROG P) C M ! pc = New Cl;$
 $ek = (\lambda s. new\text{-Addr } (heap\text{-of } s) \neq None)_{\checkmark}$
 $\implies (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter) \text{---}(ek) \rightarrow (C, M, \lfloor pc \rfloor, Normal)$
 $| CFG\text{-New-Check-Exceptional: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M,$
 $\lfloor pc \rfloor, Enter);$
 $instrs\text{-of } (PROG P) C M ! pc = New Cl;$
 $pc' = (case (match\text{-ex-table } (PROG P) OutOfMemory pc (ex\text{-table-of } (PROG$
 $P) C M)) of$
 $None \Rightarrow None$
 $| Some (pc'', d) \Rightarrow \lfloor pc'' \rfloor);$
 $ek = (\lambda s. new\text{-Addr } (heap\text{-of } s) = None)_{\checkmark}$
 $\implies (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter) \text{---}(ek) \rightarrow (C, M, \lfloor pc \rfloor, Exceptional pc'$
 $Enter)$
 $| CFG\text{-New-Update: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Nor-$
 $mal);$
 $instrs\text{-of } (PROG P) C M ! pc = New Cl;$
 $ek = \uparrow(\lambda s. let a = the (new\text{-Addr } (heap\text{-of } s))$
 $in s(Heap \mapsto Hp ((heap\text{-of } s)(a \mapsto blank (PROG P) Cl)),$
 $Stack (stkLength(P, C, M) pc) \mapsto Value (Addr a)))$
 $\implies (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Normal) \text{---}(ek) \rightarrow (C, M, \lfloor Suc pc \rfloor, Enter)$

| *CFG-New-Exceptional-prop*: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, \text{Exceptional None Enter});$
 $\text{instrs-of } (\text{PROG } P) C M ! pc = \text{New Cl};$
 $ek = \uparrow(\lambda s. s(\text{Exception} \mapsto \text{Value } (\text{Addr } (\text{addr-of-sys-xcpt } \text{OutOfMemory})))) \rrbracket$
 $\Rightarrow (P, C0, \text{Main}) \vdash (C, M, \lfloor pc \rfloor, \text{Exceptional None Enter}) \text{---}(ek) \rightarrow (C, M, \text{None}, \text{Return})$

| *CFG-New-Exceptional-handle*: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, \text{Exceptional } \lfloor pc' \rfloor \text{ Enter});$
 $\text{instrs-of } (\text{PROG } P) C M ! pc = \text{New Cl};$
 $ek = \uparrow(\lambda s. (s(\text{Exception} := \text{None}))$
 $\quad (\text{Stack } (\text{stkLength } (P, C, M) pc' - 1) \mapsto \text{Value } (\text{Addr } (\text{addr-of-sys-xcpt } \text{OutOfMemory})))) \rrbracket$
 $\Rightarrow (P, C0, \text{Main}) \vdash (C, M, \lfloor pc \rfloor, \text{Exceptional } \lfloor pc' \rfloor \text{ Enter}) \text{---}(ek) \rightarrow (C, M, \lfloor pc' \rfloor, \text{Enter})$

| *CFG-Getfield-Check-Normal*: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, \text{Enter});$
 $\text{instrs-of } (\text{PROG } P) C M ! pc = \text{Getfield } F \text{ Cl};$
 $ek = (\lambda s. \text{stkAt } s (\text{stkLength } (P, C, M) pc - 1) \neq \text{Null})_{\checkmark} \rrbracket$
 $\Rightarrow (P, C0, \text{Main}) \vdash (C, M, \lfloor pc \rfloor, \text{Enter}) \text{---}(ek) \rightarrow (C, M, \lfloor pc \rfloor, \text{Normal})$

| *CFG-Getfield-Check-Exceptional*: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, \text{Enter});$
 $\text{instrs-of } (\text{PROG } P) C M ! pc = \text{Getfield } F \text{ Cl};$
 $pc' = (\text{case } (\text{match-ex-table } (\text{PROG } P) \text{NullPointer } pc (\text{ex-table-of } (\text{PROG } P) C M)) \text{ of}$
 $\quad \text{None} \Rightarrow \text{None}$
 $\quad | \text{Some } (pc'', d) \Rightarrow \lfloor pc'' \rfloor);$
 $ek = (\lambda s. \text{stkAt } s (\text{stkLength } (P, C, M) pc - 1) = \text{Null})_{\checkmark} \rrbracket$
 $\Rightarrow (P, C0, \text{Main}) \vdash (C, M, \lfloor pc \rfloor, \text{Enter}) \text{---}(ek) \rightarrow (C, M, \lfloor pc \rfloor, \text{Exceptional } pc' \text{ Enter})$

| *CFG-Getfield-Update*: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, \text{Normal});$
 $\text{instrs-of } (\text{PROG } P) C M ! pc = \text{Getfield } F \text{ Cl};$
 $ek = \uparrow(\lambda s. \text{let } (D, fs) = \text{the } (\text{heap-of } s (\text{the-Addr } (\text{stkAt } s (\text{stkLength } (P, C, M) pc - 1))))$
 $\quad \text{in } s(\text{Stack } (\text{stkLength}(P, C, M) pc - 1) \mapsto \text{Value } (\text{the } (fs (F, Cl)))))) \rrbracket$
 $\Rightarrow (P, C0, \text{Main}) \vdash (C, M, \lfloor pc \rfloor, \text{Normal}) \text{---}(ek) \rightarrow (C, M, \lfloor \text{Suc } pc \rfloor, \text{Enter})$

| *CFG-Getfield-Exceptional-prop*: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, \text{Exceptional None Enter});$
 $\text{instrs-of } (\text{PROG } P) C M ! pc = \text{Getfield } F \text{ Cl};$
 $ek = \uparrow(\lambda s. s(\text{Exception} \mapsto \text{Value } (\text{Addr } (\text{addr-of-sys-xcpt } \text{NullPointer})))) \rrbracket$
 $\Rightarrow (P, C0, \text{Main}) \vdash (C, M, \lfloor pc \rfloor, \text{Exceptional None Enter}) \text{---}(ek) \rightarrow (C, M, \text{None}, \text{Return})$

| *CFG-Getfield-Exceptional-handle*: $\llbracket C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, \text{Exceptional } \lfloor pc' \rfloor \text{ Enter});$
 $\text{instrs-of } (\text{PROG } P) C M ! pc = \text{Getfield } F \text{ Cl};$
 $ek = \uparrow(\lambda s. (s(\text{Exception} := \text{None}))$
 $\quad (\text{Stack } (\text{stkLength } (P, C, M) pc' - 1) \mapsto \text{Value } (\text{Addr } (\text{addr-of-sys-xcpt } \text{NullPointer})))) \rrbracket$

