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

Daniel Wasserrab

December 14, 2021

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*:∀ *zs*. *xs* ≠ *ys*@*zs*
obtains *j zs* **where** *xs* = (*take j ys*)@*zs* **and** *j* < *length ys*
and ∀ *k* > *j*. ∀ *zs′*. *xs* ≠ (*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-fst $\equiv$ *distinct* $\circ$ *map fst*

lemma *distinct-fst-Nil* [*simp*]:
distinct-fst []
⟨*proof*⟩

lemma *distinct-fst-Cons* [*simp*]:
distinct-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*⟧
⟹ (*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*⟧
⟹ (*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* ⟹ (*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* $(Q\hookleftarrow_{p}f\ )(cf\#cfs)$ $= (case\ cfs\ of\ [] \Rightarrow []$
$| cf'\#cfs' \Rightarrow (f\ (fst\ cf)\ (fst\ cf'),snd\ cf')\#cfs')$
| *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* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*
(- −-→∗ - [*51*,*0*,*0*] *80*)
where
empty-path:*valid-node n* ⟹ *n* −[]→∗ *n*

| *Cons-path*:
⟦*n''* −*as*→∗ *n'*; *valid-edge a*; *sourcenode a* = *n*; *targetnode a* = *n''*⟧
⟹ *n* −*a*#*as*→∗ *n'*

lemma *path-valid-node*:
assumes *n* −*as*→∗ *n'* **shows** *valid-node n* **and** *valid-node n'*
⟨*proof*⟩

lemma *empty-path-nodes* [*dest*]:*n* −[]→∗ *n'* ⟹ *n* = *n'*
⟨*proof*⟩

lemma *path-valid-edges*:*n* −*as*→∗ *n'* ⟹ ∀ *a* ∈ *set as*. *valid-edge a*
⟨*proof*⟩

lemma *path-edge*:*valid-edge a* ⟹ *sourcenode a* −[*a*]→∗ *targetnode a*
⟨*proof*⟩

lemma *path-Append*:⟦*n* −*as*→∗ *n''*; *n''* −*as'*→∗ *n'*⟧
⟹ *n* −*as*@*as'*→∗ *n'*
⟨*proof*⟩

lemma *path-split*:
assumes *n* −*as*@*a*#*as'*→∗ *n'*
shows *n* −*as*→∗ *sourcenode a* **and** *valid-edge a* **and** *targetnode a* −*as'*→∗ *n'*
⟨*proof*⟩

lemma *path-split-Cons*:
assumes *n* −*as*→∗ *n'* **and** *as* ≠ []
obtains *a' as'* **where** *as* = *a'*#*as'* **and** *n* = *sourcenode a'*
and *valid-edge a'* **and** *targetnode a'* −*as'*→∗ *n'*
⟨*proof*⟩

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

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

definition
sourcenodes :: *'edge list* ⇒ *'node list*
where *sourcenodes xs* ≡ *map sourcenode xs*

definition
kinds :: *'edge list* ⇒ (*'var*,*'val*,*'ret*,*'pname*) *edge-kind list*
where *kinds xs* ≡ *map kind xs*

definition
targetnodes :: *'edge list* ⇒ *'node list*
where *targetnodes xs* ≡ *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*
(- $-\!\!-\rightarrow_\iota*$ - [*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 $valid\text{-}path\text{-}aux\ cs\ [] \longleftrightarrow True$
| $valid\text{-}path\text{-}aux\ cs\ (a\#as) \longleftrightarrow$
$(case\ (kind\ a)\ of\ Q{:}r\hookrightarrow_p fs \Rightarrow valid\text{-}path\text{-}aux\ (a\#cs)\ as$
$\mid Q\hookleftarrow_p f \Rightarrow case\ cs\ of\ [] \Rightarrow valid\text{-}path\text{-}aux\ []\ as$
$\mid c'\#cs' \Rightarrow a \in get\text{-}return\text{-}edges\ c' \wedge$
$valid\text{-}path\text{-}aux\ cs'\ as$
$\mid\ \text{-} \Rightarrow valid\text{-}path\text{-}aux\ cs\ as)$

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

$\bigwedge cs\ a\ as\ Q\ p\ f.\ [\![kind\ a = Q{\hookleftarrow_p}f;\ cs = [];\ valid\text{-}path\text{-}aux\ []\ as;\ P\ []\ as]\!]$
$\Longrightarrow P\ cs\ (a\#as)$
$\bigwedge cs\ a\ as\ Q\ p\ f\ c'\ cs'\ .\ [\![kind\ a = Q{\hookleftarrow_p}f;\ cs = c'\#cs';\ valid\text{-}path\text{-}aux\ cs'\ as;$
$a \in get\text{-}return\text{-}edges\ c';\ P\ cs'\ as]\!]$
$\Longrightarrow P\ cs\ (a\#as)$
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-cs cs* [] = *cs*
| *upd-cs cs* (*a*#*as*) =
(*case* (*kind a*) *of Q*:*r*↪*pfs* ⇒ *upd-cs* (*a*#*cs*) *as*
| *Q*↩*pf* ⇒ *case cs of* [] ⇒ *upd-cs cs as*
| *c'*#*cs'* ⇒ *upd-cs cs' as*
| - ⇒ *upd-cs cs as*)

lemma *upd-cs-empty* [*dest*]:
upd-cs cs [] = [] ⟹ *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-cs cs as* = [] **and** *cs* ≠ [] **and** *as* ≠ []
obtains *xs ys* **where** *as* = *xs*@*ys* **and** *xs* ≠ [] **and** *upd-cs cs xs* = []
and $\forall xs'\ ys'.\ xs = xs'@ys' \wedge ys' \neq [] \longrightarrow upd\text{-}cs\ cs\ xs' \neq []$
and *upd-cs* [] *ys* = []
⟨*proof*⟩

lemma *upd-cs-snoc-Return-Cons*:
assumes *kind a* = *Q*↩$_{pf}$
shows *upd-cs cs as* = *c*′#*cs*′ ⟹ *upd-cs cs* (*as*@[*a*]) = *cs*′
⟨*proof*⟩

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

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

lemma *valid-path-aux-Append*:
⟦*valid-path-aux cs as*; *valid-path-aux* (*upd-cs cs as*) *as*′⟧
⟹ *valid-path-aux cs* (*as*@*as*′)
⟨*proof*⟩

lemma *vpa-snoc-Call*:
assumes *kind a* = *Q*:*r*↪$_{p}$*fs*
shows *valid-path-aux cs as* ⟹ *valid-path-aux cs* (*as*@[*a*])
⟨*proof*⟩

definition *valid-path* :: ′*edge list* ⇒ *bool*
where *valid-path as* ≡ *valid-path-aux* [] *as*

lemma *valid-path-aux-valid-path*:
valid-path-aux cs as ⟹ *valid-path as*
⟨*proof*⟩

lemma *valid-path-split*:
assumes *valid-path* (*as*@*as*′) **shows** *valid-path as* **and** *valid-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*↩$_p$*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*↪$_p$*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*↪$_p$*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*↩$_p$*f* ⇒ *upd-rev-cs* (*a*#*cs*) *as*
| *Q*:*r*↪$_p$*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*↩$_p$*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*↩$_p$*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*↪$_p$*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*↪$_p$*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*↪$_p$*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*↩$_p$*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-edge r*
⟨*proof*⟩

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

lemma *vpa-upd-cs-cases*:
  ⟦*valid-path-aux cs as*; *valid-call-list cs n*; *n* −*as*→∗ *n*′⟧
  $\Longrightarrow$ *case* (*upd-cs cs as*) *of* [] ⇒ (∀ *c* ∈ *set cs*. ∃ *a* ∈ *set as*. *a* ∈ *get-return-edges c*)
  | *cx*#*csx* ⇒ *valid-call-list* (*cx*#*csx*) *n*′
⟨*proof*⟩

lemma *vpa-valid-call-list-valid-return-list-vpra*:
  ⟦*valid-path-aux cs cs*′; *valid-call-list cs n*; *valid-return-list cs*′ *n*′⟧
  $\Longrightarrow$ *valid-path-rev-aux cs*′ (*rev cs*)
⟨*proof*⟩

lemma *vpa-to-vpra*:
  ⟦*valid-path-aux cs as*; *valid-path-aux* (*upd-cs cs as*) *cs*′;
    n −*as*→∗ *n*′; *valid-call-list cs n*; *valid-return-list cs*′ *n*′′⟧
  $\Longrightarrow$ *valid-path-rev-aux cs*′ *as* ∧ *valid-path-rev-aux* (*upd-rev-cs cs*′ *as*) (*rev cs*)
⟨*proof*⟩

lemma *vp-to-vpra*:
  n −*as*→$_{\surd}$∗ *n*′ $\Longrightarrow$ *valid-path-rev-aux* [] *as*
⟨*proof*⟩

Same level paths

fun *same-level-path-aux* :: ′*edge list* ⇒ ′*edge list* ⇒ *bool*
  where *same-level-path-aux cs* [] ⟷ *True*
  | *same-level-path-aux cs* (*a*#*as*) ⟷
    (*case* (*kind a*) *of Q*:*r*↪$_{p}$*fs* ⇒ *same-level-path-aux* (*a*#*cs*) *as*
      | *Q*↩$_{p}$*f* ⇒ *case cs of* [] ⇒ *False*
        | *c*′#*cs*′ ⇒ *a* ∈ *get-return-edges c*′ ∧
          same-level-path-aux cs′ *as*
      | - ⇒ *same-level-path-aux cs as*)

lemma *slpa-induct* [*consumes 1*,*case-names slpa-empty slpa-intra slpa-Call slpa-Return*]:
assumes *major*: *same-level-path-aux xs ys*
and *rules*: $\bigwedge cs.\ P\ cs\ []$
$\bigwedge cs\ a\ as.\ [\![intra\text{-}kind(kind\ a);\ same\text{-}level\text{-}path\text{-}aux\ cs\ as;\ P\ cs\ as]\!]$
$\Longrightarrow P\ cs\ (a\#as)$
$\bigwedge cs\ a\ as\ Q\ r\ p\ fs.\ [\![kind\ a = Q{:}r{\hookrightarrow_p}fs;\ same\text{-}level\text{-}path\text{-}aux\ (a\#cs)\ as;\ P\ (a\#cs)\ as]\!]$
$\Longrightarrow P\ cs\ (a\#as)$
$\bigwedge cs\ a\ as\ Q\ p\ f\ c'\ cs'.\ [\![kind\ a = Q{\hookleftarrow_p}f;\ cs = c'\#cs';\ same\text{-}level\text{-}path\text{-}aux\ cs'\ as;$
$a \in get\text{-}return\text{-}edges\ c';\ P\ cs'\ as]\!]$
$\Longrightarrow P\ cs\ (a\#as)$
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-cs cs asx* = *c'#cs'* **and** *upd-cs cs* (*asx@*[*a*]) = []
and $a \in get\text{-}return\text{-}edges\ c'$ **and** *valid-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-level-path-aux cs as $\Longrightarrow$ *same-level-path-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* $= []$;
$\forall a \in$ *set as*. *valid-edge a*; $\forall c \in$ *set cs*. *valid-edge c*⟧
$\Longrightarrow$ *length* $[a \leftarrow as@cs.\ \exists Q\ r\ p\ fs.$ *kind* $a = Q{:}r{\hookrightarrow_p}fs] =$
length $[a \leftarrow as.\ \exists Q\ p\ f.$ *kind* $a = Q{\hookleftarrow_p}f]$
⟨*proof*⟩

lemma *slpa-get-proc*:
⟦*same-level-path-aux cs as*; *upd-cs cs as* $= []$; $n -as\rightarrow* n'$;
$\forall c \in$ *set cs*. *valid-edge c*⟧
$\Longrightarrow$ (*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* $\neq []$; *upd-cs cs as* $= []$;
$\forall xs\ ys.\ as = xs@ys \wedge ys \neq [] \longrightarrow$ *upd-cs cs xs* $\neq []$⟧
$\Longrightarrow$ *last as* $\in$ *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*)⟧
$\Longrightarrow$ *same-level-path-aux cs* (*as*@[*a*])
⟨*proof*⟩

lemma *slpa-snoc-Call*:
⟦*same-level-path-aux cs as*; *kind* $a = Q{:}r{\hookrightarrow_p}fs$⟧
$\Longrightarrow$ *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*↪*pfs*] = *length* [*a*←*as*. ∃ *Q p f*. *kind a* = *Q*↩*pf*]
⟨*proof*⟩

lemma *same-level-path-valid-path-Append*:
⟦*same-level-path as*; *valid-path as'*⟧ ⟹ *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*↩$_p$*f* ⇒ *same-level-path-rev-aux* (*a*#*cs*) *as*
| *Q*:*r*↪$_p$*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*↩$_p$*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*↪$_p$*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*↩*pf*; *targetnode a* = (*-Exit-*)⟧ ⟹ *False*
and *Entry-Exit-edge*: ∃ *a*. *valid-edge a* ∧ *sourcenode a* = (*-Entry-*) ∧
targetnode a = (*-Exit-*) ∧ *kind a* = (λ*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*↩*pf*)

lemma *method-exit-cases*:
⟦*method-exit n*; *n* = (*-Exit-*) ⟹ *P*;
⋀*a Q f p*. ⟦*n* = *sourcenode a*; *valid-edge a*; *kind a* = *Q*↩*pf*⟧ ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

lemma *method-exit-inner-path*:
assumes *method-exit n* **and** *n* −*as*→$_\iota$* *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* ≠ []⟧

