Towards Certified Slicing

Daniel Wasserrab

March 19, 2025

Abstract

Slicing is a widely-used technique with applications in e.g. compiler technology and software security. Thus verification of algorithms in these areas is often based on the correctness of slicing, which should ideally be proven independent of concrete programming languages and with the help of well-known verifying techniques such as proof assistants. As a first step in this direction, this contribution presents a framework for dynamic [2] and static intraprocedural slicing [1] based on control flow and program dependence graphs. Abstracting from concrete syntax we base the framework on a graph representation of the program fulfilling certain structural and well-formedness properties.

We provide two instantiations to show the validity of the framework: a simple While language and the sophisticated object-oriented byte code language from Jinja [3].

0.1 Auxiliary lemmas

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

abbreviation *arbitrary* == *undefined*

Lemmas about left- and rightmost elements in lists

lemma *leftmost-element-property*:
  assumes $\exists\, x \in set\ xs.\ P\ x$
  obtains $zs\ x'\ ys$ **where** $xs = zs@x'\#ys$ **and** $P\ x'$ **and** $\forall\, z \in set\ zs.\ \neg\ P\ z$
proof(*atomize-elim*)
  from ‹$\exists\, x \in set\ xs.\ P\ x$›
  show $\exists\, zs\ x'\ ys.\ xs = zs\ @\ x'\ \#\ ys \wedge P\ x' \wedge (\forall\, z{\in}set\ zs.\ \neg\ P\ z)$
  proof(*induct xs*)
    case *Nil* **thus** *?case* **by** *simp*
  next
    case (*Cons* $x'\ xs'$)
    note IH = ‹$\exists\, a{\in}set\ xs'.\ P\ a$
      $\Longrightarrow \exists\, zs\ x'\ ys.\ xs' = zs@x'\#ys \wedge P\ x' \wedge (\forall\, z{\in}set\ zs.\ \neg\ P\ z)$›
    show *?case*

proof (*cases P x'*)
case *True*
then have ($\exists$ *ys*. x' # xs' = [] @ x' # *ys*) $\wedge$ *P* x' $\wedge$ ($\forall x \in$ *set* []. $\neg$ *P x*) **by** *simp*
then show *?thesis* **by** *blast*
next
case *False*
with ‹$\exists y \in$ *set* (x'#xs'). *P y*› **have** $\exists y \in$ *set* xs'. *P y* **by** *simp*
from *IH*[*OF this*] **obtain** *y ys zs* **where** xs' = *zs*@*y*#*ys*
and *P y* **and** $\forall z \in$ *set zs*. $\neg$ *P z* **by** *blast*
from ‹$\forall z \in$ *set zs*. $\neg$ *P z*› *False* **have** $\forall z \in$ *set* (x'#*zs*). $\neg$ *P z* **by** *simp*
with ‹xs' = *zs*@*y*#*ys*› ‹*P y*› **show** *?thesis* **by** (*metis Cons-eq-append-conv*)
qed
qed
qed

lemma *rightmost-element-property*:
assumes $\exists x \in$ *set xs*. *P x*
obtains *ys* x' *zs* **where** *xs* = *ys*@x'#*zs* **and** *P* x' **and** $\forall z \in$ *set zs*. $\neg$ *P z*
proof(*atomize-elim*)
from ‹$\exists x \in$ *set xs*. *P x*›
show $\exists$ *ys* x' *zs*. *xs* = *ys* @ x' # *zs* $\wedge$ *P* x' $\wedge$ ($\forall z \in$ *set zs*. $\neg$ *P z*)
proof(*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons* x' xs')
note *IH* = ‹$\exists a \in$ *set* xs'. *P a*
$\Longrightarrow$ $\exists$ *ys* x' *zs*. xs' = *ys* @ x' # *zs* $\wedge$ *P* x' $\wedge$ ($\forall z \in$ *set zs*. $\neg$ *P z*)›
show *?case*
proof(*cases* $\exists y \in$ *set* xs'. *P y*)
case *True*
from *IH*[*OF this*] **obtain** *y ys zs* **where** xs' = *ys* @ *y* # *zs*
and *P y* **and** $\forall z \in$ *set zs*. $\neg$ *P z* **by** *blast*
thus *?thesis* **by** (*metis Cons-eq-append-conv*)
next
case *False*
with ‹$\exists y \in$ *set* (x'#xs'). *P y*› **have** *P* x' **by** *simp*
with *False* **show** *?thesis* **by** (*metis eq-Nil-appendI*)
qed
qed
qed

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 *lists*

lemma *path-split-general*:
assumes *all*:$\forall\, zs.\ xs \neq ys@zs$
obtains $j\ zs$ **where** $xs = (take\ j\ ys)@zs$ **and** $j < length\ ys$
  and $\forall\, k > j.\ \forall\, zs'.\ xs \neq (take\ k\ ys)@zs'$
proof(*atomize-elim*)
  from ‹$\forall\, zs.\ xs \neq ys@zs$›
  show $\exists\, j\ zs.\ xs = take\ j\ ys\ @\ zs \wedge j < length\ ys \wedge$
        $(\forall\, k{>}j.\ \forall\, zs'.\ xs \neq take\ k\ ys\ @\ zs')$
  proof(*induct ys arbitrary*:*xs*)
    case *Nil* **thus** *?case* **by** *auto*
  next
    case (*Cons* y' ys')
    note IH = ‹$\bigwedge xs.\ \forall\, zs.\ xs \neq ys'\ @\ zs \Longrightarrow$
     $\exists\, j\ zs.\ xs = take\ j\ ys'\ @\ zs \wedge j < length\ ys' \wedge$
     $(\forall\, k.\ j < k \longrightarrow (\forall\, zs'.\ xs \neq take\ k\ ys'\ @\ zs'))$›
    show *?case*
    proof(*cases xs*)
      case *Nil* **thus** *?thesis* **by** *simp*
    next
      case (*Cons* x' xs')
      with ‹$\forall\, zs.\ xs \neq (y'\ \#\ ys')\ @\ zs$› **have** $x' \neq y' \vee (\forall\, zs.\ xs' \neq ys'\ @\ zs)$
        by *simp*
      show *?thesis*
      proof(*cases* $x' = y'$)
        case *True*
        with ‹$x' \neq y' \vee (\forall\, zs.\ xs' \neq ys'\ @\ zs)$› **have** $\forall\, zs.\ xs' \neq ys'\ @\ zs$ **by** *simp*
        from *IH*[*OF this*] **have** $\exists\, j\ zs.\ xs' = take\ j\ ys'\ @\ zs \wedge j < length\ ys' \wedge$
          $(\forall\, k.\ j < k \longrightarrow (\forall\, zs'.\ xs' \neq take\ k\ ys'\ @\ zs'))$ **.**
        then obtain $j\ zs$ **where** $xs' = take\ j\ ys'\ @\ zs$
          and $j < length\ ys'$
          and *all-sub*:$\forall\, k.\ j < k \longrightarrow (\forall\, zs'.\ xs' \neq take\ k\ ys'\ @\ zs')$
          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*:$\forall\, k.\ j < k \longrightarrow$
          $(\forall\, zs'.\ (x'\#xs') \neq take\ (Suc\ k)\ (y'\ \#\ ys')\ @\ zs')$
          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 $\forall\, zs'.\ (x'\#xs') \neq take\ (Suc\ k)\ (y'\ \#\ ys')\ @\ zs'$
            by *simp*
          hence $\forall\, zs'.\ (x'\#xs') \neq take\ l\ (y'\ \#\ ys')\ @\ zs'$
            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

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

As slicing is a program analysis that can be completely based on the information given in the CFG, we want to provide a framework which allows us to formalize and prove properties of slicing regardless of the actual programming language. So the starting point for the formalization is the definition of an abstract CFG, i.e. without considering features specific for certain languages. By doing so we ensure that our framework is as generic as possible since all proofs hold for every language whose CFG conforms to this abstract CFG. This abstract CFG can be used as a basis for static intraprocedural slicing as well as for dynamic slicing, if in the dynamic case all method calls are inlined (i.e., abstract CFG paths conform to traces).

1.1 Basic Definitions

1.1.1 Edge kinds

datatype $'state\ edge\text{-}kind = Update\ 'state \Rightarrow 'state$ $(\langle\Uparrow\text{-}\rangle)$
$\quad | \ Predicate\ 'state \Rightarrow bool$ $(\langle '(\text{-}')_{\surd}\rangle)$

1.1.2 Transfer and predicate functions

fun $transfer :: 'state\ edge\text{-}kind \Rightarrow 'state \Rightarrow 'state$
where $transfer\ (\Uparrow f)\ s = f\ s$
$\ |\ transfer\ (P)_{\surd}\ s\ \ = s$

fun $transfers :: 'state\ edge\text{-}kind\ list \Rightarrow 'state \Rightarrow 'state$
where $transfers\ []\ s\ \ = s$
$\ |\ transfers\ (e\#es)\ s = transfers\ es\ (transfer\ e\ s)$

fun $pred :: 'state\ edge\text{-}kind \Rightarrow 'state \Rightarrow bool$
where $pred\ (\Uparrow f)\ s = True$
$\ |\ pred\ (P)_{\surd}\ s\ \ = (P\ s)$

fun *preds* :: *'state edge-kind list* ⇒ *'state* ⇒ *bool*
where *preds* [] *s* = *True*
| *preds* (*e*#*es*) *s* = (*pred e s* ∧ *preds es* (*transfer e 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*

lemma *transfers-id-no-influence*:
transfers [*et* ← *ets*. *et* ≠ ⇑*id*] *s* = *transfers ets s*
by(*induct ets arbitrary*:*s*,*auto*)

lemma *preds-True-no-influence*:
preds [*et* ← *ets*. *et* ≠ (λ*s*. *True*)$_{\surd}$] *s* = *preds ets s*
by(*induct ets arbitrary*:*s*,*auto*)

end

1.2 CFG

theory *CFG* **imports** *BasicDefs* **begin**

1.2.1 The abstract CFG

locale *CFG* =
fixes *sourcenode* :: *'edge* ⇒ *'node*
fixes *targetnode* :: *'edge* ⇒ *'node*
fixes *kind* :: *'edge* ⇒ *'state edge-kind*
fixes *valid-edge* :: *'edge* ⇒ *bool*
fixes *Entry*::*'node* (‹*'*(*'*-*Entry'*-*'*)›)
assumes *Entry-target* [*dest*]: ⟦*valid-edge a*; *targetnode a* = (-*Entry*-)⟧ ⟹ *False*
and *edge-det*:
⟦*valid-edge a*; *valid-edge a'*; *sourcenode a* = *sourcenode a'*;
targetnode a = *targetnode a'*⟧ ⟹ *a* = *a'*

begin

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

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

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

1.2.2 CFG paths and lemmas

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

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

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

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

lemma *path-valid-edges*:*n* −*as*→∗ *n'* $\Longrightarrow$ ∀ *a* ∈ *set as*. *valid-edge a*
by(*induct rule*:*path.induct*) *auto*

lemma *path-edge*:*valid-edge a* $\Longrightarrow$ *sourcenode a* −[*a*]→∗ *targetnode a*
by(*fastforce intro*:*Cons-path empty-path*)

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

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

lemma *path-split*:
assumes *n* −*as*@*a*#*as'*→∗ *n'*
shows *n* −*as*→∗ *sourcenode a* **and** *valid-edge a* **and** *targetnode a* −*as'*→∗ *n'*
using ‹*n* −*as*@*a*#*as'*→∗ *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* = ‹⋀*n*. *n* −*asx*@*a*#*as'*→∗ *n'* ⟹ *n* −*asx*→∗ *sourcenode a*›
note *IH2* = ‹⋀*n*. *n* −*asx*@*a*#*as'*→∗ *n'* ⟹ *valid-edge a*›
note *IH3* = ‹⋀*n*. *n* −*asx*@*a*#*as'*→∗ *n'* ⟹ *targetnode a* −*as'*→∗ *n'*›
{ case *1*
hence *sourcenode ax* = *n* **and** *targetnode ax* −*asx*@*a*#*as'*→∗ *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 −
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'› *that* **show** *?thesis*
by *fastforce*
qed

lemma *path-split-snoc*:
assumes n −as→∗ n' **and** $as \neq$ []
obtains a' as' **where** as = as'@[a'] **and** n −as'→∗ *sourcenode* a'
and *valid-edge* a' **and** n' = *targetnode* a'
proof −
from ‹$as \neq$ []› **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'› *that* **show** *?thesis*

by *fastforce*
qed

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

lemma *path-Entry-Cons*:
assumes (*-Entry-*) −as→∗ n' **and** $n' \neq$ (*-Entry-*)
obtains n a **where** *sourcenode* a = (*-Entry-*) **and** *targetnode* a = n
and n −*tl* as→∗ n' **and** *valid-edge* a **and** a = *hd* as
proof −
from ‹(*-Entry-*) −as→∗ n'› ‹$n' \neq$ (*-Entry-*)› **have** $as \neq$ []
by(*cases as*,*auto elim*:*path.cases*)
with ‹(*-Entry-*) −as→∗ n'› **obtain** a' as' **where** as = a'#as'
and (*-Entry-*) = *sourcenode* a' **and** *valid-edge* a' **and** *targetnode* a' −as'→∗ n'
by(*erule path-split-Cons*)
with *that* **show** *?thesis* **by** *fastforce*
qed

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

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

definition
  kinds :: *'edge list* ⇒ *'state edge-kind list*
  where *kinds xs* ≡ *map kind xs*

definition
  targetnodes :: *'edge list* ⇒ *'node list*
  where *targetnodes xs* ≡ *map targetnode xs*

lemma *path-sourcenode*:
  $[\![n -as\rightarrow* n'; as \neq []]\!] \Longrightarrow hd\ (sourcenodes\ as) = n$
by(*fastforce elim*:*path-split-Cons simp*:*sourcenodes-def*)

lemma *path-targetnode*:
  $[\![n -as\rightarrow* n'; as \neq []]\!] \Longrightarrow last\ (targetnodes\ as) = n'$
by(*fastforce elim*:*path-split-snoc simp*:*targetnodes-def*)

lemma *sourcenodes-is-n-Cons-butlast-targetnodes*:
  $[\![n -as\rightarrow* n'; as \neq []]\!] \Longrightarrow$
  $sourcenodes\ as = n\#(butlast\ (targetnodes\ as))$
proof(*induct as arbitrary*:*n*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons a' as'*)
  note $IH = \langle\bigwedge n.\ [\![n -as'\rightarrow* n'; as' \neq []]\!]$
      $\Longrightarrow sourcenodes\ as' = n\#(butlast\ (targetnodes\ as'))\rangle$

```
  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

lemma Entry-sourcenode-hd:
  assumes n −as→* n' and (-Entry-) ∈ set (sourcenodes as)
  shows n = (-Entry-) and (-Entry-) ∉ set (sourcenodes (tl as))
  using ‹n −as→* n'› ‹(-Entry-) ∈ 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* = ‹(-*Entry*-) ∈ *set*(*sourcenodes as*) ⟹ *n''* = (-*Entry*-)›
  note *IH2* = ‹(-*Entry*-) ∈ *set*(*sourcenodes as*) ⟹ (-*Entry*-) ∉ *set*(*sourcenodes*(*tl as*))›
  have (-*Entry*-) ∉ *set* (*sourcenodes*(*tl*(*a*#*as*)))
  proof
    assume (-*Entry*-) ∈ *set* (*sourcenodes* (*tl* (*a*#*as*)))
    hence (-*Entry*-) ∈ *set* (*sourcenodes as*) **by** *simp*
    from *IH1*[*OF this*] **have** *n''* = (-*Entry*-) **by** *simp*
    with ‹*targetnode a* = *n''*› ‹*valid-edge a*› **show** *False* **by** −(*erule Entry-target*,*simp*)
  qed
  hence (-*Entry*-) ∉ *set* (*sourcenodes*(*tl*(*a*#*as*))) **by** *fastforce*
  { case *1*
    with ‹(-*Entry*-) ∉ *set* (*sourcenodes*(*tl*(*a*#*as*)))› ‹*sourcenode a* = *n*›
    show *?case* **by**(*simp add*:*sourcenodes-def*)
  next
    case *2*
    with ‹(-*Entry*-) ∉ *set* (*sourcenodes*(*tl*(*a*#*as*)))› ‹*sourcenode a* = *n*›
    show *?case* **by**(*simp add*:*sourcenodes-def*)
  }
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*
  for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
  and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
  and *Entry* :: *'node* (‹'('-*Entry*'-')›) +
  fixes *Exit*::*'node* (‹'('-*Exit*'-')›)
  assumes *Exit-source* [*dest*]: ⟦*valid-edge a*; *sourcenode a* = (-*Exit*-)⟧ ⟹ *False*
  and *Entry-Exit-edge*: ∃ *a*. *valid-edge a* ∧ *sourcenode a* = (-*Entry*-) ∧
    targetnode a = (-*Exit*-) ∧ *kind a* = (λ*s*. *False*)$_{\surd}$

begin

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

proof −
  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 *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

lemma *path-Exit-source* [*dest*]:
assumes (*-Exit-*) −*as*→∗ *n'* **shows** *n'* = (*-Exit-*) **and** *as* = []
using ‹(*-Exit-*) −*as*→∗ *n'*›
proof(*induct n*≡(*-Exit-*) *as n' rule*:*path.induct*)
case (*Cons-path n'' as n' a*)
from ‹*sourcenode a* = (*-Exit-*)› ‹*valid-edge a*› **have** *False*
by −(*rule Exit-source*,*simp-all*)
{ case *1* **with** ‹*False*› **show** *?case* **..**
next
case *2* **with** ‹*False*› **show** *?case* **..**
}
qed *simp-all*

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

end

end

1.3 Postdomination

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

1.3.1 Standard Postdomination

locale *Postdomination* = *CFGExit sourcenode targetnode kind valid-edge Entry Exit*
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*

and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (‹'('-*Entry*'-')›) **and** *Exit* :: *'node* (‹'('-*Exit*'-')›) +
assumes *Entry-path*:*valid-node n* ⟹ ∃ *as*. (-*Entry*-) −*as*→∗ *n*
and *Exit-path*:*valid-node n* ⟹ ∃ *as*. *n* −*as*→∗ (-*Exit*-)

begin

definition *postdominate* :: *'node* ⇒ *'node* ⇒ *bool* (‹- *postdominates* -› [*51*,*0*])
where *postdominate-def*:*n' postdominates n* ≡
(*valid-node n* ∧ *valid-node n'* ∧
((∀ *as*. *n* −*as*→∗ (-*Exit*-) ⟶ *n'* ∈ *set* (*sourcenodes as*))))

lemma *postdominate-implies-path*:
assumes *n' postdominates n* **obtains** *as* **where** *n* −*as*→∗ *n'*
proof(*atomize-elim*)
from ‹*n' postdominates n*› **have** *valid-node n*
and *all*:∀ *as*. *n* −*as*→∗ (-*Exit*-) ⟶ *n'* ∈ *set*(*sourcenodes as*)
by(*auto simp*:*postdominate-def*)
from ‹*valid-node n*› **obtain** *as* **where** *n* −*as*→∗ (-*Exit*-) **by**(*auto dest*:*Exit-path*)
with *all* **have** *n'* ∈ *set*(*sourcenodes as*) **by** *simp*
then obtain *ns ns'* **where** *sourcenodes as* = *ns*@*n'*#*ns'* **by**(*auto dest*:*split-list*)
then obtain *as' a as''* **where** *sourcenodes as'* = *ns*
and *sourcenode a* = *n'* **and** *as* = *as'*@*a*#*as''*
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from ‹*n* −*as*→∗ (-*Exit*-)› ‹*as* = *as'*@*a*#*as''*› **have** *n* −*as'*→∗ *sourcenode a*
by(*fastforce dest*:*path-split*)
with ‹*sourcenode a* = *n'*› **show** ∃ *as*. *n* −*as*→∗ *n'* **by** *blast*
qed

lemma *postdominate-refl*:
assumes *valid*:*valid-node n* **and** *notExit*:*n* ≠ (-*Exit*-)
shows *n postdominates n*
using *valid*
proof(*induct rule*:*valid-node-cases*)
case *Entry*
{ fix *as* **assume** *path*:(-*Entry*-) −*as*→∗ (-*Exit*-)
hence *notempty*:*as* ≠ []
apply − **apply**(*erule path.cases*)
by (*drule sym*,*simp*,*drule Exit-noteq-Entry*,*auto*)
with *path* **have** *hd* (*sourcenodes as*) = (-*Entry*-)
by(*fastforce intro*:*path-sourcenode*)
with *notempty* **have** (-*Entry*-) ∈ *set* (*sourcenodes as*)
by(*fastforce intro*:*hd-in-set simp*:*sourcenodes-def*) **}**
with *Entry* **show** *?thesis* **by**(*simp add*:*postdominate-def*)
next
case *Exit*

with *notExit* **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
next
case *inner*
show *?thesis*
proof(*cases* ∃ *as*. *n* −*as*→∗ (-*Exit*-))
case *True*
{ **fix** *as'* **assume** *path'*:*n* −*as'*→∗ (-*Exit*-)
with *inner* **have** *notempty*:*as'* ≠ []
by(*cases as'*,*auto elim*!:*path.cases simp*:*inner-node-def*)
with *path' inner* **have** *hd*:*hd* (*sourcenodes as'*) = *n*
by −(*rule path-sourcenode*)
from *notempty* **have** *sourcenodes as'* ≠ [] **by**(*simp add*:*sourcenodes-def*)
with *hd*[*THEN sym*] **have** *n* ∈ *set* (*sourcenodes as'*) **by** *simp* }
hence ∀ *as*. *n* −*as*→∗ (-*Exit*-) ⟶ *n* ∈ *set* (*sourcenodes as*) **by** *simp*
with *True inner* **show** *?thesis* **by**(*simp add*:*postdominate-def inner-is-valid*)
next
case *False*
with *inner* **show** *?thesis* **by**(*simp add*:*postdominate-def inner-is-valid*)
qed
qed

lemma *postdominate-trans*:
assumes *pd1*:*n''* *postdominates n* **and** *pd2*:*n'* *postdominates n''*
shows *n'* *postdominates n*
proof −
from *pd1 pd2* **have** *valid*:*valid-node n* **and** *valid'*:*valid-node n'*
by(*simp-all add*:*postdominate-def*)
{ **fix** *as* **assume** *path*:*n* −*as*→∗ (-*Exit*-)
with *pd1* **have** *n''* ∈ *set* (*sourcenodes as*) **by**(*simp add*:*postdominate-def*)
then obtain *ns' ns''* **where** *sourcenodes as* = *ns'*@*n''*#*ns''*
by(*auto dest*:*split-list*)
then obtain *as' as'' a*
where *as''*:*sourcenodes as''* = *ns''* **and** *as*:*as*=*as'*@*a*#*as''*
and *source*:*sourcenode a* = *n''*
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from *as path* **have** *n* −*as'*@*a*#*as''*→∗ (-*Exit*-) **by** *simp*
with *source* **have** *path'*:*n''* −*a*#*as''*→∗ (-*Exit*-)
by(*fastforce dest*:*path-split-second*)
with *pd2* **have** *n'* ∈ *set*(*sourcenodes* (*a*#*as''*))
by(*auto simp*:*postdominate-def*)
with *as* **have** *n'* ∈ *set*(*sourcenodes as*) **by**(*auto simp*:*sourcenodes-def*) }
with *valid valid'* **show** *?thesis* **by**(*simp add*:*postdominate-def*)
qed

lemma *postdominate-antisym*:
assumes *pd1*:*n'* *postdominates n* **and** *pd2*:*n* *postdominates n'*

shows $n = n'$
proof −
from *pd1* **have** *valid*:*valid-node n* **and** *valid'*:*valid-node n'*
by(*auto simp*:*postdominate-def*)
from *valid* **obtain** *as* **where** *path1*:n −*as*→∗ (-*Exit*-) **by**(*fastforce dest*:*Exit-path*)
from *valid'* **obtain** *as'* **where** *path2*:n' −*as'*→∗ (-*Exit*-) **by**(*fastforce dest*:*Exit-path*)
from *pd1 path1* **have** ∃ *nx* ∈ *set*(*sourcenodes as*). *nx* = *n'*
by(*simp add*:*postdominate-def*)
then obtain *ns ns'* **where** *sources*:*sourcenodes as* = *ns*@*n'*#*ns'*
and *all*:∀ *nx* ∈ *set ns'*. *nx* ≠ *n'*
by(*fastforce elim*!: *rightmost-element-property*)
from *sources* **obtain** *asx a asx'* **where** *ns'*:*ns'* = *sourcenodes asx'*
and *as*:*as* = *asx*@*a*#*asx'* **and** *source*:*sourcenode a* = *n'*
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from *path1 as* **have** *n* −*asx*@*a*#*asx'*→∗ (-*Exit*-) **by** *simp*
with *source* **have** *n'* −*a*#*asx'*→∗ (-*Exit*-) **by**(*fastforce dest*:*path-split-second*)
with *pd2* **have** *n* ∈ *set*(*sourcenodes* (*a*#*asx'*)) **by**(*simp add*:*postdominate-def*)
with *source* **have** *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** *asx'*:*asx'* = *asi*@*a'*#*asi'*
and *source'*:*sourcenode a'* = *n*
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
with *path1 as* **have** *n* −(*asx*@*a*#*asi*)@*a'*#*asi'*→∗ (-*Exit*-) **by** *simp*
with *source'* **have** *n* −*a'*#*asi'*→∗ (-*Exit*-) **by**(*fastforce dest*:*path-split-second*)
with *pd1* **have** *n'* ∈ *set*(*sourcenodes* (*a'*#*asi'*)) **by**(*auto simp*:*postdominate-def*)
with *source'* **have** *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 *asx' ns'* **have** *n'* ∈ *set ns'* **by**(*simp add*:*sourcenodes-def*)
with *all* **have** *False* **by** *blast*
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*:
(*-Exit-*) *postdominates n* ⟹ *False*
by(*fastforce dest*:*Exit-path simp*:*postdominate-def*)

lemma *postdominate-path-targetnode*:
assumes *n′ postdominates n* **and** *n* −*as*→∗ *n″* **and** *n′* ∉ *set*(*sourcenodes as*)
shows *n′ postdominates n″*
proof −
from ‹*n′ postdominates n*› **have** *valid-node n* **and** *valid-node n′*
and *all*:∀ *as″*. *n* −*as″*→∗ (*-Exit-*) ⟶ *n′* ∈ *set*(*sourcenodes as″*)
by(*simp-all add*:*postdominate-def*)
from ‹*n* −*as*→∗ *n″*› **have** *valid-node n″* **by**(*fastforce dest*:*path-valid-node*)
have ∀ *as″*. *n″* −*as″*→∗ (*-Exit-*) ⟶ *n′* ∈ *set*(*sourcenodes as″*)
proof(*rule ccontr*)
assume ¬ (∀ *as″*. *n″* −*as″*→∗ (*-Exit-*) ⟶ *n′* ∈ *set* (*sourcenodes as″*))
then obtain *as″* **where** *n″* −*as″*→∗ (*-Exit-*)
and *n′* ∉ *set* (*sourcenodes as″*) **by** *blast*
from ‹*n* −*as*→∗ *n″*› ‹*n″* −*as″*→∗ (*-Exit-*)› **have** *n* −*as*@*as″*→∗ (*-Exit-*)
by(*rule path-Append*)
with ‹*n′* ∉ *set*(*sourcenodes as*)› ‹*n′* ∉ *set* (*sourcenodes as″*)›
have *n′* ∉ *set* (*sourcenodes* (*as*@*as″*))
by(*simp add*:*sourcenodes-def*)
with ‹*n* −*as*@*as″*→∗ (*-Exit-*)› ‹*n′ postdominates n*› **show** *False*

by(*simp add*:*postdominate-def*)
qed
with ‹*valid-node n′*› ‹*valid-node n″*› **show** *?thesis* **by**(*simp add*:*postdominate-def*)
qed

lemma *not-postdominate-source-not-postdominate-target*:
assumes ¬ *n postdominates* (*sourcenode a*) **and** *valid-node n* **and** *valid-edge 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*
where *sourcenode a* −*asx*→∗ (-*Exit*-)
and *n* ∉ *set*(*sourcenodes asx*) **by**(*auto simp*:*postdominate-def*)
from ‹*sourcenode a* −*asx*→∗ (-*Exit*-)› ‹*valid-edge a*›
obtain *ax asx′* **where** [*simp*]:*asx* = *ax*#*asx′*
apply − **apply**(*erule path.cases*)
apply(*drule-tac s*=(-*Exit*-) **in** *sym*)
apply *simp*
apply(*drule Exit-source*)
by *simp-all*
with ‹*sourcenode a* −*asx*→∗ (-*Exit*-)› **have** *sourcenode a* −[]@*ax*#*asx′*→∗ (-*Exit*-)
by *simp*
hence *valid-edge ax* **and** *sourcenode a* = *sourcenode ax*
and *targetnode ax* −*asx′*→∗ (-*Exit*-)
by(*fastforce dest*:*path-split*)+
with ‹*n* ∉ *set*(*sourcenodes asx*)› **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

lemma *inner-node-Entry-edge*:
assumes *inner-node n*
obtains *a* **where** *valid-edge 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* ∧ *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*)

```
    thus ?thesis by simp
  next
    case False
    with ‹(-Entry-) −as→* n› obtain a' as' where as = a'#as'
      and (-Entry-) = sourcenode a' and valid-edge a'
      and targetnode a' −as'→* n
      by −(erule path-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› ‹inner-node n›
      have False by simp (drule path-Exit-source,auto simp:inner-node-def)
      thus ?thesis by simp
    next
      case inner
      with ‹valid-edge a'› ‹(-Entry-) = sourcenode a'› show ?thesis by simp blast
    qed
  qed
qed


lemma inner-node-Exit-edge:
  assumes inner-node n
  obtains a where valid-edge 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 ∧ inner-node (sourcenode a) ∧ targetnode a = (-Exit-)
  proof(cases as = [])
    case True
    with ‹inner-node n› ‹n −as→* (-Exit-)› have False by fastforce
    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 path-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'› ‹inner-node n›
      have False by simp (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*
with ‹*valid-edge a′*› ‹(*-Exit-*) = *targetnode a′*› **show** *?thesis* **by** *simp blast*
qed
qed
qed

end

1.3.2 Strong Postdomination

locale *StrongPostdomination* =
Postdomination sourcenode targetnode kind valid-edge Entry Exit
for *sourcenode* :: *′edge* ⇒ *′node* **and** *targetnode* :: *′edge* ⇒ *′node*
and *kind* :: *′edge* ⇒ *′state edge-kind* **and** *valid-edge* :: *′edge* ⇒ *bool*
and *Entry* :: *′node* (‹′(′-Entry′-′)›) **and** *Exit* :: *′node* (‹′(′-Exit′-′)›) +
assumes *successor-set-finite*: *valid-node n* ⟹
finite {*n′*. ∃ *a′*. *valid-edge a′* ∧ *sourcenode a′* = *n* ∧ *targetnode a′* = *n′*}

begin

definition *strong-postdominate* :: *′node* ⇒ *′node* ⇒ *bool*
(‹- *strongly−postdominates* -› [*51*,*0*])
where *strong-postdominate-def*:*n′ strongly−postdominates n* ≡
(*n′ postdominates n* ∧
(∃ *k* ≥ *1*. ∀ *as nx*. *n* −*as*→∗ *nx* ∧
length as ≥ *k* ⟶ *n′* ∈ *set*(*sourcenodes as*)))

lemma *strong-postdominate-prop-smaller-path*:
assumes *all*:∀ *as nx*. *n* −*as*→∗ *nx* ∧ *length as* ≥ *k* ⟶ *n′* ∈ *set*(*sourcenodes as*)
and *n* −*as*→∗ *n″* **and** *length as* ≥ *k*
obtains *as′ as″* **where** *n* −*as′*→∗ *n′* **and** *length as′* < *k* **and** *n′* −*as″*→∗ *n″*
and *as* = *as′*@*as″*
proof(*atomize-elim*)
show ∃ *as′ as″*. *n* −*as′*→∗ *n′* ∧ *length as′* < *k* ∧ *n′* −*as″*→∗ *n″* ∧ *as* = *as′*@*as″*
proof(*rule ccontr*)
assume ¬ (∃ *as′ as″*. *n* −*as′*→∗ *n′* ∧ *length as′* < *k* ∧ *n′* −*as″*→∗ *n″* ∧
as = *as′*@*as″*)
hence *all′*:∀ *as′ as″*. *n* −*as′*→∗ *n′* ∧ *n′* −*as″*→∗ *n″* ∧ *as* = *as′*@*as″*
⟶ *length as′* ≥ *k* **by** *fastforce*

from *all* ‹n −*as*→∗ n''› ‹*length as* ≥ *k*› **have** ∃ *nx* ∈ *set*(*sourcenodes as*). *nx* = n'
by *fastforce*
then obtain *ns* ns' **where** *sourcenodes as* = *ns*@n'#ns'
and ∀ *nx* ∈ *set ns*. *nx* ≠ n'
by(*fastforce elim*!:*split-list-first-propE*)
then obtain *asx a* asx' **where** [*simp*]:*ns* = *sourcenodes asx*
and [*simp*]:*as* = *asx*@*a*#asx' **and** *sourcenode a* = n'
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from ‹n −*as*→∗ n''› **have** n −*asx*@*a*#asx'→∗ n'' **by** *simp*
with ‹*sourcenode a* = n'› **have** n −*asx*→∗ n' **and** *valid-edge a*
and *targetnode a* −asx'→∗ n'' **by**(*fastforce dest*:*path-split*)+
with ‹*sourcenode a* = n'› **have** n' −*a*#asx'→∗ n'' **by**(*fastforce intro*:*Cons-path*)
with ‹n −*asx*→∗ n'› all' **have** *length asx* ≥ *k* **by** *simp*
with ‹n −*asx*→∗ n'› *all* **have** n' ∈ *set*(*sourcenodes asx*) **by** *fastforce*
with ‹∀ *nx* ∈ *set ns*. *nx* ≠ n'› **show** *False* **by** *fastforce*
qed
qed

lemma *strong-postdominate-refl*:
assumes *valid-node n* **and** n ≠ (*-Exit-*)
shows n *strongly−postdominates* n
proof −
from *assms* **have** n *postdominates* n **by**(*rule postdominate-refl*)
{ fix *as nx* **assume** n −*as*→∗ *nx* **and** *length as* ≥ *1*
then obtain a' as' **where** [*simp*]:*as* = a'#as' **by**(*cases as*) *auto*
with ‹n −*as*→∗ *nx*› **have** n −[]@a'#as'→∗ *nx* **by** *simp*
hence n = *sourcenode* a' **by**(*fastforce dest*:*path-split*)
hence n ∈ *set*(*sourcenodes as*) **by**(*simp add*:*sourcenodes-def*) **}**
hence ∀ *as nx*. n −*as*→∗ *nx* ∧ *length as* ≥ *1* ⟶ n ∈ *set*(*sourcenodes as*)
by *auto*
hence ∃ *k* ≥ *1*. ∀ *as nx*. n −*as*→∗ *nx* ∧ *length as* ≥ *k* ⟶ n ∈ *set*(*sourcenodes as*)
by *blast*
with ‹n *postdominates* n› **show** *?thesis* **by**(*simp add*:*strong-postdominate-def*)
qed

lemma *strong-postdominate-trans*:
assumes n'' *strongly−postdominates* n **and** n' *strongly−postdominates* n''
shows n' *strongly−postdominates* n
proof −
from ‹n'' *strongly−postdominates* n› **have** n'' *postdominates* n
and *paths1*:∃ *k* ≥ *1*. ∀ *as nx*. n −*as*→∗ *nx* ∧ *length as* ≥ *k*
⟶ n'' ∈ *set*(*sourcenodes as*)
by(*auto simp only*:*strong-postdominate-def*)
from *paths1* **obtain** *k1*

where *all1*:∀ *as nx*. *n* −*as*→∗ *nx* ∧ *length as* ≥ *k1* ⟶ *n''* ∈ *set*(*sourcenodes as*)
and *k1* ≥ *1* **by** *blast*
from ‹*n' strongly−postdominates n''*› **have** *n' postdominates n''*
and *paths2*:∃ *k* ≥ *1*. ∀ *as nx*. *n''* −*as*→∗ *nx* ∧ *length as* ≥ *k*
⟶ *n'* ∈ *set*(*sourcenodes as*)
by(*auto simp only*:*strong-postdominate-def*)
from *paths2* **obtain** *k2*
where *all2*:∀ *as nx*. *n''* −*as*→∗ *nx* ∧ *length as* ≥ *k2* ⟶ *n'* ∈ *set*(*sourcenodes as*)
and *k2* ≥ *1* **by** *blast*
from ‹*n'' postdominates n*› ‹*n' postdominates n''*›
have *n' postdominates n* **by**(*rule postdominate-trans*)
{ fix *as nx* **assume** *n* −*as*→∗ *nx* **and** *length as* ≥ *k1* + *k2*
hence *length as* ≥ *k1* **by** *fastforce*
with ‹*n* −*as*→∗ *nx*› *all1* **obtain** *asx asx'* **where** *n* −*asx*→∗ *n''*
and *length asx* < *k1* **and** *n''* −*asx'*→∗ *nx*
and [*simp*]:*as* = *asx*@*asx'* **by** −(*erule strong-postdominate-prop-smaller-path*)
with ‹*length as* ≥ *k1* + *k2*› **have** *length asx'* ≥ *k2* **by** *fastforce*
with ‹*n''* −*asx'*→∗ *nx*› *all2* **have** *n'* ∈ *set*(*sourcenodes asx'*) **by** *fastforce*
hence *n'* ∈ *set*(*sourcenodes as*) **by**(*simp add*:*sourcenodes-def*) **}**
with ‹*k1* ≥ *1*› ‹*k2* ≥ *1*› **have** ∃ *k* ≥ *1*. ∀ *as nx*. *n* −*as*→∗ *nx* ∧ *length as* ≥ *k*
⟶ *n'* ∈ *set*(*sourcenodes as*)
by(*rule-tac x*=*k1* + *k2* **in** *exI*,*auto*)
with ‹*n' postdominates n*› **show** *?thesis* **by**(*simp add*:*strong-postdominate-def*)
qed

lemma *strong-postdominate-antisym*:
⟦*n' strongly−postdominates n*; *n strongly−postdominates n'*⟧ ⟹ *n* = *n'*
by(*fastforce intro*:*postdominate-antisym simp*:*strong-postdominate-def*)

lemma *strong-postdominate-path-branch*:
assumes *n* −*as*→∗ *n''* **and** *n' strongly−postdominates n''*
and ¬ *n' strongly−postdominates n*
obtains *a as' as''* **where** *as* = *as'*@*a*#*as''* **and** *valid-edge a*
and ¬ *n' strongly−postdominates* (*sourcenode a*)
and *n' strongly−postdominates* (*targetnode a*)
proof(*atomize-elim*)
from *assms*
show ∃ *as' a as''*. *as* = *as'*@*a*#*as''* ∧ *valid-edge a* ∧
¬ *n' strongly−postdominates* (*sourcenode a*) ∧
n' strongly−postdominates (*targetnode a*)
proof(*induct rule*:*path.induct*)
case (*Cons-path n'' as nx a n*)
note *IH* = ‹⟦*n' strongly−postdominates nx*; ¬ *n' strongly−postdominates n'*⟧
⟹ ∃ *as' a as''*. *as* = *as'*@*a*#*as''* ∧ *valid-edge a* ∧
¬ *n' strongly−postdominates sourcenode a* ∧

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

lemma *Exit-no-strong-postdominator*:
⟦(*-Exit-*) *strongly−postdominates n*; *n −as→∗* (*-Exit-*)⟧ ⟹ *False*
by(*fastforce intro*:*Exit-no-postdominator path-valid-node simp*:*strong-postdominate-def*)

lemma *strong-postdominate-path-targetnode*:
assumes *n′ strongly−postdominates n* **and** *n −as→∗ n″*
and *n′* ∉ *set*(*sourcenodes as*)
shows *n′ strongly−postdominates n″*
proof −
from ‹*n′ strongly−postdominates n*› **have** *n′ postdominates n*
and ∃ *k* ≥ *1*. ∀ *as nx*. *n −as→∗ nx* ∧ *length as* ≥ *k*
⟶ *n′* ∈ *set*(*sourcenodes as*)
by(*auto simp only*:*strong-postdominate-def*)
then obtain *k* **where** *k* ≥ *1*
and *paths*:∀ *as nx*. *n −as→∗ nx* ∧ *length as* ≥ *k*
⟶ *n′* ∈ *set*(*sourcenodes as*) **by** *auto*
from ‹*n′ postdominates n*› ‹*n −as→∗ n″*› ‹*n′* ∉ *set*(*sourcenodes as*)›
have *n′ postdominates n″*
by(*rule postdominate-path-targetnode*)
{ **fix** *as′ nx* **assume** *n″ −as′→∗ nx* **and** *length as′* ≥ *k*
with ‹*n −as→∗ n″*› **have** *n −as@as′→∗ nx* **and** *length* (*as@as′*) ≥ *k*
by(*auto intro*:*path-Append*)
with *paths* **have** *n′* ∈ *set*(*sourcenodes* (*as@as′*)) **by** *fastforce*
with ‹*n′* ∉ *set*(*sourcenodes as*)› **have** *n′* ∈ *set*(*sourcenodes as′*)
by(*fastforce simp*:*sourcenodes-def*) }
with ‹*k* ≥ *1*› **have** ∃ *k* ≥ *1*. ∀ *as′ nx*. *n″ −as′→∗ nx* ∧ *length as′* ≥ *k*
⟶ *n′* ∈ *set*(*sourcenodes as′*) **by** *auto*
with ‹*n′ postdominates n″*› **show** *?thesis* **by**(*simp add*:*strong-postdominate-def*)
qed

lemma *not-strong-postdominate-successor-set*:
assumes $\neg$ *n strongly−postdominates* (*sourcenode a*) **and** *valid-node n*
and *valid-edge a*
and *all*:$\forall$ *nx* $\in$ *N*. $\exists$ *a'*. *valid-edge a'* $\wedge$ *sourcenode a'* = *sourcenode a* $\wedge$
targetnode a' = *nx* $\wedge$ *n strongly−postdominates nx*
obtains *a'* **where** *valid-edge a'* **and** *sourcenode a'* = *sourcenode a*
and *targetnode a'* $\notin$ *N*
proof(*atomize-elim*)
show $\exists$ *a'*. *valid-edge a'* $\wedge$ *sourcenode a'* = *sourcenode a* $\wedge$ *targetnode a'* $\notin$ *N*
proof(*cases n postdominates* (*sourcenode a*))
case *False*
with ‹*valid-edge a*› ‹*valid-node n*›
obtain *a'* **where** [*simp*]:*sourcenode a* = *sourcenode a'*
and *valid-edge a'* **and** $\neg$ *n postdominates targetnode a'*
by −(*erule not-postdominate-source-not-postdominate-target*)
with *all* **have** *targetnode a'* $\notin$ *N* **by**(*auto simp*:*strong-postdominate-def*)
with ‹*valid-edge a'*› **show** *?thesis* **by** *simp blast*
next
case *True*
let *?M* = {*n'*. $\exists$ *a'*. *valid-edge a'* $\wedge$ *sourcenode a'* = *sourcenode a* $\wedge$
targetnode a' = *n'*}
let *?M'* = {*n'*. $\exists$ *a'*. *valid-edge a'* $\wedge$ *sourcenode a'* = *sourcenode a* $\wedge$
targetnode a' = *n'* $\wedge$ *n strongly−postdominates n'*}
let *?N'* = (λ*n'*. *SOME i*. *i* $\geq$ *1* $\wedge$
($\forall$ *as nx*. *n'* −*as*→∗ *nx* $\wedge$ *length as* $\geq$ *i*
$\longrightarrow$ *n* $\in$ *set*(*sourcenodes as*))) ‘ *N*
obtain *k* **where** [*simp*]:*k* = *Max ?N'* **by** *simp*
have *eq*:{*x* $\in$ *?M*. (λ*n'*. *n strongly−postdominates n'*) *x*} = *?M'* **by** *auto*
from ‹*valid-edge a*› **have** *finite ?M* **by**(*simp add*:*successor-set-finite*)
hence *finite* {*x* $\in$ *?M*. (λ*n'*. *n strongly−postdominates n'*) *x*} **by** *fastforce*
with *eq* **have** *finite ?M'* **by** *simp*
from *all* **have** *N* $\subseteq$ *?M'* **by** *auto*
with ‹*finite ?M'*› **have** *finite N* **by**(*auto intro*:*finite-subset*)
hence *finite ?N'* **by** *fastforce*
show *?thesis*
proof(*rule ccontr*)
assume $\neg$ ($\exists$ *a'*. *valid-edge a'* $\wedge$ *sourcenode a'* = *sourcenode a* $\wedge$
targetnode a' $\notin$ *N*)
hence *allImp*:$\forall$ *a'*. *valid-edge a'* $\wedge$ *sourcenode a'* = *sourcenode a*
$\longrightarrow$ *targetnode a'* $\in$ *N* **by** *blast*
from *True* ‹$\neg$ *n strongly−postdominates* (*sourcenode a*)›
have *allPaths*:$\forall$ *k* $\geq$ *1*. $\exists$ *as nx*. *sourcenode a* −*as*→∗ *nx* $\wedge$ *length as* $\geq$ *k*
$\wedge$ *n* $\notin$ *set*(*sourcenodes as*) **by**(*auto simp*:*strong-postdominate-def*)
then obtain *as nx* **where** *sourcenode a* −*as*→∗ *nx*
and *length as* $\geq$ *k* + *1* **and** *n* $\notin$ *set*(*sourcenodes as*)
by (*erule-tac x*=*k* + *1* **in** *allE*) *auto*
then obtain *ax as'* **where** [*simp*]:*as* = *ax*#*as'* **and** *valid-edge ax*
and *sourcenode ax* = *sourcenode a* **and** *targetnode ax* −*as'*→∗ *nx*
by −(*erule path.cases*,*auto*)

with *allImp* **have** *targetnode ax* ∈ *N* **by** *fastforce*
with *all* **have** *n strongly−postdominates* (*targetnode ax*)
by *auto*
then obtain *k′* **where** *k′*:*k′* = (*SOME i. i* ≥ *1* ∧
(∀ *as nx. targetnode ax* −*as*→∗ *nx* ∧ *length as* ≥ *i*
⟶ *n* ∈ *set*(*sourcenodes as*))) **by** *simp*
with ‹*n strongly−postdominates* (*targetnode ax*)›
have *k′* ≥ *1* ∧ (∀ *as nx. targetnode ax* −*as*→∗ *nx* ∧ *length as* ≥ *k′*
⟶ *n* ∈ *set*(*sourcenodes as*))
by(*auto elim*!:*someI-ex simp*:*strong-postdominate-def*)
hence *k′* ≥ *1*
and *spdAll*:∀ *as nx. targetnode ax* −*as*→∗ *nx* ∧ *length as* ≥ *k′*
⟶ *n* ∈ *set*(*sourcenodes as*)
by *simp-all*
from ‹*targetnode ax* ∈ *N*› *k′* **have** *k′* ∈ *?N′* **by** *blast*
with ‹*targetnode ax* ∈ *N*› **have** *?N′* ≠ {} **by** *auto*
with ‹*k′* ∈ *?N′*› **have** *k′* ≤ *Max ?N′* **using** ‹*finite ?N′*› **by**(*fastforce intro*:*Max-ge*)
hence *k′* ≤ *k* **by** *simp*
with ‹*targetnode ax* −*as′*→∗ *nx*› ‹*length as* ≥ *k* + *1*› *spdAll*
have *n* ∈ *set*(*sourcenodes as′*)
by *fastforce*
with ‹*n* ∉ *set*(*sourcenodes as*)› **show** *False* **by**(*simp add*:*sourcenodes-def*)
qed
qed
qed

lemma *not-strong-postdominate-predecessor-successor*:
assumes ¬ *n strongly−postdominates* (*sourcenode a*)
and *valid-node n* **and** *valid-edge a*
obtains *a′* **where** *valid-edge a′* **and** *sourcenode a′* = *sourcenode a*
and ¬ *n strongly−postdominates* (*targetnode a′*)
proof(*atomize-elim*)
show ∃ *a′. valid-edge a′* ∧ *sourcenode a′* = *sourcenode a* ∧
¬ *n strongly−postdominates* (*targetnode a′*)
proof(*rule ccontr*)
assume ¬ (∃ *a′. valid-edge a′* ∧ *sourcenode a′* = *sourcenode a* ∧
¬ *n strongly−postdominates targetnode a′*)
hence *all*:∀ *a′. valid-edge a′* ∧ *sourcenode a′* = *sourcenode a* ⟶
n strongly−postdominates (*targetnode a′*) **by** *auto*
let *?N* = {*n′*. ∃ *a′. sourcenode a′* = *sourcenode a* ∧ *valid-edge a′* ∧
targetnode a′ = *n′*}
from *all* **have** ∀ *nx* ∈ *?N*. ∃ *a′. valid-edge a′* ∧ *sourcenode a′* = *sourcenode a* ∧
targetnode a′ = *nx* ∧ *n strongly−postdominates nx*
by *auto*
with *assms* **obtain** *a′* **where** *valid-edge a′*

and *sourcenode a′ = sourcenode a* **and** *targetnode a′* ∉ *?N*
by(*erule not-strong-postdominate-successor-set*)
thus *False* **by** *auto*
qed
qed

end

end

1.4 CFG well-formedness

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

1.4.1 Well-formedness of the abstract CFG

locale *CFG-wf* = *CFG sourcenode targetnode kind valid-edge Entry*
for *sourcenode* :: *′edge* ⇒ *′node* **and** *targetnode* :: *′edge* ⇒ *′node*
and *kind* :: *′edge* ⇒ *′state edge-kind* **and** *valid-edge* :: *′edge* ⇒ *bool*
and *Entry* :: *′node* (‹′(′-Entry′-′)›) +
fixes *Def*::*′node* ⇒ *′var set*
fixes *Use*::*′node* ⇒ *′var set*
fixes *state-val*::*′state* ⇒ *′var* ⇒ *′val*
assumes *Entry-empty*:*Def* (*-Entry-*) = {} ∧ *Use* (*-Entry-*) = {}
and *CFG-edge-no-Def-equal*:
⟦*valid-edge a*; *V* ∉ *Def* (*sourcenode a*); *pred* (*kind a*) *s*⟧
⟹ *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*
and *CFG-edge-transfer-uses-only-Use*:
⟦*valid-edge a*; ∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s′ V*;
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*;
∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s′ V*⟧
⟹ *pred* (*kind a*) *s′*
and *deterministic*:⟦*valid-edge a*; *valid-edge a′*; *sourcenode a* = *sourcenode a′*;
targetnode a ≠ *targetnode a′*⟧
⟹ ∃ *Q Q′*. *kind a* = $(Q)_{\surd}$ ∧ *kind a′* = $(Q')_{\surd}$ ∧
(∀ *s*. (*Q s* ⟶ ¬ *Q′ s*) ∧ (*Q′ s* ⟶ ¬ *Q s*))

begin

lemma [*dest*!]: *V* ∈ *Use* (*-Entry-*) ⟹ *False*
by(*simp add*:*Entry-empty*)

lemma [*dest*!]: *V* ∈ *Def* (*-Entry-*) ⟹ *False*

by(*simp add*:*Entry-empty*)

lemma *CFG-path-no-Def-equal*:
  ⟦*n* −*as*→∗ *n'*; ∀ *n* ∈ *set* (*sourcenodes as*). *V* ∉ *Def n*; *preds* (*kinds as*) *s*⟧
    ⟹ *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* = ‹⋀*s*. ⟦∀ *n*∈*set* (*sourcenodes as*). *V* ∉ *Def n*; *preds* (*kinds as*) *s*⟧ ⟹
          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 ‹∀ *n*∈*set* (*sourcenodes* (*a*#*as*)). *V* ∉ *Def n*›
    have *noDef*:*V* ∉ *Def* (*sourcenode a*)
    and *all*:∀ *n*∈*set* (*sourcenodes as*). *V* ∉ *Def n*
    by(*auto simp*:*sourcenodes-def*)
  from ‹*valid-edge a*› *noDef* ‹*pred* (*kind a*) *s*›
  have *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*
    by(*rule CFG-edge-no-Def-equal*)
  with *IH*[*OF all* ‹*preds* (*kinds as*) (*transfer* (*kind a*) *s*)›] **show** *?case*
    by(*simp add*:*kinds-def*)
qed

end

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

1.4.2 New well-formedness lemmas using (*-Exit-*)

locale *CFGExit-wf* =
  CFG-wf sourcenode targetnode kind valid-edge Entry Def Use state-val +
  CFGExit sourcenode targetnode kind valid-edge Entry Exit
  for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
  and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
  and *Entry* :: *'node* (‹*'*('*-Entry'-*')›) **and** *Def* :: *'node* ⇒ *'var set*
  and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
  and *Exit* :: *'node* (‹*'*('*-Exit'-*')›) +
  assumes *Exit-empty*:*Def* (*-Exit-*) = {} ∧ *Use* (*-Exit-*) = {}

begin

lemma *Exit-Use-empty* [*dest*!]: *V* ∈ *Use* (*-Exit-*) ⟹ *False*
by(*simp add*:*Exit-empty*)

