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

Daniel Wasserrab

March 17, 2025

**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 *list*s

**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*:
⟦*i* < *length xs*; *length xs* = *length ys*; *distinct xs*⟧

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

**lemma** *fun-upds-eq*:
**assumes** $V \in set\ xs$ **and** $length\ xs = length\ ys$ **and** $distinct\ xs$
**shows** $f(xs\ [:=]\ ys)\ V = f'(xs\ [:=]\ ys)\ V$
⟨*proof*⟩

**lemma** *fun-upds-notin*:$x \notin set\ xs \Longrightarrow f(xs\ [:=]\ ys)\ x = f\ x$
⟨*proof*⟩

### 1.1.1 *distinct-fst*

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

**lemma** *distinct-fst-Nil* [*simp*]:
$distinct\text{-}fst\ []$
⟨*proof*⟩

**lemma** *distinct-fst-Cons* [*simp*]:
$distinct\text{-}fst\ ((k,x)\#kxs) = (distinct\text{-}fst\ kxs \wedge (\forall y.\ (k,y) \notin set\ kxs))$
⟨*proof*⟩

**lemma** *distinct-fst-isin-same-fst*:
$[\![(x,y) \in set\ xs;\ (x,y') \in set\ xs;\ distinct\text{-}fst\ xs]\!]$
$\Longrightarrow y = y'$
⟨*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* ⇀ *'val*) ⇒ (*'var* ⇀ *'val*) (‹⇑-›)
| *PredicateEdge* (*'var* ⇀ *'val*) ⇒ *bool* (‹'(-')$_{\surd}$›)
| *CallEdge* (*'var* ⇀ *'val*) × *'ret* ⇒ *bool 'ret 'pname*
    ((*'var* ⇀ *'val*) ⇀ *'val*) *list* (‹-:-↪$_{-}$-› *70*)
| *ReturnEdge* (*'var* ⇀ *'val*) × *'ret* ⇒ *bool 'pname*
    (*'var* ⇀ *'val*) ⇒ (*'var* ⇀ *'val*) ⇒ (*'var* ⇀ *'val*) (‹-↩$_{-}$-› *70*)

**definition** *intra-kind* :: (*'var*,*'val*,*'ret*,*'pname*) *edge-kind* ⇒ *bool*
**where** *intra-kind et* ≡ (∃ *f*. *et* = ⇑*f*) ∨ (∃ *Q*. *et* = (*Q*)$_{\surd}$)

**lemma** *edge-kind-cases* [*case-names Intra Call Return*]:
⟦*intra-kind et* ⟹ *P*; ⋀*Q r p fs*. *et* = *Q*:*r*↪$_{p}$*fs* ⟹ *P*;
  ⋀*Q p f*. *et* = *Q*↩$_{p}$*f* ⟹ *P*⟧ ⟹ *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* ⇒ *'node*
 **fixes** *targetnode* :: *'edge* ⇒ *'node*
 **fixes** *kind* :: *'edge* ⇒ (*'var*,*'val*,*'ret*,*'pname*) *edge-kind*
 **fixes** *valid-edge* :: *'edge* ⇒ *bool*
 **fixes** *Entry*::*'node* (‹'('-*Entry*'-')›)
 **fixes** *get-proc*::*'node* ⇒ *'pname*
 **fixes** *get-return-edges*::*'edge* ⇒ *'edge set*
 **fixes** *procs*::(*'pname* × *'var list* × *'var list*) *list*
 **fixes** *Main*::*'pname*
 **assumes** *Entry-target* [*dest*]: ⟦*valid-edge a*; *targetnode a* = (-*Entry*-)⟧ ⟹ *False*
 **and** *get-proc-Entry*:*get-proc* (-*Entry*-) = *Main*
 **and** *Entry-no-call-source*:
  ⟦*valid-edge a*; *kind a* = *Q*:*r*↪$_{p}$*fs*; *sourcenode a* = (-*Entry*-)⟧ ⟹ *False*
 **and** *edge-det*:
  ⟦*valid-edge a*; *valid-edge a'*; *sourcenode a* = *sourcenode a'*;
   *targetnode a* = *targetnode a'*⟧ ⟹ *a* = *a'*

**and** *Main-no-call-target*:$[\![$*valid-edge a*; *kind a* = $Q{:}r\hookrightarrow_{Main}f]\!] \Longrightarrow$ *False*
**and** *Main-no-return-source*:$[\![$*valid-edge a*; *kind a* = $Q'\hookleftarrow_{Main}f']\!] \Longrightarrow$ *False*
**and** *callee-in-procs*:
$[\![$*valid-edge a*; *kind a* = $Q{:}r\hookrightarrow_{p}fs]\!] \Longrightarrow \exists$ *ins outs*. $(p,ins,outs) \in$ *set procs*
**and** *get-proc-intra*:$[\![$*valid-edge a*; *intra-kind*(*kind a*)$]\!]$
$\Longrightarrow$ *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)
**and** *get-proc-call*:
$[\![$*valid-edge a*; *kind a* = $Q{:}r\hookrightarrow_{p}fs]\!] \Longrightarrow$ *get-proc* (*targetnode a*) = *p*
**and** *get-proc-return*:
$[\![$*valid-edge a*; *kind a* = $Q'\hookleftarrow_{p}f']\!] \Longrightarrow$ *get-proc* (*sourcenode a*) = *p*
**and** *call-edges-only*:$[\![$*valid-edge a*; *kind a* = $Q{:}r\hookrightarrow_{p}fs]\!]$
$\Longrightarrow \forall a'$. *valid-edge* $a' \wedge$ *targetnode* $a'$ = *targetnode a* $\longrightarrow$
($\exists$ *Qx rx fsx*. *kind* $a' = Qx{:}rx\hookrightarrow_{p}fsx$)
**and** *return-edges-only*:$[\![$*valid-edge a*; *kind a* = $Q'\hookleftarrow_{p}f']\!]$
$\Longrightarrow \forall a'$. *valid-edge* $a' \wedge$ *sourcenode* $a'$ = *sourcenode a* $\longrightarrow$
($\exists$ *Qx fx*. *kind* $a' = Qx\hookleftarrow_{p}fx$)
**and** *get-return-edge-call*:
$[\![$*valid-edge a*; *kind a* = $Q{:}r\hookrightarrow_{p}fs]\!] \Longrightarrow$ *get-return-edges a* $\neq \{\}$
**and** *get-return-edges-valid*:
$[\![$*valid-edge a*; $a' \in$ *get-return-edges a*$]\!] \Longrightarrow$ *valid-edge* $a'$
**and** *only-call-get-return-edges*:
$[\![$*valid-edge a*; $a' \in$ *get-return-edges a*$]\!] \Longrightarrow \exists$ *Q r p fs*. *kind a* = $Q{:}r\hookrightarrow_{p}fs$
**and** *call-return-edges*:
$[\![$*valid-edge a*; *kind a* = $Q{:}r\hookrightarrow_{p}fs$; $a' \in$ *get-return-edges a*$]\!]$
$\Longrightarrow \exists Q'\ f'$. *kind* $a' = Q'\hookleftarrow_{p}f'$
**and** *return-needs-call*: $[\![$*valid-edge a*; *kind a* = $Q'\hookleftarrow_{p}f']\!]$
$\Longrightarrow \exists!a'$. *valid-edge* $a' \wedge$ ($\exists$ *Q r fs*. *kind* $a' = Q{:}r\hookrightarrow_{p}fs$) $\wedge$ *a* $\in$ *get-return-edges*
$a'$
**and** *intra-proc-additional-edge*:
$[\![$*valid-edge a*; $a' \in$ *get-return-edges a*$]\!]$
$\Longrightarrow \exists a''$. *valid-edge* $a'' \wedge$ *sourcenode* $a''$ = *targetnode a* $\wedge$
*targetnode* $a''$ = *sourcenode* $a' \wedge$ *kind* $a'' = (\lambda cf.\ False)_{\surd}$
**and** *call-return-node-edge*:
$[\![$*valid-edge a*; $a' \in$ *get-return-edges a*$]\!]$
$\Longrightarrow \exists a''$. *valid-edge* $a'' \wedge$ *sourcenode* $a''$ = *sourcenode a* $\wedge$
*targetnode* $a''$ = *targetnode* $a' \wedge$ *kind* $a'' = (\lambda cf.\ False)_{\surd}$
**and** *call-only-one-intra-edge*:
$[\![$*valid-edge a*; *kind a* = $Q{:}r\hookrightarrow_{p}fs]\!]$
$\Longrightarrow \exists!a'$. *valid-edge* $a' \wedge$ *sourcenode* $a'$ = *sourcenode a* $\wedge$ *intra-kind*(*kind* $a'$)
**and** *return-only-one-intra-edge*:
$[\![$*valid-edge a*; *kind a* = $Q'\hookleftarrow_{p}f']\!]$
$\Longrightarrow \exists!a'$. *valid-edge* $a' \wedge$ *targetnode* $a'$ = *targetnode a* $\wedge$ *intra-kind*(*kind* $a'$)
**and** *same-proc-call-unique-target*:
$[\![$*valid-edge a*; *valid-edge* $a'$; *kind a* = $Q_1{:}r_1\hookrightarrow_{p}fs_1$; *kind* $a' = Q_2{:}r_2\hookrightarrow_{p}fs_2]\!]$
$\Longrightarrow$ *targetnode a* = *targetnode* $a'$
**and** *unique-callers*:*distinct-fst procs*
**and** *distinct-formal-ins*:$(p,ins,outs) \in$ *set procs* $\Longrightarrow$ *distinct ins*
**and** *distinct-formal-outs*:$(p,ins,outs) \in$ *set procs* $\Longrightarrow$ *distinct outs*

**begin**

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

**lemma** *call-intra-edge-False*:
  **assumes** *valid-edge a* **and** *kind a* = $Q{:}r\hookrightarrow_p fs$ **and** *valid-edge a'*
  **and** *sourcenode a* = *sourcenode a'* **and** *intra-kind*(*kind a'*)
  **shows** *kind a'* = $(\lambda cf.\ False)_{\surd}$
⟨*proof*⟩

**lemma** *formal-in-THE*:
  $[\![$*valid-edge a*; *kind a* = $Q{:}r\hookrightarrow_p fs$; (*p*,*ins*,*outs*) ∈ *set procs*$]\!]$
  $\Longrightarrow$ (*THE ins.* ∃ *outs.* (*p*,*ins*,*outs*) ∈ *set procs*) = *ins*
⟨*proof*⟩

**lemma** *formal-out-THE*:
  $[\![$*valid-edge a*; *kind a* = $Q\hookleftarrow_p f$; (*p*,*ins*,*outs*) ∈ *set procs*$]\!]$
  $\Longrightarrow$ (*THE outs.* ∃ *ins.* (*p*,*ins*,*outs*) ∈ *set procs*) = *outs*
⟨*proof*⟩

## Transfer and predicate functions

**fun** *params* :: (('*var* ⇀ '*val*) ⇀ '*val*) *list* ⇒ ('*var* ⇀ '*val*) ⇒ '*val option list*
**where** *params* [] *cf* = []
  | *params* (*f*#*fs*) *cf* = (*f cf*)#*params fs cf*

**lemma** *params-nth*:
  *i* < *length fs* $\Longrightarrow$ (*params fs cf*)!*i* = (*fs*!*i*) *cf*
⟨*proof*⟩

**lemma** [*simp*]:*length* (*params fs cf*) = *length fs*
  ⟨*proof*⟩

**fun** *transfer* :: ('*var*,'*val*,'*ret*,'*pname*) *edge-kind* ⇒ (('*var* ⇀ '*val*) × '*ret*) *list* ⇒
  (('*var* ⇀ '*val*) × '*ret*) *list*
**where** *transfer* (⇑*f*) (*cf*#*cfs*)    = (*f* (*fst cf*),*snd cf*)#*cfs*
  | *transfer* $(Q)_{\surd}$ (*cf*#*cfs*)      = (*cf*#*cfs*)
  | *transfer* ($Q{:}r\hookrightarrow_p fs$) (*cf*#*cfs*) =
      (*let ins* = *THE ins.* ∃ *outs.* (*p*,*ins*,*outs*) ∈ *set procs in*
                (*Map.empty*(*ins* [:=] *params fs* (*fst cf*)),*r*)#*cf*#*cfs*)

| *transfer* $(Q \hookleftarrow_{p} f\ )(cf\#cfs)$ $= (case\ cfs\ of\ [] \Rightarrow []$
$\mid cf'\#cfs' \Rightarrow (f\ (fst\ cf)\ (fst\ cf'),snd\ cf')\#cfs')$
| *transfer et* $[] = []$

**fun** *transfers* :: $('var,'val,'ret,'pname)$ *edge-kind list* $\Rightarrow (('var \rightharpoonup 'val) \times 'ret)$ *list* $\Rightarrow$
$(('var \rightharpoonup 'val) \times 'ret)$ *list*
**where** *transfers* $[]\ s = s$
| *transfers* $(et\#ets)\ s =$ *transfers ets* $(transfer\ et\ s)$

**fun** *pred* :: $('var,'val,'ret,'pname)$ *edge-kind* $\Rightarrow (('var \rightharpoonup 'val) \times 'ret)$ *list* $\Rightarrow$ *bool*
**where** *pred* $(\Uparrow f)\ (cf\#cfs) = True$
| *pred* $(Q)_{\surd}\ (cf\#cfs) = Q\ (fst\ cf)$
| *pred* $(Q{:}r \hookrightarrow_{p} fs)\ (cf\#cfs) = Q\ (fst\ cf,r)$
| *pred* $(Q \hookleftarrow_{p} f)\ (cf\#cfs) = (Q\ cf \wedge cfs \neq [])$
| *pred et* $[] = False$

**fun** *preds* :: $('var,'val,'ret,'pname)$ *edge-kind list* $\Rightarrow (('var \rightharpoonup 'val) \times 'ret)$ *list* $\Rightarrow$ *bool*
**where** *preds* $[]\ s = True$
| *preds* $(et\#ets)\ s = (pred\ et\ s \wedge preds\ ets\ (transfer\ et\ s))$

**lemma** *transfers-split*:
$(transfers\ (ets@ets')\ s) = (transfers\ ets'\ (transfers\ ets\ s))$
⟨*proof*⟩

**lemma** *preds-split*:
$(preds\ (ets@ets')\ s) = (preds\ ets\ s \wedge preds\ ets'\ (transfers\ ets\ s))$
⟨*proof*⟩

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

*valid-node*

**definition** *valid-node* :: $'node \Rightarrow bool$
**where** *valid-node* $n \equiv$
$(\exists\ a.\ valid\text{-}edge\ a \wedge (n = sourcenode\ a \vee n = targetnode\ a))$

**lemma** [*simp*]: *valid-edge* $a \Longrightarrow$ *valid-node* $(sourcenode\ a)$
⟨*proof*⟩

**lemma** [*simp*]: *valid-edge* $a \Longrightarrow$ *valid-node* $(targetnode\ a)$
⟨*proof*⟩

### 1.2.2 CFG paths

**inductive** *path* :: $'node \Rightarrow 'edge\ list \Rightarrow 'node \Rightarrow bool$
(‹- −-→* -› [51,0,0] 80)
**where**
*empty-path*: $valid\text{-}node\ n \Longrightarrow n\ -[]\rightarrow*\ n$

| *Cons-path*:
$[\![n''\ -as\rightarrow*\ n';\ valid\text{-}edge\ a;\ sourcenode\ a = n;\ targetnode\ a = n'']\!]$
$\Longrightarrow n\ -a\#as\rightarrow*\ n'$

**lemma** *path-valid-node*:
**assumes** $n\ -as\rightarrow*\ n'$ **shows** $valid\text{-}node\ n$ **and** $valid\text{-}node\ n'$
⟨*proof*⟩

**lemma** *empty-path-nodes* [*dest*]: $n\ -[]\rightarrow*\ n' \Longrightarrow n = n'$
⟨*proof*⟩

**lemma** *path-valid-edges*: $n\ -as\rightarrow*\ n' \Longrightarrow \forall a \in set\ as.\ valid\text{-}edge\ a$
⟨*proof*⟩

**lemma** *path-edge*: $valid\text{-}edge\ a \Longrightarrow sourcenode\ a\ -[a]\rightarrow*\ targetnode\ a$
⟨*proof*⟩

**lemma** *path-Append*: $[\![n\ -as\rightarrow*\ n'';\ n''\ -as'\rightarrow*\ n']\!]$
$\Longrightarrow n\ -as@as'\rightarrow*\ n'$
⟨*proof*⟩

**lemma** *path-split*:
**assumes** $n\ -as@a\#as'\rightarrow*\ n'$
**shows** $n\ -as\rightarrow*\ sourcenode\ a$ **and** $valid\text{-}edge\ a$ **and** $targetnode\ a\ -as'\rightarrow*\ n'$
⟨*proof*⟩

**lemma** *path-split-Cons*:
**assumes** $n\ -as\rightarrow*\ n'$ **and** $as \neq []$
**obtains** $a'\ as'$ **where** $as = a'\#as'$ **and** $n = sourcenode\ a'$
**and** $valid\text{-}edge\ a'$ **and** $targetnode\ a'\ -as'\rightarrow*\ n'$
⟨*proof*⟩

**lemma** *path-split-snoc*:
**assumes** $n\ -as\rightarrow*\ n'$ **and** $as \neq []$
**obtains** $a'\ as'$ **where** $as = as'@[a']$ **and** $n\ -as'\rightarrow*\ sourcenode\ a'$
**and** $valid\text{-}edge\ a'$ **and** $n' = targetnode\ a'$
⟨*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** $(\text{-}Entry\text{-})\ -as\rightarrow*\ n'$ **and** $n' \neq (\text{-}Entry\text{-})$
**obtains** $n\ a$ **where** $sourcenode\ a = (\text{-}Entry\text{-})$ **and** $targetnode\ a = n$
**and** $n\ -tl\ as\rightarrow*\ n'$ **and** $valid\text{-}edge\ a$ **and** $a = hd\ as$
⟨*proof*⟩

**lemma** *path-det*:
$[\![n\ -as\rightarrow*\ n';\ n\ -as\rightarrow*\ n'']\!] \Longrightarrow n' = n''$
⟨*proof*⟩

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

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

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

**lemma** *path-sourcenode*:
$[\![n\ -as\rightarrow*\ n';\ as \neq []]\!] \Longrightarrow hd\ (sourcenodes\ as) = n$
⟨*proof*⟩

**lemma** *path-targetnode*:
$[\![n\ -as\rightarrow*\ n';\ as \neq []]\!] \Longrightarrow last\ (targetnodes\ as) = n'$
⟨*proof*⟩

**lemma** *sourcenodes-is-n-Cons-butlast-targetnodes*:
$[\![n\ -as\rightarrow*\ n';\ as \neq []]\!] \Longrightarrow$
$sourcenodes\ as = n\#(butlast\ (targetnodes\ as))$
⟨*proof*⟩

**lemma** *targetnodes-is-tl-sourcenodes-App-n'*:
$[\![n\ -as\rightarrow*\ n';\ as \neq []]\!] \Longrightarrow$
$targetnodes\ as = (tl\ (sourcenodes\ as))@[n']$
⟨*proof*⟩

## Intraprocedural paths

**definition** *intra-path* :: *'node* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*
(‹- −-→ι∗ -› [51,0,0] 80)
**where** $n\ -as\rightarrow_\iota*\ n' \equiv n\ -as\rightarrow*\ n' \wedge (\forall\ a \in set\ as.\ intra\text{-}kind(kind\ a))$

**lemma** *intra-path-get-procs*:
**assumes** $n\ -as\rightarrow_\iota*\ n'$ **shows** $get\text{-}proc\ n = get\text{-}proc\ n'$
⟨*proof*⟩

**lemma** *intra-path-Append*:
$[\![n\ -as\rightarrow_\iota*\ n'';\ n''\ -as'\rightarrow_\iota*\ n']\!] \Longrightarrow n\ -as@as'\rightarrow_\iota*\ n'$
⟨*proof*⟩

**lemma** *get-proc-get-return-edges*:
**assumes** *valid-edge a* **and** $a' \in get\text{-}return\text{-}edges\ a$
**shows** $get\text{-}proc(targetnode\ a) = get\text{-}proc(sourcenode\ a')$
⟨*proof*⟩

## Valid paths

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

**fun** *valid-path-aux* :: *'edge list* ⇒ *'edge list* ⇒ *bool*
**where** *valid-path-aux cs* [] ⟷ *True*
| *valid-path-aux cs* (*a*#*as*) ⟷
(*case* (*kind a*) *of* $Q{:}r\hookrightarrow_p fs$ ⇒ *valid-path-aux* (*a*#*cs*) *as*
| $Q\hookleftarrow_p f$ ⇒ *case cs of* [] ⇒ *valid-path-aux* [] *as*
| *c'*#*cs'* ⇒ *a* ∈ *get-return-edges c'* ∧
*valid-path-aux cs' as*
| - ⇒ *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*: ⋀*cs*. *P cs* []
⋀*cs a as*. ⟦*intra-kind*(*kind a*); *valid-path-aux cs as*; *P cs as*⟧ ⟹ *P cs* (*a*#*as*)
⋀*cs a as Q r p fs*. ⟦*kind a* = $Q{:}r\hookrightarrow_p fs$; *valid-path-aux* (*a*#*cs*) *as*; *P* (*a*#*cs*) *as*⟧
⟹ *P cs* (*a*#*as*)

$\bigwedge cs\ a\ as\ Q\ p\ f.\ [\![kind\ a = Q\hookleftarrow_p f;\ cs = [];\ valid\text{-}path\text{-}aux\ []\ as;\ P\ []\ as]\!]$
$\Longrightarrow P\ cs\ (a\#as)$
$\bigwedge cs\ a\ as\ Q\ p\ f\ c'\ cs'.\ [\![kind\ a = Q\hookleftarrow_p f;\ cs = c'\#cs';\ valid\text{-}path\text{-}aux\ cs'\ as;$
$a \in get\text{-}return\text{-}edges\ c';\ P\ cs'\ as]\!]$
$\Longrightarrow P\ cs\ (a\#as)$
**shows** $P\ xs\ ys$
⟨proof⟩

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

**lemma** *valid-path-aux-callstack-prefix*:
$valid\text{-}path\text{-}aux\ (cs@cs')\ as \Longrightarrow valid\text{-}path\text{-}aux\ cs\ as$
⟨proof⟩

**fun** *upd-cs* :: 'edge list ⇒ 'edge list ⇒ 'edge list
**where** $upd\text{-}cs\ cs\ [] = cs$
| $upd\text{-}cs\ cs\ (a\#as) =$
$(case\ (kind\ a)\ of\ Q{:}r\hookrightarrow_p fs \Rightarrow upd\text{-}cs\ (a\#cs)\ as$
$\mid Q\hookleftarrow_p f \Rightarrow case\ cs\ of\ [] \Rightarrow upd\text{-}cs\ cs\ as$
$\mid c'\#cs' \Rightarrow upd\text{-}cs\ cs'\ as$
$\mid\ \_ \Rightarrow upd\text{-}cs\ cs\ as)$

**lemma** *upd-cs-empty* [*dest*]:
$upd\text{-}cs\ cs\ [] = [] \Longrightarrow cs = []$
⟨proof⟩

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

**lemma** *upd-cs-Append*:
$[\![upd\text{-}cs\ cs\ as = cs';\ upd\text{-}cs\ cs'\ as' = cs'']\!] \Longrightarrow upd\text{-}cs\ cs\ (as@as') = cs''$
⟨proof⟩

**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** $upd\text{-}cs\ []\ ys = []$
⟨proof⟩

**lemma** *upd-cs-snoc-Return-Cons*:
  **assumes** $kind\ a = Q{\hookleftarrow}_p f$
  **shows** $upd\text{-}cs\ cs\ as = c'\#cs' \Longrightarrow upd\text{-}cs\ cs\ (as@[a]) = cs'$
⟨*proof*⟩

**lemma** *upd-cs-snoc-Call*:
  **assumes** $kind\ a = Q{:}r{\hookrightarrow}_p fs$
  **shows** $upd\text{-}cs\ cs\ (as@[a]) = a\#(upd\text{-}cs\ cs\ as)$
⟨*proof*⟩

**lemma** *valid-path-aux-split*:
  **assumes** $valid\text{-}path\text{-}aux\ cs\ (as@as')$
  **shows** $valid\text{-}path\text{-}aux\ cs\ as$ **and** $valid\text{-}path\text{-}aux\ (upd\text{-}cs\ cs\ as)\ as'$
  ⟨*proof*⟩

**lemma** *valid-path-aux-Append*:
  $[\![valid\text{-}path\text{-}aux\ cs\ as;\ valid\text{-}path\text{-}aux\ (upd\text{-}cs\ cs\ as)\ as']\!]$
  $\Longrightarrow valid\text{-}path\text{-}aux\ cs\ (as@as')$
⟨*proof*⟩

**lemma** *vpa-snoc-Call*:
  **assumes** $kind\ a = Q{:}r{\hookrightarrow}_p fs$
  **shows** $valid\text{-}path\text{-}aux\ cs\ as \Longrightarrow valid\text{-}path\text{-}aux\ cs\ (as@[a])$
⟨*proof*⟩

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

**lemma** *valid-path-aux-valid-path*:
  $valid\text{-}path\text{-}aux\ cs\ as \Longrightarrow valid\text{-}path\ as$
⟨*proof*⟩

**lemma** *valid-path-split*:
  **assumes** $valid\text{-}path\ (as@as')$ **shows** $valid\text{-}path\ as$ **and** $valid\text{-}path\ as'$
  ⟨*proof*⟩

**definition** *valid-path'* :: *'node* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*
(‹- −-→$_{\surd}$∗ -› [51,0,0] 80)
**where** *vp-def*:$n$ −*as*→$_{\surd}$∗ $n'$ ≡ $n$ −*as*→∗ $n'$ ∧ *valid-path as*

**lemma** *intra-path-vp*:
**assumes** $n$ −*as*→$_{\iota}$∗ $n'$ **shows** $n$ −*as*→$_{\surd}$∗ $n'$
⟨*proof*⟩

**lemma** *vp-split-Cons*:
**assumes** $n$ −*as*→$_{\surd}$∗ $n'$ **and** *as* ≠ []
**obtains** $a'$ $as'$ **where** *as* = $a'$#$as'$ **and** $n$ = *sourcenode* $a'$
**and** *valid-edge* $a'$ **and** *targetnode* $a'$ −$as'$→$_{\surd}$∗ $n'$
⟨*proof*⟩

**lemma** *vp-split-snoc*:
**assumes** $n$ −*as*→$_{\surd}$∗ $n'$ **and** *as* ≠ []
**obtains** $a'$ $as'$ **where** *as* = $as'$@[$a'$] **and** $n$ −$as'$→$_{\surd}$∗ *sourcenode* $a'$
**and** *valid-edge* $a'$ **and** $n'$ = *targetnode* $a'$
⟨*proof*⟩

**lemma** *vp-split*:
**assumes** $n$ −*as*@*a*#$as'$→$_{\surd}$∗ $n'$
**shows** $n$ −*as*→$_{\surd}$∗ *sourcenode a* **and** *valid-edge a* **and** *targetnode a* −$as'$→$_{\surd}$∗ $n'$
⟨*proof*⟩

**lemma** *vp-split-second*:
**assumes** $n$ −*as*@*a*#$as'$→$_{\surd}$∗ $n'$ **shows** *sourcenode a* −*a*#$as'$→$_{\surd}$∗ $n'$
⟨*proof*⟩

**function** *valid-path-rev-aux* :: *'edge list* ⇒ *'edge list* ⇒ *bool*
**where** *valid-path-rev-aux cs* [] ⟷ *True*
| *valid-path-rev-aux cs* (*as*@[*a*]) ⟷
(*case* (*kind a*) *of* *Q*↩$_{p}$*f* ⇒ *valid-path-rev-aux* (*a*#*cs*) *as*
| *Q*:*r*↪$_{p}$*fs* ⇒ *case cs of* [] ⇒ *valid-path-rev-aux* [] *as*
| $c'$#$cs'$ ⇒ $c'$ ∈ *get-return-edges a* ∧
*valid-path-rev-aux* $cs'$ *as*
| - ⇒ *valid-path-rev-aux cs as*)
⟨*proof*⟩
**termination** ⟨*proof*⟩

**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*. ⟦*intra-kind*(*kind a*); *valid-path-rev-aux cs as*; *P cs as*⟧
$\Longrightarrow$ *P cs* (*as*@[*a*])
$\bigwedge$*cs a as Q p f*. ⟦*kind a* = $Q \hookleftarrow_p f$; *valid-path-rev-aux* (*a*#*cs*) *as*; *P* (*a*#*cs*) *as*⟧
$\Longrightarrow$ *P cs* (*as*@[*a*])
$\bigwedge$*cs a as Q r p fs*. ⟦*kind a* = $Q{:}r \hookrightarrow_p fs$; *cs* = []; *valid-path-rev-aux* [] *as*;
*P* [] *as*⟧ $\Longrightarrow$ *P cs* (*as*@[*a*])
$\bigwedge$*cs a as Q r p fs c' cs'*. ⟦*kind a* = $Q{:}r \hookrightarrow_p fs$; *cs* = *c'*#*cs'*;
*valid-path-rev-aux cs' as*; *c'* ∈ *get-return-edges a*; *P cs' as*⟧
$\Longrightarrow$ *P cs* (*as*@[*a*])
**shows** *P xs ys*
⟨*proof*⟩

**lemma** *vpra-callstack-prefix*:
*valid-path-rev-aux* (*cs*@*cs'*) *as* $\Longrightarrow$ *valid-path-rev-aux cs as*
⟨*proof*⟩

**function** *upd-rev-cs* :: *'edge list* ⇒ *'edge list* ⇒ *'edge list*
**where** *upd-rev-cs cs* [] = *cs*
| *upd-rev-cs cs* (*as*@[*a*]) =
(*case* (*kind a*) *of* $Q \hookleftarrow_p f$ ⇒ *upd-rev-cs* (*a*#*cs*) *as*
| $Q{:}r \hookrightarrow_p fs$ ⇒ *case cs of* [] ⇒ *upd-rev-cs cs as*
| *c'*#*cs'* ⇒ *upd-rev-cs cs' as*
| - ⇒ *upd-rev-cs cs as*)
⟨*proof*⟩
**termination** ⟨*proof*⟩

**lemma** *upd-rev-cs-empty* [*dest*]:
*upd-rev-cs cs* [] = [] $\Longrightarrow$ *cs* = []
⟨*proof*⟩

**lemma** *valid-path-rev-aux-split*:
**assumes** *valid-path-rev-aux cs* (*as*@*as'*)
**shows** *valid-path-rev-aux cs as'* **and** *valid-path-rev-aux* (*upd-rev-cs cs as'*) *as*
⟨*proof*⟩

**lemma** *valid-path-rev-aux-Append*:
⟦*valid-path-rev-aux cs as'*; *valid-path-rev-aux* (*upd-rev-cs cs as'*) *as*⟧
$\Longrightarrow$ *valid-path-rev-aux cs* (*as*@*as'*)
⟨*proof*⟩

**lemma** *vpra-Cons-intra*:
**assumes** *intra-kind*(*kind a*)
**shows** *valid-path-rev-aux cs as* $\Longrightarrow$ *valid-path-rev-aux cs* (*a*#*as*)
⟨*proof*⟩

**lemma** *vpra-Cons-Return*:
**assumes** *kind a* = $Q \hookleftarrow_p f$
**shows** *valid-path-rev-aux cs as* $\Longrightarrow$ *valid-path-rev-aux cs* (*a*#*as*)
⟨*proof*⟩
**lemma** *upd-rev-cs-Cons-intra*:
**assumes** *intra-kind*(*kind a*) **shows** *upd-rev-cs cs* (*a*#*as*) = *upd-rev-cs cs as*
⟨*proof*⟩

**lemma** *upd-rev-cs-Cons-Return*:
**assumes** *kind a* = $Q \hookleftarrow_p f$ **shows** *upd-rev-cs cs* (*a*#*as*) = *a*#(*upd-rev-cs cs as*)
⟨*proof*⟩

**lemma** *upd-rev-cs-Cons-Call-Cons*:
**assumes** *kind a* = $Q{:}r \hookrightarrow_p fs$
**shows** *upd-rev-cs cs as* = *c'*#*cs'* $\Longrightarrow$ *upd-rev-cs cs* (*a*#*as*) = *cs'*
⟨*proof*⟩

**lemma** *upd-rev-cs-Cons-Call-Cons-Empty*:
**assumes** *kind a* = $Q{:}r \hookrightarrow_p fs$
**shows** *upd-rev-cs cs as* = [] $\Longrightarrow$ *upd-rev-cs cs* (*a*#*as*) = []
⟨*proof*⟩

**definition** *valid-call-list* :: *'edge list* ⇒ *'node* ⇒ *bool*
**where** *valid-call-list cs n* ≡
∀ *cs' c cs''*. *cs* = *cs'*@*c*#*cs''* ⟶ (*valid-edge c* ∧ (∃ *Q r p fs*. (*kind c* = $Q{:}r \hookrightarrow_p fs$)
∧
*p* = *get-proc* (*case cs' of* [] ⇒ *n* | - ⇒ *last* (*sourcenodes cs'*))))

**definition** *valid-return-list* :: *'edge list* ⇒ *'node* ⇒ *bool*
**where** *valid-return-list cs n* ≡
∀ *cs' c cs''*. *cs* = *cs'*@*c*#*cs''* ⟶ (*valid-edge c* ∧ (∃ *Q p f*. (*kind c* = $Q \hookleftarrow_p f$) ∧
*p* = *get-proc* (*case cs' of* [] ⇒ *n* | - ⇒ *last* (*targetnodes cs'*))))

**lemma** *valid-call-list-valid-edges*:
**assumes** *valid-call-list cs n* **shows** ∀ *c* ∈ *set cs*. *valid-edge c*
⟨*proof*⟩

**lemma** *valid-return-list-valid-edges*:
  **assumes** *valid-return-list rs n* **shows** $\forall r \in set\ rs.\ valid\text{-}edge\ r$
⟨*proof*⟩

**lemma** *vpra-empty-valid-call-list-rev*:
  $valid\text{-}call\text{-}list\ cs\ n \Longrightarrow valid\text{-}path\text{-}rev\text{-}aux\ []\ (rev\ cs)$
⟨*proof*⟩

**lemma** *vpa-upd-cs-cases*:
  $[\![valid\text{-}path\text{-}aux\ cs\ as;\ valid\text{-}call\text{-}list\ cs\ n;\ n\ -as\rightarrow*\ n']\!]$
  $\Longrightarrow case\ (upd\text{-}cs\ cs\ as)\ of\ []\ \Rightarrow (\forall c \in set\ cs.\ \exists a \in set\ as.\ a \in get\text{-}return\text{-}edges\ c)$
  $|\ cx\#csx \Rightarrow valid\text{-}call\text{-}list\ (cx\#csx)\ n'$
⟨*proof*⟩

**lemma** *vpa-valid-call-list-valid-return-list-vpra*:
  $[\![valid\text{-}path\text{-}aux\ cs\ cs';\ valid\text{-}call\text{-}list\ cs\ n;\ valid\text{-}return\text{-}list\ cs'\ n']\!]$
  $\Longrightarrow valid\text{-}path\text{-}rev\text{-}aux\ cs'\ (rev\ cs)$
⟨*proof*⟩

**lemma** *vpa-to-vpra*:
  $[\![valid\text{-}path\text{-}aux\ cs\ as;\ valid\text{-}path\text{-}aux\ (upd\text{-}cs\ cs\ as)\ cs';$
    $n\ -as\rightarrow*\ n';\ valid\text{-}call\text{-}list\ cs\ n;\ valid\text{-}return\text{-}list\ cs'\ n'']\!]$
  $\Longrightarrow valid\text{-}path\text{-}rev\text{-}aux\ cs'\ as \wedge valid\text{-}path\text{-}rev\text{-}aux\ (upd\text{-}rev\text{-}cs\ cs'\ as)\ (rev\ cs)$
⟨*proof*⟩

**lemma** *vp-to-vpra*:
  $n\ -as\rightarrow_{\surd}*\ n' \Longrightarrow valid\text{-}path\text{-}rev\text{-}aux\ []\ as$
⟨*proof*⟩

### Same level paths

**fun** *same-level-path-aux* :: *'edge list* ⇒ *'edge list* ⇒ *bool*
  **where** $same\text{-}level\text{-}path\text{-}aux\ cs\ [] \longleftrightarrow True$
  $|\ same\text{-}level\text{-}path\text{-}aux\ cs\ (a\#as) \longleftrightarrow$
    $(case\ (kind\ a)\ of\ Q{:}r\hookrightarrow_{p}fs \Rightarrow same\text{-}level\text{-}path\text{-}aux\ (a\#cs)\ as$
    $|\ Q\hookleftarrow_{p}f \Rightarrow case\ cs\ of\ [] \Rightarrow False$
    $|\ c'\#cs' \Rightarrow a \in get\text{-}return\text{-}edges\ c' \wedge$
    $same\text{-}level\text{-}path\text{-}aux\ cs'\ as$
    $|\ \text{-} \Rightarrow same\text{-}level\text{-}path\text{-}aux\ cs\ as)$

**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*: $\bigwedge cs.\ P\ cs\ []$
$\bigwedge cs\ a\ as.\ [\![intra\text{-}kind(kind\ a);\ same\text{-}level\text{-}path\text{-}aux\ cs\ as;\ P\ cs\ as]\!]$
$\Longrightarrow P\ cs\ (a\#as)$
$\bigwedge cs\ a\ as\ Q\ r\ p\ fs.\ [\![kind\ a = Q{:}r\hookrightarrow_p fs;\ same\text{-}level\text{-}path\text{-}aux\ (a\#cs)\ as;\ P\ (a\#cs)\ as]\!]$
$\Longrightarrow P\ cs\ (a\#as)$
$\bigwedge cs\ a\ as\ Q\ p\ f\ c'\ cs'.\ [\![kind\ a = Q\hookleftarrow_p f;\ cs = c'\#cs';\ same\text{-}level\text{-}path\text{-}aux\ cs'\ as;$
$a \in get\text{-}return\text{-}edges\ c';\ P\ cs'\ as]\!]$
$\Longrightarrow P\ cs\ (a\#as)$
**shows** *P xs ys*
⟨*proof*⟩

**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** $\forall\, c \in set\ cs.\ valid\text{-}edge\ c$ **and** $\forall\, a \in set\ as.\ valid\text{-}edge\ a$
**obtains** $\forall\, a \in set\ as.\ intra\text{-}kind(kind\ a)$
| $asx\ a\ asx'\ Q\ p\ f\ c'\ cs'$ **where** $as = asx@a\#asx'$ **and** *same-level-path-aux cs asx*
**and** $kind\ a = Q\hookleftarrow_p f$ **and** $upd\text{-}cs\ cs\ asx = c'\#cs'$ **and** $upd\text{-}cs\ cs\ (asx@[a]) = []$
**and** $a \in get\text{-}return\text{-}edges\ c'$ **and** $valid\text{-}edge\ c'$
**and** $\forall\, a \in set\ asx'.\ intra\text{-}kind(kind\ a)$
⟨*proof*⟩