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

lemma *valid-Exit-path-cases*:
assumes $n\ -as{\rightarrow}_{\surd}*$ (*-Exit-*) **and** $as \neq []$
shows $(\exists a'\ as'.\ as = a'\#as' \wedge intra\text{-}kind(kind\ a')) \vee$
$(\exists a'\ as'\ Q\ p\ f.\ as = a'\#as' \wedge kind\ a' = Q{\hookleftarrow}_p f) \vee$
$(\exists as'\ as''\ n'.\ as = as'@as'' \wedge as' \neq [] \wedge n\ -as'{\rightarrow}_{sl}*\ n')$
⟨*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 $\forall a' \in set\ as'.\ intra\text{-}kind(kind\ a') \vee (\exists Q\ f\ p.\ kind\ a' = Q{\hookleftarrow}_p f)$
⟨*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* ⇒ *'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 *Def*::*'node* ⇒ *'var set*
fixes *Use*::*'node* ⇒ *'var set*
fixes *ParamDefs*::*'node* ⇒ *'var list*
fixes *ParamUses*::*'node* ⇒ *'var set list*

assumes *Entry-empty*:*Def* (*-Entry-*) = {} ∧ *Use* (*-Entry-*) = {}
and *ParamUses-call-source-length*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*; (*p*,*ins*,*outs*) ∈ *set procs*⟧
⟹ *length*(*ParamUses* (*sourcenode a*)) = *length ins*
and *distinct-ParamDefs*:*valid-edge a* ⟹ *distinct* (*ParamDefs* (*targetnode a*))
and *ParamDefs-return-target-length*:
⟦*valid-edge a*; *kind a* = *Q*′↩$_p$*f*′; (*p*,*ins*,*outs*) ∈ *set procs*⟧
⟹ *length*(*ParamDefs* (*targetnode a*)) = *length outs*
and *ParamDefs-in-Def*:
⟦*valid-node n*; *V* ∈ *set* (*ParamDefs n*)⟧ ⟹ *V* ∈ *Def n*
and *ins-in-Def*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*; (*p*,*ins*,*outs*) ∈ *set procs*; *V* ∈ *set ins*⟧
⟹ *V* ∈ *Def* (*targetnode a*)
and *call-source-Def-empty*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*⟧ ⟹ *Def* (*sourcenode a*) = {}
and *ParamUses-in-Use*:
⟦*valid-node n*; *V* ∈ *Union* (*set* (*ParamUses n*))⟧ ⟹ *V* ∈ *Use n*
and *outs-in-Use*:
⟦*valid-edge a*; *kind a* = *Q*↩$_p$*f*; (*p*,*ins*,*outs*) ∈ *set procs*; *V* ∈ *set outs*⟧
⟹ *V* ∈ *Use* (*sourcenode a*)
and *CFG-intra-edge-no-Def-equal*:
⟦*valid-edge a*; *V* ∉ *Def* (*sourcenode a*); *intra-kind* (*kind a*); *pred* (*kind a*) *s*⟧
⟹ *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*
and *CFG-intra-edge-transfer-uses-only-Use*:
⟦*valid-edge a*; ∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s*′ *V*;
intra-kind (*kind a*); *pred* (*kind a*) *s*; *pred* (*kind a*) *s*′⟧
⟹ ∀ *V* ∈ *Def* (*sourcenode a*). *state-val* (*transfer* (*kind a*) *s*) *V* =
state-val (*transfer* (*kind a*) *s*′) *V*
and *CFG-edge-Uses-pred-equal*:
⟦*valid-edge a*; *pred* (*kind a*) *s*; *snd* (*hd s*) = *snd* (*hd s*′);
∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s*′ *V*; *length s* = *length s*′⟧
⟹ *pred* (*kind a*) *s*′
and *CFG-call-edge-length*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*; (*p*,*ins*,*outs*) ∈ *set procs*⟧
⟹ *length fs* = *length ins*
and *CFG-call-determ*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*; *valid-edge a*′; *kind a*′ = *Q*′:*r*′↪$_{p'}$*fs*′;
sourcenode a = *sourcenode a*′; *pred* (*kind a*) *s*; *pred* (*kind a*′) *s*⟧
⟹ *a* = *a*′
and *CFG-call-edge-params*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*; *i* < *length ins*;
(*p*,*ins*,*outs*) ∈ *set procs*; *pred* (*kind a*) *s*; *pred* (*kind a*) *s*′;
∀ *V* ∈ (*ParamUses* (*sourcenode a*))!*i*. *state-val s V* = *state-val s*′ *V*⟧
⟹ (*params fs* (*fst* (*hd s*)))!*i* = (*params fs* (*fst* (*hd s*′)))!*i*
and *CFG-return-edge-fun*:
⟦*valid-edge a*; *kind a* = *Q*′↩$_p$*f*′; (*p*,*ins*,*outs*) ∈ *set procs*⟧
⟹ *f*′ *vmap vmap*′ = *vmap*′(*ParamDefs* (*targetnode a*) [:=] *map vmap outs*)
and *deterministic*:⟦*valid-edge a*; *valid-edge a*′; *sourcenode a* = *sourcenode a*′;
targetnode a ≠ *targetnode a*′; *intra-kind* (*kind a*); *intra-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*↪$_p$*fs* **and** *valid-edge a'*
and *kind a'* = *Q'*:*r'*↪$_{p'}$*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 ∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s' V*
shows *a* = *a'*
⟨*proof*⟩

lemma *CFG-call-edge-param-in*:
assumes *valid-edge a* **and** *kind a* = *Q*:*r*↪$_p$*fs* **and** *i* < *length ins*
and (*p*,*ins*,*outs*) ∈ *set procs* **and** *pred* (*kind a*) *s* **and** *pred* (*kind a*) *s'*
and ∀ *V* ∈ (*ParamUses* (*sourcenode a*))!*i*. *state-val s V* = *state-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*↪$_p$*fs* **and** *V* ∉ *set ins*
and (*p*,*ins*,*outs*) ∈ *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*↩$_p$*f* **and** *i* < *length outs*
and (*p*,*ins*,*outs*) ∈ *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*→$_{sl}$* *n'* **and** *valid-edge a* **and** *a'* ∈ *get-return-edges a*
and *V* ∉ *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*!]: $V \in Use\ (\text{-}Entry\text{-}) \Longrightarrow False$
⟨*proof*⟩

lemma [*dest*!]: $V \in Def\ (\text{-}Entry\text{-}) \Longrightarrow False$
⟨*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'*; ∀ *a* ∈ *set as*. *valid-edge a*; *length cs* = *length cfsx*;
preds (*kinds as*) *s*⟧
⟹ *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* ⇒ *'var set list* +
assumes *Exit-empty*:*Def* (*-Exit-*) = {} ∧ *Use* (*-Exit-*) = {}

begin

lemma *Exit-Use-empty* [*dest*!]: *V* ∈ *Use* (*-Exit-*) ⟹ *False*
⟨*proof*⟩

lemma *Exit-Def-empty* [*dest*!]: *V* ∈ *Def* (*-Exit-*) ⟹ *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* ⇒ *'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 *sem*::*'com* ⇒ (*'var* ⇀ *'val*) *list* ⇒ *'com* ⇒ (*'var* ⇀ *'val*) *list* ⇒ *bool*
(((*1*⟨*-*,*/-*⟩) ⇒/ (*1*⟨*-*,*/-*⟩)) [*0*,*0*,*0*,*0*] *81*)
fixes *identifies*::*'node* ⇒ *'com* ⇒ *bool* (*-* ≙ *-* [*51*,*0*] *80*)
assumes *fundamental-property*:
⟦*n* ≙ *c*; ⟨*c*,[*cf*]⟩ ⇒ ⟨*c'*,*s'*⟩⟧ ⟹
∃ *n'* *as*. *n* −*as*→√* *n'* ∧ *n'* ≙ *c'* ∧ *preds* (*kinds as*) [(*cf*,*undefined*)] ∧
transfers (*kinds as*) [(*cf*,*undefined*)] = *cfs'* ∧ *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* ⇒ *bool*
where *return-node n* ≡ ∃ *a a'*. *valid-edge a* ∧ *n* = *targetnode a* ∧

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

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

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

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

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

lemma *return-node-call-of-return-node*:
return-node $n \Longrightarrow \exists!n'.$ *call-of-return-node* $n\ n'$
⟨*proof*⟩

lemma *call-of-return-nodes-det* [*dest*]:
assumes *call-of-return-node* $n\ n'$ **and** *call-of-return-node* $n\ n''$
shows $n' = n''$
⟨*proof*⟩

lemma *get-return-edges-call-of-return-nodes*:
⟦*valid-call-list cs m*; *valid-return-list rs m*;
$\forall\, i <$ *length rs*. *rs*!$i \in$ *get-return-edges* (*cs*!i); *length rs* = *length cs*⟧
$\Longrightarrow \forall\, i<$*length cs*. *call-of-return-node* (*targetnodes rs*!i) (*sourcenode* (*cs*!i))
⟨*proof*⟩

end

end

1.6 Observable Sets of Nodes

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

context *CFG* **begin**

1.6.1 Intraprocedural observable sets

inductive-set *obs-intra* :: *'node* ⇒ *'node set* ⇒ *'node set*
for *n*::*'node* **and** *S*::*'node set*
where *obs-intra-elem*:
⟦*n* −*as*→$_\iota$∗ *n'*; ∀ *nx* ∈ *set*(*sourcenodes as*). *nx* ∉ *S*; *n'* ∈ *S*⟧ ⟹ *n'* ∈ *obs-intra n S*

lemma *obs-intraE*:
assumes *n'* ∈ *obs-intra n S*
obtains *as* **where** *n* −*as*→$_\iota$∗ *n'* **and** ∀ *nx* ∈ *set*(*sourcenodes as*). *nx* ∉ *S* **and** *n'* ∈ *S*
⟨*proof*⟩

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

lemma *in-obs-intra-valid*:
assumes *n'* ∈ *obs-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* ∉ *S*
shows *obs-intra* (*targetnode a*) *S* ⊆ *obs-intra* (*sourcenode a*) *S*
⟨*proof*⟩

lemma *path-obs-intra-subset*:
assumes *n* −*as*→$_\iota$∗ *n'* **and** ∀ *n'* ∈ *set*(*sourcenodes as*). *n'* ∉ *S*
shows *obs-intra n' S* ⊆ *obs-intra n S*
⟨*proof*⟩

lemma *path-ex-obs-intra*:
assumes *n* −*as*→$_\iota$∗ *n'* **and** *n'* ∈ *S*
obtains *m* **where** *m* ∈ *obs-intra n S*
⟨*proof*⟩

1.6.2 Interprocedural observable sets restricted to the slice

fun *obs* :: *'node list* ⇒ *'node set* ⇒ *'node list set*
where *obs* [] *S* = {}
| *obs* (*n*#*ns*) *S* = (*let S'* = *obs-intra n S in*
(*if* (*S'* = {} ∨ (∃ *n'* ∈ *set ns*. ∃ *nx*. *call-of-return-node n' nx* ∧ *nx* ∉ *S*))
then obs ns S else (λ*nx*. *nx*#*ns*) ‘ *S'*))

lemma *obsI*:
assumes *n'* ∈ *obs-intra n S*
and ∀ *nx* ∈ *set nsx'*. ∃ *nx'*. *call-of-return-node nx nx'* ∧ *nx'* ∈ *S*
shows ⟦*ns* = *nsx*@*n*#*nsx'*; ∀ *xs x xs'*. *nsx* = *xs*@*x*#*xs'* ∧ *obs-intra x S* ≠ {}
⟶ (∃ *x''* ∈ *set* (*xs'*@[*n*]). ∃ *nx*. *call-of-return-node x'' nx* ∧ *nx* ∉ *S*)⟧
⟹ *n'*#*nsx'* ∈ *obs ns S*
⟨*proof*⟩

lemma *obsE* [*consumes 2*]:
assumes *ns'* ∈ *obs ns S* **and** ∀ *n* ∈ *set* (*tl ns*). *return-node n*
obtains *nsx n nsx' n'* **where** *ns* = *nsx*@*n*#*nsx'* **and** *ns'* = *n'*#*nsx'*
and *n'* ∈ *obs-intra n S*
and ∀ *nx* ∈ *set nsx'*. ∃ *nx'*. *call-of-return-node nx nx'* ∧ *nx'* ∈ *S*
and ∀ *xs x xs'*. *nsx* = *xs*@*x*#*xs'* ∧ *obs-intra x S* ≠ {}
⟶ (∃ *x''* ∈ *set* (*xs'*@[*n*]). ∃ *nx*. *call-of-return-node x'' nx* ∧ *nx* ∉ *S*)
⟨*proof*⟩

lemma *obs-split-det*:
assumes *xs*@*x*#*xs'* = *ys*@*y*#*ys'*
and *obs-intra x S* ≠ {}
and ∀ *x'* ∈ *set xs'*. ∃ *x''*. *call-of-return-node x' x''* ∧ *x''* ∈ *S*
and ∀ *zs z zs'*. *xs* = *zs*@*z*#*zs'* ∧ *obs-intra z S* ≠ {}
⟶ (∃ *z''* ∈ *set* (*zs'*@[*x*]). ∃ *nx*. *call-of-return-node z'' nx* ∧ *nx* ∉ *S*)
and *obs-intra y S* ≠ {}
and ∀ *y'* ∈ *set ys'*. ∃ *y''*. *call-of-return-node y' y''* ∧ *y''* ∈ *S*
and ∀ *zs z zs'*. *ys* = *zs*@*z*#*zs'* ∧ *obs-intra z S* ≠ {}
⟶ (∃ *z''* ∈ *set* (*zs'*@[*y*]). ∃ *ny*. *call-of-return-node z'' ny* ∧ *ny* ∉ *S*)
shows *xs* = *ys* ∧ *x* = *y* ∧ *xs'* = *ys'*
⟨*proof*⟩

lemma *in-obs-valid*:
assumes *ns'* ∈ *obs ns S* **and** ∀ *n* ∈ *set ns*. *valid-node n*
shows ∀ *n* ∈ *set ns'*. *valid-node n*
⟨*proof*⟩

end

end

1.7 Postdomination

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

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

locale *Postdomination* = *CFGExit sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit*
for *sourcenode* :: *'edge* ⇒ *'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*→$_{\surd}$* *n*
and *Exit-path*:*valid-node n* ⟹ ∃ *as*. *n* −*as*→$_{\surd}$* (*-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*→$_{\iota}$* *pex* ∧ *method-exit pex*) ⟶ *n'* ∈ *set* (*sourcenodes as*)))

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

lemma *postdominate-variant*:
assumes *n' postdominates n*
shows ∀ *as*. *n* −*as*→$_{\surd}$* (*-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*→∗ *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*→$_\iota$∗ *n''* **and** *n'* ∉ *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*→$_\iota$* *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*↪$_p$*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*↩$_p$*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*↪$_p$*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*↩$_p$*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 ∈ Def_{SDG} *n* ⟹ *valid-SDG-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-node* ⇒ *'var* ⇒ *'node SDG-node* ⇒ *bool*

(- *influences* - *in* - [*51*,*0*,*0*])
where *n influences V in n'* ≡ ∃ *as*. $(V \in Def_{SDG}\ n) \wedge (V \in Use_{SDG}\ n') \wedge$
(*parent-node n* $-as\rightarrow_\iota *$ *parent-node n'*) ∧
(∀ *n''*. *valid-SDG-node n''* ∧ *parent-node n''* ∈ *set* (*sourcenodes* (*tl as*))
⟶ $V \notin Def_{SDG}\ n''$)