$\Rightarrow (P, C0, Main) \vdash (C, M, [pc], \text{Exceptional } [pc'] \text{ Enter}) \text{---}(ek) \rightarrow (C, M, [pc'], \text{Enter})$
| CFG-Putfield-Check-Normal: $\llbracket C \neq \text{ClassMain } P; (P, C0, Main) \vdash \Rightarrow (C, M, [pc], \text{Enter});$
instrs-of (PROG P) C M ! pc = Putfield F Cl;
ek = $(\lambda s. \text{stkAt } s (\text{stkLength } (P, C, M) \text{ pc} - 2) \neq \text{Null})_{\checkmark} \rrbracket$
 $\Rightarrow (P, C0, Main) \vdash (C, M, [pc], \text{Enter}) \text{---}(ek) \rightarrow (C, M, [pc], \text{Normal})$
| CFG-Putfield-Check-Exceptional: $\llbracket C \neq \text{ClassMain } P; (P, C0, Main) \vdash \Rightarrow (C, M, [pc], \text{Enter});$
instrs-of (PROG P) C M ! pc = Putfield F Cl;
pc' = (case (match-ex-table (PROG P) NullPointer pc (ex-table-of (PROG P) C M)) of
 None \Rightarrow None
 | Some (pc'', d) \Rightarrow [pc'']);
ek = $(\lambda s. \text{stkAt } s (\text{stkLength } (P, C, M) \text{ pc} - 2) = \text{Null})_{\checkmark} \rrbracket$
 $\Rightarrow (P, C0, Main) \vdash (C, M, [pc], \text{Enter}) \text{---}(ek) \rightarrow (C, M, [pc], \text{Exceptional } pc' \text{ Enter})$
| CFG-Putfield-Update: $\llbracket C \neq \text{ClassMain } P; (P, C0, Main) \vdash \Rightarrow (C, M, [pc], \text{Normal});$
instrs-of (PROG P) C M ! pc = Putfield F Cl;
ek = $\uparrow(\lambda s. \text{let } v = \text{stkAt } s (\text{stkLength } (P, C, M) \text{ pc} - 1);$
 r = $\text{stkAt } s (\text{stkLength } (P, C, M) \text{ pc} - 2);$
 a = the-Addr r;
 (D, fs) = the (heap-of s a);
 h' = (heap-of s)(a \mapsto (D, fs((F, Cl) \mapsto v)))
 in s(Heap \mapsto Hp h')) \rrbracket
 $\Rightarrow (P, C0, Main) \vdash (C, M, [pc], \text{Normal}) \text{---}(ek) \rightarrow (C, M, [Suc \text{ pc}], \text{Enter})$
| CFG-Putfield-Exceptional-prop: $\llbracket C \neq \text{ClassMain } P; (P, C0, Main) \vdash \Rightarrow (C, M, [pc], \text{Exceptional None Enter});$
instrs-of (PROG P) C M ! pc = Putfield F Cl;
ek = $\uparrow(\lambda s. s(\text{Exception} \mapsto \text{Value } (\text{Addr } (\text{addr-of-sys-xcpt } \text{NullPointer})))) \rrbracket$
 $\Rightarrow (P, C0, Main) \vdash (C, M, [pc], \text{Exceptional None Enter}) \text{---}(ek) \rightarrow (C, M, \text{None}, \text{Return})$
| CFG-Putfield-Exceptional-handle: $\llbracket C \neq \text{ClassMain } P; (P, C0, Main) \vdash \Rightarrow (C, M, [pc], \text{Exceptional } [pc'] \text{ Enter});$
instrs-of (PROG P) C M ! pc = Putfield F Cl;
ek = $\uparrow(\lambda s. (s(\text{Exception} := \text{None}))$
 (Stack (stkLength (P, C, M) pc' - 1) \mapsto Value (Addr (addr-of-sys-xcpt NullPointer)))) \rrbracket
 $\Rightarrow (P, C0, Main) \vdash (C, M, [pc], \text{Exceptional } [pc'] \text{ Enter}) \text{---}(ek) \rightarrow (C, M, [pc'], \text{Enter})$
| CFG-Checkcast-Check-Normal: $\llbracket C \neq \text{ClassMain } P; (P, C0, Main) \vdash \Rightarrow (C, M, [pc], \text{Enter});$
instrs-of (PROG P) C M ! pc = Checkcast Cl;
ek = $(\lambda s. \text{cast-ok } (\text{PROG } P) \text{ Cl } (\text{heap-of } s) (\text{stkAt } s (\text{stkLength } (P, C, M) \text{ pc} - 1)))_{\checkmark} \rrbracket$
 $\Rightarrow (P, C0, Main) \vdash (C, M, [pc], \text{Enter}) \text{---}(ek) \rightarrow (C, M, [Suc \text{ pc}], \text{Enter})$
| CFG-Checkcast-Check-Exceptional: $\llbracket C \neq \text{ClassMain } P; (P, C0, Main) \vdash \Rightarrow (C, M, [pc], \text{Enter});$

$instrs\text{-of } (PROG P) C M ! pc = Checkcast Cl;$
 $pc' = (case (match\text{-ex-table } (PROG P) ClassCast pc (ex\text{-table-of } (PROG P) C M)) of$
 $None \Rightarrow None$
 $| Some (pc'', d) \Rightarrow \lfloor pc'' \rfloor);$
 $ek = (\lambda s. \neg cast\text{-ok } (PROG P) Cl (heap\text{-of } s) (stkAt s (stkLength (P, C, M) pc - 1))) \checkmark \rfloor$
 $\Rightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter) \text{--}(ek) \rightarrow (C, M, \lfloor pc \rfloor, Exceptional pc' Enter)$
 $| CFG\text{-Checkcast-Exceptional-prop: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Exceptional None Enter);$
 $instrs\text{-of } (PROG P) C M ! pc = Checkcast Cl;$
 $ek = \uparrow(\lambda s. s(Exception \mapsto Value (Addr (addr\text{-of-sys-xcpt } ClassCast)))) \rfloor$
 $\Rightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Exceptional None Enter) \text{--}(ek) \rightarrow (C, M, None, Return)$
 $| CFG\text{-Checkcast-Exceptional-handle: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Exceptional \lfloor pc' \rfloor Enter);$
 $instrs\text{-of } (PROG P) C M ! pc = Checkcast Cl;$
 $ek = \uparrow(\lambda s. (s(Exception := None))$
 $(Stack (stkLength (P, C, M) pc' - 1) \mapsto Value (Addr (addr\text{-of-sys-xcpt } ClassCast)))) \rfloor$
 $\Rightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Exceptional \lfloor pc' \rfloor Enter) \text{--}(ek) \rightarrow (C, M, \lfloor pc' \rfloor, Enter)$
 $| CFG\text{-Throw-Check: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Enter);$
 $instrs\text{-of } (PROG P) C M ! pc = Throw;$
 $pc' = None \vee match\text{-ex-table } (PROG P) Exc pc (ex\text{-table-of } (PROG P) C M)$
 $= \lfloor (the pc', d) \rfloor;$
 $ek = (\lambda s. let v = stkAt s (stkLength (P, C, M) pc - 1);$
 $Cl = if (v = Null) then NullPointer else (cname\text{-of } (heap\text{-of } s)$
 $(the-Addr v))$
 $in case pc' of$
 $None \Rightarrow match\text{-ex-table } (PROG P) Cl pc (ex\text{-table-of } (PROG P) C M) = None$
 $| Some pc'' \Rightarrow \exists d. match\text{-ex-table } (PROG P) Cl pc (ex\text{-table-of } (PROG P) C M)$
 $= \lfloor (pc'', d) \rfloor$
 $) \checkmark \rfloor$
 $\Rightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter) \text{--}(ek) \rightarrow (C, M, \lfloor pc \rfloor, Exceptional pc' Enter)$
 $| CFG\text{-Throw-prop: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Exceptional None Enter);$
 $instrs\text{-of } (PROG P) C M ! pc = Throw;$
 $ek = \uparrow(\lambda s. s(Exception \mapsto Value (stkAt s (stkLength (P, C, M) pc - 1)))) \rfloor$
 $\Rightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Exceptional None Enter) \text{--}(ek) \rightarrow (C, M, None, Return)$
 $| CFG\text{-Throw-handle: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Exceptional \lfloor pc' \rfloor Enter);$

$pc' \neq \text{length} (\text{instrs-of } (PROG P) C M);$
 $\text{instrs-of } (PROG P) C M ! pc = \text{Throw};$
 $ek = \uparrow(\lambda s. (s(\text{Exception} := \text{None})))$
 $(\text{Stack } (\text{stkLength } (P, C, M) pc' - 1) \mapsto \text{Value } (\text{stkAt } s (\text{stkLength}$
 $(P, C, M) pc - 1)))) \mathbb{I}$
 $\implies (P, C0, \text{Main}) \vdash (C, M, [pc], \text{Exceptional } [pc'] \text{ Enter}) \text{--}(ek) \rightarrow (C, M,$
 $[pc'], \text{Enter})$
 $| \text{CFG-Invoke-Check-NP-Normal: } \mathbb{I} C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \implies (C,$
 $M, [pc], \text{Enter});$
 $\text{instrs-of } (PROG P) C M ! pc = \text{Invoke } M' n;$
 $ek = (\lambda s. \text{stkAt } s (\text{stkLength } (P, C, M) pc - \text{Suc } n) \neq \text{Null})_{\checkmark} \mathbb{I}$
 $\implies (P, C0, \text{Main}) \vdash (C, M, [pc], \text{Enter}) \text{--}(ek) \rightarrow (C, M, [pc], \text{Normal})$
 $| \text{CFG-Invoke-Check-NP-Exceptional: } \mathbb{I} C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \implies (C,$
 $M, [pc], \text{Enter});$
 $\text{instrs-of } (PROG P) C M ! pc = \text{Invoke } M' n;$
 $pc' = (\text{case } (\text{match-ex-table } (PROG P) \text{NullPointer } pc (\text{ex-table-of } (PROG P)$
 $C M)) \text{ of}$
 $\quad \text{None} \implies \text{None}$
 $\quad | \text{Some } (pc'', d) \implies [pc'']);$
 $ek = (\lambda s. \text{stkAt } s (\text{stkLength } (P, C, M) pc - \text{Suc } n) = \text{Null})_{\checkmark} \mathbb{I}$
 $\implies (P, C0, \text{Main}) \vdash (C, M, [pc], \text{Enter}) \text{--}(ek) \rightarrow (C, M, [pc], \text{Exceptional } pc'$
 $\text{Enter})$
 $| \text{CFG-Invoke-NP-prop: } \mathbb{I} C \neq \text{ClassMain } P;$
 $(P, C0, \text{Main}) \vdash \implies (C, M, [pc], \text{Exceptional } \text{None } \text{Enter});$
 $\text{instrs-of } (PROG P) C M ! pc = \text{Invoke } M' n;$
 $ek = \uparrow(\lambda s. s(\text{Exception} \mapsto \text{Value } (\text{Addr } (\text{addr-of-sys-xcpt } \text{NullPointer})))) \mathbb{I}$
 $\implies (P, C0, \text{Main}) \vdash (C, M, [pc], \text{Exceptional } \text{None } \text{Enter}) \text{--}(ek) \rightarrow (C, M,$
 $\text{None, Return})$
 $| \text{CFG-Invoke-NP-handle: } \mathbb{I} C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \implies (C, M, [pc],$
 $\text{Exceptional } [pc'] \text{ Enter});$
 $\text{instrs-of } (PROG P) C M ! pc = \text{Invoke } M' n;$
 $ek = \uparrow(\lambda s. (s(\text{Exception} := \text{None})))$
 $(\text{Stack } (\text{stkLength } (P, C, M) pc' - 1) \mapsto \text{Value } (\text{Addr } (\text{addr-of-sys-xcpt}$
 $\text{NullPointer})))) \mathbb{I}$
 $\implies (P, C0, \text{Main}) \vdash (C, M, [pc], \text{Exceptional } [pc'] \text{ Enter}) \text{--}(ek) \rightarrow (C, M,$
 $[pc'], \text{Enter})$
 $| \text{CFG-Invoke-Call: } \mathbb{I} C \neq \text{ClassMain } P; (P, C0, \text{Main}) \vdash \implies (C, M, [pc], \text{Nor-}$
 $\text{mal});$
 $\text{instrs-of } (PROG P) C M ! pc = \text{Invoke } M' n;$
 $\text{TYPING } P C M ! pc = [(ST, LT)];$
 $ST ! n = \text{Class } D';$
 $PROG P \vdash D' \text{ sees } M': Ts \rightarrow T = (mxs, mxl_0, is, xt) \text{ in } D;$
 $Q = (\lambda(s, \text{ret}). \text{let } r = \text{stkAt } s (\text{stkLength } (P, C, M) pc - \text{Suc } n);$
 $\quad C' = \text{fst } (\text{the } (\text{heap-of } s (\text{the-Addr } r)))$
 $\quad \text{in } D = \text{fst } (\text{method } (PROG P) C' M'));$
 $\text{paramDefs} = (\lambda s. s \text{Heap})$
 $\quad \# (\lambda s. s (\text{Stack } (\text{stkLength } (P, C, M) pc - \text{Suc } n)))$
 $\quad \# (\text{rev } (\text{map } (\lambda i. (\lambda s. s (\text{Stack } (\text{stkLength } (P, C, M) pc - \text{Suc } i))))$
 $[0..<n]));$