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

end

end

1.5 CFG and semantics conform

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

locale *CFG-semantics-wf* = *CFG sourcenode targetnode kind valid-edge Entry*
  for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
  and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
  and *Entry* :: *'node* (‹'('-Entry'-')›) +
  fixes *sem*::*'com* ⇒ *'state* ⇒ *'com* ⇒ *'state* ⇒ *bool*
    (‹((1⟨-,/-⟩) ⇒/ (1⟨-,/-⟩))› [0,0,0,0] 81)
  fixes *identifies*::*'node* ⇒ *'com* ⇒ *bool* (‹- ≜ -› [51,0] 80)
  assumes *fundamental-property*:
    ⟦n ≜ c; ⟨c,s⟩ ⇒ ⟨c',s'⟩⟧ ⟹
      ∃ n' as. n −as→∗ n' ∧ transfers (kinds as) s = s' ∧ preds (kinds as) s ∧
          n' ≜ c'

end

1.6 Dynamic data dependence

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

context *CFG-wf* **begin**

definition *dyn-data-dependence* ::
  'node ⇒ *'var* ⇒ *'node* ⇒ *'edge list* ⇒ *bool* (‹- *influences* - *in* - *via* -› [51,0,0])
where *n influences V in n' via as* ≡
    ((V ∈ Def n) ∧ (V ∈ Use n') ∧ (n −as→∗ n') ∧
     (∃ a' as'. (as = a'#as') ∧ (∀ n'' ∈ set (sourcenodes as'). V ∉ Def n'')))

lemma *dyn-influence-Cons-source*:
  n influences V in n' via a#as ⟹ *sourcenode a* = *n*
  by(*simp add*:*dyn-data-dependence-def*,*auto elim*:*path.cases*)

lemma *dyn-influence-source-notin-tl-edges*:
  assumes *n influences V in n' via a#as*
  shows *n* ∉ *set* (*sourcenodes as*)
proof(*rule ccontr*)

assume ¬ *n* ∉ *set* (*sourcenodes as*)
hence *n* ∈ *set* (*sourcenodes as*) **by** *simp*
from ‹*n influences V in n′ via a#as*› **have** ∀ *n″* ∈ *set* (*sourcenodes as*). *V* ∉ *Def n″*
and *V* ∈ *Def n* **by**(*simp-all add*:*dyn-data-dependence-def*)
from ‹∀ *n″* ∈ *set* (*sourcenodes as*). *V* ∉ *Def n″*›
‹*n* ∈ *set* (*sourcenodes as*)› **have** *V* ∉ *Def n* **by** *simp*
with ‹*V* ∈ *Def n*› **show** *False* **by** *simp*
qed

lemma *dyn-influence-only-first-edge*:
assumes *n influences V in n′ via a#as* **and** *preds* (*kinds* (*a#as*)) *s*
shows *state-val* (*transfers* (*kinds* (*a#as*)) *s*) *V* =
state-val (*transfer* (*kind a*) *s*) *V*
proof −
from ‹*preds* (*kinds* (*a#as*)) *s*› **have** *preds* (*kinds as*) (*transfer* (*kind a*) *s*)
by(*simp add*:*kinds-def*)
from ‹*n influences V in n′ via a#as*› **have** *n* −*a#as*→∗ *n′*
and ∀ *n″* ∈ *set* (*sourcenodes as*). *V* ∉ *Def n″*
by(*simp-all add*:*dyn-data-dependence-def*)
from ‹*n* −*a#as*→∗ *n′*› **have** *n* = *sourcenode a* **and** *targetnode a* −*as*→∗ *n′*
by(*auto elim*:*path-split-Cons*)
from ‹*n influences V in n′ via a#as*› ‹*n* = *sourcenode a*›
have *sourcenode a* ∉ *set* (*sourcenodes as*)
by(*fastforce intro*!:*dyn-influence-source-notin-tl-edges*)
{ fix *n″* **assume** *n″* ∈ *set* (*sourcenodes as*)
with ‹*sourcenode a* ∉ *set* (*sourcenodes as*)› ‹*n* = *sourcenode a*›
have *n″* ≠ *n* **by**(*fastforce simp*:*sourcenodes-def*)
with ‹∀ *n″* ∈ *set* (*sourcenodes as*). *V* ∉ *Def n″*› ‹*n″* ∈ *set* (*sourcenodes as*)›
have *V* ∉ *Def n″* **by**(*auto simp*:*sourcenodes-def*) **}**
hence ∀ *n″* ∈ *set* (*sourcenodes as*). *V* ∉ *Def n″* **by** *simp*
with ‹*targetnode a* −*as*→∗ *n′*› ‹*preds* (*kinds as*) (*transfer* (*kind a*) *s*)›
have *state-val* (*transfers* (*kinds as*) (*transfer* (*kind a*) *s*)) *V* =
state-val (*transfer* (*kind a*) *s*) *V*
by −(*rule CFG-path-no-Def-equal*)
thus *?thesis* **by**(*auto simp*:*kinds-def*)
qed

end

end

1.7 Dynamic Standard Control Dependence

theory *DynStandardControlDependence* **imports** *Postdomination* **begin**

context *Postdomination* **begin**

definition
dyn-standard-control-dependence :: *'node* ⇒ *'node* ⇒ *'edge list* ⇒ *bool*
(‹- $controls_s$ - *via* -› [*51*,*0*,*0*])
where *dyn-standard-control-dependence-def*:*n* $controls_s$ *n'* *via as* ≡
(∃ *a a' as'*. (*as* = *a*#*as'*) ∧ (*n'* ∉ *set*(*sourcenodes as*)) ∧ (*n* −*as*→∗ *n'*) ∧
(*n'* *postdominates* (*targetnode a*)) ∧
(*valid-edge a'*) ∧ (*sourcenode a'* = *n*) ∧
(¬ *n'* *postdominates* (*targetnode a'*)))

lemma *Exit-not-dyn-standard-control-dependent*:
assumes *control*:*n* $controls_s$ (*-Exit-*) *via as* **shows** *False*
proof −
from *control* **obtain** *a as'* **where** *path*:*n* −*as*→∗ (*-Exit-*) **and** *as*:*as* = *a*#*as'*
and *pd*:(*-Exit-*) *postdominates* (*targetnode a*)
by(*auto simp*:*dyn-standard-control-dependence-def*)
from *path as* **have** *n* −[]@*a*#*as'*→∗ (*-Exit-*) **by** *simp*
hence *valid-edge a* **by**(*fastforce dest*:*path-split*)
with *pd* **show** *False* **by** −(*rule Exit-no-postdominator*,*auto*)
qed

lemma *dyn-standard-control-dependence-def-variant*:
n $controls_s$ *n'* *via as* = ((*n* −*as*→∗ *n'*) ∧ (*n* ≠ *n'*) ∧
(¬ *n'* *postdominates n*) ∧ (*n'* ∉ *set*(*sourcenodes as*)) ∧
(∀ *n''* ∈ *set*(*targetnodes as*). *n'* *postdominates n''*))
proof
assume (*n* −*as*→∗ *n'*) ∧ (*n* ≠ *n'*) ∧ (¬ *n'* *postdominates n*) ∧
(*n'* ∉ *set*(*sourcenodes as*)) ∧
(∀ *n''*∈*set* (*targetnodes as*). *n'* *postdominates n''*)
hence *path*:*n* −*as*→∗ *n'* **and** *noteq*:*n* ≠ *n'*
and *not-pd*:¬ *n'* *postdominates n*
and *notin*:*n'* ∉ *set*(*sourcenodes as*)
and *all*:∀ *n''*∈*set* (*targetnodes as*). *n'* *postdominates n''*
by *auto*
have *notExit*:*n* ≠ (*-Exit-*)
proof
assume *n* = (*-Exit-*)
with *path* **have** *n* = *n'* **by**(*fastforce dest*:*path-Exit-source*)
with *noteq* **show** *False* **by** *simp*
qed
from *path* **have** *valid*:*valid-node n* **and** *valid'*:*valid-node n'*
by(*auto dest*:*path-valid-node*)
show *n* $controls_s$ *n'* *via as*
proof(*cases as*)
case *Nil*
with *path valid not-pd notExit* **have** *False*
by(*fastforce elim*:*path.cases dest*:*postdominate-refl*)
thus *?thesis* **by** *simp*

next
  case (*Cons ax asx*)
  with *all* **have** *pd*:*n′ postdominates targetnode ax*
    by(*auto simp*:*targetnodes-def*)
  from *path Cons* **have** *source*:*n* = *sourcenode ax*
    and *path2*:*targetnode ax* −*asx*→∗ *n′*
    by(*auto intro*:*path-split-Cons*)
  show *?thesis*
  proof(*cases* ∃ *as′*. *n* −*as′*→∗ (-*Exit*-))
    case *True*
    with *not-pd valid valid′* **obtain** *as′* **where** *path′*:*n* −*as′*→∗ (-*Exit*-)
      and *not-isin*:*n′* ∉ *set* (*sourcenodes as′*)
      by(*auto simp*:*postdominate-def*)
    have *as′* ≠ []
    proof
      assume *as′* = []
      with *path′* **have** *n* = (-*Exit*-) **by**(*auto elim*:*path.cases*)
      with *notExit* **show** *False* **by** *simp*
    qed
    then obtain *a′ as″* **where** *as′*:*as′* = *a′*#*as″*
      by(*cases as′*,*auto elim*:*path.cases*)
    with *path′* **have** *n* −[]@*a′*#*as″*→∗ (-*Exit*-) **by** *simp*
    hence *source′*:*n* = *sourcenode a′*
      and *valid-edge*:*valid-edge a′*
      and *path2′*:*targetnode a′* −*as″*→∗ (-*Exit*-)
      by(*fastforce dest*:*path-split*)+
    from *path2′ not-isin as′ valid′*
    have ¬ *n′ postdominates* (*targetnode a′*)
      by(*auto simp*:*postdominate-def sourcenodes-def*)
    with *pd path Cons source source′ notin valid-edge* **show** *?thesis*
      by(*auto simp*:*dyn-standard-control-dependence-def*)
  next
    case *False*
    with *valid valid′* **have** *n′ postdominates n*
      by(*auto simp*:*postdominate-def*)
    with *not-pd* **have** *False* **by** *simp*
    thus *?thesis* **by** *simp*
  qed
qed
next
  assume *n controls*$_s$ *n′ via as*
  then obtain *a nx a′ nx′ as′* **where** *notin*:*n′* ∉ *set*(*sourcenodes as*)
    and *path*:*n* −*as*→∗ *n′* **and** *as*:*as* = *a*#*as′* **and** *valid-edge*:*valid-edge a′*
    and *pd*:*n′ postdominates* (*targetnode a*)
    and *source′*:*sourcenode a′* = *n*
    and *not-pd*:¬ *n′ postdominates* (*targetnode a′*)
    by(*auto simp*:*dyn-standard-control-dependence-def*)
  from *path as* **have** *source*:*sourcenode a* = *n* **by**(*auto elim*:*path.cases*)
  from *path as* **have** *notExit*:*n* ≠ (-*Exit*-) **by**(*auto elim*:*path.cases*)

from *path* **have** *valid*:*valid-node n* **and** *valid'*:*valid-node n'*
by(*auto dest*:*path-valid-node*)
from *notin as source* **have** *noteq*:$n \neq n'$
by(*fastforce simp*:*sourcenodes-def*)
from *valid-edge* **have** *valid-target'*:*valid-node* (*targetnode a'*)
by(*fastforce simp*:*valid-node-def*)
{ **assume** *pd'*:*n' postdominates n*
hence *all*:$\forall$ *as. n* $-as\rightarrow*$ (*-Exit-*) $\longrightarrow$ $n' \in$ *set* (*sourcenodes as*)
by(*auto simp*:*postdominate-def*)
from *not-pd valid' valid-target'* **obtain** *as'*
where *targetnode a'* $-as'\rightarrow*$ (*-Exit-*)
by(*auto simp*:*postdominate-def*)
with *source' valid-edge* **have** *path'*:*n* $-a'\#as'\rightarrow*$ (*-Exit-*)
by(*fastforce intro*:*Cons-path*)
with *all* **have** $n' \in$ *set* (*sourcenodes* (*a'#as'*)) **by** *blast*
with *source'* **have** $n' = n \vee n' \in$ *set* (*sourcenodes as'*)
by(*fastforce simp*:*sourcenodes-def*)
hence *False*
proof
assume $n' = n$
with *noteq* **show** *?thesis* **by** *simp*
next
assume *isin*:$n' \in$ *set* (*sourcenodes as'*)
from *path'* **have** *path2*:*targetnode a'* $-as'\rightarrow*$ (*-Exit-*)
by(*fastforce elim*:*path-split-Cons*)
thus *?thesis*
proof(*cases as'* = [])
case *True*
with *path2* **have** *targetnode a'* = (*-Exit-*) **by**(*auto elim*:*path.cases*)
with *valid-edge all source'* **have** $n' = n$
by(*fastforce dest*:*path-edge simp*:*sourcenodes-def*)
with *noteq* **show** *?thesis* **by** *simp*
next
case *False*
from *path2 not-pd valid' valid-edge* **obtain** *as''*
where *path''*:*targetnode a'* $-as''\rightarrow*$ (*-Exit-*)
and *notin*:$n' \notin$ *set* (*sourcenodes as''*)
by(*auto simp*:*postdominate-def*)
from *valid-edge path''* **have** *sourcenode a'* $-a'\#as''\rightarrow*$ (*-Exit-*)
by(*fastforce intro*:*Cons-path*)
with *all source'* **have** $n' \in$ *set* (*sourcenodes* ([*a'*]@*as''*)) **by** *simp*
with *source'* **have** $n' = n \vee n' \in$ *set* (*sourcenodes as''*)
by(*auto simp*:*sourcenodes-def*)
thus *?thesis*
proof
assume $n' = n$
with *noteq* **show** *?thesis* **by** *simp*
next
assume $n' \in$ *set* (*sourcenodes as''*)

```
        with notin show ?thesis by simp
      qed
    qed
  qed }
hence not-pd′:¬ n′ postdominates n by blast
{ fix n′′ assume n′′ ∈ set (targetnodes as)
  with as source have n′′ = targetnode a ∨ n′′ ∈ set (targetnodes as′)
    by(auto simp:targetnodes-def)
  hence n′ postdominates n′′
  proof
    assume n′′ = targetnode a
    with pd show ?thesis by simp
  next
    assume isin:n′′ ∈ set (targetnodes as′)
    hence ∃ ni ∈ set (targetnodes as′). ni = n′′ by simp
    then obtain ns ns′ where targets:targetnodes as′ = ns@n′′#ns′
      and all-noteq:∀ ni ∈ set ns′. ni ≠ n′′
      by(fastforce elim!:rightmost-element-property)
    from targets obtain xs ax ys where ys:ns′ = targetnodes ys
      and as′:as′ = xs@ax#ys and target′′:targetnode ax = n′′
      by(fastforce elim:map-append-append-maps simp:targetnodes-def)
    from all-noteq ys have notin-target:n′′ ∉ set(targetnodes ys)
      by auto
    from path as have n −[]@a#as′→∗ n′ by simp
    hence targetnode a −as′→∗ n′
      by(fastforce dest:path-split)
    with isin have path′:targetnode a −as′→∗ n′
      by(fastforce split:if-split-asm simp:targetnodes-def)
    with as′ target′′ have path1:targetnode a −xs→∗ sourcenode ax
      and valid-edge′:valid-edge ax
      and path2:n′′ −ys→∗ n′
      by(auto intro:path-split)
   from valid-edge′ have sourcenode ax −[ax]→∗ targetnode ax by(rule path-edge)
    with path1 target′′ have path-n′′:targetnode a −xs@[ax]→∗ n′′
      by(fastforce intro:path-Append)
    from notin as as′ have notin′:n′∉ set (sourcenodes (xs@[ax]))
      by(simp add:sourcenodes-def)
    show ?thesis
    proof(rule ccontr)
      assume ¬ n′ postdominates n′′
      with valid′ target′′ valid-edge′ obtain asx′
        where Exit-path:n′′ −asx′→∗ (-Exit-)
        and notin′′:n′ ∉ set(sourcenodes asx′) by(auto simp:postdominate-def)
      from path-n′′ Exit-path
      have Exit-path′:targetnode a −(xs@[ax])@asx′→∗ (-Exit-)
        by(fastforce intro:path-Append)
      from notin′ notin′′ have n′ ∉ set(sourcenodes (xs@ax#asx′))
        by(simp add:sourcenodes-def)
      with pd Exit-path′ show False by(simp add:postdominate-def)
```

qed
qed }
with *path not-pd′ notin noteq* **show** $(n -as\rightarrow* n') \wedge (n \neq n') \wedge$
$(\neg\ n'$ *postdominates* $n) \wedge (n' \notin set(sourcenodes\ as)) \wedge$
$(\forall n'' \in set(targetnodes\ as).\ n'$ *postdominates* $n'')$ **by** *blast*
qed

lemma *which-node-dyn-standard-control-dependence-source*:
assumes *path*:(-*Entry*-) $-as@a\#as'\rightarrow*$ *n*
and *Exit-path*:$n -as''\rightarrow*$ (-*Exit*-) **and** *source*:*sourcenode a* = n'
and *source′*:*sourcenode* $a' = n'$
and *no-source*:$n \notin set(sourcenodes\ (a\#as'))$ **and** *valid-edge′*:*valid-edge* a'
and *inner-node*:*inner-node n* **and** *not-pd*:¬ *n postdominates* (*targetnode* a')
and *last*:$\forall ax\ ax'.\ ax \in set\ as' \wedge sourcenode\ ax = sourcenode\ ax' \wedge$
valid-edge $ax' \longrightarrow$ *n postdominates targetnode* ax'
shows n' $controls_s$ *n via* $a\#as'$
proof −
from *path source* **have** *path-n′n*:$n' -a\#as'\rightarrow*$ *n* **by**(*fastforce dest*:*path-split-second*)
from *path* **have** *valid-edge*:*valid-edge a* **by**(*fastforce intro*:*path-split*)
show *?thesis*
proof(*cases n postdominates* (*targetnode a*))
case *True*
with *path-n′n not-pd no-source source source′ valid-edge′* **show** *?thesis*
by(*auto simp*:*dyn-standard-control-dependence-def*)
next
case *False*
hence *not-pd′*:¬ *n postdominates* (*targetnode a*) **.**
show *?thesis*
proof(*cases* $as' = []$)
case *True*
with *path-n′n* **have** *targetnode a* = *n* **by**(*fastforce elim*:*path.cases*)
with *inner-node* **have** *n postdominates* (*targetnode a*)
by(*cases n* = (-*Exit*-),*auto intro*:*postdominate-refl simp*:*inner-node-def*)
with *not-pd path-n′n no-source source source′ valid-edge′* **show** *?thesis*
by(*fastforce simp*:*dyn-standard-control-dependence-def*)
next
case *False*
hence *notempty′*:$as' \neq []$ **.**
with *path* **have** *path-nxn*:*targetnode a* $-as'\rightarrow*$ *n*
by(*fastforce dest*:*path-split*)
from *Exit-path path-nxn* **have** $\exists as.$ *targetnode a* $-as\rightarrow*$ (-*Exit*-)
by(*fastforce dest*:*path-Append*)
with *not-pd′ inner-node valid-edge* **obtain** *asx*
where *path-Exit*:*targetnode a* $-asx\rightarrow*$ (-*Exit*-)
and *notin*:$n \notin set\ (sourcenodes\ asx)$
by(*auto simp*:*postdominate-def inner-is-valid*)
show *?thesis*
proof(*cases* $\exists asx'.\ asx = as'@asx'$)

case *True*
then obtain *asx′* **where** *asx*:*asx* = *as′*@*asx′* **by** *blast*
from *path notempty′* **have** *targetnode a* −*as′*→∗ *n*
by(*fastforce dest*:*path-split*)
with *path-Exit inner-node asx notempty′*
obtain *a′′ as′′* **where** *asx′* = *a′′*#*as′′* ∧ *sourcenode a′′* = *n*
apply(*cases asx′*)
apply(*fastforce dest*:*path-det*)
by(*fastforce dest*:*path-split path-det*)
with *asx* **have** *n* ∈ *set*(*sourcenodes asx*) **by**(*simp add*:*sourcenodes-def*)
with *notin* **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
next
case *False*
hence *all*:∀ *asx′*. *asx* ≠ *as′*@*asx′* **by** *simp*
then obtain *j asx′* **where** *asx*:*asx* = (*take j as′*)@*asx′*
and *length*:*j* < *length as′*
and *not-more*:∀ *k* > *j*. ∀ *asx′′*. *asx* ≠ (*take k as′*)@*asx′′*
by(*auto elim*:*path-split-general*)
from *asx length* **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*:*asx* = *as′1*@*asx′*
and *take*:*as′1* = *take j as′*
and *x*:*as′* = *as′1*@*as′′* **and** *x′*:*as′′* ≠ [] **by** *blast*
from *x x′* **obtain** *a1 as′2* **where** *as′*:*as′* = *as′1*@*a1*#*as′2* **and** *as′′* = *a1*#*as′2*
by(*cases as′′*) *auto*
have *notempty-x′*:*asx′* ≠ []
proof(*cases asx′* = [])
case *True*
with *asx as′* **have** *as′* = *asx*@*a1*#*as′2* **by** *simp*
with *path-n′n* **have** *n′* −(*a*#*asx*)@*a1*#*as′2*→∗ *n*
by *simp*
hence *n′* −*a*#*asx*→∗ *sourcenode a1*
and *valid-edge1*:*valid-edge a1* **by**(*fastforce elim*:*path-split*)+
hence *targetnode a* −*asx*→∗ *sourcenode a1*
by(*fastforce intro*:*path-split-Cons*)
with *path-Exit* **have** (*-Exit-*) = *sourcenode a1* **by**(*rule path-det*)
from *this*[*THEN sym*] *valid-edge1* **have** *False* **by** −(*rule Exit-source*,*simp-all*)
thus *?thesis* **by** *simp*
qed *simp*
with *asx* **obtain** *a2 asx′1*
where *asx*:*asx* = *as′1*@*a2*#*asx′1*
and *asx′*:*asx′* = *a2*#*asx′1* **by**(*cases asx′*) *auto*
from *path-n′n as′* **have** *n′* −(*a*#*as′1*)@*a1*#*as′2*→∗ *n* **by** *simp*
hence *n′* −*a*#*as′1*→∗ *sourcenode a1* **and** *valid-edge1*:*valid-edge a1*
by(*fastforce elim*:*path-split*)+
hence *path1*:*targetnode a* −*as′1*→∗ *sourcenode a1*

```
        by(fastforce intro:path-split-Cons)
      from path-Exit asx
      have targetnode a −as'1→* sourcenode a2
        and valid-edge2:valid-edge a2
        and path2:targetnode a2 −asx'1→* (-Exit-)
        by(auto intro:path-split)
      with path1 have eq12:sourcenode a1 = sourcenode a2
        by(cases as'1,auto dest:path-det)
      from asx notin have n ∉ set (sourcenodes asx'1)
        by(simp add:sourcenodes-def)
      with path2 have not-pd'2:¬ n postdominates targetnode a2
        by(cases asx'1 = [],auto simp:postdominate-def)
      from as' have a1 ∈ set as' by simp
      with eq12 last valid-edge2 have n postdominates targetnode a2 by blast
      with not-pd'2 have False by simp
      thus ?thesis by simp
    qed
  qed
 qed
qed


lemma inner-node-dyn-standard-control-dependence-predecessor:
 assumes inner-node:inner-node n
 obtains n' as where n' controls_s n via as
proof(atomize-elim)
 from inner-node obtain as' where pathExit:n −as'→* (-Exit-)
   by(fastforce dest:inner-is-valid Exit-path)
 from inner-node obtain as where pathEntry:(-Entry-) −as→* n
   by(fastforce dest:inner-is-valid Entry-path)
 with inner-node have notEmpty:as ≠ []
   by(auto elim:path.cases simp:inner-node-def)
 have ∃ a asx. (-Entry-) −a#asx→* n ∧ n ∉ set (sourcenodes (a#asx))
 proof(cases n ∈ set (sourcenodes as))
   case True
   hence ∃ n'' ∈ set(sourcenodes as). n = n'' by simp
   then obtain ns' ns'' where nodes:sourcenodes as = ns'@n#ns''
     and notin:∀ n'' ∈ set ns'. n ≠ n''
     by(fastforce elim!:split-list-first-propE)
   from nodes obtain xs ys a'
     where xs:sourcenodes xs = ns' and as:as = xs@a'#ys
     and source:sourcenode a' = n
     by(fastforce elim:map-append-append-maps simp:sourcenodes-def)
   from pathEntry as have (-Entry-) −xs@a'#ys→* n by simp
   hence path2:(-Entry-) −xs→* sourcenode a'
     by(fastforce dest:path-split)
   show ?thesis
   proof(cases xs = [])
     case True
```

```
    with path2 have (-Entry-) = sourcenode a' by(auto elim:path.cases)
    with pathEntry source notEmpty have (-Entry-) −as→∗ (-Entry-) ∧ as ≠ []
      by(auto elim:path.cases)
    hence False by(fastforce dest:path-Entry-target)
    thus ?thesis by simp
  next
    case False
    then obtain n a'' xs' where xs = a''#xs' by(cases xs) auto
    with False path2 notin xs source show ?thesis by simp blast
  qed
next
  case False
  from notEmpty obtain a as' where as = a#as' by (cases as) auto
  with False pathEntry show ?thesis by auto
qed
then obtain a asx where pathEntry':(-Entry-) −a#asx→∗ n
  and notin:n ∉ set (sourcenodes (a#asx)) by blast
show ∃ n' as. n' controls_s n via as
proof(cases ∀ a' a''. a' ∈ set asx ∧ sourcenode a' = sourcenode a'' ∧
    valid-edge a'' ⟶ n postdominates targetnode a'')
  case True
  from inner-node have not-pd:¬ n postdominates (-Exit-)
    by(fastforce intro:empty-path simp:postdominate-def sourcenodes-def)
  from pathEntry' have path':(-Entry-) −[]@a#asx→∗ n by simp
  hence eq:sourcenode a = (-Entry-)
    by(fastforce dest:path-split elim:path.cases)
  from Entry-Exit-edge obtain a' where sourcenode a' = (-Entry-)
    and targetnode a' = (-Exit-) and valid-edge a' by auto
  with path' inner-node not-pd True eq notin pathExit
  have sourcenode a controls_s n via a#asx
    by −(erule which-node-dyn-standard-control-dependence-source,auto)
  thus ?thesis by blast
next
  case False
  hence ∃ a' ∈ set asx. ∃ a''. sourcenode a' = sourcenode a'' ∧ valid-edge a'' ∧
    ¬ n postdominates targetnode a''
    by fastforce
  then obtain ax asx' asx'' where asx = asx'@ax#asx'' ∧
    (∃ a''. sourcenode ax = sourcenode a'' ∧ valid-edge a'' ∧
    ¬ n postdominates targetnode a'') ∧
    (∀ z ∈ set asx''. ¬ (∃ a''. sourcenode z = sourcenode a'' ∧ valid-edge a'' ∧
    ¬ n postdominates targetnode a''))
    by(blast elim!:rightmost-element-property)
  then obtain a'' where as':asx = asx'@ax#asx''
    and eq:sourcenode ax = sourcenode a''
    and valid-edge:valid-edge a''
    and not-pd:¬ n postdominates targetnode a''
    and last:∀ z ∈ set asx''. ¬ (∃ a''. sourcenode z = sourcenode a'' ∧
      valid-edge a'' ∧ ¬ n postdominates targetnode a'')
```

by *blast*
from *notin as'* **have** *notin'*:$n \notin$ *set* (*sourcenodes* (*ax#asx''*))
by(*auto simp*:*sourcenodes-def*)
from *as' pathEntry'* **have** (-*Entry*-) $-$(*a#asx'*)@*ax#asx''*$\rightarrow*$ *n* **by** *simp*
with *inner-node not-pd notin' eq last pathExit valid-edge*
have *sourcenode ax* $controls_s$ *n via ax#asx''*
by(*fastforce elim*!:*which-node-dyn-standard-control-dependence-source*)
thus *?thesis* **by** *blast*
qed
qed

end

end

1.8 Dynamic Weak Control Dependence

theory *DynWeakControlDependence* **imports** *Postdomination* **begin**

context *StrongPostdomination* **begin**

definition
dyn-weak-control-dependence :: *'node* $\Rightarrow$ *'node* $\Rightarrow$ *'edge list* $\Rightarrow$ *bool*
(‹- *weakly controls* - *via* -› [*51*,*0*,*0*])
where *dyn-weak-control-dependence-def*:*n weakly controls n' via as* $\equiv$
($\exists$ *a a' as'*. (*as* = *a#as'*) $\wedge$ (*n'* $\notin$ *set*(*sourcenodes as*)) $\wedge$ (*n* $-$*as*$\rightarrow*$ *n'*) $\wedge$
(*n' strongly*$-$*postdominates* (*targetnode a*)) $\wedge$
(*valid-edge a'*) $\wedge$ (*sourcenode a'* = *n*) $\wedge$
($\neg$ *n' strongly*$-$*postdominates* (*targetnode a'*)))

lemma *Exit-not-dyn-weak-control-dependent*:
assumes *control*:*n weakly controls* (-*Exit*-) *via as* **shows** *False*
proof $-$
from *control* **obtain** *as a as'* **where** *path*:*n* $-$*as*$\rightarrow*$ (-*Exit*-) **and** *as*:*as* = *a#as'*
and *pd*:(-*Exit*-) *postdominates* (*targetnode a*)
by(*auto simp*:*dyn-weak-control-dependence-def strong-postdominate-def*)
from *path as* **have** *n* $-$[]@*a#as'*$\rightarrow*$ (-*Exit*-) **by** *simp*
hence *valid-edge a* **by**(*fastforce dest*:*path-split*)
with *pd* **show** *False* **by** $-$(*rule Exit-no-postdominator*,*auto*)
qed

end

end

Chapter 2

Dynamic Slicing

2.1 Dynamic Program Dependence Graph

theory *DynPDG* **imports**
../Basic/DynDataDependence
../Basic/CFGExit-wf
../Basic/DynStandardControlDependence
../Basic/DynWeakControlDependence
begin

2.1.1 The dynamic PDG

locale *DynPDG* =
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (‹'('-Entry'-')›) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *Exit* :: *'node* (‹'('-Exit'-')›) +
fixes *dyn-control-dependence* :: *'node* ⇒ *'node* ⇒ *'edge list* ⇒ *bool*
(‹- *controls* - *via* -› [*51*,*0*,*0*])
assumes *Exit-not-dyn-control-dependent*:*n controls n' via as* ⟹ *n'* ≠ (-*Exit*-)
assumes *dyn-control-dependence-path*:
n controls n' via as ⟹ *n* −*as*→∗ *n'* ∧ *as* ≠ []

begin

inductive *cdep-edge* :: *'node* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*
(‹- −-→$_{cd}$ -› [*51*,*0*,*0*] *80*)
and *ddep-edge* :: *'node* ⇒ *'var* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*
(‹- −'{-'}-→$_{dd}$ -› [*51*,*0*,*0*,*0*] *80*)
and *DynPDG-edge* :: *'node* ⇒ *'var option* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*

where
— Syntax

$n\ -as\rightarrow_{cd}\ n' == DynPDG\text{-}edge\ n\ None\ as\ n'$
$|\ n\ -\{V\}as\rightarrow_{dd}\ n' == DynPDG\text{-}edge\ n\ (Some\ V)\ as\ n'$

— Rules
| *DynPDG-cdep-edge*:
$n\ controls\ n'\ via\ as \Longrightarrow n\ -as\rightarrow_{cd}\ n'$

| *DynPDG-ddep-edge*:
$n\ influences\ V\ in\ n'\ via\ as \Longrightarrow n\ -\{V\}as\rightarrow_{dd}\ n'$

inductive *DynPDG-path* :: *'node* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*
(‹- --→$_d$∗ -› [51,0,0] 80)

where *DynPDG-path-Nil*:
$valid\text{-}node\ n \Longrightarrow n\ -[]\rightarrow_{d}*\ n$

| *DynPDG-path-Append-cdep*:
$[\![n\ -as\rightarrow_{d}*\ n''; n''\ -as'\rightarrow_{cd}\ n']\!] \Longrightarrow n\ -as@as'\rightarrow_{d}*\ n'$

| *DynPDG-path-Append-ddep*:
$[\![n\ -as\rightarrow_{d}*\ n''; n''\ -\{V\}as'\rightarrow_{dd}\ n']\!] \Longrightarrow n\ -as@as'\rightarrow_{d}*\ n'$

lemma *DynPDG-empty-path-eq-nodes*:$n\ -[]\rightarrow_{d}*\ n' \Longrightarrow n = n'$
apply − **apply**(*erule DynPDG-path.cases*)
 apply *simp*
 apply(*auto elim*:*DynPDG-edge.cases dest*:*dyn-control-dependence-path*)
by(*auto elim*:*DynPDG-edge.cases simp*:*dyn-data-dependence-def*)

lemma *DynPDG-path-cdep*:$n\ -as\rightarrow_{cd}\ n' \Longrightarrow n\ -as\rightarrow_{d}*\ n'$
apply(*subgoal-tac* $n\ -[]@as\rightarrow_{d}*\ n'$)
 apply *simp*
apply(*rule DynPDG-path-Append-cdep*, *rule DynPDG-path-Nil*)
by(*auto elim*!:*DynPDG-edge.cases dest*:*dyn-control-dependence-path path-valid-node*)

lemma *DynPDG-path-ddep*:$n\ -\{V\}as\rightarrow_{dd}\ n' \Longrightarrow n\ -as\rightarrow_{d}*\ n'$
apply(*subgoal-tac* $n\ -[]@as\rightarrow_{d}*\ n'$)
 apply *simp*
apply(*rule DynPDG-path-Append-ddep*, *rule DynPDG-path-Nil*)
by(*auto elim*!:*DynPDG-edge.cases dest*:*path-valid-node simp*:*dyn-data-dependence-def*)

lemma *DynPDG-path-Append*:
$[\![n''\ -as'\rightarrow_{d}*\ n'; n\ -as\rightarrow_{d}*\ n'']\!] \Longrightarrow n\ -as@as'\rightarrow_{d}*\ n'$
apply(*induct rule*:*DynPDG-path.induct*)
 apply(*auto intro*:*DynPDG-path.intros*)
 apply(*rotate-tac 1*,*drule DynPDG-path-Append-cdep*,*simp*+)
apply(*rotate-tac 1*,*drule DynPDG-path-Append-ddep*,*simp*+)

done

lemma *DynPDG-path-Exit*:$[\![n\ -as\rightarrow_d*\ n';\ n' = (\text{-}Exit\text{-})]\!] \Longrightarrow n = (\text{-}Exit\text{-})$
apply(*induct rule*:*DynPDG-path.induct*)
by(*auto elim*:*DynPDG-edge.cases dest*:*Exit-not-dyn-control-dependent*
simp:*dyn-data-dependence-def*)

lemma *DynPDG-path-not-inner*:
$[\![n\ -as\rightarrow_d*\ n';\ \neg\ inner\text{-}node\ n']\!] \Longrightarrow n = n'$
proof(*induct rule*:*DynPDG-path.induct*)
case (*DynPDG-path-Nil n*)
thus *?case* **by** *simp*
next
case (*DynPDG-path-Append-cdep n as n'' as' n'*)
from ‹$n''\ -as'\rightarrow_{cd}\ n'$› ‹$\neg$ *inner-node n'*› **have** *False*
apply –
apply(*erule DynPDG-edge.cases*) **apply**(*auto simp*:*inner-node-def*)
apply(*fastforce dest*:*dyn-control-dependence-path path-valid-node*)
apply(*fastforce dest*:*dyn-control-dependence-path path-valid-node*)
by(*fastforce dest*:*Exit-not-dyn-control-dependent*)
thus *?case* **by** *simp*
next
case (*DynPDG-path-Append-ddep n as n'' V as' n'*)
from ‹$n''\ -\{V\}as'\rightarrow_{dd}\ n'$› ‹$\neg$ *inner-node n'*› **have** *False*
apply –
apply(*erule DynPDG-edge.cases*)
by(*auto dest*:*path-valid-node simp*:*inner-node-def dyn-data-dependence-def*)
thus *?case* **by** *simp*
qed

lemma *DynPDG-cdep-edge-CFG-path*:
assumes $n\ -as\rightarrow_{cd}\ n'$ **shows** $n\ -as\rightarrow*\ n'$ **and** $as \neq []$
using ‹$n\ -as\rightarrow_{cd}\ n'$›
by(*auto elim*:*DynPDG-edge.cases dest*:*dyn-control-dependence-path*)

lemma *DynPDG-ddep-edge-CFG-path*:
assumes $n\ -\{V\}as\rightarrow_{dd}\ n'$ **shows** $n\ -as\rightarrow*\ n'$ **and** $as \neq []$
using ‹$n\ -\{V\}as\rightarrow_{dd}\ n'$›
by(*auto elim*:*DynPDG-edge.cases simp*:*dyn-data-dependence-def*)

lemma *DynPDG-path-CFG-path*:
$n\ -as\rightarrow_d*\ n' \Longrightarrow n\ -as\rightarrow*\ n'$
proof(*induct rule*:*DynPDG-path.induct*)
case *DynPDG-path-Nil* **thus** *?case* **by**(*rule empty-path*)
next
case (*DynPDG-path-Append-cdep n as n'' as' n'*)

from ‹$n'' -as' \rightarrow_{cd} n'$› **have** $n'' -as' \rightarrow* \ n'$
by(*rule DynPDG-cdep-edge-CFG-path*(*1*))
with ‹$n -as \rightarrow* \ n''$› **show** *?case* **by**(*rule path-Append*)
next
case (*DynPDG-path-Append-ddep n as* n'' *V* as' n')
from ‹$n'' -\{V\}as' \rightarrow_{dd} n'$› **have** $n'' -as' \rightarrow* \ n'$
by(*rule DynPDG-ddep-edge-CFG-path*(*1*))
with ‹$n -as \rightarrow* \ n''$› **show** *?case* **by**(*rule path-Append*)
qed

lemma *DynPDG-path-split*:
$n -as \rightarrow_{d} * \ n' \Longrightarrow$
$(as = [] \wedge n = n') \vee$
$(\exists n'' \ asx \ asx'. \ (n -asx \rightarrow_{cd} n'') \wedge (n'' -asx' \rightarrow_{d} * \ n') \wedge$
$(as = asx @ asx')) \vee$
$(\exists n'' \ V \ asx \ asx'. \ (n -\{V\}asx \rightarrow_{dd} n'') \wedge (n'' -asx' \rightarrow_{d} * \ n') \wedge$
$(as = asx @ asx'))$
proof(*induct rule*:*DynPDG-path.induct*)
case (*DynPDG-path-Nil n*) **thus** *?case* **by** *auto*
next
case (*DynPDG-path-Append-cdep n as* n'' as' n')
note *IH* = ‹$as = [] \wedge n = n'' \vee$
$(\exists nx \ asx \ asx'. \ n -asx \rightarrow_{cd} nx \wedge nx -asx' \rightarrow_{d} * \ n'' \wedge as = asx @ asx') \vee$
$(\exists nx \ V \ asx \ asx'. \ n -\{V\}asx \rightarrow_{dd} nx \wedge nx -asx' \rightarrow_{d} * \ n'' \wedge as = asx @ asx')$›
from *IH* **show** *?case*
proof
assume $as = [] \wedge n = n''$
with ‹$n'' -as' \rightarrow_{cd} n'$› **have** *valid-node* n'
by(*fastforce intro*:*path-valid-node*(*2*) *DynPDG-path-CFG-path*
DynPDG-path-cdep)
with ‹$as = [] \wedge n = n''$› ‹$n'' -as' \rightarrow_{cd} n'$›
have $\exists n'' \ asx \ asx'. \ n -asx \rightarrow_{cd} n'' \wedge n'' -asx' \rightarrow_{d} * \ n' \wedge as @ as' = asx @ asx'$
by(*auto intro*:*DynPDG-path-Nil*)
thus *?thesis* **by** *simp*
next
assume $(\exists nx \ asx \ asx'. \ n -asx \rightarrow_{cd} nx \wedge nx -asx' \rightarrow_{d} * \ n'' \wedge as = asx @ asx') \vee$
$(\exists nx \ V \ asx \ asx'. \ n -\{V\}asx \rightarrow_{dd} nx \wedge nx -asx' \rightarrow_{d} * \ n'' \wedge as = asx @ asx')$
thus *?thesis*
proof
assume $\exists nx \ asx \ asx'. \ n -asx \rightarrow_{cd} nx \wedge nx -asx' \rightarrow_{d} * \ n'' \wedge as = asx @ asx'$
then obtain *nx asx* asx' **where** $n -asx \rightarrow_{cd} nx$ **and** $nx -asx' \rightarrow_{d} * \ n''$
and $as = asx @ asx'$ **by** *auto*
from ‹$n'' -as' \rightarrow_{cd} n'$› **have** $n'' -as' \rightarrow_{d} * \ n'$ **by**(*rule DynPDG-path-cdep*)
with ‹$nx -asx' \rightarrow_{d} * \ n''$› **have** $nx -asx' @ as' \rightarrow_{d} * \ n'$
by(*fastforce intro*:*DynPDG-path-Append*)
with ‹$n -asx \rightarrow_{cd} nx$› ‹$as = asx @ asx'$›
have $\exists n'' \ asx \ asx'. \ n -asx \rightarrow_{cd} n'' \wedge n'' -asx' \rightarrow_{d} * \ n' \wedge as @ as' = asx @ asx'$
by *auto*

thus *?thesis* **by** *simp*
next
assume $\exists\, nx\ V\ asx\ asx'.\ n\ -\{V\}asx\rightarrow_{dd}\ nx \wedge nx\ -asx'\rightarrow_{d}*\ n'' \wedge as = asx@asx'$
then obtain $nx\ V\ asx\ asx'$ **where** $n\ -\{V\}asx\rightarrow_{dd}\ nx$ **and** $nx\ -asx'\rightarrow_{d}*\ n''$
and $as = asx@asx'$ **by** *auto*
from $\langle n''\ -as'\rightarrow_{cd}\ n'\rangle$ **have** $n''\ -as'\rightarrow_{d}*\ n'$ **by**(*rule DynPDG-path-cdep*)
with $\langle nx\ -asx'\rightarrow_{d}*\ n''\rangle$ **have** $nx\ -asx'@as'\rightarrow_{d}*\ n'$
by(*fastforce intro*:*DynPDG-path-Append*)
with $\langle n\ -\{V\}asx\rightarrow_{dd}\ nx\rangle$ $\langle as = asx@asx'\rangle$
have $\exists\, n''\ V\ asx\ asx'.\ n\ -\{V\}asx\rightarrow_{dd}\ n'' \wedge n''\ -asx'\rightarrow_{d}*\ n' \wedge as@as' = asx@asx'$
by *auto*
thus *?thesis* **by** *simp*
qed
qed
next
case (*DynPDG-path-Append-ddep n as n'' V as' n'*)
note $IH = \langle as = [] \wedge n = n'' \vee$
$(\exists\, nx\ asx\ asx'.\ n\ -asx\rightarrow_{cd}\ nx \wedge nx\ -asx'\rightarrow_{d}*\ n'' \wedge as = asx@asx') \vee$
$(\exists\, nx\ V\ asx\ asx'.\ n\ -\{V\}asx\rightarrow_{dd}\ nx \wedge nx\ -asx'\rightarrow_{d}*\ n'' \wedge as = asx@asx')\rangle$
from *IH* **show** *?case*
proof
assume $as = [] \wedge n = n''$
with $\langle n''\ -\{V\}as'\rightarrow_{dd}\ n'\rangle$ **have** *valid-node* n'
by(*fastforce intro*:*path-valid-node*(*2*) *DynPDG-path-CFG-path*
DynPDG-path-ddep)
with $\langle as = [] \wedge n = n''\rangle$ $\langle n''\ -\{V\}as'\rightarrow_{dd}\ n'\rangle$
have $\exists\, n''\ V\ asx\ asx'.\ n\ -\{V\}asx\rightarrow_{dd}\ n'' \wedge n''\ -asx'\rightarrow_{d}*\ n' \wedge as@as' = asx@asx'$
by(*fastforce intro*:*DynPDG-path-Nil*)
thus *?thesis* **by** *simp*
next
assume $(\exists\, nx\ asx\ asx'.\ n\ -asx\rightarrow_{cd}\ nx \wedge nx\ -asx'\rightarrow_{d}*\ n'' \wedge as = asx@asx') \vee$
$(\exists\, nx\ V\ asx\ asx'.\ n\ -\{V\}asx\rightarrow_{dd}\ nx \wedge nx\ -asx'\rightarrow_{d}*\ n'' \wedge as = asx@asx')$
thus *?thesis*
proof
assume $\exists\, nx\ asx\ asx'.\ n\ -asx\rightarrow_{cd}\ nx \wedge nx\ -asx'\rightarrow_{d}*\ n'' \wedge as = asx@asx'$
then obtain $nx\ asx\ asx'$ **where** $n\ -asx\rightarrow_{cd}\ nx$ **and** $nx\ -asx'\rightarrow_{d}*\ n''$
and $as = asx@asx'$ **by** *auto*
from $\langle n''\ -\{V\}as'\rightarrow_{dd}\ n'\rangle$ **have** $n''\ -as'\rightarrow_{d}*\ n'$ **by**(*rule DynPDG-path-ddep*)
with $\langle nx\ -asx'\rightarrow_{d}*\ n''\rangle$ **have** $nx\ -asx'@as'\rightarrow_{d}*\ n'$
by(*fastforce intro*:*DynPDG-path-Append*)
with $\langle n\ -asx\rightarrow_{cd}\ nx\rangle$ $\langle as = asx@asx'\rangle$
have $\exists\, n''\ asx\ asx'.\ n\ -asx\rightarrow_{cd}\ n'' \wedge n''\ -asx'\rightarrow_{d}*\ n' \wedge as@as' = asx@asx'$
by *auto*
thus *?thesis* **by** *simp*
next
assume $\exists\, nx\ V\ asx\ asx'.\ n\ -\{V\}asx\rightarrow_{dd}\ nx \wedge nx\ -asx'\rightarrow_{d}*\ n'' \wedge as =$

$asx@asx'$

then obtain $nx\ V'\ asx\ asx'$ **where** $n\ -\{V'\}asx\rightarrow_{dd}\ nx$ **and** $nx\ -asx'\rightarrow_{d}*\ n''$

and $as = asx@asx'$ **by** *auto*

from ‹$n''\ -\{V\}as'\rightarrow_{dd}\ n'$› **have** $n''\ -as'\rightarrow_{d}*\ n'$ **by**(*rule DynPDG-path-ddep*)

with ‹$nx\ -asx'\rightarrow_{d}*\ n''$› **have** $nx\ -asx'@as'\rightarrow_{d}*\ n'$

by(*fastforce intro*:*DynPDG-path-Append*)

with ‹$n\ -\{V'\}asx\rightarrow_{dd}\ nx$› ‹$as = asx@asx'$›

have $\exists n''\ V\ asx\ asx'.\ n\ -\{V\}asx\rightarrow_{dd}\ n'' \wedge n''\ -asx'\rightarrow_{d}*\ n' \wedge as@as' = asx@asx'$

by *auto*

thus *?thesis* **by** *simp*

qed

qed

qed

lemma *DynPDG-path-rev-cases* [*consumes 1*,
case-names DynPDG-path-Nil DynPDG-path-cdep-Append DynPDG-path-ddep-Append]:

$[\![n\ -as\rightarrow_{d}*\ n';\ [\![as = [];\ n = n']\!] \Longrightarrow Q;$

$\bigwedge n''\ asx\ asx'.\ [\![n\ -asx\rightarrow_{cd}\ n'';\ n''\ -asx'\rightarrow_{d}*\ n';$
$as = asx@asx']\!] \Longrightarrow Q;$

$\bigwedge V\ n''\ asx\ asx'.\ [\![n\ -\{V\}asx\rightarrow_{dd}\ n'';\ n''\ -asx'\rightarrow_{d}*\ n';$
$as = asx@asx']\!] \Longrightarrow Q]\!]$

$\Longrightarrow Q$

by(*blast dest*:*DynPDG-path-split*)

lemma *DynPDG-ddep-edge-no-shorter-ddep-edge*:

assumes *ddep*:$n\ -\{V\}as\rightarrow_{dd}\ n'$

shows $\forall as'\ a\ as''.\ tl\ as = as'@a\#as'' \longrightarrow \neg\ sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n'$

proof −

from *ddep* **have** *influence*:*n influences V in n' via as*

by(*auto elim*!:*DynPDG-edge.cases*)

then obtain *a asx* **where** *as*:$as = a\#asx$

and *notin*:$n \notin set\ (sourcenodes\ asx)$

by(*auto dest*:*dyn-influence-source-notin-tl-edges simp*:*dyn-data-dependence-def*)

from *influence as*

have *imp*:$\forall nx \in set\ (sourcenodes\ asx).\ V \notin Def\ nx$

by(*auto simp*:*dyn-data-dependence-def*)

{ fix $as'\ a'\ as''$

assume *eq*:$tl\ as = as'@a'\#as''$

and *ddep'*:$sourcenode\ a'\ -\{V\}a'\#as''\rightarrow_{dd}\ n'$

from *eq as notin* **have** *noteq*:$sourcenode\ a' \neq n$ **by**(*auto simp*:*sourcenodes-def*)

from *ddep'* **have** $V \in Def\ (sourcenode\ a')$

by(*auto elim*!:*DynPDG-edge.cases simp*:*dyn-data-dependence-def*)

with *eq as noteq imp* **have** *False* **by**(*auto simp*:*sourcenodes-def*) **}**

thus *?thesis* **by** *blast*

qed

lemma *no-ddep-same-state*:
assumes *path*:n $-as\rightarrow*$ n' **and** *Uses*:$V \in$ *Use* n' **and** *preds*:*preds* (*kinds as*) *s*
and *no-dep*:$\forall$ *as' a as''*. *as* = *as'*@*a*#*as''* $\longrightarrow \neg$ *sourcenode a* $-\{V\}a\#as''\rightarrow_{dd}$ n'
shows *state-val* (*transfers* (*kinds as*) *s*) *V* = *state-val s V*
proof −
{ **fix** n''
assume *inset*:$n'' \in$ *set* (*sourcenodes as*) **and** *Defs*:$V \in$ *Def* n''
hence $\exists$ *nx* $\in$ *set* (*sourcenodes as*). $V \in$ *Def nx* **by** *auto*
then obtain *nx ns' ns''* **where** *nodes*:*sourcenodes as* = *ns'*@*nx*#*ns''*
and *Defs'*:$V \in$ *Def nx* **and** *notDef*:$\forall$ *nx'* $\in$ *set ns''*. $V \notin$ *Def nx'*
by(*fastforce elim*!:*rightmost-element-property*)
from *nodes* **obtain** *as' a as''*
where *as''*:*sourcenodes as''* = *ns''* **and** *as*:*as*=*as'*@*a*#*as''*
and *source*:*sourcenode a* = *nx*
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from *as path* **have** *path'*:*sourcenode a* $-a\#as''\rightarrow*$ n'
by(*fastforce dest*:*path-split-second*)
from *notDef as'' source*
have $\forall$ n'' $\in$ *set* (*sourcenodes as''*). $V \notin$ *Def* n''
by(*auto simp*:*sourcenodes-def*)
with *path' Defs' Uses source*
have *influence*:*nx influences V in n' via* (*a*#*as''*)
by(*simp add*:*dyn-data-dependence-def*)
hence *nx* $\notin$ *set* (*sourcenodes as''*) **by**(*rule dyn-influence-source-notin-tl-edges*)
with *influence source*
have $\exists$ *asx a'*. *sourcenode a'* $-\{V\}a'\#asx\rightarrow_{dd}$ n' $\wedge$ *sourcenode a'* = *nx* $\wedge$
($\exists$ *asx'*. *a*#*as''* = *asx'*@*a'*#*asx*)
by(*fastforce intro*:*DynPDG-ddep-edge*)
with *nodes no-dep as* **have** *False* **by**(*auto simp*:*sourcenodes-def*) }
hence $\forall$ *n* $\in$ *set* (*sourcenodes as*). $V \notin$ *Def n* **by** *auto*
with *wf path preds* **show** *?thesis* **by**(*fastforce intro*:*CFG-path-no-Def-equal*)
qed

lemma *DynPDG-ddep-edge-only-first-edge*:
[[*n* $-\{V\}a\#as\rightarrow_{dd}$ n'; *preds* (*kinds* (*a*#*as*)) *s*]] $\Longrightarrow$
state-val (*transfers* (*kinds* (*a*#*as*)) *s*) *V* = *state-val* (*transfer* (*kind a*) *s*) *V*
apply −
apply(*erule DynPDG-edge.cases*)
apply *auto*
apply(*frule dyn-influence-Cons-source*)
apply(*frule dyn-influence-source-notin-tl-edges*)
by(*erule dyn-influence-only-first-edge*)

lemma *Use-value-change-implies-DynPDG-ddep-edge*:
assumes $n -as\rightarrow* n'$ **and** $V \in Use\ n'$ **and** $preds\ (kinds\ as)\ s$
and $preds\ (kinds\ as)\ s'$ **and** $state\text{-}val\ s\ V = state\text{-}val\ s'\ V$
and $state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V \neq$
$state\text{-}val\ (transfers\ (kinds\ as)\ s')\ V$
obtains $as'\ a\ as''$ **where** $as = as'@a\#as''$
and $sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n'$
and $state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V =$
$state\text{-}val\ (transfers\ (kinds\ (as'@[a]))\ s)\ V$
and $state\text{-}val\ (transfers\ (kinds\ as)\ s')\ V =$
$state\text{-}val\ (transfers\ (kinds\ (as'@[a]))\ s')\ V$
proof(*atomize-elim*)
show $\exists\, as'\ a\ as''.\ as = as'@a\#as'' \wedge$
$sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n' \wedge$
$state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V =$
$state\text{-}val\ (transfers\ (kinds\ (as'@[a]))\ s)\ V \wedge$
$state\text{-}val\ (transfers\ (kinds\ as)\ s')\ V =$
$state\text{-}val\ (transfers\ (kinds\ (as'@[a]))\ s')\ V$
proof($cases\ \forall\, as'\ a\ as''.\ as = as'@a\#as'' \longrightarrow$
$\neg\ sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n'$)
case *True*
with ‹$n -as\rightarrow* n'$› ‹$V \in Use\ n'$› ‹$preds\ (kinds\ as)\ s$› ‹$preds\ (kinds\ as)\ s'$›
have $state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V = state\text{-}val\ s\ V$
and $state\text{-}val\ (transfers\ (kinds\ as)\ s')\ V = state\text{-}val\ s'\ V$
by(*auto intro*:*no-ddep-same-state*)
with ‹$state\text{-}val\ s\ V = state\text{-}val\ s'\ V$›
‹$state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V \neq state\text{-}val\ (transfers\ (kinds\ as)\ s')\ V$›
show *?thesis* **by** *simp*
next
case *False*
then obtain $as'\ a\ as''$ **where** [*simp*]:$as = as'@a\#as''$
and $sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n'$ **by** *auto*
from ‹$preds\ (kinds\ as)\ s$› **have** $preds\ (kinds\ (a\#as''))\ (transfers\ (kinds\ as')\ s)$
by(*simp add*:*kinds-def preds-split*)
with ‹$sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n'$› **have** *all*:
$state\text{-}val\ (transfers\ (kinds\ (a\#as''))\ (transfers\ (kinds\ as')\ s))\ V =$
$state\text{-}val\ (transfer\ (kind\ a)\ (transfers\ (kinds\ as')\ s))\ V$
by(*auto dest*!:*DynPDG-ddep-edge-only-first-edge*)
from ‹$preds\ (kinds\ as)\ s'$›
have $preds\ (kinds\ (a\#as''))\ (transfers\ (kinds\ as')\ s')$
by(*simp add*:*kinds-def preds-split*)
with ‹$sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n'$› **have** *all'*:
$state\text{-}val\ (transfers\ (kinds\ (a\#as''))\ (transfers\ (kinds\ as')\ s'))\ V =$
$state\text{-}val\ (transfer\ (kind\ a)\ (transfers\ (kinds\ as')\ s'))\ V$
by(*auto dest*!:*DynPDG-ddep-edge-only-first-edge*)
hence $eq{:}\bigwedge s.\ transfers\ (kinds\ as)\ s =$
$transfers\ (kinds\ (a\#as''))\ (transfers\ (kinds\ as')\ s)$
by(*simp add*:*transfers-split*[*THEN sym*] *kinds-def*)

with *all* **have** *state-val* (*transfers* (*kinds as*) *s*) *V* =
state-val (*transfers* (*kinds* (*as*′@[*a*])) *s*) *V*
by(*simp add*:*transfers-split kinds-def*)
moreover
from *eq all*′ **have** *state-val* (*transfers* (*kinds as*) *s*′) *V* =
state-val (*transfers* (*kinds* (*as*′@[*a*])) *s*′) *V*
by(*simp add*:*transfers-split kinds-def*)
ultimately show *?thesis* **using** ‹*sourcenode a* −{*V*}*a*#*as*′′→$_{dd}$ *n*′› **by** *simp blast*
qed
qed

end

2.1.2 Instantiate dynamic PDG

Standard control dependence

locale *DynStandardControlDependencePDG* =
Postdomination sourcenode targetnode kind valid-edge Entry Exit +
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: ′*edge* ⇒ ′*node* **and** *targetnode* :: ′*edge* ⇒ ′*node*
and *kind* :: ′*edge* ⇒ ′*state edge-kind* **and** *valid-edge* :: ′*edge* ⇒ *bool*
and *Entry* :: ′*node* (‹′(′-*Entry*′-′)›) **and** *Def* :: ′*node* ⇒ ′*var set*
and *Use* :: ′*node* ⇒ ′*var set* **and** *state-val* :: ′*state* ⇒ ′*var* ⇒ ′*val*
and *Exit* :: ′*node* (‹′(′-*Exit*′-′)›)

begin

lemma *DynPDG-scd*:
DynPDG sourcenode targetnode kind valid-edge (-*Entry*-)
Def Use state-val (-*Exit*-) *dyn-standard-control-dependence*
proof(*unfold-locales*)
fix *n n*′ *as* **assume** *n controls*$_s$ *n*′ *via as*
show *n*′ ≠ (-*Exit*-)
proof
assume *n*′ = (-*Exit*-)
with ‹*n controls*$_s$ *n*′ *via as*› **show** *False*
by(*fastforce intro*:*Exit-not-dyn-standard-control-dependent*)
qed
next
fix *n n*′ *as* **assume** *n controls*$_s$ *n*′ *via as*
thus *n* −*as*→∗ *n*′ ∧ *as* ≠ []
by(*fastforce simp*:*dyn-standard-control-dependence-def*)
qed

end

Weak control dependence

locale *DynWeakControlDependencePDG* =
StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit +
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (‹'('-Entry'-')›) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *Exit* :: *'node* (‹'('-Exit'-')›)

begin

lemma *DynPDG-wcd*:
DynPDG sourcenode targetnode kind valid-edge (*-Entry-*)
Def Use state-val (*-Exit-*) *dyn-weak-control-dependence*
proof(*unfold-locales*)
fix *n n' as* **assume** *n weakly controls n' via as*
show *n'* ≠ (*-Exit-*)
proof
assume *n'* = (*-Exit-*)
with ‹*n weakly controls n' via as*› **show** *False*
by(*fastforce intro*:*Exit-not-dyn-weak-control-dependent*)
qed
next
fix *n n' as* **assume** *n weakly controls n' via as*
thus *n* −*as*→∗ *n'* ∧ *as* ≠ []
by(*fastforce simp*:*dyn-weak-control-dependence-def*)
qed

end

2.1.3 Data slice

definition (**in** *CFG*) *empty-control-dependence* :: *'node* ⇒ *'node* ⇒ *'edge list* ⇒ *bool*
where *empty-control-dependence n n' as* ≡ *False*

lemma (**in** *CFGExit-wf*) *DynPDG-scd*:
DynPDG sourcenode targetnode kind valid-edge (*-Entry-*)
Def Use state-val (*-Exit-*) *empty-control-dependence*
proof(*unfold-locales*)
fix *n n' as* **assume** *empty-control-dependence n n' as*
thus *n'* ≠ (*-Exit-*) **by**(*simp add*:*empty-control-dependence-def*)
next
fix *n n' as* **assume** *empty-control-dependence n n' as*
thus *n* −*as*→∗ *n'* ∧ *as* ≠ [] **by**(*simp add*:*empty-control-dependence-def*)
qed

end

2.2 Dependent Live Variables

theory *DependentLiveVariables* **imports** *DynPDG* **begin**

dependent-live-vars calculates variables which can change the value of the *Use* variables of the target node

context *DynPDG* **begin**

inductive-set
  dependent-live-vars :: $'node \Rightarrow ('var \times 'edge\ list \times 'edge\ list)\ set$
  for n' :: $'node$
  where *dep-vars-Use*:
  $V \in Use\ n' \Longrightarrow (V,[],[]) \in dependent\text{-}live\text{-}vars\ n'$

  | *dep-vars-Cons-cdep*:
  $[\![V \in Use\ (sourcenode\ a);\ sourcenode\ a\ -a\#as'\rightarrow_{cd}\ n'';\ n''\ -as''\rightarrow_{d}*\ n']\!]$
  $\Longrightarrow (V,[],a\#as'@as'') \in dependent\text{-}live\text{-}vars\ n'$

  | *dep-vars-Cons-ddep*:
  $[\![(V,as',as) \in dependent\text{-}live\text{-}vars\ n';\ V' \in Use\ (sourcenode\ a);$
    $n' = last(targetnodes\ (a\#as));$
    $sourcenode\ a\ -\{V\}a\#as'\rightarrow_{dd}\ last(targetnodes\ (a\#as'))]\!]$
  $\Longrightarrow (V',[],a\#as) \in dependent\text{-}live\text{-}vars\ n'$

  | *dep-vars-Cons-keep*:
  $[\![(V,as',as) \in dependent\text{-}live\text{-}vars\ n';\ n' = last(targetnodes\ (a\#as));$
    $\neg\ sourcenode\ a\ -\{V\}a\#as'\rightarrow_{dd}\ last(targetnodes\ (a\#as'))]\!]$
  $\Longrightarrow (V,a\#as',a\#as) \in dependent\text{-}live\text{-}vars\ n'$