1.8.3 Control dependence

definition *control-dependence* :: *'node* ⇒ *'node* ⇒ *bool*
(- *controls* - [*51*,*0*])
where *n controls n'* ≡ ∃ *a a' as*. *n* $-a\#as\rightarrow_\iota *$ *n'* ∧ *n'* ∉ *set*(*sourcenodes* (*a*#*as*))
∧
intra-kind(*kind a*) ∧ *n' postdominates* (*targetnode a*) ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧ *sourcenode a'* = *n* ∧
¬ *n' postdominates* (*targetnode a'*)

lemma *control-dependence-path*:
assumes *n controls n'* **obtains** *as* **where** *n* $-as\rightarrow_\iota *$ *n'* **and** *as* ≠ []
⟨*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'* ≠ (-*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* ∉ *set*(*sourcenodes* (*a*#*as'*)) **and** *valid-edge a'* **and** *intra-kind*(*kind a'*)
and *inner-node n* **and** ¬ *method-exit n* **and** ¬ *n postdominates* (*targetnode a'*)
and *last*:∀ *ax ax'*. *ax* ∈ *set as'* ∧ *sourcenode ax* = *sourcenode ax'* ∧
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*
(- $-\!\!\rightarrow_{dd}$ - [*51*,*0*,*0*] *80*)
and *call-edge* :: *'node SDG-node ⇒ 'pname ⇒ 'node SDG-node ⇒ bool*
(- $-\!\!\rightarrow_{call}$ - [*51*,*0*,*0*] *80*)
and *return-edge* :: *'node SDG-node ⇒ 'pname ⇒ 'node SDG-node ⇒ bool*
(- $-\!\!\rightarrow_{ret}$ - [*51*,*0*,*0*] *80*)
and *param-in-edge* :: *'node SDG-node ⇒ 'pname ⇒ 'var ⇒ 'node SDG-node ⇒ bool*
(- $-\!:\!\rightarrow_{in}$ - [*51*,*0*,*0*,*0*] *80*)
and *param-out-edge* :: *'node SDG-node ⇒ 'pname ⇒ 'var ⇒ 'node SDG-node ⇒ bool*
(- $-\!:\!\rightarrow_{out}$ - [*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*:
⟦*valid-edge a*; *kind a* = *Q*:*r*$\hookrightarrow_{p}$*fs*; (*p*,*ins*,*outs*) ∈ *set procs*; *V* = *ins*!*x*;
x < *length ins*; *n* = *Actual-in* (*sourcenode a*,*x*); *n'* = *Formal-in* (*targetnode*

a,x)⟧
⟹ n −p:V→$_{in}$ n'
| *SDG-param-out-edge*:
⟦*valid-edge* a; *kind* a = Q↩$_p f$; (p,*ins*,*outs*) ∈ *set procs*; V = *outs*!x;
x < *length outs*; n = *Formal-out* (*sourcenode* a,x);
n' = *Actual-out* (*targetnode* a,x)⟧
⟹ n −p:V→$_{out}$ n'

lemma *cdep-edge-cases*:
⟦n ⟶$_{cd}$ n'; (*parent-node* n) *controls* (*parent-node* n') ⟹ P;
⋀a Q r p *fs* a'. ⟦*valid-edge* a; *kind* a = Q:r↪$_p$*fs*; a' ∈ *get-return-edges* a;
parent-node n = *targetnode* a; *parent-node* n' = *sourcenode* a'⟧ ⟹
P;
⋀m. ⟦n = *CFG-node* m; m = *parent-node* n'; $n \neq n'$⟧ ⟹ P⟧ ⟹ 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) ∨ *CFG-node* (*parent-node* n) ⟶$_{cd}$ n
⟨*proof*⟩

lemma *SDG-cdep-edge-CFG-node*: n ⟶$_{cd}$ n' ⟹ ∃ m. n = *CFG-node* m
⟨*proof*⟩

lemma *SDG-call-edge-CFG-node*: n −p→$_{call}$ n' ⟹ ∃ m. n = *CFG-node* m
⟨*proof*⟩

lemma *SDG-return-edge-CFG-node*: n −p→$_{ret}$ n' ⟹ ∃ 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'
⟹ ∃!a. *valid-edge* a ∧ *sourcenode* a = *parent-node* n ∧
targetnode a = *parent-node* n' ∧ (∃ Q r *fs*. *kind* a = Q:r↪$_p$*fs*)
⟨*proof*⟩

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

SDG-edge n Vopt (Some(p,False)) n′
⟹ ∃!*a. valid-edge a* ∧ *sourcenode a = parent-node n* ∧
targetnode a = parent-node n′ ∧ (∃ *Q f. kind a = Q*↩$_{pf}$)
⟨*proof*⟩

lemma *Exit-no-SDG-edge-source*:
SDG-edge (*CFG-node* (*-Exit-*)) *Vopt popt n′* ⟹ *False*
⟨*proof*⟩

1.8.5 Intraprocedural paths in the SDG

inductive *intra-SDG-path* ::
′node SDG-node ⇒ *′node SDG-node list* ⇒ *′node SDG-node* ⇒ *bool*
(- *i*-→$_{d}$* - [*51*,*0*,*0*] *80*)

where *iSp-Nil*:
valid-SDG-node n ⟹ *n i*−[]→$_{d}$* *n*

| *iSp-Append-cdep*:
⟦*n i*−*ns*→$_{d}$* *n″*; *n″* ⟶$_{cd}$ *n′*⟧ ⟹ *n i*−*ns*@[*n″*]→$_{d}$* *n′*

| *iSp-Append-ddep*:
⟦*n i*−*ns*→$_{d}$* *n″*; *n″* −*V*→$_{dd}$ *n′*; *n″* ≠ *n′*⟧ ⟹ *n i*−*ns*@[*n″*]→$_{d}$* *n′*

lemma *intra-SDG-path-Append*:
⟦*n″ i*−*ns′*→$_{d}$* *n′*; *n i*−*ns*→$_{d}$* *n″*⟧ ⟹ *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*→$_{ι}$* *parent-node n′*
⟨*proof*⟩

1.8.6 Control dependence paths in the SDG

inductive *cdep-SDG-path* ::
′node SDG-node ⇒ *′node SDG-node list* ⇒ *′node SDG-node* ⇒ *bool*
(- *cd*-→$_{d}$* - [*51*,*0*,*0*] *80*)

where *cdSp-Nil*:
valid-SDG-node n ⟹ *n cd*−[]→$_{d}$* *n*

| *cdSp-Append-cdep*:
⟦*n cd−ns→$_d$* n''*; *n'' ⟶$_{cd}$ n'*⟧ ⟹ *n cd−ns@[n']→$_d$* n'*

lemma *cdep-SDG-path-intra-SDG-path*:
n cd−ns→$_d$ n'* ⟹ *n i−ns→$_d$* n'*
⟨*proof*⟩

lemma *Entry-cdep-SDG-path*:
assumes (*-Entry-*) *−as→$_\iota$* n'* **and** *inner-node n'* **and** ¬ *method-exit n'*
obtains *ns* **where** *CFG-node* (*-Entry-*) *cd−ns→$_d$* CFG-node n'*
and *ns* ≠ [] **and** ∀ *n'' ∈ set ns. parent-node n'' ∈ set*(*sourcenodes as*)
⟨*proof*⟩

lemma *in-proc-cdep-SDG-path*:
assumes *n −as→$_\iota$* n'* **and** *n* ≠ *n'* **and** *n'* ≠ (*-Exit-*) **and** *valid-edge a*
and *kind a* = *Q*:*r*↪$_p$*fs* **and** *targetnode a* = *n*
obtains *ns* **where** *CFG-node n cd−ns→$_d$* CFG-node n'*
and *ns* ≠ [] **and** ∀ *n'' ∈ set ns. parent-node n'' ∈ set*(*sourcenodes as*)
⟨*proof*⟩

1.8.7 Paths consisting of calls and control dependences

inductive *call-cdep-SDG-path* ::
'node SDG-node ⇒ *'node SDG-node list* ⇒ *'node SDG-node* ⇒ *bool*
(- *cc−-→$_d$** - [*51*,*0*,*0*] *80*)
where *ccSp-Nil*:
valid-SDG-node n ⟹ *n cc−[]→$_d$* n*

| *ccSp-Append-cdep*:
⟦*n cc−ns→$_d$* n''*; *n'' ⟶$_{cd}$ n'*⟧ ⟹ *n cc−ns@[n']→$_d$* n'*

| *ccSp-Append-call*:
⟦*n cc−ns→$_d$* n''*; *n'' −p→$_{call}$ n'*⟧ ⟹ *n cc−ns@[n']→$_d$* n'*

lemma *cc-SDG-path-Append*:
⟦*n'' cc−ns'→$_d$* n'*; *n cc−ns→$_d$* n''*⟧ ⟹ *n cc−ns@ns'→$_d$* n'*
⟨*proof*⟩

lemma *cdep-SDG-path-cc-SDG-path*:
n cd−ns→$_d$ n'* ⟹ *n cc−ns→$_d$* 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*$\longrightarrow_{cd}$ - [*51*,*0*] *80*)
and *sum-ddep-edge* :: *'node SDG-node* ⇒ *'var* ⇒ *'node SDG-node* ⇒ *bool*
(- *s*$-\!\!\rightarrow_{dd}$ - [*51*,*0*,*0*] *80*)
and *sum-call-edge* :: *'node SDG-node* ⇒ *'pname* ⇒ *'node SDG-node* ⇒ *bool*
(- *s*$-\!\!\rightarrow_{call}$ - [*51*,*0*,*0*] *80*)
and *sum-return-edge* :: *'node SDG-node* ⇒ *'pname* ⇒ *'node SDG-node* ⇒ *bool*
(- *s*$-\!\!\rightarrow_{ret}$ - [*51*,*0*,*0*] *80*)
and *sum-param-in-edge* :: *'node SDG-node* ⇒ *'pname* ⇒ *'var* ⇒ *'node SDG-node* ⇒ *bool*
(- *s*$-:\!\!\rightarrow_{in}$ - [*51*,*0*,*0*,*0*] *80*)
and *sum-param-out-edge* :: *'node SDG-node* ⇒ *'pname* ⇒ *'var* ⇒ *'node SDG-node* ⇒ *bool*
(- *s*$-:\!\!\rightarrow_{out}$ - [*51*,*0*,*0*,*0*] *80*)
and *sum-summary-edge* :: *'node SDG-node* ⇒ *'pname* ⇒ *'node SDG-node* ⇒ *bool*

(- s–-→$_{sum}$ - [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* = $Q{:}r{\hookrightarrow}_p fs$; n = *CFG-node (targetnode a)*;
$a' \in$ *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*; $n \neq n'$⟧
⟹ $n\ s{\longrightarrow}_{cd}\ n'$
| *sum-SDG-ddep-edge*:*n influences V in n'* ⟹ $n\ s{-}V{\rightarrow}_{dd}\ n'$
| *sum-SDG-call-edge*:
⟦*valid-edge a*; *kind a* = $Q{:}r{\hookrightarrow}_p fs$; 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* = $Q{\hookleftarrow}_p fs$; 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* = $Q{:}r{\hookrightarrow}_p fs$; $(p,ins,outs) \in$ *set procs*; V = *ins*!x;
$x <$ *length ins*; n = *Actual-in (sourcenode a*,x); n' = *Formal-in (targetnode a*,x)⟧
⟹ $n\ s{-}p{:}V{\rightarrow}_{in}\ n'$
| *sum-SDG-param-out-edge*:
⟦*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)⟧
⟹ $n\ s{-}p{:}V{\rightarrow}_{out}\ n'$
| *sum-SDG-call-summary-edge*:
⟦*valid-edge a*; *kind a* = $Q{:}r{\hookrightarrow}_p fs$; $a' \in$ *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* = $Q{:}r{\hookrightarrow}_p fs$; $a' \in$ *get-return-edges a*;
matched (Formal-in (targetnode a,x)) *ns (Formal-out (sourcenode a'*,x'));
n = *Actual-in (sourcenode a*,x); n' = *Actual-out (targetnode a'*,x');
$(p,ins,outs) \in$ *set procs*; $x <$ *length ins*; $x' <$ *length outs*⟧

$\Longrightarrow n\ s{-}p{\rightarrow}_{sum}\ n'$

lemma *sum-edge-cases*:
$[\![n\ s{-}p{\rightarrow}_{sum}\ n';$
$\bigwedge a\ Q\ r\ fs\ a'.\ [\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow}_p fs;\ a' \in get\text{-}return\text{-}edges\ a;$
$n = CFG\text{-}node\ (sourcenode\ a);\ n' = CFG\text{-}node\ (targetnode\ a')]\!] \Longrightarrow$
$P;$
$\bigwedge a\ Q\ p\ r\ fs\ a'\ ns\ x\ x'\ ins\ outs.$
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow}_p fs;\ a' \in get\text{-}return\text{-}edges\ a;$
$matched\ (Formal\text{-}in\ (targetnode\ a,x))\ ns\ (Formal\text{-}out\ (sourcenode\ a',x'));$
$n = Actual\text{-}in\ (sourcenode\ a,x);\ n' = Actual\text{-}out\ (targetnode\ a',x');$
$(p,ins,outs) \in set\ procs;\ x < length\ ins;\ x' < length\ outs]\!] \Longrightarrow P]\!]$
$\Longrightarrow P$
⟨*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*↩$_p$*f*
obtains *a'* *Q'* *r'* *fs'* **where** *a* ∈ *get-return-edges a'* **and** *valid-edge a'*
and *kind a'* = *Q'*:*r'*↪$_p$*fs'*
and *CFG-node* (*sourcenode a'*) *s*−*p*→$_{sum}$ *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*−-→$_d$* - [*51*,*0*,*0*] *80*)
where *isSp-Nil*:
valid-SDG-node n ⟹ *n is*−[]→$_d$* *n*

| *isSp-Append-cdep*:
⟦*n is*−*ns*→$_d$* *n''*; *n''* *s*⟶$_{cd}$ *n'*⟧ ⟹ *n is*−*ns*@[*n'*]→$_d$* *n'*

| *isSp-Append-ddep*:
⟦*n is*−*ns*→$_d$* *n''*; *n''* *s*−*V*→$_{dd}$ *n'*; *n''* ≠ *n'*⟧ ⟹ *n is*−*ns*@[*n'*]→$_d$* *n'*

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

lemma *is-SDG-path-Append*:
⟦*n'' is*−*ns'*→$_d$* *n'*; *n is*−*ns*→$_d$* *n''*⟧ ⟹ *n is*−*ns*@*ns'*→$_d$* *n'*
⟨*proof*⟩

lemma *is-SDG-path-valid-SDG-node*:
assumes *n is*−*ns*→$_d$* *n'* **shows** *valid-SDG-node n* **and** *valid-SDG-node n'*
⟨*proof*⟩

lemma *intra-SDG-path-is-SDG-path*:
n i−*ns*→$_d$* *n'* ⟹ *n is*−*ns*→$_d$* *n'*
⟨*proof*⟩

lemma *is-SDG-path-hd*:⟦*n is*−*ns*→$_d$* *n'*; *ns* ≠ []⟧ ⟹ *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 $n\ is{-}ns{\to_d}*\ n'$
and *refl*:$\bigwedge n.\ valid\text{-}SDG\text{-}node\ n \Longrightarrow P\ n\ []\ n$
and *step-cdep*:$\bigwedge n\ ns\ n'\ n''.\ [\![n\ s{\longrightarrow_{cd}}\ n''; n''\ is{-}ns{\to_d}*\ n'; P\ n''\ ns\ n']\!] \Longrightarrow P\ n\ (n\#ns)\ n'$
and *step-ddep*:$\bigwedge n\ ns\ n'\ V\ n''.\ [\![n\ s{-}V{\to_{dd}}\ n''; n \neq n''; n''\ is{-}ns{\to_d}*\ n'; P\ n''\ ns\ n']\!] \Longrightarrow P\ n\ (n\#ns)\ n'$
and *step-sum*:$\bigwedge n\ ns\ n'\ p\ n''.\ [\![n\ s{-}p{\to_{sum}}\ n''; n''\ is{-}ns{\to_d}*\ n'; P\ n''\ ns\ n']\!] \Longrightarrow P\ n\ (n\#ns)\ n'$
shows $P\ n\ ns\ n'$
⟨*proof*⟩