$$\begin{array}{l}
ek = Q:(C, M, pc) \hookrightarrow_{(D, M')} paramDefs \\
\boxed{\begin{array}{l}
\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Normal) \text{---}(ek) \rightarrow (D, M', None, Enter) \\
| \text{CFG-Invoke-False: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Normal) \rrbracket; \\
instrs\text{-of } (PROG P) C M ! pc = Invoke M' n; \\
ek = (\lambda s. False)_{\surd}
\end{array}} \\
\boxed{\begin{array}{l}
\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Normal) \text{---}(ek) \rightarrow (C, M, \lfloor pc \rfloor, Return) \\
| \text{CFG-Invoke-Return-Check-Normal: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Return) \rrbracket; \\
instrs\text{-of } (PROG P) C M ! pc = Invoke M' n; \\
(TYPING P) C M ! pc = \lfloor (ST, LT) \rfloor; \\
ST ! n \neq NT; \\
ek = (\lambda s. s \text{Exception} = None)_{\surd}
\end{array}} \\
\boxed{\begin{array}{l}
\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Return) \text{---}(ek) \rightarrow (C, M, \lfloor Suc pc \rfloor, Enter) \\
| \text{CFG-Invoke-Return-Check-Exceptional: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Return) \rrbracket; \\
instrs\text{-of } (PROG P) C M ! pc = Invoke M' n; \\
match\text{-ex-table } (PROG P) Exc pc (ex\text{-table-of } (PROG P) C M) = \lfloor (pc', diff) \rfloor; \\
pc' \neq length (instrs\text{-of } (PROG P) C M); \\
ek = (\lambda s. \exists v d. s \text{Exception} = \lfloor v \rfloor \wedge \\
\quad match\text{-ex-table } (PROG P) (cname\text{-of } (heap\text{-of } s) (the\text{-Addr } (the\text{-Value } v))) pc (ex\text{-table-of } (PROG P) C M) = \lfloor (pc', d) \rfloor)_{\surd}
\end{array}} \\
\boxed{\begin{array}{l}
\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Return) \text{---}(ek) \rightarrow (C, M, \lfloor pc \rfloor, Exceptional \lfloor pc' \rfloor Return) \\
| \text{CFG-Invoke-Return-Exceptional-handle: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Exceptional \lfloor pc' \rfloor Return) \rrbracket; \\
instrs\text{-of } (PROG P) C M ! pc = Invoke M' n; \\
ek = \uparrow (\lambda s. s(\text{Exception} := None, \\
\quad Stack (stkLength (P, C, M) pc' - 1) := s \text{Exception})) \rrbracket \\
\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Exceptional \lfloor pc' \rfloor Return) \text{---}(ek) \rightarrow (C, M, \lfloor pc' \rfloor, Enter) \\
| \text{CFG-Invoke-Return-Exceptional-prop: } \llbracket C \neq ClassMain P; \\
(P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Return) \rrbracket; \\
instrs\text{-of } (PROG P) C M ! pc = Invoke M' n; \\
ek = (\lambda s. \exists v. s \text{Exception} = \lfloor v \rfloor \wedge \\
\quad match\text{-ex-table } (PROG P) (cname\text{-of } (heap\text{-of } s) (the\text{-Addr } (the\text{-Value } v))) pc (ex\text{-table-of } (PROG P) C M) = None)_{\surd} \rrbracket \\
\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Return) \text{---}(ek) \rightarrow (C, M, None, Return) \\
| \text{CFG-Return: } \llbracket C \neq ClassMain P; (P, C0, Main) \vdash \Rightarrow (C, M, \lfloor pc \rfloor, Enter) \rrbracket; \\
instrs\text{-of } (PROG P) C M ! pc = instr.Return; \\
ek = \uparrow (\lambda s. s(Stack 0 := s (Stack (stkLength (P, C, M) pc - 1)))) \rrbracket \\
\boxed{\begin{array}{l}
\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter) \text{---}(ek) \rightarrow (C, M, None, Return) \\
| \text{CFG-Return-from-Method: } \llbracket (P, C0, Main) \vdash \Rightarrow (C, M, None, Return); \\
(P, C0, Main) \vdash (C', M', \lfloor pc' \rfloor, Normal) \text{---}(Q':(C', M', pc') \hookrightarrow_{(C, M)} ps) \rightarrow (C,
\end{array}}
\end{array}$$

$M, \text{None}, \text{Enter};$
 $Q = (\lambda(s, \text{ret}). \text{ret} = (C', M', \text{pc}'));$
 $\text{stateUpdate} = (\lambda s s'. s'(\text{Heap} := s \text{Heap},$
 $\quad \text{Exception} := s \text{Exception},$
 $\quad \text{Stack} (\text{stkLength} (P, C', M') (\text{Suc } \text{pc}') - 1) := s (\text{Stack } 0))$
 $);$
 $ek = Q \leftrightarrow (C, M) \text{stateUpdate}$
 \mathbb{I}
 $\implies (P, C0, \text{Main}) \vdash (C, M, \text{None}, \text{Return}) \text{---}(ek) \rightarrow (C', M', \lfloor \text{pc}' \rfloor, \text{Return})$

lemma *JVMCFG-edge-det*: $\mathbb{I} P \vdash n \text{---}(et) \rightarrow n'; P \vdash n \text{---}(et') \rightarrow n' \mathbb{I} \implies et = et'$
 $\langle \text{proof} \rangle$

lemma *sourcenode-reachable*: $P \vdash n \text{---}(ek) \rightarrow n' \implies P \vdash \Rightarrow n$
 $\langle \text{proof} \rangle$

lemma *targetnode-reachable*:
assumes *edge*: $P \vdash n \text{---}(ek) \rightarrow n'$
shows $P \vdash \Rightarrow n'$
 $\langle \text{proof} \rangle$

lemmas *JVMCFG-reachable-inducts* = *JVMCFG-reachable.inducts*[*split-format* (*complete*)]

lemma *ClassMain-imp-MethodMain*:
 $(P, C0, \text{Main}) \vdash (C', M', \text{pc}', \text{nt}') \text{---}ek \rightarrow (\text{ClassMain } P, M, \text{pc}, \text{nt}) \implies M =$
 $\text{MethodMain } P$
 $(P, C0, \text{Main}) \vdash \Rightarrow (\text{ClassMain } P, M, \text{pc}, \text{nt}) \implies M = \text{MethodMain } P$
 $\langle \text{proof} \rangle$