**lemma** *same-level-path-aux-valid-path-aux*:
*same-level-path-aux cs as* $\Longrightarrow$ *valid-path-aux cs as*
⟨*proof*⟩

**lemma** *same-level-path-aux-Append*:
$[\![same\text{-}level\text{-}path\text{-}aux\ cs\ as;\ same\text{-}level\text{-}path\text{-}aux\ (upd\text{-}cs\ cs\ as)\ as']\!]$
$\Longrightarrow same\text{-}level\text{-}path\text{-}aux\ cs\ (as@as')$
⟨*proof*⟩

**lemma** *same-level-path-aux-callstack-Append*:
$same\text{-}level\text{-}path\text{-}aux\ cs\ as \Longrightarrow same\text{-}level\text{-}path\text{-}aux\ (cs@cs')\ as$
⟨*proof*⟩

**lemma** *same-level-path-upd-cs-callstack-Append*:
$[\![same\text{-}level\text{-}path\text{-}aux\ cs\ as;\ upd\text{-}cs\ cs\ as = cs']\!]$
$\Longrightarrow upd\text{-}cs\ (cs@cs'')\ as = (cs'@cs'')$
⟨*proof*⟩

**lemma** *slpa-split*:
  **assumes** *same-level-path-aux cs as* **and** $as = xs@ys$ **and** *upd-cs cs xs* = []
  **shows** *same-level-path-aux cs xs* **and** *same-level-path-aux* [] *ys*
⟨*proof*⟩

**lemma** *slpa-number-Calls-eq-number-Returns*:
  ⟦*same-level-path-aux cs as*; *upd-cs cs as* = [];
   ∀ *a* ∈ *set as*. *valid-edge a*; ∀ *c* ∈ *set cs*. *valid-edge c*⟧
  ⟹ *length* [*a*←*as*@*cs*. ∃ *Q r p fs*. *kind a* = $Q{:}r\hookrightarrow_{p}fs$] =
    *length* [*a*←*as*. ∃ *Q p f*. *kind a* = $Q\hookleftarrow_{p}f$]
⟨*proof*⟩

**lemma** *slpa-get-proc*:
  ⟦*same-level-path-aux cs as*; *upd-cs cs as* = []; *n* −*as*→∗ *n*′;
   ∀ *c* ∈ *set cs*. *valid-edge c*⟧
  ⟹ (*if cs* = [] *then get-proc n else get-proc*(*last*(*sourcenodes cs*))) = *get-proc n*′
⟨*proof*⟩

**lemma** *slpa-get-return-edges*:
  ⟦*same-level-path-aux cs as*; *cs* ≠ []; *upd-cs cs as* = [];
  ∀ *xs ys*. *as* = *xs*@*ys* ∧ *ys* ≠ [] ⟶ *upd-cs cs xs* ≠ []⟧
  ⟹ *last as* ∈ *get-return-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-cs cs as*)
⟨*proof*⟩

**lemma** *slpa-snoc-intra*:
  ⟦*same-level-path-aux cs as*; *intra-kind* (*kind a*)⟧
  ⟹ *same-level-path-aux cs* (*as*@[*a*])
⟨*proof*⟩

**lemma** *slpa-snoc-Call*:
  ⟦*same-level-path-aux cs as*; *kind a* = $Q{:}r\hookrightarrow_{p}fs$⟧
  ⟹ *same-level-path-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*⟧
⟹ *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′*⟧ ⟹ *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*⟧ ⟹
*length* [*a*←*as*. ∃ *Q r p fs*. *kind a* = $Q{:}r\hookrightarrow_{p}fs$] = *length* [*a*←*as*. ∃ *Q p f*. *kind a* = $Q\hookleftarrow_{p}f$]
⟨*proof*⟩

**lemma** *same-level-path-valid-path-Append*:
⟦*same-level-path as*; *valid-path as′*⟧ ⟹ *valid-path* (*as*@*as′*)
⟨*proof*⟩

**lemma** *valid-path-same-level-path-Append*:
⟦*valid-path as*; *same-level-path as′*⟧ ⟹ *valid-path* (*as*@*as′*)
⟨*proof*⟩

**lemma** *intras-same-level-path*:
**assumes** ∀ *a* ∈ *set as*. *intra-kind*(*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*→$_{\surd}$∗ $n'$

⟨*proof*⟩

**lemma** *intra-path-slp*: $n\ -as\rightarrow_\iota*\ n' \Longrightarrow n\ -as\rightarrow_{sl}*\ n'$
⟨*proof*⟩

**lemma** *slp-Append*:
$[\![n\ -as\rightarrow_{sl}*\ n''; n''\ -as'\rightarrow_{sl}*\ n']\!] \Longrightarrow n\ -as@as'\rightarrow_{sl}*\ n'$
⟨*proof*⟩

**lemma** *slp-vp-Append*:
$[\![n\ -as\rightarrow_{sl}*\ n''; n''\ -as'\rightarrow_{\surd}*\ n']\!] \Longrightarrow n\ -as@as'\rightarrow_{\surd}*\ n'$
⟨*proof*⟩

**lemma** *vp-slp-Append*:
$[\![n\ -as\rightarrow_{\surd}*\ n''; n''\ -as'\rightarrow_{sl}*\ n']\!] \Longrightarrow n\ -as@as'\rightarrow_{\surd}*\ n'$
⟨*proof*⟩

**lemma** *slp-get-proc*:
$n\ -as\rightarrow_{sl}*\ n' \Longrightarrow$ *get-proc* $n$ = *get-proc* $n'$
⟨*proof*⟩

**lemma** *same-level-path-inner-path*:
**assumes** $n\ -as\rightarrow_{sl}*\ n'$
**obtains** $as'$ **where** $n\ -as'\rightarrow_\iota*\ n'$ **and** *set*(*sourcenodes* $as'$) $\subseteq$ *set*(*sourcenodes* *as*)
⟨*proof*⟩

**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*) ($cf\#cfs'$) = $cf'\#cfs'$
⟨*proof*⟩

**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* $a = Q\hookleftarrow_p f$
**and** $m\ -as'@[a]\rightarrow_{sl}*$ *targetnode* $a$ **and** *targetnode* $a\ -as''\rightarrow_\iota*\ m'$
⟨*proof*⟩

**function** *same-level-path-rev-aux* :: *'edge list* ⇒ *'edge list* ⇒ *bool*
  **where** *same-level-path-rev-aux cs* [] ⟷ *True*
  | *same-level-path-rev-aux cs* (*as*@[*a*]) ⟷
    (*case* (*kind a*) *of* $Q\hookleftarrow_p f$ ⇒ *same-level-path-rev-aux* (*a*#*cs*) *as*
      | $Q{:}r\hookrightarrow_p fs$ ⇒ *case cs of* [] ⇒ *False*
        | *c'*#*cs'* ⇒ *c'* ∈ *get-return-edges a* ∧
          *same-level-path-rev-aux cs' as*
      | - ⇒ *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*: ⋀*cs*. *P cs* []
    ⋀*cs a as*. ⟦*intra-kind*(*kind a*); *same-level-path-rev-aux cs as*; *P cs as*⟧
      ⟹ *P cs* (*as*@[*a*])
    ⋀*cs a as Q p f*. ⟦*kind a* = $Q\hookleftarrow_p f$; *same-level-path-rev-aux* (*a*#*cs*) *as*; *P* (*a*#*cs*) *as*⟧
      ⟹ *P cs* (*as*@[*a*])
    ⋀*cs a as Q r p fs c' cs'*. ⟦*kind a* = $Q{:}r\hookrightarrow_p fs$; *cs* = *c'*#*cs'*;
      *same-level-path-rev-aux cs' as*; *c'* ∈ *get-return-edges a*; *P cs' as*⟧
    ⟹ *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*⟧
  ⟹ *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*→∗ *n'*;
  *valid-return-list cs n'*⟧
  ⟹ *same-level-path-aux* [] *as* ∧ *same-level-path-aux* (*upd-cs* [] *as*) *cs* ∧
    *upd-cs* (*upd-cs* [] *as*) *cs* = []
⟨*proof*⟩

## Lemmas on paths with (*-Entry-*)

**lemma** *path-Entry-target* [*dest*]:
  **assumes** *n* −*as*→∗ (*-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*))
  ⟨*proof*⟩

**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* = $Q\hookleftarrow_p f$
  **shows** *False*
⟨*proof*⟩

**lemma** *vpra-no-slpra*:
  ⟦*valid-path-rev-aux cs as*; $n\ -as\rightarrow*\ n'$; *valid-return-list cs n'*; $cs \neq []$;
    $\forall$ *xs ys. as = xs@ys* $\longrightarrow$ ($\neg$ *same-level-path-rev-aux cs ys* $\vee$ *upd-rev-cs cs ys* $\neq$
[])⟧
  $\Longrightarrow \exists\, a\ Q\ f.$ *valid-edge a* $\wedge$ *kind a* = $Q\hookleftarrow_{get\text{-}proc\ n}f$
⟨*proof*⟩

**lemma** *valid-Entry-path-cases*:
  **assumes** (*-Entry-*) $-as\rightarrow_{\surd}*\ n$ **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'* = $Q{:}r\hookrightarrow_p fs$) $\vee$
        ($\exists\, as'\ as''\ n'.\ as = as'@as'' \wedge as'' \neq [] \wedge n'\ -as''\rightarrow_{sl}*\ n$)
⟨*proof*⟩

**lemma** *valid-Entry-path-ascending-path*:
  **assumes** (*-Entry-*) $-as\rightarrow_{\surd}*\ n$
  **obtains** *as'* **where** (*-Entry-*) $-as'\rightarrow_{\surd}*\ n$
  **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'* = $Q{:}r\hookrightarrow_p fs$)
⟨*proof*⟩

**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* ⇒ *'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* ⇒ *'pname*
**and** *get-return-edges* :: *'edge* ⇒ *'edge set*
**and** *procs* :: (*'pname* × *'var list* × *'var list*) *list* **and** *Main* :: *'pname* +
**fixes** *Exit*::*'node* (‹'('-*Exit*'-')›)
**assumes** *Exit-source* [*dest*]: ⟦*valid-edge a*; *sourcenode a* = (-*Exit*-)⟧ ⟹ *False*
**and** *get-proc-Exit*:*get-proc* (-*Exit*-) = *Main*
**and** *Exit-no-return-target*:
⟦*valid-edge a*; *kind a* = $Q{\hookleftarrow}_p f$; *targetnode a* = (-*Exit*-)⟧ ⟹ *False*
**and** *Entry-Exit-edge*: ∃ *a*. *valid-edge a* ∧ *sourcenode a* = (-*Entry*-) ∧
*targetnode a* = (-*Exit*-) ∧ *kind a* = $(\lambda s.\ False)_{\surd}$

**begin**

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

**lemma** *Exit-noteq-Entry* [*dest*]:(-*Exit*-) = (-*Entry*-) ⟹ *False*
⟨*proof*⟩

**lemma** [*simp*]: *valid-node* (-*Entry*-)
⟨*proof*⟩

**lemma** [*simp*]: *valid-node* (-*Exit*-)
⟨*proof*⟩

## Definition of *method-exit*

**definition** *method-exit* :: *'node* ⇒ *bool*
**where** *method-exit n* ≡ *n* = (-*Exit*-) ∨
(∃ *a Q p f*. *n* = *sourcenode a* ∧ *valid-edge a* ∧ *kind a* = $Q{\hookleftarrow}_p f$)

**lemma** *method-exit-cases*:
⟦*method-exit n*; *n* = (-*Exit*-) ⟹ *P*;
⋀*a Q f p*. ⟦*n* = *sourcenode a*; *valid-edge a*; *kind a* = $Q{\hookleftarrow}_p f$⟧ ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

**lemma** *method-exit-inner-path*:
**assumes** *method-exit n* **and** $n\ {-as\rightarrow_\iota*}\ n'$ **shows** *as* = []
⟨*proof*⟩

### Definition of *inner-node*

**definition** *inner-node* :: *'node* ⇒ *bool*
  **where** *inner-node-def*:
  *inner-node n* ≡ *valid-node n* ∧ *n* ≠ (*-Entry-*) ∧ *n* ≠ (*-Exit-*)

**lemma** *inner-is-valid*:
  *inner-node n* ⟹ *valid-node n*
⟨*proof*⟩

**lemma** [*dest*]:
  *inner-node* (*-Entry-*) ⟹ *False*
⟨*proof*⟩

**lemma** [*dest*]:
  *inner-node* (*-Exit-*) ⟹ *False*
⟨*proof*⟩

**lemma** [*simp*]:⟦*valid-edge a*; *targetnode a* ≠ (*-Exit-*)⟧
  ⟹ *inner-node* (*targetnode a*)
  ⟨*proof*⟩

**lemma** [*simp*]:⟦*valid-edge a*; *sourcenode a* ≠ (*-Entry-*)⟧
  ⟹ *inner-node* (*sourcenode a*)
  ⟨*proof*⟩

**lemma** *valid-node-cases* [*consumes 1*, *case-names Entry Exit inner*]:
  ⟦*valid-node n*; *n* = (*-Entry-*) ⟹ *Q*; *n* = (*-Exit-*) ⟹ *Q*;
    *inner-node n* ⟹ *Q*⟧ ⟹ *Q*
⟨*proof*⟩

### Lemmas on paths with (*-Exit-*)

**lemma** *path-Exit-source*:
  ⟦*n* −*as*→∗ *n'*; *n* = (*-Exit-*)⟧ ⟹ *n'* = (*-Exit-*) ∧ *as* = []
⟨*proof*⟩

**lemma** [*dest*]:(*-Exit-*) −*as*→∗ *n'* ⟹ *n'* = (*-Exit-*) ∧ *as* = []
  ⟨*proof*⟩

**lemma** *Exit-no-sourcenode*[*dest*]:
  **assumes** *isin*:(*-Exit-*) ∈ *set* (*sourcenodes as*) **and** *path*:*n* −*as*→∗ *n'*
  **shows** *False*
⟨*proof*⟩

**lemma** *vpa-no-slpa*:
  ⟦*valid-path-aux cs as*; *n* −*as*→∗ *n'*; *valid-call-list cs n*; *cs* ≠ [];

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

**lemma** *valid-Exit-path-cases*:
**assumes** *n* $-as\rightarrow_{\surd}*$ (*-Exit-*) **and** *as* $\neq$ []
**shows** ($\exists$ *a' as'*. *as* = *a'*#*as'* $\wedge$ *intra-kind*(*kind a'*)) $\vee$
($\exists$ *a' as' Q p f*. *as* = *a'*#*as'* $\wedge$ *kind a'* = *Q*$\hookleftarrow_p$*f*) $\vee$
($\exists$ *as' as'' n'*. *as* = *as'*@*as''* $\wedge$ *as'* $\neq$ [] $\wedge$ *n* $-as'\rightarrow_{sl}*$ *n'*)
⟨*proof*⟩

**lemma** *valid-Exit-path-descending-path*:
**assumes** *n* $-as\rightarrow_{\surd}*$ (*-Exit-*)
**obtains** *as'* **where** *n* $-as'\rightarrow_{\surd}*$ (*-Exit-*)
**and** *set*(*sourcenodes as'*) $\subseteq$ *set*(*sourcenodes as*)
**and** $\forall$ *a'* $\in$ *set as'*. *intra-kind*(*kind a'*) $\vee$ ($\exists$ *Q f p*. *kind a'* = *Q*$\hookleftarrow_p$*f*)
⟨*proof*⟩

**lemma** *valid-Exit-path-intra-path*:
**assumes** *n* $-as\rightarrow_{\surd}*$ (*-Exit-*)
**obtains** *as' pex* **where** *n* $-as'\rightarrow_{\iota}*$ *pex* **and** *method-exit pex*
**and** *set*(*sourcenodes as'*) $\subseteq$ *set*(*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-empty*: $Def\ (\text{-}Entry\text{-}) = \{\} \wedge Use\ (\text{-}Entry\text{-}) = \{\}$
**and** *ParamUses-call-source-length*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow_p}fs;\ (p,ins,outs) \in set\ procs]\!]$
$\Longrightarrow length(ParamUses\ (sourcenode\ a)) = length\ ins$
**and** *distinct-ParamDefs*: $valid\text{-}edge\ a \Longrightarrow distinct\ (ParamDefs\ (targetnode\ a))$
**and** *ParamDefs-return-target-length*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q'{\hookleftarrow_p}f';\ (p,ins,outs) \in set\ procs]\!]$
$\Longrightarrow length(ParamDefs\ (targetnode\ a)) = length\ outs$
**and** *ParamDefs-in-Def*:
$[\![valid\text{-}node\ n;\ V \in set\ (ParamDefs\ n)]\!] \Longrightarrow V \in Def\ n$
**and** *ins-in-Def*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow_p}fs;\ (p,ins,outs) \in set\ procs;\ V \in set\ ins]\!]$
$\Longrightarrow V \in Def\ (targetnode\ a)$
**and** *call-source-Def-empty*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow_p}fs]\!] \Longrightarrow Def\ (sourcenode\ a) = \{\}$
**and** *ParamUses-in-Use*:
$[\![valid\text{-}node\ n;\ V \in Union\ (set\ (ParamUses\ n))]\!] \Longrightarrow V \in Use\ n$
**and** *outs-in-Use*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{\hookleftarrow_p}f;\ (p,ins,outs) \in set\ procs;\ V \in set\ outs]\!]$
$\Longrightarrow V \in Use\ (sourcenode\ a)$
**and** *CFG-intra-edge-no-Def-equal*:
$[\![valid\text{-}edge\ a;\ V \notin Def\ (sourcenode\ a);\ intra\text{-}kind\ (kind\ a);\ pred\ (kind\ a)\ s]\!]$
$\Longrightarrow state\text{-}val\ (transfer\ (kind\ a)\ s)\ V = state\text{-}val\ s\ V$
**and** *CFG-intra-edge-transfer-uses-only-Use*:
$[\![valid\text{-}edge\ a;\ \forall\ V \in Use\ (sourcenode\ a).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V;$
$intra\text{-}kind\ (kind\ a);\ pred\ (kind\ a)\ s;\ pred\ (kind\ a)\ s']\!]$
$\Longrightarrow \forall\ V \in Def\ (sourcenode\ a).\ state\text{-}val\ (transfer\ (kind\ a)\ s)\ V =$
$state\text{-}val\ (transfer\ (kind\ a)\ s')\ V$
**and** *CFG-edge-Uses-pred-equal*:
$[\![valid\text{-}edge\ a;\ pred\ (kind\ a)\ s;\ snd\ (hd\ s) = snd\ (hd\ s');$
$\forall\ V \in Use\ (sourcenode\ a).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V;\ length\ s = length\ s']\!]$
$\Longrightarrow pred\ (kind\ a)\ s'$
**and** *CFG-call-edge-length*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow_p}fs;\ (p,ins,outs) \in set\ procs]\!]$
$\Longrightarrow length\ fs = length\ ins$
**and** *CFG-call-determ*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow_p}fs;\ valid\text{-}edge\ a';\ kind\ a' = Q'{:}r'{\hookrightarrow_{p'}}fs';$
$sourcenode\ a = sourcenode\ a';\ pred\ (kind\ a)\ s;\ pred\ (kind\ a')\ s]\!]$
$\Longrightarrow a = a'$
**and** *CFG-call-edge-params*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow_p}fs;\ i < length\ ins;$
$(p,ins,outs) \in set\ procs;\ pred\ (kind\ a)\ s;\ pred\ (kind\ a)\ s';$
$\forall\ V \in (ParamUses\ (sourcenode\ a))!i.\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V]\!]$
$\Longrightarrow (params\ fs\ (fst\ (hd\ s)))!i = (params\ fs\ (fst\ (hd\ s')))!i$
**and** *CFG-return-edge-fun*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q'{\hookleftarrow_p}f';\ (p,ins,outs) \in set\ procs]\!]$
$\Longrightarrow f'\ vmap\ vmap' = vmap'(ParamDefs\ (targetnode\ a)\ [:=]\ map\ vmap\ outs)$
**and** *deterministic*: $[\![valid\text{-}edge\ a;\ valid\text{-}edge\ a';\ sourcenode\ a = sourcenode\ a';$
$targetnode\ a \neq targetnode\ a';\ intra\text{-}kind\ (kind\ a);\ intra\text{-}kind\ (kind\ a')]\!]$

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

**begin**

**lemma** *CFG-equal-Use-equal-call*:
**assumes** *valid-edge a* **and** $kind\ a = Q{:}r\hookrightarrow_{p}fs$ **and** *valid-edge a′*
**and** $kind\ a' = Q'{:}r'\hookrightarrow_{p'}fs'$ **and** *sourcenode a = sourcenode a′*
**and** *pred (kind a) s* **and** *pred (kind a′) s′*
**and** *snd (hd s) = snd (hd s′)* **and** *length s = length s′*
**and** $\forall\, V \in Use\ (sourcenode\ a).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$
**shows** $a = a'$
⟨*proof*⟩

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

**lemma** *CFG-call-edge-no-param*:
**assumes** *valid-edge a* **and** $kind\ a = Q{:}r\hookrightarrow_{p}fs$ **and** $V \notin set\ ins$
**and** $(p,ins,outs) \in set\ procs$ **and** *pred (kind a) s*
**shows** *state-val (transfer (kind a) s) V = None*
⟨*proof*⟩

**lemma** *CFG-return-edge-param-out*:
**assumes** *valid-edge a* **and** $kind\ a = Q\hookleftarrow_{p}f$ **and** $i < length\ outs$
**and** $(p,ins,outs) \in set\ procs$ **and** *state-val s (outs!i) = state-val s′ (outs!i)*
**and** *s = cf#cfx#cfs* **and** *s′ = cf′#cfx′#cfs′*
**shows** *state-val (transfer (kind a) s) ((ParamDefs (targetnode a))!i) =*
*state-val (transfer (kind a) s′) ((ParamDefs (targetnode a))!i)*
⟨*proof*⟩

**lemma** *CFG-slp-no-Def-equal*:
**assumes** $n\ {-as\rightarrow_{sl}*}\ n'$ **and** *valid-edge a* **and** $a' \in get\text{-}return\text{-}edges\ a$
**and** $V \notin set\ (ParamDefs\ (targetnode\ a'))$ **and** *preds (kinds (a#as@[a′])) s*
**shows** *state-val (transfers (kinds (a#as@[a′])) s) V = state-val s V*
⟨*proof*⟩

**lemma** [*dest*!]: $V \in Use$ (*-Entry-*) $\Longrightarrow$ *False*
⟨*proof*⟩

**lemma** [*dest*!]: $V \in Def$ (*-Entry-*) $\Longrightarrow$ *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*⟧
$\Longrightarrow$ *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* ⇒ *'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* ⇒ *'pname*
**and** *get-return-edges* :: *'edge* ⇒ *'edge set*
**and** *procs* :: (*'pname* × *'var list* × *'var list*) *list* **and** *Main* :: *'pname*
**and** *Exit*::*'node* (‹'('-Exit'-')›)
**and** *Def* :: *'node* ⇒ *'var set* **and** *Use* :: *'node* ⇒ *'var set*
**and** *ParamDefs* :: *'node* ⇒ *'var list*

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

**begin**

**lemma** *Exit-Use-empty* [*dest*!]: $V \in Use\ (\text{-}Exit\text{-}) \Longrightarrow False$
⟨*proof*⟩

**lemma** *Exit-Def-empty* [*dest*!]: $V \in Def\ (\text{-}Exit\text{-}) \Longrightarrow False$
⟨*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* :: $'edge \Rightarrow ('var,'val,'ret,'pname)\ edge\text{-}kind$
**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*::$'com \Rightarrow ('var \rightharpoonup 'val)\ list \Rightarrow 'com \Rightarrow ('var \rightharpoonup 'val)\ list \Rightarrow bool$
(‹((1⟨-,/-⟩) ⇒/ (1⟨-,/-⟩))› [0,0,0,0] 81)
**fixes** *identifies*::$'node \Rightarrow 'com \Rightarrow bool$ (‹- ≜ -› [51,0] 80)
**assumes** *fundamental-property*:
$[\![n \triangleq c;\ \langle c,[cf]\rangle \Rightarrow \langle c',s'\rangle]\!] \Longrightarrow$
$\exists n'\ as.\ n\ {-as\rightarrow_{\surd}*}\ n' \wedge n' \triangleq c' \wedge preds\ (kinds\ as)\ [(cf,undefined)] \wedge$
$transfers\ (kinds\ as)\ [(cf,undefined)] = cfs' \wedge map\ fst\ cfs' = s'$

**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** $return\text{-}node\ n \equiv \exists a\ a'.\ valid\text{-}edge\ a \wedge n = targetnode\ a \wedge$

*valid-edge a′* ∧ *a* ∈ *get-return-edges a′*

**lemma** *return-node-determines-call-node*:
**assumes** *return-node n*
**shows** $\exists!n'.\ \exists\, a\ a'.\ valid\text{-}edge\ a \wedge n' = sourcenode\ a \wedge valid\text{-}edge\ a' \wedge a' \in get\text{-}return\text{-}edges\ a \wedge n = targetnode\ a'$
⟨*proof*⟩

**lemma** *return-node-THE-call-node*:
$[\![return\text{-}node\ n;\ valid\text{-}edge\ a;\ valid\text{-}edge\ a';\ a' \in get\text{-}return\text{-}edges\ a;\ n = targetnode\ a']\!]$
$\Longrightarrow (THE\ n'.\ \exists\, a\ a'.\ valid\text{-}edge\ a \wedge n' = sourcenode\ a \wedge valid\text{-}edge\ a' \wedge a' \in get\text{-}return\text{-}edges\ a \wedge n = targetnode\ a') = sourcenode\ a$
⟨*proof*⟩

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

**definition** *call-of-return-node* :: *′node* ⇒ *′node* ⇒ *bool*
**where** $call\text{-}of\text{-}return\text{-}node\ n\ n' \equiv \exists\, a\ a'.\ return\text{-}node\ n \wedge valid\text{-}edge\ a \wedge n' = sourcenode\ a \wedge valid\text{-}edge\ a' \wedge a' \in get\text{-}return\text{-}edges\ a \wedge n = targetnode\ a'$

**lemma** *return-node-call-of-return-node*:
$return\text{-}node\ n \Longrightarrow \exists!n'.\ call\text{-}of\text{-}return\text{-}node\ n\ n'$
⟨*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*:
$[\![valid\text{-}call\text{-}list\ cs\ m;\ valid\text{-}return\text{-}list\ rs\ m;\ \forall\, i < length\ rs.\ rs!i \in get\text{-}return\text{-}edges\ (cs!i);\ length\ rs = length\ cs]\!]$
$\Longrightarrow \forall\, i{<}length\ cs.\ call\text{-}of\text{-}return\text{-}node\ (targetnodes\ rs!i)\ (sourcenode\ (cs!i))$
⟨*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*:
$[\![n\ -as\rightarrow_\iota*\ n'; \forall\, nx \in set(sourcenodes\ as).\ nx \notin S; n' \in S]\!] \Longrightarrow n' \in obs\text{-}intra\ n\ S$

**lemma** *obs-intraE*:
**assumes** $n' \in obs\text{-}intra\ n\ S$
**obtains** *as* **where** $n\ -as\rightarrow_\iota*\ n'$ **and** $\forall\, nx \in set(sourcenodes\ as).\ nx \notin S$ **and** $n' \in S$
⟨*proof*⟩

**lemma** *n-in-obs-intra*:
**assumes** *valid-node* $n$ **and** $n \in S$ **shows** $obs\text{-}intra\ n\ S = \{n\}$
⟨*proof*⟩

**lemma** *in-obs-intra-valid*:
**assumes** $n' \in obs\text{-}intra\ n\ S$ **shows** *valid-node* $n$ **and** *valid-node* $n'$
⟨*proof*⟩

**lemma** *edge-obs-intra-subset*:
**assumes** *valid-edge* $a$ **and** *intra-kind* (*kind* $a$) **and** *sourcenode* $a \notin S$
**shows** *obs-intra* (*targetnode* $a$) $S \subseteq$ *obs-intra* (*sourcenode* $a$) $S$
⟨*proof*⟩

**lemma** *path-obs-intra-subset*:
**assumes** $n\ -as\rightarrow_\iota*\ n'$ **and** $\forall\, n' \in set(sourcenodes\ as).\ n' \notin S$
**shows** $obs\text{-}intra\ n'\ S \subseteq obs\text{-}intra\ n\ S$
⟨*proof*⟩

**lemma** *path-ex-obs-intra*:
**assumes** $n\ -as\rightarrow_\iota*\ n'$ **and** $n' \in S$
**obtains** $m$ **where** $m \in obs\text{-}intra\ n\ S$
⟨*proof*⟩

### 1.6.2 Interprocedural observable sets restricted to the slice

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

**lemma** *obsI*:
  **assumes** $n' \in obs\text{-}intra\ n\ S$
  **and** $\forall\, nx \in set\ nsx'.\ \exists\, nx'.\ call\text{-}of\text{-}return\text{-}node\ nx\ nx' \wedge nx' \in S$
  **shows** $[\![ns = nsx@n\#nsx'; \forall\, xs\ x\ xs'.\ nsx = xs@x\#xs' \wedge obs\text{-}intra\ x\ S \neq \{\}$
  $\longrightarrow (\exists\, x'' \in set\ (xs'@[n]).\ \exists\, nx.\ call\text{-}of\text{-}return\text{-}node\ x''\ nx \wedge nx \notin S)]\!]$
  $\Longrightarrow n'\#nsx' \in obs\ ns\ S$
⟨*proof*⟩

**lemma** *obsE* [*consumes 2*]:
  **assumes** $ns' \in obs\ ns\ S$ **and** $\forall\, n \in set\ (tl\ ns).\ return\text{-}node\ n$
  **obtains** $nsx\ n\ nsx'\ n'$ **where** $ns = nsx@n\#nsx'$ **and** $ns' = n'\#nsx'$
  **and** $n' \in obs\text{-}intra\ n\ S$
  **and** $\forall\, nx \in set\ nsx'.\ \exists\, nx'.\ call\text{-}of\text{-}return\text{-}node\ nx\ nx' \wedge nx' \in S$
  **and** $\forall\, xs\ x\ xs'.\ nsx = xs@x\#xs' \wedge obs\text{-}intra\ x\ S \neq \{\}$
  $\longrightarrow (\exists\, x'' \in set\ (xs'@[n]).\ \exists\, nx.\ call\text{-}of\text{-}return\text{-}node\ x''\ nx \wedge nx \notin S)$
⟨*proof*⟩

**lemma** *obs-split-det*:
  **assumes** $xs@x\#xs' = ys@y\#ys'$
  **and** $obs\text{-}intra\ x\ S \neq \{\}$
  **and** $\forall\, x' \in set\ xs'.\ \exists\, x''.\ call\text{-}of\text{-}return\text{-}node\ x'\ x'' \wedge x'' \in S$
  **and** $\forall\, zs\ z\ zs'.\ xs = zs@z\#zs' \wedge obs\text{-}intra\ z\ S \neq \{\}$
  $\longrightarrow (\exists\, z'' \in set\ (zs'@[x]).\ \exists\, nx.\ call\text{-}of\text{-}return\text{-}node\ z''\ nx \wedge nx \notin S)$
  **and** $obs\text{-}intra\ y\ S \neq \{\}$
  **and** $\forall\, y' \in set\ ys'.\ \exists\, y''.\ call\text{-}of\text{-}return\text{-}node\ y'\ y'' \wedge y'' \in S$
  **and** $\forall\, zs\ z\ zs'.\ ys = zs@z\#zs' \wedge obs\text{-}intra\ z\ S \neq \{\}$
  $\longrightarrow (\exists\, z'' \in set\ (zs'@[y]).\ \exists\, ny.\ call\text{-}of\text{-}return\text{-}node\ z''\ ny \wedge ny \notin S)$
  **shows** $xs = ys \wedge x = y \wedge xs' = ys'$
⟨*proof*⟩

**lemma** *in-obs-valid*:
  **assumes** $ns' \in obs\ ns\ S$ **and** $\forall\, n \in set\ ns.\ valid\text{-}node\ n$
  **shows** $\forall\, n \in set\ ns'.\ valid\text{-}node\ n$
  ⟨*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* ⇒ *'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* ⇒ *'pname*
**and** *get-return-edges* :: *'edge* ⇒ *'edge set*
**and** *procs* :: (*'pname* × *'var list* × *'var list*) *list* **and** *Main* :: *'pname*
**and** *Exit*::*'node* (‹'('-Exit'-')›) +
**assumes** *Entry-path*:*valid-node n* ⟹ ∃ *as*. (-*Entry*-) $-as\rightarrow_{\surd}*$ *n*
**and** *Exit-path*:*valid-node n* ⟹ ∃ *as*. *n* $-as\rightarrow_{\surd}*$ (-*Exit*-)
**and** *method-exit-unique*:
⟦*method-exit n*; *method-exit n'*; *get-proc n* = *get-proc n'*⟧ ⟹ *n* = *n'*

**begin**

**lemma** *get-return-edges-unique*:
**assumes** *valid-edge a* **and** *a'* ∈ *get-return-edges a* **and** *a''* ∈ *get-return-edges a*
**shows** *a'* = *a''*
⟨*proof*⟩

**definition** *postdominate* :: *'node* ⇒ *'node* ⇒ *bool* (‹- *postdominates* -› [*51*,*0*])
**where** *postdominate-def*:*n' postdominates n* ≡
(*valid-node n* ∧ *valid-node n'* ∧
(∀ *as pex*. (*n* $-as\rightarrow_{\iota}*$ *pex* ∧ *method-exit pex*) ⟶ *n'* ∈ *set* (*sourcenodes as*)))

**lemma** *postdominate-implies-inner-path*:
**assumes** *n' postdominates n*
**obtains** *as* **where** *n* $-as\rightarrow_{\iota}*$ *n'* **and** *n'* ∉ *set* (*sourcenodes as*)
⟨*proof*⟩

**lemma** *postdominate-variant*:
**assumes** *n' postdominates n*
**shows** ∀ *as*. *n* $-as\rightarrow_{\surd}*$ (-*Exit*-) ⟶ *n'* ∈ *set* (*sourcenodes as*)
⟨*proof*⟩

**lemma** *postdominate-refl*:
**assumes** *valid-node n* **and** ¬ *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** ¬ $n'$ *postdominates n*
**obtains** *a* $as'$ $as''$ **where** $as = as'@a\#as''$ **and** *valid-edge a*
**and** ¬ $n'$ *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** $n$ $-as\rightarrow_\iota*$ $n''$ **and** $n' \notin$ *set*(*sourcenodes as*)
**shows** $n'$ *postdominates* $n''$
⟨*proof*⟩

**lemma** *not-postdominate-source-not-postdominate-target*:
**assumes** ¬ *n postdominates* (*sourcenode a*)
**and** *valid-node n* **and** *valid-edge a* **and** *intra-kind* (*kind a*)
**obtains** *ax* **where** *sourcenode a* = *sourcenode ax* **and** *valid-edge ax*
**and** ¬ *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** $n\ -as\rightarrow_\iota *\ n'$
⟨*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* ⇒ *'pname*
**and** *get-return-edges* :: *'edge* ⇒ *'edge set*
**and** *procs* :: (*'pname* × *'var list* × *'var list*) *list* **and** *Main* :: *'pname*
**and** *Exit*::*'node* (‹'('-Exit'-')›)
**and** *Def* :: *'node* ⇒ *'var set* **and** *Use* :: *'node* ⇒ *'var set*
**and** *ParamDefs* :: *'node* ⇒ *'var list* **and** *ParamUses* :: *'node* ⇒ *'var set list*

**begin**

**fun** *valid-SDG-node* :: *'node SDG-node* ⇒ *bool*
**where** *valid-SDG-node* (*CFG-node n*) ⟷ *valid-node n*
| *valid-SDG-node* (*Formal-in* (*m*,*x*)) ⟷
(∃ *a Q r p fs ins outs. valid-edge a* ∧ (*kind a* = $Q{:}r\hookrightarrow_{p}fs$) ∧ *targetnode a* = *m* ∧
(*p*,*ins*,*outs*) ∈ *set procs* ∧ *x* < *length ins*)
| *valid-SDG-node* (*Formal-out* (*m*,*x*)) ⟷
(∃ *a Q p f ins outs. valid-edge a* ∧ (*kind a* = $Q\hookleftarrow_{p}f$) ∧ *sourcenode a* = *m* ∧
(*p*,*ins*,*outs*) ∈ *set procs* ∧ *x* < *length outs*)
| *valid-SDG-node* (*Actual-in* (*m*,*x*)) ⟷
(∃ *a Q r p fs ins outs. valid-edge a* ∧ (*kind a* = $Q{:}r\hookrightarrow_{p}fs$) ∧ *sourcenode a* = *m*
∧
(*p*,*ins*,*outs*) ∈ *set procs* ∧ *x* < *length ins*)
| *valid-SDG-node* (*Actual-out* (*m*,*x*)) ⟷
(∃ *a Q p f ins outs. valid-edge a* ∧ (*kind a* = $Q\hookleftarrow_{p}f$) ∧ *targetnode a* = *m* ∧
(*p*,*ins*,*outs*) ∈ *set procs* ∧ *x* < *length outs*)

**lemma** *valid-SDG-CFG-node*:
*valid-SDG-node n* ⟹ *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-*)

⟨*proof*⟩

### 1.8.2 Data dependence

**inductive** *SDG-Use* :: *'var* ⇒ *'node SDG-node* ⇒ *bool* (‹- ∈ $Use_{SDG}$ -›)
**where** *CFG-Use-SDG-Use*:
⟦*valid-node m*; *V* ∈ *Use m*; *n* = *CFG-node m*⟧ ⟹ *V* ∈ $Use_{SDG}$ *n*
| *Actual-in-SDG-Use*:
⟦*valid-SDG-node n*; *n* = *Actual-in* (*m*,*x*); *V* ∈ (*ParamUses m*)!*x*⟧ ⟹ *V* ∈ $Use_{SDG}$ *n*
| *Formal-out-SDG-Use*:
⟦*valid-SDG-node n*; *n* = *Formal-out* (*m*,*x*); *get-proc m* = *p*; (*p*,*ins*,*outs*) ∈ *set procs*;
*V* = *outs*!*x*⟧ ⟹ *V* ∈ $Use_{SDG}$ *n*

**abbreviation** *notin-SDG-Use* :: *'var* ⇒ *'node SDG-node* ⇒ *bool* (‹- ∉ $Use_{SDG}$ -›)
**where** *V* ∉ $Use_{SDG}$ *n* ≡ ¬ *V* ∈ $Use_{SDG}$ *n*

