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

Daniel Wasserrab

March 12, 2013

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''*
by (*metis append-eq-conv-conj append-take-drop-id assms drop-map take-map that*)

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(*atomize-elim*)
from ‹∀ *zs*. *xs* ≠ *ys*@*zs*›
show ∃ *j zs*. *xs* = *take j ys* @ *zs* ∧ *j* < *length ys* ∧
(∀ *k*>*j*. ∀ *zs'*. *xs* ≠ *take k ys* @ *zs'*)

proof(*induct ys arbitrary*:*xs*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons y′ ys′*)
note *IH* = ‹⋀*xs*. ∀ *zs*. *xs* ≠ *ys′* @ *zs* ⟹
∃ *j zs*. *xs* = *take j ys′* @ *zs* ∧ *j* < *length ys′* ∧
(∀ *k*. *j* < *k* ⟶ (∀ *zs′*. *xs* ≠ *take k ys′* @ *zs′*))›
show *?case*
proof(*cases xs*)
case *Nil* **thus** *?thesis* **by** *simp*
next
case (*Cons x′ xs′*)
with ‹∀ *zs*. *xs* ≠ (*y′* # *ys′*) @ *zs*› **have** *x′* ≠ *y′* ∨ (∀ *zs*. *xs′* ≠ *ys′* @ *zs*)
by *simp*
show *?thesis*
proof(*cases x′* = *y′*)
case *True*
with ‹*x′* ≠ *y′* ∨ (∀ *zs*. *xs′* ≠ *ys′* @ *zs*)› **have** ∀ *zs*. *xs′* ≠ *ys′* @ *zs* **by** *simp*
from *IH*[*OF this*] **have** ∃ *j zs*. *xs′* = *take j ys′* @ *zs* ∧ *j* < *length ys′* ∧
(∀ *k*. *j* < *k* ⟶ (∀ *zs′*. *xs′* ≠ *take k ys′* @ *zs′*)) **.**
then obtain *j zs* **where** *xs′* = *take j ys′* @ *zs*
and *j* < *length ys′*
and *all-sub*:∀ *k*. *j* < *k* ⟶ (∀ *zs′*. *xs′* ≠ *take k ys′* @ *zs′*)
by *blast*
from ‹*xs′* = *take j ys′* @ *zs*› *True*
have (*x′*#*xs′*) = *take* (*Suc j*) (*y′* # *ys′*) @ *zs*
by *simp*
from *all-sub True* **have** *all-imp*:∀ *k*. *j* < *k* ⟶
(∀ *zs′*. (*x′*#*xs′*) ≠ *take* (*Suc k*) (*y′* # *ys′*) @ *zs′*)
by *auto*
{ fix *l* **assume** (*Suc j*) < *l*
then obtain *k* **where** [*simp*]:*l* = *Suc k* **by**(*cases l*) *auto*
with ‹(*Suc j*) < *l*› **have** *j* < *k* **by** *simp*
with *all-imp*
have ∀ *zs′*. (*x′*#*xs′*) ≠ *take* (*Suc k*) (*y′* # *ys′*) @ *zs′*
by *simp*
hence ∀ *zs′*. (*x′*#*xs′*) ≠ *take l* (*y′* # *ys′*) @ *zs′*
by *simp* **}**
with ‹(*x′*#*xs′*) = *take* (*Suc j*) (*y′* # *ys′*) @ *zs*› ‹*j* < *length ys′*› *Cons*
show *?thesis* **by** (*metis Suc-length-conv less-Suc-eq-0-disj*)
next
case *False*
with *Cons* **have** ∀ *i zs′*. *i* > *0* ⟶ *xs* ≠ *take i* (*y′* # *ys′*) @ *zs′*
by *auto*(*case-tac i*,*auto*)
moreover
have ∃ *zs*. *xs* = *take 0* (*y′* # *ys′*) @ *zs* **by** *simp*
ultimately show *?thesis* **by**(*rule-tac x*=*0* **in** *exI*,*auto*)
qed
qed

qed
qed

end

Chapter 1

The Framework

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

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

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*⟧
⟹ *f*(*xs* [:=] *ys*)(*xs*!*i*) = (*ys*!*i*)
proof(*induct xs arbitrary*:*ys i*)
case *Nil* **thus** *?case* **by** *simp*
next

case (*Cons x′ xs′*)
note *IH* = ‹⋀*ys i*. ⟦*i* < *length xs′*; *length xs′* = *length ys*; *distinct xs′*⟧
⟹ *f*(*xs′* [:=] *ys*) (*xs′*!*i*) = *ys*!*i*›
from ‹*distinct* (*x′*#*xs′*)› **have** *distinct xs′* **and** *x′* ∉ *set xs′* **by** *simp-all*
from ‹*length* (*x′*#*xs′*) = *length ys*› **obtain** *y′ ys′* **where** [*simp*]:*ys* = *y′*#*ys′*
and *length xs′* = *length ys′*
by(*cases ys*) *auto*
show *?case*
proof(*cases i*)
case *0* **thus** *?thesis* **by** *simp*
next
case (*Suc j*)
with ‹*i* < *length* (*x′*#*xs′*)› **have** *j* < *length xs′* **by** *simp*
from *IH*[*OF this* ‹*length xs′* = *length ys′*› ‹*distinct xs′*›]
have *f*(*xs′* [:=] *ys′*) (*xs′*!*j*) = *ys′*!*j* **.**
with ‹*x′* ∉ *set xs′*› ‹*j* < *length xs′*›
have *f*((*x′*#*xs′*) [:=] *ys*) ((*x′*#*xs′*)!(*Suc j*)) = *ys*!(*Suc j*) **by** *fastforce*
with *Suc* **show** *?thesis* **by** *simp*
qed
qed

lemma *fun-upds-eq*:
assumes *V* ∈ *set xs* **and** *length xs* = *length ys* **and** *distinct xs*
shows *f*(*xs* [:=] *ys*) *V* = *f′*(*xs* [:=] *ys*) *V*
proof −
from ‹*V* ∈ *set xs*› **obtain** *i* **where** *i* < *length xs* **and** *xs*!*i* = *V*
by(*fastforce simp*:*in-set-conv-nth*)
with ‹*length xs* = *length ys*› ‹*distinct xs*›
have *f*(*xs* [:=] *ys*)(*xs*!*i*) = (*ys*!*i*) **by** −(*rule fun-upds-nth*)
moreover
from ‹*i* < *length xs*› ‹*xs*!*i* = *V*› ‹*length xs* = *length ys*› ‹*distinct xs*›
have *f′*(*xs* [:=] *ys*)(*xs*!*i*) = (*ys*!*i*) **by** −(*rule fun-upds-nth*)
ultimately show *?thesis* **using** ‹*xs*!*i* = *V*› **by** *simp*
qed

lemma *fun-upds-notin*:*x* ∉ *set xs* ⟹ *f*(*xs* [:=] *ys*) *x* = *f x*
by(*induct xs arbitrary*:*ys*,*auto*,*case-tac ys*,*auto*)

1.1.1 *distinct-fst*

definition *distinct-fst* :: (′*a* × ′*b*) *list* ⇒ *bool* **where**
distinct-fst ≡ *distinct* ∘ *map fst*

lemma *distinct-fst-Nil* [*simp*]:
distinct-fst []
by(*simp add*:*distinct-fst-def*)

lemma *distinct-fst-Cons* [*simp*]:
distinct-fst ((*k*,*x*)#*kxs*) = (*distinct-fst kxs* ∧ (∀ *y*. (*k*,*y*) ∉ *set kxs*))
by(*auto simp*:*distinct-fst-def image-def*)

lemma *distinct-fst-isin-same-fst*:
⟦(*x*,*y*) ∈ *set xs*; (*x*,*y*′) ∈ *set xs*; *distinct-fst xs*⟧
⟹ *y* = *y*′
by(*induct xs*,*auto simp*:*distinct-fst-def image-def*)

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 ('*var*,'*val*,'*ret*,'*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*
by(*cases et*,*auto simp*:*intra-kind-def*)

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*↪$_{Main}$*f*⟧ ⟹ *False*
and *Main-no-return-source*:⟦*valid-edge a*; *kind a* = *Q'*↩$_{Main}$*f'*⟧ ⟹ *False*
and *callee-in-procs*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*⟧ ⟹ ∃ *ins outs*. (*p*,*ins*,*outs*) ∈ *set procs*
and *get-proc-intra*:⟦*valid-edge a*; *intra-kind*(*kind a*)⟧
⟹ *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)
and *get-proc-call*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*⟧ ⟹ *get-proc* (*targetnode a*) = *p*
and *get-proc-return*:
⟦*valid-edge a*; *kind a* = *Q'*↩$_p$*f'*⟧ ⟹ *get-proc* (*sourcenode a*) = *p*
and *call-edges-only*:⟦*valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*⟧
⟹ ∀ *a'*. *valid-edge a'* ∧ *targetnode a'* = *targetnode a* ⟶
(∃ *Qx rx fsx*. *kind a'* = *Qx*:*rx*↪$_p$*fsx*)
and *return-edges-only*:⟦*valid-edge a*; *kind a* = *Q'*↩$_p$*f'*⟧
⟹ ∀ *a'*. *valid-edge a'* ∧ *sourcenode a'* = *sourcenode a* ⟶
(∃ *Qx fx*. *kind a'* = *Qx*↩$_p$*fx*)
and *get-return-edge-call*:
⟦*valid-edge a*; *kind a* = *Q*:*r*↪$_p$*fs*⟧ ⟹ *get-return-edges a* ≠ {}
and *get-return-edges-valid*:
⟦*valid-edge a*; *a'* ∈ *get-return-edges a*⟧ ⟹ *valid-edge a'*
and *only-call-get-return-edges*:

$[\![valid\text{-}edge\ a;\ a' \in get\text{-}return\text{-}edges\ a]\!] \Longrightarrow \exists\, Q\ r\ p\ fs.\ kind\ a = Q{:}r{\hookrightarrow_p}fs$
and *call-return-edges*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow_p}fs;\ a' \in get\text{-}return\text{-}edges\ a]\!]$
$\Longrightarrow \exists\, Q'\ f'.\ kind\ a' = Q'{\hookleftarrow_p}f'$
and *return-needs-call*: $[\![valid\text{-}edge\ a;\ kind\ a = Q'{\hookleftarrow_p}f']\!]$
$\Longrightarrow \exists!a'.\ valid\text{-}edge\ a' \wedge (\exists\, Q\ r\ fs.\ kind\ a' = Q{:}r{\hookrightarrow_p}fs) \wedge a \in get\text{-}return\text{-}edges\ a'$
and *intra-proc-additional-edge*:
$[\![valid\text{-}edge\ a;\ a' \in get\text{-}return\text{-}edges\ a]\!]$
$\Longrightarrow \exists\, a''.\ valid\text{-}edge\ a'' \wedge sourcenode\ a'' = targetnode\ a \wedge$
$targetnode\ a'' = sourcenode\ a' \wedge kind\ a'' = (\lambda cf.\ False)_{\surd}$
and *call-return-node-edge*:
$[\![valid\text{-}edge\ a;\ a' \in get\text{-}return\text{-}edges\ a]\!]$
$\Longrightarrow \exists\, a''.\ valid\text{-}edge\ a'' \wedge sourcenode\ a'' = sourcenode\ a \wedge$
$targetnode\ a'' = targetnode\ a' \wedge kind\ a'' = (\lambda cf.\ False)_{\surd}$
and *call-only-one-intra-edge*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow_p}fs]\!]$
$\Longrightarrow \exists!a'.\ valid\text{-}edge\ a' \wedge sourcenode\ a' = sourcenode\ a \wedge intra\text{-}kind(kind\ a')$
and *return-only-one-intra-edge*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q'{\hookleftarrow_p}f']\!]$
$\Longrightarrow \exists!a'.\ valid\text{-}edge\ a' \wedge targetnode\ a' = targetnode\ a \wedge intra\text{-}kind(kind\ a')$
and *same-proc-call-unique-target*:
$[\![valid\text{-}edge\ a;\ valid\text{-}edge\ a';\ kind\ a = Q_1{:}r_1{\hookrightarrow_p}fs_1;\ \ kind\ a' = Q_2{:}r_2{\hookrightarrow_p}fs_2]\!]$
$\Longrightarrow targetnode\ a = targetnode\ a'$
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** $a' \in get\text{-}return\text{-}edges\ a$
shows *get-proc* $(sourcenode\ a) = get\text{-}proc\ (targetnode\ a')$
proof −
from *assms* **obtain** *ax* **where** *valid-edge ax* **and** $sourcenode\ a = sourcenode\ ax$
and $targetnode\ a' = targetnode\ ax$ **and** $intra\text{-}kind(kind\ ax)$
by(*auto dest*:*call-return-node-edge simp*:*intra-kind-def*)
thus *?thesis* **by**(*fastforce intro*:*get-proc-intra*)
qed

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** $intra\text{-}kind(kind\ a')$
shows $kind\ a' = (\lambda cf.\ False)_{\surd}$
proof −
from ‹*valid-edge a*› ‹$kind\ a = Q{:}r{\hookrightarrow_p}fs$› **obtain** *ax* **where** $ax \in get\text{-}return\text{-}edges\ a$

by(*fastforce dest*:*get-return-edge-call*)
with ‹*valid-edge a*› **obtain** *a''* **where** *valid-edge a''*
and *sourcenode a'' = sourcenode a* **and** *kind a'' = (λcf. False)$_\surd$*
by(*fastforce dest*:*call-return-node-edge*)
from ‹*kind a'' = (λcf. False)$_\surd$*› **have** *intra-kind*(*kind a''*)
by(*simp add*:*intra-kind-def*)
with *assms* ‹*valid-edge a''*› ‹*sourcenode a'' = sourcenode a*›
‹*kind a'' = (λcf. False)$_\surd$*›
show *?thesis* **by**(*fastforce dest*:*call-only-one-intra-edge*)
qed

lemma *formal-in-THE*:
⟦*valid-edge a*; *kind a = Q*:*r↪$_p$fs*; (*p*,*ins*,*outs*) ∈ *set procs*⟧
⟹ (*THE ins*. ∃ *outs*. (*p*,*ins*,*outs*) ∈ *set procs*) = *ins*
by(*fastforce dest*:*distinct-fst-isin-same-fst intro*:*unique-callers*)

lemma *formal-out-THE*:
⟦*valid-edge a*; *kind a = Q*↩$_p$*f*; (*p*,*ins*,*outs*) ∈ *set procs*⟧
⟹ (*THE outs*. ∃ *ins*. (*p*,*ins*,*outs*) ∈ *set procs*) = *outs*
by(*fastforce dest*:*distinct-fst-isin-same-fst intro*:*unique-callers*)

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*
by(*induct fs arbitrary*:*i*,*auto*,*case-tac i*,*auto*)

lemma [*simp*]:*length* (*params fs cf*) = *length fs*
by(*induct fs*) *auto*

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↪$_p$fs*) (*cf*#*cfs*) =
(*let ins = THE ins*. ∃ *outs*. (*p*,*ins*,*outs*) ∈ *set procs in*
(*empty*(*ins* [:=] *params fs* (*fst cf*)),*r*)#*cf*#*cfs*)
| *transfer* (*Q*↩$_p$*f*)(*cf*#*cfs*) = (*case cfs of* [] ⇒ []
| *cf'*#*cfs'* ⇒ (*f* (*fst cf*) (*fst cf'*),*snd cf'*)#*cfs'*)
| *transfer et* [] = []

fun *transfers* :: (*'var*,*'val*,*'ret*,*'pname*) *edge-kind list* ⇒ ((*'var* ⇀ *'val*) × *'ret*) *list* ⇒
((*'var* ⇀ *'val*) × *'ret*) *list*
where *transfers* [] *s* = *s*
| *transfers* (*et*#*ets*) *s* = *transfers ets* (*transfer et s*)

fun *pred* :: (*'var*,*'val*,*'ret*,*'pname*) *edge-kind* ⇒ ((*'var* ⇀ *'val*) × *'ret*) *list* ⇒ *bool*
where *pred* (⇑*f*) (*cf*#*cfs*) = *True*
| *pred* (*Q*)$_{\surd}$ (*cf*#*cfs*) = *Q* (*fst cf*)
| *pred* (*Q*:*r*↪$_p$*fs*) (*cf*#*cfs*) = *Q* (*fst cf*,*r*)
| *pred* (*Q*↩$_p$*f*) (*cf*#*cfs*) = (*Q cf* ∧ *cfs* ≠ [])
| *pred et* [] = *False*

fun *preds* :: (*'var*,*'val*,*'ret*,*'pname*) *edge-kind list* ⇒ ((*'var* ⇀ *'val*) × *'ret*) *list* ⇒ *bool*
where *preds* [] *s* = *True*
| *preds* (*et*#*ets*) *s* = (*pred et s* ∧ *preds ets* (*transfer et s*))

lemma *transfers-split*:
(*transfers* (*ets*@*ets'*) *s*) = (*transfers ets'* (*transfers ets s*))
by(*induct ets arbitrary*:*s*) *auto*

lemma *preds-split*:
(*preds* (*ets*@*ets'*) *s*) = (*preds ets s* ∧ *preds ets'* (*transfers ets s*))
by(*induct ets arbitrary*:*s*) *auto*

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

valid-node

definition *valid-node* :: *'node* ⇒ *bool*
where *valid-node n* ≡
(∃ *a*. *valid-edge a* ∧ (*n* = *sourcenode a* ∨ *n* = *targetnode a*))

lemma [*simp*]: *valid-edge a* ⟹ *valid-node* (*sourcenode a*)
by(*fastforce simp*:*valid-node-def*)

lemma [*simp*]: *valid-edge a* ⟹ *valid-node* (*targetnode a*)
by(*fastforce simp*:*valid-node-def*)

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*:
⟦$n'' -as\to* n'$; *valid-edge a*; *sourcenode a* = *n*; *targetnode a* = n'⟧
⟹ $n -a\#as\to* n'$

lemma *path-valid-node*:
assumes $n -as\to* n'$ **shows** *valid-node n* **and** *valid-node* n'
using ‹$n -as\to* n'$›
by(*induct rule*:*path.induct*,*auto*)

lemma *empty-path-nodes* [*dest*]:$n -[]\to* n' \Longrightarrow n = n'$
by(*fastforce elim*:*path.cases*)

lemma *path-valid-edges*:$n -as\to* n' \Longrightarrow \forall a \in set\ as.\ valid\text{-}edge\ a$
by(*induct rule*:*path.induct*) *auto*

lemma *path-edge*:*valid-edge a* ⟹ *sourcenode a* $-[a]\to*$ *targetnode a*
by(*fastforce intro*:*Cons-path empty-path*)

lemma *path-Append*:⟦$n -as\to* n''$; $n'' -as'\to* n'$⟧
⟹ $n -as@as'\to* n'$
by(*induct rule*:*path.induct*,*auto intro*:*Cons-path*)

lemma *path-split*:
assumes $n -as@a\#as'\to* n'$
shows $n -as\to*$ *sourcenode a* **and** *valid-edge a* **and** *targetnode a* $-as'\to* n'$
using ‹$n -as@a\#as'\to* n'$›
proof(*induct as arbitrary*:*n*)
case *Nil* **case** *1*
thus *?case* **by**(*fastforce elim*:*path.cases intro*:*empty-path*)
next
case *Nil* **case** *2*
thus *?case* **by**(*fastforce elim*:*path.cases intro*:*path-edge*)
next
case *Nil* **case** *3*
thus *?case* **by**(*fastforce elim*:*path.cases*)
next
case (*Cons ax asx*)
note *IH1* = ‹$\bigwedge n.\ n -asx@a\#as'\to* n' \Longrightarrow n -asx\to*$ *sourcenode a*›
note *IH2* = ‹$\bigwedge n.\ n -asx@a\#as'\to* n' \Longrightarrow$ *valid-edge a*›
note *IH3* = ‹$\bigwedge n.\ n -asx@a\#as'\to* n' \Longrightarrow$ *targetnode a* $-as'\to* n'$›
{ case *1*
hence *sourcenode ax* = *n* **and** *targetnode ax* $-asx@a\#as'\to* n'$ **and** *valid-edge ax*
by(*auto elim*:*path.cases*)

```
    from IH1[OF ‹targetnode ax −asx@a#as'→* n'›]
    have targetnode ax −asx→* sourcenode a .
   with ‹sourcenode ax = n› ‹valid-edge ax› show ?case by(fastforce intro:Cons-path)
  next
    case 2 hence targetnode ax −asx@a#as'→* n' by(auto elim:path.cases)
    from IH2[OF this] show ?case .
  next
    case 3 hence targetnode ax −asx@a#as'→* n' by(auto elim:path.cases)
    from IH3[OF this] show ?case .
  }
qed


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(atomize-elim)
  from ‹as ≠ []› obtain a' as' where as = a'#as' by(cases as) auto
  with ‹n −as→* n'› have n −[]@a'#as'→* n' by simp
  hence n −[]→* sourcenode a' and valid-edge a' and targetnode a' −as'→* n'
    by(rule path-split)+
  from ‹n −[]→* sourcenode a'› have n = sourcenode a' by fast
  with ‹as = a'#as'› ‹valid-edge a'› ‹targetnode a' −as'→* n'›
  show ∃ a' as'. as = a'#as' ∧ n = sourcenode a' ∧ valid-edge a' ∧
               targetnode a' −as'→* n'
    by fastforce
qed


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(atomize-elim)
  from ‹as ≠ []› obtain a' as' where as = as'@[a'] by(cases as rule:rev-cases)
auto
  with ‹n −as→* n'› have n −as'@a'#[]→* n' by simp
  hence n −as'→* sourcenode a' and valid-edge a' and targetnode a' −[]→* n'
    by(rule path-split)+
  from ‹targetnode a' −[]→* n'› have n' = targetnode a' by fast
  with ‹as = as'@[a']› ‹valid-edge a'› ‹n −as'→* sourcenode a'›
  show ∃ as' a'. as = as'@[a'] ∧ n −as'→* sourcenode a' ∧ valid-edge a' ∧
               n' = targetnode a'
    by fastforce
qed


lemma path-split-second:
```

assumes $n\ -as@a\#as'\to*\ n'$ **shows** *sourcenode* $a\ -a\#as'\to*\ n'$
proof −
from ‹$n\ -as@a\#as'\to*\ n'$› **have** *valid-edge* a **and** *targetnode* $a\ -as'\to*\ n'$
by(*auto intro*:*path-split*)
thus *?thesis* **by**(*fastforce intro*:*Cons-path*)
qed

lemma *path-Entry-Cons*:
assumes (*-Entry-*) $-as\to*\ n'$ **and** $n' \neq$ (*-Entry-*)
obtains $n\ a$ **where** *sourcenode* $a =$ (*-Entry-*) **and** *targetnode* $a = n$
and $n\ -tl\ as\to*\ n'$ **and** *valid-edge* a **and** $a = hd\ as$
proof(*atomize-elim*)
from ‹(*-Entry-*) $-as\to*\ n'$› ‹$n' \neq$ (*-Entry-*)› **have** $as \neq []$
by(*cases as*,*auto elim*:*path.cases*)
with ‹(*-Entry-*) $-as\to*\ n'$› **obtain** $a'\ as'$ **where** $as = a'\#as'$
and (*-Entry-*) = *sourcenode* a' **and** *valid-edge* a' **and** *targetnode* $a'\ -as'\to*\ n'$
by(*erule path-split-Cons*)
thus $\exists\, a\ n.$ *sourcenode* $a =$ (*-Entry-*) $\wedge$ *targetnode* $a = n \wedge n\ -tl\ as\to*\ n' \wedge$
valid-edge $a \wedge a = hd\ as$
by *fastforce*
qed

lemma *path-det*:
$[\![n\ -as\to*\ n';\ n\ -as\to*\ n'']\!] \Longrightarrow n' = n''$
proof(*induct as arbitrary*:n)
case *Nil* **thus** *?case* **by**(*auto elim*:*path.cases*)
next
case (*Cons* $a'\ as'$)
note *IH* = ‹$\bigwedge n.\ [\![n\ -as'\to*\ n';\ n\ -as'\to*\ n'']\!] \Longrightarrow n' = n''$›
from ‹$n\ -a'\#as'\to*\ n'$› **have** *targetnode* $a'\ -as'\to*\ n'$
by(*fastforce elim*:*path-split-Cons*)
from ‹$n\ -a'\#as'\to*\ n''$› **have** *targetnode* $a'\ -as'\to*\ n''$
by(*fastforce elim*:*path-split-Cons*)
from *IH*[*OF* ‹*targetnode* $a'\ -as'\to*\ n'$› *this*] **show** *?thesis* **.**
qed

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*→∗ *n'*; *as* ≠ []⟧ ⟹ *hd* (*sourcenodes as*) = *n*
by(*fastforce elim*:*path-split-Cons simp*:*sourcenodes-def*)

lemma *path-targetnode*:
⟦*n* −*as*→∗ *n'*; *as* ≠ []⟧ ⟹ *last* (*targetnodes as*) = *n'*
by(*fastforce elim*:*path-split-snoc simp*:*targetnodes-def*)

lemma *sourcenodes-is-n-Cons-butlast-targetnodes*:
⟦*n* −*as*→∗ *n'*; *as* ≠ []⟧ ⟹
sourcenodes as = *n*#(*butlast* (*targetnodes as*))
proof(*induct as arbitrary*:*n*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a' as'*)
note *IH* = ‹⋀*n*. ⟦*n* −*as'*→∗ *n'*; *as'* ≠ []⟧
⟹ *sourcenodes as'* = *n*#(*butlast* (*targetnodes as'*))›
from ‹*n* −*a'*#*as'*→∗ *n'*› **have** *n* = *sourcenode a'* **and** *targetnode a'* −*as'*→∗ *n'*
by(*auto elim*:*path-split-Cons*)
show *?case*
proof(*cases as'* = [])
case *True*
with ‹*targetnode a'* −*as'*→∗ *n'*› **have** *targetnode a'* = *n'* **by** *fast*
with *True* ‹*n* = *sourcenode a'*› **show** *?thesis*
by(*simp add*:*sourcenodes-def targetnodes-def*)
next
case *False*
from *IH*[*OF* ‹*targetnode a'* −*as'*→∗ *n'*› *this*]
have *sourcenodes as'* = *targetnode a'* # *butlast* (*targetnodes as'*) **.**
with ‹*n* = *sourcenode a'*› *False* **show** *?thesis*
by(*simp add*:*sourcenodes-def targetnodes-def*)
qed
qed

lemma *targetnodes-is-tl-sourcenodes-App-n'*:
⟦*n* −*as*→∗ *n'*; *as* ≠ []⟧ ⟹
targetnodes as = (*tl* (*sourcenodes as*))@[*n'*]
proof(*induct as arbitrary*:*n' rule*:*rev-induct*)
case *Nil* **thus** *?case* **by** *simp*
next

```
  case (snoc a' as')
  note IH = ⟨⋀n'. ⟦n -as'→* n'; as' ≠ []⟧
    ⟹ targetnodes as' = tl (sourcenodes as') @ [n']⟩
  from ⟨n -as'@[a']→* n'⟩ have n -as'→* sourcenode a' and n' = targetnode a'
    by(auto elim:path-split-snoc)
  show ?case
  proof(cases as' = [])
    case True
    with ⟨n -as'→* sourcenode a'⟩ have n = sourcenode a' by fast
    with True ⟨n' = targetnode a'⟩ show ?thesis
      by(simp add:sourcenodes-def targetnodes-def)
  next
    case False
    from IH[OF ⟨n -as'→* sourcenode a'⟩ this]
    have targetnodes as' = tl (sourcenodes as')@[sourcenode a'] .
    with ⟨n' = targetnode a'⟩ False show ?thesis
      by(simp add:sourcenodes-def targetnodes-def)
  qed
qed
```

Intraprocedural paths

```
definition intra-path :: 'node ⇒ 'edge list ⇒ 'node ⇒ bool
  (- -_ι→* - [51,0,0] 80)
where n -as→_ι* n' ≡ n -as→* n' ∧ (∀ a ∈ set as. intra-kind(kind a))

lemma intra-path-get-procs:
  assumes n -as→_ι* n' shows get-proc n = get-proc n'
proof -
  from ⟨n -as→_ι* n'⟩ have n -as→* n' and ∀ a ∈ set as. intra-kind(kind a)
    by(simp-all add:intra-path-def)
  thus ?thesis
  proof(induct as arbitrary:n)
    case Nil thus ?case by fastforce
  next
    case (Cons a' as')
    note IH = ⟨⋀n. ⟦n -as'→* n'; ∀ a∈set as'. intra-kind (kind a)⟧
      ⟹ get-proc n = get-proc n'⟩
    from ⟨∀ a∈set (a'#as'). intra-kind (kind a)⟩
    have intra-kind(kind a') and ∀ a∈set as'. intra-kind (kind a) by simp-all
    from ⟨n -a'#as'→* n'⟩ have sourcenode a' = n and valid-edge a'
      and targetnode a' -as'→* n' by(auto elim:path.cases)
    from IH[OF ⟨targetnode a' -as'→* n'⟩ ⟨∀ a∈set as'. intra-kind (kind a)⟩]
    have get-proc (targetnode a') = get-proc n' .
    from ⟨valid-edge a'⟩ ⟨intra-kind(kind a')⟩
    have get-proc (sourcenode a') = get-proc (targetnode a')
      by(rule get-proc-intra)
    with ⟨sourcenode a' = n⟩ ⟨get-proc (targetnode a') = get-proc n'⟩
    show ?case by simp
```

qed
qed

lemma *intra-path-Append*:
⟦$n\ -as\rightarrow_\iota*\ n''$; $n''\ -as'\rightarrow_\iota*\ n'$⟧ ⟹ $n\ -as@as'\rightarrow_\iota*\ n'$
by(*fastforce intro*:*path-Append simp*:*intra-path-def*)

lemma *get-proc-get-return-edges*:
assumes *valid-edge a* **and** $a' \in$ *get-return-edges a*
shows *get-proc*(*targetnode a*) = *get-proc*(*sourcenode a'*)
proof −
from ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*›
obtain a'' **where** *valid-edge* a'' **and** *sourcenode* a'' = *targetnode a*
and *targetnode* a'' = *sourcenode* a' **and** *kind* $a'' = (\lambda cf.\ False)_\surd$
by(*fastforce dest*:*intra-proc-additional-edge*)
from ‹*valid-edge* a''› ‹*kind* $a'' = (\lambda cf.\ False)_\surd$›
have *get-proc*(*sourcenode* a'') = *get-proc*(*targetnode* a'')
by(*fastforce intro*:*get-proc-intra simp*:*intra-kind-def*)
with ‹*sourcenode* a'' = *targetnode a*› ‹*targetnode* a'' = *sourcenode* a'›
show *?thesis* **by** *simp*
qed

Valid paths

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

fun *valid-path-aux* :: *'edge list* ⇒ *'edge list* ⇒ *bool*
where *valid-path-aux cs* [] ⟷ *True*
| *valid-path-aux cs* (*a*#*as*) ⟷
(*case* (*kind a*) *of* Q:$r\hookrightarrow_p fs$ ⇒ *valid-path-aux* (*a*#*cs*) *as*
| $Q\hookleftarrow_p f$ ⇒ *case cs of* [] ⇒ *valid-path-aux* [] *as*
| c'#cs' ⇒ *a* ∈ *get-return-edges* c' ∧
valid-path-aux cs' *as*
| - ⇒ *valid-path-aux cs as*)

lemma *vpa-induct* [*consumes 1*,*case-names vpa-empty vpa-intra vpa-Call vpa-ReturnEmpty vpa-ReturnCons*]:
assumes *major*: *valid-path-aux xs ys*
and *rules*: ⋀*cs*. *P cs* []
⋀*cs a as*. ⟦*intra-kind*(*kind a*); *valid-path-aux cs as*; *P cs as*⟧ ⟹ *P cs* (*a*#*as*)
⋀*cs a as Q r p fs*. ⟦*kind a* = Q:$r\hookrightarrow_p fs$; *valid-path-aux* (*a*#*cs*) *as*; *P* (*a*#*cs*) *as*⟧
⟹ *P cs* (*a*#*as*)
⋀*cs a as Q p f*. ⟦*kind a* = $Q\hookleftarrow_p f$; *cs* = []; *valid-path-aux* [] *as*; *P* [] *as*⟧
⟹ *P cs* (*a*#*as*)
⋀*cs a as Q p f c' cs'* . ⟦*kind a* = $Q\hookleftarrow_p f$; *cs* = c'#cs'; *valid-path-aux* cs' *as*;

$a \in$ *get-return-edges* c'; P cs' $as\rrbracket$
$\Longrightarrow P$ cs $(a\#as)$
shows P xs ys
using *major*
apply(*induct ys arbitrary*: *xs*)
by(*auto intro*:*rules split*:*edge-kind.split-asm list.split-asm simp*:*intra-kind-def*)

lemma *valid-path-aux-intra-path*:
$\forall a \in$ *set as*. *intra-kind*(*kind a*) $\Longrightarrow$ *valid-path-aux cs as*
by(*induct as*,*auto simp*:*intra-kind-def*)

lemma *valid-path-aux-callstack-prefix*:
valid-path-aux $(cs@cs')$ *as* $\Longrightarrow$ *valid-path-aux cs as*
proof(*induct cs*@*cs' as arbitrary*:*cs cs' rule*:*vpa-induct*)
case *vpa-empty* **thus** *?case* **by** *simp*
next
case (*vpa-intra a as*)
hence *valid-path-aux cs as* **by** *simp*
with ‹*intra-kind* (*kind a*)› **show** *?case* **by**(*cases kind a*,*auto simp*:*intra-kind-def*)
next
case (*vpa-Call a as Q r p fs cs'' cs'*)
note *IH* = ‹$\bigwedge$*xs ys*. *a*#*cs''*@*cs'* = *xs*@*ys* $\Longrightarrow$ *valid-path-aux xs as*›
have *a*#*cs''*@*cs'* = (*a*#*cs''*)@*cs'* **by** *simp*
from *IH*[*OF this*] **have** *valid-path-aux* (*a*#*cs''*) *as* **.**
with ‹*kind a* = *Q*:*r*↪$_p$*fs*› **show** *?case* **by** *simp*
next
case (*vpa-ReturnEmpty a as Q p f cs'' cs'*)
hence *valid-path-aux cs'' as* **by** *simp*
with ‹*kind a* = *Q*↩$_p$*f*› ‹*cs''*@*cs'* = []› **show** *?case* **by** *simp*
next
case (*vpa-ReturnCons a as Q p f c' cs' csx csx'*)
note *IH* = ‹$\bigwedge$*xs ys*. *cs'* = *xs*@*ys* $\Longrightarrow$ *valid-path-aux xs as*›
from ‹*csx*@*csx'* = *c'*#*cs'*›
have *csx* = [] $\wedge$ *csx'* = *c'*#*cs'* $\vee$ ($\exists$ *zs*. *csx* = *c'*#*zs* $\wedge$ *zs*@*csx'* = *cs'*)
by(*simp add*:*append-eq-Cons-conv*)
thus *?case*
proof
assume *csx* = [] $\wedge$ *csx'* = *c'*#*cs'*
hence *csx* = [] **and** *csx'* = *c'*#*cs'* **by** *simp-all*
from ‹*csx'* = *c'*#*cs'*› **have** *cs'* = []@*tl csx'* **by** *simp*
from *IH*[*OF this*] **have** *valid-path-aux* [] *as* **.**
with ‹*csx* = []› ‹*kind a* = *Q*↩$_p$*f*› **show** *?thesis* **by** *simp*
next
assume $\exists$ *zs*. *csx* = *c'*#*zs* $\wedge$ *zs*@*csx'* = *cs'*
then obtain *zs* **where** *csx* = *c'*#*zs* **and** *cs'* = *zs*@*csx'* **by** *auto*
from *IH*[*OF* ‹*cs'* = *zs*@*csx'*›] **have** *valid-path-aux zs as* **.**
with ‹*csx* = *c'*#*zs*› ‹*kind a* = *Q*↩$_p$*f*› ‹*a* ∈ *get-return-edges c'*›

```
    show ?thesis by simp
  qed
qed


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 = []
by(cases cs) auto


lemma upd-cs-intra-path:
  ∀ a ∈ set as. intra-kind(kind a) ⟹ upd-cs cs as = cs
by(induct as,auto simp:intra-kind-def)


lemma upd-cs-Append:
  ⟦upd-cs cs as = cs'; upd-cs cs' as' = cs''⟧ ⟹ upd-cs cs (as@as') = cs''
by(induct as arbitrary:cs,auto split:edge-kind.split list.split)


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 ∀ xs' ys'. xs = xs'@ys' ∧ ys' ≠ [] ⟶ upd-cs cs xs' ≠ []
  and upd-cs [] ys = []
proof(atomize-elim)
  from ‹upd-cs cs as = []› ‹cs ≠ []› ‹as ≠ []›
  show ∃ xs ys. as = xs@ys ∧ xs ≠ [] ∧ upd-cs cs xs = [] ∧
            (∀ xs' ys'. xs = xs'@ys' ∧ ys' ≠ [] ⟶ upd-cs cs xs' ≠ []) ∧
            upd-cs [] ys = []
  proof(induct as arbitrary:cs)
    case Nil thus ?case by simp
  next
    case (Cons a' as')
    note IH = ‹⋀cs. ⟦upd-cs cs as' = []; cs ≠ []; as' ≠ []⟧
      ⟹ ∃ xs ys. as' = xs@ys ∧ xs ≠ [] ∧ upd-cs cs xs = [] ∧
                (∀ xs' ys'. xs = xs'@ys' ∧ ys' ≠ [] ⟶ upd-cs cs xs' ≠ []) ∧
                upd-cs [] ys = []›
    show ?case
    proof(cases kind a' rule:edge-kind-cases)
```

case *Intra*
with ‹*upd-cs cs* (*a′*#*as′*) = []› **have** *upd-cs cs as′* = []
by(*fastforce simp*:*intra-kind-def*)
with ‹*cs* ≠ []› **have** *as′* ≠ [] **by** *fastforce*
from *IH*[*OF* ‹*upd-cs cs as′* = []› ‹*cs* ≠ []› *this*] **obtain** *xs ys* **where** *as′* = *xs*@*ys*
and *xs* ≠ [] **and** *upd-cs cs xs* = [] **and** *upd-cs* [] *ys* = []
and ∀ *xs′ ys′*. *xs* = *xs′*@*ys′* ∧ *ys′* ≠ [] ⟶ *upd-cs cs xs′* ≠ [] **by** *blast*
from ‹*upd-cs cs xs* = []› *Intra* **have** *upd-cs cs* (*a′*#*xs*) = []
by(*fastforce simp*:*intra-kind-def*)
from ‹∀ *xs′ ys′*. *xs* = *xs′*@*ys′* ∧ *ys′* ≠ [] ⟶ *upd-cs cs xs′* ≠ []› ‹*xs* ≠ []› *Intra*
have ∀ *xs′ ys′*. *a′*#*xs* = *xs′*@*ys′* ∧ *ys′* ≠ [] ⟶ *upd-cs cs xs′* ≠ []
apply *auto*
apply(*case-tac xs′*) **apply**(*auto simp*:*intra-kind-def*)
by(*erule-tac x*=[] **in** *allE*,*fastforce*)+
with ‹*as′* = *xs*@*ys*› ‹*upd-cs cs* (*a′*#*xs*) = []› ‹*upd-cs* [] *ys* = []›
show *?thesis* **apply**(*rule-tac x*=*a′*#*xs* **in** *exI*) **by** *fastforce*
next
case (*Call Q p f*)
with ‹*upd-cs cs* (*a′*#*as′*) = []› **have** *upd-cs* (*a′*#*cs*) *as′* = [] **by** *simp*
with ‹*cs* ≠ []› **have** *as′* ≠ [] **by** *fastforce*
from *IH*[*OF* ‹*upd-cs* (*a′*#*cs*) *as′* = []› - *this*] **obtain** *xs ys* **where** *as′* = *xs*@*ys*
and *xs* ≠ [] **and** *upd-cs* (*a′*#*cs*) *xs* = [] **and** *upd-cs* [] *ys* = []
and ∀ *xs′ ys′*. *xs* = *xs′*@*ys′* ∧ *ys′* ≠ [] ⟶ *upd-cs* (*a′*#*cs*) *xs′* ≠ [] **by** *blast*
from ‹*upd-cs* (*a′*#*cs*) *xs* = []› *Call* **have** *upd-cs cs* (*a′*#*xs*) = [] **by** *simp*
from ‹∀ *xs′ ys′*. *xs* = *xs′*@*ys′* ∧ *ys′* ≠ [] ⟶ *upd-cs* (*a′*#*cs*) *xs′* ≠ []›
‹*xs* ≠ []› ‹*cs* ≠ []› *Call*
have ∀ *xs′ ys′*. *a′*#*xs* = *xs′*@*ys′* ∧ *ys′* ≠ [] ⟶ *upd-cs cs xs′* ≠ []
by *auto*(*case-tac xs′*,*auto*)
with ‹*as′* = *xs*@*ys*› ‹*upd-cs cs* (*a′*#*xs*) = []› ‹*upd-cs* [] *ys* = []›
show *?thesis* **apply**(*rule-tac x*=*a′*#*xs* **in** *exI*) **by** *fastforce*
next
case (*Return Q p f*)
with ‹*upd-cs cs* (*a′*#*as′*) = []› ‹*cs* ≠ []› **obtain** *c′ cs′* **where** *cs* = *c′*#*cs′*
and *upd-cs cs′ as′* = [] **by**(*cases cs*) *auto*
show *?thesis*
proof(*cases cs′* = [])
case *True*
with ‹*cs* = *c′*#*cs′*› ‹*upd-cs cs′ as′* = []› *Return* **show** *?thesis*
apply(*rule-tac x*=[*a′*] **in** *exI*) **apply** *clarsimp*
by(*case-tac xs′*) *auto*
next
case *False*
with ‹*upd-cs cs′ as′* = []› **have** *as′* ≠ [] **by** *fastforce*
from *IH*[*OF* ‹*upd-cs cs′ as′* = []› *False this*] **obtain** *xs ys* **where** *as′* = *xs*@*ys*
and *xs* ≠ [] **and** *upd-cs cs′ xs* = [] **and** *upd-cs* [] *ys* = []
and ∀ *xs′ ys′*. *xs* = *xs′*@*ys′* ∧ *ys′* ≠ [] ⟶ *upd-cs cs′ xs′* ≠ [] **by** *blast*

```
      from ‹upd-cs cs' xs = []› ‹cs = c'#cs'› Return have upd-cs cs (a'#xs) = []
        by simp
      from ‹∀ xs' ys'. xs = xs'@ys' ∧ ys' ≠ [] ⟶ upd-cs cs' xs' ≠ []›
        ‹xs ≠ []› ‹cs = c'#cs'› Return
      have ∀ xs' ys'. a'#xs = xs'@ys' ∧ ys' ≠ [] ⟶ upd-cs cs xs' ≠ []
        by auto(case-tac xs',auto)
      with ‹as' = xs@ys› ‹upd-cs cs (a'#xs) = []› ‹upd-cs [] ys = []›
      show ?thesis apply(rule-tac x=a'#xs in exI) by fastforce
    qed
  qed
 qed
qed


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(induct as arbitrary:cs)
 case Nil
 with ‹kind a = Q↩pf› have upd-cs cs [a] = cs' by simp
 thus ?case by simp
next
 case (Cons a' as')
 note IH = ‹⋀cs. upd-cs cs as' = c'#cs' ⟹ upd-cs cs (as'@[a]) = cs'›
 show ?case
 proof(cases kind a' rule:edge-kind-cases)
  case Intra
  with ‹upd-cs cs (a'#as') = c'#cs'›
  have upd-cs cs as' = c'#cs' by(fastforce simp:intra-kind-def)
  from IH[OF this] have upd-cs cs (as'@[a]) = cs' .
  with Intra show ?thesis by(fastforce simp:intra-kind-def)
 next
  case Call
  with ‹upd-cs cs (a'#as') = c'#cs'›
  have upd-cs (a'#cs) as' = c'#cs' by simp
  from IH[OF this] have upd-cs (a'#cs) (as'@[a]) = cs' .
  with Call show ?thesis by simp
 next
  case Return
  show ?thesis
  proof(cases cs)
   case Nil
   with ‹upd-cs cs (a'#as') = c'#cs'› Return
   have upd-cs cs as' = c'#cs' by simp
   from IH[OF this] have upd-cs cs (as'@[a]) = cs' .
   with Nil Return show ?thesis by simp
  next
   case (Cons cx csx)
   with ‹upd-cs cs (a'#as') = c'#cs'› Return
```

```
    have upd-cs csx as' = c'#cs' by simp
    from IH[OF this] have upd-cs csx (as'@[a]) = cs' .
    with Cons Return show ?thesis by simp
   qed
  qed
qed


lemma upd-cs-snoc-Call:
  assumes kind a = Q:r↪pfs
  shows upd-cs cs (as@[a]) = a#(upd-cs cs as)
proof(induct as arbitrary:cs)
  case Nil
  with ‹kind a = Q:r↪pfs› show ?case by simp
next
  case (Cons a' as')
  note IH = ‹⋀cs. upd-cs cs (as'@[a]) = a#upd-cs cs as'›
  show ?case
  proof(cases kind a' rule:edge-kind-cases)
    case Intra
    with IH[of cs] show ?thesis by(fastforce simp:intra-kind-def)
  next
    case Call
    with IH[of a'#cs] show ?thesis by simp
  next
    case Return
    show ?thesis
    proof(cases cs)
      case Nil
      with IH[of []] Return show ?thesis by simp
    next
      case (Cons cx csx)
      with IH[of csx] Return show ?thesis by simp
    qed
  qed
qed




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'
  using ‹valid-path-aux cs (as@as')›
proof(induct cs as@as' arbitrary:as as' rule:vpa-induct)
  case (vpa-intra cs a as as'')
  note IH1 = ‹⋀xs ys. as = xs@ys ⟹ valid-path-aux cs xs›
  note IH2 = ‹⋀xs ys. as = xs@ys ⟹ valid-path-aux (upd-cs cs xs) ys›
```

```
  { case 1
    from vpa-intra
    have as'' = [] ∧ a#as = as' ∨ (∃xs. a#xs = as'' ∧ as = xs@as')
      by(simp add:Cons-eq-append-conv)
    thus ?case
    proof
      assume as'' = [] ∧ a#as = as'
      thus ?thesis by simp
    next
      assume ∃xs. a#xs = as'' ∧ as = xs@as'
      then obtain xs where a#xs = as'' and as = xs@as' by auto
      from IH1[OF ‹as = xs@as'›] have valid-path-aux cs xs .
      with ‹a#xs = as''› ‹intra-kind (kind a)›
      show ?thesis by(fastforce simp:intra-kind-def)
    qed
  next
    case 2
    from vpa-intra
    have as'' = [] ∧ a#as = as' ∨ (∃xs. a#xs = as'' ∧ as = xs@as')
      by(simp add:Cons-eq-append-conv)
    thus ?case
    proof
      assume as'' = [] ∧ a#as = as'
      hence as = []@tl as' by(cases as') auto
      from IH2[OF this] have valid-path-aux (upd-cs cs []) (tl as') by simp
      with ‹as'' = [] ∧ a#as = as'› ‹intra-kind (kind a)›
      show ?thesis by(fastforce simp:intra-kind-def)
    next
      assume ∃xs. a#xs = as'' ∧ as = xs@as'
      then obtain xs where a#xs = as'' and as = xs@as' by auto
      from IH2[OF ‹as = xs@as'›] have valid-path-aux (upd-cs cs xs) as' .
      from ‹a#xs = as''› ‹intra-kind (kind a)›
      have upd-cs cs xs = upd-cs cs as'' by(fastforce simp:intra-kind-def)
      with ‹valid-path-aux (upd-cs cs xs) as'›
      show ?thesis by simp
    qed
  }
next
  case (vpa-Call cs a as Q r p fs as'')
  note IH1 = ‹⋀xs ys. as = xs@ys ⟹ valid-path-aux (a#cs) xs›
  note IH2 = ‹⋀xs ys. as = xs@ys ⟹  valid-path-aux (upd-cs (a#cs) xs) ys›
  { case 1
    from vpa-Call
    have as'' = [] ∧ a#as = as' ∨ (∃xs. a#xs = as'' ∧ as = xs@as')
      by(simp add:Cons-eq-append-conv)
    thus ?case
    proof
      assume as'' = [] ∧ a#as = as'
      thus ?thesis by simp
```

```
    next
      assume ∃xs. a#xs = as'' ∧ as = xs@as'
      then obtain xs where a#xs = as'' and as = xs@as' by auto
      from IH1[OF ‹as = xs@as'›] have valid-path-aux (a#cs) xs .
      with ‹a#xs = as''›[THEN sym] ‹kind a = Q:r↪pfs›
      show ?thesis by simp
    qed
  next
    case 2
    from vpa-Call
    have as'' = [] ∧ a#as = as' ∨ (∃xs. a#xs = as'' ∧ as = xs@as')
      by(simp add:Cons-eq-append-conv)
    thus ?case
    proof
      assume as'' = [] ∧ a#as = as'
      hence as = []@tl as' by(cases as') auto
      from IH2[OF this] have valid-path-aux (upd-cs (a#cs) []) (tl as') .
      with ‹as'' = [] ∧ a#as = as'› ‹kind a = Q:r↪pfs›
      show ?thesis by clarsimp
    next
      assume ∃xs. a#xs = as'' ∧ as = xs@as'
      then obtain xs where a#xs = as'' and as = xs@as' by auto
      from IH2[OF ‹as = xs@as'›] have valid-path-aux (upd-cs (a # cs) xs) as' .
      with ‹a#xs = as''›[THEN sym] ‹kind a = Q:r↪pfs›
      show ?thesis by simp
    qed
  }
next
  case (vpa-ReturnEmpty cs a as Q p f as'')
  note IH1 = ‹⋀xs ys. as = xs@ys ⟹ valid-path-aux [] xs›
  note IH2 = ‹⋀xs ys. as = xs@ys ⟹ valid-path-aux (upd-cs [] xs) ys›
  { case 1
    from vpa-ReturnEmpty
    have as'' = [] ∧ a#as = as' ∨ (∃xs. a#xs = as'' ∧ as = xs@as')
      by(simp add:Cons-eq-append-conv)
    thus ?case
    proof
      assume as'' = [] ∧ a#as = as'
      thus ?thesis by simp
    next
      assume ∃xs. a#xs = as'' ∧ as = xs@as'
      then obtain xs where a#xs = as'' and as = xs@as' by auto
      from IH1[OF ‹as = xs@as'›] have valid-path-aux [] xs .
      with ‹a#xs = as''›[THEN sym] ‹kind a = Q↩pf› ‹cs = []›
      show ?thesis by simp
    qed
  next
    case 2
    from vpa-ReturnEmpty
```

```
    have as'' = [] ∧ a#as = as' ∨ (∃ xs. a#xs = as'' ∧ as = xs@as')
      by(simp add:Cons-eq-append-conv)
    thus ?case
    proof
      assume as'' = [] ∧ a#as = as'
      hence as = []@tl as' by(cases as') auto
      from IH2[OF this] have valid-path-aux [] (tl as') by simp
      with ⟨as'' = [] ∧ a#as = as'⟩ ⟨kind a = Q↩pf⟩ ⟨cs = []⟩
      show ?thesis by fastforce
    next
      assume ∃ xs. a#xs = as'' ∧ as = xs@as'
      then obtain xs where a#xs = as'' and as = xs@as' by auto
      from IH2[OF ⟨as = xs@as'⟩] have valid-path-aux (upd-cs [] xs) as' .
      from ⟨a#xs = as''⟩[THEN sym] ⟨kind a = Q↩pf⟩ ⟨cs = []⟩
      have upd-cs [] xs = upd-cs cs as'' by simp
      with ⟨valid-path-aux (upd-cs [] xs) as'⟩ show ?thesis by simp
    qed
  }
next
  case (vpa-ReturnCons cs a as Q p f c' cs' as'')
  note IH1 = ⟨⋀xs ys. as = xs@ys ⟹ valid-path-aux cs' xs⟩
  note IH2 = ⟨⋀xs ys. as = xs@ys ⟹ valid-path-aux (upd-cs cs' xs) ys⟩
  { case 1
    from vpa-ReturnCons
    have as'' = [] ∧ a#as = as' ∨ (∃ xs. a#xs = as'' ∧ as = xs@as')
      by(simp add:Cons-eq-append-conv)
    thus ?case
    proof
      assume as'' = [] ∧ a#as = as'
      thus ?thesis by simp
    next
      assume ∃ xs. a#xs = as'' ∧ as = xs@as'
      then obtain xs where a#xs = as'' and as = xs@as' by auto
      from IH1[OF ⟨as = xs@as'⟩] have valid-path-aux cs' xs .
      with ⟨a#xs = as''⟩[THEN sym] ⟨kind a = Q↩pf⟩ ⟨cs = c'#cs'⟩
        ⟨a ∈ get-return-edges c'⟩
      show ?thesis by simp
    qed
  next
    case 2
    from vpa-ReturnCons
    have as'' = [] ∧ a#as = as' ∨ (∃ xs. a#xs = as'' ∧ as = xs@as')
      by(simp add:Cons-eq-append-conv)
    thus ?case
    proof
      assume as'' = [] ∧ a#as = as'
      hence as = []@tl as' by(cases as') auto
      from IH2[OF this] have valid-path-aux (upd-cs cs' []) (tl as') .
      with ⟨as'' = [] ∧ a#as = as'⟩ ⟨kind a = Q↩pf⟩ ⟨cs = c'#cs'⟩
```

```
      ‹a ∈ get-return-edges c'›
    show ?thesis by fastforce
  next
    assume ∃xs. a#xs = as'' ∧ as = xs@as'
    then obtain xs where a#xs = as'' and as = xs@as' by auto
    from IH2[OF ‹as = xs@as'›] have valid-path-aux (upd-cs cs' xs) as' .
    from ‹a#xs = as''›[THEN sym] ‹kind a = Q↩pf› ‹cs = c'#cs'›
    have upd-cs cs' xs = upd-cs cs as'' by simp
    with ‹valid-path-aux (upd-cs cs' xs) as'› show ?thesis by simp
  qed
 }
qed simp-all


lemma valid-path-aux-Append:
 ⟦valid-path-aux cs as; valid-path-aux (upd-cs cs as) as'⟧
 ⟹ valid-path-aux cs (as@as')
by(induct rule:vpa-induct,auto simp:intra-kind-def)


lemma vpa-snoc-Call:
 assumes kind a = Q:r↪pfs
 shows valid-path-aux cs as ⟹ valid-path-aux cs (as@[a])
proof(induct rule:vpa-induct)
 case (vpa-empty cs)
 from ‹kind a = Q:r↪pfs› have valid-path-aux cs [a] by simp
 thus ?case by simp
next
 case (vpa-intra cs a' as')
 from ‹valid-path-aux cs (as'@[a])› ‹intra-kind (kind a')›
 have valid-path-aux cs (a'#(as'@[a]))
  by(fastforce simp:intra-kind-def)
 thus ?case by simp
next
 case (vpa-Call cs a' as' Q' r' p' fs')
 from ‹valid-path-aux (a'#cs) (as'@[a])› ‹kind a' = Q':r'↪p'fs'›
 have valid-path-aux cs (a'#(as'@[a])) by simp
 thus ?case by simp
next
 case (vpa-ReturnEmpty cs a' as' Q' p' f')
 from ‹valid-path-aux [] (as'@[a])› ‹kind a' = Q'↩p'f'› ‹cs = []›
 have valid-path-aux cs (a'#(as'@[a])) by simp
 thus ?case by simp
next
 case (vpa-ReturnCons cs a' as' Q' p' f' c' cs')
 from ‹valid-path-aux cs' (as'@[a])› ‹kind a' = Q'↩p'f'› ‹cs = c'#cs'›
   ‹a' ∈ get-return-edges c'›
 have valid-path-aux cs (a'#(as'@[a])) by simp
 thus ?case by simp
```

qed

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*
by(*fastforce intro*:*valid-path-aux-callstack-prefix simp*:*valid-path-def*)

lemma *valid-path-split*:
  assumes *valid-path* (*as*@*as'*) **shows** *valid-path as* **and** *valid-path as'*
  using ‹*valid-path* (*as*@*as'*)›
  apply(*auto simp*:*valid-path-def*)
   apply(*erule valid-path-aux-split*)
  apply(*drule valid-path-aux-split*(*2*))
  by(*fastforce intro*:*valid-path-aux-callstack-prefix*)

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 −
  from ‹*n* −*as*→$_{\iota}$* *n'*› **have** *n* −*as*→* *n'* **and** ∀ *a* ∈ *set as*. *intra-kind*(*kind a*)
    by(*simp-all add*:*intra-path-def*)
  from ‹∀ *a* ∈ *set as*. *intra-kind*(*kind a*)› **have** *valid-path-aux* [] *as*
    by(*rule valid-path-aux-intra-path*)
  thus *?thesis* **using** ‹*n* −*as*→* *n'*› **by**(*simp add*:*vp-def valid-path-def*)
qed

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(*atomize-elim*)
  from ‹*n* −*as*→$_{\surd}$* *n'*› ‹*as* ≠ []› **obtain** *a' as'* **where** *as* = *a'*#*as'*
    and *n* = *sourcenode a'* **and** *valid-edge a'* **and** *targetnode a'* −*as'*→* *n'*
    by(*fastforce elim*:*path-split-Cons simp*:*vp-def*)
  from ‹*n* −*as*→$_{\surd}$* *n'*› **have** *valid-path as* **by**(*simp add*:*vp-def*)
  from ‹*as* = *a'*#*as'*› **have** *as* = [*a'*]@*as'* **by** *simp*
  with ‹*valid-path as*› **have** *valid-path* ([*a'*]@*as'*) **by** *simp*

hence *valid-path as'* **by**(*rule valid-path-split*)
with ‹*targetnode a' −as'→* n'*› **have** *targetnode a' −as'→$_{\surd}$* n'* **by**(*simp add*:*vp-def*)
with ‹*as = a'#as'*› ‹*n = sourcenode a'*› ‹*valid-edge a'*›
show ∃ *a' as'*. *as = a'#as' ∧ n = sourcenode a' ∧ valid-edge a' ∧*
targetnode a' −as'→$_{\surd}$ n'* **by** *blast*
qed

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(*atomize-elim*)
from ‹*n −as→$_{\surd}$* n'*› ‹*as ≠ []*› **obtain** *a' as'* **where** *as = as'@[a']*
and *n −as'→* sourcenode a'* **and** *valid-edge a'* **and** *n' = targetnode a'*
by(*clarsimp simp*:*vp-def*)(*erule path-split-snoc*,*auto*)
from ‹*n −as→$_{\surd}$* n'*› ‹*as = as'@[a']*› **have** *valid-path (as'@[a'])* **by**(*simp add*:*vp-def*)
hence *valid-path as'* **by**(*rule valid-path-split*)
with ‹*n −as'→* sourcenode a'*› **have** *n −as'→$_{\surd}$* sourcenode a'* **by**(*simp add*:*vp-def*)
with ‹*as = as'@[a']*› ‹*valid-edge a'*› ‹*n' = targetnode a'*›
show ∃ *as' a'*. *as = as'@[a'] ∧ n −as'→$_{\surd}$* sourcenode a' ∧ valid-edge a' ∧*
n' = targetnode a'
by *blast*
qed

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 −
from ‹*n −as@a#as'→$_{\surd}$* n'*› **have** *n −as→* sourcenode a* **and** *valid-edge a*
and *targetnode a −as'→* n'*
by(*auto intro*:*path-split simp*:*vp-def*)
from ‹*n −as@a#as'→$_{\surd}$* n'*› **have** *valid-path (as@a#as')* **by**(*simp add*:*vp-def*)
hence *valid-path as* **and** *valid-path (a#as')* **by**(*auto intro*:*valid-path-split*)
from ‹*valid-path (a#as')*› **have** *valid-path ([a]@as')* **by** *simp*
hence *valid-path as'* **by**(*rule valid-path-split*)
with ‹*n −as→* sourcenode a*› ‹*valid-path as*› ‹*valid-edge a*› ‹*targetnode a −as'→**
n'›
show *n −as→$_{\surd}$* sourcenode a valid-edge a targetnode a −as'→$_{\surd}$* n'*
by(*auto simp*:*vp-def*)
qed

lemma *vp-split-second*:
assumes *n −as@a#as'→$_{\surd}$* n'* **shows** *sourcenode a −a#as'→$_{\surd}$* n'*
proof −
from ‹*n −as@a#as'→$_{\surd}$* n'*› **have** *sourcenode a −a#as'→* n'*
by(*fastforce elim*:*path-split-second simp*:*vp-def*)
from ‹*n −as@a#as'→$_{\surd}$* n'*› **have** *valid-path (as@a#as')* **by**(*simp add*:*vp-def*)
hence *valid-path (a#as')* **by**(*rule valid-path-split*)
with ‹*sourcenode a −a#as'→* n'*› **show** *?thesis* **by**(*simp add*:*vp-def*)

qed

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*)
by *auto*(*case-tac b rule*:*rev-cases*,*auto*)
termination by *lexicographic-order*

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*: ⋀*cs*. *P cs* []
⋀*cs a as*. ⟦*intra-kind*(*kind a*); *valid-path-rev-aux cs as*; *P cs as*⟧
⟹ *P cs* (*as*@[*a*])
⋀*cs a as Q p f*. ⟦*kind a* = *Q*↩$_p$*f*; *valid-path-rev-aux* (*a*#*cs*) *as*; *P* (*a*#*cs*) *as*⟧
⟹ *P cs* (*as*@[*a*])
⋀*cs a as Q r p fs*. ⟦*kind a* = *Q*:*r*↪$_p$*fs*; *cs* = []; *valid-path-rev-aux* [] *as*; *P* [] *as*⟧ ⟹ *P cs* (*as*@[*a*])
⋀*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*⟧
⟹ *P cs* (*as*@[*a*])
shows *P xs ys*
using *major*
apply(*induct ys arbitrary*:*xs rule*:*rev-induct*)
by(*auto intro*:*rules split*:*edge-kind.split-asm list.split-asm simp*:*intra-kind-def*)

lemma *vpra-callstack-prefix*:
valid-path-rev-aux (*cs*@*cs'*) *as* ⟹ *valid-path-rev-aux cs as*
proof(*induct cs*@*cs' as arbitrary*:*cs cs' rule*:*vpra-induct*)
case *vpra-empty* **thus** *?case* **by** *simp*
next
case (*vpra-intra a as*)
hence *valid-path-rev-aux cs as* **by** *simp*
with ‹*intra-kind* (*kind a*)› **show** *?case* **by**(*fastforce simp*:*intra-kind-def*)
next
case (*vpra-Return a as Q p f*)
note *IH* = ‹⋀*ds ds'*. *a*#*cs*@*cs'* = *ds*@*ds'* ⟹ *valid-path-rev-aux ds as*›

```
  have a#cs@cs' = (a#cs)@cs' by simp
  from IH[OF this] have valid-path-rev-aux (a#cs) as .
  with ‹kind a = Q↩pf› show ?case by simp
next
  case (vpra-CallEmpty a as Q r p fs)
  hence valid-path-rev-aux cs as by simp
  with ‹kind a = Q:r↪pfs› ‹cs@cs' = []› show ?case by simp
next
  case (vpra-CallCons a as Q r p fs c' csx)
  note IH = ‹⋀cs cs'. csx = cs@cs' ⟹ valid-path-rev-aux cs as›
  from ‹cs@cs' = c'#csx›
  have (cs = [] ∧ cs' = c'#csx) ∨ (∃zs. cs = c'#zs ∧ zs@cs' = csx)
    by(simp add:append-eq-Cons-conv)
  thus ?case
  proof
    assume cs = [] ∧ cs' = c'#csx
    hence cs = [] and cs' = c'#csx by simp-all
    from ‹cs' = c'#csx› have csx = []@tl cs' by simp
    from IH[OF this] have valid-path-rev-aux [] as .
    with ‹cs = []› ‹kind a = Q:r↪pfs› show ?thesis by simp
  next
    assume ∃zs. cs = c'#zs ∧ zs@cs' = csx
    then obtain zs where cs = c'#zs and csx = zs@cs' by auto
    from IH[OF ‹csx = zs@cs'›] have valid-path-rev-aux zs as .
    with ‹cs = c'#zs› ‹kind a = Q:r↪pfs› ‹c' ∈ get-return-edges a› show ?thesis
by simp
  qed
qed



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↩pf ⇒ upd-rev-cs (a#cs) as
                      | Q:r↪pfs ⇒ case cs of [] ⇒ upd-rev-cs cs as
                                          | c'#cs' ⇒ upd-rev-cs cs' as
                      |    - ⇒ upd-rev-cs cs as)
by auto(case-tac b rule:rev-cases,auto)
termination by lexicographic-order


lemma upd-rev-cs-empty [dest]:
  upd-rev-cs cs [] = [] ⟹ cs = []
by(cases cs) auto


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
  using ‹valid-path-rev-aux cs (as@as')›
proof(induct cs as@as' arbitrary:as as' rule:vpra-induct)
  case (vpra-intra cs a as as'')
  note IH1 = ‹⋀xs ys. as = xs@ys ⟹ valid-path-rev-aux cs ys›
  note IH2 = ‹⋀xs ys. as = xs@ys ⟹ valid-path-rev-aux (upd-rev-cs cs ys) xs›
  { case 1
    from vpra-intra
    have as' = [] ∧ as@[a] = as'' ∨ (∃xs. as = as''@xs ∧ xs@[a] = as')
      by(cases as' rule:rev-cases) auto
    thus ?case
    proof
      assume as' = [] ∧ as@[a] = as''
      thus ?thesis by simp
    next
      assume ∃xs. as = as''@xs ∧ xs@[a] = as'
      then obtain xs where as = as''@xs and xs@[a] = as' by auto
      from IH1[OF ‹as = as''@xs›] have valid-path-rev-aux cs xs .
      with ‹xs@[a] = as'› ‹intra-kind (kind a)›
      show ?thesis by(fastforce simp:intra-kind-def)
    qed
  next
    case 2
    from vpra-intra
    have as' = [] ∧ as@[a] = as'' ∨ (∃xs. as = as''@xs ∧ xs@[a] = as')
      by(cases as' rule:rev-cases) auto
    thus ?case
    proof
      assume as' = [] ∧ as@[a] = as''
      hence as = butlast as''@[] by(cases as) auto
      from IH2[OF this] have valid-path-rev-aux (upd-rev-cs cs []) (butlast as'') .
      with ‹as' = [] ∧ as@[a] = as''› ‹intra-kind (kind a)›
      show ?thesis by(fastforce simp:intra-kind-def)
    next
      assume ∃xs. as = as''@xs ∧ xs@[a] = as'
      then obtain xs where as = as''@xs and xs@[a] = as' by auto
      from IH2[OF ‹as = as''@xs›] have valid-path-rev-aux (upd-rev-cs cs xs) as''
.
      from ‹xs@[a] = as'› ‹intra-kind (kind a)›
      have upd-rev-cs cs xs = upd-rev-cs cs as' by(fastforce simp:intra-kind-def)
      with ‹valid-path-rev-aux (upd-rev-cs cs xs) as''›
      show ?thesis by simp
    qed
  }
next
  case (vpra-Return cs a as Q p f as'')
  note IH1 = ‹⋀xs ys. as = xs@ys ⟹ valid-path-rev-aux (a#cs) ys›
  note IH2 = ‹⋀xs ys. as = xs@ys ⟹ valid-path-rev-aux (upd-rev-cs (a#cs) ys)
xs›
```

```
  { case 1
    from vpra-Return
    have as' = [] ∧ as@[a] = as'' ∨ (∃xs. as = as''@xs ∧ xs@[a] = as')
      by(cases as' rule:rev-cases) auto
    thus ?case
    proof
      assume as' = [] ∧ as@[a] = as''
      thus ?thesis by simp
    next
      assume ∃xs. as = as''@xs ∧ xs@[a] = as'
      then obtain xs where as = as''@xs and xs@[a] = as' by auto
      from IH1[OF ‹as = as''@xs›] have valid-path-rev-aux (a#cs) xs .
      with ‹xs@[a] = as'› ‹kind a = Q↩pf›
      show ?thesis by fastforce
    qed
  next
    case 2
    from vpra-Return
    have as' = [] ∧ as@[a] = as'' ∨ (∃xs. as = as''@xs ∧ xs@[a] = as')
      by(cases as' rule:rev-cases) auto
    thus ?case
    proof
      assume as' = [] ∧ as@[a] = as''
      hence as = butlast as''@[] by(cases as) auto
      from IH2[OF this]
      have valid-path-rev-aux (upd-rev-cs (a#cs) []) (butlast as'') .
      with ‹as' = [] ∧ as@[a] = as''› ‹kind a = Q↩pf›
      show ?thesis by fastforce
    next
      assume ∃xs. as = as''@xs ∧ xs@[a] = as'
      then obtain xs where as = as''@xs and xs@[a] = as' by auto
      from IH2[OF ‹as = as''@xs›]
      have valid-path-rev-aux (upd-rev-cs (a#cs) xs) as'' .
      from ‹xs@[a] = as'› ‹kind a = Q↩pf›
      have upd-rev-cs (a#cs) xs = upd-rev-cs cs as' by fastforce
      with ‹valid-path-rev-aux (upd-rev-cs (a#cs) xs) as''›
      show ?thesis by simp
    qed
  }
next
  case (vpra-CallEmpty cs a as Q r p fs as'')
  note IH1 = ‹⋀xs ys. as = xs@ys ⟹ valid-path-rev-aux [] ys›
  note IH2 = ‹⋀xs ys. as = xs@ys ⟹ valid-path-rev-aux (upd-rev-cs [] ys) xs›
  { case 1
    from vpra-CallEmpty
    have as' = [] ∧ as@[a] = as'' ∨ (∃xs. as = as''@xs ∧ xs@[a] = as')
      by(cases as' rule:rev-cases) auto
    thus ?case
    proof
```

```
      assume as' = [] ∧ as@[a] = as''
      thus ?thesis by simp
    next
      assume ∃xs. as = as''@xs ∧ xs@[a] = as'
      then obtain xs where as = as''@xs and xs@[a] = as' by auto
      from IH1[OF ⟨as = as''@xs⟩] have valid-path-rev-aux [] xs .
      with ⟨xs@[a] = as'⟩ ⟨kind a = Q:r↪pfs⟩ ⟨cs = []⟩
      show ?thesis by fastforce
    qed
  next
    case 2
    from vpra-CallEmpty
    have as' = [] ∧ as@[a] = as'' ∨ (∃xs. as = as''@xs ∧ xs@[a] = as')
      by(cases as' rule:rev-cases) auto
    thus ?case
    proof
      assume as' = [] ∧ as@[a] = as''
      hence as = butlast as''@[] by(cases as) auto
      from IH2[OF this]
      have valid-path-rev-aux (upd-rev-cs [] []) (butlast as'') .
      with ⟨as' = [] ∧ as@[a] = as''⟩ ⟨kind a = Q:r↪pfs⟩ ⟨cs = []⟩
      show ?thesis by fastforce
    next
      assume ∃xs. as = as''@xs ∧ xs@[a] = as'
      then obtain xs where as = as''@xs and xs@[a] = as' by auto
      from IH2[OF ⟨as = as''@xs⟩]
      have valid-path-rev-aux (upd-rev-cs [] xs) as'' .
      with ⟨xs@[a] = as'⟩ ⟨kind a = Q:r↪pfs⟩ ⟨cs = []⟩
      show ?thesis by fastforce
    qed
  }
next
  case (vpra-CallCons cs a as Q r p fs c' cs' as'')
  note IH1 = ⟨⋀xs ys. as = xs@ys ⟹ valid-path-rev-aux cs' ys⟩
  note IH2 = ⟨⋀xs ys. as = xs@ys ⟹ valid-path-rev-aux (upd-rev-cs cs' ys) xs⟩
  { case 1
    from vpra-CallCons
    have as' = [] ∧ as@[a] = as'' ∨ (∃xs. as = as''@xs ∧ xs@[a] = as')
      by(cases as' rule:rev-cases) auto
    thus ?case
    proof
      assume as' = [] ∧ as@[a] = as''
      thus ?thesis by simp
    next
      assume ∃xs. as = as''@xs ∧ xs@[a] = as'
      then obtain xs where as = as''@xs and xs@[a] = as' by auto
      from IH1[OF ⟨as = as''@xs⟩] have valid-path-rev-aux cs' xs .
     with ⟨xs@[a] = as'⟩ ⟨kind a = Q:r↪pfs⟩ ⟨cs = c' # cs'⟩ ⟨c' ∈ get-return-edges
a⟩
```

```
    show ?thesis by fastforce
  qed
 next
  case 2
  from vpra-CallCons
  have as' = [] ∧ as@[a] = as'' ∨ (∃xs. as = as''@xs ∧ xs@[a] = as')
   by(cases as' rule:rev-cases) auto
  thus ?case
  proof
   assume as' = [] ∧ as@[a] = as''
   hence as = butlast as''@[] by(cases as) auto
   from IH2[OF this]
   have valid-path-rev-aux (upd-rev-cs cs' []) (butlast as'') .
   with ‹as' = [] ∧ as@[a] = as''› ‹kind a = Q:r↪pfs› ‹cs = c' # cs'›
    ‹c' ∈ get-return-edges a› show ?thesis by fastforce
  next
   assume ∃xs. as = as''@xs ∧ xs@[a] = as'
   then obtain xs where as = as''@xs and xs@[a] = as' by auto
   from IH2[OF ‹as = as''@xs›]
   have valid-path-rev-aux (upd-rev-cs cs' xs) as'' .
   with ‹xs@[a] = as'› ‹kind a = Q:r↪pfs› ‹cs = c' # cs'›
    ‹c' ∈ get-return-edges a›
   show ?thesis by fastforce
  qed
 }
qed simp-all


lemma valid-path-rev-aux-Append:
 ⟦valid-path-rev-aux cs as'; valid-path-rev-aux (upd-rev-cs cs as') as⟧
 ⟹ valid-path-rev-aux cs (as@as')
by(induct rule:vpra-induct,
   auto simp:intra-kind-def simp del:append-assoc simp:append-assoc[THEN sym])


lemma vpra-Cons-intra:
 assumes intra-kind(kind a)
 shows valid-path-rev-aux cs as ⟹ valid-path-rev-aux cs (a#as)
proof(induct rule:vpra-induct)
 case (vpra-empty cs)
 have valid-path-rev-aux cs [] by simp
 with ‹intra-kind(kind a)› have valid-path-rev-aux cs ([]@[a])
  by(simp only:valid-path-rev-aux.simps intra-kind-def,fastforce)
 thus ?case by simp
qed(simp only:append-Cons[THEN sym] valid-path-rev-aux.simps intra-kind-def,fastforce)+


lemma vpra-Cons-Return:
 assumes kind a = Q↩pf
```

shows *valid-path-rev-aux cs as* $\Longrightarrow$ *valid-path-rev-aux cs* (*a*#*as*)
proof(*induct rule*:*vpra-induct*)
case (*vpra-empty cs*)
from ‹*kind a* = *Q*↩$_p$*f*› **have** *valid-path-rev-aux cs* ([]@[*a*])
by(*simp only*:*valid-path-rev-aux.simps*,*clarsimp*)
thus *?case* **by** *simp*
next
case (*vpra-intra cs a*′ *as*′)
from ‹*valid-path-rev-aux cs* (*a*#*as*′)› ‹*intra-kind* (*kind a*′)›
have *valid-path-rev-aux cs* ((*a*#*as*′)@[*a*′])
by(*simp only*:*valid-path-rev-aux.simps*,*fastforce simp*:*intra-kind-def*)
thus *?case* **by** *simp*
next
case (*vpra-Return cs a*′ *as*′ *Q*′ *p*′ *f*′)
from ‹*valid-path-rev-aux* (*a*′#*cs*) (*a*#*as*′)› ‹*kind a*′ = *Q*′↩$_{p'}$*f*′›
have *valid-path-rev-aux cs* ((*a*#*as*′)@[*a*′])
by(*simp only*:*valid-path-rev-aux.simps*,*clarsimp*)
thus *?case* **by** *simp*
next
case (*vpra-CallEmpty cs a*′ *as*′ *Q*′ *r*′ *p*′ *fs*′)
from ‹*valid-path-rev-aux* [] (*a*#*as*′)› ‹*kind a*′ = *Q*′:*r*′↪$_{p'}$*fs*′› ‹*cs* = []›
have *valid-path-rev-aux cs* ((*a*#*as*′)@[*a*′])
by(*simp only*:*valid-path-rev-aux.simps*,*clarsimp*)
thus *?case* **by** *simp*
next
case (*vpra-CallCons cs a*′ *as*′ *Q*′ *r*′ *p*′ *fs*′ *c*′ *cs*′)
from ‹*valid-path-rev-aux cs*′ (*a*#*as*′)› ‹*kind a*′ = *Q*′:*r*′↪$_{p'}$*fs*′› ‹*cs* = *c*′#*cs*′›
‹*c*′ ∈ *get-return-edges a*′›
have *valid-path-rev-aux cs* ((*a*#*as*′)@[*a*′])
by(*simp only*:*valid-path-rev-aux.simps*,*clarsimp*)
thus *?case* **by** *simp*
qed

lemma *upd-rev-cs-Cons-intra*:
assumes *intra-kind*(*kind a*) **shows** *upd-rev-cs cs* (*a*#*as*) = *upd-rev-cs cs as*
proof(*induct as arbitrary*:*cs rule*:*rev-induct*)
case *Nil*
from ‹*intra-kind* (*kind a*)›
have *upd-rev-cs cs* ([]@[*a*]) = *upd-rev-cs cs* []
by(*simp only*:*upd-rev-cs.simps*,*auto simp*:*intra-kind-def*)
thus *?case* **by** *simp*
next
case (*snoc a*′ *as*′)
note *IH* = ‹⋀*cs*. *upd-rev-cs cs* (*a*#*as*′) = *upd-rev-cs cs as*′›
show *?case*
proof(*cases kind a*′ *rule*:*edge-kind-cases*)
case *Intra*
from *IH* **have** *upd-rev-cs cs* (*a*#*as*′) = *upd-rev-cs cs as*′ **.**

```
    with Intra have upd-rev-cs cs ((a#as')@[a']) = upd-rev-cs cs (as'@[a'])
      by(fastforce simp:intra-kind-def)
    thus ?thesis by simp
  next
    case Return
    from IH have upd-rev-cs (a'#cs) (a#as') = upd-rev-cs (a'#cs) as' .
    with Return have upd-rev-cs cs ((a#as')@[a']) = upd-rev-cs cs (as'@[a'])
      by(auto simp:intra-kind-def)
    thus ?thesis by simp
  next
    case Call
    show ?thesis
    proof(cases cs)
      case Nil
      from IH have upd-rev-cs [] (a#as') = upd-rev-cs [] as' .
      with Call Nil have upd-rev-cs cs ((a#as')@[a']) = upd-rev-cs cs (as'@[a'])
        by(auto simp:intra-kind-def)
      thus ?thesis by simp
    next
      case (Cons c' cs')
      from IH have upd-rev-cs cs' (a#as') = upd-rev-cs cs' as' .
      with Call Cons have upd-rev-cs cs ((a#as')@[a']) = upd-rev-cs cs (as'@[a'])
        by(auto simp:intra-kind-def)
      thus ?thesis by simp
    qed
  qed
qed



lemma upd-rev-cs-Cons-Return:
  assumes kind a = Q↩pf shows upd-rev-cs cs (a#as) = a#(upd-rev-cs cs as)
proof(induct as arbitrary:cs rule:rev-induct)
  case Nil
  with ‹kind a = Q↩pf› have upd-rev-cs cs ([]@[a]) = a#(upd-rev-cs cs [])
    by(simp only:upd-rev-cs.simps) clarsimp
  thus ?case by simp
next
  case (snoc a' as')
  note IH = ‹⋀cs. upd-rev-cs cs (a#as') = a#upd-rev-cs cs as'›
  show ?case
  proof(cases kind a' rule:edge-kind-cases)
    case Intra
    from IH have upd-rev-cs cs (a#as') = a#(upd-rev-cs cs as') .
    with Intra have upd-rev-cs cs ((a#as')@[a']) = a#(upd-rev-cs cs (as'@[a']))
      by(fastforce simp:intra-kind-def)
    thus ?thesis by simp
  next
    case Return
```

```
    from IH have upd-rev-cs (a'#cs) (a#as') = a#(upd-rev-cs (a'#cs) as') .
   with Return have upd-rev-cs cs ((a#as')@[a']) = a#(upd-rev-cs cs (as'@[a']))
     by(auto simp:intra-kind-def)
    thus ?thesis by simp
  next
    case Call
    show ?thesis
    proof(cases cs)
      case Nil
      from IH have upd-rev-cs [] (a#as') = a#(upd-rev-cs [] as') .
     with Call Nil have upd-rev-cs cs ((a#as')@[a']) = a#(upd-rev-cs cs (as'@[a']))
       by(auto simp:intra-kind-def)
      thus ?thesis by simp
    next
      case (Cons c' cs')
      from IH have upd-rev-cs cs' (a#as') = a#(upd-rev-cs cs' as') .
      with Call Cons
      have upd-rev-cs cs ((a#as')@[a']) = a#(upd-rev-cs cs (as'@[a']))
        by(auto simp:intra-kind-def)
      thus ?thesis by simp
    qed
  qed
qed


lemma upd-rev-cs-Cons-Call-Cons:
  assumes kind a = Q:r↪pfs
  shows upd-rev-cs cs as = c'#cs' ⟹ upd-rev-cs cs (a#as) = cs'
proof(induct as arbitrary:cs rule:rev-induct)
  case Nil
  with ‹kind a = Q:r↪pfs› have upd-rev-cs cs ([]@[a]) = cs'
    by(simp only:upd-rev-cs.simps) clarsimp
 thus ?case by simp
next
  case (snoc a' as')
  note IH = ‹⋀cs. upd-rev-cs cs as' = c'#cs' ⟹ upd-rev-cs cs (a#as') = cs'›
  show ?case
  proof(cases kind a' rule:edge-kind-cases)
    case Intra
    with ‹upd-rev-cs cs (as'@[a']) = c'#cs'›
    have upd-rev-cs cs as' = c'#cs' by(fastforce simp:intra-kind-def)
    from IH[OF this] have upd-rev-cs cs (a#as') = cs' .
    with Intra show ?thesis by(fastforce simp:intra-kind-def)
  next
    case Return
    with ‹upd-rev-cs cs (as'@[a']) = c'#cs'›
    have upd-rev-cs (a'#cs) as' = c'#cs' by simp
    from IH[OF this] have upd-rev-cs (a'#cs) (a#as') = cs' .
    with Return show ?thesis by simp
```

```
  next
    case Call
    show ?thesis
    proof(cases cs)
      case Nil
      with ‹upd-rev-cs cs (as'@[a']) = c'#cs'› Call
      have upd-rev-cs cs as' = c'#cs' by simp
      from IH[OF this] have upd-rev-cs cs (a#as') = cs' .
      with Nil Call show ?thesis by simp
    next
      case (Cons cx csx)
      with ‹upd-rev-cs cs (as'@[a']) = c'#cs'› Call
      have upd-rev-cs csx as' = c'#cs' by simp
      from IH[OF this] have upd-rev-cs csx (a#as') = cs' .
      with Cons Call show ?thesis by simp
    qed
  qed
qed


lemma upd-rev-cs-Cons-Call-Cons-Empty:
  assumes kind a = Q:r↪pfs
  shows upd-rev-cs cs as = [] ⟹ upd-rev-cs cs (a#as) = []
proof(induct as arbitrary:cs rule:rev-induct)
  case Nil
  with ‹kind a = Q:r↪pfs› have upd-rev-cs cs ([]@[a]) = []
    by(simp only:upd-rev-cs.simps) clarsimp
 thus ?case by simp
next
  case (snoc a' as')
  note IH = ‹⋀cs. upd-rev-cs cs as' = [] ⟹ upd-rev-cs cs (a#as') = []›
  show ?case
  proof(cases kind a' rule:edge-kind-cases)
    case Intra
    with ‹upd-rev-cs cs (as'@[a']) = []›
    have upd-rev-cs cs as' = [] by(fastforce simp:intra-kind-def)
    from IH[OF this] have upd-rev-cs cs (a#as') = [] .
    with Intra show ?thesis by(fastforce simp:intra-kind-def)
  next
    case Return
    with ‹upd-rev-cs cs (as'@[a']) = []›
    have upd-rev-cs (a'#cs) as' = [] by simp
    from IH[OF this] have upd-rev-cs (a'#cs) (a#as') = [] .
    with Return show ?thesis by simp
  next
    case Call
    show ?thesis
    proof(cases cs)
      case Nil
```

```
    with ‹upd-rev-cs cs (as'@[a']) = []› Call
    have upd-rev-cs cs as' = [] by simp
    from IH[OF this] have upd-rev-cs cs (a#as') = [] .
    with Nil Call show ?thesis by simp
   next
    case (Cons cx csx)
    with ‹upd-rev-cs cs (as'@[a']) = []› Call
    have upd-rev-cs csx as' = [] by simp
    from IH[OF this] have upd-rev-cs csx (a#as') = [] .
    with Cons Call show ?thesis by simp
   qed
  qed
qed


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↪pfs)
∧
                  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↩pf) ∧
                  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 −
  from ‹valid-call-list cs n›
  have ∀ cs' c cs''. cs = cs'@c#cs'' ⟶ valid-edge c
    by(simp add:valid-call-list-def)
  thus ?thesis
  proof(induct cs)
    case Nil thus ?case by simp
  next
    case (Cons cx csx)
    note IH = ‹∀ cs' c cs''. csx = cs'@c#cs'' ⟶ valid-edge c ⟹
                        ∀ a∈set csx. valid-edge a›
    from ‹∀ cs' c cs''. cx#csx = cs'@c#cs'' ⟶ valid-edge c›
    have valid-edge cx by blast
    from ‹∀ cs' c cs''. cx#csx = cs'@c#cs'' ⟶ valid-edge c›
    have ∀ cs' c cs''. csx = cs'@c#cs'' ⟶ valid-edge c
      by auto(erule-tac x=cx#cs' in allE,auto)
    from IH[OF this] ‹valid-edge cx› show ?case by simp
  qed
qed
```

lemma *valid-return-list-valid-edges*:
assumes *valid-return-list rs n* **shows** $\forall r \in set\ rs.\ valid\text{-}edge\ r$
proof −
from ‹*valid-return-list rs n*›
have $\forall rs'\ r\ rs''.\ rs = rs'@r\#rs'' \longrightarrow valid\text{-}edge\ r$
by(*simp add*:*valid-return-list-def*)
thus *?thesis*
proof(*induct rs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons rx rsx*)
note *IH* = ‹$\forall rs'\ r\ rs''.\ rsx = rs'@r\#rs'' \longrightarrow valid\text{-}edge\ r \Longrightarrow \forall a{\in}set\ rsx.\ valid\text{-}edge\ a$›
from ‹$\forall rs'\ r\ rs''.\ rx\#rsx = rs'@r\#rs'' \longrightarrow valid\text{-}edge\ r$›
have *valid-edge rx* **by** *blast*
from ‹$\forall rs'\ r\ rs''.\ rx\#rsx = rs'@r\#rs'' \longrightarrow valid\text{-}edge\ r$›
have $\forall rs'\ r\ rs''.\ rsx = rs'@r\#rs'' \longrightarrow valid\text{-}edge\ r$
by *auto*(*erule-tac x*=*rx*#*rs'* **in** *allE*,*auto*)
from *IH*[*OF this*] ‹*valid-edge rx*› **show** *?case* **by** *simp*
qed
qed

lemma *vpra-empty-valid-call-list-rev*:
valid-call-list cs n ⟹ *valid-path-rev-aux* [] (*rev cs*)
proof(*induct cs arbitrary*:*n*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons c' cs'*)
note *IH* = ‹$\bigwedge n.$ *valid-call-list cs' n* ⟹ *valid-path-rev-aux* [] (*rev cs'*)›
from ‹*valid-call-list* (*c'*#*cs'*) *n*› **have** *valid-call-list cs'* (*sourcenode c'*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*c'*#*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*,*auto simp*:*sourcenodes-def*)
from *IH*[*OF this*] **have** *valid-path-rev-aux* [] (*rev cs'*) **.**
moreover
from ‹*valid-call-list* (*c'*#*cs'*) *n*› **obtain** *Q r p fs* **where** *kind c'* = *Q*:*r*↪*pfs*
apply(*clarsimp simp*:*valid-call-list-def*)
by(*erule-tac x*=[] **in** *allE*) *fastforce*
ultimately show *?case* **by** *simp*
qed

lemma *vpa-upd-cs-cases*:
⟦*valid-path-aux cs as*; *valid-call-list cs n*; *n* −*as*→∗ *n'*⟧
⟹ *case* (*upd-cs cs as*) *of* [] ⇒ ($\forall c \in set\ cs.\ \exists a \in set\ as.\ a \in get\text{-}return\text{-}edges\ c$)
| *cx*#*csx* ⇒ *valid-call-list* (*cx*#*csx*) *n'*

proof(*induct arbitrary*:*n rule*:*vpa-induct*)
case (*vpa-empty cs*)
from ‹*n* −[]→∗ *n'*› **have** *n* = *n'* **by** *fastforce*
with ‹*valid-call-list cs n*› **show** *?case* **by**(*cases cs*) *auto*
next
case (*vpa-intra cs a' as'*)
note *IH* = ‹⋀*n*. ⟦*valid-call-list cs n*; *n* −*as'*→∗ *n'*⟧
⟹ *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'*›
from ‹*intra-kind* (*kind a'*)› **have** *upd-cs cs* (*a'*#*as'*) = *upd-cs cs as'*
by(*fastforce simp*:*intra-kind-def*)
from ‹*n* −*a'*#*as'*→∗ *n'*› **have** [*simp*]:*n* = *sourcenode a'* **and** *valid-edge a'*
and *targetnode a'* −*as'*→∗ *n'* **by**(*auto elim*:*path-split-Cons*)
from ‹*valid-edge a'*› ‹*intra-kind* (*kind a'*)›
have *get-proc* (*sourcenode a'*) = *get-proc* (*targetnode a'*) **by**(*rule get-proc-intra*)
with ‹*valid-call-list cs n*› **have** *valid-call-list cs* (*targetnode a'*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*) *auto*
from *IH*[*OF this* ‹*targetnode a'* −*as'*→∗ *n'*›] ‹*upd-cs cs* (*a'*#*as'*) = *upd-cs cs as'*›
show *?case* **by**(*cases upd-cs cs as'*) *auto*
next
case (*vpa-Call cs a' as' Q r p fs*)
note *IH* = ‹⋀*n*. ⟦*valid-call-list* (*a'*#*cs*) *n*; *n* −*as'*→∗ *n'*⟧
⟹ *case* (*upd-cs* (*a'*#*cs*) *as'*)
of [] ⇒ ∀ *c*∈*set* (*a'*#*cs*). ∃ *a*∈*set as'*. *a* ∈ *get-return-edges c*
| *cx*#*csx* ⇒ *valid-call-list* (*cx* # *csx*) *n'*›
from ‹*kind a'* = *Q*:*r*↪*p*fs› **have** *upd-cs* (*a'*#*cs*) *as'* = *upd-cs cs* (*a'*#*as'*)
by *simp*
from ‹*n* −*a'*#*as'*→∗ *n'*› **have** [*simp*]:*n* = *sourcenode a'* **and** *valid-edge a'*
and *targetnode a'* −*as'*→∗ *n'* **by**(*auto elim*:*path-split-Cons*)
from ‹*valid-edge a'*› ‹*kind a'* = *Q*:*r*↪*p*fs›
have *get-proc* (*targetnode a'*) = *p* **by**(*rule get-proc-call*)
with ‹*valid-edge a'*› ‹*kind a'* = *Q*:*r*↪*p*fs› ‹*valid-call-list cs n*›
have *valid-call-list* (*a'*#*cs*) (*targetnode a'*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x*=*list* **in** *allE*) **apply** *clarsimp*
by(*case-tac list*,*auto simp*:*sourcenodes-def*)
from *IH*[*OF this* ‹*targetnode a'* −*as'*→∗ *n'*›]
‹*upd-cs* (*a'*#*cs*) *as'* = *upd-cs cs* (*a'*#*as'*)›
have *case upd-cs cs* (*a'*#*as'*)
of [] ⇒ ∀ *c*∈*set* (*a'* # *cs*). ∃ *a*∈*set as'*. *a* ∈ *get-return-edges c*
| *cx* # *csx* ⇒ *valid-call-list* (*cx* # *csx*) *n'* **by** *simp*
thus *?case* **by**(*cases upd-cs cs* (*a'*#*as'*)) *simp*+
next
case (*vpa-ReturnEmpty cs a' as' Q p f*)
note *IH* = ‹⋀*n*. ⟦*valid-call-list* [] *n*; *n* −*as'*→∗ *n'*⟧
⟹ *case* (*upd-cs* [] *as'*)

of [] ⇒ ∀ *c*∈*set* []. ∃ *a*∈*set as'*. *a* ∈ *get-return-edges c*
| *cx*#*csx* ⇒ *valid-call-list* (*cx* # *csx*) *n'*⟩
from ⟨*kind a'* = *Q*↩*pf*⟩ ⟨*cs* = []⟩ **have** *upd-cs* [] *as'* = *upd-cs cs* (*a'*#*as'*)
by *simp*
from ⟨*n* −*a'*#*as'*→∗ *n'*⟩ **have** [*simp*]:*n* = *sourcenode a'* **and** *valid-edge a'*
and *targetnode a'* −*as'*→∗ *n'* **by**(*auto elim*:*path-split-Cons*)
have *valid-call-list* [] (*targetnode a'*) **by**(*simp add*:*valid-call-list-def*)
from *IH*[*OF this* ⟨*targetnode a'* −*as'*→∗ *n'*⟩]
⟨*upd-cs* [] *as'* = *upd-cs cs* (*a'*#*as'*)⟩
have *case* (*upd-cs cs* (*a'*#*as'*))
of [] ⇒ ∀ *c*∈*set* []. ∃ *a*∈*set as'*. *a* ∈ *get-return-edges c*
| *cx*#*csx* ⇒ *valid-call-list* (*cx*#*csx*) *n'* **by** *simp*
with ⟨*cs* = []⟩ **show** *?case* **by**(*cases upd-cs cs* (*a'*#*as'*)) *simp*+
next
case (*vpa-ReturnCons cs a' as' Q p f c' cs'*)
note *IH* = ⟨⋀*n*. ⟦*valid-call-list cs' n*; *n* −*as'*→∗ *n'*⟧
⟹ *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'*⟩
from ⟨*kind a'* = *Q*↩*pf*⟩ ⟨*cs* = *c'*#*cs'*⟩ ⟨*a'* ∈ *get-return-edges c'*⟩
have *upd-cs cs' as'* = *upd-cs cs* (*a'*#*as'*) **by** *simp*
from ⟨*n* −*a'*#*as'*→∗ *n'*⟩ **have** [*simp*]:*n* = *sourcenode a'* **and** *valid-edge a'*
and *targetnode a'* −*as'*→∗ *n'* **by**(*auto elim*:*path-split-Cons*)
from ⟨*valid-call-list cs n*⟩ ⟨*cs* = *c'*#*cs'*⟩ **have** *valid-edge c'*
apply(*clarsimp simp*:*valid-call-list-def*)
by(*erule-tac x*=[] **in** *allE*,*auto*)
with ⟨*a'* ∈ *get-return-edges c'*⟩ **obtain** *ax* **where** *valid-edge ax*
and *sources*:*sourcenode ax* = *sourcenode c'*
and *targets*:*targetnode ax* = *targetnode a'* **and** *kind ax* = (λ*cf*. *False*)$_{\surd}$
by(*fastforce dest*:*call-return-node-edge*)
from ⟨*valid-edge ax*⟩ *sources*[*THEN sym*] *targets*[*THEN sym*] ⟨*kind ax* = (λ*cf*. *False*)$_{\surd}$⟩
have *get-proc* (*sourcenode c'*) = *get-proc* (*targetnode a'*)
by(*fastforce intro*:*get-proc-intra simp*:*intra-kind-def*)
with ⟨*valid-call-list cs n*⟩ ⟨*cs* = *c'*#*cs'*⟩
have *valid-call-list cs'* (*targetnode a'*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*c'*#*cs'* **in** *allE*)
by(*case-tac cs'*,*auto simp*:*sourcenodes-def*)
from *IH*[*OF this* ⟨*targetnode a'* −*as'*→∗ *n'*⟩]
⟨*upd-cs cs' as'* = *upd-cs cs* (*a'*#*as'*)⟩
have *case* (*upd-cs cs* (*a'*#*as'*))
of [] ⇒ ∀ *c*∈*set cs'*. ∃ *a*∈*set as'*. *a* ∈ *get-return-edges c*
| *cx*#*csx* ⇒ *valid-call-list* (*cx*#*csx*) *n'* **by** *simp*
with ⟨*cs* = *c'* # *cs'*⟩ ⟨*a'* ∈ *get-return-edges c'*⟩ **show** *?case*
by(*cases upd-cs cs* (*a'*#*as'*)) *simp*+
qed

lemma *vpa-valid-call-list-valid-return-list-vpra*:
⟦*valid-path-aux cs cs'*; *valid-call-list cs n*; *valid-return-list cs' n'*⟧
⟹ *valid-path-rev-aux cs'* (*rev cs*)
proof(*induct arbitrary*:*n n' rule*:*vpa-induct*)
case (*vpa-empty cs*)
from ‹*valid-call-list cs n*› **show** *?case* **by**(*rule vpra-empty-valid-call-list-rev*)
next
case (*vpa-intra cs a as*)
from ‹*intra-kind* (*kind a*)› ‹*valid-return-list* (*a*#*as*) *n'*›
have *False* **apply**(*clarsimp simp*:*valid-return-list-def*)
by(*erule-tac x*=[] **in** *allE*,*clarsimp simp*:*intra-kind-def*)
thus *?case* **by** *simp*
next
case (*vpa-Call cs a as Q r p fs*)
from ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*valid-return-list* (*a*#*as*) *n'*›
have *False* **apply**(*clarsimp simp*:*valid-return-list-def*)
by(*erule-tac x*=[] **in** *allE*,*clarsimp*)
thus *?case* **by** *simp*
next
case (*vpa-ReturnEmpty cs a as Q p f*)
from ‹*cs* = []› **show** *?case* **by** *simp*
next
case (*vpa-ReturnCons cs a as Q p f c' cs'*)
note *IH* = ‹⋀*n n'*. ⟦*valid-call-list cs' n*; *valid-return-list as n'*⟧
⟹ *valid-path-rev-aux as* (*rev cs'*)›
from ‹*valid-return-list* (*a*#*as*) *n'*› **have** *valid-return-list as* (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*a*#*cs'* **in** *allE*)
by(*case-tac cs'*,*auto simp*:*targetnodes-def*)
from ‹*valid-call-list cs n*› ‹*cs* = *c'*#*cs'*›
have *valid-call-list cs'* (*sourcenode c'*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*c'*#*cs'* **in** *allE*)
by(*case-tac cs'*,*auto simp*:*sourcenodes-def*)
from ‹*valid-call-list cs n*› ‹*cs* = *c'*#*cs'*› **have** *valid-edge c'*
apply(*clarsimp simp*:*valid-call-list-def*)
by(*erule-tac x*=[] **in** *allE*,*auto*)
with ‹*a* ∈ *get-return-edges c'*› **obtain** *Q' r' p' f'* **where** *kind c'* = *Q'*:*r'*↪$_{p'}$*f'*
apply(*cases kind c' rule*:*edge-kind-cases*)
by(*auto dest*:*only-call-get-return-edges simp*:*intra-kind-def*)
from *IH*[*OF* ‹*valid-call-list cs'* (*sourcenode c'*)›
‹*valid-return-list as* (*targetnode a*)›]
have *valid-path-rev-aux as* (*rev cs'*) .
with ‹*kind a* = *Q*↩$_p$*f*› ‹*cs* = *c'*#*cs'*› ‹*a* ∈ *get-return-edges c'*› ‹*kind c'* = *Q'*:*r'*↪$_{p'}$*f'*›
show *?case* **by** *simp*
qed

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'*⟧
⟹ *valid-path-rev-aux cs' as* ∧ *valid-path-rev-aux* (*upd-rev-cs cs' as*) (*rev cs*)
proof(*induct arbitrary*:*n rule*:*vpa-induct*)
case *vpa-empty* **thus** *?case*
by(*fastforce intro*:*vpa-valid-call-list-valid-return-list-vpra*)
next
case (*vpa-intra cs a as*)
note *IH* = ‹⋀*n*. ⟦*valid-path-aux* (*upd-cs cs as*) *cs'*; *n* −*as*→∗ *n'*;
valid-call-list cs n; *valid-return-list cs' n'*⟧
⟹ *valid-path-rev-aux cs' as* ∧
valid-path-rev-aux (*upd-rev-cs cs' as*) (*rev cs*)›
from ‹*n* −*a*#*as*→∗ *n'*› **have** *n* = *sourcenode a* **and** *valid-edge a*
and *targetnode a* −*as*→∗ *n'* **by**(*auto intro*:*path-split-Cons*)
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
with ‹*valid-call-list cs n*› ‹*n* = *sourcenode a*›
have *valid-call-list cs* (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*) *auto*
from ‹*valid-path-aux* (*upd-cs cs* (*a*#*as*)) *cs'*› ‹*intra-kind* (*kind a*)›
have *valid-path-aux* (*upd-cs cs as*) *cs'*
by(*fastforce simp*:*intra-kind-def*)
from *IH*[*OF this* ‹*targetnode a* −*as*→∗ *n'*› ‹*valid-call-list cs* (*targetnode a*)›
‹*valid-return-list cs' n'*›]
have *valid-path-rev-aux cs' as*
and *valid-path-rev-aux* (*upd-rev-cs cs' as*) (*rev cs*) **by** *simp-all*
from ‹*intra-kind* (*kind a*)› ‹*valid-path-rev-aux cs' as*›
have *valid-path-rev-aux cs'* (*a*#*as*) **by**(*rule vpra-Cons-intra*)
from ‹*intra-kind* (*kind a*)› **have** *upd-rev-cs cs'* (*a*#*as*) = *upd-rev-cs cs' as*
by(*simp add*:*upd-rev-cs-Cons-intra*)
with ‹*valid-path-rev-aux* (*upd-rev-cs cs' as*) (*rev cs*)›
have *valid-path-rev-aux* (*upd-rev-cs cs'* (*a*#*as*)) (*rev cs*) **by** *simp*
with ‹*valid-path-rev-aux cs'* (*a*#*as*)› **show** *?case* **by** *simp*
next
case (*vpa-Call cs a as Q r p fs*)
note *IH* = ‹⋀*n*. ⟦*valid-path-aux* (*upd-cs* (*a*#*cs*) *as*) *cs'*; *n* −*as*→∗ *n'*;
valid-call-list (*a*#*cs*) *n*; *valid-return-list cs' n'*⟧
⟹ *valid-path-rev-aux cs' as* ∧
valid-path-rev-aux (*upd-rev-cs cs' as*) (*rev* (*a*#*cs*))›
from ‹*n* −*a*#*as*→∗ *n'*› **have** *n* = *sourcenode a* **and** *valid-edge a*
and *targetnode a* −*as*→∗ *n'* **by**(*auto intro*:*path-split-Cons*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*p*fs› **have** *p* = *get-proc* (*targetnode a*)
by(*rule get-proc-call*[*THEN sym*])
from ‹*valid-call-list cs n*› ‹*n* = *sourcenode a*›
have *valid-call-list cs* (*sourcenode a*) **by** *simp*

```
  with ‹kind a = Q:r↪pfs› ‹valid-edge a› ‹p = get-proc (targetnode a)›
  have valid-call-list (a#cs) (targetnode a)
    apply(clarsimp simp:valid-call-list-def)
    apply(case-tac cs') apply auto
    apply(erule-tac x=list in allE) apply clarsimp
    by(case-tac list,auto simp:sourcenodes-def)
  from ‹kind a = Q:r↪pfs› have upd-cs cs (a#as) = upd-cs (a#cs) as
    by simp
  with ‹valid-path-aux (upd-cs cs (a#as)) cs'›
  have valid-path-aux (upd-cs (a#cs) as) cs' by simp
  from IH[OF this ‹targetnode a −as→∗ n'› ‹valid-call-list (a#cs) (targetnode a)›
    ‹valid-return-list cs' n''›]
  have valid-path-rev-aux cs' as
    and valid-path-rev-aux (upd-rev-cs cs' as) (rev (a#cs)) by simp-all
  show ?case
  proof(cases upd-rev-cs cs' as)
    case Nil
    with ‹kind a = Q:r↪pfs›
    have upd-rev-cs cs' (a#as) = [] by(rule upd-rev-cs-Cons-Call-Cons-Empty)
   with ‹valid-path-rev-aux (upd-rev-cs cs' as) (rev (a#cs))› ‹kind a = Q:r↪pfs›
Nil
    have valid-path-rev-aux (upd-rev-cs cs' (a#as)) (rev cs) by simp
     from Nil ‹kind a = Q:r↪pfs› have valid-path-rev-aux (upd-rev-cs cs' as)
([]@[a])
      by(simp only:valid-path-rev-aux.simps) clarsimp
    with ‹valid-path-rev-aux cs' as› have valid-path-rev-aux cs' ([a]@as)
      by(fastforce intro:valid-path-rev-aux-Append)
    with ‹valid-path-rev-aux (upd-rev-cs cs' (a#as)) (rev cs)›
    show ?thesis by simp
  next
    case (Cons cx csx)
   with ‹valid-path-rev-aux (upd-rev-cs cs' as) (rev (a#cs))› ‹kind a = Q:r↪pfs›
    have match:cx ∈ get-return-edges a valid-path-rev-aux csx (rev cs) by auto
    from ‹kind a = Q:r↪pfs› Cons have upd-rev-cs cs' (a#as) = csx
      by(rule upd-rev-cs-Cons-Call-Cons)
    with ‹valid-path-rev-aux (upd-rev-cs cs' as) (rev(a#cs))› ‹kind a = Q:r↪pfs›
match
    have valid-path-rev-aux (upd-rev-cs cs' (a#as)) (rev cs) by simp
    from Cons ‹kind a = Q:r↪pfs› match
    have valid-path-rev-aux (upd-rev-cs cs' as) ([]@[a])
      by(simp only:valid-path-rev-aux.simps) clarsimp
    with ‹valid-path-rev-aux cs' as› have valid-path-rev-aux cs' ([a]@as)
      by(fastforce intro:valid-path-rev-aux-Append)
    with ‹valid-path-rev-aux (upd-rev-cs cs' (a#as)) (rev cs)›
    show ?thesis by simp
  qed
next
  case (vpa-ReturnEmpty cs a as Q p f)
  note IH = ‹⋀n. ⟦valid-path-aux (upd-cs [] as) cs'; n −as→∗ n';
```

```
    valid-call-list [] n; valid-return-list cs' n'⟧
    ⟹ valid-path-rev-aux cs' as ∧
       valid-path-rev-aux (upd-rev-cs cs' as) (rev [])›
  from ‹n −a#as→* n'› have n = sourcenode a and valid-edge a
    and targetnode a −as→* n' by(auto intro:path-split-Cons)
  from ‹cs = []› ‹kind a = Q↩pf› have upd-cs cs (a#as) = upd-cs [] as
    by simp
  with ‹valid-path-aux (upd-cs cs (a#as)) cs'›
  have valid-path-aux (upd-cs [] as) cs' by simp
  from IH[OF this ‹targetnode a −as→* n'› - ‹valid-return-list cs' n''›]
  have valid-path-rev-aux cs' as
    and valid-path-rev-aux (upd-rev-cs cs' as) (rev [])
    by(auto simp:valid-call-list-def)
  from ‹kind a = Q↩pf› ‹valid-path-rev-aux cs' as›
  have valid-path-rev-aux cs' (a#as) by(rule vpra-Cons-Return)
  moreover
  from ‹cs = []› have valid-path-rev-aux (upd-rev-cs cs' (a#as)) (rev cs)
    by simp
  ultimately show ?case by simp
next
  case (vpa-ReturnCons cs a as Q p f cx csx)
  note IH = ‹⋀n. ⟦valid-path-aux (upd-cs csx as) cs'; n −as→* n';
    valid-call-list csx n; valid-return-list cs' n''⟧
    ⟹ valid-path-rev-aux cs' as ∧
       valid-path-rev-aux (upd-rev-cs cs' as) (rev csx)›
  note match = ‹cs = cx#csx› ‹a ∈ get-return-edges cx›
  from ‹n −a#as→* n'› have n = sourcenode a and valid-edge a
    and targetnode a −as→* n' by(auto intro:path-split-Cons)
  from ‹cs = cx#csx› ‹valid-call-list cs n› have valid-edge cx
    apply(clarsimp simp:valid-call-list-def)
    by(erule-tac x=[] in allE) clarsimp
  with match have get-proc (sourcenode cx) = get-proc (targetnode a)
    by(fastforce intro:get-proc-get-return-edge)
  with ‹valid-call-list cs n› ‹cs = cx#csx›
  have valid-call-list csx (targetnode a)
    apply(clarsimp simp:valid-call-list-def)
    apply(erule-tac x=cx#cs' in allE) apply clarsimp
    by(case-tac cs',auto simp:sourcenodes-def)
  from ‹kind a = Q↩pf› match have upd-cs cs (a#as) = upd-cs csx as by simp
  with ‹valid-path-aux (upd-cs cs (a#as)) cs'›
  have valid-path-aux (upd-cs csx as) cs' by simp
  from IH[OF this ‹targetnode a −as→* n'› ‹valid-call-list csx (targetnode a)›
    ‹valid-return-list cs' n''›]
  have valid-path-rev-aux cs' as
    and valid-path-rev-aux (upd-rev-cs cs' as) (rev csx) by simp-all
  from ‹kind a = Q↩pf› ‹valid-path-rev-aux cs' as›
  have valid-path-rev-aux cs' (a#as) by(rule vpra-Cons-Return)
  from match ‹valid-edge cx› obtain Q' r' p' f' where kind cx = Q':r'↪p'f'
    by(fastforce dest!:only-call-get-return-edges)
```

from ‹*kind a* = *Q*↩$_p$*f*› **have** *upd-rev-cs cs'* (*a*#*as*) = *a*#(*upd-rev-cs cs'* *as*)
by(*rule upd-rev-cs-Cons-Return*)
with ‹*valid-path-rev-aux* (*upd-rev-cs cs'* *as*) (*rev csx*)› ‹*kind a* = *Q*↩$_p$*f*›
‹*kind cx* = *Q'*:*r'*↪$_{p'}$*f'*› *match*
have *valid-path-rev-aux* (*upd-rev-cs cs'* (*a*#*as*)) (*rev cs*)
by *simp*
with ‹*valid-path-rev-aux cs'* (*a*#*as*)› **show** *?case* **by** *simp*
qed

lemma *vp-to-vpra*:
n −*as*→$_{\surd}$* *n'* ⟹ *valid-path-rev-aux* [] *as*
by(*fastforce elim*:*vpa-to-vpra*[*THEN conjunct1*]
simp:*vp-def valid-path-def valid-call-list-def valid-return-list-def*)

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*: ⋀*cs*. *P cs* []
⋀*cs a as*. ⟦*intra-kind*(*kind a*); *same-level-path-aux cs as*; *P cs as*⟧
⟹ *P cs* (*a*#*as*)
⋀*cs a as Q r p fs*. ⟦*kind a* = *Q*:*r*↪$_p$*fs*; *same-level-path-aux* (*a*#*cs*) *as*; *P* (*a*#*cs*) *as*⟧
⟹ *P cs* (*a*#*as*)
⋀*cs a as Q p f c' cs'*. ⟦*kind a* = *Q*↩$_p$*f*; *cs* = *c'*#*cs'*; *same-level-path-aux cs'* *as*;
a ∈ *get-return-edges c'*; *P cs' as*⟧
⟹ *P cs* (*a*#*as*)
shows *P xs ys*
using *major*
apply(*induct ys arbitrary*: *xs*)
by(*auto intro*:*rules split*:*edge-kind.split-asm list.split-asm simp*:*intra-kind-def*)

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 ∀ *c* ∈ *set cs*. *valid-edge c* **and** ∀ *a* ∈ *set as*. *valid-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** $same\text{-}level\text{-}path\text{-}aux\ cs\ asx$
and $kind\ a = Q{\hookleftarrow_p}f$ **and** $upd\text{-}cs\ cs\ asx = c'\#cs'$ **and** $upd\text{-}cs\ cs\ (asx@[a]) = []$
and $a \in get\text{-}return\text{-}edges\ c'$ **and** $valid\text{-}edge\ c'$
and $\forall\, a \in set\ asx'.\ intra\text{-}kind(kind\ a)$
proof(*atomize-elim*)
from *assms*
show $(\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a)) \vee$
$(\exists\, asx\ a\ asx'\ Q\ p\ f\ c'\ cs'.\ as = asx@a\#asx' \wedge same\text{-}level\text{-}path\text{-}aux\ cs\ asx \wedge$
$kind\ a = Q{\hookleftarrow_p}f \wedge upd\text{-}cs\ cs\ asx = c'\#cs' \wedge upd\text{-}cs\ cs\ (asx@[a]) = [] \wedge$
$a \in get\text{-}return\text{-}edges\ c' \wedge valid\text{-}edge\ c' \wedge (\forall\, a{\in}set\ asx'.\ intra\text{-}kind\ (kind\ a)))$
proof(*induct rule*:*slpa-induct*)
case (*slpa-empty cs*)
have $\forall\, a{\in}set\ []\ .\ intra\text{-}kind\ (kind\ a)$ **by** *simp*
thus *?case* **by** *simp*
next
case (*slpa-intra cs a as*)
note *IH* = ‹$[\![upd\text{-}cs\ cs\ as = []; \forall\, c{\in}set\ cs.\ valid\text{-}edge\ c; \forall\, a'{\in}set\ as.\ valid\text{-}edge\ a']\!]$
$\Longrightarrow (\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a)) \vee$
$(\exists\, asx\ a\ asx'\ Q\ p\ f\ c'\ cs'.\ as = asx@a\#asx' \wedge same\text{-}level\text{-}path\text{-}aux\ cs\ asx \wedge$
$kind\ a = Q{\hookleftarrow_p}f \wedge\ upd\text{-}cs\ cs\ asx = c'\ \#\ cs' \wedge upd\text{-}cs\ cs\ (asx@[a]) = [] \wedge$
$a \in get\text{-}return\text{-}edges\ c' \wedge valid\text{-}edge\ c' \wedge (\forall\, a{\in}set\ asx'.\ intra\text{-}kind\ (kind\ a)))$›
from ‹$\forall\, a'{\in}set\ (a\#as).\ valid\text{-}edge\ a'$› **have** $\forall\, a'{\in}set\ as.\ valid\text{-}edge\ a'$ **by** *simp*
from ‹$intra\text{-}kind\ (kind\ a)$› ‹$upd\text{-}cs\ cs\ (a\#as) = []$›
have $upd\text{-}cs\ cs\ as = []$ **by**(*fastforce simp*:*intra-kind-def*)
from *IH*[*OF this* ‹$\forall\, c{\in}set\ cs.\ valid\text{-}edge\ c$› ‹$\forall\, a'{\in}set\ as.\ valid\text{-}edge\ a'$›] **show** *?case*
proof
assume $\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a)$
with ‹$intra\text{-}kind\ (kind\ a)$› **have** $\forall\, a'{\in}set\ (a\#as).\ intra\text{-}kind\ (kind\ a')$
by *simp*
thus *?case* **by** *simp*
next
assume $\exists\, asx\ a\ asx'\ Q\ p\ f\ c'\ cs'.\ as = asx@a\#asx' \wedge same\text{-}level\text{-}path\text{-}aux\ cs\ asx \wedge$
$kind\ a = Q{\hookleftarrow_p}f \wedge upd\text{-}cs\ cs\ asx = c'\#cs' \wedge upd\text{-}cs\ cs\ (asx@[a]) = [] \wedge$
$a \in get\text{-}return\text{-}edges\ c' \wedge valid\text{-}edge\ c' \wedge$
$(\forall\, a{\in}set\ asx'.\ intra\text{-}kind\ (kind\ a))$
then obtain $asx\ a'\ Q\ p\ f\ asx'\ c'\ cs'$ **where** $as = asx@a'\#asx'$
and $same\text{-}level\text{-}path\text{-}aux\ cs\ asx$ **and** $upd\text{-}cs\ cs\ (asx@[a']) = []$
and $upd\text{-}cs\ cs\ asx = c'\#cs'$ **and** *assms*:$a' \in get\text{-}return\text{-}edges\ c'$
$kind\ a' = Q{\hookleftarrow_p}f\ valid\text{-}edge\ c'\ \forall\, a{\in}set\ asx'.\ intra\text{-}kind\ (kind\ a)$
by *blast*
from ‹$as = asx@a'\#asx'$› **have** $a\#as = (a\#asx)@a'\#asx'$ **by** *simp*
moreover

```
    from ‹intra-kind (kind a)› ‹same-level-path-aux cs asx›
    have same-level-path-aux cs (a#asx) by(fastforce simp:intra-kind-def)
    moreover
    from ‹upd-cs cs asx = c'#cs'› ‹intra-kind (kind a)›
    have upd-cs cs (a#asx) = c'#cs' by(fastforce simp:intra-kind-def)
    moreover
    from ‹upd-cs cs (asx@[a']) = []› ‹intra-kind (kind a)›
    have upd-cs cs ((a#asx)@[a']) = [] by(fastforce simp:intra-kind-def)
    ultimately show ?case using assms by blast
  qed
next
  case (slpa-Call cs a as Q r p fs)
  note IH = ‹⟦upd-cs (a#cs) as = []; ∀ c∈set (a#cs). valid-edge c;
    ∀ a'∈set as. valid-edge a'⟧ ⟹
    (∀ a'∈set as. intra-kind (kind a')) ∨
    (∃ asx a' asx' Q' p' f' c' cs'. as = asx@a'#asx' ∧
      same-level-path-aux (a#cs) asx ∧ kind a' = Q'↩_p'f' ∧
      upd-cs (a#cs) asx = c'#cs' ∧ upd-cs (a#cs) (asx@[a']) = [] ∧
      a' ∈ get-return-edges c' ∧ valid-edge c' ∧
      (∀ a'∈set asx'. intra-kind (kind a')))›
  from ‹∀ a'∈set (a#as). valid-edge a'› have valid-edge a
    and ∀ a'∈set as. valid-edge a' by simp-all
  from ‹∀ c∈set cs. valid-edge c› ‹valid-edge a› have ∀ c∈set (a#cs). valid-edge c
    by simp
  from ‹upd-cs cs (a#as) = []› ‹kind a = Q:r↪_pfs›
  have upd-cs (a#cs) as = [] by simp
  from IH[OF this ‹∀ c∈set (a#cs). valid-edge c› ‹∀ a'∈set as. valid-edge a'›]
  show ?case
  proof
    assume ∀ a'∈set as. intra-kind (kind a')
    with ‹kind a = Q:r↪_pfs› have upd-cs cs (a#as) = a#cs
      by(fastforce intro:upd-cs-intra-path)
    with ‹upd-cs cs (a#as) = []› have False by simp
    thus ?case by simp
  next
    assume ∃ asx a' asx' Q p f c' cs'. as = asx@a'#asx' ∧
          same-level-path-aux (a#cs) asx ∧ kind a' = Q↩_pf ∧
          upd-cs (a#cs) asx = c'#cs' ∧ upd-cs (a#cs) (asx@[a']) = [] ∧
          a' ∈ get-return-edges c' ∧ valid-edge c' ∧
          (∀ a∈set asx'. intra-kind (kind a))
    then obtain asx a' Q' p' f' asx' c' cs' where as = asx@a'#asx'
      and same-level-path-aux (a#cs) asx and upd-cs (a#cs) (asx@[a']) = []
      and upd-cs (a#cs) asx = c'#cs' and assms:a' ∈ get-return-edges c'
      kind a' = Q'↩_p'f' valid-edge c' ∀ a∈set asx'. intra-kind (kind a)
      by blast
    from ‹as = asx@a'#asx'› have a#as = (a#asx)@a'#asx' by simp
    moreover
    from ‹kind a = Q:r↪_pfs› ‹same-level-path-aux (a#cs) asx›
    have same-level-path-aux cs (a#asx) by simp
```

```
    moreover
    from ‹kind a = Q:r↪pfs› ‹upd-cs (a#cs) asx = c'#cs'›
    have upd-cs cs (a#asx) = c'#cs' by simp
    moreover
    from ‹kind a = Q:r↪pfs› ‹upd-cs (a#cs) (asx@[a']) = []›
    have upd-cs cs ((a#asx)@[a']) = [] by simp
    ultimately show ?case using assms by blast
  qed
 next
  case (slpa-Return cs a as Q p f c' cs')
  note IH = ‹⟦upd-cs cs' as = []; ∀ c∈set cs'. valid-edge c;
   ∀ a'∈set as. valid-edge a'⟧ ⟹
   (∀ a'∈set as. intra-kind (kind a')) ∨
   (∃ asx a' asx' Q' p' f' c'' cs''. as = asx@a'#asx' ∧
    same-level-path-aux cs' asx ∧ kind a' = Q'↩p'f' ∧ upd-cs cs' asx = c''#cs''
∧
     upd-cs cs' (asx@[a']) = [] ∧ a' ∈ get-return-edges c'' ∧ valid-edge c'' ∧
     (∀ a'∈set asx'. intra-kind (kind a')))›
  from ‹∀ a'∈set (a#as). valid-edge a'› have valid-edge a
    and ∀ a'∈set as. valid-edge a' by simp-all
  from ‹∀ c∈set cs. valid-edge c› ‹cs = c' # cs'›
  have valid-edge c' and ∀ c∈set cs'. valid-edge c by simp-all
  from ‹upd-cs cs (a#as) = []› ‹kind a = Q↩pf› ‹cs = c'#cs'›
    ‹a ∈ get-return-edges c'› have upd-cs cs' as = [] by simp
  from IH[OF this ‹∀ c∈set cs'. valid-edge c› ‹∀ a'∈set as. valid-edge a'›] show
?case
  proof
    assume ∀ a'∈set as. intra-kind (kind a')
    hence upd-cs cs' as = cs' by(rule upd-cs-intra-path)
    with ‹upd-cs cs' as = []› have cs' = [] by simp
    with ‹cs = c'#cs'› ‹a ∈ get-return-edges c'› ‹kind a = Q↩pf›
    have upd-cs cs [a] = [] by simp
    moreover
    from ‹cs = c'#cs'› have upd-cs cs [] ≠ [] by simp
    moreover
    have same-level-path-aux cs [] by simp
    ultimately show ?case
      using ‹kind a = Q↩pf› ‹∀ a'∈set as. intra-kind (kind a')› ‹cs = c'#cs'›
        ‹a ∈ get-return-edges c'› ‹valid-edge c'›
      by fastforce
  next
    assume ∃ asx a' asx' Q' p' f' c'' cs''. as = asx@a'#asx' ∧
     same-level-path-aux cs' asx ∧ kind a' = Q'↩p'f' ∧ upd-cs cs' asx = c''#cs''
∧
     upd-cs cs' (asx@[a']) = [] ∧ a' ∈ get-return-edges c'' ∧ valid-edge c'' ∧
     (∀ a'∈set asx'. intra-kind (kind a'))
    then obtain asx a' asx' Q' p' f' c'' cs'' where as = asx@a'#asx'
      and same-level-path-aux cs' asx and upd-cs cs' asx = c''#cs''
      and upd-cs cs' (asx@[a']) = [] and assms:a' ∈ get-return-edges c''
```

```
      kind a' = Q'↩p'f' valid-edge c'' ∀ a'∈set asx'. intra-kind (kind a')
      by blast
    from ‹as = asx@a'#asx'› have a#as = (a#asx)@a'#asx' by simp
    moreover
    from ‹same-level-path-aux cs' asx› ‹cs = c'#cs'› ‹a ∈ get-return-edges c'›
      ‹kind a = Q↩pf›
    have same-level-path-aux cs (a#asx) by simp
    moreover
    from ‹upd-cs cs' asx = c''#cs''› ‹kind a = Q↩pf› ‹cs = c'#cs'›
    have upd-cs cs (a#asx) = c''#cs'' by simp
    moreover
    from ‹upd-cs cs' (asx@[a']) = []› ‹cs = c'#cs'› ‹a ∈ get-return-edges c'›
      ‹kind a = Q↩pf›
    have upd-cs cs ((a#asx)@[a']) = [] by simp
    ultimately show ?case using assms by blast
  qed
 qed
qed


lemma same-level-path-aux-valid-path-aux:
  same-level-path-aux cs as ⟹ valid-path-aux cs as
by(induct rule:slpa-induct,auto split:edge-kind.split simp:intra-kind-def)


lemma same-level-path-aux-Append:
  ⟦same-level-path-aux cs as; same-level-path-aux (upd-cs cs as) as'⟧
  ⟹ same-level-path-aux cs (as@as')
by(induct rule:slpa-induct,auto simp:intra-kind-def)


lemma same-level-path-aux-callstack-Append:
  same-level-path-aux cs as ⟹ same-level-path-aux (cs@cs') as
by(induct rule:slpa-induct,auto simp:intra-kind-def)


lemma same-level-path-upd-cs-callstack-Append:
  ⟦same-level-path-aux cs as; upd-cs cs as = cs'⟧
  ⟹ upd-cs (cs@cs'') as = (cs'@cs'')
by(induct rule:slpa-induct,auto split:edge-kind.split simp:intra-kind-def)


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
using assms
proof(induct arbitrary:xs ys rule:slpa-induct)
  case (slpa-empty cs) case 1
  from ‹[] = xs@ys› show ?case by simp
```

```
next
  case (slpa-empty cs) case 2
  from ‹[] = xs@ys› show ?case by simp
next
  case (slpa-intra cs a as)
  note IH1 = ‹⋀xs ys. ⟦as = xs@ys; upd-cs cs xs = []⟧ ⟹ same-level-path-aux cs
xs›
  note IH2 = ‹⋀xs ys. ⟦as = xs@ys; upd-cs cs xs = []⟧ ⟹ same-level-path-aux []
ys›
  { case 1
    show ?case
    proof(cases xs)
      case Nil thus ?thesis by simp
    next
      case (Cons x' xs')
      with ‹a#as = xs@ys› have a = x' and as = xs'@ys by simp-all
      with ‹upd-cs cs xs = []› Cons ‹intra-kind (kind a)›
      have upd-cs cs xs' = [] by(fastforce simp:intra-kind-def)
      from IH1[OF ‹as = xs'@ys› this] have same-level-path-aux cs xs' .
      with ‹a = x'› ‹intra-kind (kind a)› Cons
      show ?thesis by(fastforce simp:intra-kind-def)
    qed
  next
    case 2
    show ?case
    proof(cases xs)
      case Nil
      with ‹upd-cs cs xs = []› have cs = [] by fastforce
      with Nil ‹a#as = xs@ys› ‹same-level-path-aux cs as› ‹intra-kind (kind a)›
      show ?thesis by(cases ys,auto simp:intra-kind-def)
    next
      case (Cons x' xs')
      with ‹a#as = xs@ys› have a = x' and as = xs'@ys by simp-all
      with ‹upd-cs cs xs = []› Cons ‹intra-kind (kind a)›
      have upd-cs cs xs' = [] by(fastforce simp:intra-kind-def)
      from IH2[OF ‹as = xs'@ys› this] show ?thesis .
    qed
  }
next
  case (slpa-Call cs a as Q r p fs)
  note IH1 = ‹⋀xs ys. ⟦as = xs@ys; upd-cs (a#cs) xs = []⟧
    ⟹ same-level-path-aux (a#cs) xs›
  note IH2 = ‹⋀xs ys. ⟦as = xs@ys; upd-cs (a#cs) xs = []⟧
    ⟹ same-level-path-aux [] ys›
  { case 1
    show ?case
    proof(cases xs)
      case Nil thus ?thesis by simp
    next
```

```
    case (Cons x' xs')
    with ‹a#as = xs@ys› have a = x' and as = xs'@ys by simp-all
    with ‹upd-cs cs xs = []› Cons ‹kind a = Q:r↪pfs›
    have upd-cs (a#cs) xs' = [] by simp
    from IH1[OF ‹as = xs'@ys› this] have same-level-path-aux (a#cs) xs' .
    with ‹a = x'› ‹kind a = Q:r↪pfs› Cons show ?thesis by simp
   qed
  next
   case 2
   show ?case
   proof(cases xs)
    case Nil
    with ‹upd-cs cs xs = []› have cs = [] by fastforce
   with Nil ‹a#as = xs@ys› ‹same-level-path-aux (a#cs) as› ‹kind a = Q:r↪pfs›
    show ?thesis by(cases ys) auto
   next
    case (Cons x' xs')
    with ‹a#as = xs@ys› have a = x' and as = xs'@ys by simp-all
    with ‹upd-cs cs xs = []› Cons ‹kind a = Q:r↪pfs›
    have upd-cs (a#cs) xs' = [] by simp
    from IH2[OF ‹as = xs'@ys› this] show ?thesis .
   qed
  }
next
  case (slpa-Return cs a as Q p f c' cs')
  note IH1 = ‹⋀xs ys. ⟦as = xs@ys; upd-cs cs' xs = []⟧ ⟹ same-level-path-aux
cs' xs›
  note IH2 = ‹⋀xs ys. ⟦as = xs@ys; upd-cs cs' xs = []⟧ ⟹ same-level-path-aux
[] ys›
  { case 1
   show ?case
   proof(cases xs)
    case Nil thus ?thesis by simp
   next
    case (Cons x' xs')
    with ‹a#as = xs@ys› have a = x' and as = xs'@ys by simp-all
    with ‹upd-cs cs xs = []› Cons ‹kind a = Q↩pf› ‹cs = c'#cs'›
    have upd-cs cs' xs' = [] by simp
    from IH1[OF ‹as = xs'@ys› this] have same-level-path-aux cs' xs' .
   with ‹a = x'› ‹kind a = Q↩pf› ‹cs = c'#cs'› ‹a ∈ get-return-edges c'› Cons
    show ?thesis by simp
   qed
  next
   case 2
   show ?case
   proof(cases xs)
    case Nil
    with ‹upd-cs cs xs = []› have cs = [] by fastforce
    with ‹cs = c'#cs'› have False by simp
```

```
    thus ?thesis by simp
  next
    case (Cons x' xs')
    with ‹a#as = xs@ys› have a = x' and as = xs'@ys by simp-all
    with ‹upd-cs cs xs = []› Cons ‹kind a = Q↩pf› ‹cs = c'#cs'›
    have upd-cs cs' xs' = [] by simp
    from IH2[OF ‹as = xs'@ys› this] show ?thesis .
  qed
 }
qed


lemma slpa-number-Calls-eq-number-Returns:
 ⟦same-level-path-aux cs as; upd-cs cs as = [];
  ∀ a ∈ set as. valid-edge a; ∀ c ∈ set cs. valid-edge c⟧
 ⟹ length [a←as@cs. ∃ Q r p fs. kind a = Q:r↪pfs] =
   length [a←as. ∃ Q p f. kind a = Q↩pf]
apply(induct rule:slpa-induct)
by(auto split:list.split edge-kind.split intro:only-call-get-return-edges
      simp:intra-kind-def)


lemma slpa-get-proc:
 ⟦same-level-path-aux cs as; upd-cs cs as = []; n −as→∗ n';
  ∀ c ∈ set cs. valid-edge c⟧
 ⟹ (if cs = [] then get-proc n else get-proc(last(sourcenodes cs))) = get-proc n'
proof(induct arbitrary:n rule:slpa-induct)
 case slpa-empty thus ?case by fastforce
next
 case (slpa-intra cs a as)
 note IH = ‹⋀n. ⟦upd-cs cs as = []; n −as→∗ n'; ∀ a∈set cs. valid-edge a⟧
  ⟹ (if cs = [] then get-proc n else get-proc (last (sourcenodes cs))) =
     get-proc n'›
 from ‹intra-kind (kind a)› ‹upd-cs cs (a#as) = []›
 have upd-cs cs as = [] by(cases kind a,auto simp:intra-kind-def)
 from ‹n −a#as→∗ n'› have n −[]@a#as→∗ n' by simp
 hence valid-edge a and n = sourcenode a and targetnode a −as→∗ n'
  by(fastforce dest:path-split)+
 from ‹valid-edge a› ‹intra-kind (kind a)› ‹ n = sourcenode a›
 have get-proc n = get-proc (targetnode a)
  by(fastforce intro:get-proc-intra)
 from IH[OF ‹upd-cs cs as = []› ‹targetnode a −as→∗ n'› ‹∀ a∈set cs. valid-edge
a›]
 have (if cs = [] then get-proc (targetnode a)
       else get-proc (last (sourcenodes cs))) = get-proc n' .
 with ‹get-proc n = get-proc (targetnode a)› show ?case by auto
next
 case (slpa-Call cs a as Q r p fs)
 note IH = ‹⋀n. ⟦upd-cs (a#cs) as = []; n −as→∗ n'; ∀ a∈set (a#cs). valid-edge
```

```
a⟧
  ⟹ (if a#cs = [] then get-proc n else get-proc (last (sourcenodes (a#cs)))) =
      get-proc n'⟩
 from ⟨kind a = Q:r↪pfs⟩ ⟨upd-cs cs (a#as) = []⟩
 have upd-cs (a#cs) as = [] by simp
 from ⟨n -a#as→* n'⟩ have n -[]@a#as→* n' by simp
 hence valid-edge a and n = sourcenode a and targetnode a -as→* n'
  by(fastforce dest:path-split)+
 from ⟨valid-edge a⟩ ⟨∀ a∈set cs. valid-edge a⟩ have ∀ a∈set (a#cs). valid-edge a
  by simp
 from IH[OF ⟨upd-cs (a#cs) as = []⟩ ⟨targetnode a -as→* n'⟩ this]
 have get-proc (last (sourcenodes (a#cs))) = get-proc n' by simp
 with ⟨n = sourcenode a⟩ show ?case by(cases cs,auto simp:sourcenodes-def)
next
 case (slpa-Return cs a as Q p f c' cs')
 note IH = ⟨⋀n. ⟦upd-cs cs' as = []; n -as→* n'; ∀ a∈set cs'. valid-edge a⟧
  ⟹ (if cs' = [] then get-proc n else get-proc (last (sourcenodes cs'))) =
     get-proc n'⟩
 from ⟨∀ a∈set cs. valid-edge a⟩ ⟨cs = c'#cs'⟩
 have valid-edge c' and ∀ a∈set cs'. valid-edge a by simp-all
 from ⟨kind a = Q↩pf⟩ ⟨upd-cs cs (a#as) = []⟩ ⟨cs = c'#cs'⟩
 have upd-cs cs' as = [] by simp
 from ⟨n -a#as→* n'⟩ have n -[]@a#as→* n' by simp
 hence n = sourcenode a and targetnode a -as→* n'
  by(fastforce dest:path-split)+
 from ⟨valid-edge c'⟩ ⟨a ∈ get-return-edges c'⟩
 have get-proc (sourcenode c') = get-proc (targetnode a)
  by(rule get-proc-get-return-edge)
 from IH[OF ⟨upd-cs cs' as = []⟩ ⟨targetnode a -as→* n'⟩ ⟨∀ a∈set cs'. valid-edge
a⟩]
 have (if cs' = [] then get-proc (targetnode a)
       else get-proc (last (sourcenodes cs'))) = get-proc n' .
 with ⟨cs = c'#cs'⟩ ⟨get-proc (sourcenode c') = get-proc (targetnode a)⟩
 show ?case by(auto simp:sourcenodes-def)
qed


lemma slpa-get-return-edges:
 ⟦same-level-path-aux cs as; cs ≠ []; upd-cs cs as = [];
 ∀ xs ys. as = xs@ys ∧ ys ≠ [] ⟶ upd-cs cs xs ≠ []⟧
 ⟹ last as ∈ get-return-edges (last cs)
proof(induct rule:slpa-induct)
 case (slpa-empty cs)
 from ⟨cs ≠ []⟩ ⟨upd-cs cs [] = []⟩ have False by fastforce
 thus ?case by simp
next
 case (slpa-intra cs a as)
 note IH = ⟨⟦cs ≠ []; upd-cs cs as = [];
           ∀ xs ys. as = xs@ys ∧ ys ≠ [] ⟶ upd-cs cs xs ≠ []⟧
```

```
    ⟹ last as ∈ get-return-edges (last cs)⟩
  show ?case
  proof(cases as = [])
    case True
    with ⟨intra-kind (kind a)⟩ ⟨upd-cs cs (a#as) = []⟩ have cs = []
      by(fastforce simp:intra-kind-def)
    with ⟨cs ≠ []⟩ have False by simp
    thus ?thesis by simp
  next
    case False
    from ⟨intra-kind (kind a)⟩ ⟨upd-cs cs (a#as) = []⟩ have upd-cs cs as = []
      by(fastforce simp:intra-kind-def)
    from ⟨∀ xs ys. a#as = xs@ys ∧ ys ≠ [] ⟶ upd-cs cs xs ≠ []⟩ ⟨intra-kind (kind
a)⟩
    have ∀ xs ys. as = xs@ys ∧ ys ≠ [] ⟶ upd-cs cs xs ≠ []
      apply(clarsimp,erule-tac x=a#xs in allE)
      by(auto simp:intra-kind-def)
    from IH[OF ⟨cs ≠ []⟩ ⟨upd-cs cs as = []⟩ this]
    have last as ∈ get-return-edges (last cs) .
    with False show ?thesis by simp
  qed
next
  case (slpa-Call cs a as Q r p fs)
  note IH = ⟨⟦a#cs ≠ []; upd-cs (a#cs) as = [];
    ∀ xs ys. as = xs@ys ∧ ys ≠ [] ⟶ upd-cs (a#cs) xs ≠ []⟧
    ⟹ last as ∈ get-return-edges (last (a#cs))⟩
  show ?case
  proof(cases as = [])
    case True
    with ⟨kind a = Q:r↪pfs⟩ ⟨upd-cs cs (a#as) = []⟩ have a#cs = [] by simp
    thus ?thesis by simp
  next
    case False
    from ⟨kind a = Q:r↪pfs⟩ ⟨upd-cs cs (a#as) = []⟩ have upd-cs (a#cs) as = []
      by simp
     from ⟨∀ xs ys. a#as = xs@ys ∧ ys ≠ [] ⟶ upd-cs cs xs ≠ []⟩ ⟨kind a =
Q:r↪pfs⟩
    have ∀ xs ys. as = xs@ys ∧ ys ≠ [] ⟶ upd-cs (a#cs) xs ≠ []
      by(clarsimp,erule-tac x=a#xs in allE,simp)
    from IH[OF - ⟨upd-cs (a#cs) as = []⟩ this]
    have last as ∈ get-return-edges (last (a#cs)) by simp
    with False ⟨cs ≠ []⟩ show ?thesis by(simp add:targetnodes-def)
  qed
next
  case (slpa-Return cs a as Q p f c' cs')
  note IH = ⟨⟦cs' ≠ []; upd-cs cs' as = [];
    ∀ xs ys. as = xs@ys ∧ ys ≠ [] ⟶ upd-cs cs' xs ≠ []⟧
    ⟹ last as ∈ get-return-edges (last cs')⟩
  show ?case
```

```
  proof(cases as = [])
    case True
    with ‹kind a = Q↩pf› ‹cs = c'#cs'› ‹upd-cs cs (a#as) = []›
    have cs' = [] by simp
    with ‹cs = c'#cs'› ‹a ∈ get-return-edges c'› True
    show ?thesis by simp
  next
    case False
    from ‹kind a = Q↩pf› ‹cs = c'#cs'› ‹upd-cs cs (a#as) = []›
    have upd-cs cs' as = [] by simp
    show ?thesis
    proof(cases cs' = [])
      case True
      with ‹cs = c'#cs'› ‹kind a = Q↩pf› have upd-cs cs [a] = [] by simp
       with ‹∀ xs ys. a#as = xs@ys ∧ ys ≠ [] ⟶ upd-cs cs xs ≠ []› False have
False
        apply(erule-tac x=[a] in allE) by fastforce
      thus ?thesis by simp
    next
      case False
      from ‹∀ xs ys. a#as = xs@ys ∧ ys ≠ [] ⟶ upd-cs cs xs ≠ []›
        ‹kind a = Q↩pf› ‹cs = c'#cs'›
      have ∀ xs ys. as = xs@ys ∧ ys ≠ [] ⟶ upd-cs cs' xs ≠ []
        by(clarsimp,erule-tac x=a#xs in allE,simp)
      from IH[OF False ‹upd-cs cs' as = []› this]
      have last as ∈ get-return-edges (last cs') .
     with ‹as ≠ []› False ‹cs = c'#cs'› show ?thesis by(simp add:targetnodes-def)
    qed
  qed
qed


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(atomize-elim)
 from assms show ∃ cfsx' cfx. transfers (kinds as) (cfsx@cf#cfs) = cfsx'@cfx#cfs
∧
    transfers (kinds as) (cfsx@cf#cfs') = cfsx'@cfx#cfs' ∧
    length cfsx' = length (upd-cs cs as)
  proof(induct arbitrary:cfsx cf rule:slpa-induct)
    case (slpa-empty cs) thus ?case by(simp add:kinds-def)
  next
    case (slpa-intra cs a as)
    note IH = ‹⋀cfsx cf. length cs = length cfsx ⟹
      ∃ cfsx' cfx. transfers (kinds as) (cfsx@cf#cfs) = cfsx'@cfx#cfs ∧
                   transfers (kinds as) (cfsx@cf#cfs') = cfsx'@cfx#cfs' ∧
```

```
                length cfsx' = length (upd-cs cs as)›
  from ‹intra-kind (kind a)›
  have length (upd-cs cs (a#as)) = length (upd-cs cs as)
    by(fastforce simp:intra-kind-def)
  show ?case
  proof(cases cfsx)
    case Nil
    with ‹length cs = length cfsx› have length cs = length [] by simp
    from Nil ‹intra-kind (kind a)›
    obtain cfx where transfer:transfer (kind a) (cfsx@cf#cfs) = []@cfx#cfs
      transfer (kind a) (cfsx@cf#cfs') = []@cfx#cfs'
      by(cases kind a,auto simp:kinds-def intra-kind-def)
    from IH[OF ‹length cs = length []›] obtain cfsx' cfx'
      where transfers (kinds as) ([]@cfx#cfs) = cfsx'@cfx'#cfs
      and transfers (kinds as) ([]@cfx#cfs') = cfsx'@cfx'#cfs'
      and length cfsx' = length (upd-cs cs as) by blast
    with ‹length (upd-cs cs (a#as)) = length (upd-cs cs as)› transfer
    show ?thesis by(fastforce simp:kinds-def)
  next
    case (Cons x xs)
    with ‹intra-kind (kind a)› obtain cfx'
      where transfer:transfer (kind a) (cfsx@cf#cfs) = (cfx'#xs)@cf#cfs
      transfer (kind a) (cfsx@cf#cfs') = (cfx'#xs)@cf#cfs'
      by(cases kind a,auto simp:kinds-def intra-kind-def)
    from ‹length cs = length cfsx› Cons have length cs = length (cfx'#xs)
      by simp
    from IH[OF this] obtain cfs'' cf''
      where transfers (kinds as) ((cfx'#xs)@cf#cfs) = cfs''@cf''#cfs
      and transfers (kinds as) ((cfx'#xs)@cf#cfs') = cfs''@cf''#cfs'
      and length cfs'' = length (upd-cs cs as) by blast
    with ‹length (upd-cs cs (a#as)) = length (upd-cs cs as)› transfer
    show ?thesis by(fastforce simp:kinds-def)
  qed
next
  case (slpa-Call cs a as Q r p fs)
  note IH = ⟨⋀cfsx cf. length (a#cs) = length cfsx ⟹
    ∃ cfsx' cfx. transfers (kinds as) (cfsx@cf#cfs) = cfsx'@cfx#cfs ∧
                 transfers (kinds as) (cfsx@cf#cfs') = cfsx'@cfx#cfs' ∧
                 length cfsx' = length (upd-cs (a#cs) as)⟩
  from ‹kind a = Q:r↪pfs›
 obtain cfx where transfer:transfer (kind a) (cfsx@cf#cfs) = (cfx#cfsx)@cf#cfs
    transfer (kind a) (cfsx@cf#cfs') = (cfx#cfsx)@cf#cfs'
    by(cases cfsx) auto
  from ‹length cs = length cfsx› have length (a#cs) = length (cfx#cfsx)
    by simp
  from IH[OF this] obtain cfsx' cfx'
    where transfers (kinds as) ((cfx#cfsx)@cf#cfs) = cfsx'@cfx'#cfs
    and transfers (kinds as) ((cfx#cfsx)@cf#cfs') = cfsx'@cfx'#cfs'
    and length cfsx' = length (upd-cs (a#cs) as) by blast
```

```
    with ‹kind a = Q:r↪pfs› transfer show ?case by(fastforce simp:kinds-def)
  next
    case (slpa-Return cs a as Q p f c' cs')
    note IH = ‹⋀cfsx cf. length cs' = length cfsx ⟹
      ∃ cfsx' cfx. transfers (kinds as) (cfsx@cf#cfs) = cfsx'@cfx#cfs ∧
                   transfers (kinds as) (cfsx@cf#cfs') = cfsx'@cfx#cfs' ∧
                   length cfsx' = length (upd-cs cs' as)›
    from ‹kind a = Q↩pf› ‹cs = c'#cs'›
    have length (upd-cs cs (a#as)) = length (upd-cs cs' as) by simp
    show ?case
    proof(cases cs')
      case Nil
      with ‹cs = c'#cs'› ‹length cs = length cfsx› obtain cfx
        where [simp]:cfsx = [cfx] by(cases cfsx) auto
      with ‹kind a = Q↩pf› obtain cf'
        where transfer:transfer (kind a) (cfsx@cf#cfs) = []@cf'#cfs
        transfer (kind a) (cfsx@cf#cfs') = []@cf'#cfs'
        by fastforce
      from Nil have length cs' = length [] by simp
      from IH[OF this] obtain cfsx' cfx'
        where transfers (kinds as) ([]@cf'#cfs) = cfsx'@cfx'#cfs
        and transfers (kinds as) ([]@cf'#cfs') = cfsx'@cfx'#cfs'
        and length cfsx' = length (upd-cs cs' as) by blast
      with ‹length (upd-cs cs (a#as)) = length (upd-cs cs' as)› transfer
      show ?thesis by(fastforce simp:kinds-def)
    next
      case (Cons cx csx)
      with ‹cs = c'#cs'› ‹length cs = length cfsx› obtain x x' xs
        where [simp]:cfsx = x#x'#xs and length xs = length csx
        by(cases cfsx,auto,case-tac list,fastforce+)
      with ‹kind a = Q↩pf› obtain cf'
       where transfer:transfer (kind a) ((x#x'#xs)@cf#cfs) = (cf'#xs)@cf#cfs
        transfer (kind a) ((x#x'#xs)@cf#cfs') = (cf'#xs)@cf#cfs'
        by fastforce
     from ‹cs = c'#cs'› ‹length cs = length cfsx› have length cs' = length (cf'#xs)
        by simp
      from IH[OF this] obtain cfsx' cfx
        where transfers (kinds as) ((cf'#xs)@cf#cfs) = cfsx'@cfx#cfs
        and transfers (kinds as) ((cf'#xs)@cf#cfs') = cfsx'@cfx#cfs'
        and length cfsx' = length (upd-cs cs' as) by blast
      with ‹length (upd-cs cs (a#as)) = length (upd-cs cs' as)› transfer
      show ?thesis by(fastforce simp:kinds-def)
    qed
  qed
qed


lemma slpa-snoc-intra:
  ⟦same-level-path-aux cs as; intra-kind (kind a)⟧
```

$\Longrightarrow$ *same-level-path-aux cs* (*as*@[*a*])
by(*induct rule*:*slpa-induct*,*auto simp*:*intra-kind-def*)

lemma *slpa-snoc-Call*:
⟦*same-level-path-aux cs as*; *kind a* = *Q*:*r*↪$_p$*fs*⟧
$\Longrightarrow$ *same-level-path-aux cs* (*as*@[*a*])
by(*induct rule*:*slpa-induct*,*auto simp*:*intra-kind-def*)

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*⟧
$\Longrightarrow$ *same-level-path-aux cs as* ∧ *upd-cs cs as* = []
proof(*induct arbitrary*:*m rule*:*vpa-induct*)
case (*vpa-empty cs*) **thus** *?case* **by** *simp*
next
case (*vpa-intra cs a as*)
note *IH* = ‹⋀*m*. ⟦*m* −*as*→∗ *m*′; *as* ≠ []; *valid-call-list cs m*; *get-proc m*′ = *Main*; *get-proc* (*case cs of* [] ⇒ *m* | *a* # *list* ⇒ *sourcenode* (*last cs*)) = *Main*⟧ $\Longrightarrow$ *same-level-path-aux cs as* ∧ *upd-cs cs as* = []›
from ‹*m* −*a* # *as*→∗ *m*′› **have** *sourcenode a* = *m* **and** *valid-edge a* **and** *targetnode a* −*as*→∗ *m*′ **by**(*auto elim*:*path-split-Cons*)
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
show *?case*
proof(*cases as* = [])
case *True*
with ‹*targetnode a* −*as*→∗ *m*′› **have** *targetnode a* = *m*′ **by** *fastforce*
with ‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›
‹*sourcenode a* = *m*› ‹*get-proc m*′ = *Main*›
have *get-proc m* = *Main* **by** *simp*
have *cs* = []
proof(*cases cs*)
case *Cons*
with ‹*valid-call-list cs m*›
obtain *c Q r p fs* **where** *valid-edge c* **and** *kind c* = *Q*:*r*↪$_{get\text{-}proc\ m}$*fs*
by(*auto simp*:*valid-call-list-def*,*erule-tac x*=[] **in** *allE*,
auto simp:*sourcenodes-def*)
with ‹*get-proc m* = *Main*› **have** *kind c* = *Q*:*r*↪$_{Main}$*fs* **by** *simp*
with ‹*valid-edge c*› **have** *False* **by**(*rule Main-no-call-target*)
thus *?thesis* **by** *simp*
qed *simp*
with *True* ‹*intra-kind* (*kind a*)› **show** *?thesis* **by**(*fastforce simp*:*intra-kind-def*)
next
case *False*
from ‹*valid-call-list cs m*› ‹*sourcenode a* = *m*›
‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›

```
    have valid-call-list cs (targetnode a)
      apply(clarsimp simp:valid-call-list-def)
      apply(erule-tac x=cs' in allE)
      apply(erule-tac x=c in allE)
      by(auto split:list.split)
    from ‹get-proc (case cs of [] ⇒ m | - ⇒ sourcenode (last cs)) = Main›
      ‹sourcenode a = m› ‹get-proc (sourcenode a) = get-proc (targetnode a)›
    have get-proc (case cs of [] ⇒ targetnode a | - ⇒ sourcenode (last cs)) = Main
      by(cases cs) auto
    from IH[OF ‹targetnode a −as→∗ m'› False ‹valid-call-list cs (targetnode a)›
      ‹get-proc m' = Main› this]
    have same-level-path-aux cs as ∧ upd-cs cs as = [] .
    with ‹intra-kind (kind a)› show ?thesis by(fastforce simp:intra-kind-def)
  qed
next
  case (vpa-Call cs a as Q r p fs)
  note IH = ‹⋀m. ⟦m −as→∗ m'; as ≠ []; valid-call-list (a # cs) m;
    get-proc m' = Main;
    get-proc (case a # cs of [] ⇒ m | - ⇒ sourcenode (last (a # cs))) = Main⟧
    ⟹ same-level-path-aux (a # cs) as ∧ upd-cs (a # cs) as = []›
  from ‹m −a # as→∗ m'› have sourcenode a = m and valid-edge a
    and targetnode a −as→∗ m' by(auto elim:path-split-Cons)
  from ‹valid-edge a› ‹kind a = Q:r↪pfs› have get-proc (targetnode a) = p
    by(rule get-proc-call)
  show ?case
  proof(cases as = [])
    case True
    with ‹targetnode a −as→∗ m'› have targetnode a = m' by fastforce
    with ‹get-proc (targetnode a) = p› ‹get-proc m' = Main› ‹kind a = Q:r↪pfs›
    have kind a = Q:r↪MainPfs by simp
    with ‹valid-edge a› have False by(rule Main-no-call-target)
    thus ?thesis by simp
  next
    case False
    from ‹get-proc (targetnode a) = p› ‹valid-call-list cs m› ‹valid-edge a›
      ‹kind a = Q:r↪pfs› ‹sourcenode a = m›
    have valid-call-list (a # cs) (targetnode a)
      apply(clarsimp simp:valid-call-list-def)
      apply(case-tac cs') apply auto
      apply(erule-tac x=list in allE)
      by(case-tac list)(auto simp:sourcenodes-def)
    from ‹get-proc (case cs of [] ⇒ m | - ⇒ sourcenode (last cs)) = Main›
      ‹sourcenode a = m›
    have get-proc (case a # cs of [] ⇒ targetnode a
      | - ⇒ sourcenode (last (a # cs))) = Main
      by(cases cs) auto
    from IH[OF ‹targetnode a −as→∗ m'› False ‹valid-call-list (a#cs) (targetnode
a)›
      ‹get-proc m' = Main› this]
```

```
    have same-level-path-aux (a # cs) as ∧ upd-cs (a # cs) as = [] .
    with ‹kind a = Q:r↪pfs› show ?thesis by simp
  qed
next
  case (vpa-ReturnEmpty cs a as Q p f)
  note IH = ‹⋀m. ⟦m −as→* m'; as ≠ []; valid-call-list [] m; get-proc m' = Main;
    get-proc (case [] of [] ⇒ m | a # list ⇒ sourcenode (last [])) = Main⟧
    ⟹ same-level-path-aux [] as ∧ upd-cs [] as = []›
  from ‹m −a # as→* m'› have sourcenode a = m and valid-edge a
    and targetnode a −as→* m' by(auto elim:path-split-Cons)
  from ‹valid-edge a› ‹kind a = Q↩pf› have get-proc (sourcenode a) = p
    by(rule get-proc-return)
  from ‹get-proc (case cs of [] ⇒ m | a # list ⇒ sourcenode (last cs)) = Main›
    ‹cs = []›
  have get-proc m = Main by simp
  with ‹sourcenode a = m› ‹get-proc (sourcenode a) = p› have p = Main by simp
  with ‹kind a = Q↩pf› have kind a = Q↩Main f by simp
  with ‹valid-edge a› have False by(rule Main-no-return-source)
  thus ?case by simp
next
  case (vpa-ReturnCons cs a as Q p f c' cs')
  note IH = ‹⋀m. ⟦m −as→* m'; as ≠ []; valid-call-list cs' m; get-proc m' =
Main;
    get-proc (case cs' of [] ⇒ m | a # list ⇒ sourcenode (last cs')) = Main⟧
    ⟹ same-level-path-aux cs' as ∧ upd-cs cs' as = []›
  from ‹m −a # as→* m'› have sourcenode a = m and valid-edge a
    and targetnode a −as→* m' by(auto elim:path-split-Cons)
  from ‹valid-edge a› ‹kind a = Q↩pf› have get-proc (sourcenode a) = p
    by(rule get-proc-return)
  from ‹valid-call-list cs m› ‹cs = c' # cs'›
  have valid-edge c'
    by(auto simp:valid-call-list-def,erule-tac x=[] in allE,auto)
  from ‹valid-edge c'› ‹a ∈ get-return-edges c'›
  have get-proc (sourcenode c') = get-proc (targetnode a)
    by(rule get-proc-get-return-edge)
  show ?case
  proof(cases as = [])
    case True
    with ‹targetnode a −as→* m'› have targetnode a = m' by fastforce
    with ‹get-proc m' = Main› have get-proc (targetnode a) = Main by simp
    from ‹get-proc (sourcenode c') = get-proc (targetnode a)›
      ‹get-proc (targetnode a) = Main›
    have get-proc (sourcenode c') = Main by simp
    have cs' = []
    proof(cases cs')
      case (Cons cx csx)
      with ‹cs = c' # cs'› ‹valid-call-list cs m›
      obtain Qx rx fsx where valid-edge cx
        and kind cx = Qx:rx↪get-proc (sourcenode c')fsx
```

by(*auto simp*:*valid-call-list-def*,*erule-tac x*=[*c'*] **in** *allE*,
auto simp:*sourcenodes-def*)
with ‹*get-proc* (*sourcenode c'*) = *Main*› **have** *kind cx* = *Qx*:*rx*↪$_{Main}$*fsx* **by** *simp*
with ‹*valid-edge cx*› **have** *False* **by**(*rule Main-no-call-target*)
thus *?thesis* **by** *simp*
qed *simp*
with *True* ‹*cs* = *c'* # *cs'*› ‹*a* ∈ *get-return-edges c'*› ‹*kind a* = *Q*↩$_p$*f*›
show *?thesis* **by** *simp*
next
case *False*
from ‹*valid-call-list cs m*› ‹*cs* = *c'* # *cs'*›
‹*get-proc* (*sourcenode c'*) = *get-proc* (*targetnode a*)›
have *valid-call-list cs'* (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*c'* # *cs'* **in** *allE*)
by(*case-tac cs'*)(*auto simp*:*sourcenodes-def*)
from ‹*get-proc* (*case cs of* [] ⇒ *m* | *a* # *list* ⇒ *sourcenode* (*last cs*)) = *Main*›
‹*cs* = *c'* # *cs'*› ‹*get-proc* (*sourcenode c'*) = *get-proc* (*targetnode a*)›
have *get-proc* (*case cs' of* [] ⇒ *targetnode a*
| - ⇒ *sourcenode* (*last cs'*)) = *Main*
by(*cases cs'*) *auto*
from *IH*[*OF* ‹*targetnode a* −*as*→∗ *m'*› *False* ‹*valid-call-list cs'* (*targetnode a*)›
‹*get-proc m'* = *Main*› *this*]
have *same-level-path-aux cs' as* ∧ *upd-cs cs' as* = [] **.**
with ‹*kind a* = *Q*↩$_p$*f*› ‹*cs* = *c'* # *cs'*› ‹*a* ∈ *get-return-edges c'*›
show *?thesis* **by** *simp*
qed
qed

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*
by(*fastforce intro*:*same-level-path-aux-valid-path-aux*
simp:*same-level-path-def valid-path-def*)

lemma *same-level-path-Append*:
⟦*same-level-path as*; *same-level-path as'*⟧ ⟹ *same-level-path* (*as*@*as'*)
by(*fastforce elim*:*same-level-path-aux-Append upd-cs-Append simp*:*same-level-path-def*)

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↪$_p$*fs*] = *length* [a←*as*. ∃ *Q p f*. *kind a* = *Q*↩$_p$*f*]
by(*fastforce dest*:*slpa-number-Calls-eq-number-Returns simp*:*same-level-path-def*)

lemma *same-level-path-valid-path-Append*:
⟦*same-level-path as*; *valid-path as'*⟧ ⟹ *valid-path* (*as*@*as'*)
by(*fastforce intro*:*valid-path-aux-Append elim*:*same-level-path-aux-valid-path-aux simp*:*valid-path-def same-level-path-def*)

lemma *valid-path-same-level-path-Append*:
⟦*valid-path as*; *same-level-path as'*⟧ ⟹ *valid-path* (*as*@*as'*)
apply(*auto simp*:*valid-path-def same-level-path-def*)
apply(*erule valid-path-aux-Append*)
by(*fastforce intro*!:*same-level-path-aux-valid-path-aux dest*:*same-level-path-aux-callstack-Append*)

lemma *intras-same-level-path*:
assumes ∀ *a* ∈ *set as*. *intra-kind*(*kind a*) **shows** *same-level-path as*
proof −
from ‹∀ *a* ∈ *set as*. *intra-kind*(*kind a*)› **have** *same-level-path-aux* [] *as*
by(*induct as*)(*auto simp*:*intra-kind-def*)
moreover
from ‹∀ *a* ∈ *set as*. *intra-kind*(*kind a*)› **have** *upd-cs* [] *as* = []
by(*induct as*)(*auto simp*:*intra-kind-def*)
ultimately show *?thesis* **by**(*simp add*:*same-level-path-def*)
qed

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'
by(*fastforce intro*:*same-level-path-valid-path simp*:*slp-def vp-def*)

lemma *intra-path-slp*: n −*as*→$_\iota$* n' ⟹ n −*as*→$_{sl}$* n'
by(*fastforce intro*:*intras-same-level-path simp*:*slp-def intra-path-def*)

lemma *slp-Append*:
⟦n −*as*→$_{sl}$* n''; n'' −*as'*→$_{sl}$* n'⟧ ⟹ n −*as*@*as'*→$_{sl}$* n'
by(*fastforce simp*:*slp-def intro*:*path-Append same-level-path-Append*)

lemma *slp-vp-Append*:
⟦n −*as*→$_{sl}$* n''; n'' −*as'*→$_{\surd}$* n'⟧ ⟹ n −*as*@*as'*→$_{\surd}$* n'
by(*fastforce simp*:*slp-def vp-def intro*:*path-Append same-level-path-valid-path-Append*)

lemma *vp-slp-Append*:
 $[\![n\ -as\rightarrow_{\surd}*\ n''; n''\ -as'\rightarrow_{sl}*\ n']\!] \Longrightarrow n\ -as@as'\rightarrow_{\surd}*\ n'$
 by(*fastforce simp*:*slp-def vp-def intro*:*path-Append valid-path-same-level-path-Append*)

lemma *slp-get-proc*:
 $n\ -as\rightarrow_{sl}*\ n' \Longrightarrow$ *get-proc* n = *get-proc* n'
by(*fastforce dest*:*slpa-get-proc simp*:*same-level-path-def slp-def*)

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(*atomize-elim*)
 from ‹$n\ -as\rightarrow_{sl}*\ n'$› **have** $n\ -as\rightarrow*\ n'$ **and** *same-level-path* as
  by(*simp-all add*:*slp-def*)
 from ‹*same-level-path* as› **have** *same-level-path-aux* [] as **and** *upd-cs* [] as = []
  by(*simp-all add*:*same-level-path-def*)
 from ‹$n\ -as\rightarrow*\ n'$› ‹*same-level-path-aux* [] as› ‹*upd-cs* [] as = []›
 show $\exists as'.\ n\ -as'\rightarrow_{\iota}*\ n' \wedge$ *set*(*sourcenodes* as') $\subseteq$ *set*(*sourcenodes* as)
 proof(*induct as arbitrary*:*n rule*:*length-induct*)
  fix $as\ n$
  assume *IH*:$\forall as''.$ *length* $as'' <$ *length* $as \longrightarrow$
   $(\forall n''.\ n''\ -as''\rightarrow*\ n' \longrightarrow$ *same-level-path-aux* [] $as'' \longrightarrow$
    upd-cs [] $as'' =$ [] $\longrightarrow$
    $(\exists as'.\ n''\ -as'\rightarrow_{\iota}*\ n' \wedge$ *set* (*sourcenodes* as') $\subseteq$ *set* (*sourcenodes* as'')))
   and $n\ -as\rightarrow*\ n'$ **and** *same-level-path-aux* [] as **and** *upd-cs* [] as = []
  show $\exists as'.\ n\ -as'\rightarrow_{\iota}*\ n' \wedge$ *set* (*sourcenodes* as') $\subseteq$ *set* (*sourcenodes* as)
  proof(*cases as*)
   case *Nil*
   with ‹$n\ -as\rightarrow*\ n'$› **show** *?thesis* **by**(*fastforce simp*:*intra-path-def*)
  next
   case (*Cons* $a'\ as'$)
   with ‹$n\ -as\rightarrow*\ n'$› *Cons* **have** n = *sourcenode* a' **and** *valid-edge* a'
    and *targetnode* $a'\ -as'\rightarrow*\ n'$
    by(*auto intro*:*path-split-Cons*)
   show *?thesis*
   proof(*cases kind* a' *rule*:*edge-kind-cases*)
    case *Intra*
    with *Cons* ‹*same-level-path-aux* [] as› **have** *same-level-path-aux* [] as'
     by(*fastforce simp*:*intra-kind-def*)
    moreover
    from *Intra Cons* ‹*upd-cs* [] as = []› **have** *upd-cs* [] as' = []
     by(*fastforce simp*:*intra-kind-def*)
    ultimately obtain as'' **where** *targetnode* $a'\ -as''\rightarrow_{\iota}*\ n'$
     and *set* (*sourcenodes* as'') $\subseteq$ *set* (*sourcenodes* as')

using *IH Cons* ‹*targetnode a′ −as′→∗ n′*›
by(*erule-tac x=as′* **in** *allE*) *auto*
from ‹*n = sourcenode a′*› ‹*valid-edge a′*› *Intra* ‹*targetnode a′ −as′′→ι∗ n′*›
have *n −a′#as′′→ι∗ n′* **by**(*fastforce intro*:*Cons-path simp*:*intra-path-def*)
with ‹*set* (*sourcenodes as′′*) ⊆ *set* (*sourcenodes as′*)› *Cons* **show** *?thesis*
by(*rule-tac x=a′#as′′* **in** *exI*,*auto simp*:*sourcenodes-def*)
next
case (*Call Q p f*)
with *Cons* ‹*same-level-path-aux* [] *as*›
have *same-level-path-aux* [*a′*] *as′* **by** *simp*
from *Call Cons* ‹*upd-cs* [] *as* = []› **have** *upd-cs* [*a′*] *as′* = [] **by** *simp*
hence *as′* ≠ [] **by** *fastforce*
with ‹*upd-cs* [*a′*] *as′* = []› **obtain** *xs ys* **where** *as′* = *xs*@*ys* **and** *xs* ≠ []
and *upd-cs* [*a′*] *xs* = [] **and** *upd-cs* [] *ys* = []
and ∀ *xs′ ys′*. *xs* = *xs′*@*ys′* ∧ *ys′* ≠ [] ⟶ *upd-cs* [*a′*] *xs′* ≠ []
by −(*erule upd-cs-empty-split*,*auto*)
from ‹*same-level-path-aux* [*a′*] *as′*› ‹*as′* = *xs*@*ys*› ‹*upd-cs* [*a′*] *xs* = []›
have *same-level-path-aux* [*a′*] *xs* **and** *same-level-path-aux* [] *ys*
by(*auto intro*:*slpa-split*)
from ‹*same-level-path-aux* [*a′*] *xs*› ‹*upd-cs* [*a′*] *xs* = []›
‹∀ *xs′ ys′*. *xs* = *xs′*@*ys′* ∧ *ys′* ≠ [] ⟶ *upd-cs* [*a′*] *xs′* ≠ []›
have *last xs* ∈ *get-return-edges* (*last* [*a′*])
by(*fastforce intro*!:*slpa-get-return-edges*)
with ‹*valid-edge a′*› *Call*
obtain *a* **where** *valid-edge a* **and** *sourcenode a* = *sourcenode a′*
and *targetnode a* = *targetnode* (*last xs*) **and** *kind a* = $(\lambda cf.\ False)_{\surd}$
by −(*drule call-return-node-edge*,*auto*)
from ‹*targetnode a* = *targetnode* (*last xs*)› ‹*xs* ≠ []›
have *targetnode a* = *targetnode* (*last* (*a′#xs*)) **by** *simp*
from ‹*as′* = *xs*@*ys*› ‹*xs* ≠ []› *Cons* **have** *length ys* < *length as* **by** *simp*
from ‹*targetnode a′ −as′→∗ n′*› ‹*as′* = *xs*@*ys*› ‹*xs* ≠ []›
have *targetnode* (*last* (*a′#xs*)) *−ys→∗ n′*
by(*cases xs rule*:*rev-cases*,*auto dest*:*path-split*)
with *IH* ‹*length ys* < *length as*› ‹*same-level-path-aux* [] *ys*›
‹*upd-cs* [] *ys* = []›
obtain *as′′* **where** *targetnode* (*last* (*a′#xs*)) *−as′′→ι∗ n′*
and *set*(*sourcenodes as′′*) ⊆ *set*(*sourcenodes ys*)
apply(*erule-tac x=ys* **in** *allE*) **apply** *clarsimp*
apply(*erule-tac x=targetnode* (*last* (*a′#xs*)) **in** *allE*)
by *clarsimp*
from ‹*sourcenode a* = *sourcenode a′*› ‹*n* = *sourcenode a′*›
‹*targetnode a* = *targetnode* (*last* (*a′#xs*))› ‹*valid-edge a*›
‹*kind a* = $(\lambda cf.\ False)_{\surd}$› ‹*targetnode* (*last* (*a′#xs*)) *−as′′→ι∗ n′*›
have *n −a#as′′→ι∗ n′*
by(*fastforce intro*:*Cons-path simp*:*intra-path-def intra-kind-def*)
moreover
from ‹*set*(*sourcenodes as′′*) ⊆ *set*(*sourcenodes ys*)› *Cons* ‹*as′* = *xs*@*ys*›
‹*sourcenode a* = *sourcenode a′*›
have *set*(*sourcenodes* (*a#as′′*)) ⊆ *set*(*sourcenodes as*)

```
      by(auto simp:sourcenodes-def)
    ultimately show ?thesis by blast
   next
    case (Return Q p f)
    with ⟨same-level-path-aux [] as⟩ have False by simp
    thus ?thesis by simp
   qed
  qed
 qed
qed


lemma slp-callstack-length-equal:
  assumes n −as→sl* n' obtains cf' where transfers (kinds as) (cf#cfs) =
cf'#cfs
  and transfers (kinds as) (cf#cfs') = cf'#cfs'
proof(atomize-elim)
  from ⟨n −as→sl* n'⟩ have same-level-path-aux [] as and upd-cs [] as = []
    by(simp-all add:slp-def same-level-path-def)
  then obtain cfx cfsx where transfers (kinds as) (cf#cfs) = cfsx@cfx#cfs
    and transfers (kinds as) (cf#cfs') = cfsx@cfx#cfs'
    and length cfsx = length (upd-cs [] as)
    by(fastforce elim:slpa-callstack-length)
  with ⟨upd-cs [] as = []⟩ have cfsx = [] by(cases cfsx) auto
  with ⟨transfers (kinds as) (cf#cfs) = cfsx@cfx#cfs⟩
    ⟨transfers (kinds as) (cf#cfs') = cfsx@cfx#cfs'⟩
  show ∃ cf'. transfers (kinds as) (cf#cfs) = cf'#cfs ∧
    transfers (kinds as) (cf#cfs') = cf'#cfs' by fastforce
qed


lemma slp-cases [consumes 1,case-names intra-path return-intra-path]:
  assumes m −as→sl* m'
  obtains m −as→ι* m'
  | as' a as'' Q p f where as = as'@a#as'' and kind a = Q↩pf
  and m −as'@[a]→sl* targetnode a and targetnode a −as''→ι* m'
proof(atomize-elim)
  from ⟨m −as→sl* m'⟩ have m −as→* m' and same-level-path-aux [] as
    and upd-cs [] as = [] by(simp-all add:slp-def same-level-path-def)
  from ⟨m −as→* m'⟩ have ∀ a ∈ set as. valid-edge a by(rule path-valid-edges)
  have ∀ a ∈ set []. valid-edge a by simp
  with ⟨same-level-path-aux [] as⟩ ⟨upd-cs [] as = []⟩ ⟨∀ a ∈ set []. valid-edge a⟩
    ⟨∀ a ∈ set as. valid-edge a⟩
  show m −as→ι* m' ∨
    (∃ as' a as'' Q p f. as = as' @ a # as'' ∧ kind a = Q↩pf ∧
    m −as' @ [a]→sl* targetnode a ∧ targetnode a −as''→ι* m')
  proof(cases rule:slpa-cases)
    case intra-path
    with ⟨m −as→* m'⟩ have m −as→ι* m' by(simp add:intra-path-def)
```

thus *?thesis* **by** *blast*
next
case (*return-intra-path as′ a as″ Q p f c′ cs′*)
from ‹*m* −*as*→∗ *m′*› ‹*as* = *as′* @ *a* # *as″*›
have *m* −*as′*→∗ *sourcenode a* **and** *valid-edge a* **and** *targetnode a* −*as″*→∗ *m′*
by(*auto intro*:*path-split*)
from ‹*m* −*as′*→∗ *sourcenode a*› ‹*valid-edge a*›
have *m* −*as′*@[*a*]→∗ *targetnode a* **by**(*fastforce intro*:*path-Append path-edge*)
with ‹*same-level-path-aux* [] *as′*› ‹*upd-cs* [] *as′* = *c′* # *cs′*› ‹*kind a* = *Q*↩*pf*›
‹*a* ∈ *get-return-edges c′*›
have *same-level-path-aux* [] (*as′*@[*a*])
by(*fastforce intro*:*same-level-path-aux-Append*)
with ‹*upd-cs* [] (*as′* @ [*a*]) = []› ‹*m* −*as′*@[*a*]→∗ *targetnode a*›
have *m* −*as′*@[*a*]→*sl*∗ *targetnode a* **by**(*simp add*:*slp-def same-level-path-def*)
moreover
from ‹∀ *a*∈*set as″*. *intra-kind* (*kind a*)› ‹*targetnode a* −*as″*→∗ *m′*›
have *targetnode a* −*as″*→*ι*∗ *m′* **by**(*simp add*:*intra-path-def*)
ultimately show *?thesis* **using** ‹*as* = *as′* @ *a* # *as″*› ‹*kind a* = *Q*↩*pf*› **by** *blast*
qed
qed

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*↩*pf* ⇒ *same-level-path-rev-aux* (*a*#*cs*) *as*
| *Q*:*r*↪*pfs* ⇒ *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*)
by *auto*(*case-tac b rule*:*rev-cases*,*auto*)
termination by *lexicographic-order*

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*↩*pf*; *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*↪*pfs*; *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*
using *major*

apply(*induct ys arbitrary*: *xs rule*:*rev-induct*)
by(*auto intro*:*rules split*:*edge-kind.split-asm list.split-asm simp*:*intra-kind-def*)

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'*)
by(*induct rule*:*slpra-induct*,
auto simp:*intra-kind-def simp del*:*append-assoc simp*:*append-assoc*[*THEN sym*])

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(*induct arbitrary*:*n'* *rule*:*slpra-induct*)
case *slpra-empty* **thus** *?case* **by** *simp*
next
case (*slpra-intra cs a as*)
note *IH* = ‹⋀*n'*. ⟦*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* = []›
from ‹*n* −*as*@[*a*]→∗ *n'*› **have** *n* −*as*→∗ *sourcenode a* **and** *valid-edge a*
and *n'* = *targetnode a* **by**(*auto intro*:*path-split-snoc*)
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)
by(*rule get-proc-intra*)
with ‹*valid-return-list cs n'*› ‹*n'* = *targetnode a*›
have *valid-return-list cs* (*sourcenode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*)(*auto simp*:*targetnodes-def*)
from ‹*upd-rev-cs cs* (*as*@[*a*]) = []› ‹*intra-kind* (*kind a*)›
have *upd-rev-cs cs as* = [] **by**(*fastforce simp*:*intra-kind-def*)
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
from *IH*[*OF* ‹*upd-rev-cs cs as* = []› ‹*n* −*as*→∗ *sourcenode a*›
‹*valid-return-list cs* (*sourcenode a*)›]
have *same-level-path-aux* [] *as*
and *same-level-path-aux* (*upd-cs* [] *as*) *cs*
and *upd-cs* (*upd-cs* [] *as*) *cs* = [] **by** *simp-all*
from ‹*same-level-path-aux* [] *as*› ‹*intra-kind* (*kind a*)›
have *same-level-path-aux* [] (*as*@[*a*]) **by**(*rule slpa-snoc-intra*)
from ‹*intra-kind* (*kind a*)›
have *upd-cs* [] (*as*@[*a*]) = *upd-cs* [] *as*
by(*fastforce simp*:*upd-cs-Append intra-kind-def*)
moreover
from ‹*same-level-path-aux* [] *as*› ‹*intra-kind* (*kind a*)›

```
  have same-level-path-aux [] (as@[a]) by(rule slpa-snoc-intra)
  ultimately show ?case using ‹same-level-path-aux (upd-cs [] as) cs›
    ‹upd-cs (upd-cs [] as) cs = []›
    by simp
next
  case (slpra-Return cs a as Q p f)
  note IH = ‹⋀n' n''. ⟦upd-rev-cs (a#cs) as = []; n -as→* n';
    valid-return-list (a#cs) n'⟧
  ⟹ same-level-path-aux [] as ∧
     same-level-path-aux (upd-cs [] as) (a#cs) ∧
     upd-cs (upd-cs [] as) (a#cs) = []›
  from ‹n -as@[a]→* n'› have n -as→* sourcenode a and valid-edge a
    and n' = targetnode a by(auto intro:path-split-snoc)
  from ‹valid-edge a› ‹kind a = Q↩pf› have p = get-proc (sourcenode a)
     by(rule get-proc-return[THEN sym])
   from ‹valid-return-list cs n'› ‹n' = targetnode a›
   have valid-return-list cs (targetnode a) by simp
   with ‹valid-edge a› ‹kind a = Q↩pf› ‹p = get-proc (sourcenode a)›
   have valid-return-list (a#cs) (sourcenode a)
     apply(clarsimp simp:valid-return-list-def)
     apply(case-tac cs') apply auto
     apply(erule-tac x=list in allE) apply clarsimp
     by(case-tac list,auto simp:targetnodes-def)
   from ‹upd-rev-cs cs (as@[a]) = []› ‹kind a = Q↩pf›
   have upd-rev-cs (a#cs) as = [] by simp
   from IH[OF this ‹n -as→* sourcenode a› ‹valid-return-list (a#cs) (sourcenode
a)›]
   have same-level-path-aux [] as
     and same-level-path-aux (upd-cs [] as) (a#cs)
     and upd-cs (upd-cs [] as) (a#cs) = [] by simp-all
   show ?case
  proof(cases upd-cs [] as)
    case Nil
    with ‹kind a = Q↩pf› ‹same-level-path-aux (upd-cs [] as) (a#cs)›
    have False by simp
    thus ?thesis by simp
  next
    case (Cons cx csx)
    with ‹kind a = Q↩pf› ‹same-level-path-aux (upd-cs [] as) (a#cs)›
    obtain Qx fx
      where match:a ∈ get-return-edges cx same-level-path-aux csx cs by auto
    from ‹kind a = Q↩pf› Cons have upd-cs [] (as@[a]) = csx
      by(rule upd-cs-snoc-Return-Cons)
    with ‹same-level-path-aux (upd-cs [] as) (a#cs)›
      ‹kind a = Q↩pf› match
    have same-level-path-aux (upd-cs [] (as@[a])) cs by simp
    from ‹upd-cs [] (as@[a]) = csx› ‹kind a = Q↩pf› Cons
      ‹upd-cs (upd-cs [] as) (a#cs) = []›
    have upd-cs (upd-cs [] (as@[a])) cs = [] by simp
```

from *Cons* ‹*kind a = Q↪pf*› *match*
have *same-level-path-aux* (*upd-cs* [] *as*) [*a*] **by** *simp*
with ‹*same-level-path-aux* [] *as*› **have** *same-level-path-aux* [] (*as*@[*a*])
by(*rule same-level-path-aux-Append*)
with ‹*same-level-path-aux* (*upd-cs* [] (*as*@[*a*])) *cs*›
‹*upd-cs* (*upd-cs* [] (*as*@[*a*])) *cs* = []›
show *?thesis* **by** *simp*
qed
next
case (*slpra-Call cs a as Q r p fs cx csx*)
note *IH* = ‹⋀*n'*. ⟦*upd-rev-cs csx as* = []; *n* −*as*→∗ *n'*; *valid-return-list csx n'*⟧
⟹ *same-level-path-aux* [] *as* ∧
same-level-path-aux (*upd-cs* [] *as*) *csx* ∧ *upd-cs* (*upd-cs* [] *as*) *csx* = []›
note *match* = ‹*cs* = *cx*#*csx*› ‹*cx* ∈ *get-return-edges a*›
from ‹*n* −*as*@[*a*]→∗ *n'*› **have** *n* −*as*→∗ *sourcenode a* **and** *valid-edge a*
and *n'* = *targetnode a* **by**(*auto intro*:*path-split-snoc*)
from ‹*valid-edge a*› *match*
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode cx*)
by(*fastforce intro*:*get-proc-get-return-edge*)
with ‹*valid-return-list cs n'*› ‹*cs* = *cx*#*csx*›
have *valid-return-list csx* (*sourcenode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*cx*#*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*,*auto simp*:*targetnodes-def*)
from ‹*kind a* = *Q*:*r*↪*pfs*› *match* ‹*upd-rev-cs cs* (*as*@[*a*]) = []›
have *upd-rev-cs csx as* = [] **by** *simp*
from *IH*[*OF this* ‹*n* −*as*→∗ *sourcenode a*› ‹*valid-return-list csx* (*sourcenode a*)›]
have *same-level-path-aux* [] *as*
and *same-level-path-aux* (*upd-cs* [] *as*) *csx* **and** *upd-cs* (*upd-cs* [] *as*) *csx* = []
by *simp-all*
from ‹*same-level-path-aux* [] *as*› ‹*kind a* = *Q*:*r*↪*pfs*›
have *same-level-path-aux* [] (*as*@[*a*]) **by**(*rule slpa-snoc-Call*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*pfs*› *match* **obtain** *Q' f'* **where** *kind cx* = *Q'*↩*pf'*
by(*fastforce dest*!:*call-return-edges*)
from ‹*kind a* = *Q*:*r*↪*pfs*› **have** *upd-cs* [] (*as*@[*a*]) = *a*#(*upd-cs* [] *as*)
by(*rule upd-cs-snoc-Call*)
with ‹*same-level-path-aux* (*upd-cs* [] *as*) *csx*› ‹*kind a* = *Q*:*r*↪*pfs*›
‹*kind cx* = *Q'*↩*pf'*› *match*
have *same-level-path-aux* (*upd-cs* [] (*as*@[*a*])) *cs* **by** *simp*
from ‹*upd-cs* (*upd-cs* [] *as*) *csx* = []› ‹*upd-cs* [] (*as*@[*a*]) = *a*#(*upd-cs* [] *as*)›
‹*kind a* = *Q*:*r*↪*pfs*› ‹*kind cx* = *Q'*↩*pf'*› *match*
have *upd-cs* (*upd-cs* [] (*as*@[*a*])) *cs* = [] **by** *simp*
with ‹*same-level-path-aux* [] (*as*@[*a*])›
‹*same-level-path-aux* (*upd-cs* [] (*as*@[*a*])) *cs*› **show** *?case* **by** *simp*
qed

Lemmas on paths with (*-Entry-*)

lemma *path-Entry-target* [*dest*]:
assumes $n\ -as\rightarrow*\ (\text{-}Entry\text{-})$
shows $n = (\text{-}Entry\text{-})$ **and** $as = []$
using ‹$n\ -as\rightarrow*\ (\text{-}Entry\text{-})$›
proof(*induct n as n'*≡(*-Entry-*) *rule*:*path.induct*)
case (*Cons-path n'' as a n*)
from ‹$n'' = (\text{-}Entry\text{-})$› ‹*targetnode a* = *n''*› ‹*valid-edge a*› **have** *False*
by −(*rule Entry-target*,*simp-all*)
{ case *1*
from ‹*False*› **show** *?case* **..**
next
case *2*
from ‹*False*› **show** *?case* **..**
}
qed *simp-all*

lemma *Entry-sourcenode-hd*:
assumes $n\ -as\rightarrow*\ n'$ **and** $(\text{-}Entry\text{-}) \in set\ (sourcenodes\ as)$
shows $n = (\text{-}Entry\text{-})$ **and** $(\text{-}Entry\text{-}) \notin set\ (sourcenodes\ (tl\ as))$
using ‹$n\ -as\rightarrow*\ n'$› ‹$(\text{-}Entry\text{-}) \in set\ (sourcenodes\ as)$›
proof(*induct rule*:*path.induct*)
case (*empty-path n*) **case** *1*
thus *?case* **by**(*simp add*:*sourcenodes-def*)
next
case (*empty-path n*) **case** *2*
thus *?case* **by**(*simp add*:*sourcenodes-def*)
next
case (*Cons-path n'' as n' a n*)
note *IH1* = ‹$(\text{-}Entry\text{-}) \in set(sourcenodes\ as) \Longrightarrow n'' = (\text{-}Entry\text{-})$›
note *IH2* = ‹$(\text{-}Entry\text{-}) \in set(sourcenodes\ as) \Longrightarrow (\text{-}Entry\text{-}) \notin set(sourcenodes(tl\ as))$›
have $(\text{-}Entry\text{-}) \notin set\ (sourcenodes(tl(a\#as)))$
proof(*rule ccontr*)
assume ¬ $(\text{-}Entry\text{-}) \notin set\ (sourcenodes\ (tl\ (a\#as)))$
hence $(\text{-}Entry\text{-}) \in set\ (sourcenodes\ as)$ **by** *simp*
from *IH1*[*OF this*] **have** $n'' = (\text{-}Entry\text{-})$ **by** *simp*
with ‹*targetnode a* = *n''*› ‹*valid-edge a*› **show** *False* **by** −(*erule Entry-target*,*simp*)
qed
hence $(\text{-}Entry\text{-}) \notin set\ (sourcenodes(tl(a\#as)))$ **by** *fastforce*
{ case *1*
with ‹$(\text{-}Entry\text{-}) \notin set\ (sourcenodes(tl(a\#as)))$› ‹*sourcenode a* = *n*›
show *?case* **by**(*simp add*:*sourcenodes-def*)
next
case *2*
with ‹$(\text{-}Entry\text{-}) \notin set\ (sourcenodes(tl(a\#as)))$› ‹*sourcenode a* = *n*›
show *?case* **by**(*simp add*:*sourcenodes-def*)

```
  }
qed


lemma Entry-no-inner-return-path:
  assumes (-Entry-) -as@[a]→* n and ∀ a ∈ set as. intra-kind(kind a)
  and kind a = Q↩pf
  shows False
proof −
  from ‹(-Entry-) -as@[a]→* n› have (-Entry-) -as→* sourcenode a
    and valid-edge a and targetnode a = n by(auto intro:path-split-snoc)
  from ‹(-Entry-) -as→* sourcenode a› ‹∀ a ∈ set as. intra-kind(kind a)›
  have (-Entry-) -as→ι* sourcenode a by(simp add:intra-path-def)
  hence get-proc (sourcenode a) = Main
    by(fastforce dest:intra-path-get-procs simp:get-proc-Entry)
  with ‹valid-edge a› ‹kind a = Q↩pf› have p = Main
    by(fastforce dest:get-proc-return)
  with ‹valid-edge a› ‹kind a = Q↩pf› show ?thesis
    by(fastforce intro:Main-no-return-source)
qed


lemma vpra-no-slpra:
  ⟦valid-path-rev-aux cs as; n −as→* n'; valid-return-list cs n'; cs ≠ [];
    ∀ xs ys. as = xs@ys ⟶ (¬ same-level-path-rev-aux cs ys ∨ upd-rev-cs cs ys ≠
[])⟧
  ⟹ ∃ a Q f. valid-edge a ∧ kind a = Q↩get-proc nf
proof(induct arbitrary:n' rule:vpra-induct)
  case (vpra-empty cs)
  from ‹valid-return-list cs n'› ‹cs ≠ []› obtain Q f where valid-edge (hd cs)
    and kind (hd cs) = Q↩get-proc n'f
    apply(unfold valid-return-list-def)
    apply(drule hd-Cons-tl[THEN sym])
    apply(erule-tac x=[] in allE)
    apply(erule-tac x=hd cs in allE)
    by auto
  from ‹n −[]→* n'› have n = n' by fastforce
  with ‹valid-edge (hd cs)› ‹kind (hd cs) = Q↩get-proc n'f› show ?case by blast
next
  case (vpra-intra cs a as)
  note IH = ‹⋀n'. ⟦n −as→* n'; valid-return-list cs n'; cs ≠ [];
    ∀ xs ys. as = xs@ys ⟶ ¬ same-level-path-rev-aux cs ys ∨ upd-rev-cs cs ys ≠
[]⟧
    ⟹ ∃ a Q f. valid-edge a ∧ kind a = Q↩get-proc nf›
  note all = ‹∀ xs ys. as@[a] = xs@ys
    ⟶ ¬ same-level-path-rev-aux cs ys ∨ upd-rev-cs cs ys ≠ []›
  from ‹n −as@[a]→* n'› have n −as→* sourcenode a and valid-edge a
    and targetnode a = n' by(auto intro:path-split-snoc)
```

from ‹*valid-return-list cs n'*› ‹*cs* ≠ []› **obtain** *Q f* **where** *valid-edge* (*hd cs*)
and *kind* (*hd cs*) = $Q\!\hookleftarrow_{\textit{get-proc } n'}\!f$
apply(*unfold valid-return-list-def*)
apply(*drule hd-Cons-tl*[*THEN sym*])
apply(*erule-tac x*=[] **in** *allE*)
apply(*erule-tac x*=*hd cs* **in** *allE*)
by *auto*
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
with ‹*kind* (*hd cs*) = $Q\!\hookleftarrow_{\textit{get-proc } n'}\!f$› ‹*targetnode a* = *n'*›
have *kind* (*hd cs*) = $Q\!\hookleftarrow_{\textit{get-proc (sourcenode } a)}\!f$ **by** *simp*
from ‹*valid-return-list cs n'*› ‹*targetnode a* = *n'*›
‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›
have *valid-return-list cs* (*sourcenode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*)
apply(*erule-tac x*=*c* **in** *allE*)
by(*auto split*:*list.split*)
from *all* ‹*intra-kind* (*kind a*)›
have ∀ *xs ys*. *as* = *xs*@*ys*
⟶ ¬ *same-level-path-rev-aux cs ys* ∨ *upd-rev-cs cs ys* ≠ []
apply *clarsimp* **apply**(*erule-tac x*=*xs* **in** *allE*)
by(*auto simp*:*intra-kind-def*)
from *IH*[*OF* ‹*n* −*as*→∗ *sourcenode a*› ‹*valid-return-list cs* (*sourcenode a*)›
‹*cs* ≠ []› *this*] **show** *?case* **.**
next
case (*vpra-Return cs a as Q p f*)
note *IH* = ‹⋀*n'*. ⟦*n* −*as*→∗ *n'*; *valid-return-list* (*a*#*cs*) *n'*; *a*#*cs* ≠ [];
∀ *xs ys*. *as* = *xs* @ *ys* ⟶
¬ *same-level-path-rev-aux* (*a*#*cs*) *ys* ∨ *upd-rev-cs* (*a*#*cs*) *ys* ≠ []⟧
⟹ ∃ *a Q f*. *valid-edge a* ∧ *kind a* = $Q\!\hookleftarrow_{\textit{get-proc } n}\!f$›
from ‹*n* −*as*@[*a*]→∗ *n'*› **have** *n* −*as*→∗ *sourcenode a* **and** *valid-edge a*
and *targetnode a* = *n'* **by**(*auto intro*:*path-split-snoc*)
from ‹*valid-edge a*› ‹*kind a* = $Q\!\hookleftarrow_{p}\!f$› **have** *get-proc* (*sourcenode a*) = *p*
by(*rule get-proc-return*)
with ‹*kind a* = $Q\!\hookleftarrow_{p}\!f$› ‹*valid-return-list cs n'*› ‹*valid-edge a*› ‹*targetnode a* = *n'*›
have *valid-return-list* (*a*#*cs*) (*sourcenode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x*=*list* **in** *allE*)
apply(*erule-tac x*=*c* **in** *allE*)
by(*auto split*:*list.split simp*:*targetnodes-def*)
from ‹∀ *xs ys*. *as*@[*a*] = *xs*@*ys* ⟶
¬ *same-level-path-rev-aux cs ys* ∨ *upd-rev-cs cs ys* ≠ []› ‹*kind a* = $Q\!\hookleftarrow_{p}\!f$›
have ∀ *xs ys*. *as* = *xs*@*ys* ⟶
¬ *same-level-path-rev-aux* (*a*#*cs*) *ys* ∨ *upd-rev-cs* (*a*#*cs*) *ys* ≠ []
apply *clarsimp* **apply**(*erule-tac x*=*xs* **in** *allE*)
by *auto*
from *IH*[*OF* ‹*n* −*as*→∗ *sourcenode a*› ‹*valid-return-list* (*a*#*cs*) (*sourcenode a*)›

```
    - this] show ?case by simp
next
  case (vpra-CallEmpty cs a as Q p fs)
  from ‹cs = []› ‹cs ≠ []› have False by simp
  thus ?case by simp
next
  case (vpra-CallCons cs a as Q r p fs c' cs')
  note IH = ‹⋀n'. ⟦n -as→* n'; valid-return-list cs' n'; cs' ≠ [];
    ∀xs ys. as = xs@ys ⟶
       ¬ same-level-path-rev-aux cs' ys ∨ upd-rev-cs cs' ys ≠ []⟧
    ⟹ ∃a Q f. valid-edge a ∧ kind a = Q↩get-proc n f›
  note all = ‹∀xs ys. as@[a] = xs@ys ⟶
     ¬ same-level-path-rev-aux cs ys ∨ upd-rev-cs cs ys ≠ []›
  from ‹n -as@[a]→* n'› have n -as→* sourcenode a and valid-edge a
    and targetnode a = n' by(auto intro:path-split-snoc)
  from ‹valid-return-list cs n'› ‹cs = c'#cs'› have valid-edge c'
    apply(clarsimp simp:valid-return-list-def)
    apply(erule-tac x=[] in allE)
    by auto
  show ?case
  proof(cases cs' = [])
    case True
    with ‹cs = c'#cs'› ‹kind a = Q:r↪pfs› ‹c' ∈ get-return-edges a›
    have same-level-path-rev-aux cs ([]@[a])
      and upd-rev-cs cs ([]@[a]) = []
      by(simp only:same-level-path-rev-aux.simps upd-rev-cs.simps,clarsimp)+
    with all have False by(erule-tac x=as in allE) fastforce
    thus ?thesis by simp
  next
    case False
    with ‹valid-return-list cs n'› ‹cs = c'#cs'›
    have valid-return-list cs' (targetnode c')
      apply(clarsimp simp:valid-return-list-def)
      apply(erule-tac x=c'#cs' in allE)
      apply(auto simp:targetnodes-def)
       apply(case-tac cs') apply auto
      apply(case-tac list) apply(auto simp:targetnodes-def)
      done
    from ‹valid-edge a› ‹c' ∈ get-return-edges a›
    have get-proc (sourcenode a) = get-proc (targetnode c')
      by(rule get-proc-get-return-edge)
    with ‹valid-return-list cs' (targetnode c')›
    have valid-return-list cs' (sourcenode a)
      apply(clarsimp simp:valid-return-list-def)
    apply(erule-tac x=cs' in allE)
    apply(erule-tac x=c in allE)
    by(auto split:list.split)
    from all ‹kind a = Q:r↪pfs› ‹cs = c'#cs'› ‹c' ∈ get-return-edges a›
    have ∀xs ys. as = xs@ys
```

$\longrightarrow \neg$ *same-level-path-rev-aux cs' ys* $\vee$ *upd-rev-cs cs' ys* $\neq []$
apply *clarsimp* **apply**(*erule-tac x=xs* **in** *allE*)
by *auto*
from *IH*[*OF* ‹*n* $-as\rightarrow*$ *sourcenode a*› ‹*valid-return-list cs'* (*sourcenode a*)›
False this] **show** *?thesis* **.**
qed
qed

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 $-$
from ‹*as* $\neq []$› **obtain** *a' as'* **where** *as* $= as'@[a']$ **by**(*cases as rule*:*rev-cases*)
auto
thus *?thesis*
proof(*cases kind a' rule*:*edge-kind-cases*)
case *Intra* **with** ‹*as* $= as'@[a']$› **show** *?thesis* **by** *simp*
next
case *Call* **with** ‹*as* $= as'@[a']$› **show** *?thesis* **by** *simp*
next
case (*Return Q p f*)
from ‹(*-Entry-*) $-as\rightarrow_{\surd}*$ *n*› **have** (*-Entry-*) $-as\rightarrow*$ *n* **and** *valid-path-rev-aux*
$[]$ *as*
by(*auto intro*:*vp-to-vpra simp*:*vp-def valid-path-def*)
from ‹(*-Entry-*) $-as\rightarrow*$ *n*› ‹*as* $= as'@[a']$›
have (*-Entry-*) $-as'\rightarrow*$ *sourcenode a'* **and** *valid-edge a'*
and *targetnode a'* $=$ *n*
by(*auto intro*:*path-split-snoc*)
from ‹*valid-path-rev-aux* $[]$ *as*› ‹*as* $= as'@[a']$› *Return*
have *valid-path-rev-aux* $[a']$ *as'* **by** *simp*
from ‹*valid-edge a'*› *Return*
have *valid-return-list* $[a']$ (*sourcenode a'*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*case-tac cs'*)
by(*auto intro*:*get-proc-return*[*THEN sym*])
show *?thesis*
proof(*cases* $\forall$ *xs ys. as'* $=$ *xs*@*ys* $\longrightarrow$
($\neg$ *same-level-path-rev-aux* $[a']$ *ys* $\vee$ *upd-rev-cs* $[a']$ *ys* $\neq []$))
case *True*
with ‹*valid-path-rev-aux* $[a']$ *as'*› ‹(*-Entry-*) $-as'\rightarrow*$ *sourcenode a'*›
‹*valid-return-list* $[a']$ (*sourcenode a'*)›
obtain *ax Qx fx* **where** *valid-edge ax* **and** *kind ax* $= Qx\hookleftarrow_{get\text{-}proc\ (-Entry-)} fx$
by(*fastforce dest*!:*vpra-no-slpra*)
hence *False* **by**(*fastforce intro*:*Main-no-return-source simp*:*get-proc-Entry*)
thus *?thesis* **by** *simp*
next

case *False*
then obtain *xs ys* **where** $as' = xs@ys$ **and** *same-level-path-rev-aux* $[a']$ *ys*
and *upd-rev-cs* $[a']$ *ys* $= []$ **by** *auto*
with *Return* **have** *same-level-path-rev-aux* $[] (ys@[a'])$
and *upd-rev-cs* $[] (ys@[a']) = []$ **by** *simp-all*
from ‹*upd-rev-cs* $[a']$ *ys* $= []$› **have** $ys \neq []$ **by** *auto*
with ‹(*-Entry-*) $-as'\rightarrow* sourcenode\ a'$› ‹$as' = xs@ys$›
have $hd(sourcenodes\ ys) -ys\rightarrow* sourcenode\ a'$
by(*cases ys*)(*auto dest*:*path-split-second simp*:*sourcenodes-def*)
with ‹*targetnode* $a' = n$› ‹*valid-edge* a'›
have $hd(sourcenodes\ ys) -ys@[a']\rightarrow* n$
by(*fastforce intro*:*path-Append path-edge*)
with ‹*same-level-path-rev-aux* $[] (ys@[a'])$› ‹*upd-rev-cs* $[] (ys@[a']) = []$›
have *same-level-path* $(ys@[a'])$
by(*fastforce dest*:*slpra-to-slpa simp*:*same-level-path-def valid-return-list-def*)
with ‹$hd(sourcenodes\ ys) -ys@[a']\rightarrow* n$› **have** $hd(sourcenodes\ ys) -ys@[a']\rightarrow_{sl}* n$
by(*simp add*:*slp-def*)
with ‹$as = as'@[a']$› ‹$as' = xs@ys$› *Return*
have $\exists as'\ as''\ n'.\ as = as'@as'' \wedge as'' \neq [] \wedge n' -as''\rightarrow_{sl}* n$
by(*rule-tac x*=*xs* **in** *exI*) *auto*
thus *?thesis* **by** *simp*
qed
qed
qed

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 $\forall a' \in set\ as'.\ intra\text{-}kind(kind\ a') \vee (\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs)$
proof(*atomize-elim*)
from ‹(*-Entry-*) $-as\rightarrow_{\surd}* n$›
show $\exists as'.$ (*-Entry-*) $-as'\rightarrow_{\surd}* n \wedge set(sourcenodes\ as') \subseteq set(sourcenodes\ as) \wedge$
$(\forall a' \in set\ as'.\ intra\text{-}kind(kind\ a') \vee (\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs))$
proof(*induct as arbitrary*:*n rule*:*length-induct*)
fix *as n*
assume *IH*:$\forall as''.\ length\ as'' < length\ as \longrightarrow$
$(\forall n'.$ (*-Entry-*) $-as''\rightarrow_{\surd}* n' \longrightarrow$
$(\exists as'.$ (*-Entry-*) $-as'\rightarrow_{\surd}* n' \wedge set\ (sourcenodes\ as') \subseteq set\ (sourcenodes\ as'') \wedge$
$(\forall a' \in set\ as'.\ intra\text{-}kind\ (kind\ a') \vee (\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs))))$
and (*-Entry-*) $-as\rightarrow_{\surd}* n$
show $\exists as'.$ (*-Entry-*) $-as'\rightarrow_{\surd}* n \wedge set(sourcenodes\ as') \subseteq set(sourcenodes\ as) \wedge$
$(\forall a' \in set\ as'.\ intra\text{-}kind(kind\ a') \vee (\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs))$
proof(*cases as* $= []$)
case *True*

with ‹(-*Entry*-) $-as\rightarrow_{\surd}* n$› **show** *?thesis* **by**(*fastforce simp*:*sourcenodes-def vp-def*)

next

case *False*

with ‹(-*Entry*-) $-as\rightarrow_{\surd}* n$›

have $((\exists a'\ as'.\ as = as'@[a'] \wedge$ *intra-kind*$(kind\ a')) \vee$
$(\exists a'\ as'\ Q\ r\ p\ fs.\ as = as'@[a'] \wedge kind\ a' = Q{:}r\hookrightarrow_p fs)) \vee$
$(\exists as'\ as''\ n'.\ as = as'@as'' \wedge as'' \neq [] \wedge n' -as''\rightarrow_{sl}* n)$
by(*fastforce dest*!:*valid-Entry-path-cases*)

thus *?thesis* **apply** −

proof(*erule disjE*)+

assume $\exists a'\ as'.\ as = as'@[a'] \wedge$ *intra-kind*$(kind\ a')$

then obtain $a'\ as'$ **where** $as = as'@[a']$ **and** *intra-kind*$(kind\ a')$ **by** *blast*

from ‹(-*Entry*-) $-as\rightarrow_{\surd}* n$› ‹$as = as'@[a']$›

have (-*Entry*-) $-as'\rightarrow_{\surd}*$ *sourcenode* a' **and** *valid-edge* a'
and *targetnode* $a' = n$
by(*auto intro*:*vp-split-snoc*)

from ‹*valid-edge* a'› ‹*intra-kind*$(kind\ a')$›

have *sourcenode* $a' -[a']\rightarrow_{sl}*$ *targetnode* a'
by(*fastforce intro*:*path-edge intras-same-level-path simp*:*slp-def*)

from *IH* ‹(-*Entry*-) $-as'\rightarrow_{\surd}*$ *sourcenode* a'› ‹$as = as'@[a']$›

obtain *xs* **where** (-*Entry*-) $-xs\rightarrow_{\surd}*$ *sourcenode* a'
and *set* (*sourcenodes xs*) $\subseteq$ *set* (*sourcenodes as'*)
and $\forall a'{\in}set\ xs.$ *intra-kind* (*kind* a') $\vee$ $(\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs)$
apply(*erule-tac* $x{=}as'$ **in** *allE*) **by** *auto*

from ‹(-*Entry*-) $-xs\rightarrow_{\surd}*$ *sourcenode* a'› ‹*sourcenode* $a' -[a']\rightarrow_{sl}*$ *targetnode* a'›

have (-*Entry*-) $-xs@[a']\rightarrow_{\surd}*$ *targetnode* a' **by**(*rule vp-slp-Append*)

with ‹*targetnode* $a' = n$› **have** (-*Entry*-) $-xs@[a']\rightarrow_{\surd}* n$ **by** *simp*

moreover

from ‹*set* (*sourcenodes xs*) $\subseteq$ *set* (*sourcenodes as'*)› ‹$as = as'@[a']$›

have *set* (*sourcenodes* ($xs@[a']$)) $\subseteq$ *set* (*sourcenodes as*)
by(*auto simp*:*sourcenodes-def*)

moreover

from ‹$\forall a'{\in}set\ xs.$ *intra-kind* (*kind* a') $\vee$ $(\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs)$›
‹*intra-kind*$(kind\ a')$›

have $\forall a'{\in}set\ (xs@[a']).$ *intra-kind* (*kind* a') $\vee$
$(\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs)$
by *fastforce*

ultimately show *?thesis* **by** *blast*

next

assume $\exists a'\ as'\ Q\ r\ p\ fs.\ as = as'@[a'] \wedge kind\ a' = Q{:}r\hookrightarrow_p fs$

then obtain $a'\ as'\ Q\ r\ p\ fs$ **where** $as = as'@[a']$ **and** $kind\ a' = Q{:}r\hookrightarrow_p fs$
by *blast*

from ‹(-*Entry*-) $-as\rightarrow_{\surd}* n$› ‹$as = as'@[a']$›

have (-*Entry*-) $-as'\rightarrow_{\surd}*$ *sourcenode* a' **and** *valid-edge* a'
and *targetnode* $a' = n$
by(*auto intro*:*vp-split-snoc*)

from *IH* ‹(-*Entry*-) $-as'\rightarrow_{\surd}*$ *sourcenode* a'› ‹$as = as'@[a']$›

obtain xs **where** (*-Entry-*) $-xs\rightarrow_{\sqrt{}}* sourcenode\ a'$
and $set\ (sourcenodes\ xs) \subseteq set\ (sourcenodes\ as')$
and $\forall a' \in set\ xs.\ intra\text{-}kind\ (kind\ a') \vee (\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs)$
apply(*erule-tac* $x{=}as'$ **in** *allE*) **by** *auto*
from ‹$targetnode\ a' = n$› ‹$valid\text{-}edge\ a'$› ‹$kind\ a' = Q{:}r\hookrightarrow_p fs$›
‹(*-Entry-*) $-xs\rightarrow_{\sqrt{}}* sourcenode\ a'$›
have (*-Entry-*) $-xs@[a']\rightarrow_{\sqrt{}}* n$
by(*fastforce intro*:*path-Append path-edge vpa-snoc-Call*
simp:*vp-def valid-path-def*)
moreover
from ‹$set\ (sourcenodes\ xs) \subseteq set\ (sourcenodes\ as')$› ‹$as = as'@[a']$›
have $set\ (sourcenodes\ (xs@[a'])) \subseteq set\ (sourcenodes\ as)$
by(*auto simp*:*sourcenodes-def*)
moreover
from ‹$\forall a' \in set\ xs.\ intra\text{-}kind\ (kind\ a') \vee (\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs)$›
‹$kind\ a' = Q{:}r\hookrightarrow_p fs$›
have $\forall a' \in set\ (xs@[a']).\ intra\text{-}kind\ (kind\ a') \vee$
$(\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs)$
by *fastforce*
ultimately show *?thesis* **by** *blast*
next
assume $\exists as'\ as''\ n'.\ as = as'@as'' \wedge as'' \neq [] \wedge n' -as''\rightarrow_{sl}* n$
then obtain $as'\ as''\ n'$ **where** $as = as'@as''$ **and** $as'' \neq []$
and $n' -as''\rightarrow_{sl}* n$ **by** *blast*
from ‹(*-Entry-*) $-as\rightarrow_{\sqrt{}}* n$› ‹$as = as'@as''$› ‹$as'' \neq []$›
have (*-Entry-*) $-as'\rightarrow_{\sqrt{}}* hd(sourcenodes\ as'')$
by(*cases* as'',*auto intro*:*vp-split simp*:*sourcenodes-def*)
from ‹$n' -as''\rightarrow_{sl}* n$› ‹$as'' \neq []$› **have** $hd(sourcenodes\ as'') = n'$
by(*fastforce intro*:*path-sourcenode simp*:*slp-def*)
from ‹$as = as'@as''$› ‹$as'' \neq []$› **have** $length\ as' < length\ as$ **by** *simp*
with *IH* ‹(*-Entry-*) $-as'\rightarrow_{\sqrt{}}* hd(sourcenodes\ as'')$›
‹$hd(sourcenodes\ as'') = n'$›
obtain xs **where** (*-Entry-*) $-xs\rightarrow_{\sqrt{}}* n'$
and $set\ (sourcenodes\ xs) \subseteq set\ (sourcenodes\ as')$
and $\forall a' \in set\ xs.\ intra\text{-}kind\ (kind\ a') \vee (\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs)$
apply(*erule-tac* $x{=}as'$ **in** *allE*) **by** *auto*
from ‹$n' -as''\rightarrow_{sl}* n$› **obtain** ys **where** $n' -ys\rightarrow_{\iota}* n$
and $set(sourcenodes\ ys) \subseteq set(sourcenodes\ as'')$
by(*erule same-level-path-inner-path*)
from ‹(*-Entry-*) $-xs\rightarrow_{\sqrt{}}* n'$› ‹$n' -ys\rightarrow_{\iota}* n$› **have** (*-Entry-*) $-xs@ys\rightarrow_{\sqrt{}}* n$
by(*fastforce intro*:*vp-slp-Append intra-path-slp*)
moreover
from ‹$set\ (sourcenodes\ xs) \subseteq set\ (sourcenodes\ as')$›
‹$set(sourcenodes\ ys) \subseteq set(sourcenodes\ as'')$› ‹$as = as'@as''$›
have $set\ (sourcenodes\ (xs@ys)) \subseteq set(sourcenodes\ as)$
by(*auto simp*:*sourcenodes-def*)
moreover
from ‹$\forall a' \in set\ xs.\ intra\text{-}kind\ (kind\ a') \vee (\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r\hookrightarrow_p fs)$›
‹$n' -ys\rightarrow_{\iota}* n$›

have $\forall a' \in set\ (xs@ys).\ intra\text{-}kind\ (kind\ a') \vee (\exists Q\ r\ p\ fs.\ kind\ a' = Q{:}r{\hookrightarrow_p}fs)$
by(*fastforce simp*:*intra-path-def*)
ultimately show *?thesis* **by** *blast*
qed
qed
qed
qed

end

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

1.2.3 Adds an exit node to the abstract CFG

locale *CFGExit* = *CFG sourcenode targetnode kind valid-edge Entry*
get-proc get-return-edges procs Main
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ (*'var*,*'val*,*'ret*,*'pname*) *edge-kind*
and *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'*(*'-Entry'-*)) **and** *get-proc* :: *'node* ⇒ *'pname*
and *get-return-edges* :: *'edge* ⇒ *'edge set*
and *procs* :: (*'pname* × *'var list* × *'var list*) *list* **and** *Main* :: *'pname* +
fixes *Exit*::*'node* (*'*(*'-Exit'-*))
assumes *Exit-source* [*dest*]: ⟦*valid-edge a*; *sourcenode a* = (*-Exit-*)⟧ ⟹ *False*
and *get-proc-Exit*:*get-proc* (*-Exit-*) = *Main*
and *Exit-no-return-target*:
⟦*valid-edge a*; *kind a* = $Q{\hookleftarrow_p}pf$; *targetnode a* = (*-Exit-*)⟧ ⟹ *False*
and *Entry-Exit-edge*: ∃ *a*. *valid-edge a* ∧ *sourcenode a* = (*-Entry-*) ∧
targetnode a = (*-Exit-*) ∧ *kind a* = $(\lambda s.\ False)_{\surd}$

begin

lemma *Entry-noteq-Exit* [*dest*]:
assumes *eq*:(*-Entry-*) = (*-Exit-*) **shows** *False*
proof −
from *Entry-Exit-edge* **obtain** *a* **where** *sourcenode a* = (*-Entry-*)
and *valid-edge a* **by** *blast*
with *eq* **show** *False* **by** *simp*(*erule Exit-source*)
qed

lemma *Exit-noteq-Entry* [*dest*]:(*-Exit-*) = (*-Entry-*) ⟹ *False*
by(*rule Entry-noteq-Exit*[*OF sym*],*simp*)

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

proof −
from *Entry-Exit-edge* **obtain** *a* **where** *sourcenode a* = (*-Entry-*)
and *valid-edge a* **by** *blast*
thus *?thesis* **by**(*fastforce simp*:*valid-node-def*)
qed

lemma [*simp*]: *valid-node* (*-Exit-*)
proof −
from *Entry-Exit-edge* **obtain** *a* **where** *targetnode a* = (*-Exit-*)
and *valid-edge a* **by** *blast*
thus *?thesis* **by**(*fastforce simp*:*valid-node-def*)
qed

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*
by(*fastforce simp*:*method-exit-def*)

lemma *method-exit-inner-path*:
assumes *method-exit n* **and** *n* −*as*→ι∗ *n'* **shows** *as* = []
using ‹*method-exit n*›
proof(*rule method-exit-cases*)
assume *n* = (*-Exit-*)
show *?thesis*
proof(*cases as*)
case (*Cons a' as'*)
with ‹*n* −*as*→ι∗ *n'*› **have** *n* = *sourcenode a'* **and** *valid-edge a'*
by(*auto elim*:*path-split-Cons simp*:*intra-path-def*)
with ‹*n* = (*-Exit-*)› **have** *sourcenode a'* = (*-Exit-*) **by** *simp*
with ‹*valid-edge a'*› **have** *False* **by**(*rule Exit-source*)
thus *?thesis* **by** *simp*
qed *simp*
next
fix *a Q f p*
assume *n* = *sourcenode a* **and** *valid-edge a* **and** *kind a* = *Q*↩*pf*
show *?thesis*
proof(*cases as*)
case (*Cons a' as'*)
with ‹*n* −*as*→ι∗ *n'*› **have** *n* = *sourcenode a'* **and** *valid-edge a'*
and *intra-kind* (*kind a'*)
by(*auto elim*:*path-split-Cons simp*:*intra-path-def*)

from ‹*valid-edge a*› ‹*kind a* = *Q*↪*pf*› ‹*valid-edge a'*› ‹*n* = *sourcenode a*›
‹*n* = *sourcenode a'*› ‹*intra-kind* (*kind a'*)›
have *False* **by**(*fastforce dest*:*return-edges-only simp*:*intra-kind-def*)
thus *?thesis* **by** *simp*
qed *simp*
qed

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*
by(*simp add*:*inner-node-def valid-node-def*)

lemma [*dest*]:
inner-node (*-Entry-*) ⟹ *False*
by(*simp add*:*inner-node-def*)

lemma [*dest*]:
inner-node (*-Exit-*) ⟹ *False*
by(*simp add*:*inner-node-def*)

lemma [*simp*]:⟦*valid-edge a*; *targetnode a* ≠ (*-Exit-*)⟧
⟹ *inner-node* (*targetnode a*)
by(*simp add*:*inner-node-def*,*rule ccontr*,*simp*,*erule Entry-target*)

lemma [*simp*]:⟦*valid-edge a*; *sourcenode a* ≠ (*-Entry-*)⟧
⟹ *inner-node* (*sourcenode a*)
by(*simp add*:*inner-node-def*,*rule ccontr*,*simp*,*erule Exit-source*)

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*
apply(*auto simp*:*valid-node-def*)
apply(*case-tac sourcenode a* = (*-Entry-*)) **apply** *auto*
apply(*case-tac targetnode a* = (*-Exit-*)) **apply** *auto*
done

Lemmas on paths with (*-Exit-*)

lemma *path-Exit-source*:
⟦*n* −*as*→∗ *n'*; *n* = (*-Exit-*)⟧ ⟹ *n'* = (*-Exit-*) ∧ *as* = []
proof(*induct rule*:*path.induct*)
case (*Cons-path n''as n' a n*)
from ‹*n* = (*-Exit-*)› ‹*sourcenode a* = *n*› ‹*valid-edge a*› **have** *False*
by −(*rule Exit-source*,*simp-all*)

thus *?case* **by** *simp*
qed *simp*

lemma [*dest*]:(*-Exit-*) $-as\rightarrow* n' \Longrightarrow n' = (\text{-}Exit\text{-}) \wedge as = []$
by(*fastforce elim*!:*path-Exit-source*)

lemma *Exit-no-sourcenode*[*dest*]:
assumes *isin*:(*-Exit-*) ∈ *set* (*sourcenodes as*) **and** *path*:$n -as\rightarrow* n'$
shows *False*
proof −
from *isin* **obtain** *ns' ns''* **where** *sourcenodes as* = *ns'*@(*-Exit-*)#*ns''*
by(*auto dest*:*split-list simp*:*sourcenodes-def*)
then obtain *as' as'' a* **where** *as* = *as'*@*a*#*as''*
and *source*:*sourcenode a* = (*-Exit-*)
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
with *path* **have** *valid-edge a* **by**(*fastforce dest*:*path-split*)
with *source* **show** *?thesis* **by** −(*erule Exit-source*)
qed

lemma *vpa-no-slpa*:
⟦*valid-path-aux cs as*; $n -as\rightarrow* n'$; *valid-call-list cs n*; *cs* ≠ []; ∀ *xs ys*. *as* = *xs*@*ys* ⟶ (¬ *same-level-path-aux cs xs* ∨ *upd-cs cs xs* ≠ [])⟧
⟹ ∃ *a Q r fs*. *valid-edge a* ∧ *kind a* = $Q:r\hookrightarrow_{get\text{-}proc\ n'}fs$
proof(*induct arbitrary*:*n rule*:*vpa-induct*)
case (*vpa-empty cs*)
from ‹*valid-call-list cs n*› ‹*cs* ≠ []› **obtain** *Q r fs* **where** *valid-edge* (*hd cs*)
and *kind* (*hd cs*) = $Q:r\hookrightarrow_{get\text{-}proc\ n}fs$
apply(*unfold valid-call-list-def*)
apply(*drule hd-Cons-tl*[*THEN sym*])
apply(*erule-tac x*=[] **in** *allE*)
apply(*erule-tac x*=*hd cs* **in** *allE*)
by *auto*
from ‹$n -[]\rightarrow* n'$› **have** *n* = *n'* **by** *fastforce*
with ‹*valid-edge* (*hd cs*)› ‹*kind* (*hd cs*) = $Q:r\hookrightarrow_{get\text{-}proc\ n}fs$› **show** *?case* **by** *blast*
next
case (*vpa-intra cs a as*)
note *IH* = ‹⋀*n*. ⟦$n -as\rightarrow* n'$; *valid-call-list cs n*; *cs* ≠ []; ∀ *xs ys*. *as* = *xs*@*ys* ⟶ ¬ *same-level-path-aux cs xs* ∨ *upd-cs cs xs* ≠ []⟧
⟹ ∃ *a' Q' r' fs'*. *valid-edge a'* ∧ *kind a'* = $Q':r'\hookrightarrow_{get\text{-}proc\ n'}fs'$›
note *all* = ‹∀ *xs ys*. *a*#*as* = *xs*@*ys*
⟶ ¬ *same-level-path-aux cs xs* ∨ *upd-cs cs xs* ≠ []›
from ‹$n -a\#as\rightarrow* n'$› **have** *sourcenode a* = *n* **and** *valid-edge a*
and *targetnode a* $-as\rightarrow* n'$
by(*auto intro*:*path-split-Cons*)
from ‹*valid-call-list cs n*› ‹*cs* ≠ []› **obtain** *Q r fs* **where** *valid-edge* (*hd cs*)
and *kind* (*hd cs*) = $Q:r\hookrightarrow_{get\text{-}proc\ n}fs$
apply(*unfold valid-call-list-def*)

apply(*drule hd-Cons-tl*[*THEN sym*])
apply(*erule-tac x*=[] **in** *allE*)
apply(*erule-tac x*=*hd cs* **in** *allE*)
by *auto*
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
with ‹*kind* (*hd cs*) = *Q*:*r*↪$_{\textit{get-proc } n}$*fs*› ‹*sourcenode a* = *n*›
have *kind* (*hd cs*) = *Q*:*r*↪$_{\textit{get-proc}\ (\textit{targetnode}\ a)}$*fs* **by** *simp*
from ‹*valid-call-list cs n*› ‹*sourcenode a* = *n*›
‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›
have *valid-call-list cs* (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*)
apply(*erule-tac x*=*c* **in** *allE*)
by(*auto split*:*list.split*)
from *all* ‹*intra-kind* (*kind a*)›
have ∀ *xs ys*. *as* = *xs*@*ys* ⟶ ¬ *same-level-path-aux cs xs* ∨ *upd-cs cs xs* ≠ []
apply *clarsimp* **apply**(*erule-tac x*=*a*#*xs* **in** *allE*)
by(*auto simp*:*intra-kind-def*)
from *IH*[*OF* ‹*targetnode a* −*as*→∗ *n'*› ‹*valid-call-list cs* (*targetnode a*)›
‹*cs* ≠ []› *this*] **show** *?case* **.**
next
case (*vpa-Call cs a as Q r p fs*)
note *IH* = ‹⋀*n*. ⟦*n* −*as*→∗ *n'*; *valid-call-list* (*a*#*cs*) *n*; *a*#*cs* ≠ []; ∀ *xs ys*. *as* = *xs*@*ys* ⟶ ¬ *same-level-path-aux* (*a*#*cs*) *xs* ∨ *upd-cs* (*a*#*cs*) *xs* ≠ []⟧
⟹ ∃ *a' Q' r' fs'*. *valid-edge a'* ∧ *kind a'* = *Q'*:*r'*↪$_{\textit{get-proc } n'}$*fs'*›
note *all* = ‹∀ *xs ys*.
a#*as* = *xs*@*ys* ⟶ ¬ *same-level-path-aux cs xs* ∨ *upd-cs cs xs* ≠ []›
from ‹*n* −*a*#*as*→∗ *n'*› **have** *sourcenode a* = *n* **and** *valid-edge a*
and *targetnode a* −*as*→∗ *n'*
by(*auto intro*:*path-split-Cons*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_{p}$*fs*› **have** *get-proc* (*targetnode a*) = *p*
by(*rule get-proc-call*)
with ‹*kind a* = *Q*:*r*↪$_{p}$*fs*› **have** *kind a* = *Q*:*r*↪$_{\textit{get-proc}\ (\textit{targetnode}\ a)}$*fs* **by** *simp*
with ‹*valid-call-list cs n*› ‹*valid-edge a*› ‹*sourcenode a* = *n*›
have *valid-call-list* (*a*#*cs*) (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x*=*list* **in** *allE*)
apply(*erule-tac x*=*c* **in** *allE*)
by(*auto split*:*list.split simp*:*sourcenodes-def*)
from *all* ‹*kind a* = *Q*:*r*↪$_{p}$*fs*›
have ∀ *xs ys*. *as* = *xs*@*ys*
⟶ ¬ *same-level-path-aux* (*a*#*cs*) *xs* ∨ *upd-cs* (*a*#*cs*) *xs* ≠ []
apply *clarsimp* **apply**(*erule-tac x*=*a*#*xs* **in** *allE*)
by *auto*
from *IH*[*OF* ‹*targetnode a* −*as*→∗ *n'*› ‹*valid-call-list* (*a*#*cs*) (*targetnode a*)›
- *this*] **show** *?case* **by** *simp*

next
case (*vpa-ReturnEmpty cs a as Q p fx*)
from ‹$cs \neq []$› ‹$cs = []$› **have** *False* **by** *simp*
thus *?case* **by** *simp*
next
case (*vpa-ReturnCons cs a as Q p f c' cs'*)
note *IH* = ‹$\bigwedge n.\ [\![n\ -as\rightarrow*\ n';\ valid\text{-}call\text{-}list\ cs'\ n;\ cs' \neq [];$
$\forall xs\ ys.\ as = xs@ys \longrightarrow \neg\ same\text{-}level\text{-}path\text{-}aux\ cs'\ xs \vee upd\text{-}cs\ cs'\ xs \neq []]\!]$
$\Longrightarrow \exists a'\ Q'\ r'\ fs'.\ valid\text{-}edge\ a' \wedge kind\ a' = Q':r'\hookrightarrow_{get\text{-}proc\ n'}fs'$›
note *all* = ‹$\forall xs\ ys.\ a\#as = xs@ys$
$\longrightarrow \neg\ same\text{-}level\text{-}path\text{-}aux\ cs\ xs \vee upd\text{-}cs\ cs\ xs \neq []$›
from ‹$n\ -a\#as\rightarrow*\ n'$› **have** *sourcenode a* = *n* **and** *valid-edge a*
and *targetnode a* $-as\rightarrow*$ *n'*
by(*auto intro*:*path-split-Cons*)
from ‹*valid-call-list cs n*› ‹$cs = c'\#cs'$› **have** *valid-edge c'*
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=[] **in** *allE*)
by *auto*
show *?case*
proof(*cases cs'* = [])
case *True*
with *all* ‹$cs = c'\#cs'$› ‹*kind a* = $Q\hookleftarrow_p f$› ‹$a \in$ *get-return-edges c'*› **have** *False*
by(*erule-tac x*=[*a*] **in** *allE*,*fastforce*)
thus *?thesis* **by** *simp*
next
case *False*
with ‹*valid-call-list cs n*› ‹$cs = c'\#cs'$›
have *valid-call-list cs'* (*sourcenode c'*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*c'*#*cs'* **in** *allE*)
apply(*auto simp*:*sourcenodes-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*case-tac list*) **apply**(*auto simp*:*sourcenodes-def*)
done
from ‹*valid-edge c'*› ‹$a \in$ *get-return-edges c'*›
have *get-proc* (*sourcenode c'*) = *get-proc* (*targetnode a*)
by(*rule get-proc-get-return-edge*)
with ‹*valid-call-list cs'* (*sourcenode c'*)›
have *valid-call-list cs'* (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*)
apply(*erule-tac x*=*c* **in** *allE*)
by(*auto split*:*list.split*)
from *all* ‹*kind a* = $Q\hookleftarrow_p f$› ‹$cs = c'\#cs'$› ‹$a \in$ *get-return-edges c'*›
have $\forall xs\ ys.\ as = xs@ys \longrightarrow \neg\ same\text{-}level\text{-}path\text{-}aux\ cs'\ xs \vee upd\text{-}cs\ cs'\ xs \neq []$
apply *clarsimp* **apply**(*erule-tac x*=*a*#*xs* **in** *allE*)
by *auto*
from *IH*[*OF* ‹*targetnode a* $-as\rightarrow*$ *n'*› ‹*valid-call-list cs'* (*targetnode a*)›
False this] **show** *?thesis* **.**

```
  qed
qed


lemma valid-Exit-path-cases:
  assumes n −as→√* (-Exit-) and as ≠ []
  shows (∃ a' as'. as = a'#as' ∧ intra-kind(kind a')) ∨
        (∃ a' as' Q p f. as = a'#as' ∧ kind a' = Q↩pf) ∨
        (∃ as' as'' n'. as = as'@as'' ∧ as' ≠ [] ∧ n −as'→sl* n')
proof −
  from ‹as ≠ []› obtain a' as' where as = a'#as' by(cases as) auto
  thus ?thesis
  proof(cases kind a' rule:edge-kind-cases)
    case Intra with ‹as = a'#as'› show ?thesis by simp
  next
    case Return with ‹as = a'#as'› show ?thesis by simp
  next
    case (Call Q r p f)
    from ‹n −as→√* (-Exit-)› have n −as→* (-Exit-) and valid-path-aux [] as
      by(simp-all add:vp-def valid-path-def)
    from ‹n −as→* (-Exit-)› ‹as = a'#as'›
    have sourcenode a' = n and valid-edge a' and targetnode a' −as'→* (-Exit-)
      by(auto intro:path-split-Cons)
    from ‹valid-path-aux [] as› ‹as = a'#as'› Call
    have valid-path-aux [a'] as' by simp
    from ‹valid-edge a'› Call
    have valid-call-list [a'] (targetnode a')
      apply(clarsimp simp:valid-call-list-def)
      apply(case-tac cs')
      by(auto intro:get-proc-call[THEN sym])
    show ?thesis
    proof(cases ∀ xs ys. as' = xs@ys ⟶
        (¬ same-level-path-aux [a'] xs ∨ upd-cs [a'] xs ≠ []))
      case True
      with ‹valid-path-aux [a'] as'› ‹targetnode a' −as'→* (-Exit-)›
        ‹valid-call-list [a'] (targetnode a')›
      obtain ax Qx rx fsx where valid-edge ax and kind ax = Qx:rx↪get-proc (-Exit-)fsx
        by(fastforce dest!:vpa-no-slpa)
      hence False by(fastforce intro:Main-no-call-target simp:get-proc-Exit)
      thus ?thesis by simp
    next
      case False
      then obtain xs ys where as' = xs@ys and same-level-path-aux [a'] xs
        and upd-cs [a'] xs = [] by auto
      with Call have same-level-path (a'#xs) by(simp add:same-level-path-def)
      from ‹upd-cs [a'] xs = []› have xs ≠ [] by auto
      with ‹targetnode a' −as'→* (-Exit-)› ‹as' = xs@ys›
      have targetnode a' −xs→* last(targetnodes xs)
        apply(cases xs rule:rev-cases)
```

```
      by(auto intro:path-Append path-split path-edge simp:targetnodes-def)
    with ‹sourcenode a' = n› ‹valid-edge a'› ‹same-level-path (a'#xs)›
    have n -a'#xs→sl* last(targetnodes xs)
      by(fastforce intro:Cons-path simp:slp-def)
    with ‹as = a'#as'› ‹as' = xs@ys› Call
    have ∃ as' as'' n'. as = as'@as'' ∧ as' ≠ [] ∧ n -as'→sl* n'
      by(rule-tac x=a'#xs in exI) auto
    thus ?thesis by simp
  qed
 qed
qed


lemma valid-Exit-path-descending-path:
 assumes n -as→√* (-Exit-)
 obtains as' where n -as'→√* (-Exit-)
 and set(sourcenodes as') ⊆ set(sourcenodes as)
 and ∀ a' ∈ set as'. intra-kind(kind a') ∨ (∃ Q f p. kind a' = Q↩pf)
proof(atomize-elim)
 from ‹n -as→√* (-Exit-)›
 show ∃ as'. n -as'→√* (-Exit-) ∧ set(sourcenodes as') ⊆ set(sourcenodes as)∧
           (∀ a' ∈ set as'. intra-kind(kind a') ∨ (∃ Q f p. kind a' = Q↩pf))
 proof(induct as arbitrary:n rule:length-induct)
  fix as n
  assume IH:∀ as''. length as'' < length as ⟶
   (∀ n'. n' -as''→√* (-Exit-) ⟶
    (∃ as'. n' -as'→√* (-Exit-) ∧ set (sourcenodes as') ⊆ set (sourcenodes as'')
∧
           (∀ a'∈set as'. intra-kind (kind a') ∨ (∃ Q f p. kind a' = Q↩pf))))
    and n -as→√* (-Exit-)
  show ∃ as'. n -as'→√* (-Exit-) ∧ set(sourcenodes as') ⊆ set(sourcenodes as)∧
           (∀ a' ∈ set as'. intra-kind(kind a') ∨ (∃ Q f p. kind a' = Q↩pf))
  proof(cases as = [])
   case True
    with ‹n -as→√* (-Exit-)› show ?thesis by(fastforce simp:sourcenodes-def
vp-def)
  next
   case False
   with ‹n -as→√* (-Exit-)›
   have ((∃ a' as'. as = a'#as' ∧ intra-kind(kind a')) ∨
     (∃ a' as' Q p f. as = a'#as' ∧ kind a' = Q↩pf)) ∨
     (∃ as' as'' n'. as = as'@as'' ∧ as' ≠ [] ∧ n -as'→sl* n')
    by(auto dest!:valid-Exit-path-cases)
   thus ?thesis apply -
   proof(erule disjE)+
    assume ∃ a' as'. as = a'#as' ∧ intra-kind(kind a')
    then obtain a' as' where as = a'#as' and intra-kind(kind a') by blast
    from ‹n -as→√* (-Exit-)› ‹as = a'#as'›
    have sourcenode a' = n and valid-edge a'
```

and *targetnode* $a' -as'\rightarrow_{\surd}*$ (*-Exit-*)
by(*auto intro*:*vp-split-Cons*)
from ‹*valid-edge* a'› ‹*intra-kind*(*kind* a')›
have *sourcenode* $a' -[a']\rightarrow_{sl}*$ *targetnode* a'
by(*fastforce intro*:*path-edge intras-same-level-path simp*:*slp-def*)
from *IH* ‹*targetnode* $a' -as'\rightarrow_{\surd}*$ (*-Exit-*)› ‹$as = a'\#as'$›
obtain *xs* **where** *targetnode* $a' -xs\rightarrow_{\surd}*$ (*-Exit-*)
and *set* (*sourcenodes xs*) $\subseteq$ *set* (*sourcenodes as'*)
and $\forall a'\in$*set xs*. *intra-kind* (*kind* a') $\vee$ ($\exists$ *Q f p*. *kind* $a' = Q\hookleftarrow_p f$)
apply(*erule-tac x*=*as'* **in** *allE*) **by** *auto*
from ‹*sourcenode* $a' -[a']\rightarrow_{sl}*$ *targetnode* a'› ‹*targetnode* $a' -xs\rightarrow_{\surd}*$ (*-Exit-*)›
have *sourcenode* $a' -[a']@xs\rightarrow_{\surd}*$ (*-Exit-*) **by**(*rule slp-vp-Append*)
with ‹*sourcenode* $a' = n$› **have** $n -a'\#xs\rightarrow_{\surd}*$ (*-Exit-*) **by** *simp*
moreover
from ‹*set* (*sourcenodes xs*) $\subseteq$ *set* (*sourcenodes as'*)› ‹$as = a'\#as'$›
have *set* (*sourcenodes* ($a'\#xs$)) $\subseteq$ *set* (*sourcenodes as*)
by(*auto simp*:*sourcenodes-def*)
moreover
from ‹$\forall a'\in$*set xs*. *intra-kind* (*kind* a') $\vee$ ($\exists$ *Q f p*. *kind* $a' = Q\hookleftarrow_p f$)›
‹*intra-kind*(*kind* a')›
have $\forall a'\in$*set* ($a'\#xs$). *intra-kind* (*kind* a') $\vee$ ($\exists$ *Q f p*. *kind* $a' = Q\hookleftarrow_p f$)
by *fastforce*
ultimately show *?thesis* **by** *blast*
next
assume $\exists a'$ *as' Q p f*. $as = a'\#as' \wedge$ *kind* $a' = Q\hookleftarrow_p f$
then obtain a' *as' Q p f* **where** $as = a'\#as'$ **and** *kind* $a' = Q\hookleftarrow_p f$ **by** *blast*
from ‹$n -as\rightarrow_{\surd}*$ (*-Exit-*)› ‹$as = a'\#as'$›
have *sourcenode* $a' = n$ **and** *valid-edge* a'
and *targetnode* $a' -as'\rightarrow_{\surd}*$ (*-Exit-*)
by(*auto intro*:*vp-split-Cons*)
from *IH* ‹*targetnode* $a' -as'\rightarrow_{\surd}*$ (*-Exit-*)› ‹$as = a'\#as'$›
obtain *xs* **where** *targetnode* $a' -xs\rightarrow_{\surd}*$ (*-Exit-*)
and *set* (*sourcenodes xs*) $\subseteq$ *set* (*sourcenodes as'*)
and $\forall a'\in$*set xs*. *intra-kind* (*kind* a') $\vee$ ($\exists$ *Q f p*. *kind* $a' = Q\hookleftarrow_p f$)
apply(*erule-tac x*=*as'* **in** *allE*) **by** *auto*
from ‹*sourcenode* $a' = n$› ‹*valid-edge* a'› ‹*kind* $a' = Q\hookleftarrow_p f$›
‹*targetnode* $a' -xs\rightarrow_{\surd}*$ (*-Exit-*)›
have $n -a'\#xs\rightarrow_{\surd}*$ (*-Exit-*)
by(*fastforce intro*:*Cons-path simp*:*vp-def valid-path-def*)
moreover
from ‹*set* (*sourcenodes xs*) $\subseteq$ *set* (*sourcenodes as'*)› ‹$as = a'\#as'$›
have *set* (*sourcenodes* ($a'\#xs$)) $\subseteq$ *set* (*sourcenodes as*)
by(*auto simp*:*sourcenodes-def*)
moreover
from ‹$\forall a'\in$*set xs*. *intra-kind* (*kind* a') $\vee$ ($\exists$ *Q f p*. *kind* $a' = Q\hookleftarrow_p f$)›
‹*kind* $a' = Q\hookleftarrow_p f$›
have $\forall a'\in$*set* ($a'\#xs$). *intra-kind* (*kind* a') $\vee$ ($\exists$ *Q f p*. *kind* $a' = Q\hookleftarrow_p f$)
by *fastforce*

ultimately show *?thesis* **by** *blast*
next
assume $\exists\, as'\ as''\ n'.\ as = as'@as'' \wedge as' \neq [] \wedge n\ -as'\rightarrow_{sl}*\ n'$
then obtain *as' as'' n'* **where** $as = as'@as''$ **and** $as' \neq []$
and $n\ -as'\rightarrow_{sl}*\ n'$ **by** *blast*
from ‹$n\ -as\rightarrow_{\surd}*$ (*-Exit-*)› ‹$as = as'@as''$› ‹$as' \neq []$›
have *last*(*targetnodes as'*) $-as''\rightarrow_{\surd}*$ (*-Exit-*)
by(*cases as' rule*:*rev-cases*,*auto intro*:*vp-split simp*:*targetnodes-def*)
from ‹$n\ -as'\rightarrow_{sl}*\ n'$› ‹$as' \neq []$› **have** *last*(*targetnodes as'*) $= n'$
by(*fastforce intro*:*path-targetnode simp*:*slp-def*)
from ‹$as = as'@as''$› ‹$as' \neq []$› **have** *length as''* $<$ *length as* **by** *simp*
with *IH* ‹*last*(*targetnodes as'*) $-as''\rightarrow_{\surd}*$ (*-Exit-*)›
‹*last*(*targetnodes as'*) $= n'$›
obtain *xs* **where** $n'\ -xs\rightarrow_{\surd}*$ (*-Exit-*)
and *set* (*sourcenodes xs*) $\subseteq$ *set* (*sourcenodes as''*)
and $\forall\, a' \in$ *set xs*. *intra-kind* (*kind a'*) $\vee$ ($\exists\, Q\ f\ p.$ *kind a'* $= Q\hookleftarrow_{p} f$)
apply(*erule-tac x*=*as''* **in** *allE*) **by** *auto*
from ‹$n\ -as'\rightarrow_{sl}*\ n'$› **obtain** *ys* **where** $n\ -ys\rightarrow_{\iota}*\ n'$
and *set*(*sourcenodes ys*) $\subseteq$ *set*(*sourcenodes as'*)
by(*erule same-level-path-inner-path*)
from ‹$n\ -ys\rightarrow_{\iota}*\ n'$› ‹$n'\ -xs\rightarrow_{\surd}*$ (*-Exit-*)› **have** $n\ -ys@xs\rightarrow_{\surd}*$ (*-Exit-*)
by(*fastforce intro*:*slp-vp-Append intra-path-slp*)
moreover
from ‹*set* (*sourcenodes xs*) $\subseteq$ *set* (*sourcenodes as''*)›
‹*set*(*sourcenodes ys*) $\subseteq$ *set*(*sourcenodes as'*)› ‹$as = as'@as''$›
have *set* (*sourcenodes* (*ys*@*xs*)) $\subseteq$ *set*(*sourcenodes as*)
by(*auto simp*:*sourcenodes-def*)
moreover
from ‹$\forall\, a' \in$ *set xs*. *intra-kind* (*kind a'*) $\vee$ ($\exists\, Q\ f\ p.$ *kind a'* $= Q\hookleftarrow_{p} f$)›
‹$n\ -ys\rightarrow_{\iota}*\ n'$›
have $\forall\, a' \in$ *set* (*ys*@*xs*). *intra-kind* (*kind a'*) $\vee$ ($\exists\, Q\ f\ p.$ *kind a'* $= Q\hookleftarrow_{p} f$)
by(*fastforce simp*:*intra-path-def*)
ultimately show *?thesis* **by** *blast*
qed
qed
qed
qed