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

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

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

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

SDG paths without return edges

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

| *icsSp-Append-cdep*:
$[\![n\ ics{-}ns{\to_d}*\ n''; n''\ s{\longrightarrow_{cd}}\ n']\!] \Longrightarrow n\ ics{-}ns@[n']{\to_d}*\ n'$

| *icsSp-Append-ddep*:
$[\![n\ ics{-}ns{\to_d}*\ n''; n''\ s{-}V{\to_{dd}}\ n'; n'' \neq n']\!] \Longrightarrow n\ ics{-}ns@[n']{\to_d}*\ n'$

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

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

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

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

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

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

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

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

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

lemma *ics-SDG-path-realizable*:
assumes $n\ ics{-}ns{\to_d}*\ n'$
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'→$_d$* 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−-→$_d$** - [*51*,*0*,*0*] *80*)
where *irsSp-Nil*:
valid-SDG-node n ⟹ n irs−[]→$_d$ n*

| *irsSp-Cons-cdep*:
⟦*n'' irs−ns→$_d$* n'*; *n s⟶$_{cd}$ n''*⟧ ⟹ *n irs−n#ns→$_d$* n'*

| *irsSp-Cons-ddep*:
⟦*n'' irs−ns→$_d$* n'*; *n s−V→$_{dd}$ n''*; *n ≠ n''*⟧ ⟹ *n irs−n#ns→$_d$* n'*

| *irsSp-Cons-sum*:
⟦*n'' irs−ns→$_d$* n'*; *n s−p→$_{sum}$ n''*⟧ ⟹ *n irs−n#ns→$_d$* n'*

| *irsSp-Cons-return*:
⟦*n'' irs−ns→$_d$* n'*; *n s−p→$_{ret}$ n''*⟧ ⟹ *n irs−n#ns→$_d$* n'*

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

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

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

lemma *irs-SDG-path-split*:
assumes *n irs−ns→$_d$* n'*
obtains *n is−ns→$_d$* n'*
| *nsx nsx' nx nx' p* **where** *ns = nsx@nx#nsx'* **and** *n irs−nsx→$_d$* nx*
and *nx s−p→$_{ret}$ nx'* ∨ (∃ *V*. *nx s−p:V→$_{out}$ nx'*) **and** *nx' is−nsx'→$_d$* 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* $n \Longrightarrow n \in$ *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* ∈ *sum-SDG-slice1* n; n $s\longrightarrow_{cd}$ n'⟧ ⟹ *nx* ∈ *sum-SDG-slice1* n'
⟨*proof*⟩

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

lemma *slice1-sum-slice1*:
⟦*nx* ∈ *sum-SDG-slice1* n; n $s-p\rightarrow_{sum}$ n'⟧ ⟹ *nx* ∈ *sum-SDG-slice1* n'
⟨*proof*⟩

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

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

lemma *is-SDG-path-slice1*:
$[\![n\ is{-}ns{\rightarrow}_{d}{*}\ n';\ n' \in$ *sum-SDG-slice1* $n'']\!] \Longrightarrow n \in$ *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* ⟹ *n* ∈ *sum-SDG-slice2 n*
| *cdep-slice2*:
$[\![n''\ s{\longrightarrow}_{cd}\ n';\ n' \in$ *sum-SDG-slice2* $n]\!] \Longrightarrow n'' \in$ *sum-SDG-slice2 n*
| *ddep-slice2*:
$[\![n''\ s{-}V{\rightarrow}_{dd}\ n';\ n' \in$ *sum-SDG-slice2* $n]\!] \Longrightarrow n'' \in$ *sum-SDG-slice2 n*
| *return-slice2*:
$[\![n''\ s{-}p{\rightarrow}_{ret}\ n';\ n' \in$ *sum-SDG-slice2* $n]\!] \Longrightarrow n'' \in$ *sum-SDG-slice2 n*
| *param-out-slice2*:
$[\![n''\ s{-}p{:}V{\rightarrow}_{out}\ n';\ n' \in$ *sum-SDG-slice2* $n]\!] \Longrightarrow n'' \in$ *sum-SDG-slice2 n*
| *sum-slice2*:
$[\![n''\ s{-}p{\rightarrow}_{sum}\ n';\ n' \in$ *sum-SDG-slice2* $n]\!] \Longrightarrow n'' \in$ *sum-SDG-slice2 n*

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

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

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

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

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

⟨*proof*⟩

lemma *is-SDG-path-slice2*:
⟦*n is−ns→$_d$* n′*; *n′ ∈ sum-SDG-slice2 n″*⟧ ⟹ *n ∈ sum-SDG-slice2 n″*
⟨*proof*⟩

lemma *slice2-is-SDG-path-slice2*:
⟦*n is−ns→$_d$* n′*; *n″ ∈ sum-SDG-slice2 n*⟧ ⟹ *n″ ∈ sum-SDG-slice2 n′*
⟨*proof*⟩

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

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

inductive-set *combine-SDG-slices* :: *'node SDG-node set ⇒ 'node SDG-node set*
for *S*::*'node SDG-node set*
where *combSlice-refl*:*n ∈ S ⟹ n ∈ combine-SDG-slices S*
| *combSlice-Return-parent-node*:
⟦*n′ ∈ S*; *n″ s−p→$_{ret}$ CFG-node* (*parent-node n′*); *n ∈ sum-SDG-slice2 n″*⟧
⟹ *n ∈ combine-SDG-slices S*

definition *HRB-slice* :: *'node SDG-node set ⇒ 'node SDG-node set*
where *HRB-slice S* ≡ {*n′*. ∃ *n* ∈ *S*. *n′ ∈ combine-SDG-slices* (*sum-SDG-slice1 n*)}

lemma *HRB-slice-cases*[*consumes 1*,*case-names phase1 phase2*]:
⟦*x ∈ HRB-slice S*; ⋀*n nx*. ⟦*n ∈ sum-SDG-slice1 nx*; *nx ∈ S*⟧ ⟹ *P n*;
⋀*nx n′ n″ p n*. ⟦*n′ ∈ sum-SDG-slice1 nx*; *n″ s−p→$_{ret}$ CFG-node* (*parent-node n′*);
n ∈ sum-SDG-slice2 n′; *nx ∈ S*⟧ ⟹ *P n*⟧
⟹ *P x*
⟨*proof*⟩

lemma *HRB-slice-refl*:
assumes *valid-node m* **and** *CFG-node m ∈ S* **shows** *CFG-node m ∈ HRB-slice S*
⟨*proof*⟩

lemma *HRB-slice-valid-node*: *n ∈ HRB-slice S ⟹ valid-SDG-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'; $n'' \in$ *HRB-slice* $\{n\}$; $n' \in S$⟧ $\Longrightarrow$ $n'' \in$ *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⟧ $\Longrightarrow$ $n'' \in$ *sum-SDG-slice2* n'
⟨*proof*⟩

lemma *in-intra-SDG-path-in-HRB-slice*:
⟦n $i-ns\rightarrow_d*$ n'; $n'' \in$ *set* ns; $n' \in S$⟧ $\Longrightarrow$ $n'' \in$ *HRB-slice* S
⟨*proof*⟩

lemma *in-matched-in-slice2*:
⟦*matched* n ns n'; $n'' \in$ *set* ns⟧ $\Longrightarrow$ $n'' \in$ *sum-SDG-slice2* n'
⟨*proof*⟩

lemma *in-matched-in-HRB-slice*:
⟦*matched* n ns n'; $n'' \in$ *set* ns; $n' \in S$⟧ $\Longrightarrow$ $n'' \in$ *HRB-slice* S
⟨*proof*⟩

theorem *in-realizable-in-HRB-slice*:
⟦*realizable* n ns n'; $n'' \in$ *set* ns; $n' \in S$⟧ $\Longrightarrow$ $n'' \in$ *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'; $n \notin S$⟧ ⟹
$n \in$ *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 $\lfloor S \rfloor_{CFG} \equiv \{m.$ *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-*) $\in$ *obs-intra* $n' \lfloor$*HRB-slice* $S\rfloor_{CFG}$
shows *CFG-node* (*-Exit-*) $\in S$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *obs-the-element*:
$[\![ms \in obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG};\ \forall\, n \in set\ (tl\ ns).\ return\text{-}node\ n]\!]$
$\Longrightarrow (THE\ ms.\ ms \in obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}) = ms$
⟨*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 ∃!*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*
⟨*proof*⟩

lemma [*simp*]: *length* (*cspp m S fs*) = *length fs*
⟨*proof*⟩

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

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

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

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

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

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

1.12.2 Defining the sliced edge kinds

primrec *slice-kind-aux* :: *'node* ⇒ *'node* ⇒ *'node SDG-node set* ⇒
(*'var*,*'val*,*'ret*,*'pname*) *edge-kind* ⇒ (*'var*,*'val*,*'ret*,*'pname*) *edge-kind*
where *slice-kind-aux m m' S* ⇑*f* = (*if m* ∈ $\lfloor S \rfloor_{CFG}$ *then* ⇑*f else* ⇑*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*) ∧

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

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* ∈ ⌊*HRB-slice S*⌋$_{CFG}$; *intra-kind* (*kind a*)⟧
⟹ *slice-kind S a* = *kind a*
⟨*proof*⟩

lemma *slice-kind-Upd*:
⟦*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$; *kind a* = ⇑*f*⟧ ⟹ *slice-kind S a* = ⇑*id*
⟨*proof*⟩

lemma *slice-kind-Pred-empty-obs-nearer-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 $a' \wedge$ *intra-kind*(*kind* a') $\wedge$
targetnode $a' = n'$)
shows *slice-kind* $S\ a = (\lambda s.\ True)_{\surd}$
⟨*proof*⟩

lemma *slice-kind-Pred-empty-obs-nearer-not-SOME*:
assumes *sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$ **and** *kind* $a = (Q)_{\surd}$
and *obs-intra* (*sourcenode* a) $\lfloor$*HRB-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* $a \neq$ (*SOME* n'. $\exists a'$. *sourcenode* a = *sourcenode* $a' \wedge$
distance (*targetnode* a') *mex* $x \wedge$
valid-edge $a' \wedge$ *intra-kind*(*kind* a') $\wedge$
targetnode $a' = n'$)
shows *slice-kind* $S\ a = (\lambda s.\ False)_{\surd}$
⟨*proof*⟩

lemma *slice-kind-Pred-empty-obs-not-nearer*:
assumes *sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$ **and** *kind* $a = (Q)_{\surd}$
and *obs-intra* (*sourcenode* a) $\lfloor$*HRB-slice* $S\rfloor_{CFG} = \{\}$
and *method-exit mex* **and** *get-proc* (*sourcenode* a) = *get-proc mex*
and *dist*:*distance* (*sourcenode* a) *mex* $(x + 1)$ $\neg$ *distance* (*targetnode* a) *mex* x
shows *slice-kind* $S\ a = (\lambda s.\ False)_{\surd}$
⟨*proof*⟩

lemma *slice-kind-Pred-obs-nearer-SOME*:
assumes *sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$ **and** *kind* $a = (Q)_{\surd}$
and $m \in$ *obs-intra* (*sourcenode* a) $\lfloor$*HRB-slice* $S\rfloor_{CFG}$
and *distance* (*targetnode* a) m x *distance* (*sourcenode* a) m $(x + 1)$
and *targetnode* a = (*SOME* n'. $\exists a'$. *sourcenode* a = *sourcenode* $a' \wedge$
distance (*targetnode* a') m $x \wedge$
valid-edge $a' \wedge$ *intra-kind*(*kind* a') $\wedge$
targetnode $a' = n'$)
shows *slice-kind* $S\ a = (\lambda s.\ True)_{\surd}$
⟨*proof*⟩

lemma *slice-kind-Pred-obs-nearer-not-SOME*:
assumes *sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$ **and** *kind* $a = (Q)_{\surd}$
and $m \in$ *obs-intra* (*sourcenode* a) $\lfloor$*HRB-slice* $S\rfloor_{CFG}$
and *distance* (*targetnode* a) m x *distance* (*sourcenode* a) m $(x + 1)$
and *targetnode* $a \neq$ (*SOME* nx'. $\exists a'$. *sourcenode* a = *sourcenode* $a' \wedge$
distance (*targetnode* a') m $x \wedge$
valid-edge $a' \wedge$ *intra-kind*(*kind* a') $\wedge$
targetnode $a' = nx'$)
shows *slice-kind* $S\ a = (\lambda s.\ False)_{\surd}$

⟨*proof*⟩

lemma *slice-kind-Pred-obs-not-nearer*:
assumes *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *kind a* = (*Q*)$_{\surd}$
and *in-obs*:*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$
and *dist*:*distance* (*sourcenode a*) *m* (*x* + *1*)
¬ *distance* (*targetnode a*) *m x*
shows *slice-kind S a* = (λ*s*. *False*)$_{\surd}$
⟨*proof*⟩

lemma *kind-Predicate-notin-slice-slice-kind-Predicate*:
assumes *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *valid-edge a* **and** *kind a* = (*Q*)$_{\surd}$
obtains *Q*′ **where** *slice-kind S a* = (*Q*′)$_{\surd}$ **and** *Q*′ = (λ*s*. *False*) ∨ *Q*′ = (λ*s*. *True*)
⟨*proof*⟩

lemma *slice-kind-Call*:
⟦*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$; *kind a* = *Q*:*r*↪$_p$*fs*⟧
⟹ *slice-kind S a* = (λ*cf*. *False*):*r*↪$_p$*fs*
⟨*proof*⟩

lemma *slice-kind-Call-in-slice*:
⟦*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$; *kind a* = *Q*:*r*↪$_p$*fs*⟧
⟹ *slice-kind S a* = *Q*:*r*↪$_p$(*cspp* (*targetnode a*) (*HRB-slice S*) *fs*)
⟨*proof*⟩

lemma *slice-kind-Call-in-slice-Formal-in-not*:
assumes *sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *kind a* = *Q*:*r*↪$_p$*fs*
and ∀ *x* < *length fs*. *Formal-in*(*targetnode a*,*x*) ∉ *HRB-slice S*
shows *slice-kind S a* = *Q*:*r*↪$_p$*replicate* (*length fs*) *Map.empty*
⟨*proof*⟩

lemma *slice-kind-Call-in-slice-Formal-in-also*:
assumes *sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *kind a* = *Q*:*r*↪$_p$*fs*
and ∀ *x* < *length fs*. *Formal-in*(*targetnode a*,*x*) ∈ *HRB-slice S*
shows *slice-kind S a* = *Q*:*r*↪$_p$*fs*
⟨*proof*⟩

lemma *slice-kind-Call-intra-notin-slice*:
assumes *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *valid-edge a*
and *intra-kind* (*kind a*) **and** *valid-edge a*′ **and** *kind a*′ = *Q*:*r*↪$_p$*fs*

and *sourcenode* $a' = sourcenode\ a$
shows *slice-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-edge $a' \wedge$ *intra-kind*$(kind\ a') \wedge$
$targetnode\ a' = (SOME\ n'.\ \exists\, a'.\ sourcenode\ a = sourcenode\ a' \wedge$
$distance\ (targetnode\ a')\ mex\ x \wedge$
valid-edge $a' \wedge$ *intra-kind*$(kind\ a') \wedge$
$targetnode\ a' = n')$
⟨*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'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
shows *n* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
⟨*proof*⟩