lemma *dependent-live-vars-fst-prefix-snd*:
  $(V,as',as) \in dependent\text{-}live\text{-}vars\ n' \Longrightarrow \exists as''.\ as'@as'' = as$
by(*induct rule*:*dependent-live-vars.induct*,*simp-all*)

lemma *dependent-live-vars-Exit-empty* [*dest*]:
  $(V,as',as) \in dependent\text{-}live\text{-}vars\ (\text{-}Exit\text{-}) \Longrightarrow False$
proof(*induct rule*:*dependent-live-vars.induct*)
  case (*dep-vars-Cons-cdep V a as' n'' as''*)
  from ‹$n''\ -as''\rightarrow_{d}*\ (\text{-}Exit\text{-})$› **have** $n'' = (\text{-}Exit\text{-})$
    by(*fastforce intro*:*DynPDG-path-Exit*)
  with ‹$sourcenode\ a\ -a\#as'\rightarrow_{cd}\ n''$› **have** $sourcenode\ a\ -a\#as'\rightarrow_{d}*\ (\text{-}Exit\text{-})$
    by(*fastforce intro*:*DynPDG-path-cdep*)
  hence $sourcenode\ a = (\text{-}Exit\text{-})$ **by**(*fastforce intro*:*DynPDG-path-Exit*)
  with ‹$V \in Use\ (sourcenode\ a)$› **show** *False* **by** *simp*(*erule Exit-Use-empty*)
qed *auto*

lemma *dependent-live-vars-lastnode*:
$[\![(V,as',as) \in \textit{dependent-live-vars } n'; as \neq []]\!]$
$\Longrightarrow n' = \textit{last}(\textit{targetnodes as})$
proof(*induct rule*:*dependent-live-vars.induct*)
case (*dep-vars-Cons-cdep V a as' n'' as''*)
from ‹*sourcenode a* $-a\#as'\rightarrow_{cd}$ n''› **have** *sourcenode a* $-a\#as'\rightarrow*$ n''
by(*rule DynPDG-cdep-edge-CFG-path*(*1*))
from ‹n'' $-as''\rightarrow_d*$ n'› **have** n'' $-as''\rightarrow*$ n' **by**(*rule DynPDG-path-CFG-path*)
show *?case*
proof(*cases* $as'' = []$)
case *True*
with ‹n'' $-as''\rightarrow*$ n'› **have** $n'' = n'$ **by** (*auto elim*: *DynPDG.dependent-live-vars.cases*)
with ‹*sourcenode a* $-a\#as'\rightarrow*$ n''› *True*
show *?thesis* **by**(*fastforce intro*:*path-targetnode*[*THEN sym*])
next
case *False*
with ‹n'' $-as''\rightarrow*$ n'› **have** $n' = \textit{last}(\textit{targetnodes } as'')$
by(*fastforce intro*:*path-targetnode*[*THEN sym*])
with *False* **show** *?thesis* **by**(*fastforce simp*:*targetnodes-def*)
qed
qed *simp-all*

lemma *dependent-live-vars-Use-cases*:
$[\![(V,as',as) \in \textit{dependent-live-vars } n'; n -as\rightarrow* n']\!]$
$\Longrightarrow \exists\, nx\ as''.\ as = as'@as'' \wedge n -as'\rightarrow* nx \wedge nx -as''\rightarrow_d* n' \wedge V \in \textit{Use } nx \wedge$
$(\forall\, n'' \in \textit{set } (\textit{sourcenodes } as').\ V \notin \textit{Def } n'')$
proof(*induct arbitrary*:*n rule*:*dependent-live-vars.induct*)
case (*dep-vars-Use V*)
from ‹$n -[]\rightarrow* n'$› **have** *valid-node* n' **by**(*rule path-valid-node*(*2*))
hence $n' -[]\rightarrow_d* n'$ **by**(*rule DynPDG-path-Nil*)
with ‹$V \in \textit{Use } n'$› ‹$n -[]\rightarrow* n'$› **show** *?case*
by(*auto simp*:*sourcenodes-def*)
next
case (*dep-vars-Cons-cdep V a as' n'' as'' n*)
from ‹$n -a\#as'@as''\rightarrow* n'$› **have** *sourcenode a* $= n$
by(*auto elim*:*path.cases*)
from ‹*sourcenode a* $-a\#as'\rightarrow_{cd}$ n''› **have** *sourcenode a* $-a\#as'\rightarrow*$ n''
by(*rule DynPDG-cdep-edge-CFG-path*(*1*))
hence *valid-edge a* **by**(*auto elim*:*path.cases*)
hence *sourcenode a* $-[]\rightarrow*$ *sourcenode a* **by**(*fastforce intro*:*empty-path*)
from ‹*sourcenode a* $-a\#as'\rightarrow_{cd}$ n''› **have** *sourcenode a* $-a\#as'\rightarrow_d*$ n''
by(*rule DynPDG-path-cdep*)
with ‹n'' $-as''\rightarrow_d*$ n'› **have** *sourcenode a* $-(a\#as')@as''\rightarrow_d*$ n'
by(*rule DynPDG-path-Append*)
with ‹*sourcenode a* $-[]\rightarrow*$ *sourcenode a*› ‹$V \in \textit{Use}$ (*sourcenode a*)› ‹*sourcenode a* $= n$›

show *?case* **by**(*auto simp*:*sourcenodes-def*)
next
case(*dep-vars-Cons-ddep V as′ as V′ a n*)
note *ddep* = ‹*sourcenode a* −{*V*}*a#as′*$\rightarrow_{dd}$ *last* (*targetnodes* (*a#as′*))›
note *IH* = ‹⋀*n*. *n* −*as*→∗ *n′*
⟹ ∃ *nx as″*. *as* = *as′*@*as″* ∧ *n* −*as′*→∗ *nx* ∧ *nx* −*as″*$\rightarrow_{d}$∗ *n′* ∧
V ∈ *Use nx* ∧ (∀ *n″*∈*set* (*sourcenodes as′*). *V* ∉ *Def n″*)›
from ‹*n* −*a#as*→∗ *n′*› **have** *n* −[]@*a#as*→∗ *n′* **by** *simp*
hence *n* = *sourcenode a* **and** *targetnode a* −*as*→∗ *n′* **and** *valid-edge a*
by(*fastforce dest*:*path-split*)+
hence *n* −[]→∗ *n*
by(*fastforce intro*:*empty-path simp*:*valid-node-def*)
from *IH*[*OF* ‹*targetnode a* −*as*→∗ *n′*›]
have ∃ *nx as″*. *as* = *as′*@*as″* ∧ *targetnode a* −*as′*→∗ *nx* ∧ *nx* −*as″*$\rightarrow_{d}$∗ *n′* ∧
V ∈ *Use nx* ∧ (∀ *n″*∈*set* (*sourcenodes as′*). *V* ∉ *Def n″*) **.**
then obtain *nx″ as″* **where** *targetnode a* −*as′*→∗ *nx″*
and *nx″* −*as″*$\rightarrow_{d}$∗ *n′* **and** *as* = *as′*@*as″* **by** *blast*
have *last* (*targetnodes* (*a#as′*)) −*as″*$\rightarrow_{d}$∗ *n′*
proof(*cases as′*)
case *Nil*
with ‹*targetnode a* −*as′*→∗ *nx″*› **have** *nx″* = *targetnode a*
by(*auto elim*:*path.cases*)
with ‹*nx″* −*as″*$\rightarrow_{d}$∗ *n′*› *Nil* **show** *?thesis* **by**(*simp add*:*targetnodes-def*)
next
case (*Cons ax asx*)
hence *last* (*targetnodes* (*a#as′*)) = *last* (*targetnodes as′*)
by(*simp add*:*targetnodes-def*)
from *Cons* ‹*targetnode a* −*as′*→∗ *nx″*› **have** *last* (*targetnodes as′*) = *nx″*
by(*fastforce intro*:*path-targetnode*)
with ‹*last* (*targetnodes* (*a#as′*)) = *last* (*targetnodes as′*)› ‹*nx″* −*as″*$\rightarrow_{d}$∗ *n′*›
show *?thesis* **by** *simp*
qed
with *ddep* ‹*as* = *as′*@*as″*› **have** *sourcenode a* −*a#as*$\rightarrow_{d}$∗ *n′*
by(*fastforce dest*:*DynPDG-path-ddep DynPDG-path-Append*)
with ‹*V′* ∈ *Use* (*sourcenode a*)› ‹*n* = *sourcenode a*› ‹*n* −[]→∗ *n*›
show *?case* **by**(*auto simp*:*sourcenodes-def*)
next
case (*dep-vars-Cons-keep V as′ as a n*)
note *no-dep* = ‹¬ *sourcenode a* −{*V*}*a#as′*$\rightarrow_{dd}$ *last* (*targetnodes* (*a#as′*))›
note *IH* = ‹⋀*n*. *n* −*as*→∗ *n′*
⟹ ∃ *nx as″*. (*as* = *as′*@*as″*) ∧ (*n* −*as′*→∗ *nx*) ∧ (*nx* −*as″*$\rightarrow_{d}$∗ *n′*) ∧
V ∈ *Use nx* ∧ (∀ *n″*∈*set* (*sourcenodes as′*). *V* ∉ *Def n″*)›
from ‹*n* −*a#as*→∗ *n′*› **have** *n* = *sourcenode a* **and** *valid-edge a*
and *targetnode a* −*as*→∗ *n′* **by**(*auto elim*:*path-split-Cons*)
from *IH*[*OF* ‹*targetnode a* −*as*→∗ *n′*›]
have ∃ *nx as″*. *as* = *as′*@*as″* ∧ *targetnode a* −*as′*→∗ *nx* ∧ *nx* −*as″*$\rightarrow_{d}$∗ *n′* ∧
V ∈ *Use nx* ∧ (∀ *n″*∈*set* (*sourcenodes as′*). *V* ∉ *Def n″*) **.**
then obtain *nx″ as″* **where** *V* ∈ *Use nx″*
and ∀ *n″*∈*set* (*sourcenodes as′*). *V* ∉ *Def n″* **and** *targetnode a* −*as′*→∗ *nx″*

and $nx'' -as''\rightarrow_d* n'$ **and** $as = as'@as''$ **by** *blast*
from ‹*valid-edge a*› ‹*targetnode a* $-as'\rightarrow*$ nx''› **have** *sourcenode a* $-a\#as'\rightarrow*$ nx''
by(*fastforce intro*:*Cons-path*)
hence *last*(*targetnodes* $(a\#as')$) $= nx''$ **by**(*fastforce dest*:*path-targetnode*)
{ assume $V \in$ *Def* (*sourcenode a*)
with ‹$V \in$ *Use* nx''› ‹*sourcenode a* $-a\#as'\rightarrow*$ nx''›
‹$\forall n''\in$*set* (*sourcenodes as'*). $V \notin$ *Def* n''›
have (*sourcenode a*) *influences V in* nx'' *via* $a\#as'$
by(*simp add*:*dyn-data-dependence-def sourcenodes-def*)
with *no-dep* ‹*last*(*targetnodes* $(a\#as')$) $= nx''$›
‹$\forall n''\in$*set* (*sourcenodes as'*). $V \notin$ *Def* n''› ‹$V \in$ *Def* (*sourcenode a*)›
have *False* **by**(*fastforce dest*:*DynPDG-ddep-edge*) **}**
with ‹$\forall n''\in$*set* (*sourcenodes as'*). $V \notin$ *Def* n''›
have $\forall n''\in$*set* (*sourcenodes* $(a\#as')$). $V \notin$ *Def* n''
by(*fastforce simp*:*sourcenodes-def*)
with ‹$V \in$ *Use* nx''› ‹*sourcenode a* $-a\#as'\rightarrow*$ nx''› ‹$nx'' -as''\rightarrow_d* n'$›
‹$as = as'@as''$› ‹$n =$ *sourcenode a*› **show** *?case* **by** *fastforce*
qed

lemma *dependent-live-vars-dependent-edge*:
assumes $(V,as',as) \in$ *dependent-live-vars* n'
and *targetnode a* $-as\rightarrow*$ n'
and $V \in$ *Def* (*sourcenode a*) **and** *valid-edge a*
obtains *nx as''* **where** $as = as'@as''$ **and** *sourcenode a* $-\{V\}a\#as'\rightarrow_{dd}$ *nx*
and $nx -as''\rightarrow_d* n'$
proof(*atomize-elim*)
from ‹$(V,as',as) \in$ *dependent-live-vars* n'› ‹*targetnode a* $-as\rightarrow*$ n'›
have $\exists$ *nx as''*. $as = as'@as'' \wedge$ *targetnode a* $-as'\rightarrow*$ $nx \wedge nx -as''\rightarrow_d* n' \wedge$
$V \in$ *Use nx* $\wedge$ ($\forall n'' \in$ *set* (*sourcenodes as'*). $V \notin$ *Def* n'')
by(*rule dependent-live-vars-Use-cases*)
then obtain *nx as''* **where** $V \in$ *Use nx*
and $\forall n''\in$ *set*(*sourcenodes as'*). $V \notin$ *Def* n''
and *targetnode a* $-as'\rightarrow*$ *nx* **and** $nx -as''\rightarrow_d* n'$
and $as = as'@as''$ **by** *blast*
from ‹*targetnode a* $-as'\rightarrow*$ *nx*› ‹*valid-edge a*› **have** *sourcenode a* $-a\#as'\rightarrow*$ *nx*
by(*fastforce intro*:*Cons-path*)
with ‹$V \in$ *Def* (*sourcenode a*)› ‹$V \in$ *Use nx*›
‹$\forall n''\in$ *set*(*sourcenodes as'*). $V \notin$ *Def* n''›
have *sourcenode a influences V in nx via* $a\#as'$
by(*auto simp*:*dyn-data-dependence-def sourcenodes-def*)
hence *sourcenode a* $-\{V\}a\#as'\rightarrow_{dd}$ *nx* **by**(*rule DynPDG-ddep-edge*)
with ‹$nx -as''\rightarrow_d* n'$› ‹$as = as'@as''$›
show $\exists$ *as'' nx*. $(as = as'@as'') \wedge$ (*sourcenode a* $-\{V\}a\#as'\rightarrow_{dd}$ *nx*) $\wedge$
$(nx -as''\rightarrow_d* n')$ **by** *fastforce*
qed

lemma *dependent-live-vars-same-pathsI*:
  assumes $V \in Use\ n'$
  shows $[\![\forall as'\ a\ as''.\ as = as'@a\#as'' \longrightarrow \neg\ sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n';$
    $as \neq [] \longrightarrow n' = last(targetnodes\ as)]\!]$
  $\Longrightarrow (V,as,as) \in dependent\text{-}live\text{-}vars\ n'$
proof(*induct as*)
  case *Nil*
  from ‹$V \in Use\ n'$› **show** *?case* **by**(*rule dep-vars-Use*)
next
  case (*Cons ax asx*)
  note *lastnode* = ‹$ax\#asx \neq [] \longrightarrow n' = last\ (targetnodes\ (ax\#asx))$›
  note *IH* = ‹$[\![\forall as'\ a\ as''.\ asx = as'@a\#as'' \longrightarrow$
    $\neg\ sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n';$
    $asx \neq [] \longrightarrow n' = last\ (targetnodes\ asx)]\!]$
    $\Longrightarrow (V,\ asx,\ asx) \in dependent\text{-}live\text{-}vars\ n'$›
  from ‹$\forall as'\ a\ as''.\ ax\#asx = as'@a\#as'' \longrightarrow \neg\ sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}$
n'›
  have *all'*:$\forall as'\ a\ as''.\ asx = as'@a\#as'' \longrightarrow \neg\ sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n'$
    and $\neg\ sourcenode\ ax\ -\{V\}ax\#asx\rightarrow_{dd}\ n'$
    by *simp-all*
  show *?case*
  proof(*cases* $asx = []$)
    case *True*
    from ‹$V \in Use\ n'$› **have** $(V,[],[]) \in dependent\text{-}live\text{-}vars\ n'$ **by**(*rule dep-vars-Use*)
    with ‹$\neg\ sourcenode\ ax\ -\{V\}ax\#asx\rightarrow_{dd}\ n'$› *True lastnode*
    have $(V,[ax],[ax]) \in dependent\text{-}live\text{-}vars\ n'$
      by(*fastforce intro*:*dep-vars-Cons-keep*)
    with *True* **show** *?thesis* **by** *simp*
  next
    case *False*
    with *lastnode* **have** $asx \neq [] \longrightarrow n' = last\ (targetnodes\ asx)$
      by(*simp add*:*targetnodes-def*)
    from *IH*[*OF all' this*] **have** $(V,\ asx,\ asx) \in dependent\text{-}live\text{-}vars\ n'$ **.**
    with ‹$\neg\ sourcenode\ ax\ -\{V\}ax\#asx\rightarrow_{dd}\ n'$› *lastnode*
    show *?thesis* **by**(*fastforce intro*:*dep-vars-Cons-keep*)
  qed
qed

lemma *dependent-live-vars-same-pathsD*:
  $[\![(V,as,as) \in dependent\text{-}live\text{-}vars\ n';\ \ as \neq [] \longrightarrow n' = last(targetnodes\ as)]\!]$
  $\Longrightarrow V \in Use\ n' \wedge (\forall as'\ a\ as''.\ as = as'@a\#as'' \longrightarrow$
    $\neg\ sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n')$
proof(*induct as*)
  case *Nil*
  have $(V,[],[]) \in dependent\text{-}live\text{-}vars\ n'$ **by** *fact*
  thus *?case*
  by(*fastforce elim*:*dependent-live-vars.cases simp*:*targetnodes-def sourcenodes-def*)

next
  case (*Cons ax asx*)
  note *IH* = ‹⟦(*V*,*asx*,*asx*) ∈ *dependent-live-vars n'*;
      asx ≠ [] ⟶ *n'* = *last* (*targetnodes asx*)⟧
    ⟹ *V* ∈ *Use n'* ∧ (∀ *as' a as''*. *asx* = *as'*@*a*#*as''* ⟶
              ¬ *sourcenode a* −{*V*}*a*#*as''*→$_{dd}$ *n'*)›
  from ‹(*V*,*ax*#*asx*,*ax*#*asx*) ∈ *dependent-live-vars n'*›
  have (*V*,*asx*,*asx*) ∈ *dependent-live-vars n'*
    and ¬ *sourcenode ax* −{*V*}*ax*#*asx*→$_{dd}$ *last*(*targetnodes* (*ax*#*asx*))
    by(*auto elim*:*dependent-live-vars.cases*)
  from ‹*ax*#*asx* ≠ [] ⟶ *n'* = *last* (*targetnodes* (*ax*#*asx*))›
  have *n'* = *last* (*targetnodes* (*ax*#*asx*)) **by** *simp*
  show *?case*
  proof(*cases asx* = [])
    case *True*
    with ‹(*V*,*asx*,*asx*) ∈ *dependent-live-vars n'*› **have** *V* ∈ *Use n'*
      by(*fastforce elim*:*dependent-live-vars.cases*)
    from ‹¬ *sourcenode ax* −{*V*}*ax*#*asx*→$_{dd}$ *last*(*targetnodes* (*ax*#*asx*))›
      True ‹*n'* = *last* (*targetnodes* (*ax*#*asx*))›
    have ∀ *as' a as''*. *ax*#*asx* = *as'*@*a*#*as''* ⟶ ¬ *sourcenode a* −{*V*}*a*#*as''*→$_{dd}$
n'
      by *auto*(*case-tac as'*,*auto*)
    with ‹*V* ∈ *Use n'*› **show** *?thesis* **by** *simp*
  next
    case *False*
    with ‹*n'* = *last* (*targetnodes* (*ax*#*asx*))›
    have *asx* ≠ [] ⟶ *n'* = *last* (*targetnodes asx*)
      by(*simp add*:*targetnodes-def*)
    from *IH*[*OF* ‹(*V*,*asx*,*asx*) ∈ *dependent-live-vars n'*› *this*]
    have *V* ∈ *Use n'* ∧ (∀ *as' a as''*. *asx* = *as'*@*a*#*as''* ⟶
              ¬ *sourcenode a* −{*V*}*a*#*as''*→$_{dd}$ *n'*) **.**
    with ‹¬ *sourcenode ax* −{*V*}*ax*#*asx*→$_{dd}$ *last*(*targetnodes* (*ax*#*asx*))›
      ‹*n'* = *last* (*targetnodes* (*ax*#*asx*))› **have** *V* ∈ *Use n'*
      and ∀ *as' a as''*. *ax*#*asx* = *as'*@*a*#*as''* ⟶
              ¬ *sourcenode a* −{*V*}*a*#*as''*→$_{dd}$ *n'*
      by *auto*(*case-tac as'*,*auto*)
    thus *?thesis* **by** *simp*
  qed
qed

lemma *dependent-live-vars-same-paths*:
  as ≠ [] ⟶ *n'* = *last*(*targetnodes as*) ⟹
  (*V*,*as*,*as*) ∈ *dependent-live-vars n'* =
  (*V* ∈ *Use n'* ∧ (∀ *as' a as''*. *as* = *as'*@*a*#*as''* ⟶
              ¬ *sourcenode a* −{*V*}*a*#*as''*→$_{dd}$ *n'*))
by(*fastforce intro*!:*dependent-live-vars-same-pathsD dependent-live-vars-same-pathsI*)

lemma *dependent-live-vars-cdep-empty-fst*:
assumes $n'' -as\rightarrow_{cd} n'$ **and** $V' \in$ *Use* n''
shows $(V',[],as) \in$ *dependent-live-vars* n'
proof(*cases as*)
case *Nil*
with ‹$n'' -as\rightarrow_{cd} n'$› **show** *?thesis*
by(*fastforce elim*:*DynPDG-edge.cases dest*:*dyn-control-dependence-path*)
next
case (*Cons ax asx*)
with ‹$n'' -as\rightarrow_{cd} n'$› **have** *sourcenode ax* $= n''$
by(*auto dest*:*DynPDG-cdep-edge-CFG-path elim*:*path.cases*)
from ‹$n'' -as\rightarrow_{cd} n'$› **have** *valid-node* n'
by(*fastforce intro*:*path-valid-node*(*2*) *DynPDG-cdep-edge-CFG-path*(*1*))
from *Cons* ‹$n'' -as\rightarrow_{cd} n'$› **have** *last*(*targetnodes as*) $= n'$
by(*fastforce intro*:*path-targetnode dest*:*DynPDG-cdep-edge-CFG-path*)
with *Cons* ‹$n'' -as\rightarrow_{cd} n'$› ‹$V' \in$ *Use* n''› ‹*sourcenode ax* $= n''$› ‹*valid-node* n'›
have $(V', [], ax\#asx@[]) \in$ *dependent-live-vars* n'
by(*fastforce intro*:*dep-vars-Cons-cdep DynPDG-path-Nil*)
with *Cons* **show** *?thesis* **by** *simp*
qed

lemma *dependent-live-vars-ddep-empty-fst*:
assumes $n'' -\{V\}as\rightarrow_{dd} n'$ **and** $V' \in$ *Use* n''
shows $(V',[],as) \in$ *dependent-live-vars* n'
proof(*cases as*)
case *Nil*
with ‹$n'' -\{V\}as\rightarrow_{dd} n'$› **show** *?thesis*
by(*fastforce elim*:*DynPDG-edge.cases simp*:*dyn-data-dependence-def*)
next
case (*Cons ax asx*)
with ‹$n'' -\{V\}as\rightarrow_{dd} n'$› **have** *sourcenode ax* $= n''$
by(*auto dest*:*DynPDG-ddep-edge-CFG-path elim*:*path.cases*)
from *Cons* ‹$n'' -\{V\}as\rightarrow_{dd} n'$› **have** *last*(*targetnodes as*) $= n'$
by(*fastforce intro*:*path-targetnode elim*:*DynPDG-ddep-edge-CFG-path*(*1*))
from *Cons* ‹$n'' -\{V\}as\rightarrow_{dd} n'$› **have** *all*:$\forall as'\ a\ as''.\ asx = as'@a\#as'' \longrightarrow$
$\neg$ *sourcenode a* $-\{V\}a\#as''\rightarrow_{dd} n'$
by(*fastforce dest*:*DynPDG-ddep-edge-no-shorter-ddep-edge*)
from ‹$n'' -\{V\}as\rightarrow_{dd} n'$› **have** $V \in$ *Use* n'
by(*auto elim*!:*DynPDG-edge.cases simp*:*dyn-data-dependence-def*)
from *Cons* ‹$n'' -\{V\}as\rightarrow_{dd} n'$› **have** $as \neq [] \longrightarrow n' =$ *last*(*targetnodes as*)
by(*fastforce dest*:*DynPDG-ddep-edge-CFG-path path-targetnode*)
with *Cons* **have** $asx \neq [] \longrightarrow n' =$ *last*(*targetnodes asx*)
by(*fastforce simp*:*targetnodes-def*)
with *all* ‹$V \in$ *Use* n'› **have** $(V,asx,asx) \in$ *dependent-live-vars* n'
by −(*rule dependent-live-vars-same-pathsI*)
with ‹$V' \in$ *Use* n''› ‹$n'' -\{V\}as\rightarrow_{dd} n'$› ‹*last*(*targetnodes as*) $= n'$›
Cons ‹*sourcenode ax* $= n''$› **show** *?thesis*
by(*fastforce intro*:*dep-vars-Cons-ddep*)

qed

lemma *ddep-dependent-live-vars-keep-notempty*:
assumes $n -\{V\}a\#as\rightarrow_{dd}\ n''$ **and** $as' \neq []$
and $(V,as'',as') \in$ *dependent-live-vars* n'
shows $(V,as@as'',as@as') \in$ *dependent-live-vars* n'
proof −
from ‹$n -\{V\}a\#as\rightarrow_{dd}\ n''$› **have** $\forall n'' \in$ *set* (*sourcenodes as*). $V \notin$ *Def* n''
by(*auto elim*:*DynPDG-edge.cases simp*:*dyn-data-dependence-def*)
with ‹$(V,as'',as') \in$ *dependent-live-vars* n'› **show** *?thesis*
proof(*induct as*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons ax asx*)
note *IH* = ‹⟦$(V,as'',as') \in$ *dependent-live-vars* n';
$\forall n''\in$*set* (*sourcenodes asx*). $V \notin$ *Def* n''⟧
$\Longrightarrow (V, asx@as'',asx@as') \in$ *dependent-live-vars* n'›
from ‹$\forall n''\in$*set* (*sourcenodes* (*ax#asx*)). $V \notin$ *Def* n''›
have $\forall n''\in$*set* (*sourcenodes asx*). $V \notin$ *Def* n''
by(*auto simp*:*sourcenodes-def*)
from *IH*[*OF* ‹$(V,as'',as') \in$ *dependent-live-vars* n'› *this*]
have $(V,asx@as'',asx@as') \in$ *dependent-live-vars* n' **.**
from ‹$as' \neq []$› ‹$(V,as'',as') \in$ *dependent-live-vars* n'›
have $n' =$ *last*(*targetnodes* as')
by(*fastforce intro*:*dependent-live-vars-lastnode*)
with ‹$as' \neq []$› **have** $n' =$ *last*(*targetnodes* (*ax#asx@as'*))
by(*fastforce simp*:*targetnodes-def*)
have ¬ *sourcenode ax* $-\{V\}ax\#asx@as''\rightarrow_{dd}$ *last*(*targetnodes* (*ax#asx@as''*))
proof
assume *sourcenode ax* $-\{V\}ax\#asx@as''\rightarrow_{dd}$ *last*(*targetnodes* (*ax#asx@as''*))
hence *sourcenode ax* $-\{V\}ax\#asx@as''\rightarrow_{dd}$ *last*(*targetnodes* (*ax#asx@as''*))
by *simp*
with ‹$\forall n''\in$*set* (*sourcenodes* (*ax#asx*)). $V \notin$ *Def* n''›
show *False*
by(*fastforce elim*:*DynPDG-edge.cases*
simp:*dyn-data-dependence-def sourcenodes-def*)
qed
with ‹$(V,asx@as'',asx@as') \in$ *dependent-live-vars* n'›
‹$n' =$ *last*(*targetnodes* (*ax#asx@as'*))›
show *?case* **by**(*fastforce intro*:*dep-vars-Cons-keep*)
qed
qed

lemma *dependent-live-vars-cdep-dependent-live-vars*:

assumes $n'' -as''\rightarrow_{cd} n'$ **and** (*V'*,*as'*,*as*) ∈ *dependent-live-vars n''*
shows (*V'*,*as'*,*as*@*as''*) ∈ *dependent-live-vars n'*
proof −
from ‹$n'' -as''\rightarrow_{cd} n'$› **have** *as''* ≠ []
by(*fastforce elim*:*DynPDG-edge.cases dest*:*dyn-control-dependence-path*)
with ‹$n'' -as''\rightarrow_{cd} n'$› **have** *last*(*targetnodes as''*) = *n'*
by(*fastforce intro*:*path-targetnode elim*:*DynPDG-cdep-edge-CFG-path*(*1*))
from ‹(*V'*,*as'*,*as*) ∈ *dependent-live-vars n''*› **show** *?thesis*
proof(*induct rule*:*dependent-live-vars.induct*)
case (*dep-vars-Use V'*)
from ‹*V'* ∈ *Use n''*› ‹$n'' -as''\rightarrow_{cd} n'$› ‹*last*(*targetnodes as''*) = *n'*› **show** *?case*
by(*fastforce intro*:*dependent-live-vars-cdep-empty-fst simp*:*targetnodes-def*)
next
case (*dep-vars-Cons-cdep V a as' nx asx*)
from ‹$n'' -as''\rightarrow_{cd} n'$› **have** $n'' -as''\rightarrow_{d}* n'$ **by**(*rule DynPDG-path-cdep*)
with ‹$nx -asx\rightarrow_{d}* n''$› **have** $nx -asx@as''\rightarrow_{d}* n'$
by −(*rule DynPDG-path-Append*)
with ‹*V* ∈ *Use* (*sourcenode a*)› ‹(*sourcenode a*) $-a\#as'\rightarrow_{cd}$ *nx*›
show *?case* **by**(*fastforce intro*:*dependent-live-vars.dep-vars-Cons-cdep*)
next
case (*dep-vars-Cons-ddep V as' as V' a*)
from ‹*as''* ≠ []› ‹*last*(*targetnodes as''*) = *n'*›
have *n'* = *last*(*targetnodes* ((*a*#*as*)@*as''*))
by(*simp add*:*targetnodes-def*)
with *dep-vars-Cons-ddep*
show *?case* **by**(*fastforce intro*:*dependent-live-vars.dep-vars-Cons-ddep*)
next
case (*dep-vars-Cons-keep V as' as a*)
from ‹*as''* ≠ []› ‹*last*(*targetnodes as''*) = *n'*›
have *n'* = *last*(*targetnodes* ((*a*#*as*)@*as''*))
by(*simp add*:*targetnodes-def*)
with *dep-vars-Cons-keep*
show *?case* **by**(*fastforce intro*:*dependent-live-vars.dep-vars-Cons-keep*)
qed
qed

lemma *dependent-live-vars-ddep-dependent-live-vars*:
assumes $n'' -\{V\}as''\rightarrow_{dd} n'$ **and** (*V'*,*as'*,*as*) ∈ *dependent-live-vars n''*
shows (*V'*,*as'*,*as*@*as''*) ∈ *dependent-live-vars n'*
proof −
from ‹$n'' -\{V\}as''\rightarrow_{dd} n'$› **have** *as''* ≠ []
by(*rule DynPDG-ddep-edge-CFG-path*(*2*))
with ‹$n'' -\{V\}as''\rightarrow_{dd} n'$› **have** *last*(*targetnodes as''*) = *n'*
by(*fastforce intro*:*path-targetnode elim*:*DynPDG-ddep-edge-CFG-path*(*1*))
from ‹$n'' -\{V\}as''\rightarrow_{dd} n'$› **have** *notExit*:*n'* ≠ (*-Exit-*)
by(*fastforce elim*:*DynPDG-edge.cases simp*:*dyn-data-dependence-def*)
from ‹(*V'*,*as'*,*as*) ∈ *dependent-live-vars n''*› **show** *?thesis*
proof(*induct rule*:*dependent-live-vars.induct*)

case (*dep-vars-Use V'*)
from ‹$V' \in$ *Use* n''› ‹$n'' -\{V\}as''\rightarrow_{dd}$ n'› ‹*last*(*targetnodes* as'') = n'› **show** *?case*
by(*fastforce intro*:*dependent-live-vars-ddep-empty-fst simp*:*targetnodes-def*)
next
case (*dep-vars-Cons-cdep V' a as' nx asx*)
from ‹$n'' -\{V\}as''\rightarrow_{dd}$ n'› **have** $n'' -as''\rightarrow_{d*}$ n' **by**(*rule DynPDG-path-ddep*)
with ‹$nx -asx\rightarrow_{d*}$ n''› **have** $nx -asx@as''\rightarrow_{d*}$ n'
by −(*rule DynPDG-path-Append*)
with ‹$V' \in$ *Use* (*sourcenode a*)› ‹*sourcenode a* $-a\#as'\rightarrow_{cd}$ *nx*› *notExit*
show *?case* **by**(*fastforce intro*:*dependent-live-vars.dep-vars-Cons-cdep*)
next
case (*dep-vars-Cons-ddep V as' as V' a*)
from ‹$as'' \neq []$› ‹*last*(*targetnodes* as'') = n'›
have n' = *last*(*targetnodes* ((*a*#*as*)@as''))
by(*simp add*:*targetnodes-def*)
with *dep-vars-Cons-ddep*
show *?case* **by**(*fastforce intro*:*dependent-live-vars.dep-vars-Cons-ddep*)
next
case (*dep-vars-Cons-keep V as' as a*)
from ‹$as'' \neq []$› ‹*last*(*targetnodes* as'') = n'›
have n' = *last*(*targetnodes* ((*a*#*as*)@as''))
by(*simp add*:*targetnodes-def*)
with *dep-vars-Cons-keep*
show *?case* **by**(*fastforce intro*:*dependent-live-vars.dep-vars-Cons-keep*)
qed
qed

lemma *dependent-live-vars-dep-dependent-live-vars*:
$[\![n'' -as''\rightarrow_{d*} n'; (V',as',as) \in \textit{dependent-live-vars } n'']\!]$
$\Longrightarrow (V',as',as@as'') \in \textit{dependent-live-vars } n'$
proof(*induct rule*:*DynPDG-path.induct*)
case (*DynPDG-path-Nil n*) **thus** *?case* **by** *simp*
next
case (*DynPDG-path-Append-cdep n asx n'' asx' n'*)
note *IH* = ‹(V', as', *as*) ∈ *dependent-live-vars n* ⟹
(V', as', *as* @ *asx*) ∈ *dependent-live-vars* n''›
from *IH*[*OF* ‹(V',as',*as*) ∈ *dependent-live-vars n*›]
have (V',as',*as*@*asx*) ∈ *dependent-live-vars* n'' **.**
with ‹$n'' -asx'\rightarrow_{cd}$ n'› **have** (V',as',(*as*@*asx*)@asx') ∈ *dependent-live-vars* n'
by(*rule dependent-live-vars-cdep-dependent-live-vars*)
thus *?case* **by** *simp*
next
case (*DynPDG-path-Append-ddep n asx n'' V asx' n'*)
note *IH* = ‹(V', as', *as*) ∈ *dependent-live-vars n* ⟹
(V', as', *as* @ *asx*) ∈ *dependent-live-vars* n''›
from *IH*[*OF* ‹(V',as',*as*) ∈ *dependent-live-vars n*›]
have (V',as',*as*@*asx*) ∈ *dependent-live-vars* n'' **.**

with ‹$n'' -\{V\}asx' \rightarrow_{dd} n'$› **have** $(V',as',(as@asx)@asx') \in$ *dependent-live-vars* n'
by(*rule dependent-live-vars-ddep-dependent-live-vars*)
thus *?case* **by** *simp*
qed

end

end

2.3 Formalization of Bit Vectors

theory *BitVector* **imports** *Main* **begin**

type-synonym *bit-vector* = *bool list*

fun *bv-leqs* :: *bit-vector* ⇒ *bit-vector* ⇒ *bool* (‹- $\preceq_b$ -› *99*)
where *bv-Nils*:[] $\preceq_b$ [] = *True*
| *bv-Cons*:(*x*#*xs*) $\preceq_b$ (*y*#*ys*) = ((*x* ⟶ *y*) ∧ *xs* $\preceq_b$ *ys*)
| *bv-rest*:*xs* $\preceq_b$ *ys* = *False*

2.3.1 Some basic properties

lemma *bv-length*: *xs* $\preceq_b$ *ys* ⟹ *length xs* = *length ys*
by(*induct rule*:*bv-leqs.induct*)*auto*

lemma [*dest*!]: *xs* $\preceq_b$ [] ⟹ *xs* = []
by(*induct xs*) *auto*

lemma *bv-leqs-AppendI*:
⟦*xs* $\preceq_b$ *ys*; *xs'* $\preceq_b$ *ys'*⟧ ⟹ (*xs*@*xs'*) $\preceq_b$ (*ys*@*ys'*)
by(*induct xs ys rule*:*bv-leqs.induct*,*auto*)

lemma *bv-leqs-AppendD*:
⟦(*xs*@*xs'*) $\preceq_b$ (*ys*@*ys'*); *length xs* = *length ys*⟧
⟹ *xs* $\preceq_b$ *ys* ∧ *xs'* $\preceq_b$ *ys'*
by(*induct xs ys rule*:*bv-leqs.induct*,*auto*)

lemma *bv-leqs-eq*:
xs $\preceq_b$ *ys* = ((∀ *i* < *length xs*. *xs* ! *i* ⟶ *ys* ! *i*) ∧ *length xs* = *length ys*)
proof(*induct xs ys rule*:*bv-leqs.induct*)
case (*2 x xs y ys*)
note *eq* = ‹*xs* $\preceq_b$ *ys* =
((∀ *i* < *length xs*. *xs* ! *i* ⟶ *ys* ! *i*) ∧ *length xs* = *length ys*)›

show *?case*
proof
assume *leqs*:$x\#xs \preceq_b y\#ys$
with *eq* **have** $x \longrightarrow y$ **and** $\forall i < length\ xs.\ xs\ !\ i \longrightarrow ys\ !\ i$
and $length\ xs = length\ ys$ **by** *simp-all*
from ‹$x \longrightarrow y$› **have** $(x\#xs)\ !\ 0 \longrightarrow (y\#ys)\ !\ 0$ **by** *simp*
{ fix i **assume** $i > 0$ **and** $i < length\ (x\#xs)$
then obtain j **where** $i = Suc\ j$ **and** $j < length\ xs$ **by**(*cases i*) *auto*
with ‹$\forall i < length\ xs.\ xs\ !\ i \longrightarrow ys\ !\ i$›
have $(x\#xs)\ !\ i \longrightarrow (y\#ys)\ !\ i$ **by** *auto* **}**
hence $\forall i < length\ (x\#xs).\ i > 0 \longrightarrow (x\#xs)\ !\ i \longrightarrow (y\#ys)\ !\ i$ **by** *simp*
with ‹$(x\#xs)\ !\ 0 \longrightarrow (y\#ys)\ !\ 0$› ‹$length\ xs = length\ ys$›
show $(\forall i < length\ (x\#xs).\ (x\#xs)\ !\ i \longrightarrow (y\#ys)\ !\ i) \wedge$
$length\ (x\#xs) = length\ (y\#ys)$
by *clarsimp*(*case-tac i>0*,*auto*)
next
assume $(\forall i < length\ (x\#xs).\ (x\#xs)\ !\ i \longrightarrow (y\#ys)\ !\ i) \wedge$
$length\ (x\#xs) = length\ (y\#ys)$
hence $\forall i < length\ (x\#xs).\ (x\#xs)\ !\ i \longrightarrow (y\#ys)\ !\ i$
and $length\ (x\#xs) = length\ (y\#ys)$ **by** *simp-all*
from ‹$\forall i < length\ (x\#xs).\ (x\#xs)\ !\ i \longrightarrow (y\#ys)\ !\ i$›
have $\forall i < length\ xs.\ xs\ !\ i \longrightarrow ys\ !\ i$
by *clarsimp*(*erule-tac x=Suc i* **in** *allE*,*auto*)
with *eq* ‹$length\ (x\#xs) = length\ (y\#ys)$› **have** $xs \preceq_b ys$ **by** *simp*
from ‹$\forall i < length\ (x\#xs).\ (x\#xs)\ !\ i \longrightarrow (y\#ys)\ !\ i$›
have $x \longrightarrow y$ **by**(*erule-tac x=0* **in** *allE*) *simp*
with ‹$xs \preceq_b ys$› **show** $x\#xs \preceq_b y\#ys$ **by** *simp*
qed
qed *simp-all*

2.3.2 $\preceq_b$ is an order on bit vectors with minimal and maximal element

lemma *minimal-element*:
replicate (*length xs*) *False* $\preceq_b$ *xs*
by(*induct xs*) *auto*

lemma *maximal-element*:
xs $\preceq_b$ *replicate* (*length xs*) *True*
by(*induct xs*) *auto*

lemma *bv-leqs-refl*:*xs* $\preceq_b$ *xs*
by(*induct xs*) *auto*

lemma *bv-leqs-trans*:$[\![xs \preceq_b ys;\ ys \preceq_b zs]\!] \Longrightarrow xs \preceq_b zs$
proof(*induct xs ys arbitrary*:*zs rule*:*bv-leqs.induct*)
case (*2 x xs y ys*)
note *IH* = ‹$\bigwedge zs.\ [\![xs \preceq_b ys;\ ys \preceq_b zs]\!] \Longrightarrow xs \preceq_b zs$›

from ‹$(x\#xs) \preceq_b (y\#ys)$› **have** $xs \preceq_b ys$ **and** $x \longrightarrow y$ **by** *simp-all*
from ‹$(y\#ys) \preceq_b zs$› **obtain** $z\ zs'$ **where** $zs = z\#zs'$ **by**(*cases zs*) *auto*
with ‹$(y\#ys) \preceq_b zs$› **have** $ys \preceq_b zs'$ **and** $y \longrightarrow z$ **by** *simp-all*
from *IH*[*OF* ‹$xs \preceq_b ys$› ‹$ys \preceq_b zs'$›] **have** $xs \preceq_b zs'$ **.**
with ‹$x \longrightarrow y$› ‹$y \longrightarrow z$› ‹$zs = z\#zs'$› **show** *?case* **by** *simp*
qed *simp-all*

lemma *bv-leqs-antisym*:⟦$xs \preceq_b ys$; $ys \preceq_b xs$⟧ $\Longrightarrow xs = ys$
by(*induct xs ys rule*:*bv-leqs.induct*)*auto*

definition *bv-less* :: *bit-vector* ⇒ *bit-vector* ⇒ *bool* (‹- $\prec_b$ -› *99*)
where $xs \prec_b ys \equiv xs \preceq_b ys \wedge xs \neq ys$

interpretation *order bv-leqs bv-less*
by(*unfold-locales*,
auto intro:*bv-leqs-refl bv-leqs-trans bv-leqs-antisym simp*:*bv-less-def*)

end

2.4 Dynamic Backward Slice

theory *DynSlice* **imports** *DependentLiveVariables BitVector ../Basic/SemanticsCFG*
begin

2.4.1 Backward slice of paths

context *DynPDG* **begin**

fun *slice-path* :: *'edge list* ⇒ *bit-vector*
where *slice-path* [] = []
| *slice-path* (*a*#*as*) = (*let* n' = *last*(*targetnodes* (*a*#*as*)) *in*
(*sourcenode a* $-a\#as\rightarrow_d*$ n')#*slice-path as*)

lemma *slice-path-length*:
length(*slice-path as*) = *length as*
by(*induct as*) *auto*

lemma *slice-path-right-Cons*:
assumes *slice*:*slice-path as* = *x*#*xs*
obtains a' as' **where** *as* = $a'\#as'$ **and** *slice-path* as' = *xs*
proof(*atomize-elim*)
from *slice* **show** ∃ a' as'. *as* = $a'\#as'$ ∧ *slice-path* as' = *xs*
by(*induct as*) *auto*
qed

2.4.2 The proof of the fundamental property of (dynamic) slicing

fun *select-edge-kinds* :: *'edge list* ⇒ *bit-vector* ⇒ *'state edge-kind list*
where *select-edge-kinds* [] [] = []
| *select-edge-kinds* (*a#as*) (*b#bs*) = (*if b then kind a*
else (*case kind a of* ⇑*f* ⇒ ⇑*id* | $(Q)_{\surd}$ ⇒ $(\lambda s.\ True)_{\surd}$))#*select-edge-kinds as bs*

definition *slice-kinds* :: *'edge list* ⇒ *'state edge-kind list*
where *slice-kinds as* = *select-edge-kinds as* (*slice-path as*)

lemma *select-edge-kinds-max-bv*:
select-edge-kinds as (*replicate* (*length as*) *True*) = *kinds as*
by(*induct as*,*auto simp*:*kinds-def*)

lemma *slice-path-leqs-information-same-Uses*:
⟦*n* −*as*→* *n'*; *bs* $\preceq_b$ *bs'*; *slice-path as* = *bs*;
select-edge-kinds as bs = *es*; *select-edge-kinds as bs'* = *es'*;
∀ *V xs*. (*V*,*xs*,*as*) ∈ *dependent-live-vars n'* ⟶ *state-val s V* = *state-val s' V*;
preds es' s'⟧
⟹ (∀ *V* ∈ *Use n'*. *state-val* (*transfers es s*) *V* =
state-val (*transfers es' s'*) *V*) ∧ *preds es s*
proof(*induct bs bs' arbitrary*:*as es es' n s s' rule*:*bv-leqs.induct*)
case *1*
from ‹*slice-path as* = []› **have** *as* = [] **by**(*cases as*) *auto*
with ‹*select-edge-kinds as* [] = *es*› ‹*select-edge-kinds as* [] = *es'*›
have *es* = [] **and** *es'* = [] **by** *simp-all*
{ fix *V* **assume** *V* ∈ *Use n'*
hence (*V*,[],[]) ∈ *dependent-live-vars n'* **by**(*rule dep-vars-Use*)
with ‹∀ *V xs*. (*V*,*xs*,*as*) ∈ *dependent-live-vars n'* ⟶
state-val s V = *state-val s' V*› ‹*V* ∈ *Use n'*› ‹*as* = []›
have *state-val s V* = *state-val s' V* **by** *blast* **}**
with ‹*es* = []› ‹*es'* = []› **show** *?case* **by** *simp*
next
case (*2 x xs y ys*)
note *all* = ‹∀ *V xs*. (*V*,*xs*,*as*) ∈ *dependent-live-vars n'* ⟶
state-val s V = *state-val s' V*›
note *IH* = ‹⋀*as es es' n s s'*. ⟦*n* −*as*→* *n'*; *xs* $\preceq_b$ *ys*; *slice-path as* = *xs*;
select-edge-kinds as xs = *es*; *select-edge-kinds as ys* = *es'*;
∀ *V xs*. (*V*,*xs*,*as*) ∈ *dependent-live-vars n'* ⟶
state-val s V = *state-val s' V*;
preds es' s'⟧
⟹ (∀ *V* ∈ *Use n'*. *state-val* (*transfers es s*) *V* =
state-val (*transfers es' s'*) *V*) ∧ *preds es s*›
from ‹*x#xs* $\preceq_b$ *y#ys*› **have** *x* ⟶ *y* **and** *xs* $\preceq_b$ *ys* **by** *simp-all*
from ‹*slice-path as* = *x#xs*› **obtain** *a' as'* **where** *as* = *a'#as'*

and *slice-path as'* = *xs* **by**(*erule slice-path-right-Cons*)
from ‹*as* = *a'#as'*› ‹*select-edge-kinds as* (*x#xs*) = *es*›
obtain *ex esx* **where** *es* = *ex#esx*
and *ex*:*ex* = (*if x then kind a'*
else (*case kind a' of* ⇑*f* ⇒ ⇑*id* | $(Q)_{\surd}$ ⇒ $(\lambda s.\ True)_{\surd}$))
and *select-edge-kinds as' xs* = *esx* **by** *auto*
from ‹*as* = *a'#as'*› ‹*select-edge-kinds as* (*y#ys*) = *es'*› **obtain** *ex' esx'*
where *es'* = *ex'#esx'*
and *ex'*:*ex'* = (*if y then kind a'*
else (*case kind a' of* ⇑*f* ⇒ ⇑*id* | $(Q)_{\surd}$ ⇒ $(\lambda s.\ True)_{\surd}$))
and *select-edge-kinds as' ys* = *esx'* **by** *auto*
from ‹*n* −*as*→∗ *n'*› ‹*as* = *a'#as'*› **have** [*simp*]:*n* = *sourcenode a'*
and *valid-edge a'* **and** *targetnode a'* −*as'*→∗ *n'*
by(*auto elim*:*path-split-Cons*)
from ‹*n* −*as*→∗ *n'*› ‹*as* = *a'#as'*› **have** *last*(*targetnodes as*) = *n'*
by(*fastforce intro*:*path-targetnode*)
from ‹*preds es' s'*› ‹*es'* = *ex'#esx'*› **have** *pred ex' s'*
and *preds esx'* (*transfer ex' s'*) **by** *simp-all*
show *?case*
proof(*cases as'* = [])
case *True*
hence [*simp*]:*as'* = [] **by** *simp*
with ‹*slice-path as'* = *xs*› ‹$xs \preceq_b ys$›
have [*simp*]:*xs* = [] ∧ *ys* = [] **by** *auto*(*cases ys*,*auto*)+
with ‹*select-edge-kinds as' xs* = *esx*› ‹*select-edge-kinds as' ys* = *esx'*›
have [*simp*]:*esx* = [] **and** [*simp*]:*esx'* = [] **by** *simp-all*
from *True* ‹*targetnode a'* −*as'*→∗ *n'*›
have [*simp*]:*n'* = *targetnode a'* **by**(*auto elim*:*path.cases*)
show *?thesis*
proof(*cases x*)
case *True*
with ‹*x* ⟶ *y*› *ex ex'* **have** [*simp*]:*ex* = *kind a'* ∧ *ex'* = *kind a'* **by** *simp*
have *pred ex s*
proof(*cases ex*)
case (*Predicate Q*)
with *ex ex' True* ‹*x* ⟶ *y*› **have** [*simp*]:*transfer ex s* = *s*
and [*simp*]:*transfer ex' s'* = *s'*
by(*cases kind a'*,*auto*)+
show *?thesis*
proof(*cases* $n\ -[a']\rightarrow_{cd}\ n'$)
case *True*
{ **fix** *V'* **assume** *V'* ∈ *Use n*
with *True* ‹*valid-edge a'*›
have (*V'*,[],*a'#*[]@[]) ∈ *dependent-live-vars n'*
by(*fastforce intro*:*dep-vars-Cons-cdep DynPDG-path-Nil*
simp:*targetnodes-def*)
with *all* ‹*as* = *a'#as'*› **have** *state-val s V'* = *state-val s' V'*
by *fastforce* }
with ‹*pred ex' s'*› ‹*valid-edge a'*›

show *?thesis* **by**(*fastforce elim*:*CFG-edge-Uses-pred-equal*)
next
case *False*
from *ex True Predicate* **have** *kind* $a' = (Q)_{\surd}$ **by**(*auto split*:*if-split-asm*)
from *True* ‹*slice-path as* = *x#xs*› ‹*as* = *a'#as'*› **have** n $-[a']\rightarrow_d*$ n'
by(*auto simp*:*targetnodes-def*)
thus *?thesis*
proof(*induct rule*:*DynPDG-path.cases*)
case (*DynPDG-path-Nil nx*)
hence *False* **by** *simp*
thus *?case* **by** *simp*
next
case (*DynPDG-path-Append-cdep nx asx n'' asx' nx'*)
from ‹$[a'] = asx@asx'$›
have $(asx = [a'] \wedge asx' = []) \vee (asx = [] \wedge asx' = [a'])$
by (*cases asx*) *auto*
hence *False*
proof
assume $asx = [a'] \wedge asx' = []$
with ‹n'' $-asx'\rightarrow_{cd}$ nx'› **show** *False*
by(*fastforce elim*:*DynPDG-edge.cases dest*:*dyn-control-dependence-path*)
next
assume $asx = [] \wedge asx' = [a']$
with ‹nx $-asx\rightarrow_d*$ n''› **have** $nx = n''$ **and** $asx' = [a']$
by(*auto intro*:*DynPDG-empty-path-eq-nodes*)
with ‹$n = nx$› ‹$n' = nx'$› ‹n'' $-asx'\rightarrow_{cd}$ nx'› *False*
show *False* **by** *simp*
qed
thus *?thesis* **by** *simp*
next
case (*DynPDG-path-Append-ddep nx asx n'' V asx' nx'*)
from ‹$[a'] = asx@asx'$›
have $(asx = [a'] \wedge asx' = []) \vee (asx = [] \wedge asx' = [a'])$
by (*cases asx*) *auto*
thus *?case*
proof
assume $asx = [a'] \wedge asx' = []$
with ‹n'' $-\{V\}asx'\rightarrow_{dd}$ nx'› **have** *False*
by(*fastforce elim*:*DynPDG-edge.cases simp*:*dyn-data-dependence-def*)
thus *?thesis* **by** *simp*
next
assume $asx = [] \wedge asx' = [a']$
with ‹nx $-asx\rightarrow_d*$ n''› **have** $nx = n''$
by(*simp add*:*DynPDG-empty-path-eq-nodes*)
{ fix V' **assume** $V' \in$ *Use n*
from ‹n'' $-\{V\}asx'\rightarrow_{dd}$ nx'› ‹$asx = [] \wedge asx' = [a']$› ‹$n' = nx'$›
have $(V,[],[]) \in$ *dependent-live-vars* n'
by(*fastforce intro*:*dep-vars-Use elim*:*DynPDG-edge.cases*
simp:*dyn-data-dependence-def*)

with ‹$V' \in$ *Use* n› ‹$n'' -\{V\}asx'\rightarrow_{dd}$ nx'› ‹$asx = [] \wedge asx' = [a']$›
‹$n = nx$› ‹$nx = n''$› ‹$n' = nx'$›
have $(V',[],[a']) \in$ *dependent-live-vars* n'
by(*auto elim*:*dep-vars-Cons-ddep simp*:*targetnodes-def*)
with *all* ‹$as = a'\#as'$› **have** *state-val s* V' = *state-val s'* V'
by *fastforce* **}**
with ‹*pred ex' s'*› ‹*valid-edge a'*› *ex ex' True* ‹$x \longrightarrow y$› **show** *?thesis*
by(*fastforce elim*:*CFG-edge-Uses-pred-equal*)
qed
qed
qed
qed *simp*
{ fix V **assume** $V \in$ *Use* n'
from ‹$V \in$ *Use* n'› **have** $(V,[],[]) \in$ *dependent-live-vars* n'
by(*rule dep-vars-Use*)
have *state-val* (*transfer ex s*) V = *state-val* (*transfer ex' s'*) V
proof(*cases* $n -\{V\}[a']\rightarrow_{dd} n'$)
case *True*
hence $V \in$ *Def n*
by(*auto elim*:*DynPDG-edge.cases simp*:*dyn-data-dependence-def*)
have $\bigwedge V.\ V \in$ *Use n* $\Longrightarrow$ *state-val s V* = *state-val s' V*
proof −
fix V' **assume** $V' \in$ *Use n*
with ‹$(V,[],[]) \in$ *dependent-live-vars* n'› *True*
have $(V',[],[a']) \in$ *dependent-live-vars* n'
by(*fastforce intro*:*dep-vars-Cons-ddep simp*:*targetnodes-def*)
with *all* ‹$as = a'\#as'$› **show** *state-val s* V' = *state-val s'* V' **by** *auto*
qed
with ‹*valid-edge a'*› ‹*pred ex' s'*› ‹*pred ex s*›
have $\forall V \in$ *Def n*. *state-val* (*transfer* (*kind a'*) *s*) V =
state-val (*transfer* (*kind a'*) *s'*) V
by *simp*(*rule CFG-edge-transfer-uses-only-Use*,*auto*)
with ‹$V \in$ *Def n*› **have** *state-val* (*transfer* (*kind a'*) *s*) V =
state-val (*transfer* (*kind a'*) *s'*) V
by *simp*
thus *?thesis* **by** *fastforce*
next
case *False*
with ‹*last*(*targetnodes as*) = n'› ‹$as = a'\#as'$›
‹$(V,[],[]) \in$ *dependent-live-vars* n'›
have $(V,[a'],[a']) \in$ *dependent-live-vars* n'
by(*fastforce intro*:*dep-vars-Cons-keep*)
from ‹$(V,[a'],[a']) \in$ *dependent-live-vars* n'› *all* ‹$as = a'\#as'$›
have *states-eq*:*state-val s V* = *state-val s' V*
by *auto*
from ‹*valid-edge a'*› ‹$V \in$ *Use* n'› *False* ‹*pred ex s*›
have *state-val* (*transfers* (*kinds* $[a']$) *s*) V = *state-val s V*
apply(*auto intro*!:*no-ddep-same-state path-edge simp*:*targetnodes-def*)
apply(*simp add*:*kinds-def*)