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(*atomize-elim*)
from ‹$n\ -as\rightarrow_{\surd}*$ (*-Exit-*)›
obtain *as'* **where** $n\ -as'\rightarrow_{\surd}*$ (*-Exit-*)
and *set*(*sourcenodes as'*) $\subseteq$ *set*(*sourcenodes as*)
and *all*:$\forall\, a' \in$ *set as'*. *intra-kind*(*kind a'*) $\vee$ ($\exists\, Q\ f\ p.$ *kind a'* $= Q\hookleftarrow_{p} f$)
by(*erule valid-Exit-path-descending-path*)

show $\exists as'\ pex.\ n\ -as'\rightarrow_\iota *\ pex \wedge method\text{-}exit\ pex \wedge$
$set(sourcenodes\ as') \subseteq set(sourcenodes\ as)$
proof(*cases* $\exists a' \in set\ as'.\ \exists Q\ f\ p.\ kind\ a' = Q\hookleftarrow_p f$)
case *True*
then obtain *asx ax asx'* **where** [*simp*]:$as' = asx@ax\#asx'$
and $\exists Q\ f\ p.\ kind\ ax = Q\hookleftarrow_p f$ **and** $\forall a' \in set\ asx.\ \neg\ (\exists Q\ f\ p.\ kind\ a' = Q\hookleftarrow_p f)$
by(*erule split-list-first-propE*)
with *all* **have** $\forall a' \in set\ asx.\ intra\text{-}kind(kind\ a')$ **by** *auto*
from ‹$n\ -as'\rightarrow_{\surd} *$ (*-Exit-*)› **have** $n\ -asx\rightarrow *\ sourcenode\ ax$
and *valid-edge ax* **by**(*auto elim*:*path-split simp*:*vp-def*)
from ‹$n\ -asx\rightarrow *\ sourcenode\ ax$› ‹$\forall a' \in set\ asx.\ intra\text{-}kind(kind\ a')$›
have $n\ -asx\rightarrow_\iota *\ sourcenode\ ax$ **by**(*simp add*:*intra-path-def*)
moreover
from ‹*valid-edge ax*› ‹$\exists Q\ f\ p.\ kind\ ax = Q\hookleftarrow_p f$›
have *method-exit* (*sourcenode ax*) **by**(*fastforce simp*:*method-exit-def*)
moreover
from ‹$set(sourcenodes\ as') \subseteq set(sourcenodes\ as)$›
have $set(sourcenodes\ asx) \subseteq set(sourcenodes\ as)$ **by**(*simp add*:*sourcenodes-def*)
ultimately show *?thesis* **by** *blast*
next
case *False*
with *all* ‹$n\ -as'\rightarrow_{\surd} *$ (*-Exit-*)› **have** $n\ -as'\rightarrow_\iota *$ (*-Exit-*)
by(*fastforce simp*:*vp-def intra-path-def*)
moreover have *method-exit* (*-Exit-*) **by**(*simp add*:*method-exit-def*)
ultimately show *?thesis* **using** ‹$set(sourcenodes\ as') \subseteq set(sourcenodes\ as)$›
by *blast*
qed
qed

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'*$\hookleftarrow_p$*f'*; (*p*,*ins*,*outs*) ∈ *set procs*$]\!]$
$\Longrightarrow$ *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'*)$]\!]$
$\Longrightarrow$ ∃ *Q Q'*. *kind a* = $(Q)_\surd$ ∧ *kind a'* = $(Q')_\surd$ ∧
(∀ *s*. (*Q s* ⟶ ¬ *Q' s*) ∧ (*Q' s* ⟶ ¬ *Q s*))

begin

lemma *CFG-equal-Use-equal-call*:
assumes *valid-edge a* **and** *kind a* = *Q*:*r*$\hookrightarrow_p$*fs* **and** *valid-edge a'*
and *kind a'* = *Q'*:*r'*$\hookrightarrow_{p'}$*fs'* **and** *sourcenode a* = *sourcenode a'*
and *pred* (*kind a*) *s* **and** *pred* (*kind a'*) *s'*
and *snd* (*hd s*) = *snd* (*hd s'*) **and** *length s* = *length s'*
and ∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s' V*
shows *a* = *a'*
proof −
from ‹*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'*›
have *pred* (*kind a*) *s'* **by**(*rule CFG-edge-Uses-pred-equal*)
with ‹*valid-edge a*› ‹*kind a* = *Q*:*r*$\hookrightarrow_p$*fs*› ‹*valid-edge a'*› ‹*kind a'* = *Q'*:*r'*$\hookrightarrow_{p'}$*fs'*›
‹*sourcenode a* = *sourcenode a'*› ‹*pred* (*kind a'*) *s'*›
show *?thesis* **by** −(*rule CFG-call-determ*)
qed

lemma *CFG-call-edge-param-in*:
assumes *valid-edge a* **and** *kind a* = *Q*:*r*$\hookrightarrow_p$*fs* **and** *i* < *length ins*
and (*p*,*ins*,*outs*) ∈ *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 −
from *assms* **have** *params*:(*params fs* (*fst* (*hd s*)))!*i* = (*params fs* (*fst* (*hd s'*)))!*i*
by(*rule CFG-call-edge-params*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*$\hookrightarrow_p$*fs*› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
have [*simp*]:(*THE ins*. ∃ *outs*. (*p*,*ins*,*outs*) ∈ *set procs*) = *ins*
by(*rule formal-in-THE*)
from ‹*pred* (*kind a*) *s*› **obtain** *cf cfs* **where** [*simp*]:*s* = *cf*#*cfs* **by**(*cases s*) *auto*
from ‹*pred* (*kind a*) *s'*› **obtain** *cf' cfs'* **where** [*simp*]:*s'* = *cf'*#*cfs'*
by(*cases s'*) *auto*
from ‹*kind a* = *Q*:*r*$\hookrightarrow_p$*fs*›
have *eqs*:*fst* (*hd* (*transfer* (*kind a*) *s*)) = (*empty*(*ins* [:=] *params fs* (*fst cf*)))
fst (*hd* (*transfer* (*kind a*) *s'*)) = (*empty*(*ins* [:=] *params fs* (*fst cf'*)))
by *simp-all*
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*$\hookrightarrow_p$*fs*› ‹(*p*,*ins*,*outs*) ∈ *set procs*›

have *length fs = length ins* **by**(*rule CFG-call-edge-length*)
from ‹(*p*,*ins*,*outs*) ∈ *set procs*› **have** *distinct ins* **by**(*rule distinct-formal-ins*)
with ‹*i* < *length ins*› ‹*length fs = length ins*›
have (*empty*(*ins* [:=] *params fs* (*fst cf*))) (*ins*!*i*) = (*params fs* (*fst cf*))!*i*
(*empty*(*ins* [:=] *params fs* (*fst cf'*))) (*ins*!*i*) = (*params fs* (*fst cf'*))!*i*
by(*fastforce intro*:*fun-upds-nth*)+
with *eqs* ‹*kind a* = *Q*:*r*↪*pfs*› *params*
show *?thesis* **by** *simp*
qed

lemma *CFG-call-edge-no-param*:
assumes *valid-edge a* **and** *kind a* = *Q*:*r*↪*pfs* **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 −
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*pfs*› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
have [*simp*]:(*THE ins*. ∃ *outs*. (*p*,*ins*,*outs*) ∈ *set procs*) = *ins*
by(*rule formal-in-THE*)
from ‹*pred* (*kind a*) *s*› **obtain** *cf cfs* **where** [*simp*]:*s* = *cf*#*cfs* **by**(*cases s*) *auto*
from ‹*V* ∉ *set ins*› **have** (*empty*(*ins* [:=] *params fs* (*fst cf*))) *V* = *None*
by(*auto dest*:*fun-upds-notin*)
with ‹*kind a* = *Q*:*r*↪*pfs*› **show** *?thesis* **by** *simp*
qed

lemma *CFG-return-edge-param-out*:
assumes *valid-edge a* **and** *kind a* = *Q*↩*pf* **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 −
from ‹*valid-edge a*› ‹*kind a* = *Q*↩*pf*› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
have [*simp*]:(*THE outs*. ∃ *ins*. (*p*,*ins*,*outs*) ∈ *set procs*) = *outs*
by(*rule formal-out-THE*)
from ‹*valid-edge a*› ‹*kind a* = *Q*↩*pf*› ‹(*p*,*ins*,*outs*) ∈ *set procs*› ‹*s* = *cf*#*cfx*#*cfs*›
have *transfer*:*fst* (*hd* (*transfer* (*kind a*) *s*)) =
(*fst cfx*)(*ParamDefs* (*targetnode a*) [:=] *map* (*fst cf*) *outs*)
by(*fastforce intro*:*CFG-return-edge-fun*)
from ‹*valid-edge a*› ‹*kind a* = *Q*↩*pf*› ‹(*p*,*ins*,*outs*) ∈ *set procs*› ‹*s'* = *cf'*#*cfx'*#*cfs'*›
have *transfer'*:*fst* (*hd* (*transfer* (*kind a*) *s'*)) =
(*fst cfx'*)(*ParamDefs* (*targetnode a*) [:=] *map* (*fst cf'*) *outs*)
by(*fastforce intro*:*CFG-return-edge-fun*)
from ‹*state-val s* (*outs*!*i*) = *state-val s'* (*outs*!*i*)› ‹*i* < *length outs*›
‹*s* = *cf*#*cfx*#*cfs*› ‹*s'* = *cf'*#*cfx'*#*cfs'*›
have (*fst cf*) (*outs*!*i*) = (*fst cf'*) (*outs*!*i*) **by** *simp*
from ‹*valid-edge a*› **have** *distinct* (*ParamDefs* (*targetnode a*))

by(*fastforce intro*:*distinct-ParamDefs*)
from ‹*valid-edge a*› ‹*kind a* = *Q*↪*pf*› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
have *length*(*ParamDefs* (*targetnode a*)) = *length outs*
by(*fastforce intro*:*ParamDefs-return-target-length*)
with ‹*i* < *length outs*› ‹*distinct* (*ParamDefs* (*targetnode a*))›
have (*fst cfx*)(*ParamDefs* (*targetnode a*) [:=] *map* (*fst cf*) *outs*)
((*ParamDefs* (*targetnode a*))!*i*) = (*map* (*fst cf*) *outs*)!*i*
and (*fst cfx'*)(*ParamDefs* (*targetnode a*) [:=] *map* (*fst cf'*) *outs*)
((*ParamDefs* (*targetnode a*))!*i*) = (*map* (*fst cf'*) *outs*)!*i*
by(*fastforce intro*:*fun-upds-nth*)+
with *transfer transfer'* ‹(*fst cf*) (*outs*!*i*) = (*fst cf'*) (*outs*!*i*)› ‹*i* < *length outs*›
show *?thesis* **by** *simp*
qed

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 −
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
obtain *Q r p fs* **where** *kind a* = *Q*:*r*↪$_p$*fs*
by(*fastforce dest*!:*only-call-get-return-edges*)
with ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **obtain** *Q' f'* **where** *kind a'* = *Q'*↩$_p$*f'*
by(*fastforce dest*!:*call-return-edges*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹*preds* (*kinds* (*a*#*as*@[*a'*])) *s*› **obtain** *cf cfs* **where** [*simp*]:*s* = *cf*#*cfs*
by(*cases s*,*auto simp*:*kinds-def*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› **obtain** *ins outs*
where (*p*,*ins*,*outs*) ∈ *set procs* **by**(*fastforce dest*!:*callee-in-procs*)
from ‹*kind a* = *Q*:*r*↪$_p$*fs*› **obtain** *cfx* **where** *transfer* (*kind a*) *s* = *cfx*#*cf*#*cfs*
by *simp*
moreover
from ‹*n* −*as*→$_{sl}$* *n'*› **obtain** *cfx'*
where *transfers* (*kinds as*) (*cfx*#*cf*#*cfs*) = *cfx'*#*cf*#*cfs*
by(*fastforce elim*:*slp-callstack-length-equal*)
moreover
from ‹*kind a'* = *Q'*↩$_p$*f'*› ‹*valid-edge a'*› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
have *fst* (*hd* (*transfer* (*kind a'*) (*cfx'*#*cf*#*cfs*))) =
(*fst cf*)(*ParamDefs* (*targetnode a'*) [:=] *map* (*fst cfx'*) *outs*)
by(*simp*,*simp only*:*formal-out-THE*,*fastforce intro*:*CFG-return-edge-fun*)
ultimately have *fst* (*hd* (*transfers* (*kinds* (*a*#*as*@[*a'*])) *s*)) =
(*fst cf*)(*ParamDefs* (*targetnode a'*) [:=] *map* (*fst cfx'*) *outs*)
by(*simp add*:*kinds-def transfers-split*)
with ‹*V* ∉ *set* (*ParamDefs* (*targetnode a'*))› **show** *?thesis*
by(*simp add*:*fun-upds-notin*)
qed

lemma [*dest*!]: $V \in Use\ (\text{-}Entry\text{-}) \Longrightarrow False$
by(*simp add*:*Entry-empty*)

lemma [*dest*!]: $V \in Def\ (\text{-}Entry\text{-}) \Longrightarrow False$
by(*simp add*:*Entry-empty*)

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 −
from ‹$n\ -as\rightarrow_\iota*\ n'$› **have** $n\ -as\rightarrow*\ n'$ **and** $\forall\, a \in set\ as.\ intra\text{-}kind\ (kind\ a)$
by(*simp-all add*:*intra-path-def*)
from *this* ‹$\forall\, n \in set\ (sourcenodes\ as).\ V \notin Def\ n$› ‹*preds* (*kinds as*) *s*›
have *state-val* (*transfers* (*kinds as*) *s*) *V* = *state-val s V*
proof(*induct arbitrary*:*s rule*:*path.induct*)
case (*empty-path n*)
thus *?case* **by**(*simp add*:*sourcenodes-def kinds-def*)
next
case (*Cons-path n*′′ *as n*′ *a n*)
note *IH* = ‹$\bigwedge s.\ [\![\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a);$
$\forall\, n{\in}set\ (sourcenodes\ as).\ V \notin Def\ n;\ preds\ (kinds\ as)\ s]\!]$
$\Longrightarrow$ *state-val* (*transfers* (*kinds as*) *s*) *V* = *state-val s V*›
from ‹*preds* (*kinds* (*a*#*as*)) *s*› **have** *pred* (*kind a*) *s*
and *preds* (*kinds as*) (*transfer* (*kind a*) *s*) **by**(*simp-all add*:*kinds-def*)
from ‹$\forall\, n{\in}set\ (sourcenodes\ (a\#as)).\ V \notin Def\ n$›
have *noDef*:$V \notin Def\ (sourcenode\ a)$
and *all*:$\forall\, n{\in}set\ (sourcenodes\ as).\ V \notin Def\ n$
by(*auto simp*:*sourcenodes-def*)
from ‹$\forall\, a{\in}set\ (a\#as).\ intra\text{-}kind\ (kind\ a)$›
have *intra-kind* (*kind a*) **and** *all*′:$\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a)$
by *auto*
from ‹*valid-edge a*› *noDef* ‹*intra-kind* (*kind a*)› ‹*pred* (*kind a*) *s*›
have *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*
by −(*rule CFG-intra-edge-no-Def-equal*)
with *IH*[*OF all*′ *all* ‹*preds* (*kinds as*) (*transfer* (*kind a*) *s*)›] **show** *?case*
by(*simp add*:*kinds-def*)
qed
thus *?thesis* **by** *blast*
qed

lemma *slpa-preds*:
$[\![$*same-level-path-aux cs as*; *s* = *cfsx*@*cf*#*cfs*; *s*′ = *cfsx*@*cf*#*cfs*′;
length cfs = *length cfs*′; $\forall\, a \in set\ as.\ valid\text{-}edge\ a$; *length cs* = *length cfsx*;

```
    preds (kinds as) s⟧
  ⟹ preds (kinds as) s'
proof(induct arbitrary:s s' cf cfsx rule:slpa-induct)
  case (slpa-empty cs) thus ?case by(simp add:kinds-def)
next
  case (slpa-intra cs a as)
  note IH = ‹⋀s s' cf cfsx. ⟦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'›
  from ‹∀ a∈set (a#as). valid-edge a› have valid-edge a
    and ∀ a ∈ set as. valid-edge a by simp-all
  from ‹preds (kinds (a#as)) s› have pred (kind a) s
    and preds (kinds as) (transfer (kind a) s) by(simp-all add:kinds-def)
  show ?case
  proof(cases cfsx)
    case Nil
    with ‹length cs = length cfsx› have length cs = length [] by simp
    from Nil ‹s = cfsx@cf#cfs› ‹s' = cfsx@cf#cfs'› ‹intra-kind (kind a)›
    obtain cfx where transfer (kind a) s = []@cfx#cfs
      and transfer (kind a) s' = []@cfx#cfs'
      by(cases kind a,auto simp:kinds-def intra-kind-def)
    from IH[OF this ‹length cfs = length cfs'› ‹∀ a ∈ set as. valid-edge a›
      ‹length cs = length []› ‹preds (kinds as) (transfer (kind a) s)›]
    have preds (kinds as) (transfer (kind a) s') .
    moreover
   from Nil ‹valid-edge a› ‹pred (kind a) s› ‹s = cfsx@cf#cfs› ‹s' = cfsx@cf#cfs'›
      ‹length cfs = length cfs'›
    have pred (kind a) s' by(fastforce intro:CFG-edge-Uses-pred-equal)
    ultimately show ?thesis by(simp add:kinds-def)
  next
    case (Cons x xs)
    with ‹s = cfsx@cf#cfs› ‹s' = cfsx@cf#cfs'› ‹intra-kind (kind a)›
    obtain cfx where transfer (kind a) s = (cfx#xs)@cf#cfs
      and transfer (kind a) s' = (cfx#xs)@cf#cfs'
      by(cases kind a,auto simp:kinds-def intra-kind-def)
    from IH[OF this ‹length cfs = length cfs'› ‹∀ a ∈ set as. valid-edge a› -
      ‹preds (kinds as) (transfer (kind a) s)›] ‹length cs = length cfsx› Cons
    have preds (kinds as) (transfer (kind a) s') by simp
    moreover
   from Cons ‹valid-edge a› ‹pred (kind a) s› ‹s = cfsx@cf#cfs› ‹s' = cfsx@cf#cfs'›
      ‹length cfs = length cfs'›
    have pred (kind a) s' by(fastforce intro:CFG-edge-Uses-pred-equal)
    ultimately show ?thesis by(simp add:kinds-def)
  qed
next
  case (slpa-Call cs a as Q r p fs)
  note IH = ‹⋀s s' cf cfsx. ⟦s = cfsx@cf#cfs; s' = cfsx@cf#cfs';
    length cfs = length cfs'; ∀ a ∈ set as. valid-edge a; length (a#cs) = length cfsx;
    preds (kinds as) s⟧ ⟹ preds (kinds as) s'›
```

```
  from ‹∀ a∈set (a#as). valid-edge a› have valid-edge a
    and ∀ a ∈ set as. valid-edge a by simp-all
  from ‹preds (kinds (a#as)) s› have pred (kind a) s
    and preds (kinds as) (transfer (kind a) s) by(simp-all add:kinds-def)
  from ‹kind a = Q:r↪pfs› ‹s = cfsx@cf#cfs› ‹s' = cfsx@cf#cfs'› obtain cfx
    where transfer (kind a) s = (cfx#cfsx)@cf#cfs
    and transfer (kind a) s' = (cfx#cfsx)@cf#cfs' by(cases cfsx) auto
  from IH[OF this ‹length cfs = length cfs'› ‹∀ a ∈ set as. valid-edge a› -
    ‹preds (kinds as) (transfer (kind a) s)›] ‹length cs = length cfsx›
  have preds (kinds as) (transfer (kind a) s') by simp
  moreover
  from ‹valid-edge a› ‹pred (kind a) s› ‹s = cfsx@cf#cfs› ‹s' = cfsx@cf#cfs'›
    ‹length cfs = length cfs'› have pred (kind a) s'
    by(cases cfsx)(auto intro:CFG-edge-Uses-pred-equal)
  ultimately show ?case by(simp add:kinds-def)
next
  case (slpa-Return cs a as Q p f c' cs')
  note IH = ‹⋀s s' cf cfsx. ⟦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'›
  from ‹∀ a∈set (a#as). valid-edge a› have valid-edge a
    and ∀ a ∈ set as. valid-edge a by simp-all
  from ‹preds (kinds (a#as)) s› have pred (kind a) s
    and preds (kinds as) (transfer (kind a) s) by(simp-all add:kinds-def)
  show ?case
  proof(cases cs')
    case Nil
    with ‹cs = c'#cs'› ‹s = cfsx@cf#cfs› ‹s' = cfsx@cf#cfs'›
      ‹length cs = length cfsx›
    obtain cf' where s = cf'#cf#cfs and s' = cf'#cf#cfs' by(cases cfsx) auto
    with ‹kind a = Q↩pf› obtain cf'' where transfer (kind a) s = []@cf''#cfs
      and transfer (kind a) s' = []@cf''#cfs' by auto
    from IH[OF this ‹length cfs = length cfs'› ‹∀ a ∈ set as. valid-edge a› -
      ‹preds (kinds as) (transfer (kind a) s)›] Nil
    have preds (kinds as) (transfer (kind a) s') by simp
    moreover
    from ‹valid-edge a› ‹pred (kind a) s› ‹s = cfsx@cf#cfs› ‹s' = cfsx@cf#cfs'›
      ‹length cfs = length cfs'› have pred (kind a) s'
      by(cases cfsx)(auto intro:CFG-edge-Uses-pred-equal)
    ultimately show ?thesis by(simp add:kinds-def)
  next
    case (Cons cx csx)
   with ‹cs = c'#cs'› ‹length cs = length cfsx› ‹s = cfsx@cf#cfs› ‹s' = cfsx@cf#cfs'›
    obtain x x' xs where s = (x#x'#xs)@cf#cfs and s' = (x#x'#xs)@cf#cfs'
      and length xs = length csx
      by(cases cfsx,auto,case-tac list,fastforce+)
   with ‹kind a = Q↩pf› obtain cf' where transfer (kind a) s = (cf'#xs)@cf#cfs
      and transfer (kind a) s' = (cf'#xs)@cf#cfs'
      by fastforce
```

from *IH*[*OF this* ‹*length cfs* = *length cfs'*› ‹∀ *a* ∈ *set as*. *valid-edge a*› -
‹*preds* (*kinds as*) (*transfer* (*kind a*) *s*)›] *Cons* ‹*length xs* = *length csx*›
have *preds* (*kinds as*) (*transfer* (*kind a*) *s'*) **by** *simp*
moreover
from ‹*valid-edge a*› ‹*pred* (*kind a*) *s*› ‹*s* = *cfsx*@*cf*#*cfs*› ‹*s'* = *cfsx*@*cf*#*cfs'*›
‹*length cfs* = *length cfs'*› **have** *pred* (*kind a*) *s'*
by(*cases cfsx*)(*auto intro*:*CFG-edge-Uses-pred-equal*)
ultimately show *?thesis* **by**(*simp add*:*kinds-def*)
qed
qed

lemma *slp-preds*:
assumes *n* −*as*→$_{sl}$* *n'* **and** *preds* (*kinds as*) (*cf*#*cfs*)
and *length cfs* = *length cfs'*
shows *preds* (*kinds as*) (*cf*#*cfs'*)
proof −
from ‹*n* −*as*→$_{sl}$* *n'*› **have** *n* −*as*→* *n'* **and** *same-level-path-aux* [] *as*
by(*simp-all add*:*slp-def same-level-path-def*)
from ‹*n* −*as*→* *n'*› **have** ∀ *a* ∈ *set as*. *valid-edge a* **by**(*rule path-valid-edges*)
with ‹*same-level-path-aux* [] *as*› ‹*preds* (*kinds as*) (*cf*#*cfs*)›
‹*length cfs* = *length cfs'*›
show *?thesis* **by**(*fastforce elim*!:*slpa-preds*)
qed
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*!]: $V \in Use\ (\text{-}Exit\text{-}) \Longrightarrow False$
by(*simp add*:*Exit-empty*)

lemma *Exit-Def-empty* [*dest*!]: $V \in Def\ (\text{-}Exit\text{-}) \Longrightarrow False$
by(*simp add*:*Exit-empty*)

end

end

1.4 CFG and semantics conform

theory *SemanticsCFG* **imports** *CFG* **begin**

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

end

1.5 Return and their corresponding call nodes

theory *ReturnAndCallNodes* **imports** *CFG* **begin**

context *CFG* **begin**

1.5.1 Defining *return-node*

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

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(*rule ex-ex1I*)
from ‹*return-node n*›
show $\exists\, n'\ a\ a'.$ *valid-edge a* $\wedge$ $n' =$ *sourcenode a* $\wedge$ *valid-edge a'* $\wedge$
$a' \in$ *get-return-edges a* $\wedge$ $n =$ *targetnode a'*
by(*simp add*:*return-node-def*) *blast*
next
fix n' *nx*
assume $\exists\, a\ a'.$ *valid-edge a* $\wedge$ $n' =$ *sourcenode a* $\wedge$ *valid-edge a'* $\wedge$
$a' \in$ *get-return-edges a* $\wedge$ $n =$ *targetnode a'*
and $\exists\, a\ a'.$ *valid-edge a* $\wedge$ *nx* $=$ *sourcenode a* $\wedge$ *valid-edge a'* $\wedge$
$a' \in$ *get-return-edges a* $\wedge$ $n =$ *targetnode a'*
then obtain *a a' ax ax'* **where** *valid-edge a* **and** $n' =$ *sourcenode a*
and *valid-edge a'* **and** $a' \in$ *get-return-edges a*
and $n =$ *targetnode a'* **and** *valid-edge ax* **and** *nx* $=$ *sourcenode ax*
and *valid-edge ax'* **and** $ax' \in$ *get-return-edges ax*
and $n =$ *targetnode ax'*
by *blast*
from ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*› **obtain** a''
where *intra-edge1*:*valid-edge a''* *sourcenode a''* $=$ *sourcenode a*
targetnode a'' $=$ *targetnode a'* *kind a''* $= (\lambda cf.\ False)_{\surd}$
by(*fastforce dest*:*call-return-node-edge*)
from ‹*valid-edge ax*› ‹$ax' \in$ *get-return-edges ax*› **obtain** ax''
where *intra-edge2*:*valid-edge ax''* *sourcenode ax''* $=$ *sourcenode ax*
targetnode ax'' $=$ *targetnode ax'* *kind ax''* $= (\lambda cf.\ False)_{\surd}$
by(*fastforce dest*:*call-return-node-edge*)
from ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*›
obtain *Q r p fs* **where** *kind a* $= Q$:$r \hookrightarrow_p fs$
by(*fastforce dest*!:*only-call-get-return-edges*)
with ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*› **obtain** Q' *p f'*
where *kind a'* $= Q' \hookleftarrow_p f'$ **by**(*fastforce dest*!:*call-return-edges*)
with ‹*valid-edge a'*›
have $\exists!a''.$ *valid-edge a''* $\wedge$ *targetnode a''* $=$ *targetnode a'* $\wedge$ *intra-kind*(*kind a''*)
by(*rule return-only-one-intra-edge*)
with *intra-edge1 intra-edge2* ‹$n =$ *targetnode a'*› ‹$n =$ *targetnode ax'*›
have $a'' = ax''$ **by**(*fastforce simp*:*intra-kind-def*)
with ‹*sourcenode a''* $=$ *sourcenode a*› ‹*sourcenode ax''* $=$ *sourcenode ax*›
‹$n' =$ *sourcenode a*› ‹*nx* $=$ *sourcenode ax*›
show $n' = nx$ **by** *simp*
qed

lemma *return-node-THE-call-node*:
⟦*return-node n*; *valid-edge a*; *valid-edge a'*; $a' \in$ *get-return-edges a*;

$n = targetnode\ a'⟧$
$\Longrightarrow (THE\ n'.\ \exists\, a\ a'.\ valid\text{-}edge\ a \wedge n' = sourcenode\ a \wedge valid\text{-}edge\ a' \wedge$
$a' \in get\text{-}return\text{-}edges\ a \wedge n = targetnode\ a') = sourcenode\ a$
by(*fastforce intro*!:*the1-equality return-node-determines-call-node*)

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

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

lemma *return-node-call-of-return-node*:
return-node n $\Longrightarrow \exists!n'.$ *call-of-return-node n n'*
by −(*frule return-node-determines-call-node*,*unfold call-of-return-node-def*,*simp*)

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 −
from ‹*call-of-return-node n n'*› **have** *return-node n*
by(*simp add*:*call-of-return-node-def*)
hence $\exists!n'.$ *call-of-return-node n n'* **by**(*rule return-node-call-of-return-node*)
with ‹*call-of-return-node n n'*› ‹*call-of-return-node n n''*›
show *?thesis* **by** *auto*
qed

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(*induct cs arbitrary*:*m rs*)
case *Nil* **thus** *?case* **by** *fastforce*
next
case (*Cons c' cs'*)
note *IH* = ‹$\bigwedge$*m rs*. ⟦*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*))›
from ‹*length rs* = *length* (*c'* # *cs'*)› **obtain** *r' rs'* **where** *rs* = *r'* # *rs'*
and *length rs'* = *length cs'* **by**(*cases rs*) *auto*
with ‹$\forall\, i<$*length rs*. *rs* ! *i* $\in$ *get-return-edges* ((*c'* # *cs'*) ! *i*)›
have $\forall\, i<$*length rs'*. *rs'* ! *i* $\in$ *get-return-edges* (*cs'* ! *i*)
and *r'* $\in$ *get-return-edges c'* **by** *auto*
from ‹*valid-call-list* (*c'*#*cs'*) *m*› **have** *valid-edge c'*
by(*fastforce simp*:*valid-call-list-def*)

from *this* ‹$r' \in$ *get-return-edges* c'› **have** *get-proc* (*sourcenode* c') = *get-proc* (*targetnode* r') **by**(*rule get-proc-get-return-edge*) **from** ‹*valid-call-list* (c'#cs') m› **have** *valid-call-list* cs' (*sourcenode* c') **apply**(*clarsimp simp*:*valid-call-list-def*) **apply**(*erule-tac* x=c'#cs' **in** *allE*) **apply** *clarsimp* **by**(*case-tac* cs')(*auto simp*:*sourcenodes-def*) **from** ‹*valid-return-list rs m*› ‹$rs = r' \# rs'$› ‹*get-proc* (*sourcenode* c') = *get-proc* (*targetnode* r')› **have** *valid-return-list* rs' (*sourcenode* c') **apply**(*clarsimp simp*:*valid-return-list-def*) **apply**(*erule-tac* x=r'#cs' **in** *allE*) **apply** *clarsimp* **by**(*case-tac* cs')(*auto simp*:*targetnodes-def*) **from** *IH*[*OF* ‹*valid-call-list* cs' (*sourcenode* c')› ‹*valid-return-list* rs' (*sourcenode* c')› ‹$\forall i$<*length* rs'. rs' ! $i \in$ *get-return-edges* (cs' ! i)› ‹*length* rs' = *length* cs'›] **have** *all*:$\forall i$<*length* cs'. *call-of-return-node* (*targetnodes* rs' ! i) (*sourcenode* (cs' ! i)) **.** **from** ‹*valid-edge* c'› ‹$r' \in$ *get-return-edges* c'› **have** *valid-edge* r' **by**(*rule get-return-edges-valid*) **from** ‹*valid-edge* r'› ‹*valid-edge* c'› ‹$r' \in$ *get-return-edges* c'› **have** *return-node* (*targetnode* r') **by**(*fastforce simp*:*return-node-def*) **with** ‹*valid-edge* c'› ‹$r' \in$ *get-return-edges* c'› ‹*valid-edge* r'› **have** *call-of-return-node* (*targetnode* r') (*sourcenode* c') **by**(*simp add*:*call-of-return-node-def*) *blast* **with** *all* ‹$rs = r' \# rs'$› **show** *?case* **by** *auto*(*case-tac* i,*auto simp*:*targetnodes-def*)
qed

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'; ∀ $nx \in$ *set*(*sourcenodes as*). $nx \notin S$; $n' \in S$⟧ ⟹ $n' \in$ *obs-intra* n S

lemma *obs-intraE*:
assumes $n' \in$ *obs-intra n S*
obtains *as* **where** $n\ -as\rightarrow_\iota *\ n'$ **and** $\forall\, nx \in set(sourcenodes\ as).\ nx \notin S$ **and** $n' \in S$
using ‹$n' \in$ *obs-intra n S*›
by(*fastforce elim*:*obs-intra.cases*)

lemma *n-in-obs-intra*:
assumes *valid-node n* **and** $n \in S$ **shows** *obs-intra n S* $= \{n\}$
proof −
from ‹*valid-node n*› **have** $n\ -[]\rightarrow *\ n$ **by**(*rule empty-path*)
hence $n\ -[]\rightarrow_\iota *\ n$ **by**(*simp add*:*intra-path-def*)
with ‹$n \in S$› **have** $n \in$ *obs-intra n S*
by(*fastforce elim*:*obs-intra-elem simp*:*sourcenodes-def*)
{ fix n' **assume** $n' \in$ *obs-intra n S*
have $n' = n$
proof(*rule ccontr*)
assume $n' \neq n$
from ‹$n' \in$ *obs-intra n S*› **obtain** *as* **where** $n\ -as\rightarrow_\iota *\ n'$
and $\forall\, nx \in set(sourcenodes\ as).\ nx \notin S$
and $n' \in S$ **by**(*fastforce elim*:*obs-intra.cases*)
from ‹$n\ -as\rightarrow_\iota *\ n'$› **have** $n\ -as\rightarrow *\ n'$ **by**(*simp add*:*intra-path-def*)
from *this* ‹$\forall\, nx \in set(sourcenodes\ as).\ nx \notin S$› ‹$n' \neq n$› ‹$n \in S$›
show *False*
proof(*induct rule*:*path.induct*)
case (*Cons-path n'' as n' a n*)
from ‹$\forall\, nx{\in}set\ (sourcenodes\ (a\#as)).\ nx \notin S$› ‹*sourcenode a* $= n$›
have $n \notin S$ **by**(*simp add*:*sourcenodes-def*)
with ‹$n \in S$› **show** *False* **by** *simp*
qed *simp*
qed }
with ‹$n \in$ *obs-intra n S*› **show** *?thesis* **by** *fastforce*
qed

lemma *in-obs-intra-valid*:
assumes $n' \in$ *obs-intra n S* **shows** *valid-node n* **and** *valid-node* n'
using ‹$n' \in$ *obs-intra n S*›
by(*auto elim*!:*obs-intraE intro*:*path-valid-node simp*:*intra-path-def*)

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

then obtain *as* **where** *targetnode a* $-as\rightarrow_\iota*$ *n*
and *all*:$\forall\, nx \in set(sourcenodes\ as).\ nx \notin S$ **and** $n \in S$ **by**(*erule obs-intraE*)
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)› ‹*targetnode a* $-as\rightarrow_\iota*$ *n*›
have *sourcenode a* $-[a]@as\rightarrow_\iota*$ *n* **by**(*fastforce intro*:*Cons-path simp*:*intra-path-def*)
moreover
from *all* ‹*sourcenode a* $\notin$ *S*› **have** $\forall\, nx \in set(sourcenodes\ (a\#as)).\ nx \notin S$
by(*simp add*:*sourcenodes-def*)
ultimately show $n \in$ *obs-intra* (*sourcenode a*) *S* **using** ‹$n \in S$›
by(*fastforce intro*:*obs-intra-elem*)
qed

lemma *path-obs-intra-subset*:
assumes $n\ -as\rightarrow_\iota*\ n'$ **and** $\forall\, n' \in set(sourcenodes\ as).\ n' \notin S$
shows *obs-intra n' S* $\subseteq$ *obs-intra n S*
proof −
from ‹$n\ -as\rightarrow_\iota*\ n'$› **have** $n\ -as\rightarrow*\ n'$ **and** $\forall\, a \in set\ as.\ intra\text{-}kind\ (kind\ a)$
by(*simp-all add*:*intra-path-def*)
from *this* ‹$\forall\, n' \in set(sourcenodes\ as).\ n' \notin S$› **show** *?thesis*
proof(*induct rule*:*path.induct*)
case (*Cons-path n'' as n' a n*)
note *IH* = ‹⟦$\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a);\ \forall\, n'{\in}set\ (sourcenodes\ as).\ n' \notin S$⟧
$\Longrightarrow$ *obs-intra n' S* $\subseteq$ *obs-intra n'' S*›
from ‹$\forall\, n'{\in}set\ (sourcenodes\ (a\#as)).\ n' \notin S$›
have *all*:$\forall\, n'{\in}set\ (sourcenodes\ as).\ n' \notin S$ **and** *sourcenode a* $\notin$ *S*
by(*simp-all add*:*sourcenodes-def*)
from ‹$\forall\, a \in set\ (a\#as).\ intra\text{-}kind\ (kind\ a)$›
have *intra-kind* (*kind a*) **and** $\forall\, a \in set\ as.\ intra\text{-}kind\ (kind\ a)$
by(*simp-all add*:*intra-path-def*)
from *IH*[*OF* ‹$\forall\, a \in set\ as.\ intra\text{-}kind\ (kind\ a)$› *all*]
have *obs-intra n' S* $\subseteq$ *obs-intra n'' S* **.**
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)› ‹*targetnode a* = *n''*›
‹*sourcenode a* = *n*› ‹*sourcenode a* $\notin$ *S*›
have *obs-intra n'' S* $\subseteq$ *obs-intra n S* **by**(*fastforce dest*:*edge-obs-intra-subset*)
with ‹*obs-intra n' S* $\subseteq$ *obs-intra n'' S*› **show** *?case* **by** *fastforce*
qed *simp*
qed

lemma *path-ex-obs-intra*:
assumes $n\ -as\rightarrow_\iota*\ n'$ **and** $n' \in S$
obtains *m* **where** $m \in$ *obs-intra n S*
proof(*atomize-elim*)
show $\exists\, m.\ m \in$ *obs-intra n S*
proof(*cases* $\forall\, nx \in set(sourcenodes\ as).\ nx \notin S$)
case *True*
with ‹$n\ -as\rightarrow_\iota*\ n'$› ‹$n' \in S$› **have** $n' \in$ *obs-intra n S* **by** −(*rule obs-intra-elem*)
thus *?thesis* **by** *fastforce*

```
  next
    case False
    hence ∃ nx ∈ set(sourcenodes as). nx ∈ S by fastforce
    then obtain nx ns ns' where sourcenodes as = ns@nx#ns'
      and nx ∈ S and ∀ n' ∈ set ns. n' ∉ S
      by(fastforce elim!:split-list-first-propE)
    from ‹sourcenodes as = ns@nx#ns'› obtain as' a as''
      where ns = sourcenodes as'
      and as = as'@a#as'' and sourcenode a = nx
      by(fastforce elim:map-append-append-maps simp:sourcenodes-def)
    with ‹n -as→ι* n'› have n -as'→ι* nx
      by(fastforce dest:path-split simp:intra-path-def)
    with ‹nx ∈ S› ‹∀ n' ∈ set ns. n' ∉ S› ‹ns = sourcenodes as'›
    have nx ∈ obs-intra n S by(fastforce intro:obs-intra-elem)
    thus ?thesis by fastforce
  qed
qed
```

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 (λnx. 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(induct ns arbitrary:nsx)
case (Cons x xs)
  note IH = ‹⋀nsx. ⟦xs = 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 xs S›
  note 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)›
  show ?case
  proof(cases nsx)
    case Nil
    with ‹x#xs = nsx@n#nsx'› have n = x and xs = nsx' by simp-all
    with ‹n' ∈ obs-intra n S›
      ‹∀ nx∈set nsx'. ∃ nx'. call-of-return-node nx nx' ∧ nx' ∈ S›
    show ?thesis by(fastforce simp:Let-def)
  next
```

```
    case (Cons z zs)
    with ‹x#xs = nsx@n#nsx'› have [simp]:x = z xs = zs@n#nsx' by simp-all
    from nsx Cons
    have ∀ xs x xs'. zs = xs @ x # xs' ∧ obs-intra x S ≠ {} ⟶
      (∃ x''∈set (xs' @ [n]). ∃ nx. call-of-return-node x'' nx ∧ nx ∉ S)
      by clarsimp(erule-tac x=z#xs in allE,auto)
    from IH[OF ‹xs = zs@n#nsx'› this] have n'#nsx' ∈ obs xs S by simp
    show ?thesis
    proof(cases obs-intra z S = {})
      case True
      with Cons ‹n'#nsx' ∈ obs xs S› show ?thesis by(simp add:Let-def)
    next
      case False
      from nsx Cons
      have obs-intra z S ≠ {} ⟶
        (∃ x''∈set (zs @ [n]). ∃ nx. call-of-return-node x'' nx ∧ nx ∉ S)
        by clarsimp(erule-tac x=[] in allE,auto)
      with False have ∃ x''∈set (zs @ [n]). ∃ nx. call-of-return-node x'' nx ∧ nx ∉
S
        by simp
      with ‹xs = zs@n#nsx'›
      have ∃ n' ∈ set xs. ∃ nx. call-of-return-node n' nx ∧ nx ∉ S by fastforce
      with Cons ‹n'#nsx' ∈ obs xs S› show ?thesis by(simp add:Let-def)
    qed
  qed
qed simp



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(atomize-elim)
  from ‹ns' ∈ obs ns S› ‹∀ n ∈ set (tl ns). return-node n›
  show ∃ nsx n nsx' n'. ns = nsx @ n # nsx' ∧ ns' = n' # nsx' ∧
    n' ∈ obs-intra n S ∧ (∀ nx∈set nsx'. ∃ nx'. call-of-return-node nx nx' ∧ nx' ∈
S) ∧
    (∀ 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(induct ns)
    case Nil thus ?case by simp
  next
    case (Cons nx ns'')
    note IH = ‹⟦ns' ∈ obs ns'' S; ∀ a∈set (tl ns''). return-node a⟧
      ⟹ ∃ nsx n nsx' n'. ns'' = nsx @ n # nsx' ∧ ns' = n' # nsx' ∧
```

```
    n' ∈ obs-intra n S ∧
    (∀ nx∈set nsx'. ∃ nx'. call-of-return-node nx nx' ∧ nx' ∈ S) ∧
    (∀ xs x xs'. nsx = xs @ x # xs' ∧ obs-intra x S ≠ {} ⟶
      (∃ x''∈set (xs' @ [n]). ∃ nx. call-of-return-node x'' nx ∧ nx ∉ S))⟩
  from ⟨∀ a∈set (tl (nx # ns'')). return-node a⟩ have ∀ n ∈ set ns''. return-node
n
    by simp
  show ?case
  proof(cases ns'')
    case Nil
    with ⟨ns' ∈ obs (nx#ns'') S⟩ obtain x where ns' = [x] and x ∈ obs-intra
nx S
      by(auto simp:Let-def split:split-if-asm)
    with Nil show ?thesis by fastforce
  next
    case Cons
    with ⟨∀ n ∈ set ns''. return-node n⟩ have ∀ a∈set (tl ns''). return-node a
      by simp
    show ?thesis
    proof(cases ∃ n' ∈ set ns''. ∃ nx'. call-of-return-node n' nx' ∧ nx' ∉ S)
      case True
      with ⟨ns' ∈ obs (nx#ns'') S⟩ have ns' ∈ obs ns'' S by simp
      from IH[OF this ⟨∀ a∈set (tl ns''). return-node a⟩]
      obtain nsx n nsx' n' where split:ns'' = nsx @ n # nsx'
        ns' = n' # nsx' n' ∈ obs-intra n S
        ∀ nx∈set nsx'. ∃ nx'. call-of-return-node nx nx' ∧ nx' ∈ S
        and imp:∀ xs x xs'. nsx = xs @ x # xs' ∧ obs-intra x S ≠ {} ⟶
        (∃ x''∈set (xs' @ [n]). ∃ nx. call-of-return-node x'' nx ∧ nx ∉ S)
        by blast
      from True ⟨ns'' = nsx @ n # nsx'⟩
        ⟨∀ nx∈set nsx'. ∃ nx'. call-of-return-node nx nx' ∧ nx' ∈ S⟩
      have (∃ nx'. call-of-return-node n nx' ∧ nx' ∉ S) ∨
        (∃ n'∈set nsx. ∃ nx'. call-of-return-node n' nx' ∧ nx' ∉ S) by fastforce
      thus ?thesis
      proof
        assume ∃ nx'. call-of-return-node n nx' ∧ nx' ∉ S
        with split show ?thesis by clarsimp
      next
        assume ∃ n'∈set nsx. ∃ nx'. call-of-return-node n' nx' ∧ nx' ∉ S
        with imp have ∀ xs x xs'. nx#nsx = xs @ x # xs' ∧ obs-intra x S ≠ {}
⟶
        (∃ x''∈set (xs' @ [n]). ∃ nx. call-of-return-node x'' nx ∧ nx ∉ S)
          apply clarsimp apply(case-tac xs) apply auto
          by(erule-tac x=list in allE,auto)+
        with split Cons show ?thesis by auto
      qed
    next
      case False
      hence ∀ n'∈set ns''. ∀ nx'. call-of-return-node n' nx' ⟶ nx' ∈ S by simp
```

```
      show ?thesis
      proof(cases obs-intra nx S = {})
        case True
        with ‹ns' ∈ obs (nx#ns'') S› have ns' ∈ obs ns'' S by simp
        from IH[OF this ‹∀ a∈set (tl ns''). return-node a›]
        obtain nsx n nsx' n' where split:ns'' = nsx @ n # nsx'
          ns' = n' # nsx' n' ∈ obs-intra n S
          ∀ nx∈set nsx'. ∃ nx'. call-of-return-node nx nx' ∧ nx' ∈ S
          and imp:∀ xs x xs'. nsx = xs @ x # xs' ∧ obs-intra x S ≠ {} ⟶
          (∃ x''∈set (xs' @ [n]). ∃ nx. call-of-return-node x'' nx ∧ nx ∉ S)
          by blast
        from True imp Cons
        have ∀ xs x xs'. nx#nsx = xs @ x # xs' ∧ obs-intra x S ≠ {} ⟶
          (∃ x''∈set (xs' @ [n]). ∃ nx. call-of-return-node x'' nx ∧ nx ∉ S)
          by clarsimp(case-tac xs,clarsimp+,erule-tac x=list in allE,auto)
        with split Cons show ?thesis by auto
      next
        case False
        with ‹∀ n'∈set ns''. ∀ nx'. call-of-return-node n' nx' ⟶ nx' ∈ S›
          ‹ns' ∈ obs (nx # ns'') S›
        obtain nx'' where ns' = nx''#ns'' and nx'' ∈ obs-intra nx S
        by(fastforce simp:Let-def split:split-if-asm)
        { fix n' assume n'∈set ns''
          with ‹∀ n ∈ set ns''. return-node n› have return-node n' by simp
          hence ∃!n''. call-of-return-node n' n''
            by(rule return-node-call-of-return-node)
          from ‹n'∈set ns''›
            ‹∀ n'∈set ns''. ∀ nx'. call-of-return-node n' nx' ⟶ nx' ∈ S›
          have ∀ nx'. call-of-return-node n' nx' ⟶ nx' ∈ S by simp
          with ‹∃!n''. call-of-return-node n' n''›
          have ∃ n''. call-of-return-node n' n'' ∧ n'' ∈ S by fastforce }
        with ‹ns' = nx''#ns''› ‹nx'' ∈ obs-intra nx S› show ?thesis by fastforce
      qed
    qed
  qed
qed


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''. z'' ∈ set (zs'@[y]). ∃ ny. call-of-return-node z'' ny ∧ ny ∉ S)
  shows xs = ys ∧ x = y ∧ xs' = ys'
using assms
proof(induct xs arbitrary:ys)
  case Nil
  note impy = ⟨∀ zs z zs'. ys = zs@z#zs' ∧ obs-intra z S ≠ {}
    ⟶ (∃ z''. z'' ∈ set (zs'@[y]). ∃ ny. call-of-return-node z'' ny ∧ ny ∉ S)⟩
  show ?case
  proof(cases ys = [])
    case True
    with Nil ⟨[]@x#xs' = ys@y#ys'⟩ show ?thesis by simp
  next
    case False
    with ⟨[] @ x # xs' = ys @ y # ys'⟩
   obtain zs where x#zs = ys and xs' = zs@y#ys' by(auto simp:Cons-eq-append-conv)
    from ⟨x#zs = ys⟩ ⟨obs-intra x S ≠ {}⟩ impy
    have ∃ z''. z'' ∈ set (zs@[y]). ∃ ny. call-of-return-node z'' ny ∧ ny ∉ S
      by blast
    with ⟨xs' = zs@y#ys'⟩ ⟨∀ x'. x' ∈ set xs'. ∃ x''. call-of-return-node x' x'' ∧ x'' ∈
S⟩
    have False by fastforce
    thus ?thesis by simp
  qed
next
  case (Cons w ws)
  note IH = ⟨⋀ys. ⟦ws @ x # xs' = ys @ y # ys'; obs-intra x S ≠ {};
    ∀ x'∈set xs'. ∃ x''. call-of-return-node x' x'' ∧ x'' ∈ S;
    ∀ zs z zs'. ws = zs @ z # zs' ∧ obs-intra z S ≠ {} ⟶
      (∃ z''∈set (zs' @ [x]). ∃ nx. call-of-return-node z'' nx ∧ nx ∉ S);
    obs-intra y S ≠ {}; ∀ y'∈set ys'. ∃ y''. call-of-return-node y' y'' ∧ y'' ∈ S;
    ∀ zs z zs'. ys = zs @ z # zs' ∧ obs-intra z S ≠ {} ⟶
      (∃ z''∈set (zs' @ [y]). ∃ ny. call-of-return-node z'' ny ∧ ny ∉ S)⟧
    ⟹ ws = ys ∧ x = y ∧ xs' = ys'⟩
  note impw = ⟨∀ zs z zs'. w # ws = zs @ z # zs' ∧ obs-intra z S ≠ {} ⟶
    (∃ z''∈set (zs' @ [x]). ∃ nx. call-of-return-node z'' nx ∧ nx ∉ S)⟩
  note impy = ⟨∀ zs z zs'. ys = zs @ z # zs' ∧ obs-intra z S ≠ {} ⟶
    (∃ z''∈set (zs' @ [y]). ∃ ny. call-of-return-node z'' ny ∧ ny ∉ S)⟩
  show ?case
  proof(cases ys)
    case Nil
    with ⟨(w#ws) @ x # xs' = ys @ y # ys'⟩ have y = w and ys' = ws @ x #
xs'
      by simp-all
    from ⟨y = w⟩ ⟨obs-intra y S ≠ {}⟩ impw
    have ∃ z''∈set (ws @ [x]). ∃ nx. call-of-return-node z'' nx ∧ nx ∉ S by blast
    with ⟨ys' = ws @ x # xs'⟩
      ⟨∀ y'∈set ys'. ∃ y''. call-of-return-node y' y'' ∧ y'' ∈ S⟩
    have False by fastforce
    thus ?thesis by simp
```

next
case (*Cons w′ ws′*)
with ‹(*w # ws*) @ *x # xs′* = *ys* @ *y # ys′*› **have** *w* = *w′*
and *ws* @ *x # xs′* = *ws′* @ *y # ys′* **by** *simp-all*
from *impw* **have** *imp1*:∀ *zs z zs′*. *ws* = *zs* @ *z # zs′* ∧ *obs-intra z S* ≠ {} ⟶
(∃ *z′′*∈*set* (*zs′* @ [*x*]). ∃ *nx*. *call-of-return-node z′′ nx* ∧ *nx* ∉ *S*)
by *clarsimp*(*erule-tac x*=*w*#*zs* **in** *allE*,*clarsimp*)
from *Cons impy* **have** *imp2*:∀ *zs z zs′*. *ws′* = *zs* @ *z # zs′* ∧ *obs-intra z S* ≠ {} ⟶
(∃ *z′′*∈*set* (*zs′* @ [*y*]). ∃ *ny*. *call-of-return-node z′′ ny* ∧ *ny* ∉ *S*)
by *clarsimp*(*erule-tac x*=*w′*#*zs* **in** *allE*,*clarsimp*)
from *IH*[*OF* ‹*ws* @ *x # xs′* = *ws′* @ *y # ys′*› ‹*obs-intra x S* ≠ {}›
‹∀ *x′*∈*set xs′*. ∃ *x′′*. *call-of-return-node x′ x′′* ∧ *x′′* ∈ *S*› *imp1*
‹*obs-intra y S* ≠ {}› ‹∀ *y′*∈*set ys′*. ∃ *y′′*. *call-of-return-node y′ y′′* ∧ *y′′* ∈ *S*›
imp2]
have *ws* = *ws′* ∧ *x* = *y* ∧ *xs′* = *ys′* **.**
with ‹*w* = *w′*› *Cons* **show** *?thesis* **by** *simp*
qed
qed

lemma *in-obs-valid*:
assumes *ns′* ∈ *obs ns S* **and** ∀ *n* ∈ *set ns*. *valid-node n*
shows ∀ *n* ∈ *set ns′*. *valid-node n*
using ‹*ns′* ∈ *obs ns S*› ‹∀ *n* ∈ *set ns*. *valid-node n*›
by(*induct ns*)(*auto intro*:*in-obs-intra-valid simp*:*Let-def split*:*split-if-asm*)

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* $\Longrightarrow \exists$ *as*. (-*Entry*-) $-as\rightarrow_{\surd}*$ *n*
and *Exit-path*:*valid-node n* $\Longrightarrow \exists$ *as*. *n* $-as\rightarrow_{\surd}*$ (-*Exit*-)
and *method-exit-unique*:
$[\![$*method-exit n*; *method-exit n'*; *get-proc n* = *get-proc n'*$]\!] \Longrightarrow n = n'$

begin

lemma *get-return-edges-unique*:
assumes *valid-edge a* **and** $a' \in$ *get-return-edges a* **and** $a'' \in$ *get-return-edges a*
shows $a' = a''$
proof −
from ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*›
obtain *Q r p fs* **where** *kind a* = *Q*:$r\hookrightarrow_p fs$
by(*fastforce dest*!:*only-call-get-return-edges*)
with ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*› **obtain** *Q' f'* **where** *kind a'* = $Q'\hookleftarrow_p f'$
by(*fastforce dest*!:*call-return-edges*)
from ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from *this* ‹*kind a'* = $Q'\hookleftarrow_p f'$› **have** *get-proc* (*sourcenode a'*) = *p*
by(*rule get-proc-return*)
from ‹*valid-edge a'*› ‹*kind a'* = $Q'\hookleftarrow_p f'$› **have** *method-exit* (*sourcenode a'*)
by(*fastforce simp*:*method-exit-def*)
from ‹*valid-edge a*› ‹$a'' \in$ *get-return-edges a*› ‹*kind a* = *Q*:$r\hookrightarrow_p fs$›
obtain *Q'' f''* **where** *kind a''* = $Q''\hookleftarrow_p f''$ **by**(*fastforce dest*!:*call-return-edges*)
from ‹*valid-edge a*› ‹$a'' \in$ *get-return-edges a*› **have** *valid-edge a''*
by(*rule get-return-edges-valid*)
from *this* ‹*kind a''* = $Q''\hookleftarrow_p f''$› **have** *get-proc* (*sourcenode a''*) = *p*
by(*rule get-proc-return*)
from ‹*valid-edge a''*› ‹*kind a''* = $Q''\hookleftarrow_p f''$› **have** *method-exit* (*sourcenode a''*)
by(*fastforce simp*:*method-exit-def*)
with ‹*method-exit* (*sourcenode a'*)› ‹*get-proc* (*sourcenode a'*) = *p*›
‹*get-proc* (*sourcenode a''*) = *p*› **have** *sourcenode a'* = *sourcenode a''*
by(*fastforce elim*!:*method-exit-unique*)
from ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*›
obtain *ax'* **where** *valid-edge ax'* **and** *sourcenode ax'* = *sourcenode a*
and *targetnode ax'* = *targetnode a'* **and** *intra-kind*(*kind ax'*)
by −(*drule call-return-node-edge*,*auto simp*:*intra-kind-def*)
from ‹*valid-edge a*› ‹$a'' \in$ *get-return-edges a*›
obtain *ax''* **where** *valid-edge ax''* **and** *sourcenode ax''* = *sourcenode a*
and *targetnode ax''* = *targetnode a''* **and** *intra-kind*(*kind ax''*)
by −(*drule call-return-node-edge*,*auto simp*:*intra-kind-def*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:$r\hookrightarrow_p fs$› ‹*valid-edge ax'*›
‹*sourcenode ax'* = *sourcenode a*› ‹*intra-kind*(*kind ax'*)›
‹*valid-edge ax''*› ‹*sourcenode ax''* = *sourcenode a*› ‹*intra-kind*(*kind ax''*)›
have *ax'* = *ax''* **by** −(*drule call-only-one-intra-edge*,*auto*)
with ‹*targetnode ax'* = *targetnode a'*› ‹*targetnode ax''* = *targetnode a''*›
have *targetnode a'* = *targetnode a''* **by** *simp*

with ‹*valid-edge a'*› ‹*valid-edge a''*› ‹*sourcenode a' = sourcenode a''*›
show *?thesis* **by**(*rule edge-det*)
qed

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(*atomize-elim*)
from ‹*n' postdominates n*› **have** *valid-node n*
and *all*:∀ *as pex*. (*n* −*as*→$_\iota$* *pex* ∧ *method-exit pex*) ⟶ *n'* ∈ *set* (*sourcenodes as*)
by(*auto simp*:*postdominate-def*)
from ‹*valid-node n*› **obtain** *asx* **where** *n* −*asx*→$_{\sqrt{}}$* (*-Exit-*) **by**(*auto dest*:*Exit-path*)
then obtain *as* **where** *n* −*as*→$_{\sqrt{}}$* (*-Exit-*)
and ∀ *a* ∈ *set as*. *intra-kind*(*kind a*) ∨ (∃ *Q f p*. *kind a* = *Q*↩$_p$*f*)
by −(*erule valid-Exit-path-descending-path*)
show ∃ *as*. *n* −*as*→$_\iota$* *n'* ∧ *n'* ∉ *set* (*sourcenodes as*)
proof(*cases* ∃ *a* ∈ *set as*. ∃ *Q f p*. *kind a* = *Q*↩$_p$*f*)
case *True*
then obtain *asx ax asx'* **where** [*simp*]:*as* = *asx*@*ax*#*asx'*
and ∃ *Q f p*. *kind ax* = *Q*↩$_p$*f* **and** ∀ *a* ∈ *set asx*. ∀ *Q f p*. *kind a* ≠ *Q*↩$_p$*f*
by −(*erule split-list-first-propE*,*simp*)
with ‹∀ *a* ∈ *set as*. *intra-kind*(*kind a*) ∨ (∃ *Q f p*. *kind a* = *Q*↩$_p$*f*)›
have ∀ *a* ∈ *set asx*. *intra-kind*(*kind a*) **by** *auto*
from ‹*n* −*as*→$_{\sqrt{}}$* (*-Exit-*)› **have** *n* −*asx*→$_{\sqrt{}}$* *sourcenode ax*
and *valid-edge ax* **by**(*auto dest*:*vp-split*)
from ‹*n* −*asx*→$_{\sqrt{}}$* *sourcenode ax*› ‹∀ *a* ∈ *set asx*. *intra-kind*(*kind a*)›
have *n* −*asx*→$_\iota$* *sourcenode ax* **by**(*simp add*:*vp-def intra-path-def*)
from ‹*valid-edge ax*› ‹∃ *Q f p*. *kind ax* = *Q*↩$_p$*f*›
have *method-exit* (*sourcenode ax*) **by**(*fastforce simp*:*method-exit-def*)
with ‹*n* −*asx*→$_\iota$* *sourcenode ax*› *all* **have** *n'* ∈ *set* (*sourcenodes asx*) **by** *fastforce*
then obtain *xs ys* **where** *sourcenodes asx* = *xs*@*n'*#*ys* **and** *n'* ∉ *set xs*
by(*fastforce dest*:*split-list-first*)
then obtain *as' a as''* **where** *xs* = *sourcenodes as'*
and [*simp*]:*asx* = *as'*@*a*#*as''* **and** *sourcenode a* = *n'*
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from ‹*n* −*asx*→$_\iota$* *sourcenode ax*› **have** *n* −*as'*→$_\iota$* *sourcenode a*
by(*fastforce dest*:*path-split simp*:*intra-path-def*)
with ‹*sourcenode a* = *n'*› ‹*n'* ∉ *set xs*› ‹*xs* = *sourcenodes as'*›
show *?thesis* **by** *fastforce*
next

```
    case False
    with ‹∀ a ∈ set as. intra-kind(kind a) ∨ (∃ Q f p. kind a = Q↩pf)›
    have ∀ a ∈ set as. intra-kind(kind a) by fastforce
    with ‹n −as→√* (-Exit-)› all have n' ∈ set (sourcenodes as)
      by(auto simp:vp-def intra-path-def simp:method-exit-def)
    then obtain xs ys where sourcenodes as = xs@n'#ys and n' ∉ set xs
      by(fastforce dest:split-list-first)
    then obtain as' a as'' where xs = sourcenodes as'
      and [simp]:as = as'@a#as'' and sourcenode a = n'
      by(fastforce elim:map-append-append-maps simp:sourcenodes-def)
   from ‹n −as→√* (-Exit-)› ‹∀ a ∈ set as. intra-kind(kind a)› ‹as = as'@a#as''›
    have n −as'→ι* sourcenode a
      by(fastforce dest:path-split simp:vp-def intra-path-def)
    with ‹sourcenode a = n'› ‹n' ∉ set xs› ‹xs = sourcenodes as'›
    show ?thesis by fastforce
  qed
qed


lemma postdominate-variant:
  assumes n' postdominates n
  shows ∀ as. n −as→√* (-Exit-) ⟶ n' ∈ set (sourcenodes as)
proof −
  from ‹n' postdominates n›
  have all:∀ as pex. (n −as→ι* pex ∧ method-exit pex) ⟶ n' ∈ set (sourcenodes
as)
    by(simp add:postdominate-def)
  { fix as assume n −as→√* (-Exit-)
    then obtain as' pex where n −as'→ι* pex and method-exit pex
      and set(sourcenodes as') ⊆ set(sourcenodes as)
      by(erule valid-Exit-path-intra-path)
    from ‹n −as'→ι* pex› ‹method-exit pex› ‹n' postdominates n›
    have n' ∈ set (sourcenodes as') by(fastforce simp:postdominate-def)
    with ‹set(sourcenodes as') ⊆ set(sourcenodes as)›
    have n' ∈ set (sourcenodes as) by fastforce }
  thus ?thesis by simp
qed


lemma postdominate-refl:
  assumes valid-node n and ¬ method-exit n shows n postdominates n
using ‹valid-node n›
proof(induct rule:valid-node-cases)
  case Entry
  { fix as pex assume (-Entry-) −as→ι* pex and method-exit pex
    from ‹method-exit pex› have (-Entry-) ∈ set (sourcenodes as)
    proof(rule method-exit-cases)
      assume pex = (-Exit-)
      with ‹(-Entry-) −as→ι* pex› have as ≠ []
```

```
      apply(clarsimp simp:intra-path-def) apply(erule path.cases)
      by (drule sym,simp,drule Exit-noteq-Entry,auto)
    with ‹(-Entry-) −as→ι* pex› have hd (sourcenodes as) = (-Entry-)
      by(fastforce intro:path-sourcenode simp:intra-path-def)
    with ‹as ≠ []›show ?thesis by(fastforce intro:hd-in-set simp:sourcenodes-def)
  next
    fix a Q p f assume pex = sourcenode a and valid-edge a and kind a =
Q↩pf
    from ‹(-Entry-) −as→ι* pex› have get-proc (-Entry-) = get-proc pex
      by(rule intra-path-get-procs)
    hence get-proc pex = Main by(simp add:get-proc-Entry)
    from ‹valid-edge a› ‹kind a = Q↩pf› have get-proc (sourcenode a) = p
      by(rule get-proc-return)
    with ‹pex = sourcenode a› ‹get-proc pex = Main› have p = Main by simp
    with ‹valid-edge a› ‹kind a = Q↩pf› have False
      by simp (rule Main-no-return-source)
    thus ?thesis by simp
  qed }
 with Entry show ?thesis
  by(fastforce intro:empty-path simp:postdominate-def intra-path-def)
next
 case Exit
 with ‹¬ method-exit n› have False by(simp add:method-exit-def)
 thus ?thesis by simp
next
 case inner
 show ?thesis
 proof(cases ∃ as. n −as→√* (-Exit-))
  case True
  { fix as pex assume n −as→ι* pex and method-exit pex
    with ‹¬ method-exit n› have as ≠ []
      by(fastforce elim:path.cases simp:intra-path-def)
    with ‹n −as→ι* pex› inner have hd (sourcenodes as) = n
      by(fastforce intro:path-sourcenode simp:intra-path-def)
    from ‹as ≠ []› have sourcenodes as ≠ [] by(simp add:sourcenodes-def)
    with ‹hd (sourcenodes as) = n›[THEN sym]
    have n ∈ set (sourcenodes as) by simp }
  hence ∀ as pex. (n −as→ι* pex ∧ method-exit pex) ⟶ n ∈ set (sourcenodes
as)
    by fastforce
  with True inner show ?thesis
    by(fastforce intro:empty-path
              simp:postdominate-def inner-is-valid intra-path-def)
 next
  case False
  with inner show ?thesis by(fastforce dest:inner-is-valid Exit-path)
 qed
qed
```

lemma *postdominate-trans*:
assumes n'' *postdominates* n **and** n' *postdominates* n''
shows n' *postdominates* n
proof −
from ‹n'' *postdominates* n› ‹n' *postdominates* n''›
have *valid-node* n **and** *valid-node* n' **by**(*simp-all add*:*postdominate-def*)
{ fix *as pex* **assume** $n -as\rightarrow_\iota * pex$ **and** *method-exit pex*
with ‹n'' *postdominates* n› **have** $n'' \in set\ (sourcenodes\ as)$
by(*fastforce simp*:*postdominate-def*)
then obtain $ns'\ ns''$ **where** *sourcenodes as* = $ns'@n''\#ns''$
by(*auto dest*:*split-list*)
then obtain $as'\ as''\ a$ **where** *sourcenodes* $as'' = ns''$ **and** [*simp*]:$as=as'@a\#as''$
and [*simp*]:*sourcenode a* = n''
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from ‹$n -as\rightarrow_\iota * pex$› **have** $n -as'@a\#as''\rightarrow_\iota * pex$ **by** *simp*
hence $n'' -a\#as''\rightarrow_\iota * pex$
by(*fastforce dest*:*path-split-second simp*:*intra-path-def*)
with ‹n' *postdominates* n''› ‹*method-exit pex*›
have $n' \in set(sourcenodes\ (a\#as''))$ **by**(*fastforce simp*:*postdominate-def*)
hence $n' \in set\ (sourcenodes\ as)$ **by**(*fastforce simp*:*sourcenodes-def*) **}**
with ‹*valid-node n*› ‹*valid-node* n'›
show *?thesis* **by**(*fastforce simp*:*postdominate-def*)
qed

lemma *postdominate-antisym*:
assumes n' *postdominates* n **and** n *postdominates* n'
shows $n = n'$
proof −
from ‹n' *postdominates* n› **have** *valid-node n* **and** *valid-node* n'
by(*auto simp*:*postdominate-def*)
from ‹*valid-node n*› **obtain** *asx* **where** $n -asx\rightarrow_{\sqrt{}} * (\text{-}Exit\text{-})$ **by**(*auto dest*:*Exit-path*)
then obtain $as'\ pex$ **where** $n -as'\rightarrow_\iota * pex$ **and** *method-exit pex*
by −(*erule valid-Exit-path-intra-path*)
with ‹n' *postdominates* n› **have** $\exists nx \in set(sourcenodes\ as').\ nx = n'$
by(*fastforce simp*:*postdominate-def*)
then obtain $ns\ ns'$ **where** *sourcenodes* $as' = ns@n'\#ns'$
and $\forall nx \in set\ ns'.\ nx \neq n'$
by(*fastforce elim*!:*split-list-last-propE*)
from ‹*sourcenodes* $as' = ns@n'\#ns'$› **obtain** *asx a* asx'
where [*simp*]:$ns' = sourcenodes\ asx'\ as' = asx@a\#asx'\ sourcenode\ a = n'$
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from ‹$n -as'\rightarrow_\iota * pex$› **have** $n' -a\#asx'\rightarrow_\iota * pex$
by(*fastforce dest*:*path-split-second simp*:*intra-path-def*)
with ‹*n postdominates* n'› ‹*method-exit pex*› **have** $n \in set(sourcenodes\ (a\#asx'))$

by(*fastforce simp*:*postdominate-def*)

```
  hence n = n' ∨ n ∈ set(sourcenodes asx') by(simp add:sourcenodes-def)
  thus ?thesis
  proof
    assume n = n' thus ?thesis .
  next
    assume n ∈ set(sourcenodes asx')
    then obtain nsx' nsx'' where sourcenodes asx' = nsx'@n#nsx''
      by(auto dest:split-list)
    then obtain asi asi' a' where [simp]:asx' = asi@a'#asi' sourcenode a' = n
      by(fastforce elim:map-append-append-maps simp:sourcenodes-def)
    with ‹n -as'→ι* pex› have n -(asx@a#asi)@a'#asi'→ι* pex by simp
    hence n -(asx@a#asi)@a'#asi'→* pex
      and ∀ a ∈ set ((asx@a#asi)@a'#asi'). intra-kind (kind a)
      by(simp-all add:intra-path-def)
    from ‹n -(asx@a#asi)@a'#asi'→* pex›
    have n -a'#asi'→* pex by(fastforce dest:path-split-second)
    with ‹∀ a ∈ set ((asx@a#asi)@a'#asi'). intra-kind (kind a)›
    have n -a'#asi'→ι* pex by(simp add:intra-path-def)
    with ‹n' postdominates n› ‹method-exit pex›
    have n' ∈ set(sourcenodes (a'#asi')) by(fastforce simp:postdominate-def)
    hence n' = n ∨ n' ∈ set(sourcenodes asi')
      by(simp add:sourcenodes-def)
    thus ?thesis
    proof
      assume n' = n thus ?thesis by(rule sym)
    next
      assume n' ∈ set(sourcenodes asi')
     with ‹∀ nx ∈ set ns'. nx ≠ n'› have False by(fastforce simp:sourcenodes-def)
      thus ?thesis by simp
    qed
  qed
qed


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(atomize-elim)
  from assms
  show ∃ as' a as''. as = as'@a#as'' ∧ valid-edge a ∧
    ¬ n' postdominates (sourcenode a) ∧ n' postdominates (targetnode a)
  proof(induct rule:path.induct)
    case (Cons-path n'' as nx a n)
    note IH = ‹⟦n' postdominates nx; ¬ n' postdominates n''⟧
      ⟹ ∃ as' a as''. as = as'@a#as'' ∧ valid-edge a ∧
        ¬ n' postdominates sourcenode a ∧ n' postdominates targetnode a›
    show ?case
    proof(cases n' postdominates n'')
```

```
    case True
    with ‹¬ n' postdominates n› ‹sourcenode a = n› ‹targetnode a = n''›
      ‹valid-edge a› show ?thesis by blast
  next
    case False
    from IH[OF ‹n' postdominates nx› this] show ?thesis
      by clarsimp(rule-tac x=a#as' in exI,clarsimp)
  qed
 qed simp
qed


lemma Exit-no-postdominator:
 assumes (-Exit-) postdominates n shows False
proof −
 from ‹(-Exit-) postdominates n› have valid-node n by(simp add:postdominate-def)
 from ‹valid-node n› obtain asx where n −asx→√* (-Exit-) by(auto dest:Exit-path)
 then obtain as' pex where n −as'→ι* pex and method-exit pex
   by −(erule valid-Exit-path-intra-path)
 with ‹(-Exit-) postdominates n› have (-Exit-) ∈ set (sourcenodes as')
   by(fastforce simp:postdominate-def)
 with ‹n −as'→ι* pex› show False by(fastforce simp:intra-path-def)
qed


lemma postdominate-inner-path-targetnode:
 assumes n' postdominates n and n −as→ι* n'' and n' ∉ set(sourcenodes as)
 shows n' postdominates n''
proof −
 from ‹n' postdominates n› obtain asx
   where valid-node n and valid-node n'
   and all:∀ as pex. (n −as→ι* pex ∧ method-exit pex) ⟶ n' ∈ set (sourcenodes
as)
   by(auto simp:postdominate-def)
 from ‹n −as→ι* n''› have valid-node n''
   by(fastforce dest:path-valid-node simp:intra-path-def)
 have ∀ as' pex'. (n'' −as'→ι* pex' ∧ method-exit pex') ⟶
                  n' ∈ set (sourcenodes as')
 proof(rule ccontr)
   assume ¬ (∀ as' pex'. (n'' −as'→ι* pex' ∧ method-exit pex') ⟶
                         n' ∈ set (sourcenodes as'))
   then obtain as' pex' where n'' −as'→ι* pex' and method-exit pex'
     and n' ∉ set (sourcenodes as') by blast
   from ‹n −as→ι* n''› ‹n'' −as'→ι* pex'› have n −as@as'→ι* pex'
     by(fastforce intro:path-Append simp:intra-path-def)
   from ‹n' ∉ set(sourcenodes as)› ‹n' ∉ set (sourcenodes as')›
   have n' ∉ set (sourcenodes (as@as'))
     by(simp add:sourcenodes-def)
   with ‹n −as@as'→ι* pex'› ‹method-exit pex'› ‹n' postdominates n›
```

```
    show False by(fastforce simp:postdominate-def)
  qed
  with ‹valid-node n'› ‹valid-node n''›
  show ?thesis by(auto simp:postdominate-def)
qed


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(atomize-elim)
  show ∃ ax. sourcenode a = sourcenode ax ∧ valid-edge ax ∧
    ¬ n postdominates targetnode ax
  proof −
    from assms obtain asx pex
      where sourcenode a −asx→ₜ* pex and method-exit pex
      and n ∉ set(sourcenodes asx) by(fastforce simp:postdominate-def)
    show ?thesis
    proof(cases asx)
      case Nil
      with ‹sourcenode a −asx→ₜ* pex› have pex = sourcenode a
        by(fastforce simp:intra-path-def)
      with ‹method-exit pex› have method-exit (sourcenode a) by simp
      thus ?thesis
      proof(rule method-exit-cases)
        assume sourcenode a = (-Exit-)
        with ‹valid-edge a› have False by(rule Exit-source)
        thus ?thesis by simp
      next
        fix a' Q f p assume sourcenode a = sourcenode a'
          and valid-edge a' and kind a' = Q↩pf
        hence False using ‹intra-kind (kind a)› ‹valid-edge a›
          by(fastforce dest:return-edges-only simp:intra-kind-def)
        thus ?thesis by simp
      qed
    next
      case (Cons ax asx')
      with ‹sourcenode a −asx→ₜ* pex›
      have sourcenode a −[]@ax#asx'→* pex
       and ∀ a ∈ set (ax#asx'). intra-kind (kind a) by(simp-all add:intra-path-def)
      from ‹sourcenode a −[]@ax#asx'→* pex›
      have sourcenode a = sourcenode ax and valid-edge ax
        and targetnode ax −asx'→* pex by(fastforce dest:path-split)+
      with ‹∀ a ∈ set (ax#asx'). intra-kind (kind a)›
      have targetnode ax −asx'→ₜ* pex by(simp add:intra-path-def)
      with ‹n ∉ set(sourcenodes asx)› Cons ‹method-exit pex›
      have ¬ n postdominates targetnode ax
```

```
      by(fastforce simp:postdominate-def sourcenodes-def)
    with ‹sourcenode a = sourcenode ax› ‹valid-edge ax› show ?thesis by blast
  qed
 qed
qed


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(atomize-elim)
 from ‹inner-node n› have valid-node n by(rule inner-is-valid)
 then obtain as where n −as→√* (-Exit-) by(fastforce dest:Exit-path)
 show ∃ a. valid-edge a ∧ intra-kind (kind a) ∧ inner-node (sourcenode a) ∧
   targetnode a = (-Exit-)
 proof(cases as = [])
  case True
  with ‹inner-node n› ‹n −as→√* (-Exit-)› have False by(fastforce simp:vp-def)
  thus ?thesis by simp
 next
  case False
  with ‹n −as→√* (-Exit-)› obtain a' as' where as = as'@[a']
   and n −as'→√* sourcenode a' and valid-edge a'
   and (-Exit-) = targetnode a' by −(erule vp-split-snoc)
  from ‹valid-edge a'› have valid-node (sourcenode a') by simp
  thus ?thesis
  proof(cases sourcenode a' rule:valid-node-cases)
   case Entry
   with ‹n −as'→√* sourcenode a'› have n −as'→* (-Entry-) by(simp add:vp-def)
   with ‹inner-node n›
   have False by −(drule path-Entry-target,auto simp:inner-node-def)
   thus ?thesis by simp
  next
   case Exit
   from ‹valid-edge a'› this have False by(rule Exit-source)
   thus ?thesis by simp
  next
   case inner
   have intra-kind (kind a')
   proof(cases kind a' rule:edge-kind-cases)
    case Intra thus ?thesis by simp
   next
    case (Call Q r p fs)
    with ‹valid-edge a'› have get-proc(targetnode a') = p by(rule get-proc-call)
    with ‹(-Exit-) = targetnode a'› get-proc-Exit have p = Main by simp
    with ‹kind a' = Q:r↪pfs› have kind a' = Q:r↪Mainfs by simp
    with ‹valid-edge a'› have False by(rule Main-no-call-target)
    thus ?thesis by simp
```

```
    next
      case (Return Q p f)
       from ‹valid-edge a'› ‹kind a' = Q↩pf› ‹(-Exit-) = targetnode a'›[THEN
sym]
       have False by(rule Exit-no-return-target)
       thus ?thesis by simp
    qed
    with ‹valid-edge a'› ‹(-Exit-) = targetnode a'› ‹inner-node (sourcenode a')›
    show ?thesis by simp blast
  qed
 qed
qed


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(atomize-elim)
 from ‹inner-node n› have valid-node n by(rule inner-is-valid)
 then obtain as where (-Entry-) −as→√* n by(fastforce dest:Entry-path)
 show ∃ a. valid-edge a ∧ intra-kind (kind a) ∧ inner-node (targetnode a) ∧
   sourcenode a = (-Entry-)
 proof(cases as = [])
   case True
   with ‹inner-node n› ‹(-Entry-) −as→√* n› have False
    by(fastforce simp:inner-node-def vp-def)
   thus ?thesis by simp
 next
   case False
   with ‹(-Entry-) −as→√* n› obtain a' as' where as = a'#as'
    and targetnode a' −as'→√* n and valid-edge a'
    and (-Entry-) = sourcenode a' by −(erule vp-split-Cons)
   from ‹valid-edge a'› have valid-node (targetnode a') by simp
   thus ?thesis
   proof(cases targetnode a' rule:valid-node-cases)
    case Entry
    from ‹valid-edge a'› this have False by(rule Entry-target)
    thus ?thesis by simp
   next
    case Exit
   with ‹targetnode a' −as'→√* n› have (-Exit-) −as'→* n by(simp add:vp-def)
    with ‹inner-node n›
    have False by −(drule path-Exit-source,auto simp:inner-node-def)
    thus ?thesis by simp
   next
    case inner
    have intra-kind (kind a')
    proof(cases kind a' rule:edge-kind-cases)
```

```
      case Intro thus ?thesis by simp
    next
      case (Call Q r p fs)
      from ‹valid-edge a'› ‹kind a' = Q:r↪p fs›
        ‹(-Entry-) = sourcenode a'›[THEN sym]
      have False by(rule Entry-no-call-source)
      thus ?thesis by simp
    next
      case (Return Q p f)
      with ‹valid-edge a'› have get-proc(sourcenode a') = p
        by(rule get-proc-return)
      with ‹(-Entry-) = sourcenode a'› get-proc-Entry have p = Main by simp
      with ‹kind a' = Q↩p f› have kind a' = Q↩Main f by simp
      with ‹valid-edge a'› have False by(rule Main-no-return-source)
      thus ?thesis by simp
    qed
    with ‹valid-edge a'› ‹(-Entry-) = sourcenode a'› ‹inner-node (targetnode a')›
    show ?thesis by simp blast
  qed
 qed
qed


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→ι* n'
proof(atomize-elim)
  from ‹valid-node n› obtain as' where n −as'→√* (-Exit-)
    by(fastforce dest:Exit-path)
  then obtain as mex where n −as→ι* mex and method-exit mex
    by(fastforce elim:valid-Exit-path-intra-path)
  from ‹n −as→ι* mex› have get-proc n = get-proc mex
    by(rule intra-path-get-procs)
  with ‹method-exit n'› ‹get-proc n = get-proc n'› ‹method-exit mex›
  have mex = n' by(fastforce intro:method-exit-unique)
  with ‹n −as→ι* mex› show ∃as. n −as→ι* n' by fastforce
qed


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 $\Longrightarrow$ *valid-node* (*parent-node n*)
by(*cases n*) *auto*

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 −
from ‹*valid-SDG-node* (*Formal-in* (*m*,*x*))› **obtain** *a Q r p fs ins outs*
where *valid-edge a* **and** *kind a* = *Q*:*r*↪$_p$*fs* **and** *targetnode a* = *m*
and (*p*,*ins*,*outs*) ∈ *set procs* **and** *x* < *length ins* **by** *fastforce*
from ‹*valid-SDG-node* (*Formal-in* (*m'*,*x'*))› **obtain** *a' Q' r' p' f' ins' outs'*
where *valid-edge a'* **and** *kind a'* = *Q'*:*r'*↪$_{p'}$*f'* **and** *targetnode a'* = *m'*
and (*p'*,*ins'*,*outs'*) ∈ *set procs* **and** *x'* < *length ins'* **by** *fastforce*
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*targetnode a* = *m*›
have *get-proc m* = *p* **by**(*fastforce intro*:*get-proc-call*)
moreover
from ‹*valid-edge a'*› ‹*kind a'* = *Q'*:*r'*↪$_{p'}$*f'*› ‹*targetnode a'* = *m'*›
have *get-proc m'* = *p'* **by**(*fastforce intro*:*get-proc-call*)
ultimately have *p* = *p'* **using** ‹*get-proc m* = *get-proc m'*› **by** *simp*
with ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*valid-edge a'*› ‹*kind a'* = *Q'*:*r'*↪$_{p'}$*f'*›
‹*targetnode a* = *m*› ‹*targetnode a'* = *m'*›
show *?thesis* **by**(*fastforce intro*:*same-proc-call-unique-target*)
qed

lemma *valid-SDG-node-parent-Entry*:
assumes *valid-SDG-node n* **and** *parent-node n* = (*-Entry-*)
shows *n* = *CFG-node* (*-Entry-*)
proof(*cases n*)
case *CFG-node* **with** ‹*parent-node n* = (*-Entry-*)› **show** *?thesis* **by** *simp*
next
case (*Formal-in z*)
with ‹*parent-node n* = (*-Entry-*)› **obtain** *x*
where [*simp*]:*z* = ((*-Entry-*),*x*) **by**(*cases z*) *auto*
with ‹*valid-SDG-node n*› *Formal-in* **obtain** *a* **where** *valid-edge a*
and *targetnode a* = (*-Entry-*) **by** *auto*
hence *False* **by** −(*rule Entry-target*,*simp*+)
thus *?thesis* **by** *simp*
next
case (*Formal-out z*)
with ‹*parent-node n* = (*-Entry-*)› **obtain** *x*
where [*simp*]:*z* = ((*-Entry-*),*x*) **by**(*cases z*) *auto*
with ‹*valid-SDG-node n*› *Formal-out* **obtain** *a Q p f* **where** *valid-edge a*

```
    and kind a = Q↩pf and  sourcenode a = (-Entry-) by auto
  from ‹valid-edge a› ‹kind a = Q↩pf› have get-proc (sourcenode a) = p
    by(rule get-proc-return)
  with ‹sourcenode a = (-Entry-)› have p = Main
    by(auto simp:get-proc-Entry)
  with ‹valid-edge a› ‹kind a = Q↩pf› have False
    by(fastforce intro:Main-no-return-source)
  thus ?thesis by simp
next
  case (Actual-in z)
  with ‹parent-node n = (-Entry-)› obtain x
    where [simp]:z = ((-Entry-),x) by(cases z) auto
  with ‹valid-SDG-node n› Actual-in obtain a Q r p fs where valid-edge a
    and kind a = Q:r↪pfs and sourcenode a = (-Entry-) by fastforce
  hence False by −(rule Entry-no-call-source,auto)
  thus ?thesis by simp
next
  case (Actual-out z)
  with ‹parent-node n = (-Entry-)› obtain x
    where [simp]:z = ((-Entry-),x) by(cases z) auto
  with ‹valid-SDG-node n› Actual-out obtain a where valid-edge a
    targetnode a = (-Entry-) by auto
  hence False by −(rule Entry-target,simp+)
  thus ?thesis by simp
qed


lemma valid-SDG-node-parent-Exit:
  assumes valid-SDG-node n and parent-node n = (-Exit-)
  shows n = CFG-node (-Exit-)
proof(cases n)
  case CFG-node with ‹parent-node n = (-Exit-)› show ?thesis by simp
next
  case (Formal-in z)
  with ‹parent-node n = (-Exit-)› obtain x
    where [simp]:z = ((-Exit-),x) by(cases z) auto
  with ‹valid-SDG-node n› Formal-in obtain a Q r p fs where valid-edge a
    and kind a = Q:r↪pfs and targetnode a = (-Exit-) by fastforce
  from ‹valid-edge a› ‹kind a = Q:r↪pfs› have get-proc (targetnode a) = p
    by(rule get-proc-call)
  with ‹targetnode a = (-Exit-)› have p = Main
    by(auto simp:get-proc-Exit)
  with ‹valid-edge a› ‹kind a = Q:r↪pfs› have False
    by(fastforce intro:Main-no-call-target)
  thus ?thesis by simp
next
  case (Formal-out z)
  with ‹parent-node n = (-Exit-)› obtain x
    where [simp]:z = ((-Exit-),x) by(cases z) auto
```

with ‹*valid-SDG-node n*› *Formal-out* **obtain** *a* **where** *valid-edge a*
and *sourcenode a* = (*-Exit-*) **by** *auto*
hence *False* **by** −(*rule Exit-source*,*simp*+)
thus *?thesis* **by** *simp*
next
case (*Actual-in z*)
with ‹*parent-node n* = (*-Exit-*)› **obtain** *x*
where [*simp*]:*z* = ((*-Exit-*),*x*) **by**(*cases z*) *auto*
with ‹*valid-SDG-node n*› *Actual-in* **obtain** *a* **where** *valid-edge a*
and *sourcenode a* = (*-Exit-*) **by** *auto*
hence *False* **by** −(*rule Exit-source*,*simp*+)
thus *?thesis* **by** *simp*
next
case (*Actual-out z*)
with ‹*parent-node n* = (*-Exit-*)› **obtain** *x*
where [*simp*]:*z* = ((*-Exit-*),*x*) **by**(*cases z*) *auto*
with ‹*valid-SDG-node n*› *Actual-out* **obtain** *a Q p f* **where** *valid-edge a*
and *kind a* = *Q*↩$_{pf}$ **and** *targetnode a* = (*-Exit-*) **by** *auto*
hence *False* **by** −(*erule Exit-no-return-target*,*auto*)
thus *?thesis* **by** *simp*
qed

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*
by(*induct rule*:*SDG-Use.induct*,*auto intro*:*valid-SDG-CFG-node*)

lemma *SDG-Use-parent-Use*:
V ∈ *Use*$_{SDG}$ *n* ⟹ *V* ∈ *Use* (*parent-node n*)
proof(*induct rule*:*SDG-Use.induct*)
case *CFG-Use-SDG-Use* **thus** *?case* **by** *simp*

next
case (*Actual-in-SDG-Use n m x V*)
from ‹*valid-SDG-node n*› ‹*n = Actual-in* (*m*, *x*)› **obtain** *a Q r p fs ins outs*
where *valid-edge a* **and** *kind a = Q*:*r*↪$_p$*fs* **and** *sourcenode a = m*
and (*p*,*ins*,*outs*) ∈ *set procs* **and** *x < length ins* **by** *fastforce*
from ‹*valid-edge a*› ‹*kind a = Q*:*r*↪$_p$*fs*› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
have *length*(*ParamUses* (*sourcenode a*)) = *length ins*
by(*fastforce intro*:*ParamUses-call-source-length*)
with ‹*x < length ins*›
have (*ParamUses* (*sourcenode a*))!*x* ∈ *set* (*ParamUses* (*sourcenode a*)) **by** *simp*
with ‹*V* ∈ (*ParamUses m*)!*x*› ‹*sourcenode a = m*›
have *V* ∈ *Union* (*set* (*ParamUses m*)) **by** *fastforce*
with ‹*valid-edge a*› ‹*sourcenode a = m*› ‹*n = Actual-in* (*m*, *x*)› **show** *?case*
by(*fastforce intro*:*ParamUses-in-Use*)
next
case (*Formal-out-SDG-Use n m x p ins outs V*)
from ‹*valid-SDG-node n*› ‹*n = Formal-out* (*m*, *x*)› **obtain** *a Q p' f ins' outs'*
where *valid-edge a* **and** *kind a = Q*↩$_{p'}$*f* **and** *sourcenode a = m*
and (*p'*,*ins'*,*outs'*) ∈ *set procs* **and** *x < length outs'* **by** *fastforce*
from ‹*valid-edge a*› ‹*kind a = Q*↩$_{p'}$*f*› **have** *get-proc* (*sourcenode a*) = *p'*
by(*rule get-proc-return*)
with ‹*get-proc m = p*› ‹*sourcenode a = m*› **have** [*simp*]:*p = p'* **by** *simp*
with ‹(*p'*,*ins'*,*outs'*) ∈ *set procs*› ‹(*p*,*ins*,*outs*) ∈ *set procs*› *unique-callers*
have [*simp*]:*ins' = ins outs' = outs* **by**(*auto dest*:*distinct-fst-isin-same-fst*)
from ‹*x < length outs'*› ‹*V = outs* ! *x*› **have** *V* ∈ *set outs* **by** *fastforce*
with ‹*valid-edge a*› ‹*kind a = Q*↩$_{p'}$*f*› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
have *V* ∈ *Use* (*sourcenode a*) **by**(*fastforce intro*:*outs-in-Use*)
with ‹*sourcenode a = m*› ‹*valid-SDG-node n*› ‹*n = Formal-out* (*m*, *x*)›
show *?case* **by** *simp*
qed

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

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

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

$V \in Def_{SDG}\ n \Longrightarrow$ *valid-SDG-node n*
by(*induct rule*:*SDG-Def.induct*,*auto intro*:*valid-SDG-CFG-node*)

lemma *SDG-Def-parent-Def*:
$V \in Def_{SDG}\ n \Longrightarrow V \in Def\ (parent\text{-}node\ n)$
proof(*induct rule*:*SDG-Def.induct*)
case *CFG-Def-SDG-Def* **thus** *?case* **by** *simp*
next
case (*Formal-in-SDG-Def n m x p ins outs V*)
from ‹*valid-SDG-node n*› ‹$n = Formal\text{-}in\ (m, x)$› **obtain** *a Q r p' fs ins' outs'*
where *valid-edge a* **and** *kind a* $= Q{:}r{\hookrightarrow}_{p'}fs$ **and** *targetnode a* $= m$
and (*p'*,*ins'*,*outs'*) ∈ *set procs* **and** $x <$ *length ins'* **by** *fastforce*
from ‹*valid-edge a*› ‹*kind a* $= Q{:}r{\hookrightarrow}_{p'}fs$› **have** *get-proc* (*targetnode a*) $= p'$
by(*rule get-proc-call*)
with ‹*get-proc m* $= p$› ‹*targetnode a* $= m$› **have** [*simp*]:$p = p'$ **by** *simp*
with ‹(*p'*,*ins'*,*outs'*) ∈ *set procs*› ‹(*p*,*ins*,*outs*) ∈ *set procs*› *unique-callers*
have [*simp*]:*ins'* = *ins outs'* = *outs* **by**(*auto dest*:*distinct-fst-isin-same-fst*)
from ‹$x <$ *length ins'*› ‹$V =$ *ins* ! *x*› **have** $V \in$ *set ins* **by** *fastforce*
with ‹*valid-edge a*› ‹*kind a* $= Q{:}r{\hookrightarrow}_{p'}fs$› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
have $V \in Def$ (*targetnode a*) **by**(*fastforce intro*:*ins-in-Def*)
with ‹*targetnode a* $= m$› ‹*valid-SDG-node n*› ‹$n = Formal\text{-}in\ (m, x)$›
show *?case* **by** *simp*
next
case (*Actual-out-SDG-Def n m x V*)
from ‹*valid-SDG-node n*› ‹$n = Actual\text{-}out\ (m, x)$› **obtain** *a Q p f ins outs*
where *valid-edge a* **and** *kind a* $= Q{\leftarrow}_{p}f$ **and** *targetnode a* $= m$
and (*p*,*ins*,*outs*) ∈ *set procs* **and** $x <$ *length outs* **by** *fastforce*
from ‹*valid-edge a*› ‹*kind a* $= Q{\leftarrow}_{p}f$› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
have *length*(*ParamDefs* (*targetnode a*)) = *length outs*
by(*rule ParamDefs-return-target-length*)
with ‹$x <$ *length outs*› ‹$V =$ *ParamDefs m* ! *x*› ‹*targetnode a* $= m$›
have $V \in$ *set* (*ParamDefs* (*targetnode a*)) **by**(*fastforce simp*:*set-conv-nth*)
with ‹$n = Actual\text{-}out\ (m, x)$› ‹*targetnode a* $= m$› ‹*valid-edge a*›
show *?case* **by**(*fastforce intro*:*ParamDefs-in-Def*)
qed

definition *data-dependence* :: *'node SDG-node* ⇒ *'var* ⇒ *'node SDG-node* ⇒ *bool*

(- *influences* - *in* - [*51*,*0*,*0*])
where *n influences V in n'* ≡ ∃ *as*. ($V \in Def_{SDG}\ n$) ∧ ($V \in Use_{SDG}\ n'$) ∧
(*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*→$_\iota$* *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*→$_\iota$* *n'* **and** *as* ≠ []
using ‹*n controls n'*›
by(*fastforce simp*:*control-dependence-def*)

lemma *Exit-does-not-control* [*dest*]:
assumes (*-Exit-*) *controls n'* **shows** *False*
proof −
from ‹(*-Exit-*) *controls n'*› **obtain** *a* **where** *valid-edge a*
and *sourcenode a* = (*-Exit-*) **by**(*auto simp*:*control-dependence-def*)
thus *?thesis* **by**(*rule Exit-source*)
qed

lemma *Exit-not-control-dependent*:
assumes *n controls n'* **shows** *n'* ≠ (*-Exit-*)
proof −
from ‹*n controls n'*› **obtain** *a as* **where** *n* −*a*#*as*→$_\iota$* *n'*
and *n' postdominates* (*targetnode a*)
by(*auto simp*:*control-dependence-def*)
from ‹*n* −*a*#*as*→$_\iota$* *n'*› **have** *valid-edge a*
by(*fastforce elim*:*path.cases simp*:*intra-path-def*)
hence *valid-node* (*targetnode a*) **by** *simp*
with ‹*n' postdominates* (*targetnode a*)› ‹*n* −*a*#*as*→$_\iota$* *n'*› **show** *?thesis*
by(*fastforce elim*:*Exit-no-postdominator*)
qed

lemma *which-node-intra-standard-control-dependence-source*:
assumes *nx* −*as*@*a*#*as'*→$_\iota$* *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 −
from ‹*nx* −*as*@*a*#*as'*→$_\iota$* *n*› ‹*sourcenode a* = *n'*› **have** *n'* −*a*#*as'*→$_\iota$* *n*
by(*fastforce dest*:*path-split-second simp*:*intra-path-def*)

```
from ‹nx −as@a#as'→ι* n› have valid-edge a
  by(fastforce intro:path-split simp:intra-path-def)
show ?thesis
proof(cases n postdominates (targetnode a))
  case True
  with ‹n' −a#as'→ι* n› ‹n ∉ set(sourcenodes (a#as'))›
    ‹valid-edge a'› ‹intra-kind(kind a')› ‹sourcenode a' = n'›
    ‹¬ n postdominates (targetnode a')› show ?thesis
    by(fastforce simp:control-dependence-def intra-path-def)
next
  case False
  show ?thesis
  proof(cases as' = [])
    case True
    with ‹n' −a#as'→ι* n› have targetnode a = n
      by(fastforce elim:path.cases simp:intra-path-def)
    with ‹inner-node n› ‹¬ method-exit n› have n postdominates (targetnode a)
      by(fastforce dest:inner-is-valid intro:postdominate-refl)
    with ‹¬ n postdominates (targetnode a)› show ?thesis by simp
  next
    case False
    with ‹nx −as@a#as'→ι* n› have targetnode a −as'→ι* n
      by(fastforce intro:path-split simp:intra-path-def)
   with ‹¬ n postdominates (targetnode a)› ‹valid-edge a› ‹inner-node n›
      ‹targetnode a −as'→ι* n›
    obtain asx pex where targetnode a −asx→ι* pex and method-exit pex
      and n ∉ set (sourcenodes asx)
      by(fastforce dest:inner-is-valid simp:postdominate-def)
    show ?thesis
    proof(cases ∃ asx'. asx = as'@asx')
      case True
      then obtain asx' where [simp]:asx = as'@asx' by blast
      from ‹targetnode a −asx→ι* pex› ‹targetnode a −as'→ι* n›
        ‹as' ≠ []› ‹method-exit pex› ‹¬ method-exit n›
      obtain a'' as'' where asx' = a''#as'' ∧ sourcenode a'' = n
        by(cases asx')(auto dest:path-split path-det simp:intra-path-def)
      hence n ∈ set(sourcenodes asx) by(simp add:sourcenodes-def)
      with ‹n ∉ set (sourcenodes asx)› have False by simp
      thus ?thesis by simp
    next
      case False
      hence ∀ asx'. asx ≠ as'@asx' by simp
      then obtain j asx' where asx = (take j as')@asx'
        and j < length as' and ∀ k > j. ∀ asx''. asx ≠ (take k as')@asx''
        by(auto elim:path-split-general)
      from ‹asx = (take j as')@asx'› ‹j < length as'›
      have ∃ as'1 as'2. asx = as'1@asx' ∧
        as' = as'1@as'2 ∧ as'2 ≠ [] ∧ as'1 = take j as'
        by simp(rule-tac x= drop j as' in exI,simp)
```

then obtain *as'1 as''* **where** *asx = as'1@asx'*
and *as'1 = take j as'*
and *as' = as'1@as''* **and** *as'' ≠ []* **by** *blast*
from ‹*as' = as'1@as''*› ‹*as'' ≠ []*› **obtain** *a1 as'2*
where *as' = as'1@a1#as'2* **and** *as'' = a1#as'2*
by(*cases as''*) *auto*
have *asx' ≠ []*
proof(*cases asx' = []*)
case *True*
with ‹*asx = as'1@asx'*› ‹*as' = as'1@as''*› ‹*as'' = a1#as'2*›
have *as' = asx@a1#as'2* **by** *simp*
with ‹*n' −a#as'→ι∗ n*› **have** *n' −(a#asx)@a1#as'2→ι∗ n* **by** *simp*
hence *n' −(a#asx)@a1#as'2→∗ n*
and ∀ *ax* ∈ *set*((*a#asx*)*@a1#as'2*). *intra-kind*(*kind ax*)
by(*simp-all add*:*intra-path-def*)
from ‹*n' −(a#asx)@a1#as'2→∗ n*›
have *n' −a#asx→∗ sourcenode a1* **and** *valid-edge a1*
by −(*erule path-split*)+
from ‹∀ *ax* ∈ *set*((*a#asx*)*@a1#as'2*). *intra-kind*(*kind ax*)›
have ∀ *ax* ∈ *set*(*a#asx*). *intra-kind*(*kind ax*) **by** *simp*
with ‹*n' −a#asx→∗ sourcenode a1*› **have** *n' −a#asx→ι∗ sourcenode a1*
by(*simp add*:*intra-path-def*)
hence *targetnode a −asx→ι∗ sourcenode a1*
by(*fastforce intro*:*path-split-Cons simp*:*intra-path-def*)
with ‹*targetnode a −asx→ι∗ pex*› **have** *pex = sourcenode a1*
by(*fastforce intro*:*path-det simp*:*intra-path-def*)
from ‹∀ *ax* ∈ *set*((*a#asx*)*@a1#as'2*). *intra-kind*(*kind ax*)›
have *intra-kind* (*kind a1*) **by** *simp*
from ‹*method-exit pex*› **have** *False*
proof(*rule method-exit-cases*)
assume *pex* = (*-Exit-*)
with ‹*pex = sourcenode a1*› **have** *sourcenode a1* = (*-Exit-*) **by** *simp*
with ‹*valid-edge a1*› **show** *False* **by**(*rule Exit-source*)
next
fix *a Q f p* **assume** *pex = sourcenode a* **and** *valid-edge a*
and *kind a = Q↩pf*
from ‹*valid-edge a*› ‹*kind a = Q↩pf*› ‹*pex = sourcenode a*›
‹*pex = sourcenode a1*› ‹*valid-edge a1*› ‹*intra-kind* (*kind a1*)›
show *False* **by**(*fastforce dest*:*return-edges-only simp*:*intra-kind-def*)
qed
thus *?thesis* **by** *simp*
qed *simp*
with ‹*asx = as'1@asx'*› **obtain** *a2 asx'1*
where *asx = as'1@a2#asx'1*
and *asx' = a2#asx'1* **by**(*cases asx'*) *auto*
from ‹*n' −a#as'→ι∗ n*› ‹*as' = as'1@a1#as'2*›
have *n' −(a#as'1)@a1#as'2→ι∗ n* **by** *simp*
hence *n' −(a#as'1)@a1#as'2→∗ n*
and ∀ *ax* ∈ *set*((*a#as'1*)*@a1#as'2*). *intra-kind*(*kind ax*)

by(*simp-all add*: *intra-path-def*)
from ‹$n' -(a\#as'1)@a1\#as'2\to* n$› **have** $n' -a\#as'1\to* sourcenode\ a1$
and *valid-edge a1* **by** $-(erule\ path\text{-}split)+$
from ‹$\forall ax \in set((a\#as'1)@a1\#as'2).\ intra\text{-}kind(kind\ ax)$›
have $\forall ax \in set(a\#as'1).\ intra\text{-}kind(kind\ ax)$ **by** *simp*
with ‹$n' -a\#as'1\to* sourcenode\ a1$› **have** $n' -a\#as'1\to_\iota* sourcenode\ a1$
by(*simp add*:*intra-path-def*)
hence $targetnode\ a -as'1\to_\iota* sourcenode\ a1$
by(*fastforce intro*:*path-split-Cons simp*:*intra-path-def*)
from ‹$targetnode\ a -asx\to_\iota* pex$› ‹$asx = as'1@a2\#asx'1$›
have $targetnode\ a -as'1@a2\#asx'1\to* pex$ **by**(*simp add*:*intra-path-def*)
hence $targetnode\ a -as'1\to* sourcenode\ a2$ **and** *valid-edge a2*
and $targetnode\ a2 -asx'1\to* pex$ **by**(*auto intro*:*path-split*)
from ‹$targetnode\ a2 -asx'1\to* pex$› ‹$asx = as'1@a2\#asx'1$›
‹$targetnode\ a -asx\to_\iota* pex$›
have $targetnode\ a2 -asx'1\to_\iota* pex$ **by**(*simp add*:*intra-path-def*)
from ‹$targetnode\ a -as'1\to* sourcenode\ a2$›
‹$targetnode\ a -as'1\to_\iota* sourcenode\ a1$›
have $sourcenode\ a1 = sourcenode\ a2$
by(*fastforce intro*:*path-det simp*:*intra-path-def*)
from ‹$asx = as'1@a2\#asx'1$› ‹$n \notin set\ (sourcenodes\ asx)$›
have $n \notin set\ (sourcenodes\ asx'1)$ **by**(*simp add*:*sourcenodes-def*)
with ‹$targetnode\ a2 -asx'1\to_\iota* pex$› ‹*method-exit pex*›
‹$asx = as'1@a2\#asx'1$›
have $\neg\ n\ postdominates\ targetnode\ a2$ **by**(*fastforce simp*:*postdominate-def*)
from ‹$asx = as'1@a2\#asx'1$› ‹$targetnode\ a -asx\to_\iota* pex$›
have $intra\text{-}kind\ (kind\ a2)$ **by**(*simp add*:*intra-path-def*)
from ‹$as' = as'1@a1\#as'2$› **have** $a1 \in set\ as'$ **by** *simp*
with ‹$sourcenode\ a1 = sourcenode\ a2$› *last* ‹*valid-edge a2*›
‹$intra\text{-}kind\ (kind\ a2)$›
have $n\ postdominates\ targetnode\ a2$ **by** *blast*
with ‹$\neg\ n\ postdominates\ targetnode\ a2$› **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
qed
qed
qed
qed

1.8.4 SDG without summary edges

inductive $cdep\text{-}edge :: 'node\ SDG\text{-}node \Rightarrow 'node\ SDG\text{-}node \Rightarrow bool$
$(- \longrightarrow_{cd} - [51,0]\ 80)$
and $ddep\text{-}edge :: 'node\ SDG\text{-}node \Rightarrow 'var \Rightarrow 'node\ SDG\text{-}node \Rightarrow bool$
$(- -\text{-}\!\!\rightarrow_{dd} - [51,0,0]\ 80)$
and $call\text{-}edge :: 'node\ SDG\text{-}node \Rightarrow 'pname \Rightarrow 'node\ SDG\text{-}node \Rightarrow bool$
$(- -\text{-}\!\!\rightarrow_{call} - [51,0,0]\ 80)$
and $return\text{-}edge :: 'node\ SDG\text{-}node \Rightarrow 'pname \Rightarrow 'node\ SDG\text{-}node \Rightarrow bool$
$(- -\text{-}\!\!\rightarrow_{ret} - [51,0,0]\ 80)$
and $param\text{-}in\text{-}edge :: 'node\ SDG\text{-}node \Rightarrow 'pname \Rightarrow 'var \Rightarrow 'node\ SDG\text{-}node \Rightarrow$

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

lemma *cdep-edge-cases*:
⟦$n \longrightarrow_{cd} n'$; *(parent-node n) controls (parent-node n')* ⟹ *P*;
⋀*a Q r p fs a'*. ⟦*valid-edge a*; *kind a* = $Q:r\hookrightarrow_p fs$; *a' ∈ get-return-edges a*;
parent-node n = targetnode a; *parent-node n' = sourcenode a'*⟧ ⟹

P;
$\bigwedge m.\ [\![n = \mathit{CFG\text{-}node}\ m;\ m = \mathit{parent\text{-}node}\ n';\ n \neq n']\!] \Longrightarrow P]\!] \Longrightarrow P$
by $-$(*erule SDG-edge.cases*,*auto*,*fastforce*)

lemma *SDG-edge-valid-SDG-node*:
assumes *SDG-edge n Vopt popt n'*
shows *valid-SDG-node n* **and** *valid-SDG-node n'*
using ‹*SDG-edge n Vopt popt n'*›
proof(*induct rule*:*SDG-edge.induct*)
case (*SDG-cdep-edge n m n' m'*)
thus *valid-SDG-node n valid-SDG-node n'*
by(*fastforce elim*:*control-dependence-path elim*:*path-valid-node*
simp:*intra-path-def*)+
next
case (*SDG-proc-entry-exit-cdep a Q r p f n a' n'*) **case** *1*
from ‹*valid-edge a*› ‹*n = CFG-node* (*targetnode a*)› **show** *?case* **by** *simp*
next
case (*SDG-proc-entry-exit-cdep a Q r p f n a' n'*) **case** *2*
from ‹*valid-edge a*› ‹*a' ∈ get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
with ‹*n' = CFG-node* (*sourcenode a'*)› **show** *?case* **by** *simp*
next
case (*SDG-ddep-edge n V n'*)
thus *valid-SDG-node n valid-SDG-node n'*
by(*auto intro*:*in-Use-valid-SDG-node in-Def-valid-SDG-node*
simp:*data-dependence-def*)
qed(*fastforce intro*:*valid-SDG-CFG-node*)+

lemma *valid-SDG-node-cases*:
assumes *valid-SDG-node n*
shows *n = CFG-node* (*parent-node n*) ∨ *CFG-node* (*parent-node n*) $\longrightarrow_{cd}$ *n*
proof(*cases n*)
case (*CFG-node m*) **thus** *?thesis* **by** *simp*
next
case (*Formal-in z*)
from ‹*n = Formal-in z*› **obtain** *m x* **where** *z* = (*m*,*x*) **by**(*cases z*) *auto*
with ‹*valid-SDG-node n*› ‹*n = Formal-in z*› **have** *CFG-node* (*parent-node n*)
$\longrightarrow_{cd}$ *n*
by $-$(*rule SDG-parent-cdep-edge*,*auto*)
thus *?thesis* **by** *fastforce*
next
case (*Formal-out z*)
from ‹*n = Formal-out z*› **obtain** *m x* **where** *z* = (*m*,*x*) **by**(*cases z*) *auto*
with ‹*valid-SDG-node n*› ‹*n = Formal-out z*› **have** *CFG-node* (*parent-node n*)
$\longrightarrow_{cd}$ *n*
by $-$(*rule SDG-parent-cdep-edge*,*auto*)
thus *?thesis* **by** *fastforce*

next
case (*Actual-in z*)
from ‹*n* = *Actual-in z*› **obtain** *m x* **where** *z* = (*m*,*x*) **by**(*cases z*) *auto*
with ‹*valid-SDG-node n*› ‹*n* = *Actual-in z*› **have** *CFG-node* (*parent-node n*) $\longrightarrow_{cd}$ *n*
by −(*rule SDG-parent-cdep-edge*,*auto*)
thus *?thesis* **by** *fastforce*
next
case (*Actual-out z*)
from ‹*n* = *Actual-out z*› **obtain** *m x* **where** *z* = (*m*,*x*) **by**(*cases z*) *auto*
with ‹*valid-SDG-node n*› ‹*n* = *Actual-out z*› **have** *CFG-node* (*parent-node n*) $\longrightarrow_{cd}$ *n*
by −(*rule SDG-parent-cdep-edge*,*auto*)
thus *?thesis* **by** *fastforce*
qed

lemma *SDG-cdep-edge-CFG-node*: $n \longrightarrow_{cd} n' \Longrightarrow \exists\, m.\ n = CFG\text{-}node\ m$
by(*induct n Vopt*≡*None*::*'var option popt*≡*None*::(*'pname* × *bool*) *option n'* *rule*:*SDG-edge.induct*) *auto*

lemma *SDG-call-edge-CFG-node*: $n\ -p\rightarrow_{call}\ n' \Longrightarrow \exists\, m.\ n = CFG\text{-}node\ m$
by(*induct n Vopt*≡*None*::*'var option popt*≡*Some*(*p*,*True*) *n'* *rule*:*SDG-edge.induct*) *auto*

lemma *SDG-return-edge-CFG-node*: $n\ -p\rightarrow_{ret}\ n' \Longrightarrow \exists\, m.\ n = CFG\text{-}node\ m$
by(*induct n Vopt*≡*None*::*'var option popt*≡*Some*(*p*,*False*) *n'* *rule*:*SDG-edge.induct*) *auto*

lemma *SDG-call-or-param-in-edge-unique-CFG-call-edge*:
SDG-edge n Vopt (*Some*(*p*,*True*)) *n'*
$\Longrightarrow \exists!a.$ *valid-edge a* ∧ *sourcenode a* = *parent-node n* ∧
targetnode a = *parent-node n'* ∧ (∃ *Q r fs*. *kind a* = *Q*:*r*↪$_p$*fs*)
proof(*induct n Vopt Some*(*p*,*True*) *n' rule*:*SDG-edge.induct*)
case (*SDG-call-edge a Q r fs n n'*)
{ fix *a'*
assume *valid-edge a'* **and** *sourcenode a'* = *parent-node n*
and *targetnode a'* = *parent-node n'*
from ‹*sourcenode a'* = *parent-node n*› ‹*n* = *CFG-node* (*sourcenode a*)›
have *sourcenode a'* = *sourcenode a* **by** *fastforce*
moreover from ‹*targetnode a'* = *parent-node n'*› ‹*n'* = *CFG-node* (*targetnode a*)›
have *targetnode a'* = *targetnode a* **by** *fastforce*
ultimately have *a'* = *a* **using** ‹*valid-edge a'*› ‹*valid-edge a*›
by(*fastforce intro*:*edge-det*) **}**
with ‹*valid-edge a*› ‹*n* = *CFG-node* (*sourcenode a*)› ‹*n'* = *CFG-node* (*targetnode a*)›

‹*kind* $a = Q{:}r{\hookrightarrow_p}fs$› **show** *?case* **by**(*fastforce intro*!:*ex1I*[*of* - a])
next
case (*SDG-param-in-edge a Q r fs ins outs V x n n'*)
{ fix a'
assume *valid-edge* a' **and** *sourcenode* $a' =$ *parent-node* n
and *targetnode* $a' =$ *parent-node* n'
from ‹*sourcenode* $a' =$ *parent-node* n› ‹$n =$ *Actual-in* (*sourcenode* a,x)›
have *sourcenode* $a' =$ *sourcenode* a **by** *fastforce*
moreover from ‹*targetnode* $a' =$ *parent-node* n'› ‹$n' =$ *Formal-in* (*targetnode* a,x)›
have *targetnode* $a' =$ *targetnode* a **by** *fastforce*
ultimately have $a' = a$ **using** ‹*valid-edge* a'› ‹*valid-edge* a›
by(*fastforce intro*:*edge-det*) **}**
with ‹*valid-edge* a› ‹$n =$ *Actual-in* (*sourcenode* a,x)›
‹$n' =$ *Formal-in* (*targetnode* a,x)› ‹*kind* $a = Q{:}r{\hookrightarrow_p}fs$›
show *?case* **by**(*fastforce intro*!:*ex1I*[*of* - a])
qed *simp-all*

lemma *SDG-return-or-param-out-edge-unique-CFG-return-edge*:
SDG-edge n Vopt (*Some*(*p*,*False*)) n'
$\Longrightarrow \exists !a.$ *valid-edge* $a \wedge$ *sourcenode* $a =$ *parent-node* $n \wedge$
targetnode $a =$ *parent-node* $n' \wedge (\exists\, Q\, f.$ *kind* $a = Q{\hookleftarrow_p}f)$
proof(*induct n Vopt Some*(*p*,*False*) n' *rule*:*SDG-edge.induct*)
case (*SDG-return-edge a Q f n n'*)
{ fix a'
assume *valid-edge* a' **and** *sourcenode* $a' =$ *parent-node* n
and *targetnode* $a' =$ *parent-node* n'
from ‹*sourcenode* $a' =$ *parent-node* n› ‹$n =$ *CFG-node* (*sourcenode* a)›
have *sourcenode* $a' =$ *sourcenode* a **by** *fastforce*
moreover from ‹*targetnode* $a' =$ *parent-node* n'› ‹$n' =$ *CFG-node* (*targetnode* a)›
have *targetnode* $a' =$ *targetnode* a **by** *fastforce*
ultimately have $a' = a$ **using** ‹*valid-edge* a'› ‹*valid-edge* a›
by(*fastforce intro*:*edge-det*) **}**
with ‹*valid-edge* a› ‹$n =$ *CFG-node* (*sourcenode* a)› ‹$n' =$ *CFG-node* (*targetnode* a)›
‹*kind* $a = Q{\hookleftarrow_p}f$› **show** *?case* **by**(*fastforce intro*!:*ex1I*[*of* - a])
next
case (*SDG-param-out-edge a Q f ins outs V x n n'*)
{ fix a'
assume *valid-edge* a' **and** *sourcenode* $a' =$ *parent-node* n
and *targetnode* $a' =$ *parent-node* n'
from ‹*sourcenode* $a' =$ *parent-node* n› ‹$n =$ *Formal-out* (*sourcenode* a,x)›
have *sourcenode* $a' =$ *sourcenode* a **by** *fastforce*
moreover from ‹*targetnode* $a' =$ *parent-node* n'› ‹$n' =$ *Actual-out* (*targetnode* a,x)›
have *targetnode* $a' =$ *targetnode* a **by** *fastforce*
ultimately have $a' = a$ **using** ‹*valid-edge* a'› ‹*valid-edge* a›

by(*fastforce intro*:*edge-det*) **}**
with ‹*valid-edge a*› ‹*n* = *Formal-out* (*sourcenode a*,*x*)›
‹*n'* = *Actual-out* (*targetnode a*,*x*)› ‹*kind a* = *Q*↩$_{pf}$›
show *?case* **by**(*fastforce intro*!:*ex1I*[*of* - *a*])
qed *simp-all*

lemma *Exit-no-SDG-edge-source*:
SDG-edge (*CFG-node* (-*Exit*-)) *Vopt popt n'* ⟹ *False*
proof(*induct CFG-node* (-*Exit*-) *Vopt popt n' rule*:*SDG-edge.induct*)
case (*SDG-cdep-edge m n' m'*)
hence (-*Exit*-) *controls m'* **by** *simp*
thus *?case* **by** *fastforce*
next
case (*SDG-proc-entry-exit-cdep a Q r p fs a' n'*)
from ‹*CFG-node* (-*Exit*-) = *CFG-node* (*targetnode a*)›
have *targetnode a* = (-*Exit*-) **by** *simp*
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_{p}$*fs*› **have** *get-proc* (*targetnode a*) = *p*
by(*rule get-proc-call*)
with ‹*targetnode a* = (-*Exit*-)› **have** *p* = *Main*
by(*auto simp*:*get-proc-Exit*)
with ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_{p}$*fs*› **have** *False*
by(*fastforce intro*:*Main-no-call-target*)
thus *?thesis* **by** *simp*
next
case (*SDG-parent-cdep-edge n' m*)
from ‹*CFG-node* (-*Exit*-) = *CFG-node m*›
have [*simp*]:*m* = (-*Exit*-) **by** *simp*
with ‹*valid-SDG-node n'*› ‹*m* = *parent-node n'*› ‹*CFG-node* (-*Exit*-) ≠ *n'*›
have *False* **by** −(*drule valid-SDG-node-parent-Exit*,*simp*+)
thus *?thesis* **by** *simp*
next
case (*SDG-ddep-edge V n'*)
hence (*CFG-node* (-*Exit*-)) *influences V in n'* **by** *simp*
with *Exit-empty* **show** *?case*
by(*fastforce dest*:*path-Exit-source SDG-Def-parent-Def*
simp:*data-dependence-def intra-path-def*)
next
case (*SDG-call-edge a Q r p fs n'*)
from ‹*CFG-node* (-*Exit*-) = *CFG-node* (*sourcenode a*)›
have *sourcenode a* = (-*Exit*-) **by** *simp*
with ‹*valid-edge a*› **show** *?case* **by**(*rule Exit-source*)
next
case (*SDG-return-edge a Q p f n'*)
from ‹*CFG-node* (-*Exit*-) = *CFG-node* (*sourcenode a*)›
have *sourcenode a* = (-*Exit*-) **by** *simp*
with ‹*valid-edge a*› **show** *?case* **by**(*rule Exit-source*)
qed *simp-all*

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'*
by(*induct rule*:*intra-SDG-path.induct*,
auto intro:*intra-SDG-path.intros simp*:*append-assoc*[*THEN sym*] *simp del*:*append-assoc*)

lemma *intra-SDG-path-valid-SDG-node*:
assumes *n i−ns→$_d$* n'* **shows** *valid-SDG-node n* **and** *valid-SDG-node n'*
using ‹*n i−ns→$_d$* n'*›
by(*induct rule*:*intra-SDG-path.induct*,
auto intro:*SDG-edge-valid-SDG-node valid-SDG-CFG-node*)

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(*atomize-elim*)
from ‹*n i−ns→$_d$* n'*›
show ∃ *as*. *parent-node n −as→$_ι$* parent-node n'*
proof(*induct rule*:*intra-SDG-path.induct*)
case (*iSp-Nil n*)
from ‹*valid-SDG-node n*› **have** *valid-node* (*parent-node n*)
by(*rule valid-SDG-CFG-node*)
hence *parent-node n −[]→* parent-node n* **by**(*rule empty-path*)
thus *?case* **by**(*auto simp*:*intra-path-def*)
next
case (*iSp-Append-cdep n ns n'' n'*)
from ‹∃ *as*. *parent-node n −as→$_ι$* parent-node n''*›
obtain *as* **where** *parent-node n −as→$_ι$* parent-node n''* **by** *blast*
from ‹*n'' ⟶$_{cd}$ n'*› **show** *?case*
proof(*rule cdep-edge-cases*)
assume *parent-node n'' controls parent-node n'*
then obtain *as'* **where** *parent-node n'' −as'→$_ι$* parent-node n'* **and** *as' ≠*
[]

by(*erule control-dependence-path*)
with ‹*parent-node n* $-as\rightarrow_\iota*$ *parent-node n''*›
have *parent-node n* $-as@as'\rightarrow_\iota*$ *parent-node n'* **by** $-$(*rule intra-path-Append*)
thus *?thesis* **by** *blast*
next
fix *a Q r p fs a'*
assume *valid-edge a* **and** *kind a* $= Q{:}r\hookrightarrow_p fs$ **and** $a' \in$ *get-return-edges a*
and *parent-node n''* $=$ *targetnode a* **and** *parent-node n'* $=$ *sourcenode a'*
then obtain *a''* **where** *valid-edge a''* **and** *sourcenode a''* $=$ *targetnode a*
and *targetnode a''* $=$ *sourcenode a'* **and** *kind a''* $= (\lambda cf.\ False)_\surd$
by(*auto dest*:*intra-proc-additional-edge*)
hence *targetnode a* $-[a'']\rightarrow_\iota*$ *sourcenode a'*
by(*fastforce dest*:*path-edge simp*:*intra-path-def intra-kind-def*)
with ‹*parent-node n''* $=$ *targetnode a*› ‹*parent-node n'* $=$ *sourcenode a'*›
have $\exists as'.$ *parent-node n''* $-as'\rightarrow_\iota*$ *parent-node n'* $\wedge$ $as' \neq []$ **by** *fastforce*
then obtain *as'* **where** *parent-node n''* $-as'\rightarrow_\iota*$ *parent-node n'* **and** $as' \neq []$
by *blast*
with ‹*parent-node n* $-as\rightarrow_\iota*$ *parent-node n''*›
have *parent-node n* $-as@as'\rightarrow_\iota*$ *parent-node n'* **by** $-$(*rule intra-path-Append*)
thus *?thesis* **by** *blast*
next
fix *m* **assume** *n''* $=$ *CFG-node m* **and** *m* $=$ *parent-node n'*
with ‹*parent-node n* $-as\rightarrow_\iota*$ *parent-node n''*› **show** *?thesis* **by** *fastforce*
qed
next
case (*iSp-Append-ddep n ns n'' V n'*)
from ‹$\exists as.$ *parent-node n* $-as\rightarrow_\iota*$ *parent-node n''*›
obtain *as* **where** *parent-node n* $-as\rightarrow_\iota*$ *parent-node n''* **by** *blast*
from ‹*n''* $-V\rightarrow_{dd}$ *n'*› **have** *n'' influences V in n'*
by(*fastforce elim*:*SDG-edge.cases*)
then obtain *as'* **where** *parent-node n''* $-as'\rightarrow_\iota*$ *parent-node n'*
by(*auto simp*:*data-dependence-def*)
with ‹*parent-node n* $-as\rightarrow_\iota*$ *parent-node n''*›
have *parent-node n* $-as@as'\rightarrow_\iota*$ *parent-node n'* **by** $-$(*rule intra-path-Append*)
thus *?case* **by** *blast*
qed
qed

1.8.6 Control dependence paths in the SDG

inductive *cdep-SDG-path* ::
'node SDG-node $\Rightarrow$ *'node SDG-node list* $\Rightarrow$ *'node SDG-node* $\Rightarrow$ *bool*
(- *cd*$-\!\!\rightarrow_d*$ - [*51*,*0*,*0*] *80*)

where *cdSp-Nil*:
valid-SDG-node n $\Longrightarrow$ *n cd*$-[]\rightarrow_d*$ *n*

| *cdSp-Append-cdep*:

$[\![n\ cd{-}ns{\to_d}*\ n''; n'' \longrightarrow_{cd} n']\!] \Longrightarrow n\ cd{-}ns@[n'']{\to_d}*\ n'$

lemma *cdep-SDG-path-intra-SDG-path*:
$n\ cd{-}ns{\to_d}*\ n' \Longrightarrow n\ i{-}ns{\to_d}*\ n'$
by(*induct rule*:*cdep-SDG-path.induct*,*auto intro*:*intra-SDG-path.intros*)

lemma *Entry-cdep-SDG-path*:
assumes (*-Entry-*) $-as{\to_\iota}*\ n'$ **and** *inner-node* n' **and** $\neg$ *method-exit* n'
obtains *ns* **where** *CFG-node* (*-Entry-*) $cd{-}ns{\to_d}*$ *CFG-node* n'
and $ns \neq []$ **and** $\forall n'' \in set\ ns.\ parent\text{-}node\ n'' \in set(sourcenodes\ as)$
proof(*atomize-elim*)
from ‹(*-Entry-*) $-as{\to_\iota}*\ n'$› ‹*inner-node* n'› ‹$\neg$ *method-exit* n'›
show $\exists ns.$ *CFG-node* (*-Entry-*) $cd{-}ns{\to_d}*$ *CFG-node* $n' \wedge ns \neq [] \wedge$
$(\forall n'' \in set\ ns.\ parent\text{-}node\ n'' \in set(sourcenodes\ as))$
proof(*induct as arbitrary*:n' *rule*:*length-induct*)
fix *as* n'
assume *IH*:$\forall as'.\ length\ as' < length\ as \longrightarrow$
$(\forall n''.\ (\text{-}Entry\text{-})\ -as'{\to_\iota}*\ n'' \longrightarrow inner\text{-}node\ n'' \longrightarrow \neg\ method\text{-}exit\ n'' \longrightarrow$
$(\exists ns.\ CFG\text{-}node\ (\text{-}Entry\text{-})\ cd{-}ns{\to_d}*\ CFG\text{-}node\ n'' \wedge ns \neq [] \wedge$
$(\forall nx{\in}set\ ns.\ parent\text{-}node\ nx \in set\ (sourcenodes\ as'))))$
and (*-Entry-*) $-as{\to_\iota}*\ n'$ **and** *inner-node* n' **and** $\neg$ *method-exit* n'
thus $\exists ns.$ *CFG-node* (*-Entry-*) $cd{-}ns{\to_d}*$ *CFG-node* $n' \wedge ns \neq [] \wedge$
$(\forall n''{\in}set\ ns.\ parent\text{-}node\ n'' \in set\ (sourcenodes\ as))$
proof −
have $\exists ax\ asx\ zs.$ (*-Entry-*) $-ax\#asx{\to_\iota}*\ n' \wedge n' \notin set\ (sourcenodes\ (ax\#asx))$
$\wedge$
$as = (ax\#asx)@zs$
proof(*cases* $n' \in set\ (sourcenodes\ as)$)
case *True*
hence $\exists n'' \in set(sourcenodes\ as).\ n' = n''$ **by** *simp*
then obtain $ns'\ ns''$ **where** $sourcenodes\ as = ns'@n'\#ns''$
and $\forall n'' \in set\ ns'.\ n' \neq n''$
by(*fastforce elim*!:*split-list-first-propE*)
from ‹$sourcenodes\ as = ns'@n'\#ns''$› **obtain** *xs ys ax*
where $sourcenodes\ xs = ns'$ **and** $as = xs@ax\#ys$
and $sourcenode\ ax = n'$
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from ‹$\forall n'' \in set\ ns'.\ n' \neq n''$› ‹$sourcenodes\ xs = ns'$›
have $n' \notin set(sourcenodes\ xs)$ **by** *fastforce*
from ‹(*-Entry-*) $-as{\to_\iota}*\ n'$› ‹$as = xs@ax\#ys$› **have** (*-Entry-*) $-xs@ax\#ys{\to_\iota}*\ n'$
by *simp*
with ‹$sourcenode\ ax = n'$› **have** (*-Entry-*) $-xs{\to_\iota}*\ n'$
by(*fastforce dest*:*path-split simp*:*intra-path-def*)
with ‹*inner-node* n'› **have** $xs \neq []$
by(*fastforce elim*:*path.cases simp*:*intra-path-def*)
with ‹$n' \notin set(sourcenodes\ xs)$› ‹(*-Entry-*) $-xs{\to_\iota}*\ n'$› ‹$as = xs@ax\#ys$›

show *?thesis* **by**(*cases xs*) *auto*
next
case *False*
with ‹(*-Entry-*) $-as\to_\iota*$ n'› ‹*inner-node* n'›
show *?thesis* **by**(*cases as*)(*auto elim*:*path.cases simp*:*intra-path-def*)
qed
then obtain *ax asx zs* **where** (*-Entry-*) $-ax\#asx\to_\iota*$ n'
and $n' \notin$ *set* (*sourcenodes* (*ax*#*asx*)) **and** *as* = (*ax*#*asx*)@*zs* **by** *blast*
show *?thesis*
proof(*cases* $\forall\, a'\ a''.\ a' \in$ *set asx* $\wedge$ *sourcenode* a' = *sourcenode* a'' $\wedge$
valid-edge a'' $\wedge$ *intra-kind*(*kind* a'') $\longrightarrow$ n' *postdominates targetnode* a'')
case *True*
have (*-Exit-*) $-[]\to_\iota*$ (*-Exit-*)
by(*fastforce intro*:*empty-path simp*:*intra-path-def*)
hence $\neg$ n' *postdominates* (*-Exit-*)
by(*fastforce simp*:*postdominate-def sourcenodes-def method-exit-def*)
from ‹(*-Entry-*) $-ax\#asx\to_\iota*$ n'› **have** (*-Entry-*) $-[]@ax\#asx\to_\iota*$ n' **by** *simp*
from ‹(*-Entry-*) $-ax\#asx\to_\iota*$ n'› **have** [*simp*]:*sourcenode ax* = (*-Entry-*)
and *valid-edge ax*
by(*auto intro*:*path-split-Cons simp*:*intra-path-def*)
from *Entry-Exit-edge* **obtain** a' **where** *sourcenode* a' = (*-Entry-*)
and *targetnode* a' = (*-Exit-*) **and** *valid-edge* a'
and *intra-kind*(*kind* a') **by**(*auto simp*:*intra-kind-def*)
with ‹(*-Entry-*) $-[]@ax\#asx\to_\iota*$ n'› ‹$\neg$ n' *postdominates* (*-Exit-*)›
‹*valid-edge ax*› *True* ‹*sourcenode ax* = (*-Entry-*)›
‹$n' \notin$ *set* (*sourcenodes* (*ax*#*asx*))› ‹*inner-node* n'› ‹$\neg$ *method-exit* n'›
have *sourcenode ax controls* n'
by $-$(*erule which-node-intra-standard-control-dependence-source*
[*of* - - - - - - a'],*auto*)
hence *CFG-node* (*-Entry-*) $\longrightarrow_{cd}$ *CFG-node* n'
by(*fastforce intro*:*SDG-cdep-edge*)
hence *CFG-node* (*-Entry-*) *cd*$-[]@[$*CFG-node* (*-Entry-*)$]\to_d*$ *CFG-node* n'
by(*fastforce intro*:*cdSp-Append-cdep cdSp-Nil*)
moreover
from ‹*as* = (*ax*#*asx*)@*zs*› **have** (*-Entry-*) $\in$ *set*(*sourcenodes as*)
by(*simp add*:*sourcenodes-def*)
ultimately show *?thesis* **by** *fastforce*
next
case *False*
hence $\exists\, a' \in$ *set asx*. $\exists\, a''$. *sourcenode* a' = *sourcenode* a'' $\wedge$ *valid-edge* a'' $\wedge$
intra-kind(*kind* a'') $\wedge$ $\neg$ n' *postdominates targetnode* a''
by *fastforce*
then obtain ax' asx' asx'' **where** $asx = asx'@ax'\#asx'' \wedge$
($\exists\, a''$. *sourcenode* ax' = *sourcenode* a'' $\wedge$ *valid-edge* a'' $\wedge$
intra-kind(*kind* a'') $\wedge$ $\neg$ n' *postdominates targetnode* a'') $\wedge$
($\forall\, z \in$ *set* asx''. $\neg$ ($\exists\, a''$. *sourcenode* z = *sourcenode* a'' $\wedge$ *valid-edge* a'' $\wedge$
intra-kind(*kind* a'') $\wedge$ $\neg$ n' *postdominates targetnode* a''))

by(*blast elim*!:*split-list-last-propE*)
then obtain *ai* **where** *asx* = *asx'*@*ax'*#*asx''*
and *sourcenode ax'* = *sourcenode ai*
and *valid-edge ai* **and** *intra-kind*(*kind ai*)
and ¬ *n'* *postdominates targetnode ai*
and ∀ *z* ∈ *set asx''*. ¬ (∃ *a''*. *sourcenode z* = *sourcenode a''* ∧
valid-edge a'' ∧ *intra-kind*(*kind a''*) ∧ ¬ *n'* *postdominates targetnode a''*)
by *blast*
from ‹(-*Entry*-) −*ax*#*asx*→$_\iota$* *n'*› ‹*asx* = *asx'*@*ax'*#*asx''*›
have (-*Entry*-) −(*ax*#*asx'*)@*ax'*#*asx''*→$_\iota$* *n'* **by** *simp*
from ‹*n'* ∉ *set* (*sourcenodes* (*ax*#*asx*))› ‹*asx* = *asx'*@*ax'*#*asx''*›
have *n'* ∉ *set* (*sourcenodes* (*ax'*#*asx''*))
by(*auto simp*:*sourcenodes-def*)
with ‹*inner-node n'*› ‹¬ *n'* *postdominates targetnode ai*›
‹*n'* ∉ *set* (*sourcenodes* (*ax'*#*asx''*))› ‹*sourcenode ax'* = *sourcenode ai*›
‹∀ *z* ∈ *set asx''*. ¬ (∃ *a''*. *sourcenode z* = *sourcenode a''* ∧
valid-edge a'' ∧ *intra-kind*(*kind a''*) ∧ ¬ *n'* *postdominates targetnode a''*)›
‹*valid-edge ai*› ‹*intra-kind*(*kind ai*)› ‹¬ *method-exit n'*›
‹(-*Entry*-) −(*ax*#*asx'*)@*ax'*#*asx''*→$_\iota$* *n'*›
have *sourcenode ax'* *controls n'*
by(*fastforce intro*!:*which-node-intra-standard-control-dependence-source*)
hence *CFG-node* (*sourcenode ax'*) ⟶$_{cd}$ *CFG-node n'*
by(*fastforce intro*:*SDG-cdep-edge*)
from ‹(-*Entry*-) −(*ax*#*asx'*)@*ax'*#*asx''*→$_\iota$* *n'*›
have (-*Entry*-) −*ax*#*asx'*→$_\iota$* *sourcenode ax'* **and** *valid-edge ax'*
by(*auto intro*:*path-split simp*:*intra-path-def simp del*:*append-Cons*)
from ‹*asx* = *asx'*@*ax'*#*asx''*› ‹*as* = (*ax*#*asx*)@*zs*›
have *length* (*ax*#*asx'*) < *length as* **by** *simp*
from ‹*valid-edge ax'*› **have** *valid-node* (*sourcenode ax'*) **by** *simp*
hence *inner-node* (*sourcenode ax'*)
proof(*cases sourcenode ax' rule*:*valid-node-cases*)
case *Entry*
with ‹(-*Entry*-) −*ax*#*asx'*→$_\iota$* *sourcenode ax'*›
have (-*Entry*-) −*ax*#*asx'*→* (-*Entry*-) **by**(*simp add*:*intra-path-def*)
hence *False* **by**(*fastforce dest*:*path-Entry-target*)
thus *?thesis* **by** *simp*
next
case *Exit*
with ‹*valid-edge ax'*› **have** *False* **by**(*rule Exit-source*)
thus *?thesis* **by** *simp*
qed *simp*
from ‹*asx* = *asx'*@*ax'*#*asx''*› ‹(-*Entry*-) −*ax*#*asx*→$_\iota$* *n'*›
have *intra-kind* (*kind ax'*) **by**(*simp add*:*intra-path-def*)
have ¬ *method-exit* (*sourcenode ax'*)
proof
assume *method-exit* (*sourcenode ax'*)
thus *False*
proof(*rule method-exit-cases*)
assume *sourcenode ax'* = (-*Exit*-)

```
        with ‹valid-edge ax'› show False by(rule Exit-source)
      next
        fix x Q f p assume  sourcenode ax' = sourcenode x
          and valid-edge x and kind x = Q↩pf
        from ‹valid-edge x› ‹kind x = Q↩pf› ‹sourcenode ax' = sourcenode x›
        ‹valid-edge ax'› ‹intra-kind (kind ax')› show False
          by(fastforce dest:return-edges-only simp:intra-kind-def)
      qed
    qed
   with IH ‹length (ax#asx') < length as› ‹(-Entry-) -ax#asx'→ι* sourcenode
ax'›
    ‹inner-node (sourcenode ax')›
    obtain ns where CFG-node (-Entry-) cd−ns→d* CFG-node (sourcenode
ax')
      and ns ≠ []
      and ∀ n'' ∈ set ns. parent-node n'' ∈ set(sourcenodes (ax#asx'))
      by blast
    from ‹CFG-node (-Entry-) cd−ns→d* CFG-node (sourcenode ax')›
      ‹CFG-node (sourcenode ax') ⟶cd CFG-node n'›
   have CFG-node (-Entry-) cd−ns@[CFG-node (sourcenode ax')]→d* CFG-node
n'
      by(fastforce intro:cdSp-Append-cdep)
    from ‹as = (ax#asx)@zs› ‹asx = asx'@ax'#asx''›
    have sourcenode ax' ∈ set(sourcenodes as) by(simp add:sourcenodes-def)
    with ‹∀ n'' ∈ set ns. parent-node n'' ∈ set(sourcenodes (ax#asx'))›
      ‹as = (ax#asx)@zs› ‹asx = asx'@ax'#asx''›
    have ∀ n'' ∈ set (ns@[CFG-node (sourcenode ax')]).
      parent-node n'' ∈ set(sourcenodes as)
      by(fastforce simp:sourcenodes-def)
   with ‹CFG-node (-Entry-) cd−ns@[CFG-node (sourcenode ax')]→d* CFG-node
n'›
    show ?thesis by fastforce
   qed
  qed
 qed
qed


lemma in-proc-cdep-SDG-path:
 assumes n −as→ι* n' and n ≠ n' and n' ≠ (-Exit-) and valid-edge a
 and kind a = Q:r↪pfs 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(atomize-elim)
 show ∃ ns. CFG-node n cd−ns→d* CFG-node n' ∧
        ns ≠ [] ∧ (∀ n''∈set ns. parent-node n'' ∈ set (sourcenodes as))
 proof(cases ∀ ax. valid-edge ax ∧ sourcenode ax = n' ⟶
              ax ∉ get-return-edges a)
  case True
```

from ‹$n\ -as\rightarrow_\iota*\ n'$› ‹$n \neq n'$› ‹$n' \neq$ (*-Exit-*)›
‹$\forall\, ax.$ *valid-edge* $ax \wedge$ *sourcenode* $ax = n' \longrightarrow ax \notin$ *get-return-edges* a›
show $\exists\, ns.$ *CFG-node* n $cd-ns\rightarrow_d*$ *CFG-node* $n' \wedge ns \neq [] \wedge$
$(\forall\, n'' \in$ *set* $ns.$ *parent-node* $n'' \in$ *set*(*sourcenodes* as))
proof(*induct as arbitrary*:n' *rule*:*length-induct*)
fix $as\ n'$
assume *IH*:$\forall\, as'.$ *length* $as' <$ *length* $as \longrightarrow$
$(\forall\, n''.\ n\ -as'\rightarrow_\iota*\ n'' \longrightarrow n \neq n'' \longrightarrow n'' \neq$ (*-Exit-*) $\longrightarrow$
$(\forall\, ax.$ *valid-edge* $ax \wedge$ *sourcenode* $ax = n'' \longrightarrow ax \notin$ *get-return-edges* a)
$\longrightarrow$
$(\exists\, ns.$ *CFG-node* n $cd-ns\rightarrow_d*$ *CFG-node* $n'' \wedge ns \neq [] \wedge$
$(\forall\, n''{\in}$*set* $ns.$ *parent-node* $n'' \in$ *set* (*sourcenodes* as'))))
and $n\ -as\rightarrow_\iota*\ n'$ **and** $n \neq n'$ **and** $n' \neq$ (*-Exit-*)
and $\forall\, ax.$ *valid-edge* $ax \wedge$ *sourcenode* $ax = n' \longrightarrow ax \notin$ *get-return-edges* a
show $\exists\, ns.$ *CFG-node* n $cd-ns\rightarrow_d*$ *CFG-node* $n' \wedge ns \neq [] \wedge$
$(\forall\, n''{\in}$*set* $ns.$ *parent-node* $n'' \in$ *set* (*sourcenodes* as))
proof(*cases method-exit* n')
case *True*
thus *?thesis*
proof(*rule method-exit-cases*)
assume $n' =$ (*-Exit-*)
with ‹$n' \neq$ (*-Exit-*)› **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
next
fix $a'\ Q'\ f'\ p'$
assume $n' =$ *sourcenode* a' **and** *valid-edge* a' **and** *kind* $a' = Q'{\hookleftarrow}_{p'}f'$
from ‹*valid-edge* a› ‹*kind* $a = Q{:}r{\hookrightarrow}_p fs$› **have** *get-proc*(*targetnode* a) $= p$
by(*rule get-proc-call*)
from ‹$n\ -as\rightarrow_\iota*\ n'$› **have** *get-proc* $n =$ *get-proc* n'
by(*rule intra-path-get-procs*)
with ‹*get-proc*(*targetnode* a) $= p$› ‹*targetnode* $a = n$›
have *get-proc* (*targetnode* a) $=$ *get-proc* n' **by** *simp*
from ‹*valid-edge* a'› ‹*kind* $a' = Q'{\hookleftarrow}_{p'}f'$›
have *get-proc* (*sourcenode* a') $= p'$ **by**(*rule get-proc-return*)
with ‹$n' =$ *sourcenode* a'› ‹*get-proc* (*targetnode* a) $=$ *get-proc* n'›
‹*get-proc* (*targetnode* a) $= p$› **have** $p = p'$ **by** *simp*
with ‹*valid-edge* a'› ‹*kind* $a' = Q'{\hookleftarrow}_{p'}f'$›
obtain ax **where** *valid-edge* ax **and** $\exists\, Q\ r\ fs.$ *kind* $ax = Q{:}r{\hookrightarrow}_p fs$
and $a' \in$ *get-return-edges* ax **by**(*auto dest*:*return-needs-call*)
hence *CFG-node* (*targetnode* ax) $\longrightarrow_{cd}$ *CFG-node* (*sourcenode* a')
by(*fastforce intro*:*SDG-proc-entry-exit-cdep*)
with ‹*valid-edge* ax›
have *CFG-node* (*targetnode* ax) $cd-[]@[$*CFG-node* (*targetnode* ax)$]\rightarrow_d*$
CFG-node (*sourcenode* a')
by(*fastforce intro*:*cdep-SDG-path.intros*)
from ‹*valid-edge* a› ‹*kind* $a = Q{:}r{\hookrightarrow}_p fs$› ‹*valid-edge* ax›
‹$\exists\, Q\ r\ fs.$ *kind* $ax = Q{:}r{\hookrightarrow}_p fs$› **have** *targetnode* $a =$ *targetnode* ax
by(*fastforce intro*:*same-proc-call-unique-target*)
from ‹$n\ -as\rightarrow_\iota*\ n'$› ‹$n \neq n'$›

have $as \neq []$ **by**(*fastforce elim*:*path.cases simp*:*intra-path-def*)
with ‹$n\ -as\rightarrow_\iota*\ n'$› **have** $hd\ (sourcenodes\ as) = n$
by(*fastforce intro*:*path-sourcenode simp*:*intra-path-def*)
moreover
from ‹$as \neq []$› **have** $hd\ (sourcenodes\ as) \in set\ (sourcenodes\ as)$
by(*fastforce intro*:*hd-in-set simp*:*sourcenodes-def*)
ultimately have $n \in set\ (sourcenodes\ as)$ **by** *simp*
with ‹$n' = sourcenode\ a'$› ‹$targetnode\ a = targetnode\ ax$›
‹$targetnode\ a = n$›
‹$CFG\text{-}node\ (targetnode\ ax)\ cd-[]@[CFG\text{-}node\ (targetnode\ ax)]\rightarrow_d*$
$CFG\text{-}node\ (sourcenode\ a')$›
show *?thesis* **by** *fastforce*
qed
next
case *False*
from ‹$valid\text{-}edge\ a$› ‹$kind\ a = Q{:}r\hookrightarrow_p fs$› **obtain** a'
where $a' \in get\text{-}return\text{-}edges\ a$
by(*fastforce dest*:*get-return-edge-call*)
with ‹$valid\text{-}edge\ a$› ‹$kind\ a = Q{:}r\hookrightarrow_p fs$› **obtain** $Q'\ f'$ **where** $kind\ a' = Q'\hookleftarrow_p f'$
by(*fastforce dest*!:*call-return-edges*)
with ‹$valid\text{-}edge\ a$› ‹$kind\ a = Q{:}r\hookrightarrow_p fs$› ‹$a' \in get\text{-}return\text{-}edges\ a$› **obtain** a''
where $valid\text{-}edge\ a''$ **and** $sourcenode\ a'' = targetnode\ a$
and $targetnode\ a'' = sourcenode\ a'$ **and** $kind\ a'' = (\lambda cf.\ False)_\surd$
by −(*drule intra-proc-additional-edge*,*auto*)
from ‹$valid\text{-}edge\ a$› ‹$a' \in get\text{-}return\text{-}edges\ a$› **have** $valid\text{-}edge\ a'$
by(*rule get-return-edges-valid*)
have $\exists\, ax\ asx\ zs.\ n\ -ax\#asx\rightarrow_\iota*\ n' \wedge n' \notin set\ (sourcenodes\ (ax\#asx)) \wedge as = (ax\#asx)@zs$
proof(*cases* $n' \in set\ (sourcenodes\ as)$)
case *True*
hence $\exists\, n'' \in set(sourcenodes\ as).\ n' = n''$ **by** *simp*
then obtain $ns'\ ns''$ **where** $sourcenodes\ as = ns'@n'\#ns''$
and $\forall\, n'' \in set\ ns'.\ n' \neq n''$
by(*fastforce elim*!:*split-list-first-propE*)
from ‹$sourcenodes\ as = ns'@n'\#ns''$› **obtain** $xs\ ys\ ax$
where $sourcenodes\ xs = ns'$ **and** $as = xs@ax\#ys$
and $sourcenode\ ax = n'$
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from ‹$\forall\, n'' \in set\ ns'.\ n' \neq n''$› ‹$sourcenodes\ xs = ns'$›
have $n' \notin set(sourcenodes\ xs)$ **by** *fastforce*
from ‹$n\ -as\rightarrow_\iota*\ n'$› ‹$as = xs@ax\#ys$› **have** $n\ -xs@ax\#ys\rightarrow_\iota*\ n'$ **by** *simp*
with ‹$sourcenode\ ax = n'$› **have** $n\ -xs\rightarrow_\iota*\ n'$
by(*fastforce dest*:*path-split simp*:*intra-path-def*)
with ‹$n \neq n'$› **have** $xs \neq []$ **by**(*fastforce simp*:*intra-path-def*)
with ‹$n' \notin set(sourcenodes\ xs)$› ‹$n\ -xs\rightarrow_\iota*\ n'$› ‹$as = xs@ax\#ys$› **show** *?thesis*
by(*cases xs*) *auto*

next
case *False*
with ‹$n\ -as\rightarrow_\iota*\ n'$› ‹$n \neq n'$›
show *?thesis* **by**(*cases as*)(*auto simp*:*intra-path-def*)
qed
then obtain *ax asx zs* **where** $n\ -ax\#asx\rightarrow_\iota*\ n'$
and $n' \notin$ *set* (*sourcenodes* (*ax*#*asx*)) **and** $as = (ax\#asx)@zs$ **by** *blast*
from ‹$n\ -ax\#asx\rightarrow_\iota*\ n'$› ‹$n' \neq$ (*-Exit-*)› **have** *inner-node* n'
by(*fastforce intro*:*path-valid-node simp*:*inner-node-def intra-path-def*)
from ‹*valid-edge a*› ‹*targetnode a* = *n*› **have** *valid-node n* **by** *fastforce*
show *?thesis*
proof(*cases* $\forall\, a'\ a''.\ a' \in$ *set asx* $\wedge$ *sourcenode* a' = *sourcenode* a'' $\wedge$
valid-edge a'' $\wedge$ *intra-kind*(*kind* a'') $\longrightarrow$
n' *postdominates targetnode* a'')
case *True*
from ‹*targetnode a* = *n*› ‹*sourcenode* a'' = *targetnode a*›
‹*kind* a'' = $(\lambda cf.\ False)_{\surd}$›
have *sourcenode* a'' = *n* **and** *intra-kind*(*kind* a'')
by(*auto simp*:*intra-kind-def*)
{ fix *as'* **assume** *targetnode* $a''\ -as'\rightarrow_\iota*\ n'$
from ‹*valid-edge a'*› ‹*targetnode* a'' = *sourcenode* a'›
‹$a' \in$ *get-return-edges a*›
‹$\forall\, ax.$ *valid-edge ax* $\wedge$ *sourcenode ax* = $n' \longrightarrow ax \notin$ *get-return-edges a*›
have *targetnode* $a'' \neq n'$ **by** *fastforce*
with ‹*targetnode* $a''\ -as'\rightarrow_\iota*\ n'$› **obtain** ax' **where** *valid-edge* ax'
and *targetnode* a'' = *sourcenode* ax' **and** *intra-kind*(*kind* ax')
by(*clarsimp simp*:*intra-path-def*)(*erule path.cases*,*fastforce*+)
from ‹*valid-edge a'*› ‹*kind* a' = $Q'\hookleftarrow_p f'$› ‹*valid-edge* ax'›
‹*targetnode* a'' = *sourcenode a'*› ‹*targetnode* a'' = *sourcenode* ax'›
‹*intra-kind*(*kind* ax')›
have *False* **by**(*fastforce dest*:*return-edges-only simp*:*intra-kind-def*) **}**
hence ¬ n' *postdominates targetnode* a''
by(*fastforce elim*:*postdominate-implies-inner-path*)
from ‹$n\ -ax\#asx\rightarrow_\iota*\ n'$› **have** *sourcenode ax* = *n*
by(*auto intro*:*path-split-Cons simp*:*intra-path-def*)
from ‹$n\ -ax\#asx\rightarrow_\iota*\ n'$› **have** $n\ -[]@ax\#asx\rightarrow_\iota*\ n'$ **by** *simp*
from *this* ‹*sourcenode* a'' = *n*› ‹*sourcenode ax* = *n*› *True*
‹$n' \notin$ *set* (*sourcenodes* (*ax*#*asx*))› ‹*valid-edge* a''› ‹*intra-kind*(*kind* a'')›
‹*inner-node* n'› ‹¬ *method-exit* n'› ‹¬ n' *postdominates targetnode* a''›
have *n controls* n'
by(*fastforce intro*!:*which-node-intra-standard-control-dependence-source*)
hence *CFG-node n* $\longrightarrow_{cd}$ *CFG-node* n'
by(*fastforce intro*:*SDG-cdep-edge*)
with ‹*valid-node n*› **have** *CFG-node n cd*$-[]@[$*CFG-node n*$]\rightarrow_d*$ *CFG-node* n'
by(*fastforce intro*:*cdSp-Append-cdep cdSp-Nil*)
moreover
from ‹$as = (ax\#asx)@zs$› ‹*sourcenode ax* = *n*› **have** *n* $\in$ *set*(*sourcenodes as*)

by(*simp add*:*sourcenodes-def*)
ultimately show *?thesis* **by** *fastforce*
next
case *False*
hence $\exists\, a' \in set\ asx.\ \exists\, a''.\ sourcenode\ a' = sourcenode\ a'' \wedge$
$valid\text{-}edge\ a'' \wedge intra\text{-}kind(kind\ a'') \wedge$
$\neg\ n'\ postdominates\ targetnode\ a''$
by *fastforce*
then obtain $ax'\ asx'\ asx''$ **where** $asx = asx'@ax'\#asx'' \wedge$
$(\exists\, a''.\ sourcenode\ ax' = sourcenode\ a'' \wedge valid\text{-}edge\ a'' \wedge$
$intra\text{-}kind(kind\ a'') \wedge \neg\ n'\ postdominates\ targetnode\ a'') \wedge$
$(\forall\, z \in set\ asx''.\ \neg\ (\exists\, a''.\ sourcenode\ z = sourcenode\ a'' \wedge$
$valid\text{-}edge\ a'' \wedge intra\text{-}kind(kind\ a'') \wedge$
$\neg\ n'\ postdominates\ targetnode\ a''))$
by(*blast elim*!:*split-list-last-propE*)
then obtain *ai* **where** $asx = asx'@ax'\#asx''$
and $sourcenode\ ax' = sourcenode\ ai$
and *valid-edge ai* **and** $intra\text{-}kind(kind\ ai)$
and $\neg\ n'\ postdominates\ targetnode\ ai$
and $\forall\, z \in set\ asx''.\ \neg\ (\exists\, a''.\ sourcenode\ z = sourcenode\ a'' \wedge$
$valid\text{-}edge\ a'' \wedge intra\text{-}kind(kind\ a'') \wedge$
$\neg\ n'\ postdominates\ targetnode\ a'')$
by *blast*
from ‹$asx = asx'@ax'\#asx''$› ‹$n\ -ax\#asx\to_\iota *\ n'$›
have $n\ -(ax\#asx')@ax'\#asx''\to_\iota *\ n'$ **by** *simp*
from ‹$n' \notin set\ (sourcenodes\ (ax\#asx))$› ‹$asx = asx'@ax'\#asx''$›
have $n' \notin set\ (sourcenodes\ (ax'\#asx''))$
by(*auto simp*:*sourcenodes-def*)
with ‹*inner-node n'*› ‹$\neg\ n'\ postdominates\ targetnode\ ai$›
‹$n\ -(ax\#asx')@ax'\#asx''\to_\iota *\ n'$› ‹$sourcenode\ ax' = sourcenode\ ai$›
‹$\forall\, z \in set\ asx''.\ \neg\ (\exists\, a''.\ sourcenode\ z = sourcenode\ a'' \wedge$
$valid\text{-}edge\ a'' \wedge intra\text{-}kind(kind\ a'') \wedge$
$\neg\ n'\ postdominates\ targetnode\ a'')$›
‹*valid-edge ai*› ‹$intra\text{-}kind(kind\ ai)$› ‹$\neg$ *method-exit n'*›
have $sourcenode\ ax'\ controls\ n'$
by(*fastforce intro*!:*which-node-intra-standard-control-dependence-source*)
hence $CFG\text{-}node\ (sourcenode\ ax') \longrightarrow_{cd} CFG\text{-}node\ n'$
by(*fastforce intro*:*SDG-cdep-edge*)
from ‹$n\ -(ax\#asx')@ax'\#asx''\to_\iota *\ n'$›
have $n\ -ax\#asx'\to_\iota *\ sourcenode\ ax'$ **and** *valid-edge ax'*
by(*auto intro*:*path-split simp*:*intra-path-def simp del*:*append-Cons*)
from ‹$asx = asx'@ax'\#asx''$› ‹$as = (ax\#asx)@zs$›
have $length\ (ax\#asx') < length\ as$ **by** *simp*
from ‹$as = (ax\#asx)@zs$› ‹$asx = asx'@ax'\#asx''$›
have $sourcenode\ ax' \in set(sourcenodes\ as)$ **by**(*simp add*:*sourcenodes-def*)
show *?thesis*
proof(*cases n = sourcenode ax'*)
case *True*
with ‹$CFG\text{-}node\ (sourcenode\ ax') \longrightarrow_{cd} CFG\text{-}node\ n'$› ‹*valid-edge ax'*›

have *CFG-node n cd−[]@[CFG-node n]→$_d$* CFG-node n′*
by(*fastforce intro*:*cdSp-Append-cdep cdSp-Nil*)
with ‹*sourcenode ax′ ∈ set*(*sourcenodes as*)› *True* **show** *?thesis* **by** *fastforce*
next
case *False*
from ‹*valid-edge ax′*› **have** *sourcenode ax′ ≠* (*-Exit-*)
by −(*rule ccontr*,*fastforce elim*!:*Exit-source*)
from ‹*n −ax#asx′→$_\iota$* sourcenode ax′*› **have** *n = sourcenode ax*
by(*fastforce intro*:*path-split-Cons simp*:*intra-path-def*)
show *?thesis*
proof(*cases* ∀ *ax. valid-edge ax* ∧ *sourcenode ax = sourcenode ax′* ⟶ *ax* ∉ *get-return-edges a*)
case *True*
from ‹*asx = asx′@ax′#asx″*› ‹*n −ax#asx→$_\iota$* n′*›
have *intra-kind* (*kind ax′*) **by**(*simp add*:*intra-path-def*)
have ¬ *method-exit* (*sourcenode ax′*)
proof
assume *method-exit* (*sourcenode ax′*)
thus *False*
proof(*rule method-exit-cases*)
assume *sourcenode ax′ =* (*-Exit-*)
with ‹*valid-edge ax′*› **show** *False* **by**(*rule Exit-source*)
next
fix *x Q f p* **assume** *sourcenode ax′ = sourcenode x*
and *valid-edge x* **and** *kind x = Q↩$_p$f*
from ‹*valid-edge x*› ‹*kind x = Q↩$_p$f*› ‹*sourcenode ax′ = sourcenode x*›
‹*valid-edge ax′*› ‹*intra-kind* (*kind ax′*)› **show** *False*
by(*fastforce dest*:*return-edges-only simp*:*intra-kind-def*)
qed
qed
with *IH* ‹*length* (*ax#asx′*) < *length as*› ‹*n −ax#asx′→$_\iota$* sourcenode ax′*›
‹*n ≠ sourcenode ax′*› ‹*sourcenode ax′ ≠* (*-Exit-*)› *True*
obtain *ns* **where** *CFG-node n cd−ns→$_d$* CFG-node* (*sourcenode ax′*)
and *ns ≠* []
and ∀ *n″∈set ns. parent-node n″ ∈ set* (*sourcenodes* (*ax#asx′*))
by *blast*
from ‹*CFG-node n cd−ns→$_d$* CFG-node* (*sourcenode ax′*)›
‹*CFG-node* (*sourcenode ax′*) ⟶$_{cd}$ *CFG-node n′*›
have *CFG-node n cd−ns@*[*CFG-node* (*sourcenode ax′*)]*→$_d$* CFG-node n′*
by(*rule cdSp-Append-cdep*)
moreover
from ‹∀ *n″∈set ns. parent-node n″ ∈ set* (*sourcenodes* (*ax#asx′*))›
‹*asx = asx′@ax′#asx″*› ‹*as* = (*ax#asx*)*@zs*›
‹*sourcenode ax′ ∈ set*(*sourcenodes as*)›
have ∀ *n″∈set* (*ns@*[*CFG-node* (*sourcenode ax′*)]).

parent-node n'' ∈ *set* (*sourcenodes as*)
by(*fastforce simp*:*sourcenodes-def*)
ultimately show *?thesis* **by** *fastforce*
next
case *False*
then obtain ai' **where** *valid-edge* ai'
and *sourcenode* ai' = *sourcenode* ax'
and ai' ∈ *get-return-edges a* **by** *blast*
with ‹*valid-edge a*› ‹*kind a* = Q:r↪$_p$$fs$› ‹*targetnode a* = n›
have *CFG-node n* ⟶$_{cd}$ *CFG-node* (*sourcenode* ax')
by(*fastforce intro*!:*SDG-proc-entry-exit-cdep*[*of* - - - - - - ai'])
with ‹*valid-node n*›
have *CFG-node n cd*−[]@[*CFG-node n*]→$_d$* *CFG-node* (*sourcenode* ax')
by(*fastforce intro*:*cdSp-Append-cdep cdSp-Nil*)
with ‹*CFG-node* (*sourcenode* ax') ⟶$_{cd}$ *CFG-node* n'›
have *CFG-node n cd*−[*CFG-node n*]@[*CFG-node* (*sourcenode* ax')]→$_d$*
CFG-node n'
by(*fastforce intro*:*cdSp-Append-cdep*)
moreover
from ‹*sourcenode* ax' ∈ *set*(*sourcenodes as*)› ‹n = *sourcenode ax*›
‹*as* = (*ax*#*asx*)@*zs*›
have ∀ n''∈*set* ([*CFG-node n*]@[*CFG-node* (*sourcenode* ax')]).
parent-node n'' ∈ *set* (*sourcenodes as*)
by(*fastforce simp*:*sourcenodes-def*)
ultimately show *?thesis* **by** *fastforce*
qed
qed
qed
qed
qed
next
case *False*
then obtain a' **where** *valid-edge* a' **and** *sourcenode* a' = n'
and a' ∈ *get-return-edges a* **by** *auto*
with ‹*valid-edge a*› ‹*kind a* = Q:r↪$_p$$fs$› ‹*targetnode a* = n›
have *CFG-node n* ⟶$_{cd}$ *CFG-node* n' **by**(*fastforce intro*:*SDG-proc-entry-exit-cdep*)
with ‹*valid-edge a*› ‹*targetnode a* = n›[*THEN sym*]
have *CFG-node n cd*−[]@[*CFG-node n*]→$_d$* *CFG-node* n'
by(*fastforce intro*:*cdep-SDG-path.intros*)
from ‹n −*as*→$_ι$* n'› ‹n ≠ n'› **have** *as* ≠ []
by(*fastforce elim*:*path.cases simp*:*intra-path-def*)
with ‹n −*as*→$_ι$* n'› **have** *hd* (*sourcenodes as*) = n
by(*fastforce intro*:*path-sourcenode simp*:*intra-path-def*)
with ‹*as* ≠ []› **have** n ∈ *set* (*sourcenodes as*)
by(*fastforce intro*:*hd-in-set simp*:*sourcenodes-def*)
with ‹*CFG-node n cd*−[]@[*CFG-node n*]→$_d$* *CFG-node* n'›
show *?thesis* **by** *auto*

qed
qed

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'*
by(*induct rule*:*call-cdep-SDG-path.induct*,
   auto intro:*call-cdep-SDG-path.intros simp*:*append-assoc*[*THEN sym*]
                                simp del:*append-assoc*)

lemma *cdep-SDG-path-cc-SDG-path*:
  n cd−ns→$_d$ n' ⟹ n cc−ns→$_d$* n'*
by(*induct rule*:*cdep-SDG-path.induct*,*auto intro*:*call-cdep-SDG-path.intros*)

lemma *Entry-cc-SDG-path-to-inner-node*:
  assumes *valid-SDG-node n* **and** *parent-node n ≠* (*-Exit-*)
  obtains *ns* **where** *CFG-node* (*-Entry-*) *cc−ns→$_d$* n*
proof(*atomize-elim*)
  obtain *m* **where** *m = parent-node n* **by** *simp*
  from ‹*valid-SDG-node n*› **have** *valid-node* (*parent-node n*)
    by(*rule valid-SDG-CFG-node*)
  thus ∃ *ns*. *CFG-node* (*-Entry-*) *cc−ns→$_d$* n*
  proof(*cases parent-node n rule*:*valid-node-cases*)
    case *Entry*
    with ‹*valid-SDG-node n*› **have** *n = CFG-node* (*-Entry-*)
      by(*rule valid-SDG-node-parent-Entry*)
    with ‹*valid-SDG-node n*› **show** *?thesis* **by**(*fastforce intro*:*ccSp-Nil*)
  next
    case *Exit*
    with ‹*parent-node n ≠* (*-Exit-*)› **have** *False* **by** *simp*
    thus *?thesis* **by** *simp*
  next
    case *inner*

with ‹m = *parent-node* n› **obtain** *asx* **where** (*-Entry-*) $-asx\rightarrow_{\surd}* m$
by(*fastforce dest*:*Entry-path inner-is-valid*)
then obtain *as* **where** (*-Entry-*) $-as\rightarrow_{\surd}* m$
and $\forall a' \in set\ as.$ *intra-kind*(*kind* a') $\vee$ $(\exists Q\ r\ p\ fs.$ *kind* $a' = Q$:$r\hookrightarrow_p fs)$
by $-$(*erule valid-Entry-path-ascending-path*,*fastforce*)
from ‹*inner-node* (*parent-node* n)› ‹m = *parent-node* n›
have *inner-node* m **by** *simp*
with ‹(*-Entry-*) $-as\rightarrow_{\surd}* m$› ‹$m$ = *parent-node* n› ‹*valid-SDG-node* n›
‹$\forall a' \in set\ as.$ *intra-kind*(*kind* a') $\vee$ $(\exists Q\ r\ p\ fs.$ *kind* $a' = Q$:$r\hookrightarrow_p fs)$›
show *?thesis*
proof(*induct as arbitrary*:*m n rule*:*length-induct*)
fix *as m n*
assume *IH*:$\forall as'.$ *length* $as' <$ *length as* $\longrightarrow$
$(\forall m'.$ (*-Entry-*) $-as'\rightarrow_{\surd}* m' \longrightarrow$
$(\forall n'.\ m' =$ *parent-node* $n' \longrightarrow$ *valid-SDG-node* $n' \longrightarrow$
$(\forall a' \in set\ as'.$ *intra-kind*(*kind* a') $\vee$ $(\exists Q\ r\ p\ fs.$ *kind* $a' = Q$:$r\hookrightarrow_p fs)) \longrightarrow$
inner-node $m' \longrightarrow (\exists ns.$ *CFG-node* (*-Entry-*) $cc-ns\rightarrow_d* n'))$
and (*-Entry-*) $-as\rightarrow_{\surd}* m$
and $m =$ *parent-node* n **and** *valid-SDG-node* n **and** *inner-node* m
and $\forall a' \in set\ as.$ *intra-kind*(*kind* a') $\vee$ $(\exists Q\ r\ p\ fs.$ *kind* $a' = Q$:$r\hookrightarrow_p fs)$
show $\exists ns.$ *CFG-node* (*-Entry-*) $cc-ns\rightarrow_d* n$
proof(*cases* $\forall a' \in set\ as.$ *intra-kind*(*kind* a'))
case *True*
with ‹(*-Entry-*) $-as\rightarrow_{\surd}* m$› **have** (*-Entry-*) $-as\rightarrow_{\iota}* m$
by(*fastforce simp*:*intra-path-def vp-def*)
have $\neg$ *method-exit* m
proof
assume *method-exit* m
thus *False*
proof(*rule method-exit-cases*)
assume m = (*-Exit-*)
with ‹*inner-node* m› **show** *False* **by**(*simp add*:*inner-node-def*)
next
fix $a\ Q\ f\ p$ **assume** $m =$ *sourcenode* a **and** *valid-edge* a
and *kind* $a = Q\hookleftarrow_p f$
from ‹(*-Entry-*) $-as\rightarrow_{\iota}* m$› **have** *get-proc* m = *Main*
by(*fastforce dest*:*intra-path-get-procs simp*:*get-proc-Entry*)
from ‹*valid-edge* a› ‹*kind* $a = Q\hookleftarrow_p f$›
have *get-proc* (*sourcenode* a) = p **by**(*rule get-proc-return*)
with ‹*get-proc* m = *Main*› ‹m = *sourcenode* a› **have** p = *Main* **by** *simp*
with ‹*valid-edge* a› ‹*kind* $a = Q\hookleftarrow_p f$› **show** *False*
by(*fastforce intro*:*Main-no-return-source*)
qed
qed
with ‹*inner-node* m› ‹(*-Entry-*) $-as\rightarrow_{\iota}* m$›
obtain *ns* **where** *CFG-node* (*-Entry-*) $cd-ns\rightarrow_d*$ *CFG-node* m
and $ns \neq []$ **and** $\forall n'' \in set\ ns.$ *parent-node* $n'' \in set$(*sourcenodes as*)
by $-$(*erule Entry-cdep-SDG-path*)
then obtain n' **where** $n' \longrightarrow_{cd}$ *CFG-node* m

and *parent-node* $n' \in$ *set*(*sourcenodes as*)
by −(*erule cdep-SDG-path.cases*,*auto*)
from ‹*parent-node* $n' \in$ *set*(*sourcenodes as*)› **obtain** *ms ms'*
where *sourcenodes as* = *ms*@(*parent-node n'*)#*ms'*
by(*fastforce dest*:*split-list simp*:*sourcenodes-def*)
then obtain *as' a as''* **where** *ms* = *sourcenodes as'*
and *ms'* = *sourcenodes as''* **and** *as* = *as'*@*a*#*as''*
and *parent-node n'* = *sourcenode a*
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
with ‹(*-Entry-*) −*as*→$_\iota$* *m*› **have** (*-Entry-*) −*as'*→$_\iota$* *parent-node n'*
by(*fastforce intro*:*path-split simp*:*intra-path-def*)
from ‹*n'* ⟶$_{cd}$ *CFG-node m*› **have** *valid-SDG-node n'*
by(*rule SDG-edge-valid-SDG-node*)
hence *n'-cases*:
n' = *CFG-node* (*parent-node n'*) ∨ *CFG-node* (*parent-node n'*) ⟶$_{cd}$ *n'*
by(*rule valid-SDG-node-cases*)
show *?thesis*
proof(*cases as'* = [])
case *True*
with ‹(*-Entry-*) −*as'*→$_\iota$* *parent-node n'*› **have** *parent-node n'* = (*-Entry-*)
by(*fastforce simp*:*intra-path-def*)
from *n'-cases* **have** ∃ *ns*. *CFG-node* (*-Entry-*) *cd*−*ns*→$_d$* *CFG-node m*
proof
assume *n'* = *CFG-node* (*parent-node n'*)
with ‹*n'* ⟶$_{cd}$ *CFG-node m*› ‹*parent-node n'* = (*-Entry-*)›
have *CFG-node* (*-Entry-*) *cd*−[]@[*CFG-node* (*-Entry-*)]→$_d$* *CFG-node m*
by −(*rule cdSp-Append-cdep*,*rule cdSp-Nil*,*auto*)
thus *?thesis* **by** *fastforce*
next
assume *CFG-node* (*parent-node n'*) ⟶$_{cd}$ *n'*
with ‹*parent-node n'* = (*-Entry-*)›
have *CFG-node* (*-Entry-*) *cd*−[]@[*CFG-node* (*-Entry-*)]→$_d$* *n'*
by −(*rule cdSp-Append-cdep*,*rule cdSp-Nil*,*auto*)
with ‹*n'* ⟶$_{cd}$ *CFG-node m*›
have *CFG-node* (*-Entry-*) *cd*−[*CFG-node* (*-Entry-*)]@[*n'*]→$_d$* *CFG-node m*
by(*fastforce intro*:*cdSp-Append-cdep*)
thus *?thesis* **by** *fastforce*
qed
then obtain *ns* **where** *CFG-node* (*-Entry-*) *cc*−*ns*→$_d$* *CFG-node m*
by(*fastforce intro*:*cdep-SDG-path-cc-SDG-path*)
show *?thesis*
proof(*cases n* = *CFG-node m*)
case *True*
with ‹*CFG-node* (*-Entry-*) *cc*−*ns*→$_d$* *CFG-node m*›
show *?thesis* **by** *fastforce*
next
case *False*

with ‹*inner-node m*› ‹*valid-SDG-node n*› ‹*m = parent-node n*›
have *CFG-node m* $\longrightarrow_{cd}$ *n*
by(*fastforce intro*:*SDG-parent-cdep-edge inner-is-valid*)
with ‹*CFG-node* (*-Entry-*) *cc*−*ns*→$_d$* *CFG-node m*›
have *CFG-node* (*-Entry-*) *cc*−*ns*@[*CFG-node m*]→$_d$* *n*
by(*fastforce intro*:*ccSp-Append-cdep*)
thus *?thesis* **by** *fastforce*
qed
next
case *False*
with ‹*as = as'*@*a*#*as''*› **have** *length as'* < *length as* **by** *simp*
from ‹(*-Entry-*) −*as'*→$_\iota$* *parent-node n'*› **have** *valid-node* (*parent-node n'*)
by(*fastforce intro*:*path-valid-node simp*:*intra-path-def*)
hence *inner-node* (*parent-node n'*)
proof(*cases parent-node n' rule*:*valid-node-cases*)
case *Entry*
with ‹(*-Entry-*) −*as'*→$_\iota$* (*parent-node n'*)›
have (*-Entry-*) −*as'*→* (*-Entry-*) **by**(*fastforce simp*:*intra-path-def*)
with *False* **have** *False* **by** *fastforce*
thus *?thesis* **by** *simp*
next
case *Exit*
with ‹*n'* $\longrightarrow_{cd}$ *CFG-node m*› **have** *n' = CFG-node* (*-Exit-*)
by −(*rule valid-SDG-node-parent-Exit*,*erule SDG-edge-valid-SDG-node*,*simp*)
with ‹*n'* $\longrightarrow_{cd}$ *CFG-node m*› *Exit* **have** *False*
by *simp*(*erule Exit-no-SDG-edge-source*)
thus *?thesis* **by** *simp*
next
case *inner*
thus *?thesis* **by** *simp*
qed
from ‹*valid-node* (*parent-node n'*)›
have *valid-SDG-node* (*CFG-node* (*parent-node n'*)) **by** *simp*
from ‹(*-Entry-*) −*as'*→$_\iota$* (*parent-node n'*)›
have (*-Entry-*) −*as'*→$_\surd$* (*parent-node n'*)
by(*rule intra-path-vp*)
from ‹∀ *a'* ∈ *set as*. *intra-kind*(*kind a'*) ∨ (∃ *Q r p fs*. *kind a'* = *Q*:*r*↪$_p$*fs*)›
‹*as = as'*@*a*#*as''*›
have ∀ *a'* ∈ *set as'*. *intra-kind*(*kind a'*) ∨ (∃ *Q r p fs*. *kind a'* = *Q*:*r*↪$_p$*fs*)
by *auto*
with *IH* ‹*length as'* < *length as*› ‹(*-Entry-*) −*as'*→$_\surd$* (*parent-node n'*)›
‹*valid-SDG-node* (*CFG-node* (*parent-node n'*))› ‹*inner-node* (*parent-node n'*)›
obtain *ns* **where** *CFG-node* (*-Entry-*) *cc*−*ns*→$_d$* *CFG-node* (*parent-node n'*)
apply(*erule-tac x*=*as'* **in** *allE*) **apply** *clarsimp*
apply(*erule-tac x*=(*parent-node n'*) **in** *allE*) **apply** *clarsimp*
apply(*erule-tac x*=*CFG-node* (*parent-node n'*) **in** *allE*) **by** *clarsimp*
from *n'-cases* **have** ∃ *ns*. *CFG-node* (*-Entry-*) *cc*−*ns*→$_d$* *n'*

```
    proof
      assume n' = CFG-node (parent-node n')
      with ‹CFG-node (-Entry-) cc−ns→d* CFG-node (parent-node n')›
      show ?thesis by fastforce
    next
      assume CFG-node (parent-node n') ⟶cd n'
      with ‹CFG-node (-Entry-) cc−ns→d* CFG-node (parent-node n')›
      have CFG-node (-Entry-) cc−ns@[CFG-node (parent-node n')]→d* n'
        by(fastforce intro:ccSp-Append-cdep)
      thus ?thesis by fastforce
    qed
    then obtain ns' where CFG-node (-Entry-) cc−ns'→d* n' by blast
    with ‹n' ⟶cd CFG-node m›
    have CFG-node (-Entry-) cc−ns'@[n']→d* CFG-node m
      by(fastforce intro:ccSp-Append-cdep)
    show ?thesis
    proof(cases n = CFG-node m)
      case True
      with ‹CFG-node (-Entry-) cc−ns'@[n']→d* CFG-node m›
      show ?thesis by fastforce
    next
      case False
      with ‹inner-node m› ‹valid-SDG-node n› ‹m = parent-node n›
      have CFG-node m ⟶cd n
        by(fastforce intro:SDG-parent-cdep-edge inner-is-valid)
      with ‹CFG-node (-Entry-) cc−ns'@[n']→d* CFG-node m›
      have CFG-node (-Entry-) cc−(ns'@[n'])@[CFG-node m]→d* n
        by(fastforce intro:ccSp-Append-cdep)
      thus ?thesis by fastforce
    qed
  qed
next
  case False
  hence ∃ a' ∈ set as. ¬ intra-kind (kind a') by fastforce
  then obtain a as' as'' where as = as'@a#as'' and ¬ intra-kind (kind a)
    and ∀ a' ∈ set as''. intra-kind (kind a')
    by(fastforce elim!:split-list-last-propE)
  from ‹∀ a' ∈ set as. intra-kind(kind a') ∨ (∃ Q r p fs. kind a' = Q:r↪pfs)›
    ‹as = as'@a#as''› ‹¬ intra-kind (kind a)›
  obtain Q r p fs where kind a = Q:r↪pfs
    and ∀ a' ∈ set as'. intra-kind(kind a') ∨ (∃ Q r p fs. kind a' = Q:r↪pfs)
    by auto
  from ‹as = as'@a#as''› have length as' < length as by fastforce
  from ‹(-Entry-) −as→√* m› ‹as = as'@a#as''›
  have (-Entry-) −as'→√* sourcenode a and valid-edge a
    and targetnode a −as''→√* m
    by(auto intro:vp-split)
  hence valid-SDG-node (CFG-node (sourcenode a)) by simp
  have ∃ ns'. CFG-node (-Entry-) cc−ns'→d* CFG-node m
```

proof(*cases targetnode a = m*)
case *True*
with ‹*valid-edge a*› ‹*kind a = Q:r↪pfs*›
have *CFG-node* (*sourcenode a*) $-p\rightarrow_{call}$ *CFG-node m*
by(*fastforce intro*:*SDG-call-edge*)
have ∃ *ns*. *CFG-node* (*-Entry-*) *cc*$-ns\rightarrow_{d}*$ *CFG-node* (*sourcenode a*)
proof(*cases as′ = []*)
case *True*
with ‹(*-Entry-*) $-as'\rightarrow_{\surd}*$ *sourcenode a*› **have** (*-Entry-*) = *sourcenode a*
by(*fastforce simp*:*vp-def*)
with ‹*CFG-node* (*sourcenode a*) $-p\rightarrow_{call}$ *CFG-node m*›
have *CFG-node* (*-Entry-*) *cc*$-[]\rightarrow_{d}*$ *CFG-node* (*sourcenode a*)
by(*fastforce intro*:*ccSp-Nil SDG-edge-valid-SDG-node*)
thus *?thesis* **by** *fastforce*
next
case *False*
from ‹*valid-edge a*› **have** *valid-node* (*sourcenode a*) **by** *simp*
hence *inner-node* (*sourcenode a*)
proof(*cases sourcenode a rule*:*valid-node-cases*)
case *Entry*
with ‹(*-Entry-*) $-as'\rightarrow_{\surd}*$ *sourcenode a*›
have (*-Entry-*) $-as'\rightarrow*$ (*-Entry-*) **by**(*fastforce simp*:*vp-def*)
with *False* **have** *False* **by** *fastforce*
thus *?thesis* **by** *simp*
next
case *Exit*
with ‹*valid-edge a*› **have** *False* **by** −(*erule Exit-source*)
thus *?thesis* **by** *simp*
next
case *inner*
thus *?thesis* **by** *simp*
qed
with *IH* ‹*length as′ < length as*› ‹(*-Entry-*) $-as'\rightarrow_{\surd}*$ *sourcenode a*›
‹*valid-SDG-node* (*CFG-node* (*sourcenode a*))›
‹∀ *a′* ∈ *set as′*. *intra-kind*(*kind a′*) ∨ (∃ *Q r p fs*. *kind a′ = Q:r↪pfs*)›
obtain *ns* **where** *CFG-node* (*-Entry-*) *cc*$-ns\rightarrow_{d}*$ *CFG-node* (*sourcenode a*)
apply(*erule-tac x*=*as′* **in** *allE*) **apply** *clarsimp*
apply(*erule-tac x*=*sourcenode a* **in** *allE*) **apply** *clarsimp*
apply(*erule-tac x*=*CFG-node* (*sourcenode a*) **in** *allE*) **by** *clarsimp*
thus *?thesis* **by** *fastforce*
qed
then obtain *ns* **where** *CFG-node* (*-Entry-*) *cc*$-ns\rightarrow_{d}*$ *CFG-node* (*sourcenode a*)
by *blast*
with ‹*CFG-node* (*sourcenode a*) $-p\rightarrow_{call}$ *CFG-node m*›
show *?thesis* **by**(*fastforce intro*:*ccSp-Append-call*)
next
case *False*

from ‹*targetnode a* $-as''\rightarrow_{\surd}*$ *m*› ‹$\forall\, a' \in$ *set as''*. *intra-kind* (*kind a'*)›
have *targetnode a* $-as''\rightarrow_{\iota}*$ *m* **by**(*fastforce simp*:*vp-def intra-path-def*)
hence *get-proc* (*targetnode a*) = *get-proc m* **by**(*rule intra-path-get-procs*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*$\hookrightarrow_p$*fs*› **have** *get-proc* (*targetnode a*) = *p*
by(*rule get-proc-call*)
from ‹*inner-node m*› ‹*valid-edge a*› ‹*targetnode a* $-as''\rightarrow_{\iota}*$ *m*›
‹*kind a* = *Q*:*r*$\hookrightarrow_p$*fs*› ‹*targetnode a* ≠ *m*›
obtain *ns* **where** *CFG-node* (*targetnode a*) *cd*$-ns\rightarrow_{d}*$ *CFG-node m*
and *ns* ≠ []
and $\forall\, n'' \in$ *set ns*. *parent-node n''* ∈ *set*(*sourcenodes as''*)
by(*fastforce elim*!:*in-proc-cdep-SDG-path*)
then obtain *n'* **where** *n'* $\longrightarrow_{cd}$ *CFG-node m*
and *parent-node n'* ∈ *set*(*sourcenodes as''*)
by −(*erule cdep-SDG-path.cases*,*auto*)
from ‹(*parent-node n'*) ∈ *set*(*sourcenodes as''*)› **obtain** *ms ms'*
where *sourcenodes as''* = *ms*@(*parent-node n'*)#*ms'*
by(*fastforce dest*:*split-list simp*:*sourcenodes-def*)
then obtain *xs a' ys* **where** *ms* = *sourcenodes xs*
and *ms'* = *sourcenodes ys* **and** *as''* = *xs*@*a'*#*ys*
and *parent-node n'* = *sourcenode a'*
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from ‹(*-Entry-*) $-as\rightarrow_{\surd}*$ *m*› ‹*as* = *as'*@*a*#*as''*› ‹*as''* = *xs*@*a'*#*ys*›
have (*-Entry-*) $-(as'@a\#xs)@a'\#ys\rightarrow_{\surd}*$ *m* **by** *simp*
hence (*-Entry-*) $-as'@a\#xs\rightarrow_{\surd}*$ *sourcenode a'*
and *valid-edge a'* **by**(*auto intro*:*vp-split*)
from ‹*as* = *as'*@*a*#*as''*› ‹*as''* = *xs*@*a'*#*ys*›
have *length* (*as'*@*a*#*xs*) < *length as* **by** *simp*
from ‹*valid-edge a'*› **have** *valid-node* (*sourcenode a'*) **by** *simp*
hence *inner-node* (*sourcenode a'*)
proof(*cases sourcenode a' rule*:*valid-node-cases*)
case *Entry*
with ‹(*-Entry-*) $-as'@a\#xs\rightarrow_{\surd}*$ *sourcenode a'*›
have (*-Entry-*) $-as'@a\#xs\rightarrow*$ (*-Entry-*) **by**(*fastforce simp*:*vp-def*)
hence *False* **by** *fastforce*
thus *?thesis* **by** *simp*
next
case *Exit*
with ‹*valid-edge a'*› **have** *False* **by** −(*erule Exit-source*)
thus *?thesis* **by** *simp*
next
case *inner*
thus *?thesis* **by** *simp*
qed
from ‹*valid-edge a'*› **have** *valid-SDG-node* (*CFG-node* (*sourcenode a'*))
by *simp*
from ‹$\forall\, a' \in$ *set as*. *intra-kind*(*kind a'*) ∨ (∃ *Q r p fs*. *kind a'* = *Q*:*r*$\hookrightarrow_p$*fs*)›
‹*as* = *as'*@*a*#*as''*› ‹*as''* = *xs*@*a'*#*ys*›
have $\forall\, a' \in$ *set* (*as'*@*a*#*xs*).
intra-kind(*kind a'*) ∨ (∃ *Q r p fs*. *kind a'* = *Q*:*r*$\hookrightarrow_p$*fs*)

```
          by auto
        with IH ‹length (as'@a#xs) < length as›
          ‹(-Entry-) −as'@a#xs→√* sourcenode a'›
          ‹valid-SDG-node (CFG-node (sourcenode a'))›
          ‹inner-node (sourcenode a')› ‹parent-node n' = sourcenode a'›
       obtain ns where CFG-node (-Entry-) cc−ns→d* CFG-node (parent-node
n')
          apply(erule-tac x=as'@a#xs in allE) apply clarsimp
          apply(erule-tac x=sourcenode a' in allE) apply clarsimp
          apply(erule-tac x=CFG-node (sourcenode a') in allE) by clarsimp
        from ‹n' −→cd CFG-node m› have valid-SDG-node n'
          by(rule SDG-edge-valid-SDG-node)
          hence n' = CFG-node (parent-node n') ∨ CFG-node (parent-node n')
−→cd n'
          by(rule valid-SDG-node-cases)
        thus ?thesis
        proof
          assume n' = CFG-node (parent-node n')
          with ‹CFG-node (-Entry-) cc−ns→d* CFG-node (parent-node n')›
            ‹n' −→cd CFG-node m› show ?thesis
            by(fastforce intro:ccSp-Append-cdep)
        next
          assume CFG-node (parent-node n') −→cd n'
          with ‹CFG-node (-Entry-) cc−ns→d* CFG-node (parent-node n')›
          have CFG-node (-Entry-) cc−ns@[CFG-node (parent-node n')]→d* n'
            by(fastforce intro:ccSp-Append-cdep)
          with ‹n' −→cd CFG-node m› show ?thesis
            by(fastforce intro:ccSp-Append-cdep)
        qed
      qed
       then obtain ns where CFG-node (-Entry-) cc−ns→d* CFG-node m by
blast
      show ?thesis
      proof(cases n = CFG-node m)
        case True
          with ‹CFG-node (-Entry-) cc−ns→d* CFG-node m› show ?thesis by
fastforce
      next
        case False
        with ‹inner-node m› ‹valid-SDG-node n› ‹m = parent-node n›
        have CFG-node m −→cd n
          by(fastforce intro:SDG-parent-cdep-edge inner-is-valid)
        with ‹CFG-node (-Entry-) cc−ns→d* CFG-node m› show ?thesis
          by(fastforce dest:ccSp-Append-cdep)
      qed
    qed
  qed
 qed
qed
```

1.8.8 Same level paths in the SDG

inductive *matched* :: *'node SDG-node* ⇒ *'node SDG-node list* ⇒ *'node SDG-node* ⇒ *bool*
where *matched-Nil*:
valid-SDG-node n ⟹ *matched n* [] *n*
| *matched-Append-intra-SDG-path*:
⟦*matched n ns n''*; *n'' i−ns'→$_d$* n'*⟧ ⟹ *matched n* (*ns@ns'*) *n'*
| *matched-bracket-call*:
⟦*matched* n_0 *ns* n_1; n_1 *−p→$_{call}$* n_2; *matched* n_2 *ns'* n_3;
(n_3 *−p→$_{ret}$* n_4 ∨ n_3 *−p:V→$_{out}$* n_4); *valid-edge a*; *a'* ∈ *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⟧
⟹ *matched* n_0 (*ns@*n_1#*ns'*@[n_3]) n_4
| *matched-bracket-param*:
⟦*matched* n_0 *ns* n_1; n_1 *−p:V→$_{in}$* n_2; *matched* n_2 *ns'* n_3;
n_3 *−p:V'→$_{out}$* n_4; *valid-edge a*; *a'* ∈ *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⟧
⟹ *matched* n_0 (*ns@*n_1#*ns'*@[n_3]) n_4

lemma *matched-Append*:
⟦*matched n'' ns' n'*; *matched n ns n''*⟧ ⟹ *matched n* (*ns@ns'*) *n'*
by(*induct rule*:*matched.induct*,
auto intro:*matched.intros simp*:*append-assoc*[*THEN sym*] *simp del*:*append-assoc*)

lemma *intra-SDG-path-matched*:
assumes *n i−ns→$_d$* n'* **shows** *matched n ns n'*
proof −
from ‹*n i−ns→$_d$* n'*› **have** *valid-SDG-node n*
by(*rule intra-SDG-path-valid-SDG-node*)
hence *matched n* [] *n* **by**(*rule matched-Nil*)
with ‹*n i−ns→$_d$* n'*› **have** *matched n* ([]@*ns*) *n'*
by −(*rule matched-Append-intra-SDG-path*)
thus *?thesis* **by** *simp*
qed

lemma *intra-proc-matched*:
assumes *valid-edge a* **and** *kind a* = *Q*:*r↪$_p$fs* **and** *a'* ∈ *get-return-edges a*
shows *matched* (*CFG-node* (*targetnode a*)) [*CFG-node* (*targetnode a*)]
(*CFG-node* (*sourcenode a'*))
proof −
from *assms* **have** *CFG-node* (*targetnode a*) ⟶$_{cd}$ *CFG-node* (*sourcenode a'*)
by(*fastforce intro*:*SDG-proc-entry-exit-cdep*)
with ‹*valid-edge a*›

have *CFG-node* (*targetnode a*) *i*−[]@[*CFG-node* (*targetnode a*)]→$_d$*
CFG-node (*sourcenode a'*)
by(*fastforce intro*:*intra-SDG-path.intros*)
with ‹*valid-edge a*›
have *matched* (*CFG-node* (*targetnode a*)) ([]@[*CFG-node* (*targetnode a*)])
(*CFG-node* (*sourcenode a'*))
by(*fastforce intro*:*matched.intros*)
thus *?thesis* **by** *simp*
qed

lemma *matched-intra-CFG-path*:
assumes *matched n ns n'*
obtains *as* **where** *parent-node n* −*as*→$_\iota$* *parent-node n'*
proof(*atomize-elim*)
from ‹*matched n ns n'*› **show** ∃ *as*. *parent-node n* −*as*→$_\iota$* *parent-node n'*
proof(*induct rule*:*matched.induct*)
case *matched-Nil* **thus** *?case*
by(*fastforce dest*:*empty-path valid-SDG-CFG-node simp*:*intra-path-def*)
next
case (*matched-Append-intra-SDG-path n ns n'' ns' n'*)
from ‹∃ *as*. *parent-node n* −*as*→$_\iota$* *parent-node n''*› **obtain** *as*
where *parent-node n* −*as*→$_\iota$* *parent-node n''* **by** *blast*
from ‹*n''* *i*−*ns'*→$_d$* *n'*› **obtain** *as'* **where** *parent-node n''* −*as'*→$_\iota$* *parent-node n'*
by(*fastforce elim*:*intra-SDG-path-intra-CFG-path*)
with ‹*parent-node n* −*as*→$_\iota$* *parent-node n''*›
have *parent-node n* −*as*@*as'*→$_\iota$* *parent-node n'*
by(*rule intra-path-Append*)
thus *?case* **by** *fastforce*
next
case (*matched-bracket-call* n_0 *ns* n_1 *p* n_2 *ns'* n_3 n_4 *V a a'*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› ‹*sourcenode a* = *parent-node* n_1›
‹*targetnode a'* = *parent-node* n_4›
obtain *a''* **where** *valid-edge a''* **and** *sourcenode a''* = *parent-node* n_1
and *targetnode a''* = *parent-node* n_4 **and** *kind a''* = $(\lambda cf.\ False)_\surd$
by(*fastforce dest*:*call-return-node-edge*)
hence *parent-node* n_1 −[*a''*]→* *parent-node* n_4 **by**(*fastforce dest*:*path-edge*)
moreover
from ‹*kind a''* = $(\lambda cf.\ False)_\surd$› **have** ∀ *a* ∈ *set* [*a''*]. *intra-kind*(*kind a*)
by(*fastforce simp*:*intra-kind-def*)
ultimately have *parent-node* n_1 −[*a''*]→$_\iota$* *parent-node* n_4
by(*auto simp*:*intra-path-def*)
with ‹∃ *as*. *parent-node* n_0 −*as*→$_\iota$* *parent-node* n_1› **show** *?case*
by(*fastforce intro*:*intra-path-Append*)
next
case (*matched-bracket-param* n_0 *ns* n_1 *p V* n_2 *ns'* n_3 *V'* n_4 *a a'*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› ‹*sourcenode a* = *parent-node* n_1›
‹*targetnode a'* = *parent-node* n_4›

```
    obtain a'' where valid-edge a'' and sourcenode a'' = parent-node n₁
      and targetnode a'' = parent-node n₄ and kind a'' = (λcf. False)√
      by(fastforce dest:call-return-node-edge)
    hence parent-node n₁ −[a'']→* parent-node n₄ by(fastforce dest:path-edge)
    moreover
    from ‹kind a'' = (λcf. False)√› have ∀ a ∈ set [a'']. intra-kind(kind a)
      by(fastforce simp:intra-kind-def)
    ultimately have parent-node n₁ −[a'']→ι* parent-node n₄
      by(auto simp:intra-path-def)
    with ‹∃ as. parent-node n₀ −as→ι* parent-node n₁› show ?case
      by(fastforce intro:intra-path-Append)
  qed
qed


lemma matched-same-level-CFG-path:
  assumes matched n ns n'
  obtains as where parent-node n −as→sl* parent-node n'
proof(atomize-elim)
  from ‹matched n ns n'›
  show ∃ as. parent-node n −as→sl* parent-node n'
  proof(induct rule:matched.induct)
    case matched-Nil thus ?case
    by(fastforce dest:empty-path valid-SDG-CFG-node simp:slp-def same-level-path-def)
  next
    case (matched-Append-intra-SDG-path n ns n'' ns' n')
    from ‹∃ as. parent-node n −as→sl* parent-node n''›
    obtain as where parent-node n −as→sl* parent-node n'' by blast
   from ‹n'' i−ns'→d* n'› obtain as' where parent-node n'' −as'→ι* parent-node
n'
      by(erule intra-SDG-path-intra-CFG-path)
    from ‹parent-node n'' −as'→ι* parent-node n'›
    have parent-node n'' −as'→sl* parent-node n' by(rule intra-path-slp)
    with ‹parent-node n −as→sl* parent-node n''›
    have parent-node n −as@as'→sl* parent-node n'
      by(rule slp-Append)
    thus ?case by fastforce
  next
    case (matched-bracket-call n₀ ns n₁ p n₂ ns' n₃ n₄ V a a')
    from ‹valid-edge a› ‹a' ∈ get-return-edges a›
    obtain Q r p' fs where kind a = Q:r↪p'fs
      by(fastforce dest!:only-call-get-return-edges)
    from ‹∃ as. parent-node n₀ −as→sl* parent-node n₁›
    obtain as where parent-node n₀ −as→sl* parent-node n₁ by blast
    from ‹∃ as. parent-node n₂ −as→sl* parent-node n₃›
    obtain as' where parent-node n₂ −as'→sl* parent-node n₃ by blast
    from ‹valid-edge a› ‹a' ∈ get-return-edges a› ‹kind a = Q:r↪p'fs›
    obtain Q' f' where kind a' = Q'↩p'f' by(fastforce dest!:call-return-edges)
    from ‹valid-edge a› ‹a' ∈ get-return-edges a› have valid-edge a'
```

by(*rule get-return-edges-valid*)
from ‹*parent-node* n_2 $-as'\rightarrow_{sl}*$ *parent-node* n_3› **have** *same-level-path as'*
by(*simp add*:*slp-def*)
hence *same-level-path-aux* ($[]@[a]$) *as'*
by(*fastforce intro*:*same-level-path-aux-callstack-Append simp*:*same-level-path-def*)
from ‹*same-level-path as'*› **have** *upd-cs* ($[]@[a]$) *as'* = ($[]@[a]$)
by(*fastforce intro*:*same-level-path-upd-cs-callstack-Append*
simp:*same-level-path-def*)
with ‹*same-level-path-aux* ($[]@[a]$) *as'*› ‹*a'* ∈ *get-return-edges a*›
‹*kind a* = *Q*:$r\hookrightarrow_{p'}fs$› ‹*kind a'* = $Q'\hookleftarrow_{p'}f'$›
have *same-level-path* ($a\#as'@[a']$)
by(*fastforce intro*:*same-level-path-aux-Append upd-cs-Append*
simp:*same-level-path-def*)
from ‹*valid-edge a'*› ‹*sourcenode a'* = *parent-node* n_3›
‹*targetnode a'* = *parent-node* n_4›
have *parent-node* n_3 $-[a']\rightarrow*$ *parent-node* n_4 **by**(*fastforce dest*:*path-edge*)
with ‹*parent-node* n_2 $-as'\rightarrow_{sl}*$ *parent-node* n_3›
have *parent-node* n_2 $-as'@[a']\rightarrow*$ *parent-node* n_4
by(*fastforce intro*:*path-Append simp*:*slp-def*)
with ‹*valid-edge a*› ‹*sourcenode a* = *parent-node* n_1›
‹*targetnode a* = *parent-node* n_2›
have *parent-node* n_1 $-a\#as'@[a']\rightarrow*$ *parent-node* n_4 **by** $-$(*rule Cons-path*)
with ‹*same-level-path* ($a\#as'@[a']$)›
have *parent-node* n_1 $-a\#as'@[a']\rightarrow_{sl}*$ *parent-node* n_4 **by**(*simp add*:*slp-def*)
with ‹*parent-node* n_0 $-as\rightarrow_{sl}*$ *parent-node* n_1›
have *parent-node* n_0 $-as@a\#as'@[a']\rightarrow_{sl}*$ *parent-node* n_4 **by**(*rule slp-Append*)
with ‹*sourcenode a* = *parent-node* n_1› ‹*sourcenode a'* = *parent-node* n_3›
show *?case* **by** *fastforce*
next
case (*matched-bracket-param* n_0 *ns* n_1 *p V* n_2 *ns'* n_3 *V'* n_4 *a a'*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
obtain *Q r p' fs* **where** *kind a* = *Q*:$r\hookrightarrow_{p'}fs$
by(*fastforce dest*!:*only-call-get-return-edges*)
from ‹∃ *as*. *parent-node* n_0 $-as\rightarrow_{sl}*$ *parent-node* n_1›
obtain *as* **where** *parent-node* n_0 $-as\rightarrow_{sl}*$ *parent-node* n_1 **by** *blast*
from ‹∃ *as*. *parent-node* n_2 $-as\rightarrow_{sl}*$ *parent-node* n_3›
obtain *as'* **where** *parent-node* n_2 $-as'\rightarrow_{sl}*$ *parent-node* n_3 **by** *blast*
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› ‹*kind a* = *Q*:$r\hookrightarrow_{p'}fs$›
obtain Q' f' **where** *kind a'* = $Q'\hookleftarrow_{p'}f'$ **by**(*fastforce dest*!:*call-return-edges*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹*parent-node* n_2 $-as'\rightarrow_{sl}*$ *parent-node* n_3› **have** *same-level-path as'*
by(*simp add*:*slp-def*)
hence *same-level-path-aux* ($[]@[a]$) *as'*
by(*fastforce intro*:*same-level-path-aux-callstack-Append simp*:*same-level-path-def*)
from ‹*same-level-path as'*› **have** *upd-cs* ($[]@[a]$) *as'* = ($[]@[a]$)
by(*fastforce intro*:*same-level-path-upd-cs-callstack-Append*
simp:*same-level-path-def*)
with ‹*same-level-path-aux* ($[]@[a]$) *as'*› ‹*a'* ∈ *get-return-edges a*›

⟨*kind* $a = Q{:}r\hookrightarrow_p fs$⟩ ⟨*kind* $a' = Q'{:}r'\hookrightarrow_p f'$⟩
have *same-level-path* $(a\#as'@[a'])$
by(*fastforce intro*:*same-level-path-aux-Append upd-cs-Append simp*:*same-level-path-def*)
from ⟨*valid-edge* a'⟩ ⟨*sourcenode* $a' =$ *parent-node* n_3⟩
⟨*targetnode* $a' =$ *parent-node* n_4⟩
have *parent-node* n_3 $-[a']\rightarrow*$ *parent-node* n_4 **by**(*fastforce dest*:*path-edge*)
with ⟨*parent-node* n_2 $-as'\rightarrow_{sl}*$ *parent-node* n_3⟩
have *parent-node* n_2 $-as'@[a']\rightarrow*$ *parent-node* n_4
by(*fastforce intro*:*path-Append simp*:*slp-def*)
with ⟨*valid-edge* a⟩ ⟨*sourcenode* $a =$ *parent-node* n_1⟩
⟨*targetnode* $a =$ *parent-node* n_2⟩
have *parent-node* n_1 $-a\#as'@[a']\rightarrow*$ *parent-node* n_4 **by** $-$(*rule Cons-path*)
with ⟨*same-level-path* $(a\#as'@[a'])$⟩
have *parent-node* n_1 $-a\#as'@[a']\rightarrow_{sl}*$ *parent-node* n_4 **by**(*simp add*:*slp-def*)
with ⟨*parent-node* n_0 $-as\rightarrow_{sl}*$ *parent-node* n_1⟩
have *parent-node* n_0 $-as@a\#as'@[a']\rightarrow_{sl}*$ *parent-node* n_4 **by**(*rule slp-Append*)
with ⟨*sourcenode* $a =$ *parent-node* n_1⟩ ⟨*sourcenode* $a' =$ *parent-node* n_3⟩
show *?case* **by** *fastforce*
qed
qed

1.8.9 Realizable paths in the SDG

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

lemma *realizable-Append-matched*:
⟦*realizable n ns n''*; *matched n'' ns' n'*⟧ $\Longrightarrow$ *realizable n* $(ns@ns')$ *n'*
proof(*induct rule*:*realizable.induct*)
case (*realizable-matched n ns n''*)
from ⟨*matched n'' ns' n'*⟩ ⟨*matched n ns n''*⟩ **have** *matched n* $(ns@ns')$ *n'*
by(*rule matched-Append*)
thus *?case* **by**(*rule realizable.realizable-matched*)
next
case (*realizable-call* n_0 *ns* n_1 *p* n_2 *V ns''* n_3)
from ⟨*matched* n_3 *ns' n'*⟩ ⟨*matched* n_2 *ns''* n_3⟩ **have** *matched* n_2 $(ns''@ns')$ *n'*
by(*rule matched-Append*)
with ⟨*realizable* n_0 *ns* n_1⟩ ⟨n_1 $-p\rightarrow_{call}$ n_2 $\vee$ n_1 $-p{:}V\rightarrow_{in}$ n_2⟩
have *realizable* n_0 $(ns@n_1\#(ns''@ns'))$ *n'*
by(*rule realizable.realizable-call*)
thus *?case* **by** *simp*
qed

lemma *realizable-valid-CFG-path*:
assumes *realizable n ns n′*
obtains *as* **where** *parent-node n* $-as\rightarrow_{\surd}*$ *parent-node n′*
proof(*atomize-elim*)
from ‹*realizable n ns n′*›
show ∃ *as*. *parent-node n* $-as\rightarrow_{\surd}*$ *parent-node n′*
proof(*induct rule*:*realizable.induct*)
case (*realizable-matched n ns n′*)
from ‹*matched n ns n′*› **obtain** *as* **where** *parent-node n* $-as\rightarrow_{sl}*$ *parent-node n′*
by(*erule matched-same-level-CFG-path*)
thus *?case* **by**(*fastforce intro*:*slp-vp*)
next
case (*realizable-call* n_0 *ns* n_1 *p* n_2 *V ns′* n_3)
from ‹∃ *as*. *parent-node* n_0 $-as\rightarrow_{\surd}*$ *parent-node* n_1›
obtain *as* **where** *parent-node* n_0 $-as\rightarrow_{\surd}*$ *parent-node* n_1 **by** *blast*
from ‹*matched* n_2 *ns′* n_3› **obtain** *as′* **where** *parent-node* n_2 $-as'\rightarrow_{sl}*$ *parent-node* n_3
by(*erule matched-same-level-CFG-path*)
from ‹n_1 $-p\rightarrow_{call}$ n_2 ∨ n_1 $-p$:$V\rightarrow_{in}$ n_2›
obtain *a Q r fs* **where** *valid-edge a*
and *sourcenode a* = *parent-node* n_1 **and** *targetnode a* = *parent-node* n_2
and *kind a* = *Q*:$r\hookrightarrow_p$*fs* **by**(*fastforce elim*:*SDG-edge.cases*)+
hence *parent-node* n_1 $-[a]\rightarrow*$ *parent-node* n_2
by(*fastforce dest*:*path-edge*)
from ‹*parent-node* n_0 $-as\rightarrow_{\surd}*$ *parent-node* n_1›
have *parent-node* n_0 $-as\rightarrow*$ *parent-node* n_1 **and** *valid-path as*
by(*simp-all add*:*vp-def*)
with ‹*kind a* = *Q*:$r\hookrightarrow_p$*fs*› **have** *valid-path* (*as*@[*a*])
by(*fastforce elim*:*valid-path-aux-Append simp*:*valid-path-def*)
moreover
from ‹*parent-node* n_0 $-as\rightarrow*$ *parent-node* n_1› ‹*parent-node* n_1 $-[a]\rightarrow*$ *parent-node* n_2›
have *parent-node* n_0 $-as@[a]\rightarrow*$ *parent-node* n_2 **by**(*rule path-Append*)
ultimately have *parent-node* n_0 $-as@[a]\rightarrow_{\surd}*$ *parent-node* n_2 **by**(*simp add*:*vp-def*)
with ‹*parent-node* n_2 $-as'\rightarrow_{sl}*$ *parent-node* n_3›
have *parent-node* n_0 $-(as@[a])@as'\rightarrow_{\surd}*$ *parent-node* n_3 **by** −(*rule vp-slp-Append*)
with ‹*sourcenode a* = *parent-node* n_1› **show** *?case* **by** *fastforce*
qed
qed

lemma *cdep-SDG-path-realizable*:
n cc$-ns\rightarrow_d*$ *n′* ⟹ *realizable n ns n′*
proof(*induct rule*:*call-cdep-SDG-path.induct*)
case (*ccSp-Nil n*)
from ‹*valid-SDG-node n*› **show** *?case*
by(*fastforce intro*:*realizable-matched matched-Nil*)

next
case (*ccSp-Append-cdep n ns n'' n'*)
from ‹$n'' \longrightarrow_{cd} n'$› **have** *valid-SDG-node n''* **by**(*rule SDG-edge-valid-SDG-node*)
hence *matched n'' [] n''* **by**(*rule matched-Nil*)
from ‹$n'' \longrightarrow_{cd} n'$› ‹*valid-SDG-node n''*›
have $n''\ i{-}[]@[n']{\rightarrow_d}*\ n'$
by(*fastforce intro*:*iSp-Append-cdep iSp-Nil*)
with ‹*matched n'' [] n''*› **have** *matched n''* ($[]@[n']$) *n'*
by(*fastforce intro*:*matched-Append-intra-SDG-path*)
with ‹*realizable n ns n''*› **show** *?case*
by(*fastforce intro*:*realizable-Append-matched*)
next
case (*ccSp-Append-call n ns n'' p n'*)
from ‹$n'' -p\rightarrow_{call} n'$› **have** *valid-SDG-node n'* **by**(*rule SDG-edge-valid-SDG-node*)
hence *matched n' [] n'* **by**(*rule matched-Nil*)
with ‹*realizable n ns n''*› ‹$n'' -p\rightarrow_{call} n'$›
show *?case* **by**(*fastforce intro*:*realizable-call*)
qed

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

where

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

| $n\ s{-}p{\rightarrow}_{sum}\ n' == sum\text{-}SDG\text{-}edge\ n\ None\ (Some(p,True))\ True\ n'$

| *sum-SDG-cdep-edge*:
$[\![n = CFG\text{-}node\ m;\ n' = CFG\text{-}node\ m';\ m\ controls\ m']\!] \Longrightarrow n\ s{\longrightarrow}_{cd}\ n'$
| *sum-SDG-proc-entry-exit-cdep*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow}_{p}fs;\ n = CFG\text{-}node\ (targetnode\ a);$
$a' \in get\text{-}return\text{-}edges\ a;\ n' = CFG\text{-}node\ (sourcenode\ a')]\!] \Longrightarrow n\ s{\longrightarrow}_{cd}\ n'$
| *sum-SDG-parent-cdep-edge*:
$[\![valid\text{-}SDG\text{-}node\ n';\ m = parent\text{-}node\ n';\ n = CFG\text{-}node\ m;\ n \neq n']\!]$
$\Longrightarrow n\ s{\longrightarrow}_{cd}\ n'$
| *sum-SDG-ddep-edge*:$n\ influences\ V\ in\ n' \Longrightarrow n\ s{-}V{\rightarrow}_{dd}\ n'$
| *sum-SDG-call-edge*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow}_{p}fs;\ n = CFG\text{-}node\ (sourcenode\ a);$
$n' = CFG\text{-}node\ (targetnode\ a)]\!] \Longrightarrow n\ s{-}p{\rightarrow}_{call}\ n'$
| *sum-SDG-return-edge*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{\hookleftarrow}_{p}fs;\ n = CFG\text{-}node\ (sourcenode\ a);$
$n' = CFG\text{-}node\ (targetnode\ a)]\!] \Longrightarrow n\ s{-}p{\rightarrow}_{ret}\ n'$
| *sum-SDG-param-in-edge*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow}_{p}fs;\ (p,ins,outs) \in set\ procs;\ V = ins!x;$
$x < length\ ins;\ n = Actual\text{-}in\ (sourcenode\ a,x);\ n' = Formal\text{-}in\ (targetnode\ a,x)]\!]$
$\Longrightarrow n\ s{-}p{:}V{\rightarrow}_{in}\ n'$
| *sum-SDG-param-out-edge*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{\hookleftarrow}_{p}f;\ (p,ins,outs) \in set\ procs;\ V = outs!x;$
$x < length\ outs;\ n = Formal\text{-}out\ (sourcenode\ a,x);$
$n' = Actual\text{-}out\ (targetnode\ a,x)]\!]$
$\Longrightarrow n\ s{-}p{:}V{\rightarrow}_{out}\ n'$
| *sum-SDG-call-summary-edge*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow}_{p}fs;\ a' \in get\text{-}return\text{-}edges\ a;$
$n = CFG\text{-}node\ (sourcenode\ a);\ n' = CFG\text{-}node\ (targetnode\ a')]\!]$
$\Longrightarrow n\ s{-}p{\rightarrow}_{sum}\ n'$
| *sum-SDG-param-summary-edge*:
$[\![valid\text{-}edge\ a;\ kind\ a = Q{:}r{\hookrightarrow}_{p}fs;\ a' \in get\text{-}return\text{-}edges\ a;$
$matched\ (Formal\text{-}in\ (targetnode\ a,x))\ ns\ (Formal\text{-}out\ (sourcenode\ a',x'));$
$n = Actual\text{-}in\ (sourcenode\ a,x);\ n' = Actual\text{-}out\ (targetnode\ a',x');$
$(p,ins,outs) \in set\ procs;\ x < length\ ins;\ x' < length\ outs]\!]$
$\Longrightarrow n\ s{-}p{\rightarrow}_{sum}\ n'$

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-in (sourcenode a,x); n' = Actual-out (targetnode a',x');
      (p,ins,outs) ∈ set procs; x < length ins; x' < length outs⟧ ⟹ P⟧
  ⟹ P
by −(erule sum-SDG-edge.cases,auto,fastforce+)


lemma SDG-edge-sum-SDG-edge:
  SDG-edge n Vopt popt n' ⟹ sum-SDG-edge n Vopt popt False n'
  by(induct rule:SDG-edge.induct,auto intro:sum-SDG-edge.intros)


lemma sum-SDG-edge-SDG-edge:
  sum-SDG-edge n Vopt popt False n' ⟹ SDG-edge n Vopt popt n'
by(induct n Vopt popt x≡False n' rule:sum-SDG-edge.induct,
   auto intro:SDG-edge.intros)


lemma sum-SDG-edge-valid-SDG-node:
  assumes sum-SDG-edge n Vopt popt b n'
  shows valid-SDG-node n and valid-SDG-node n'
proof −
  have valid-SDG-node n ∧ valid-SDG-node n'
  proof(cases b)
    case True
    with ‹sum-SDG-edge n Vopt popt b n'› show ?thesis
    proof(induct rule:sum-SDG-edge.induct)
      case (sum-SDG-call-summary-edge a Q r p f a' n n')
      from ‹valid-edge a› ‹n = CFG-node (sourcenode a)›
      have valid-SDG-node n by fastforce
      from ‹valid-edge a› ‹a' ∈ get-return-edges a› have valid-edge a'
        by(rule get-return-edges-valid)
      with ‹n' = CFG-node (targetnode a')› have valid-SDG-node n' by fastforce
      with ‹valid-SDG-node n› show ?case by simp
    next
      case (sum-SDG-param-summary-edge a Q r p fs a' x ns x' n n' ins outs)
      from ‹valid-edge a› ‹kind a = Q:r↪pfs› ‹n = Actual-in (sourcenode a,x)›
        ‹(p,ins,outs) ∈ set procs› ‹x < length ins›
      have valid-SDG-node n by fastforce
      from ‹valid-edge a› ‹a' ∈ get-return-edges a› have valid-edge a'
        by(rule get-return-edges-valid)
      from ‹valid-edge a› ‹a' ∈ get-return-edges a› ‹kind a = Q:r↪pfs›
      obtain Q' f' where kind a' = Q'↩pf' by(fastforce dest!:call-return-edges)
      with ‹valid-edge a'› ‹n' = Actual-out (targetnode a',x')›
        ‹(p,ins,outs) ∈ set procs› ‹x' < length outs›
      have valid-SDG-node n' by fastforce
      with ‹valid-SDG-node n› show ?case by simp
    qed simp-all
  next
```

case *False*
with ‹*sum-SDG-edge n Vopt popt b n'*› **have** *SDG-edge n Vopt popt n'*
by(*fastforce intro*:*sum-SDG-edge-SDG-edge*)
thus *?thesis* **by**(*fastforce intro*:*SDG-edge-valid-SDG-node*)
qed
thus *valid-SDG-node n* **and** *valid-SDG-node n'* **by** *simp-all*
qed

lemma *Exit-no-sum-SDG-edge-source*:
assumes *sum-SDG-edge* (*CFG-node* (*-Exit-*)) *Vopt popt b n'* **shows** *False*
proof(*cases b*)
case *True*
with ‹*sum-SDG-edge* (*CFG-node* (*-Exit-*)) *Vopt popt b n'*› **show** *?thesis*
proof(*induct CFG-node* (*-Exit-*) *Vopt popt b n' rule*:*sum-SDG-edge.induct*)
case (*sum-SDG-call-summary-edge a Q r p f a' n'*)
from ‹*CFG-node* (*-Exit-*) = *CFG-node* (*sourcenode a*)›
have *sourcenode a* = (*-Exit-*) **by** *simp*
with ‹*valid-edge a*› **show** *?case* **by**(*rule Exit-source*)
next
case (*sum-SDG-param-summary-edge a Q r p f a' x ns x' n' ins outs*)
thus *?case* **by** *simp*
qed *simp-all*
next
case *False*
with ‹*sum-SDG-edge* (*CFG-node* (*-Exit-*)) *Vopt popt b n'*›
have *SDG-edge* (*CFG-node* (*-Exit-*)) *Vopt popt n'*
by(*fastforce intro*:*sum-SDG-edge-SDG-edge*)
thus *?thesis* **by**(*fastforce intro*:*Exit-no-SDG-edge-source*)
qed

lemma *Exit-no-sum-SDG-edge-target*:
sum-SDG-edge n Vopt popt b (*CFG-node* (*-Exit-*)) ⟹ *False*
proof(*induct CFG-node* (*-Exit-*) *rule*:*sum-SDG-edge.induct*)
case (*sum-SDG-cdep-edge n m m'*)
from ‹*m controls m'*› ‹*CFG-node* (*-Exit-*) = *CFG-node m'*›
have *m controls* (*-Exit-*) **by** *simp*
hence *False* **by**(*fastforce dest*:*Exit-not-control-dependent*)
thus *?case* **by** *simp*
next
case (*sum-SDG-proc-entry-exit-cdep a Q r p f n a'*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
moreover
from ‹*CFG-node* (*-Exit-*) = *CFG-node* (*sourcenode a'*)›
have *sourcenode a'* = (*-Exit-*) **by** *simp*
ultimately have *False* **by**(*rule Exit-source*)
thus *?case* **by** *simp*

next
case (*sum-SDG-ddep-edge n V*) **thus** *?case*
by(*fastforce elim*:*SDG-Use.cases simp*:*data-dependence-def*)
next
case (*sum-SDG-call-edge a Q r p fs n*)
from ‹*CFG-node* (*-Exit-*) = *CFG-node* (*targetnode a*)›
have *targetnode a* = (*-Exit-*) **by** *simp*
with ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› **have** *get-proc* (*-Exit-*) = *p*
by(*fastforce intro*:*get-proc-call*)
hence *p* = *Main* **by**(*simp add*:*get-proc-Exit*)
with ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› **have** *False*
by(*fastforce intro*:*Main-no-call-target*)
thus *?case* **by** *simp*
next
case (*sum-SDG-return-edge a Q p f n*)
from ‹*CFG-node* (*-Exit-*) = *CFG-node* (*targetnode a*)›
have *targetnode a* = (*-Exit-*) **by** *simp*
with ‹*valid-edge a*› ‹*kind a* = *Q*↩$_p$*f*› **have** *False* **by**(*rule Exit-no-return-target*)
thus *?case* **by** *simp*
next
case (*sum-SDG-call-summary-edge a Q r p fs a' n*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*a'* ∈ *get-return-edges a*›
obtain *Q' f'* **where** *kind a'* = *Q'*↩$_p$*f'* **by**(*fastforce dest*!:*call-return-edges*)
from ‹*CFG-node* (*-Exit-*) = *CFG-node* (*targetnode a'*)›
have *targetnode a'* = (*-Exit-*) **by** *simp*
with ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩$_p$*f'*› **have** *False* **by**(*rule Exit-no-return-target*)
thus *?case* **by** *simp*
qed *simp+*

lemma *sum-SDG-summary-edge-matched*:
assumes *n s*−*p*→$_{sum}$ *n'*
obtains *ns* **where** *matched n ns n'* **and** *n* ∈ *set ns*
and *get-proc* (*parent-node*(*last ns*)) = *p*
proof(*atomize-elim*)
from ‹*n s*−*p*→$_{sum}$ *n'*›
show ∃ *ns*. *matched n ns n'* ∧ *n* ∈ *set ns* ∧ *get-proc* (*parent-node*(*last ns*)) = *p*
proof(*induct n None*::*'var option Some*(*p*,*True*) *True n'*
rule:*sum-SDG-edge.induct*)
case (*sum-SDG-call-summary-edge a Q r fs a' n n'*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*n* = *CFG-node* (*sourcenode a*)›
have *n* −*p*→$_{call}$ *CFG-node* (*targetnode a*) **by**(*fastforce intro*:*SDG-call-edge*)
hence *valid-SDG-node n* **by**(*rule SDG-edge-valid-SDG-node*)
hence *matched n* [] *n* **by**(*rule matched-Nil*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)

from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*a'* ∈ *get-return-edges a*›
have *matched*:*matched* (*CFG-node* (*targetnode a*)) [*CFG-node* (*targetnode a*)] (*CFG-node* (*sourcenode a'*)) **by**(*rule intra-proc-matched*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
obtain *Q' f'* **where** *kind a'* = *Q'*↩$_p$*f'* **by**(*fastforce dest*!:*call-return-edges*)
with ‹*valid-edge a'*› **have** *get-proc* (*sourcenode a'*) = *p* **by**(*rule get-proc-return*)
from ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩$_p$*f'*› ‹*n'* = *CFG-node* (*targetnode a'*)›
have *CFG-node* (*sourcenode a'*) −*p*→$_{ret}$ *n'* **by**(*fastforce intro*:*SDG-return-edge*)
from ‹*matched n* [] *n*› ‹*n* −*p*→$_{call}$ *CFG-node* (*targetnode a*)› *matched* ‹*CFG-node* (*sourcenode a'*) −*p*→$_{ret}$ *n'*› ‹*a'* ∈ *get-return-edges a*› ‹*n* = *CFG-node* (*sourcenode a*)› ‹*n'* = *CFG-node* (*targetnode a'*)› ‹*valid-edge a*›
have *matched n* ([]@*n*#[*CFG-node* (*targetnode a*)]@[*CFG-node* (*sourcenode a'*)]) *n'*
by(*fastforce intro*:*matched-bracket-call*)
with ‹*get-proc* (*sourcenode a'*) = *p*› **show** *?case* **by** *auto*
next
case (*sum-SDG-param-summary-edge a Q r fs a' x ns x' n n' ins outs*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹(*p*,*ins*,*outs*) ∈ *set procs*› ‹*x* < *length ins*› ‹*n* = *Actual-in* (*sourcenode a*,*x*)›
have *n* −*p*:*ins*!*x*→$_{in}$ *Formal-in* (*targetnode a*,*x*)
by(*fastforce intro*:*SDG-param-in-edge*)
hence *valid-SDG-node n* **by**(*rule SDG-edge-valid-SDG-node*)
hence *matched n* [] *n* **by**(*rule matched-Nil*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
obtain *Q' f'* **where** *kind a'* = *Q'*↩$_p$*f'* **by**(*fastforce dest*!:*call-return-edges*)
with ‹*valid-edge a'*› **have** *get-proc* (*sourcenode a'*) = *p* **by**(*rule get-proc-return*)
from ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩$_p$*f'*› ‹(*p*,*ins*,*outs*) ∈ *set procs*› ‹*x'* < *length outs*› ‹*n'* = *Actual-out* (*targetnode a'*,*x'*)›
have *Formal-out* (*sourcenode a'*,*x'*) −*p*:*outs*!*x'*→$_{out}$ *n'*
by(*fastforce intro*:*SDG-param-out-edge*)
from ‹*matched n* [] *n*› ‹*n* −*p*:*ins*!*x*→$_{in}$ *Formal-in* (*targetnode a*,*x*)› ‹*matched* (*Formal-in* (*targetnode a*,*x*)) *ns* (*Formal-out* (*sourcenode a'*,*x'*))› ‹*Formal-out* (*sourcenode a'*,*x'*) −*p*:*outs*!*x'*→$_{out}$ *n'*› ‹*a'* ∈ *get-return-edges a*› ‹*n* = *Actual-in* (*sourcenode a*,*x*)› ‹*n'* = *Actual-out* (*targetnode a'*,*x'*)› ‹*valid-edge a*›
have *matched n* ([]@*n*#*ns*@[*Formal-out* (*sourcenode a'*,*x'*)]) *n'*
by(*fastforce intro*:*matched-bracket-param*)
with ‹*get-proc* (*sourcenode a'*) = *p*› **show** *?case* **by** *auto*
qed *simp-all*
qed

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(*atomize-elim*)
from ‹*valid-edge a*› ‹*kind a = Q↩pf*›
have *CFG-node* (*sourcenode a*) *s−p→ret CFG-node* (*targetnode a*)
by(*fastforce intro*:*sum-SDG-return-edge*)
from ‹*valid-edge a*› ‹*kind a = Q↩pf*›
obtain *a' Q' r' fs'* **where** *valid-edge a'* **and** *kind a' = Q':r'↪pfs'*
and *a ∈ get-return-edges a'* **by**(*blast dest*:*return-needs-call*)
hence *CFG-node* (*sourcenode a'*) *s−p→call CFG-node* (*targetnode a'*)
by(*fastforce intro*:*sum-SDG-call-edge*)
from ‹*valid-edge a'*› ‹*kind a' = Q':r'↪pfs'*› ‹*valid-edge a*› ‹*a ∈ get-return-edges a'*›
have *CFG-node* (*targetnode a'*) *⟶cd CFG-node* (*sourcenode a*)
by(*fastforce intro*!:*SDG-proc-entry-exit-cdep*)
hence *valid-SDG-node* (*CFG-node* (*targetnode a'*))
by(*rule SDG-edge-valid-SDG-node*)
with ‹*CFG-node* (*targetnode a'*) *⟶cd CFG-node* (*sourcenode a*)›
have *CFG-node* (*targetnode a'*) *i−[]@[CFG-node* (*targetnode a'*)*]→d** *CFG-node* (*sourcenode a*)
by(*fastforce intro*:*iSp-Append-cdep iSp-Nil*)
from ‹*valid-SDG-node* (*CFG-node* (*targetnode a'*))›
have *matched* (*CFG-node* (*targetnode a'*)) [] (*CFG-node* (*targetnode a'*))
by(*rule matched-Nil*)
with ‹*CFG-node* (*targetnode a'*) *i−[]@[CFG-node* (*targetnode a'*)*]→d** *CFG-node* (*sourcenode a*)›
have *matched* (*CFG-node* (*targetnode a'*)) ([]@[*CFG-node* (*targetnode a'*)]) (*CFG-node* (*sourcenode a*))
by(*fastforce intro*:*matched-Append-intra-SDG-path*)
with ‹*valid-edge a'*› ‹*kind a' = Q':r'↪pfs'*› ‹*valid-edge a*› ‹*kind a = Q↩pf*›
‹*a ∈ get-return-edges a'*›
have *CFG-node* (*sourcenode a'*) *s−p→sum CFG-node* (*targetnode a*)
by(*fastforce intro*!:*sum-SDG-call-summary-edge*)
with ‹*a ∈ get-return-edges a'*› ‹*valid-edge a'*› ‹*kind a' = Q':r'↪pfs'*›
show ∃ *a' Q' r' fs'*. *a ∈ get-return-edges a'* ∧ *valid-edge a'* ∧ *kind a' = Q':r'↪pfs'* ∧ *CFG-node* (*sourcenode a'*) *s−p→sum CFG-node* (*targetnode a*)
by *fastforce*
qed

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*:

$[\![n\ is{-}ns{\to_d}*\ n''; n''\ s{\longrightarrow_{cd}}\ n']\!] \Longrightarrow n\ is{-}ns@[n'']{\to_d}*\ n'$

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

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

lemma *is-SDG-path-Append*:
$[\![n''\ is{-}ns'{\to_d}*\ n'; n\ is{-}ns{\to_d}*\ n'']\!] \Longrightarrow n\ is{-}ns@ns'{\to_d}*\ n'$
by(*induct rule*:*intra-sum-SDG-path.induct*,
auto intro:*intra-sum-SDG-path.intros simp*:*append-assoc*[*THEN sym*]
simp del:*append-assoc*)

lemma *is-SDG-path-valid-SDG-node*:
assumes $n\ is{-}ns{\to_d}*\ n'$ **shows** *valid-SDG-node n* **and** *valid-SDG-node n'*
using $\langle n\ is{-}ns{\to_d}*\ n'\rangle$
by(*induct rule*:*intra-sum-SDG-path.induct*,
auto intro:*sum-SDG-edge-valid-SDG-node valid-SDG-CFG-node*)

lemma *intra-SDG-path-is-SDG-path*:
$n\ i{-}ns{\to_d}*\ n' \Longrightarrow n\ is{-}ns{\to_d}*\ n'$
by(*induct rule*:*intra-SDG-path.induct*,
auto intro:*intra-sum-SDG-path.intros SDG-edge-sum-SDG-edge*)

lemma *is-SDG-path-hd*:$[\![n\ is{-}ns{\to_d}*\ n'; ns \neq []]\!] \Longrightarrow hd\ ns = n$
apply(*induct rule*:*intra-sum-SDG-path.induct*) **apply** *clarsimp*
by(*case-tac ns*,*auto elim*:*intra-sum-SDG-path.cases*)+

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-SDG-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 $P\ n\ ns\ n'$
using $\langle n\ is{-}ns{\to_d}*\ n'\rangle$
proof(*induct ns arbitrary*:*n*)
case *Nil* **thus** *?case* **by**(*fastforce elim*:*intra-sum-SDG-path.cases intro*:*refl*)
next

case (*Cons nx nsx*)
note *IH* = ⟨⋀*n*. *n is−nsx→*$_{d}$** n′* ⟹ *P n nsx n′*⟩
from ⟨*n is−nx#nsx→*$_{d}$** n′*⟩ **have** [*simp*]:*n* = *nx*
by(*fastforce dest*:*is-SDG-path-hd*)
from ⟨*n is−nx#nsx→*$_{d}$** n′*⟩ **have** ((∃ *n″*. *n s⟶*$_{cd}$ *n″* ∧ *n″ is−nsx→*$_{d}$** n′*) ∨
(∃ *n″ V*. *n s−V→*$_{dd}$ *n″* ∧ *n* ≠ *n″* ∧ *n″ is−nsx→*$_{d}$** n′*)) ∨
(∃ *n″ p*. *n s−p→*$_{sum}$ *n″* ∧ *n″ is−nsx→*$_{d}$** n′*)
proof(*induct nsx arbitrary*:*n′ rule*:*rev-induct*)
case *Nil*
from ⟨*n is−*[*nx*]*→*$_{d}$** n′*⟩ **have** *n is−*[]*→*$_{d}$** nx*
and *disj*:*nx s⟶*$_{cd}$ *n′* ∨ (∃ *V*. *nx s−V→*$_{dd}$ *n′* ∧ *nx* ≠ *n′*) ∨ (∃ *p*. *nx s−p→*$_{sum}$ *n′*)
by(*induct n ns*≡[*nx*] *n′ rule*:*intra-sum-SDG-path.induct*,*auto*)
from ⟨*n is−*[]*→*$_{d}$** nx*⟩ **have** [*simp*]:*n* = *nx*
by(*fastforce elim*:*intra-sum-SDG-path.cases*)
from *disj* **have** *valid-SDG-node n′* **by**(*fastforce intro*:*sum-SDG-edge-valid-SDG-node*)
hence *n′ is−*[]*→*$_{d}$** n′* **by**(*rule isSp-Nil*)
with *disj* **show** *?case* **by** *fastforce*
next
case (*snoc x xs*)
note ⟨⋀*n′*. *n is−nx # xs→*$_{d}$** n′* ⟹
(∃ *n″*. *n s⟶*$_{cd}$ *n″* ∧ *n″ is−xs→*$_{d}$** n′*) ∨
(∃ *n″ V*. *n s−V→*$_{dd}$ *n″* ∧ *n* ≠ *n″* ∧ *n″ is−xs→*$_{d}$** n′*)) ∨
(∃ *n″ p*. *n s−p→*$_{sum}$ *n″* ∧ *n″ is−xs→*$_{d}$** n′*)⟩
with ⟨*n is−nx#xs@*[*x*]*→*$_{d}$** n′*⟩ **show** *?case*
proof(*induct n nx#xs@*[*x*] *n′ rule*:*intra-sum-SDG-path.induct*)
case (*isSp-Append-cdep m ms m″ n′*)
note *IH* = ⟨⋀*n′*. *m is−nx # xs→*$_{d}$** n′* ⟹
(∃ *n″*. *m s⟶*$_{cd}$ *n″* ∧ *n″ is−xs→*$_{d}$** n′*) ∨
(∃ *n″ V*. *m s−V→*$_{dd}$ *n″* ∧ *m* ≠ *n″* ∧ *n″ is−xs→*$_{d}$** n′*)) ∨
(∃ *n″ p*. *m s−p→*$_{sum}$ *n″* ∧ *n″ is−xs→*$_{d}$** n′*)⟩
from ⟨*ms* @ [*m″*] = *nx#xs@*[*x*]⟩ **have** [*simp*]:*ms* = *nx#xs*
and [*simp*]:*m″* = *x* **by** *simp-all*
from ⟨*m is−ms→*$_{d}$** m″*⟩ **have** *m is−nx#xs→*$_{d}$** m″* **by** *simp*
from *IH*[*OF this*] **obtain** *n″* **where** *n″ is−xs→*$_{d}$** m″*
and (*m s⟶*$_{cd}$ *n″* ∨ (∃ *V*. *m s−V→*$_{dd}$ *n″* ∧ *m* ≠ *n″*)) ∨ (∃ *p*. *m s−p→*$_{sum}$ *n″*)
by *fastforce*
from ⟨*n″ is−xs→*$_{d}$** m″*⟩ ⟨*m″ s⟶*$_{cd}$ *n′*⟩
have *n″ is−xs@*[*m″*]*→*$_{d}$** n′* **by**(*rule intra-sum-SDG-path.intros*)
with ⟨(*m s⟶*$_{cd}$ *n″* ∨ (∃ *V*. *m s−V→*$_{dd}$ *n″* ∧ *m* ≠ *n″*)) ∨ (∃ *p*. *m s−p→*$_{sum}$ *n″*)⟩
show *?case* **by** *fastforce*
next
case (*isSp-Append-ddep m ms m″ V n′*)
note *IH* = ⟨⋀*n′*. *m is−nx # xs→*$_{d}$** n′* ⟹
(∃ *n″*. *m s⟶*$_{cd}$ *n″* ∧ *n″ is−xs→*$_{d}$** n′*) ∨
(∃ *n″ V*. *m s−V→*$_{dd}$ *n″* ∧ *m* ≠ *n″* ∧ *n″ is−xs→*$_{d}$** n′*)) ∨
(∃ *n″ p*. *m s−p→*$_{sum}$ *n″* ∧ *n″ is−xs→*$_{d}$** n′*)⟩

from ‹ms @ $[m'] = nx\#xs@[x]$› **have** [*simp*]:$ms = nx\#xs$
and [*simp*]:$m'' = x$ **by** *simp-all*
from ‹m $is{-}ms{\rightarrow_d}*$ m''› **have** m $is{-}nx\#xs{\rightarrow_d}*$ m'' **by** *simp*
from *IH*[*OF this*] **obtain** n'' **where** n'' $is{-}xs{\rightarrow_d}*$ m''
and (m $s{\longrightarrow_{cd}}$ $n'' \vee (\exists\, V.\ m\ s{-}V{\rightarrow_{dd}}\ n'' \wedge m \neq n'')) \vee (\exists\, p.\ m\ s{-}p{\rightarrow_{sum}}$ $n'')$
by *fastforce*
from ‹n'' $is{-}xs{\rightarrow_d}*$ m''› ‹m'' $s{-}V{\rightarrow_{dd}}$ n'› ‹$m'' \neq n'$›
have n'' $is{-}xs@[m']{\rightarrow_d}*$ n' **by**(*rule intra-sum-SDG-path.intros*)
with ‹(m $s{\longrightarrow_{cd}}$ $n'' \vee (\exists\, V.\ m\ s{-}V{\rightarrow_{dd}}\ n'' \wedge m \neq n'')) \vee (\exists\, p.\ m\ s{-}p{\rightarrow_{sum}}$ $n'')$›
show *?case* **by** *fastforce*
next
case (*isSp-Append-sum* m ms m'' p n')
note *IH* = ‹$\bigwedge n'.\ m$ $is{-}nx$ # $xs{\rightarrow_d}*$ $n' \Longrightarrow$
($(\exists\, n''.\ m\ s{\longrightarrow_{cd}}\ n'' \wedge n''$ $is{-}xs{\rightarrow_d}*$ $n')\ \vee$
$(\exists\, n''\ V.\ m\ s{-}V{\rightarrow_{dd}}\ n'' \wedge m \neq n'' \wedge n''$ $is{-}xs{\rightarrow_d}*$ $n')) \vee$
$(\exists\, n''\ p.\ m\ s{-}p{\rightarrow_{sum}}\ n'' \wedge n''$ $is{-}xs{\rightarrow_d}*$ $n')$›
from ‹ms @ $[m'] = nx\#xs@[x]$› **have** [*simp*]:$ms = nx\#xs$
and [*simp*]:$m'' = x$ **by** *simp-all*
from ‹m $is{-}ms{\rightarrow_d}*$ m''› **have** m $is{-}nx\#xs{\rightarrow_d}*$ m'' **by** *simp*
from *IH*[*OF this*] **obtain** n'' **where** n'' $is{-}xs{\rightarrow_d}*$ m''
and (m $s{\longrightarrow_{cd}}$ $n'' \vee (\exists\, V.\ m\ s{-}V{\rightarrow_{dd}}\ n'' \wedge m \neq n'')) \vee (\exists\, p.\ m\ s{-}p{\rightarrow_{sum}}$ $n'')$
by *fastforce*
from ‹n'' $is{-}xs{\rightarrow_d}*$ m''› ‹m'' $s{-}p{\rightarrow_{sum}}$ n'›
have n'' $is{-}xs@[m']{\rightarrow_d}*$ n' **by**(*rule intra-sum-SDG-path.intros*)
with ‹(m $s{\longrightarrow_{cd}}$ $n'' \vee (\exists\, V.\ m\ s{-}V{\rightarrow_{dd}}\ n'' \wedge m \neq n'')) \vee (\exists\, p.\ m\ s{-}p{\rightarrow_{sum}}$ $n'')$›
show *?case* **by** *fastforce*
qed
qed
thus *?case* **apply** −
proof(*erule disjE*)+
assume $\exists\, n''.\ n\ s{\longrightarrow_{cd}}\ n'' \wedge n''$ $is{-}nsx{\rightarrow_d}*$ n'
then obtain n'' **where** $n\ s{\longrightarrow_{cd}}\ n''$ **and** n'' $is{-}nsx{\rightarrow_d}*$ n' **by** *blast*
from *IH*[*OF* ‹n'' $is{-}nsx{\rightarrow_d}*$ n'›] **have** $P\ n''\ nsx\ n'$ **.**
from *step-cdep*[*OF* ‹$n\ s{\longrightarrow_{cd}}\ n''$› ‹$n''$ $is{-}nsx{\rightarrow_d}*$ n'› *this*] **show** *?thesis* **by** *simp*
next
assume $\exists\, n''\ V.\ n\ s{-}V{\rightarrow_{dd}}\ n'' \wedge n \neq n'' \wedge n''$ $is{-}nsx{\rightarrow_d}*$ n'
then obtain $n''\ V$ **where** $n\ s{-}V{\rightarrow_{dd}}\ n''$ **and** $n \neq n''$ **and** n'' $is{-}nsx{\rightarrow_d}*$ n'
by *blast*
from *IH*[*OF* ‹n'' $is{-}nsx{\rightarrow_d}*$ n'›] **have** $P\ n''\ nsx\ n'$ **.**
from *step-ddep*[*OF* ‹$n\ s{-}V{\rightarrow_{dd}}\ n''$› ‹$n \neq n''$› ‹$n''$ $is{-}nsx{\rightarrow_d}*$ n'› *this*]
show *?thesis* **by** *simp*
next
assume $\exists\, n''\ p.\ n\ s{-}p{\rightarrow_{sum}}\ n'' \wedge n''$ $is{-}nsx{\rightarrow_d}*$ n'