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

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

1.13.1 Silent moves

inductive *silent-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*
(-,- ⊢ '(-,-') −−⟶$_{\tau}$ '(-,-') [*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* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*); *msx* ∈ *obs ms'* ⌊*HRB-slice S*⌋$_{CFG}$;
∀ *n* ∈ *set* (*tl ms'*). *return-node n*⟧
⟹ *msx* ∈ *obs ms* ⌊*HRB-slice S*⌋$_{CFG}$

⟨*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* ::
$'node\ SDG\text{-}node\ set \Rightarrow ('edge \Rightarrow ('var,'val,'ret,'pname)\ edge\text{-}kind) \Rightarrow 'node\ list \Rightarrow$
$(('var \rightharpoonup 'val) \times 'ret)\ list \Rightarrow 'edge\ list \Rightarrow 'node\ list \Rightarrow (('var \rightharpoonup 'val) \times 'ret)\ list \Rightarrow bool$
$(\text{-},\text{-} \vdash '(\text{-},\text{-}') =\text{-}\Rightarrow_\tau '(\text{-},\text{-}')\ [51,50,0,0,50,0,0]\ 51)$

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

obtains *as* **where** S,*slice-kind* $S \vdash (m\#msx',s) =as\Rightarrow_\tau (m'\#msx',s)$
⟨*proof*⟩

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

1.13.2 Observable moves

inductive *observable-move* ::
$'node\ SDG\text{-}node\ set \Rightarrow ('edge \Rightarrow ('var,'val,'ret,'pname)\ edge\text{-}kind) \Rightarrow 'node\ list \Rightarrow$
$(('var \rightharpoonup 'val) \times 'ret)\ list \Rightarrow 'edge \Rightarrow 'node\ list \Rightarrow (('var \rightharpoonup 'val) \times 'ret)\ list \Rightarrow bool$
(-,- ⊢ '(-,-') −-→ '(-,-') [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);
$\forall\, m \in set\ (tl\ ms).\ \exists\, m'.\ call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge m' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$;
$hd\ ms \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$; $length\ s' = length\ s$; $length\ ms = length\ s$;
$hd\ ms = sourcenode\ a$; $ms' = (targetnode\ a)\#tl\ ms$⟧
⟹ S,$f \vdash (ms,s)\ -a\rightarrow\ (ms',s')$

| *observable-move-call*:
⟦*pred* $(f\ a)\ s$; *transfer* $(f\ a)\ s = s'$; *valid-edge* a; *kind* $a = Q{:}r\hookrightarrow_p fs$;
valid-edge a'; $a' \in get\text{-}return\text{-}edges\ a$;
$\forall\, m \in set\ (tl\ ms).\ \exists\, m'.\ call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge m' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$;
$hd\ ms \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$; $length\ ms = length\ s$; $length\ s' = Suc(length\ s)$;
$hd\ ms = sourcenode\ a$; $ms' = (targetnode\ a)\#(targetnode\ a')\#tl\ ms$⟧
⟹ S,$f \vdash (ms,s)\ -a\rightarrow\ (ms',s')$

| *observable-move-return*:
⟦*pred* $(f\ a)\ s$; *transfer* $(f\ a)\ s = s'$; *valid-edge* a; *kind* $a = Q{\hookleftarrow_p}f'$;
$\forall\, m \in set\ (tl\ ms).\ \exists\, m'.\ call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge m' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$;
$length\ ms = length\ s$; $length\ s = Suc(length\ s')$; $s' \neq []$;
$hd\ ms = sourcenode\ a$; $hd(tl\ ms) = targetnode\ a$; $ms' = tl\ ms$⟧
⟹ S,$f \vdash (ms,s)\ -a\rightarrow\ (ms',s')$

inductive *observable-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 ⇒ *bool*
(-,- ⊢ '(-,-') =-⇒ '(-,-') [51,50,0,0,50,0,0] 51)

where *observable-moves-snoc*:
⟦*S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms'*,*s'*); *S*,*f* ⊢ (*ms'*,*s'*) −*a*→ (*ms''*,*s''*)⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) =*as*@[*a*]⇒ (*ms''*,*s''*)

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

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

lemma *observable-move-notempty*:
⟦*S*,*f* ⊢ (*ms*,*s*) =*as*⇒ (*ms'*,*s'*); *as* = []⟧ ⟹ *False*
⟨*proof*⟩

lemma *silent-move-observable-moves*:
⟦*S*,*f* ⊢ (*ms''*,*s''*) =*as*⇒ (*ms'*,*s'*); *S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms''*,*s''*)⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) =*a*#*as*⇒ (*ms'*,*s'*)
⟨*proof*⟩

lemma *silent-append-observable-moves*:
⟦*S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms''*,*s''*); *S*,*f* ⊢ (*ms''*,*s''*) =*as'*⇒ (*ms'*,*s'*)⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) =*as*@*as'*⇒ (*ms'*,*s'*)
⟨*proof*⟩

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

lemma *observable-move-vpa-path*:
⟦*S*,*f* ⊢ (*m*#*ms*,*s*) −*a*→ (*m'*#*ms'*,*s'*); *valid-node m*;
∀ *i* < *length rs*. *rs*!*i* ∈ *get-return-edges* (*cs*!*i*); *ms* = *targetnodes rs*;
valid-return-list rs m; *length rs* = *length cs*⟧ ⟹ *valid-path-aux cs* [*a*]
⟨*proof*⟩

1.13.3 Relevant variables

inductive-set *relevant-vars* ::
  'node SDG-node set ⇒ *'node SDG-node* ⇒ *'var set* (*rv* -)
for *S* :: *'node SDG-node set* **and** *n* :: *'node SDG-node*

where *rvI*:
  ⟦*parent-node n* −*as*→$_\iota$∗ *parent-node n'*; *n'* ∈ *HRB-slice S*; *V* ∈ *Use*$_{SDG}$ *n'*;
    ∀ *n''*. *valid-SDG-node n''* ∧ *parent-node n''* ∈ *set* (*sourcenodes as*)
        ⟶ *V* ∉ *Def*$_{SDG}$ *n''*⟧
  ⟹ *V* ∈ *rv S n*

lemma *rvE*:
  assumes *rv*:*V* ∈ *rv S n*
  obtains *as n'* **where** *parent-node n* −*as*→$_\iota$∗ *parent-node n'*
  and *n'* ∈ *HRB-slice S* **and** *V* ∈ *Use*$_{SDG}$ *n'*
  and ∀ *n''*. *valid-SDG-node n''* ∧ *parent-node n''* ∈ *set* (*sourcenodes as*)
    ⟶ *V* ∉ *Def*$_{SDG}$ *n''*
⟨*proof*⟩

lemma *rv-parent-node*:
  parent-node n = *parent-node n'* ⟹ *rv* (*S*::*'node SDG-node set*) *n* = *rv S n'*
⟨*proof*⟩

lemma *obs-intra-empty-rv-empty*:
  assumes *obs-intra m* ⌊*HRB-slice S*⌋$_{CFG}$ = {} **shows** *rv S* (*CFG-node m*) = {}
⟨*proof*⟩

lemma *eq-obs-intra-in-rv*:
  assumes *obs-eq*:*obs-intra* (*parent-node n*) ⌊*HRB-slice S*⌋$_{CFG}$ =
        obs-intra (*parent-node n'*) ⌊*HRB-slice S*⌋$_{CFG}$
  and *x* ∈ *rv S n* **shows** *x* ∈ *rv S n'*
⟨*proof*⟩

lemma *closed-eq-obs-eq-rvs*:
  fixes *S* :: *'node SDG-node set*
  assumes *obs-eq*:*obs-intra* (*parent-node n*) ⌊*HRB-slice S*⌋$_{CFG}$ =
  obs-intra (*parent-node n'*) ⌊*HRB-slice S*⌋$_{CFG}$
  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* ≠ *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* ⊢ (*ms*$_1$,*s*$_1$) −*a*→$_\tau$ (*ms*$_1$',*s*$_1$') **and** ((*ms*$_1$,*s*$_1$),(*ms*$_2$,*s*$_2$)) ∈ *WS S*
shows ((*ms*$_1$',*s*$_1$'),(*ms*$_2$,*s*$_2$)) ∈ *WS S*

⟨*proof*⟩

lemma *WS-silent-moves*:
⟦*S*,*kind* ⊢ (*ms*$_1$,*s*$_1$) =*as*⇒$_\tau$ (*ms*$_1$′,*s*$_1$′); ((*ms*$_1$,*s*$_1$),(*ms*$_2$,*s*$_2$)) ∈ *WS S*⟧
⟹ ((*ms*$_1$′,*s*$_1$′),(*ms*$_2$,*s*$_2$)) ∈ *WS S*
⟨*proof*⟩

lemma *WS-observable-move*:
assumes ((*ms*$_1$,*s*$_1$),(*ms*$_2$,*s*$_2$)) ∈ *WS S*
and *S*,*kind* ⊢ (*ms*$_1$,*s*$_1$) −*a*→ (*ms*$_1$′,*s*$_1$′) **and** *s*$_1$′ ≠ []
obtains *as* **where** ((*ms*$_1$′,*s*$_1$′),(*ms*$_1$′,*transfer* (*slice-kind S a*) *s*$_2$)) ∈ *WS S*
and *S*,*slice-kind S* ⊢ (*ms*$_2$,*s*$_2$) =*as*@[*a*]⇒ (*ms*$_1$′,*transfer* (*slice-kind S a*) *s*$_2$)
⟨*proof*⟩

1.13.5 The weak simulation

definition *is-weak-sim* ::
(('*node list* × (('*var* ⇀ '*val*) × '*ret*) *list*) ×
('*node list* × (('*var* ⇀ '*val*) × '*ret*) *list*)) *set* ⇒ '*node SDG-node set* ⇒ *bool*
where *is-weak-sim R S* ≡
∀ *ms*$_1$ *s*$_1$ *ms*$_2$ *s*$_2$ *ms*$_1$′ *s*$_1$′ *as*.
((*ms*$_1$,*s*$_1$),(*ms*$_2$,*s*$_2$)) ∈ *R* ∧ *S*,*kind* ⊢ (*ms*$_1$,*s*$_1$) =*as*⇒ (*ms*$_1$′,*s*$_1$′) ∧ *s*$_1$′ ≠ []
⟶ (∃ *ms*$_2$′ *s*$_2$′ *as*′. ((*ms*$_1$′,*s*$_1$′),(*ms*$_2$′,*s*$_2$′)) ∈ *R* ∧
S,*slice-kind S* ⊢ (*ms*$_2$,*s*$_2$) =*as*′⇒ (*ms*$_2$′,*s*$_2$′))

lemma *WS-weak-sim*:
assumes ((*ms*$_1$,*s*$_1$),(*ms*$_2$,*s*$_2$)) ∈ *WS S*
and *S*,*kind* ⊢ (*ms*$_1$,*s*$_1$) =*as*⇒ (*ms*$_1$′,*s*$_1$′) **and** *s*$_1$′ ≠ []
obtains *as*′ **where** ((*ms*$_1$′,*s*$_1$′),(*ms*$_1$′,*transfer* (*slice-kind S* (*last as*)) *s*$_2$)) ∈ *WS S*
and *S*,*slice-kind S* ⊢ (*ms*$_2$,*s*$_2$) =*as*′@[*last as*]⇒
(*ms*$_1$′,*transfer* (*slice-kind S* (*last as*)) *s*$_2$)
⟨*proof*⟩

The following lemma states the correctness of static intraprocedural slicing: the simulation *WS S* is a desired weak simulation

theorem *WS-is-weak-sim*:*is-weak-sim* (*WS S*) *S*
⟨*proof*⟩

end

end

1.14 The fundamental property of slicing

theory *FundamentalProperty* **imports** *WeakSimulation SemanticsCFG* **begin**

context *SDG* **begin**

1.14.1 Auxiliary lemmas for moves in the graph

lemma *observable-set-stack-in-slice*:
S,$f \vdash (ms,s) -a\rightarrow (ms',s')$
$\Longrightarrow \forall\, mx \in set\ (tl\ ms').\ \exists\, mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
⟨*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-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-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-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-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-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-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 $\forall\, i < \mathit{length}\ rs.\ rs!i \in \mathit{get\text{-}return\text{-}edges}\ (cs!i)$ **and** $\mathit{valid\text{-}return\text{-}list}\ rs\ m$
and $\mathit{length}\ rs = \mathit{length}\ cs$ **and** $ms = \mathit{targetnodes}\ rs$
obtains $cs'\ rs'$ **where** $\mathit{valid\text{-}node}\ m'$ **and** $\mathit{valid\text{-}call\text{-}list}\ cs'\ m'$
and $\forall\, i < \mathit{length}\ rs'.\ rs'!i \in \mathit{get\text{-}return\text{-}edges}\ (cs'!i)$
and $\mathit{valid\text{-}return\text{-}list}\ rs'\ m'$ **and** $\mathit{length}\ rs' = \mathit{length}\ cs'$
and $ms' = \mathit{targetnodes}\ rs'$ **and** $\mathit{upd\text{-}cs}\ cs\ as = cs'$
⟨*proof*⟩