**lemma** *in-Use-valid-SDG-node*:
*V* ∈ $Use_{SDG}$ *n* ⟹ *valid-SDG-node n*
⟨*proof*⟩

**lemma** *SDG-Use-parent-Use*:
*V* ∈ $Use_{SDG}$ *n* ⟹ *V* ∈ *Use* (*parent-node n*)
⟨*proof*⟩

**inductive** *SDG-Def* :: *'var* ⇒ *'node SDG-node* ⇒ *bool* (‹- ∈ $Def_{SDG}$ -›)
**where** *CFG-Def-SDG-Def*:
⟦*valid-node m*; *V* ∈ *Def m*; *n* = *CFG-node m*⟧ ⟹ *V* ∈ $Def_{SDG}$ *n*
| *Formal-in-SDG-Def*:
⟦*valid-SDG-node n*; *n* = *Formal-in* (*m*,*x*); *get-proc m* = *p*; (*p*,*ins*,*outs*) ∈ *set procs*;
*V* = *ins*!*x*⟧ ⟹ *V* ∈ $Def_{SDG}$ *n*
| *Actual-out-SDG-Def*:
⟦*valid-SDG-node n*; *n* = *Actual-out* (*m*,*x*); *V* = (*ParamDefs m*)!*x*⟧ ⟹ *V* ∈ $Def_{SDG}$ *n*

**abbreviation** *notin-SDG-Def* :: *'var* ⇒ *'node SDG-node* ⇒ *bool* (‹- ∉ $Def_{SDG}$ -›)
**where** *V* ∉ $Def_{SDG}$ *n* ≡ ¬ *V* ∈ $Def_{SDG}$ *n*

**lemma** *in-Def-valid-SDG-node*:

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

**lemma** *SDG-Def-parent-Def*:
$V \in Def_{SDG}\ n \Longrightarrow V \in Def\ (parent\text{-}node\ n)$
⟨proof⟩

**definition** *data-dependence* :: $'node\ SDG\text{-}node \Rightarrow 'var \Rightarrow 'node\ SDG\text{-}node \Rightarrow bool$

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

### 1.8.3 Control dependence

**definition** *control-dependence* :: $'node \Rightarrow 'node \Rightarrow bool$
(‹- *controls* -› [51,0])
**where** $n\ controls\ n' \equiv \exists\, a\ a'\ as.\ n\ -a\#as\rightarrow_\iota *\ n' \wedge n' \notin set(sourcenodes\ (a\#as))$
$\wedge$
$intra\text{-}kind(kind\ a) \wedge n'\ postdominates\ (targetnode\ a) \wedge$
$valid\text{-}edge\ a' \wedge intra\text{-}kind(kind\ a') \wedge sourcenode\ a' = n \wedge$
$\neg\ n'\ postdominates\ (targetnode\ a')$

**lemma** *control-dependence-path*:
**assumes** $n\ controls\ n'$ **obtains** $as$ **where** $n\ -as\rightarrow_\iota *\ n'$ **and** $as \neq []$
⟨proof⟩

**lemma** *Exit-does-not-control* [*dest*]:
**assumes** (-*Exit*-) $controls\ n'$ **shows** *False*
⟨proof⟩

**lemma** *Exit-not-control-dependent*:
**assumes** $n\ controls\ n'$ **shows** $n' \neq$ (-*Exit*-)
⟨proof⟩

**lemma** *which-node-intra-standard-control-dependence-source*:
**assumes** $nx\ -as@a\#as'\rightarrow_\iota *\ n$ **and** $sourcenode\ a = n'$ **and** $sourcenode\ a' = n'$
**and** $n \notin set(sourcenodes\ (a\#as'))$ **and** $valid\text{-}edge\ a'$ **and** $intra\text{-}kind(kind\ a')$
**and** $inner\text{-}node\ n$ **and** $\neg\ method\text{-}exit\ n$ **and** $\neg\ n\ postdominates\ (targetnode\ a')$
**and** $last{:}\forall\, ax\ ax'.\ ax \in set\ as' \wedge sourcenode\ ax = sourcenode\ ax' \wedge$

*valid-edge ax′* ∧ *intra-kind*(*kind ax′*) ⟶ *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*
(‹- $\longrightarrow_{cd}$ -› [*51*,*0*] *80*)
**and** *ddep-edge* :: *′node SDG-node* ⇒ *′var* ⇒ *′node SDG-node* ⇒ *bool*
(‹- $-\text{-}\!\!\rightarrow_{dd}$ -› [*51*,*0*,*0*] *80*)
**and** *call-edge* :: *′node SDG-node* ⇒ *′pname* ⇒ *′node SDG-node* ⇒ *bool*
(‹- $-\text{-}\!\!\rightarrow_{call}$ -› [*51*,*0*,*0*] *80*)
**and** *return-edge* :: *′node SDG-node* ⇒ *′pname* ⇒ *′node SDG-node* ⇒ *bool*
(‹- $-\text{-}\!\!\rightarrow_{ret}$ -› [*51*,*0*,*0*] *80*)
**and** *param-in-edge* :: *′node SDG-node* ⇒ *′pname* ⇒ *′var* ⇒ *′node SDG-node* ⇒ *bool*
(‹- $-\text{-:-}\!\!\rightarrow_{in}$ -› [*51*,*0*,*0*,*0*] *80*)
**and** *param-out-edge* :: *′node SDG-node* ⇒ *′pname* ⇒ *′var* ⇒ *′node SDG-node* ⇒ *bool*
(‹- $-\text{-:-}\!\!\rightarrow_{out}$ -› [*51*,*0*,*0*,*0*] *80*)
**and** *SDG-edge* :: *′node SDG-node* ⇒ *′var option* ⇒
(*′pname* × *bool*) *option* ⇒ *′node SDG-node* ⇒ *bool*

**where**

*n* $\longrightarrow_{cd}$ *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*:
⟦*n* = *CFG-node m*; *n′* = *CFG-node m′*; *m controls m′*⟧ ⟹ *n* $\longrightarrow_{cd}$ *n′*
| *SDG-proc-entry-exit-cdep*:
⟦*valid-edge a*; *kind a* = $Q{:}r\hookrightarrow_{p}fs$; *n* = *CFG-node* (*targetnode a*);
*a′* ∈ *get-return-edges a*; *n′* = *CFG-node* (*sourcenode a′*)⟧ ⟹ *n* $\longrightarrow_{cd}$ *n′*
| *SDG-parent-cdep-edge*:
⟦*valid-SDG-node n′*; *m* = *parent-node n′*; *n* = *CFG-node m*; *n* ≠ *n′*⟧
⟹ *n* $\longrightarrow_{cd}$ *n′*
| *SDG-ddep-edge*:*n influences V in n′* ⟹ *n* $-V\rightarrow_{dd}$ *n′*
| *SDG-call-edge*:
⟦*valid-edge a*; *kind a* = $Q{:}r\hookrightarrow_{p}fs$; *n* = *CFG-node* (*sourcenode a*);
*n′* = *CFG-node* (*targetnode a*)⟧ ⟹ *n* $-p\rightarrow_{call}$ *n′*
| *SDG-return-edge*:
⟦*valid-edge a*; *kind a* = $Q\hookleftarrow_{p}f$; *n* = *CFG-node* (*sourcenode a*);
*n′* = *CFG-node* (*targetnode a*)⟧ ⟹ *n* $-p\rightarrow_{ret}$ *n′*
| *SDG-param-in-edge*:

$\llbracket$*valid-edge a*; *kind a* = $Q{:}r{\hookrightarrow}_{p}fs$; (*p*,*ins*,*outs*) $\in$ *set procs*; $V$ = *ins*!*x*;
$x$ < *length ins*; $n$ = *Actual-in* (*sourcenode a*,*x*); $n'$ = *Formal-in* (*targetnode a*,*x*)$\rrbracket$
$\Longrightarrow n \; {-}p{:}V{\rightarrow}_{in} \; n'$
| *SDG-param-out-edge*:
$\llbracket$*valid-edge a*; *kind a* = $Q{\hookleftarrow}_{p}f$; (*p*,*ins*,*outs*) $\in$ *set procs*; $V$ = *outs*!*x*;
$x$ < *length outs*; $n$ = *Formal-out* (*sourcenode a*,*x*);
$n'$ = *Actual-out* (*targetnode a*,*x*)$\rrbracket$
$\Longrightarrow n \; {-}p{:}V{\rightarrow}_{out} \; n'$

**lemma** *cdep-edge-cases*:
$\llbracket n \longrightarrow_{cd} n'$; (*parent-node n*) *controls* (*parent-node n'*) $\Longrightarrow P$;
$\bigwedge$*a Q r p fs a'*. $\llbracket$*valid-edge a*; *kind a* = $Q{:}r{\hookrightarrow}_{p}fs$; *a'* $\in$ *get-return-edges a*;
*parent-node n* = *targetnode a*; *parent-node n'* = *sourcenode a'*$\rrbracket \Longrightarrow P$;
$\bigwedge$*m*. $\llbracket n$ = *CFG-node m*; *m* = *parent-node n'*; $n \neq n'\rrbracket \Longrightarrow P\rrbracket \Longrightarrow P$
⟨*proof*⟩

**lemma** *SDG-edge-valid-SDG-node*:
**assumes** *SDG-edge n Vopt popt n'*
**shows** *valid-SDG-node n* **and** *valid-SDG-node n'*
⟨*proof*⟩

**lemma** *valid-SDG-node-cases*:
**assumes** *valid-SDG-node n*
**shows** $n$ = *CFG-node* (*parent-node n*) $\vee$ *CFG-node* (*parent-node n*) $\longrightarrow_{cd} n$
⟨*proof*⟩

**lemma** *SDG-cdep-edge-CFG-node*: $n \longrightarrow_{cd} n' \Longrightarrow \exists\, m.\; n$ = *CFG-node m*
⟨*proof*⟩

**lemma** *SDG-call-edge-CFG-node*: $n \; {-}p{\rightarrow}_{call} \; n' \Longrightarrow \exists\, m.\; n$ = *CFG-node m*
⟨*proof*⟩

**lemma** *SDG-return-edge-CFG-node*: $n \; {-}p{\rightarrow}_{ret} \; n' \Longrightarrow \exists\, m.\; n$ = *CFG-node m*
⟨*proof*⟩

**lemma** *SDG-call-or-param-in-edge-unique-CFG-call-edge*:
*SDG-edge n Vopt* (*Some*(*p*, *True*)) *n'*
$\Longrightarrow \exists !a.$ *valid-edge a* $\wedge$ *sourcenode a* = *parent-node n* $\wedge$
*targetnode a* = *parent-node n'* $\wedge$ ($\exists$ *Q r fs*. *kind a* = $Q{:}r{\hookrightarrow}_{p}fs$)
⟨*proof*⟩

**lemma** *SDG-return-or-param-out-edge-unique-CFG-return-edge*:
*SDG-edge n Vopt (Some(p,False)) n'*
$\Longrightarrow \exists !a.$ *valid-edge a* $\wedge$ *sourcenode a = parent-node n* $\wedge$
*targetnode a = parent-node n'* $\wedge$ ($\exists$ *Q f. kind a = Q* $\hookleftarrow_p f$)
⟨*proof*⟩

**lemma** *Exit-no-SDG-edge-source*:
*SDG-edge (CFG-node (-Exit-)) Vopt popt n'* $\Longrightarrow$ *False*
⟨*proof*⟩

### 1.8.5 Intraprocedural paths in the SDG

**inductive** *intra-SDG-path* ::
*'node SDG-node* $\Rightarrow$ *'node SDG-node list* $\Rightarrow$ *'node SDG-node* $\Rightarrow$ *bool*
(‹- i–-→$_d$∗ -› [51,0,0] 80)

**where** *iSp-Nil*:
*valid-SDG-node n* $\Longrightarrow$ *n i–[]→$_d$∗ n*

| *iSp-Append-cdep*:
⟦*n i–ns→$_d$∗ n''*; *n''* $\longrightarrow_{cd}$ *n'*⟧ $\Longrightarrow$ *n i–ns@[n'']→$_d$∗ n'*

| *iSp-Append-ddep*:
⟦*n i–ns→$_d$∗ n''*; *n'' –V→$_{dd}$ n'*; *n''* $\neq$ *n'*⟧ $\Longrightarrow$ *n i–ns@[n'']→$_d$∗ n'*

**lemma** *intra-SDG-path-Append*:
⟦*n'' i–ns'→$_d$∗ n'*; *n i–ns→$_d$∗ n''*⟧ $\Longrightarrow$ *n i–ns@ns'→$_d$∗ n'*
⟨*proof*⟩

**lemma** *intra-SDG-path-valid-SDG-node*:
**assumes** *n i–ns→$_d$∗ n'* **shows** *valid-SDG-node n* **and** *valid-SDG-node n'*
⟨*proof*⟩

**lemma** *intra-SDG-path-intra-CFG-path*:
**assumes** *n i–ns→$_d$∗ n'*
**obtains** *as* **where** *parent-node n –as→$_\iota$∗ parent-node n'*
⟨*proof*⟩

### 1.8.6 Control dependence paths in the SDG

**inductive** *cdep-SDG-path* ::
*'node SDG-node* $\Rightarrow$ *'node SDG-node list* $\Rightarrow$ *'node SDG-node* $\Rightarrow$ *bool*
(‹- cd–-→$_d$∗ -› [51,0,0] 80)

**where** *cdSp-Nil*:

*valid-SDG-node n* $\Longrightarrow$ *n cd*$-[]\rightarrow_d*$ *n*

| *cdSp-Append-cdep*:
$[\![n\ cd{-}ns{\rightarrow_d}*\ n''; n'' \longrightarrow_{cd} n']\!] \Longrightarrow n\ cd{-}ns@[n'']{\rightarrow_d}*\ n'$

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

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

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

### 1.8.7 Paths consisting of calls and control dependences

**inductive** *call-cdep-SDG-path* ::
*'node SDG-node* $\Rightarrow$ *'node SDG-node list* $\Rightarrow$ *'node SDG-node* $\Rightarrow$ *bool*
(‹- *cc*−-$\rightarrow_d*$ -› [*51*,*0*,*0*] *80*)
**where** *ccSp-Nil*:
*valid-SDG-node n* $\Longrightarrow$ *n cc*$-[]\rightarrow_d*$ *n*

| *ccSp-Append-cdep*:
$[\![n\ cc{-}ns{\rightarrow_d}*\ n''; n'' \longrightarrow_{cd} n']\!] \Longrightarrow n\ cc{-}ns@[n'']{\rightarrow_d}*\ n'$

| *ccSp-Append-call*:
$[\![n\ cc{-}ns{\rightarrow_d}*\ n''; n''\ {-}p{\rightarrow_{call}}\ n']\!] \Longrightarrow n\ cc{-}ns@[n'']{\rightarrow_d}*\ n'$

**lemma** *cc-SDG-path-Append*:
$[\![n''\ cc{-}ns'{\rightarrow_d}*\ n'; n\ cc{-}ns{\rightarrow_d}*\ n'']\!] \Longrightarrow n\ cc{-}ns@ns'{\rightarrow_d}*\ n'$
⟨*proof*⟩

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

### 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*:
⟦*matched n ns n''*; *n'' i*$-$*ns'*$\rightarrow_d *$ *n'*⟧ $\Longrightarrow$ *matched n* (*ns*@*ns'*) *n'*
| *matched-bracket-call*:
⟦*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* $n_4$⟧
$\Longrightarrow$ *matched* $n_0$ (*ns*@$n_1$#*ns'*@[$n_3$]) $n_4$
| *matched-bracket-param*:
⟦*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* $n_4$⟧
$\Longrightarrow$ *matched* $n_0$ (*ns*@$n_1$#*ns'*@[$n_3$]) $n_4$

**lemma** *matched-Append*:
⟦*matched n'' ns' n'*; *matched n ns n''*⟧ $\Longrightarrow$ *matched n* (*ns*@*ns'*) *n'*
⟨*proof*⟩

**lemma** *intra-SDG-path-matched*:
**assumes** *n i*$-$*ns*$\rightarrow_d *$ *n'* **shows** *matched n ns n'*
⟨*proof*⟩

**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'*))
⟨*proof*⟩

**lemma** *matched-intra-CFG-path*:
**assumes** *matched n ns n'*

**obtains** *as* **where** *parent-node n* $-as\rightarrow_\iota*$ *parent-node n'*
⟨*proof*⟩

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

### 1.8.9 Realizable paths in the SDG

**inductive** *realizable* ::
*'node SDG-node* ⇒ *'node SDG-node list* ⇒ *'node SDG-node* ⇒ *bool*
**where** *realizable-matched*:*matched n ns n'* ⟹ *realizable n ns n'*
| *realizable-call*:
⟦*realizable* $n_0$ *ns* $n_1$; $n_1$ $-p\rightarrow_{call}$ $n_2$ ∨ $n_1$ $-p{:}V\rightarrow_{in}$ $n_2$; *matched* $n_2$ *ns'* $n_3$⟧
⟹ *realizable* $n_0$ (*ns*@$n_1$#*ns'*) $n_3$

**lemma** *realizable-Append-matched*:
⟦*realizable n ns n''*; *matched n'' ns' n'*⟧ ⟹ *realizable n* (*ns*@*ns'*) *n'*
⟨*proof*⟩

**lemma** *realizable-valid-CFG-path*:
**assumes** *realizable n ns n'*
**obtains** *as* **where** *parent-node n* $-as\rightarrow_{\surd}*$ *parent-node n'*
⟨*proof*⟩

**lemma** *cdep-SDG-path-realizable*:
*n cc*$-ns\rightarrow_d*$ *n'* ⟹ *realizable n ns n'*
⟨*proof*⟩

### 1.8.10 SDG with summary edges

**inductive** *sum-cdep-edge* :: *'node SDG-node* ⇒ *'node SDG-node* ⇒ *bool*
(‹- *s*⟶$_{cd}$ -› [*51*,*0*] *80*)
**and** *sum-ddep-edge* :: *'node SDG-node* ⇒ *'var* ⇒ *'node SDG-node* ⇒ *bool*
(‹- *s*-‑⟶$_{dd}$ -› [*51*,*0*,*0*] *80*)
**and** *sum-call-edge* :: *'node SDG-node* ⇒ *'pname* ⇒ *'node SDG-node* ⇒ *bool*
(‹- *s*-‑⟶$_{call}$ -› [*51*,*0*,*0*] *80*)
**and** *sum-return-edge* :: *'node SDG-node* ⇒ *'pname* ⇒ *'node SDG-node* ⇒ *bool*
(‹- *s*-‑⟶$_{ret}$ -› [*51*,*0*,*0*] *80*)
**and** *sum-param-in-edge* :: *'node SDG-node* ⇒ *'pname* ⇒ *'var* ⇒ *'node SDG-node* ⇒ *bool*
(‹- *s*-‑:-⟶$_{in}$ -› [*51*,*0*,*0*,*0*] *80*)
**and** *sum-param-out-edge* :: *'node SDG-node* ⇒ *'pname* ⇒ *'var* ⇒ *'node SDG-node* ⇒ *bool*
(‹- *s*-‑:-⟶$_{out}$ -› [*51*,*0*,*0*,*0*] *80*)

**and** *sum-summary-edge* :: *'node SDG-node* ⇒ *'pname* ⇒ *'node SDG-node* ⇒ *bool*

(‹- $s-{-}\rightarrow_{sum}$ -› [*51,0*] *80*)
**and** *sum-SDG-edge* :: *'node SDG-node* ⇒ *'var option* ⇒
(*'pname* × *bool*) *option* ⇒ *bool* ⇒ *'node SDG-node* ⇒ *bool*

**where**

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

| *sum-SDG-cdep-edge*:
⟦*n* = *CFG-node m*; *n'* = *CFG-node m'*; *m controls m'*⟧ ⟹ $n\ s{\longrightarrow}_{cd}\ n'$
| *sum-SDG-proc-entry-exit-cdep*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪*pfs*; *n* = *CFG-node* (*targetnode a*);
*a'* ∈ *get-return-edges a*; *n'* = *CFG-node* (*sourcenode a'*)⟧ ⟹ $n\ s{\longrightarrow}_{cd}\ n'$
| *sum-SDG-parent-cdep-edge*:
⟦*valid-SDG-node n'*; *m* = *parent-node n'*; *n* = *CFG-node m*; *n* ≠ *n'*⟧
⟹ $n\ s{\longrightarrow}_{cd}\ n'$
| *sum-SDG-ddep-edge*:*n influences V in n'* ⟹ $n\ s{-}V{\rightarrow}_{dd}\ n'$
| *sum-SDG-call-edge*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪*pfs*; *n* = *CFG-node* (*sourcenode a*);
*n'* = *CFG-node* (*targetnode a*)⟧ ⟹ $n\ s{-}p{\rightarrow}_{call}\ n'$
| *sum-SDG-return-edge*:
⟦*valid-edge a*; *kind a* = *Q*↩*pfs*; *n* = *CFG-node* (*sourcenode a*);
*n'* = *CFG-node* (*targetnode a*)⟧ ⟹ $n\ s{-}p{\rightarrow}_{ret}\ n'$
| *sum-SDG-param-in-edge*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪*pfs*; (*p*,*ins*,*outs*) ∈ *set procs*; *V* = *ins*!*x*;
*x* < *length ins*; *n* = *Actual-in* (*sourcenode a*,*x*); *n'* = *Formal-in* (*targetnode a*,*x*)⟧
⟹ $n\ s{-}p{:}V{\rightarrow}_{in}\ n'$
| *sum-SDG-param-out-edge*:
⟦*valid-edge a*; *kind a* = *Q*↩*pf*; (*p*,*ins*,*outs*) ∈ *set procs*; *V* = *outs*!*x*;
*x* < *length outs*; *n* = *Formal-out* (*sourcenode a*,*x*);
*n'* = *Actual-out* (*targetnode a*,*x*)⟧
⟹ $n\ s{-}p{:}V{\rightarrow}_{out}\ n'$
| *sum-SDG-call-summary-edge*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪*pfs*; *a'* ∈ *get-return-edges a*;
*n* = *CFG-node* (*sourcenode a*); *n'* = *CFG-node* (*targetnode a'*)⟧
⟹ $n\ s{-}p{\rightarrow}_{sum}\ n'$
| *sum-SDG-param-summary-edge*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪*pfs*; *a'* ∈ *get-return-edges a*;
*matched* (*Formal-in* (*targetnode a*,*x*)) *ns* (*Formal-out* (*sourcenode a'*,*x'*));

$n = Actual\text{-}in\ (sourcenode\ a,x)$; $n' = Actual\text{-}out\ (targetnode\ a',x')$;
$(p,ins,outs) \in set\ procs$; $x < length\ ins$; $x' < length\ outs$⟧
$\Longrightarrow n\ s\text{-}p\rightarrow_{sum}\ n'$

**lemma** *sum-edge-cases*:
⟦$n\ s\text{-}p\rightarrow_{sum}\ n'$;
$\bigwedge a\ Q\ r\ fs\ a'$. ⟦$valid\text{-}edge\ a$; $kind\ a = Q{:}r\hookrightarrow_p fs$; $a' \in get\text{-}return\text{-}edges\ a$;
$n = CFG\text{-}node\ (sourcenode\ a)$; $n' = CFG\text{-}node\ (targetnode\ a')$⟧ $\Longrightarrow$
$P$;
$\bigwedge a\ Q\ p\ r\ fs\ a'\ ns\ x\ x'\ ins\ outs$.
⟦$valid\text{-}edge\ a$; $kind\ a = Q{:}r\hookrightarrow_p fs$; $a' \in get\text{-}return\text{-}edges\ a$;
$matched\ (Formal\text{-}in\ (targetnode\ a,x))\ ns\ (Formal\text{-}out\ (sourcenode\ a',x'))$;
$n = Actual\text{-}in\ (sourcenode\ a,x)$; $n' = Actual\text{-}out\ (targetnode\ a',x')$;
$(p,ins,outs) \in set\ procs$; $x < length\ ins$; $x' < length\ outs$⟧ $\Longrightarrow P$⟧
$\Longrightarrow P$
⟨*proof*⟩

**lemma** *SDG-edge-sum-SDG-edge*:
$SDG\text{-}edge\ n\ Vopt\ popt\ n' \Longrightarrow sum\text{-}SDG\text{-}edge\ n\ Vopt\ popt\ False\ n'$
⟨*proof*⟩

**lemma** *sum-SDG-edge-SDG-edge*:
$sum\text{-}SDG\text{-}edge\ n\ Vopt\ popt\ False\ n' \Longrightarrow SDG\text{-}edge\ n\ Vopt\ popt\ n'$
⟨*proof*⟩

**lemma** *sum-SDG-edge-valid-SDG-node*:
**assumes** $sum\text{-}SDG\text{-}edge\ n\ Vopt\ popt\ b\ n'$
**shows** $valid\text{-}SDG\text{-}node\ n$ **and** $valid\text{-}SDG\text{-}node\ n'$
⟨*proof*⟩

**lemma** *Exit-no-sum-SDG-edge-source*:
**assumes** $sum\text{-}SDG\text{-}edge\ (CFG\text{-}node\ (\text{-}Exit\text{-}))\ Vopt\ popt\ b\ n'$ **shows** $False$
⟨*proof*⟩

**lemma** *Exit-no-sum-SDG-edge-target*:
$sum\text{-}SDG\text{-}edge\ n\ Vopt\ popt\ b\ (CFG\text{-}node\ (\text{-}Exit\text{-})) \Longrightarrow False$
⟨*proof*⟩

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

**assumes** $n\ s{-}p{\rightarrow}_{sum}\ n'$
**obtains** *ns* **where** *matched n ns n'* **and** $n \in set\ ns$
**and** *get-proc* (*parent-node*(*last ns*)) = *p*
⟨*proof*⟩

**lemma** *return-edge-determines-call-and-sum-edge*:
**assumes** *valid-edge a* **and** $kind\ a = Q{\hookleftarrow}_{p}f$
**obtains** *a' Q' r' fs'* **where** $a \in get\text{-}return\text{-}edges\ a'$ **and** *valid-edge a'*
**and** $kind\ a' = Q'{:}r'{\hookrightarrow}_{p}fs'$
**and** *CFG-node* (*sourcenode a'*) $s{-}p{\rightarrow}_{sum}$ *CFG-node* (*targetnode a*)
⟨*proof*⟩

### 1.8.11 Paths consisting of intraprocedural and summary edges in the SDG

**inductive** *intra-sum-SDG-path* ::
*'node SDG-node* ⇒ *'node SDG-node list* ⇒ *'node SDG-node* ⇒ *bool*
(‹- $is{-}{-}{\rightarrow}_{d}*$ -› [*51*,*0*,*0*] *80*)
**where** *isSp-Nil*:
*valid-SDG-node n* ⟹ $n\ is{-}[]{\rightarrow}_{d}*\ n$

| *isSp-Append-cdep*:
⟦$n\ is{-}ns{\rightarrow}_{d}*\ n''$; $n''\ s{\longrightarrow}_{cd}\ n'$⟧ ⟹ $n\ is{-}ns@[n'']{\rightarrow}_{d}*\ n'$

| *isSp-Append-ddep*:
⟦$n\ is{-}ns{\rightarrow}_{d}*\ n''$; $n''\ s{-}V{\rightarrow}_{dd}\ n'$; $n'' \neq n'$⟧ ⟹ $n\ is{-}ns@[n'']{\rightarrow}_{d}*\ n'$

| *isSp-Append-sum*:
⟦$n\ is{-}ns{\rightarrow}_{d}*\ n''$; $n''\ s{-}p{\rightarrow}_{sum}\ n'$⟧ ⟹ $n\ is{-}ns@[n'']{\rightarrow}_{d}*\ n'$

**lemma** *is-SDG-path-Append*:
⟦$n''\ is{-}ns'{\rightarrow}_{d}*\ n'$; $n\ is{-}ns{\rightarrow}_{d}*\ n''$⟧ ⟹ $n\ is{-}ns@ns'{\rightarrow}_{d}*\ n'$
⟨*proof*⟩

**lemma** *is-SDG-path-valid-SDG-node*:
**assumes** $n\ is{-}ns{\rightarrow}_{d}*\ n'$ **shows** *valid-SDG-node n* **and** *valid-SDG-node n'*
⟨*proof*⟩

**lemma** *intra-SDG-path-is-SDG-path*:
$n\ i{-}ns{\rightarrow}_{d}*\ n'$ ⟹ $n\ is{-}ns{\rightarrow}_{d}*\ n'$
⟨*proof*⟩

**lemma** *is-SDG-path-hd*:⟦$n\ is{-}ns{\rightarrow}_{d}*\ n'$; $ns \neq []$⟧ ⟹ *hd ns* = *n*
⟨*proof*⟩

**lemma** *intra-sum-SDG-path-rev-induct* [*consumes 1*, *case-names isSp-Nil isSp-Cons-cdep isSp-Cons-ddep isSp-Cons-sum*]:
**assumes** $n\ is{-}ns{\rightarrow}_d{*}\ n'$
**and** *refl*: $\bigwedge n.\ valid\text{-}SDG\text{-}node\ n \Longrightarrow P\ n\ []\ n$
**and** *step-cdep*: $\bigwedge n\ ns\ n'\ n''.\ [\![n\ s{\longrightarrow}_{cd}\ n''; n''\ is{-}ns{\rightarrow}_d{*}\ n'; P\ n''\ ns\ n']\!]$
$\Longrightarrow P\ n\ (n\#ns)\ n'$
**and** *step-ddep*: $\bigwedge n\ ns\ n'\ V\ n''.\ [\![n\ s{-}V{\rightarrow}_{dd}\ n''; n \neq n''; n''\ is{-}ns{\rightarrow}_d{*}\ n';$
$P\ n''\ ns\ n']\!] \Longrightarrow P\ n\ (n\#ns)\ n'$
**and** *step-sum*: $\bigwedge n\ ns\ n'\ p\ n''.\ [\![n\ s{-}p{\rightarrow}_{sum}\ n''; n''\ is{-}ns{\rightarrow}_d{*}\ n'; P\ n''\ ns\ n']\!]$
$\Longrightarrow P\ n\ (n\#ns)\ n'$
**shows** $P\ n\ ns\ n'$
⟨*proof*⟩

**lemma** *is-SDG-path-CFG-path*:
**assumes** $n\ is{-}ns{\rightarrow}_d{*}\ n'$
**obtains** *as* **where** $parent\text{-}node\ n\ {-}as{\rightarrow}_\iota{*}\ parent\text{-}node\ n'$
⟨*proof*⟩

**lemma** *matched-is-SDG-path*:
**assumes** *matched n ns n'* **obtains** *ns'* **where** $n\ is{-}ns'{\rightarrow}_d{*}\ n'$
⟨*proof*⟩

**lemma** *is-SDG-path-matched*:
**assumes** $n\ is{-}ns{\rightarrow}_d{*}\ n'$ **obtains** *ns'* **where** *matched n ns' n'* **and** $set\ ns \subseteq set\ ns'$
⟨*proof*⟩

**lemma** *is-SDG-path-intra-CFG-path*:
**assumes** $n\ is{-}ns{\rightarrow}_d{*}\ n'$
**obtains** *as* **where** $parent\text{-}node\ n\ {-}as{\rightarrow}_\iota{*}\ parent\text{-}node\ n'$
⟨*proof*⟩

SDG paths without return edges

**inductive** *intra-call-sum-SDG-path* ::
*'node SDG-node ⇒ 'node SDG-node list ⇒ 'node SDG-node ⇒ bool*
(‹- ics−-→$_d$∗ -› [*51*,*0*,*0*] *80*)
**where** *icsSp-Nil*:
$valid\text{-}SDG\text{-}node\ n \Longrightarrow n\ ics{-}[]{\rightarrow}_d{*}\ n$

| *icsSp-Append-cdep*:
$[\![n\ ics{-}ns{\rightarrow}_d{*}\ n''; n''\ s{\longrightarrow}_{cd}\ n']\!] \Longrightarrow n\ ics{-}ns@[n'']{\rightarrow}_d{*}\ n'$

| *icsSp-Append-ddep*:

$[\![n\ ics{-}ns{\rightarrow_d}*\ n''; n''\ s{-}V{\rightarrow_{dd}}\ n'; n'' \neq n']\!] \Longrightarrow n\ ics{-}ns@[n'']{\rightarrow_d}*\ n'$

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

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

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

**lemma** *ics-SDG-path-valid-SDG-node*:
**assumes** $n\ ics{-}ns{\rightarrow_d}*\ n'$ **shows** *valid-SDG-node n* **and** *valid-SDG-node n'*
⟨*proof*⟩

**lemma** *ics-SDG-path-Append*:
$[\![n''\ ics{-}ns'{\rightarrow_d}*\ n'; n\ ics{-}ns{\rightarrow_d}*\ n'']\!] \Longrightarrow n\ ics{-}ns@ns'{\rightarrow_d}*\ n'$
⟨*proof*⟩

**lemma** *is-SDG-path-ics-SDG-path*:
$n\ is{-}ns{\rightarrow_d}*\ n' \Longrightarrow n\ ics{-}ns{\rightarrow_d}*\ n'$
⟨*proof*⟩

**lemma** *cc-SDG-path-ics-SDG-path*:
$n\ cc{-}ns{\rightarrow_d}*\ n' \Longrightarrow n\ ics{-}ns{\rightarrow_d}*\ n'$
⟨*proof*⟩

**lemma** *ics-SDG-path-split*:
**assumes** $n\ ics{-}ns{\rightarrow_d}*\ n'$ **and** $n'' \in set\ ns$
**obtains** $ns'\ ns''$ **where** $ns = ns'@ns''$ **and** $n\ ics{-}ns'{\rightarrow_d}*\ n''$
**and** $n''\ ics{-}ns''{\rightarrow_d}*\ n'$
⟨*proof*⟩

**lemma** *realizable-ics-SDG-path*:
**assumes** *realizable n ns n'* **obtains** *ns'* **where** $n\ ics{-}ns'{\rightarrow_d}*\ n'$
⟨*proof*⟩

**lemma** *ics-SDG-path-realizable*:
**assumes** $n\ ics{-}ns{\rightarrow_d}*\ n'$
**obtains** *ns'* **where** *realizable n ns' n'* **and** $set\ ns \subseteq set\ ns'$
⟨*proof*⟩

**lemma** *realizable-Append-ics-SDG-path*:
**assumes** *realizable n ns n''* **and** $n''$ *ics*$-ns'\rightarrow_d*$ $n'$
**obtains** $ns''$ **where** *realizable n* $(ns@ns'')$ $n'$
⟨*proof*⟩

### 1.8.12 SDG paths without call edges

**inductive** *intra-return-sum-SDG-path* ::
*'node SDG-node* ⇒ *'node SDG-node list* ⇒ *'node SDG-node* ⇒ *bool*
(‹- *irs*−-$\rightarrow_d*$ -› [*51*,*0*,*0*] *80*)
**where** *irsSp-Nil*:
*valid-SDG-node n* ⟹ *n irs*−[]$\rightarrow_d*$ *n*

| *irsSp-Cons-cdep*:
⟦$n''$ *irs*−*ns*$\rightarrow_d*$ $n'$; *n s*$\longrightarrow_{cd}$ $n''$⟧ ⟹ *n irs*−*n*#*ns*$\rightarrow_d*$ $n'$

| *irsSp-Cons-ddep*:
⟦$n''$ *irs*−*ns*$\rightarrow_d*$ $n'$; *n s*−*V*$\rightarrow_{dd}$ $n''$; $n \neq n''$⟧ ⟹ *n irs*−*n*#*ns*$\rightarrow_d*$ $n'$

| *irsSp-Cons-sum*:
⟦$n''$ *irs*−*ns*$\rightarrow_d*$ $n'$; *n s*−*p*$\rightarrow_{sum}$ $n''$⟧ ⟹ *n irs*−*n*#*ns*$\rightarrow_d*$ $n'$

| *irsSp-Cons-return*:
⟦$n''$ *irs*−*ns*$\rightarrow_d*$ $n'$; *n s*−*p*$\rightarrow_{ret}$ $n''$⟧ ⟹ *n irs*−*n*#*ns*$\rightarrow_d*$ $n'$

| *irsSp-Cons-param-out*:
⟦$n''$ *irs*−*ns*$\rightarrow_d*$ $n'$; *n s*−*p*:*V*$\rightarrow_{out}$ $n''$⟧ ⟹ *n irs*−*n*#*ns*$\rightarrow_d*$ $n'$

**lemma** *irs-SDG-path-Append*:
⟦*n irs*−*ns*$\rightarrow_d*$ $n''$; $n''$ *irs*−$ns'\rightarrow_d*$ $n'$⟧ ⟹ *n irs*−*ns*@$ns'\rightarrow_d*$ $n'$
⟨*proof*⟩

**lemma** *is-SDG-path-irs-SDG-path*:
*n is*−*ns*$\rightarrow_d*$ $n'$ ⟹ *n irs*−*ns*$\rightarrow_d*$ $n'$
⟨*proof*⟩

**lemma** *irs-SDG-path-split*:
**assumes** *n irs*−*ns*$\rightarrow_d*$ $n'$
**obtains** *n is*−*ns*$\rightarrow_d*$ $n'$
| *nsx* $nsx'$ *nx* $nx'$ *p* **where** *ns* = *nsx*@*nx*#$nsx'$ **and** *n irs*−*nsx*$\rightarrow_d*$ *nx*
**and** *nx s*−*p*$\rightarrow_{ret}$ $nx'$ ∨ (∃ *V*. *nx s*−*p*:*V*$\rightarrow_{out}$ $nx'$) **and** $nx'$ *is*−$nsx'\rightarrow_d*$ $n'$
⟨*proof*⟩

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

**lemma** *irs-SDG-path-realizable*:
**assumes** $n$ $irs{-}ns{\rightarrow_d}{*}$ $n'$ **and** $n \neq n'$
**obtains** $ns'$ **where** *realizable* (*CFG-node* (*-Entry-*)) $ns'$ $n'$ **and** $n \in set\ ns'$
⟨*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* ⇒ *'node SDG-node set*
**for** *n*::*'node SDG-node*
**where** *refl-slice1*:*valid-SDG-node n* ⟹ $n \in$ *sum-SDG-slice1 n*
| *cdep-slice1*:
⟦$n''$ $s{\longrightarrow_{cd}}$ $n'$; $n' \in$ *sum-SDG-slice1 n*⟧ ⟹ $n'' \in$ *sum-SDG-slice1 n*
| *ddep-slice1*:
⟦$n''$ $s{-}V{\rightarrow_{dd}}$ $n'$; $n' \in$ *sum-SDG-slice1 n*⟧ ⟹ $n'' \in$ *sum-SDG-slice1 n*
| *call-slice1*:
⟦$n''$ $s{-}p{\rightarrow_{call}}$ $n'$; $n' \in$ *sum-SDG-slice1 n*⟧ ⟹ $n'' \in$ *sum-SDG-slice1 n*
| *param-in-slice1*:
⟦$n''$ $s{-}p{:}V{\rightarrow_{in}}$ $n'$; $n' \in$ *sum-SDG-slice1 n*⟧ ⟹ $n'' \in$ *sum-SDG-slice1 n*
| *sum-slice1*:
⟦$n''$ $s{-}p{\rightarrow_{sum}}$ $n'$; $n' \in$ *sum-SDG-slice1 n*⟧ ⟹ $n'' \in$ *sum-SDG-slice1 n*

**lemma** *slice1-cdep-slice1*:
⟦$nx \in$ *sum-SDG-slice1 n*; $n$ $s{\longrightarrow_{cd}}$ $n'$⟧ ⟹ $nx \in$ *sum-SDG-slice1* $n'$
⟨*proof*⟩