```
    then obtain n'' p where n s−p→sum n'' and n'' is−nsx→d* n' by blast
    from IH[OF ⟨n'' is−nsx→d* n'⟩] have P n'' nsx n' .
     from step-sum[OF ⟨n s−p→sum n''⟩ ⟨n'' is−nsx→d* n'⟩ this] show ?thesis
by simp
  qed
qed


lemma is-SDG-path-CFG-path:
  assumes n is−ns→d* n'
  obtains as where parent-node n −as→ι* parent-node n'
proof(atomize-elim)
  from ⟨n is−ns→d* n'⟩
  show ∃ as. parent-node n −as→ι* parent-node n'
  proof(induct rule:intra-sum-SDG-path.induct)
    case (isSp-Nil n)
    from ⟨valid-SDG-node n⟩ have valid-node (parent-node n)
      by(rule valid-SDG-CFG-node)
    hence parent-node n −[]→* parent-node n by(rule empty-path)
    thus ?case by(auto simp:intra-path-def)
  next
    case (isSp-Append-cdep n ns n'' n')
    from ⟨∃ as. parent-node n −as→ι* parent-node n''⟩
    obtain as where parent-node n −as→ι* parent-node n'' by blast
    from ⟨n'' s⟶cd n'⟩ have n'' ⟶cd n' by(rule sum-SDG-edge-SDG-edge)
    thus ?case
    proof(rule cdep-edge-cases)
      assume parent-node n'' controls parent-node n'
      then obtain as' where parent-node n'' −as'→ι* parent-node n' and as' ≠
[]
        by(erule control-dependence-path)
      with ⟨parent-node n −as→ι* parent-node n''⟩
     have parent-node n −as@as'→ι* parent-node n' by −(rule intra-path-Append)
      thus ?thesis by blast
    next
      fix a Q r p fs a'
      assume valid-edge a and kind a = Q:r↪pfs and a' ∈ get-return-edges a
        and parent-node n'' = targetnode a and parent-node n' = sourcenode a'
      then obtain a'' where valid-edge a'' and sourcenode a'' = targetnode a
        and targetnode a'' = sourcenode a' and kind a'' = (λcf. False)√
        by(auto dest:intra-proc-additional-edge)
      hence targetnode a −[a'']→ι* sourcenode a'
        by(fastforce dest:path-edge simp:intra-path-def intra-kind-def)
      with ⟨parent-node n'' = targetnode a⟩ ⟨parent-node n' = sourcenode a'⟩
      have ∃ as'. parent-node n'' −as'→ι* parent-node n' ∧ as' ≠ [] by fastforce
      then obtain as' where parent-node n'' −as'→ι* parent-node n' and as' ≠
[]
        by blast
      with ⟨parent-node n −as→ι* parent-node n''⟩
```

have *parent-node n −as@as′→ι* parent-node n′* **by** −(*rule intra-path-Append*)
thus *?thesis* **by** *blast*
next
fix *m* **assume** *n″ = CFG-node m* **and** *m = parent-node n′*
with ‹*parent-node n −as→ι* parent-node n″*› **show** *?thesis* **by** *fastforce*
qed
next
case (*isSp-Append-ddep n ns n″ V n′*)
from ‹∃ *as*. *parent-node n −as→ι* parent-node n″*›
obtain *as* **where** *parent-node n −as→ι* parent-node n″* **by** *blast*
from ‹*n″ s−V→dd n′*› **have** *n″ influences V in n′*
by(*fastforce elim*:*sum-SDG-edge.cases*)
then obtain *as′* **where** *parent-node n″ −as′→ι* parent-node n′*
by(*auto simp*:*data-dependence-def*)
with ‹*parent-node n −as→ι* parent-node n″*›
have *parent-node n −as@as′→ι* parent-node n′* **by** −(*rule intra-path-Append*)
thus *?case* **by** *blast*
next
case (*isSp-Append-sum n ns n″ p n′*)
from ‹∃ *as*. *parent-node n −as→ι* parent-node n″*›
obtain *as* **where** *parent-node n −as→ι* parent-node n″* **by** *blast*
from ‹*n″ s−p→sum n′*› **have** ∃ *as′*. *parent-node n″ −as′→ι* parent-node n′*
proof(*rule sum-edge-cases*)
fix *a Q fs a′*
assume *valid-edge a* **and** *a′ ∈ get-return-edges a*
and *n″ = CFG-node* (*sourcenode a*) **and** *n′ = CFG-node* (*targetnode a′*)
from ‹*valid-edge a*› ‹*a′ ∈ get-return-edges a*›
obtain *a″* **where** *sourcenode a −[a″]→ι* targetnode a′*
apply − **apply**(*drule call-return-node-edge*)
apply(*auto simp*:*intra-path-def*) **apply**(*drule path-edge*)
by(*auto simp*:*intra-kind-def*)
with ‹*n″ = CFG-node* (*sourcenode a*)› ‹*n′ = CFG-node* (*targetnode a′*)›
show *?thesis* **by** *simp blast*
next
fix *a Q p fs a′ ns x x′ ins outs*
assume *valid-edge a* **and** *a′ ∈ get-return-edges a*
and *n″ = Actual-in* (*sourcenode a*, *x*)
and *n′ = Actual-out* (*targetnode a′*, *x′*)
from ‹*valid-edge a*› ‹*a′ ∈ get-return-edges a*›
obtain *a″* **where** *sourcenode a −[a″]→ι* targetnode a′*
apply − **apply**(*drule call-return-node-edge*)
apply(*auto simp*:*intra-path-def*) **apply**(*drule path-edge*)
by(*auto simp*:*intra-kind-def*)
with ‹*n″ = Actual-in* (*sourcenode a*, *x*)› ‹*n′ = Actual-out* (*targetnode a′*, *x′*)›
show *?thesis* **by** *simp blast*
qed
then obtain *as′* **where** *parent-node n″ −as′→ι* parent-node n′* **by** *blast*
with ‹*parent-node n −as→ι* parent-node n″*›
have *parent-node n −as@as′→ι* parent-node n′* **by** −(*rule intra-path-Append*)

thus *?case* **by** *blast*
qed
qed

lemma *matched-is-SDG-path*:
assumes *matched n ns n'* **obtains** *ns'* **where** $n\ is{-}ns'{\to_d}*\ n'$
proof(*atomize-elim*)
from ‹*matched n ns n'*› **show** $\exists\, ns'.\ n\ is{-}ns'{\to_d}*\ n'$
proof(*induct rule*:*matched.induct*)
case *matched-Nil* **thus** *?case* **by**(*fastforce intro*:*isSp-Nil*)
next
case *matched-Append-intra-SDG-path* **thus** *?case*
by(*fastforce intro*:*is-SDG-path-Append intra-SDG-path-is-SDG-path*)
next
case (*matched-bracket-call* n_0 *ns* n_1 *p* n_2 *ns'* n_3 n_4 *V a a'*)
from ‹$\exists\, ns'.\ n_0\ is{-}ns'{\to_d}*\ n_1$› **obtain** *nsx* **where** $n_0\ is{-}nsx{\to_d}*\ n_1$ **by** *blast*
from ‹$n_1\ -p{\to_{call}}\ n_2$› ‹*sourcenode a* = *parent-node* n_1› ‹*targetnode a* = *parent-node* n_2›
have n_1 = *CFG-node* (*sourcenode a*) **and** n_2 = *CFG-node* (*targetnode a*)
by(*auto elim*:*SDG-edge.cases*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
obtain *Q r p' fs* **where** *kind a* = $Q{:}r{\hookrightarrow_{p'}}fs$
by(*fastforce dest*!:*only-call-get-return-edges*)
with ‹$n_1\ -p{\to_{call}}\ n_2$› ‹*valid-edge a*›
‹n_1 = *CFG-node* (*sourcenode a*)› ‹n_2 = *CFG-node* (*targetnode a*)›
have [*simp*]:p' = *p* **by** −(*erule SDG-edge.cases*,(*fastforce dest*:*edge-det*)+)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹$n_3\ -p{\to_{ret}}\ n_4 \vee n_3\ -p{:}V{\to_{out}}\ n_4$› **show** *?case*
proof
assume $n_3\ -p{\to_{ret}}\ n_4$
then obtain *ax Q' f'* **where** *valid-edge ax* **and** *kind ax* = $Q'{\hookleftarrow_p}f'$
and n_3 = *CFG-node* (*sourcenode ax*) **and** n_4 = *CFG-node* (*targetnode ax*)
by(*fastforce elim*:*SDG-edge.cases*)
with ‹*sourcenode a'* = *parent-node* n_3› ‹*targetnode a'* = *parent-node* n_4›
‹*valid-edge a'*› **have** [*simp*]:*ax* = *a'* **by**(*fastforce dest*:*edge-det*)
from ‹*valid-edge a*› ‹*kind a* = $Q{:}r{\hookrightarrow_{p'}}fs$› ‹*valid-edge ax*› ‹*kind ax* = $Q'{\hookleftarrow_p}f'$›
‹*a'* ∈ *get-return-edges a*› ‹*matched* n_2 *ns'* n_3›
‹n_1 = *CFG-node* (*sourcenode a*)› ‹n_2 = *CFG-node* (*targetnode a*)›
‹n_3 = *CFG-node* (*sourcenode ax*)› ‹n_4 = *CFG-node* (*targetnode ax*)›
have $n_1\ s{-}p{\to_{sum}}\ n_4$
by(*fastforce intro*!:*sum-SDG-call-summary-edge*[*of a - - - - ax*])
with ‹$n_0\ is{-}nsx{\to_d}*\ n_1$› **have** $n_0\ is{-}nsx@[n_1]{\to_d}*\ n_4$ **by**(*rule isSp-Append-sum*)
thus *?case* **by** *blast*
next
assume $n_3\ -p{:}V{\to_{out}}\ n_4$
then obtain *ax Q' f' x* **where** *valid-edge ax* **and** *kind ax* = $Q'{\hookleftarrow_p}f'$
and n_3 = *Formal-out* (*sourcenode ax*,*x*)

and n_4 = *Actual-out* (*targetnode ax*,*x*)
by(*fastforce elim*:*SDG-edge.cases*)
with ‹*sourcenode a'* = *parent-node* n_3› ‹*targetnode a'* = *parent-node* n_4›
‹*valid-edge a'*› **have** [*simp*]:*ax* = *a'* **by**(*fastforce dest*:*edge-det*)
from ‹*valid-edge ax*› ‹*kind ax* = $Q'\hookleftarrow_{pf'}$› ‹n_3 = *Formal-out* (*sourcenode ax*,*x*)›
‹n_4 = *Actual-out* (*targetnode ax*,*x*)›
have *CFG-node* (*sourcenode a'*) $-p\rightarrow_{ret}$ *CFG-node* (*targetnode a'*)
by(*fastforce intro*:*SDG-return-edge*)
from ‹*valid-edge a*› ‹*kind a* = Q:$r\hookrightarrow_{p'fs}$› ‹*valid-edge a'*›
‹*a'* ∈ *get-return-edges a*› ‹n_4 = *Actual-out* (*targetnode ax*,*x*)›
have *CFG-node* (*targetnode a*) $\longrightarrow_{cd}$ *CFG-node* (*sourcenode a'*)
by(*fastforce intro*!:*SDG-proc-entry-exit-cdep*)
with ‹n_2 = *CFG-node* (*targetnode a*)›
have *matched* n_2 ([]@([]@[n_2])) (*CFG-node* (*sourcenode a'*))
by(*fastforce intro*:*matched.intros intra-SDG-path.intros*
SDG-edge-valid-SDG-node)
with ‹*valid-edge a*› ‹*kind a* = Q:$r\hookrightarrow_{p'fs}$› ‹*valid-edge a'*› ‹*kind ax* = $Q'\hookleftarrow_{pf'}$›
‹*a'* ∈ *get-return-edges a*› ‹n_1 = *CFG-node* (*sourcenode a*)›
‹n_2 = *CFG-node* (*targetnode a*)› ‹n_4 = *Actual-out* (*targetnode ax*,*x*)›
have n_1 $s-p\rightarrow_{sum}$ *CFG-node* (*targetnode a'*)
by(*fastforce intro*!:*sum-SDG-call-summary-edge*[*of a - - - - a'*])
with ‹n_0 *is*$-nsx\rightarrow_{d*}$ n_1› **have** n_0 *is*$-nsx@[n_1]\rightarrow_{d*}$ *CFG-node* (*targetnode a'*)
by(*rule isSp-Append-sum*)
from ‹n_4 = *Actual-out* (*targetnode ax*,*x*)› ‹n_3 $-p$:$V\rightarrow_{out}$ n_4›
have *CFG-node* (*targetnode a'*) $s\longrightarrow_{cd}$ n_4
by(*fastforce intro*:*sum-SDG-parent-cdep-edge SDG-edge-valid-SDG-node*)
with ‹n_0 *is*$-nsx@[n_1]\rightarrow_{d*}$ *CFG-node* (*targetnode a'*)›
have n_0 *is*$-(nsx@[n_1])@[$*CFG-node* (*targetnode a'*)$]\rightarrow_{d*}$ n_4
by(*rule isSp-Append-cdep*)
thus *?case* **by** *blast*
qed
next
case (*matched-bracket-param* n_0 *ns* n_1 *p* *V* n_2 *ns'* n_3 *V'* n_4 *a a'*)
from ‹∃*ns'*. n_0 *is*$-ns'\rightarrow_{d*}$ n_1› **obtain** *nsx* **where** n_0 *is*$-nsx\rightarrow_{d*}$ n_1 **by** *blast*
from ‹n_1 $-p$:$V\rightarrow_{in}$ n_2› ‹*sourcenode a* = *parent-node* n_1›
‹*targetnode a* = *parent-node* n_2› **obtain** *x ins outs*
where n_1 = *Actual-in* (*sourcenode a*,*x*) **and** n_2 = *Formal-in* (*targetnode a*,*x*)
and (*p*,*ins*,*outs*) ∈ *set procs* **and** *V* = *ins*!*x* **and** *x* < *length ins*
by(*fastforce elim*:*SDG-edge.cases*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
obtain *Q r p' fs* **where** *kind a* = Q:$r\hookrightarrow_{p'fs}$
by(*fastforce dest*!:*only-call-get-return-edges*)
with ‹n_1 $-p$:$V\rightarrow_{in}$ n_2› ‹*valid-edge a*›
‹n_1 = *Actual-in* (*sourcenode a*,*x*)› ‹n_2 = *Formal-in* (*targetnode a*,*x*)›
have [*simp*]:*p'* = *p* **by** −(*erule SDG-edge.cases*,(*fastforce dest*:*edge-det*)+)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹n_3 $-p$:$V'\rightarrow_{out}$ n_4› **obtain** *ax Q' f' x' ins' outs'* **where** *valid-edge ax*

and *kind ax* = $Q' \hookleftarrow_{pf}$ *f'* **and** n_3 = *Formal-out* (*sourcenode ax*,*x'*)
and n_4 = *Actual-out* (*targetnode ax*,*x'*) **and** (*p*,*ins'*,*outs'*) ∈ *set procs*
and *V'* = *outs'*!*x'* **and** *x'* < *length outs'*
by(*fastforce elim*:*SDG-edge.cases*)
with ‹*sourcenode a'* = *parent-node* n_3› ‹*targetnode a'* = *parent-node* n_4›
‹*valid-edge a'*› **have** [*simp*]:*ax* = *a'* **by**(*fastforce dest*:*edge-det*)
from *unique-callers* ‹(*p*,*ins*,*outs*) ∈ *set procs*› ‹(*p*,*ins'*,*outs'*) ∈ *set procs*›
have [*simp*]:*ins* = *ins' outs* = *outs'*
by(*auto dest*:*distinct-fst-isin-same-fst*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_{p'}$*fs*› ‹*valid-edge a'*› ‹*kind ax* = $Q' \hookleftarrow_{pf}$ *f'*›
‹*a'* ∈ *get-return-edges a*› ‹*matched* n_2 *ns'* n_3› ‹n_1 = *Actual-in* (*sourcenode a*,*x*)›
‹n_2 = *Formal-in* (*targetnode a*,*x*)› ‹n_3 = *Formal-out* (*sourcenode ax*,*x'*)›
‹n_4 = *Actual-out* (*targetnode ax*,*x'*)› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
‹*x* < *length ins*› ‹*x'* < *length outs'*› ‹*V* = *ins*!*x*› ‹*V'* = *outs'*!*x'*›
have n_1 *s*−*p*→$_{sum}$ n_4
by(*fastforce intro*!:*sum-SDG-param-summary-edge*[*of a - - - - a'*])
with ‹n_0 *is*−*nsx*→$_{d*}$ n_1› **have** n_0 *is*−*nsx*@[n_1]→$_{d*}$ n_4 **by**(*rule isSp-Append-sum*)
thus *?case* **by** *blast*
qed
qed

lemma *is-SDG-path-matched*:
assumes *n is*−*ns*→$_{d*}$ *n'* **obtains** *ns'* **where** *matched n ns' n'* **and** *set ns* ⊆ *set ns'*
proof(*atomize-elim*)
from ‹*n is*−*ns*→$_{d*}$ *n'*› **show** ∃ *ns'*. *matched n ns' n'* ∧ *set ns* ⊆ *set ns'*
proof(*induct rule*:*intra-sum-SDG-path.induct*)
case (*isSp-Nil n*)
from ‹*valid-SDG-node n*› **have** *matched n* [] *n* **by**(*rule matched-Nil*)
thus *?case* **by** *fastforce*
next
case (*isSp-Append-cdep n ns n'' n'*)
from ‹∃ *ns'*. *matched n ns' n''* ∧ *set ns* ⊆ *set ns'*›
obtain *ns'* **where** *matched n ns' n''* **and** *set ns* ⊆ *set ns'* **by** *blast*
from ‹*n''* *s*⟶$_{cd}$ *n'*› **have** *n''* *i*−[]@[*n'*]→$_{d*}$ *n'*
by(*fastforce intro*:*intra-SDG-path.intros sum-SDG-edge-valid-SDG-node*
sum-SDG-edge-SDG-edge)
with ‹*matched n ns' n''*› **have** *matched n* (*ns'*@[*n'*]) *n'*
by(*fastforce intro*!:*matched-Append-intra-SDG-path*)
with ‹*set ns* ⊆ *set ns'*› **show** *?case* **by** *fastforce*
next
case (*isSp-Append-ddep n ns n'' V n'*)
from ‹∃ *ns'*. *matched n ns' n''* ∧ *set ns* ⊆ *set ns'*›
obtain *ns'* **where** *matched n ns' n''* **and** *set ns* ⊆ *set ns'* **by** *blast*
from ‹*n''* *s*−*V*→$_{dd}$ *n'*› ‹*n''* ≠ *n'*› **have** *n''* *i*−[]@[*n'*]→$_{d*}$ *n'*
by(*fastforce intro*:*intra-SDG-path.intros sum-SDG-edge-valid-SDG-node*
sum-SDG-edge-SDG-edge)

with ‹*matched n ns' n''*› **have** *matched n (ns'@[n'']) n'*
by(*fastforce intro*!:*matched-Append-intra-SDG-path*)
with ‹*set ns ⊆ set ns'*› **show** *?case* **by** *fastforce*
next
case (*isSp-Append-sum n ns n'' p n'*)
from ‹∃ *ns'*. *matched n ns' n''* ∧ *set ns ⊆ set ns'*›
obtain *ns'* **where** *matched n ns' n''* **and** *set ns ⊆ set ns'* **by** *blast*
from ‹*n'' s−p→sum n'*› **obtain** *ns''* **where** *matched n'' ns'' n'* **and** *n'' ∈ set ns''*
by −(*erule sum-SDG-summary-edge-matched*)
with ‹*matched n ns' n''*› **have** *matched n (ns'@ns'') n'* **by** −(*rule matched-Append*)
with ‹*set ns ⊆ set ns'*› ‹*n'' ∈ set ns''*› **show** *?case* **by** *fastforce*
qed
qed

lemma *is-SDG-path-intra-CFG-path*:
assumes *n is−ns→d* n'*
obtains *as* **where** *parent-node n −as→ι* parent-node n'*
proof(*atomize-elim*)
from ‹*n is−ns→d* n'*›
show ∃ *as*. *parent-node n −as→ι* parent-node n'*
proof(*induct rule*:*intra-sum-SDG-path.induct*)
case (*isSp-Nil n*)
from ‹*valid-SDG-node n*› **have** *parent-node n −[]→* parent-node n*
by(*fastforce intro*:*empty-path valid-SDG-CFG-node*)
thus *?case* **by**(*auto simp*:*intra-path-def*)
next
case (*isSp-Append-cdep n ns n'' n'*)
from ‹∃ *as*. *parent-node n −as→ι* parent-node n''*›
obtain *as* **where** *parent-node n −as→ι* parent-node n''* **by** *blast*
from ‹*n'' s⟶cd n'*› **have** *n'' ⟶cd n'* **by**(*rule sum-SDG-edge-SDG-edge*)
thus *?case*
proof(*rule cdep-edge-cases*)
assume *parent-node n'' controls parent-node n'*
then obtain *as'* **where** *parent-node n'' −as'→ι* parent-node n'* **and** *as' ≠ []*
by(*erule control-dependence-path*)
with ‹*parent-node n −as→ι* parent-node n''*›
have *parent-node n −as@as'→ι* parent-node n'* **by** −(*rule intra-path-Append*)
thus *?thesis* **by** *blast*
next
fix *a Q r p fs a'*
assume *valid-edge a* **and** *kind a = Q:r↪pfs a'* ∈ *get-return-edges a*
and *parent-node n'' = targetnode a* **and** *parent-node n' = sourcenode a'*
then obtain *a''* **where** *valid-edge a''* **and** *sourcenode a'' = targetnode a*
and *targetnode a'' = sourcenode a'* **and** *kind a'' = (λcf. False)√*
by(*auto dest*:*intra-proc-additional-edge*)
hence *targetnode a −[a']→ι* sourcenode a'*

by(*fastforce dest*:*path-edge simp*:*intra-path-def intra-kind-def*)
with ‹*parent-node n'' = targetnode a*› ‹*parent-node n' = sourcenode a'*›
have ∃ *as'*. *parent-node n'' −as'→ι* parent-node n'* ∧ *as'* ≠ [] **by** *fastforce*
then obtain *as'* **where** *parent-node n'' −as'→ι* parent-node n'* **and** *as'* ≠ []
by *blast*
with ‹*parent-node n −as→ι* parent-node n''*›
have *parent-node n −as@as'→ι* parent-node n'* **by** −(*rule intra-path-Append*)
thus *?thesis* **by** *blast*
next
fix *m* **assume** *n'' = CFG-node m* **and** *m = parent-node n'*
with ‹*parent-node n −as→ι* parent-node n''*› **show** *?thesis* **by** *fastforce*
qed
next
case (*isSp-Append-ddep n ns n'' V n'*)
from ‹∃ *as*. *parent-node n −as→ι* parent-node n''*›
obtain *as* **where** *parent-node n −as→ι* parent-node n''* **by** *blast*
from ‹*n'' s−V→dd n'*› **have** *n'' influences V in n'*
by(*fastforce elim*:*sum-SDG-edge.cases*)
then obtain *as'* **where** *parent-node n'' −as'→ι* parent-node n'*
by(*auto simp*:*data-dependence-def*)
with ‹*parent-node n −as→ι* parent-node n''*›
have *parent-node n −as@as'→ι* parent-node n'* **by** −(*rule intra-path-Append*)
thus *?case* **by** *blast*
next
case (*isSp-Append-sum n ns n'' p n'*)
from ‹∃ *as*. *parent-node n −as→ι* parent-node n''*›
obtain *as* **where** *parent-node n −as→ι* parent-node n''* **by** *blast*
from ‹*n'' s−p→sum n'*› **obtain** *ns'* **where** *matched n'' ns' n'*
by −(*erule sum-SDG-summary-edge-matched*)
then obtain *as'* **where** *parent-node n'' −as'→ι* parent-node n'*
by(*erule matched-intra-CFG-path*)
with ‹*parent-node n −as→ι* parent-node n''*›
have *parent-node n −as@as'→ι* parent-node n'*
by(*fastforce intro*:*path-Append simp*:*intra-path-def*)
thus *?case* **by** *blast*
qed
qed

SDG paths without return edges

inductive *intra-call-sum-SDG-path* ::
'node SDG-node ⇒ *'node SDG-node list* ⇒ *'node SDG-node* ⇒ *bool*
(- *ics−−→d** - [*51*,*0*,*0*] *80*)
where *icsSp-Nil*:
valid-SDG-node n ⟹ *n ics−[]→d* n*

| *icsSp-Append-cdep*:
⟦*n ics−ns→d* n''*; *n'' s⟶cd n'*⟧ ⟹ *n ics−ns@[n']→d* n'*

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

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

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

| *icsSp-Append-param-in*:
⟦*n ics−ns→$_d$* n''*; *n'' s−p:V→$_{in}$ n'*⟧ ⟹ *n ics−ns@[n'']→$_d$* n'*

lemma *ics-SDG-path-valid-SDG-node*:
assumes *n ics−ns→$_d$* n'* **shows** *valid-SDG-node n* **and** *valid-SDG-node n'*
using ‹*n ics−ns→$_d$* n'*›
by(*induct rule*:*intra-call-sum-SDG-path.induct*,
auto intro:*sum-SDG-edge-valid-SDG-node valid-SDG-CFG-node*)

lemma *ics-SDG-path-Append*:
⟦*n'' ics−ns'→$_d$* n'*; *n ics−ns→$_d$* n''*⟧ ⟹ *n ics−ns@ns'→$_d$* n'*
by(*induct rule*:*intra-call-sum-SDG-path.induct*,
auto intro:*intra-call-sum-SDG-path.intros simp*:*append-assoc*[*THEN sym*]
simp del:*append-assoc*)

lemma *is-SDG-path-ics-SDG-path*:
n is−ns→$_d$ n'* ⟹ *n ics−ns→$_d$* n'*
by(*induct rule*:*intra-sum-SDG-path.induct*,*auto intro*:*intra-call-sum-SDG-path.intros*)

lemma *cc-SDG-path-ics-SDG-path*:
n cc−ns→$_d$ n'* ⟹ *n ics−ns→$_d$* n'*
by(*induct rule*:*call-cdep-SDG-path.induct*,
auto intro:*intra-call-sum-SDG-path.intros SDG-edge-sum-SDG-edge*)

lemma *ics-SDG-path-split*:
assumes *n ics−ns→$_d$* n'* **and** *n'' ∈ set ns*
obtains *ns' ns''* **where** *ns = ns'@ns''* **and** *n ics−ns'→$_d$* n''*
and *n'' ics−ns''→$_d$* n'*
proof(*atomize-elim*)
from ‹*n ics−ns→$_d$* n'*› ‹*n'' ∈ set ns*›
show ∃ *ns' ns''*. *ns = ns'@ns''* ∧ *n ics−ns'→$_d$* n''* ∧ *n'' ics−ns''→$_d$* n'*
proof(*induct rule*:*intra-call-sum-SDG-path.induct*)
case *icsSp-Nil* **thus** *?case* **by** *simp*
next
case (*icsSp-Append-cdep n ns nx n'*)

note $IH = \langle n'' \in set\ ns \Longrightarrow$
$\exists ns'\ ns''.\ ns = ns' @ ns'' \wedge n\ ics{-}ns'{\to_{d}*}\ n'' \wedge n''\ ics{-}ns''{\to_{d}*}\ nx\rangle$
from $\langle n'' \in set\ (ns@[nx])\rangle$ **have** $n'' \in set\ ns \vee n'' = nx$ **by** *fastforce*
thus *?case*
proof
assume $n'' \in set\ ns$
from $IH[OF\ this]$ **obtain** $ns'\ ns''$ **where** $ns = ns' @ ns''$
and $n\ ics{-}ns'{\to_{d}*}\ n''$ **and** $n''\ ics{-}ns''{\to_{d}*}\ nx$ **by** *blast*
from $\langle n''\ ics{-}ns''{\to_{d}*}\ nx\rangle\ \langle nx\ s{\longrightarrow_{cd}}\ n'\rangle$
have $n''\ ics{-}ns''@[nx]{\to_{d}*}\ n'$
by(*rule intra-call-sum-SDG-path.icsSp-Append-cdep*)
with $\langle ns = ns'@ns''\rangle\ \langle n\ ics{-}ns'{\to_{d}*}\ n''\rangle$ **show** *?thesis* **by** *fastforce*
next
assume $n'' = nx$
from $\langle nx\ s{\longrightarrow_{cd}}\ n'\rangle$ **have** $nx\ ics{-}[]{\to_{d}*}\ nx$
by(*fastforce intro:icsSp-Nil SDG-edge-valid-SDG-node sum-SDG-edge-SDG-edge*)
with $\langle nx\ s{\longrightarrow_{cd}}\ n'\rangle$ **have** $nx\ ics{-}[]@[nx]{\to_{d}*}\ n'$
by $-$(*rule intra-call-sum-SDG-path.icsSp-Append-cdep*)
with $\langle n\ ics{-}ns{\to_{d}*}\ nx\rangle\ \langle n'' = nx\rangle$ **show** *?thesis* **by** *fastforce*
qed
next
case (*icsSp-Append-ddep n ns nx V n'*)
note $IH = \langle n'' \in set\ ns \Longrightarrow$
$\exists ns'\ ns''.\ ns = ns' @ ns'' \wedge n\ ics{-}ns'{\to_{d}*}\ n'' \wedge n''\ ics{-}ns''{\to_{d}*}\ nx\rangle$
from $\langle n'' \in set\ (ns@[nx])\rangle$ **have** $n'' \in set\ ns \vee n'' = nx$ **by** *fastforce*
thus *?case*
proof
assume $n'' \in set\ ns$
from $IH[OF\ this]$ **obtain** $ns'\ ns''$ **where** $ns = ns' @ ns''$
and $n\ ics{-}ns'{\to_{d}*}\ n''$ **and** $n''\ ics{-}ns''{\to_{d}*}\ nx$ **by** *blast*
from $\langle n''\ ics{-}ns''{\to_{d}*}\ nx\rangle\ \langle nx\ s{-}V{\to_{dd}}\ n'\rangle\ \langle nx \neq n'\rangle$
have $n''\ ics{-}ns''@[nx]{\to_{d}*}\ n'$
by(*rule intra-call-sum-SDG-path.icsSp-Append-ddep*)
with $\langle ns = ns'@ns''\rangle\ \langle n\ ics{-}ns'{\to_{d}*}\ n''\rangle$ **show** *?thesis* **by** *fastforce*
next
assume $n'' = nx$
from $\langle nx\ s{-}V{\to_{dd}}\ n'\rangle$ **have** $nx\ ics{-}[]{\to_{d}*}\ nx$
by(*fastforce intro:icsSp-Nil SDG-edge-valid-SDG-node sum-SDG-edge-SDG-edge*)
with $\langle nx\ s{-}V{\to_{dd}}\ n'\rangle\ \langle nx \neq n'\rangle$ **have** $nx\ ics{-}[]@[nx]{\to_{d}*}\ n'$
by $-$(*rule intra-call-sum-SDG-path.icsSp-Append-ddep*)
with $\langle n\ ics{-}ns{\to_{d}*}\ nx\rangle\ \langle n'' = nx\rangle$ **show** *?thesis* **by** *fastforce*
qed
next
case (*icsSp-Append-sum n ns nx p n'*)
note $IH = \langle n'' \in set\ ns \Longrightarrow$
$\exists ns'\ ns''.\ ns = ns' @ ns'' \wedge n\ ics{-}ns'{\to_{d}*}\ n'' \wedge n''\ ics{-}ns''{\to_{d}*}\ nx\rangle$
from $\langle n'' \in set\ (ns@[nx])\rangle$ **have** $n'' \in set\ ns \vee n'' = nx$ **by** *fastforce*
thus *?case*
proof

assume $n'' \in set\ ns$
from *IH*[*OF this*] **obtain** $ns'\ ns''$ **where** $ns = ns' @ ns''$
and $n\ ics{-}ns'{\rightarrow_d}*\ n''$ **and** $n''\ ics{-}ns''{\rightarrow_d}*\ nx$ **by** *blast*
from $\langle n''\ ics{-}ns''{\rightarrow_d}*\ nx\rangle$ $\langle nx\ s{-}p{\rightarrow_{sum}}\ n'\rangle$
have $n''\ ics{-}ns''@[nx]{\rightarrow_d}*\ n'$
by(*rule intra-call-sum-SDG-path.icsSp-Append-sum*)
with $\langle ns = ns'@ns''\rangle$ $\langle n\ ics{-}ns'{\rightarrow_d}*\ n''\rangle$ **show** *?thesis* **by** *fastforce*
next
assume $n'' = nx$
from $\langle nx\ s{-}p{\rightarrow_{sum}}\ n'\rangle$ **have** *valid-SDG-node nx*
by(*fastforce elim*:*sum-SDG-edge.cases*)
hence $nx\ ics{-}[]{\rightarrow_d}*\ nx$ **by**(*fastforce intro*:*icsSp-Nil*)
with $\langle nx\ s{-}p{\rightarrow_{sum}}\ n'\rangle$ **have** $nx\ ics{-}[]@[nx]{\rightarrow_d}*\ n'$
by −(*rule intra-call-sum-SDG-path.icsSp-Append-sum*)
with $\langle n\ ics{-}ns{\rightarrow_d}*\ nx\rangle$ $\langle n'' = nx\rangle$ **show** *?thesis* **by** *fastforce*
qed
next
case (*icsSp-Append-call n ns nx p n'*)
note $IH = \langle n'' \in set\ ns \Longrightarrow$
$\exists ns'\ ns''.\ ns = ns' @ ns'' \wedge n\ ics{-}ns'{\rightarrow_d}*\ n'' \wedge n''\ ics{-}ns''{\rightarrow_d}*\ nx\rangle$
from $\langle n'' \in set\ (ns@[nx])\rangle$ **have** $n'' \in set\ ns \vee n'' = nx$ **by** *fastforce*
thus *?case*
proof
assume $n'' \in set\ ns$
from *IH*[*OF this*] **obtain** $ns'\ ns''$ **where** $ns = ns' @ ns''$
and $n\ ics{-}ns'{\rightarrow_d}*\ n''$ **and** $n''\ ics{-}ns''{\rightarrow_d}*\ nx$ **by** *blast*
from $\langle n''\ ics{-}ns''{\rightarrow_d}*\ nx\rangle$ $\langle nx\ s{-}p{\rightarrow_{call}}\ n'\rangle$
have $n''\ ics{-}ns''@[nx]{\rightarrow_d}*\ n'$
by(*rule intra-call-sum-SDG-path.icsSp-Append-call*)
with $\langle ns = ns'@ns''\rangle$ $\langle n\ ics{-}ns'{\rightarrow_d}*\ n''\rangle$ **show** *?thesis* **by** *fastforce*
next
assume $n'' = nx$
from $\langle nx\ s{-}p{\rightarrow_{call}}\ n'\rangle$ **have** $nx\ ics{-}[]{\rightarrow_d}*\ nx$
by(*fastforce intro*:*icsSp-Nil SDG-edge-valid-SDG-node sum-SDG-edge-SDG-edge*)
with $\langle nx\ s{-}p{\rightarrow_{call}}\ n'\rangle$ **have** $nx\ ics{-}[]@[nx]{\rightarrow_d}*\ n'$
by −(*rule intra-call-sum-SDG-path.icsSp-Append-call*)
with $\langle n\ ics{-}ns{\rightarrow_d}*\ nx\rangle$ $\langle n'' = nx\rangle$ **show** *?thesis* **by** *fastforce*
qed
next
case (*icsSp-Append-param-in n ns nx p V n'*)
note $IH = \langle n'' \in set\ ns \Longrightarrow$
$\exists ns'\ ns''.\ ns = ns' @ ns'' \wedge n\ ics{-}ns'{\rightarrow_d}*\ n'' \wedge n''\ ics{-}ns''{\rightarrow_d}*\ nx\rangle$
from $\langle n'' \in set\ (ns@[nx])\rangle$ **have** $n'' \in set\ ns \vee n'' = nx$ **by** *fastforce*
thus *?case*
proof
assume $n'' \in set\ ns$
from *IH*[*OF this*] **obtain** $ns'\ ns''$ **where** $ns = ns' @ ns''$
and $n\ ics{-}ns'{\rightarrow_d}*\ n''$ **and** $n''\ ics{-}ns''{\rightarrow_d}*\ nx$ **by** *blast*
from $\langle n''\ ics{-}ns''{\rightarrow_d}*\ nx\rangle$ $\langle nx\ s{-}p{:}V{\rightarrow_{in}}\ n'\rangle$

```
    have n'' ics−ns''@[nx]→d* n'
      by(rule intra-call-sum-SDG-path.icsSp-Append-param-in)
    with ⟨ns = ns'@ns''⟩ ⟨n ics−ns'→d* n''⟩ show ?thesis by fastforce
  next
    assume n'' = nx
    from ⟨nx s−p:V→in n'⟩ have nx ics−[]→d* nx
    by(fastforce intro:icsSp-Nil SDG-edge-valid-SDG-node sum-SDG-edge-SDG-edge)
    with ⟨nx s−p:V→in n'⟩ have nx ics−[]@[nx]→d* n'
      by −(rule intra-call-sum-SDG-path.icsSp-Append-param-in)
    with ⟨n ics−ns→d* nx⟩ ⟨n'' = nx⟩ show ?thesis by fastforce
  qed
 qed
qed


lemma realizable-ics-SDG-path:
 assumes realizable n ns n' obtains ns' where n ics−ns'→d* n'
proof(atomize-elim)
 from ⟨realizable n ns n'⟩ show ∃ns'. n ics−ns'→d* n'
 proof(induct rule:realizable.induct)
   case (realizable-matched n ns n')
   from ⟨matched n ns n'⟩ obtain ns' where n is−ns'→d* n'
     by(erule matched-is-SDG-path)
   thus ?case by(fastforce intro:is-SDG-path-ics-SDG-path)
 next
   case (realizable-call n0 ns n1 p n2 V ns' n3)
    from ⟨∃ns'. n0 ics−ns'→d* n1⟩ obtain nsx where n0 ics−nsx→d* n1 by
blast
   with ⟨n1 −p→call n2 ∨ n1 −p:V→in n2⟩ have n0 ics−nsx@[n1]→d* n2
   by(fastforce intro:SDG-edge-sum-SDG-edge icsSp-Append-call icsSp-Append-param-in)
   from ⟨matched n2 ns' n3⟩ obtain nsx' where n2 is−nsx'→d* n3
     by(erule matched-is-SDG-path)
   hence n2 ics−nsx'→d* n3 by(rule is-SDG-path-ics-SDG-path)
   from ⟨n2 ics−nsx'→d* n3⟩ ⟨n0 ics−nsx@[n1]→d* n2⟩
   have n0 ics−(nsx@[n1])@nsx'→d* n3 by(rule ics-SDG-path-Append)
   thus ?case by blast
 qed
qed


lemma ics-SDG-path-realizable:
 assumes n ics−ns→d* n'
 obtains ns' where realizable n ns' n' and set ns ⊆ set ns'
proof(atomize-elim)
 from ⟨n ics−ns→d* n'⟩ show ∃ns'. realizable n ns' n' ∧ set ns ⊆ set ns'
 proof(induct rule:intra-call-sum-SDG-path.induct)
   case (icsSp-Nil n)
   hence matched n [] n by(rule matched-Nil)
   thus ?case by(fastforce intro:realizable-matched)
```

next
case (*icsSp-Append-cdep n ns n'' n'*)
from ‹∃ *ns'*. *realizable n ns' n''* ∧ *set ns* ⊆ *set ns'*›
obtain *ns'* **where** *realizable n ns' n''* **and** *set ns* ⊆ *set ns'* **by** *blast*
from ‹*n'' s*⟶$_{cd}$ *n'*› **have** *valid-SDG-node n''* **by**(*rule sum-SDG-edge-valid-SDG-node*)
hence *n'' i*−[]→$_{d}$* *n''* **by**(*rule iSp-Nil*)
with ‹*n'' s*⟶$_{cd}$ *n'*› **have** *n'' i*−[]@[*n''*]→$_{d}$* *n'*
by(*fastforce elim*:*iSp-Append-cdep sum-SDG-edge-SDG-edge*)
hence *matched n'' [n''] n'* **by**(*fastforce intro*:*intra-SDG-path-matched*)
with ‹*realizable n ns' n''*› **have** *realizable n* (*ns'*@[*n''*]) *n'*
by(*rule realizable-Append-matched*)
with ‹*set ns* ⊆ *set ns'*› **show** *?case* **by** *fastforce*
next
case (*icsSp-Append-ddep n ns n'' V n'*)
from ‹∃ *ns'*. *realizable n ns' n''* ∧ *set ns* ⊆ *set ns'*›
obtain *ns'* **where** *realizable n ns' n''* **and** *set ns* ⊆ *set ns'* **by** *blast*
from ‹*n'' s*−*V*→$_{dd}$ *n'*› **have** *valid-SDG-node n''*
by(*rule sum-SDG-edge-valid-SDG-node*)
hence *n'' i*−[]→$_{d}$* *n''* **by**(*rule iSp-Nil*)
with ‹*n'' s*−*V*→$_{dd}$ *n'*› ‹*n''* ≠ *n'*› **have** *n'' i*−[]@[*n''*]→$_{d}$* *n'*
by(*fastforce elim*:*iSp-Append-ddep sum-SDG-edge-SDG-edge*)
hence *matched n'' [n''] n'* **by**(*fastforce intro*:*intra-SDG-path-matched*)
with ‹*realizable n ns' n''*› **have** *realizable n* (*ns'*@[*n''*]) *n'*
by(*fastforce intro*:*realizable-Append-matched*)
with ‹*set ns* ⊆ *set ns'*› **show** *?case* **by** *fastforce*
next
case (*icsSp-Append-sum n ns n'' p n'*)
from ‹∃ *ns'*. *realizable n ns' n''* ∧ *set ns* ⊆ *set ns'*›
obtain *ns'* **where** *realizable n ns' n''* **and** *set ns* ⊆ *set ns'* **by** *blast*
from ‹*n'' s*−*p*→$_{sum}$ *n'*› **show** *?case*
proof(*rule sum-edge-cases*)
fix *a Q r fs a'*
assume *valid-edge a* **and** *kind a* = *Q*:*r*↪$_{p}$*fs* **and** *a'* ∈ *get-return-edges a*
and *n''* = *CFG-node* (*sourcenode a*) **and** *n'* = *CFG-node* (*targetnode a'*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_{p}$*fs*› ‹*a'* ∈ *get-return-edges a*›
have *match'*:*matched* (*CFG-node* (*targetnode a*)) [*CFG-node* (*targetnode a*)]
(*CFG-node* (*sourcenode a'*))
by(*rule intra-proc-matched*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_{p}$*fs*› ‹*n''* = *CFG-node* (*sourcenode a*)›
have *n''* −*p*→$_{call}$ *CFG-node* (*targetnode a*)
by(*fastforce intro*:*SDG-call-edge*)
hence *matched n'' [] n''*
by(*fastforce intro*:*matched-Nil SDG-edge-valid-SDG-node*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_{p}$*fs*› ‹*a'* ∈ *get-return-edges a*›
obtain *Q' f'* **where** *kind a'* = *Q'*↩$_{p}$*f'* **by**(*fastforce dest*!:*call-return-edges*)
from ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩$_{p}$*f'*› ‹*n'* = *CFG-node* (*targetnode a'*)›
have *CFG-node* (*sourcenode a'*) −*p*→$_{ret}$ *n'*

```
      by(fastforce intro:SDG-return-edge)
    from ‹matched n'' [] n''› ‹n'' -p→call CFG-node (targetnode a)›
      match' ‹CFG-node (sourcenode a') -p→ret n'› ‹valid-edge a›
      ‹a' ∈ get-return-edges a› ‹n' = CFG-node (targetnode a')›
      ‹n'' = CFG-node (sourcenode a)›
    have matched n'' ([]@n''#[CFG-node (targetnode a)]@[CFG-node (sourcenode
a')])
      n'
      by(fastforce intro:matched-bracket-call)
    with ‹realizable n ns' n''›
    have realizable n
      (ns'@(n''#[CFG-node (targetnode a),CFG-node (sourcenode a')])) n'
      by(fastforce intro:realizable-Append-matched)
    with ‹set ns ⊆ set ns'› show ?thesis by fastforce
  next
    fix a Q r p fs a' ns'' x x' ins outs
    assume valid-edge a and kind a = Q:r↪pfs and a' ∈ get-return-edges a
      and match':matched (Formal-in (targetnode a,x)) ns''
                         (Formal-out (sourcenode a',x'))
      and n'' = Actual-in (sourcenode a,x)
      and n' = Actual-out (targetnode a',x') and (p,ins,outs) ∈ set procs
      and x < length ins and x' < length outs
    from ‹valid-edge a› ‹kind a = Q:r↪pfs› ‹n'' = Actual-in (sourcenode a,x)›
      ‹(p,ins,outs) ∈ set procs› ‹x < length ins›
    have n'' -p:ins!x→in Formal-in (targetnode a,x)
      by(fastforce intro!:SDG-param-in-edge)
    hence matched n'' [] n''
      by(fastforce intro:matched-Nil SDG-edge-valid-SDG-node)
    from ‹valid-edge a› ‹a' ∈ get-return-edges a› have valid-edge a'
      by(rule get-return-edges-valid)
    from ‹valid-edge a› ‹kind a = Q:r↪pfs› ‹a' ∈ get-return-edges a›
    obtain Q' f' where kind a' = Q'↩pf' by(fastforce dest!:call-return-edges)
    from ‹valid-edge a'› ‹kind a' = Q'↩pf'› ‹n' = Actual-out (targetnode a',x')›
      ‹(p,ins,outs) ∈ set procs› ‹x' < length outs›
    have Formal-out (sourcenode a',x') -p:outs!x'→out n'
      by(fastforce intro:SDG-param-out-edge)
    from ‹matched n'' [] n''› ‹n'' -p:ins!x→in Formal-in (targetnode a,x)›
      match' ‹Formal-out (sourcenode a',x') -p:outs!x'→out n'› ‹valid-edge a›
      ‹a' ∈ get-return-edges a› ‹n' = Actual-out (targetnode a',x')›
      ‹n'' = Actual-in (sourcenode a,x)›
    have matched n'' ([]@n''#ns''@[Formal-out (sourcenode a',x')]) n'
      by(fastforce intro:matched-bracket-param)
    with ‹realizable n ns' n''›
    have realizable n (ns'@(n''#ns''@[Formal-out (sourcenode a',x')])) n'
      by(fastforce intro:realizable-Append-matched)
    with ‹set ns ⊆ set ns'› show ?thesis by fastforce
  qed
next
  case (icsSp-Append-call n ns n'' p n')
```

```
    from ‹∃ ns'. realizable n ns' n'' ∧ set ns ⊆ set ns'›
    obtain ns' where realizable n ns' n'' and set ns ⊆ set ns' by blast
    from ‹n'' s−p→call n'› have valid-SDG-node n'
      by(rule sum-SDG-edge-valid-SDG-node)
    hence matched n' [] n' by(rule matched-Nil)
    with ‹realizable n ns' n''› ‹n'' s−p→call n'›
    have realizable n (ns'@n''#[]) n'
      by(fastforce intro:realizable-call sum-SDG-edge-SDG-edge)
    with ‹set ns ⊆ set ns'› show ?case by fastforce
  next
    case (icsSp-Append-param-in n ns n'' p V n')
    from ‹∃ ns'. realizable n ns' n'' ∧ set ns ⊆ set ns'›
    obtain ns' where realizable n ns' n'' and set ns ⊆ set ns' by blast
    from ‹n'' s−p:V→in n'› have valid-SDG-node n'
      by(rule sum-SDG-edge-valid-SDG-node)
    hence matched n' [] n' by(rule matched-Nil)
    with ‹realizable n ns' n''› ‹n'' s−p:V→in n'›
    have realizable n (ns'@n''#[]) n'
      by(fastforce intro:realizable-call sum-SDG-edge-SDG-edge)
    with ‹set ns ⊆ set ns'› show ?case by fastforce
  qed
qed



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(atomize-elim)
  from ‹n'' ics−ns'→d* n'› ‹realizable n ns n''›
  show ∃ ns''. realizable n (ns@ns'') n'
  proof(induct rule:intra-call-sum-SDG-path.induct)
    case (icsSp-Nil n'') thus ?case by(rule-tac x=[] in exI) fastforce
  next
    case (icsSp-Append-cdep n'' ns' nx n')
    then obtain ns'' where realizable n (ns@ns'') nx by fastforce
   from ‹nx s⟶cd n'› have valid-SDG-node nx by(rule sum-SDG-edge-valid-SDG-node)
    hence matched nx [] nx by(rule matched-Nil)
    from ‹nx s⟶cd n'› ‹valid-SDG-node nx›
    have nx i−[]@[nx]→d* n'
      by(fastforce intro:iSp-Append-cdep iSp-Nil sum-SDG-edge-SDG-edge)
    with ‹matched nx [] nx› have matched nx ([]@[nx]) n'
      by(fastforce intro:matched-Append-intra-SDG-path)
    with ‹realizable n (ns@ns'') nx› have realizable n ((ns@ns'')@[nx]) n'
      by(fastforce intro:realizable-Append-matched)
    thus ?case by fastforce
  next
    case (icsSp-Append-ddep n'' ns' nx V n')
    then obtain ns'' where realizable n (ns@ns'') nx by fastforce
```

from ‹$nx\ s{-}V{\to}_{dd}\ n'$› **have** *valid-SDG-node nx* **by**(*rule sum-SDG-edge-valid-SDG-node*)
hence *matched nx* [] *nx* **by**(*rule matched-Nil*)
from ‹$nx\ s{-}V{\to}_{dd}\ n'$› ‹$nx \neq n'$› ‹*valid-SDG-node nx*›
have $nx\ i{-}[]@[nx]{\to}_{d}*\ n'$
by(*fastforce intro*:*iSp-Append-ddep iSp-Nil sum-SDG-edge-SDG-edge*)
with ‹*matched nx* [] *nx*› **have** *matched nx* ([]@[*nx*]) n'
by(*fastforce intro*:*matched-Append-intra-SDG-path*)
with ‹*realizable n* (*ns*@*ns''*) *nx*› **have** *realizable n* ((*ns*@*ns''*)@[*nx*]) n'
by(*fastforce intro*:*realizable-Append-matched*)
thus *?case* **by** *fastforce*
next
case (*icsSp-Append-sum n'' ns' nx p n'*)
then obtain *ns''* **where** *realizable n* (*ns*@*ns''*) *nx* **by** *fastforce*
from ‹$nx\ s{-}p{\to}_{sum}\ n'$› **obtain** *nsx* **where** *matched nx nsx n'*
by −(*erule sum-SDG-summary-edge-matched*)
with ‹*realizable n* (*ns*@*ns''*) *nx*› **have** *realizable n* ((*ns*@*ns''*)@*nsx*) n'
by(*rule realizable-Append-matched*)
thus *?case* **by** *fastforce*
next
case (*icsSp-Append-call n'' ns' nx p n'*)
then obtain *ns''* **where** *realizable n* (*ns*@*ns''*) *nx* **by** *fastforce*
from ‹$nx\ s{-}p{\to}_{call}\ n'$› **have** *valid-SDG-node n'* **by**(*rule sum-SDG-edge-valid-SDG-node*)
hence *matched n'* [] *n'* **by**(*rule matched-Nil*)
with ‹*realizable n* (*ns*@*ns''*) *nx*› ‹$nx\ s{-}p{\to}_{call}\ n'$›
have *realizable n* ((*ns*@*ns''*)@[*nx*]) n'
by(*fastforce intro*:*realizable-call sum-SDG-edge-SDG-edge*)
thus *?case* **by** *fastforce*
next
case (*icsSp-Append-param-in n'' ns' nx p V n'*)
then obtain *ns''* **where** *realizable n* (*ns*@*ns''*) *nx* **by** *fastforce*
from ‹$nx\ s{-}p{:}V{\to}_{in}\ n'$› **have** *valid-SDG-node n'*
by(*rule sum-SDG-edge-valid-SDG-node*)
hence *matched n'* [] *n'* **by**(*rule matched-Nil*)
with ‹*realizable n* (*ns*@*ns''*) *nx*› ‹$nx\ s{-}p{:}V{\to}_{in}\ n'$›
have *realizable n* ((*ns*@*ns''*)@[*nx*]) n'
by(*fastforce intro*:*realizable-call sum-SDG-edge-SDG-edge*)
thus *?case* **by** *fastforce*
qed
qed

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*:

$[\![n'' \ irs{-}ns{\to_d}* \ n'; \ n \ s{\longrightarrow_{cd}} \ n'']\!] \Longrightarrow n \ irs{-}n\#ns{\to_d}* \ n'$

| *irsSp-Cons-ddep*:
$[\![n'' \ irs{-}ns{\to_d}* \ n'; \ n \ s{-}V{\to_{dd}} \ n''; \ n \neq n'']\!] \Longrightarrow n \ irs{-}n\#ns{\to_d}* \ n'$

| *irsSp-Cons-sum*:
$[\![n'' \ irs{-}ns{\to_d}* \ n'; \ n \ s{-}p{\to_{sum}} \ n'']\!] \Longrightarrow n \ irs{-}n\#ns{\to_d}* \ n'$

| *irsSp-Cons-return*:
$[\![n'' \ irs{-}ns{\to_d}* \ n'; \ n \ s{-}p{\to_{ret}} \ n'']\!] \Longrightarrow n \ irs{-}n\#ns{\to_d}* \ n'$

| *irsSp-Cons-param-out*:
$[\![n'' \ irs{-}ns{\to_d}* \ n'; \ n \ s{-}p{:}V{\to_{out}} \ n'']\!] \Longrightarrow n \ irs{-}n\#ns{\to_d}* \ n'$

lemma *irs-SDG-path-Append*:
$[\![n \ irs{-}ns{\to_d}* \ n''; \ n'' \ irs{-}ns'{\to_d}* \ n']\!] \Longrightarrow n \ irs{-}ns@ns'{\to_d}* \ n'$
by(*induct rule*:*intra-return-sum-SDG-path.induct*,
auto intro:*intra-return-sum-SDG-path.intros*)

lemma *is-SDG-path-irs-SDG-path*:
$n \ is{-}ns{\to_d}* \ n' \Longrightarrow n \ irs{-}ns{\to_d}* \ n'$
proof(*induct rule*:*intra-sum-SDG-path.induct*)
case (*isSp-Nil n*)
from ‹*valid-SDG-node n*› **show** *?case* **by**(*rule irsSp-Nil*)
next
case (*isSp-Append-cdep n ns n'' n'*)
from ‹$n'' \ s{\longrightarrow_{cd}} \ n'$› **have** $n'' \ irs{-}[n'']{\to_d}* \ n'$
by(*fastforce intro*:*irsSp-Cons-cdep irsSp-Nil sum-SDG-edge-valid-SDG-node*)
with ‹$n \ irs{-}ns{\to_d}* \ n''$› **show** *?case* **by**(*rule irs-SDG-path-Append*)
next
case (*isSp-Append-ddep n ns n'' V n'*)
from ‹$n'' \ s{-}V{\to_{dd}} \ n'$› ‹$n'' \neq n'$› **have** $n'' \ irs{-}[n'']{\to_d}* \ n'$
by(*fastforce intro*:*irsSp-Cons-ddep irsSp-Nil sum-SDG-edge-valid-SDG-node*)
with ‹$n \ irs{-}ns{\to_d}* \ n''$› **show** *?case* **by**(*rule irs-SDG-path-Append*)
next
case (*isSp-Append-sum n ns n'' p n'*)
from ‹$n'' \ s{-}p{\to_{sum}} \ n'$› **have** $n'' \ irs{-}[n'']{\to_d}* \ n'$
by(*fastforce intro*:*irsSp-Cons-sum irsSp-Nil sum-SDG-edge-valid-SDG-node*)
with ‹$n \ irs{-}ns{\to_d}* \ n''$› **show** *?case* **by**(*rule irs-SDG-path-Append*)
qed

lemma *irs-SDG-path-split*:
assumes $n \ irs{-}ns{\to_d}* \ n'$
obtains $n \ is{-}ns{\to_d}* \ n'$
| *nsx nsx' nx nx' p* **where** $ns = nsx@nx\#nsx'$ **and** $n \ irs{-}nsx{\to_d}* \ nx$

```
  and nx s−p→ret nx' ∨ (∃ V. nx s−p:V→out nx') and nx' is−nsx'→d* n'
proof(atomize-elim)
  from ‹n irs−ns→d* n'› show n is−ns→d* n' ∨
    (∃ nsx nx nsx' p nx'. ns = nsx@nx#nsx' ∧ n irs−nsx→d* nx ∧
                  (nx s−p→ret nx' ∨ (∃ V. nx s−p:V→out nx')) ∧ nx' is−nsx'→d*
n')
  proof(induct rule:intra-return-sum-SDG-path.induct)
    case (irsSp-Nil n)
    from ‹valid-SDG-node n› have n is−[]→d* n by(rule isSp-Nil)
    thus ?case by simp
  next
    case (irsSp-Cons-cdep n'' ns n' n)
    from ‹n'' is−ns→d* n' ∨
      (∃ nsx nx nsx' p nx'. ns = nsx@nx#nsx' ∧ n'' irs−nsx→d* nx ∧
                  (nx s−p→ret nx' ∨ (∃ V. nx s−p:V→out nx')) ∧ nx' is−nsx'→d*
n')›
    show ?case
    proof
      assume n'' is−ns→d* n'
      from ‹n s⟶cd n''› have n is−[]@[n]→d* n''
      by(fastforce intro:isSp-Append-cdep isSp-Nil sum-SDG-edge-valid-SDG-node)
      with ‹n'' is−ns→d* n'› have n is−[n]@ns→d* n'
        by(fastforce intro:is-SDG-path-Append)
      thus ?case by simp
    next
      assume ∃ nsx nx nsx' p nx'. ns = nsx@nx#nsx' ∧ n'' irs−nsx→d* nx ∧
                  (nx s−p→ret nx' ∨ (∃ V. nx s−p:V→out nx')) ∧ nx' is−nsx'→d*
n'
    then obtain 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'
by blast
      from ‹n'' irs−nsx→d* nx› ‹n s⟶cd n''› have n irs−n#nsx→d* nx
        by(rule intra-return-sum-SDG-path.irsSp-Cons-cdep)
      with ‹ns = nsx@nx#nsx'› ‹nx s−p→ret nx' ∨ (∃ V. nx s−p:V→out nx')›
        ‹nx' is−nsx'→d* n'›
      show ?case by fastforce
    qed
  next
    case (irsSp-Cons-ddep n'' ns n' n V)
    from ‹n'' is−ns→d* n' ∨
      (∃ nsx nx nsx' p nx'. ns = nsx@nx#nsx' ∧ n'' irs−nsx→d* nx ∧
                  (nx s−p→ret nx' ∨ (∃ V. nx s−p:V→out nx')) ∧ nx' is−nsx'→d*
n')›
    show ?case
    proof
      assume n'' is−ns→d* n'
      from ‹n s−V→dd n''› ‹n ≠ n''› have n is−[]@[n]→d* n''
      by(fastforce intro:isSp-Append-ddep isSp-Nil sum-SDG-edge-valid-SDG-node)
```

with ⟨n'' $is{-}ns{\to_d}*$ n'⟩ **have** n $is{-}[n]@ns{\to_d}*$ n'
by(*fastforce intro*:*is-SDG-path-Append*)
thus *?case* **by** *simp*
next
assume $\exists\, nsx\ nx\ nsx'\ p\ nx'.\ ns = nsx@nx\#nsx' \wedge n''\ irs{-}nsx{\to_d}*\ nx \wedge (nx\ s{-}p{\to_{ret}}\ nx' \vee (\exists\, V.\ nx\ s{-}p{:}V{\to_{out}}\ nx')) \wedge nx'\ is{-}nsx'{\to_d}*\ n'$
then obtain $nsx\ nsx'\ nx\ nx'\ p$ **where** $ns = nsx@nx\#nsx'$ **and** $n''\ irs{-}nsx{\to_d}*\ nx$
and $nx\ s{-}p{\to_{ret}}\ nx' \vee (\exists\, V.\ nx\ s{-}p{:}V{\to_{out}}\ nx')$ **and** $nx'\ is{-}nsx'{\to_d}*\ n'$ **by** *blast*
from ⟨$n''\ irs{-}nsx{\to_d}*\ nx$⟩ ⟨$n\ s{-}V{\to_{dd}}\ n''$⟩ ⟨$n \neq n''$⟩ **have** $n\ irs{-}n\#nsx{\to_d}*\ nx$
by(*rule intra-return-sum-SDG-path.irsSp-Cons-ddep*)
with ⟨$ns = nsx@nx\#nsx'$⟩ ⟨$nx\ s{-}p{\to_{ret}}\ nx' \vee (\exists\, V.\ nx\ s{-}p{:}V{\to_{out}}\ nx')$⟩ ⟨$nx'\ is{-}nsx'{\to_d}*\ n'$⟩
show *?case* **by** *fastforce*
qed
next
case (*irsSp-Cons-sum* $n''\ ns\ n'\ n\ p$)
from ⟨$n''\ is{-}ns{\to_d}*\ n' \vee$
$(\exists\, nsx\ nx\ nsx'\ p\ nx'.\ ns = nsx@nx\#nsx' \wedge n''\ irs{-}nsx{\to_d}*\ nx \wedge (nx\ s{-}p{\to_{ret}}\ nx' \vee (\exists\, V.\ nx\ s{-}p{:}V{\to_{out}}\ nx')) \wedge nx'\ is{-}nsx'{\to_d}*\ n')$⟩
show *?case*
proof
assume $n''\ is{-}ns{\to_d}*\ n'$
from ⟨$n\ s{-}p{\to_{sum}}\ n''$⟩ **have** $n\ is{-}[]@[n]{\to_d}*\ n''$
by(*fastforce intro*:*isSp-Append-sum isSp-Nil sum-SDG-edge-valid-SDG-node*)
with ⟨$n''\ is{-}ns{\to_d}*\ n'$⟩ **have** $n\ is{-}[n]@ns{\to_d}*\ n'$
by(*fastforce intro*:*is-SDG-path-Append*)
thus *?case* **by** *simp*
next
assume $\exists\, nsx\ nx\ nsx'\ p\ nx'.\ ns = nsx@nx\#nsx' \wedge n''\ irs{-}nsx{\to_d}*\ nx \wedge (nx\ s{-}p{\to_{ret}}\ nx' \vee (\exists\, V.\ nx\ s{-}p{:}V{\to_{out}}\ nx')) \wedge nx'\ is{-}nsx'{\to_d}*\ n'$
then obtain $nsx\ nsx'\ nx\ nx'\ p'$ **where** $ns = nsx@nx\#nsx'$ **and** $n''\ irs{-}nsx{\to_d}*\ nx$
and $nx\ s{-}p'{\to_{ret}}\ nx' \vee (\exists\, V.\ nx\ s{-}p'{:}V{\to_{out}}\ nx')$
and $nx'\ is{-}nsx'{\to_d}*\ n'$ **by** *blast*
from ⟨$n''\ irs{-}nsx{\to_d}*\ nx$⟩ ⟨$n\ s{-}p{\to_{sum}}\ n''$⟩ **have** $n\ irs{-}n\#nsx{\to_d}*\ nx$
by(*rule intra-return-sum-SDG-path.irsSp-Cons-sum*)
with ⟨$ns = nsx@nx\#nsx'$⟩ ⟨$nx\ s{-}p'{\to_{ret}}\ nx' \vee (\exists\, V.\ nx\ s{-}p'{:}V{\to_{out}}\ nx')$⟩ ⟨$nx'\ is{-}nsx'{\to_d}*\ n'$⟩
show *?case* **by** *fastforce*
qed
next
case (*irsSp-Cons-return* $n''\ ns\ n'\ n\ p$)
from ⟨$n''\ is{-}ns{\to_d}*\ n' \vee$

```
    (∃ nsx nx nsx' p nx'. ns = nsx@nx#nsx' ∧ n'' irs−nsx→d* nx ∧
                (nx s−p→ret nx' ∨ (∃ V. nx s−p:V→out nx')) ∧ nx' is−nsx'→d*
n')⟩
   show ?case
   proof
    assume n'' is−ns→d* n'
   from ⟨n s−p→ret n''⟩ have valid-SDG-node n by(rule sum-SDG-edge-valid-SDG-node)
    hence n irs−[]→d* n by(rule irsSp-Nil)
    with ⟨n s−p→ret n''⟩ ⟨n'' is−ns→d* n'⟩ show ?thesis by fastforce
   next
    assume ∃ nsx nx nsx' p nx'. ns = nsx@nx#nsx' ∧ n'' irs−nsx→d* nx ∧
                (nx s−p→ret nx' ∨ (∃ V. nx s−p:V→out nx')) ∧ nx' is−nsx'→d*
n'
   then obtain 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' by blast
    from ⟨n'' irs−nsx→d* nx⟩ ⟨n s−p→ret n''⟩ have n irs−n#nsx→d* nx
     by(rule intra-return-sum-SDG-path.irsSp-Cons-return)
    with ⟨ns = nsx@nx#nsx'⟩ ⟨nx s−p'→ret nx' ∨ (∃ V. nx s−p':V→out nx')⟩
     ⟨nx' is−nsx'→d* n'⟩
    show ?thesis by fastforce
   qed
  next
   case (irsSp-Cons-param-out n'' ns n' n p V)
   from ⟨n'' is−ns→d* n' ∨
    (∃ nsx nx nsx' p nx'. ns = nsx@nx#nsx' ∧ n'' irs−nsx→d* nx ∧
                (nx s−p→ret nx' ∨ (∃ V. nx s−p:V→out nx')) ∧ nx' is−nsx'→d*
n')⟩
   show ?case
   proof
    assume n'' is−ns→d* n'
    from ⟨n s−p:V→out n''⟩ have valid-SDG-node n
     by(rule sum-SDG-edge-valid-SDG-node)
    hence n irs−[]→d* n by(rule irsSp-Nil)
    with ⟨n s−p:V→out n''⟩ ⟨n'' is−ns→d* n'⟩ show ?thesis by fastforce
   next
    assume ∃ nsx nx nsx' p nx'. ns = nsx@nx#nsx' ∧ n'' irs−nsx→d* nx ∧
                (nx s−p→ret nx' ∨ (∃ V. nx s−p:V→out nx')) ∧ nx' is−nsx'→d*
n'
   then obtain 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' by blast
    from ⟨n'' irs−nsx→d* nx⟩ ⟨n s−p:V→out n''⟩ have n irs−n#nsx→d* nx
     by(rule intra-return-sum-SDG-path.irsSp-Cons-param-out)
    with ⟨ns = nsx@nx#nsx'⟩ ⟨nx s−p'→ret nx' ∨ (∃ V. nx s−p':V→out nx')⟩
     ⟨nx' is−nsx'→d* n'⟩
    show ?thesis by fastforce
```

qed
qed
qed

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(*atomize-elim*)
from *assms*
show $\exists$ *nx nsx*. *matched nx nsx* $n' \wedge n \in$ *set nsx* $\wedge$
nx $s-p\rightarrow_{sum}$ *CFG-node* (*parent-node* n')
proof(*induct ns arbitrary*:n'' n' p V *rule*:*length-induct*)
fix *ns* n'' n' p V
assume *IH*:$\forall$ *ns'*. *length ns'* $<$ *length ns* $\longrightarrow$
($\forall$ n''. n *irs*$-ns'\rightarrow_d*$ $n'' \longrightarrow$
($\forall$ nx' p' V'. (n'' $s-p'\rightarrow_{ret}$ $nx' \vee n''$ $s-p'$:$V'\rightarrow_{out}$ nx') $\longrightarrow$
($\exists$ *nx nsx*. *matched nx nsx* $nx' \wedge n \in$ *set nsx* $\wedge$
nx $s-p'\rightarrow_{sum}$ *CFG-node* (*parent-node* nx'))))
and n *irs*$-ns\rightarrow_d*$ n'' **and** n'' $s-p\rightarrow_{ret}$ $n' \vee n''$ $s-p$:$V\rightarrow_{out}$ n'
from ‹n'' $s-p\rightarrow_{ret}$ $n' \vee n''$ $s-p$:$V\rightarrow_{out}$ n'› **have** *valid-SDG-node* n''
by(*fastforce intro*:*sum-SDG-edge-valid-SDG-node*)
from ‹n'' $s-p\rightarrow_{ret}$ $n' \vee n''$ $s-p$:$V\rightarrow_{out}$ n'›
have n'' $-p\rightarrow_{ret}$ $n' \vee n''$ $-p$:$V\rightarrow_{out}$ n'
by(*fastforce intro*:*sum-SDG-edge-SDG-edge SDG-edge-sum-SDG-edge*)
from ‹n'' $s-p\rightarrow_{ret}$ $n' \vee n''$ $s-p$:$V\rightarrow_{out}$ n'›
have *CFG-node* (*parent-node* n'') $s-p\rightarrow_{ret}$ *CFG-node* (*parent-node* n')
by(*fastforce elim*:*sum-SDG-edge.cases intro*:*sum-SDG-return-edge*)
then obtain a Q f **where** *valid-edge* a **and** *kind* $a = Q\hookleftarrow_p f$
and *parent-node* $n'' =$ *sourcenode* a **and** *parent-node* $n' =$ *targetnode* a
by(*fastforce elim*:*sum-SDG-edge.cases*)
from ‹*valid-edge* a› ‹*kind* $a = Q\hookleftarrow_p f$› **obtain** a' Q' r' fs'
where $a \in$ *get-return-edges* a' **and** *valid-edge* a' **and** *kind* $a' = Q'$:$r'\hookrightarrow_p fs'$
and *CFG-node* (*sourcenode* a') $s-p\rightarrow_{sum}$ *CFG-node* (*targetnode* a)
by(*erule return-edge-determines-call-and-sum-edge*)
from ‹*valid-edge* a'› ‹*kind* $a' = Q'$:$r'\hookrightarrow_p fs'$›
have *CFG-node* (*sourcenode* a') $s-p\rightarrow_{call}$ *CFG-node* (*targetnode* a')
by(*fastforce intro*:*sum-SDG-call-edge*)
from ‹*CFG-node* (*parent-node* n'') $s-p\rightarrow_{ret}$ *CFG-node* (*parent-node* n')›
have *get-proc* (*parent-node* n'') $= p$
by(*auto elim*!:*sum-SDG-edge.cases intro*:*get-proc-return*)
from ‹n *irs*$-ns\rightarrow_d*$ n''›
show $\exists$ *nx nsx*. *matched nx nsx* $n' \wedge n \in$ *set nsx* $\wedge$
nx $s-p\rightarrow_{sum}$ *CFG-node* (*parent-node* n')
proof(*rule irs-SDG-path-split*)
assume n *is*$-ns\rightarrow_d*$ n''
hence *valid-SDG-node* n **by**(*rule is-SDG-path-valid-SDG-node*)
then obtain *asx* **where** (*-Entry-*) $-asx\rightarrow_{\sqrt{}}*$ *parent-node* n

by(*fastforce dest*:*valid-SDG-CFG-node Entry-path*)
then obtain *asx'* **where** (*-Entry-*) $-asx'\rightarrow_{\surd}*$ *parent-node n*
and $\forall\, a' \in$ *set asx'*. *intra-kind*(*kind a'*) $\vee$ ($\exists\, Q\ r\ p\ fs$. *kind a'* = $Q{:}r{\hookrightarrow_p}fs$)
by $-$(*erule valid-Entry-path-ascending-path*)
from ‹*n is*$-ns\rightarrow_d*$ *n''*› **obtain** *as* **where** *parent-node n* $-as\rightarrow_\iota*$ *parent-node n''*
by(*erule is-SDG-path-CFG-path*)
hence *get-proc* (*parent-node n*) = *get-proc* (*parent-node n''*)
by(*rule intra-path-get-procs*)
from ‹*valid-SDG-node n*› **have** *valid-node* (*parent-node n*)
by(*rule valid-SDG-CFG-node*)
hence *valid-SDG-node* (*CFG-node* (*parent-node n*)) **by** *simp*
have $\exists\, a\ as$. *valid-edge a* $\wedge$ ($\exists\, Q\ p\ r\ fs$. *kind a* = $Q{:}r{\hookrightarrow_p}fs$) $\wedge$
targetnode a $-as\rightarrow_\iota*$ *parent-node n*
proof(*cases* $\forall\, a' \in$ *set asx'*. *intra-kind*(*kind a'*))
case *True*
with ‹(*-Entry-*) $-asx'\rightarrow_{\surd}*$ *parent-node n*›
have (*-Entry-*) $-asx'\rightarrow_\iota*$ *parent-node n*
by(*fastforce simp*:*intra-path-def vp-def*)
hence *get-proc* (*-Entry-*) = *get-proc* (*parent-node n*)
by(*rule intra-path-get-procs*)
with *get-proc-Entry* **have** *get-proc* (*parent-node n*) = *Main* **by** *simp*
from ‹*get-proc* (*parent-node n*) = *get-proc* (*parent-node n''*)›
‹*get-proc* (*parent-node n*) = *Main*›
have *get-proc* (*parent-node n''*) = *Main* **by** *simp*
from ‹*valid-edge a*› ‹*kind a* = $Q{\leftarrow_p}f$› **have** *get-proc* (*sourcenode a*) = *p*
by(*rule get-proc-return*)
with ‹*parent-node n''* = *sourcenode a*› ‹*get-proc* (*parent-node n''*) = *Main*›
have *p* = *Main* **by** *simp*
with ‹*kind a* = $Q{\leftarrow_p}f$› **have** *kind a* = $Q{\leftarrow_{Main}}f$ **by** *simp*
with ‹*valid-edge a*› **have** *False* **by**(*rule Main-no-return-source*)
thus *?thesis* **by** *simp*
next
assume $\neg$ ($\forall\, a' \in$ *set asx'*. *intra-kind* (*kind a'*))
with ‹$\forall\, a' \in$ *set asx'*. *intra-kind*(*kind a'*) $\vee$ ($\exists\, Q\ r\ p\ fs$. *kind a'* = $Q{:}r{\hookrightarrow_p}fs$)›
have $\exists\, a' \in$ *set asx'*. $\exists\, Q\ r\ p\ fs$. *kind a'* = $Q{:}r{\hookrightarrow_p}fs$
by(*fastforce simp*:*intra-kind-def*)
then obtain *as a' as'* **where** *asx'* = *as*@*a'*#*as'*
and $\exists\, Q\ r\ p\ fs$. *kind a'* = $Q{:}r{\hookrightarrow_p}fs$
and $\forall\, a' \in$ *set as'*. $\neg$ ($\exists\, Q\ r\ p\ fs$. *kind a'* = $Q{:}r{\hookrightarrow_p}fs$)
by(*erule split-list-last-propE*)
with ‹$\forall\, a' \in$ *set asx'*. *intra-kind*(*kind a'*) $\vee$ ($\exists\, Q\ r\ p\ fs$. *kind a'* = $Q{:}r{\hookrightarrow_p}fs$)›
have $\forall\, a' \in$*set as'*. *intra-kind* (*kind a'*) **by**(*auto simp*:*intra-kind-def*)
from ‹(*-Entry-*) $-asx'\rightarrow_{\surd}*$ *parent-node n*› ‹*asx'* = *as*@*a'*#*as'*›
have *valid-edge a'* **and** *targetnode a'* $-as'\rightarrow*$ *parent-node n*
by(*auto dest*:*path-split simp*:*vp-def*)
with ‹$\forall\, a' \in$*set as'*. *intra-kind* (*kind a'*)› ‹$\exists\, Q\ r\ p\ fs$. *kind a'* = $Q{:}r{\hookrightarrow_p}fs$›
show *?thesis* **by**(*fastforce simp*:*intra-path-def*)
qed

then obtain *ax asx Qx rx fsx px* **where** *valid-edge ax*
and *kind ax* = *Qx*:*rx*↪$_{px}$*fsx* **and** *targetnode ax* −*asx*→$_\iota$∗ *parent-node n*
by *blast*
from ‹*valid-edge ax*› ‹*kind ax* = *Qx*:*rx*↪$_{px}$*fsx*›
have *get-proc* (*targetnode ax*) = *px*
by(*rule get-proc-call*)
from ‹*targetnode ax* −*asx*→$_\iota$∗ *parent-node n*›
have *get-proc* (*targetnode ax*) = *get-proc* (*parent-node n*)
by(*rule intra-path-get-procs*)
with ‹*get-proc* (*parent-node n*) = *get-proc* (*parent-node n''*)›
‹*get-proc* (*targetnode ax*) = *px*›
have *get-proc* (*parent-node n''*) = *px* **by** *simp*
with ‹*get-proc* (*parent-node n''*) = *p*› **have** [*simp*]:*px* = *p* **by** *simp*
from ‹*valid-edge a'*› ‹*valid-edge ax*› ‹*kind a'* = *Q'*:*r'*↪$_p$*fs'*›
‹*kind ax* = *Qx*:*rx*↪$_{px}$*fsx*›
have *targetnode a'* = *targetnode ax* **by** *simp*(*rule same-proc-call-unique-target*)
have *parent-node n* ≠ (*-Exit-*)
proof
assume *parent-node n* = (*-Exit-*)
from ‹*n is*−*ns*→$_d$∗ *n''*› **obtain** *as* **where** *parent-node n* −*as*→$_\iota$∗ *parent-node n''*
by(*erule is-SDG-path-CFG-path*)
with ‹*parent-node n* = (*-Exit-*)›
have (*-Exit-*) −*as*→∗ *parent-node n''* **by**(*simp add*:*intra-path-def*)
hence *parent-node n''* = (*-Exit-*) **by**(*fastforce dest*:*path-Exit-source*)
from ‹*get-proc* (*parent-node n''*) = *p*› ‹*parent-node n''* = (*-Exit-*)›
‹*parent-node n''* = *sourcenode a*› *get-proc-Exit*
have *p* = *Main* **by** *simp*
with ‹*kind a* = *Q*↩$_p$*f*› **have** *kind a* = *Q*↩$_{Main}$*f* **by** *simp*
with ‹*valid-edge a*› **show** *False* **by**(*rule Main-no-return-source*)
qed
have ∃*nsx*. *CFG-node* (*targetnode a'*) *cd*−*nsx*→$_d$∗ *CFG-node* (*parent-node n*)
proof(*cases targetnode a'* = *parent-node n*)
case *True*
with ‹*valid-SDG-node* (*CFG-node* (*parent-node n*))›
have *CFG-node* (*targetnode a'*) *cd*−[]→$_d$∗ *CFG-node* (*parent-node n*)
by(*fastforce intro*:*cdSp-Nil*)
thus *?thesis* **by** *blast*
next
case *False*
with ‹*targetnode ax* −*asx*→$_\iota$∗ *parent-node n*› ‹*parent-node n* ≠ (*-Exit-*)›
‹*valid-edge ax*› ‹*kind ax* = *Qx*:*rx*↪$_{px}$*fsx*› ‹*targetnode a'* = *targetnode ax*›
obtain *nsx*
where *CFG-node* (*targetnode a'*) *cd*−*nsx*→$_d$∗ *CFG-node* (*parent-node n*)
by(*fastforce elim*!:*in-proc-cdep-SDG-path*)
thus *?thesis* **by** *blast*
qed
then obtain *nsx*

where *CFG-node (targetnode a′) cd−nsx→$_{d}$* CFG-node (parent-node n)*
by *blast*
hence *CFG-node (targetnode a′) i−nsx→$_{d}$* CFG-node (parent-node n)*
by(*rule cdep-SDG-path-intra-SDG-path*)
show *?thesis*
proof(*cases ns*)
case *Nil*
with ‹*n is−ns→$_{d}$* n′′*› **have** *n = n′′*
by(*fastforce elim:intra-sum-SDG-path.cases*)
from ‹*valid-edge a′*› ‹*kind a′ = Q′:r′↪$_{pfs}$′*› ‹*a ∈ get-return-edges a′*›
have *matched (CFG-node (targetnode a′)) [CFG-node (targetnode a′)]*
(CFG-node (sourcenode a)) **by**(*rule intra-proc-matched*)
from ‹*valid-SDG-node n′′*›
have *n′′ = CFG-node (parent-node n′′) ∨ CFG-node (parent-node n′′) ⟶$_{cd}$ n′′*
by(*rule valid-SDG-node-cases*)
hence ∃*nsx. CFG-node (parent-node n′′) i−nsx→$_{d}$* n′′*
proof
assume *n′′ = CFG-node (parent-node n′′)*
with ‹*valid-SDG-node n′′*› **have** *CFG-node (parent-node n′′) i−[]→$_{d}$* n′′*
by(*fastforce intro:iSp-Nil*)
thus *?thesis* **by** *blast*
next
assume *CFG-node (parent-node n′′) ⟶$_{cd}$ n′′*
from ‹*valid-SDG-node n′′*› **have** *valid-node (parent-node n′′)*
by(*rule valid-SDG-CFG-node*)
hence *valid-SDG-node (CFG-node (parent-node n′′))* **by** *simp*
hence *CFG-node (parent-node n′′) i−[]→$_{d}$* CFG-node (parent-node n′′)*
by(*rule iSp-Nil*)
with ‹*CFG-node (parent-node n′′) ⟶$_{cd}$ n′′*›
have *CFG-node (parent-node n′′) i−[]@[CFG-node (parent-node n′′)]→$_{d}$* n′′*
by(*fastforce intro:iSp-Append-cdep sum-SDG-edge-SDG-edge*)
thus *?thesis* **by** *blast*
qed
with ‹*parent-node n′′ = sourcenode a*›
obtain *nsx* **where** *CFG-node (sourcenode a) i−nsx→$_{d}$* n′′* **by** *fastforce*
with ‹*matched (CFG-node (targetnode a′)) [CFG-node (targetnode a′)]*
(CFG-node (sourcenode a))›
have *matched (CFG-node (targetnode a′)) ([CFG-node (targetnode a′)]@nsx) n′′*
by(*fastforce intro:matched-Append intra-SDG-path-matched*)
moreover
from ‹*valid-edge a′*› ‹*kind a′ = Q′:r′↪$_{pfs}$′*›
have *CFG-node (sourcenode a′) −p→$_{call}$ CFG-node (targetnode a′)*
by(*fastforce intro:SDG-call-edge*)
moreover
from ‹*valid-edge a′*› **have** *valid-SDG-node (CFG-node (sourcenode a′))*
by *simp*

hence *matched* (*CFG-node* (*sourcenode a'*)) [] (*CFG-node* (*sourcenode a'*))
by(*rule matched-Nil*)
ultimately have *matched* (*CFG-node* (*sourcenode a'*))
([]@(*CFG-node* (*sourcenode a'*))#([*CFG-node* (*targetnode a'*)]@*nsx*)@[*n''*])
n'
using ‹*n'' s−p→ret n'* ∨ *n'' s−p:V→out n'*› ‹*parent-node n' = targetnode a*›
‹*parent-node n'' = sourcenode a*› ‹*valid-edge a'*› ‹*a* ∈ *get-return-edges a'*›
by(*fastforce intro*:*matched-bracket-call dest*:*sum-SDG-edge-SDG-edge*)
with ‹*n* = *n''*› ‹*CFG-node* (*sourcenode a'*) *s−p→sum CFG-node* (*targetnode a*)›
‹*parent-node n' = targetnode a*›
show *?thesis* **by** *fastforce*
next
case *Cons*
with ‹*n is−ns→d* n''*› **have** *n* ∈ *set ns*
by(*induct rule*:*intra-sum-SDG-path-rev-induct*) *auto*
from ‹*n is−ns→d* n''*› **obtain** *ns'* **where** *matched n ns' n''*
and *set ns* ⊆ *set ns'* **by**(*erule is-SDG-path-matched*)
with ‹*n* ∈ *set ns*› **have** *n* ∈ *set ns'* **by** *fastforce*
from ‹*valid-SDG-node n*›
have *n* = *CFG-node* (*parent-node n*) ∨ *CFG-node* (*parent-node n*) ⟶*cd n*
by(*rule valid-SDG-node-cases*)
hence ∃ *nsx*. *CFG-node* (*parent-node n*) *i−nsx→d* n*
proof
assume *n* = *CFG-node* (*parent-node n*)
with ‹*valid-SDG-node n*› **have** *CFG-node* (*parent-node n*) *i−[]→d* n*
by(*fastforce intro*:*iSp-Nil*)
thus *?thesis* **by** *blast*
next
assume *CFG-node* (*parent-node n*) ⟶*cd n*
from ‹*valid-SDG-node* (*CFG-node* (*parent-node n*))›
have *CFG-node* (*parent-node n*) *i−[]→d* CFG-node* (*parent-node n*)
by(*rule iSp-Nil*)
with ‹*CFG-node* (*parent-node n*) ⟶*cd n*›
have *CFG-node* (*parent-node n*) *i−[]@[CFG-node* (*parent-node n*)*]→d* n*
by(*fastforce intro*:*iSp-Append-cdep sum-SDG-edge-SDG-edge*)
thus *?thesis* **by** *blast*
qed
then obtain *nsx'* **where** *CFG-node* (*parent-node n*) *i−nsx'→d* n* **by** *blast*
with ‹*CFG-node* (*targetnode a'*) *i−nsx→d* CFG-node* (*parent-node n*)›
have *CFG-node* (*targetnode a'*) *i−nsx@nsx'→d* n*
by −(*rule intra-SDG-path-Append*)
hence *matched* (*CFG-node* (*targetnode a'*)) (*nsx@nsx'*) *n*
by(*rule intra-SDG-path-matched*)
with ‹*matched n ns' n''*›
have *matched* (*CFG-node* (*targetnode a'*)) ((*nsx@nsx'*)@*ns'*) *n''*
by(*rule matched-Append*)
moreover

from ‹*valid-edge a'*› ‹*kind a' = Q':r'↪$_{pfs'}$*›
have *CFG-node* (*sourcenode a'*) −*p*→$_{call}$ *CFG-node* (*targetnode a'*)
by(*fastforce intro*:*SDG-call-edge*)
moreover
from ‹*valid-edge a'*› **have** *valid-SDG-node* (*CFG-node* (*sourcenode a'*))
by *simp*
hence *matched* (*CFG-node* (*sourcenode a'*)) [] (*CFG-node* (*sourcenode a'*))
by(*rule matched-Nil*)
ultimately have *matched* (*CFG-node* (*sourcenode a'*))
([]@(*CFG-node* (*sourcenode a'*))#((*nsx*@*nsx'*)@*ns'*)@[*n''*]) *n'*
using ‹*n'' s*−*p*→$_{ret}$ *n'* ∨ *n'' s*−*p*:*V*→$_{out}$ *n'*› ‹*parent-node n' = targetnode a*›
‹*parent-node n'' = sourcenode a*› ‹*valid-edge a'*› ‹*a* ∈ *get-return-edges a'*›
by(*fastforce intro*:*matched-bracket-call dest*:*sum-SDG-edge-SDG-edge*)
with ‹*CFG-node* (*sourcenode a'*) *s*−*p*→$_{sum}$ *CFG-node* (*targetnode a*)›
‹*parent-node n' = targetnode a*› ‹*n* ∈ *set ns'*›
show *?thesis* **by** *fastforce*
qed
next
fix *ms ms' m m' px*
assume *ns* = *ms*@*m*#*ms'* **and** *n irs*−*ms*→$_{d}$* *m*
and *m s*−*px*→$_{ret}$ *m'* ∨ (∃ *V*. *m s*−*px*:*V*→$_{out}$ *m'*) **and** *m' is*−*ms'*→$_{d}$* *n''*
from ‹*ns* = *ms*@*m*#*ms'*› **have** *length ms* < *length ns* **by** *simp*
with *IH* ‹*n irs*−*ms*→$_{d}$* *m*› ‹*m s*−*px*→$_{ret}$ *m'* ∨ (∃ *V*. *m s*−*px*:*V*→$_{out}$ *m'*)›
obtain *mx msx*
where *matched mx msx m'* **and** *n* ∈ *set msx*
and *mx s*−*px*→$_{sum}$ *CFG-node* (*parent-node m'*) **by** *fastforce*
from ‹*m' is*−*ms'*→$_{d}$* *n''*› **obtain** *msx'* **where** *matched m' msx' n''*
by −(*erule is-SDG-path-matched*)
with ‹*matched mx msx m'*› **have** *matched mx* (*msx*@*msx'*) *n''*
by −(*rule matched-Append*)
from ‹*m s*−*px*→$_{ret}$ *m'* ∨ (∃ *V*. *m s*−*px*:*V*→$_{out}$ *m'*)›
have *m* −*px*→$_{ret}$ *m'* ∨ (∃ *V*. *m* −*px*:*V*→$_{out}$ *m'*)
by(*auto intro*:*sum-SDG-edge-SDG-edge SDG-edge-sum-SDG-edge*)
from ‹*m s*−*px*→$_{ret}$ *m'* ∨ (∃ *V*. *m s*−*px*:*V*→$_{out}$ *m'*)›
have *CFG-node* (*parent-node m*) *s*−*px*→$_{ret}$ *CFG-node* (*parent-node m'*)
by(*fastforce elim*:*sum-SDG-edge.cases intro*:*sum-SDG-return-edge*)
then obtain *ax Qx fx* **where** *valid-edge ax* **and** *kind ax* = *Qx*↩$_{px}$*fx*
and *parent-node m* = *sourcenode ax* **and** *parent-node m'* = *targetnode ax*
by(*fastforce elim*:*sum-SDG-edge.cases*)
from ‹*valid-edge ax*› ‹*kind ax* = *Qx*↩$_{px}$*fx*› **obtain** *ax' Qx' rx' fsx'*
where *ax* ∈ *get-return-edges ax'* **and** *valid-edge ax'*
and *kind ax'* = *Qx':rx'*↪$_{px}$*fsx'*
and *CFG-node* (*sourcenode ax'*) *s*−*px*→$_{sum}$ *CFG-node* (*targetnode ax*)
by(*erule return-edge-determines-call-and-sum-edge*)
from ‹*valid-edge ax'*› ‹*kind ax'* = *Qx':rx'*↪$_{px}$*fsx'*›
have *CFG-node* (*sourcenode ax'*) *s*−*px*→$_{call}$ *CFG-node* (*targetnode ax'*)
by(*fastforce intro*:*sum-SDG-call-edge*)
from ‹*mx s*−*px*→$_{sum}$ *CFG-node* (*parent-node m'*)›

have *valid-SDG-node mx* **by**(*rule sum-SDG-edge-valid-SDG-node*)
have $\exists\, msx''.$ *CFG-node* (*targetnode a'*) *cd*$-msx''\rightarrow_{d*}$ *mx*
proof(*cases targetnode a' = parent-node mx*)
case *True*
from ‹*valid-SDG-node mx*›
have *mx = CFG-node* (*parent-node mx*) $\vee$ *CFG-node* (*parent-node mx*) $\longrightarrow_{cd}$ *mx*
by(*rule valid-SDG-node-cases*)
thus *?thesis*
proof
assume *mx = CFG-node* (*parent-node mx*)
with ‹*valid-SDG-node mx*› *True*
have *CFG-node* (*targetnode a'*) *cd*$-[]\rightarrow_{d*}$ *mx* **by**(*fastforce intro*:*cdSp-Nil*)
thus *?thesis* **by** *blast*
next
assume *CFG-node* (*parent-node mx*) $\longrightarrow_{cd}$ *mx*
with ‹*valid-edge a'*› *True*[*THEN sym*]
have *CFG-node* (*targetnode a'*) *cd*$-[]@[$*CFG-node* (*targetnode a'*)$]\rightarrow_{d*}$ *mx*
by(*fastforce intro*:*cdep-SDG-path.intros*)
thus *?thesis* **by** *blast*
qed
next
case *False*
show *?thesis*
proof(*cases* $\forall$ *ai. valid-edge ai* $\wedge$ *sourcenode ai = parent-node mx* $\longrightarrow$ *ai* $\notin$ *get-return-edges a'*)
case *True*
{ assume *parent-node mx* = (*-Exit-*)
with ‹*mx s*$-px\rightarrow_{sum}$ *CFG-node* (*parent-node m'*)›
obtain *ai* **where** *valid-edge ai* **and** *sourcenode ai* = (*-Exit-*)
by $-$(*erule sum-SDG-edge.cases*,*auto*)
hence *False* **by**(*rule Exit-source*) **}**
hence *parent-node mx* $\neq$ (*-Exit-*) **by** *fastforce*
from ‹*valid-SDG-node mx*› **have** *valid-node* (*parent-node mx*)
by(*rule valid-SDG-CFG-node*)
then obtain *asx* **where** (*-Entry-*) *asx*$\rightarrow_{\surd*}$ *parent-node mx*
by(*fastforce intro*:*Entry-path*)
then obtain *asx'* **where** (*-Entry-*) *asx'*$\rightarrow_{\surd*}$ *parent-node mx*
and $\forall\, a' \in$ *set asx'. intra-kind*(*kind a'*) $\vee$ ($\exists\, Q\ r\ p\ fs.$ *kind a'* = *Q*:*r*$\hookrightarrow_p$*fs*)
by $-$(*erule valid-Entry-path-ascending-path*)
from ‹*mx s*$-px\rightarrow_{sum}$ *CFG-node* (*parent-node m'*)›
obtain *nsi* **where** *matched mx nsi* (*CFG-node* (*parent-node m'*))
by $-$(*erule sum-SDG-summary-edge-matched*)
then obtain *asi* **where** *parent-node mx* $-$*asi*$\rightarrow_{sl*}$ *parent-node m'*
by(*fastforce elim*:*matched-same-level-CFG-path*)
hence *get-proc* (*parent-node mx*) = *get-proc* (*parent-node m'*)
by(*rule slp-get-proc*)
from ‹*m' is*$-ms'\rightarrow_{d*}$ *n''*› **obtain** *nsi'* **where** *matched m' nsi' n''*
by $-$(*erule is-SDG-path-matched*)

then obtain *asi'* **where** *parent-node m' −asi'→$_{sl}$* parent-node n''*
by −(*erule matched-same-level-CFG-path*)
hence *get-proc* (*parent-node m'*) = *get-proc* (*parent-node n''*)
by(*rule slp-get-proc*)
with ‹*get-proc* (*parent-node mx*) = *get-proc* (*parent-node m'*)›
have *get-proc* (*parent-node mx*) = *get-proc* (*parent-node n''*) **by** *simp*
from ‹*get-proc* (*parent-node n''*) = *p*›
‹*get-proc* (*parent-node mx*) = *get-proc* (*parent-node n''*)›
have *get-proc* (*parent-node mx*) = *p* **by** *simp*
have ∃ *asx*. *targetnode a' −asx→$_{\iota}$* parent-node mx*
proof(*cases* ∀ *a'* ∈ *set asx'*. *intra-kind*(*kind a'*))
case *True*
with ‹(*-Entry-*) *−asx'→$_{\surd}$* parent-node mx*›
have (*-Entry-*) *−asx'→$_{\iota}$* parent-node mx*
by(*simp add*:*vp-def intra-path-def*)
hence *get-proc* (*-Entry-*) = *get-proc* (*parent-node mx*)
by(*rule intra-path-get-procs*)
with ‹*get-proc* (*parent-node mx*) = *p*› **have** *get-proc* (*-Entry-*) = *p*
by *simp*
with ‹*CFG-node* (*parent-node n''*) *s−p→$_{ret}$ CFG-node* (*parent-node n'*)›
have *False*
by −(*erule sum-SDG-edge.cases*,
auto intro:*Main-no-return-source simp*:*get-proc-Entry*)
thus *?thesis* **by** *simp*
next
case *False*
hence ∃ *a'* ∈ *set asx'*. ¬ *intra-kind* (*kind a'*) **by** *fastforce*
then obtain *ai as' as''* **where** *asx'* = *as'*@*ai*#*as''*
and ¬ *intra-kind* (*kind ai*) **and** ∀ *a'* ∈ *set as''*. *intra-kind* (*kind a'*)
by(*fastforce elim!*:*split-list-last-propE*)
from ‹*asx'* = *as'*@*ai*#*as''*› ‹¬ *intra-kind* (*kind ai*)›
‹∀ *a'* ∈ *set asx'*. *intra-kind*(*kind a'*) ∨ (∃ *Q r p fs*. *kind a'* = *Q*:*r*↪$_{p}$*fs*)›
obtain *Qi ri pi fsi* **where** *kind ai* = *Qi*:*ri*↪$_{pi}$*fsi*
and ∀ *a'* ∈ *set as'*. *intra-kind*(*kind a'*) ∨
(∃ *Q r p fs*. *kind a'* = *Q*:*r*↪$_{p}$*fs*)
by *auto*
from ‹(*-Entry-*) *−asx'→$_{\surd}$* parent-node mx*› ‹*asx'* = *as'*@*ai*#*as''*›
‹∀ *a'* ∈ *set as''*. *intra-kind* (*kind a'*)›
have *valid-edge ai* **and** *targetnode ai −as''→$_{\iota}$* parent-node mx*
by(*auto intro*:*path-split simp*:*vp-def intra-path-def*)
hence *get-proc* (*targetnode ai*) = *get-proc* (*parent-node mx*)
by −(*rule intra-path-get-procs*)
with ‹*get-proc* (*parent-node mx*) = *p*› ‹*valid-edge ai*›
‹*kind ai* = *Qi*:*ri*↪$_{pi}$*fsi*›
have [*simp*]:*pi* = *p* **by**(*fastforce dest*:*get-proc-call*)
from ‹*valid-edge ai*› ‹*valid-edge a'*›
‹*kind ai* = *Qi*:*ri*↪$_{pi}$*fsi*› ‹*kind a'* = *Q'*:*r'*↪$_{pfs'}$›
have *targetnode ai* = *targetnode a'*
by(*fastforce intro*:*same-proc-call-unique-target*)

with ‹*targetnode ai* $-as''\rightarrow_\iota*$ *parent-node mx*›
show *?thesis* **by** *fastforce*
qed
then obtain *asx* **where** *targetnode a'* $-asx\rightarrow_\iota*$ *parent-node mx* **by** *blast*
from *this* ‹*valid-edge a'*› ‹*kind a'* $= Q':r'\hookrightarrow_p fs'$›
‹*parent-node mx* $\neq$ (*-Exit-*)› ‹*targetnode a'* $\neq$ *parent-node mx*› *True*
obtain *msi*
where *CFG-node*(*targetnode a'*) *cd*$-msi\rightarrow_d*$ *CFG-node*(*parent-node mx*)
by(*fastforce elim*!:*in-proc-cdep-SDG-path*)
from ‹*valid-SDG-node mx*›
have *mx* = *CFG-node* (*parent-node mx*) $\vee$ *CFG-node* (*parent-node mx*) $\longrightarrow_{cd}$ *mx*
by(*rule valid-SDG-node-cases*)
thus *?thesis*
proof
assume *mx* = *CFG-node* (*parent-node mx*)
with ‹*CFG-node*(*targetnode a'*)*cd*$-msi\rightarrow_d*$ *CFG-node*(*parent-node mx*)›
show *?thesis* **by** *fastforce*
next
assume *CFG-node* (*parent-node mx*) $\longrightarrow_{cd}$ *mx*
with ‹*CFG-node*(*targetnode a'*)*cd*$-msi\rightarrow_d*$ *CFG-node*(*parent-node mx*)›
have *CFG-node*(*targetnode a'*) *cd*$-msi@[$*CFG-node*(*parent-node mx*)$]\rightarrow_d*$ *mx*
by(*fastforce intro*:*cdSp-Append-cdep*)
thus *?thesis* **by** *fastforce*
qed
next
case *False*
then obtain *ai* **where** *valid-edge ai* **and** *sourcenode ai* = *parent-node mx*
and *ai* $\in$ *get-return-edges a'* **by** *blast*
with ‹*valid-edge a'*› ‹*kind a'* $= Q':r'\hookrightarrow_p fs'$›
have *CFG-node* (*targetnode a'*) $\longrightarrow_{cd}$ *CFG-node* (*parent-node mx*)
by(*auto intro*:*SDG-proc-entry-exit-cdep*)
with ‹*valid-edge a'*›
have *cd-path*:*CFG-node* (*targetnode a'*) *cd*$-[]@[$*CFG-node* (*targetnode a'*)$]\rightarrow_d*$
CFG-node (*parent-node mx*)
by(*fastforce intro*:*cdSp-Append-cdep cdSp-Nil*)
from ‹*valid-SDG-node mx*›
have *mx* = *CFG-node* (*parent-node mx*) $\vee$ *CFG-node* (*parent-node mx*) $\longrightarrow_{cd}$ *mx*
by(*rule valid-SDG-node-cases*)
thus *?thesis*
proof
assume *mx* = *CFG-node* (*parent-node mx*)
with *cd-path* **show** *?thesis* **by** *fastforce*
next
assume *CFG-node* (*parent-node mx*) $\longrightarrow_{cd}$ *mx*
with *cd-path* **have** *CFG-node* (*targetnode a'*)

```
        cd−[CFG-node (targetnode a')]@[CFG-node (parent-node mx)]→d* mx
        by(fastforce intro:cdSp-Append-cdep)
      thus ?thesis by fastforce
    qed
  qed
 qed
 then obtain msx''
  where CFG-node (targetnode a') cd−msx''→d* mx by blast
 hence CFG-node (targetnode a') i−msx''→d* mx
  by(rule cdep-SDG-path-intra-SDG-path)
 with ⟨valid-edge a'⟩
 have matched (CFG-node (targetnode a')) ([]@msx'') mx
  by(fastforce intro:matched-Append-intra-SDG-path matched-Nil)
 with ⟨matched mx (msx@msx') n''⟩
 have matched (CFG-node (targetnode a')) (msx''@(msx@msx')) n''
  by(fastforce intro:matched-Append)
with ⟨valid-edge a'⟩ ⟨CFG-node (sourcenode a') s−p→call CFG-node (targetnode
a')⟩
    ⟨n'' −p→ret n' ∨ n'' −p:V→out n'⟩ ⟨a ∈ get-return-edges a'⟩
    ⟨parent-node n'' = sourcenode a⟩ ⟨parent-node n' = targetnode a⟩
 have matched (CFG-node (sourcenode a'))
  ([]@CFG-node (sourcenode a')#(msx''@(msx@msx'))@[n'']) n'
 by(fastforce intro:matched-bracket-call matched-Nil sum-SDG-edge-SDG-edge)
with ⟨n ∈ set msx⟩ ⟨CFG-node (sourcenode a') s−p→sum CFG-node (targetnode
a)⟩
    ⟨parent-node n' = targetnode a⟩
 show ?thesis by fastforce
qed
qed
qed


lemma irs-SDG-path-realizable:
 assumes n irs−ns→d* n' and n ≠ n'
 obtains ns' where realizable (CFG-node (-Entry-)) ns' n' and n ∈ set ns'
proof(atomize-elim)
 from ⟨n irs−ns→d* n'⟩
 have n = n' ∨ (∃ns'. realizable (CFG-node (-Entry-)) ns' n' ∧ n ∈ set ns')
 proof(rule irs-SDG-path-split)
  assume n is−ns→d* n'
  show ?thesis
  proof(cases ns = [])
   case True
  with ⟨n is−ns→d* n'⟩ have n = n' by(fastforce elim:intra-sum-SDG-path.cases)
   thus ?thesis by simp
  next
   case False
   with ⟨n is−ns→d* n'⟩ have n ∈ set ns by(fastforce dest:is-SDG-path-hd)
   from ⟨n is−ns→d* n'⟩ have valid-SDG-node n and valid-SDG-node n'
```

```
      by(rule is-SDG-path-valid-SDG-node)+
    hence valid-node (parent-node n) by −(rule valid-SDG-CFG-node)
   from ‹n is−ns→d* n'› obtain ns' where matched n ns' n' and set ns ⊆ set
ns'
      by(erule is-SDG-path-matched)
    with ‹n ∈ set ns› have n ∈ set ns' by fastforce
    from ‹valid-node (parent-node n)›
    show ?thesis
    proof(cases parent-node n = (-Exit-))
     case True
     with ‹valid-SDG-node n› have n = CFG-node (-Exit-)
      by(rule valid-SDG-node-parent-Exit)
     from ‹n is−ns→d* n'› obtain as where parent-node n −as→ι* parent-node
n'
       by −(erule is-SDG-path-intra-CFG-path)
     with ‹n = CFG-node (-Exit-)› have parent-node n' = (-Exit-)
      by(fastforce dest:path-Exit-source simp:intra-path-def)
     with ‹valid-SDG-node n'› have n' = CFG-node (-Exit-)
      by(rule valid-SDG-node-parent-Exit)
     with ‹n = CFG-node (-Exit-)› show ?thesis by simp
    next
     case False
     with ‹valid-SDG-node n›
     obtain nsx where CFG-node (-Entry-) cc−nsx→d* n
      by(erule Entry-cc-SDG-path-to-inner-node)
     hence realizable (CFG-node (-Entry-)) nsx n
      by(rule cdep-SDG-path-realizable)
     with ‹matched n ns' n'›
     have realizable (CFG-node (-Entry-)) (nsx@ns') n'
      by −(rule realizable-Append-matched)
     with ‹n ∈ set ns'› show ?thesis by fastforce
    qed
  qed
 next
  fix nsx nsx' nx nx' p
  assume 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'
  from ‹nx s−p→ret nx' ∨ (∃ V. nx s−p:V→out nx')›
  have CFG-node (parent-node nx) s−p→ret CFG-node (parent-node nx')
   by(fastforce elim:sum-SDG-edge.cases intro:sum-SDG-return-edge)
  then obtain a Q f where valid-edge a and kind a = Q↩pf
   and parent-node nx = sourcenode a and parent-node nx' = targetnode a
   by(fastforce elim:sum-SDG-edge.cases)
  from ‹valid-edge a› ‹kind a = Q↩pf› obtain a' Q' r' fs'
   where a ∈ get-return-edges a' and valid-edge a' and kind a' = Q':r'↪pfs'
   and CFG-node (sourcenode a') s−p→sum CFG-node (targetnode a)
   by(erule return-edge-determines-call-and-sum-edge)
  from ‹valid-edge a'› ‹kind a' = Q':r'↪pfs'›
  have CFG-node (sourcenode a') s−p→call CFG-node (targetnode a')
```

by(*fastforce intro*:*sum-SDG-call-edge*)
from ‹*n irs*−*nsx*→$_d$* *nx*› ‹*nx s*−*p*→$_{ret}$ *nx'* ∨ (∃ *V*. *nx s*−*p*:*V*→$_{out}$ *nx'*)›
obtain *m ms* **where** *matched m ms nx'* **and** *n* ∈ *set ms*
and *m s*−*p*→$_{sum}$ *CFG-node* (*parent-node nx'*)
by(*fastforce elim*:*irs-SDG-path-matched*)
from ‹*nx' is*−*nsx'*→$_d$* *n'*› **obtain** *ms'* **where** *matched nx' ms' n'*
and *set nsx'* ⊆ *set ms'* **by**(*erule is-SDG-path-matched*)
with ‹*matched m ms nx'*› **have** *matched m* (*ms*@*ms'*) *n'* **by** −(*rule matched-Append*)
from ‹*m s*−*p*→$_{sum}$ *CFG-node* (*parent-node nx'*)› **have** *valid-SDG-node m*
by(*rule sum-SDG-edge-valid-SDG-node*)
hence *valid-node* (*parent-node m*) **by**(*rule valid-SDG-CFG-node*)
thus *?thesis*
proof(*cases parent-node m* = (*-Exit-*))
case *True*
from ‹*m s*−*p*→$_{sum}$ *CFG-node* (*parent-node nx'*)› **obtain** *a* **where** *valid-edge a*
and *sourcenode a* = *parent-node m*
by(*fastforce elim*:*sum-SDG-edge.cases*)
with *True* **have** *False* **by** −(*rule Exit-source*,*simp-all*)
thus *?thesis* **by** *simp*
next
case *False*
with ‹*valid-SDG-node m*›
obtain *ms''* **where** *CFG-node* (*-Entry-*) *cc*−*ms''*→$_d$* *m*
by(*erule Entry-cc-SDG-path-to-inner-node*)
hence *realizable* (*CFG-node* (*-Entry-*)) *ms''* *m*
by(*rule cdep-SDG-path-realizable*)
with ‹*matched m* (*ms*@*ms'*) *n'*›
have *realizable* (*CFG-node* (*-Entry-*)) (*ms''*@(*ms*@*ms'*)) *n'*
by −(*rule realizable-Append-matched*)
with ‹*n* ∈ *set ms*› **show** *?thesis* **by** *fastforce*
qed
qed
with ‹*n* ≠ *n'*› **show** ∃ *ns'*. *realizable* (*CFG-node* (*-Entry-*)) *ns' n'* ∧ *n* ∈ *set ns'*
by *simp*
qed

end

end

1.9 Horwitz-Reps-Binkley Slice

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

context *SDG* **begin**

1.9.1 Set describing phase 1 of the two-phase slicer

inductive-set *sum-SDG-slice1* :: *'node SDG-node* ⇒ *'node SDG-node set*
for *n*::*'node SDG-node*
where *refl-slice1*:*valid-SDG-node n* ⟹ *n* ∈ *sum-SDG-slice1 n*
| *cdep-slice1*:
⟦*n'' s*⟶$_{cd}$ *n'*; *n'* ∈ *sum-SDG-slice1 n*⟧ ⟹ *n''* ∈ *sum-SDG-slice1 n*
| *ddep-slice1*:
⟦*n'' s*−*V*→$_{dd}$ *n'*; *n'* ∈ *sum-SDG-slice1 n*⟧ ⟹ *n''* ∈ *sum-SDG-slice1 n*
| *call-slice1*:
⟦*n'' s*−*p*→$_{call}$ *n'*; *n'* ∈ *sum-SDG-slice1 n*⟧ ⟹ *n''* ∈ *sum-SDG-slice1 n*
| *param-in-slice1*:
⟦*n'' s*−*p*:*V*→$_{in}$ *n'*; *n'* ∈ *sum-SDG-slice1 n*⟧ ⟹ *n''* ∈ *sum-SDG-slice1 n*
| *sum-slice1*:
⟦*n'' s*−*p*→$_{sum}$ *n'*; *n'* ∈ *sum-SDG-slice1 n*⟧ ⟹ *n''* ∈ *sum-SDG-slice1 n*

lemma *slice1-cdep-slice1*:
⟦*nx* ∈ *sum-SDG-slice1 n*; *n s*⟶$_{cd}$ *n'*⟧ ⟹ *nx* ∈ *sum-SDG-slice1 n'*
by(*induct rule*:*sum-SDG-slice1.induct*,
auto intro:*sum-SDG-slice1.intros sum-SDG-edge-valid-SDG-node*)

lemma *slice1-ddep-slice1*:
⟦*nx* ∈ *sum-SDG-slice1 n*; *n s*−*V*→$_{dd}$ *n'*⟧ ⟹ *nx* ∈ *sum-SDG-slice1 n'*
by(*induct rule*:*sum-SDG-slice1.induct*,
auto intro:*sum-SDG-slice1.intros sum-SDG-edge-valid-SDG-node*)

lemma *slice1-sum-slice1*:
⟦*nx* ∈ *sum-SDG-slice1 n*; *n s*−*p*→$_{sum}$ *n'*⟧ ⟹ *nx* ∈ *sum-SDG-slice1 n'*
by(*induct rule*:*sum-SDG-slice1.induct*,
auto intro:*sum-SDG-slice1.intros sum-SDG-edge-valid-SDG-node*)

lemma *slice1-call-slice1*:
⟦*nx* ∈ *sum-SDG-slice1 n*; *n s*−*p*→$_{call}$ *n'*⟧ ⟹ *nx* ∈ *sum-SDG-slice1 n'*
by(*induct rule*:*sum-SDG-slice1.induct*,
auto intro:*sum-SDG-slice1.intros sum-SDG-edge-valid-SDG-node*)

lemma *slice1-param-in-slice1*:
⟦*nx* ∈ *sum-SDG-slice1 n*; *n s*−*p*:*V*→$_{in}$ *n'*⟧ ⟹ *nx* ∈ *sum-SDG-slice1 n'*
by(*induct rule*:*sum-SDG-slice1.induct*,
auto intro:*sum-SDG-slice1.intros sum-SDG-edge-valid-SDG-node*)

lemma *is-SDG-path-slice1*:
⟦*n is*−*ns*→$_{d}$* *n'*; *n'* ∈ *sum-SDG-slice1 n''*⟧ ⟹ *n* ∈ *sum-SDG-slice1 n''*
proof(*induct rule*:*intra-sum-SDG-path.induct*)
case *isSp-Nil* **thus** *?case* **by** *simp*
next
case (*isSp-Append-cdep n ns nx n'*)
note *IH* = ‹*nx* ∈ *sum-SDG-slice1 n''* ⟹ *n* ∈ *sum-SDG-slice1 n''*›

from ‹*nx* $s \longrightarrow_{cd} n'$› ‹$n' \in$ *sum-SDG-slice1* n''›
have $nx \in$ *sum-SDG-slice1* n'' **by**(*rule cdep-slice1*)
from *IH*[*OF this*] **show** *?case* **.**
next
case (*isSp-Append-ddep n ns nx V n'*)
note *IH* = ‹$nx \in$ *sum-SDG-slice1* $n'' \Longrightarrow n \in$ *sum-SDG-slice1* n''›
from ‹*nx* $s-V\rightarrow_{dd} n'$› ‹$n' \in$ *sum-SDG-slice1* n''›
have $nx \in$ *sum-SDG-slice1* n'' **by**(*rule ddep-slice1*)
from *IH*[*OF this*] **show** *?case* **.**
next
case (*isSp-Append-sum n ns nx p n'*)
note *IH* = ‹$nx \in$ *sum-SDG-slice1* $n'' \Longrightarrow n \in$ *sum-SDG-slice1* n''›
from ‹*nx* $s-p\rightarrow_{sum} n'$› ‹$n' \in$ *sum-SDG-slice1* n''›
have $nx \in$ *sum-SDG-slice1* n'' **by**(*rule sum-slice1*)
from *IH*[*OF this*] **show** *?case* **.**
qed

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* ⟹ $n \in$ *sum-SDG-slice2 n*
| *cdep-slice2*:
⟦n'' $s \longrightarrow_{cd} n'$; $n' \in$ *sum-SDG-slice2 n*⟧ ⟹ $n'' \in$ *sum-SDG-slice2 n*
| *ddep-slice2*:
⟦n'' $s-V\rightarrow_{dd} n'$; $n' \in$ *sum-SDG-slice2 n*⟧ ⟹ $n'' \in$ *sum-SDG-slice2 n*
| *return-slice2*:
⟦n'' $s-p\rightarrow_{ret} n'$; $n' \in$ *sum-SDG-slice2 n*⟧ ⟹ $n'' \in$ *sum-SDG-slice2 n*
| *param-out-slice2*:
⟦n'' $s-p$:$V\rightarrow_{out} n'$; $n' \in$ *sum-SDG-slice2 n*⟧ ⟹ $n'' \in$ *sum-SDG-slice2 n*
| *sum-slice2*:
⟦n'' $s-p\rightarrow_{sum} n'$; $n' \in$ *sum-SDG-slice2 n*⟧ ⟹ $n'' \in$ *sum-SDG-slice2 n*

lemma *slice2-cdep-slice2*:
⟦$nx \in$ *sum-SDG-slice2 n*; n $s \longrightarrow_{cd} n'$⟧ ⟹ $nx \in$ *sum-SDG-slice2* n'
by(*induct rule*:*sum-SDG-slice2.induct*,
auto intro:*sum-SDG-slice2.intros sum-SDG-edge-valid-SDG-node*)

lemma *slice2-ddep-slice2*:
⟦$nx \in$ *sum-SDG-slice2 n*; n $s-V\rightarrow_{dd} n'$⟧ ⟹ $nx \in$ *sum-SDG-slice2* n'
by(*induct rule*:*sum-SDG-slice2.induct*,
auto intro:*sum-SDG-slice2.intros sum-SDG-edge-valid-SDG-node*)

lemma *slice2-sum-slice2*:
⟦$nx \in$ *sum-SDG-slice2 n*; n $s-p\rightarrow_{sum} n'$⟧ ⟹ $nx \in$ *sum-SDG-slice2* n'
by(*induct rule*:*sum-SDG-slice2.induct*,
auto intro:*sum-SDG-slice2.intros sum-SDG-edge-valid-SDG-node*)

lemma *slice2-ret-slice2*:
⟦*nx* ∈ *sum-SDG-slice2 n*; *n s*−*p*→$_{ret}$ *n'*⟧ ⟹ *nx* ∈ *sum-SDG-slice2 n'*
by(*induct rule*:*sum-SDG-slice2.induct*,
auto intro:*sum-SDG-slice2.intros sum-SDG-edge-valid-SDG-node*)

lemma *slice2-param-out-slice2*:
⟦*nx* ∈ *sum-SDG-slice2 n*; *n s*−*p*:*V*→$_{out}$ *n'*⟧ ⟹ *nx* ∈ *sum-SDG-slice2 n'*
by(*induct rule*:*sum-SDG-slice2.induct*,
auto intro:*sum-SDG-slice2.intros sum-SDG-edge-valid-SDG-node*)

lemma *is-SDG-path-slice2*:
⟦*n is*−*ns*→$_d$* *n'*; *n'* ∈ *sum-SDG-slice2 n''*⟧ ⟹ *n* ∈ *sum-SDG-slice2 n''*
proof(*induct rule*:*intra-sum-SDG-path.induct*)
case *isSp-Nil* **thus** *?case* **by** *simp*
next
case (*isSp-Append-cdep n ns nx n'*)
note *IH* = ‹*nx* ∈ *sum-SDG-slice2 n''* ⟹ *n* ∈ *sum-SDG-slice2 n''*›
from ‹*nx s*⟶$_{cd}$ *n'*› ‹*n'* ∈ *sum-SDG-slice2 n''*›
have *nx* ∈ *sum-SDG-slice2 n''* **by**(*rule cdep-slice2*)
from *IH*[*OF this*] **show** *?case* **.**
next
case (*isSp-Append-ddep n ns nx V n'*)
note *IH* = ‹*nx* ∈ *sum-SDG-slice2 n''* ⟹ *n* ∈ *sum-SDG-slice2 n''*›
from ‹*nx s*−*V*→$_{dd}$ *n'*› ‹*n'* ∈ *sum-SDG-slice2 n''*›
have *nx* ∈ *sum-SDG-slice2 n''* **by**(*rule ddep-slice2*)
from *IH*[*OF this*] **show** *?case* **.**
next
case (*isSp-Append-sum n ns nx p n'*)
note *IH* = ‹*nx* ∈ *sum-SDG-slice2 n''* ⟹ *n* ∈ *sum-SDG-slice2 n''*›
from ‹*nx s*−*p*→$_{sum}$ *n'*› ‹*n'* ∈ *sum-SDG-slice2 n''*›
have *nx* ∈ *sum-SDG-slice2 n''* **by**(*rule sum-slice2*)
from *IH*[*OF this*] **show** *?case* **.**
qed

lemma *slice2-is-SDG-path-slice2*:
⟦*n is*−*ns*→$_d$* *n'*; *n''* ∈ *sum-SDG-slice2 n*⟧ ⟹ *n''* ∈ *sum-SDG-slice2 n'*
proof(*induct rule*:*intra-sum-SDG-path.induct*)
case *isSp-Nil* **thus** *?case* **by** *simp*
next
case (*isSp-Append-cdep n ns nx n'*)
from ‹*n''* ∈ *sum-SDG-slice2 n* ⟹ *n''* ∈ *sum-SDG-slice2 nx*› ‹*n''* ∈ *sum-SDG-slice2 n*›
have *n''* ∈ *sum-SDG-slice2 nx* **.**
with ‹*nx s*⟶$_{cd}$ *n'*› **show** *?case* **by** −(*rule slice2-cdep-slice2*)
next
case (*isSp-Append-ddep n ns nx V n'*)

from ‹$n'' \in$ *sum-SDG-slice2* $n \Longrightarrow n'' \in$ *sum-SDG-slice2* nx› ‹$n'' \in$ *sum-SDG-slice2* n›
have $n'' \in$ *sum-SDG-slice2* nx **.**
with ‹nx $s{-}V{\rightarrow}_{dd}$ n'› **show** *?case* **by** −(*rule slice2-ddep-slice2*)
next
case (*isSp-Append-sum* n ns nx p n')
from ‹$n'' \in$ *sum-SDG-slice2* $n \Longrightarrow n'' \in$ *sum-SDG-slice2* nx› ‹$n'' \in$ *sum-SDG-slice2* n›
have $n'' \in$ *sum-SDG-slice2* nx **.**
with ‹nx $s{-}p{\rightarrow}_{sum}$ n'› **show** *?case* **by** −(*rule slice2-sum-slice2*)
qed

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

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

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

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

lemma *HRB-slice-cases*[*consumes 1*,*case-names phase1 phase2*]:
⟦$x \in$ *HRB-slice S*; ⋀n nx. ⟦$n \in$ *sum-SDG-slice1* nx; $nx \in S$⟧ ⟹ *P n*;
⋀nx n' n'' p n. ⟦$n' \in$ *sum-SDG-slice1* nx; n'' $s{-}p{\rightarrow}_{ret}$ *CFG-node* (*parent-node* n');
$n \in$ *sum-SDG-slice2* n'; $nx \in S$⟧ ⟹ *P n*⟧
⟹ *P x*
by(*fastforce elim*:*combine-SDG-slices.cases simp*:*HRB-slice-def*)

lemma *HRB-slice-refl*:
assumes *valid-node m* **and** *CFG-node m* ∈ *S* **shows** *CFG-node m* ∈ *HRB-slice S*
proof −
from ‹*valid-node m*› **have** *CFG-node m* ∈ *sum-SDG-slice1* (*CFG-node m*)
by(*fastforce intro*:*refl-slice1*)
with ‹*CFG-node m* ∈ *S*› **show** *?thesis*
by(*simp add*:*HRB-slice-def*)(*blast intro*:*combSlice-refl*)
qed

lemma *HRB-slice-valid-node*: $n \in$ *HRB-slice* $S \Longrightarrow$ *valid-SDG-node* n
proof(*induct rule*:*HRB-slice-cases*)
case (*phase1 n nx*) **thus** *?case*
by(*induct rule*:*sum-SDG-slice1.induct*,*auto intro*:*sum-SDG-edge-valid-SDG-node*)
next
case (*phase2 nx n' n'' p n*)
from ‹$n \in$ *sum-SDG-slice2* n'›
show *?case*
by(*induct rule*:*sum-SDG-slice2.induct*,*auto intro*:*sum-SDG-edge-valid-SDG-node*)
qed

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 −
from ‹$n \in$ *HRB-slice* S› **have** *valid-SDG-node* n **by**(*rule HRB-slice-valid-node*)
hence $n =$ *CFG-node* (*parent-node* n) $\vee$ *CFG-node* (*parent-node* n) $\longrightarrow_{cd} n$
by(*rule valid-SDG-node-cases*)
thus *?thesis*
proof
assume $n =$ *CFG-node* (*parent-node* n)
with ‹$n \in$ *HRB-slice* S› **show** *?thesis* **by** *simp*
next
assume *CFG-node* (*parent-node* n) $\longrightarrow_{cd} n$
hence *CFG-node* (*parent-node* n) $s\longrightarrow_{cd} n$ **by**(*rule SDG-edge-sum-SDG-edge*)
with ‹$n \in$ *HRB-slice* S› **show** *?thesis*
by(*fastforce elim*:*combine-SDG-slices.cases*
intro:*combine-SDG-slices.intros cdep-slice1 cdep-slice2*
simp:*HRB-slice-def*)
qed
qed

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(*induct arbitrary*:S *rule*:*intra-sum-SDG-path.induct*)
case (*isSp-Nil n*) **thus** *?case* **by**(*fastforce simp*:*HRB-slice-def*)
next
case (*isSp-Append-cdep n ns nx n'*)
note $IH = ‹\bigwedge S.\ [\![n'' \in$ *HRB-slice* $\{n\}; nx \in S]\!] \Longrightarrow n'' \in$ *HRB-slice* S›
from $IH[OF$ ‹$n'' \in$ *HRB-slice* $\{n\}$›$]$ **have** $n'' \in$ *HRB-slice* $\{nx\}$ **by** *simp*
thus *?case*
proof(*induct rule*:*HRB-slice-cases*)
case (*phase1 n nx'*)
from ‹$nx' \in \{nx\}$›**have** $nx' = nx$ **by** *simp*
with ‹$n \in$ *sum-SDG-slice1* nx'› ‹$nx\ s\longrightarrow_{cd} n'$› **have** $n \in$ *sum-SDG-slice1* n'
by(*fastforce intro*:*slice1-cdep-slice1*)
with ‹$n' \in S$› **show** *?case*

```
    by(fastforce intro:combine-SDG-slices.combSlice-refl simp:HRB-slice-def)
  next
    case (phase2 nx'' nx' n'' p n)
    from ⟨nx'' ∈ {nx}⟩ have nx'' = nx by simp
    with ⟨nx' ∈ sum-SDG-slice1 nx''⟩ ⟨nx s⟶_cd n'⟩ have nx' ∈ sum-SDG-slice1
n'
      by(fastforce intro:slice1-cdep-slice1)
    with ⟨n'' s−p→_ret CFG-node (parent-node nx')⟩ ⟨n ∈ sum-SDG-slice2 nx'⟩ ⟨n'
∈ S⟩
    show ?case by(fastforce intro:combine-SDG-slices.combSlice-Return-parent-node

                                simp:HRB-slice-def)
  qed
next
  case (isSp-Append-ddep n ns nx V n')
  note IH = ⟨⋀S. ⟦n'' ∈ HRB-slice {n}; nx ∈ S⟧ ⟹ n'' ∈ HRB-slice S⟩
  from IH[OF ⟨n'' ∈ HRB-slice {n}⟩] have n'' ∈ HRB-slice {nx} by simp
  thus ?case
  proof(induct rule:HRB-slice-cases)
    case (phase1 n nx')
    from ⟨nx' ∈ {nx}⟩ have nx' = nx by simp
    with ⟨n ∈ sum-SDG-slice1 nx'⟩ ⟨nx s−V→_dd n'⟩ have n ∈ sum-SDG-slice1 n'
      by(fastforce intro:slice1-ddep-slice1)
    with ⟨n' ∈ S⟩ show ?case
      by(fastforce intro:combine-SDG-slices.combSlice-refl simp:HRB-slice-def)
  next
    case (phase2 nx'' nx' n'' p n)
    from ⟨nx'' ∈ {nx}⟩ have nx'' = nx by simp
    with ⟨nx' ∈ sum-SDG-slice1 nx''⟩ ⟨nx s−V→_dd n'⟩ have nx' ∈ sum-SDG-slice1
n'
      by(fastforce intro:slice1-ddep-slice1)
    with ⟨n'' s−p→_ret CFG-node (parent-node nx')⟩ ⟨n ∈ sum-SDG-slice2 nx'⟩ ⟨n'
∈ S⟩
    show ?case by(fastforce intro:combine-SDG-slices.combSlice-Return-parent-node

                                simp:HRB-slice-def)
  qed
next
  case (isSp-Append-sum n ns nx p n')
  note IH = ⟨⋀S. ⟦n'' ∈ HRB-slice {n}; nx ∈ S⟧ ⟹ n'' ∈ HRB-slice S⟩
  from IH[OF ⟨n'' ∈ HRB-slice {n}⟩] have n'' ∈ HRB-slice {nx} by simp
  thus ?case
  proof(induct rule:HRB-slice-cases)
    case (phase1 n nx')
    from ⟨nx' ∈ {nx}⟩ have nx' = nx by simp
    with ⟨n ∈ sum-SDG-slice1 nx'⟩ ⟨nx s−p→_sum n'⟩ have n ∈ sum-SDG-slice1
n'
      by(fastforce intro:slice1-sum-slice1)
    with ⟨n' ∈ S⟩ show ?case
```

by(*fastforce intro*:*combine-SDG-slices.combSlice-refl simp*:*HRB-slice-def*)
next
case (*phase2 nx'' nx' n'' p' n*)
from ‹*nx''* ∈ {*nx*}› **have** *nx''* = *nx* **by** *simp*
with ‹*nx'* ∈ *sum-SDG-slice1 nx''*› ‹*nx s*−*p*→$_{sum}$ *n'*› **have** *nx'* ∈ *sum-SDG-slice1 n'*
by(*fastforce intro*:*slice1-sum-slice1*)
with ‹*n''* *s*−*p'*→$_{ret}$ *CFG-node* (*parent-node nx'*)› ‹*n* ∈ *sum-SDG-slice2 nx'*› ‹*n'* ∈ *S*›
show *?case* **by**(*fastforce intro*:*combine-SDG-slices.combSlice-Return-parent-node simp*:*HRB-slice-def*)
qed
qed

lemma *call-return-nodes-in-slice*:
assumes *valid-edge a* **and** *kind a* = *Q*↩$_{pf}$
and *valid-edge a'* **and** *kind a'* = *Q'*:*r'*↪$_{pfs'}$ **and** *a* ∈ *get-return-edges a'*
and *CFG-node* (*targetnode a*) ∈ *HRB-slice S*
shows *CFG-node* (*sourcenode a*) ∈ *HRB-slice S*
and *CFG-node* (*sourcenode a'*) ∈ *HRB-slice S*
and *CFG-node* (*targetnode a'*) ∈ *HRB-slice S*
proof −
from ‹*valid-edge a'*› ‹*kind a'* = *Q'*:*r'*↪$_{pfs'}$› ‹*a* ∈ *get-return-edges a'*›
have *CFG-node* (*sourcenode a'*) *s*−*p*→$_{sum}$ *CFG-node* (*targetnode a*)
by(*fastforce intro*:*sum-SDG-call-summary-edge*)
with ‹*CFG-node* (*targetnode a*) ∈ *HRB-slice S*›
show *CFG-node* (*sourcenode a'*) ∈ *HRB-slice S*
by(*fastforce elim*!:*combine-SDG-slices.cases intro*:*combine-SDG-slices.intros sum-slice1 sum-slice2 simp*:*HRB-slice-def*)
from ‹*CFG-node* (*targetnode a*) ∈ *HRB-slice S*›
obtain n_c **where** *CFG-node* (*targetnode a*) ∈ *combine-SDG-slices* (*sum-SDG-slice1* n_c)
and n_c ∈ *S*
by(*simp add*:*HRB-slice-def*) *blast*
thus *CFG-node* (*sourcenode a*) ∈ *HRB-slice S*
proof(*induct CFG-node* (*targetnode a*) *rule*:*combine-SDG-slices.induct*)
case *combSlice-refl*
from ‹*valid-edge a*› ‹*kind a* = *Q*↩$_{pf}$›
have *CFG-node* (*sourcenode a*) *s*−*p*→$_{ret}$ *CFG-node* (*targetnode a*)
by(*fastforce intro*:*sum-SDG-return-edge*)
with ‹*valid-edge a*›
have *CFG-node* (*sourcenode a*) ∈ *sum-SDG-slice2* (*CFG-node* (*targetnode a*))
by(*fastforce intro*:*sum-SDG-slice2.intros*)
with ‹*CFG-node* (*targetnode a*) ∈ *sum-SDG-slice1* n_c› ‹n_c ∈ *S*›
‹*CFG-node* (*sourcenode a*) *s*−*p*→$_{ret}$ *CFG-node* (*targetnode a*)›
show *?case* **by**(*fastforce intro*:*combSlice-Return-parent-node simp*:*HRB-slice-def*)

next
case (*combSlice-Return-parent-node* n' n'' p')
from ‹*valid-edge a*› ‹*kind a* = $Q\hookleftarrow_p f$›
have *CFG-node* (*sourcenode a*) $s-p\rightarrow_{ret}$ *CFG-node* (*targetnode a*)
by(*fastforce intro*:*sum-SDG-return-edge*)
with ‹*CFG-node* (*targetnode a*) ∈ *sum-SDG-slice2* n'›
have *CFG-node* (*sourcenode a*) ∈ *sum-SDG-slice2* n'
by(*fastforce intro*:*sum-SDG-slice2.intros*)
with ‹n' ∈ *sum-SDG-slice1* n_c› ‹n'' $s-p'\rightarrow_{ret}$ *CFG-node* (*parent-node* n')› ‹n_c ∈ S›
show *?case* **by**(*fastforce intro*:*combine-SDG-slices.combSlice-Return-parent-node*
simp:*HRB-slice-def*)
qed
from ‹*valid-edge a'*› ‹*kind a'* = Q':$r'\hookrightarrow_p fs'$› ‹*a* ∈ *get-return-edges a'*›
have *CFG-node* (*targetnode a'*) $s\longrightarrow_{cd}$ *CFG-node* (*sourcenode a*)
by(*fastforce intro*:*sum-SDG-proc-entry-exit-cdep*)
with ‹*CFG-node* (*sourcenode a*) ∈ *HRB-slice S*› ‹n_c ∈ S›
show *CFG-node* (*targetnode a'*) ∈ *HRB-slice S*
by(*fastforce elim*!:*combine-SDG-slices.cases*
intro:*combine-SDG-slices.intros cdep-slice1 cdep-slice2*
simp:*HRB-slice-def*)
qed

1.9.4 Proof of Precision

lemma *in-intra-SDG-path-in-slice2*:
⟦n $i-ns\rightarrow_d*$ n'; n'' ∈ *set ns*⟧ ⟹ n'' ∈ *sum-SDG-slice2* n'
proof(*induct rule*:*intra-SDG-path.induct*)
case *iSp-Nil* **thus** *?case* **by** *simp*
next
case (*iSp-Append-cdep n ns nx n'*)
note *IH* = ‹n'' ∈ *set ns* ⟹ n'' ∈ *sum-SDG-slice2 nx*›
from ‹n'' ∈ *set* (*ns*@[*nx*])› **have** n'' ∈ *set ns* ∨ n'' = *nx* **by** *auto*
thus *?case*
proof
assume n'' ∈ *set ns*
from *IH*[*OF this*] **have** n'' ∈ *sum-SDG-slice2 nx* **by** *simp*
with ‹*nx* $\longrightarrow_{cd}$ n'› **show** *?thesis*
by(*fastforce intro*:*slice2-cdep-slice2 SDG-edge-sum-SDG-edge*)
next
assume n'' = *nx*
from ‹*nx* $\longrightarrow_{cd}$ n'› **have** *valid-SDG-node* n' **by**(*rule SDG-edge-valid-SDG-node*)
hence n' ∈ *sum-SDG-slice2* n' **by**(*rule refl-slice2*)
with ‹*nx* $\longrightarrow_{cd}$ n'› **have** *nx* ∈ *sum-SDG-slice2* n'
by(*fastforce intro*:*cdep-slice2 SDG-edge-sum-SDG-edge*)
with ‹n'' = *nx*› **show** *?thesis* **by** *simp*
qed
next

case (*iSp-Append-ddep n ns nx V n'*)
note *IH* = ‹$n'' \in set\ ns \Longrightarrow n'' \in$ *sum-SDG-slice2 nx*›
from ‹$n'' \in set\ (ns@[nx])$› **have** $n'' \in set\ ns \vee n'' = nx$ **by** *auto*
thus *?case*
proof
assume $n'' \in set\ ns$
from *IH*[*OF this*] **have** $n'' \in$ *sum-SDG-slice2 nx* **by** *simp*
with ‹$nx -V\rightarrow_{dd} n'$› **show** *?thesis*
by(*fastforce intro*:*slice2-ddep-slice2 SDG-edge-sum-SDG-edge*)
next
assume $n'' = nx$
from ‹$nx -V\rightarrow_{dd} n'$› **have** *valid-SDG-node n'* **by**(*rule SDG-edge-valid-SDG-node*)
hence $n' \in$ *sum-SDG-slice2 n'* **by**(*rule refl-slice2*)
with ‹$nx -V\rightarrow_{dd} n'$› **have** $nx \in$ *sum-SDG-slice2 n'*
by(*fastforce intro*:*ddep-slice2 SDG-edge-sum-SDG-edge*)
with ‹$n'' = nx$› **show** *?thesis* **by** *simp*
qed
qed

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(*induct arbitrary*:*S rule*:*intra-SDG-path.induct*)
case *iSp-Nil* **thus** *?case* **by** *simp*
next
case (*iSp-Append-cdep n ns nx n'*)
note *IH* = ‹$\bigwedge S.\ [\![n'' \in set\ ns;\ nx \in S]\!] \Longrightarrow n'' \in$ *HRB-slice S*›
from ‹$n'' \in set\ (ns@[nx])$› **have** $n'' \in set\ ns \vee n'' = nx$ **by** *auto*
thus *?case*
proof
assume $n'' \in set\ ns$
from *IH*[*OF* ‹$n'' \in set\ ns$›] **have** $n'' \in$ *HRB-slice* $\{nx\}$ **by** *simp*
from *this* ‹$nx \longrightarrow_{cd} n'$› ‹$n' \in S$› **show** *?case*
by(*fastforce elim*:*HRB-slice-cases slice1-cdep-slice1*
intro:*bexI*[**where** $x{=}n'$] *combine-SDG-slices.intros SDG-edge-sum-SDG-edge*

simp:*HRB-slice-def*)
next
assume $n'' = nx$
from ‹$nx \longrightarrow_{cd} n'$› **have** *valid-SDG-node n'* **by**(*rule SDG-edge-valid-SDG-node*)
hence $n' \in$ *sum-SDG-slice1 n'* **by**(*rule refl-slice1*)
with ‹$nx \longrightarrow_{cd} n'$› **have** $nx \in$ *sum-SDG-slice1 n'*
by(*fastforce intro*:*cdep-slice1 SDG-edge-sum-SDG-edge*)
with ‹$n'' = nx$› ‹$n' \in S$› **show** *?case*
by(*fastforce intro*:*combSlice-refl simp*:*HRB-slice-def*)
qed
next
case (*iSp-Append-ddep n ns nx V n'*)
note *IH* = ‹$\bigwedge S.\ [\![n'' \in set\ ns;\ nx \in S]\!] \Longrightarrow n'' \in$ *HRB-slice S*›

from ‹$n'' \in set\ (ns@[nx])$› **have** $n'' \in set\ ns \vee n'' = nx$ **by** *auto*
thus *?case*
proof
assume $n'' \in set\ ns$
from *IH*[*OF* ‹$n'' \in set\ ns$›] **have** $n'' \in HRB\text{-}slice\ \{nx\}$ **by** *simp*
from *this* ‹$nx\ -V\rightarrow_{dd}\ n'$› ‹$n' \in S$› **show** *?case*
by(*fastforce elim*:*HRB-slice-cases slice1-ddep-slice1*
intro:*bexI*[**where** $x{=}n'$] *combine-SDG-slices.intros SDG-edge-sum-SDG-edge*

simp:*HRB-slice-def*)
next
assume $n'' = nx$
from ‹$nx\ -V\rightarrow_{dd}\ n'$› **have** *valid-SDG-node* n' **by**(*rule SDG-edge-valid-SDG-node*)
hence $n' \in$ *sum-SDG-slice1* n' **by**(*rule refl-slice1*)
with ‹$nx\ -V\rightarrow_{dd}\ n'$› **have** $nx \in$ *sum-SDG-slice1* n'
by(*fastforce intro*:*ddep-slice1 SDG-edge-sum-SDG-edge*)
with ‹$n'' = nx$› ‹$n' \in S$› **show** *?case*
by(*fastforce intro*:*combSlice-refl simp*:*HRB-slice-def*)
qed
qed

lemma *in-matched-in-slice2*:
⟦*matched n ns n'*; $n'' \in set\ ns$⟧ $\Longrightarrow$ $n'' \in$ *sum-SDG-slice2* n'
proof(*induct rule*:*matched.induct*)
case *matched-Nil* **thus** *?case* **by** *simp*
next
case (*matched-Append-intra-SDG-path n ns nx ns' n'*)
note *IH* = ‹$n'' \in set\ ns \Longrightarrow n'' \in$ *sum-SDG-slice2 nx*›
from ‹$n'' \in set\ (ns@ns')$› **have** $n'' \in set\ ns \vee n'' \in set\ ns'$ **by** *simp*
thus *?case*
proof
assume $n'' \in set\ ns$
from *IH*[*OF this*] **have** $n'' \in$ *sum-SDG-slice2 nx* **.**
with ‹$nx\ i{-}ns'\rightarrow_{d}*\ n'$› **show** *?thesis*
by(*fastforce intro*:*slice2-is-SDG-path-slice2*
intra-SDG-path-is-SDG-path)
next
assume $n'' \in set\ ns'$
with ‹$nx\ i{-}ns'\rightarrow_{d}*\ n'$› **show** *?case* **by**(*rule in-intra-SDG-path-in-slice2*)
qed
next
case (*matched-bracket-call* $n_0\ ns\ n_1\ p\ n_2\ ns'\ n_3\ n_4\ V\ a\ a'$)
note *IH1* = ‹$n'' \in set\ ns \Longrightarrow n'' \in$ *sum-SDG-slice2* n_1›
note *IH2* = ‹$n'' \in set\ ns' \Longrightarrow n'' \in$ *sum-SDG-slice2* n_3›
from ‹$n_1\ -p\rightarrow_{call}\ n_2$› ‹*matched* $n_2\ ns'\ n_3$› ‹$n_3\ -p\rightarrow_{ret}\ n_4 \vee n_3\ -p{:}V\rightarrow_{out}\ n_4$›
‹$a' \in$ *get-return-edges a*› ‹*valid-edge 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›

have *matched* n_1 $([]@n_1\#ns'@[n_3])$ n_4
by(*fastforce intro*:*matched.matched-bracket-call matched-Nil elim*:*SDG-edge-valid-SDG-node*)
then obtain *nsx* **where** n_1 *is*$-nsx\to_d*$ n_4 **by**(*erule matched-is-SDG-path*)
from ‹n'' ∈ *set* $(ns@n_1\#ns'@[n_3])$›
have $((n'' \in set\ ns \vee n'' = n_1) \vee n'' \in set\ ns') \vee n'' = n_3$ **by** *auto*
thus *?case* **apply** −
proof(*erule disjE*)+
assume $n'' \in set\ ns$
from *IH1*[*OF this*] **have** $n'' \in$ *sum-SDG-slice2* n_1 **.**
with ‹n_1 *is*$-nsx\to_d*$ n_4› **show** *?thesis*
by −(*rule slice2-is-SDG-path-slice2*)
next
assume $n'' = n_1$
from ‹n_1 *is*$-nsx\to_d*$ n_4› **have** $n_1 \in$ *sum-SDG-slice2* n_4
by(*fastforce intro*:*is-SDG-path-slice2 refl-slice2 is-SDG-path-valid-SDG-node*)
with ‹$n'' = n_1$› **show** *?thesis* **by**(*fastforce intro*:*combSlice-refl simp*:*HRB-slice-def*)
next
assume $n'' \in set\ ns'$
from *IH2*[*OF this*] **have** $n'' \in$ *sum-SDG-slice2* n_3 **.**
with ‹n_3 $-p\to_{ret}$ $n_4 \vee n_3$ $-p$:$V\to_{out}$ n_4› **show** *?thesis*
by(*fastforce intro*:*slice2-ret-slice2 slice2-param-out-slice2 SDG-edge-sum-SDG-edge*)
next
assume $n'' = n_3$
from ‹n_3 $-p\to_{ret}$ $n_4 \vee n_3$ $-p$:$V\to_{out}$ n_4› **have** n_3 $s-p\to_{ret}$ $n_4 \vee n_3$ $s-p$:$V\to_{out}$ n_4
by(*fastforce intro*:*SDG-edge-sum-SDG-edge*)
hence $n_3 \in$ *sum-SDG-slice2* n_4
by(*fastforce intro*:*return-slice2 param-out-slice2 refl-slice2 sum-SDG-edge-valid-SDG-node*)
with ‹$n'' = n_3$› **show** *?thesis* **by** *simp*
qed
next
case (*matched-bracket-param* n_0 *ns* n_1 *p* *V* n_2 *ns'* n_3 *V'* n_4 *a a'*)
note *IH1* = ‹$n'' \in set\ ns \Longrightarrow n'' \in$ *sum-SDG-slice2* n_1›
note *IH2* = ‹$n'' \in set\ ns' \Longrightarrow n'' \in$ *sum-SDG-slice2* n_3›
from ‹n_1 $-p$:$V\to_{in}$ n_2› ‹*matched* n_2 *ns'* n_3› ‹n_3 $-p$:$V'\to_{out}$ n_4›
‹$a' \in$ *get-return-edges a*› ‹*valid-edge 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›
have *matched* n_1 $([]@n_1\#ns'@[n_3])$ n_4
by(*fastforce intro*:*matched.matched-bracket-param matched-Nil elim*:*SDG-edge-valid-SDG-node*)
then obtain *nsx* **where** n_1 *is*$-nsx\to_d*$ n_4 **by**(*erule matched-is-SDG-path*)
from ‹$n'' \in set\ (ns@n_1\#ns'@[n_3])$›
have $((n'' \in set\ ns \vee n'' = n_1) \vee n'' \in set\ ns') \vee n'' = n_3$ **by** *auto*
thus *?case* **apply** −
proof(*erule disjE*)+

assume $n'' \in set\ ns$
from *IH1*[*OF this*] **have** $n'' \in$ *sum-SDG-slice2* n_1 **.**
with $\langle n_1\ is{-}nsx{\to_d}*\ n_4\rangle$ **show** *?thesis*
by −(*rule slice2-is-SDG-path-slice2*)
next
assume $n'' = n_1$
from $\langle n_1\ is{-}nsx{\to_d}*\ n_4\rangle$ **have** $n_1 \in$ *sum-SDG-slice2* n_4
by(*fastforce intro*:*is-SDG-path-slice2 refl-slice2 is-SDG-path-valid-SDG-node*)
with $\langle n'' = n_1\rangle$ **show** *?thesis* **by**(*fastforce intro*:*combSlice-refl simp*:*HRB-slice-def*)
next
assume $n'' \in set\ ns'$
from *IH2*[*OF this*] **have** $n'' \in$ *sum-SDG-slice2* n_3 **.**
with $\langle n_3\ {-}p{:}V'{\to_{out}}\ n_4\rangle$ **show** *?thesis*
by(*fastforce intro*:*slice2-param-out-slice2 SDG-edge-sum-SDG-edge*)
next
assume $n'' = n_3$
from $\langle n_3\ {-}p{:}V'{\to_{out}}\ n_4\rangle$ **have** $n_3\ s{-}p{:}V'{\to_{out}}\ n_4$ **by**(*rule SDG-edge-sum-SDG-edge*)
hence $n_3 \in$ *sum-SDG-slice2* n_4
by(*fastforce intro*:*param-out-slice2 refl-slice2 sum-SDG-edge-valid-SDG-node*)
with $\langle n'' = n_3\rangle$ **show** *?thesis* **by** *simp*
qed
qed

lemma *in-matched-in-HRB-slice*:
$[\![matched\ n\ ns\ n';\ n'' \in set\ ns;\ n' \in S]\!] \Longrightarrow n'' \in$ *HRB-slice* S
proof(*induct arbitrary*:S *rule*:*matched.induct*)
case *matched-Nil* **thus** *?case* **by** *simp*
next
case (*matched-Append-intra-SDG-path n ns nx ns' n'*)
note $IH = \langle\bigwedge S.\ [\![n'' \in set\ ns;\ nx \in S]\!] \Longrightarrow n'' \in$ *HRB-slice* $S\rangle$
from $\langle n'' \in set\ (ns@ns')\rangle$ **have** $n'' \in set\ ns \vee n'' \in set\ ns'$ **by** *simp*
thus *?case*
proof
assume $n'' \in set\ ns$
from IH[*OF* $\langle n'' \in set\ ns\rangle$] **have** $n'' \in$ *HRB-slice* $\{nx\}$ **by** *simp*
with $\langle nx\ i{-}ns'{\to_d}*\ n'\rangle\ \langle n' \in S\rangle$ **show** *?thesis*
by(*fastforce intro*:*HRB-slice-is-SDG-path-HRB-slice*
intra-SDG-path-is-SDG-path)
next
assume $n'' \in set\ ns'$
with $\langle nx\ i{-}ns'{\to_d}*\ n'\rangle\ \langle n' \in S\rangle$ **show** *?case*
by(*fastforce intro*:*in-intra-SDG-path-in-HRB-slice simp*:*HRB-slice-def*)
qed
next
case (*matched-bracket-call* $n_0\ ns\ n_1\ p\ n_2\ ns'\ n_3\ n_4\ V\ a\ a'$)
note $IH1 = \langle\bigwedge S.\ [\![n'' \in set\ ns;\ n_1 \in S]\!] \Longrightarrow n'' \in$ *HRB-slice* $S\rangle$
note $IH2 = \langle\bigwedge S.\ [\![n'' \in set\ ns';\ n_3 \in S]\!] \Longrightarrow n'' \in$ *HRB-slice* $S\rangle$
from $\langle n_1\ {-}p{\to_{call}}\ n_2\rangle\ \langle matched\ n_2\ ns'\ n_3\rangle\ \langle n_3\ {-}p{\to_{ret}}\ n_4 \vee n_3\ {-}p{:}V{\to_{out}}\ n_4\rangle$

```
  ‹a' ∈ get-return-edges a› ‹valid-edge a›
  ‹sourcenode a = parent-node n1› ‹targetnode a = parent-node n2›
  ‹sourcenode a' = parent-node n3› ‹targetnode a' = parent-node n4›
have matched n1 ([]@n1#ns'@[n3]) n4
  by(fastforce intro:matched.matched-bracket-call matched-Nil
              elim:SDG-edge-valid-SDG-node)
then obtain nsx where n1 is−nsx→d* n4 by(erule matched-is-SDG-path)
from ‹n'' ∈ set (ns@n1#ns'@[n3])›
have ((n'' ∈ set ns ∨ n'' = n1) ∨ n'' ∈ set ns') ∨ n'' = n3 by auto
thus ?case apply −
proof(erule disjE)+
  assume n'' ∈ set ns
  from IH1[OF this] have n'' ∈ HRB-slice {n1} by simp
  with ‹n1 is−nsx→d* n4› ‹n4 ∈ S› show ?thesis
    by −(rule HRB-slice-is-SDG-path-HRB-slice)
next
  assume n'' = n1
  from ‹n1 is−nsx→d* n4› have n1 ∈ sum-SDG-slice1 n4
    by(fastforce intro:is-SDG-path-slice1 refl-slice1 is-SDG-path-valid-SDG-node)
  with ‹n'' = n1› ‹n4 ∈ S› show ?thesis
    by(fastforce intro:combSlice-refl simp:HRB-slice-def)
next
  assume n'' ∈ set ns'
  with ‹matched n2 ns' n3› have n'' ∈ sum-SDG-slice2 n3
    by(rule in-matched-in-slice2)
  with ‹n3 −p→ret n4 ∨ n3 −p:V→out n4› have n'' ∈ sum-SDG-slice2 n4
    by(fastforce intro:slice2-ret-slice2 slice2-param-out-slice2
                       SDG-edge-sum-SDG-edge)
  from ‹n3 −p→ret n4 ∨ n3 −p:V→out n4› have valid-SDG-node n4
    by(fastforce intro:SDG-edge-valid-SDG-node)
  hence n4 ∈ sum-SDG-slice1 n4 by(rule refl-slice1)
  from ‹n3 −p→ret n4 ∨ n3 −p:V→out n4›
  have CFG-node (parent-node n3) −p→ret CFG-node (parent-node n4)
    by(fastforce elim:SDG-edge.cases intro:SDG-return-edge)
  with ‹n'' ∈ sum-SDG-slice2 n4› ‹n4 ∈ sum-SDG-slice1 n4› ‹n4 ∈ S›
 show ?case by(fastforce intro:combSlice-Return-parent-node SDG-edge-sum-SDG-edge

                         simp:HRB-slice-def)
next
  assume n'' = n3
  from ‹n3 −p→ret n4 ∨ n3 −p:V→out n4›
  have CFG-node (parent-node n3) −p→ret CFG-node (parent-node n4)
    by(fastforce elim:SDG-edge.cases intro:SDG-return-edge)
  from ‹n3 −p→ret n4 ∨ n3 −p:V→out n4› have valid-SDG-node n4
    by(fastforce intro:SDG-edge-valid-SDG-node)
  hence n4 ∈ sum-SDG-slice1 n4 by(rule refl-slice1)
  from ‹valid-SDG-node n4› have n4 ∈ sum-SDG-slice2 n4 by(rule refl-slice2)
  with ‹n3 −p→ret n4 ∨ n3 −p:V→out n4› have n3 ∈ sum-SDG-slice2 n4
    by(fastforce intro:return-slice2 param-out-slice2 SDG-edge-sum-SDG-edge)
```

with ‹$n_4 \in$ *sum-SDG-slice1* n_4›
‹*CFG-node* (*parent-node* n_3) $-p\rightarrow_{ret}$ *CFG-node* (*parent-node* n_4)› ‹$n'' = n_3$›
‹$n_4 \in S$›
show *?case* **by**(*fastforce intro*:*combSlice-Return-parent-node SDG-edge-sum-SDG-edge*
simp:*HRB-slice-def*)
qed
next
case (*matched-bracket-param* n_0 *ns* n_1 *p V* n_2 *ns'* n_3 *V'* n_4 *a a'*)
note *IH1* = ‹$\bigwedge S.\ [\![n'' \in set\ ns;\ n_1 \in S]\!] \Longrightarrow n'' \in$ *HRB-slice* S›
note *IH2* = ‹$\bigwedge S.\ [\![n'' \in set\ ns';\ n_3 \in S]\!] \Longrightarrow n'' \in$ *HRB-slice* S›
from ‹$n_1 -p{:}V\rightarrow_{in} n_2$› ‹*matched* n_2 *ns'* n_3› ‹$n_3 -p{:}V'\rightarrow_{out} n_4$›
‹$a' \in$ *get-return-edges* a› ‹*valid-edge* 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›
have *matched* n_1 ($[]@n_1\#ns'@[n_3]$) n_4
by(*fastforce intro*:*matched.matched-bracket-param matched-Nil*
elim:*SDG-edge-valid-SDG-node*)
then obtain *nsx* **where** n_1 *is*$-nsx\rightarrow_{d}*$ n_4 **by**(*erule matched-is-SDG-path*)
from ‹$n'' \in$ *set* ($ns@n_1\#ns'@[n_3]$)›
have (($n'' \in$ *set ns* $\vee$ $n'' = n_1$) $\vee$ $n'' \in$ *set ns'*) $\vee$ $n'' = n_3$ **by** *auto*
thus *?case* **apply** $-$
proof(*erule disjE*)+
assume $n'' \in$ *set ns*
from *IH1*[*OF this*] **have** $n'' \in$ *HRB-slice* $\{n_1\}$ **by** *simp*
with ‹n_1 *is*$-nsx\rightarrow_{d}*$ n_4› ‹$n_4 \in S$› **show** *?thesis*
by $-$(*rule HRB-slice-is-SDG-path-HRB-slice*)
next
assume $n'' = n_1$
from ‹n_1 *is*$-nsx\rightarrow_{d}*$ n_4› **have** $n_1 \in$ *sum-SDG-slice1* n_4
by(*fastforce intro*:*is-SDG-path-slice1 refl-slice1 is-SDG-path-valid-SDG-node*)
with ‹$n'' = n_1$› ‹$n_4 \in S$› **show** *?thesis*
by(*fastforce intro*:*combSlice-refl simp*:*HRB-slice-def*)
next
assume $n'' \in$ *set ns'*
with ‹*matched* n_2 *ns'* n_3› **have** $n'' \in$ *sum-SDG-slice2* n_3
by(*rule in-matched-in-slice2*)
with ‹$n_3 -p{:}V'\rightarrow_{out} n_4$› **have** $n'' \in$ *sum-SDG-slice2* n_4
by(*fastforce intro*:*slice2-param-out-slice2 SDG-edge-sum-SDG-edge*)
from ‹$n_3 -p{:}V'\rightarrow_{out} n_4$› **have** *valid-SDG-node* n_4 **by**(*rule SDG-edge-valid-SDG-node*)
hence $n_4 \in$ *sum-SDG-slice1* n_4 **by**(*rule refl-slice1*)
from ‹$n_3 -p{:}V'\rightarrow_{out} n_4$›
have *CFG-node* (*parent-node* n_3) $-p\rightarrow_{ret}$ *CFG-node* (*parent-node* n_4)
by(*fastforce elim*:*SDG-edge.cases intro*:*SDG-return-edge*)
with ‹$n'' \in$ *sum-SDG-slice2* n_4› ‹$n_4 \in$ *sum-SDG-slice1* n_4› ‹$n_4 \in S$›
show *?case* **by**(*fastforce intro*:*combSlice-Return-parent-node SDG-edge-sum-SDG-edge*

simp:*HRB-slice-def*)
next
assume $n'' = n_3$

```
  from ‹n3 −p:V'→out n4› have n3 s−p:V'→out n4 by(rule SDG-edge-sum-SDG-edge)
  from ‹n3 −p:V'→out n4› have valid-SDG-node n4 by(rule SDG-edge-valid-SDG-node)
   hence n4 ∈ sum-SDG-slice1 n4 by(rule refl-slice1)
   from ‹valid-SDG-node n4› have n4 ∈ sum-SDG-slice2 n4 by(rule refl-slice2)
  with ‹n3 s−p:V'→out n4› have n3 ∈ sum-SDG-slice2 n4 by(rule param-out-slice2)
   from ‹n3 −p:V'→out n4›
   have CFG-node (parent-node n3) −p→ret CFG-node (parent-node n4)
    by(fastforce elim:SDG-edge.cases intro:SDG-return-edge)
   with ‹n3 ∈ sum-SDG-slice2 n4› ‹n4 ∈ sum-SDG-slice1 n4› ‹n'' = n3› ‹n4 ∈ S›
  show ?case by(fastforce intro:combSlice-Return-parent-node SDG-edge-sum-SDG-edge
                          simp:HRB-slice-def)
 qed
qed


theorem in-realizable-in-HRB-slice:
 ⟦realizable n ns n'; n'' ∈ set ns; n' ∈ S⟧ ⟹ n'' ∈ HRB-slice S
proof(induct arbitrary:S rule:realizable.induct)
 case (realizable-matched n ns n') thus ?case by(rule in-matched-in-HRB-slice)
next
 case (realizable-call n0 ns n1 p n2 V ns' n3)
 note IH = ‹⋀S. ⟦n'' ∈ set ns; n1 ∈ S⟧ ⟹ n'' ∈ HRB-slice S›
 from ‹n'' ∈ set (ns@n1#ns')› have (n'' ∈ set ns ∨ n'' = n1) ∨ n'' ∈ set ns'
  by auto
 thus ?case apply −
 proof(erule disjE)+
  assume n'' ∈ set ns
  from IH[OF this] have n'' ∈ HRB-slice {n1} by simp
  hence n'' ∈ HRB-slice {n2}
  proof(induct rule:HRB-slice-cases)
   case (phase1 n nx)
   from ‹nx ∈ {n1}› have nx = n1 by simp
   with ‹n ∈ sum-SDG-slice1 nx› ‹n1 −p→call n2 ∨ n1 −p:V→in n2›
   have n ∈ sum-SDG-slice1 n2
    by(fastforce intro:slice1-call-slice1 slice1-param-in-slice1
                       SDG-edge-sum-SDG-edge)
   thus ?case
    by(fastforce intro:combine-SDG-slices.combSlice-refl simp:HRB-slice-def)
  next
   case (phase2 nx n' n'' p' n)
   from ‹nx ∈ {n1}› have nx = n1 by simp
   with ‹n' ∈ sum-SDG-slice1 nx› ‹n1 −p→call n2 ∨ n1 −p:V→in n2›
   have n' ∈ sum-SDG-slice1 n2
    by(fastforce intro:slice1-call-slice1 slice1-param-in-slice1
                       SDG-edge-sum-SDG-edge)
   with ‹n'' s−p'→ret CFG-node (parent-node n')› ‹n ∈ sum-SDG-slice2 n'›
show ?case
    by(fastforce intro:combine-SDG-slices.combSlice-Return-parent-node
                       simp:HRB-slice-def)
```

```
    qed
    from ⟨matched n2 ns' n3⟩ obtain nsx where n2 is−nsx→d* n3
      by(erule matched-is-SDG-path)
    with ⟨n'' ∈ HRB-slice {n2}⟩ ⟨n3 ∈ S⟩ show ?thesis
      by(fastforce intro:HRB-slice-is-SDG-path-HRB-slice)
  next
    assume n'' = n1
    from ⟨matched n2 ns' n3⟩ obtain nsx where n2 is−nsx→d* n3
      by(erule matched-is-SDG-path)
    hence n2 ∈ sum-SDG-slice1 n2
      by(fastforce intro:refl-slice1 is-SDG-path-valid-SDG-node)
    with ⟨n1 −p→call n2 ∨ n1 −p:V→in n2⟩
    have n1 ∈ sum-SDG-slice1 n2
      by(fastforce intro:call-slice1 param-in-slice1 SDG-edge-sum-SDG-edge)
   hence n1 ∈ HRB-slice {n2} by(fastforce intro:combSlice-refl simp:HRB-slice-def)
    with ⟨n2 is−nsx→d* n3⟩ ⟨n'' = n1⟩ ⟨n3 ∈ S⟩ show ?thesis
      by(fastforce intro:HRB-slice-is-SDG-path-HRB-slice)
  next
    assume n'' ∈ set ns'
    from ⟨matched n2 ns' n3⟩ this ⟨n3 ∈ S⟩ show ?thesis
      by(rule in-matched-in-HRB-slice)
  qed
qed


lemma slice1-ics-SDG-path:
  assumes n ∈ sum-SDG-slice1 n' and n ≠ n'
  obtains ns where CFG-node (-Entry-) ics−ns→d* n' and n ∈ set ns
proof(atomize-elim)
  from ⟨n ∈ sum-SDG-slice1 n'⟩
  have n = n' ∨ (∃ns. CFG-node (-Entry-) ics−ns→d* n' ∧ n ∈ set ns)
  proof(induct rule:sum-SDG-slice1.induct)
    case refl-slice1 thus ?case by simp
  next
    case (cdep-slice1 n'' n)
   from ⟨n'' s⟶cd n⟩ have valid-SDG-node n'' by(rule sum-SDG-edge-valid-SDG-node)
    hence n'' ics−[]→d* n'' by(rule icsSp-Nil)
    from ⟨valid-SDG-node n''⟩ have valid-node (parent-node n'')
      by(rule valid-SDG-CFG-node)
    thus ?case
    proof(cases parent-node n'' = (-Exit-))
      case True
      with ⟨valid-SDG-node n''⟩ have n'' = CFG-node (-Exit-)
        by(rule valid-SDG-node-parent-Exit)
     with ⟨n'' s⟶cd n⟩ have False by(fastforce intro:Exit-no-sum-SDG-edge-source)
      thus ?thesis by simp
    next
      case False
      from ⟨n'' s⟶cd n⟩ have valid-SDG-node n''
```

```
      by(rule sum-SDG-edge-valid-SDG-node)
    from this False obtain ns
      where CFG-node (-Entry-) cc−ns→d* n''
      by(erule Entry-cc-SDG-path-to-inner-node)
    with ⟨n'' s⟶cd n⟩ have CFG-node (-Entry-) cc−ns@[n'']→d* n
      by(fastforce intro:ccSp-Append-cdep sum-SDG-edge-SDG-edge)
    hence CFG-node (-Entry-) ics−ns@[n'']→d* n
      by(rule cc-SDG-path-ics-SDG-path)
    from ⟨n = n' ∨ (∃ ns. CFG-node (-Entry-) ics−ns→d* n' ∧ n ∈ set ns)⟩
    show ?thesis
    proof
      assume n = n'
      with ⟨CFG-node (-Entry-) ics−ns@[n'']→d* n⟩ show ?thesis by fastforce
    next
      assume ∃ ns. CFG-node (-Entry-) ics−ns→d* n' ∧ n ∈ set ns
      then obtain nsx where CFG-node (-Entry-) ics−nsx→d* n' and n ∈ set
nsx
        by blast
      then obtain ns' ns'' where nsx = ns'@ns'' and n ics−ns''→d* n'
        by −(erule ics-SDG-path-split)
      with ⟨CFG-node (-Entry-) ics−ns@[n'']→d* n⟩
      show ?thesis by(fastforce intro:ics-SDG-path-Append)
    qed
  qed
next
  case (ddep-slice1 n'' V n)
 from ⟨n'' s−V→dd n⟩ have valid-SDG-node n'' by(rule sum-SDG-edge-valid-SDG-node)
  hence n'' ics−[]→d* n'' by(rule icsSp-Nil)
  from ⟨valid-SDG-node n''⟩ have valid-node (parent-node n'')
    by(rule valid-SDG-CFG-node)
  thus ?case
  proof(cases parent-node n'' = (-Exit-))
    case True
    with ⟨valid-SDG-node n''⟩ have n'' = CFG-node (-Exit-)
      by(rule valid-SDG-node-parent-Exit)
  with ⟨n'' s−V→dd n⟩ have False by(fastforce intro:Exit-no-sum-SDG-edge-source)
    thus ?thesis by simp
  next
    case False
    from ⟨n'' s−V→dd n⟩ have valid-SDG-node n''
      by(rule sum-SDG-edge-valid-SDG-node)
    from this False obtain ns
      where CFG-node (-Entry-) cc−ns→d* n''
      by(erule Entry-cc-SDG-path-to-inner-node)
   hence CFG-node (-Entry-) ics−ns→d* n''
      by(rule cc-SDG-path-ics-SDG-path)
    show ?thesis
    proof(cases n'' = n)
      case True
```

from ‹$n = n' \vee (\exists\, ns.\ CFG\text{-}node\ (\text{-}Entry\text{-})\ ics{-}ns{\rightarrow_d}*\ n' \wedge n \in set\ ns)$›
show *?thesis*
proof
assume $n = n'$
with ‹$n'' = n$› **show** *?thesis* **by** *simp*
next
assume $\exists\, ns.\ CFG\text{-}node\ (\text{-}Entry\text{-})\ ics{-}ns{\rightarrow_d}*\ n' \wedge n \in set\ ns$
with ‹$n'' = n$› **show** *?thesis* **by** *fastforce*
qed
next
case *False*
with ‹$n''\ s{-}V{\rightarrow_{dd}}\ n$› ‹$CFG\text{-}node\ (\text{-}Entry\text{-})\ ics{-}ns{\rightarrow_d}*\ n''$›
have $CFG\text{-}node\ (\text{-}Entry\text{-})\ ics{-}ns@[n'']{\rightarrow_d}*\ n$
by −(*rule icsSp-Append-ddep*)
from ‹$n = n' \vee (\exists\, ns.\ CFG\text{-}node\ (\text{-}Entry\text{-})\ ics{-}ns{\rightarrow_d}*\ n' \wedge n \in set\ ns)$›
show *?thesis*
proof
assume $n = n'$
with ‹$CFG\text{-}node\ (\text{-}Entry\text{-})\ ics{-}ns@[n'']{\rightarrow_d}*\ n$› **show** *?thesis* **by** *fastforce*
next
assume $\exists\, ns.\ CFG\text{-}node\ (\text{-}Entry\text{-})\ ics{-}ns{\rightarrow_d}*\ n' \wedge n \in set\ ns$
then obtain *nsx* **where** $CFG\text{-}node\ (\text{-}Entry\text{-})\ ics{-}nsx{\rightarrow_d}*\ n'$ **and** $n \in set\ nsx$
by *blast*
then obtain $ns'\ ns''$ **where** $nsx = ns'@ns''$ **and** $n\ ics{-}ns''{\rightarrow_d}*\ n'$
by −(*erule ics-SDG-path-split*)
with ‹$CFG\text{-}node\ (\text{-}Entry\text{-})\ ics{-}ns@[n'']{\rightarrow_d}*\ n$›
show *?thesis* **by**(*fastforce intro*:*ics-SDG-path-Append*)
qed
qed
qed
next
case (*call-slice1* $n''\ p\ n$)
from ‹$n''\ s{-}p{\rightarrow_{call}}\ n$› **have** *valid-SDG-node* n''
by(*rule sum-SDG-edge-valid-SDG-node*)
hence $n''\ ics{-}[]{\rightarrow_d}*\ n''$ **by**(*rule icsSp-Nil*)
from ‹*valid-SDG-node* n''› **have** *valid-node* (*parent-node* n'')
by(*rule valid-SDG-CFG-node*)
thus *?case*
proof(*cases parent-node* $n'' = (\text{-}Exit\text{-})$)
case *True*
with ‹*valid-SDG-node* n''› **have** $n'' = CFG\text{-}node\ (\text{-}Exit\text{-})$
by(*rule valid-SDG-node-parent-Exit*)
with ‹$n''\ s{-}p{\rightarrow_{call}}\ n$› **have** *False* **by**(*fastforce intro*:*Exit-no-sum-SDG-edge-source*)
thus *?thesis* **by** *simp*
next
case *False*
from ‹$n''\ s{-}p{\rightarrow_{call}}\ n$› **have** *valid-SDG-node* n''
by(*rule sum-SDG-edge-valid-SDG-node*)

from *this False* **obtain** *ns*
where *CFG-node* (*-Entry-*) $cc{-}ns{\rightarrow_d}*\ n''$
by(*erule Entry-cc-SDG-path-to-inner-node*)
with ‹$n''\ s{-}p{\rightarrow_{call}}\ n$› **have** *CFG-node* (*-Entry-*) $cc{-}ns@[n']{\rightarrow_d}*\ n$
by(*fastforce intro*:*ccSp-Append-call sum-SDG-edge-SDG-edge*)
hence *CFG-node* (*-Entry-*) $ics{-}ns@[n']{\rightarrow_d}*\ n$
by(*rule cc-SDG-path-ics-SDG-path*)
from ‹$n = n' \vee (\exists\, ns.$ *CFG-node* (*-Entry-*) $ics{-}ns{\rightarrow_d}*\ n' \wedge n \in$ *set ns*)›
show *?thesis*
proof
assume $n = n'$
with ‹*CFG-node* (*-Entry-*) $ics{-}ns@[n']{\rightarrow_d}*\ n$› **show** *?thesis* **by** *fastforce*
next
assume $\exists\, ns.$ *CFG-node* (*-Entry-*) $ics{-}ns{\rightarrow_d}*\ n' \wedge n \in$ *set ns*
then obtain *nsx* **where** *CFG-node* (*-Entry-*) $ics{-}nsx{\rightarrow_d}*\ n'$ **and** $n \in$ *set nsx*
by *blast*
then obtain $ns'\ ns''$ **where** $nsx = ns'@ns''$ **and** $n\ ics{-}ns''{\rightarrow_d}*\ n'$
by −(*erule ics-SDG-path-split*)
with ‹*CFG-node* (*-Entry-*) $ics{-}ns@[n']{\rightarrow_d}*\ n$›
show *?thesis* **by**(*fastforce intro*:*ics-SDG-path-Append*)
qed
qed
next
case (*param-in-slice1* $n''\ p\ V\ n$)
from ‹$n''\ s{-}p{:}V{\rightarrow_{in}}\ n$› **have** *valid-SDG-node* n''
by(*rule sum-SDG-edge-valid-SDG-node*)
hence $n''\ ics{-}[]{\rightarrow_d}*\ n''$ **by**(*rule icsSp-Nil*)
from ‹*valid-SDG-node* n''› **have** *valid-node* (*parent-node* n'')
by(*rule valid-SDG-CFG-node*)
thus *?case*
proof(*cases parent-node* n'' = (*-Exit-*))
case *True*
with ‹*valid-SDG-node* n''› **have** n'' = *CFG-node* (*-Exit-*)
by(*rule valid-SDG-node-parent-Exit*)
with ‹$n''\ s{-}p{:}V{\rightarrow_{in}}\ n$› **have** *False* **by**(*fastforce intro*:*Exit-no-sum-SDG-edge-source*)
thus *?thesis* **by** *simp*
next
case *False*
from ‹$n''\ s{-}p{:}V{\rightarrow_{in}}\ n$› **have** *valid-SDG-node* n''
by(*rule sum-SDG-edge-valid-SDG-node*)
from *this False* **obtain** *ns*
where *CFG-node* (*-Entry-*) $cc{-}ns{\rightarrow_d}*\ n''$
by(*erule Entry-cc-SDG-path-to-inner-node*)
hence *CFG-node* (*-Entry-*) $ics{-}ns{\rightarrow_d}*\ n''$
by(*rule cc-SDG-path-ics-SDG-path*)
with ‹$n''\ s{-}p{:}V{\rightarrow_{in}}\ n$› **have** *CFG-node* (*-Entry-*) $ics{-}ns@[n']{\rightarrow_d}*\ n$
by −(*rule icsSp-Append-param-in*)
from ‹$n = n' \vee (\exists\, ns.$ *CFG-node* (*-Entry-*) $ics{-}ns{\rightarrow_d}*\ n' \wedge n \in$ *set ns*)›

show *?thesis*
proof
assume $n = n'$
with ‹*CFG-node* (*-Entry-*) *ics*$-ns@[n'']\rightarrow_{d*}$ *n*› **show** *?thesis* **by** *fastforce*
next
assume $\exists$ *ns*. *CFG-node* (*-Entry-*) *ics*$-ns\rightarrow_{d*}$ $n' \wedge n \in$ *set ns*
then obtain *nsx* **where** *CFG-node* (*-Entry-*) *ics*$-nsx\rightarrow_{d*}$ n' **and** $n \in$ *set nsx*
by *blast*
then obtain *ns' ns''* **where** *nsx* = *ns'@ns''* **and** *n ics*$-ns''\rightarrow_{d*}$ n'
by $-$(*erule ics-SDG-path-split*)
with ‹*CFG-node* (*-Entry-*) *ics*$-ns@[n'']\rightarrow_{d*}$ *n*›
show *?thesis* **by**(*fastforce intro*:*ics-SDG-path-Append*)
qed
qed
next
case (*sum-slice1 n'' p n*)
from ‹*n''* *s*$-p\rightarrow_{sum}$ *n*› **have** *valid-SDG-node n''*
by(*rule sum-SDG-edge-valid-SDG-node*)
hence *n''* *ics*$-[]\rightarrow_{d*}$ *n''* **by**(*rule icsSp-Nil*)
from ‹*valid-SDG-node n''*› **have** *valid-node* (*parent-node n''*)
by(*rule valid-SDG-CFG-node*)
thus *?case*
proof(*cases parent-node n'' = (-Exit-)*)
case *True*
with ‹*valid-SDG-node n''*› **have** *n''* = *CFG-node* (*-Exit-*)
by(*rule valid-SDG-node-parent-Exit*)
with ‹*n''* *s*$-p\rightarrow_{sum}$ *n*› **have** *False* **by**(*fastforce intro*:*Exit-no-sum-SDG-edge-source*)
thus *?thesis* **by** *simp*
next
case *False*
from ‹*n''* *s*$-p\rightarrow_{sum}$ *n*› **have** *valid-SDG-node n''*
by(*rule sum-SDG-edge-valid-SDG-node*)
from *this False* **obtain** *ns*
where *CFG-node* (*-Entry-*) *cc*$-ns\rightarrow_{d*}$ *n''*
by(*erule Entry-cc-SDG-path-to-inner-node*)
hence *CFG-node* (*-Entry-*) *ics*$-ns\rightarrow_{d*}$ *n''*
by(*rule cc-SDG-path-ics-SDG-path*)
with ‹*n''* *s*$-p\rightarrow_{sum}$ *n*› **have** *CFG-node* (*-Entry-*) *ics*$-ns@[n'']\rightarrow_{d*}$ *n*
by $-$(*rule icsSp-Append-sum*)
from ‹$n = n' \vee (\exists$ *ns*. *CFG-node* (*-Entry-*) *ics*$-ns\rightarrow_{d*}$ $n' \wedge n \in$ *set ns*)›
show *?thesis*
proof
assume $n = n'$
with ‹*CFG-node* (*-Entry-*) *ics*$-ns@[n'']\rightarrow_{d*}$ *n*› **show** *?thesis* **by** *fastforce*
next
assume $\exists$ *ns*. *CFG-node* (*-Entry-*) *ics*$-ns\rightarrow_{d*}$ $n' \wedge n \in$ *set ns*
then obtain *nsx* **where** *CFG-node* (*-Entry-*) *ics*$-nsx\rightarrow_{d*}$ n' **and** $n \in$ *set nsx*

```
      by blast
    then obtain ns' ns'' where nsx = ns'@ns'' and n ics−ns''→d* n'
      by −(erule ics-SDG-path-split)
    with ‹CFG-node (-Entry-) ics−ns@[n'']→d* n›
    show ?thesis by(fastforce intro:ics-SDG-path-Append)
  qed
 qed
 qed
 with ‹n ≠ n'› show ∃ns. CFG-node (-Entry-) ics−ns→d* n' ∧ n ∈ set ns by
simp
qed


lemma slice2-irs-SDG-path:
 assumes n ∈ sum-SDG-slice2 n' and valid-SDG-node n'
 obtains ns where n irs−ns→d* n'
using assms
by(induct rule:sum-SDG-slice2.induct,auto intro:intra-return-sum-SDG-path.intros)


theorem HRB-slice-realizable:
 assumes n ∈ HRB-slice S and ∀n' ∈ S. valid-SDG-node n' and n ∉ S
 obtains n' ns where n' ∈ S and realizable (CFG-node (-Entry-)) ns n'
 and n ∈ set ns
proof(atomize-elim)
 from ‹n ∈ HRB-slice S› ‹n ∉ S›
 show ∃n' ns. n' ∈ S ∧ realizable (CFG-node (-Entry-)) ns n' ∧ n ∈ set ns
 proof(induct rule:HRB-slice-cases)
  case (phase1 n nx)
  with ‹n ∉ S› show ?case
   by(fastforce elim:slice1-ics-SDG-path ics-SDG-path-realizable)
 next
  case (phase2 n' nx n'' p n)
  from ‹∀n' ∈ S. valid-SDG-node n'› ‹n' ∈ S› have valid-SDG-node n' by simp
  with ‹nx ∈ sum-SDG-slice1 n'› have valid-SDG-node nx
   by(auto elim:slice1-ics-SDG-path ics-SDG-path-split
         intro:ics-SDG-path-valid-SDG-node)
  with ‹n ∈ sum-SDG-slice2 nx›
  obtain nsx where n irs−nsx→d* nx by(erule slice2-irs-SDG-path)
  show ?case
  proof(cases n = nx)
   case True
   show ?thesis
   proof(cases nx = n')
    case True
    with ‹n = nx› ‹n ∉ S› ‹n' ∈ S› have False by simp
    thus ?thesis by simp
   next
    case False
```

with ‹$nx \in$ *sum-SDG-slice1* n'› **obtain** *ns*
where *realizable* (*CFG-node* (*-Entry-*)) *ns* n' **and** $nx \in$ *set ns*
by(*fastforce elim*:*slice1-ics-SDG-path ics-SDG-path-realizable*)
with ‹$n = nx$› ‹$n' \in S$› **show** *?thesis* **by** *blast*
qed
next
case *False*
with ‹n *irs*$-nsx\to_{d}*$ nx› **obtain** *ns*
where *realizable* (*CFG-node* (*-Entry-*)) *ns* nx **and** $n \in$ *set ns*
by(*erule irs-SDG-path-realizable*)
show *?thesis*
proof(*cases* $nx = n'$)
case *True*
with ‹*realizable* (*CFG-node* (*-Entry-*)) *ns* nx› ‹$n \in$ *set ns*› ‹$n' \in S$›
show *?thesis* **by** *blast*
next
case *False*
with ‹$nx \in$ *sum-SDG-slice1* n'› **obtain** nsx'
where *CFG-node* (*-Entry-*) *ics*$-nsx'\to_{d}*$ n' **and** $nx \in$ *set* nsx'
by(*erule slice1-ics-SDG-path*)
then obtain ns' **where** nx *ics*$-ns'\to_{d}*$ n' **by** $-$(*erule ics-SDG-path-split*)
with ‹*realizable* (*CFG-node* (*-Entry-*)) *ns* nx›
obtain ns'' **where** *realizable* (*CFG-node* (*-Entry-*)) (ns@ns'') n'
by(*erule realizable-Append-ics-SDG-path*)
with ‹$n \in$ *set ns*› ‹$n' \in S$› **show** *?thesis* **by** *fastforce*
qed
qed
qed
qed

theorem *HRB-slice-precise*:
$[\![\forall n' \in S.$ *valid-SDG-node* n'; $n \notin S]\!] \Longrightarrow$
$n \in$ *HRB-slice* $S =$
$(\exists n'$ *ns*. $n' \in S \wedge$ *realizable* (*CFG-node* (*-Entry-*)) *ns* $n' \wedge n \in$ *set ns*)
by(*fastforce elim*:*HRB-slice-realizable intro*:*in-realizable-in-HRB-slice*)

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(*atomize-elim*)
from ‹$n_1\ -p\rightarrow_{call}\ n_2 \vee n_1\ -p{:}V'\rightarrow_{in}\ n_2$› **obtain** $a\ Q\ r\ fs$ **where** *valid-edge* a
and *kind* $a = Q{:}r\hookrightarrow_p fs$ **and** *parent-node* n_1 = *sourcenode* a
and *parent-node* n_2 = *targetnode* a
by(*fastforce elim*:*SDG-edge.cases*)
from ‹$n_3\ -p\rightarrow_{ret}\ n_4 \vee n_3\ -p{:}V\rightarrow_{out}\ n_4$› **obtain** $a'\ Q'\ f'$
where *valid-edge* a' **and** *kind* $a' = Q'\hookleftarrow_p f'$
and *parent-node* n_3 = *sourcenode* a' **and** *parent-node* n_4 = *targetnode* a'
by(*fastforce elim*:*SDG-edge.cases*)
from ‹*valid-edge* a'› ‹*kind* $a' = Q'\hookleftarrow_p f'$›
obtain ax **where** *valid-edge* ax **and** $\exists\, Q\ r\ fs.$ *kind* $ax = Q{:}r\hookrightarrow_p fs$
and $a' \in$ *get-return-edges* ax
by −(*drule return-needs-call*,*fastforce*+)
from ‹*valid-edge* a› ‹*valid-edge* ax› ‹*kind* $a = Q{:}r\hookrightarrow_p fs$› ‹$\exists\, Q\ r\ fs.$ *kind* $ax = Q{:}r\hookrightarrow_p fs$›
have *targetnode* a = *targetnode* ax **by**(*fastforce dest*:*same-proc-call-unique-target*)
from ‹*valid-edge* a'› ‹$a' \in$ *get-return-edges* ax› ‹*valid-edge* ax›
have *call-of-return-node* (*targetnode* a') (*sourcenode* ax)
by(*fastforce simp*:*return-node-def call-of-return-node-def*)
with ‹*call-of-return-node* (*parent-node* n_4) (*parent-node* n_1)›
‹*parent-node* n_4 = *targetnode* a'›
have *sourcenode* ax = *parent-node* n_1 **by** *fastforce*
with ‹*valid-edge* ax› ‹$a' \in$ *get-return-edges* ax› ‹*targetnode* a = *targetnode* ax›
‹*parent-node* n_2 = *targetnode* a› ‹*parent-node* n_3 = *sourcenode* a'›
‹*parent-node* n_4 = *targetnode* a'›
show $\exists\, a\ a'.$ *valid-edge* $a \wedge a' \in$ *get-return-edges* $a \wedge$
sourcenode a = *parent-node* $n_1 \wedge$ *targetnode* a = *parent-node* $n_2 \wedge$
sourcenode a' = *parent-node* $n_3 \wedge$ *targetnode* a' = *parent-node* n_4
by *fastforce*
qed

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
by(*fastforce simp*:*SDG-to-CFG-set-def*)

lemma *Exit-HRB-Slice*:
assumes $n \in \lfloor$*HRB-slice* {*CFG-node* (*-Exit-*)}$\rfloor_{CFG}$ **shows** n = (*-Exit-*)
proof −
from ‹$n \in \lfloor$*HRB-slice* {*CFG-node* (*-Exit-*)}$\rfloor_{CFG}$›
have *CFG-node n* ∈ *HRB-slice* {*CFG-node* (*-Exit-*)}
by(*simp add*:*SDG-to-CFG-set-def*)
thus *?thesis*
proof(*induct CFG-node n rule*:*HRB-slice-cases*)
case (*phase1 nx*)
from ‹*nx* ∈ {*CFG-node* (*-Exit-*)}› **have** *nx* = *CFG-node* (*-Exit-*) **by** *simp*
with ‹*CFG-node n* ∈ *sum-SDG-slice1 nx*›
have *CFG-node n* = *CFG-node* (*-Exit-*) ∨
(∃ *n Vopt popt b*. *sum-SDG-edge n Vopt popt b* (*CFG-node* (*-Exit-*)))
by(*induct rule*:*sum-SDG-slice1.induct*) *auto*
then show *?thesis* **by**(*fastforce dest*:*Exit-no-sum-SDG-edge-target*)
next
case (*phase2 nx n' n'' p*)
from ‹*nx* ∈ {*CFG-node* (*-Exit-*)}› **have** *nx* = *CFG-node* (*-Exit-*) **by** *simp*
with ‹*n'* ∈ *sum-SDG-slice1 nx*›
have *n'* = *CFG-node* (*-Exit-*) ∨
(∃ *n Vopt popt b*. *sum-SDG-edge n Vopt popt b* (*CFG-node* (*-Exit-*)))
by(*induct rule*:*sum-SDG-slice1.induct*) *auto*
hence *n'* = *CFG-node* (*-Exit-*) **by**(*fastforce dest*:*Exit-no-sum-SDG-edge-target*)
with ‹*CFG-node n* ∈ *sum-SDG-slice2 n'*›
have *CFG-node n* = *CFG-node* (*-Exit-*) ∨
(∃ *n Vopt popt b*. *sum-SDG-edge n Vopt popt b* (*CFG-node* (*-Exit-*)))
by(*induct rule*:*sum-SDG-slice2.induct*) *auto*
then show *?thesis* **by**(*fastforce dest*:*Exit-no-sum-SDG-edge-target*)
qed
qed

lemma *Exit-in-obs-intra-slice-node*:
assumes (*-Exit-*) ∈ *obs-intra n'* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
shows *CFG-node* (*-Exit-*) ∈ *S*
proof −
let *?S'* = $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
from ‹(*-Exit-*) ∈ *obs-intra n' ?S'*› **obtain** *as* **where** *n'* −*as*→$_\iota$* (*-Exit-*)
and ∀ *nx* ∈ *set*(*sourcenodes as*). *nx* ∉ *?S'* **and** (*-Exit-*) ∈ *?S'*
by(*erule obs-intraE*)
from ‹(*-Exit-*) ∈ *?S'*›
have *CFG-node* (*-Exit-*) ∈ *HRB-slice S* **by**(*simp add*:*SDG-to-CFG-set-def*)
thus *?thesis*
proof(*induct CFG-node* (*-Exit-*) *rule*:*HRB-slice-cases*)
case (*phase1 nx*)
thus *?case*
by(*induct CFG-node* (*-Exit-*) *rule*:*sum-SDG-slice1.induct*,
auto dest:*Exit-no-sum-SDG-edge-source*)
next

```
    case (phase2 nx n' n'' p)
    from ‹CFG-node (-Exit-) ∈ sum-SDG-slice2 n'› ‹n' ∈ sum-SDG-slice1 nx› ‹nx
∈ S›
    show ?case
      apply(induct n≡CFG-node (-Exit-) rule:sum-SDG-slice2.induct)
      apply(auto dest:Exit-no-sum-SDG-edge-source)
      apply(induct n≡CFG-node (-Exit-) rule:sum-SDG-slice1.induct)
      apply(auto dest:Exit-no-sum-SDG-edge-source)
      done
  qed
qed


lemma obs-intra-postdominate:
  assumes n ∈ obs-intra n' ⌊HRB-slice S⌋CFG and ¬ method-exit n
  shows n postdominates n'
proof(rule ccontr)
  assume ¬ n postdominates n'
  from ‹n ∈ obs-intra n' ⌊HRB-slice S⌋CFG› have valid-node n
    by(fastforce dest:in-obs-intra-valid)
  with ‹n ∈ obs-intra n' ⌊HRB-slice S⌋CFG› ‹¬ method-exit n› have n postdomi-
nates n
    by(fastforce intro:postdominate-refl)
  from ‹n ∈ obs-intra n' ⌊HRB-slice S⌋CFG› obtain as where n' −as→ι* n
    and all-notinS:∀ n' ∈ set(sourcenodes as). n' ∉ ⌊HRB-slice S⌋CFG
    and n ∈ ⌊HRB-slice S⌋CFG by(erule obs-intraE)
  from ‹n postdominates n› ‹¬ n postdominates n'› ‹n' −as→ι* n›
  obtain as' a as'' where [simp]:as = as'@a#as''
    and valid-edge a and ¬ n postdominates (sourcenode a)
    and n postdominates (targetnode a) and intra-kind (kind a)
    by(fastforce elim!:postdominate-path-branch simp:intra-path-def)
  from ‹n' −as→ι* n› have sourcenode a −a#as''→ι* n
    by(fastforce elim:path-split intro:Cons-path simp:intra-path-def)
  with ‹¬ n postdominates (sourcenode a)› ‹valid-edge a› ‹valid-node n›
  obtain asx pex where sourcenode a −asx→ι* pex and method-exit pex
    and n ∉ set(sourcenodes asx) by(fastforce simp:postdominate-def)
  have asx ≠ []
  proof
    assume asx = []
    with ‹sourcenode a −asx→ι* pex› have sourcenode a = pex
      by(fastforce simp:intra-path-def)
    from ‹method-exit pex› show False
    proof(rule method-exit-cases)
      assume pex = (-Exit-)
      with ‹sourcenode a = pex› have sourcenode a = (-Exit-) by simp
      with ‹valid-edge a› show False by(rule Exit-source)
    next
      fix a' Q f p
      assume pex = sourcenode a' and valid-edge a' and kind a' = Q↩pf
```

from ‹*valid-edge a'*› ‹*kind a' = Q↩pf*› ‹*valid-edge a*› ‹*intra-kind (kind a)*›
‹*sourcenode a = pex*› ‹*pex = sourcenode a'*›
show *False* **by**(*fastforce dest*:*return-edges-only simp*:*intra-kind-def*)
qed
qed
then obtain *ax asx'* **where** [*simp*]:*asx = ax#asx'* **by**(*cases asx*) *auto*
with ‹*sourcenode a −asx→ι* pex*› **have** *sourcenode a −ax#asx'→* pex*
by(*simp add*:*intra-path-def*)
hence *valid-edge ax* **and** [*simp*]:*sourcenode a = sourcenode ax*
and *targetnode ax −asx'→* pex* **by**(*auto elim*:*path-split-Cons*)
with ‹*sourcenode a −asx→ι* pex*› **have** *targetnode ax −asx'→ι* pex*
by(*simp add*:*intra-path-def*)
with ‹*valid-edge ax*› ‹*n ∉ set*(*sourcenodes asx*)› ‹*method-exit pex*›
have ¬ *n postdominates targetnode ax*
by(*fastforce simp*:*postdominate-def sourcenodes-def*)
from ‹*n ∈ obs-intra n' ⌊HRB-slice S⌋CFG*› *all-notinS*
have *n ∉ set* (*sourcenodes* (*a#as''*))
by(*fastforce elim*:*obs-intra.cases simp*:*sourcenodes-def*)
from ‹*sourcenode a −asx→ι* pex*› **have** *intra-kind* (*kind ax*)
by(*simp add*:*intra-path-def*)
with ‹*sourcenode a −a#as''→ι* n*› ‹*n postdominates* (*targetnode a*)›
‹¬ *n postdominates targetnode ax*› ‹*valid-edge ax*›
‹*n ∉ set* (*sourcenodes* (*a#as''*))› ‹*intra-kind* (*kind a*)›
have (*sourcenode a*) *controls n*
by(*fastforce simp*:*control-dependence-def*)
hence *CFG-node* (*sourcenode a*) *s⟶cd CFG-node n*
by(*fastforce intro*:*sum-SDG-cdep-edge*)
with ‹*n ∈ obs-intra n' ⌊HRB-slice S⌋CFG*› **have** *sourcenode a ∈ ⌊HRB-slice S⌋CFG*
by(*auto elim*!:*obs-intraE combine-SDG-slices.cases*
intro:*combine-SDG-slices.intros sum-SDG-slice1.intros*
sum-SDG-slice2.intros simp:*HRB-slice-def SDG-to-CFG-set-def*)
with *all-notinS* **show** *False* **by**(*simp add*:*sourcenodes-def*)
qed

lemma *obs-intra-singleton-disj*:
assumes *valid-node n*
shows (∃ *m*. *obs-intra n ⌊HRB-slice S⌋CFG = {m}*) ∨
obs-intra n ⌊HRB-slice S⌋CFG = {}
proof(*rule ccontr*)
assume ¬ ((∃ *m*. *obs-intra n ⌊HRB-slice S⌋CFG = {m}*) ∨
obs-intra n ⌊HRB-slice S⌋CFG = {})
hence ∃ *nx nx'*. *nx ∈ obs-intra n ⌊HRB-slice S⌋CFG ∧*
nx' ∈ obs-intra n ⌊HRB-slice S⌋CFG ∧ nx ≠ nx' **by** *auto*
then obtain *nx nx'* **where** *nx ∈ obs-intra n ⌊HRB-slice S⌋CFG*
and *nx' ∈ obs-intra n ⌊HRB-slice S⌋CFG* **and** *nx ≠ nx'* **by** *auto*
from ‹*nx ∈ obs-intra n ⌊HRB-slice S⌋CFG*› **obtain** *as* **where** *n −as→ι* nx*

and *all*:$\forall\, n' \in set(sourcenodes\ as).\ n' \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
and $nx \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*erule obs-intraE*)
from ‹$n\ -as\rightarrow_\iota *\ nx$› **have** $n\ -as\rightarrow *\ nx$ **and** $\forall\, a \in set\ as.\ intra\text{-}kind\ (kind\ a)$
by(*simp-all add*:*intra-path-def*)
hence *valid-node nx* **by**(*fastforce dest*:*path-valid-node*)
with ‹$nx \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$› **have** *obs-intra nx* $\lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{nx\}$
by −(*rule n-in-obs-intra*)
with ‹$n\ -as\rightarrow *\ nx$› ‹$nx \in obs\text{-}intra\ n \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
‹$nx' \in obs\text{-}intra\ n \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$› ‹$nx \neq nx'$› **have** $as \neq []$
by(*fastforce elim*:*path.cases simp*:*intra-path-def*)
with ‹$n\ -as\rightarrow *\ nx$› ‹$nx \in obs\text{-}intra\ n \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
‹$nx' \in obs\text{-}intra\ n \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$› ‹$nx \neq nx'$›
‹*obs-intra nx* $\lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{nx\}$› ‹$\forall\, a \in set\ as.\ intra\text{-}kind\ (kind\ a)$› *all*
have $\exists\, a\ as'\ as''.\ n\ -as'\rightarrow_\iota *\ sourcenode\ a \wedge targetnode\ a\ -as''\rightarrow_\iota *\ nx \wedge$
$valid\text{-}edge\ a \wedge as = as'@a\#as'' \wedge intra\text{-}kind\ (kind\ a) \wedge$
$obs\text{-}intra\ (targetnode\ a) \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{nx\} \wedge$
$(\neg\ (\exists\, m.\ obs\text{-}intra\ (sourcenode\ a) \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{m\} \vee$
$obs\text{-}intra\ (sourcenode\ a) \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{\}))$
proof(*induct arbitrary*:*nx'* *rule*:*path.induct*)
case (*Cons-path n'' as n' a n*)
note *IH* = ‹$\bigwedge nx'.\ [\![n' \in obs\text{-}intra\ n'' \lfloor HRB\text{-}slice\ S \rfloor_{CFG};$
$nx' \in obs\text{-}intra\ n'' \lfloor HRB\text{-}slice\ S \rfloor_{CFG};\ n' \neq nx';$
$obs\text{-}intra\ n' \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{n'\};$
$\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a);$
$\forall\, n'{\in}set\ (sourcenodes\ as).\ n' \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG};\ as \neq []]\!]$
$\Longrightarrow \exists\, a\ as'\ as''.\ n''\ -as'\rightarrow_\iota *\ sourcenode\ a \wedge targetnode\ a\ -as''\rightarrow_\iota *\ n' \wedge$
$valid\text{-}edge\ a \wedge as = as'@a\#as'' \wedge intra\text{-}kind\ (kind\ a) \wedge$
$obs\text{-}intra\ (targetnode\ a) \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{n'\} \wedge$
$(\neg\ (\exists\, m.\ obs\text{-}intra\ (sourcenode\ a) \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{m\} \vee$
$obs\text{-}intra\ (sourcenode\ a) \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{\}))$›
note *more-than-one* = ‹$n' \in obs\text{-}intra\ n \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
‹$nx' \in obs\text{-}intra\ n \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$› ‹$n' \neq nx'$›
from ‹$\forall\, a{\in}set\ (a\#as).\ intra\text{-}kind\ (kind\ a)$›
have $\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a)$ **and** $intra\text{-}kind\ (kind\ a)$ **by** *simp-all*
from ‹$\forall\, n'{\in}set\ (sourcenodes\ (a\#as)).\ n' \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
have *all*:$\forall\, n'{\in}set\ (sourcenodes\ as).\ n' \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*simp add*:*sourcenodes-def*)
show *?case*
proof(*cases as* = [])
case *True*
with ‹$n''\ -as\rightarrow *\ n'$› **have** [*simp*]:$n'' = n'$ **by**(*fastforce elim*:*path.cases*)
from *more-than-one* ‹*sourcenode a* = *n*›
have $\neg\ (\exists\, m.\ obs\text{-}intra\ (sourcenode\ a) \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{m\} \vee$
$obs\text{-}intra\ (sourcenode\ a) \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{\})$
by *auto*
with ‹*targetnode a* = *n''*› ‹*obs-intra n'* $\lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{n'\}$›
‹*sourcenode a* = *n*› *True* ‹*valid-edge a*› ‹$intra\text{-}kind\ (kind\ a)$›
show *?thesis*

apply(*rule-tac x=a* **in** *exI*)
apply(*rule-tac x=[]* **in** *exI*)
apply(*rule-tac x=[]* **in** *exI*)
by(*auto intro:empty-path simp:intra-path-def*)
next
case *False*
from ‹$n'' -as\to* n'$› ‹$\forall a{\in}set\ (a\ \#\ as).\ intra\text{-}kind\ (kind\ a)$›
have $n'' -as\to_\iota* n'$ **by**(*simp add:intra-path-def*)
with *all*
have *subset:obs-intra n' ⌊HRB-slice S⌋CFG ⊆ obs-intra n'' ⌊HRB-slice S⌋CFG*
by −(*rule path-obs-intra-subset*)
thus *?thesis*
proof(*cases obs-intra n' ⌊HRB-slice S⌋CFG = obs-intra n'' ⌊HRB-slice S⌋CFG*)
case *True*
with ‹$n'' -as\to_\iota* n'$› ‹*valid-edge a*› ‹*sourcenode a = n*› ‹*targetnode a = n''*›
‹*obs-intra n' ⌊HRB-slice S⌋CFG = {n'}*› ‹*intra-kind (kind a)*› *more-than-one*
show *?thesis*
apply(*rule-tac x=a* **in** *exI*)
apply(*rule-tac x=[]* **in** *exI*)
apply(*rule-tac x=as* **in** *exI*)
by(*fastforce intro:empty-path simp:intra-path-def*)
next
case *False*
with *subset*
have *obs-intra n' ⌊HRB-slice S⌋CFG ⊂ obs-intra n'' ⌊HRB-slice S⌋CFG* **by** *simp*
with ‹*obs-intra n' ⌊HRB-slice S⌋CFG = {n'}*›
obtain *ni* **where** *n' ∈ obs-intra n'' ⌊HRB-slice S⌋CFG*
and *ni ∈ obs-intra n'' ⌊HRB-slice S⌋CFG* **and** *n' ≠ ni* **by** *auto*
from *IH*[*OF this* ‹*obs-intra n' ⌊HRB-slice S⌋CFG = {n'}*›
‹$\forall a{\in}set\ as.\ intra\text{-}kind\ (kind\ a)$› *all* ‹*as ≠ []*›] **obtain** *a' as' as''*
where *n'' −as'→ι* sourcenode a'*
and *hyps:targetnode a' −as''→ι* n' valid-edge a' as = as'@a'#as''*
intra-kind (kind a') obs-intra (targetnode a') ⌊HRB-slice S⌋CFG = {n'}
¬ (∃ m. obs-intra (sourcenode a') ⌊HRB-slice S⌋CFG = {m} ∨
obs-intra (sourcenode a') ⌊HRB-slice S⌋CFG = {})
by *blast*
from ‹*n'' −as'→ι* sourcenode a'› ‹*valid-edge a*› ‹*sourcenode a = n*›
‹*targetnode a = n''*› ‹*intra-kind (kind a)*› ‹*intra-kind (kind a')*›
have *n −a#as'→ι* sourcenode a'*
by(*fastforce intro:path.Cons-path simp:intra-path-def*)
with *hyps* **show** *?thesis*
apply(*rule-tac x=a'* **in** *exI*)
apply(*rule-tac x=a#as'* **in** *exI*)
apply(*rule-tac x=as''* **in** *exI*)
by *fastforce*
qed
qed

qed *simp*
then obtain *a as′ as″* **where** *valid-edge a* **and** *intra-kind* (*kind a*)
and *obs-intra* (*targetnode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {*nx*}
and *more-than-one*:¬ (∃ *m*. *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {*m*}
∨
obs-intra (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {})
by *blast*
have *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
proof(*rule ccontr*)
assume ¬ *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
hence *sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$ **by** *simp*
with ‹*valid-edge a*›
have *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {*sourcenode a*}
by(*fastforce intro*!:*n-in-obs-intra*)
with *more-than-one* **show** *False* **by** *simp*
qed
with ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *obs-intra* (*targetnode a*) ⌊*HRB-slice S*⌋$_{CFG}$ ⊆
obs-intra (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$
by(*rule edge-obs-intra-subset*)
with ‹*obs-intra* (*targetnode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {*nx*}›
have *nx* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ **by** *simp*
with *more-than-one* **obtain** *m*
where *m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ **and** *nx* ≠ *m* **by** *auto*
from ‹*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$› **have** *valid-node m*
by(*fastforce dest*:*in-obs-intra-valid*)
from ‹*obs-intra* (*targetnode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {*nx*}› **have** *valid-node nx*
by(*fastforce dest*:*in-obs-intra-valid*)
show *False*
proof(*cases m postdominates* (*sourcenode a*))
case *True*
with ‹*nx* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
‹*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
have *m postdominates nx*
by(*fastforce intro*:*postdominate-inner-path-targetnode elim*:*obs-intraE*)
with ‹*nx* ≠ *m*› **have** ¬ *nx postdominates m* **by**(*fastforce dest*:*postdominate-antisym*)
with ‹*valid-node nx*› ‹*valid-node m*› **obtain** *asx pex* **where** *m* −*asx*→$_\iota$* *pex*
and *method-exit pex* **and** *nx* ∉ *set*(*sourcenodes asx*)
by(*fastforce simp*:*postdominate-def*)
have ¬ *nx postdominates* (*sourcenode a*)
proof
assume *nx postdominates sourcenode a*
from ‹*nx* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
‹*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
obtain *asx′* **where** *sourcenode a* −*asx′*→$_\iota$* *m* **and** *nx* ∉ *set*(*sourcenodes asx′*)
by(*fastforce elim*:*obs-intraE*)
with ‹*m* −*asx*→$_\iota$* *pex*› **have** *sourcenode a* −*asx′*@*asx*→$_\iota$* *pex*
by(*fastforce intro*:*path-Append simp*:*intra-path-def*)

```
    with ‹nx ∉ set(sourcenodes asx)› ‹nx ∉ set(sourcenodes asx')›
      ‹nx postdominates sourcenode a› ‹method-exit pex› show False
      by(fastforce simp:sourcenodes-def postdominate-def)
  qed
  show False
  proof(cases method-exit nx)
    case True
    from ‹m postdominates nx› obtain xs where nx −xs→ι* m
      by −(erule postdominate-implies-inner-path)
    with True have nx = m
      by(fastforce dest!:method-exit-inner-path simp:intra-path-def)
    with ‹nx ≠ m› show False by simp
  next
    case False
    with ‹nx ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG›
    have nx postdominates sourcenode a by(rule obs-intra-postdominate)
    with ‹¬ nx postdominates (sourcenode a)› show False by simp
  qed
next
  case False
  show False
  proof(cases method-exit m)
    case True
    from ‹m ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG›
      ‹nx ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG›
    obtain xs where sourcenode a −xs→ι* m and nx ∉ set(sourcenodes xs)
      by(fastforce elim:obs-intraE)
    obtain x' xs' where [simp]:xs = x'#xs'
    proof(cases xs)
      case Nil
      with ‹sourcenode a −xs→ι* m› have [simp]:sourcenode a = m
        by(fastforce simp:intra-path-def)
      with ‹m ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG›
      have m ∈ ⌊HRB-slice S⌋CFG by(fastforce elim:obs-intraE)
      with ‹valid-node m› have obs-intra m ⌊HRB-slice S⌋CFG = {m}
        by(rule n-in-obs-intra)
       with ‹nx ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG› ‹nx ≠ m› have
False
        by fastforce
      thus ?thesis by simp
    qed blast
    from ‹sourcenode a −xs→ι* m› have sourcenode a = sourcenode x'
      and valid-edge x' and targetnode x' −xs'→ι* m
      and intra-kind (kind x')
      by(auto elim:path-split-Cons simp:intra-path-def)
    from ‹targetnode x' −xs'→ι* m› ‹nx ∉ set(sourcenodes xs)› ‹valid-edge x'›
      ‹valid-node m› True
    have ¬ nx postdominates (targetnode x')
      by(fastforce simp:postdominate-def sourcenodes-def)
```

show *False*
proof(*cases method-exit nx*)
case *True*
from ‹$m \in$ *obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
‹$nx \in$ *obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
have *get-proc m* = *get-proc nx*
by(*fastforce elim*:*obs-intraE dest*:*intra-path-get-procs*)
with ‹*method-exit m*› ‹*method-exit nx*› **have** *m* = *nx*
by(*rule method-exit-unique*)
with ‹*nx* ≠ *m*› **show** *False* **by** *simp*
next
case *False*
with ‹*obs-intra* (*targetnode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {*nx*}›
have *nx postdominates* (*targetnode a*)
by(*fastforce intro*:*obs-intra-postdominate*)
from ‹*obs-intra* (*targetnode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {*nx*}›
obtain *ys* **where** *targetnode a* $-ys\rightarrow_\iota *$ *nx*
and $\forall$ *nx'* ∈ *set*(*sourcenodes ys*). *nx'* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
and *nx* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by**(*fastforce elim*:*obs-intraE*)
hence *nx* ∉ *set*(*sourcenodes ys*)**by** *fastforce*
have *sourcenode a* ≠ *nx*
proof
assume *sourcenode a* = *nx*
from ‹*nx* ∈ *obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
have *nx* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by** −(*erule obs-intraE*)
with ‹*valid-node nx*›
have *obs-intra nx* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {*nx*} **by** −(*erule n-in-obs-intra*)
with ‹*sourcenode a* = *nx*› ‹*m* ∈ *obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
‹*nx* ≠ *m*› **show** *False* **by** *fastforce*
qed
with ‹*nx* ∉ *set*(*sourcenodes ys*)› **have** *nx* ∉ *set*(*sourcenodes* (*a*#*ys*))
by(*fastforce simp*:*sourcenodes-def*)
from ‹*valid-edge a*› ‹*targetnode a* $-ys\rightarrow_\iota *$ *nx*› ‹*intra-kind* (*kind a*)›
have *sourcenode a* $-a\#ys\rightarrow_\iota *$ *nx*
by(*fastforce intro*:*Cons-path simp*:*intra-path-def*)
from ‹*sourcenode a* $-a\#ys\rightarrow_\iota *$ *nx*› ‹*nx* ∉ *set*(*sourcenodes* (*a*#*ys*))›
‹*intra-kind* (*kind a*)› ‹*nx postdominates* (*targetnode a*)›
‹*valid-edge x'*› ‹*intra-kind* (*kind x'*)› ‹¬ *nx postdominates* (*targetnode x'*)›
‹*sourcenode a* = *sourcenode x'*›
have (*sourcenode a*) *controls nx*
by(*fastforce simp*:*control-dependence-def*)
hence *CFG-node* (*sourcenode a*) $\longrightarrow_{cd}$ *CFG-node nx*
by(*fastforce intro*:*SDG-cdep-edge*)
with ‹*nx* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› **have** *sourcenode a* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
by(*fastforce elim*!:*combine-SDG-slices.cases*
dest:*SDG-edge-sum-SDG-edge cdep-slice1 cdep-slice2*
intro:*combine-SDG-slices.intros*
simp:*HRB-slice-def SDG-to-CFG-set-def*)

with ‹*valid-edge a*›
have *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {*sourcenode a*}
by(*fastforce intro*!:*n-in-obs-intra*)
with ‹*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
‹*nx* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$› ‹*nx* ≠ *m*›
show *False* **by** *simp*
qed
next
case *False*
with ‹*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
have *m postdominates* (*sourcenode a*) **by**(*rule obs-intra-postdominate*)
with ‹¬ *m postdominates* (*sourcenode a*)› **show** *False* **by** *simp*
qed
qed
qed

lemma *obs-intra-finite*:*valid-node n* ⟹ *finite* (*obs-intra n* ⌊*HRB-slice S*⌋$_{CFG}$)
by(*fastforce dest*:*obs-intra-singleton-disj*[*of - S*])

lemma *obs-intra-singleton*:*valid-node n* ⟹ *card* (*obs-intra n* ⌊*HRB-slice S*⌋$_{CFG}$) ≤ *1*
by(*fastforce dest*:*obs-intra-singleton-disj*[*of - S*])

lemma *obs-intra-singleton-element*:
m ∈ *obs-intra n* ⌊*HRB-slice S*⌋$_{CFG}$ ⟹ *obs-intra n* ⌊*HRB-slice S*⌋$_{CFG}$ = {*m*}
apply −
apply(*frule in-obs-intra-valid*)
apply(*drule obs-intra-singleton-disj*) **apply** *auto*
done

lemma *obs-intra-the-element*:
m ∈ *obs-intra n* ⌊*HRB-slice S*⌋$_{CFG}$ ⟹ (*THE m*. *m* ∈ *obs-intra n* ⌊*HRB-slice S*⌋$_{CFG}$) = *m*
by(*fastforce dest*:*obs-intra-singleton-element*)

lemma *obs-singleton-element*:
assumes *ms* ∈ *obs ns* ⌊*HRB-slice S*⌋$_{CFG}$ **and** ∀ *n* ∈ *set* (*tl ns*). *return-node n*
shows *obs ns* ⌊*HRB-slice S*⌋$_{CFG}$ = {*ms*}
proof −
from ‹*ms* ∈ *obs ns* ⌊*HRB-slice S*⌋$_{CFG}$› ‹∀ *n* ∈ *set* (*tl ns*). *return-node n*›
obtain *nsx n nsx' n'* **where** *ns* = *nsx*@*n*#*nsx'* **and** *ms* = *n'*#*nsx'*
and *split*:*n'* ∈ *obs-intra n* ⌊*HRB-slice S*⌋$_{CFG}$
∀ *nx* ∈ *set nsx'*. ∃ *nx'*. *call-of-return-node nx nx'* ∧ *nx'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$
∀ *xs x xs'*. *nsx* = *xs*@*x*#*xs'* ∧ *obs-intra x* ⌊*HRB-slice S*⌋$_{CFG}$ ≠ {}

$\longrightarrow (\exists x''. x'' \in set\ (xs'@[n]).\ \exists nx.\ call\text{-}of\text{-}return\text{-}node\ x''\ nx \wedge nx \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG})$
by(*erule obsE*)
from ‹$n' \in obs\text{-}intra\ n\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
have $obs\text{-}intra\ n\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{n'\}$
by(*fastforce intro*!:*obs-intra-singleton-element*)
{ fix *xs* **assume** $xs \neq ms$ **and** $xs \in obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
from ‹$xs \in obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$› ‹$\forall n \in set\ (tl\ ns).\ return\text{-}node\ n$›
obtain *zs z zs' z'* **where** $ns = zs@z\#zs'$ **and** $xs = z'\#zs'$
and $z' \in obs\text{-}intra\ z\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
and $\forall z' \in set\ zs'.\ \exists nx'.\ call\text{-}of\text{-}return\text{-}node\ z'\ nx' \wedge nx' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
and $\forall xs\ x\ xs'.\ zs = xs@x\#xs' \wedge obs\text{-}intra\ x\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} \neq \{\} \longrightarrow (\exists x'' \in set\ (xs'@[z]).\ \exists nx.\ call\text{-}of\text{-}return\text{-}node\ x''\ nx \wedge nx \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG})$
by(*erule obsE*)
with ‹$ns = nsx@n\#nsx'$› *split*
have $nsx = zs \wedge n = z \wedge nsx' = zs'$
by −(*rule obs-split-det*[*of* - - - - - - $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$],*fastforce*+)
with ‹$obs\text{-}intra\ n\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{n'\}$› ‹$z' \in obs\text{-}intra\ z\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
have $z' = n'$ **by** *simp*
with ‹$xs \neq ms$› ‹$ms = n'\#nsx'$› ‹$xs = z'\#zs'$› ‹$nsx = zs \wedge n = z \wedge nsx' = zs'$›
have *False* **by** *simp* **}**
with ‹$ms \in obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$› **show** *?thesis* **by** *fastforce*
qed

lemma *obs-finite*:$\forall n \in set\ (tl\ ns).\ return\text{-}node\ n \Longrightarrow finite\ (obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG})$
by(*cases* $obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{\}$,*auto dest*:*obs-singleton-element*[*of* - - *S*])

lemma *obs-singleton*:$\forall n \in set\ (tl\ ns).\ return\text{-}node\ n \Longrightarrow card\ (obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}) \leq 1$
by(*cases* $obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{\}$,*auto dest*:*obs-singleton-element*[*of* - - *S*])

lemma *obs-the-element*:
$[\![ms \in obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}; \forall n \in set\ (tl\ ns).\ return\text{-}node\ n]\!] \Longrightarrow (THE\ ms.\ ms \in obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}) = ms$
by(*cases* $obs\ ns\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{\}$,*auto dest*:*obs-singleton-element*[*of* - - *S*])

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*→$_\iota$* *n'*; *length as* = *x*; ∀ *as'*. *n* −*as'*→$_\iota$* *n'* ⟶ *x* ≤ *length as'*⟧
⟹ *distance n n' x*

lemma *every-path-distance*:
assumes *n* −*as*→$_\iota$* *n'*
obtains *x* **where** *distance n n' x* **and** *x* ≤ *length as*
proof(*atomize-elim*)
show ∃ *x*. *distance n n' x* ∧ *x* ≤ *length as*
proof(*cases* ∃ *as'*. *n* −*as'*→$_\iota$* *n'* ∧
(∀ *asx*. *n* −*asx*→$_\iota$* *n'* ⟶ *length as'* ≤ *length asx*))
case *True*
then obtain *as'*
where *n* −*as'*→$_\iota$* *n'* ∧ (∀ *asx*. *n* −*asx*→$_\iota$* *n'* ⟶ *length as'* ≤ *length asx*)
by *blast*
hence *n* −*as'*→$_\iota$* *n'* **and** *all*:∀ *asx*. *n* −*asx*→$_\iota$* *n'* ⟶ *length as'* ≤ *length asx*
by *simp-all*
hence *distance n n'* (*length as'*) **by**(*fastforce intro*:*distanceI*)
from ‹*n* −*as*→$_\iota$* *n'*› *all* **have** *length as'* ≤ *length as* **by** *fastforce*
with ‹*distance n n'* (*length as'*)› **show** *?thesis* **by** *blast*
next
case *False*
hence *all*:∀ *as'*. *n* −*as'*→$_\iota$* *n'* ⟶ (∃ *asx*. *n* −*asx*→$_\iota$* *n'* ∧ *length as'* > *length asx*)
by *fastforce*
have *wf* (*measure length*) **by** *simp*
from ‹*n* −*as*→$_\iota$* *n'*› **have** *as* ∈ {*as*. *n* −*as*→$_\iota$* *n'*} **by** *simp*
with ‹*wf* (*measure length*)› **obtain** *as'* **where** *as'* ∈ {*as*. *n* −*as*→$_\iota$* *n'*}
and *notin*:⋀*as''*. (*as''*,*as'*) ∈ (*measure length*) ⟹ *as''* ∉ {*as*. *n* −*as*→$_\iota$* *n'*}
by(*erule wfE-min*)
from ‹*as'* ∈ {*as*. *n* −*as*→$_\iota$* *n'*}› **have** *n* −*as'*→$_\iota$* *n'* **by** *simp*
with *all* **obtain** *asx* **where** *n* −*asx*→$_\iota$* *n'*
and *length as'* > *length asx*
by *blast*
with *notin* **have** *asx* ∉ {*as*. *n* −*as*→$_\iota$* *n'*} **by** *simp*
hence ¬ *n* −*asx*→$_\iota$* *n'* **by** *simp*
with ‹*n* −*asx*→$_\iota$* *n'*› **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
qed
qed

lemma *distance-det*:
⟦*distance n n' x*; *distance n n' x'*⟧ ⟹ *x* = *x'*
apply(*erule distance.cases*)+ **apply** *clarsimp*

apply(*erule-tac x=asa* **in** *allE*) **apply**(*erule-tac x=as* **in** *allE*)
by *simp*

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(*rule ex-ex1I*)
show ∃ *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 −
have (∃ *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*)) =
(∃ *nx*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧ *distance* (*targetnode a'*) *n' x* ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧ *targetnode a'* = *nx*)
apply(*unfold some-eq-ex*[*of* λ*nx*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *n' x* ∧ *valid-edge a'* ∧ *intra-kind*(*kind a'*) ∧
targetnode a' = *nx*])
by *simp*
also have . . .
using ‹*valid-edge a*› ‹*intra-kind*(*kind a*)› ‹*distance* (*targetnode a*) *n' x*›
by *blast*
finally show *?thesis* **.**
qed
next
fix *a' ax*
assume *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*)
and *sourcenode a* = *sourcenode ax* ∧
distance (*targetnode ax*) *n' x* ∧ *valid-edge ax* ∧ *intra-kind*(*kind ax*) ∧
targetnode ax = (*SOME nx*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *n' x* ∧

```
                                valid-edge a' ∧ intra-kind(kind a') ∧
                                targetnode a' = nx)
  thus a' = ax by(fastforce intro!:edge-det)
qed