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

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

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

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↪pfs*
and *call-of-return-node (hd ms') (sourcenode a)*
and *targetnode a −as''→sl* m'*
| *ms' = ms*
⟨*proof*⟩

lemma *silent-moves-called-node-in-slice1-hd-nodestack-in-slice1*:
assumes *S,kind ⊢ (m#ms,s) =as⇒τ (m'#ms',s')* **and** *valid-node m*
and *CFG-node m' ∈ sum-SDG-slice1 nx*
and $\forall$ *mx ∈ set ms.* $\exists$ *mx'. call-of-return-node mx mx' ∧*
mx' ∈ ⌊HRB-slice S⌋CFG
and $\forall$ *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*
obtains *as' a as''* **where** *as = as'@a#as''* **and** $\exists$ *Q r p fs. kind a = Q:r↪pfs*
and *call-of-return-node (hd ms') (sourcenode a)*
and *targetnode a −as''→sl* m'* **and** *CFG-node (sourcenode a) ∈ sum-SDG-slice1 nx*
| *ms' = ms*
⟨*proof*⟩

lemma *silent-moves-called-node-in-slice1-nodestack-in-slice1*:
⟦*S,kind ⊢ (m#ms,s) =as⇒τ (m'#ms',s')*; *valid-node m*;
CFG-node m' ∈ sum-SDG-slice1 nx; *nx ∈ S*;
$\forall$ *mx ∈ set ms.* $\exists$ *mx'. call-of-return-node mx mx' ∧ mx' ∈ ⌊HRB-slice S⌋CFG*;
$\forall$ *i < length rs. rs!i ∈ get-return-edges (cs!i)*; *ms = targetnodes rs*;
valid-return-list rs m; *length rs = length cs*⟧
⟹ $\forall$ *mx ∈ set ms'.* $\exists$ *mx'. call-of-return-node mx mx' ∧ mx' ∈ ⌊HRB-slice S⌋CFG*
⟨*proof*⟩

lemma *silent-moves-slice-intra-path*:
assumes *S,slice-kind S ⊢ (m#ms,s) =as⇒τ (m'#ms',s')*
and $\forall$ *mx ∈ set ms.* $\exists$ *mx'. call-of-return-node mx mx' ∧ mx' ∈ ⌊HRB-slice S⌋CFG*
shows $\forall$ *a ∈ set as. intra-kind (kind a)*
⟨*proof*⟩

lemma *silent-moves-slice-keeps-state*:
assumes *S,slice-kind S ⊢ (m#ms,s) =as⇒τ (m'#ms',s')*
and $\forall$ *mx ∈ set ms.* $\exists$ *mx'. call-of-return-node mx mx' ∧ mx' ∈ ⌊HRB-slice S⌋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* $\equiv$ ($\forall$ *c* $\in$ *set cs. sourcenode c* $\in$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$) $\wedge$
(*case* (*kind a*) *of Q*$\hookleftarrow_{pf}$ $\Rightarrow$ *True* | - $\Rightarrow$ *sourcenode a* $\in$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$)

lemma *silent-move-no-slice-edge*:
⟦*S*,*f* $\vdash$ (*ms*,*s*) $-a\rightarrow_\tau$ (*ms*′,*s*′); *tl ms* = *targetnodes rs*; *length rs* = *length cs*;
$\forall$ *i*<*length cs. call-of-return-node* (*tl ms*!*i*) (*sourcenode* (*cs*!*i*))⟧
$\Longrightarrow$ $\neg$ *slice-edge S cs a*
⟨*proof*⟩

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

function *slice-edges* :: *'node SDG-node set* $\Rightarrow$ *'edge list* $\Rightarrow$ *'edge list* $\Rightarrow$ *'edge list*
where *slice-edges S cs* [] = []
| *slice-edge S cs a* $\Longrightarrow$
slice-edges S cs (*a*#*as*) = *a*#*slice-edges S* (*upd-cs cs* [*a*]) *as*
| $\neg$ *slice-edge S cs a* $\Longrightarrow$
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*′⟧
$\Longrightarrow$ *slice-edges S cs* (*as*@*asx*) = *as*′@*asx*′
⟨*proof*⟩

lemma *slice-edges-Nil-split*:
slice-edges S cs (*as*@*as*′) = []
$\Longrightarrow$ *slice-edges S cs as* = [] $\wedge$ *slice-edges S* (*upd-cs cs as*) *as*′ = []
⟨*proof*⟩

lemma *slice-intra-edges-no-nodes-in-slice*:
⟦*slice-edges S cs as* = []; $\forall$ *a* $\in$ *set as. intra-kind* (*kind a*);
$\forall$ *c* $\in$ *set cs. sourcenode c* $\in$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$⟧
$\Longrightarrow$ $\forall$ *nx* $\in$ *set*(*sourcenodes as*). *nx* $\notin$ $\lfloor$*HRB-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-node m*
and $ms' \neq []$ **and** *CFG-node* $m' \in$ *sum-SDG-slice1 nx* **and** $nx \in S$
shows *False*
⟨*proof*⟩

lemma *transfers-intra-slice-kinds-slice-edges*:
$[\![\forall a \in set\ as.\ intra\text{-}kind\ (kind\ a); \forall c \in set\ cs.\ sourcenode\ c \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}]\!]$
$\Longrightarrow transfers\ (slice\text{-}kinds\ S\ (slice\text{-}edges\ S\ cs\ as))\ s =$
$transfers\ (slice\text{-}kinds\ S\ as)\ s$
⟨*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 ⇒ (*'edge* ⇒ (*'var*,*'val*,*'ret*,*'pname*) *edge-kind*) ⇒ *'node list* ⇒
((*'var* ⇀ *'val*) × *'ret*) *list* ⇒ *'edge list* ⇒ *'node list* ⇒
((*'var* ⇀ *'val*) × *'ret*) *list* ⇒ *bool*
(-,- ⊢ *'*(-,-*'*) =-⇒∗ *'*(-,-*'*) [*51*,*50*,*0*,*0*,*50*,*0*,*0*,*0] *51*)

where *tom-Nil*:
length ms = *length s* ⟹ *S*,*f* ⊢ (*ms*,*s*) =[]⇒∗ (*ms*,*s*)

| *tom-Cons*:
⟦*S*,*f* ⊢ (*ms*,*s*) =*as*⇒ (*ms'*,*s'*); *S*,*f* ⊢ (*ms'*,*s'*) =*as'*⇒∗ (*ms''*,*s''*)⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) =(*last as*)#*as'*⇒∗ (*ms''*,*s''*)

lemma *tom-split-snoc*:
assumes *S*,*f* ⊢ (*ms*,*s*) =*as*⇒∗ (*ms'*,*s'*) **and** *as* ≠ []
obtains *asx asx' ms'' s''* **where** *as* = *asx*@[*last asx'*]
and *S*,*f* ⊢ (*ms*,*s*) =*asx*⇒∗ (*ms''*,*s''*) **and** *S*,*f* ⊢ (*ms''*,*s''*) =*asx'*⇒ (*ms'*,*s'*)
⟨*proof*⟩

lemma *tom-preserves-stacks*:
assumes *S*,*f* ⊢ (*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*
obtains *cs' rs'* **where** *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'*
⟨*proof*⟩

lemma *vpa-trans-observable-moves*:
assumes *valid-path-aux cs as* **and** *m* −*as*→∗ *m'* **and** *preds* (*kinds as*) *s*
and *transfers* (*kinds as*) *s* = *s'* **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** *length s* = *Suc* (*length cs*)
obtains *ms ms'' s'' ms' as' as''*
where *S*,*kind* ⊢ (*m*#*ms*,*s*) =*slice-edges S cs as*⇒∗ (*ms''*,*s''*)
and *S*,*kind* ⊢ (*ms''*,*s''*) =*as'*⇒$_\tau$ (*m'*#*ms'*,*s'*)
and *ms* = *targetnodes rs* **and** *length ms* = *length cs*
and ∀ *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'*→∗ *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* $\vdash$ ([*m*],[*cf*]) =*slice-edges S* [] *as*$\Rightarrow$∗ (ms'',s'')
and S,*kind* $\vdash$ (ms'',s'') =$as'\Rightarrow_{\tau}$ (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)) ∈ *WS S*
and S,*kind* $\vdash$ (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)) ∈ *WS S* ∧
S,*slice-kind S* $\vdash$ (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* $\vdash$ (*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'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$
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'* ∈ *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 $m \triangleq c$ **and** ⟨*c*,[*cf*]⟩ ⇒ ⟨*c'*,*s'*⟩
obtains m' *as cfs'* **where** $m\ -as\rightarrow_{\surd}*\ m'$ **and** $m' \triangleq c'$
and *preds* (*slice-kinds* {*CFG-node m'*} *as*) [(*cf*,*undefined*)]
and $\forall\, V \in$ *Use* m'.
state-val (*transfers* (*slice-kinds* {*CFG-node m'*} *as*) [(*cf*,*undefined*)]) V =
state-val cfs' V **and** *map fst cfs'* = s'
⟨*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 ⊢ *Entry* −*IEdge* $(\lambda s.\ False)_{\surd}\!\!\rightarrow_p$ *Exit*

| *Proc-CFG-Entry*:
prog ⊢ *Entry* −*IEdge* $(\lambda s.\ True)_{\surd}\!\!\rightarrow_p$ *Label 0*

| *Proc-CFG-Skip*:
Skip ⊢ *Label 0* −*IEdge* ⇑*id*→$_p$ *Exit*

| *Proc-CFG-LAss*:
V:=*e* ⊢ *Label 0* −*IEdge* ⇑(λ*cf*. *update cf V e*)→$_p$ *Label 1*

| *Proc-CFG-LAssSkip*:
$V{:=}e \vdash \mathit{Label}\ 1\ {-}\mathit{IEdge}\ {\Uparrow}id{\rightarrow_p}\ \mathit{Exit}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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* × *vname list* × *vname list* × *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 ⇒ *cmd* ⇒ *pname list* ⇒ *pname* ⇒ *bool*
where *containsCall procs Skip ps p* ⟷ *False*
  | *containsCall procs* (*V*:=*e*) *ps p* ⟷ *False*
  | *containsCall procs* (*c*$_1$;;*c*$_2$) *ps p* ⟷
      containsCall procs c$_1$ *ps p* ∨ *containsCall procs c*$_2$ *ps p*
  | *containsCall procs* (*if* (*b*) *c*$_1$ *else c*$_2$) *ps p* ⟷
      containsCall procs c$_1$ *ps p* ∨ *containsCall procs c*$_2$ *ps p*
  | *containsCall procs* (*while* (*b*) *c*) *ps p* ⟷
      containsCall procs c ps p
  | *containsCall procs* (*Call q es' rets'*) *ps p* ⟷ *p* = *q* ∧ *ps* = [] ∨
      (∃ *ins outs c ps'*. *ps* = *q*#*ps'* ∧ (*q*,*ins*,*outs*,*c*) ∈ *set procs* ∧
                  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
  ⟹ *ps* = [] ∧ *containsCall procs prog ps p* ∨
  (∃ *q ins outs c ps'*. *ps* = *ps'*@[*q*] ∧ (*q*,*ins*,*outs*,*c*) ∈ *set procs* ∧
  containsCall procs c [] *p* ∧ *containsCall procs prog ps' q*)
⟨*proof*⟩

lemma *containsCallE*:
  ⟦*containsCall procs prog ps p*;
    ⟦*ps* = []; *containsCall procs prog ps p*⟧ ⟹ *P procs prog ps p*;
    ⋀*q ins outs c es' rets' ps'*. ⟦*ps* = *ps'*@[*q*]; (*q*,*ins*,*outs*,*c*) ∈ *set procs*;
      containsCall procs c [] *p*; *containsCall procs prog ps' q*⟧
    ⟹ *P procs prog ps p*⟧ ⟹ *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*)→$_p$ *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*)→$_p$ *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*→$_p$ *n'* ⟹ *prog*,*procs* ⊢ (*Main*,*n*) −*et*→ (*Main*,*n'*)

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

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

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

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

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

| *ProcCallReturn*:
⟦(*p*,*ins*,*outs*,*c*) ∈ *set procs*; *c* ⊢ *n* −*CEdge* (*p'*,*es'*,*rets'*)→$_p$ *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*) ∈ *set procs*⟧ ⟹ *False*
⟨*proof*⟩

lemma *well-formed-same-procs* [*dest*]:
⟦*well-formed procs*; (*p*,*ins*,*outs*,*c*) ∈ *set procs*; (*p*,*ins*′,*outs*′,*c*′) ∈ *set procs*⟧
⟹ *ins* = *ins*′ ∧ *outs* = *outs*′ ∧ *c* = *c*′
⟨*proof*⟩

lemma *PCFG-sourcelabel-None-less-num-nodes*:
⟦*prog*,*procs* ⊢ (*Main*,*Label l*) −*et*→ *n*′; *well-formed procs*⟧ ⟹ *l* < #:*prog*
⟨*proof*⟩

lemma *Proc-CFG-sourcelabel-Some-less-num-nodes*:
⟦*prog*,*procs* ⊢ (*p*,*Label l*) −*et*→ *n*′; (*p*,*ins*,*outs*,*c*) ∈ *set procs*;
well-formed procs⟧ ⟹ *l* < #:*c*
⟨*proof*⟩

lemma *Proc-CFG-targetlabel-Main-less-num-nodes*:
⟦*prog*,*procs* ⊢ *n* −*et*→ (*Main*,*Label l*); *well-formed procs*⟧ ⟹ *l* < #:*prog*
⟨*proof*⟩

lemma *Proc-CFG-targetlabel-Some-less-num-nodes*:
⟦*prog*,*procs* ⊢ *n* −*et*→ (*p*,*Label l*); (*p*,*ins*,*outs*,*c*) ∈ *set procs*;
well-formed procs⟧ ⟹ *l* < #:*c*
⟨*proof*⟩

lemma *Proc-CFG-edge-det*:
⟦*prog*,*procs* ⊢ *n* −*et*→ *n*′; *prog*,*procs* ⊢ *n* −*et*′→ *n*′; *well-formed procs*⟧
⟹ *et* = *et*′
⟨*proof*⟩

lemma *Proc-CFG-deterministic*:
⟦*prog*,*procs* ⊢ n_1 −et_1→ n_1′; *prog*,*procs* ⊢ n_2 −et_2→ n_2′; n_1 = n_2; n_1′ ≠ n_2′;
intra-kind et_1; *intra-kind* et_2; *well-formed procs*⟧
⟹ ∃ *Q* *Q*′. et_1 = $(Q)_\surd$ ∧ et_2 = $(Q')_\surd$ ∧
(∀ *s*. (*Q s* ⟶ ¬ *Q*′ *s*) ∧ (*Q*′ *s* ⟶ ¬ *Q s*))
⟨*proof*⟩

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

definition *wf* :: *cmd* ⇒ *procs* ⇒ *bool*
where *wf prog procs* ≡ *well-formed procs* ∧

$(\forall\, ps\ p.\ containsCall\ procs\ prog\ ps\ p \longrightarrow (\exists\, ins\ outs\ c.\ (p,ins,outs,c) \in set\ procs$
$\wedge$
$(\forall\, c'\ n\ n'\ es\ rets.\ c' \vdash n\ -CEdge\ (p,es,rets)\rightarrow_p n' \longrightarrow$
$distinct\ rets \wedge length\ rets = length\ outs \wedge length\ es = length\ ins)))$

lemma *wf-well-formed* [*intro*]:*wf prog procs* $\Longrightarrow$ *well-formed procs*
⟨*proof*⟩

lemma *wf-distinct-rets* [*intro*]:
⟦*wf prog procs*; *containsCall procs prog ps p*; $(p,ins,outs,c) \in set\ procs$;
$c' \vdash n\ -CEdge\ (p,es,rets)\rightarrow_p n'$⟧ $\Longrightarrow$ *distinct rets*
⟨*proof*⟩

lemma
assumes *wf prog procs* **and** *containsCall procs prog ps p*
and $(p,ins,outs,c) \in set\ procs$ **and** $c' \vdash n\ -CEdge\ (p,es,rets)\rightarrow_p n'$
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*:*Rep-wf-prog wfp* = (*prog*,*procs*) $\Longrightarrow$ *wf prog procs*
⟨*proof*⟩