**lemma** *slice1-ddep-slice1*:
⟦$nx \in$ *sum-SDG-slice1 n*; $n$ $s{-}V{\rightarrow_{dd}}$ $n'$⟧ ⟹ $nx \in$ *sum-SDG-slice1* $n'$
⟨*proof*⟩

**lemma** *slice1-sum-slice1*:

$[\![nx \in \textit{sum-SDG-slice1 } n;\ n\ s\!-\!p\!\to_{sum}\ n']\!] \Longrightarrow nx \in \textit{sum-SDG-slice1 } n'$
⟨*proof*⟩

**lemma** *slice1-call-slice1*:
$[\![nx \in \textit{sum-SDG-slice1 } n;\ n\ s\!-\!p\!\to_{call}\ n']\!] \Longrightarrow nx \in \textit{sum-SDG-slice1 } n'$
⟨*proof*⟩

**lemma** *slice1-param-in-slice1*:
$[\![nx \in \textit{sum-SDG-slice1 } n;\ n\ s\!-\!p{:}V\!\to_{in}\ n']\!] \Longrightarrow nx \in \textit{sum-SDG-slice1 } n'$
⟨*proof*⟩

**lemma** *is-SDG-path-slice1*:
$[\![n\ is\!-\!ns\!\to_{d}\!*\ n';\ n' \in \textit{sum-SDG-slice1 } n'']\!] \Longrightarrow n \in \textit{sum-SDG-slice1 } n''$
⟨*proof*⟩

### 1.9.2 Set describing phase 2 of the two-phase slicer

**inductive-set** *sum-SDG-slice2* :: *'node SDG-node* ⇒ *'node SDG-node set*
**for** *n*::*'node SDG-node*
**where** *refl-slice2*:*valid-SDG-node* $n \Longrightarrow n \in \textit{sum-SDG-slice2 } n$
| *cdep-slice2*:
$[\![n''\ s\!\longrightarrow_{cd}\ n';\ n' \in \textit{sum-SDG-slice2 } n]\!] \Longrightarrow n'' \in \textit{sum-SDG-slice2 } n$
| *ddep-slice2*:
$[\![n''\ s\!-\!V\!\to_{dd}\ n';\ n' \in \textit{sum-SDG-slice2 } n]\!] \Longrightarrow n'' \in \textit{sum-SDG-slice2 } n$
| *return-slice2*:
$[\![n''\ s\!-\!p\!\to_{ret}\ n';\ n' \in \textit{sum-SDG-slice2 } n]\!] \Longrightarrow n'' \in \textit{sum-SDG-slice2 } n$
| *param-out-slice2*:
$[\![n''\ s\!-\!p{:}V\!\to_{out}\ n';\ n' \in \textit{sum-SDG-slice2 } n]\!] \Longrightarrow n'' \in \textit{sum-SDG-slice2 } n$
| *sum-slice2*:
$[\![n''\ s\!-\!p\!\to_{sum}\ n';\ n' \in \textit{sum-SDG-slice2 } n]\!] \Longrightarrow n'' \in \textit{sum-SDG-slice2 } n$

**lemma** *slice2-cdep-slice2*:
$[\![nx \in \textit{sum-SDG-slice2 } n;\ n\ s\!\longrightarrow_{cd}\ n']\!] \Longrightarrow nx \in \textit{sum-SDG-slice2 } n'$
⟨*proof*⟩

**lemma** *slice2-ddep-slice2*:
$[\![nx \in \textit{sum-SDG-slice2 } n;\ n\ s\!-\!V\!\to_{dd}\ n']\!] \Longrightarrow nx \in \textit{sum-SDG-slice2 } n'$
⟨*proof*⟩

**lemma** *slice2-sum-slice2*:
$[\![nx \in \textit{sum-SDG-slice2 } n;\ n\ s\!-\!p\!\to_{sum}\ n']\!] \Longrightarrow nx \in \textit{sum-SDG-slice2 } n'$
⟨*proof*⟩

**lemma** *slice2-ret-slice2*:
$[\![nx \in \textit{sum-SDG-slice2 } n;\ n\ s\!-\!p\!\to_{ret}\ n']\!] \Longrightarrow nx \in \textit{sum-SDG-slice2 } n'$
⟨*proof*⟩

**lemma** *slice2-param-out-slice2*:
$[\![nx \in sum\text{-}SDG\text{-}slice2\ n;\ n\ s{-}p{:}V{\to}_{out}\ n']\!] \Longrightarrow nx \in sum\text{-}SDG\text{-}slice2\ n'$
⟨*proof*⟩

**lemma** *is-SDG-path-slice2*:
$[\![n\ is{-}ns{\to}_{d*}\ n';\ n' \in sum\text{-}SDG\text{-}slice2\ n'']\!] \Longrightarrow n \in sum\text{-}SDG\text{-}slice2\ n''$
⟨*proof*⟩

**lemma** *slice2-is-SDG-path-slice2*:
$[\![n\ is{-}ns{\to}_{d*}\ n';\ n'' \in sum\text{-}SDG\text{-}slice2\ n]\!] \Longrightarrow n'' \in sum\text{-}SDG\text{-}slice2\ n'$
⟨*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* ⇒ *'node SDG-node set*
**for** *S*::*'node SDG-node set*
**where** *combSlice-refl*: $n \in S \Longrightarrow n \in combine\text{-}SDG\text{-}slices\ S$
| *combSlice-Return-parent-node*:
$[\![n' \in S;\ n''\ s{-}p{\to}_{ret}\ CFG\text{-}node\ (parent\text{-}node\ n');\ n \in sum\text{-}SDG\text{-}slice2\ n']\!]$
$\Longrightarrow n \in combine\text{-}SDG\text{-}slices\ S$

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

**lemma** *HRB-slice-cases*[*consumes 1*,*case-names phase1 phase2*]:
$[\![x \in HRB\text{-}slice\ S;\ \bigwedge n\ nx.\ [\![n \in sum\text{-}SDG\text{-}slice1\ nx;\ nx \in S]\!] \Longrightarrow P\ n;$
$\bigwedge nx\ n'\ n''\ p\ n.\ [\![n' \in sum\text{-}SDG\text{-}slice1\ nx;\ n''\ s{-}p{\to}_{ret}\ CFG\text{-}node\ (parent\text{-}node\ n');$
$n \in sum\text{-}SDG\text{-}slice2\ n';\ nx \in S]\!] \Longrightarrow P\ n]\!]$
$\Longrightarrow P\ x$
⟨*proof*⟩

**lemma** *HRB-slice-refl*:
**assumes** *valid-node m* **and** $CFG\text{-}node\ m \in S$ **shows** $CFG\text{-}node\ m \in HRB\text{-}slice\ S$
⟨*proof*⟩

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

⟨*proof*⟩

**lemma** *valid-SDG-node-in-slice-parent-node-in-slice*:
**assumes** $n \in$ *HRB-slice S* **shows** *CFG-node* (*parent-node n*) $\in$ *HRB-slice S*
⟨*proof*⟩

**lemma** *HRB-slice-is-SDG-path-HRB-slice*:
$[\![n\ is{-}ns{\rightarrow_d}*\ n';\ n'' \in \textit{HRB-slice}\ \{n\};\ n' \in S]\!] \Longrightarrow n'' \in \textit{HRB-slice}\ S$
⟨*proof*⟩

**lemma** *call-return-nodes-in-slice*:
**assumes** *valid-edge a* **and** *kind* $a = Q\hookleftarrow_p f$
**and** *valid-edge* $a'$ **and** *kind* $a' = Q'{:}r'\hookrightarrow_p fs'$ **and** $a \in$ *get-return-edges* $a'$
**and** *CFG-node* (*targetnode a*) $\in$ *HRB-slice S*
**shows** *CFG-node* (*sourcenode a*) $\in$ *HRB-slice S*
**and** *CFG-node* (*sourcenode* $a'$) $\in$ *HRB-slice S*
**and** *CFG-node* (*targetnode* $a'$) $\in$ *HRB-slice S*
⟨*proof*⟩

### 1.9.4 Proof of Precision

**lemma** *in-intra-SDG-path-in-slice2*:
$[\![n\ i{-}ns{\rightarrow_d}*\ n';\ n'' \in set\ ns]\!] \Longrightarrow n'' \in \textit{sum-SDG-slice2}\ n'$
⟨*proof*⟩

**lemma** *in-intra-SDG-path-in-HRB-slice*:
$[\![n\ i{-}ns{\rightarrow_d}*\ n';\ n'' \in set\ ns;\ n' \in S]\!] \Longrightarrow n'' \in \textit{HRB-slice}\ S$
⟨*proof*⟩

**lemma** *in-matched-in-slice2*:
$[\![matched\ n\ ns\ n';\ n'' \in set\ ns]\!] \Longrightarrow n'' \in \textit{sum-SDG-slice2}\ n'$
⟨*proof*⟩

**lemma** *in-matched-in-HRB-slice*:
$[\![matched\ n\ ns\ n';\ n'' \in set\ ns;\ n' \in S]\!] \Longrightarrow n'' \in \textit{HRB-slice}\ S$
⟨*proof*⟩

**theorem** *in-realizable-in-HRB-slice*:
$[\![realizable\ n\ ns\ n';\ n'' \in set\ ns;\ n' \in S]\!] \Longrightarrow n'' \in \textit{HRB-slice}\ S$
⟨*proof*⟩

**lemma** *slice1-ics-SDG-path*:
  **assumes** $n \in$ *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*:
  $[\![\forall n' \in S.$ *valid-SDG-node* $n'$; $n \notin S]\!] \Longrightarrow$
    $n \in$ *HRB-slice S* $=$
    ($\exists n'$ *ns*. $n' \in S \wedge$ *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* ⇒ *'node set* (‹$\lfloor$-$\rfloor_{CFG}$›)
**where** $\lfloor S\rfloor_{CFG} \equiv \{m.\ \textit{CFG-node } m \in S\}$

**lemma** [*intro*]:$\forall n \in S.$ *valid-SDG-node n* $\Longrightarrow \forall n \in \lfloor S\rfloor_{CFG}.$ *valid-node n*
⟨*proof*⟩

**lemma** *Exit-HRB-Slice*:
**assumes** $n \in \lfloor$*HRB-slice* {*CFG-node* (*-Exit-*)}$\rfloor_{CFG}$ **shows** $n$ = (*-Exit-*)
⟨*proof*⟩

**lemma** *Exit-in-obs-intra-slice-node*:
**assumes** (*-Exit-*) ∈ *obs-intra n'* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
**shows** *CFG-node* (*-Exit-*) ∈ *S*
⟨*proof*⟩

**lemma** *obs-intra-postdominate*:
**assumes** $n \in$ *obs-intra n'* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ **and** ¬ *method-exit n*
**shows** *n postdominates n'*
⟨*proof*⟩

**lemma** *obs-intra-singleton-disj*:
**assumes** *valid-node n*
**shows** (∃ *m. obs-intra n* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {*m*}) ∨
*obs-intra n* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {}
⟨*proof*⟩

**lemma** *obs-intra-finite*:*valid-node n* ⟹ *finite* (*obs-intra n* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$)
⟨*proof*⟩

**lemma** *obs-intra-singleton*:*valid-node n* ⟹ *card* (*obs-intra n* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$) ≤ *1*
⟨*proof*⟩

**lemma** *obs-intra-singleton-element*:
*m* ∈ *obs-intra n* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ ⟹ *obs-intra n* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {*m*}
⟨*proof*⟩

**lemma** *obs-intra-the-element*:
$m \in$ *obs-intra n* $\lfloor$*HRB-slice S*$\rfloor_{CFG} \Longrightarrow$ (*THE m.* $m \in$ *obs-intra n* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$) = *m*
⟨*proof*⟩

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

**lemma** *obs-finite*:$\forall n \in$ *set* (*tl ns*). *return-node n*
$\Longrightarrow$ *finite* (*obs ns* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$)
⟨*proof*⟩

**lemma** *obs-singleton*:$\forall n \in$ *set* (*tl ns*). *return-node n*
$\Longrightarrow$ *card* (*obs ns* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$) $\leq$ *1*
⟨*proof*⟩

**lemma** *obs-the-element*:
⟦*ms* $\in$ *obs ns* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$; $\forall n \in$ *set* (*tl ns*). *return-node n*⟧
$\Longrightarrow$ (*THE ms. ms* $\in$ *obs ns* $\lfloor$*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* ⇒ *'node* ⇒ *nat* ⇒ *bool*
**where** *distanceI*:
⟦$n -as\rightarrow_\iota* n'$; *length as* = *x*; $\forall as'$. $n -as'\rightarrow_\iota* n' \longrightarrow x \leq$ *length as'*⟧
$\Longrightarrow$ *distance n n' x*

**lemma** *every-path-distance*:
**assumes** $n -as\rightarrow_\iota* n'$
**obtains** *x* **where** *distance n n' x* **and** $x \leq$ *length as*
⟨*proof*⟩

**lemma** *distance-det*:

$[\![distance\ n\ n'\ x;\ distance\ n\ n'\ x']\!] \Longrightarrow x = x'$
⟨*proof*⟩

**lemma** *only-one-SOME-dist-edge*:
**assumes** *valid-edge a* **and** *intra-kind*(*kind a*) **and** *distance* (*targetnode a*) *n′ x*
**shows** $\exists!a'.$ *sourcenode a* = *sourcenode a′* ∧ *distance* (*targetnode a′*) *n′ x* ∧
*valid-edge a′* ∧ *intra-kind*(*kind a′*) ∧
*targetnode a′* = (*SOME nx*. ∃ *a′*. *sourcenode a* = *sourcenode a′* ∧
*distance* (*targetnode a′*) *n′ x* ∧
*valid-edge a′* ∧ *intra-kind*(*kind a′*) ∧
*targetnode a′* = *nx*)
⟨*proof*⟩

**lemma** *distance-successor-distance*:
**assumes** *distance n n′ x* **and** *x* ≠ *0*
**obtains** *a* **where** *valid-edge a* **and** *n* = *sourcenode a* **and** *intra-kind*(*kind a*)
**and** *distance* (*targetnode a*) *n′* (*x* − *1*)
**and** *targetnode a* = (*SOME nx*. ∃ *a′*. *sourcenode a* = *sourcenode a′* ∧
*distance* (*targetnode a′*) *n′* (*x* − *1*) ∧
*valid-edge a′* ∧ *intra-kind*(*kind a′*) ∧
*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* ⇒ *′node SDG-node set* ⇒ *nat* ⇒
(((*′var* ⇀ *′val*) ⇒ *′val option*) *list*) ⇒ (((*′var* ⇀ *′val*) ⇒ *′val option*) *list*)
**where** *csppa m S x* [] = []
| *csppa m S x* (*f*#*fs*) =
(*if Formal-in*(*m*,*x*) ∉ *S then Map.empty else f*)#*csppa m S* (*Suc x*) *fs*

**definition** *cspp* :: *′node* ⇒ *′node SDG-node set* ⇒
(((*′var* ⇀ *′val*) ⇒ *′val option*) *list*) ⇒ (((*′var* ⇀ *′val*) ⇒ *′val option*) *list*)
**where** *cspp m S fs* ≡ *csppa m S 0 fs*

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

$\langle proof \rangle$

**lemma** [*simp*]: *length* (*cspp m S fs*) = *length fs*
$\langle proof \rangle$

**lemma** *csppa-Formal-in-notin-slice*:
$[\![x < length\ fs;\ Formal\text{-}in(m,x + i) \notin S]\!]$
$\Longrightarrow (csppa\ m\ S\ i\ fs)!x = Map.empty$
$\langle proof \rangle$

**lemma** *csppa-Formal-in-in-slice*:
$[\![x < length\ fs;\ Formal\text{-}in(m,x + i) \in S]\!]$
$\Longrightarrow (csppa\ m\ S\ i\ fs)!x = fs!x$
$\langle proof \rangle$

**definition** *map-merge* :: $('var \rightharpoonup 'val) \Rightarrow ('var \rightharpoonup 'val) \Rightarrow (nat \Rightarrow bool) \Rightarrow$
$'var\ list \Rightarrow ('var \rightharpoonup 'val)$
**where** *map-merge f g Q xs* $\equiv (\lambda V.\ if\ (\exists\, i.\ i < length\ xs \wedge xs!i = V \wedge Q\ i)\ then$
$g\ V$
$else\ f\ V)$

**definition** *rspp* :: $'node \Rightarrow 'node\ SDG\text{-}node\ set \Rightarrow 'var\ list \Rightarrow$
$('var \rightharpoonup 'val) \Rightarrow ('var \rightharpoonup 'val) \Rightarrow ('var \rightharpoonup 'val)$
**where** *rspp m S xs f g* $\equiv$ *map-merge f* (*Map.empty*(*ParamDefs m* [:=] *map g xs*))
($\lambda i.$ *Actual-out*($m$,$i$) $\in S$) (*ParamDefs m*)

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

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

### 1.12.2 Defining the sliced edge kinds

**primrec** *slice-kind-aux* :: $'node \Rightarrow 'node \Rightarrow 'node\ SDG\text{-}node\ set \Rightarrow$
$('var,'val,'ret,'pname)\ edge\text{-}kind \Rightarrow ('var,'val,'ret,'pname)\ edge\text{-}kind$
**where** *slice-kind-aux m m' S* $\Uparrow f = (if\ m \in \lfloor S \rfloor_{CFG}\ then\ \Uparrow f\ else\ \Uparrow id)$

| *slice-kind-aux m m′ S* $(Q)_{\surd}$ = (*if m* ∈ $\lfloor S\rfloor_{CFG}$ *then* $(Q)_{\surd}$ *else*
(*if obs-intra m* $\lfloor S\rfloor_{CFG}$ = {} *then*
(*let mex* = (*THE mex. method-exit mex* ∧ *get-proc m* = *get-proc mex*) *in*
(*if* (∃ *x. distance m′ mex x* ∧ *distance m mex* (*x* + *1*) ∧
(*m′* = (*SOME mx′*. ∃ *a′*. *m* = *sourcenode a′* ∧
*distance* (*targetnode a′*) *mex x* ∧
*valid-edge a′* ∧ *intra-kind*(*kind a′*) ∧
*targetnode a′* = *mx′*)))
*then* $(\lambda cf.\ True)_{\surd}$ *else* $(\lambda cf.\ False)_{\surd}$))
*else* (*let mx* = *THE mx. mx* ∈ *obs-intra m* $\lfloor S\rfloor_{CFG}$ *in*
(*if* (∃ *x. distance m′ mx x* ∧ *distance m mx* (*x* + *1*) ∧
(*m′* = (*SOME mx′*. ∃ *a′*. *m* = *sourcenode a′* ∧
*distance* (*targetnode a′*) *mx x* ∧
*valid-edge a′* ∧ *intra-kind*(*kind a′*) ∧
*targetnode a′* = *mx′*)))
*then* $(\lambda cf.\ True)_{\surd}$ *else* $(\lambda cf.\ False)_{\surd}$))))
| *slice-kind-aux m m′ S* ($Q{:}r\hookrightarrow_p fs$) = (*if m* ∈ $\lfloor S\rfloor_{CFG}$ *then* ($Q{:}r\hookrightarrow_p$(*cspp m′ S fs*))
*else* (($\lambda cf.\ False$)$:r\hookrightarrow_p fs$))
| *slice-kind-aux m m′ S* ($Q\hookleftarrow_p f$) = (*if m* ∈ $\lfloor S\rfloor_{CFG}$ *then*
(*let outs* = *THE outs*. ∃ *ins*. (*p*,*ins*,*outs*) ∈ *set procs in*
($Q\hookleftarrow_p$($\lambda cf\ cf'$. *rspp m′ S outs cf′ cf*)))
*else* (($\lambda cf.\ True$)$\hookleftarrow_p$($\lambda cf\ cf'.\ cf'$)))

**definition** *slice-kind* :: *′node SDG-node set* ⇒ *′edge* ⇒
(*′var*,*′val*,*′ret*,*′pname*) *edge-kind*
**where** *slice-kind S a* ≡
*slice-kind-aux* (*sourcenode a*) (*targetnode a*) (*HRB-slice S*) (*kind a*)

**definition** *slice-kinds* :: *′node SDG-node set* ⇒ *′edge list* ⇒
(*′var*,*′val*,*′ret*,*′pname*) *edge-kind list*
**where** *slice-kinds S as* ≡ *map* (*slice-kind S*) *as*

**lemma** *slice-intra-kind-in-slice*:
⟦*sourcenode a* ∈ $\lfloor HRB\text{-}slice\ S\rfloor_{CFG}$; *intra-kind* (*kind a*)⟧
⟹ *slice-kind S a* = *kind a*
⟨*proof*⟩

**lemma** *slice-kind-Upd*:
⟦*sourcenode a* ∉ $\lfloor HRB\text{-}slice\ S\rfloor_{CFG}$; *kind a* = ⇑*f*⟧ ⟹ *slice-kind S a* = ⇑*id*
⟨*proof*⟩

**lemma** *slice-kind-Pred-empty-obs-nearer-SOME*:
**assumes** *sourcenode a* ∉ $\lfloor HRB\text{-}slice\ S\rfloor_{CFG}$ **and** *kind a* = $(Q)_{\surd}$
**and** *obs-intra* (*sourcenode a*) $\lfloor HRB\text{-}slice\ S\rfloor_{CFG}$ = {}

**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* = (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
*distance* (*targetnode a'*) *mex x* ∧
*valid-edge a'* ∧ *intra-kind*(*kind a'*) ∧
*targetnode a'* = *n'*)
**shows** *slice-kind S a* = (λ*s*. *True*)$_{\surd}$
⟨*proof*⟩

**lemma** *slice-kind-Pred-empty-obs-nearer-not-SOME*:
**assumes** *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *kind a* = (*Q*)$_{\surd}$
**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* ≠ (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
*distance* (*targetnode a'*) *mex x* ∧
*valid-edge a'* ∧ *intra-kind*(*kind a'*) ∧
*targetnode a'* = *n'*)
**shows** *slice-kind S a* = (λ*s*. *False*)$_{\surd}$
⟨*proof*⟩

**lemma** *slice-kind-Pred-empty-obs-not-nearer*:
**assumes** *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *kind a* = (*Q*)$_{\surd}$
**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*) ¬ *distance* (*targetnode a*) *mex x*
**shows** *slice-kind S a* = (λ*s*. *False*)$_{\surd}$
⟨*proof*⟩

**lemma** *slice-kind-Pred-obs-nearer-SOME*:
**assumes** *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *kind a* = (*Q*)$_{\surd}$
**and** *m* ∈ *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'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
*distance* (*targetnode a'*) *m x* ∧
*valid-edge a'* ∧ *intra-kind*(*kind a'*) ∧
*targetnode a'* = *n'*)
**shows** *slice-kind S a* = (λ*s*. *True*)$_{\surd}$
⟨*proof*⟩

**lemma** *slice-kind-Pred-obs-nearer-not-SOME*:
**assumes** *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *kind a* = (*Q*)$_{\surd}$
**and** *m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$
**and** *distance* (*targetnode a*) *m x distance* (*sourcenode a*) *m* (*x* + *1*)
**and** *targetnode a* ≠ (*SOME nx'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧

$distance\ (targetnode\ a')\ m\ x \wedge$
$valid\text{-}edge\ a' \wedge intra\text{-}kind(kind\ a') \wedge$
$targetnode\ a' = nx')$
**shows** $slice\text{-}kind\ S\ a = (\lambda s.\ False)_{\surd}$
⟨*proof*⟩

**lemma** *slice-kind-Pred-obs-not-nearer*:
**assumes** $sourcenode\ a \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **and** $kind\ a = (Q)_{\surd}$
**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** $slice\text{-}kind\ S\ a = (\lambda s.\ False)_{\surd}$
⟨*proof*⟩

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

**lemma** *slice-kind-Call*:
$[\![sourcenode\ a \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG};\ kind\ a = Q{:}r \hookrightarrow_{p} fs]\!]$
$\Longrightarrow slice\text{-}kind\ S\ a = (\lambda cf.\ False){:}r \hookrightarrow_{p} fs$
⟨*proof*⟩

**lemma** *slice-kind-Call-in-slice*:
$[\![sourcenode\ a \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG};\ kind\ a = Q{:}r \hookrightarrow_{p} fs]\!]$
$\Longrightarrow slice\text{-}kind\ S\ a = Q{:}r \hookrightarrow_{p} (cspp\ (targetnode\ a)\ (HRB\text{-}slice\ S)\ fs)$
⟨*proof*⟩

**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** $slice\text{-}kind\ S\ a = Q{:}r \hookrightarrow_{p} replicate\ (length\ fs)\ Map.empty$
⟨*proof*⟩

**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** $slice\text{-}kind\ S\ a = Q{:}r \hookrightarrow_{p} fs$
⟨*proof*⟩

**lemma** *slice-kind-Call-intra-notin-slice*:
**assumes** $sourcenode\ a \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **and** *valid-edge a*
**and** *intra-kind* (*kind a*) **and** *valid-edge a'* **and** $kind\ a' = Q{:}r\hookrightarrow_p fs$
**and** *sourcenode a'* = *sourcenode a*
**shows** $slice\text{-}kind\ S\ a = (\lambda s.\ True)_{\surd}$
⟨*proof*⟩

**lemma** *slice-kind-Return*:
$[\![sourcenode\ a \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG};\ kind\ a = Q\hookleftarrow_p f]\!]$
$\Longrightarrow slice\text{-}kind\ S\ a = (\lambda cf.\ True)\hookleftarrow_p(\lambda cf\ cf'.\ cf')$
⟨*proof*⟩

**lemma** *slice-kind-Return-in-slice*:
$[\![sourcenode\ a \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG};\ valid\text{-}edge\ a;\ kind\ a = Q\hookleftarrow_p f;$
$(p,ins,outs) \in set\ procs]\!]$
$\Longrightarrow slice\text{-}kind\ S\ a = Q\hookleftarrow_p(\lambda cf\ cf'.\ rspp\ (targetnode\ a)\ (HRB\text{-}slice\ S)\ outs\ cf'$
$cf)$
⟨*proof*⟩

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

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

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

**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* = $(Q)_{\surd}$ **and** *slice-kind S a'* = $(Q')_{\surd}$
**and** ∀ *s*. (*Q s* ⟶ ¬ *Q' s*) ∧ (*Q' s* ⟶ ¬ *Q s*)
⟨*proof*⟩

**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'* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
**shows** *n* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
⟨*proof*⟩

**lemma** *edge-obs-intra-slice-eq*:
**assumes** *valid-edge a* **and** *intra-kind* (*kind a*) **and** *sourcenode a* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
**shows** *obs-intra* (*targetnode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ =
*obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
⟨*proof*⟩

**lemma** *intra-edge-obs-slice*:
**assumes** *ms* ≠ [] **and** *ms''* ∈ *obs ms'* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ **and** *valid-edge a*
**and** *intra-kind* (*kind a*)
**and** *disj*:(∃ *m* ∈ *set* (*tl ms*). ∃ *m'*. *call-of-return-node m m'* ∧
*m'* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$) ∨ *hd ms* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
**and** *hd ms* = *sourcenode a* **and** *ms'* = *targetnode a#tl ms*
**and** ∀ *n* ∈ *set* (*tl ms'*). *return-node n*
**shows** *ms''* ∈ *obs ms* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
⟨*proof*⟩

### 1.13.1 Silent moves

**inductive** *silent-move* ::
*'node SDG-node set* ⇒ (*'edge* ⇒ (*'var*,*'val*,*'ret*,*'pname*) *edge-kind*) ⇒ *'node list* ⇒

$(('var \rightharpoonup 'val) \times 'ret)$ *list* $\Rightarrow$ *'edge* $\Rightarrow$ *'node list* $\Rightarrow$ $(('var \rightharpoonup 'val) \times 'ret)$ *list* $\Rightarrow$ *bool*
(‹-,- ⊢ '(-,-') −-→$_\tau$ '(-,-')› [51,50,0,0,50,0,0] 51)

**where** *silent-move-intra*:
⟦*pred* (*f a*) *s*; *transfer* (*f a*) *s* = *s'*; *valid-edge a*; *intra-kind*(*kind a*);
(∃ *m* ∈ *set* (*tl ms*). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$) ∨
*hd ms* ∉ ⌊*HRB-slice S*⌋$_{CFG}$; ∀ *m* ∈ *set* (*tl ms*). *return-node m*;
*length s'* = *length s*; *length ms* = *length s*;
*hd ms* = *sourcenode a*; *ms'* = (*targetnode a*)#*tl ms*⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*)

| *silent-move-call*:
⟦*pred* (*f a*) *s*; *transfer* (*f a*) *s* = *s'*; *valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*;
*valid-edge a'*; *a'* ∈ *get-return-edges a*;
(∃ *m* ∈ *set* (*tl ms*). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$) ∨
*hd ms* ∉ ⌊*HRB-slice S*⌋$_{CFG}$; ∀ *m* ∈ *set* (*tl ms*). *return-node m*;
*length ms* = *length s*; *length s'* = *Suc*(*length s*);
*hd ms* = *sourcenode a*; *ms'* = (*targetnode a*)#(*targetnode a'*)#*tl ms*⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*)

| *silent-move-return*:
⟦*pred* (*f a*) *s*; *transfer* (*f a*) *s* = *s'*; *valid-edge a*; *kind a* = *Q*↩$_p$*f'*;
∃ *m* ∈ *set* (*tl ms*). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$;
∀ *m* ∈ *set* (*tl ms*). *return-node m*; *length ms* = *length s*; *length s* = *Suc*(*length s'*);
*s'* ≠ []; *hd ms* = *sourcenode a*; *hd*(*tl ms*) = *targetnode a*; *ms'* = *tl ms*⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*)

**lemma** *silent-move-valid-nodes*:
⟦*S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*); ∀ *m* ∈ *set ms'*. *valid-node m*⟧
⟹ ∀ *m* ∈ *set ms*. *valid-node m*
⟨*proof*⟩

**lemma** *silent-move-return-node*:
*S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*) ⟹ ∀ *m* ∈ *set* (*tl ms'*). *return-node m*
⟨*proof*⟩

**lemma** *silent-move-equal-length*:
**assumes** *S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*)
**shows** *length ms* = *length s* **and** *length ms'* = *length s'*
⟨*proof*⟩

**lemma** *silent-move-obs-slice*:
$[\![S,kind \vdash (ms,s)\ -a\rightarrow_\tau\ (ms',s');\ msx \in obs\ ms'\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG};$
$\forall n \in set\ (tl\ ms').\ return\text{-}node\ n]\!]$
$\Longrightarrow msx \in obs\ ms\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG}$
⟨*proof*⟩

**lemma** *silent-move-empty-obs-slice*:
**assumes** $S,f \vdash (ms,s)\ -a\rightarrow_\tau\ (ms',s')$ **and** $obs\ ms'\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG} = \{\}$
**shows** $obs\ ms\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG} = \{\}$
⟨*proof*⟩

**inductive** *silent-moves* ::
$'node\ SDG\text{-}node\ set \Rightarrow ('edge \Rightarrow ('var,'val,'ret,'pname)\ edge\text{-}kind) \Rightarrow 'node\ list$
$\Rightarrow$
$(('var \rightharpoonup 'val) \times 'ret)\ list \Rightarrow 'edge\ list \Rightarrow 'node\ list \Rightarrow (('var \rightharpoonup 'val) \times 'ret)\ list$
$\Rightarrow bool$
(‹-,- ⊢ '(-,-') =-⇒$_\tau$ '(-,-')› [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*:
$[\![S,f \vdash (ms,s)\ -a\rightarrow_\tau\ (ms',s');\ S,f \vdash (ms',s') =as\Rightarrow_\tau (ms'',s'')]\!]$
$\Longrightarrow S,f \vdash (ms,s) =a\#as\Rightarrow_\tau (ms'',s'')$

**lemma** *silent-moves-equal-length*:
**assumes** $S,f \vdash (ms,s) =as\Rightarrow_\tau (ms',s')$
**shows** $length\ ms = length\ s$ **and** $length\ ms' = length\ s'$
⟨*proof*⟩

**lemma** *silent-moves-Append*:
$[\![S,f \vdash (ms,s) =as\Rightarrow_\tau (ms'',s'');\ S,f \vdash (ms'',s'') =as'\Rightarrow_\tau (ms',s')]\!]$
$\Longrightarrow S,f \vdash (ms,s) =as@as'\Rightarrow_\tau (ms',s')$
⟨*proof*⟩

**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** $S,f \vdash (ms'',s'') =as'\Rightarrow_\tau (ms',s')$
⟨*proof*⟩

**lemma** *valid-nodes-silent-moves*:

$[\![S,f\vdash (ms,s) =as'\Rightarrow_\tau (ms',s'); \forall m \in set\ ms.\ valid\text{-}node\ m]\!]$
$\Longrightarrow \forall m \in set\ ms'.\ valid\text{-}node\ m$
⟨*proof*⟩

**lemma** *return-nodes-silent-moves*:
$[\![S,f \vdash (ms,s) =as'\Rightarrow_\tau (ms',s'); \forall m \in set\ (tl\ ms).\ return\text{-}node\ m]\!]$
$\Longrightarrow \forall m \in set\ (tl\ ms').\ return\text{-}node\ m$
⟨*proof*⟩

**lemma** *silent-moves-intra-path*:
$[\![S,f \vdash (m\#ms,s) =as\Rightarrow_\tau (m'\#ms',s'); \forall a \in set\ as.\ intra\text{-}kind(kind\ a)]\!]$
$\Longrightarrow ms = ms' \wedge get\text{-}proc\ m = get\text{-}proc\ m'$
⟨*proof*⟩

**lemma** *silent-moves-nodestack-notempty*:
$[\![S,f \vdash (ms,s) =as\Rightarrow_\tau (ms',s'); ms \neq []]\!] \Longrightarrow ms' \neq []$
⟨*proof*⟩

**lemma** *silent-moves-obs-slice*:
$[\![S,kind \vdash (ms,s) =as\Rightarrow_\tau (ms',s'); mx \in obs\ ms'\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG};$
$\forall n \in set\ (tl\ ms').\ return\text{-}node\ n]\!]$
$\Longrightarrow mx \in obs\ ms\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG} \wedge (\forall n \in set\ (tl\ ms).\ return\text{-}node\ n)$
⟨*proof*⟩

**lemma** *silent-moves-empty-obs-slice*:
$[\![S,f \vdash (ms,s) =as\Rightarrow_\tau (ms',s'); obs\ ms'\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG} = \{\}]\!]$
$\Longrightarrow obs\ ms\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG} = \{\}$
⟨*proof*⟩

**lemma** *silent-moves-preds-transfers*:
**assumes** $S,f \vdash (ms,s) =as\Rightarrow_\tau (ms',s')$
**shows** *preds* (*map f as*) *s* **and** *transfers* (*map f as*) $s = s'$
⟨*proof*⟩

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

**lemma** *silent-moves-intra-path-no-obs*:
**assumes** *obs-intra m* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {} **and** *method-exit m′*
**and** *get-proc m* = *get-proc m′* **and** *valid-node m* **and** *length s* = *length* (*m*#*msx′*)
**and** ∀ *m* ∈ *set msx′*. *return-node m*
**obtains** *as* **where** *S*,*slice-kind S* ⊢ (*m*#*msx′*,*s*) =*as*⇒$_\tau$ (*m′*#*msx′*,*s*)
⟨*proof*⟩

**lemma** *silent-moves-vpa-path*:
**assumes** *S*,*f* ⊢ (*m*#*ms*,*s*) =*as*⇒$_\tau$ (*m′*#*ms′*,*s′*) **and** *valid-node m*
**and** ∀ *i* < *length rs*. *rs*!*i* ∈ *get-return-edges* (*cs*!*i*)
**and** *ms* = *targetnodes rs* **and** *valid-return-list rs m*
**and** *length rs* = *length cs*
**shows** *m* −*as*→* *m′* **and** *valid-path-aux cs as*
⟨*proof*⟩

### 1.13.2 Observable moves

**inductive** *observable-move* ::
′*node SDG-node set* ⇒ (′*edge* ⇒ (′*var*,′*val*,′*ret*,′*pname*) *edge-kind*) ⇒ ′*node list* ⇒
((′*var* ⇀ ′*val*) × ′*ret*) *list* ⇒ ′*edge* ⇒ ′*node list* ⇒ ((′*var* ⇀ ′*val*) × ′*ret*) *list* ⇒ *bool*
(‹-,- ⊢ ′(-,-′) −-→ ′(-,-′)› [*51*,*50*,*0*,*0*,*50*,*0*,*0*] *51*)

**where** *observable-move-intra*:
⟦*pred* (*f a*) *s*; *transfer* (*f a*) *s* = *s′*; *valid-edge a*; *intra-kind*(*kind a*);
∀ *m* ∈ *set* (*tl ms*). ∃ *m′*. *call-of-return-node m m′* ∧ *m′* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$;
*hd ms* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$; *length s′* = *length s*; *length ms* = *length s*;
*hd ms* = *sourcenode a*; *ms′* = (*targetnode a*)#*tl ms*⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) −*a*→ (*ms′*,*s′*)

| *observable-move-call*:
⟦*pred* (*f a*) *s*; *transfer* (*f a*) *s* = *s′*; *valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*;
*valid-edge a′*; *a′* ∈ *get-return-edges a*;
∀ *m* ∈ *set* (*tl ms*). ∃ *m′*. *call-of-return-node m m′* ∧ *m′* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$;
*hd ms* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$; *length ms* = *length s*; *length s′* = *Suc*(*length s*);
*hd ms* = *sourcenode a*; *ms′* = (*targetnode a*)#(*targetnode a′*)#*tl ms*⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) −*a*→ (*ms′*,*s′*)

| *observable-move-return*:
⟦*pred* (*f a*) *s*; *transfer* (*f a*) *s* = *s′*; *valid-edge a*; *kind a* = *Q*↩$_p$*f′*;
∀ *m* ∈ *set* (*tl ms*). ∃ *m′*. *call-of-return-node m m′* ∧ *m′* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$;
*length ms* = *length s*; *length s* = *Suc*(*length s′*); *s′* ≠ [];
*hd ms* = *sourcenode a*; *hd*(*tl ms*) = *targetnode a*; *ms′* = *tl ms*⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) −*a*→ (*ms′*,*s′*)

**inductive** *observable-moves* ::
$'node$ *SDG-node set* $\Rightarrow$ $('edge \Rightarrow ('var,'val,'ret,'pname)$ *edge-kind*$) \Rightarrow 'node$ *list* $\Rightarrow$
$(('var \rightharpoonup 'val) \times 'ret)$ *list* $\Rightarrow 'edge$ *list* $\Rightarrow 'node$ *list* $\Rightarrow (('var \rightharpoonup 'val) \times 'ret)$ *list* $\Rightarrow$ *bool*
$(\langle$-,- $\vdash$ $'$(-,-$'$) =-$\Rightarrow$ $'$(-,-$'$)$\rangle$ $[51,50,0,0,50,0,0]$ $51$)