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(atomize-elim)
  show ∃ a. valid-edge a ∧ n = sourcenode a ∧ intra-kind(kind a) ∧
    distance (targetnode a) n' (x − 1) ∧
    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(rule ccontr)
    assume ¬ (∃ a. valid-edge a ∧ n = sourcenode a ∧ intra-kind(kind a) ∧
                distance (targetnode a) n' (x − 1) ∧
                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))
    hence imp:∀ a. valid-edge a ∧ n = sourcenode a ∧ intra-kind(kind a) ∧
                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)
              ⟶ ¬ distance (targetnode a) n' (x − 1) by blast
    from ‹distance n n' x› obtain as where n −as→ι* n' and x = length as
      and all:∀ as'. n −as'→ι* n' ⟶ x ≤ length as'
      by(auto elim:distance.cases)
    from ‹n −as→ι* n'› have n −as→* n' and ∀ a ∈ set as. intra-kind(kind a)
      by(simp-all add:intra-path-def)
    from this ‹x = length as› all imp show False
    proof(induct rule:path.induct)
      case (empty-path n)
      from ‹x = length []› ‹x ≠ 0› show False by simp
    next
      case (Cons-path n'' as n' a n)
      note imp = ‹∀ a. valid-edge a ∧ n = sourcenode a ∧ intra-kind (kind a) ∧
                  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)
                    ⟶ ¬ distance (targetnode a) n' (x − 1)⟩
      note all = ⟨∀ as'. n −as'→ι* n' ⟶ x ≤ length as'⟩
      from ⟨∀ a∈set (a#as). intra-kind (kind a)⟩
      have intra-kind (kind a) and ∀ a∈set as. intra-kind (kind a)
        by simp-all
      from ⟨n'' −as→* n'⟩ ⟨∀ a∈set as. intra-kind (kind a)⟩
      have n'' −as→ι* n' by(simp add:intra-path-def)
      then obtain y where distance n'' n' y
        and y ≤ length as by(erule every-path-distance)
      from ⟨distance n'' n' y⟩ obtain as' where n'' −as'→ι* n'
        and y = length as' by(auto elim:distance.cases)
      hence n'' −as'→* n' and ∀ a∈set as'. intra-kind (kind a)
        by(simp-all add:intra-path-def)
      show False
      proof(cases y < length as)
        case True
       from ⟨valid-edge a⟩ ⟨sourcenode a = n⟩ ⟨targetnode a = n''⟩ ⟨n'' −as'→* n'⟩
        have n −a#as'→* n' by −(rule path.Cons-path)
        with ⟨∀ a∈set as'. intra-kind (kind a)⟩ ⟨intra-kind (kind a)⟩
        have n −a#as'→ι* n' by(simp add:intra-path-def)
        with all have x ≤ length (a#as') by blast
        with ⟨x = length (a#as)⟩ True ⟨y = length as'⟩ show False by simp
      next
        case False
        with ⟨y ≤ length as⟩ ⟨x = length (a#as)⟩ have y = x − 1 by simp
        from ⟨targetnode a = n''⟩ ⟨distance n'' n' y⟩
        have distance (targetnode a) n' y by simp
        with ⟨valid-edge a⟩ ⟨intra-kind(kind a)⟩
        obtain a' where sourcenode a = sourcenode a'
          and distance (targetnode a') n' y and valid-edge a'
          and intra-kind(kind a')
          and targetnode a' = (SOME nx. ∃ a'. sourcenode a = sourcenode a' ∧
                                        distance (targetnode a') n' y ∧
                                        valid-edge a' ∧ intra-kind(kind a') ∧
                                        targetnode a' = nx)
          by(auto dest:only-one-SOME-dist-edge)
        with imp ⟨sourcenode a = n⟩ ⟨y = x − 1⟩ show False by fastforce
      qed
    qed
  qed
qed

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 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*
by(*induct fs arbitrary*:*x*)(*auto*)

lemma [*simp*]: *length* (*cspp m S fs*) = *length fs*
by(*simp add*:*cspp-def*)

lemma *csppa-Formal-in-notin-slice*:
⟦*x* < *length fs*; *Formal-in*(*m*,*x* + *i*) ∉ *S*⟧
⟹ (*csppa m S i fs*)!*x* = *empty*
by(*induct fs arbitrary*:*i x*,*auto simp*:*nth-Cons'*)

lemma *csppa-Formal-in-in-slice*:
⟦*x* < *length fs*; *Formal-in*(*m*,*x* + *i*) ∈ *S*⟧
⟹ (*csppa m S i fs*)!*x* = *fs*!*x*
by(*induct fs arbitrary*:*i x*,*auto simp*:*nth-Cons'*)

definition *map-merge* :: (*'var* ⇀ *'val*) ⇒ (*'var* ⇀ *'val*) ⇒ (*nat* ⇒ *bool*) ⇒
'var list ⇒ (*'var* ⇀ *'val*)
where *map-merge f g Q xs* ≡ (λ*V*. *if* (∃ *i*. *i* < *length xs* ∧ *xs*!*i* = *V* ∧ *Q i*) *then* *g V*
else f V)

definition *rspp* :: *'node* ⇒ *'node SDG-node set* ⇒ *'var list* ⇒
(*'var* ⇀ *'val*) ⇒ (*'var* ⇀ *'val*) ⇒ (*'var* ⇀ *'val*)
where *rspp m S xs f g* ≡ *map-merge f* (*empty*(*ParamDefs m* [:=] *map g xs*))
(λ*i*. *Actual-out*(*m*,*i*) ∈ *S*) (*ParamDefs m*)

lemma *rspp-Actual-out-in-slice*:
assumes $x < length\ (ParamDefs\ (targetnode\ a))$ **and** *valid-edge a*
and $length\ (ParamDefs\ (targetnode\ a)) = length\ xs$
and $Actual\text{-}out\ (targetnode\ a,x) \in S$
shows $(rspp\ (targetnode\ a)\ S\ xs\ f\ g)\ ((ParamDefs\ (targetnode\ a))!x) = g(xs!x)$
proof −
from ‹*valid-edge a*› **have** $distinct(ParamDefs\ (targetnode\ a))$
by(*rule distinct-ParamDefs*)
from ‹$x < length\ (ParamDefs\ (targetnode\ a))$›
‹$length\ (ParamDefs\ (targetnode\ a)) = length\ xs$›
‹$distinct(ParamDefs\ (targetnode\ a))$›
have $(empty(ParamDefs\ (targetnode\ a)\ [:=]\ map\ g\ xs))$
$((ParamDefs\ (targetnode\ a))!x) = (map\ g\ xs)!x$
by(*fastforce intro*:*fun-upds-nth*)
with ‹$Actual\text{-}out(targetnode\ a,x) \in S$› ‹$x < length\ (ParamDefs\ (targetnode\ a))$›
‹$length\ (ParamDefs\ (targetnode\ a)) = length\ xs$› **show** *?thesis*
by(*fastforce simp*:*rspp-def map-merge-def*)
qed

lemma *rspp-Actual-out-notin-slice*:
assumes $x < length\ (ParamDefs\ (targetnode\ a))$ **and** *valid-edge a*
and $length\ (ParamDefs\ (targetnode\ a)) = length\ xs$
and $Actual\text{-}out((targetnode\ a),x) \notin S$
shows $(rspp\ (targetnode\ a)\ S\ xs\ f\ g)\ ((ParamDefs\ (targetnode\ a))!x) =$
$f((ParamDefs\ (targetnode\ a))!x)$
proof −
from ‹*valid-edge a*› **have** $distinct(ParamDefs\ (targetnode\ a))$
by(*rule distinct-ParamDefs*)
from ‹$x < length\ (ParamDefs\ (targetnode\ a))$›
‹$length\ (ParamDefs\ (targetnode\ a)) = length\ xs$›
‹$distinct(ParamDefs\ (targetnode\ a))$›
have $(empty(ParamDefs\ (targetnode\ a)\ [:=]\ map\ g\ xs))$
$((ParamDefs\ (targetnode\ a))!x) = (map\ g\ xs)!x$
by(*fastforce intro*:*fun-upds-nth*)
with ‹$Actual\text{-}out((targetnode\ a),x) \notin S$› ‹$distinct(ParamDefs\ (targetnode\ a))$›
‹$x < length\ (ParamDefs\ (targetnode\ a))$›
show *?thesis* **by**(*fastforce simp*:*rspp-def map-merge-def nth-eq-iff-index-eq*)
qed

1.12.2 Defining the sliced edge kinds

primrec *slice-kind-aux* :: *'node* ⇒ *'node* ⇒ *'node SDG-node set* ⇒
$('var,'val,'ret,'pname)\ edge\text{-}kind$ ⇒ $('var,'val,'ret,'pname)\ edge\text{-}kind$
where $slice\text{-}kind\text{-}aux\ m\ m'\ S\ \Uparrow f = (if\ m \in \lfloor S \rfloor_{CFG}\ then\ \Uparrow f\ else\ \Uparrow id)$
| $slice\text{-}kind\text{-}aux\ m\ m'\ S\ (Q)_{\surd} = (if\ m \in \lfloor S \rfloor_{CFG}\ then\ (Q)_{\surd}\ else$
$(if\ obs\text{-}intra\ m\ \lfloor S \rfloor_{CFG} = \{\}\ then$
$(let\ mex = (THE\ mex.\ method\text{-}exit\ mex \wedge get\text{-}proc\ m = get\text{-}proc\ mex)\ in$
$(if\ (\exists x.\ distance\ m'\ mex\ x \wedge distance\ m\ mex\ (x + 1) \wedge$
$(m' = (SOME\ mx'.\ \exists a'.\ m = sourcenode\ a' \wedge$

distance (*targetnode a'*) *mex x* ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧
targetnode a' = *mx'*)))
then (λ*cf*. *True*)$_{\surd}$ *else* (λ*cf*. *False*)$_{\surd}$))
else (*let mx* = *THE mx*. *mx* ∈ *obs-intra m* ⌊*S*⌋$_{CFG}$ *in*
(*if* (∃ *x*. *distance m' mx x* ∧ *distance m mx* (*x* + *1*) ∧
(*m'* = (*SOME mx'*. ∃ *a'*. *m* = *sourcenode a'* ∧
distance (*targetnode a'*) *mx x* ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧
targetnode a' = *mx'*)))
then (λ*cf*. *True*)$_{\surd}$ *else* (λ*cf*. *False*)$_{\surd}$))))
| *slice-kind-aux m m' S* (*Q*:*r*↪$_p$*fs*) = (*if m* ∈ ⌊*S*⌋$_{CFG}$ *then* (*Q*:*r*↪$_p$(*cspp m' S fs*))
else ((λ*cf*. *False*):*r*↪$_p$*fs*))
| *slice-kind-aux m m' S* (*Q*↩$_p$*f*) = (*if m* ∈ ⌊*S*⌋$_{CFG}$ *then*
(*let outs* = *THE outs*. ∃ *ins*. (*p*,*ins*,*outs*) ∈ *set procs in*
(*Q*↩$_p$(λ*cf cf'*. *rspp m' S outs cf' cf*)))
else ((λ*cf*. *True*)↩$_p$(λ*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*
by(*fastforce simp*:*intra-kind-def slice-kind-def*)

lemma *slice-kind-Upd*:
⟦*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$; *kind a* = ⇑*f*⟧ ⟹ *slice-kind S a* = ⇑*id*
by(*simp add*:*slice-kind-def*)

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' ∧ *intra-kind*(*kind a'*) ∧

targetnode $a' = n'$)
shows *slice-kind* $S\ a = (\lambda s.\ True)_\surd$
proof −
from ‹*method-exit mex*› ‹*get-proc* (*sourcenode* a) = *get-proc mex*›
have $mex = (THE\ mex.$ *method-exit mex* $\wedge$ *get-proc* (*sourcenode* a) = *get-proc mex*)
by(*auto intro*!:*the-equality*[*THEN sym*] *intro*:*method-exit-unique*)
with ‹*sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$› ‹*kind* $a = (Q)_\surd$›
‹*obs-intra* (*sourcenode* a) $\lfloor$*HRB-slice* $S\rfloor_{CFG} = \{\}$›
have *slice-kind* $S\ a =$
(*if* $(\exists x.$ *distance* (*targetnode* a) *mex* $x \wedge$ *distance* (*sourcenode* a) *mex* $(x + 1)$ $\wedge$
(*targetnode* $a = (SOME\ mx'. \exists a'.$ *sourcenode* $a =$ *sourcenode* $a' \wedge$
distance (*targetnode* a') *mex* $x \wedge$ *valid-edge* $a' \wedge$ *intra-kind*(*kind* a') $\wedge$
targetnode $a' = mx'$))) *then* $(\lambda cf.\ True)_\surd$ *else* $(\lambda cf.\ False)_\surd$)
by(*simp add*:*slice-kind-def Let-def*)
with ‹*distance* (*targetnode* a) *mex* x› ‹*distance* (*sourcenode* a) *mex* $(x + 1)$›
‹*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 $a' = n'$)›
show *?thesis* **by** *fastforce*
qed

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 −
from ‹*method-exit mex*› ‹*get-proc* (*sourcenode* a) = *get-proc mex*›
have $mex = (THE\ mex.$ *method-exit mex* $\wedge$ *get-proc* (*sourcenode* a) = *get-proc mex*)
by(*auto intro*!:*the-equality*[*THEN sym*] *intro*:*method-exit-unique*)
with ‹*sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$› ‹*kind* $a = (Q)_\surd$›
‹*obs-intra* (*sourcenode* a) $\lfloor$*HRB-slice* $S\rfloor_{CFG} = \{\}$›
have *slice-kind* $S\ a =$
(*if* $(\exists x.$ *distance* (*targetnode* a) *mex* $x \wedge$ *distance* (*sourcenode* a) *mex* $(x + 1)$ $\wedge$
(*targetnode* $a = (SOME\ mx'. \exists a'.$ *sourcenode* $a =$ *sourcenode* $a' \wedge$
distance (*targetnode* a') *mex* $x \wedge$ *valid-edge* $a' \wedge$ *intra-kind*(*kind* a') $\wedge$
targetnode $a' = mx'$))) *then* $(\lambda cf.\ True)_\surd$ *else* $(\lambda cf.\ False)_\surd$)
by(*simp add*:*slice-kind-def Let-def*)

with ‹*distance* (*targetnode a*) *mex x*› ‹*distance* (*sourcenode a*) *mex* (*x* + *1*)›
‹*targetnode a* ≠ (*SOME n′*. ∃ *a′*. *sourcenode a* = *sourcenode a′* ∧
distance (*targetnode a′*) *mex x* ∧
valid-edge a′ ∧ *intra-kind*(*kind a′*) ∧
targetnode a′ = *n′*)›
show *?thesis* **by**(*auto dest*:*distance-det*)
qed

lemma *slice-kind-Pred-empty-obs-not-nearer*:
assumes *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *kind a* = (*Q*)$_{\surd}$
and *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {}
and *method-exit mex* **and** *get-proc* (*sourcenode a*) = *get-proc mex*
and *dist*:*distance* (*sourcenode a*) *mex* (*x* + *1*) ¬ *distance* (*targetnode a*) *mex x*
shows *slice-kind S a* = (λ*s*. *False*)$_{\surd}$
proof −
from ‹*method-exit mex*› ‹*get-proc* (*sourcenode a*) = *get-proc mex*›
have *mex* = (*THE mex*. *method-exit mex* ∧ *get-proc* (*sourcenode a*) = *get-proc mex*)
by(*auto intro*!:*the-equality*[*THEN sym*] *intro*:*method-exit-unique*)
moreover
from *dist* **have** ¬ (∃ *x*. *distance* (*targetnode a*) *mex x* ∧
distance (*sourcenode a*) *mex* (*x* + *1*))
by(*fastforce dest*:*distance-det*)
ultimately show *?thesis* **using** *assms* **by**(*auto simp*:*slice-kind-def Let-def*)
qed

lemma *slice-kind-Pred-obs-nearer-SOME*:
assumes *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *kind a* = (*Q*)$_{\surd}$
and *m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$
and *distance* (*targetnode a*) *m x distance* (*sourcenode a*) *m* (*x* + *1*)
and *targetnode a* = (*SOME n′*. ∃ *a′*. *sourcenode a* = *sourcenode a′* ∧
distance (*targetnode a′*) *m x* ∧
valid-edge a′ ∧ *intra-kind*(*kind a′*) ∧
targetnode a′ = *n′*)
shows *slice-kind S a* = (λ*s*. *True*)$_{\surd}$
proof −
from ‹*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
have *m* = (*THE m*. *m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$)
by(*rule obs-intra-the-element*[*THEN sym*])
with *assms* **show** *?thesis* **by**(*auto simp*:*slice-kind-def Let-def*)
qed

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

and *targetnode a* ≠ (*SOME nx'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *m x* ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧
targetnode a' = *nx'*)
shows *slice-kind S a* = (λ*s*. *False*)$_{\surd}$
proof −
from ‹*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
have *m* = (*THE m*. *m* ∈ *obs-intra* (*sourcenode a*) (⌊*HRB-slice S*⌋$_{CFG}$))
by(*rule obs-intra-the-element*[*THEN sym*])
with *assms* **show** *?thesis* **by**(*auto dest*:*distance-det simp*:*slice-kind-def Let-def*)
qed

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 −
from *in-obs* **have** *the*:*m* = (*THE m*. *m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$)
by(*rule obs-intra-the-element*[*THEN sym*])
from *dist* **have** ¬ (∃ *x*. *distance* (*targetnode a*) *m x* ∧
distance (*sourcenode a*) *m* (*x* + *1*))
by(*fastforce dest*:*distance-det*)
with ‹*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*kind a* = (*Q*)$_{\surd}$› *in-obs the* **show** *?thesis*
by(*auto simp*:*slice-kind-def Let-def*)
qed

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(*atomize-elim*)
show ∃ *Q'*. *slice-kind S a* = (*Q'*)$_{\surd}$ ∧ (*Q'* = (λ*s*. *False*) ∨ *Q'* = (λ*s*. *True*))
proof(*cases obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {})
case *True*
from ‹*valid-edge a*› **have** *valid-node* (*sourcenode a*) **by** *simp*
then obtain *as* **where** *sourcenode a* −*as*→$_{\surd}$* (*-Exit-*) **by**(*fastforce dest*:*Exit-path*)
then obtain *as' mex* **where** *sourcenode a* −*as'*→$_{\iota}$* *mex* **and** *method-exit mex*
by −(*erule valid-Exit-path-intra-path*)
from ‹*sourcenode a* −*as'*→$_{\iota}$* *mex*› **have** *get-proc* (*sourcenode a*) = *get-proc mex*
by(*rule intra-path-get-procs*)

show *?thesis*
proof(*cases* $\exists x.$ *distance* (*targetnode a*) *mex x* $\wedge$
distance (*sourcenode a*) *mex* (*x* + *1*))
case *True*
then obtain *x* **where** *distance* (*targetnode a*) *mex x*
and *distance* (*sourcenode a*) *mex* (*x* + *1*) **by** *blast*
show *?thesis*
proof(*cases 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'*))
case *True*
with ‹*sourcenode a* $\notin$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*kind a* = $(Q)_{\surd}$›
‹*obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {}›
‹*method-exit mex*› ‹*get-proc* (*sourcenode a*) = *get-proc mex*›
‹*distance* (*targetnode a*) *mex x*› ‹*distance* (*sourcenode a*) *mex* (*x* + *1*)›
have *slice-kind S a* = $(\lambda s.\ True)_{\surd}$
by(*rule slice-kind-Pred-empty-obs-nearer-SOME*)
thus *?thesis* **by** *simp*
next
case *False*
with ‹*sourcenode a* $\notin$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*kind a* = $(Q)_{\surd}$›
‹*obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {}›
‹*method-exit mex*› ‹*get-proc* (*sourcenode a*) = *get-proc mex*›
‹*distance* (*targetnode a*) *mex x*› ‹*distance* (*sourcenode a*) *mex* (*x* + *1*)›
have *slice-kind S a* = $(\lambda s.\ False)_{\surd}$
by(*rule slice-kind-Pred-empty-obs-nearer-not-SOME*)
thus *?thesis* **by** *simp*
qed
next
case *False*
from ‹*method-exit mex*› ‹*get-proc* (*sourcenode a*) = *get-proc mex*›
have *mex* = (*THE mex*. *method-exit mex* $\wedge$ *get-proc* (*sourcenode a*) = *get-proc mex*)
by(*auto intro*!:*the-equality*[*THEN sym*] *intro*:*method-exit-unique*)
with ‹*sourcenode a* $\notin$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*kind a* = $(Q)_{\surd}$›
‹*obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {}› *False*
have *slice-kind S a* = $(\lambda s.\ False)_{\surd}$
by(*auto simp*:*slice-kind-def Let-def*)
thus *?thesis* **by** *simp*
qed
next
case *False*
then obtain *m* **where** *m* $\in$ *obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by** *blast*
show *?thesis*
proof(*cases* $\exists x.$ *distance* (*targetnode a*) *m x* $\wedge$
distance (*sourcenode a*) *m* (*x* + *1*))
case *True*

then obtain x **where** *distance* (*targetnode a*) *m x*
and *distance* (*sourcenode a*) *m* (*x* + *1*) **by** *blast*
show *?thesis*
proof(*cases targetnode a* = (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *m x* ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧
targetnode a' = *n'*))
case *True*
with ‹*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*kind a* = (*Q*)$_{\surd}$›
‹*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
‹*distance* (*targetnode a*) *m x*› ‹*distance* (*sourcenode a*) *m* (*x* + *1*)›
have *slice-kind S a* = (λ*s*. *True*)$_{\surd}$
by(*rule slice-kind-Pred-obs-nearer-SOME*)
thus *?thesis* **by** *simp*
next
case *False*
with ‹*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*kind a* = (*Q*)$_{\surd}$›
‹*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
‹*distance* (*targetnode a*) *m x*› ‹*distance* (*sourcenode a*) *m* (*x* + *1*)›
have *slice-kind S a* = (λ*s*. *False*)$_{\surd}$
by(*rule slice-kind-Pred-obs-nearer-not-SOME*)
thus *?thesis* **by** *simp*
qed
next
case *False*
from ‹*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
have *m* = (*THE m*. *m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$)
by(*rule obs-intra-the-element*[*THEN sym*])
with ‹*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*kind a* = (*Q*)$_{\surd}$› *False*
‹*m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$›
have *slice-kind S a* = (λ*s*. *False*)$_{\surd}$
by(*auto simp*:*slice-kind-def Let-def*)
thus *?thesis* **by** *simp*
qed
qed
qed

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*
by(*simp add*:*slice-kind-def*)

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*)
by(*simp add*:*slice-kind-def*)

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*) *empty*
proof −
from ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
have *slice-kind S a* = *Q*:*r*↪$_p$(*cspp* (*targetnode a*) (*HRB-slice S*) *fs*)
by(*simp add*:*slice-kind-def*)
from ‹∀ *x* < *length fs*. *Formal-in*(*targetnode a*,*x*) ∉ *HRB-slice S*›
have *cspp* (*targetnode a*) (*HRB-slice S*) *fs* = *replicate* (*length fs*) *empty*
by(*fastforce intro*:*nth-equalityI csppa-Formal-in-notin-slice simp*:*cspp-def*)
with ‹*slice-kind S a* = *Q*:*r*↪$_p$(*cspp* (*targetnode a*) (*HRB-slice S*) *fs*)›
show *?thesis* **by** *simp*
qed

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 −
from ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
have *slice-kind S a* = *Q*:*r*↪$_p$(*cspp* (*targetnode a*) (*HRB-slice S*) *fs*)
by(*simp add*:*slice-kind-def*)
from ‹∀ *x* < *length fs*. *Formal-in*(*targetnode a*,*x*) ∈ *HRB-slice S*›
have *cspp* (*targetnode a*) (*HRB-slice S*) *fs* = *fs*
by(*fastforce intro*:*nth-equalityI csppa-Formal-in-in-slice simp*:*cspp-def*)
with ‹*slice-kind S a* = *Q*:*r*↪$_p$(*cspp* (*targetnode a*) (*HRB-slice S*) *fs*)›
show *?thesis* **by** *simp*
qed

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* = (λ*s*. *True*)$_\surd$
proof −
from ‹*valid-edge a'*› ‹*kind a'* = *Q*:*r*↪$_p$*fs*› **obtain** *a''*
where *a''* ∈ *get-return-edges a'*
by(*fastforce dest*:*get-return-edge-call*)
with ‹*valid-edge a'*› **obtain** *ax* **where** *valid-edge ax*
and *sourcenode ax* = *sourcenode a'* **and** *targetnode ax* = *targetnode a''*
and *kind ax* = (λ*cf*. *False*)$_\surd$
by(*fastforce dest*:*call-return-node-edge*)
from ‹*valid-edge a'*› ‹*kind a'* = *Q*:*r*↪$_p$*fs*›
have ∃!*a''*. *valid-edge a''* ∧ *sourcenode a''* = *sourcenode a'* ∧
intra-kind(*kind a''*)

by(*rule call-only-one-intra-edge*)
with ‹*valid-edge a*› ‹*sourcenode a' = sourcenode a*› ‹*intra-kind* (*kind a*)›
have *all*:∀ *a''*. *valid-edge a''* ∧ *sourcenode a'' = sourcenode a'* ∧
intra-kind(*kind a''*) ⟶ *a'' = a* **by** *fastforce*
with ‹*valid-edge ax*› ‹*sourcenode ax = sourcenode a'*› ‹*kind ax* = (λ*cf*. *False*)$_{\surd}$›
have [*simp*]:*ax = a* **by**(*fastforce simp*:*intra-kind-def*)
show *?thesis*
proof(*cases obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {})
case *True*
from ‹*valid-edge a*› **have** *valid-node* (*sourcenode a*) **by** *simp*
then obtain *asx* **where** *sourcenode a* −*asx*→$_{\surd}$* (*-Exit-*) **by**(*fastforce dest*:*Exit-path*)
then obtain *as pex* **where** *sourcenode a*−*as*→$_{\iota}$* *pex* **and** *method-exit pex*
by −(*erule valid-Exit-path-intra-path*)
from ‹*sourcenode a*−*as*→$_{\iota}$* *pex*› **have** *get-proc* (*sourcenode a*) = *get-proc pex*
by(*rule intra-path-get-procs*)
from ‹*sourcenode a*−*as*→$_{\iota}$* *pex*› **obtain** *x* **where** *distance* (*sourcenode a*) *pex*
x
and *x* ≤ *length as* **by**(*erule every-path-distance*)
from ‹*method-exit pex*› **have** *sourcenode a* ≠ *pex*
proof(*rule method-exit-cases*)
assume *pex* = (*-Exit-*)
show *?thesis*
proof
assume *sourcenode a* = *pex*
with ‹*pex* = (*-Exit-*)› **have** *sourcenode a* = (*-Exit-*) **by** *simp*
with ‹*valid-edge a*› **show** *False* **by**(*rule Exit-source*)
qed
next
fix *ax Qx px fx*
assume *pex = sourcenode ax* **and** *valid-edge ax* **and** *kind ax = Qx*↩$_{px}$*fx*
hence ∀ *a'*. *valid-edge a'* ∧ *sourcenode a' = sourcenode ax* ⟶
(∃ *Qx' fx'*. *kind a' = Qx'*↩$_{px}$*fx'*) **by** −(*rule return-edges-only*)
with ‹*valid-edge a*› ‹*intra-kind* (*kind a*)› ‹*pex = sourcenode ax*›
show *?thesis* **by**(*fastforce simp*:*intra-kind-def*)
qed
have *x* ≠ *0*
proof
assume *x = 0*
with ‹*distance* (*sourcenode a*) *pex x*› **have** *sourcenode a = pex*
by(*fastforce elim*:*distance.cases simp*:*intra-path-def*)
with ‹*sourcenode a* ≠ *pex*› **show** *False* **by** *simp*
qed
with ‹*distance* (*sourcenode a*) *pex x*› **obtain** *ax'* **where** *valid-edge ax'*
and *sourcenode a = sourcenode ax'* **and** *intra-kind*(*kind ax'*)
and *distance* (*targetnode ax'*) *pex* (*x* − *1*)
and *Some*:*targetnode ax' = (SOME nx*. ∃ *a'*. *sourcenode ax' = sourcenode a'*
∧
distance (*targetnode a'*) *pex* (*x* − *1*) ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧

targetnode $a' = nx$)
by(*erule distance-successor-distance*)
from ‹*valid-edge* ax'› ‹*sourcenode* a = *sourcenode* ax'› ‹*intra-kind*(*kind* ax')›
‹*sourcenode* a' = *sourcenode* a› *all*
have [*simp*]:$ax' = a$ **by** *fastforce*
from ‹*sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$› ‹*kind* $ax = (\lambda cf.\ False)_{\surd}$›
True ‹*method-exit pex*› ‹*get-proc* (*sourcenode* a) = *get-proc pex*› ‹$x \neq 0$›
‹*distance* (*targetnode* ax') *pex* ($x - 1$)› ‹*distance* (*sourcenode* a) *pex* x› *Some*
show *?thesis* **by**(*fastforce elim*:*slice-kind-Pred-empty-obs-nearer-SOME*)
next
case *False*
then obtain m **where** $m \in$ *obs-intra* (*sourcenode* a) $\lfloor$*HRB-slice* $S\rfloor_{CFG}$ **by** *fastforce*
then obtain *as* **where** *sourcenode* $a - as \rightarrow_\iota * m$ **and** $m \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$
by −(*erule obs-intraE*)
from ‹*sourcenode* $a - as \rightarrow_\iota * m$› **obtain** x **where** *distance* (*sourcenode* a) m x
and $x \leq$ *length as* **by**(*erule every-path-distance*)
from ‹*sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$› ‹$m \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$›
have *sourcenode* $a \neq m$ **by** *fastforce*
have $x \neq 0$
proof
assume $x = 0$
with ‹*distance* (*sourcenode* a) m x› **have** *sourcenode* $a = m$
by(*fastforce elim*:*distance.cases simp*:*intra-path-def*)
with ‹*sourcenode* $a \neq m$› **show** *False* **by** *simp*
qed
with ‹*distance* (*sourcenode* a) m x› **obtain** ax' **where** *valid-edge* ax'
and *sourcenode* a = *sourcenode* ax' **and** *intra-kind*(*kind* ax')
and *distance* (*targetnode* ax') m ($x - 1$)
and *Some*:*targetnode* ax' = (*SOME* nx. $\exists a'$. *sourcenode* ax' = *sourcenode* $a' \wedge$
distance (*targetnode* a') m ($x - 1$) $\wedge$
valid-edge $a' \wedge$ *intra-kind*(*kind* a') $\wedge$
targetnode $a' = nx$)
by(*erule distance-successor-distance*)
from ‹*valid-edge* ax'› ‹*sourcenode* a = *sourcenode* ax'› ‹*intra-kind*(*kind* ax')›
‹*sourcenode* a' = *sourcenode* a› *all*
have [*simp*]:$ax' = a$ **by** *fastforce*
from ‹*sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$› ‹*kind* $ax = (\lambda cf.\ False)_{\surd}$›
‹$m \in$ *obs-intra* (*sourcenode* a) $\lfloor$*HRB-slice* $S\rfloor_{CFG}$› ‹$x \neq 0$›
‹*distance* (*targetnode* ax') m ($x - 1$)› ‹*distance* (*sourcenode* a) m x› *Some*
show *?thesis* **by**(*fastforce elim*:*slice-kind-Pred-obs-nearer-SOME*)
qed
qed

lemma *slice-kind-Return*:
⟦*sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$; *kind* $a = Q \hookleftarrow_p pf$⟧
⟹ *slice-kind* S $a = (\lambda cf.\ True) \hookleftarrow_p (\lambda cf\ cf'.\ cf')$

by(*simp add*:*slice-kind-def*)

lemma *slice-kind-Return-in-slice*:
⟦*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$; *valid-edge a*; *kind a* = *Q*↩$_p$*f*;
(*p*,*ins*,*outs*) ∈ *set procs*⟧
⟹ *slice-kind S a* = *Q*↩$_p$(λ*cf cf'*. *rspp* (*targetnode a*) (*HRB-slice S*) *outs cf'* *cf*)
by(*simp add*:*slice-kind-def*,*unfold formal-out-THE*,*simp*)

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-kind S a*) s_2 = s_2'
shows *length* s_1' = *length* s_2'
proof(*cases kind a rule*:*edge-kind-cases*)
case *Intra*
show *?thesis*
proof(*cases sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$)
case *True*
with *Intra assms* **show** *?thesis*
by(*cases* s_1)(*cases* s_2,*auto dest*:*slice-intra-kind-in-slice simp*:*intra-kind-def*)+
next
case *False*
with *Intra assms* **show** *?thesis*
by(*cases* s_1)(*cases* s_2,*auto dest*:*slice-kind-Upd*
elim:*kind-Predicate-notin-slice-slice-kind-Predicate simp*:*intra-kind-def*)+
qed
next
case (*Call Q r p fs*)
show *?thesis*
proof(*cases sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$)
case *True*
with *Call assms* **show** *?thesis*
by(*cases* s_1)(*cases* s_2,*auto dest*:*slice-kind-Call-in-slice*)+
next
case *False*
with *Call assms* **show** *?thesis*
by(*cases* s_1)(*cases* s_2,*auto dest*:*slice-kind-Call*)+
qed
next
case (*Return Q p f*)
show *?thesis*
proof(*cases sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$)
case *True*
from *Return* ‹*valid-edge a*› **obtain** *a' Q' r fs*
where *valid-edge a'* **and** *kind a'* = *Q'*:*r*↪$_p$*fs*
by −(*drule return-needs-call*,*auto*)
then obtain *ins outs* **where** (*p*,*ins*,*outs*) ∈ *set procs*

by(*fastforce dest*!:*callee-in-procs*)
with *True* ‹*valid-edge a*› *Return assms* **show** *?thesis*
by(*cases* s_1)(*cases* s_2,*auto dest*:*slice-kind-Return-in-slice split*:*list.split*)+
next
case *False*
with *Return assms* **show** *?thesis*
by(*cases* s_1)(*cases* s_2,*auto dest*:*slice-kind-Return split*:*list.split*)+
qed
qed

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(*rule ex-ex1I*)
show $\exists\, a'$. *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 −
have ($\exists\, a'$. *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'*)) =
($\exists\, n'$. $\exists\, a'$. *sourcenode a* = *sourcenode a'* $\wedge$ *distance* (*targetnode a'*) *mex x* $\wedge$
valid-edge a' $\wedge$ *intra-kind*(*kind a'*) $\wedge$ *targetnode a'* = *n'*)
apply(*unfold some-eq-ex*[*of* $\lambda n'$. $\exists\, a'$. *sourcenode a* = *sourcenode a'* $\wedge$
distance (*targetnode a'*) *mex x* $\wedge$
valid-edge a' $\wedge$ *intra-kind*(*kind a'*) $\wedge$
targetnode a' = *n'*])
by *simp*
also have ...
using ‹*valid-edge a*› ‹*intra-kind*(*kind a*)› ‹*distance* (*targetnode a*) *mex x*›
by *blast*
finally show *?thesis* **.**
qed
next
fix *a' ax*

assume *sourcenode a = sourcenode a′ ∧ distance (targetnode a′) mex x ∧*
valid-edge a′ ∧ intra-kind(kind a′) ∧
targetnode a′ = (SOME n′. ∃ a′. sourcenode a = sourcenode a′ ∧
distance (targetnode a′) mex x ∧
valid-edge a′ ∧ intra-kind(kind a′) ∧
targetnode a′ = n′)
and *sourcenode a = sourcenode ax ∧ distance (targetnode ax) mex x ∧*
valid-edge ax ∧ intra-kind(kind ax) ∧
targetnode ax = (SOME n′. ∃ a′. sourcenode a = sourcenode a′ ∧
distance (targetnode a′) mex x ∧
valid-edge a′ ∧ intra-kind(kind a′) ∧
targetnode a′ = n′)
thus *a′ = ax* **by**(*fastforce intro*!:*edge-det*)
qed

lemma *slice-kind-only-one-True-edge*:
assumes *sourcenode a = sourcenode a′* **and** *targetnode a ≠ targetnode a′*
and *valid-edge a* **and** *valid-edge a′* **and** *intra-kind (kind a)*
and *intra-kind (kind a′)* **and** *slice-kind S a* $= (\lambda s.\ True)_{\surd}$
shows *slice-kind S a′* $= (\lambda s.\ False)_{\surd}$
proof −
from *assms* **obtain** *Q Q′* **where** *kind a* $= (Q)_{\surd}$
and *kind a′* $= (Q')_{\surd}$ **and** *det*:$\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s)$
by(*auto dest*:*deterministic*)
show *?thesis*
proof(*cases sourcenode a* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$)
case *True*
with ‹*slice-kind S a* $= (\lambda s.\ True)_{\surd}$› ‹*kind a* $= (Q)_{\surd}$› **have** $Q = (\lambda s.\ True)$
by(*simp add*:*slice-kind-def Let-def*)
with *det* **have** $Q' = (\lambda s.\ False)$ **by**(*simp add*:*fun-eq-iff*)
with *True* ‹*kind a′* $= (Q')_{\surd}$› ‹*sourcenode a = sourcenode a′*› **show** *?thesis*
by(*simp add*:*slice-kind-def Let-def*)
next
case *False*
hence *sourcenode a* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by** *simp*
thus *?thesis*
proof(*cases obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG} = \{\}$)
case *True*
with ‹*sourcenode a* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*slice-kind S a* $= (\lambda s.\ True)_{\surd}$›
‹*kind a* $= (Q)_{\surd}$›
obtain *mex x* **where** *mex*:*mex* = (*THE mex. method-exit mex ∧*
get-proc (*sourcenode a*) = *get-proc mex*)
and *dist*:*distance* (*targetnode a*) *mex x distance* (*sourcenode a*) *mex* (*x* + *1*)
and *target*:*targetnode a* = (*SOME n′. ∃ a′. sourcenode a = sourcenode a′ ∧*
distance (*targetnode a′*) *mex x ∧*
valid-edge a′ ∧ intra-kind(*kind a′*) *∧*
targetnode a′ = n′)

by(*auto simp*:*slice-kind-def Let-def fun-eq-iff split*:*split-if-asm*)
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)› ‹*distance* (*targetnode a*) *mex x*›
have *ex1*:∃!*a'*. *sourcenode a* = *sourcenode a'* ∧ *distance* (*targetnode a'*) *mex x* ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧
targetnode a' = (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *mex x* ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧
targetnode a' = *n'*)
by(*rule only-one-SOME-edge*)
have *targetnode a'* ≠ (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *mex x* ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧
targetnode a' = *n'*)
proof(*rule ccontr*)
assume ¬ *targetnode a'* ≠ (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *mex x* ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧
targetnode a' = *n'*)
hence *targetnode a'* = (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *mex x* ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧
targetnode a' = *n'*)
by *simp*
with *ex1 target* ‹*sourcenode a* = *sourcenode a'*› ‹*valid-edge a*› ‹*valid-edge a'*›
‹*intra-kind*(*kind a*)› ‹*intra-kind*(*kind a'*)› ‹*distance* (*targetnode a*) *mex x*›
have *a* = *a'* **by** *fastforce*
with ‹*targetnode a* ≠ *targetnode a'*› **show** *False* **by** *simp*
qed
with ‹*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› *True* ‹*kind a'* = $(Q')_{\surd}$›
‹*sourcenode a* = *sourcenode a'*› *mex dist*
show *?thesis* **by**(*auto dest*:*distance-det*
simp:*slice-kind-def Let-def fun-eq-iff split*:*split-if-asm*)
next
case *False*
hence *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ ≠ {} .
then obtain *m* **where** *m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ **by** *auto*
hence *m* = (*THE m*. *m* ∈ *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$)
by(*auto dest*:*obs-intra-the-element*)
with ‹*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ ≠ {}›
‹*slice-kind S a* = $(\lambda s.\ True)_{\surd}$› ‹*kind a* = $(Q)_{\surd}$›
obtain *x x'* **where** *distance* (*targetnode a*) *m x*
distance (*sourcenode a*) *m* (*x* + *1*)
and *target*:*targetnode a* = (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *m x* ∧
valid-edge a' ∧ *intra-kind*(*kind a'*) ∧

```
                                        targetnode a' = n')
    by(auto simp:slice-kind-def Let-def fun-eq-iff split:split-if-asm)
  show ?thesis
  proof(cases distance (targetnode a') m x)
    case False
    with ‹sourcenode a ∉ ⌊HRB-slice S⌋CFG› ‹kind a' = (Q')√›
      ‹m ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG›
      ‹distance (targetnode a) m x› ‹distance (sourcenode a) m (x + 1)›
      ‹sourcenode a = sourcenode a'› show ?thesis
      by(fastforce intro:slice-kind-Pred-obs-not-nearer)
  next
    case True
    from ‹valid-edge a› ‹intra-kind(kind a)› ‹distance (targetnode a) m x›
      ‹distance (sourcenode a) m (x + 1)›
    have ex1:∃!a'. sourcenode a = sourcenode a' ∧
          distance (targetnode a') m x ∧ valid-edge a' ∧ intra-kind(kind a') ∧
          targetnode a' = (SOME nx. ∃ a'. sourcenode a = sourcenode a' ∧
                                        distance (targetnode a') m x ∧
                                        valid-edge a' ∧ intra-kind(kind a') ∧
                                        targetnode a' = nx)
      by −(rule only-one-SOME-dist-edge)
    have targetnode a' ≠ (SOME n'. ∃ a'. sourcenode a = sourcenode a' ∧
                                        distance (targetnode a') m x ∧
                                        valid-edge a' ∧ intra-kind(kind a') ∧
                                        targetnode a' = n')
    proof(rule ccontr)
      assume ¬ targetnode a' ≠ (SOME n'. ∃ a'. sourcenode a = sourcenode a'
∧
                                        distance (targetnode a') m x ∧
                                        valid-edge a' ∧ intra-kind(kind a') ∧
                                        targetnode a' = n')
      hence targetnode a' = (SOME n'. ∃ a'. sourcenode a = sourcenode a' ∧
                                        distance (targetnode a') m x ∧
                                        valid-edge a' ∧ intra-kind(kind a') ∧
                                        targetnode a' = n')
        by simp
      with ex1 target ‹sourcenode a = sourcenode a'›
        ‹valid-edge a› ‹valid-edge a'› ‹intra-kind(kind a)› ‹intra-kind(kind a')›
        ‹distance (targetnode a) m x› ‹distance (sourcenode a) m (x + 1)›
      have a = a' by auto
      with ‹targetnode a ≠ targetnode a'› show False by simp
    qed
    with ‹sourcenode a ∉ ⌊HRB-slice S⌋CFG›
      ‹kind a' = (Q')√› ‹m ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG›
      ‹distance (targetnode a) m x› ‹distance (sourcenode a) m (x + 1)›
      True ‹sourcenode a = sourcenode a'› show ?thesis
      by(fastforce intro:slice-kind-Pred-obs-nearer-not-SOME)
  qed
qed
```

```
  qed
qed


lemma slice-deterministic:
  assumes valid-edge a and valid-edge a'
  and intra-kind (kind a) and intra-kind (kind a')
  and sourcenode a = sourcenode a' and targetnode a ≠ targetnode a'
  obtains Q Q' where slice-kind S a = (Q)_√ and slice-kind S a' = (Q')_√
  and ∀ s. (Q s ⟶ ¬ Q' s) ∧ (Q' s ⟶ ¬ Q s)
proof(atomize-elim)
  from assms obtain Q Q'
    where kind a = (Q)_√ and kind a' = (Q')_√
    and det:∀ s. (Q s ⟶ ¬ Q' s) ∧ (Q' s ⟶ ¬ Q s)
    by(auto dest:deterministic)
  show ∃ Q Q'. slice-kind S a = (Q)_√ ∧ slice-kind S a' = (Q')_√ ∧
               (∀ s. (Q s ⟶ ¬ Q' s) ∧ (Q' s ⟶ ¬ Q s))
  proof(cases sourcenode a ∈ ⌊HRB-slice S⌋_CFG)
    case True
    with ‹kind a = (Q)_√› have slice-kind S a = (Q)_√
      by(simp add:slice-kind-def Let-def)
    from True ‹kind a' = (Q')_√› ‹sourcenode a = sourcenode a'›
    have slice-kind S a' = (Q')_√
      by(simp add:slice-kind-def Let-def)
    with ‹slice-kind S a = (Q)_√› det show ?thesis by blast
  next
    case False
    with ‹kind a = (Q)_√›
    have slice-kind S a = (λs. True)_√ ∨ slice-kind S a = (λs. False)_√
      by(simp add:slice-kind-def Let-def)
    thus ?thesis
    proof
      assume true:slice-kind S a = (λs. True)_√
      with ‹sourcenode a = sourcenode a'› ‹targetnode a ≠ targetnode a'›
        ‹valid-edge a› ‹valid-edge a'› ‹intra-kind (kind a)› ‹intra-kind (kind a')›
      have slice-kind S a' = (λs. False)_√
        by(rule slice-kind-only-one-True-edge)
      with true show ?thesis by simp
    next
      assume false:slice-kind S a = (λs. False)_√
      from False ‹kind a' = (Q')_√› ‹sourcenode a = sourcenode a'›
      have slice-kind S a' = (λs. True)_√ ∨ slice-kind S a' = (λs. False)_√
        by(simp add:slice-kind-def Let-def)
      with false show ?thesis by auto
    qed
  qed
qed

end
```

end

1.13 The weak simulation

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

context *SDG* **begin**

lemma *call-node-notin-slice-return-node-neither*:
assumes *call-of-return-node n n'* **and** $n' \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$
shows $n \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$
proof −
from ‹*call-of-return-node n n'*› **obtain** *a a'* **where** *return-node n*
and *valid-edge a* **and** $n' =$ *sourcenode a*
and *valid-edge a'* **and** $a' \in$ *get-return-edges a*
and $n =$ *targetnode a'* **by**(*fastforce simp*:*call-of-return-node-def*)
from ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*› **obtain** *Q p r fs*
where *kind a* = *Q*:$r \hookrightarrow_p fs$ **by**(*fastforce dest*!:*only-call-get-return-edges*)
with ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*› **obtain** *Q' f'* **where** *kind a'* = $Q' \hookleftarrow_p f'$
by(*fastforce dest*!:*call-return-edges*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:$r \hookrightarrow_p fs$› ‹$a' \in$ *get-return-edges a*›
have *CFG-node* (*sourcenode a*) $s{-}p{\rightarrow}_{sum}$ *CFG-node* (*targetnode a'*)
by(*fastforce intro*:*sum-SDG-call-summary-edge*)
show *?thesis*
proof
assume $n \in \lfloor$*HRB-slice S*$\rfloor_{CFG}$
with ‹$n =$ *targetnode a'*› **have** *CFG-node* (*targetnode a'*) $\in$ *HRB-slice S*
by(*simp add*:*SDG-to-CFG-set-def*)
hence *CFG-node* (*sourcenode a*) $\in$ *HRB-slice S*
proof(*induct CFG-node* (*targetnode a'*) *rule*:*HRB-slice-cases*)
case (*phase1 nx*)
with ‹*CFG-node* (*sourcenode a*) $s{-}p{\rightarrow}_{sum}$ *CFG-node* (*targetnode a'*)›
show *?case* **by**(*fastforce intro*:*combine-SDG-slices.combSlice-refl sum-slice1*
simp:*HRB-slice-def*)
next
case (*phase2 nx n' n'' p'*)
from ‹*CFG-node* (*targetnode a'*) $\in$ *sum-SDG-slice2 n'*›
‹*CFG-node* (*sourcenode a*) $s{-}p{\rightarrow}_{sum}$ *CFG-node* (*targetnode a'*)› ‹*valid-edge a*›
have *CFG-node* (*sourcenode a*) $\in$ *sum-SDG-slice2 n'*
by(*fastforce intro*:*sum-slice2*)
with ‹$n' \in$ *sum-SDG-slice1 nx*› ‹n'' $s{-}p'{\rightarrow}_{ret}$ *CFG-node* (*parent-node n'*)› ‹$nx \in S$›
show *?case* **by**(*fastforce intro*:*combine-SDG-slices.combSlice-Return-parent-node*
simp:*HRB-slice-def*)
qed

```
    with ‹n' ∉ ⌊HRB-slice S⌋CFG› ‹n' = sourcenode a› show False
      by(simp add:SDG-to-CFG-set-def HRB-slice-def)
  qed
qed


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 −
  from assms have obs-intra (targetnode a) ⌊HRB-slice S⌋CFG ⊆
                  obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG
    by(rule edge-obs-intra-subset)
  from ‹valid-edge a› have valid-node (sourcenode a) by simp
  { fix x assume x ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG
    and obs-intra (targetnode a) ⌊HRB-slice S⌋CFG = {}
    have ∃ as. targetnode a −as→ι∗ x
    proof(cases method-exit x)
      case True
      from ‹valid-edge a› have valid-node (targetnode a) by simp
      then obtain asx where targetnode a −asx→√∗ (-Exit-)
        by(fastforce dest:Exit-path)
      then obtain as pex where targetnode a −as→ι∗ pex and method-exit pex
        by −(erule valid-Exit-path-intra-path)
      hence get-proc pex = get-proc (targetnode a)
        by −(rule intra-path-get-procs[THEN sym])
      also from ‹valid-edge a› ‹intra-kind (kind a)›
      have ... = get-proc (sourcenode a)
        by −(rule get-proc-intra[THEN sym])
      also from ‹x ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG› True
      have ... = get-proc x
        by(fastforce elim:obs-intraE intro:intra-path-get-procs)
      finally have pex = x using ‹method-exit pex› True
        by −(rule method-exit-unique)
      with ‹targetnode a −as→ι∗ pex› show ?thesis by fastforce
    next
      case False
      with ‹x ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG›
      have x postdominates (sourcenode a) by(rule obs-intra-postdominate)
      with ‹valid-edge a› ‹intra-kind (kind a)› ‹sourcenode a ∉ ⌊HRB-slice S⌋CFG›
        ‹x ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG›
      have x postdominates (targetnode a)
        by(fastforce elim:postdominate-inner-path-targetnode path-edge obs-intraE
                  simp:intra-path-def sourcenodes-def)
      thus ?thesis by(fastforce elim:postdominate-implies-inner-path)
    qed
    then obtain as where targetnode a −as→ι∗ x by blast
```

```
    from ‹x ∈ obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG›
    have x ∈ ⌊HRB-slice S⌋CFG by −(erule obs-intraE)
    have ∃x' ∈ ⌊HRB-slice S⌋CFG. ∃as'. targetnode a −as'→ι* x' ∧
      (∀a' ∈ set (sourcenodes as'). a' ∉ ⌊HRB-slice S⌋CFG)
    proof(cases ∃a' ∈ set (sourcenodes as). a' ∈ ⌊HRB-slice S⌋CFG)
      case True
      then obtain zs z zs' where sourcenodes as = zs@z#zs'
        and z ∈ ⌊HRB-slice S⌋CFG and ∀z' ∈ set zs. z' ∉ ⌊HRB-slice S⌋CFG
        by(erule split-list-first-propE)
      then obtain ys y ys'
        where sourcenodes ys = zs and as = ys@y#ys'
        and sourcenode y = z
        by(fastforce elim:map-append-append-maps simp:sourcenodes-def)
      from ‹targetnode a −as→ι* x› ‹as = ys@y#ys'›
      have targetnode a −ys@y#ys'→* x and ∀y' ∈ set ys. intra-kind (kind y')
        by(simp-all add:intra-path-def)
      from ‹targetnode a −ys@y#ys'→* x› have targetnode a −ys→* sourcenode
y
        by(rule path-split)
      with ‹∀y' ∈ set ys. intra-kind (kind y')› ‹sourcenode y = z›
        ‹∀z' ∈ set zs. z' ∉ ⌊HRB-slice S⌋CFG› ‹z ∈ ⌊HRB-slice S⌋CFG›
        ‹sourcenodes ys = zs›
      show ?thesis by(fastforce simp:intra-path-def)
    next
      case False
      with ‹targetnode a −as→ι* x› ‹x ∈ ⌊HRB-slice S⌋CFG›
      show ?thesis by fastforce
    qed
    hence ∃y. y ∈ obs-intra (targetnode a) ⌊HRB-slice S⌋CFG
      by(fastforce intro:obs-intra-elem)
    with ‹obs-intra (targetnode a) ⌊HRB-slice S⌋CFG = {}›
    have False by simp }
  with ‹obs-intra (targetnode a) ⌊HRB-slice S⌋CFG ⊆
        obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG› ‹valid-node (sourcenode a)›
  show ?thesis by(cases obs-intra (targetnode a) ⌊HRB-slice S⌋CFG = {})
                  (auto dest!:obs-intra-singleton-disj)
qed


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 −
  from ‹ms'' ∈ obs ms' ⌊HRB-slice S⌋CFG› ‹∀n ∈ set (tl ms'). return-node n›
```

obtain $msx\ m\ msx'\ mx\ m'$ **where** $ms' = msx@m\#msx'$ **and** $ms'' = mx\#msx'$
and $mx \in$ *obs-intra* $m\ \lfloor$*HRB-slice* $S\rfloor_{CFG}$
and $\forall nx \in$ *set* $msx'.\ \exists nx'.$ *call-of-return-node* $nx\ nx' \wedge nx' \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$
and *imp*:$\forall xs\ x\ xs'.\ msx = xs@x\#xs' \wedge$ *obs-intra* $x\ \lfloor$*HRB-slice* $S\rfloor_{CFG} \neq \{\}$
$\longrightarrow (\exists x'' \in$ *set* $(xs'@[m]).\ \exists mx.$ *call-of-return-node* $x''\ mx \wedge$
$mx \notin \lfloor$*HRB-slice* $S\rfloor_{CFG})$
by(*erule obsE*)
show *?thesis*
proof(*cases msx*)
case *Nil*
with ‹$\forall nx \in$ *set* $msx'.\ \exists nx'.$ *call-of-return-node* $nx\ nx' \wedge nx' \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$›
disj ‹$ms' = msx@m\#msx'$› ‹*hd* $ms =$ *sourcenode* a› ‹$ms' =$ *targetnode* $a\#tl\ ms$›
have *sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$ **by**(*cases ms*) *auto*
from ‹$ms' = msx@m\#msx'$› ‹$ms' =$ *targetnode* $a\#tl\ ms$› *Nil*
have $m =$ *targetnode* a **by** *simp*
with ‹*valid-edge* a› ‹*intra-kind* (*kind* a)› ‹*sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$›
‹$mx \in$ *obs-intra* $m\ \lfloor$*HRB-slice* $S\rfloor_{CFG}$›
have $mx \in$ *obs-intra* (*sourcenode* a) $\lfloor$*HRB-slice* $S\rfloor_{CFG}$
by(*fastforce dest*:*edge-obs-intra-subset*)
from ‹$ms' = msx@m\#msx'$› *Nil* ‹$ms' =$ *targetnode* $a\ \#\ tl\ ms$›
‹*hd* $ms =$ *sourcenode* a› ‹$ms \neq []$›
have $ms = []@$*sourcenode* $a\#msx'$ **by**(*cases ms*) *auto*
with ‹$ms'' = mx\#msx'$› ‹$mx \in$ *obs-intra* (*sourcenode* a) $\lfloor$*HRB-slice* $S\rfloor_{CFG}$›
‹$\forall nx \in$ *set* $msx'.\ \exists nx'.$ *call-of-return-node* $nx\ nx' \wedge nx' \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$› *Nil*
show *?thesis* **by**(*fastforce intro*!:*obsI*)
next
case (*Cons* $x\ xs$)
with ‹$ms' = msx@m\#msx'$› ‹$ms' =$ *targetnode* $a\ \#\ tl\ ms$›
have $msx =$ *targetnode* $a\#xs$ **by** *simp*
from *Cons* ‹$ms' = msx@m\#msx'$› ‹$ms' =$ *targetnode* $a\ \#\ tl\ ms$› ‹*hd* $ms =$ *sourcenode* a›
have $ms =$ (*sourcenode* $a\#xs$)$@m\#msx'$ **by**(*cases ms*) *auto*
from *disj* ‹$ms =$ (*sourcenode* $a\#xs$)$@m\#msx'$›
‹$\forall nx \in$ *set* $msx'.\ \exists nx'.$ *call-of-return-node* $nx\ nx' \wedge nx' \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$›
have *disj2*:$(\exists m \in$ *set* $(xs@[m]).\ \exists m'.$ *call-of-return-node* $m\ m' \wedge$
$m' \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}) \vee$ *hd* $ms \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$
by *fastforce*
hence $\forall zs\ z\ zs'.$ *sourcenode* $a\#xs = zs@z\#zs' \wedge$ *obs-intra* $z\ \lfloor$*HRB-slice* $S\rfloor_{CFG} \neq \{\}$
$\longrightarrow (\exists z'' \in$ *set* $(zs'@[m]).\ \exists mx.$ *call-of-return-node* $z''\ mx \wedge$
$mx \notin \lfloor$*HRB-slice* $S\rfloor_{CFG})$
proof(*cases hd ms* $\notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$)
case *True*
with ‹*hd* $ms =$ *sourcenode* a› **have** *sourcenode* $a \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$ **by** *simp*

with ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *obs-intra* (*targetnode a*) ⌊*HRB-slice S*⌋$_{CFG}$ =
obs-intra (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$
by(*rule edge-obs-intra-slice-eq*)
with *imp* ‹*msx* = *targetnode a*#*xs*› **show** *?thesis*
by *auto*(*case-tac zs*,*fastforce*,*erule-tac x*=*targetnode a*#*list* **in** *allE*,*fastforce*)
next
case *False*
with ‹*hd ms* = *sourcenode a*› ‹*valid-edge a*›
have *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {*sourcenode a*}
by(*fastforce intro*!:*n-in-obs-intra*)
from *False disj2*
have ∃ *m* ∈ *set* (*xs*@[*m*]). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
by *simp*
with *imp* ‹*obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {*sourcenode a*}›
‹*msx* = *targetnode a*#*xs*› **show** *?thesis*
by *auto*(*case-tac zs*,*fastforce*,*erule-tac x*=*targetnode a*#*list* **in** *allE*,*fastforce*)
qed
with ‹*ms'* = *msx*@*m*#*msx'*› ‹*ms'* = *targetnode a* # *tl ms*› ‹*hd ms* = *sourcenode a*›
‹*ms''* = *mx*#*msx'*› ‹*mx* ∈ *obs-intra m* ⌊*HRB-slice S*⌋$_{CFG}$›
‹∀ *nx* ∈ *set msx'*. ∃ *nx'*. *call-of-return-node nx nx'* ∧ *nx'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*ms* = (*sourcenode a*#*xs*)@*m*#*msx'*›
show *?thesis* **by**(*simp del*:*obs.simps*)(*rule obsI*,*auto*)
qed
qed

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→_τ (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→_τ (ms',s')


lemma silent-move-valid-nodes:
  ⟦S,f ⊢ (ms,s) −a→_τ (ms',s'); ∀ m ∈ set ms'. valid-node m⟧
  ⟹ ∀ m ∈ set ms. valid-node m
by(induct rule:silent-move.induct)(case-tac ms,auto)+


lemma silent-move-return-node:
  S,f ⊢ (ms,s) −a→_τ (ms',s') ⟹ ∀ m ∈ set (tl ms'). return-node m
proof(induct rule:silent-move.induct)
  case (silent-move-intra f a s s' ms n_c ms')
  thus ?case by simp
next
  case (silent-move-call f a s s' Q r p fs a' ms n_c ms')
  from ‹valid-edge a'› ‹valid-edge a› ‹a' ∈ get-return-edges a›
  have return-node (targetnode a') by(fastforce simp:return-node-def)
  with ‹∀ m∈set (tl ms). return-node m› ‹ms' = targetnode a # targetnode a' # tl
ms›
  show ?case by simp
next
  case (silent-move-return f a s s' Q p f' ms n_c ms')
  thus ?case by(cases tl ms) auto
qed


lemma silent-move-equal-length:
  assumes S,f ⊢ (ms,s) −a→_τ (ms',s')
  shows length ms = length s and length ms' = length s'
proof −
  from ‹S,f ⊢ (ms,s) −a→_τ (ms',s')›
  have length ms = length s ∧ length ms' = length s'
  proof(induct rule:silent-move.induct)
    case (silent-move-intra f a s s' ms n_c ms')
    from ‹pred (f a) s› obtain cf cfs where [simp]:s = cf#cfs by(cases s) auto
```

from ‹*length ms = length s*› ‹*ms' = targetnode a # tl ms*›
‹*length s' = length s*› **show** *?case* **by** *simp*
next
case (*silent-move-call f a s s' Q r p fs a' ms* n_c *ms'*)
from ‹*pred* (*f a*) *s*› **obtain** *cf cfs* **where** [*simp*]:*s = cf#cfs* **by**(*cases s*) *auto*
from ‹*length ms = length s*› ‹*length s' = Suc* (*length s*)›
‹*ms' = targetnode a # targetnode a' # tl ms*› **show** *?case* **by** *simp*
next
case (*silent-move-return f a s s' Q p f' ms* n_c *ms'*)
from ‹*length ms = length s*› ‹*length s = Suc* (*length s'*)› ‹*ms' = tl ms*› ‹*s' ≠ []*›
show *?case* **by** *simp*
qed
thus *length ms = length s* **and** *length ms' = length s'* **by** *simp-all*
qed

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(*induct S f*≡*kind ms s a ms' s' rule*:*silent-move.induct*)
case (*silent-move-intra a s s' ms* n_c *ms'*)
from ‹*pred* (*kind a*) *s*› ‹*length ms = length s*› **have** *ms ≠ []*
by(*cases s*) *auto*
with *silent-move-intra* **show** *?case* **by** −(*rule intra-edge-obs-slice*)
next
case (*silent-move-call a s s' Q r p fs a' ms S ms'*)
note *disj* = ‹(∃ *m*∈*set* (*tl ms*). ∃ *m'*. *call-of-return-node m m'* ∧
m' ∉ ⌊*HRB-slice S*⌋$_{CFG}$) ∨ *hd ms* ∉ ⌊*HRB-slice S*⌋$_{CFG}$›
from ‹*valid-edge a'*› ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
have *return-node* (*targetnode a'*) **by**(*fastforce simp*:*return-node-def*)
with ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› ‹*valid-edge a'*›
have *call-of-return-node* (*targetnode a'*) (*sourcenode a*)
by(*simp add*:*call-of-return-node-def*) *blast*
from ‹*pred* (*kind a*) *s*› ‹*length ms = length s*›
have *ms ≠ []* **by**(*cases s*) *auto*
from *disj*
show *?case*
proof
assume *hd ms* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
with ‹*hd ms = sourcenode a*› **have** *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **by** *simp*
with ‹*call-of-return-node* (*targetnode a'*) (*sourcenode a*)›
‹*ms' = targetnode a # targetnode a' # tl ms*›
have ∃ *n'* ∈ *set* (*tl ms'*). ∃ *nx*. *call-of-return-node n' nx* ∧ *nx* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
by *fastforce*
with ‹*msx* ∈ *obs ms'* ⌊*HRB-slice S*⌋$_{CFG}$› ‹*ms' = targetnode a # targetnode a' # tl ms*›
have *msx* ∈ *obs* (*targetnode a' # tl ms*) ⌊*HRB-slice S*⌋$_{CFG}$ **by** *simp*

from ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*›
obtain a'' **where** *valid-edge* a'' **and** [*simp*]:*sourcenode* $a'' =$ *sourcenode a*
and [*simp*]:*targetnode* $a'' =$ *targetnode* a' **and** *intra-kind*(*kind* a'')
by −(*drule call-return-node-edge*,*auto simp*:*intra-kind-def*)
from ‹$\forall$ *m*∈*set* (*tl ms'*). *return-node m*› ‹$ms' =$ *targetnode a* # *targetnode* a' # *tl ms*›
have $\forall$ *m*∈*set* (*tl ms*). *return-node m* **by** *simp*
with ‹$ms \neq []$› ‹*msx* ∈ *obs* (*targetnode* a'#*tl ms*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
‹*valid-edge* a''› ‹*intra-kind*(*kind* a'')› *disj*
‹*hd ms* = *sourcenode a*›
show *?case* **by** −(*rule intra-edge-obs-slice*,*fastforce*+)
next
assume $\exists$ *m*∈*set* (*tl ms*).
$\exists m'$. *call-of-return-node m* $m' \wedge m' \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$
with ‹$ms \neq []$› ‹*msx* ∈ *obs ms'* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
‹$ms' =$ *targetnode a* # *targetnode* a' # *tl ms*›
show *?thesis* **by**(*cases ms*) *auto*
qed
next
case (*silent-move-return a s s' Q p f' ms S ms'*)
from ‹*length ms* = *length s*› ‹*length s* = *Suc* (*length s'*)› ‹$s' \neq []$›
have $ms \neq []$ **and** *tl ms* ≠ [] **by**(*auto simp*:*length-Suc-conv*)
from ‹$\exists$ *m*∈*set* (*tl ms*).
$\exists m'$. *call-of-return-node m* $m' \wedge m' \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
‹*tl ms* ≠ []› ‹*hd* (*tl ms*) = *targetnode a*›
have ($\exists m'$. *call-of-return-node* (*targetnode a*) $m' \wedge m' \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$) ∨
($\exists$ *m*∈*set* (*tl* (*tl ms*)). $\exists m'$. *call-of-return-node m* $m' \wedge m' \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$)
by(*cases tl ms*) *auto*
hence *obs ms* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = *obs* (*tl ms*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
proof
assume $\exists m'$. *call-of-return-node* (*targetnode a*) $m' \wedge m' \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$
from ‹*tl ms* ≠ []› **have** *hd* (*tl ms*) ∈ *set* (*tl ms*) **by** *simp*
with ‹*hd* (*tl ms*) = *targetnode a*› **have** *targetnode a* ∈ *set* (*tl ms*) **by** *simp*
with ‹$ms \neq []$›
‹$\exists m'$. *call-of-return-node* (*targetnode a*) $m' \wedge m' \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
have $\exists$ *m*∈*set* (*tl ms*). $\exists m'$. *call-of-return-node m* $m' \wedge$
$m' \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by**(*cases ms*) *auto*
with ‹$ms \neq []$› **show** *?thesis* **by**(*cases ms*) *auto*
next
assume $\exists$ *m*∈*set* (*tl* (*tl ms*)). $\exists m'$. *call-of-return-node m* $m' \wedge$
$m' \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$
with ‹$ms \neq []$› ‹*tl ms* ≠ []› **show** *?thesis*
by(*cases ms*,*auto simp*:*Let-def*)(*case-tac list*,*auto*)+
qed
with ‹$ms' =$ *tl ms*› ‹*msx* ∈ *obs ms'* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› **show** *?case* **by** *simp*
qed

lemma *silent-move-empty-obs-slice*:
assumes $S,f \vdash (ms,s) -a\rightarrow_\tau (ms',s')$ **and** *obs ms'* $\lfloor$*HRB-slice S*$\rfloor_{CFG} = \{\}$
shows *obs ms* $\lfloor$*HRB-slice S*$\rfloor_{CFG} = \{\}$
proof(*rule ccontr*)
assume *obs ms* $\lfloor$*HRB-slice S*$\rfloor_{CFG} \neq \{\}$
then obtain *xs* **where** *xs* $\in$ *obs ms* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by** *fastforce*
from ‹$S,f \vdash (ms,s) -a\rightarrow_\tau (ms',s')$› **have** $\forall\, m \in set\ (tl\ ms).\ return\text{-}node\ m$
by(*fastforce elim*!:*silent-move.cases simp*:*call-of-return-node-def*)
with ‹*xs* $\in$ *obs ms* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
obtain *msx m msx' m'* **where** *assms*:*ms* = *msx*@*m*#*msx' xs* = *m'*#*msx'*
m' $\in$ *obs-intra m* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
$\forall\, mx \in set\ msx'.\ \exists\, mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \in \lfloor$*HRB-slice S*$\rfloor_{CFG}$
$\forall\, xs\ x\ xs'.\ msx = xs@x\#xs' \wedge obs\text{-}intra\ x\ \lfloor$*HRB-slice S*$\rfloor_{CFG} \neq \{\}$
$\longrightarrow (\exists\, x'' \in set\ (xs'@[m]).\ \exists\, mx.\ call\text{-}of\text{-}return\text{-}node\ x''\ mx \wedge$
$mx \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$)
by(*erule obsE*)
from ‹$S,f \vdash (ms,s) -a\rightarrow_\tau (ms',s')$› ‹*obs ms'* $\lfloor$*HRB-slice S*$\rfloor_{CFG} = \{\}$› *assms*
show *False*
proof(*induct rule*:*silent-move.induct*)
case (*silent-move-intra f a s s' ms S ms'*)
note *disj* = ‹($\exists\, m{\in}set\ (tl\ ms).\ \exists\, m'.\ call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge$
m' $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$) $\vee$ *hd ms* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
note *msx* = ‹$\forall\, xs\ x\ xs'.\ msx = xs@x\#xs' \wedge obs\text{-}intra\ x\ \lfloor$*HRB-slice S*$\rfloor_{CFG} \neq \{\} \longrightarrow$
($\exists\, x''{\in}set\ (xs'\ @\ [m]).\ \exists\, mx.\ call\text{-}of\text{-}return\text{-}node\ x''\ mx \wedge mx \notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$)›
note *msx'* = ‹$\forall\, mx{\in}set\ msx'.\ \exists\, mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge$
mx' $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
show *False*
proof(*cases msx*)
case *Nil*
with ‹*ms* = *msx* @ *m* # *msx'*› ‹*hd ms* = *sourcenode a*› **have** [*simp*]:*m* = *sourcenode a*
and *tl ms* = *msx'* **by** *simp-all*
from *Nil* ‹*ms'* = *targetnode a* # *tl ms*› ‹*ms* = *msx* @ *m* # *msx'*›
have *ms'* = *msx* @ *targetnode a* # *msx'* **by** *simp*
from *msx' disj* ‹*tl ms* = *msx'*› ‹*hd ms* = *sourcenode a*›
have *sourcenode a* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by** *fastforce*
with ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *obs-intra* (*targetnode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ =
obs-intra (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by**(*rule edge-obs-intra-slice-eq*)
with ‹*m'* $\in$ *obs-intra m* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
have *m'* $\in$ *obs-intra* (*targetnode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by** *simp*
from *msx Nil* **have** $\forall\, xs\ x\ xs'.\ msx = xs@x\#xs' \wedge$
obs-intra x $\lfloor$*HRB-slice S*$\rfloor_{CFG} \neq \{\} \longrightarrow$
($\exists\, x''{\in}set\ (xs'\ @\ [targetnode\ a]).\ \exists\, mx.\ call\text{-}of\text{-}return\text{-}node\ x''\ mx \wedge$
mx $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$) **by** *simp*
with ‹*m'* $\in$ *obs-intra* (*targetnode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› *msx'*

```
      ‹ms' = msx @ targetnode a # msx'›
    have m'#msx' ∈ obs ms' ⌊HRB-slice S⌋CFG by(rule obsI)
    with ‹obs ms' ⌊HRB-slice S⌋CFG = {}› show False by simp
  next
    case (Cons y ys)
     with ‹ms = msx @ m # msx'› ‹ms' = targetnode a # tl ms› ‹hd ms =
sourcenode a›
    have ms' = targetnode a # ys @ m # msx' and y = sourcenode a
     and tl ms = ys @ m # msx' by simp-all
    { fix x assume x ∈ obs-intra (targetnode a) ⌊HRB-slice S⌋CFG
     have obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG ≠ {}
     proof(cases sourcenode a ∈ ⌊HRB-slice S⌋CFG)
       case True
       from ‹valid-edge a› have valid-node (sourcenode a) by simp
       from this True
       have obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG = {sourcenode a}
        by(rule n-in-obs-intra)
       thus ?thesis by simp
     next
       case False
       from ‹valid-edge a› ‹intra-kind (kind a)› False
       have obs-intra (targetnode a) ⌊HRB-slice S⌋CFG =
         obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG
         by(rule edge-obs-intra-slice-eq)
       with ‹x ∈ obs-intra (targetnode a) ⌊HRB-slice S⌋CFG› show ?thesis
         by fastforce
     qed }
    with msx Cons ‹y = sourcenode a›
    have ∀ xs x xs'. targetnode a # ys = xs@x#xs' ∧
     obs-intra x ⌊HRB-slice S⌋CFG ≠ {} ⟶ (∃ x''∈set (xs' @ [m]).
     ∃ mx. call-of-return-node x'' mx ∧ mx ∉ ⌊HRB-slice S⌋CFG)
     apply clarsimp apply(case-tac xs) apply auto
     apply(erule-tac x=[] in allE) apply clarsimp
     apply(erule-tac x=sourcenode a # list in allE) apply auto
     done
    with ‹m' ∈ obs-intra m ⌊HRB-slice S⌋CFG› msx'
     ‹ms' = targetnode a # ys @ m # msx'›
    have m'#msx' ∈ obs ms' ⌊HRB-slice S⌋CFG by −(rule obsI,auto)
    with ‹obs ms' ⌊HRB-slice S⌋CFG = {}› show False by simp
  qed
 next
  case (silent-move-call f a s s' Q r p fs a' ms S ms')
  note disj = ‹(∃ m∈set (tl ms). ∃ m'. call-of-return-node m m' ∧
    m' ∉ ⌊HRB-slice S⌋CFG) ∨ hd ms ∉ ⌊HRB-slice S⌋CFG›
  note msx = ‹∀ xs x xs'. msx = xs@x#xs' ∧ obs-intra x ⌊HRB-slice S⌋CFG ≠
{} ⟶
    (∃ x''∈set (xs' @ [m]). ∃ mx. call-of-return-node x'' mx ∧ mx ∉ ⌊HRB-slice
S⌋CFG)›
  note msx' = ‹∀ mx∈set msx'. ∃ mx'. call-of-return-node mx mx' ∧
```

$mx' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}\rangle$
from ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*› **obtain** a'' **where** *valid-edge* a''
and *sourcenode* $a'' =$ *sourcenode a* **and** *targetnode* $a'' =$ *targetnode* a'
and *intra-kind* (*kind* a'')
by(*fastforce dest*:*call-return-node-edge simp*:*intra-kind-def*)
from ‹*valid-edge* a'› ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*›
have *call-of-return-node* (*targetnode* a') (*sourcenode a*)
by(*fastforce simp*:*call-of-return-node-def return-node-def*)
show *False*
proof(*cases msx*)
case *Nil*
with ‹$ms = msx$ @ $m \# msx'$› ‹*hd ms* = *sourcenode a*› **have** [*simp*]:$m =$ *sourcenode a*
and *tl ms* = msx' **by** *simp-all*
from *Nil* ‹$ms' =$ *targetnode a* # *targetnode* $a' \#$ *tl ms*› ‹$ms = msx$ @ $m \# msx'$›
have $ms' =$ *targetnode a* # *targetnode* $a' \# msx'$ **by** *simp*
from msx' *disj* ‹*tl ms* = msx'› ‹*hd ms* = *sourcenode a*›
have *sourcenode a* $\notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **by** *fastforce*
from ‹*valid-edge* a''› ‹*intra-kind* (*kind* a'')› ‹*sourcenode a* $\notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
‹*sourcenode* $a'' =$ *sourcenode a*› ‹*targetnode* $a'' =$ *targetnode* a'›
have *obs-intra* (*targetnode* a') $\lfloor HRB\text{-}slice\ S \rfloor_{CFG} =$
obs-intra (*sourcenode a*) $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*fastforce dest*:*edge-obs-intra-slice-eq*)
with ‹$m' \in$ *obs-intra m* $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
have $m' \in$ *obs-intra* (*targetnode* a') $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **by** *simp*
from *this* msx' **have** $m'\#msx' \in$ *obs* (*targetnode* $a'\#msx'$) $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*fastforce intro*:*obsI*)
from ‹*call-of-return-node* (*targetnode* a') (*sourcenode a*)›
‹*sourcenode a* $\notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
have $\exists\, m' \in$ *set* (*targetnode* $a'\#msx'$).
$\exists\, mx.$ *call-of-return-node* m' $mx \wedge mx \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by *fastforce*
with ‹$m'\#msx' \in$ *obs* (*targetnode* $a'\#msx'$) $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
have $m'\#msx' \in$ *obs* (*targetnode a*#*targetnode* $a'\#msx'$) $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by *simp*
with ‹$ms' =$ *targetnode a* # *targetnode* $a' \#$ msx'› ‹*obs* ms' $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ $= \{\}$›
show *False* **by** *simp*
next
case (*Cons y ys*)
with ‹$ms = msx$ @ $m \# msx'$› ‹$ms' =$ *targetnode a* # *targetnode* $a' \#$ *tl ms*›
‹*hd ms* = *sourcenode a*›
have $ms' =$ *targetnode a* # *targetnode* $a' \#$ *ys* @ $m \# msx'$
and $y =$ *sourcenode a* **and** *tl ms* = *ys* @ $m \# msx'$ **by** *simp-all*
show *False*
proof(*cases obs-intra* (*targetnode a*) $\lfloor HRB\text{-}slice\ S \rfloor_{CFG} \neq \{\} \longrightarrow$
($\exists\, x'' \in$*set* (*targetnode* $a' \#$ *ys* @ [m]).
$\exists\, mx.$ *call-of-return-node* x'' $mx \wedge mx \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$))

case *True*
hence *imp*:*obs-intra* (*targetnode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG} \neq \{\} \longrightarrow$
($\exists x''{\in}$*set* (*targetnode a'* # *ys* @ [*m*]).
$\exists$ *mx*. *call-of-return-node x'' mx* $\wedge$ *mx* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$) .
show *False*
proof(*cases obs-intra* (*targetnode a'*) $\lfloor$*HRB-slice S*$\rfloor_{CFG} \neq \{\} \longrightarrow$
($\exists x''{\in}$*set* (*ys* @ [*m*]). $\exists$ *mx*. *call-of-return-node x'' mx* $\wedge$
mx $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$))
case *True*
with *imp msx Cons* ‹*y* = *sourcenode a*›
have $\forall$ *xs x xs'*. *targetnode a* # *targetnode a'* # *ys* = *xs*@*x*#*xs'* $\wedge$
obs-intra x $\lfloor$*HRB-slice S*$\rfloor_{CFG} \neq \{\} \longrightarrow$ ($\exists x''{\in}$*set* (*xs'* @ [*m*]).
$\exists$ *mx*. *call-of-return-node x'' mx* $\wedge$ *mx* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$)
apply *clarsimp* **apply**(*case-tac xs*) **apply** *fastforce*
apply(*case-tac list*) **apply** *fastforce* **apply** *clarsimp*
apply(*erule-tac x*=*sourcenode a* # *lista* **in** *allE*) **apply** *auto*
done
with ‹*m'* $\in$ *obs-intra m* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› *msx'*
‹*ms'* = *targetnode a* # *targetnode a'* # *ys* @ *m* # *msx'*›
have *m'*#*msx'* $\in$ *obs ms'* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by** $-$(*rule obsI*,*auto*)
with ‹*obs ms'* $\lfloor$*HRB-slice S*$\rfloor_{CFG} = \{\}$› **show** *False* **by** *simp*
next
case *False*
hence *obs-intra* (*targetnode a'*) $\lfloor$*HRB-slice S*$\rfloor_{CFG} \neq \{\}$
and *all*:$\forall x''{\in}$*set* (*ys* @ [*m*]). $\forall$ *mx*. *call-of-return-node x'' mx* $\longrightarrow$
mx $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$
by *fastforce*+
have *obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG} \neq \{\}$
proof(*cases sourcenode a* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$)
case *True*
from ‹*valid-edge a*› **have** *valid-node* (*sourcenode a*) **by** *simp*
from *this True*
have *obs-intra* (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG} = \{$*sourcenode a*$\}$
by(*rule n-in-obs-intra*)
thus *?thesis* **by** *simp*
next
case *False*
with ‹*sourcenode a''* = *sourcenode a*›
have *sourcenode a''* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by** *simp*
with ‹*valid-edge a''*› ‹*intra-kind* (*kind a''*)›
have *obs-intra* (*targetnode a''*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ =
obs-intra (*sourcenode a''*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
by(*rule edge-obs-intra-slice-eq*)
with ‹*obs-intra* (*targetnode a'*) $\lfloor$*HRB-slice S*$\rfloor_{CFG} \neq \{\}$›
‹*sourcenode a''* = *sourcenode a*› ‹*targetnode a''* = *targetnode a'*›
show *?thesis* **by** *fastforce*
qed
with *msx Cons* ‹*y* = *sourcenode a*› *all*
show *False* **by** *simp blast*

```
      qed
    next
      case False
      hence obs-intra (targetnode a) ⌊HRB-slice S⌋CFG ≠ {}
        and all:∀ x''∈set (targetnode a' # ys @ [m]).
        ∀ mx. call-of-return-node x'' mx ⟶ mx ∈ ⌊HRB-slice S⌋CFG
        by fastforce+
      with Cons ‹y = sourcenode a› msx
      have obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG = {} by auto blast
      from ‹call-of-return-node (targetnode a') (sourcenode a)› all
      have sourcenode a ∈ ⌊HRB-slice S⌋CFG by fastforce
      from ‹valid-edge a› have valid-node (sourcenode a) by simp
      from this ‹sourcenode a ∈ ⌊HRB-slice S⌋CFG›
      have obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG = {sourcenode a}
        by(rule n-in-obs-intra)
       with ‹obs-intra (sourcenode a) ⌊HRB-slice S⌋CFG = {}› show False by
simp
    qed
  qed
 next
  case (silent-move-return f a s s' Q p f' ms S ms')
  note msx = ‹∀ xs x xs'. msx = xs@x#xs' ∧ obs-intra x ⌊HRB-slice S⌋CFG ≠
{} ⟶
    (∃ x''∈set (xs' @ [m]). ∃ mx. call-of-return-node x'' mx ∧ mx ∉ ⌊HRB-slice
S⌋CFG)›
  note msx' = ‹∀ mx∈set msx'. ∃ mx'. call-of-return-node mx mx' ∧
    mx' ∈ ⌊HRB-slice S⌋CFG›
  show False
  proof(cases msx)
    case Nil
    with ‹ms = msx @ m # msx'› ‹hd ms = sourcenode a› have  tl ms = msx'
by simp
     with ‹∃ m∈set (tl ms). ∃ m'. call-of-return-node m m' ∧ m' ∉ ⌊HRB-slice
S⌋CFG›
      msx'
    show False by fastforce
  next
    case (Cons y ys)
    with ‹ms = msx @ m # msx'› ‹hd ms = sourcenode a› ‹ms' = tl ms›
    have ms' = ys @ m # msx' and y = sourcenode a by simp-all
    from msx Cons have ∀ xs x xs'. ys = xs@x#xs' ∧
      obs-intra x ⌊HRB-slice S⌋CFG ≠ {} ⟶  (∃ x''∈set (xs' @ [m]).
      ∃ mx. call-of-return-node x'' mx ∧ mx ∉ ⌊HRB-slice S⌋CFG)
      by auto (erule-tac x=y # xs in allE,auto)
    with ‹m' ∈ obs-intra m ⌊HRB-slice S⌋CFG› msx' ‹ms' = ys @ m # msx'›
    have m'#msx' ∈ obs ms' ⌊HRB-slice S⌋CFG by(rule obsI)
    with ‹obs ms' ⌊HRB-slice S⌋CFG = {}› show False by simp
  qed
qed
```

qed

inductive *silent-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*
(-,- ⊢ '(-,-') =-⇒$_\tau$ '(-,-') [*51*,*50*,*0*,*0*,*50*,*0*,*0*] *51*)

where *silent-moves-Nil*: *length ms* = *length s* ⟹ *S*,*f* ⊢ (*ms*,*s*) =[]⇒$_\tau$ (*ms*,*s*)

| *silent-moves-Cons*:
⟦*S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*); *S*,*f* ⊢ (*ms'*,*s'*) =*as*⇒$_\tau$ (*ms''*,*s''*)⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) =*a*#*as*⇒$_\tau$ (*ms''*,*s''*)

lemma *silent-moves-equal-length*:
assumes *S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms'*,*s'*)
shows *length ms* = *length s* **and** *length ms'* = *length s'*
proof −
from ‹*S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms'*,*s'*)›
have *length ms* = *length s* ∧ *length ms'* = *length s'*
proof(*induct rule*:*silent-moves.induct*)
case (*silent-moves-Cons S f ms s a ms' s' as ms'' s''*)
from ‹*S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*)›
have *length ms* = *length s* **and** *length ms'* = *length s'*
by(*rule silent-move-equal-length*)+
with ‹*length ms'* = *length s'* ∧ *length ms''* = *length s''*›
show *?case* **by** *simp*
qed *simp*
thus *length ms* = *length s length ms'* = *length s'* **by** *simp-all*
qed

lemma *silent-moves-Append*:
⟦*S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms''*,*s''*); *S*,*f* ⊢ (*ms''*,*s''*) =*as'*⇒$_\tau$ (*ms'*,*s'*)⟧
⟹ *S*,*f* ⊢ (*ms*,*s*) =*as*@*as'*⇒$_\tau$ (*ms'*,*s'*)
by(*induct rule*:*silent-moves.induct*)(*auto intro*:*silent-moves.intros*)

lemma *silent-moves-split*:
assumes *S*,*f* ⊢ (*ms*,*s*) =*as*@*as'*⇒$_\tau$ (*ms'*,*s'*)
obtains *ms''* *s''* **where** *S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms''*,*s''*)
and *S*,*f* ⊢ (*ms''*,*s''*) =*as'*⇒$_\tau$ (*ms'*,*s'*)
proof(*atomize-elim*)
from ‹*S*,*f* ⊢ (*ms*,*s*) =*as*@*as'*⇒$_\tau$ (*ms'*,*s'*)›
show ∃ *ms''* *s''*. *S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms''*,*s''*) ∧ *S*,*f* ⊢ (*ms''*,*s''*) =*as'*⇒$_\tau$

```
(ms',s')
  proof(induct as arbitrary:ms s)
    case Nil
    from ‹S,f ⊢ (ms,s) =[] @ as'⇒τ (ms',s')› have length ms = length s
      by(fastforce intro:silent-moves-equal-length)
    hence S,f ⊢ (ms,s) =[]⇒τ (ms,s) by(rule silent-moves-Nil)
    with ‹S,f ⊢ (ms,s) =[] @ as'⇒τ (ms',s')› show ?case by fastforce
  next
    case (Cons ax asx)
    note IH = ‹⋀ms s. S,f ⊢ (ms,s) =asx @ as'⇒τ (ms',s') ⟹
     ∃ms'' s''. S,f ⊢ (ms,s) =asx⇒τ (ms'',s'') ∧ S,f ⊢ (ms'',s'') =as'⇒τ (ms',s')›
    from ‹S,f ⊢ (ms,s) =(ax # asx) @ as'⇒τ (ms',s')›
    obtain msx sx where S,f ⊢ (ms,s) −ax→τ (msx,sx)
      and S,f ⊢ (msx,sx) =asx @ as'⇒τ (ms',s')
      by(auto elim:silent-moves.cases)
   from IH[OF this(2)] obtain ms'' s'' where S,f ⊢ (msx,sx) =asx⇒τ (ms'',s'')
      and S,f ⊢ (ms'',s'') =as'⇒τ (ms',s') by blast
    from ‹S,f ⊢ (ms,s) −ax→τ (msx,sx)› ‹S,f ⊢ (msx,sx) =asx⇒τ (ms'',s'')›
    have S,f ⊢ (ms,s) =ax#asx⇒τ (ms'',s'') by(rule silent-moves-Cons)
    with ‹S,f ⊢ (ms'',s'') =as'⇒τ (ms',s')› show ?case by blast
  qed
qed


lemma valid-nodes-silent-moves:
  ⟦S,f⊢ (ms,s) =as'⇒τ (ms',s'); ∀m ∈ set ms. valid-node m⟧
  ⟹ ∀m ∈ set ms'. valid-node m
proof(induct rule:silent-moves.induct)
  case (silent-moves-Cons S f ms s a ms' s' as ms'' s'')
  note IH = ‹∀m∈set ms'. valid-node m ⟹ ∀m∈set ms''. valid-node m›
  from ‹S,f ⊢ (ms,s) −a→τ (ms',s')› ‹∀m∈set ms. valid-node m›
  have ∀m∈set ms'. valid-node m
    apply − apply(erule silent-move.cases) apply auto
    by(cases ms,auto dest:get-return-edges-valid)+
  from IH[OF this] show ?case .
qed simp


lemma return-nodes-silent-moves:
  ⟦S,f ⊢ (ms,s) =as'⇒τ (ms',s'); ∀m ∈ set (tl ms). return-node m⟧
  ⟹ ∀m ∈ set (tl ms'). return-node m
by(induct rule:silent-moves.induct,auto dest:silent-move-return-node)


lemma silent-moves-intra-path:
  ⟦S,f ⊢ (m#ms,s) =as⇒τ (m'#ms',s'); ∀a ∈ set as. intra-kind(kind a)⟧
  ⟹ ms = ms' ∧ get-proc m = get-proc m'
proof(induct S f m#ms s as m'#ms' s' arbitrary:m
  rule:silent-moves.induct)
```

case (*silent-moves-Cons S f sx a msx' sx' as s''*)
thus *?case*
proof(*induct - - m # ms - - - - rule:silent-move.induct*)
case (*silent-move-intra f a s s' n$_c$ msx'*)
note *IH* = ‹$\bigwedge$*m*. ⟦*msx'* = *m* # *ms*; ∀ *a*∈*set as*. *intra-kind* (*kind a*)⟧
⟹ *ms* = *ms'* ∧ *get-proc m* = *get-proc m'*›
from ‹*msx'* = *targetnode a* # *tl* (*m* # *ms*)›
have *msx'* = *targetnode a* # *ms* **by** *simp*
from ‹∀ *a*∈*set* (*a* # *as*). *intra-kind* (*kind a*)› **have** ∀ *a*∈*set as*. *intra-kind* (*kind a*)
by *simp*
from *IH*[*OF* ‹*msx'* = *targetnode a* # *ms*› *this*]
have *ms* = *ms'* **and** *get-proc* (*targetnode a*) = *get-proc m'* **by** *simp-all*
moreover
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
moreover
from ‹*hd* (*m* # *ms*) = *sourcenode a*› **have** *m* = *sourcenode a* **by** *simp*
ultimately show *?case* **using** ‹*ms* = *ms'*› **by** *simp*
qed (*auto simp:intra-kind-def*)
qed *simp*

lemma *silent-moves-nodestack-notempty*:
⟦*S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms'*,*s'*); *ms* ≠ []⟧ ⟹ *ms'* ≠ []
apply(*induct S f ms s as ms' s' rule:silent-moves.induct*) **apply** *auto*
apply(*erule silent-move.cases*) **apply** *auto*
apply(*case-tac tl msa*) **by** *auto*

lemma *silent-moves-obs-slice*:
⟦*S*,*kind* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms'*,*s'*); *mx* ∈ *obs ms'* ⌊*HRB-slice S*⌋$_{CFG}$;
∀ *n* ∈ *set* (*tl ms'*). *return-node n*⟧
⟹ *mx* ∈ *obs ms* ⌊*HRB-slice S*⌋$_{CFG}$ ∧ (∀ *n* ∈ *set* (*tl ms*). *return-node n*)
proof(*induct S f*≡*kind ms s as ms' s' rule:silent-moves.induct*)
case *silent-moves-Nil* **thus** *?case* **by** *simp*
next
case (*silent-moves-Cons S ms s a ms' s' as ms'' s''*)
note *IH* = ‹⟦*mx* ∈ *obs ms''* ⌊*HRB-slice S*⌋$_{CFG}$; ∀ *m*∈*set* (*tl ms''*). *return-node m*⟧
⟹ *mx* ∈ *obs ms'* ⌊*HRB-slice S*⌋$_{CFG}$ ∧ (∀ *m*∈*set* (*tl ms'*). *return-node m*)›
from *IH*[*OF* ‹*mx* ∈ *obs ms''* ⌊*HRB-slice S*⌋$_{CFG}$› ‹∀ *m*∈*set* (*tl ms''*). *return-node m*›]
have *mx* ∈ *obs ms'* ⌊*HRB-slice S*⌋$_{CFG}$ **and** ∀ *m*∈*set* (*tl ms'*). *return-node m*
by *simp-all*
with ‹*S*,*kind* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*)›
have *mx* ∈ *obs ms* ⌊*HRB-slice S*⌋$_{CFG}$ **by**(*fastforce intro:silent-move-obs-slice*)
moreover
from ‹*S*,*kind* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*)› **have** ∀ *m*∈*set* (*tl ms*). *return-node m*

by(*fastforce elim*:*silent-move.cases*)
ultimately show *?case* **by** *simp*
qed

lemma *silent-moves-empty-obs-slice*:
⟦*S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms'*,*s'*); *obs ms'* ⌊*HRB-slice S*⌋$_{CFG}$ = {}⟧
⟹ *obs ms* ⌊*HRB-slice S*⌋$_{CFG}$ = {}
proof(*induct rule*:*silent-moves.induct*)
case *silent-moves-Nil* **thus** *?case* **by** *simp*
next
case (*silent-moves-Cons S f ms s a ms' s' as ms'' s''*)
note *IH* = ‹*obs ms''* ⌊*HRB-slice S*⌋$_{CFG}$ = {} ⟹ *obs ms'* ⌊*HRB-slice S*⌋$_{CFG}$
= {}›
from *IH*[*OF* ‹*obs ms''* ⌊*HRB-slice S*⌋$_{CFG}$ = {}›]
have *obs ms'* ⌊*HRB-slice S*⌋$_{CFG}$ = {} **by** *simp*
with ‹*S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*)›
show *?case* **by** −(*rule silent-move-empty-obs-slice*,*fastforce*)
qed

lemma *silent-moves-preds-transfers*:
assumes *S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms'*,*s'*)
shows *preds* (*map f as*) *s* **and** *transfers* (*map f as*) *s* = *s'*
proof −
from ‹*S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms'*,*s'*)›
have *preds* (*map f as*) *s* ∧ *transfers* (*map f as*) *s* = *s'*
proof(*induct rule*:*silent-moves.induct*)
case *silent-moves-Nil* **thus** *?case* **by** *simp*
next
case (*silent-moves-Cons S f ms s a ms' s' as ms'' s''*)
from ‹*S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*)›
have *pred* (*f a*) *s* **and** *transfer* (*f a*) *s* = *s'* **by**(*auto elim*:*silent-move.cases*)
with ‹*preds* (*map f as*) *s'* ∧ *transfers* (*map f as*) *s'* = *s''*›
show *?case* **by** *fastforce*
qed
thus *preds* (*map f as*) *s* **and** *transfers* (*map f as*) *s* = *s'* **by** *simp-all*
qed

lemma *silent-moves-intra-path-obs*:
assumes *m'* ∈ *obs-intra m* ⌊*HRB-slice S*⌋$_{CFG}$ **and** *length s* = *length* (*m*#*msx'*)
and ∀ *m* ∈ *set msx'*. *return-node m*
obtains *as'* **where** *S*,*slice-kind S* ⊢ (*m*#*msx'*,*s*) =*as'*⇒$_\tau$ (*m'*#*msx'*,*s*)
proof(*atomize-elim*)
from ‹*m'* ∈ *obs-intra m* ⌊*HRB-slice S*⌋$_{CFG}$›
obtain *as* **where** *m* −*as*→$_\iota$* *m'* **and** *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$
by −(*erule obs-intraE*)

from ‹$m\ -as\rightarrow_\iota*\ m'$› **obtain** x **where** *distance* $m\ m'\ x$ **and** $x \leq$ *length as*
by(*erule every-path-distance*)
from ‹*distance* $m\ m'\ x$› ‹$m' \in$ *obs-intra* $m\ \lfloor$*HRB-slice* $S\rfloor_{CFG}$›
‹*length* s = *length* $(m\#msx')$› ‹$\forall\, m \in$ *set* msx'. *return-node* m›
show $\exists$ *as*. S,*slice-kind* $S \vdash (m\#msx',s) =as\Rightarrow_\tau (m'\#msx',s)$
proof(*induct x arbitrary*:m s *rule*:*nat.induct*)
fix m **fix** s::$(('var \rightharpoonup 'val) \times 'ret)$ *list*
assume *distance* $m\ m'\ 0$ **and** *length* s = *length* $(m\#msx')$
then obtain *as'* **where** $m\ -as'\rightarrow_\iota*\ m'$ **and** *length as'* = 0
by(*auto elim*:*distance.cases*)
hence $m\ -[]\rightarrow_\iota*\ m'$ **by**(*cases as*) *auto*
hence [*simp*]:$m = m'$ **by**(*fastforce elim*:*path.cases simp*:*intra-path-def*)
with ‹*length* s = *length* $(m\#msx')$›[*THEN sym*]
have S,*slice-kind* $S \vdash (m\#msx',s) =[]\Rightarrow_\tau (m\#msx',s)$
by −(*rule silent-moves-Nil*)
thus $\exists$ *as*. S,*slice-kind* $S \vdash (m\#msx',s) =as\Rightarrow_\tau (m'\#msx',s)$ **by** *simp blast*
next
fix x m **fix** s::$(('var \rightharpoonup 'val) \times 'ret)$ *list*
assume *distance* $m\ m'$ (*Suc* x) **and** $m' \in$ *obs-intra* $m\ \lfloor$*HRB-slice* $S\rfloor_{CFG}$
and *length* s = *length* $(m\#msx')$ **and** $\forall\, m \in$ *set* msx'. *return-node* m
and *IH*:$\bigwedge m\ s$. $[\![$*distance* $m\ m'\ x$; $m' \in$ *obs-intra* $m\ \lfloor$*HRB-slice* $S\rfloor_{CFG}$;
length s = *length* $(m\#msx')$; $\forall\, m \in$ *set* msx'. *return-node* $m]\!]$
$\Longrightarrow \exists$ *as*. S,*slice-kind* $S \vdash (m\#msx',s) =as\Rightarrow_\tau (m'\#msx',s)$
from ‹$m' \in$ *obs-intra* $m\ \lfloor$*HRB-slice* $S\rfloor_{CFG}$› **have** *valid-node* m
by(*rule in-obs-intra-valid*)
with ‹*distance* $m\ m'$ (*Suc* x)› **have** $m \neq m'$
by(*fastforce elim*:*distance.cases dest*:*empty-path simp*:*intra-path-def*)
have $m \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$
proof
assume *isin*:$m \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$
with ‹*valid-node* m› **have** *obs-intra* $m\ \lfloor$*HRB-slice* $S\rfloor_{CFG} = \{m\}$
by(*fastforce intro*!:*n-in-obs-intra*)
with ‹$m' \in$ *obs-intra* $m\ \lfloor$*HRB-slice* $S\rfloor_{CFG}$› ‹$m \neq m'$› **show** *False* **by** *simp*
qed
from ‹*distance* $m\ m'$ (*Suc* x)› **obtain** a **where** *valid-edge* a **and** m = *sourcenode* a
and *intra-kind*(*kind* a) **and** *distance* (*targetnode* a) $m'\ x$
and *target*:*targetnode* a = (*SOME* mx. $\exists\, a'$. *sourcenode* a = *sourcenode* $a' \wedge$
distance (*targetnode* a') $m'\ x \wedge$
valid-edge $a' \wedge$ *intra-kind* (*kind* a') $\wedge$
targetnode $a' = mx$)
by −(*erule distance-successor-distance*,*simp*+)
from ‹$m' \in$ *obs-intra* $m\ \lfloor$*HRB-slice* $S\rfloor_{CFG}$›
have *obs-intra* $m\ \lfloor$*HRB-slice* $S\rfloor_{CFG} = \{m'\}$
by(*rule obs-intra-singleton-element*)
with ‹*valid-edge* a› ‹$m \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$› ‹m = *sourcenode* a› ‹*intra-kind*(*kind* a)›
have *disj*:*obs-intra* (*targetnode* a) $\lfloor$*HRB-slice* $S\rfloor_{CFG} = \{\} \vee$
obs-intra (*targetnode* a) $\lfloor$*HRB-slice* $S\rfloor_{CFG} = \{m'\}$

by $-$(*drule-tac* S=$\lfloor$*HRB-slice* $S\rfloor_{CFG}$ **in** *edge-obs-intra-subset*,*auto*)
from ‹*intra-kind*(*kind a*)› ‹*length s* = *length* (*m*#*msx*′)› ‹*m* ∉ $\lfloor$*HRB-slice* $S\rfloor_{CFG}$›
‹*m* = *sourcenode a*›
have *length*:*length* (*transfer* (*slice-kind S a*) *s*) = *length* (*targetnode a*#*msx*′)
by(*cases s*)
(*auto split*:*split-if-asm simp add*:*Let-def slice-kind-def intra-kind-def*)
from ‹*distance* (*targetnode a*) *m*′ *x*› **obtain** *asx* **where** *targetnode a* $-asx\rightarrow_\iota*$ *m*′
and *length asx* = *x* **and** ∀ *as*′. *targetnode a* $-as'\rightarrow_\iota*$ *m*′ ⟶ *x* ≤ *length as*′
by(*auto elim*:*distance.cases*)
from ‹*targetnode a* $-asx\rightarrow_\iota*$ *m*′› ‹*m*′ ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
obtain *mx* **where** *mx* ∈ *obs-intra* (*targetnode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
by(*erule path-ex-obs-intra*)
with *disj* **have** *m*′ ∈ *obs-intra* (*targetnode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by** *fastforce*
from *IH*[*OF* ‹*distance* (*targetnode a*) *m*′ *x*› *this length*
‹∀ *m* ∈ *set msx*′. *return-node m*›]
obtain *asx*′ **where** *moves*:*S*,*slice-kind S* ⊢
(*targetnode a*#*msx*′,*transfer* (*slice-kind S a*) *s*) =*asx*′⇒$_\tau$
(*m*′#*msx*′,*transfer* (*slice-kind S a*) *s*) **by** *blast*
have *pred* (*slice-kind S a*) *s* ∧ *transfer* (*slice-kind S a*) *s* = *s*
proof(*cases kind a*)
fix *f* **assume** *kind a* = ⇑*f*
with ‹*m* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*m* = *sourcenode a*› **have** *slice-kind S a* = ⇑*id*
by(*fastforce intro*:*slice-kind-Upd*)
with ‹*length s* = *length* (*m*#*msx*′)› **show** *?thesis* **by**(*cases s*) *auto*
next
fix *Q* **assume** *kind a* = $(Q)_{\surd}$
with ‹*m* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*m* = *sourcenode a*›
‹*m*′ ∈ *obs-intra m* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*distance* (*targetnode a*) *m*′ *x*›
‹*distance m m*′ (*Suc x*)› *target*
have *slice-kind S a* = $(\lambda s.\ True)_{\surd}$
by(*fastforce intro*:*slice-kind-Pred-obs-nearer-SOME*)
with ‹*length s* = *length* (*m*#*msx*′)› **show** *?thesis* **by**(*cases s*) *auto*
next
fix *Q r p fs* **assume** *kind a* = *Q*:*r*↪$_p$*fs*
with ‹*intra-kind*(*kind a*)› **have** *False* **by**(*simp add*:*intra-kind-def*)
thus *?thesis* **by** *simp*
next
fix *Q p f* **assume** *kind a* = *Q*↩$_p$*f*
with ‹*intra-kind*(*kind a*)› **have** *False* **by**(*simp add*:*intra-kind-def*)
thus *?thesis* **by** *simp*
qed
hence *pred* (*slice-kind S a*) *s* **and** *transfer* (*slice-kind S a*) *s* = *s*
by *simp-all*
with ‹*m* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*m* = *sourcenode a*› ‹*valid-edge a*›
‹*intra-kind*(*kind a*)› ‹*length s* = *length* (*m*#*msx*′)› ‹∀ *m* ∈ *set msx*′. *return-node m*›
have *S*,*slice-kind S* ⊢ (*sourcenode a*#*msx*′,*s*) $-a\rightarrow_\tau$

```
                    (targetnode a#msx',transfer (slice-kind S a) s)
    by(fastforce intro:silent-move-intra)
   with moves ‹transfer (slice-kind S a) s = s› ‹m = sourcenode a›
   have S,slice-kind S ⊢ (m#msx',s) =a#asx'⇒τ (m'#msx',s)
    by(fastforce intro:silent-moves-Cons)
   thus ∃as. S,slice-kind S ⊢ (m#msx',s) =as⇒τ (m'#msx',s) by blast
  qed
qed


lemma silent-moves-intra-path-no-obs:
  assumes obs-intra m ⌊HRB-slice S⌋CFG = {} and method-exit m'
 and get-proc m = get-proc m' and valid-node m and length s = length (m#msx')
  and ∀m ∈ set msx'. return-node m
  obtains as where S,slice-kind S ⊢ (m#msx',s) =as⇒τ (m'#msx',s)
proof(atomize-elim)
  from ‹method-exit m'› ‹get-proc m = get-proc m'› ‹valid-node m›
  obtain as where m −as→ι* m' by(erule intra-path-to-matching-method-exit)
  then obtain x where distance m m' x and x ≤ length as
    by(erule every-path-distance)
  from ‹distance m m' x› ‹m −as→ι* m'› ‹obs-intra m ⌊HRB-slice S⌋CFG = {}›
    ‹length s = length (m#msx')› ‹∀m ∈ set msx'. return-node m›
  show ∃as. S,slice-kind S ⊢ (m#msx',s) =as⇒τ (m'#msx',s)
  proof(induct x arbitrary:m as s rule:nat.induct)
    fix m fix s::(('var ⇀ 'val) × 'ret) list
    assume distance m m' 0 and length s = length (m#msx')
    then obtain as' where m −as'→ι* m' and length as' = 0
      by(auto elim:distance.cases)
    hence m −[]→ι* m' by(cases as) auto
    hence [simp]:m = m' by(fastforce elim:path.cases simp:intra-path-def)
    with ‹length s = length (m#msx')›[THEN sym]
    have S,slice-kind S ⊢ (m#msx',s) =[]⇒τ (m#msx',s)
      by(fastforce intro:silent-moves-Nil)
    thus ∃as. S,slice-kind S ⊢ (m#msx',s) =as⇒τ (m'#msx',s) by simp blast
  next
    fix x m as fix s::(('var ⇀ 'val) × 'ret) list
    assume distance m m' (Suc x) and m −as→ι* m'
      and obs-intra m ⌊HRB-slice S⌋CFG = {}
      and length s = length (m#msx') and ∀m ∈ set msx'. return-node m
      and IH:⋀m as s. ⟦distance m m' x; m −as→ι* m';
      obs-intra m ⌊HRB-slice S⌋CFG = {}; length s = length (m#msx');
      ∀m ∈ set msx'. return-node m⟧
      ⟹ ∃as. S,slice-kind S ⊢ (m#msx',s) =as⇒τ (m'#msx',s)
    from ‹m −as→ι* m'› have valid-node m
      by(fastforce intro:path-valid-node simp:intra-path-def)
   from ‹m −as→ι* m'› have get-proc m = get-proc m' by(rule intra-path-get-procs)
    have m ∉ ⌊HRB-slice S⌋CFG
    proof
      assume m ∈ ⌊HRB-slice S⌋CFG
```

with ‹*valid-node m*› **have** *obs-intra m* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {*m*}
by(*fastforce intro*!:*n-in-obs-intra*)
with ‹*obs-intra m* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {}› **show** *False* **by** *simp*
qed
from ‹*distance m m′* (*Suc x*)› **obtain** *a* **where** *valid-edge a* **and** *m* = *sourcenode a*
and *intra-kind*(*kind a*) **and** *distance* (*targetnode a*) *m′ x*
and *target*:*targetnode a* = (*SOME mx*. ∃ *a′*. *sourcenode a* = *sourcenode a′* ∧
distance (*targetnode a′*) *m′ x* ∧
valid-edge a′ ∧ *intra-kind* (*kind a′*) ∧
targetnode a′ = *mx*)
by −(*erule distance-successor-distance*,*simp*+)
from ‹*intra-kind*(*kind a*)› ‹*length s* = *length* (*m*#*msx′*)› ‹*m* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
‹*m* = *sourcenode a*›
have *length*:*length* (*transfer* (*slice-kind S a*) *s*) = *length* (*targetnode a*#*msx′*)
by(*cases s*)
(*auto split*:*split-if-asm simp add*:*Let-def slice-kind-def intra-kind-def*)
from ‹*distance* (*targetnode a*) *m′ x*› **obtain** *asx* **where** *targetnode a* −*asx*→$_\iota$* *m′*
and *length asx* = *x* **and** ∀ *as′*. *targetnode a* −*as′*→$_\iota$* *m′* ⟶ *x* ≤ *length as′*
by(*auto elim*:*distance.cases*)
from ‹*valid-edge a*› ‹*intra-kind*(*kind a*)› ‹*m* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
‹*m* = *sourcenode a*› ‹*obs-intra m* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {}›
have *obs-intra* (*targetnode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {}
by(*fastforce dest*:*edge-obs-intra-subset*)
from *IH*[*OF* ‹*distance* (*targetnode a*) *m′ x*› ‹*targetnode a* −*asx*→$_\iota$* *m′*› *this*
length ‹∀ *m* ∈ *set msx′*. *return-node m*›] **obtain** *as′*
where *moves*:*S*,*slice-kind S* ⊢
(*targetnode a*#*msx′*,*transfer* (*slice-kind S a*) *s*) =*as′*⇒$_\tau$
(*m′*#*msx′*,*transfer* (*slice-kind S a*) *s*) **by** *blast*
have *pred* (*slice-kind S a*) *s* ∧ *transfer* (*slice-kind S a*) *s* = *s*
proof(*cases kind a*)
fix *f* **assume** *kind a* = ⇑*f*
with ‹*m* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*m* = *sourcenode a*› **have** *slice-kind S a* = ⇑*id*
by(*fastforce intro*:*slice-kind-Upd*)
with ‹*length s* = *length* (*m*#*msx′*)› **show** *?thesis* **by**(*cases s*) *auto*
next
fix *Q* **assume** *kind a* = (*Q*)$_\surd$
with ‹*m* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*m* = *sourcenode a*›
‹*obs-intra m* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ = {}› ‹*distance* (*targetnode a*) *m′ x*›
‹*distance m m′* (*Suc x*)› ‹*method-exit m′*› ‹*get-proc m* = *get-proc m′*› *target*
have *slice-kind S a* = (*λs*. *True*)$_\surd$
by(*fastforce intro*:*slice-kind-Pred-empty-obs-nearer-SOME*)
with ‹*length s* = *length* (*m*#*msx′*)› **show** *?thesis* **by**(*cases s*) *auto*
next
fix *Q r p fs* **assume** *kind a* = *Q*:*r*↪$_p$*fs*
with ‹*intra-kind*(*kind a*)› **have** *False* **by**(*simp add*:*intra-kind-def*)
thus *?thesis* **by** *simp*

```
    next
      fix Q p f assume kind a = Q↩pf
      with ‹intra-kind(kind a)› have False by(simp add:intra-kind-def)
      thus ?thesis by simp
    qed
    hence pred (slice-kind S a) s and transfer (slice-kind S a) s = s
      by simp-all
    with ‹m ∉ ⌊HRB-slice S⌋CFG› ‹m = sourcenode a› ‹valid-edge a›
     ‹intra-kind(kind a)› ‹length s = length (m#msx')› ‹∀ m ∈ set msx'. return-node
m›
    have S,slice-kind S ⊢ (sourcenode a#msx',s) −a→τ
                          (targetnode a#msx',transfer (slice-kind S a) s)
      by(fastforce intro:silent-move-intra)
    with moves ‹transfer (slice-kind S a) s = s› ‹m = sourcenode a›
    have S,slice-kind S ⊢ (m#msx',s) =a#as'⇒τ (m'#msx',s)
      by(fastforce intro:silent-moves-Cons)
    thus ∃ as. S,slice-kind S ⊢ (m#msx',s) =as⇒τ (m'#msx',s) by blast
  qed
qed


lemma silent-moves-vpa-path:
  assumes S,f ⊢ (m#ms,s) =as⇒τ (m'#ms',s') and valid-node m
  and ∀ i < length rs. rs!i ∈ get-return-edges (cs!i)
  and ms = targetnodes rs and valid-return-list rs m
  and length rs = length cs
  shows m −as→* m' and valid-path-aux cs as
proof −
  from assms have m −as→* m' ∧ valid-path-aux cs as
  proof(induct S f m#ms s as m'#ms' s' arbitrary:m cs ms rs
      rule:silent-moves.induct)
    case (silent-moves-Nil msx sx nc f)
    from ‹valid-node m'› have m' −[]→* m'
      by (rule empty-path)
    thus ?case by fastforce
  next
    case (silent-moves-Cons S f sx a msx' sx' as s'')
    thus ?case
    proof(induct - - m # ms - - - - rule:silent-move.induct)
      case (silent-move-intra f a sx sx' nc msx')
      note IH = ‹⋀m cs ms rs. ⟦msx' = m # ms; 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⟧
        ⟹ m −as→* m' ∧ valid-path-aux cs as›
      from ‹msx' = targetnode a # tl (m # ms)›
      have msx' = targetnode a # ms by simp
      from ‹valid-edge a› ‹intra-kind (kind a)›
      have get-proc (sourcenode a) = get-proc (targetnode a)
```

by(*rule get-proc-intra*)
with ‹*valid-return-list rs m*› ‹*hd* (*m* # *ms*) = *sourcenode a*›
have *valid-return-list rs* (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*) *auto*
from ‹*valid-edge a*› **have** *valid-node* (*targetnode a*) **by** *simp*
from *IH*[*OF* ‹*msx'* = *targetnode a* # *ms*› *this*
‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*ms* = *targetnodes rs*› ‹*valid-return-list rs* (*targetnode a*)›
‹*length rs* = *length cs*›]
have *targetnode a* −*as*→∗ *m'* **and** *valid-path-aux cs as* **by** *simp-all*
from ‹*valid-edge a*› ‹*targetnode a* −*as*→∗ *m'*›
‹*hd* (*m* # *ms*) = *sourcenode a*›
have *m* −*a*#*as*→∗ *m'* **by**(*fastforce intro*:*Cons-path*)
moreover
from ‹*intra-kind* (*kind a*)› ‹*valid-path-aux cs as*›
have *valid-path-aux cs* (*a* # *as*) **by**(*fastforce simp*:*intra-kind-def*)
ultimately show *?case* **by** *simp*
next
case (*silent-move-call f a sx sx' Q r p fs a' n$_c$ msx'*)
note *IH* = ‹⋀*m cs ms rs*. ⟦*msx'* = *m* # *ms*; *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*⟧
⟹ *m* −*as*→∗ *m'* ∧ *valid-path-aux cs as*›
from ‹*valid-edge a*› **have** *valid-node* (*targetnode a*) **by** *simp*
from ‹*length rs* = *length cs*›
have *length* (*a'*#*rs*) = *length* (*a*#*cs*) **by** *simp*
from ‹*msx'* = *targetnode a* # *targetnode a'* # *tl* (*m* # *ms*)›
have *msx'* = *targetnode a* # *targetnode a'* # *ms* **by** *simp*
from ‹*ms* = *targetnodes rs*› **have** *targetnode a'* # *ms* = *targetnodes* (*a'* # *rs*)
by(*simp add*:*targetnodes-def*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› **have** *get-proc* (*targetnode a*) = *p*
by(*rule get-proc-call*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*a'* ∈ *get-return-edges a*›
obtain *Q' f'* **where** *kind a'* = *Q'*↩$_p$*f'* **by**(*fastforce dest*!:*call-return-edges*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a'*)
by(*rule get-proc-get-return-edge*)
with ‹*valid-return-list rs m*› ‹*hd* (*m* # *ms*) = *sourcenode a*›
‹*get-proc* (*targetnode a*) = *p*› ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩$_p$*f'*›
have *valid-return-list* (*a'* # *rs*) (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x*=*list* **in** *allE*) **apply** *clarsimp*

by(*case-tac list*)(*auto simp*:*targetnodes-def*)
from ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*a'* ∈ *get-return-edges a*›
have ∀ *i*<*length* (*a'*#*rs*). (*a'*#*rs*) ! *i* ∈ *get-return-edges* ((*a*#*cs*) ! *i*)
by *auto*(*case-tac i*,*auto*)
from *IH*[*OF* ‹*msx'* = *targetnode a* # *targetnode a'* # *ms*› ‹*valid-node* (*targetnode a*)› *this*
‹*targetnode a'* # *ms* = *targetnodes* (*a'* # *rs*)›
‹*valid-return-list* (*a'* # *rs*) (*targetnode a*)› ‹*length* (*a'*#*rs*) = *length* (*a*#*cs*)›]
have *targetnode a* −*as*→∗ *m'* **and** *valid-path-aux* (*a* # *cs*) *as* **by** *simp-all*
from ‹*valid-edge a*› ‹*targetnode a* −*as*→∗ *m'*›
‹*hd* (*m* # *ms*) = *sourcenode a*›
have *m* −*a*#*as*→∗ *m'* **by**(*fastforce intro*:*Cons-path*)
moreover
from ‹*valid-path-aux* (*a* # *cs*) *as*› ‹*kind a* = *Q*:*r*↪*p*f*s*›
have *valid-path-aux cs* (*a* # *as*) **by** *simp*
ultimately show *?case* **by** *simp*
next
case (*silent-move-return f a sx sx' Q p f' n*$_c$ *msx'*)
note *IH* = ‹⋀*m cs ms rs*. ⟦*msx'* = *m* # *ms*; *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*⟧
⟹ *m* −*as*→∗ *m'* ∧ *valid-path-aux cs as*›
from ‹*valid-edge a*› **have** *valid-node* (*targetnode a*) **by** *simp*
from ‹*length* (*m* # *ms*) = *length sx*› ‹*length sx* = *Suc* (*length sx'*)›
‹*sx'* ≠ []›
obtain *x xs* **where** *ms* = *x*#*xs* **by**(*cases ms*) *auto*
with ‹*ms* = *targetnodes rs*› **obtain** *r' rs'* **where** *rs* = *r'*#*rs'*
and *x* = *targetnode r'* **and** *xs* = *targetnodes rs'*
by(*auto simp*:*targetnodes-def*)
with ‹*length rs* = *length cs*› **obtain** *c' cs'* **where** *cs* = *c'*#*cs'*
and *length rs'* = *length cs'*
by(*cases cs*) *auto*
from ‹*ms* = *x*#*xs*› ‹*length* (*m* # *ms*) = *length sx*›
‹*length sx* = *Suc* (*length sx'*)›
have *length sx'* = *Suc* (*length xs*) **by** *simp*
from ‹*ms* = *x*#*xs*› ‹*msx'* = *tl* (*m* # *ms*)› ‹*hd* (*tl* (*m* # *ms*)) = *targetnode a*›
‹*length* (*m* # *ms*) = *length sx*› ‹*length sx* = *Suc* (*length sx'*)› ‹*sx'* ≠ []›
have *msx'* = *targetnode a*#*xs* **by** *simp*
from ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*rs* = *r'*#*rs'*› ‹*cs* = *c'*#*cs'*›
have *r'* ∈ *get-return-edges c'* **by** *fastforce*
from ‹*ms* = *x*#*xs*› ‹*hd* (*tl* (*m* # *ms*)) = *targetnode a*›
have *x* = *targetnode a* **by** *simp*
with ‹*valid-return-list rs m*› ‹*rs* = *r'*#*rs'*› ‹*x* = *targetnode r'*›
have *valid-return-list rs'* (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*r'*#*cs'* **in** *allE*) **apply** *clarsimp*

by(*case-tac cs'*)(*auto simp*:*targetnodes-def*)
from ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*rs* = *r'*#*rs'*› ‹*cs* = *c'*#*cs'*›
have ∀ *i*<*length rs'*. *rs'* ! *i* ∈ *get-return-edges* (*cs'* ! *i*)
and *r'* ∈ *get-return-edges c'* **by** *auto*
from *IH*[*OF* ‹*msx'* = *targetnode a*#*xs*› ‹*valid-node* (*targetnode a*)›
‹∀ *i*<*length rs'*. *rs'* ! *i* ∈ *get-return-edges* (*cs'* ! *i*)› ‹*xs* = *targetnodes rs'*›
‹*valid-return-list rs'* (*targetnode a*)› ‹*length rs'* = *length cs'*›]
have *targetnode a* −*as*→∗ *m'* **and** *valid-path-aux cs' as* **by** *simp-all*
from ‹*valid-edge a*› ‹*targetnode a* −*as*→∗ *m'*›
‹*hd* (*m* # *ms*) = *sourcenode a*›
have *m* −*a*#*as*→∗ *m'* **by**(*fastforce intro*:*Cons-path*)
moreover
from ‹*ms* = *x*#*xs*› ‹*hd* (*tl* (*m* # *ms*)) = *targetnode a*›
have *x* = *targetnode a* **by** *simp*
from ‹*valid-edge a*› ‹*kind a* = *Q*↪*pf'*›
have *method-exit* (*sourcenode a*) **by**(*fastforce simp*:*method-exit-def*)
from ‹*valid-return-list rs m*› ‹*hd* (*m* # *ms*) = *sourcenode a*›
‹*rs* = *r'*#*rs'*›
have *get-proc* (*sourcenode a*) = *get-proc* (*sourcenode r'*) ∧
method-exit (*sourcenode r'*) ∧ *valid-edge r'*
apply(*clarsimp simp*:*valid-return-list-def method-exit-def*)
apply(*erule-tac x*=[] **in** *allE*)
by(*auto dest*:*get-proc-return*)
hence *get-proc* (*sourcenode a*) = *get-proc* (*sourcenode r'*)
and *method-exit* (*sourcenode r'*) **and** *valid-edge r'* **by** *simp-all*
with ‹*method-exit* (*sourcenode a*)› **have** *sourcenode r'* = *sourcenode a*
by(*fastforce intro*:*method-exit-unique*)
with ‹*valid-edge a*› ‹*valid-edge r'*› ‹*x* = *targetnode r'*› ‹*x* = *targetnode a*›
have *r'* = *a* **by**(*fastforce intro*:*edge-det*)
with ‹*r'* ∈ *get-return-edges c'*› ‹*valid-path-aux cs' as*› ‹*cs* = *c'*#*cs'*›
‹*kind a* = *Q*↪*pf'*›
have *valid-path-aux cs* (*a* # *as*) **by** *simp*
ultimately show *?case* **by** *simp*
qed
qed
thus *m* −*as*→∗ *m'* **and** *valid-path-aux cs as* **by** *simp-all*
qed

1.13.2 Observable moves

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

where *observable-move-intra*:

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

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

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

inductive *observable-moves* ::
'node SDG-node set ⇒ (*'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] *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 −
from ‹*S*,*f* ⊢ (*ms*,*s*) −*a*→ (*ms'*,*s'*)›
have *length ms* = *length s* ∧ *length ms'* = *length s'*
proof(*induct rule*:*observable-move.induct*)
case (*observable-move-intra f a s s' ms S ms'*)
from ‹*pred* (*f a*) *s*› **obtain** *cf cfs* **where** [*simp*]:*s* = *cf*#*cfs* **by**(*cases s*) *auto*
from ‹*length ms* = *length s*› ‹*ms'* = *targetnode a* # *tl ms*›
‹*length s'* = *length s*› **show** *?case* **by** *simp*
next
case (*observable-move-call f a s s' Q r p fs a' ms S ms'*)
from ‹*pred* (*f a*) *s*› **obtain** *cf cfs* **where** [*simp*]:*s* = *cf*#*cfs* **by**(*cases s*) *auto*

from ‹*length ms = length s*› ‹*length s' = Suc (length s)*›
‹*ms' = targetnode a # targetnode a' # tl ms*› **show** *?case* **by** *simp*
next
case (*observable-move-return f a s s' Q p f' ms S ms'*)
from ‹*length ms = length s*› ‹*length s = Suc (length s')*› ‹*ms' = tl ms*› ‹*s' ≠ []*›
show *?case* **by** *simp*
qed
thus *length ms = length s* **and** *length ms' = length s'* **by** *simp-all*
qed

lemma *observable-moves-equal-length*:
assumes *S,f ⊢ (ms,s) =as⇒ (ms',s')*
shows *length ms = length s* **and** *length ms' = length s'*
using ‹*S,f ⊢ (ms,s) =as⇒ (ms',s')*›
proof(*induct rule:observable-moves.induct*)
case (*observable-moves-snoc S f ms s as ms' s' a ms'' s''*)
from ‹*S,f ⊢ (ms',s') −a→ (ms'',s'')*›
have *length ms' = length s' length ms'' = length s''*
by(*rule observable-move-equal-length*)+
moreover
from ‹*S,f ⊢ (ms,s) =as⇒τ (ms',s')*›
have *length ms = length s* **and** *length ms' = length s'*
by(*rule silent-moves-equal-length*)+
ultimately show *length ms = length s length ms'' = length s''* **by** *simp-all*
qed

lemma *observable-move-notempty*:
⟦*S,f ⊢ (ms,s) =as⇒ (ms',s')*; *as = []*⟧ ⟹ *False*
by(*induct rule:observable-moves.induct,simp*)

lemma *silent-move-observable-moves*:
⟦*S,f ⊢ (ms'',s'') =as⇒ (ms',s')*; *S,f ⊢ (ms,s) −a→τ (ms'',s'')*⟧
⟹ *S,f ⊢ (ms,s) =a#as⇒ (ms',s')*
proof(*induct rule:observable-moves.induct*)
case (*observable-moves-snoc S f msx sx as ms' s' a' ms'' s''*)
from ‹*S,f ⊢ (ms,s) −a→τ (msx,sx)*› ‹*S,f ⊢ (msx,sx) =as⇒τ (ms',s')*›
have *S,f ⊢ (ms,s) =a#as⇒τ (ms',s')* **by**(*fastforce intro:silent-moves-Cons*)
with ‹*S,f ⊢ (ms',s') −a'→ (ms'',s'')*›
have *S,f ⊢ (ms,s) =(a#as)@[a']⇒ (ms'',s'')*
by(*fastforce intro:observable-moves.observable-moves-snoc*)
thus *?case* **by** *simp*
qed

lemma *silent-append-observable-moves*:
⟦*S,f ⊢ (ms,s) =as⇒τ (ms'',s'')*; *S,f ⊢ (ms'',s'') =as'⇒ (ms',s')*⟧

$\Longrightarrow S,f \vdash (ms,s) =as@as'\Rightarrow (ms',s')$
by(*induct rule*:*silent-moves.induct*)(*auto elim*:*silent-move-observable-moves*)

lemma *observable-moves-preds-transfers*:
assumes $S,f \vdash (ms,s) =as\Rightarrow (ms',s')$
shows *preds* (*map f as*) *s* **and** *transfers* (*map f as*) *s* = *s'*
proof −
from ‹$S,f \vdash (ms,s) =as\Rightarrow (ms',s')$›
have *preds* (*map f as*) *s* ∧ *transfers* (*map f as*) *s* = *s'*
proof(*induct rule*:*observable-moves.induct*)
case (*observable-moves-snoc S f ms s as ms' s' a ms'' s''*)
from ‹$S,f \vdash (ms,s) =as\Rightarrow_\tau (ms',s')$›
have *preds* (*map f as*) *s* **and** *transfers* (*map f as*) *s* = *s'*
by(*rule silent-moves-preds-transfers*)+
from ‹$S,f \vdash (ms',s') -a\rightarrow (ms'',s'')$›
have *pred* (*f a*) *s'* **and** *transfer* (*f a*) *s'* = *s''*
by(*auto elim*:*observable-move.cases*)
with ‹*preds* (*map f as*) *s*› ‹*transfers* (*map f as*) *s* = *s'*›
show *?case* **by**(*simp add*:*preds-split transfers-split*)
qed
thus *preds* (*map f as*) *s* **and** *transfers* (*map f as*) *s* = *s'* **by** *simp-all*
qed

lemma *observable-move-vpa-path*:
⟦$S,f \vdash (m\#ms,s) -a\rightarrow (m'\#ms',s')$; *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(*induct S f m#ms s a m'#ms' s' rule*:*observable-move.induct*)
case (*observable-move-return f a sx sx' Q p f' n$_c$*)
from ‹*length* (*m* # *ms*) = *length sx*› ‹*length sx* = *Suc* (*length sx'*)›
‹*sx'* ≠ []›
obtain *x xs* **where** *ms* = *x*#*xs* **by**(*cases ms*) *auto*
with ‹*ms* = *targetnodes rs*› **obtain** *r' rs'* **where** *rs* = *r'*#*rs'*
and *x* = *targetnode r'* **and** *xs* = *targetnodes rs'*
by(*auto simp*:*targetnodes-def*)
with ‹*length rs* = *length cs*› **obtain** *c' cs'* **where** *cs* = *c'*#*cs'*
and *length rs'* = *length cs'*
by(*cases cs*) *auto*
from ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*rs* = *r'*#*rs'*› ‹*cs* = *c'*#*cs'*›
have ∀ *i*<*length rs'*. *rs'* ! *i* ∈ *get-return-edges* (*cs'* ! *i*)
and *r'* ∈ *get-return-edges c'* **by** *auto*
from ‹*ms* = *x*#*xs*› ‹*hd* (*tl* (*m* # *ms*)) = *targetnode a*›
have *x* = *targetnode a* **by** *simp*
from ‹*valid-edge a*› ‹*kind a* = *Q*↩$_p$*f'*›
have *method-exit* (*sourcenode a*) **by**(*fastforce simp*:*method-exit-def*)
from ‹*valid-return-list rs m*› ‹*hd* (*m* # *ms*) = *sourcenode a*›

$\langle rs = r'\#rs'\rangle$
have *get-proc* (*sourcenode a*) = *get-proc* (*sourcenode r'*) ∧
method-exit (*sourcenode r'*) ∧ *valid-edge r'*
apply(*clarsimp simp*:*valid-return-list-def method-exit-def*)
apply(*erule-tac x*=[] **in** *allE*)
by(*auto dest*:*get-proc-return*)
hence *get-proc* (*sourcenode a*) = *get-proc* (*sourcenode r'*)
and *method-exit* (*sourcenode r'*) **and** *valid-edge r'* **by** *simp-all*
with ⟨*method-exit* (*sourcenode a*)⟩ **have** *sourcenode r'* = *sourcenode a*
by(*fastforce intro*:*method-exit-unique*)
with ⟨*valid-edge a*⟩ ⟨*valid-edge r'*⟩ ⟨*x* = *targetnode r'*⟩ ⟨*x* = *targetnode a*⟩
have *r'* = *a* **by**(*fastforce intro*:*edge-det*)
with ⟨*r'* ∈ *get-return-edges c'*⟩ ⟨*cs* = *c'*#*cs'*⟩ ⟨*kind a* = *Q*↩$_{pf}$'⟩
show *?case* **by** *simp*
qed(*auto simp*:*intra-kind-def*)

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''*
using *rv*
by(*atomize-elim*,*auto elim*!:*relevant-vars.cases*)

lemma *rv-parent-node*:
parent-node n = *parent-node n'* ⟹ *rv* (*S*::*'node SDG-node set*) *n* = *rv S n'*
by(*fastforce elim*:*rvE intro*:*rvI*)

lemma *obs-intra-empty-rv-empty*:
assumes *obs-intra m* ⌊*HRB-slice S*⌋$_{CFG}$ = {} **shows** *rv S* (*CFG-node m*) = {}
proof(*rule ccontr*)
assume *rv S* (*CFG-node m*) ≠ {}
then obtain *x* **where** *x* ∈ *rv S* (*CFG-node m*) **by** *fastforce*

then obtain n' *as* **where** $m\ -as\rightarrow_\iota *\ parent\text{-}node\ n'$ **and** $n' \in HRB\text{-}slice\ S$
by(*fastforce elim*:*rvE*)
hence $parent\text{-}node\ n' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*fastforce intro*:*valid-SDG-node-in-slice-parent-node-in-slice*
simp:*SDG-to-CFG-set-def*)
with ‹$m\ -as\rightarrow_\iota *\ parent\text{-}node\ n'$› **obtain** *mx* **where** $mx \in obs\text{-}intra\ m\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*erule path-ex-obs-intra*)
with ‹$obs\text{-}intra\ m\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{\}$› **show** *False* **by** *simp*
qed

lemma *eq-obs-intra-in-rv*:
assumes *obs-eq*:$obs\text{-}intra\ (parent\text{-}node\ n)\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = obs\text{-}intra\ (parent\text{-}node\ n')\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
and $x \in rv\ S\ n$ **shows** $x \in rv\ S\ n'$
proof −
from ‹$x \in rv\ S\ n$› **obtain** *as* n''
where $parent\text{-}node\ n\ -as\rightarrow_\iota *\ parent\text{-}node\ n''$ **and** $n'' \in HRB\text{-}slice\ S$
and $x \in Use_{SDG}\ n''$
and $\forall\, n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge parent\text{-}node\ n'' \in set\ (sourcenodes\ as) \longrightarrow x \notin Def_{SDG}\ n''$
by(*erule rvE*)
from ‹$parent\text{-}node\ n\ -as\rightarrow_\iota *\ parent\text{-}node\ n''$› **have** $valid\text{-}node\ (parent\text{-}node\ n'')$
by(*fastforce dest*:*path-valid-node simp*:*intra-path-def*)
from ‹$parent\text{-}node\ n\ -as\rightarrow_\iota *\ parent\text{-}node\ n''$› ‹$n'' \in HRB\text{-}slice\ S$›
have $\exists\, nx\ as'\ as''.\ parent\text{-}node\ nx \in obs\text{-}intra\ (parent\text{-}node\ n)\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} \wedge$
$parent\text{-}node\ n\ -as'\rightarrow_\iota *\ parent\text{-}node\ nx \wedge$
$parent\text{-}node\ nx\ -as''\rightarrow_\iota *\ parent\text{-}node\ n'' \wedge as = as'@as''$
proof(*cases* $\forall\, nx.\ parent\text{-}node\ nx \in set\ (sourcenodes\ as) \longrightarrow nx \notin HRB\text{-}slice\ S$)
case *True*
with ‹$parent\text{-}node\ n\ -as\rightarrow_\iota *\ parent\text{-}node\ n''$› ‹$n'' \in HRB\text{-}slice\ S$›
have $parent\text{-}node\ n'' \in obs\text{-}intra\ (parent\text{-}node\ n)\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*fastforce intro*:*obs-intra-elem valid-SDG-node-in-slice-parent-node-in-slice*
simp:*SDG-to-CFG-set-def*)
with ‹$parent\text{-}node\ n\ -as\rightarrow_\iota *\ parent\text{-}node\ n''$› ‹$valid\text{-}node\ (parent\text{-}node\ n'')$›
show *?thesis* **by**(*fastforce intro*:*empty-path simp*:*intra-path-def*)
next
case *False*
hence $\exists\, nx.\ parent\text{-}node\ nx \in set\ (sourcenodes\ as) \wedge nx \in HRB\text{-}slice\ S$ **by** *simp*
hence $\exists\, mx \in set\ (sourcenodes\ as).\ \exists\, nx.\ mx = parent\text{-}node\ nx \wedge nx \in HRB\text{-}slice\ S$
by *fastforce*
then obtain *mx ms ms'* **where** $sourcenodes\ as = ms@mx\#ms'$
and $\exists\, nx.\ mx = parent\text{-}node\ nx \wedge nx \in HRB\text{-}slice\ S$
and *all*:$\forall\, x \in set\ ms.\ \neg\ (\exists\, nx.\ x = parent\text{-}node\ nx \wedge nx \in HRB\text{-}slice\ S)$
by(*fastforce elim*!:*split-list-first-propE*)

then obtain nx' **where** $mx = \textit{parent-node}\ nx'$ **and** $nx' \in \textit{HRB-slice}\ S$ **by** *blast*
from ‹*sourcenodes as* = *ms*@*mx*#*ms'*›
obtain *as' a' as''* **where** *ms* = *sourcenodes as'*
and [*simp*]:*as* = *as'*@*a'*#*as''* **and** *sourcenode a'* = *mx*
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from *all* ‹*ms* = *sourcenodes as'*›
have ∀ *nx*∈*set* (*sourcenodes as'*). *nx* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
by(*fastforce simp*:*SDG-to-CFG-set-def*)
from ‹*parent-node n* −*as*→$_\iota$* *parent-node n''*› ‹*sourcenode a'* = *mx*›
have *parent-node n* −*as'*→$_\iota$* *mx* **and** *valid-edge a'* **and** *intra-kind*(*kind a'*)
and *targetnode a'* −*as''*→$_\iota$* *parent-node n''*
by(*fastforce dest*:*path-split simp*:*intra-path-def*)+
with ‹*sourcenode a'* = *mx*› **have** *mx* −*a'*#*as''*→$_\iota$* *parent-node n''*
by(*fastforce intro*:*Cons-path simp*:*intra-path-def*)
from ‹*parent-node n* −*as'*→$_\iota$* *mx*› ‹*mx* = *parent-node nx'*› ‹*nx'* ∈ *HRB-slice S*›
‹∀ *nx*∈*set* (*sourcenodes as'*). *nx* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*ms* = *sourcenodes as'*›
have *mx* ∈ *obs-intra* (*parent-node n*) ⌊*HRB-slice S*⌋$_{CFG}$
by(*fastforce intro*:*obs-intra-elem valid-SDG-node-in-slice-parent-node-in-slice*
simp:*SDG-to-CFG-set-def*)
with ‹*parent-node n* −*as'*→$_\iota$* *mx*› ‹*mx* −*a'*#*as''*→$_\iota$* *parent-node n''*›
‹*mx* = *parent-node nx'*›
show *?thesis* **by** *simp blast*
qed
then obtain *nx as' as''*
where *parent-node nx* ∈ *obs-intra* (*parent-node n*) ⌊*HRB-slice S*⌋$_{CFG}$
and *parent-node n* −*as'*→$_\iota$* *parent-node nx*
and *parent-node nx* −*as''*→$_\iota$* *parent-node n''* **and** [*simp*]:*as* = *as'*@*as''*
by *blast*
from ‹*parent-node nx* ∈ *obs-intra* (*parent-node n*) ⌊*HRB-slice S*⌋$_{CFG}$› *obs-eq*
have *parent-node nx* ∈ *obs-intra* (*parent-node n'*) ⌊*HRB-slice S*⌋$_{CFG}$ **by** *auto*
then obtain *asx* **where** *parent-node n'* −*asx*→$_\iota$* *parent-node nx*
and ∀ *ni* ∈ *set*(*sourcenodes asx*). *ni* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
and *parent-node nx* ∈ ⌊*HRB-slice S*⌋$_{CFG}$
by(*erule obs-intraE*)
from ‹∀ *n''*. *valid-SDG-node n''* ∧ *parent-node n''* ∈ *set* (*sourcenodes as*)
⟶ *x* ∉ *Def*$_{SDG}$ *n''*›
have ∀ *ni*. *valid-SDG-node ni* ∧ *parent-node ni* ∈ *set* (*sourcenodes as''*)
⟶ *x* ∉ *Def*$_{SDG}$ *ni*
by(*auto simp*:*sourcenodes-def*)
from ‹∀ *ni* ∈ *set*(*sourcenodes asx*). *ni* ∉ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*parent-node n'* −*asx*→$_\iota$* *parent-node nx*›
have ∀ *ni*. *valid-SDG-node ni* ∧ *parent-node ni* ∈ *set* (*sourcenodes asx*)
⟶ *x* ∉ *Def*$_{SDG}$ *ni*
proof(*induct asx arbitrary*:*n'*)
case *Nil* **thus** *?case* **by**(*simp add*:*sourcenodes-def*)
next
case (*Cons ax' asx'*)

note $IH = \langle\bigwedge n'.\ [\![\forall\, ni{\in}set\ (sourcenodes\ asx').\ ni \notin \lfloor HRB\text{-}slice\ S\rfloor_{CFG};$
$parent\text{-}node\ n' -asx'\rightarrow_\iota * parent\text{-}node\ nx]\!]$
$\Longrightarrow \forall\, ni.\ valid\text{-}SDG\text{-}node\ ni \wedge parent\text{-}node\ ni \in set\ (sourcenodes\ asx')$
$\longrightarrow x \notin Def_{SDG}\ ni\rangle$
from $\langle parent\text{-}node\ n' -ax'\#asx'\rightarrow_\iota * parent\text{-}node\ nx\rangle$
have $parent\text{-}node\ n' -[]@ax'\#asx'\rightarrow * parent\text{-}node\ nx$
and $\forall\, a \in set\ (ax'\#asx').\ intra\text{-}kind(kind\ a)$ **by**$(simp\text{-}all\ add{:}intra\text{-}path\text{-}def)$
hence $targetnode\ ax' -asx'\rightarrow * parent\text{-}node\ nx$ **and** $valid\text{-}edge\ ax'$
and $parent\text{-}node\ n' = sourcenode\ ax'$ **by**$(fastforce\ dest{:}path\text{-}split)+$
with $\langle\forall\, a \in set\ (ax'\#asx').\ intra\text{-}kind(kind\ a)\rangle$
have $path{:}parent\text{-}node\ (CFG\text{-}node\ (targetnode\ ax')) -asx'\rightarrow_\iota * parent\text{-}node\ nx$
by$(simp\ add{:}intra\text{-}path\text{-}def)$
from $\langle\forall\, ni{\in}set\ (sourcenodes\ (ax'\#asx')).\ ni \notin \lfloor HRB\text{-}slice\ S\rfloor_{CFG}\rangle$
have $all{:}\forall\, ni{\in}set\ (sourcenodes\ asx').\ ni \notin \lfloor HRB\text{-}slice\ S\rfloor_{CFG}$
and $sourcenode\ ax' \notin \lfloor HRB\text{-}slice\ S\rfloor_{CFG}$
by$(auto\ simp{:}sourcenodes\text{-}def)$
from $IH[OF\ all\ path]$
have $\forall\, ni.\ valid\text{-}SDG\text{-}node\ ni \wedge parent\text{-}node\ ni \in set\ (sourcenodes\ asx')$
$\longrightarrow x \notin Def_{SDG}\ ni$ **.**
with $\langle\forall\, ni.\ valid\text{-}SDG\text{-}node\ ni \wedge parent\text{-}node\ ni \in set\ (sourcenodes\ as'')$
$\longrightarrow x \notin Def_{SDG}\ ni\rangle$
have $all{:}\forall\, ni.\ valid\text{-}SDG\text{-}node\ ni \wedge parent\text{-}node\ ni \in set\ (sourcenodes\ (asx'@as''))$

$\longrightarrow x \notin Def_{SDG}\ ni$
by$(auto\ simp{:}sourcenodes\text{-}def)$
from $\langle parent\text{-}node\ n' -ax'\#asx'\rightarrow_\iota * parent\text{-}node\ nx\rangle$
$\langle parent\text{-}node\ nx\ -as''\rightarrow_\iota * parent\text{-}node\ n''\rangle$
have $path{:}parent\text{-}node\ n' -ax'\#asx'@as''\rightarrow_\iota * parent\text{-}node\ n''$
by$(fastforce\ intro{:}path\text{-}Append[of\ -\ ax'\#asx', simplified]\ simp{:}intra\text{-}path\text{-}def)$
have $\forall\, nx'.\ parent\text{-}node\ nx' = sourcenode\ ax' \longrightarrow x \notin Def_{SDG}\ nx'$
proof
fix nx'
show $parent\text{-}node\ nx' = sourcenode\ ax' \longrightarrow x \notin Def_{SDG}\ nx'$
proof
assume $parent\text{-}node\ nx' = sourcenode\ ax'$
show $x \notin Def_{SDG}\ nx'$
proof
assume $x \in Def_{SDG}\ nx'$
from $\langle parent\text{-}node\ n' = sourcenode\ ax'\rangle$ $\langle parent\text{-}node\ nx' = sourcenode\ ax'\rangle$
have $parent\text{-}node\ nx' = parent\text{-}node\ n'$ **by** $simp$
with $\langle x \in Def_{SDG}\ nx'\rangle$ $\langle x \in Use_{SDG}\ n''\rangle$ $all\ path$
have $nx'\ influences\ x\ in\ n''$ **by**$(fastforce\ simp{:}data\text{-}dependence\text{-}def)$
hence $nx'\ s{-}x\rightarrow_{dd} n''$ **by**$(rule\ sum\text{-}SDG\text{-}ddep\text{-}edge)$
with $\langle n'' \in HRB\text{-}slice\ S\rangle$ **have** $nx' \in HRB\text{-}slice\ S$
by$(fastforce\ elim{:}combine\text{-}SDG\text{-}slices.cases$
$intro{:}combine\text{-}SDG\text{-}slices.intros\ ddep\text{-}slice1\ ddep\text{-}slice2$
$simp{:}HRB\text{-}slice\text{-}def)$
hence $CFG\text{-}node\ (parent\text{-}node\ nx') \in HRB\text{-}slice\ S$
by$(rule\ valid\text{-}SDG\text{-}node\text{-}in\text{-}slice\text{-}parent\text{-}node\text{-}in\text{-}slice)$

```
        with ⟨sourcenode ax' ∉ ⌊HRB-slice S⌋CFG⟩ ⟨parent-node n' = sourcenode
ax'⟩
          ⟨parent-node nx' = sourcenode ax'⟩ show False
          by(simp add:SDG-to-CFG-set-def)
       qed
     qed
   qed
   with all show ?case by(auto simp add:sourcenodes-def)
 qed
 with ⟨∀ ni. valid-SDG-node ni ∧ parent-node ni ∈ set (sourcenodes as'')
          ⟶ x ∉ DefSDG ni⟩
 have all:∀ ni. valid-SDG-node ni ∧ parent-node ni ∈ set (sourcenodes (asx@as''))

                ⟶ x ∉ DefSDG ni
   by(auto simp:sourcenodes-def)
 with ⟨parent-node n' −asx→ι* parent-node nx⟩
   ⟨parent-node nx −as''→ι* parent-node n''⟩
 have parent-node n' −asx@as''→ι* parent-node n''
   by(fastforce intro:path-Append simp:intra-path-def)
 from this ⟨n'' ∈ HRB-slice S⟩ ⟨x ∈ UseSDG n''⟩ all
 show x ∈ rv S n' by(rule rvI)
qed


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
 show rv S n ⊆ rv S n'
 proof
   fix x assume x ∈ rv S n
   with obs-eq show x ∈ rv S n' by(rule eq-obs-intra-in-rv)
 qed
next
 show rv S n' ⊆ rv S n
 proof
   fix x assume x ∈ rv S n'
   with obs-eq[THEN sym] show x ∈ rv S n by(rule eq-obs-intra-in-rv)
 qed
qed


lemma closed-eq-obs-eq-rvs':
 fixes S :: 'node SDG-node set
 assumes obs-eq:obs-intra m ⌊HRB-slice S⌋CFG = obs-intra m' ⌊HRB-slice S⌋CFG
 shows rv S (CFG-node m) = rv S (CFG-node m')
```

proof
show *rv S* (*CFG-node m*) ⊆ *rv S* (*CFG-node m′*)
proof
fix *x* **assume** *x* ∈ *rv S* (*CFG-node m*)
with *obs-eq* **show** *x* ∈ *rv S* (*CFG-node m′*)
by −(*rule eq-obs-intra-in-rv*,*auto*)
qed
next
show *rv S* (*CFG-node m′*) ⊆ *rv S* (*CFG-node m*)
proof
fix *x* **assume** *x* ∈ *rv S* (*CFG-node m′*)
with *obs-eq*[*THEN sym*] **show** *x* ∈ *rv S* (*CFG-node m*)
by −(*rule eq-obs-intra-in-rv*,*auto*)
qed
qed

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 −
from ‹*valid-edge a*› ‹*valid-edge ax*› ‹*sourcenode a* = *sourcenode ax*›
‹*targetnode a* ≠ *targetnode ax*› ‹*intra-kind* (*kind a*)› ‹*intra-kind* (*kind ax*)›
obtain *Q Q′* **where** *kind a* = $(Q)_\surd$ **and** *kind ax* = $(Q')_\surd$
and ∀ *s*. (*Q s* ⟶ ¬ *Q′ s*) ∧ (*Q′ s* ⟶ ¬ *Q s*)
by(*auto dest*:*deterministic*)
from ‹*valid-edge a*› ‹*valid-edge ax*› ‹*sourcenode a* = *sourcenode ax*›
‹*targetnode a* ≠ *targetnode ax*› ‹*intra-kind* (*kind a*)› ‹*intra-kind* (*kind ax*)›
obtain *P P′* **where** *slice-kind S a* = $(P)_\surd$
and *slice-kind S ax* = $(P')_\surd$
and ∀ *s*. (*P s* ⟶ ¬ *P′ s*) ∧ (*P′ s* ⟶ ¬ *P s*)
by −(*erule slice-deterministic*,*auto*)
show *?thesis*
proof(*cases sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$)
case *True*
with ‹*intra-kind* (*kind a*)›
have *slice-kind S a* = *kind a* **by** −(*rule slice-intra-kind-in-slice*)
with ‹*preds* (*slice-kinds S* (*a*#*as*)) *s*› ‹*kind a* = $(Q)_\surd$›
‹*slice-kind S a* = $(P)_\surd$› **have** *pred* (*kind a*) *s*
by(*simp add*:*slice-kinds-def*)
from *True* ‹*sourcenode a* = *sourcenode ax*› ‹*intra-kind* (*kind ax*)›
have *slice-kind S ax* = *kind ax*
by(*fastforce intro*:*slice-intra-kind-in-slice*)

with ‹*preds* (*slice-kinds S* (*ax*#*asx*)) *s'*› ‹*kind ax* = $(Q')_{\surd}$›
‹*slice-kind S ax* = $(P')_{\surd}$› **have** *pred* (*kind ax*) *s'*
by(*simp add*:*slice-kinds-def*)
with ‹*kind ax* = $(Q')_{\surd}$› **have** *Q'* (*fst* (*hd s'*)) **by**(*cases s'*) *auto*
from ‹*valid-edge a*› **have** *sourcenode a* $-[]\rightarrow_\iota *$ *sourcenode a*
by(*fastforce intro*:*empty-path simp*:*intra-path-def*)
with *True* ‹*valid-edge a*›
have ∀ *V* ∈ *Use* (*sourcenode a*). *V* ∈ *rv S* (*CFG-node* (*sourcenode a*))
by(*auto intro*!:*rvI CFG-Use-SDG-Use simp*:*sourcenodes-def SDG-to-CFG-set-def*)
with ‹∀ *V*∈*rv S* (*CFG-node* (*sourcenode a*)). *state-val s V* = *state-val s' V*›
have ∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s' V* **by** *blast*
with ‹*valid-edge a*› ‹*pred* (*kind a*) *s*› ‹*pred* (*kind ax*) *s'*› ‹*length s* = *length s'*›
‹*snd* (*hd s*) = *snd* (*hd s'*)›
have *pred* (*kind a*) *s'* **by**(*auto intro*:*CFG-edge-Uses-pred-equal*)
with ‹*kind a* = $(Q)_{\surd}$› **have** *Q* (*fst* (*hd s'*)) **by**(*cases s'*) *auto*
with ‹*Q'* (*fst* (*hd s'*))› ‹∀ *s*. (*Q s* ⟶ ¬ *Q' s*) ∧ (*Q' s* ⟶ ¬ *Q s*)›
have *False* **by** *simp*
thus *?thesis* **by** *simp*
next
case *False*
with ‹*kind a* = $(Q)_{\surd}$› ‹*slice-kind S a* = $(P)_{\surd}$› ‹*valid-edge a*›
have *P* = (λ*s*. *False*) ∨ *P* = (λ*s*. *True*)
by(*fastforce elim*:*kind-Predicate-notin-slice-slice-kind-Predicate*)
with ‹*slice-kind S a* = $(P)_{\surd}$›
‹*preds* (*slice-kinds S* (*a*#*as*)) *s*›
have *P* = (λ*s*. *True*) **by**(*cases s*)(*auto simp*:*slice-kinds-def*)
from ‹*sourcenode a* = *sourcenode ax*› *False*
have *sourcenode ax* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **by** *simp*
with ‹*kind ax* = $(Q')_{\surd}$› ‹*slice-kind S ax* = $(P')_{\surd}$› ‹*valid-edge ax*›
have *P'* = (λ*s*. *False*) ∨ *P'* = (λ*s*. *True*)
by(*fastforce elim*:*kind-Predicate-notin-slice-slice-kind-Predicate*)
with ‹*slice-kind S ax* = $(P')_{\surd}$›
‹*preds* (*slice-kinds S* (*ax*#*asx*)) *s'*›
have *P'* = (λ*s*. *True*) **by**(*cases s'*)(*auto simp*:*slice-kinds-def*)
with ‹*P* = (λ*s*. *True*)› ‹∀ *s*. (*P s* ⟶ ¬ *P' s*) ∧ (*P' s* ⟶ ¬ *P s*)›
have *False* **by** *blast*
thus *?thesis* **by** *simp*
qed
qed

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

fix V **assume** $V \in rv\ S\ (CFG\text{-}node\ (targetnode\ a))$
from ‹$preds\ (slice\text{-}kinds\ S\ (a\#as))\ s$›
have $s \neq []$ **by**(*cases s*,*auto simp*:*slice-kinds-def*)
from ‹$preds\ (slice\text{-}kinds\ S\ (a\#asx))\ s'$›
have $s' \neq []$ **by**(*cases s'*,*auto simp*:*slice-kinds-def*)
show $state\text{-}val\ (transfer\ (slice\text{-}kind\ S\ a)\ s)\ V =$
$state\text{-}val\ (transfer\ (slice\text{-}kind\ S\ a)\ s')\ V$
proof(*cases* $V \in Def\ (sourcenode\ a)$)
case *True*
show *?thesis*
proof(*cases sourcenode* $a \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$)
case *True*
with ‹$intra\text{-}kind\ (kind\ a)$› **have** $slice\text{-}kind\ S\ a = kind\ a$
by −(*rule slice-intra-kind-in-slice*)
with ‹$preds\ (slice\text{-}kinds\ S\ (a\#as))\ s$› **have** $pred\ (kind\ a)\ s$
by(*simp add*:*slice-kinds-def*)
from ‹$slice\text{-}kind\ S\ a = kind\ a$›
‹$preds\ (slice\text{-}kinds\ S\ (a\#asx))\ s'$›
have $pred\ (kind\ a)\ s'$ **by**(*simp add*:*slice-kinds-def*)
from ‹$valid\text{-}edge\ a$› **have** $sourcenode\ a\ -[]\rightarrow_\iota *\ sourcenode\ a$
by(*fastforce intro*:*empty-path simp*:*intra-path-def*)
with *True* ‹$valid\text{-}edge\ a$›
have $\forall\ V \in Use\ (sourcenode\ a).\ V \in rv\ S\ (CFG\text{-}node\ (sourcenode\ a))$
by(*auto intro*!:*rvI CFG-Use-SDG-Use simp*:*sourcenodes-def SDG-to-CFG-set-def*)
with ‹$\forall\ V \in rv\ S\ (CFG\text{-}node\ (sourcenode\ a)).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$›
have $\forall\ V \in Use\ (sourcenode\ a).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$ **by** *blast*
from ‹$valid\text{-}edge\ a$› *this* ‹$pred\ (kind\ a)\ s$› ‹$pred\ (kind\ a)\ s'$›
‹$intra\text{-}kind\ (kind\ a)$›
have $\forall\ V \in Def\ (sourcenode\ a).$
$state\text{-}val\ (transfer\ (kind\ a)\ s)\ V = state\text{-}val\ (transfer\ (kind\ a)\ s')\ V$
by −(*rule CFG-intra-edge-transfer-uses-only-Use*,*auto*)
with ‹$V \in Def\ (sourcenode\ a)$› ‹$slice\text{-}kind\ S\ a = kind\ a$›
show *?thesis* **by** *simp*
next
case *False*
from ‹$V \in rv\ S\ (CFG\text{-}node\ (targetnode\ a))$›
obtain *xs nx* **where** $targetnode\ a\ -xs\rightarrow_\iota *\ parent\text{-}node\ nx$
and $nx \in HRB\text{-}slice\ S$ **and** $V \in Use_{SDG}\ nx$
and $\forall\ n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge parent\text{-}node\ n'' \in set\ (sourcenodes\ xs)$
$\longrightarrow V \notin Def_{SDG}\ n''$ **by**(*fastforce elim*:*rvE*)
from ‹$valid\text{-}edge\ a$› **have** $valid\text{-}node\ (sourcenode\ a)$ **by** *simp*
from ‹$valid\text{-}edge\ a$› ‹$targetnode\ a\ -xs\rightarrow_\iota *\ parent\text{-}node\ nx$› ‹$intra\text{-}kind\ (kind\ a)$›
have $sourcenode\ a\ -a\#xs\rightarrow_\iota *\ parent\text{-}node\ nx$
by(*fastforce intro*:*path.Cons-path simp*:*intra-path-def*)
with ‹$V \in Def\ (sourcenode\ a)$› ‹$V \in Use_{SDG}\ nx$› ‹$valid\text{-}node\ (sourcenode\ a)$›
‹$\forall\ n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge parent\text{-}node\ n'' \in set\ (sourcenodes\ xs)$
$\longrightarrow V \notin Def_{SDG}\ n''$›