lemma *ClassMain-no-Call-target* [*dest!*]:
 $(P, C0, \text{Main}) \vdash (C, M, \text{pc}, \text{nt}) \text{---}Q:(C', M', \text{pc}') \leftrightarrow_{(D, M'')} \text{paramDefs} \rightarrow (\text{ClassMain}$
 $P, M''', \text{pc}'', \text{nt}'')$
 $\implies \text{False}$
and
 $(P, C0, \text{Main}) \vdash \Rightarrow (C, M, \text{pc}, \text{nt}) \implies \text{True}$
 $\langle \text{proof} \rangle$

lemma *method-of-src-and-trg-exists*:
 $\mathbb{I} (P, C0, \text{Main}) \vdash (C', M', \text{pc}', \text{nt}') \text{---}ek \rightarrow (C, M, \text{pc}, \text{nt}); C \neq \text{ClassMain } P;$
 $C' \neq \text{ClassMain } P \mathbb{I}$
 $\implies (\exists Ts T mb. (\text{PROG } P) \vdash C \text{ sees } M:Ts \rightarrow T = mb \text{ in } C) \wedge$
 $(\exists Ts T mb. (\text{PROG } P) \vdash C' \text{ sees } M':Ts \rightarrow T = mb \text{ in } C')$
and *method-of-reachable-node-exists*:
 $\mathbb{I} (P, C0, \text{Main}) \vdash \Rightarrow (C, M, \text{pc}, \text{nt}); C \neq \text{ClassMain } P \mathbb{I}$
 $\implies \exists Ts T mb. (\text{PROG } P) \vdash C \text{ sees } M:Ts \rightarrow T = mb \text{ in } C$
 $\langle \text{proof} \rangle$

lemma $\llbracket (P, C0, Main) \vdash (C', M', pc', nt') -ek \rightarrow (C, M, pc, nt); C \neq \text{ClassMain } P \rrbracket$
 $P; C' \neq \text{ClassMain } P \rrbracket$
 $\Rightarrow (\text{case } pc \text{ of } \text{None} \Rightarrow \text{True} \mid$
 $\quad \llbracket pc'' \rrbracket \Rightarrow (\text{TYPING } P) C M ! pc'' \neq \text{None} \wedge pc'' < \text{length } (\text{instrs-of } (\text{PROG } P) C M)) \wedge$
 $(\text{case } pc' \text{ of } \text{None} \Rightarrow \text{True} \mid$
 $\quad \llbracket pc'' \rrbracket \Rightarrow (\text{TYPING } P) C' M' ! pc'' \neq \text{None} \wedge pc'' < \text{length } (\text{instrs-of } (\text{PROG } P) C' M'))$
and *instr-of-reachable-node-typable*: $\llbracket (P, C0, Main) \vdash \Rightarrow (C, M, pc, nt); C \neq \text{ClassMain } P \rrbracket$
 $\Rightarrow \text{case } pc \text{ of } \text{None} \Rightarrow \text{True} \mid$
 $\quad \llbracket pc'' \rrbracket \Rightarrow (\text{TYPING } P) C M ! pc'' \neq \text{None} \wedge pc'' < \text{length } (\text{instrs-of } (\text{PROG } P) C M)$
 $\langle \text{proof} \rangle$

lemma *reachable-node-impl-wt-instr*:
assumes $(P, C0, Main) \vdash \Rightarrow (C, M, \llbracket pc \rrbracket, nt)$
and $C \neq \text{ClassMain } P$
shows $\exists T \text{ mxs mpc xt. } \text{PROG } P, T, \text{mxs}, \text{mpc}, \text{xt} \vdash (\text{instrs-of } (\text{PROG } P) C M ! pc), pc :: \text{TYPING } P C M$
 $\langle \text{proof} \rangle$

lemma
 $\llbracket (P, C0, Main) \vdash (C, M, pc, nt) -ek \rightarrow (C', M', pc', nt'); C \neq \text{ClassMain } P \vee C' \neq \text{ClassMain } P \rrbracket$
 $\Rightarrow \exists T \text{ mb } D. \text{PROG } P \vdash C0 \text{ sees } \text{Main}: \llbracket \rightarrow T = \text{mb in } D$
and *reachable-node-impl-Main-ex*:
 $\llbracket (P, C0, Main) \vdash \Rightarrow (C, M, pc, nt); C \neq \text{ClassMain } P \rrbracket$
 $\Rightarrow \exists T \text{ mb } D. \text{PROG } P \vdash C0 \text{ sees } \text{Main}: \llbracket \rightarrow T = \text{mb in } D$
 $\langle \text{proof} \rangle$

end

theory *JVMInterpretation* **imports** *JVMCFG ../StaticInter/CFGExit* **begin**

3.2 Instatiation of the *CFG* locale

abbreviation *sourcenode* $:: \text{cfg-edge} \Rightarrow \text{cfg-node}$
where *sourcenode* $e \equiv \text{fst } e$

abbreviation *targetnode* $:: \text{cfg-edge} \Rightarrow \text{cfg-node}$
where *targetnode* $e \equiv \text{snd}(\text{snd } e)$

abbreviation *kind* $:: \text{cfg-edge} \Rightarrow (\text{var}, \text{val}, \text{cname} \times \text{mname} \times \text{pc}, \text{cname} \times \text{mname}) \text{ edge-kind}$
where *kind* $e \equiv \text{fst}(\text{snd } e)$

definition *valid-edge* $:: \text{jvm-method} \Rightarrow \text{cfg-edge} \Rightarrow \text{bool}$
where *valid-edge* $P e \equiv P \vdash (\text{sourcenode } e) -(\text{kind } e) \rightarrow (\text{targetnode } e)$

fun *methods* :: *cname* \Rightarrow *JVMInstructions.jvm-method mdecl list* \Rightarrow ((*cname* \times *mname*) \times *var list* \times *var list*) *list*
where *methods* *C* [] = []
| *methods* *C* ((*M*, *Ts*, *T*, *mb*) # *ms*)
= ((*C*, *M*), *Heap* # (map *Local* [0..*Suc* (length *Ts*)]), [*Heap*, *Stack 0*, *Exception*])
(*methods* *C* *ms*)

fun *procs* :: *jvm-prog* \Rightarrow ((*cname* \times *mname*) \times *var list* \times *var list*) *list*
where *procs* [] = []
| *procs* ((*C*, *D*, *fs*, *ms*) # *P*) = (*methods* *C* *ms*) @ (*procs* *P*)

lemma *in-set-methodsI*: *map-of ms M* = [(*Ts*, *T*, *mxs*, *mxl0*, *is*, *xt*)]
 \Rightarrow ((*C'*, *M*), *Heap* # map *Local* [0..*length Ts*] @ [*Local* (length *Ts*)], [*Heap*,
Stack 0, *Exception*])
 \in set (*methods* *C'* *ms*)
<*proof*>

lemma *in-methods-in-msD*: ((*C*, *M*), *ins*, *outs*) \in set (*methods* *D* *ms*)
 \Rightarrow *M* \in set (map *fst* *ms*) \wedge *D* = *C*
<*proof*>

lemma *in-methods-in-msD'*: ((*C*, *M*), *ins*, *outs*) \in set (*methods* *D* *ms*)
 \Rightarrow \exists *Ts* *T* *mb*. (*M*, *Ts*, *T*, *mb*) \in set *ms*
 \wedge *D* = *C*
 \wedge *ins* = *Heap* # (map *Local* [0..*Suc* (length *Ts*)])
 \wedge *outs* = [*Heap*, *Stack 0*, *Exception*]
<*proof*>

lemma *in-set-methodsE*:
assumes ((*C*, *M*), *ins*, *outs*) \in set (*methods* *D* *ms*)
obtains *Ts* *T* *mb*
where (*M*, *Ts*, *T*, *mb*) \in set *ms*
and *D* = *C*
and *ins* = *Heap* # (map *Local* [0..*Suc* (length *Ts*)])
and *outs* = [*Heap*, *Stack 0*, *Exception*]
<*proof*>

lemma *in-set-procsI*:
assumes *sees*: *P* \vdash *D* *sees* *M*: *Ts* \rightarrow *T* = *mb* *in* *D*
and *ins-def*: *ins* = *Heap* # map *Local* [0..*Suc* (length *Ts*)]
and *outs-def*: *outs* = [*Heap*, *Stack 0*, *Exception*]
shows ((*D*, *M*), *ins*, *outs*) \in set (*procs* *P*)
<*proof*>

lemma *distinct-methods*: *distinct* (map *fst* *ms*) \Rightarrow *distinct* (map *fst* (*methods* *C* *ms*))
<*proof*>

lemma *in-set-procsD*:

$((C, M), ins, out) \in set (procs P) \implies \exists D fs ms. (C, D, fs, ms) \in set P \wedge M \in set (map fst ms)$
 $\langle proof \rangle$

lemma *in-set-procsE'*:

assumes $((C, M), ins, outs) \in set (procs P)$
obtains $D fs ms Ts T mb$
where $(C, D, fs, ms) \in set P$
and $(M, Ts, T, mb) \in set ms$
and $ins = Heap \# (map (\lambda n. Local n) [0..<Suc (length Ts)])$
and $outs = [Heap, Stack 0, Exception]$
 $\langle proof \rangle$

lemma *distinct-Local-vars [simp]*: $distinct (map Local [0..<n])$
 $\langle proof \rangle$

lemma *distinct-Stack-vars [simp]*: $distinct (map Stack [0..<n])$
 $\langle proof \rangle$

inductive-set *get-return-edges* :: $wf-jvmprog \implies cfg-edge \implies cfg-edge set$

for $P :: wf-jvmprog$
and $a :: cfg-edge$
where
 $kind a = Q:(C, M, pc) \leftrightarrow_{(D, M')} paramDefs$
 $\implies ((D, M', None, Return),$
 $(\lambda(s, ret). ret = (C, M, pc)) \leftrightarrow_{(D, M')} (\lambda s s'. s'(Heap := s Heap, Exception := s$
 $Exception,$
 $Stack (stkLength (P, C, M) (Suc pc) - 1)$
 $:= s (Stack 0))),$
 $(C, M, [pc], Return)) \in (get-return-edges P a)$

lemma *get-return-edgesE [elim!]*:

assumes $a \in get-return-edges P a'$
obtains $Q C M pc D M' paramDefs$ **where**
 $kind a' = Q:(C, M, pc) \leftrightarrow_{(D, M')} paramDefs$
and $a = ((D, M', None, Return),$
 $(\lambda(s, ret). ret = (C, M, pc)) \leftrightarrow_{(D, M')} (\lambda s s'. s'(Heap := s Heap, Exception := s$
 $Exception,$
 $Stack (stkLength (P, C, M) (Suc pc) - 1) := s (Stack 0))),$
 $(C, M, [pc], Return))$
 $\langle proof \rangle$

lemma *distinct-class-names*: $distinct-fst (PROG P)$
 $\langle proof \rangle$

lemma *distinct-method-names*:

$class (PROG P) C = [(D, fs, ms)] \implies distinct-fst ms$
 $\langle proof \rangle$

lemma *distinct-fst-is-distinct-fst*: *distinct-fst = BasicDefs.distinct-fst*
 ⟨proof⟩

lemma *ClassMain-not-in-set-PROG* [*dest!*]: $(\text{ClassMain } P, D, fs, ms) \in \text{set } (\text{PROG } P) \implies \text{False}$
 ⟨proof⟩

lemma *in-set-procsE*:
assumes $((C, M), ins, outs) \in \text{set } (\text{procs } (\text{PROG } P))$
obtains $D fs ms Ts T mb$
where $\text{class } (\text{PROG } P) C = [(D, fs, ms)]$
and $\text{PROG } P \vdash C \text{ sees } M:Ts \rightarrow T = mb \text{ in } C$
and $ins = \text{Heap} \# (\text{map } (\lambda n. \text{Local } n) [0..< \text{Suc } (\text{length } Ts)])$
and $outs = [\text{Heap}, \text{Stack } 0, \text{Exception}]$
 ⟨proof⟩

declare *has-method-def* [*simp*]

interpretation *JVMCFG-Interpret*:
CFG sourcenode targetnode kind valid-edge $(P, C0, \text{Main})$
 $(\text{ClassMain } P, \text{MethodMain } P, \text{None}, \text{Enter})$
 $(\lambda(C, M, pc, type). (C, M)) \text{ get-return-edges } P$
 $((\text{ClassMain } P, \text{MethodMain } P), [], []) \# \text{procs } (\text{PROG } P) (\text{ClassMain } P, \text{MethodMain } P)$
for $P C0 \text{Main}$
 ⟨proof⟩

interpretation *JVMCFG-Exit-Interpret*:
CFGExit sourcenode targetnode kind valid-edge $(P, C0, \text{Main})$
 $(\text{ClassMain } P, \text{MethodMain } P, \text{None}, \text{Enter})$
 $(\lambda(C, M, pc, type). (C, M)) \text{ get-return-edges } P$
 $((\text{ClassMain } P, \text{MethodMain } P), [], []) \# \text{procs } (\text{PROG } P)$
 $(\text{ClassMain } P, \text{MethodMain } P) (\text{ClassMain } P, \text{MethodMain } P, \text{None}, \text{Return})$
for $P C0 \text{Main}$
 ⟨proof⟩

end

theory *JVMCFG-wf* **imports** *JVMInterpretation ../StaticInter/CFGExit-wf* **begin**

inductive-set *Def* :: *wf-jvmprog* \Rightarrow *cfg-node* \Rightarrow *var set*

for $P :: \text{wf-jvmprog}$
and $n :: \text{cfg-node}$

where

Def-Main-Heap:
 $n = (\text{ClassMain } P, \text{MethodMain } P, [0], \text{Return})$
 $\implies \text{Heap} \in \text{Def } P n$

| *Def-Main-Exception*:

$n = (\text{ClassMain } P, \text{MethodMain } P, [0], \text{Return})$

$\implies \text{Exception} \in \text{Def } P \ n$
| *Def-Main-Stack-0*:
 $n = (\text{ClassMain } P, \text{MethodMain } P, [0], \text{Return})$
 $\implies \text{Stack } 0 \in \text{Def } P \ n$
| *Def-Load*:
 $\llbracket n = (C, M, [pc], \text{Enter});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc = \text{Load } \text{idx};$
 $i = \text{stkLength } (P, C, M) \ pc \rrbracket$
 $\implies \text{Stack } i \in \text{Def } P \ n$
| *Def-Store*:
 $\llbracket n = (C, M, [pc], \text{Enter});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc = \text{Store } \text{idx} \rrbracket$
 $\implies \text{Local } \text{idx} \in \text{Def } P \ n$
| *Def-Push*:
 $\llbracket n = (C, M, [pc], \text{Enter});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc = \text{Push } v;$
 $i = \text{stkLength } (P, C, M) \ pc \rrbracket$
 $\implies \text{Stack } i \in \text{Def } P \ n$
| *Def-IAdd*:
 $\llbracket n = (C, M, [pc], \text{Enter});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc = \text{IAdd};$
 $i = \text{stkLength } (P, C, M) \ pc - 2 \rrbracket$
 $\implies \text{Stack } i \in \text{Def } P \ n$
| *Def-CmpEq*:
 $\llbracket n = (C, M, [pc], \text{Enter});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc = \text{CmpEq};$
 $i = \text{stkLength } (P, C, M) \ pc - 2 \rrbracket$
 $\implies \text{Stack } i \in \text{Def } P \ n$
| *Def-New-Heap*:
 $\llbracket n = (C, M, [pc], \text{Normal});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc = \text{New } Cl \rrbracket$
 $\implies \text{Heap} \in \text{Def } P \ n$
| *Def-New-Stack*:
 $\llbracket n = (C, M, [pc], \text{Normal});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc = \text{New } Cl;$
 $i = \text{stkLength } (P, C, M) \ pc \rrbracket$
 $\implies \text{Stack } i \in \text{Def } P \ n$
| *Def-Exception*:
 $\llbracket n = (C, M, [pc], \text{Exceptional } pco \ nt);$
 $C \neq \text{ClassMain } P \rrbracket$
 $\implies \text{Exception} \in \text{Def } P \ n$
| *Def-Exception-handle*:

$\llbracket n = (C, M, \lfloor pc \rfloor, \text{Exceptional } \lfloor pc' \rfloor \text{ Enter});$
 $C \neq \text{ClassMain } P;$
 $i = \text{stkLength } (P, C, M) \text{ } pc' - 1 \rrbracket$
 $\implies \text{Stack } i \in \text{Def } P \ n$

| *Def-Exception-handle-return:*
 $\llbracket n = (C, M, \lfloor pc \rfloor, \text{Exceptional } \lfloor pc' \rfloor \text{ Return});$
 $C \neq \text{ClassMain } P;$
 $i = \text{stkLength } (P, C, M) \text{ } pc' - 1 \rrbracket$
 $\implies \text{Stack } i \in \text{Def } P \ n$

| *Def-Getfield:*
 $\llbracket n = (C, M, \lfloor pc \rfloor, \text{Normal});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (PROG \ P) \ C \ M \ ! \ pc = \text{Getfield } Cl \ Fd;$
 $i = \text{stkLength } (P, C, M) \text{ } pc - 1 \rrbracket$
 $\implies \text{Stack } i \in \text{Def } P \ n$

| *Def-Putfield:*
 $\llbracket n = (C, M, \lfloor pc \rfloor, \text{Normal});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (PROG \ P) \ C \ M \ ! \ pc = \text{Putfield } Cl \ Fd \rrbracket$
 $\implies \text{Heap} \in \text{Def } P \ n$