**where** *observable-moves-snoc*:
$[\![S,f \vdash (ms,s) =as\Rightarrow_\tau (ms',s'); S,f \vdash (ms',s') -a\rightarrow (ms'',s'')]\!]$
$\Longrightarrow S,f \vdash (ms,s) =as@[a]\Rightarrow (ms'',s'')$

**lemma** *observable-move-equal-length*:
**assumes** $S,f \vdash (ms,s) -a\rightarrow (ms',s')$
**shows** *length ms* = *length s* **and** *length ms'* = *length s'*
⟨*proof*⟩

**lemma** *observable-moves-equal-length*:
**assumes** $S,f \vdash (ms,s) =as\Rightarrow (ms',s')$
**shows** *length ms* = *length s* **and** *length ms'* = *length s'*
⟨*proof*⟩

**lemma** *observable-move-notempty*:
$[\![S,f \vdash (ms,s) =as\Rightarrow (ms',s'); as = []]\!] \Longrightarrow$ *False*
⟨*proof*⟩

**lemma** *silent-move-observable-moves*:
$[\![S,f \vdash (ms'',s'') =as\Rightarrow (ms',s'); S,f \vdash (ms,s) -a\rightarrow_\tau (ms'',s'')]\!]$
$\Longrightarrow S,f \vdash (ms,s) =a\#as\Rightarrow (ms',s')$
⟨*proof*⟩

**lemma** *silent-append-observable-moves*:
$[\![S,f \vdash (ms,s) =as\Rightarrow_\tau (ms'',s''); S,f \vdash (ms'',s'') =as'\Rightarrow (ms',s')]\!]$
$\Longrightarrow S,f \vdash (ms,s) =as@as'\Rightarrow (ms',s')$
⟨*proof*⟩

**lemma** *observable-moves-preds-transfers*:
**assumes** $S,f \vdash (ms,s) =as\Rightarrow (ms',s')$
**shows** *preds* (*map f as*) *s* **and** *transfers* (*map f as*) *s* = *s'*
⟨*proof*⟩

**lemma** *observable-move-vpa-path*:
$[\![S,f \vdash (m\#ms,s) -a\rightarrow (m'\#ms',s');$ *valid-node m*;

$\forall i < length\ rs.\ rs!i \in get\text{-}return\text{-}edges\ (cs!i);\ ms = targetnodes\ rs;$
$valid\text{-}return\text{-}list\ rs\ m;\ length\ rs = length\ cs]\!] \Longrightarrow valid\text{-}path\text{-}aux\ cs\ [a]$
⟨proof⟩

### 1.13.3 Relevant variables

**inductive-set** *relevant-vars* ::
$'node\ SDG\text{-}node\ set \Rightarrow 'node\ SDG\text{-}node \Rightarrow 'var\ set$ (‹rv -›)
**for** $S$ :: $'node\ SDG\text{-}node\ set$ **and** $n$ :: $'node\ SDG\text{-}node$

**where** *rvI*:
$[\![parent\text{-}node\ n\ -as\rightarrow_\iota *\ parent\text{-}node\ n';\ n' \in HRB\text{-}slice\ S;\ V \in Use_{SDG}\ n';$
$\forall n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge parent\text{-}node\ n'' \in set\ (sourcenodes\ as)$
$\longrightarrow V \notin Def_{SDG}\ n'']\!]$
$\Longrightarrow V \in rv\ S\ n$

**lemma** *rvE*:
**assumes** *rv*: $V \in rv\ S\ n$
**obtains** *as* $n'$ **where** $parent\text{-}node\ n\ -as\rightarrow_\iota *\ parent\text{-}node\ n'$
**and** $n' \in HRB\text{-}slice\ S$ **and** $V \in Use_{SDG}\ n'$
**and** $\forall n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge parent\text{-}node\ n'' \in set\ (sourcenodes\ as)$
$\longrightarrow V \notin Def_{SDG}\ n''$
⟨proof⟩

**lemma** *rv-parent-node*:
$parent\text{-}node\ n = parent\text{-}node\ n' \Longrightarrow rv\ (S::'node\ SDG\text{-}node\ set)\ n = rv\ S\ n'$
⟨proof⟩

**lemma** *obs-intra-empty-rv-empty*:
**assumes** $obs\text{-}intra\ m\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG} = \{\}$ **shows** $rv\ S\ (CFG\text{-}node\ m) = \{\}$
⟨proof⟩

**lemma** *eq-obs-intra-in-rv*:
**assumes** *obs-eq*: $obs\text{-}intra\ (parent\text{-}node\ n)\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG} =$
$obs\text{-}intra\ (parent\text{-}node\ n')\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG}$
**and** $x \in rv\ S\ n$ **shows** $x \in rv\ S\ n'$
⟨proof⟩

**lemma** *closed-eq-obs-eq-rvs*:
**fixes** $S$ :: $'node\ SDG\text{-}node\ set$
**assumes** *obs-eq*: $obs\text{-}intra\ (parent\text{-}node\ n)\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG} =$
$obs\text{-}intra\ (parent\text{-}node\ n')\ \lfloor HRB\text{-}slice\ S\rfloor_{CFG}$
**shows** $rv\ S\ n = rv\ S\ n'$
⟨proof⟩

**lemma** *closed-eq-obs-eq-rvs'*:
**fixes** $S$ :: *'node SDG-node set*
**assumes** *obs-eq*:*obs-intra m* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = *obs-intra m'* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
**shows** *rv S* (*CFG-node m*) = *rv S* (*CFG-node m'*)
⟨*proof*⟩

**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-kinds S* (*a#as*)) *s*
**and** *preds* (*slice-kinds S* (*ax#asx*)) *s'*
**and** *length s* = *length s'* **and** *snd* (*hd s*) = *snd* (*hd s'*)
**and** ∀ *V*∈*rv S* (*CFG-node* (*sourcenode a*)). *state-val s V* = *state-val s' V*
**shows** *False*
⟨*proof*⟩

**lemma** *rv-edge-slice-kinds*:
**assumes** *valid-edge a* **and** *intra-kind* (*kind a*)
**and** ∀ *V*∈*rv S* (*CFG-node* (*sourcenode a*)). *state-val s V* = *state-val s' V*
**and** *preds* (*slice-kinds S* (*a#as*)) *s* **and** *preds* (*slice-kinds S* (*a#asx*)) *s'*
**shows** ∀ *V*∈*rv S* (*CFG-node* (*targetnode a*)).
*state-val* (*transfer* (*slice-kind S a*) *s*) *V* =
*state-val* (*transfer* (*slice-kind S a*) *s'*) *V*
⟨*proof*⟩

### 1.13.4 The weak simulation relational set *WS*

**inductive-set** *WS* :: *'node SDG-node set* ⇒ ((*'node list* × ((*'var* ⇀ *'val*) × *'ret*) *list*) ×
(*'node list* × ((*'var* ⇀ *'val*) × *'ret*) *list*)) *set*
**for** *S* :: *'node SDG-node set*
**where** *WSI*: ⟦∀ *m* ∈ *set ms*. *valid-node m*; ∀ *m'* ∈ *set ms'*. *valid-node m'*;
*length ms* = *length s*; *length ms'* = *length s'*; *s* ≠ []; *s'* ≠ []; *ms* = *msx*@*mx*#*tl ms'*;
*get-proc mx* = *get-proc* (*hd ms'*);
∀ *m* ∈ *set* (*tl ms'*). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$;
*msx* ≠ [] ⟶ (∃ *mx'*. *call-of-return-node mx mx'* ∧ *mx'* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$);
∀ *i* < *length ms'*. *snd* (*s*!(*length msx* + *i*)) = *snd* (*s'*!*i*);
∀ *m* ∈ *set* (*tl ms*). *return-node m*;
∀ *i* < *length ms'*. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx#tl ms'*)!*i*)).
(*fst* (*s*!(*length msx* + *i*))) *V* = (*fst* (*s'*!*i*)) *V*;
*obs ms* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = *obs ms'* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$⟧
⟹ ((*ms*,*s*),(*ms'*,*s'*)) ∈ *WS S*

**lemma** *WS-silent-move*:
**assumes** $S,kind \vdash (ms_1,s_1) -a\rightarrow_\tau (ms_1',s_1')$ **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*:
$[\![S,kind \vdash (ms_1,s_1) =as\Rightarrow_\tau (ms_1',s_1'); ((ms_1,s_1),(ms_2,s_2)) \in WS\ S]\!]$
$\Longrightarrow ((ms_1',s_1'),(ms_2,s_2)) \in WS\ S$
⟨*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** $((ms_1',s_1'),(ms_1',transfer\ (slice\text{-}kind\ S\ a)\ s_2)) \in WS\ S$
**and** $S,slice\text{-}kind\ S \vdash (ms_2,s_2) =as@[a]\Rightarrow (ms_1',transfer\ (slice\text{-}kind\ S\ a)\ s_2)$
⟨*proof*⟩

### 1.13.5 The weak simulation

**definition** *is-weak-sim* ::
$(('node\ list \times (('var \rightharpoonup 'val) \times 'ret)\ list) \times$
$('node\ list \times (('var \rightharpoonup 'val) \times 'ret)\ list))\ set \Rightarrow 'node\ SDG\text{-}node\ set \Rightarrow bool$
**where** $is\text{-}weak\text{-}sim\ R\ S \equiv$
$\forall ms_1\ s_1\ ms_2\ s_2\ ms_1'\ s_1'\ as.$
$((ms_1,s_1),(ms_2,s_2)) \in R \wedge S,kind \vdash (ms_1,s_1) =as\Rightarrow (ms_1',s_1') \wedge s_1' \neq []$
$\longrightarrow (\exists ms_2'\ s_2'\ as'.\ ((ms_1',s_1'),(ms_2',s_2')) \in R \wedge$
$S,slice\text{-}kind\ S \vdash (ms_2,s_2) =as'\Rightarrow (ms_2',s_2'))$

**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** $((ms_1',s_1'),(ms_1',transfer\ (slice\text{-}kind\ S\ (last\ as))\ s_2)) \in WS\ S$
**and** $S,slice\text{-}kind\ S \vdash (ms_2,s_2) =as'@[last\ as]\Rightarrow$
$(ms_1',transfer\ (slice\text{-}kind\ S\ (last\ as))\ s_2)$
⟨*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*:
$S,f \vdash (ms,s) -a\rightarrow (ms',s')$
$\Longrightarrow \forall mx \in set\ (tl\ ms').\ \exists mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
⟨*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** $\forall i < length\ rs.\ rs!i \in get\text{-}return\text{-}edges\ (cs!i)$ **and** *valid-return-list rs m*
**and** $length\ rs = length\ cs$ **and** $ms = targetnodes\ rs$
**obtains** $cs'\ rs'$ **where** $valid\text{-}node\ m'$ **and** $valid\text{-}call\text{-}list\ cs'\ m'$
**and** $\forall i < length\ rs'.\ rs'!i \in get\text{-}return\text{-}edges\ (cs'!i)$
**and** $valid\text{-}return\text{-}list\ rs'\ m'$ **and** $length\ rs' = length\ cs'$
**and** $ms' = targetnodes\ rs'$ **and** $upd\text{-}cs\ cs\ [a] = cs'$
⟨*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** $\forall i < length\ rs.\ rs!i \in get\text{-}return\text{-}edges\ (cs!i)$ **and** *valid-return-list rs m*
**and** $length\ rs = length\ cs$ **and** $ms = targetnodes\ rs$
**obtains** $cs'\ rs'$ **where** $valid\text{-}node\ m'$ **and** $valid\text{-}call\text{-}list\ cs'\ m'$
**and** $\forall i < length\ rs'.\ rs'!i \in get\text{-}return\text{-}edges\ (cs'!i)$
**and** $valid\text{-}return\text{-}list\ rs'\ m'$ **and** $length\ rs' = length\ cs'$
**and** $ms' = targetnodes\ rs'$ **and** $upd\text{-}cs\ cs\ as = cs'$
⟨*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** $\forall i < length\ rs.\ rs!i \in get\text{-}return\text{-}edges\ (cs!i)$ **and** *valid-return-list rs m*
**and** $length\ rs = length\ cs$ **and** $ms = targetnodes\ rs$
**obtains** $cs'\ rs'$ **where** $valid\text{-}node\ m'$ **and** $valid\text{-}call\text{-}list\ cs'\ m'$
**and** $\forall i < length\ rs'.\ rs'!i \in get\text{-}return\text{-}edges\ (cs'!i)$
**and** $valid\text{-}return\text{-}list\ rs'\ m'$ **and** $length\ rs' = length\ cs'$
**and** $ms' = targetnodes\ rs'$ **and** $upd\text{-}cs\ cs\ [a] = cs'$
⟨*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** $\forall i < length\ rs.\ rs!i \in get\text{-}return\text{-}edges\ (cs!i)$ **and** *valid-return-list rs m*
**and** $length\ rs = length\ cs$ **and** $ms = targetnodes\ rs$
**obtains** $cs'\ rs'$ **where** *valid-node m'* **and** *valid-call-list cs' m'*
**and** $\forall i < length\ rs'.\ rs'!i \in get\text{-}return\text{-}edges\ (cs'!i)$
**and** *valid-return-list rs' m'* **and** $length\ rs' = length\ cs'$
**and** $ms' = targetnodes\ rs'$ **and** $upd\text{-}cs\ cs\ as = cs'$
⟨*proof*⟩

**lemma** *silent-moves-slpa-path*:
$[\![S,f \vdash (m\#ms''@ms,s) =as\Rightarrow_\tau (m'\#ms',s');\ valid\text{-}node\ m;\ valid\text{-}call\text{-}list\ cs\ m;$
$\forall i < length\ rs.\ rs!i \in get\text{-}return\text{-}edges\ (cs!i);\ valid\text{-}return\text{-}list\ rs\ m;$
$length\ rs = length\ cs;\ ms'' = targetnodes\ rs;$
$\forall mx \in set\ ms.\ \exists mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \in \lfloor HRB\text{-}slice\ S\rfloor_{CFG};$
$ms'' \neq [] \longrightarrow (\exists mx'.\ call\text{-}of\text{-}return\text{-}node\ (last\ ms'')\ mx' \wedge mx' \notin \lfloor HRB\text{-}slice\ S\rfloor_{CFG});$
$\forall mx \in set\ ms'.\ \exists mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \in \lfloor HRB\text{-}slice\ S\rfloor_{CFG}]\!]$
$\Longrightarrow same\text{-}level\text{-}path\text{-}aux\ cs\ as \wedge upd\text{-}cs\ cs\ as = [] \wedge m\ -as\rightarrow*\ m' \wedge ms = ms'$
⟨*proof*⟩

**lemma** *silent-moves-slp*:
$[\![S,f \vdash (m\#ms,s) =as\Rightarrow_\tau (m'\#ms',s');\ valid\text{-}node\ m;$
$\forall mx \in set\ ms.\ \exists mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \in \lfloor HRB\text{-}slice\ S\rfloor_{CFG};$
$\forall mx \in set\ ms'.\ \exists mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \in \lfloor HRB\text{-}slice\ S\rfloor_{CFG}]\!]$
$\Longrightarrow m\ -as\rightarrow_{sl}*\ m' \wedge ms = ms'$
⟨*proof*⟩

**lemma** *slpa-silent-moves-callstacks-eq*:
$[\![same\text{-}level\text{-}path\text{-}aux\ cs\ as;\ S,f \vdash (m\#msx@ms,s) =as\Rightarrow_\tau (m'\#ms',s');$
$length\ ms = length\ ms';\ valid\text{-}call\text{-}list\ cs\ m;$
$\forall i < length\ rs.\ rs!i \in get\text{-}return\text{-}edges\ (cs!i);\ valid\text{-}return\text{-}list\ rs\ m;$
$length\ rs = length\ cs;\ msx = targetnodes\ rs]\!]$
$\Longrightarrow ms = ms'$
⟨*proof*⟩

**lemma** *silent-moves-same-level-path*:
**assumes** $S,kind \vdash (m\#ms,s) =as\Rightarrow_\tau (m'\#ms',s')$ **and** $m\ -as\rightarrow_{sl}*\ m'$ **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* $\in$ *set ms*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$
*mx'* $\in$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
**and** *rest*:$\forall$ $i$ $<$ *length rs*. *rs*!$i$ $\in$ *get-return-edges* (*cs*!$i$)
*ms* = *targetnodes rs valid-return-list rs m length rs* = *length cs*
**obtains** *as' a as''* **where** *as* = *as'@a#as''* **and** $\exists$ $Q$ $r$ $p$ *fs*. *kind a* = $Q$:$r\hookrightarrow_p$*fs*
**and** *call-of-return-node* (*hd ms'*) (*sourcenode a*)
**and** *targetnode a* $-as''\rightarrow_{sl}*$ $m'$
| *ms'* = *ms*
⟨*proof*⟩

**lemma** *silent-moves-called-node-in-slice1-hd-nodestack-in-slice1*:
**assumes** $S$,*kind* $\vdash$ $(m\#ms,s)$ $=as\Rightarrow_\tau$ $(m'\#ms',s')$ **and** *valid-node m*
**and** *CFG-node m'* $\in$ *sum-SDG-slice1 nx*
**and** $\forall$ *mx* $\in$ *set ms*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$
*mx'* $\in$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
**and** $\forall$ $i$ $<$ *length rs*. *rs*!$i$ $\in$ *get-return-edges* (*cs*!$i$) **and** *ms* = *targetnodes rs*
**and** *valid-return-list rs m* **and** *length rs* = *length cs*
**obtains** *as' a as''* **where** *as* = *as'@a#as''* **and** $\exists$ $Q$ $r$ $p$ *fs*. *kind a* = $Q$:$r\hookrightarrow_p$*fs*
**and** *call-of-return-node* (*hd ms'*) (*sourcenode a*)
**and** *targetnode a* $-as''\rightarrow_{sl}*$ $m'$ **and** *CFG-node* (*sourcenode a*) $\in$ *sum-SDG-slice1*
*nx*
| *ms'* = *ms*
⟨*proof*⟩

**lemma** *silent-moves-called-node-in-slice1-nodestack-in-slice1*:
⟦$S$,*kind* $\vdash$ $(m\#ms,s)$ $=as\Rightarrow_\tau$ $(m'\#ms',s')$; *valid-node m*;
*CFG-node m'* $\in$ *sum-SDG-slice1 nx*; *nx* $\in$ $S$;
$\forall$ *mx* $\in$ *set ms*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$ *mx'* $\in$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$;
$\forall$ $i$ $<$ *length rs*. *rs*!$i$ $\in$ *get-return-edges* (*cs*!$i$); *ms* = *targetnodes rs*;
*valid-return-list rs m*; *length rs* = *length cs*⟧
$\Longrightarrow$ $\forall$ *mx* $\in$ *set ms'*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$ *mx'* $\in$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
⟨*proof*⟩

**lemma** *silent-moves-slice-intra-path*:
**assumes** $S$,*slice-kind S* $\vdash$ $(m\#ms,s)$ $=as\Rightarrow_\tau$ $(m'\#ms',s')$
**and** $\forall$ *mx* $\in$ *set ms*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$ *mx'* $\in$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
**shows** $\forall$ $a$ $\in$ *set as*. *intra-kind* (*kind a*)
⟨*proof*⟩

**lemma** *silent-moves-slice-keeps-state*:
**assumes** $S$,*slice-kind S* $\vdash$ $(m\#ms,s)$ $=as\Rightarrow_\tau$ $(m'\#ms',s')$
**and** $\forall$ *mx* $\in$ *set ms*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$ *mx'* $\in$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
**shows** $s = s'$

⟨*proof*⟩

### 1.14.2 Definition of *slice-edges*

**definition** *slice-edge* :: *'node SDG-node set* ⇒ *'edge list* ⇒ *'edge* ⇒ *bool*
**where** *slice-edge S cs a* ≡ (∀ *c* ∈ *set cs*. *sourcenode c* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$) ∧
(*case* (*kind a*) *of* $Q\hookleftarrow_{p}f$ ⇒ *True* | - ⇒ *sourcenode a* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$)

**lemma** *silent-move-no-slice-edge*:
⟦*S*,*f* ⊢ (*ms*,*s*) $-a\rightarrow_{\tau}$ (*ms'*,*s'*); *tl ms* = *targetnodes rs*; *length rs* = *length cs*;
∀ *i*<*length cs*. *call-of-return-node* (*tl ms*!*i*) (*sourcenode* (*cs*!*i*))⟧
⟹ ¬ *slice-edge S cs a*
⟨*proof*⟩

**lemma** *observable-move-slice-edge*:
⟦*S*,*f* ⊢ (*ms*,*s*) $-a\rightarrow$ (*ms'*,*s'*); *tl ms* = *targetnodes rs*; *length rs* = *length cs*;
∀ *i*<*length cs*. *call-of-return-node* (*tl ms*!*i*) (*sourcenode* (*cs*!*i*))⟧
⟹ *slice-edge S cs a*
⟨*proof*⟩

**function** *slice-edges* :: *'node SDG-node set* ⇒ *'edge list* ⇒ *'edge list* ⇒ *'edge list*
**where** *slice-edges S cs* [] = []
| *slice-edge S cs a* ⟹
*slice-edges S cs* (*a*#*as*) = *a*#*slice-edges S* (*upd-cs cs* [*a*]) *as*
| ¬ *slice-edge S cs a* ⟹
*slice-edges S cs* (*a*#*as*) = *slice-edges S* (*upd-cs cs* [*a*]) *as*
⟨*proof*⟩
**termination** ⟨*proof*⟩

**lemma** *slice-edges-Append*:
⟦*slice-edges S cs as* = *as'*; *slice-edges S* (*upd-cs cs as*) *asx* = *asx'*⟧
⟹ *slice-edges S cs* (*as*@*asx*) = *as'*@*asx'*
⟨*proof*⟩

**lemma** *slice-edges-Nil-split*:
*slice-edges S cs* (*as*@*as'*) = []
⟹ *slice-edges S cs as* = [] ∧ *slice-edges S* (*upd-cs cs as*) *as'* = []
⟨*proof*⟩

**lemma** *slice-intra-edges-no-nodes-in-slice*:
⟦*slice-edges S cs as* = []; ∀ *a* ∈ *set as*. *intra-kind* (*kind a*);
∀ *c* ∈ *set cs*. *sourcenode c* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$⟧

$\Longrightarrow \forall\, nx \in set(sourcenodes\ as).\ nx \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
⟨proof⟩

**lemma** *silent-moves-no-slice-edges*:
$[\![S,f \vdash (ms,s) =as\Rightarrow_\tau (ms',s');\ tl\ ms = targetnodes\ rs;\ length\ rs = length\ cs;$
$\forall\, i<length\ cs.\ call\text{-}of\text{-}return\text{-}node\ (tl\ ms!i)\ (sourcenode\ (cs!i))]\!]$
$\Longrightarrow slice\text{-}edges\ S\ cs\ as = [] \wedge (\exists\, rs'.\ tl\ ms' = targetnodes\ rs' \wedge$
$length\ rs' = length\ (upd\text{-}cs\ cs\ as) \wedge (\forall\, i<length\ (upd\text{-}cs\ cs\ as).$
$call\text{-}of\text{-}return\text{-}node\ (tl\ ms'!i)\ (sourcenode\ ((upd\text{-}cs\ cs\ as)!i))))$
⟨proof⟩

**lemma** *observable-moves-singular-slice-edge*:
$[\![S,f \vdash (ms,s) =as\Rightarrow (ms',s');\ tl\ ms = targetnodes\ rs;\ length\ rs = length\ cs;$
$\forall\, i<length\ cs.\ call\text{-}of\text{-}return\text{-}node\ (tl\ ms!i)\ (sourcenode\ (cs!i))]\!]$
$\Longrightarrow slice\text{-}edges\ S\ cs\ as = [last\ as]$
⟨proof⟩

**lemma** *silent-moves-nonempty-nodestack-False*:
**assumes** $S,kind \vdash ([m],[cf]) =as\Rightarrow_\tau (m'\#ms',s')$ **and** $valid\text{-}node\ m$
**and** $ms' \neq []$ **and** $CFG\text{-}node\ m' \in sum\text{-}SDG\text{-}slice1\ nx$ **and** $nx \in S$
**shows** *False*
⟨proof⟩

**lemma** *transfers-intra-slice-kinds-slice-edges*:
$[\![\forall\, a \in set\ as.\ intra\text{-}kind\ (kind\ a);\ \forall\, c \in set\ cs.\ sourcenode\ c \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}]\!]$
$\Longrightarrow transfers\ (slice\text{-}kinds\ S\ (slice\text{-}edges\ S\ cs\ as))\ s =$
$transfers\ (slice\text{-}kinds\ S\ as)\ s$
⟨proof⟩

**lemma** *exists-sliced-intra-path-preds*:
**assumes** $m -as\rightarrow_\iota* m'$ **and** $slice\text{-}edges\ S\ cs\ as = []$
**and** $m' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **and** $\forall\, c \in set\ cs.\ sourcenode\ c \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
**obtains** $as'$ **where** $m -as'\rightarrow_\iota* m'$ **and** $preds\ (slice\text{-}kinds\ S\ as')\ (cf\#cfs)$
**and** $slice\text{-}edges\ S\ cs\ as' = []$
⟨proof⟩

**lemma** *slp-to-intra-path-with-slice-edges*:
**assumes** $n -as\rightarrow_{sl}* n'$ **and** $slice\text{-}edges\ S\ cs\ as = []$
**obtains** $as'$ **where** $n -as'\rightarrow_\iota* n'$ **and** $slice\text{-}edges\ S\ cs\ as' = []$
⟨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* ::
$'node$ *SDG-node set* $\Rightarrow$ $('edge \Rightarrow ('var,'val,'ret,'pname)$ *edge-kind*$) \Rightarrow$ $'node$ *list* $\Rightarrow$
$(('var \rightharpoonup 'val) \times 'ret)$ *list* $\Rightarrow$ $'edge$ *list* $\Rightarrow$ $'node$ *list* $\Rightarrow$
$(('var \rightharpoonup 'val) \times 'ret)$ *list* $\Rightarrow$ *bool*
(‹-,- ⊢ '(-,-') =-⇒∗ '(-,-')› [51,50,0,0,50,0,0] 51)

**where** *tom-Nil*:
*length ms* = *length s* $\Longrightarrow$ $S,f \vdash (ms,s) =[]\Rightarrow* (ms,s)$

| *tom-Cons*:
$[\![S,f \vdash (ms,s) =as\Rightarrow (ms',s'); S,f \vdash (ms',s') =as'\Rightarrow* (ms'',s'')]\!]$
$\Longrightarrow S,f \vdash (ms,s) =(last\ as)\#as'\Rightarrow* (ms'',s'')$

**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** $S,f \vdash (ms'',s'') =asx'\Rightarrow (ms',s')$
⟨*proof*⟩

**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** $\forall\, i < length\ rs.\ rs!i \in$ *get-return-edges* $(cs!i)$
**and** *valid-return-list rs m* **and** *length rs* = *length cs* **and** *ms* = *targetnodes rs*
**obtains** *cs' rs'* **where** *valid-node m'* **and** *valid-call-list cs' m'*
**and** $\forall\, i < length\ rs'.\ rs'!i \in$ *get-return-edges* $(cs'!i)$
**and** *valid-return-list rs' m'* **and** *length rs'* = *length cs'*
**and** *ms'* = *targetnodes rs'*
⟨*proof*⟩

**lemma** *vpa-trans-observable-moves*:
**assumes** *valid-path-aux cs as* **and** $m\ -as\rightarrow*\ m'$ **and** *preds* (*kinds as*) *s*
**and** *transfers* (*kinds as*) *s* = *s'* **and** *valid-call-list cs m*
**and** $\forall\, i < length\ rs.\ rs!i \in$ *get-return-edges* $(cs!i)$
**and** *valid-return-list rs m*
**and** *length rs* = *length cs* **and** *length s* = *Suc* (*length cs*)
**obtains** *ms ms'' s'' ms' as' as''*
**where** $S,kind \vdash (m\#ms,s) =$*slice-edges S cs as*$\Rightarrow* (ms'',s'')$
**and** $S,kind \vdash (ms'',s'') =as'\Rightarrow_\tau (m'\#ms',s')$
**and** *ms* = *targetnodes rs* **and** *length ms* = *length cs*
**and** $\forall\, i<length\ cs.$ *call-of-return-node* $(ms!i)$ (*sourcenode* $(cs!i)$)
**and** *slice-edges S cs as* = *slice-edges S cs as''*

**and** $m$ $-as''@as'\rightarrow*$ $m'$ **and** *valid-path-aux cs* $(as''@as')$
⟨*proof*⟩

**lemma** *valid-path-trans-observable-moves*:
**assumes** $m$ $-as\rightarrow_{\surd}*$ $m'$ **and** *preds* (*kinds as*) [*cf*]
**and** *transfers* (*kinds as*) [*cf*] = $s'$
**obtains** $ms''$ $s''$ $ms'$ $as'$ $as''$
**where** $S$,*kind* ⊢ ([$m$],[*cf*]) =*slice-edges S* [] *as*⇒∗ ($ms''$,$s''$)
**and** $S$,*kind* ⊢ ($ms''$,$s''$) =$as'$⇒$_\tau$ ($m'$#$ms'$,$s'$)
**and** *slice-edges S* [] *as* = *slice-edges S* [] $as''$
**and** $m$ $-as''@as'\rightarrow_{\surd}*$ $m'$
⟨*proof*⟩

**lemma** *WS-weak-sim-trans*:
**assumes** $((ms_1,s_1),(ms_2,s_2)) \in$ *WS S*
**and** $S$,*kind* ⊢ $(ms_1,s_1)$ =*as*⇒∗ $(ms_1',s_1')$ **and** *as* ≠ []
**shows** $((ms_1',s_1'),(ms_1'$,*transfers* (*slice-kinds S as*) $s_2)) \in$ *WS S* ∧
$S$,*slice-kind S* ⊢ $(ms_2,s_2)$ =*as*⇒∗ $(ms_1'$,*transfers* (*slice-kinds S as*) $s_2)$
⟨*proof*⟩

**lemma** *stacks-rewrite*:
**assumes** *valid-call-list cs m* **and** *valid-return-list rs m*
**and** ∀ *i* < *length rs*. *rs*!*i* ∈ *get-return-edges* (*cs*!*i*)
**and** *length rs* = *length cs* **and** *ms* = *targetnodes rs*
**shows** ∀ *i*<*length cs*. *call-of-return-node* (*ms*!*i*) (*sourcenode* (*cs*!*i*))
⟨*proof*⟩

**lemma** *slice-tom-preds-vp*:
**assumes** $S$,*slice-kind S* ⊢ ($m$#*ms*,*s*) =*as*⇒∗ ($m'$#$ms'$,$s'$) **and** *valid-node m*
**and** *valid-call-list cs m* **and** ∀ *i* < *length rs*. *rs*!*i* ∈ *get-return-edges* (*cs*!*i*)
**and** *valid-return-list rs m* **and** *length rs* = *length cs* **and** *ms* = *targetnodes rs*
**and** ∀ *mx* ∈ *set ms*. ∃ $mx'$. *call-of-return-node mx* $mx'$ ∧ $mx' \in \lfloor$*HRB-slice S*$\rfloor_{CFG}$
**obtains** $as'$ $cs'$ $rs'$ **where** *preds* (*slice-kinds S* $as'$) *s*
**and** *slice-edges S cs* $as'$ = *as* **and** $m$ $-as'\rightarrow*$ $m'$ **and** *valid-path-aux cs* $as'$
**and** *upd-cs cs* $as'$ = $cs'$ **and** *valid-node* $m'$ **and** *valid-call-list* $cs'$ $m'$
**and** ∀ *i* < *length* $rs'$. $rs'$!*i* ∈ *get-return-edges* ($cs'$!*i*)
**and** *valid-return-list* $rs'$ $m'$ **and** *length* $rs'$ = *length* $cs'$
**and** $ms'$ = *targetnodes* $rs'$ **and** *transfers* (*slice-kinds S* $as'$) *s* ≠ []
**and** *transfers* (*slice-kinds S* (*slice-edges S cs* $as'$)) *s* =
*transfers* (*slice-kinds S* $as'$) *s*
⟨*proof*⟩

### 1.14.4 The fundamental property of static interprocedural slicing

**theorem** *fundamental-property-of-static-slicing*:
  **assumes** $m -as\rightarrow_{\surd}* m'$ **and** *preds* (*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** $m -as'\rightarrow_{\surd}* m'$
⟨*proof*⟩

**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* ⇒ *'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* ⇒ *'pname*
  **and** *get-return-edges* :: *'edge* ⇒ *'edge set*
  **and** *procs* :: (*'pname* × *'var list* × *'var list*) *list* **and** *Main* :: *'pname*
  **and** *Exit*::*'node* (‹'('-Exit'-')›)
  **and** *Def* :: *'node* ⇒ *'var set* **and** *Use* :: *'node* ⇒ *'var set*
  **and** *ParamDefs* :: *'node* ⇒ *'var list* **and** *ParamUses* :: *'node* ⇒ *'var set list*
  **and** *sem* :: *'com* ⇒ (*'var* ⇀ *'val*) *list* ⇒ *'com* ⇒ (*'var* ⇀ *'val*) *list* ⇒ *bool*
  (‹((*1*⟨-,/-⟩) ⇒/ (*1*⟨-,/-⟩))› [*0*,*0*,*0*,*0*] *81*)
  **and** *identifies* :: *'node* ⇒ *'com* ⇒ *bool* (‹- ≜ -› [*51*,*0*] *80*)
**begin**

**theorem** *fundamental-property-of-path-slicing-semantically*:
  **assumes** *m* ≜ *c* **and** ⟨*c*,[*cf*]⟩ ⇒ ⟨*c'*,*s'*⟩
  **obtains** *m' as cfs'* **where** $m -as\rightarrow_{\surd}* m'$ **and** *m'* ≜ *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'*
⟨*proof*⟩

**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* $v_1$ $v_2$ = *Some*(*Bool*($v_1$ = $v_2$))
| *binop And* (*Bool* $b_1$) (*Bool* $b_2$) = *Some*(*Bool*($b_1 \wedge b_2$))
| *binop Less* (*Intg* $i_1$) (*Intg* $i_2$) = *Some*(*Bool*($i_1 < i_2$))
| *binop Add* (*Intg* $i_1$) (*Intg* $i_2$) = *Some*(*Intg*($i_1 + i_2$))
| *binop Sub* (*Intg* $i_1$) (*Intg* $i_2$) = *Some*(*Intg*($i_1 - i_2$))
| *binop bop* $v_1$ $v_2$ = *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
| #:($c_1$;;$c_2$) = #:$c_1$ + #:$c_2$
| #:(*if* (*b*) $c_1$ *else* $c_2$) = #:$c_1$ + #:$c_2$ + 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* ($e_1$«*bop*»$e_2$) *cf* =
(*case interpret* $e_1$ *cf of None* ⇒ *None*
| *Some* $v_1$ ⇒ (*case interpret* $e_2$ *cf of None* ⇒ *None*
| *Some* $v_2$ ⇒ (

*case binop bop* $v_1$ $v_2$ *of None* ⇒ *None* | *Some v* ⇒ *Some v*)))

**abbreviation** *update* :: (*vname* ⇀ *val*) ⇒ *vname* ⇒ *expr* ⇒ (*vname* ⇀ *val*)
**where** *update cf V e* ≡ *cf*(*V*:=(*interpret e cf*))

**abbreviation** *state-check* :: (*vname* ⇀ *val*) ⇒ *expr* ⇒ *val option* ⇒ *bool*
**where** *state-check cf b v* ≡ (*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 ⊕

**fun** *label-incr* :: *label* ⇒ *nat* ⇒ *label* (‹- ⊕ -› *60*)
**where** (*Label l*) ⊕ *i* = *Label* (*l* + *i*)
| *Entry* ⊕ *i* = *Entry*
| *Exit* ⊕ *i* = *Exit*

**lemma** *Exit-label-incr* [*dest*]: *Exit* = *n* ⊕ *i* ⟹ *n* = *Exit*
⟨*proof*⟩

**lemma** *label-incr-Exit* [*dest*]: *n* ⊕ *i* = *Exit* ⟹ *n* = *Exit*
⟨*proof*⟩

**lemma** *Entry-label-incr* [*dest*]: *Entry* = *n* ⊕ *i* ⟹ *n* = *Entry*
⟨*proof*⟩

**lemma** *label-incr-Entry* [*dest*]: *n* ⊕ *i* = *Entry* ⟹ *n* = *Entry*
⟨*proof*⟩

**lemma** *label-incr-inj*:
*n* ⊕ *c* = *n'* ⊕ *c* ⟹ *n* = *n'*
⟨*proof*⟩

**lemma** *label-incr-simp*:*n* ⊕ *i* = *m* ⊕ (*i* + *j*) ⟹ *n* = *m* ⊕ *j*
⟨*proof*⟩

**lemma** *label-incr-simp-rev*:$m \oplus (j + i) = n \oplus i \Longrightarrow m \oplus j = n$
⟨*proof*⟩

**lemma** *label-incr-start-Node-smaller*:
$Label\ l = n \oplus i \Longrightarrow n = Label\ (l - i)$
⟨*proof*⟩

**lemma** *label-incr-start-Node-smaller-rev*:
$n \oplus i = Label\ l \Longrightarrow n = Label\ (l - i)$
⟨*proof*⟩

**lemma** *label-incr-ge*:$Label\ l = n \oplus i \Longrightarrow l \geq i$
⟨*proof*⟩

**lemma** *label-incr-0* [*dest*]:
$[\![Label\ 0 = n \oplus i;\ i > 0]\!] \Longrightarrow False$
⟨*proof*⟩

**lemma** *label-incr-0-rev* [*dest*]:
$[\![n \oplus i = Label\ 0;\ i > 0]\!] \Longrightarrow False$
⟨*proof*⟩

### The edges of the procedure CFG

Control flow information in this language is the node, to which we return after the calles 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*:
$V{:=}e \vdash \textit{Label 1} \ -\textit{IEdge} \Uparrow id\rightarrow_p \textit{Exit}$

| *Proc-CFG-SeqFirst*:
$[\![c_1 \vdash n \ -et\rightarrow_p n'; n' \neq \textit{Exit}]\!] \Longrightarrow c_1;;c_2 \vdash n \ -et\rightarrow_p n'$

| *Proc-CFG-SeqConnect*:
$[\![c_1 \vdash n \ -et\rightarrow_p \textit{Exit}; n \neq \textit{Entry}]\!] \Longrightarrow c_1;;c_2 \vdash n \ -et\rightarrow_p \textit{Label} \ \#{:}c_1$

| *Proc-CFG-SeqSecond*:
$[\![c_2 \vdash n \ -et\rightarrow_p n'; n \neq \textit{Entry}]\!] \Longrightarrow c_1;;c_2 \vdash n \oplus \#{:}c_1 \ -et\rightarrow_p n' \oplus \#{:}c_1$

| *Proc-CFG-CondTrue*:
$\textit{if} \ (b) \ c_1 \ \textit{else} \ c_2 \vdash \textit{Label 0}$
$-\textit{IEdge} \ (\lambda cf. \ \textit{state-check cf b (Some true)})_{\surd}\rightarrow_p \textit{Label 1}$

| *Proc-CFG-CondFalse*:
$\textit{if} \ (b) \ c_1 \ \textit{else} \ c_2 \vdash \textit{Label 0} \ -\textit{IEdge} \ (\lambda cf. \ \textit{state-check cf b (Some false)})_{\surd}\rightarrow_p$
$\textit{Label} \ (\#{:}c_1 + 1)$

| *Proc-CFG-CondThen*:
$[\![c_1 \vdash n \ -et\rightarrow_p n'; n \neq \textit{Entry}]\!] \Longrightarrow \textit{if} \ (b) \ c_1 \ \textit{else} \ c_2 \vdash n \oplus 1 \ -et\rightarrow_p n' \oplus 1$

| *Proc-CFG-CondElse*:
$[\![c_2 \vdash n \ -et\rightarrow_p n'; n \neq \textit{Entry}]\!]$
$\Longrightarrow \textit{if} \ (b) \ c_1 \ \textit{else} \ c_2 \vdash n \oplus (\#{:}c_1 + 1) \ -et\rightarrow_p n' \oplus (\#{:}c_1 + 1)$

| *Proc-CFG-WhileTrue*:
$\textit{while} \ (b) \ c' \vdash \textit{Label 0} \ -\textit{IEdge} \ (\lambda cf. \ \textit{state-check cf b (Some true)})_{\surd}\rightarrow_p \textit{Label 2}$

| *Proc-CFG-WhileFalse*:
$\textit{while} \ (b) \ c' \vdash \textit{Label 0} \ -\textit{IEdge} \ (\lambda cf. \ \textit{state-check cf b (Some false)})_{\surd}\rightarrow_p \textit{Label 1}$

| *Proc-CFG-WhileFalseSkip*:
$\textit{while} \ (b) \ c' \vdash \textit{Label 1} \ -\textit{IEdge} \Uparrow id\rightarrow_p \textit{Exit}$

| *Proc-CFG-WhileBody*:
$[\![c' \vdash n \ -et\rightarrow_p n'; n \neq \textit{Entry}; n' \neq \textit{Exit}]\!]$
$\Longrightarrow \textit{while} \ (b) \ c' \vdash n \oplus 2 \ -et\rightarrow_p n' \oplus 2$

| *Proc-CFG-WhileBodyExit*:
$[\![c' \vdash n \ -et\rightarrow_p \textit{Exit}; n \neq \textit{Entry}]\!] \Longrightarrow \textit{while} \ (b) \ c' \vdash n \oplus 2 \ -et\rightarrow_p \textit{Label 0}$

| *Proc-CFG-Call*:
$\textit{Call p es rets} \vdash \textit{Label 0} \ -\textit{CEdge} \ (p,es,rets)\rightarrow_p \textit{Label 1}$

| *Proc-CFG-CallSkip*:
$\textit{Call p es rets} \vdash \textit{Label 1} \ -\textit{IEdge} \Uparrow id\rightarrow_p \textit{Exit}$