```
    have (CFG-node (sourcenode a)) influences V in nx
      by(fastforce intro:CFG-Def-SDG-Def simp:data-dependence-def)
    hence (CFG-node (sourcenode a)) s−V→dd nx by(rule sum-SDG-ddep-edge)
    from ⟨nx ∈ HRB-slice S⟩ ⟨(CFG-node (sourcenode a)) s−V→dd nx⟩
    have CFG-node (sourcenode a) ∈ HRB-slice S
    proof(induct rule:HRB-slice-cases)
      case (phase1 n nx')
      with ⟨(CFG-node (sourcenode a)) s−V→dd nx⟩ show ?case
        by(fastforce intro:intro:ddep-slice1 combine-SDG-slices.combSlice-refl
                  simp:HRB-slice-def)
    next
      case (phase2 nx' n' n'' p n)
      from ⟨(CFG-node (sourcenode a)) s−V→dd n⟩ ⟨n ∈ sum-SDG-slice2 n'⟩
      have CFG-node (sourcenode a) ∈ sum-SDG-slice2 n' by(rule ddep-slice2)
      with phase2 show ?thesis
        by(fastforce intro:combine-SDG-slices.combSlice-Return-parent-node
                  simp:HRB-slice-def)
    qed
    with False have False by(simp add:SDG-to-CFG-set-def)
    thus ?thesis by simp
  qed
next
  case False
  from ⟨V ∈ rv S (CFG-node (targetnode a))⟩
  obtain xs nx where targetnode a −xs→ι* parent-node nx
    and nx ∈ HRB-slice S and V ∈ Use_SDG nx
    and all:∀ n''. valid-SDG-node n'' ∧ parent-node n'' ∈ set (sourcenodes xs)
            ⟶ V ∉ Def_SDG n'' by(fastforce elim:rvE)
  from ⟨valid-edge a⟩ have valid-node (sourcenode a) by simp
  from ⟨valid-edge a⟩ ⟨targetnode a −xs→ι* parent-node nx⟩ ⟨intra-kind (kind a)⟩
  have sourcenode a −a#xs →ι* parent-node nx
    by(fastforce intro:path.Cons-path simp:intra-path-def)
  from False all
  have ∀ n''. valid-SDG-node n'' ∧ parent-node n'' ∈ set (sourcenodes (a#xs))
            ⟶ V ∉ Def_SDG n''
    by(fastforce dest:SDG-Def-parent-Def simp:sourcenodes-def)
  with ⟨sourcenode a −a#xs →ι* parent-node nx⟩ ⟨nx ∈ HRB-slice S⟩
    ⟨V ∈ Use_SDG nx⟩
  have V ∈ rv S (CFG-node (sourcenode a)) by(fastforce intro:rvI)
  from ⟨intra-kind (kind a)⟩ show ?thesis
  proof(cases kind a)
    case(UpdateEdge f)
    show ?thesis
    proof(cases sourcenode a ∈ ⌊HRB-slice S⌋_CFG)
      case True
      with ⟨intra-kind (kind a)⟩ have slice-kind S a = kind a
        by(fastforce intro:slice-intra-kind-in-slice)
      from UpdateEdge ⟨s ≠ []⟩ have pred (kind a) s by(cases s) auto
      with ⟨valid-edge a⟩ ⟨V ∉ Def (sourcenode a)⟩ ⟨intra-kind (kind a)⟩
```

have *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*
by(*fastforce intro*:*CFG-intra-edge-no-Def-equal*)
from *UpdateEdge* ‹*s'* ≠ []› **have** *pred* (*kind a*) *s'* **by**(*cases s'*) *auto*
with ‹*valid-edge a*› ‹*V* ∉ *Def* (*sourcenode a*)› ‹*intra-kind* (*kind a*)›
have *state-val* (*transfer* (*kind a*) *s'*) *V* = *state-val s'* *V*
by(*fastforce intro*:*CFG-intra-edge-no-Def-equal*)
with ‹∀ *V*∈*rv S* (*CFG-node* (*sourcenode a*)). *state-val s V* = *state-val s'* *V*›
‹*state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*›
‹*V* ∈ *rv S* (*CFG-node* (*sourcenode a*))› ‹*slice-kind S a* = *kind a*›
show *?thesis* **by** *fastforce*
next
case *False*
with *UpdateEdge* **have** *slice-kind S a* = ⇑*id*
by −(*rule slice-kind-Upd*)
with ‹∀ *V*∈*rv S* (*CFG-node* (*sourcenode a*)). *state-val s V* = *state-val s'* *V*›
‹*V* ∈ *rv S* (*CFG-node* (*sourcenode a*))› ‹*s* ≠ []› ‹*s'* ≠ []›
show *?thesis* **by**(*cases s*,*auto*,*cases s'*,*auto*)
qed
next
case (*PredicateEdge Q*)
show *?thesis*
proof(*cases sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$)
case *True*
with *PredicateEdge* ‹*intra-kind* (*kind a*)›
have *slice-kind S a* = (*Q*)$_{\surd}$
by(*simp add*:*slice-intra-kind-in-slice*)
with ‹∀ *V*∈*rv S* (*CFG-node* (*sourcenode a*)). *state-val s V* = *state-val s'* *V*›
‹*V* ∈ *rv S* (*CFG-node* (*sourcenode a*))› ‹*s* ≠ []› ‹*s'* ≠ []›
show *?thesis* **by**(*cases s*,*auto*,*cases s'*,*auto*)
next
case *False*
with *PredicateEdge* ‹*valid-edge a*›
obtain *Q'* **where** *slice-kind S a* = (*Q'*)$_{\surd}$
by −(*erule kind-Predicate-notin-slice-slice-kind-Predicate*)
with‹∀ *V*∈*rv S* (*CFG-node* (*sourcenode a*)). *state-val s V* = *state-val s'* *V*›
‹*V* ∈ *rv S* (*CFG-node* (*sourcenode a*))› ‹*s* ≠ []› ‹*s'* ≠ []›
show *?thesis* **by**(*cases s*,*auto*,*cases s'*,*auto*)
qed
qed (*auto simp*:*intra-kind-def*)
qed
qed

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′ ∈ ⌊HRB-slice S⌋CFG*;
msx ≠ [] ⟶ (∃ *mx′*. *call-of-return-node mx mx′* ∧ *mx′ ∉ ⌊HRB-slice S⌋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 ⌊HRB-slice S⌋CFG = obs ms′ ⌊HRB-slice S⌋CFG⟧
⟹ *((ms,s),(ms′,s′)) ∈ WS S*

lemma *WS-silent-move*:
assumes *S,kind* ⊢ *(ms₁,s₁) −a→τ (ms₁′,s₁′)* **and** *((ms₁,s₁),(ms₂,s₂)) ∈ WS S*
shows *((ms₁′,s₁′),(ms₂,s₂)) ∈ WS S*
proof −
from ‹*((ms₁,s₁),(ms₂,s₂)) ∈ WS S*› **obtain** *msx mx*
where *WSE*:∀ *m ∈ set ms₁*. *valid-node m* ∀ *m ∈ set ms₂*. *valid-node m*
length ms₁ = length s₁ length ms₂ = length s₂ s₁ ≠ [] s₂ ≠ []
ms₁ = msx@mx#tl ms₂ get-proc mx = get-proc (hd ms₂)
∀ *m ∈ set (tl ms₂)*. ∃ *m′*. *call-of-return-node m m′* ∧ *m′ ∈ ⌊HRB-slice S⌋CFG*
msx ≠ [] ⟶ (∃ *mx′*. *call-of-return-node mx mx′* ∧ *mx′ ∉ ⌊HRB-slice S⌋CFG*)
∀ *m ∈ set (tl ms₁)*. *return-node m*
∀ *i < length ms₂*. *snd (s₁!(length msx + i)) = snd (s₂!i)*
∀ *i < length ms₂*. ∀ *V ∈ rv S (CFG-node ((mx#tl ms₂)!i))*.
(fst (s₁!(length msx + i))) V = (fst (s₂!i)) V
obs ms₁ ⌊HRB-slice S⌋CFG = obs ms₂ ⌊HRB-slice S⌋CFG
by(*fastforce elim*:*WS.cases*)
{ assume ∀ *m ∈ set (tl ms₁′)*. *return-node m*
have *obs ms₁′ ⌊HRB-slice S⌋CFG = obs ms₂ ⌊HRB-slice S⌋CFG*
proof(*cases obs ms₁′ ⌊HRB-slice S⌋CFG = {}*)
case *True*
with ‹*S,kind* ⊢ *(ms₁,s₁) −a→τ (ms₁′,s₁′)*› **have** *obs ms₁ ⌊HRB-slice S⌋CFG = {}*
by(*rule silent-move-empty-obs-slice*)
with ‹*obs ms₁ ⌊HRB-slice S⌋CFG = obs ms₂ ⌊HRB-slice S⌋CFG*›
‹*obs ms₁′ ⌊HRB-slice S⌋CFG = {}*›
show *?thesis* **by** *simp*
next
case *False*
from *this* ‹∀ *m ∈ set (tl ms₁′)*. *return-node m*›
obtain *ms′* **where** *obs ms₁′ ⌊HRB-slice S⌋CFG = {ms′}*
by(*fastforce dest*:*obs-singleton-element*)
hence *ms′ ∈ obs ms₁′ ⌊HRB-slice S⌋CFG* **by** *fastforce*
from ‹*S,kind* ⊢ *(ms₁,s₁) −a→τ (ms₁′,s₁′)*› ‹*ms′ ∈ obs ms₁′ ⌊HRB-slice S⌋CFG*›
‹∀ *m ∈ set (tl ms₁′)*. *return-node m*›
have *ms′ ∈ obs ms₁ ⌊HRB-slice S⌋CFG* **by**(*fastforce intro*:*silent-move-obs-slice*)

from *this* ‹∀ *m* ∈ *set* (*tl ms*$_1$). *return-node m*›
have *obs ms*$_1$ ⌊*HRB-slice S*⌋$_{CFG}$ = {*ms'*} **by**(*rule obs-singleton-element*)
with ‹*obs ms*$_1$' ⌊*HRB-slice S*⌋$_{CFG}$ = {*ms'*}›
‹*obs ms*$_1$ ⌊*HRB-slice S*⌋$_{CFG}$ = *obs ms*$_2$ ⌊*HRB-slice S*⌋$_{CFG}$›
show *?thesis* **by** *simp*
qed }
with ‹*S*,*kind* ⊢ (*ms*$_1$,*s*$_1$) −*a*→$_\tau$ (*ms*$_1$',*s*$_1$')› *WSE*
show *?thesis*
proof(*induct S f*≡*kind ms*$_1$ *s*$_1$ *a ms*$_1$' *s*$_1$' *rule*:*silent-move.induct*)
case (*silent-move-intra a s*$_1$ *s*$_1$' *ms*$_1$ *S ms*$_1$')
note *obs-eq* = ‹∀ *a*∈*set* (*tl ms*$_1$'). *return-node a* ⟹
obs ms$_1$' ⌊*HRB-slice S*⌋$_{CFG}$ = *obs ms*$_2$ ⌊*HRB-slice S*⌋$_{CFG}$›
from ‹*s*$_1$ ≠ []› ‹*s*$_2$ ≠ []› **obtain** *cf*$_1$ *cfs*$_1$ *cf*$_2$ *cfs*$_2$ **where** [*simp*]:*s*$_1$ = *cf*$_1$#*cfs*$_1$
and [*simp*]:*s*$_2$ = *cf*$_2$#*cfs*$_2$ **by**(*cases s*$_1$,*auto*,*cases s*$_2$,*fastforce*+)
from ‹*transfer* (*kind a*) *s*$_1$ = *s*$_1$'› ‹*intra-kind* (*kind a*)›
obtain *cf*$_1$' **where** [*simp*]:*s*$_1$' = *cf*$_1$'#*cfs*$_1$
by(*cases cf*$_1$,*cases kind a*,*auto simp*:*intra-kind-def*)
from ‹∀ *m* ∈ *set ms*$_1$. *valid-node m*› ‹*ms*$_1$' = *targetnode a* # *tl ms*$_1$› ‹*valid-edge a*›
have ∀ *m* ∈ *set ms*$_1$'. *valid-node m* **by**(*cases ms*$_1$) *auto*
from ‹*length ms*$_1$ = *length s*$_1$› ‹*length s*$_1$' = *length s*$_1$›
‹*ms*$_1$' = *targetnode a* # *tl ms*$_1$›
have *length ms*$_1$' = *length s*$_1$' **by**(*cases ms*$_1$) *auto*
from ‹∀ *m* ∈ *set* (*tl ms*$_1$). *return-node m*› ‹*ms*$_1$' = *targetnode a* # *tl ms*$_1$›
have ∀ *m* ∈ *set* (*tl ms*$_1$'). *return-node m* **by** *simp*
from *obs-eq*[*OF this*] **have** *obs ms*$_1$' ⌊*HRB-slice S*⌋$_{CFG}$ = *obs ms*$_2$ ⌊*HRB-slice S*⌋$_{CFG}$.
from ‹∀ *i* < *length ms*$_2$. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl ms*$_2$)!*i*)).
(*fst* (*s*$_1$!(*length msx* + *i*))) *V* = (*fst* (*s*$_2$!*i*)) *V*› ‹*length ms*$_2$ = *length s*$_2$›
have ∀ *V*∈*rv S* (*CFG-node mx*). (*fst* (*s*$_1$! *length msx*)) *V* = *state-val s*$_2$ *V*
by(*cases ms*$_2$) *auto*
show *?case*
proof(*cases msx*)
case *Nil*
with ‹*ms*$_1$ = *msx*@*mx*#*tl ms*$_2$› ‹*hd ms*$_1$ = *sourcenode a*›
have [*simp*]:*mx* = *sourcenode a* **and** [*simp*]:*tl ms*$_1$ = *tl ms*$_2$ **by** *simp-all*
from ‹∀ *m*∈*set* (*tl ms*$_2$). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹(∃ *m*∈*set* (*tl ms*$_1$). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$)
∨
hd ms$_1$ ∉ ⌊*HRB-slice S*⌋$_{CFG}$›
have *hd ms*$_1$ ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **by** *fastforce*
with ‹*hd ms*$_1$ = *sourcenode a*› **have** *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **by** *simp*
from ‹*ms*$_1$' = *targetnode a* # *tl ms*$_1$› **have** *ms*$_1$' = [] @ *targetnode a* # *tl ms*$_2$
by *simp*
from ‹*valid-edge a*› ‹*intra-kind*(*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
with ‹*get-proc mx* = *get-proc* (*hd ms*$_2$)›

have *get-proc* (*targetnode a*) = *get-proc* (*hd* ms_2) **by** *simp*
from ‹*transfer* (*kind a*) $s_1 = s_1{}'$› ‹*intra-kind* (*kind a*)›
have *snd* $cf_1{}'$ = *snd* cf_1 **by**(*auto simp*:*intra-kind-def*)
with ‹$\forall\, i$<*length* ms_2. *snd* (s_1 ! (*length msx* + i)) = *snd* (s_2 ! i)› *Nil*
have $\forall\, i$<*length* ms_2. *snd* ($s_1{}'$! i) = *snd* (s_2 ! i)
by *auto*(*case-tac i*,*auto*)
have $\forall\, V \in$ *rv S* (*CFG-node* (*targetnode a*)). *fst* $cf_1{}'$ V = *fst* cf_2 V
proof
fix V **assume** $V \in$ *rv S* (*CFG-node* (*targetnode a*))
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)› ‹*sourcenode a* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
have *obs-intra* (*targetnode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ =
obs-intra (*sourcenode a*) $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
by(*rule edge-obs-intra-slice-eq*)
hence *rv S* (*CFG-node* (*targetnode a*)) = *rv S* (*CFG-node* (*sourcenode a*))
by(*rule closed-eq-obs-eq-rvs'*)
with ‹$V \in$ *rv S* (*CFG-node* (*targetnode a*))›
have $V \in$ *rv S* (*CFG-node* (*sourcenode a*)) **by** *simp*
then obtain *as* n' **where** *sourcenode a* $-as\rightarrow_\iota *$ *parent-node* n'
and $n' \in$ *HRB-slice S* **and** $V \in$ Use_{SDG} n'
and $\forall\, n''$. *valid-SDG-node* $n'' \wedge$ *parent-node* $n'' \in$ *set* (*sourcenodes as*)
$\longrightarrow$ $V \notin$ Def_{SDG} n''
by(*fastforce elim*:*rvE*)
with ‹*sourcenode a* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*valid-edge a*›
have $V \notin$ Def_{SDG} (*CFG-node* (*sourcenode a*))
apply(*clarsimp simp*:*intra-path-def*)
apply(*erule path.cases*)
by(*auto dest*:*valid-SDG-node-in-slice-parent-node-in-slice*
simp:*sourcenodes-def SDG-to-CFG-set-def*)
from ‹*valid-edge a*› **have** *valid-node* (*sourcenode a*) **by** *simp*
with ‹$V \notin$ Def_{SDG} (*CFG-node* (*sourcenode a*))› **have** $V \notin$ *Def* (*sourcenode a*)
by(*fastforce intro*:*CFG-Def-SDG-Def valid-SDG-CFG-node*)
with ‹*valid-edge a*› ‹*intra-kind* (*kind a*)› ‹*pred* (*kind a*) s_1›
have *state-val* (*transfer* (*kind a*) s_1) V = *state-val* s_1 V
by(*fastforce intro*:*CFG-intra-edge-no-Def-equal*)
with ‹*transfer* (*kind a*) $s_1 = s_1{}'$› **have** *fst* $cf_1{}'$ V = *fst* cf_1 V **by** *simp*
from ‹$V \in$ *rv S* (*CFG-node* (*sourcenode a*))› ‹*msx* = []›
‹$\forall\, V$∈*rv S* (*CFG-node mx*). (*fst* (s_1 ! *length msx*)) V = *state-val* s_2 V›
have *fst* cf_1 V = *fst* cf_2 V **by** *simp*
with ‹*fst* $cf_1{}'$ V = *fst* cf_1 V› **show** *fst* $cf_1{}'$ V = *fst* cf_2 V **by** *simp*
qed
with ‹$\forall\, i$<*length* ms_2. $\forall\, V$∈*rv S* (*CFG-node* ((*mx* # *tl* ms_2) ! i)).
(*fst* (s_1 ! (*length msx* + i))) V = (*fst* (s_2 ! i)) V› *Nil*
have $\forall\, i$<*length* ms_2. $\forall\, V$∈*rv S* (*CFG-node* ((*targetnode a* # *tl* ms_2) ! i)).
(*fst* ($s_1{}'$! (*length* [] + i))) V = (*fst* (s_2 ! i)) V
by *auto* (*case-tac i*,*auto*)
with ‹$\forall\, m \in$ *set* $ms_1{}'$. *valid-node m*› ‹$\forall\, m \in$ *set* ms_2. *valid-node m*›
‹*length* $ms_1{}'$ = *length* $s_1{}'$› ‹*length* ms_2 = *length* s_2›
‹$ms_1{}'$ = [] @ *targetnode a* # *tl* ms_2›

⟨*get-proc* (*targetnode a*) = *get-proc* (*hd* ms_2)⟩
⟨∀ *m* ∈ *set* (*tl* ms_2). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$⟩
⟨∀ *m* ∈ *set* (*tl* ms_1). *return-node m*⟩
⟨*obs* ms_1' ⌊*HRB-slice S*⌋$_{CFG}$ = *obs* ms_2 ⌊*HRB-slice S*⌋$_{CFG}$⟩
⟨∀ *i*<*length* ms_2. *snd* (s_1' ! *i*) = *snd* (s_2 ! *i*)⟩
show *?thesis* **by**(*auto intro*!:*WSI*)
next
case (*Cons mx' msx'*)
with ⟨ms_1 = *msx*@*mx*#*tl* ms_2⟩ ⟨*hd* ms_1 = *sourcenode a*⟩
have [*simp*]:*mx'* = *sourcenode a* **and** [*simp*]:*tl* ms_1 = *msx'*@*mx*#*tl* ms_2
by *simp-all*
from ⟨ms_1' = *targetnode a* # *tl* ms_1⟩ **have** ms_1' = ((*targetnode a*)#*msx'*)@*mx*#*tl* ms_2
by *simp*
from ⟨∀ *V*∈*rv S* (*CFG-node mx*). (*fst* (s_1 ! *length msx*)) *V* = *state-val* s_2 *V*⟩ *Cons*
have *rv*:∀ *V*∈*rv S* (*CFG-node mx*).
(*fst* (s_1' ! *length* (*targetnode a*#*msx'*))) *V* = *state-val* s_2 *V* **by** *fastforce*
from ⟨ms_1 = *msx*@*mx*#*tl* ms_2⟩ *Cons* ⟨ms_1' = *targetnode a* # *tl* ms_1⟩
have ms_1' = ((*targetnode a*)#*msx'*)@*mx*#*tl* ms_2 **by** *simp*
from ⟨∀ *i*<*length* ms_2. *snd* (s_1 ! (*length msx* + *i*)) = *snd* (s_2 ! *i*)⟩ *Cons*
have ∀ *i*<*length* ms_2. *snd* (s_1' ! (*length msx* + *i*)) = *snd* (s_2 ! *i*) **by** *fastforce*
from ⟨∀ *V*∈*rv S* (*CFG-node mx*). (*fst* (s_1 ! *length msx*)) *V* = *state-val* s_2 *V*⟩ *Cons*
have ∀ *V*∈*rv S* (*CFG-node mx*). (*fst* (s_1' ! *length msx*)) *V* = *state-val* s_2 *V*
by *simp*
with ⟨∀ *i* < *length* ms_2. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl* ms_2)!*i*)).
(*fst* (s_1!(*length msx* + *i*))) *V* = (*fst* (s_2!*i*)) *V*⟩ *Cons*
have ∀ *i*<*length* ms_2. ∀ *V*∈*rv S* (*CFG-node* ((*mx* # *tl* ms_2)!*i*)).
(*fst* (s_1'!(*length* (*targetnode a* # *msx'*) + *i*))) *V* = (*fst* (s_2!*i*)) *V*
by *clarsimp*
with ⟨∀ *m*∈*set* ms_1'. *valid-node m*⟩ ⟨∀ *m*∈*set* ms_2. *valid-node m*⟩
⟨*length* ms_1' = *length* s_1'⟩ ⟨*length* ms_2 = *length* s_2⟩
⟨ms_1' = ((*targetnode a*)#*msx'*)@*mx*#*tl* ms_2⟩
⟨∀ *m*∈*set* (*tl* ms_2). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$⟩
⟨∀ *m* ∈ *set* (*tl* ms_1'). *return-node m*⟩ ⟨*get-proc mx* = *get-proc* (*hd* ms_2)⟩
⟨*msx* ≠ [] ⟶ (∃ *mx'*. *call-of-return-node mx mx'* ∧ *mx'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$)⟩
⟨*obs* ms_1' ⌊*HRB-slice S*⌋$_{CFG}$ = *obs* ms_2 ⌊*HRB-slice S*⌋$_{CFG}$⟩ *Cons*
⟨∀ *i*<*length* ms_2. *snd* (s_1' ! (*length msx* + *i*)) = *snd* (s_2 ! *i*)⟩
show *?thesis* **by** −(*rule WSI*,*clarsimp*+,*fastforce*,*clarsimp*+)
qed
next
case (*silent-move-call a* s_1 s_1' *Q r p fs a'* ms_1 *S* ms_1')
note *obs-eq* = ⟨∀ *a*∈*set* (*tl* ms_1'). *return-node a* ⟹
obs ms_1' ⌊*HRB-slice S*⌋$_{CFG}$ = *obs* ms_2 ⌊*HRB-slice S*⌋$_{CFG}$⟩
from ⟨s_1 ≠ []⟩ ⟨s_2 ≠ []⟩ **obtain** cf_1 cfs_1 cf_2 cfs_2 **where** [*simp*]:s_1 = cf_1#cfs_1
and [*simp*]:s_2 = cf_2#cfs_2 **by**(*cases* s_1,*auto*,*cases* s_2,*fastforce*+)

from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
obtain *ins outs* **where** (*p*,*ins*,*outs*) ∈ *set procs*
by(*fastforce dest*!:*callee-in-procs*)
with ‹*transfer* (*kind a*) s_1 = s_1'› ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
have [*simp*]:s_1' = (*Map.empty*(*ins* [:=] *params fs* (*fst* cf_1)), *r*) # cf_1 # cfs_1
by *simp*(*unfold formal-in-THE*,*simp*)
from ‹*length* ms_1 = *length* s_1› ‹ms_1' = *targetnode a* # *targetnode a'* # *tl* ms_1›
have *length* ms_1' = *length* s_1' **by** *simp*
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
with ‹∀ *m*∈*set* ms_1. *valid-node m*› ‹*valid-edge a*›
‹ms_1' = *targetnode a* # *targetnode a'* # *tl* ms_1›
have ∀ *m*∈*set* ms_1'. *valid-node m* **by**(*cases* ms_1) *auto*
from ‹*valid-edge a'*› ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
have *return-node* (*targetnode a'*) **by**(*fastforce simp*:*return-node-def*)
with ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› ‹*valid-edge a'*›
have *call-of-return-node* (*targetnode a'*) (*sourcenode a*)
by(*simp add*:*call-of-return-node-def*) *blast*
from ‹∀ *m* ∈ *set* (*tl* ms_1). *return-node m*› ‹*return-node* (*targetnode a'*)›
‹ms_1' = *targetnode a* # *targetnode a'* # *tl* ms_1›
have ∀ *m* ∈ *set* (*tl* ms_1'). *return-node m* **by** *simp*
from *obs-eq*[*OF this*] **have** *obs* ms_1' ⌊*HRB-slice S*⌋$_{CFG}$ = *obs* ms_2 ⌊*HRB-slice S*⌋$_{CFG}$.
from ‹∀ *i* < *length* ms_2. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl* ms_2)!*i*)).
(*fst* (s_1!(*length msx* + *i*))) *V* = (*fst* (s_2!*i*)) *V*› ‹*length* ms_2 = *length* s_2›
have ∀ *V*∈*rv S* (*CFG-node mx*). (*fst* (s_1 ! *length msx*)) *V* = *state-val* s_2 *V*
by(*erule-tac x*=*0* **in** *allE*) *auto*
show *?case*
proof(*cases msx*)
case *Nil*
with ‹ms_1 = *msx*@*mx*#*tl* ms_2› ‹*hd* ms_1 = *sourcenode a*›
have [*simp*]:*mx* = *sourcenode a* **and** [*simp*]:*tl* ms_1 = *tl* ms_2 **by** *simp-all*
from ‹∀ *m*∈*set* (*tl* ms_2). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹(∃ *m*∈*set* (*tl* ms_1). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$) ∨
hd ms_1 ∉ ⌊*HRB-slice S*⌋$_{CFG}$›
have *hd* ms_1 ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **by** *fastforce*
with ‹*hd* ms_1 = *sourcenode a*› **have** *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **by** *simp*
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
obtain *a''* **where** *valid-edge a''* **and** *sourcenode a''* = *sourcenode a*
and *targetnode a''* = *targetnode a'* **and** *intra-kind*(*kind a''*)
by −(*drule call-return-node-edge*,*auto simp*:*intra-kind-def*)
from ‹*valid-edge a''*› ‹*intra-kind*(*kind a''*)›
have *get-proc* (*sourcenode a''*) = *get-proc* (*targetnode a''*)
by(*rule get-proc-intra*)
with ‹*sourcenode a''* = *sourcenode a*› ‹*targetnode a''* = *targetnode a'*›
‹*get-proc mx* = *get-proc* (*hd* ms_2)›

have *get-proc* (*targetnode a'*) = *get-proc* (*hd ms*$_2$) **by** *simp*
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*a'* ∈ *get-return-edges a*›
have *CFG-node* (*sourcenode a*) *s*−*p*→$_{sum}$ *CFG-node* (*targetnode a'*)
by(*fastforce intro*:*sum-SDG-call-summary-edge*)
have *targetnode a'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
proof
assume *targetnode a'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$
hence *CFG-node* (*targetnode a'*) ∈ *HRB-slice S* **by**(*simp add*:*SDG-to-CFG-set-def*)
hence *CFG-node* (*sourcenode a*) ∈ *HRB-slice S*
proof(*induct CFG-node* (*targetnode a'*) *rule*:*HRB-slice-cases*)
case (*phase1 nx*)
with ‹*CFG-node* (*sourcenode a*) *s*−*p*→$_{sum}$ *CFG-node* (*targetnode a'*)›
show *?case* **by**(*fastforce intro*:*combine-SDG-slices.combSlice-refl sum-slice1*
simp:*HRB-slice-def*)
next
case (*phase2 nx n' n'' p'*)
from ‹*CFG-node* (*targetnode a'*) ∈ *sum-SDG-slice2 n'*›
‹*CFG-node* (*sourcenode a*) *s*−*p*→$_{sum}$ *CFG-node* (*targetnode a'*)› ‹*valid-edge*
a›
have *CFG-node* (*sourcenode a*) ∈ *sum-SDG-slice2 n'*
by(*fastforce intro*:*sum-slice2*)
with ‹*n'* ∈ *sum-SDG-slice1 nx*› ‹*n''* *s*−*p'*→$_{ret}$ *CFG-node* (*parent-node n'*)›

‹*nx* ∈ *S*›
show *?case*
by(*fastforce intro*:*combine-SDG-slices.combSlice-Return-parent-node*
simp:*HRB-slice-def*)
qed
with ‹*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› **show** *False*
by(*simp add*:*SDG-to-CFG-set-def HRB-slice-def*)
qed
from ‹*ms*$_1$' = *targetnode a* # *targetnode a'* # *tl ms*$_1$›
have *ms*$_1$' = [*targetnode a*] @ *targetnode a'* # *tl ms*$_2$ **by** *simp*
from ‹∀ *i*<*length ms*$_2$. *snd* (*s*$_1$! (*length msx* + *i*)) = *snd* (*s*$_2$! *i*)› *Nil*
have ∀ *i*<*length ms*$_2$. *snd* (*s*$_1$' ! (*length* [*targetnode a*] + *i*)) = *snd* (*s*$_2$! *i*)
by *fastforce*
have ∀ *V*∈*rv S* (*CFG-node* (*targetnode a'*)). (*fst* (*s*$_1$' ! *1*)) *V* = *state-val s*$_2$
V
proof
fix *V* **assume** *V* ∈ *rv S* (*CFG-node* (*targetnode a'*))
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
obtain *a''* **where** *edge*:*valid-edge a''* *sourcenode a''* = *sourcenode a*
targetnode a'' = *targetnode a'* *intra-kind*(*kind a''*)
by −(*drule call-return-node-edge*,*auto simp*:*intra-kind-def*)
from ‹*V* ∈ *rv S* (*CFG-node* (*targetnode a'*))›
obtain *as n'* **where** *targetnode a'* −*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''*

by(*fastforce elim*:*rvE*)
from ‹*targetnode a'* $-as\rightarrow_\iota*$ *parent-node n'*› *edge*
have *sourcenode a* $-a''\#as\rightarrow_\iota*$ *parent-node n'*
by(*fastforce intro*:*Cons-path simp*:*intra-path-def*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:$r\hookrightarrow_p fs$›
have *V* ∉ *Def* (*sourcenode a*)
by(*fastforce dest*:*call-source-Def-empty*)
with ‹∀ *n''*. *valid-SDG-node n''* ∧ *parent-node n''* ∈ *set* (*sourcenodes as*)
⟶ *V* ∉ Def_{SDG} *n''*› ‹*sourcenode a''* = *sourcenode a*›
have ∀ *n''*. *valid-SDG-node n''* ∧ *parent-node n''* ∈ *set* (*sourcenodes* (*a''*#*as*))
⟶ *V* ∉ Def_{SDG} *n''*
by(*fastforce dest*:*SDG-Def-parent-Def simp*:*sourcenodes-def*)
with ‹*sourcenode a* $-a''\#as\rightarrow_\iota*$ *parent-node n'*› ‹*n'* ∈ *HRB-slice S*›
‹*V* ∈ Use_{SDG} *n'*›
have *V* ∈ *rv S* (*CFG-node* (*sourcenode a*)) **by**(*fastforce intro*:*rvI*)
from ‹∀ *V*∈*rv S* (*CFG-node mx*). (*fst* (*s*$_1$! *length msx*)) *V* = *state-val s*$_2$ *V*› *Nil*
have ∀ *V*∈*rv S* (*CFG-node* (*sourcenode a*)). *fst cf*$_1$ *V* = *fst cf*$_2$ *V* **by** *simp*
with ‹*V* ∈ *rv S* (*CFG-node* (*sourcenode a*))› **have** *fst cf*$_1$ *V* = *fst cf*$_2$ *V* **by** *simp*
thus (*fst* (*s*$_1$' ! *1*)) *V* = *state-val s*$_2$ *V* **by** *simp*
qed
with ‹∀ *i* < *length ms*$_2$. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl ms*$_2$)!*i*)).
(*fst* (*s*$_1$!(*length msx* + *i*))) *V* = (*fst* (*s*$_2$!*i*)) *V*› *Nil*
have ∀ *i*<*length ms*$_2$. ∀ *V*∈*rv S* (*CFG-node* ((*targetnode a'* # *tl ms*$_2$)!*i*)).
(*fst* (*s*$_1$'!(*length* [*targetnode a*] + *i*))) *V* = (*fst* (*s*$_2$!*i*)) *V*
by *clarsimp*(*case-tac i*,*auto*)
with ‹∀ *m*∈*set ms*$_1$'. *valid-node m*› ‹∀ *m*∈*set ms*$_2$. *valid-node m*›
‹*length ms*$_1$' = *length s*$_1$'› ‹*length ms*$_2$ = *length s*$_2$›
‹∀ *m*∈*set* (*tl ms*$_2$). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*ms*$_1$' = [*targetnode a*] @ *targetnode a'* # *tl ms*$_2$›
‹*targetnode a'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*return-node* (*targetnode a'*)›
‹*obs ms*$_1$' ⌊*HRB-slice S*⌋$_{CFG}$ = *obs ms*$_2$ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*get-proc* (*targetnode a'*) = *get-proc* (*hd ms*$_2$)›
‹∀ *m* ∈ *set* (*tl ms*$_1$'). *return-node m*› ‹*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*call-of-return-node* (*targetnode a'*) (*sourcenode a*)›
‹∀ *i*<*length ms*$_2$. *snd* (*s*$_1$' ! (*length* [*targetnode a*] + *i*)) = *snd* (*s*$_2$! *i*)›
show *?thesis* **by**(*auto intro*!:*WSI*)
next
case (*Cons mx' msx'*)
with ‹*ms*$_1$ = *msx*@*mx*#*tl ms*$_2$› ‹*hd ms*$_1$ = *sourcenode a*›
have [*simp*]:*mx'* = *sourcenode a* **and** [*simp*]:*tl ms*$_1$ = *msx'*@*mx*#*tl ms*$_2$
by *simp-all*
from ‹*ms*$_1$' = *targetnode a* # *targetnode a'* # *tl ms*$_1$›
have *ms*$_1$' = (*targetnode a* # *targetnode a'* # *msx'*)@*mx*#*tl ms*$_2$
by *simp*
from ‹∀ *i*<*length ms*$_2$. *snd* (*s*$_1$! (*length msx* + *i*)) = *snd* (*s*$_2$! *i*)› *Cons*
have ∀ *i*<*length ms*$_2$.

snd ($s_1{}'$! (*length* (*targetnode a* # *targetnode a'* # *msx'*) + *i*)) = *snd* (s_2 ! *i*)
by *fastforce*
from ‹∀ *V*∈*rv S* (*CFG-node mx*). (*fst* (s_1 ! *length msx*)) *V* = *state-val* s_2 *V*› *Cons*
have ∀ *V*∈*rv S* (*CFG-node mx*).
(*fst* ($s_1{}'$! *length*(*targetnode a* # *targetnode a'* # *msx'*))) *V* = *state-val* s_2 *V*
by *simp*
with ‹∀ *i* < *length* ms_2. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl* ms_2)!*i*)).
(*fst* (s_1!(*length msx* + *i*))) *V* = (*fst* (s_2!*i*)) *V*› *Cons*
have ∀ *i*<*length* ms_2. ∀ *V*∈*rv S* (*CFG-node* ((*mx* # *tl* ms_2)!*i*)).
(*fst* ($s_1{}'$!(*length* (*targetnode a* # *targetnode a'* # *msx'*) + *i*))) *V* =
(*fst* (s_2!*i*)) *V*
by *clarsimp*
with ‹∀ *m*∈*set* $ms_1{}'$. *valid-node m*› ‹∀ *m*∈*set* ms_2. *valid-node m*›
‹*length* $ms_1{}'$ = *length* $s_1{}'$› ‹*length* ms_2 = *length* s_2›
‹$ms_1{}'$ = (*targetnode a* # *targetnode a'* # *msx'*)@*mx*#*tl* ms_2›
‹*return-node* (*targetnode a'*)›
‹∀ *m*∈*set* (*tl* ms_2). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*msx* ≠ [] ⟶ (∃ *mx'*. *call-of-return-node mx mx'* ∧ *mx'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$)›
‹*obs* $ms_1{}'$ ⌊*HRB-slice S*⌋$_{CFG}$ = *obs* ms_2 ⌊*HRB-slice S*⌋$_{CFG}$› *Cons*
‹*get-proc mx* = *get-proc* (*hd* ms_2)› ‹∀ *m* ∈ *set* (*tl* $ms_1{}'$). *return-node m*›
‹∀ *i*<*length* ms_2.
snd ($s_1{}'$! (*length* (*targetnode a* # *targetnode a'* # *msx'*) + *i*)) = *snd* (s_2 ! *i*)›
show *?thesis* **by** −(*rule WSI*,*clarsimp*+,*fastforce*,*clarsimp*+)
qed
next
case (*silent-move-return a* s_1 $s_1{}'$ *Q p f'* ms_1 *S* $ms_1{}'$)
note *obs-eq* = ‹∀ *a*∈*set* (*tl* $ms_1{}'$). *return-node a* ⟹
obs $ms_1{}'$ ⌊*HRB-slice S*⌋$_{CFG}$ = *obs* ms_2 ⌊*HRB-slice S*⌋$_{CFG}$›
from ‹*transfer* (*kind a*) s_1 = $s_1{}'$› ‹*kind a* = *Q*↩$_{pf'}$› ‹s_1 ≠ []› ‹$s_1{}'$ ≠ []›
obtain cf_1 cfx_1 cfs_1 $cf_1{}'$ **where** [*simp*]:s_1 = cf_1#cfx_1#cfs_1
and $s_1{}'$ = (*f'* (*fst* cf_1) (*fst* cfx_1),*snd* cfx_1)#cfs_1
by(*cases* s_1,*auto*,*case-tac list*,*fastforce*+)
from ‹s_2 ≠ []› **obtain** cf_2 cfs_2 **where** [*simp*]:s_2 = cf_2#cfs_2 **by**(*cases* s_2) *auto*
from ‹*length* ms_1 = *length* s_1› **have** ms_1 ≠ [] **and** *tl* ms_1 ≠ [] **by**(*cases* ms_1,*auto*)+
from ‹*valid-edge a*› ‹*kind a* = *Q*↩$_{p}$*f'*›
obtain *a' Q' r' fs'* **where** *valid-edge a'* **and** *kind a'* = *Q'*:*r'*↪$_{p}$*fs'*
and *a* ∈ *get-return-edges a'*
by −(*drule return-needs-call*,*auto*)
then obtain *ins outs* **where** (*p*,*ins*,*outs*) ∈ *set procs*
by(*fastforce dest*!:*callee-in-procs*)
with ‹*valid-edge a*› ‹*kind a* = *Q*↩$_{pf'}$›
have *f'* (*fst* cf_1) (*fst* cfx_1) =
(*fst* cfx_1)(*ParamDefs* (*targetnode a*) [:=] *map* (*fst* cf_1) *outs*)

by(*rule CFG-return-edge-fun*)
with ‹$s_1' = (f'\ (fst\ cf_1)\ (fst\ cfx_1), snd\ cfx_1) \# cfs_1$›
have [*simp*]:$s_1' = ((fst\ cfx_1)$
($ParamDefs\ (targetnode\ a)$ [:=] $map\ (fst\ cf_1)\ outs), snd\ cfx_1) \# cfs_1$ **by** *simp*
from ‹$\forall\, m{\in}set\ ms_1.\ valid\text{-}node\ m$› ‹$ms_1' = tl\ ms_1$› **have** $\forall\, m{\in}set\ ms_1'.\ valid\text{-}node\ m$
by(*cases* ms_1) *auto*
from ‹$length\ ms_1 = length\ s_1$› ‹$ms_1' = tl\ ms_1$›
have $length\ ms_1' = length\ s_1'$ **by** *simp*
from ‹$\forall\, m{\in}set\ (tl\ ms_1).\ return\text{-}node\ m$› ‹$ms_1' = tl\ ms_1$› ‹$ms_1 \neq []$› ‹$tl\ ms_1 \neq []$›
have $\forall\, m{\in}set\ (tl\ ms_1').\ return\text{-}node\ m$ **by**(*cases* ms_1)(*auto*,*cases* ms_1',*auto*)
from *obs-eq*[*OF this*] **have** $obs\ ms_1'\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = obs\ ms_2\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ ·
show *?case*
proof(*cases msx*)
case *Nil*
with ‹$ms_1 = msx@mx\#tl\ ms_2$› ‹$hd\ ms_1 = sourcenode\ a$›
have $mx = sourcenode\ a$ **and** $tl\ ms_1 = tl\ ms_2$ **by** *simp-all*
with ‹$\exists\, m{\in}set\ (tl\ ms_1).\ \exists\, m'.\ call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge m' \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
‹$\forall\, m{\in}set\ (tl\ ms_2).\ \exists\, m'.\ call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge m' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
have *False* **by** *fastforce*
thus *?thesis* **by** *simp*
next
case ($Cons\ mx'\ msx'$)
with ‹$ms_1 = msx@mx\#tl\ ms_2$› ‹$hd\ ms_1 = sourcenode\ a$›
have [*simp*]:$mx' = sourcenode\ a$ **and** [*simp*]:$tl\ ms_1 = msx'@mx\#tl\ ms_2$
by *simp-all*
from ‹$ms_1' = tl\ ms_1$› **have** $ms_1' = msx'@mx\#tl\ ms_2$ **by** *simp*
with ‹$ms_1 = msx@mx\#tl\ ms_2$› ‹$\forall\, m{\in}set\ (tl\ ms_1).\ return\text{-}node\ m$› *Cons*
have $\forall\, m{\in}set\ (tl\ ms_1').\ return\text{-}node\ m$
by(*cases* msx') *auto*
from ‹$\forall\, i{<}length\ ms_2.\ snd\ (s_1\ !\ (length\ msx + i)) = snd\ (s_2\ !\ i)$› *Cons*
have $\forall\, i{<}length\ ms_2.\ snd\ (s_1'\ !\ (length\ msx' + i)) = snd\ (s_2\ !\ i)$
by *auto*(*case-tac i*,*auto*,*cases* msx',*auto*)
from ‹$\forall\, i{<}length\ ms_2.\ \forall\, V{\in}rv\ S\ (CFG\text{-}node\ ((mx\ \#\ tl\ ms_2)\ !\ i)).$
$(fst\ (s_1\ !\ (length\ msx + i)))\ V = (fst\ (s_2\ !\ i))\ V$›
‹$length\ ms_2 = length\ s_2$› ‹$s_2 \neq []$›
have $\forall\, V{\in}rv\ S\ (CFG\text{-}node\ mx).\ (fst\ (s_1\ !\ length\ msx))\ V = state\text{-}val\ s_2\ V$
by *fastforce*
have $\forall\, V{\in}rv\ S\ (CFG\text{-}node\ mx).\ (fst\ (s_1'\ !\ length\ msx'))\ V = state\text{-}val\ s_2\ V$
proof(*cases* msx')
case *Nil*
with ‹$\forall\, V{\in}rv\ S\ (CFG\text{-}node\ mx).\ (fst\ (s_1\ !\ length\ msx))\ V = state\text{-}val\ s_2\ V$›
‹$msx = mx'\#msx'$›
have *rv*:$\forall\, V{\in}rv\ S\ (CFG\text{-}node\ mx).\ fst\ cfx_1\ V = fst\ cf_2\ V$ **by** *fastforce*
from *Nil* ‹$tl\ ms_1 = msx'@mx\#tl\ ms_2$› ‹$hd\ (tl\ ms_1) = targetnode\ a$›

have [*simp*]:$mx = targetnode\ a$ **by** *simp*
from *Cons*
‹$msx \neq [] \longrightarrow (\exists\, mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$)›
obtain mx'' **where** *call-of-return-node* $mx\ mx''$ **and** $mx'' \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by *blast*
hence $mx \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*rule call-node-notin-slice-return-node-neither*)
have $\forall\, V{\in}rv\ S\ (CFG\text{-}node\ mx).$
$(fst\ cfx_1)(ParamDefs\ (targetnode\ a)\ [:=]\ map\ (fst\ cf_1)\ outs)\ V = fst\ cf_2\ V$
proof
fix V **assume** $V{\in}rv\ S\ (CFG\text{-}node\ mx)$
show $(fst\ cfx_1)(ParamDefs\ (targetnode\ a)\ [:=]\ map\ (fst\ cf_1)\ outs)\ V = fst\ cf_2\ V$
proof(*cases* $V \in set\ (ParamDefs\ (targetnode\ a))$)
case *True*
with ‹*valid-edge a*› **have** $V \in Def\ (targetnode\ a)$
by(*fastforce intro*:*ParamDefs-in-Def*)
with ‹*valid-edge a*› **have** $V \in Def_{SDG}\ (CFG\text{-}node\ (targetnode\ a))$
by(*auto intro*!:*CFG-Def-SDG-Def*)
from ‹$V{\in}rv\ S\ (CFG\text{-}node\ mx)$› **obtain** *as* n'
where $targetnode\ a\ -as\rightarrow_\iota *\ parent\text{-}node\ n'$
and $n' \in HRB\text{-}slice\ S\ V \in Use_{SDG}\ n'$
and $\forall\, n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge parent\text{-}node\ n'' \in set\ (sourcenodes\ as) \longrightarrow V \notin Def_{SDG}\ n''$ **by**(*fastforce elim*:*rvE*)
from ‹$targetnode\ a\ -as\rightarrow_\iota *\ parent\text{-}node\ n'$› ‹$n' \in HRB\text{-}slice\ S$›
‹$mx \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
obtain *ax asx* **where** $as = ax\#asx$
by(*auto simp*:*intra-path-def*)(*erule path.cases*,
auto dest:*valid-SDG-node-in-slice-parent-node-in-slice*
simp:*SDG-to-CFG-set-def*)
with ‹$targetnode\ a\ -as\rightarrow_\iota *\ parent\text{-}node\ n'$›
have $targetnode\ a = sourcenode\ ax$ **and** *valid-edge ax*
by(*auto elim*:*path.cases simp*:*intra-path-def*)
with ‹$\forall\, n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge parent\text{-}node\ n'' \in set\ (sourcenodes\ as) \longrightarrow V \notin Def_{SDG}\ n''$› ‹$as = ax\#asx$› ‹$V \in Def_{SDG}\ (CFG\text{-}node\ (targetnode\ a))$›
have *False* **by**(*fastforce simp*:*sourcenodes-def*)
thus *?thesis* **by** *simp*
next
case *False*
with ‹$V{\in}rv\ S\ (CFG\text{-}node\ mx)$› *rv* **show** *?thesis*
by(*fastforce dest*:*fun-upds-notin*[*of* - - $fst\ cfx_1$])
qed
qed
with *Nil* ‹$msx = mx'\#msx'$› **show** *?thesis* **by** *fastforce*
next

case *Cons*
with ‹∀ *V*∈*rv S* (*CFG-node mx*). (*fst* (*s*$_1$! *length msx*)) *V* = *state-val s*$_2$ *V*›
‹*msx* = *mx*′#*msx*′›
show *?thesis* **by** *fastforce*
qed
with ‹∀ *V*∈*rv S* (*CFG-node mx*). (*fst* (*s*$_1$! *length msx*)) *V* = *state-val s*$_2$ *V*› *Cons*
have ∀ *V*∈*rv S* (*CFG-node mx*). (*fst* (*s*$_1$′ ! *length msx*′)) *V* = *state-val s*$_2$ *V*
by(*cases msx*′) *auto*
with ‹∀ *i* < *length ms*$_2$. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl ms*$_2$)!*i*)).
(*fst* (*s*$_1$!(*length msx* + *i*))) *V* = (*fst* (*s*$_2$!*i*)) *V*› *Cons*
have ∀ *i*<*length ms*$_2$. ∀ *V*∈*rv S* (*CFG-node* ((*mx* # *tl ms*$_2$) ! *i*)).
(*fst* (*s*$_1$′ ! (*length msx*′ + *i*))) *V* = (*fst* (*s*$_2$! *i*)) *V*
by *clarsimp*(*case-tac i*,*auto*)
with ‹∀ *m*∈*set ms*$_1$′. *valid-node m*› ‹∀ *m*∈*set ms*$_2$. *valid-node m*›
‹*length ms*$_1$′ = *length s*$_1$′› ‹*length ms*$_2$ = *length s*$_2$›
‹*ms*$_1$′ = *msx*′@*mx*#*tl ms*$_2$› ‹*get-proc mx* = *get-proc* (*hd ms*$_2$)›
‹∀ *m*∈*set* (*tl ms*$_2$). ∃ *m*′. *call-of-return-node m m*′ ∧ *m*′ ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*msx* ≠ [] ⟶ (∃ *mx*′. *call-of-return-node mx mx*′ ∧ *mx*′ ∉ ⌊*HRB-slice S*⌋$_{CFG}$)›
‹∀ *m*∈*set* (*tl ms*$_1$′). *return-node m*› *Cons* ‹*get-proc mx* = *get-proc* (*hd ms*$_2$)›
‹∀ *m*∈*set* (*tl ms*$_2$). ∃ *m*′. *call-of-return-node m m*′ ∧ *m*′ ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*obs ms*$_1$′ ⌊*HRB-slice S*⌋$_{CFG}$ = *obs ms*$_2$ ⌊*HRB-slice S*⌋$_{CFG}$›
‹∀ *i*<*length ms*$_2$. *snd* (*s*$_1$′ ! (*length msx*′ + *i*)) = *snd* (*s*$_2$! *i*)›
show *?thesis* **by**(*auto intro*!:*WSI*)
qed
qed
qed

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*
by(*induct S f*≡*kind ms*$_1$ *s*$_1$ *as ms*$_1$′ *s*$_1$′ *rule*:*silent-moves.induct*,
auto dest:*WS-silent-move*)

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(*atomize-elim*)
from ‹((*ms*$_1$,*s*$_1$),(*ms*$_2$,*s*$_2$)) ∈ *WS S*› **obtain** *msx mx*
where *assms*:∀ *m* ∈ *set ms*$_1$. *valid-node m* ∀ *m* ∈ *set ms*$_2$. *valid-node m*
length ms$_1$ = *length s*$_1$ *length ms*$_2$ = *length s*$_2$ *s*$_1$ ≠ [] *s*$_2$ ≠ []
ms$_1$ = *msx*@*mx*#*tl ms*$_2$ *get-proc mx* = *get-proc* (*hd ms*$_2$)
∀ *m* ∈ *set* (*tl ms*$_2$). ∃ *m*′. *call-of-return-node m m*′ ∧ *m*′ ∈ ⌊*HRB-slice S*⌋$_{CFG}$

$msx \neq [] \longrightarrow (\exists\, mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG})$
$\forall\, m \in set\ (tl\ ms_1).\ return\text{-}node\ m$
$\forall\, i < length\ ms_2.\ snd\ (s_1!(length\ msx + i)) = snd\ (s_2!i)$
$\forall\, i < length\ ms_2.\ \forall\, V \in rv\ S\ (CFG\text{-}node\ ((mx\#tl\ ms_2)!i)).$
$(fst\ (s_1!(length\ msx + i)))\ V = (fst\ (s_2!i))\ V$
$obs\ ms_1\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = obs\ ms_2\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*fastforce elim*:*WS.cases*)
from ‹S,$kind \vdash (ms_1,s_1)\ -a\rightarrow\ (ms_1',s_1')$› *assms*
show $\exists\, as.\ ((ms_1',s_1'),(ms_1',transfer\ (slice\text{-}kind\ S\ a)\ s_2)) \in WS\ S \wedge$
S,$slice\text{-}kind\ S \vdash (ms_2,s_2)\ =as\ @\ [a]\Rightarrow\ (ms_1',transfer\ (slice\text{-}kind\ S\ a)\ s_2)$
proof(*induct S f*≡*kind ms*$_1$ *s*$_1$ *a ms*$_1'$ *s*$_1'$ *rule*:*observable-move.induct*)
case (*observable-move-intra a s*$_1$ *s*$_1'$ *ms*$_1$ *S ms*$_1'$)
from ‹$s_1 \neq []$› ‹$s_2 \neq []$› **obtain** $cf_1\ cfs_1\ cf_2\ cfs_2$ **where** [*simp*]:$s_1 = cf_1\#cfs_1$
and [*simp*]:$s_2 = cf_2\#cfs_2$ **by**(*cases s*$_1$,*auto*,*cases s*$_2$,*fastforce*+)
from ‹*length ms*$_1$ = *length s*$_1$› ‹$s_1 \neq []$› **have** [*simp*]:$ms_1 \neq []$ **by**(*cases ms*$_1$) *auto*
from ‹*length ms*$_2$ = *length s*$_2$› ‹$s_2 \neq []$› **have** [*simp*]:$ms_2 \neq []$ **by**(*cases ms*$_2$) *auto*
from ‹$\forall\, m \in set\ (tl\ ms_1).\ \exists\, m'.\ call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge m' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
‹*hd ms*$_1$ = *sourcenode a*› ‹*ms*$_1$ = *msx*@*mx*#*tl ms*$_2$›
‹$msx \neq [] \longrightarrow (\exists\, mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG})$›
have [*simp*]:*mx* = *sourcenode a msx* = [] **and** [*simp*]:*tl ms*$_2$ = *tl ms*$_1$
by(*cases msx*,*auto*)+
hence *length ms*$_1$ = *length ms*$_2$ **by**(*cases ms*$_2$) *auto*
with ‹*length ms*$_1$ = *length s*$_1$› ‹*length ms*$_2$ = *length s*$_2$›
have *length s*$_1$ = *length s*$_2$ **by** *simp*
from ‹*hd ms*$_1$ ∈ $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$› ‹*hd ms*$_1$ = *sourcenode a*›
have *sourcenode a* ∈ $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **by** *simp*
with ‹*valid-edge a*›
have $obs\text{-}intra\ (sourcenode\ a)\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{sourcenode\ a\}$
by(*fastforce intro*!:*n-in-obs-intra*)
from ‹$\forall\, m \in set\ (tl\ ms_2).\ \exists\, m'.\ call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge m' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
‹$obs\text{-}intra\ (sourcenode\ a)\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{sourcenode\ a\}$›
‹*hd ms*$_1$ = *sourcenode a*›
have (*hd ms*$_1$#*tl ms*$_1$) ∈ *obs* ([]@*hd ms*$_1$#*tl ms*$_1$) $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*cases ms*$_1$)(*auto intro*!:*obsI*)
hence *ms*$_1$ ∈ *obs ms*$_1$ $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **by** *simp*
with ‹*obs ms*$_1$ $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ = *obs ms*$_2$ $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
have *ms*$_1$ ∈ *obs ms*$_2$ $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **by** *simp*
from ‹*ms*$_2 \neq []$› ‹*length ms*$_2$ = *length s*$_2$› **have** *length s*$_2$ = *length* (*hd ms*$_2$#*tl ms*$_2$)
by(*fastforce dest*!:*hd-Cons-tl*)
from ‹$\forall\, m \in set\ (tl\ ms_1).\ return\text{-}node\ m$› **have** $\forall\, m \in set\ (tl\ ms_2).\ return\text{-}node\ m$
by *simp*
with ‹*ms*$_1$ ∈ *obs ms*$_2$ $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
have *hd ms*$_1$ ∈ *obs-intra* (*hd ms*$_2$) $\lfloor HRB\text{-}slice\ S \rfloor_{CFG}$

proof(*rule obsE*)
fix *nsx n nsx' n'*
assume $ms_2 = nsx\ @\ n\ \#\ nsx'$ **and** $ms_1 = n'\ \#\ nsx'$
and $n' \in$ *obs-intra n* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
from ‹$ms_2 = nsx\ @\ n\ \#\ nsx'$› ‹$ms_1 = n'\ \#\ nsx'$› ‹*tl* $ms_2 =$ *tl* ms_1›
have [*simp*]:$nsx = []$ **by**(*cases nsx*) *auto*
with ‹$ms_2 = nsx\ @\ n\ \#\ nsx'$› **have** [*simp*]:$n =$ *hd* ms_2 **by** *simp*
from ‹$ms_1 = n'\ \#\ nsx'$› **have** [*simp*]:$n' =$ *hd* ms_1 **by** *simp*
with ‹$n' \in$ *obs-intra n* $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› **show** *?thesis* **by** *simp*
qed
with ‹*length* $s_2 =$ *length* (*hd* ms_2#*tl* ms_2)› ‹$\forall\, m \in$ *set* (*tl* ms_2). *return-node m*›
obtain *as* **where** *S*,*slice-kind S* $\vdash$ (*hd* ms_2#*tl* ms_2,s_2) =*as*⇒$_\tau$ (*hd* ms_1#*tl* ms_1,s_2)
by(*fastforce elim*:*silent-moves-intra-path-obs*[*of - - - s*$_2$ *tl* ms_2])
with ‹$ms_2 \neq []$› **have** *S*,*slice-kind S* $\vdash$ (ms_2,s_2) =*as*⇒$_\tau$ (ms_1,s_2)
by(*fastforce dest*!:*hd-Cons-tl*)
from ‹*valid-edge a*› **have** *valid-node* (*sourcenode a*) **by** *simp*
hence *sourcenode a* −[]→$_\iota$* *sourcenode a*
by(*fastforce intro*:*empty-path simp*:*intra-path-def*)
with ‹*sourcenode a* $\in$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
have $\forall\, V.\ V \in Use_{SDG}$ (*CFG-node* (*sourcenode a*))
$\longrightarrow$ $V \in$ *rv S* (*CFG-node* (*sourcenode a*))
by *auto*(*rule rvI*,*auto simp*:*SDG-to-CFG-set-def sourcenodes-def*)
with ‹*valid-node* (*sourcenode a*)›
have $\forall\, V \in$ *Use* (*sourcenode a*). $V \in$ *rv S* (*CFG-node* (*sourcenode a*))
by(*fastforce intro*:*CFG-Use-SDG-Use*)
from ‹$\forall\, i <$ *length* ms_2. $\forall\, V \in$ *rv S* (*CFG-node* ((mx#*tl* ms_2)!i)).
(*fst* (s_1!(*length msx* + i))) V = (*fst* (s_2!i)) V› ‹*length* ms_2 = *length* s_2›
have $\forall\, V \in rv\ S$ (*CFG-node mx*). (*fst* (s_1 ! *length msx*)) V = *state-val* s_2 V
by(*cases* ms_2) *auto*
with ‹$\forall\, V \in$ *Use* (*sourcenode a*). $V \in$ *rv S* (*CFG-node* (*sourcenode a*))›
have $\forall\, V \in$ *Use* (*sourcenode a*). *fst* cf_1 V = *fst* cf_2 V **by** *fastforce*
moreover
from ‹$\forall\, i <$*length* ms_2. *snd* (s_1 ! (*length msx* + i)) = *snd* (s_2 ! i)›
have *snd* (*hd* s_1) = *snd* (*hd* s_2) **by**(*erule-tac x*=*0* **in** *allE*) *auto*
ultimately have *pred* (*kind a*) s_2
using ‹*valid-edge a*› ‹*pred* (*kind a*) s_1› ‹*length* s_1 = *length* s_2›
by(*fastforce intro*:*CFG-edge-Uses-pred-equal*)
from ‹ms_1' = *targetnode a* # *tl* ms_1› ‹*length* s_1' = *length* s_1›
‹*length* ms_1 = *length* s_1› **have** *length* ms_1' = *length* s_1' **by** *simp*
from ‹*transfer* (*kind a*) s_1 = s_1'› ‹*intra-kind* (*kind a*)›
obtain cf_1' **where** [*simp*]:s_1' = cf_1'#cfs_1
by(*cases* cf_1,*cases kind a*,*auto simp*:*intra-kind-def*)
from ‹*intra-kind* (*kind a*)› ‹*sourcenode a* $\in$ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*pred* (*kind a*) s_2›
have *pred* (*slice-kind S a*) s_2 **by**(*simp add*:*slice-intra-kind-in-slice*)
from ‹*valid-edge a*› ‹*length* s_1 = *length* s_2› ‹*transfer* (*kind a*) s_1 = s_1'›
have *length* s_1' = *length* (*transfer* (*slice-kind S a*) s_2)
by(*fastforce intro*:*length-transfer-kind-slice-kind*)

with ‹*length* s_1 = *length* s_2›
have *length* s_2 = *length* (*transfer* (*slice-kind S a*) s_2) **by** *simp*
with ‹*pred* (*slice-kind S a*) s_2› ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
‹∀ *m* ∈ *set* (*tl* ms_1). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*hd* ms_1 ∈ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*hd* ms_1 = *sourcenode a*›
‹*length* ms_1 = *length* s_1› ‹*length* s_1 = *length* s_2›
‹ms_1' = *targetnode a* # *tl* ms_1› ‹∀ *m* ∈ *set* (*tl* ms_2). *return-node m*›
have *S*,*slice-kind S* ⊢ (ms_1,s_2) −*a*→ (ms_1',*transfer* (*slice-kind S a*) s_2)
by(*auto intro*:*observable-move*.*observable-move-intra*)
with ‹*S*,*slice-kind S* ⊢ (ms_2,s_2) =*as*⇒$_\tau$ (ms_1,s_2)›
have *S*,*slice-kind S* ⊢ (ms_2,s_2) =*as*@[*a*]⇒ (ms_1',*transfer* (*slice-kind S a*) s_2)
by(*rule observable-moves-snoc*)
from ‹∀ *m* ∈ *set* ms_1. *valid-node m*› ‹ms_1' = *targetnode a* # *tl* ms_1› ‹*valid-edge a*›
have ∀ *m* ∈ *set* ms_1'. *valid-node m* **by**(*cases* ms_1) *auto*
from ‹∀ *m* ∈ *set* (*tl* ms_2). *return-node m*› ‹ms_1' = *targetnode a* # *tl* ms_1›
‹ms_1' = *targetnode a* # *tl* ms_1›
have ∀ *m* ∈ *set* (*tl* ms_1'). *return-node m* **by** *fastforce*
from ‹ms_1' = *targetnode a* # *tl* ms_1› ‹*tl* ms_2 = *tl* ms_1›
have ms_1' = [] @ *targetnode a* # *tl* ms_2 **by** *simp*
from ‹*intra-kind* (*kind a*)› ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
have *cf2'*:∃ cf_2'. *transfer* (*slice-kind S a*) s_2 = cf_2'#cfs_2 ∧ *snd* cf_2' = *snd* cf_2
by(*cases* cf_2)(*auto dest*:*slice-intra-kind-in-slice simp*:*intra-kind-def*)
from ‹*transfer* (*kind a*) s_1 = s_1'› ‹*intra-kind* (*kind a*)›
have *snd* cf_1' = *snd* cf_1 **by**(*auto simp*:*intra-kind-def*)
with ‹∀ *i*<*length* ms_2. *snd* (s_1 ! (*length msx* + *i*)) = *snd* (s_2 ! *i*)›
‹*snd* (*hd* s_1) = *snd* (*hd* s_2)› ‹ms_1' = [] @ *targetnode a* # *tl* ms_2›
cf2' ‹*length* ms_1 = *length* ms_2›
have ∀ *i*<*length* ms_1'. *snd* (s_1' ! *i*) = *snd* (*transfer* (*slice-kind S a*) s_2 ! *i*)
by *auto*(*case-tac i*,*auto*)
have ∀ *V* ∈ *rv S* (*CFG-node* (*targetnode a*)).
fst cf_1' *V* = *state-val* (*transfer* (*slice-kind S a*) s_2) *V*
proof
fix *V* **assume** *V* ∈ *rv S* (*CFG-node* (*targetnode a*))
show *fst* cf_1' *V* = *state-val* (*transfer* (*slice-kind S a*) s_2) *V*
proof(*cases V* ∈ *Def* (*sourcenode a*))
case *True*
from ‹*intra-kind* (*kind a*)› **have** (∃ *f*. *kind a* = ⇑*f*) ∨ (∃ *Q*. *kind a* = (*Q*)$_\surd$)
by(*simp add*:*intra-kind-def*)
thus *?thesis*
proof
assume ∃ *f*. *kind a* = ⇑*f*
then obtain *f'* **where** *kind a* = ⇑*f'* **by** *blast*
with ‹*transfer* (*kind a*) s_1 = s_1'›
have s_1' = (*f'* (*fst* cf_1),*snd* cf_1) # cfs_1 **by** *simp*
from ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*kind a* = ⇑*f'*›
have *slice-kind S a* = ⇑*f'*
by(*fastforce dest*:*slice-intra-kind-in-slice simp*:*intra-kind-def*)

hence *transfer* (*slice-kind S a*) s_2 = (*f'* (*fst cf*$_2$),*snd cf*$_2$) # *cfs*$_2$ **by** *simp*
from ‹*valid-edge a*› ‹∀ *V* ∈ *Use* (*sourcenode a*). *fst cf*$_1$ *V* = *fst cf*$_2$ *V*›
‹*intra-kind* (*kind a*)› ‹*pred* (*kind a*) s_1› ‹*pred* (*kind a*) s_2›
have ∀ *V* ∈ *Def* (*sourcenode a*). *state-val* (*transfer* (*kind a*) s_1) *V* =
state-val (*transfer* (*kind a*) s_2) *V*
by −(*erule CFG-intra-edge-transfer-uses-only-Use*,*auto*)
with ‹*kind a* = ⇑*f'*› ‹s_1' = (*f'* (*fst cf*$_1$),*snd cf*$_1$) # *cfs*$_1$› *True*
‹*transfer* (*slice-kind S a*) s_2 = (*f'* (*fst cf*$_2$),*snd cf*$_2$) # *cfs*$_2$›
show *?thesis* **by** *simp*
next
assume ∃ *Q*. *kind a* = (*Q*)$_\surd$
then obtain *Q* **where** *kind a* = (*Q*)$_\surd$ **by** *blast*
with ‹*transfer* (*kind a*) s_1 = s_1'› **have** s_1' = *cf*$_1$ # *cfs*$_1$ **by** *simp*
from ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*kind a* = (*Q*)$_\surd$›
have *slice-kind S a* = (*Q*)$_\surd$
by(*fastforce dest*:*slice-intra-kind-in-slice simp*:*intra-kind-def*)
hence *transfer* (*slice-kind S a*) s_2 = s_2 **by** *simp*
from ‹*valid-edge a*› ‹∀ *V* ∈ *Use* (*sourcenode a*). *fst cf*$_1$ *V* = *fst cf*$_2$ *V*›
‹*intra-kind* (*kind a*)› ‹*pred* (*kind a*) s_1› ‹*pred* (*kind a*) s_2›
have ∀ *V* ∈ *Def* (*sourcenode a*). *state-val* (*transfer* (*kind a*) s_1) *V* =
state-val (*transfer* (*kind a*) s_2) *V*
by −(*erule CFG-intra-edge-transfer-uses-only-Use*,*auto simp*:*intra-kind-def*)
with *True* ‹*kind a* = (*Q*)$_\surd$› ‹s_1' = *cf*$_1$ # *cfs*$_1$›
‹*transfer* (*slice-kind S a*) s_2 = s_2›
show *?thesis* **by** *simp*
qed
next
case *False*
with ‹*valid-edge a*› ‹*intra-kind* (*kind a*)› ‹*pred* (*kind a*) s_1›
have *state-val* (*transfer* (*kind a*) s_1) *V* = *state-val* s_1 *V*
by(*fastforce intro*:*CFG-intra-edge-no-Def-equal*)
with ‹*transfer* (*kind a*) s_1 = s_1'› **have** *fst cf*$_1$' *V* = *fst cf*$_1$ *V* **by** *simp*
from ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*intra-kind* (*kind a*)›
have *slice-kind S a* = *kind a* **by**(*fastforce intro*:*slice-intra-kind-in-slice*)
from *False* ‹*valid-edge a*› ‹*pred* (*kind a*) s_2› ‹*intra-kind* (*kind a*)›
have *state-val* (*transfer* (*kind a*) s_2) *V* = *state-val* s_2 *V*
by(*fastforce intro*:*CFG-intra-edge-no-Def-equal*)
with ‹*slice-kind S a* = *kind a*›
have *state-val* (*transfer* (*slice-kind S a*) s_2) *V* = *fst cf*$_2$ *V* **by** *simp*
from ‹*V* ∈ *rv S* (*CFG-node* (*targetnode a*))› **obtain** *as' nx*
where *targetnode a* −*as'*→$_\iota$* *parent-node nx*
and *nx* ∈ *HRB-slice S* **and** *V* ∈ *Use*$_{SDG}$ *nx*
and ∀ *n''*. *valid-SDG-node n''* ∧ *parent-node n''* ∈ *set* (*sourcenodes as'*)
⟶ *V* ∉ *Def*$_{SDG}$ *n''*
by(*fastforce elim*:*rvE*)
with ‹∀ *n''*. *valid-SDG-node n''* ∧ *parent-node n''* ∈ *set* (*sourcenodes as'*)
⟶ *V* ∉ *Def*$_{SDG}$ *n''*› *False*
have *all*:∀ *n''*. *valid-SDG-node n''* ∧
parent-node n'' ∈ *set* (*sourcenodes* (*a*#*as'*)) ⟶ *V* ∉ *Def*$_{SDG}$ *n''*

by(*fastforce dest*:*SDG-Def-parent-Def simp*:*sourcenodes-def*)
from ‹*valid-edge a*› ‹*targetnode a* $-as'\rightarrow_\iota *$ *parent-node nx*›
‹*intra-kind* (*kind a*)›
have *sourcenode a* $-a\#as'\rightarrow_\iota *$ *parent-node nx*
by(*fastforce intro*:*Cons-path simp*:*intra-path-def*)
with ‹*nx* ∈ *HRB-slice S*› ‹*V* ∈ Use_{SDG} *nx*› *all*
have *V* ∈ *rv S* (*CFG-node* (*sourcenode a*)) **by**(*fastforce intro*:*rvI*)
with ‹∀ *V* ∈ *rv S* (*CFG-node mx*). (*fst* (s_1!(*length msx*))) *V* = *state-val* s_2 *V*›
‹*state-val* (*transfer* (*slice-kind S a*) s_2) *V* = *fst* cf_2 *V*›
‹*fst* cf_1' *V* = *fst* cf_1 *V*›
show *?thesis* **by** *fastforce*
qed
qed
with ‹∀ *i* < *length* ms_2. ∀ *V* ∈ *rv S* (*CFG-node* (($mx\#tl\ ms_2$)!*i*)).
(*fst* (s_1!(*length msx* + *i*))) *V* = (*fst* (s_2!*i*)) *V*› *cf2'*
‹ms_1' = [] @ *targetnode a* # *tl* ms_2›
‹*length* ms_1 = *length* s_1› ‹*length* ms_2 = *length* s_2› ‹*length* s_1 = *length* s_2›
have ∀ *i*<*length* ms_1'. ∀ *V*∈*rv S* (*CFG-node* ((ms_1')!*i*)).
(*fst* (s_1'!(*length* [] + *i*))) *V* = (*fst* (*transfer* (*slice-kind S a*) s_2 ! *i*)) *V*
by *clarsimp*(*case-tac i*,*auto*)
with ‹∀ *m* ∈ *set* ms_2. *valid-node m*› ‹∀ *m* ∈ *set* ms_1'. *valid-node m*›
‹*length* ms_2 = *length* s_2› ‹*length* s_1' = *length* (*transfer* (*slice-kind S a*) s_2)›
‹*length* ms_1' = *length* s_1'› ‹∀ *m* ∈ *set* (*tl* ms_1'). *return-node m*›
‹ms_1' = [] @ *targetnode a* # *tl* ms_2› ‹*get-proc mx* = *get-proc* (*hd* ms_2)›
‹∀ *m* ∈ *set* (*tl* ms_1). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹∀ *i*<*length* ms_1'. *snd* (s_1' ! *i*) = *snd* (*transfer* (*slice-kind S a*) s_2 ! *i*)›
have ((ms_1',s_1'),(ms_1',*transfer* (*slice-kind S a*) s_2)) ∈ *WS S*
by(*fastforce intro*!:*WSI*)
with ‹*S*,*slice-kind S* ⊢ (ms_2,s_2) =*as*@[*a*]⇒ (ms_1',*transfer* (*slice-kind S a*) s_2)›
show *?case* **by** *blast*
next
case (*observable-move-call a* s_1 s_1' *Q r p fs a'* ms_1 *S* ms_1')
from ‹s_1 ≠ []› ‹s_2 ≠ []› **obtain** cf_1 cfs_1 cf_2 cfs_2 **where** [*simp*]:s_1 = cf_1#cfs_1
and [*simp*]:s_2 = cf_2#cfs_2 **by**(*cases* s_1,*auto*,*cases* s_2,*fastforce*+)
from ‹*length* ms_1 = *length* s_1› ‹s_1 ≠ []› **have** [*simp*]:ms_1 ≠ [] **by**(*cases* ms_1) *auto*
from ‹*length* ms_2 = *length* s_2› ‹s_2 ≠ []› **have** [*simp*]:ms_2 ≠ [] **by**(*cases* ms_2) *auto*
from ‹∀ *m* ∈ *set* (*tl* ms_1). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*hd* ms_1 = *sourcenode a*› ‹ms_1 = *msx*@*mx*#*tl* ms_2›
‹*msx* ≠ [] ⟶ (∃ *mx'*. *call-of-return-node mx mx'* ∧ *mx'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$)›
have [*simp*]:*mx* = *sourcenode a msx* = [] **and** [*simp*]:*tl* ms_2 = *tl* ms_1
by(*cases msx*,*auto*)+
hence *length* ms_1 = *length* ms_2 **by**(*cases* ms_2) *auto*
with ‹*length* ms_1 = *length* s_1› ‹*length* ms_2 = *length* s_2›
have *length* s_1 = *length* s_2 **by** *simp*
from ‹*hd* ms_1 ∈ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*hd* ms_1 = *sourcenode a*›

have *sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$ **by** *simp*
with ‹*valid-edge a*›
have *obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {*sourcenode a*}
by(*fastforce intro*!:*n-in-obs-intra*)
from ‹∀ *m* ∈ *set* (*tl ms*$_2$). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*obs-intra* (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {*sourcenode a*}›
‹*hd ms*$_1$ = *sourcenode a*›
have (*hd ms*$_1$#*tl ms*$_1$) ∈ *obs* ([]@*hd ms*$_1$#*tl ms*$_1$) ⌊*HRB-slice S*⌋$_{CFG}$
by(*cases ms*$_1$)(*auto intro*!:*obsI*)
hence *ms*$_1$ ∈ *obs ms*$_1$ ⌊*HRB-slice S*⌋$_{CFG}$ **by** *simp*
with ‹*obs ms*$_1$ ⌊*HRB-slice S*⌋$_{CFG}$ = *obs ms*$_2$ ⌊*HRB-slice S*⌋$_{CFG}$›
have *ms*$_1$ ∈ *obs ms*$_2$ ⌊*HRB-slice S*⌋$_{CFG}$ **by** *simp*
from ‹*ms*$_2$ ≠ []› ‹*length ms*$_2$ = *length s*$_2$› **have** *length s*$_2$ = *length* (*hd ms*$_2$#*tl ms*$_2$)
by(*fastforce dest*!:*hd-Cons-tl*)
from ‹∀ *m* ∈ *set* (*tl ms*$_1$). *return-node m*› **have** ∀ *m* ∈ *set* (*tl ms*$_2$). *return-node m*
by *simp*
with ‹*ms*$_1$ ∈ *obs ms*$_2$ ⌊*HRB-slice S*⌋$_{CFG}$›
have *hd ms*$_1$ ∈ *obs-intra* (*hd ms*$_2$) ⌊*HRB-slice S*⌋$_{CFG}$
proof(*rule obsE*)
fix *nsx n nsx' n'*
assume *ms*$_2$ = *nsx* @ *n* # *nsx'* **and** *ms*$_1$ = *n'* # *nsx'*
and *n'* ∈ *obs-intra n* ⌊*HRB-slice S*⌋$_{CFG}$
from ‹*ms*$_2$ = *nsx* @ *n* # *nsx'*› ‹*ms*$_1$ = *n'* # *nsx'*› ‹*tl ms*$_2$ = *tl ms*$_1$›
have [*simp*]:*nsx* = [] **by**(*cases nsx*) *auto*
with ‹*ms*$_2$ = *nsx* @ *n* # *nsx'*› **have** [*simp*]:*n* = *hd ms*$_2$ **by** *simp*
from ‹*ms*$_1$ = *n'* # *nsx'*› **have** [*simp*]:*n'* = *hd ms*$_1$ **by** *simp*
with ‹*n'* ∈ *obs-intra n* ⌊*HRB-slice S*⌋$_{CFG}$› **show** *?thesis* **by** *simp*
qed
with ‹*length s*$_2$ = *length* (*hd ms*$_2$#*tl ms*$_2$)› ‹∀ *m* ∈ *set* (*tl ms*$_2$). *return-node m*›
obtain *as* **where** *S*,*slice-kind S* ⊢ (*hd ms*$_2$#*tl ms*$_2$,*s*$_2$) =*as*⇒$_\tau$ (*hd ms*$_1$#*tl ms*$_1$,*s*$_2$)
by(*fastforce elim*:*silent-moves-intra-path-obs*[*of - - - s*$_2$ *tl ms*$_2$])
with ‹*ms*$_2$ ≠ []› **have** *S*,*slice-kind S* ⊢ (*ms*$_2$,*s*$_2$) =*as*⇒$_\tau$ (*ms*$_1$,*s*$_2$)
by(*fastforce dest*!:*hd-Cons-tl*)
from ‹*valid-edge a*› **have** *valid-node* (*sourcenode a*) **by** *simp*
hence *sourcenode a* −[]→$_\iota$∗ *sourcenode a*
by(*fastforce intro*:*empty-path simp*:*intra-path-def*)
with ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
have ∀ *V*. *V* ∈ *Use*$_{SDG}$ (*CFG-node* (*sourcenode a*))
⟶ *V* ∈ *rv S* (*CFG-node* (*sourcenode a*))
by *auto*(*rule rvI*,*auto simp*:*SDG-to-CFG-set-def sourcenodes-def*)
with ‹*valid-node* (*sourcenode a*)›
have ∀ *V* ∈ *Use* (*sourcenode a*). *V* ∈ *rv S* (*CFG-node* (*sourcenode a*))
by(*fastforce intro*:*CFG-Use-SDG-Use*)
from ‹∀ *i* < *length ms*$_2$. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl ms*$_2$)!*i*)).
(*fst* (*s*$_1$!(*length msx* + *i*))) *V* = (*fst* (*s*$_2$!*i*)) *V*› ‹*length ms*$_2$ = *length s*$_2$›

have $\forall V{\in}rv\ S\ (\mathit{CFG\text{-}node}\ mx).\ (\mathit{fst}\ (s_1\ !\ \mathit{length}\ msx))\ V = \mathit{state\text{-}val}\ s_2\ V$
by(*cases* ms_2) *auto*
with ‹$\forall V \in \mathit{Use}\ (\mathit{sourcenode}\ a).\ V \in rv\ S\ (\mathit{CFG\text{-}node}\ (\mathit{sourcenode}\ a))$›
have $\forall V \in \mathit{Use}\ (\mathit{sourcenode}\ a).\ \mathit{fst}\ cf_1\ V = \mathit{fst}\ cf_2\ V$ **by** *fastforce*
moreover
from ‹$\forall i{<}\mathit{length}\ ms_2.\ \mathit{snd}\ (s_1\ !\ (\mathit{length}\ msx + i)) = \mathit{snd}\ (s_2\ !\ i)$›
have $\mathit{snd}\ (\mathit{hd}\ s_1) = \mathit{snd}\ (\mathit{hd}\ s_2)$ **by**(*erule-tac* $x{=}0$ **in** *allE*) *auto*
ultimately have $\mathit{pred}\ (\mathit{kind}\ a)\ s_2$
using ‹*valid-edge a*› ‹$\mathit{pred}\ (\mathit{kind}\ a)\ s_1$› ‹$\mathit{length}\ s_1 = \mathit{length}\ s_2$›
by(*fastforce intro*:*CFG-edge-Uses-pred-equal*)
from ‹$ms_1' = (\mathit{targetnode}\ a)\#(\mathit{targetnode}\ a')\#tl\ ms_1$› ‹$\mathit{length}\ s_1' = Suc(\mathit{length}\ s_1)$›
‹$\mathit{length}\ ms_1 = \mathit{length}\ s_1$› **have** $\mathit{length}\ ms_1' = \mathit{length}\ s_1'$ **by** *simp*
from ‹*valid-edge a*› ‹$\mathit{kind}\ a = Q{:}r{\hookrightarrow_p}fs$› **obtain** *ins outs*
where $(p,\mathit{ins},\mathit{outs}) \in \mathit{set\ procs}$ **by**(*fastforce dest*!:*callee-in-procs*)
with ‹*valid-edge a*› ‹$\mathit{kind}\ a = Q{:}r{\hookrightarrow_p}fs$›
have $(\mathit{THE}\ \mathit{ins}.\ \exists\,\mathit{outs}.\ (p,\mathit{ins},\mathit{outs}) \in \mathit{set\ procs}) = \mathit{ins}$
by(*rule formal-in-THE*)
with ‹$\mathit{transfer}\ (\mathit{kind}\ a)\ s_1 = s_1'$› ‹$\mathit{kind}\ a = Q{:}r{\hookrightarrow_p}fs$›
have $[\mathit{simp}]{:}s_1' = (\mathit{empty}(\mathit{ins}\ [:=]\ \mathit{params}\ fs\ (\mathit{fst}\ cf_1)),r)\#cf_1\#cfs_1$ **by** *simp*
from ‹*valid-edge a'*› ‹$a' \in \mathit{get\text{-}return\text{-}edges}\ a$› ‹*valid-edge a*›
have $\mathit{return\text{-}node}\ (\mathit{targetnode}\ a')$ **by**(*fastforce simp*:*return-node-def*)
with ‹*valid-edge a*› ‹*valid-edge a'*› ‹$a' \in \mathit{get\text{-}return\text{-}edges}\ a$›
have $\mathit{call\text{-}of\text{-}return\text{-}node}\ (\mathit{targetnode}\ a')\ (\mathit{sourcenode}\ a)$
by(*simp add*:*call-of-return-node-def*) *blast*
from ‹$\mathit{sourcenode}\ a \in \lfloor \mathit{HRB\text{-}slice}\ S \rfloor_{CFG}$› ‹$\mathit{pred}\ (\mathit{kind}\ a)\ s_2$› ‹$\mathit{kind}\ a = Q{:}r{\hookrightarrow_p}fs$›
have $\mathit{pred}\ (\mathit{slice\text{-}kind}\ S\ a)\ s_2$ **by**(*fastforce dest*:*slice-kind-Call-in-slice*)
from ‹*valid-edge a*› ‹$\mathit{length}\ s_1 = \mathit{length}\ s_2$› ‹$\mathit{transfer}\ (\mathit{kind}\ a)\ s_1 = s_1'$›
have $\mathit{length}\ s_1' = \mathit{length}\ (\mathit{transfer}\ (\mathit{slice\text{-}kind}\ S\ a)\ s_2)$
by(*fastforce intro*:*length-transfer-kind-slice-kind*)
with ‹$\mathit{pred}\ (\mathit{slice\text{-}kind}\ S\ a)\ s_2$› ‹*valid-edge a*› ‹$\mathit{kind}\ a = Q{:}r{\hookrightarrow_p}fs$›
‹$\forall m \in \mathit{set}\ (tl\ ms_1).\ \exists m'.\ \mathit{call\text{-}of\text{-}return\text{-}node}\ m\ m' \wedge m' \in \lfloor \mathit{HRB\text{-}slice}\ S \rfloor_{CFG}$›
‹$\mathit{hd}\ ms_1 \in \lfloor \mathit{HRB\text{-}slice}\ S \rfloor_{CFG}$› ‹$\mathit{hd}\ ms_1 = \mathit{sourcenode}\ a$›
‹$\mathit{length}\ ms_1 = \mathit{length}\ s_1$› ‹$\mathit{length}\ s_1 = \mathit{length}\ s_2$› ‹*valid-edge a'*›
‹$ms_1' = (\mathit{targetnode}\ a)\#(\mathit{targetnode}\ a')\#tl\ ms_1$› ‹$a' \in \mathit{get\text{-}return\text{-}edges}\ a$›
‹$\forall m \in \mathit{set}\ (tl\ ms_2).\ \mathit{return\text{-}node}\ m$›
have $S,\mathit{slice\text{-}kind}\ S \vdash (ms_1,s_2)\ -a\rightarrow\ (ms_1',\mathit{transfer}\ (\mathit{slice\text{-}kind}\ S\ a)\ s_2)$
by(*auto intro*:*observable-move.observable-move-call*)
with ‹$S,\mathit{slice\text{-}kind}\ S \vdash (ms_2,s_2)\ {=}as{\Rightarrow_\tau}\ (ms_1,s_2)$›
have $S,\mathit{slice\text{-}kind}\ S \vdash (ms_2,s_2)\ {=}as@[a]{\Rightarrow}\ (ms_1',\mathit{transfer}\ (\mathit{slice\text{-}kind}\ S\ a)\ s_2)$
by(*rule observable-moves-snoc*)
from ‹$\forall m \in \mathit{set}\ ms_1.\ \mathit{valid\text{-}node}\ m$› ‹$ms_1' = (\mathit{targetnode}\ a)\#(\mathit{targetnode}\ a')\#tl\ ms_1$›
‹*valid-edge a*› ‹*valid-edge a'*›
have $\forall m \in \mathit{set}\ ms_1'.\ \mathit{valid\text{-}node}\ m$ **by**(*cases* ms_1) *auto*
from ‹$\mathit{kind}\ a = Q{:}r{\hookrightarrow_p}fs$› ‹$\mathit{sourcenode}\ a \in \lfloor \mathit{HRB\text{-}slice}\ S \rfloor_{CFG}$›
have *cf2'*:$\exists\,cf_2'.\ \mathit{transfer}\ (\mathit{slice\text{-}kind}\ S\ a)\ s_2 = cf_2'\#s_2 \wedge \mathit{snd}\ cf_2' = r$
by(*auto dest*:*slice-kind-Call-in-slice*)
with ‹$\forall i{<}\mathit{length}\ ms_2.\ \mathit{snd}\ (s_1\ !\ (\mathit{length}\ msx + i)) = \mathit{snd}\ (s_2\ !\ i)$›

$\langle length\ ms_1' = length\ s_1' \rangle$ $\langle msx = [] \rangle$ $\langle length\ ms_1 = length\ ms_2 \rangle$
$\langle length\ ms_1 = length\ s_1 \rangle$
have $\forall i{<}length\ ms_1'.\ snd\ (s_1'\ !\ i) = snd\ (transfer\ (slice\text{-}kind\ S\ a)\ s_2\ !\ i)$
by *auto*(*case-tac i*,*auto*)
have $\forall\, V \in rv\ S\ (CFG\text{-}node\ (targetnode\ a'))$.
$V \in rv\ S\ (CFG\text{-}node\ (sourcenode\ a))$
proof
fix V **assume** $V \in rv\ S\ (CFG\text{-}node\ (targetnode\ a'))$
then obtain *as* n' **where** $targetnode\ a' -as\rightarrow_\iota * parent\text{-}node\ n'$
and $n' \in HRB\text{-}slice\ S$ **and** $V \in Use_{SDG}\ n'$
and $\forall\, n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge parent\text{-}node\ n'' \in set\ (sourcenodes\ as)$
$\longrightarrow V \notin Def_{SDG}\ n''$ **by**(*fastforce elim*:*rvE*)
from $\langle valid\text{-}edge\ a \rangle$ $\langle a' \in get\text{-}return\text{-}edges\ a \rangle$
obtain a'' **where** *valid-edge* a'' **and** $sourcenode\ a'' = sourcenode\ a$
and $targetnode\ a'' = targetnode\ a'$ **and** *intra-kind*(*kind* a'')
by $-$(*drule call-return-node-edge*,*auto simp*:*intra-kind-def*)
with $\langle targetnode\ a' -as\rightarrow_\iota * parent\text{-}node\ n' \rangle$
have $sourcenode\ a -a''\#as\rightarrow_\iota * parent\text{-}node\ n'$
by(*fastforce intro*:*Cons-path simp*:*intra-path-def*)
from $\langle sourcenode\ a'' = sourcenode\ a \rangle$ $\langle valid\text{-}edge\ a \rangle$ $\langle kind\ a = Q{:}r{\hookrightarrow_p}fs \rangle$
have $\forall\, n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge parent\text{-}node\ n'' = sourcenode\ a''$
$\longrightarrow V \notin Def_{SDG}\ n''$
by(*fastforce dest*:*SDG-Def-parent-Def call-source-Def-empty*)
with $\langle \forall\, n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge parent\text{-}node\ n'' \in set\ (sourcenodes\ as)$
$\longrightarrow V \notin Def_{SDG}\ n'' \rangle$
have $\forall\, n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge parent\text{-}node\ n'' \in set\ (sourcenodes\ (a''\#as))$

$\longrightarrow V \notin Def_{SDG}\ n''$ **by**(*fastforce simp*:*sourcenodes-def*)
with $\langle sourcenode\ a -a''\#as\rightarrow_\iota * parent\text{-}node\ n' \rangle$ $\langle n' \in HRB\text{-}slice\ S \rangle$
$\langle V \in Use_{SDG}\ n' \rangle$
show $V \in rv\ S\ (CFG\text{-}node\ (sourcenode\ a))$ **by**(*fastforce intro*:*rvI*)
qed
have $\forall\, V \in rv\ S\ (CFG\text{-}node\ (targetnode\ a))$.
$(empty(ins\ [:=]\ params\ fs\ (fst\ cf_1)))\ V =$
$state\text{-}val\ (transfer\ (slice\text{-}kind\ S\ a)\ s_2)\ V$
proof
fix V **assume** $V \in rv\ S\ (CFG\text{-}node\ (targetnode\ a))$
from $\langle sourcenode\ a \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG} \rangle$ $\langle kind\ a = Q{:}r{\hookrightarrow_p}fs \rangle$
$\langle (THE\ ins.\ \exists\, outs.\ (p,ins,outs) \in set\ procs) = ins \rangle$
have *eq*:$fst\ (hd\ (transfer\ (slice\text{-}kind\ S\ a)\ s_2)) =$
$empty(ins\ [:=]\ params\ (cspp\ (targetnode\ a)\ (HRB\text{-}slice\ S)\ fs)\ (fst\ cf_2))$
by(*auto dest*:*slice-kind-Call-in-slice*)
show $(empty(ins\ [:=]\ params\ fs\ (fst\ cf_1)))\ V =$
$state\text{-}val\ (transfer\ (slice\text{-}kind\ S\ a)\ s_2)\ V$
proof(*cases* $V \in set\ ins$)
case *True*
then obtain i **where** $V = ins!i$ **and** $i < length\ ins$
by(*auto simp*:*in-set-conv-nth*)
from $\langle valid\text{-}edge\ a \rangle$ $\langle kind\ a = Q{:}r{\hookrightarrow_p}fs \rangle$ $\langle (p,ins,outs) \in set\ procs \rangle$

⟨$i < length\ ins$⟩
have *valid-SDG-node* (*Formal-in* (*targetnode a*,*i*)) **by** *fastforce*
from ⟨*valid-edge a*⟩ ⟨*kind a* = *Q*:$r \hookrightarrow_p fs$⟩ **have** *get-proc*(*targetnode a*) = *p*
by(*rule get-proc-call*)
with ⟨*valid-SDG-node* (*Formal-in* (*targetnode a*,*i*))⟩
⟨(*p*,*ins*,*outs*) ∈ *set procs*⟩ ⟨*V* = *ins*!*i*⟩
have *V* ∈ Def_{SDG} (*Formal-in* (*targetnode a*,*i*))
by(*fastforce intro*:*Formal-in-SDG-Def*)
from ⟨*V* ∈ *rv S* (*CFG-node* (*targetnode a*))⟩ **obtain** *as*′ *nx*
where *targetnode a* $-as'\rightarrow_\iota *$ *parent-node nx*
and *nx* ∈ *HRB-slice S* **and** *V* ∈ Use_{SDG} *nx*
and ∀ *n*′′. *valid-SDG-node n*′′ ∧
parent-node n′′ ∈ *set* (*sourcenodes as*′) ⟶ *V* ∉ Def_{SDG} *n*′′
by(*fastforce elim*:*rvE*)
with ⟨*valid-SDG-node* (*Formal-in* (*targetnode a*,*i*))⟩
⟨*V* ∈ Def_{SDG} (*Formal-in* (*targetnode a*,*i*))⟩
have *targetnode a* = *parent-node nx*
apply(*auto simp*:*intra-path-def sourcenodes-def*)
apply(*erule path.cases*) **apply** *fastforce*
apply(*erule-tac x*=*Formal-in* (*targetnode a*,*i*) **in** *allE*) **by** *fastforce*
with ⟨*V* ∈ Use_{SDG} *nx*⟩ **have** *V* ∈ *Use* (*targetnode a*)
by(*fastforce intro*:*SDG-Use-parent-Use*)
with ⟨*valid-edge a*⟩ **have** *V* ∈ Use_{SDG} (*CFG-node* (*targetnode a*))
by(*auto intro*!:*CFG-Use-SDG-Use*)
from ⟨*targetnode a* = *parent-node nx*⟩[*THEN sym*] ⟨*valid-edge a*⟩
have *parent-node* (*Formal-in* (*targetnode a*,*i*)) $-[]\rightarrow_\iota *$ *parent-node nx*
by(*fastforce intro*:*empty-path simp*:*intra-path-def*)
with ⟨*V* ∈ Def_{SDG} (*Formal-in* (*targetnode a*,*i*))⟩
⟨*V* ∈ Use_{SDG} (*CFG-node* (*targetnode a*))⟩ ⟨*targetnode a* = *parent-node nx*⟩
have *Formal-in* (*targetnode a*,*i*) *influences V in* (*CFG-node* (*targetnode a*))
by(*fastforce simp*:*data-dependence-def sourcenodes-def*)
hence *ddep*:*Formal-in* (*targetnode a*,*i*) $s-V\rightarrow_{dd}$ (*CFG-node* (*targetnode a*))
by(*rule sum-SDG-ddep-edge*)
from ⟨*targetnode a* = *parent-node nx*⟩ ⟨*nx* ∈ *HRB-slice S*⟩
have *CFG-node* (*targetnode a*) ∈ *HRB-slice S*
by(*fastforce dest*:*valid-SDG-node-in-slice-parent-node-in-slice*)
hence *Formal-in* (*targetnode a*,*i*) ∈ *HRB-slice S*
proof(*induct CFG-node* (*targetnode a*) *rule*:*HRB-slice-cases*)
case (*phase1 nx*)
with *ddep* **show** *?case*
by(*fastforce intro*:*ddep-slice1 combine-SDG-slices.combSlice-refl*
simp:*HRB-slice-def*)
next
case (*phase2 nx n*′ *n*′′ *p*)
from ⟨*CFG-node* (*targetnode a*) ∈ *sum-SDG-slice2 n*′⟩ *ddep*
have *Formal-in* (*targetnode a*,*i*) ∈ *sum-SDG-slice2 n*′
by(*fastforce intro*:*ddep-slice2*)
with ⟨*n*′′ $s-p\rightarrow_{ret}$ *CFG-node* (*parent-node n*′)⟩ ⟨*n*′ ∈ *sum-SDG-slice1 nx*⟩

⟨$nx \in S$⟩
show *?case* **by**(*fastforce intro*:*combine-SDG-slices.combSlice-Return-parent-node simp*:*HRB-slice-def*)
qed
from ⟨*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$⟩ ⟨*kind a* = *Q*:*r*↪$_p$*fs*⟩
have *slice-kind*:*slice-kind S a* =
Q:*r*↪$_p$(*cspp* (*targetnode a*) (*HRB-slice S*) *fs*)
by(*rule slice-kind-Call-in-slice*)
from ⟨*valid-edge a*⟩ ⟨*kind a* = *Q*:*r*↪$_p$*fs*⟩ ⟨(*p*,*ins*,*outs*) ∈ *set procs*⟩
have *length fs* = *length ins* **by**(*rule CFG-call-edge-length*)
from ⟨*Formal-in* (*targetnode a*,*i*) ∈ *HRB-slice S*⟩
⟨*length fs* = *length ins*⟩ ⟨*i* < *length ins*⟩
have *cspp*:(*cspp* (*targetnode a*) (*HRB-slice S*) *fs*)!*i* = *fs*!*i*
by(*fastforce intro*:*csppa-Formal-in-in-slice simp*:*cspp-def*)
from ⟨*i* < *length ins*⟩ ⟨*length fs* = *length ins*⟩
have (*params* (*cspp* (*targetnode a*) (*HRB-slice S*) *fs*) (*fst cf*$_2$))!*i* =
((*cspp* (*targetnode a*) (*HRB-slice S*) *fs*)!*i*) (*fst cf*$_2$)
by(*fastforce intro*:*params-nth*)
with *cspp*
have *eq*:(*params* (*cspp* (*targetnode a*) (*HRB-slice S*) *fs*) (*fst cf*$_2$))!*i* =
(*fs*!*i*) (*fst cf*$_2$) **by** *simp*
from ⟨*valid-edge a*⟩ ⟨*kind a* = *Q*:*r*↪$_p$*fs*⟩ ⟨(*p*,*ins*,*outs*) ∈ *set procs*⟩
have (*THE ins*. ∃ *outs*. (*p*,*ins*,*outs*) ∈ *set procs*) = *ins*
by(*rule formal-in-THE*)
with *slice-kind*
have *fst* (*hd* (*transfer* (*slice-kind S a*) *s*$_2$)) =
empty(*ins* [:=] *params* (*cspp* (*targetnode a*) (*HRB-slice S*) *fs*) (*fst cf*$_2$))
by *simp*
moreover
from ⟨(*p*,*ins*,*outs*) ∈ *set procs*⟩ **have** *distinct ins*
by(*rule distinct-formal-ins*)
ultimately have *state-val* (*transfer* (*slice-kind S a*) *s*$_2$) *V* =
(*params* (*cspp* (*targetnode a*) (*HRB-slice S*) *fs*) (*fst cf*$_2$))!*i*
using ⟨*V* = *ins*!*i*⟩ ⟨*i* < *length ins*⟩ ⟨*length fs* = *length ins*⟩
by(*fastforce intro*:*fun-upds-nth*)
with *eq*
have *2*:*state-val* (*transfer* (*slice-kind S a*) *s*$_2$) *V* = (*fs*!*i*) (*fst cf*$_2$)
by *simp*
from ⟨*V* = *ins*!*i*⟩ ⟨*i* < *length ins*⟩ ⟨*length fs* = *length ins*⟩
⟨*distinct ins*⟩
have *empty*(*ins* [:=] *params fs* (*fst cf*$_1$)) *V* = (*params fs* (*fst cf*$_1$))!*i*
by(*fastforce intro*:*fun-upds-nth*)
with ⟨*i* < *length ins*⟩ ⟨*length fs* = *length ins*⟩
have *1*:*empty*(*ins* [:=] *params fs* (*fst cf*$_1$)) *V* = (*fs*!*i*) (*fst cf*$_1$)
by(*fastforce intro*:*params-nth*)
from ⟨∀ *i* < *length ms*$_2$. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl ms*$_2$)!*i*)).
(*fst* (*s*$_1$!(*length msx* + *i*))) *V* = (*fst* (*s*$_2$!*i*)) *V*⟩
have *rv*:∀ *V*∈*rv S* (*CFG-node* (*sourcenode a*)). *fst cf*$_1$ *V* = *fst cf*$_2$ *V*
by(*erule-tac x*=*0* **in** *allE*) *auto*

from ‹*valid-edge a*› ‹*kind a = Q:r↪pfs*› ‹*(p,ins,outs) ∈ set procs*›
‹*i < length ins*› **have** ∀ *V*∈(*ParamUses* (*sourcenode a*)!*i*).
V ∈ *Use*$_{SDG}$ (*Actual-in* (*sourcenode a*,*i*))
by(*fastforce intro*:*Actual-in-SDG-Use*)
with ‹*valid-edge a*› **have** ∀ *V*∈(*ParamUses* (*sourcenode a*)!*i*).
V ∈ *Use*$_{SDG}$ (*CFG-node* (*sourcenode a*))
by(*auto intro*!:*CFG-Use-SDG-Use dest*:*SDG-Use-parent-Use*)
moreover
from ‹*valid-edge a*› **have** *parent-node* (*CFG-node* (*sourcenode a*)) −[]→$_\iota$*
parent-node (*CFG-node* (*sourcenode a*))
by(*fastforce intro*:*empty-path simp*:*intra-path-def*)
ultimately
have ∀ *V*∈(*ParamUses* (*sourcenode a*)!*i*). *V* ∈ *rv S* (*CFG-node* (*sourcenode a*))
using ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*valid-edge a*›
by(*fastforce intro*:*rvI simp*:*SDG-to-CFG-set-def sourcenodes-def*)
with *rv* **have** ∀ *V* ∈ (*ParamUses* (*sourcenode a*))!*i*. *fst cf*$_1$ *V* = *fst cf*$_2$ *V*
by *fastforce*
with ‹*valid-edge a*› ‹*kind a = Q:r↪pfs*› ‹*i < length ins*›
‹(*p*,*ins*,*outs*) ∈ *set procs*› ‹*pred* (*kind a*) *s*$_1$› ‹*pred* (*kind a*) *s*$_2$›
have (*params fs* (*fst cf*$_1$))!*i* = (*params fs* (*fst cf*$_2$))!*i*
by(*fastforce dest*!:*CFG-call-edge-params*)
moreover
from ‹*i < length ins*› ‹*length fs = length ins*›
have (*params fs* (*fst cf*$_1$))!*i* = (*fs*!*i*) (*fst cf*$_1$)
and (*params fs* (*fst cf*$_2$))!*i* = (*fs*!*i*) (*fst cf*$_2$)
by(*auto intro*:*params-nth*)
ultimately show *?thesis* **using** *1 2* **by** *simp*
next
case *False*
with *eq* **show** *?thesis* **by**(*fastforce simp*:*fun-upds-notin*)
qed
qed
with ‹∀ *i* < *length ms*$_2$. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl ms*$_2$)!*i*)).
(*fst* (*s*$_1$!(*length msx* + *i*))) *V* = (*fst* (*s*$_2$!*i*)) *V*› *cf2*′ ‹*tl ms*$_2$ = *tl ms*$_1$›
‹*length ms*$_2$ = *length s*$_2$› ‹*length ms*$_1$ = *length s*$_1$› ‹*length s*$_1$ = *length s*$_2$›
‹*ms*$_1$′ = (*targetnode a*)#(*targetnode a*′)#*tl ms*$_1$›
‹∀ *V* ∈ *rv S* (*CFG-node* (*targetnode a*′)). *V* ∈ *rv S* (*CFG-node* (*sourcenode a*))›
have ∀ *i*<*length ms*$_1$′. ∀ *V*∈*rv S* (*CFG-node* ((*targetnode a* # *tl ms*$_1$′)!*i*)).
(*fst* (*s*$_1$′!(*length* [] + *i*))) *V* = (*fst* (*transfer* (*slice-kind S a*) *s*$_2$!*i*)) *V*
apply *clarsimp* **apply**(*case-tac i*) **apply** *auto*
apply(*erule-tac x*=*nat* **in** *allE*)
apply(*case-tac nat*) **apply** *auto* **done**
with ‹∀ *m* ∈ *set ms*$_2$. *valid-node m*› ‹∀ *m* ∈ *set ms*$_1$′. *valid-node m*›
‹*length ms*$_2$ = *length s*$_2$› ‹*length s*$_1$′ = *length* (*transfer* (*slice-kind S a*) *s*$_2$)›
‹*length ms*$_1$′ = *length s*$_1$′› ‹*ms*$_1$′ = (*targetnode a*)#(*targetnode a*′)#*tl ms*$_1$›
‹*get-proc mx* = *get-proc* (*hd ms*$_2$)› ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹∀ *m* ∈ *set* (*tl ms*$_1$). ∃ *m*′. *call-of-return-node m m*′ ∧ *m*′ ∈ ⌊*HRB-slice S*⌋$_{CFG}$›

‹return-node (targetnode a')› ‹∀ m ∈ set (tl ms$_1$). return-node m›
‹call-of-return-node (targetnode a') (sourcenode a)›
‹∀ i<length ms$_1$'. snd (s$_1$' ! i) = snd (transfer (slice-kind S a) s$_2$! i)›
have ((ms$_1$',s$_1$'),(ms$_1$',transfer (slice-kind S a) s$_2$)) ∈ WS S
by(*fastforce intro*!:*WSI*)
with ‹S,slice-kind S ⊢ (ms$_2$,s$_2$) =as@[a]⇒ (ms$_1$',transfer (slice-kind S a) s$_2$)›
show *?case* **by** *blast*
next
case (*observable-move-return a s$_1$ s$_1$' Q p f' ms$_1$ S ms$_1$'*)
from ‹s$_1$ ≠ []› ‹s$_2$ ≠ []› **obtain** *cf$_1$ cfs$_1$ cf$_2$ cfs$_2$* **where** [*simp*]:*s$_1$ = cf$_1$#cfs$_1$*
and [*simp*]:*s$_2$ = cf$_2$#cfs$_2$* **by**(*cases s$_1$,auto,cases s$_2$,fastforce+*)
from ‹*length ms$_1$ = length s$_1$*› ‹*s$_1$ ≠ []*› **have** [*simp*]:*ms$_1$ ≠ []* **by**(*cases ms$_1$*) *auto*
from ‹*length ms$_2$ = length s$_2$*› ‹*s$_2$ ≠ []*› **have** [*simp*]:*ms$_2$ ≠ []* **by**(*cases ms$_2$*) *auto*
from ‹∀ m ∈ set (tl ms$_1$). ∃ m'. call-of-return-node m m' ∧ m' ∈ ⌊HRB-slice S⌋$_{CFG}$›
‹*hd ms$_1$ = sourcenode a*› ‹*ms$_1$ = msx@mx#tl ms$_2$*›
‹*msx ≠ [] ⟶ (∃ mx'. call-of-return-node mx mx' ∧ mx' ∉ ⌊HRB-slice S⌋$_{CFG}$)*›
have [*simp*]:*mx = sourcenode a msx = []* **and** [*simp*]:*tl ms$_2$ = tl ms$_1$*
by(*cases msx,auto*)+
hence *length ms$_1$ = length ms$_2$* **by**(*cases ms$_2$*) *auto*
with ‹*length ms$_1$ = length s$_1$*› ‹*length ms$_2$ = length s$_2$*›
have *length s$_1$ = length s$_2$* **by** *simp*
have ∃ *as. S,slice-kind S ⊢ (ms$_2$,s$_2$) =as⇒$_\tau$ (ms$_1$,s$_2$)*
proof(*cases obs-intra (hd ms$_2$) ⌊HRB-slice S⌋$_{CFG}$ = {}*)
case *True*
from ‹*valid-edge a*› ‹*hd ms$_1$ = sourcenode a*› ‹*kind a = Q↩$_p$f'*›
have *method-exit (hd ms$_1$)* **by**(*fastforce simp*:*method-exit-def*)
from ‹∀ m∈*set ms$_2$. valid-node m*› **have** *valid-node (hd ms$_2$)* **by**(*cases ms$_2$*) *auto*
then obtain *asx* **where** *hd ms$_2$ −asx→$_{\sqrt{}}$* (-Exit-)* **by**(*fastforce dest*!:*Exit-path*)
then obtain *as pex* **where** *hd ms$_2$ −as→$_\iota$* pex* **and** *method-exit pex*
by(*fastforce elim*:*valid-Exit-path-intra-path*)
from ‹*hd ms$_2$ −as→$_\iota$* pex*› **have** *get-proc (hd ms$_2$) = get-proc pex*
by(*rule intra-path-get-procs*)
with ‹*get-proc mx = get-proc (hd ms$_2$)*›
have *get-proc mx = get-proc pex* **by** *simp*
with ‹*method-exit (hd ms$_1$)*› ‹ *hd ms$_1$ = sourcenode a*› ‹*method-exit pex*›
have [*simp*]:*pex = hd ms$_1$* **by**(*fastforce intro*:*method-exit-unique*)
from ‹*obs-intra (hd ms$_2$) ⌊HRB-slice S⌋$_{CFG}$ = {}*› ‹*method-exit pex*›
‹*get-proc (hd ms$_2$) = get-proc pex*› ‹*valid-node (hd ms$_2$)*›
‹*length ms$_2$ = length s$_2$*› ‹∀ m∈*set (tl ms$_1$). return-node m*› ‹*ms$_2$ ≠ []*›
obtain *as'*
where *S,slice-kind S ⊢ (hd ms$_2$#tl ms$_2$,s$_2$) =as'⇒$_\tau$ (hd ms$_1$#tl ms$_1$,s$_2$)*
by(*fastforce elim*!:*silent-moves-intra-path-no-obs*[*of - - - s$_2$ tl ms$_2$*]
dest:*hd-Cons-tl*)
with ‹*ms$_2$ ≠ []*› **have** *S,slice-kind S ⊢ (ms$_2$,s$_2$) =as'⇒$_\tau$ (ms$_1$,s$_2$)*
by(*fastforce dest*!:*hd-Cons-tl*)

thus *?thesis* **by** *blast*
next
case *False*
then obtain x **where** $x \in$ *obs-intra* (*hd* ms_2) $\lfloor$*HRB-slice* $S\rfloor_{CFG}$ **by** *fastforce*
hence *obs-intra* (*hd* ms_2) $\lfloor$*HRB-slice* $S\rfloor_{CFG} = \{x\}$
by(*rule obs-intra-singleton-element*)
with ‹$\forall\, m \in$ *set* (*tl* ms_2). $\exists\, m'$. *call-of-return-node* $m\ m' \wedge m' \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$›
have $x\#$*tl* $ms_1 \in$ *obs* ([]@*hd* $ms_2\#$*tl* ms_2) $\lfloor$*HRB-slice* $S\rfloor_{CFG}$
by(*fastforce intro*:*obsI*)
with ‹$ms_2 \neq$ []› **have** $x\#$*tl* $ms_1 \in$ *obs* ms_2 $\lfloor$*HRB-slice* $S\rfloor_{CFG}$
by(*fastforce dest*:*hd-Cons-tl simp del*:*obs.simps*)
with ‹*obs* ms_1 $\lfloor$*HRB-slice* $S\rfloor_{CFG}$ = *obs* ms_2 $\lfloor$*HRB-slice* $S\rfloor_{CFG}$›
have $x\#$*tl* $ms_1 \in$ *obs* ms_1 $\lfloor$*HRB-slice* $S\rfloor_{CFG}$ **by** *simp*
from *this* ‹$\forall\, m{\in}$*set* (*tl* ms_1). *return-node* m›
have $x \in$ *obs-intra* (*hd* ms_1) $\lfloor$*HRB-slice* $S\rfloor_{CFG}$
proof(*rule obsE*)
fix *nsx n nsx' n'*
assume ms_1 = *nsx* @ *n* # *nsx'* **and** *x* # *tl* ms_1 = *n'* # *nsx'*
and *n'* ∈ *obs-intra n* $\lfloor$*HRB-slice* $S\rfloor_{CFG}$
from ‹ms_1 = *nsx* @ *n* # *nsx'›* ‹*x* # *tl* ms_1 = *n'* # *nsx'›* ‹*tl* ms_2 = *tl* ms_1›
have [*simp*]:*nsx* = [] **by**(*cases nsx*) *auto*
with ‹ms_1 = *nsx* @ *n* # *nsx'›* **have** [*simp*]:*n* = *hd* ms_1 **by** *simp*
from ‹*x* # *tl* ms_1 = *n'* # *nsx'›* **have** [*simp*]:*n'* = *x* **by** *simp*
with ‹*n'* ∈ *obs-intra n* $\lfloor$*HRB-slice* $S\rfloor_{CFG}$› **show** *?thesis* **by** *simp*
qed
{ fix *m as* **assume** *hd* ms_1 $-as\rightarrow_\iota*$ *m*
hence *hd* ms_1 $-as\rightarrow*$ *m* **and** ∀ *a* ∈ *set as*. *intra-kind* (*kind a*)
by(*simp-all add*:*intra-path-def*)
hence *m* = *hd* ms_1
proof(*induct hd* ms_1 *as m rule*:*path.induct*)
case (*Cons-path m'' as' m' a'*)
from ‹$\forall\, a{\in}$*set* (*a'* # *as'*). *intra-kind* (*kind a*)›
have *intra-kind* (*kind a'*) **by** *simp*
with ‹*valid-edge a*› ‹*kind a* = *Q*↩$_{pf}$*'*› ‹*valid-edge a'*›
‹*sourcenode a'* = *hd* ms_1› ‹*hd* ms_1 = *sourcenode a*›
have *False* **by**(*fastforce dest*:*return-edges-only simp*:*intra-kind-def*)
thus *?case* **by** *simp*
qed *simp* **}**
with ‹*x* ∈ *obs-intra* (*hd* ms_1) $\lfloor$*HRB-slice* $S\rfloor_{CFG}$›
have *x* = *hd* ms_1 **by**(*fastforce elim*:*obs-intraE*)
with ‹*x* ∈ *obs-intra* (*hd* ms_2) $\lfloor$*HRB-slice* $S\rfloor_{CFG}$› ‹*length* ms_2 = *length* s_2›
‹$\forall\, m{\in}$*set* (*tl* ms_1). *return-node* m› ‹$ms_2 \neq$ []›
obtain *as* **where** *S*,*slice-kind S* ⊢ (*hd* $ms_2\#$*tl* ms_2,s_2) =*as*⇒$_\tau$ (*hd* $ms_1\#$*tl* ms_1,s_2)
by(*fastforce elim*!:*silent-moves-intra-path-obs*[*of - - - s*$_2$ *tl* ms_2]
dest:*hd-Cons-tl*)
with ‹$ms_2 \neq$ []› **have** *S*,*slice-kind S* ⊢ (ms_2,s_2) =*as*⇒$_\tau$ (ms_1,s_2)
by(*fastforce dest*!:*hd-Cons-tl*)

thus *?thesis* **by** *blast*
qed
then obtain *as* **where** S,*slice-kind* $S \vdash (ms_2,s_2) =as\Rightarrow_\tau (ms_1,s_2)$ **by** *blast*
from ‹$ms_1' = tl\ ms_1$› ‹*length* $s_1 = Suc(length\ s_1')$›
‹*length* $ms_1 = length\ s_1$› **have** *length* $ms_1' = length\ s_1'$ **by** *simp*
from ‹*valid-edge a*› ‹*kind* $a = Q\hookleftarrow_p f'$› **obtain** $a''\ Q'\ r'\ fs'$ **where** *valid-edge* a''
and *kind* $a'' = Q':r'\hookrightarrow_p fs'$ **and** $a \in$ *get-return-edges* a''
by $-$(*drule return-needs-call*,*auto*)
then obtain *ins outs* **where** (p,*ins*,*outs*) $\in$ *set procs*
by(*fastforce dest*!:*callee-in-procs*)
from ‹*length* $s_1 = Suc(length\ s_1')$› ‹$s_1' \neq []$›
obtain *cfx cfsx* **where** [*simp*]:$cfs_1 = cfx\#cfsx$ **by**(*cases* cfs_1) *auto*
with ‹*length* $s_1 = length\ s_2$› **obtain** *cfx' cfsx'* **where** [*simp*]:$cfs_2 = cfx'\#cfsx'$
by(*cases* cfs_2) *auto*
from ‹*length* $ms_1 = length\ s_1$› **have** $tl\ ms_1 = []@hd(tl\ ms_1)\#tl(tl\ ms_1)$
by(*auto simp*:*length-Suc-conv*)
from ‹*kind* $a = Q\hookleftarrow_p f'$› ‹*transfer* (*kind a*) $s_1 = s_1'$›
have $s_1' = (f'\ (fst\ cf_1)\ (fst\ cfx)$,*snd cfx*)#*cfsx* **by** *simp*
from ‹*valid-edge a*› ‹*kind* $a = Q\hookleftarrow_p f'$› ‹(p,*ins*,*outs*) $\in$ *set procs*›
have $f'\ (fst\ cf_1)\ (fst\ cfx) =$
(*fst cfx*)(*ParamDefs* (*targetnode a*) [:=] *map* (*fst* cf_1) *outs*)
by(*rule CFG-return-edge-fun*)
with ‹$s_1' = (f'\ (fst\ cf_1)\ (fst\ cfx)$,*snd cfx*)#*cfsx*›
have [*simp*]:$s_1' =$
((*fst cfx*)(*ParamDefs* (*targetnode a*) [:=] *map* (*fst* cf_1) *outs*),*snd cfx*)#*cfsx*
by *simp*
have *pred* (*slice-kind S a*) s_2
proof(*cases sourcenode a* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$)
case *True*
from ‹*valid-edge a*› **have** *valid-node* (*sourcenode a*) **by** *simp*
hence *sourcenode a* $-[]\rightarrow_\iota *$ *sourcenode a*
by(*fastforce intro*:*empty-path simp*:*intra-path-def*)
with ‹*sourcenode a* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
have $\forall\ V.\ V \in Use_{SDG}$ (*CFG-node* (*sourcenode a*))
$\longrightarrow V \in rv\ S$ (*CFG-node* (*sourcenode a*))
by *auto*(*rule rvI*,*auto simp*:*SDG-to-CFG-set-def sourcenodes-def*)
with ‹*valid-node* (*sourcenode a*)›
have $\forall\ V \in Use$ (*sourcenode a*). $V \in rv\ S$ (*CFG-node* (*sourcenode a*))
by(*fastforce intro*:*CFG-Use-SDG-Use*)
from ‹$\forall\ i < length\ ms_2.\ \forall\ V \in rv\ S$ (*CFG-node* ($(mx\#tl\ ms_2)!i$)).
(*fst* ($s_1!(length\ msx + i)$)) V = (*fst* ($s_2!i$)) V› ‹*length* $ms_2 = length\ s_2$›
have $\forall\ V \in rv\ S$ (*CFG-node mx*). (*fst* (s_1 ! *length msx*)) V = *state-val* s_2 V
by(*cases* ms_2) *auto*
with ‹$\forall\ V \in Use$ (*sourcenode a*). $V \in rv\ S$ (*CFG-node* (*sourcenode a*))›
have $\forall\ V \in Use$ (*sourcenode a*). *fst* cf_1 V = *fst* cf_2 V **by** *fastforce*
moreover
from ‹$\forall\ i$<*length* ms_2. *snd* (s_1 ! (*length msx* + i)) = *snd* (s_2 ! i)›
have *snd* (*hd* s_1) = *snd* (*hd* s_2) **by**(*erule-tac* x=*0* **in** *allE*) *auto*

ultimately have *pred (kind a) s_2*
using ‹*valid-edge a*› ‹*pred (kind a) s_1*› ‹*length s_1 = length s_2*›
by(*fastforce intro*:*CFG-edge-Uses-pred-equal*)
with ‹*valid-edge a*› ‹*kind a = Q↩$_p$f'*› ‹*(p,ins,outs) ∈ set procs*›
‹*sourcenode a ∈ ⌊HRB-slice S⌋$_{CFG}$*›
show *?thesis* **by**(*fastforce dest*:*slice-kind-Return-in-slice*)
next
case *False*
with ‹*kind a = Q↩$_p$f'*› **have** *slice-kind S a = (λcf. True)↩$_p$(λcf cf'. cf')*
by −(*rule slice-kind-Return*)
thus *?thesis* **by** *simp*
qed
from ‹*valid-edge a*› ‹*length s_1 = length s_2*› ‹*transfer (kind a) s_1 = s_1'*›
have *length s_1' = length (transfer (slice-kind S a) s_2)*
by(*fastforce intro*:*length-transfer-kind-slice-kind*)
with ‹*pred (slice-kind S a) s_2*› ‹*valid-edge a*› ‹*kind a = Q↩$_p$f'*›
‹*∀ m ∈ set (tl ms_1). ∃ m'. call-of-return-node m m' ∧ m' ∈ ⌊HRB-slice S⌋$_{CFG}$*›
‹*hd ms_1 = sourcenode a*›
‹*length ms_1 = length s_1*› ‹*length s_1 = length s_2*›
‹*ms_1' = tl ms_1*› ‹*hd(tl ms_1) = targetnode a*› ‹*∀ m ∈ set (tl ms_1). return-node m*›
have *S,slice-kind S ⊢ (ms_1,s_2) −a→ (ms_1',transfer (slice-kind S a) s_2)*
by(*fastforce intro*!:*observable-move.observable-move-return*)
with ‹*S,slice-kind S ⊢ (ms_2,s_2) =as⇒$_\tau$ (ms_1,s_2)*›
have *S,slice-kind S ⊢ (ms_2,s_2) =as@[a]⇒ (ms_1',transfer (slice-kind S a) s_2)*
by(*rule observable-moves-snoc*)
from ‹*∀ m ∈ set ms_1. valid-node m*› ‹*ms_1' = tl ms_1*›
have *∀ m ∈ set ms_1'. valid-node m* **by**(*cases ms_1*) *auto*
from ‹*length ms_1' = length s_1'*› **have** *ms_1' = []@hd ms_1'#tl ms_1'*
by(*cases ms_1'*) *auto*
from ‹*∀ i<length ms_2. snd (s_1 ! (length msx + i)) = snd (s_2 ! i)*›
‹*length ms_1 = length ms_2*› ‹*length ms_1 = length s_1*›
have *snd cfx = snd cfx'* **by**(*erule-tac x=1* **in** *allE*) *auto*
from ‹*valid-edge a*› ‹*kind a = Q↩$_p$f'*› ‹*(p,ins,outs) ∈ set procs*›
have *cf2'*:*∃ cf_2'. transfer (slice-kind S a) s_2 = cf_2'#cfsx' ∧ snd cf_2' = snd cfx'*
by(*cases cfx'*,*cases sourcenode a ∈ ⌊HRB-slice S⌋$_{CFG}$*,
auto dest:*slice-kind-Return slice-kind-Return-in-slice*)
with ‹*∀ i<length ms_2. snd (s_1 ! (length msx + i)) = snd (s_2 ! i)*›
‹*length ms_1' = length s_1'*› ‹*msx = []*› ‹*length ms_1 = length ms_2*›
‹*length ms_1 = length s_1*› ‹*snd cfx = snd cfx'*›
have *∀ i<length ms_1'. snd (s_1' ! i) = snd (transfer (slice-kind S a) s_2 ! i)*
apply *auto* **apply**(*case-tac i*) **apply** *auto*
by(*erule-tac x=Suc(Suc nat)* **in** *allE*) *auto*
from ‹*∀ m ∈ set (tl ms_1). ∃ m'. call-of-return-node m m' ∧ m' ∈ ⌊HRB-slice S⌋$_{CFG}$*›
have *∀ m ∈ set (tl (tl ms_1)).*
∃ m'. call-of-return-node m m' ∧ m' ∈ ⌊HRB-slice S⌋$_{CFG}$
by(*cases tl ms_1*) *auto*

from ‹$\forall\, m \in set\ (tl\ ms_1).\ return\text{-}node\ m$›
have $\forall\, m \in set\ (tl\ (tl\ ms_1)).\ return\text{-}node\ m$ **by**(*cases tl* ms_1 *auto*
have $\forall\, V{\in}rv\ S\ (CFG\text{-}node\ (hd\ (tl\ ms_1)))$.
$(fst\ cfx)(ParamDefs\ (targetnode\ a)\ [:=]\ map\ (fst\ cf_1)\ outs)\ V =$
$state\text{-}val\ (transfer\ (slice\text{-}kind\ S\ a)\ s_2)\ V$
proof
fix V **assume** $V{\in}rv\ S\ (CFG\text{-}node\ (hd\ (tl\ ms_1)))$
with ‹$hd(tl\ ms_1) = targetnode\ a$› **have** $V{\in}rv\ S\ (CFG\text{-}node\ (targetnode\ a))$
by *simp*
show $(fst\ cfx)(ParamDefs\ (targetnode\ a)\ [:=]\ map\ (fst\ cf_1)\ outs)\ V =$
$state\text{-}val\ (transfer\ (slice\text{-}kind\ S\ a)\ s_2)\ V$
proof(*cases* $V \in set\ (ParamDefs\ (targetnode\ a))$)
case *True*
then obtain i **where** $V = (ParamDefs\ (targetnode\ a))!i$
and $i < length(ParamDefs\ (targetnode\ a))$
by(*auto simp*:*in-set-conv-nth*)
moreover
from ‹*valid-edge a*› ‹$kind\ a = Q{\hookleftarrow}_p f'$› ‹$(p,ins,outs) \in set\ procs$›
have *length*:$length(ParamDefs\ (targetnode\ a)) = length\ outs$
by(*fastforce intro*:*ParamDefs-return-target-length*)
from ‹*valid-edge a*› ‹$kind\ a = Q{\hookleftarrow}_p f'$› ‹$(p,ins,outs) \in set\ procs$›
‹$i < length(ParamDefs\ (targetnode\ a))$›
‹$length(ParamDefs\ (targetnode\ a)) = length\ outs$›
have *valid-SDG-node* $(Actual\text{-}out(targetnode\ a,i))$ **by** *fastforce*
with ‹$V = (ParamDefs\ (targetnode\ a))!i$›
have $V \in Def_{SDG}\ (Actual\text{-}out(targetnode\ a,i))$
by(*fastforce intro*:*Actual-out-SDG-Def*)
from ‹$V \in rv\ S\ (CFG\text{-}node\ (targetnode\ a))$› **obtain** $as'\ nx$
where $targetnode\ a\ {-as'}{\rightarrow_\iota}*\ parent\text{-}node\ nx$
and $nx \in HRB\text{-}slice\ S$ **and** $V \in Use_{SDG}\ nx$
and $\forall\, n''.\ valid\text{-}SDG\text{-}node\ n'' \wedge$
$parent\text{-}node\ n'' \in set\ (sourcenodes\ as') \longrightarrow V \notin Def_{SDG}\ n''$
by(*fastforce elim*:*rvE*)
with ‹$valid\text{-}SDG\text{-}node\ (Actual\text{-}out(targetnode\ a,i))$›
‹$V \in Def_{SDG}\ (Actual\text{-}out(targetnode\ a,i))$›
have $targetnode\ a = parent\text{-}node\ nx$
apply(*auto simp*:*intra-path-def sourcenodes-def*)
apply(*erule path.cases*) **apply** *fastforce*
apply(*erule-tac* $x{=}(Actual\text{-}out(targetnode\ a,i))$ **in** *allE*) **by** *fastforce*
with ‹$V \in Use_{SDG}\ nx$› **have** $V \in Use\ (targetnode\ a)$
by(*fastforce intro*:*SDG-Use-parent-Use*)
with ‹*valid-edge a*› **have** $V \in Use_{SDG}\ (CFG\text{-}node\ (targetnode\ a))$
by(*auto intro*!:*CFG-Use-SDG-Use*)
from ‹$targetnode\ a = parent\text{-}node\ nx$›[*THEN sym*] ‹*valid-edge a*›
have $parent\text{-}node\ (Actual\text{-}out(targetnode\ a,i))\ {-[]}{\rightarrow_\iota}*\ parent\text{-}node\ nx$
by(*fastforce intro*:*empty-path simp*:*intra-path-def*)
with ‹$V \in Def_{SDG}\ (Actual\text{-}out(targetnode\ a,i))$›
‹$V \in Use_{SDG}\ (CFG\text{-}node\ (targetnode\ a))$› ‹$targetnode\ a = parent\text{-}node\ nx$›
have $Actual\text{-}out(targetnode\ a,i)$ *influences* V *in* $(CFG\text{-}node\ (targetnode\ a))$

by(*fastforce simp*:*data-dependence-def sourcenodes-def*)
hence *ddep*:*Actual-out*(*targetnode a*,*i*) $s-V\rightarrow_{dd}$ (*CFG-node* (*targetnode a*))
by(*rule sum-SDG-ddep-edge*)
from ‹*targetnode a* = *parent-node nx*› ‹*nx* ∈ *HRB-slice S*›
have *CFG-node* (*targetnode a*) ∈ *HRB-slice S*
by(*fastforce dest*:*valid-SDG-node-in-slice-parent-node-in-slice*)
hence *Actual-out*(*targetnode a*,*i*) ∈ *HRB-slice S*
proof(*induct CFG-node* (*targetnode a*) *rule*:*HRB-slice-cases*)
case (*phase1 nx'*)
with *ddep* **show** *?case*
by(*fastforce intro*: *ddep-slice1 combine-SDG-slices.combSlice-refl*
simp:*HRB-slice-def*)
next
case (*phase2 nx' n' n'' p*)
from ‹*CFG-node* (*targetnode a*) ∈ *sum-SDG-slice2 n'*› *ddep*
have *Actual-out*(*targetnode a*,*i*) ∈ *sum-SDG-slice2 n'*
by(*fastforce intro*:*ddep-slice2*)
with ‹*n''* $s-p\rightarrow_{ret}$ *CFG-node* (*parent-node n'*)› ‹*n'* ∈ *sum-SDG-slice1 nx'*›
‹*nx'* ∈ *S*›
show *?case* **by**(*fastforce intro*:*combine-SDG-slices.combSlice-Return-parent-node*
simp:*HRB-slice-def*)
qed
from ‹*valid-edge a*› ‹*kind a* = *Q*$\hookleftarrow_p$*f'*› ‹*valid-edge a''*›
‹*kind a''* = *Q'*:*r'*$\hookrightarrow_p$*fs'*› ‹*a* ∈ *get-return-edges a''*›
‹*CFG-node* (*targetnode a*) ∈ *HRB-slice S*›
have *CFG-node* (*sourcenode a*) ∈ *HRB-slice S*
by(*rule call-return-nodes-in-slice*)
hence *sourcenode a* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$ **by**(*simp add*:*SDG-to-CFG-set-def*)
from ‹*sourcenode a* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*valid-edge a*› ‹*kind a* = *Q*$\hookleftarrow_p$*f'*›
‹(*p*,*ins*,*outs*) ∈ *set procs*›
have *slice-kind*:*slice-kind S a* =
Q$\hookleftarrow_p$(λ*cf cf'*. *rspp* (*targetnode a*) (*HRB-slice S*) *outs cf' cf*)
by(*rule slice-kind-Return-in-slice*)
from ‹*Actual-out*(*targetnode a*,*i*) ∈ *HRB-slice S*›
‹*i* < *length*(*ParamDefs* (*targetnode a*))› ‹*valid-edge a*›
‹*V* = (*ParamDefs* (*targetnode a*))!*i*› *length*
have *2*:*rspp* (*targetnode a*) (*HRB-slice S*) *outs* (*fst cfx'*) (*fst cf*$_2$) *V* =
(*fst cf*$_2$)(*outs*!*i*)
by(*fastforce intro*:*rspp-Actual-out-in-slice*)
from ‹*i* < *length*(*ParamDefs* (*targetnode a*))› *length* ‹*valid-edge a*›
have (*fst cfx*)(*ParamDefs* (*targetnode a*) [:=] *map* (*fst cf*$_1$) *outs*)
((*ParamDefs* (*targetnode a*))!*i*) = (*map* (*fst cf*$_1$) *outs*)!*i*
by(*fastforce intro*:*fun-upds-nth distinct-ParamDefs*)
with ‹*V* = (*ParamDefs* (*targetnode a*))!*i*›
‹*i* < *length*(*ParamDefs* (*targetnode a*))› *length*
have *1*:(*fst cfx*)(*ParamDefs* (*targetnode a*) [:=] *map* (*fst cf*$_1$) *outs*) *V* =
(*fst cf*$_1$)(*outs*!*i*)
by *simp*
from ‹*valid-edge a*› ‹*kind a* = *Q*$\hookleftarrow_p$*f'*› ‹(*p*,*ins*,*outs*) ∈ *set procs*›

$\langle i < length(ParamDefs\ (targetnode\ a))\rangle\ length$
have *po*:$Formal\text{-}out(sourcenode\ a,i)\ s{-}p{:}outs!i{\rightarrow}_{out}\ Actual\text{-}out(targetnode\ a,i)$
by(*fastforce intro*:*sum-SDG-param-out-edge*)
from $\langle valid\text{-}edge\ a\rangle\ \langle kind\ a = Q{\hookleftarrow}_{pf}{'}\rangle$
have $CFG\text{-}node\ (sourcenode\ a)\ s{-}p{\rightarrow}_{ret}\ CFG\text{-}node\ (targetnode\ a)$
by(*fastforce intro*:*sum-SDG-return-edge*)
from $\langle Actual\text{-}out(targetnode\ a,i) \in HRB\text{-}slice\ S\rangle$
have $Formal\text{-}out(sourcenode\ a,i) \in HRB\text{-}slice\ S$
proof(*induct* $Actual\text{-}out(targetnode\ a,i)$ *rule*:*HRB-slice-cases*)
case (*phase1 nx'*)
let *?AO* = $Actual\text{-}out(targetnode\ a,i)$
from $\langle valid\text{-}SDG\text{-}node\ ?AO\rangle$ **have** $?AO \in sum\text{-}SDG\text{-}slice2\ ?AO$
by(*rule refl-slice2*)
with *po* **have** $Formal\text{-}out(sourcenode\ a,i) \in sum\text{-}SDG\text{-}slice2\ ?AO$
by(*rule param-out-slice2*)
with $\langle CFG\text{-}node\ (sourcenode\ a)\ s{-}p{\rightarrow}_{ret}\ CFG\text{-}node\ (targetnode\ a)\rangle$
$\langle Actual\text{-}out\ (targetnode\ a,\ i) \in sum\text{-}SDG\text{-}slice1\ nx'\rangle\ \langle nx' \in S\rangle$
show *?case*
by(*fastforce intro*:*combSlice-Return-parent-node simp*:*HRB-slice-def*)
next
case (*phase2 nx' n' n'' p*)
from $\langle Actual\text{-}out\ (targetnode\ a,\ i) \in sum\text{-}SDG\text{-}slice2\ n'\rangle\ po$
have $Formal\text{-}out(sourcenode\ a,i) \in sum\text{-}SDG\text{-}slice2\ n'$
by(*fastforce intro*:*param-out-slice2*)
with $\langle n' \in sum\text{-}SDG\text{-}slice1\ nx'\rangle\ \langle n''\ s{-}p{\rightarrow}_{ret}\ CFG\text{-}node\ (parent\text{-}node\ n')\rangle$
$\langle nx' \in S\rangle$
show *?case* **by**(*fastforce intro*:*combine-SDG-slices.combSlice-Return-parent-node simp*:*HRB-slice-def*)
qed
with $\langle valid\text{-}edge\ a\rangle\ \langle kind\ a = Q{\hookleftarrow}_{pf}{'}\rangle\ \langle(p,ins,outs) \in set\ procs\rangle$
$\langle i < length(ParamDefs\ (targetnode\ a))\rangle\ length$
have $outs!i \in Use_{SDG}\ Formal\text{-}out(sourcenode\ a,i)$
by(*fastforce intro*!:*Formal-out-SDG-Use get-proc-return*)
with $\langle valid\text{-}edge\ a\rangle$ **have** $outs!i \in Use_{SDG}\ (CFG\text{-}node\ (sourcenode\ a))$
by(*auto intro*!:*CFG-Use-SDG-Use dest*:*SDG-Use-parent-Use*)
moreover
from $\langle valid\text{-}edge\ a\rangle$ **have** $parent\text{-}node\ (CFG\text{-}node\ (sourcenode\ a))\ {-}[]{\rightarrow}_{\iota}{*}$
$parent\text{-}node\ (CFG\text{-}node\ (sourcenode\ a))$
by(*fastforce intro*:*empty-path simp*:*intra-path-def*)
ultimately have $outs!i \in rv\ S\ (CFG\text{-}node\ (sourcenode\ a))$
using $\langle sourcenode\ a \in \lfloor HRB\text{-}slice\ S\rfloor_{CFG}\rangle\ \langle valid\text{-}edge\ a\rangle$
by(*fastforce intro*:*rvI simp*:*SDG-to-CFG-set-def sourcenodes-def*)
with $\langle\forall\, i < length\ ms_2.\ \forall\, V \in rv\ S\ (CFG\text{-}node\ ((mx\#tl\ ms_2)!i)).$
$(fst\ (s_1!(length\ msx + i)))\ V = (fst\ (s_2!i))\ V\rangle$
have $(fst\ cf_1)(outs!i) = (fst\ cf_2)(outs!i)$
by $auto(erule\text{-}tac\ x{=}0$ **in** $allE,auto)$
with *1 2 slice-kind* **show** *?thesis* **by** *simp*

next
case *False*
with ‹*transfer* (*kind a*) $s_1 = s_1'$›
have (*fst cfx*)(*ParamDefs* (*targetnode a*) [:=] *map* (*fst* cf_1) *outs*) =
(*fst* (*hd* cfs_1))(*ParamDefs* (*targetnode a*) [:=] *map* (*fst* cf_1) *outs*)
by(*cases* cfs_1,*auto intro*:*CFG-return-edge-fun*)
show *?thesis*
proof(*cases sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$)
case *True*
from ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*valid-edge a*› ‹*kind a* = $Q\hookleftarrow_p pf'$›
‹(*p*,*ins*,*outs*) ∈ *set procs*›
have *slice-kind S a* =
$Q\hookleftarrow_p$(λ*cf cf'*. *rspp* (*targetnode a*) (*HRB-slice S*) *outs cf' cf*)
by(*rule slice-kind-Return-in-slice*)
with ‹*length* s_1' = *length* (*transfer* (*slice-kind S a*) s_2)›
‹*length* s_1 = *length* s_2›
have *state-val* (*transfer* (*slice-kind S a*) s_2) *V* =
rspp (*targetnode a*) (*HRB-slice S*) *outs* (*fst cfx'*) (*fst* cf_2) *V*
by *simp*
with ‹*V* ∉ *set* (*ParamDefs* (*targetnode a*))›
have *state-val* (*transfer* (*slice-kind S a*) s_2) *V* = *state-val* cfs_2 *V*
by(*fastforce simp*:*rspp-def map-merge-def*)
with ‹∀ *i* < *length* ms_2. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl* ms_2)!*i*)).
(*fst* (s_1!(*length msx* + *i*))) *V* = (*fst* (s_2!*i*)) *V*›
‹*hd*(*tl* ms_1) = *targetnode a*›
‹*length* ms_1 = *length* s_1› ‹*length* s_1 = *length* s_2›[*THEN sym*] *False*
‹*tl* ms_2 = *tl* ms_1› ‹*length* ms_2 = *length* s_2›
‹*V*∈*rv S* (*CFG-node* (*targetnode a*))›
show *?thesis* **by**(*fastforce simp*:*length-Suc-conv fun-upds-notin*)
next
case *False*
from ‹*sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*kind a* = $Q\hookleftarrow_p pf'$›
have *slice-kind S a* = (λ*cf*. *True*)$\hookleftarrow_p$(λ*cf cf'*. *cf'*)
by(*rule slice-kind-Return*)
from ‹*length* ms_2 = *length* s_2› **have** *1* < *length* ms_2 **by** *simp*
with ‹∀ *i* < *length* ms_2. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl* ms_2)!*i*)).
(*fst* (s_1!(*length msx* + *i*))) *V* = (*fst* (s_2!*i*)) *V*›
‹*V*∈*rv S* (*CFG-node* (*hd* (*tl* ms_1)))›
‹ms_1' = *tl* ms_1› ‹ms_1' = []@*hd* ms_1'#*tl* ms_1'›
have *fst cfx V* = *fst cfx' V* **apply** *auto*
apply(*erule-tac x*=*1* **in** *allE*)
by(*cases tl* ms_1) *auto*
with ‹∀ *i* < *length* ms_2. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl* ms_2)!*i*)).
(*fst* (s_1!(*length msx* + *i*))) *V* = (*fst* (s_2!*i*)) *V*›
‹*hd*(*tl* ms_1) = *targetnode a*›
‹*length* ms_1 = *length* s_1› ‹*length* s_1 = *length* s_2›[*THEN sym*] *False*
‹*tl* ms_2 = *tl* ms_1› ‹*length* ms_2 = *length* s_2›
‹*V*∈*rv S* (*CFG-node* (*targetnode a*))›
‹*V* ∉ *set* (*ParamDefs* (*targetnode a*))›

‹slice-kind S a = (λcf. True)↩p(λcf cf'. cf')›
show *?thesis* **by**(*auto simp*:*fun-upds-notin*)
qed
qed
qed
with ‹*hd*(*tl ms*$_1$) = *targetnode a*› ‹*tl ms*$_2$ = *tl ms*$_1$› ‹*ms*$_1$' = *tl ms*$_1$›
‹∀ *i* < *length ms*$_2$. ∀ *V* ∈ *rv S* (*CFG-node* ((*mx*#*tl ms*$_2$)!*i*)).
(*fst* (*s*$_1$!(*length msx* + *i*))) *V* = (*fst* (*s*$_2$!*i*)) *V*› ‹*length ms*$_1$' = *length s*$_1$'›
‹*length ms*$_1$ = *length s*$_1$› ‹*length ms*$_2$ = *length s*$_2$› ‹*length s*$_1$ = *length s*$_2$› *cf2'*
have ∀ *i*<*length ms*$_1$'. ∀ *V*∈*rv S* (*CFG-node* ((*hd* (*tl ms*$_1$) # *tl ms*$_1$')!*i*)).
(*fst* (*s*$_1$'!(*length* [] + *i*))) *V* = (*fst* (*transfer* (*slice-kind S a*) *s*$_2$!*i*)) *V*
apply(*case-tac tl ms*$_1$) **apply** *auto*
apply(*cases ms*$_2$) **apply** *auto*
apply(*case-tac i*) **apply** *auto*
by(*erule-tac x*=*Suc*(*Suc nat*) **in** *allE*,*auto*)
with ‹∀ *m* ∈ *set ms*$_2$. *valid-node m*› ‹∀ *m* ∈ *set ms*$_1$'. *valid-node m*›
‹*length ms*$_2$ = *length s*$_2$› ‹*length s*$_1$' = *length* (*transfer* (*slice-kind S a*) *s*$_2$)›
‹*length ms*$_1$' = *length s*$_1$'› ‹*ms*$_1$' = *tl ms*$_1$› ‹*ms*$_1$' = []@*hd ms*$_1$'#*tl ms*$_1$'›
‹*tl ms*$_1$ = []@*hd*(*tl ms*$_1$)#*tl*(*tl ms*$_1$)›
‹*get-proc mx* = *get-proc* (*hd ms*$_2$)›
‹∀ *m* ∈ *set* (*tl* (*tl ms*$_1$)). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹∀ *m* ∈ *set* (*tl* (*tl ms*$_1$)). *return-node m*›
‹∀ *i*<*length ms*$_1$'. *snd* (*s*$_1$' ! *i*) = *snd* (*transfer* (*slice-kind S a*) *s*$_2$! *i*)›
have ((*ms*$_1$',*s*$_1$'),(*ms*$_1$',*transfer* (*slice-kind S a*) *s*$_2$)) ∈ *WS S*
by(*auto intro*!:*WSI*)
with ‹*S*,*slice-kind S* ⊢ (*ms*$_2$,*s*$_2$) =*as*@[*a*]⇒ (*ms*$_1$',*transfer* (*slice-kind S a*) *s*$_2$)›
show *?case* **by** *blast*
qed
qed

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*]⇒

$(ms_1{}',transfer\ (slice\text{-}kind\ S\ (last\ as))\ s_2)$
proof(*atomize-elim*)
from ‹S,$kind \vdash (ms_1,s_1) =as\Rightarrow (ms_1{}',s_1{}')$› **obtain** $ms'\ s'\ as'\ a'$
where S,$kind \vdash (ms_1,s_1) =as'\Rightarrow_\tau (ms',s')$
and S,$kind \vdash (ms',s') -a'\rightarrow (ms_1{}',s_1{}')$ **and** $as = as'@[a']$
by(*fastforce elim*:*observable-moves.cases*)
from ‹S,$kind \vdash (ms',s') -a'\rightarrow (ms_1{}',s_1{}')$›
have $\forall m \in set\ (tl\ ms').\ \exists m'.\ call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge m' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
and $\forall n \in set\ (tl\ ms').\ return\text{-}node\ n$ **and** $ms' \neq []$
by(*auto elim*:*observable-move.cases simp*:*call-of-return-node-def*)
from ‹S,$kind \vdash (ms_1,s_1) =as'\Rightarrow_\tau (ms',s')$› ‹$((ms_1,s_1),(ms_2,s_2)) \in WS\ S$›
have $((ms',s'),(ms_2,s_2)) \in WS\ S$ **by**(*rule WS-silent-moves*)
with ‹S,$kind \vdash (ms',s') -a'\rightarrow (ms_1{}',s_1{}')$› ‹$s_1{}' \neq []$›
obtain as'' **where** $((ms_1{}',s_1{}'),(ms_1{}',transfer\ (slice\text{-}kind\ S\ a')\ s_2)) \in WS\ S$
and S,$slice\text{-}kind\ S \vdash (ms_2,s_2) =as''@[a']\Rightarrow$
$(ms_1{}',transfer\ (slice\text{-}kind\ S\ a')\ s_2)$
by(*fastforce elim*:*WS-observable-move*)
with ‹$((ms_1{}',s_1{}'),(ms_1{}',transfer\ (slice\text{-}kind\ S\ a')\ s_2)) \in WS\ S$› ‹$as = as'@[a']$›
show $\exists as'.\ ((ms_1{}',s_1{}'),(ms_1{}',transfer\ (slice\text{-}kind\ S\ (last\ as))\ s_2)) \in WS\ S \wedge$
S,$slice\text{-}kind\ S \vdash (ms_2,s_2) =as'@[last\ as]\Rightarrow$
$(ms_1{}',transfer\ (slice\text{-}kind\ S\ (last\ as))\ s_2)$
by *fastforce*
qed

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* $(WS\ S)\ S$
by(*fastforce elim*:*WS-weak-sim simp*:*is-weak-sim-def*)

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(*induct rule*:*observable-move.induct*)

case (*observable-move-intra f a s s' ms S ms'*) **thus** *?case* **by** *simp*
next
case (*observable-move-call f a s s' Q r p fs a' ms S ms'*)
from ‹*valid-edge a*› ‹*valid-edge a'*› ‹*a' ∈ get-return-edges a*›
have *call-of-return-node* (*targetnode a'*) (*sourcenode a*)
by(*fastforce simp*:*return-node-def call-of-return-node-def*)
with ‹*hd ms = sourcenode a*› ‹*hd ms ∈ ⌊HRB-slice S⌋$_{CFG}$*›
‹*ms' = targetnode a # targetnode a' # tl ms*›
‹∀ *mx*∈*set* (*tl ms*). ∃ *mx'*. *call-of-return-node mx mx'* ∧ *mx'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
show *?case* **by** *fastforce*
next
case (*observable-move-return f a s s' Q p f' ms S ms'*)
thus *?case* **by**(*cases tl ms*) *auto*
qed

lemma *silent-move-preserves-stacks*:
assumes *S*,*f* ⊢ (*m*#*ms*,*s*) −*a*→$_\tau$ (*m'*#*ms'*,*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** *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'* **and** *upd-cs cs* [*a*] = *cs'*
proof(*atomize-elim*)
from *assms* **show** ∃ *cs' rs'*. *valid-node m'* ∧ *valid-call-list cs' m'* ∧
(∀ *i*<*length rs'*. *rs'* ! *i* ∈ *get-return-edges* (*cs'* ! *i*)) ∧
valid-return-list rs' m' ∧ *length rs'* = *length cs'* ∧ *ms'* = *targetnodes rs'* ∧
upd-cs cs [*a*] = *cs'*
proof(*induct S f m*#*ms s a m'*#*ms' s' rule*:*silent-move.induct*)
case (*silent-move-intra f a s s' n$_c$*)
from ‹*hd* (*m* # *ms*) = *sourcenode a*› **have** *m* = *sourcenode a* **by** *simp*
from ‹*m'* # *ms'* = *targetnode a* # *tl* (*m* # *ms*)›
have [*simp*]:*m'* = *targetnode a ms'* = *ms* **by** *simp-all*
from ‹*valid-edge a*› **have** *valid-node m'* **by** *simp*
moreover
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
from ‹*valid-call-list cs m*› ‹*m* = *sourcenode a*›
‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›
have *valid-call-list cs m'*
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*)
apply(*erule-tac x*=*c* **in** *allE*)
by(*auto split*:*list.split*)
moreover
from ‹*valid-return-list rs m*› ‹*m* = *sourcenode a*›
‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›

have *valid-return-list rs m'*
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*) *auto*
moreover
from ‹*intra-kind* (*kind a*)› **have** *upd-cs cs* [*a*] = *cs*
by(*fastforce simp*:*intra-kind-def*)
ultimately show *?case* **using** ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*length rs* = *length cs*› ‹*ms* = *targetnodes rs*›
apply(*rule-tac x*=*cs* **in** *exI*)
apply(*rule-tac x*=*rs* **in** *exI*)
by *clarsimp*
next
case (*silent-move-call f a s s' Q r p fs a' S*)
from ‹*hd* (*m* # *ms*) = *sourcenode a*›
‹*m'* # *ms'* = *targetnode a* # *targetnode a'* # *tl* (*m* # *ms*)›
have [*simp*]:*m* = *sourcenode a m'* = *targetnode a*
ms' = *targetnode a'* # *tl* (*m* # *ms*)
by *simp-all*
from ‹*valid-edge a*› **have** *valid-node m'* **by** *simp*
moreover
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*pfs*› **have** *get-proc* (*targetnode a*) = *p*
by(*rule get-proc-call*)
with ‹*valid-call-list cs m*› ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*pfs*› ‹*m* = *sourcenode a*›
have *valid-call-list* (*a* # *cs*) (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x*=*list* **in** *allE*)
by(*case-tac list*)(*auto simp*:*sourcenodes-def*)
moreover
from ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)› ‹*a'* ∈ *get-return-edges a*›
have ∀ *i*<*length* (*a'*#*rs*). (*a'*#*rs*) ! *i* ∈ *get-return-edges* ((*a*#*cs*) ! *i*)
by *auto*(*case-tac i*,*auto*)
moreover
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*pfs*› ‹*a'* ∈ *get-return-edges a*›
obtain *Q' f'* **where** *kind a'* = *Q'*↩*pf'* **by**(*fastforce dest*!:*call-return-edges*)
from ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩*pf'*› **have** *get-proc* (*sourcenode a'*) = *p*
by(*rule get-proc-return*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a'*)
by(*rule get-proc-get-return-edge*)
with ‹*valid-return-list rs m*› ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩*pf'*›
‹*get-proc* (*sourcenode a'*) = *p*› ‹*get-proc* (*targetnode a*) = *p*› ‹*m* = *sourcenode a*›
have *valid-return-list* (*a'*#*rs*) (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)

apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x=list* **in** *allE*)
by(*case-tac list*)(*auto simp*:*targetnodes-def*)
moreover
from ‹*length rs = length cs*› **have** *length* (*a'#rs*) = *length* (*a#cs*) **by** *simp*
moreover
from ‹*ms = targetnodes rs*› **have** *targetnode a' # ms = targetnodes* (*a' # rs*)
by(*simp add*:*targetnodes-def*)
moreover
from ‹*kind a = Q*:*r*↪$_p$*fs*› **have** *upd-cs cs* [*a*] = *a#cs* **by** *simp*
ultimately show *?case*
apply(*rule-tac x=a#cs* **in** *exI*)
apply(*rule-tac x=a'#rs* **in** *exI*)
by *clarsimp*
next
case (*silent-move-return f a s s' Q p f' S*)
from ‹*hd* (*m # ms*) = *sourcenode a*›
‹*hd* (*tl* (*m # ms*)) = *targetnode a*› ‹*m' # ms' = tl* (*m # ms*)› [*symmetric*]
have [*simp*]:*m = sourcenode a m' = targetnode a* **by** *simp-all*
from ‹*length* (*m # ms*) = *length s*› ‹*length s = Suc* (*length s'*)› ‹*s' ≠* []›
‹*hd* (*tl* (*m # ms*)) = *targetnode a*› ‹*m' # ms' = tl* (*m # ms*)›
have *ms = targetnode a # ms'*
by(*cases ms*) *auto*
with ‹*ms = targetnodes rs*›
obtain *r' rs'* **where** *rs = r' # rs'*
and *targetnode a = targetnode r'* **and** *ms' = targetnodes rs'*
by(*cases rs*)(*auto simp*:*targetnodes-def*)
moreover
from ‹*rs = r' # rs'*› ‹*length rs = length cs*› **obtain** *c' cs'* **where** *cs = c' # cs'*
and *length rs' = length cs'* **by**(*cases cs*) *auto*
moreover
from ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*rs = r' # rs'*› ‹*cs = c' # cs'*›
have ∀ *i*<*length rs'*. *rs'* ! *i* ∈ *get-return-edges* (*cs'* ! *i*)
and *r'* ∈ *get-return-edges c'* **by** *auto*
moreover
from ‹*valid-edge a*› **have** *valid-node* (*targetnode a*) **by** *simp*
moreover
from ‹*valid-call-list cs m*› ‹*cs = c' # cs'*›
obtain *p' Q' r fs'* **where** *valid-edge c'* **and** *kind c' = Q'*:*r*↪$_{p'}$*fs'*
and *p' = get-proc m*
apply(*auto simp*:*valid-call-list-def*)
by(*erule-tac x*=[] **in** *allE*) *auto*
from ‹*valid-edge a*› ‹*kind a = Q*↩$_p$*f'*›
have *get-proc* (*sourcenode a*) = *p* **by**(*rule get-proc-return*)
with ‹*p' = get-proc m*› **have** [*simp*]:*p' = p* **by** *simp*
from ‹*valid-edge c'*› ‹*kind c' = Q'*:*r*↪$_{p'}$*fs'*›
have *get-proc* (*targetnode c'*) = *p* **by**(*fastforce intro*:*get-proc-call*)

from ‹*valid-edge c'*› ‹*r' ∈ get-return-edges c'*› **have** *valid-edge r'*
by(*rule get-return-edges-valid*)
from ‹*valid-edge c'*› ‹*kind c' = Q':r↪$_{p'}$fs'*› ‹*r' ∈ get-return-edges c'*›
obtain *Q'' f''* **where** *kind r' = Q''↩$_p$f''* **by**(*fastforce dest*!:*call-return-edges*)
with ‹*valid-edge r'*› **have** *get-proc* (*sourcenode r'*) = *p* **by**(*rule get-proc-return*)
from ‹*valid-edge r'*› ‹*kind r' = Q''↩$_p$f''*› **have** *method-exit* (*sourcenode r'*)
by(*fastforce simp*:*method-exit-def*)
from ‹*valid-edge a*› ‹*kind a = Q↩$_p$f'*› **have** *method-exit* (*sourcenode a*)
by(*fastforce simp*:*method-exit-def*)
with ‹*method-exit* (*sourcenode r'*)› ‹*get-proc* (*sourcenode r'*) = *p*›
‹*get-proc* (*sourcenode a*) = *p*›
have *sourcenode a = sourcenode r'* **by**(*fastforce intro*:*method-exit-unique*)
with ‹*valid-edge a*› ‹*valid-edge r'*› ‹*targetnode a = targetnode r'*›
have *a = r'* **by**(*fastforce intro*:*edge-det*)
from ‹*valid-edge c'*› ‹*r' ∈ get-return-edges c'*› ‹*targetnode a = targetnode r'*›
have *get-proc* (*sourcenode c'*) = *get-proc* (*targetnode a*)
by(*fastforce intro*:*get-proc-get-return-edge*)
from ‹*valid-call-list cs m*› ‹*cs = c' # cs'*›
‹*get-proc* (*sourcenode c'*) = *get-proc* (*targetnode a*)›
have *valid-call-list cs'* (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*c' # cs'* **in** *allE*)
by(*case-tac cs'*)(*auto simp*:*sourcenodes-def*)
moreover
from ‹*valid-return-list rs m*› ‹*rs = r' # rs'*› ‹*targetnode a = targetnode r'*›
have *valid-return-list rs'* (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*r' # cs'* **in** *allE*)
by(*case-tac cs'*)(*auto simp*:*targetnodes-def*)
moreover
from ‹*kind a = Q↩$_p$f'*› ‹*cs = c' # cs'*› **have** *upd-cs cs* [*a*] = *cs'* **by** *simp*
ultimately show *?case*
apply(*rule-tac x*=*cs'* **in** *exI*)
apply(*rule-tac x*=*rs'* **in** *exI*)
by *clarsimp*
qed
qed

lemma *silent-moves-preserves-stacks*:
assumes *S,f* ⊢ (*m#ms,s*) =*as*⇒$_\tau$ (*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'* **and** *upd-cs cs as* = *cs'*
proof(*atomize-elim*)

```
  from assms show ∃ cs' rs'. valid-node m' ∧ valid-call-list cs' m' ∧
    (∀ i<length rs'. rs' ! i ∈ get-return-edges (cs' ! i)) ∧
    valid-return-list rs' m' ∧ length rs' = length cs' ∧ ms' = targetnodes rs' ∧
    upd-cs cs as = cs'
  proof(induct S f m#ms s as m'#ms' s'
      arbitrary:m ms cs rs rule:silent-moves.induct)
    case (silent-moves-Nil s n_c f)
    thus ?case
      apply(rule-tac x=cs in exI)
      apply(rule-tac x=rs in exI)
      by clarsimp
  next
    case (silent-moves-Cons S f s a msx'' s'' as sx')
    note IH = ‹⋀m ms cs rs. ⟦msx'' = m # ms; valid-node m; valid-call-list cs m;
      ∀ i<length rs. rs ! i ∈ get-return-edges (cs ! i);
      valid-return-list rs m; length rs = length cs; ms = targetnodes rs⟧
      ⟹ ∃ cs' rs'. valid-node m' ∧ valid-call-list cs' m' ∧
      (∀ i<length rs'. rs' ! i ∈ get-return-edges (cs' ! i)) ∧
      valid-return-list rs' m' ∧ length rs' = length cs' ∧ ms' = targetnodes rs' ∧
      upd-cs cs as = cs'›
    from ‹S,f ⊢ (m # ms,s) −a→τ (msx'',s'')›
    obtain m'' ms'' where msx'' = m''#ms''
      by(cases msx'')(auto elim:silent-move.cases)
    with ‹S,f ⊢ (m # ms,s) −a→τ (msx'',s'')› ‹valid-call-list cs m›
      ‹∀ i<length rs. rs ! i ∈ get-return-edges (cs ! i)› ‹valid-return-list rs m›
      ‹length rs = length cs› ‹ms = targetnodes rs›
    obtain cs'' rs'' where hyps:valid-node m'' valid-call-list cs'' m''
      ∀ i < length rs''. rs''!i ∈ get-return-edges (cs''!i)
      valid-return-list rs'' m'' length rs'' = length cs''
      ms'' = targetnodes rs'' and upd-cs cs [a] = cs''
      by(auto elim!:silent-move-preserves-stacks)
    from IH[OF - hyps] ‹msx'' = m'' # ms''›
    obtain cs' rs' where results:valid-node m' valid-call-list cs' m'
      ∀ i<length rs'. rs' ! i ∈ get-return-edges (cs' ! i)
      valid-return-list rs' m' length rs' = length cs' ms' = targetnodes rs'
      and upd-cs cs'' as = cs' by blast
    from ‹upd-cs cs [a] = cs''› ‹upd-cs cs'' as = cs'›
    have upd-cs cs ([a] @ as) = cs' by(rule upd-cs-Append)
    with results show ?case
      apply(rule-tac x=cs' in exI)
      apply(rule-tac x=rs' in exI)
      by clarsimp
  qed
qed


lemma observable-move-preserves-stacks:
  assumes S,f ⊢ (m#ms,s) −a→ (m'#ms',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** *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'* **and** *upd-cs cs* [*a*] = *cs'*
proof(*atomize-elim*)
from *assms* **show** ∃ *cs' rs'*. *valid-node m'* ∧ *valid-call-list cs' m'* ∧
(∀ *i*<*length rs'*. *rs'* ! *i* ∈ *get-return-edges* (*cs'* ! *i*)) ∧
valid-return-list rs' m' ∧ *length rs'* = *length cs'* ∧ *ms'* = *targetnodes rs'* ∧
upd-cs cs [*a*] = *cs'*
proof(*induct S f m*#*ms s a m'*#*ms' s' rule*:*observable-move.induct*)
case (*observable-move-intra f a s s'* n_c)
from ‹*hd* (*m* # *ms*) = *sourcenode a*› **have** *m* = *sourcenode a* **by** *simp*
from ‹*m'* # *ms'* = *targetnode a* # *tl* (*m* # *ms*)›
have [*simp*]:*m'* = *targetnode a ms'* = *ms* **by** *simp-all*
from ‹*valid-edge a*› **have** *valid-node m'* **by** *simp*
moreover
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
from ‹*valid-call-list cs m*› ‹*m* = *sourcenode a*›
‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›
have *valid-call-list cs m'*
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*)
apply(*erule-tac x*=*c* **in** *allE*)
by(*auto split*:*list.split*)
moreover
from ‹*valid-return-list rs m*› ‹*m* = *sourcenode a*›
‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›
have *valid-return-list rs m'*
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*) *auto*
moreover
from ‹*intra-kind* (*kind a*)› **have** *upd-cs cs* [*a*] = *cs*
by(*fastforce simp*:*intra-kind-def*)
ultimately show *?case* **using** ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*length rs* = *length cs*› ‹*ms* = *targetnodes rs*›
apply(*rule-tac x*=*cs* **in** *exI*)
apply(*rule-tac x*=*rs* **in** *exI*)
by *clarsimp*
next
case (*observable-move-call f a s s' Q r p fs a' S*)
from ‹*hd* (*m* # *ms*) = *sourcenode a*›
‹*m'* # *ms'* = *targetnode a* # *targetnode a'* # *tl* (*m* # *ms*)›
have [*simp*]:*m* = *sourcenode a m'* = *targetnode a*
ms' = *targetnode a'* # *tl* (*m* # *ms*)
by *simp-all*
from ‹*valid-edge a*› **have** *valid-node m'* **by** *simp*

moreover
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*pfs*› **have** *get-proc* (*targetnode a*) = *p*
by(*rule get-proc-call*)
with ‹*valid-call-list cs m*› ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*pfs*› ‹*m* = *sourcenode a*›
have *valid-call-list* (*a* # *cs*) (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x*=*list* **in** *allE*)
by(*case-tac list*)(*auto simp*:*sourcenodes-def*)
moreover
from ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)› ‹*a'* ∈ *get-return-edges a*›
have ∀ *i*<*length* (*a'*#*rs*). (*a'*#*rs*) ! *i* ∈ *get-return-edges* ((*a*#*cs*) ! *i*)
by *auto*(*case-tac i*,*auto*)
moreover
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*pfs*› ‹*a'* ∈ *get-return-edges a*›
obtain *Q' f'* **where** *kind a'* = *Q'*↩*pf'* **by**(*fastforce dest*!:*call-return-edges*)
from ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩*pf'*› **have** *get-proc* (*sourcenode a'*) = *p*
by(*rule get-proc-return*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a'*)
by(*rule get-proc-get-return-edge*)
with ‹*valid-return-list rs m*› ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩*pf'*›
‹*get-proc* (*sourcenode a'*) = *p*› ‹*get-proc* (*targetnode a*) = *p*› ‹*m* = *sourcenode a*›
have *valid-return-list* (*a'*#*rs*) (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x*=*list* **in** *allE*)
by(*case-tac list*)(*auto simp*:*targetnodes-def*)
moreover
from ‹*length rs* = *length cs*› **have** *length* (*a'*#*rs*) = *length* (*a*#*cs*) **by** *simp*
moreover
from ‹*ms* = *targetnodes rs*› **have** *targetnode a'* # *ms* = *targetnodes* (*a'* # *rs*)
by(*simp add*:*targetnodes-def*)
moreover
from ‹*kind a* = *Q*:*r*↪*pfs*› **have** *upd-cs cs* [*a*] = *a*#*cs* **by** *simp*
ultimately show *?case*
apply(*rule-tac x*=*a*#*cs* **in** *exI*)
apply(*rule-tac x*=*a'*#*rs* **in** *exI*)
by *clarsimp*
next
case (*observable-move-return f a s s' Q p f' S*)
from ‹*hd* (*m* # *ms*) = *sourcenode a*›
‹*hd* (*tl* (*m* # *ms*)) = *targetnode a*› ‹*m'* # *ms'* = *tl* (*m* # *ms*)› [*symmetric*]
have [*simp*]:*m* = *sourcenode a m'* = *targetnode a* **by** *simp-all*
from ‹*length* (*m* # *ms*) = *length s*› ‹*length s* = *Suc* (*length s'*)› ‹*s'* ≠ []›

```
    ‹hd (tl (m # ms)) = targetnode a› ‹m' # ms' = tl (m # ms)›
  have ms = targetnode a # ms'
    by(cases ms) auto
  with ‹ms = targetnodes rs›
  obtain r' rs' where rs = r' # rs'
    and targetnode a = targetnode r' and ms' = targetnodes rs'
    by(cases rs)(auto simp:targetnodes-def)
  moreover
  from ‹rs = r' # rs'› ‹length rs = length cs› obtain c' cs' where cs = c' #
cs'
    and length rs' = length cs' by(cases cs) auto
  moreover
  from ‹∀ i<length rs. rs ! i ∈ get-return-edges (cs ! i)›
    ‹rs = r' # rs'› ‹cs = c' # cs'›
  have ∀ i<length rs'. rs' ! i ∈ get-return-edges (cs' ! i)
    and r' ∈ get-return-edges c' by auto
  moreover
  from ‹valid-edge a› have valid-node (targetnode a) by simp
  moreover
  from ‹valid-call-list cs m› ‹cs = c' # cs'›
  obtain p' Q' r fs' where valid-edge c' and kind c' = Q':r↪p'fs'
    and p' = get-proc m
    apply(auto simp:valid-call-list-def)
    by(erule-tac x=[] in allE) auto
  from ‹valid-edge a› ‹kind a = Q↩pf'›
  have get-proc (sourcenode a) = p by(rule get-proc-return)
  with ‹p' = get-proc m› have [simp]:p' = p by simp
  from ‹valid-edge c'› ‹kind c' = Q':r↪p'fs'›
  have get-proc (targetnode c') = p by(fastforce intro:get-proc-call)
  from ‹valid-edge c'› ‹r' ∈ get-return-edges c'› have valid-edge r'
    by(rule get-return-edges-valid)
  from ‹valid-edge c'› ‹kind c' = Q':r↪p'fs'› ‹r' ∈ get-return-edges c'›
  obtain Q'' f'' where kind r' = Q''↩pf'' by(fastforce dest!:call-return-edges)
  with ‹valid-edge r'› have get-proc (sourcenode r') = p by(rule get-proc-return)
  from ‹valid-edge r'› ‹kind r' = Q''↩pf''› have method-exit (sourcenode r')
    by(fastforce simp:method-exit-def)
  from ‹valid-edge a› ‹kind a = Q↩pf'› have method-exit (sourcenode a)
    by(fastforce simp:method-exit-def)
  with ‹method-exit (sourcenode r')› ‹get-proc (sourcenode r') = p›
    ‹get-proc (sourcenode a) = p›
  have sourcenode a = sourcenode r' by(fastforce intro:method-exit-unique)
  with ‹valid-edge a› ‹valid-edge r'› ‹targetnode a = targetnode r'›
  have a = r' by(fastforce intro:edge-det)
  from ‹valid-edge c'› ‹r' ∈ get-return-edges c'› ‹targetnode a = targetnode r'›
  have get-proc (sourcenode c') = get-proc (targetnode a)
    by(fastforce intro:get-proc-get-return-edge)
  from ‹valid-call-list cs m› ‹cs = c' # cs'›
    ‹get-proc (sourcenode c') = get-proc (targetnode a)›
  have valid-call-list cs' (targetnode a)
```

apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*c′ # cs′* **in** *allE*)
by(*case-tac cs′*)(*auto simp*:*sourcenodes-def*)
moreover
from ‹*valid-return-list rs m*› ‹*rs = r′ # rs′*› ‹*targetnode a = targetnode r′*›
have *valid-return-list rs′* (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*r′ # cs′* **in** *allE*)
by(*case-tac cs′*)(*auto simp*:*targetnodes-def*)
moreover
from ‹*kind a = Q↩pf′*› ‹*cs = c′ # cs′*› **have** *upd-cs cs* [*a*] = *cs′* **by** *simp*
ultimately show *?case*
apply(*rule-tac x*=*cs′* **in** *exI*)
apply(*rule-tac x*=*rs′* **in** *exI*)
by *clarsimp*
qed
qed

lemma *observable-moves-preserves-stack*:
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′* **and** *upd-cs cs as* = *cs′*
proof(*atomize-elim*)
from ‹*S*,*f* ⊢ (*m*#*ms*,*s*) =*as*⇒ (*m′*#*ms′*,*s′*)› **obtain** *msx s″ as′ a′*
where *as* = *as′*@[*a′*] **and** *S*,*f* ⊢ (*m*#*ms*,*s*) =*as′*⇒$_\tau$ (*msx*,*s″*)
and *S*,*f* ⊢ (*msx*,*s″*) −*a′*→ (*m′*#*ms′*,*s′*)
by(*fastforce elim*:*observable-moves.cases*)
from ‹*S*,*f* ⊢ (*msx*,*s″*) −*a′*→ (*m′*#*ms′*,*s′*)› **obtain** *m″ ms″*
where [*simp*]:*msx* = *m″*#*ms″* **by**(*cases msx*)(*auto elim*:*observable-move.cases*)
from ‹*S*,*f* ⊢ (*m*#*ms*,*s*) =*as′*⇒$_\tau$ (*msx*,*s″*)› ‹*valid-node m*› ‹*valid-call-list cs m*›
‹∀ *i* < *length rs*. *rs*!*i* ∈ *get-return-edges* (*cs*!*i*)› ‹*valid-return-list rs m*›
‹*length rs* = *length cs*› ‹*ms* = *targetnodes rs*›
obtain *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″* **and** *upd-cs cs as′* = *cs″*
by(*auto elim*!:*silent-moves-preserves-stacks*)
with ‹*S*,*f* ⊢ (*msx*,*s″*) −*a′*→ (*m′*#*ms′*,*s′*)›
obtain *cs′ rs′* **where** *results*:*valid-node m′ valid-call-list cs′ m′*
∀ *i*<*length rs′*. *rs′* ! *i* ∈ *get-return-edges* (*cs′* ! *i*)
valid-return-list rs′ m′ length rs′ = *length cs′ ms′* = *targetnodes rs′*
and *upd-cs cs″* [*a′*] = *cs′*
by(*auto elim*!:*observable-move-preserves-stacks*)

from ‹*upd-cs cs as′ = cs″*› ‹*upd-cs cs″ [a′] = cs′*›
have *upd-cs cs (as′@[a′]) = cs′* **by**(*rule upd-cs-Append*)
with ‹*as = as′@[a′]*› *results*
show ∃ *cs′ rs′. valid-node m′ ∧ valid-call-list cs′ m′ ∧*
(∀ *i*<*length rs′. rs′ ! i ∈ get-return-edges (cs′ ! i)*) ∧
valid-return-list rs′ m′ ∧ length rs′ = length cs′ ∧ ms′ = targetnodes rs′ ∧
upd-cs cs as = cs′
apply(*rule-tac x*=*cs′* **in** *exI*)
apply(*rule-tac x*=*rs′* **in** *exI*)
by *clarsimp*
qed

lemma *silent-moves-slpa-path*:
⟦*S*,*f* ⊢ (*m*#*ms″*@*ms*,*s*) =*as*⇒$_\tau$ (*m′*#*ms′*,*s′*); *valid-node m*; *valid-call-list cs m*;
∀ *i* < *length rs*. *rs*!*i* ∈ *get-return-edges* (*cs*!*i*); *valid-return-list rs m*;
length rs = *length cs*; *ms″* = *targetnodes rs*;
∀ *mx* ∈ *set ms*. ∃ *mx′*. *call-of-return-node mx mx′* ∧ *mx′* ∈ ⌊*HRB-slice S*⌋$_{CFG}$;
ms″ ≠ [] ⟶ (∃ *mx′*. *call-of-return-node* (*last ms″*) *mx′* ∧ *mx′* ∉ ⌊*HRB-slice S*⌋$_{CFG}$);
∀ *mx* ∈ *set ms′*. ∃ *mx′*. *call-of-return-node mx mx′* ∧ *mx′* ∈ ⌊*HRB-slice S*⌋$_{CFG}$⟧
⟹ *same-level-path-aux cs as* ∧ *upd-cs cs as* = [] ∧ *m* −*as*→∗ *m′* ∧ *ms* = *ms′*
proof(*induct S f m*#*ms″*@*ms s as m′*#*ms′ s′ arbitrary*:*m ms″ ms cs rs*
rule:*silent-moves.induct*)
case (*silent-moves-Nil sx S f*) **thus** *?case*
apply(*cases ms″ rule*:*rev-cases*) **apply**(*auto intro*:*empty-path simp*:*targetnodes-def*)
by(*cases rs rule*:*rev-cases*,*auto*)+
next
case (*silent-moves-Cons S f sx a msx′ sx′ as sx″*)
thus *?case*
proof(*induct - - m*#*ms″*@*ms - - - - rule*:*silent-move.induct*)
case (*silent-move-intra f a s s′ S msx′*)
note *IH* = ‹⋀*m ms″ ms cs rs*. ⟦*msx′* = *m* # *ms″* @ *ms*; *valid-node m*;
valid-call-list cs m; ∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*);
valid-return-list rs m; *length rs* = *length cs*; *ms″* = *targetnodes rs*;
∀ *mx*∈*set ms*. ∃ *mx′*. *call-of-return-node mx mx′* ∧ *mx′* ∈ ⌊*HRB-slice S*⌋$_{CFG}$;
ms″ ≠ [] ⟶
(∃ *mx′*. *call-of-return-node* (*last ms″*) *mx′* ∧ *mx′* ∉ ⌊*HRB-slice S*⌋$_{CFG}$);
∀ *mx*∈*set ms′*. ∃ *mx′*. *call-of-return-node mx mx′* ∧ *mx′* ∈ ⌊*HRB-slice S*⌋$_{CFG}$⟧
⟹ *same-level-path-aux cs as* ∧ *upd-cs cs as* = [] ∧ *m* −*as*→∗ *m′* ∧ *ms* = *ms′*›
note *callstack* = ‹∀ *mx*∈*set ms*. ∃ *mx′*. *call-of-return-node mx mx′* ∧
mx′ ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
note *callstack″* = ‹*ms″* ≠ [] ⟶
(∃ *mx′*. *call-of-return-node* (*last ms″*) *mx′* ∧ *mx′* ∉ ⌊*HRB-slice S*⌋$_{CFG}$)›
note *callstack′* = ‹∀ *mx*∈*set ms′*. ∃ *mx′*. *call-of-return-node mx mx′* ∧
mx′ ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
from ‹*valid-edge a*› **have** *valid-node* (*targetnode a*) **by** *simp*
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›

have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
from ‹*hd* (*m* # *ms''* @ *ms*) = *sourcenode a*› **have** *m* = *sourcenode a*
by *simp*
from ‹*valid-call-list cs m*› ‹*m* = *sourcenode a*›
‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›
have *valid-call-list cs* (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*)
apply(*erule-tac x*=*c* **in** *allE*)
by(*auto split*:*list.split*)
from ‹*valid-return-list rs m*› ‹*m* = *sourcenode a*›
‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›
have *valid-return-list rs* (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*) *auto*
from ‹*msx'* = *targetnode a* # *tl* (*m* # *ms''* @ *ms*)›
have *msx'* = *targetnode a* # *ms''* @ *ms* **by** *simp*
from *IH*[*OF this* ‹*valid-node* (*targetnode a*)› ‹*valid-call-list cs* (*targetnode a*)›
‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*valid-return-list rs* (*targetnode a*)› ‹*length rs* = *length cs*›
‹*ms''* = *targetnodes rs*› *callstack callstack'' callstack'*]
have *same-level-path-aux cs as* **and** *upd-cs cs as* = []
and *targetnode a* −*as*→∗ *m'* **and** *ms* = *ms'* **by** *simp-all*
from ‹*intra-kind* (*kind a*)› ‹*same-level-path-aux cs as*›
have *same-level-path-aux cs* (*a* # *as*) **by**(*fastforce simp*:*intra-kind-def*)
moreover
from ‹*intra-kind* (*kind a*)› ‹*upd-cs cs as* = []›
have *upd-cs cs* (*a* # *as*) = [] **by**(*fastforce simp*:*intra-kind-def*)
moreover
from ‹*valid-edge a*› ‹*m* = *sourcenode a*› ‹*targetnode a* −*as*→∗ *m'*›
have *m* −*a* # *as*→∗ *m'* **by**(*fastforce intro*:*Cons-path*)
ultimately show *?case* **using** ‹*ms* = *ms'*› **by** *simp*
next
case (*silent-move-call f a s s' Q r p fs a' S msx'*)
note *IH* = ‹⋀*m ms'' ms cs rs*. ⟦*msx'* = *m* # *ms''* @ *ms*; *valid-node m*; *valid-call-list cs m*;
∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*); *valid-return-list rs m*;
length rs = *length cs*; *ms''* = *targetnodes rs*;
∀ *mx*∈*set ms*. ∃ *mx'*. *call-of-return-node mx mx'* ∧ *mx'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$;
ms'' ≠ [] ⟶
(∃ *mx'*. *call-of-return-node* (*last ms''*) *mx'* ∧ *mx'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$);
∀ *mx*∈*set ms'*. ∃ *mx'*. *call-of-return-node mx mx'* ∧ *mx'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$⟧
⟹ *same-level-path-aux cs as* ∧ *upd-cs cs as* = [] ∧ *m* −*as*→∗ *m'* ∧ *ms* = *ms'*›
note *callstack* = ‹∀ *mx*∈*set ms*. ∃ *mx'*. *call-of-return-node mx mx'* ∧
mx' ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
note *callstack''* = ‹*ms''* ≠ [] ⟶
(∃ *mx'*. *call-of-return-node* (*last ms''*) *mx'* ∧ *mx'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$)›

note *callstack'* = ‹∀ *mx*∈*set ms'*. ∃ *mx'*. *call-of-return-node mx mx'* ∧
mx' ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
from ‹*valid-edge a*› **have** *valid-node* (*targetnode a*) **by** *simp*
from ‹*hd* (*m* # *ms''* @ *ms*) = *sourcenode a*› **have** *m* = *sourcenode a*
by *simp*
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› **have** *get-proc* (*targetnode a*) = *p*
by(*rule get-proc-call*)
with ‹*valid-call-list cs m*› ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*m* = *sourcenode a*›
have *valid-call-list* (*a* # *cs*) (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x*=*list* **in** *allE*)
by(*case-tac list*)(*auto simp*:*sourcenodes-def*)
from ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)› ‹*a'* ∈ *get-return-edges a*›
have ∀ *i*<*length* (*a'*#*rs*). (*a'*#*rs*) ! *i* ∈ *get-return-edges* ((*a*#*cs*) ! *i*)
by *auto*(*case-tac i*,*auto*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*a'* ∈ *get-return-edges a*›
obtain *Q' f'* **where** *kind a'* = *Q'*↩$_p$*f'* **by**(*fastforce dest*!:*call-return-edges*)
from ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩$_p$*f'*› **have** *get-proc* (*sourcenode a'*) = *p*
by(*rule get-proc-return*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a'*)
by(*rule get-proc-get-return-edge*)
with ‹*valid-return-list rs m*› ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩$_p$*f'*›
‹*get-proc* (*sourcenode a'*) = *p*› ‹*get-proc* (*targetnode a*) = *p*› ‹*m* = *sourcenode a*›
have *valid-return-list* (*a'*#*rs*) (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x*=*list* **in** *allE*)
by(*case-tac list*)(*auto simp*:*targetnodes-def*)
from ‹*length rs* = *length cs*› **have** *length* (*a'*#*rs*) = *length* (*a* # *cs*) **by** *simp*
from ‹*ms''* = *targetnodes rs*›
have *targetnode a'* # *ms''* = *targetnodes* (*a'*#*rs*) **by**(*simp add*:*targetnodes-def*)
from ‹*msx'* = *targetnode a* # *targetnode a'* # *tl* (*m* # *ms''* @ *ms*)›
have *msx'* = *targetnode a* # *targetnode a'* # *ms''* @ *ms* **by** *simp*
have ∃ *mx'*. *call-of-return-node* (*last* (*targetnode a'* # *ms''*)) *mx'* ∧
mx' ∉ ⌊*HRB-slice S*⌋$_{CFG}$
proof(*cases ms''* = [])
case *True*
with ‹(∃ *m*∈*set* (*tl* (*m* # *ms''* @ *ms*)).
∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$) ∨
hd (*m* # *ms''* @ *ms*) ∉ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*m* = *sourcenode a*› *callstack*
have *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **by** *fastforce*
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)

with ‹*valid-edge a*› ‹$a' \in$ *get-return-edges a*›
have *call-of-return-node* (*targetnode a'*) (*sourcenode a*)
by(*fastforce simp*:*call-of-return-node-def return-node-def*)
with ‹*sourcenode a* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$› *True* **show** *?thesis* **by** *fastforce*
next
case *False*
with *callstack''* **show** *?thesis* **by** *fastforce*
qed
hence *targetnode a' # ms''* $\neq []$ $\longrightarrow$
($\exists$ *mx'*. *call-of-return-node* (*last* (*targetnode a' # ms''*)) *mx'* $\wedge$
mx' $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$) **by** *simp*
from *IH*[*OF* - ‹*valid-node* (*targetnode a*)› ‹*valid-call-list* (*a # cs*) (*targetnode a*)›
‹$\forall$ *i*<*length* (*a'*#*rs*). (*a'*#*rs*) ! *i* $\in$ *get-return-edges* ((*a*#*cs*) ! *i*)›
‹*valid-return-list* (*a'*#*rs*) (*targetnode a*)› ‹*length* (*a'*#*rs*) = *length* (*a # cs*)›
‹*targetnode a' # ms''* = *targetnodes* (*a'*#*rs*)› *callstack this callstack'*]
‹*msx'* = *targetnode a # targetnode a' # ms''* @ *ms*›
have *same-level-path-aux* (*a # cs*) *as* **and** *upd-cs* (*a # cs*) *as* = []
and *targetnode a* $-as\rightarrow*$ *m'* **and** *ms* = *ms'* **by** *simp-all*
from ‹*kind a* = *Q*:*r*$\hookrightarrow_p$*fs*› ‹*same-level-path-aux* (*a # cs*) *as*›
have *same-level-path-aux cs* (*a # as*) **by** *simp*
moreover
from ‹*kind a* = *Q*:*r*$\hookrightarrow_p$*fs*› ‹*upd-cs* (*a # cs*) *as* = []› **have** *upd-cs cs* (*a # as*) = []
by *simp*
moreover
from ‹*valid-edge a*› ‹*m* = *sourcenode a*› ‹*targetnode a* $-as\rightarrow*$ *m'*›
have *m* $-a$ *# as*$\rightarrow*$ *m'* **by**(*fastforce intro*:*Cons-path*)
ultimately show *?case* **using** ‹*ms* = *ms'*› **by** *simp*
next
case (*silent-move-return f a s s' Q p f' S msx'*)
note *IH* = ‹$\bigwedge$*m ms'' ms cs rs*. ⟦*msx'* = *m # ms''* @ *ms*; *valid-node m*;
valid-call-list cs m; $\forall$ *i*<*length rs*. *rs* ! *i* $\in$ *get-return-edges* (*cs* ! *i*);
valid-return-list rs m; *length rs* = *length cs*; *ms''* = *targetnodes rs*;
$\forall$ *mx*$\in$*set ms*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$ *mx'* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$;
ms'' $\neq []$ $\longrightarrow$
($\exists$ *mx'*. *call-of-return-node* (*last ms''*) *mx'* $\wedge$ *mx'* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$);
$\forall$ *mx*$\in$*set ms'*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$ *mx'* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$⟧
$\Longrightarrow$ *same-level-path-aux cs as* $\wedge$ *upd-cs cs as* = [] $\wedge$ *m* $-as\rightarrow*$ *m'* $\wedge$ *ms* = *ms'*›
note *callstack* = ‹$\forall$ *mx*$\in$*set ms*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$
mx' $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
note *callstack''* = ‹*ms''* $\neq []$ $\longrightarrow$
($\exists$ *mx'*. *call-of-return-node* (*last ms''*) *mx'* $\wedge$ *mx'* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$)›
note *callstack'* = ‹$\forall$ *mx*$\in$*set ms'*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$
mx' $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
have *ms''* $\neq []$
proof
assume *ms''* = []

with *callstack*
⟨$\exists\, m{\in}set\ (tl\ (m\ \#\ ms''\ @\ ms)).\ \exists\, m'.\ call\text{-}of\text{-}return\text{-}node\ m\ m' \wedge m' \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$⟩
show *False* **by** *fastforce*
qed
with ⟨$hd\ (tl\ (m\ \#\ ms''\ @\ ms)) = targetnode\ a$⟩
obtain *xs* **where** $ms'' = targetnode\ a\ \#\ xs$ **by**(*cases ms''*) *auto*
with ⟨$ms'' = targetnodes\ rs$⟩ **obtain** $r'\ rs'$ **where** $rs = r'\ \#\ rs'$
and $targetnode\ a = targetnode\ r'$ **and** $xs = targetnodes\ rs'$
by(*cases rs*)(*auto simp*:*targetnodes-def*)
from ⟨$rs = r'\ \#\ rs'$⟩ ⟨$length\ rs = length\ cs$⟩ **obtain** $c'\ cs'$ **where** $cs = c'\ \#\ cs'$
and $length\ rs' = length\ cs'$ **by**(*cases cs*) *auto*
from ⟨$\forall\, i{<}length\ rs.\ rs\ !\ i \in get\text{-}return\text{-}edges\ (cs\ !\ i)$⟩
⟨$rs = r'\ \#\ rs'$⟩ ⟨$cs = c'\ \#\ cs'$⟩
have $\forall\, i{<}length\ rs'.\ rs'\ !\ i \in get\text{-}return\text{-}edges\ (cs'\ !\ i)$
and $r' \in get\text{-}return\text{-}edges\ c'$ **by** *auto*
from ⟨*valid-edge a*⟩ **have** $valid\text{-}node\ (targetnode\ a)$ **by** *simp*
from ⟨$hd\ (m\ \#\ ms''\ @\ ms) = sourcenode\ a$⟩ **have** $m = sourcenode\ a$
by *simp*
from ⟨*valid-call-list cs m*⟩ ⟨$cs = c'\ \#\ cs'$⟩
obtain $p'\ Q'\ r\ fs'$ **where** *valid-edge c'* **and** $kind\ c' = Q'{:}r{\hookrightarrow}_{p'}fs'$
and $p' = get\text{-}proc\ m$
apply(*auto simp*:*valid-call-list-def*)
by(*erule-tac x*=[] **in** *allE*) *auto*
from ⟨*valid-edge a*⟩ ⟨$kind\ a = Q{\leftarrow}_p f'$⟩
have $get\text{-}proc\ (sourcenode\ a) = p$ **by**(*rule get-proc-return*)
with ⟨$m = sourcenode\ a$⟩ ⟨$p' = get\text{-}proc\ m$⟩ **have** $[simp]{:}p' = p$ **by** *simp*
from ⟨*valid-edge c'*⟩ ⟨$kind\ c' = Q'{:}r{\hookrightarrow}_{p'}fs'$⟩
have $get\text{-}proc\ (targetnode\ c') = p$ **by**(*fastforce intro*:*get-proc-call*)
from ⟨*valid-edge c'*⟩ ⟨$r' \in get\text{-}return\text{-}edges\ c'$⟩ **have** *valid-edge r'*
by(*rule get-return-edges-valid*)
from ⟨*valid-edge c'*⟩ ⟨$kind\ c' = Q'{:}r{\hookrightarrow}_{p'}fs'$⟩ ⟨$r' \in get\text{-}return\text{-}edges\ c'$⟩
obtain $Q''\ f''$ **where** $kind\ r' = Q''{\hookleftarrow}_p f''$ **by**(*fastforce dest*!:*call-return-edges*)
with ⟨*valid-edge r'*⟩ **have** $get\text{-}proc\ (sourcenode\ r') = p$ **by**(*rule get-proc-return*)
from ⟨*valid-edge r'*⟩ ⟨$kind\ r' = Q''{\hookleftarrow}_p f''$⟩ **have** $method\text{-}exit\ (sourcenode\ r')$
by(*fastforce simp*:*method-exit-def*)
from ⟨*valid-edge a*⟩ ⟨$kind\ a = Q{\leftarrow}_p f'$⟩ **have** $method\text{-}exit\ (sourcenode\ a)$
by(*fastforce simp*:*method-exit-def*)
with ⟨$method\text{-}exit\ (sourcenode\ r')$⟩ ⟨$get\text{-}proc\ (sourcenode\ r') = p$⟩
⟨$get\text{-}proc\ (sourcenode\ a) = p$⟩
have $sourcenode\ a = sourcenode\ r'$ **by**(*fastforce intro*:*method-exit-unique*)
with ⟨*valid-edge a*⟩ ⟨*valid-edge r'*⟩ ⟨$targetnode\ a = targetnode\ r'$⟩
have $a = r'$ **by**(*fastforce intro*:*edge-det*)
from ⟨*valid-edge c'*⟩ ⟨$r' \in get\text{-}return\text{-}edges\ c'$⟩ ⟨$targetnode\ a = targetnode\ r'$⟩
have $get\text{-}proc\ (sourcenode\ c') = get\text{-}proc\ (targetnode\ a)$
by(*fastforce intro*:*get-proc-get-return-edge*)
from ⟨*valid-call-list cs m*⟩ ⟨$cs = c'\ \#\ cs'$⟩
⟨$get\text{-}proc\ (sourcenode\ c') = get\text{-}proc\ (targetnode\ a)$⟩

have *valid-call-list cs′ (targetnode a)*
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*c′ # cs′* **in** *allE*)
by(*case-tac cs′*)(*auto simp*:*sourcenodes-def*)
from ‹*valid-return-list rs m*› ‹*rs = r′ # rs′*› ‹*targetnode a = targetnode r′*›
have *valid-return-list rs′ (targetnode a)*
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*r′ # cs′* **in** *allE*)
by(*case-tac cs′*)(*auto simp*:*targetnodes-def*)
from ‹*msx′ = tl (m # ms′′ @ ms)*› ‹*ms′′ = targetnode a # xs*›
have *msx′ = targetnode a # xs @ ms* **by** *simp*
from *callstack′′* ‹*ms′′ = targetnode a # xs*›
have *xs ≠ []* ⟶
(∃ *mx′. call-of-return-node (last xs) mx′ ∧ mx′ ∉ ⌊HRB-slice S⌋*$_{CFG}$)
by *fastforce*
from *IH*[*OF* ‹*msx′ = targetnode a # xs @ ms*› ‹*valid-node (targetnode a)*›
‹*valid-call-list cs′ (targetnode a)*›
‹∀ *i*<*length rs′. rs′ ! i ∈ get-return-edges (cs′ ! i)*›
‹*valid-return-list rs′ (targetnode a)*› ‹*length rs′ = length cs′*›
‹*xs = targetnodes rs′*› *callstack this callstack′*]
have *same-level-path-aux cs′ as* **and** *upd-cs cs′ as = []*
and *targetnode a −as→∗ m′* **and** *ms = ms′* **by** *simp-all*
from ‹*kind a = Q↩*$_{pf}$ *p′*› ‹*same-level-path-aux cs′ as*› ‹*cs = c′ # cs′*›
‹*r′ ∈ get-return-edges c′*› ‹*a = r′*›
have *same-level-path-aux cs (a # as)* **by** *simp*
moreover
from ‹*upd-cs cs′ as = []*› ‹*kind a = Q↩*$_{pf}$ *p′*› ‹*cs = c′ # cs′*›
have *upd-cs cs (a # as) = []* **by** *simp*
moreover
from ‹*valid-edge a*› ‹*m = sourcenode a*› ‹*targetnode a −as→∗ m′*›
have *m −a # as→∗ m′* **by**(*fastforce intro*:*Cons-path*)
ultimately show *?case* **using** ‹*ms = ms′*› **by** *simp*
qed
qed

lemma *silent-moves-slp*:
⟦*S*,*f* ⊢ (*m*#*ms*,*s*) =*as*⇒$_{\tau}$ (*m′*#*ms′*,*s′*); *valid-node m*;
∀ *mx ∈ set ms*. ∃ *mx′. call-of-return-node mx mx′ ∧ mx′ ∈ ⌊HRB-slice S⌋*$_{CFG}$;
∀ *mx ∈ set ms′*. ∃ *mx′. call-of-return-node mx mx′ ∧ mx′ ∈ ⌊HRB-slice S⌋*$_{CFG}$⟧
⟹ *m −as→*$_{sl}$∗ *m′ ∧ ms = ms′*
by(*fastforce dest*!:*silent-moves-slpa-path*
[*of* - - - [] - - - - - - [] [],*simplified*]
simp:*targetnodes-def valid-call-list-def valid-return-list-def*
same-level-path-def slp-def)

lemma *slpa-silent-moves-callstacks-eq*:
⟦*same-level-path-aux cs as*; *S*,*f* ⊢ (*m*#*msx*@*ms*,*s*) =*as*⇒$_{\tau}$ (*m′*#*ms′*,*s′*);

length ms = length ms'; *valid-call-list cs m*;
$\forall i < length\ rs.\ rs!i \in get\text{-}return\text{-}edges\ (cs!i)$; *valid-return-list rs m*;
length rs = length cs; *msx = targetnodes rs*⟧
⟹ *ms = ms'*
proof(*induct arbitrary*:*m msx s rs rule*:*slpa-induct*)
case (*slpa-empty cs*)
from ‹*S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) =[]⇒$_\tau$ (*m'* # *ms'*,*s'*)›
have *msx*@*ms = ms'* **by**(*fastforce elim*:*silent-moves.cases*)
with ‹*length ms = length ms'*› **show** *?case* **by** *fastforce*
next
case (*slpa-intra cs a as*)
note *IH* = ‹⋀*m msx s rs*. ⟦*S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) =*as*⇒$_\tau$ (*m'* # *ms'*,*s'*);
length ms = length ms'; *valid-call-list cs m*;
$\forall i{<}length\ rs.\ rs\ !\ i \in get\text{-}return\text{-}edges\ (cs\ !\ i)$;
valid-return-list rs m; *length rs = length cs*; *msx = targetnodes rs*⟧
⟹ *ms = ms'*›
from ‹*S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) =*a* # *as*⇒$_\tau$ (*m'* # *ms'*,*s'*)› **obtain** *ms''* *s''*
where *S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) −*a*→$_\tau$ (*ms''*,*s''*)
and *S*,*f* ⊢ (*ms''*,*s''*) =*as*⇒$_\tau$ (*m'* # *ms'*,*s'*)
by(*auto elim*:*silent-moves.cases*)
from ‹*S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) −*a*→$_\tau$ (*ms''*,*s''*)› ‹*intra-kind* (*kind a*)›
have *valid-edge a* **and** [*simp*]:*m = sourcenode a ms'' = targetnode a* # *msx* @ *ms*
by(*fastforce elim*:*silent-move.cases simp*:*intra-kind-def*)+
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
from ‹*valid-call-list cs m*› ‹*m = sourcenode a*›
‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›
have *valid-call-list cs* (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*)
apply(*erule-tac x*=*c* **in** *allE*)
by(*auto split*:*list.split*)
from ‹*valid-return-list rs m*› ‹*m = sourcenode a*›
‹*get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)›
have *valid-return-list rs* (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*) *auto*
from ‹*S*,*f* ⊢ (*ms''*,*s''*) =*as*⇒$_\tau$ (*m'* # *ms'*,*s'*)›
have *S*,*f* ⊢ (*targetnode a* # *msx* @ *ms*,*s''*) =*as*⇒$_\tau$ (*m'* # *ms'*,*s'*) **by** *simp*
from *IH*[*OF this* ‹*length ms = length ms'*› ‹*valid-call-list cs* (*targetnode a*)›
‹$\forall i{<}length\ rs.\ rs\ !\ i \in get\text{-}return\text{-}edges\ (cs\ !\ i)$›
‹*valid-return-list rs* (*targetnode a*)› ‹*length rs = length cs*›
‹*msx = targetnodes rs*›] **show** *?case* **.**
next
case (*slpa-Call cs a as Q r p fs*)
note *IH* = ‹⋀*m msx s rs*. ⟦*S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) =*as*⇒$_\tau$ (*m'* # *ms'*,*s'*);
length ms = length ms'; *valid-call-list* (*a* # *cs*) *m*;

$\forall i{<}length\ rs.\ rs\ !\ i \in get\text{-}return\text{-}edges\ ((a\ \#\ cs)\ !\ i)$;
valid-return-list rs m; *length rs* = *length* (*a* # *cs*);
msx = *targetnodes rs*⟧
⟹ *ms* = *ms'*⟩
from ⟨*S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) =*a* # *as*⇒$_\tau$ (*m'* # *ms'*,*s'*)⟩ **obtain** *ms''* *s''*
where *S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) −*a*→$_\tau$ (*ms''*,*s''*)
and *S*,*f* ⊢ (*ms''*,*s''*) =*as*⇒$_\tau$ (*m'* # *ms'*,*s'*)
by(*auto elim*:*silent-moves.cases*)
from ⟨*S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) −*a*→$_\tau$ (*ms''*,*s''*)⟩ ⟨*kind a* = *Q*:*r*↪$_p$*fs*⟩
obtain *a'* **where** *valid-edge a* **and** [*simp*]:*m* = *sourcenode a*
and [*simp*]:*ms''* = *targetnode a* # *targetnode a'* # *msx* @ *ms*
and *a'* ∈ *get-return-edges a*
by(*auto elim*:*silent-move.cases simp*:*intra-kind-def*)
from ⟨*valid-edge a*⟩ ⟨*kind a* = *Q*:*r*↪$_p$*fs*⟩ **have** *get-proc* (*targetnode a*) = *p*
by(*rule get-proc-call*)
with ⟨*valid-call-list cs m*⟩ ⟨*valid-edge a*⟩ ⟨*kind a* = *Q*:*r*↪$_p$*fs*⟩ ⟨*m* = *sourcenode a*⟩
have *valid-call-list* (*a* # *cs*) (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x*=*list* **in** *allE*)
by(*case-tac list*)(*auto simp*:*sourcenodes-def*)
from ⟨∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)⟩ ⟨*a'* ∈ *get-return-edges a*⟩
have ∀ *i*<*length* (*a'*#*rs*). (*a'*#*rs*) ! *i* ∈ *get-return-edges* ((*a*#*cs*) ! *i*)
by *auto*(*case-tac i*,*auto*)
from ⟨*valid-edge a*⟩ ⟨*a'* ∈ *get-return-edges a*⟩ **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ⟨*valid-edge a*⟩ ⟨*kind a* = *Q*:*r*↪$_p$*fs*⟩ ⟨*a'* ∈ *get-return-edges a*⟩
obtain *Q' f'* **where** *kind a'* = *Q'*↩$_p$*f'* **by**(*fastforce dest*!:*call-return-edges*)
from ⟨*valid-edge a'*⟩ ⟨*kind a'* = *Q'*↩$_p$*f'*⟩ **have** *get-proc* (*sourcenode a'*) = *p*
by(*rule get-proc-return*)
from ⟨*valid-edge a*⟩ ⟨*a'* ∈ *get-return-edges a*⟩
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a'*)
by(*rule get-proc-get-return-edge*)
with ⟨*valid-return-list rs m*⟩ ⟨*valid-edge a'*⟩ ⟨*kind a'* = *Q'*↩$_p$*f'*⟩
⟨*get-proc* (*sourcenode a'*) = *p*⟩ ⟨*get-proc* (*targetnode a*) = *p*⟩ ⟨*m* = *sourcenode a*⟩
have *valid-return-list* (*a'*#*rs*) (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x*=*list* **in** *allE*)
by(*case-tac list*)(*auto simp*:*targetnodes-def*)
from ⟨*length rs* = *length cs*⟩ **have** *length* (*a'*#*rs*) = *length* (*a*#*cs*) **by** *simp*
from ⟨*msx* = *targetnodes rs*⟩ **have** *targetnode a'* # *msx* = *targetnodes* (*a'* # *rs*)
by(*simp add*:*targetnodes-def*)
from ⟨*S*,*f* ⊢ (*ms''*,*s''*) =*as*⇒$_\tau$ (*m'* # *ms'*,*s'*)⟩
have *S*,*f* ⊢ (*targetnode a* # (*targetnode a'* # *msx*) @ *ms*,*s''*) =*as*⇒$_\tau$ (*m'* # *ms'*,*s'*)
by *simp*
from *IH*[*OF this* ⟨*length ms* = *length ms'*⟩ ⟨*valid-call-list* (*a* # *cs*) (*targetnode a*)⟩

⟨$\forall$ i<*length* (*a'*#*rs*). (*a'*#*rs*) ! i ∈ *get-return-edges* ((*a*#*cs*) ! i)⟩
⟨*valid-return-list* (*a'*#*rs*) (*targetnode a*)⟩ ⟨*length* (*a'*#*rs*) = *length* (*a*#*cs*)⟩
⟨*targetnode a'* # *msx* = *targetnodes* (*a'* # *rs*)⟩] **show** *?case* **.**
next
case (*slpa-Return cs a as Q p f' c' cs'*)
note *IH* = ⟨$\bigwedge$*m msx s rs*. ⟦*S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) =*as*⇒$_\tau$ (*m'* # *ms'*,*s'*);
length ms = *length ms'*; *valid-call-list cs' m*;
$\forall$ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs'* ! *i*); *valid-return-list rs m*;
length rs = *length cs'*; *msx* = *targetnodes rs*⟧
⟹ *ms* = *ms'*⟩
from ⟨*S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) =*a* # *as*⇒$_\tau$ (*m'* # *ms'*,*s'*)⟩ **obtain** *ms''* *s''*
where *S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) −*a*→$_\tau$ (*ms''*,*s''*)
and *S*,*f* ⊢ (*ms''*,*s''*) =*as*⇒$_\tau$ (*m'* # *ms'*,*s'*)
by(*auto elim*:*silent-moves.cases*)
from ⟨*S*,*f* ⊢ (*m* # *msx* @ *ms*,*s*) −*a*→$_\tau$ (*ms''*,*s''*)⟩ ⟨*kind a* = *Q*↩$_p$*f'*⟩
have *valid-edge a* **and** *m* = *sourcenode a* **and** *hd* (*msx* @ *ms*) = *targetnode a*
and *ms''* = *msx* @ *ms* **and** *s''* ≠ [] **and** *length s* = *Suc*(*length s''*)
and *length* (*m* # *msx* @ *ms*) = *length s*
by(*auto elim*:*silent-move.cases simp*:*intra-kind-def*)
from ⟨*msx* = *targetnodes rs*⟩ ⟨*length rs* = *length cs*⟩ ⟨*cs* = *c'* # *cs'*⟩
obtain *mx'* *msx'* **where** *msx* = *mx'*#*msx'*
by(*cases msx*)(*fastforce simp*:*targetnodes-def*)+
with ⟨*hd* (*msx* @ *ms*) = *targetnode a*⟩ **have** *mx'* = *targetnode a* **by** *simp*
from ⟨*valid-call-list cs m*⟩ ⟨*cs* = *c'* # *cs'*⟩ **have** *valid-edge c'*
by(*fastforce simp*:*valid-call-list-def*)
from ⟨*valid-edge c'*⟩ ⟨*a* ∈ *get-return-edges c'*⟩
have *get-proc* (*sourcenode c'*) = *get-proc* (*targetnode a*)
by(*rule get-proc-get-return-edge*)
from ⟨*valid-call-list cs m*⟩ ⟨*cs* = *c'* # *cs'*⟩
⟨*get-proc* (*sourcenode c'*) = *get-proc* (*targetnode a*)⟩
have *valid-call-list cs'* (*targetnode a*)
apply(*clarsimp simp*:*valid-call-list-def*)
apply(*erule-tac x*=*c'* # *cs'* **in** *allE*)
by(*case-tac cs'*)(*auto simp*:*sourcenodes-def*)
from ⟨*length rs* = *length cs*⟩ ⟨*cs* = *c'* # *cs'*⟩ **obtain** *r'* *rs'*
where [*simp*]:*rs* = *r'*#*rs'* **and** *length rs'* = *length cs'* **by**(*cases rs*) *auto*
from ⟨$\forall$ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)⟩ ⟨*cs* = *c'* # *cs'*⟩
have $\forall$ *i*<*length rs'*. *rs'* ! *i* ∈ *get-return-edges* (*cs'* ! *i*)
and *r'* ∈ *get-return-edges c'* **by** *auto*
with ⟨*valid-edge c'*⟩ ⟨*a* ∈ *get-return-edges c'*⟩ **have** [*simp*]:*a* = *r'*
by −(*rule get-return-edges-unique*)
with ⟨*valid-return-list rs m*⟩
have *valid-return-list rs'* (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*r'* # *cs'* **in** *allE*)
by(*case-tac cs'*)(*auto simp*:*targetnodes-def*)
from ⟨*msx* = *targetnodes rs*⟩ ⟨*msx* = *mx'*#*msx'*⟩ ⟨*rs* = *r'*#*rs'*⟩
have *msx'* = *targetnodes rs'* **by**(*simp add*:*targetnodes-def*)
from ⟨*S*,*f* ⊢ (*ms''*,*s''*) =*as*⇒$_\tau$ (*m'* # *ms'*,*s'*)⟩ ⟨*msx* = *mx'*#*msx'*⟩

```
    ‹ms'' = msx @ ms› ‹mx' = targetnode a›
  have S,f ⊢ (targetnode a # msx' @ ms,s'') =as⇒τ (m' # ms',s') by simp
  from IH[OF this ‹length ms = length ms'› ‹valid-call-list cs' (targetnode a)›
    ‹∀ i<length rs'. rs' ! i ∈ get-return-edges (cs' ! i)›
    ‹valid-return-list rs' (targetnode a)› ‹length rs' = length cs'›
    ‹msx' = targetnodes rs'›] show ?case .
qed


lemma silent-moves-same-level-path:
  assumes S,kind ⊢ (m#ms,s) =as⇒τ (m'#ms',s') and m −as→sl* m' shows
ms = ms'
proof −
  from ‹S,kind ⊢ (m#ms,s) =as⇒τ (m'#ms',s')› obtain cf cfs where s = cf#cfs
    by(cases s)(auto dest:silent-moves-equal-length)
  with ‹S,kind ⊢ (m#ms,s) =as⇒τ (m'#ms',s')›
  have transfers (kinds as) (cf#cfs) = s'
    by(fastforce intro:silent-moves-preds-transfers simp:kinds-def)
  with ‹m −as→sl* m'› obtain cf' where s' = cf'#cfs
    by −(drule slp-callstack-length-equal,auto)
  from ‹S,kind ⊢ (m#ms,s) =as⇒τ (m'#ms',s')›
  have length (m#ms) = length s and length (m'#ms') = length s'
    by(rule silent-moves-equal-length)+
  with ‹s = cf#cfs› ‹s' = cf'#cfs› have length ms = length ms' by simp
  from ‹m −as→sl* m'› have same-level-path-aux [] as
    by(simp add:slp-def same-level-path-def)
  with ‹S,kind ⊢ (m#ms,s) =as⇒τ (m'#ms',s')› ‹length ms = length ms'›
  show ?thesis by(auto elim!:slpa-silent-moves-callstacks-eq
    simp:targetnodes-def valid-call-list-def valid-return-list-def)
qed


lemma silent-moves-call-edge:
  assumes S,kind ⊢ (m#ms,s) =as⇒τ (m'#ms',s') and valid-node m
  and callstack:∀ mx ∈ set ms. ∃ mx'. call-of-return-node mx mx' ∧
      mx' ∈ ⌊HRB-slice S⌋CFG
  and rest:∀ i < length rs. rs!i ∈ get-return-edges (cs!i)
  ms = targetnodes rs valid-return-list rs m length rs = length cs
  obtains as' a as'' where as = as'@a#as'' and ∃ 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(atomize-elim)
  from ‹S,kind ⊢ (m#ms,s) =as⇒τ (m'#ms',s')›
  show (∃ as' a as''. as = as' @ a # as'' ∧ (∃ Q r p fs. kind a = Q:r↪pfs) ∧
    call-of-return-node (hd ms') (sourcenode a) ∧ targetnode a −as''→sl* m') ∨
    ms' = ms
  proof(induct as arbitrary:m' ms' s' rule:length-induct)
    fix as m' ms' s'
```

assume *IH*:$\forall\, as'.\ length\ as' < length\ as \longrightarrow$
$(\forall\, mx\ msx\ sx.\ S,kind \vdash (m\#ms,s) =as'\Rightarrow_\tau (mx\#msx,sx) \longrightarrow$
$(\exists\, asx\ a\ asx'.\ as' = asx\ @\ a\ \#\ asx' \wedge (\exists\, Q\ r\ p\ fs.\ kind\ a = Q{:}r{\hookrightarrow}_p fs) \wedge$
$call\text{-}of\text{-}return\text{-}node\ (hd\ msx)\ (sourcenode\ a) \wedge targetnode\ a\ -asx'{\rightarrow}_{sl^*}\ mx) \vee$
$msx = ms)$
and $S,kind \vdash (m\#ms,s) =as\Rightarrow_\tau (m'\#ms',s')$
show $(\exists\, as'\ a\ as''.\ as = as'\ @\ a\ \#\ as'' \wedge (\exists\, Q\ r\ p\ fs.\ kind\ a = Q{:}r{\hookrightarrow}_p fs) \wedge$
$call\text{-}of\text{-}return\text{-}node\ (hd\ ms')\ (sourcenode\ a) \wedge targetnode\ a\ -as''{\rightarrow}_{sl^*}\ m') \vee$
$ms' = ms$
proof(*cases as rule*:*rev-cases*)
case *Nil*
with ‹$S,kind \vdash (m\#ms,s) =as\Rightarrow_\tau (m'\#ms',s')$› **have** $ms = ms'$
by(*fastforce elim*:*silent-moves.cases*)
thus *?thesis* **by** *simp*
next
case (*snoc as' a'*)
with ‹$S,kind \vdash (m\#ms,s) =as\Rightarrow_\tau (m'\#ms',s')$›
obtain $ms''\ s''$ **where** $S,kind \vdash (m\#ms,s) =as'\Rightarrow_\tau (ms'',s'')$
and $S,kind \vdash (ms'',s'') =[a']\Rightarrow_\tau (m'\#ms',s')$
by(*fastforce elim*:*silent-moves-split*)
from *snoc* **have** $length\ as' < length\ as$ **by** *simp*
from ‹$S,kind \vdash (ms'',s'') =[a']\Rightarrow_\tau (m'\#ms',s')$›
have $S,kind \vdash (ms'',s'') -a'{\rightarrow}_\tau (m'\#ms',s')$
by(*fastforce elim*:*silent-moves.cases*)
show *?thesis*
proof(*cases kind a' rule*:*edge-kind-cases*)
case *Intra*
with ‹$S,kind \vdash (ms'',s'') -a'{\rightarrow}_\tau (m'\#ms',s')$›
have *valid-edge a'* **and** $m' = targetnode\ a'$
by(*auto elim*:*silent-move.cases simp*:*intra-kind-def*)
from ‹$S,kind \vdash (ms'',s'') -a'{\rightarrow}_\tau (m'\#ms',s')$› ‹*intra-kind* (*kind a'*)›
have $ms'' = sourcenode\ a'\#ms'$
by −(*erule silent-move.cases*,*auto simp*:*intra-kind-def*,(*cases ms''*,*auto*)+)
with *IH* ‹$length\ as' < length\ as$› ‹$S,kind \vdash (m\#ms,s) =as'\Rightarrow_\tau (ms'',s'')$›
have $(\exists\, asx\ ax\ asx'.\ as' = asx\ @\ ax\ \#\ asx' \wedge (\exists\, Q\ r\ p\ fs.\ kind\ ax =$
$Q{:}r{\hookrightarrow}_p fs) \wedge$
$call\text{-}of\text{-}return\text{-}node\ (hd\ ms')\ (sourcenode\ ax) \wedge$
$targetnode\ ax\ -asx'{\rightarrow}_{sl^*}\ sourcenode\ a') \vee ms' = ms$
by *simp blast*
thus *?thesis*
proof
assume $\exists\, asx\ ax\ asx'.\ as' = asx\ @\ ax\ \#\ asx' \wedge$
$(\exists\, Q\ r\ p\ fs.\ kind\ ax = Q{:}r{\hookrightarrow}_p fs) \wedge$
$call\text{-}of\text{-}return\text{-}node\ (hd\ ms')\ (sourcenode\ ax) \wedge$
$targetnode\ ax\ -asx'{\rightarrow}_{sl^*}\ sourcenode\ a'$
then obtain $asx\ ax\ asx'$ **where** $as' = asx\ @\ ax\ \#\ asx'$
and $\exists\, Q\ r\ p\ fs.\ kind\ ax = Q{:}r{\hookrightarrow}_p fs$
and $call\text{-}of\text{-}return\text{-}node\ (hd\ ms')\ (sourcenode\ ax)$
and $targetnode\ ax\ -asx'{\rightarrow}_{sl^*}\ sourcenode\ a'$

by *blast*
from ‹$as' = asx$ @ ax # asx'› **have** $as'@[a'] = asx$ @ ax # (asx' @ $[a']$)
by *simp*
moreover
from ‹*targetnode ax* $-asx'\rightarrow_{sl}*$ *sourcenode a'*› ‹*intra-kind* (*kind a'*)›
‹$m' =$ *targetnode a'*› ‹*valid-edge a'*›
have *targetnode ax* $-asx'@[a']\rightarrow_{sl}*$ m'
by(*fastforce intro*:*path-Append path-edge same-level-path-aux-Append upd-cs-Append simp*:*slp-def same-level-path-def intra-kind-def*)
ultimately show *?thesis* **using** ‹$\exists Q\ r\ p\ fs.$ *kind ax* = $Q:r\hookrightarrow_p fs$›
‹*call-of-return-node* (*hd ms'*) (*sourcenode ax*)› *snoc* **by** *blast*
next
assume $ms' = ms$ **thus** *?thesis* **by** *simp*
qed
next
case (*Call Q r p fs*)
with ‹S,*kind* ⊢ (ms'',s'') $-a'\rightarrow_\tau$ ($m'\#ms'$,s')› **obtain** a''
where *valid-edge a'* **and** $a'' \in$ *get-return-edges a'*
and *hd ms''* = *sourcenode a'* **and** $m' =$ *targetnode a'*
and $ms' =$ (*targetnode a''*)#*tl ms''* **and** *length ms''* = *length s''*
and *pred* (*kind a'*) *s''*
by(*auto elim*:*silent-move.cases simp*:*intra-kind-def*)
from ‹*valid-edge a'*› ‹$a'' \in$ *get-return-edges a'*› **have** *valid-edge a''*
by(*rule get-return-edges-valid*)
from ‹*valid-edge a''*› ‹*valid-edge a'*› ‹$a'' \in$ *get-return-edges a'*›
have *return-node* (*targetnode a''*) **by**(*fastforce simp*:*return-node-def*)
with ‹*valid-edge a'*› ‹*valid-edge a''*›
‹$a'' \in$ *get-return-edges a'*› ‹$ms' =$ (*targetnode a''*)#*tl ms''*›
have *call-of-return-node* (*hd ms'*) (*sourcenode a'*)
by(*simp add*:*call-of-return-node-def*) *blast*
with *snoc* ‹*kind a'* = $Q:r\hookrightarrow_p fs$› ‹$m' =$ *targetnode a'*› ‹*valid-edge a'*›
show *?thesis* **by**(*fastforce intro*:*empty-path simp*:*slp-def same-level-path-def*)
next
case (*Return Q p f*)
with ‹S,*kind* ⊢ (ms'',s'') $-a'\rightarrow_\tau$ ($m'\#ms'$,s')›
have *valid-edge a'* **and** *hd ms''* = *sourcenode a'*
and *hd*(*tl ms''*) = *targetnode a'* **and** $m'\#ms' =$ *tl ms''*
and *length ms''* = *length s''* **and** *length s''* = *Suc*(*length s'*)
and $s' \neq []$
by(*auto elim*:*silent-move.cases simp*:*intra-kind-def*)
hence *ms''* = *sourcenode a'* # *targetnode a'* # *ms'* **by**(*cases ms''*) *auto*
with ‹*length as'* < *length as*› ‹S,*kind* ⊢ ($m\#ms$,s) =as'⇒$_\tau$ (ms'',s'')› *IH*
have ($\exists$ *asx ax asx'*. *as'* = *asx* @ *ax* # *asx'* ∧ ($\exists Q\ r\ p\ fs.$ *kind ax* = $Q:r\hookrightarrow_p fs$) ∧
call-of-return-node (*targetnode a'*) (*sourcenode ax*) ∧
targetnode ax $-asx'\rightarrow_{sl}*$ *sourcenode a'*) ∨ *ms* = *targetnode a'* # *ms'*
apply − **apply**(*erule-tac x*=*as'* **in** *allE*) **apply** *clarsimp*
apply(*erule-tac x*=*sourcenode a'* **in** *allE*)
apply(*erule-tac x*=*targetnode a'* # *ms'* **in** *allE*)

```
    by fastforce
  thus ?thesis
  proof
    assume ∃ asx ax asx'. as' = asx @ ax # asx' ∧
      (∃ Q r p fs. kind ax = Q:r↪p fs) ∧
      call-of-return-node (targetnode a') (sourcenode ax) ∧
      targetnode ax −asx'→sl* sourcenode a'
    then obtain asx ax asx' where as' = asx @ ax # asx'
      and ∃ Q r p fs. kind ax = Q:r↪p fs
      and call-of-return-node (targetnode a') (sourcenode ax)
      and targetnode ax −asx'→sl* sourcenode a' by blast
    from ‹as' = asx @ ax # asx'› snoc have length asx < length as by simp
    moreover
    from ‹S,kind ⊢ (m#ms,s) =as⇒τ (m'#ms',s')› snoc ‹as' = asx @ ax #
asx'›
    obtain msx sx where S,kind ⊢ (m#ms,s) =asx⇒τ (msx,sx)
      and S,kind ⊢ (msx,sx) =ax#asx'@[a']⇒τ (m'#ms',s')
      by(fastforce elim:silent-moves-split)
    from ‹S,kind ⊢ (msx,sx) =ax#asx'@[a']⇒τ (m'#ms',s')›
    obtain xs x ys y where S,kind ⊢ (msx,sx) −ax→τ (xs,x)
      and S,kind ⊢ (xs,x) =asx'⇒τ (ys,y)
      and S,kind ⊢ (ys,y) =[a']⇒τ (m'#ms',s')
      apply − apply(erule silent-moves.cases) apply auto
      by(erule silent-moves-split) auto
    from ‹S,kind ⊢ (msx,sx) −ax→τ (xs,x)› ‹∃ Q r p fs. kind ax = Q:r↪p fs›
    obtain msx' ax' where msx = sourcenode ax#msx'
      and ax' ∈ get-return-edges ax
      and [simp]:xs = (targetnode ax)#(targetnode ax')#msx'
      and length x = Suc(length sx) and length msx = length sx
     apply − apply(erule silent-move.cases) apply(auto simp:intra-kind-def)
      by(cases msx,auto)+
    from ‹S,kind ⊢ (ys,y) =[a']⇒τ (m'#ms',s')› obtain msy
      where ys = sourcenode a'#msy
      apply − apply(erule silent-moves.cases) apply auto
      apply(erule silent-move.cases)
      by(cases ys,auto)+
    with ‹S,kind ⊢ (xs,x) =asx'⇒τ (ys,y)›
      ‹targetnode ax −asx'→sl* sourcenode a'›
      ‹xs = (targetnode ax)#(targetnode ax')#msx'›
    have (targetnode ax')#msx' = msy apply simp
      by(fastforce intro:silent-moves-same-level-path)
    with ‹S,kind ⊢ (ys,y) =[a']⇒τ (m'#ms',s')› ‹kind a' = Q↩pf›
      ‹ys = sourcenode a'#msy›
    have m' = targetnode a' and msx' = ms'
      by(fastforce elim:silent-moves.cases silent-move.cases
                  simp:intra-kind-def)+
  with ‹S,kind ⊢ (m#ms,s) =asx⇒τ (msx,sx)› ‹msx = sourcenode ax#msx'›
    have S,kind ⊢ (m#ms,s) =asx⇒τ (sourcenode ax#ms',sx) by simp
    ultimately have (∃ xs x xs'. asx = xs@x#xs' ∧
```

$(\exists\, Q\ r\ p\ fs.\ kind\ x = Q{:}r{\hookrightarrow_p}fs) \wedge$
$call\text{-}of\text{-}return\text{-}node\ (hd\ ms')\ (sourcenode\ x) \wedge$
$targetnode\ x\ -xs'{\rightarrow_{sl}*}\ sourcenode\ ax) \vee ms = ms'$ **using** *IH*
by *simp blast*
thus *?thesis*
proof
assume $\exists\, xs\ x\ xs'.\ asx = xs@x\#xs' \wedge (\exists\, Q\ r\ p\ fs.\ kind\ x = Q{:}r{\hookrightarrow_p}fs) \wedge$
$call\text{-}of\text{-}return\text{-}node\ (hd\ ms')\ (sourcenode\ x) \wedge$
$targetnode\ x\ -xs'{\rightarrow_{sl}*}\ sourcenode\ ax$
then obtain *xs x xs'* **where** $asx = xs@x\#xs'$
and $\exists\, Q\ r\ p\ fs.\ kind\ x = Q{:}r{\hookrightarrow_p}fs$
and $call\text{-}of\text{-}return\text{-}node\ (hd\ ms')\ (sourcenode\ x)$
and $targetnode\ x\ -xs'{\rightarrow_{sl}*}\ sourcenode\ ax$ **by** *blast*
from $\langle asx = xs@x\#xs'\rangle\ \langle as' = asx\ @\ ax\ \#\ asx'\rangle$ *snoc*
have $as = xs@x\#(xs'@ax\#asx'@[a'])$ **by** *simp*
from $\langle S{,}kind \vdash (m\#ms{,}s) =as{\Rightarrow_\tau}\ (m'\#ms'{,}s')\rangle\ \langle valid\text{-}node\ m\rangle$ *rest*
have $m\ -as{\rightarrow}*\ m'$ **and** *valid-path-aux cs as*
by(*auto dest*:*silent-moves-vpa-path*[*of - - - - - - - - - - rs cs*]
simp:*valid-call-list-def valid-return-list-def targetnodes-def*)
hence $m\ -as{\rightarrow_\surd}*\ m'$
by(*fastforce intro*:*valid-path-aux-valid-path simp*:*vp-def*)
with *snoc* **have** $m\ -as'{\rightarrow_\surd}*\ sourcenode\ a'$
by(*auto elim*:*path-split-snoc dest*:*valid-path-aux-split*
simp:*vp-def valid-path-def*)
with $\langle as' = asx\ @\ ax\ \#\ asx'\rangle$
have *valid-edge ax* **and** $targetnode\ ax\ -asx'{\rightarrow}*\ sourcenode\ a'$
by(*auto dest*:*path-split simp*:*vp-def*)
hence $sourcenode\ ax\ -ax\#asx'{\rightarrow}*\ sourcenode\ a'$
by(*fastforce intro*:*Cons-path*)
from $\langle valid\text{-}edge\ a'\rangle$ **have** $sourcenode\ a'\ -[a']{\rightarrow}*\ targetnode\ a'$
by(*rule path-edge*)
with $\langle sourcenode\ ax\ -ax\#asx'{\rightarrow}*\ sourcenode\ a'\rangle$
have $sourcenode\ ax\ -(ax\#asx')@[a']{\rightarrow}*\ targetnode\ a'$
by(*rule path-Append*)
from $\langle m\ -as{\rightarrow_\surd}*\ m'\rangle$ *snoc* $\langle as' = asx\ @\ ax\ \#\ asx'\rangle$ *snoc*
have $valid\text{-}path\text{-}aux\ ([]@(upd\text{-}cs\ []\ asx))\ (ax\ \#\ asx'\ @\ [a'])$
by(*fastforce dest*:*valid-path-aux-split simp*:*vp-def valid-path-def*)
hence $valid\text{-}path\text{-}aux\ []\ (ax\#asx'@[a'])$
by(*rule valid-path-aux-callstack-prefix*)
with $\langle \exists\, Q\ r\ p\ fs.\ kind\ ax = Q{:}r{\hookrightarrow_p}fs\rangle$
have $valid\text{-}path\text{-}aux\ [ax]\ (asx'@[a'])$ **by** *fastforce*
hence $valid\text{-}path\text{-}aux\ (upd\text{-}cs\ [ax]\ asx')\ [a']$
by(*rule valid-path-aux-split*)
from $\langle targetnode\ ax\ -asx'{\rightarrow_{sl}*}\ sourcenode\ a'\rangle$
have *same-level-path-aux* $[]\ asx'$ **and** $upd\text{-}cs\ []\ asx' = []$
by(*simp-all add*:*slp-def same-level-path-def*)
hence $upd\text{-}cs\ ([]@[ax])\ asx' = []@[ax]$
by(*rule same-level-path-upd-cs-callstack-Append*)
with $\langle valid\text{-}path\text{-}aux\ (upd\text{-}cs\ [ax]\ asx')\ [a']\rangle$

```
        have valid-path-aux [ax] [a'] by(simp del:valid-path-aux.simps)
        with ⟨∃ Q r p fs. kind ax = Q:r↪pfs⟩ ⟨kind a' = Q↩pf⟩
        have a' ∈ get-return-edges ax by simp
        with ⟨upd-cs ([]@[ax]) asx' = []@[ax]⟩ ⟨kind a' = Q↩pf⟩
        have upd-cs [ax] (asx'@[a']) = [] by(fastforce intro:upd-cs-Append)
        with ⟨∃ Q r p fs. kind ax = Q:r↪pfs⟩
        have upd-cs [] (ax#asx'@[a']) = [] by fastforce
        from ⟨targetnode ax −asx'→sl* sourcenode a'⟩
        have same-level-path-aux [] asx' and upd-cs [] asx' = []
          by(simp-all add:slp-def same-level-path-def)
        hence same-level-path-aux ([]@[ax]) asx'
          by −(rule same-level-path-aux-callstack-Append)
        with ⟨∃ Q r p fs. kind ax = Q:r↪pfs⟩ ⟨kind a' = Q↩pf⟩
          ⟨a' ∈ get-return-edges ax⟩ ⟨upd-cs ([]@[ax]) asx' = []@[ax]⟩
        have same-level-path-aux [] ((ax#asx')@[a'])
          by(fastforce intro:same-level-path-aux-Append)
        with ⟨upd-cs [] (ax#asx'@[a']) = []⟩
          ⟨sourcenode ax −(ax#asx')@[a']→* targetnode a'⟩
        have sourcenode ax −(ax#asx')@[a']→sl* targetnode a'
          by(simp add:slp-def same-level-path-def)
        with ⟨targetnode x −xs'→sl* sourcenode ax⟩
        have targetnode x −xs'@((ax#asx')@[a'])→sl* targetnode a'
          by(rule slp-Append)
        with ⟨∃ Q r p fs. kind x = Q:r↪pfs⟩
          ⟨call-of-return-node (hd ms') (sourcenode x)⟩
          ⟨as = xs@x#(xs'@ax#asx'@[a'])⟩ ⟨m' = targetnode a'⟩
        show ?thesis by simp blast
      next
        assume ms = ms' thus ?thesis by simp
      qed
    next
      assume ms = targetnode a' # ms'
      from ⟨S,kind ⊢ (ms'',s'') −a'→τ (m'#ms',s')⟩ ⟨kind a' = Q↩pf⟩
        ⟨ms'' = sourcenode a' # targetnode a' # ms'⟩
      have ∃ m ∈ set (targetnode a' # ms'). ∃ m'. call-of-return-node m m' ∧
        m' ∉ ⌊HRB-slice S⌋CFG
        by(fastforce elim!:silent-move.cases simp:intra-kind-def)
      with ⟨ms = targetnode a' # ms'⟩ callstack
      have False by fastforce
      thus ?thesis by simp
    qed
  qed
 qed
qed


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 *∀ mx ∈ set ms. ∃ mx′. call-of-return-node mx mx′ ∧ mx′ ∈ ⌊HRB-slice S⌋CFG*
and *∀ i < length rs. rs!i ∈ get-return-edges (cs!i)* **and** *ms = targetnodes rs*
and *valid-return-list rs m* **and** *length rs = length cs*
obtains *as′ a as″* **where** *as = as′@a#as″* **and** *∃ 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(*atomize-elim*)
from ‹*S,kind ⊢ (m#ms,s) =as⇒τ (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*›
have *m −as→* m′*
by(*auto dest*:*silent-moves-vpa-path*[*of - - - - - - - - - rs cs*]
simp:*valid-call-list-def valid-return-list-def targetnodes-def*)
from ‹*S,kind ⊢ (m#ms,s) =as⇒τ (m′#ms′,s′)*› ‹*valid-node m*›
‹*∀ mx ∈ set ms. ∃ mx′. call-of-return-node mx mx′ ∧ mx′ ∈ ⌊HRB-slice S⌋CFG*›
‹*∀ i < length rs. rs!i ∈ get-return-edges (cs!i)*› ‹*ms = targetnodes rs*›
‹*valid-return-list rs m*› ‹*length rs = length cs*›
show *(∃ as′ a as″. as = as′ @ a # as″ ∧ (∃ Q r p fs. kind a = Q:r↪pfs) ∧ call-of-return-node (hd ms′) (sourcenode a) ∧ targetnode a −as″→sl* m′ ∧ CFG-node (sourcenode a) ∈ sum-SDG-slice1 nx) ∨ ms′ = ms*
proof(*rule silent-moves-call-edge*)
fix *as′ a as″* **assume** *as = as′@a#as″* **and** *∃ Q r p fs. kind a = Q:r↪pfs*
and *call-of-return-node (hd ms′) (sourcenode a)*
and *targetnode a −as″→sl* m′*
from ‹*∃ Q r p fs. kind a = Q:r↪pfs*› **obtain** *Q r p fs*
where *kind a = Q:r↪pfs* **by** *blast*
from ‹*targetnode a −as″→sl* m′*› **obtain** *asx* **where** *targetnode a −asx→ι* m′*
by −(*erule same-level-path-inner-path*)
from ‹*m −as→* m′*› ‹*as = as′@a#as″*› **have** *valid-edge a*
by(*fastforce dest*:*path-split simp*:*vp-def*)
have *m′ ≠ (-Exit-)*
proof
assume *m′ = (-Exit-)*
have *get-proc (-Exit-) = Main* **by**(*rule get-proc-Exit*)
from ‹*targetnode a −asx→ι* m′*›
have *get-proc (targetnode a) = get-proc m′* **by**(*rule intra-path-get-procs*)
with ‹*m′ = (-Exit-)*› ‹*get-proc (-Exit-) = Main*›
have *get-proc (targetnode a) = Main* **by** *simp*
with ‹*kind a = Q:r↪pfs*› ‹*valid-edge a*›
have *kind a = Q:r↪Mainfs* **by**(*fastforce dest*:*get-proc-call*)
with ‹*valid-edge a*› **show** *False* **by**(*rule Main-no-call-target*)
qed
show *?thesis*
proof(*cases targetnode a = m′*)

case *True*
with ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*pfs*›
have *CFG-node* (*sourcenode a*) *s*−*p*→$_{call}$ *CFG-node m'*
by(*fastforce intro*:*sum-SDG-call-edge*)
with ‹*CFG-node m'* ∈ *sum-SDG-slice1 nx*›
have *CFG-node* (*sourcenode a*) ∈ *sum-SDG-slice1 nx* **by** −(*rule call-slice1*)
with ‹*as* = *as'*@*a*#*as''*› ‹∃ *Q r p fs*. *kind a* = *Q*:*r*↪*pfs*›
‹*call-of-return-node* (*hd ms'*) (*sourcenode a*)›
‹*targetnode a* −*as''*→$_{sl}$* *m'*› **show** *?thesis* **by** *blast*
next
case *False*
with ‹*targetnode a* −*asx*→$_{\iota}$* *m'*› ‹*m'* ≠ (*-Exit-*)› ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*pfs*›
obtain *ns* **where** *CFG-node* (*targetnode a*) *cd*−*ns*→$_{d}$* *CFG-node m'*
by(*fastforce elim*!:*in-proc-cdep-SDG-path*)
hence *CFG-node* (*targetnode a*) *is*−*ns*→$_{d}$* *CFG-node m'*
by(*fastforce intro*:*intra-SDG-path-is-SDG-path cdep-SDG-path-intra-SDG-path*)
with ‹*CFG-node m'* ∈ *sum-SDG-slice1 nx*›
have *CFG-node* (*targetnode a*) ∈ *sum-SDG-slice1 nx*
by −(*rule is-SDG-path-slice1*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪*pfs*›
have *CFG-node* (*sourcenode a*) *s*−*p*→$_{call}$ *CFG-node* (*targetnode a*)
by(*fastforce intro*:*sum-SDG-call-edge*)
with ‹*CFG-node* (*targetnode a*) ∈ *sum-SDG-slice1 nx*›
have *CFG-node* (*sourcenode a*) ∈ *sum-SDG-slice1 nx* **by** −(*rule call-slice1*)
with ‹*as* = *as'*@*a*#*as''*› ‹∃ *Q r p fs*. *kind a* = *Q*:*r*↪*pfs*›
‹*call-of-return-node* (*hd ms'*) (*sourcenode a*)›
‹*targetnode a* −*as''*→$_{sl}$* *m'*› **show** *?thesis* **by** *blast*
qed
next
assume *ms'* = *ms* **thus** *?thesis* **by** *simp*
qed
qed

lemma *silent-moves-called-node-in-slice1-nodestack-in-slice1*:
⟦*S*,*kind* ⊢ (*m*#*ms*,*s*) =*as*⇒$_{\tau}$ (*m'*#*ms'*,*s'*); *valid-node m*;
CFG-node m' ∈ *sum-SDG-slice1 nx*; *nx* ∈ *S*;
∀ *mx* ∈ *set ms*. ∃ *mx'*. *call-of-return-node mx mx'* ∧ *mx'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$;
∀ *i* < *length rs*. *rs*!*i* ∈ *get-return-edges* (*cs*!*i*); *ms* = *targetnodes rs*;
valid-return-list rs m; *length rs* = *length cs*⟧
⟹ ∀ *mx* ∈ *set ms'*. ∃ *mx'*. *call-of-return-node mx mx'* ∧ *mx'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$
proof(*induct ms' arbitrary*:*as m' s'*)
case (*Cons mx msx*)
note *IH* = ‹⋀*as m' s'*. ⟦*S*,*kind* ⊢ (*m*#*ms*,*s*) =*as*⇒$_{\tau}$ (*m'* # *msx*,*s'*); *valid-node m*;
CFG-node m' ∈ *sum-SDG-slice1 nx*; *nx* ∈ *S*;
∀ *mx*∈*set ms*. ∃ *mx'*. *call-of-return-node mx mx'* ∧ *mx'* ∈ ⌊*HRB-slice S*⌋$_{CFG}$;

$\forall i{<}$*length rs*. *rs* ! $i \in$ *get-return-edges* (*cs* ! i); *ms* = *targetnodes rs*;
valid-return-list rs m; *length rs* = *length cs*⟧
$\Longrightarrow \forall mx{\in}$*set msx*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$ *mx'* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$⟩
from ‹*S*,*kind* $\vdash$ (*m*#*ms*,*s*) =*as*$\Rightarrow_\tau$ (*m'* # *mx* # *msx*,*s'*)› ‹*valid-node m*›
‹*CFG-node m'* $\in$ *sum-SDG-slice1 nx*›
‹$\forall$ *mx* $\in$ *set ms*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$ *mx'* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
‹$\forall i <$ *length rs*. *rs*!*i* $\in$ *get-return-edges* (*cs*!*i*)› ‹*ms* = *targetnodes rs*›
‹*valid-return-list rs m*› ‹*length rs* = *length cs*›
show *?case*
proof(*rule silent-moves-called-node-in-slice1-hd-nodestack-in-slice1*)
fix *as' a as''* **assume** *as* = *as'*@*a*#*as''* **and** $\exists$ *Q r p fs*. *kind a* = *Q*:*r*$\hookrightarrow_p$*fs*
and *call-of-return-node* (*hd* (*mx* # *msx*)) (*sourcenode a*)
and *CFG-node* (*sourcenode a*) $\in$ *sum-SDG-slice1 nx*
and *targetnode a* $-as''\rightarrow_{sl}*$ *m'*
from ‹*CFG-node* (*sourcenode a*) $\in$ *sum-SDG-slice1 nx*› ‹*nx* $\in$ *S*›
have *sourcenode a* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$
by(*fastforce intro*:*combSlice-refl simp*:*SDG-to-CFG-set-def HRB-slice-def*)
from ‹*S*,*kind* $\vdash$ (*m*#*ms*,*s*) =*as*$\Rightarrow_\tau$ (*m'* # *mx* # *msx*,*s'*)› ‹*as* = *as'*@*a*#*as''*›
obtain *xs x* **where** *S*,*kind* $\vdash$ (*m*#*ms*,*s*) =*as'*$\Rightarrow_\tau$ (*xs*,*x*)
and *S*,*kind* $\vdash$ (*xs*,*x*) =*a*#*as''*$\Rightarrow_\tau$ (*m'* # *mx* # *msx*,*s'*)
by(*fastforce elim*:*silent-moves-split*)
from ‹*S*,*kind* $\vdash$ (*xs*,*x*) =*a*#*as''*$\Rightarrow_\tau$ (*m'* # *mx* # *msx*,*s'*)›
obtain *ys y* **where** *S*,*kind* $\vdash$ (*xs*,*x*) $-a\rightarrow_\tau$ (*ys*,*y*)
and *S*,*kind* $\vdash$ (*ys*,*y*) =*as''*$\Rightarrow_\tau$ (*m'* # *mx* # *msx*,*s'*)
by(*fastforce elim*:*silent-moves.cases*)
from ‹*S*,*kind* $\vdash$ (*xs*,*x*) $-a\rightarrow_\tau$ (*ys*,*y*)› ‹$\exists$ *Q r p fs*. *kind a* = *Q*:*r*$\hookrightarrow_p$*fs*›
obtain *xs' a'* **where** *xs* = *sourcenode a*#*xs'*
and *ys* = *targetnode a*#*targetnode a'*#*xs'*
apply − **apply**(*erule silent-move.cases*) **apply**(*auto simp*:*intra-kind-def*)
by(*cases xs*,*auto*)+
from ‹*S*,*kind* $\vdash$ (*ys*,*y*) =*as''*$\Rightarrow_\tau$ (*m'* # *mx* # *msx*,*s'*)›
‹*ys* = *targetnode a*#*targetnode a'*#*xs'*› ‹*targetnode a* $-as''\rightarrow_{sl}*$ *m'*›
have *mx* = *targetnode a'* **and** *xs'* = *msx*
by(*auto dest*:*silent-moves-same-level-path*)
with ‹*xs* = *sourcenode a*#*xs'*› ‹*S*,*kind* $\vdash$ (*m*#*ms*,*s*) =*as'*$\Rightarrow_\tau$ (*xs*,*x*)›
have *S*,*kind* $\vdash$ (*m*#*ms*,*s*) =*as'*$\Rightarrow_\tau$ (*sourcenode a*#*msx*,*x*) **by** *simp*
from *IH*[*OF* ‹*S*,*kind* $\vdash$ (*m*#*ms*,*s*) =*as'*$\Rightarrow_\tau$ (*sourcenode a*#*msx*,*x*)›
‹*valid-node m*› ‹*CFG-node* (*sourcenode a*) $\in$ *sum-SDG-slice1 nx*› ‹*nx* $\in$ *S*›
‹$\forall$ *mx* $\in$ *set ms*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$ *mx'* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
‹$\forall i <$ *length rs*. *rs*!*i* $\in$ *get-return-edges* (*cs*!*i*)› ‹*ms* = *targetnodes rs*›
‹*valid-return-list rs m*› ‹*length rs* = *length cs*›]
have *callstack*:$\forall$ *mx*$\in$*set msx*.
$\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$ *mx'* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$.
with ‹*as* = *as'*@*a*#*as''*› ‹*call-of-return-node* (*hd* (*mx* # *msx*)) (*sourcenode a*)›
‹*sourcenode a* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$› **show** *?thesis* **by** *fastforce*
next
assume *mx* # *msx* = *ms*
with ‹$\forall$ *mx* $\in$ *set ms*. $\exists$ *mx'*. *call-of-return-node mx mx'* $\wedge$ *mx'* $\in \lfloor$*HRB-slice*

$S\rfloor_{CFG}\rangle$
show *?thesis* **by** *fastforce*
qed
qed *simp*

lemma *silent-moves-slice-intra-path*:
assumes S,*slice-kind* $S \vdash (m\#ms,s) =as\Rightarrow_\tau (m'\#ms',s')$
and $\forall\, mx \in set\ ms.\ \exists\, mx'.$ *call-of-return-node* $mx\ mx' \wedge mx' \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$
shows $\forall\, a \in set\ as.$ *intra-kind* (*kind* a)
proof(*rule ccontr*)
assume $\neg\ (\forall\, a{\in}set\ as.$ *intra-kind* (*kind* a))
hence $\exists\, a \in set\ as.\ \neg$ *intra-kind* (*kind* a) **by** *fastforce*
then obtain *asx ax asx'* **where** $as = asx@ax\#asx'$
and $\forall\, a{\in}set\ asx.$ *intra-kind* (*kind* a) **and** $\neg$ *intra-kind* (*kind* ax)
by(*fastforce elim*!:*split-list-first-propE*)
from ‹S,*slice-kind* $S \vdash (m\#ms,s) =as\Rightarrow_\tau (m'\#ms',s')$› ‹$as = asx@ax\#asx'$›
obtain *msx sx msx' sx'* **where** S,*slice-kind* $S \vdash (m\#ms,s) =asx\Rightarrow_\tau (msx,sx)$
and S,*slice-kind* $S \vdash (msx,sx) -ax\rightarrow_\tau (msx',sx')$
and S,*slice-kind* $S \vdash (msx',sx') =asx'\Rightarrow_\tau (m'\#ms',s')$
by(*auto elim*!:*silent-moves-split elim*:*silent-moves.cases*)
from ‹S,*slice-kind* $S \vdash (msx,sx) -ax\rightarrow_\tau (msx',sx')$› **obtain** *xs*
where [*simp*]:$msx =$ *sourcenode* $ax\#xs$ **by**(*cases msx*)(*auto elim*:*silent-move.cases*)
from ‹S,*slice-kind* $S \vdash (m\#ms,s) =asx\Rightarrow_\tau (msx,sx)$› ‹$\forall\, a{\in}set\ asx.$ *intra-kind* (*kind* a)›
have [*simp*]:$xs = ms$ **by**(*fastforce dest*:*silent-moves-intra-path*)
show *False*
proof(*cases kind ax rule*:*edge-kind-cases*)
case *Intra* **with** ‹$\neg$ *intra-kind* (*kind* ax)› **show** *False* **by** *simp*
next
case (*Call Q r p fs*)
with ‹S,*slice-kind* $S \vdash (msx,sx) -ax\rightarrow_\tau (msx',sx')$›
‹$\forall\, mx \in set\ ms.\ \exists\, mx'.$ *call-of-return-node* $mx\ mx' \wedge mx' \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$›
have *sourcenode* $ax \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$ **and** *pred* (*slice-kind* S ax) sx
by(*auto elim*!:*silent-move.cases simp*:*intra-kind-def*)
from ‹*sourcenode* $ax \notin \lfloor$*HRB-slice* $S\rfloor_{CFG}$› ‹*kind* $ax = Q$:$r\hookrightarrow_p fs$›
have *slice-kind* S $ax = (\lambda cf.\ False)$:$r\hookrightarrow_p fs$
by(*rule slice-kind-Call*)
with ‹*pred* (*slice-kind* S ax) sx› **show** *False* **by**(*cases sx*) *auto*
next
case (*Return Q p f*)
with ‹S,*slice-kind* $S \vdash (msx,sx) -ax\rightarrow_\tau (msx',sx')$›
‹$\forall\, mx \in set\ ms.\ \exists\, mx'.$ *call-of-return-node* $mx\ mx' \wedge mx' \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$›
show *False* **by**(*fastforce elim*!:*silent-move.cases simp*:*intra-kind-def*)
qed
qed

lemma *silent-moves-slice-keeps-state*:

assumes S,*slice-kind* $S \vdash (m\#ms,s) =as\Rightarrow_\tau (m'\#ms',s')$
and $\forall\, mx \in set\ ms.\ \exists\, mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
shows $s = s'$
proof −
from *assms* **have** $\forall\, a \in set\ as.\ intra\text{-}kind\ (kind\ a)$
by(*rule silent-moves-slice-intra-path*)
with *assms* **show** *?thesis*
proof(*induct S slice-kind S m#ms s as m'#ms' s'*
arbitrary:*m rule*:*silent-moves.induct*)
case (*silent-moves-Nil sx* n_c) **thus** *?case* **by** *simp*
next
case (*silent-moves-Cons S sx a msx' sx' as s''*)
note *IH* = ‹$\bigwedge m.$
$[\![msx' = m\ \#\ ms;$
$\forall\, mx{\in}set\ ms.\ \exists\, mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge mx' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG};$
$\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a)]\!] \Longrightarrow sx' = s''$›
note *callstack* = ‹$\forall\, mx{\in}set\ ms.\ \exists\, mx'.\ call\text{-}of\text{-}return\text{-}node\ mx\ mx' \wedge$
$mx' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
from ‹$\forall\, a{\in}set\ (a\ \#\ as).\ intra\text{-}kind\ (kind\ a)$› **have** *intra-kind* (*kind a*)
and $\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a)$ **by** *simp-all*
from ‹S,*slice-kind S* $\vdash (m\ \#\ ms,sx) -a\rightarrow_\tau (msx',sx')$› ‹*intra-kind* (*kind a*)›
callstack
have [*simp*]:$msx' = targetnode\ a\#ms$ **and** $sx' = transfer\ (slice\text{-}kind\ S\ a)\ sx$
and *sourcenode a* $\notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **and** *valid-edge a* **and** $sx \neq []$
by(*auto elim*!:*silent-move.cases simp*:*intra-kind-def*)
from *IH*[*OF* ‹$msx' = targetnode\ a\#ms$› *callstack* ‹$\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind$
$a)$›]
have $sx' = s''$ **.**
from ‹*intra-kind* (*kind a*)›
have $sx = sx'$
proof(*cases kind a*)
case (*UpdateEdge f'*)
with ‹*sourcenode a* $\notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
have *slice-kind S a* = $\Uparrow id$ **by**(*rule slice-kind-Upd*)
with ‹$sx' = transfer\ (slice\text{-}kind\ S\ a)\ sx$› ‹$sx \neq []$›
show *?thesis* **by**(*cases sx*) *auto*
next
case (*PredicateEdge Q*)
with ‹*sourcenode a* $\notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$› ‹*valid-edge a*›
obtain Q' **where** *slice-kind S a* = $(Q')_\surd$
by −(*erule kind-Predicate-notin-slice-slice-kind-Predicate*)
with ‹$sx' = transfer\ (slice\text{-}kind\ S\ a)\ sx$› ‹$sx \neq []$›
show *?thesis* **by**(*cases sx*) *auto*
qed (*auto simp*:*intra-kind-def*)
with ‹$sx' = s''$› **show** *?case* **by** *simp*
qed
qed

1.14.2 Definition of *slice-edges*

definition *slice-edge* :: *'node SDG-node set* ⇒ *'edge list* ⇒ *'edge* ⇒ *bool*
where *slice-edge S cs a* ≡ (∀ *c* ∈ *set cs*. *sourcenode c* ∈ ⌊*HRB-slice S*⌋$_{CFG}$) ∧
(*case* (*kind a*) *of Q*↩$_{pf}$ ⇒ *True* | - ⇒ *sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$)

lemma *silent-move-no-slice-edge*:
⟦*S*,*f* ⊢ (*ms*,*s*) −*a*→$_{\tau}$ (*ms'*,*s'*); *tl ms* = *targetnodes rs*; *length rs* = *length cs*;
∀ *i*<*length cs*. *call-of-return-node* (*tl ms*!*i*) (*sourcenode* (*cs*!*i*))⟧
⟹ ¬ *slice-edge S cs a*
proof(*induct rule*:*silent-move.induct*)
case (*silent-move-intra f a s s' ms S ms'*)
note *disj* = ‹(∃ *m*∈*set* (*tl ms*). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$)
∨ *hd ms* ∉ ⌊*HRB-slice S*⌋$_{CFG}$›
from ‹*pred* (*f a*) *s*› ‹*length ms* = *length s*› **obtain** *x xs* **where** *ms* = *x*#*xs*
by(*cases ms*) *auto*
from ‹*length rs* = *length cs*› ‹*tl ms* = *targetnodes rs*›
have *length* (*tl ms*) = *length cs* **by**(*simp add*:*targetnodes-def*)
from *disj* **show** *?case*
proof
assume ∃ *m*∈*set* (*tl ms*). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
with ‹∀ *i*<*length cs*. *call-of-return-node* (*tl ms* ! *i*) (*sourcenode* (*cs* ! *i*))›
‹*length* (*tl ms*) = *length cs*›
have ∃ *c* ∈ *set cs*. *sourcenode c* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
apply(*auto simp*:*in-set-conv-nth*)
by(*erule-tac x*=*i* **in** *allE*) *auto*
thus *?thesis* **by**(*auto simp*:*slice-edge-def*)
next
assume *hd ms* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
with ‹*hd ms* = *sourcenode a*› ‹*intra-kind* (*kind a*)›
show *?case* **by**(*auto simp*:*slice-edge-def simp*:*intra-kind-def*)
qed
next
case (*silent-move-call f a s s' Q r p fs a' ms S ms'*)
note *disj* = ‹(∃ *m*∈*set* (*tl ms*). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$)
∨ *hd ms* ∉ ⌊*HRB-slice S*⌋$_{CFG}$›
from ‹*pred* (*f a*) *s*› ‹*length ms* = *length s*› **obtain** *x xs* **where** *ms* = *x*#*xs*
by(*cases ms*) *auto*
from ‹*length rs* = *length cs*› ‹*tl ms* = *targetnodes rs*›
have *length* (*tl ms*) = *length cs* **by**(*simp add*:*targetnodes-def*)
from *disj* **show** *?case*
proof
assume ∃ *m*∈*set* (*tl ms*). ∃ *m'*. *call-of-return-node m m'* ∧ *m'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
with ‹∀ *i*<*length cs*. *call-of-return-node* (*tl ms* ! *i*) (*sourcenode* (*cs* ! *i*))›
‹*length* (*tl ms*) = *length cs*›

have $\exists\, c \in$ *set cs. sourcenode c* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$
apply(*auto simp*:*in-set-conv-nth*)
by(*erule-tac x*=*i* **in** *allE*) *auto*
thus *?thesis* **by**(*auto simp*:*slice-edge-def*)
next
assume *hd ms* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$
with ‹*hd ms = sourcenode a*› ‹*kind a = Q*:$r \hookrightarrow_p$*fs*›
show *?case* **by**(*auto simp*:*slice-edge-def*)
qed
next
case (*silent-move-return f a s s' Q p f' ms S ms'*)
from ‹*pred* (*f a*) *s*› ‹*length ms = length s*› **obtain** *x xs* **where** *ms = x*#*xs*
by(*cases ms*) *auto*
from ‹*length rs = length cs*› ‹*tl ms = targetnodes rs*›
have *length* (*tl ms*) = *length cs* **by**(*simp add*:*targetnodes-def*)
from ‹∀ *i*<*length cs. call-of-return-node* (*tl ms* ! *i*) (*sourcenode* (*cs* ! *i*))›
‹∃ *m*∈*set* (*tl ms*). ∃ *m'. call-of-return-node m m'* ∧ *m'* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
‹*length* (*tl ms*) = *length cs*›
have $\exists\, c \in$ *set cs. sourcenode c* $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$
apply(*auto simp*:*in-set-conv-nth*)
by(*erule-tac x*=*i* **in** *allE*) *auto*
thus *?case* **by**(*auto simp*:*slice-edge-def*)
qed

lemma *observable-move-slice-edge*:
⟦*S*,*f* ⊢ (*ms*,*s*) −*a*→ (*ms'*,*s'*); *tl ms = targetnodes rs*; *length rs = length cs*;
∀ *i*<*length cs. call-of-return-node* (*tl ms*!*i*) (*sourcenode* (*cs*!*i*))⟧
⟹ *slice-edge S cs a*
proof(*induct rule*:*observable-move.induct*)
case (*observable-move-intra f a s s' ms S ms'*)
from ‹*pred* (*f a*) *s*› ‹*length ms = length s*› **obtain** *x xs* **where** *ms = x*#*xs*
by(*cases ms*) *auto*
from ‹*length rs = length cs*› ‹*tl ms = targetnodes rs*›
have *length* (*tl ms*) = *length cs* **by**(*simp add*:*targetnodes-def*)
with ‹∀ *m*∈*set* (*tl ms*). ∃ *m'. call-of-return-node m m'* ∧ *m'* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›
‹∀ *i*<*length cs. call-of-return-node* (*tl ms*!*i*) (*sourcenode* (*cs*!*i*))›
have ∀ *c* ∈ *set cs. sourcenode c* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
apply(*auto simp*:*in-set-conv-nth*)
by(*erule-tac x*=*i* **in** *allE*) *auto*
with ‹*hd ms = sourcenode a*› ‹*hd ms* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*intra-kind* (*kind a*)›
show *?case* **by**(*auto simp*:*slice-edge-def simp*:*intra-kind-def*)
next
case (*observable-move-call f a s s' Q r p fs a' ms S ms'*)
from ‹*pred* (*f a*) *s*› ‹*length ms = length s*› **obtain** *x xs* **where** *ms = x*#*xs*
by(*cases ms*) *auto*
from ‹*length rs = length cs*› ‹*tl ms = targetnodes rs*›
have *length* (*tl ms*) = *length cs* **by**(*simp add*:*targetnodes-def*)
with ‹∀ *m*∈*set* (*tl ms*). ∃ *m'. call-of-return-node m m'* ∧ *m'* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$›

```
    ⟨∀ i<length cs. call-of-return-node (tl ms!i) (sourcenode (cs!i))⟩
  have ∀ c ∈ set cs. sourcenode c ∈ ⌊HRB-slice S⌋CFG
    apply(auto simp:in-set-conv-nth)
    by(erule-tac x=i in allE) auto
  with ⟨hd ms = sourcenode a⟩ ⟨hd ms ∈ ⌊HRB-slice S⌋CFG⟩ ⟨kind a = Q:r↪pfs⟩
  show ?case by(auto simp:slice-edge-def)
next
  case (observable-move-return f a s s' Q p f' ms S ms')
  from ⟨pred (f a) s⟩ ⟨length ms = length s⟩ obtain x xs where ms = x#xs
    by(cases ms) auto
  from ⟨length rs = length cs⟩ ⟨tl ms = targetnodes rs⟩
  have length (tl ms) = length cs by(simp add:targetnodes-def)
  with ⟨∀ m∈set (tl ms). ∃ m'. call-of-return-node m m' ∧ m' ∈ ⌊HRB-slice S⌋CFG⟩
    ⟨∀ i<length cs. call-of-return-node (tl ms!i) (sourcenode (cs!i))⟩
  have ∀ c ∈ set cs. sourcenode c ∈ ⌊HRB-slice S⌋CFG
    apply(auto simp:in-set-conv-nth)
    by(erule-tac x=i in allE) auto
  with ⟨kind a = Q↩pf'⟩ show ?case by(auto simp:slice-edge-def)
qed



function slice-edges :: 'node SDG-node set ⇒ 'edge list ⇒ 'edge list ⇒ 'edge list
where slice-edges S cs [] = []
  | slice-edge S cs a ⟹
     slice-edges S cs (a#as) = a#slice-edges S (upd-cs cs [a]) as
  | ¬ slice-edge S cs a ⟹
     slice-edges S cs (a#as) = slice-edges S (upd-cs cs [a]) as
by(atomize-elim)(auto,case-tac b,auto)
termination by(lexicographic-order)



lemma slice-edges-Append:
  ⟦slice-edges S cs as = as'; slice-edges S (upd-cs cs as) asx = asx'⟧
  ⟹ slice-edges S cs (as@asx) = as'@asx'
proof(induct as arbitrary:cs as')
  case Nil thus ?case by simp
next
  case (Cons x xs)
  note IH = ⟨⋀cs as'. ⟦slice-edges S cs xs = as';
    slice-edges S (upd-cs cs xs) asx = asx'⟧
    ⟹ slice-edges S cs (xs @ asx) = as' @ asx'⟩
  from ⟨slice-edges S (upd-cs cs (x # xs)) asx = asx'⟩
  have slice-edges S (upd-cs (upd-cs cs [x]) xs) asx = asx'
    by(cases kind x)(auto,cases cs,auto)
  show ?case
  proof(cases slice-edge S cs x)
    case True
    with ⟨slice-edges S cs (x # xs) = as'⟩
```

have *x#slice-edges S (upd-cs cs [x]) xs = as'* **by** *simp*
then obtain *xs'* **where** *as' = x#xs'*
and *slice-edges S (upd-cs cs [x]) xs = xs'* **by**(*cases as'*) *auto*
from *IH*[*OF* ‹*slice-edges S (upd-cs cs [x]) xs = xs'*›
‹*slice-edges S (upd-cs (upd-cs cs [x]) xs) asx = asx'*›]
have *slice-edges S (upd-cs cs [x]) (xs @ asx) = xs' @ asx'* .
with *True* ‹*as' = x#xs'*› **show** *?thesis* **by** *simp*
next
case *False*
with ‹*slice-edges S cs (x # xs) = as'*›
have *slice-edges S (upd-cs cs [x]) xs = as'* **by** *simp*
from *IH*[*OF this* ‹*slice-edges S (upd-cs (upd-cs cs [x]) xs) asx = asx'*›]
have *slice-edges S (upd-cs cs [x]) (xs @ asx) = as' @ asx'* .
with *False* **show** *?thesis* **by** *simp*
qed
qed

lemma *slice-edges-Nil-split*:
slice-edges S cs (as@as') = []
⟹ *slice-edges S cs as = [] ∧ slice-edges S (upd-cs cs as) as' = []*
apply(*induct as arbitrary*:*cs*)
apply *clarsimp*
apply(*case-tac slice-edge S cs a*) **apply** *auto*
apply(*case-tac kind a*) **apply** *auto*
apply(*case-tac cs*) **apply** *auto*
done

lemma *slice-intra-edges-no-nodes-in-slice*:
⟦*slice-edges S cs as = []*; ∀ *a* ∈ *set as*. *intra-kind* (*kind a*);
∀ *c* ∈ *set cs*. *sourcenode c* ∈ ⌊*HRB-slice S*⌋$_{CFG}$⟧
⟹ ∀ *nx* ∈ *set*(*sourcenodes as*). *nx* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
proof(*induct as*)
case *Nil* **thus** *?case* **by**(*fastforce simp*:*sourcenodes-def*)
next
case (*Cons a' as'*)
note *IH* = ‹⟦*slice-edges S cs as' = []*; ∀ *a*∈*set as'*. *intra-kind* (*kind a*);
∀ *c*∈*set cs*. *sourcenode c* ∈ ⌊*HRB-slice S*⌋$_{CFG}$⟧
⟹ ∀ *nx*∈*set* (*sourcenodes as'*). *nx* ∉ ⌊*HRB-slice S*⌋$_{CFG}$›
from ‹∀ *a*∈*set* (*a' # as'*). *intra-kind* (*kind a*)›
have *intra-kind* (*kind a'*) **and** ∀ *a*∈*set as'*. *intra-kind* (*kind a*) **by** *simp-all*
from ‹*slice-edges S cs* (*a' # as'*) = []› ‹*intra-kind* (*kind a'*)›
‹∀ *c*∈*set cs*. *sourcenode c* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
have *sourcenode a'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$ **and** *slice-edges S cs as' = []*
by(*cases slice-edge S cs a'*,*auto simp*:*intra-kind-def slice-edge-def*)+
from *IH*[*OF* ‹*slice-edges S cs as' = []*› ‹∀ *a*∈*set as'*. *intra-kind* (*kind a*)›
‹∀ *c*∈*set cs*. *sourcenode c* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›]
have ∀ *nx*∈*set* (*sourcenodes as'*). *nx* ∉ ⌊*HRB-slice S*⌋$_{CFG}$.

with ‹*sourcenode a'* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› **show** *?case* **by**(*simp add*:*sourcenodes-def*)
qed

lemma *silent-moves-no-slice-edges*:
⟦*S*,*f* ⊢ (*ms*,*s*) =*as*⇒$_\tau$ (*ms'*,*s'*); *tl ms* = *targetnodes rs*; *length rs* = *length cs*;
∀ *i*<*length cs*. *call-of-return-node* (*tl ms*!*i*) (*sourcenode* (*cs*!*i*))⟧
⟹ *slice-edges S cs as* = [] ∧ (∃ *rs'*. *tl ms'* = *targetnodes rs'* ∧
length rs' = *length* (*upd-cs cs as*) ∧ (∀ *i*<*length* (*upd-cs cs as*).
call-of-return-node (*tl ms'*!*i*) (*sourcenode* ((*upd-cs cs as*)!*i*))))
proof(*induct arbitrary*:*rs cs rule*:*silent-moves.induct*)
case (*silent-moves-Cons S f ms s a ms' s' as ms'' s''*)
from ‹*S*,*f* ⊢ (*ms*,*s*) −*a*→$_\tau$ (*ms'*,*s'*)› ‹*tl ms* = *targetnodes rs*› ‹*length rs* = *length cs*›
‹∀ *i*<*length cs*. *call-of-return-node* (*tl ms* ! *i*) (*sourcenode* (*cs* ! *i*))›
have ¬ *slice-edge S cs a* **by**(*rule silent-move-no-slice-edge*)
with *silent-moves-Cons* **show** *?case*
proof(*induct rule*:*silent-move.induct*)
case (*silent-move-intra f a s s' ms S ms'*)
note *IH* = ‹⋀*rs cs*. ⟦*tl ms'* = *targetnodes rs*; *length rs* = *length cs*;
∀ *i*<*length cs*. *call-of-return-node* (*tl ms'* ! *i*) (*sourcenode* (*cs* ! *i*))⟧
⟹ *slice-edges S cs as* = [] ∧ (∃ *rs'*. *tl ms''* = *targetnodes rs'* ∧
length rs' = *length* (*upd-cs cs as*) ∧ (∀ *i*<*length* (*upd-cs cs as*).
call-of-return-node (*tl ms''* ! *i*) (*sourcenode* (*upd-cs cs as* ! *i*))))›
from ‹*ms'* = *targetnode a* # *tl ms*› ‹*tl ms* = *targetnodes rs*›
have *tl ms'* = *targetnodes rs* **by** *simp*
from ‹*ms'* = *targetnode a* # *tl ms*› ‹*tl ms* = *targetnodes rs*›
‹∀ *i*<*length cs*. *call-of-return-node* (*tl ms* ! *i*) (*sourcenode* (*cs* ! *i*))›
have ∀ *i*<*length cs*. *call-of-return-node* (*tl ms'* ! *i*) (*sourcenode* (*cs* ! *i*))
by *simp*
from *IH*[*OF* ‹*tl ms'* = *targetnodes rs*› ‹*length rs* = *length cs*› *this*]
have *slice-edges S cs as* = []
and ∃ *rs'*. *tl ms''* = *targetnodes rs'* ∧ *length rs'* = *length* (*upd-cs cs as*) ∧
(∀ *i*<*length* (*upd-cs cs as*).
call-of-return-node (*tl ms''* ! *i*) (*sourcenode* (*upd-cs cs as* ! *i*))) **by** *simp-all*
with ‹*intra-kind* (*kind a*)› ‹¬ *slice-edge S cs a*›
show *?case* **by**(*fastforce simp*:*intra-kind-def*)
next
case (*silent-move-call f a s s' Q r p fs a' ms S ms'*)
note *IH* = ‹⋀*rs cs*. ⟦*tl ms'* = *targetnodes rs*; *length rs* = *length cs*;
∀ *i*<*length cs*. *call-of-return-node* (*tl ms'* ! *i*) (*sourcenode* (*cs* ! *i*))⟧
⟹ *slice-edges S cs as* = [] ∧ (∃ *rs'*. *tl ms''* = *targetnodes rs'* ∧
length rs' = *length* (*upd-cs cs as*) ∧ (∀ *i*<*length* (*upd-cs cs as*).
call-of-return-node (*tl ms''* ! *i*) (*sourcenode* (*upd-cs cs as* ! *i*))))›
from ‹*tl ms* = *targetnodes rs*› ‹*ms'* = *targetnode a* # *targetnode a'* # *tl ms*›
have *tl ms'* = *targetnodes* (*a'*#*rs*) **by**(*simp add*:*targetnodes-def*)
from ‹*length rs* = *length cs*› **have** *length* (*a'*#*rs*) = *length* (*a*#*cs*) **by** *simp*
from ‹*valid-edge a'*› ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
have *return-node* (*targetnode a'*) **by**(*fastforce simp*:*return-node-def*)

with ‹*valid-edge a*› ‹*valid-edge a'*› ‹*a'* ∈ *get-return-edges a*›
have *call-of-return-node* (*targetnode a'*) (*sourcenode a*)
by(*simp add*:*call-of-return-node-def*) *blast*
with ‹∀ *i*<*length cs*. *call-of-return-node* (*tl ms* ! *i*) (*sourcenode* (*cs* ! *i*))›
‹*ms'* = *targetnode a* # *targetnode a'* # *tl ms*›
have ∀ *i*<*length* (*a*#*cs*).
call-of-return-node (*tl ms'* ! *i*) (*sourcenode* ((*a*#*cs*) ! *i*))
by *auto* (*case-tac i*,*auto*)
from *IH*[*OF* ‹*tl ms'* = *targetnodes* (*a'*#*rs*)› ‹*length* (*a'*#*rs*) = *length* (*a*#*cs*)›
this]
have *slice-edges S* (*a* # *cs*) *as* = []
and ∃ *rs'*. *tl ms''* = *targetnodes rs'* ∧
length rs' = *length* (*upd-cs* (*a* # *cs*) *as*) ∧
(∀ *i*<*length* (*upd-cs* (*a* # *cs*) *as*).
call-of-return-node (*tl ms''* ! *i*) (*sourcenode* (*upd-cs* (*a* # *cs*) *as* ! *i*)))
by *simp-all*
with ‹¬ *slice-edge S cs a*› ‹*kind a* = *Q*:*r*↪*pfs*› **show** *?case* **by** *simp*
next
case (*silent-move-return f a s s' Q p f' ms S ms'*)
note *IH* = ‹⋀*rs cs*. ⟦*tl ms'* = *targetnodes rs*; *length rs* = *length cs*;
∀ *i*<*length cs*. *call-of-return-node* (*tl ms'* ! *i*) (*sourcenode* (*cs* ! *i*))⟧
⟹ *slice-edges S cs as* = [] ∧ (∃ *rs'*. *tl ms''* = *targetnodes rs'* ∧
length rs' = *length* (*upd-cs cs as*) ∧ (∀ *i*<*length* (*upd-cs cs as*).
call-of-return-node (*tl ms''* ! *i*) (*sourcenode* (*upd-cs cs as* ! *i*))))›
from ‹*length s* = *Suc* (*length s'*)› ‹*s'* ≠ []› ‹*length ms* = *length s*› ‹*ms'* = *tl ms*›
obtain *x xs* **where** [*simp*]:*ms'* = *x*#*xs* **by**(*cases ms*)(*auto*,*case-tac ms'*,*auto*)
from ‹*ms'* = *tl ms*› ‹*tl ms* = *targetnodes rs*› **obtain** *r' rs'* **where** *rs* = *r'*#*rs'*
and *x* = *targetnode r'* **and** *tl ms'* = *targetnodes rs'*
by(*cases rs*)(*auto simp*:*targetnodes-def*)
from ‹*length rs* = *length cs*› ‹*rs* = *r'*#*rs'*› **obtain** *c' cs'* **where** *cs* = *c'*#*cs'*
and *length rs'* = *length cs'* **by**(*cases cs*) *auto*
from ‹∀ *i*<*length cs*. *call-of-return-node* (*tl ms* ! *i*) (*sourcenode* (*cs* ! *i*))›
‹*cs* = *c'*#*cs'*› ‹*ms'* = *tl ms*›
have ∀ *i*<*length cs'*. *call-of-return-node* (*tl ms'* ! *i*) (*sourcenode* (*cs'* ! *i*))
by *auto*(*erule-tac x*=*Suc i* **in** *allE*,*cases tl ms*,*auto*)
from *IH*[*OF* ‹*tl ms'* = *targetnodes rs'*› ‹*length rs'* = *length cs'*› *this*]
have *slice-edges S cs' as* = [] **and** ∃ *rs'*. *tl ms''* = *targetnodes rs'* ∧
length rs' = *length* (*upd-cs cs' as*) ∧ (∀ *i*<*length* (*upd-cs cs' as*).
call-of-return-node (*tl ms''* ! *i*) (*sourcenode* (*upd-cs cs' as* ! *i*)))
by *simp-all*
with ‹¬ *slice-edge S cs a*› ‹*kind a* = *Q*↩*pf'*› ‹*cs* = *c'*#*cs'*›
show *?case* **by** *simp*
qed
qed *fastforce*

lemma *observable-moves-singular-slice-edge*:
⟦*S*,*f* ⊢ (*ms*,*s*) =*as*⇒ (*ms'*,*s'*); *tl ms* = *targetnodes rs*; *length rs* = *length cs*;

```
    ∀ i<length cs. call-of-return-node (tl ms!i) (sourcenode (cs!i))⟧
  ⟹ slice-edges S cs as = [last as]
proof(induct rule:observable-moves.induct)
  case (observable-moves-snoc S f ms s as ms' s' a ms'' s'')
  from ‹S,f ⊢ (ms,s) =as⇒τ (ms',s')› ‹tl ms = targetnodes rs› ‹length rs = length
cs›
    ‹∀ i<length cs. call-of-return-node (tl ms ! i) (sourcenode (cs ! i))›
  obtain rs' where slice-edges S cs as = []
    and tl ms' = targetnodes rs' and length rs' = length (upd-cs cs as)
    and ∀ i<length (upd-cs cs as).
    call-of-return-node (tl ms'!i) (sourcenode ((upd-cs cs as)!i))
    by(fastforce dest!:silent-moves-no-slice-edges)
  from ‹S,f ⊢ (ms',s') −a→ (ms'',s'')› this
  have slice-edge S (upd-cs cs as) a by −(rule observable-move-slice-edge)
  with ‹slice-edges S cs as = []› have slice-edges S cs (as @ [a]) = []@[a]
    by(fastforce intro:slice-edges-Append)
  thus ?case by simp
qed


lemma silent-moves-nonempty-nodestack-False:
  assumes S,kind ⊢ ([m],[cf]) =as⇒τ (m'#ms',s') and valid-node m
  and ms' ≠ [] and CFG-node m' ∈ sum-SDG-slice1 nx and nx ∈ S
  shows False
proof −
  from assms(1−4) have slice-edges S [] as ≠ []
  proof(induct ms' arbitrary:as m' s')
    case (Cons mx msx)
    note IH = ‹⋀as m' s'. ⟦S,kind ⊢ ([m],[cf]) =as⇒τ (m' # msx,s'); valid-node
m;
      msx ≠ []; CFG-node m' ∈ sum-SDG-slice1 nx⟧
      ⟹ slice-edges S [] as ≠ []›
    from ‹S,kind ⊢ ([m],[cf]) =as⇒τ (m' # mx # msx,s')› ‹valid-node m›
      ‹CFG-node m' ∈ sum-SDG-slice1 nx›
    obtain as' a as'' where as = as'@a#as'' and ∃ Q r p fs. kind a = Q:r↪pfs
      and call-of-return-node mx (sourcenode a)
      and CFG-node (sourcenode a) ∈ sum-SDG-slice1 nx
      and targetnode a −as''→sl* m'
      by(fastforce elim!:silent-moves-called-node-in-slice1-hd-nodestack-in-slice1
      [of - - - - - - - - - [] []] simp:targetnodes-def valid-return-list-def)
    from ‹S,kind ⊢ ([m],[cf]) =as⇒τ (m' # mx # msx,s')› ‹as = as'@a#as''›
    obtain xs x where S,kind ⊢ ([m],[cf]) =as'⇒τ (xs,x)
      and S,kind ⊢ (xs,x) =a#as''⇒τ (m' # mx # msx,s')
      by(fastforce elim:silent-moves-split)
    from ‹S,kind ⊢ (xs,x) =a#as''⇒τ (m' # mx # msx,s')›
    obtain ys y where S,kind ⊢ (xs,x) −a→τ (ys,y)
      and S,kind ⊢ (ys,y) =as''⇒τ (m' # mx # msx,s')
      by(fastforce elim:silent-moves.cases)
    from ‹S,kind ⊢ (xs,x) −a→τ (ys,y)› ‹∃ Q r p fs. kind a = Q:r↪pfs›
```

```
    obtain xs' a' where xs = sourcenode a#xs'
      and ys = targetnode a#targetnode a'#xs'
      apply − apply(erule silent-move.cases) apply(auto simp:intra-kind-def)
      by(cases xs,auto)+
    from ‹S,kind ⊢ (ys,y) =as''⇒τ (m' # mx # msx,s')›
      ‹ys = targetnode a#targetnode a'#xs'› ‹targetnode a −as''→sl* m'›
    have mx = targetnode a' and xs' = msx
      by(auto dest:silent-moves-same-level-path)
    with ‹xs = sourcenode a#xs'› ‹S,kind ⊢ ([m],[cf]) =as'⇒τ (xs,x)›
    have S,kind ⊢ ([m],[cf]) =as'⇒τ (sourcenode a#msx,x) by simp
    show ?case
    proof(cases msx = [])
      case True
      from ‹S,kind ⊢ ([m],[cf]) =as'⇒τ (sourcenode a#msx,x)›
      obtain rs' where msx = targetnodes rs' ∧ length rs' = length (upd-cs [] as')
        by(fastforce dest!:silent-moves-no-slice-edges[where cs=[] and rs=[]]
                   simp:targetnodes-def)
      with True have upd-cs [] as' = []  by(cases rs')(auto simp:targetnodes-def)
      with ‹CFG-node (sourcenode a) ∈ sum-SDG-slice1 nx› ‹nx ∈ S›
      have slice-edge S (upd-cs [] as') a
        by(cases kind a,auto intro:combSlice-refl
          simp:slice-edge-def SDG-to-CFG-set-def HRB-slice-def)
      hence slice-edges S (upd-cs [] as') (a#as'') ≠ [] by simp
      with ‹as = as'@a#as''› show ?thesis by(fastforce dest:slice-edges-Nil-split)
    next
      case False
      from IH[OF ‹S,kind ⊢ ([m],[cf]) =as'⇒τ (sourcenode a#msx,x)›
        ‹valid-node m› this ‹CFG-node (sourcenode a) ∈ sum-SDG-slice1 nx›]
      have slice-edges S [] as' ≠ [] .
      with ‹as = as'@a#as''› show ?thesis by(fastforce dest:slice-edges-Nil-split)
    qed
  qed simp
  moreover
  from ‹S,kind ⊢ ([m],[cf]) =as⇒τ (m'#ms',s')› have slice-edges S [] as = []
    by(fastforce dest!:silent-moves-no-slice-edges[where cs=[] and rs=[]]
                 simp:targetnodes-def)
  ultimately show False by simp
qed



lemma transfers-intra-slice-kinds-slice-edges:
  ⟦∀ a ∈ set as. intra-kind (kind a); ∀ c ∈ set cs. sourcenode c ∈ ⌊HRB-slice S⌋CFG⟧
  ⟹ transfers (slice-kinds S (slice-edges S cs as)) s =
  transfers (slice-kinds S as) s
proof(induct as arbitrary:s)
  case Nil thus ?case by(simp add:slice-kinds-def)
next
  case (Cons a' as')
```

```
  note IH = ‹⋀s. ⟦∀ a∈set as'. intra-kind (kind a);
    ∀ c∈set cs. sourcenode c ∈ ⌊HRB-slice S⌋_CFG⟧ ⟹
    transfers (slice-kinds S (slice-edges S cs as')) s =
    transfers (slice-kinds S as') s›
  from ‹∀ a∈set (a' # as'). intra-kind (kind a)›
  have intra-kind (kind a') and ∀ a∈set as'. intra-kind (kind a)
    by simp-all
  show ?case
  proof(cases slice-edge S cs a')
    case True
    with ‹intra-kind (kind a')›
    have eq:transfers (slice-kinds S (slice-edges S cs (a'#as'))) s
          = transfers (slice-kinds S (slice-edges S cs as'))
              (transfer (slice-kind S a') s)
      by(cases kind a')(auto simp:slice-kinds-def intra-kind-def)
    have transfers (slice-kinds S (a'#as')) s
       = transfers (slice-kinds S as') (transfer (slice-kind S a') s)
      by(simp add:slice-kinds-def)
    with eq IH[OF ‹∀ a∈set as'. intra-kind (kind a)›
      ‹∀ c∈set cs. sourcenode c ∈ ⌊HRB-slice S⌋_CFG›,
      of transfer (slice-kind S a') s]
    show ?thesis by simp
  next
    case False
    with ‹intra-kind (kind a')›
    have eq:transfers (slice-kinds S (slice-edges S cs (a'#as'))) s
          = transfers (slice-kinds S (slice-edges S cs as')) s
      by(cases kind a')(auto simp:slice-kinds-def intra-kind-def)
   from False ‹intra-kind (kind a')› ‹∀ c∈set cs. sourcenode c ∈ ⌊HRB-slice S⌋_CFG›
    have sourcenode a' ∉ ⌊HRB-slice S⌋_CFG
      by(fastforce simp:slice-edge-def intra-kind-def)
    with ‹intra-kind (kind a')› have transfer (slice-kind S a') s = s
      by(cases s)(auto,cases kind a',
        auto simp:slice-kind-def Let-def intra-kind-def)
    hence transfers (slice-kinds S (a'#as')) s
        = transfers (slice-kinds S as') s
      by(simp add:slice-kinds-def)
    with eq IH[OF ‹∀ a∈set as'. intra-kind (kind a)›
      ‹∀ c∈set cs. sourcenode c ∈ ⌊HRB-slice S⌋_CFG›,of s] show ?thesis by simp
  qed
qed


lemma exists-sliced-intra-path-preds:
  assumes m -as→ι* m' and slice-edges S cs as = []
 and m' ∈ ⌊HRB-slice S⌋_CFG and ∀ c ∈ set cs. sourcenode c ∈ ⌊HRB-slice S⌋_CFG
  obtains as' where m -as'→ι* m' and preds (slice-kinds S as') (cf#cfs)
  and slice-edges S cs as' = []
```

proof(*atomize-elim*)
from ‹$m\ -as\rightarrow_\iota*\ m'$› **have** $m\ -as\rightarrow*\ m'$ **and** $\forall\, a \in set\ as.\ intra\text{-}kind(kind\ a)$
by(*simp-all add*:*intra-path-def*)
from ‹*slice-edges S cs as* = []› ‹$\forall\, a \in set\ as.\ intra\text{-}kind(kind\ a)$›
‹$\forall\, c \in set\ cs.\ sourcenode\ c \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$›
have $\forall\, nx \in set(sourcenodes\ as).\ nx \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*rule slice-intra-edges-no-nodes-in-slice*)
with ‹$m\ -as\rightarrow_\iota*\ m'$› ‹$m' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$› **have** $m' \in obs\text{-}intra\ m\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
by(*fastforce intro*:*obs-intra-elem*)
hence $obs\text{-}intra\ m\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{m'\}$ **by**(*rule obs-intra-singleton-element*)
from ‹$m\ -as\rightarrow*\ m'$› **have** *valid-node m* **and** *valid-node m'*
by(*fastforce dest*:*path-valid-node*)+
from ‹$m\ -as\rightarrow_\iota*\ m'$› **obtain** *x* **where** *distance m m' x* **and** $x \leq length\ as$
by(*erule every-path-distance*)
from ‹*distance m m' x*› ‹$obs\text{-}intra\ m\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{m'\}$›
show $\exists\, as'.\ m\ -as'\rightarrow_\iota*\ m' \wedge preds\ (slice\text{-}kinds\ S\ as')\ (cf \# cfs) \wedge$
slice-edges S cs as' = []
proof(*induct x arbitrary*:*m rule*:*nat.induct*)
case *zero*
from ‹*distance m m' 0*› **have** $m = m'$
by(*fastforce elim*:*distance.cases simp*:*intra-path-def*)
with ‹*valid-node m'*› **show** *?case*
by(*rule-tac x*=[] **in** *exI*,
auto intro:*empty-path simp*:*slice-kinds-def intra-path-def*)
next
case (*Suc x*)
note *IH* = ‹$\bigwedge m.\ [\![distance\ m\ m'\ x;\ obs\text{-}intra\ m\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{m'\}]\!]$
$\Longrightarrow \exists\, as'.\ m\ -as'\rightarrow_\iota*\ m' \wedge preds\ (slice\text{-}kinds\ S\ as')\ (cf\ \#\ cfs) \wedge$
slice-edges S cs as' = []›
from ‹*distance m m' (Suc x)*› **obtain** *a*
where *valid-edge a* **and** $m = sourcenode\ a$ **and** $intra\text{-}kind(kind\ a)$
and *distance* (*targetnode a*) *m' x*
and *target*:$targetnode\ a = (SOME\ nx.\ \exists\, a'.\ sourcenode\ a = sourcenode\ a' \wedge$
distance (*targetnode a'*) *m' x* $\wedge$
valid-edge a' $\wedge$ $intra\text{-}kind(kind\ a') \wedge targetnode\ a' = nx)$
by(*auto elim*:*distance-successor-distance*)
have $m \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
proof
assume $m \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
from ‹*valid-edge a*› ‹$m = sourcenode\ a$› **have** *valid-node m* **by** *simp*
with ‹$m \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$› **have** $obs\text{-}intra\ m\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{m\}$
by −(*rule n-in-obs-intra*)
with ‹$obs\text{-}intra\ m\ \lfloor HRB\text{-}slice\ S \rfloor_{CFG} = \{m'\}$› **have** $m = m'$ **by** *simp*
with ‹*valid-node m*› **have** $m\ -[]\rightarrow_\iota*\ m'$
by(*fastforce intro*:*empty-path simp*:*intra-path-def*)
with ‹*distance m m' (Suc x)*› **show** *False*
by(*fastforce elim*:*distance.cases*)
qed

from ‹*distance* (*targetnode a*) *m′ x*› ‹*m′* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
obtain *mx* **where** *mx* ∈ *obs-intra* (*targetnode a*) ⌊*HRB-slice S*⌋$_{CFG}$
by(*fastforce elim*:*distance.cases path-ex-obs-intra*)
from ‹*valid-edge a*› ‹*intra-kind*(*kind a*)› ‹*m* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*m* = *sourcenode a*›
have *obs-intra* (*targetnode a*) ⌊*HRB-slice S*⌋$_{CFG}$ ⊆
obs-intra (*sourcenode a*) ⌊*HRB-slice S*⌋$_{CFG}$
by −(*rule edge-obs-intra-subset*,*auto*)
with ‹*mx* ∈ *obs-intra* (*targetnode a*) ⌊*HRB-slice S*⌋$_{CFG}$› ‹*m* = *sourcenode a*›
‹*obs-intra m* ⌊*HRB-slice S*⌋$_{CFG}$ = {*m′*}›
have *m′* ∈ *obs-intra* (*targetnode a*) ⌊*HRB-slice S*⌋$_{CFG}$ **by** *auto*
hence *obs-intra* (*targetnode a*) ⌊*HRB-slice S*⌋$_{CFG}$ = {*m′*}
by(*rule obs-intra-singleton-element*)
from *IH*[*OF* ‹*distance* (*targetnode a*) *m′ x*› *this*]
obtain *as* **where** *targetnode a* −*as*→$_\iota$* *m′* **and** *preds* (*slice-kinds S as*) (*cf*#*cfs*)
and *slice-edges S cs as* = [] **by** *blast*
from ‹*targetnode a* −*as*→$_\iota$* *m′*› ‹*valid-edge a*› ‹*intra-kind*(*kind a*)›
‹*m* = *sourcenode a*›
have *m* −*a*#*as*→$_\iota$* *m′* **by**(*fastforce intro*:*Cons-path simp*:*intra-path-def*)
from ‹∀ *c* ∈ *set cs*. *sourcenode c* ∈ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*m* ∉ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*m* = *sourcenode a*› ‹*intra-kind*(*kind a*)›
have ¬ *slice-edge S cs a* **by**(*fastforce simp*:*slice-edge-def intra-kind-def*)
with ‹*slice-edges S cs as* = []› ‹*intra-kind*(*kind a*)›
have *slice-edges S cs* (*a*#*as*) = [] **by**(*fastforce simp*:*intra-kind-def*)
from ‹*intra-kind*(*kind a*)›
show *?case*
proof(*cases kind a*)
case (*UpdateEdge f*)
with ‹*m* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*m* = *sourcenode a*› **have** *slice-kind S a* = ⇑*id*
by(*fastforce intro*:*slice-kind-Upd*)
hence *transfer* (*slice-kind S a*) (*cf*#*cfs*) = (*cf*#*cfs*)
and *pred* (*slice-kind S a*) (*cf*#*cfs*) **by** *simp-all*
with ‹*preds* (*slice-kinds S as*) (*cf*#*cfs*)›
have *preds* (*slice-kinds S* (*a*#*as*)) (*cf*#*cfs*)
by(*simp add*:*slice-kinds-def*)
with ‹*m* −*a*#*as*→$_\iota$* *m′*› ‹*slice-edges S cs* (*a*#*as*) = []› **show** *?thesis*
by *blast*
next
case (*PredicateEdge Q*)
with ‹*m* ∉ ⌊*HRB-slice S*⌋$_{CFG}$› ‹*m* = *sourcenode a*› ‹*distance m m′* (*Suc x*)›
‹*obs-intra m* ⌊*HRB-slice S*⌋$_{CFG}$ = {*m′*}› ‹*distance* (*targetnode a*) *m′ x*›
target
have *slice-kind S a* = (*λs*. *True*)$_\surd$
by(*fastforce intro*:*slice-kind-Pred-obs-nearer-SOME*)
hence *transfer* (*slice-kind S a*) (*cf*#*cfs*) = (*cf*#*cfs*)
and *pred* (*slice-kind S a*) (*cf*#*cfs*) **by** *simp-all*
with ‹*preds* (*slice-kinds S as*) (*cf*#*cfs*)›
have *preds* (*slice-kinds S* (*a*#*as*)) (*cf*#*cfs*)
by(*simp add*:*slice-kinds-def*)

```
    with ‹m −a#as→ι* m'› ‹slice-edges S cs (a#as) = []› show ?thesis by blast
   qed (auto simp:intra-kind-def)
  qed
qed


lemma slp-to-intra-path-with-slice-edges:
  assumes n −as→sl* n' and slice-edges S cs as = []
  obtains as' where n −as'→ι* n' and slice-edges S cs as' = []
proof(atomize-elim)
  from ‹n −as→sl* n'› have n −as→* n' and same-level-path as
    by(simp-all add:slp-def)
  from ‹same-level-path as› have same-level-path-aux [] as and upd-cs [] as = []
    by(simp-all add:same-level-path-def)
  from ‹n −as→* n'› ‹same-level-path-aux [] as› ‹upd-cs [] as = []›
    ‹slice-edges S cs as = []›
  show ∃as'. n −as'→ι* n' ∧ slice-edges S cs as' = []
  proof(induct as arbitrary:n cs rule:length-induct)
    fix as n cs
    assume IH:∀as''. length as'' < length as ⟶
      (∀n''. n'' −as''→* n' ⟶ same-level-path-aux [] as'' ⟶
          upd-cs [] as'' = [] ⟶ (∀cs'. slice-edges S cs' as'' = [] ⟶
          (∃as'. n'' −as'→ι* n' ∧ slice-edges S cs' as' = [])))
      and n −as→* n' and same-level-path-aux [] as and upd-cs [] as = []
      and slice-edges S cs as = []
    show ∃as'. n −as'→ι* n' ∧ slice-edges S cs as' = []
    proof(cases as)
      case Nil
      with ‹n −as→* n'› show ?thesis by(fastforce simp:intra-path-def)
    next
      case (Cons a' as')
      with ‹n −as→* n'› Cons have n = sourcenode a' and valid-edge a'
        and targetnode a' −as'→* n'
        by(auto intro:path-split-Cons)
      show ?thesis
      proof(cases kind a' rule:edge-kind-cases)
        case Intra
        with Cons ‹same-level-path-aux [] as› have same-level-path-aux [] as'
          by(fastforce simp:intra-kind-def)
        moreover
        from Intra Cons ‹upd-cs [] as = []› have upd-cs [] as' = []
          by(fastforce simp:intra-kind-def)
        moreover
        from ‹slice-edges S cs as = []› Cons Intra
        have slice-edges S cs as' = [] and ¬ slice-edge S cs a'
          by(cases slice-edge S cs a',auto simp:intra-kind-def)+
        ultimately obtain as'' where targetnode a' −as''→ι* n'
          and slice-edges S cs as'' = []
          using IH Cons ‹targetnode a' −as'→* n'›
```

by(*erule-tac x=as′* **in** *allE*) *auto*
from ‹*n* = *sourcenode a′*› ‹*valid-edge a′*› *Intra* ‹*targetnode a′* −*as″*→$_\iota$∗ *n′*›
have *n* −*a′*#*as″*→$_\iota$∗ *n′* **by**(*fastforce intro*:*Cons-path simp*:*intra-path-def*)
moreover
from ‹*slice-edges S cs as″* = []› ‹¬ *slice-edge S cs a′*› *Intra*
have *slice-edges S cs* (*a′*#*as″*) = [] **by**(*auto simp*:*intra-kind-def*)
ultimately show *?thesis* **by** *blast*
next
case (*Call Q r p f*)
with *Cons* ‹*same-level-path-aux* [] *as*›
have *same-level-path-aux* [*a′*] *as′* **by** *simp*
from *Call Cons* ‹*upd-cs* [] *as* = []› **have** *upd-cs* [*a′*] *as′* = []
by *simp*
hence *as′* ≠ [] **by** *fastforce*
with ‹*upd-cs* [*a′*] *as′* = []› **obtain** *xs ys* **where** *as′* = *xs*@*ys* **and** *xs* ≠ []
and *upd-cs* [*a′*] *xs* = [] **and** *upd-cs* [] *ys* = []
and ∀ *xs′ ys′*. *xs* = *xs′*@*ys′* ∧ *ys′* ≠ [] ⟶ *upd-cs* [*a′*] *xs′* ≠ []
by −(*erule upd-cs-empty-split*,*auto*)
from ‹*same-level-path-aux* [*a′*] *as′*› ‹*as′* = *xs*@*ys*› ‹*upd-cs* [*a′*] *xs* = []›
have *same-level-path-aux* [*a′*] *xs* **and** *same-level-path-aux* [] *ys*
by(*rule slpa-split*)+
with ‹*upd-cs* [*a′*] *xs* = []› **have** *upd-cs* ([*a′*]@*cs*) *xs* = []@*cs*
by(*fastforce intro*:*same-level-path-upd-cs-callstack-Append*)
from ‹*slice-edges S cs as* = []› *Cons Call*
have *slice-edges S* (*a′*#*cs*) *as′* = [] **and** ¬ *slice-edge S cs a′*
by(*cases slice-edge S cs a′*,*auto*)+
from ‹*slice-edges S* (*a′*#*cs*) *as′* = []› ‹*as′* = *xs*@*ys*›
‹*upd-cs* ([*a′*]@*cs*) *xs* = []@*cs*›
have *slice-edges S cs ys* = []
by(*fastforce dest*:*slice-edges-Nil-split*)
from ‹*same-level-path-aux* [*a′*] *xs*› ‹*upd-cs* [*a′*] *xs* = []›
‹∀ *xs′ ys′*. *xs* = *xs′*@*ys′* ∧ *ys′* ≠ [] ⟶ *upd-cs* [*a′*] *xs′* ≠ []›
have *last xs* ∈ *get-return-edges* (*last* [*a′*])
by(*fastforce intro*!:*slpa-get-return-edges*)
with ‹*valid-edge a′*› *Call*
obtain *a* **where** *valid-edge a* **and** *sourcenode a* = *sourcenode a′*
and *targetnode a* = *targetnode* (*last xs*) **and** *kind a* = (λ*cf*. *False*)$_\surd$
by −(*drule call-return-node-edge*,*auto*)
from ‹*targetnode a* = *targetnode* (*last xs*)› ‹*xs* ≠ []›
have *targetnode a* = *targetnode* (*last* (*a′*#*xs*)) **by** *simp*
from ‹*as′* = *xs*@*ys*› ‹*xs* ≠ []› *Cons* **have** *length ys* < *length as* **by** *simp*
from ‹*targetnode a′* −*as′*→∗ *n′*› ‹*as′* = *xs*@*ys*› ‹*xs* ≠ []›
have *targetnode* (*last* (*a′*#*xs*)) −*ys*→∗ *n′*
by(*cases xs rule*:*rev-cases*,*auto dest*:*path-split*)
with *IH* ‹*length ys* < *length as*› ‹*same-level-path-aux* [] *ys*›
‹*upd-cs* [] *ys* = []› ‹*slice-edges S cs ys* = []›
obtain *as″* **where** *targetnode* (*last* (*a′*#*xs*)) −*as″*→$_\iota$∗ *n′*
and *slice-edges S cs as″* = []
apply(*erule-tac x=ys* **in** *allE*) **apply** *clarsimp*

apply(*erule-tac x=targetnode* (*last* (*a′#xs*)) **in** *allE*)
apply *clarsimp* **apply**(*erule-tac x=cs* **in** *allE*)
by *clarsimp*
from ‹*sourcenode a = sourcenode a′*› ‹*n = sourcenode a′*›
‹*targetnode a = targetnode* (*last* (*a′#xs*))› ‹*valid-edge a*›
‹*kind a* = $(\lambda cf.\ False)_{\surd}$› ‹*targetnode* (*last* (*a′#xs*)) $-as''\to_\iota*\ n'$›
have $n\ -a\#as''\to_\iota*\ n'$
by(*fastforce intro*:*Cons-path simp*:*intra-path-def intra-kind-def*)
moreover
from ‹*kind a* = $(\lambda cf.\ False)_{\surd}$› ‹*slice-edges S cs as″ = []*›
‹¬ *slice-edge S cs a′*› ‹*sourcenode a = sourcenode a′*›
have *slice-edges S cs* (*a#as″*) = []
by(*cases kind a′*)(*auto simp*:*slice-edge-def*)
ultimately show *?thesis* **by** *blast*
next
case (*Return Q p f*)
with *Cons* ‹*same-level-path-aux* [] *as*› **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
qed
qed
qed
qed

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] *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(*atomize-elim*)
from *assms* **show** ∃ *asx asx′ ms″ s″*. *as* = *asx* @ [*last asx′*] ∧
S,*f* ⊢ (*ms*,*s*) =*asx*⇒* (*ms″*,*s″*) ∧ *S*,*f* ⊢ (*ms″*,*s″*) =*asx′*⇒ (*ms′*,*s′*)
proof(*induct rule*:*trans-observable-moves.induct*)

```
    case (tom-Cons S f ms s as ms' s' as' ms'' s'')
    note IH = ‹as' ≠ [] ⟹ ∃ asx asx' msx sx. as' = asx @ [last asx'] ∧
      S,f ⊢ (ms',s') =asx⇒* (msx,sx) ∧ S,f ⊢ (msx,sx) =asx'⇒ (ms'',s'')›
    show ?case
    proof(cases as' = [])
      case True
      with ‹S,f ⊢ (ms',s') =as'⇒* (ms'',s'')› have [simp]:ms'' = ms' s'' = s'
        by(auto elim:trans-observable-moves.cases)
      from ‹S,f ⊢ (ms,s) =as⇒ (ms',s')› have length ms = length s
        by(rule observable-moves-equal-length)
      hence S,f ⊢ (ms,s) =[]⇒* (ms,s) by(rule tom-Nil)
      with ‹S,f ⊢ (ms,s) =as⇒ (ms',s')› True show ?thesis by fastforce
    next
      case False
      from IH[OF this] obtain xs xs' msx sx where as' = xs @ [last xs']
        and S,f ⊢ (ms',s') =xs⇒* (msx,sx)
        and S,f ⊢ (msx,sx) =xs'⇒ (ms'',s'') by blast
      from ‹S,f ⊢ (ms,s) =as⇒ (ms',s')› ‹S,f ⊢ (ms',s') =xs⇒* (msx,sx)›
      have S,f ⊢ (ms,s) =(last as)#xs⇒* (msx,sx)
        by(rule trans-observable-moves.tom-Cons)
      with ‹S,f ⊢ (msx,sx) =xs'⇒ (ms'',s'')› ‹as' = xs @ [last xs']›
      show ?thesis by fastforce
    qed
  qed simp
qed


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(atomize-elim)
  from assms show ∃ cs' rs'. valid-node m' ∧ valid-call-list cs' m' ∧
    (∀ i<length rs'. rs' ! i ∈ get-return-edges (cs' ! i)) ∧ valid-return-list rs' m' ∧
    length rs' = length cs' ∧ ms' = targetnodes rs'
  proof(induct S f m#ms s as m'#ms' s' arbitrary:m ms cs rs
      rule:trans-observable-moves.induct)
    case (tom-Nil sx n_c f)
    thus ?case
      apply(rule-tac x=cs in exI)
      apply(rule-tac x=rs in exI)
      by clarsimp
  next
    case (tom-Cons S f sx as msx' sx' as' sx'')
    note IH = ‹⋀m ms cs rs. ⟦msx' = m # ms; valid-node m; valid-call-list cs m;
```

```
    ∀ i<length rs. rs ! i ∈ get-return-edges (cs ! i); valid-return-list rs m;
    length rs = length cs; ms = targetnodes rs⟧
    ⟹ ∃ cs' rs'. valid-node m' ∧ valid-call-list cs' m' ∧
    (∀ i<length rs'. rs' ! i ∈ get-return-edges (cs' ! i)) ∧
    valid-return-list rs' m' ∧ length rs' = length cs' ∧
    ms' = targetnodes rs'⟩
  from ‹S,f ⊢ (m # ms,sx) =as⇒ (msx',sx')›
  obtain m'' ms'' where msx' = m''#ms''
  apply(cases msx') apply(auto elim!:observable-moves.cases observable-move.cases)
    by(case-tac msaa) auto
  with ‹S,f ⊢ (m # ms,sx) =as⇒ (msx',sx')› ‹valid-node m›
    ‹valid-call-list cs m› ‹∀ i<length rs. rs ! i ∈ get-return-edges (cs ! i)›
    ‹valid-return-list rs m› ‹length rs = length cs› ‹ms = targetnodes rs›
  obtain 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''
    by(auto elim!:observable-moves-preserves-stack)
  from IH[OF ‹msx' = m''#ms''› this(1-6)]
  show ?case by fastforce
 qed
qed

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'⇒τ (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(atomize-elim)
 from assms show ∃ ms ms'' s'' as' ms' as''.
   S,kind ⊢ (m # ms,s) =slice-edges S cs as⇒* (ms'',s'') ∧
   S,kind ⊢ (ms'',s'') =as'⇒τ (m' # ms',s') ∧
   ms = targetnodes rs ∧ length ms = length cs ∧
   (∀ i<length cs. call-of-return-node (ms ! i) (sourcenode (cs ! i))) ∧
   slice-edges S cs as = slice-edges S cs as'' ∧
   m -as'' @ as'→* m' ∧ valid-path-aux cs (as'' @ as')
 proof(induct arbitrary:m s rs rule:vpa-induct)
   case (vpa-empty cs)
```

from ‹m $-[]\rightarrow*$ m'› **have** [*simp*]:$m' = m$ **by** *fastforce*
from ‹*transfers* (*kinds* []) $s = s'$› ‹*length* s = *Suc* (*length* *cs*)›
have [*simp*]:$s' = s$ **by**(*cases* *cs*)(*auto* *simp*:*kinds-def*)
from ‹*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*›
have $\forall i$<*length* *cs*. *call-of-return-node* (*targetnodes* *rs*!i) (*sourcenode* (*cs*!i))
by(*rule* *get-return-edges-call-of-return-nodes*)
with ‹*length* s = *Suc* (*length* *cs*)› ‹m $-[]\rightarrow*$ m'› ‹*length* *rs* = *length* *cs*› **show** *?case*
apply(*rule-tac* *x*=*targetnodes* *rs* **in** *exI*)
apply(*rule-tac* *x*=*m*#*targetnodes* *rs* **in** *exI*)
apply(*rule-tac* *x*=s **in** *exI*)
apply(*rule-tac* *x*=[] **in** *exI*)
apply(*rule-tac* *x*=*targetnodes* *rs* **in** *exI*)
apply(*rule-tac* *x*=[] **in** *exI*)
by(*fastforce* *intro*:*tom-Nil* *silent-moves-Nil* *simp*:*targetnodes-def*)
next
case (*vpa-intra* *cs* *a* *as*)
note *IH* = ‹$\bigwedge m$ s *rs*. ⟦m $-as\rightarrow*$ m'; *preds* (*kinds* *as*) s; *transfers* (*kinds* *as*) s = s';
valid-call-list *cs* *m*; $\forall i$<*length* *rs*. *rs* ! $i \in$ *get-return-edges* (*cs* ! i);
valid-return-list *rs* *m*; *length* *rs* = *length* *cs*; *length* s = *Suc* (*length* *cs*)⟧
$\Longrightarrow \exists$ *ms* *ms*'' s'' *as*' *ms*' *as*''.
S,*kind* ⊢ (m # *ms*,s) =*slice-edges* *S* *cs* *as*⇒* (*ms*'',s'') ∧
S,*kind* ⊢ (*ms*'',s'') =*as*'⇒$_\tau$ (m' # *ms*',s') ∧ *ms* = *targetnodes* *rs* ∧
length *ms* = *length* *cs* ∧
($\forall i$<*length* *cs*. *call-of-return-node* (*ms* ! i) (*sourcenode* (*cs* ! i))) ∧
slice-edges *S* *cs* *as* = *slice-edges* *S* *cs* *as*'' ∧
m $-as''$ @ *as*'$\rightarrow*$ m' ∧ *valid-path-aux* *cs* (*as*'' @ *as*')›
from ‹m $-a$ # *as*$\rightarrow*$ m'› **have** m = *sourcenode* a **and** *valid-edge* a
and *targetnode* a $-as\rightarrow*$ m' **by**(*auto* *elim*:*path-split-Cons*)
from ‹*preds* (*kinds* (a # *as*)) s› **have** *pred* (*kind* a) s
and *preds* (*kinds* *as*) (*transfer* (*kind* a) s) **by**(*auto* *simp*:*kinds-def*)
from ‹*transfers* (*kinds* (a # *as*)) s = s'›
have *transfers* (*kinds* *as*) (*transfer* (*kind* a) s) = s' **by**(*fastforce* *simp*:*kinds-def*)
from ‹*valid-edge* a› ‹*intra-kind* (*kind* a)›
have *get-proc* (*sourcenode* a) = *get-proc* (*targetnode* a) **by**(*rule* *get-proc-intra*)
from ‹*valid-call-list* *cs* *m*› ‹m = *sourcenode* a›
‹*get-proc* (*sourcenode* a) = *get-proc* (*targetnode* a)›
have *valid-call-list* *cs* (*targetnode* a)
apply(*clarsimp* *simp*:*valid-call-list-def*)
apply(*erule-tac* *x*=*cs*' **in** *allE*)
apply(*erule-tac* *x*=*c* **in** *allE*)
by(*auto* *split*:*list.split*)
from ‹*intra-kind* (*kind* a)› ‹*length* s = *Suc* (*length* *cs*)›
have *length* (*transfer* (*kind* a) s) = *Suc* (*length* *cs*)
by(*cases* s)(*auto* *simp*:*intra-kind-def*)
from ‹*valid-return-list* *rs* *m*› ‹m = *sourcenode* a›
‹*get-proc* (*sourcenode* a) = *get-proc* (*targetnode* a)›

have *valid-return-list rs* (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*) *auto*
from *IH*[*OF* ‹*targetnode a* −*as*→∗ *m'*› ‹*preds* (*kinds as*) (*transfer* (*kind a*) *s*)›
‹*transfers* (*kinds as*) (*transfer* (*kind a*) *s*) = *s'*›
‹*valid-call-list cs* (*targetnode a*)›
‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)› *this* ‹*length rs* = *length cs*›
‹*length* (*transfer* (*kind a*) *s*) = *Suc* (*length cs*)›]
obtain *ms ms''* *s''* *as'* *ms'* *as''* **where** *length ms* = *length cs*
and *S*,*kind* ⊢ (*targetnode a* # *ms*,*transfer* (*kind a*) *s*) =*slice-edges S cs as*⇒∗
(*ms''*,*s''*)
and *paths*:*S*,*kind* ⊢ (*ms''*,*s''*) =*as'*⇒$_\tau$ (*m'* # *ms'*,*s'*)
ms = *targetnodes rs*
∀ *i*<*length cs*. *call-of-return-node* (*ms* ! *i*) (*sourcenode* (*cs* ! *i*))
slice-edges S cs as = *slice-edges S cs as''*
targetnode a −*as''* @ *as'*→∗ *m'* *valid-path-aux cs* (*as''* @ *as'*)
by *blast*
from ‹∀ *i*<*length cs*. *call-of-return-node* (*ms* ! *i*) (*sourcenode* (*cs* ! *i*))›
‹*length ms* = *length cs*›
have ∀ *mx* ∈ *set ms*. *return-node mx*
by(*auto simp*:*call-of-return-node-def in-set-conv-nth*)
show *?case*
proof(*cases* (∀ *m* ∈ *set ms*. ∃ *m'*. *call-of-return-node m m'* ∧
m' ∈ ⌊*HRB-slice S*⌋$_{CFG}$) ∧ *m* ∈ ⌊*HRB-slice S*⌋$_{CFG}$)
case *True*
with ‹*m* = *sourcenode a*› ‹*length ms* = *length cs*› ‹*intra-kind* (*kind a*)›
‹∀ *i*<*length cs*. *call-of-return-node* (*ms* ! *i*) (*sourcenode* (*cs* ! *i*))›
have *slice-edge S cs a*
by(*fastforce simp*:*slice-edge-def in-set-conv-nth intra-kind-def*)
with ‹*intra-kind* (*kind a*)›
have *slice-edges S cs* (*a*#*as*) = *a*#*slice-edges S cs as*
by(*fastforce simp*:*intra-kind-def*)
from *True* ‹*pred* (*kind a*) *s*› ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
‹∀ *mx* ∈ *set ms*. *return-node mx*› ‹*length ms* = *length cs*› ‹*m* = *sourcenode a*›
‹*length s* = *Suc* (*length cs*)› ‹*length* (*transfer* (*kind a*) *s*) = *Suc* (*length cs*)›
have *S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) −*a*→ (*targetnode a*#*ms*,*transfer* (*kind a*) *s*)
by(*fastforce intro*!:*observable-move-intra*)
with ‹*length ms* = *length cs*› ‹*length s* = *Suc* (*length cs*)›
have *S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) =[]@[*a*]⇒
(*targetnode a*#*ms*,*transfer* (*kind a*) *s*)
by(*fastforce intro*:*observable-moves-snoc silent-moves-Nil*)
with ‹*S*,*kind* ⊢ (*targetnode a* # *ms*,*transfer* (*kind a*) *s*) =*slice-edges S cs as*⇒∗
(*ms''*,*s''*)›
have *S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) =*last* [*a*]#*slice-edges S cs as*⇒∗ (*ms''*,*s''*)
by(*fastforce intro*:*tom-Cons*)

with ‹*slice-edges S cs* (*a*#*as*) = *a*#*slice-edges S cs as*›
have *S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) =*slice-edges S cs* (*a*#*as*)⇒∗ (*ms*′′,*s*′′)
by *simp*
moreover
from ‹*slice-edges S cs as* = *slice-edges S cs as*′′› ‹*slice-edge S cs a*›
‹*intra-kind* (*kind a*)›
have *slice-edges S cs* (*a*#*as*) = *slice-edges S cs* (*a*#*as*′′)
by(*fastforce simp*:*intra-kind-def*)
ultimately show *?thesis*
using *paths* ‹*m* = *sourcenode a*› ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
‹*length ms* = *length cs*› ‹*slice-edges S cs* (*a*#*as*) = *a*#*slice-edges S cs as*›
apply(*rule-tac x*=*ms* **in** *exI*)
apply(*rule-tac x*=*ms*′′ **in** *exI*)
apply(*rule-tac x*=*s*′′ **in** *exI*)
apply(*rule-tac x*=*as*′ **in** *exI*)
apply(*rule-tac x*=*ms*′ **in** *exI*)
apply(*rule-tac x*=*a*#*as*′′ **in** *exI*)
by(*auto intro*:*Cons-path simp*:*intra-kind-def*)
next
case *False*
with ‹∀ *mx* ∈ *set ms*. *return-node mx*›
have *disj*:(∃ *m* ∈ *set ms*. ∃ *m*′. *call-of-return-node m m*′ ∧
m′ ∉ ⌊*HRB-slice S*⌋$_{CFG}$) ∨ *m* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
by(*fastforce dest*:*return-node-call-of-return-node*)
with ‹*m* = *sourcenode a*› ‹*length ms* = *length cs*› ‹*intra-kind* (*kind a*)›
‹∀ *i*<*length cs*. *call-of-return-node* (*ms* ! *i*) (*sourcenode* (*cs* ! *i*))›
have ¬ *slice-edge S cs a*
by(*fastforce simp*:*slice-edge-def in-set-conv-nth intra-kind-def*)
with ‹*intra-kind* (*kind a*)›
have *slice-edges S cs* (*a*#*as*) = *slice-edges S cs as*
by(*fastforce simp*:*intra-kind-def*)
from *disj* ‹*pred* (*kind a*) *s*› ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
‹∀ *mx* ∈ *set ms*. *return-node mx*› ‹*length ms* = *length cs*› ‹*m* = *sourcenode a*›
‹*length s* = *Suc* (*length cs*)› ‹*length* (*transfer* (*kind a*) *s*) = *Suc* (*length cs*)›
have *S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) −*a*→$_\tau$ (*targetnode a*#*ms*,*transfer* (*kind a*) *s*)
by(*fastforce intro*!:*silent-move-intra*)
from ‹*S*,*kind* ⊢ (*targetnode a* # *ms*,*transfer* (*kind a*) *s*) =*slice-edges S cs as*⇒∗
(*ms*′′,*s*′′)›
show *?thesis*
proof(*rule trans-observable-moves.cases*)
fix *msx sx* n_c′ *f*
assume *targetnode a* # *ms* = *msx*
and *transfer* (*kind a*) *s* = *sx* **and** *slice-edges S cs as* = []
and [*simp*]:*ms*′′ = *msx s*′′ = *sx* **and** *length msx* = *length sx*
from ‹*slice-edges S cs* (*a*#*as*) = *slice-edges S cs as*›
‹*slice-edges S cs as* = []›

have *slice-edges S cs* (*a*#*as*) = [] **by** *simp*
with ‹*length ms* = *length cs*› ‹*length s* = *Suc* (*length cs*)›
have *S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) =*slice-edges S cs* (*a*#*as*)⇒∗
(*sourcenode a*#*ms*,*s*)
by(*fastforce intro*:*tom-Nil*)
moreover
from ‹*S*,*kind* ⊢ (*ms*′′,*s*′′) =*as*′⇒$_\tau$ (*m*′#*ms*′,*s*′)› ‹*targetnode a* # *ms* = *msx*›
‹*transfer* (*kind a*) *s* = *sx*› ‹*ms*′′ = *msx*› ‹*s*′′ = *sx*›
‹*S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) −*a*→$_\tau$ (*targetnode a*#*ms*,*transfer* (*kind a*) *s*)›
have *S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) =*a*#*as*′⇒$_\tau$ (*m*′#*ms*′,*s*′)
by(*fastforce intro*:*silent-moves-Cons*)
from *this* ‹*valid-edge a*› ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*ms* = *targetnodes rs*› ‹*valid-return-list rs m*› ‹*length rs* = *length cs*›
‹*length s* = *Suc* (*length cs*)› ‹*m* = *sourcenode a*›
have *sourcenode a* −*a*#*as*′→∗ *m*′ **and** *valid-path-aux cs* (*a*#*as*′)
by −(*rule silent-moves-vpa-path*,(*fastforce simp*:*targetnodes-def*)+)+
ultimately show *?thesis* **using** ‹*m* = *sourcenode a*› ‹*length ms* = *length cs*›
‹∀ *i*<*length cs*. *call-of-return-node* (*ms* ! *i*) (*sourcenode* (*cs* ! *i*))›
‹*slice-edges S cs* (*a*#*as*) = []› ‹*intra-kind* (*kind a*)›
‹*S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) =*a*#*as*′⇒$_\tau$ (*m*′#*ms*′,*s*′)›
‹*ms* = *targetnodes rs*›
apply(*rule-tac x*=*ms* **in** *exI*)
apply(*rule-tac x*=*sourcenode a*#*ms* **in** *exI*)
apply(*rule-tac x*=*s* **in** *exI*)
apply(*rule-tac x*=*a*#*as*′ **in** *exI*)
apply(*rule-tac x*=*ms*′ **in** *exI*)
apply(*rule-tac x*=[] **in** *exI*)
by(*auto simp*:*intra-kind-def*)
next
fix *S*′ *f msx sx asx msx*′ *sx*′ *asx*′ *msx*′′ *sx*′′
assume [*simp*]:*S* = *S*′ **and** *kind* = *f* **and** *targetnode a* # *ms* = *msx*
and *transfer* (*kind a*) *s* = *sx* **and** *slice-edges S cs as* = *last asx* # *asx*′
and *ms*′′ = *msx*′′ **and** *s*′′ = *sx*′′
and *S*′,*f* ⊢ (*msx*,*sx*) =*asx*⇒ (*msx*′,*sx*′)
and *S*′,*f* ⊢ (*msx*′,*sx*′) =*asx*′⇒∗ (*msx*′′,*sx*′′)
from ‹*kind* = *f*› **have** [*simp*]:*f* = *kind* **by** *simp*
from ‹*S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) −*a*→$_\tau$
(*targetnode a*#*ms*,*transfer* (*kind a*) *s*)› ‹*S*′,*f* ⊢ (*msx*,*sx*) =*asx*⇒ (*msx*′,*sx*′)›
‹*transfer* (*kind a*) *s* = *sx*› ‹*targetnode a* # *ms* = *msx*›
have *S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) =*a*#*asx*⇒ (*msx*′,*sx*′)
by(*fastforce intro*:*silent-move-observable-moves*)
with ‹*S*′,*f* ⊢ (*msx*′,*sx*′) =*asx*′⇒∗ (*msx*′′,*sx*′′)› ‹*ms*′′ = *msx*′′› ‹*s*′′ = *sx*′′›
have *S*,*kind* ⊢ (*sourcenode a*#*ms*,*s*) =*last* (*a*#*asx*)#*asx*′⇒∗ (*ms*′′,*s*′′)
by(*fastforce intro*:*trans-observable-moves.tom-Cons*)
moreover
from ‹*S*′,*f* ⊢ (*msx*,*sx*) =*asx*⇒ (*msx*′,*sx*′)› **have** *asx* ≠ []
by(*fastforce elim*:*observable-moves.cases*)