by(*case-tac as′*,*auto*)
moreover
from ‹*valid-edge a′*› ‹*V* ∈ *Use n′*› *False* ‹*pred ex′ s′*›
have *state-val* (*transfers* (*kinds* [*a′*]) *s′*) *V* = *state-val s′ V*
apply(*auto intro*!:*no-ddep-same-state path-edge simp*:*targetnodes-def*)
apply(*simp add*:*kinds-def*)
by(*case-tac as′*,*auto*)
ultimately show *?thesis* **using** *states-eq* **by**(*auto simp*:*kinds-def*)
qed }
hence ∀ *V* ∈ *Use n′*. *state-val* (*transfer ex s*) *V* =
state-val (*transfer ex′ s′*) *V* **by** *simp*
with ‹*pred ex s*› ‹*es* = *ex*#*esx*› ‹*es′* = *ex′*#*esx′*› **show** *?thesis* **by** *simp*
next
case *False*
with *ex* **have** *cases-x*:*ex* = (*case kind a′ of* ⇑*f* ⇒ ⇑*id* | (*Q*)$_\surd$ ⇒ (λ*s*. *True*)$_\surd$)
by *simp*
from *cases-x* **have** *pred ex s* **by**(*cases kind a′*,*auto*)
show *?thesis*
proof(*cases y*)
case *True*
with *ex′* **have** [*simp*]:*ex′* = *kind a′* **by** *simp*
{ fix *V* **assume** *V* ∈ *Use n′*
from ‹*V* ∈ *Use n′*› **have** (*V*,[],[]) ∈ *dependent-live-vars n′*
by(*rule dep-vars-Use*)
from ‹*slice-path as* = *x*#*xs*› ‹*as* = *a′*#*as′*› ‹¬ *x*›
have ¬ *n* −[*a′*]→$_d$∗ *n′* **by**(*simp add*:*targetnodes-def*)
hence ¬ *n* −{*V*}[*a′*]→$_{dd}$ *n′* **by**(*fastforce dest*:*DynPDG-path-ddep*)
with ‹*last*(*targetnodes as*) = *n′*› ‹*as* = *a′*#*as′*›
‹(*V*,[],[]) ∈ *dependent-live-vars n′*›
have (*V*,[*a′*],[*a′*]) ∈ *dependent-live-vars n′*
by(*fastforce intro*:*dep-vars-Cons-keep*)
with *all* ‹*as* = *a′*#*as′*› **have** *state-val s V* = *state-val s′ V* **by** *auto*
from ‹*valid-edge a′*› ‹*V* ∈ *Use n′*› ‹*pred ex′ s′*›
‹¬ *n* −{*V*}[*a′*]→$_{dd}$ *n′*› ‹*last*(*targetnodes as*) = *n′*› ‹*as* = *a′*#*as′*›
have *state-val* (*transfers* (*kinds* [*a′*]) *s′*) *V* = *state-val s′ V*
apply(*auto intro*!:*no-ddep-same-state path-edge*)
apply(*simp add*:*kinds-def*)
by(*case-tac as′*,*auto*)
with ‹*state-val s V* = *state-val s′ V*› *cases-x*
have *state-val* (*transfer ex s*) *V* =
state-val (*transfer ex′ s′*) *V*
by(*cases kind a′*,*simp-all add*:*kinds-def*) **}**
hence ∀ *V* ∈ *Use n′*. *state-val* (*transfer ex s*) *V* =
state-val (*transfer ex′ s′*) *V* **by** *simp*
with ‹*as* = *a′*#*as′*› ‹*es* = *ex*#*esx*› ‹*es′* = *ex′*#*esx′*› ‹*pred ex s*›
show *?thesis* **by** *simp*
next
case *False*
with *ex′* **have** *cases-y*:*ex′* = (*case kind a′ of* ⇑*f* ⇒ ⇑*id* | (*Q*)$_\surd$ ⇒ (λ*s*.

```
True)√)
        by simp
      with cases-x have [simp]:ex = ex' by(cases kind a') auto
      { fix V assume V ∈ Use n'
        from ‹V ∈ Use n'› have (V,[],[]) ∈ dependent-live-vars n'
          by(rule dep-vars-Use)
        from ‹slice-path as = x#xs› ‹as = a'#as'› ‹¬ x›
        have ¬ n −[a']→d* n' by(simp add:targetnodes-def)
        hence no-dep:¬ n −{V}[a']→dd n' by(fastforce dest:DynPDG-path-ddep)
        with ‹last(targetnodes as) = n'› ‹as = a'#as'›
          ‹(V,[],[]) ∈ dependent-live-vars n'›
        have (V,[a'],[a']) ∈ dependent-live-vars n'
          by(fastforce intro:dep-vars-Cons-keep)
        with all ‹as = a'#as'› have state-val s V = state-val s' V by auto }
      with ‹as = a'#as'› cases-x ‹es = ex#esx› ‹es' = ex'#esx'› ‹pred ex s›
      show ?thesis by(cases kind a',auto)
    qed
  qed
next
  case False
  show ?thesis
  proof(cases ∀ V xs. (V,xs,as') ∈ dependent-live-vars n' ⟶
                   state-val (transfer ex s) V = state-val (transfer ex' s') V)
    case True
    hence imp':∀ V xs. (V,xs,as') ∈ dependent-live-vars n' ⟶
                 state-val (transfer ex s) V = state-val (transfer ex' s') V .
    from IH[OF ‹targetnode a' −as'→* n'› ‹xs ⪯b ys› ‹slice-path as' = xs›
      ‹select-edge-kinds as' xs = esx› ‹select-edge-kinds as' ys = esx'›
      this ‹preds esx' (transfer ex' s')›]
    have all':∀ V∈Use n'. state-val (transfers esx (transfer ex s)) V =
                    state-val (transfers esx' (transfer ex' s')) V
      and preds esx (transfer ex s) by simp-all
    have pred ex s
    proof(cases ex)
      case (Predicate Q)
      with ‹slice-path as = x#xs› ‹as = a'#as'› ‹last(targetnodes as) = n'› ex
      have ex = (λs. True)√ ∨ n −a'#as'→d* n'
        by(cases kind a',auto split:if-split-asm)
      thus ?thesis
      proof
        assume ex = (λs. True)√ thus ?thesis by simp
      next
        assume n −a'#as'→d* n'
        with ‹slice-path as = x#xs› ‹as = a'#as'› ‹last(targetnodes as) = n'› ex
        have [simp]:ex = kind a' by clarsimp
        with ‹x ⟶ y› ex ex' have [simp]:ex' = ex by(cases x) auto
        from ‹n −a'#as'→d* n'› show ?thesis
        proof(induct rule:DynPDG-path-rev-cases)
          case DynPDG-path-Nil
```

hence *False* **by** *simp*
thus *?thesis* **by** *simp*
next
case (*DynPDG-path-cdep-Append* n'' *asx* asx')
from ‹$n -asx\rightarrow_{cd} n''$›**have** $asx \neq []$
by(*auto elim*:*DynPDG-edge.cases dest*:*dyn-control-dependence-path*)
with ‹$n -asx\rightarrow_{cd} n''$› ‹$n'' -asx'\rightarrow_{d}* n'$› ‹$a'\#as' = asx@asx'$›
have *cdep*:$\exists$ *as1 as2* n''. $n -a'\#as1\rightarrow_{cd} n'' \wedge$
$n'' -as2\rightarrow_{d}* n' \wedge as' = as1@as2$
by(*cases asx*) *auto*
{ fix *V* **assume** $V \in$ *Use n*
with *cdep* ‹*last*(*targetnodes as*) = n'› ‹$as = a'\#as'$›
have (*V*,[],*as*) $\in$ *dependent-live-vars* n'
by(*fastforce intro*:*dep-vars-Cons-cdep*)
with *all* **have** *state-val s V* = *state-val* s' *V* **by** *blast* **}**
with ‹*valid-edge* a'› ‹*pred ex*$'$ s'›
show *?thesis* **by**(*fastforce elim*:*CFG-edge-Uses-pred-equal*)
next
case (*DynPDG-path-ddep-Append V* n'' *asx* asx')
from ‹$n -\{V\}asx\rightarrow_{dd} n''$› **obtain** *ai ais* **where** $asx = ai\#ais$
by(*cases asx*)(*auto dest*:*DynPDG-ddep-edge-CFG-path*)
with ‹$n -\{V\}asx\rightarrow_{dd} n''$› **have** *sourcenode ai* = *n*
by(*fastforce dest*:*DynPDG-ddep-edge-CFG-path elim*:*path.cases*)
from ‹$n -\{V\}asx\rightarrow_{dd} n''$› ‹$asx = ai\#ais$›
have *last*(*targetnodes asx*) = n''
by(*fastforce intro*:*path-targetnode dest*:*DynPDG-ddep-edge-CFG-path*)
{ fix V' **assume** $V' \in$ *Use n*
from ‹$n -\{V\}asx\rightarrow_{dd} n''$› **have** (*V*,[],[]) $\in$ *dependent-live-vars* n''
by(*fastforce elim*:*DynPDG-edge.cases dep-vars-Use*
simp:*dyn-data-dependence-def*)
with ‹$n'' -asx'\rightarrow_{d}* n'$› **have** (*V*,[],[]@$asx'$) $\in$ *dependent-live-vars* n'
by(*rule dependent-live-vars-dep-dependent-live-vars*)
have (V',[],*as*) $\in$ *dependent-live-vars* n'
proof(*cases* $asx' = []$)
case *True*
with ‹$n'' -asx'\rightarrow_{d}* n'$› **have** $n'' = n'$
by(*fastforce intro*:*DynPDG-empty-path-eq-nodes*)
with ‹$n -\{V\}asx\rightarrow_{dd} n''$› ‹$V' \in$ *Use n*› *True* ‹$as = a'\#as'$›
‹$a'\#as' = asx@asx'$›
show *?thesis* **by**(*fastforce intro*:*dependent-live-vars-ddep-empty-fst*)
next
case *False*
with ‹$n -\{V\}asx\rightarrow_{dd} n''$› ‹$asx = ai\#ais$›
‹(*V*,[],[]@asx') $\in$ *dependent-live-vars* n'›
have (*V*,*ais*@[],*ais*@asx') $\in$ *dependent-live-vars* n'
by(*fastforce intro*:*ddep-dependent-live-vars-keep-notempty*)
from ‹$n'' -asx'\rightarrow_{d}* n'$› *False* **have** *last*(*targetnodes* asx') = n'
by −(*rule path-targetnode*,*rule DynPDG-path-CFG-path*)
with ‹(*V*,*ais*@[],*ais*@asx') $\in$ *dependent-live-vars* n'›

‹$V' \in$ *Use* n› ‹n −{V}*asx*$\rightarrow_{dd}$ n''› ‹*asx* = *ai*#*ais*›
‹*sourcenode ai* = n› ‹*last*(*targetnodes asx*) = n''› *False*
have (V',[],*ai*#*ais*@*asx*$'$) ∈ *dependent-live-vars* n'
by(*fastforce intro*:*dep-vars-Cons-ddep simp*:*targetnodes-def*)
with ‹*asx* = *ai*#*ais*› ‹a'#*as*$'$ = *asx*@*asx*$'$› ‹*as* = a'#*as*$'$›
show *?thesis* **by** *simp*
qed
with *all* **have** *state-val s* V' = *state-val s*$'$ V' **by** *blast* **}**
with ‹*pred ex*$'$ *s*$'$› ‹*valid-edge* a'›
show *?thesis* **by**(*fastforce elim*:*CFG-edge-Uses-pred-equal*)
qed
qed
qed *simp*
with *all*$'$ ‹*preds esx* (*transfer ex s*)› ‹*es* = *ex*#*esx*› ‹*es*$'$ = *ex*$'$#*esx*$'$›
show *?thesis* **by** *simp*
next
case *False*
then obtain V' *xs*$'$ **where** (V',*xs*$'$,*as*$'$) ∈ *dependent-live-vars* n'
and *state-val* (*transfer ex s*) $V' \neq$ *state-val* (*transfer ex*$'$ *s*$'$) V'
by *auto*
show *?thesis*
proof(*cases* n −a'#*as*$'$$\rightarrow_{d}$∗ n')
case *True*
with ‹*slice-path as* = *x*#*xs*› ‹*as* = a'#*as*$'$› ‹*last*(*targetnodes as*) = n'› *ex*
have [*simp*]:*ex* = *kind* a' **by** *clarsimp*
with ‹$x \longrightarrow y$› *ex ex*$'$ **have** [*simp*]:*ex*$'$ = *ex* **by**(*cases x*) *auto*
{ fix V **assume** $V \in$ *Use* (*sourcenode* a')
hence (V,[],[]) ∈ *dependent-live-vars* (*sourcenode* a')
by(*rule dep-vars-Use*)
with ‹n −a'#*as*$'$$\rightarrow_{d}$∗ n'› **have** (V,[],[]@a'#*as*$'$) ∈ *dependent-live-vars* n'
by(*fastforce intro*:*dependent-live-vars-dep-dependent-live-vars*)
with *all* ‹*as* = a'#*as*$'$› **have** *state-val s* V = *state-val s*$'$ V
by *fastforce* **}**
with ‹*pred ex*$'$ *s*$'$› ‹*valid-edge* a'› **have** *pred ex s*
by(*fastforce intro*:*CFG-edge-Uses-pred-equal*)
show *?thesis*
proof(*cases* $V' \in$ *Def n*)
case *True*
with ‹*state-val* (*transfer ex s*) $V' \neq$ *state-val* (*transfer ex*$'$ *s*$'$) V'›
‹*valid-edge* a'› ‹*pred ex*$'$ *s*$'$› ‹*pred ex s*›
CFG-edge-transfer-uses-only-Use[*of* a' *s s*$'$]
obtain V'' **where** $V'' \in$ *Use n*
and *state-val s* $V'' \neq$ *state-val s*$'$ V''
by *auto*
from *True* ‹(V',*xs*$'$,*as*$'$) ∈ *dependent-live-vars* n'›
‹*targetnode* a' −*as*$'$→∗ n'› ‹*last*(*targetnodes as*) = n'› ‹*as* = a'#*as*$'$›
‹*valid-edge* a'› ‹n = *sourcenode* a'›[*THEN sym*]
have n −{V'}a'#*xs*$'$$\rightarrow_{dd}$ *last*(*targetnodes* (a'#*xs*$'$))
by −(*drule dependent-live-vars-dependent-edge*,

auto dest!: *dependent-live-vars-dependent-edge*
dest:*DynPDG-ddep-edge-CFG-path path-targetnode*
simp del:‹*n* = *sourcenode a*′›)
with ‹(V',*xs*′,*as*′) ∈ *dependent-live-vars n*′› ‹V'' ∈ *Use n*›
‹*last*(*targetnodes as*) = *n*′› ‹*as* = *a*′#*as*′›
have (V'',[],*as*) ∈ *dependent-live-vars n*′
by(*fastforce intro*:*dep-vars-Cons-ddep*)
with *all* **have** *state-val s* V'' = *state-val s*′ V'' **by** *blast*
with ‹*state-val s* V'' ≠ *state-val s*′ V''› **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
next
case *False*
with ‹*valid-edge a*′› ‹*pred ex s*›
have *state-val* (*transfer* (*kind a*′) *s*) V' = *state-val s* V'
by(*fastforce intro*:*CFG-edge-no-Def-equal*)
moreover
from *False* ‹*valid-edge a*′› ‹*pred ex*′ *s*′›
have *state-val* (*transfer* (*kind a*′) *s*′) V' = *state-val s*′ V'
by(*fastforce intro*:*CFG-edge-no-Def-equal*)
ultimately have *state-val s* V' ≠ *state-val s*′ V'
using ‹*state-val* (*transfer ex s*) V' ≠ *state-val* (*transfer ex*′ *s*′) V'›
by *simp*
from *False* **have** ¬ *n* −{V'}*a*′#*xs*′$\rightarrow_{dd}$
last(*targetnodes* (*a*′#*xs*′))
by(*auto elim*:*DynPDG-edge.cases simp*:*dyn-data-dependence-def*)
with ‹(V',*xs*′,*as*′) ∈ *dependent-live-vars n*′› ‹*last*(*targetnodes as*) = *n*′›
‹*as* = *a*′#*as*′›
have (V',*a*′#*xs*′,*a*′#*as*′) ∈ *dependent-live-vars n*′
by(*fastforce intro*:*dep-vars-Cons-keep*)
with ‹*as* = *a*′#*as*′› *all* **have** *state-val s* V' = *state-val s*′ V' **by** *auto*
with ‹*state-val s* V' ≠ *state-val s*′ V'› **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
qed
next
case *False*
{ assume V' ∈ *Def n*
with ‹(V',*xs*′,*as*′) ∈ *dependent-live-vars n*′› ‹*targetnode a*′ −*as*′→∗ *n*′›
‹*valid-edge a*′›
have *n* −*a*′#*as*′$\rightarrow_d$∗ *n*′
by −(*drule dependent-live-vars-dependent-edge*,
auto dest:*DynPDG-path-ddep DynPDG-path-Append*)
with *False* **have** *False* **by** *simp* **}**
hence V' ∉ *Def* (*sourcenode a*′) **by** *fastforce*
from *False* ‹*slice-path as* = *x*#*xs*› ‹*as* = *a*′#*as*′›
‹*last*(*targetnodes as*) = *n*′› ‹*as*′ ≠ []›
have ¬ *x* **by**(*auto simp*:*targetnodes-def*)
with *ex* **have** *cases*:*ex* = (*case kind a*′ *of* ⇑*f* ⇒ ⇑*id* | $(Q)_{\surd}$ ⇒ $(\lambda s.\ True)_{\surd}$)
by *simp*
have *state-val s* V' ≠ *state-val s*′ V'

proof(*cases y*)
case *True*
with *ex'* **have** [*simp*]:*ex'* = *kind a'* **by** *simp*
from ‹*V'* ∉ *Def* (*sourcenode a'*)› ‹*valid-edge a'*› ‹*pred ex' s'*›
have *states-eq*:*state-val* (*transfer* (*kind a'*) *s'*) *V'* = *state-val s' V'*
by(*fastforce intro*:*CFG-edge-no-Def-equal*)
from *cases* **have** *state-val s V'* = *state-val* (*transfer ex s*) *V'*
by(*cases kind a'*) *auto*
with *states-eq*
‹*state-val* (*transfer ex s*) *V'* ≠ *state-val* (*transfer ex' s'*) *V'*›
show *?thesis* **by** *simp*
next
case *False*
with *ex'* **have** *ex'* = (*case kind a' of* ⇑*f* ⇒ ⇑*id* | $(Q)_{\surd}$ ⇒ $(\lambda s.\ True)_{\surd}$)
by *simp*
with *cases* **have** *state-val s V'* = *state-val* (*transfer ex s*) *V'*
and *state-val s' V'* = *state-val* (*transfer ex' s'*) *V'*
by(*cases kind a'*,*auto*)+
with ‹*state-val* (*transfer ex s*) *V'* ≠ *state-val* (*transfer ex' s'*) *V'*›
show *?thesis* **by** *simp*
qed
from ‹*V'* ∉ *Def* (*sourcenode a'*)›
have ¬ *n* −{*V'*}*a'*#*xs'*$\rightarrow_{dd}$ *last*(*targetnodes* (*a'*#*xs'*))
by(*auto elim*:*DynPDG-edge.cases simp*:*dyn-data-dependence-def*)
with ‹(*V'*,*xs'*,*as'*) ∈ *dependent-live-vars n'*› ‹*last*(*targetnodes as*) = *n'*›
‹*as* = *a'*#*as'*›
have (*V'*,*a'*#*xs'*,*a'*#*as'*) ∈ *dependent-live-vars n'*
by(*fastforce intro*:*dep-vars-Cons-keep*)
with ‹*as* = *a'*#*as'*› *all* **have** *state-val s V'* = *state-val s' V'* **by** *auto*
with ‹*state-val s V'* ≠ *state-val s' V'*› **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
qed
qed
qed
qed *simp-all*

theorem *fundamental-property-of-path-slicing*:
assumes *n* −*as*→∗ *n'* **and** *preds* (*kinds as*) *s*
shows (∀ *V* ∈ *Use n'*. *state-val* (*transfers* (*slice-kinds as*) *s*) *V* =
state-val (*transfers* (*kinds as*) *s*) *V*)
and *preds* (*slice-kinds as*) *s*
proof −
have *length as* = *length* (*slice-path as*) **by**(*simp add*:*slice-path-length*)
hence *slice-path as* $\preceq_b$ *replicate* (*length as*) *True*
by(*simp add*:*maximal-element*)
have *select-edge-kinds as* (*replicate* (*length as*) *True*) = *kinds as*
by(*rule select-edge-kinds-max-bv*)
with ‹*n* −*as*→∗ *n'*› ‹*slice-path as* $\preceq_b$ *replicate* (*length as*) *True*›

‹*preds* (*kinds as*) *s*›
have (∀ *V*∈*Use n*′. *state-val* (*transfers* (*slice-kinds as*) *s*) *V* =
state-val (*transfers* (*kinds as*) *s*) *V*) ∧ *preds* (*slice-kinds as*) *s*
by −(*rule slice-path-leqs-information-same-Uses*,*simp-all add*:*slice-kinds-def*)
thus ∀ *V*∈*Use n*′. *state-val* (*transfers* (*slice-kinds as*) *s*) *V* =
state-val (*transfers* (*kinds as*) *s*) *V* **and** *preds* (*slice-kinds as*) *s*
by *simp-all*
qed

end

2.4.3 The fundamental property of (dynamic) slicing related to the semantics

locale *BackwardPathSlice-wf* =
DynPDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
dyn-control-dependence +
CFG-semantics-wf sourcenode targetnode kind valid-edge Entry sem identifies
for *sourcenode* :: ′*edge* ⇒ ′*node* **and** *targetnode* :: ′*edge* ⇒ ′*node*
and *kind* :: ′*edge* ⇒ ′*state edge-kind* **and** *valid-edge* :: ′*edge* ⇒ *bool*
and *Entry* :: ′*node* (‹′(′-*Entry*′-′)›) **and** *Def* :: ′*node* ⇒ ′*var set*
and *Use* :: ′*node* ⇒ ′*var set* **and** *state-val* :: ′*state* ⇒ ′*var* ⇒ ′*val*
and *dyn-control-dependence* :: ′*node* ⇒ ′*node* ⇒ ′*edge list* ⇒ *bool*
(‹- *controls* - *via* -› [*51*, *0*, *0*] *1000*)
and *Exit* :: ′*node* (‹′(′-*Exit*′-′)›)
and *sem* :: ′*com* ⇒ ′*state* ⇒ ′*com* ⇒ ′*state* ⇒ *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 *n* ≜ *c* **and** ⟨*c*,*s*⟩ ⇒ ⟨*c*′,*s*′⟩
obtains *n*′ *as* **where** *n* −*as*→∗ *n*′ **and** *preds* (*slice-kinds as*) *s*
and *n*′ ≜ *c*′
and ∀ *V* ∈ *Use n*′. *state-val* (*transfers* (*slice-kinds as*) *s*) *V* =
state-val s′ *V*
proof(*atomize-elim*)
from ‹*n* ≜ *c*› ‹⟨*c*,*s*⟩ ⇒ ⟨*c*′,*s*′⟩› **obtain** *n*′ *as* **where** *n* −*as*→∗ *n*′
and *transfers* (*kinds as*) *s* = *s*′
and *preds* (*kinds as*) *s*
and *n*′ ≜ *c*′
by(*fastforce dest*:*fundamental-property*)
with ‹*n* −*as*→∗ *n*′› ‹*preds* (*kinds as*) *s*›
have ∀ *V* ∈ *Use n*′. *state-val* (*transfers* (*slice-kinds as*) *s*) *V* =
state-val (*transfers* (*kinds as*) *s*) *V* **and** *preds* (*slice-kinds as*) *s*
by −(*rule fundamental-property-of-path-slicing*,*simp-all*)+
with ‹*transfers* (*kinds as*) *s* = *s*′› **have** ∀ *V* ∈ *Use n*′.
state-val (*transfers* (*slice-kinds as*) *s*) *V* =

state-val s' V **by** *simp*
with ‹n $-as\rightarrow*$ n'› ‹*preds* (*slice-kinds as*) *s*› ‹$n' \triangleq c'$›
show $\exists$ *as* n'. n $-as\rightarrow*$ n' $\wedge$ *preds* (*slice-kinds as*) *s* $\wedge$ $n' \triangleq c'$ $\wedge$
($\forall$ $V \in$*Use* n'. *state-val* (*transfers* (*slice-kinds as*) *s*) V = *state-val* s' V)
by *blast*
qed

end

end

2.5 Observable Sets of Nodes

theory *Observable* **imports** ../*Basic*/*CFG* **begin**

context *CFG* **begin**

inductive-set *obs* :: $'node \Rightarrow 'node\ set \Rightarrow 'node\ set$
for n::$'node$ **and** S::$'node\ set$
where *obs-elem*:
$[\![n$ $-as\rightarrow*$ n'; $\forall$ $nx \in$ *set*(*sourcenodes as*). $nx \notin S$; $n' \in S]\!] \Longrightarrow n' \in$ *obs* n S

lemma *obsE*:
assumes $n' \in$ *obs* n S
obtains *as* **where** n $-as\rightarrow*$ n' **and** $\forall$ $nx \in$ *set*(*sourcenodes as*). $nx \notin S$
and $n' \in S$
proof(*atomize-elim*)
from ‹$n' \in$ *obs* n S›
have $\exists$ *as*. n $-as\rightarrow*$ n' $\wedge$ ($\forall$ $nx \in$ *set*(*sourcenodes as*). $nx \notin S$) $\wedge$ $n' \in S$
by(*auto elim*:*obs.cases*)
thus $\exists$ *as*. n $-as\rightarrow*$ n' $\wedge$ ($\forall$ $nx \in$*set* (*sourcenodes as*). $nx \notin S$) $\wedge$ $n' \in S$ **by** *blast*
qed

lemma *n-in-obs*:
assumes *valid-node* n **and** $n \in S$ **shows** *obs* n S = $\{n\}$
proof −
from ‹*valid-node* n› **have** n $-[]\rightarrow*$ n **by**(*rule empty-path*)
with ‹$n \in S$› **have** $n \in$ *obs* n S **by**(*fastforce elim*:*obs-elem simp*:*sourcenodes-def*)
{ fix n' **assume** $n' \in$ *obs* n S
have $n' = n$
proof(*rule ccontr*)
assume $n' \neq n$
from ‹$n' \in$ *obs* n S› **obtain** *as* **where** n $-as\rightarrow*$ n'
and $\forall$ $nx \in$ *set*(*sourcenodes as*). $nx \notin S$
and $n' \in S$ **by**(*erule obsE*)
from ‹n $-as\rightarrow*$ n'› ‹$\forall$ $nx \in$ *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 ‹∀ *nx*∈*set* (*sourcenodes* (*a*#*as*)). *nx* ∉ *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* ∈ *obs n S*› **show** *?thesis* **by** *fastforce*
qed

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

lemma *edge-obs-subset*:
assumes*valid-edge a* **and** *sourcenode a* ∉ *S*
shows *obs* (*targetnode a*) *S* ⊆ *obs* (*sourcenode a*) *S*
proof
fix *n* **assume** *n* ∈ *obs* (*targetnode a*) *S*
then obtain *as* **where** *targetnode a* −*as*→∗ *n*
and *all*:∀ *nx* ∈ *set*(*sourcenodes as*). *nx* ∉ *S* **and** *n* ∈ *S* **by**(*erule obsE*)
from ‹*valid-edge a*› ‹*targetnode a* −*as*→∗ *n*›
have *sourcenode a* −*a*#*as*→∗ *n* **by**(*fastforce intro*:*Cons-path*)
moreover
from *all* ‹*sourcenode a* ∉ *S*› **have** ∀ *nx* ∈ *set*(*sourcenodes* (*a*#*as*)). *nx* ∉ *S*
by(*simp add*:*sourcenodes-def*)
ultimately show *n* ∈ *obs* (*sourcenode a*) *S* **using** ‹*n* ∈ *S*›
by(*rule obs-elem*)
qed

lemma *path-obs-subset*:
⟦*n* −*as*→∗ n'; ∀ n' ∈ *set*(*sourcenodes as*). $n' \notin S$⟧
⟹ *obs* n' *S* ⊆ *obs n S*
proof(*induct rule*:*path.induct*)
case (*Cons-path* n'' *as* n' *a* n)
note *IH* = ‹∀ n'∈*set* (*sourcenodes as*). $n' \notin S$ ⟹ *obs* n' *S* ⊆ *obs* n'' *S*›
from ‹∀ n'∈*set* (*sourcenodes* (*a*#*as*)). $n' \notin S$›
have *all*:∀ n'∈*set* (*sourcenodes as*). $n' \notin S$ **and** *sourcenode a* ∉ *S*
by(*simp-all add*:*sourcenodes-def*)
from *IH*[*OF all*] **have** *obs* n' *S* ⊆ *obs* n'' *S* **.**
from ‹*valid-edge a*› ‹*targetnode a* = n''› ‹*sourcenode a* = *n*› ‹*sourcenode a* ∉ *S*›
have *obs* n'' *S* ⊆ *obs n S* **by**(*fastforce dest*:*edge-obs-subset*)
with ‹*obs* n' *S* ⊆ *obs* n'' *S*› **show** *?case* **by** *fastforce*
qed *simp*

lemma *path-ex-obs*:
  assumes $n\ {-as\rightarrow*}\ n'$ **and** $n' \in S$
  obtains m **where** $m \in obs\ n\ S$
proof(*atomize-elim*)
  show $\exists\, m.\ m \in obs\ n\ S$
  proof(*cases* $\forall\, nx \in set(sourcenodes\ as).\ nx \notin S$)
    case *True*
    with ‹$n\ {-as\rightarrow*}\ n'$› ‹$n' \in S$› **have** $n' \in obs\ n\ S$ **by** −(*rule obs-elem*)
    thus *?thesis* **by** *fastforce*
  next
    case *False*
    hence $\exists\, nx \in set(sourcenodes\ as).\ nx \in S$ **by** *fastforce*
    then obtain *nx ns ns'* **where** $sourcenodes\ as = ns@nx\#ns'$
      and $nx \in S$ **and** $\forall\, n' \in set\ ns.\ n' \notin S$
      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 ‹$n\ {-as\rightarrow*}\ n'$› **have** $n\ {-as'\rightarrow*}\ nx$ **by**(*fastforce dest*:*path-split*)
    with ‹$nx \in S$› ‹$\forall\, n' \in set\ ns.\ n' \notin S$› ‹$ns = sourcenodes\ as'$› **have** $nx \in obs\ n$
S
      by(*fastforce intro*:*obs-elem*)
    thus *?thesis* **by** *fastforce*
  qed
qed

end

end

Chapter 3

Static Intraprocedural Slicing

theory *Distance* **imports** *../Basic/CFG* **begin**

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 differently than correctness of dynamic slicing by 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).

We also show how a "dynamic slicing style" correctness criterion for static slicing of a given trace and initial state could look like.

This formalization of static intraprocedural slicing is instantiable with three different kinds of control dependences: standard control, weak control and weak order dependence. The correctness proof for slicing is independent of the control dependence used, it bases only on one property every control dependence definition hass to fulfill.

3.1 Distance of Paths

context *CFG* **begin**

inductive *distance* :: *'node* ⇒ *'node* ⇒ *nat* ⇒ *bool*
where *distanceI*:
$[\![n \ -as\rightarrow* \ n'; \ length \ as = x; \ \forall \, as'. \ n \ -as'\rightarrow* \ n' \longrightarrow x \leq length \ as']\!]$
$\Longrightarrow distance \ n \ n' \ x$

lemma *every-path-distance*:
  assumes $n -as\rightarrow* n'$
  obtains x **where** *distance* $n\ n'\ x$ **and** $x \leq$ *length as*
proof –
  have $\exists x.$ *distance* $n\ n'\ x \wedge x \leq$ *length as*
  proof(*cases* $\exists as'.\ n -as'\rightarrow* n' \wedge$
                 $(\forall asx.\ n -asx\rightarrow* n' \longrightarrow$ *length* $as' \leq$ *length asx*$)$)
    case *True*
    then obtain as'
      where $n -as'\rightarrow* n' \wedge (\forall asx.\ n -asx\rightarrow* n' \longrightarrow$ *length* $as' \leq$ *length asx*$)$
      by *blast*
    hence $n -as'\rightarrow* n'$ **and** *all*:$\forall asx.\ n -asx\rightarrow* n' \longrightarrow$ *length* $as' \leq$ *length asx*
      by *simp-all*
    hence *distance* $n\ n'$ (*length* as') **by**(*fastforce intro*:*distanceI*)
    from ‹$n -as\rightarrow* n'$› *all* **have** *length* $as' \leq$ *length as* **by** *fastforce*
    with ‹*distance* $n\ n'$ (*length* as')› **show** *?thesis* **by** *blast*
  next
    case *False*
    hence *all*:$\forall as'.\ n -as'\rightarrow* n' \longrightarrow (\exists asx.\ n -asx\rightarrow* n' \wedge$ *length* $as' >$ *length asx*$)$
      by *fastforce*
    have *wf* (*measure length*) **by** *simp*
    from ‹$n -as\rightarrow* n'$› **have** $as \in \{as.\ n -as\rightarrow* n'\}$ **by** *simp*
    with ‹*wf* (*measure length*)› **obtain** as' **where** $as' \in \{as.\ n -as\rightarrow* n'\}$
      and *notin*:$\bigwedge as''.\ (as'',as') \in$ (*measure length*) $\Longrightarrow as'' \notin \{as.\ n -as\rightarrow* n'\}$
      by(*erule wfE-min*)
    from ‹$as' \in \{as.\ n -as\rightarrow* n'\}$› **have** $n -as'\rightarrow* n'$ **by** *simp*
    with *all* **obtain** *asx* **where** $n -asx\rightarrow* n'$
      and *length* $as' >$ *length asx*
      by *blast*
    with *notin* **have** $asx \notin \{as.\ n -as\rightarrow* n'\}$ **by** *simp*
    hence $\neg\ n -asx\rightarrow* n'$ **by** *simp*
    with ‹$n -asx\rightarrow* n'$› **have** *False* **by** *simp*
    thus *?thesis* **by** *simp*
  qed
  with *that* **show** *?thesis* **by** *blast*
qed

lemma *distance-det*:
  $[\![$*distance* $n\ n'\ x$; *distance* $n\ n'\ x']\!] \Longrightarrow x = x'$
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*:*valid-edge a* **and** *dist*:*distance* (*targetnode a*) $n'\ x$

shows $\exists!a'.$ *sourcenode a* = *sourcenode a'* ∧ *distance* (*targetnode a'*) *n' x* ∧
valid-edge a' ∧
targetnode a' = (*SOME nx.* ∃ *a'. sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *n' x* ∧
valid-edge a' ∧ *targetnode a'* = *nx*)
proof(*rule ex-ex1I*)
show ∃ *a'. sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *n' x* ∧ *valid-edge a'* ∧
targetnode a' = (*SOME nx.* ∃ *a'. sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *n' x* ∧
valid-edge a' ∧ *targetnode a'* = *nx*)
proof –
have (∃ *a'. sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *n' x* ∧ *valid-edge a'* ∧
targetnode a' = (*SOME nx.* ∃ *a'. sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *n' x* ∧
valid-edge a' ∧ *targetnode a'* = *nx*)) =
(∃ *nx.* ∃ *a'. sourcenode a* = *sourcenode a'* ∧ *distance* (*targetnode a'*) *n' x* ∧
valid-edge 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'* ∧ *targetnode a'* = *nx*])
by *simp*
also have ... **using** *valid dist* **by** *blast*
finally show *?thesis* **.**
qed
next
fix *a' ax*
assume *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *n' x* ∧ *valid-edge a'* ∧
targetnode a' = (*SOME nx.* ∃ *a'. sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *n' x* ∧
valid-edge a' ∧ *targetnode a'* = *nx*)
and *sourcenode a* = *sourcenode ax* ∧
distance (*targetnode ax*) *n' x* ∧ *valid-edge ax* ∧
targetnode ax = (*SOME nx.* ∃ *a'. sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *n' x* ∧
valid-edge 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 *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' ∧ *targetnode a'* = *nx*)
proof –

have $\exists a.$ *valid-edge* $a \wedge n =$ *sourcenode* $a \wedge$ *distance* (*targetnode* a) n' $(x - 1)$ $\wedge$
targetnode $a =$ (*SOME* $nx.$ $\exists a'.$ *sourcenode* $a =$ *sourcenode* $a' \wedge$
distance (*targetnode* a') n' $(x - 1) \wedge$
valid-edge $a' \wedge$ *targetnode* $a' = nx$)
proof(*rule ccontr*)
assume $\neg$ ($\exists a.$ *valid-edge* $a \wedge n =$ *sourcenode* $a \wedge$
distance (*targetnode* a) n' $(x - 1) \wedge$
targetnode $a =$ (*SOME* $nx.$ $\exists a'.$ *sourcenode* $a =$ *sourcenode* $a' \wedge$
distance (*targetnode* a') n' $(x - 1) \wedge$
valid-edge $a' \wedge$ *targetnode* $a' = nx$))
hence *imp*:$\forall a.$ *valid-edge* $a \wedge n =$ *sourcenode* $a \wedge$
targetnode $a =$ (*SOME* $nx.$ $\exists a'.$ *sourcenode* $a =$ *sourcenode* $a' \wedge$
distance (*targetnode* a') n' $(x - 1) \wedge$
valid-edge $a' \wedge$ *targetnode* $a' = nx$)
$\longrightarrow \neg$ *distance* (*targetnode* a) n' $(x - 1)$ **by** *blast*
from ‹*distance* n n' x› **obtain** *as* **where** n *−as→∗* n' **and** $x =$ *length as*
and $\forall as'.$ n *−as'→∗* $n' \longrightarrow x \leq$ *length* as'
by(*auto elim*:*distance.cases*)
thus *False* **using** *imp*
proof(*induct rule*:*path.induct*)
case (*empty-path* n)
from ‹$x =$ *length* []› ‹$x \neq 0$› **show** *False* **by** *simp*
next
case (*Cons-path* n'' *as* n' a n)
note *imp* = ‹$\forall a.$ *valid-edge* $a \wedge n =$ *sourcenode* $a \wedge$
targetnode $a =$ (*SOME* $nx.$ $\exists a'.$ *sourcenode* $a =$ *sourcenode* $a' \wedge$
distance (*targetnode* a') n' $(x - 1) \wedge$
valid-edge $a' \wedge$ *targetnode* $a' = nx$)
$\longrightarrow \neg$ *distance* (*targetnode* a) n' $(x - 1)$›
note *all* = ‹$\forall as'.$ n *−as'→∗* $n' \longrightarrow x \leq$ *length* as'›
from ‹n'' *−as→∗* n'› **obtain** y **where** *distance* n'' n' y
and $y \leq$ *length as* **by**(*erule every-path-distance*)
from ‹*distance* n'' n' y› **obtain** as' **where** n'' *−as'→∗* n'
and $y =$ *length* as'
by(*auto elim*:*distance.cases*)
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 *all* **have** $x \leq$ *length* (*a#as'*) **by** *blast*
with ‹$x =$ *length* (*a#as*)› *True* ‹$y =$ *length* as'› **show** *False* **by** *simp*
next
case *False*
with ‹$y \leq$ *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*›
obtain a' **where** *sourcenode a* = *sourcenode* a'
and *distance* (*targetnode* a') n' *y* **and** *valid-edge* a'
and *targetnode* a' = (*SOME nx*. $\exists\, a'$. *sourcenode a* = *sourcenode* $a' \wedge$
distance (*targetnode* a') n' *y* $\wedge$
valid-edge $a' \wedge$ *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
with *that* **show** *?thesis* **by** *blast*
qed

end

end

3.2 Static data dependence

theory *DataDependence* **imports** *../Basic/DynDataDependence* **begin**

context *CFG-wf* **begin**

definition *data-dependence* :: *'node* ⇒ *'var* ⇒ *'node* ⇒ *bool*
(‹- *influences* - *in* -› [*51*,*0*])
where *data-dependences-eq*:*n influences V in* n' ≡ ∃ *as*. *n influences V in* n' *via as*

lemma *data-dependence-def*: *n influences V in* n' =
(∃ a' as'. (*V* ∈ *Def n*) ∧ (*V* ∈ *Use* n') ∧
(n −a'#as'→∗ n') ∧ (∀ n'' ∈ *set* (*sourcenodes* as'). *V* ∉ *Def* n''))
by(*auto simp*:*data-dependences-eq dyn-data-dependence-def*)

end

end

3.3 Static backward slice

theory *Slice*
imports *Observable Distance DataDependence ../Basic/SemanticsCFG*
begin

locale *BackwardSlice* =
CFG-wf sourcenode targetnode kind valid-edge Entry Def Use state-val
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*

and *Entry* :: *'node* (‹'('-*Entry*'-')›) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val* +
fixes *backward-slice* :: *'node set* ⇒ *'node set*
assumes *valid-nodes*:*n* ∈ *backward-slice S* ⟹ *valid-node n*
and *refl*:⟦*valid-node n*; *n* ∈ *S*⟧ ⟹ *n* ∈ *backward-slice S*
and *dd-closed*:⟦*n'* ∈ *backward-slice S*; *n influences V in n'*⟧
⟹ *n* ∈ *backward-slice S*
and *obs-finite*:*finite* (*obs n* (*backward-slice S*))
and *obs-singleton*:*card* (*obs n* (*backward-slice S*)) ≤ *1*

begin

lemma *slice-n-in-obs*:
n ∈ *backward-slice S* ⟹ *obs n* (*backward-slice S*) = {*n*}
by(*fastforce intro*!:*n-in-obs dest*:*valid-nodes*)

lemma *obs-singleton-disj*:
(∃ *m*. *obs n* (*backward-slice S*) = {*m*}) ∨ *obs n* (*backward-slice S*) = {}
proof −
have *finite*(*obs n* (*backward-slice S*)) **by**(*rule obs-finite*)
show *?thesis*
proof(*cases card*(*obs n* (*backward-slice S*)) = *0*)
case *True*
with ‹*finite*(*obs n* (*backward-slice S*))› **have** *obs n* (*backward-slice S*) = {}
by *simp*
thus *?thesis* **by** *simp*
next
case *False*
have *card*(*obs n* (*backward-slice S*)) ≤ *1* **by**(*rule obs-singleton*)
with *False* **have** *card*(*obs n* (*backward-slice S*)) = *1*
by *simp*
hence ∃ *m*. *obs n* (*backward-slice S*) = {*m*} **by**(*fastforce dest*:*card-eq-SucD*)
thus *?thesis* **by** *simp*
qed
qed

lemma *obs-singleton-element*:
assumes *m* ∈ *obs n* (*backward-slice S*) **shows** *obs n* (*backward-slice S*) = {*m*}
proof −
have (∃ *m*. *obs n* (*backward-slice S*) = {*m*}) ∨ *obs n* (*backward-slice S*) = {}
by(*rule obs-singleton-disj*)
with ‹*m* ∈ *obs n* (*backward-slice S*)› **show** *?thesis* **by** *fastforce*
qed

lemma *obs-the-element*:
m ∈ *obs n* (*backward-slice S*) ⟹ (*THE m*. *m* ∈ *obs n* (*backward-slice S*)) = *m*
by(*fastforce dest*:*obs-singleton-element*)

3.3.1 Traversing the sliced graph

slice-kind S a conforms to *kind a* in the sliced graph

definition *slice-kind* :: *'node set ⇒ 'edge ⇒ 'state edge-kind*
  where *slice-kind S a = (let S' = backward-slice S; n = sourcenode a in*
  (if sourcenode a ∈ S' then kind a
   else (case kind a of ⇑f ⇒ ⇑id | (Q)$_\surd$ ⇒
    (if obs (sourcenode a) S' = {} then
      (let nx = (SOME n'. ∃ a'. n = sourcenode a' ∧ valid-edge a' ∧ targetnode a'
= n')
      in (if (targetnode a = nx) then (λs. True)$_\surd$ else (λs. False)$_\surd$))
     else (let m = THE m. m ∈ obs n S' in
       (if (∃ x. distance (targetnode a) m x ∧ distance n m (x + 1) ∧
            (targetnode a = (SOME nx'. ∃ a'. sourcenode a = sourcenode a' ∧
                                    distance (targetnode a') m x ∧
                                    valid-edge a' ∧ targetnode a' = nx')))
          then (λs. True)$_\surd$ else (λs. False)$_\surd$
       ))
     ))
  ))

definition
  slice-kinds :: *'node set ⇒ 'edge list ⇒ 'state edge-kind list*
  where *slice-kinds S as ≡ map (slice-kind S) as*

lemma *slice-kind-in-slice*:
  sourcenode a ∈ backward-slice S ⟹ slice-kind S a = kind a
by(*simp add:slice-kind-def*)

lemma *slice-kind-Upd*:
  ⟦*sourcenode a ∉ backward-slice S*; *kind a = ⇑f*⟧ ⟹ *slice-kind S a = ⇑id*
by(*simp add:slice-kind-def*)

lemma *slice-kind-Pred-empty-obs-SOME*:
  ⟦*sourcenode a ∉ backward-slice S*; *kind a = (Q)$_\surd$*;
    obs (sourcenode a) (backward-slice S) = {};
    targetnode a = (SOME n'. ∃ a'. sourcenode a = sourcenode a' ∧ valid-edge a' ∧
                              targetnode a' = n')⟧
  ⟹ *slice-kind S a = (λs. True)$_\surd$*
by(*simp add:slice-kind-def*)

lemma *slice-kind-Pred-empty-obs-not-SOME*:
  ⟦*sourcenode a ∉ backward-slice S*; *kind a = (Q)$_\surd$*;
    obs (sourcenode a) (backward-slice S) = {};

targetnode a ≠ (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧ *valid-edge a'* ∧
targetnode a' = *n'*)⟧
⟹ *slice-kind S a* = (λ*s*. *False*)$_{\surd}$
by(*simp add*:*slice-kind-def*)

lemma *slice-kind-Pred-obs-nearer-SOME*:
assumes *sourcenode a* ∉ *backward-slice S* **and** *kind a* = (*Q*)$_{\surd}$
and *m* ∈ *obs* (*sourcenode a*) (*backward-slice S*)
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' ∧ *targetnode a'* = *n'*)
shows *slice-kind S a* = (λ*s*. *True*)$_{\surd}$
proof −
from ‹*m* ∈ *obs* (*sourcenode a*) (*backward-slice S*)›
have *m* = (*THE m*. *m* ∈ *obs* (*sourcenode a*) (*backward-slice S*))
by(*rule obs-the-element*[*THEN sym*])
with *assms* **show** *?thesis*
by(*fastforce simp*:*slice-kind-def Let-def*)
qed

lemma *slice-kind-Pred-obs-nearer-not-SOME*:
assumes *sourcenode a* ∉ *backward-slice S* **and** *kind a* = (*Q*)$_{\surd}$
and *m* ∈ *obs* (*sourcenode a*) (*backward-slice S*)
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' ∧ *targetnode a'* = *nx'*)
shows *slice-kind S a* = (λ*s*. *False*)$_{\surd}$
proof −
from ‹*m* ∈ *obs* (*sourcenode a*) (*backward-slice S*)›
have *m* = (*THE m*. *m* ∈ *obs* (*sourcenode a*) (*backward-slice S*))
by(*rule obs-the-element*[*THEN sym*])
with *assms* **show** *?thesis*
by(*fastforce dest*:*distance-det simp*:*slice-kind-def Let-def*)
qed

lemma *slice-kind-Pred-obs-not-nearer*:
assumes *sourcenode a* ∉ *backward-slice S* **and** *kind a* = (*Q*)$_{\surd}$
and *in-obs*:*m* ∈ *obs* (*sourcenode a*) (*backward-slice S*)
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* (*sourcenode a*) (*backward-slice S*))
by(*rule obs-the-element*[*THEN sym*])

from *dist* **have** $\neg$ ($\exists$ *x*. *distance* (*targetnode a*) *m x* $\wedge$
distance (*sourcenode a*) *m* (*x* + *1*))
by(*fastforce dest*:*distance-det*)
with ‹*sourcenode a* $\notin$ *backward-slice S*› ‹*kind a* = $(Q)_{\surd}$› *in-obs the* **show** *?thesis*
by(*fastforce simp*:*slice-kind-def Let-def*)
qed

lemma *kind-Predicate-notin-slice-slice-kind-Predicate*:
assumes *kind a* = $(Q)_{\surd}$ **and** *sourcenode a* $\notin$ *backward-slice S*
obtains *Q′* **where** *slice-kind S a* = $(Q')_{\surd}$ **and** *Q′* = (λ*s*. *False*) $\vee$ *Q′* = (λ*s*. *True*)
proof(*atomize-elim*)
show $\exists$ *Q′*. *slice-kind S a* = $(Q')_{\surd}$ $\wedge$ (*Q′* = (λ*s*. *False*) $\vee$ *Q′* = (λ*s*. *True*))
proof(*cases obs* (*sourcenode a*) (*backward-slice S*) = {})
case *True*
show *?thesis*
proof(*cases targetnode a* = (*SOME n′*. $\exists$ *a′*. *sourcenode a* = *sourcenode a′* $\wedge$
valid-edge a′ $\wedge$ *targetnode a′* = *n′*))
case *True*
with ‹*sourcenode a* $\notin$ *backward-slice S*› ‹*kind a* = $(Q)_{\surd}$›
‹*obs* (*sourcenode a*) (*backward-slice S*) = {}›
have *slice-kind S a* = (λ*s*. *True*)$_{\surd}$ **by**(*rule slice-kind-Pred-empty-obs-SOME*)
thus *?thesis* **by** *simp*
next
case *False*
with ‹*sourcenode a* $\notin$ *backward-slice S*› ‹*kind a* = $(Q)_{\surd}$›
‹*obs* (*sourcenode a*) (*backward-slice S*) = {}›
have *slice-kind S a* = (λ*s*. *False*)$_{\surd}$
by(*rule slice-kind-Pred-empty-obs-not-SOME*)
thus *?thesis* **by** *simp*
qed
next
case *False*
then obtain *m* **where** *m* $\in$ *obs* (*sourcenode a*) (*backward-slice S*) **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′*. $\exists$ *a′*. *sourcenode a* = *sourcenode a′* $\wedge$
distance (*targetnode a′*) *m x* $\wedge$
valid-edge a′ $\wedge$ *targetnode a′* = *n′*))
case *True*
with ‹*sourcenode a* $\notin$ *backward-slice S*› ‹*kind a* = $(Q)_{\surd}$›
‹*m* $\in$ *obs* (*sourcenode a*) (*backward-slice S*)›
‹*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* ∉ *backward-slice S*› ‹*kind a* = (*Q*)$_{\surd}$›
      ‹*m* ∈ *obs* (*sourcenode a*) (*backward-slice S*)›
      ‹*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* (*sourcenode a*) (*backward-slice S*)›
  have *m* = (*THE m. m* ∈ *obs* (*sourcenode a*) (*backward-slice S*))
    by(*rule obs-the-element*[*THEN sym*])
  with ‹*sourcenode a* ∉ *backward-slice S*› ‹*kind a* = (*Q*)$_{\surd}$› *False*
    ‹*m* ∈ *obs* (*sourcenode a*) (*backward-slice S*)›
  have *slice-kind S a* = (λ*s. False*)$_{\surd}$
    by(*fastforce simp*:*slice-kind-def Let-def*)
  thus *?thesis* **by** *simp*
qed
qed
qed

lemma *only-one-SOME-edge*:
  assumes *valid-edge a*
  shows ∃!*a*′. *sourcenode a* = *sourcenode a*′ ∧ *valid-edge a*′ ∧
      targetnode a′ = (*SOME n*′. ∃ *a*′. *sourcenode a* = *sourcenode a*′ ∧
        valid-edge a′ ∧ *targetnode a*′ = *n*′)
proof(*rule ex-ex1I*)
  show ∃ *a*′. *sourcenode a* = *sourcenode a*′ ∧ *valid-edge a*′ ∧
      targetnode a′ = (*SOME n*′. ∃ *a*′. *sourcenode a* = *sourcenode a*′ ∧
        valid-edge a′ ∧ *targetnode a*′ = *n*′)
  proof −
    have (∃ *a*′. *sourcenode a* = *sourcenode a*′ ∧ *valid-edge a*′ ∧
        targetnode a′ = (*SOME n*′. ∃ *a*′. *sourcenode a* = *sourcenode a*′ ∧
          valid-edge a′ ∧ *targetnode a*′ = *n*′)) =
      (∃ *n*′. ∃ *a*′. *sourcenode a* = *sourcenode a*′ ∧ *valid-edge a*′ ∧ *targetnode a*′ = *n*′)
      apply(*unfold some-eq-ex*[*of* λ*n*′. ∃ *a*′. *sourcenode a* = *sourcenode a*′ ∧
          valid-edge a′ ∧ *targetnode a*′ = *n*′])
      by *simp*
    also have ... **using** ‹*valid-edge a*› **by** *blast*
    finally show *?thesis* **.**
  qed
next
  fix *a*′ *ax*

assume *sourcenode a* = *sourcenode a'* ∧ *valid-edge a'* ∧
targetnode a' = (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
valid-edge a' ∧ *targetnode a'* = *n'*)
and *sourcenode a* = *sourcenode ax* ∧ *valid-edge ax* ∧
targetnode ax = (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
valid-edge 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** *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*:∀ *s*. (*Q s* ⟶ ¬ *Q' s*) ∧ (*Q' s* ⟶ ¬ *Q s*)
by(*auto dest*:*deterministic*)
from ‹*valid-edge a*› **have** *ex1*:∃!*a'*. *sourcenode a* = *sourcenode a'* ∧ *valid-edge a'*
∧
targetnode a' = (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
valid-edge a' ∧ *targetnode a'* = *n'*)
by(*rule only-one-SOME-edge*)
show *?thesis*
proof(*cases sourcenode a* ∈ *backward-slice S*)
case *True*
with ‹*slice-kind S a* = $(\lambda s.\ True)_{\surd}$› ‹*kind a* = $(Q)_{\surd}$› **have** *Q* = (λ*s*. *True*)
by(*simp add*:*slice-kind-def Let-def*)
with *det* **have** *Q'* = (λ*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* ∉ *backward-slice S* **by** *simp*
thus *?thesis*
proof(*cases obs* (*sourcenode a*) (*backward-slice S*) = {})
case *True*
with ‹*sourcenode a* ∉ *backward-slice S*› ‹*slice-kind S a* = $(\lambda s.\ True)_{\surd}$›
‹*kind a* = $(Q)_{\surd}$›
have *target*:*targetnode a* = (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
valid-edge a' ∧ *targetnode a'* = *n'*)
by(*auto simp*:*slice-kind-def Let-def fun-eq-iff split*:*if-split-asm*)
have *targetnode a'* ≠ (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
valid-edge a' ∧ *targetnode a'* = *n'*)
proof(*rule ccontr*)
assume ¬ *targetnode a'* ≠ (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
valid-edge a' ∧ *targetnode a'* = *n'*)
hence *targetnode a'* = (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
valid-edge a' ∧ *targetnode a'* = *n'*)

```
      by simp
    with ex1 target ‹sourcenode a = sourcenode a'› ‹valid-edge a›
      ‹valid-edge a'› have a = a' by blast
    with ‹targetnode a ≠ targetnode a'› show False by simp
  qed
  with ‹sourcenode a ∉ backward-slice S› True ‹kind a' = (Q')√›
    ‹sourcenode a = sourcenode a'› show ?thesis
    by(auto simp:slice-kind-def Let-def fun-eq-iff split:if-split-asm)
next
  case False
  hence obs (sourcenode a) (backward-slice S) ≠ {} .
  then obtain m where m ∈ obs (sourcenode a) (backward-slice S) by auto
  hence m = (THE m. m ∈ obs (sourcenode a) (backward-slice S))
    by(auto dest:obs-the-element)
  with ‹sourcenode a ∉ backward-slice S›
    ‹obs (sourcenode a) (backward-slice S) ≠ {}›
    ‹slice-kind S a = (λs. True)√› ‹kind a = (Q)√›
  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' ∧ targetnode a' = n')
    by(auto simp:slice-kind-def Let-def fun-eq-iff split:if-split-asm)
  show ?thesis
  proof(cases distance (targetnode a') m x)
    case False
    with ‹sourcenode a ∉ backward-slice S› ‹kind a' = (Q')√›
      ‹m ∈ obs (sourcenode a) (backward-slice S)›
      ‹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› ‹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' ∧
           targetnode a' = (SOME nx. ∃ a'. sourcenode a = sourcenode a' ∧
                                   distance (targetnode a') m x ∧
                                   valid-edge a' ∧ targetnode a' = nx)
      by(fastforce intro!:only-one-SOME-dist-edge)
    have targetnode a' ≠ (SOME n'. ∃ a'. sourcenode a = sourcenode a' ∧
                                   distance (targetnode a') m x ∧
                                   valid-edge 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' ∧ targetnode a' = n')
```

hence *targetnode a′* = (*SOME n′*. ∃ *a′*. *sourcenode a* = *sourcenode a′* ∧
distance (*targetnode a′*) *m x* ∧
valid-edge a′ ∧ *targetnode a′* = *n′*)
by *simp*
with *ex1 target* ‹*sourcenode a* = *sourcenode a′*›
‹*valid-edge a*› ‹*valid-edge 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* ∉ *backward-slice S*›
‹*kind a′* = $(Q')_{\surd}$› ‹*m* ∈ *obs* (*sourcenode a*) (*backward-slice S*)›
‹*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 *sourcenode a* = *sourcenode a′* **and** *targetnode a* ≠ *targetnode a′*
obtains *Q Q′* **where** *slice-kind S a* = $(Q)_{\surd}$ **and** *slice-kind S a′* = $(Q')_{\surd}$
and ∀ *s*. (*Q s* ⟶ ¬ *Q′ s*) ∧ (*Q′ s* ⟶ ¬ *Q s*)
proof(*atomize-elim*)
from *assms* **obtain** *Q Q′*
where *kind a* = $(Q)_{\surd}$ **and** *kind a′* = $(Q')_{\surd}$
and *det*:∀ *s*. (*Q s* ⟶ ¬ *Q′ s*) ∧ (*Q′ s* ⟶ ¬ *Q s*)
by(*auto dest*:*deterministic*)
from ‹*valid-edge a*› **have** *ex1*:∃!*a′*. *sourcenode a* = *sourcenode a′* ∧ *valid-edge a′*
∧
targetnode a′ = (*SOME n′*. ∃ *a′*. *sourcenode a* = *sourcenode a′* ∧
valid-edge a′ ∧ *targetnode a′* = *n′*)
by(*rule only-one-SOME-edge*)
show ∃ *Q Q′*. *slice-kind S a* = $(Q)_{\surd}$ ∧ *slice-kind S a′* = $(Q')_{\surd}$ ∧
(∀ *s*. (*Q s* ⟶ ¬ *Q′ s*) ∧ (*Q′ s* ⟶ ¬ *Q s*))
proof(*cases sourcenode a* ∈ *backward-slice S*)
case *True*
with ‹*kind a* = $(Q)_{\surd}$› **have** *slice-kind S a* = $(Q)_{\surd}$
by(*simp add*:*slice-kind-def Let-def*)
from *True* ‹*kind a′* = $(Q')_{\surd}$› ‹*sourcenode a* = *sourcenode a′*›
have *slice-kind S a′* = $(Q')_{\surd}$
by(*simp add*:*slice-kind-def Let-def*)
with ‹*slice-kind S a* = $(Q)_{\surd}$› *det* **show** *?thesis* **by** *blast*
next
case *False*
with ‹*kind a* = $(Q)_{\surd}$›

have *slice-kind S a* = (λ*s. True*)$_{\surd}$ ∨ *slice-kind S a* = (λ*s. False*)$_{\surd}$
by(*simp add*:*slice-kind-def Let-def*)
thus *?thesis*
proof
assume *true*:*slice-kind S a* = (λ*s. True*)$_{\surd}$
with ‹*sourcenode a* = *sourcenode a'*› ‹*targetnode a* ≠ *targetnode a'*›
‹*valid-edge a*› ‹*valid-edge a'*›
have *slice-kind S a'* = (λ*s. False*)$_{\surd}$
by(*rule slice-kind-only-one-True-edge*)
with *true* **show** *?thesis* **by** *simp*
next
assume *false*:*slice-kind S a* = (λ*s. False*)$_{\surd}$
from *False* ‹*kind a'* = (*Q'*)$_{\surd}$› ‹*sourcenode a* = *sourcenode a'*›
have *slice-kind S a'* = (λ*s. True*)$_{\surd}$ ∨ *slice-kind S a'* = (λ*s. False*)$_{\surd}$
by(*simp add*:*slice-kind-def Let-def*)
with *false* **show** *?thesis* **by** *auto*
qed
qed
qed