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

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

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

lemma *wfp-Call*: **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 *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 *get-return-edges wfp a* ≡
case kind a of Q:*r*↪$_p$*fs* ⇒ {*a'*. *valid-edge wfp a'* ∧ (∃ *Q' f'*. *kind a'* = *Q'*↩$_p$*f'*)
∧
targetnode a' = *r*}
| - ⇒ {}

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

lemma *call-has-return-edge*:

assumes *valid-edge wfp a* **and** *kind a* = *Q*:*r*↪*pfs*
obtains *a'* **where** *valid-edge wfp a'* **and** ∃ *Q' f'*. *kind a'* = *Q'*↩*pf'*
and *targetnode a'* = *r*
⟨*proof*⟩

lemma *get-return-edges-call-nonempty*:
⟦*valid-edge wfp a*; *kind a* = *Q*:*r*↪*pfs*⟧ ⟹ *get-return-edges wfp a* ≠ {}
⟨*proof*⟩

lemma *only-return-edges-in-get-return-edges*:
⟦*valid-edge wfp a*; *kind a* = *Q*:*r*↪*pfs*; *a'* ∈ *get-return-edges wfp a*⟧
⟹ ∃ *Q' f'*. *kind a'* = *Q'*↩*pf'*
⟨*proof*⟩

abbreviation *lift-procs* :: *wf-prog* ⇒ (*pname* × *vname list* × *vname list*) *list*
where *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* $\Longrightarrow$ *labels* (c_1;;c_2) *l* (c;;c_2)

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

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

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

| *Labels-WhileBody*:
labels c' *l c* $\Longrightarrow$ *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' $\Longrightarrow$ 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*)→$_p$ *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′*⟧
$\Longrightarrow$ ∃ *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′*
$\Longrightarrow$ (*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 *ParamDefs wfp n* ≡ *let* (*prog*,*procs*) = *Rep-wf-prog wfp*; (*p*,*l*) = *n in*
(*if* (*p* = *Main*) *then ParamDefs-proc prog l*
else (*if* (∃ *ins outs c*. (*p*,*ins*,*outs*,*c*) ∈ *set procs*)
then ParamDefs-proc (*THE c'*. ∃ *ins' outs'*. (*p*,*ins'*,*outs'*,*c'*) ∈ *set procs*) *l*
else []))

lemma *ParamDefs-Main-Return-target*:
⟦*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* ⊢ *n* −*CEdge*(*p'*,*es*,*rets*)→$_p$ *n'*⟧
⟹ *ParamDefs wfp* (*Main*,*n'*) = *rets*
⟨*proof*⟩

lemma *ParamDefs-Proc-Return-target*:
assumes *Rep-wf-prog wfp* = (*prog*,*procs*)
and (*p*,*ins*,*outs*,*c*) ∈ *set procs* **and** *c* ⊢ *n* −*CEdge*(*p'*,*es*,*rets*)→$_p$ *n'*
shows *ParamDefs wfp* (*p*,*n'*) = *rets*
⟨*proof*⟩

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

lemma *ParamDefs-Proc-IEdge-Nil*:
assumes *Rep-wf-prog wfp* = (*prog*,*procs*)
and (*p*,*ins*,*outs*,*c*) ∈ *set procs* **and** *c* ⊢ *n* −*IEdge et*→$_p$ *n'*
shows *ParamDefs wfp* (*p*,*n'*) = []
⟨*proof*⟩

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

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

lemma assumes *valid-edge wfp a* **and** *kind a* = $Q' \hookleftarrow_p f'$
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 ⊢ *n* −*CEdge*(*p*,*es*,*rets*)→$_p$ *n'*
⟹ (*THE es'*. ∃ *p' rets' n'*. *prog* ⊢ *n* −*CEdge*(*p'*,*es'*,*rets'*)→$_p$ *n'*) = *es*
⟨*proof*⟩

definition *ParamUses-proc* :: *cmd* ⇒ *label* ⇒ *vname set list*
where *ParamUses-proc c n* ≡
if (∃ *n' p' es' rets'*. *c* ⊢ *n* −*CEdge*(*p'*,*es'*,*rets'*)→$_p$ *n'*) *then*
(*map fv* (*THE es'*. ∃ *p' rets' n'*. *c* ⊢ *n* −*CEdge*(*p'*,*es'*,*rets'*)→$_p$ *n'*))
else []

definition *ParamUses* :: *wf-prog* ⇒ *node* ⇒ *vname set list*
where *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*:
⟦*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* ⊢ *n* −*CEdge*(*p'*,*es*,*rets*)→$_p$ *n'* ⟧

$\Longrightarrow$ *ParamUses wfp* (*Main*,*n*) = *map fv es*
⟨*proof*⟩

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

lemma *ParamUses-Main-IEdge-Nil*:
⟦*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* ⊢ *n* −*IEdge et*→$_p$ *n'*⟧
$\Longrightarrow$ *ParamUses wfp* (*Main*,*n*) = []
⟨*proof*⟩

lemma *ParamUses-Proc-IEdge-Nil*:
assumes *Rep-wf-prog wfp* = (*prog*,*procs*)
and (*p*,*ins*,*outs*,*c*) ∈ *set procs* **and** *c* ⊢ *n* −*IEdge et*→$_p$ *n'*
shows *ParamUses wfp* (*p*,*n*) = []
⟨*proof*⟩

lemma *ParamUses-Main-CEdge-Nil*:
⟦*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* ⊢ *n'* −*CEdge*(*p'*,*es*,*rets*)→$_p$ *n*⟧
$\Longrightarrow$ *ParamUses wfp* (*Main*,*n*) = []
⟨*proof*⟩

lemma *ParamUses-Proc-CEdge-Nil*:
assumes *Rep-wf-prog wfp* = (*prog*,*procs*)
and (*p*,*ins*,*outs*,*c*) ∈ *set procs* **and** *c* ⊢ *n'* −*CEdge*(*p'*,*es*,*rets*)→$_p$ *n*
shows *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*)→$_p$ *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 *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*:*Def wfp* (*Main*, *Entry*) = {}
⟨*proof*⟩

lemma *Exit-Def-empty*:*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*)→$_p$ *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 *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*:*Use wfp* (*Main*, *Entry*) = {}
⟨*proof*⟩

lemma *Exit-Use-empty*:*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*:
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*:
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* ⊢ *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*:
$prog \vdash Label\ l\ -et\rightarrow_p\ Label\ l' \Longrightarrow prog;;c_2 \vdash Label\ l\ -et\rightarrow_p\ Label\ l'$
⟨*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*:
assumes *Rep-wf-prog wfp* = (*prog*,*procs*) **and** *Rep-wf-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*:

assumes *CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*Main*,*n*)
and $n \neq$ *Entry* **and** *Rep-wf-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*:
assumes *Rep-wf-prog wfp* = (*prog*,*procs*) **and** *Rep-wf-prog wfp'* = (c_1;;*prog*,*procs*)
shows ⟦*wfp* ⊢ (*Main*,*n*) $-as\rightarrow*$ (*p'*,*n'*); ∀ *a* ∈ *set as*. *intra-kind* (*kind a*); $n \neq$ *Entry*⟧
$\Longrightarrow$ *wfp'* ⊢ (*Main*,$n \oplus \#{:}c_1$) $-as \oplus s\ \#{:}c_1\rightarrow*$ (*p'*,*n'* ⊕ $\#{:}c_1$)
⟨*proof*⟩

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

lemma *Proc-CFG-edge-CondTrue-source-not-Entry*:
⟦*prog* ⊢ $n -et\rightarrow_p n'$; $n \neq$ *Entry*⟧ $\Longrightarrow$ *if* (*b*) *prog else* c_2 ⊢ $n \oplus 1 -et\rightarrow_p n' \oplus 1$
⟨*proof*⟩

lemma *PCFG-Main-edge-CondTrue-source-not-Entry*:
⟦*prog*,*procs* ⊢ (*Main*,*n*) $-et\rightarrow$ (*p'*,*n'*); $n \neq$ *Entry*; *intra-kind et*; *well-formed procs*⟧
$\Longrightarrow$ *if* (*b*) *prog else* c_2,*procs* ⊢ (*Main*,$n \oplus 1$) $-et\rightarrow$ (*p'*,$n' \oplus 1$)
⟨*proof*⟩

lemma *valid-node-Main-CondTrue*:
assumes *CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*Main*,*n*)
and $n \neq$ *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*, $n \oplus 1$)
⟨*proof*⟩

lemma *path-Main-CondTrue*:
assumes *Rep-wf-prog wfp* = (*prog*,*procs*)
and *Rep-wf-prog wfp'* = (*if* (*b*) *prog else* c_2,*procs*)
shows ⟦*wfp* ⊢ (*Main*,*n*) $-as\rightarrow*$ (*p'*,*n'*); ∀ *a* ∈ *set as*. *intra-kind* (*kind a*); $n \neq$ *Entry*⟧
$\Longrightarrow$ *wfp'* ⊢ (*Main*,$n \oplus 1$) $-as \oplus s\ 1\rightarrow*$ (*p'*,$n' \oplus 1$)
⟨*proof*⟩

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

lemma *Proc-CFG-edge-CondFalse-source-not-Entry*:
⟦*prog* ⊢ $n -et\rightarrow_p n'$; $n \neq$ *Entry*⟧
$\Longrightarrow$ *if* (*b*) c_1 *else prog* ⊢ $n \oplus (\#{:}c_1 + 1) -et\rightarrow_p n' \oplus (\#{:}c_1 + 1)$
⟨*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*:
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*:
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*:
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*:
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*:
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 (λa. lift-edge a p) as*

lemma *lift-path-Proc*:
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*:
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*:
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*:
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 *listE-length* [*simp del*]

3.1.1 Type definitions

Wellformed Programs

definition *wf-jvmprog* = {(*P*, *Phi*). *wf-jvm-prog*$_{Phi}$ *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*: *wf-jvm-prog*$_{TYPING\ P}$ (*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-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-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-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-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-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-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-of $(PROG\ P)\ C\ M\ !\ pc =$ *Getfield F Cl*;
$ek = \Uparrow(\lambda s.\ s(Exception := None)$
$(Stack\ (stkLength\ (P,\ C,\ M)\ pc' - 1) \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer))))$ ⟧

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

| *CFG-Putfield-Check-Normal*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
instrs-of $(PROG\ P)\ C\ M\ !\ pc = Putfield\ F\ Cl;$
$ek = (\lambda s.\ stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 2) \neq Null)_{\surd}\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc \rfloor,\ Normal)$

| *CFG-Putfield-Check-Exceptional*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
instrs-of $(PROG\ P)\ C\ M\ !\ pc = Putfield\ F\ Cl;$
$pc' = (case\ (match\text{-}ex\text{-}table\ (PROG\ P)\ NullPointer\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M))\ of$
$None \Rightarrow None$
$| Some\ (pc'',\ d) \Rightarrow \lfloor pc'' \rfloor);$
$ek = (\lambda s.\ stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 2) = Null)_{\surd}\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ pc'\ Enter)$

| *CFG-Putfield-Update*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Normal);$
instrs-of $(PROG\ P)\ C\ M\ !\ pc = Putfield\ F\ Cl;$
$ek = \Uparrow(\lambda s.\ let\ v = stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 1);$
$r = stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 2);$
$a = the\text{-}Addr\ r;$
$(D,\ fs) = the\ (heap\text{-}of\ s\ a);$
$h' = (heap\text{-}of\ s)(a \mapsto (D,\ fs((F,\ Cl) \mapsto v)))$
$in\ s(Heap \mapsto Hp\ h'))\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Normal)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$

| *CFG-Putfield-Exceptional-prop*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ None\ Enter);$
instrs-of $(PROG\ P)\ C\ M\ !\ pc = Putfield\ F\ Cl;$
$ek = \Uparrow(\lambda s.\ s(Exception \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer))))\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ None\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ None,\ Return)$

| *CFG-Putfield-Exceptional-handle*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Enter);$
instrs-of $(PROG\ P)\ C\ M\ !\ pc = Putfield\ F\ Cl;$
$ek = \Uparrow(\lambda s.\ s(Exception := None)$
$(Stack\ (stkLength\ (P,\ C,\ M)\ pc' - 1) \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer))))\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc' \rfloor,\ Enter)$

| *CFG-Checkcast-Check-Normal*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
instrs-of $(PROG\ P)\ C\ M\ !\ pc = Checkcast\ Cl;$
$ek = (\lambda s.\ cast\text{-}ok\ (PROG\ P)\ Cl\ (heap\text{-}of\ s)\ (stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 1)))_{\surd}\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$

| *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*)))√ ⟧
⟹ (*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*)⌋
)√ ⟧
⟹ (*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*);