```
      with ‹slice-edges S cs (a#as) = slice-edges S cs as›
        ‹slice-edges S cs as = last asx # asx'›
      have slice-edges S cs (a#as) = last (a#asx)#asx' by simp
      moreover
      from ‹¬ slice-edge S cs a› ‹slice-edges S cs as = slice-edges S cs as''›
        ‹intra-kind (kind a)›
      have slice-edges S cs (a # as) = slice-edges S cs (a # as'')
        by(fastforce simp:intra-kind-def)
      ultimately show ?thesis using paths ‹m = sourcenode a› ‹intra-kind (kind
a)›
        ‹length ms = length cs› ‹ms = targetnodes rs› ‹valid-edge a›
        apply(rule-tac x=ms in exI)
        apply(rule-tac x=ms'' in exI)
        apply(rule-tac x=s'' in exI)
        apply(rule-tac x=as' in exI)
        apply(rule-tac x=ms' in exI)
        apply(rule-tac x=a#as'' in exI)
        by(auto intro:Cons-path simp:intra-kind-def)
    qed
   qed
  next
   case (vpa-Call cs a as Q r p fs)
   note IH = ‹⋀m s rs. ⟦m −as→* m'; preds (kinds as) s; transfers (kinds as)
s = s';
    valid-call-list (a # cs) m;
    ∀ i<length rs. rs ! i ∈ get-return-edges ((a # cs) ! i);
    valid-return-list rs m; length rs = length (a # cs);
    length s = Suc (length (a # cs))⟧
    ⟹ ∃ms ms'' s'' as' ms' as''.
    S,kind ⊢ (m # ms,s) =slice-edges S (a # cs) as⇒* (ms'',s'') ∧
    S,kind ⊢ (ms'',s'') =as'⇒τ (m' # ms',s') ∧ ms = targetnodes rs ∧
    length ms = length (a # cs) ∧
    (∀ i<length (a # cs). call-of-return-node (ms ! i) (sourcenode ((a # cs) ! i)))
∧
    slice-edges S (a # cs) as = slice-edges S (a # cs) as'' ∧
    m −as'' @ as'→* m' ∧ valid-path-aux (a # cs) (as'' @ as')›
   from ‹m −a # as→* m'› have m = sourcenode a and valid-edge a
    and targetnode a −as→* m' by(auto elim:path-split-Cons)
   from ‹preds (kinds (a # as)) s› have pred (kind a) s
    and preds (kinds as) (transfer (kind a) s) by(auto simp:kinds-def)
   from ‹transfers (kinds (a # as)) s = s'›
  have transfers (kinds as) (transfer (kind a) s) = s' by(fastforce simp:kinds-def)
   from ‹valid-edge a› ‹kind a = Q:r↪pfs› have get-proc (targetnode a) = p
    by(rule get-proc-call)
   with ‹valid-call-list cs m› ‹valid-edge a› ‹kind a = Q:r↪pfs› ‹m = sourcenode
a›
   have valid-call-list (a # cs) (targetnode a)
    apply(clarsimp simp:valid-call-list-def)
    apply(case-tac cs') apply auto
```

apply(*erule-tac x=list* **in** *allE*)
by(*case-tac list*)(*auto simp*:*sourcenodes-def*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› **obtain** *a'* **where** *a'* ∈ *get-return-edges a*
by(*fastforce dest*:*get-return-edge-call*)
with ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› **obtain** *Q' f'* **where** *kind a'* = *Q'*↩$_p$*f'*
by(*fastforce dest*!:*call-return-edges*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*› **have** *valid-edge a'*
by(*rule get-return-edges-valid*)
from ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩$_p$*f'*› **have** *get-proc* (*sourcenode a'*) = *p*
by(*rule get-proc-return*)
from ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)› ‹*a'* ∈ *get-return-edges a*›
have ∀ *i*<*length* (*a'*#*rs*). (*a'*#*rs*) ! *i* ∈ *get-return-edges* ((*a*#*cs*) ! *i*)
by *auto*(*case-tac i*,*auto*)
from ‹*valid-edge a*› ‹*a'* ∈ *get-return-edges a*›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a'*)
by(*rule get-proc-get-return-edge*)
with ‹*valid-return-list rs m*› ‹*valid-edge a'*› ‹*kind a'* = *Q'*↩$_p$*f'*›
‹*get-proc* (*sourcenode a'*) = *p*› ‹*get-proc* (*targetnode a*) = *p*› ‹*m* = *sourcenode a*›
have *valid-return-list* (*a'*#*rs*) (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*case-tac cs'*) **apply** *auto*
apply(*erule-tac x=list* **in** *allE*)
by(*case-tac list*)(*auto simp*:*targetnodes-def*)
from ‹*length rs* = *length cs*› **have** *length* (*a'*#*rs*) = *length* (*a*#*cs*) **by** *simp*
from ‹*length s* = *Suc* (*length cs*)› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
have *length* (*transfer* (*kind a*) *s*) = *Suc* (*length* (*a*#*cs*))
by(*cases s*) *auto*
from *IH*[*OF* ‹*targetnode a* −*as*→∗ *m'*› ‹*preds* (*kinds as*) (*transfer* (*kind a*) *s*)›
‹*transfers* (*kinds as*) (*transfer* (*kind a*) *s*) = *s'*›
‹*valid-call-list* (*a* # *cs*) (*targetnode a*)›
‹∀ *i*<*length* (*a'*#*rs*). (*a'*#*rs*) ! *i* ∈ *get-return-edges* ((*a*#*cs*) ! *i*)›
‹*valid-return-list* (*a'*#*rs*) (*targetnode a*)› ‹*length* (*a'*#*rs*) = *length* (*a*#*cs*)›
‹*length* (*transfer* (*kind a*) *s*) = *Suc* (*length* (*a*#*cs*))›]
obtain *ms ms''* *s''* *as'* *ms'* *as''* **where** *length ms* = *length* (*a*#*cs*)
and *S*,*kind* ⊢ (*targetnode a* # *ms*,*transfer* (*kind a*) *s*)
=*slice-edges S* (*a*#*cs*) *as*⇒∗ (*ms''*,*s''*)
and *paths*:*S*,*kind* ⊢ (*ms''*,*s''*) =*as'*⇒$_\tau$ (*m'* # *ms'*,*s'*)
ms = *targetnodes* (*a'*#*rs*)
∀ *i*<*length* (*a*#*cs*). *call-of-return-node* (*ms* ! *i*) (*sourcenode* ((*a*#*cs*) ! *i*))
slice-edges S (*a*#*cs*) *as* = *slice-edges S* (*a*#*cs*) *as''*
targetnode a −*as''* @ *as'*→∗ *m' valid-path-aux* (*a*#*cs*) (*as''* @ *as'*)
by *blast*
from ‹*ms* = *targetnodes* (*a'*#*rs*)› **obtain** *x xs* **where** [*simp*]:*ms* = *x*#*xs*
and *x* = *targetnode a'* **and** *xs* = *targetnodes rs*
by(*cases ms*)(*auto simp*:*targetnodes-def*)
from ‹∀ *i*<*length* (*a*#*cs*). *call-of-return-node* (*ms* ! *i*) (*sourcenode* ((*a*#*cs*) ! *i*))›

```
    ‹length ms = length (a#cs)›
  have ∀ mx ∈ set xs. return-node mx
    apply(auto simp:in-set-conv-nth) apply(case-tac i)
    apply(erule-tac x=Suc 0 in allE)
    by(auto simp:call-of-return-node-def)
  show ?case
  proof(cases (∀ m ∈ set xs. ∃ m'. call-of-return-node m m' ∧
      m' ∈ ⌊HRB-slice S⌋CFG) ∧ sourcenode a ∈ ⌊HRB-slice S⌋CFG)
    case True
    with ‹∀ i<length (a#cs). call-of-return-node (ms ! i) (sourcenode ((a#cs) !
i))›
      ‹length ms = length (a#cs)› ‹kind a = Q:r↪pfs›
    have slice-edge S cs a
      apply(auto simp:slice-edge-def in-set-conv-nth)
      by(erule-tac x=Suc i in allE) auto
    with ‹kind a = Q:r↪pfs›
    have slice-edges S cs (a#as) = a#slice-edges S (a#cs) as by simp
    from True ‹pred (kind a) s› ‹valid-edge a› ‹kind a = Q:r↪pfs›
      ‹valid-edge a'› ‹a' ∈ get-return-edges a›
     ‹∀ mx ∈ set xs. return-node mx› ‹length ms = length (a#cs)› ‹m = sourcenode
a›
      ‹length s = Suc (length cs)›
      ‹length (transfer (kind a) s) = Suc (length (a#cs))›
    have S,kind ⊢ (sourcenode a#xs,s) −a→
      (targetnode a#targetnode a'#xs,transfer (kind a) s)
      by −(rule-tac a'=a' in observable-move-call,fastforce+)
    with ‹length ms = length (a#cs)› ‹length s = Suc (length cs)›
    have S,kind ⊢ (sourcenode a#xs,s) =[]@[a]⇒
      (targetnode a#targetnode a'#xs,transfer (kind a) s)
      by(fastforce intro:observable-moves-snoc silent-moves-Nil)
    with ‹S,kind ⊢ (targetnode a # ms,transfer (kind a) s)
      =slice-edges S (a#cs) as⇒* (ms'',s'')› ‹x = targetnode a'›
    have S,kind ⊢ (sourcenode a#xs,s) =last [a]#slice-edges S (a#cs) as⇒*
      (ms'',s'')
      by −(rule tom-Cons,auto)
    with ‹slice-edges S cs (a#as) = a#slice-edges S (a#cs) as›
    have S,kind ⊢ (sourcenode a#xs,s) =slice-edges S cs (a#as)⇒* (ms'',s'')
      by simp
    moreover
    from ‹slice-edges S (a#cs) as = slice-edges S (a#cs) as''›
      ‹slice-edge S cs a› ‹kind a = Q:r↪pfs›
    have slice-edges S cs (a#as) = slice-edges S cs (a#as'') by simp
    ultimately show ?thesis
      using paths ‹m = sourcenode a› ‹valid-edge a› ‹kind a = Q:r↪pfs›
        ‹length ms = length (a#cs)› ‹xs = targetnodes rs›
        ‹slice-edges S cs (a#as) = a#slice-edges S (a#cs) as›
      apply(rule-tac x=xs in exI)
      apply(rule-tac x=ms'' in exI)
      apply(rule-tac x=s'' in exI)
```

apply(*rule-tac x=as′* **in** *exI*)
apply(*rule-tac x=ms′* **in** *exI*)
apply(*rule-tac x=a#as′′* **in** *exI*)
by(*auto intro:Cons-path simp:targetnodes-def*)
next
case *False*
with ‹∀ *mx* ∈ *set xs*. *return-node mx*›
have *disj*:(∃ *m* ∈ *set xs*. ∃ *m′*. *call-of-return-node m m′* ∧
m′ ∉ ⌊*HRB-slice S*⌋$_{CFG}$) ∨ *sourcenode a* ∉ ⌊*HRB-slice S*⌋$_{CFG}$
by(*fastforce dest:return-node-call-of-return-node*)
with ‹∀ *i*<*length* (*a*#*cs*). *call-of-return-node* (*ms* ! *i*) (*sourcenode* ((*a*#*cs*) ! *i*))›
‹*length ms* = *length* (*a*#*cs*)› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
have ¬ *slice-edge S cs a*
apply(*auto simp:slice-edge-def in-set-conv-nth*)
by(*erule-tac x=Suc i* **in** *allE*) *auto*
with ‹*kind a* = *Q*:*r*↪$_p$*fs*›
have *slice-edges S cs* (*a*#*as*) = *slice-edges S* (*a*#*cs*) *as* **by** *simp*
from *disj* ‹*pred* (*kind a*) *s*› ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
‹*valid-edge a′*› ‹*a′* ∈ *get-return-edges a*›
‹∀ *mx* ∈ *set xs*. *return-node mx*› ‹*length ms* = *length* (*a*#*cs*)› ‹*m* = *sourcenode a*›
‹*length s* = *Suc* (*length cs*)›
‹*length* (*transfer* (*kind a*) *s*) = *Suc* (*length* (*a*#*cs*))›
have *S*,*kind* ⊢ (*sourcenode a*#*xs*,*s*) −*a*→$_\tau$
(*targetnode a*#*targetnode a′*#*xs*,*transfer* (*kind a*) *s*)
by −(*rule-tac a′*=*a′* **in** *silent-move-call*,*fastforce*+)
from ‹*S*,*kind* ⊢ (*targetnode a* # *ms*,*transfer* (*kind a*) *s*)
=*slice-edges S* (*a*#*cs*) *as*⇒∗ (*ms′′*,*s′′*)›
show *?thesis*
proof(*rule trans-observable-moves.cases*)
fix *msx sx S′ f*
assume *targetnode a* # *ms* = *msx*
and *transfer* (*kind a*) *s* = *sx* **and** *slice-edges S* (*a*#*cs*) *as* = []
and [*simp*]:*ms′′* = *msx s′′* = *sx* **and** *length msx* = *length sx*
from ‹*slice-edges S cs* (*a*#*as*) = *slice-edges S* (*a*#*cs*) *as*›
‹*slice-edges S* (*a*#*cs*) *as* = []›
have *slice-edges S cs* (*a*#*as*) = [] **by** *simp*
with ‹*length ms* = *length* (*a*#*cs*)› ‹*length s* = *Suc* (*length cs*)›
have *S*,*kind* ⊢ (*sourcenode a*#*xs*,*s*) =*slice-edges S cs* (*a*#*as*)⇒∗
(*sourcenode a*#*xs*,*s*)
by(*fastforce intro:tom-Nil*)
moreover
from ‹*S*,*kind* ⊢ (*ms′′*,*s′′*) =*as′*⇒$_\tau$ (*m′*#*ms′*,*s′*)› ‹*targetnode a* # *ms* = *msx*›
‹*transfer* (*kind a*) *s* = *sx*› ‹*ms′′* = *msx*› ‹*s′′* = *sx*› ‹*x* = *targetnode a′*›
‹*S*,*kind* ⊢ (*sourcenode a*#*xs*,*s*) −*a*→$_\tau$
(*targetnode a*#*targetnode a′*#*xs*,*transfer* (*kind a*) *s*)›
have *S*,*kind* ⊢ (*sourcenode a*#*xs*,*s*) =*a*#*as′*⇒$_\tau$ (*m′*#*ms′*,*s′*)
by(*auto intro:silent-moves-Cons*)

from *this* ‹*valid-edge a*›
‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*xs* = *targetnodes rs*› ‹*valid-return-list rs m*› ‹*length rs* = *length cs*›
‹*length s* = *Suc* (*length cs*)› ‹*m* = *sourcenode a*›
have *sourcenode a* −*a*#*as*′→∗ *m*′ **and** *valid-path-aux cs* (*a*#*as*′)
by −(*rule silent-moves-vpa-path*,(*fastforce simp*:*targetnodes-def*)+)+
ultimately show *?thesis* **using** ‹*m* = *sourcenode a*› ‹*length ms* = *length* (*a*#*cs*)›
‹∀ *i*<*length* (*a*#*cs*). *call-of-return-node* (*ms* ! *i*) (*sourcenode* ((*a*#*cs*) ! *i*))›
‹*slice-edges S cs* (*a*#*as*) = []› ‹*kind a* = *Q*:*r*↪*pfs*›
‹*S*,*kind* ⊢ (*sourcenode a*#*xs*,*s*) =*a*#*as*′⇒τ (*m*′#*ms*′,*s*′)›
‹*xs* = *targetnodes rs*›
apply(*rule-tac x*=*xs* **in** *exI*)
apply(*rule-tac x*=*sourcenode a*#*xs* **in** *exI*)
apply(*rule-tac x*=*s* **in** *exI*)
apply(*rule-tac x*=*a*#*as*′ **in** *exI*)
apply(*rule-tac x*=*ms*′ **in** *exI*)
apply(*rule-tac x*=[] **in** *exI*)
by *auto*
next
fix *S*′ *f msx sx asx msx*′ *sx*′ *asx*′ *msx*′′ *sx*′′
assume [*simp*]:*S* = *S*′ **and** *kind* = *f* **and** *targetnode a* # *ms* = *msx*
and *transfer* (*kind a*) *s* = *sx*
and *slice-edges S* (*a*#*cs*) *as* = *last asx* # *asx*′
and *ms*′′ = *msx*′′ **and** *s*′′ = *sx*′′
and *S*′,*f* ⊢ (*msx*,*sx*) =*asx*⇒ (*msx*′,*sx*′)
and *S*′,*f* ⊢ (*msx*′,*sx*′) =*asx*′⇒∗ (*msx*′′,*sx*′′)
from ‹*kind* = *f*› **have** [*simp*]:*f* = *kind* **by** *simp*
from ‹*S*,*kind* ⊢ (*sourcenode a*#*xs*,*s*) −*a*→τ
(*targetnode a*#*targetnode a*′#*xs*,*transfer* (*kind a*) *s*)›
‹*S*′,*f* ⊢ (*msx*,*sx*) =*asx*⇒ (*msx*′,*sx*′)› ‹*x* = *targetnode a*′›
‹*transfer* (*kind a*) *s* = *sx*› ‹*targetnode a* # *ms* = *msx*›
have *S*,*kind* ⊢ (*sourcenode a*#*xs*,*s*) =*a*#*asx*⇒ (*msx*′,*sx*′)
by(*auto intro*:*silent-move-observable-moves*)
with ‹*S*′,*f* ⊢ (*msx*′,*sx*′) =*asx*′⇒∗ (*msx*′′,*sx*′′)› ‹*ms*′′ = *msx*′′› ‹*s*′′ = *sx*′′›
have *S*,*kind* ⊢ (*sourcenode a*#*xs*,*s*) =*last* (*a*#*asx*)#*asx*′⇒∗ (*ms*′′,*s*′′)
by(*fastforce intro*:*trans-observable-moves*.*tom-Cons*)
moreover
from ‹*S*′,*f* ⊢ (*msx*,*sx*) =*asx*⇒ (*msx*′,*sx*′)› **have** *asx* ≠ []
by(*fastforce elim*:*observable-moves*.*cases*)
with ‹*slice-edges S cs* (*a*#*as*) = *slice-edges S* (*a*#*cs*) *as*›
‹*slice-edges S* (*a*#*cs*) *as* = *last asx* # *asx*′›
have *slice-edges S cs* (*a*#*as*) = *last* (*a*#*asx*)#*asx*′ **by** *simp*
moreover
from ‹¬ *slice-edge S cs a*› ‹*kind a* = *Q*:*r*↪*pfs*›
‹*slice-edges S* (*a*#*cs*) *as* = *slice-edges S* (*a*#*cs*) *as*′′›
have *slice-edges S cs* (*a* # *as*) = *slice-edges S cs* (*a* # *as*′′) **by** *simp*
ultimately show *?thesis* **using** *paths* ‹*m* = *sourcenode a*› ‹*kind a* = *Q*:*r*↪*pfs*›

```
        ‹length ms = length (a#cs)› ‹xs = targetnodes rs› ‹valid-edge a›
        apply(rule-tac x=xs in exI)
        apply(rule-tac x=ms'' in exI)
        apply(rule-tac x=s'' in exI)
        apply(rule-tac x=as' in exI)
        apply(rule-tac x=ms' in exI)
        apply(rule-tac x=a#as'' in exI)
        by(auto intro:Cons-path simp:targetnodes-def)
    qed
  qed
 next
  case (vpa-ReturnEmpty cs a as Q p f)
  from ‹preds (kinds (a # as)) s› ‹length s = Suc (length cs)› ‹kind a = Q↩pf›
    ‹cs = []›
  have False by(cases s)(auto simp:kinds-def)
  thus ?case by simp
 next
  case (vpa-ReturnCons cs a as Q p f c' cs')
  note IH = ‹⋀m s rs. ⟦m −as→* m'; preds (kinds as) s; transfers (kinds as)
s = s';
    valid-call-list cs' m; ∀ i<length rs. rs ! i ∈ get-return-edges (cs' ! i);
    valid-return-list rs m; length rs = length cs'; length s = Suc (length cs')⟧
    ⟹ ∃ms ms'' s'' as' ms' as''.
    S,kind ⊢ (m # ms,s) =slice-edges S cs' as⇒* (ms'',s'') ∧
    S,kind ⊢ (ms'',s'') =as'⇒τ (m' # ms',s') ∧ ms = targetnodes rs ∧
    length ms = length cs' ∧
    (∀ i<length cs'. call-of-return-node (ms ! i) (sourcenode (cs' ! i))) ∧
    slice-edges S cs' as = slice-edges S cs' as'' ∧
    m −as'' @ as'→* m' ∧ valid-path-aux cs' (as'' @ as')›
  from ‹m −a # as→* m'› have m = sourcenode a and valid-edge a
    and targetnode a −as→* m' by(auto elim:path-split-Cons)
  from ‹preds (kinds (a # as)) s› have pred (kind a) s
    and preds (kinds as) (transfer (kind a) s) by(auto simp:kinds-def)
  from ‹transfers (kinds (a # as)) s = s'›
 have transfers (kinds as) (transfer (kind a) s) = s' by(fastforce simp:kinds-def)
  from ‹valid-call-list cs m› ‹cs = c' # cs'› have valid-edge c'
    by(fastforce simp:valid-call-list-def)
  from ‹valid-edge c'› ‹a ∈ get-return-edges c'›
  have get-proc (sourcenode c') = get-proc (targetnode a)
    by(rule get-proc-get-return-edge)
  from ‹valid-call-list cs m› ‹cs = c' # cs'›
    ‹get-proc (sourcenode c') = get-proc (targetnode a)›
  have valid-call-list cs' (targetnode a)
    apply(clarsimp simp:valid-call-list-def)
    apply(erule-tac x=c' # cs' in allE)
    by(case-tac cs')(auto simp:sourcenodes-def)
  from ‹length rs = length cs› ‹cs = c' # cs'› obtain r' rs'
    where [simp]:rs = r'#rs' and length rs' = length cs' by(cases rs) auto
  from ‹∀ i<length rs. rs ! i ∈ get-return-edges (cs ! i)› ‹cs = c' # cs'›
```

have $\forall i{<}length\ rs'.\ rs'\ !\ i \in get\text{-}return\text{-}edges\ (cs'\ !\ i)$
and $r' \in get\text{-}return\text{-}edges\ c'$ **by** *auto*
with ‹*valid-edge* c'› ‹$a \in get\text{-}return\text{-}edges\ c'$› **have** [*simp*]:$a = r'$
by −(*rule get-return-edges-unique*)
with ‹*valid-return-list rs m*›
have *valid-return-list* rs' (*targetnode a*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac* $x{=}r'\ \#\ cs'$ **in** *allE*)
by(*case-tac* cs')(*auto simp*:*targetnodes-def*)
from ‹*length* $s = Suc\ (length\ cs)$› ‹$cs = c'\ \#\ cs'$› ‹*kind* $a = Q{\hookleftarrow}_{pf}$›
have *length* (*transfer* (*kind a*) *s*) = *Suc* (*length* cs')
by(*cases s*)(*auto*,*case-tac list*,*auto*)
from *IH*[*OF* ‹*targetnode a* $-as{\rightarrow}*$ m'› ‹*preds* (*kinds as*) (*transfer* (*kind a*) *s*)›
‹*transfers* (*kinds as*) (*transfer* (*kind a*) *s*) = s'›
‹*valid-call-list* cs' (*targetnode a*)›
‹$\forall i{<}length\ rs'.\ rs'\ !\ i \in get\text{-}return\text{-}edges\ (cs'\ !\ i)$›
‹*valid-return-list* rs' (*targetnode a*)› ‹*length* $rs' = length\ cs'$› *this*]
obtain *ms* ms'' s'' as' ms' as'' **where** *length ms* = *length* cs'
and S,*kind* $\vdash$ (*targetnode a* # *ms*,*transfer* (*kind a*) *s*)
$=$*slice-edges S* cs' *as*$\Rightarrow*$ (ms'',s'')
and *paths*:S,*kind* $\vdash$ (ms'',s'') $=as'\Rightarrow_{\tau}$ (m' # ms',s')
ms = *targetnodes* rs'
$\forall i{<}length\ cs'.$ *call-of-return-node* (*ms* ! *i*) (*sourcenode* (cs' ! *i*))
slice-edges S cs' *as* = *slice-edges S* cs' as''
targetnode a $-as''$ @ $as'{\rightarrow}*$ m' *valid-path-aux* cs' (as'' @ as')
by *blast*
from ‹$\forall i{<}length\ cs'.$ *call-of-return-node* (*ms* ! *i*) (*sourcenode* (cs' ! *i*))›
‹*length ms* = *length* cs'›
have $\forall mx \in set\ ms.$ *return-node mx*
by(*auto simp*:*in-set-conv-nth call-of-return-node-def*)
from ‹*valid-edge a*› ‹*valid-edge* c'› ‹$a \in get\text{-}return\text{-}edges\ c'$›
have *return-node* (*targetnode a*) **by**(*fastforce simp*:*return-node-def*)
with ‹*valid-edge* c'› ‹*valid-edge a*› ‹$a \in get\text{-}return\text{-}edges\ c'$›
have *call-of-return-node* (*targetnode a*) (*sourcenode* c')
by(*simp add*:*call-of-return-node-def*) *blast*
show *?case*
proof(*cases* ($\forall m \in set$ (*targetnode a*#*ms*). $\exists m'.$ *call-of-return-node m* $m' \wedge$
$m' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$))
case *True*
then obtain *x* **where** *call-of-return-node* (*targetnode a*) *x*
and $x \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **by** *fastforce*
with ‹*call-of-return-node* (*targetnode a*) (*sourcenode* c')›
have *sourcenode* $c' \in \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$ **by** *fastforce*
with *True* ‹$\forall i{<}length\ cs'.$ *call-of-return-node* (*ms* ! *i*) (*sourcenode* (cs' ! *i*))›
‹*length ms* = *length* cs'› ‹$cs = c'\ \#\ cs'$› ‹*kind* $a = Q{\hookleftarrow}_{pf}$›
have *slice-edge S cs a*
apply(*auto simp*:*slice-edge-def in-set-conv-nth*)
by(*erule-tac* $x{=}i$ **in** *allE*) *auto*
with ‹*kind* $a = Q{\hookleftarrow}_{pf}$› ‹$cs = c'\ \#\ cs'$›

have *slice-edges S cs* (*a*#*as*) = *a*#*slice-edges S cs′ as* **by** *simp*
from *True* ‹*pred* (*kind a*) *s*› ‹*valid-edge a*› ‹*kind a* = *Q*↪$_{pf}$›
‹∀ *mx* ∈ *set ms*. *return-node mx*› ‹*length ms* = *length cs′*›
‹*length s* = *Suc* (*length cs*)› ‹*m* = *sourcenode a*›
‹*length* (*transfer* (*kind a*) *s*) = *Suc* (*length cs′*)›
‹*return-node* (*targetnode a*)› ‹*cs* = *c′* # *cs′*›
have *S*,*kind* ⊢ (*sourcenode a*#*targetnode a*#*ms*,*s*) −*a*→
(*targetnode a*#*ms*,*transfer* (*kind a*) *s*)
by(*auto intro*!:*observable-move-return*)
with ‹*length ms* = *length cs′*› ‹*length s* = *Suc* (*length cs*)› ‹*cs* = *c′* # *cs′*›
have *S*,*kind* ⊢ (*sourcenode a*#*targetnode a*#*ms*,*s*) =[]@[*a*]⇒
(*targetnode a*#*ms*,*transfer* (*kind a*) *s*)
by(*fastforce intro*:*observable-moves-snoc silent-moves-Nil*)
with ‹*S*,*kind* ⊢ (*targetnode a* # *ms*,*transfer* (*kind a*) *s*)
=*slice-edges S cs′ as*⇒∗ (*ms′′*,*s′′*)›
have *S*,*kind* ⊢ (*sourcenode a*#*targetnode a*#*ms*,*s*)
=*last* [*a*]#*slice-edges S cs′ as*⇒∗ (*ms′′*,*s′′*)
by −(*rule tom-Cons*,*auto*)
with ‹*slice-edges S cs* (*a*#*as*) = *a*#*slice-edges S cs′ as*›
have *S*,*kind* ⊢ (*sourcenode a*#*targetnode a*#*ms*,*s*) =*slice-edges S cs* (*a*#*as*)⇒∗

(*ms′′*,*s′′*) **by** *simp*
moreover
from ‹*slice-edges S cs′ as* = *slice-edges S cs′ as′′*›
‹*slice-edge S cs a*› ‹*kind a* = *Q*↪$_{pf}$› ‹*cs* = *c′* # *cs′*›
have *slice-edges S cs* (*a*#*as*) = *slice-edges S cs* (*a*#*as′′*) **by** *simp*
ultimately show *?thesis*
using *paths* ‹*m* = *sourcenode a*› ‹*valid-edge a*› ‹*kind a* = *Q*↪$_{pf}$›
‹*length ms* = *length cs′*› ‹*ms* = *targetnodes rs′*› ‹*cs* = *c′* # *cs′*›
‹*slice-edges S cs* (*a*#*as*) = *a*#*slice-edges S cs′ as*›
‹*a* ∈ *get-return-edges c′*›
‹*call-of-return-node* (*targetnode a*) (*sourcenode c′*)›
apply(*rule-tac x*=*targetnode a*#*ms* **in** *exI*)
apply(*rule-tac x*=*ms′′* **in** *exI*)
apply(*rule-tac x*=*s′′* **in** *exI*)
apply(*rule-tac x*=*as′* **in** *exI*)
apply(*rule-tac x*=*ms′* **in** *exI*)
apply(*rule-tac x*=*a*#*as′′* **in** *exI*)
apply(*auto intro*:*Cons-path simp*:*targetnodes-def*)
by(*case-tac i*) *auto*
next
case *False*
with ‹∀ *mx* ∈ *set ms*. *return-node mx*› ‹*return-node* (*targetnode a*)›
have ∃ *m* ∈ *set* (*targetnode a* # *ms*). ∃ *m′*. *call-of-return-node m m′* ∧
m′ ∉ ⌊*HRB-slice S*⌋$_{CFG}$
by(*fastforce dest*:*return-node-call-of-return-node*)
with ‹∀ *i*<*length cs′*. *call-of-return-node* (*ms* ! *i*) (*sourcenode* (*cs′* ! *i*))›
‹*length ms* = *length cs′*› ‹*cs* = *c′* # *cs′*› ‹*kind a* = *Q*↪$_{pf}$›
‹*call-of-return-node* (*targetnode a*) (*sourcenode c′*)›

have ¬ *slice-edge S cs a*
apply(*auto simp*:*slice-edge-def in-set-conv-nth*)
by(*erule-tac x*=*i* **in** *allE*) *auto*
with ‹*kind a* = *Q*↩*pf*› ‹*cs* = *c'* # *cs'*›
have *slice-edges S cs* (*a*#*as*) = *slice-edges S cs' as* **by** *simp*
from ‹*pred* (*kind a*) *s*› ‹*valid-edge a*› ‹*kind a* = *Q*↩*pf*›
‹∀ *mx* ∈ *set ms*. *return-node mx*› ‹*length ms* = *length cs'*›
‹*length s* = *Suc* (*length cs*)› ‹*m* = *sourcenode a*›
‹*length* (*transfer* (*kind a*) *s*) = *Suc* (*length cs'*)›
‹*return-node* (*targetnode a*)› ‹*cs* = *c'* # *cs'*›
‹∃ *m* ∈ *set* (*targetnode a* # *ms*). ∃ *m'*. *call-of-return-node m m'* ∧
m' ∉ ⌊*HRB-slice S*⌋*CFG*›
have *S*,*kind* ⊢ (*sourcenode a*#*targetnode a*#*ms*,*s*) −*a*→τ
(*targetnode a*#*ms*,*transfer* (*kind a*) *s*)
by(*auto intro*!:*silent-move-return*)
from ‹*S*,*kind* ⊢ (*targetnode a* # *ms*,*transfer* (*kind a*) *s*)
=*slice-edges S cs' as*⇒∗ (*ms''*,*s''*)›
show *?thesis*
proof(*rule trans-observable-moves.cases*)
fix *msx sx S' f'*
assume *targetnode a* # *ms* = *msx*
and *transfer* (*kind a*) *s* = *sx* **and** *slice-edges S cs' as* = []
and [*simp*]:*ms''* = *msx s''* = *sx* **and** *length msx* = *length sx*
from ‹*slice-edges S cs* (*a*#*as*) = *slice-edges S cs' as*›
‹*slice-edges S cs' as* = []›
have *slice-edges S cs* (*a*#*as*) = [] **by** *simp*
with ‹*length ms* = *length cs'*› ‹*length s* = *Suc* (*length cs*)› ‹*cs* = *c'* # *cs'*›
have *S*,*kind* ⊢ (*sourcenode a*#*targetnode a*#*ms*,*s*) =*slice-edges S cs* (*a*#*as*)⇒∗
(*sourcenode a*#*targetnode a*#*ms*,*s*)
by(*fastforce intro*:*tom-Nil*)
moreover
from ‹*S*,*kind* ⊢ (*ms''*,*s''*) =*as'*⇒τ (*m'*#*ms'*,*s'*)› ‹*targetnode a* # *ms* = *msx*›
‹*transfer* (*kind a*) *s* = *sx*› ‹*ms''* = *msx*› ‹*s''* = *sx*›
‹*S*,*kind* ⊢ (*sourcenode a*#*targetnode a*#*ms*,*s*) −*a*→τ
(*targetnode a*#*ms*,*transfer* (*kind a*) *s*)›
have *S*,*kind* ⊢ (*sourcenode a*#*targetnode a*#*ms*,*s*) =*a*#*as'*⇒τ (*m'*#*ms'*,*s'*)
by(*auto intro*:*silent-moves-Cons*)
from *this* ‹*valid-edge a*›
‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*valid-return-list rs m*› ‹*length rs* = *length cs*›
‹*length s* = *Suc* (*length cs*)› ‹*m* = *sourcenode a*›
‹*ms* = *targetnodes rs'*› ‹*rs* = *r'*#*rs'*› ‹*cs* = *c'* # *cs'*›
have *sourcenode a* −*a*#*as'*→∗ *m'* **and** *valid-path-aux cs* (*a*#*as'*)
by −(*rule silent-moves-vpa-path*,(*fastforce simp*:*targetnodes-def*)+)+
ultimately show *?thesis* **using** ‹*m* = *sourcenode a*› ‹*length ms* = *length cs'*›
‹∀ *i*<*length cs'*. *call-of-return-node* (*ms* ! *i*) (*sourcenode* (*cs'* ! *i*))›
‹*slice-edges S cs* (*a*#*as*) = []› ‹*kind a* = *Q*↩*pf*›

```
    ‹S,kind ⊢ (sourcenode a#targetnode a#ms,s) =a#as'⇒τ (m'#ms',s')›
    ‹ms = targetnodes rs'› ‹rs = r'#rs'› ‹cs = c' # cs'›
    ‹call-of-return-node (targetnode a) (sourcenode c')›
    apply(rule-tac x=targetnode a#ms in exI)
    apply(rule-tac x=sourcenode a#targetnode a#ms in exI)
    apply(rule-tac x=s in exI)
    apply(rule-tac x=a#as' in exI)
    apply(rule-tac x=ms' in exI)
    apply(rule-tac x=[] in exI)
    apply(auto simp:targetnodes-def)
    by(case-tac i) auto
next
  fix S' f' msx sx asx msx' sx' asx' msx'' sx''
  assume [simp]:S = S' and kind = f' and targetnode a # ms = msx
    and transfer (kind a) s = sx
    and slice-edges S cs' as = last asx # asx'
    and ms'' = msx'' and s'' = sx''
    and S',f' ⊢ (msx,sx) =asx⇒ (msx',sx')
    and S',f' ⊢ (msx',sx') =asx'⇒* (msx'',sx'')
  from ‹kind = f'› have [simp]:f' = kind by simp
  from ‹S,kind ⊢ (sourcenode a#targetnode a#ms,s) −a→τ
    (targetnode a#ms,transfer (kind a) s)›
    ‹S',f' ⊢ (msx,sx) =asx⇒ (msx',sx')›
    ‹transfer (kind a) s = sx› ‹targetnode a # ms = msx›
  have S,kind ⊢ (sourcenode a#targetnode a#ms,s) =a#asx⇒ (msx',sx')
    by(auto intro:silent-move-observable-moves)
  with ‹S',f' ⊢ (msx',sx') =asx'⇒* (msx'',sx'')› ‹ms'' = msx''› ‹s'' = sx''›
 have S,kind ⊢ (sourcenode a#targetnode a#ms,s) =last (a#asx)#asx'⇒*
    (ms'',s'')
    by(fastforce intro:trans-observable-moves.tom-Cons)
  moreover
  from ‹S',f' ⊢ (msx,sx) =asx⇒ (msx',sx')› have asx ≠ []
    by(fastforce elim:observable-moves.cases)
  with ‹slice-edges S cs (a#as) = slice-edges S cs' as›
    ‹slice-edges S cs' as = last asx # asx'›
  have slice-edges S cs (a#as) = last (a#asx)#asx' by simp
  moreover
  from ‹¬ slice-edge S cs a› ‹kind a = Q↩pf›
    ‹slice-edges S cs' as = slice-edges S cs' as''› ‹cs = c' # cs'›
  have slice-edges S cs (a # as) = slice-edges S cs (a # as'') by simp
 ultimately show ?thesis using paths ‹m = sourcenode a› ‹kind a = Q↩pf›
    ‹length ms = length cs'› ‹ms = targetnodes rs'› ‹valid-edge a›
    ‹rs = r'#rs'› ‹cs = c' # cs'› ‹r' ∈ get-return-edges c'›
    ‹call-of-return-node (targetnode a) (sourcenode c')›
    apply(rule-tac x=targetnode a#ms in exI)
    apply(rule-tac x=ms'' in exI)
    apply(rule-tac x=s'' in exI)
    apply(rule-tac x=as' in exI)
    apply(rule-tac x=ms' in exI)
```

```
      apply(rule-tac x=a#as'' in exI)
      apply(auto intro:Cons-path simp:targetnodes-def)
      by(case-tac i) auto
    qed
  qed
 qed
qed


lemma valid-path-trans-observable-moves:
 assumes m −as→√* m' and preds (kinds as) [cf]
 and transfers (kinds as) [cf] = s'
 obtains ms'' s'' ms' as' as''
 where S,kind ⊢ ([m],[cf]) =slice-edges S [] as⇒* (ms'',s'')
 and S,kind ⊢ (ms'',s'') =as'⇒τ (m'#ms',s')
 and slice-edges S [] as = slice-edges S [] as''
 and m −as''@as'→√* m'
proof(atomize-elim)
 from ‹m −as→√* m'› have valid-path-aux [] as and m −as→* m'
  by(simp-all add:vp-def valid-path-def)
 from this ‹preds (kinds as) [cf]› ‹transfers (kinds as) [cf] = s'›
 show ∃ms'' s'' as' ms' as''.
  S,kind ⊢ ([m],[cf]) =slice-edges S [] as⇒* (ms'',s'') ∧
  S,kind ⊢ (ms'',s'') =as'⇒τ (m' # ms',s') ∧
  slice-edges S [] as = slice-edges S [] as'' ∧ m −as'' @ as'→√* m'
  by −(erule vpa-trans-observable-moves[of - - - - - - [] S],
    auto simp:valid-call-list-def valid-return-list-def vp-def valid-path-def)
qed


lemma WS-weak-sim-trans:
 assumes ((ms1,s1),(ms2,s2)) ∈ WS S
 and S,kind ⊢ (ms1,s1) =as⇒* (ms1',s1') and as ≠ []
 shows ((ms1',s1'),(ms1',transfers (slice-kinds S as) s2)) ∈ WS S ∧
       S,slice-kind S ⊢ (ms2,s2) =as⇒* (ms1',transfers (slice-kinds S as) s2)
proof −
 obtain f where f = kind by simp
 with ‹S,kind ⊢ (ms1,s1) =as⇒* (ms1',s1')›
 have S,f ⊢ (ms1,s1) =as⇒* (ms1',s1') by simp
 from ‹S,f ⊢ (ms1,s1) =as⇒* (ms1',s1')› ‹((ms1,s1),(ms2,s2)) ∈ WS S›
  ‹as ≠ []› ‹f = kind›
 show ?thesis
 proof(induct arbitrary:ms2 s2 rule:trans-observable-moves.induct)
  case tom-Nil thus ?case by simp
 next
  case (tom-Cons S f ms s as ms' s' as' ms'' s'')
  note IH = ‹⋀ms2 s2. ⟦((ms',s'),(ms2,s2)) ∈ WS S; as' ≠ []; f = kind⟧
   ⟹ ((ms'',s''),(ms'',transfers (slice-kinds S as') s2)) ∈ WS S ∧
```

```
  S,slice-kind S ⊢ (ms₂,s₂) =as'⇒* (ms'',transfers (slice-kinds S as') s₂)⟩
from ⟨S,f ⊢ (ms,s) =as⇒ (ms',s')⟩ have s' ≠ []
  by(fastforce elim:observable-moves.cases observable-move.cases)
from ⟨S,f ⊢ (ms,s) =as⇒ (ms',s')⟩
obtain asx ax msx sx where S,f ⊢ (ms,s) =asx⇒τ (msx,sx)
  and S,f ⊢ (msx,sx) −ax→ (ms',s') and as = asx@[ax]
  by(fastforce elim:observable-moves.cases)
from ⟨S,f ⊢ (ms,s) =asx⇒τ (msx,sx)⟩ ⟨((ms,s),(ms₂,s₂)) ∈ WS S⟩ ⟨f = kind⟩
have ((msx,sx),(ms₂,s₂)) ∈ WS S by(fastforce intro:WS-silent-moves)
from ⟨((msx,sx),(ms₂,s₂)) ∈ WS S⟩ ⟨S,f ⊢ (msx,sx) −ax→ (ms',s')⟩ ⟨s' ≠ []⟩
  ⟨f = kind⟩
obtain asx' where ((ms',s'),(ms',transfer (slice-kind S ax) s₂)) ∈ WS S
  and S,slice-kind S ⊢ (ms₂,s₂) =asx'@[ax]⇒
  (ms',transfer (slice-kind S ax) s₂)
  by(fastforce elim:WS-observable-move)
show ?case
proof(cases as' = [])
  case True
  with ⟨S,f ⊢ (ms',s') =as'⇒* (ms'',s'')⟩ have ms' = ms'' ∧ s' = s''
 by(fastforce elim:trans-observable-moves.cases dest:observable-move-notempty)
  from ⟨((ms',s'),(ms',transfer (slice-kind S ax) s₂)) ∈ WS S⟩
  have length ms' = length (transfer (slice-kind S ax) s₂)
    by(fastforce elim:WS.cases)
  with ⟨S,slice-kind S ⊢ (ms₂,s₂) =asx'@[ax]⇒
    (ms',transfer (slice-kind S ax) s₂)⟩
  have S,slice-kind S ⊢ (ms₂,s₂) =(last (asx'@[ax]))#[]⇒*
    (ms',transfer (slice-kind S ax) s₂)
    by(fastforce intro:trans-observable-moves.intros)
  with ⟨((ms',s'),(ms',transfer (slice-kind S ax) s₂)) ∈ WS S⟩ ⟨as = asx@[ax]⟩
    ⟨ms' = ms'' ∧ s' = s''⟩ True
  show ?thesis by(fastforce simp:slice-kinds-def)
next
  case False
  from IH[OF ⟨((ms',s'),(ms',transfer (slice-kind S ax) s₂)) ∈ WS S⟩ this
    ⟨f = kind⟩]
  have ((ms'',s''),(ms'',transfers (slice-kinds S as')
    (transfer (slice-kind S ax) s₂))) ∈ WS S
    and S,slice-kind S ⊢ (ms',transfer (slice-kind S ax) s₂) =as'⇒*
    (ms'',transfers (slice-kinds S as') (transfer (slice-kind S ax) s₂))
    by simp-all
  with ⟨S,slice-kind S ⊢ (ms₂,s₂) =asx'@[ax]⇒
                        (ms',transfer (slice-kind S ax) s₂)⟩
  have S,slice-kind S ⊢ (ms₂,s₂) =(last (asx'@[ax]))#as'⇒*
    (ms'',transfers (slice-kinds S as') (transfer (slice-kind S ax) s₂))
    by(fastforce intro:trans-observable-moves.tom-Cons)
  with ⟨((ms'',s''),(ms'',transfers (slice-kinds S as')
    (transfer (slice-kind S ax) s₂))) ∈ WS S⟩ False ⟨as = asx@[ax]⟩
  show ?thesis by(fastforce simp:slice-kinds-def)
qed
```

qed
qed

lemma *stacks-rewrite*:
assumes *valid-call-list cs m* **and** *valid-return-list rs m*
and $\forall\, i < \mathit{length}\ rs.\ rs!i \in \mathit{get\text{-}return\text{-}edges}\ (cs!i)$
and *length rs* = *length cs* **and** *ms* = *targetnodes rs*
shows $\forall\, i{<}\mathit{length}\ cs.\ \mathit{call\text{-}of\text{-}return\text{-}node}\ (ms!i)\ (\mathit{sourcenode}\ (cs!i))$
proof
fix *i* **show** $i < \mathit{length}\ cs \longrightarrow$
call-of-return-node (*ms* ! *i*) (*sourcenode* (*cs* ! *i*))
proof
assume $i < \mathit{length}\ cs$
with ‹$\forall\, i < \mathit{length}\ rs.\ rs!i \in \mathit{get\text{-}return\text{-}edges}\ (cs!i)$› ‹*length rs* = *length cs*›
have $rs!i \in \mathit{get\text{-}return\text{-}edges}\ (cs!i)$ **by** *fastforce*
from ‹*valid-return-list rs m*› **have** $\forall\, r \in \mathit{set}\ rs.\ \mathit{valid\text{-}edge}\ r$
by(*rule valid-return-list-valid-edges*)
with ‹$i < \mathit{length}\ cs$› ‹*length rs* = *length cs*›
have *valid-edge* (*rs*!*i*) **by**(*simp add*:*all-set-conv-all-nth*)
from ‹*valid-call-list cs m*› **have** $\forall\, c \in \mathit{set}\ cs.\ \mathit{valid\text{-}edge}\ c$
by(*rule valid-call-list-valid-edges*)
with ‹$i < \mathit{length}\ cs$› **have** *valid-edge* (*cs*!*i*) **by**(*simp add*:*all-set-conv-all-nth*)
with ‹*valid-edge* (*rs*!*i*)› ‹*rs*!*i* ∈ *get-return-edges* (*cs*!*i*)› ‹*ms* = *targetnodes rs*›
‹$i < \mathit{length}\ cs$› ‹*length rs* = *length cs*›
show *call-of-return-node* (*ms* ! *i*) (*sourcenode* (*cs* ! *i*))
by(*fastforce simp*:*call-of-return-node-def return-node-def targetnodes-def*)
qed
qed

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** $\forall\, i < \mathit{length}\ rs.\ rs!i \in \mathit{get\text{-}return\text{-}edges}\ (cs!i)$
and *valid-return-list rs m* **and** *length rs* = *length cs* **and** *ms* = *targetnodes rs*
and $\forall\, mx \in \mathit{set}\ ms.\ \exists\, mx'.\ \mathit{call\text{-}of\text{-}return\text{-}node}\ mx\ mx' \wedge mx' \in \lfloor \mathit{HRB\text{-}slice}\ S \rfloor_{CFG}$
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 $\forall\, i < \mathit{length}\ rs'.\ rs'!i \in \mathit{get\text{-}return\text{-}edges}\ (cs'!i)$
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(*atomize-elim*)
from *assms* **show** $\exists\, as'\ cs'\ rs'.\ \mathit{preds}\ (\mathit{slice\text{-}kinds}\ S\ as')\ s \wedge$
slice-edges S cs as′ = *as* ∧ $m\ {-as'}{\rightarrow}*\ m'$ ∧ *valid-path-aux cs as′* ∧
upd-cs cs as′ = *cs′* ∧ *valid-node m′* ∧ *valid-call-list cs′ m′* ∧
$(\forall\, i{<}\mathit{length}\ rs'.\ rs'\ !\ i \in \mathit{get\text{-}return\text{-}edges}\ (cs'\ !\ i)) \wedge \mathit{valid\text{-}return\text{-}list}\ rs'\ m' \wedge$

length rs′ = length cs′ ∧ ms′ = targetnodes rs′ ∧
transfers (slice-kinds S as′) s ≠ [] ∧
transfers (slice-kinds S (slice-edges S cs as′)) s =
transfers (slice-kinds S as′) s
proof(*induct S slice-kind S m#ms s as m′#ms′ s′*
arbitrary:*m ms cs rs rule*:*trans-observable-moves.induct*)
case (*tom-Nil s* n_c)
from ‹*length* (*m′ # ms′*) = *length s*› **have** *s ≠ []* **by**(*cases s*) *auto*
have *preds* (*slice-kinds S []*) *s* **by**(*fastforce simp*:*slice-kinds-def*)
moreover
have *slice-edges S cs [] = []* **by** *simp*
moreover
from ‹*valid-node m′*› **have** *m′ −[]→∗ m′* **by**(*fastforce intro*:*empty-path*)
moreover
have *valid-path-aux cs []* **by** *simp*
moreover
have *upd-cs cs [] = cs* **by** *simp*
ultimately show *?case* **using** ‹*valid-call-list cs m′*› ‹*valid-return-list rs m′*›
‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)› ‹*length rs = length cs*›
‹*ms′ = targetnodes rs*› ‹*s ≠ []*› ‹*valid-node m′*›
apply(*rule-tac x*=[] **in** *exI*)
apply(*rule-tac x*=*cs* **in** *exI*)
apply(*rule-tac x*=*rs* **in** *exI*)
by(*clarsimp simp*:*slice-kinds-def*)
next
case (*tom-Cons S s as msx′ s′ as′ sx′*)
note *IH* = ‹⋀*m ms cs rs*. ⟦*msx′ = m # ms*; *valid-node m*; *valid-call-list cs m*;
∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*); *valid-return-list rs m*;
length rs = length cs; *ms = targetnodes rs*;
∀ *mx*∈*set ms*. ∃ *mx′*. *call-of-return-node mx mx′* ∧ *mx′* ∈ ⌊*HRB-slice S*⌋$_{CFG}$⟧
⟹ ∃ *as″ cs′ rs′*. *preds* (*slice-kinds S as″*) *s′* ∧
slice-edges S cs as″ = as′ ∧ *m −as″→∗ m′* ∧ *valid-path-aux cs as″* ∧
upd-cs cs as″ = cs′ ∧ *valid-node m′* ∧ *valid-call-list cs′ m′* ∧
(∀ *i*<*length rs′*. *rs′* ! *i* ∈ *get-return-edges* (*cs′* ! *i*)) ∧
valid-return-list rs′ m′ ∧ *length rs′ = length cs′* ∧ *ms′ = targetnodes rs′* ∧
transfers (*slice-kinds S as″*) *s′ ≠ []* ∧
transfers (*slice-kinds S* (*slice-edges S cs as″*)) *s′ =*
transfers (*slice-kinds S as″*) *s′*›
note *callstack* = ‹∀ *mx*∈*set ms*.
∃ *mx′*. *call-of-return-node mx mx′* ∧ *mx′* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
from ‹*S*,*slice-kind S* ⊢ (*m # ms*,*s*) =*as*⇒ (*msx′*,*s′*)›
obtain *asx ax xs s″* **where** *as = asx*@[*ax*]
and *S*,*slice-kind S* ⊢ (*m*#*ms*,*s*) =*asx*⇒$_\tau$ (*xs*,*s″*)
and *S*,*slice-kind S* ⊢ (*xs*,*s″*) −*ax*→ (*msx′*,*s′*)
by(*fastforce elim*:*observable-moves.cases*)
from ‹*S*,*slice-kind S* ⊢ (*xs*,*s″*) −*ax*→ (*msx′*,*s′*)›
obtain *xs′ ms″* **where** [*simp*]:*xs = sourcenode ax#xs′ msx′ = targetnode ax#ms″*
by(*cases xs*)(*auto elim*!:*observable-move.cases*,*case-tac list*,*auto*)

from ‹S,*slice-kind* $S \vdash (m \# ms,s) =as\Rightarrow (msx',s')$› *tom-Cons*
obtain cs'' rs'' **where** *results*:*valid-node* (*targetnode ax*)
valid-call-list cs'' (*targetnode ax*)
$\forall i < length\ rs''.\ rs''!i \in$ *get-return-edges* ($cs''!i$)
valid-return-list rs'' (*targetnode ax*) *length* $rs'' =$ *length* cs''
$ms'' =$ *targetnodes* rs'' **and** *upd-cs cs as* $=$ cs''
by(*auto elim*!:*observable-moves-preserves-stack*)
from ‹S,*slice-kind* $S \vdash (m\#ms,s) =asx\Rightarrow_\tau (xs,s'')$› *callstack*
have $\forall a \in set\ asx.$ *intra-kind* (*kind a*)
by *simp*(*rule silent-moves-slice-intra-path*)
with ‹S,*slice-kind* $S \vdash (m\#ms,s) =asx\Rightarrow_\tau (xs,s'')$›
have [*simp*]:$xs' = ms$ **by**(*fastforce dest*:*silent-moves-intra-path*)
from ‹S,*slice-kind* $S \vdash (xs,s'') -ax\rightarrow (msx',s')$›
have $\forall mx \in set\ ms''.\ \exists mx'.$ *call-of-return-node* $mx\ mx' \wedge mx' \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$
by(*fastforce dest*:*observable-set-stack-in-slice*)
from *IH*[*OF* ‹$msx' =$ *targetnode* $ax\#ms''$› *results this*]
obtain asx' cs' rs' **where** *preds* (*slice-kinds S* asx') s'
and *slice-edges S* cs'' $asx' = as'$ **and** *targetnode ax* $-asx'\rightarrow* m'$
and *valid-path-aux* cs'' asx' **and** *upd-cs* cs'' $asx' = cs'$
and *valid-node* m' **and** *valid-call-list* cs' m'
and $\forall i<$*length* $rs'.\ rs'\ !\ i \in$ *get-return-edges* ($cs'\ !\ i$)
and *valid-return-list* rs' m' **and** *length* $rs' =$ *length* cs'
and $ms' =$ *targetnodes* rs' **and** *transfers* (*slice-kinds S* asx') $s' \neq []$
and *trans-eq*:*transfers* (*slice-kinds S* (*slice-edges S* cs'' asx')) $s' =$
transfers (*slice-kinds S* asx') s'
by *blast*
from ‹S,*slice-kind* $S \vdash (m\#ms,s) =asx\Rightarrow_\tau (xs,s'')$›
have *preds* (*slice-kinds S asx*) s **and** *transfers* (*slice-kinds S asx*) $s = s''$
by(*auto intro*:*silent-moves-preds-transfers simp*:*slice-kinds-def*)
from ‹S,*slice-kind* $S \vdash (xs,s'') -ax\rightarrow (msx',s')$›
have *pred* (*slice-kind S ax*) s'' **and** *transfer* (*slice-kind S ax*) $s'' = s'$
by(*auto elim*:*observable-move.cases*)
with ‹*preds* (*slice-kinds S asx*) s› ‹$as = asx@[ax]$›
‹*transfers* (*slice-kinds S asx*) $s = s''$›
have *preds* (*slice-kinds S as*) s **by**(*simp add*:*preds-split slice-kinds-def*)
from ‹*transfers* (*slice-kinds S asx*) $s = s''$›
‹*transfer* (*slice-kind S ax*) $s'' = s'$› ‹$as = asx@[ax]$›
have *transfers* (*slice-kinds S as*) $s = s'$
by(*simp add*:*transfers-split slice-kinds-def*)
with ‹*preds* (*slice-kinds S* asx') s'› ‹*preds* (*slice-kinds S as*) s›
have *preds* (*slice-kinds S* ($as@asx'$)) s **by**(*simp add*:*preds-split slice-kinds-def*)
moreover
from ‹*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*›
‹$ms =$ *targetnodes rs*›
have $\forall i<$*length cs*. *call-of-return-node* ($ms!i$) (*sourcenode* ($cs!i$))
by(*rule stacks-rewrite*)
with ‹S,*slice-kind* $S \vdash (m \# ms,s) =as\Rightarrow (msx',s')$› ‹$ms =$ *targetnodes rs*›

⟨*length rs = length cs*⟩
have *slice-edges S cs as = [last as]*
by(*fastforce elim*:*observable-moves-singular-slice-edge*)
with ⟨*slice-edges S cs'' asx' = as'*⟩ ⟨*upd-cs cs as = cs''*⟩
have *slice-edges S cs (as@asx') = [last as]@as'*
by(*fastforce intro*:*slice-edges-Append*)
moreover
from ⟨*S*,*slice-kind S* ⊢ (*m*#*ms*,*s*) =*asx*⇒$_\tau$ (*xs*,*s''*)⟩ ⟨*valid-node m*⟩
⟨*valid-call-list cs m*⟩ ⟨∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)⟩
⟨*valid-return-list rs m*⟩ ⟨*length rs = length cs*⟩ ⟨*ms = targetnodes rs*⟩
have *m* −*asx*→∗ *sourcenode ax* **by**(*fastforce intro*:*silent-moves-vpa-path*)
from ⟨*S*,*slice-kind S* ⊢ (*xs*,*s''*) −*ax*→ (*msx'*,*s'*)⟩ **have** *valid-edge ax*
by(*fastforce elim*:*observable-move.cases*)
hence *sourcenode ax* −[*ax*]→∗ *targetnode ax* **by**(*rule path-edge*)
with ⟨*m* −*asx*→∗ *sourcenode ax*⟩ ⟨*as = asx*@[*ax*]⟩
have *m* −*as*→∗ *targetnode ax* **by**(*fastforce intro*:*path-Append*)
with ⟨*targetnode ax* −*asx'*→∗ *m'*⟩ **have** *m* −*as*@*asx'*→∗ *m'*
by −(*rule path-Append*)
moreover
from ⟨∀ *a* ∈ *set asx*. *intra-kind* (*kind a*)⟩ **have** *valid-path-aux cs asx*
by(*rule valid-path-aux-intra-path*)
from ⟨∀ *a* ∈ *set asx*. *intra-kind* (*kind a*)⟩ **have** *upd-cs cs asx = cs*
by(*rule upd-cs-intra-path*)
from ⟨*m* −*asx*→∗ *sourcenode ax*⟩ ⟨∀ *a* ∈ *set asx*. *intra-kind* (*kind a*)⟩
have *get-proc m = get-proc* (*sourcenode ax*)
by(*fastforce intro*:*intra-path-get-procs simp*:*intra-path-def*)
with ⟨*valid-return-list rs m*⟩ **have** *valid-return-list rs* (*sourcenode ax*)
apply(*clarsimp simp*:*valid-return-list-def*)
apply(*erule-tac x*=*cs'* **in** *allE*) **apply** *clarsimp*
by(*case-tac cs'*) *auto*
with ⟨*S*,*slice-kind S* ⊢ (*xs*,*s''*) −*ax*→ (*msx'*,*s'*)⟩ ⟨*valid-edge ax*⟩
⟨∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)⟩ ⟨*ms = targetnodes rs*⟩
⟨*length rs = length cs*⟩
have *valid-path-aux cs* [*ax*]
by(*auto intro*!:*observable-move-vpa-path simp del*:*valid-path-aux.simps*)
with ⟨*valid-path-aux cs asx*⟩ ⟨*upd-cs cs asx = cs*⟩ ⟨*as = asx*@[*ax*]⟩
have *valid-path-aux cs as* **by**(*fastforce intro*:*valid-path-aux-Append*)
with ⟨*upd-cs cs as = cs''*⟩ ⟨*valid-path-aux cs'' asx'*⟩
have *valid-path-aux cs* (*as*@*asx'*) **by**(*fastforce intro*:*valid-path-aux-Append*)
moreover
from ⟨*upd-cs cs as = cs''*⟩ ⟨*upd-cs cs'' asx' = cs'*⟩
have *upd-cs cs* (*as*@*asx'*) = *cs'* **by**(*rule upd-cs-Append*)
moreover
from ⟨*transfers* (*slice-kinds S as*) *s* = *s'*⟩
⟨*transfers* (*slice-kinds S asx'*) *s'* ≠ []⟩
have *transfers* (*slice-kinds S* (*as*@*asx'*)) *s* ≠ []
by(*simp add*:*slice-kinds-def transfers-split*)
moreover
from ⟨*S*,*slice-kind S* ⊢ (*m* # *ms*,*s*) =*as*⇒ (*msx'*,*s'*)⟩

have *transfers* (*map* (*slice-kind S*) *as*) *s* = *s'*
by *simp*(*rule observable-moves-preds-transfers*)
from ‹*S*,*slice-kind S* ⊢ (*m* # *ms*,*s*) =*as*⇒ (*msx'*,*s'*)› ‹*ms* = *targetnodes rs*›
‹*length rs* = *length cs*› ‹∀ *i*<*length rs*. *rs* ! *i* ∈ *get-return-edges* (*cs* ! *i*)›
‹*valid-call-list cs m*› ‹*valid-return-list rs m*›
have *slice-edges S cs as* = [*last as*]
by(*fastforce intro*!:*observable-moves-singular-slice-edge*
[*OF - - - stacks-rewrite*])
from ‹*S*,*slice-kind S* ⊢ (*m*#*ms*,*s*) =*asx*⇒$_\tau$ (*xs*,*s''*)› *callstack*
have *s* = *s''* **by**(*fastforce intro*:*silent-moves-slice-keeps-state*)
with ‹*S*,*slice-kind S* ⊢ (*xs*,*s''*) −*ax*→ (*msx'*,*s'*)›
have *transfer* (*slice-kind S ax*) *s* = *s'* **by**(*fastforce elim*:*observable-move.cases*)
with ‹*slice-edges S cs as* = [*last as*]› ‹*as* = *asx*@[*ax*]›
have *s'* = *transfers* (*slice-kinds S* (*slice-edges S cs as*)) *s*
by(*simp add*:*slice-kinds-def*)
from ‹*upd-cs cs as* = *cs''*›
have *slice-edges S cs* (*as* @ *asx'*) =
(*slice-edges S cs as*)@(*slice-edges S cs''* *asx'*)
by(*fastforce intro*:*slice-edges-Append*)
hence *trans-eq'*:*transfers* (*slice-kinds S* (*slice-edges S cs* (*as* @ *asx'*))) *s* =
transfers (*slice-kinds S* (*slice-edges S cs''* *asx'*))
(*transfers* (*slice-kinds S* (*slice-edges S cs as*)) *s*)
by(*simp add*:*slice-kinds-def transfers-split*)
from ‹*s'* = *transfers* (*slice-kinds S* (*slice-edges S cs as*)) *s*›
‹*transfers* (*map* (*slice-kind S*) *as*) *s* = *s'*›
have *transfers* (*map* (*slice-kind S*) (*slice-edges S cs as*)) *s* =
transfers (*map* (*slice-kind S*) *as*) *s*
by(*simp add*:*slice-kinds-def*)
with *trans-eq trans-eq'*
‹*s'* = *transfers* (*slice-kinds S* (*slice-edges S cs as*)) *s*›
have *transfers* (*slice-kinds S* (*slice-edges S cs* (*as* @ *asx'*))) *s* =
transfers (*slice-kinds S* (*as* @ *asx'*)) *s*
by(*simp add*:*slice-kinds-def transfers-split*)
ultimately show *?case*
using ‹*valid-node m'*› ‹*valid-call-list cs' m'*›
‹∀ *i*<*length rs'*. *rs'* ! *i* ∈ *get-return-edges* (*cs'* ! *i*)›
‹*valid-return-list rs' m'*› ‹*length rs'* = *length cs'*› ‹*ms'* = *targetnodes rs'*›
apply(*rule-tac x*=*as*@*asx'* **in** *exI*)
apply(*rule-tac x*=*cs'* **in** *exI*)
apply(*rule-tac x*=*rs'* **in** *exI*)
by *clarsimp*
qed
qed

1.14.4 The fundamental property of static interprocedural slicing

theorem *fundamental-property-of-static-slicing*:
assumes *m* −*as*→$_{\surd}$* *m'* **and** *preds* (*kinds as*) [*cf*] **and** *CFG-node m'* ∈ *S*

```
  obtains as' where preds (slice-kinds S as') [cf]
  and ∀ V ∈ 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] ≠ [] and m −as'→√* m'
proof(atomize-elim)
  from ‹m −as→√* m'› ‹preds (kinds as) [cf]› obtain ms'' s'' ms' as' as''
    where S,kind ⊢ ([m],[cf]) =slice-edges S [] as⇒*
                               (ms'',s'')
    and S,kind ⊢ (ms'',s'') =as'⇒τ (m'#ms',transfers (kinds as) [cf])
    and slice-edges S [] as = slice-edges S [] as''
    and m −as''@as'→√* m'
    by(auto elim:valid-path-trans-observable-moves[of - - - - - S])
  from ‹m −as→√* m'› have valid-node m and valid-node m'
    by(auto intro:path-valid-node simp:vp-def)
  with ‹CFG-node m' ∈ S› have CFG-node m' ∈ HRB-slice S
    by −(rule HRB-slice-refl)
  from ‹valid-node m› ‹CFG-node m' ∈ S› have (([m],[cf]),([m],[cf])) ∈ WS S
    by(fastforce intro:WSI)
  { fix V assume V ∈ Use m'
    with ‹valid-node m'› have V ∈ Use_SDG (CFG-node m')
      by(fastforce intro:CFG-Use-SDG-Use)
    moreover
    from ‹valid-node m'›
    have parent-node (CFG-node m') −[]→ι* parent-node (CFG-node m')
      by(fastforce intro:empty-path simp:intra-path-def)
    ultimately have V ∈ rv S (CFG-node m')
      using ‹CFG-node m' ∈ HRB-slice S› ‹CFG-node m' ∈ S›
      by(fastforce intro:rvI simp:sourcenodes-def) }
  hence ∀ V ∈ Use m'. V ∈ rv S (CFG-node m') by simp
  show ∃ as'. preds (slice-kinds S as') [cf] ∧
    (∀ V∈Use m'. state-val (transfers (slice-kinds S as') [cf]) V =
    state-val (transfers (kinds as) [cf]) V) ∧
    slice-edges S [] as = slice-edges S [] as' ∧
     transfers (kinds as) [cf] ≠ [] ∧ m −as'→√* m'
  proof(cases slice-edges S [] as = [])
    case True
    hence preds (slice-kinds S []) [cf]
      and slice-edges S [] [] = slice-edges S [] as
      by(simp-all add:slice-kinds-def)
    with ‹S,kind ⊢ ([m],[cf]) =slice-edges S [] as⇒* (ms'',s'')›
    have [simp]:ms'' = [m] s'' = [cf] by(auto elim:trans-observable-moves.cases)
    with ‹S,kind ⊢ (ms'',s'') =as'⇒τ (m'#ms',transfers (kinds as) [cf])›
    have S,kind ⊢ ([m],[cf]) =as'⇒τ (m'#ms',transfers (kinds as) [cf])
      by simp
    with ‹valid-node m› have m −as'→* m' and valid-path-aux [] as'
      by(auto intro:silent-moves-vpa-path[of - - - - - - - - - []]
              simp:targetnodes-def valid-return-list-def)
    hence m −as'→√* m' by(simp add:vp-def valid-path-def)
```

from ‹S,$kind \vdash ([m],[cf]) =as'\Rightarrow_\tau (m'\#ms',transfers\ (kinds\ as)\ [cf])$›
have *slice-edges* $S\ []\ as' = []$
by(*fastforce dest*:*silent-moves-no-slice-edges*[**where** *cs*=[] **and** *rs*=[]]
simp:*targetnodes-def*)
from ‹S,$kind \vdash ([m],[cf]) =as'\Rightarrow_\tau (m'\#ms',transfers\ (kinds\ as)\ [cf])$›
‹*valid-node m*› ‹*valid-node m'*› ‹*CFG-node* $m' \in S$›
have *returns*:$\forall\, mx \in set\ ms'.$
$\exists\, mx'.$ *call-of-return-node mx* $mx' \wedge mx' \in \lfloor$*HRB-slice* $S\rfloor_{CFG}$
by −(*erule silent-moves-called-node-in-slice1-nodestack-in-slice1*
[*of* - - - - - - - - - [] []],
auto intro:*refl-slice1 simp*:*targetnodes-def valid-return-list-def*)
from ‹S,$kind \vdash ([m],[cf]) =as'\Rightarrow_\tau (m'\#ms',transfers\ (kinds\ as)\ [cf])$›
‹$(([m],[cf]),([m],[cf])) \in WS\ S$›
have WS:$((m'\#ms',transfers\ (kinds\ as)\ [cf]),([m],[cf])) \in WS\ S$
by(*rule WS-silent-moves*)
hence $transfers\ (kinds\ as)\ [cf] \neq []$ **by**(*auto elim*!:*WS.cases*)
with *WS returns* ‹$transfers\ (kinds\ as)\ [cf] \neq []$›
have $\forall\, V \in rv\ S\ (CFG\text{-}node\ m').$
state-val $(transfers\ (kinds\ as)\ [cf])\ V = fst\ cf\ V$
apply − **apply**(*erule WS.cases*) **apply** *clarsimp*
by(*case-tac msx*)(*auto simp*:*hd-conv-nth*)
with ‹$\forall\, V \in Use\ m'.\ V \in rv\ S\ (CFG\text{-}node\ m')$›
have *Uses*:$\forall\, V \in Use\ m'.$ *state-val* $(transfers\ (kinds\ as)\ [cf])\ V = fst\ cf\ V$
by *simp*
have [*simp*]:$ms' = []$
proof(*rule ccontr*)
assume $ms' \neq []$
with ‹S,$kind \vdash ([m],[cf]) =as'\Rightarrow_\tau (m'\#ms',transfers\ (kinds\ as)\ [cf])$›
‹*valid-node m*› ‹*valid-node m'*› ‹*CFG-node* $m' \in S$›
show *False*
by(*fastforce elim*:*silent-moves-nonempty-nodestack-False intro*:*refl-slice1*)
qed
with ‹S,$kind \vdash ([m],[cf]) =as'\Rightarrow_\tau (m'\#ms',transfers\ (kinds\ as)\ [cf])$›
have S,$kind \vdash ([m],[cf]) =as'\Rightarrow_\tau ([m'],transfers\ (kinds\ as)\ [cf])$
by *simp*
with ‹*valid-node m*› **have** $m\ -as'\rightarrow_{sl}*\ m'$ **by**(*fastforce dest*:*silent-moves-slp*)
from *this* ‹*slice-edges* $S\ []\ as' = []$›
obtain *asx* **where** $m\ -asx\rightarrow_\iota *\ m'$ **and** *slice-edges* $S\ []\ asx = []$
by(*erule slp-to-intra-path-with-slice-edges*)
with ‹*CFG-node* $m' \in$ *HRB-slice* S›
obtain *asx'* **where** $m\ -asx'\rightarrow_\iota *\ m'$
and *preds* (*slice-kinds S asx'*) [*cf*]
and *slice-edges* $S\ []\ asx' = []$
by −(*erule exists-sliced-intra-path-preds*,*auto simp*:*SDG-to-CFG-set-def*)
from ‹$m\ -asx'\rightarrow_\iota *\ m'$› **have** $m\ -asx'\rightarrow_\surd *\ m'$ **by**(*rule intra-path-vp*)
from *Uses* ‹*slice-edges* $S\ []\ asx' = []$›
have *hd* (*transfers* (*slice-kinds S*
(*slice-edges* $S\ []\ asx'$)) [*cf*]) = *cf* **by**(*simp add*:*slice-kinds-def*)
from ‹$m\ -asx'\rightarrow_\iota *\ m'$› ‹*CFG-node* $m' \in S$›

have *transfers* (*slice-kinds S* (*slice-edges S* [] *asx'*)) [*cf*] =
transfers (*slice-kinds S asx'*) [*cf*]
by(*fastforce intro*:*transfers-intra-slice-kinds-slice-edges simp*:*intra-path-def*)
with ‹*hd* (*transfers* (*slice-kinds S* (*slice-edges S* [] *asx'*)) [*cf*]) = *cf*›
have *hd* (*transfers* (*slice-kinds S asx'*) [*cf*]) = *cf* **by** *simp*
with *Uses* **have** ∀ *V*∈*Use m'*. *state-val* (*transfers* (*slice-kinds S asx'*) [*cf*]) *V*
=
state-val (*transfers* (*kinds as*) [*cf*]) *V* **by** *simp*
with ‹*m* −*asx'*→$_{\surd}$* *m'*› ‹*preds* (*slice-kinds S asx'*) [*cf*]›
‹*slice-edges S* [] *asx'* = []› ‹*transfers* (*kinds as*) [*cf*] ≠ []› *True*
show *?thesis* **by** *fastforce*
next
case *False*
with ‹(([*m*],[*cf*]),([*m*],[*cf*])) ∈ *WS S*›
‹*S*,*kind* ⊢ ([*m*],[*cf*]) =*slice-edges S* [] *as*⇒* (*ms''*,*s''*)›
have *WS*:((*ms''*,*s''*),(*ms''*,*transfers* (*slice-kinds S* (*slice-edges S* [] *as*)) [*cf*]))
∈ *WS S*
and *tom*:*S*,*slice-kind S* ⊢ ([*m*],[*cf*]) =*slice-edges S* [] *as*⇒*
(*ms''*,*transfers* (*slice-kinds S* (*slice-edges S* [] *as*)) [*cf*])
by(*fastforce dest*:*WS-weak-sim-trans*)+
from *WS* **obtain** *mx msx* **where** [*simp*]:*ms''* = *mx*#*msx* **and** *valid-node mx*
by −(*erule WS.cases*,*cases ms''*,*auto*)
from ‹*S*,*kind* ⊢ (*ms''*,*s''*) =*as'*⇒$_{\tau}$ (*m'*#*ms'*,*transfers* (*kinds as*) [*cf*])› *WS*
have *WS'*:((*m'*#*ms'*,*transfers* (*kinds as*) [*cf*]),
(*mx*#*msx*,*transfers* (*slice-kinds S* (*slice-edges S* [] *as*)) [*cf*])) ∈ *WS S*
by *simp*(*rule WS-silent-moves*)
from *tom* ‹*valid-node m*›
obtain *asx csx rsx* **where** *preds* (*slice-kinds S asx*) [*cf*]
and *slice-edges S* [] *asx* = *slice-edges S* [] *as*
and *m* −*asx*→$_{\surd}$* *mx* **and** *transfers* (*slice-kinds S asx*) [*cf*] ≠ []
and *upd-cs* [] *asx* = *csx* **and** *stack*:*valid-node mx valid-call-list csx mx*
∀ *i* < *length rsx*. *rsx*!*i* ∈ *get-return-edges* (*csx*!*i*)
valid-return-list rsx mx length rsx = *length csx*
msx = *targetnodes rsx*
and *trans-eq*:*transfers* (*slice-kinds S*
(*slice-edges S* [] *asx*)) [*cf*] =
transfers (*slice-kinds S asx*) [*cf*]
by(*auto elim*:*slice-tom-preds-vp*[*of* - - - - - - - - [] []]
simp:*valid-call-list-def valid-return-list-def targetnodes-def*
vp-def valid-path-def)
from ‹*transfers* (*slice-kinds S asx*) [*cf*] ≠ []›
obtain *cf' cfs'* **where** *eq*:*transfers* (*slice-kinds S asx*) [*cf*] =
cf'#*cfs'* **by**(*cases transfers* (*slice-kinds S asx*) [*cf*]) *auto*
from *WS'* **have** *callstack*:∀ *mx* ∈ *set msx*. ∃ *mx'*. *call-of-return-node mx mx'* ∧

mx' ∈ ⌊*HRB-slice S*⌋$_{CFG}$
by(*fastforce elim*:*WS.cases*)
with ‹*S*,*kind* ⊢ (*ms''*,*s''*) =*as'*⇒$_{\tau}$ (*m'*#*ms'*,*transfers* (*kinds as*) [*cf*])›
‹*valid-node m'*› *stack* ‹*CFG-node m'* ∈ *S*›

have *callstack'*:$\forall\, mx \in set\ ms'.\ \exists\, mx'.$ *call-of-return-node mx mx'* $\wedge$
mx' $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$
by *simp*(*erule silent-moves-called-node-in-slice1-nodestack-in-slice1*
[*of - - - - - - - - - - rsx csx*],*auto intro*:*refl-slice1*)
with ‹*S*,*kind* $\vdash$ (*ms''*,*s''*) =*as'*$\Rightarrow_{\tau}$ (*m'*#*ms'*,*transfers* (*kinds as*) [*cf*])›
stack callstack
have *mx* $-$*as'*$\rightarrow_{sl}*$ *m'* **and** *msx* = *ms'* **by**(*auto dest*!:*silent-moves-slp*)
from ‹*S*,*kind* $\vdash$ (*ms''*,*s''*) =*as'*$\Rightarrow_{\tau}$ (*m'*#*ms'*,*transfers* (*kinds as*) [*cf*])›
stack
have *slice-edges S csx as'* = []
by(*auto dest*:*silent-moves-no-slice-edges*[*OF - - - stacks-rewrite*])
with ‹*mx* $-$*as'*$\rightarrow_{sl}*$ *m'*› **obtain** *asx''* **where** *mx* $-$*asx''*$\rightarrow_{\iota}*$ *m'*
and *slice-edges S csx asx''* = []
by(*erule slp-to-intra-path-with-slice-edges*)
from *stack* **have** $\forall\, i$<*length csx*. *call-of-return-node* (*msx*!*i*) (*sourcenode* (*csx*!*i*))
by $-$(*rule stacks-rewrite*)
with *callstack* ‹*msx* = *targetnodes rsx*› ‹*length rsx* = *length csx*›
have $\forall\, c \in$*set csx*. *sourcenode c* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$
by(*auto simp*:*all-set-conv-all-nth targetnodes-def*)
with ‹*mx* $-$*asx''*$\rightarrow_{\iota}*$ *m'*› ‹*slice-edges S csx asx''* = []› ‹*valid-node m'*›
eq ‹*CFG-node m'* $\in$ *S*›
obtain *asx'* **where** *mx* $-$*asx'*$\rightarrow_{\iota}*$ *m'*
and *preds* (*slice-kinds S asx'*) (*cf'*#*cfs'*)
and *slice-edges S csx asx'* = []
by $-$(*erule exists-sliced-intra-path-preds*,
auto intro:*HRB-slice-refl simp*:*SDG-to-CFG-set-def*)
with *eq* **have** *preds* (*slice-kinds S asx'*)
(*transfers* (*slice-kinds S asx*) [*cf*]) **by** *simp*
with ‹*preds* (*slice-kinds S asx*) [*cf*]›
have *preds* (*slice-kinds S* (*asx*@*asx'*)) [*cf*]
by(*simp add*:*slice-kinds-def preds-split*)
from ‹*m* $-$*asx*$\rightarrow_{\surd}*$ *mx*› ‹*mx* $-$*asx'*$\rightarrow_{\iota}*$ *m'*› **have** *m* $-$*asx*@*asx'*$\rightarrow_{\surd}*$ *m'*
by(*fastforce elim*:*vp-slp-Append intra-path-slp*)
from ‹*upd-cs* [] *asx* = *csx*› ‹*slice-edges S csx asx'* = []›
have *slice-edges S* [] (*asx*@*asx'*) =
(*slice-edges S* [] *asx*)@[]
by(*fastforce intro*:*slice-edges-Append*)
from ‹*mx* $-$*asx'*$\rightarrow_{\iota}*$ *m'*› ‹$\forall\, c \in$*set csx*. *sourcenode c* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$›
have *trans-eq'*:*transfers* (*slice-kinds S* (*slice-edges S csx asx'*))
(*transfers* (*slice-kinds S asx*) [*cf*]) =
transfers (*slice-kinds S asx'*) (*transfers* (*slice-kinds S asx*) [*cf*])
by(*fastforce intro*:*transfers-intra-slice-kinds-slice-edges simp*:*intra-path-def*)
from ‹*upd-cs* [] *asx* = *csx*›
have *slice-edges S* [] (*asx*@*asx'*) =
(*slice-edges S* [] *asx*)@(*slice-edges S csx asx'*)
by(*fastforce intro*:*slice-edges-Append*)
hence *transfers* (*slice-kinds S* (*slice-edges S* [] (*asx*@*asx'*))) [*cf*] =
transfers (*slice-kinds S* (*slice-edges S csx asx'*))
(*transfers* (*slice-kinds S* (*slice-edges S* [] *asx*)) [*cf*])

by(*simp add*:*slice-kinds-def transfers-split*)
with *trans-eq* **have** *transfers* (*slice-kinds S* (*slice-edges S* [] (*asx*@*asx'*))) [*cf*]
=
transfers (*slice-kinds S* (*slice-edges S csx asx'*))
(*transfers* (*slice-kinds S asx*) [*cf*]) **by** *simp*
with *trans-eq'* **have** *trans-eq''*:
transfers (*slice-kinds S* (*slice-edges S* [] (*asx*@*asx'*))) [*cf*] =
transfers (*slice-kinds S* (*asx*@*asx'*)) [*cf*]
by(*simp add*:*slice-kinds-def transfers-split*)
from *WS'* **obtain** *x xs* **where** *m'*#*ms'* = *xs*@*x*#*msx*
and *xs* ≠ [] ⟶ (∃ *mx'*. *call-of-return-node x mx'* ∧
mx' ∉ ⌊*HRB-slice S*⌋$_{CFG}$)
and *rest*:∀ *i* < *length* (*mx*#*msx*). ∀ *V* ∈ *rv S* (*CFG-node* ((*x*#*msx*)!*i*)).
(*fst* ((*transfers* (*kinds as*) [*cf*])!(*length xs* + *i*))) *V* =
(*fst* ((*transfers* (*slice-kinds S*
(*slice-edges S* [] *as*)) [*cf*])!*i*)) *V*
transfers (*kinds as*) [*cf*] ≠ []
transfers (*slice-kinds S*
(*slice-edges S* [] *as*)) [*cf*] ≠ []
by(*fastforce elim*:*WS.cases*)
from ‹*m'*#*ms'* = *xs*@*x*#*msx*› ‹*xs* ≠ [] ⟶ (∃ *mx'*. *call-of-return-node x mx'* ∧
mx' ∉ ⌊*HRB-slice S*⌋$_{CFG}$)› *callstack'*
have [*simp*]:*xs* = [] *x* = *m'* *ms'* = *msx* **by**(*cases xs*,*auto*)+
from *rest* **have** ∀ *V* ∈ *rv S* (*CFG-node m'*).
state-val (*transfers* (*kinds as*) [*cf*]) *V* =
state-val (*transfers* (*slice-kinds S* (*slice-edges S* [] *as*)) [*cf*]) *V*
by(*fastforce dest*:*hd-conv-nth*)
with ‹∀ *V* ∈ *Use m'*. *V* ∈ *rv S* (*CFG-node m'*)›
‹*slice-edges S* [] *asx* = *slice-edges S* [] *as*›
have ∀ *V* ∈ *Use m'*. *state-val* (*transfers* (*kinds as*) [*cf*]) *V* =
state-val (*transfers* (*slice-kinds S* (*slice-edges S* [] *asx*)) [*cf*]) *V*
by *simp*
with ‹*slice-edges S* [] (*asx*@*asx'*) = (*slice-edges S* [] *asx*)@[]›
have ∀ *V* ∈ *Use m'*. *state-val* (*transfers* (*kinds as*) [*cf*]) *V* =
state-val (*transfers* (*slice-kinds S* (*slice-edges S* [] (*asx*@*asx'*))) [*cf*]) *V*
by *simp*
with *trans-eq''* **have** ∀ *V* ∈ *Use m'*. *state-val* (*transfers* (*kinds as*) [*cf*]) *V* =
state-val (*transfers* (*slice-kinds S* (*asx*@*asx'*)) [*cf*]) *V*
by *simp*
with ‹*preds* (*slice-kinds S* (*asx*@*asx'*)) [*cf*]›
‹*m* −*asx*@*asx'*→$_{\surd}$* *m'*› ‹*slice-edges S* [] (*asx*@*asx'*) =
(*slice-edges S* [] *asx*)@[]› ‹*transfers* (*kinds as*) [*cf*] ≠ []›
‹*slice-edges S* [] *asx* = *slice-edges S* [] *as*›
show *?thesis* **by** *fastforce*
qed
qed

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** $\langle c,[cf]\rangle \Rightarrow \langle c',s'\rangle$
obtains m' *as cfs'* **where** $m\ -as\rightarrow_{\surd}*\ m'$ **and** $m' \triangleq c'$
and *preds* (*slice-kinds* {*CFG-node m'*} *as*) [(*cf*,*undefined*)]
and ∀ $V \in$ *Use m'*.
state-val (*transfers* (*slice-kinds* {*CFG-node m'*} *as*) [(*cf*,*undefined*)]) V =
state-val cfs' V **and** *map fst cfs'* = s'
proof(*atomize-elim*)
from ‹$m \triangleq c$› ‹$\langle c,[cf]\rangle \Rightarrow \langle c',s'\rangle$› **obtain** m' *as cfs'* **where** $m\ -as\rightarrow_{\surd}*\ m'$
and *transfers* (*kinds as*) [(*cf*,*undefined*)] = *cfs'*
and *preds* (*kinds as*) [(*cf*,*undefined*)] **and** $m' \triangleq c'$ **and** *map fst cfs'* = s'
by(*fastforce dest*:*fundamental-property*)
from ‹$m\ -as\rightarrow_{\surd}*\ m'$› ‹*preds* (*kinds as*) [(*cf*,*undefined*)]› **obtain** *as'*
where *preds* (*slice-kinds* {*CFG-node m'*} *as'*) [(*cf*,*undefined*)]
and *vals*:∀ $V \in$ *Use m'*. *state-val* (*transfers* (*slice-kinds* {*CFG-node m'*} *as'*) [(*cf*,*undefined*)]) V = *state-val* (*transfers* (*kinds as*) [(*cf*,*undefined*)]) V
and $m\ -as'\rightarrow_{\surd}*\ m'$
by −(*erule fundamental-property-of-static-slicing*,*auto*)
from ‹*transfers* (*kinds as*) [(*cf*,*undefined*)] = *cfs'*› *vals* **have** ∀ $V \in$ *Use m'*.
state-val (*transfers* (*slice-kinds* {*CFG-node m'*} *as'*) [(*cf*,*undefined*)]) V =
state-val cfs' V **by** *simp*
with ‹*preds* (*slice-kinds* {*CFG-node m'*} *as'*) [(*cf*,*undefined*)]› ‹$m\ -as'\rightarrow_{\surd}*\ m'$›
‹$m' \triangleq c'$› ‹*map fst cfs'* = s'›
show ∃ *as m' cfs'*. $m\ -as\rightarrow_{\surd}*\ m' \wedge m' \triangleq c' \wedge$
preds (*slice-kinds* {*CFG-node m'*} *as*) [(*cf*, *undefined*)] ∧
(∀ V∈*Use m'*. *state-val* (*transfers* (*slice-kinds* {*CFG-node m'*} *as*)
[(*cf*, *undefined*)]) V = *state-val cfs'* V) ∧ *map fst cfs'* = s'

by *blast*
qed

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
by(*induct c*) *auto*

lemma [*dest*]:#:*c* = 0 ⟹ *False*
by(*induct c*) *auto*

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*

$$\quad \mid \mathit{Some}\ v_2 \Rightarrow (\mathit{case}\ \mathit{binop}\ \mathit{bop}\ v_1\ v_2\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{None} \mid \mathit{Some}\ v \Rightarrow \mathit{Some}\ v)))$$

abbreviation *update* :: $(\mathit{vname} \rightharpoonup \mathit{val}) \Rightarrow \mathit{vname} \Rightarrow \mathit{expr} \Rightarrow (\mathit{vname} \rightharpoonup \mathit{val})$
  where $\mathit{update}\ \mathit{cf}\ V\ e \equiv \mathit{cf}(V{:=}(\mathit{interpret}\ e\ \mathit{cf}))$

abbreviation *state-check* :: $(\mathit{vname} \rightharpoonup \mathit{val}) \Rightarrow \mathit{expr} \Rightarrow \mathit{val\ option} \Rightarrow \mathit{bool}$
where $\mathit{state\text{-}check}\ \mathit{cf}\ b\ v \equiv (\mathit{interpret}\ b\ \mathit{cf} = v)$

end

2.3 Definition of the CFG

theory *PCFG* **imports** *ProcState* **begin**

definition *Main* :: *pname*
  where *Main* = *''Main''*

datatype $\mathit{label} = \mathit{Label}\ \mathit{nat} \mid \mathit{Entry} \mid \mathit{Exit}$

2.3.1 The CFG for every procedure

Definition of $\oplus$

fun *label-incr* :: $\mathit{label} \Rightarrow \mathit{nat} \Rightarrow \mathit{label}$ $(\text{-} \oplus \text{-}\ 60)$
where $(\mathit{Label}\ l) \oplus i = \mathit{Label}\ (l + i)$
  $\mid \mathit{Entry} \oplus i \quad = \mathit{Entry}$
  $\mid \mathit{Exit} \oplus i \quad = \mathit{Exit}$

lemma *Exit-label-incr* [*dest*]: $\mathit{Exit} = n \oplus i \Longrightarrow n = \mathit{Exit}$
  by(*cases n*,*auto*)

lemma *label-incr-Exit* [*dest*]: $n \oplus i = \mathit{Exit} \Longrightarrow n = \mathit{Exit}$
  by(*cases n*,*auto*)

lemma *Entry-label-incr* [*dest*]: $\mathit{Entry} = n \oplus i \Longrightarrow n = \mathit{Entry}$
  by(*cases n*,*auto*)

lemma *label-incr-Entry* [*dest*]: $n \oplus i = \mathit{Entry} \Longrightarrow n = \mathit{Entry}$
  by(*cases n*,*auto*)

lemma *label-incr-inj*:
  $n \oplus c = n' \oplus c \Longrightarrow n = n'$
by(*cases n*)(*cases n'*,*auto*)+

lemma *label-incr-simp*:$n \oplus i = m \oplus (i + j) \Longrightarrow n = m \oplus j$
by(*cases n*,*auto*,*cases m*,*auto*)

lemma *label-incr-simp-rev*:$m \oplus (j + i) = n \oplus i \Longrightarrow m \oplus j = n$
by(*cases n*,*auto*,*cases m*,*auto*)

lemma *label-incr-start-Node-smaller*:
$Label\ l = n \oplus i \Longrightarrow n = Label\ (l - i)$
by(*cases n*,*auto*)

lemma *label-incr-start-Node-smaller-rev*:
$n \oplus i = Label\ l \Longrightarrow n = Label\ (l - i)$
by(*cases n*,*auto*)

lemma *label-incr-ge*:$Label\ l = n \oplus i \Longrightarrow l \geq i$
by(*cases n*) *auto*

lemma *label-incr-0* [*dest*]:
$[\![Label\ 0 = n \oplus i; i > 0]\!] \Longrightarrow False$
by(*cases n*) *auto*

lemma *label-incr-0-rev* [*dest*]:
$[\![n \oplus i = Label\ 0; i > 0]\!] \Longrightarrow False$
by(*cases n*) *auto*

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*
(- ⊢ - $-\!\!\rightarrow_p$ -)
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* $\Uparrow id\!\!\rightarrow_p$ *Exit*

| *Proc-CFG-LAss*:

$V{:=}e \vdash \mathit{Label}\ 0\ {-}\mathit{IEdge}\ {\Uparrow}(\lambda cf.\ \mathit{update}\ cf\ V\ e){\to_p}\ \mathit{Label}\ 1$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

| *Proc-CFG-CallSkip*:

Call p es rets ⊢ *Label 1* $-$*IEdge* $\Uparrow id \rightarrow_p$ *Exit*

Some lemmas about the procedure CFG

lemma *Proc-CFG-Exit-no-sourcenode* [*dest*]:
prog ⊢ *Exit* $-et\rightarrow_p n' \Longrightarrow$ *False*
by(*induct prog n*≡*Exit et n' rule*:*Proc-CFG.induct*,*auto*)

lemma *Proc-CFG-Entry-no-targetnode* [*dest*]:
prog ⊢ *n* $-et\rightarrow_p$ *Entry* $\Longrightarrow$ *False*
by(*induct prog n et n'*≡*Entry rule*:*Proc-CFG.induct*,*auto*)

lemma *Proc-CFG-IEdge-intra-kind*:
prog ⊢ *n* $-$*IEdge et*$\rightarrow_p n' \Longrightarrow$ *intra-kind et*
by(*induct prog n x*≡*IEdge et n' rule*:*Proc-CFG.induct*,*auto simp*:*intra-kind-def*)

lemma [*dest*]:*prog* ⊢ *n* $-$*IEdge* $(Q{:}r\hookrightarrow_p fs)\rightarrow_p n' \Longrightarrow$ *False*
by(*fastforce dest*:*Proc-CFG-IEdge-intra-kind simp*:*intra-kind-def*)

lemma [*dest*]:*prog* ⊢ *n* $-$*IEdge* $(Q\hookleftarrow_p f)\rightarrow_p n' \Longrightarrow$ *False*
by(*fastforce dest*:*Proc-CFG-IEdge-intra-kind simp*:*intra-kind-def*)

lemma *Proc-CFG-sourcelabel-less-num-nodes*:
prog ⊢ *Label l* $-et\rightarrow_p n' \Longrightarrow l < \#{:}prog$
proof(*induct prog Label l et n' arbitrary*:*l rule*:*Proc-CFG.induct*)
case (*Proc-CFG-SeqFirst* c_1 *et n'* c_2 *l*)
thus *?case* **by** *simp*
next
case (*Proc-CFG-SeqConnect* c_1 *et* c_2 *l*)
thus *?case* **by** *simp*
next
case (*Proc-CFG-SeqSecond* c_2 *n et n'* c_1 *l*)
note $n = \langle n \oplus \#{:}c_1 = Label\ l\rangle$
note $IH = \langle\bigwedge l.\ n = Label\ l \Longrightarrow l < \#{:}c_2\rangle$
from *n* **obtain** *l'* **where** *l'*:$n = Label\ l'$ **by**(*cases n*) *auto*
from *IH*[*OF this*] **have** $l' < \#{:}c_2$ **.**
with *n l'* **show** *?case* **by** *simp*
next
case (*Proc-CFG-CondThen* c_1 *n et n' b* c_2 *l*)
note $n = \langle n \oplus 1 = Label\ l\rangle$
note $IH = \langle\bigwedge l.\ n = Label\ l \Longrightarrow l < \#{:}c_1\rangle$
from *n* **obtain** *l'* **where** *l'*:$n = Label\ l'$ **by**(*cases n*) *auto*
from *IH*[*OF this*] **have** $l' < \#{:}c_1$ **.**
with *n l'* **show** *?case* **by** *simp*
next

case (*Proc-CFG-CondElse* c_2 n *et* n' b c_1 l)
note $n = \langle n \oplus (\#{:}c_1 + 1) = \mathit{Label}\ l \rangle$
note $IH = \langle \bigwedge l.\ n = \mathit{Label}\ l \Longrightarrow l < \#{:}c_2 \rangle$
from n **obtain** l' **where** $l'{:}n = \mathit{Label}\ l'$ **by**(*cases* n) *auto*
from IH[*OF this*] **have** $l' < \#{:}c_2$ **.**
with n l' **show** *?case* **by** *simp*
next
case (*Proc-CFG-WhileBody* c' n *et* n' b l)
note $n = \langle n \oplus 2 = \mathit{Label}\ l \rangle$
note $IH = \langle \bigwedge l.\ n = \mathit{Label}\ l \Longrightarrow l < \#{:}c' \rangle$
from n **obtain** l' **where** $l'{:}n = \mathit{Label}\ l'$ **by**(*cases* n) *auto*
from IH[*OF this*] **have** $l' < \#{:}c'$ **.**
with n l' **show** *?case* **by** *simp*
next
case (*Proc-CFG-WhileBodyExit* c' n *et* b l)
note $n = \langle n \oplus 2 = \mathit{Label}\ l \rangle$
note $IH = \langle \bigwedge l.\ n = \mathit{Label}\ l \Longrightarrow l < \#{:}c' \rangle$
from n **obtain** l' **where** $l'{:}n = \mathit{Label}\ l'$ **by**(*cases* n) *auto*
from IH[*OF this*] **have** $l' < \#{:}c'$ **.**
with n l' **show** *?case* **by** *simp*
qed (*auto simp*:*num-inner-nodes-gr-0*)

lemma *Proc-CFG-targetlabel-less-num-nodes*:
$\mathit{prog} \vdash n\ -et\rightarrow_p\ \mathit{Label}\ l \Longrightarrow l < \#{:}\mathit{prog}$
proof(*induct prog n et Label l arbitrary*:*l rule*:*Proc-CFG.induct*)
case (*Proc-CFG-SeqFirst* c_1 n *et* c_2 l)
thus *?case* **by** *simp*
next
case (*Proc-CFG-SeqSecond* c_2 n *et* n' c_1 l)
note $n' = \langle n' \oplus \#{:}c_1 = \mathit{Label}\ l \rangle$
note $IH = \langle \bigwedge l.\ n' = \mathit{Label}\ l \Longrightarrow l < \#{:}c_2 \rangle$
from n' **obtain** l' **where** $l'{:}n' = \mathit{Label}\ l'$ **by**(*cases* n') *auto*
from IH[*OF this*] **have** $l' < \#{:}c_2$ **.**
with n' l' **show** *?case* **by** *simp*
next
case (*Proc-CFG-CondThen* c_1 n *et* n' b c_2 l)
note $n' = \langle n' \oplus 1 = \mathit{Label}\ l \rangle$
note $IH = \langle \bigwedge l.\ n' = \mathit{Label}\ l \Longrightarrow l < \#{:}c_1 \rangle$
from n' **obtain** l' **where** $l'{:}n' = \mathit{Label}\ l'$ **by**(*cases* n') *auto*
from IH[*OF this*] **have** $l' < \#{:}c_1$ **.**
with n' l' **show** *?case* **by** *simp*
next
case (*Proc-CFG-CondElse* c_2 n *et* n' b c_1 l)
note $n' = \langle n' \oplus (\#{:}c_1 + 1) = \mathit{Label}\ l \rangle$
note $IH = \langle \bigwedge l.\ n' = \mathit{Label}\ l \Longrightarrow l < \#{:}c_2 \rangle$
from n' **obtain** l' **where** $l'{:}n' = \mathit{Label}\ l'$ **by**(*cases* n') *auto*
from IH[*OF this*] **have** $l' < \#{:}c_2$ **.**
with n' l' **show** *?case* **by** *simp*

next
case (*Proc-CFG-WhileBody c′ n et n′ b l*)
note $n' = \langle n' \oplus 2 = Label\ l \rangle$
note $IH = \langle \bigwedge l.\ n' = Label\ l \Longrightarrow l < \#{:}c' \rangle$
from n' **obtain** l' **where** $l'{:}n' = Label\ l'$ **by**(*cases n′*) *auto*
from $IH[OF\ this]$ **have** $l' < \#{:}c'$ **.**
with $n'\ l'$ **show** *?case* **by** *simp*
qed (*auto simp*:*num-inner-nodes-gr-0*)

lemma *Proc-CFG-EntryD*:
$prog \vdash Entry\ -et\rightarrow_p\ n'$
$\Longrightarrow (n' = Exit \wedge et = IEdge(\lambda s.\ False)_{\surd}) \vee (n' = Label\ 0 \wedge et = IEdge\ (\lambda s.\ True)_{\surd})$
by(*induct prog n*≡*Entry et n′ rule*:*Proc-CFG.induct*,*auto*)

lemma *Proc-CFG-Exit-edge*:
obtains *l et* **where** $prog \vdash Label\ l\ -IEdge\ et\rightarrow_p\ Exit$ **and** $l \leq \#{:}prog$
proof(*atomize-elim*)
show $\exists\, l\ et.\ prog \vdash Label\ l\ -IEdge\ et\rightarrow_p\ Exit \wedge l \leq \#{:}prog$
proof(*induct prog*)
case *Skip*
have $Skip \vdash Label\ 0\ -IEdge\ \Uparrow id\rightarrow_p\ Exit$ **by**(*rule Proc-CFG-Skip*)
thus *?case* **by** *fastforce*
next
case (*LAss V e*)
have $V{:=}e \vdash Label\ 1\ -IEdge\ \Uparrow id\rightarrow_p\ Exit$ **by**(*rule Proc-CFG-LAssSkip*)
thus *?case* **by** *fastforce*
next
case (*Seq* c_1 c_2)
from $\langle \exists\, l\ et.\ c_2 \vdash Label\ l\ -IEdge\ et\rightarrow_p\ Exit \wedge l \leq \#{:}c_2 \rangle$
obtain *l et* **where** $c_2 \vdash Label\ l\ -IEdge\ et\rightarrow_p\ Exit$ **and** $l \leq \#{:}c_2$ **by** *blast*
hence $c_1;;c_2 \vdash Label\ l \oplus \#{:}c_1\ -IEdge\ et\rightarrow_p\ Exit \oplus \#{:}c_1$
by(*fastforce intro*:*Proc-CFG-SeqSecond*)
with $\langle l \leq \#{:}c_2 \rangle$ **show** *?case* **by** *fastforce*
next
case (*Cond b* c_1 c_2)
from $\langle \exists\, l\ et.\ c_1 \vdash Label\ l\ -IEdge\ et\rightarrow_p\ Exit \wedge l \leq \#{:}c_1 \rangle$
obtain *l et* **where** $c_1 \vdash Label\ l\ -IEdge\ et\rightarrow_p\ Exit$ **and** $l \leq \#{:}c_1$ **by** *blast*
hence *if* (*b*) c_1 *else* $c_2 \vdash Label\ l \oplus 1\ -IEdge\ et\rightarrow_p\ Exit \oplus 1$
by(*fastforce intro*:*Proc-CFG-CondThen*)
with $\langle l \leq \#{:}c_1 \rangle$ **show** *?case* **by** *fastforce*
next
case (*While b c′*)
have *while* (*b*) $c' \vdash Label\ 1\ -IEdge\ \Uparrow id\rightarrow_p\ Exit$ **by**(*rule Proc-CFG-WhileFalseSkip*)
thus *?case* **by** *fastforce*
next
case (*Call p es rets*)

have *Call p es rets* $\vdash$ *Label 1* $-$*IEdge* $\Uparrow id \rightarrow_p$ *Exit* **by**(*rule Proc-CFG-CallSkip*)
thus *?case* **by** *fastforce*
qed
qed

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)
by(*induct prog n et*$\equiv$*CEdge* $(p,es,rets)$ n' *rule*:*Proc-CFG.induct*,*auto*)

lemma *Proc-CFG-Call-target-0*:
$prog \vdash n -$*CEdge* $(p,es,rets)\rightarrow_p$ *Label 0* $\Longrightarrow n =$ *Entry*
by(*induct prog n et*$\equiv$*CEdge* $(p,es,rets)$ $n'\equiv$*Label 0 rule*:*Proc-CFG.induct*)
(*auto dest*:*Proc-CFG-Call-Labels*)

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(*induct prog n CEdge* $(p,es,rets)$ n' *arbitrary*:n'' *rule*:*Proc-CFG.induct*)
case (*Proc-CFG-SeqFirst* c_1 n n' c_2)
note *IH* = $\langle\bigwedge n''.\ c_1 \vdash n -$*IEdge* $et\rightarrow_p n'' \Longrightarrow$ *False*$\rangle$
from $\langle c_1;;c_2 \vdash n -$*IEdge* $et\rightarrow_p n''\rangle$ $\langle c_1 \vdash n -$*CEdge* $(p,$ *es*, *rets*$)\rightarrow_p n'\rangle$
$\langle n' \neq$ *Exit*$\rangle$
obtain *nx* **where** $c_1 \vdash n -$*IEdge* $et\rightarrow_p nx$
apply $-$ **apply**(*erule Proc-CFG.cases*)
apply(*auto intro*:*Proc-CFG-Entry-Exit Proc-CFG-Entry*)
by(*case-tac n*)(*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
then show *?case* **by** (*rule IH*)
next
case (*Proc-CFG-SeqConnect* c_1 n c_2)
from $\langle c_1 \vdash n -$*CEdge* $(p,$ *es*, *rets*$)\rightarrow_p$ *Exit*$\rangle$
show *?case* **by**(*fastforce dest*:*Proc-CFG-Call-Labels*)
next
case (*Proc-CFG-SeqSecond* c_2 n n' c_1)
note *IH* = $\langle\bigwedge n''.\ c_2 \vdash n -$*IEdge* $et\rightarrow_p n'' \Longrightarrow$ *False*$\rangle$
from $\langle c_1;;c_2 \vdash n \oplus \#{:}c_1 -$*IEdge* $et\rightarrow_p n''\rangle$ $\langle c_2 \vdash n -$*CEdge* $(p,$ *es*, *rets*$)\rightarrow_p n'\rangle$
$\langle n \neq$ *Entry*$\rangle$
obtain *nx* **where** $c_2 \vdash n -$*IEdge* $et\rightarrow_p nx$
apply $-$ **apply**(*erule Proc-CFG.cases*,*auto*)
apply(*cases n*) **apply**(*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
apply(*cases n*) **apply**(*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
by(*cases n*,*auto*,*case-tac n*,*auto*)
then show *?case* **by** (*rule IH*)
next
case (*Proc-CFG-CondThen* c_1 n n' b c_2)
note *IH* = $\langle\bigwedge n''.\ c_1 \vdash n -$*IEdge* $et\rightarrow_p n'' \Longrightarrow$ *False*$\rangle$
from $\langle$*if* (b) c_1 *else* $c_2 \vdash n \oplus 1 -$*IEdge* $et\rightarrow_p n''\rangle$ $\langle c_1 \vdash n -$*CEdge* $(p,$ *es*, *rets*$)\rightarrow_p n'\rangle$

```
    ‹n ≠ Entry›
  obtain nx where c1 ⊢ n −IEdge et→p nx
    apply − apply(erule Proc-CFG.cases,auto)
     apply(cases n) apply auto apply(case-tac n) apply auto
    apply(cases n) apply auto
    by(case-tac n)(auto dest:Proc-CFG-sourcelabel-less-num-nodes)
  then show ?case by (rule IH)
next
  case (Proc-CFG-CondElse c2 n n' b c1)
  note IH = ‹⋀n''. c2 ⊢ n −IEdge et→p n'' ⟹ False›
  from ‹if (b) c1 else c2 ⊢ n ⊕ #:c1 + 1 −IEdge et→p n''› ‹c2 ⊢ n −CEdge (p,
es, rets)→p n'›
    ‹n ≠ Entry›
  obtain nx where c2 ⊢ n −IEdge et→p nx
    apply − apply(erule Proc-CFG.cases,auto)
     apply(cases n) apply auto
     apply(case-tac n) apply(auto dest:Proc-CFG-sourcelabel-less-num-nodes)
    by(cases n,auto,case-tac n,auto)
  then show ?case by (rule IH)
next
  case (Proc-CFG-WhileBody c' n n' b)
  note IH = ‹⋀n''. c' ⊢ n −IEdge et→p n'' ⟹ False›
  from ‹while (b) c' ⊢ n ⊕ 2 −IEdge et→p n''› ‹c' ⊢ n −CEdge (p, es, rets)→p
n'›
    ‹n ≠ Entry› ‹n' ≠ Exit›
  obtain nx where c' ⊢ n −IEdge et→p nx
    apply − apply(erule Proc-CFG.cases,auto)
      apply(drule label-incr-ge[OF sym]) apply simp
     apply(cases n) apply auto apply(case-tac n) apply auto
    by(cases n,auto,case-tac n,auto)
  then show ?case by (rule IH)
next
  case (Proc-CFG-WhileBodyExit c' n b)
  from ‹c' ⊢ n −CEdge (p, es, rets)→p Exit›
  show ?case by(fastforce dest:Proc-CFG-Call-Labels)
next
  case Proc-CFG-Call
  from ‹Call p es rets ⊢ Label 0 −IEdge et→p n''›
  show ?case by(fastforce elim:Proc-CFG.cases)
qed


lemma Proc-CFG-Call-Intra-edge-not-same-target:
  ⟦prog ⊢ n −CEdge (p,es,rets)→p n'; prog ⊢ n'' −IEdge et→p n'⟧ ⟹ False
proof(induct prog n CEdge (p,es,rets) n' arbitrary:n'' rule:Proc-CFG.induct)
  case (Proc-CFG-SeqFirst c1 n n' c2)
  note IH = ‹⋀n''. c1 ⊢ n'' −IEdge et→p n' ⟹ False›
  from ‹c1;;c2 ⊢ n'' −IEdge et→p n'› ‹c1 ⊢ n −CEdge (p, es, rets)→p n'›
    ‹n' ≠ Exit›
```

```
  have c1 ⊢ n'' −IEdge et→p n'
    apply − apply(erule Proc-CFG.cases)
    apply(auto intro:Proc-CFG-Entry dest:Proc-CFG-targetlabel-less-num-nodes)
    by(case-tac n')(auto dest:Proc-CFG-targetlabel-less-num-nodes)
  then show ?case by (rule IH)
next
  case (Proc-CFG-SeqConnect c1 n c2)
  from ‹c1 ⊢ n −CEdge (p, es, rets)→p Exit›
  show ?case by(fastforce dest:Proc-CFG-Call-Labels)
next
  case (Proc-CFG-SeqSecond c2 n n' c1)
  note IH = ‹⋀n''. c2 ⊢ n'' −IEdge et→p n' ⟹ False›
  from ‹c1;;c2 ⊢ n'' −IEdge et→p n' ⊕ #:c1› ‹c2 ⊢ n −CEdge (p, es, rets)→p n'›

    ‹n ≠ Entry›
  obtain nx where c2 ⊢ nx −IEdge et→p n'
    apply − apply(erule Proc-CFG.cases,auto)
      apply(fastforce intro:Proc-CFG-Entry-Exit)
     apply(cases n') apply(auto dest:Proc-CFG-targetlabel-less-num-nodes)
    apply(cases n') apply(auto dest:Proc-CFG-Call-target-0)
    apply(cases n') apply(auto dest:Proc-CFG-Call-Labels)
    by(case-tac n') auto
  then show ?case by (rule IH)
next
  case (Proc-CFG-CondThen c1 n n' b c2)
  note IH = ‹⋀n''. c1 ⊢ n'' −IEdge et→p n' ⟹ False›
  from ‹if (b) c1 else c2 ⊢ n'' −IEdge et→p n' ⊕ 1› ‹c1 ⊢ n −CEdge (p, es,
rets)→p n'›
    ‹n ≠ Entry›
  obtain nx where c1 ⊢ nx −IEdge et→p n'
    apply − apply(erule Proc-CFG.cases,auto)
       apply(cases n') apply(auto intro:Proc-CFG-Entry-Exit)
      apply(cases n') apply(auto dest:Proc-CFG-Call-target-0)
     apply(cases n') apply(auto dest:Proc-CFG-targetlabel-less-num-nodes)
    apply(cases n') apply auto apply(case-tac n') apply auto
    apply(cases n') apply auto
    apply(case-tac n') apply(auto dest:Proc-CFG-targetlabel-less-num-nodes)
    by(case-tac n')(auto dest:Proc-CFG-Call-Labels)
  then show ?case by (rule IH)
next
  case (Proc-CFG-CondElse c2 n n' b c1)
  note IH = ‹⋀n''. c2 ⊢ n'' −IEdge et→p n' ⟹ False›
  from ‹if (b) c1 else c2 ⊢ n'' −IEdge et→p n' ⊕ #:c1 + 1› ‹c2 ⊢ n −CEdge (p,
es, rets)→p n'›
    ‹n ≠ Entry›
  obtain nx where c2 ⊢ nx −IEdge et→p n'
    apply − apply(erule Proc-CFG.cases,auto)
       apply(cases n') apply(auto intro:Proc-CFG-Entry-Exit)
      apply(cases n') apply(auto dest:Proc-CFG-Call-target-0)
```

```
    apply(cases n') apply(auto dest:Proc-CFG-Call-target-0)
   apply(cases n') apply auto
    apply(case-tac n') apply(auto dest:Proc-CFG-targetlabel-less-num-nodes)
   apply(case-tac n') apply(auto dest:Proc-CFG-Call-Labels)
  by(cases n',auto,case-tac n',auto)
 then show ?case by (rule IH)
next
 case (Proc-CFG-WhileBody c' n n' b)
 note IH = ‹⋀n''. c' ⊢ n'' −IEdge et→p n' ⟹ False›
 from ‹while (b) c' ⊢ n'' −IEdge et→p n' ⊕ 2› ‹c' ⊢ n −CEdge (p, es, rets)→p
n'›
   ‹n ≠ Entry› ‹n' ≠ Exit›
 obtain nx where c' ⊢ nx −IEdge et→p n'
  apply − apply(erule Proc-CFG.cases,auto)
    apply(cases n') apply(auto dest:Proc-CFG-Call-target-0)
   apply(cases n') apply auto
  by(cases n',auto,case-tac n',auto)
 then show ?case by (rule IH)
next
 case (Proc-CFG-WhileBodyExit c' n b)
 from ‹c' ⊢ n −CEdge (p, es, rets)→p Exit›
 show ?case by(fastforce dest:Proc-CFG-Call-Labels)
next
 case Proc-CFG-Call
 from ‹Call p es rets ⊢ n'' −IEdge et→p Label 1›
 show ?case by(fastforce elim:Proc-CFG.cases)
qed


lemma Proc-CFG-Call-nodes-eq:
 ⟦prog ⊢ n −CEdge (p,es,rets)→p n'; prog ⊢ n −CEdge (p',es',rets')→p n'⟧
 ⟹ n' = n'' ∧ p = p' ∧ es = es' ∧ rets = rets'
proof(induct prog n CEdge (p,es,rets) n' arbitrary:n'' rule:Proc-CFG.induct)
 case (Proc-CFG-SeqFirst c1 n n' c2)
 note IH = ‹⋀n''. c1 ⊢ n −CEdge (p',es',rets')→p n''
   ⟹ n' = n'' ∧ p = p' ∧ es = es' ∧ rets = rets'›
 from ‹c1;; c2 ⊢ n −CEdge (p',es',rets')→p n''› ‹c1 ⊢ n −CEdge (p,es,rets)→p
n'›
 have c1 ⊢ n −CEdge (p',es',rets')→p n''
  apply − apply(erule Proc-CFG.cases,auto)
   apply(fastforce dest:Proc-CFG-Call-Labels)
  by(case-tac n,(fastforce dest:Proc-CFG-sourcelabel-less-num-nodes)+)
 then show ?case by (rule IH)
next
 case (Proc-CFG-SeqConnect c1 n c2)
 from ‹c1 ⊢ n −CEdge (p,es,rets)→p Exit› have False
  by(fastforce dest:Proc-CFG-Call-Labels)
 thus ?case by simp
next
```

case (*Proc-CFG-SeqSecond* c_2 n n' c_1)
note $IH = \langle \bigwedge n''.\ c_2 \vdash n\ -CEdge\ (p',es',rets') \rightarrow_p n'' \Longrightarrow n' = n'' \wedge p = p' \wedge es = es' \wedge rets = rets' \rangle$
from $\langle c_1;;c_2 \vdash n \oplus \#{:}c_1\ -CEdge\ (p',es',rets') \rightarrow_p n'' \rangle$ $\langle n \neq Entry \rangle$
obtain nx **where** $edge{:}c_2 \vdash n\ -CEdge\ (p',es',rets') \rightarrow_p nx$ **and** $nx{:}nx \oplus \#{:}c_1 = n''$
apply − **apply**(*erule Proc-CFG.cases*,*auto*)
by(*cases n*,*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes label-incr-inj*)+
from *edge* **have** $n' = nx \wedge p = p' \wedge es = es' \wedge rets = rets'$ **by** (*rule IH*)
with *nx* **show** *?case* **by** *auto*
next
case (*Proc-CFG-CondThen* c_1 n n' b c_2)
note $IH = \langle \bigwedge n''.\ c_1 \vdash n\ -CEdge\ (p',es',rets') \rightarrow_p n'' \Longrightarrow n' = n'' \wedge p = p' \wedge es = es' \wedge rets = rets' \rangle$
from $\langle$*if* (b) c_1 *else* $c_2 \vdash n \oplus 1\ -CEdge\ (p',es',rets') \rightarrow_p n'' \rangle$
obtain nx **where** $c_1 \vdash n\ -CEdge\ (p',es',rets') \rightarrow_p nx \wedge nx \oplus 1 = n''$
proof(*rule Proc-CFG.cases*)
fix c_2' nx etx nx' bx c_1'
assume *if* (b) c_1 *else* c_2 = *if* (bx) c_1' *else* c_2'
and $n \oplus 1 = nx \oplus \#{:}c_1' + 1$ **and** $nx \neq Entry$
with $\langle c_1 \vdash n\ -CEdge\ (p,es,rets) \rightarrow_p n' \rangle$ **obtain** l **where** $n = Label\ l$ **and** $l \geq \#{:}c_1$
by(*cases n*,*auto*,*cases nx*,*auto*)
with $\langle c_1 \vdash n\ -CEdge\ (p,es,rets) \rightarrow_p n' \rangle$ **have** *False*
by(*fastforce dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
thus *?thesis* **by** *simp*
qed (*auto dest*:*label-incr-inj*)
then obtain nx **where** $edge{:}c_1 \vdash n\ -CEdge\ (p',es',rets') \rightarrow_p nx$ **and** $nx{:}nx \oplus 1 = n''$ **by** *blast*
from *IH*[*OF edge*] nx **show** *?case* **by** *simp*
next
case (*Proc-CFG-CondElse* c_2 n n' b c_1)
note $IH = \langle \bigwedge n''.\ c_2 \vdash n\ -CEdge\ (p',es',rets') \rightarrow_p n'' \Longrightarrow n' = n'' \wedge p = p' \wedge es = es' \wedge rets = rets' \rangle$
from $\langle$*if* (b) c_1 *else* $c_2 \vdash n \oplus \#{:}c_1 + 1\ -CEdge\ (p',es',rets') \rightarrow_p n'' \rangle$
obtain nx **where** $c_2 \vdash n\ -CEdge\ (p',es',rets') \rightarrow_p nx \wedge nx \oplus \#{:}c_1 + 1 = n''$
proof(*rule Proc-CFG.cases*)
fix c_1' nx etx nx' bx c_2'
assume *ifs*:*if* (b) c_1 *else* c_2 = *if* (bx) c_1' *else* c_2'
and $n \oplus \#{:}c_1 + 1 = nx \oplus 1$ **and** $nx \neq Entry$
and $edge{:}c_1' \vdash nx\ -etx \rightarrow_p nx'$
then obtain l **where** $nx = Label\ l$ **and** $l \geq \#{:}c_1$
by(*cases n*,*auto*,*cases nx*,*auto*)
with *edge ifs* **have** *False*
by(*fastforce dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
thus *?thesis* **by** *simp*
qed (*auto dest*:*label-incr-inj*)
then obtain nx **where** $edge{:}c_2 \vdash n\ -CEdge\ (p',es',rets') \rightarrow_p nx$ **and** $nx{:}nx \oplus \#{:}c_1 + 1 = n''$

by *blast*
from *IH*[*OF edge*] *nx* **show** *?case* **by** *simp*
next
case (*Proc-CFG-WhileBody c′ n n′ b*)
note *IH* = ‹⋀*n″*. *c′* ⊢ *n* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n″*
⟹ *n′* = *n″* ∧ *p* = *p′* ∧ *es* = *es′* ∧ *rets* = *rets′*›
from ‹*while* (*b*) *c′* ⊢ *n* ⊕ *2* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n″*›
obtain *nx* **where** *c′* ⊢ *n* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *nx* ∧ *nx* ⊕ *2* = *n″*
by(*rule Proc-CFG.cases*,*auto dest*:*label-incr-inj Proc-CFG-Call-Labels*)
then obtain *nx* **where** *edge*:*c′* ⊢ *n* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *nx*
and *nx*:*nx* ⊕ *2* = *n″* **by** *blast*
from *IH*[*OF edge*] *nx* **show** *?case* **by** *simp*
next
case (*Proc-CFG-WhileBodyExit c′ n b*)
from ‹*c′* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *Exit*› **have** *False*
by(*fastforce dest*:*Proc-CFG-Call-Labels*)
thus *?case* **by** *simp*
next
case *Proc-CFG-Call*
from ‹*Call p es rets* ⊢ *Label 0* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n″*›
have *p* = *p′* ∧ *es* = *es′* ∧ *rets* = *rets′* ∧ *n″* = *Label 1*
by(*auto elim*:*Proc-CFG.cases*)
then show *?case* **by** *simp*
qed

lemma *Proc-CFG-Call-nodes-eq′*:
⟦*prog* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *n′*; *prog* ⊢ *n″* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n′*⟧
⟹ *n* = *n″* ∧ *p* = *p′* ∧ *es* = *es′* ∧ *rets* = *rets′*
proof(*induct prog n CEdge* (*p*,*es*,*rets*) *n′ arbitrary*:*n″ rule*:*Proc-CFG.induct*)
case (*Proc-CFG-SeqFirst c*$_1$ *n n′ c*$_2$)
note *IH* = ‹⋀*n″*. *c*$_1$ ⊢ *n″* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n′*
⟹ *n* = *n″* ∧ *p* = *p′* ∧ *es* = *es′* ∧ *rets* = *rets′*›
from ‹*c*$_1$;;*c*$_2$ ⊢ *n″* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n′*› ‹*c*$_1$ ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *n′*›
have *c*$_1$ ⊢ *n″* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n′*
apply − **apply**(*erule Proc-CFG.cases*,*auto*)
apply(*fastforce dest*:*Proc-CFG-Call-Labels*)
by(*case-tac n′*,*auto dest*:*Proc-CFG-targetlabel-less-num-nodes Proc-CFG-Call-Labels*)
then show *?case* **by** (*rule IH*)
next
case (*Proc-CFG-SeqConnect c*$_1$ *n c*$_2$)
from ‹*c*$_1$ ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *Exit*› **have** *False*
by(*fastforce dest*:*Proc-CFG-Call-Labels*)
thus *?case* **by** *simp*
next
case (*Proc-CFG-SeqSecond c*$_2$ *n n′ c*$_1$)
note *IH* = ‹⋀*n″*. *c*$_2$ ⊢ *n″* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n′*
⟹ *n* = *n″* ∧ *p* = *p′* ∧ *es* = *es′* ∧ *rets* = *rets′*›

from ‹c_1;;$c_2 \vdash n'' -CEdge\ (p',es',rets')\rightarrow_p n' \oplus \#{:}c_1$›
obtain *nx* **where** *edge*:$c_2 \vdash nx -CEdge\ (p',es',rets')\rightarrow_p n'$ **and** *nx*:$nx \oplus \#{:}c_1 = n''$
apply − **apply**(*erule Proc-CFG.cases*,*auto*)
by(*cases n'*,
auto dest:*Proc-CFG-targetlabel-less-num-nodes Proc-CFG-Call-Labels label-incr-inj*)
from *edge* **have** $n = nx \wedge p = p' \wedge es = es' \wedge rets = rets'$ **by** (*rule IH*)
with *nx* **show** *?case* **by** *auto*
next
case (*Proc-CFG-CondThen* c_1 *n n' b* c_2)
note *IH* = ‹$\bigwedge n''.\ c_1 \vdash n'' -CEdge\ (p',es',rets')\rightarrow_p n' \Longrightarrow n = n'' \wedge p = p' \wedge es = es' \wedge rets = rets'$›
from ‹*if* (*b*) c_1 *else* $c_2 \vdash n'' -CEdge\ (p',es',rets')\rightarrow_p n' \oplus 1$›
obtain *nx* **where** $c_1 \vdash nx -CEdge\ (p',es',rets')\rightarrow_p n' \wedge nx \oplus 1 = n''$
proof(*cases*)
case (*Proc-CFG-CondElse nx nx'*)
from ‹$n' \oplus 1 = nx' \oplus \#{:}c_1 + 1$›
‹$c_1 \vdash n -CEdge\ (p,es,rets)\rightarrow_p n'$›
obtain *l* **where** $n' = Label\ l$ **and** $l \geq \#{:}c_1$
by(*cases n'*, *auto dest*:*Proc-CFG-Call-Labels*,*cases nx'*,*auto*)
with ‹$c_1 \vdash n -CEdge\ (p,es,rets)\rightarrow_p n'$› **have** *False*
by(*fastforce dest*:*Proc-CFG-targetlabel-less-num-nodes*)
thus *?thesis* **by** *simp*
qed (*auto dest*:*label-incr-inj*)
then obtain *nx* **where** *edge*:$c_1 \vdash nx -CEdge\ (p',es',rets')\rightarrow_p n'$
and *nx*:$nx \oplus 1 = n''$
by *blast*
from *IH*[*OF edge*] *nx* **show** *?case* **by** *simp*
next
case (*Proc-CFG-CondElse* c_2 *n n' b* c_1)
note *IH* = ‹$\bigwedge n''.\ c_2 \vdash n'' -CEdge\ (p',es',rets')\rightarrow_p n' \Longrightarrow n = n'' \wedge p = p' \wedge es = es' \wedge rets = rets'$›
from ‹*if* (*b*) c_1 *else* $c_2 \vdash n'' -CEdge\ (p',es',rets')\rightarrow_p n' \oplus \#{:}c_1 + 1$›
obtain *nx* **where** $c_2 \vdash nx -CEdge\ (p',es',rets')\rightarrow_p n' \wedge nx \oplus \#{:}c_1 + 1 = n''$
proof(*cases*)
case (*Proc-CFG-CondThen nx nx'*)
from ‹$n' \oplus \#{:}c_1 + 1 = nx' \oplus 1$›
‹$c_1 \vdash nx -CEdge\ (p',es',rets')\rightarrow_p nx'$›
obtain *l* **where** $nx' = Label\ l$ **and** $l \geq \#{:}c_1$
by(*cases n'*,*auto*,*cases nx'*,*auto dest*:*Proc-CFG-Call-Labels*)
with ‹$c_1 \vdash nx -CEdge\ (p',es',rets')\rightarrow_p nx'$›
have *False* **by**(*fastforce dest*:*Proc-CFG-targetlabel-less-num-nodes*)
thus *?thesis* **by** *simp*
qed (*auto dest*:*label-incr-inj*)
then obtain *nx* **where** *edge*:$c_2 \vdash nx -CEdge\ (p',es',rets')\rightarrow_p n'$
and *nx*:$nx \oplus \#{:}c_1 + 1 = n''$
by *blast*
from *IH*[*OF edge*] *nx* **show** *?case* **by** *simp*

next
case (*Proc-CFG-WhileBody c′ n n′ b*)
note *IH* = ‹⋀*n″*. *c′* ⊢ *n″* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n′* ⟹ *n* = *n″* ∧ *p* = *p′* ∧ *es* = *es′* ∧ *rets* = *rets′*›
from ‹*while* (*b*) *c′* ⊢ *n″* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n′* ⊕ *2*›
obtain *nx* **where** *edge*:*c′* ⊢ *nx* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n′* **and** *nx*:*nx* ⊕ *2* = *n″*
by(*rule Proc-CFG.cases*,*auto dest*:*label-incr-inj*)
from *IH*[*OF edge*] *nx* **show** *?case* **by** *simp*
next
case (*Proc-CFG-WhileBodyExit c′ n b*)
from ‹*c′* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *Exit*›
have *False* **by**(*fastforce dest*:*Proc-CFG-Call-Labels*)
thus *?case* **by** *simp*
next
case *Proc-CFG-Call*
from ‹*Call p es rets* ⊢ *n″* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *Label 1*›
have *p* = *p′* ∧ *es* = *es′* ∧ *rets* = *rets′* ∧ *n″* = *Label 0*
by(*auto elim*:*Proc-CFG.cases*)
then show *?case* **by** *simp*
qed

lemma *Proc-CFG-Call-targetnode-no-Call-sourcenode*:
⟦*prog* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *n′*; *prog* ⊢ *n′* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n″*⟧
⟹ *False*
proof(*induct prog n CEdge* (*p*,*es*,*rets*) *n′ arbitrary*:*n″ rule*:*Proc-CFG.induct*)
case (*Proc-CFG-SeqFirst* c_1 *n n′* c_2)
note *IH* = ‹⋀*n″*. c_1 ⊢ *n′* −*CEdge* (*p′*, *es′*, *rets′*)→$_p$ *n″* ⟹ *False*›
from ‹c_1;; c_2 ⊢ *n′* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n″*› ‹c_1 ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *n′*›
have c_1 ⊢ *n′* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *n″*
apply − **apply**(*erule Proc-CFG.cases*,*auto*)
apply(*fastforce dest*:*Proc-CFG-Call-Labels*)
by(*case-tac n*)(*auto dest*:*Proc-CFG-targetlabel-less-num-nodes*)
then show *?case* **by** (*rule IH*)
next
case (*Proc-CFG-SeqConnect* c_1 *n* c_2)
from ‹c_1 ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *Exit*› **have** *False*
by(*fastforce dest*:*Proc-CFG-Call-Labels*)
thus *?case* **by** *simp*
next
case (*Proc-CFG-SeqSecond* c_2 *n n′* c_1)
note *IH* = ‹⋀*n″*. c_2 ⊢ *n′* −*CEdge* (*p′*, *es′*, *rets′*)→$_p$ *n″* ⟹ *False*›
from ‹c_1;; c_2 ⊢ *n′* ⊕ #:c_1 −*CEdge* (*p′*, *es′*, *rets′*)→$_p$ *n″*› ‹c_2 ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *n′*›
obtain *nx* **where** c_2 ⊢ *n′* −*CEdge* (*p′*,*es′*,*rets′*)→$_p$ *nx*
apply − **apply**(*erule Proc-CFG.cases*,*auto*)
apply(*cases n′*) **apply**(*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)

```
    apply(fastforce dest:Proc-CFG-Call-Labels)
   by(cases n',auto,case-tac n,auto)
 then show ?case by (rule IH)
next
 case (Proc-CFG-CondThen c₁ n n' b c₂)
 note IH = ⟨⋀n''. c₁ ⊢ n' −CEdge (p',es',rets')→p n'' ⟹ False⟩
 from ⟨if (b) c₁ else c₂ ⊢ n' ⊕ 1 −CEdge (p', es', rets')→p n''⟩ ⟨c₁ ⊢ n −CEdge
(p,es,rets)→p n'⟩
 obtain nx where c₁ ⊢ n' −CEdge (p',es',rets')→p nx
   apply − apply(erule Proc-CFG.cases,auto)
    apply(cases n') apply auto apply(case-tac n) apply auto
   apply(cases n') apply auto
   by(case-tac n)(auto dest:Proc-CFG-targetlabel-less-num-nodes)
 then show ?case by (rule IH)
next
 case (Proc-CFG-CondElse c₂ n n' b c₁)
 note IH = ⟨⋀n''. c₂ ⊢ n' −CEdge (p',es',rets')→p n'' ⟹ False⟩
 from ⟨if (b) c₁ else c₂ ⊢ n' ⊕ #:c₁ + 1 −CEdge (p', es', rets')→p n''⟩
   ⟨c₂ ⊢ n −CEdge (p,es,rets)→p n'⟩
 obtain nx where c₂ ⊢ n' −CEdge (p',es',rets')→p nx
   apply − apply(erule Proc-CFG.cases,auto)
    apply(cases n') apply auto
    apply(case-tac n) apply(auto dest:Proc-CFG-sourcelabel-less-num-nodes)
   by(cases n',auto,case-tac n,auto)
 then show ?case by (rule IH)
next
 case (Proc-CFG-WhileBody c' n n' b)
 note IH = ⟨⋀n''. c' ⊢ n' −CEdge (p',es',rets')→p n'' ⟹ False⟩
  from ⟨while (b) c' ⊢ n' ⊕ 2 −CEdge (p', es', rets')→p n''⟩ ⟨c' ⊢ n −CEdge
(p,es,rets)→p n'⟩
 obtain nx where c' ⊢ n' −CEdge (p',es',rets')→p nx
   apply − apply(erule Proc-CFG.cases,auto)
   by(cases n',auto,case-tac n,auto)+
 then show ?case by (rule IH)
next
 case (Proc-CFG-WhileBodyExit c' n b)
 from ⟨c' ⊢ n −CEdge (p, es, rets)→p Exit⟩
 show ?case by(fastforce dest:Proc-CFG-Call-Labels)
next
 case Proc-CFG-Call
 from ⟨Call p es rets ⊢ Label 1 −CEdge (p', es', rets')→p n''⟩
 show ?case by(fastforce elim:Proc-CFG.cases)
qed


lemma Proc-CFG-Call-follows-id-edge:
 ⟦prog ⊢ n −CEdge (p,es,rets)→p n'; prog ⊢ n' −IEdge et→p n''⟧ ⟹ et = ↑id
proof(induct prog n CEdge (p,es,rets) n' arbitrary:n'' rule:Proc-CFG.induct)
 case (Proc-CFG-SeqFirst c₁ n n' c₂)
```

note *IH* = ‹$\bigwedge n''$. $c_1 \vdash n' -IEdge\ et\rightarrow_p n'' \Longrightarrow et = \Uparrow id$›
from ‹$c_1;;c_2 \vdash n' -IEdge\ et\rightarrow_p n''$› ‹$c_1 \vdash n -CEdge\ (p,es,rets)\rightarrow_p n'$› ‹$n' \neq Exit$›
obtain *nx* **where** $c_1 \vdash n' -IEdge\ et\rightarrow_p nx$
apply − **apply**(*erule Proc-CFG.cases*,*auto*)
by(*case-tac n*)(*auto dest*:*Proc-CFG-targetlabel-less-num-nodes*)
then show *?case* **by** (*rule IH*)
next
case (*Proc-CFG-SeqConnect* c_1 *n* c_2)
from ‹$c_1 \vdash n -CEdge\ (p,\ es,\ rets)\rightarrow_p Exit$›
show *?case* **by**(*fastforce dest*:*Proc-CFG-Call-Labels*)
next
case (*Proc-CFG-SeqSecond* c_2 *n* n' c_1)
note *IH* = ‹$\bigwedge n''$. $c_2 \vdash n' -IEdge\ et\rightarrow_p n'' \Longrightarrow et = \Uparrow id$›
from ‹$c_1;;c_2 \vdash n' \oplus \#:c_1 -IEdge\ et\rightarrow_p n''$› ‹$c_2 \vdash n -CEdge\ (p,es,rets)\rightarrow_p n'$›
obtain *nx* **where** $c_2 \vdash n' -IEdge\ et\rightarrow_p nx$
apply − **apply**(*erule Proc-CFG.cases*,*auto*)
apply(*cases n'*) **apply**(*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
apply(*cases n'*) **apply**(*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
by(*cases n'*,*auto*,*case-tac n*,*auto*)
then show *?case* **by** (*rule IH*)
next
case (*Proc-CFG-CondThen* c_1 *n* n' *b* c_2)
note *IH* = ‹$\bigwedge n''$. $c_1 \vdash n' -IEdge\ et\rightarrow_p n'' \Longrightarrow et = \Uparrow id$›
from ‹*if* (*b*) c_1 *else* $c_2 \vdash n' \oplus 1 -IEdge\ et\rightarrow_p n''$› ‹$c_1 \vdash n -CEdge\ (p,es,rets)\rightarrow_p n'$›
‹$n \neq Entry$›
obtain *nx* **where** $c_1 \vdash n' -IEdge\ et\rightarrow_p nx$
apply − **apply**(*erule Proc-CFG.cases*,*auto*)
apply(*cases n'*) **apply** *auto* **apply**(*case-tac n*) **apply** *auto*
apply(*cases n'*) **apply** *auto*
by(*case-tac n*)(*auto dest*:*Proc-CFG-targetlabel-less-num-nodes*)
then show *?case* **by** (*rule IH*)
next
case (*Proc-CFG-CondElse* c_2 *n* n' *b* c_1)
note *IH* = ‹$\bigwedge n''$. $c_2 \vdash n' -IEdge\ et\rightarrow_p n'' \Longrightarrow et = \Uparrow id$›
from ‹*if* (*b*) c_1 *else* $c_2 \vdash n' \oplus \#:c_1 + 1 -IEdge\ et\rightarrow_p n''$› ‹$c_2 \vdash n -CEdge\ (p,es,rets)\rightarrow_p n'$›
obtain *nx* **where** $c_2 \vdash n' -IEdge\ et\rightarrow_p nx$
apply − **apply**(*erule Proc-CFG.cases*,*auto*)
apply(*cases n'*) **apply** *auto*
apply(*case-tac n*) **apply**(*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
by(*cases n'*,*auto*,*case-tac n*,*auto*)
then show *?case* **by** (*rule IH*)
next
case (*Proc-CFG-WhileBody* c' *n* n' *b*)
note *IH* = ‹$\bigwedge n''$. $c' \vdash n' -IEdge\ et\rightarrow_p n'' \Longrightarrow et = \Uparrow id$›
from ‹*while* (*b*) $c' \vdash n' \oplus 2 -IEdge\ et\rightarrow_p n''$› ‹$c' \vdash n -CEdge\ (p,es,rets)\rightarrow_p n'$›

```
  obtain nx where c' ⊢ n' −IEdge et→p nx
    apply − apply(erule Proc-CFG.cases,auto)
      apply(cases n') apply auto
     apply(cases n') apply auto apply(case-tac n) apply auto
    by(cases n',auto,case-tac n,auto)
  then show ?case by (rule IH)
next
  case (Proc-CFG-WhileBodyExit c' n et' b)
  from ‹c' ⊢ n −CEdge (p, es, rets)→p Exit›
  show ?case by(fastforce dest:Proc-CFG-Call-Labels)
next
  case Proc-CFG-Call
  from ‹Call p es rets ⊢ Label 1 −IEdge et→p n''› show ?case
    by(fastforce elim:Proc-CFG.cases)
qed


lemma Proc-CFG-edge-det:
  ⟦prog ⊢ n −et→p n'; prog ⊢ n −et'→p n'⟧ ⟹ et = et'
proof(induct rule:Proc-CFG.induct)
  case Proc-CFG-Entry-Exit thus ?case by(fastforce dest:Proc-CFG-EntryD)
next
  case Proc-CFG-Entry thus ?case by(fastforce dest:Proc-CFG-EntryD)
next
  case Proc-CFG-Skip thus ?case by(fastforce elim:Proc-CFG.cases)
next
  case Proc-CFG-LAss thus ?case by(fastforce elim:Proc-CFG.cases)
next
  case Proc-CFG-LAssSkip thus ?case by(fastforce elim:Proc-CFG.cases)
next
  case (Proc-CFG-SeqFirst c1 n et n' c2)
  note edge = ‹c1 ⊢ n −et→p n'›
  note IH = ‹c1 ⊢ n −et'→p n' ⟹ et = et'›
  from edge ‹n' ≠ Exit› obtain l where l:n' = Label l by (cases n') auto
 with edge have l < #:c1 by(fastforce intro:Proc-CFG-targetlabel-less-num-nodes)
  with ‹c1;;c2 ⊢ n −et'→p n'› l have c1 ⊢ n −et'→p n'
    by(fastforce elim:Proc-CFG.cases intro:Proc-CFG.intros dest:label-incr-ge)
  from IH[OF this] show ?case .
next
  case (Proc-CFG-SeqConnect c1 n et c2)
  note edge = ‹c1 ⊢ n −et→p Exit›
  note IH = ‹c1 ⊢ n −et'→p Exit ⟹ et = et'›
  from edge ‹n ≠ Entry› obtain l where l:n = Label l by (cases n) auto
 with edge have l < #:c1 by(fastforce intro: Proc-CFG-sourcelabel-less-num-nodes)
  with ‹c1;;c2 ⊢ n −et'→p Label #:c1› l have c1 ⊢ n −et'→p Exit
    by(fastforce elim:Proc-CFG.cases
              dest:Proc-CFG-targetlabel-less-num-nodes label-incr-ge)
  from IH[OF this] show ?case .
next
```

case (*Proc-CFG-SeqSecond* c_2 n et n' c_1)
note *edge* = ‹$c_2 \vdash n\ -et\rightarrow_p\ n'$›
note *IH* = ‹$c_2 \vdash n\ -et'\rightarrow_p\ n' \Longrightarrow et = et'$›
from *edge* ‹$n \neq$ *Entry*› **obtain** l **where** l:n = *Label* l **by** (*cases n*) *auto*
with *edge* **have** $l < \#$:c_2 **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
with ‹c_1;;$c_2 \vdash n \oplus \#$:$c_1\ -et'\rightarrow_p\ n' \oplus \#$:$c_1$› l **have** $c_2 \vdash n\ -et'\rightarrow_p\ n'$
by −(*erule Proc-CFG.cases*,
(*fastforce dest*:*Proc-CFG-sourcelabel-less-num-nodes label-incr-ge*
dest!:*label-incr-inj*)+)
from *IH*[*OF this*] **show** *?case* **.**
next
case *Proc-CFG-CondTrue* **thus** *?case* **by**(*fastforce elim*:*Proc-CFG.cases*)
next
case *Proc-CFG-CondFalse* **thus** *?case* **by**(*fastforce elim*:*Proc-CFG.cases*)
next
case (*Proc-CFG-CondThen* c_1 n et n' b c_2)
note *edge* = ‹$c_1 \vdash n\ -et\rightarrow_p\ n'$›
note *IH* = ‹$c_1 \vdash n\ -et'\rightarrow_p\ n' \Longrightarrow et = et'$›
from *edge* ‹$n \neq$ *Entry*› **obtain** l **where** l:n = *Label* l **by** (*cases n*) *auto*
with *edge* **have** $l < \#$:c_1 **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
with ‹*if* (*b*) c_1 *else* $c_2 \vdash n \oplus 1\ -et'\rightarrow_p\ n' \oplus 1$› l **have** $c_1 \vdash n\ -et'\rightarrow_p\ n'$
by −(*erule Proc-CFG.cases*,(*fastforce dest*:*label-incr-ge label-incr-inj*)+)
from *IH*[*OF this*] **show** *?case* **.**
next
case (*Proc-CFG-CondElse* c_2 n et n' b c_1)
note *edge* = ‹$c_2 \vdash n\ -et\rightarrow_p\ n'$›
note *IH* = ‹$c_2 \vdash n\ -et'\rightarrow_p\ n' \Longrightarrow et = et'$›
from *edge* ‹$n \neq$ *Entry*› **obtain** l **where** l:n = *Label* l **by** (*cases n*) *auto*
with *edge* **have** $l < \#$:c_2 **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
with ‹*if* (*b*) c_1 *else* $c_2 \vdash n \oplus (\#$:$c_1 + 1)\ -et'\rightarrow_p\ n' \oplus (\#$:$c_1 + 1)$› l
have $c_2 \vdash n\ -et'\rightarrow_p\ n'$
by −(*erule Proc-CFG.cases*,(*fastforce dest*:*Proc-CFG-sourcelabel-less-num-nodes*
label-incr-inj label-incr-ge label-incr-simp-rev)+)
from *IH*[*OF this*] **show** *?case* **.**
next
case *Proc-CFG-WhileTrue* **thus** *?case* **by**(*fastforce elim*:*Proc-CFG.cases*)
next
case *Proc-CFG-WhileFalse* **thus** *?case* **by**(*fastforce elim*:*Proc-CFG.cases*)
next
case *Proc-CFG-WhileFalseSkip* **thus** *?case* **by**(*fastforce elim*:*Proc-CFG.cases*)
next
case (*Proc-CFG-WhileBody* c' n et n' b)
note *edge* = ‹$c' \vdash n\ -et\rightarrow_p\ n'$›
note *IH* = ‹$c' \vdash n\ -et'\rightarrow_p\ n' \Longrightarrow et = et'$›
from *edge* ‹$n \neq$ *Entry*› **obtain** l **where** l:n = *Label* l **by** (*cases n*) *auto*
with *edge* **have** *less*:$l < \#$:c'
by(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
from *edge* ‹$n' \neq$ *Exit*› **obtain** l' **where** l':n' = *Label* l' **by** (*cases n'*) *auto*

with *edge* **have** $l' < \#{:}c'$ **by**(*fastforce intro*:*Proc-CFG-targetlabel-less-num-nodes*)
with ‹*while* (*b*) $c' \vdash n \oplus 2\ -et'\!\rightarrow_p\ n' \oplus 2$› *l less l'* **have** $c' \vdash n\ -et'\!\rightarrow_p\ n'$
by(*fastforce elim*:*Proc-CFG.cases dest*:*label-incr-start-Node-smaller*)
from *IH*[*OF this*] **show** *?case* **.**
next
case (*Proc-CFG-WhileBodyExit c' n et b*)
note *edge* = ‹$c' \vdash n\ -et\!\rightarrow_p$ *Exit*›
note *IH* = ‹$c' \vdash n\ -et'\!\rightarrow_p$ *Exit* $\Longrightarrow$ *et* = *et'*›
from *edge* ‹$n \neq$ *Entry*› **obtain** *l* **where** *l*:n = *Label l* **by** (*cases n*) *auto*
with *edge* **have** $l < \#{:}c'$ **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
with ‹*while* (*b*) $c' \vdash n \oplus 2\ -et'\!\rightarrow_p$ *Label 0*› *l* **have** $c' \vdash n\ -et'\!\rightarrow_p$ *Exit*
by −(*erule Proc-CFG.cases*,*auto dest*:*label-incr-start-Node-smaller*)
from *IH*[*OF this*] **show** *?case* **.**
next
case *Proc-CFG-Call* **thus** *?case* **by**(*fastforce elim*:*Proc-CFG.cases*)
next
case *Proc-CFG-CallSkip* **thus** *?case* **by**(*fastforce elim*:*Proc-CFG.cases*)
qed

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(*induct arbitrary*:$n_2\ n_2'$ *rule*:*Proc-CFG.induct*)
case (*Proc-CFG-Entry-Exit prog*)
from ‹*prog* $\vdash n_2\ -et_2\!\rightarrow_p\ n_2'$› ‹*Entry* = n_2› ‹*Exit* $\neq n_2'$›
have $et_2 = IEdge\ (\lambda s.\ True)_{\surd}$ **by**(*fastforce dest*:*Proc-CFG-EntryD*)
thus *?case* **by** *simp*
next
case (*Proc-CFG-Entry prog*)
from ‹*prog* $\vdash n_2\ -et_2\!\rightarrow_p\ n_2'$› ‹*Entry* = n_2› ‹*Label 0* $\neq n_2'$›
have $et_2 = IEdge\ (\lambda s.\ False)_{\surd}$ **by**(*fastforce dest*:*Proc-CFG-EntryD*)
thus *?case* **by** *simp*
next
case *Proc-CFG-Skip*
from ‹*Skip* $\vdash n_2\ -et_2\!\rightarrow_p\ n_2'$› ‹*Label 0* = n_2› ‹*Exit* $\neq n_2'$›
have *False* **by**(*fastforce elim*:*Proc-CFG.cases*)
thus *?case* **by** *simp*
next
case (*Proc-CFG-LAss V e*)
from ‹$V{:=}e \vdash n_2\ -et_2\!\rightarrow_p\ n_2'$› ‹*Label 0* = n_2› ‹*Label 1* $\neq n_2'$›
have *False* **by** −(*erule Proc-CFG.cases*,*auto*)
thus *?case* **by** *simp*
next
case (*Proc-CFG-LAssSkip V e*)
from ‹$V{:=}e \vdash n_2\ -et_2\!\rightarrow_p\ n_2'$› ‹*Label 1* = n_2› ‹*Exit* $\neq n_2'$›
have *False* **by** −(*erule Proc-CFG.cases*,*auto*)
thus *?case* **by** *simp*

next
case (*Proc-CFG-SeqFirst* c_1 n *et* n' c_2)
note *IH* = ‹$\bigwedge n_2\ {n_2}'.\ [\![c_1 \vdash n_2 -et_2\to_p {n_2}';\ n = n_2;\ n' \neq n_2]\!]$
$\Longrightarrow \exists Q\ Q'.\ et = IEdge\ (Q)_\surd \wedge et_2 = IEdge\ (Q')_\surd \wedge$
$(\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$›
from ‹$c_1;;c_2 \vdash n_2 -et_2\to_p {n_2}'$› ‹$c_1 \vdash n -et\to_p {n'}$› ‹$n = n_2$› ‹$n' \neq {n_2}'$›
have $c_1 \vdash n_2 -et_2\to_p {n_2}' \vee (c_1 \vdash n_2 -et_2\to_p Exit \wedge {n_2}' = Label\ \#{:}c_1)$
apply − **apply**(*erule Proc-CFG.cases*)
apply(*auto intro*:*Proc-CFG.intros*)
by(*case-tac n*,*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)+
thus *?case*
proof
assume $c_1 \vdash n_2 -et_2\to_p {n_2}'$
from *IH*[*OF this* ‹$n = n_2$› ‹$n' \neq {n_2}'$›] **show** *?case* **.**
next
assume $c_1 \vdash n_2 -et_2\to_p Exit \wedge {n_2}' = Label\ \#{:}c_1$
hence *edge*:$c_1 \vdash n_2 -et_2\to_p Exit$ **and** *n2'*:${n_2}' = Label\ \#{:}c_1$ **by** *simp-all*
from *IH*[*OF edge* ‹$n = n_2$› ‹$n' \neq Exit$›] **show** *?case* **.**
qed
next
case (*Proc-CFG-SeqConnect* c_1 n *et* c_2)
note *IH* = ‹$\bigwedge n_2\ {n_2}'.\ [\![c_1 \vdash n_2 -et_2\to_p {n_2}';\ n = n_2;\ Exit \neq {n_2}']\!]$
$\Longrightarrow \exists Q\ Q'.\ et = IEdge\ (Q)_\surd \wedge et_2 = IEdge\ (Q')_\surd \wedge$
$(\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$›
from ‹$c_1;;c_2 \vdash n_2 -et_2\to_p {n_2}'$› ‹$c_1 \vdash n -et\to_p Exit$› ‹$n = n_2$› ‹$n \neq Entry$›
‹$Label\ \#{:}c_1 \neq {n_2}'$› **have** $c_1 \vdash n_2 -et_2\to_p {n_2}' \wedge Exit \neq {n_2}'$
apply − **apply**(*erule Proc-CFG.cases*)
apply(*auto intro*:*Proc-CFG.intros*)
by(*case-tac n*,*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)+
from *IH*[*OF this*[*THEN conjunct1*] ‹$n = n_2$› *this*[*THEN conjunct2*]]
show *?case* **.**
next
case (*Proc-CFG-SeqSecond* c_2 n *et* n' c_1)
note *IH* = ‹$\bigwedge n_2\ {n_2}'.\ [\![c_2 \vdash n_2 -et_2\to_p {n_2}';\ n = n_2;\ n' \neq n_2]\!]$
$\Longrightarrow \exists Q\ Q'.\ et = IEdge\ (Q)_\surd \wedge et_2 = IEdge\ (Q')_\surd \wedge$
$(\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$›
from ‹$c_1;;c_2 \vdash n_2 -et_2\to_p {n_2}'$› ‹$c_2 \vdash n -et\to_p {n'}$› ‹$n \oplus \#{:}c_1 = n_2$›
‹$n' \oplus \#{:}c_1 \neq {n_2}'$› ‹$n \neq Entry$›
obtain *nx* **where** $c_2 \vdash n -et_2\to_p nx \wedge nx \oplus \#{:}c_1 = {n_2}'$
apply − **apply**(*erule Proc-CFG.cases*)
apply(*auto intro*:*Proc-CFG.intros*)
apply(*cases n*,*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
apply(*cases n*,*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
by(*fastforce dest*:*label-incr-inj*)
with ‹$n' \oplus \#{:}c_1 \neq {n_2}'$› **have** *edge*:$c_2 \vdash n -et_2\to_p nx$ **and** *neq*:$n' \neq nx$
by *auto*
from *IH*[*OF edge - neq*] **show** *?case* **by** *simp*
next
case (*Proc-CFG-CondTrue* b c_1 c_2)

from ‹*if* (*b*) c_1 *else* $c_2 \vdash n_2 -et_2\rightarrow_p n_2'$› ‹*Label 0* = n_2› ‹*Label 1* ≠ n_2'›
show *?case* **by** −(*erule Proc-CFG.cases*,*auto*)
next
case (*Proc-CFG-CondFalse b* c_1 c_2)
from ‹*if* (*b*) c_1 *else* $c_2 \vdash n_2 -et_2\rightarrow_p n_2'$› ‹*Label 0* = n_2› ‹*Label* (#:c_1 + *1*) ≠ n_2'›
show *?case* **by** −(*erule Proc-CFG.cases*,*auto*)
next
case (*Proc-CFG-CondThen* c_1 *n et* n' *b* c_2)
note *IH* = ‹$\bigwedge n_2\ n_2'.\ [\![c_1 \vdash n_2 -et_2\rightarrow_p n_2'; n = n_2; n' \neq n_2]\!] \Longrightarrow \exists Q\ Q'.\ et = IEdge\ (Q)_{\surd} \wedge et_2 = IEdge\ (Q')_{\surd} \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$›
from ‹*if* (*b*) c_1 *else* $c_2 \vdash n_2 -et_2\rightarrow_p n_2'$› ‹$c_1 \vdash n -et\rightarrow_p n'$› ‹$n \neq$ *Entry*› ‹$n \oplus 1 = n_2$› ‹$n' \oplus 1 \neq n_2'$›
obtain *nx* **where** $c_1 \vdash n -et_2\rightarrow_p nx \wedge n' \neq nx$
apply − **apply**(*erule Proc-CFG.cases*)
apply(*auto intro*:*Proc-CFG.intros simp del*:*One-nat-def*)
apply(*drule label-incr-inj*) **apply**(*auto simp del*:*One-nat-def*)
apply(*drule label-incr-simp-rev*[*OF sym*])
by(*case-tac na*,*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
from *IH*[*OF this*[*THEN conjunct1*] - *this*[*THEN conjunct2*]] **show** *?case* **by** *simp*
next
case (*Proc-CFG-CondElse* c_2 *n et* n' *b* c_1)
note *IH* = ‹$\bigwedge n_2\ n_2'.\ [\![c_2 \vdash n_2 -et_2\rightarrow_p n_2'; n = n_2; n' \neq n_2]\!] \Longrightarrow \exists Q\ Q'.\ et = IEdge\ (Q)_{\surd} \wedge et_2 = IEdge\ (Q')_{\surd} \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$›
from ‹*if* (*b*) c_1 *else* $c_2 \vdash n_2 -et_2\rightarrow_p n_2'$› ‹$c_2 \vdash n -et\rightarrow_p n'$› ‹$n \neq$ *Entry*› ‹$n \oplus$ #:c_1 + *1* = n_2› ‹$n' \oplus$ #:c_1 + *1* ≠ n_2'›
obtain *nx* **where** $c_2 \vdash n -et_2\rightarrow_p nx \wedge n' \neq nx$
apply − **apply**(*erule Proc-CFG.cases*)
apply(*auto intro*:*Proc-CFG.intros simp del*:*One-nat-def*)
apply(*drule label-incr-simp-rev*)
apply(*case-tac na*,*auto*,*cases n*,*auto dest*:*Proc-CFG-sourcelabel-less-num-nodes*)
by(*fastforce dest*:*label-incr-inj*)
from *IH*[*OF this*[*THEN conjunct1*] - *this*[*THEN conjunct2*]] **show** *?case* **by** *simp*
next
case (*Proc-CFG-WhileTrue b c'*)
from ‹*while* (*b*) $c' \vdash n_2 -et_2\rightarrow_p n_2'$› ‹*Label 0* = n_2› ‹*Label 2* ≠ n_2'›
show *?case* **by** −(*erule Proc-CFG.cases*,*auto*)
next
case (*Proc-CFG-WhileFalse b c'*)
from ‹*while* (*b*) $c' \vdash n_2 -et_2\rightarrow_p n_2'$› ‹*Label 0* = n_2› ‹*Label 1* ≠ n_2'›
show *?case* **by** −(*erule Proc-CFG.cases*,*auto*)
next
case (*Proc-CFG-WhileFalseSkip b c'*)
from ‹*while* (*b*) $c' \vdash n_2 -et_2\rightarrow_p n_2'$› ‹*Label 1* = n_2› ‹*Exit* ≠ n_2'›
show *?case* **by** −(*erule Proc-CFG.cases*,*auto dest*:*label-incr-ge*)

next
case (*Proc-CFG-WhileBody c′ n et n′ b*)
note *IH* = ‹$\bigwedge n_2\ n_2'.\ [\![c' \vdash n_2 -et_2\to_p n_2';\ n = n_2;\ n' \neq n_2']\!] \Longrightarrow \exists Q\ Q'.\ et = IEdge\ (Q)_\surd \wedge et_2 = IEdge\ (Q')_\surd \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$›
from ‹*while* (*b*) *c′* ⊢ *n*₂ −*et*₂→ₚ *n*₂′› ‹*c′* ⊢ *n* −*et*→ₚ *n′*› ‹*n* ≠ *Entry*› ‹*n′* ≠ *Exit*› ‹*n* ⊕ 2 = *n*₂› ‹*n′* ⊕ 2 ≠ *n*₂′›
obtain *nx* **where** *c′* ⊢ *n* −*et*₂→ₚ *nx* ∧ *n′* ≠ *nx*
apply − **apply**(*erule Proc-CFG.cases*)
apply(*auto intro*:*Proc-CFG.intros*)
apply(*fastforce dest*:*label-incr-ge*[*OF sym*])
apply(*fastforce dest*:*label-incr-inj*)
by(*fastforce dest*:*label-incr-inj*)
from *IH*[*OF this*[*THEN conjunct1*] - *this*[*THEN conjunct2*]] **show** *?case* **by** *simp*
next
case (*Proc-CFG-WhileBodyExit c′ n et b*)
note *IH* = ‹$\bigwedge n_2\ n_2'.\ [\![c' \vdash n_2 -et_2\to_p n_2';\ n = n_2;\ Exit \neq n_2']\!] \Longrightarrow \exists Q\ Q'.\ et = IEdge\ (Q)_\surd \wedge et_2 = IEdge\ (Q')_\surd \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$›
from ‹*while* (*b*) *c′* ⊢ *n*₂ −*et*₂→ₚ *n*₂′› ‹*c′* ⊢ *n* −*et*→ₚ *Exit*› ‹*n* ≠ *Entry*› ‹*n* ⊕ 2 = *n*₂› ‹*Label* 0 ≠ *n*₂′›
obtain *nx* **where** *c′* ⊢ *n* −*et*₂→ₚ *nx* ∧ *Exit* ≠ *nx*
apply − **apply**(*erule Proc-CFG.cases*)
apply(*auto intro*:*Proc-CFG.intros*)
apply(*fastforce dest*:*label-incr-ge*[*OF sym*])
by(*fastforce dest*:*label-incr-inj*)
from *IH*[*OF this*[*THEN conjunct1*] - *this*[*THEN conjunct2*]] **show** *?case* **by** *simp*
next
case *Proc-CFG-Call* **thus** *?case* **by** −(*erule Proc-CFG.cases*,*auto*)
next
case *Proc-CFG-CallSkip* **thus** *?case* **by** −(*erule Proc-CFG.cases*,*auto*)
qed

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* $\longleftrightarrow$ *False*
| *containsCall procs* (*V*:=*e*) *ps p* $\longleftrightarrow$ *False*
| *containsCall procs* (*c*$_1$;;*c*$_2$) *ps p* $\longleftrightarrow$
containsCall procs c$_1$ *ps p* $\vee$ *containsCall procs c*$_2$ *ps p*
| *containsCall procs* (*if* (*b*) *c*$_1$ *else c*$_2$) *ps p* $\longleftrightarrow$
containsCall procs c$_1$ *ps p* $\vee$ *containsCall procs c*$_2$ *ps p*
| *containsCall procs* (*while* (*b*) *c*) *ps p* $\longleftrightarrow$
containsCall procs c ps p
| *containsCall procs* (*Call q es' rets'*) *ps p* $\longleftrightarrow$ *p* = *q* $\wedge$ *ps* = [] $\vee$
($\exists$ *ins outs c ps'*. *ps* = *q*#*ps'* $\wedge$ (*q*,*ins*,*outs*,*c*) $\in$ *set procs* $\wedge$
containsCall procs c ps' p)
by *pat-completeness auto*
termination *containsCall*
by(*relation measures* [λ(*procs*,*c*,*ps*,*p*). *length ps*,
λ(*procs*,*c*,*ps*,*p*). *size c*]) *auto*

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

lemma *containsCallcases*:
containsCall procs prog ps p
$\Longrightarrow$ *ps* = [] $\wedge$ *containsCall procs prog ps p* $\vee$
($\exists$ *q ins outs c ps'*. *ps* = *ps'*@[*q*] $\wedge$ (*q*,*ins*,*outs*,*c*) $\in$ *set procs* $\wedge$
containsCall procs c [] *p* $\wedge$ *containsCall procs prog ps' q*)
proof(*induct procs prog ps p rule*:*containsCall-induct*)
case (*Call procs q es' rets' ps p*)
note *IH* = ‹$\bigwedge$*x y z ps'*. ⟦*ps* = *q*#*ps'*; (*q*,*x*,*y*,*z*) $\in$ *set procs*;
containsCall procs z ps' p⟧
$\Longrightarrow$ *ps'* = [] $\wedge$ *containsCall procs z ps' p* $\vee$
($\exists$ *qx ins outs c psx*. *ps'* = *psx*@[*qx*] $\wedge$ (*qx*,*ins*,*outs*,*c*) $\in$ *set procs* $\wedge$
containsCall procs c [] *p* $\wedge$
containsCall procs z psx qx)›
from ‹*containsCall procs* (*Call q es' rets'*) *ps p*›
have *p* = *q* $\wedge$ *ps* = [] $\vee$
($\exists$ *ins outs c ps'*. *ps* = *q*#*ps'* $\wedge$ (*q*,*ins*,*outs*,*c*) $\in$ *set procs* $\wedge$
containsCall procs c ps' p) **by** *simp*
thus *?case*
proof
assume *assms*:*p* = *q* $\wedge$ *ps* = []
hence *containsCall procs* (*Call q es' rets'*) *ps p* **by** *simp*
with *assms* **show** *?thesis* **by** *simp*
next
assume $\exists$ *ins outs c ps'*. *ps* = *q*#*ps'* $\wedge$ (*q*,*ins*,*outs*,*c*) $\in$ *set procs* $\wedge$
containsCall procs c ps' p
then obtain *ins outs c ps'* **where** *ps* = *q*#*ps'* **and** (*q*,*ins*,*outs*,*c*) $\in$ *set procs*
and *containsCall procs c ps' p* **by** *blast*
from *IH*[*OF this*] **have** *ps'* = [] $\wedge$ *containsCall procs c ps' p* $\vee$

```
    (∃ qx insx outsx cx psx.
       ps' = psx @ [qx] ∧ (qx,insx,outsx,cx) ∈ set procs ∧
       containsCall procs cx [] p ∧ containsCall procs c psx qx) .
  thus ?thesis
  proof
    assume assms:ps' = [] ∧ containsCall procs c ps' p
    have containsCall procs (Call q es' rets') [] q by simp
   with assms ‹ps = q#ps'› ‹(q,ins,outs,c) ∈ set procs› show ?thesis by fastforce
  next
    assume ∃ qx insx outsx cx psx.
      ps' = psx@[qx] ∧ (qx,insx,outsx,cx) ∈ set procs ∧
      containsCall procs cx [] p ∧ containsCall procs c psx qx
    then obtain qx insx outsx cx psx
      where ps' = psx@[qx] and (qx,insx,outsx,cx) ∈ set procs
      and containsCall procs cx [] p
      and containsCall procs c psx qx by blast
    from ‹(q,ins,outs,c) ∈ set procs› ‹containsCall procs c psx qx›
    have containsCall procs (Call q es' rets') (q#psx) qx by fastforce
    with ‹ps' = psx@[qx]› ‹ps = q#ps'› ‹(qx,insx,outsx,cx) ∈ set procs›
      ‹containsCall procs cx [] p› show ?thesis by fastforce
  qed
 qed
qed auto



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
 by(auto dest:containsCallcases)



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(induct procs prog qs q rule:containsCall-induct)
 case (Call procs qx esx retsx ps p')
 note IH = ‹⋀x y z psx. ⟦ps = qx#psx; (qx,x,y,z) ∈ set procs;
   containsCall procs z psx p'; (p',ins,outs,c) ∈ set procs;
   containsCall procs c [] p⟧ ⟹ containsCall procs z (psx@[p']) p›
 from ‹containsCall procs (Call qx esx retsx) ps p'›
 have p' = qx ∧ ps = [] ∨
   (∃ insx outsx cx psx. ps = qx#psx ∧ (qx,insx,outsx,cx) ∈ set procs ∧
   containsCall procs cx psx p') by simp
 thus ?case
```

proof
assume *assms*:$p' = qx \wedge ps = []$
with ‹$(p', ins, outs, c) \in set\ procs$› ‹$containsCall\ procs\ c\ []\ p$›
have $containsCall\ procs\ (Call\ qx\ esx\ retsx)\ [p']\ p$ **by** *fastforce*
with *assms* **show** *?thesis* **by** *simp*
next
assume $\exists insx\ outsx\ cx\ psx.\ ps = qx\#psx \wedge (qx,insx,outsx,cx) \in set\ procs \wedge$
$containsCall\ procs\ cx\ psx\ p'$
then obtain $insx\ outsx\ cx\ psx$ **where** $ps = qx\#psx$
and $(qx,insx,outsx,cx) \in set\ procs$
and $containsCall\ procs\ cx\ psx\ p'$ **by** *blast*
from *IH*[*OF this* ‹$(p', ins, outs, c) \in set\ procs$›
‹$containsCall\ procs\ c\ []\ p$›]
have $containsCall\ procs\ cx\ (psx\ @\ [p'])\ p$ **.**
with ‹$ps = qx\#psx$› ‹$(qx,insx,outsx,cx) \in set\ procs$›
show *?thesis* **by** *fastforce*
qed
qed *auto*

lemma *containsCall-indirection*:
⟦$containsCall\ procs\ prog\ qs\ q$; $containsCall\ procs\ c\ ps\ p$;
$(q,ins,outs,c) \in set\ procs$⟧
⟹ $containsCall\ procs\ prog\ (qs@q\#ps)\ p$
proof(*induct procs prog qs q rule*:*containsCall-induct*)
case $(Call\ procs\ px\ esx\ retsx\ ps'\ p')$
note *IH* = ‹$\bigwedge x\ y\ z\ psx.$ ⟦$ps' = px\ \#\ psx$; $(px, x, y, z) \in set\ procs$;
$containsCall\ procs\ z\ psx\ p'$; $containsCall\ procs\ c\ ps\ p$;
$(p', ins, outs, c) \in set\ procs$⟧
⟹ $containsCall\ procs\ z\ (psx\ @\ p'\ \#\ ps)\ p$›
from ‹$containsCall\ procs\ (Call\ px\ esx\ retsx)\ ps'\ p'$›
have $p' = px \wedge ps' = [] \vee$
$(\exists insx\ outsx\ cx\ psx.\ ps' = px\#psx \wedge (px,insx,outsx,cx) \in set\ procs \wedge$
$containsCall\ procs\ cx\ psx\ p')$ **by** *simp*
thus *?case*
proof
assume $p' = px \wedge ps' = []$
with ‹$containsCall\ procs\ c\ ps\ p$› ‹$(p', ins, outs, c) \in set\ procs$›
show *?thesis* **by** *fastforce*
next
assume $\exists insx\ outsx\ cx\ psx.\ ps' = px\#psx \wedge (px,insx,outsx,cx) \in set\ procs \wedge$
$containsCall\ procs\ cx\ psx\ p'$
then obtain $insx\ outsx\ cx\ psx$ **where** $ps' = px\#psx$
and $(px,insx,outsx,cx) \in set\ procs$
and $containsCall\ procs\ cx\ psx\ p'$ **by** *blast*
from *IH*[*OF this* ‹$containsCall\ procs\ c\ ps\ p$›
‹$(p', ins, outs, c) \in set\ procs$›]
have $containsCall\ procs\ cx\ (psx\ @\ p'\ \#\ ps)\ p$ **.**
with ‹$ps' = px\#psx$› ‹$(px,insx,outsx,cx) \in set\ procs$›

show *?thesis* **by** *fastforce*
qed
qed *auto*

lemma *Proc-CFG-Call-containsCall*:
prog ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *n'* ⟹ *containsCall procs prog* [] *p*
by(*induct prog n et*≡*CEdge* (*p*,*es*,*rets*) *n' rule*:*Proc-CFG.induct*,*auto*)

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(*atomize-elim*)
from ‹*containsCall procs prog* [] *p*›
show ∃ *l es rets l'*. *prog* ⊢ *Label l* −*CEdge* (*p*,*es*,*rets*)→$_p$ *Label l'*
proof(*induct procs prog ps*≡[]::*pname list p rule*:*containsCall-induct*)
case *Seq* **thus** *?case*
by *auto*(*fastforce dest*:*Proc-CFG-SeqFirst*,*fastforce dest*:*Proc-CFG-SeqSecond*)
next
case *Cond* **thus** *?case*
by *auto*(*fastforce dest*:*Proc-CFG-CondThen*,*fastforce dest*:*Proc-CFG-CondElse*)
next
case *While* **thus** *?case* **by**(*fastforce dest*:*Proc-CFG-WhileBody*)
next
case *Call* **thus** *?case* **by**(*fastforce intro*:*Proc-CFG-Call*)
qed *auto*
qed

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'*;

$\mathit{containsCall\ procs\ prog\ ps\ p}\rrbracket$
$\Longrightarrow \mathit{prog,procs} \vdash (p,n)\ -\mathit{et}\rightarrow\ (p,n')$

| *MainCall*:
$\llbracket \mathit{prog} \vdash \mathit{Label\ l}\ -\mathit{CEdge}\ (p,es,rets)\rightarrow_p\ n';\ (p,ins,outs,c) \in \mathit{set\ procs}\rrbracket$
$\Longrightarrow \mathit{prog,procs} \vdash (\mathit{Main,Label\ l})$
$-(\lambda s.\ \mathit{True}):(\mathit{Main},n')\hookrightarrow_p \mathit{map}\ (\lambda e\ cf.\ \mathit{interpret\ e\ cf})\ es\rightarrow\ (p,\mathit{Entry})$

| *ProcCall*:
$\llbracket (p,ins,outs,c) \in \mathit{set\ procs};\ c \vdash \mathit{Label\ l}\ -\mathit{CEdge}\ (p',es',rets')\rightarrow_p\ \mathit{Label\ l'};$
$(p',ins',outs',c') \in \mathit{set\ procs};\ \mathit{containsCall\ procs\ prog\ ps\ p}\rrbracket$
$\Longrightarrow \mathit{prog,procs} \vdash (p,\mathit{Label\ l})$
$-(\lambda s.\ \mathit{True}):(p,\mathit{Label\ l'})\hookrightarrow_{p'} \mathit{map}\ (\lambda e\ cf.\ \mathit{interpret\ e\ cf})\ es'\rightarrow\ (p',\mathit{Entry})$

| *MainReturn*:
$\llbracket \mathit{prog} \vdash \mathit{Label\ l}\ -\mathit{CEdge}\ (p,es,rets)\rightarrow_p\ \mathit{Label\ l'};\ (p,ins,outs,c) \in \mathit{set\ procs}\rrbracket$
$\Longrightarrow \mathit{prog,procs} \vdash (p,\mathit{Exit})\ -(\lambda cf.\ \mathit{snd\ cf} = (\mathit{Main,Label\ l'}))\hookleftarrow_p$
$(\lambda cf\ cf'.\ cf'(\mathit{rets}\ [:=]\ \mathit{map\ cf\ outs}))\rightarrow\ (\mathit{Main,Label\ l'})$

| *ProcReturn*:
$\llbracket (p,ins,outs,c) \in \mathit{set\ procs};\ c \vdash \mathit{Label\ l}\ -\mathit{CEdge}\ (p',es',rets')\rightarrow_p\ \mathit{Label\ l'};$
$(p',ins',outs',c') \in \mathit{set\ procs};\ \mathit{containsCall\ procs\ prog\ ps\ p}\rrbracket$
$\Longrightarrow \mathit{prog,procs} \vdash (p',\mathit{Exit})\ -(\lambda cf.\ \mathit{snd\ cf} = (p,\mathit{Label\ l'}))\hookleftarrow_{p'}$
$(\lambda cf\ cf'.\ cf'(\mathit{rets'}\ [:=]\ \mathit{map\ cf\ outs'}))\rightarrow\ (p,\mathit{Label\ l'})$

| *MainCallReturn*:
$\mathit{prog} \vdash n\ -\mathit{CEdge}\ (p,es,rets)\rightarrow_p\ n'$
$\Longrightarrow \mathit{prog,procs} \vdash (\mathit{Main},n)\ -(\lambda s.\ \mathit{False})_{\surd}\rightarrow\ (\mathit{Main},n')$

| *ProcCallReturn*:
$\llbracket (p,ins,outs,c) \in \mathit{set\ procs};\ c \vdash n\ -\mathit{CEdge}\ (p',es',rets')\rightarrow_p\ n';$
$\mathit{containsCall\ procs\ prog\ ps\ p}\rrbracket$
$\Longrightarrow \mathit{prog,procs} \vdash (p,n)\ -(\lambda s.\ \mathit{False})_{\surd}\rightarrow\ (p,n')$

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*
  by(*fastforce simp*:*well-formed-def wf-proc-def*)

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'*
  apply(*auto simp*:*well-formed-def distinct-fst-def distinct-map inj-on-def*)
by(*erule-tac x*=(*p*,*ins*,*outs*,*c*) **in** *ballE*,*auto*)+

lemma *PCFG-sourcelabel-None-less-num-nodes*:
  ⟦*prog*,*procs* ⊢ (*Main*,*Label l*) −*et*→ *n'*; *well-formed procs*⟧ ⟹ *l* < #:*prog*
proof(*induct* (*Main*,*Label l*) *et n'*
    arbitrary:*l rule*:*PCFG.induct*)
  case (*Main et n'*)
  from ‹*prog* ⊢ *Label l* −*IEdge et*→$_p$ *n'*›
  show *?case* **by**(*fastforce elim*:*Proc-CFG-sourcelabel-less-num-nodes*)
next
  case (*MainCall l p es rets n' ins outs c*)
  from ‹*prog* ⊢ *Label l* −*CEdge* (*p*,*es*,*rets*)→$_p$ *n'*›
  show *?case* **by**(*fastforce elim*:*Proc-CFG-sourcelabel-less-num-nodes*)
next
  case (*MainCallReturn p es rets n' l*)
  from ‹*prog* ⊢ *Label l* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n'*›
  show *?case* **by**(*fastforce elim*:*Proc-CFG-sourcelabel-less-num-nodes*)
qed *auto*

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(*induct* (*p*,*Label l*) *et n' arbitrary*:*l rule*:*PCFG.induct*)
  case (*Proc ins' outs' c' et n'*)
  from ‹*c'* ⊢ *Label l* −*IEdge et*→$_p$ *n'*› **have** *l* < #:*c'*
    by(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
  with ‹*well-formed procs*› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
    ‹(*p*,*ins'*,*outs'*,*c'*) ∈ *set procs*›
  show *?case* **by** *fastforce*
next
  case (*ProcCall ins' outs' c' l' p' es rets l'' ins'' outs'' c'' ps*)
  from ‹*c'* ⊢ *Label l'* −*CEdge* (*p'*,*es*,*rets*)→$_p$ *Label l''*› **have** *l'* < #:*c'*
    by(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
  with ‹*well-formed procs*› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
    ‹(*p*, *ins'*, *outs'*, *c'*) ∈ *set procs*›
  show *?case* **by** *fastforce*
next

case (*ProcCallReturn ins′ outs′ c′ p′ es rets n′*)
from ‹*c′* ⊢ *Label l* −*CEdge* (*p′*, *es*, *rets*)→$_p$ *n′*› **have** *l* < #:*c′*
by(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
with ‹*well-formed procs*› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
‹(*p*,*ins′*,*outs′*,*c′*) ∈ *set procs*›
show *?case* **by** *fastforce*
qed *auto*

lemma *Proc-CFG-targetlabel-Main-less-num-nodes*:
⟦*prog*,*procs* ⊢ *n* −*et*→ (*Main*,*Label l*); *well-formed procs*⟧ ⟹ *l* < #:*prog*
proof(*induct n et* (*Main*,*Label l*)
arbitrary:*l rule*:*PCFG.induct*)
case (*Main n et*)
from ‹*prog* ⊢ *n* −*IEdge et*→$_p$ *Label l*›
show *?case* **by**(*fastforce elim*:*Proc-CFG-targetlabel-less-num-nodes*)
next
case (*MainReturn l′ p es rets l″ ins outs c*)
from ‹*prog* ⊢ *Label l′* −*CEdge* (*p*,*es*,*rets*)→$_p$ *Label l″*›
show *?case* **by**(*fastforce elim*:*Proc-CFG-targetlabel-less-num-nodes*)
next
case (*MainCallReturn n p es rets*)
from ‹*prog* ⊢ *n* −*CEdge* (*p*, *es*, *rets*)→$_p$ *Label l*›
show *?case* **by**(*fastforce elim*:*Proc-CFG-targetlabel-less-num-nodes*)
qed *auto*

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(*induct n et* (*p*,*Label l*) *arbitrary*:*l rule*:*PCFG.induct*)
case (*Proc ins′ outs′ c′ n et*)
from ‹*c′* ⊢ *n* −*IEdge et*→$_p$ *Label l*› **have** *l* < #:*c′*
by(*fastforce intro*:*Proc-CFG-targetlabel-less-num-nodes*)
with ‹*well-formed procs*› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
‹(*p*,*ins′*,*outs′*,*c′*) ∈ *set procs*›
show *?case* **by** *fastforce*
next
case (*ProcReturn ins′ outs′ c′ l′ p′ es rets l ins″ outs″ c″ ps*)
from ‹*c′* ⊢ *Label l′* −*CEdge* (*p′*,*es*,*rets*)→$_p$ *Label l*› **have** *l* < #:*c′*
by(*fastforce intro*:*Proc-CFG-targetlabel-less-num-nodes*)
with ‹*well-formed procs*› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
‹(*p*, *ins′*, *outs′*, *c′*) ∈ *set procs*›
show *?case* **by** *fastforce*
next
case (*ProcCallReturn ins′ outs′ c′ n p″ es rets*)
from ‹*c′* ⊢ *n* −*CEdge* (*p″*, *es*, *rets*)→$_p$ *Label l*› **have** *l* < #:*c′*
by(*fastforce intro*:*Proc-CFG-targetlabel-less-num-nodes*)
with ‹*well-formed procs*› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›

```
    ‹(p,ins',outs',c') ∈ set procs›
  show ?case by fastforce
qed auto


lemma Proc-CFG-edge-det:
  ⟦prog,procs ⊢ n −et→ n'; prog,procs ⊢ n −et'→ n'; well-formed procs⟧
  ⟹ et = et'
proof(induct rule:PCFG.induct)
  case Main thus ?case by(auto elim:PCFG.cases dest:Proc-CFG-edge-det)
next
  case Proc thus ?case by(auto elim:PCFG.cases dest:Proc-CFG-edge-det)
next
  case (MainCall l p es rets n' ins outs c)
  from ‹prog,procs ⊢ (Main,Label l) −et'→ (p,Entry)› ‹well-formed procs›
  obtain es' rets' n'' ins' outs' c'
    where prog ⊢ Label l −CEdge (p,es',rets')→p n''
    and (p,ins',outs',c') ∈ set procs
    and et' = (λs. True):(Main,n'')↪pmap (λe cf. interpret e cf) es'
    by(auto elim:PCFG.cases)
  from ‹(p,ins,outs,c) ∈ set procs› ‹(p,ins',outs',c') ∈ set procs›
    ‹well-formed procs›
  have ins = ins' by fastforce
  from ‹prog ⊢ Label l −CEdge (p,es,rets)→p n'›
    ‹prog ⊢ Label l −CEdge (p,es',rets')→p n''›
  have es = es' and n' = n'' by(auto dest:Proc-CFG-Call-nodes-eq)
  with ‹et' = (λs. True):(Main,n'')↪pmap (λe cf. interpret e cf) es'› ‹ins = ins'›
  show ?case by simp
next
  case (ProcCall p ins outs c l p' es' rets' l' ins' outs' c' ps)
  from ‹prog,procs ⊢ (p,Label l) −et'→ (p',Entry)› ‹(p',ins',outs',c') ∈ set procs›
    ‹(p, ins, outs, c) ∈ set procs› ‹well-formed procs›
    ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'›
  show ?case
  proof(induct (p,Label l) et' (p',Entry) rule:PCFG.induct)
    case (ProcCall insx outsx cx es'x rets'x l'x ins'x outs'x c'x ps)
    from ‹well-formed procs› ‹(p, insx, outsx, cx) ∈ set procs›
      ‹(p, ins, outs, c) ∈ set procs›
    have [simp]:cx = c by auto
    from ‹cx ⊢ Label l −CEdge (p', es'x, rets'x)→p Label l'x›
      ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'›
    have [simp]:es'x = es' l'x = l' by(auto dest:Proc-CFG-Call-nodes-eq)
    show ?case by simp
  qed auto
next
  case MainReturn
  thus ?case by −(erule PCFG.cases,auto dest:Proc-CFG-Call-nodes-eq')
next
  case (ProcReturn p ins outs c l p' es' rets' l' ins' outs' c' ps)
```

```
  from ‹prog,procs ⊢ (p',Exit) −et'→ (p, Label l')›
    ‹(p, ins, outs, c) ∈ set procs› ‹(p', ins', outs', c') ∈ set procs›
    ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'›
    ‹containsCall procs prog ps p› ‹well-formed procs›
  show ?case
  proof(induct (p',Exit) et' (p,Label l') rule:PCFG.induct)
    case (ProcReturn insx outsx cx lx es'x rets'x ins'x outs'x c'x psx)
    from ‹(p', ins'x, outs'x, c'x) ∈ set procs›
      ‹(p', ins', outs', c') ∈ set procs› ‹well-formed procs›
    have [simp]:outs'x = outs' by fastforce
    from ‹(p, insx, outsx, cx) ∈ set procs› ‹(p, ins, outs, c) ∈ set procs›
      ‹well-formed procs›
    have [simp]:cx = c by auto
    from ‹cx ⊢ Label lx −CEdge (p', es'x, rets'x)→p Label l'›
      ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'›
    have [simp]:rets'x = rets' by(fastforce dest:Proc-CFG-Call-nodes-eq')
    show ?case by simp
  qed auto
next
  case MainCallReturn thus ?case by(auto elim:PCFG.cases dest:Proc-CFG-edge-det)
next
  case ProcCallReturn thus ?case by(auto elim:PCFG.cases dest:Proc-CFG-edge-det)
qed


lemma Proc-CFG-deterministic:
  ⟦prog,procs ⊢ n1 −et1→ n1'; prog,procs ⊢ n2 −et2→ n2'; n1 = n2; n1' ≠ n2';
    intra-kind et1; intra-kind et2; well-formed procs⟧
  ⟹ ∃ Q Q'. et1 = (Q)√ ∧ et2 = (Q')√ ∧
          (∀ s. (Q s ⟶ ¬ Q' s) ∧ (Q' s ⟶ ¬ Q s))
proof(induct arbitrary:n2 n2' rule:PCFG.induct)
  case (Main n et n')
  from ‹prog,procs ⊢ n2 −et2→ n2'› ‹(Main,n) = n2›
    ‹intra-kind et2› ‹well-formed procs›
  obtain m m' where (Main,m) = n2 and (Main,m') = n2'
    and disj:prog ⊢ m −IEdge et2→p m' ∨
    (∃ p es rets. prog ⊢ m −CEdge (p,es,rets)→p m' ∧ et2 = (λs. False)√)
    by(induct rule:PCFG.induct)(fastforce simp:intra-kind-def)+
  from disj show ?case
  proof
    assume prog ⊢ m −IEdge et2→p m'
    with ‹(Main,m) = n2› ‹(Main,m') = n2'›
      ‹prog ⊢ n −IEdge et→p n'› ‹(Main,n) = n2› ‹(Main,n') ≠ n2'›
    show ?thesis by(auto dest:WCFG-deterministic)
  next
    assume ∃ p es rets. prog ⊢ m −CEdge (p, es, rets)→p m' ∧ et2 = (λs. False)√
    with ‹(Main,m) = n2› ‹(Main,m') = n2'›
      ‹prog ⊢ n −IEdge et→p n'› ‹(Main,n) = n2› ‹(Main,n') ≠ n2'›
    have False by(fastforce dest:Proc-CFG-Call-Intra-edge-not-same-source)
```

thus *?thesis* **by** *simp*
qed
next
case (*Proc p ins outs c n et n'*)
from ‹*prog*,*procs* ⊢ n_2 −et_2→ n_2'› ‹(*p*,*n*) = n_2› ‹*intra-kind* et_2›
‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*› ‹*well-formed procs*›
obtain *m m'* **where** (*p*,*m*) = n_2 **and** (*p*,*m'*) = n_2'
and *disj*:*c* ⊢ *m* −*IEdge* et_2→$_p$ *m'* ∨
(∃ *p' es' rets'*. *c* ⊢ *m* −*CEdge* (*p'*,*es'*,*rets'*)→$_p$ *m'* ∧ et_2 = (λ*s*. *False*)$_\surd$)
by(*induct rule*:*PCFG.induct*)(*fastforce simp*:*intra-kind-def*)+
from *disj* **show** *?case*
proof
assume *c* ⊢ *m* −*IEdge* et_2→$_p$ *m'*
with ‹(*p*,*m*) = n_2› ‹(*p*,*m'*) = n_2'›
‹*c* ⊢ *n* −*IEdge et*→$_p$ *n'*› ‹(*p*,*n*) = n_2› ‹(*p*,*n'*) ≠ n_2'›
show *?thesis* **by**(*auto dest*:*WCFG-deterministic*)
next
assume ∃ *p' es' rets'*. *c* ⊢ *m* −*CEdge* (*p'*, *es'*, *rets'*)→$_p$ *m'* ∧ et_2 = (λ*s*. *False*)$_\surd$
with ‹(*p*,*m*) = n_2› ‹(*p*,*m'*) = n_2'›
‹*c* ⊢ *n* −*IEdge et*→$_p$ *n'*› ‹(*p*,*n*) = n_2› ‹(*p*,*n'*) ≠ n_2'›
have *False* **by**(*fastforce dest*:*Proc-CFG-Call-Intra-edge-not-same-source*)
thus *?thesis* **by** *simp*
qed
next
case (*MainCallReturn n p es rets n' n*$_2$ n_2')
from ‹*prog*,*procs* ⊢ n_2 −et_2→ n_2'› ‹(*Main*,*n*) = n_2›
‹*intra-kind* et_2› ‹*well-formed procs*›
obtain *m m'* **where** (*Main*,*m*) = n_2 **and** (*Main*,*m'*) = n_2'
and *disj*:*prog* ⊢ *m* −*IEdge* et_2→$_p$ *m'* ∨
(∃ *p es rets*. *prog* ⊢ *m* −*CEdge* (*p*,*es*,*rets*)→$_p$ *m'* ∧ et_2 = (λ*s*. *False*)$_\surd$)
by(*induct rule*:*PCFG.induct*)(*fastforce simp*:*intra-kind-def*)+
from *disj* **show** *?case*
proof
assume *prog* ⊢ *m* −*IEdge* et_2→$_p$ *m'*
with ‹(*Main*,*m*) = n_2› ‹(*Main*,*m'*) = n_2'› ‹*prog* ⊢ *n* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n'*›
‹(*Main*, *n*) = n_2› ‹(*Main*, *n'*) ≠ n_2'›
have *False* **by**(*fastforce dest*:*Proc-CFG-Call-Intra-edge-not-same-source*)
thus *?thesis* **by** *simp*
next
assume ∃ *p es rets*. *prog* ⊢ *m* −*CEdge* (*p*,*es*,*rets*)→$_p$ *m'* ∧ et_2 = (λ*s*. *False*)$_\surd$
with ‹(*Main*,*m*) = n_2› ‹(*Main*,*m'*) = n_2'› ‹*prog* ⊢ *n* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n'*›
‹(*Main*, *n*) = n_2› ‹(*Main*, *n'*) ≠ n_2'›
show *?thesis* **by**(*fastforce dest*:*Proc-CFG-Call-nodes-eq*)
qed
next
case (*ProcCallReturn p ins outs c n p' es rets n' ps n*$_2$ n_2')
from ‹*prog*,*procs* ⊢ n_2 −et_2→ n_2'› ‹(*p*,*n*) = n_2› ‹*intra-kind* et_2›

‹(p,ins,outs,c) ∈ set procs› ‹well-formed procs›
obtain m m′ **where** (p,m) = n_2 **and** (p,m′) = n_2′
and disj:c ⊢ m −IEdge et_2→$_p$ m′ ∨
(∃ p′ es′ rets′. c ⊢ m −CEdge (p′,es′,rets′)→$_p$ m′ ∧ et_2 = $(\lambda s.\ False)_\surd$)
by(induct rule:PCFG.induct)(fastforce simp:intra-kind-def)+
from disj **show** ?case
proof
assume c ⊢ m −IEdge et_2→$_p$ m′
with ‹(p,m) = n_2› ‹(p,m′) = n_2′›
‹c ⊢ n −CEdge (p′, es, rets)→$_p$ n′› ‹(p,n) = n_2› ‹(p,n′) ≠ n_2′›
have False **by**(fastforce dest:Proc-CFG-Call-Intra-edge-not-same-source)
thus ?thesis **by** simp
next
assume ∃ p′ es′ rets′. c ⊢ m −CEdge (p′, es′, rets′)→$_p$ m′ ∧ et_2 = $(\lambda s.\ False)_\surd$
with ‹(p,m) = n_2› ‹(p,m′) = n_2′›
‹c ⊢ n −CEdge (p′, es, rets)→$_p$ n′› ‹(p,n) = n_2› ‹(p,n′) ≠ n_2′›
show ?thesis **by**(fastforce dest:Proc-CFG-Call-nodes-eq)
qed
qed(auto simp:intra-kind-def)

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

lemma wf-well-formed [intro]:wf prog procs ⟹ well-formed procs
by(simp add:wf-def)

lemma wf-distinct-rets [intro]:
⟦wf prog procs; containsCall procs prog ps p; (p,ins,outs,c) ∈ set procs;
c′ ⊢ n −CEdge (p,es,rets)→$_p$ n′⟧ ⟹ distinct rets
by(fastforce simp:wf-def)

lemma
assumes wf prog procs **and** containsCall procs prog ps p
and (p,ins,outs,c) ∈ set procs **and** c′ ⊢ n −CEdge (p,es,rets)→$_p$ n′
shows wf-length-retsI [intro]:length rets = length outs
and wf-length-esI [intro]:length es = length ins
proof −
from ‹wf prog procs› **have** well-formed procs **by** fastforce
from assms

obtain *ins′ outs′ c′* **where** $(p,ins',outs',c') \in set\ procs$
and *lengths*:*length rets* = *length outs′ length es* = *length ins′*
by(*simp add*:*wf-def*) *blast*
from ‹$(p,ins,outs,c) \in set\ procs$› ‹$(p,ins',outs',c') \in set\ procs$›
‹*well-formed procs*›
have *ins′* = *ins outs′* = *outs c′* = *c* **by** *auto*
with *lengths* **show** *length rets* = *length outs length es* = *length ins*
by *simp-all*
qed

2.4.3 Type of well-formed programs

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

typedef *wf-prog* = *wf-prog*
unfolding *wf-prog-def*
apply (*rule-tac x*=(*Skip*,[]) **in** *exI*)
apply (*simp add*:*wf-def well-formed-def*)
done

lemma *wf-wf-prog*:*Rep-wf-prog wfp* = (*prog*,*procs*) ⟹ *wf prog procs*
using *Rep-wf-prog*[*of wfp*] **by**(*simp add*:*wf-prog-def*)

lemma *wfp-Seq1*: **assumes** *Rep-wf-prog wfp* = (c_1;; c_2, *procs*)
obtains *wfp′* **where** *Rep-wf-prog wfp′* = (c_1, *procs*)
using ‹*Rep-wf-prog wfp* = (c_1;; c_2, *procs*)›
apply(*cases wfp*) **apply**(*auto simp*:*Abs-wf-prog-inverse wf-prog-def wf-def*)
apply(*erule-tac x*=*Abs-wf-prog* (c_1, *procs*) **in** *meta-allE*)
by(*auto elim*:*meta-mp simp*:*Abs-wf-prog-inverse wf-prog-def wf-def*)

lemma *wfp-Seq2*: **assumes** *Rep-wf-prog wfp* = (c_1;; c_2, *procs*)
obtains *wfp′* **where** *Rep-wf-prog wfp′* = (c_2, *procs*)
using ‹*Rep-wf-prog wfp* = (c_1;; c_2, *procs*)›
apply(*cases wfp*) **apply**(*auto simp*:*Abs-wf-prog-inverse wf-prog-def wf-def*)
apply(*erule-tac x*=*Abs-wf-prog* (c_2, *procs*) **in** *meta-allE*)
by(*auto elim*:*meta-mp simp*:*Abs-wf-prog-inverse wf-prog-def wf-def*)

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*)
using ‹*Rep-wf-prog wfp* = (*if* (*b*) c_1 *else* c_2, *procs*)›
apply(*cases wfp*) **apply**(*auto simp*:*Abs-wf-prog-inverse wf-prog-def wf-def*)
apply(*erule-tac x*=*Abs-wf-prog* (c_1, *procs*) **in** *meta-allE*)
by(*auto elim*:*meta-mp simp*:*Abs-wf-prog-inverse wf-prog-def wf-def*)

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*)
using ‹*Rep-wf-prog wfp* = (*if* (*b*) c_1 *else* c_2, *procs*)›
apply(*cases wfp*) **apply**(*auto simp*:*Abs-wf-prog-inverse wf-prog-def wf-def*)

apply(*erule-tac x=Abs-wf-prog* (c_2, *procs*) **in** *meta-allE*)
by(*auto elim:meta-mp simp:Abs-wf-prog-inverse wf-prog-def wf-def*)

lemma *wfp-WhileBody*: **assumes** *Rep-wf-prog wfp* = (*while* (*b*) *c'*, *procs*)
obtains *wfp'* **where** *Rep-wf-prog wfp'* = (*c'*, *procs*)
using ‹*Rep-wf-prog wfp* = (*while* (*b*) *c'*, *procs*)›
apply(*cases wfp*) **apply**(*auto simp:Abs-wf-prog-inverse wf-prog-def wf-def*)
apply(*erule-tac x=Abs-wf-prog* (*c'*, *procs*) **in** *meta-allE*)
by(*auto elim:meta-mp simp:Abs-wf-prog-inverse wf-prog-def wf-def*)

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*)
using *assms*
apply(*cases wfp*) **apply**(*auto simp:Abs-wf-prog-inverse wf-prog-def wf-def*)
apply(*erule-tac x=Abs-wf-prog* (*c*, *procs*) **in** *meta-allE*)
apply(*erule meta-mp*) **apply**(*rule Abs-wf-prog-inverse*)
by(*auto dest:containsCall-indirection simp:wf-prog-def wf-def*)

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* = {}
  by(*cases kind a*,*auto simp*:*get-return-edges-def*)

lemma *call-has-return-edge*:
  assumes *valid-edge wfp a* **and** *kind a* = *Q*:*r*↪$_p$*fs*
  obtains *a'* **where** *valid-edge wfp a'* **and** ∃ *Q' f'*. *kind a'* = *Q'*↩$_p$*f'*
  and *targetnode a'* = *r*
proof(*atomize-elim*)
  from ‹*valid-edge wfp a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
  obtain *prog procs* **where** *Rep-wf-prog wfp* = (*prog*,*procs*)
    and *prog*,*procs* ⊢ *sourcenode a* −*Q*:*r*↪$_p$*fs*→ *targetnode a*
    by(*fastforce simp*:*valid-edge-def*)
  from ‹*prog*,*procs* ⊢ *sourcenode a* −*Q*:*r*↪$_p$*fs*→ *targetnode a*›
  show ∃ *a'*. *valid-edge wfp a'* ∧ (∃ *Q' f'*. *kind a'* = *Q'*↩$_p$*f'*) ∧ *targetnode a'* = *r*
  proof(*induct sourcenode a Q*:*r*↪$_p$*fs targetnode a rule*:*PCFG.induct*)
    case (*MainCall l es rets n' ins outs c*)
    from ‹*prog* ⊢ *Label l* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n'*› **obtain** *l'*
      where [*simp*]:*n'* = *Label l'*
      by(*fastforce dest*:*Proc-CFG-Call-Labels*)
    from *MainCall*
    have *prog*,*procs* ⊢ (*p*,*Exit*) −(λ*cf*. *snd cf* = (*Main*,*Label l'*))↩$_p$
      (λ*cf cf'*. *cf'*(*rets* [:=] *map cf outs*))→ (*Main*,*Label l'*)
      by(*fastforce intro*:*MainReturn*)
    with ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› ‹(*Main*, *n'*) = *r*› **show** *?thesis*
      by(*fastforce simp*:*valid-edge-def*)
  next
    case (*ProcCall px ins outs c l es' rets' l' ins' outs' c' ps*)
    from *ProcCall* **have** *prog*,*procs* ⊢ (*p*,*Exit*) −(λ*cf*. *snd cf* = (*px*,*Label l'*))↩$_p$
      (λ*cf cf'*. *cf'*(*rets'* [:=] *map cf outs'*))→ (*px*,*Label l'*)
      by(*fastforce intro*:*ProcReturn*)
    with ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› ‹(*px*, *Label l'*) = *r*› **show** *?thesis*
      by(*fastforce simp*:*valid-edge-def*)
  qed *auto*
qed

lemma *get-return-edges-call-nonempty*:
  ⟦*valid-edge wfp a*; *kind a* = *Q*:*r*↪$_p$*fs*⟧ ⟹ *get-return-edges wfp a* ≠ {}
by −(*erule call-has-return-edge*,(*fastforce simp*:*get-return-edges-def*)+)

lemma *only-return-edges-in-get-return-edges*:
  ⟦*valid-edge wfp a*; *kind a* = *Q*:*r*↪$_p$*fs*; *a'* ∈ *get-return-edges wfp a*⟧

$\Longrightarrow \exists\, Q'\, f'.\ kind\ a' = Q'\!\hookleftarrow_p\! f'$
by(*cases kind a*,*auto simp*:*get-return-edges-def*)

abbreviation *lift-procs* :: *wf-prog* $\Rightarrow$ (*pname* $\times$ *vname list* $\times$ *vname list*) *list*
where *lift-procs wfp* $\equiv$ *let* (*prog*,*procs*) = *Rep-wf-prog wfp in*
map ($\lambda x.$ (*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 −
from *Rep-wf-prog*[*of wfp*]
obtain *prog procs* **where** [*simp*]:*Rep-wf-prog wfp* = (*prog*,*procs*)
by(*fastforce simp*:*wf-prog-def*)
hence *wf*:*well-formed procs* **by**(*fastforce intro*:*wf-wf-prog*)
show *CFG sourcenode targetnode kind* (*valid-edge wfp*) (*Main*, *Entry*)
get-proc (*get-return-edges wfp*) (*lift-procs wfp*) *Main*
proof
fix *a* **assume** *valid-edge wfp a* **and** *targetnode a* = (*Main*, *Entry*)
from *this wf* **show** *False* **by**(*auto elim*:*PCFG.cases simp*:*valid-edge-def*)
next
show *get-proc* (*Main*, *Entry*) = *Main* **by** *simp*
next
fix *a Q r p fs*
assume *valid-edge wfp a* **and** *kind a* = *Q*:$r\!\hookrightarrow_p\! fs$
and *sourcenode a* = (*Main*, *Entry*)
thus *False* **by**(*auto elim*:*PCFG.cases simp*:*valid-edge-def*)
next
fix *a a'*
assume *valid-edge wfp a* **and** *valid-edge wfp a'*
and *sourcenode a* = *sourcenode a'* **and** *targetnode a* = *targetnode a'*
with *wf* **show** *a* = *a'*
by(*cases a*,*cases a'*,*auto dest*:*Proc-CFG-edge-det simp*:*valid-edge-def*)
next
fix *a Q r f*
assume *valid-edge wfp a* **and** *kind a* = *Q*:$r\!\hookrightarrow_{Main}\! f$
from *this wf* **show** *False* **by**(*auto elim*:*PCFG.cases simp*:*valid-edge-def*)
next
fix *a Q' f'*
assume *valid-edge wfp a* **and** *kind a* = *Q'*$\!\hookleftarrow_{Main}\! f'$
from *this wf* **show** *False* **by**(*auto elim*:*PCFG.cases simp*:*valid-edge-def*)
next
fix *a Q r p fs*
assume *valid-edge wfp a* **and** *kind a* = *Q*:$r\!\hookrightarrow_p\! fs$
thus $\exists$ *ins outs*. (*p*, *ins*, *outs*) $\in$ *set* (*lift-procs wfp*)

apply(*auto simp*:*valid-edge-def*) **apply**(*erule PCFG.cases*) **apply** *auto*
apply(*fastforce dest*:*Proc-CFG-IEdge-intra-kind simp*:*intra-kind-def*)
apply(*fastforce dest*:*Proc-CFG-IEdge-intra-kind simp*:*intra-kind-def*)
apply(*rule-tac x*=*ins* **in** *exI*) **apply**(*rule-tac x*=*outs* **in** *exI*)
apply(*rule-tac x*=(*p*,*ins*,*outs*,*c*) **in** *image-eqI*) **apply** *auto*
apply(*rule-tac x*=*ins'* **in** *exI*) **apply**(*rule-tac x*=*outs'* **in** *exI*)
apply(*rule-tac x*=(*p*,*ins'*,*outs'*,*c'*) **in** *image-eqI*) **by**(*auto simp*:*set-conv-nth*)
next
fix *a* **assume** *valid-edge wfp a* **and** *intra-kind* (*kind a*)
thus *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*)
by(*auto elim*:*PCFG.cases simp*:*valid-edge-def intra-kind-def*)
next
fix *a Q r p fs*
assume *valid-edge wfp a* **and** *kind a* = *Q*:*r*↪$_p$*fs*
thus *get-proc* (*targetnode a*) = *p* **by**(*auto elim*:*PCFG.cases simp*:*valid-edge-def*)

next
fix *a Q' p f'*
assume *valid-edge wfp a* **and** *kind a* = *Q'*↩$_p$*f'*
thus *get-proc* (*sourcenode a*) = *p* **by**(*auto elim*:*PCFG.cases simp*:*valid-edge-def*)

next
fix *a Q r p fs*
assume *valid-edge wfp a* **and** *kind a* = *Q*:*r*↪$_p$*fs*
hence *prog*,*procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*
by(*simp add*:*valid-edge-def*)
from *this* ‹*kind a* = *Q*:*r*↪$_p$*fs*›
show ∀ *a'*. *valid-edge wfp a'* ∧ *targetnode a'* = *targetnode a* ⟶
(∃ *Qx rx fsx*. *kind a'* = *Qx*:*rx*↪$_p$*fsx*)
proof(*induct sourcenode a kind a targetnode a rule*:*PCFG.induct*)
case (*MainCall l p' es rets n' ins outs c*)
from ‹λ*s*. *True*:(*Main*, *n'*)↪$_{p'}$*map interpret es* = *kind a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
have [*simp*]:*p'* = *p* **by** *simp*
{ fix *a'* **assume** *valid-edge wfp a'* **and** *targetnode a'* = (*p'*, *Entry*)
hence ∃ *Qx rx fsx*. *kind a'* = *Qx*:*rx*↪$_p$*fsx*
by(*auto elim*:*PCFG.cases simp*:*valid-edge-def*) **}**
with ‹(*p'*,*Entry*) = *targetnode a*› **show** *?case* **by** *simp*
next
case (*ProcCall px ins outs c l p' es rets l' ins' outs' c' ps*)
from ‹λ*s*. *True*:(*px*, *Label l'*)↪$_{p'}$*map interpret es* = *kind a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
have [*simp*]:*p'* = *p* **by** *simp*
{ fix *a'* **assume** *valid-edge wfp a'* **and** *targetnode a'* = (*p'*, *Entry*)
hence ∃ *Qx rx fsx*. *kind a'* = *Qx*:*rx*↪$_p$*fsx*
by(*auto elim*:*PCFG.cases simp*:*valid-edge-def*) **}**
with ‹(*p'*, *Entry*) = *targetnode a*› **show** *?case* **by** *simp*
qed *auto*
next
fix *a Q' p f'*

assume *valid-edge wfp a* **and** *kind a* = $Q'\hookleftarrow_p f'$
hence *prog,procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*
by(*simp add*:*valid-edge-def*)
from *this* ‹*kind a* = $Q'\hookleftarrow_p f'$›
show ∀ *a'*. *valid-edge wfp a'* ∧ *sourcenode a'* = *sourcenode a* ⟶
(∃ *Qx fx*. *kind a'* = $Qx\hookleftarrow_p fx$)
proof(*induct sourcenode a kind a targetnode a rule*:*PCFG.induct*)
case (*MainReturn l p' es rets l' ins outs c*)
from ‹λ*cf*. *snd cf* = (*Main*, *Label l'*)$\hookleftarrow_{p'}$λ*cf cf'*. *cf'*(*rets* [:=] *map cf outs*) =
kind a› ‹*kind a* = $Q'\hookleftarrow_p f'$› **have** [*simp*]:*p'* = *p* **by** *simp*
{ **fix** *a'* **assume** *valid-edge wfp a'* **and** *sourcenode a'* = (*p'*, *Exit*)
hence ∃ *Qx fx*. *kind a'* = $Qx\hookleftarrow_p fx$
by(*auto elim*:*PCFG.cases simp*:*valid-edge-def*) }
with ‹(*p'*, *Exit*) = *sourcenode a*› **show** *?case* **by** *simp*
next
case (*ProcReturn px ins outs c l p' es rets l' ins' outs' c' ps*)
from ‹λ*cf*. *snd cf* = (*px*, *Label l'*)$\hookleftarrow_{p'}$λ*cf cf'*. *cf'*(*rets* [:=] *map cf outs'*) =
kind a› ‹*kind a* = $Q'\hookleftarrow_p f'$› **have** [*simp*]:*p'* = *p* **by** *simp*
{ **fix** *a'* **assume** *valid-edge wfp a'* **and** *sourcenode a'* = (*p'*, *Exit*)
hence ∃ *Qx fx*. *kind a'* = $Qx\hookleftarrow_p fx$
by(*auto elim*:*PCFG.cases simp*:*valid-edge-def*) }
with ‹(*p'*, *Exit*) = *sourcenode a*› **show** *?case* **by** *simp*
qed *auto*
next
fix *a Q r p fs*
assume *valid-edge wfp a* **and** *kind a* = Q:$r\hookrightarrow_p fs$
thus *get-return-edges wfp a* ≠ {} **by**(*rule get-return-edges-call-nonempty*)
next
fix *a a'*
assume *valid-edge wfp a* **and** *a'* ∈ *get-return-edges wfp a*
thus *valid-edge wfp a'*
by(*cases kind a*,*auto simp*:*get-return-edges-def*)
next
fix *a a'*
assume *valid-edge wfp a* **and** *a'* ∈ *get-return-edges wfp a*
thus ∃ *Q r p fs*. *kind a* = Q:$r\hookrightarrow_p fs$
by(*cases kind a*)(*auto simp*:*get-return-edges-def*)
next
fix *a Q r p fs a'*
assume *valid-edge wfp a* **and** *kind a* = Q:$r\hookrightarrow_p fs$
and *a'* ∈ *get-return-edges wfp a*
thus ∃ *Q' f'*. *kind a'* = $Q'\hookleftarrow_p f'$ **by**(*rule only-return-edges-in-get-return-edges*)
next
fix *a Q' p f'*
assume *valid-edge wfp a* **and** *kind a* = $Q'\hookleftarrow_p f'$
hence *prog,procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*
by(*simp add*:*valid-edge-def*)
from *this* ‹*kind a* = $Q'\hookleftarrow_p f'$›
show ∃!*a'*. *valid-edge wfp a'* ∧ (∃ *Q r fs*. *kind a'* = Q:$r\hookrightarrow_p fs$) ∧

```
    a ∈ get-return-edges wfp a'
  proof(induct sourcenode a kind a targetnode a rule:PCFG.induct)
    case (MainReturn l px es rets l' ins outs c)
    from ‹λcf. snd cf = (Main, Label l')↩px λcf cf'. cf'(rets [:=] map cf outs) =
      kind a› ‹kind a = Q'↩pf'› have [simp]:px = p by simp
    from ‹prog ⊢ Label l −CEdge (px, es, rets)→p Label l'› have l' = Suc l
      by(fastforce dest:Proc-CFG-Call-Labels)
    from ‹prog ⊢ Label l −CEdge (px, es, rets)→p Label l'›
    have containsCall procs prog [] px by(rule Proc-CFG-Call-containsCall)
    with ‹prog ⊢ Label l −CEdge (px, es, rets)→p Label l'›
      ‹(px, ins, outs, c) ∈ set procs›
    have prog,procs ⊢ (p,Exit) −(λcf. snd cf = (Main,Label l'))↩p
      (λcf cf'. cf'(rets [:=] map cf outs))→ (Main,Label l')
      by(fastforce intro:PCFG.MainReturn)
    with ‹(px, Exit) = sourcenode a› ‹(Main, Label l') = targetnode a›
      ‹λcf. snd cf = (Main, Label l')↩px λcf cf'. cf'(rets [:=] map cf outs) =
      kind a›
    have edge:prog,procs ⊢ sourcenode a −kind a→ targetnode a by simp
    from ‹prog ⊢ Label l −CEdge (px, es, rets)→p Label l'›
      ‹(px, ins, outs, c) ∈ set procs›
    have edge':prog,procs ⊢ (Main,Label l)
      −(λs. True):(Main,Label l')↪pmap (λe cf. interpret e cf) es→ (p,Entry)
      by(fastforce intro:MainCall)
    show ?case
    proof(rule ex-ex1I)
      from edge edge' ‹(Main, Label l') = targetnode a›
        ‹l' = Suc l› ‹kind a = Q'↩pf'›
      show ∃ a'. valid-edge wfp a' ∧
        (∃ Q r fs. kind a' = Q:r↪pfs) ∧ a ∈ get-return-edges wfp a'
        by(fastforce simp:valid-edge-def get-return-edges-def)
    next
      fix a' a''
      assume valid-edge wfp a' ∧
        (∃ Q r fs. kind a' = Q:r↪pfs) ∧ a ∈ get-return-edges wfp a'
        and valid-edge wfp a'' ∧
        (∃ Q r fs. kind a'' = Q:r↪pfs) ∧ a ∈ get-return-edges wfp a''
      then obtain Q r fs Q' r' fs' where valid-edge wfp a'
        and kind a' = Q:r↪pfs and a ∈ get-return-edges wfp a'
        and valid-edge wfp a'' and kind a'' = Q':r'↪pfs'
        and a ∈ get-return-edges wfp a'' by blast
     from ‹valid-edge wfp a'› ‹kind a' = Q:r↪pfs›[THEN sym] edge wf ‹l' = Suc
l›
        ‹a ∈ get-return-edges wfp a'› ‹(Main, Label l') = targetnode a›
      have nodes:sourcenode a' = (Main,Label l) ∧ targetnode a' = (p,Entry)
        apply(auto simp:valid-edge-def get-return-edges-def)
        by(erule PCFG.cases,auto dest:Proc-CFG-Call-Labels)+
      from ‹valid-edge wfp a''› ‹kind a'' = Q':r'↪pfs'›[THEN sym] ‹l' = Suc l›
          ‹a ∈ get-return-edges wfp a''› ‹(Main, Label l') = targetnode a› wf edge'
      have nodes':sourcenode a'' = (Main,Label l) ∧ targetnode a'' = (p,Entry)
```

```
      apply(auto simp:valid-edge-def get-return-edges-def)
      by(erule PCFG.cases,auto dest:Proc-CFG-Call-Labels)+
    with nodes ‹valid-edge wfp a'› ‹valid-edge wfp a''› wf
    have kind a' = kind a''
      by(fastforce dest:Proc-CFG-edge-det simp:valid-edge-def)
    with nodes nodes' show a' = a'' by(cases a',cases a'',auto)
  qed
 next
  case (ProcReturn p' ins outs c l px esx retsx l' ins' outs' c' ps)
  from ‹λcf. snd cf = (p', Label l')↩px λcf cf'. cf'(retsx [:=] map cf outs') =
    kind a› ‹kind a = Q'↩pf'› have [simp]:px = p by simp
  from ‹c ⊢ Label l −CEdge (px, esx, retsx)→p Label l'› have l' = Suc l
    by(fastforce dest:Proc-CFG-Call-Labels)
  from ‹(p',ins,outs,c) ∈ set procs›
    ‹c ⊢ Label l −CEdge (px, esx, retsx)→p Label l'›
    ‹(px, ins', outs', c') ∈ set procs› ‹containsCall procs prog ps p'›
  have prog,procs ⊢ (p,Exit) −(λcf. snd cf = (p',Label l'))↩p
    (λcf cf'. cf'(retsx [:=] map cf outs'))→ (p',Label l')
    by(fastforce intro:PCFG.ProcReturn)
  with ‹(px, Exit) = sourcenode a› ‹(p', Label l') = targetnode a›
    ‹λcf. snd cf = (p', Label l')↩px λcf cf'. cf'(retsx [:=] map cf outs') =
     kind a› have edge:prog,procs ⊢ sourcenode a −kind a→ targetnode a by
simp
  from ‹(p',ins,outs,c) ∈ set procs›
    ‹c ⊢ Label l −CEdge (px, esx, retsx)→p Label l'›
    ‹(px, ins', outs', c') ∈ set procs› ‹containsCall procs prog ps p'›
  have edge':prog,procs ⊢ (p',Label l)
    −(λs. True):(p',Label l')↪pmap (λe cf. interpret e cf) esx→ (p,Entry)
    by(fastforce intro:ProcCall)
  show ?case
  proof(rule ex-ex1I)
    from edge edge' ‹(p', Label l') = targetnode a› ‹l' = Suc l›
      ‹(p', ins, outs, c) ∈ set procs› ‹kind a = Q'↩pf'›
    show ∃ a'. valid-edge wfp a' ∧
      (∃ Q r fs. kind a' = Q:r↪pfs) ∧ a ∈ get-return-edges wfp a'
      by(fastforce simp:valid-edge-def get-return-edges-def)
  next
    fix a' a''
    assume valid-edge wfp a' ∧
      (∃ Q r fs. kind a' = Q:r↪pfs) ∧ a ∈ get-return-edges wfp a'
      and valid-edge wfp a'' ∧
      (∃ Q r fs. kind a'' = Q:r↪pfs) ∧ a ∈ get-return-edges wfp a''
    then obtain Q r fs Q' r' fs' where valid-edge wfp a'
      and kind a' = Q:r↪pfs and a ∈ get-return-edges wfp a'
      and valid-edge wfp a'' and kind a'' = Q':r'↪pfs'
      and a ∈ get-return-edges wfp a'' by blast
    from ‹valid-edge wfp a'› ‹kind a' = Q:r↪pfs›[THEN sym]
      ‹a ∈ get-return-edges wfp a'› edge ‹(p', Label l') = targetnode a› wf
      ‹(p', ins, outs, c) ∈ set procs› ‹l' = Suc l›
```

```
    have nodes:sourcenode a' = (p',Label l) ∧ targetnode a' = (p,Entry)
      apply(auto simp:valid-edge-def get-return-edges-def)
      by(erule PCFG.cases,auto dest:Proc-CFG-Call-Labels)+
    from ‹valid-edge wfp a''› ‹kind a'' = Q':r'↪pfs'›[THEN sym]
      ‹a ∈ get-return-edges wfp a''› edge ‹(p', Label l') = targetnode a› wf
      ‹(p', ins, outs, c) ∈ set procs› ‹l' = Suc l›
    have nodes':sourcenode a'' = (p',Label l) ∧ targetnode a'' = (p,Entry)
      apply(auto simp:valid-edge-def get-return-edges-def)
      by(erule PCFG.cases,auto dest:Proc-CFG-Call-Labels)+
    with nodes ‹valid-edge wfp a'› ‹valid-edge wfp a''› wf
    have kind a' = kind a''
      by(fastforce dest:Proc-CFG-edge-det simp:valid-edge-def)
    with nodes nodes' show a' = a'' by(cases a',cases a'',auto)
  qed
 qed auto
next
 fix a a'
 assume valid-edge wfp a and a' ∈ get-return-edges wfp a
 then obtain Q r p fs l'
  where kind a = Q:r↪pfs and valid-edge wfp a'
  by(cases kind a)(fastforce simp:valid-edge-def get-return-edges-def)+
 from ‹valid-edge wfp a› ‹kind a = Q:r↪pfs› ‹a' ∈ get-return-edges wfp a›
 obtain Q' f' where kind a' = Q'↩pf'
  by(fastforce dest!:only-return-edges-in-get-return-edges)
 with ‹valid-edge wfp a'› have sourcenode a' = (p,Exit)
  by(auto elim:PCFG.cases simp:valid-edge-def)
 from ‹valid-edge wfp a› ‹kind a = Q:r↪pfs›
 have prog,procs ⊢ sourcenode a −Q:r↪pfs→ targetnode a
  by(simp add:valid-edge-def)
 thus ∃ a''. valid-edge wfp a'' ∧ sourcenode a'' = targetnode a ∧
  targetnode a'' = sourcenode a' ∧ kind a'' = (λcf. False)√
 proof(induct sourcenode a Q:r↪pfs targetnode a rule:PCFG.induct)
  case (MainCall l es rets n' ins outs c)
  have c ⊢ Entry −IEdge (λs. False)√→p Exit by(rule Proc-CFG-Entry-Exit)
  moreover
  from ‹prog ⊢ Label l −CEdge (p, es, rets)→p n'›
  have containsCall procs prog [] p by(rule Proc-CFG-Call-containsCall)
  ultimately have prog,procs ⊢ (p,Entry) −(λs. False)√→ (p,Exit)
   using ‹(p, ins, outs, c) ∈ set procs› by(fastforce intro:Proc)
  with ‹sourcenode a' = (p,Exit)› ‹(p, Entry) = targetnode a›[THEN sym]
  show ?case by(fastforce simp:valid-edge-def)
 next
  case (ProcCall px ins outs c l es' rets' l' ins' outs' c' ps)
  have c' ⊢ Entry −IEdge (λs. False)√→p Exit by(rule Proc-CFG-Entry-Exit)
  moreover
  from ‹c ⊢ Label l −CEdge (p, es', rets')→p Label l'›
  have containsCall procs c [] p by(rule Proc-CFG-Call-containsCall)
  with ‹containsCall procs prog ps px› ‹(px,ins,outs,c) ∈ set procs›
  have containsCall procs prog (ps@[px]) p
```

```
    by(rule containsCall-in-proc)
   ultimately have prog,procs ⊢ (p,Entry) −(λs. False)√→ (p,Exit)
    using ‹(p, ins', outs', c') ∈ set procs› by(fastforce intro:Proc)
   with ‹sourcenode a' = (p,Exit)› ‹(p, Entry) = targetnode a›[THEN sym]
   show ?case by(fastforce simp:valid-edge-def)
  qed auto
next
  fix a a'
  assume valid-edge wfp a and a' ∈ get-return-edges wfp a
  then obtain Q r p fs l'
    where kind a = Q:r↪pfs and valid-edge wfp a'
    by(cases kind a)(fastforce simp:valid-edge-def get-return-edges-def)+
  from ‹valid-edge wfp a› ‹kind a = Q:r↪pfs› ‹a' ∈ get-return-edges wfp a›
  obtain Q' f' where kind a' = Q'↩pf' and targetnode a' = r
    by(auto simp:get-return-edges-def)
  from ‹valid-edge wfp a› ‹kind a = Q:r↪pfs›
  have prog,procs ⊢ sourcenode a −Q:r↪pfs→ targetnode a
    by(simp add:valid-edge-def)
  thus ∃ a''. valid-edge wfp a'' ∧ sourcenode a'' = sourcenode a ∧
    targetnode a'' = targetnode a' ∧ kind a'' = (λcf. False)√
  proof(induct sourcenode a Q:r↪pfs targetnode a rule:PCFG.induct)
    case (MainCall l es rets n' ins outs c)
    from ‹prog ⊢ Label l −CEdge (p, es, rets)→p n'›
    have prog,procs ⊢ (Main,Label l) −(λs. False)√→ (Main,n')
      by(rule MainCallReturn)
    with ‹(Main, Label l) = sourcenode a›[THEN sym] ‹targetnode a' = r›
      ‹(Main, n') = r›[THEN sym]
    show ?case by(auto simp:valid-edge-def)
  next
    case (ProcCall px ins outs c l es' rets' l' ins' outs' c' ps)
    from ‹(px,ins,outs,c) ∈ set procs›  ‹containsCall procs prog ps px›
      ‹c ⊢ Label l −CEdge (p, es', rets')→p Label l'›
    have prog,procs ⊢ (px,Label l) −(λs. False)√→ (px,Label l')
      by(fastforce intro:ProcCallReturn)
    with ‹(px, Label l) = sourcenode a›[THEN sym] ‹targetnode a' = r›
      ‹(px, Label l') = r›[THEN sym]
    show ?case by(auto simp:valid-edge-def)
  qed auto
next
  fix a Q r p fs
  assume valid-edge wfp a and kind a = Q:r↪pfs
  hence prog,procs ⊢ sourcenode a −kind a→ targetnode a
    by(simp add:valid-edge-def)
  from this ‹kind a = Q:r↪pfs›
  show ∃!a'. valid-edge wfp a' ∧
    sourcenode a' = sourcenode a ∧ intra-kind (kind a')
  proof(induct sourcenode a kind a targetnode a rule:PCFG.induct)
    case (MainCall l p' es rets n' ins outs c)
    show ?thesis
```

```
  proof(rule ex-ex1I)
    from ‹prog ⊢ Label l −CEdge (p', es, rets)→_p n'›
    have prog,procs ⊢ (Main,Label l) −(λs. False)_√→ (Main,n')
      by(rule MainCallReturn)
    with ‹(Main, Label l) = sourcenode a›[THEN sym]
    show ∃ a'. valid-edge wfp a' ∧
      sourcenode a' = sourcenode a ∧ intra-kind (kind a')
      by(fastforce simp:valid-edge-def intra-kind-def)
  next
    fix a' a''
    assume valid-edge wfp a' ∧ sourcenode a' = sourcenode a ∧
      intra-kind (kind a') and valid-edge wfp a'' ∧
      sourcenode a'' = sourcenode a ∧ intra-kind (kind a'')
    hence valid-edge wfp a' and sourcenode a' = sourcenode a
      and intra-kind (kind a') and valid-edge wfp a''
      and sourcenode a'' = sourcenode a and intra-kind (kind a'') by simp-all
    from ‹valid-edge wfp a'› ‹sourcenode a' = sourcenode a›
      ‹intra-kind (kind a')› ‹prog ⊢ Label l −CEdge (p', es, rets)→_p n'›
      ‹(Main, Label l) = sourcenode a› wf
    have targetnode a' = (Main,Label (Suc l))
     by(auto elim!:PCFG.cases dest:Proc-CFG-Call-Intra-edge-not-same-source

        Proc-CFG-Call-Labels simp:intra-kind-def valid-edge-def)
    from ‹valid-edge wfp a''› ‹sourcenode a'' = sourcenode a›
      ‹intra-kind (kind a'')› ‹prog ⊢ Label l −CEdge (p', es, rets)→_p n'›
      ‹(Main, Label l) = sourcenode a› wf
    have targetnode a'' = (Main,Label (Suc l))
     by(auto elim!:PCFG.cases dest:Proc-CFG-Call-Intra-edge-not-same-source

        Proc-CFG-Call-Labels simp:intra-kind-def valid-edge-def)
    with ‹valid-edge wfp a'› ‹sourcenode a' = sourcenode a›
      ‹valid-edge wfp a''› ‹sourcenode a'' = sourcenode a›
      ‹targetnode a' = (Main,Label (Suc l))› wf
    show a' = a'' by(cases a',cases a'')
    (auto dest:Proc-CFG-edge-det simp:valid-edge-def)
  qed
next
  case (ProcCall px ins outs c l p' es' rets' l' ins' outs' c' ps)
  show ?thesis
  proof(rule ex-ex1I)
    from ‹(px, ins, outs, c) ∈ set procs› ‹containsCall procs prog ps px›
      ‹c ⊢ Label l −CEdge (p', es', rets')→_p Label l'›
    have prog,procs ⊢ (px,Label l) −(λs. False)_√→ (px,Label l')
      by −(rule ProcCallReturn)
    with ‹(px, Label l) = sourcenode a›[THEN sym]
    show ∃ a'. valid-edge wfp a' ∧ sourcenode a' = sourcenode a ∧
            intra-kind (kind a')
      by(fastforce simp:valid-edge-def intra-kind-def)
  next
```

```
    fix a' a''
    assume valid-edge wfp a' ∧ sourcenode a' = sourcenode a ∧
      intra-kind (kind a') and valid-edge wfp a'' ∧
      sourcenode a'' = sourcenode a ∧ intra-kind (kind a'')
    hence valid-edge wfp a' and sourcenode a' = sourcenode a
      and intra-kind (kind a') and valid-edge wfp a''
      and sourcenode a'' = sourcenode a and intra-kind (kind a'') by simp-all
    from ‹valid-edge wfp a'› ‹sourcenode a' = sourcenode a›
      ‹intra-kind (kind a')› ‹(px, ins, outs, c) ∈ set procs›
      ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'›
      ‹(p', ins', outs', c') ∈ set procs› wf
      ‹containsCall procs prog ps px› ‹(px, Label l) = sourcenode a›
    have targetnode a' = (px,Label (Suc l))
      apply(auto simp:valid-edge-def) apply(erule PCFG.cases)
      by(auto dest:Proc-CFG-Call-Intra-edge-not-same-source
        Proc-CFG-Call-nodes-eq Proc-CFG-Call-Labels simp:intra-kind-def)
    from ‹valid-edge wfp a''› ‹sourcenode a'' = sourcenode a›
      ‹intra-kind (kind a'')› ‹(px, ins, outs, c) ∈ set procs›
      ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'›
      ‹(p', ins', outs', c') ∈ set procs› wf
      ‹containsCall procs prog ps px› ‹(px, Label l) = sourcenode a›
    have targetnode a'' = (px,Label (Suc l))
      apply(auto simp:valid-edge-def) apply(erule PCFG.cases)
      by(auto dest:Proc-CFG-Call-Intra-edge-not-same-source
        Proc-CFG-Call-nodes-eq Proc-CFG-Call-Labels simp:intra-kind-def)
    with ‹valid-edge wfp a'› ‹sourcenode a' = sourcenode a›
      ‹valid-edge wfp a''› ‹sourcenode a'' = sourcenode a›
      ‹targetnode a' = (px,Label (Suc l))› wf
    show a' = a'' by(cases a',cases a'')
    (auto dest:Proc-CFG-edge-det simp:valid-edge-def)
  qed
 qed auto
next
 fix a Q' p f'
 assume valid-edge wfp a and kind a = Q'↩pf'
 hence prog,procs ⊢ sourcenode a −kind a→ targetnode a
  by(simp add:valid-edge-def)
 from this ‹kind a = Q'↩pf'›
 show ∃!a'. valid-edge wfp a' ∧
  targetnode a' = targetnode a ∧ intra-kind (kind a')
 proof(induct sourcenode a kind a targetnode a rule:PCFG.induct)
  case (MainReturn l p' es rets l' ins outs c)
  show ?thesis
  proof(rule ex-ex1I)
   from ‹prog ⊢ Label l −CEdge (p', es, rets)→p Label l'›
   have prog,procs ⊢ (Main,Label l) −(λs. False)√→
    (Main,Label l') by(rule MainCallReturn)
   with ‹(Main, Label l') = targetnode a›[THEN sym]
   show ∃ a'. valid-edge wfp a' ∧
```