### Some lemmas about the procedure CFG

**lemma** *Proc-CFG-Exit-no-sourcenode* [*dest*]:
$prog \vdash Exit\ -et\rightarrow_p\ n' \Longrightarrow False$
⟨*proof*⟩

**lemma** *Proc-CFG-Entry-no-targetnode* [*dest*]:
$prog \vdash n\ -et\rightarrow_p\ Entry \Longrightarrow False$
⟨*proof*⟩

**lemma** *Proc-CFG-IEdge-intra-kind*:
$prog \vdash n\ -IEdge\ et\rightarrow_p\ n' \Longrightarrow intra\text{-}kind\ et$
⟨*proof*⟩

**lemma** [*dest*]:$prog \vdash n\ -IEdge\ (Q{:}r\hookrightarrow_p fs)\rightarrow_p\ n' \Longrightarrow False$
⟨*proof*⟩

**lemma** [*dest*]:$prog \vdash n\ -IEdge\ (Q\hookleftarrow_p f)\rightarrow_p\ n' \Longrightarrow False$
⟨*proof*⟩

**lemma** *Proc-CFG-sourcelabel-less-num-nodes*:
$prog \vdash Label\ l\ -et\rightarrow_p\ n' \Longrightarrow l < \#{:}prog$
⟨*proof*⟩

**lemma** *Proc-CFG-targetlabel-less-num-nodes*:
$prog \vdash n\ -et\rightarrow_p\ Label\ l \Longrightarrow l < \#{:}prog$
⟨*proof*⟩

**lemma** *Proc-CFG-EntryD*:
$prog \vdash Entry\ -et\rightarrow_p\ n'$
$\Longrightarrow (n' = Exit \wedge et = IEdge(\lambda s.\ False)_{\surd}) \vee (n' = Label\ 0 \wedge et = IEdge\ (\lambda s.\ True)_{\surd})$
⟨*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*:
$prog \vdash n\ -CEdge\ (p,es,rets)\rightarrow_p\ n' \Longrightarrow \exists\, l.\ n = Label\ l \wedge n' = Label\ (Suc\ l)$
⟨*proof*⟩

**lemma** *Proc-CFG-Call-target-0*:
$prog \vdash n\ {-}CEdge\ (p,es,rets){\rightarrow_p}\ Label\ 0 \Longrightarrow n = Entry$
⟨*proof*⟩

**lemma** *Proc-CFG-Call-Intra-edge-not-same-source*:
$[\![prog \vdash n\ {-}CEdge\ (p,es,rets){\rightarrow_p}\ n';\ prog \vdash n\ {-}IEdge\ et{\rightarrow_p}\ n'']\!] \Longrightarrow False$
⟨*proof*⟩

**lemma** *Proc-CFG-Call-Intra-edge-not-same-target*:
$[\![prog \vdash n\ {-}CEdge\ (p,es,rets){\rightarrow_p}\ n';\ prog \vdash n''\ {-}IEdge\ et{\rightarrow_p}\ n']\!] \Longrightarrow False$
⟨*proof*⟩

**lemma** *Proc-CFG-Call-nodes-eq*:
$[\![prog \vdash n\ {-}CEdge\ (p,es,rets){\rightarrow_p}\ n';\ prog \vdash n\ {-}CEdge\ (p',es',rets'){\rightarrow_p}\ n'']\!]$
$\Longrightarrow n' = n'' \wedge p = p' \wedge es = es' \wedge rets = rets'$
⟨*proof*⟩

**lemma** *Proc-CFG-Call-nodes-eq'*:
$[\![prog \vdash n\ {-}CEdge\ (p,es,rets){\rightarrow_p}\ n';\ prog \vdash n''\ {-}CEdge\ (p',es',rets'){\rightarrow_p}\ n']\!]$
$\Longrightarrow n = n'' \wedge p = p' \wedge es = es' \wedge rets = rets'$
⟨*proof*⟩

**lemma** *Proc-CFG-Call-targetnode-no-Call-sourcenode*:
$[\![prog \vdash n\ {-}CEdge\ (p,es,rets){\rightarrow_p}\ n';\ prog \vdash n'\ {-}CEdge\ (p',es',rets'){\rightarrow_p}\ n'']\!]$
$\Longrightarrow False$
⟨*proof*⟩

**lemma** *Proc-CFG-Call-follows-id-edge*:
$[\![prog \vdash n\ {-}CEdge\ (p,es,rets){\rightarrow_p}\ n';\ prog \vdash n'\ {-}IEdge\ et{\rightarrow_p}\ n'']\!] \Longrightarrow et = \Uparrow id$
⟨*proof*⟩

**lemma** *Proc-CFG-edge-det*:
$[\![prog \vdash n\ {-}et{\rightarrow_p}\ n';\ prog \vdash n\ {-}et'{\rightarrow_p}\ n']\!] \Longrightarrow et = et'$
⟨*proof*⟩

**lemma** *WCFG-deterministic*:
$[\![prog \vdash n_1\ {-}et_1{\rightarrow_p}\ n_1';\ prog \vdash n_2\ {-}et_2{\rightarrow_p}\ n_2';\ n_1 = n_2;\ n_1' \neq n_2']\!]$
$\Longrightarrow \exists\, Q\ Q'.\ et_1 = IEdge\ (Q)_{\surd} \wedge et_2 = IEdge\ (Q')_{\surd} \wedge$
$(\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$
⟨*proof*⟩

### 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** $containsCall\ procs\ Skip\ ps\ p \longleftrightarrow False$
$|\ containsCall\ procs\ (V{:=}e)\ ps\ p \longleftrightarrow False$
$|\ containsCall\ procs\ (c_1;;c_2)\ ps\ p \longleftrightarrow$
$containsCall\ procs\ c_1\ ps\ p \lor containsCall\ procs\ c_2\ ps\ p$
$|\ containsCall\ procs\ (if\ (b)\ c_1\ else\ c_2)\ ps\ p \longleftrightarrow$
$containsCall\ procs\ c_1\ ps\ p \lor containsCall\ procs\ c_2\ ps\ p$
$|\ containsCall\ procs\ (while\ (b)\ c)\ ps\ p \longleftrightarrow$
$containsCall\ procs\ c\ ps\ p$
$|\ containsCall\ procs\ (Call\ q\ es'\ rets')\ ps\ p \longleftrightarrow p = q \land ps = [] \lor$
$(\exists\, ins\ outs\ c\ ps'.\ ps = q\#ps' \land (q,ins,outs,c) \in set\ procs \land$
$containsCall\ procs\ c\ ps'\ p)$
⟨*proof*⟩
**termination** *containsCall*
⟨*proof*⟩

**lemmas** *containsCall-induct*[*case-names Skip LAss Seq Cond While Call*] =
*containsCall.induct*

**lemma** *containsCallcases*:
$containsCall\ procs\ prog\ ps\ p$
$\Longrightarrow ps = [] \land containsCall\ procs\ prog\ ps\ p \lor$
$(\exists\, q\ ins\ outs\ c\ ps'.\ ps = ps'@[q] \land (q,ins,outs,c) \in set\ procs \land$
$containsCall\ procs\ c\ []\ p \land containsCall\ procs\ prog\ ps'\ q)$
⟨*proof*⟩

**lemma** *containsCallE*:
$[\![containsCall\ procs\ prog\ ps\ p;$
$[\![ps = [];\ containsCall\ procs\ prog\ ps\ p]\!] \Longrightarrow P\ procs\ prog\ ps\ p;$
$\bigwedge q\ ins\ outs\ c\ es'\ rets'\ ps'.\ [\![ps = ps'@[q];\ (q,ins,outs,c) \in set\ procs;$
$containsCall\ procs\ c\ []\ p;\ containsCall\ procs\ prog\ ps'\ q]\!]$
$\Longrightarrow P\ procs\ prog\ ps\ p]\!] \Longrightarrow P\ procs\ prog\ ps\ p$

⟨*proof*⟩

**lemma** *containsCall-in-proc*:
⟦*containsCall procs prog qs q*; (*q*,*ins*,*outs*,*c*) ∈ *set procs*;
*containsCall procs c* [] *p*⟧
⟹ *containsCall procs prog* (*qs*@[*q*]) *p*
⟨*proof*⟩

**lemma** *containsCall-indirection*:
⟦*containsCall procs prog qs q*; *containsCall procs c ps p*;
(*q*,*ins*,*outs*,*c*) ∈ *set procs*⟧
⟹ *containsCall procs prog* (*qs*@*q*#*ps*) *p*
⟨*proof*⟩

**lemma** *Proc-CFG-Call-containsCall*:
*prog* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)$\rightarrow_p$ *n'* ⟹ *containsCall procs prog* [] *p*
⟨*proof*⟩

**lemma** *containsCall-empty-Proc-CFG-Call-edge*:
**assumes** *containsCall procs prog* [] *p*
**obtains** *l es rets l'* **where** *prog* ⊢ *Label l* −*CEdge* (*p*,*es*,*rets*)$\rightarrow_p$ *Label l'*
⟨*proof*⟩

## The edges of the combined CFG

**type-synonym** *node* = (*pname* × *label*)
**type-synonym** *edge* = (*node* × (*vname*,*val*,*node*,*pname*) *edge-kind* × *node*)

**fun** *get-proc* :: *node* ⇒ *pname*
**where** *get-proc* (*p*,*l*) = *p*

**inductive** *PCFG* ::
*cmd* ⇒ *procs* ⇒ *node* ⇒ (*vname*,*val*,*node*,*pname*) *edge-kind* ⇒ *node* ⇒ *bool*
(‹-,- ⊢ - −-→ -› [*51*,*51*,*0*,*0*,*0*] *81*)
**for** *prog*::*cmd* **and** *procs*::*procs*
**where**

*Main*:
*prog* ⊢ *n* −*IEdge et*$\rightarrow_p$ *n'* ⟹ *prog*,*procs* ⊢ (*Main*,*n*) −*et*→ (*Main*,*n'*)

| *Proc*:
⟦(*p*,*ins*,*outs*,*c*) ∈ *set procs*; *c* ⊢ *n* −*IEdge et*$\rightarrow_p$ *n'*;
*containsCall procs prog ps p*⟧
⟹ *prog*,*procs* ⊢ (*p*,*n*) −*et*→ (*p*,*n'*)

| *MainCall*:
⟦*prog* ⊢ *Label l* −*CEdge* (*p*,*es*,*rets*)$\rightarrow_p$ *n*′; (*p*,*ins*,*outs*,*c*) ∈ *set procs*⟧
⟹ *prog*,*procs* ⊢ (*Main*,*Label l*)
−(λ*s*. *True*):(*Main*,*n*′)$\hookrightarrow_p$*map* (λ*e cf*. *interpret e cf*) *es*→ (*p*,*Entry*)

| *ProcCall*:
⟦(*p*,*ins*,*outs*,*c*) ∈ *set procs*; *c* ⊢ *Label l* −*CEdge* (*p*′,*es*′,*rets*′)$\rightarrow_p$ *Label l*′;
(*p*′,*ins*′,*outs*′,*c*′) ∈ *set procs*; *containsCall procs prog ps p*⟧
⟹ *prog*,*procs* ⊢ (*p*,*Label l*)
−(λ*s*. *True*):(*p*,*Label l*′)$\hookrightarrow_{p'}$*map* (λ*e cf*. *interpret e cf*) *es*′→ (*p*′,*Entry*)

| *MainReturn*:
⟦*prog* ⊢ *Label l* −*CEdge* (*p*,*es*,*rets*)$\rightarrow_p$ *Label l*′; (*p*,*ins*,*outs*,*c*) ∈ *set procs*⟧
⟹ *prog*,*procs* ⊢ (*p*,*Exit*) −(λ*cf*. *snd cf* = (*Main*,*Label l*′))$\hookleftarrow_p$
(λ*cf cf*′. *cf*′(*rets* [:=] *map cf outs*))→ (*Main*,*Label l*′)

| *ProcReturn*:
⟦(*p*,*ins*,*outs*,*c*) ∈ *set procs*; *c* ⊢ *Label l* −*CEdge* (*p*′,*es*′,*rets*′)$\rightarrow_p$ *Label l*′;
(*p*′,*ins*′,*outs*′,*c*′) ∈ *set procs*; *containsCall procs prog ps p*⟧
⟹ *prog*,*procs* ⊢ (*p*′,*Exit*) −(λ*cf*. *snd cf* = (*p*,*Label l*′))$\hookleftarrow_{p'}$
(λ*cf cf*′. *cf*′(*rets*′ [:=] *map cf outs*′))→ (*p*,*Label l*′)

| *MainCallReturn*:
*prog* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)$\rightarrow_p$ *n*′
⟹ *prog*,*procs* ⊢ (*Main*,*n*) −(λ*s*. *False*)$_{\surd}$→ (*Main*,*n*′)

| *ProcCallReturn*:
⟦(*p*,*ins*,*outs*,*c*) ∈ *set procs*; *c* ⊢ *n* −*CEdge* (*p*′,*es*′,*rets*′)$\rightarrow_p$ *n*′;
*containsCall procs prog ps p*⟧
⟹ *prog*,*procs* ⊢ (*p*,*n*) −(λ*s*. *False*)$_{\surd}$→ (*p*,*n*′)

**end**

## 2.4 Well-formedness of programs

**theory** *WellFormProgs* **imports** *PCFG* **begin**

### 2.4.1 Well-formedness of procedure lists.

**definition** *wf-proc* :: *proc* ⇒ *bool*
**where** *wf-proc x* ≡ *let* (*p*,*ins*,*outs*,*c*) = *x in*
*p* ≠ *Main* ∧ *distinct ins* ∧ *distinct outs*

**definition** *well-formed* :: *procs* ⇒ *bool*
**where** *well-formed procs* ≡ *distinct-fst procs* ∧
(∀ (*p*,*ins*,*outs*,*c*) ∈ *set procs*. *wf-proc* (*p*,*ins*,*outs*,*c*))

**lemma** [*dest*]:$[\![$*well-formed procs*; $(Main,ins,outs,c) \in$ *set procs*$]\!] \Longrightarrow$ *False*
⟨*proof*⟩

**lemma** *well-formed-same-procs* [*dest*]:
$[\![$*well-formed procs*; $(p,ins,outs,c) \in$ *set procs*; $(p,ins',outs',c') \in$ *set procs*$]\!]$
$\Longrightarrow ins = ins' \wedge outs = outs' \wedge c = c'$
⟨*proof*⟩

**lemma** *PCFG-sourcelabel-None-less-num-nodes*:
$[\![prog,procs \vdash (Main,Label\ l)\ -et\rightarrow\ n'$; *well-formed procs*$]\!] \Longrightarrow l < \#:prog$
⟨*proof*⟩

**lemma** *Proc-CFG-sourcelabel-Some-less-num-nodes*:
$[\![prog,procs \vdash (p,Label\ l)\ -et\rightarrow\ n'$; $(p,ins,outs,c) \in$ *set procs*;
*well-formed procs*$]\!] \Longrightarrow l < \#:c$
⟨*proof*⟩

**lemma** *Proc-CFG-targetlabel-Main-less-num-nodes*:
$[\![prog,procs \vdash n\ -et\rightarrow\ (Main,Label\ l)$; *well-formed procs*$]\!] \Longrightarrow l < \#:prog$
⟨*proof*⟩

**lemma** *Proc-CFG-targetlabel-Some-less-num-nodes*:
$[\![prog,procs \vdash n\ -et\rightarrow\ (p,Label\ l)$; $(p,ins,outs,c) \in$ *set procs*;
*well-formed procs*$]\!] \Longrightarrow l < \#:c$
⟨*proof*⟩

**lemma** *Proc-CFG-edge-det*:
$[\![prog,procs \vdash n\ -et\rightarrow\ n'$; $prog,procs \vdash n\ -et'\rightarrow\ n'$; *well-formed procs*$]\!]$
$\Longrightarrow et = et'$
⟨*proof*⟩

**lemma** *Proc-CFG-deterministic*:
$[\![prog,procs \vdash n_1\ -et_1\rightarrow\ n_1'$; $prog,procs \vdash n_2\ -et_2\rightarrow\ n_2'$; $n_1 = n_2$; $n_1' \neq n_2'$;
*intra-kind* $et_1$; *intra-kind* $et_2$; *well-formed procs*$]\!]$
$\Longrightarrow \exists\ Q\ Q'.\ et_1 = (Q)_{\surd} \wedge et_2 = (Q')_{\surd} \wedge$
$(\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$
⟨*proof*⟩

### 2.4.2 Well-formedness of programs in combination with a procedure list.

**definition** *wf* :: *cmd* $\Rightarrow$ *procs* $\Rightarrow$ *bool*
**where** *wf prog procs* $\equiv$ *well-formed procs* $\wedge$

(∀ *ps p. containsCall procs prog ps p* ⟶ (∃ *ins outs c.* (*p*,*ins*,*outs*,*c*) ∈ *set procs*
∧
(∀ *c′ n n′ es rets. c′* ⊢ *n −CEdge* (*p*,*es*,*rets*)$\rightarrow_p$ *n′* ⟶
*distinct rets* ∧ *length rets* = *length outs* ∧ *length es* = *length ins*)))

**lemma** *wf-well-formed* [*intro*]:*wf prog procs* ⟹ *well-formed procs*
⟨*proof*⟩

**lemma** *wf-distinct-rets* [*intro*]:
⟦*wf prog procs*; *containsCall procs prog ps p*; (*p*,*ins*,*outs*,*c*) ∈ *set procs*;
*c′* ⊢ *n −CEdge* (*p*,*es*,*rets*)$\rightarrow_p$ *n′*⟧ ⟹ *distinct rets*
⟨*proof*⟩

**lemma**
**assumes** *wf prog procs* **and** *containsCall procs prog ps p*
**and** (*p*,*ins*,*outs*,*c*) ∈ *set procs* **and** *c′* ⊢ *n −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*
⟨*proof*⟩

### 2.4.3 Type of well-formed programs

**definition** *wf-prog* = {(*prog*,*procs*). *wf prog procs*}

**typedef** *wf-prog* = *wf-prog*
⟨*proof*⟩

**lemma** *wf-wf-prog*:
**fixes** *wfp*
**shows** *Rep-wf-prog wfp* = (*prog*,*procs*) ⟹ *wf prog procs*
⟨*proof*⟩

**lemma** *wfp-Seq1*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = ($c_1$;; $c_2$, *procs*)
**obtains** *wfp′* **where** *Rep-wf-prog wfp′* = ($c_1$, *procs*)
⟨*proof*⟩

**lemma** *wfp-Seq2*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = ($c_1$;; $c_2$, *procs*)
**obtains** *wfp′* **where** *Rep-wf-prog wfp′* = ($c_2$, *procs*)
⟨*proof*⟩

**lemma** *wfp-CondTrue*:

**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*if* (*b*) $c_1$ *else* $c_2$, *procs*)
**obtains** *wfp′* **where** *Rep-wf-prog wfp′* = ($c_1$, *procs*)
⟨*proof*⟩

**lemma** *wfp-CondFalse*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*if* (*b*) $c_1$ *else* $c_2$, *procs*)
**obtains** *wfp′* **where** *Rep-wf-prog wfp′* = ($c_2$, *procs*)
⟨*proof*⟩

**lemma** *wfp-WhileBody*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*while* (*b*) *c′*, *procs*)
**obtains** *wfp′* **where** *Rep-wf-prog wfp′* = (*c′*, *procs*)
⟨*proof*⟩

**lemma** *wfp-Call*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*prog*,*procs*)
**and** (*p*,*ins*,*outs*,*c*) ∈ *set procs* **and** *containsCall procs prog ps p*
**obtains** *wfp′* **where** *Rep-wf-prog wfp′* = (*c*,*procs*)
⟨*proof*⟩

**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* ⇒ *node*
**where** *sourcenode e* ≡ *fst e*

**abbreviation** *targetnode* :: *edge* ⇒ *node*
**where** *targetnode e* ≡ *snd*(*snd e*)

**abbreviation** *kind* :: *edge* ⇒ (*vname*,*val*,*node*,*pname*) *edge-kind*
**where** *kind e* ≡ *fst*(*snd e*)

**definition** *valid-edge* :: *wf-prog* ⇒ *edge* ⇒ *bool*
**where** ⋀*wfp*. *valid-edge wfp a* ≡ *let* (*prog*,*procs*) = *Rep-wf-prog wfp in*
*prog*,*procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*

**definition** *get-return-edges* :: *wf-prog* ⇒ *edge* ⇒ *edge set*
**where** $\bigwedge$*wfp*. *get-return-edges wfp a* ≡
*case kind a of* $Q{:}r{\hookrightarrow}_{p}fs$ ⇒ {*a*′. *valid-edge wfp a*′ ∧ (∃ *Q*′ *f*′. *kind a*′ = $Q'{\hookleftarrow}_{p}f'$) ∧
*targetnode a*′ = *r*}
| - ⇒ {}

**lemma** *get-return-edges-non-call-empty*:
**fixes** *wfp*
**shows** ∀ *Q r p fs*. *kind a* ≠ $Q{:}r{\hookrightarrow}_{p}fs$ ⟹ *get-return-edges wfp a* = {}
⟨*proof*⟩

**lemma** *call-has-return-edge*:
**fixes** *wfp*
**assumes** *valid-edge wfp a* **and** *kind a* = $Q{:}r{\hookrightarrow}_{p}fs$
**obtains** *a*′ **where** *valid-edge wfp a*′ **and** ∃ *Q*′ *f*′. *kind a*′ = $Q'{\hookleftarrow}_{p}f'$
**and** *targetnode a*′ = *r*
⟨*proof*⟩

**lemma** *get-return-edges-call-nonempty*:
**fixes** *wfp*
**shows** ⟦*valid-edge wfp a*; *kind a* = $Q{:}r{\hookrightarrow}_{p}fs$⟧ ⟹ *get-return-edges wfp a* ≠ {}
⟨*proof*⟩

**lemma** *only-return-edges-in-get-return-edges*:
**fixes** *wfp*
**shows** ⟦*valid-edge wfp a*; *kind a* = $Q{:}r{\hookrightarrow}_{p}fs$; *a*′ ∈ *get-return-edges wfp a*⟧
⟹ ∃ *Q*′ *f*′. *kind a*′ = $Q'{\hookleftarrow}_{p}f'$
⟨*proof*⟩

**abbreviation** *lift-procs* :: *wf-prog* ⇒ (*pname* × *vname list* × *vname list*) *list*
**where** $\bigwedge$*wfp*. *lift-procs wfp* ≡ *let* (*prog*,*procs*) = *Rep-wf-prog wfp in*
*map* (λ*x*. (*fst x*,*fst*(*snd x*),*fst*(*snd*(*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* ⇒ *nat* ⇒ *cmd* ⇒ *bool*
**where**

*Labels-Base*:
*labels c 0 c*

| *Labels-LAss*:
*labels* (*V*:=*e*) *1 Skip*

| *Labels-Seq1*:
*labels* $c_1$ *l c* ⟹ *labels* ($c_1$;;$c_2$) *l* (*c*;;$c_2$)

| *Labels-Seq2*:
*labels* $c_2$ *l c* ⟹ *labels* ($c_1$;;$c_2$) (*l* + #:$c_1$) *c*

| *Labels-CondTrue*:
*labels* $c_1$ *l c* ⟹ *labels* (*if* (*b*) $c_1$ *else* $c_2$) (*l* + *1*) *c*

| *Labels-CondFalse*:
*labels* $c_2$ *l c* ⟹ *labels* (*if* (*b*) $c_1$ *else* $c_2$) (*l* + #:$c_1$ + *1*) *c*

| *Labels-WhileBody*:
*labels* $c'$ *l c* ⟹ *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* $c'$ ⟹ *l* < #:*c*
⟨*proof*⟩

**declare** *One-nat-def* [*simp del*]

**lemma** *less-num-inner-nodes-label*:
**assumes** $l < \#:c$ **obtains** $c'$ **where** *labels c l c'*
⟨*proof*⟩

**lemma** *labels-det*:
*labels c l c'*$\Longrightarrow$ ($\bigwedge c''$. *labels c l c''*$\Longrightarrow$ $c' = c''$)
⟨*proof*⟩

**definition** *label* :: *cmd* ⇒ *nat* ⇒ *cmd*
**where** *label c n* ≡ (*THE c'*. *labels c n c'*)

**lemma** *labels-THE*:
*labels c l c'* $\Longrightarrow$ (*THE c'*. *labels c l c'*) = $c'$
⟨*proof*⟩

**lemma** *labels-label*:*labels c l c'* $\Longrightarrow$ *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* ⇒ *cmd*
**where** *fst-cmd* ($c_1$;;$c_2$) = *fst-cmd* $c_1$
| *fst-cmd c* = *c*

**lemma** *Proc-CFG-Call-target-fst-cmd-Skip*:
⟦*labels prog l' c*; *prog* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)$\rightarrow_p$ *Label l'*⟧
$\Longrightarrow$ *fst-cmd c* = *Skip*
⟨*proof*⟩

**lemma** *Proc-CFG-Call-source-fst-cmd-Call*:

$[\![$*labels prog l c*; *prog* ⊢ *Label l* −*CEdge* (*p*,*es*,*rets*)→$_p$ *n*$]\!]$
⟹ ∃ *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* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *n'*
⟹ (*THE rets'*. ∃ *p' es' n*. *prog* ⊢ *n* −*CEdge*(*p'*,*es'*,*rets'*)→$_p$ *n'*) = *rets*
⟨*proof*⟩

**definition** *ParamDefs-proc* :: *cmd* ⇒ *label* ⇒ *vname list*
**where** *ParamDefs-proc c n* ≡
*if* (∃ *n' p' es' rets'*. *c* ⊢ *n'* −*CEdge*(*p'*,*es'*,*rets'*)→$_p$ *n*) *then*
(*THE rets'*. ∃ *p' es' n'*. *c* ⊢ *n'* −*CEdge*(*p'*,*es'*,*rets'*)→$_p$ *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** ⋀*wfp*. *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*:
**fixes** *wfp*
**shows** $[\![$*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*:
**fixes** *wfp*
**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*:

**fixes** *wfp*
**shows** ⟦*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* ⊢ $n$ −*IEdge et*→$_p$ $n'$⟧
⟹ *ParamDefs wfp* (*Main*,$n'$) = []
⟨*proof*⟩

**lemma** *ParamDefs-Proc-IEdge-Nil*:
**fixes** *wfp*
**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*:
**fixes** *wfp*
**shows** ⟦*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* ⊢ $n'$ −*CEdge*($p'$,*es*,*rets*)→$_p$ $n''$⟧
⟹ *ParamDefs wfp* (*Main*,$n'$) = []
⟨*proof*⟩

**lemma** *ParamDefs-Proc-CEdge-Nil*:
**fixes** *wfp*
**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**
**fixes** *wfp*
**assumes** *valid-edge wfp a* **and** *kind a* = $Q'$↩$_p$$f'$
**and** (*p*, *ins*, *outs*) ∈ *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*)
⟨*proof*⟩

*ParamUses*

**fun** *fv* :: *expr* ⇒ *vname set*
**where**
*fv* (*Val v*) = {}
| *fv* (*Var V*) = {*V*}
| *fv* (*e1* «*bop*» *e2*) = (*fv e1* ∪ *fv e2*)

**lemma** *rhs-interpret-eq*:
⟦*state-check cf e* $v'$; ∀ *V* ∈ *fv e*. *cf V* = $cf'$ *V*⟧
⟹ *state-check* $cf'$ *e* $v'$
⟨*proof*⟩

**lemma** *PCFG-CallEdge-THE-es*:
$prog \vdash n -CEdge(p,es,rets)\rightarrow_p n'$
$\Longrightarrow (THE\ es'.\ \exists p'\ rets'\ n'.\ prog \vdash n -CEdge(p',es',rets')\rightarrow_p n') = es$
⟨*proof*⟩

**definition** *ParamUses-proc* :: *cmd* ⇒ *label* ⇒ *vname set list*
**where** *ParamUses-proc c n* ≡
*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* []

**definition** *ParamUses* :: *wf-prog* ⇒ *node* ⇒ *vname set list*
**where** ⋀*wfp*. *ParamUses wfp n* ≡ *let* (*prog*,*procs*) = *Rep-wf-prog wfp*; (*p*,*l*) = *n*
*in*
(*if* (*p* = *Main*) *then ParamUses-proc prog l*
*else* (*if* (∃ *ins outs c*. (*p*,*ins*,*outs*,*c*) ∈ *set procs*)
*then ParamUses-proc* (*THE c'*. ∃ *ins' outs'*. (*p*,*ins'*,*outs'*,*c'*) ∈ *set procs*) *l*
*else* []))

**lemma** *ParamUses-Main-Return-target*:
**fixes** *wfp*
**shows** $[\![Rep\text{-}wf\text{-}prog\ wfp = (prog,procs);\ prog \vdash n -CEdge(p',es,rets)\rightarrow_p n']\!]$
$\Longrightarrow ParamUses\ wfp\ (Main,n) = map\ fv\ es$
⟨*proof*⟩

**lemma** *ParamUses-Proc-Return-target*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*prog*,*procs*)
**and** (*p*,*ins*,*outs*,*c*) ∈ *set procs* **and** $c \vdash n -CEdge(p',es,rets)\rightarrow_p n'$
**shows** *ParamUses wfp* (*p*,*n*) = *map fv es*
⟨*proof*⟩

**lemma** *ParamUses-Main-IEdge-Nil*:
**fixes** *wfp*
**shows** $[\![Rep\text{-}wf\text{-}prog\ wfp = (prog,procs);\ prog \vdash n -IEdge\ et\rightarrow_p n']\!]$
$\Longrightarrow ParamUses\ wfp\ (Main,n) = []$
⟨*proof*⟩

**lemma** *ParamUses-Proc-IEdge-Nil*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*prog*,*procs*)
**and** (*p*,*ins*,*outs*,*c*) ∈ *set procs* **and** $c \vdash n -IEdge\ et\rightarrow_p n'$
**shows** *ParamUses wfp* (*p*,*n*) = []

⟨*proof*⟩

**lemma** *ParamUses-Main-CEdge-Nil*:
**fixes** *wfp*
**shows** $[\![$*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* ⊢ $n'$ $-CEdge(p',es,rets)\rightarrow_p$ $n]\!]$
⟹ *ParamUses wfp* (*Main*,*n*) = []
⟨*proof*⟩

**lemma** *ParamUses-Proc-CEdge-Nil*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*prog*,*procs*)
**and** (*p*,*ins*,*outs*,*c*) ∈ *set procs* **and** *c* ⊢ $n'$ $-CEdge(p',es,rets)\rightarrow_p$ *n*
**shows** *ParamUses wfp* (*p*,*n*) = []
⟨*proof*⟩

## *Def*

**fun** *lhs* :: *cmd* ⇒ *vname set*
**where**
*lhs Skip* = {}
| *lhs* (*V*:=*e*) = {*V*}
| *lhs* ($c_1$;;$c_2$) = *lhs* $c_1$
| *lhs* (*if* (*b*) $c_1$ *else* $c_2$) = {}
| *lhs* (*while* (*b*) *c*) = {}
| *lhs* (*Call p es rets*) = {}

**lemma** *lhs-fst-cmd*:*lhs* (*fst-cmd c*) = *lhs c* ⟨*proof*⟩

**lemma** *Proc-CFG-Call-source-empty-lhs*:
**assumes** *prog* ⊢ *Label l* $-CEdge$ $(p,es,rets)\rightarrow_p$ $n'$
**shows** *lhs* (*label prog l*) = {}
⟨*proof*⟩

**lemma** *in-procs-THE-in-procs-ins*:
$[\![$*well-formed procs*; (*p*,*ins*,*outs*,*c*) ∈ *set procs*$]\!]$
⟹ (*THE* $ins'$. ∃ $c'$ $outs'$. ($p$,$ins'$,$outs'$,$c'$) ∈ *set procs*) = *ins*
⟨*proof*⟩

**definition** *Def* :: *wf-prog* ⇒ *node* ⇒ *vname set*
**where** ⋀*wfp*. *Def wfp n* ≡ (*let* (*prog*,*procs*) = *Rep-wf-prog wfp*; (*p*,*l*) = *n in*
(*case l of Label lx* ⇒
(*if p* = *Main then lhs* (*label prog lx*)
*else* (*if* (∃ *ins outs c*. (*p*,*ins*,*outs*,*c*) ∈ *set procs*)
*then*
*lhs* (*label* (*THE* $c'$. ∃ $ins'$ $outs'$. ($p$,$ins'$,$outs'$,$c'$) ∈ *set procs*) *lx*)
*else* {}))
| *Entry* ⇒ *if* (∃ *ins outs c*. (*p*,*ins*,*outs*,*c*) ∈ *set procs*)

*then* (*set*
(*THE ins′*. ∃ *c′ outs′*. (*p*,*ins′*,*outs′*,*c′*) ∈ *set procs*)) *else* {}
| *Exit* ⇒ {}))
∪ *set* (*ParamDefs wfp n*)

**lemma** *Entry-Def-empty*:
**fixes** *wfp*
**shows** *Def wfp* (*Main*, *Entry*) = {}
⟨*proof*⟩

**lemma** *Exit-Def-empty*:
**fixes** *wfp*
**shows** *Def wfp* (*Main*, *Exit*) = {}
⟨*proof*⟩

*Use*

**fun** *rhs* :: *cmd* ⇒ *vname set*
**where**
*rhs Skip* = {}
| *rhs* (*V*:=*e*) = *fv e*
| *rhs* ($c_1$;;$c_2$) = *rhs* $c_1$
| *rhs* (*if* (*b*) $c_1$ *else* $c_2$) = *fv b*
| *rhs* (*while* (*b*) *c*) = *fv b*
| *rhs* (*Call p es rets*) = {}

**lemma** *rhs-fst-cmd*:*rhs* (*fst-cmd c*) = *rhs c* ⟨*proof*⟩

**lemma** *Proc-CFG-Call-target-empty-rhs*:
**assumes** *prog* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)$\rightarrow_p$ *Label l′*
**shows** *rhs* (*label prog l′*) = {}
⟨*proof*⟩

**lemma** *in-procs-THE-in-procs-outs*:
⟦*well-formed procs*; (*p*,*ins*,*outs*,*c*) ∈ *set procs*⟧
⟹ (*THE outs′*. ∃ *c′ ins′*. (*p*,*ins′*,*outs′*,*c′*) ∈ *set procs*) = *outs*
⟨*proof*⟩

**definition** *Use* :: *wf-prog* ⇒ *node* ⇒ *vname set*
**where** ⋀*wfp*. *Use wfp n* ≡ (*let* (*prog*,*procs*) = *Rep-wf-prog wfp*; (*p*,*l*) = *n in*
(*case l of Label lx* ⇒
(*if p* = *Main then rhs* (*label prog lx*)
*else* (*if* (∃ *ins outs c*. (*p*,*ins*,*outs*,*c*) ∈ *set procs*)
*then*
*rhs* (*label* (*THE c′*. ∃ *ins′ outs′*. (*p*,*ins′*,*outs′*,*c′*) ∈ *set procs*) *lx*)

*else* {}))
| *Exit* ⇒ *if* (∃ *ins outs c*. (*p*,*ins*,*outs*,*c*) ∈ *set procs*)
*then* (*set* (*THE outs*′. ∃ *c*′ *ins*′. (*p*,*ins*′,*outs*′,*c*′) ∈ *set procs*) )
*else* {}
| *Entry* ⇒ *if* (∃ *ins outs c*. (*p*,*ins*,*outs*,*c*) ∈ *set procs*)
*then* (*set* (*THE ins*′. ∃ *c*′ *outs*′. (*p*,*ins*′,*outs*′,*c*′) ∈ *set procs*))
*else* {}))
∪ *Union* (*set* (*ParamUses wfp n*)) ∪ *set* (*ParamDefs wfp n*)

**lemma** *Entry-Use-empty*:
**fixes** *wfp*
**shows** *Use wfp* (*Main*, *Entry*) = {}
⟨*proof*⟩

**lemma** *Exit-Use-empty*:
**fixes** *wfp*
**shows** *Use wfp* (*Main*, *Exit*) = {}
⟨*proof*⟩

### 2.7.3 Lemmas about edges and call frames

**lemmas** *transfers-simps* = *ProcCFG.transfer.simps*[*simplified*]
**declare** *transfers-simps* [*simp*]

**abbreviation** *state-val* :: ((′*var* ⇀ ′*val*) × ′*ret*) *list* ⇒ ′*var* ⇀ ′*val*
**where** *state-val s V* ≡ (*fst* (*hd s*)) *V*