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

$ek = Q{:}(C,\ M,\ pc){\hookrightarrow}_{(D,M')} paramDefs$
$\rrbracket$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Normal)\ -(ek)\rightarrow\ (D,\ M',\ None,\ Enter)$
| *CFG-Invoke-False*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Normal);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n;$
$ek = (\lambda s.\ False)_{\surd}$
$\rrbracket$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Normal)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc \rfloor,\ Return)$
| *CFG-Invoke-Return-Check-Normal*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Return);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n;$
$(TYPING\ P)\ C\ M\ !\ pc = \lfloor (ST,\ LT) \rfloor;$
$ST\ !\ n \neq NT;$
$ek = (\lambda s.\ s\ Exception = None)_{\surd}$
$\rrbracket$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Return)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$
| *CFG-Invoke-Return-Check-Exceptional*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Return);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n;$
$match\text{-}ex\text{-}table\ (PROG\ P)\ Exc\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M) = \lfloor (pc',\ diff) \rfloor;$
$pc' \neq length\ (instrs\text{-}of\ (PROG\ P)\ C\ M);$
$ek = (\lambda s.\ \exists v\ d.\ s\ Exception = \lfloor v \rfloor\ \wedge$
$match\text{-}ex\text{-}table\ (PROG\ P)\ (cname\text{-}of\ (heap\text{-}of\ s)\ (the\text{-}Addr\ (the\text{-}Value\ v)))\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M) = \lfloor (pc',\ d) \rfloor)_{\surd}$
$\rrbracket$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Return)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Return)$
| *CFG-Invoke-Return-Exceptional-handle*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Return);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n;$
$ek = \Uparrow (\lambda s.\ s(Exception := None,$
$Stack\ (stkLength\ (P,\ C,\ M)\ pc' - 1) := s\ Exception))\ \rrbracket$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Return)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc' \rfloor,\ Enter)$
| *CFG-Invoke-Return-Exceptional-prop*: $[\![\ C \neq ClassMain\ P;$
$(P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Return);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n;$
$ek = (\lambda s.\ \exists v.\ s\ Exception = \lfloor v \rfloor\ \wedge$
$match\text{-}ex\text{-}table\ (PROG\ P)\ (cname\text{-}of\ (heap\text{-}of\ s)\ (the\text{-}Addr\ (the\text{-}Value\ v)))\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M) = None)_{\surd}\ \rrbracket$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Return)\ -(ek)\rightarrow\ (C,\ M,\ None,\ Return)$
| *CFG-Return*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = instr.Return;$
$ek = \Uparrow (\lambda s.\ s(Stack\ 0 := s\ (Stack\ (stkLength\ (P,\ C,\ M)\ pc - 1))))$
$\rrbracket$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ None,\ Return)$
| *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*:
⟦ $(P,\ C0,\ Main) \vdash (C',\ M',\ pc',\ nt')\ -ek\rightarrow\ (C,\ M,\ pc,\ nt)$; $C \neq ClassMain\ P$; $C' \neq ClassMain\ P$ ⟧
$\Longrightarrow (\exists\ Ts\ T\ mb.\ (PROG\ P) \vdash C\ sees\ M{:}Ts{\rightarrow}T = mb\ in\ C) \wedge$
$(\exists\ Ts\ T\ mb.\ (PROG\ P) \vdash C'\ sees\ M'{:}Ts{\rightarrow}T = mb\ in\ C')$
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 ⟦ $(P, C0, Main) \vdash (C', M', pc', nt') -ek\rightarrow (C, M, pc, nt)$; $C \neq ClassMain\ P$; $C' \neq ClassMain\ P$ ⟧
$\Longrightarrow$ (*case pc of None* ⇒ *True* |
$\lfloor pc'' \rfloor$ ⇒ $(TYPING\ P)\ C\ M\ !\ pc'' \neq None \wedge pc'' < length\ (instrs\text{-}of\ (PROG\ P)\ C\ M))$ ∧
(*case pc′ of None* ⇒ *True* |
$\lfloor pc'' \rfloor$ ⇒ $(TYPING\ P)\ C'\ M'\ !\ pc'' \neq None \wedge pc'' < length\ (instrs\text{-}of\ (PROG\ P)\ C'\ M'))$
and *instr-of-reachable-node-typable*: ⟦ $(P, C0, Main) \vdash \Rightarrow(C, M, pc, nt)$; $C \neq ClassMain\ P$ ⟧
$\Longrightarrow$ *case pc of None* ⇒ *True* |
$\lfloor pc'' \rfloor$ ⇒ $(TYPING\ P)\ C\ M\ !\ pc'' \neq None \wedge pc'' < length\ (instrs\text{-}of\ (PROG\ P)\ C\ M)$
⟨*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
⟦ $(P, C0, Main) \vdash (C, M, pc, nt) -ek\rightarrow (C', M', pc', nt')$; $C \neq ClassMain\ P \vee C' \neq ClassMain\ P$ ⟧
$\Longrightarrow \exists\ T\ mb\ D.\ PROG\ P \vdash C0\ sees\ Main{:}[]\rightarrow T = mb\ in\ D$
and *reachable-node-impl-Main-ex*:
⟦ $(P, C0, Main) \vdash \Rightarrow(C, M, pc, nt)$; $C \neq ClassMain\ P$ ⟧
$\Longrightarrow \exists\ T\ mb\ D.\ PROG\ P \vdash C0\ sees\ Main{:}[]\rightarrow T = mb\ in\ D$
⟨*proof*⟩

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

3.2 Instatiation of the *CFG* locale

abbreviation *sourcenode* :: *cfg-edge* ⇒ *cfg-node*
where *sourcenode e* ≡ *fst e*

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

abbreviation *kind* :: *cfg-edge* ⇒ (*var*, *val*, *cname* × *mname* × *pc*, *cname* × *mname*) *edge-kind*
where *kind e* ≡ *fst*(*snd e*)

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

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) ∈ set (procs P) ⟹ ∃ D fs ms. (C, D, fs, ms) ∈ set P ∧ M ∈ set (map fst ms)
⟨proof⟩

lemma *in-set-procsE'*:
assumes ((C, M), ins, outs) ∈ set (procs P)
obtains D fs ms Ts T mb
where (C, D, fs, ms) ∈ set P
and (M, Ts, T, mb) ∈ set ms
and ins = Heap # (map (λn. 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-jvmprog* ⇒ *cfg-edge* ⇒ *cfg-edge set*
for P :: *wf-jvmprog*
and a :: *cfg-edge*
where
kind a = Q:(C, M, pc)↪(D, M')paramDefs
⟹ ((D, M', None, Return),
(λ(s, ret). ret = (C, M, pc))↩(D, M')(λs s'. s'(Heap := s Heap, Exception := s Exception,
Stack (stkLength (P, C, M) (Suc pc) − 1) := s (Stack 0))),
(C, M, ⌊pc⌋, Return)) ∈ (get-return-edges P a)

lemma *get-return-edgesE* [elim!]:
assumes a ∈ get-return-edges P a'
obtains Q C M pc D M' paramDefs **where**
kind a' = Q:(C, M, pc)↪(D, M')paramDefs
and a = ((D, M', None, Return),
(λ(s, ret). ret = (C, M, pc))↩(D, M')(λs s'. s'(Heap := s Heap, Exception := s Exception,
Stack (stkLength (P, C, M) (Suc pc) − 1) := s (Stack 0))),
(C, M, ⌊pc⌋, Return))
⟨proof⟩

lemma *distinct-class-names*: distinct-fst (PROG P)
⟨proof⟩

lemma *distinct-method-names*:
class (PROG P) C = ⌊(D, fs, ms)⌋ ⟹ distinct-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*:

$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ \mathit{Exceptional}\ \lfloor pc' \rfloor\ \mathit{Enter});$
$C \neq \mathit{ClassMain}\ P;$
$i = \mathit{stkLength}\ (P,\ C,\ M)\ pc' - 1\ ]\!]$
$\Longrightarrow \mathit{Stack}\ i \in \mathit{Def}\ P\ n$

| *Def-Exception-handle-return*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ \mathit{Exceptional}\ \lfloor pc' \rfloor\ \mathit{Return});$
$C \neq \mathit{ClassMain}\ P;$
$i = \mathit{stkLength}\ (P,\ C,\ M)\ pc' - 1\ ]\!]$
$\Longrightarrow \mathit{Stack}\ i \in \mathit{Def}\ P\ n$

| *Def-Getfield*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ \mathit{Normal});$
$C \neq \mathit{ClassMain}\ P;$
$\mathit{instrs\text{-}of}\ (\mathit{PROG}\ P)\ C\ M\ !\ pc = \mathit{Getfield}\ Cl\ Fd;$
$i = \mathit{stkLength}\ (P,\ C,\ M)\ pc - 1\ ]\!]$
$\Longrightarrow \mathit{Stack}\ i \in \mathit{Def}\ P\ n$

| *Def-Putfield*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ \mathit{Normal});$
$C \neq \mathit{ClassMain}\ P;$
$\mathit{instrs\text{-}of}\ (\mathit{PROG}\ P)\ C\ M\ !\ pc = \mathit{Putfield}\ Cl\ Fd\ ]\!]$
$\Longrightarrow \mathit{Heap} \in \mathit{Def}\ P\ n$

| *Def-Invoke-Return-Heap*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ \mathit{Return});$
$C \neq \mathit{ClassMain}\ P;$
$\mathit{instrs\text{-}of}\ (\mathit{PROG}\ P)\ C\ M\ !\ pc = \mathit{Invoke}\ M'\ n'\ ]\!]$
$\Longrightarrow \mathit{Heap} \in \mathit{Def}\ P\ n$

| *Def-Invoke-Return-Exception*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ \mathit{Return});$
$C \neq \mathit{ClassMain}\ P;$
$\mathit{instrs\text{-}of}\ (\mathit{PROG}\ P)\ C\ M\ !\ pc = \mathit{Invoke}\ M'\ n'\ ]\!]$
$\Longrightarrow \mathit{Exception} \in \mathit{Def}\ P\ n$

| *Def-Invoke-Return-Stack*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ \mathit{Return});$
$C \neq \mathit{ClassMain}\ P;$
$\mathit{instrs\text{-}of}\ (\mathit{PROG}\ P)\ C\ M\ !\ pc = \mathit{Invoke}\ M'\ n';$
$i = \mathit{stkLength}\ (P,\ C,\ M)\ (\mathit{Suc}\ pc) - 1\ ]\!]$
$\Longrightarrow \mathit{Stack}\ i \in \mathit{Def}\ P\ n$

| *Def-Invoke-Call-Heap*:
$[\![n = (C,\ M,\ \mathit{None},\ \mathit{Enter});$
$C \neq \mathit{ClassMain}\ P\ ]\!]$
$\Longrightarrow \mathit{Heap} \in \mathit{Def}\ P\ n$

| *Def-Invoke-Call-Local*:
$[\![n = (C,\ M,\ \mathit{None},\ \mathit{Enter});$
$C \neq \mathit{ClassMain}\ P;$
$i < \mathit{locLength}\ (P,\ C,\ M)\ 0\ ]\!]$
$\Longrightarrow \mathit{Local}\ i \in \mathit{Def}\ P\ n$

| *Def-Return*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ \mathit{Enter});$
$C \neq \mathit{ClassMain}\ P;$
$\mathit{instrs\text{-}of}\ (\mathit{PROG}\ P)\ C\ M\ !\ pc = \mathit{instr.Return}\ ]\!]$

$\Longrightarrow$ *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*)

of New Cl ⇒ True
| Getfield F Cl ⇒ True
| Putfield F Cl ⇒ True
| Invoke M′ n′ ⇒ True
| - ⇒ False ⟧
⟹ *Heap ∈ Use P n*
| *Use-Normal-Stack*:
⟦ *n = (C, M, ⌊pc⌋, Normal)*;
C ≠ ClassMain P;
case (instrs-of (PROG P) C M ! pc)
of Getfield F Cl ⇒ d = 1
| Putfield F Cl ⇒ d ∈ {1, 2}
| Invoke M′ n′ ⇒ d > 0 ∧ d ≤ Suc n′
| - ⇒ False;
i = stkLength (P, C, M) pc − d ⟧
⟹ *Stack i ∈ Use P n*
| *Use-Return-Heap*:
⟦ *n = (C, M, ⌊pc⌋, Return)*;
instrs-of (PROG P) C M ! pc = Invoke M′ n′ ∨ C = ClassMain P ⟧
⟹ *Heap ∈ Use P n*
| *Use-Return-Stack*:
⟦ *n = (C, M, ⌊pc⌋, Return)*;
(*instrs-of (PROG P) C M ! pc = Invoke M′ n′ ∧ i = stkLength (P, C, M) (Suc pc) − 1*) ∨
(*C = ClassMain P ∧ i = 0*) ⟧
⟹ *Stack i ∈ Use P n*
| *Use-Return-Exception*:
⟦ *n = (C, M, ⌊pc⌋, Return)*;
instrs-of (PROG P) C M ! pc = Invoke M′ n′ ∨ C = ClassMain P ⟧
⟹ *Exception ∈ Use P n*
| *Use-Exceptional-Stack*:
⟦ *n = (C, M, ⌊pc⌋, Exceptional opc′ nt)*;
case (instrs-of (PROG P) C M ! pc)
of Throw ⇒ True
| - ⇒ False;
i = stkLength (P, C, M) pc − 1 ⟧
⟹ *Stack i ∈ Use P n*
| *Use-Exceptional-Exception*:
⟦ *n = (C, M, ⌊pc⌋, Exceptional ⌊pc′⌋ Return)*;
instrs-of (PROG P) C M ! pc = Invoke M′ n′ ⟧
⟹ *Exception ∈ Use P n*
| *Use-Method-Leave-Exception*:
⟦ *n = (C, M, None, Return)*;
C ≠ ClassMain P ⟧
⟹ *Exception ∈ Use P n*
| *Use-Method-Leave-Heap*:
⟦ *n = (C, M, None, Return)*;
C ≠ ClassMain P ⟧
⟹ *Heap ∈ Use P n*

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

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

lemma *in-set-ParamDefsE*:
⟦ $V \in set\ (ParamDefs\ P\ n)$;
⋀*C M pc*. ⟦ $n = (C,\ M,\ \lfloor pc \rfloor,\ Return)$;
$V \in \{Heap,\ Stack\ (stkLength\ (P,\ C,\ M)\ (Suc\ pc) - 1),\ Exception\}$ ⟧ ⟹ *thesis* ⟧
⟹ *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-of* $(PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n'$
and $V = Heap \vee V = Stack\ (stkLength\ (P,\ C,\ M)\ pc - Suc\ i)$ **and** $i < Suc\ n'$ **and** $C \neq ClassMain\ P$
⟨*proof*⟩

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

interpretation *JVMCFG-wf*:
CFG-wf sourcenode targetnode kind valid-edge $(P,\ C0,\ Main)$
$(ClassMain\ P,\ MethodMain\ P,\ None,\ Enter)$
$(\lambda(C,\ M,\ pc,\ type).\ (C,\ M))$ *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 $(P,\ C0,\ Main)$
$(ClassMain\ P,\ MethodMain\ P,\ None,\ Enter)$
$(\lambda(C,\ M,\ pc,\ type).\ (C,\ M))$ *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*:
$[\![n\ -as\to_{\surd}*\ n';\ n'\ -[a]\to*\ n'';\ \forall\ Q\ p\ ret\ fs.\ kind\ a \neq Q\hookleftarrow_p ret\ ]\!] \Longrightarrow n\ -as\ @\ [a]\to_{\surd}*\ n''$
⟨*proof*⟩

lemma *valid-node-cases'* [*case-names Source Target*, *consumes 1*]:
$[\![$ *valid-node n*; $\bigwedge e.\ [\![$ *valid-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*: *wf-jvm-prog*$_{Phi\text{-}EP}$ *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*)
(λ(*C*, *M*, *pc*, *type*). (*C*, *M*)) *get-return-edges* (*fst*$_{CFG}$ *P*)
((*ClassMain* (*fst*$_{CFG}$ *P*), *MethodMain* (*fst*$_{CFG}$ *P*)),[],[]) # *procs* (*PROG* (*fst*$_{CFG}$ *P*))
(*ClassMain* (*fst*$_{CFG}$ *P*), *MethodMain* (*fst*$_{CFG}$ *P*))
(*ClassMain* (*fst*$_{CFG}$ *P*), *MethodMain* (*fst*$_{CFG}$ *P*), *None*, *Return*)
Def (*fst*$_{CFG}$ *P*) *Use* (*fst*$_{CFG}$ *P*) *ParamDefs* (*fst*$_{CFG}$ *P*) *ParamUses* (*fst*$_{CFG}$ *P*)
for *P*
⟨*proof*⟩

end
theory *HRBSlicing* **imports**
StaticInter/*CFGExit-wf*
StaticInter/*SemanticsCFG*
StaticInter/*FundamentalProperty*
Proc/*ProcSDG*
JinjaVM-Inter/*JVMSDG*
begin

end

Bibliography

[1] Susan Horwitz and Thomas Reps and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[2] Thomas Reps and Susan Horwitz and Mooly Sagiv and Genevieve Rosay. Speeding up slicing. In *Proc. of FSE'94*, pages 11–20. ACM, 1994

[3] Daniel Wasserrab. Towards certified slicing. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://isa-afp.org/entries/Slicing.shtml, September 2008. Formal proof development.