```
    targetnode a' = targetnode a ∧ intra-kind (kind a')
    by(fastforce simp:valid-edge-def intra-kind-def)
  next
   fix a' a''
   assume valid-edge wfp a' ∧ targetnode a' = targetnode a ∧
    intra-kind (kind a') and valid-edge wfp a'' ∧
    targetnode a'' = targetnode a ∧ intra-kind (kind a'')
   hence valid-edge wfp a' and targetnode a' = targetnode a
    and intra-kind (kind a') and valid-edge wfp a''
    and targetnode a'' = targetnode a and intra-kind (kind a'') by simp-all
   from ‹valid-edge wfp a'› ‹targetnode a' = targetnode a›
    ‹intra-kind (kind a')› ‹prog ⊢ Label l -CEdge (p', es, rets)→p Label l'›
    ‹(Main, Label l') = targetnode a› wf
   have sourcenode a' = (Main,Label l)
   apply(auto elim!:PCFG.cases dest:Proc-CFG-Call-Intra-edge-not-same-target

                simp:valid-edge-def intra-kind-def)
    by(fastforce dest:Proc-CFG-Call-nodes-eq' Proc-CFG-Call-Labels)
   from ‹valid-edge wfp a''› ‹targetnode a'' = targetnode a›
    ‹intra-kind (kind a'')› ‹prog ⊢ Label l -CEdge (p', es, rets)→p Label l'›
    ‹(Main, Label l') = targetnode a› wf
   have sourcenode a'' = (Main,Label l)
   apply(auto elim!:PCFG.cases dest:Proc-CFG-Call-Intra-edge-not-same-target

                simp:valid-edge-def intra-kind-def)
    by(fastforce dest:Proc-CFG-Call-nodes-eq' Proc-CFG-Call-Labels)
   with ‹valid-edge wfp a'› ‹targetnode a' = targetnode a›
    ‹valid-edge wfp a''› ‹targetnode a'' = targetnode a›
    ‹sourcenode a' = (Main,Label l)› wf
   show a' = a'' by(cases a',cases a'')
   (auto dest:Proc-CFG-edge-det simp:valid-edge-def)
  qed
next
  case (ProcReturn px ins outs c l p' es' rets' l' ins' outs' c' ps)
  show ?thesis
  proof(rule ex-ex1I)
   from ‹(px, ins, outs, c) ∈ set procs› ‹containsCall procs prog ps px›
    ‹c ⊢ Label l -CEdge (p', es', rets')→p Label l'›
   have prog,procs ⊢ (px,Label l) -(λs. False)√→ (px,Label l')
    by -(rule ProcCallReturn)
   with ‹(px, Label l') = targetnode a›[THEN sym]
   show ∃ a'. valid-edge wfp a' ∧
    targetnode a' = targetnode a ∧ intra-kind (kind a')
    by(fastforce simp:valid-edge-def intra-kind-def)
  next
   fix a' a''
   assume valid-edge wfp a' ∧ targetnode a' = targetnode a ∧
    intra-kind (kind a') and valid-edge wfp a'' ∧
    targetnode a'' = targetnode a ∧ intra-kind (kind a'')
```

```
      hence valid-edge wfp a' and targetnode a' = targetnode a
        and intra-kind (kind a') and valid-edge wfp a''
        and targetnode a'' = targetnode a and intra-kind (kind a'') by simp-all
      from ‹valid-edge wfp a'› ‹targetnode a' = targetnode a›
        ‹intra-kind (kind a')› ‹(px, ins, outs, c) ∈ set procs›
        ‹(p', ins', outs', c') ∈ set procs› wf
        ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'›
        ‹containsCall procs prog ps px› ‹(px, Label l') = targetnode a›
      have sourcenode a' = (px,Label l)
        apply(auto simp:valid-edge-def) apply(erule PCFG.cases)
        by(auto dest:Proc-CFG-Call-Intra-edge-not-same-target
          Proc-CFG-Call-nodes-eq' simp:intra-kind-def)
      from ‹valid-edge wfp a''› ‹targetnode a'' = targetnode a›
        ‹intra-kind (kind a'')› ‹(px, ins, outs, c) ∈ set procs›
        ‹(p', ins', outs', c') ∈ set procs› wf
        ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'›
        ‹containsCall procs prog ps px› ‹(px, Label l') = targetnode a›
      have sourcenode a'' = (px,Label l)
        apply(auto simp:valid-edge-def) apply(erule PCFG.cases)
        by(auto dest:Proc-CFG-Call-Intra-edge-not-same-target
          Proc-CFG-Call-nodes-eq' simp:intra-kind-def)
      with ‹valid-edge wfp a'› ‹targetnode a' = targetnode a›
        ‹valid-edge wfp a''› ‹targetnode a'' = targetnode a›
        ‹sourcenode a' = (px,Label l)› wf
      show a' = a'' by(cases a',cases a'')
      (auto dest:Proc-CFG-edge-det simp:valid-edge-def)
    qed
  qed auto
 next
  fix a a' Q1 r1 p fs1 Q2 r2 fs2
  assume valid-edge wfp a and valid-edge wfp a'
    and kind a = Q1:r1↪pfs1 and kind a' = Q2:r2↪pfs2
 thus targetnode a = targetnode a' by(auto elim!:PCFG.cases simp:valid-edge-def)
 next
  from wf show distinct-fst (lift-procs wfp)
    by(fastforce simp:well-formed-def distinct-fst-def o-def)
 next
  fix p ins outs assume (p, ins, outs) ∈ set (lift-procs wfp)
  from ‹(p, ins, outs) ∈ set (lift-procs wfp)› wf
  show distinct ins by(fastforce simp:well-formed-def wf-proc-def)
 next
  fix p ins outs assume (p, ins, outs) ∈ set (lift-procs wfp)
  from ‹(p, ins, outs) ∈ set (lift-procs wfp)› wf
  show distinct outs by(fastforce simp:well-formed-def wf-proc-def)
 qed
qed
```

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 −
from *Rep-wf-prog*[*of wfp*]
obtain *prog procs* **where** [*simp*]:*Rep-wf-prog wfp* = (*prog*,*procs*)
by(*fastforce simp*:*wf-prog-def*)
hence *wf*:*well-formed procs* **by**(*fastforce intro*:*wf-wf-prog*)
show *CFGExit sourcenode targetnode kind* (*valid-edge wfp*) (*Main*, *Entry*)
get-proc (*get-return-edges wfp*) (*lift-procs wfp*) *Main* (*Main*, *Exit*)
proof
fix *a* **assume** *valid-edge wfp a* **and** *sourcenode a* = (*Main*, *Exit*)
with *wf* **show** *False* **by**(*auto elim*:*PCFG.cases simp*:*valid-edge-def*)
next
show *get-proc* (*Main*, *Exit*) = *Main* **by** *simp*
next
fix *a Q p f*
assume *valid-edge wfp a* **and** *kind a* = $Q\hookleftarrow_{pf}$
and *targetnode a* = (*Main*, *Exit*)
thus *False* **by**(*auto elim*:*PCFG.cases simp*:*valid-edge-def*)
next
have *prog*,*procs* ⊢ (*Main*,*Entry*) −(*λs. False*)$_{\surd}\rightarrow$ (*Main*,*Exit*)
by(*fastforce intro*:*Main Proc-CFG-Entry-Exit*)
thus ∃ *a. valid-edge wfp a* ∧
sourcenode a = (*Main*, *Entry*) ∧
targetnode a = (*Main*, *Exit*) ∧ *kind a* = (*λs. False*)$_{\surd}$
by(*fastforce simp*:*valid-edge-def*)
qed
qed

end

2.6 Labels

theory *Labels* **imports** *Com* **begin**

Labels describe a mapping from the inner node label to the matching command

inductive *labels* :: *cmd* ⇒ *nat* ⇒ *cmd* ⇒ *bool*
where

Labels-Base:
labels c 0 c

| *Labels-LAss*:

labels (*V*:=*e*) *1 Skip*

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

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

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

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

| *Labels-WhileBody*:
labels *c*′ *l c* ⟹ *labels* (*while*(*b*) *c*′) (*l* + *2*) (*c*;;*while*(*b*) *c*′)

| *Labels-WhileExit*:
labels (*while*(*b*) *c*′) *1 Skip*

| *Labels-Call*:
labels (*Call p es rets*) *1 Skip*

lemma *label-less-num-inner-nodes*:
labels c l c′ ⟹ *l* < #:*c*
proof(*induct c arbitrary*:*l c*′)
case *Skip*
from ‹*labels Skip l c*′› **show** *?case* **by**(*fastforce elim*:*labels.cases*)
next
case (*LAss V e*)
from ‹*labels* (*V*:=*e*) *l c*′› **show** *?case* **by**(*fastforce elim*:*labels.cases*)
next
case (*Seq* c_1 c_2)
note *IH1* = ‹⋀*l c*′. *labels* c_1 *l c*′ ⟹ *l* < #:c_1›
note *IH2* = ‹⋀*l c*′. *labels* c_2 *l c*′ ⟹ *l* < #:c_2›
from ‹*labels* (c_1;;c_2) *l c*′› *IH1 IH2* **show** *?case*
by *simp*(*erule labels.cases*,*auto*,*force*)
next
case (*Cond b* c_1 c_2)
note *IH1* = ‹⋀*l c*′. *labels* c_1 *l c*′ ⟹ *l* < #:c_1›
note *IH2* = ‹⋀*l c*′. *labels* c_2 *l c*′ ⟹ *l* < #:c_2›
from ‹*labels* (*if* (*b*) c_1 *else* c_2) *l c*′› *IH1 IH2* **show** *?case*
by *simp*(*erule labels.cases*,*auto*,*force*)
next
case (*While b c*)
note *IH* = ‹⋀*l c*′. *labels c l c*′ ⟹ *l* < #:*c*›
from ‹*labels* (*while* (*b*) *c*) *l c*′› *IH* **show** *?case*
by *simp*(*erule labels.cases*,*fastforce*+)

next
case (*Call p es rets*)
thus *?case* **by** *simp*(*erule labels.cases*,*fastforce*+)
qed

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

lemma *less-num-inner-nodes-label*:
assumes $l < \#{:}c$ **obtains** c' **where** *labels c l c'*
proof(*atomize-elim*)
from ‹$l < \#{:}c$› **show** $\exists\, c'.$ *labels c l c'*
proof(*induct c arbitrary*:*l*)
case *Skip*
from ‹$l < \#{:}$*Skip*› **have** $l = 0$ **by** *simp*
thus *?case* **by**(*fastforce intro*:*Labels-Base*)
next
case (*LAss V e*)
from ‹$l < \#{:}(V{:=}e)$› **have** $l = 0 \vee l = 1$ **by** *auto*
thus *?case* **by**(*auto intro*:*Labels-Base Labels-LAss*)
next
case (*Seq* c_1 c_2)
note *IH1* = ‹$\bigwedge l.\ l < \#{:}c_1 \Longrightarrow \exists\, c'.$ *labels* c_1 *l c'*›
note *IH2* = ‹$\bigwedge l.\ l < \#{:}c_2 \Longrightarrow \exists\, c'.$ *labels* c_2 *l c'*›
show *?case*
proof(*cases* $l < \#{:}c_1$)
case *True*
from *IH1*[*OF this*] **obtain** c' **where** *labels* c_1 *l c'* **by** *auto*
hence *labels* (c_1;;c_2) *l* (c';;c_2) **by**(*fastforce intro*:*Labels-Seq1*)
thus *?thesis* **by** *auto*
next
case *False*
hence $\#{:}c_1 \leq l$ **by** *simp*
then obtain l' **where** $l = l' + \#{:}c_1$ **and** $l' = l - \#{:}c_1$ **by** *simp*
from ‹$l = l' + \#{:}c_1$› ‹$l < \#{:}c_1$;;c_2› **have** $l' < \#{:}c_2$ **by** *simp*
from *IH2*[*OF this*] **obtain** c' **where** *labels* c_2 l' c' **by** *auto*
with ‹$l = l' + \#{:}c_1$› **have** *labels* (c_1;;c_2) *l c'*
by(*fastforce intro*:*Labels-Seq2*)
thus *?thesis* **by** *auto*
qed
next
case (*Cond b* c_1 c_2)
note *IH1* = ‹$\bigwedge l.\ l < \#{:}c_1 \Longrightarrow \exists\, c'.$ *labels* c_1 *l c'*›
note *IH2* = ‹$\bigwedge l.\ l < \#{:}c_2 \Longrightarrow \exists\, c'.$ *labels* c_2 *l c'*›
show *?case*
proof(*cases* $l = 0$)
case *True*
thus *?thesis* **by**(*fastforce intro*:*Labels-Base*)
next

```
    case False
    hence 0 < l by simp
    then obtain l' where l = l' + 1 and l' = l - 1 by simp
    thus ?thesis
    proof(cases l' < #:c₁)
      case True
      from IH1[OF this] obtain c' where labels c₁ l' c' by auto
      with ‹l = l' + 1› have labels (if (b) c₁ else c₂) l c'
        by(fastforce dest:Labels-CondTrue)
      thus ?thesis by auto
    next
      case False
      hence #:c₁ ≤ l' by simp
      then obtain l'' where l' = l'' + #:c₁ and l'' = l' - #:c₁ by simp
      from ‹l' = l'' + #:c₁› ‹l = l' + 1› ‹l < #:if (b) c₁ else c₂›
      have l'' < #:c₂ by simp
      from IH2[OF this] obtain c' where labels c₂ l'' c' by auto
      with ‹l' = l'' + #:c₁› ‹l = l' + 1› have labels (if (b) c₁ else c₂) l c'
        by(fastforce dest:Labels-CondFalse)
      thus ?thesis by auto
    qed
  qed
next
  case (While b c')
  note IH = ‹⋀l. l < #:c' ⟹ ∃c''. labels c' l c''›
  show ?case
  proof(cases l < 1)
    case True
    hence l = 0 by simp
    thus ?thesis by(fastforce intro:Labels-Base)
  next
    case False
    show ?thesis
    proof(cases l < 2)
      case True
      with ‹¬ l < 1› have l = 1 by simp
      thus ?thesis by(fastforce intro:Labels-WhileExit)
    next
      case False
      with ‹¬ l < 1› have 2 ≤ l by simp
      then obtain l' where l = l' + 2 and l' = l - 2
        by(simp del:add-2-eq-Suc')
      from ‹l = l' + 2› ‹l < #:while (b) c'› have l' < #:c' by simp
      from IH[OF this] obtain c'' where labels c' l' c'' by auto
      with ‹l = l' + 2› have labels (while (b) c') l (c'';;while (b) c')
        by(fastforce dest:Labels-WhileBody)
      thus ?thesis by auto
    qed
  qed
```

```
  next
    case (Call p es rets)
    show ?case
    proof(cases l < 1)
      case True
      hence l = 0 by simp
      thus ?thesis by(fastforce intro:Labels-Base)
    next
      case False
      with ‹l < #:Call p es rets› have l = 1 by simp
      thus ?thesis by(fastforce intro:Labels-Call)
    qed
  qed
qed


lemma labels-det:
  labels c l c' ⟹ (⋀c''. labels c l c'' ⟹ c' = c'')
proof(induct rule:labels.induct)
  case (Labels-Base c c'')
  from ‹labels c 0 c''› obtain l where labels c l c'' and l = 0 by auto
  thus ?case by(induct rule:labels.induct,auto)
next
  case (Labels-Seq1 c₁ l c c₂)
  note IH = ‹⋀c''. labels c₁ l c'' ⟹ c = c''›
  from ‹labels c₁ l c› have l < #:c₁ by(fastforce intro:label-less-num-inner-nodes)
  with ‹labels (c₁;;c₂) l c''› obtain cx where c'' = cx;;c₂ ∧ labels c₁ l cx
    by(fastforce elim:labels.cases intro:Labels-Base)
  hence [simp]:c'' = cx;;c₂ and labels c₁ l cx by simp-all
  from IH[OF ‹labels c₁ l cx›] show ?case by simp
next
  case (Labels-Seq2 c₂ l c c₁)
  note IH = ‹⋀c''. labels c₂ l c'' ⟹ c = c''›
  from ‹labels (c₁;;c₂) (l + #:c₁) c''› ‹labels c₂ l c› have labels c₂ l c''
    by(auto elim:labels.cases dest:label-less-num-inner-nodes)
  from IH[OF this] show ?case .
next
  case (Labels-CondTrue c₁ l c b c₂)
  note IH = ‹⋀c''. labels c₁ l c'' ⟹  c = c''›
  from ‹labels (if (b) c₁ else c₂) (l + 1) c''› ‹labels c₁ l c› have labels c₁ l c''
    by(fastforce elim:labels.cases dest:label-less-num-inner-nodes)
  from IH[OF this] show ?case .
next
  case (Labels-CondFalse c₂ l c b c₁)
  note IH = ‹⋀c''. labels c₂ l c'' ⟹  c = c''›
  from ‹labels (if (b) c₁ else c₂) (l + #:c₁ + 1) c''› ‹labels c₂ l c›
  have labels c₂ l c''
    by(fastforce elim:labels.cases dest:label-less-num-inner-nodes)
  from IH[OF this] show ?case .
```

next
case (*Labels-WhileBody c′ l c b*)
note *IH* = ‹⋀*c″*. *labels c′ l c″* ⟹ *c* = *c″*›
from ‹*labels* (*while* (*b*) *c′*) (*l* + *2*) *c″*› ‹*labels c′ l c*›
obtain *cx* **where** *c″* = *cx*;;*while* (*b*) *c′* ∧ *labels c′ l cx*
by −(*erule labels.cases*,*auto*)
hence [*simp*]:*c″* = *cx*;;*while* (*b*) *c′* **and** *labels c′ l cx* **by** *simp-all*
from *IH*[*OF* ‹*labels c′ l cx*›] **show** *?case* **by** *simp*
qed (*fastforce elim*:*labels.cases*)+

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

lemma *labels-THE*:
labels c l c′ ⟹ (*THE c′*. *labels c l c′*) = *c′*
by(*fastforce intro*:*the-equality dest*:*labels-det*)

lemma *labels-label*:*labels c l c′* ⟹ *label c l* = *c′*
by(*fastforce intro*:*labels-THE simp*:*label-def*)

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″*⟧
⟹ *fst-cmd c* = *Skip*
proof(*induct arbitrary*:*n rule*:*labels.induct*)
case (*Labels-Seq1 c*$_1$ *l c c*$_2$)
note *IH* = ‹⋀*n*. *c*$_1$ ⊢ *n* −*CEdge* (*p*, *es*, *rets*)→$_p$ *Label l* ⟹ *fst-cmd c* = *Skip*›
from ‹*c*$_1$;; *c*$_2$ ⊢ *n* −*CEdge* (*p*, *es*, *rets*)→$_p$ *Label l*› ‹*labels c*$_1$ *l c*›
have *c*$_1$ ⊢ *n* −*CEdge* (*p*, *es*, *rets*)→$_p$ *Label l*
apply − **apply**(*erule Proc-CFG.cases*,*auto dest*:*Proc-CFG-Call-Labels*)

```
    by(case-tac n')(auto dest:label-less-num-inner-nodes)
  from IH[OF this] show ?case by simp
next
  case (Labels-Seq2 c₂ l c c₁)
  note IH = ⟨⋀n. c₂ ⊢ n −CEdge (p, es, rets)→p Label l ⟹ fst-cmd c = Skip⟩
  from ⟨c₁;; c₂ ⊢ n −CEdge (p, es, rets)→p Label (l + #:c₁)⟩ ⟨labels c₂ l c⟩
  obtain nx where c₂ ⊢ nx −CEdge (p, es, rets)→p Label l
    apply − apply(erule Proc-CFG.cases)
    apply(auto dest:Proc-CFG-targetlabel-less-num-nodes Proc-CFG-Call-Labels)
    by(case-tac n') auto
  from IH[OF this] show ?case by simp
next
  case (Labels-CondTrue c₁ l c b c₂)
  note IH = ⟨⋀n. c₁ ⊢ n −CEdge (p, es, rets)→p Label l ⟹ fst-cmd c = Skip⟩
  from ⟨if (b) c₁ else c₂ ⊢ n −CEdge (p, es, rets)→p Label (l + 1)⟩ ⟨labels c₁ l c⟩
  obtain nx where c₁ ⊢ nx −CEdge (p, es, rets)→p Label l
    apply − apply(erule Proc-CFG.cases,auto)
     apply(case-tac n') apply auto
    by(case-tac n')(auto dest:label-less-num-inner-nodes)
  from IH[OF this] show ?case by simp
next
  case (Labels-CondFalse c₂ l c b c₁)
  note IH = ⟨⋀n. c₂ ⊢ n −CEdge (p, es, rets)→p Label l ⟹ fst-cmd c = Skip⟩
  from ⟨if (b) c₁ else c₂ ⊢ n −CEdge (p, es, rets)→p Label (l + #:c₁ + 1)⟩
    ⟨labels c₂ l c⟩
  obtain nx where c₂ ⊢ nx −CEdge (p, es, rets)→p Label l
    apply − apply(erule Proc-CFG.cases,auto)
     apply(case-tac n') apply(auto dest:Proc-CFG-targetlabel-less-num-nodes)
    by(case-tac n') auto
  from IH[OF this] show ?case by simp
next
  case (Labels-WhileBody c' l c b)
  note IH = ⟨⋀n. c' ⊢ n −CEdge (p, es, rets)→p Label l ⟹ fst-cmd c = Skip⟩
  from ⟨while (b) c' ⊢ n −CEdge (p, es, rets)→p Label (l + 2)⟩ ⟨labels c' l c⟩
  obtain nx where c' ⊢ nx −CEdge (p, es, rets)→p Label l
    apply − apply(erule Proc-CFG.cases,auto)
    by(case-tac n') auto
  from IH[OF this] show ?case by simp
next
  case (Labels-Call px esx retsx)
  from ⟨Call px esx retsx ⊢ n −CEdge (p, es, rets)→p Label 1⟩
  show ?case by(fastforce elim:Proc-CFG.cases)
qed(auto dest:Proc-CFG-Call-Labels)


lemma Proc-CFG-Call-source-fst-cmd-Call:
  ⟦labels prog l c; prog ⊢ Label l −CEdge (p,es,rets)→p n'⟧
  ⟹ ∃p es rets. fst-cmd c = Call p es rets
proof(induct arbitrary:n' rule:labels.induct)
```

```
  case (Labels-Base c n')
  from ‹c ⊢ Label 0 −CEdge (p, es, rets)→p n'› show ?case
    by(induct c Label 0 CEdge (p, es, rets) n' rule:Proc-CFG.induct) auto
next
  case (Labels-LAss V e n')
  from ‹V:=e ⊢ Label 1 −CEdge (p, es, rets)→p n'› show ?case
    by(fastforce elim:Proc-CFG.cases)
next
  case (Labels-Seq1 c1 l c c2)
  note IH = ‹⋀n'. c1 ⊢ Label l −CEdge (p, es, rets)→p n'
    ⟹ ∃p es rets. fst-cmd c = Call p es rets›
  from ‹c1;; c2 ⊢ Label l −CEdge (p, es, rets)→p n'› ‹labels c1 l c›
  have c1 ⊢ Label l −CEdge (p, es, rets)→p n'
    apply − apply(erule Proc-CFG.cases,auto dest:Proc-CFG-Call-Labels)
    by(case-tac n)(auto dest:label-less-num-inner-nodes)
  from IH[OF this] show ?case by simp
next
  case (Labels-Seq2 c2 l c c1)
  note IH = ‹⋀n'. c2 ⊢ Label l −CEdge (p, es, rets)→p n'
    ⟹ ∃p es rets. fst-cmd c = Call p es rets›
  from ‹c1;; c2 ⊢ Label (l + #:c1) −CEdge (p, es, rets)→p n'› ‹labels c2 l c›
  obtain nx where c2 ⊢ Label l −CEdge (p, es, rets)→p nx
    apply − apply(erule Proc-CFG.cases)
    apply(auto dest:Proc-CFG-sourcelabel-less-num-nodes Proc-CFG-Call-Labels)
    by(case-tac n) auto
  from IH[OF this] show ?case by simp
next
  case (Labels-CondTrue c1 l c b c2)
  note IH = ‹⋀n'. c1 ⊢ Label l −CEdge (p, es, rets)→p n'
    ⟹ ∃p es rets. fst-cmd c = Call p es rets›
  from ‹if (b) c1 else c2 ⊢ Label (l + 1) −CEdge (p, es, rets)→p n'› ‹labels c1 l
c›
  obtain nx where c1 ⊢ Label l −CEdge (p, es, rets)→p nx
    apply − apply(erule Proc-CFG.cases,auto)
     apply(case-tac n) apply auto
    by(case-tac n)(auto dest:label-less-num-inner-nodes)
  from IH[OF this] show ?case by simp
next
  case (Labels-CondFalse c2 l c b c1)
  note IH = ‹⋀n'. c2 ⊢ Label l −CEdge (p, es, rets)→p n'
    ⟹ ∃p es rets. fst-cmd c = Call p es rets›
  from ‹if (b) c1 else c2 ⊢ Label (l + #:c1 + 1)−CEdge (p, es, rets)→p n'›
    ‹labels c2 l c›
  obtain nx where c2 ⊢ Label l −CEdge (p, es, rets)→p nx
    apply − apply(erule Proc-CFG.cases,auto)
     apply(case-tac n) apply(auto dest:Proc-CFG-sourcelabel-less-num-nodes)
    by(case-tac n) auto
  from IH[OF this] show ?case by simp
next
```

case (*Labels-WhileBody c′ l c b*)
note *IH* = ‹⋀*n′*. *c′* ⊢ *Label l* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n′*
⟹ ∃ *p es rets*. *fst-cmd c* = *Call p es rets*›
from ‹*while* (*b*) *c′* ⊢ *Label* (*l* + *2*) −*CEdge* (*p*, *es*, *rets*)→$_p$ *n′*› ‹*labels c′ l c*›
obtain *nx* **where** *c′* ⊢ *Label l* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx*
apply − **apply**(*erule Proc-CFG.cases*,*auto dest*:*Proc-CFG-Call-Labels*)
by(*case-tac n*) *auto*
from *IH*[*OF this*] **show** *?case* **by** *simp*
next
case (*Labels-WhileExit b c′ n′*)
have *while* (*b*) *c′* ⊢ *Label 1* −*IEdge* ↑*id*→$_p$ *Exit* **by**(*rule Proc-CFG-WhileFalseSkip*)
with ‹*while* (*b*) *c′* ⊢ *Label 1* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n′*›
have *False* **by**(*rule Proc-CFG-Call-Intra-edge-not-same-source*)
thus *?case* **by** *simp*
next
case (*Labels-Call px esx retsx*)
from ‹*Call px esx retsx* ⊢ *Label 1* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n′*›
show *?case* **by**(*fastforce elim*:*Proc-CFG.cases*)
qed

2.7.2 Definition of *Def* and *Use* sets

ParamDefs

lemma *PCFG-CallEdge-THE-rets*:
prog ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *n′*
⟹ (*THE rets′*. ∃ *p′ es′ n*. *prog* ⊢ *n* −*CEdge*(*p′*,*es′*,*rets′*)→$_p$ *n′*) = *rets*
by(*fastforce intro*:*the-equality dest*:*Proc-CFG-Call-nodes-eq′*)

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*
by(*fastforce intro*:*the-equality*)

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*
by(*fastforce dest*:*PCFG-CallEdge-THE-rets simp*:*ParamDefs-def ParamDefs-proc-def*)

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 −
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
with ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*› **have** *p* ≠ *Main* **by** *fastforce*
moreover
from ‹*well-formed procs*› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
have (*THE c'*. ∃ *ins' outs'*. (*p*,*ins'*,*outs'*,*c'*) ∈ *set procs*) = *c*
by(*rule in-procs-THE-in-procs-cmd*)
ultimately show *?thesis* **using** *assms*
by(*fastforce dest*:*PCFG-CallEdge-THE-rets simp*:*ParamDefs-def ParamDefs-proc-def*)
qed

lemma *ParamDefs-Main-IEdge-Nil*:
⟦*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* ⊢ *n* −*IEdge et*→$_p$ *n'*⟧
⟹ *ParamDefs wfp* (*Main*,*n'*) = []
by(*fastforce dest*:*Proc-CFG-Call-Intra-edge-not-same-target*
simp:*ParamDefs-def ParamDefs-proc-def*)

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 −
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
with ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*› **have** *p* ≠ *Main* **by** *fastforce*
moreover
from ‹*well-formed procs*› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
have (*THE c'*. ∃ *ins' outs'*. (*p*,*ins'*,*outs'*,*c'*) ∈ *set procs*) = *c*
by(*rule in-procs-THE-in-procs-cmd*)
ultimately show *?thesis* **using** *assms*
by(*fastforce dest*:*Proc-CFG-Call-Intra-edge-not-same-target*
simp:*ParamDefs-def ParamDefs-proc-def*)
qed

lemma *ParamDefs-Main-CEdge-Nil*:
⟦*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* ⊢ *n'* −*CEdge*(*p'*,*es*,*rets*)→$_p$ *n''*⟧
⟹ *ParamDefs wfp* (*Main*,*n'*) = []
by(*fastforce dest*:*Proc-CFG-Call-targetnode-no-Call-sourcenode*

simp:*ParamDefs-def ParamDefs-proc-def*)

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 −
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
with ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*› **have** *p* ≠ *Main* **by** *fastforce*
moreover
from ‹*well-formed procs*› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
have (*THE c'*. ∃ *ins' outs'*. (*p*,*ins'*,*outs'*,*c'*) ∈ *set procs*) = *c*
by(*rule in-procs-THE-in-procs-cmd*)
ultimately show *?thesis* **using** *assms*
by(*fastforce dest*:*Proc-CFG-Call-targetnode-no-Call-sourcenode*
simp:*ParamDefs-def ParamDefs-proc-def*)
qed

lemma assumes *valid-edge wfp a* **and** *kind a* = *Q'*↪$_p$*f'*
and (*p*, *ins*, *outs*) ∈ *set* (*lift-procs wfp*)
shows *ParamDefs-length*:*length* (*ParamDefs wfp* (*targetnode a*)) = *length outs*
(**is** *?length*)
and *Return-update*:*f' cf cf'* = *cf'*(*ParamDefs wfp* (*targetnode a*) [:=] *map cf outs*)
(**is** *?update*)
proof −
from *Rep-wf-prog*[*of wfp*]
obtain *prog procs* **where** [*simp*]:*Rep-wf-prog wfp* = (*prog*,*procs*)
by(*fastforce simp*:*wf-prog-def*)
hence *wf prog procs* **by**(*rule wf-wf-prog*)
hence *wf*:*well-formed procs* **by** *fastforce*
from *assms* **have** *prog*,*procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*
by(*simp add*:*valid-edge-def*)
from *this* ‹*kind a* = *Q'*↪$_p$*f'*› *wf* **have** *?length* ∧ *?update*
proof(*induct sourcenode a kind a targetnode a rule*:*PCFG.induct*)
case (*MainReturn l p' es rets l' insx outsx cx*)
from ‹λ*cf*. *snd cf* = (*Main*, *Label l'*)↪$_{p'}$λ*cf cf'*. *cf'*(*rets* [:=] *map cf outsx*) =
kind a› ‹*kind a* = *Q'*↪$_p$*f'*› **have** *p'* = *p*
and *f'*:*f'* = (λ*cf cf'*. *cf'*(*rets* [:=] *map cf outsx*)) **by** *simp-all*
with ‹*well-formed procs*› ‹(*p'*, *insx*, *outsx*, *cx*) ∈ *set procs*›
‹(*p*, *ins*, *outs*) ∈ *set* (*lift-procs wfp*)›
have [*simp*]:*outsx* = *outs* **by** *fastforce*
from ‹*prog* ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *Label l'*›
have *containsCall procs prog* [] *p'* **by**(*rule Proc-CFG-Call-containsCall*)
with ‹*wf prog procs*› ‹(*p'*, *insx*, *outsx*, *cx*) ∈ *set procs*›
‹*prog* ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *Label l'*›
have *length rets* = *length outs* **by** *fastforce*

```
    from ‹prog ⊢ Label l −CEdge (p', es, rets)→p Label l'›
    have ParamDefs wfp (Main,Label l') = rets
      by(fastforce intro:ParamDefs-Main-Return-target)
    with ‹(Main, Label l') = targetnode a› f' ‹length rets = length outs›
    show ?thesis by simp
  next
    case (ProcReturn px insx outsx cx l p' es rets l' ins' outs' c' ps)
    from ‹λcf. snd cf = (px, Label l')↩p'λcf cf'. cf'(rets [:=] map cf outs') =
      kind a› ‹kind a = Q'↩pf'›
    have p' = p and f':f' = (λcf cf'. cf'(rets [:=] map cf outs'))
      by simp-all
    with ‹well-formed procs› ‹(p', ins', outs', c') ∈ set procs›
      ‹(p, ins, outs) ∈ set (lift-procs wfp)›
    have [simp]:outs' = outs by fastforce
    from ‹cx ⊢ Label l −CEdge (p', es, rets)→p Label l'›
    have containsCall procs cx [] p' by(rule Proc-CFG-Call-containsCall)
    with ‹containsCall procs prog ps px› ‹(px, insx, outsx, cx) ∈ set procs›
    have containsCall procs prog (ps@[px]) p' by(rule containsCall-in-proc)
    with ‹wf prog procs› ‹(p', ins', outs', c') ∈ set procs›
      ‹cx ⊢ Label l −CEdge (p', es, rets)→p Label l'›
    have length rets = length outs by fastforce
    from ‹(px, insx, outsx, cx) ∈ set procs›
      ‹cx ⊢ Label l −CEdge (p', es, rets)→p Label l'›
    have ParamDefs wfp (px,Label l') = rets
      by(fastforce intro:ParamDefs-Proc-Return-target simp:set-conv-nth)
    with ‹(px, Label l') = targetnode a› f' ‹length rets = length outs›
    show ?thesis by simp
  qed auto
  thus ?length and ?update by simp-all
qed
```

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(induct e arbitrary:v')
  case (Val v)
  from ‹state-check cf (Val v) v'› have v' = Some v
    by(fastforce elim:interpret.cases)
  thus ?case by simp
next
```

```
  case (Var V)
  hence cf' (V) = v' by(fastforce elim:interpret.cases)
  thus ?case by simp
next
  case (BinOp b1 bop b2)
  note IH1 = ⟨⋀v'. ⟦state-check cf b1 v'; ∀ V∈fv b1. cf V = cf' V⟧
    ⟹ state-check cf' b1 v'⟩
  note IH2 = ⟨⋀v'. ⟦state-check cf b2 v'; ∀ V∈fv b2. cf V = cf' V⟧
    ⟹ state-check cf' b2 v'⟩
  from ⟨∀ V ∈ fv (b1 ≪bop≫ b2). cf V = cf' V⟩ have ∀ V ∈ fv b1. cf V = cf' V
    and ∀ V ∈ fv b2. cf V = cf' V by simp-all
  from ⟨state-check cf (b1 ≪bop≫ b2) v'⟩
  have ((state-check cf b1 None ∧ v' = None) ∨
        (state-check cf b2 None ∧ v' = None)) ∨
    (∃ v1 v2. state-check cf b1 (Some v1) ∧ state-check cf b2 (Some v2) ∧
    binop bop v1 v2 = v')
    apply(cases interpret b1 cf,simp)
    apply(cases interpret b2 cf,simp)
    by(case-tac binop bop a aa,simp+)
  thus ?case apply −
  proof(erule disjE)+
    assume state-check cf b1 None ∧ v' = None
    hence check:state-check cf b1 None and v' = None by simp-all
   from IH1[OF check ⟨∀ V ∈ fv b1. cf V = cf' V⟩] have state-check cf' b1 None
.
    with ⟨v' = None⟩ show ?case by simp
  next
    assume state-check cf b2 None ∧ v' = None
    hence check:state-check cf b2 None and v' = None by simp-all
   from IH2[OF check ⟨∀ V ∈ fv b2. cf V = cf' V⟩] have state-check cf' b2 None
.
    with ⟨v' = None⟩ show ?case by(cases interpret b1 cf') simp+
  next
    assume ∃ v1 v2. state-check cf b1 (Some v1) ∧
      state-check cf b2 (Some v2) ∧ binop bop v1 v2 = v'
    then obtain v1 v2 where state-check cf b1 (Some v1)
      and state-check cf b2 (Some v2) and binop bop v1 v2 = v' by blast
    from ⟨∀ V ∈ fv (b1 ≪bop≫ b2). cf V = cf' V⟩ have ∀ V ∈ fv b1. cf V = cf'
V
      by simp
    from IH1[OF ⟨state-check cf b1 (Some v1)⟩ this]
    have interpret b1 cf' = Some v1 .
    from ⟨∀ V ∈ fv (b1 ≪bop≫ b2). cf V = cf' V⟩ have ∀ V ∈ fv b2. cf V = cf'
V
      by simp
    from IH2[OF ⟨state-check cf b2 (Some v2)⟩ this]
    have interpret b2 cf' = Some v2 .
    with ⟨interpret b1 cf' = Some v1⟩ ⟨binop bop v1 v2 = v'⟩
    show ?thesis by(cases v') simp+
```

qed
qed

lemma *PCFG-CallEdge-THE-es*:
prog $\vdash$ *n* $-$*CEdge*(*p*,*es*,*rets*)$\rightarrow_p$ *n'*
$\Longrightarrow$ (*THE es'*. $\exists$ *p' rets' n'*. *prog* $\vdash$ *n* $-$*CEdge*(*p'*,*es'*,*rets'*)$\rightarrow_p$ *n'*) = *es*
by(*fastforce intro*:*the-equality dest*:*Proc-CFG-Call-nodes-eq*)

definition *ParamUses-proc* :: *cmd* $\Rightarrow$ *label* $\Rightarrow$ *vname set list*
where *ParamUses-proc c n* $\equiv$
if ($\exists$ *n' p' es' rets'*. *c* $\vdash$ *n* $-$*CEdge*(*p'*,*es'*,*rets'*)$\rightarrow_p$ *n'*) *then*
(*map fv* (*THE es'*. $\exists$ *p' rets' n'*. *c* $\vdash$ *n* $-$*CEdge*(*p'*,*es'*,*rets'*)$\rightarrow_p$ *n'*))
else []

definition *ParamUses* :: *wf-prog* $\Rightarrow$ *node* $\Rightarrow$ *vname set list*
where *ParamUses wfp n* $\equiv$ *let* (*prog*,*procs*) = *Rep-wf-prog wfp*; (*p*,*l*) = *n in*
(*if* (*p* = *Main*) *then ParamUses-proc prog l*
else (*if* ($\exists$ *ins outs c*. (*p*,*ins*,*outs*,*c*) $\in$ *set procs*)
then ParamUses-proc (*THE c'*. $\exists$ *ins' outs'*. (*p*,*ins'*,*outs'*,*c'*) $\in$ *set procs*) *l*
else []))

lemma *ParamUses-Main-Return-target*:
⟦*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* $\vdash$ *n* $-$*CEdge*(*p'*,*es*,*rets*)$\rightarrow_p$ *n'* ⟧
$\Longrightarrow$ *ParamUses wfp* (*Main*,*n*) = *map fv es*
by(*fastforce dest*:*PCFG-CallEdge-THE-es simp*:*ParamUses-def ParamUses-proc-def*)

lemma *ParamUses-Proc-Return-target*:
assumes *Rep-wf-prog wfp* = (*prog*,*procs*)
and (*p*,*ins*,*outs*,*c*) $\in$ *set procs* **and** *c* $\vdash$ *n* $-$*CEdge*(*p'*,*es*,*rets*)$\rightarrow_p$ *n'*
shows *ParamUses wfp* (*p*,*n*) = *map fv es*
proof $-$
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
with ‹(*p*,*ins*,*outs*,*c*) $\in$ *set procs*› **have** *p* $\neq$ *Main* **by** *fastforce*
moreover
from ‹*well-formed procs*› ‹(*p*,*ins*,*outs*,*c*) $\in$ *set procs*›
have (*THE c'*. $\exists$ *ins' outs'*. (*p*,*ins'*,*outs'*,*c'*) $\in$ *set procs*) = *c*
by(*rule in-procs-THE-in-procs-cmd*)
ultimately show *?thesis* **using** *assms*
by(*fastforce dest*:*PCFG-CallEdge-THE-es simp*:*ParamUses-def ParamUses-proc-def*)
qed

lemma *ParamUses-Main-IEdge-Nil*:
⟦*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* $\vdash$ *n* $-$*IEdge et*$\rightarrow_p$ *n'*⟧

$\Longrightarrow$ *ParamUses wfp* (*Main*,*n*) = []
by(*fastforce dest*:*Proc-CFG-Call-Intra-edge-not-same-source*
simp:*ParamUses-def ParamUses-proc-def*)

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 −
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
with ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*› **have** *p* ≠ *Main* **by** *fastforce*
moreover
from ‹*well-formed procs*› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
have (*THE c'*. ∃ *ins' outs'*. (*p*,*ins'*,*outs'*,*c'*) ∈ *set procs*) = *c*
by(*rule in-procs-THE-in-procs-cmd*)
ultimately show *?thesis* **using** *assms*
by(*fastforce dest*:*Proc-CFG-Call-Intra-edge-not-same-source*
simp:*ParamUses-def ParamUses-proc-def*)
qed

lemma *ParamUses-Main-CEdge-Nil*:
⟦*Rep-wf-prog wfp* = (*prog*,*procs*); *prog* ⊢ *n'* −*CEdge*(*p'*,*es*,*rets*)→$_p$ *n*⟧
$\Longrightarrow$ *ParamUses wfp* (*Main*,*n*) = []
by(*fastforce dest*:*Proc-CFG-Call-targetnode-no-Call-sourcenode*
simp:*ParamUses-def ParamUses-proc-def*)

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 −
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
with ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*› **have** *p* ≠ *Main* **by** *fastforce*
moreover
from ‹*well-formed procs*›
‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
have (*THE c'*. ∃ *ins' outs'*. (*p*,*ins'*,*outs'*,*c'*) ∈ *set procs*) = *c*
by(*rule in-procs-THE-in-procs-cmd*)
ultimately show *?thesis* **using** *assms*
by(*fastforce dest*:*Proc-CFG-Call-targetnode-no-Call-sourcenode*
simp:*ParamUses-def ParamUses-proc-def*)
qed

Def

fun *lhs* :: *cmd* ⇒ *vname set*
where

```
  lhs Skip                = {}
  | lhs (V:=e)              = {V}
  | lhs (c1;;c2)            = lhs c1
  | lhs (if (b) c1 else c2) = {}
  | lhs (while (b) c)       = {}
  | lhs (Call p es rets)    = {}

lemma lhs-fst-cmd:lhs (fst-cmd c) = lhs c by(induct c) auto

lemma Proc-CFG-Call-source-empty-lhs:
  assumes prog ⊢ Label l −CEdge (p,es,rets)→p n'
  shows lhs (label prog l) = {}
proof −
  from ‹prog ⊢ Label l −CEdge (p,es,rets)→p n'› have l < #:prog
    by(rule Proc-CFG-sourcelabel-less-num-nodes)
  then obtain c' where labels prog l c'
    by(erule less-num-inner-nodes-label)
  hence label prog l = c' by(rule labels-label)
  from ‹labels prog l c'› ‹prog ⊢ Label l −CEdge (p,es,rets)→p n'›
  have ∃p es rets. fst-cmd c' = Call p es rets
    by(rule Proc-CFG-Call-source-fst-cmd-Call)
  with lhs-fst-cmd[of c'] have lhs c' = {} by auto
  with ‹label prog l = c'› show ?thesis by simp
qed


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
  by(fastforce intro:the-equality)


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 −
  obtain prog procs where [simp]:Rep-wf-prog wfp = (prog,procs)
```

by(*cases Rep-wf-prog wfp*) *auto*
hence *well-formed procs* **by**(*fastforce intro*:*wf-wf-prog*)
thus *?thesis* **by**(*auto simp*:*Def-def ParamDefs-def ParamDefs-proc-def*)
qed

lemma *Exit-Def-empty*:*Def wfp* (*Main*, *Exit*) = {}
proof −
obtain *prog procs* **where** [*simp*]:*Rep-wf-prog wfp* = (*prog*,*procs*)
by(*cases Rep-wf-prog wfp*) *auto*
hence *well-formed procs* **by**(*fastforce intro*:*wf-wf-prog*)
thus *?thesis*
by(*auto dest*:*Proc-CFG-Call-Labels simp*:*Def-def ParamDefs-def ParamDefs-proc-def*)
qed

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* **by**(*induct c*) *auto*

lemma *Proc-CFG-Call-target-empty-rhs*:
assumes *prog* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *Label l*′
shows *rhs* (*label prog l*′) = {}
proof −
from ‹*prog* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *Label l*′› **have** *l*′ < #:*prog*
by(*rule Proc-CFG-targetlabel-less-num-nodes*)
then obtain *c*′ **where** *labels prog l*′ *c*′
by(*erule less-num-inner-nodes-label*)
hence *label prog l*′ = *c*′ **by**(*rule labels-label*)
from ‹*labels prog l*′ *c*′› ‹*prog* ⊢ *n* −*CEdge* (*p*,*es*,*rets*)→$_p$ *Label l*′›
have *fst-cmd c*′ = *Skip* **by**(*rule Proc-CFG-Call-target-fst-cmd-Skip*)
with *rhs-fst-cmd*[*of c*′] **have** *rhs c*′ = {} **by** *simp*
with ‹*label prog l*′ = *c*′› **show** *?thesis* **by** *simp*
qed

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*
by(*fastforce intro*:*the-equality*)

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 −
obtain *prog procs* **where** [*simp*]:*Rep-wf-prog wfp* = (*prog*,*procs*)
by(*cases Rep-wf-prog wfp*) *auto*
hence *well-formed procs* **by**(*fastforce intro*:*wf-wf-prog*)
thus *?thesis* **by**(*auto dest*:*Proc-CFG-Call-Labels*
simp:*Use-def ParamUses-def ParamUses-proc-def ParamDefs-def ParamDefs-proc-def*)
qed

lemma *Exit-Use-empty*:*Use wfp* (*Main*, *Exit*) = {}
proof −
obtain *prog procs* **where** [*simp*]:*Rep-wf-prog wfp* = (*prog*,*procs*)
by(*cases Rep-wf-prog wfp*) *auto*
hence *well-formed procs* **by**(*fastforce intro*:*wf-wf-prog*)
thus *?thesis* **by**(*auto dest*:*Proc-CFG-Call-Labels*
simp:*Use-def ParamUses-def ParamUses-proc-def ParamDefs-def ParamDefs-proc-def*)
qed

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 −

from ‹*prog* ⊢ *Label l* −*IEdge et*→$_p$ *n'*›
obtain *x* **where** *IEdge et* = *x* **and** *prog* ⊢ *Label l* −*x*→$_p$ *n'* **by** *simp-all*
from ‹*prog* ⊢ *Label l* −*x*→$_p$ *n'*› ‹*IEdge et* = *x*› ‹*V* ∉ *lhs* (*label prog l*)›
show *?thesis*
proof(*induct prog Label l x n' arbitrary*:*l rule*:*Proc-CFG.induct*)
case (*Proc-CFG-LAss V' e*)
have *labels* (*V'*:=*e*) *0* (*V'*:=*e*) **by**(*rule Labels-Base*)
hence *label* (*V'*:=*e*) *0* = (*V'*:=*e*) **by**(*rule labels-label*)
have *V'* ∈ *lhs* (*V'*:=*e*) **by** *simp*
with ‹*V* ∉ *lhs* (*label* (*V'*:=*e*) *0*)›
‹*IEdge et* = *IEdge* ⇑λ*cf*. *update cf V' e*› ‹*label* (*V'*:=*e*) *0* = (*V'*:=*e*)›
show *?case* **by** *fastforce*
next
case (*Proc-CFG-SeqFirst* c_1 *et' n' c*$_2$)
note *IH* = ‹⟦*IEdge et* = *et'*; *V* ∉ *lhs* (*label* c_1 *l*)⟧
⟹ *state-val* (*CFG.transfer* (*lift-procs wfp*) *et* (*cf* # *cfs*)) *V* = *fst cf V*›
from ‹c_1 ⊢ *Label l* −*et'*→$_p$ *n'*› **have** *l* < #:c_1
by(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *c'* **where** *labels* c_1 *l c'* **by**(*erule less-num-inner-nodes-label*)
hence *labels* (c_1;;c_2) *l* (*c'*;;c_2) **by**(*rule Labels-Seq1*)
hence *label* (c_1;;c_2) *l* = *c'*;;c_2 **by**(*rule labels-label*)
with ‹*V* ∉ *lhs* (*label* (c_1;; c_2) *l*)› ‹*labels* c_1 *l c'*›
have *V* ∉ *lhs* (*label* c_1 *l*) **by**(*fastforce dest*:*labels-label*)
with ‹*IEdge et* = *et'*› **show** *?case* **by** (*rule IH*)
next
case (*Proc-CFG-SeqConnect* c_1 *et' c*$_2$)
note *IH* = ‹⟦*IEdge et* = *et'*; *V* ∉ *lhs* (*label* c_1 *l*)⟧
⟹ *state-val* (*CFG.transfer* (*lift-procs wfp*) *et* (*cf* # *cfs*)) *V* = *fst cf V*›
from ‹c_1 ⊢ *Label l* −*et'*→$_p$ *Exit*› **have** *l* < #:c_1
by(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *c'* **where** *labels* c_1 *l c'* **by**(*erule less-num-inner-nodes-label*)
hence *labels* (c_1;;c_2) *l* (*c'*;;c_2) **by**(*rule Labels-Seq1*)
hence *label* (c_1;;c_2) *l* = *c'*;;c_2 **by**(*rule labels-label*)
with ‹*V* ∉ *lhs* (*label* (c_1;; c_2) *l*)› ‹*labels* c_1 *l c'*›
have *V* ∉ *lhs* (*label* c_1 *l*) **by**(*fastforce dest*:*labels-label*)
with ‹*IEdge et* = *et'*› **show** *?case* **by** (*rule IH*)
next
case (*Proc-CFG-SeqSecond* c_2 *n et' n' c*$_1$ *l*)
note *IH* = ‹⋀*l*. ⟦*n* = *Label l*; *IEdge et* = *et'*; *V* ∉ *lhs* (*label* c_2 *l*)⟧
⟹ *state-val* (*CFG.transfer* (*lift-procs wfp*) *et* (*cf* # *cfs*)) *V* = *fst cf V*›
from ‹*n* ⊕ #:c_1 = *Label l*› **obtain** *l'*
where *n* = *Label l'* **and** *l* = *l'* + #:c_1 **by**(*cases n*) *auto*
from ‹*n* = *Label l'*› ‹c_2 ⊢ *n* −*et'*→$_p$ *n'*› **have** *l'* < #:c_2
by(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *c'* **where** *labels* c_2 *l' c'* **by**(*erule less-num-inner-nodes-label*)
with ‹*l* = *l'* + #:c_1› **have** *labels* (c_1;;c_2) *l c'*
by(*fastforce intro*:*Labels-Seq2*)
hence *label* (c_1;;c_2) *l* = *c'* **by**(*rule labels-label*)
with ‹*V* ∉ *lhs* (*label* (c_1;;c_2) *l*)› ‹*labels* c_2 *l' c'*› ‹*l* = *l'* + #:c_1›

have $V \notin lhs\ (label\ c_2\ l')$ **by**(*fastforce dest*:*labels-label*)
with ‹$n = Label\ l'$› ‹$IEdge\ et = et'$› **show** *?case* **by** (*rule IH*)
next
case (*Proc-CFG-CondThen* $c_1\ n\ et'\ n'\ b\ c_2\ l$)
note $IH = ‹\bigwedge l.\ [\![n = Label\ l;\ IEdge\ et = et';\ V \notin lhs\ (label\ c_1\ l)]\!]$
$\Longrightarrow state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ (cf\ \#\ cfs))\ V = fst\ cf\ V›$
from ‹$n \oplus 1 = Label\ l$› **obtain** l'
where $n = Label\ l'$ **and** $l = l' + 1$ **by**(*cases n*) *auto*
from ‹$n = Label\ l'$› ‹$c_1 \vdash n\ -et'\rightarrow_p\ n'$› **have** $l' < \#{:}c_1$
by(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain c' **where** *labels* $c_1\ l'\ c'$ **by**(*erule less-num-inner-nodes-label*)
with ‹$l = l' + 1$› **have** *labels* $(if\ (b)\ c_1\ else\ c_2)\ l\ c'$
by(*fastforce intro*:*Labels-CondTrue*)
hence $label\ (if\ (b)\ c_1\ else\ c_2)\ l = c'$ **by**(*rule labels-label*)
with ‹$V \notin lhs\ (label\ (if\ (b)\ c_1\ else\ c_2)\ l)$› ‹*labels* $c_1\ l'\ c'$› ‹$l = l' + 1$›
have $V \notin lhs\ (label\ c_1\ l')$ **by**(*fastforce dest*:*labels-label*)
with ‹$n = Label\ l'$› ‹$IEdge\ et = et'$› **show** *?case* **by** (*rule IH*)
next
case (*Proc-CFG-CondElse* $c_2\ n\ et'\ n'\ b\ c_1\ l$)
note $IH = ‹\bigwedge l.\ [\![n = Label\ l;\ IEdge\ et = et';\ V \notin lhs\ (label\ c_2\ l)]\!]$
$\Longrightarrow state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ (cf\ \#\ cfs))\ V = fst\ cf\ V›$
from ‹$n \oplus \#{:}c_1 + 1 = Label\ l$› **obtain** l'
where $n = Label\ l'$ **and** $l = l' + \#{:}c_1 + 1$ **by**(*cases n*) *auto*
from ‹$n = Label\ l'$› ‹$c_2 \vdash n\ -et'\rightarrow_p\ n'$› **have** $l' < \#{:}c_2$
by(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain c' **where** *labels* $c_2\ l'\ c'$ **by**(*erule less-num-inner-nodes-label*)
with ‹$l = l' + \#{:}c_1 + 1$› **have** *labels* $(if\ (b)\ c_1\ else\ c_2)\ l\ c'$
by(*fastforce intro*:*Labels-CondFalse*)
hence $label\ (if\ (b)\ c_1\ else\ c_2)\ l = c'$ **by**(*rule labels-label*)
with ‹$V \notin lhs\ (label\ (if\ (b)\ c_1\ else\ c_2)\ l)$› ‹*labels* $c_2\ l'\ c'$› ‹$l = l' + \#{:}c_1 + 1$›
have $V \notin lhs\ (label\ c_2\ l')$ **by**(*fastforce dest*:*labels-label*)
with ‹$n = Label\ l'$› ‹$IEdge\ et = et'$› **show** *?case* **by** (*rule IH*)
next
case (*Proc-CFG-WhileBody* $c'\ n\ et'\ n'\ b\ l$)
note $IH = ‹\bigwedge l.\ [\![n = Label\ l;\ IEdge\ et = et';\ V \notin lhs\ (label\ c'\ l)]\!]$
$\Longrightarrow state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ (cf\ \#\ cfs))\ V = fst\ cf\ V›$
from ‹$n \oplus 2 = Label\ l$› **obtain** l'
where $n = Label\ l'$ **and** $l = l' + 2$ **by**(*cases n*) *auto*
from ‹$n = Label\ l'$› ‹$c' \vdash n\ -et'\rightarrow_p\ n'$› **have** $l' < \#{:}c'$
by(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *cx* **where** *labels* $c'\ l'\ cx$ **by**(*erule less-num-inner-nodes-label*)
with ‹$l = l' + 2$› **have** *labels* $(while\ (b)\ c')\ l\ (cx;;while\ (b)\ c')$
by(*fastforce intro*:*Labels-WhileBody*)
hence $label\ (while\ (b)\ c')\ l = cx;;while\ (b)\ c'$ **by**(*rule labels-label*)
with ‹$V \notin lhs\ (label\ (while\ (b)\ c')\ l)$› ‹*labels* $c'\ l'\ cx$› ‹$l = l' + 2$›
have $V \notin lhs\ (label\ c'\ l')$ **by**(*fastforce dest*:*labels-label*)
with ‹$n = Label\ l'$› ‹$IEdge\ et = et'$› **show** *?case* **by** (*rule IH*)
next
case (*Proc-CFG-WhileBodyExit* $c'\ n\ et'\ b\ l$)

note *IH* = ‹$\bigwedge l$. $[\![n = Label\ l; IEdge\ et = et'; V \notin lhs\ (label\ c'\ l)]\!]$
$\Longrightarrow$ *state-val* (*CFG.transfer* (*lift-procs wfp*) *et* (*cf* # *cfs*)) *V* = *fst cf V*›
from ‹*n* ⊕ *2* = *Label l*› **obtain** *l'*
where *n* = *Label l'* **and** *l* = *l'* + *2* **by**(*cases n*) *auto*
from ‹*n* = *Label l'*› ‹*c'* ⊢ *n* −*et'*→$_p$ *Exit*› **have** *l'* < #:*c'*
by(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *cx* **where** *labels c' l' cx* **by**(*erule less-num-inner-nodes-label*)
with ‹*l* = *l'* + *2*› **have** *labels* (*while* (*b*) *c'*) *l* (*cx*;;*while* (*b*) *c'*)
by(*fastforce intro*:*Labels-WhileBody*)
hence *label* (*while* (*b*) *c'*) *l* = *cx*;;*while* (*b*) *c'* **by**(*rule labels-label*)
with ‹*V* ∉ *lhs* (*label* (*while* (*b*) *c'*) *l*)› ‹*labels c' l' cx*› ‹*l* = *l'* + *2*›
have *V* ∉ *lhs* (*label c' l'*) **by**(*fastforce dest*:*labels-label*)
with ‹*n* = *Label l'*› ‹*IEdge et* = *et'*› **show** *?case* **by** (*rule IH*)
qed *auto*
qed

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 −
from ‹*prog* ⊢ *Label l* −*IEdge et*→$_p$ *n'*›
obtain *x* **where** *IEdge et* = *x* **and** *prog* ⊢ *Label l* −*x*→$_p$ *n'* **by** *simp-all*
from ‹*CFG.pred et s*› **obtain** *cf cfs* **where** [*simp*]:*s* = *cf*#*cfs* **by**(*cases s*) *auto*
from ‹*CFG.pred et s'*› **obtain** *cf' cfs'* **where** [*simp*]:*s'* = *cf'*#*cfs'*
by(*cases s'*) *auto*
from ‹*prog* ⊢ *Label l* −*x*→$_p$ *n'*› ‹*IEdge et* = *x*›
‹∀ *V*∈*rhs* (*label prog l*). *state-val s V* = *state-val s' V*›
show *?thesis*
proof(*induct prog Label l x n' arbitrary*:*l rule*:*Proc-CFG.induct*)
case *Proc-CFG-Skip*
have *labels Skip 0 Skip* **by**(*rule Labels-Base*)
hence *label Skip 0* = *Skip* **by**(*rule labels-label*)
hence ∀ *V*. *V* ∉ *lhs* (*label Skip 0*) **by** *simp*
then show *?case* **by** *fastforce*
next
case (*Proc-CFG-LAss V e*)
have *labels* (*V*:=*e*) *0* (*V*:=*e*) **by**(*rule Labels-Base*)
hence *label* (*V*:=*e*) *0* = *V*:=*e* **by**(*rule labels-label*)
then have *lhs* (*label* (*V*:=*e*) *0*) = {*V*}
and *rhs* (*label* (*V*:=*e*) *0*) = *fv e* **by** *auto*
with ‹*IEdge et* = *IEdge* ↑↑λ*cf*. *update cf V e*›
‹∀ *V*∈*rhs* (*label* (*V*:=*e*) *0*). *state-val s V* = *state-val s' V*›
show *?case* **by**(*fastforce intro*:*rhs-interpret-eq*)
next

```
  case (Proc-CFG-LAssSkip V e)
  have labels (V:=e) 1 Skip by(rule Labels-LAss)
  hence label (V:=e) 1 = Skip by(rule labels-label)
  hence ∀ V'. V' ∉ lhs (label (V:=e) 1) by simp
  then show ?case by fastforce
next
  case (Proc-CFG-SeqFirst c1 et' n' c2)
  note IH = ‹⟦IEdge et = et';
    ∀ V∈rhs (label c1 l). state-val s V = state-val s' V⟧
    ⟹ ∀ V∈lhs (label c1 l). state-val (CFG.transfer (lift-procs wfp) et s) V =
      state-val (CFG.transfer (lift-procs wfp) et s') V›
  from ‹c1 ⊢ Label l -et'→p n'›
  have l < #:c1 by(fastforce intro:Proc-CFG-sourcelabel-less-num-nodes)
  then obtain c' where labels c1 l c' by(erule less-num-inner-nodes-label)
  hence labels (c1;;c2) l (c';;c2) by(rule Labels-Seq1)
  with ‹labels c1 l c'› ‹∀ V∈rhs (label (c1;; c2) l). state-val s V = state-val s' V›
  have ∀ V∈rhs (label c1 l). state-val s V = state-val s' V
    by(fastforce dest:labels-label)
  with ‹IEdge et = et'›
  have ∀ V∈lhs (label c1 l). state-val (CFG.transfer (lift-procs wfp) et s) V =
    state-val (CFG.transfer (lift-procs wfp) et s') V by (rule IH)
  with ‹labels c1 l c'› ‹labels (c1;;c2) l (c';;c2)›
  show ?case by(fastforce dest:labels-label)
next
  case (Proc-CFG-SeqConnect c1 et' c2)
  note IH = ‹⟦IEdge et = et';
    ∀ V∈rhs (label c1 l). state-val s V = state-val s' V⟧
    ⟹ ∀ V∈lhs (label c1 l). state-val (CFG.transfer (lift-procs wfp) et s) V =
      state-val (CFG.transfer (lift-procs wfp) et s') V›
  from ‹c1 ⊢ Label l -et'→p Exit›
  have l < #:c1 by(fastforce intro:Proc-CFG-sourcelabel-less-num-nodes)
  then obtain c' where labels c1 l c' by(erule less-num-inner-nodes-label)
  hence labels (c1;;c2) l (c';;c2) by(rule Labels-Seq1)
  with ‹labels c1 l c'› ‹∀ V∈rhs (label (c1;; c2) l). state-val s V = state-val s' V›
  have ∀ V∈rhs (label c1 l). state-val s V = state-val s' V
    by(fastforce dest:labels-label)
  with ‹IEdge et = et'›
  have ∀ V∈lhs (label c1 l). state-val (CFG.transfer (lift-procs wfp) et s) V =
    state-val (CFG.transfer (lift-procs wfp) et s') V by (rule IH)
  with ‹labels c1 l c'› ‹labels (c1;;c2) l (c';;c2)›
  show ?case by(fastforce dest:labels-label)
next
  case (Proc-CFG-SeqSecond c2 n et' n' c1)
  note IH = ‹⋀l. ⟦n = Label l; IEdge et = et';
    ∀ V∈rhs (label c2 l). state-val s V = state-val s' V⟧
    ⟹ ∀ V∈lhs (label c2 l). state-val (CFG.transfer (lift-procs wfp) et s) V =
      state-val (CFG.transfer (lift-procs wfp) et s') V›
  from ‹n ⊕ #:c1 = Label l› obtain l' where n = Label l' and l = l' + #:c1
    by(cases n) auto
```

from ‹$c_2 \vdash n\ -et'\!\!\rightarrow_p\ n'$› ‹$n = Label\ l'$›
have $l' < \#{:}c_2$ **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain c' **where** *labels* $c_2\ l'\ c'$ **by**(*erule less-num-inner-nodes-label*)
with ‹$l = l' + \#{:}c_1$› **have** *labels* $(c_1;;c_2)\ l\ c'$ **by**(*fastforce intro*:*Labels-Seq2*)
with ‹*labels* $c_2\ l'\ c'$› ‹$\forall\ V{\in}rhs\ (label\ (c_1;;c_2)\ l).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$›
have $\forall\ V{\in}rhs\ (label\ c_2\ l').\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$
by(*fastforce dest*:*labels-label*)
with ‹$n = Label\ l'$› ‹*IEdge et* = *et'*›
have $\forall\ V{\in}lhs\ (label\ c_2\ l').\ state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ s)\ V =$
state-val (*CFG.transfer* (*lift-procs wfp*) *et s'*) *V* **by** (*rule IH*)
with ‹*labels* $c_2\ l'\ c'$› ‹*labels* $(c_1;;c_2)\ l\ c'$›
show *?case* **by**(*fastforce dest*:*labels-label*)
next
case (*Proc-CFG-CondTrue b* $c_1\ c_2$)
have *labels* (*if* (*b*) c_1 *else* c_2) *0* (*if* (*b*) c_1 *else* c_2) **by**(*rule Labels-Base*)
hence *label* (*if* (*b*) c_1 *else* c_2) *0* = *if* (*b*) c_1 *else* c_2 **by**(*rule labels-label*)
hence $\forall\ V.\ V \notin lhs\ (label\ (if\ (b)\ c_1\ else\ c_2)\ 0)$ **by** *simp*
then show *?case* **by** *fastforce*
next
case (*Proc-CFG-CondFalse b* $c_1\ c_2$)
have *labels* (*if* (*b*) c_1 *else* c_2) *0* (*if* (*b*) c_1 *else* c_2) **by**(*rule Labels-Base*)
hence *label* (*if* (*b*) c_1 *else* c_2) *0* = *if* (*b*) c_1 *else* c_2 **by**(*rule labels-label*)
hence $\forall\ V.\ V \notin lhs\ (label\ (if\ (b)\ c_1\ else\ c_2)\ 0)$ **by** *simp*
then show *?case* **by** *fastforce*
next
case (*Proc-CFG-CondThen* $c_1\ n\ et'\ n'\ b\ c_2$)
note *IH* = ‹$\bigwedge l.\ [\![n = Label\ l;\ IEdge\ et = et';$
$\forall\ V{\in}rhs\ (label\ c_1\ l).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V]\!]$
$\Longrightarrow \forall\ V{\in}lhs\ (label\ c_1\ l).\ state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ s)\ V =$
state-val (*CFG.transfer* (*lift-procs wfp*) *et s'*) *V*›
from ‹$n \oplus 1 = Label\ l$› **obtain** l' **where** $n = Label\ l'$ **and** $l = l' + 1$
by(*cases n*) *auto*
from ‹$c_1 \vdash n\ -et'\!\!\rightarrow_p\ n'$› ‹$n = Label\ l'$›
have $l' < \#{:}c_1$ **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain c' **where** *labels* $c_1\ l'\ c'$ **by**(*erule less-num-inner-nodes-label*)
with ‹$l = l' + 1$› **have** *labels* (*if* (*b*) c_1 *else* c_2) *l c'*
by(*fastforce intro*:*Labels-CondTrue*)
with ‹*labels* $c_1\ l'\ c'$› ‹$\forall\ V{\in}rhs\ (label\ (if\ (b)\ c_1\ else\ c_2)\ l).\ state\text{-}val\ s\ V =$
state-val $s'\ V$›
have $\forall\ V{\in}rhs\ (label\ c_1\ l').\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$
by(*fastforce dest*:*labels-label*)
with ‹$n = Label\ l'$› ‹*IEdge et* = *et'*›
have $\forall\ V{\in}lhs\ (label\ c_1\ l').\ state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ s)\ V =$
state-val (*CFG.transfer* (*lift-procs wfp*) *et s'*) *V* **by** (*rule IH*)
with ‹*labels* $c_1\ l'\ c'$› ‹*labels* (*if* (*b*) c_1 *else* c_2) *l c'*›
show *?case* **by**(*fastforce dest*:*labels-label*)
next
case (*Proc-CFG-CondElse* $c_2\ n\ et'\ n'\ b\ c_1$)
note *IH* = ‹$\bigwedge l.\ [\![n = Label\ l;\ IEdge\ et = et';$

$\forall\, V{\in}rhs\ (label\ c_2\ l).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V\rrbracket$
$\Longrightarrow \forall\, V{\in}lhs\ (label\ c_2\ l).\ state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ s)\ V =$
$state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ s')\ V\rangle$
from $\langle n \oplus \#{:}c_1 + 1 = Label\ l\rangle$ **obtain** l' **where** $n = Label\ l'$ **and** $l = l' + \#{:}c_1{+}1$
by(*cases n*) *auto*
from $\langle c_2 \vdash n\ -et'\!\rightarrow_p n'\rangle\ \langle n = Label\ l'\rangle$
have $l' < \#{:}c_2$ **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain c' **where** *labels* $c_2\ l'\ c'$ **by**(*erule less-num-inner-nodes-label*)
with $\langle l = l' + \#{:}c_1 + 1\rangle$ **have** *labels* $(if\ (b)\ c_1\ else\ c_2)\ l\ c'$
by(*fastforce intro*:*Labels-CondFalse*)
with $\langle labels\ c_2\ l'\ c'\rangle\ \langle\forall\, V{\in}rhs\ (label\ (if\ (b)\ c_1\ else\ c_2)\ l).$
$state\text{-}val\ s\ V = state\text{-}val\ s'\ V\rangle$
have $\forall\, V{\in}rhs\ (label\ c_2\ l').\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$
by(*fastforce dest*:*labels-label*)
with $\langle n = Label\ l'\rangle\ \langle IEdge\ et = et'\rangle$
have $\forall\, V{\in}lhs\ (label\ c_2\ l').\ state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ s)\ V =$
$state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ s')\ V$ **by** (*rule IH*)
with $\langle labels\ c_2\ l'\ c'\rangle\ \langle labels\ (if\ (b)\ c_1\ else\ c_2)\ l\ c'\rangle$
show *?case* **by**(*fastforce dest*:*labels-label*)
next
case (*Proc-CFG-WhileTrue b c'*)
have *labels* $(while\ (b)\ c')\ 0\ (while\ (b)\ c')$ **by**(*rule Labels-Base*)
hence $label\ (while\ (b)\ c')\ 0 = while\ (b)\ c'$ **by**(*rule labels-label*)
hence $\forall\, V.\ V \notin lhs\ (label\ (while\ (b)\ c')\ 0)$ **by** *simp*
then show *?case* **by** *fastforce*
next
case (*Proc-CFG-WhileFalse b c'*)
have *labels* $(while\ (b)\ c')\ 0\ (while\ (b)\ c')$ **by**(*rule Labels-Base*)
hence $label\ (while\ (b)\ c')\ 0 = while\ (b)\ c'$ **by**(*rule labels-label*)
hence $\forall\, V.\ V \notin lhs\ (label\ (while\ (b)\ c')\ 0)$ **by** *simp*
then show *?case* **by** *fastforce*
next
case (*Proc-CFG-WhileFalseSkip b c'*)
have *labels* $(while\ (b)\ c')\ 1\ Skip$ **by**(*rule Labels-WhileExit*)
hence $label\ (while\ (b)\ c')\ 1 = Skip$ **by**(*rule labels-label*)
hence $\forall\, V.\ V \notin lhs\ (label\ (while\ (b)\ c')\ 1)$ **by** *simp*
then show *?case* **by** *fastforce*
next
case (*Proc-CFG-WhileBody c' n et' n' b*)
note $IH = \langle\bigwedge l.\ \llbracket n = Label\ l;\ IEdge\ et = et';$
$\forall\, V{\in}rhs\ (label\ c'\ l).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V\rrbracket$
$\Longrightarrow \forall\, V{\in}lhs\ (label\ c'\ l).\ state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ s)\ V =$
$state\text{-}val\ (CFG.transfer\ (lift\text{-}procs\ wfp)\ et\ s')\ V\rangle$
from $\langle n \oplus 2 = Label\ l\rangle$ **obtain** l' **where** $n = Label\ l'$ **and** $l = l' + 2$
by(*cases n*) *auto*
from $\langle c' \vdash n\ -et'\!\rightarrow_p n'\rangle\ \langle n = Label\ l'\rangle$
have $l' < \#{:}c'$ **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *cx* **where** *labels* $c'\ l'\ cx$ **by**(*erule less-num-inner-nodes-label*)

with ‹$l = l' + 2$› **have** *labels* (*while* (*b*) *c'*) *l* (*cx*;;*while* (*b*) *c'*)
by(*fastforce intro*:*Labels-WhileBody*)
with ‹*labels c' l' cx*› ‹∀ *V*∈*rhs* (*label* (*while* (*b*) *c'*) *l*).
state-val s V = *state-val s' V*›
have ∀ *V*∈*rhs* (*label c' l'*). *state-val s V* = *state-val s' V*
by(*fastforce dest*:*labels-label*)
with ‹n = *Label l'*› ‹*IEdge et* = *et'*›
have ∀ *V*∈*lhs* (*label c' l'*). *state-val* (*CFG.transfer* (*lift-procs wfp*) *et s*) *V* =
state-val (*CFG.transfer* (*lift-procs wfp*) *et s'*) *V* **by** (*rule IH*)
with ‹*labels c' l' cx*› ‹*labels* (*while* (*b*) *c'*) *l* (*cx*;;*while* (*b*) *c'*)›
show *?case* **by**(*fastforce dest*:*labels-label*)
next
case (*Proc-CFG-WhileBodyExit c' n et' b*)
note *IH* = ‹⋀*l*. ⟦*n* = *Label l*; *IEdge et* = *et'*;
∀ *V*∈*rhs* (*label c' l*). *state-val s V* = *state-val s' V*⟧
⟹ ∀ *V*∈*lhs* (*label c' l*). *state-val* (*CFG.transfer* (*lift-procs wfp*) *et s*) *V* =
state-val (*CFG.transfer* (*lift-procs wfp*) *et s'*) *V*›
from ‹$n \oplus 2$ = *Label l*› **obtain** *l'* **where** n = *Label l'* **and** $l = l' + 2$
by(*cases n*) *auto*
from ‹*c'* ⊢ n −*et'*→$_p$ *Exit*› ‹n = *Label l'*›
have $l' < \#{:}c'$ **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *cx* **where** *labels c' l' cx* **by**(*erule less-num-inner-nodes-label*)
with ‹$l = l' + 2$› **have** *labels* (*while* (*b*) *c'*) *l* (*cx*;;*while* (*b*) *c'*)
by(*fastforce intro*:*Labels-WhileBody*)
with ‹*labels c' l' cx*› ‹∀ *V*∈*rhs* (*label* (*while* (*b*) *c'*) *l*).
state-val s V = *state-val s' V*›
have ∀ *V*∈*rhs* (*label c' l'*). *state-val s V* = *state-val s' V*
by(*fastforce dest*:*labels-label*)
with ‹n = *Label l'*› ‹*IEdge et* = *et'*›
have ∀ *V*∈*lhs* (*label c' l'*). *state-val* (*CFG.transfer* (*lift-procs wfp*) *et s*) *V* =
state-val (*CFG.transfer* (*lift-procs wfp*) *et s'*) *V* **by** (*rule IH*)
with ‹*labels c' l' cx*› ‹*labels* (*while* (*b*) *c'*) *l* (*cx*;;*while* (*b*) *c'*)›
show *?case* **by**(*fastforce dest*:*labels-label*)
next
case (*Proc-CFG-CallSkip p es rets*)
have *labels* (*Call p es rets*) *1 Skip* **by**(*rule Labels-Call*)
hence *label* (*Call p es rets*) *1* = *Skip* **by**(*rule labels-label*)
hence ∀ *V*. *V* ∉ *lhs* (*label* (*Call p es rets*) *1*) **by** *simp*
then show *?case* **by** *fastforce*
qed *auto*
qed

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 −

from ‹*prog* ⊢ *Label l* −*IEdge et*→$_p$ *n'*›
obtain *x* **where** *IEdge et* = *x* **and** *prog* ⊢ *Label l* −*x*→$_p$ *n'* **by** *simp-all*
from ‹*CFG.pred et s*› **obtain** *cf cfs* **where** [*simp*]:*s* = *cf*#*cfs* **by**(*cases s*) *auto*
from ‹*length s* = *length s'*› **obtain** *cf' cfs'* **where** [*simp*]:*s'* = *cf'*#*cfs'*
by(*cases s'*) *auto*
from ‹*prog* ⊢ *Label l* −*x*→$_p$ *n'*› ‹*IEdge et* = *x*›
‹∀ *V*∈*rhs* (*label prog l*). *state-val s V* = *state-val s' V*›
show *?thesis*
proof(*induct prog Label l x n' arbitrary*:*l rule*:*Proc-CFG.induct*)
case (*Proc-CFG-SeqFirst* c_1 *et' n' c*$_2$)
note *IH* = ‹⟦*IEdge et* = *et'*; ∀ *V*∈*rhs* (*label* c_1 *l*).
state-val s V = *state-val s' V*⟧ ⟹ *CFG.pred et s'*›
from ‹c_1 ⊢ *Label l* −*et'*→$_p$ *n'*›
have *l* < #:c_1 **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *c'* **where** *labels* c_1 *l c'* **by**(*erule less-num-inner-nodes-label*)
hence *labels* (c_1;;c_2) *l* (*c'*;;c_2) **by**(*rule Labels-Seq1*)
with ‹*labels* c_1 *l c'*› ‹∀ *V*∈*rhs* (*label* (c_1;; c_2) *l*). *state-val s V* = *state-val s' V*›
have ∀ *V*∈*rhs* (*label* c_1 *l*). *state-val s V* = *state-val s' V*
by(*fastforce dest*:*labels-label*)
with ‹*IEdge et* = *et'*› **show** *?case* **by** (*rule IH*)
next
case (*Proc-CFG-SeqConnect* c_1 *et'* c_2)
note *IH* = ‹⟦*IEdge et* = *et'*;
∀ *V*∈*rhs* (*label* c_1 *l*). *state-val s V* = *state-val s' V*⟧
⟹ *CFG.pred et s'*›
from ‹c_1 ⊢ *Label l* −*et'*→$_p$ *Exit*›
have *l* < #:c_1 **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *c'* **where** *labels* c_1 *l c'* **by**(*erule less-num-inner-nodes-label*)
hence *labels* (c_1;;c_2) *l* (*c'*;;c_2) **by**(*rule Labels-Seq1*)
with ‹*labels* c_1 *l c'*› ‹∀ *V*∈*rhs* (*label* (c_1;; c_2) *l*). *state-val s V* = *state-val s' V*›
have ∀ *V*∈*rhs* (*label* c_1 *l*). *state-val s V* = *state-val s' V*
by(*fastforce dest*:*labels-label*)
with ‹*IEdge et* = *et'*› **show** *?case* **by** (*rule IH*)
next
case (*Proc-CFG-SeqSecond* c_2 *n et' n' c*$_1$)
note *IH* = ‹⋀*l*. ⟦*n* = *Label l*; *IEdge et* = *et'*;
∀ *V*∈*rhs* (*label* c_2 *l*). *state-val s V* = *state-val s' V*⟧
⟹ *CFG.pred et s'*›
from ‹*n* ⊕ #:c_1 = *Label l*› **obtain** *l'* **where** *n* = *Label l'* **and** *l* = *l'* + #:c_1
by(*cases n*) *auto*
from ‹c_2 ⊢ *n* −*et'*→$_p$ *n'*› ‹*n* = *Label l'*›
have *l'* < #:c_2 **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *c'* **where** *labels* c_2 *l' c'* **by**(*erule less-num-inner-nodes-label*)
with ‹*l* = *l'* + #:c_1› **have** *labels* (c_1;;c_2) *l c'* **by**(*fastforce intro*:*Labels-Seq2*)
with ‹*labels* c_2 *l' c'*› ‹∀ *V*∈*rhs* (*label* (c_1;;c_2) *l*). *state-val s V* = *state-val s' V*›
have ∀ *V*∈*rhs* (*label* c_2 *l'*). *state-val s V* = *state-val s' V*
by(*fastforce dest*:*labels-label*)
with ‹*n* = *Label l'*› ‹*IEdge et* = *et'*› **show** *?case* **by** (*rule IH*)
next

case (*Proc-CFG-CondTrue* b c_1 c_2)
from ‹*CFG.pred et s*› ‹*IEdge et* = *IEdge* (λcf. *state-check cf b* (*Some true*))$_{\surd}$›
have *state-check* (*fst cf*) b (*Some true*) **by** *simp*
moreover
have *labels* (*if* (b) c_1 *else* c_2) *0* (*if* (b) c_1 *else* c_2) **by**(*rule Labels-Base*)
hence *label* (*if* (b) c_1 *else* c_2) *0* = *if* (b) c_1 *else* c_2 **by**(*rule labels-label*)
with ‹$\forall$ V∈*rhs* (*label* (*if* (b) c_1 *else* c_2) *0*). *state-val s V* = *state-val s'* V›
have $\forall$ V∈ *fv b*. *state-val s V* = *state-val s'* V **by** *fastforce*
ultimately have *state-check* (*fst cf'*) b (*Some true*)
by *simp*(*rule rhs-interpret-eq*)
with ‹*IEdge et* = *IEdge* (λcf. *state-check cf b* (*Some true*))$_{\surd}$›
show *?case* **by** *simp*
next
case (*Proc-CFG-CondFalse* b c_1 c_2)
from ‹*CFG.pred et s*›
‹*IEdge et* = *IEdge* (λcf. *state-check cf b* (*Some false*))$_{\surd}$›
have *state-check* (*fst cf*) b (*Some false*) **by** *simp*
moreover
have *labels* (*if* (b) c_1 *else* c_2) *0* (*if* (b) c_1 *else* c_2) **by**(*rule Labels-Base*)
hence *label* (*if* (b) c_1 *else* c_2) *0* = *if* (b) c_1 *else* c_2 **by**(*rule labels-label*)
with ‹$\forall$ V∈*rhs* (*label* (*if* (b) c_1 *else* c_2) *0*). *state-val s V* = *state-val s'* V›
have $\forall$ V∈ *fv b*. *state-val s V* = *state-val s'* V **by** *fastforce*
ultimately have *state-check* (*fst cf'*) b (*Some false*)
by *simp*(*rule rhs-interpret-eq*)
with ‹*IEdge et* = *IEdge* (λcf. *state-check cf b* (*Some false*))$_{\surd}$›
show *?case* **by** *simp*
next
case (*Proc-CFG-CondThen* c_1 n *et'* n' b c_2)
note *IH* = ‹$\bigwedge l$. ⟦n = *Label* l; *IEdge et* = *et'*;
$\forall$ V∈*rhs* (*label* c_1 l). *state-val s V* = *state-val s'* V⟧
$\Longrightarrow$ *CFG.pred et s'*›
from ‹$n \oplus 1$ = *Label* l› **obtain** l' **where** n = *Label* l' **and** $l = l' + 1$
by(*cases* n) *auto*
from ‹$c_1 \vdash n -et'\rightarrow_p n'$› ‹$n$ = *Label* l'›
have $l' < \#{:}c_1$ **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain c' **where** *labels* c_1 l' c' **by**(*erule less-num-inner-nodes-label*)
with ‹$l = l' + 1$› **have** *labels* (*if* (b) c_1 *else* c_2) l c'
by(*fastforce intro*:*Labels-CondTrue*)
with ‹*labels* c_1 l' c'› ‹$\forall$ V∈*rhs* (*label* (*if* (b) c_1 *else* c_2) l).
state-val s V = *state-val s'* V›
have $\forall$ V∈*rhs* (*label* c_1 l'). *state-val s V* = *state-val s'* V
by(*fastforce dest*:*labels-label*)
with ‹n = *Label* l'› ‹*IEdge et* = *et'*› **show** *?case* **by** (*rule IH*)
next
case (*Proc-CFG-CondElse* c_2 n *et'* n' b c_1)
note *IH* = ‹$\bigwedge l$. ⟦n = *Label* l; *IEdge et* = *et'*;
$\forall$ V∈*rhs* (*label* c_2 l). *state-val s V* = *state-val s'* V⟧
$\Longrightarrow$ *CFG.pred et s'*›
from ‹$n \oplus \#{:}c_1 + 1$= *Label* l› **obtain** l' **where** n = *Label* l' **and** $l = l' +$

$\#{:}c_1{+}1$
by(*cases n*) *auto*
from ‹$c_2 \vdash n\ -et'\!\rightarrow_p n'$› ‹$n = Label\ l'$›
have $l' < \#{:}c_2$ **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain c' **where** *labels* $c_2\ l'\ c'$ **by**(*erule less-num-inner-nodes-label*)
with ‹$l = l' + \#{:}c_1 + 1$› **have** *labels* $(if\ (b)\ c_1\ else\ c_2)\ l\ c'$
by(*fastforce intro*:*Labels-CondFalse*)
with ‹*labels* $c_2\ l'\ c'$› ‹$\forall\ V{\in}rhs\ (label\ (if\ (b)\ c_1\ else\ c_2)\ l)$.
state-val s V = *state-val s' V*›
have $\forall\ V{\in}rhs\ (label\ c_2\ l').$ *state-val s V* = *state-val s' V*
by(*fastforce dest*:*labels-label*)
with ‹$n = Label\ l'$› ‹*IEdge et* = *et'*› **show** *?case* **by** (*rule IH*)
next
case (*Proc-CFG-WhileTrue b c'*)
from ‹*CFG.pred et s*› ‹*IEdge et* = *IEdge* $(\lambda cf.$ *state-check cf b* (*Some true*))$_{\surd}$›
have *state-check* (*fst cf*) *b* (*Some true*) **by** *simp*
moreover
have *labels* (*while* (*b*) *c'*) *0* (*while* (*b*) *c'*) **by**(*rule Labels-Base*)
hence *label* (*while* (*b*) *c'*) *0* = *while* (*b*) *c'* **by**(*rule labels-label*)
with ‹$\forall\ V{\in}rhs$ (*label* (*while* (*b*) *c'*) *0*). *state-val s V* = *state-val s' V*›
have $\forall\ V{\in}$ *fv b*. *state-val s V* = *state-val s' V* **by** *fastforce*
ultimately have *state-check* (*fst cf'*) *b* (*Some true*)
by *simp*(*rule rhs-interpret-eq*)
with ‹*IEdge et* = *IEdge* $(\lambda cf.$ *state-check cf b* (*Some true*))$_{\surd}$›
show *?case* **by** *simp*
next
case (*Proc-CFG-WhileFalse b c'*)
from ‹*CFG.pred et s*›
‹*IEdge et* = *IEdge* $(\lambda cf.$ *state-check cf b* (*Some false*))$_{\surd}$›
have *state-check* (*fst cf*) *b* (*Some false*) **by** *simp*
moreover
have *labels* (*while* (*b*) *c'*) *0* (*while* (*b*) *c'*) **by**(*rule Labels-Base*)
hence *label* (*while* (*b*) *c'*) *0* = *while* (*b*) *c'* **by**(*rule labels-label*)
with ‹$\forall\ V{\in}rhs$ (*label* (*while* (*b*) *c'*) *0*). *state-val s V* = *state-val s' V*›
have $\forall\ V{\in}$ *fv b*. *state-val s V* = *state-val s' V* **by** *fastforce*
ultimately have *state-check* (*fst cf'*) *b* (*Some false*)
by *simp*(*rule rhs-interpret-eq*)
with ‹*IEdge et* = *IEdge* $(\lambda cf.$ *state-check cf b* (*Some false*))$_{\surd}$›
show *?case* **by** *simp*
next
case (*Proc-CFG-WhileBody c' n et' n' b*)
note *IH* = ‹$\bigwedge l.$ ⟦$n = Label\ l$; *IEdge et* = *et'*;
$\forall\ V{\in}rhs$ (*label c' l*). *state-val s V* = *state-val s' V*⟧
$\Longrightarrow$ *CFG.pred et s'*›
from ‹$n \oplus 2 = Label\ l$› **obtain** l' **where** $n = Label\ l'$ **and** $l = l' + 2$
by(*cases n*) *auto*
from ‹$c' \vdash n\ -et'\!\rightarrow_p n'$› ‹$n = Label\ l'$›
have $l' < \#{:}c'$ **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *cx* **where** *labels* $c'\ l'$ *cx* **by**(*erule less-num-inner-nodes-label*)

with ‹$l = l' + 2$› **have** *labels* (*while* (*b*) *c'*) *l* (*cx*;;*while* (*b*) *c'*)
by(*fastforce intro*:*Labels-WhileBody*)
with ‹*labels c' l' cx*› ‹∀ *V*∈*rhs* (*label* (*while* (*b*) *c'*) *l*).
state-val s V = *state-val s' V*›
have ∀ *V*∈*rhs* (*label c' l'*). *state-val s V* = *state-val s' V*
by(*fastforce dest*:*labels-label*)
with ‹*n* = *Label l'*› ‹*IEdge et* = *et'*› **show** *?case* **by** (*rule IH*)
next
case (*Proc-CFG-WhileBodyExit c' n et' b*)
note *IH* = ‹⋀*l*. ⟦*n* = *Label l*; *IEdge et* = *et'*;
∀ *V*∈*rhs* (*label c' l*). *state-val s V* = *state-val s' V*⟧
⟹ *CFG.pred et s'*›
from ‹*n* ⊕ *2* = *Label l*› **obtain** *l'* **where** *n* = *Label l'* **and** *l* = *l'* + *2*
by(*cases n*) *auto*
from ‹*c'* ⊢ *n* −*et'*→$_p$ *Exit*› ‹*n* = *Label l'*›
have *l'* < #:*c'* **by**(*fastforce intro*:*Proc-CFG-sourcelabel-less-num-nodes*)
then obtain *cx* **where** *labels c' l' cx* **by**(*erule less-num-inner-nodes-label*)
with ‹*l* = *l'* + *2*› **have** *labels* (*while* (*b*) *c'*) *l* (*cx*;;*while* (*b*) *c'*)
by(*fastforce intro*:*Labels-WhileBody*)
with ‹*labels c' l' cx*› ‹∀ *V*∈*rhs* (*label* (*while* (*b*) *c'*) *l*).
state-val s V = *state-val s' V*›
have ∀ *V*∈*rhs* (*label c' l'*). *state-val s V* = *state-val s' V*
by(*fastforce dest*:*labels-label*)
with ‹*n* = *Label l'*› ‹*IEdge et* = *et'*› **show** *?case* **by** (*rule IH*)
qed *auto*
qed

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 −
from *Rep-wf-prog*[*of wfp*]
obtain *prog procs* **where** [*simp*]:*Rep-wf-prog wfp* = (*prog*,*procs*)
by(*fastforce simp*:*wf-prog-def*)
hence *wf prog procs* **by**(*rule wf-wf-prog*)
hence *wf*:*well-formed procs* **by** *fastforce*
show *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*)
proof
from *Entry-Def-empty Entry-Use-empty*
show *Def wfp* (*Main*, *Entry*) = {} ∧ *Use wfp* (*Main*, *Entry*) = {} **by** *simp*
next
fix *a Q r p fs ins outs*
assume *valid-edge wfp a* **and** *kind a* = *Q*:*r*↪$_p$*fs*

and $(p,\ ins,\ outs) \in set\ (lift\text{-}procs\ wfp)$
hence *prog,procs* $\vdash$ *sourcenode a* $-kind\ a\rightarrow$ *targetnode a*
by(*simp add*:*valid-edge-def*)
from *this* ‹*kind a* = *Q*:$r\hookrightarrow_p fs$› ‹$(p,\ ins,\ outs) \in set\ (lift\text{-}procs\ wfp)$›
show *length* (*ParamUses wfp* (*sourcenode a*)) = *length ins*
proof(*induct n*≡*sourcenode a et*≡*kind a n'*≡*targetnode a rule*:*PCFG.induct*)
case (*MainCall l p' es rets n' insx outsx cx*)
with *wf* **have** [*simp*]:*insx* = *ins* **by** *fastforce*
from ‹*prog* ⊢ *Label l* $-CEdge\ (p',\ es,\ rets)\rightarrow_p$ *n'*›
have *containsCall procs prog* [] *p'* **by**(*rule Proc-CFG-Call-containsCall*)
with ‹*wf prog procs*› ‹$(p',\ insx,\ outsx,\ cx) \in set\ procs$›
‹*prog* ⊢ *Label l* $-CEdge\ (p',\ es,\ rets)\rightarrow_p$ *n'*›
have *length es* = *length ins* **by** *fastforce*
from ‹*prog* ⊢ *Label l* $-CEdge\ (p',\ es,\ rets)\rightarrow_p$ *n'*›
have *ParamUses wfp* (*Main*, *Label l*) = *map fv es*
by(*fastforce intro*:*ParamUses-Main-Return-target*)
with ‹(*Main*, *Label l*) = *sourcenode a*› ‹*length es* = *length ins*›
show *?case* **by** *simp*
next
case (*ProcCall px insx outsx cx l p' es rets l' ins' outs' c' ps*)
with *wf* **have** [*simp*]:*ins'* = *ins* **by** *fastforce*
from ‹*cx* ⊢ *Label l* $-CEdge\ (p',\ es,\ rets)\rightarrow_p$ *Label l'*›
have *containsCall procs cx* [] *p'* **by**(*rule Proc-CFG-Call-containsCall*)
with ‹*containsCall procs prog ps px*› ‹$(px,\ insx,\ outsx,\ cx) \in set\ procs$›
have *containsCall procs prog* (*ps*@[*px*]) *p'* **by**(*rule containsCall-in-proc*)
with ‹*wf prog procs*› ‹$(p',\ ins',\ outs',\ c') \in set\ procs$›
‹*cx* ⊢ *Label l* $-CEdge\ (p',\ es,\ rets)\rightarrow_p$ *Label l'*›
have *length es* = *length ins* **by** *fastforce*
from ‹$(px,\ insx,\ outsx,\ cx) \in set\ procs$›
‹*cx* ⊢ *Label l* $-CEdge\ (p',\ es,\ rets)\rightarrow_p$ *Label l'*›
have *ParamUses wfp* (*px*,*Label l*) = *map fv es*
by(*fastforce intro*:*ParamUses-Proc-Return-target simp*:*set-conv-nth*)
with ‹(*px*, *Label l*) = *sourcenode a*› ‹*length es* = *length ins*›
show *?case* **by** *simp*
qed *auto*
next
fix *a* **assume** *valid-edge wfp a*
hence *prog,procs* ⊢ *sourcenode a* $-kind\ a\rightarrow$ *targetnode a*
by(*simp add*:*valid-edge-def*)
thus *distinct* (*ParamDefs wfp* (*targetnode a*))
proof(*induct sourcenode a kind a targetnode a rule*:*PCFG.induct*)
case (*Main n n'*)
from ‹*prog* ⊢ *n* $-IEdge\ (kind\ a)\rightarrow_p$ *n'*› ‹(*Main*, *n'*) = *targetnode a*›
have *ParamDefs wfp* (*Main*,*n'*) = [] **by**(*fastforce intro*:*ParamDefs-Main-IEdge-Nil*)
with ‹(*Main*, *n'*) = *targetnode a*› **show** *?case* **by** *simp*
next
case (*Proc p ins outs c n n'*)
from ‹$(p,\ ins,\ outs,\ c) \in set\ procs$› ‹*c* ⊢ *n* $-IEdge\ (kind\ a)\rightarrow_p$ *n'*›
have *ParamDefs wfp* (*p*,*n'*) = [] **by**(*fastforce intro*:*ParamDefs-Proc-IEdge-Nil*)

with ‹$(p,\ n') = targetnode\ a$› **show** *?case* **by** *simp*
next
case (*MainCall l p es rets n' ins outs c*)
with ‹$(p,\ ins,\ outs,\ c) \in set\ procs$› *wf* **have** [*simp*]:$p \neq Main$
by *fastforce*
from *wf* ‹$(p,\ ins,\ outs,\ c) \in set\ procs$›
have (*THE c'*. $\exists\, ins'\ outs'.\ (p,ins',outs',c') \in set\ procs) = c$
by(*rule in-procs-THE-in-procs-cmd*)
with ‹$(p,\ Entry) = targetnode\ a$›[*THEN sym*] **show** *?case*
by(*auto simp*:*ParamDefs-def ParamDefs-proc-def*)
next
case (*ProcCall p ins outs c l p' es' rets' l' ins' outs' c'*)
with ‹$(p',\ ins',\ outs',\ c') \in set\ procs$› *wf*
have [*simp*]:$p' \neq Main$ **by** *fastforce*
from *wf* ‹$(p',\ ins',\ outs',\ c') \in set\ procs$›
have (*THE cx*. $\exists\, insx\ outsx.\ (p',insx,outsx,cx) \in set\ procs) = c'$
by(*rule in-procs-THE-in-procs-cmd*)
with ‹$(p',\ Entry) = targetnode\ a$›[*THEN sym*] **show** *?case*
by(*fastforce simp*:*ParamDefs-def ParamDefs-proc-def*)
next
case (*MainReturn l p es rets l' ins outs c*)
from ‹$prog \vdash Label\ l -CEdge\ (p,\ es,\ rets) \rightarrow_p Label\ l'$›
have *containsCall procs prog* [] *p* **by**(*rule Proc-CFG-Call-containsCall*)
with ‹*wf prog procs*› ‹$(p,\ ins,\ outs,\ c) \in set\ procs$›
‹$prog \vdash Label\ l -CEdge\ (p,\ es,\ rets) \rightarrow_p Label\ l'$›
have *distinct rets* **by** *fastforce*
from ‹$prog \vdash Label\ l -CEdge\ (p,\ es,\ rets) \rightarrow_p Label\ l'$›
have *ParamDefs wfp* (*Main*,*Label l'*) = *rets*
by(*fastforce intro*:*ParamDefs-Main-Return-target*)
with ‹*distinct rets*› ‹$(Main,\ Label\ l') = targetnode\ a$› **show** *?case*
by(*fastforce simp*:*distinct-map inj-on-def*)
next
case (*ProcReturn p ins outs c l p' es' rets' l' ins' outs' c' ps*)
from ‹$c \vdash Label\ l -CEdge\ (p',\ es',\ rets') \rightarrow_p Label\ l'$›
have *containsCall procs c* [] *p'* **by**(*rule Proc-CFG-Call-containsCall*)
with ‹*containsCall procs prog ps p*› ‹$(p,\ ins,\ outs,\ c) \in set\ procs$›
have *containsCall procs prog* (*ps*@[*p*]) *p'* **by**(*rule containsCall-in-proc*)
with ‹*wf prog procs*› ‹$(p',\ ins',\ outs',\ c') \in set\ procs$›
‹$c \vdash Label\ l -CEdge\ (p',\ es',\ rets') \rightarrow_p Label\ l'$›
have *distinct rets'* **by** *fastforce*
from ‹$(p,\ ins,\ outs,\ c) \in set\ procs$›
‹$c \vdash Label\ l -CEdge\ (p',\ es',\ rets') \rightarrow_p Label\ l'$›
have *ParamDefs wfp* (*p*,*Label l'*) = *rets'*
by(*fastforce intro*:*ParamDefs-Proc-Return-target simp*:*set-conv-nth*)
with ‹*distinct rets'*› ‹$(p,\ Label\ l') = targetnode\ a$› **show** *?case*
by(*fastforce simp*:*distinct-map inj-on-def*)
next
case (*MainCallReturn n p es rets n'*)
from ‹$prog \vdash n -CEdge\ (p,\ es,\ rets) \rightarrow_p n'$›

have *containsCall procs prog* [] *p* **by**(*rule Proc-CFG-Call-containsCall*)
with ‹*wf prog procs*› **obtain** *ins outs c* **where** (*p*, *ins*, *outs*, *c*) ∈ *set procs*
by(*simp add*:*wf-def*) *blast*
with ‹*wf prog procs*› ‹*containsCall procs prog* [] *p*›
‹*prog* ⊢ *n* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n*′›
have *distinct rets* **by** *fastforce*
from ‹*prog* ⊢ *n* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n*′›
have *ParamDefs wfp* (*Main*,*n*′) = *rets*
by(*fastforce intro*:*ParamDefs-Main-Return-target*)
with ‹*distinct rets*› ‹(*Main*, *n*′) = *targetnode a*› **show** *?case*
by(*fastforce simp*:*distinct-map inj-on-def*)
next
case (*ProcCallReturn p ins outs c n p*′ *es*′ *rets*′ *n*′ *ps*)
from ‹*c* ⊢ *n* −*CEdge* (*p*′, *es*′, *rets*′)→$_p$ *n*′›
have *containsCall procs c* [] *p*′ **by**(*rule Proc-CFG-Call-containsCall*)
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*›
‹*containsCall procs prog ps p*›
obtain *wfp*′ **where** *Rep-wf-prog wfp*′ = (*c*,*procs*) **by**(*erule wfp-Call*)
hence *wf c procs* **by**(*rule wf-wf-prog*)
with ‹*containsCall procs c* [] *p*′› **obtain** *ins*′ *outs*′ *c*′
where (*p*′, *ins*′, *outs*′, *c*′) ∈ *set procs*
by(*simp add*:*wf-def*) *blast*
from ‹*containsCall procs prog ps p*› ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*›
‹*containsCall procs c* [] *p*′›
have *containsCall procs prog* (*ps*@[*p*]) *p*′ **by**(*rule containsCall-in-proc*)
with ‹*wf prog procs*› ‹(*p*′, *ins*′, *outs*′, *c*′) ∈ *set procs*›
‹*c* ⊢ *n* −*CEdge* (*p*′, *es*′, *rets*′)→$_p$ *n*′›
have *distinct rets*′ **by** *fastforce*
from ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹*c* ⊢ *n* −*CEdge* (*p*′, *es*′, *rets*′)→$_p$ *n*′›
have *ParamDefs wfp* (*p*,*n*′) = *rets*′
by(*fastforce intro*:*ParamDefs-Proc-Return-target*)
with ‹*distinct rets*′› ‹(*p*, *n*′) = *targetnode a*› **show** *?case*
by(*fastforce simp*:*distinct-map inj-on-def*)
qed
next
fix *a Q*′ *p f*′ *ins outs*
assume *valid-edge wfp a* **and** *kind a* = *Q*′↩$_p$*f*′
and (*p*, *ins*, *outs*) ∈ *set* (*lift-procs wfp*)
thus *length* (*ParamDefs wfp* (*targetnode a*)) = *length outs*
by(*rule ParamDefs-length*)
next
fix *n V* **assume** *CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) *n*
and *V* ∈ *set* (*ParamDefs wfp n*)
thus *V* ∈ *Def wfp n* **by**(*simp add*:*Def-def*)
next
fix *a Q r p fs ins outs V*
assume *valid-edge wfp a* **and** *kind a* = *Q*:*r*↪$_p$*fs*
and (*p*, *ins*, *outs*) ∈ *set* (*lift-procs wfp*) **and** *V* ∈ *set ins*
hence *prog*,*procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*

by(*simp add*:*valid-edge-def*)
from *this* ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹(*p*, *ins*, *outs*) ∈ *set* (*lift-procs wfp*)› ‹*V* ∈ *set ins*›
show *V* ∈ *Def wfp* (*targetnode a*)
proof(*induct n*≡*sourcenode a et*≡*kind a n'*≡*targetnode a rule*:*PCFG.induct*)
case (*MainCall l p' es rets n' insx outsx cx*)
with *wf* **have** [*simp*]:*insx* = *ins* **by** *fastforce*
from *wf* ‹(*p'*, *insx*, *outsx*, *cx*) ∈ *set procs*›
have (*THE ins'*. ∃ *c' outs'*. (*p'*,*ins'*,*outs'*,*c'*) ∈ *set procs*) =
insx **by**(*rule in-procs-THE-in-procs-ins*)
with ‹(*p'*, *Entry*) = *targetnode a*›[*THEN sym*] ‹*V* ∈ *set ins*›
‹(*p'*, *insx*, *outsx*, *cx*) ∈ *set procs*› **show** *?case* **by**(*auto simp*:*Def-def*)
next
case (*ProcCall px insx outsx cx l p' es rets l' ins' outs' c'*)
with *wf* **have** [*simp*]:*ins'* = *ins* **by** *fastforce*
from *wf* ‹(*p'*, *ins'*, *outs'*, *c'*) ∈ *set procs*›
have (*THE insx*. ∃ *cx outsx*. (*p'*,*insx*,*outsx*,*cx*) ∈ *set procs*) =
ins' **by**(*rule in-procs-THE-in-procs-ins*)
with ‹(*p'*, *Entry*) = *targetnode a*›[*THEN sym*] ‹*V* ∈ *set ins*›
‹(*p'*, *ins'*, *outs'*, *c'*) ∈ *set procs*› **show** *?case* **by**(*auto simp*:*Def-def*)
qed *auto*
next
fix *a Q r p fs*
assume *valid-edge wfp a* **and** *kind a* = *Q*:*r*↪$_p$*fs*
hence *prog*,*procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*
by(*simp add*:*valid-edge-def*)
from *this* ‹*kind a* = *Q*:*r*↪$_p$*fs*› **show** *Def wfp* (*sourcenode a*) = {}
proof(*induct n*≡*sourcenode a et*≡*kind a n'*≡*targetnode a rule*:*PCFG.induct*)
case (*MainCall l p' es rets n' insx outsx cx*)
from ‹(*Main*, *Label l*) = *sourcenode a*›[*THEN sym*]
‹*prog* ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *n'*›
have *ParamDefs wfp* (*sourcenode a*) = []
by(*fastforce intro*:*ParamDefs-Main-CEdge-Nil*)
with ‹*prog* ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *n'*›
‹(*Main*, *Label l*) = *sourcenode a*›[*THEN sym*]
show *?case* **by**(*fastforce dest*:*Proc-CFG-Call-source-empty-lhs simp*:*Def-def*)
next
case (*ProcCall px insx outsx cx l p' es' rets' l' ins' outs' c'*)
from ‹(*px*, *insx*, *outsx*, *cx*) ∈ *set procs*› *wf*
have [*simp*]:*px* ≠ *Main* **by** *fastforce*
from ‹*cx* ⊢ *Label l* −*CEdge* (*p'*, *es'*, *rets'*)→$_p$ *Label l'*›
have *lhs* (*label cx l*) = {} **by**(*rule Proc-CFG-Call-source-empty-lhs*)
from *wf* ‹(*px*, *insx*, *outsx*, *cx*) ∈ *set procs*›
have *THE*:(*THE c'*. ∃ *ins' outs'*. (*px*,*ins'*,*outs'*,*c'*) ∈ *set procs*) = *cx*
by(*rule in-procs-THE-in-procs-cmd*)
with ‹(*px*, *Label l*) = *sourcenode a*›[*THEN sym*]
‹*cx* ⊢ *Label l* −*CEdge* (*p'*, *es'*, *rets'*)→$_p$ *Label l'*› *wf*
have *ParamDefs wfp* (*sourcenode a*) = []
by(*fastforce dest*:*Proc-CFG-Call-targetnode-no-Call-sourcenode*

```
    [of - - - - - Label l] simp:ParamDefs-def ParamDefs-proc-def)
   with ‹(px, Label l) = sourcenode a›[THEN sym] ‹lhs (label cx l) = {}› THE
   show ?case by(auto simp:Def-def)
  qed auto
 next
  fix n V assume CFG.valid-node sourcenode targetnode (valid-edge wfp) n
   and V ∈ ⋃set (ParamUses wfp n)
  thus V ∈ Use wfp n by(fastforce simp:Use-def)
 next
  fix a Q p f ins outs V
  assume valid-edge wfp a and kind a = Q↩pf
   and (p, ins, outs) ∈ set (lift-procs wfp) and V ∈ set outs
  hence prog,procs ⊢ sourcenode a −kind a→ targetnode a
   by(simp add:valid-edge-def)
  from this ‹kind a = Q↩pf› ‹(p, ins, outs) ∈ set (lift-procs wfp)› ‹V ∈ set outs›
  show V ∈ Use wfp (sourcenode a)
  proof(induct sourcenode a kind a targetnode a rule:PCFG.induct)
   case (MainReturn l p' es rets l' insx outsx cx)
   with wf have [simp]:outsx = outs by fastforce
   from wf ‹(p', insx, outsx, cx) ∈ set procs›
   have (THE outs'. ∃ c' ins'. (p',ins',outs',c') ∈ set procs) =
     outsx by(rule in-procs-THE-in-procs-outs)
   with ‹(p', Exit) = sourcenode a›[THEN sym] ‹V ∈ set outs›
    ‹(p', insx, outsx, cx) ∈ set procs› show ?case by(auto simp:Use-def)
  next
   case (ProcReturn px insx outsx cx l p' es' rets' l' ins' outs' c')
   with wf have [simp]:outs' = outs by fastforce
   from wf ‹(p', ins', outs', c') ∈ set procs›
   have (THE outsx. ∃ cx insx. (p',insx,outsx,cx) ∈ set procs) =
     outs' by(rule in-procs-THE-in-procs-outs)
   with ‹(p', Exit) = sourcenode a›[THEN sym] ‹V ∈ set outs›
    ‹(p', ins', outs', c') ∈ set procs› show ?case by(auto simp:Use-def)
  qed auto
 next
  fix a V s
  assume valid-edge wfp a and V ∉ Def wfp (sourcenode a)
   and intra-kind (kind a) and CFG.pred (kind a) s
  hence prog,procs ⊢ sourcenode a −kind a→ targetnode a
   by(simp add:valid-edge-def)
  from this ‹V ∉ Def wfp (sourcenode a)› ‹intra-kind (kind a)› ‹CFG.pred (kind
a) s›
  show state-val (CFG.transfer (lift-procs wfp) (kind a) s) V = state-val s V
  proof(induct sourcenode a kind a targetnode a rule:PCFG.induct)
   case (Main n n')
    from ‹CFG.pred (kind a) s› obtain cf cfs where s = cf#cfs by(cases s)
auto
   show ?case
   proof(cases n)
    case (Label l)
```

```
      with ‹V ∉ Def wfp (sourcenode a)› ‹(Main, n) = sourcenode a›
      have V ∉ lhs (label prog l) by(fastforce simp:Def-def)
      with ‹prog ⊢ n −IEdge (kind a)→p n'› ‹n = Label l›
      have state-val (CFG.transfer (lift-procs wfp) (kind a) (cf#cfs)) V = fst cf
V
        by(fastforce intro:Proc-CFG-edge-no-lhs-equal)
      with ‹s = cf#cfs› show ?thesis by simp
    next
      case Entry
      with ‹prog ⊢ n −IEdge (kind a)→p n'› ‹s = cf#cfs›
      show ?thesis
      by(fastforce dest:Proc-CFG-EntryD simp:transfers-simps[of wfp,simplified])
    next
      case Exit
      with ‹prog ⊢ n −IEdge (kind a)→p n'› have False by fastforce
      thus ?thesis by simp
    qed
  next
    case (Proc p ins outs c n n')
    from ‹CFG.pred (kind a) s› obtain cf cfs where s = cf#cfs by(cases s)
auto
    from wf ‹(p, ins, outs, c) ∈ set procs›
    have THE1:(THE ins'. ∃ c' outs'. (p,ins',outs',c') ∈ set procs) = ins
      by(rule in-procs-THE-in-procs-ins)
    from wf ‹(p, ins, outs, c) ∈ set procs›
    have THE2:(THE c'. ∃ ins' outs'. (p,ins',outs',c') ∈ set procs) = c
      by(rule in-procs-THE-in-procs-cmd)
    from wf ‹(p, ins, outs, c) ∈ set procs›
    have [simp]:p ≠ Main by fastforce
    show ?case
    proof(cases n)
      case (Label l)
      with ‹V ∉ Def wfp (sourcenode a)› ‹(p, n) = sourcenode a›
        ‹(p, ins, outs, c) ∈ set procs› wf THE1 THE2
      have V ∉ lhs (label c l) by(fastforce simp:Def-def split:split-if-asm)
      with ‹c ⊢ n −IEdge (kind a)→p n'› ‹n = Label l›
      have state-val (CFG.transfer (lift-procs wfp) (kind a) (cf#cfs)) V = fst cf
V
        by(fastforce intro:Proc-CFG-edge-no-lhs-equal)
      with ‹s = cf#cfs› show ?thesis by simp
    next
      case Entry
      with ‹c ⊢ n −IEdge (kind a)→p n'› ‹s = cf#cfs›
      show ?thesis
      by(fastforce dest:Proc-CFG-EntryD simp:transfers-simps[of wfp,simplified])
    next
      case Exit
      with ‹c ⊢ n −IEdge (kind a)→p n'› have False by fastforce
      thus ?thesis by simp
```

```
    qed
  next
    case MainCallReturn thus ?case by(cases s,auto simp:intra-kind-def)
  next
    case ProcCallReturn thus ?case by(cases s,auto simp:intra-kind-def)
  qed(auto simp:intra-kind-def)
next
  fix a s s'
  assume valid-edge wfp a
    and ∀ V∈Use wfp (sourcenode a). state-val s V = state-val s' V
   and intra-kind (kind a) and CFG.pred (kind a) s and CFG.pred (kind a) s'
  hence prog,procs ⊢ sourcenode a −kind a→ targetnode a
    by(simp add:valid-edge-def)
  from ‹CFG.pred (kind a) s› obtain cf cfs where [simp]:s = cf#cfs
    by(cases s) auto
  from ‹CFG.pred (kind a) s'› obtain cf' cfs' where [simp]:s' = cf'#cfs'
    by(cases s') auto
  from ‹prog,procs ⊢ sourcenode a −kind a→ targetnode a› ‹intra-kind (kind a)›
    ‹∀ V∈Use wfp (sourcenode a). state-val s V = state-val s' V›
    ‹CFG.pred (kind a) s› ‹CFG.pred (kind a) s'›
  show ∀ V∈Def wfp (sourcenode a).
    state-val (CFG.transfer (lift-procs wfp) (kind a) s) V =
    state-val (CFG.transfer (lift-procs wfp) (kind a) s') V
  proof(induct sourcenode a kind a targetnode a rule:PCFG.induct)
    case (Main n n')
    show ?case
    proof(cases n)
      case (Label l)
      with ‹∀ V∈Use wfp (sourcenode a). state-val s V = state-val s' V›
        ‹(Main, n) = sourcenode a›[THEN sym]
      have rhs:∀ V∈rhs (label prog l). state-val s V = state-val s' V
        and PDef:∀ V∈set (ParamDefs wfp (sourcenode a)).
        state-val s V = state-val s' V
        by(auto simp:Use-def)
      from rhs ‹prog ⊢ n −IEdge (kind a)→p n'› ‹n = Label l› ‹CFG.pred (kind
a) s›
        ‹CFG.pred (kind a) s'›
      have lhs:∀ V∈lhs (label prog l).
        state-val (CFG.transfer (lift-procs wfp) (kind a) s) V =
        state-val (CFG.transfer (lift-procs wfp) (kind a) s') V
        by −(rule Proc-CFG-edge-uses-only-rhs,auto)
       from PDef ‹prog ⊢ n −IEdge (kind a)→p n'› ‹(Main, n) = sourcenode
a›[THEN sym]
      have ∀ V∈set (ParamDefs wfp (sourcenode a)).
        state-val (CFG.transfer (lift-procs wfp) (kind a) s) V =
        state-val (CFG.transfer (lift-procs wfp) (kind a) s') V
        by(fastforce dest:Proc-CFG-Call-follows-id-edge
        simp:ParamDefs-def ParamDefs-proc-def transfers-simps[of wfp,simplified]
          split:split-if-asm)
```

```
      with lhs ‹(Main, n) = sourcenode a›[THEN sym] Label show ?thesis
        by(fastforce simp:Def-def)
    next
      case Entry
      with ‹(Main, n) = sourcenode a›[THEN sym]
      show ?thesis by(fastforce simp:Entry-Def-empty)
    next
      case Exit
      with ‹prog ⊢ n −IEdge (kind a)→p n'› have False by fastforce
      thus ?thesis by simp
    qed
  next
    case (Proc p ins outs c n n')
    show ?case
    proof(cases n)
      case (Label l)
      with ‹∀ V∈Use wfp (sourcenode a). state-val s V = state-val s' V› wf
        ‹(p, n) = sourcenode a›[THEN sym] ‹(p, ins, outs, c) ∈ set procs›
      have rhs:∀ V∈rhs (label c l). state-val s V = state-val s' V
        and PDef:∀ V∈set (ParamDefs wfp (sourcenode a)).
        state-val s V = state-val s' V
        by(auto dest:in-procs-THE-in-procs-cmd simp:Use-def split:split-if-asm)
      from rhs ‹c ⊢ n −IEdge (kind a)→p n'› ‹n = Label l› ‹CFG.pred (kind a)
s›
        ‹CFG.pred (kind a) s'›
      have lhs:∀ V∈lhs (label c l).
        state-val (CFG.transfer (lift-procs wfp) (kind a) s) V =
        state-val (CFG.transfer (lift-procs wfp) (kind a) s') V
        by −(rule Proc-CFG-edge-uses-only-rhs,auto)
      from ‹(p, ins, outs, c) ∈ set procs› wf have [simp]:p ≠ Main by fastforce
      from wf ‹(p, ins, outs, c) ∈ set procs›
      have THE:(THE c'. ∃ins' outs'. (p,ins',outs',c') ∈ set procs) = c
        by(fastforce intro:in-procs-THE-in-procs-cmd)
      with PDef ‹c ⊢ n −IEdge (kind a)→p n'› ‹(p, n) = sourcenode a›[THEN
sym]
      have ∀ V∈set (ParamDefs wfp (sourcenode a)).
        state-val (CFG.transfer (lift-procs wfp) (kind a) s) V =
        state-val (CFG.transfer (lift-procs wfp) (kind a) s') V
        by(fastforce dest:Proc-CFG-Call-follows-id-edge
        simp:ParamDefs-def ParamDefs-proc-def transfers-simps[of wfp,simplified]
          split:split-if-asm)
      with lhs ‹(p, n) = sourcenode a›[THEN sym] Label THE
      show ?thesis by(auto simp:Def-def)
    next
      case Entry
      with wf ‹(p, ins, outs, c) ∈ set procs› have ParamDefs wfp (p,n) = []
        by(fastforce simp:ParamDefs-def ParamDefs-proc-def)
      moreover
      from Entry ‹c ⊢ n −IEdge (kind a)→p n'› ‹(p, ins, outs, c) ∈ set procs›
```

```
    have ParamUses wfp (p,n) = [] by(fastforce intro:ParamUses-Proc-IEdge-Nil)
      ultimately have ∀ V∈set ins. state-val s V = state-val s' V
        using wf ‹(p, ins, outs, c) ∈ set procs› ‹(p,n) = sourcenode a›
        ‹∀ V∈Use wfp (sourcenode a). state-val s V = state-val s' V› Entry
       by(fastforce dest:in-procs-THE-in-procs-ins simp:Use-def split:split-if-asm)
      with ‹c ⊢ n −IEdge (kind a)→p n'› Entry
      have ∀ V∈set ins. state-val (CFG.transfer (lift-procs wfp) (kind a) s) V =
        state-val (CFG.transfer (lift-procs wfp) (kind a) s') V
       by(fastforce dest:Proc-CFG-EntryD simp:transfers-simps[of wfp,simplified])
      with ‹(p,n) = sourcenode a›[THEN sym] Entry wf
        ‹(p, ins, outs, c) ∈ set procs› ‹ParamDefs wfp (p,n) = []›
      show ?thesis by(auto dest:in-procs-THE-in-procs-ins simp:Def-def)
    next
      case Exit
      with ‹c ⊢ n −IEdge (kind a)→p n'› have False by fastforce
      thus ?thesis by simp
    qed
  qed(auto simp:intra-kind-def)
next
  fix a s fix s'::((char list ⇀ val) × node) list
  assume valid-edge wfp a and CFG.pred (kind a) s
    and ∀ V∈Use wfp (sourcenode a). state-val s V = state-val s' V
    and length s = length s' and snd (hd s) = snd (hd s')
  hence prog,procs ⊢ sourcenode a −kind a→ targetnode a
    by(simp add:valid-edge-def)
  from ‹CFG.pred (kind a) s› obtain cf cfs where [simp]:s = cf#cfs
    by(cases s) auto
  from ‹length s = length s'› obtain cf' cfs' where [simp]:s' = cf'#cfs'
    by(cases s') auto
  from ‹prog,procs ⊢ sourcenode a −kind a→ targetnode a› ‹CFG.pred (kind a)
s›
    ‹∀ V∈Use wfp (sourcenode a). state-val s V = state-val s' V›
    ‹length s = length s'› ‹snd (hd s) = snd (hd s')›
  show CFG.pred (kind a) s'
  proof(induct sourcenode a kind a targetnode a rule:PCFG.induct)
    case (Main n n')
    show ?case
    proof(cases n)
      case (Label l)
      with ‹∀ V∈Use wfp (sourcenode a). state-val s V = state-val s' V›
        ‹(Main, n) = sourcenode a›
      have ∀ V∈rhs (label prog l). state-val s V = state-val s' V
        by(fastforce simp:Use-def)
      with ‹prog ⊢ n −IEdge (kind a)→p n'› Label ‹CFG.pred (kind a) s›
        ‹length s = length s'›
      show ?thesis by(fastforce intro:Proc-CFG-edge-rhs-pred-eq)
    next
      case Entry
      with ‹prog ⊢ n −IEdge (kind a)→p n'› ‹CFG.pred (kind a) s›
```

```
      show ?thesis by(fastforce dest:Proc-CFG-EntryD)
    next
      case Exit
      with ‹prog ⊢ n −IEdge (kind a)→p n'› have False by fastforce
      thus ?thesis by simp
    qed
  next
    case (Proc p ins outs c n n')
    show ?case
    proof(cases n)
      case (Label l)
      with ‹∀ V∈Use wfp (sourcenode a). state-val s V = state-val s' V› wf
        ‹(p, n) = sourcenode a›[THEN sym] ‹(p, ins, outs, c) ∈ set procs›
      have ∀ V∈rhs (label c l). state-val s V = state-val s' V
        by(auto dest:in-procs-THE-in-procs-cmd simp:Use-def split:split-if-asm)
      with ‹c ⊢ n −IEdge (kind a)→p n'› Label ‹CFG.pred (kind a) s›
        ‹length s = length s'›
      show ?thesis by(fastforce intro:Proc-CFG-edge-rhs-pred-eq)
    next
      case Entry
      with ‹c ⊢ n −IEdge (kind a)→p n'› ‹CFG.pred (kind a) s›
      show ?thesis by(fastforce dest:Proc-CFG-EntryD)
    next
      case Exit
      with ‹c ⊢ n −IEdge (kind a)→p n'› have False by fastforce
      thus ?thesis by simp
    qed
  next
    case (MainReturn l p es rets l' ins outs c)
    with ‹λcf. snd cf = (Main, Label l')↩pλcf cf'. cf'(rets [:=] map cf outs) =
      kind a›[THEN sym]
    show ?case by fastforce
  next
    case (ProcReturn p ins outs c l p' es rets l' ins' outs' c')
    with ‹λcf. snd cf = (p, Label l')↩p'λcf cf'. cf'(rets [:=] map cf outs') =
      kind a›[THEN sym]
    show ?case by fastforce
  qed(auto dest:sym)
next
  fix a Q r p fs ins outs
  assume valid-edge wfp a and kind a = Q:r↪pfs
    and (p, ins, outs) ∈ set (lift-procs wfp)
  hence prog,procs ⊢ sourcenode a −kind a→ targetnode a
    by(simp add:valid-edge-def)
  from this ‹kind a = Q:r↪pfs› ‹(p, ins, outs) ∈ set (lift-procs wfp)›
  show length fs = length ins
  proof(induct rule:PCFG.induct)
    case (MainCall l p' es rets n' ins' outs' c)
    hence fs = map interpret es and p' = p by simp-all
```

with *wf* ‹(*p*, *ins*, *outs*) ∈ *set* (*lift-procs wfp*)›
‹(*p'*, *ins'*, *outs'*, *c*) ∈ *set procs*›
have [*simp*]:*ins'* = *ins* **by** *fastforce*
from ‹*prog* ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *n'*›
have *containsCall procs prog* [] *p'* **by**(*rule Proc-CFG-Call-containsCall*)
with ‹*wf prog procs*› ‹(*p'*, *ins'*, *outs'*, *c*) ∈ *set procs*›
‹*prog* ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *n'*›
have *length es* = *length ins* **by** *fastforce*
with ‹*fs* = *map interpret es*› **show** *?case* **by** *simp*
next
case (*ProcCall px insx outsx c l p' es' rets' l' ins' outs' c' ps*)
hence *fs* = *map interpret es'* **and** *p'* = *p* **by** *simp-all*
with *wf* ‹(*p*, *ins*, *outs*) ∈ *set* (*lift-procs wfp*)›
‹(*p'*, *ins'*, *outs'*, *c'*) ∈ *set procs*›
have [*simp*]:*ins'* = *ins* **by** *fastforce*
from ‹*c* ⊢ *Label l* −*CEdge* (*p'*, *es'*, *rets'*)→$_p$ *Label l'*›
have *containsCall procs c* [] *p'* **by**(*rule Proc-CFG-Call-containsCall*)
with ‹*containsCall procs prog ps px*› ‹(*px*, *insx*, *outsx*, *c*) ∈ *set procs*›
have *containsCall procs prog* (*ps*@[*px*]) *p'* **by**(*rule containsCall-in-proc*)
with ‹*wf prog procs*› ‹(*p'*, *ins'*, *outs'*, *c'*) ∈ *set procs*›
‹*c* ⊢ *Label l* −*CEdge* (*p'*, *es'*, *rets'*)→$_p$ *Label l'*›
have *length es'* = *length ins* **by** *fastforce*
with ‹*fs* = *map interpret es'*› **show** *?case* **by** *simp*
qed *auto*
next
fix *a Q r p fs a' Q' r' p' fs' s s'*
assume *valid-edge wfp a* **and** *kind a* = *Q*:*r*↪$_p$*fs*
and *valid-edge wfp a'* **and** *kind a'* = *Q'*:*r'*↪$_{p'}$*fs'*
and *sourcenode a* = *sourcenode a'*
hence *prog*,*procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*
and *prog*,*procs* ⊢ *sourcenode a'* −*kind a'*→ *targetnode a'*
by(*simp-all add*:*valid-edge-def*)
from *this* ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*kind a'* = *Q'*:*r'*↪$_{p'}$*fs'*› **show** *a* = *a'*
proof(*induct sourcenode a kind a targetnode a rule*:*PCFG.induct*)
case (*MainCall l px es rets n' insx outsx cx*)
from ‹*prog*,*procs* ⊢ *sourcenode a'* −*kind a'*→ *targetnode a'*›
‹*kind a'* = *Q'*:*r'*↪$_{p'}$*fs'*›
‹(*Main*, *Label l*) = *sourcenode a*› ‹*sourcenode a* = *sourcenode a'*›
‹*prog* ⊢ *Label l* −*CEdge* (*px*, *es*, *rets*)→$_p$ *n'*› *wf*
have *targetnode a'* = (*px*, *Entry*)
by(*fastforce elim*!:*PCFG.cases dest*:*Proc-CFG-Call-nodes-eq*)
with ‹*valid-edge wfp a*› ‹*valid-edge wfp a'*›
‹*sourcenode a* = *sourcenode a'*› ‹(*px*, *Entry*) = *targetnode a*› *wf*
have *kind a* = *kind a'* **by**(*fastforce intro*:*Proc-CFG-edge-det simp*:*valid-edge-def*)
with ‹*sourcenode a* = *sourcenode a'*› ‹(*px*, *Entry*) = *targetnode a*›
‹*targetnode a'* = (*px*, *Entry*)›
show *?case* **by**(*cases a*,*cases a'*,*auto*)
next
case (*ProcCall px ins outs c l px' es rets l' insx outsx cx*)

with *wf* **have** $px \neq Main$ **by** *fastforce*
with ‹*prog*,*procs* ⊢ *sourcenode a'* −*kind a'*→ *targetnode a'*›
‹*kind a'* = *Q'*:*r'*↪$_p$*'fs'*›
‹(*px*, *Label l*) = *sourcenode a*› ‹*sourcenode a* = *sourcenode a'*›
‹*c* ⊢ *Label l* −*CEdge* (*px'*, *es*, *rets*)→$_p$ *Label l'*›
‹(*px'*, *insx*, *outsx*, *cx*) ∈ *set procs*› ‹(*px*, *ins*, *outs*, *c*) ∈ *set procs*›
have *targetnode a'* = (*px'*, *Entry*)
proof(*induct n*≡*sourcenode a' et*≡*kind a' n'*≡*targetnode a' rule*:*PCFG.induct*)
case (*ProcCall p insa outsa ca la p'a es' rets' l'a ins' outs' c'*)
hence [*simp*]:*px* = *p l* = *la* **by**(*auto dest*:*sym*)
from ‹(*p*, *insa*, *outsa*, *ca*) ∈ *set procs*›
‹(*px*, *ins*, *outs*, *c*) ∈ *set procs*› *wf* **have** [*simp*]:*ca* = *c* **by** *auto*
from ‹*ca* ⊢ *Label la* −*CEdge* (*p'a*, *es'*, *rets'*)→$_p$ *Label l'a*›
‹*c* ⊢ *Label l* −*CEdge* (*px'*, *es*, *rets*)→$_p$ *Label l'*›
have *p'a* = *px'* **by**(*fastforce dest*:*Proc-CFG-Call-nodes-eq*)
with ‹(*p'a*, *Entry*) = *targetnode a'*› **show** *?case* **by** *simp*
qed(*auto dest*:*sym*)
with ‹*valid-edge wfp a*› ‹*valid-edge wfp a'*›
‹*sourcenode a* = *sourcenode a'*› ‹(*px'*, *Entry*) = *targetnode a*› *wf*
have *kind a* = *kind a'* **by**(*fastforce intro*:*Proc-CFG-edge-det simp*:*valid-edge-def*)
with ‹*sourcenode a* = *sourcenode a'*› ‹(*px'*, *Entry*) = *targetnode a*›
‹*targetnode a'* = (*px'*, *Entry*)› **show** *?case* **by**(*cases a*,*cases a'*,*auto*)
qed *auto*
next
fix *a Q r p fs i ins outs* **fix** *s s'*::((*char list* ⇀ *val*) × *node*) *list*
assume *valid-edge wfp a* **and** *kind a* = *Q*:*r*↪$_p$*fs* **and** *i* < *length ins*
and (*p*, *ins*, *outs*) ∈ *set* (*lift-procs wfp*)
and ∀ *V*∈*ParamUses wfp* (*sourcenode a*) ! *i*. *state-val s V* = *state-val s' V*
hence *prog*,*procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*
by(*simp add*:*valid-edge-def*)
from *this* ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹*i* < *length ins*›
‹(*p*, *ins*, *outs*) ∈ *set* (*lift-procs wfp*)›
‹∀ *V*∈*ParamUses wfp* (*sourcenode a*) ! *i*. *state-val s V* = *state-val s' V*›
show *CFG.params fs* (*state-val s*) ! *i* = *CFG.params fs* (*state-val s'*) ! *i*
proof(*induct sourcenode a kind a targetnode a rule*:*PCFG.induct*)
case (*MainCall l p' es rets n' insx outsx cx*)
with *wf* **have** [*simp*]:*insx* = *ins fs* = *map interpret es* **by** *auto*
from ‹*prog* ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *n'*›
have *containsCall procs prog* [] *p'* **by**(*rule Proc-CFG-Call-containsCall*)
with ‹*wf prog procs*› ‹(*p'*, *insx*, *outsx*, *cx*) ∈ *set procs*›
‹*prog* ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *n'*›
have *length es* = *length ins* **by** *fastforce*
with ‹*i* < *length ins*› **have** *i* < *length* (*map interpret es*) **by** *simp*
from ‹*prog* ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *n'*›
have *ParamUses wfp* (*Main*,*Label l*) = *map fv es*
by(*fastforce intro*:*ParamUses-Main-Return-target*)
with ‹∀ *V*∈*ParamUses wfp* (*sourcenode a*) ! *i*. *state-val s V* = *state-val s' V*›
‹*i* < *length* (*map interpret es*)› ‹(*Main*, *Label l*) = *sourcenode a*›
have ((*map* (λ*e cf*. *interpret e cf*) *es*)!*i*) (*fst* (*hd s*)) =

```
      ((map (λe cf. interpret e cf) es)!i) (fst (hd s'))
      by(cases interpret (es ! i) (fst (hd s)))(auto dest:rhs-interpret-eq)
    with ‹i < length (map interpret es)› show ?case by(simp add:ProcCFG.params-nth)
    next
     case (ProcCall px insx outsx cx l p' es' rets' l' ins' outs' c' ps)
     with wf have [simp]:ins' = ins by fastforce
     from ‹cx ⊢ Label l −CEdge (p', es', rets')→p Label l'›
     have containsCall procs cx [] p' by(rule Proc-CFG-Call-containsCall)
     with ‹containsCall procs prog ps px› ‹(px, insx, outsx, cx) ∈ set procs›
     have containsCall procs prog (ps@[px]) p' by(rule containsCall-in-proc)
     with ‹wf prog procs› ‹(p', ins', outs', c') ∈ set procs›
       ‹cx ⊢ Label l −CEdge (p', es', rets')→p Label l'›
     have length es' = length ins by fastforce
      from ‹λs. True:(px, Label l')↪p'map interpret es' = kind a› ‹kind a =
Q:r↪pfs›
     have fs = map interpret es' by simp-all
     from ‹i < length ins› ‹fs = map interpret es'›
       ‹length es' = length ins› have i < length fs by simp
     from ‹(px, insx, outsx, cx) ∈ set procs›
       ‹cx ⊢ Label l −CEdge (p', es', rets')→p Label l'›
     have ParamUses wfp (px,Label l) = map fv es'
       by(auto intro!:ParamUses-Proc-Return-target simp:set-conv-nth)
     with ‹∀ V∈ParamUses wfp (sourcenode a) ! i. state-val s V = state-val s' V›
       ‹(px, Label l) = sourcenode a› ‹i < length fs›
       ‹fs = map interpret es'›
     have ((map (λe cf. interpret e cf) es')!i) (fst (hd s)) =
       ((map (λe cf. interpret e cf) es')!i) (fst (hd s'))
       by(cases interpret (es' ! i) (fst (hd s)))(auto dest:rhs-interpret-eq)
     with ‹i < length fs› ‹fs = map interpret es'›
     show ?case by(simp add:ProcCFG.params-nth)
    qed auto
  next
    fix a Q' p f' ins outs cf cf'
    assume valid-edge wfp a and kind a = Q'↩pf'
      and (p, ins, outs) ∈ set (lift-procs wfp)
    thus f' cf cf' = cf'(ParamDefs wfp (targetnode a) [:=] map cf outs)
      by(rule Return-update)
  next
    fix a a'
    assume valid-edge wfp a and valid-edge wfp a'
      and sourcenode a = sourcenode a' and targetnode a ≠ targetnode a'
      and intra-kind (kind a) and intra-kind (kind a')
    with wf show ∃ Q Q'. kind a = (Q)√ ∧ kind a' = (Q')√ ∧
      (∀ cf. (Q cf ⟶ ¬ Q' cf) ∧ (Q' cf ⟶ ¬ Q cf))
      by(auto dest:Proc-CFG-deterministic simp:valid-edge-def)
  qed
qed
```

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
from *Exit-Def-empty Exit-Use-empty*
show *Def wfp* (*Main*, *Exit*) = {} ∧ *Use wfp* (*Main*, *Exit*) = {} **by** *simp*
qed

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 ⊢ *Label l* −*et*→$_p$ *Label l'* ⟹ *prog*;;c_2 ⊢ *Label l* −*et*→$_p$ *Label l'*
proof(*induct prog Label l et Label l' rule*:*Proc-CFG.induct*)
case (*Proc-CFG-SeqSecond* c_2' *n et n'* c_1)
hence (c_1;; c_2');; c_2 ⊢ *n* ⊕ #:c_1 −*et*→$_p$ *n'* ⊕ #:c_1
by(*fastforce intro*:*Proc-CFG-SeqFirst Proc-CFG.Proc-CFG-SeqSecond*)
with ‹*n* ⊕ #:c_1 = *Label l*› ‹*n'* ⊕ #:c_1 = *Label l'*› **show** *?case* **by** *fastforce*
next
case (*Proc-CFG-CondThen* c_1 *n et n'* *b* c_2')
hence *if* (*b*) c_1 *else* c_2';; c_2 ⊢ *n* ⊕ *1* −*et*→$_p$ *n'* ⊕ *1*
by(*fastforce intro*:*Proc-CFG-SeqFirst Proc-CFG.Proc-CFG-CondThen*)
with ‹*n* ⊕ *1* = *Label l*› ‹*n'* ⊕ *1* = *Label l'*› **show** *?case* **by** *fastforce*
next
case (*Proc-CFG-CondElse* c_1 *n et n'* *b* c_2')

hence *if* (*b*) c_2' *else* c_1 ;; $c_2 \vdash n \oplus \#{:}c_2' + 1 \ -et\rightarrow_p\ n' \oplus (\#{:}c_2' + 1)$
by(*fastforce intro*:*Proc-CFG-SeqFirst Proc-CFG.Proc-CFG-CondElse*)
with ‹$n \oplus \#{:}c_2' + 1 = Label\ l$› ‹$n' \oplus \#{:}c_2' + 1 = Label\ l'$› **show** *?case* **by** *fastforce*
next
case (*Proc-CFG-WhileBody* c' *n et* n' *b*)
hence *while* (*b*) c';; $c_2 \vdash n \oplus 2 \ -et\rightarrow_p\ n' \oplus 2$
by(*fastforce intro*:*Proc-CFG-SeqFirst Proc-CFG.Proc-CFG-WhileBody*)
with ‹$n \oplus 2 = Label\ l$› ‹$n' \oplus 2 = Label\ l'$› **show** *?case* **by** *fastforce*
next
case (*Proc-CFG-WhileBodyExit* c' *n et b*)
hence *while* (*b*) c';; $c_2 \vdash n \oplus 2 \ -et\rightarrow_p\ Label\ 0$
by(*fastforce intro*:*Proc-CFG-SeqFirst Proc-CFG.Proc-CFG-WhileBodyExit*)
with ‹$n \oplus 2 = Label\ l$› ‹$0 = l'$› **show** *?case* **by** *fastforce*
qed (*auto intro*:*Proc-CFG.intros*)

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(*atomize-elim*)
from ‹*prog* $\vdash Label\ l \ -et\rightarrow_p\ n'$› **obtain** *n* **where** *prog* $\vdash n \ -et\rightarrow_p\ n'$ **and** $Label\ l = n$
by *simp*
thus $\exists nx.$ *prog*;;$c_2 \vdash Label\ l \ -et\rightarrow_p\ nx$
proof(*induct prog n et* n' *rule*:*Proc-CFG.induct*)
case (*Proc-CFG-SeqSecond* c_2' *n et* n' c_1)
show *?case*
proof(*cases* $n' = Exit$)
case *True*
with ‹$c_2' \vdash n \ -et\rightarrow_p\ n'$› ‹$n \neq Entry$› **have** c_1;; $c_2' \vdash n \oplus \#{:}c_1 \ -et\rightarrow_p\ Exit \oplus \#{:}c_1$
by(*fastforce intro*:*Proc-CFG.Proc-CFG-SeqSecond*)
moreover from ‹$n \neq Entry$› **have** $n \oplus \#{:}c_1 \neq Entry$ **by**(*cases n*) *auto*
ultimately
have c_1;; c_2';; $c_2 \vdash n \oplus \#{:}c_1 \ -et\rightarrow_p\ Label\ (\#{:}c_1;;\ c_2')$
by(*fastforce intro*:*Proc-CFG-SeqConnect*)
with ‹$Label\ l = n \oplus \#{:}c_1$› **show** *?thesis* **by** *fastforce*
next
case *False*
with *Proc-CFG-SeqSecond*
have (c_1;; c_2');; $c_2 \vdash n \oplus \#{:}c_1 \ -et\rightarrow_p\ n' \oplus \#{:}c_1$
by(*fastforce intro*:*Proc-CFG-SeqFirst Proc-CFG.Proc-CFG-SeqSecond*)
with ‹$Label\ l = n \oplus \#{:}c_1$› **show** *?thesis* **by** *fastforce*
qed
next
case (*Proc-CFG-CondThen* c_1 *n et* n' *b* c_2')
show *?case*
proof(*cases* $n' = Exit$)

```
      case True
      with ‹c1 ⊢ n −et→p n'› ‹n ≠ Entry›
      have if (b) c1 else c2' ⊢ n ⊕ 1 −et→p Exit ⊕ 1
        by(fastforce intro:Proc-CFG.Proc-CFG-CondThen)
      moreover from ‹n ≠ Entry› have n ⊕ 1 ≠ Entry by(cases n) auto
      ultimately
      have if (b) c1 else c2';; c2 ⊢ n ⊕ 1 −et→p Label (#:if (b) c1 else c2')
        by(fastforce intro:Proc-CFG-SeqConnect)
      with ‹Label l = n ⊕ 1› show ?thesis by fastforce
    next
      case False
      hence n' ⊕ 1 ≠ Exit by(cases n') auto
      with Proc-CFG-CondThen
      have  if (b) c1 else c2';; c2 ⊢ Label l −et→p n' ⊕ 1
        by(fastforce intro:Proc-CFG-SeqFirst Proc-CFG.Proc-CFG-CondThen)
      with ‹Label l = n ⊕ 1› show ?thesis by fastforce
    qed
  next
    case (Proc-CFG-CondElse c1 n et n' b c2')
    show ?case
    proof(cases n' = Exit)
      case True
      with ‹c1 ⊢ n −et→p n'› ‹n ≠ Entry›
      have if (b) c2' else c1 ⊢ n ⊕ (#:c2' + 1) −et→p Exit ⊕ (#:c2' + 1)
        by(fastforce intro:Proc-CFG.Proc-CFG-CondElse)
       moreover from ‹n ≠ Entry› have n ⊕ (#:c2' + 1) ≠ Entry by(cases n)
auto
      ultimately
      have if (b) c2' else c1;; c2 ⊢ n ⊕ (#:c2' + 1) −et→p
        Label (#:if (b) c2' else c1)
        by(fastforce intro:Proc-CFG-SeqConnect)
      with ‹Label l = n ⊕ (#:c2' + 1)› show ?thesis by fastforce
    next
      case False
      hence n' ⊕ (#:c2' + 1) ≠ Exit by(cases n') auto
      with Proc-CFG-CondElse
      have  if (b) c2' else c1 ;; c2 ⊢ Label l −et→p n' ⊕ (#:c2' + 1)
        by(fastforce intro:Proc-CFG-SeqFirst Proc-CFG.Proc-CFG-CondElse)
      with ‹Label l = n ⊕ (#:c2' + 1)› show ?thesis by fastforce
    qed
  qed (auto intro:Proc-CFG.intros)
qed


lemma Proc-CFG-edge-SeqFirst-target-Label:
  ⟦prog ⊢ n −et→p n'; Label l' = n'⟧ ⟹ prog;;c2 ⊢ n −et→p Label l'
proof(induct prog n et n' rule:Proc-CFG.induct)
  case (Proc-CFG-SeqSecond c2' n et n' c1)
  from ‹Label l' = n' ⊕ #:c1› have n' ≠ Exit by(cases n') auto
```

with *Proc-CFG-SeqSecond*
show *?case* **by**(*fastforce intro*:*Proc-CFG-SeqFirst intro*:*Proc-CFG.Proc-CFG-SeqSecond*)
next
case (*Proc-CFG-CondThen* c_1 *n et n' b* c_2')
from ‹*Label l'* = *n'* ⊕ *1*› **have** *n'* ≠ *Exit* **by**(*cases n'*) *auto*
with *Proc-CFG-CondThen*
show *?case* **by**(*fastforce intro*:*Proc-CFG-SeqFirst Proc-CFG.Proc-CFG-CondThen*)
qed (*auto intro*:*Proc-CFG.intros*)

lemma *PCFG-edge-SeqFirst-source-Label*:
assumes *prog*,*procs* ⊢ (*p*,*Label l*) −*et*→ (*p'*,*n'*)
obtains *nx* **where** *prog*;;c_2,*procs* ⊢ (*p*,*Label l*) −*et*→ (*p'*,*nx*)
proof(*atomize-elim*)
from ‹*prog*,*procs* ⊢ (*p*,*Label l*) −*et*→ (*p'*,*n'*)›
show ∃ *nx*. *prog*;;c_2,*procs* ⊢ (*p*,*Label l*) −*et*→ (*p'*,*nx*)
proof(*induct* (*p*,*Label l*) *et* (*p'*,*n'*) *rule*:*PCFG.induct*)
case (*Main et*)
from ‹*prog* ⊢ *Label l* −*IEdge et*→$_p$ *n'*›
obtain *nx'* **where** *prog*;;c_2 ⊢ *Label l* −*IEdge et*→$_p$ *nx'*
by(*auto elim*:*Proc-CFG-edge-SeqFirst-source-Label*)
with ‹*Main* = *p*› ‹*Main* = *p'*› **show** *?case*
by(*fastforce dest*:*PCFG.Main*)
next
case (*Proc ins outs c et ps*)
from ‹*containsCall procs prog ps p*›
have *containsCall procs* (*prog*;;c_2) *ps p* **by** *simp*
with *Proc* **show** *?case* **by**(*fastforce dest*:*PCFG.Proc*)
next
case (*MainCall es rets nx ins outs c*)
from ‹*prog* ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *nx*›
obtain *lx* **where** [*simp*]:*nx* = *Label lx* **by**(*fastforce dest*:*Proc-CFG-Call-Labels*)
with ‹*prog* ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *nx*›
have *prog*;;c_2 ⊢ *Label l* −*CEdge* (*p'*, *es*, *rets*)→$_p$ *Label lx*
by(*auto intro*:*Proc-CFG-edge-SeqFirst-nodes-Label*)
with *MainCall* **show** *?case* **by**(*fastforce dest*:*PCFG.MainCall*)
next
case (*ProcCall ins outs c es' rets' l' ins' outs' c' ps*)
from ‹*containsCall procs prog ps p*›
have *containsCall procs* (*prog*;;c_2) *ps p* **by** *simp*
with *ProcCall* **show** *?case* **by**(*fastforce intro*:*PCFG.ProcCall*)
next
case (*MainCallReturn px es rets*)
from ‹*prog* ⊢ *Label l* −*CEdge* (*px*, *es*, *rets*)→$_p$ *n'*› ‹*Main* = *p*›
obtain *nx''* **where** *prog*;;c_2 ⊢ *Label l* −*CEdge* (*px*, *es*, *rets*)→$_p$ *nx''*
by(*auto elim*:*Proc-CFG-edge-SeqFirst-source-Label*)
with *MainCallReturn* **show** *?case* **by**(*fastforce dest*:*PCFG.MainCallReturn*)
next
case (*ProcCallReturn ins outs c px' es' rets' ps*)

from ‹*containsCall procs prog ps p*›
have *containsCall procs* (*prog*;;c_2) *ps p* **by** *simp*
with *ProcCallReturn* **show** *?case* **by**(*fastforce dest*!:*PCFG.ProcCallReturn*)
qed
qed

lemma *PCFG-edge-SeqFirst-target-Label*:
prog,*procs* ⊢ (*p*,*n*) −*et*→ (*p'*,*Label l'*)
⟹ *prog*;;c_2,*procs* ⊢ (*p*,*n*) −*et*→ (*p'*,*Label l'*)
proof(*induct* (*p*,*n*) *et* (*p'*,*Label l'*) *rule*:*PCFG.induct*)
case *Main*
thus *?case* **by**(*fastforce dest*:*Proc-CFG-edge-SeqFirst-target-Label intro*:*PCFG.Main*)
next
case (*Proc ins outs c et ps*)
from ‹*containsCall procs prog ps p*›
have *containsCall procs* (*prog*;;c_2) *ps p* **by** *simp*
with *Proc* **show** *?case* **by**(*fastforce dest*:*PCFG.Proc*)
next
case *MainReturn* **thus** *?case*
by(*fastforce dest*:*Proc-CFG-edge-SeqFirst-target-Label*
intro!:*PCFG.MainReturn*[*simplified*])
next
case (*ProcReturn ins outs c lx es' rets' ins' outs' c' ps*)
from ‹*containsCall procs prog ps p'*›
have *containsCall procs* (*prog*;;c_2) *ps p'* **by** *simp*
with *ProcReturn* **show** *?case* **by**(*fastforce intro*:*PCFG.ProcReturn*)
next
case *MainCallReturn* **thus** *?case*
by(*fastforce dest*:*Proc-CFG-edge-SeqFirst-target-Label intro*:*PCFG.MainCallReturn*)
next
case (*ProcCallReturn ins outs c px' es' rets' ps*)
from ‹*containsCall procs prog ps p*›
have *containsCall procs* (*prog*;;c_2) *ps p* **by** *simp*
with *ProcCallReturn* **show** *?case* **by**(*fastforce dest*!:*PCFG.ProcCallReturn*)
qed

lemma *path-SeqFirst*:
assumes *Rep-wf-prog wfp* = (*prog*,*procs*) **and** *Rep-wf-prog wfp'* = (*prog*;;c_2,*procs*)
shows ⟦*wfp* ⊢ (*p*,*n*) −*as*→∗ (*p*,*Label l*); ∀ *a* ∈ *set as*. *intra-kind* (*kind a*)⟧
⟹ *wfp'* ⊢ (*p*,*n*) −*as*→∗ (*p*,*Label l*)
proof(*induct* (*p*,*n*) *as* (*p*,*Label l*) *arbitrary*:*n rule*:*ProcCFG.path.induct*)
case *empty-path*
from ‹*CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*p*, *Label l*)›
‹*Rep-wf-prog wfp* = (*prog*, *procs*)› ‹*Rep-wf-prog wfp'* = (*prog*;; c_2, *procs*)›
have *CFG.valid-node sourcenode targetnode* (*valid-edge wfp'*) (*p*, *Label l*)
apply(*auto simp*:*ProcCFG.valid-node-def valid-edge-def*)
apply(*erule PCFG-edge-SeqFirst-source-Label*,*fastforce*)

by(*drule PCFG-edge-SeqFirst-target-Label*,*fastforce*)
thus *?case* **by**(*fastforce intro*:*ProcCFG.empty-path*)
next
case (*Cons-path n'' as a nx*)
note *IH* = ‹⋀*n*. ⟦*n''* = (*p*, *n*); ∀ *a*∈*set as*. *intra-kind* (*kind a*)⟧
⟹ *wfp'* ⊢ (*p*, *n*) −*as*→∗ (*p*, *Label l*)›
note [*simp*] = ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› ‹*Rep-wf-prog wfp'* = (*prog*;;*c*$_2$,*procs*)›
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *wf*:*well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
from ‹∀ *a*∈*set* (*a* # *as*). *intra-kind* (*kind a*)› **have** *intra-kind* (*kind a*)
and ∀ *a*∈*set as*. *intra-kind* (*kind a*) **by** *simp-all*
from ‹*valid-edge wfp a*› ‹*sourcenode a* = (*p*, *nx*)› ‹*targetnode a* = *n''*›
‹*intra-kind* (*kind a*)› *wf*
obtain *nx'* **where** *n''* = (*p*,*nx'*)
by(*auto elim*:*PCFG.cases simp*:*valid-edge-def intra-kind-def*)
from *IH*[*OF this* ‹∀ *a*∈*set as*. *intra-kind* (*kind a*)›]
have *path*:*wfp'* ⊢ (*p*, *nx'*) −*as*→∗ (*p*, *Label l*) **.**
have *valid-edge wfp' a*
proof(*cases nx'*)
case (*Label lx*)
with ‹*valid-edge wfp a*› ‹*sourcenode a* = (*p*, *nx*)› ‹*targetnode a* = *n''*›
‹*n''* = (*p*,*nx'*)› **show** *?thesis*
by(*fastforce intro*:*PCFG-edge-SeqFirst-target-Label*
simp:*intra-kind-def valid-edge-def*)
next
case *Entry*
with ‹*valid-edge wfp a*› ‹*targetnode a* = *n''*› ‹*n''* = (*p*,*nx'*)›
‹*intra-kind* (*kind a*)› **have** *False*
by(*auto elim*:*PCFG.cases simp*:*valid-edge-def intra-kind-def*)
thus *?thesis* **by** *simp*
next
case *Exit*
with *path* ‹∀ *a*∈*set as*. *intra-kind* (*kind a*)› **have** *False*
by(*induct* (*p*,*nx'*) *as* (*p*,*Label l*) *rule*:*ProcCFG.path.induct*)
(*auto elim*!:*PCFG.cases simp*:*valid-edge-def intra-kind-def*)
thus *?thesis* **by** *simp*
qed
with ‹*sourcenode a* = (*p*, *nx*)› ‹*targetnode a* = *n''*› ‹*n''* = (*p*,*nx'*)› *path*
show *?case* **by**(*fastforce intro*:*ProcCFG.Cons-path*)
qed

From *prog* to *c*$_1$;;*prog*

lemma *Proc-CFG-edge-SeqSecond-source-not-Entry*:
⟦*prog* ⊢ *n* −*et*→$_p$ *n'*; *n* ≠ *Entry*⟧ ⟹ *c*$_1$;;*prog* ⊢ *n* ⊕ #:*c*$_1$ −*et*→$_p$ *n'* ⊕ #:*c*$_1$
by(*induct rule*:*Proc-CFG.induct*)(*fastforce intro*:*Proc-CFG-SeqSecond Proc-CFG.intros*)+

lemma *PCFG-Main-edge-SeqSecond-source-not-Entry*:

```
  ⟦prog,procs ⊢ (Main,n) −et→ (p′,n′); n ≠ Entry; intra-kind et; well-formed procs⟧
  ⟹ c1;;prog,procs ⊢ (Main,n ⊕ #:c1) −et→ (p′,n′ ⊕ #:c1)
proof(induct (Main,n) et (p′,n′) rule:PCFG.induct)
  case Main
  thus ?case
   by(fastforce dest:Proc-CFG-edge-SeqSecond-source-not-Entry intro:PCFG.Main)
next
  case (MainCallReturn p es rets)
  from ‹prog ⊢ n −CEdge (p, es, rets)→p n′› ‹n ≠ Entry›
  have c1;;prog ⊢ n ⊕ #:c1 −CEdge (p, es, rets)→p n′ ⊕ #:c1
    by(rule Proc-CFG-edge-SeqSecond-source-not-Entry)
  with MainCallReturn show ?case by(fastforce intro:PCFG.MainCallReturn)
qed (auto simp:intra-kind-def)


lemma valid-node-Main-SeqSecond:
  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′ = (c1;;prog,procs)
  shows CFG.valid-node sourcenode targetnode (valid-edge wfp′) (Main, n ⊕ #:c1)
proof −
  note [simp] = ‹Rep-wf-prog wfp = (prog,procs)› ‹Rep-wf-prog wfp′ = (c1;;prog,procs)›
  from ‹Rep-wf-prog wfp = (prog,procs)› have wf:well-formed procs
    by(fastforce intro:wf-wf-prog)
  from ‹CFG.valid-node sourcenode targetnode (valid-edge wfp) (Main,n)›
  obtain a where prog,procs ⊢ sourcenode a −kind a→ targetnode a
    and (Main,n) = sourcenode a ∨ (Main,n) = targetnode a
    by(fastforce simp:ProcCFG.valid-node-def valid-edge-def)
  from this ‹n ≠ Entry› wf show ?thesis
  proof(induct sourcenode a kind a targetnode a rule:PCFG.induct)
    case (Main nx nx′)
    from ‹(Main,n) = sourcenode a ∨ (Main,n) = targetnode a› show ?case
    proof
      assume (Main,n) = sourcenode a
      with ‹(Main, nx) = sourcenode a›[THEN sym] have [simp]:nx = n by simp
      from ‹n ≠ Entry› ‹prog ⊢ nx −IEdge (kind a)→p nx′›
      have c1;;prog ⊢ n ⊕ #:c1 −IEdge (kind a)→p nx′ ⊕ #:c1
        by(fastforce intro:Proc-CFG-edge-SeqSecond-source-not-Entry)
      hence c1;;prog,procs ⊢ (Main,n ⊕ #:c1) −kind a→ (Main,nx′ ⊕ #:c1)
        by(rule PCFG.Main)
      thus ?thesis by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
    next
      assume (Main, n) = targetnode a
      show ?thesis
      proof(cases nx = Entry)
        case True
        with ‹prog ⊢ nx −IEdge (kind a)→p nx′›
        have nx′ = Exit ∨ nx′ = Label 0 by(fastforce dest:Proc-CFG-EntryD)
```

```
      thus ?thesis
      proof
        assume nx' = Exit
       with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
        show ?thesis by simp
      next
        assume nx' = Label 0
        obtain l etx where c1 ⊢ Label l −IEdge etx→p Exit and l ≤ #:c1
          by(erule Proc-CFG-Exit-edge)
        hence c1;;prog ⊢ Label l −IEdge etx→p Label #:c1
          by(fastforce intro:Proc-CFG-SeqConnect)
        with ‹nx' = Label 0›
        have c1;;prog,procs ⊢ (Main,Label l) −etx→ (Main,nx'⊕#:c1)
          by(fastforce intro:PCFG.Main)
       with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
        show ?thesis
          by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
      qed
    next
      case False
      with ‹prog ⊢ nx −IEdge (kind a)→p nx'›
      have c1;;prog ⊢ nx ⊕ #:c1 −IEdge (kind a)→p nx' ⊕ #:c1
        by(fastforce intro:Proc-CFG-edge-SeqSecond-source-not-Entry)
      hence c1;;prog,procs ⊢ (Main,nx ⊕ #:c1) −kind a→ (Main,nx' ⊕ #:c1)
        by(rule PCFG.Main)
      with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
    show ?thesis by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
    qed
  qed
next
  case (Proc p ins outs c nx n' ps)
  from ‹(p, nx) = sourcenode a›[THEN sym] ‹(p, n') = targetnode a›[THEN sym]
    ‹(Main, n) = sourcenode a ∨ (Main, n) = targetnode a›
    ‹(p, ins, outs, c) ∈ set procs› ‹well-formed procs› have False by fastforce
  thus ?case by simp
next
  case (MainCall l p es rets n' ins outs c)
  from ‹(p, ins, outs, c) ∈ set procs› wf ‹(p, Entry) = targetnode a›[THEN sym]
    ‹(Main, Label l) = sourcenode a›[THEN sym]
    ‹(Main, n) = sourcenode a ∨ (Main, n) = targetnode a›
   have [simp]:n = Label l by fastforce
  from ‹prog ⊢ Label l −CEdge (p, es, rets)→p n'›
  have c1;;prog ⊢ Label l ⊕ #:c1 −CEdge (p, es, rets)→p n' ⊕ #:c1
    by −(rule Proc-CFG-edge-SeqSecond-source-not-Entry,auto)
  with ‹(p, ins, outs, c) ∈ set procs›
  have c1;;prog,procs ⊢ (Main,Label (l + #:c1))
    −(λs. True):(Main,n' ⊕ #:c1)↪pmap (λe cf. interpret e cf) es→ (p,Entry)
    by(fastforce intro:PCFG.MainCall)
 thus ?case by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
```

next
case (*ProcCall p ins outs c l p′ es′ rets′ l′ ins′ outs′ c′*)
from ‹(*p*, *Label l*) = *sourcenode a*›[*THEN sym*]
‹(*p′*, *Entry*) = *targetnode a*›[*THEN sym*] ‹*well-formed procs*›
‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹(*p′*, *ins′*, *outs′*, *c′*) ∈ *set procs*›
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have *False* **by** *fastforce*
thus *?case* **by** *simp*
next
case (*MainReturn l p es rets l′ ins outs c*)
from ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› *wf* ‹(*p*, *Exit*) = *sourcenode a*›[*THEN sym*]
‹(*Main*, *Label l′*) = *targetnode a*›[*THEN sym*]
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have [*simp*]:*n* = *Label l′* **by** *fastforce*
from ‹*prog* ⊢ *Label l* −*CEdge* (*p*, *es*, *rets*)→$_p$ *Label l′*›
have c_1;;*prog* ⊢ *Label l* ⊕ #:c_1 −*CEdge* (*p*, *es*, *rets*)→$_p$ *Label l′* ⊕ #:c_1
by −(*rule Proc-CFG-edge-SeqSecond-source-not-Entry*,*auto*)
with ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*›
have c_1;;*prog*,*procs* ⊢ (*p*,*Exit*) −(λ*cf*. *snd cf* = (*Main*,*Label l′* ⊕ #:c_1))↩$_p$
(λ*cf cf′*. *cf′*(*rets* [:=] *map cf outs*))→ (*Main*,*Label* (*l′* + #:c_1))
by(*fastforce intro*:*PCFG.MainReturn*)
thus *?case* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
next
case (*ProcReturn p ins outs c l p′ es′ rets′ l′ ins′ outs′ c′ ps*)
from ‹(*p′*, *Exit*) = *sourcenode a*›[*THEN sym*]
‹(*p*, *Label l′*) = *targetnode a*›[*THEN sym*] ‹*well-formed procs*›
‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹(*p′*, *ins′*, *outs′*, *c′*) ∈ *set procs*›
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have *False* **by** *fastforce*
thus *?case* **by** *simp*
next
case (*MainCallReturn nx p es rets nx′*)
from ‹(*Main*,*n*) = *sourcenode a* ∨ (*Main*,*n*) = *targetnode a*› **show** *?case*
proof
assume (*Main*,*n*) = *sourcenode a*
with ‹(*Main*, *nx*) = *sourcenode a*›[*THEN sym*] **have** [*simp*]:*nx* = *n* **by** *simp*
from ‹*n* ≠ *Entry*› ‹*prog* ⊢ *nx* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx′*›
have c_1;;*prog* ⊢ *n* ⊕ #:c_1 −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx′* ⊕ #:c_1
by(*fastforce intro*:*Proc-CFG-edge-SeqSecond-source-not-Entry*)
hence c_1;;*prog*,*procs* ⊢ (*Main*,*n* ⊕ #:c_1) −(λ*s*. *False*)$_{\surd}$→ (*Main*,*nx′* ⊕ #:c_1)
by −(*rule PCFG.MainCallReturn*)
thus *?thesis* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
next
assume (*Main*, *n*) = *targetnode a*
from ‹*prog* ⊢ *nx* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx′*›
have *nx* ≠ *Entry* **by**(*fastforce dest*:*Proc-CFG-Call-Labels*)
with ‹*prog* ⊢ *nx* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx′*›
have c_1;;*prog* ⊢ *nx* ⊕ #:c_1 −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx′* ⊕ #:c_1
by(*fastforce intro*:*Proc-CFG-edge-SeqSecond-source-not-Entry*)

hence $c_1;;prog,procs \vdash (Main,nx \oplus \#:c_1)\ -(\lambda s.\ False)_{\surd}\rightarrow\ (Main,nx' \oplus \#:c_1)$
by $-(rule\ PCFG.MainCallReturn)$
with $\langle(Main,\ n) = targetnode\ a\rangle\ \langle(Main,\ nx') = targetnode\ a\rangle[THEN\ sym]$
show *?thesis* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
qed
next
case (*ProcCallReturn p ins outs c nx p' es' rets' n' ps*)
from $\langle(p,\ nx) = sourcenode\ a\rangle[THEN\ sym]\ \langle(p,\ n') = targetnode\ a\rangle[THEN\ sym]$
$\langle(p,\ ins,\ outs,\ c) \in set\ procs\rangle\ \langle well\text{-}formed\ procs\rangle$
$\langle(Main,\ n) = sourcenode\ a \vee (Main,\ n) = targetnode\ a\rangle$
have *False* **by** *fastforce*
thus *?case* **by** *simp*
qed
qed

lemma *path-Main-SeqSecond*:
assumes *Rep-wf-prog wfp* = (*prog*,*procs*) **and** *Rep-wf-prog wfp'* = (c_1;;*prog*,*procs*)
shows $[\![wfp \vdash (Main,n)\ -as\rightarrow*\ (p',n');\ \forall\ a \in set\ as.\ intra\text{-}kind\ (kind\ a);\ n \neq Entry]\!]$
$\Longrightarrow wfp' \vdash (Main,n \oplus \#:c_1)\ -as\ \oplus s\ \#:c_1\rightarrow*\ (p',n' \oplus \#:c_1)$
proof(*induct* (*Main*,*n*) *as* (*p'*,*n'*) *arbitrary*:*n rule*:*ProcCFG.path.induct*)
case *empty-path*
from $\langle CFG.valid\text{-}node\ sourcenode\ targetnode\ (valid\text{-}edge\ wfp)\ (Main,\ n')\rangle$
$\langle n' \neq Entry\rangle\ \langle Rep\text{-}wf\text{-}prog\ wfp = (prog,procs)\rangle$
$\langle Rep\text{-}wf\text{-}prog\ wfp' = (c_1;;prog,procs)\rangle$
have *CFG.valid-node sourcenode targetnode* (*valid-edge wfp'*) (*Main*, $n' \oplus \#:c_1$)
by(*fastforce intro*:*valid-node-Main-SeqSecond*)
with $\langle Main = p'\rangle$ **show** *?case*
by(*fastforce intro*:*ProcCFG.empty-path simp*:*label-incrs-def*)
next
case (*Cons-path n'' as a n*)
note *IH* = $\langle\bigwedge n.\ [\![n'' = (Main,\ n);\ \forall\ a{\in}set\ as.\ intra\text{-}kind\ (kind\ a);\ n \neq Entry]\!]$
$\Longrightarrow wfp' \vdash (Main,\ n \oplus \#:c_1)\ -as\ \oplus s\ \#:c_1\rightarrow*\ (p',\ n' \oplus \#:c_1)\rangle$
note [*simp*] = $\langle Rep\text{-}wf\text{-}prog\ wfp = (prog,procs)\rangle\ \langle Rep\text{-}wf\text{-}prog\ wfp' = (c_1;;prog,procs)\rangle$
from $\langle Rep\text{-}wf\text{-}prog\ wfp = (prog,procs)\rangle$ **have** *wf*:*well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
from $\langle\forall\ a{\in}set\ (a\ \#\ as).\ intra\text{-}kind\ (kind\ a)\rangle$ **have** *intra-kind* (*kind a*)
and $\forall\ a{\in}set\ as.\ intra\text{-}kind\ (kind\ a)$ **by** *simp-all*
from $\langle valid\text{-}edge\ wfp\ a\rangle\ \langle sourcenode\ a = (Main,\ n)\rangle\ \langle targetnode\ a = n''\rangle$
$\langle intra\text{-}kind\ (kind\ a)\rangle\ wf$
obtain *nx''* **where** $n'' = (Main,nx'')$ **and** $nx'' \neq Entry$
by(*auto elim*!:*PCFG.cases simp*:*valid-edge-def intra-kind-def*)
from *IH*[*OF* $\langle n'' = (Main,nx'')\rangle\ \langle\forall\ a{\in}set\ as.\ intra\text{-}kind\ (kind\ a)\rangle\ \langle nx'' \neq Entry\rangle$]
have *path*:$wfp' \vdash (Main,\ nx'' \oplus \#:c_1)\ -as\ \oplus s\ \#:c_1\rightarrow*\ (p',\ n' \oplus \#:c_1)$ **.**
from $\langle valid\text{-}edge\ wfp\ a\rangle\ \langle sourcenode\ a = (Main,\ n)\rangle\ \langle targetnode\ a = n''\rangle$
$\langle n'' = (Main,nx'')\rangle\ \langle n \neq Entry\rangle\ \langle intra\text{-}kind\ (kind\ a)\rangle\ wf$
have $c_1;;\ prog,procs \vdash (Main,\ n \oplus \#:c_1)\ -kind\ a\rightarrow\ (Main,\ nx'' \oplus \#:c_1)$
by(*fastforce intro*:*PCFG-Main-edge-SeqSecond-source-not-Entry simp*:*valid-edge-def*)

with *path* ‹*sourcenode a = (Main, n)*› ‹*targetnode a = n''*› ‹*n'' = (Main,nx'')*›
show *?case* **apply**(*cases a*) **apply**(*clarsimp simp*:*label-incrs-def*)
by(*auto intro*:*ProcCFG.Cons-path simp*:*valid-edge-def*)
qed

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

lemma *Proc-CFG-edge-CondTrue-source-not-Entry*:
⟦*prog* ⊢ *n* −*et*→$_p$ *n'*; *n* ≠ *Entry*⟧ ⟹ *if (b) prog else* c_2 ⊢ *n* ⊕ *1* −*et*→$_p$ *n'* ⊕ *1*
by(*induct rule*:*Proc-CFG.induct*)(*fastforce intro*:*Proc-CFG-CondThen Proc-CFG.intros*)+

lemma *PCFG-Main-edge-CondTrue-source-not-Entry*:
⟦*prog*,*procs* ⊢ (*Main*,*n*) −*et*→ (*p'*,*n'*); *n* ≠ *Entry*; *intra-kind et*; *well-formed procs*⟧
⟹ *if (b) prog else* c_2,*procs* ⊢ (*Main*,*n* ⊕ *1*) −*et*→ (*p'*,*n'* ⊕ *1*)
proof(*induct* (*Main*,*n*) *et* (*p'*,*n'*) *rule*:*PCFG.induct*)
case *Main*
thus *?case* **by**(*fastforce dest*:*Proc-CFG-edge-CondTrue-source-not-Entry intro*:*PCFG.Main*)
next
case (*MainCallReturn p es rets*)
from ‹*prog* ⊢ *n* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n'*› ‹*n* ≠ *Entry*›
have *if (b) prog else* c_2 ⊢ *n* ⊕ *1* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n'* ⊕ *1*
by(*rule Proc-CFG-edge-CondTrue-source-not-Entry*)
with *MainCallReturn* **show** *?case* **by**(*fastforce intro*:*PCFG.MainCallReturn*)
qed (*auto simp*:*intra-kind-def*)

lemma *valid-node-Main-CondTrue*:
assumes *CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*Main*,*n*)
and *n* ≠ *Entry* **and** *Rep-wf-prog wfp* = (*prog*,*procs*)
and *Rep-wf-prog wfp'* = (*if (b) prog else* c_2,*procs*)
shows *CFG.valid-node sourcenode targetnode* (*valid-edge wfp'*) (*Main*, *n* ⊕ *1*)
proof −
note [*simp*] = ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
‹*Rep-wf-prog wfp'* = (*if (b) prog else* c_2,*procs*)›
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *wf*:*well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
from ‹*CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*Main*,*n*)›
obtain *a* **where** *prog*,*procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*
and (*Main*,*n*) = *sourcenode a* ∨ (*Main*,*n*) = *targetnode a*
by(*fastforce simp*:*ProcCFG.valid-node-def valid-edge-def*)
from *this* ‹*n* ≠ *Entry*› *wf* **show** *?thesis*
proof(*induct sourcenode a kind a targetnode a rule*:*PCFG.induct*)
case (*Main nx nx'*)
from ‹(*Main*,*n*) = *sourcenode a* ∨ (*Main*,*n*) = *targetnode a*› **show** *?case*
proof
assume (*Main*,*n*) = *sourcenode a*
with ‹(*Main*, *nx*) = *sourcenode a*›[*THEN sym*] **have** [*simp*]:*nx* = *n* **by** *simp*

```
    from ‹n ≠ Entry› ‹prog ⊢ nx −IEdge (kind a)→p nx'›
    have if (b) prog else c2 ⊢ n ⊕ 1 −IEdge (kind a)→p nx' ⊕ 1
      by(fastforce intro:Proc-CFG-edge-CondTrue-source-not-Entry)
    hence if (b) prog else c2,procs ⊢ (Main,n ⊕ 1) −kind a→ (Main,nx' ⊕ 1)
      by(rule PCFG.Main)
  thus ?thesis by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
 next
    assume (Main, n) = targetnode a
    show ?thesis
    proof(cases nx = Entry)
      case True
      with ‹prog ⊢ nx −IEdge (kind a)→p nx'›
      have nx' = Exit ∨ nx' = Label 0 by(fastforce dest:Proc-CFG-EntryD)
      thus ?thesis
      proof
        assume nx' = Exit
       with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
        show ?thesis by simp
      next
        assume nx' = Label 0
        have if (b) prog else c2 ⊢ Label 0
          −IEdge (λcf. state-check cf b (Some true))√→p Label 1
          by(rule Proc-CFG-CondTrue)
        with ‹nx' = Label 0›
        have if (b) prog else c2,procs ⊢ (Main,Label 0)
          −(λcf. state-check cf b (Some true))√→ (Main,nx' ⊕ 1)
          by(fastforce intro:PCFG.Main)
       with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
        show ?thesis
          by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
      qed
    next
      case False
      with ‹prog ⊢ nx −IEdge (kind a)→p nx'›
      have if (b) prog else c2 ⊢ nx ⊕ 1 −IEdge (kind a)→p nx' ⊕ 1
        by(fastforce intro:Proc-CFG-edge-CondTrue-source-not-Entry)
      hence if (b) prog else c2,procs ⊢ (Main,nx ⊕ 1) −kind a→
        (Main,nx' ⊕ 1) by(rule PCFG.Main)
      with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
    show ?thesis by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
    qed
  qed
next
  case (Proc p ins outs c nx n' ps)
  from ‹(p, nx) = sourcenode a›[THEN sym] ‹(p, n') = targetnode a›[THEN sym]
    ‹(p, ins, outs, c) ∈ set procs› ‹well-formed procs›
    ‹(Main, n) = sourcenode a ∨ (Main, n) = targetnode a›
  have False by fastforce
  thus ?case by simp
```

next
case (*MainCall l p es rets n′ ins outs c*)
from ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹(*p*, *Entry*) = *targetnode a*›[*THEN sym*]
‹(*Main*, *Label l*) = *sourcenode a*›[*THEN sym*] *wf*
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have [*simp*]:*n* = *Label l* **by** *fastforce*
from ‹*prog* ⊢ *Label l* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n′*›
have *if* (*b*) *prog else c*$_2$ ⊢ *Label l* ⊕ *1* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n′* ⊕ *1*
by −(*rule Proc-CFG-edge-CondTrue-source-not-Entry*,*auto*)
with ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*›
have *if* (*b*) *prog else c*$_2$,*procs* ⊢ (*Main*,*Label* (*l* + *1*))
−(λ*s*. *True*):(*Main*,*n′* ⊕ *1*)↪$_p$*map* (λ*e cf*. *interpret e cf*) *es*→ (*p*,*Entry*)
by(*fastforce intro*:*PCFG.MainCall*)
thus *?case* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
next
case (*ProcCall p ins outs c l p′ es′ rets′ l′ ins′ outs′ c′ ps*)
from ‹(*p*, *Label l*) = *sourcenode a*›[*THEN sym*]
‹(*p′*, *Entry*) = *targetnode a*›[*THEN sym*] ‹*well-formed procs*›
‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹(*p′*, *ins′*, *outs′*, *c′*) ∈ *set procs*›
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have *False* **by** *fastforce*
thus *?case* **by** *simp*
next
case (*MainReturn l p es rets l′ ins outs c*)
from ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹(*p*, *Exit*) = *sourcenode a*›[*THEN sym*]
‹(*Main*, *Label l′*) = *targetnode a*›[*THEN sym*] *wf*
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have [*simp*]:*n* = *Label l′* **by** *fastforce*
from ‹*prog* ⊢ *Label l* −*CEdge* (*p*, *es*, *rets*)→$_p$ *Label l′*›
have *if* (*b*) *prog else c*$_2$ ⊢ *Label l* ⊕ *1* −*CEdge* (*p*, *es*, *rets*)→$_p$ *Label l′* ⊕ *1*
by −(*rule Proc-CFG-edge-CondTrue-source-not-Entry*,*auto*)
with ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*›
have *if* (*b*) *prog else c*$_2$,*procs* ⊢ (*p*,*Exit*) −(λ*cf*. *snd cf* = (*Main*,*Label l′* ⊕ *1*))↩$_p$
(λ*cf cf′*. *cf′*(*rets* [:=] *map cf outs*))→ (*Main*,*Label* (*l′* + *1*))
by(*fastforce intro*:*PCFG.MainReturn*)
thus *?case* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
next
case (*ProcReturn p ins outs c l p′ es′ rets′ l′ ins′ outs′ c′ ps*)
from ‹(*p′*, *Exit*) = *sourcenode a*›[*THEN sym*]
‹(*p*, *Label l′*) = *targetnode a*›[*THEN sym*] ‹*well-formed procs*›
‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹(*p′*, *ins′*, *outs′*, *c′*) ∈ *set procs*›
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have *False* **by** *fastforce*
thus *?case* **by** *simp*
next
case (*MainCallReturn nx p es rets nx′*)
from ‹(*Main*,*n*) = *sourcenode a* ∨ (*Main*,*n*) = *targetnode a*› **show** *?case*
proof

assume (*Main*,*n*) = *sourcenode a*
with ‹(*Main*, *nx*) = *sourcenode a*›[*THEN sym*] **have** [*simp*]:*nx* = *n* **by** *simp*
from ‹*n* ≠ *Entry*› ‹*prog* ⊢ *nx* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx*'›
have *if* (*b*) *prog else* c_2 ⊢ *n* ⊕ *1* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx*' ⊕ *1*
by(*fastforce intro*:*Proc-CFG-edge-CondTrue-source-not-Entry*)
hence *if* (*b*) *prog else* c_2,*procs* ⊢ (*Main*,*n* ⊕ *1*) −(λ*s*. *False*)$_\surd$→ (*Main*,*nx*' ⊕ *1*) **by** −(*rule PCFG.MainCallReturn*)
thus *?thesis* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
next
assume (*Main*, *n*) = *targetnode a*
from ‹*prog* ⊢ *nx* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx*'›
have *nx* ≠ *Entry* **by**(*fastforce dest*:*Proc-CFG-Call-Labels*)
with ‹*prog* ⊢ *nx* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx*'›
have *if* (*b*) *prog else* c_2 ⊢ *nx* ⊕ *1* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx*' ⊕ *1*
by(*fastforce intro*:*Proc-CFG-edge-CondTrue-source-not-Entry*)
hence *if* (*b*) *prog else* c_2,*procs* ⊢ (*Main*,*nx* ⊕ *1*) −(λ*s*. *False*)$_\surd$→ (*Main*,*nx*' ⊕ *1*)
by −(*rule PCFG.MainCallReturn*)
with ‹(*Main*, *n*) = *targetnode a*› ‹(*Main*, *nx*') = *targetnode a*›[*THEN sym*]
show *?thesis* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
qed
next
case (*ProcCallReturn p ins outs c nx p*' *es*' *rets*' *n*' *ps*)
from ‹(*p*, *nx*) = *sourcenode a*›[*THEN sym*] ‹(*p*, *n*') = *targetnode a*›[*THEN sym*]
‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹*well-formed procs*›
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have *False* **by** *fastforce*
thus *?case* **by** *simp*
qed
qed

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*→∗ (*p*',*n*'); ∀ *a* ∈ *set as*. *intra-kind* (*kind a*); *n* ≠ *Entry*⟧
⟹ *wfp*' ⊢ (*Main*,*n* ⊕ *1*) −*as* ⊕*s* *1*→∗ (*p*',*n*' ⊕ *1*)
proof(*induct* (*Main*,*n*) *as* (*p*',*n*') *arbitrary*:*n rule*:*ProcCFG.path.induct*)
case *empty-path*
from ‹*CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*Main*, *n*')›
‹*n*' ≠ *Entry*› ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
‹*Rep-wf-prog wfp*' = (*if* (*b*) *prog else* c_2,*procs*)›
have *CFG.valid-node sourcenode targetnode* (*valid-edge wfp*') (*Main*, *n*' ⊕ *1*)
by(*fastforce intro*:*valid-node-Main-CondTrue*)
with ‹*Main* = *p*'› **show** *?case*
by(*fastforce intro*:*ProcCFG.empty-path simp*:*label-incrs-def*)
next
case (*Cons-path n*'' *as a n*)

note *IH* = ‹$\bigwedge n$. $[\![n'' = (Main,\ n);\ \forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a);\ n \neq Entry]\!]$
$\Longrightarrow$ *wfp'* $\vdash$ $(Main,\ n \oplus 1)\ -as\ \oplus s\ 1{\to}*\ (p',\ n' \oplus 1)$›
note [*simp*] = ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
‹*Rep-wf-prog wfp'* = (*if* (*b*) *prog else* c_2,*procs*)›
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *wf*:*well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
from ‹$\forall\, a{\in}set\ (a\ \#\ as).$ *intra-kind* (*kind a*)› **have** *intra-kind* (*kind a*)
and $\forall\, a{\in}set\ as.$ *intra-kind* (*kind a*) **by** *simp-all*
from ‹*valid-edge wfp a*› ‹*sourcenode a* = (*Main*, *n*)› ‹*targetnode a* = *n''*›
‹*intra-kind* (*kind a*)› *wf*
obtain *nx''* **where** *n''* = (*Main*,*nx''*) **and** *nx''* $\neq$ *Entry*
by(*auto elim*!:*PCFG.cases simp*:*valid-edge-def intra-kind-def*)
from *IH*[*OF* ‹*n''* = (*Main*,*nx''*)› ‹$\forall\, a{\in}set\ as.$ *intra-kind* (*kind a*)› ‹*nx''* $\neq$ *Entry*›]
have *path*:*wfp'* $\vdash$ $(Main,\ nx'' \oplus 1)\ -as\ \oplus s\ 1{\to}*\ (p',\ n' \oplus 1)$ **.**
from ‹*valid-edge wfp a*› ‹*sourcenode a* = (*Main*, *n*)› ‹*targetnode a* = *n''*›
‹*n''* = (*Main*,*nx''*)› ‹*n* $\neq$ *Entry*› ‹*intra-kind* (*kind a*)› *wf*
have *if* (*b*) *prog else* c_2,*procs* $\vdash$ $(Main,\ n \oplus 1)\ -kind\ a{\to}\ (Main,\ nx'' \oplus 1)$
by(*fastforce intro*:*PCFG-Main-edge-CondTrue-source-not-Entry simp*:*valid-edge-def*)
with *path* ‹*sourcenode a* = (*Main*, *n*)› ‹*targetnode a* = *n''*› ‹*n''* = (*Main*,*nx''*)›
show *?case*
apply(*cases a*) **apply**(*clarsimp simp*:*label-incrs-def*)
by(*auto intro*:*ProcCFG.Cons-path simp*:*valid-edge-def*)
qed

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

lemma *Proc-CFG-edge-CondFalse-source-not-Entry*:
$[\![$*prog* $\vdash$ *n* $-et{\to}_p$ *n'*; *n* $\neq$ *Entry*$]\!]$
$\Longrightarrow$ *if* (*b*) c_1 *else prog* $\vdash$ *n* $\oplus$ (#:c_1 + 1) $-et{\to}_p$ *n'* $\oplus$ (#:c_1 + 1)
by(*induct rule*:*Proc-CFG.induct*)(*fastforce intro*:*Proc-CFG-CondElse Proc-CFG.intros*)+

lemma *PCFG-Main-edge-CondFalse-source-not-Entry*:
$[\![$*prog*,*procs* $\vdash$ (*Main*,*n*) $-et{\to}$ (*p'*,*n'*); *n* $\neq$ *Entry*; *intra-kind et*; *well-formed procs*$]\!]$
$\Longrightarrow$ *if* (*b*) c_1 *else prog*,*procs* $\vdash$ (*Main*,*n* $\oplus$ (#:c_1 + 1)) $-et{\to}$ (*p'*,*n'* $\oplus$ (#:c_1 + 1))
proof(*induct* (*Main*,*n*) *et* (*p'*,*n'*) *rule*:*PCFG.induct*)
case *Main*
thus *?case*
by(*fastforce dest*:*Proc-CFG-edge-CondFalse-source-not-Entry intro*:*PCFG.Main*)
next
case (*MainCallReturn p es rets*)
from ‹*prog* $\vdash$ *n* $-CEdge\ (p,\ es,\ rets){\to}_p$ *n'*› ‹*n* $\neq$ *Entry*›
have *if* (*b*) c_1 *else prog* $\vdash$ *n* $\oplus$ (#:c_1 + 1) $-CEdge\ (p,\ es,\ rets){\to}_p$ *n'* $\oplus$ (#:c_1 + 1)
by(*rule Proc-CFG-edge-CondFalse-source-not-Entry*)
with *MainCallReturn* **show** *?case* **by**(*fastforce intro*:*PCFG.MainCallReturn*)
qed (*auto simp*:*intra-kind-def*)

```
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) c₁ else prog,procs)
  shows CFG.valid-node sourcenode targetnode (valid-edge wfp')
  (Main, n ⊕ (#:c₁ + 1))
proof −
  note [simp] = ‹Rep-wf-prog wfp = (prog,procs)›
    ‹Rep-wf-prog wfp' = (if (b) c₁ else prog,procs)›
  from ‹Rep-wf-prog wfp = (prog,procs)› have wf:well-formed procs
    by(fastforce intro:wf-wf-prog)
  from ‹CFG.valid-node sourcenode targetnode (valid-edge wfp) (Main,n)›
  obtain a where prog,procs ⊢ sourcenode a −kind a→ targetnode a
    and (Main,n) = sourcenode a ∨ (Main,n) = targetnode a
    by(fastforce simp:ProcCFG.valid-node-def valid-edge-def)
  from this ‹n ≠ Entry› wf show ?thesis
  proof(induct sourcenode a kind a targetnode a rule:PCFG.induct)
    case (Main nx nx')
    from ‹(Main,n) = sourcenode a ∨ (Main,n) = targetnode a› show ?case
    proof
      assume (Main,n) = sourcenode a
      with ‹(Main, nx) = sourcenode a›[THEN sym] have [simp]:nx = n by simp
      from ‹n ≠ Entry› ‹prog ⊢ nx −IEdge (kind a)→ₚ nx'›
      have if (b) c₁ else prog ⊢ n ⊕ (#:c₁ + 1) −IEdge (kind a)→ₚ nx' ⊕ (#:c₁
+ 1)
        by(fastforce intro:Proc-CFG-edge-CondFalse-source-not-Entry)
      hence if (b) c₁ else prog,procs ⊢ (Main,n ⊕ (#:c₁ + 1)) −kind a→
        (Main,nx' ⊕ (#:c₁ + 1)) by(rule PCFG.Main)
    thus ?thesis by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
    next
      assume (Main, n) = targetnode a
      show ?thesis
      proof(cases nx = Entry)
        case True
        with ‹prog ⊢ nx −IEdge (kind a)→ₚ nx'›
        have nx' = Exit ∨ nx' = Label 0 by(fastforce dest:Proc-CFG-EntryD)
        thus ?thesis
        proof
          assume nx' = Exit
         with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
          show ?thesis by simp
        next
          assume nx' = Label 0
          have if (b) c₁ else prog ⊢ Label 0
            −IEdge (λcf. state-check cf b (Some false))√→ₚ Label (#:c₁ + 1)
            by(rule Proc-CFG-CondFalse)
          with ‹nx' = Label 0›
```

have *if* (*b*) c_1 *else prog*,*procs* ⊢ (*Main*,*Label 0*)
−(λ*cf*. *state-check cf b* (*Some false*))$_{\surd}\rightarrow$ (*Main*,*nx'* ⊕ (#:c_1 + *1*))
by(*fastforce intro*:*PCFG.Main*)
with ‹(*Main*, *n*) = *targetnode a*› ‹(*Main*, *nx'*) = *targetnode a*›[*THEN sym*]
show *?thesis*
by(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
qed
next
case *False*
with ‹*prog* ⊢ *nx* −*IEdge* (*kind a*)$\rightarrow_p$ *nx'*›
have *if* (*b*) c_1 *else prog* ⊢ *nx* ⊕ (#:c_1 + *1*) −*IEdge* (*kind a*)$\rightarrow_p$ *nx'* ⊕ (#:c_1 + *1*)
by(*fastforce intro*:*Proc-CFG-edge-CondFalse-source-not-Entry*)
hence *if* (*b*) c_1 *else prog*,*procs* ⊢ (*Main*,*nx* ⊕ (#:c_1 + *1*)) −*kind a*→ (*Main*,*nx'* ⊕ (#:c_1 + *1*)) **by**(*rule PCFG.Main*)
with ‹(*Main*, *n*) = *targetnode a*› ‹(*Main*, *nx'*) = *targetnode a*›[*THEN sym*]
show *?thesis* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
qed
qed
next
case (*Proc p ins outs c nx n' ps*)
from ‹(*p*, *nx*) = *sourcenode a*›[*THEN sym*] ‹(*p*, *n'*) = *targetnode a*›[*THEN sym*]
‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹*well-formed procs*›
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have *False* **by** *fastforce*
thus *?case* **by** *simp*
next
case (*MainCall l p es rets n' ins outs c*)
from ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹(*p*, *Entry*) = *targetnode a*›[*THEN sym*]
‹(*Main*, *Label l*) = *sourcenode a*›[*THEN sym*] *wf*
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have [*simp*]:*n* = *Label l* **by** *fastforce*
from ‹*prog* ⊢ *Label l* −*CEdge* (*p*, *es*, *rets*)$\rightarrow_p$ *n'*›
have *if* (*b*) c_1 *else prog* ⊢ *Label l* ⊕ (#:c_1 + *1*) −*CEdge* (*p*, *es*, *rets*)$\rightarrow_p$ *n'* ⊕ (#:c_1 + *1*) **by** −(*rule Proc-CFG-edge-CondFalse-source-not-Entry*,*auto*)
with ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*›
have *if* (*b*) c_1 *else prog*,*procs* ⊢ (*Main*,*Label* (*l* + (#:c_1 + *1*)))
−(λ*s*. *True*):(*Main*,*n'* ⊕ (#:c_1 + *1*))↪$_p$*map* (λ*e cf*. *interpret e cf*) *es*→(*p*,*Entry*)
by(*fastforce intro*:*PCFG.MainCall*)
thus *?case* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
next
case (*ProcCall p ins outs c l p' es' rets' l' ins' outs' c' ps*)
from ‹(*p*, *Label l*) = *sourcenode a*›[*THEN sym*]
‹(*p'*, *Entry*) = *targetnode a*›[*THEN sym*] ‹*well-formed procs*›
‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹(*p'*, *ins'*, *outs'*, *c'*) ∈ *set procs*›
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have *False* **by** *fastforce*
thus *?case* **by** *simp*

next
case (*MainReturn l p es rets l′ ins outs c*)
from ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹(*p*, *Exit*) = *sourcenode a*›[*THEN sym*]
‹(*Main*, *Label l′*) = *targetnode a*›[*THEN sym*] *wf*
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have [*simp*]:*n* = *Label l′* **by** *fastforce*
from ‹*prog* ⊢ *Label l* −*CEdge* (*p*, *es*, *rets*)→$_p$ *Label l′*›
have *if* (*b*) *c*$_1$ *else prog* ⊢ *Label l* ⊕ (#:*c*$_1$ + *1*) −*CEdge* (*p*, *es*, *rets*)→$_p$
Label l′ ⊕ (#:*c*$_1$ + *1*) **by** −(*rule Proc-CFG-edge-CondFalse-source-not-Entry*,*auto*)
with ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*›
have *if* (*b*) *c*$_1$ *else prog*,*procs* ⊢ (*p*,*Exit*)
−(λ*cf*. *snd cf* = (*Main*,*Label l′* ⊕ (#:*c*$_1$ + *1*)))↩$_p$
(λ*cf cf′*. *cf′*(*rets* [:=] *map cf outs*))→ (*Main*,*Label* (*l′* + (#:*c*$_1$ + *1*)))
by(*fastforce intro*:*PCFG.MainReturn*)
thus *?case* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
next
case (*ProcReturn p ins outs c l p′ es′ rets′ l′ ins′ outs′ c′ ps*)
from ‹(*p′*, *Exit*) = *sourcenode a*›[*THEN sym*]
‹(*p*, *Label l′*) = *targetnode a*›[*THEN sym*] ‹*well-formed procs*›
‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹(*p′*, *ins′*, *outs′*, *c′*) ∈ *set procs*›
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have *False* **by** *fastforce*
thus *?case* **by** *simp*
next
case (*MainCallReturn nx p es rets nx′*)
from ‹(*Main*,*n*) = *sourcenode a* ∨ (*Main*,*n*) = *targetnode a*› **show** *?case*
proof
assume (*Main*,*n*) = *sourcenode a*
with ‹(*Main*, *nx*) = *sourcenode a*›[*THEN sym*] **have** [*simp*]:*nx* = *n* **by** *simp*
from ‹*n* ≠ *Entry*› ‹*prog* ⊢ *nx* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx′*›
have *if* (*b*) *c*$_1$ *else prog* ⊢ *n* ⊕ (#:*c*$_1$ + *1*) −*CEdge* (*p*, *es*, *rets*)→$_p$
nx′ ⊕ (#:*c*$_1$ + *1*) **by**(*fastforce intro*:*Proc-CFG-edge-CondFalse-source-not-Entry*)
hence *if* (*b*) *c*$_1$ *else prog*,*procs* ⊢ (*Main*,*n* ⊕ (#:*c*$_1$ + *1*))
−(λ*s*. *False*)$_{\surd}$→ (*Main*,*nx′* ⊕ (#:*c*$_1$ + *1*)) **by** −(*rule PCFG.MainCallReturn*)
thus *?thesis* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
next
assume (*Main*, *n*) = *targetnode a*
from ‹*prog* ⊢ *nx* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx′*›
have *nx* ≠ *Entry* **by**(*fastforce dest*:*Proc-CFG-Call-Labels*)
with ‹*prog* ⊢ *nx* −*CEdge* (*p*, *es*, *rets*)→$_p$ *nx′*›
have *if* (*b*) *c*$_1$ *else prog* ⊢ *nx* ⊕ (#:*c*$_1$ + *1*) −*CEdge* (*p*, *es*, *rets*)→$_p$
nx′ ⊕ (#:*c*$_1$ + *1*) **by**(*fastforce intro*:*Proc-CFG-edge-CondFalse-source-not-Entry*)
hence *if* (*b*) *c*$_1$ *else prog*,*procs* ⊢ (*Main*,*nx* ⊕ (#:*c*$_1$ + *1*))
−(λ*s*. *False*)$_{\surd}$→ (*Main*,*nx′* ⊕ (#:*c*$_1$ + *1*)) **by** −(*rule PCFG.MainCallReturn*)
with ‹(*Main*, *n*) = *targetnode a*› ‹(*Main*, *nx′*) = *targetnode a*›[*THEN sym*]
show *?thesis* **by**(*simp add*:*ProcCFG.valid-node-def*)(*fastforce simp*:*valid-edge-def*)
qed
next
case (*ProcCallReturn p ins outs c nx p′ es′ rets′ n′ ps*)

from ‹(*p*, *nx*) = *sourcenode a*›[*THEN sym*] ‹(*p*, *n'*) = *targetnode a*›[*THEN sym*]
‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*› ‹*well-formed procs*›
‹(*Main*, *n*) = *sourcenode a* ∨ (*Main*, *n*) = *targetnode a*›
have *False* **by** *fastforce*
thus *?case* **by** *simp*
qed
qed

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(*induct* (*Main*,*n*) *as* (*p'*,*n'*) *arbitrary*:*n rule*:*ProcCFG.path.induct*)
case *empty-path*
from ‹*CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*Main*, *n'*)›
‹*n'* ≠ *Entry*› ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
‹*Rep-wf-prog wfp'* = (*if* (*b*) c_1 *else prog*,*procs*)›
have *CFG.valid-node sourcenode targetnode* (*valid-edge wfp'*) (*Main*, *n'* ⊕ (#:c_1 + *1*))
by(*fastforce intro*:*valid-node-Main-CondFalse*)
with ‹*Main* = *p'*› **show** *?case*
by(*fastforce intro*:*ProcCFG.empty-path simp*:*label-incrs-def*)
next
case (*Cons-path n'' as a n*)
note *IH* = ‹⋀*n*. ⋀*n*. ⟦*n''* = (*Main*, *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*))›
note [*simp*] = ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
‹*Rep-wf-prog wfp'* = (*if* (*b*) c_1 *else prog*,*procs*)›
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *wf*:*well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
from ‹∀ *a*∈*set* (*a* # *as*). *intra-kind* (*kind a*)› **have** *intra-kind* (*kind a*)
and ∀ *a*∈*set as*. *intra-kind* (*kind a*) **by** *simp-all*
from ‹*valid-edge wfp a*› ‹*sourcenode a* = (*Main*, *n*)› ‹*targetnode a* = *n''*›
‹*intra-kind* (*kind a*)› *wf*
obtain *nx''* **where** *n''* = (*Main*,*nx''*) **and** *nx''* ≠ *Entry*
by(*auto elim*!:*PCFG.cases simp*:*valid-edge-def intra-kind-def*)
from *IH*[*OF* ‹*n''* = (*Main*,*nx''*)› ‹∀ *a*∈*set as*. *intra-kind* (*kind a*)› ‹*nx''* ≠ *Entry*›]
have *path*:*wfp'* ⊢ (*Main*, *nx''* ⊕ (#:c_1 + *1*)) −*as* ⊕*s* (#:c_1 + *1*)→*
(*p'*, *n'* ⊕ (#:c_1 + *1*)) **.**
from ‹*valid-edge wfp a*› ‹*sourcenode a* = (*Main*, *n*)› ‹*targetnode a* = *n''*›
‹*n''* = (*Main*,*nx''*)› ‹*n* ≠ *Entry*› ‹*intra-kind* (*kind a*)› *wf*
have *if* (*b*) c_1 *else prog*,*procs* ⊢ (*Main*, *n* ⊕ (#:c_1 + *1*)) −*kind a*→
(*Main*, *nx''* ⊕ (#:c_1 + *1*))
by(*fastforce intro*:*PCFG-Main-edge-CondFalse-source-not-Entry simp*:*valid-edge-def*)

with *path* ‹*sourcenode a* = (*Main*, *n*)› ‹*targetnode a* = *n''*› ‹*n''* = (*Main*,*nx''*)›
show *?case*
apply(*cases a*) **apply**(*clarsimp simp*:*label-incrs-def*)
by(*auto intro*:*ProcCFG.Cons-path simp*:*valid-edge-def*)
qed

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
by(*induct rule*:*Proc-CFG.induct*)(*fastforce intro*:*Proc-CFG-WhileBody Proc-CFG.intros*)+

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(*induct* (*Main*,*n*) *et* (*p'*,*n'*) *rule*:*PCFG.induct*)
case *Main*
thus *?case*
by(*fastforce dest*:*Proc-CFG-edge-WhileBody-source-not-Entry intro*:*PCFG.Main*)
next
case (*MainCallReturn p es rets*)
from ‹*prog* ⊢ *n* −*CEdge* (*p*, *es*, *rets*)→$_p$ *n'*› ‹*n* ≠ *Entry*› ‹*n'* ≠ *Exit*›
have *while* (*b*) *prog* ⊢ *n* ⊕ 2 −*CEdge* (*p*, *es*, *rets*)→$_p$ *n'* ⊕ 2
by(*rule Proc-CFG-edge-WhileBody-source-not-Entry*)
with *MainCallReturn* **show** *?case* **by**(*fastforce intro*:*PCFG.MainCallReturn*)
qed (*auto simp*:*intra-kind-def*)

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 −
note [*simp*] = ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
‹*Rep-wf-prog wfp'* = (*while* (*b*) *prog*,*procs*)›
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *wf*:*well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
from ‹*CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*Main*,*n*)›
obtain *a* **where** *prog*,*procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*
and (*Main*,*n*) = *sourcenode a* ∨ (*Main*,*n*) = *targetnode a*
by(*fastforce simp*:*ProcCFG.valid-node-def valid-edge-def*)
from *this* ‹*n* ≠ *Entry*› *wf* **show** *?thesis*
proof(*induct sourcenode a kind a targetnode a rule*:*PCFG.induct*)
case (*Main nx nx'*)
from ‹(*Main*,*n*) = *sourcenode a* ∨ (*Main*,*n*) = *targetnode a*› **show** *?case*
proof

```
  assume (Main,n) = sourcenode a
  with ‹(Main, nx) = sourcenode a›[THEN sym] have [simp]:nx = n by simp
  show ?thesis
  proof(cases nx' = Exit)
    case True
    with ‹n ≠ Entry› ‹prog ⊢ nx −IEdge (kind a)→p nx'›
    have while (b) prog ⊢ n ⊕ 2 −IEdge (kind a)→p Label 0
      by(fastforce intro:Proc-CFG-WhileBodyExit)
    hence while (b) prog,procs ⊢ (Main,n ⊕ 2) −kind a→ (Main,Label 0)
      by(rule PCFG.Main)
  thus ?thesis by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
  next
    case False
    with ‹n ≠ Entry› ‹prog ⊢ nx −IEdge (kind a)→p nx'›
    have while (b) prog ⊢ n ⊕ 2 −IEdge (kind a)→p nx' ⊕ 2
      by(fastforce intro:Proc-CFG-edge-WhileBody-source-not-Entry)
    hence while (b) prog,procs ⊢ (Main,n ⊕ 2) −kind a→ (Main,nx' ⊕ 2)
      by(rule PCFG.Main)
  thus ?thesis by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
  qed
next
  assume (Main, n) = targetnode a
  show ?thesis
  proof(cases nx = Entry)
    case True
    with ‹prog ⊢ nx −IEdge (kind a)→p nx'›
    have nx' = Exit ∨ nx' = Label 0 by(fastforce dest:Proc-CFG-EntryD)
    thus ?thesis
    proof
      assume nx' = Exit
     with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
      show ?thesis by simp
    next
      assume nx' = Label 0
      have while (b) prog ⊢ Label 0
        −IEdge (λcf. state-check cf b (Some true))√→p Label 2
        by(rule Proc-CFG-WhileTrue)
      hence while (b) prog,procs ⊢ (Main,Label 0)
        −(λcf. state-check cf b (Some true))√→ (Main,Label 2)
        by(fastforce intro:PCFG.Main)
     with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
        ‹nx' = Label 0› show ?thesis
        by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
    qed
  next
    case False
    show ?thesis
    proof(cases nx' = Exit)
      case True
```

```
      with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
       show ?thesis by simp
     next
      case False
      with ‹prog ⊢ nx −IEdge (kind a)→p nx'› ‹nx ≠ Entry›
      have while (b) prog ⊢ nx ⊕ 2 −IEdge (kind a)→p nx' ⊕ 2
        by(fastforce intro:Proc-CFG-edge-WhileBody-source-not-Entry)
      hence while (b) prog,procs ⊢ (Main,nx ⊕ 2) −kind a→
        (Main,nx' ⊕ 2) by(rule PCFG.Main)
     with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
      show ?thesis
        by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
     qed
    qed
  qed
next
  case (Proc p ins outs c nx n' ps)
 from ‹(p, nx) = sourcenode a›[THEN sym] ‹(p, n') = targetnode a›[THEN sym]
    ‹(Main, n) = sourcenode a ∨ (Main, n) = targetnode a›
    ‹(p, ins, outs, c) ∈ set procs› ‹well-formed procs›
  have False by fastforce
  thus ?case by simp
next
  case (MainCall l p es rets n' ins outs c)
  from ‹(p, ins, outs, c) ∈ set procs› ‹(p, Entry) = targetnode a›[THEN sym]
    ‹(Main, Label l) = sourcenode a›[THEN sym] wf
    ‹(Main, n) = sourcenode a ∨ (Main, n) = targetnode a›
  have [simp]:n = Label l by fastforce
  from ‹prog ⊢ Label l −CEdge (p, es, rets)→p n'› have n' ≠ Exit
    by(fastforce dest:Proc-CFG-Call-Labels)
  with ‹prog ⊢ Label l −CEdge (p, es, rets)→p n'›
  have while (b) prog ⊢ Label l ⊕ 2 −CEdge (p, es, rets)→p
    n' ⊕ 2 by −(rule Proc-CFG-edge-WhileBody-source-not-Entry,auto)
  with ‹(p, ins, outs, c) ∈ set procs›
  have while (b) prog,procs ⊢ (Main,Label l ⊕ 2)
    −(λs. True):(Main,n' ⊕ 2)↪pmap (λe cf. interpret e cf) es→ (p,Entry)
    by(fastforce intro:PCFG.MainCall)
 thus ?case by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
next
  case (ProcCall p ins outs c l p' es' rets' l' ins' outs' c')
  from ‹(p, Label l) = sourcenode a›[THEN sym]
    ‹(p', Entry) = targetnode a›[THEN sym] ‹well-formed procs›
    ‹(p, ins, outs, c) ∈ set procs› ‹(p', ins', outs', c') ∈ set procs›
    ‹(Main, n) = sourcenode a ∨ (Main, n) = targetnode a›
  have False by fastforce
  thus ?case by simp
next
  case (MainReturn l p es rets l' ins outs c)
  from ‹(p, ins, outs, c) ∈ set procs› ‹(p, Exit) = sourcenode a›[THEN sym]
```

```
    ‹(Main, Label l') = targetnode a›[THEN sym] wf
    ‹(Main, n) = sourcenode a ∨ (Main, n) = targetnode a›
  have [simp]:n = Label l' by fastforce
  from ‹prog ⊢ Label l −CEdge (p, es, rets)→p Label l'›
  have while (b) prog ⊢ Label l ⊕ 2 −CEdge (p, es, rets)→p
    Label l' ⊕ 2 by −(rule Proc-CFG-edge-WhileBody-source-not-Entry,auto)
  with ‹(p, ins, outs, c) ∈ set procs›
  have while (b) prog,procs ⊢ (p,Exit) −(λcf. snd cf = (Main,Label l' ⊕ 2))↩p
    (λcf cf'. cf'(rets [:=] map cf outs))→ (Main,Label l' ⊕ 2)
    by(fastforce intro:PCFG.MainReturn)
 thus ?case by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
next
  case (ProcReturn p ins outs c l p' es' rets' l' ins' outs' c' ps)
  from ‹(p', Exit) = sourcenode a›[THEN sym]
    ‹(p, Label l') = targetnode a›[THEN sym] ‹well-formed procs›
    ‹(p, ins, outs, c) ∈ set procs› ‹(p', ins', outs', c') ∈ set procs›
    ‹(Main, n) = sourcenode a ∨ (Main, n) = targetnode a›
  have False by fastforce
  thus ?case by simp
next
  case (MainCallReturn nx p es rets nx')
  from ‹(Main,n) = sourcenode a ∨ (Main,n) = targetnode a› show ?case
  proof
    assume (Main,n) = sourcenode a
    with ‹(Main, nx) = sourcenode a›[THEN sym] have [simp]:nx = n by simp
    from ‹prog ⊢ nx −CEdge (p, es, rets)→p nx'› have nx' ≠ Exit
      by(fastforce dest:Proc-CFG-Call-Labels)
    with ‹n ≠ Entry› ‹prog ⊢ nx −CEdge (p, es, rets)→p nx'›
    have while (b) prog ⊢ n ⊕ 2 −CEdge (p, es, rets)→p
      nx' ⊕ 2 by(fastforce intro:Proc-CFG-edge-WhileBody-source-not-Entry)
   hence while (b) prog,procs ⊢ (Main,n ⊕ 2) −(λs. False)√→ (Main,nx' ⊕ 2)
      by −(rule PCFG.MainCallReturn)
  thus ?thesis by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
  next
    assume (Main, n) = targetnode a
    from ‹prog ⊢ nx −CEdge (p, es, rets)→p nx'›
    have nx ≠ Entry and nx' ≠ Exit by(auto dest:Proc-CFG-Call-Labels)
    with ‹prog ⊢ nx −CEdge (p, es, rets)→p nx'›
    have while (b) prog ⊢ nx ⊕ 2 −CEdge (p, es, rets)→p
      nx' ⊕ 2 by(fastforce intro:Proc-CFG-edge-WhileBody-source-not-Entry)
     hence while (b) prog,procs ⊢ (Main,nx ⊕ 2) −(λs. False)√→ (Main,nx' ⊕
2)
      by −(rule PCFG.MainCallReturn)
    with ‹(Main, n) = targetnode a› ‹(Main, nx') = targetnode a›[THEN sym]
   show ?thesis by(simp add:ProcCFG.valid-node-def)(fastforce simp:valid-edge-def)
  qed
next
  case (ProcCallReturn p ins outs c nx p' es' rets' n' ps)
 from ‹(p, nx) = sourcenode a›[THEN sym] ‹(p, n') = targetnode a›[THEN sym]
```

```
    ‹(p, ins, outs, c) ∈ set procs› ‹well-formed procs›
    ‹(Main, n) = sourcenode a ∨ (Main, n) = targetnode a›
   have False by fastforce
   thus ?case by simp
  qed
qed


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(induct (Main,n) as (p',n') arbitrary:n rule:ProcCFG.path.induct)
  case empty-path
  from ‹CFG.valid-node sourcenode targetnode (valid-edge wfp) (Main, n')›
    ‹n' ≠ Entry› ‹Rep-wf-prog wfp = (prog,procs)›
    ‹Rep-wf-prog wfp' = (while (b) prog,procs)›
  have CFG.valid-node sourcenode targetnode (valid-edge wfp') (Main, n' ⊕ 2)
    by(fastforce intro:valid-node-Main-WhileBody)
  with ‹Main = p'› show ?case
    by(fastforce intro:ProcCFG.empty-path simp:label-incrs-def)
next
  case (Cons-path n'' as a n)
  note IH = ‹⋀n. ⟦n'' = (Main, n); ∀ a∈set as. intra-kind (kind a); n ≠ Entry;
    n' ≠ Exit⟧ ⟹ wfp' ⊢ (Main, n ⊕ 2) −as ⊕s 2→* (p', n' ⊕ 2)›
  note [simp] = ‹Rep-wf-prog wfp = (prog,procs)›
    ‹Rep-wf-prog wfp' = (while (b) prog,procs)›
  from ‹Rep-wf-prog wfp = (prog,procs)› have wf:well-formed procs
    by(fastforce intro:wf-wf-prog)
  from ‹∀ a∈set (a # as). intra-kind (kind a)› have intra-kind (kind a)
    and ∀ a∈set as. intra-kind (kind a) by simp-all
  from ‹valid-edge wfp a› ‹sourcenode a = (Main, n)› ‹targetnode a = n''›
    ‹intra-kind (kind a)› wf
  obtain nx'' where n'' = (Main,nx'') and nx'' ≠ Entry
    by(auto elim!:PCFG.cases simp:valid-edge-def intra-kind-def)
  from IH[OF ‹n'' = (Main,nx'')› ‹∀ a∈set as. intra-kind (kind a)›
    ‹nx'' ≠ Entry› ‹n' ≠ Exit›]
  have path:wfp' ⊢ (Main, nx'' ⊕ 2) −as ⊕s 2→* (p', n' ⊕ 2) .
 with ‹n' ≠ Exit› have nx'' ≠ Exit by(fastforce dest:ProcCFGExit.path-Exit-source)
  with ‹valid-edge wfp a› ‹sourcenode a = (Main, n)› ‹targetnode a = n''›
    ‹n'' = (Main,nx'')› ‹n ≠ Entry› ‹intra-kind (kind a)› wf
  have while (b) prog,procs ⊢ (Main, n ⊕ 2) −kind a→ (Main, nx'' ⊕ 2)
   by(fastforce intro:PCFG-Main-edge-WhileBody-source-not-Entry simp:valid-edge-def)
  with path ‹sourcenode a = (Main, n)› ‹targetnode a = n''› ‹n'' = (Main,nx'')›
  show ?case
    apply(cases a) apply(clarsimp simp:label-incrs-def)
    by(auto intro:ProcCFG.Cons-path simp:valid-edge-def)
qed
```

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(*atomize-elim*)
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *wf*:*well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
from ‹*l* < #:*prog*› ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
show ∃ *as as'*. *wfp* ⊢ (*Main*, *Label l*) −*as*→∗ (*Main*, *Exit*) ∧
(∀ *a*∈*set as*. *intra-kind* (*kind a*)) ∧
wfp ⊢ (*Main*, *Entry*) −*as'*→∗ (*Main*, *Label l*) ∧ (∀ *a*∈*set as'*. *intra-kind* (*kind a*))
proof(*induct prog arbitrary*:*l wfp*)
case *Skip*
note [*simp*] = ‹*Rep-wf-prog wfp* = (*Skip*, *procs*)›
from ‹*l* < #:*Skip*› **have** [*simp*]:*l* = *0* **by** *simp*
have *wfp* ⊢ (*Main*,*Entry*) −[((*Main*,*Entry*),(λ*s*. *True*)$_{\surd}$,(*Main*,*Label 0*))]→∗
(*Main*,*Label 0*)
by(*fastforce intro*:*ProcCFG.path.intros Main Proc-CFG-Entry*
simp:*valid-edge-def ProcCFG.valid-node-def*)
moreover
have *wfp* ⊢ (*Main*,*Label l*) −[((*Main*,*Label l*),⇑*id*,(*Main*,*Exit*))]→∗ (*Main*,*Exit*)

by(*fastforce intro*:*ProcCFG.path.intros Main Proc-CFG-Skip simp*:*valid-edge-def*)
ultimately show *?case* **by**(*fastforce simp*:*intra-kind-def*)
next
case (*LAss V e*)
note [*simp*] = ‹*Rep-wf-prog wfp* = (*V*:=*e*, *procs*)›
from ‹*l* < #:*V*:=*e*› **have** *l* = *0* ∨ *l* = *1* **by** *auto*
thus *?case*
proof
assume [*simp*]:*l* = *0*
have *wfp* ⊢ (*Main*,*Entry*) −[((*Main*,*Entry*),(λ*s*. *True*)$_{\surd}$,(*Main*,*Label 0*))]→∗
(*Main*,*Label 0*)
by(*fastforce intro*:*ProcCFG.path.intros Main Proc-CFG-Entry*
simp:*valid-edge-def ProcCFG.valid-node-def*)
moreover
have *wfp* ⊢ (*Main*,*Label 0*)
−((*Main*,*Label 0*),⇑(λ*cf*. *update cf V e*),(*Main*,*Label 1*))#
[((*Main*,*Label 1*),⇑*id*,(*Main*,*Exit*))]→∗ (*Main*,*Exit*)
by(*fastforce intro*:*ProcCFG.Cons-path ProcCFG.path.intros Main Proc-CFG-LAss*

Proc-CFG-LAssSkip simp:*valid-edge-def ProcCFG.valid-node-def*)
ultimately show *?thesis* **by**(*fastforce simp*:*intra-kind-def*)
next
assume [*simp*]:*l* = *1*

have *wfp* ⊢ (*Main*,*Entry*) −((*Main*,*Entry*),(λ*s*. *True*)$_{\surd}$,(*Main*,*Label 0*))#
[((*Main*,*Label 0*),⇑(λ*cf*. *update cf V e*),(*Main*,*Label 1*))]→∗ (*Main*,*Label 1*)
by(*fastforce intro*:*ProcCFG.path.intros Main Proc-CFG-LAss ProcCFG.Cons-path*

Main Proc-CFG-Entry simp:*ProcCFG.valid-node-def valid-edge-def*)
moreover
have *wfp* ⊢ (*Main*,*Label 1*) −[((*Main*,*Label 1*),⇑*id*,(*Main*,*Exit*))]→∗
(*Main*,*Exit*) **by**(*fastforce intro*:*ProcCFG.path.intros Main Proc-CFG-LAssSkip*
simp:*valid-edge-def ProcCFG.valid-node-def*)
ultimately show *?thesis* **by**(*fastforce simp*:*intra-kind-def*)
qed
next
case (*Seq* c_1 c_2)
note *IH1* = ‹⋀*l wfp*. ⟦*l* < #:c_1; *Rep-wf-prog wfp* = (c_1, *procs*)⟧ ⟹
∃ *as as'*. *wfp* ⊢ (*Main*, *Label l*) −*as*→∗ (*Main*, *Exit*) ∧
(∀ *a*∈*set as*. *intra-kind* (*kind a*)) ∧
wfp ⊢ (*Main*, *Entry*) −*as'*→∗ (*Main*, *Label l*) ∧ (∀ *a*∈*set as'*. *intra-kind* (*kind*
a))›
note *IH2* = ‹⋀*l wfp*. ⟦*l* < #:c_2; *Rep-wf-prog wfp* = (c_2, *procs*)⟧ ⟹
∃ *as as'*. *wfp* ⊢ (*Main*, *Label l*) −*as*→∗ (*Main*, *Exit*) ∧
(∀ *a*∈*set as*. *intra-kind* (*kind a*)) ∧
wfp ⊢ (*Main*, *Entry*) −*as'*→∗ (*Main*, *Label l*) ∧ (∀ *a*∈*set as'*. *intra-kind* (*kind*
a))›
note [*simp*] = ‹*Rep-wf-prog wfp* = (c_1;; c_2, *procs*)›
show *?case*
proof(*cases l* < #:c_1)
case *True*
from ‹*Rep-wf-prog wfp* = (c_1;; c_2, *procs*)›
obtain *wfp'* **where** [*simp*]:*Rep-wf-prog wfp'* = (c_1, *procs*) **by**(*erule wfp-Seq1*)
from *IH1*[*OF True this*] **obtain** *as as'*
where *path1*:*wfp'* ⊢ (*Main*, *Label l*) −*as*→∗ (*Main*, *Exit*)
and *intra1*:∀ *a*∈*set as*. *intra-kind* (*kind a*)
and *path2*:*wfp'* ⊢ (*Main*, *Entry*) −*as'*→∗ (*Main*, *Label l*)
and *intra2*:∀ *a*∈*set as'*. *intra-kind* (*kind a*) **by** *blast*
from *path1* **have** *as* ≠ [] **by**(*fastforce elim*:*ProcCFG.path.cases*)
then obtain *ax asx* **where** [*simp*]:*as* = *asx*@[*ax*]
by(*cases as rule*:*rev-cases*) *fastforce*+
with *path1* **have** *wfp'* ⊢ (*Main*, *Label l*) −*asx*→∗ *sourcenode ax*
and *valid-edge wfp' ax* **and** *targetnode ax* = (*Main*, *Exit*)
by(*auto elim*:*ProcCFG.path-split-snoc*)
from ‹*valid-edge wfp' ax*› ‹*targetnode ax* = (*Main*, *Exit*)›
obtain *nx* **where** *sourcenode ax* = (*Main*,*nx*)
by(*fastforce elim*:*PCFG.cases simp*:*valid-edge-def*)
with ‹*wfp'* ⊢ (*Main*, *Label l*) −*asx*→∗ *sourcenode ax*› **have** *nx* ≠ *Entry*
by *fastforce*
moreover
from ‹*valid-edge wfp' ax*› ‹*sourcenode ax* = (*Main*,*nx*)› **have** *nx* ≠ *Exit*
by(*fastforce intro*:*ProcCFGExit.Exit-source*)
ultimately obtain *lx* **where** [*simp*]:*nx* = *Label lx* **by**(*cases nx*) *auto*

with ‹*wfp'* ⊢ (*Main*, *Label l*) −*asx*→∗ *sourcenode ax*›
‹*sourcenode ax* = (*Main*,*nx*)› *intra1*
have *path3*:*wfp* ⊢ (*Main*, *Label l*) −*asx*→∗ (*Main*, *Label lx*)
by −(*rule path-SeqFirst*,*auto*)
from ‹*valid-edge wfp' ax*› ‹*targetnode ax* = (*Main*, *Exit*)›
‹*sourcenode ax* = (*Main*,*nx*)› *wf*
obtain *etx* **where** c_1 ⊢ *Label lx* −*etx*→$_p$ *Exit*
by(*fastforce elim*!:*PCFG.cases simp*:*valid-edge-def*)
then obtain *et* **where** [*simp*]:*etx* = *IEdge et*
by(*cases etx*)(*auto dest*:*Proc-CFG-Call-Labels*)
with ‹c_1 ⊢ *Label lx* −*etx*→$_p$ *Exit*› **have** *intra-kind et*
by(*fastforce intro*:*Proc-CFG-IEdge-intra-kind*)
from ‹c_1 ⊢ *Label lx* −*etx*→$_p$ *Exit*› *path3*
have *path4*:*wfp* ⊢ (*Main*, *Label l*) −*asx*@
[((*Main*, *Label lx*),*et*,(*Main*,*Label 0* ⊕ #:c_1))] →∗ (*Main*,*Label 0* ⊕ #:c_1)
by(*fastforce intro*:*ProcCFG.path-Append ProcCFG.path.intros Proc-CFG-SeqConnect*
Main simp:*ProcCFG.valid-node-def valid-edge-def*)
from ‹*Rep-wf-prog wfp* = (c_1;; c_2, *procs*)›
obtain *wfp''* **where** [*simp*]:*Rep-wf-prog wfp''* = (c_2, *procs*) **by**(*erule wfp-Seq2*)
from *IH2*[*OF - this*,*of 0*] **obtain** *asx'*
where *wfp''* ⊢ (*Main*, *Label 0*) −*asx'*→∗ (*Main*, *Exit*)
and ∀ *a*∈*set asx'*. *intra-kind* (*kind a*) **by** *blast*
with *path4 intra1* ‹*intra-kind et*› **have** *wfp* ⊢ (*Main*, *Label l*)
−(*asx*@[((*Main*, *Label lx*),*et*,(*Main*,*Label 0* ⊕ #:c_1))])@(*asx'* ⊕*s* #:c_1)→∗
(*Main*, *Exit* ⊕ #:c_1)
by −(*erule ProcCFG.path-Append*,*rule path-Main-SeqSecond*,*auto*)
moreover
from *intra1* ‹*intra-kind et*› ‹∀ *a*∈*set asx'*. *intra-kind* (*kind a*)›
have ∀ *a* ∈ *set* ((*asx*@[((*Main*, *Label lx*),*et*,(*Main*,*Label* #:c_1))])@(*asx'* ⊕*s*
#:c_1)).
intra-kind (*kind a*) **by**(*auto simp*:*label-incrs-def*)
moreover
from *path2 intra2* **have** *wfp* ⊢ (*Main*, *Entry*) −*as'*→∗ (*Main*, *Label l*)
by −(*rule path-SeqFirst*,*auto*)
ultimately show *?thesis* **using** ‹∀ *a*∈*set as'*. *intra-kind* (*kind a*)› **by** *fastforce*
next
case *False*
hence #:c_1 ≤ *l* **by** *simp*
then obtain *l'* **where** [*simp*]:*l* = *l'* + #:c_1 **and** *l'* = *l* − #:c_1 **by** *simp*
from ‹*l* < #:c_1;; c_2› **have** *l'* < #:c_2 **by** *simp*
from ‹*Rep-wf-prog wfp* = (c_1;; c_2, *procs*)›
obtain *wfp'* **where** [*simp*]:*Rep-wf-prog wfp'* = (c_2, *procs*) **by**(*erule wfp-Seq2*)
from *IH2*[*OF* ‹*l'* < #:c_2› *this*] **obtain** *as as'*
where *path1*:*wfp'* ⊢ (*Main*, *Label l'*) −*as*→∗ (*Main*, *Exit*)
and *intra1*:∀ *a*∈*set as*. *intra-kind* (*kind a*)
and *path2*:*wfp'* ⊢ (*Main*, *Entry*) −*as'*→∗ (*Main*, *Label l'*)
and *intra2*:∀ *a*∈*set as'*. *intra-kind* (*kind a*) **by** *blast*
from *path1 intra1*
have *wfp* ⊢ (*Main*, *Label l'* ⊕ #:c_1) −*as* ⊕*s* #:c_1→∗ (*Main*, *Exit* ⊕ #:c_1)

by −(*rule path-Main-SeqSecond*,*auto*)
moreover
from *path2* **have** $as' \neq []$ **by**(*fastforce elim*:*ProcCFG.path.cases*)
with *path2* **obtain** ax' asx' **where** [*simp*]:$as' = ax'\#asx'$
and *sourcenode* $ax' = (Main,\ Entry)$ **and** *valid-edge* wfp' ax'
and $wfp' \vdash$ *targetnode* $ax' -asx'\rightarrow* (Main,\ Label\ l')$
by −(*erule ProcCFG.path-split-Cons*,*fastforce*+)
from ‹$wfp' \vdash$ *targetnode* $ax' -asx'\rightarrow* (Main,\ Label\ l')$›
have *targetnode* $ax' \neq (Main,Exit)$ **by** *fastforce*
with ‹*valid-edge* wfp' ax'› ‹*sourcenode* $ax' = (Main,\ Entry)$› *wf*
have *targetnode* $ax' = (Main,Label\ 0)$
by(*fastforce elim*:*PCFG.cases dest*:*Proc-CFG-EntryD simp*:*valid-edge-def*)
with ‹$wfp' \vdash$ *targetnode* $ax' -asx'\rightarrow* (Main,\ Label\ l')$› *intra2*
have *path3*:$wfp \vdash (Main,Label\ 0 \oplus \#{:}c_1)\ -asx' \oplus s\ \#{:}c_1\rightarrow*$
$(Main,\ Label\ l' \oplus \#{:}c_1)$ **by** −(*rule path-Main-SeqSecond*,*auto*)
from ‹*Rep-wf-prog* $wfp = (c_1;;\ c_2,\ procs)$›
obtain wfp'' **where** [*simp*]:*Rep-wf-prog* $wfp'' = (c_1,\ procs)$ **by**(*erule wfp-Seq1*)
from *IH1*[*OF - this*,*of 0*] **obtain** *xs*
where $wfp'' \vdash (Main,\ Label\ 0)\ -xs\rightarrow* (Main,\ Exit)$
and $\forall\ a{\in}set\ xs.$ *intra-kind* (*kind a*) **by** *blast*
from ‹$wfp'' \vdash (Main,\ Label\ 0)\ -xs\rightarrow* (Main,\ Exit)$› **have** $xs \neq []$
by(*fastforce elim*:*ProcCFG.path.cases*)
then obtain *x xs'* **where** [*simp*]:$xs = xs'@[x]$
by(*cases xs rule*:*rev-cases*) *fastforce*+
with ‹$wfp'' \vdash (Main,\ Label\ 0)\ -xs\rightarrow* (Main,\ Exit)$›
have $wfp'' \vdash (Main,\ Label\ 0)\ -xs'\rightarrow*$ *sourcenode x*
and *valid-edge* wfp'' *x* **and** *targetnode* $x = (Main,\ Exit)$
by(*auto elim*:*ProcCFG.path-split-snoc*)
from ‹*valid-edge* wfp'' *x*› ‹*targetnode* $x = (Main,\ Exit)$›
obtain *nx* **where** *sourcenode* $x = (Main,nx)$
by(*fastforce elim*:*PCFG.cases simp*:*valid-edge-def*)
with ‹$wfp'' \vdash (Main,\ Label\ 0)\ -xs'\rightarrow*$ *sourcenode x*› **have** $nx \neq Entry$
by *fastforce*
from ‹*valid-edge* wfp'' *x*› ‹*sourcenode* $x = (Main,nx)$› **have** $nx \neq Exit$
by(*fastforce intro*:*ProcCFGExit.Exit-source*)
with ‹$nx \neq Entry$› **obtain** *lx* **where** [*simp*]:$nx = Label\ lx$ **by**(*cases nx*) *auto*
from ‹$wfp'' \vdash (Main,\ Label\ 0)\ -xs'\rightarrow*$ *sourcenode x*›
‹*sourcenode* $x = (Main,nx)$› ‹$\forall\ a{\in}set\ xs.$ *intra-kind* (*kind a*)›
have $wfp \vdash (Main,\ Entry)$
$-((Main,\ Entry),(\lambda s.\ True)_{\surd},(Main,\ Label\ 0))\#xs'\rightarrow*$ *sourcenode x*
apply *simp* **apply**(*rule path-SeqFirst*[*OF* ‹*Rep-wf-prog* $wfp'' = (c_1,\ procs)$›])
apply(*auto intro*!:*ProcCFG.Cons-path*)
by(*auto intro*:*Main Proc-CFG-Entry simp*:*valid-edge-def intra-kind-def*)
with ‹*valid-edge* wfp'' *x*› ‹*targetnode* $x = (Main,\ Exit)$› *path3*
‹*sourcenode* $x = (Main,nx)$› ‹$nx \neq Entry$› ‹*sourcenode* $x = (Main,nx)$› *wf*
have $wfp \vdash (Main,\ Entry)\ -((((Main,\ Entry),(\lambda s.\ True)_{\surd},(Main,\ Label\ 0))\#xs')@$
$[(sourcenode\ x,kind\ x,(Main,Label\ \#{:}c_1))])@(asx' \oplus s\ \#{:}c_1)\rightarrow*$
$(Main,\ Label\ l' \oplus \#{:}c_1)$

```
      by(fastforce intro:ProcCFG.path-Append ProcCFG.path.intros Main
        Proc-CFG-SeqConnect elim!:PCFG.cases dest:Proc-CFG-Call-Labels
        simp:ProcCFG.valid-node-def valid-edge-def)
    ultimately show ?thesis using intra1 intra2 ‹∀ a∈set xs. intra-kind (kind
a)›
      by(fastforce simp:label-incrs-def intra-kind-def)
  qed
 next
  case (Cond b c1 c2)
  note IH1 = ‹⋀l wfp. ⟦l < #:c1; Rep-wf-prog wfp = (c1, procs)⟧ ⟹
   ∃as as'. wfp ⊢ (Main, Label l) −as→* (Main, Exit) ∧
   (∀ a∈set as. intra-kind (kind a)) ∧
   wfp ⊢ (Main, Entry) −as'→* (Main, Label l) ∧ (∀ a∈set as'. intra-kind (kind
a))›
  note IH2 = ‹⋀l wfp. ⟦l < #:c2; Rep-wf-prog wfp = (c2, procs)⟧ ⟹
   ∃as as'. wfp ⊢ (Main, Label l) −as→* (Main, Exit) ∧
   (∀ a∈set as. intra-kind (kind a)) ∧
   wfp ⊢ (Main, Entry) −as'→* (Main, Label l) ∧ (∀ a∈set as'. intra-kind (kind
a))›
  note [simp] = ‹Rep-wf-prog wfp = (if (b) c1 else c2, procs)›
  show ?case
  proof(cases l = 0)
   case True
   from ‹Rep-wf-prog wfp = (if (b) c1 else c2, procs)›
  obtain wfp' where [simp]:Rep-wf-prog wfp' = (c1, procs) by(erule wfp-CondTrue)
   from IH1[OF - this,of 0] obtain as
    where path:wfp' ⊢ (Main, Label 0) −as→* (Main, Exit)
    and intra:∀ a∈set as. intra-kind (kind a) by blast
   have if (b) c1 else c2,procs ⊢ (Main,Label 0)
    −(λcf. state-check cf b (Some true))√→ (Main,Label 0 ⊕ 1)
    by(fastforce intro:Main Proc-CFG-CondTrue)
   with path intra have wfp ⊢ (Main,Label 0)
     −[((Main,Label 0),(λcf. state-check cf b (Some true))√,(Main,Label 0 ⊕
1))]@
     (as ⊕s 1)→* (Main,Exit ⊕ 1)
   apply − apply(rule ProcCFG.path-Append) apply(rule ProcCFG.path.intros)+
    prefer 5 apply(rule path-Main-CondTrue)
    apply(auto intro:ProcCFG.path.intros simp:valid-edge-def)
    by(fastforce simp:ProcCFG.valid-node-def valid-edge-def)
   moreover
   have if (b) c1 else c2,procs ⊢ (Main,Entry) −(λs. True)√→
    (Main,Label 0) by(fastforce intro:Main Proc-CFG-Entry)
  hence wfp ⊢ (Main,Entry) −[((Main,Entry),(λs. True)√,(Main,Label 0))]→*

    (Main,Label 0)
    by(fastforce intro:ProcCFG.path.intros
            simp:ProcCFG.valid-node-def valid-edge-def)
   ultimately show ?thesis using ‹l = 0› ‹∀ a∈set as. intra-kind (kind a)›
    by(fastforce simp:label-incrs-def intra-kind-def)
```

```
next
  case False
  hence 0 < l by simp
  then obtain l' where [simp]:l = l' + 1 and l' = l - 1 by simp
  show ?thesis
  proof(cases l' < #:c1)
    case True
    from ‹Rep-wf-prog wfp = (if (b) c1 else c2, procs)›
    obtain wfp' where [simp]:Rep-wf-prog wfp' = (c1, procs)
      by(erule wfp-CondTrue)
    from IH1[OF True this] obtain as as'
      where path1:wfp' ⊢ (Main, Label l') -as→* (Main, Exit)
      and intra1:∀ a∈set as. intra-kind (kind a)
      and path2:wfp' ⊢ (Main, Entry) -as'→* (Main, Label l')
      and intra2:∀ a∈set as'. intra-kind (kind a) by blast
    from path1 intra1
    have wfp ⊢ (Main, Label l' ⊕ 1) -as ⊕s 1→* (Main, Exit ⊕ 1)
      by -(rule path-Main-CondTrue,auto)
    moreover
    from path2 obtain ax' asx' where [simp]:as' = ax'#asx'
      and sourcenode ax' = (Main,Entry) and valid-edge wfp' ax'
      and wfp' ⊢ targetnode ax' -asx'→* (Main, Label l')
      by -(erule ProcCFG.path.cases,fastforce+)
    with wf have targetnode ax' = (Main,Label 0)
    by(fastforce elim:PCFG.cases dest:Proc-CFG-EntryD Proc-CFG-Call-Labels

                simp:valid-edge-def)
    with ‹wfp' ⊢ targetnode ax' -asx'→* (Main, Label l')› intra2
    have wfp ⊢ (Main,Entry) -((Main,Entry),(λs. True)√,(Main,Label 0))#
       ((Main,Label 0),(λcf. state-check cf b (Some true))√,(Main,Label 0 ⊕
1))#
      (asx' ⊕s 1)→* (Main,Label l' ⊕ 1)
    apply - apply(rule ProcCFG.path.intros)+ apply(rule path-Main-CondTrue)

    by(auto intro:Main Proc-CFG-Entry Proc-CFG-CondTrue simp:valid-edge-def)
    ultimately show ?thesis using intra1 intra2
      by(fastforce simp:label-incrs-def intra-kind-def)
  next
    case False
    hence #:c1 ≤ l' by simp
    then obtain l'' where [simp]:l' = l'' + #:c1 and l'' = l' - #:c1 by simp
    from ‹l < #:(if (b) c1 else c2)› have l'' < #:c2 by simp
    from ‹Rep-wf-prog wfp = (if (b) c1 else c2, procs)›
    obtain wfp'' where [simp]:Rep-wf-prog wfp'' = (c2, procs)
      by(erule wfp-CondFalse)
    from IH2[OF ‹l'' < #:c2› this] obtain as as'
      where path1:wfp'' ⊢ (Main, Label l'') -as→* (Main, Exit)
      and intra1:∀ a∈set as. intra-kind (kind a)
      and path2:wfp'' ⊢ (Main, Entry) -as'→* (Main, Label l'')
```

```
      and intra2:∀ a∈set as'. intra-kind (kind a) by blast
     from path1 intra1
     have wfp ⊢ (Main, Label l'' ⊕ (#:c1 + 1)) −as ⊕s (#:c1 + 1)→∗
      (Main, Exit ⊕ (#:c1 + 1))
      by −(rule path-Main-CondFalse,auto simp:add-assoc)
     moreover
     from path2 obtain ax' asx' where [simp]:as' = ax'#asx'
      and sourcenode ax' = (Main,Entry) and valid-edge wfp'' ax'
      and wfp'' ⊢ targetnode ax' −asx'→∗ (Main, Label l'')
      by −(erule ProcCFG.path.cases,fastforce+)
     with wf have targetnode ax' = (Main,Label 0)
     by(fastforce elim:PCFG.cases dest:Proc-CFG-EntryD Proc-CFG-Call-Labels
                 simp:valid-edge-def)
     with ‹wfp'' ⊢ targetnode ax' −asx'→∗ (Main, Label l'')› intra2
     have wfp ⊢ (Main,Entry) −((Main,Entry),(λs. True)√,(Main,Label 0))#
      ((Main,Label 0),(λcf. state-check cf b (Some false))√,
      (Main,Label (#:c1 + 1)))#(asx' ⊕s (#:c1 + 1))→∗
      (Main,Label l'' ⊕ (#:c1 + 1))
    apply − apply(rule ProcCFG.path.intros)+ apply(rule path-Main-CondFalse)
    by(auto intro:Main Proc-CFG-Entry Proc-CFG-CondFalse simp:valid-edge-def)
     ultimately show ?thesis using intra1 intra2
      by(fastforce simp:label-incrs-def intra-kind-def add-assoc)
    qed
   qed
  next
   case (While b c')
   note IH = ‹⋀l wfp. ⟦l < #:c'; Rep-wf-prog wfp = (c', procs)⟧ ⟹
    ∃as as'. wfp ⊢ (Main, Label l) −as→∗ (Main, Exit) ∧
    (∀ a∈set as. intra-kind (kind a)) ∧
    wfp ⊢ (Main, Entry) −as'→∗ (Main, Label l) ∧ (∀ a∈set as'. intra-kind (kind
a))›
   note [simp] = ‹Rep-wf-prog wfp = (while (b) c', procs)›
   show ?case
   proof(cases l = 0)
    case True
    hence wfp ⊢ (Main,Label l) −
     ((Main,Label 0),(λcf. state-check cf b (Some false))√,(Main,Label 1))#
     [((Main,Label 1),⇑id,(Main,Exit))]→∗ (Main,Exit)
     by(fastforce intro:ProcCFG.path.intros Main Proc-CFG-WhileFalseSkip
      Proc-CFG-WhileFalse simp:valid-edge-def)
    moreover
   have while (b) c' ⊢ Entry −IEdge (λs. True)√→p Label 0 by(rule Proc-CFG-Entry)
    with ‹l = 0› have wfp ⊢ (Main,Entry)
     −[((Main,Entry),(λs. True)√,(Main,Label 0))]→∗ (Main,Label l)
     by(fastforce intro:ProcCFG.path.intros Main
              simp:ProcCFG.valid-node-def valid-edge-def)
    ultimately show ?thesis by(fastforce simp:intra-kind-def)
   next
```

case *False*
hence *1* $\leq$ *l* **by** *simp*
thus *?thesis*
proof(*cases l* $<$ *2*)
case *True*
with ‹*1* $\leq$ *l*› **have** [*simp*]:*l* = *1* **by** *simp*
have *wfp* $\vdash$ (*Main*,*Label l*) $-[((Main,Label\ 1),\Uparrow id,(Main,Exit))]\rightarrow*$ (*Main*,*Exit*)
by(*fastforce intro*:*ProcCFG.path.intros Main Proc-CFG-WhileFalseSkip*
simp:*valid-edge-def*)
moreover
have *while* (*b*) *c'* $\vdash$ *Label 0* $-$*IEdge* $(\lambda cf.\ state\text{-}check\ cf\ b\ (Some\ false))_{\surd}\rightarrow_p$
Label 1 **by**(*rule Proc-CFG-WhileFalse*)
hence *wfp* $\vdash$ (*Main*,*Entry*) $-((Main,Entry),(\lambda s.\ True)_{\surd},(Main,Label\ 0))\#$
$[((Main,Label\ 0),(\lambda cf.\ state\text{-}check\ cf\ b\ (Some\ false))_{\surd},(Main,Label\ 1))]\rightarrow*$
(*Main*,*Label l*)
by(*fastforce intro*:*ProcCFG.path.intros Main Proc-CFG-Entry*
simp:*ProcCFG.valid-node-def valid-edge-def*)
ultimately show *?thesis* **by**(*fastforce simp*:*intra-kind-def*)
next
case *False*
with ‹*1* $\leq$ *l*› **have** *2* $\leq$ *l* **by** *simp*
then obtain *l'* **where** [*simp*]:*l* = *l'* + *2* **and** *l'* = *l* $-$ *2*
by(*simp del*:*add-2-eq-Suc'*)
from ‹*l* $<$ #:*while* (*b*) *c'*› **have** *l'* $<$ #:*c'* **by** *simp*
from ‹*Rep-wf-prog wfp* = (*while* (*b*) *c'*, *procs*)›
obtain *wfp'* **where** [*simp*]:*Rep-wf-prog wfp'* = (*c'*, *procs*)
by(*erule wfp-WhileBody*)
from *IH*[*OF* ‹*l'* $<$ #:*c'*› *this*] **obtain** *as as'*
where *path1*:*wfp'* $\vdash$ (*Main*, *Label l'*) $-as\rightarrow*$ (*Main*, *Exit*)
and *intra1*:$\forall$ *a*$\in$*set as. intra-kind* (*kind a*)
and *path2*:*wfp'* $\vdash$ (*Main*, *Entry*) $-as'\rightarrow*$ (*Main*, *Label l'*)
and *intra2*:$\forall$ *a*$\in$*set as'. intra-kind* (*kind a*) **by** *blast*
from *path1* **have** *as* $\neq$ [] **by**(*fastforce elim*:*ProcCFG.path.cases*)
with *path1* **obtain** *ax asx* **where** [*simp*]:*as* = *asx*@[*ax*]
and *wfp'* $\vdash$ (*Main*, *Label l'*) $-asx\rightarrow*$ *sourcenode ax*
and *valid-edge wfp' ax* **and** *targetnode ax* = (*Main*, *Exit*)
by $-$(*erule ProcCFG.path-split-snoc*,*fastforce*+)
with *wf* **obtain** *lx etx* **where** *sourcenode ax* = (*Main*,*Label lx*)
and *intra-kind* (*kind ax*)
apply(*auto elim*!:*PCFG.cases dest*:*Proc-CFG-Call-Labels simp*:*valid-edge-def*)
by(*case-tac n*)(*auto dest*:*Proc-CFG-IEdge-intra-kind*)
with ‹*wfp'* $\vdash$ (*Main*, *Label l'*) $-asx\rightarrow*$ *sourcenode ax*› *intra1*
have *wfp* $\vdash$ (*Main*, *Label l'* $\oplus$ *2*) $-asx\ \oplus s\ 2\rightarrow*$ (*Main*,*Label lx* $\oplus$ *2*)
by $-$(*rule path-Main-WhileBody*,*auto*)
from ‹*valid-edge wfp' ax*› ‹*sourcenode ax* = (*Main*,*Label lx*)›
‹*targetnode ax* = (*Main*, *Exit*)› ‹*intra-kind* (*kind ax*)› *wf*
have *while* (*b*) *c'*,*procs* $\vdash$ (*Main*,*Label lx* $\oplus$ *2*) $-$*kind ax*$\rightarrow$
(*Main*,*Label 0*)

```
      by(fastforce intro!:Main Proc-CFG-WhileBodyExit elim!:PCFG.cases
                dest:Proc-CFG-Call-Labels simp:valid-edge-def)
    hence wfp ⊢ (Main,Label lx ⊕ 2)
      −((Main,Label lx ⊕ 2),kind ax,(Main,Label 0))#
      ((Main,Label 0),(λcf. state-check cf b (Some false))√,(Main,Label 1))#
      [((Main,Label 1),⇑id,(Main,Exit))]→* (Main,Exit)
      by(fastforce intro:ProcCFG.path.intros Main Proc-CFG-WhileFalse
        Proc-CFG-WhileFalseSkip simp:valid-edge-def)
    with ‹wfp ⊢ (Main, Label l' ⊕ 2) −asx ⊕s 2→* (Main,Label lx ⊕ 2)›
    have wfp ⊢ (Main, Label l) −(asx ⊕s 2)@
      (((Main,Label lx ⊕ 2),kind ax,(Main,Label 0))#
      ((Main,Label 0),(λcf. state-check cf b (Some false))√,(Main,Label 1))#
      [((Main,Label 1),⇑id,(Main,Exit))])→* (Main,Exit)
      by(fastforce intro:ProcCFG.path-Append)
    moreover
    from path2 have as' ≠ [] by(fastforce elim:ProcCFG.path.cases)
    with path2 obtain ax' asx' where [simp]:as' = ax'#asx'
      and wfp' ⊢ targetnode ax' −asx'→* (Main,Label l')
      and valid-edge wfp' ax' and sourcenode ax' = (Main, Entry)
      by −(erule ProcCFG.path-split-Cons,fastforce+)
    with wf have targetnode ax' = (Main,Label 0) and intra-kind (kind ax')
      by(fastforce elim!:PCFG.cases dest:Proc-CFG-Call-Labels
        Proc-CFG-EntryD simp:intra-kind-def valid-edge-def)+
    with ‹wfp' ⊢ targetnode ax' −asx'→* (Main,Label l')› intra2
    have wfp ⊢ (Main, Label 0 ⊕ 2) −asx' ⊕s 2→* (Main,Label l' ⊕ 2)
      by −(rule path-Main-WhileBody,auto simp del:add-2-eq-Suc')
    hence wfp ⊢ (Main,Entry) −((Main,Entry),(λs. True)√,(Main,Label 0))#
      ((Main,Label 0),(λcf. state-check cf b (Some true))√,(Main,Label 2))#
      (asx' ⊕s 2)→* (Main,Label l)
      by(fastforce intro:ProcCFG.path.intros Main Proc-CFG-WhileTrue
        Proc-CFG-Entry simp:valid-edge-def)
    ultimately show ?thesis using ‹intra-kind (kind ax)› intra1 intra2
      by(fastforce simp:label-incrs-def intra-kind-def)
  qed
 qed
next
 case (Call p es rets)
 note Rep [simp] = ‹Rep-wf-prog wfp = (Call p es rets, procs)›
 have cC:containsCall procs (Call p es rets) [] p by simp
 show ?case
 proof(cases l = 0)
  case True
  have wfp ⊢ (Main,Label 0) −((Main,Label 0),(λs. False)√,(Main,Label 1))#
    [((Main,Label 1),⇑id,(Main,Exit))]→* (Main,Exit)
  by(fastforce intro:ProcCFG.path.intros Main Proc-CFG-CallSkip MainCallReturn
      Proc-CFG-Call simp:valid-edge-def)
  moreover
  have Call p es rets,procs ⊢ (Main,Entry) −(λs. True)√→ (Main,Label 0)
    by(fastforce intro:Main Proc-CFG-Entry)
```

hence *wfp* ⊢ (*Main*,*Entry*) −[((*Main*,*Entry*),(λ*s*. *True*)$_{\surd}$,(*Main*,*Label 0*))]→∗ (*Main*,*Label 0*)
by(*fastforce intro*:*ProcCFG.path.intros simp*:*ProcCFG.valid-node-def valid-edge-def*)
ultimately show *?thesis* **using** ‹*l* = *0*› **by**(*fastforce simp*:*intra-kind-def*)
next
case *False*
with ‹*l* < #:*Call p es rets*› **have** *l* = *1* **by** *simp*
have *wfp* ⊢ (*Main*,*Label 1*) −[((*Main*,*Label 1*),⇑*id*,(*Main*,*Exit*))]→∗ (*Main*,*Exit*)
by(*fastforce intro*:*ProcCFG.path.intros Main Proc-CFG-CallSkip simp*:*valid-edge-def*)
moreover
have *Call p es rets*,*procs* ⊢ (*Main*,*Label 0*) −(λ*s*. *False*)$_{\surd}$→ (*Main*,*Label 1*)
by(*fastforce intro*:*MainCallReturn Proc-CFG-Call*)
hence *wfp* ⊢ (*Main*,*Entry*) −((*Main*,*Entry*),(λ*s*. *True*)$_{\surd}$,(*Main*,*Label 0*))# [((*Main*,*Label 0*),(λ*s*. *False*)$_{\surd}$,(*Main*,*Label 1*))]→∗ (*Main*,*Label 1*)
by(*fastforce intro*:*ProcCFG.path.intros Main Proc-CFG-Entry simp*:*ProcCFG.valid-node-def valid-edge-def*)
ultimately show *?thesis* **using** ‹*l* = *1*› **by**(*fastforce simp*:*intra-kind-def*)
qed
qed
qed