**lemma** *Proc-CFG-edge-no-lhs-equal*:
**fixes** *wfp*
**assumes** *prog* ⊢ *Label l* −*IEdge et*→$_p$ *n*′ **and** *V* ∉ *lhs* (*label prog l*)
**shows** *state-val* (*CFG.transfer* (*lift-procs wfp*) *et* (*cf*#*cfs*)) *V* = *fst cf V*
⟨*proof*⟩

**lemma** *Proc-CFG-edge-uses-only-rhs*:
**fixes** *wfp*
**assumes** *prog* ⊢ *Label l* −*IEdge et*→$_p$ *n*′ **and** *CFG.pred et s*
**and** *CFG.pred et s*′ **and** ∀ *V*∈*rhs* (*label prog l*). *state-val s V* = *state-val s*′ *V*
**shows** ∀ *V*∈*lhs* (*label prog l*).
*state-val* (*CFG.transfer* (*lift-procs wfp*) *et s*) *V* =
*state-val* (*CFG.transfer* (*lift-procs wfp*) *et s*′) *V*
⟨*proof*⟩

**lemma** *Proc-CFG-edge-rhs-pred-eq*:
**assumes** *prog* ⊢ *Label l* −*IEdge et*→$_p$ *n*′ **and** *CFG.pred et s*
**and** ∀ *V*∈*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* ⇒ *node* ⇒ *edge list* ⇒ *node* ⇒ *bool* (‹- ⊢ - −-→∗ -›)
**where** ⋀*wfp*. *wfp* ⊢ *n* −*as*→∗ *n'* ≡ *CFG.path sourcenode targetnode* (*valid-edge wfp*) *n as n'*

**definition** *label-incrs* :: *edge list* ⇒ *nat* ⇒ *edge list* (‹- ⊕s -› *60*)
**where** *as* ⊕*s i* ≡ *map* (λ((*p*,*n*),*et*,(*p'*,*n'*)). ((*p*,*n* ⊕ *i*),*et*,(*p'*,*n'* ⊕ *i*))) *as*

**declare** *One-nat-def* [*simp del*]

**From** *prog* **to** *prog*;;$c_2$

**lemma** *Proc-CFG-edge-SeqFirst-nodes-Label*:
*prog* ⊢ *Label l* −*et*→$_p$ *Label l'* ⟹ *prog*;;$c_2$ ⊢ *Label l* −*et*→$_p$ *Label l'*
⟨*proof*⟩

**lemma** *Proc-CFG-edge-SeqFirst-source-Label*:
**assumes** $prog \vdash Label\ l\ -et\rightarrow_p\ n'$
**obtains** $nx$ **where** $prog;;c_2 \vdash Label\ l\ -et\rightarrow_p\ nx$
⟨*proof*⟩

**lemma** *Proc-CFG-edge-SeqFirst-target-Label*:
$[\![prog \vdash n\ -et\rightarrow_p\ n';\ Label\ l' = n']\!] \Longrightarrow prog;;c_2 \vdash n\ -et\rightarrow_p\ Label\ l'$
⟨*proof*⟩

**lemma** *PCFG-edge-SeqFirst-source-Label*:
**assumes** $prog,procs \vdash (p,Label\ l)\ -et\rightarrow\ (p',n')$
**obtains** $nx$ **where** $prog;;c_2,procs \vdash (p,Label\ l)\ -et\rightarrow\ (p',nx)$
⟨*proof*⟩

**lemma** *PCFG-edge-SeqFirst-target-Label*:
$prog,procs \vdash (p,n)\ -et\rightarrow\ (p',Label\ l')$
$\Longrightarrow prog;;c_2,procs \vdash (p,n)\ -et\rightarrow\ (p',Label\ l')$
⟨*proof*⟩

**lemma** *path-SeqFirst*:
**fixes** $wfp$
**assumes** $Rep\text{-}wf\text{-}prog\ wfp = (prog,procs)$ **and** $Rep\text{-}wf\text{-}prog\ wfp' = (prog;;c_2,procs)$
**shows** $[\![wfp \vdash (p,n)\ -as\rightarrow*\ (p,Label\ l);\ \forall\ a \in set\ as.\ intra\text{-}kind\ (kind\ a)]\!]$
$\Longrightarrow wfp' \vdash (p,n)\ -as\rightarrow*\ (p,Label\ l)$
⟨*proof*⟩

### From *prog* to $c_1;;prog$

**lemma** *Proc-CFG-edge-SeqSecond-source-not-Entry*:
$[\![prog \vdash n\ -et\rightarrow_p\ n';\ n \neq Entry]\!] \Longrightarrow c_1;;prog \vdash n \oplus \#:c_1\ -et\rightarrow_p\ n' \oplus \#:c_1$
⟨*proof*⟩

**lemma** *PCFG-Main-edge-SeqSecond-source-not-Entry*:
$[\![prog,procs \vdash (Main,n)\ -et\rightarrow\ (p',n');\ n \neq Entry;\ intra\text{-}kind\ et;\ well\text{-}formed\ procs]\!]$
$\Longrightarrow c_1;;prog,procs \vdash (Main,n \oplus \#:c_1)\ -et\rightarrow\ (p',n' \oplus \#:c_1)$
⟨*proof*⟩

**lemma** *valid-node-Main-SeqSecond*:
**fixes** $wfp$
**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-wf-prog wfp'* = ($c_1$;;*prog*,*procs*)
**shows** *CFG.valid-node sourcenode targetnode* (*valid-edge wfp'*) (*Main*, $n \oplus \#{:}c_1$)
⟨*proof*⟩

**lemma** *path-Main-SeqSecond*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*prog*,*procs*) **and** *Rep-wf-prog wfp'* = ($c_1$;;*prog*,*procs*)
**shows** ⟦*wfp* ⊢ (*Main*,*n*) −*as*→∗ (*p'*,*n'*); ∀ *a* ∈ *set as. intra-kind* (*kind a*); *n* ≠ *Entry*⟧
⟹ *wfp'* ⊢ (*Main*,$n \oplus \#{:}c_1$) −*as* ⊕*s* $\#{:}c_1$→∗ (*p'*,$n' \oplus \#{:}c_1$)
⟨*proof*⟩

**From** *prog* **to** *if* (*b*) *prog else* $c_2$

**lemma** *Proc-CFG-edge-CondTrue-source-not-Entry*:
⟦*prog* ⊢ *n* −*et*→$_p$ *n'*; *n* ≠ *Entry*⟧ ⟹ *if* (*b*) *prog else* $c_2$ ⊢ *n* ⊕ *1* −*et*→$_p$ *n'* ⊕ *1*
⟨*proof*⟩

**lemma** *PCFG-Main-edge-CondTrue-source-not-Entry*:
⟦*prog*,*procs* ⊢ (*Main*,*n*) −*et*→ (*p'*,*n'*); *n* ≠ *Entry*; *intra-kind et*; *well-formed procs*⟧
⟹ *if* (*b*) *prog else* $c_2$,*procs* ⊢ (*Main*,*n* ⊕ *1*) −*et*→ (*p'*,*n'* ⊕ *1*)
⟨*proof*⟩

**lemma** *valid-node-Main-CondTrue*:
**fixes** *wfp*
**assumes** *CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*Main*,*n*)
**and** *n* ≠ *Entry* **and** *Rep-wf-prog wfp* = (*prog*,*procs*)
**and** *Rep-wf-prog wfp'* = (*if* (*b*) *prog else* $c_2$,*procs*)
**shows** *CFG.valid-node sourcenode targetnode* (*valid-edge wfp'*) (*Main*, *n* ⊕ *1*)
⟨*proof*⟩

**lemma** *path-Main-CondTrue*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*prog*,*procs*)
**and** *Rep-wf-prog wfp'* = (*if* (*b*) *prog else* $c_2$,*procs*)
**shows** ⟦*wfp* ⊢ (*Main*,*n*) −*as*→∗ (*p'*,*n'*); ∀ *a* ∈ *set as. intra-kind* (*kind a*); *n* ≠ *Entry*⟧
⟹ *wfp'* ⊢ (*Main*,*n* ⊕ *1*) −*as* ⊕*s* *1*→∗ (*p'*,*n'* ⊕ *1*)
⟨*proof*⟩

**From** *prog* **to** *if* (*b*) $c_1$ *else prog*

**lemma** *Proc-CFG-edge-CondFalse-source-not-Entry*:
⟦*prog* ⊢ *n* −*et*→$_p$ *n'*; *n* ≠ *Entry*⟧
⟹ *if* (*b*) $c_1$ *else prog* ⊢ $n \oplus (\#{:}c_1 + 1)$ −*et*→$_p$ $n' \oplus (\#{:}c_1 + 1)$
⟨*proof*⟩

**lemma** *PCFG-Main-edge-CondFalse-source-not-Entry*:
⟦*prog*,*procs* ⊢ (*Main*,*n*) −*et*→ (*p'*,*n'*); *n* ≠ *Entry*; *intra-kind et*; *well-formed procs*⟧
⟹ *if* (*b*) $c_1$ *else prog*,*procs* ⊢ (*Main*,*n* ⊕ (#:$c_1$ + *1*)) −*et*→ (*p'*,*n'* ⊕ (#:$c_1$ + *1*))
⟨*proof*⟩

**lemma** *valid-node-Main-CondFalse*:
**fixes** *wfp*
**assumes** *CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*Main*,*n*)
**and** *n* ≠ *Entry* **and** *Rep-wf-prog wfp* = (*prog*,*procs*)
**and** *Rep-wf-prog wfp'* = (*if* (*b*) $c_1$ *else prog*,*procs*)
**shows** *CFG.valid-node sourcenode targetnode* (*valid-edge wfp'*)
(*Main*, *n* ⊕ (#:$c_1$ + *1*))
⟨*proof*⟩

**lemma** *path-Main-CondFalse*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*prog*,*procs*)
**and** *Rep-wf-prog wfp'* = (*if* (*b*) $c_1$ *else prog*,*procs*)
**shows** ⟦*wfp* ⊢ (*Main*,*n*) −*as*→∗ (*p'*,*n'*); ∀ *a* ∈ *set as*. *intra-kind* (*kind a*); *n* ≠ *Entry*⟧
⟹ *wfp'* ⊢ (*Main*,*n* ⊕ (#:$c_1$ + *1*)) −*as* ⊕*s* (#:$c_1$ + *1*)→∗ (*p'*,*n'* ⊕ (#:$c_1$ + *1*))
⟨*proof*⟩

### From *prog* to *while* (*b*) *prog*

**lemma** *Proc-CFG-edge-WhileBody-source-not-Entry*:
⟦*prog* ⊢ *n* −*et*→$_p$ *n'*; *n* ≠ *Entry*; *n'* ≠ *Exit*⟧
⟹ *while* (*b*) *prog* ⊢ *n* ⊕ *2* −*et*→$_p$ *n'* ⊕ *2*
⟨*proof*⟩

**lemma** *PCFG-Main-edge-WhileBody-source-not-Entry*:
⟦*prog*,*procs* ⊢ (*Main*,*n*) −*et*→ (*p'*,*n'*); *n* ≠ *Entry*; *n'* ≠ *Exit*; *intra-kind et*;
*well-formed procs*⟧ ⟹ *while* (*b*) *prog*,*procs* ⊢ (*Main*,*n* ⊕ *2*) −*et*→ (*p'*,*n'* ⊕ *2*)
⟨*proof*⟩

**lemma** *valid-node-Main-WhileBody*:
**fixes** *wfp*
**assumes** *CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*Main*,*n*)
**and** *n* ≠ *Entry* **and** *Rep-wf-prog wfp* = (*prog*,*procs*)
**and** *Rep-wf-prog wfp'* = (*while* (*b*) *prog*,*procs*)
**shows** *CFG.valid-node sourcenode targetnode* (*valid-edge wfp'*) (*Main*, *n* ⊕ *2*)
⟨*proof*⟩

**lemma** *path-Main-WhileBody*:
  **fixes** *wfp*
  **assumes** *Rep-wf-prog wfp* = (*prog*,*procs*)
  **and** *Rep-wf-prog wfp′* = (*while* (*b*) *prog*,*procs*)
  **shows** ⟦*wfp* ⊢ (*Main*,*n*) −*as*→∗ (*p′*,*n′*); ∀ *a* ∈ *set as*. *intra-kind* (*kind a*);
    *n* ≠ *Entry*; *n′* ≠ *Exit*⟧ ⟹ *wfp′* ⊢ (*Main*,*n* ⊕ *2*) −*as* ⊕*s 2*→∗ (*p′*,*n′* ⊕ *2*)
⟨*proof*⟩

### Existence of intraprodecural paths

**lemma** *Label-Proc-CFG-Entry-Exit-path-Main*:
  **fixes** *wfp*
  **assumes** *Rep-wf-prog wfp* = (*prog*,*procs*) **and** *l* < #:*prog*
  **obtains** *as as′* **where** *wfp* ⊢ (*Main*,*Label l*) −*as*→∗ (*Main*,*Exit*)
  **and** ∀ *a* ∈ *set as*. *intra-kind* (*kind a*)
  **and** *wfp* ⊢ (*Main*,*Entry*) −*as′*→∗ (*Main*,*Label l*)
  **and** ∀ *a* ∈ *set as′*. *intra-kind* (*kind a*)
⟨*proof*⟩

## 2.8.2 Lifting from edges in procedure Main to arbitrary procedures

**lemma** *lift-edge-Main-Main*:
  ⟦*c*,*procs* ⊢ (*Main*, *n*) −*et*→ (*Main*, *n′*); (*p*,*ins*,*outs*,*c*) ∈ *set procs*;
  *containsCall procs prog ps p*; *well-formed procs*⟧
  ⟹ *prog*,*procs* ⊢ (*p*, *n*) −*et*→ (*p*, *n′*)
⟨*proof*⟩

**lemma** *lift-edge-Main-Proc*:
  ⟦*c*,*procs* ⊢ (*Main*, *n*) −*et*→ (*q*, *n′*); *q* ≠ *Main*; (*p*,*ins*,*outs*,*c*) ∈ *set procs*;
  *containsCall procs prog ps p*; *well-formed procs*⟧
  ⟹ ∃ *et′*. *prog*,*procs* ⊢ (*p*, *n*) −*et′*→ (*q*, *n′*)
⟨*proof*⟩

**lemma** *lift-edge-Proc-Main*:
  ⟦*c*,*procs* ⊢ (*q*, *n*) −*et*→ (*Main*, *n′*); *q* ≠ *Main*; (*p*,*ins*,*outs*,*c*) ∈ *set procs*;
  *containsCall procs prog ps p*; *well-formed procs*⟧
  ⟹ ∃ *et′*. *prog*,*procs* ⊢ (*q*, *n*) −*et′*→ (*p*, *n′*)
⟨*proof*⟩

**fun** *lift-edge* :: *edge* ⇒ *pname* ⇒ *edge*
**where** *lift-edge a p* = ((*p*,*snd*(*sourcenode a*)),*kind a*,(*p*,*snd*(*targetnode a*)))

**fun** *lift-path* :: *edge list* ⇒ *pname* ⇒ *edge list*
  **where** *lift-path as p* = *map* (λ*a*. *lift-edge a p*) *as*

**lemma** *lift-path-Proc*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp′* = (*c*,*procs*) **and** *Rep-wf-prog wfp* = (*prog*,*procs*)
**and** (*p*,*ins*,*outs*,*c*) ∈ *set procs* **and** *containsCall procs prog ps p*
**shows** ⟦*wfp′* ⊢ (*Main*,*n*) −*as*→∗ (*Main*,*n′*); ∀ *a* ∈ *set as*. *intra-kind* (*kind a*)⟧
⟹ *wfp* ⊢ (*p*,*n*) −*lift-path as p*→∗ (*p*,*n′*)
⟨*proof*⟩

### 2.8.3 Existence of paths from Entry and to Exit

**lemma** *Label-Proc-CFG-Entry-Exit-path-Proc*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*prog*,*procs*) **and** *l* < #:*c*
**and** (*p*,*ins*,*outs*,*c*) ∈ *set procs* **and** *containsCall procs prog ps p*
**obtains** *as as′* **where** *wfp* ⊢ (*p*,*Label l*) −*as*→∗ (*p*,*Exit*)
**and** ∀ *a* ∈ *set as*. *intra-kind* (*kind a*)
**and** *wfp* ⊢ (*p*,*Entry*) −*as′*→∗ (*p*,*Label l*)
**and** ∀ *a* ∈ *set as′*. *intra-kind* (*kind a*)
⟨*proof*⟩

**lemma** *Entry-to-Entry-and-Exit-to-Exit*:
**fixes** *wfp*
**assumes** *Rep-wf-prog wfp* = (*prog*,*procs*)
**and** *containsCall procs prog ps p* **and** (*p*,*ins*,*outs*,*c*) ∈ *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*:
**fixes** *wfp*
**assumes** *prog*,*procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*
**and** *disj*:(*p*,*n*) = *sourcenode a* ∨ (*p*,*n*) = *targetnode a*
**and** [*simp*]:*Rep-wf-prog wfp* = (*prog*,*procs*)
**shows** ∃ *as as′*. *CFG.valid-path′ sourcenode targetnode kind* (*valid-edge wfp*)
(*get-return-edges wfp*) (*Main*,*Entry*) *as* (*p*,*n*) ∧
*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*). $\textit{wf-jvm-prog}_{\textit{Phi}}$ *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* ⇒ $ty_P$
  **where** *TYPING P* ≡ *snd*(*Rep-wf-jvmprog*(*P*))

**lemma** *wf-jvmprog-is-wf-typ*: $\textit{wf-jvm-prog}_{\textit{TYPING P}}$ (*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* ⇒ *pc* ⇒ *nat*
**where**
*locLength m pc* ≡ *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*: ∃ *C*. ¬ *is-class P C*
⟨*proof*⟩

**lemma** *ClassMain-unique-in-P*:
**assumes** *is-class* (*PROG P*) *C*
**shows** *ClassMain P* ≠ *C*
⟨*proof*⟩

**lemma** *map-of-fstD*: ⟦ *map-of xs a* = ⌊*b*⌋; ∀ *x* ∈ *set xs*. *fst x* ≠ *a* ⟧ ⟹ *False*
⟨*proof*⟩

**lemma** *map-of-fstE*: ⟦ *map-of xs a* = ⌊*b*⌋; ∃ *x* ∈ *set xs*. *fst x* = *a* ⟹ *thesis* ⟧ ⟹ *thesis*
⟨*proof*⟩

**lemma** *ex-unique-method-name*:
∃ *Name*. ∀ *C D fs ms*. *class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋ ⟶ (∀ *m*∈*set ms*. *Name* ≠ *fst m*)
⟨*proof*⟩

**lemma** *MethodMain-unique-in-P*:
**assumes** *PROG P* ⊢ *D sees M*:*Ts*→*T* = *mb in C*
**shows** *MethodMain P* ≠ *M*
⟨*proof*⟩

**lemma** *ClassMain-is-no-class* [*dest*!]: *is-class* (*PROG P*) (*ClassMain P*) ⟹ *False*
⟨*proof*⟩

**lemma** *MethodMain-not-seen* [*dest*!]: *PROG P* ⊢ *C sees* (*MethodMain P*):*Ts*→*T* = *mb in D* ⟹ *False*
⟨*proof*⟩

**lemma** *no-Call-from-ClassMain* [*dest*!]: *PROG P* ⊢ *ClassMain P sees M*:*Ts*→*T* = *mb in C* ⟹ *False*
⟨*proof*⟩

**lemma** *no-Call-in-ClassMain* [*dest*!]: *PROG P* ⊢ *C sees M*:*Ts*→*T* = *mb in ClassMain P* ⟹ *False*

⟨*proof*⟩

**inductive** *JVMCFG* :: *jvm-method* ⇒ *cfg-node* ⇒ (*var*, *val*, *cname* × *mname* × *pc*, *cname* × *mname*) *edge-kind* ⇒ *cfg-node* ⇒ *bool* (‹ - ⊢ - −-→ -›)
**and** *reachable* :: *jvm-method* ⇒ *cfg-node* ⇒ *bool* (‹ - ⊢ ⇒-›)
**where**
*Entry-reachable*: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, *None*, *Enter*)
| *reachable-step*: ⟦ *P* ⊢ ⇒*n*; *P* ⊢ *n* −(*e*)→ *n'* ⟧ ⟹ *P* ⊢ ⇒*n'*
| *Main-to-Call*: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Enter*)
⟹ (*P*, *C0*, *Main*) ⊢ (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Enter*) −⇑*id*→ (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)
| *Main-Call-LFalse*: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)
⟹ (*P*, *C0*, *Main*) ⊢ (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*) −(λ*s*. *False*)$_{\surd}$→ (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Return*)
| *Main-Call*: ⟦ (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*);
*PROG P* ⊢ *C0 sees Main*:[]→*T* = (*mxs*, $mxl_0$, *is*, *xt*) *in D*;
*initParams* = [(λ*s*. *s Heap*),(λ*s*. ⌊*Value Null*⌋)];
*ek* = (λ(*s*, *ret*). *True*):(*ClassMain P*, *MethodMain P*, *0*)↪$_{(D,\ Main)}$*initParams* ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*) −(*ek*)→ (*D*, *Main*, *None*, *Enter*)
| *Main-Return-to-Exit*: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Return*)
⟹ (*P*, *C0*, *Main*) ⊢ (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Return*) −(⇑*id*)→ (*ClassMain P*, *MethodMain P*, *None*, *Return*)
| *Method-LFalse*: (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, *None*, *Enter*)
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, *None*, *Enter*) −(λ*s*. *False*)$_{\surd}$→ (*C*, *M*, *None*, *Return*)
| *Method-LTrue*: (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, *None*, *Enter*)
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, *None*, *Enter*) −(λ*s*. *True*)$_{\surd}$→ (*C*, *M*, ⌊*0*⌋, *Enter*)
| *CFG-Load*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*); *instrs-of* (*PROG P*) *C M* ! *pc* = *Load n*;
*ek* = ⇑(λ*s*. *s*(*Stack* (*stkLength* (*P*, *C*, *M*) *pc*) := *s* (*Local n*))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
| *CFG-Store*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*); *instrs-of* (*PROG P*) *C M* ! *pc* = *Store n*;
*ek* = ⇑(λ*s*. *s*(*Local n* := *s* (*Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
| *CFG-Push*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*); *instrs-of* (*PROG P*) *C M* ! *pc* = *Push v*;
*ek* = ⇑(λ*s*. *s*(*Stack* (*stkLength* (*P*, *C*, *M*) *pc*) ↦ *Value v*)) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
| *CFG-Pop*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*); *instrs-of* (*PROG P*) *C M* ! *pc* = *Pop*;
*ek* = ⇑*id* ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
| *CFG-IAdd*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);

*instrs-of* (*PROG P*) *C M* ! *pc* = *IAdd*;
*ek* = ⇑(λ*s*. *let i1* = *the-Intg* (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *1*));
*i2* = *the-Intg* (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *2*))
*in s*(*Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *2*) ↦ *Value* (*Intg* (*i1* + *i2*))))
⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
| *CFG-Goto*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*); *instrs-of* (*PROG P*) *C M* ! *pc* = *Goto i* ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −((λ*s*. *True*)$_{\surd}$)→ (*C*, *M*, ⌊*nat* (*int pc* + *i*)⌋, *Enter*)
| *CFG-CmpEq*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*); *instrs-of* (*PROG P*) *C M* ! *pc* = *CmpEq*;
*ek* = ⇑(λ*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*) ↦ *Value* (*Bool* (*e1* = *e2*))))
⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
| *CFG-IfFalse-False*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *IfFalse i*;
*i* ≠ *1*;
*ek* = (λ*s*. *stkAt s* (*stkLength*(*P*, *C*, *M*) *pc* − *1*) = *Bool False*)$_{\surd}$ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*nat* (*int pc* + *i*)⌋, *Enter*)
| *CFG-IfFalse-True*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *IfFalse i*;
*ek* = (λ*s*. *stkAt s* (*stkLength*(*P*, *C*, *M*) *pc* − *1*) ≠ *Bool False* ∨ *i* = *1*)$_{\surd}$ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
| *CFG-New-Check-Normal*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *New Cl*;
*ek* = (λ*s*. *new-Addr* (*heap-of s*) ≠ *None*)$_{\surd}$ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*⌋, *Normal*)
| *CFG-New-Check-Exceptional*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *New Cl*;
*pc′* = (*case* (*match-ex-table* (*PROG P*) *OutOfMemory pc* (*ex-table-of* (*PROG P*) *C M*)) *of*
*None* ⇒ *None*
| *Some* (*pc″*, *d*) ⇒ ⌊*pc″*⌋);
*ek* = (λ*s*. *new-Addr* (*heap-of s*) = *None*)$_{\surd}$ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*⌋, *Exceptional pc′ Enter*)
| *CFG-New-Update*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Normal*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *New Cl*;
*ek* = ⇑(λ*s*. *let a* = *the* (*new-Addr* (*heap-of s*))
*in s*(*Heap* ↦ *Hp* ((*heap-of s*)(*a* ↦ *blank* (*PROG P*) *Cl*)),
*Stack* (*stkLength*(*P*, *C*, *M*) *pc*) ↦ *Value* (*Addr a*))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Normal*) −(*ek*)→ (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)

| *CFG-New-Exceptional-prop*: $[\![$ $C \neq ClassMain\ P$; $(P, C0, Main) \vdash \Rightarrow(C, M, \lfloor pc \rfloor, Exceptional\ None\ Enter)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = New\ Cl$;
$ek = \Uparrow(\lambda s.\ s(Exception \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ OutOfMemory))))$ $]\!]$
$\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Exceptional\ None\ Enter)\ -(ek)\rightarrow\ (C, M, None, Return)$
| *CFG-New-Exceptional-handle*: $[\![$ $C \neq ClassMain\ P$; $(P, C0, Main) \vdash \Rightarrow(C, M, \lfloor pc \rfloor, Exceptional\ \lfloor pc' \rfloor\ Enter)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = New\ Cl$;
$ek = \Uparrow(\lambda s.\ (s(Exception := None))$
$(Stack\ (stkLength\ (P, C, M)\ pc' - 1) \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ OutOfMemory))))$ $]\!]$
$\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Exceptional\ \lfloor pc' \rfloor\ Enter)\ -(ek)\rightarrow\ (C, M, \lfloor pc' \rfloor, Enter)$
| *CFG-Getfield-Check-Normal*: $[\![$ $C \neq ClassMain\ P$; $(P, C0, Main) \vdash \Rightarrow(C, M, \lfloor pc \rfloor, Enter)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Getfield\ F\ Cl$;
$ek = (\lambda s.\ stkAt\ s\ (stkLength\ (P, C, M)\ pc - 1) \neq Null)_{\surd}$ $]\!]$
$\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter)\ -(ek)\rightarrow\ (C, M, \lfloor pc \rfloor, Normal)$
| *CFG-Getfield-Check-Exceptional*: $[\![$ $C \neq ClassMain\ P$; $(P, C0, Main) \vdash \Rightarrow(C, M, \lfloor pc \rfloor, Enter)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Getfield\ F\ Cl$;
$pc' = (case\ (match\text{-}ex\text{-}table\ (PROG\ P)\ NullPointer\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M))\ of$
$None \Rightarrow None$
$\mid Some\ (pc'', d) \Rightarrow \lfloor pc'' \rfloor)$;
$ek = (\lambda s.\ stkAt\ s\ (stkLength\ (P, C, M)\ pc - 1) = Null)_{\surd}$ $]\!]$
$\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Enter)\ -(ek)\rightarrow\ (C, M, \lfloor pc \rfloor, Exceptional\ pc'\ Enter)$
| *CFG-Getfield-Update*: $[\![$ $C \neq ClassMain\ P$; $(P, C0, Main) \vdash \Rightarrow(C, M, \lfloor pc \rfloor, Normal)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Getfield\ F\ Cl$;
$ek = \Uparrow(\lambda s.\ let\ (D, fs) = the\ (heap\text{-}of\ s\ (the\text{-}Addr\ (stkAt\ s\ (stkLength\ (P, C, M)\ pc - 1))))$
$in\ s(Stack\ (stkLength(P, C, M)\ pc - 1) \mapsto Value\ (the\ (fs\ (F, Cl)))))$
$]\!]$
$\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Normal)\ -(ek)\rightarrow\ (C, M, \lfloor Suc\ pc \rfloor, Enter)$
| *CFG-Getfield-Exceptional-prop*: $[\![$ $C \neq ClassMain\ P$; $(P, C0, Main) \vdash \Rightarrow(C, M, \lfloor pc \rfloor, Exceptional\ None\ Enter)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Getfield\ F\ Cl$;
$ek = \Uparrow(\lambda s.\ s(Exception \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer))))$ $]\!]$
$\Longrightarrow (P, C0, Main) \vdash (C, M, \lfloor pc \rfloor, Exceptional\ None\ Enter)\ -(ek)\rightarrow\ (C, M, None, Return)$
| *CFG-Getfield-Exceptional-handle*: $[\![$ $C \neq ClassMain\ P$; $(P, C0, Main) \vdash \Rightarrow(C, M, \lfloor pc \rfloor, Exceptional\ \lfloor pc' \rfloor\ Enter)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Getfield\ F\ Cl$;
$ek = \Uparrow(\lambda s.\ (s(Exception := None))$
$(Stack\ (stkLength\ (P, C, M)\ pc' - 1) \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer))))$ $]\!]$

$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Enter) -(ek)\rightarrow (C,\ M,\ \lfloor pc' \rfloor,\ Enter)$

$\mid$ *CFG-Putfield-Check-Normal*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Enter);$

$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Putfield\ F\ Cl;$

$ek = (\lambda s.\ stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 2) \neq Null)_{\surd}\ ]\!]$

$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter) -(ek)\rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Normal)$

$\mid$ *CFG-Putfield-Check-Exceptional*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Enter);$

$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Putfield\ F\ Cl;$

$pc' = (case\ (match\text{-}ex\text{-}table\ (PROG\ P)\ NullPointer\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M))\ of$

$None \Rightarrow None$

$\mid Some\ (pc'',\ d) \Rightarrow \lfloor pc'' \rfloor);$

$ek = (\lambda s.\ stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 2) = Null)_{\surd}\ ]\!]$

$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter) -(ek)\rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ pc'\ Enter)$

$\mid$ *CFG-Putfield-Update*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Normal);$

$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Putfield\ F\ Cl;$

$ek = \Uparrow(\lambda s.\ let\ v = stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 1);$

$r = stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 2);$

$a = the\text{-}Addr\ r;$

$(D,\ fs) = the\ (heap\text{-}of\ s\ a);$

$h' = (heap\text{-}of\ s)(a \mapsto (D,\ fs((F,\ Cl) \mapsto v)))$

$in\ s(Heap \mapsto Hp\ h'))\ ]\!]$

$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Normal) -(ek)\rightarrow (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$

$\mid$ *CFG-Putfield-Exceptional-prop*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ None\ Enter);$

$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Putfield\ F\ Cl;$

$ek = \Uparrow(\lambda s.\ s(Exception \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer))))\ ]\!]$

$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ None\ Enter) -(ek)\rightarrow (C,\ M,\ None,\ Return)$

$\mid$ *CFG-Putfield-Exceptional-handle*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Enter);$

$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Putfield\ F\ Cl;$

$ek = \Uparrow(\lambda s.\ (s(Exception := None))$

$(Stack\ (stkLength\ (P,\ C,\ M)\ pc' - 1) \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer))))\ ]\!]$

$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Enter) -(ek)\rightarrow (C,\ M,\ \lfloor pc' \rfloor,\ Enter)$

$\mid$ *CFG-Checkcast-Check-Normal*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Enter);$

$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Checkcast\ Cl;$

$ek = (\lambda s.\ cast\text{-}ok\ (PROG\ P)\ Cl\ (heap\text{-}of\ s)\ (stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 1)))_{\surd}\ ]\!]$

$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter) -(ek)\rightarrow (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$

$\mid$ *CFG-Checkcast-Check-Exceptional*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Enter);$

*instrs-of* (*PROG P*) *C M* ! *pc* = *Checkcast Cl*;
*pc′* = (*case* (*match-ex-table* (*PROG P*) *ClassCast pc* (*ex-table-of* (*PROG P*) *C M*)) *of*
*None* ⇒ *None*
| *Some* (*pc″*, *d*) ⇒ ⌊*pc″*⌋);
*ek* = (λ*s*. ¬ *cast-ok* (*PROG P*) *Cl* (*heap-of s*) (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *1*)))$_{\surd}$ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*⌋, *Exceptional pc′ Enter*)
| *CFG-Checkcast-Exceptional-prop*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *Checkcast Cl*;
*ek* = ⇑(λ*s*. *s*(*Exception* ↦ *Value* (*Addr* (*addr-of-sys-xcpt ClassCast*)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*) −(*ek*)→ (*C*, *M*, *None*, *Return*)
| *CFG-Checkcast-Exceptional-handle*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *Checkcast Cl*;
*ek* = ⇑(λ*s*. (*s*(*Exception* := *None*))
(*Stack* (*stkLength* (*P*, *C*, *M*) *pc′* − *1*) ↦ *Value* (*Addr* (*addr-of-sys-xcpt ClassCast*)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc′*⌋, *Enter*)
| *CFG-Throw-Check*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *Throw*;
*pc′* = *None* ∨ *match-ex-table* (*PROG P*) *Exc pc* (*ex-table-of* (*PROG P*) *C M*) = ⌊(*the pc′*, *d*)⌋;
*ek* = (λ*s*. *let v* = *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *1*);
*Cl* = *if* (*v* = *Null*) *then NullPointer else* (*cname-of* (*heap-of s*) (*the-Addr v*))
*in case pc′ of*
*None* ⇒ *match-ex-table* (*PROG P*) *Cl pc* (*ex-table-of* (*PROG P*) *C M*) = *None*
| *Some pc″* ⇒ ∃ *d*. *match-ex-table* (*PROG P*) *Cl pc* (*ex-table-of* (*PROG P*) *C M*)
= ⌊(*pc″*, *d*)⌋
)$_{\surd}$ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*⌋, *Exceptional pc′ Enter*)

| *CFG-Throw-prop*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *Throw*;
*ek* = ⇑(λ*s*. *s*(*Exception* ↦ *Value* (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *1*)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*) −(*ek*)→ (*C*, *M*, *None*, *Return*)
| *CFG-Throw-handle*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*);

$pc' \neq$ *length* (*instrs-of* (*PROG P*) *C M*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *Throw*;
*ek* = ⇑(λ*s*. (*s*(*Exception* := *None*))
(*Stack* (*stkLength* (*P*, *C*, *M*) $pc' - 1$) ↦ *Value* (*stkAt s* (*stkLength* (*P*, *C*, *M*) $pc - 1$)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc'*⌋, *Enter*)
| *CFG-Invoke-Check-NP-Normal*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*;
*ek* = (λ*s*. *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc n*) ≠ *Null*)$_{\surd}$ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*⌋, *Normal*)
| *CFG-Invoke-Check-NP-Exceptional*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*;
*pc'* = (*case* (*match-ex-table* (*PROG P*) *NullPointer pc* (*ex-table-of* (*PROG P*) *C M*)) *of*
*None* ⇒ *None*
| *Some* (*pc''*, *d*) ⇒ ⌊*pc''*⌋);
*ek* = (λ*s*. *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc n*) = *Null*)$_{\surd}$ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*⌋, *Exceptional pc' Enter*)
| *CFG-Invoke-NP-prop*: ⟦ *C* ≠ *ClassMain P*;
(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*;
*ek* = ⇑(λ*s*. *s*(*Exception* ↦ *Value* (*Addr* (*addr-of-sys-xcpt NullPointer*)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*) −(*ek*)→ (*C*, *M*, *None*, *Return*)
| *CFG-Invoke-NP-handle*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*;
*ek* = ⇑(λ*s*. (*s*(*Exception* := *None*))
(*Stack* (*stkLength* (*P*, *C*, *M*) $pc' - 1$) ↦ *Value* (*Addr* (*addr-of-sys-xcpt NullPointer*)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc'*⌋, *Enter*)
| *CFG-Invoke-Call*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Normal*);
*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*;
*TYPING P C M* ! *pc* = ⌊(*ST*, *LT*)⌋;
*ST* ! *n* = *Class D'*;
*PROG P* ⊢ *D'* *sees M'*:*Ts*→*T* = (*mxs*, $mxl_0$, *is*, *xt*) *in D*;
*Q* = (λ(*s*, *ret*). *let r* = *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc n*);
*C'* = *fst* (*the* (*heap-of s* (*the-Addr r*)))
*in D* = *fst* (*method* (*PROG P*) *C' M'*));
*paramDefs* = (λ*s*. *s Heap*)
\# (λ*s*. *s* (*Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc n*)))
\# (*rev* (*map* (λ*i*. (λ*s*. *s* (*Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc i*)))) [*0*..<*n*]));

$ek = Q{:}(C,\ M,\ pc) \hookrightarrow_{(D, M')} paramDefs$
⟧
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Normal)\ -(ek) \rightarrow\ (D,\ M',\ None,\ Enter)$
| *CFG-Invoke-False*: ⟦ $C \neq ClassMain\ P$; $(P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Normal)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n$;
$ek = (\lambda s.\ False)_{\surd}$
⟧
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Normal)\ -(ek) \rightarrow\ (C,\ M,\ \lfloor pc \rfloor,\ Return)$
| *CFG-Invoke-Return-Check-Normal*: ⟦ $C \neq ClassMain\ P$; $(P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Return)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n$;
$(TYPING\ P)\ C\ M\ !\ pc = \lfloor (ST,\ LT) \rfloor$;
$ST\ !\ n \neq NT$;
$ek = (\lambda s.\ s\ Exception = None)_{\surd}$
⟧
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Return)\ -(ek) \rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$
| *CFG-Invoke-Return-Check-Exceptional*: ⟦ $C \neq ClassMain\ P$; $(P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Return)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n$;
$match\text{-}ex\text{-}table\ (PROG\ P)\ Exc\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M) = \lfloor (pc',\ diff) \rfloor$;
$pc' \neq length\ (instrs\text{-}of\ (PROG\ P)\ C\ M)$;
$ek = (\lambda s.\ \exists\, v\ d.\ s\ Exception = \lfloor v \rfloor\ \wedge$
$match\text{-}ex\text{-}table\ (PROG\ P)\ (cname\text{-}of\ (heap\text{-}of\ s)\ (the\text{-}Addr\ (the\text{-}Value\ v)))\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M) = \lfloor (pc',\ d) \rfloor)_{\surd}$
⟧
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Return)\ -(ek) \rightarrow\ (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Return)$
| *CFG-Invoke-Return-Exceptional-handle*: ⟦ $C \neq ClassMain\ P$; $(P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Return)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n$;
$ek = \Uparrow(\lambda s.\ s(Exception := None,$
$Stack\ (stkLength\ (P,\ C,\ M)\ pc' - 1) := s\ Exception))$ ⟧
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Return)\ -(ek) \rightarrow\ (C,\ M,\ \lfloor pc' \rfloor,\ Enter)$
| *CFG-Invoke-Return-Exceptional-prop*: ⟦ $C \neq ClassMain\ P$;
$(P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Return)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n$;
$ek = (\lambda s.\ \exists\, v.\ s\ Exception = \lfloor v \rfloor\ \wedge$
$match\text{-}ex\text{-}table\ (PROG\ P)\ (cname\text{-}of\ (heap\text{-}of\ s)\ (the\text{-}Addr\ (the\text{-}Value\ v)))\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M) = None)_{\surd}$ ⟧
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Return)\ -(ek) \rightarrow\ (C,\ M,\ None,\ Return)$
| *CFG-Return*: ⟦ $C \neq ClassMain\ P$; $(P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Enter)$;
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = instr.Return$;
$ek = \Uparrow(\lambda s.\ s(Stack\ 0 := s\ (Stack\ (stkLength\ (P,\ C,\ M)\ pc - 1))))$
⟧
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek) \rightarrow\ (C,\ M,\ None,\ Return)$
| *CFG-Return-from-Method*: ⟦ $(P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ None,\ Return)$;
$(P,\ C0,\ Main) \vdash (C',\ M',\ \lfloor pc' \rfloor,\ Normal)\ -(Q'{:}(C',\ M',\ pc') \hookrightarrow_{(C,M)} ps) \rightarrow\ (C,$