3.3.2 Observable and silent moves

inductive *silent-move* ::
'node set ⇒ (*'edge* ⇒ *'state edge-kind*) ⇒ *'node* ⇒ *'state* ⇒ *'edge* ⇒
'node ⇒ *'state* ⇒ *bool* (‹-,- ⊢ '(-,-') −-→$_{\tau}$ '(-,-')› [*51*,*50*,*0*,*0*,*50*,*0*,*0*] *51*)

where *silent-moveI*:
⟦*pred* (*f a*) *s*; *transfer* (*f a*) *s* = *s'*; *sourcenode a* ∉ *backward-slice S*;
valid-edge a⟧
⟹ *S*,*f* ⊢ (*sourcenode a*,*s*) −*a*→$_{\tau}$ (*targetnode a*,*s'*)

inductive *silent-moves* ::
'node set ⇒ (*'edge* ⇒ *'state edge-kind*) ⇒ *'node* ⇒ *'state* ⇒ *'edge list* ⇒
'node ⇒ *'state* ⇒ *bool* (‹-,- ⊢ '(-,-') =-⇒$_{\tau}$ '(-,-')› [*51*,*50*,*0*,*0*,*50*,*0*,*0*] *51*)

where *silent-moves-Nil*: *S*,*f* ⊢ (*n*,*s*) =[]⇒$_{\tau}$ (*n*,*s*)

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

lemma *silent-moves-obs-slice*:
⟦*S*,*f* ⊢ (*n*,*s*) =*as*⇒$_{\tau}$ (*n'*,*s'*); *nx* ∈ *obs n'* (*backward-slice S*)⟧
⟹ *nx* ∈ *obs n* (*backward-slice S*)
proof(*induct rule*:*silent-moves.induct*)
case *silent-moves-Nil* **thus** *?case* **by** *simp*
next

case (*silent-moves-Cons S f n s a n′ s′ as n″ s″*)
from ‹*nx* ∈ *obs n″* (*backward-slice S*)›
‹*nx* ∈ *obs n″* (*backward-slice S*) ⟹ *nx* ∈ *obs n′* (*backward-slice S*)›
have *obs*:*nx* ∈ *obs n′* (*backward-slice S*) **by** *simp*
from ‹*S*,*f* ⊢ (*n*,*s*) −*a*→$_\tau$ (*n′*,*s′*)›
have *n* = *sourcenode a* **and** *n′* = *targetnode a* **and** *valid-edge a*
and *n* ∉ (*backward-slice S*)
by(*auto elim*:*silent-move.cases*)
hence *obs n′* (*backward-slice S*) ⊆ *obs n* (*backward-slice S*)
by *simp*(*rule edge-obs-subset*,*simp*+)
with *obs* **show** *?case* **by** *blast*
qed

lemma *silent-moves-preds-transfers-path*:
⟦*S*,*f* ⊢ (*n*,*s*) =*as*⇒$_\tau$ (*n′*,*s′*); *valid-node n*⟧
⟹ *preds* (*map f as*) *s* ∧ *transfers* (*map f as*) *s* = *s′* ∧ *n* −*as*→∗ *n′*
proof(*induct rule*:*silent-moves.induct*)
case *silent-moves-Nil* **thus** *?case* **by**(*simp add*:*path.empty-path*)
next
case (*silent-moves-Cons S f n s a n′ s′ as n″ s″*)
note *IH* = ‹*valid-node n′* ⟹
preds (*map f as*) *s′* ∧ *transfers* (*map f as*) *s′* = *s″* ∧ *n′* −*as*→∗ *n″*›
from ‹*S*,*f* ⊢ (*n*,*s*) −*a*→$_\tau$ (*n′*,*s′*)› **have** *pred* (*f a*) *s* **and** *transfer* (*f a*) *s* = *s′*
and *n* = *sourcenode a* **and** *n′* = *targetnode a* **and** *valid-edge a*
by(*auto elim*:*silent-move.cases*)
from ‹*n′* = *targetnode a*› ‹*valid-edge a*› **have** *valid-node n′* **by** *simp*
from *IH*[*OF this*] **have** *preds* (*map f as*) *s′* **and** *transfers* (*map f as*) *s′* = *s″*
and *n′* −*as*→∗ *n″* **by** *simp-all*
from ‹*n* = *sourcenode a*› ‹*n′* = *targetnode a*› ‹*valid-edge a*› ‹*n′* −*as*→∗ *n″*›
have *n* −*a*#*as*→∗ *n″* **by**(*fastforce intro*:*Cons-path*)
with ‹*pred* (*f a*) *s*› ‹*preds* (*map f as*) *s′*› ‹*transfer* (*f a*) *s* = *s′*›
‹*transfers* (*map f as*) *s′* = *s″*› **show** *?case* **by** *simp*
qed

lemma *obs-silent-moves*:
assumes *obs n* (*backward-slice S*) = {*n′*}
obtains *as* **where** *S*,*slice-kind S* ⊢ (*n*,*s*) =*as*⇒$_\tau$ (*n′*,*s*)
proof(*atomize-elim*)
from ‹*obs n* (*backward-slice S*) = {*n′*}›
have *n′* ∈ *obs n* (*backward-slice S*) **by** *simp*
then obtain *as* **where** *n* −*as*→∗ *n′*
and ∀ *nx* ∈ *set*(*sourcenodes as*). *nx* ∉ (*backward-slice S*)
and *n′* ∈ (*backward-slice S*) **by**(*erule obsE*)
from ‹*n* −*as*→∗ *n′*› **obtain** *x* **where** *distance n n′ x* **and** *x* ≤ *length as*
by(*erule every-path-distance*)
from ‹*distance n n′ x*› ‹*n′* ∈ *obs n* (*backward-slice S*)›
show ∃ *as*. *S*,*slice-kind S* ⊢ (*n*,*s*) =*as*⇒$_\tau$ (*n′*,*s*)

proof(*induct x arbitrary*:*n s rule*:*nat.induct*)
fix *n s* **assume** *distance n n′ 0*
then obtain *as′* **where** *n* −*as′*→∗ *n′* **and** *length as′* = *0*
by(*auto elim*:*distance.cases*)
hence *n* −[]→∗ *n′* **by**(*cases as*) *auto*
hence *n* = *n′* **by**(*fastforce elim*:*path.cases*)
hence *S*,*slice-kind S* ⊢ (*n*,*s*) =[]⇒$_\tau$ (*n′*,*s*) **by**(*fastforce intro*:*silent-moves-Nil*)
thus ∃ *as. S*,*slice-kind S* ⊢ (*n*,*s*) =*as*⇒$_\tau$ (*n′*,*s*) **by** *blast*
next
fix *x n s*
assume *distance n n′* (*Suc x*) **and** *n′* ∈ *obs n* (*backward-slice S*)
and *IH*:⋀*n s.* ⟦*distance n n′ x*; *n′* ∈ *obs n* (*backward-slice S*)⟧
⟹ ∃ *as. S*,*slice-kind S* ⊢ (*n*,*s*) =*as*⇒$_\tau$ (*n′*,*s*)
from ‹*n′* ∈ *obs n* (*backward-slice S*)›
have *valid-node n* **by**(*rule in-obs-valid*)
with ‹*distance n n′* (*Suc x*)›
have *n* ≠ *n′* **by**(*fastforce elim*:*distance.cases dest*:*empty-path*)
have *n* ∉ *backward-slice S*
proof
assume *isin*:*n* ∈ *backward-slice S*
with ‹*valid-node n*› **have** *obs n* (*backward-slice S*) = {*n*}
by(*fastforce intro*!:*n-in-obs*)
with ‹*n′* ∈ *obs n* (*backward-slice S*)› ‹*n* ≠ *n′*› **show** *False* **by** *simp*
qed
from ‹*distance n n′* (*Suc x*)› **obtain** *a* **where** *valid-edge a*
and *n* = *sourcenode a* **and** *distance* (*targetnode a*) *n′ x*
and *target*:*targetnode a* = (*SOME nx.* ∃ *a′. sourcenode a* = *sourcenode a′* ∧
distance (*targetnode a′*) *n′ x* ∧
valid-edge a′ ∧ *targetnode a′* = *nx*)
by −(*erule distance-successor-distance*,*simp*+)
from ‹*n′* ∈ *obs n* (*backward-slice S*)›
have *obs n* (*backward-slice S*) = {*n′*}
by(*rule obs-singleton-element*)
with ‹*valid-edge a*› ‹*n* ∉ *backward-slice S*› ‹*n* = *sourcenode a*›
have *disj*:*obs* (*targetnode a*) (*backward-slice S*) = {} ∨
obs (*targetnode a*) (*backward-slice S*) = {*n′*}
by −(*drule-tac S*=*backward-slice S* **in** *edge-obs-subset*,*auto*)
from ‹*distance* (*targetnode a*) *n′ x*› **obtain** *asx* **where** *targetnode a* −*asx*→∗
n′
and *length asx* = *x* **and** ∀ *as′. targetnode a* −*as′*→∗ *n′* ⟶ *x* ≤ *length as′*
by(*auto elim*:*distance.cases*)
from ‹*targetnode a* −*asx*→∗ *n′*› ‹*n′* ∈ (*backward-slice S*)›
obtain *m* **where** ∃ *m. m* ∈ *obs* (*targetnode a*) (*backward-slice S*)
by(*fastforce elim*:*path-ex-obs*)
with *disj* **have** *n′* ∈ *obs* (*targetnode a*) (*backward-slice S*) **by** *fastforce*
from *IH*[*OF* ‹*distance* (*targetnode a*) *n′ x*› *this*,*of transfer* (*slice-kind S a*) *s*]
obtain *asx′* **where**
moves:*S*,*slice-kind S* ⊢ (*targetnode a*,*transfer* (*slice-kind S a*) *s*) =*asx′*⇒$_\tau$
(*n′*,*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*)
case (*Update f*)
with ‹*n* ∉ *backward-slice S*› ‹*n* = *sourcenode a*› **have** *slice-kind S a* = ⇑*id*
by(*fastforce intro*:*slice-kind-Upd*)
thus *?thesis* **by** *simp*
next
case (*Predicate Q*)
with ‹*n* ∉ *backward-slice S*› ‹*n* = *sourcenode a*›
‹*n′* ∈ *obs n* (*backward-slice S*)› ‹*distance* (*targetnode a*) *n′ x*›
‹*distance n n′* (*Suc x*)› *target*
have *slice-kind S a* = (λ*s*. *True*)$_{\surd}$
by(*fastforce intro*:*slice-kind-Pred-obs-nearer-SOME*)
thus *?thesis* **by** *simp*
qed
hence *pred* (*slice-kind S a*) *s* **and** *transfer* (*slice-kind S a*) *s* = *s*
by *simp-all*
with ‹*n* ∉ *backward-slice S*› ‹*n* = *sourcenode a*› ‹*valid-edge a*›
have *S*,*slice-kind S* ⊢ (*sourcenode a*,*s*) −*a*→$_\tau$
(*targetnode a*,*transfer* (*slice-kind S a*) *s*)
by(*fastforce intro*:*silent-moveI*)
with *moves* ‹*transfer* (*slice-kind S a*) *s* = *s*› ‹*n* = *sourcenode a*›
have *S*,*slice-kind S* ⊢ (*n*,*s*) =*a*#*asx′*⇒$_\tau$ (*n′*,*s*)
by(*fastforce intro*:*silent-moves-Cons*)
thus ∃ *as*. *S*,*slice-kind S* ⊢ (*n*,*s*) =*as*⇒$_\tau$ (*n′*,*s*) **by** *blast*
qed
qed

inductive *observable-move* ::
′node set ⇒ (*′edge* ⇒ *′state edge-kind*) ⇒ *′node* ⇒ *′state* ⇒ *′edge* ⇒
′node ⇒ *′state* ⇒ *bool* (‹-,- ⊢ ′(-,-′) −-→ ′(-,-′)› [*51*,*50*,*0*,*0*,*50*,*0*,*0*] *51*)

where *observable-moveI*:
⟦*pred* (*f a*) *s*; *transfer* (*f a*) *s* = *s′*; *sourcenode a* ∈ *backward-slice S*;
valid-edge a⟧
⟹ *S*,*f* ⊢ (*sourcenode a*,*s*) −*a*→ (*targetnode a*,*s′*)

inductive *observable-moves* ::
′node set ⇒ (*′edge* ⇒ *′state edge-kind*) ⇒ *′node* ⇒ *′state* ⇒ *′edge list* ⇒
′node ⇒ *′state* ⇒ *bool* (‹-,- ⊢ ′(-,-′) =-⇒ ′(-,-′)› [*51*,*50*,*0*,*0*,*50*,*0*,*0*] *51*)

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

lemma *observable-move-notempty*:

$[\![S,f \vdash (n,s) =as\Rightarrow (n',s');\ as = []]\!] \Longrightarrow False$
by(*induct rule*:*observable-moves.induct*,*simp*)

lemma *silent-move-observable-moves*:
$[\![S,f \vdash (n'',s'') =as\Rightarrow (n',s');\ S,f \vdash (n,s) -a\rightarrow_\tau (n'',s'')]\!]$
$\Longrightarrow S,f \vdash (n,s) =a\#as\Rightarrow (n',s')$
proof(*induct rule*:*observable-moves.induct*)
case (*observable-moves-snoc S f nx sx as n' s' a' n'' s''*)
from ‹$S,f \vdash (n,s) -a\rightarrow_\tau (nx,sx)$› ‹$S,f \vdash (nx,sx) =as\Rightarrow_\tau (n',s')$›
have $S,f \vdash (n,s) =a\#as\Rightarrow_\tau (n',s')$ **by**(*rule silent-moves-Cons*)
with ‹$S,f \vdash (n',s') -a'\rightarrow (n'',s'')$›
have $S,f \vdash (n,s) =(a\#as)@[a']\Rightarrow (n'',s'')$
by −(*rule observable-moves.observable-moves-snoc*)
thus *?case* **by** *simp*
qed

lemma *observable-moves-preds-transfers-path*:
$S,f \vdash (n,s) =as\Rightarrow (n',s')$
$\Longrightarrow$ *preds* (*map f as*) *s* $\wedge$ *transfers* (*map f as*) $s = s' \wedge n -as\rightarrow* n'$
proof(*induct rule*:*observable-moves.induct*)
case (*observable-moves-snoc S f n s as n' s' a n'' s''*)
have *valid-node n*
proof(*cases as*)
case *Nil*
with ‹$S,f \vdash (n,s) =as\Rightarrow_\tau (n',s')$› **have** $n = n'$ **and** $s = s'$
by(*auto elim*:*silent-moves.cases*)
with ‹$S,f \vdash (n',s') -a\rightarrow (n'',s'')$› **show** *?thesis*
by(*fastforce elim*:*observable-move.cases*)
next
case (*Cons a' as'*)
with ‹$S,f \vdash (n,s) =as\Rightarrow_\tau (n',s')$› **show** *?thesis*
by(*fastforce elim*:*silent-moves.cases silent-move.cases*)
qed
with ‹$S,f \vdash (n,s) =as\Rightarrow_\tau (n',s')$›
have *preds* (*map f as*) *s* **and** *transfers* (*map f as*) $s = s'$
and $n -as\rightarrow* n'$ **by**(*auto dest*:*silent-moves-preds-transfers-path*)
from ‹$S,f \vdash (n',s') -a\rightarrow (n'',s'')$› **have** *pred* (*f a*) s'
and *transfer* (*f a*) $s' = s''$ **and** $n' =$ *sourcenode a* **and** $n'' =$ *targetnode a*
and *valid-edge a*
by(*auto elim*:*observable-move.cases*)
from ‹$n' =$ *sourcenode a*› ‹$n'' =$ *targetnode a*› ‹*valid-edge a*›
have $n' -[a]\rightarrow* n''$ **by**(*fastforce intro*:*path.intros*)
with ‹$n -as\rightarrow* n'$› **have** $n -as@[a]\rightarrow* n''$ **by**(*rule path-Append*)
with ‹*preds* (*map f as*) *s*› ‹*pred* (*f a*) s'› ‹*transfer* (*f a*) $s' = s''$›
‹*transfers* (*map f as*) $s = s'$›
show *?case* **by**(*simp add*:*transfers-split preds-split*)
qed

3.3.3 Relevant variables

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

where *rvI*:
  ⟦*n* −*as*→∗ *n'*; *n'* ∈ *backward-slice S*; *V* ∈ *Use n'*;
   ∀ *nx* ∈ *set*(*sourcenodes as*). *V* ∉ *Def nx*⟧
  ⟹ *V* ∈ *rv S n*

lemma *rvE*:
  assumes *rv*:*V* ∈ *rv S n*
  obtains *as n'* **where** *n* −*as*→∗ *n'* **and** *n'* ∈ *backward-slice S* **and** *V* ∈ *Use n'*
  and ∀ *nx* ∈ *set*(*sourcenodes as*). *V* ∉ *Def nx*
using *rv*
by(*atomize-elim*,*auto elim*!:*relevant-vars.cases*)

lemma *eq-obs-in-rv*:
  assumes *obs-eq*:*obs n* (*backward-slice S*) = *obs n'* (*backward-slice S*)
  and *x* ∈ *rv S n* **shows** *x* ∈ *rv S n'*
proof −
  from ‹*x* ∈ *rv S n*› **obtain** *as m*
    where *n* −*as*→∗ *m* **and** *m* ∈ *backward-slice S* **and** *x* ∈ *Use m*
    and ∀ *nx*∈*set* (*sourcenodes as*). *x* ∉ *Def nx*
    by(*erule rvE*)
  from ‹*n* −*as*→∗ *m*› **have** *valid-node m* **by**(*fastforce dest*:*path-valid-node*)
  from ‹*n* −*as*→∗ *m*› ‹*m* ∈ *backward-slice S*›
  have ∃ *nx as' as''*. *nx* ∈ *obs n* (*backward-slice S*) ∧ *n* −*as'*→∗ *nx* ∧
                nx −*as''*→∗ *m* ∧ *as* = *as'*@*as''*
  proof(*cases* ∀ *nx* ∈ *set*(*sourcenodes as*). *nx* ∉ *backward-slice S*)
    case *True*
    with ‹*n* −*as*→∗ *m*› ‹*m* ∈ *backward-slice S*› **have** *m* ∈ *obs n* (*backward-slice S*)
      by −(*rule obs-elem*)
    with ‹*n* −*as*→∗ *m*› ‹*valid-node m*› **show** *?thesis* **by**(*blast intro*:*empty-path*)
  next
    case *False*
    hence ∃ *nx* ∈ *set*(*sourcenodes as*). *nx* ∈ *backward-slice S* **by** *simp*
    then obtain *nx' ns ns'* **where** *sourcenodes as* = *ns*@*nx'*#*ns'*
      and *nx'* ∈ *backward-slice S*
      and ∀ *x* ∈ *set ns*. *x* ∉ *backward-slice 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*)
    from ‹*n* −*as*→∗ *m*› ‹*as* = *as'*@*a'*#*as''*› ‹*sourcenode a'* = *nx'*›
    have *n* −*as'*→∗ *nx'* **and** *valid-edge a'* **and** *targetnode a'* −*as''*→∗ *m*

by(*fastforce dest*:*path-split*)+
with ‹*sourcenode a′* = *nx′*› **have** *nx′* −*a′*#*as″*→∗ *m* **by**(*fastforce intro*:*Cons-path*)
from ‹*n* −*as′*→∗ *nx′*› ‹*nx′* ∈ *backward-slice S*›
‹∀ *x* ∈ *set ns*. *x* ∉ *backward-slice S*› ‹*ns* = *sourcenodes as′*›
have *nx′* ∈ *obs n* (*backward-slice S*)
by(*fastforce intro*:*obs-elem*)
with ‹*n* −*as′*→∗ *nx′*› ‹*nx′* −*a′*#*as″*→∗ *m*› ‹*as* = *as′*@*a′*#*as″*› **show** *?thesis*
by *blast*
qed
then obtain *nx as′ as″* **where** *nx* ∈ *obs n* (*backward-slice S*)
and *n* −*as′*→∗ *nx* **and** *nx* −*as″*→∗ *m* **and** *as* = *as′*@*as″*
by *blast*
from ‹*nx* ∈ *obs n* (*backward-slice S*)› *obs-eq*
have *nx* ∈ *obs n′* (*backward-slice S*) **by** *auto*
then obtain *asx* **where** *n′* −*asx*→∗ *nx*
and ∀ *ni* ∈ *set*(*sourcenodes asx*). *ni* ∉ *backward-slice S*
and *nx* ∈ *backward-slice S*
by(*erule obsE*)
from ‹*as* = *as′*@*as″*› ‹∀ *nx*∈*set* (*sourcenodes as*). *x* ∉ *Def nx*›
have ∀ *ni*∈*set* (*sourcenodes as″*). *x* ∉ *Def ni*
by(*auto simp*:*sourcenodes-def*)
from ‹∀ *ni* ∈ *set*(*sourcenodes asx*). *ni* ∉ *backward-slice S*› ‹*n′* −*asx*→∗ *nx*›
have ∀ *ni* ∈ *set*(*sourcenodes asx*). *x* ∉ *Def ni*
proof(*induct asx arbitrary*:*n′*)
case *Nil* **thus** *?case* **by**(*simp add*:*sourcenodes-def*)
next
case (*Cons ax′ asx′*)
note *IH* = ‹⋀*n′*. ⟦∀ *ni*∈*set* (*sourcenodes asx′*). *ni* ∉ *backward-slice S*;
n′ −*asx′*→∗ *nx*⟧
⟹ ∀ *ni*∈*set* (*sourcenodes asx′*). *x* ∉ *Def ni*›
from ‹*n′* −*ax′*#*asx′*→∗ *nx*› **have** *n′* −[]@*ax′*#*asx′*→∗ *nx* **by** *simp*
hence *targetnode ax′* −*asx′*→∗ *nx* **and** *n′* = *sourcenode ax′*
by(*fastforce dest*:*path-split*)+
from ‹∀ *ni*∈*set* (*sourcenodes* (*ax′*#*asx′*)). *ni* ∉ *backward-slice S*›
have *all*:∀ *ni*∈*set* (*sourcenodes asx′*). *ni* ∉ *backward-slice S*
and *sourcenode ax′* ∉ *backward-slice S*
by(*auto simp*:*sourcenodes-def*)
from *IH*[*OF all* ‹*targetnode ax′* −*asx′*→∗ *nx*›]
have ∀ *ni*∈*set* (*sourcenodes asx′*). *x* ∉ *Def ni* **.**
with ‹∀ *ni*∈*set* (*sourcenodes as″*). *x* ∉ *Def ni*›
have ∀ *ni*∈*set* (*sourcenodes* (*asx′*@*as″*)). *x* ∉ *Def ni*
by(*auto simp*:*sourcenodes-def*)
from ‹*n′* −*ax′*#*asx′*→∗ *nx*› ‹*nx* −*as″*→∗ *m*› **have** *n′* −(*ax′*#*asx′*)@*as″*→∗ *m*
by−(*rule path-Append*)
hence *n′* −*ax′*#*asx′*@*as″*→∗ *m* **by** *simp*
have *x* ∉ *Def* (*sourcenode ax′*)
proof
assume *x* ∈ *Def* (*sourcenode ax′*)
with ‹*x* ∈ *Use m*› ‹∀ *ni*∈*set* (*sourcenodes* (*asx′*@*as″*)). *x* ∉ *Def ni*›

⟨n′ −ax′#asx′@as′′→∗ m⟩ ⟨n′ = sourcenode ax′⟩
have *n′ influences x in m*
by(*auto simp*:*data-dependence-def*)
with ⟨*m* ∈ *backward-slice S*⟩ *dd-closed* **have** *n′* ∈ *backward-slice S*
by(*auto simp*:*dd-closed*)
with ⟨*n′* = *sourcenode ax′*⟩ ⟨*sourcenode ax′* ∉ *backward-slice S*⟩
show *False* **by** *simp*
qed
with ⟨∀ *ni*∈*set* (*sourcenodes* (*asx′*@*as′′*)). *x* ∉ *Def ni*⟩
show *?case* **by**(*simp add*:*sourcenodes-def*)
qed
with ⟨∀ *ni*∈*set* (*sourcenodes as′′*). *x* ∉ *Def ni*⟩
have ∀ *ni*∈*set* (*sourcenodes* (*asx*@*as′′*)). *x* ∉ *Def ni*
by(*auto simp*:*sourcenodes-def*)
from ⟨*n′* −*asx*→∗ *nx*⟩ ⟨*nx* −*as′′*→∗ *m*⟩ **have** *n′* −*asx*@*as′′*→∗ *m* **by**(*rule path-Append*)
with ⟨*m* ∈ *backward-slice S*⟩ ⟨*x* ∈ *Use m*⟩
⟨∀ *ni*∈*set* (*sourcenodes* (*asx*@*as′′*)). *x* ∉ *Def ni*⟩ **show** *x* ∈ *rv S n′* **by** −(*rule rvI*)
qed

lemma *closed-eq-obs-eq-rvs*:
fixes *S* :: *′node set*
assumes *valid-node n* **and** *valid-node n′*
and *obs-eq*:*obs n* (*backward-slice S*) = *obs n′* (*backward-slice S*)
shows *rv S n* = *rv S n′*
proof
show *rv S n* ⊆ *rv S n′*
proof
fix *x* **assume** *x* ∈ *rv S n*
with ⟨*valid-node n*⟩ *obs-eq* **show** *x* ∈ *rv S n′* **by** −(*rule eq-obs-in-rv*)
qed
next
show *rv S n′* ⊆ *rv S n*
proof
fix *x* **assume** *x* ∈ *rv S n′*
with ⟨*valid-node n′*⟩ *obs-eq*[*THEN sym*] **show** *x* ∈ *rv S n* **by** −(*rule eq-obs-in-rv*)
qed
qed

lemma *rv-edge-slice-kinds*:
assumes *valid-edge a* **and** *sourcenode a* = *n* **and** *targetnode a* = *n′′*
and ∀ *V*∈*rv S n*. *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 n′′*. *state-val* (*transfer* (*slice-kind S a*) *s*) *V* =
state-val (*transfer* (*slice-kind S a*) *s′*) *V*
proof
fix *V* **assume** *V* ∈ *rv S n′′*

show *state-val* (*transfer* (*slice-kind S a*) *s*) *V* =
  state-val (*transfer* (*slice-kind S a*) *s'*) *V*
proof(*cases V* ∈ *Def n*)
  case *True*
  show *?thesis*
  proof(*cases sourcenode a* ∈ *backward-slice S*)
    case *True*
    hence *slice-kind S a* = *kind a* **by**(*rule slice-kind-in-slice*)
    with ‹*preds* (*slice-kinds S* (*a#as*)) *s*› **have** *pred* (*kind a*) *s*
      by(*simp add*:*slice-kinds-def*)
    from ‹*slice-kind S a* = *kind a*› ‹*preds* (*slice-kinds S* (*a#asx*)) *s'*›
    have *pred* (*kind a*) *s'*
      by(*simp add*:*slice-kinds-def*)
    from ‹*valid-edge a*› ‹*sourcenode a* = *n*› **have** *n* −[]→∗ *n*
      by(*fastforce intro*:*empty-path*)
    with *True* ‹*sourcenode a* = *n*› **have** ∀ *V* ∈ *Use n*. *V* ∈ *rv S n*
      by(*fastforce intro*:*rvI simp*:*sourcenodes-def*)
    with ‹∀ *V*∈*rv S n*. *state-val s V* = *state-val s' V*› ‹*sourcenode a* = *n*›
    have ∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s' V* **by** *blast*
    from ‹*valid-edge a*› *this* ‹*pred* (*kind a*) *s*› ‹*pred* (*kind a*) *s'*›
    have ∀ *V* ∈ *Def* (*sourcenode a*). *state-val* (*transfer* (*kind a*) *s*) *V* =
      state-val (*transfer* (*kind a*) *s'*) *V*
      by(*rule CFG-edge-transfer-uses-only-Use*)
    with ‹*V* ∈ *Def n*› ‹*sourcenode a* = *n*› ‹*slice-kind S a* = *kind a*›
    show *?thesis* **by** *simp*
  next
    case *False*
    from ‹*V* ∈ *rv S n''*› **obtain** *xs nx* **where** *n''* −*xs*→∗ *nx*
      and *nx* ∈ *backward-slice S* **and** *V* ∈ *Use nx*
      and ∀ *nx'* ∈ *set*(*sourcenodes xs*). *V* ∉ *Def nx'* **by**(*erule rvE*)
    from ‹*valid-edge a*› ‹*sourcenode a* = *n*› ‹*targetnode a* = *n''*›
      ‹*n''* −*xs*→∗ *nx*›
    have *n* −*a#xs*→∗ *nx* **by** −(*rule path.Cons-path*)
    with ‹*V* ∈ *Def n*› ‹*V* ∈ *Use nx*› ‹∀ *nx'* ∈ *set*(*sourcenodes xs*). *V* ∉ *Def nx'*›
    have *n influences V in nx* **by**(*fastforce simp*:*data-dependence-def*)
    with ‹*nx* ∈ *backward-slice S*› **have** *n* ∈ *backward-slice S*
      by(*rule dd-closed*)
    with ‹*sourcenode a* = *n*› *False* **have** *False* **by** *simp*
    thus *?thesis* **by** *simp*
  qed
next
  case *False*
  from ‹*V* ∈ *rv S n''*› **obtain** *xs nx* **where** *n''* −*xs*→∗ *nx*
    and *nx* ∈ *backward-slice S* **and** *V* ∈ *Use nx*
    and ∀ *nx'* ∈ *set*(*sourcenodes xs*). *V* ∉ *Def nx'* **by**(*erule rvE*)
  from ‹*valid-edge a*› ‹*sourcenode a* = *n*› ‹*targetnode a* = *n''*› ‹*n''* −*xs*→∗ *nx*›
  have *n* −*a#xs*→∗ *nx* **by** −(*rule path.Cons-path*)
  from *False* ‹∀ *nx'* ∈ *set*(*sourcenodes xs*). *V* ∉ *Def nx'*› ‹*sourcenode a* = *n*›
  have ∀ *nx'* ∈ *set*(*sourcenodes* (*a#xs*)). *V* ∉ *Def nx'*

by(*simp add*:*sourcenodes-def*)
with ‹*n* −*a*#*xs*→* *nx*› ‹*nx* ∈ *backward-slice S*› ‹*V* ∈ *Use nx*›
have *V* ∈ *rv S n* **by**(*rule rvI*)
show *?thesis*
proof(*cases kind a*)
case (*Predicate Q*)
show *?thesis*
proof(*cases sourcenode a* ∈ *backward-slice S*)
case *True*
with *Predicate* **have** *slice-kind S a* = $(Q)_{\surd}$
by(*simp add*:*slice-kind-in-slice*)
with ‹∀ *V*∈*rv S n*. *state-val s V* = *state-val s' V*› ‹*V* ∈ *rv S n*›
show *?thesis* **by** *simp*
next
case *False*
with *Predicate* **obtain** *Q'* **where** *slice-kind S a* = $(Q')_{\surd}$
by −(*erule kind-Predicate-notin-slice-slice-kind-Predicate*)
with ‹∀ *V*∈*rv S n*. *state-val s V* = *state-val s' V*› ‹*V* ∈ *rv S n*›
show *?thesis* **by** *simp*
qed
next
case (*Update f*)
show *?thesis*
proof(*cases sourcenode a* ∈ *backward-slice S*)
case *True*
hence *slice-kind S a* = *kind a* **by**(*rule slice-kind-in-slice*)
from *Update* **have** *pred* (*kind a*) *s* **by** *simp*
with ‹*valid-edge a*› ‹*sourcenode a* = *n*› ‹*V* ∉ *Def n*›
have *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*
by(*fastforce intro*:*CFG-edge-no-Def-equal*)
from *Update* **have** *pred* (*kind a*) *s'* **by** *simp*
with ‹*valid-edge a*› ‹*sourcenode a* = *n*› ‹*V* ∉ *Def n*›
have *state-val* (*transfer* (*kind a*) *s'*) *V* = *state-val s' V*
by(*fastforce intro*:*CFG-edge-no-Def-equal*)
with ‹∀ *V*∈*rv S n*. *state-val s V* = *state-val s' V*› ‹*V* ∈ *rv S n*›
‹*state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*›
‹*slice-kind S a* = *kind a*›
show *?thesis* **by** *fastforce*
next
case *False*
with *Update* **have** *slice-kind S a* = ⇑*id* **by** −(*rule slice-kind-Upd*)
with ‹∀ *V*∈*rv S n*. *state-val s V* = *state-val s' V*› ‹*V* ∈ *rv S n*›
show *?thesis* **by** *fastforce*
qed
qed
qed
qed

lemma *rv-branching-edges-slice-kinds-False*:
assumes *valid-edge a* **and** *valid-edge ax*
and *sourcenode a* = *n* **and** *sourcenode ax* = *n*
and *targetnode a* = *n''* **and** *targetnode ax* ≠ *n''*
and *preds* (*slice-kinds S* (*a#as*)) *s* **and** *preds* (*slice-kinds S* (*ax#asx*)) *s'*
and ∀ *V*∈*rv S n*. *state-val s V* = *state-val s' V*
shows *False*
proof −
from ‹*valid-edge a*› ‹*valid-edge ax*› ‹*sourcenode a* = *n*› ‹*sourcenode ax* = *n*›
‹*targetnode a* = *n''*› ‹*targetnode ax* ≠ *n''*›
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* = *n*› ‹*sourcenode ax* = *n*›
‹*targetnode a* = *n''*› ‹*targetnode ax* ≠ *n''*›
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* ∈ *backward-slice S*)
case *True*
hence *slice-kind S a* = *kind a* **by**(*rule slice-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* = *n*› ‹*sourcenode ax* = *n*›
have *slice-kind S ax* = *kind ax* **by**(*fastforce simp*:*slice-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' s'* **by** *simp*
from ‹*valid-edge a*› ‹*sourcenode a* = *n*› **have** *n* −[]→* *n*
by(*fastforce intro*:*empty-path*)
with *True* ‹*sourcenode a* = *n*› **have** ∀ *V* ∈ *Use n*. *V* ∈ *rv S n*
by(*fastforce intro*:*rvI simp*:*sourcenodes-def*)
with ‹∀ *V*∈*rv S n*. *state-val s V* = *state-val s' V*› ‹*sourcenode a* = *n*›
have ∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s' V* **by** *blast*
with ‹*valid-edge a*› ‹*pred* (*kind a*) *s*› **have** *pred* (*kind a*) *s'*
by(*rule CFG-edge-Uses-pred-equal*)
with ‹*kind a* = $(Q)_{\surd}$› **have** *Q s'* **by** *simp*
with ‹*Q' 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* = $(P)_{\surd}$›
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**(*fastforce simp*:*slice-kinds-def*)
from ‹*kind ax* = $(Q')_{\surd}$› ‹*slice-kind S ax* = $(P')_{\surd}$›
‹*sourcenode a* = *n*› ‹*sourcenode ax* = *n*› *False*
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**(*fastforce 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

3.3.4 The set *WS*

inductive-set *WS* :: *'node set* ⇒ ((*'node* × *'state*) × (*'node* × *'state*)) *set*
for *S* :: *'node set*
where *WSI*:⟦*obs n* (*backward-slice S*) = *obs n'* (*backward-slice S*);
∀ *V* ∈ *rv S n*. *state-val s V* = *state-val s' V*;
valid-node n; *valid-node n'*⟧
⟹ ((*n*,*s*),(*n'*,*s'*)) ∈ *WS S*

lemma *WSD*:
((*n*,*s*),(*n'*,*s'*)) ∈ *WS S*
⟹ *obs n* (*backward-slice S*) = *obs n'* (*backward-slice S*) ∧
(∀ *V* ∈ *rv S n*. *state-val s V* = *state-val s' V*) ∧
valid-node n ∧ *valid-node n'*
by(*auto elim*:*WS.cases*)

lemma *WS-silent-move*:
assumes ((n_1,s_1),(n_2,s_2)) ∈ *WS S* **and** *S*,*kind* ⊢ (n_1,s_1) $-a\rightarrow_\tau$ (n_1',s_1')
and *obs* n_1' (*backward-slice S*) ≠ {} **shows** ((n_1',s_1'),(n_2,s_2)) ∈ *WS S*
proof −
from ‹((n_1,s_1),(n_2,s_2)) ∈ *WS S*› **have** *valid-node* n_1 **and** *valid-node* n_2
by(*auto dest*:*WSD*)
from ‹*S*,*kind* ⊢ (n_1,s_1) $-a\rightarrow_\tau$ (n_1',s_1')› **have** *sourcenode a* = n_1
and *targetnode a* = n_1' **and** *transfer* (*kind a*) s_1 = s_1'
and n_1 ∉ *backward-slice S* **and** *valid-edge a* **and** *pred* (*kind a*) s_1
by(*auto elim*:*silent-move.cases*)
from ‹*targetnode a* = n_1'› ‹*valid-edge a*› **have** *valid-node* n_1'
by(*auto simp*:*valid-node-def*)
have (∃ *m*. *obs* n_1' (*backward-slice S*) = {*m*}) ∨ *obs* n_1' (*backward-slice S*) = {}
by(*rule obs-singleton-disj*)
with ‹*obs* n_1' (*backward-slice S*) ≠ {}› **obtain** *n*
where *obs* n_1' (*backward-slice S*) = {*n*} **by** *fastforce*
hence *n* ∈ *obs* n_1' (*backward-slice S*) **by** *auto*

then obtain *as* **where** $n_1{}' -as\rightarrow* n$
and $\forall nx \in set(sourcenodes\ as).\ nx \notin (backward\text{-}slice\ S)$
and $n \in (backward\text{-}slice\ S)$ **by**(*erule obsE*)
from ‹$n_1{}' -as\rightarrow* n$› ‹*valid-edge a*› ‹$sourcenode\ a = n_1$› ‹$targetnode\ a = n_1{}'$›
have $n_1 -a\#as\rightarrow* n$ **by**(*rule Cons-path*)
moreover
from ‹$\forall nx \in set(sourcenodes\ as).\ nx \notin (backward\text{-}slice\ S)$› ‹$sourcenode\ a = n_1$›
‹$n_1 \notin backward\text{-}slice\ S$›
have $\forall nx \in set(sourcenodes\ (a\#as)).\ nx \notin (backward\text{-}slice\ S)$
by(*simp add:sourcenodes-def*)
ultimately have $n \in obs\ n_1\ (backward\text{-}slice\ S)$ **using** ‹$n \in (backward\text{-}slice\ S)$›
by(*rule obs-elem*)
hence $obs\ n_1\ (backward\text{-}slice\ S) = \{n\}$ **by**(*rule obs-singleton-element*)
with ‹$obs\ n_1{}'\ (backward\text{-}slice\ S) = \{n\}$›
have $obs\ n_1\ (backward\text{-}slice\ S) = obs\ n_1{}'\ (backward\text{-}slice\ S)$
by *simp*
with ‹$valid\text{-}node\ n_1$› ‹$valid\text{-}node\ n_1{}'$› **have** $rv\ S\ n_1 = rv\ S\ n_1{}'$
by(*rule closed-eq-obs-eq-rvs*)
from ‹$n \in obs\ n_1\ (backward\text{-}slice\ S)$› ‹$((n_1,s_1),(n_2,s_2)) \in WS\ S$›
have $obs\ n_1\ (backward\text{-}slice\ S) = obs\ n_2\ (backward\text{-}slice\ S)$
and $\forall V \in rv\ S\ n_1.\ state\text{-}val\ s_1\ V = state\text{-}val\ s_2\ V$
by(*fastforce dest:WSD*)+
from ‹$obs\ n_1\ (backward\text{-}slice\ S) = obs\ n_2\ (backward\text{-}slice\ S)$›
‹$obs\ n_1\ (backward\text{-}slice\ S) = \{n\}$› ‹$obs\ n_1{}'\ (backward\text{-}slice\ S) = \{n\}$›
have $obs\ n_1{}'\ (backward\text{-}slice\ S) = obs\ n_2\ (backward\text{-}slice\ S)$ **by** *simp*
have $\forall V \in rv\ S\ n_1{}'.\ state\text{-}val\ s_1{}'\ V = state\text{-}val\ s_2\ V$
proof
fix V **assume** $V \in rv\ S\ n_1{}'$
with ‹$rv\ S\ n_1 = rv\ S\ n_1{}'$› **have** $V \in rv\ S\ n_1$ **by** *simp*
then obtain $as\ n'$ **where** $n_1 -as\rightarrow* n'$ **and** $n' \in (backward\text{-}slice\ S)$
and $V \in Use\ n'$ **and** $\forall nx \in set(sourcenodes\ as).\ V \notin Def\ nx$
by(*erule rvE*)
with ‹$n_1 \notin backward\text{-}slice\ S$› **have** $V \notin Def\ n_1$
by(*auto elim:path.cases simp:sourcenodes-def*)
with ‹*valid-edge a*› ‹$sourcenode\ a = n_1$› ‹$pred\ (kind\ a)\ s_1$›
have $state\text{-}val\ (transfer\ (kind\ a)\ s_1)\ V = state\text{-}val\ s_1\ V$
by(*fastforce intro:CFG-edge-no-Def-equal*)
with ‹$transfer\ (kind\ a)\ s_1 = s_1{}'$› **have** $state\text{-}val\ s_1{}'\ V = state\text{-}val\ s_1\ V$ **by**
simp
from ‹$V \in rv\ S\ n_1$› ‹$\forall V \in rv\ S\ n_1.\ state\text{-}val\ s_1\ V = state\text{-}val\ s_2\ V$›
have $state\text{-}val\ s_1\ V = state\text{-}val\ s_2\ V$ **by** *simp*
with ‹$state\text{-}val\ s_1{}'\ V = state\text{-}val\ s_1\ V$›
show $state\text{-}val\ s_1{}'\ V = state\text{-}val\ s_2\ V$ **by** *simp*
qed
with ‹$obs\ n_1{}'\ (backward\text{-}slice\ S) = obs\ n_2\ (backward\text{-}slice\ S)$›
‹$valid\text{-}node\ n_1{}'$› ‹$valid\text{-}node\ n_2$› **show** *?thesis* **by**(*fastforce intro:WSI*)
qed

lemma *WS-silent-moves*:
$[\![S,f \vdash (n_1,s_1) =as\Rightarrow_\tau (n_1{}',s_1{}'); ((n_1,s_1),(n_2,s_2)) \in WS\ S; f = kind;$
$obs\ n_1{}'\ (backward\text{-}slice\ S) \neq \{\}]\!]$
$\Longrightarrow ((n_1{}',s_1{}'),(n_2,s_2)) \in WS\ S$
proof(*induct rule*:*silent-moves.induct*)
case *silent-moves-Nil* **thus** *?case* **by** *simp*
next
case (*silent-moves-Cons S f n s a n′ s′ as n″ s″*)
note $IH = \langle[\![((n',s'),(n_2,s_2)) \in WS\ S; f = kind; obs\ n''\ (backward\text{-}slice\ S) \neq \{\}]\!]$
$\Longrightarrow ((n'',s''),(n_2,s_2)) \in WS\ S\rangle$
from $\langle S,f \vdash (n',s') =as\Rightarrow_\tau (n'',s'')\rangle$ $\langle obs\ n''\ (backward\text{-}slice\ S) \neq \{\}\rangle$
have $obs\ n'\ (backward\text{-}slice\ S) \neq \{\}$ **by**(*fastforce dest*:*silent-moves-obs-slice*)
with $\langle((n,s),(n_2,s_2)) \in WS\ S\rangle$ $\langle S,f \vdash (n,s)\ {-a}{\rightarrow_\tau}\ (n',s')\rangle$ $\langle f = kind\rangle$
have $((n',s'),(n_2,s_2)) \in WS\ S$ **by** −(*rule WS-silent-move*,*simp*+)
from *IH*[*OF this* $\langle f = kind\rangle$ $\langle obs\ n''\ (backward\text{-}slice\ S) \neq \{\}\rangle$]
show *?case* **.**
qed

lemma *WS-observable-move*:
assumes $((n_1,s_1),(n_2,s_2)) \in WS\ S$ **and** $S,kind \vdash (n_1,s_1)\ {-a}{\rightarrow}\ (n_1{}',s_1{}')$
obtains *as* **where** $((n_1{}',s_1{}'),(n_1{}',transfer\ (slice\text{-}kind\ S\ a)\ s_2)) \in WS\ S$
and $S,slice\text{-}kind\ S \vdash (n_2,s_2) =as@[a]\Rightarrow (n_1{}',transfer\ (slice\text{-}kind\ S\ a)\ s_2)$
proof(*atomize-elim*)
from $\langle((n_1,s_1),(n_2,s_2)) \in WS\ S\rangle$ **have** *valid-node* n_1 **by**(*auto dest*:*WSD*)
from $\langle S,kind \vdash (n_1,s_1)\ {-a}{\rightarrow}\ (n_1{}',s_1{}')\rangle$ **have** [*simp*]:$n_1 = sourcenode\ a$
and [*simp*]:$n_1{}' = targetnode\ a$ **and** *pred* (*kind a*) s_1
and *transfer* (*kind a*) $s_1 = s_1{}'$ **and** $n_1 \in (backward\text{-}slice\ S)$
and *valid-edge a* **and** *pred* (*kind a*) s_1
by(*auto elim*:*observable-move.cases*)
from ‹*valid-edge a*› **have** *valid-node* $n_1{}'$ **by**(*auto simp*:*valid-node-def*)
from ‹*valid-node* n_1› $\langle n_1 \in (backward\text{-}slice\ S)\rangle$
have $obs\ n_1\ (backward\text{-}slice\ S) = \{n_1\}$ **by**(*rule n-in-obs*)
with $\langle((n_1,s_1),(n_2,s_2)) \in WS\ S\rangle$ **have** $obs\ n_2\ (backward\text{-}slice\ S) = \{n_1\}$
and $\forall V \in rv\ S\ n_1.\ state\text{-}val\ s_1\ V = state\text{-}val\ s_2\ V$ **by**(*auto dest*:*WSD*)
from ‹*valid-node* n_1› **have** $n_1\ {-[]}{\rightarrow*}\ n_1$ **by**(*rule empty-path*)
with $\langle n_1 \in (backward\text{-}slice\ S)\rangle$ **have** $\forall V \in Use\ n_1.\ V \in rv\ S\ n_1$
by(*fastforce intro*:*rvI simp*:*sourcenodes-def*)
with $\langle\forall V \in rv\ S\ n_1.\ state\text{-}val\ s_1\ V = state\text{-}val\ s_2\ V\rangle$
have $\forall V \in Use\ n_1.\ state\text{-}val\ s_1\ V = state\text{-}val\ s_2\ V$ **by** *blast*
with ‹*valid-edge a*› ‹*pred* (*kind a*) s_1› **have** *pred* (*kind a*) s_2
by(*fastforce intro*:*CFG-edge-Uses-pred-equal*)
with $\langle n_1 \in (backward\text{-}slice\ S)\rangle$ **have** *pred* (*slice-kind S a*) s_2
by(*simp add*:*slice-kind-in-slice*)
from $\langle n_1 \in (backward\text{-}slice\ S)\rangle$ **obtain** $s_2{}'$
where *transfer* (*slice-kind S a*) $s_2 = s_2{}'$
by(*simp add*:*slice-kind-in-slice*)
with ‹*pred* (*slice-kind S a*) s_2› $\langle n_1 \in (backward\text{-}slice\ S)\rangle$ ‹*valid-edge a*›

```
have S,slice-kind S ⊢ (n_1,s_2) −a→ (n_1',s_2')
  by(fastforce intro:observable-moveI)
from ‹obs n_2 (backward-slice S) = {n_1}›
obtain as where S,slice-kind S ⊢ (n_2,s_2) =as⇒_τ (n_1,s_2)
  by(erule obs-silent-moves)
with ‹S,slice-kind S ⊢ (n_1,s_2) −a→ (n_1',s_2')›
have S,slice-kind S ⊢ (n_2,s_2) =as@[a]⇒ (n_1',s_2')
  by −(rule observable-moves-snoc)
have ∀ V ∈ rv S n_1'. state-val s_1' V = state-val s_2' V
proof
  fix V assume rv:V ∈ rv S n_1'
  show state-val s_1' V = state-val s_2' V
  proof(cases V ∈ Def n_1)
    case True
    thus ?thesis
    proof(cases kind a)
      case (Update f)
      with ‹transfer (kind a) s_1 = s_1'› have s_1' = f s_1 by simp
      from Update[THEN sym] ‹n_1 ∈ (backward-slice S)›
      have slice-kind S a = ⇑f
        by(fastforce intro:slice-kind-in-slice)
      with ‹transfer (slice-kind S a) s_2 = s_2'› have s_2' = f s_2 by simp
      from ‹valid-edge a› ‹∀ V ∈ Use n_1. state-val s_1 V = state-val s_2 V›
        True Update ‹s_1' = f s_1› ‹s_2' = f s_2› show ?thesis
        by(fastforce dest:CFG-edge-transfer-uses-only-Use)
    next
      case (Predicate Q)
      with ‹transfer (kind a) s_1 = s_1'› have s_1' = s_1 by simp
      from Predicate[THEN sym] ‹n_1 ∈ (backward-slice S)›
      have slice-kind S a = (Q)_√
        by(fastforce intro:slice-kind-in-slice)
      with ‹transfer (slice-kind S a) s_2 = s_2'› have s_2' = s_2 by simp
      with ‹valid-edge a› ‹∀ V ∈ Use n_1. state-val s_1 V = state-val s_2 V›
        True Predicate ‹s_1' = s_1› ‹pred (kind a) s_1› ‹pred (kind a) s_2›
      show ?thesis by(auto dest:CFG-edge-transfer-uses-only-Use)
    qed
  next
    case False
    with ‹valid-edge a› ‹transfer (kind a) s_1 = s_1'›[THEN sym]
      ‹pred (kind a) s_1› ‹pred (kind a) s_2›
    have state-val s_1' V = state-val s_1 V
      by(fastforce intro:CFG-edge-no-Def-equal)
    have state-val s_2' V = state-val s_2 V
    proof(cases kind a)
      case (Update f)
      with ‹n_1 ∈ (backward-slice S)› have slice-kind S a = kind a
        by(fastforce intro:slice-kind-in-slice)
      with ‹valid-edge a› ‹transfer (slice-kind S a) s_2 = s_2'›[THEN sym]
        False ‹pred (kind a) s_2›
```

show *?thesis* **by**(*fastforce intro*:*CFG-edge-no-Def-equal*)
next
case (*Predicate Q*)
with ‹*transfer* (*slice-kind S a*) s_2 = s_2'› **have** s_2 = s_2'
by(*cases slice-kind S a*,
auto split:*if-split-asm simp*:*slice-kind-def Let-def*)
thus *?thesis* **by** *simp*
qed
from *rv* **obtain** *as′ nx* **where** n_1' −*as′*→∗ *nx*
and *nx* ∈ (*backward-slice S*)
and *V* ∈ *Use nx* **and** ∀ *nx* ∈ *set*(*sourcenodes as′*). *V* ∉ *Def nx*
by(*erule rvE*)
from ‹∀ *nx* ∈ *set*(*sourcenodes as′*). *V* ∉ *Def nx*› *False*
have ∀ *nx* ∈ *set*(*sourcenodes* (*a*#*as′*)). *V* ∉ *Def nx*
by(*auto simp*:*sourcenodes-def*)
from ‹*valid-edge a*› ‹n_1' −*as′*→∗ *nx*› **have** n_1 −*a*#*as′*→∗ *nx*
by(*fastforce intro*:*Cons-path*)
with ‹*nx* ∈ (*backward-slice S*)› ‹*V* ∈ *Use nx*›
‹∀ *nx* ∈ *set*(*sourcenodes* (*a*#*as′*)). *V* ∉ *Def nx*›
have *V* ∈ *rv S* n_1 **by** −(*rule rvI*)
with ‹∀ *V* ∈ *rv S* n_1. *state-val* s_1 *V* = *state-val* s_2 *V*›
‹*state-val* s_1' *V* = *state-val* s_1 *V*› ‹*state-val* s_2' *V* = *state-val* s_2 *V*›
show *?thesis* **by** *fastforce*
qed
qed
with ‹*valid-node* n_1'› **have** ((n_1',s_1'),(n_1',s_2')) ∈ *WS S* **by**(*fastforce intro*:*WSI*)
with ‹*S*,*slice-kind S* ⊢ (n_2,s_2) =*as*@[*a*]⇒ (n_1',s_2')›
‹*transfer* (*slice-kind S a*) s_2 = s_2'›
show ∃ *as*. ((n_1',s_1'),(n_1',*transfer* (*slice-kind S a*) s_2)) ∈ *WS S* ∧
S,*slice-kind S* ⊢ (n_2,s_2) =*as*@[*a*]⇒ (n_1',*transfer* (*slice-kind S a*) s_2)
by *blast*
qed

definition *is-weak-sim* ::
((′*node* × ′*state*) × (′*node* × ′*state*)) *set* ⇒ ′*node set* ⇒ *bool*
where *is-weak-sim R S* ≡
∀ n_1 s_1 n_2 s_2 n_1' s_1' *as*. ((n_1,s_1),(n_2,s_2)) ∈ *R* ∧ *S*,*kind* ⊢ (n_1,s_1) =*as*⇒ (n_1',s_1')
⟶ (∃ n_2' s_2' *as′*. ((n_1',s_1'),(n_2',s_2')) ∈ *R* ∧
S,*slice-kind S* ⊢ (n_2,s_2) =*as′*⇒ (n_2',s_2'))

lemma *WS-weak-sim*:
assumes ((n_1,s_1),(n_2,s_2)) ∈ *WS S*
and *S*,*kind* ⊢ (n_1,s_1) =*as*⇒ (n_1',s_1')
shows ((n_1',s_1'),(n_1',*transfer* (*slice-kind S* (*last as*)) s_2)) ∈ *WS S* ∧
(∃ *as′*. *S*,*slice-kind S* ⊢ (n_2,s_2) =*as′*@[*last as*]⇒
(n_1',*transfer* (*slice-kind S* (*last as*)) s_2))

proof −
  from ‹S,*kind* ⊢ (n_1,s_1) =*as*⇒ (n_1',s_1')› **obtain** a' *as'* n' s'
    where S,*kind* ⊢ (n_1,s_1) =*as'*⇒$_\tau$ (n',s')
    and S,*kind* ⊢ (n',s') −a'→ (n_1',s_1') **and** *as* = *as'*@[a']
    by(*fastforce elim*:*observable-moves.cases*)
  from ‹S,*kind* ⊢ (n',s') −a'→ (n_1',s_1')› **have** *obs* n' (*backward-slice* S) = $\{n'\}$
    by(*fastforce elim*:*observable-move.cases intro*!:*n-in-obs*)
  hence *obs* n' (*backward-slice* S) ≠ {} **by** *fast*
  with ‹S,*kind* ⊢ (n_1,s_1) =*as'*⇒$_\tau$ (n',s')› ‹$((n_1,s_1),(n_2,s_2))$ ∈ *WS S*›
  have $((n',s'),(n_2,s_2))$ ∈ *WS S*
    by −(*rule WS-silent-moves*,*simp*+)
  with ‹S,*kind* ⊢ (n',s') −a'→ (n_1',s_1')› **obtain** *asx*
    where $((n_1',s_1'),(n_1'$,*transfer* (*slice-kind S* a') $s_2))$ ∈ *WS S*
    and S,*slice-kind S* ⊢ (n_2,s_2) =*asx*@[a']⇒
    $(n_1'$,*transfer* (*slice-kind S* a') $s_2)$
    by(*fastforce elim*:*WS-observable-move*)
  with ‹*as* = *as'*@[a']› **show**
    $((n_1',s_1'),(n_1'$,*transfer* (*slice-kind S* (*last as*)) $s_2))$ ∈ *WS S* ∧
    (∃ *as'*. S,*slice-kind S* ⊢ (n_2,s_2) =*as'*@[*last as*]⇒
        $(n_1'$,*transfer* (*slice-kind S* (*last as*)) $s_2)$) **by** *simp blast*
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 dest*:*WS-weak-sim simp*:*is-weak-sim-def*)