| *Def-Invoke-Return-Heap:*
 $\llbracket n = (C, M, \lfloor pc \rfloor, \text{Return});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (PROG \ P) \ C \ M \ ! \ pc = \text{Invoke } M' \ n' \rrbracket$
 $\implies \text{Heap} \in \text{Def } P \ n$

| *Def-Invoke-Return-Exception:*
 $\llbracket n = (C, M, \lfloor pc \rfloor, \text{Return});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (PROG \ P) \ C \ M \ ! \ pc = \text{Invoke } M' \ n' \rrbracket$
 $\implies \text{Exception} \in \text{Def } P \ n$

| *Def-Invoke-Return-Stack:*
 $\llbracket n = (C, M, \lfloor pc \rfloor, \text{Return});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (PROG \ P) \ C \ M \ ! \ pc = \text{Invoke } M' \ n';$
 $i = \text{stkLength } (P, C, M) \text{ } (\text{Suc } pc) - 1 \rrbracket$
 $\implies \text{Stack } i \in \text{Def } P \ n$

| *Def-Invoke-Call-Heap:*
 $\llbracket n = (C, M, \text{None}, \text{Enter});$
 $C \neq \text{ClassMain } P \rrbracket$
 $\implies \text{Heap} \in \text{Def } P \ n$

| *Def-Invoke-Call-Local:*
 $\llbracket n = (C, M, \text{None}, \text{Enter});$
 $C \neq \text{ClassMain } P;$
 $i < \text{locLength } (P, C, M) \ 0 \rrbracket$
 $\implies \text{Local } i \in \text{Def } P \ n$

| *Def-Return:*
 $\llbracket n = (C, M, \lfloor pc \rfloor, \text{Enter});$
 $C \neq \text{ClassMain } P;$
 $\text{instrs-of } (PROG \ P) \ C \ M \ ! \ pc = \text{instr.Return} \rrbracket$

$\implies \text{Stack } 0 \in \text{Def } P \ n$

inductive-set $\text{Use} :: \text{wf-jvmprog} \Rightarrow \text{cfg-node} \Rightarrow \text{var set}$

for $P :: \text{wf-jvmprog}$

and $n :: \text{cfg-node}$

where

Use-Main-Heap:

$n = (\text{ClassMain } P, \text{MethodMain } P, [0], \text{Normal})$

$\implies \text{Heap} \in \text{Use } P \ n$

| *Use-Load:*

$\llbracket n = (C, M, [pc], \text{Enter});$

$C \neq \text{ClassMain } P;$

$\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc = \text{Load } \text{idx} \rrbracket$

$\implies \text{Local } \text{idx} \in \text{Use } P \ n$

| *Use-Enter-Stack:*

$\llbracket n = (C, M, [pc], \text{Enter});$

$C \neq \text{ClassMain } P;$

$\text{case } (\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc)$

$\text{of } \text{Store } n' \Rightarrow d = 1$

| $\text{Getfield } F \ Cl \Rightarrow d = 1$

| $\text{Putfield } F \ Cl \Rightarrow d = 2$

| $\text{Checkcast } Cl \Rightarrow d = 1$

| $\text{Invoke } M' \ n' \Rightarrow d = \text{Suc } n'$

| $\text{IAdd} \Rightarrow d \in \{1, 2\}$

| $\text{IfFalse } i \Rightarrow d = 1$

| $\text{CmpEq} \Rightarrow d \in \{1, 2\}$

| $\text{Throw} \Rightarrow d = 1$

| $\text{instr.Return} \Rightarrow d = 1$

| $- \Rightarrow \text{False};$

$i = \text{stkLength } (P, C, M) \ pc - d \rrbracket$

$\implies \text{Stack } i \in \text{Use } P \ n$

| *Use-Enter-Local:*

$\llbracket n = (C, M, [pc], \text{Enter});$

$C \neq \text{ClassMain } P;$

$\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc = \text{Load } n' \rrbracket$

$\implies \text{Local } n' \in \text{Use } P \ n$

| *Use-Enter-Heap:*

$\llbracket n = (C, M, [pc], \text{Enter});$

$C \neq \text{ClassMain } P;$

$\text{case } (\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc)$

$\text{of } \text{New } Cl \Rightarrow \text{True}$

| $\text{Checkcast } Cl \Rightarrow \text{True}$

| $\text{Throw} \Rightarrow \text{True}$

| $- \Rightarrow \text{False} \rrbracket$

$\implies \text{Heap} \in \text{Use } P \ n$

| *Use-Normal-Heap:*

$\llbracket n = (C, M, [pc], \text{Normal});$

$C \neq \text{ClassMain } P;$

$\text{case } (\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc)$

```

    of New Cl  $\Rightarrow$  True
  | Getfield F Cl  $\Rightarrow$  True
  | Putfield F Cl  $\Rightarrow$  True
  | Invoke M' n'  $\Rightarrow$  True
  | -  $\Rightarrow$  False ]
 $\Rightarrow$  Heap  $\in$  Use P n
| Use-Normal-Stack:
[[ n = (C, M, [pc], Normal);
C  $\neq$  ClassMain P;
case (instrs-of (PROG P) C M ! pc)
  of Getfield F Cl  $\Rightarrow$  d = 1
  | Putfield F Cl  $\Rightarrow$  d  $\in$  {1, 2}
  | Invoke M' n'  $\Rightarrow$  d > 0  $\wedge$  d  $\leq$  Suc n'
  | -  $\Rightarrow$  False;
i = stkLength (P, C, M) pc - d ]
 $\Rightarrow$  Stack i  $\in$  Use P n
| Use-Return-Heap:
[[ n = (C, M, [pc], Return);
instrs-of (PROG P) C M ! pc = Invoke M' n'  $\vee$  C = ClassMain P ]
 $\Rightarrow$  Heap  $\in$  Use P n
| Use-Return-Stack:
[[ n = (C, M, [pc], Return);
(instrs-of (PROG P) C M ! pc = Invoke M' n'  $\wedge$  i = stkLength (P, C, M) (Suc
pc) - 1)  $\vee$ 
(C = ClassMain P  $\wedge$  i = 0) ]
 $\Rightarrow$  Stack i  $\in$  Use P n
| Use-Return-Exception:
[[ n = (C, M, [pc], Return);
instrs-of (PROG P) C M ! pc = Invoke M' n'  $\vee$  C = ClassMain P ]
 $\Rightarrow$  Exception  $\in$  Use P n
| Use-Exceptional-Stack:
[[ n = (C, M, [pc], Exceptional opc' nt);
case (instrs-of (PROG P) C M ! pc)
  of Throw  $\Rightarrow$  True
  | -  $\Rightarrow$  False;
i = stkLength (P, C, M) pc - 1 ]
 $\Rightarrow$  Stack i  $\in$  Use P n
| Use-Exceptional-Exception:
[[ n = (C, M, [pc], Exceptional [pc'] Return);
instrs-of (PROG P) C M ! pc = Invoke M' n' ]
 $\Rightarrow$  Exception  $\in$  Use P n
| Use-Method-Leave-Exception:
[[ n = (C, M, None, Return);
C  $\neq$  ClassMain P ]
 $\Rightarrow$  Exception  $\in$  Use P n
| Use-Method-Leave-Heap:
[[ n = (C, M, None, Return);
C  $\neq$  ClassMain P ]
 $\Rightarrow$  Heap  $\in$  Use P n

```

| *Use-Method-Leave-Stack*:
 $\llbracket n = (C, M, \text{None}, \text{Return});$
 $C \neq \text{ClassMain } P \rrbracket$
 $\implies \text{Stack } 0 \in \text{Use } P \ n$

| *Use-Method-Entry-Heap*:
 $\llbracket n = (C, M, \text{None}, \text{Enter});$
 $C \neq \text{ClassMain } P \rrbracket$
 $\implies \text{Heap} \in \text{Use } P \ n$

| *Use-Method-Entry-Local*:
 $\llbracket n = (C, M, \text{None}, \text{Enter});$
 $C \neq \text{ClassMain } P;$
 $i < \text{locLength } (P, C, M) \ 0 \rrbracket$
 $\implies \text{Local } i \in \text{Use } P \ n$

fun *ParamDefs* :: *wf-jvmprog* \Rightarrow *cfg-node* \Rightarrow *var list*
where
 $\text{ParamDefs } P \ (C, M, \lfloor pc \rfloor, \text{Return}) = [\text{Heap}, \text{Stack } (\text{stkLength } (P, C, M) (\text{Suc } pc) - 1), \text{Exception}]$
 $\mid \text{ParamDefs } P \ (C, M, \text{opc}, nt) = []$