$M$, $None$, $Enter$);
  $Q = (\lambda(s, ret).\ ret = (C', M', pc'))$;
  $stateUpdate = (\lambda s\ s'.\ s'(Heap := s\ Heap,$
    $Exception := s\ Exception,$
    $Stack\ (stkLength\ (P, C', M')\ (Suc\ pc') - 1) := s\ (Stack\ 0))$
  );
  $ek = Q\hookleftarrow_{(C,\ M)}stateUpdate$
 $]\!]$
 $\Longrightarrow (P, C0, Main) \vdash (C, M, None, Return) -(ek)\rightarrow (C', M', \lfloor pc' \rfloor, Return)$

**lemma** *JVMCFG-edge-det*: $[\![\ P \vdash n -(et)\rightarrow n';\ P \vdash n -(et')\rightarrow n'\ ]\!] \Longrightarrow et = et'$
 ⟨*proof*⟩

**lemma** *sourcenode-reachable*: $P \vdash n -(ek)\rightarrow n' \Longrightarrow P \vdash \Rightarrow n$
 ⟨*proof*⟩

**lemma** *targetnode-reachable*:
 **assumes** *edge*: $P \vdash n -(ek)\rightarrow n'$
 **shows** $P \vdash \Rightarrow n'$
⟨*proof*⟩

**lemmas** *JVMCFG-reachable-inducts* = *JVMCFG-reachable.inducts*[*split-format* (*complete*)]

**lemma** *ClassMain-imp-MethodMain*:
 $(P, C0, Main) \vdash (C', M', pc', nt') -ek\rightarrow (ClassMain\ P, M, pc, nt) \Longrightarrow M = MethodMain\ P$
 $(P, C0, Main) \vdash \Rightarrow(ClassMain\ P, M, pc, nt) \Longrightarrow M = MethodMain\ P$
⟨*proof*⟩

**lemma** *ClassMain-no-Call-target* [*dest*!]:
 $(P, C0, Main) \vdash (C, M, pc, nt) -Q{:}(C', M', pc')\hookrightarrow_{(D,M'')}paramDefs\rightarrow (ClassMain\ P, M''', pc'', nt')$
 $\Longrightarrow False$
 **and**
 $(P, C0, Main) \vdash \Rightarrow(C, M, pc, nt) \Longrightarrow True$
 ⟨*proof*⟩

**lemma** *method-of-src-and-trg-exists*:
 $[\![\ (P, C0, Main) \vdash (C', M', pc', nt') -ek\rightarrow (C, M, pc, nt);\ C \neq ClassMain\ P;\ C' \neq ClassMain\ P\ ]\!]$
 $\Longrightarrow (\exists\ Ts\ T\ mb.\ (PROG\ P) \vdash C\ sees\ M{:}Ts\rightarrow T = mb\ in\ C) \land$
  $(\exists\ Ts\ T\ mb.\ (PROG\ P) \vdash C'\ sees\ M'{:}Ts\rightarrow T = mb\ in\ C')$
 **and** *method-of-reachable-node-exists*:
 $[\![\ (P, C0, Main) \vdash \Rightarrow(C, M, pc, nt);\ C \neq ClassMain\ P\ ]\!]$
 $\Longrightarrow \exists\ Ts\ T\ mb.\ (PROG\ P) \vdash C\ sees\ M{:}Ts\rightarrow T = mb\ in\ C$
⟨*proof*⟩

**lemma** $\llbracket$ $(P, C0, Main) \vdash (C', M', pc', nt') -ek\rightarrow (C, M, pc, nt)$; $C \neq ClassMain$ $P$; $C' \neq ClassMain\ P$ $\rrbracket$
$\Longrightarrow$ (*case pc of None* $\Rightarrow$ *True* |
$\lfloor pc'' \rfloor \Rightarrow (TYPING\ P)\ C\ M\ !\ pc'' \neq None \wedge pc'' < length\ (instrs\text{-}of\ (PROG\ P)\ C\ M)) \wedge$
(*case pc' of None* $\Rightarrow$ *True* |
$\lfloor pc'' \rfloor \Rightarrow (TYPING\ P)\ C'\ M'\ !\ pc'' \neq None \wedge pc'' < length\ (instrs\text{-}of\ (PROG\ P)\ C'\ M'))$
**and** *instr-of-reachable-node-typable*: $\llbracket$ $(P, C0, Main) \vdash \Rightarrow(C, M, pc, nt)$; $C \neq ClassMain\ P$ $\rrbracket$
$\Longrightarrow$ *case pc of None* $\Rightarrow$ *True* |
$\lfloor pc'' \rfloor \Rightarrow (TYPING\ P)\ C\ M\ !\ pc'' \neq None \wedge pc'' < length\ (instrs\text{-}of\ (PROG\ P)\ C\ M)$
⟨*proof*⟩

**lemma** *reachable-node-impl-wt-instr*:
**assumes** $(P, C0, Main) \vdash \Rightarrow(C, M, \lfloor pc \rfloor, nt)$
**and** $C \neq ClassMain\ P$
**shows** $\exists T\ mxs\ mpc\ xt.\ PROG\ P,T,mxs,mpc,xt \vdash (instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc),pc :: TYPING\ P\ C\ M$
⟨*proof*⟩

**lemma**
$\llbracket$ $(P, C0, Main) \vdash (C, M, pc, nt) -ek\rightarrow (C', M', pc', nt')$; $C \neq ClassMain\ P \vee C' \neq ClassMain\ P$ $\rrbracket$
$\Longrightarrow \exists T\ mb\ D.\ PROG\ P \vdash C0\ sees\ Main:[]\rightarrow T = mb\ in\ D$
**and** *reachable-node-impl-Main-ex*:
$\llbracket$ $(P, C0, Main) \vdash \Rightarrow(C, M, pc, nt)$; $C \neq ClassMain\ P$ $\rrbracket$
$\Longrightarrow \exists T\ mb\ D.\ PROG\ P \vdash C0\ sees\ Main:[]\rightarrow T = mb\ in\ D$
⟨*proof*⟩

**end**
**theory** *JVMInterpretation* **imports** *JVMCFG ../StaticInter/CFGExit* **begin**

## 3.2 Instatiation of the *CFG* locale

**abbreviation** *sourcenode* :: *cfg-edge* $\Rightarrow$ *cfg-node*
**where** *sourcenode e* $\equiv$ *fst e*

**abbreviation** *targetnode* :: *cfg-edge* $\Rightarrow$ *cfg-node*
**where** *targetnode e* $\equiv$ *snd(snd e)*

**abbreviation** *kind* :: *cfg-edge* $\Rightarrow$ (*var*, *val*, *cname* $\times$ *mname* $\times$ *pc*, *cname* $\times$ *mname*) *edge-kind*
**where** *kind e* $\equiv$ *fst(snd e)*

**definition** *valid-edge* :: *jvm-method* $\Rightarrow$ *cfg-edge* $\Rightarrow$ *bool*
**where** *valid-edge P e* $\equiv$ $P \vdash (sourcenode\ e) -(kind\ e)\rightarrow (targetnode\ e)$

**fun** *methods* :: *cname* ⇒ *JVMInstructions.jvm-method mdecl list* ⇒ ((*cname* × *mname*) × *var list* × *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* ⇒ ((*cname* × *mname*) × *var list* × *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*, $mxl_0$, *is*, *xt*)⌋
  ⟹ ((*C′*, *M*), *Heap* # *map Local* [*0*..<*length Ts*] @ [*Local* (*length Ts*)], [*Heap*, *Stack 0*, *Exception*])
  ∈ *set* (*methods C′ ms*)
  ⟨*proof*⟩

**lemma** *in-methods-in-msD*: ((*C*, *M*), *ins*, *outs*) ∈ *set* (*methods D ms*)
  ⟹ *M* ∈ *set* (*map fst ms*) ∧ *D* = *C*
  ⟨*proof*⟩

**lemma** *in-methods-in-msD′*: ((*C*, *M*), *ins*, *outs*) ∈ *set* (*methods D ms*)
  ⟹ ∃ *Ts T mb*. (*M*, *Ts*, *T*, *mb*) ∈ *set ms*
  ∧ *D* = *C*
  ∧ *ins* = *Heap* # (*map Local* [*0*..<*Suc* (*length Ts*)])
  ∧ *outs* = [*Heap*, *Stack 0*, *Exception*]
  ⟨*proof*⟩

**lemma** *in-set-methodsE*:
  **assumes** ((*C*, *M*), *ins*, *outs*) ∈ *set* (*methods D ms*)
  **obtains** *Ts T mb*
  **where** (*M*, *Ts*, *T*, *mb*) ∈ *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* ⊢ *D sees M*: *Ts*→*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*) ∈ *set* (*procs P*)
⟨*proof*⟩

**lemma** *distinct-methods*: *distinct* (*map fst ms*) ⟹ *distinct* (*map fst* (*methods C ms*))
⟨*proof*⟩

**lemma** *in-set-procsD*:

$((C, M), ins, out) \in set\ (procs\ P) \Longrightarrow \exists D\ fs\ ms.\ (C, D, fs, ms) \in set\ P \wedge M \in set\ (map\ fst\ ms)$
⟨proof⟩

**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** $outs = [Heap, Stack\ 0, Exception]$
⟨proof⟩

**lemma** *distinct-Local-vars* [*simp*]: $distinct\ (map\ Local\ [0..<n])$
⟨proof⟩

**lemma** *distinct-Stack-vars* [*simp*]: $distinct\ (map\ Stack\ [0..<n])$
⟨proof⟩

**inductive-set** *get-return-edges* :: $wf\text{-}jvmprog \Rightarrow cfg\text{-}edge \Rightarrow cfg\text{-}edge\ set$
**for** $P$ :: *wf-jvmprog*
**and** $a$ :: *cfg-edge*
**where**
$kind\ a = Q{:}(C, M, pc)\hookrightarrow_{(D,\ M')}paramDefs$
$\Longrightarrow ((D, M', None, Return),$
$(\lambda(s, ret).\ ret = (C, M, pc))\hookleftarrow_{(D,\ M')}(\lambda s\ s'.\ s'(Heap := s\ Heap, Exception := s\ Exception,$
$Stack\ (stkLength\ (P, C, M)\ (Suc\ pc) - 1) := s\ (Stack\ 0))),$
$(C, M, \lfloor pc \rfloor, Return)) \in (get\text{-}return\text{-}edges\ P\ a)$

**lemma** *get-return-edgesE* [*elim*!]:
**assumes** $a \in get\text{-}return\text{-}edges\ P\ a'$
**obtains** *Q C M pc D M′ paramDefs* **where**
$kind\ a' = Q{:}(C, M, pc)\hookrightarrow_{(D,\ M')}paramDefs$
**and** $a = ((D, M', None, Return),$
$(\lambda(s, ret).\ ret = (C, M, pc))\hookleftarrow_{(D,\ M')}(\lambda s\ s'.\ s'(Heap := s\ Heap, Exception := s\ Exception,$
$Stack\ (stkLength\ (P, C, M)\ (Suc\ pc) - 1) := s\ (Stack\ 0))),$
$(C, M, \lfloor pc \rfloor, Return))$
⟨proof⟩

**lemma** *distinct-class-names*: $distinct\text{-}fst\ (PROG\ P)$
⟨proof⟩

**lemma** *distinct-method-names*:
$class\ (PROG\ P)\ C = \lfloor (D, fs, ms) \rfloor \Longrightarrow distinct\text{-}fst\ ms$
⟨proof⟩

**lemma** *distinct-fst-is-distinct-fst*: *distinct-fst* = *BasicDefs.distinct-fst*
⟨*proof*⟩

**lemma** *ClassMain-not-in-set-PROG* [*dest*!]: (*ClassMain P*, *D*, *fs*, *ms*) ∈ *set* (*PROG P*) ⟹ *False*
⟨*proof*⟩

**lemma** *in-set-procsE*:
**assumes** ((*C*, *M*), *ins*, *outs*) ∈ *set* (*procs* (*PROG P*))
**obtains** *D fs ms Ts T mb*
**where** *class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋
**and** *PROG P* ⊢ *C sees M*:*Ts*→*T* = *mb in C*
**and** *ins* = *Heap* # (*map* (λ*n*. *Local n*) [*0*..<*Suc* (*length Ts*)])
**and** *outs* = [*Heap*, *Stack 0*, *Exception*]
⟨*proof*⟩

**declare** *has-method-def* [*simp*]

**interpretation** *JVMCFG-Interpret*:
*CFG sourcenode targetnode kind valid-edge* (*P*, *C0*, *Main*)
(*ClassMain P*, *MethodMain P*, *None*, *Enter*)
(λ(*C*, *M*, *pc*, *type*). (*C*, *M*)) *get-return-edges P*
((*ClassMain P*, *MethodMain P*),[],[]) # *procs* (*PROG P*) (*ClassMain P*, *MethodMain P*)
**for** *P C0 Main*
⟨*proof*⟩

**interpretation** *JVMCFG-Exit-Interpret*:
*CFGExit sourcenode targetnode kind valid-edge* (*P*, *C0*, *Main*)
(*ClassMain P*, *MethodMain P*, *None*, *Enter*)
(λ(*C*, *M*, *pc*, *type*). (*C*, *M*)) *get-return-edges P*
((*ClassMain P*, *MethodMain P*),[],[]) # *procs* (*PROG P*)
(*ClassMain P*, *MethodMain P*) (*ClassMain P*, *MethodMain P*, *None*, *Return*)
**for** *P C0 Main*
⟨*proof*⟩

**end**
**theory** *JVMCFG-wf* **imports** *JVMInterpretation* ../*StaticInter*/*CFGExit-wf* **begin**

**inductive-set** *Def* :: *wf-jvmprog* ⇒ *cfg-node* ⇒ *var set*
**for** *P* :: *wf-jvmprog*
**and** *n* :: *cfg-node*
**where**
*Def-Main-Heap*:
*n* = (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Return*)
⟹ *Heap* ∈ *Def P n*
| *Def-Main-Exception*:
*n* = (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Return*)

$\Longrightarrow$ *Exception* $\in$ *Def P n*
| *Def-Main-Stack-0*:
*n* = (*ClassMain P*, *MethodMain P*, $\lfloor 0 \rfloor$, *Return*)
$\Longrightarrow$ *Stack 0* $\in$ *Def P n*
| *Def-Load*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Enter*);
*C* $\neq$ *ClassMain P*;
*instrs-of* (*PROG P*) *C M* ! *pc* = *Load idx*;
*i* = *stkLength* (*P*, *C*, *M*) *pc*$]\!]$
$\Longrightarrow$ *Stack i* $\in$ *Def P n*
| *Def-Store*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Enter*);
*C* $\neq$ *ClassMain P*;
*instrs-of* (*PROG P*) *C M* ! *pc* = *Store idx* $]\!]$
$\Longrightarrow$ *Local idx* $\in$ *Def P n*
| *Def-Push*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Enter*);
*C* $\neq$ *ClassMain P*;
*instrs-of* (*PROG P*) *C M* ! *pc* = *Push v*;
*i* = *stkLength* (*P*, *C*, *M*) *pc* $]\!]$
$\Longrightarrow$ *Stack i* $\in$ *Def P n*
| *Def-IAdd*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Enter*);
*C* $\neq$ *ClassMain P*;
*instrs-of* (*PROG P*) *C M* ! *pc* = *IAdd*;
*i* = *stkLength* (*P*, *C*, *M*) *pc* − *2* $]\!]$
$\Longrightarrow$ *Stack i* $\in$ *Def P n*
| *Def-CmpEq*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Enter*);
*C* $\neq$ *ClassMain P*;
*instrs-of* (*PROG P*) *C M* ! *pc* = *CmpEq*;
*i* = *stkLength* (*P*, *C*, *M*) *pc* − *2* $]\!]$
$\Longrightarrow$ *Stack i* $\in$ *Def P n*
| *Def-New-Heap*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Normal*);
*C* $\neq$ *ClassMain P*;
*instrs-of* (*PROG P*) *C M* ! *pc* = *New Cl* $]\!]$
$\Longrightarrow$ *Heap* $\in$ *Def P n*
| *Def-New-Stack*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Normal*);
*C* $\neq$ *ClassMain P*;
*instrs-of* (*PROG P*) *C M* ! *pc* = *New Cl*;
*i* = *stkLength* (*P*, *C*, *M*) *pc* $]\!]$
$\Longrightarrow$ *Stack i* $\in$ *Def P n*
| *Def-Exception*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Exceptional pco nt*);
*C* $\neq$ *ClassMain P* $]\!]$
$\Longrightarrow$ *Exception* $\in$ *Def P n*
| *Def-Exception-handle*:

$\llbracket\ n = (C,\ M,\ \lfloor pc\rfloor,\ Exceptional\ \lfloor pc'\rfloor\ Enter);$
$C \neq ClassMain\ P;$
$i = stkLength\ (P,\ C,\ M)\ pc' - 1\ \rrbracket$
$\Longrightarrow Stack\ i \in Def\ P\ n$
$|\ Def\text{-}Exception\text{-}handle\text{-}return:$
$\llbracket\ n = (C,\ M,\ \lfloor pc\rfloor,\ Exceptional\ \lfloor pc'\rfloor\ Return);$
$C \neq ClassMain\ P;$
$i = stkLength\ (P,\ C,\ M)\ pc' - 1\ \rrbracket$
$\Longrightarrow Stack\ i \in Def\ P\ n$
$|\ Def\text{-}Getfield:$
$\llbracket\ n = (C,\ M,\ \lfloor pc\rfloor,\ Normal);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Getfield\ Cl\ Fd;$
$i = stkLength\ (P,\ C,\ M)\ pc - 1\ \rrbracket$
$\Longrightarrow Stack\ i \in Def\ P\ n$
$|\ Def\text{-}Putfield:$
$\llbracket\ n = (C,\ M,\ \lfloor pc\rfloor,\ Normal);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Putfield\ Cl\ Fd\ \rrbracket$
$\Longrightarrow Heap \in Def\ P\ n$
$|\ Def\text{-}Invoke\text{-}Return\text{-}Heap:$
$\llbracket\ n = (C,\ M,\ \lfloor pc\rfloor,\ Return);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n'\ \rrbracket$
$\Longrightarrow Heap \in Def\ P\ n$
$|\ Def\text{-}Invoke\text{-}Return\text{-}Exception:$
$\llbracket\ n = (C,\ M,\ \lfloor pc\rfloor,\ Return);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n'\ \rrbracket$
$\Longrightarrow Exception \in Def\ P\ n$
$|\ Def\text{-}Invoke\text{-}Return\text{-}Stack:$
$\llbracket\ n = (C,\ M,\ \lfloor pc\rfloor,\ Return);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n';$
$i = stkLength\ (P,\ C,\ M)\ (Suc\ pc) - 1\ \rrbracket$
$\Longrightarrow Stack\ i \in Def\ P\ n$
$|\ Def\text{-}Invoke\text{-}Call\text{-}Heap:$
$\llbracket\ n = (C,\ M,\ None,\ Enter);$
$C \neq ClassMain\ P\ \rrbracket$
$\Longrightarrow Heap \in Def\ P\ n$
$|\ Def\text{-}Invoke\text{-}Call\text{-}Local:$
$\llbracket\ n = (C,\ M,\ None,\ Enter);$
$C \neq ClassMain\ P;$
$i < locLength\ (P,\ C,\ M)\ 0\ \rrbracket$
$\Longrightarrow Local\ i \in Def\ P\ n$
$|\ Def\text{-}Return:$
$\llbracket\ n = (C,\ M,\ \lfloor pc\rfloor,\ Enter);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = instr.Return\ \rrbracket$

$\Longrightarrow$ *Stack 0* $\in$ *Def P n*

**inductive-set** *Use* :: *wf-jvmprog* $\Rightarrow$ *cfg-node* $\Rightarrow$ *var set*
**for** *P* :: *wf-jvmprog*
**and** *n* :: *cfg-node*
**where**
*Use-Main-Heap*:
*n* = (*ClassMain P*, *MethodMain P*, $\lfloor 0 \rfloor$, *Normal*)
$\Longrightarrow$ *Heap* $\in$ *Use P n*
| *Use-Load*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Enter*);
*C* $\neq$ *ClassMain P*;
*instrs-of* (*PROG P*) *C M* ! *pc* = *Load idx* $]\!]$
$\Longrightarrow$ *Local idx* $\in$ *Use P n*
| *Use-Enter-Stack*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Enter*);
*C* $\neq$ *ClassMain P*;
*case* (*instrs-of* (*PROG P*) *C M* ! *pc*)
*of Store n'* $\Rightarrow$ *d* = *1*
| *Getfield F Cl* $\Rightarrow$ *d* = *1*
| *Putfield F Cl* $\Rightarrow$ *d* = *2*
| *Checkcast Cl* $\Rightarrow$ *d* = *1*
| *Invoke M' n'* $\Rightarrow$ *d* = *Suc n'*
| *IAdd* $\Rightarrow$ *d* $\in$ {*1*, *2*}
| *IfFalse i* $\Rightarrow$ *d* = *1*
| *CmpEq* $\Rightarrow$ *d* $\in$ {*1* , *2*}
| *Throw* $\Rightarrow$ *d* = *1*
| *instr.Return* $\Rightarrow$ *d* = *1*
| - $\Rightarrow$ *False*;
*i* = *stkLength* (*P*, *C*, *M*) *pc* − *d* $]\!]$
$\Longrightarrow$ *Stack i* $\in$ *Use P n*
| *Use-Enter-Local*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Enter*);
*C* $\neq$ *ClassMain P*;
*instrs-of* (*PROG P*) *C M* ! *pc* = *Load n'* $]\!]$
$\Longrightarrow$ *Local n'* $\in$ *Use P n*
| *Use-Enter-Heap*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Enter*);
*C* $\neq$ *ClassMain P*;
*case* (*instrs-of* (*PROG P*) *C M* ! *pc*)
*of New Cl* $\Rightarrow$ *True*
| *Checkcast Cl* $\Rightarrow$ *True*
| *Throw* $\Rightarrow$ *True*
| - $\Rightarrow$ *False* $]\!]$
$\Longrightarrow$ *Heap* $\in$ *Use P n*
| *Use-Normal-Heap*:
$[\![$ *n* = (*C*, *M*, $\lfloor pc \rfloor$, *Normal*);
*C* $\neq$ *ClassMain P*;
*case* (*instrs-of* (*PROG P*) *C M* ! *pc*)

$\quad$ *of New Cl* $\Rightarrow$ *True*
$\quad$ | *Getfield F Cl* $\Rightarrow$ *True*
$\quad$ | *Putfield F Cl* $\Rightarrow$ *True*
$\quad$ | *Invoke M' n'* $\Rightarrow$ *True*
$\quad$ | - $\Rightarrow$ *False* $]\!]$
$\Longrightarrow$ *Heap* $\in$ *Use P n*
| *Use-Normal-Stack*:
$[\![$ $n = (C, M, \lfloor pc \rfloor, Normal)$;
$C \neq$ *ClassMain P*;
*case* (*instrs-of* (*PROG P*) *C M* ! *pc*)
$\quad$ *of Getfield F Cl* $\Rightarrow d = 1$
$\quad$ | *Putfield F Cl* $\Rightarrow d \in \{1, 2\}$
$\quad$ | *Invoke M' n'* $\Rightarrow d > 0 \wedge d \leq$ *Suc n'*
$\quad$ | - $\Rightarrow$ *False*;
$i =$ *stkLength* $(P, C, M)$ $pc - d$ $]\!]$
$\Longrightarrow$ *Stack i* $\in$ *Use P n*
| *Use-Return-Heap*:
$[\![$ $n = (C, M, \lfloor pc \rfloor, Return)$;
*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n'* $\vee$ *C* = *ClassMain P* $]\!]$
$\Longrightarrow$ *Heap* $\in$ *Use P n*
| *Use-Return-Stack*:
$[\![$ $n = (C, M, \lfloor pc \rfloor, Return)$;
(*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n'* $\wedge$ $i =$ *stkLength* $(P, C, M)$ (*Suc* *pc*) $- 1$) $\vee$
($C$ = *ClassMain P* $\wedge$ $i = 0$) $]\!]$
$\Longrightarrow$ *Stack i* $\in$ *Use P n*
| *Use-Return-Exception*:
$[\![$ $n = (C, M, \lfloor pc \rfloor, Return)$;
*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n'* $\vee$ *C* = *ClassMain P* $]\!]$
$\Longrightarrow$ *Exception* $\in$ *Use P n*
| *Use-Exceptional-Stack*:
$[\![$ $n = (C, M, \lfloor pc \rfloor, Exceptional\ opc'\ nt)$;
*case* (*instrs-of* (*PROG P*) *C M* ! *pc*)
$\quad$ *of Throw* $\Rightarrow$ *True*
$\quad$ | - $\Rightarrow$ *False*;
$i =$ *stkLength* $(P, C, M)$ $pc - 1$ $]\!]$
$\Longrightarrow$ *Stack i* $\in$ *Use P n*
| *Use-Exceptional-Exception*:
$[\![$ $n = (C, M, \lfloor pc \rfloor, Exceptional\ \lfloor pc' \rfloor\ Return)$;
*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n'* $]\!]$
$\Longrightarrow$ *Exception* $\in$ *Use P n*
| *Use-Method-Leave-Exception*:
$[\![$ $n = (C, M, None, Return)$;
$C \neq$ *ClassMain P* $]\!]$
$\Longrightarrow$ *Exception* $\in$ *Use P n*
| *Use-Method-Leave-Heap*:
$[\![$ $n = (C, M, None, Return)$;
$C \neq$ *ClassMain P* $]\!]$
$\Longrightarrow$ *Heap* $\in$ *Use P n*

| *Use-Method-Leave-Stack*:
  $\llbracket$ $n = (C, M, None, Return)$;
  $C \neq ClassMain\ P$ $\rrbracket$
  $\Longrightarrow Stack\ 0 \in Use\ P\ n$
| *Use-Method-Entry-Heap*:
  $\llbracket$ $n = (C, M, None, Enter)$;
  $C \neq ClassMain\ P$ $\rrbracket$
  $\Longrightarrow Heap \in Use\ P\ n$
| *Use-Method-Entry-Local*:
  $\llbracket$ $n = (C, M, None, Enter)$;
  $C \neq ClassMain\ P$;
  $i < locLength\ (P, C, M)\ 0$ $\rrbracket$
  $\Longrightarrow Local\ i \in Use\ P\ n$

**fun** *ParamDefs* :: *wf-jvmprog* ⇒ *cfg-node* ⇒ *var list*
**where**
  $ParamDefs\ P\ (C, M, \lfloor pc \rfloor, Return) = [Heap, Stack\ (stkLength\ (P, C, M)\ (Suc\ pc) - 1), Exception]$
  | $ParamDefs\ P\ (C, M, opc, nt) = []$

**function** *ParamUses* :: *wf-jvmprog* ⇒ *cfg-node* ⇒ *var set list*
**where**
  $ParamUses\ P\ (ClassMain\ P, MethodMain\ P, \lfloor 0 \rfloor, Normal) = [\{Heap\},\{\}]$
  |
  $M \neq MethodMain\ P \vee opc \neq \lfloor 0 \rfloor \vee nt \neq Normal$
  $\Longrightarrow ParamUses\ P\ (ClassMain\ P, M, opc, nt) = []$
  |
  $C \neq ClassMain\ P$
  $\Longrightarrow ParamUses\ P\ (C, M, opc, nt) = (case\ opc\ of\ None \Rightarrow []$
  $| \lfloor pc \rfloor \Rightarrow (case\ nt\ of\ Normal \Rightarrow (case\ (instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc)\ of$
      $Invoke\ M'\ n \Rightarrow ($
          $\{Heap\}\ \#\ rev\ (map\ (\lambda n.\ \{Stack\ (stkLength\ (P, C, M)\ pc - (Suc\ n))\})\ [0..<n + 1])$
      )
      $|\ \text{-} \Rightarrow [])$
    $|\ \text{-} \Rightarrow []$
    )
  )
  ⟨*proof*⟩
**termination** ⟨*proof*⟩

**lemma** *in-set-ParamDefsE*:
  $\llbracket$ $V \in set\ (ParamDefs\ P\ n)$;
  $\bigwedge C\ M\ pc.$ $\llbracket$ $n = (C, M, \lfloor pc \rfloor, Return)$;
        $V \in \{Heap, Stack\ (stkLength\ (P, C, M)\ (Suc\ pc) - 1), Exception\}$ $\rrbracket \Longrightarrow thesis$ $\rrbracket$
  $\Longrightarrow thesis$
  ⟨*proof*⟩

**lemma** *in-set-ParamUsesE*:
**assumes** *V-in-ParamUses*: $V \in \bigcup(set\ (ParamUses\ P\ n))$
**obtains** $n = (ClassMain\ P,\ MethodMain\ P,\ \lfloor 0 \rfloor,\ Normal)$ **and** $V = Heap$
| $C\ M\ pc\ M'\ n'\ i$ **where** $n = (C,\ M,\ \lfloor pc \rfloor,\ Normal)$ **and** $instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n'$
**and** $V = Heap \lor V = Stack\ (stkLength\ (P,\ C,\ M)\ pc - Suc\ i)$ **and** $i < Suc\ n'$ **and** $C \neq ClassMain\ P$
⟨*proof*⟩

**lemma** *sees-method-fun-wf*:
**assumes** $PROG\ P \vdash D\ sees\ M'{:}\ Ts \rightarrow T = (mxs,\ mxl_0,\ is,\ xt)\ in\ D$
**and** $(D,\ D',\ fs,\ ms) \in set\ (PROG\ P)$
**and** $(M',\ Ts',\ T',\ mxs',\ mxl_0',\ is',\ xt') \in set\ ms$
**shows** $Ts = Ts' \land T = T' \land mxs = mxs' \land mxl_0 = mxl_0' \land is = is' \land xt = xt'$
⟨*proof*⟩

**interpretation** *JVMCFG-wf*:
*CFG-wf sourcenode targetnode kind valid-edge* (*P*, *C0*, *Main*)
(*ClassMain P*, *MethodMain P*, *None*, *Enter*)
(λ(*C*, *M*, *pc*, *type*). (*C*, *M*)) *get-return-edges P*
((*ClassMain P*, *MethodMain P*),[],[]) # *procs* (*PROG P*)
(*ClassMain P*, *MethodMain P*)
*Def P Use P ParamDefs P ParamUses P*
**for** *P C0 Main*
⟨*proof*⟩

**interpretation** *JVMCFGExit-wf* :
*CFGExit-wf sourcenode targetnode kind valid-edge* (*P*, *C0*, *Main*)
(*ClassMain P*, *MethodMain P*, *None*, *Enter*)
(λ(*C*, *M*, *pc*, *type*). (*C*, *M*)) *get-return-edges P*
((*ClassMain P*, *MethodMain P*),[],[]) # *procs* (*PROG P*)
(*ClassMain P*, *MethodMain P*)
(*ClassMain P*, *MethodMain P*, *None*, *Return*)
*Def P Use P ParamDefs P ParamUses P*
⟨*proof*⟩

**end**
**theory** *JVMPostdomination* **imports** *JVMInterpretation ../StaticInter/Postdomination*
**begin**

**context** *CFG* **begin**

**lemma** *vp-snocI*:
$[\![n\ {-as\rightarrow_{\surd}*}\ n';\ n'\ {-[a]\rightarrow*}\ n'';\ \forall\ Q\ p\ ret\ fs.\ kind\ a \neq Q\hookleftarrow_{p}ret\ ]\!] \Longrightarrow n\ {-as\ @\ [a]\rightarrow_{\surd}*}\ n''$
⟨*proof*⟩

**lemma** *valid-node-cases'* [*case-names Source Target*, *consumes 1*]:
$[\![\ valid\text{-}node\ n;\ \bigwedge e.\ [\![\ valid\text{-}edge\ e;\ sourcenode\ e = n\ ]\!] \Longrightarrow thesis;$

$\bigwedge e.$ ⟦ *valid-edge e*; *targetnode e* = *n* ⟧ ⟹ *thesis* ⟧
⟹ *thesis*
⟨*proof*⟩

**end**

**lemma** *disjE-strong*: ⟦*P* ∨ *Q*; *P* ⟹ *R*; ⟦*Q*; ¬ *P*⟧ ⟹ *R*⟧ ⟹ *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* ⇒ *cname* ⇒ *mname* ⇒ *cfg-edge list* ⇒ *cfg-node* ⇒ (*var*, *val*, *cname* × *mname* × *pc*, *cname* × *mname*) *edge-kind* ⇒ *cfg-node* ⇒ *bool*
**where** *vp-snoc P C0 Main as n ek n′*
≡ *JVMCFG-Interpret.valid-path′ P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) (*as* @ [(*n*,*ek*,*n′*)]) *n′*

**lemma**
(*P*, *C0*, *Main*) ⊢ (*C*, *M*, *pc*, *nt*) −*ek*→ (*C′*, *M′*, *pc′*, *nt′*)
⟹ (∃ *as*. *CFG.valid-path′ sourcenode targetnode kind* (*valid-edge* (*P*, *C0*, *Main*)) (*get-return-edges P*) (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, *pc*, *nt*)) ∧
(∃ *as*. *CFG.valid-path′ sourcenode targetnode kind* (*valid-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*) ⊢ ⇒(*C*, *M*, *pc*, *nt*)
⟹ ∃ *as*. *CFG.valid-path′ sourcenode targetnode kind* (*valid-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* :: $ty_P$
**where** *Phi-EP C M* = (*if C* = ′′*C*′′ ∧ *M* = ′′*M*′′
*then* [⌊([],[*OK* (*Class* ′′*C*′′)])⌋,⌊([*Void*],[*OK* (*Class* ′′*C*′′)])⌋] *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* < *Suc* (*Suc 0*) ⟹ *i* = *0* ∨ *i* = *1* ⟨*proof*⟩

**lemma** *EP-wf*: $\textit{wf-jvm-prog}_{\textit{Phi-EP}}$ *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*→*T* = (*mxs*, *mxl*, *is*, *xt*) *in D*
  ⟹ *C* = *''C''* ∧ *M* = *''M''* ∧ *Ts* = [] ∧ *T* = *Void* ∧ *mxs* = *1* ∧ *mxl* = *0* ∧
  *is* = [*Push Unit*, *instr.Return*] ∧ *xt* = [] ∧ *D* = *''C''*
  ⟨*proof*⟩

**lemma** [*simp*]:
  ∃ *T Ts mxs mxl is*. (∃ *xt*. *EP* ⊢ *''C'' sees ''M''*: *Ts*→*T* = (*mxs*, *mxl*, *is*, *xt*) *in*
*''C''*) ∧ *is* ≠ []
  ⟨*proof*⟩

**lemma** [*simp*]:
  ∃ *T Ts mxs mxl is*. (∃ *xt*. *EP* ⊢ *''C'' sees ''M''*: *Ts*→*T* = (*mxs*, *mxl*, *is*, *xt*) *in*
*''C''*) ∧
  *Suc 0* < *length is*
  ⟨*proof*⟩

**lemma** *C-sees-M-in-EP* [*simp*]:
  *EP* ⊢ *''C'' sees ''M''*: []→*Void* = (*Suc 0*, *0*, [*Push Unit*, *instr.Return*], []) *in ''C''*
⟨*proof*⟩

**lemma** *instrs-of-EP-C-M* [*simp*]:
  *instrs-of EP ''C'' ''M''* = [*Push Unit*, *instr.Return*]
  ⟨*proof*⟩

**lemma** *ClassMain-not-C* [*simp*]: *ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)) ≠ ''*C*''
⟨*proof*⟩

**lemma** *method-entry* [*dest*!]: (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), ''*C*'', ''*M*'') ⊢ ⇒(*C*, *M*, *None*, *Enter*)
⟹ (*C* = *ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)) ∧ *M* = *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)))
∨ (*C* = ''*C*'' ∧ *M* = ''*M*'')
⟨*proof*⟩

**lemma** *valid-node-in-EP-D*:
**assumes** *vn*: *JVMCFG-Interpret.valid-node* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)) ''*C*'' ''*M*'' *n*
**shows** *n* ∈ {
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *None*, *Enter*),
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *None*, *Return*),
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), ⌊*0*⌋, *Enter*),
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), ⌊*0*⌋, *Normal*),
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), ⌊*0*⌋, *Return*),
(''*C*'', ''*M*'', *None*, *Enter*),
(''*C*'', ''*M*'', ⌊*0*⌋, *Enter*),
(''*C*'', ''*M*'', ⌊*1*⌋, *Enter*),
(''*C*'', ''*M*'', *None*, *Return*)
}
⟨*proof*⟩

**lemma** *Main-Entry-valid* [*simp*]:
*JVMCFG-Interpret.valid-node* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)) ''*C*'' ''*M*''
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *None*, *Enter*)
⟨*proof*⟩

**lemma** *main-0-Enter-reachable* [*simp*]: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Enter*)
⟨*proof*⟩

**lemma** *main-0-Normal-reachable* [*simp*]: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)
⟨*proof*⟩

**lemma** *main-0-Return-reachable* [*simp*]: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Return*)
⟨*proof*⟩

**lemma** *Exit-reachable* [*simp*]: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, *None*, *Return*)
⟨*proof*⟩

**definition**
*cfg-wf-prog* =
{(*P*, *C0*, *Main*). (∀ *n*. *JVMCFG-Interpret.valid-node P C0 Main n* ⟶
(∃ *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* ⇒ ′*a*) ⇒ (*cfg-wf-prog* ⇒ ′*a*)
(‹-$_{CFG}$›)
**where** $f_{CFG}$ ≡ (λ*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*)
(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*)
**for** *P*
⟨*proof*⟩

**end**
**theory** *JVMSDG* **imports** *JVMCFG-wf JVMPostdomination ../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*)
(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*⟩

**interpretation** *JVM-SDG* :
*SDG 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*⟩

**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.