3.3.5 n −*as*→∗ n' and transitive closure of S,f ⊢ (n,s) =*as*⇒$_\tau$ (n',s')

inductive *trans-observable-moves* ::
  'node set ⇒ (*'edge* ⇒ *'state edge-kind*) ⇒ *'node* ⇒ *'state* ⇒ *'edge list* ⇒
  'node ⇒ *'state* ⇒ *bool* (‹-,- ⊢ '(-,-') =-⇒∗ '(-,-')› [*51*,*50*,*0*,*0*,*50*,*0*,*0*] *51*)

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

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

definition *slice-edges* :: *'node set* ⇒ *'edge list* ⇒ *'edge list*
  where *slice-edges S as* ≡ [a ← *as*. *sourcenode a* ∈ *backward-slice S*]

lemma *silent-moves-no-slice-edges*:
  S,f ⊢ (n,s) =*as*⇒$_\tau$ (n',s') ⟹ *slice-edges S as* = []
by(*induct rule*:*silent-moves.induct*,*auto elim*:*silent-move.cases simp*:*slice-edges-def*)

lemma *observable-moves-last-slice-edges*:
  $S,f \vdash (n,s) =as\Rightarrow (n',s') \Longrightarrow$ *slice-edges S as* = [*last as*]
by(*induct rule*:*observable-moves.induct*,
  fastforce dest:*silent-moves-no-slice-edges elim*:*observable-move.cases*
    simp:*slice-edges-def*)

lemma *slice-edges-no-nodes-in-slice*:
  slice-edges S as = []
  $\Longrightarrow \forall$ *nx* $\in$ *set*(*sourcenodes as*). *nx* $\notin$ (*backward-slice S*)
proof(*induct as*)
  case *Nil* **thus** *?case* **by**(*simp add*:*slice-edges-def sourcenodes-def*)
next
  case (*Cons a' as'*)
  note *IH* = ‹*slice-edges S as'* = [] $\Longrightarrow$
    $\forall$ *nx*$\in$*set* (*sourcenodes as'*). *nx* $\notin$ *backward-slice S*›
  from ‹*slice-edges S* (*a'*#*as'*) = []› **have** *slice-edges S as'* = []
    and *sourcenode a'* $\notin$ *backward-slice S*
    by(*auto simp*:*slice-edges-def split*:*if-split-asm*)
  from *IH*[*OF* ‹*slice-edges S as'* = []›] ‹*sourcenode a'* $\notin$ *backward-slice S*›
  show *?case* **by**(*simp add*:*sourcenodes-def*)
qed

lemma *sliced-path-determ*:
  ⟦*n* −*as*→* *n'*; *n* −*as'*→* *n'*; *slice-edges S as* = *slice-edges S as'*;
    preds (*slice-kinds S as*) *s*; *preds* (*slice-kinds S as'*) *s'*; *n'* $\in$ *S*;
    $\forall$ *V* $\in$ *rv S n*. *state-val s V* = *state-val s' V*⟧ $\Longrightarrow$ *as* = *as'*
proof(*induct arbitrary*:*as' s s' rule*:*path.induct*)
  case (*empty-path n*)
  from ‹*slice-edges S* [] = *slice-edges S as'*›
  have $\forall$ *nx* $\in$ *set*(*sourcenodes as'*). *nx* $\notin$ (*backward-slice S*)
    by(*fastforce intro*!:*slice-edges-no-nodes-in-slice simp*:*slice-edges-def*)
  with ‹*n* −*as'*→* *n*› **show** *?case*
  proof(*induct nx*≡*n as' nx'*≡*n rule*:*path.induct*)
    case (*Cons-path n'' as a*)
    from ‹*valid-node n*› ‹*n* $\in$ *S*› **have** *n* $\in$ *backward-slice S* **by**(*rule refl*)
    with ‹$\forall$ *nx*$\in$*set* (*sourcenodes* (*a* # *as*)). *nx* $\notin$ *backward-slice S*›
      ‹*sourcenode a* = *n*›
    have *False* **by**(*simp add*:*sourcenodes-def*)
    thus *?case* **by** *simp*
  qed *simp*
next
  case (*Cons-path n'' as n' a n*)
  note *IH* = ‹$\bigwedge$*as' s s'*. ⟦*n''* −*as'*→* *n'*; *slice-edges S as* = *slice-edges S as'*;
    preds (*slice-kinds S as*) *s*; *preds* (*slice-kinds S as'*) *s'*; *n'* $\in$ *S*;

∀ V∈rv S n''. state-val s V = state-val s' V⟧ ⟹ as = as'›
show *?case*
proof(*cases as'*)
case *Nil*
with ‹n −as'→∗ n'› **have** n = n' **by** *fastforce*
from *Nil* ‹slice-edges S (a#as) = slice-edges S as'› ‹sourcenode a = n›
have n ∉ backward-slice S **by**(*fastforce simp:slice-edges-def*)
from ‹valid-edge a› ‹sourcenode a = n› ‹n = n'› ‹n' ∈ S›
have n ∈ backward-slice S **by**(*fastforce intro:refl*)
with ‹n = n'› ‹n ∉ backward-slice S› **have** *False* **by** *simp*
thus *?thesis* **by** *simp*
next
case (*Cons ax asx*)
with ‹n −as'→∗ n'› **have** n = sourcenode ax **and** valid-edge ax
and targetnode ax −asx→∗ n' **by**(*auto elim:path-split-Cons*)
show *?thesis*
proof(*cases targetnode ax = n''*)
case *True*
with ‹targetnode ax −asx→∗ n'› **have** n'' −asx→∗ n' **by** *simp*
from ‹valid-edge ax› ‹valid-edge a› ‹n = sourcenode ax› ‹sourcenode a = n›
True ‹targetnode a = n''› **have** ax = a **by**(*fastforce intro:edge-det*)
from ‹slice-edges S (a#as) = slice-edges S as'› *Cons*
‹n = sourcenode ax› ‹sourcenode a = n›
have slice-edges S as = slice-edges S asx
by(*cases n ∈ backward-slice S*)(*auto simp:slice-edges-def*)
from ‹preds (slice-kinds S (a#as)) s›
have *preds1*:preds (slice-kinds S as) (transfer (slice-kind S a) s)
by(*simp add:slice-kinds-def*)
from ‹preds (slice-kinds S as') s'› *Cons* ‹ax = a›
have *preds2*:preds (slice-kinds S asx) (transfer (slice-kind S a) s')
by(*simp add:slice-kinds-def*)
from ‹valid-edge a› ‹sourcenode a = n› ‹targetnode a = n''›
‹preds (slice-kinds S (a#as)) s› ‹preds (slice-kinds S as') s'›
‹ax = a› *Cons* ‹∀ V∈rv S n. state-val s V = state-val s' V›
have ∀ V∈rv S n''. state-val (transfer (slice-kind S a) s) V =
state-val (transfer (slice-kind S a) s') V
by −(*rule rv-edge-slice-kinds,auto*)
from *IH*[*OF* ‹n'' −asx→∗ n'› ‹slice-edges S as = slice-edges S asx›
preds1 preds2 ‹n' ∈ S› *this*] *Cons* ‹ax = a› **show** *?thesis* **by** *simp*
next
case *False*
with ‹valid-edge a› ‹valid-edge ax› ‹sourcenode a = n› ‹n = sourcenode ax›
‹targetnode a = n''› ‹preds (slice-kinds S (a#as)) s›
‹preds (slice-kinds S as') s'› *Cons*
‹∀ V∈rv S n. state-val s V = state-val s' V›
have *False* **by** −(*erule rv-branching-edges-slice-kinds-False,auto*)
thus *?thesis* **by** *simp*
qed
qed

qed

lemma *path-trans-observable-moves*:
  assumes n −*as*→∗ n' **and** *preds* (*kinds as*) *s* **and** *transfers* (*kinds as*) *s* = s'
  obtains n'' s'' as' as'' **where** *S*,*kind* ⊢ (*n*,*s*) =*slice-edges S as*⇒∗ (n'',s'')
  and *S*,*kind* ⊢ (n'',s'') =as'⇒$_\tau$ (n',s')
  and *slice-edges S as* = *slice-edges S* as'' **and** *n* −as''@as'→∗ n'
proof(*atomize-elim*)
  from ‹*n* −*as*→∗ n'› ‹*preds* (*kinds as*) *s*› ‹*transfers* (*kinds as*) *s* = s'›
  show ∃ n'' s'' as' as''.
    S,*kind* ⊢ (*n*,*s*) =*slice-edges S as*⇒∗ (n'',s'') ∧
    S,*kind* ⊢ (n'',s'') =as'⇒$_\tau$ (n',s') ∧ *slice-edges S as* = *slice-edges S* as'' ∧
    n −as''@as'→∗ n'
  proof(*induct arbitrary*:*s rule*:*path.induct*)
    case (*empty-path n*)
    from ‹*transfers* (*kinds* []) *s* = s'› **have** *s* = s' **by**(*simp add*:*kinds-def*)
    have *S*,*kind* ⊢ (*n*,*s*) =[]⇒∗ (*n*,*s*) **by**(*rule tom-Nil*)
    have *S*,*kind* ⊢ (*n*,*s*) =[]⇒$_\tau$ (*n*,*s*) **by**(*rule silent-moves-Nil*)
    with ‹*S*,*kind* ⊢ (*n*,*s*) =[]⇒∗ (*n*,*s*)› ‹*s* = s'› ‹*valid-node n*›
    show *?case*
      apply(*rule-tac x*=*n* **in** *exI*)
      apply(*rule-tac x*=*s* **in** *exI*)
      apply(*rule-tac x*=[] **in** *exI*)
      apply(*rule-tac x*=[] **in** *exI*)
      by(*fastforce intro*:*path.empty-path simp*:*slice-edges-def*)
  next
    case (*Cons-path* n'' *as* n' *a n*)
    note *IH* = ‹⋀*s*. ⟦*preds* (*kinds as*) *s*; *transfers* (*kinds as*) *s* = s'⟧
      ⟹ ∃ *nx* s'' as' as''. *S*,*kind* ⊢ (n'',*s*) =*slice-edges S as*⇒∗ (*nx*,s'') ∧
          S,*kind* ⊢ (*nx*,s'') =as'⇒$_\tau$ (n',s') ∧
          slice-edges S as = *slice-edges S* as'' ∧ n'' −as''@as'→∗ n'›
    from ‹*preds* (*kinds* (*a*#*as*)) *s*› ‹*transfers* (*kinds* (*a*#*as*)) *s* = s'›
    have *preds* (*kinds as*) (*transfer* (*kind a*) *s*)
      transfers (*kinds as*) (*transfer* (*kind a*) *s*) = s' **by**(*simp-all add*:*kinds-def*)
    from *IH*[*OF this*] **obtain** *nx sx asx* asx'
      where *S*,*kind* ⊢ (n'',*transfer* (*kind a*) *s*) =*slice-edges S as*⇒∗ (*nx*,*sx*)
      and *S*,*kind* ⊢ (*nx*,*sx*) =*asx*⇒$_\tau$ (n',s')
      and *slice-edges S as* = *slice-edges S* asx'
      and n'' −asx'@*asx*→∗ n'
      by *clarsimp*
    from ‹*preds* (*kinds* (*a*#*as*)) *s*› **have** *pred* (*kind a*) *s* **by**(*simp add*:*kinds-def*)
    show *?case*
    proof(*cases n* ∈ *backward-slice S*)
      case *True*
      with ‹*valid-edge a*› ‹*sourcenode a* = *n*› ‹*targetnode a* = n''› ‹*pred* (*kind a*) *s*›
      have *S*,*kind* ⊢ (*n*,*s*) −*a*→ (n'',*transfer* (*kind a*) *s*)
        by(*fastforce intro*:*observable-moveI*)

hence *S*,*kind* ⊢ (*n*,*s*) =[]@[*a*]⇒ (*n''*,*transfer* (*kind a*) *s*)
by(*fastforce intro*:*observable-moves-snoc silent-moves-Nil*)
with ‹*S*,*kind* ⊢ (*n''*,*transfer* (*kind a*) *s*) =*slice-edges S as*⇒∗ (*nx*,*sx*)›
have *S*,*kind* ⊢ (*n*,*s*) =*a*#*slice-edges S as*⇒∗ (*nx*,*sx*)
by(*fastforce dest*:*tom-Cons*)
with ‹*S*,*kind* ⊢ (*nx*,*sx*) =*asx*⇒$_\tau$ (*n'*,*s'*)›
‹*slice-edges S as* = *slice-edges S asx'*› ‹*n''* −*asx'*@*asx*→∗ *n'*›
‹*sourcenode a* = *n*› ‹*valid-edge a*› ‹*targetnode a* = *n''*› *True*
show *?thesis*
apply(*rule-tac x*=*nx* **in** *exI*)
apply(*rule-tac x*=*sx* **in** *exI*)
apply(*rule-tac x*=*asx* **in** *exI*)
apply(*rule-tac x*=*a*#*asx'* **in** *exI*)
by(*auto intro*:*path.Cons-path simp*:*slice-edges-def*)
next
case *False*
with ‹*valid-edge a*› ‹*sourcenode a* = *n*› ‹*targetnode a* = *n''*› ‹*pred* (*kind a*) *s*›
have *S*,*kind* ⊢ (*n*,*s*) −*a*→$_\tau$ (*n''*,*transfer* (*kind a*) *s*)
by(*fastforce intro*:*silent-moveI*)
from ‹*S*,*kind* ⊢ (*n''*,*transfer* (*kind a*) *s*) =*slice-edges S as*⇒∗ (*nx*,*sx*)›
obtain *f s'' asx''* **where** *S*,*f* ⊢ (*n''*,*s''*) =*asx''*⇒∗ (*nx*,*sx*)
and *f* = *kind* **and** *s''* = *transfer* (*kind a*) *s*
and *asx''* = *slice-edges S as* **by** *simp*
from ‹*S*,*f* ⊢ (*n''*,*s''*) =*asx''*⇒∗ (*nx*,*sx*)› ‹*f* = *kind*›
‹*asx''* = *slice-edges S as*› ‹*s''* = *transfer* (*kind a*) *s*›
‹*S*,*kind* ⊢ (*n*,*s*) −*a*→$_\tau$ (*n''*,*transfer* (*kind a*) *s*)›
‹*S*,*kind* ⊢ (*nx*,*sx*) =*asx*⇒$_\tau$ (*n'*,*s'*)› ‹*slice-edges S as* = *slice-edges S asx'*›
‹*n''* −*asx'*@*asx*→∗ *n'*› *False*
show *?thesis*
proof(*induct rule*:*trans-observable-moves.induct*)
case (*tom-Nil S f ni si*)
have *S*,*kind* ⊢ (*n*,*s*) =[]⇒∗ (*n*,*s*) **by**(*rule trans-observable-moves.tom-Nil*)
from ‹*S*,*kind* ⊢ (*ni*,*si*) =*asx*⇒$_\tau$ (*n'*,*s'*)›
‹*S*,*kind* ⊢ (*n*,*s*) −*a*→$_\tau$ (*ni*,*transfer* (*kind a*) *s*)›
‹*si* = *transfer* (*kind a*) *s*›
have *S*,*kind* ⊢ (*n*,*s*) =*a*#*asx*⇒$_\tau$ (*n'*,*s'*)
by(*fastforce intro*:*silent-moves-Cons*)
with ‹*valid-edge a*› ‹*sourcenode a* = *n*›
have *n* −*a*#*asx*→∗ *n'* **by**(*fastforce dest*:*silent-moves-preds-transfers-path*)
with ‹*sourcenode a* = *n*› ‹*valid-edge a*› ‹*targetnode a* = *n''*›
‹[] = *slice-edges S as*› ‹*n* ∉ *backward-slice S*›
‹*S*,*kind* ⊢ (*n*,*s*) =*a*#*asx*⇒$_\tau$ (*n'*,*s'*)›
show *?case*
apply(*rule-tac x*=*n* **in** *exI*)
apply(*rule-tac x*=*s* **in** *exI*)
apply(*rule-tac x*=*a*#*asx* **in** *exI*)
apply(*rule-tac x*=[] **in** *exI*)
by(*fastforce simp*:*slice-edges-def intro*:*trans-observable-moves.tom-Nil*)
next

case (*tom-Cons S f ni si asi ni′ si′ asi′ n′′ s′′*)
from ‹*S*,*f* ⊢ (*ni*,*si*) =*asi*⇒ (*ni′*,*si′*)› **have** *asi* ≠ []
by(*fastforce dest*:*observable-move-notempty*)
from ‹*S*,*kind* ⊢ (*n*,*s*) −*a*→$_\tau$ (*ni*,*transfer* (*kind a*) *s*)›
have *valid-edge a* **and** *sourcenode a* = *n* **and** *targetnode a* = *ni*
by(*auto elim*:*silent-move.cases*)
from ‹*S*,*kind* ⊢ (*n*,*s*) −*a*→$_\tau$ (*ni*,*transfer* (*kind a*) *s*)› ‹*f* = *kind*›
‹*si* = *transfer* (*kind a*) *s*› ‹*S*,*f* ⊢ (*ni*,*si*) =*asi*⇒ (*ni′*,*si′*)›
have *S*,*f* ⊢ (*n*,*s*) =*a*#*asi*⇒ (*ni′*,*si′*)
by(*fastforce intro*:*silent-move-observable-moves*)
with ‹*S*,*f* ⊢ (*ni′*,*si′*) =*asi′*⇒∗ (*n′′*,*s′′*)›
have *S*,*f* ⊢ (*n*,*s*) =(*last* (*a*#*asi*))#*asi′*⇒∗ (*n′′*,*s′′*)
by −(*rule trans-observable-moves.tom-Cons*)
with ‹*f* = *kind*› ‹*last asi* # *asi′* = *slice-edges S as*› ‹*n* ∉ *backward-slice S*›
‹*S*,*kind* ⊢ (*n′′*,*s′′*) =*asx*⇒$_\tau$ (*n′*,*s′*)› ‹*sourcenode a* = *n*› ‹*asi* ≠ []›
‹*ni* −*asx′*@*asx*→∗ *n′*› ‹*slice-edges S as* = *slice-edges S asx′*›
‹*valid-edge a*› ‹*sourcenode a* = *n*› ‹*targetnode a* = *ni*›
show *?case*
apply(*rule-tac x*=*n′′* **in** *exI*)
apply(*rule-tac x*=*s′′* **in** *exI*)
apply(*rule-tac x*=*asx* **in** *exI*)
apply(*rule-tac x*=*a*#*asx′* **in** *exI*)
by(*auto intro*:*path.Cons-path simp*:*slice-edges-def*)
qed
qed
qed
qed

lemma *WS-weak-sim-trans*:
assumes ((n_1,s_1),(n_2,s_2)) ∈ *WS S*
and *S*,*kind* ⊢ (n_1,s_1) =*as*⇒∗ (n_1',s_1') **and** *as* ≠ []
shows ((n_1',s_1'),(n_1',*transfers* (*slice-kinds S as*) s_2)) ∈ *WS S* ∧
S,*slice-kind S* ⊢ (n_2,s_2) =*as*⇒∗ (n_1',*transfers* (*slice-kinds S as*) s_2)
proof −
obtain *f* **where** *f* = *kind* **by** *simp*
with ‹*S*,*kind* ⊢ (n_1,s_1) =*as*⇒∗ (n_1',s_1')›
have *S*,*f* ⊢ (n_1,s_1) =*as*⇒∗ (n_1',s_1') **by** *simp*
from ‹*S*,*f* ⊢ (n_1,s_1) =*as*⇒∗ (n_1',s_1')› ‹((n_1,s_1),(n_2,s_2)) ∈ *WS S*› ‹*as* ≠ []› ‹*f* = *kind*›
show ((n_1',s_1'),(n_1',*transfers* (*slice-kinds S as*) s_2)) ∈ *WS S* ∧
S,*slice-kind S* ⊢ (n_2,s_2) =*as*⇒∗ (n_1',*transfers* (*slice-kinds S as*) s_2)
proof(*induct arbitrary*:n_2 s_2 *rule*:*trans-observable-moves.induct*)
case *tom-Nil* **thus** *?case* **by** *simp*
next
case (*tom-Cons S f n s as n′ s′ as′ n′′ s′′*)
note *IH* = ‹⋀n_2 s_2. ⟦((*n′*,*s′*),(n_2,s_2)) ∈ *WS S*; *as′* ≠ []; *f* = *kind*⟧
⟹ ((*n′′*,*s′′*),(*n′′*,*transfers* (*slice-kinds S as′*) s_2)) ∈ *WS S* ∧
S,*slice-kind S* ⊢ (n_2,s_2) =*as′*⇒∗ (*n′′*,*transfers* (*slice-kinds S as′*) s_2)›

from ‹$S,f \vdash (n,s) = as \Rightarrow (n',s')$›
obtain *asx ax nx sx* **where** $S,f \vdash (n,s) = asx \Rightarrow_\tau (nx,sx)$
and $S,f \vdash (nx,sx) -ax\rightarrow (n',s')$ **and** $as = asx@[ax]$
by(*fastforce elim*:*observable-moves.cases*)
from ‹$S,f \vdash (nx,sx) -ax\rightarrow (n',s')$› **have** *obs nx* (*backward-slice S*) = {*nx*}
by(*fastforce intro*!:*n-in-obs elim*:*observable-move.cases*)
with ‹$S,f \vdash (n,s) = asx \Rightarrow_\tau (nx,sx)$› ‹$((n,s),(n_2,s_2)) \in WS\ S$› ‹$f = kind$›
have $((nx,sx),(n_2,s_2)) \in WS\ S$ **by**(*fastforce intro*:*WS-silent-moves*)
with ‹$S,f \vdash (nx,sx) -ax\rightarrow (n',s')$› ‹$f = kind$›
obtain *asx'* **where** $((n',s'),(n',transfer\ (slice\text{-}kind\ S\ ax)\ s_2)) \in WS\ S$
and $S,slice\text{-}kind\ S \vdash (n_2,s_2) = asx'@[ax] \Rightarrow$
$(n',transfer\ (slice\text{-}kind\ S\ ax)\ s_2)$
by(*fastforce elim*:*WS-observable-move*)
show *?case*
proof(*cases as'* = [])
case *True*
with ‹$S,f \vdash (n',s') = as' \Rightarrow* (n'',s'')$› **have** $n' = n'' \wedge s' = s''$
by(*fastforce elim*:*trans-observable-moves.cases dest*:*observable-move-notempty*)
from ‹$S,slice\text{-}kind\ S \vdash (n_2,s_2) = asx'@[ax] \Rightarrow$
$(n',transfer\ (slice\text{-}kind\ S\ ax)\ s_2)$›
have $S,slice\text{-}kind\ S \vdash (n_2,s_2) = (last\ (asx'@[ax]))\#[] \Rightarrow*$
$(n',transfer\ (slice\text{-}kind\ S\ ax)\ s_2)$
by(*fastforce intro*:*trans-observable-moves.intros*)
with ‹$((n',s'),(n',transfer\ (slice\text{-}kind\ S\ ax)\ s_2)) \in WS\ S$› ‹$as = asx@[ax]$›
‹$n' = n'' \wedge s' = s''$› *True*
show *?thesis* **by**(*fastforce simp*:*slice-kinds-def*)
next
case *False*
from *IH*[*OF* ‹$((n',s'),(n',transfer\ (slice\text{-}kind\ S\ ax)\ s_2)) \in WS\ S$› *this*
‹$f = kind$›]
have $((n'',s''),(n'',transfers\ (slice\text{-}kinds\ S\ as')$
$(transfer\ (slice\text{-}kind\ S\ ax)\ s_2))) \in WS\ S$
and $S,slice\text{-}kind\ S \vdash (n',transfer\ (slice\text{-}kind\ S\ ax)\ s_2)$
$= as' \Rightarrow* (n'',transfers\ (slice\text{-}kinds\ S\ as')$
$(transfer\ (slice\text{-}kind\ S\ ax)\ s_2))$ **by** *simp-all*
with ‹$S,slice\text{-}kind\ S \vdash (n_2,s_2) = asx'@[ax] \Rightarrow$
$(n',transfer\ (slice\text{-}kind\ S\ ax)\ s_2)$›
have $S,slice\text{-}kind\ S \vdash (n_2,s_2) = (last\ (asx'@[ax]))\#as' \Rightarrow*$
$(n'',transfers\ (slice\text{-}kinds\ S\ as')\ (transfer\ (slice\text{-}kind\ S\ ax)\ s_2))$
by(*fastforce intro*:*trans-observable-moves.tom-Cons*)
with ‹$((n'',s''),(n'',transfers\ (slice\text{-}kinds\ S\ as')$
$(transfer\ (slice\text{-}kind\ S\ ax)\ s_2))) \in WS\ S$› *False* ‹$as = asx@[ax]$›
show *?thesis* **by**(*fastforce simp*:*slice-kinds-def*)
qed
qed
qed

lemma *transfers-slice-kinds-slice-edges*:

transfers (*slice-kinds* *S* (*slice-edges* *S* *as*)) *s* = *transfers* (*slice-kinds* *S* *as*) *s*
proof(*induct as arbitrary*:*s*)
case *Nil* **thus** *?case* **by**(*simp add*:*slice-kinds-def slice-edges-def*)
next
case (*Cons a′ as′*)
note *IH* = ‹⋀*s*. *transfers* (*slice-kinds S* (*slice-edges S as′*)) *s* =
transfers (*slice-kinds S as′*) *s*›
show *?case*
proof(*cases sourcenode a′* ∈ *backward-slice S*)
case *True*
hence *eq*:*transfers* (*slice-kinds S* (*slice-edges S* (*a′#as′*))) *s*
= *transfers* (*slice-kinds S* (*slice-edges S as′*))
(*transfer* (*slice-kind S a′*) *s*)
by(*simp add*:*slice-edges-def slice-kinds-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 transfer* (*slice-kind S a′*) *s*] **show** *?thesis* **by** *simp*
next
case *False*
hence *eq*:*transfers* (*slice-kinds S* (*slice-edges S* (*a′#as′*))) *s*
= *transfers* (*slice-kinds S* (*slice-edges S as′*)) *s*
by(*simp add*:*slice-edges-def slice-kinds-def*)
from *False* **have** *transfer* (*slice-kind S a′*) *s* = *s*
by(*cases kind a′*,*auto simp*:*slice-kind-def Let-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 s*] **show** *?thesis* **by** *simp*
qed
qed

lemma *trans-observable-moves-preds*:
assumes *S*,*f* ⊢ (*n*,*s*) =*as*⇒∗ (*n′*,*s′*) **and** *valid-node n*
obtains *as′* **where** *preds* (*map f as′*) *s* **and** *slice-edges S as′* = *as*
and *n* −*as′*→∗ *n′*
proof(*atomize-elim*)
from ‹*S*,*f* ⊢ (*n*,*s*) =*as*⇒∗ (*n′*,*s′*)› ‹*valid-node n*›
show ∃ *as′*. *preds* (*map f as′*) *s* ∧ *slice-edges S as′* = *as* ∧ *n* −*as′*→∗ *n′*
proof(*induct rule*:*trans-observable-moves.induct*)
case *tom-Nil* **thus** *?case*
by(*rule-tac x*=[] **in** *exI*,*fastforce intro*:*empty-path simp*:*slice-edges-def*)
next
case (*tom-Cons S f n s as n′ s′ as′ n″ s″*)
note *IH* = ‹*valid-node n′*
⟹ ∃ *asx*. *preds* (*map f asx*) *s′* ∧ *slice-edges S asx* = *as′* ∧ *n′* −*asx*→∗ *n″*›
from ‹*S*,*f* ⊢ (*n*,*s*) =*as*⇒ (*n′*,*s′*)›
have *preds* (*map f as*) *s* **and** *transfers* (*map f as*) *s* = *s′*

and n $-as\rightarrow*$ n'
by(*fastforce dest*:*observable-moves-preds-transfers-path*)+
from ‹n $-as\rightarrow*$ n'› **have** *valid-node* n' **by**(*fastforce dest*:*path-valid-node*)
from ‹$S,f \vdash (n,s) =as\Rightarrow (n',s')$› **have** *slice-edges S as* = [*last as*]
by(*rule observable-moves-last-slice-edges*)
from *IH*[*OF* ‹*valid-node* n'›]
obtain *asx* **where** *preds* (*map f asx*) s' **and** *slice-edges S asx* = as'
and n' $-asx\rightarrow*$ n''
by *blast*
from ‹n $-as\rightarrow*$ n'› ‹n' $-asx\rightarrow*$ n''› **have** n $-as@asx\rightarrow*$ n'' **by**(*rule path-Append*)
from ‹*preds* (*map f asx*) s'› ‹*transfers* (*map f as*) $s = s'$›[*THEN sym*]
‹*preds* (*map f as*) *s*›
have *preds* (*map f* (*as*@*asx*)) *s* **by**(*simp add*:*preds-split*)
with ‹*slice-edges S as* = [*last as*]› ‹*slice-edges S asx* = as'›
‹n $-as@asx\rightarrow*$ n''› **show** *?case*
by(*rule-tac x*=*as*@*asx* **in** *exI*,*auto simp*:*slice-edges-def*)
qed
qed

lemma *exists-sliced-path-preds*:
assumes n $-as\rightarrow*$ n' **and** *slice-edges S as* = [] **and** $n' \in$ *backward-slice S*
obtains as' **where** n $-as'\rightarrow*$ n' **and** *preds* (*slice-kinds S* as') *s*
and *slice-edges S* as' = []
proof(*atomize-elim*)
from ‹*slice-edges S as* = []›
have $\forall$ *nx* $\in$ *set*(*sourcenodes as*). *nx* $\notin$ (*backward-slice S*)
by(*rule slice-edges-no-nodes-in-slice*)
with ‹n $-as\rightarrow*$ n'› ‹$n' \in$ *backward-slice S*› **have** $n' \in$ *obs n* (*backward-slice S*)
by −(*rule obs-elem*)
hence *obs n* (*backward-slice S*) = $\{n'\}$ **by**(*rule obs-singleton-element*)
from ‹n $-as\rightarrow*$ n'› **have** *valid-node n* **and** *valid-node* n'
by(*fastforce dest*:*path-valid-node*)+
from ‹n $-as\rightarrow*$ n'› **obtain** *x* **where** *distance n* n' *x* **and** $x \leq$ *length as*
by(*erule every-path-distance*)
from ‹*distance n* n' *x*› ‹*obs n* (*backward-slice S*) = $\{n'\}$›
show $\exists$ as'. n $-as'\rightarrow*$ n' $\wedge$ *preds* (*slice-kinds S* as') *s* $\wedge$
slice-edges S as' = []
proof(*induct x arbitrary*:*n rule*:*nat.induct*)
case *zero*
from ‹*distance n* n' *0*› **have** $n = n'$ **by**(*fastforce elim*:*distance.cases*)
with ‹*valid-node* n'› **show** *?case*
by(*rule-tac x*=[] **in** *exI*,
auto intro:*empty-path simp*:*slice-kinds-def slice-edges-def*)
next
case (*Suc x*)
note *IH* = ‹$\bigwedge n$. ⟦*distance n* n' *x*; *obs n* (*backward-slice S*) = $\{n'\}$⟧
$\Longrightarrow$ $\exists$ as'. n $-as'\rightarrow*$ n' $\wedge$ *preds* (*slice-kinds S* as') *s* $\wedge$

slice-edges S as′ = []›
from ‹*distance n n′* (*Suc x*)› **obtain** *a*
where *valid-edge a* **and** *n* = *sourcenode a*
and *distance* (*targetnode a*) *n′ x*
and *target*:*targetnode a* = (*SOME nx*. ∃ *a′*. *sourcenode a* = *sourcenode a′* ∧
distance (*targetnode a′*) *n′ x* ∧
valid-edge a′ ∧ *targetnode a′* = *nx*)
by(*auto elim*:*distance-successor-distance*)
have *n* ∉ *backward-slice S*
proof
assume *n* ∈ *backward-slice S*
from ‹*valid-edge a*› ‹*n* = *sourcenode a*› **have** *valid-node n* **by** *simp*
with ‹*n* ∈ *backward-slice S*› **have** *obs n* (*backward-slice S*) = {*n*}
by −(*rule n-in-obs*)
with ‹*obs n* (*backward-slice S*) = {*n′*}› **have** *n* = *n′* **by** *simp*
with ‹*valid-node n*› **have** *n* −[]→∗ *n′* **by**(*fastforce intro*:*empty-path*)
with ‹*distance n n′* (*Suc x*)› **show** *False*
by(*fastforce elim*:*distance.cases*)
qed
from ‹*distance* (*targetnode a*) *n′ x*› ‹*n′* ∈ *backward-slice S*›
obtain *m* **where** *m* ∈ *obs* (*targetnode a*) (*backward-slice S*)
by(*fastforce elim*:*distance.cases path-ex-obs*)
from ‹*valid-edge a*› ‹*n* ∉ *backward-slice S*› ‹*n* = *sourcenode a*›
have *obs* (*targetnode a*) (*backward-slice S*) ⊆
obs (*sourcenode a*) (*backward-slice S*)
by −(*rule edge-obs-subset*,*auto*)
with ‹*m* ∈ *obs* (*targetnode a*) (*backward-slice S*)› ‹*n* = *sourcenode a*›
‹*obs n* (*backward-slice S*) = {*n′*}›
have *n′* ∈ *obs* (*targetnode a*) (*backward-slice S*) **by** *auto*
hence *obs* (*targetnode a*) (*backward-slice S*) = {*n′*}
by(*rule obs-singleton-element*)
from *IH*[*OF* ‹*distance* (*targetnode a*) *n′ x*› *this*]
obtain *as* **where** *targetnode a* −*as*→∗ *n′* **and** *preds* (*slice-kinds S as*) *s*
and *slice-edges S as* = [] **by** *blast*
from ‹*targetnode a* −*as*→∗ *n′*› ‹*valid-edge a*› ‹*n* = *sourcenode a*›
have *n* −*a*#*as*→∗ *n′* **by**(*fastforce intro*:*Cons-path*)
from ‹*slice-edges S as* = []› ‹*n* ∉ *backward-slice S*› ‹*n* = *sourcenode a*›
have *slice-edges S* (*a*#*as*) = [] **by**(*simp add*:*slice-edges-def*)
show *?case*
proof(*cases kind a*)
case (*Update f*)
with ‹*n* ∉ *backward-slice S*› ‹*n* = *sourcenode a*› **have** *slice-kind S a* = ⇑*id*
by(*fastforce intro*:*slice-kind-Upd*)
hence *transfer* (*slice-kind S a*) *s* = *s* **and** *pred* (*slice-kind S a*) *s*
by *simp-all*
with ‹*preds* (*slice-kinds S as*) *s*› **have** *preds* (*slice-kinds S* (*a*#*as*)) *s*
by(*simp add*:*slice-kinds-def*)
with ‹*n* −*a*#*as*→∗ *n′*› ‹*slice-edges S* (*a*#*as*) = []› **show** *?thesis*
by *blast*

next
  case (*Predicate Q*)
  with ‹$n \notin$ *backward-slice S*› ‹n = *sourcenode a*› ‹*distance n n′* (*Suc x*)›
    ‹*obs n* (*backward-slice S*) = {*n′*}› ‹*distance* (*targetnode a*) *n′ x*›
    ‹*targetnode a* = (*SOME nx*. ∃ *a′*. *sourcenode a* = *sourcenode a′* ∧
    distance (*targetnode a′*) *n′ x* ∧
    valid-edge a′ ∧ *targetnode a′* = *nx*)›
  have *slice-kind S a* = (λ*s*. *True*)$_{\surd}$
    by(*fastforce intro*:*slice-kind-Pred-obs-nearer-SOME*)
  hence *transfer* (*slice-kind S a*) *s* = *s* **and** *pred* (*slice-kind S a*) *s*
    by *simp-all*
  with ‹*preds* (*slice-kinds S as*) *s*› **have** *preds* (*slice-kinds S* (*a#as*)) *s*
    by(*simp add*:*slice-kinds-def*)
  with ‹*n* −*a#as*→∗ *n′*› ‹*slice-edges S* (*a#as*) = []› **show** *?thesis* **by** *blast*
qed
qed
qed

theorem *fundamental-property-of-static-slicing*:
  assumes *path*:*n* −*as*→∗ *n′* **and** *preds*:*preds* (*kinds as*) *s* **and** *n′* ∈ *S*
  obtains *as′* **where** *preds* (*slice-kinds S as′*) *s*
  and (∀ *V* ∈ *Use n′*. *state-val* (*transfers* (*slice-kinds S as′*) *s*) *V* =
      state-val (*transfers* (*kinds as*) *s*) *V*)
  and *slice-edges S as* = *slice-edges S as′* **and** *n* −*as′*→∗ *n′*
proof(*atomize-elim*)
  from *path preds* **obtain** *n″ s″ as′ as″*
    where *S*,*kind* ⊢ (*n*,*s*) =*slice-edges S as*⇒∗ (*n″*,*s″*)
    and *S*,*kind* ⊢ (*n″*,*s″*) =*as′*⇒$_{\tau}$ (*n′*,*transfers* (*kinds as*) *s*)
    and *slice-edges S as* = *slice-edges S as″*
    and *n* −*as″*@*as′*→∗ *n′*
    by −(*erule-tac S*=*S* **in** *path-trans-observable-moves*,*auto*)
  from *path* **have** *valid-node n* **and** *valid-node n′*
    by(*fastforce dest*:*path-valid-node*)+
  from ‹*valid-node n*› **have** ((*n*,*s*),(*n*,*s*)) ∈ *WS S* **by**(*fastforce intro*:*WSI*)
  from ‹*valid-node n′*› ‹*n′* ∈ *S*› **have** *obs n′* (*backward-slice S*) = {*n′*}
    by(*fastforce intro*!:*n-in-obs refl*)
  from ‹*valid-node n′*› **have** *n′*−[]→∗ *n′* **by**(*fastforce intro*:*empty-path*)
  with ‹*valid-node n′*› ‹*n′* ∈ *S*› **have** ∀ *V* ∈ *Use n′*. *V* ∈ *rv S n′*
    by(*fastforce intro*:*rvI refl simp*:*sourcenodes-def*)
  show ∃ *as′*. *preds* (*slice-kinds S as′*) *s* ∧
    (∀ *V* ∈ *Use n′*. *state-val* (*transfers* (*slice-kinds S as′*) *s*) *V* =
      state-val (*transfers* (*kinds as*) *s*) *V*) ∧
    slice-edges S as = *slice-edges S as′* ∧ *n* −*as′*→∗ *n′*
  proof(*cases slice-edges S as* = [])
    case *True*
    hence *preds* (*slice-kinds S* []) *s* **and** *slice-edges S* [] = *slice-edges S as*
      by(*simp-all add*:*slice-kinds-def slice-edges-def*)
    from ‹*S*,*kind* ⊢ (*n*,*s*) =*slice-edges S as*⇒∗ (*n″*,*s″*)› *True*

have *n* = *n''* **and** *s* = *s''*
  by(*fastforce elim*:*trans-observable-moves.cases*)+
with ‹*S*,*kind* ⊢ (*n''*,*s''*) =*as'*⇒$_\tau$ (*n'*,*transfers* (*kinds as*) *s*)›
have *S*,*kind* ⊢ (*n*,*s*) =*as'*⇒$_\tau$ (*n'*,*transfers* (*kinds as*) *s*) **by** *simp*
with ‹*valid-node n*› **have** *n* −*as'*→∗ *n'*
  by(*fastforce dest*:*silent-moves-preds-transfers-path*)
from ‹*S*,*kind* ⊢ (*n*,*s*) =*as'*⇒$_\tau$ (*n'*,*transfers* (*kinds as*) *s*)›
have *slice-edges S as'* = [] **by**(*fastforce dest*:*silent-moves-no-slice-edges*)
with ‹*n* −*as'*→∗ *n'*› ‹*valid-node n'*› ‹*n'* ∈ *S*› **obtain** *asx*
  where *n* −*asx*→∗ *n'* **and** *preds* (*slice-kinds S asx*) *s*
  and *slice-edges S asx* = []
  by −(*erule exists-sliced-path-preds*,*auto intro*:*refl*)
from ‹*S*,*kind* ⊢ (*n*,*s*) =*as'*⇒$_\tau$ (*n'*,*transfers* (*kinds as*) *s*)›
  ‹((*n*,*s*),(*n*,*s*)) ∈ *WS S*› ‹*obs n'* (*backward-slice S*) = {*n'*}›
have ((*n'*,*transfers* (*kinds as*) *s*),(*n*,*s*)) ∈ *WS S*
  by(*fastforce intro*:*WS-silent-moves*)
with *True* **have** ∀ *V* ∈ *rv S n'*. *state-val* (*transfers* (*kinds as*) *s*) *V* =
  state-val (*transfers* (*slice-kinds S* (*slice-edges S as*)) *s*) *V*
  by(*fastforce dest*:*WSD simp*:*slice-edges-def slice-kinds-def*)
with ‹∀ *V* ∈ *Use n'*. *V* ∈ *rv S n'*›
have ∀ *V* ∈ *Use n'*. *state-val* (*transfers* (*kinds as*) *s*) *V* =
  state-val (*transfers* (*slice-kinds S* (*slice-edges S as*)) *s*) *V* **by** *simp*
with ‹*slice-edges S asx* = []› ‹*slice-edges S* [] = *slice-edges S as*›
have ∀ *V* ∈ *Use n'*. *state-val* (*transfers* (*kinds as*) *s*) *V* =
  state-val (*transfers* (*slice-kinds S* (*slice-edges S asx*)) *s*) *V*
  by(*simp add*:*slice-edges-def*)
hence ∀ *V* ∈ *Use n'*. *state-val* (*transfers* (*kinds as*) *s*) *V* =
  state-val (*transfers* (*slice-kinds S asx*) *s*) *V*
  by(*simp add*:*transfers-slice-kinds-slice-edges*)
with ‹*n* −*asx*→∗ *n'*› ‹*preds* (*slice-kinds S asx*) *s*›
  ‹*slice-edges S asx* = []› ‹*slice-edges S* [] = *slice-edges S as*›
show *?thesis*
  by(*rule-tac x*=*asx* **in** *exI*,*simp add*:*slice-edges-def*)
next
case *False*
with ‹*S*,*kind* ⊢ (*n*,*s*) =*slice-edges S as*⇒∗ (*n''*,*s''*)› ‹((*n*,*s*),(*n*,*s*)) ∈ *WS S*›
have ((*n''*,*s''*),(*n''*,*transfers* (*slice-kinds S* (*slice-edges S as*)) *s*)) ∈ *WS S*
  S,*slice-kind S* ⊢ (*n*,*s*) =*slice-edges S as*⇒∗
  (*n''*,*transfers* (*slice-kinds S* (*slice-edges S as*)) *s*)
  by(*fastforce dest*:*WS-weak-sim-trans*)+
from ‹*S*,*slice-kind S* ⊢ (*n*,*s*) =*slice-edges S as*⇒∗
      (*n''*,*transfers* (*slice-kinds S* (*slice-edges S as*)) *s*)›
  ‹*valid-node n*›
obtain *asx* **where** *preds* (*slice-kinds S asx*) *s*
  and *slice-edges S asx* = *slice-edges S as*
  and *n* −*asx*→∗ *n''*
  by(*fastforce elim*:*trans-observable-moves-preds simp*:*slice-kinds-def*)
from ‹*n* −*asx*→∗ *n''*› **have** *valid-node n''* **by**(*fastforce dest*:*path-valid-node*)
with ‹*S*,*kind* ⊢ (*n''*,*s''*) =*as'*⇒$_\tau$ (*n'*,*transfers* (*kinds as*) *s*)›

have $n'' -as' \rightarrow * \ n'$
by(*fastforce dest*:*silent-moves-preds-transfers-path*)
from ‹*S*,*kind* ⊢ (n'',s'') =as'⇒$_\tau$ (n',*transfers* (*kinds as*) *s*)›
have *slice-edges S* as' = [] **by**(*fastforce dest*:*silent-moves-no-slice-edges*)
with ‹$n'' -as' \rightarrow * \ n'$› ‹*valid-node* n'› ‹$n' \in S$› **obtain** asx'
where $n'' -asx' \rightarrow * \ n'$ **and** *slice-edges S* asx' = []
and *preds* (*slice-kinds S* asx') (*transfers* (*slice-kinds S asx*) *s*)
by −(*erule exists-sliced-path-preds*,*auto intro*:*refl*)
from ‹$n -asx \rightarrow * \ n''$› ‹$n'' -asx' \rightarrow * \ n'$› **have** $n -asx@asx' \rightarrow * \ n'$
by(*rule path-Append*)
from ‹*slice-edges S asx* = *slice-edges S as*› ‹*slice-edges S* asx' = []›
have *slice-edges S as* = *slice-edges S* (*asx*@asx')
by(*auto simp*:*slice-edges-def*)
from ‹*preds* (*slice-kinds S* asx') (*transfers* (*slice-kinds S asx*) *s*)›
‹*preds* (*slice-kinds S asx*) *s*›
have *preds* (*slice-kinds S* (*asx*@asx')) *s*
by(*simp add*:*slice-kinds-def preds-split*)
from ‹*obs* n' (*backward-slice S*) = {n'}›
‹*S*,*kind* ⊢ (n'',s'') =as'⇒$_\tau$ (n',*transfers* (*kinds as*) *s*)›
‹((n'',s''),(n'',*transfers* (*slice-kinds S* (*slice-edges S as*)) *s*)) ∈ *WS S*›
have ((n',*transfers* (*kinds as*) *s*),
(n'',*transfers* (*slice-kinds S* (*slice-edges S as*)) *s*)) ∈ *WS S*
by(*fastforce intro*:*WS-silent-moves*)
hence ∀ *V* ∈ *rv S* n'. *state-val* (*transfers* (*kinds as*) *s*) *V* =
state-val (*transfers* (*slice-kinds S* (*slice-edges S as*)) *s*) *V*
by(*fastforce dest*:*WSD*)
with ‹∀ *V* ∈ *Use* n'. *V* ∈ *rv S* n'› ‹*slice-edges S asx* = *slice-edges S as*›
have ∀ *V* ∈ *Use* n'. *state-val* (*transfers* (*kinds as*) *s*) *V* =
state-val (*transfers* (*slice-kinds S* (*slice-edges S asx*)) *s*) *V*
by *fastforce*
with ‹*slice-edges S* asx' = []›
have ∀ *V* ∈ *Use* n'. *state-val* (*transfers* (*kinds as*) *s*) *V* =
state-val (*transfers* (*slice-kinds S* (*slice-edges S* (*asx*@asx'))) *s*) *V*
by(*auto simp*:*slice-edges-def*)
hence ∀ *V* ∈ *Use* n'. *state-val* (*transfers* (*kinds as*) *s*) *V* =
state-val (*transfers* (*slice-kinds S* (*asx*@asx')) *s*) *V*
by(*simp add*:*transfers-slice-kinds-slice-edges*)
with ‹*preds* (*slice-kinds S* (*asx*@asx')) *s*› ‹$n -asx@asx' \rightarrow * \ n'$›
‹*slice-edges S as* = *slice-edges S* (*asx*@asx')›
show *?thesis* **by** *simp blast*
qed
qed

end

3.3.6 The fundamental property of (static) slicing related to the semantics

locale *BackwardSlice-wf* =
BackwardSlice sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice +
CFG-semantics-wf sourcenode targetnode kind valid-edge Entry sem identifies
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (‹'('-Entry'-')›) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *backward-slice* :: *'node set* ⇒ *'node set*
and *sem* :: *'com* ⇒ *'state* ⇒ *'com* ⇒ *'state* ⇒ *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 *n* ≜ *c* **and** ⟨*c*,*s*⟩ ⇒ ⟨*c'*,*s'*⟩
obtains *n' as* **where** *n* −*as*→∗ *n'* **and** *preds* (*slice-kinds* {*n'*} *as*) *s* **and** *n'* ≜ *c'*
and ∀ *V* ∈ *Use n'*. *state-val* (*transfers* (*slice-kinds* {*n'*} *as*) *s*) *V* = *state-val s' V*
proof(*atomize-elim*)
from ‹*n* ≜ *c*› ‹⟨*c*,*s*⟩ ⇒ ⟨*c'*,*s'*⟩› **obtain** *n' as* **where** *n* −*as*→∗ *n'*
and *transfers* (*kinds as*) *s* = *s'* **and** *preds* (*kinds as*) *s* **and** *n'* ≜ *c'*
by(*fastforce dest*:*fundamental-property*)
from ‹*n* −*as*→∗ *n'*› ‹*preds* (*kinds as*) *s*› **obtain** *as'*
where *preds* (*slice-kinds* {*n'*} *as'*) *s*
and *vals*:∀ *V* ∈ *Use n'*. *state-val* (*transfers* (*slice-kinds* {*n'*} *as'*) *s*) *V* =
state-val (*transfers* (*kinds as*) *s*) *V* **and** *n* −*as'*→∗ *n'*
by −(*erule fundamental-property-of-static-slicing*,*auto*)
from ‹*transfers* (*kinds as*) *s* = *s'*› *vals* **have** ∀ *V* ∈ *Use n'*.
state-val (*transfers* (*slice-kinds* {*n'*} *as'*) *s*) *V* = *state-val s' V*
by *simp*
with ‹*preds* (*slice-kinds* {*n'*} *as'*) *s*› ‹*n* −*as'*→∗ *n'*› ‹ *n'* ≜ *c'*›
show ∃ *as n'*. *n* −*as*→∗ *n'* ∧ *preds* (*slice-kinds* {*n'*} *as*) *s* ∧ *n'* ≜ *c'* ∧
(∀ *V*∈*Use n'*. *state-val* (*transfers* (*slice-kinds* {*n'*} *as*) *s*) *V* = *state-val s' V*)
by *blast*
qed

end

end

3.4 Static Standard Control Dependence

theory *StandardControlDependence* **imports**

../*Basic*/*Postdomination*
../*Basic*/*DynStandardControlDependence*
begin

context *Postdomination* **begin**

Definition and some lemmas

definition *standard-control-dependence* :: $'node \Rightarrow 'node \Rightarrow bool$
(‹- $controls_s$ -› [*51*,*0*])
where *standard-control-dependences-eq*:$n\ controls_s\ n' \equiv \exists\, as.\ n\ controls_s\ n'\ via\ as$

lemma *standard-control-dependence-def*:$n\ controls_s\ n' =$
$(\exists\, a\ a'\ as.\ (n' \notin set(sourcenodes\ (a\#as))) \wedge (n\ -a\#as\rightarrow*\ n') \wedge$
$(n'\ postdominates\ (targetnode\ a)) \wedge$
$(valid\text{-}edge\ a') \wedge (sourcenode\ a' = n) \wedge$
$(\neg\ n'\ postdominates\ (targetnode\ a')))$
by(*auto simp*:*standard-control-dependences-eq dyn-standard-control-dependence-def*)

lemma *Exit-not-standard-control-dependent*:
$n\ controls_s\ (\text{-}Exit\text{-}) \Longrightarrow False$
by(*auto simp*:*standard-control-dependences-eq*
intro:*Exit-not-dyn-standard-control-dependent*)

lemma *standard-control-dependence-def-variant*:
$n\ controls_s\ n' = (\exists\, as.\ (n\ -as\rightarrow*\ n') \wedge (n \neq n') \wedge$
$(\neg\ n'\ postdominates\ n) \wedge (n' \notin set(sourcenodes\ as)) \wedge$
$(\forall\, n'' \in set(targetnodes\ as).\ n'\ postdominates\ n''))$
by(*auto simp*:*standard-control-dependences-eq*
dyn-standard-control-dependence-def-variant)

lemma *inner-node-standard-control-dependence-predecessor*:
assumes *inner-node n* (-*Entry*-) $-as\rightarrow*$ n n $-as'\rightarrow*$ (-*Exit*-)
obtains n' **where** $n'\ controls_s\ n$
using *assms*
by(*auto elim*!:*inner-node-dyn-standard-control-dependence-predecessor*
simp:*standard-control-dependences-eq*)

end

end

3.5 Static Weak Control Dependence

theory *WeakControlDependence* **imports**
../*Basic*/*Postdomination*

../Basic/DynWeakControlDependence
begin

context *StrongPostdomination* **begin**

definition
weak-control-dependence :: *'node* ⇒ *'node* ⇒ *bool*
(‹- *weakly controls* -› [*51*,*0*])
where *weak-control-dependences-eq*:
n weakly controls n' ≡ ∃ *as*. *n weakly controls n' via as*

lemma
weak-control-dependence-def:*n weakly controls n'* =
(∃ *a a' as*. (*n'* ∉ *set*(*sourcenodes* (*a*#*as*))) ∧ (*n* −*a*#*as*→∗ *n'*) ∧
(*n' strongly*−*postdominates* (*targetnode a*)) ∧
(*valid-edge a'*) ∧ (*sourcenode a'* = *n*) ∧
(¬ *n' strongly*−*postdominates* (*targetnode a'*)))
by(*auto simp*:*weak-control-dependences-eq dyn-weak-control-dependence-def*)

lemma *Exit-not-weak-control-dependent*:
n weakly controls (*-Exit-*) ⟹ *False*
by(*auto simp*:*weak-control-dependences-eq*
intro:*Exit-not-dyn-weak-control-dependent*)

end

end

3.6 Program Dependence Graph

theory *PDG* **imports**
DataDependence
StandardControlDependence
WeakControlDependence
../Basic/CFGExit-wf
begin

locale *PDG* =
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (‹'('-*Entry*'-')›) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *Exit* :: *'node* (‹'('-*Exit*'-')›) +
fixes *control-dependence* :: *'node* ⇒ *'node* ⇒ *bool*
(‹- *controls* - › [*51*,*0*])
assumes *Exit-not-control-dependent*:*n controls n'* ⟹ *n'* ≠ (*-Exit-*)
assumes *control-dependence-path*:

n controls n′
$\Longrightarrow \exists$ *as. CFG.path sourcenode targetnode valid-edge n as n′* $\wedge$ *as* $\neq$ []

begin

inductive *cdep-edge* :: *′node* ⇒ *′node* ⇒ *bool*
(‹- $\longrightarrow_{cd}$ -› [*51*,*0*] *80*)
and *ddep-edge* :: *′node* ⇒ *′var* ⇒ *′node* ⇒ *bool*
(‹- $-$-$\rightarrow_{dd}$ -› [*51*,*0*,*0*] *80*)
and *PDG-edge* :: *′node* ⇒ *′var option* ⇒ *′node* ⇒ *bool*

where

$n \longrightarrow_{cd} n'$ == *PDG-edge n None n′*
| $n -V\rightarrow_{dd} n'$ == *PDG-edge n* (*Some V*) *n′*

| *PDG-cdep-edge*:
n controls n′ $\Longrightarrow n \longrightarrow_{cd} n'$

| *PDG-ddep-edge*:
n influences V in n′ $\Longrightarrow n -V\rightarrow_{dd} n'$

inductive *PDG-path* :: *′node* ⇒ *′node* ⇒ *bool*
(‹- $\longrightarrow_{d}*$ -› [*51*,*0*] *80*)

where *PDG-path-Nil*:
valid-node n $\Longrightarrow n \longrightarrow_{d}* n$

| *PDG-path-Append-cdep*:
$[\![n \longrightarrow_{d}* n''; n'' \longrightarrow_{cd} n']\!] \Longrightarrow n \longrightarrow_{d}* n'$

| *PDG-path-Append-ddep*:
$[\![n \longrightarrow_{d}* n''; n'' -V\rightarrow_{dd} n']\!] \Longrightarrow n \longrightarrow_{d}* n'$

lemma *PDG-path-cdep*:$n \longrightarrow_{cd} n' \Longrightarrow n \longrightarrow_{d}* n'$
apply $-$
apply(*rule PDG-path-Append-cdep*, *rule PDG-path-Nil*)
by(*auto elim*!:*PDG-edge.cases dest*:*control-dependence-path path-valid-node*)

lemma *PDG-path-ddep*:$n -V\rightarrow_{dd} n' \Longrightarrow n \longrightarrow_{d}* n'$
apply $-$
apply(*rule PDG-path-Append-ddep*, *rule PDG-path-Nil*)
by(*auto elim*!:*PDG-edge.cases dest*:*path-valid-node simp*:*data-dependence-def*)

lemma *PDG-path-Append*:

$[\![n'' \longrightarrow_d* n'; n \longrightarrow_d* n'']\!] \Longrightarrow n \longrightarrow_d* n'$
by(*induct rule*:*PDG-path.induct*,*auto intro*:*PDG-path.intros*)

lemma *PDG-cdep-edge-CFG-path*:
assumes $n \longrightarrow_{cd} n'$ **obtains** *as* **where** $n -as\rightarrow* n'$ **and** $as \neq []$
using ‹$n \longrightarrow_{cd} n'$›
by(*auto elim*:*PDG-edge.cases dest*:*control-dependence-path*)

lemma *PDG-ddep-edge-CFG-path*:
assumes $n -V\rightarrow_{dd} n'$ **obtains** *as* **where** $n -as\rightarrow* n'$ **and** $as \neq []$
using ‹$n -V\rightarrow_{dd} n'$›
by(*auto elim*!:*PDG-edge.cases simp*:*data-dependence-def*)

lemma *PDG-path-CFG-path*:
assumes $n \longrightarrow_d* n'$ **obtains** *as* **where** $n -as\rightarrow* n'$
proof(*atomize-elim*)
from ‹$n \longrightarrow_d* n'$› **show** $\exists$ *as*. $n -as\rightarrow* n'$
proof(*induct rule*:*PDG-path.induct*)
case (*PDG-path-Nil n*)
hence $n -[]\rightarrow* n$ **by**(*rule empty-path*)
thus *?case* **by** *blast*
next
case (*PDG-path-Append-cdep n n'' n'*)
from ‹$n'' \longrightarrow_{cd} n'$› **obtain** *as* **where** $n'' -as\rightarrow* n'$
by(*fastforce elim*:*PDG-cdep-edge-CFG-path*)
with ‹$\exists$ *as*. $n -as\rightarrow* n''$› **obtain** *as'* **where** $n -as'@as\rightarrow* n'$
by(*auto dest*:*path-Append*)
thus *?case* **by** *blast*
next
case (*PDG-path-Append-ddep n n'' V n'*)
from ‹$n'' -V\rightarrow_{dd} n'$› **obtain** *as* **where** $n'' -as\rightarrow* n'$
by(*fastforce elim*:*PDG-ddep-edge-CFG-path*)
with ‹$\exists$ *as*. $n -as\rightarrow* n''$› **obtain** *as'* **where** $n -as'@as\rightarrow* n'$
by(*auto dest*:*path-Append*)
thus *?case* **by** *blast*
qed
qed

lemma *PDG-path-Exit*:$[\![n \longrightarrow_d* n'; n' = (\text{-}Exit\text{-})]\!] \Longrightarrow n = (\text{-}Exit\text{-})$
apply(*induct rule*:*PDG-path.induct*)
by(*auto elim*:*PDG-edge.cases dest*:*Exit-not-control-dependent*
simp:*data-dependence-def*)

lemma *PDG-path-not-inner*:
$[\![n \longrightarrow_d* n'; \neg$ *inner-node* $n']\!] \Longrightarrow n = n'$
proof(*induct rule*:*PDG-path.induct*)

case (*PDG-path-Nil n*)
thus *?case* **by** *simp*
next
case (*PDG-path-Append-cdep n n'' n'*)
from ‹$n'' \longrightarrow_{cd} n'$› ‹¬ *inner-node n'*› **have** *False*
apply −
apply(*erule PDG-edge.cases*) **apply**(*auto simp*:*inner-node-def*)
apply(*fastforce dest*:*control-dependence-path path-valid-node*)
apply(*fastforce dest*:*control-dependence-path path-valid-node*)
by(*fastforce dest*:*Exit-not-control-dependent*)
thus *?case* **by** *simp*
next
case (*PDG-path-Append-ddep n n'' V n'*)
from ‹$n'' -V\rightarrow_{dd} n'$› ‹¬ *inner-node n'*› **have** *False*
apply −
apply(*erule PDG-edge.cases*)
by(*auto dest*:*path-valid-node simp*:*inner-node-def data-dependence-def*)
thus *?case* **by** *simp*
qed

3.6.1 Definition of the static backward slice

Node: instead of a single node, we calculate the backward slice of a set of nodes.

definition *PDG-BS* :: *'node set* ⇒ *'node set*
where *PDG-BS S* ≡ $\{n'. \exists n. n' \longrightarrow_{d}* n \wedge n \in S \wedge$ *valid-node n*$\}$

lemma *PDG-BS-valid-node*:$n \in$ *PDG-BS S* ⟹ *valid-node n*
by(*auto elim*:*PDG-path-CFG-path dest*:*path-valid-node simp*:*PDG-BS-def split*:*if-split-asm*)

lemma *Exit-PDG-BS*:$n \in$ *PDG-BS* {(-*Exit*-)} ⟹ n = (-*Exit*-)
by(*fastforce dest*:*PDG-path-Exit simp*:*PDG-BS-def*)

end

3.6.2 Instantiate static PDG

Standard control dependence

locale *StandardControlDependencePDG* =
Postdomination sourcenode targetnode kind valid-edge Entry Exit +
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (‹'('-Entry'-')›) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*

and *Exit* :: *'node* (‹'('-*Exit*'-')›)

begin

lemma *PDG-scd*:
PDG sourcenode targetnode kind valid-edge (-*Entry*-)
Def Use state-val (-*Exit*-) *standard-control-dependence*
proof(*unfold-locales*)
fix *n n'* **assume** *n* $controls_s$ *n'*
show *n'* ≠ (-*Exit*-)
proof
assume *n'* = (-*Exit*-)
with ‹*n* $controls_s$ *n'*› **show** *False*
by(*fastforce intro*:*Exit-not-standard-control-dependent*)
qed
next
fix *n n'* **assume** *n* $controls_s$ *n'*
thus ∃ *as*. *n* −*as*→∗ *n'* ∧ *as* ≠ []
by(*fastforce simp*:*standard-control-dependence-def*)
qed

end

Weak control dependence

locale *WeakControlDependencePDG* =
StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit +
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (‹'('-*Entry*'-')›) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *Exit* :: *'node* (‹'('-*Exit*'-')›)

begin

lemma *PDG-wcd*:
PDG sourcenode targetnode kind valid-edge (-*Entry*-)
Def Use state-val (-*Exit*-) *weak-control-dependence*
proof(*unfold-locales*)
fix *n n'* **assume** *n weakly controls n'*
show *n'* ≠ (-*Exit*-)
proof
assume *n'* = (-*Exit*-)
with ‹*n weakly controls n'*› **show** *False*
by(*fastforce intro*:*Exit-not-weak-control-dependent*)
qed
next

fix $n\ n'$ **assume** n *weakly controls* n'
thus $\exists\, as.\ n\ -as\rightarrow*\ n' \wedge as \neq []$
by(*fastforce simp*:*weak-control-dependence-def*)
qed

end

end

3.7 Weak Order Dependence

theory *WeakOrderDependence* **imports** ../*Basic*/*CFG DataDependence* **begin**

Weak order dependence is just defined as a static control dependence

3.7.1 Definition and some lemmas

definition (**in** *CFG*) *weak-order-dependence* :: $'node \Rightarrow 'node \Rightarrow 'node \Rightarrow bool$
$(\langle\text{-} \longrightarrow_{wod} \text{-},\text{-}\rangle)$
where *wod-def*: $n \longrightarrow_{wod} n_1,n_2 \equiv ((n_1 \neq n_2) \wedge$
$(\exists\, as.\ (n\ -as\rightarrow*\ n_1) \wedge (n_2 \notin set\ (sourcenodes\ as))) \wedge$
$(\exists\, as.\ (n\ -as\rightarrow*\ n_2) \wedge (n_1 \notin set\ (sourcenodes\ as))) \wedge$
$(\exists\, a.\ (valid\text{-}edge\ a) \wedge (n = sourcenode\ a) \wedge$
$((\exists\, as.\ (targetnode\ a\ -as\rightarrow*\ n_1) \wedge$
$(\forall\, as'.\ (targetnode\ a\ -as'\rightarrow*\ n_2) \longrightarrow n_1 \in set(sourcenodes\ as'))) \vee$
$(\exists\, as.\ (targetnode\ a\ -as\rightarrow*\ n_2) \wedge$
$(\forall\, as'.\ (targetnode\ a\ -as'\rightarrow*\ n_1) \longrightarrow n_2 \in set(sourcenodes\ as'))))))$

inductive-set (**in** *CFG-wf*) *wod-backward-slice* :: $'node\ set \Rightarrow 'node\ set$
for S :: $'node\ set$
where *refl*: $[\![valid\text{-}node\ n;\ n \in S]\!] \Longrightarrow n \in wod\text{-}backward\text{-}slice\ S$

| *cd-closed*:
$[\![n' \longrightarrow_{wod} n_1,n_2;\ n_1 \in wod\text{-}backward\text{-}slice\ S;\ n_2 \in wod\text{-}backward\text{-}slice\ S]\!]$
$\Longrightarrow n' \in wod\text{-}backward\text{-}slice\ S$

| *dd-closed*: $[\![n'\ influences\ V\ in\ n'';\ n'' \in wod\text{-}backward\text{-}slice\ S]\!]$
$\Longrightarrow n' \in wod\text{-}backward\text{-}slice\ S$

lemma (**in** *CFG-wf*)
wod-backward-slice-valid-node: $n \in wod\text{-}backward\text{-}slice\ S \Longrightarrow valid\text{-}node\ n$
by(*induct rule*:*wod-backward-slice.induct*,
auto dest:*path-valid-node simp*:*wod-def data-dependence-def*)

end

3.8 Instantiate framework with control dependences

theory *CDepInstantiations* **imports**
Slice
PDG
WeakOrderDependence
begin

3.8.1 Standard control dependence

context *StandardControlDependencePDG* **begin**

lemma *Exit-in-obs-slice-node*:(-*Exit*-) ∈ *obs n′* (*PDG-BS S*) ⟹ (-*Exit*-) ∈ *S*
by(*auto elim*:*obsE PDG-path-CFG-path simp*:*PDG-BS-def split*:*if-split-asm*)

abbreviation *PDG-path′* :: ′*node* ⇒ ′*node* ⇒ *bool* (‹- ⟶$_d$∗ -› [*51*,*0*] *80*)
where *n* ⟶$_d$∗ *n′* ≡ *PDG.PDG-path sourcenode targetnode valid-edge Def Use standard-control-dependence n n′*

lemma *cd-closed*:
⟦*n′* ∈ *PDG-BS S*; *n controls$_s$ n′*⟧ ⟹ *n* ∈ *PDG-BS S*
by(*simp add*:*PDG-BS-def*)(*blast dest*:*PDG-cdep-edge PDG-path-Append PDG-path-cdep*)

lemma *obs-postdominate*:
assumes *n* ∈ *obs n′* (*PDG-BS S*) **and** *n* ≠ (-*Exit*-) **shows** *n postdominates n′*
proof(*rule ccontr*)
assume ¬ *n postdominates n′*
from ‹*n* ∈ *obs n′* (*PDG-BS S*)› **have** *valid-node n* **by**(*fastforce dest*:*in-obs-valid*)
with ‹*n* ∈ *obs n′* (*PDG-BS S*)› ‹*n* ≠ (-*Exit*-)› **have** *n postdominates n*
by(*fastforce intro*:*postdominate-refl*)
from ‹*n* ∈ *obs n′* (*PDG-BS S*)› **obtain** *as* **where** *n′* −*as*→∗ *n*
and ∀ *n′* ∈ *set*(*sourcenodes as*). *n′* ∉ (*PDG-BS S*)
and *n* ∈ (*PDG-BS S*) **by**(*erule obsE*)
from ‹*n postdominates n*› ‹¬ *n postdominates n′*› ‹*n′* −*as*→∗ *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*)
by −(*erule postdominate-path-branch*)
from ‹¬ *n postdominates* (*sourcenode a*)› ‹*valid-edge a*› ‹*valid-node n*›
obtain *asx* **where** *sourcenode a* −*asx*→∗ (-*Exit*-)
and *n* ∉ *set*(*sourcenodes asx*) **by**(*auto simp*:*postdominate-def*)
from ‹*sourcenode a* −*asx*→∗ (-*Exit*-)› ‹*valid-edge a*›
obtain *ax asx′* **where** [*simp*]:*asx* = *ax*#*asx′*
apply − **apply**(*erule path.cases*)
apply(*drule-tac s*=(-*Exit*-) **in** *sym*)
apply *simp*

    apply(*drule Exit-source*)
    by *simp-all*
  with ‹*sourcenode a* −*asx*→∗ (*-Exit-*)› **have** *sourcenode a* −[]@*ax*#*asx*′→∗ (*-Exit-*)

    by *simp*
  hence *valid-edge ax* **and** [*simp*]:*sourcenode a* = *sourcenode ax*
    and *targetnode ax* −*asx*′→∗ (*-Exit-*)
    by(*fastforce dest*:*path-split*)+
  with ‹*n* ∉ *set*(*sourcenodes asx*)› **have** ¬ *n postdominates targetnode ax*
    by(*fastforce simp*:*postdominate-def sourcenodes-def*)
  from ‹*n* ∈ *obs n*′ (*PDG-BS S*)› ‹∀ *n*′ ∈ *set*(*sourcenodes as*). *n*′ ∉ (*PDG-BS S*)›
  have *n* ∉ *set* (*sourcenodes* (*a*#*as*′′))
    by(*fastforce elim*:*obs.cases simp*:*sourcenodes-def*)
  from ‹*n*′ −*as*→∗ *n*› **have** *sourcenode a* −*a*#*as*′′→∗ *n*
    by(*fastforce dest*:*path-split-second*)
  with ‹*n postdominates* (*targetnode a*)› ‹¬ *n postdominates targetnode ax*›
    ‹*valid-edge ax*› ‹*n* ∉ *set* (*sourcenodes* (*a*#*as*′′))›
  have *sourcenode a* $controls_s$ *n* **by**(*fastforce simp*:*standard-control-dependence-def*)
  with ‹*n* ∈ *obs n*′ (*PDG-BS S*)› **have** *sourcenode a* ∈ (*PDG-BS S*)
    by(*fastforce intro*:*cd-closed PDG-cdep-edge elim*:*obs.cases*)
  with ‹∀ *n*′ ∈ *set*(*sourcenodes as*). *n*′ ∉ (*PDG-BS S*)›
  show *False* **by**(*simp add*:*sourcenodes-def*)
qed

lemma *obs-singleton*:(∃ *m*. *obs n* (*PDG-BS S*) = {*m*}) ∨ *obs n* (*PDG-BS S*) = {}
proof(*rule ccontr*)
  assume ¬ ((∃ *m*. *obs n* (*PDG-BS S*) = {*m*}) ∨ *obs n* (*PDG-BS S*) = {})
  hence ∃ *nx nx*′. *nx* ∈ *obs n* (*PDG-BS S*) ∧ *nx*′ ∈ *obs n* (*PDG-BS S*) ∧
    nx ≠ *nx*′ **by** *auto*
  then obtain *nx nx*′ **where** *nx* ∈ *obs n* (*PDG-BS S*) **and** *nx*′ ∈ *obs n* (*PDG-BS S*)
    and *nx* ≠ *nx*′ **by** *auto*
  from ‹*nx* ∈ *obs n* (*PDG-BS S*)› **obtain** *as* **where** *n* −*as*→∗ *nx*
    and ∀ *n*′ ∈ *set*(*sourcenodes as*). *n*′ ∉ (*PDG-BS S*) **and** *nx* ∈ (*PDG-BS S*)
    by(*erule obsE*)
  from ‹*n* −*as*→∗ *nx*› **have** *valid-node nx* **by**(*fastforce dest*:*path-valid-node*)
  with ‹*nx* ∈ (*PDG-BS S*)› **have** *obs nx* (*PDG-BS S*) = {*nx*} **by** −(*rule n-in-obs*)
  with ‹*n* −*as*→∗ *nx*› ‹*nx* ∈ *obs n* (*PDG-BS S*)› ‹*nx*′ ∈ *obs n* (*PDG-BS S*)› ‹*nx* ≠ *nx*′›
  have *as* ≠ [] **by**(*fastforce elim*:*path.cases*)
  with ‹*n* −*as*→∗ *nx*› ‹*nx* ∈ *obs n* (*PDG-BS S*)› ‹*nx*′ ∈ *obs n* (*PDG-BS S*)›
    ‹*nx* ≠ *nx*′› ‹*obs nx* (*PDG-BS S*) = {*nx*}› ‹∀ *n*′ ∈ *set*(*sourcenodes as*). *n*′ ∉ (*PDG-BS S*)›
  have ∃ *a as*′ *as*′′. *n* −*as*′→∗ *sourcenode a* ∧ *targetnode a* −*as*′′→∗ *nx* ∧
      valid-edge a ∧ *as* = *as*′@*a*#*as*′′ ∧
      obs (*targetnode a*) (*PDG-BS S*) = {*nx*} ∧
      (¬ (∃ *m*. *obs* (*sourcenode a*) (*PDG-BS S*) = {*m*} ∨
        obs (*sourcenode a*) (*PDG-BS S*) = {}))

proof(*induct arbitrary*:*nx′ rule*:*path.induct*)
case (*Cons-path n′′ as n′ a n*)
note [*simp*] = ‹*sourcenode a* = *n*›[*THEN sym*] ‹*targetnode a* = *n′′*›[*THEN sym*]
note *more-than-one* = ‹*n′* ∈ *obs n* (*PDG-BS S*)› ‹*nx′* ∈ *obs n* (*PDG-BS S*)› ‹*n′* ≠ *nx′*›
note *IH* = ‹⋀*nx′*. ⟦*n′* ∈ *obs n′′* (*PDG-BS S*); *nx′* ∈ *obs n′′* (*PDG-BS S*); *n′* ≠ *nx′*;
obs n′ (*PDG-BS S*) = {*n′*}; ∀ *n′*∈*set* (*sourcenodes as*). *n′* ∉ (*PDG-BS S*); *as* ≠ []⟧
⟹ ∃ *a as′ as′′*. *n′′* −*as′*→∗ *sourcenode a* ∧ *targetnode a* −*as′′*→∗ *n′* ∧
valid-edge a ∧ *as* = *as′*@*a*#*as′′* ∧ *obs* (*targetnode a*) (*PDG-BS S*) = {*n′*} ∧
(¬ (∃ *m*. *obs* (*sourcenode a*) (*PDG-BS S*) = {*m*} ∨
obs (*sourcenode a*) (*PDG-BS S*) = {}))›
from ‹∀ *n′*∈*set* (*sourcenodes* (*a*#*as*)). *n′* ∉ (*PDG-BS S*)›
have ∀ *n′*∈*set* (*sourcenodes as*). *n′* ∉ (*PDG-BS S*) **and** *sourcenode a* ∉ (*PDG-BS S*)
by(*simp-all add*:*sourcenodes-def*)
show *?case*
proof(*cases as* = [])
case *True*
with ‹*n′′* −*as*→∗ *n′*› **have** [*simp*]:*n′* = *n′′* **by**(*fastforce elim*:*path.cases*)
from *more-than-one*
have ¬ (∃ *m*. *obs* (*sourcenode a*) (*PDG-BS S*) = {*m*} ∨
obs (*sourcenode a*) (*PDG-BS S*) = {})
by *auto*
with ‹*obs n′* (*PDG-BS S*) = {*n′*}› *True* ‹*valid-edge 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*)
next
case *False*
hence *as* ≠ [] **.**
from ‹*n′′* −*as*→∗ *n′*› ‹∀ *n′*∈*set* (*sourcenodes as*). *n′* ∉ (*PDG-BS S*)›
have *obs n′* (*PDG-BS S*) ⊆ *obs n′′* (*PDG-BS S*) **by**(*rule path-obs-subset*)
show *?thesis*
proof(*cases obs n′* (*PDG-BS S*) = *obs n′′* (*PDG-BS S*))
case *True*
with ‹*n′′* −*as*→∗ *n′*› ‹*valid-edge a*› ‹*obs n′* (*PDG-BS S*) = {*n′*}› *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*)
next
case *False*
with ‹*obs n′* (*PDG-BS S*) ⊆ *obs n′′* (*PDG-BS S*)›
have *obs n′* (*PDG-BS S*) ⊂ *obs n′′* (*PDG-BS S*) **by** *simp*
with ‹*obs n′* (*PDG-BS S*) = {*n′*}› **obtain** *ni* **where** *n′* ∈ *obs n′′* (*PDG-BS*

S)
and $ni \in obs\ n''\ (PDG\text{-}BS\ S)$ **and** $n' \neq ni$ **by** *auto*
from *IH*[*OF this* ‹$obs\ n'\ (PDG\text{-}BS\ S) = \{n'\}$›
‹$\forall n' \in set\ (sourcenodes\ as).\ n' \notin (PDG\text{-}BS\ S)$› ‹$as \neq []$›] **obtain** $a'\ as'\ as''$
where $n''\ -as' \rightarrow *\ sourcenode\ a'$ **and** $targetnode\ a'\ -as'' \rightarrow *\ n'$
and *valid-edge* a' **and** [*simp*]:$as = as'@a'\#as''$
and $obs\ (targetnode\ a')\ (PDG\text{-}BS\ S) = \{n'\}$
and *more-than-one'*:$\neg\ (\exists\ m.\ obs\ (sourcenode\ a')\ (PDG\text{-}BS\ S) = \{m\} \vee$
$obs\ (sourcenode\ a')\ (PDG\text{-}BS\ S) = \{\})$
by *blast*
from ‹$n''\ -as' \rightarrow *\ sourcenode\ a'$› ‹*valid-edge a*›
have $n\ -a\#as' \rightarrow *\ sourcenode\ a'$ **by**(*fastforce intro*:*path.Cons-path*)
with ‹$targetnode\ a'\ -as'' \rightarrow *\ n'$› ‹$obs\ (targetnode\ a')\ (PDG\text{-}BS\ S) = \{n'\}$›
more-than-one' ‹*valid-edge a'*› **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 $obs\ (targetnode\ a)\ (PDG\text{-}BS\ S) = \{nx\}$
and *more-than-one*:$\neg\ (\exists\ m.\ obs\ (sourcenode\ a)\ (PDG\text{-}BS\ S) = \{m\} \vee$
$obs\ (sourcenode\ a)\ (PDG\text{-}BS\ S) = \{\})$
by *blast*
have $sourcenode\ a \notin (PDG\text{-}BS\ S)$
proof(*rule ccontr*)
assume $\neg\ sourcenode\ a \notin PDG\text{-}BS\ S$
hence $sourcenode\ a \in PDG\text{-}BS\ S$ **by** *simp*
with ‹*valid-edge a*› **have** $obs\ (sourcenode\ a)\ (PDG\text{-}BS\ S) = \{sourcenode\ a\}$
by(*fastforce intro*!:*n-in-obs*)
with *more-than-one* **show** *False* **by** *simp*
qed
with ‹*valid-edge a*›
have $obs\ (targetnode\ a)\ (PDG\text{-}BS\ S) \subseteq obs\ (sourcenode\ a)\ (PDG\text{-}BS\ S)$
by(*rule edge-obs-subset*)
with ‹$obs\ (targetnode\ a)\ (PDG\text{-}BS\ S) = \{nx\}$›
have $nx \in obs\ (sourcenode\ a)\ (PDG\text{-}BS\ S)$ **by** *simp*
with *more-than-one* **obtain** *m* **where** $m \in obs\ (sourcenode\ a)\ (PDG\text{-}BS\ S)$
and $nx \neq m$ **by** *auto*
from ‹$m \in obs\ (sourcenode\ a)\ (PDG\text{-}BS\ S)$›
have *valid-node m* **by**(*fastforce dest*:*in-obs-valid*)
from ‹$obs\ (targetnode\ a)\ (PDG\text{-}BS\ S) = \{nx\}$› **have** *valid-node nx*
by(*fastforce dest*:*in-obs-valid*)
show *False*
proof(*cases m postdominates* (*sourcenode a*))
case *True*
with ‹$nx \in obs\ (sourcenode\ a)\ (PDG\text{-}BS\ S)$› ‹$m \in obs\ (sourcenode\ a)\ (PDG\text{-}BS$

S)›
  have *m postdominates nx*
   by(*fastforce intro*:*postdominate-path-targetnode elim*:*obs.cases*)
 with ‹*nx* ≠ *m*› **have** ¬ *nx postdominates m* **by**(*fastforce dest*:*postdominate-antisym*)
  have (*-Exit-*) −[]→∗ (*-Exit-*) **by**(*fastforce intro*:*empty-path*)
  with ‹*m postdominates nx*› **have** *nx* ≠ (*-Exit-*)
   by(*fastforce simp*:*postdominate-def sourcenodes-def*)
  have ¬ *nx postdominates* (*sourcenode a*)
  proof(*rule ccontr*)
   assume ¬ ¬ *nx postdominates sourcenode a*
   hence *nx postdominates sourcenode a* **by** *simp*
    from ‹*m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)› ‹*nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)›
   obtain *asx′* **where** *sourcenode a* −*asx′*→∗ *m* **and** *nx* ∉ *set*(*sourcenodes asx′*)
    by(*fastforce elim*:*obs.cases*)
   with ‹*nx postdominates sourcenode a*› **have** *nx postdominates m*
    by(*rule postdominate-path-targetnode*)
   with ‹¬ *nx postdominates m*› **show** *False* **by** *simp*
  qed
  with ‹*nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)› ‹*valid-node nx*› ‹*nx* ≠ (*-Exit-*)›
  show *False* **by**(*fastforce dest*:*obs-postdominate*)
 next
  case *False*
  show *False*
  proof(*cases m* = *Exit*)
   case *True*
    from ‹*m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)› ‹*nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)›
   obtain *xs* **where** *sourcenode a* −*xs*→∗ *m* **and** *nx* ∉ *set*(*sourcenodes xs*)
    by(*fastforce elim*:*obsE*)
   obtain *x′ xs′* **where** [*simp*]:*xs* = *x′*#*xs′*
   proof(*cases xs*)
    case *Nil*
    with ‹*sourcenode a* −*xs*→∗ *m*› **have** [*simp*]:*sourcenode a* = *m* **by** *fastforce*
    with ‹*m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)›
    have *m* ∈ (*PDG-BS S*) **by**(*metis obsE*)
    with ‹*valid-node m*› **have** *obs m* (*PDG-BS S*) = {*m*}
     by(*rule n-in-obs*)
    with ‹*nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)› ‹*nx* ≠ *m*› **have** *False*
     by *fastforce*
    thus *?thesis* **by** *simp*
   qed *blast*
   from ‹*sourcenode a* −*xs*→∗ *m*› **have** *sourcenode a* = *sourcenode x′*
    and *valid-edge x′* **and** *targetnode x′* −*xs′*→∗ *m*
    by(*auto elim*:*path-split-Cons*)
   from ‹*targetnode x′* −*xs′*→∗ *m*› ‹*nx* ∉ *set*(*sourcenodes xs*)› ‹*valid-edge x′*›
    ‹*valid-node m*› *True*
   have ¬ *nx postdominates* (*targetnode x′*)
    by(*fastforce simp*:*postdominate-def sourcenodes-def*)

```
      from ‹nx ≠ m› True have nx ≠ (-Exit-) by simp
      with ‹obs (targetnode a) (PDG-BS S) = {nx}›
      have nx postdominates (targetnode a)
        by(fastforce intro:obs-postdominate)
      from ‹obs (targetnode a) (PDG-BS S) = {nx}›
      obtain ys where targetnode a −ys→* nx
        and ∀ nx' ∈ set(sourcenodes ys). nx' ∉ (PDG-BS S)
        and nx ∈ (PDG-BS S) by(fastforce elim:obsE)
      hence nx ∉ set(sourcenodes ys)by fastforce
      have sourcenode a ≠ nx
      proof
        assume sourcenode a = nx
        from ‹nx ∈ obs (sourcenode a) (PDG-BS S)›
        have nx ∈ (PDG-BS S) by −(erule obsE)
        with ‹valid-node nx› have obs nx (PDG-BS S) = {nx} by −(erule n-in-obs)
        with ‹sourcenode a = nx› ‹m ∈ obs (sourcenode a) (PDG-BS S)›
          ‹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→* nx›
      have sourcenode a −a#ys→* nx by(fastforce intro:Cons-path)
      from ‹sourcenode a −a#ys→* nx› ‹nx ∉ set(sourcenodes (a#ys))›
        ‹nx postdominates (targetnode a)› ‹valid-edge x'›
        ‹¬ nx postdominates (targetnode x')› ‹sourcenode a = sourcenode x'›
      have (sourcenode a) controls_s nx
        by(fastforce simp:standard-control-dependence-def)
      with ‹nx ∈ (PDG-BS S)› have sourcenode a ∈ (PDG-BS S)
        by(rule cd-closed)
      with ‹valid-edge a› have obs (sourcenode a) (PDG-BS S) = {sourcenode a}
        by(fastforce intro!:n-in-obs)
      with ‹m ∈ obs (sourcenode a) (PDG-BS S)›
        ‹nx ∈ obs (sourcenode a) (PDG-BS S)› ‹nx ≠ m›
      show False by simp
    next
      case False
      with ‹m ∈ obs (sourcenode a) (PDG-BS S)› ‹valid-node m›
        ‹¬ m postdominates sourcenode a›
      show False by(fastforce dest:obs-postdominate)
    qed
  qed
qed


lemma PDGBackwardSliceCorrect:
  BackwardSlice sourcenode targetnode kind valid-edge
        (-Entry-) Def Use state-val PDG-BS
proof(unfold-locales)
  fix n S assume n ∈ PDG-BS S
```

thus *valid-node n* **by**(*rule PDG-BS-valid-node*)
next
fix *n S* **assume** *valid-node n* **and** $n \in S$
thus $n \in$ *PDG-BS S* **by**(*fastforce intro*:*PDG-path-Nil simp*:*PDG-BS-def*)
next
fix *n′ S n V*
assume $n' \in$ *PDG-BS S* **and** *n influences V in n′*
thus $n \in$ *PDG-BS S*
by(*auto dest*:*PDG.PDG-path-ddep*[*OF PDG-scd*,*OF PDG.PDG-ddep-edge*[*OF PDG-scd*]]
dest:*PDG-path-Append simp*:*PDG-BS-def split*:*if-split-asm*)
next
fix *n S*
have ($\exists$ *m. obs n* (*PDG-BS S*) = {*m*}) $\vee$ *obs n* (*PDG-BS S*) = {}
by(*rule obs-singleton*)
thus *finite* (*obs n* (*PDG-BS S*)) **by** *fastforce*
next
fix *n S*
have ($\exists$ *m. obs n* (*PDG-BS S*) = {*m*}) $\vee$ *obs n* (*PDG-BS S*) = {}
by(*rule obs-singleton*)
thus *card* (*obs n* (*PDG-BS S*)) $\leq$ *1* **by** *fastforce*
qed

end

3.8.2 Weak control dependence

context *WeakControlDependencePDG* **begin**

lemma *Exit-in-obs-slice-node*:(-*Exit*-) $\in$ *obs n′* (*PDG-BS S*) $\Longrightarrow$ (-*Exit*-) $\in$ *S*
by(*auto elim*:*obsE PDG-path-CFG-path simp*:*PDG-BS-def split*:*if-split-asm*)

lemma *cd-closed*:
⟦$n' \in$ *PDG-BS S*; *n weakly controls n′*⟧ $\Longrightarrow$ $n \in$ *PDG-BS S*
by(*simp add*:*PDG-BS-def*)(*blast dest*:*PDG-cdep-edge PDG-path-Append PDG-path-cdep*)

lemma *obs-strong-postdominate*:
assumes $n \in$ *obs n′* (*PDG-BS S*) **and** $n \neq$ (-*Exit*-)
shows *n strongly−postdominates n′*
proof(*rule ccontr*)
assume ¬ *n strongly−postdominates n′*
from ‹$n \in$ *obs n′* (*PDG-BS S*)› **have** *valid-node n* **by**(*fastforce dest*:*in-obs-valid*)
with ‹$n \in$ *obs n′* (*PDG-BS S*)› ‹$n \neq$ (-*Exit*-)› **have** *n strongly−postdominates n*
by(*fastforce intro*:*strong-postdominate-refl*)
from ‹$n \in$ *obs n′* (*PDG-BS S*)› **obtain** *as* **where** *n′ −as→∗ n*
and $\forall n' \in$ *set*(*sourcenodes as*). $n' \notin$ (*PDG-BS S*)
and $n \in$ (*PDG-BS S*) **by**(*erule obsE*)

from ‹*n strongly−postdominates n*› ‹¬ *n strongly−postdominates n′*› ‹*n′ −as→∗ n*›
obtain *as′ a as″* **where** [*simp*]:*as* = *as′*@*a*#*as″* **and** *valid-edge a*
and ¬ *n strongly−postdominates* (*sourcenode a*) **and**
n strongly−postdominates (*targetnode a*)
by −(*erule strong-postdominate-path-branch*)
from ‹*n* ∈ *obs n′* (*PDG-BS S*)› ‹∀ *n′* ∈ *set*(*sourcenodes as*). *n′* ∉ (*PDG-BS S*)›
have *n* ∉ *set* (*sourcenodes* (*a*#*as″*))
by(*fastforce elim*:*obs.cases simp*:*sourcenodes-def*)
from ‹*n′ −as→∗ n*› **have** *sourcenode a −a*#*as″*→∗ *n*
by(*fastforce dest*:*path-split-second*)
from ‹¬ *n strongly−postdominates* (*sourcenode a*)› ‹*valid-edge a*› ‹*valid-node n*›
obtain *a′* **where** *sourcenode a′* = *sourcenode a*
and *valid-edge a′* **and** ¬ *n strongly−postdominates* (*targetnode a′*)
by(*fastforce elim*:*not-strong-postdominate-predecessor-successor*)
with ‹*n strongly−postdominates* (*targetnode a*)› ‹*n* ∉ *set* (*sourcenodes* (*a*#*as″*))›
‹*sourcenode a −a*#*as″*→∗ *n*›
have *sourcenode a weakly controls n*
by(*fastforce simp*:*weak-control-dependence-def*)
with ‹*n* ∈ *obs n′* (*PDG-BS S*)› **have** *sourcenode a* ∈ (*PDG-BS S*)
by(*fastforce intro*:*cd-closed PDG-cdep-edge elim*:*obs.cases*)
with ‹∀ *n′* ∈ *set*(*sourcenodes as*). *n′* ∉ (*PDG-BS S*)›
show *False* **by**(*simp add*:*sourcenodes-def*)
qed

lemma *obs-singleton*:(∃ *m*. *obs n* (*PDG-BS S*) = {*m*}) ∨ *obs n* (*PDG-BS S*) = {}
proof(*rule ccontr*)
assume ¬ ((∃ *m*. *obs n* (*PDG-BS S*) = {*m*}) ∨ *obs n* (*PDG-BS S*) = {})
hence ∃ *nx nx′*. *nx* ∈ *obs n* (*PDG-BS S*) ∧ *nx′* ∈ *obs n* (*PDG-BS S*) ∧
nx ≠ *nx′* **by** *auto*
then obtain *nx nx′* **where** *nx* ∈ *obs n* (*PDG-BS S*) **and** *nx′* ∈ *obs n* (*PDG-BS S*)
and *nx* ≠ *nx′* **by** *auto*
from ‹*nx* ∈ *obs n* (*PDG-BS S*)› **obtain** *as* **where** *n −as→∗ nx*
and ∀ *n′* ∈ *set*(*sourcenodes as*). *n′* ∉ (*PDG-BS S*) **and** *nx* ∈ (*PDG-BS S*)
by(*erule obsE*)
from ‹*n −as→∗ nx*› **have** *valid-node nx* **by**(*fastforce dest*:*path-valid-node*)
with ‹*nx* ∈ (*PDG-BS S*)› **have** *obs nx* (*PDG-BS S*) = {*nx*} **by** −(*rule n-in-obs*)
with ‹*n −as→∗ nx*› ‹*nx* ∈ *obs n* (*PDG-BS S*)› ‹*nx′* ∈ *obs n* (*PDG-BS S*)› ‹*nx* ≠ *nx′*›
have *as* ≠ [] **by**(*fastforce elim*:*path.cases*)
with ‹*n −as→∗ nx*› ‹*nx* ∈ *obs n* (*PDG-BS S*)› ‹*nx′* ∈ *obs n* (*PDG-BS S*)›
‹*nx* ≠ *nx′*› ‹*obs nx* (*PDG-BS S*) = {*nx*}› ‹∀ *n′* ∈ *set*(*sourcenodes as*). *n′* ∉ (*PDG-BS S*)›
have ∃ *a as′ as″*. *n −as′→∗ sourcenode a* ∧ *targetnode a −as″→∗ nx* ∧
valid-edge a ∧ *as* = *as′*@*a*#*as″* ∧
obs (*targetnode a*) (*PDG-BS S*) = {*nx*} ∧
(¬ (∃ *m*. *obs* (*sourcenode a*) (*PDG-BS S*) = {*m*} ∨

obs (*sourcenode a*) (*PDG-BS S*) = {}))
proof(*induct arbitrary*:*nx'* *rule*:*path.induct*)
case (*Cons-path n'' as n' a n*)
note [*simp*] = ‹*sourcenode a* = *n*›[*THEN sym*] ‹*targetnode a* = *n''*›[*THEN sym*]
note *more-than-one* = ‹*n'* ∈ *obs n* (*PDG-BS S*)› ‹*nx'* ∈ *obs n* (*PDG-BS S*)› ‹*n'* ≠ *nx'*›
note *IH* = ‹⋀*nx'*. ⟦*n'* ∈ *obs n''* (*PDG-BS S*); *nx'* ∈ *obs n''* (*PDG-BS S*); *n'* ≠ *nx'*;
obs n' (*PDG-BS S*) = {*n'*}; ∀ *n'*∈*set* (*sourcenodes as*). *n'* ∉ (*PDG-BS S*); *as* ≠ []⟧
⟹ ∃ *a as' as''*. *n''* −*as'*→∗ *sourcenode a* ∧ *targetnode a* −*as''*→∗ *n'* ∧
valid-edge a ∧ *as* = *as'*@*a*#*as''* ∧ *obs* (*targetnode a*) (*PDG-BS S*) = {*n'*} ∧
(¬ (∃ *m*. *obs* (*sourcenode a*) (*PDG-BS S*) = {*m*} ∨
obs (*sourcenode a*) (*PDG-BS S*) = {}))›
from ‹∀ *n'*∈*set* (*sourcenodes* (*a*#*as*)). *n'* ∉ (*PDG-BS S*)›
have ∀ *n'*∈*set* (*sourcenodes as*). *n'* ∉ (*PDG-BS S*) **and** *sourcenode a* ∉ (*PDG-BS S*)
by(*simp-all add*:*sourcenodes-def*)
show *?case*
proof(*cases as* = [])
case *True*
with ‹*n''* −*as*→∗ *n'*› **have** [*simp*]:*n'* = *n''* **by**(*fastforce elim*:*path.cases*)
from *more-than-one*
have ¬ (∃ *m*. *obs* (*sourcenode a*) (*PDG-BS S*) = {*m*} ∨
obs (*sourcenode a*) (*PDG-BS S*) = {})
by *auto*
with ‹*obs n'* (*PDG-BS S*) = {*n'*}› *True* ‹*valid-edge 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*)
next
case *False*
hence *as* ≠ [] **.**
from ‹*n''* −*as*→∗ *n'*› ‹∀ *n'*∈*set* (*sourcenodes as*). *n'* ∉ (*PDG-BS S*)›
have *obs n'* (*PDG-BS S*) ⊆ *obs n''* (*PDG-BS S*) **by**(*rule path-obs-subset*)
show *?thesis*
proof(*cases obs n'* (*PDG-BS S*) = *obs n''* (*PDG-BS S*))
case *True*
with ‹*n''* −*as*→∗ *n'*› ‹*valid-edge a*› ‹*obs n'* (*PDG-BS S*) = {*n'*}› *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*)
next
case *False*
with ‹*obs n'* (*PDG-BS S*) ⊆ *obs n''* (*PDG-BS S*)›
have *obs n'* (*PDG-BS S*) ⊂ *obs n''* (*PDG-BS S*) **by** *simp*

with ‹*obs n′* (*PDG-BS S*) = {*n′*}› **obtain** *ni* **where** *n′* ∈ *obs n″* (*PDG-BS S*)
and *ni* ∈ *obs n″* (*PDG-BS S*) **and** *n′* ≠ *ni* **by** *auto*
from *IH*[*OF this* ‹*obs n′* (*PDG-BS S*) = {*n′*}›
‹∀ *n′*∈*set* (*sourcenodes as*). *n′* ∉ (*PDG-BS S*)› ‹*as* ≠ []›] **obtain** *a′ as′ as″*
where *n″* −*as′*→∗ *sourcenode a′* **and** *targetnode a′* −*as″*→∗ *n′*
and *valid-edge a′* **and** [*simp*]:*as* = *as′*@*a′*#*as″*
and *obs* (*targetnode a′*) (*PDG-BS S*) = {*n′*}
and *more-than-one′*:¬ (∃ *m*. *obs* (*sourcenode a′*) (*PDG-BS S*) = {*m*} ∨
obs (*sourcenode a′*) (*PDG-BS S*) = {})
by *blast*
from ‹*n″* −*as′*→∗ *sourcenode a′*› ‹*valid-edge a*›
have *n* −*a*#*as′*→∗ *sourcenode a′* **by**(*fastforce intro*:*path.Cons-path*)
with ‹*targetnode a′* −*as″*→∗ *n′*› ‹*obs* (*targetnode a′*) (*PDG-BS S*) = {*n′*}›
more-than-one′ ‹*valid-edge a′*› **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 *obs* (*targetnode a*) (*PDG-BS S*) = {*nx*}
and *more-than-one*:¬ (∃ *m*. *obs* (*sourcenode a*) (*PDG-BS S*) = {*m*} ∨
obs (*sourcenode a*) (*PDG-BS S*) = {})
by *blast*
have *sourcenode a* ∉ (*PDG-BS S*)
proof(*rule ccontr*)
assume ¬ *sourcenode a* ∉ *PDG-BS S*
hence *sourcenode a* ∈ *PDG-BS S* **by** *simp*
with ‹*valid-edge a*› **have** *obs* (*sourcenode a*) (*PDG-BS S*) = {*sourcenode a*}
by(*fastforce intro*!:*n-in-obs*)
with *more-than-one* **show** *False* **by** *simp*
qed
with ‹*valid-edge a*›
have *obs* (*targetnode a*) (*PDG-BS S*) ⊆ *obs* (*sourcenode a*) (*PDG-BS S*)
by(*rule edge-obs-subset*)
with ‹*obs* (*targetnode a*) (*PDG-BS S*) = {*nx*}›
have *nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*) **by** *simp*
with *more-than-one* **obtain** *m* **where** *m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)
and *nx* ≠ *m* **by** *auto*
from ‹*m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)›
have *valid-node m* **by**(*fastforce dest*:*in-obs-valid*)
from ‹*obs* (*targetnode a*) (*PDG-BS S*) = {*nx*}› **have** *valid-node nx*
by(*fastforce dest*:*in-obs-valid*)
show *False*
proof(*cases m strongly*−*postdominates* (*sourcenode a*))
case *True*

with ‹*nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)› ‹*m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)›
have *m strongly−postdominates nx*
by(*fastforce intro*:*strong-postdominate-path-targetnode elim*:*obs.cases*)
with ‹*nx* ≠ *m*› **have** ¬ *nx strongly−postdominates m*
by(*fastforce dest*:*strong-postdominate-antisym*)
have (*-Exit-*) −[]→∗ (*-Exit-*) **by**(*fastforce intro*:*empty-path*)
with ‹*m strongly−postdominates nx*› **have** *nx* ≠ (*-Exit-*)
by(*fastforce simp*:*strong-postdominate-def sourcenodes-def postdominate-def*)
have ¬ *nx strongly−postdominates* (*sourcenode a*)
proof(*rule ccontr*)
assume ¬ ¬ *nx strongly−postdominates sourcenode a*
hence *nx strongly−postdominates sourcenode a* **by** *simp*
from ‹*m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)› ‹*nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)›
obtain *asx′* **where** *sourcenode a* −*asx′*→∗ *m* **and** *nx* ∉ *set*(*sourcenodes asx′*)
by(*fastforce elim*:*obs.cases*)
with ‹*nx strongly−postdominates sourcenode a*› **have** *nx strongly−postdominates m*
by(*rule strong-postdominate-path-targetnode*)
with ‹¬ *nx strongly−postdominates m*› **show** *False* **by** *simp*
qed
with ‹*nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)› ‹*valid-node nx*› ‹*nx* ≠ (*-Exit-*)›
show *False* **by**(*fastforce dest*:*obs-strong-postdominate*)
next
case *False*
show *False*
proof(*cases m* = *Exit*)
case *True*
from ‹*m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)› ‹*nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)›
obtain *xs* **where** *sourcenode a* −*xs*→∗ *m* **and** *nx* ∉ *set*(*sourcenodes xs*)
by(*fastforce elim*:*obsE*)
obtain *x′ xs′* **where** [*simp*]:*xs* = *x′*#*xs′*
proof(*cases xs*)
case *Nil*
with ‹*sourcenode a* −*xs*→∗ *m*› **have** [*simp*]:*sourcenode a* = *m* **by** *fastforce*
with ‹*m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)›
have *m* ∈ (*PDG-BS S*) **by** (*metis obsE*)
with ‹*valid-node m*› **have** *obs m* (*PDG-BS S*) = {*m*}
by(*rule n-in-obs*)
with ‹*nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)› ‹*nx* ≠ *m*› **have** *False*
by *fastforce*
thus *?thesis* **by** *simp*
qed *blast*
from ‹*sourcenode a* −*xs*→∗ *m*› **have** *sourcenode a* = *sourcenode x′*
and *valid-edge x′* **and** *targetnode x′* −*xs′*→∗ *m*
by(*auto elim*:*path-split-Cons*)
from ‹*targetnode x′* −*xs′*→∗ *m*› ‹*nx* ∉ *set*(*sourcenodes xs*)› ‹*valid-edge x′*›

‹*valid-node m*› *True*
have ¬ *nx strongly−postdominates* (*targetnode x'*)
by(*fastforce simp*:*strong-postdominate-def postdominate-def sourcenodes-def*)
from ‹*nx* ≠ *m*› *True* **have** *nx* ≠ (*-Exit-*) **by** *simp*
with ‹*obs* (*targetnode a*) (*PDG-BS S*) = {*nx*}›
have *nx strongly−postdominates* (*targetnode a*)
by(*fastforce intro*:*obs-strong-postdominate*)
from ‹*obs* (*targetnode a*) (*PDG-BS S*) = {*nx*}›
obtain *ys* **where** *targetnode a* −*ys*→∗ *nx*
and ∀ *nx'* ∈ *set*(*sourcenodes ys*). *nx'* ∉ (*PDG-BS S*)
and *nx* ∈ (*PDG-BS S*) **by**(*fastforce elim*:*obsE*)
hence *nx* ∉ *set*(*sourcenodes ys*)**by** *fastforce*
have *sourcenode a* ≠ *nx*
proof
assume *sourcenode a* = *nx*
from ‹*nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)›
have *nx* ∈ (*PDG-BS S*) **by** −(*erule obsE*)
with ‹*valid-node nx*› **have** *obs nx* (*PDG-BS S*) = {*nx*} **by** −(*erule n-in-obs*)
with ‹*sourcenode a* = *nx*› ‹*m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)›
‹*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*→∗ *nx*›
have *sourcenode a* −*a*#*ys*→∗ *nx* **by**(*fastforce intro*:*Cons-path*)
from ‹*sourcenode a* −*a*#*ys*→∗ *nx*› ‹*nx* ∉ *set*(*sourcenodes* (*a*#*ys*))›
‹*nx strongly−postdominates* (*targetnode a*)› ‹*valid-edge x'*›
‹¬ *nx strongly−postdominates* (*targetnode x'*)› ‹*sourcenode a* = *sourcenode x'*›
have (*sourcenode a*) *weakly controls nx*
by(*fastforce simp*:*weak-control-dependence-def*)
with ‹*nx* ∈ (*PDG-BS S*)› **have** *sourcenode a* ∈ (*PDG-BS S*)
by(*rule cd-closed*)
with ‹*valid-edge a*› **have** *obs* (*sourcenode a*) (*PDG-BS S*) = {*sourcenode a*}
by(*fastforce intro*!:*n-in-obs*)
with ‹*m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)›
‹*nx* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)› ‹*nx* ≠ *m*›
show *False* **by** *simp*
next
case *False*
with ‹*m* ∈ *obs* (*sourcenode a*) (*PDG-BS S*)› ‹*valid-node m*›
‹¬ *m strongly−postdominates sourcenode a*›
show *False* **by**(*fastforce dest*:*obs-strong-postdominate*)
qed
qed
qed

lemma *WeakPDGBackwardSliceCorrect*:

BackwardSlice sourcenode targetnode kind valid-edge
(-Entry-) Def Use state-val PDG-BS
proof(*unfold-locales*)
fix *n S* **assume** *n* ∈ *PDG-BS S*
thus *valid-node n* **by**(*rule PDG-BS-valid-node*)
next
fix *n S* **assume** *valid-node n* **and** *n* ∈ *S*
thus *n* ∈ *PDG-BS S* **by**(*fastforce intro*:*PDG-path-Nil simp*:*PDG-BS-def*)
next
fix *n′ S n V* **assume** *n′* ∈ *PDG-BS S* **and** *n influences V in n′*
thus *n* ∈ *PDG-BS S*
by(*auto dest*:*PDG.PDG-path-ddep*[*OF PDG-wcd*,*OF PDG.PDG-ddep-edge*[*OF PDG-wcd*]]
dest:*PDG-path-Append simp*:*PDG-BS-def split*:*if-split-asm*)
next
fix *n S*
have (∃ *m. obs n* (*PDG-BS S*) = {*m*}) ∨ *obs n* (*PDG-BS S*) = {}
by(*rule obs-singleton*)
thus *finite* (*obs n* (*PDG-BS S*)) **by** *fastforce*
next
fix *n S*
have (∃ *m. obs n* (*PDG-BS S*) = {*m*}) ∨ *obs n* (*PDG-BS S*) = {}
by(*rule obs-singleton*)
thus *card* (*obs n* (*PDG-BS S*)) ≤ *1* **by** *fastforce*
qed

end

3.8.3 Weak order dependence

context *CFG-wf* **begin**

lemma *obs-singleton*:

shows (∃ *m. obs n* (*wod-backward-slice S*) = {*m*}) ∨
obs n (*wod-backward-slice S*) = {}
proof(*rule ccontr*)
let *?WOD-BS* = *wod-backward-slice S*
assume ¬ ((∃ *m. obs n ?WOD-BS* = {*m*}) ∨ *obs n ?WOD-BS* = {})
hence ∃ *nx nx′. nx* ∈ *obs n ?WOD-BS* ∧ *nx′* ∈ *obs n ?WOD-BS* ∧
nx ≠ *nx′* **by** *auto*
then obtain *nx nx′* **where** *nx* ∈ *obs n ?WOD-BS* **and** *nx′* ∈ *obs n ?WOD-BS*
and *nx* ≠ *nx′* **by** *auto*
from ‹*nx* ∈ *obs n ?WOD-BS*› **obtain** *as* **where** *n −as→∗ nx*
and ∀ *n′* ∈ *set*(*sourcenodes as*). *n′* ∉ *?WOD-BS* **and** *nx* ∈ *?WOD-BS*
by(*erule obsE*)
from ‹*n −as→∗ nx*› **have** *valid-node nx* **by**(*fastforce dest*:*path-valid-node*)
with ‹*nx* ∈ *?WOD-BS*› **have** *obs nx ?WOD-BS* = {*nx*} **by** −(*rule n-in-obs*)
with ‹*n −as→∗ nx*› ‹*nx* ∈ *obs n ?WOD-BS*› ‹*nx′* ∈ *obs n ?WOD-BS*› ‹*nx* ≠