function *ParamUses* :: *wf-jvmprog* \Rightarrow *cfg-node* \Rightarrow *var set list*
where
 $\text{ParamUses } P \ (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Normal}) = [\{\text{Heap}\}, \{\}]$
 \mid
 $M \neq \text{MethodMain } P \vee \text{opc} \neq \lfloor 0 \rfloor \vee nt \neq \text{Normal}$
 $\implies \text{ParamUses } P \ (\text{ClassMain } P, M, \text{opc}, nt) = []$
 \mid
 $C \neq \text{ClassMain } P$
 $\implies \text{ParamUses } P \ (C, M, \text{opc}, nt) = (\text{case } \text{opc} \text{ of } \text{None} \Rightarrow []$
 $\mid \lfloor pc \rfloor \Rightarrow (\text{case } nt \text{ of } \text{Normal} \Rightarrow (\text{case } (\text{instrs-of } (\text{PROG } P) \ C \ M \ ! \ pc) \ \text{of}$
 $\text{Invoke } M' \ n \Rightarrow ($
 $\{\text{Heap}\} \ \# \ \text{rev } (\text{map } (\lambda n. \{\text{Stack } (\text{stkLength } (P, C, M) \ pc - (\text{Suc } n)\}))$
 $[0..<n + 1])$
 $)$
 $\mid - \Rightarrow [])$
 $\mid - \Rightarrow []$
 $)$
 $)$
 $\langle \text{proof} \rangle$
termination $\langle \text{proof} \rangle$

lemma *in-set-ParamDefsE*:
 $\llbracket V \in \text{set } (\text{ParamDefs } P \ n);$
 $\bigwedge C \ M \ pc. \llbracket n = (C, M, \lfloor pc \rfloor, \text{Return});$
 $V \in \{\text{Heap}, \text{Stack } (\text{stkLength } (P, C, M) (\text{Suc } pc) - 1), \text{Exception}\} \rrbracket \implies$
thesis \llbracket
 $\implies \text{thesis}$
 $\langle \text{proof} \rangle$

lemma *in-set-ParamUsesE*:

assumes *V-in-ParamUses*: $V \in \bigcup (\text{set } (\text{ParamUses } P \ n))$
obtains $n = (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Normal})$ **and** $V = \text{Heap}$
 $| C \ M \ pc \ M' \ n' \ i$ **where** $n = (C, M, \lfloor pc \rfloor, \text{Normal})$ **and** *instrs-of* $(\text{PROG } P) \ C$
 $M \ ! \ pc = \text{Invoke } M' \ n'$
and $V = \text{Heap} \vee V = \text{Stack } (\text{stkLength } (P, C, M) \ pc - \text{Suc } i)$ **and** $i < \text{Suc } n'$ **and** $C \neq \text{ClassMain } P$
 $\langle \text{proof} \rangle$

lemma *sees-method-fun-wf*:

assumes $\text{PROG } P \vdash D \text{ sees } M'$: $Ts \rightarrow T = (mxs, mxl_0, is, xt)$ *in* D
and $(D, D', fs, ms) \in \text{set } (\text{PROG } P)$
and $(M', Ts', T', mxs', mxl_0', is', xt') \in \text{set } ms$
shows $Ts = Ts' \wedge T = T' \wedge mxs = mxs' \wedge mxl_0 = mxl_0' \wedge is = is' \wedge xt = xt'$
 $\langle \text{proof} \rangle$

interpretation *JVMCFG-wf*:

CFG-wf sourcenode targetnode kind valid-edge $(P, C0, \text{Main})$
 $(\text{ClassMain } P, \text{MethodMain } P, \text{None}, \text{Enter})$
 $(\lambda(C, M, pc, type). (C, M)) \text{ get-return-edges } P$
 $((\text{ClassMain } P, \text{MethodMain } P), [], []) \# \text{procs } (\text{PROG } P)$
 $(\text{ClassMain } P, \text{MethodMain } P)$
 $\text{Def } P \ \text{Use } P \ \text{ParamDefs } P \ \text{ParamUses } P$
for $P \ C0 \ \text{Main}$
 $\langle \text{proof} \rangle$

interpretation *JVMCFGExit-wf* :

CFGExit-wf sourcenode targetnode kind valid-edge $(P, C0, \text{Main})$
 $(\text{ClassMain } P, \text{MethodMain } P, \text{None}, \text{Enter})$
 $(\lambda(C, M, pc, type). (C, M)) \text{ get-return-edges } P$
 $((\text{ClassMain } P, \text{MethodMain } P), [], []) \# \text{procs } (\text{PROG } P)$
 $(\text{ClassMain } P, \text{MethodMain } P)$
 $(\text{ClassMain } P, \text{MethodMain } P, \text{None}, \text{Return})$
 $\text{Def } P \ \text{Use } P \ \text{ParamDefs } P \ \text{ParamUses } P$
 $\langle \text{proof} \rangle$

end

theory *JVMPostdomination* **imports** *JVMInterpretation* *../StaticInter/Postdomination*
begin

context *CFG* **begin**

lemma *vp-snocI*:

$\llbracket n -as \rightarrow \sqrt{*} \ n'; \ n' -[a] \rightarrow \sqrt{*} \ n''; \ \forall Q \ p \ \text{ret} \ fs. \ \text{kind } a \neq Q \leftrightarrow_p \text{ret} \ \rrbracket \implies n -as \ @$
 $[a] \rightarrow \sqrt{*} \ n''$
 $\langle \text{proof} \rangle$

lemma *valid-node-cases'* [*case-names Source Target, consumes 1*]:

$\llbracket \text{valid-node } n; \ \wedge e. \ \llbracket \text{valid-edge } e; \ \text{sourcenode } e = n \ \rrbracket \implies \text{thesis}$

$\bigwedge e. \llbracket \text{valid-edge } e; \text{targetnode } e = n \rrbracket \implies \text{thesis} \rrbracket$
 $\implies \text{thesis}$
 <proof>

end

lemma *disjE-strong*: $\llbracket P \vee Q; P \implies R; \llbracket Q; \neg P \rrbracket \implies R \rrbracket \implies R$
 <proof>

lemmas *path-intros* [*intro*] = *JVMCFG-Interpret.path.Cons-path JVMCFG-Interpret.path.empty-path*
declare *JVMCFG-Interpret.vp-snocI* [*intro*]
declare *JVMCFG-Interpret.valid-node-def* [*simp add*]
valid-edge-def [*simp add*]
JVMCFG-Interpret.intra-path-def [*simp add*]

abbreviation *vp-snoc* :: *wf-jvmprog* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *cfg-edge list* \Rightarrow *cfg-node*
 \Rightarrow (*var*, *val*, *cname* \times *mname* \times *pc*, *cname* \times *mname*) *edge-kind* \Rightarrow *cfg-node* \Rightarrow
bool

where *vp-snoc* *P C0 Main* *as n ek n'*
 \equiv *JVMCFG-Interpret.valid-path' P C0 Main*
 (*ClassMain P*, *MethodMain P*, *None*, *Enter*) (*as* @ [(*n,ek,n'*)]) *n'*

lemma

$(P, C0, Main) \vdash (C, M, pc, nt) \text{--}ek \rightarrow (C', M', pc', nt')$
 $\implies (\exists \text{as. } CFG.\text{valid-path}' \text{ sourcenode targetnode kind } (valid\text{-edge } (P, C0, Main))$
 (*get-return-edges* *P*) (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, *pc*,
nt)) \wedge
 ($\exists \text{as. } CFG.\text{valid-path}' \text{ sourcenode targetnode kind } (valid\text{-edge } (P, C0, Main))$
 (*get-return-edges* *P*) (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C'*, *M'*,
pc', *nt'*))
and *valid-Entry-path*: $(P, C0, Main) \vdash \Rightarrow (C, M, pc, nt)$
 $\implies \exists \text{as. } CFG.\text{valid-path}' \text{ sourcenode targetnode kind } (valid\text{-edge } (P, C0, Main))$
 (*get-return-edges* *P*) (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, *pc*,
nt)
 <proof>

declare *JVMCFG-Interpret.vp-snocI* []
declare *JVMCFG-Interpret.valid-node-def* [*simp del*]
valid-edge-def [*simp del*]
JVMCFG-Interpret.intra-path-def [*simp del*]

definition *EP* :: *jvm-prog*

where *EP* = ("*C''*", *Object*, [],
 [("*M''*", [], *Void*, *1::nat*, *0::nat*, [*Push Unit*, *instr.Return*], [])] # *SystemClasses*

definition *Phi-EP* :: *typ*

where *Phi-EP C M* = (*if C = "C''* \wedge *M = "M''*
 then $\llbracket ([], [OK (Class "C'')]) \rrbracket, \llbracket [Void], [OK (Class "C'')] \rrbracket$ else []

lemma *distinct-classes''*:

"C'' ≠ Object
"C'' ≠ NullPointer
"C'' ≠ OutOfMemory
"C'' ≠ ClassCast
⟨proof⟩

lemmas *distinct-classes =*

distinct-classes distinct-classes'' distinct-classes'' [symmetric]

declare *distinct-classes* [simp add]

lemma *i-max-2D*: $i < \text{Suc } (\text{Suc } 0) \implies i = 0 \vee i = 1$ ⟨proof⟩

lemma *EP-wf*: *wf-jvm-prog* *Phi-EP* *EP*

⟨proof⟩

lemma [simp]: *PROG* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)) = *EP*