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(*induct* (*Main*,*n*) *et* (*Main*,*n*′) *rule*:*PCFG.induct*)
case *Main* **thus** *?case* **by**(*fastforce intro*:*Proc*)
next
case *MainCallReturn* **thus** *?case* **by**(*fastforce intro*:*ProcCallReturn*)
qed *auto*

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(*induct* (*Main*,*n*) *et* (*q*,*n*′) *rule*:*PCFG.induct*)
case (*MainCall l esx retsx n*′*x insx outsx cx*)
from ‹*c* ⊢ *Label l* −*CEdge* (*q*, *esx*, *retsx*)→$_p$ *n*′*x*›
obtain *l*′ **where** [*simp*]:*n*′*x* = *Label l*′ **by**(*fastforce dest*:*Proc-CFG-Call-Labels*)
with *MainCall* **have** *prog*,*procs* ⊢ (*p*, *n*) −(λ*s*. *True*):(*p*,*n*′*x*)↪$_q$*map* (λ*e cf*. *interpret e cf*) *esx*→ (*q*, *n*′)
by(*fastforce intro*:*ProcCall*)
thus *?case* **by** *fastforce*
qed *auto*

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(*induct* (*q*,*n*) *et* (*Main*,*n′*) *rule*:*PCFG.induct*)
  case (*MainReturn l esx retsx l′ insx outsx cx*)
  note [*simp*] = ‹*Exit* = *n*›[*THEN sym*] ‹*Label l′* = *n′*›[*THEN sym*]
  from *MainReturn* **have** *prog*,*procs* ⊢ (*q*,*Exit*) −(λ*cf*. *snd cf* = (*p*,*Label l′*))↩*q*
    (λ*cf cf′*. *cf′*(*retsx* [:=] *map cf outsx*))→ (*p*,*Label l′*)
    by(*fastforce intro*!:*ProcReturn*)
  thus *?case* **by** *fastforce*
qed *auto*

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

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

lemma *lift-path-Proc*:
  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(*induct* (*Main*,*n*) *as* (*Main*,*n′*) *arbitrary*:*n rule*:*ProcCFG.path.induct*)
  case *empty-path*
  from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *wf*:*well-formed procs*
    by(*fastforce intro*:*wf-wf-prog*)
  from ‹*CFG.valid-node sourcenode targetnode* (*valid-edge wfp′*) (*Main*, *n′*)›
    assms wf
  have *CFG.valid-node sourcenode targetnode* (*valid-edge wfp*) (*p*,*n′*)
    apply(*auto simp*:*ProcCFG.valid-node-def valid-edge-def*)
     apply(*case-tac ab* = *Main*)
      apply(*fastforce dest*:*lift-edge-Main-Main*)
     apply(*fastforce dest*!:*lift-edge-Main-Proc*)
    apply(*case-tac a* = *Main*)
     apply(*fastforce dest*:*lift-edge-Main-Main*)
    by(*fastforce dest*!:*lift-edge-Proc-Main*)
  thus *?case* **by**(*fastforce dest*:*ProcCFG.empty-path*)
next
  case (*Cons-path m″ as a n*)
  note *IH* = ‹⋀*n*. ⟦*m″* = (*Main*, *n*); ∀ *a*∈*set as*. *intra-kind* (*kind a*)⟧
    ⟹ *wfp* ⊢ (*p*, *n*) −*lift-path as p*→∗ (*p*, *n′*)›
  from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *wf*:*well-formed procs*
    by(*fastforce intro*:*wf-wf-prog*)
  from ‹∀ *a*∈*set* (*a* # *as*). *intra-kind* (*kind a*)› **have** *intra-kind* (*kind a*)

and $\forall\, a{\in}set\ as.\ intra\text{-}kind\ (kind\ a)$ **by** *simp-all*
from ‹*valid-edge wfp′ a*› ‹*intra-kind* (*kind a*)› ‹*sourcenode a* = (*Main*, *n*)›
‹*targetnode a* = *m″*› ‹*Rep-wf-prog wfp′* = (*c*,*procs*)›
obtain *n″* **where** *m″* = (*Main*, *n″*)
by(*fastforce elim*:*PCFG.cases simp*:*valid-edge-def intra-kind-def*)
with ‹*valid-edge wfp′ a*› ‹*Rep-wf-prog wfp′* = (*c*,*procs*)›
‹*sourcenode a* = (*Main*, *n*)› ‹*targetnode a* = *m″*›
‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*› ‹*containsCall procs prog ps p*›
‹*Rep-wf-prog wfp* = (*prog*,*procs*)› *wf*
have *prog*,*procs* ⊢ (*p*, *n*) −*kind a*→ (*p*, *n″*)
by(*auto intro*:*lift-edge-Main-Main simp*:*valid-edge-def*)
from *IH*[*OF* ‹*m″* = (*Main*, *n″*)› ‹∀ *a*∈*set as. intra-kind* (*kind a*)›]
have *wfp* ⊢ (*p*, *n″*) −*lift-path as p*→∗ (*p*, *n′*) **.**
with ‹*prog*,*procs* ⊢ (*p*, *n*) −*kind a*→ (*p*, *n″*)› ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
‹*sourcenode a* = (*Main*, *n*)› ‹*targetnode a* = *m″*› ‹*m″* = (*Main*, *n″*)›
show *?case* **by** *simp* (*rule ProcCFG.Cons-path*,*auto simp*:*valid-edge-def*)
qed

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(*atomize-elim*)
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
‹*containsCall procs prog ps p*›
obtain *wfp′* **where** *Rep-wf-prog wfp′* = (*c*,*procs*) **by**(*erule wfp-Call*)
from *this* ‹*l* < #:*c*› **obtain** *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*)
by(*erule Label-Proc-CFG-Entry-Exit-path-Main*)
from ‹*Rep-wf-prog wfp′* = (*c*,*procs*)› ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*› ‹*containsCall procs prog ps p*›
‹*wfp′* ⊢ (*Main*,*Label l*) −*as*→∗ (*Main*,*Exit*)› ‹∀ *a* ∈ *set as. intra-kind* (*kind a*)›
have *wfp* ⊢ (*p*,*Label l*) −*lift-path as p*→∗ (*p*,*Exit*)
by(*fastforce intro*:*lift-path-Proc*)
moreover
from ‹*Rep-wf-prog wfp′* = (*c*,*procs*)› ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*› ‹*containsCall procs prog ps p*›
‹*wfp′* ⊢ (*Main*,*Entry*) −*as′*→∗ (*Main*,*Label l*)› ‹∀ *a* ∈ *set as′. intra-kind* (*kind*
a)›
have *wfp* ⊢ (*p*,*Entry*) −*lift-path as′ p*→∗ (*p*,*Label l*)
by(*fastforce intro*:*lift-path-Proc*)
moreover

from ‹∀ *a* ∈ *set as*. *intra-kind* (*kind a*)› ‹∀ *a* ∈ *set as′*. *intra-kind* (*kind a*)›
have ∀ *a* ∈ *set* (*lift-path as p*). *intra-kind* (*kind a*)
and ∀ *a* ∈ *set* (*lift-path as′ p*). *intra-kind* (*kind a*) **by** *auto*
ultimately
show ∃ *as as′*. *wfp* ⊢ (*p*, *Label l*) −*as*→∗ (*p*, *Exit*) ∧
(∀ *a*∈*set as*. *intra-kind* (*kind a*)) ∧ *wfp* ⊢ (*p*, *Entry*) −*as′*→∗ (*p*, *Label l*) ∧
(∀ *a*∈*set as′*. *intra-kind* (*kind a*)) **by** *fastforce*
qed

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(*atomize-elim*)
from ‹*containsCall procs prog ps p*› ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*›
show ∃ *as as′*. *CFG.valid-path′ sourcenode targetnode kind* (*valid-edge wfp*)
(*get-return-edges wfp*) (*Main*, *Entry*) *as* (*p*, *Entry*) ∧
CFG.valid-path′ sourcenode targetnode kind (*valid-edge wfp*)
(*get-return-edges wfp*) (*p*, *Exit*) *as′* (*Main*, *Exit*)
proof(*induct ps arbitrary*:*p ins outs c rule*:*rev-induct*)
case *Nil*
from ‹*containsCall procs prog* [] *p*›
obtain *lx es rets lx′* **where** *prog* ⊢ *Label lx* −*CEdge* (*p*,*es*,*rets*)→*p* *Label lx′*
by(*erule containsCall-empty-Proc-CFG-Call-edge*)
with ‹(*p*, *ins*, *outs*, *c*) ∈ *set procs*›
have *prog*,*procs* ⊢ (*Main*,*Label lx*) −(λ*s*. *True*):(*Main*,*Label lx′*)↪*p*
map (λ*e cf*. *interpret e cf*) *es*→ (*p*,*Entry*)
and *prog*,*procs* ⊢ (*p*,*Exit*) −(λ*cf*. *snd cf* = (*Main*,*Label lx′*))↩*p*
(λ*cf cf′*. *cf′*(*rets* [:=] *map cf outs*))→ (*Main*,*Label lx′*)
by −(*rule MainCall*,*assumption*+,*rule MainReturn*)
with ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
have *wfp* ⊢ (*Main*,*Label lx*) −[((*Main*,*Label lx*),
(λ*s*. *True*):(*Main*,*Label lx′*)↪*pmap* (λ*e cf*. *interpret e cf*) *es*,(*p*,*Entry*))]→∗
(*p*,*Entry*)
and *wfp* ⊢ (*p*,*Exit*) −[((*p*,*Exit*),(λ*cf*. *snd cf* = (*Main*,*Label lx′*))↩*p*
(λ*cf cf′*. *cf′*(*rets* [:=] *map cf outs*)),(*Main*,*Label lx′*))]→∗ (*Main*,*Label lx′*)
by(*fastforce intro*:*ProcCFG.path.intros*
simp:*ProcCFG.valid-node-def valid-edge-def*)+
moreover
from ‹*prog* ⊢ *Label lx* −*CEdge* (*p*,*es*,*rets*)→*p* *Label lx′*›
have *lx* < #:*prog* **and** *lx′* < #:*prog*
by(*auto intro*:*Proc-CFG-sourcelabel-less-num-nodes*
Proc-CFG-targetlabel-less-num-nodes)
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› ‹*lx* < #:*prog*› **obtain** *as*
where *wfp* ⊢ (*Main*,*Entry*) −*as*→∗ (*Main*,*Label lx*)

and $\forall\, a \in set\ as.\ intra\text{-}kind\ (kind\ a)$
by $-(erule\ Label\text{-}Proc\text{-}CFG\text{-}Entry\text{-}Exit\text{-}path\text{-}Main)$
moreover
from ‹$Rep\text{-}wf\text{-}prog\ wfp = (prog,procs)$› ‹$lx' < \#{:}prog$› **obtain** as'
where $wfp \vdash (Main,Label\ lx') -as'\rightarrow* (Main,Exit)$
and $\forall\, a \in set\ as'.\ intra\text{-}kind\ (kind\ a)$
by $-(erule\ Label\text{-}Proc\text{-}CFG\text{-}Entry\text{-}Exit\text{-}path\text{-}Main)$
moreover
from ‹$\forall\, a \in set\ as.\ intra\text{-}kind\ (kind\ a)$›
have $CFG.valid\text{-}path\ kind\ (get\text{-}return\text{-}edges\ wfp)$
$(as@[((Main,Label\ lx),(\lambda s.\ True){:}(Main,Label\ lx')\hookrightarrow_p$
$map\ (\lambda e\ cf.\ interpret\ e\ cf)\ es,(p,Entry))]$)
by($fastforce\ intro{:}ProcCFG.same\text{-}level\text{-}path\text{-}valid\text{-}path\text{-}Append$
$ProcCFG.intras\text{-}same\text{-}level\text{-}path\ simp{:}ProcCFG.valid\text{-}path\text{-}def$)
moreover
from ‹$\forall\, a \in set\ as'.\ intra\text{-}kind\ (kind\ a)$›
have $CFG.valid\text{-}path\ kind\ (get\text{-}return\text{-}edges\ wfp)$
$([((p,Exit),(\lambda cf.\ snd\ cf = (Main,Label\ lx'))\hookleftarrow_p$
$(\lambda cf\ cf'.\ cf'(rets\ [:=]\ map\ cf\ outs)),(Main,Label\ lx'))]@as'$)
by($fastforce\ intro{:}ProcCFG.valid\text{-}path\text{-}same\text{-}level\text{-}path\text{-}Append$
$ProcCFG.intras\text{-}same\text{-}level\text{-}path\ simp{:}ProcCFG.valid\text{-}path\text{-}def$)
ultimately show $?case$ **by**($fastforce\ intro{:}ProcCFG.path\text{-}Append\ simp{:}ProcCFG.vp\text{-}def$)
next
case $(snoc\ p'\ ps')$
note $IH = ‹\bigwedge p\ ins\ outs\ c.$
$[\![containsCall\ procs\ prog\ ps'\ p;\ (p,ins,outs,c) \in set\ procs]\!]$
$\Longrightarrow \exists\, as\ as'.\ CFG.valid\text{-}path'\ sourcenode\ targetnode\ kind\ (valid\text{-}edge\ wfp)$
$(get\text{-}return\text{-}edges\ wfp)\ (Main,\ Entry)\ as\ (p,\ Entry) \wedge$
$CFG.valid\text{-}path'\ sourcenode\ targetnode\ kind\ (valid\text{-}edge\ wfp)$
$(get\text{-}return\text{-}edges\ wfp)\ (p,\ Exit)\ as'\ (Main,\ Exit)$›
from ‹$containsCall\ procs\ prog\ (ps' @ [p'])\ p$›
obtain $ins'\ outs'\ c'$ **where** $(p',ins',outs',c') \in set\ procs$
and $containsCall\ procs\ c'\ []\ p$
and $containsCall\ procs\ prog\ ps'\ p'$ **by**($auto\ elim{:}containsCallE$)
from $IH[OF$ ‹$containsCall\ procs\ prog\ ps'\ p'$› ‹$(p',ins',outs',c') \in set\ procs$›$]$
obtain $as\ as'$ **where** $pathE{:}CFG.valid\text{-}path'\ sourcenode\ targetnode\ kind$
$(valid\text{-}edge\ wfp)\ (get\text{-}return\text{-}edges\ wfp)\ (Main,\ Entry)\ as\ (p',\ Entry)$
and $pathX{:}CFG.valid\text{-}path'\ sourcenode\ targetnode\ kind\ (valid\text{-}edge\ wfp)$
$(get\text{-}return\text{-}edges\ wfp)\ (p',\ Exit)\ as'\ (Main,\ Exit)$ **by** $blast$
from ‹$containsCall\ procs\ c'\ []\ p$›
obtain $lx\ es\ rets\ lx'$ **where** $edge{:}c' \vdash Label\ lx -CEdge\ (p,es,rets)\rightarrow_p Label\ lx'$
by($erule\ containsCall\text{-}empty\text{-}Proc\text{-}CFG\text{-}Call\text{-}edge$)
hence $lx < \#{:}c'$ **and** $lx' < \#{:}c'$
by($auto\ intro{:}Proc\text{-}CFG\text{-}sourcelabel\text{-}less\text{-}num\text{-}nodes$
$Proc\text{-}CFG\text{-}targetlabel\text{-}less\text{-}num\text{-}nodes$)
from ‹$lx < \#{:}c'$› ‹$Rep\text{-}wf\text{-}prog\ wfp = (prog,procs)$› ‹$(p',ins',outs',c') \in set\ procs$›
‹$containsCall\ procs\ prog\ ps'\ p'$› **obtain** asx
where $wfp \vdash (p',Entry) -asx\rightarrow* (p',Label\ lx)$
and $\forall\, a \in set\ asx.\ intra\text{-}kind\ (kind\ a)$

by(*fastforce elim*:*Label-Proc-CFG-Entry-Exit-path-Proc*)
with *pathE* **have** *pathE2*:*CFG.valid-path' sourcenode targetnode kind*
(*valid-edge wfp*) (*get-return-edges wfp*) (*Main*, *Entry*) (*as*@*asx*) (*p'*, *Label lx*)
by(*fastforce intro*:*ProcCFG.path-Append ProcCFG.valid-path-same-level-path-Append*
ProcCFG.intras-same-level-path simp:*ProcCFG.vp-def*)
from ‹*lx'* < #:*c'*› ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
‹(*p'*,*ins'*,*outs'*,*c'*) ∈ *set procs*› ‹*containsCall procs prog ps' p'*›
obtain *asx'* **where** *wfp* ⊢ (*p'*,*Label lx'*) −*asx'*→∗ (*p'*,*Exit*)
and ∀ *a* ∈ *set asx'*. *intra-kind* (*kind a*)
by(*fastforce elim*:*Label-Proc-CFG-Entry-Exit-path-Proc*)
with *pathX* **have** *pathX2*:*CFG.valid-path' sourcenode targetnode kind*
(*valid-edge wfp*) (*get-return-edges wfp*) (*p'*, *Label lx'*) (*asx'*@*as'*) (*Main*, *Exit*)
by(*fastforce intro*:*ProcCFG.path-Append ProcCFG.same-level-path-valid-path-Append*
ProcCFG.intras-same-level-path simp:*ProcCFG.vp-def*)
from *edge* ‹(*p*,*ins*,*outs*,*c*) ∈ *set procs*› ‹(*p'*,*ins'*,*outs'*,*c'*) ∈ *set procs*›
‹*containsCall procs prog ps' p'*›
have *prog*,*procs* ⊢ (*p'*,*Label lx*) −(λ*s*. *True*):(*p'*,*Label lx'*)↪*p*
map (λ*e cf*. *interpret e cf*) *es*→ (*p*,*Entry*)
and *prog*,*procs* ⊢ (*p*,*Exit*) −(λ*cf*. *snd cf* = (*p'*,*Label lx'*))↩*p*
(λ*cf cf'*. *cf'*(*rets* [:=] *map cf outs*))→ (*p'*,*Label lx'*)
by(*fastforce intro*:*ProcCall ProcReturn*)+
with ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
have *path*:*wfp* ⊢ (*p'*,*Label lx*) −[((*p'*,*Label lx*),(λ*s*. *True*):(*p'*,*Label lx'*)↪*p*
map (λ*e cf*. *interpret e cf*) *es*,(*p*,*Entry*))]→∗ (*p*,*Entry*)
and *path'*:*wfp* ⊢ (*p*,*Exit*) −[((*p*,*Exit*),(λ*cf*. *snd cf* = (*p'*,*Label lx'*))↩*p*
(λ*cf cf'*. *cf'*(*rets* [:=] *map cf outs*)),(*p'*,*Label lx'*))]→∗
(*p'*,*Label lx'*)
by(*fastforce intro*:*ProcCFG.path.intros*
simp:*ProcCFG.valid-node-def valid-edge-def*)+
from *path pathE2* **have** *CFG.valid-path' sourcenode targetnode kind* (*valid-edge wfp*)
(*get-return-edges wfp*) (*Main*, *Entry*) ((*as*@*asx*)@[((*p'*,*Label lx*),
(λ*s*. *True*):(*p'*,*Label lx'*)↪*p map* (λ*e cf*. *interpret e cf*) *es*,(*p*,*Entry*))])
(*p*,*Entry*)
apply(*unfold ProcCFG.vp-def*) **apply**(*rule conjI*)
apply(*fastforce intro*:*ProcCFG.path-Append*)
by(*unfold ProcCFG.valid-path-def*,*fastforce intro*:*ProcCFG.vpa-snoc-Call*)
moreover
from *path' pathX2* **have** *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge wfp*) (*get-return-edges wfp*) (*p*,*Exit*)
([((*p*,*Exit*),(λ*cf*. *snd cf* = (*p'*,*Label lx'*))↩*p*
(λ*cf cf'*. *cf'*(*rets* [:=] *map cf outs*)),(*p'*,*Label lx'*))]@(*asx'*@*as'*)) (*Main*, *Exit*)
apply(*unfold ProcCFG.vp-def*) **apply**(*rule conjI*)
apply(*fastforce intro*:*ProcCFG.path-Append*)
by(*simp add*:*ProcCFG.valid-path-def ProcCFG.valid-path-def*)
ultimately show *?case* **by** *blast*
qed
qed

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 −
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› **have** *wf*:*well-formed procs*
by(*fastforce intro*:*wf-wf-prog*)
from ‹*prog*,*procs* ⊢ *sourcenode a* −*kind a*→ *targetnode a*›
show *?thesis*
proof(*induct sourcenode a kind a targetnode a rule*:*PCFG.induct*)
case (*Main nx nx'*)
from ‹(*Main*, *nx*) = *sourcenode a*›[*THEN sym*] ‹(*Main*, *nx'*) = *targetnode a*›[*THEN sym*]
disj **have** [*simp*]:*p* = *Main* **by** *auto*
have *prog*,*procs* ⊢ (*Main*, *Entry*) −(λ*s*. *False*)$_{\surd}$→ (*Main*, *Exit*)
by(*fastforce intro*:*PCFG.Main Proc-CFG-Entry-Exit*)
hence *EXpath*:*wfp* ⊢ (*Main*,*Entry*) −[((*Main*,*Entry*),(λ*s*. *False*)$_{\surd}$,(*Main*,*Exit*))]→∗ (*Main*,*Exit*)
by(*fastforce intro*:*ProcCFG.path.intros simp*:*valid-edge-def ProcCFG.valid-node-def*)
show *?case*
proof(*cases n*)
case (*Label l*)
with ‹*prog* ⊢ *nx* −*IEdge* (*kind a*)→$_p$ *nx'*› ‹(*Main*, *nx*) = *sourcenode a*›[*THEN sym*]
‹(*Main*, *nx'*) = *targetnode a*›[*THEN sym*] *disj*
have *l* < #:*prog* **by**(*auto intro*:*Proc-CFG-sourcelabel-less-num-nodes Proc-CFG-targetlabel-less-num-nodes*)
with ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
obtain *as as'* **where** *wfp* ⊢ (*Main*,*Entry*) −*as*→∗ (*Main*,*Label l*)
and ∀ *a* ∈ *set as*. *intra-kind* (*kind a*)
and *wfp* ⊢ (*Main*,*Label l*) −*as'*→∗ (*Main*,*Exit*)
and ∀ *a* ∈ *set as'*. *intra-kind* (*kind a*)
by −(*erule Label-Proc-CFG-Entry-Exit-path-Main*)+
with *Label* **show** *?thesis*
apply(*rule-tac x*=*as* **in** *exI*) **apply**(*rule-tac x*=*as'* **in** *exI*) **apply** *simp*
by(*fastforce intro*:*ProcCFG.intra-path-vp simp*:*ProcCFG.intra-path-def*)
next
case *Entry*
hence *wfp* ⊢ (*Main*,*Entry*) −[]→∗ (*Main*,*n*) **by**(*fastforce intro*:*ProcCFG.empty-path*)
with *EXpath* **show** *?thesis* **by**(*fastforce simp*:*ProcCFG.vp-def ProcCFG.valid-path-def*)
next
case *Exit*
hence *wfp* ⊢ (*Main*,*n*) −[]→∗ (*Main*,*Exit*) **by**(*fastforce intro*:*ProcCFG.empty-path*)

```
    with Exit EXpath show ?thesis using Exit
     apply(rule-tac x=[((Main,Entry),(λs. False)√,(Main,Exit))] in exI)
     apply simp
     by(fastforce intro:ProcCFG.intra-path-vp
       simp:ProcCFG.intra-path-def intra-kind-def)
   qed
  next
   case (Proc px ins outs c nx nx' ps)
   from ‹(px, ins, outs, c) ∈ set procs› wf have [simp]:px ≠ Main by auto
    from disj ‹(px, nx) = sourcenode a›[THEN sym] ‹(px, nx') = targetnode
a›[THEN sym]
   have [simp]:p = px by auto
   from ‹Rep-wf-prog wfp = (prog,procs)›
    ‹containsCall procs prog ps px› ‹(px, ins, outs, c) ∈ set procs›
   obtain asx asx' where path:CFG.valid-path' sourcenode targetnode kind
    (valid-edge wfp) (get-return-edges wfp) (Main,Entry) asx (px,Entry)
    and path':CFG.valid-path' sourcenode targetnode kind
    (valid-edge wfp) (get-return-edges wfp) (px,Exit) asx' (Main,Exit)
    by −(erule Entry-to-Entry-and-Exit-to-Exit)+
   from ‹containsCall procs prog ps px› ‹(px, ins, outs, c) ∈ set procs›
   have prog,procs ⊢ (px, Entry) −(λs. False)√→ (px, Exit)
    by(fastforce intro:PCFG.Proc Proc-CFG-Entry-Exit)
   hence EXpath:wfp ⊢ (px,Entry) −[((px,Entry),(λs. False)√,(px,Exit))]→∗
    (px,Exit) by(fastforce intro:ProcCFG.path.intros
     simp:valid-edge-def ProcCFG.valid-node-def)
   show ?case
   proof(cases n)
    case (Label l)
    with ‹c ⊢ nx −IEdge (kind a)→p nx'› disj ‹(px, nx) = sourcenode a›[THEN
sym]
     ‹(px, nx') = targetnode a›[THEN sym]
    have l < #:c by(auto intro:Proc-CFG-sourcelabel-less-num-nodes
     Proc-CFG-targetlabel-less-num-nodes)
    with ‹Rep-wf-prog wfp = (prog,procs)› ‹(px, ins, outs, c) ∈ set procs›
     ‹containsCall procs prog ps px›
    obtain as as' where wfp ⊢ (px,Entry) −as→∗ (px,Label l)
     and ∀ a ∈ set as. intra-kind (kind a)
     and wfp ⊢ (px,Label l) −as'→∗ (px,Exit)
     and ∀ a ∈ set as'. intra-kind (kind a)
     by −(erule Label-Proc-CFG-Entry-Exit-path-Proc)+
    with path path' show ?thesis using Label
     apply(rule-tac x=asx@as in exI) apply(rule-tac x=as'@asx' in exI)
    by(auto intro:ProcCFG.path-Append ProcCFG.valid-path-same-level-path-Append
     ProcCFG.same-level-path-valid-path-Append ProcCFG.intras-same-level-path
      simp:ProcCFG.vp-def)
   next
    case Entry
    from EXpath path' have CFG.valid-path' sourcenode targetnode kind
     (valid-edge wfp) (get-return-edges wfp) (px,Entry)
```

```
    ([((px,Entry),(λs. False)√,(px,Exit))]@asx') (Main, Exit)
    apply(unfold ProcCFG.vp-def) apply(erule conjE) apply(rule conjI)
    by(fastforce intro:ProcCFG.path-Append
    ProcCFG.same-level-path-valid-path-Append ProcCFG.intras-same-level-path
      simp:intra-kind-def)+
  with path Entry show ?thesis by simp blast
 next
  case Exit
  with path EXpath path' show ?thesis
   apply(rule-tac x=asx@[((px,Entry),(λs. False)√,(px,Exit))] in exI)
   apply simp
   by(fastforce intro:ProcCFG.path-Append
   ProcCFG.valid-path-same-level-path-Append ProcCFG.intras-same-level-path
     simp:ProcCFG.vp-def ProcCFG.intra-path-def intra-kind-def)
 qed
next
 case (MainCall l px es rets nx' ins outs c)
 from disj show ?case
 proof
  assume (p,n) = sourcenode a
  with ‹(Main, Label l) = sourcenode a›[THEN sym]
  have [simp]:n = Label l p = Main by simp-all
  with ‹prog ⊢ Label l −CEdge (px, es, rets)→p nx'› have l < #:prog
   by(fastforce intro:Proc-CFG-sourcelabel-less-num-nodes)
  with ‹Rep-wf-prog wfp = (prog,procs)›
  obtain as as' where wfp ⊢ (Main,Entry) −as→∗ (Main,Label l)
   and ∀ a ∈ set as. intra-kind (kind a)
   and wfp ⊢ (Main,Label l) −as'→∗ (Main,Exit)
   and ∀ a ∈ set as'. intra-kind (kind a)
   by −(erule Label-Proc-CFG-Entry-Exit-path-Main)+
  thus ?thesis
   by(fastforce intro:ProcCFG.intra-path-vp simp:ProcCFG.intra-path-def)
 next
  assume (p,n) = targetnode a
  with ‹(px, Entry) = targetnode a›[THEN sym]
  have [simp]:n = Entry p = px by simp-all
  from ‹prog ⊢ Label l −CEdge (px, es, rets)→p nx'›
  have containsCall procs prog [] px
   by(rule Proc-CFG-Call-containsCall)
  with ‹Rep-wf-prog wfp = (prog,procs)› ‹(px, ins, outs, c) ∈ set procs›
  obtain as' where Xpath:CFG.valid-path' sourcenode targetnode kind
   (valid-edge wfp) (get-return-edges wfp) (px,Exit) as' (Main,Exit)
   by −(erule Entry-to-Entry-and-Exit-to-Exit)
  from ‹containsCall procs prog [] px› ‹(px, ins, outs, c) ∈ set procs›
  have prog,procs ⊢ (px, Entry) −(λs. False)√→ (px, Exit)
   by(fastforce intro:PCFG.Proc Proc-CFG-Entry-Exit)
  hence wfp ⊢ (px,Entry) −[((px,Entry),(λs. False)√,(px,Exit))]→∗ (px,Exit)
   by(fastforce intro:ProcCFG.path.intros
     simp:valid-edge-def ProcCFG.valid-node-def)
```

```
    with Xpath have CFG.valid-path' sourcenode targetnode kind
     (valid-edge wfp) (get-return-edges wfp) (px,Entry)
     ([((px,Entry),(λs. False)√,(px,Exit))]@as') (Main,Exit)
     apply(unfold ProcCFG.vp-def) apply(erule conjE) apply(rule conjI)
     by(fastforce intro:ProcCFG.path-Append
     ProcCFG.same-level-path-valid-path-Append ProcCFG.intras-same-level-path
       simp:intra-kind-def)+
    with ‹containsCall procs prog [] px› ‹Rep-wf-prog wfp = (prog,procs)›
     ‹(px, ins, outs, c) ∈ set procs›
    show ?thesis by(fastforce elim:Entry-to-Entry-and-Exit-to-Exit)
  qed
next
  case (ProcCall px ins outs c l p' es' rets' l' ins' outs' c' ps)
  from disj show ?case
  proof
    assume (p,n) = sourcenode a
    with ‹(px, Label l) = sourcenode a›[THEN sym]
    have [simp]:n = Label l p = px by simp-all
    with ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'› have l < #:c
      by(fastforce intro:Proc-CFG-sourcelabel-less-num-nodes)
    from ‹Rep-wf-prog wfp = (prog,procs)› ‹l < #:c›
     ‹containsCall procs prog ps px› ‹(px, ins, outs, c) ∈ set procs›
    obtain as as' where wfp ⊢ (px,Label l) −as→∗ (px,Exit)
      and ∀ a ∈ set as. intra-kind (kind a)
      and wfp ⊢ (px,Entry) −as'→∗ (px,Label l)
      and ∀ a ∈ set as'. intra-kind (kind a)
      by −(erule Label-Proc-CFG-Entry-Exit-path-Proc)+
    moreover
    from ‹Rep-wf-prog wfp = (prog,procs)› ‹containsCall procs prog ps px›
     ‹(px, ins, outs, c) ∈ set procs› obtain asx asx'
      where CFG.valid-path' sourcenode targetnode kind
      (valid-edge wfp) (get-return-edges wfp) (Main,Entry) asx (px,Entry)
      and CFG.valid-path' sourcenode targetnode kind
      (valid-edge wfp) (get-return-edges wfp) (px,Exit) asx' (Main,Exit)
      by −(erule Entry-to-Entry-and-Exit-to-Exit)+
    ultimately show ?thesis
      apply(rule-tac x=asx@as' in exI) apply(rule-tac x=as@asx' in exI)
     by(auto intro:ProcCFG.path-Append ProcCFG.valid-path-same-level-path-Append
      ProcCFG.same-level-path-valid-path-Append ProcCFG.intras-same-level-path
        simp:ProcCFG.vp-def)
  next
    assume (p,n) = targetnode a
    with ‹(p', Entry) = targetnode a›[THEN sym]
    have [simp]:n = Entry p = p' by simp-all
    from ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'›
    have containsCall procs c [] p' by(rule Proc-CFG-Call-containsCall)
    with ‹containsCall procs prog ps px› ‹(px, ins, outs, c) ∈ set procs›
    have containsCall procs prog (ps@[px]) p'
      by(rule containsCall-in-proc)
```

```
    with ‹(p', ins', outs', c') ∈ set procs›
    have prog,procs ⊢ (p', Entry) −(λs. False)√→ (p', Exit)
      by(fastforce intro:PCFG.Proc Proc-CFG-Entry-Exit)
    hence wfp ⊢ (p',Entry) −[((p',Entry),(λs. False)√,(p',Exit))]→* (p',Exit)
      by(fastforce intro:ProcCFG.path.intros
        simp:valid-edge-def ProcCFG.valid-node-def)
    moreover
    from ‹Rep-wf-prog wfp = (prog,procs)› ‹(p', ins', outs', c') ∈ set procs›
      ‹containsCall procs prog (ps@[px]) p'›
    obtain 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)
      by −(erule Entry-to-Entry-and-Exit-to-Exit)+
    ultimately show ?thesis
      apply(rule-tac x=as in exI)
      apply(rule-tac x=[((p',Entry),(λs. False)√,(p',Exit))]@as' in exI)
      apply(unfold ProcCFG.vp-def)
      by(fastforce intro:ProcCFG.path-Append
     ProcCFG.same-level-path-valid-path-Append ProcCFG.intras-same-level-path
        simp:intra-kind-def)+
  qed
next
  case (MainReturn l px es rets l' ins outs c)
  from disj show ?case
  proof
    assume (p,n) = sourcenode a
    with ‹(px, Exit) = sourcenode a›[THEN sym]
    have [simp]:n = Exit p = px by simp-all
    from ‹prog ⊢ Label l −CEdge (px, es, rets)→p Label l'›
    have containsCall procs prog [] px by(rule Proc-CFG-Call-containsCall)
    with ‹(px, ins, outs, c) ∈ set procs›
    have prog,procs ⊢ (px, Entry) −(λs. False)√→ (px, Exit)
      by(fastforce intro:PCFG.Proc Proc-CFG-Entry-Exit)
    hence wfp ⊢ (px,Entry) −[((px,Entry),(λs. False)√,(px,Exit))]→* (px,Exit)
      by(fastforce intro:ProcCFG.path.intros
        simp:valid-edge-def ProcCFG.valid-node-def)
    moreover
    from ‹Rep-wf-prog wfp = (prog,procs)› ‹(px, ins, outs, c) ∈ set procs›
      ‹containsCall procs prog [] px›
    obtain as as' where CFG.valid-path' sourcenode targetnode kind
      (valid-edge wfp) (get-return-edges wfp) (Main,Entry) as (px,Entry)
      and CFG.valid-path' sourcenode targetnode kind
      (valid-edge wfp) (get-return-edges wfp) (px,Exit) as' (Main,Exit)
      by −(erule Entry-to-Entry-and-Exit-to-Exit)+
    ultimately show ?thesis
      apply(rule-tac x=as@[((px,Entry),(λs. False)√,(px,Exit))] in exI)
      apply(rule-tac x=as' in exI)
      apply(unfold ProcCFG.vp-def)
```

```
    by(fastforce intro:ProcCFG.path-Append
    ProcCFG.valid-path-same-level-path-Append ProcCFG.intras-same-level-path
      simp:intra-kind-def)+
  next
    assume (p, n) = targetnode a
    with ‹(Main, Label l') = targetnode a›[THEN sym]
    have [simp]:n = Label l' p = Main by simp-all
    with ‹prog ⊢ Label l −CEdge (px, es, rets)→p Label l'› have l' < #:prog
      by(fastforce intro:Proc-CFG-targetlabel-less-num-nodes)
    with ‹Rep-wf-prog wfp = (prog,procs)›
    obtain as as' where wfp ⊢ (Main,Entry) −as→* (Main,Label l')
      and ∀ a ∈ set as. intra-kind (kind a)
      and wfp ⊢ (Main,Label l') −as'→* (Main,Exit)
      and ∀ a ∈ set as'. intra-kind (kind a)
      by −(erule Label-Proc-CFG-Entry-Exit-path-Main)+
    thus ?thesis
      by(fastforce intro:ProcCFG.intra-path-vp simp:ProcCFG.intra-path-def)
  qed
next
  case (ProcReturn px ins outs c l p' es' rets' l' ins' outs' c' ps)
  from disj show ?case
  proof
    assume (p,n) = sourcenode a
    with ‹(p', Exit) = sourcenode a›[THEN sym]
    have [simp]:n = Exit p = p' by simp-all
    from ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'›
    have containsCall procs c [] p' by(rule Proc-CFG-Call-containsCall)
    with ‹containsCall procs prog ps px› ‹(px, ins, outs, c) ∈ set procs›
    have containsCall procs prog (ps@[px]) p'
      by(rule containsCall-in-proc)
    with ‹(p', ins', outs', c') ∈ set procs›
    have prog,procs ⊢ (p', Entry) −(λs. False)√→ (p', Exit)
      by(fastforce intro:PCFG.Proc Proc-CFG-Entry-Exit)
    hence wfp ⊢ (p',Entry) −[((p',Entry),(λs. False)√,(p',Exit))]→* (p',Exit)
      by(fastforce intro:ProcCFG.path.intros
        simp:valid-edge-def ProcCFG.valid-node-def)
    moreover
    from ‹Rep-wf-prog wfp = (prog,procs)› ‹(p', ins', outs', c') ∈ set procs›
      ‹containsCall procs prog (ps@[px]) p'›
    obtain 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)
      by −(erule Entry-to-Entry-and-Exit-to-Exit)+
    ultimately show ?thesis
      apply(rule-tac x=as@[((p',Entry),(λs. False)√,(p',Exit))] in exI)
      apply(rule-tac x=as' in exI)
      apply(unfold ProcCFG.vp-def)
      by(fastforce intro:ProcCFG.path-Append
```

```
      ProcCFG.valid-path-same-level-path-Append ProcCFG.intras-same-level-path
        simp:intra-kind-def)+
    next
      assume (p, n) = targetnode a
      with ‹(px, Label l') = targetnode a›[THEN sym]
      have [simp]:n = Label l' p = px by simp-all
      with ‹c ⊢ Label l −CEdge (p', es', rets')→p Label l'› have l' < #:c
        by(fastforce intro:Proc-CFG-targetlabel-less-num-nodes)
      from ‹Rep-wf-prog wfp = (prog,procs)› ‹l' < #:c›
        ‹containsCall procs prog ps px› ‹(px, ins, outs, c) ∈ set procs›
      obtain as as' where wfp ⊢ (px,Label l') −as→∗ (px,Exit)
        and ∀ a ∈ set as. intra-kind (kind a)
        and wfp ⊢ (px,Entry) −as'→∗ (px,Label l')
        and ∀ a ∈ set as'. intra-kind (kind a)
        by −(erule Label-Proc-CFG-Entry-Exit-path-Proc)+
      moreover
      from ‹Rep-wf-prog wfp = (prog,procs)› ‹containsCall procs prog ps px›
        ‹(px, ins, outs, c) ∈ set procs› obtain asx asx'
        where CFG.valid-path' sourcenode targetnode kind
        (valid-edge wfp) (get-return-edges wfp) (Main,Entry) asx (px,Entry)
        and CFG.valid-path' sourcenode targetnode kind
        (valid-edge wfp) (get-return-edges wfp) (px,Exit) asx' (Main,Exit)
        by −(erule Entry-to-Entry-and-Exit-to-Exit)+
      ultimately show ?thesis
        apply(rule-tac x=asx@as' in exI) apply(rule-tac x=as@asx' in exI)
      by(auto intro:ProcCFG.path-Append ProcCFG.valid-path-same-level-path-Append
        ProcCFG.same-level-path-valid-path-Append ProcCFG.intras-same-level-path
        simp:ProcCFG.vp-def)
    qed
  next
    case (MainCallReturn nx px es rets nx')
    from ‹prog ⊢ nx −CEdge (px, es, rets)→p nx'› disj
      ‹(Main, nx) = sourcenode a›[THEN sym] ‹(Main, nx') = targetnode a›[THEN
sym]
    obtain l where [simp]:n = Label l p = Main
      by(fastforce dest:Proc-CFG-Call-Labels)
    from ‹prog ⊢ nx −CEdge (px, es, rets)→p nx'› disj
      ‹(Main, nx) = sourcenode a›[THEN sym] ‹(Main, nx') = targetnode a›[THEN
sym]
    have l < #:prog by(auto intro:Proc-CFG-sourcelabel-less-num-nodes
      Proc-CFG-targetlabel-less-num-nodes)
    with ‹Rep-wf-prog wfp = (prog,procs)›
    obtain as as' where wfp ⊢ (Main,Entry) −as→∗ (Main,Label l)
      and ∀ a ∈ set as. intra-kind (kind a)
      and wfp ⊢ (Main,Label l) −as'→∗ (Main,Exit)
      and ∀ a ∈ set as'. intra-kind (kind a)
      by −(erule Label-Proc-CFG-Entry-Exit-path-Main)+
    thus ?thesis
      apply(rule-tac x=as in exI) apply(rule-tac x=as' in exI) apply simp
```

by(*fastforce intro*:*ProcCFG.intra-path-vp simp*:*ProcCFG.intra-path-def*)
next
case (*ProcCallReturn px ins outs c nx p′ es′ rets′ nx′ ps*)
from ‹(*px*, *ins*, *outs*, *c*) ∈ *set procs*› *wf* **have** [*simp*]:*px* ≠ *Main* **by** *auto*
from ‹*c* ⊢ *nx* −*CEdge* (*p′*, *es′*, *rets′*)→$_p$ *nx′*› *disj*
‹(*px*, *nx*) = *sourcenode a*›[*THEN sym*] ‹(*px*, *nx′*) = *targetnode a*›[*THEN sym*]
obtain *l* **where** [*simp*]:*n* = *Label l p* = *px*
by(*fastforce dest*:*Proc-CFG-Call-Labels*)
from ‹*c* ⊢ *nx* −*CEdge* (*p′*, *es′*, *rets′*)→$_p$ *nx′*› *disj*
‹(*px*, *nx*) = *sourcenode a*›[*THEN sym*] ‹(*px*, *nx′*) = *targetnode a*›[*THEN sym*]
have *l* < #:*c*
by(*auto intro*:*Proc-CFG-sourcelabel-less-num-nodes*
Proc-CFG-targetlabel-less-num-nodes)
with ‹*Rep-wf-prog wfp* = (*prog*,*procs*)› ‹(*px*, *ins*, *outs*, *c*) ∈ *set procs*›
‹*containsCall procs prog ps px*›
obtain *as as′* **where** *wfp* ⊢ (*px*,*Entry*) −*as*→∗ (*px*,*Label l*)
and ∀ *a* ∈ *set as*. *intra-kind* (*kind a*)
and *wfp* ⊢ (*px*,*Label l*) −*as′*→∗ (*px*,*Exit*)
and ∀ *a* ∈ *set as′*. *intra-kind* (*kind a*)
by −(*erule Label-Proc-CFG-Entry-Exit-path-Proc*)+
moreover
from ‹*Rep-wf-prog wfp* = (*prog*,*procs*)›
‹*containsCall procs prog ps px*› ‹(*px*, *ins*, *outs*, *c*) ∈ *set procs*›
obtain *asx asx′* **where** *CFG.valid-path′ sourcenode targetnode kind*
(*valid-edge wfp*) (*get-return-edges wfp*) (*Main*,*Entry*) *asx* (*px*,*Entry*)
and *CFG.valid-path′ sourcenode targetnode kind*
(*valid-edge wfp*) (*get-return-edges wfp*) (*px*,*Exit*) *asx′* (*Main*,*Exit*)
by −(*erule Entry-to-Entry-and-Exit-to-Exit*)+
ultimately show *?thesis*
apply(*rule-tac x*=*asx*@*as* **in** *exI*) **apply**(*rule-tac x*=*as′*@*asx′* **in** *exI*)
by(*auto intro*:*ProcCFG.path-Append ProcCFG.valid-path-same-level-path-Append*
ProcCFG.same-level-path-valid-path-Append ProcCFG.intras-same-level-path
simp:*ProcCFG.vp-def*)
qed
qed

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 −
from *Rep-wf-prog*[*of wfp*]
obtain *prog procs* **where** [*simp*]:*Rep-wf-prog wfp* = (*prog*,*procs*)
by(*fastforce simp*:*wf-prog-def*)
hence *wf*:*well-formed procs* **by**(*fastforce intro*:*wf-wf-prog*)
show *Postdomination sourcenode targetnode kind* (*valid-edge wfp*)
(*Main*, *Entry*) *get-proc* (*get-return-edges wfp*) (*lift-procs wfp*) *Main* (*Main*, *Exit*)

```
  proof
    fix m
    assume CFG.valid-node sourcenode targetnode (valid-edge wfp) m
    then obtain a where valid-edge wfp a
      and m = sourcenode a ∨ m = targetnode a
      by(fastforce simp:ProcCFG.valid-node-def)
    obtain p n where [simp]:m = (p,n) by(cases m) auto
    from ‹valid-edge wfp a› ‹m = sourcenode a ∨ m = targetnode a›
      ‹Rep-wf-prog wfp = (prog,procs)›
    show ∃ as. CFG.valid-path' sourcenode targetnode kind (valid-edge wfp)
      (get-return-edges wfp) (Main, Entry) as m
      by(auto dest!:edge-valid-paths simp:valid-edge-def)
  next
    fix m
    assume CFG.valid-node sourcenode targetnode (valid-edge wfp) m
    then obtain a where valid-edge wfp a
      and m = sourcenode a ∨ m = targetnode a
      by(fastforce simp:ProcCFG.valid-node-def)
    obtain p n where [simp]:m = (p,n) by(cases m) auto
    from ‹valid-edge wfp a› ‹m = sourcenode a ∨ m = targetnode a›
      ‹Rep-wf-prog wfp = (prog,procs)›
    show ∃ as. CFG.valid-path' sourcenode targetnode kind (valid-edge wfp)
      (get-return-edges wfp) m as (Main,Exit)
      by(auto dest!:edge-valid-paths simp:valid-edge-def)
  next
    fix n n'
   assume mex1:CFGExit.method-exit sourcenode kind (valid-edge wfp) (Main,Exit)
n
     and mex2:CFGExit.method-exit sourcenode kind (valid-edge wfp) (Main,Exit)
n'
     and get-proc n = get-proc n'
    from mex1
    have n = (Main,Exit) ∨ (∃ a Q p f. n = sourcenode a ∧ valid-edge wfp a ∧
      kind a = Q↩pf) by(simp add:ProcCFGExit.method-exit-def)
    thus n = n'
    proof
      assume n = (Main,Exit)
      from mex2 have n' = (Main,Exit) ∨ (∃ a Q p f. n' = sourcenode a ∧
        valid-edge wfp a ∧ kind a = Q↩pf)
        by(simp add:ProcCFGExit.method-exit-def)
      thus ?thesis
      proof
        assume n' = (Main,Exit)
        with ‹n = (Main,Exit)› show ?thesis by simp
      next
        assume ∃ a Q p f. n' = sourcenode a ∧
          valid-edge wfp a ∧ kind a = Q↩pf
        then obtain a Q p f where n' = sourcenode a
          and valid-edge wfp a and kind a = Q↩pf by blast
```

from ‹*valid-edge wfp a*› ‹*kind a* = $Q\hookleftarrow_p f$›
have *get-proc* (*sourcenode a*) = *p* **by**(*rule ProcCFG.get-proc-return*)
with ‹*get-proc n* = *get-proc n'*› ‹*n* = (*Main*,*Exit*)› ‹*n'* = *sourcenode a*›
have *get-proc* (*Main*,*Exit*) = *p* **by** *simp*
hence *p* = *Main* **by** *simp*
with ‹*kind a* = $Q\hookleftarrow_p f$› **have** *kind a* = $Q\hookleftarrow_{Main} f$ **by** *simp*
with ‹*valid-edge wfp a*› **have** *False* **by**(*rule ProcCFG.Main-no-return-source*)
thus *?thesis* **by** *simp*
qed
next
assume ∃ *a Q p f*. *n* = *sourcenode a* ∧
valid-edge wfp a ∧ *kind a* = $Q\hookleftarrow_p f$
then obtain *a Q p f* **where** *n* = *sourcenode a*
and *valid-edge wfp a* **and** *kind a* = $Q\hookleftarrow_p f$ **by** *blast*
from ‹*valid-edge wfp a*› ‹*kind a* = $Q\hookleftarrow_p f$›
have *get-proc* (*sourcenode a*) = *p* **by**(*rule ProcCFG.get-proc-return*)
from *mex2* **have** *n'* = (*Main*,*Exit*) ∨ (∃ *a Q p f*. *n'* = *sourcenode a* ∧
valid-edge wfp a ∧ *kind a* = $Q\hookleftarrow_p f$)
by(*simp add*:*ProcCFGExit.method-exit-def*)
thus *?thesis*
proof
assume *n'* = (*Main*,*Exit*)
from ‹*get-proc* (*sourcenode a*) = *p*› ‹*get-proc n* = *get-proc n'*›
‹*n'* = (*Main*,*Exit*)› ‹*n* = *sourcenode a*›
have *get-proc* (*Main*,*Exit*) = *p* **by** *simp*
hence *p* = *Main* **by** *simp*
with ‹*kind a* = $Q\hookleftarrow_p f$› **have** *kind a* = $Q\hookleftarrow_{Main} f$ **by** *simp*
with ‹*valid-edge wfp a*› **have** *False* **by**(*rule ProcCFG.Main-no-return-source*)
thus *?thesis* **by** *simp*
next
assume ∃ *a Q p f*. *n'* = *sourcenode a* ∧
valid-edge wfp a ∧ *kind a* = $Q\hookleftarrow_p f$
then obtain *a' Q' p' f'* **where** *n'* = *sourcenode a'*
and *valid-edge wfp a'* **and** *kind a'* = $Q'\hookleftarrow_{p'} f'$ **by** *blast*
from ‹*valid-edge wfp a'*› ‹*kind a'* = $Q'\hookleftarrow_{p'} f'$›
have *get-proc* (*sourcenode a'*) = *p'* **by**(*rule ProcCFG.get-proc-return*)
with ‹*get-proc n* = *get-proc n'*› ‹*get-proc* (*sourcenode a*) = *p*›
‹*n* = *sourcenode a*› ‹*n'* = *sourcenode a'*›
have *p'* = *p* **by** *simp*
from ‹*valid-edge wfp a*› ‹*kind a* = $Q\hookleftarrow_p f$›
have *sourcenode a* = (*p*,*Exit*) **by**(*auto elim*:*PCFG.cases simp*:*valid-edge-def*)
from ‹*valid-edge wfp a'*› ‹*kind a'* = $Q'\hookleftarrow_{p'} f'$›
have *sourcenode a'* = (*p'*,*Exit*) **by**(*auto elim*:*PCFG.cases simp*:*valid-edge-def*)
with ‹*n* = *sourcenode a*› ‹*n'* = *sourcenode a'*› ‹*p'* = *p*›
‹*sourcenode a* = (*p*,*Exit*)› **show** *?thesis* **by** *simp*
qed
qed
qed
qed

end

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* **..**

end

Chapter 3

A Control Flow Graph for Jinja Byte Code

3.1 Formalizing the CFG

theory *JVMCFG* **imports** *../StaticInter/BasicDefs ../../Jinja/BV/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
  show (*E*, *Phi*) ∈ *wf-jvmprog*
    unfolding *wf-jvmprog-def* **by** (*auto intro*: *wf-prog*)
qed

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*)
using *Rep-wf-jvmprog* [*of P*]
  by (*auto simp*: *wf-jvmprog-def split-beta*)

lemma *wf-jvmprog-is-wf*: *wf-jvm-prog* (*PROG P*)
using *wf-jvmprog-is-wf-typ* **unfolding** *wf-jvm-prog-def*
by *blast*

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 −
have ¬ *finite* (*UNIV* :: *cname set*)
by (*rule infinite-UNIV-listI*)
hence ∃ *C*. *C* ∉ *set* (*map fst P*)
by −(*rule ex-new-if-finite*, *auto*)
then obtain *C* **where** *C* ∉ *set* (*map fst P*)
by *blast*
have ¬ *is-class P C*
proof
assume *is-class P C*
then obtain *D fs ms* **where** *class P C* = ⌊(*D*, *fs*, *ms*)⌋
by *auto*
with ‹*C* ∉ *set* (*map fst P*)› **show** *False*
by (*auto dest*: *map-of-SomeD intro*!: *image-eqI simp*: *class-def*)
qed
thus *?thesis*
by *blast*
qed

lemma *ClassMain-unique-in-P*:
assumes *is-class* (*PROG P*) *C*
shows *ClassMain P* ≠ *C*
proof −
from *ex-new-class-name* [*of PROG P*] **obtain** *D* **where** ¬ *is-class* (*PROG P*)
D
by *blast*
with ‹*is-class* (*PROG P*) *C*› **show** *?thesis*
unfolding *ClassMain-def*
by −(*rule someI2*, *fastforce*+)
qed

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

by (*induct xs*, *auto*)

lemma *map-of-fstE*: ⟦ *map-of xs a* = ⌊*b*⌋; ∃ *x* ∈ *set xs*. *fst x* = *a* ⟹ *thesis* ⟧ ⟹ *thesis*
by (*induct xs*) (*auto split*: *split-if-asm*)

lemma *ex-unique-method-name*:
∃ *Name*. ∀ *C D fs ms*. *class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋ ⟶ (∀ *m*∈*set ms*. *Name* ≠ *fst m*)
proof −
from *wf-jvmprog-is-wf* [*of P*]
have *distinct-fst* (*PROG P*)
by (*simp add*: *wf-jvm-prog-def wf-jvm-prog-phi-def wf-prog-def*)
hence {*C*. ∃ *D fs ms*. *class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋} = *fst* ‘ *set* (*PROG P*)
by (*fastforce elim*: *map-of-fstE simp*: *class-def intro*: *map-of-SomeI*)
hence *finite* {*C*. ∃ *D fs ms*. *class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋}
by *auto*
moreover have {*ms*. ∃ *C D fs*. *class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋}
= *snd* ‘ *snd* ‘ *the* ‘ (λ*C*. *class* (*PROG P*) *C*) ‘ {*C*. ∃ *D fs ms*. *class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋}
by (*fastforce intro*: *rev-image-eqI map-of-SomeI simp*: *class-def*)
ultimately have *finite* {*ms*. ∃ *C D fs*. *class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋}
by *auto*
moreover have ¬ *finite* (*UNIV* :: *mname set*)
by (*rule infinite-UNIV-listI*)
ultimately
have ∃ *Name*. *Name* ∉ *fst* ‘ (⋃ *ms* ∈ {*ms*. ∃ *C D fs*. *class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋}. *set ms*)
by −(*rule ex-new-if-finite*, *auto*)
thus *?thesis*
by *fastforce*
qed

lemma *MethodMain-unique-in-P*:
assumes *PROG P* ⊢ *D sees M*:*Ts*→*T* = *mb in C*
shows *MethodMain P* ≠ *M*
proof −
from *ex-unique-method-name* [*of P*] **obtain** *M′*
where ⋀*C D fs ms*. *class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋ ⟹ (∀ *m* ∈ *set ms*. *M′* ≠ *fst m*)
by *blast*
with ‹*PROG P* ⊢ *D sees M*:*Ts*→*T* = *mb in C*›
show *?thesis*
unfolding *MethodMain-def*
by −(*rule someI2-ex*, *fastforce*, *fastforce dest*!: *visible-method-exists elim*: *map-of-fstE*)
qed

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

proof (*erule rev-notE*)
from *ex-new-class-name* [*of PROG P*] **obtain** *C* **where** ¬ *is-class* (*PROG P*) *C*
by *blast*
thus ¬ *is-class* (*PROG P*) (*ClassMain P*) **unfolding** *ClassMain-def*
by (*rule someI*)
qed

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

lemma *no-Call-from-ClassMain* [*dest*!]: *PROG P* ⊢ *ClassMain P sees M*:*Ts*→*T* = *mb in C* ⟹ *False*
by (*fastforce dest*: *sees-method-is-class*)

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

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*)

$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ None,\ Enter)\ -(\lambda s.\ True)_{\surd}\rightarrow\ (C,\ M,\ \lfloor 0 \rfloor,\ Enter)$
| *CFG-Load*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Load\ n;$
$ek = \Uparrow(\lambda s.\ s(Stack\ (stkLength\ (P,\ C,\ M)\ pc) := s\ (Local\ n)))\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$
| *CFG-Store*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Store\ n;$
$ek = \Uparrow(\lambda s.\ s(Local\ n := s\ (Stack\ (stkLength\ (P,\ C,\ M)\ pc - 1))))\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$
| *CFG-Push*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Push\ v;$
$ek = \Uparrow(\lambda s.\ s(Stack\ (stkLength\ (P,\ C,\ M)\ pc) \mapsto Value\ v))\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$
| *CFG-Pop*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Pop;$
$ek = \Uparrow id\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$
| *CFG-IAdd*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = IAdd;$
$ek = \Uparrow(\lambda s.\ let\ i1 = the\text{-}Intg\ (stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 1));$
$i2 = the\text{-}Intg\ (stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 2))$
$in\ s(Stack\ (stkLength\ (P,\ C,\ M)\ pc - 2) \mapsto Value\ (Intg\ (i1 + i2))))$
$]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$
| *CFG-Goto*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Goto\ i\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -((\lambda s.\ True)_{\surd})\rightarrow\ (C,\ M,\ \lfloor nat\ (int\ pc + i) \rfloor,\ Enter)$
| *CFG-CmpEq*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = CmpEq;$
$ek = \Uparrow(\lambda s.\ let\ e1 = stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 1);$
$e2 = stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 2)$
$in\ s(Stack\ (stkLength\ (P,\ C,\ M)\ pc - 2) \mapsto Value\ (Bool\ (e1 = e2))))$
$]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$
| *CFG-IfFalse-False*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,$
$Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = IfFalse\ i;$
$i \neq 1;$
$ek = (\lambda s.\ stkAt\ s\ (stkLength(P,\ C,\ M)\ pc - 1) = Bool\ False)_{\surd}\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor nat\ (int\ pc + i) \rfloor,$
$Enter)$
| *CFG-IfFalse-True*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = IfFalse\ i;$
$ek = (\lambda s.\ stkAt\ s\ (stkLength(P,\ C,\ M)\ pc - 1) \neq Bool\ False \vee i = 1)_{\surd}\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$
| *CFG-New-Check-Normal*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,$
$Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = New\ Cl;$

$ek = (\lambda s.\ new\text{-}Addr\ (heap\text{-}of\ s) \neq None)_{\surd}\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc \rfloor,\ Normal)$
| *CFG-New-Check-Exceptional*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = New\ Cl;$
$pc' = (case\ (match\text{-}ex\text{-}table\ (PROG\ P)\ OutOfMemory\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M))\ of$
$None \Rightarrow None$
$| Some\ (pc'',\ d) \Rightarrow \lfloor pc'' \rfloor);$
$ek = (\lambda s.\ new\text{-}Addr\ (heap\text{-}of\ s) = None)_{\surd}\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ pc'\ Enter)$
| *CFG-New-Update*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Normal);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = New\ Cl;$
$ek = \Uparrow(\lambda s.\ let\ a = the\ (new\text{-}Addr\ (heap\text{-}of\ s))$
$in\ s(Heap \mapsto Hp\ ((heap\text{-}of\ s)(a \mapsto blank\ (PROG\ P)\ Cl)))$
$(Stack\ (stkLength(P,\ C,\ M)\ pc) \mapsto Value\ (Addr\ a)))\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Normal)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor Suc\ pc \rfloor,\ Enter)$
| *CFG-New-Exceptional-prop*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ None\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = New\ Cl;$
$ek = \Uparrow(\lambda s.\ s(Exception \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ OutOfMemory))))\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ None\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ None,\ Return)$
| *CFG-New-Exceptional-handle*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = New\ Cl;$
$ek = \Uparrow(\lambda s.\ s(Exception := None)$
$(Stack\ (stkLength\ (P,\ C,\ M)\ pc' - 1) \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ OutOfMemory))))\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc' \rfloor,\ Enter)$
| *CFG-Getfield-Check-Normal*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Getfield\ F\ Cl;$
$ek = (\lambda s.\ stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 1) \neq Null)_{\surd}\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc \rfloor,\ Normal)$
| *CFG-Getfield-Check-Exceptional*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow (C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Getfield\ F\ Cl;$
$pc' = (case\ (match\text{-}ex\text{-}table\ (PROG\ P)\ NullPointer\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M))\ of$
$None \Rightarrow None$
$| Some\ (pc'',\ d) \Rightarrow \lfloor pc'' \rfloor);$
$ek = (\lambda s.\ stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 1) = Null)_{\surd}\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ pc'\ Enter)$
| *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 = ↑(λ*s*. *let* (*D*, *fs*) = *the* (*heap-of s* (*the-Addr* (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *1*))))
in s(*Stack* (*stkLength*(*P*, *C*, *M*) *pc* − *1*) ↦ *Value* (*the* (*fs* (*F*, *Cl*)))))
⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Normal*) −(*ek*)→ (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
| *CFG-Getfield-Exceptional-prop*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*);
instrs-of (*PROG P*) *C M* ! *pc* = *Getfield F Cl*;
ek = ↑(λ*s*. *s*(*Exception* ↦ *Value* (*Addr* (*addr-of-sys-xcpt NullPointer*)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*) −(*ek*)→ (*C*, *M*, *None*, *Return*)
| *CFG-Getfield-Exceptional-handle*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc*′⌋ *Enter*);
instrs-of (*PROG P*) *C M* ! *pc* = *Getfield F Cl*;
ek = ↑(λ*s*. *s*(*Exception* := *None*)
(*Stack* (*stkLength* (*P*, *C*, *M*) *pc*′ − *1*) ↦ *Value* (*Addr* (*addr-of-sys-xcpt NullPointer*)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc*′⌋ *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*′⌋, *Enter*)
| *CFG-Putfield-Check-Normal*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);
instrs-of (*PROG P*) *C M* ! *pc* = *Putfield F Cl*;
ek = (λ*s*. *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *2*) ≠ *Null*)√ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*⌋, *Normal*)
| *CFG-Putfield-Check-Exceptional*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);
instrs-of (*PROG P*) *C M* ! *pc* = *Putfield F Cl*;
pc′ = (*case* (*match-ex-table* (*PROG P*) *NullPointer pc* (*ex-table-of* (*PROG P*) *C M*)) *of*
None ⇒ *None*
| *Some* (*pc*″, *d*) ⇒ ⌊*pc*″⌋);
ek = (λ*s*. *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *2*) = *Null*)√ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*⌋, *Exceptional pc*′ *Enter*)
| *CFG-Putfield-Update*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Normal*);
instrs-of (*PROG P*) *C M* ! *pc* = *Putfield F Cl*;
ek = ↑(λ*s*. *let v* = *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *1*);
r = *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *2*);
a = *the-Addr r*;
(*D*, *fs*) = *the* (*heap-of s a*);
h′ = (*heap-of s*)(*a* ↦ (*D*, *fs*((*F*, *Cl*) ↦ *v*)))
in s(*Heap* ↦ *Hp h*′)) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Normal*) −(*ek*)→ (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
| *CFG-Putfield-Exceptional-prop*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *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*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Putfield\ F\ Cl;$
$ek = \Uparrow(\lambda s.\ s(Exception := None)$
$(Stack\ (stkLength\ (P,\ C,\ M)\ pc' - 1) \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer))))\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc' \rfloor,\ Enter)$

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

| *CFG-Checkcast-Check-Exceptional*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Checkcast\ Cl;$
$pc' = (case\ (match\text{-}ex\text{-}table\ (PROG\ P)\ ClassCast\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M))\ of$
$None \Rightarrow None$
$| Some\ (pc'',\ d) \Rightarrow \lfloor pc'' \rfloor);$
$ek = (\lambda s.\ \neg\ cast\text{-}ok\ (PROG\ P)\ Cl\ (heap\text{-}of\ s)\ (stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 1)))_{\surd}\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ pc'\ Enter)$

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

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

| *CFG-Throw-Check*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Throw;$
$pc' = None \vee match\text{-}ex\text{-}table\ (PROG\ P)\ Exc\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M) = \lfloor (the\ pc',\ d) \rfloor;$
$ek = (\lambda s.\ let\ v = stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - 1);$
$Cl = if\ (v = Null)\ then\ NullPointer\ else\ (cname\text{-}of\ (heap\text{-}of\ s)$

(*the-Addr v*))
in case pc′ of
None ⇒ *match-ex-table* (*PROG P*) *Cl pc* (*ex-table-of* (*PROG P*) *C M*) = *None*
| *Some pc″* ⇒ ∃ *d*. *match-ex-table* (*PROG P*) *Cl pc* (*ex-table-of* (*PROG P*) *C M*)
= ⌊(*pc″*, *d*)⌋
)$_\surd$ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*⌋, *Exceptional pc′ Enter*)

| *CFG-Throw-prop*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*);
instrs-of (*PROG P*) *C M* ! *pc* = *Throw*;
ek = ↑(λ*s*. *s*(*Exception* ↦ *Value* (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *1*)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*) −(*ek*)→ (*C*, *M*, *None*, *Return*)
| *CFG-Throw-handle*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*);
pc′ ≠ *length* (*instrs-of* (*PROG P*) *C M*);
instrs-of (*PROG P*) *C M* ! *pc* = *Throw*;
ek = ↑(λ*s*. *s*(*Exception* := *None*)
(*Stack* (*stkLength* (*P*, *C*, *M*) *pc′* − *1*) ↦ *Value* (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *1*)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc′*⌋, *Enter*)
| *CFG-Invoke-Check-NP-Normal*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);
instrs-of (*PROG P*) *C M* ! *pc* = *Invoke M′ n*;
ek = (λ*s*. *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc n*) ≠ *Null*)$_\surd$ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*⌋, *Normal*)
| *CFG-Invoke-Check-NP-Exceptional*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*);
instrs-of (*PROG P*) *C M* ! *pc* = *Invoke M′ n*;
pc′ = (*case* (*match-ex-table* (*PROG P*) *NullPointer pc* (*ex-table-of* (*PROG P*) *C M*)) *of*
None ⇒ *None*
| *Some* (*pc″*, *d*) ⇒ ⌊*pc″*⌋);
ek = (λ*s*. *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc n*) = *Null*)$_\surd$ ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −(*ek*)→ (*C*, *M*, ⌊*pc*⌋, *Exceptional pc′ Enter*)
| *CFG-Invoke-NP-prop*: ⟦ *C* ≠ *ClassMain P*;
(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*);
instrs-of (*PROG P*) *C M* ! *pc* = *Invoke M′ n*;
ek = ↑(λ*s*. *s*(*Exception* ↦ *Value* (*Addr* (*addr-of-sys-xcpt NullPointer*)))) ⟧
⟹ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*) −(*ek*)→ (*C*, *M*, *None*, *Return*)
| *CFG-Invoke-NP-handle*: ⟦ *C* ≠ *ClassMain P*; (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*);

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

$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n;$
$ek = \Uparrow(\lambda s.\ s(Exception := None,$
$Stack\ (stkLength\ (P,\ C,\ M)\ pc' - 1) := s\ Exception))\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Return)\ -(ek)\rightarrow\ (C,\ M,\ \lfloor pc' \rfloor,\ Enter)$

| *CFG-Invoke-Return-Exceptional-prop*: $[\![\ C \neq ClassMain\ P;$
$(P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Return);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n;$
$ek = (\lambda s.\ \exists v.\ s\ Exception = \lfloor v \rfloor\ \wedge$
$match\text{-}ex\text{-}table\ (PROG\ P)\ (cname\text{-}of\ (heap\text{-}of\ s)\ (the\text{-}Addr\ (the\text{-}Value\ v)))\ pc\ (ex\text{-}table\text{-}of\ (PROG\ P)\ C\ M) = None)_{\surd}\ ]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Return)\ -(ek)\rightarrow\ (C,\ M,\ None,\ Return)$

| *CFG-Return*: $[\![\ C \neq ClassMain\ P;\ (P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc \rfloor,\ Enter);$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = instr.Return;$
$ek = \Uparrow(\lambda s.\ s(Stack\ 0 := s\ (Stack\ (stkLength\ (P,\ C,\ M)\ pc - 1))))$
$]\!]$
$\Longrightarrow (P,\ C0,\ Main) \vdash (C,\ M,\ \lfloor pc \rfloor,\ Enter)\ -(ek)\rightarrow\ (C,\ M,\ None,\ Return)$

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

lemma *JVMCFG-edge-det*: $[\![\ P \vdash n\ -(et)\rightarrow\ n';\ P \vdash n\ -(et')\rightarrow\ n'\ ]\!] \Longrightarrow et = et'$
by (*erule JVMCFG.cases*) (*erule JVMCFG.cases*, (*fastforce dest*: *sees-method-fun*)+)+

lemma *sourcenode-reachable*: $P \vdash n\ -(ek)\rightarrow\ n' \Longrightarrow P \vdash \Rightarrow n$
by (*erule JVMCFG.cases*, *auto*)

lemma *targetnode-reachable*:
assumes *edge*: $P \vdash n\ -(ek)\rightarrow\ n'$
shows $P \vdash \Rightarrow n'$
proof −
from *edge* **have** $P \vdash \Rightarrow n$
by −(*drule sourcenode-reachable*)
with *edge* **show** *?thesis*
by −(*rule JVMCFG-reachable.intros*)
qed

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

lemma *ClassMain-imp-MethodMain*:
(*P*, *C0*, *Main*) ⊢ (*C'*, *M'*, *pc'*, *nt'*) −*ek*→ (*ClassMain P*, *M*, *pc*, *nt*) ⟹ *M* = *MethodMain P*
(*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *M*, *pc*, *nt*) ⟹ *M* = *MethodMain P*
proof (*induct P*==*P C0*≡*C0 Main*≡*Main C' M' pc' nt' ek C''*==*ClassMain P M pc nt* **and**
P==*P C0*≡*C0 Main*≡*Main C'*==*ClassMain P M pc nt*
rule: *JVMCFG-reachable-inducts*)
case *CFG-Return-from-Method*
thus *?case*
by (*fastforce elim*: *JVMCFG.cases*)
qed *auto*

lemma *ClassMain-no-Call-target* [*dest*!]:
(*P*, *C0*, *Main*) ⊢ (*C*, *M*, *pc*, *nt*) −*Q*:(*C'*, *M'*, *pc'*)↪$_{(D,M'')}$*paramDefs*→ (*ClassMain P*, *M'''*, *pc''*, *nt'*)
⟹ *False*
and
(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, *pc*, *nt*) ⟹ *True*
by (*induct P C0 Main C M pc nt ek*==*Q*:(*C'*, *M'*, *pc'*)↪$_{(D,M'')}$*paramDefs*
C''==*ClassMain P M''' pc'' nt'* **and**
P C0 Main C M pc nt
rule: *JVMCFG-reachable-inducts*) *auto*

lemma *method-of-src-and-trg-exists*:
⟦ (*P*, *C0*, *Main*) ⊢ (*C'*, *M'*, *pc'*, *nt'*) −*ek*→ (*C*, *M*, *pc*, *nt*); *C* ≠ *ClassMain P*; *C'* ≠ *ClassMain P* ⟧
⟹ (∃ *Ts T mb*. (*PROG P*) ⊢ *C sees M*:*Ts*→*T* = *mb in C*) ∧
(∃ *Ts T mb*. (*PROG P*) ⊢ *C' sees M'*:*Ts*→*T* = *mb in C'*)
and *method-of-reachable-node-exists*:
⟦ (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, *pc*, *nt*); *C* ≠ *ClassMain P* ⟧
⟹ ∃ *Ts T mb*. (*PROG P*) ⊢ *C sees M*:*Ts*→*T* = *mb in C*
proof (*induct rule*: *JVMCFG-reachable-inducts*)
case *CFG-Invoke-Call*
thus *?case*
by (*blast dest*: *sees-method-idemp*)
next
case (*reachable-step P C0 Main C M pc nt ek C' M' pc' nt'*)
show *?case*
proof (*cases C* = *ClassMain P*)
case *True*
with ‹(*P*, *C0*, *Main*) ⊢ (*C*, *M*, *pc*, *nt*) −*ek*→ (*C'*, *M'*, *pc'*, *nt'*)› ‹*C'* ≠ *ClassMain P*›
show *?thesis*
proof *cases*
case *Main-Call*
thus *?thesis*
by (*blast dest*: *sees-method-idemp*)
qed *auto*

next
case *False*
with *reachable-step* **show** *?thesis*
by *simp*
qed
qed *simp-all*

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* $\Rightarrow$ *True* |
$\lfloor pc'' \rfloor \Rightarrow (TYPING\ P)\ C\ M\ !\ pc'' \neq None \wedge pc'' < length\ (instrs\text{-}of\ (PROG\ P)\ C\ M))\ \wedge$
(*case pc' of None* $\Rightarrow$ *True* |
$\lfloor pc'' \rfloor \Rightarrow (TYPING\ P)\ C'\ M'\ !\ pc'' \neq None \wedge pc'' < length\ (instrs\text{-}of\ (PROG\ P)\ C'\ M'))$
and *instr-of-reachable-node-typable*: ⟦ $(P, C0, Main) \vdash \Rightarrow(C, M, pc, nt)$; $C \neq ClassMain\ P$ ⟧
$\Longrightarrow$ *case pc of None* $\Rightarrow$ *True* |
$\lfloor pc'' \rfloor \Rightarrow (TYPING\ P)\ C\ M\ !\ pc'' \neq None \wedge pc'' < length\ (instrs\text{-}of\ (PROG\ P)\ C\ M)$
proof (*induct rule*: *JVMCFG-reachable-inducts*)
case (*CFG-Load C P C0 Main M pc n ek*)
from ‹$(P, C0, Main) \vdash \Rightarrow(C, M, \lfloor pc \rfloor, Enter)$› ‹$C \neq ClassMain\ P$›
obtain *Ts T mxs* mxl_0 *is xt* **where** $PROG\ P \vdash C\ sees\ M{:}Ts\rightarrow T = (mxs, mxl_0, is, xt)\ in\ C$
and $instrs\text{-}of\ (PROG\ P)\ C\ M = is$
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-Load* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-Store C P C0 Main M pc n ek*)
from ‹$(P, C0, Main) \vdash \Rightarrow(C, M, \lfloor pc \rfloor, Enter)$› ‹$C \neq ClassMain\ P$›
obtain *Ts T mxs* mxl_0 *is xt* **where** $PROG\ P \vdash C\ sees\ M{:}Ts\rightarrow T = (mxs, mxl_0, is, xt)\ in\ C$
and $instrs\text{-}of\ (PROG\ P)\ C\ M = is$
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-Store* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-Push C P C0 Main M pc v ek*)
from ‹$(P, C0, Main) \vdash \Rightarrow(C, M, \lfloor pc \rfloor, Enter)$› ‹$C \neq ClassMain\ P$›
obtain *Ts T mxs* mxl_0 *is xt* **where** $PROG\ P \vdash C\ sees\ M{:}Ts\rightarrow T = (mxs, mxl_0, is, xt)\ in\ C$
and $instrs\text{-}of\ (PROG\ P)\ C\ M = is$
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-Push* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-Pop C P C0 Main M pc ek*)

from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs* mxl_0 *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, mxl_0, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-Pop* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-IAdd C P C0 Main M pc ek*)
from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs* mxl_0 *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, mxl_0, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-IAdd* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-Goto C P C0 Main M pc i*)
from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs* mxl_0 *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, mxl_0, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-Goto* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-CmpEq C P C0 Main M pc ek*)
from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs* mxl_0 *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, mxl_0, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-CmpEq* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-IfFalse-False C P C0 Main M pc i ek*)
from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs* mxl_0 *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, mxl_0, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-IfFalse-False* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-IfFalse-True C P C0 Main M pc i ek*)
from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs* mxl_0 *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, mxl_0, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*

by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-IfFalse-True* **show** *?case*
using [[*simproc del*: *list-to-set-comprehension*]] **by** (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-New-Update C P C0 Main M pc Cl ek*)
from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Normal*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs* mxl_0 *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, mxl_0, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-New-Update* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-New-Exceptional-handle C P C0 Main M pc pc′ Cl ek*)
hence *TYPING P C M* ! *pc* ≠ *None* **and** *pc* < *length* (*instrs-of* (*PROG P*) *C M*)
by *simp-all*
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs* mxl_0 **where**
PROG P ⊢ *C sees M*:*Ts*→*T* = (*mxs*, mxl_0, *instrs-of* (*PROG P*) *C M*, *ex-table-of* (*PROG P*) *C M*) *in C*
by (*fastforce dest*: *method-of-reachable-node-exists*)
with ‹*pc* < *length* (*instrs-of* (*PROG P*) *C M*)› ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *New Cl*›
have *PROG P*,*T*,*mxs*,*length* (*instrs-of* (*PROG P*) *C M*),*ex-table-of* (*PROG P*) *C M*
⊢ *New Cl*,*pc* :: *TYPING P C M*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *New Cl*› **obtain** *d′*
where *match-ex-table* (*PROG P*) *OutOfMemory pc* (*ex-table-of* (*PROG P*) *C M*) = ⌊(*pc′*, *d′*)⌋
by *cases* (*fastforce elim*: *JVMCFG.cases*)
hence ∃(*f*, *t*, *D*, *h*, *d*)∈*set* (*ex-table-of* (*PROG P*) *C M*).
matches-ex-entry (*PROG P*) *OutOfMemory pc* (*f*, *t*, *D*, *h*, *d*) ∧ *h* = *pc′* ∧ *d* = *d′*
by −(*drule match-ex-table-SomeD*)
ultimately show *?case* **using** ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *New Cl*›
by (*fastforce simp*: *relevant-entries-def is-relevant-entry-def matches-ex-entry-def*)
next
case (*CFG-Getfield-Update C P C0 Main M pc F Cl ek*)
from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Normal*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs* mxl_0 *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, mxl_0, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*
by −(*drule method-of-reachable-node-exists*, *auto*)

with *CFG-Getfield-Update* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-Getfield-Exceptional-handle C P C0 Main M pc pc' F Cl ek*)
hence *TYPING P C M ! pc* ≠ *None* **and** *pc* < *length* (*instrs-of* (*PROG P*) *C M*)
by *simp-all*
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs mxl*$_0$ **where**
PROG P ⊢ *C sees M*:*Ts*→*T* = (*mxs*, *mxl*$_0$, *instrs-of* (*PROG P*) *C M*, *ex-table-of* (*PROG P*) *C M*) *in C*
by (*fastforce dest*: *method-of-reachable-node-exists*)
with ‹*pc* < *length* (*instrs-of* (*PROG P*) *C M*)› ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Getfield F Cl*›
have *PROG P*,*T*,*mxs*,*length* (*instrs-of* (*PROG P*) *C M*),*ex-table-of* (*PROG P*) *C M*
⊢ *Getfield F Cl*,*pc* :: *TYPING P C M*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Getfield F Cl*› **obtain** *d'*
where *match-ex-table* (*PROG P*) *NullPointer pc* (*ex-table-of* (*PROG P*) *C M*) = ⌊(*pc'*, *d'*)⌋
by *cases* (*fastforce elim*: *JVMCFG.cases*)
hence ∃(*f*, *t*, *D*, *h*, *d*)∈*set* (*ex-table-of* (*PROG P*) *C M*).
matches-ex-entry (*PROG P*) *NullPointer pc* (*f*, *t*, *D*, *h*, *d*) ∧ *h* = *pc'* ∧ *d* = *d'*
by −(*drule match-ex-table-SomeD*)
ultimately show *?case* **using** ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Getfield F Cl*›
by (*fastforce simp*: *relevant-entries-def is-relevant-entry-def matches-ex-entry-def*)
next
case (*CFG-Putfield-Update C P C0 Main M pc F Cl ek*)
from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Normal*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs mxl*$_0$ *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, *mxl*$_0$, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-Putfield-Update* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-Putfield-Exceptional-handle C P C0 Main M pc pc' F Cl ek*)
hence *TYPING P C M ! pc* ≠ *None* **and** *pc* < *length* (*instrs-of* (*PROG P*) *C M*)
by *simp-all*
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs mxl*$_0$ **where**
PROG P ⊢ *C sees M*:*Ts*→*T* = (*mxs*, *mxl*$_0$, *instrs-of* (*PROG P*) *C M*, *ex-table-of* (*PROG P*) *C M*) *in C*

by (*fastforce dest*: *method-of-reachable-node-exists*)
with ‹*pc* < *length* (*instrs-of* (*PROG P*) *C M*)› ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Putfield F Cl*›
have *PROG P*,*T*,*mxs*,*length* (*instrs-of* (*PROG P*) *C M*),*ex-table-of* (*PROG P*) *C M*
⊢ *Putfield F Cl*,*pc* :: *TYPING P C M*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Putfield F Cl*› **obtain** *d'*
where *match-ex-table* (*PROG P*) *NullPointer pc* (*ex-table-of* (*PROG P*) *C M*) = ⌊(*pc'*, *d'*)⌋
by *cases* (*fastforce elim*: *JVMCFG.cases*)
hence ∃(*f*, *t*, *D*, *h*, *d*)∈*set* (*ex-table-of* (*PROG P*) *C M*).
matches-ex-entry (*PROG P*) *NullPointer pc* (*f*, *t*, *D*, *h*, *d*) ∧ *h* = *pc'* ∧ *d* = *d'*
by −(*drule match-ex-table-SomeD*)
ultimately show *?case* **using** ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Putfield F Cl*›
by (*fastforce simp*: *relevant-entries-def is-relevant-entry-def matches-ex-entry-def*)
next
case (*CFG-Checkcast-Check-Normal C P C0 Main M pc Cl ek*)
from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs mxl*$_0$ *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, *mxl*$_0$, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-Checkcast-Check-Normal* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*CFG-Checkcast-Exceptional-handle C P C0 Main M pc pc' Cl ek*)
hence *TYPING P C M* ! *pc* ≠ *None* **and** *pc* < *length* (*instrs-of* (*PROG P*) *C M*)
by *simp-all*
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs mxl*$_0$ **where**
PROG P ⊢ *C sees M*:*Ts*→*T* = (*mxs*, *mxl*$_0$, *instrs-of* (*PROG P*) *C M*, *ex-table-of* (*PROG P*) *C M*) *in C*
by (*fastforce dest*: *method-of-reachable-node-exists*)
with ‹*pc* < *length* (*instrs-of* (*PROG P*) *C M*)› ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Checkcast Cl*›
have *PROG P*,*T*,*mxs*,*length* (*instrs-of* (*PROG P*) *C M*),*ex-table-of* (*PROG P*) *C M*
⊢ *Checkcast Cl*,*pc* :: *TYPING P C M*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Checkcast Cl*› **obtain** *d'*
where *match-ex-table* (*PROG P*) *ClassCast pc* (*ex-table-of* (*PROG P*) *C M*) = ⌊(*pc'*, *d'*)⌋

by *cases* (*fastforce elim*: *JVMCFG.cases*)
hence ∃(*f*, *t*, *D*, *h*, *d*)∈*set* (*ex-table-of* (*PROG P*) *C M*).
matches-ex-entry (*PROG P*) *ClassCast pc* (*f*, *t*, *D*, *h*, *d*) ∧ *h* = *pc*′ ∧ *d* = *d*′
by −(*drule match-ex-table-SomeD*)
ultimately show *?case* **using** ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Checkcast Cl*›
by (*fastforce simp*: *relevant-entries-def is-relevant-entry-def matches-ex-entry-def*)
next
case (*CFG-Throw-handle C P C0 Main M pc pc*′ *ek*)
hence *TYPING P C M* ! *pc* ≠ *None* **and** *pc* < *length* (*instrs-of* (*PROG P*) *C M*)
by *simp-all*
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc*′⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs mxl*$_0$ **where**
PROG P ⊢ *C sees M*:*Ts*→*T* = (*mxs*, *mxl*$_0$, *instrs-of* (*PROG P*) *C M*, *ex-table-of* (*PROG P*) *C M*) *in C*
by (*fastforce dest*: *method-of-reachable-node-exists*)
with ‹*pc* < *length* (*instrs-of* (*PROG P*) *C M*)› ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Throw*›
have *PROG P*,*T*,*mxs*,*length* (*instrs-of* (*PROG P*) *C M*),*ex-table-of* (*PROG P*) *C M*
⊢ *Throw*,*pc* :: *TYPING P C M*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc*′⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Throw*› **obtain** *d*′ *Exc*
where *match-ex-table* (*PROG P*) *Exc pc* (*ex-table-of* (*PROG P*) *C M*) = ⌊(*pc*′, *d*′)⌋
by *cases* (*fastforce elim*: *JVMCFG.cases*)
hence ∃(*f*, *t*, *D*, *h*, *d*)∈*set* (*ex-table-of* (*PROG P*) *C M*).
matches-ex-entry (*PROG P*) *Exc pc* (*f*, *t*, *D*, *h*, *d*) ∧ *h* = *pc*′ ∧ *d* = *d*′
by −(*drule match-ex-table-SomeD*)
ultimately show *?case* **using** ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Throw*›
by (*fastforce simp*: *relevant-entries-def is-relevant-entry-def matches-ex-entry-def*)
next
case (*CFG-Invoke-NP-handle C P C0 Main M pc pc*′ *M*′ *n ek*)
hence *TYPING P C M* ! *pc* ≠ *None* **and** *pc* < *length* (*instrs-of* (*PROG P*) *C M*)
by *simp-all*
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc*′⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs mxl*$_0$ **where**
PROG P ⊢ *C sees M*:*Ts*→*T* = (*mxs*, *mxl*$_0$, *instrs-of* (*PROG P*) *C M*, *ex-table-of* (*PROG P*) *C M*) *in C*
by (*fastforce dest*: *method-of-reachable-node-exists*)
with ‹*pc* < *length* (*instrs-of* (*PROG P*) *C M*)› ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M*′ *n*›
have *PROG P*,*T*,*mxs*,*length* (*instrs-of* (*PROG P*) *C M*),*ex-table-of* (*PROG P*) *C M*

⊢ *Invoke M′ n,pc* :: *TYPING P C M*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*)› ‹*C* ≠ *ClassMain P*›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M′ n*› **obtain** *d′*
where *match-ex-table* (*PROG P*) *NullPointer pc* (*ex-table-of* (*PROG P*) *C M*) = ⌊(*pc′*, *d′*)⌋
by *cases* (*fastforce elim*: *JVMCFG.cases*)
hence ∃(*f*, *t*, *D*, *h*, *d*)∈*set* (*ex-table-of* (*PROG P*) *C M*).
matches-ex-entry (*PROG P*) *NullPointer pc* (*f*, *t*, *D*, *h*, *d*) ∧ *h* = *pc′* ∧ *d* = *d′*
by −(*drule match-ex-table-SomeD*)
ultimately show *?case* **using** ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M′ n*›
by (*fastforce simp*: *relevant-entries-def is-relevant-entry-def matches-ex-entry-def*)
next
case (*CFG-Invoke-Return-Exceptional-handle C P C0 Main M pc pc′ M′ n ek*)
hence *TYPING P C M* ! *pc* ≠ *None* **and** *pc* < *length* (*instrs-of* (*PROG P*) *C M*)
by *simp-all*
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Return*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs mxl*$_0$ **where**
PROG P ⊢ *C sees M*:*Ts*→*T* = (*mxs*, *mxl*$_0$, *instrs-of* (*PROG P*) *C M*, *ex-table-of* (*PROG P*) *C M*) *in C*
by (*fastforce dest*: *method-of-reachable-node-exists*)
with ‹*pc* < *length* (*instrs-of* (*PROG P*) *C M*)› ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M′ n*›
have *PROG P*,*T*,*mxs*,*length* (*instrs-of* (*PROG P*) *C M*),*ex-table-of* (*PROG P*) *C M*
⊢ *Invoke M′ n*,*pc* :: *TYPING P C M*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
moreover from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Return*)› ‹*C* ≠ *ClassMain P*›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M′ n*› **obtain** *d′ Exc*
where *match-ex-table* (*PROG P*) *Exc pc* (*ex-table-of* (*PROG P*) *C M*) = ⌊(*pc′*, *d′*)⌋
by *cases* (*fastforce elim*: *JVMCFG.cases*)
hence ∃(*f*, *t*, *D*, *h*, *d*)∈*set* (*ex-table-of* (*PROG P*) *C M*).
matches-ex-entry (*PROG P*) *Exc pc* (*f*, *t*, *D*, *h*, *d*) ∧ *h* = *pc′* ∧ *d* = *d′*
by −(*drule match-ex-table-SomeD*)
ultimately show *?case* **using** ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M′ n*›
by (*fastforce simp*: *relevant-entries-def is-relevant-entry-def matches-ex-entry-def*)
next
case (*CFG-Invoke-Return-Check-Normal C P C0 Main M pc M′ n ST LT ek*)
from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Return*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs mxl*$_0$ *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, *mxl*$_0$, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*
by −(*drule method-of-reachable-node-exists*, *auto*)
with *CFG-Invoke-Return-Check-Normal* **show** *?case*

by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr* [*OF wf-jvmprog-is-wf-typ*])
next
case (*Method-LTrue P C0 Main C M*)
from ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, *None*, *Enter*)› ‹*C* ≠ *ClassMain P*›
obtain *Ts T mxs* mxl_0 *is xt* **where** *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, mxl_0, *is*, *xt*) *in C*
and *instrs-of* (*PROG P*) *C M* = *is*
by −(*drule method-of-reachable-node-exists*, *auto*)
with *Method-LTrue* **show** *?case*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-start* [*OF wf-jvmprog-is-wf-typ*] *simp*: *wt-start-def*)
next
case (*reachable-step P C0 Main C M opc nt ek C' M' opc' nt'*)
thus *?case*
by (*cases C* = *ClassMain P*) (*fastforce elim*: *JVMCFG.cases*, *simp*)
qed *simp-all*

lemma *reachable-node-impl-wt-instr*:
assumes (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *nt*)
and *C* ≠ *ClassMain P*
shows ∃ *T mxs mpc xt*. *PROG P*,*T*,*mxs*,*mpc*,*xt* ⊢ (*instrs-of* (*PROG P*) *C M* ! *pc*),*pc* :: *TYPING P C M*
proof −
from ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *nt*)›
method-of-reachable-node-exists [*of P C0 Main C M* ⌊*pc*⌋ *nt*]
instr-of-reachable-node-typable [*of P C0 Main C M* ⌊*pc*⌋ *nt*]
obtain *Ts T mxs* mxl_0 *is xt*
where *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, mxl_0, *is*, *xt*) *in C*
and *TYPING P C M* ! *pc* ≠ *None*
and *pc* < *length* (*instrs-of* (*PROG P*) *C M*)
by *fastforce*+
with *wf-jvmprog-is-wf-typ* [*of P*]
have *PROG P*,*T*,*mxs*,*length is*,*xt* ⊢ *instrs-of* (*PROG P*) *C M* ! *pc*,*pc* :: *TYPING P C M*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr*)
thus *?thesis*
by *blast*
qed

lemma
⟦ (*P*, *C0*, *Main*) ⊢ (*C*, *M*, *pc*, *nt*) −*ek*→ (*C'*, *M'*, *pc'*, *nt'*); *C* ≠ *ClassMain P* ∨ *C'* ≠ *ClassMain P* ⟧
⟹ ∃ *T mb D*. *PROG P* ⊢ *C0 sees Main*:[]→*T* = *mb in D*
and *reachable-node-impl-Main-ex*:
⟦ (*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, *pc*, *nt*); *C* ≠ *ClassMain P*⟧
⟹ ∃ *T mb D*. *PROG P* ⊢ *C0 sees Main*:[]→*T* = *mb in D*
by (*induct rule*: *JVMCFG-reachable-inducts*) *fastforce*+

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* ⊢ (*sourcenode e*) −(*kind e*)→ (*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*)
  by (*induct rule*: *methods.induct*) (*auto split*: *split-if-asm*)

lemma *in-methods-in-msD*: ((*C*, *M*), *ins*, *outs*) ∈ *set* (*methods D ms*)
  ⟹ *M* ∈ *set* (*map fst ms*) ∧ *D* = *C*
  by (*induct ms*) *auto*

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*]
  by (*induct rule*: *methods.induct*) *fastforce*+

lemma *in-set-methodsE*:
  assumes ((*C*, *M*), *ins*, *outs*) ∈ *set* (*methods D ms*)

obtains *Ts T mb*
where $(M, Ts, T, mb) \in set\ ms$
and $D = C$
and $ins = Heap\ \#\ (map\ Local\ [0..<Suc\ (length\ Ts)])$
and $outs = [Heap, Stack\ 0, Exception]$
using *assms*
by (*induct ms*) *fastforce+*

lemma *in-set-procsI*:
assumes *sees*: $P \vdash D\ sees\ M: Ts \to T = mb\ in\ D$
and *ins-def*: $ins = Heap\ \#\ map\ Local\ [0..<Suc\ (length\ Ts)]$
and *outs-def*: $outs = [Heap, Stack\ 0, Exception]$
shows $((D, M), ins, outs) \in set\ (procs\ P)$
proof −
from *sees* **obtain** $D'\ fs\ ms$ **where** $map\text{-}of\ P\ D = \lfloor (D', fs, ms) \rfloor$ **and** *map-of ms* $M = \lfloor (Ts, T, mb) \rfloor$
by (*fastforce dest*: *visible-method-exists simp*: *class-def*)
hence $(D, D', fs, ms) \in set\ P$
by −(*drule map-of-SomeD*)
thus *?thesis*
proof (*induct P*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons Class P*)
with *ins-def outs-def* ‹*map-of ms* $M = \lfloor (Ts, T, mb) \rfloor$› **show** *?case*
by (*cases Class*, *cases mb*) (*auto intro*: *in-set-methodsI*)
qed
qed

lemma *distinct-methods*: $distinct\ (map\ fst\ ms) \Longrightarrow distinct\ (map\ fst\ (methods\ C\ ms))$
proof (*induct ms*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons M ms*)
thus *?case*
by (*cases M*) (*auto dest*: *in-methods-in-msD*)
qed

lemma *in-set-procsD*:
$((C, M), ins, out) \in set\ (procs\ P) \Longrightarrow \exists D\ fs\ ms.\ (C, D, fs, ms) \in set\ P \wedge M \in set\ (map\ fst\ ms)$
proof (*induct P*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons Class P*)
thus *?case*
by (*cases Class*) (*fastforce dest*: *in-methods-in-msD intro*: *rev-image-eqI*)
qed

lemma *in-set-procsE'*:
assumes $((C, M), ins, outs) \in set\ (procs\ P)$
obtains *D fs ms Ts T mb*
where $(C, D, fs, ms) \in set\ P$
and $(M, Ts, T, mb) \in set\ ms$
and $ins = Heap\ \#\ (map\ (\lambda n.\ Local\ n)\ [0..<Suc\ (length\ Ts)])$
and $outs = [Heap, Stack\ 0, Exception]$
using *assms*
by (*induct P*) (*fastforce elim*: *in-set-methodsE*)+

lemma *distinct-Local-vars* [*simp*]: *distinct* (*map Local* [*0..<n*])
by (*induct n*) *auto*

lemma *distinct-Stack-vars* [*simp*]: *distinct* (*map Stack* [*0..<n*])
by (*induct n*) *auto*

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

lemma *get-return-edgesE* [*elim*!]:
assumes $a \in get\text{-}return\text{-}edges\ P\ a'$
obtains *Q C M pc D M' paramDefs* **where**
$kind\ a' = Q{:}(C, M, pc){\hookrightarrow}_{(D, M')}paramDefs$
and $a = ((D, M', None, Return),$
$(\lambda(s, ret).\ ret = (C, M, pc)){\hookleftarrow}_{(D, M')}(\lambda s\ s'.\ s'(Heap := s\ Heap, Exception :=$
s Exception,
$Stack\ (stkLength\ (P, C, M)\ (Suc\ pc) - 1) := s\ (Stack\ 0))),$
$(C, M, \lfloor pc \rfloor, Return))$
using *assms*
by −(*cases a*, *cases a'*, *clarsimp*, *erule get-return-edges.cases*, *fastforce*)

lemma *distinct-class-names*: *distinct-fst* (*PROG P*)
using *wf-jvmprog-is-wf-typ* [*of P*]
by (*clarsimp simp*: *wf-jvm-prog-phi-def wf-prog-def*)

lemma *distinct-method-names*:
$class\ (PROG\ P)\ C = \lfloor (D, fs, ms) \rfloor \Longrightarrow distinct\text{-}fst\ ms$
using *wf-jvmprog-is-wf-typ* [*of P*]
unfolding *wf-jvm-prog-phi-def*

by (*fastforce dest*: *class-wf simp*: *wf-cdecl-def*)

lemma *distinct-fst-is-distinct-fst*: *distinct-fst* = *BasicDefs.distinct-fst*
by (*simp add*: *distinct-fst-def BasicDefs.distinct-fst-def*)

lemma *ClassMain-not-in-set-PROG* [*dest*!]: (*ClassMain P*, *D*, *fs*, *ms*) ∈ *set* (*PROG P*) ⟹ *False*
using *distinct-class-names* [*of P*] *ClassMain-is-no-class* [*of P*]
by (*fastforce intro*: *map-of-SomeI simp*: *class-def*)

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 −
from ‹((*C*, *M*), *ins*, *outs*) ∈ *set* (*procs* (*PROG P*))›
obtain *D fs ms Ts T mxs mxl*$_0$ *is xt*
where (*C*, *D*, *fs*, *ms*) ∈ *set* (*PROG P*)
and (*M*, *Ts*, *T*, *mxs*, *mxl*$_0$, *is*, *xt*) ∈ *set ms*
and *ins* = *Heap* # (*map* (λ*n*. *Local n*) [*0*..<*Suc* (*length Ts*)])
and *outs* = [*Heap*, *Stack 0*, *Exception*]
by (*fastforce elim*: *in-set-procsE'*)
moreover from ‹(*C*, *D*, *fs*, *ms*) ∈ *set* (*PROG P*)› *distinct-class-names* [*of P*]
have *class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋
by (*fastforce intro*: *map-of-SomeI simp*: *class-def*)
moreover from *wf-jvmprog-is-wf-typ* [*of P*]
‹(*M*, *Ts*, *T*, *mxs*, *mxl*$_0$, *is*, *xt*) ∈ *set ms*› ‹(*C*, *D*, *fs*, *ms*) ∈ *set* (*PROG P*)›
have *PROG P* ⊢ *C sees M*:*Ts*→*T* = (*mxs*, *mxl*$_0$, *is*, *xt*) *in C*
by (*fastforce intro*: *mdecl-visible simp*: *wf-jvm-prog-phi-def*)
ultimately show *?thesis* **using** ‹(⋀*D fs ms Ts T mb*.
⟦*class* (*PROG P*) *C* = ⌊(*D*, *fs*, *ms*)⌋; *PROG P* ⊢ *C sees M*: *Ts*→*T* = *mb in C*;
ins = *Heap* # *map Local* [*0*..<*Suc* (*length Ts*)]; *outs* = [*Heap*, *Stack 0*, *Exception*]⟧
⟹ *thesis*)›
by *blast*
qed

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 (*unfold-locales*)
fix *e*
assume *valid-edge* (*P*, *C0*, *Main*) *e*
and *targetnode e* = (*ClassMain P*, *MethodMain P*, *None*, *Enter*)
thus *False*
by (*auto simp*: *valid-edge-def*)(*erule JVMCFG.cases*, *auto*)+
next
show (λ(*C*, *M*, *pc*, *type*). (*C*, *M*)) (*ClassMain P*, *MethodMain P*, *None*, *Enter*)
=
(*ClassMain P*, *MethodMain P*)
by *simp*
next
fix *a Q r p fs*
assume *valid-edge* (*P*, *C0*, *Main*) *a*
and *kind a* = *Q*:*r*↪$_p$*fs*
and *sourcenode a* = (*ClassMain P*, *MethodMain P*, *None*, *Enter*)
thus *False*
by (*auto simp*: *valid-edge-def*) (*erule JVMCFG.cases*, *auto*)
next
fix *a a'*
assume *valid-edge* (*P*, *C0*, *Main*) *a*
and *valid-edge* (*P*, *C0*, *Main*) *a'*
and *sourcenode a* = *sourcenode a'*
and *targetnode a* = *targetnode a'*
thus *a* = *a'*
by (*cases a*, *cases a'*) (*fastforce simp*: *valid-edge-def dest*: *JVMCFG-edge-det*)
next
fix *a Q r f*
assume *valid-edge* (*P*, *C0*, *Main*) *a*
and *kind a* = *Q*:*r*↪$_{(ClassMain\ P,\ MethodMain\ P)}{}^f$
thus *False*
by (*clarsimp simp*: *valid-edge-def*) (*erule JVMCFG.cases*, *auto*)
next
fix *a Q' f'*
assume *valid-edge* (*P*, *C0*, *Main*) *a* **and** *kind a* = *Q'*↩$_{(ClassMain\ P,\ MethodMain\ P)}{}^{f'}$
thus *False*
by (*clarsimp simp*: *valid-edge-def*) (*erule JVMCFG.cases*, *auto*)+
next
fix *a Q r p fs*
assume *valid-edge* (*P*, *C0*, *Main*) *a*
and *kind a* = *Q*:*r*↪$_p$*fs*
then obtain *C M pc nt C' M' pc' nt'*
where (*P*, *C0*, *Main*) ⊢ (*C*, *M*, *pc*, *nt*) −*Q*:*r*↪$_p$*fs*→ (*C'*, *M'*, *pc'*, *nt'*)
by (*cases a*) (*clarsimp simp*: *valid-edge-def*)
thus ∃ *ins outs*.
(*p*, *ins*, *outs*) ∈ *set* (((*ClassMain P*, *MethodMain P*), [], []) # *procs* (*PROG P*))
proof *cases*

```
    case (Main-Call T mxs mxl0 is xt initParams)
    hence ((C', Main), [Heap, Local 0], [Heap, Stack 0, Exception]) ∈ set (procs
(PROG P))
      and p = (C', Main)
      by (auto intro: in-set-procsI dest: sees-method-idemp)
    thus ?thesis by fastforce
  next
    case (CFG-Invoke-Call - n - - - Ts)
    hence ((C', M'), Heap # map (λn. Local n) [0..<Suc (length Ts)],
      [Heap, Stack 0, Exception]) ∈ set (procs (PROG P))
      and p = (C',M')
      by (auto intro: in-set-procsI dest: sees-method-idemp)
    thus ?thesis by fastforce
  qed simp-all
next
  fix a
  assume valid-edge (P, C0, Main) a and intra-kind (kind a)
  thus (λ(C, M, pc, type). (C, M)) (sourcenode a) =
    (λ(C, M, pc, type). (C, M)) (targetnode a)
   by (clarsimp simp: valid-edge-def) (erule JVMCFG.cases, auto simp: intra-kind-def)
next
  fix a Q r p fs
  assume valid-edge (P, C0, Main) a and kind a = Q:r↪pfs
  thus (λ(C, M, pc, type). (C, M)) (targetnode a) = p
    by (clarsimp simp: valid-edge-def) (erule JVMCFG.cases, auto)
next
  fix a Q' p f'
  assume valid-edge (P, C0, Main) a and kind a = Q'↩pf'
  thus (λ(C, M, pc, type). (C, M)) (sourcenode a) = p
    by (clarsimp simp: valid-edge-def) (erule JVMCFG.cases, auto)
next
  fix a Q r p fs
  assume valid-edge (P, C0, Main) a and kind a = Q:r↪pfs
  thus ∀ a'. valid-edge (P, C0, Main) a' ∧ targetnode a' = targetnode a
    ⟶ (∃ Qx rx fsx. kind a' = Qx:rx↪pfsx)
    by (cases a, clarsimp simp: valid-edge-def) (erule JVMCFG.cases, auto)+
next
  fix a Q' p f'
  assume valid-edge (P, C0, Main) a and kind a = Q'↩pf'
  thus ∀ a'. valid-edge (P, C0, Main) a' ∧ sourcenode a' = sourcenode a
    ⟶ (∃ Qx fx. kind a' = Qx↩pfx)
    by (cases a, clarsimp simp: valid-edge-def) (erule JVMCFG.cases, auto)+
next
  fix a Q r p fs
  assume valid-edge (P, C0, Main) a and kind a = Q:r↪pfs
  then have ∃ a'. a' ∈ get-return-edges P a
    by (cases p, cases r) (fastforce intro: get-return-edges.intros)
  then show get-return-edges P a ≠ {}
    by (simp only: ex-in-conv) simp
```

next
fix *a a'*
assume *valid-edge (P, C0, Main) a a' ∈ get-return-edges P a*
then obtain *Q C M pc D M' paramDefs*
where *(P, C0, Main) ⊢ sourcenode a −Q:(C, M, pc)↪(D, M')paramDefs→ targetnode a*
and *kind a = Q:(C, M, pc)↪(D, M')paramDefs*
and *a'-def*: *a' = ((D, M', None, nodeType.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⌋, nodeType.Return)*
by (*fastforce simp*: *valid-edge-def*)
thus *valid-edge (P, C0, Main) a'*
proof *cases*
case (*Main-Call T mxs mxl0 is xt D'*)
hence *D = D'* **and** *M' = Main*
by *simp-all*
with ‹*(P, C0, Main) ⊢ ⇒(ClassMain P, MethodMain P, ⌊0⌋, Normal)*›
‹*PROG P ⊢ C0 sees Main*: *[]→T = (mxs, mxl0, is, xt) in D'*›
have *(P, C0, Main) ⊢ ⇒(D, M', None, Enter)*
by −(*rule reachable-step*, *fastforce*, *fastforce intro*: *JVMCFG-reachable.Main-Call*)
hence *(P, C0, Main) ⊢ ⇒(D, M', None, nodeType.Return)*
by −(*rule reachable-step*, *fastforce*, *fastforce intro*: *JVMCFG-reachable.Method-LFalse*)
with *a'-def Main-Call* **show** *?thesis*
by (*fastforce intro*: *CFG-Return-from-Method JVMCFG-reachable.Main-Call simp*: *valid-edge-def*)
next
case (*CFG-Invoke-Call - - - M'' - - - - - - - - - - D'*)
hence *D = D'* **and** *M' = M''*
by *simp-all*
with *CFG-Invoke-Call*
have *(P, C0, Main) ⊢ ⇒(D, M', None, Enter)*
by −(*rule reachable-step*, *fastforce*, *fastforce intro*: *JVMCFG-reachable.CFG-Invoke-Call*)
hence *(P, C0, Main) ⊢ ⇒(D, M', None, nodeType.Return)*
by −(*rule reachable-step*, *fastforce*, *fastforce intro*: *JVMCFG-reachable.Method-LFalse*)
with *a'-def CFG-Invoke-Call* **show** *?thesis*
by (*fastforce intro*: *CFG-Return-from-Method JVMCFG-reachable.CFG-Invoke-Call simp*: *valid-edge-def*)
qed *simp-all*
next
fix *a a'*
assume *valid-edge (P, C0, Main) a* **and** *a' ∈ get-return-edges P a*
thus *∃ Q r p fs. kind a = Q:r↪pfs*
by *clarsimp*
next
fix *a Q r p fs a'*
assume *valid-edge (P, C0, Main) a* **and** *kind a = Q:r↪pfs* **and** *a' ∈ get-return-edges P a*

```
  thus ∃ Q' f'. kind a' = Q'↩_p f'
    by clarsimp
next
  fix a Q' p f'
  assume valid-edge (P, C0, Main) a and kind a = Q'↩_p f'
  show ∃!a'. valid-edge (P, C0, Main) a' ∧
              (∃ Q r fs. kind a' = Q:r↪_p fs) ∧ a ∈ get-return-edges P a'
  proof (rule ex-ex1I)
    from ‹valid-edge (P, C0, Main) a›
    have (P, C0, Main) ⊢ sourcenode a −kind a→ targetnode a
      by (clarsimp simp: valid-edge-def)
    from this ‹kind a = Q'↩_p f'›
    show ∃ a'. valid-edge (P, C0, Main) a' ∧ (∃ Q r fs. kind a' = Q:r↪_p fs)
      ∧ a ∈ get-return-edges P a'
       by cases (cases a, fastforce intro: get-return-edges.intros[simplified] simp:
valid-edge-def)+
  next
    fix a' a''
    assume valid-edge (P, C0, Main) a'
      ∧ (∃ Q r fs. kind a' = Q:r↪_p fs) ∧ a ∈ get-return-edges P a'
       and valid-edge (P, C0, Main) a''
      ∧ (∃ Q r fs. kind a'' = Q:r↪_p fs) ∧ a ∈ get-return-edges P a''
    thus a' = a''
      by (cases a', cases a'', clarsimp simp: valid-edge-def)
    (erule JVMCFG.cases, simp-all, clarsimp? )+
  qed
next
  fix a a'
  assume valid-edge (P, C0, Main) a and a' ∈ get-return-edges P a
  thus ∃ a''. valid-edge (P, C0, Main) a'' ∧
    sourcenode a'' = targetnode a ∧
    targetnode a'' = sourcenode a' ∧ kind a'' = (λcf. False)_√
   by (clarsimp simp: valid-edge-def) (erule JVMCFG.cases, auto intro: JVMCFG-reachable.intros)
next
  fix a a'
  assume valid-edge (P, C0, Main) a and a' ∈ get-return-edges P a
  thus ∃ a''. valid-edge (P, C0, Main) a'' ∧
    sourcenode a'' = sourcenode a ∧
    targetnode a'' = targetnode a' ∧ kind a'' = (λcf. False)_√
   by (clarsimp simp: valid-edge-def) (erule JVMCFG.cases, auto intro: JVMCFG-reachable.intros)
next
  fix a Q r p fs
  assume valid-edge (P, C0, Main) a and kind a = Q:r↪_p fs
  hence call: (P, C0, Main) ⊢ sourcenode a −Q:r↪_p fs→ targetnode a
    by (clarsimp simp: valid-edge-def)
  show ∃!a'. valid-edge (P, C0, Main) a' ∧
    sourcenode a' = sourcenode a ∧ intra-kind (kind a')
  proof (rule ex-ex1I)
    from call
```

show $\exists a'.$ *valid-edge* $(P,\ C0,\ Main)\ a' \wedge$ *sourcenode* $a' =$ *sourcenode* $a \wedge$ *intra-kind* $(kind\ a')$
by *cases* (*fastforce intro*: *JVMCFG-reachable.intros simp*: *intra-kind-def valid-edge-def*)+
next
fix $a'\ a''$
assume *valid-edge* $(P,\ C0,\ Main)\ a' \wedge$ *sourcenode* $a' =$ *sourcenode* $a \wedge$ *intra-kind* $(kind\ a')$
and *valid-edge* $(P,\ C0,\ Main)\ a'' \wedge$ *sourcenode* $a'' =$ *sourcenode* $a \wedge$ *intra-kind* $(kind\ a'')$
with *call* **show** $a' = a''$
by (*cases* a, *cases* a', *cases* a'', *clarsimp simp*: *valid-edge-def intra-kind-def*)
(*erule JVMCFG.cases*, *simp-all*, *clarsimp?*)+
qed
next
fix $a\ Q'\ p\ f'$
assume *valid-edge* $(P,\ C0,\ Main)\ a$ **and** $kind\ a = Q'\hookleftarrow_p f'$
hence *return*: $(P,\ C0,\ Main) \vdash$ *sourcenode* $a\ -Q'\hookleftarrow_p f'\rightarrow$ *targetnode* a
by (*clarsimp simp*: *valid-edge-def*)
show $\exists!a'.$ *valid-edge* $(P,\ C0,\ Main)\ a' \wedge$
targetnode $a' =$ *targetnode* $a \wedge$ *intra-kind* $(kind\ a')$
proof (*rule ex-ex1I*)
from *return*
show $\exists a'.$ *valid-edge* $(P,\ C0,\ Main)\ a' \wedge$ *targetnode* $a' =$ *targetnode* $a \wedge$ *intra-kind* $(kind\ a')$
proof *cases*
case (*CFG-Return-from-Method C M C' M' pc' Q'' ps Q stateUpdate*)
hence [*simp*]: $Q = Q'$ **and** [*simp*]: $p = (C,\ M)$ **and** [*simp*]: $f' = stateUpdate$
by *simp-all*
from ‹$(P,\ C0,\ Main) \vdash (C',\ M',\ \lfloor pc' \rfloor,\ Normal)\ -Q'':(C',\ M',\ pc')\hookrightarrow_{(C,\ M)} ps\rightarrow (C,\ M,\ None,\ Enter)$›
have *invoke-reachable*: $(P,\ C0,\ Main) \vdash \Rightarrow(C',\ M',\ \lfloor pc' \rfloor,\ Normal)$
by −(*drule sourcenode-reachable*)
show *?thesis*
proof (*cases* $C' =$ *ClassMain* P)
case *True*
with *invoke-reachable CFG-Return-from-Method* **show** *?thesis*
by −(*erule JVMCFG.cases*, *simp-all*,
fastforce intro: *Main-Call-LFalse simp*: *valid-edge-def intra-kind-def*)
next
case *False*
with *invoke-reachable CFG-Return-from-Method* **show** *?thesis*
by −(*erule JVMCFG.cases*, *simp-all*,
fastforce intro: *CFG-Invoke-False simp*: *valid-edge-def intra-kind-def*)
qed
qed *simp-all*
next
fix $a'\ a''$
assume *valid-edge* $(P,\ C0,\ Main)\ a' \wedge$ *targetnode* $a' =$ *targetnode* $a \wedge$ *intra-kind*

```
(kind a')
    and valid-edge (P, C0, Main) a'' ∧ targetnode a'' = targetnode a ∧ intra-kind
(kind a'')
   with return show a' = a''
     by (cases, auto, cases a, cases a', cases a'', clarsimp simp: valid-edge-def
intra-kind-def)
   (erule JVMCFG.cases, simp-all, clarsimp?)+
 qed
next
 fix a a' Q1 r1 p fs1 Q2 r2 fs2
 assume valid-edge (P, C0, Main) a and valid-edge (P, C0, Main) a'
  and kind a = Q1:r1↪pfs1 and kind a' = Q2:r2↪pfs2
 thus targetnode a = targetnode a'
  by (cases a, cases a', clarsimp simp: valid-edge-def)
 (erule JVMCFG.cases, simp-all, clarsimp?)+
next
 from distinct-method-names [of P] distinct-class-names [of P]
 have ⋀C D fs ms. (C, D, fs, ms) ∈ set (PROG P) ⟹ distinct-fst ms
  by (fastforce intro: map-of-SomeI simp: class-def)
 moreover {
  fix P
  assume distinct-fst (P :: jvm-prog)
   and ⋀C D fs ms. (C, D, fs, ms) ∈ set P ⟹ distinct-fst ms
  hence distinct-fst (procs P)
   by (induct P, simp)
  (fastforce intro: equals0I rev-image-eqI dest: in-methods-in-msD in-set-procsD
    simp: distinct-methods distinct-fst-def)
 }
 ultimately have distinct-fst (procs (PROG P)) using distinct-class-names [of
P]
  by blast
 hence BasicDefs.distinct-fst (procs (PROG P))
  by (simp add: distinct-fst-is-distinct-fst)
 thus BasicDefs.distinct-fst (((ClassMain P, MethodMain P), [], []) # procs
(PROG P))
  by (fastforce elim: in-set-procsE)
next
 fix C M P p ins outs
 assume (p, ins, outs) ∈ set (((C, M), [], []) # procs P)
 thus distinct ins
 proof (induct P)
  case Nil
  thus ?case by simp
 next
  case (Cons Cl P)
  then obtain C D fs ms where Cl = (C, D, fs, ms)
   by (cases Cl) blast
  with Cons show ?case
   by auto (induct ms, auto)
```

qed
next
fix *C M P p ins outs*
assume (*p*, *ins*, *outs*) ∈ *set* (((*C*, *M*), [], []) # *procs P*)
thus *distinct outs*
proof (*induct P*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons Cl P*)
then obtain *C D fs ms* **where** *Cl* = (*C*, *D*, *fs*, *ms*)
by (*cases Cl*) *blast*
with *Cons* **show** *?case*
by *auto* (*induct ms*, *auto*)
qed
qed

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 (*unfold-locales*)
fix *a*
assume *valid-edge* (*P*, *C0*, *Main*) *a*
and *sourcenode a* = (*ClassMain P*, *MethodMain P*, *None*, *nodeType.Return*)
thus *False*
by (*cases a*, *clarsimp simp*: *valid-edge-def*) (*erule JVMCFG.cases*, *simp-all*, *clarsimp*)
next
show (λ(*C*, *M*, *pc*, *type*). (*C*, *M*)) (*ClassMain P*, *MethodMain P*, *None*, *nodeType.Return*) =
(*ClassMain P*, *MethodMain P*)
by *simp*
next
fix *a Q p f*
assume *valid-edge* (*P*, *C0*, *Main*) *a*
and *kind a* = *Q*↪*pf*
and *targetnode a* = (*ClassMain P*, *MethodMain P*, *None*, *nodeType.Return*)
thus *False*
by (*cases a*, *clarsimp simp*: *valid-edge-def*) (*erule JVMCFG.cases*, *simp-all*)
next
show ∃ *a*. *valid-edge* (*P*, *C0*, *Main*) *a* ∧
sourcenode a = (*ClassMain P*, *MethodMain P*, *None*, *Enter*) ∧
targetnode a = (*ClassMain P*, *MethodMain P*, *None*, *nodeType.Return*) ∧
kind a = (λ*s*. *False*)$_{\surd}$
by (*fastforce intro*: *JVMCFG-reachable.intros simp*: *valid-edge-def*)

qed

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*)
  ⟹ *Exception* ∈ *Def P n*
| *Def-Main-Stack-0*:
  n = (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Return*)
  ⟹ *Stack 0* ∈ *Def P n*
| *Def-Load*:
  ⟦ *n* = (*C*, *M*, ⌊*pc*⌋, *Enter*);
  C ≠ *ClassMain P*;
  instrs-of (*PROG P*) *C M* ! *pc* = *Load idx*;
  i = *stkLength* (*P*, *C*, *M*) *pc*⟧
  ⟹ *Stack i* ∈ *Def P n*
| *Def-Store*:
  ⟦ *n* = (*C*, *M*, ⌊*pc*⌋, *Enter*);
  C ≠ *ClassMain P*;
  instrs-of (*PROG P*) *C M* ! *pc* = *Store idx* ⟧
  ⟹ *Local idx* ∈ *Def P n*
| *Def-Push*:
  ⟦ *n* = (*C*, *M*, ⌊*pc*⌋, *Enter*);
  C ≠ *ClassMain P*;
  instrs-of (*PROG P*) *C M* ! *pc* = *Push v*;
  i = *stkLength* (*P*, *C*, *M*) *pc* ⟧
  ⟹ *Stack i* ∈ *Def P n*
| *Def-IAdd*:
  ⟦ *n* = (*C*, *M*, ⌊*pc*⌋, *Enter*);
  C ≠ *ClassMain P*;
  instrs-of (*PROG P*) *C M* ! *pc* = *IAdd*;
  i = *stkLength* (*P*, *C*, *M*) *pc* − *2* ⟧
  ⟹ *Stack i* ∈ *Def P n*
| *Def-CmpEq*:
  ⟦ *n* = (*C*, *M*, ⌊*pc*⌋, *Enter*);
  C ≠ *ClassMain P*;
  instrs-of (*PROG P*) *C M* ! *pc* = *CmpEq*;
  i = *stkLength* (*P*, *C*, *M*) *pc* − *2* ⟧
  ⟹ *Stack i* ∈ *Def P n*
| *Def-New-Heap*:

$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ Normal);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = New\ Cl\ ]\!]$
$\Longrightarrow Heap \in Def\ P\ n$
| *Def-New-Stack*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ Normal);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = New\ Cl;$
$i = stkLength\ (P,\ C,\ M)\ pc\ ]\!]$
$\Longrightarrow Stack\ i \in Def\ P\ n$
| *Def-Exception*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ pco\ nt);$
$C \neq ClassMain\ P\ ]\!]$
$\Longrightarrow Exception \in Def\ P\ n$
| *Def-Exception-handle*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Enter);$
$C \neq ClassMain\ P;$
$i = stkLength\ (P,\ C,\ M)\ pc' - 1\ ]\!]$
$\Longrightarrow Stack\ i \in Def\ P\ n$
| *Def-Exception-handle-return*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ Exceptional\ \lfloor pc' \rfloor\ Return);$
$C \neq ClassMain\ P;$
$i = stkLength\ (P,\ C,\ M)\ pc' - 1\ ]\!]$
$\Longrightarrow Stack\ i \in Def\ P\ n$
| *Def-Getfield*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ Normal);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Getfield\ Cl\ Fd;$
$i = stkLength\ (P,\ C,\ M)\ pc - 1\ ]\!]$
$\Longrightarrow Stack\ i \in Def\ P\ n$
| *Def-Putfield*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ Normal);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Putfield\ Cl\ Fd\ ]\!]$
$\Longrightarrow Heap \in Def\ P\ n$
| *Def-Invoke-Return-Heap*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ Return);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n'\ ]\!]$
$\Longrightarrow Heap \in Def\ P\ n$
| *Def-Invoke-Return-Exception*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ Return);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n'\ ]\!]$
$\Longrightarrow Exception \in Def\ P\ n$
| *Def-Invoke-Return-Stack*:
$[\![n = (C,\ M,\ \lfloor pc \rfloor,\ Return);$
$C \neq ClassMain\ P;$
$instrs\text{-}of\ (PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n';$

```
  i = stkLength (P, C, M) (Suc pc) - 1 ⟧
  ⟹ Stack i ∈ Def P n
| Def-Invoke-Call-Heap:
  ⟦ n = (C, M, None, Enter);
  C ≠ ClassMain P ⟧
  ⟹ Heap ∈ Def P n
| Def-Invoke-Call-Local:
  ⟦ n = (C, M, None, Enter);
  C ≠ ClassMain P;
  i < locLength (P, C, M) 0 ⟧
  ⟹ Local i ∈ Def P n
| Def-Return:
  ⟦ n = (C, M, ⌊pc⌋, Enter);
  C ≠ ClassMain P;
  instrs-of (PROG P) C M ! pc = instr.Return ⟧
  ⟹ Stack 0 ∈ Def P n

inductive-set Use :: wf-jvmprog ⇒ cfg-node ⇒ var set
  for P :: wf-jvmprog
  and n :: cfg-node
where
  Use-Main-Heap:
  n = (ClassMain P, MethodMain P, ⌊0⌋, Normal)
  ⟹ Heap ∈ Use P n
| Use-Load:
  ⟦ n = (C, M, ⌊pc⌋, Enter);
  C ≠ ClassMain P;
  instrs-of (PROG P) C M ! pc = Load idx ⟧
  ⟹ Local idx ∈ Use P n
| Use-Enter-Stack:
  ⟦ n = (C, M, ⌊pc⌋, Enter);
  C ≠ ClassMain P;
  case (instrs-of (PROG P) C M ! pc)
    of Store n' ⇒ d = 1
    | Getfield F Cl ⇒ d = 1
    | Putfield F Cl ⇒ d = 2
    | Checkcast Cl ⇒ d = 1
    | Invoke M' n' ⇒ d = Suc n'
    | IAdd ⇒ d ∈ {1, 2}
    | IfFalse i ⇒ d = 1
    | CmpEq ⇒ d ∈ {1 , 2}
    | Throw ⇒ d = 1
    | instr.Return ⇒ d = 1
    | - ⇒ False;
  i = stkLength (P, C, M) pc - d ⟧
  ⟹ Stack i ∈ Use P n
| Use-Enter-Local:
  ⟦ n = (C, M, ⌊pc⌋, Enter);
  C ≠ ClassMain P;
```

instrs-of (*PROG P*) *C M* ! *pc* = *Load n'* ⟧
⟹ *Local n'* ∈ *Use P n*
| *Use-Enter-Heap*:
⟦ *n* = (*C*, *M*, ⌊*pc*⌋, *Enter*);
C ≠ *ClassMain P*;
case (*instrs-of* (*PROG P*) *C M* ! *pc*)
of New Cl ⇒ *True*
| *Checkcast Cl* ⇒ *True*
| *Throw* ⇒ *True*
| - ⇒ *False* ⟧
⟹ *Heap* ∈ *Use P n*
| *Use-Normal-Heap*:
⟦ *n* = (*C*, *M*, ⌊*pc*⌋, *Normal*);
C ≠ *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 ≠ ClassMain P ⟧
  ⟹ Stack 0 ∈ Use P n
| Use-Method-Entry-Heap:
  ⟦ n = (C, M, None, Enter);
  C ≠ ClassMain P ⟧
  ⟹ Heap ∈ Use P n
| Use-Method-Entry-Local:
  ⟦ n = (C, M, None, Enter);
  C ≠ ClassMain P;
  i < locLength (P, C, M) 0 ⟧
  ⟹ Local i ∈ 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 ≠ MethodMain P ∨ opc ≠ ⌊0⌋ ∨ nt ≠ Normal
  ⟹ ParamUses P (ClassMain P, M, opc, nt) = []
  |
  C ≠ 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 (λn. {Stack (stkLength (P, C, M) pc − (Suc n))})
[0..<n + 1])
      )
```

```
    | - ⇒ [])
   | - ⇒ []
   )
 )
 by atomize-elim auto
termination by lexicographic-order

lemma in-set-ParamDefsE:
 ⟦ V ∈ set (ParamDefs P n);
 ⋀C M pc. ⟦ n = (C, M, ⌊pc⌋, Return);
       V ∈ {Heap, Stack (stkLength (P, C, M) (Suc pc) − 1), Exception} ⟧ ⟹
thesis ⟧
 ⟹ thesis
 by (cases (P, n) rule: ParamDefs.cases) auto

lemma in-set-ParamUsesE:
 assumes V-in-ParamUses: V ∈ ⋃ set (ParamUses P n)
 obtains n = (ClassMain P, MethodMain P, ⌊0⌋, Normal) and V = Heap
 | C M pc M' n' i where n = (C, M, ⌊pc⌋, Normal) and instrs-of (PROG P)
C M ! pc = Invoke M' n'
   and V = Heap ∨ V = Stack (stkLength (P, C, M) pc − Suc i) and i < Suc
n' and C ≠ ClassMain P
proof (cases (P, n) rule: ParamUses.cases)
 case 1 with V-in-ParamUses that show ?thesis by clarsimp
next
 case 2 with V-in-ParamUses that show ?thesis by clarsimp
next
 case (3 C P M pc nt)
 with V-in-ParamUses that show ?thesis
   using [[simproc del: list-to-set-comprehension]] by (cases nt, auto) (case-tac
instrs-of (PROG P) C M ! a, simp-all, fastforce)
qed

lemma sees-method-fun-wf:
 assumes PROG P ⊢ D sees M': Ts→T = (mxs, mxl₀, is, xt) in D
 and (D, D', fs, ms) ∈ set (PROG P)
 and (M', Ts', T', mxs', mxl₀', is', xt') ∈ set ms
 shows Ts = Ts' ∧ T = T' ∧ mxs = mxs' ∧ mxl₀ = mxl₀' ∧ is = is' ∧ xt = xt'
proof −
 from distinct-class-names [of P] ‹(D, D', fs, ms) ∈ set (PROG P)›
 have class (PROG P) D = ⌊(D', fs, ms)⌋
   by (fastforce intro: map-of-SomeI simp: class-def)
 moreover with distinct-method-names have distinct-fst ms
   by fastforce
 ultimately show ?thesis using
   ‹PROG P ⊢ D sees M': Ts→T = (mxs, mxl₀, is, xt) in D›
   ‹(M', Ts', T', mxs', mxl₀', is', xt') ∈ set ms›
   by (fastforce dest: visible-method-exists map-of-SomeD distinct-fst-isin-same-fst
     simp: distinct-fst-is-distinct-fst)
```

qed

interpretation *JVMCFG-wf*:
CFG-wf sourcenode targetnode kind valid-edge (*P*, *C0*, *Main*)
(*ClassMain P*, *MethodMain P*, *None*, *Enter*)
(λ(*C*, *M*, *pc*, *type*). (*C*, *M*)) *get-return-edges P*
((*ClassMain P*, *MethodMain P*),[],[]) # *procs* (*PROG P*)
(*ClassMain P*, *MethodMain P*)
Def P Use P ParamDefs P ParamUses P
for *P C0 Main*
proof (*unfold-locales*)
show *Def P* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) = {} ∧
Use P (*ClassMain P*, *MethodMain P*, *None*, *Enter*) = {}
by (*fastforce elim*: *Def.cases Use.cases*)
next
fix *a Q r p fs ins outs*
assume *valid-edge* (*P*, *C0*, *Main*) *a*
and *kind a* = *Q*:*r*↪*pfs*
and *params*: (*p*, *ins*, *outs*) ∈ *set* (((*ClassMain P*, *MethodMain P*), [], []) # *procs* (*PROG P*))
hence (*P*, *C0*, *Main*) ⊢ *sourcenode a* −*Q*:*r*↪*pfs*→ *targetnode a*
by (*simp add*: *valid-edge-def*)
from *this params* **show** *length* (*ParamUses P* (*sourcenode a*)) = *length ins*
proof *cases*
case *Main-Call*
with *params* **show** *?thesis*
by *auto* (*erule in-set-procsE*, *auto dest*: *sees-method-idemp sees-method-fun*)
next
case (*CFG-Invoke-Call C M pc M' n ST LT D' Ts T mxs mxl0 is xt D Q' paramDefs*)
hence [*simp*]: *Q'* = *Q* **and** [*simp*]: *r* = (*C*, *M*, *pc*) **and** [*simp*]: *p* = (*D*, *M'*)
and [*simp*]: *fs* = *paramDefs*
by *simp-all*
from *CFG-Invoke-Call* **obtain** *T' mxs' mpc' xt'* **where**
PROG P,*T'*,*mxs'*,*mpc'*,*xt'* ⊢ *instrs-of* (*PROG P*) *C M* ! *pc*,*pc* :: *TYPING P C M*
by (*blast dest*: *reachable-node-impl-wt-instr*)
moreover from ‹*PROG P* ⊢ *D' sees M'*: *Ts*→*T* = (*mxs*, *mxl0*, *is*, *xt*) *in D*›
have *PROG P* ⊢ *D sees M'*: *Ts*→*T* = (*mxs*, *mxl0*, *is*, *xt*) *in D*
by −(*drule sees-method-idemp*)
with *params* **have** *PROG P* ⊢ *D sees M'*: *Ts*→*T*=(*mxs*, *mxl0*, *is*, *xt*) *in D*
and *ins* = *Heap* # *map Local* [*0*..<*Suc* (*length Ts*)]
by (*fastforce elim*: *in-set-procsE dest*: *sees-method-fun*)+
ultimately show *?thesis* **using** *CFG-Invoke-Call*
by (*fastforce dest*: *sees-method-fun list-all2-lengthD simp*: *min-def*)
qed *simp-all*
next
fix *a*
assume *valid-edge* (*P*, *C0*, *Main*) *a*

```
  thus distinct (ParamDefs P (targetnode a))
    by (clarsimp simp: valid-edge-def) (erule JVMCFG.cases, auto)
next
  fix a Q' p f' ins outs
  assume valid-edge (P, C0, Main) a
    and kind a = Q'↩pf'
    and params: (p, ins, outs) ∈ set (((ClassMain P, MethodMain P), [], []) #
procs (PROG P))
  hence (P, C0, Main) ⊢ sourcenode a -Q'↩pf'→ targetnode a
    by (simp add: valid-edge-def)
  from this params
  show length (ParamDefs P (targetnode a)) = length outs
    by cases (auto elim: in-set-procsE)
next
  fix n V
  assume params: V ∈ set (ParamDefs P n)
    and vn: CFG.valid-node sourcenode targetnode (valid-edge (P, C0, Main)) n
  then obtain ek n'
   where ve:valid-edge (P, C0, Main) (n, ek, n') ∨ valid-edge (P, C0, Main) (n',
ek, n)
    by (fastforce simp: JVMCFG-Interpret.valid-node-def)
  from params obtain C M pc where [simp]: n = (C, M, ⌊pc⌋, Return)
    and V: V ∈ {Heap, Stack (stkLength (P, C, M) (Suc pc) - 1), Exception}
    by (blast elim: in-set-ParamDefsE)
  from ve show V ∈ Def P n
  proof
    assume valid-edge (P, C0, Main) (n, ek, n')
    thus ?thesis unfolding valid-edge-def
    proof cases
      case Main-Return-to-Exit with V show ?thesis
       by (auto intro: Def-Main-Heap Def-Main-Stack-0 Def-Main-Exception simp:
stkLength-def)
    next
      case CFG-Invoke-Return-Check-Normal with V show ?thesis
        by (fastforce intro: Def-Invoke-Return-Heap
          Def-Invoke-Return-Stack Def-Invoke-Return-Exception)
    next
      case CFG-Invoke-Return-Check-Exceptional with V show ?thesis
        by (fastforce intro: Def-Invoke-Return-Heap
          Def-Invoke-Return-Stack Def-Invoke-Return-Exception)
    next
      case CFG-Invoke-Return-Exceptional-prop with V show ?thesis
        by (fastforce intro: Def-Invoke-Return-Heap
          Def-Invoke-Return-Stack Def-Invoke-Return-Exception)
    qed simp-all
  next
    assume valid-edge (P, C0, Main) (n', ek, n)
    thus ?thesis unfolding valid-edge-def
    proof cases
```

```
    case Main-Call-LFalse with V show ?thesis
     by (auto intro: Def-Main-Heap Def-Main-Stack-0 Def-Main-Exception simp:
stkLength-def)
   next
    case CFG-Invoke-False with V show ?thesis
     by (fastforce intro: Def-Invoke-Return-Heap
       Def-Invoke-Return-Stack Def-Invoke-Return-Exception)
   next
    case CFG-Return-from-Method with V show ?thesis
     by (fastforce elim!: JVMCFG.cases intro!: Def-Main-Stack-0
       intro: Def-Main-Heap Def-Main-Exception Def-Invoke-Return-Heap
      Def-Invoke-Return-Exception Def-Invoke-Return-Stack simp: stkLength-def)
   qed simp-all
  qed
next
  fix a Q r p fs ins outs V
  assume ve: valid-edge (P, C0, Main) a
    and kind: kind a = Q:r↪pfs
    and params: (p, ins, outs) ∈ set (((ClassMain P, MethodMain P), [], []) #
procs (PROG P))
    and V: V ∈ set ins
  from params V obtain D fs ms Ts T mb where class (PROG P) (fst p) = ⌊(D,
fs, ms)⌋
    and method: PROG P ⊢ (fst p) sees (snd p): Ts→T = mb in (fst p)
    and ins: ins = Heap # map Local [0..<Suc (length Ts)]
    by (cases p) (fastforce elim: in-set-procsE)
  from ve kind show V ∈ Def P (targetnode a) unfolding valid-edge-def
  proof cases
    case (Main-Call T' mxs mxl0 is xt D' initParams)
    with kind have PROG P ⊢ D' sees Main: []→T' = (mxs, mxl0, is, xt) in D'
      and [simp]: p = (D', Main)
      by (auto dest: sees-method-idemp)
     with method have [simp]: Ts = [] and [simp]: T' = T and [simp]: mb =
(mxs, mxl0, is, xt)
      by (fastforce dest: sees-method-fun)+
    from Main-Call ins V show ?thesis
      by (fastforce intro!: Def-Invoke-Call-Heap Def-Invoke-Call-Local
        dest: sees-method-idemp wt-jvm-prog-impl-wt-start[OF wf-jvmprog-is-wf-typ]
        simp: locLength-def wt-start-def)
  next
    case (CFG-Invoke-Call C M pc M' n ST LT D' Ts' T' mxs mxl0 is xt D'')
    with kind have PROG P ⊢ D'' sees M': Ts'→T' = (mxs, mxl0, is, xt) in D''
      and [simp]: p = (D'', M')
      by (auto dest: sees-method-idemp)
     with method have [simp]: Ts' = Ts and [simp]: T' = T and [simp]: mb =
(mxs, mxl0, is, xt)
      by (fastforce dest: sees-method-fun)+
    from CFG-Invoke-Call ins V show ?thesis
      by (fastforce intro!: Def-Invoke-Call-Local Def-Invoke-Call-Heap
```

```
      dest: sees-method-idemp wt-jvm-prog-impl-wt-start[OF wf-jvmprog-is-wf-typ]
list-all2-lengthD
      simp: locLength-def min-def wt-start-def)
  qed simp-all
next
  fix a Q r p fs
  assume valid-edge (P, C0, Main) a and kind a = Q:r↪pfs
  thus Def P (sourcenode a) = {} unfolding valid-edge-def
    by cases (auto elim: Def.cases)
next
  fix n V
  assume CFG.valid-node sourcenode targetnode (valid-edge (P, C0, Main)) n
    and V: V ∈ ⋃ set (ParamUses P n)
  then obtain ek n'
    where ve:valid-edge (P, C0, Main) (n, ek, n') ∨ valid-edge (P, C0, Main) (n',
ek, n)
    by (fastforce simp: JVMCFG-Interpret.valid-node-def)
  from V obtain C M pc M' n'' i where
    V: n = (ClassMain P, MethodMain P, ⌊0⌋, Normal) ∧ V = Heap ∨
    n = (C, M, ⌊pc⌋, Normal) ∧ instrs-of (PROG P) C M ! pc = Invoke M' n''
∧
      (V = Heap ∨ V = Stack (stkLength (P, C, M) pc − Suc i)) ∧ i < Suc n''
∧ C ≠ ClassMain P
    by −(erule in-set-ParamUsesE, fastforce+)
  from ve show V ∈ Use P n
  proof
    assume valid-edge (P, C0, Main) (n, ek, n')
    from this V show ?thesis unfolding valid-edge-def
    proof cases
    case Main-Call-LFalse with V show ?thesis by (fastforce intro: Use-Main-Heap)
    next
      case Main-Call with V show ?thesis by (fastforce intro: Use-Main-Heap)
    next
      case CFG-Invoke-Call with V show ?thesis
        by (fastforce intro: Use-Normal-Heap Use-Normal-Stack [where d=Suc i])
    next
      case CFG-Invoke-False with V show ?thesis
        by (fastforce intro: Use-Normal-Heap Use-Normal-Stack [where d=Suc i])
    qed simp-all
  next
    assume valid-edge (P, C0, Main) (n', ek, n)
    from this V show ?thesis unfolding valid-edge-def
    proof cases
     case Main-to-Call with V show ?thesis by (fastforce intro: Use-Main-Heap)
    next
      case CFG-Invoke-Check-NP-Normal with V show ?thesis
        by (fastforce intro: Use-Normal-Heap Use-Normal-Stack [where d=Suc i])
    qed simp-all
  qed
```

next
fix *a Q p f ins outs V*
assume *valid-edge* (*P*, *C0*, *Main*) *a*
and *kind a* = *Q*↪*pf*
and (*p*, *ins*, *outs*) ∈ *set* (((*ClassMain P*, *MethodMain P*), [], []) # *procs* (*PROG P*))
and *V* ∈ *set outs*
thus *V* ∈ *Use P* (*sourcenode a*) **unfolding** *valid-edge-def*
by (*cases*, *simp-all*)
(*fastforce elim*: *in-set-procsE*
intro: *Use-Method-Leave-Heap Use-Method-Leave-Stack Use-Method-Leave-Exception*)
next
fix *a V s*
assume *ve*: *valid-edge* (*P*, *C0*, *Main*) *a*
and *V-notin-Def*: *V* ∉ *Def P* (*sourcenode a*)
and *ik*: *intra-kind* (*kind a*)
and *pred*: *JVMCFG-Interpret.pred* (*kind a*) *s*
show *JVMCFG-Interpret.state-val*
(*CFG.transfer* (((*ClassMain P*, *MethodMain P*), [], []) # *procs* (*PROG P*)) (*kind a*) *s*) *V*
= *JVMCFG-Interpret.state-val s V*
proof (*cases s*)
case *Nil*
thus *?thesis* **by** *simp*
next
case *Cons* [*simp*]
with *ve V-notin-Def ik pred* **show** *?thesis* **unfolding** *valid-edge-def*
proof *cases*
case *CFG-Load* **with** *V-notin-Def* **show** *?thesis* **by** (*fastforce intro*: *Def-Load*)
next case *CFG-Store* **with** *V-notin-Def* **show** *?thesis* **by** (*fastforce intro*: *Def-Store*)
next case *CFG-Push* **with** *V-notin-Def* **show** *?thesis* **by** (*fastforce intro*: *Def-Push*)
next case *CFG-IAdd* **with** *V-notin-Def* **show** *?thesis* **by** (*fastforce intro*: *Def-IAdd*)
next case *CFG-CmpEq* **with** *V-notin-Def* **show** *?thesis* **by** (*fastforce intro*: *Def-CmpEq*)
next case *CFG-New-Update* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-New-Heap Def-New-Stack*)
next case *CFG-New-Exceptional-prop* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Exception*)
next case *CFG-New-Exceptional-handle* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Exception Def-Exception-handle*)
next case *CFG-Getfield-Update* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Getfield split*: *split-split*)
next case *CFG-Getfield-Exceptional-prop* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Exception*)
next case *CFG-Getfield-Exceptional-handle* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Exception Def-Exception-handle*)

next case *CFG-Putfield-Update* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Putfield split*: *split-split*)
next case *CFG-Putfield-Exceptional-prop* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Exception*)
next case *CFG-Putfield-Exceptional-handle* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Exception Def-Exception-handle*)
next case *CFG-Checkcast-Exceptional-prop* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Exception*)
next case *CFG-Checkcast-Exceptional-handle* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Exception Def-Exception-handle*)
next case *CFG-Throw-prop* **with** *V-notin-Def* **show** *?thesis* **by** (*fastforce intro*: *Def-Exception*)
next case *CFG-Throw-handle* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Exception Def-Exception-handle*)
next case *CFG-Invoke-NP-prop* **with** *V-notin-Def* **show** *?thesis* **by** (*fastforce intro*: *Def-Exception*)
next case *CFG-Invoke-NP-handle* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Exception Def-Exception-handle*)
next case *CFG-Invoke-Return-Exceptional-handle* **with** *V-notin-Def* **show** *?thesis*
by (*fastforce intro*: *Def-Exception-handle-return Def-Exception*)
next case *CFG-Return* **with** *V-notin-Def* **show** *?thesis* **by** (*fastforce intro*: *Def-Return*)
qed (*simp-all add*: *intra-kind-def*)
qed
next
fix *a s s′*
assume *ve*: *valid-edge* (*P*, *C0*, *Main*) *a*
and *use-Eq*: ∀ *V*∈*Use P* (*sourcenode a*). *JVMCFG-Interpret.state-val s V* = *JVMCFG-Interpret.state-val s′ V*
and *ik*: *intra-kind* (*kind a*)
and *pred-s*: *JVMCFG-Interpret.pred* (*kind a*) *s*
and *pred-s′*: *JVMCFG-Interpret.pred* (*kind a*) *s′*
then obtain *cfs C M pc cs cfs′ C′ M′ pc′ cs′* **where** [*simp*]: *s* = (*cfs*, (*C*, *M*, *pc*)) # *cs*
and [*simp*]: *s′* = (*cfs′*, (*C′*, *M′*, *pc′*)) # *cs′*
by (*cases s*, *fastforce*) (*cases s′*, *fastforce*+)
from *ve* **show** ∀ *V*∈*Def P* (*sourcenode a*).
JVMCFG-Interpret.state-val
(*CFG.transfer* (((*ClassMain P*, *MethodMain P*), [], []) # *procs* (*PROG P*)) (*kind a*) *s*) *V* =
JVMCFG-Interpret.state-val
(*CFG.transfer* (((*ClassMain P*, *MethodMain P*), [], []) # *procs* (*PROG P*)) (*kind a*) *s′*) *V*
unfolding *valid-edge-def*
proof *cases*
case *Main-Call* **with** *ik* **show** *?thesis* **by** (*simp add*: *intra-kind-def*)
next case *Main-Return-to-Exit* **with** *use-Eq* **show** *?thesis*
by (*fastforce elim*: *Def.cases intro*: *Use-Return-Heap Use-Return-Exception*

Use-Return-Stack)
next case *Method-LFalse* **with** *use-Eq* **show** *?thesis*
by (*fastforce elim*: *Def.cases intro*: *Use-Method-Entry-Heap Use-Method-Entry-Local*)

next case *Method-LTrue* **with** *use-Eq* **show** *?thesis*
by (*fastforce elim*: *Def.cases intro*: *Use-Method-Entry-Heap Use-Method-Entry-Local*)
next case *CFG-Load* **with** *use-Eq* **show** *?thesis*
by (*fastforce elim*: *Def.cases intro*: *Use-Enter-Local*)
next case *CFG-Store* **with** *use-Eq* **show** *?thesis*
by (*fastforce elim*: *Def.cases intro*: *Use-Enter-Stack*)
next case (*CFG-IAdd C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
and *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *2*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack*)+
with *use-Eq CFG-IAdd* **show** *?thesis* **by** (*auto elim*!: *Def.cases*)
next case (*CFG-CmpEq C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
and *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *2*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack*)+
with *use-Eq CFG-CmpEq* **show** *?thesis* **by** (*auto elim*!: *Def.cases*)
next case *CFG-New-Update*
hence *Heap* ∈ *Use P* (*sourcenode a*) **by** (*fastforce intro*: *Use-Normal-Heap*)
with *use-Eq CFG-New-Update* **show** *?thesis* **by** (*fastforce elim*: *Def.cases*)
next case (*CFG-Getfield-Update C M pc*)
hence *Heap* ∈ *Use P* (*sourcenode a*)
and *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Normal-Heap Use-Normal-Stack*)+
with *use-Eq CFG-Getfield-Update* **show** *?thesis* **by** (*auto elim*!: *Def.cases split*: *split-split*)
next case (*CFG-Putfield-Update C M pc*)
hence *Heap* ∈ *Use P* (*sourcenode a*)
and *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
and *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *2*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Normal-Heap Use-Normal-Stack*)+
with *use-Eq CFG-Putfield-Update* **show** *?thesis* **by** (*auto elim*!: *Def.cases split*: *split-split*)
next case (*CFG-Throw-prop C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Exceptional-Stack*)
with *use-Eq CFG-Throw-prop* **show** *?thesis* **by** (*fastforce elim*: *Def.cases*)
next case (*CFG-Throw-handle C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Exceptional-Stack*)
with *use-Eq CFG-Throw-handle* **show** *?thesis* **by** (*fastforce elim*: *Def.cases*)
next case *CFG-Invoke-Call* **with** *ik* **show** *?thesis* **by** (*simp add*: *intra-kind-def*)
next case *CFG-Invoke-Return-Check-Normal* **with** *use-Eq* **show** *?thesis*
by (*fastforce elim*: *Def.cases intro*: *Use-Return-Heap Use-Return-Exception Use-Return-Stack*)
next case *CFG-Invoke-Return-Check-Exceptional* **with** *use-Eq* **show** *?thesis*

by (*fastforce elim*: *Def.cases intro*: *Use-Return-Heap Use-Return-Exception Use-Return-Stack*)
next case *CFG-Invoke-Return-Exceptional-handle* **with** *use-Eq* **show** *?thesis*
by (*fastforce elim*: *Def.cases intro*: *Use-Exceptional-Exception*)
next case *CFG-Invoke-Return-Exceptional-prop* **with** *use-Eq* **show** *?thesis*
by (*fastforce elim*: *Def.cases intro*: *Use-Return-Heap Use-Return-Exception Use-Return-Stack*)
next case *CFG-Return* **with** *use-Eq* **show** *?thesis*
by (*fastforce elim*!: *Def.cases intro*: *Use-Enter-Stack*)
next case *CFG-Return-from-Method* **with** *ik* **show** *?thesis* **by** (*simp add*: *intra-kind-def*)
qed (*fastforce elim*: *Def.cases*)+
next
fix *a s s'*
assume *ve*: *valid-edge* (*P*, *C0*, *Main*) *a*
and *pred*: *JVMCFG-Interpret.pred* (*kind a*) *s*
and *snd* (*hd s*) = *snd* (*hd s'*)
and *use-Eq*: ∀ *V*∈*Use P* (*sourcenode a*).
JVMCFG-Interpret.state-val s V = *JVMCFG-Interpret.state-val s' V*
and *length s* = *length s'*
then obtain *cfs C M pc cs cfs' cs'* **where** [*simp*]: *s* = (*cfs*, (*C*, *M*, *pc*)) # *cs*
and [*simp*]: *s'* = (*cfs'*, (*C*, *M*, *pc*)) # *cs'* **and** *length-cs*: *length cs* = *length cs'*
by (*cases s*, *fastforce*) (*cases s'*, *fastforce*+)
from *ve pred* **show** *JVMCFG-Interpret.pred* (*kind a*) *s'*
unfolding *valid-edge-def*
proof *cases*
case *Main-Call-LFalse* **with** *pred* **show** *?thesis* **by** *simp*
next case *Main-Call* **with** *pred use-Eq* **show** *?thesis* **by** *simp*
next case *Method-LTrue* **with** *pred use-Eq* **show** *?thesis* **by** *simp*
next case *CFG-Goto* **with** *pred use-Eq* **show** *?thesis* **by** *simp*
next case (*CFG-IfFalse-False C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack*)
with *use-Eq CFG-IfFalse-False pred* **show** *?thesis* **by** *fastforce*
next case (*CFG-IfFalse-True C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack*)
with *pred use-Eq CFG-IfFalse-True* **show** *?thesis* **by** *fastforce*
next case *CFG-New-Check-Normal*
hence *Heap* ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Heap*)
with *pred use-Eq CFG-New-Check-Normal* **show** *?thesis* **by** *fastforce*
next case *CFG-New-Check-Exceptional*
hence *Heap* ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Heap*)
with *pred use-Eq CFG-New-Check-Exceptional* **show** *?thesis* **by** *fastforce*
next case (*CFG-Getfield-Check-Normal C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack*)
with *pred use-Eq CFG-Getfield-Check-Normal* **show** *?thesis* **by** *fastforce*

next case (*CFG-Getfield-Check-Exceptional C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack*)
with *pred use-Eq CFG-Getfield-Check-Exceptional* **show** *?thesis* **by** *fastforce*
next case (*CFG-Putfield-Check-Normal C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *2*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack*)
with *pred use-Eq CFG-Putfield-Check-Normal* **show** *?thesis* **by** *fastforce*
next case (*CFG-Putfield-Check-Exceptional C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *2*) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack*)
with *pred use-Eq CFG-Putfield-Check-Exceptional* **show** *?thesis* **by** *fastforce*
next case (*CFG-Checkcast-Check-Normal C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
and *Heap* ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack Use-Enter-Heap*)+
with *pred use-Eq CFG-Checkcast-Check-Normal* **show** *?thesis* **by** *fastforce*
next case (*CFG-Checkcast-Check-Exceptional C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
and *Heap* ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack Use-Enter-Heap*)+
with *pred use-Eq CFG-Checkcast-Check-Exceptional* **show** *?thesis* **by** *fastforce*
next case (*CFG-Throw-Check C M pc*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ∈ *Use P* (*sourcenode a*)
and *Heap* ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack Use-Enter-Heap*)+
with *pred use-Eq CFG-Throw-Check* **show** *?thesis* **by** *fastforce*
next case (*CFG-Invoke-Check-NP-Normal C M pc M′ n*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − (*Suc n*)) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack*)
with *pred use-Eq CFG-Invoke-Check-NP-Normal* **show** *?thesis* **by** *fastforce*
next case (*CFG-Invoke-Check-NP-Exceptional C M pc M′ n*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − (*Suc n*)) ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Enter-Stack*)
with *pred use-Eq CFG-Invoke-Check-NP-Exceptional* **show** *?thesis* **by** *fastforce*
next case (*CFG-Invoke-Call C M pc M′ n*)
hence *Stack* (*stkLength* (*P*, *C*, *M*) *pc* − (*Suc n*)) ∈ *Use P* (*sourcenode a*)
and *Heap* ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Normal-Heap Use-Normal-Stack*)+
with *pred use-Eq CFG-Invoke-Call* **show** *?thesis* **by** *fastforce*
next case *CFG-Invoke-Return-Check-Normal*
hence *Exception* ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Return-Exception*)
with *pred use-Eq CFG-Invoke-Return-Check-Normal* **show** *?thesis* **by** *fastforce*
next case *CFG-Invoke-Return-Check-Exceptional*
hence *Exception* ∈ *Use P* (*sourcenode a*) **and** *Heap* ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Return-Exception Use-Return-Heap*)+
with *pred use-Eq CFG-Invoke-Return-Check-Exceptional* **show** *?thesis* **by** *fastforce*

```
  next case CFG-Invoke-Return-Exceptional-prop
    hence Exception ∈ Use P (sourcenode a) and Heap ∈ Use P (sourcenode a)
      by (fastforce intro: Use-Return-Exception Use-Return-Heap)+
   with pred use-Eq CFG-Invoke-Return-Exceptional-prop show ?thesis by fastforce
  next case CFG-Return-from-Method with pred length-cs show ?thesis by clarsimp
  qed auto
next
  fix a Q r p fs ins outs
  assume valid-edge (P, C0, Main) a
    and kind: kind a = Q:r↪pfs
     and params: (p, ins, outs) ∈ set (((ClassMain P, MethodMain P), [], []) #
procs (PROG P))
  thus length fs = length ins unfolding valid-edge-def
  proof cases
    case (Main-Call  T mxs mxl0 is xt D)
    with kind params have [simp]: p = (D, Main)
      and PROG P ⊢ D sees Main: []→T = (mxs, mxl0, is, xt) in D
      and ins = Heap # map Local [0..<Suc 0]
      by (auto elim!: in-set-procsE dest: sees-method-fun sees-method-idemp)
    with Main-Call kind show ?thesis
      by auto
  next
    case (CFG-Invoke-Call C  M pc M' n ST LT D' Ts T mxs mxl0 is xt D)
    with kind params have [simp]: p = (D, M')
      and PROG P ⊢ D' sees M': Ts→T = (mxs, mxl0, is, xt) in D
      and ins = Heap # map Local [0..<Suc (length Ts)]
      by (auto elim!: in-set-procsE dest: sees-method-fun sees-method-idemp)
    moreover with ‹(P, C0, Main) ⊢ ⇒(C, M, ⌊pc⌋, Normal)› ‹C ≠ ClassMain
P›
      ‹instrs-of (PROG P) C M ! pc = Invoke M' n› ‹TYPING P C M ! pc =
⌊(ST, LT)⌋›
      ‹ST ! n = Class D'› have n = length Ts
      by (fastforce dest!: reachable-node-impl-wt-instr dest: sees-method-fun list-all2-lengthD)
    ultimately show ?thesis using CFG-Invoke-Call kind by auto
  qed simp-all
next
  fix a Q r p fs a' Q' r' p' fs' s s'
  assume ve-a: valid-edge (P, C0, Main) a
    and kind-a: kind a = Q:r↪pfs
    and ve-a': valid-edge (P, C0, Main) a'
    and kind-a': kind a' = Q':r'↪p'fs'
    and src: sourcenode a = sourcenode a'
    and pred-s: JVMCFG-Interpret.pred (kind a) s
    and pred-s': JVMCFG-Interpret.pred (kind a') s
  then obtain cfs C M pc cs cfs' C' M' pc' cs'
    where [simp]: s = (cfs, (C, M, pc)) # cs
    by (cases s) fastforce+
  with ve-a kind-a show a = a' unfolding valid-edge-def
  proof cases
```

```
    case Main-Call with ve-a' kind-a' src pred-s pred-s' show ?thesis unfolding
valid-edge-def
      by (cases a, cases a') (fastforce elim: JVMCFG.cases dest: sees-method-fun)
  next
    case CFG-Invoke-Call
    note invoke-call1 = this
    from ve-a' kind-a' show ?thesis unfolding valid-edge-def
    proof cases
      case Main-Call with CFG-Invoke-Call src have False by simp
      thus ?thesis by simp
    next
      case CFG-Invoke-Call with src invoke-call1 show ?thesis
        by clarsimp (cases a, cases a', fastforce dest: sees-method-fun)
    qed simp-all
  qed simp-all
next
  fix a Q r p fs i ins outs s s'
  assume ve: valid-edge (P, C0, Main) a
    and kind: kind a = Q:r↪pfs
    and i < length ins
   and (p, ins, outs) ∈ set (((ClassMain P, MethodMain P), [], []) # procs (PROG
P))
    and JVMCFG-Interpret.pred (kind a) s
    and JVMCFG-Interpret.pred (kind a) s'
    and use-Eq: ∀ V∈ParamUses P (sourcenode a) ! i.
          JVMCFG-Interpret.state-val s V = JVMCFG-Interpret.state-val s' V
  then obtain cfs C M pc cs cfs' C' M' pc' cs' where [simp]: s = (cfs, (C, M,
pc)) # cs
    and [simp]: s' = (cfs', (C', M', pc')) # cs'
    by (cases s, fastforce) (cases s', fastforce+)
  from ve kind
  show JVMCFG-Interpret.params fs (JVMCFG-Interpret.state-val s) ! i =
       JVMCFG-Interpret.params fs (JVMCFG-Interpret.state-val s') ! i
    unfolding valid-edge-def
  proof cases
    case Main-Call with kind use-Eq ‹i < length ins› show ?thesis
      by (cases i) auto
  next
    case CFG-Invoke-Call
    { fix P C M pc n st st' i
     have ∀ V∈rev (map (λn. {Stack (stkLength (P, C, M) pc − Suc n)}) [0..<n])
! i. st V = st' V
        ⟹ JVMCFG-Interpret.params
        (rev (map (λi s. s (Stack (stkLength (P, C, M) pc − Suc i))) [0..<n])) st
! i =
        JVMCFG-Interpret.params
        (rev (map (λi s. s (Stack (stkLength (P, C, M) pc − Suc i))) [0..<n])) st'
! i
        by (induct n arbitrary: i) (simp, case-tac i, auto)
```

```
    }
    note stack-params = this
    from CFG-Invoke-Call kind use-Eq ‹i < length ins› show ?thesis
      by (cases i, auto) (case-tac nat, auto intro: stack-params)
  qed simp-all
next
  fix a Q' p f' ins outs vmap vmap'
  assume valid-edge (P, C0, Main) a
    and kind a = Q'↪pf'
   and (p, ins, outs) ∈ set (((ClassMain P, MethodMain P), [], []) # procs (PROG
P))
  thus f' vmap vmap' = vmap'(ParamDefs P (targetnode a) [:=] map vmap outs)
    unfolding valid-edge-def
    by (cases, simp-all) (fastforce elim: in-set-procsE simp: fun-upd-twist)
next
  fix a a'
  { fix P n f n' e n''
    assume P ⊢ n -⇑f→ n' and P ⊢ n -e→ n''
    hence e = ⇑f ∧ n' = n''
      by cases (simp-all, (fastforce elim: JVMCFG.cases)+)
  }
  note upd-det = this
  { fix P n Q n' Q' n'' s
   assume P ⊢ n -(Q)√→ n' and edge': P ⊢ n -(Q')√→ n'' and trg: n' ≠ n''
    hence (Q s ⟶ ¬ Q' s) ∧ (Q' s ⟶ ¬ Q s)
    proof cases
      case CFG-Throw-Check with edge' trg show ?thesis by cases fastforce+
    qed (simp-all, (fastforce elim: JVMCFG.cases)+)
  }
  note pred-det = this
  assume valid-edge (P, C0, Main) a
    and ve': valid-edge (P, C0, Main) a'
    and src: sourcenode a = sourcenode a'
    and trg: targetnode a ≠ targetnode a'
    and intra-kind (kind a)
    and intra-kind (kind a')
  thus ∃ Q Q'. kind a = (Q)√ ∧ kind a' = (Q')√ ∧ (∀ s. (Q s ⟶ ¬ Q' s) ∧ (Q'
s ⟶ ¬ Q s))
    unfolding valid-edge-def intra-kind-def
    by (auto dest: upd-det pred-det)
qed

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

Def P Use P ParamDefs P ParamUses P
proof
show *Def P* (*ClassMain P*, *MethodMain P*, *None*, *nodeType.Return*) = {} ∧
Use P (*ClassMain P*, *MethodMain P*, *None*, *nodeType.Return*) = {}
by (*fastforce elim*: *Def.cases Use.cases*)
qed

end

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

context *CFG* **begin**

lemma *vp-snocI*:
⟦*n* −*as*→$_{\surd}$* *n'*; *n'* −[*a*]→* *n''*; ∀ *Q p ret fs*. *kind a* ≠ *Q*↩$_p$*ret* ⟧ ⟹ *n* −*as* @ [*a*]→$_{\surd}$* *n''*
by (*cases kind a*) (*auto intro*: *path-Append valid-path-aux-Append simp*: *vp-def valid-path-def*)

lemma *valid-node-cases'* [*case-names Source Target*, *consumes 1*]:
⟦ *valid-node n*; ⋀*e*. ⟦ *valid-edge e*; *sourcenode e* = *n* ⟧ ⟹ *thesis*;
⋀*e*. ⟦ *valid-edge e*; *targetnode e* = *n* ⟧ ⟹ *thesis* ⟧
⟹ *thesis*
by (*auto simp*: *valid-node-def*)

end

lemma *disjE-strong*: ⟦*P* ∨ *Q*; *P* ⟹ *R*; ⟦*Q*; ¬ *P*⟧ ⟹ *R*⟧ ⟹ *R*
by *auto*

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 (*induct rule*: *JVMCFG-reachable-inducts*)
case (*Entry-reachable P C0 Main*)
hence *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) [] (*ClassMain P*, *MethodMain P*, *None*, *Enter*)
by (*fastforce intro*: *JVMCFG-Interpret.intra-path-vp Method-LTrue*
JVMCFG-reachable.Entry-reachable)
thus *?case* **by** *blast*
next
case (*reachable-step P C0 Main C M pc nt ek C' M' pc' nt'*)
thus *?case* **by** *simp*
next
case (*Main-to-Call P C0 Main*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Enter*)
by *blast*
moreover with ‹(*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Enter*)›
have *vp-snoc P C0 Main as* (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Enter*) ⇑*id*
(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)
by (*fastforce intro*: *JVMCFG-reachable.Main-to-Call*)
ultimately show *?case* **by** *blast*
next
case (*Main-Call-LFalse P C0 Main*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)
by *blast*
moreover with ‹(*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)›
have *vp-snoc P C0 Main as* (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*) (*λs*. *False*)$_{\surd}$
(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Return*)
by (*fastforce intro*: *JVMCFG-reachable.Main-Call-LFalse*)
ultimately show *?case* **by** *blast*
next
case (*Main-Call P C0 Main T mxs mxl*$_0$ *is xt D initParams ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)
by *blast*
moreover with ‹(*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Nor-*

mal)›
‹*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*›
have *vp-snoc P C0 Main as* (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)
((λ(*s*, *ret*). *True*):(*ClassMain P*, *MethodMain P*, *0*)↪$_{(D,\ Main)}$[(λ*s*. *s Heap*),(λ*s*. ⌊*Value Null*⌋)])
(*D*, *Main*, *None*, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.Main-Call*)
ultimately show *?case* **by** *blast*
next
case (*Main-Return-to-Exit P C0 Main*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *nodeType.Return*)
by *blast*
moreover with ‹(*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *nodeType.Return*)›
have *vp-snoc P C0 Main as* (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *nodeType.Return*) ⇑*id*
(*ClassMain P*, *MethodMain P*, *None*, *nodeType.Return*)
by (*fastforce intro*: *JVMCFG-reachable.Main-Return-to-Exit*)
ultimately show *?case* **by** *blast*
next
case (*Method-LFalse P C0 Main C M*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, *None*, *Enter*)
by *blast*
moreover with ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, *None*, *Enter*)›
have *vp-snoc P C0 Main as* (*C*, *M*, *None*, *Enter*) (λ*s*. *False*)$_{\surd}$ (*C*, *M*, *None*, *Return*)
by (*fastforce intro*: *JVMCFG-reachable.Method-LFalse*)
ultimately show *?case* **by** *blast*
next
case (*Method-LTrue P C0 Main C M*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, *None*, *Enter*)
by *blast*
moreover with ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, *None*, *Enter*)›
have *vp-snoc P C0 Main as* (*C*, *M*, *None*, *Enter*) (λ*s*. *True*)$_{\surd}$ (*C*, *M*, ⌊*0*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.Method-LTrue*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Load C P C0 Main M pc n ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)›

```
    ‹instrs-of (PROG P) C M ! pc = Load n›
    ‹ek = ↑λs. s(Stack (stkLength (P, C, M) pc) := s (Local n))›
  have vp-snoc P C0 Main as (C, M, ⌊pc⌋, Enter) ek (C, M, ⌊Suc pc⌋, Enter)
    by (fastforce intro: JVMCFG-reachable.CFG-Load)
  ultimately show ?case by blast
next
  case (CFG-Store C P C0 Main M pc n ek)
  then obtain as where JVMCFG-Interpret.valid-path' P C0 Main
    (ClassMain P, MethodMain P, None, Enter) as (C, M, ⌊pc⌋, Enter)
    by blast
  moreover with ‹C ≠ ClassMain P› ‹(P, C0, Main) ⊢ ⇒(C, M, ⌊pc⌋, Enter)›
    ‹instrs-of (PROG P) C M ! pc = Store n›
    ‹ek = ↑λs. s(Local n := s (Stack (stkLength (P, C, M) pc − 1)))›
  have vp-snoc P C0 Main as (C, M, ⌊pc⌋, Enter) ek (C, M, ⌊Suc pc⌋, Enter)
    by (fastforce intro: JVMCFG-reachable.CFG-Store)
  ultimately show ?case by blast
next
  case (CFG-Push C P C0 Main M pc v ek)
  then obtain as where JVMCFG-Interpret.valid-path' P C0 Main
    (ClassMain P, MethodMain P, None, Enter) as (C, M, ⌊pc⌋, Enter)
    by blast
  moreover with ‹C ≠ ClassMain P› ‹(P, C0, Main) ⊢ ⇒(C, M, ⌊pc⌋, Enter)›
    ‹instrs-of (PROG P) C M ! pc = Push v›
    ‹ek = ↑λs. s(Stack (stkLength (P, C, M) pc) ↦ Value v)›
  have vp-snoc P C0 Main as (C, M, ⌊pc⌋, Enter) ek (C, M, ⌊Suc pc⌋, Enter)
    by (fastforce intro: JVMCFG-reachable.CFG-Push)
  ultimately show ?case by blast
next
  case (CFG-Pop C P C0 Main M pc ek)
  then obtain as where JVMCFG-Interpret.valid-path' P C0 Main
    (ClassMain P, MethodMain P, None, Enter) as (C, M, ⌊pc⌋, Enter)
    by blast
  moreover with ‹C ≠ ClassMain P› ‹(P, C0, Main) ⊢ ⇒(C, M, ⌊pc⌋, Enter)›
    ‹instrs-of (PROG P) C M ! pc = Pop› ‹ek = ↑id›
  have vp-snoc P C0 Main as (C, M, ⌊pc⌋, Enter) ek (C, M, ⌊Suc pc⌋, Enter)
    by (fastforce intro: JVMCFG-reachable.CFG-Pop)
  ultimately show ?case by blast
next
  case (CFG-IAdd C P C0 Main M pc ek)
  then obtain as where JVMCFG-Interpret.valid-path' P C0 Main
    (ClassMain P, MethodMain P, None, Enter) as (C, M, ⌊pc⌋, Enter)
    by blast
  moreover with ‹C ≠ ClassMain P› ‹(P, C0, Main) ⊢ ⇒(C, M, ⌊pc⌋, Enter)›
    ‹instrs-of (PROG P) C M ! pc = IAdd›
    ‹ek = ↑λs. 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)))›
  have vp-snoc P C0 Main as (C, M, ⌊pc⌋, Enter) ek (C, M, ⌊Suc pc⌋, Enter)
    by (fastforce intro: JVMCFG-reachable.CFG-IAdd)
```

ultimately show *?case* **by** *blast*
next
case (*CFG-Goto C P C0 Main M pc i*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Goto i*›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) (λ*s*. *True*)$_{\surd}$ (*C*, *M*, ⌊*nat* (*int pc* + *i*)⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Goto*)
ultimately show *?case* **by** *blast*
next
case (*CFG-CmpEq C P C0 Main M pc ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*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*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) *ek* (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-CmpEq*)
ultimately show *?case* **by** *blast*
next
case (*CFG-IfFalse-False C P C0 Main M pc i ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*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*)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) *ek* (*C*, *M*, ⌊*nat* (*int pc* + *i*)⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-IfFalse-False*)
ultimately show *?case* **by** *blast*
next
case (*CFG-IfFalse-True C P C0 Main M pc i ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*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*)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) *ek* (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-IfFalse-True*)
ultimately show *?case* **by** *blast*
next

case (*CFG-New-Check-Normal C P C0 Main M pc Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*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*)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) *ek* (*C*, *M*, ⌊*pc*⌋, *Normal*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-New-Check-Normal*)
ultimately show *?case* **by** *blast*
next
case (*CFG-New-Check-Exceptional C P C0 Main M pc Cl pc' ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*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*
| ⌊(*pc''*, *d*)⌋ ⇒ ⌊*pc''*⌋)› ‹*ek* = (λ*s*. *new-Addr* (*heap-of s*) = *None*)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) *ek* (*C*, *M*, ⌊*pc*⌋, *Exceptional pc' Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-New-Check-Exceptional*)
ultimately show *?case* **by** *blast*
next
case (*CFG-New-Update C P C0 Main M pc Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Normal*)
by *blast*
moreover with ‹*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*))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Normal*) *ek* (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-New-Update*)
ultimately show *?case* **by** *blast*
next
case (*CFG-New-Exceptional-prop C P C0 Main M pc Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *New Cl*›
‹*ek* = ⇑λ*s*. *s*(*Exception* ↦ *Value* (*Addr* (*addr-of-sys-xcpt OutOfMemory*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*) *ek* (*C*, *M*, *None*, *Return*)

by (*fastforce intro*: *JVMCFG-reachable.CFG-New-Exceptional-prop*)
ultimately show *?case* **by** *blast*
next
case (*CFG-New-Exceptional-handle C P C0 Main M pc pc′ Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path′ P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *New Cl*›
‹*ek* = ⇑λ*s*. *s*(*Exception* := *None*)(*Stack* (*stkLength* (*P*, *C*, *M*) *pc′* − 1) ↦ *Value* (*Addr* (*addr-of-sys-xcpt OutOfMemory*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*) *ek* (*C*, *M*, ⌊*pc′*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-New-Exceptional-handle*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Getfield-Check-Normal C P C0 Main M pc F Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path′ P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Getfield F Cl*›
‹*ek* = (λ*s*. *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − 1) ≠ *Null*)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) *ek* (*C*, *M*, ⌊*pc*⌋, *Normal*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Getfield-Check-Normal*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Getfield-Check-Exceptional C P C0 Main M pc F Cl pc′ ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path′ P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Getfield F Cl*›
‹*pc′* = (*case match-ex-table* (*PROG P*) *NullPointer pc* (*ex-table-of* (*PROG P*) *C M*) *of None* ⇒ *None*
| ⌊(*pc″*, *d*)⌋ ⇒ ⌊*pc″*⌋)› ‹*ek* = (λ*s*. *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − 1) = *Null*)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) *ek* (*C*, *M*, ⌊*pc*⌋, *Exceptional pc′ Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Getfield-Check-Exceptional*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Getfield-Update C P C0 Main M pc F Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path′ P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Normal*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Normal*)›

‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Getfield F Cl*›
‹*ek* = ⇑λ*s*. *let* (*D*, *fs*) = *the* (*heap-of s* (*the-Addr* (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *1*))))
in s(*Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*) ↦ *Value* (*the* (*fs* (*F*, *Cl*))))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Normal*) *ek* (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Getfield-Update*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Getfield-Exceptional-prop C P C0 Main M pc F Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Getfield F Cl*›
‹*ek* = ⇑λ*s*. *s*(*Exception* ↦ *Value* (*Addr* (*addr-of-sys-xcpt NullPointer*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*) *ek* (*C*, *M*, *None*, *Return*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Getfield-Exceptional-prop*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Getfield-Exceptional-handle C P C0 Main M pc pc' F Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Getfield F Cl*›
‹*ek* = ⇑λ*s*. *s*(*Exception* := *None*)(*Stack* (*stkLength* (*P*, *C*, *M*) *pc'* − *1*) ↦
Value (*Addr* (*addr-of-sys-xcpt NullPointer*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*) *ek* (*C*, *M*, ⌊*pc'*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Getfield-Exceptional-handle*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Putfield-Check-Normal C P C0 Main M pc F Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Putfield F Cl*›
‹*ek* = (λ*s*. *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *2*) ≠ *Null*)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) *ek* (*C*, *M*, ⌊*pc*⌋, *Normal*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Putfield-Check-Normal*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Putfield-Check-Exceptional C P C0 Main M pc F Cl pc' ek*)

then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Putfield F Cl*›
‹*pc'* = (*case match-ex-table* (*PROG P*) *NullPointer pc* (*ex-table-of* (*PROG P*) *C M*) *of None* ⇒ *None*
| ⌊(*pc''*, *d*)⌋ ⇒ ⌊*pc''*⌋)› ‹*ek* = (λ*s*. *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *2*) = *Null*)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) *ek* (*C*, *M*, ⌊*pc*⌋, *Exceptional pc' Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Putfield-Check-Exceptional*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Putfield-Update C P C0 Main M pc F Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Normal*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Normal*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Putfield F Cl*›
‹*ek* = ↑λ*s*. *let v* = *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *1*);
r = *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *2*);
a = *the-Addr r*; (*D*, *fs*) = *the* (*heap-of s a*); *h'* = *heap-of s*(*a* ↦ (*D*, *fs*((*F*, *Cl*) ↦ *v*)))
in s(*Heap* ↦ *Hp h'*)›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Normal*) *ek* (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Putfield-Update*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Putfield-Exceptional-prop C P C0 Main M pc F Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Putfield F Cl*›
‹*ek* = ↑λ*s*. *s*(*Exception* ↦ *Value* (*Addr* (*addr-of-sys-xcpt NullPointer*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*) *ek* (*C*, *M*, *None*, *Return*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Putfield-Exceptional-prop*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Putfield-Exceptional-handle C P C0 Main M pc pc' F Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Excep-*

tional $\lfloor pc' \rfloor$ *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Putfield F Cl*›
‹*ek* = ↑λ*s*. *s*(*Exception* := *None*)(*Stack* (*stkLength* (*P*, *C*, *M*) *pc'* − 1) ↦ *Value* (*Addr* (*addr-of-sys-xcpt NullPointer*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, $\lfloor pc \rfloor$, *Exceptional* $\lfloor pc' \rfloor$ *Enter*) *ek* (*C*, *M*, $\lfloor pc' \rfloor$, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Putfield-Exceptional-handle*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Checkcast-Check-Normal C P C0 Main M pc Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, $\lfloor pc \rfloor$, *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, $\lfloor pc \rfloor$, *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Checkcast Cl*›
‹*ek* = (λ*s*. *cast-ok* (*PROG P*) *Cl* (*heap-of s*) (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − 1)))$_\surd$›
have *vp-snoc P C0 Main as* (*C*, *M*, $\lfloor pc \rfloor$, *Enter*) *ek* (*C*, *M*, $\lfloor Suc\ pc \rfloor$, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Checkcast-Check-Normal*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Checkcast-Check-Exceptional C P C0 Main M pc Cl pc' ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, $\lfloor pc \rfloor$, *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*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* | $\lfloor (pc'', d) \rfloor$ ⇒ $\lfloor pc'' \rfloor$)›
‹*ek* = (λ*s*. ¬ *cast-ok* (*PROG P*) *Cl* (*heap-of s*) (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − 1)))$_\surd$›
have *vp-snoc P C0 Main as* (*C*, *M*, $\lfloor pc \rfloor$, *Enter*) *ek* (*C*, *M*, $\lfloor pc \rfloor$, *Exceptional pc' Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Checkcast-Check-Exceptional*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Checkcast-Exceptional-prop C P C0 Main M pc Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, $\lfloor pc \rfloor$, *Exceptional None Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, $\lfloor pc \rfloor$, *Exceptional None Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Checkcast Cl*›
‹*ek* = ↑λ*s*. *s*(*Exception* ↦ *Value* (*Addr* (*addr-of-sys-xcpt ClassCast*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, $\lfloor pc \rfloor$, *Exceptional None Enter*) *ek* (*C*, *M*, *None*, *Return*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Checkcast-Exceptional-prop*)

ultimately show *?case* **by** *blast*
next
case (*CFG-Checkcast-Exceptional-handle C P C0 Main M pc pc′ Cl ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path′ P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*)
by *blast*
moreover with ‹*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*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc′*⌋ *Enter*) *ek* (*C*, *M*, ⌊*pc′*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Checkcast-Exceptional-handle*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Throw-Check C P C0 Main M pc pc′ Exc d ek*)
then obtain *as* **where** *path-src*: *JVMCFG-Interpret.valid-path′ P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
from ‹*pc′* = *None* ∨ *match-ex-table* (*PROG P*) *Exc pc* (*ex-table-of* (*PROG P*) *C M*) = ⌊(*the pc′*, *d*)⌋›
show *?case*
proof (*elim disjE-strong*)
assume *pc′* = *None*
with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Throw*›
‹*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*
| ⌊*pc″*⌋ ⇒
∃ *d*. *match-ex-table* (*PROG P*) *Cl pc* (*ex-table-of* (*PROG P*) *C M*) = ⌊(*pc″*, *d*)⌋)$_{\surd}$›
have (*P*, *C0*, *Main*) ⊢ (*C*, *M*, ⌊*pc*⌋, *Enter*) −
(λ*s*. (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc 0*) = *Null* ⟶
match-ex-table (*PROG P*) *NullPointer pc* (*ex-table-of* (*PROG P*) *C M*) = *None*) ∧
(*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc 0*) ≠ *Null* ⟶
match-ex-table (*PROG P*) (*cname-of* (*heap-of s*)
(*the-Addr* (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc 0*)))) *pc* (*ex-table-of* (*PROG P*) *C M*) =
None))$_{\surd}$→ (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*)
by −(*erule JVMCFG-reachable.CFG-Throw-Check*, *simp-all*)
with *path-src* ‹*pc′* = *None*› ‹*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
        | ⌊pc''⌋ ⇒
            ∃ d. match-ex-table (PROG P) Cl pc (ex-table-of (PROG P) C M) =
⌊(pc'', d)⌋)_√›
   have vp-snoc P C0 Main as (C, M, ⌊pc⌋, Enter) ek (C, M, ⌊pc⌋, Exceptional
None Enter)
     by (fastforce intro: JVMCFG-reachable.CFG-Throw-Check)
   with path-src ‹pc' = None› show ?thesis by blast
 next
   assume met: match-ex-table (PROG P) Exc pc (ex-table-of (PROG P) C M)
= ⌊(the pc', d)⌋
     and pc': pc' ≠ None
   with ‹C ≠ ClassMain P› ‹(P, C0, Main) ⊢ ⇒(C, M, ⌊pc⌋, Enter)›
   ‹instrs-of (PROG P) C M ! pc = Throw›
   ‹ek = (λs. 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
        | ⌊pc''⌋ ⇒
            ∃ d. match-ex-table (PROG P) Cl pc (ex-table-of (PROG P) C M) =
⌊(pc'', d)⌋)_√›
   have (P, C0, Main) ⊢ (C, M, ⌊pc⌋, Enter) −
     (λs. (stkAt s (stkLength (P, C, M) pc − Suc 0) = Null ⟶
                              (∃ d. match-ex-table (PROG P) NullPointer pc
                                    (ex-table-of (PROG P) C M) =
                                   ⌊(the pc', d)⌋)) ∧
                          (stkAt s (stkLength (P, C, M) pc − Suc 0) ≠ Null ⟶
                              (∃ d. match-ex-table (PROG P)
                                    (cname-of (heap-of s)
                                      (the-Addr
                                      (stkAt s (stkLength (P, C, M) pc − Suc 0))))
                                    pc (ex-table-of (PROG P) C M) =
                                   ⌊(the pc', d)⌋)))_√→
     (C, M, ⌊pc⌋, Exceptional ⌊the pc'⌋ Enter)
     by −(rule JVMCFG-reachable.CFG-Throw-Check, simp-all)
   with met pc' path-src ‹ek = (λs. 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
        | ⌊pc''⌋ ⇒
            ∃ d. match-ex-table (PROG P) Cl pc (ex-table-of (PROG P) C M) =
⌊(pc'', d)⌋)_√›
   have vp-snoc P C0 Main as (C, M, ⌊pc⌋, Enter) ek (C, M, ⌊pc⌋, Exceptional
pc' Enter)
     by (fastforce intro: JVMCFG-reachable.CFG-Throw-Check)
   with path-src show ?thesis by blast
 qed
next
```

case (*CFG-Throw-prop C P C0 Main M pc ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*)
by *blast*
moreover with ‹*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*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Exceptional None Enter*) *ek* (*C*, *M*, *None*, *nodeType.Return*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Throw-prop*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Throw-handle C P C0 Main M pc pc' ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*)›
‹*pc'* ≠ *length* (*instrs-of* (*PROG P*) *C M*)› ‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Throw*›
‹*ek* = ↑λ*s*. *s*(*Exception* := *None*)(*Stack* (*stkLength* (*P*, *C*, *M*) *pc'* − *1*) ↦
Value (*stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *1*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Enter*) *ek* (*C*, *M*, ⌊*pc'*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Throw-handle*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Invoke-Check-NP-Normal C P C0 Main M pc M' n ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*›
‹*ek* = (λ*s*. *stkAt s* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc n*) ≠ *Null*)√›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) *ek* (*C*, *M*, ⌊*pc*⌋, *Normal*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Invoke-Check-NP-Normal*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Invoke-Check-NP-Exceptional C P C0 Main M pc M' n pc' ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*›
‹*pc'* = (*case match-ex-table* (*PROG P*) *NullPointer pc* (*ex-table-of* (*PROG P*) *C M*) *of None* ⇒ *None*

| $\lfloor(pc'', d)\rfloor \Rightarrow \lfloor pc''\rfloor$⟩
⟨$ek = (\lambda s.\ stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - Suc\ n) = Null)_{\surd}$⟩
have *vp-snoc P C0 Main as* $(C,\ M,\ \lfloor pc\rfloor,\ Enter)$ *ek* $(C,\ M,\ \lfloor pc\rfloor,\ Exceptional\ pc'\ Enter)$
by (*fastforce intro*: *JVMCFG-reachable.CFG-Invoke-Check-NP-Exceptional*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Invoke-NP-prop C P C0 Main M pc M' n ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* $(C,\ M,\ \lfloor pc\rfloor,\ Exceptional\ None\ Enter)$
by *blast*
moreover with ⟨$C \neq ClassMain\ P$⟩ ⟨$(P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc\rfloor,\ Exceptional\ None\ Enter)$⟩
⟨*instrs-of* $(PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n$⟩
⟨$ek = \Uparrow\lambda s.\ s(Exception \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer)))$⟩
have *vp-snoc P C0 Main as* $(C,\ M,\ \lfloor pc\rfloor,\ Exceptional\ None\ Enter)$ *ek* $(C,\ M,\ None,\ Return)$
by (*fastforce intro*: *JVMCFG-reachable.CFG-Invoke-NP-prop*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Invoke-NP-handle C P C0 Main M pc pc' M' n ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* $(C,\ M,\ \lfloor pc\rfloor,\ Exceptional\ \lfloor pc'\rfloor\ Enter)$
by *blast*
moreover with ⟨$C \neq ClassMain\ P$⟩ ⟨$(P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc\rfloor,\ Exceptional\ \lfloor pc'\rfloor\ Enter)$⟩
⟨*instrs-of* $(PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n$⟩
⟨$ek = \Uparrow\lambda s.\ s(Exception := None)(Stack\ (stkLength\ (P,\ C,\ M)\ pc' - 1) \mapsto Value\ (Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer)))$⟩
have *vp-snoc P C0 Main as* $(C,\ M,\ \lfloor pc\rfloor,\ Exceptional\ \lfloor pc'\rfloor\ Enter)$ *ek* $(C,\ M,\ \lfloor pc'\rfloor,\ Enter)$
by (*fastforce intro*: *JVMCFG-reachable.CFG-Invoke-NP-handle*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Invoke-Call C P C0 Main M pc M' n ST LT D' Ts T mxs* mxl_0 *is xt D Q paramDefs ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main* (*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* $(C,\ M,\ \lfloor pc\rfloor,\ Normal)$
by *blast*
moreover with ⟨$C \neq ClassMain\ P$⟩ ⟨$(P,\ C0,\ Main) \vdash \Rightarrow(C,\ M,\ \lfloor pc\rfloor,\ Normal)$⟩
⟨*instrs-of* $(PROG\ P)\ C\ M\ !\ pc = Invoke\ M'\ n$⟩ ⟨$TYPING\ P\ C\ M\ !\ pc = \lfloor(ST,\ LT)\rfloor$⟩
⟨$ST\ !\ n = Class\ D'$⟩ ⟨$PROG\ P \vdash D'\ sees\ M'\text{:}\ Ts\rightarrow T = (mxs,\ mxl_0,\ is,\ xt)\ in\ D$⟩
⟨$Q = (\lambda(s,\ ret).\ let\ r = stkAt\ s\ (stkLength\ (P,\ C,\ M)\ pc - Suc\ n);$
$C' = cname\text{-}of\ (heap\text{-}of\ s)\ (the\text{-}Addr\ r)\ in\ D = fst\ (method\ (PROG\ P)\ C'\ M'))$⟩

‹*paramDefs* = (λ*s*. *s Heap*) # (λ*s*. *s* (*Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc n*))) #
rev (*map* (λ*i s*. *s* (*Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *Suc i*))) [*0*..<*n*])›
‹*ek* = *Q*:(*C*, *M*, *pc*)↪$_{(D, M')}$*paramDefs*›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Normal*) *ek* (*D*, *M'*, *None*, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Invoke-Call*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Invoke-False C P C0 Main M pc M' n ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Normal*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Normal*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*› ‹*ek* = (λ*s*. *False*)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Normal*) *ek* (*C*, *M*, ⌊*pc*⌋, *Return*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Invoke-False*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Invoke-Return-Check-Normal C P C0 Main M pc M' n ST LT ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *nodeType.Return*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *nodeType.Return*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*› ‹*TYPING P C M* ! *pc* = ⌊(*ST*, *LT*)⌋›
‹*ST* ! *n* ≠ *NT*› ‹*ek* = (λ*s*. *s Exception* = *None*)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Return*) *ek* (*C*, *M*, ⌊*Suc pc*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Invoke-Return-Check-Normal*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Invoke-Return-Check-Exceptional C P C0 Main M pc M' n Exc pc' diff ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *nodeType.Return*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *nodeType.Return*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*›
‹*match-ex-table* (*PROG P*) *Exc pc* (*ex-table-of* (*PROG P*) *C M*) = ⌊(*pc'*, *diff*)⌋›
‹*pc'* ≠ *length* (*instrs-of* (*PROG P*) *C M*)›
‹*ek* = (λ*s*. ∃*v d*. *s Exception* = ⌊*v*⌋ ∧
match-ex-table (*PROG P*) (*cname-of* (*heap-of s*) (*the-Addr* (*the-Value v*))) *pc*
(*ex-table-of* (*PROG P*) *C M*) = ⌊(*pc'*, *d*)⌋)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Return*) *ek* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Return*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Invoke-Return-Check-Exceptional*)
ultimately show *?case* **by** *blast*

next
case (*CFG-Invoke-Return-Exceptional-handle C P C0 Main M pc pc' M' n ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *nodeType.Return*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *nodeType.Return*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*›
‹*ek* = ↑λ*s*. *s*(*Exception* := *None*, *Stack* (*stkLength* (*P*, *C*, *M*) *pc'* − *1*) := *s Exception*)›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Exceptional* ⌊*pc'*⌋ *Return*) *ek* (*C*, *M*, ⌊*pc'*⌋, *Enter*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Invoke-Return-Exceptional-handle*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Invoke-Return-Exceptional-prop C P C0 Main M pc M' n ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *nodeType.Return*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *nodeType.Return*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *Invoke M' n*›
‹*ek* = (λ*s*. ∃ *v*. *s Exception* = ⌊*v*⌋ ∧
match-ex-table (*PROG P*) (*cname-of* (*heap-of s*) (*the-Addr* (*the-Value v*))) *pc*
(*ex-table-of* (*PROG P*) *C M*) = *None*)$_{\surd}$›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Return*) *ek* (*C*, *M*, *None*, *Return*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Invoke-Return-Exceptional-prop*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Return C P C0 Main M pc ek*)
then obtain *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C*, *M*, ⌊*pc*⌋, *Enter*)
by *blast*
moreover with ‹*C* ≠ *ClassMain P*› ‹(*P*, *C0*, *Main*) ⊢ ⇒(*C*, *M*, ⌊*pc*⌋, *Enter*)›
‹*instrs-of* (*PROG P*) *C M* ! *pc* = *instr.Return*›
‹*ek* = ↑λ*s*. *s*(*Stack 0* := *s* (*Stack* (*stkLength* (*P*, *C*, *M*) *pc* − *1*)))›
have *vp-snoc P C0 Main as* (*C*, *M*, ⌊*pc*⌋, *Enter*) *ek* (*C*, *M*, *None*, *Return*)
by (*fastforce intro*: *JVMCFG-reachable.CFG-Return*)
ultimately show *?case* **by** *blast*
next
case (*CFG-Return-from-Method P C0 Main C M C' M' pc' Q' ps Q stateUpdate ek*)
from ‹(*P*, *C0*, *Main*) ⊢ (*C'*, *M'*, ⌊*pc'*⌋, *Normal*) −*Q'*:(*C'*, *M'*, *pc'*)↪$_{(C,\ M)}$*ps*→ (*C*, *M*, *None*, *Enter*)›
show *?case*
proof *cases*
case *Main-Call*

with *CFG-Return-from-Method* **obtain** *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)
by *blast*
moreover with *Main-Call* **have** *vp-snoc P C0 Main as* (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)
$(\lambda s.\ False)_{\surd}$ (*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Return*)
by (*fastforce intro*: *Main-Call-LFalse*)
ultimately show *?thesis* **using** *Main-Call CFG-Return-from-Method* **by** *blast*
next
case *CFG-Invoke-Call*
with *CFG-Return-from-Method* **obtain** *as* **where** *JVMCFG-Interpret.valid-path' P C0 Main*
(*ClassMain P*, *MethodMain P*, *None*, *Enter*) *as* (*C'*, *M'*, ⌊*pc'*⌋, *Normal*)
by *blast*
moreover with *CFG-Invoke-Call*
have *vp-snoc P C0 Main as* (*C'*, *M'*, ⌊*pc'*⌋, *Normal*) $(\lambda s.\ False)_{\surd}$ (*C'*, *M'*, ⌊*pc'*⌋, *Return*)
by (*fastforce intro*: *CFG-Invoke-False*)
ultimately show *?thesis* **using** *CFG-Invoke-Call CFG-Return-from-Method* **by** *blast*
qed
qed

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*
by (*simp-all add*: *Object-def NullPointer-def OutOfMemory-def ClassCast-def*)

lemmas *distinct-classes* =
distinct-classes distinct-classes'' distinct-classes'' [*symmetric*]

declare *distinct-classes* [*simp add*]

lemma *i-max-2D*: $i < Suc\ (Suc\ 0) \Longrightarrow i = 0 \vee i = 1$ **by** *auto*

lemma *EP-wf*: *wf-jvm-prog*$_{Phi\text{-}EP}$ *EP*
unfolding *wf-jvm-prog-phi-def wf-prog-def*
proof
show *wf-syscls EP*
by (*simp add*: *EP-def wf-syscls-def SystemClasses-def sys-xcpts-def ObjectC-def NullPointerC-def OutOfMemoryC-def ClassCastC-def*)
next
have *distinct-EP*: *distinct-fst EP*
by (*auto simp*: *EP-def SystemClasses-def ObjectC-def NullPointerC-def OutOfMemoryC-def ClassCastC-def*)
moreover have *classes-wf*:
$\forall\, c{\in}set\ EP.$ *wf-cdecl*
$(\lambda P\ C\ (M,\ Ts,\ T_r,\ mxs,\ mxl_0,\ is,\ xt).$ *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* (*Phi-EP C M*)) *EP c*
proof
fix *C*
assume *C-in-EP*: $C \in set\ EP$
show *wf-cdecl*
$(\lambda P\ C\ (M,\ Ts,\ T_r,\ mxs,\ mxl_0,\ is,\ xt).$ *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* (*Phi-EP C M*)) *EP C*
proof (*cases* $C \in set\ SystemClasses$)
case *True*
thus *?thesis*
by (*auto simp*: *wf-cdecl-def SystemClasses-def ObjectC-def NullPointerC-def OutOfMemoryC-def ClassCastC-def EP-def class-def*)
next
case *False*
with *C-in-EP* **have** $C = (''C'', the\ (class\ EP\ ''C''))$
by (*auto simp*: *EP-def SystemClasses-def class-def*)
thus *?thesis*
by (*auto dest*!: *i-max-2D elim*: *Methods.cases simp*: *wf-cdecl-def class-def EP-def wf-mdecl-def wt-method-def Phi-EP-def wt-start-def check-types-def states-def JVM-SemiType.sl-def SystemClasses-def stk-esl-def upto-esl-def loc-sl-def SemiType.esl-def ObjectC-def SemiType.sup-def Err.sl-def Err.le-def err-def Listn.sl-def Method-def Err.esl-def Opt.esl-def Product.esl-def relevant-entries-def*)
qed
qed
ultimately show $(\forall\, c{\in}set\ EP.$ *wf-cdecl*
$(\lambda P\ C\ (M,\ Ts,\ T_r,\ mxs,\ mxl_0,\ is,\ xt).$ *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* (*Phi-EP C M*)) *EP c*) $\wedge$
distinct-fst EP
by *simp*
qed

lemma [*simp*]: *PROG* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)) = *EP*
proof (*cases* (*EP*, *Phi-EP*) ∈ *wf-jvmprog*)
case *True* **thus** *?thesis* **by** (*simp add*: *Abs-wf-jvmprog-inverse*)
next
case *False* **with** *EP-wf* **show** *?thesis* **by** (*simp add*: *wf-jvmprog-def*)
qed

lemma [*simp*]: *TYPING* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)) = *Phi-EP*
proof (*cases* (*EP*, *Phi-EP*) ∈ *wf-jvmprog*)
case *True* **thus** *?thesis* **by** (*simp add*: *Abs-wf-jvmprog-inverse*)
next
case *False* **with** *EP-wf* **show** *?thesis* **by** (*simp add*: *wf-jvmprog-def*)
qed

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''*
by (*fastforce elim*: *Methods.cases*
simp: *class-def SystemClasses-def ObjectC-def NullPointerC-def OutOfMemoryC-def ClassCastC-def*
split-if-eq1 EP-def Method-def)

lemma [*simp*]:
∃ *T Ts mxs mxl is*. (∃ *xt*. *EP* ⊢ *''C''* *sees* *''M''*: *Ts*→*T* = (*mxs*, *mxl*, *is*, *xt*) *in* *''C''*) ∧ *is* ≠ []
using *EP-wf*
by (*fastforce dest*: *mdecl-visible simp*: *wf-jvm-prog-phi-def EP-def*)

lemma [*simp*]:
∃ *T Ts mxs mxl is*. (∃ *xt*. *EP* ⊢ *''C''* *sees* *''M''*: *Ts*→*T* = (*mxs*, *mxl*, *is*, *xt*) *in* *''C''*) ∧
Suc 0 < *length is*
using *EP-wf*
by (*fastforce dest*: *mdecl-visible simp*: *wf-jvm-prog-phi-def EP-def*)

lemma *C-sees-M-in-EP* [*simp*]:
EP ⊢ *''C''* *sees* *''M''*: []→*Void* = (*Suc 0*, *0*, [*Push Unit*, *instr.Return*], []) *in* *''C''*
proof −
have *EP* ⊢ *''C''* *sees-methods* [*''M''* ↦ (([], *Void*, *1*, *0*, [*Push Unit*, *instr.Return*], []), *''C''*)]
by (*fastforce intro*: *Methods.intros simp*: *class-def SystemClasses-def ObjectC-def EP-def*)
thus *?thesis* **by** (*fastforce simp*: *Method-def*)
qed

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

unfolding *method-def*
by (*rule theI2* [**where** $P = \lambda(D,\ Ts,\ T,\ m).\ EP \vdash\ ''C''$ *sees* $''M''$: $Ts \rightarrow T = m$ *in* D])
(*auto dest*: *method-in-EP-is-M*)

lemma *ClassMain-not-C* [*simp*]: *ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)) $\neq$ $''C''$
by (*fastforce intro*: *no-Call-in-ClassMain* [**where** *P*=*Abs-wf-jvmprog* (*EP*, *Phi-EP*)] *C-sees-M-in-EP*)

lemma *method-entry* [*dest*!]: (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), $''C''$, $''M''$) $\vdash \Rightarrow$(*C*, *M*, *None*, *Enter*)
$\Longrightarrow$ (*C* = *ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)) $\wedge$ *M* = *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)))
$\vee$ (*C* = $''C''$ $\wedge$ *M* = $''M''$)
by (*fastforce elim*: *reachable.cases elim*!: *JVMCFG.cases dest*!: *method-in-EP-is-M*)

lemma *valid-node-in-EP-D*:
assumes *vn*: *JVMCFG-Interpret.valid-node* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)) $''C''$ $''M''$ *n*
shows *n* $\in$ {
(*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*)), $\lfloor 0 \rfloor$, *Enter*),
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), $\lfloor 0 \rfloor$, *Normal*),
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), $\lfloor 0 \rfloor$, *Return*),
($''C''$, $''M''$, *None*, *Enter*),
($''C''$, $''M''$, $\lfloor 0 \rfloor$, *Enter*),
($''C''$, $''M''$, $\lfloor 1 \rfloor$, *Enter*),
($''C''$, $''M''$, *None*, *Return*)
}
using *vn*
proof (*cases rule*: *JVMCFG-Interpret.valid-node-cases'*)
let *?prog* = *Abs-wf-jvmprog* (*EP*, *Phi-EP*)
case (*Source e*)
then obtain *C M pc nt ek C' M' pc' nt'*
where [*simp*]: *e* = ((*C*, *M*, *pc*, *nt*), *ek*, (*C'*, *M'*, *pc'*, *nt'*))
and [*simp*]: *n* = (*C*, *M*, *pc*, *nt*)
and *edge*: (*?prog*, $''C''$, $''M''$) $\vdash$ (*C*, *M*, *pc*, *nt*) $-ek\rightarrow$ (*C'*, *M'*, *pc'*, *nt'*)
by (*cases e*) (*fastforce simp*: *valid-edge-def*)
from *edge* **have** *src-reachable*: (*?prog*, $''C''$, $''M''$) $\vdash \Rightarrow$(*C*, *M*, *pc*, *nt*)
by $-$(*drule sourcenode-reachable*)
show *?thesis*
proof (*cases C* = *ClassMain ?prog*)

```
    case True
    with src-reachable have M = MethodMain ?prog
      by (fastforce dest: ClassMain-imp-MethodMain)
    with True edge show ?thesis
      by clarsimp (erule JVMCFG.cases, simp-all)
  next
    case False
    with src-reachable obtain T Ts mb where EP ⊢ C sees M:Ts→T = mb in C
      by (fastforce dest: method-of-reachable-node-exists)
    hence [simp]: C = ''C''
      and [simp]: M = ''M''
      and [simp]: Ts = []
      and [simp]: T = Void
      and [simp]: mb = (1, 0, [Push Unit, instr.Return], [])
      by (cases mb, fastforce dest: method-in-EP-is-M)+
    from src-reachable False have pc ∈ {None, ⌊0⌋, ⌊1⌋}
      by (fastforce dest: instr-of-reachable-node-typable)
    show ?thesis
    proof (cases pc)
      case None
      with edge False show ?thesis
        by clarsimp (erule JVMCFG.cases, simp-all)
    next
      case (Some pc')
      show ?thesis
      proof (cases pc')
        case 0
        with Some False edge show ?thesis
          by clarsimp (erule JVMCFG.cases, fastforce+)
      next
        case (Suc n)
        with ‹pc ∈ {None, ⌊0⌋, ⌊1⌋}› Some have pc = ⌊1⌋
          by simp
        with False edge show ?thesis
          by clarsimp (erule JVMCFG.cases, fastforce+)
      qed
    qed
  qed
next
  let ?prog = Abs-wf-jvmprog (EP, Phi-EP)
  case (Target e)
  then obtain C M pc nt ek C' M' pc' nt'
    where [simp]: e = ((C, M, pc, nt), ek, (C', M', pc', nt'))
    and [simp]: n = (C', M', pc', nt')
    and edge: (?prog, ''C'', ''M'') ⊢ (C, M, pc, nt) −ek→ (C', M', pc', nt')
    by (cases e) (fastforce simp: valid-edge-def)
  from edge have trg-reachable: (?prog, ''C'', ''M'') ⊢ ⇒(C', M', pc', nt')
    by −(drule targetnode-reachable)
  show ?thesis
```

```
  proof (cases C' = ClassMain ?prog)
    case True
    with trg-reachable have M' = MethodMain ?prog
      by (fastforce dest: ClassMain-imp-MethodMain)
    with True edge show ?thesis
      by -(clarsimp, (erule JVMCFG.cases, simp-all))+
  next
    case False
    with trg-reachable obtain T Ts mb where EP ⊢ C' sees M':Ts→T = mb in
C'
      by (fastforce dest: method-of-reachable-node-exists)
    hence [simp]: C' = ''C''
      and [simp]: M' = ''M''
      and [simp]: Ts = []
      and [simp]: T = Void
      and [simp]: mb = (1, 0, [Push Unit, instr.Return], [])
      by (cases mb, fastforce dest: method-in-EP-is-M)+
    from trg-reachable False have pc' ∈ {None, ⌊0⌋, ⌊1⌋}
      by (fastforce dest: instr-of-reachable-node-typable)
    show ?thesis
    proof (cases pc')
      case None
      with edge False show ?thesis
        by clarsimp (erule JVMCFG.cases, simp-all)
    next
      case (Some pc'')
      show ?thesis
      proof (cases pc'')
        case 0
        with Some False edge show ?thesis
          by -(clarsimp, (erule JVMCFG.cases, fastforce+))+
      next
        case (Suc n)
        with ‹pc' ∈ {None, ⌊0⌋, ⌊1⌋}› Some have pc' = ⌊1⌋
          by simp
        with False edge show ?thesis
          by -(clarsimp, (erule JVMCFG.cases, fastforce+))+
      qed
    qed
  qed
qed

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 -
  have valid-edge (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M'')
     ((ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
```

(*EP*, *Phi-EP*)), *None*,
Enter),
(λ*s*. *False*)$_{\surd}$,
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *None*,
Return))
by (*auto simp*: *valid-edge-def intro*: *JVMCFG-reachable.intros*)
thus *?thesis* **by** (*fastforce simp*: *JVMCFG-Interpret.valid-node-def*)
qed

lemma *main-0-Enter-reachable* [*simp*]: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Enter*)
by (*rule reachable-step* [**where** *n*=(*ClassMain P*, *MethodMain P*, *None*, *Enter*)])
(*fastforce intro*: *JVMCFG-reachable.intros*)+

lemma *main-0-Normal-reachable* [*simp*]: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)
by (*rule reachable-step* [**where** *n*=(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Enter*)], *simp*)
(*fastforce intro*: *JVMCFG-reachable.intros*)

lemma *main-0-Return-reachable* [*simp*]: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Return*)
by (*rule reachable-step* [**where** *n*=(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Normal*)], *simp*)
(*fastforce intro*: *JVMCFG-reachable.intros*)

lemma *Exit-reachable* [*simp*]: (*P*, *C0*, *Main*) ⊢ ⇒(*ClassMain P*, *MethodMain P*, *None*, *Return*)
by (*rule reachable-step* [**where** *n*=(*ClassMain P*, *MethodMain P*, ⌊*0*⌋, *Return*)], *simp*)
(*fastforce intro*: *JVMCFG-reachable.intros*)

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*
unfolding *cfg-wf-prog-def*
proof
let *?prog* = (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*)
let *?edge-main0* = ((*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), *None*, *Enter*),
(λ*s*. *False*)$_{\surd}$,
(*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), *None*, *Return*))

let *?edge-main1* = ((*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), *None*, *Enter*),
(λ*s*. *True*)$_{\surd}$,
(*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), ⌊*0*⌋, *Enter*))
let *?edge-main2* = ((*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), ⌊*0*⌋, *Enter*),
⇑*id*,
(*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), ⌊*0*⌋, *Normal*))
let *?edge-main3* = ((*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), ⌊*0*⌋, *Normal*),
(λ*s*. *False*)$_{\surd}$,
(*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), ⌊*0*⌋, *Return*))
let *?edge-main4* = ((*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), ⌊*0*⌋, *Return*),
⇑*id*,
(*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), *None*, *Return*))
let *?edge-call* = ((*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), ⌊*0*⌋, *Normal*),
((λ(*s*, *ret*). *True*):(*ClassMain* (*fst ?prog*),
MethodMain (*fst ?prog*), *0*)↪$_{(''C'',\ ''M'')}$[(λ*s*. *s Heap*),(λ*s*. ⌊*Value Null*⌋)]),
(*''C''*, *''M''*, *None*, *Enter*))
let *?edge-C0* = ((*''C''*, *''M''*, *None*, *Enter*),
(λ*s*. *False*)$_{\surd}$,
(*''C''*, *''M''*, *None*, *Return*))
let *?edge-C1* = ((*''C''*, *''M''*, *None*, *Enter*),
(λ*s*. *True*)$_{\surd}$,
(*''C''*, *''M''*, ⌊*0*⌋, *Enter*))
let *?edge-C2* = ((*''C''*, *''M''*, ⌊*0*⌋, *Enter*),
⇑(λ*s*. *s*(*Stack 0* ↦ *Value Unit*)),
(*''C''*, *''M''*, ⌊*1*⌋, *Enter*))
let *?edge-C3* = ((*''C''*, *''M''*, ⌊*1*⌋, *Enter*),
⇑(λ*s*. *s*(*Stack 0* := *s* (*Stack 0*))),
(*''C''*, *''M''*, *None*, *Return*))
let *?edge-return* = ((*''C''*, *''M''*, *None*, *Return*),
(λ(*s*, *ret*). *ret* = (*ClassMain* (*fst ?prog*),
MethodMain (*fst ?prog*), *0*))↩$_{(''C'',''M'')}$(λ*s s'*. *s'*(*Heap* := *s Heap*,
Exception := *s Exception*,
Stack 0 := *s* (*Stack 0*))),
(*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), ⌊*0*⌋, *Return*))
have [*simp*]:
(*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*) ⊢ ⇒(*''C''*, *''M''*, *None*, *Enter*)
by (*rule reachable-step* [**where** *n*=(*ClassMain* (*fst ?prog*), *MethodMain* (*fst ?prog*), ⌊*0*⌋, *Normal*)]
, *simp*)
(*fastforce intro*: *Main-Call C-sees-M-in-EP*)
hence [*simp*]:
(*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*) ⊢ ⇒(*''C''*, *''M''*, *None*, *nodeType.Return*)
by (*rule reachable-step* [**where** *n*=(*''C''*, *''M''*, *None*, *Enter*)])
(*fastforce intro*: *JVMCFG-reachable.intros*)
have [*simp*]:

```
  (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M'') ⊢ ⇒(''C'', ''M'', ⌊0⌋, Enter)
  by (rule reachable-step [where n=(''C'', ''M'', None, Enter)], simp)
 (fastforce intro: JVMCFG-reachable.intros)
 hence [simp]:
  (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M'') ⊢ ⇒(''C'', ''M'', ⌊Suc 0⌋, Enter)
 by (fastforce intro: reachable-step [where n=(''C'', ''M'', ⌊0⌋, Enter)] CFG-Push
    simp: ClassMain-not-C [symmetric])
 show ?prog ∈ {(P, C0, Main).
       ∀ n. CFG.valid-node sourcenode targetnode (valid-edge (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, nodeType.Return))}
 proof (auto dest!: valid-node-in-EP-D)
  have CFG.valid-path' sourcenode targetnode kind
       (valid-edge (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M''))
       (get-return-edges (Abs-wf-jvmprog (EP, Phi-EP)))
      (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
        None, Enter)
       [?edge-main0]
      (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
          None, nodeType.Return)
  by (fastforce intro: JVMCFG-Interpret.intra-path-vp JVMCFG-reachable.intros
     simp: JVMCFG-Interpret.intra-path-def intra-kind-def valid-edge-def)
  thus ∃ as. CFG.valid-path' sourcenode targetnode kind
       (valid-edge (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M''))
       (get-return-edges (Abs-wf-jvmprog (EP, Phi-EP)))
      (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
        None, Enter)
     as (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
          None, nodeType.Return)
   by blast
 next
  have CFG.valid-path' sourcenode targetnode kind
       (valid-edge (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M''))
       (get-return-edges (Abs-wf-jvmprog (EP, Phi-EP)))
      (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
        None, nodeType.Return)
   [] (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
          None, nodeType.Return)
  by (fastforce intro: JVMCFG-Interpret.intra-path-vp simp: JVMCFG-Interpret.intra-path-def)
  thus ∃ as. CFG.valid-path' sourcenode targetnode kind
```

(*valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*))
(*get-return-edges* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)))
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
None, *nodeType.Return*)
as (*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
None, *nodeType.Return*)
by *blast*
next
have *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*))
(*get-return-edges* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)))
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
⌊*0*⌋, *Enter*)
[*?edge-main2*, *?edge-main3*, *?edge-main4*]
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
None, *nodeType.Return*)
by (*fastforce intro*: *JVMCFG-Interpret.intra-path-vp JVMCFG-reachable.intros*
simp: *JVMCFG-Interpret.intra-path-def intra-kind-def valid-edge-def*)
thus ∃ *as*. *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*))
(*get-return-edges* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)))
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
⌊*0*⌋, *Enter*)
as (*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
None, *nodeType.Return*)
by *blast*
next
have *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*))
(*get-return-edges* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)))
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
⌊*0*⌋, *Normal*)
[*?edge-main3*, *?edge-main4*]
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
None, *nodeType.Return*)
by (*fastforce intro*: *JVMCFG-Interpret.intra-path-vp JVMCFG-reachable.intros*
simp: *JVMCFG-Interpret.intra-path-def intra-kind-def valid-edge-def*)
thus ∃ *as*. *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*))
(*get-return-edges* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)))
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog*

```
(EP, Phi-EP)),
          ⌊0⌋, Normal)
       as (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
             None, nodeType.Return)
     by blast
  next
    have CFG.valid-path' sourcenode targetnode kind
          (valid-edge (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M''))
          (get-return-edges (Abs-wf-jvmprog (EP, Phi-EP)))
         (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
          ⌊0⌋, nodeType.Return)
      [?edge-main4]
       (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
             None, nodeType.Return)
    by (fastforce intro: JVMCFG-Interpret.intra-path-vp JVMCFG-reachable.intros
       simp: JVMCFG-Interpret.intra-path-def intra-kind-def valid-edge-def)
    thus ∃as. CFG.valid-path' sourcenode targetnode kind
          (valid-edge (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M''))
          (get-return-edges (Abs-wf-jvmprog (EP, Phi-EP)))
         (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
          ⌊0⌋, nodeType.Return)
       as (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
             None, nodeType.Return)
     by blast
  next
    have CFG.valid-path' sourcenode targetnode kind
          (valid-edge (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M''))
            (get-return-edges (Abs-wf-jvmprog (EP, Phi-EP))) (''C'', ''M'', None,
Enter)
      [?edge-C0, ?edge-return, ?edge-main4]
         (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
           None, nodeType.Return)
     by (fastforce intro: JVMCFG-reachable.intros C-sees-M-in-EP
     simp: JVMCFG-Interpret.vp-def valid-edge-def stkLength-def JVMCFG-Interpret.valid-path-def)
    thus ∃as. CFG.valid-path' sourcenode targetnode kind
          (valid-edge (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M''))
            (get-return-edges (Abs-wf-jvmprog (EP, Phi-EP))) (''C'', ''M'', None,
Enter) as
         (ClassMain (Abs-wf-jvmprog (EP, Phi-EP)), MethodMain (Abs-wf-jvmprog
(EP, Phi-EP)),
           None, nodeType.Return)
     by blast
  next
```

have *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*))
(*get-return-edges* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*))) (*''C''*, *''M''*, ⌊*0*⌋, *Enter*)
[*?edge-C2*, *?edge-C3*, *?edge-return*, *?edge-main4*]
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
None, *nodeType.Return*)
by (*fastforce intro*: *Main-Return-to-Exit CFG-Return-from-Method Main-Call C-sees-M-in-EP CFG-Return CFG-Push*
simp: *JVMCFG-Interpret.vp-def valid-edge-def stkLength-def Phi-EP-def ClassMain-not-C* [*symmetric*] *JVMCFG-Interpret.valid-path-def*)
thus ∃ *as*. *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*))
(*get-return-edges* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*))) (*''C''*, *''M''*, ⌊*0*⌋, *Enter*) *as*
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
None, *nodeType.Return*)
by *blast*
next
have *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*))
(*get-return-edges* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*))) (*''C''*, *''M''*, ⌊*Suc 0*⌋, *Enter*)
[*?edge-C3*, *?edge-return*, *?edge-main4*]
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
None, *nodeType.Return*)
by (*fastforce intro*: *JVMCFG-reachable.intros C-sees-M-in-EP*
simp: *JVMCFG-Interpret.vp-def valid-edge-def stkLength-def Phi-EP-def ClassMain-not-C* [*symmetric*] *JVMCFG-Interpret.valid-path-def*)
thus ∃ *as*. *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*))
(*get-return-edges* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*))) (*''C''*, *''M''*, ⌊*Suc 0*⌋, *Enter*) *as*
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
None, *nodeType.Return*)
by *blast*
next
have *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*))
(*get-return-edges* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*))) (*''C''*, *''M''*, *None*, *nodeType.Return*)
[*?edge-return*, *?edge-main4*]
(*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
None, *nodeType.Return*)
by (*fastforce intro*: *JVMCFG-reachable.intros C-sees-M-in-EP*

simp: *JVMCFG-Interpret.vp-def valid-edge-def JVMCFG-Interpret.valid-path-def stkLength-def*)
thus ∃ *as*. *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), *''C''*, *''M''*))
(*get-return-edges* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*))) (*''C''*, *''M''*, *None*, *nodeType.Return*)
as (*ClassMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)), *MethodMain* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*)),
None, *nodeType.Return*)
by *blast*
qed
qed

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*))
by (*cases P*) (*clarsimp simp*: *Abs-cfg-wf-prog-inverse*)

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*
unfolding *valid-edge-CFG-def*
proof
fix *n*
obtain *P' C0 Main* **where** [*simp*]: *fst*$_{CFG}$ *P* = *P'* **and** [*simp*]: *fst* (*snd*$_{CFG}$ *P*) = *C0*
and [*simp*]: *snd* (*snd*$_{CFG}$ *P*) = *Main*
by (*cases P*) *clarsimp*
assume *CFG.valid-node sourcenode targetnode*
(*valid-edge* (*fst*$_{CFG}$ *P*, *fst* (*snd*$_{CFG}$ *P*), *snd* (*snd*$_{CFG}$ *P*))) *n*
thus ∃ *as*. *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*fst*$_{CFG}$ *P*, *fst* (*snd*$_{CFG}$ *P*), *snd* (*snd*$_{CFG}$ *P*)))
(*get-return-edges* (*fst*$_{CFG}$ *P*))
(*ClassMain* (*fst*$_{CFG}$ *P*), *MethodMain* (*fst*$_{CFG}$ *P*), *None*, *Enter*) *as n*
by (*auto dest*: *sourcenode-reachable targetnode-reachable valid-Entry-path*
simp: *JVMCFG-Interpret.valid-node-def valid-edge-def*)
next
fix *n*
obtain *P' C0 Main* **where** [*simp*]: *fst*$_{CFG}$ *P* = *P'* **and** [*simp*]: *fst* (*snd*$_{CFG}$ *P*)

= *C0*
and [*simp*]: *snd* (*snd*$_{CFG}$ *P*) = *Main*
and (*P'*, *C0*, *Main*) ∈ *cfg-wf-prog*
by (*cases P*) (*clarsimp simp*: *Abs-cfg-wf-prog-inverse*)
assume *CFG.valid-node sourcenode targetnode*
(*valid-edge* (*fst*$_{CFG}$ *P*, *fst* (*snd*$_{CFG}$ *P*), *snd* (*snd*$_{CFG}$ *P*))) *n*
with ‹(*P'*, *C0*, *Main*) ∈ *cfg-wf-prog*›
show ∃ *as*. *CFG.valid-path' sourcenode targetnode kind*
(*valid-edge* (*fst*$_{CFG}$ *P*, *fst* (*snd*$_{CFG}$ *P*), *snd* (*snd*$_{CFG}$ *P*)))
(*get-return-edges* (*fst*$_{CFG}$ *P*)) *n as*
(*ClassMain* (*fst*$_{CFG}$ *P*), *MethodMain* (*fst*$_{CFG}$ *P*), *None*, *nodeType.Return*)
by (*cases n*) (*fastforce simp*: *cfg-wf-prog-def*)
next
fix *n n'*
obtain *P' C0 Main* **where** [*simp*]: *fst*$_{CFG}$ *P* = *P'* **and** [*simp*]: *fst* (*snd*$_{CFG}$ *P*)
= *C0*
and [*simp*]: *snd* (*snd*$_{CFG}$ *P*) = *Main*
by (*cases P*) *clarsimp*
assume *CFGExit.method-exit sourcenode kind*
(*valid-edge* (*fst*$_{CFG}$ *P*, *fst* (*snd*$_{CFG}$ *P*), *snd* (*snd*$_{CFG}$ *P*)))
(*ClassMain* (*fst*$_{CFG}$ *P*), *MethodMain* (*fst*$_{CFG}$ *P*), *None*, *nodeType.Return*) *n*
and *CFGExit.method-exit sourcenode kind*
(*valid-edge* (*fst*$_{CFG}$ *P*, *fst* (*snd*$_{CFG}$ *P*), *snd* (*snd*$_{CFG}$ *P*)))
(*ClassMain* (*fst*$_{CFG}$ *P*), *MethodMain* (*fst*$_{CFG}$ *P*), *None*, *nodeType.Return*) *n'*
and (λ(*C*, *M*, *pc*, *type*). (*C*, *M*)) *n* = (λ(*C*, *M*, *pc*, *type*). (*C*, *M*)) *n'*
thus *n* = *n'*
by (*auto simp*: *JVMCFG-Exit-Interpret.method-exit-def valid-edge-def*)
(*fastforce elim*: *JVMCFG.cases*)+
qed

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*
unfolding *valid-edge-CFG-def*
..

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*
..

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://afp.sf.net/entries/Slicing.shtml, September 2008. Formal proof development.