⟨proof⟩

lemma [simp]: *TYPING* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)) = *Phi-EP*

⟨proof⟩

lemma *method-in-EP-is-M*:

EP ⊢ *C* sees *M*: $Ts \rightarrow T = (m\text{xs}, m\text{x}l, \text{is}, \text{xt})$ in *D*
 $\implies C = \text{"C''} \wedge M = \text{"M''} \wedge Ts = [] \wedge T = \text{Void} \wedge m\text{xs} = 1 \wedge m\text{x}l = 0 \wedge$
 $\text{is} = [\text{Push Unit}, \text{instr.Return}] \wedge \text{xt} = [] \wedge D = \text{"C''}$
⟨proof⟩

lemma [simp]:

$\exists T Ts m\text{xs} m\text{x}l \text{is}. (\exists \text{xt}. EP \vdash \text{"C''} \text{ sees } \text{"M''}: Ts \rightarrow T = (m\text{xs}, m\text{x}l, \text{is}, \text{xt}) \text{ in } \text{"C''}) \wedge \text{is} \neq []$
⟨proof⟩

lemma [simp]:

$\exists T Ts m\text{xs} m\text{x}l \text{is}. (\exists \text{xt}. EP \vdash \text{"C''} \text{ sees } \text{"M''}: Ts \rightarrow T = (m\text{xs}, m\text{x}l, \text{is}, \text{xt}) \text{ in } \text{"C''}) \wedge$
 $\text{Suc } 0 < \text{length is}$
⟨proof⟩

lemma *C-sees-M-in-EP* [simp]:

EP ⊢ *C''* sees *M''*: $[] \rightarrow \text{Void} = (\text{Suc } 0, 0, [\text{Push Unit}, \text{instr.Return}], [])$ in *C''*
⟨proof⟩

lemma *instrs-of-EP-C-M* [simp]:

instrs-of *EP* *C''* *M''* = $[\text{Push Unit}, \text{instr.Return}]$
⟨proof⟩

lemma *ClassMain-not-C* [simp]: $\text{ClassMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})) \neq \text{"C"}$

$\langle \text{proof} \rangle$

lemma *method-entry* [dest!]: $(\text{Abs-wf-jvmprog } (EP, \text{Phi-EP}), \text{"C"}, \text{"M"}) \vdash \Rightarrow (C, M, \text{None}, \text{Enter})$

$\Rightarrow (C = \text{ClassMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})) \wedge M = \text{MethodMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})))$

$\vee (C = \text{"C"} \wedge M = \text{"M"})$

$\langle \text{proof} \rangle$

lemma *valid-node-in-EP-D*:

assumes *vn*: $\text{JVMCFG-Interpret.valid-node } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})) \text{"C"} \text{"M"} n$

shows $n \in \{$

$(\text{ClassMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \text{MethodMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \text{None}, \text{Enter}),$

$(\text{ClassMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \text{MethodMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \text{None}, \text{Return}),$

$(\text{ClassMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \text{MethodMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \lfloor 0 \rfloor, \text{Enter}),$

$(\text{ClassMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \text{MethodMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \lfloor 0 \rfloor, \text{Normal}),$

$(\text{ClassMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \text{MethodMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \lfloor 0 \rfloor, \text{Return}),$

$(\text{"C"}, \text{"M"}, \text{None}, \text{Enter}),$

$(\text{"C"}, \text{"M"}, \lfloor 0 \rfloor, \text{Enter}),$

$(\text{"C"}, \text{"M"}, \lfloor 1 \rfloor, \text{Enter}),$

$(\text{"C"}, \text{"M"}, \text{None}, \text{Return})$

$\}$

$\langle \text{proof} \rangle$

lemma *Main-Entry-valid* [simp]:

$\text{JVMCFG-Interpret.valid-node } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})) \text{"C"} \text{"M"}$

$(\text{ClassMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \text{MethodMain } (\text{Abs-wf-jvmprog } (EP, \text{Phi-EP})), \text{None}, \text{Enter})$

$\langle \text{proof} \rangle$

lemma *main-0-Enter-reachable* [simp]: $(P, C0, \text{Main}) \vdash \Rightarrow (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Enter})$

$\langle \text{proof} \rangle$

lemma *main-0-Normal-reachable* [simp]: $(P, C0, \text{Main}) \vdash \Rightarrow (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Normal})$

$\langle \text{proof} \rangle$

lemma *main-0-Return-reachable* [simp]: $(P, C0, \text{Main}) \vdash \Rightarrow (\text{ClassMain } P, \text{MethodMain } P, \lfloor 0 \rfloor, \text{Return})$

$\langle \text{proof} \rangle$

lemma *Exit-reachable [simp]*: $(P, C0, Main) \vdash \Rightarrow (ClassMain P, MethodMain P, None, Return)$
 ⟨proof⟩

definition

cfg-wf-prog =
 $\{(P, C0, Main). (\forall n. JVMCFG-Interpret.valid-node P C0 Main n \longrightarrow$
 $(\exists as. CFG.valid-path' sourcenode targetnode kind (valid-edge (P, C0, Main))$
 $(get-return-edges P) n as (ClassMain P, MethodMain P, None,$
 $Return))))\}$

typedef *cfg-wf-prog* = *cfg-wf-prog*
 ⟨proof⟩

abbreviation *lift-to-cfg-wf-prog* :: $(jvm-method \Rightarrow 'a) \Rightarrow (cfg-wf-prog \Rightarrow 'a)$
 $(-CFG)$
where $f_{CFG} \equiv (\lambda P. f (Rep-cfg-wf-prog P))$

lemma *valid-edge-CFG-def*: $valid-edge_{CFG} P = valid-edge (fst_{CFG} P, fst (snd_{CFG} P), snd (snd_{CFG} P))$
 ⟨proof⟩

interpretation *JVMCFG-Postdomination*:

Postdomination sourcenode targetnode kind valid-edge_{CFG} P
 $(ClassMain (fst_{CFG} P), MethodMain (fst_{CFG} P), None, Enter)$
 $(\lambda(C, M, pc, type). (C, M)) get-return-edges (fst_{CFG} P)$
 $((ClassMain (fst_{CFG} P), MethodMain (fst_{CFG} P)), [], []) \# procs (PROG (fst_{CFG} P))$
 $(ClassMain (fst_{CFG} P), MethodMain (fst_{CFG} P))$
 $(ClassMain (fst_{CFG} P), MethodMain (fst_{CFG} P), None, Return)$
for *P*
 ⟨proof⟩

end

theory *JVMSDG imports JVMCFG-wf JVMPPostdomination ../StaticInter/SDG*
begin

interpretation *JVMCFGExit-wf-new-type*:

CFGExit-wf sourcenode targetnode kind valid-edge_{CFG} P
 $(ClassMain (fst_{CFG} P), MethodMain (fst_{CFG} P), None, Enter)$
 $(\lambda(C, M, pc, type). (C, M)) get-return-edges (fst_{CFG} P)$
 $((ClassMain (fst_{CFG} P), MethodMain (fst_{CFG} P)), [], []) \# procs (PROG (fst_{CFG} P))$
 $(ClassMain (fst_{CFG} P), MethodMain (fst_{CFG} P))$
 $(ClassMain (fst_{CFG} P), MethodMain (fst_{CFG} P), None, Return)$
 $Def (fst_{CFG} P) Use (fst_{CFG} P) ParamDefs (fst_{CFG} P) ParamUses (fst_{CFG} P)$
for *P*

<proof>

interpretation *JVM-SDG* :

SDG sourcenode targetnode kind valid-edge CFG P
(ClassMain (fst_CFG P), MethodMain (fst_CFG P), None, Enter)
(λ(C, M, pc, type). (C, M)) get-return-edges (fst_CFG P)
((ClassMain (fst_CFG P), MethodMain (fst_CFG P)), [], []) # procs (PROG (fst_CFG P))
(ClassMain (fst_CFG P), MethodMain (fst_CFG P))
(ClassMain (fst_CFG P), MethodMain (fst_CFG P), None, Return)
Def (fst_CFG P) Use (fst_CFG P) ParamDefs (fst_CFG P) ParamUses (fst_CFG P)
for *P*
<proof>

end

theory *HRBSlicing imports*

StaticInter/CFGExit-wf
StaticInter/SemanticsCFG
StaticInter/FundamentalProperty
Proc/ProcSDG
JinjaVM-Inter/JVMSDG

begin

end

Bibliography

- [1] Susan Horwitz and Thomas Reps and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.
- [2] Thomas Reps and Susan Horwitz and Mooly Sagiv and Genevieve Rosay. Speeding up slicing. In *Proc. of FSE'94*, pages 11–20. ACM, 1994
- [3] Daniel Wasserrab. Towards certified slicing. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. <http://isa-afp.org/entries/Slicing.shtml>, September 2008. Formal proof development.