nx′›
have *as* ≠ [] **by**(*fastforce elim*:*path.cases*)
with ‹*n* −*as*→∗ *nx*› ‹*nx* ∈ *obs n ?WOD-BS*› ‹*nx*′ ∈ *obs n ?WOD-BS*› ‹*nx* ≠ *nx*′›
‹*obs nx ?WOD-BS* = {*nx*}› ‹∀ *n*′ ∈ *set*(*sourcenodes as*). *n*′ ∉ *?WOD-BS*›
have ∃ *a as*′ *as*′′. *n* −*as*′→∗ *sourcenode a* ∧ *targetnode a* −*as*′′→∗ *nx* ∧
valid-edge a ∧ *as* = *as*′@*a*#*as*′′ ∧
obs (*targetnode a*) *?WOD-BS* = {*nx*} ∧
(¬ (∃ *m*. *obs* (*sourcenode a*) *?WOD-BS* = {*m*} ∨
obs (*sourcenode a*) *?WOD-BS* = {}))
proof(*induct arbitrary*:*nx*′ *rule*:*path.induct*)
case (*Cons-path n*′′ *as n*′ *a n*)
note [*simp*] = ‹*sourcenode a* = *n*›[*THEN sym*] ‹*targetnode a* = *n*′′›[*THEN sym*]
note *more-than-one* = ‹*n*′ ∈ *obs n* (*?WOD-BS*)› ‹*nx*′ ∈ *obs n* (*?WOD-BS*)› ‹*n*′ ≠ *nx*′›
note *IH* = ‹⋀*nx*′. ⟦*n*′ ∈ *obs n*′′ (*?WOD-BS*); *nx*′ ∈ *obs n*′′ (*?WOD-BS*); *n*′ ≠ *nx*′;
obs n′ (*?WOD-BS*) = {*n*′}; ∀ *n*′∈*set* (*sourcenodes as*). *n*′ ∉ (*?WOD-BS*); *as* ≠ []⟧
⟹ ∃ *a as*′ *as*′′. *n*′′ −*as*′→∗ *sourcenode a* ∧ *targetnode a* −*as*′′→∗ *n*′ ∧
valid-edge a ∧ *as* = *as*′@*a*#*as*′′ ∧ *obs* (*targetnode a*) (*?WOD-BS*) = {*n*′} ∧
(¬ (∃ *m*. *obs* (*sourcenode a*) (*?WOD-BS*) = {*m*} ∨
obs (*sourcenode a*) (*?WOD-BS*) = {}))›
from ‹∀ *n*′∈*set* (*sourcenodes* (*a*#*as*)). *n*′ ∉ (*?WOD-BS*)›
have ∀ *n*′∈*set* (*sourcenodes as*). *n*′ ∉ (*?WOD-BS*) **and** *sourcenode a* ∉ (*?WOD-BS*)
by(*simp-all add*:*sourcenodes-def*)
show *?case*
proof(*cases as* = [])
case *True*
with ‹*n*′′ −*as*→∗ *n*′› **have** [*simp*]:*n*′ = *n*′′ **by**(*fastforce elim*:*path.cases*)
from *more-than-one*
have ¬ (∃ *m*. *obs* (*sourcenode a*) (*?WOD-BS*) = {*m*} ∨
obs (*sourcenode a*) (*?WOD-BS*) = {})
by *auto*
with ‹*obs n*′ (*?WOD-BS*) = {*n*′}› *True* ‹*valid-edge 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*)
next
case *False*
hence *as* ≠ [] **.**
from ‹*n*′′ −*as*→∗ *n*′› ‹∀ *n*′∈*set* (*sourcenodes as*). *n*′ ∉ (*?WOD-BS*)›
have *obs n*′ (*?WOD-BS*) ⊆ *obs n*′′ (*?WOD-BS*) **by**(*rule path-obs-subset*)
show *?thesis*
proof(*cases obs n*′ (*?WOD-BS*) = *obs n*′′ (*?WOD-BS*))
case *True*
with ‹*n*′′ −*as*→∗ *n*′› ‹*valid-edge a*› ‹*obs n*′ (*?WOD-BS*) = {*n*′}› *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)
    next
      case False
      with ‹obs n′ (?WOD-BS) ⊆ obs n″ (?WOD-BS)›
      have obs n′ (?WOD-BS) ⊂ obs n″ (?WOD-BS) by simp
      with ‹obs n′ (?WOD-BS) = {n′}› obtain ni where n′ ∈ obs n″ (?WOD-BS)
        and ni ∈ obs n″ (?WOD-BS) and n′ ≠ ni by auto
      from IH[OF this ‹obs n′ (?WOD-BS) = {n′}›
        ‹∀ n′∈set (sourcenodes as). n′ ∉ (?WOD-BS)› ‹as ≠ []›] obtain a′ as′ as″
        where n″ −as′→* sourcenode a′ and targetnode a′ −as″→* n′
        and valid-edge a′ and [simp]:as = as′@a′#as″
        and obs (targetnode a′) (?WOD-BS) = {n′}
        and more-than-one′:¬ (∃ m. obs (sourcenode a′) (?WOD-BS) = {m} ∨
        obs (sourcenode a′) (?WOD-BS) = {})
        by blast
      from ‹n″ −as′→* sourcenode a′› ‹valid-edge a›
      have n −a#as′→* sourcenode a′ by(fastforce intro:path.Cons-path)
      with ‹targetnode a′ −as″→* n′› ‹obs (targetnode a′) (?WOD-BS) = {n′}›
        more-than-one′ ‹valid-edge a′› 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 obs (targetnode a) (?WOD-BS) = {nx}
  and more-than-one:¬ (∃ m. obs (sourcenode a) (?WOD-BS) = {m} ∨
                   obs (sourcenode a) (?WOD-BS) = {})
  by blast
have sourcenode a ∉ (?WOD-BS)
proof(rule ccontr)
  assume ¬ sourcenode a ∉ ?WOD-BS
  hence sourcenode a ∈ ?WOD-BS by simp
  with ‹valid-edge a› have obs (sourcenode a) (?WOD-BS) = {sourcenode a}
    by(fastforce intro!:n-in-obs)
  with more-than-one show False by simp
qed
with ‹valid-edge a›
have obs (targetnode a) (?WOD-BS) ⊆ obs (sourcenode a) (?WOD-BS)
  by(rule edge-obs-subset)
with ‹obs (targetnode a) (?WOD-BS) = {nx}›
have nx ∈ obs (sourcenode a) (?WOD-BS) by simp
with more-than-one obtain m where m ∈ obs (sourcenode a) (?WOD-BS)
  and nx ≠ m by auto
```

with ‹$nx \in obs$ (*sourcenode a*) (*?WOD-BS*)› **obtain** *as2*
where *sourcenode a* $-as2\rightarrow*$ *m* **and** $nx \notin set$(*sourcenodes as2*)
by(*fastforce elim*:*obsE*)
from ‹$nx \in obs$ (*sourcenode a*) (*?WOD-BS*)› ‹$m \in obs$ (*sourcenode a*) (*?WOD-BS*)›

obtain *as1* **where** *sourcenode a* $-as1\rightarrow*$ *nx* **and** $m \notin set$(*sourcenodes as1*)
by(*fastforce elim*:*obsE*)
from ‹*obs* (*targetnode a*) (*?WOD-BS*) = {*nx*}› **obtain** *asx*
where *targetnode a* $-asx\rightarrow*$ *nx* **by**(*fastforce elim*:*obsE*)
have ∀ *asx'*. *targetnode a* $-asx'\rightarrow*$ *m* ⟶ $nx \in set$(*sourcenodes asx'*)
proof(*rule ccontr*)
assume ¬ (∀ *asx'*. *targetnode a* $-asx'\rightarrow*$ *m* ⟶ $nx \in set$ (*sourcenodes asx'*))
then obtain *asx'* **where** *targetnode a* $-asx'\rightarrow*$ *m* **and** $nx \notin set$ (*sourcenodes asx'*)
by *blast*
show *False*
proof(*cases* ∀ $nx \in set$(*sourcenodes asx'*). $nx \notin$ *?WOD-BS*)
case *True*
with ‹*targetnode a* $-asx'\rightarrow*$ *m*› ‹$m \in obs$ (*sourcenode a*) (*?WOD-BS*)›
have $m \in obs$ (*targetnode a*) *?WOD-BS* **by**(*fastforce intro*:*obs-elem elim*:*obsE*)
with ‹$nx \neq m$› ‹*obs* (*targetnode a*) (*?WOD-BS*) = {*nx*}› **show** *False* **by** *simp*
next
case *False*
hence ∃ $nx \in set$(*sourcenodes asx'*). $nx \in$ *?WOD-BS* **by** *blast*
then obtain *nx' ns ns'* **where** *sourcenodes asx'* = *ns*@*nx'*#*ns'* **and** $nx' \in$ *?WOD-BS*
and ∀ $nx \in set\ ns$. $nx \notin$ *?WOD-BS* **by**(*fastforce elim*!:*split-list-first-propE*)
from ‹*sourcenodes asx'* = *ns*@*nx'*#*ns'*› **obtain** *ax ai ai'*
where [*simp*]:*asx'* = *ai*@*ax*#*ai'* *ns* = *sourcenodes ai* *nx'* = *sourcenode ax*
by(*fastforce elim*:*map-append-append-maps simp*:*sourcenodes-def*)
from ‹*targetnode a* $-asx'\rightarrow*$ *m*› **have** *targetnode a* $-ai\rightarrow*$ *sourcenode ax*
by(*fastforce dest*:*path-split*)
with ‹$nx' \in$ *?WOD-BS*› ‹∀ $nx \in set\ ns$. $nx \notin$ *?WOD-BS*›
have $nx' \in obs$ (*targetnode a*) *?WOD-BS* **by**(*fastforce intro*:*obs-elem*)
with ‹*obs* (*targetnode a*) (*?WOD-BS*) = {*nx*}› **have** *nx'* = *nx* **by** *simp*
with ‹$nx \notin set$ (*sourcenodes asx'*)› **show** *False* **by**(*simp add*:*sourcenodes-def*)
qed
qed
with ‹$nx \neq m$› ‹*sourcenode a* $-as1\rightarrow*$ *nx*› ‹$m \notin set$(*sourcenodes as1*)›
‹*sourcenode a* $-as2\rightarrow*$ *m*› ‹$nx \notin set$(*sourcenodes as2*)› ‹*valid-edge a*›
‹*targetnode a* $-asx\rightarrow*$ *nx*›
have *sourcenode a* $\longrightarrow_{wod}$ *nx*,*m* **by**(*simp add*:*wod-def*,*blast*)
with ‹$nx \in obs$ (*sourcenode a*) (*?WOD-BS*)› ‹$m \in obs$ (*sourcenode a*) (*?WOD-BS*)›

have *sourcenode a* ∈ *?WOD-BS* **by**(*fastforce elim*:*cd-closed elim*:*obsE*)
with ‹*sourcenode a* ∉ *?WOD-BS*› **show** *False* **by** *simp*
qed

lemma *WODBackwardSliceCorrect*:
BackwardSlice sourcenode targetnode kind valid-edge
(-Entry-) Def Use state-val wod-backward-slice
proof(*unfold-locales*)
fix *n S* **assume** $n \in$ *wod-backward-slice S*
thus *valid-node n* **by**(*rule wod-backward-slice-valid-node*)
next
fix *n S* **assume** *valid-node n* **and** $n \in S$
thus $n \in$ *wod-backward-slice S* **by**(*rule refl*)
next
fix *n' S n V* **assume** $n' \in$ *wod-backward-slice S n influences V in n'*
thus $n \in$ *wod-backward-slice S*
by −(*rule dd-closed*)
next
fix *n S*
have ($\exists$ *m. obs n* (*wod-backward-slice S*) = {*m*}) $\vee$
obs n (*wod-backward-slice S*) = {}
by(*rule obs-singleton*)
thus *finite* (*obs n* (*wod-backward-slice S*)) **by** *fastforce*
next
fix *n S*
have ($\exists$ *m. obs n* (*wod-backward-slice S*) = {*m*}) $\vee$ *obs n* (*wod-backward-slice S*) = {}
by(*rule obs-singleton*)
thus *card* (*obs n* (*wod-backward-slice S*)) $\leq$ *1* **by** *fastforce*
qed

end

end

3.9 Relations between control dependences

theory *ControlDependenceRelations*
imports *WeakOrderDependence StandardControlDependence*
begin

context *StrongPostdomination* **begin**

lemma *standard-control-implies-weak-order*:
assumes *n* $controls_s$ *n'* **shows** $n \longrightarrow_{wod} n'$,(*-Exit-*)
proof −
from ‹*n* $controls_s$ *n'*› **obtain** *as a a' as'* **where** *as* = *a#as'*
and $n' \notin$ *set*(*sourcenodes as*) **and** *n −as→* n'*
and *n' postdominates* (*targetnode a*)
and *valid-edge a'* **and** *sourcenode a'* = *n*
and ¬ *n' postdominates* (*targetnode a'*)
by(*auto simp*:*standard-control-dependence-def*)
from ‹*n −as→* n'*› ‹*as* = *a#as'*› **have** *sourcenode a* = *n* **by**(*auto elim*:*path.cases*)

from ‹n −as→* n'› ‹as = a#as'› ‹n' ∉ set($sourcenodes$ as)› **have** $n \neq n'$
  by(*induct rule*:*path.induct*,*auto simp*:*sourcenodes-def*)
from ‹n −as→* n'› ‹as = a#as'› **have** *valid-edge a*
  by(*auto elim*:*path.cases*)
from ‹n *controls*$_s$ n'› **have** $n' \neq$ (*-Exit-*)
  by(*fastforce dest*:*Exit-not-standard-control-dependent*)
from ‹n −as→* n'› **have** (*-Exit-*) ∉ *set* (*sourcenodes as*) **by** *fastforce*
from ‹n −as→* n'› **have** *valid-node n* **and** *valid-node* n'
  by(*auto dest*:*path-valid-node*)
with ‹¬ n' *postdominates* (*targetnode* a')› ‹*valid-edge* a'›
obtain *asx* **where** *targetnode* a' −*asx*→* (*-Exit-*) **and** n' ∉ *set*(*sourcenodes asx*)
  by(*auto simp*:*postdominate-def*)
with ‹*valid-edge* a'› ‹*sourcenode* a' = n› **have** n −a'#*asx*→* (*-Exit-*)
  by(*fastforce intro*:*Cons-path*)
with ‹$n \neq n'$› ‹*sourcenode* a' = n› ‹n' ∉ *set*(*sourcenodes asx*)›
have n' ∉ *set*(*sourcenodes* (a'#*asx*)) **by**(*simp add*:*sourcenodes-def*)
from ‹n' *postdominates* (*targetnode a*)›
obtain asx' **where** *targetnode a* −asx'→* n' **by**(*erule postdominate-implies-path*)
from ‹n' *postdominates* (*targetnode a*)›
have ∀ as'. *targetnode a* −as'→* (*-Exit-*) ⟶ n' ∈ *set*(*sourcenodes* as')
  by(*auto simp*:*postdominate-def*)
with ‹$n' \neq$ (*-Exit-*)› ‹n −as→* n'› ‹(*-Exit-*) ∉ *set* (*sourcenodes as*)›
  ‹n −a'#*asx*→* (*-Exit-*)› ‹n' ∉ *set*(*sourcenodes* (a'#*asx*))›
  ‹*valid-edge a*› ‹*sourcenode a* = n› ‹*targetnode a* −asx'→* n'›
show *?thesis* **by**(*auto simp*:*wod-def*)
qed

end

end

Chapter 4

Instantiating the Framework with a simple While-Language

4.1 Commands

theory *Com* **imports** *Main* **begin**

4.1.1 Variables and Values

type-synonym *vname* = *string* — names for variables

datatype *val*
  = *Bool bool* — Boolean value
  | *Intg int* — integer value

abbreviation *true* == *Bool True*
abbreviation *false* == *Bool False*

4.1.2 Expressions and Commands

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*

datatype *cmd*
= *Skip*
| *LAss vname expr* (‹-:=-› [*70*,*70*] *70*) — local assignment
| *Seq cmd cmd* (‹-;;/ -› [*61*,*60*] *60*)
| *Cond expr cmd cmd* (‹*if* ′(-′) -/ *else* -› [*80*,*79*,*79*] *70*)
| *While expr cmd* (‹*while* ′(-′) -› [*80*,*79*] *70*)

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*

lemma *num-inner-nodes-gr-0*:#:*c* > *0*
by(*induct c*) *auto*

lemma [*dest*]:#:*c* = *0* ⟹ *False*
by(*induct c*) *auto*

4.1.3 The state

type-synonym *state* = *vname* ⇀ *val*

fun *interpret* :: *expr* ⇒ *state* ⇒ *val option*
where *Val*: *interpret* (*Val v*) *s* = *Some v*
| *Var*: *interpret* (*Var V*) *s* = *s V*
| *BinOp*: *interpret* (e_1«*bop*»e_2) *s* =
(*case interpret* e_1 *s of None* ⇒ *None*
| *Some* v_1 ⇒ (*case interpret* e_2 *s of None* ⇒ *None*
| *Some* v_2 ⇒ (
case binop bop v_1 v_2 *of None* ⇒ *None* | *Some v* ⇒ *Some v*)))

end

4.2 CFG

theory *WCFG* **imports** *Com* ../*Basic*/*BasicDefs* **begin**

4.2.1 CFG nodes

datatype *w-node* = *Node nat* (‹′(′- - ′-′)›)

| *Entry* (‹'('-*Entry*'-')›)
| *Exit* (‹'('-*Exit*'-')›)

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

lemma *Exit-label-incr* [*dest*]: (-*Exit*-) = *n* ⊕ *i* ⟹ *n* = (-*Exit*-)
by(*cases n*,*auto*)

lemma *label-incr-Exit* [*dest*]: *n* ⊕ *i* = (-*Exit*-) ⟹ *n* = (-*Exit*-)
by(*cases n*,*auto*)

lemma *Entry-label-incr* [*dest*]: (-*Entry*-) = *n* ⊕ *i* ⟹ *n* = (-*Entry*-)
by(*cases n*,*auto*)

lemma *label-incr-Entry* [*dest*]: *n* ⊕ *i* = (-*Entry*-) ⟹ *n* = (-*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*:
(- *l* -) = *n* ⊕ *i* ⟹ *n* = (-(*l* − *i*)-)
by(*cases n*,*auto*)

lemma *label-incr-ge*:(- *l* -) = *n* ⊕ *i* ⟹ $l \geq i$
by(*cases n*) *auto*

lemma *label-incr-0* [*dest*]:
⟦(-*0*-) = *n* ⊕ *i*; *i* > *0*⟧ ⟹ *False*
by(*cases n*) *auto*

lemma *label-incr-0-rev* [*dest*]:
⟦*n* ⊕ *i* = (-*0*-); *i* > *0*⟧ ⟹ *False*
by(*cases n*) *auto*

4.2.2 CFG edges

type-synonym *w-edge* = (*w-node* × *state edge-kind* × *w-node*)

inductive *While-CFG* :: *cmd* ⇒ *w-node* ⇒ *state edge-kind* ⇒ *w-node* ⇒ *bool*
(‹- ⊢ - −-→ -›)
where

WCFG-Entry-Exit:
prog ⊢ (-*Entry*-) −($\lambda s.$ *False*)$_{\surd}$→ (-*Exit*-)

| *WCFG-Entry*:
prog ⊢ (-*Entry*-) −($\lambda s.$ *True*)$_{\surd}$→ (-*0*-)

| *WCFG-Skip*:
Skip ⊢ (-*0*-) −⇑*id*→ (-*Exit*-)

| *WCFG-LAss*:
V:=*e* ⊢ (-*0*-) −⇑($\lambda s.$ *s*(*V*:=(*interpret e s*)))→ (-*1*-)

| *WCFG-LAssSkip*:
V:=*e* ⊢ (-*1*-) −⇑*id*→ (-*Exit*-)

| *WCFG-SeqFirst*:
⟦c_1 ⊢ *n* −*et*→ *n'*; *n'* ≠ (-*Exit*-)⟧ ⟹ c_1;;c_2 ⊢ *n* −*et*→ *n'*

| *WCFG-SeqConnect*:
⟦c_1 ⊢ *n* −*et*→ (-*Exit*-); *n* ≠ (-*Entry*-)⟧ ⟹ c_1;;c_2 ⊢ *n* −*et*→ (-*0*-) ⊕ #:c_1

| *WCFG-SeqSecond*:
⟦c_2 ⊢ *n* −*et*→ *n'*; *n* ≠ (-*Entry*-)⟧ ⟹ c_1;;c_2 ⊢ *n* ⊕ #:c_1 −*et*→ *n'* ⊕ #:c_1

| *WCFG-CondTrue*:
if (*b*) c_1 *else* c_2 ⊢ (-*0*-) −($\lambda s.$ *interpret b s* = *Some true*)$_{\surd}$→ (-*0*-) ⊕ *1*

| *WCFG-CondFalse*:
if (*b*) c_1 *else* c_2 ⊢ (-*0*-) −($\lambda s.$ *interpret b s* = *Some false*)$_{\surd}$→ (-*0*-) ⊕ (#:c_1 + *1*)

| *WCFG-CondThen*:
⟦c_1 ⊢ *n* −*et*→ *n'*; *n* ≠ (-*Entry*-)⟧ ⟹ *if* (*b*) c_1 *else* c_2 ⊢ *n* ⊕ *1* −*et*→ *n'* ⊕ *1*

| *WCFG-CondElse*:
⟦c_2 ⊢ *n* −*et*→ *n'*; *n* ≠ (-*Entry*-)⟧
⟹ *if* (*b*) c_1 *else* c_2 ⊢ *n* ⊕ (#:c_1 + *1*) −*et*→ *n'* ⊕ (#:c_1 + *1*)

| *WCFG-WhileTrue*:
while (*b*) *c'* ⊢ (-*0*-) −($\lambda s.$ *interpret b s* = *Some true*)$_{\surd}$→ (-*0*-) ⊕ *2*

| *WCFG-WhileFalse*:
while (*b*) *c'* ⊢ (-*0*-) −($\lambda s.$ *interpret b s* = *Some false*)$_{\surd}$→ (-*1*-)

| *WCFG-WhileFalseSkip*:
while (*b*) $c' \vdash$ (*-1-*) $-\Uparrow id\rightarrow$ (*-Exit-*)

| *WCFG-WhileBody*:
$[\![c' \vdash n -et\rightarrow n'; n \neq$ (*-Entry-*); $n' \neq$ (*-Exit-*)$]\!]$
$\Longrightarrow$ *while* (*b*) $c' \vdash n \oplus 2 -et\rightarrow n' \oplus 2$

| *WCFG-WhileBodyExit*:
$[\![c' \vdash n -et\rightarrow$ (*-Exit-*); $n \neq$ (*-Entry-*)$]\!] \Longrightarrow$ *while* (*b*) $c' \vdash n \oplus 2 -et\rightarrow$ (*-0-*)

lemmas *WCFG-intros* = *While-CFG.intros*[*split-format* (*complete*)]
lemmas *WCFG-elims* = *While-CFG.cases*[*split-format* (*complete*)]
lemmas *WCFG-induct* = *While-CFG.induct*[*split-format* (*complete*)]

4.2.3 Some lemmas about the CFG

lemma *WCFG-Exit-no-sourcenode* [*dest*]:
prog $\vdash$ (*-Exit-*) $-et\rightarrow n' \Longrightarrow$ *False*
by(*induct prog n*≡(*-Exit-*) *et n' rule*: *WCFG-induct*,*auto*)

lemma *WCFG-Entry-no-targetnode* [*dest*]:
prog $\vdash n -et\rightarrow$ (*-Entry-*) $\Longrightarrow$ *False*
by(*induct prog n et n'*≡(*-Entry-*) *rule*: *WCFG-induct*,*auto*)

lemma *WCFG-sourcelabel-less-num-nodes*:
prog $\vdash$ (*- l -*) $-et\rightarrow n' \Longrightarrow l <$ #:*prog*
proof(*induct prog* (*- l -*) *et n' arbitrary*:*l rule*: *WCFG-induct*)
case (*WCFG-SeqFirst* c_1 *et n'* c_2)
from ‹$l <$ #:c_1› **show** *?case* **by** *simp*
next
case (*WCFG-SeqConnect* c_1 *et* c_2)
from ‹$l <$ #:c_1› **show** *?case* **by** *simp*
next
case (*WCFG-SeqSecond* c_2 *n et n'* c_1)
note *IH* = ‹$\bigwedge l.\ n =$ (*- l -*) $\Longrightarrow l <$ #:c_2›
from ‹$n \oplus$ #:c_1 = (*- l -*)› **obtain** l' **where** n = (*- l' -*) **by**(*cases n*) *auto*
from *IH*[*OF this*] **have** $l' <$ #:c_2 **.**
with ‹$n \oplus$ #:c_1 = (*- l -*)› ‹n = (*- l' -*)› **show** *?case* **by** *simp*
next
case (*WCFG-CondThen* c_1 *n et n' b* c_2)
note *IH* = ‹$\bigwedge l.\ n =$ (*- l -*) $\Longrightarrow l <$ #:c_1›
from ‹$n \oplus 1$ = (*- l -*)› **obtain** l' **where** n = (*- l' -*) **by**(*cases n*) *auto*
from *IH*[*OF this*] **have** $l' <$ #:c_1 **.**
with ‹$n \oplus 1$ = (*- l -*)› ‹n = (*- l' -*)› **show** *?case* **by** *simp*
next
case (*WCFG-CondElse* c_2 *n et n' b* c_1)
note *IH* = ‹$\bigwedge l.\ n =$ (*- l -*) $\Longrightarrow l <$ #:c_2›

from ‹$n \oplus (\#{:}c_1 + 1) = (\text{- } l \text{ -})$› **obtain** l' **where** $n = (\text{- } l' \text{ -})$ **by**(*cases n*) *auto*
from *IH*[*OF this*] **have** $l' < \#{:}c_2$ **.**
with ‹$n \oplus (\#{:}c_1 + 1) = (\text{- } l \text{ -})$› ‹$n = (\text{- } l' \text{ -})$› **show** *?case* **by** *simp*
next
case (*WCFG-WhileBody* c' *n et* n' *b*)
note *IH* = ‹$\bigwedge l.\ n = (\text{- } l \text{ -}) \Longrightarrow l < \#{:}c'$›
from ‹$n \oplus 2 = (\text{- } l \text{ -})$› **obtain** l' **where** $n = (\text{- } l' \text{ -})$ **by**(*cases n*) *auto*
from *IH*[*OF this*] **have** $l' < \#{:}c'$ **.**
with ‹$n \oplus 2 = (\text{- } l \text{ -})$› ‹$n = (\text{- } l' \text{ -})$› **show** *?case* **by** *simp*
next
case (*WCFG-WhileBodyExit* c' *n et b*)
note *IH* = ‹$\bigwedge l.\ n = (\text{- } l \text{ -}) \Longrightarrow l < \#{:}c'$›
from ‹$n \oplus 2 = (\text{- } l \text{ -})$› **obtain** l' **where** $n = (\text{- } l' \text{ -})$ **by**(*cases n*) *auto*
from *IH*[*OF this*] **have** $l' < \#{:}c'$ **.**
with ‹$n \oplus 2 = (\text{- } l \text{ -})$› ‹$n = (\text{- } l' \text{ -})$› **show** *?case* **by** *simp*
qed (*auto simp*:*num-inner-nodes-gr-0*)

lemma *WCFG-targetlabel-less-num-nodes*:
$prog \vdash n\ {-et\rightarrow}\ (\text{- } l \text{ -}) \Longrightarrow l < \#{:}prog$
proof(*induct prog n et* (- *l* -) *arbitrary*:*l rule*:*WCFG-induct*)
case (*WCFG-SeqFirst* c_1 *n et* c_2)
from ‹$l < \#{:}c_1$› **show** *?case* **by** *simp*
next
case (*WCFG-SeqSecond* c_2 *n et* n' c_1)
note *IH* = ‹$\bigwedge l.\ n' = (\text{- } l \text{ -}) \Longrightarrow l < \#{:}c_2$›
from ‹$n' \oplus \#{:}c_1 = (\text{- } l \text{ -})$› **obtain** l' **where** $n' = (\text{- } l' \text{ -})$ **by**(*cases* n') *auto*
from *IH*[*OF this*] **have** $l' < \#{:}c_2$ **.**
with ‹$n' \oplus \#{:}c_1 = (\text{- } l \text{ -})$› ‹$n' = (\text{- } l' \text{ -})$› **show** *?case* **by** *simp*
next
case (*WCFG-CondThen* c_1 *n et* n' *b* c_2)
note *IH* = ‹$\bigwedge l.\ n' = (\text{- } l \text{ -}) \Longrightarrow l < \#{:}c_1$›
from ‹$n' \oplus 1 = (\text{- } l \text{ -})$› **obtain** l' **where** $n' = (\text{- } l' \text{ -})$ **by**(*cases* n') *auto*
from *IH*[*OF this*] **have** $l' < \#{:}c_1$ **.**
with ‹$n' \oplus 1 = (\text{- } l \text{ -})$› ‹$n' = (\text{- } l' \text{ -})$› **show** *?case* **by** *simp*
next
case (*WCFG-CondElse* c_2 *n et* n' *b* c_1)
note *IH* = ‹$\bigwedge l.\ n' = (\text{- } l \text{ -}) \Longrightarrow l < \#{:}c_2$›
from ‹$n' \oplus (\#{:}c_1 + 1) = (\text{- } l \text{ -})$› **obtain** l' **where** $n' = (\text{- } l' \text{ -})$ **by**(*cases* n')
auto
from *IH*[*OF this*] **have** $l' < \#{:}c_2$ **.**
with ‹$n' \oplus (\#{:}c_1 + 1) = (\text{- } l \text{ -})$› ‹$n' = (\text{- } l' \text{ -})$› **show** *?case* **by** *simp*
next
case (*WCFG-WhileBody* c' *n et* n' *b*)
note *IH* = ‹$\bigwedge l.\ n' = (\text{- } l \text{ -}) \Longrightarrow l < \#{:}c'$›
from ‹$n' \oplus 2 = (\text{- } l \text{ -})$› **obtain** l' **where** $n' = (\text{- } l' \text{ -})$ **by**(*cases* n') *auto*
from *IH*[*OF this*] **have** $l' < \#{:}c'$ **.**
with ‹$n' \oplus 2 = (\text{- } l \text{ -})$› ‹$n' = (\text{- } l' \text{ -})$› **show** *?case* **by** *simp*
qed (*auto simp*:*num-inner-nodes-gr-0*)

lemma *WCFG-EntryD*:
  prog ⊢ (*-Entry-*) −*et*→ *n′*
  ⟹ (*n′* = (*-Exit-*) ∧ *et* = (λ*s*. *False*)$_{\surd}$) ∨ (*n′* = (*-0-*) ∧ *et* = (λ*s*. *True*)$_{\surd}$)
by(*induct prog n*≡(*-Entry-*) *et n′ rule*: *WCFG-induct*,*auto*)

lemma *WCFG-edge-det*:
  ⟦*prog* ⊢ *n* −*et*→ *n′*; *prog* ⊢ *n* −*et′*→ *n′*⟧ ⟹ *et* = *et′*
proof(*induct rule*: *WCFG-induct*)
  case *WCFG-Entry-Exit* **thus** *?case* **by**(*fastforce dest*: *WCFG-EntryD*)
next
  case *WCFG-Entry* **thus** *?case* **by**(*fastforce dest*: *WCFG-EntryD*)
next
  case *WCFG-Skip* **thus** *?case* **by**(*fastforce elim*: *WCFG-elims*)
next
  case *WCFG-LAss* **thus** *?case* **by**(*fastforce elim*: *WCFG-elims*)
next
  case *WCFG-LAssSkip* **thus** *?case* **by**(*fastforce elim*: *WCFG-elims*)
next
  case (*WCFG-SeqFirst* c_1 *n et n′* c_2)
  note *IH* = ‹c_1 ⊢ *n* −*et′*→ *n′* ⟹ *et* = *et′*›
  from ‹c_1 ⊢ *n* −*et*→ *n′*› ‹*n′* ≠ (*-Exit-*)› **obtain** *l* **where** *n′* = (- *l* -)
    by (*cases n′*) *auto*
  with ‹c_1 ⊢ *n* −*et*→ *n′*› **have** *l* < #:c_1
    by(*fastforce intro*: *WCFG-targetlabel-less-num-nodes*)
  with ‹c_1;;c_2 ⊢ *n* −*et′*→ *n′*› ‹*n′* = (- *l* -)› **have** c_1 ⊢ *n* −*et′*→ *n′*
    by(*fastforce elim*: *WCFG-elims intro*: *WCFG-intros dest*: *label-incr-ge*)
  from *IH*[*OF this*] **show** *?case* **.**
next
  case (*WCFG-SeqConnect* c_1 *n et* c_2)
  note *IH* = ‹c_1 ⊢ *n* −*et′*→ (*-Exit-*) ⟹ *et* = *et′*›
  from ‹c_1 ⊢ *n* −*et*→ (*-Exit-*)› ‹*n* ≠ (*-Entry-*)› **obtain** *l* **where** *n* = (- *l* -)
    by (*cases n*) *auto*
  with ‹c_1 ⊢ *n* −*et*→ (*-Exit-*)› **have** *l* < #:c_1
    by(*fastforce intro*: *WCFG-sourcelabel-less-num-nodes*)
  with ‹c_1;;c_2 ⊢ *n* −*et′*→ (- *0* -) ⊕ #:c_1› ‹*n* = (- *l* -)› **have** c_1 ⊢ *n* −*et′*→ (*-Exit-*)
    by(*fastforce elim*: *WCFG-elims dest*: *WCFG-targetlabel-less-num-nodes label-incr-ge*)
  from *IH*[*OF this*] **show** *?case* **.**
next
  case (*WCFG-SeqSecond* c_2 *n et n′* c_1)
  note *IH* = ‹c_2 ⊢ *n* −*et′*→ *n′* ⟹ *et* = *et′*›
  from ‹c_2 ⊢ *n* −*et*→ *n′*› ‹*n* ≠ (*-Entry-*)› **obtain** *l* **where** *n* = (- *l* -)
    by (*cases n*) *auto*
  with ‹c_2 ⊢ *n* −*et*→ *n′*› **have** *l* < #:c_2
    by(*fastforce intro*: *WCFG-sourcelabel-less-num-nodes*)
  with ‹c_1;;c_2 ⊢ *n* ⊕ #:c_1 −*et′*→ *n′* ⊕ #:c_1› ‹*n* = (- *l* -)› **have** c_2 ⊢ *n* −*et′*→ *n′*
    by −(*erule WCFG-elims*,(*fastforce dest*: *WCFG-sourcelabel-less-num-nodes la-*

bel-incr-ge

dest!:*label-incr-inj*)+)

from *IH*[*OF this*] **show** *?case* **.**

next

case *WCFG-CondTrue* **thus** *?case* **by**(*fastforce elim*: *WCFG-elims*)

next

case *WCFG-CondFalse* **thus** *?case* **by**(*fastforce elim*: *WCFG-elims*)

next

case (*WCFG-CondThen* c_1 *n et n′ b* c_2)

note *IH* = ‹c_1 ⊢ *n* −*et′*→ *n′* ⟹ *et* = *et′*›

from ‹c_1 ⊢ *n* −*et*→ *n′*› ‹*n* ≠ (-*Entry*-)› **obtain** *l* **where** *n* = (- *l* -)

by (*cases n*) *auto*

with ‹c_1 ⊢ *n* −*et*→ *n′*› **have** *l* < #:c_1

by(*fastforce intro*: *WCFG-sourcelabel-less-num-nodes*)

with ‹*if* (*b*) c_1 *else* c_2 ⊢ *n* ⊕ *1* −*et′*→ *n′* ⊕ *1*› ‹*n* = (- *l* -)›

have c_1 ⊢ *n* −*et′*→ *n′*

by −(*erule WCFG-elims*,(*fastforce dest*:*label-incr-ge label-incr-inj*)+)

from *IH*[*OF this*] **show** *?case* **.**

next

case (*WCFG-CondElse* c_2 *n et n′ b* c_1)

note *IH* = ‹c_2 ⊢ *n* −*et′*→ *n′* ⟹ *et* = *et′*›

from ‹c_2 ⊢ *n* −*et*→ *n′*› ‹*n* ≠ (-*Entry*-)› **obtain** *l* **where** *n* = (- *l* -)

by (*cases n*) *auto*

with ‹c_2 ⊢ *n* −*et*→ *n′*› **have** *l* < #:c_2

by(*fastforce intro*: *WCFG-sourcelabel-less-num-nodes*)

with ‹*if* (*b*) c_1 *else* c_2 ⊢ *n* ⊕ (#:c_1 + *1*) −*et′*→ *n′* ⊕ (#:c_1 + *1*)› ‹*n* = (- *l* -)›

have c_2 ⊢ *n* −*et′*→ *n′*

by −(*erule WCFG-elims*,(*fastforce dest*: *WCFG-sourcelabel-less-num-nodes*

label-incr-inj label-incr-ge label-incr-simp-rev)+)

from *IH*[*OF this*] **show** *?case* **.**

next

case *WCFG-WhileTrue* **thus** *?case* **by**(*fastforce elim*: *WCFG-elims*)

next

case *WCFG-WhileFalse* **thus** *?case* **by**(*fastforce elim*: *WCFG-elims*)

next

case *WCFG-WhileFalseSkip* **thus** *?case* **by**(*fastforce elim*: *WCFG-elims*)

next

case (*WCFG-WhileBody c′ n et n′ b*)

note *IH* = ‹*c′* ⊢ *n* −*et′*→ *n′* ⟹ *et* = *et′*›

from ‹*c′* ⊢ *n* −*et*→ *n′*› ‹*n* ≠ (-*Entry*-)› **obtain** *l* **where** *n* = (- *l* -)

by (*cases n*) *auto*

moreover

with ‹*c′* ⊢ *n* −*et*→ *n′*› **have** *l* < #:*c′*

by(*fastforce intro*: *WCFG-sourcelabel-less-num-nodes*)

moreover

from ‹*c′* ⊢ *n* −*et*→ *n′*› ‹*n′* ≠ (-*Exit*-)› **obtain** *l′* **where** *n′* = (- *l′* -)

by (*cases n′*) *auto*

moreover

with ‹*c′* ⊢ *n* −*et*→ *n′*› **have** *l′* < #:*c′*

by(*fastforce intro*: *WCFG-targetlabel-less-num-nodes*)
ultimately have *c′* ⊢ *n* −*et′*→ *n′* **using** ‹*while* (*b*) *c′* ⊢ *n* ⊕ *2* −*et′*→ *n′* ⊕ *2*›
by(*fastforce elim*: *WCFG-elims dest*:*label-incr-start-Node-smaller*)
from *IH*[*OF this*] **show** *?case* **.**
next
case (*WCFG-WhileBodyExit c′ n et b*)
note *IH* = ‹*c′* ⊢ *n* −*et′*→ (-*Exit*-) ⟹ *et* = *et′*›
from ‹*c′* ⊢ *n* −*et*→ (-*Exit*-)› ‹*n* ≠ (-*Entry*-)› **obtain** *l* **where** *n* = (- *l* -)
by (*cases n*) *auto*
with ‹*c′* ⊢ *n* −*et*→ (-*Exit*-)› **have** *l* < #:*c′*
by(*fastforce intro*: *WCFG-sourcelabel-less-num-nodes*)
with ‹*while* (*b*) *c′* ⊢ *n* ⊕ *2* −*et′*→ (-*0*-)› ‹*n* = (- *l* -)›
have *c′* ⊢ *n* −*et′*→ (-*Exit*-)
by −(*erule WCFG-elims*,*auto dest*:*label-incr-start-Node-smaller*)
from *IH*[*OF this*] **show** *?case* **.**
qed

lemma *less-num-nodes-edge-Exit*:
obtains *l et* **where** *l* < #:*prog* **and** *prog* ⊢ (- *l* -) −*et*→ (-*Exit*-)
proof −
have ∃ *l et*. *l* < #:*prog* ∧ *prog* ⊢ (- *l* -) −*et*→ (-*Exit*-)
proof(*induct prog*)
case *Skip*
have *0* < #:*Skip* **by** *simp*
moreover have *Skip* ⊢ (-*0*-) −⇑*id*→ (-*Exit*-) **by**(*rule WCFG-Skip*)
ultimately show *?case* **by** *blast*
next
case (*LAss V e*)
have *1* < #:(*V*:=*e*) **by** *simp*
moreover have *V*:=*e* ⊢ (-*1*-) −⇑*id*→ (-*Exit*-) **by**(*rule WCFG-LAssSkip*)
ultimately show *?case* **by** *blast*
next
case (*Seq prog1 prog2*)
from ‹∃ *l et*. *l* < #:*prog2* ∧ *prog2* ⊢ (- *l* -) −*et*→ (-*Exit*-)›
obtain *l et* **where** *l* < #:*prog2* **and** *prog2* ⊢ (- *l* -) −*et*→ (-*Exit*-)
by *blast*
from ‹*prog2* ⊢ (- *l* -) −*et*→ (-*Exit*-)›
have *prog1*;;*prog2* ⊢ (- *l* -) ⊕ #:*prog1* −*et*→ (-*Exit*-) ⊕ #:*prog1*
by(*fastforce intro*: *WCFG-SeqSecond*)
with ‹*l* < #:*prog2*› **show** *?case* **by**(*rule-tac x*=*l* + #:*prog1* **in** *exI*,*auto*)
next
case (*Cond b prog1 prog2*)
from ‹∃ *l et*. *l* < #:*prog1* ∧ *prog1* ⊢ (- *l* -) −*et*→ (-*Exit*-)›
obtain *l et* **where** *l* < #:*prog1* **and** *prog1* ⊢ (- *l* -) −*et*→ (-*Exit*-)
by *blast*
from ‹*prog1* ⊢ (- *l* -) −*et*→ (-*Exit*-)›
have *if* (*b*) *prog1 else prog2* ⊢ (- *l* -) ⊕ *1* −*et*→ (-*Exit*-) ⊕ *1*
by(*fastforce intro*: *WCFG-CondThen*)
with ‹*l* < #:*prog1*› **show** *?case* **by**(*rule-tac x*=*l* + *1* **in** *exI*,*auto*)

```
  next
    case (While b prog')
    have 1 < #:(while (b) prog') by simp
    moreover have while (b) prog' ⊢ (-1-) −⇑id→ (-Exit-)
      by(rule WCFG-WhileFalseSkip)
    ultimately show ?case by blast
  qed
  with that show ?thesis by blast
qed


lemma less-num-nodes-edge:
  l < #:prog ⟹ ∃ n et. prog ⊢ n −et→ (- l -) ∨ prog ⊢ (- l -) −et→ n
proof(induct prog arbitrary:l)
  case Skip
  from ‹l < #:Skip› have l = 0 by simp
  hence Skip ⊢ (- l -) −⇑id→ (-Exit-) by(fastforce intro:WCFG-Skip)
  thus ?case by blast
next
  case (LAss V e)
  from ‹l < #:V:=e› have l = 0 ∨ l = 1 by auto
  thus ?case
  proof
    assume l = 0
   hence V:=e ⊢ (-Entry-) −(λs. True)√→ (- l -) by(fastforce intro:WCFG-Entry)
    thus ?thesis by blast
  next
    assume l = 1
    hence V:=e ⊢ (- l -) −⇑id→ (-Exit-) by(fastforce intro:WCFG-LAssSkip)
    thus ?thesis by blast
  qed
next
  case (Seq prog1 prog2)
  note IH1 = ‹⋀l. l < #:prog1 ⟹
            ∃ n et. prog1 ⊢ n −et→ (- l -) ∨ prog1 ⊢ (- l -) −et→ n›
  note IH2 = ‹⋀l. l < #:prog2 ⟹
            ∃ n et. prog2 ⊢ n −et→ (- l -) ∨ prog2 ⊢ (- l -) −et→ n›
  show ?case
  proof(cases l < #:prog1)
    case True
    from IH1[OF this] obtain n et
      where prog1 ⊢ n −et→ (- l -) ∨ prog1 ⊢ (- l -) −et→ n by blast
    thus ?thesis
    proof
      assume prog1 ⊢ n −et→ (- l -)
      hence prog1;; prog2 ⊢ n −et→ (- l -) by(fastforce intro:WCFG-SeqFirst)
      thus ?thesis by blast
    next
      assume edge:prog1 ⊢ (- l -) −et→ n
```

show *?thesis*
proof(*cases n* = (*-Exit-*))
case *True*
with *edge* **have** *prog1*;; *prog2* ⊢ (*- l -*) *−et→* (*-0-*) ⊕ #:*prog1*
by(*fastforce intro*: *WCFG-SeqConnect*)
thus *?thesis* **by** *blast*
next
case *False*
with *edge* **have** *prog1*;; *prog2* ⊢ (*- l -*) *−et→ n*
by(*fastforce intro*: *WCFG-SeqFirst*)
thus *?thesis* **by** *blast*
qed
qed
next
case *False*
hence #:*prog1* ≤ *l* **by** *simp*
then obtain *l′* **where** *l* = *l′* + #:*prog1* **and** *l′* = *l* − #:*prog1* **by** *simp*
from ‹*l* = *l′* + #:*prog1*› ‹*l* < #:*prog1*;; *prog2*› **have** *l′* < #:*prog2* **by** *simp*
from *IH2*[*OF this*] **obtain** *n et*
where *prog2* ⊢ *n −et→* (*- l′ -*) ∨ *prog2* ⊢ (*- l′ -*) *−et→ n* **by** *blast*
thus *?thesis*
proof
assume *prog2* ⊢ *n −et→* (*- l′ -*)
show *?thesis*
proof(*cases n* = (*-Entry-*))
case *True*
with ‹*prog2* ⊢ *n −et→* (*- l′ -*)› **have** *l′* = *0* **by**(*auto dest*: *WCFG-EntryD*)
obtain *l′′ et′′* **where** *l′′* < #:*prog1*
and *prog1* ⊢ (*- l′′ -*) *−et′′→* (*-Exit-*)
by(*erule less-num-nodes-edge-Exit*)
hence *prog1*;;*prog2* ⊢ (*- l′′ -*) *−et′′→* (*-0-*) ⊕ #:*prog1*
by(*fastforce intro*: *WCFG-SeqConnect*)
with ‹*l′* = *0*› ‹*l* = *l′* + #:*prog1*› **show** *?thesis* **by** *simp blast*
next
case *False*
with ‹*prog2* ⊢ *n −et→* (*- l′ -*)›
have *prog1*;;*prog2* ⊢ *n* ⊕ #:*prog1* *−et→* (*- l′ -*) ⊕ #:*prog1*
by(*fastforce intro*: *WCFG-SeqSecond*)
with ‹*l* = *l′* + #:*prog1*› **show** *?thesis* **by** *simp blast*
qed
next
assume *prog2* ⊢ (*- l′ -*) *−et→ n*
hence *prog1*;;*prog2* ⊢ (*- l′ -*) ⊕ #:*prog1* *−et→ n* ⊕ #:*prog1*
by(*fastforce intro*: *WCFG-SeqSecond*)
with ‹*l* = *l′* + #:*prog1*› **show** *?thesis* **by** *simp blast*
qed
qed
next
case (*Cond b prog1 prog2*)

```
note IH1 = ‹⋀l. l < #:prog1 ⟹
          ∃ n et. prog1 ⊢ n −et→ (- l -) ∨ prog1 ⊢ (- l -) −et→ n›
note IH2 = ‹⋀l. l < #:prog2 ⟹
          ∃ n et. prog2 ⊢ n −et→ (- l -) ∨ prog2 ⊢ (- l -) −et→ n›
show ?case
proof(cases l = 0)
  case True
  have if (b) prog1 else prog2 ⊢ (-Entry-) −(λs. True)√→ (-0-)
    by(rule WCFG-Entry)
  with True show ?thesis by simp blast
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' < #:prog1)
    case True
    from IH1[OF this] obtain n et
      where prog1 ⊢ n −et→ (- l' -) ∨ prog1 ⊢ (- l' -) −et→ n by blast
    thus ?thesis
    proof
      assume edge:prog1 ⊢ n −et→ (- l' -)
      show ?thesis
      proof(cases n = (-Entry-))
        case True
        with edge have l' = 0 by(auto dest:WCFG-EntryD)
        have if (b) prog1 else prog2 ⊢ (-0-) −(λs. interpret b s = Some true)√→
                                    (-0-) ⊕ 1
          by(rule WCFG-CondTrue)
        with ‹l' = 0› ‹l = l' + 1› show ?thesis by simp blast
      next
        case False
        with edge have if (b) prog1 else prog2 ⊢ n ⊕ 1 −et→ (- l' -) ⊕ 1
          by(fastforce intro:WCFG-CondThen)
        with ‹l = l' + 1› show ?thesis by simp blast
      qed
    next
      assume prog1 ⊢ (- l' -) −et→ n
      hence if (b) prog1 else prog2 ⊢ (- l' -) ⊕ 1 −et→ n ⊕ 1
        by(fastforce intro:WCFG-CondThen)
      with ‹l = l' + 1› show ?thesis by simp blast
    qed
  next
    case False
    hence #:prog1 ≤ l' by simp
    then obtain l'' where l' = l'' + #:prog1 and l'' = l' − #:prog1
      by simp
    from ‹l' = l'' + #:prog1› ‹l = l' + 1› ‹l < #:(if (b) prog1 else prog2)›
    have l'' < #:prog2 by simp
```

```
      from IH2[OF this] obtain n et
        where prog2 ⊢ n −et→ (- l'' -) ∨ prog2 ⊢ (- l'' -) −et→ n by blast
      thus ?thesis
      proof
        assume prog2 ⊢ n −et→ (- l'' -)
        show ?thesis
        proof(cases n = (-Entry-))
          case True
          with ‹prog2 ⊢ n −et→ (- l'' -)› have l'' = 0 by(auto dest:WCFG-EntryD)
          have if (b) prog1 else prog2 ⊢ (-0-) −(λs. interpret b s = Some false)√→
                                  (-0-) ⊕ (#:prog1 + 1)
            by(rule WCFG-CondFalse)
          with ‹l'' = 0› ‹l' = l'' + #:prog1› ‹l = l' + 1› show ?thesis by simp
blast
        next
          case False
          with ‹prog2 ⊢ n −et→ (- l'' -)›
          have if (b) prog1 else prog2 ⊢ n ⊕ (#:prog1 + 1) −et→
                                  (- l'' -) ⊕ (#:prog1 + 1)
            by(fastforce intro:WCFG-CondElse)
          with ‹l = l' + 1› ‹l' = l'' + #:prog1› show ?thesis by simp blast
        qed
      next
        assume prog2 ⊢ (- l'' -) −et→ n
        hence if (b) prog1 else prog2 ⊢ (- l'' -) ⊕ (#:prog1 + 1) −et→
                                  n ⊕ (#:prog1 + 1)
          by(fastforce intro:WCFG-CondElse)
        with ‹l = l' + 1› ‹l' = l'' + #:prog1› show ?thesis by simp blast
      qed
    qed
  qed
next
  case (While b prog')
  note IH = ‹⋀l. l < #:prog'
           ⟹ ∃ n et. prog' ⊢ n −et→ (- l -) ∨ prog' ⊢ (- l -) −et→ n›
  show ?case
  proof(cases l < 1)
    case True
    have while (b) prog' ⊢ (-Entry-) −(λs. True)√→ (-0-) by(rule WCFG-Entry)
    with True show ?thesis by simp blast
  next
    case False
    hence 1 ≤ l by simp
    thus ?thesis
    proof(cases l < 2)
      case True
      with ‹1 ≤ l› have l = 1 by simp
      have while (b) prog' ⊢ (-0-) −(λs. interpret b s = Some false)√→ (-1-)
        by(rule WCFG-WhileFalse)
```

```
      with ‹l = 1› show ?thesis by simp blast
    next
      case False
      with ‹1 ≤ l› 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) prog'› have l' < #:prog' by simp
      from IH[OF this] obtain n et
        where prog' ⊢ n -et→ (- l' -) ∨ prog' ⊢ (- l' -) -et→ n by blast
      thus ?thesis
      proof
        assume prog' ⊢ n -et→ (- l' -)
        show ?thesis
        proof(cases n = (-Entry-))
          case True
          with ‹prog' ⊢ n -et→ (- l' -)› have l' = 0 by(auto dest:WCFG-EntryD)
          have while (b) prog' ⊢ (-0-) -(λs. interpret b s = Some true)√→
                                 (-0-) ⊕ 2
            by(rule WCFG-WhileTrue)
          with ‹l' = 0› ‹l = l' + 2› show ?thesis by simp blast
        next
          case False
          with ‹prog' ⊢ n -et→ (- l' -)›
          have while (b) prog' ⊢ n ⊕ 2 -et→ (- l' -) ⊕ 2
            by(fastforce intro:WCFG-WhileBody)
          with ‹l = l' + 2› show ?thesis by simp blast
        qed
      next
        assume prog' ⊢ (- l' -) -et→ n
        show ?thesis
        proof(cases n = (-Exit-))
          case True
          with ‹prog' ⊢ (- l' -) -et→ n›
          have while (b) prog' ⊢ (- l' -) ⊕ 2 -et→ (-0-)
            by(fastforce intro:WCFG-WhileBodyExit)
          with ‹l = l' + 2› show ?thesis by simp blast
        next
          case False
          with ‹prog' ⊢ (- l' -) -et→ n›
          have while (b) prog' ⊢ (- l' -) ⊕ 2 -et→ n ⊕ 2
            by(fastforce intro:WCFG-WhileBody)
          with ‹l = l' + 2› show ?thesis by simp blast
        qed
      qed
    qed
  qed
qed
```

lemma *WCFG-deterministic*:
$[\![prog \vdash n_1 -et_1\rightarrow n_1'; prog \vdash n_2 -et_2\rightarrow n_2'; n_1 = n_2; n_1' \neq n_2']\!]$
$\Longrightarrow \exists\, Q\ Q'.\ et_1 = (Q)_{\surd} \wedge et_2 = (Q')_{\surd} \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$
proof(*induct arbitrary*:n_2 n_2' *rule*:*WCFG-induct*)
  case (*WCFG-Entry-Exit prog*)
  from ‹$prog \vdash n_2 -et_2\rightarrow n_2'$› ‹(-*Entry*-) = n_2› ‹(-*Exit*-) ≠ n_2'›
  have $et_2 = (\lambda s.\ True)_{\surd}$ **by**(*fastforce dest*:*WCFG-EntryD*)
  thus *?case* **by** *simp*
next
  case (*WCFG-Entry prog*)
  from ‹$prog \vdash n_2 -et_2\rightarrow n_2'$› ‹(-*Entry*-) = n_2› ‹(-*0*-) ≠ n_2'›
  have $et_2 = (\lambda s.\ False)_{\surd}$ **by**(*fastforce dest*:*WCFG-EntryD*)
  thus *?case* **by** *simp*
next
  case *WCFG-Skip*
  from ‹$Skip \vdash n_2 -et_2\rightarrow n_2'$› ‹(-*0*-) = n_2› ‹(-*Exit*-) ≠ n_2'›
  have *False* **by**(*fastforce elim*:*WCFG.While-CFG.cases*)
  thus *?case* **by** *simp*
next
  case (*WCFG-LAss V e*)
  from ‹$V{:=}e \vdash n_2 -et_2\rightarrow n_2'$› ‹(-*0*-) = n_2› ‹(-*1*-) ≠ n_2'›
  have *False* **by** −(*erule WCFG.While-CFG.cases*,*auto*)
  thus *?case* **by** *simp*
next
  case (*WCFG-LAssSkip V e*)
  from ‹$V{:=}e \vdash n_2 -et_2\rightarrow n_2'$› ‹(-*1*-) = n_2› ‹(-*Exit*-) ≠ n_2'›
  have *False* **by** −(*erule WCFG.While-CFG.cases*,*auto*)
  thus *?case* **by** *simp*
next
  case (*WCFG-SeqFirst* c_1 *n et n'* c_2)
  note *IH* = ‹$\bigwedge n_2\ n_2'.\ [\![c_1 \vdash n_2 -et_2\rightarrow n_2'; n = n_2; n' \neq n_2']\!]$
  $\Longrightarrow \exists\, Q\ Q'.\ et = (Q)_{\surd} \wedge et_2 = (Q')_{\surd} \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$›
  from ‹$c_1;;c_2 \vdash n_2 -et_2\rightarrow n_2'$› ‹$c_1 \vdash n -et\rightarrow n'$› ‹$n = n_2$› ‹$n' \neq n_2'$›
  have $c_1 \vdash n_2 -et_2\rightarrow n_2' \vee (c_1 \vdash n_2 -et_2\rightarrow$ (-*Exit*-) $\wedge\ n_2'$ = (-*0*-) ⊕ #:c_1)
    apply *hypsubst-thin* **apply**(*erule WCFG.While-CFG.cases*)
    apply(*auto intro*:*WCFG.While-CFG.intros*)
    by(*case-tac n*,*auto dest*:*WCFG-sourcelabel-less-num-nodes*)+
  thus *?case*
  proof
    assume $c_1 \vdash n_2 -et_2\rightarrow n_2'$
    from *IH*[*OF this* ‹$n = n_2$› ‹$n' \neq n_2'$›] **show** *?case* **.**
  next
    assume $c_1 \vdash n_2 -et_2\rightarrow$ (-*Exit*-) $\wedge\ n_2'$ = (-*0*-) ⊕ #:c_1
    hence *edge*:$c_1 \vdash n_2 -et_2\rightarrow$ (-*Exit*-) **and** *n2'*:n_2' = (-*0*-) ⊕ #:c_1 **by** *simp-all*
    from *IH*[*OF edge* ‹$n = n_2$› ‹$n' \neq$ (-*Exit*-)›] **show** *?case* **.**
  qed
next

case (*WCFG-SeqConnect* c_1 *n et* c_2)
note *IH* = ‹$\bigwedge n_2\ n_2{}'$. ⟦$c_1 \vdash n_2\ {-et_2\rightarrow}\ n_2{}'$; $n = n_2$; (*-Exit-*) $\neq n_2{}'$⟧
$\Longrightarrow \exists\, Q\ Q'.\ et = (Q)_{\surd} \wedge et_2 = (Q')_{\surd} \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg$
$Q\ s))$›
from ‹c_1;;$c_2 \vdash n_2\ {-et_2\rightarrow}\ n_2{}'$› ‹$c_1 \vdash n\ {-et\rightarrow}$ (*-Exit-*)› ‹$n = n_2$› ‹$n \neq$ (*-Entry-*)›
‹(*-0-*) $\oplus$ #:$c_1 \neq n_2{}'$› **have** $c_1 \vdash n_2\ {-et_2\rightarrow}\ n_2{}' \wedge$ (*-Exit-*) $\neq n_2{}'$
apply *hypsubst-thin* **apply**(*erule WCFG.While-CFG.cases*)
apply(*auto intro*:*WCFG.While-CFG.intros*)
by(*case-tac n*,*auto dest*:*WCFG-sourcelabel-less-num-nodes*)+
from *IH*[*OF this*[*THEN conjunct1*] ‹$n = n_2$› *this*[*THEN conjunct2*]]
show *?case* **.**
next
case (*WCFG-SeqSecond* c_2 *n et* n' c_1)
note *IH* = ‹$\bigwedge n_2\ n_2{}'$. ⟦$c_2 \vdash n_2\ {-et_2\rightarrow}\ n_2{}'$; $n = n_2$; $n' \neq n_2{}'$⟧
$\Longrightarrow \exists\, Q\ Q'.\ et = (Q)_{\surd} \wedge et_2 = (Q')_{\surd} \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg$
$Q\ s))$›
from ‹c_1;;$c_2 \vdash n_2\ {-et_2\rightarrow}\ n_2{}'$› ‹$c_2 \vdash n\ {-et\rightarrow}\ n'$› ‹$n \oplus$ #:$c_1 = n_2$›
‹$n' \oplus$ #:$c_1 \neq n_2{}'$› ‹$n \neq$ (*-Entry-*)›
obtain *nx* **where** $c_2 \vdash n\ {-et_2\rightarrow}\ nx \wedge nx \oplus$ #:$c_1 = n_2{}'$
apply − **apply**(*erule WCFG.While-CFG.cases*)
apply(*auto intro*:*WCFG.While-CFG.intros*)
apply(*cases n*,*auto dest*:*WCFG-sourcelabel-less-num-nodes*)
apply(*cases n*,*auto dest*:*WCFG-sourcelabel-less-num-nodes*)
by(*fastforce dest*:*label-incr-inj*)
with ‹$n' \oplus$ #:$c_1 \neq n_2{}'$› **have** *edge*:$c_2 \vdash n\ {-et_2\rightarrow}\ nx$ **and** *neq*:$n' \neq nx$
by *auto*
from *IH*[*OF edge - neq*] **show** *?case* **by** *simp*
next
case (*WCFG-CondTrue b* c_1 c_2)
from ‹*if* (*b*) c_1 *else* $c_2 \vdash n_2\ {-et_2\rightarrow}\ n_2{}'$› ‹(*-0-*) $= n_2$› ‹(*-0-*) $\oplus$ *1* $\neq n_2{}'$›
show *?case* **by** −(*erule WCFG.While-CFG.cases*,*auto*)
next
case (*WCFG-CondFalse b* c_1 c_2)
from ‹*if* (*b*) c_1 *else* $c_2 \vdash n_2\ {-et_2\rightarrow}\ n_2{}'$› ‹(*-0-*) $= n_2$› ‹(*-0-*) $\oplus$ #:c_1 + *1* $\neq n_2{}'$›
show *?case* **by** −(*erule WCFG.While-CFG.cases*,*auto*)
next
case (*WCFG-CondThen* c_1 *n et* n' *b* c_2)
note *IH* = ‹$\bigwedge n_2\ n_2{}'$. ⟦$c_1 \vdash n_2\ {-et_2\rightarrow}\ n_2{}'$; $n = n_2$; $n' \neq n_2{}'$⟧
$\Longrightarrow \exists\, Q\ Q'.\ et = (Q)_{\surd} \wedge et_2 = (Q')_{\surd} \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg$
$Q\ s))$›
from ‹*if* (*b*) c_1 *else* $c_2 \vdash n_2\ {-et_2\rightarrow}\ n_2{}'$› ‹$c_1 \vdash n\ {-et\rightarrow}\ n'$› ‹$n \neq$ (*-Entry-*)›
‹$n \oplus$ *1* $= n_2$› ‹$n' \oplus$ *1* $\neq n_2{}'$›
obtain *nx* **where** $c_1 \vdash n\ {-et_2\rightarrow}\ nx \wedge n' \neq nx$
apply − **apply**(*erule WCFG.While-CFG.cases*)
apply(*auto intro*:*WCFG.While-CFG.intros*)
apply(*drule label-incr-inj*) **apply** *auto*
apply(*drule label-incr-simp-rev*[*OF sym*])
by(*case-tac na*,*auto dest*:*WCFG-sourcelabel-less-num-nodes*)
from *IH*[*OF this*[*THEN conjunct1*] - *this*[*THEN conjunct2*]] **show** *?case* **by**

simp
next
  case (*WCFG-CondElse* c_2 *n et n′ b* c_1)
  note *IH* = ‹$\bigwedge n_2\ n_2'$. ⟦$c_2 \vdash n_2 -et_2\rightarrow n_2'$; $n = n_2$; $n' \neq n_2'$⟧
    $\Longrightarrow \exists\, Q\ Q'.\ et = (Q)_{\surd} \wedge et_2 = (Q')_{\surd} \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg$
$Q\ s))$›
  from ‹*if* (*b*) c_1 *else* $c_2 \vdash n_2 -et_2\rightarrow n_2'$› ‹$c_2 \vdash n -et\rightarrow n'$› ‹$n \neq$ (*-Entry-*)›
    ‹$n \oplus$ #:c_1 + *1* = n_2› ‹$n' \oplus$ #:c_1 + *1* $\neq n_2'$›
  obtain *nx* **where** $c_2 \vdash n -et_2\rightarrow nx \wedge n' \neq nx$
    apply − **apply**(*erule WCFG.While-CFG.cases*)
    apply(*auto intro*:*WCFG.While-CFG.intros*)
     apply(*drule label-incr-simp-rev*)
     apply(*case-tac na*,*auto*,*cases n*,*auto dest*:*WCFG-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 (*WCFG-WhileTrue b c′*)
  from ‹*while* (*b*) $c' \vdash n_2 -et_2\rightarrow n_2'$› ‹(*-0-*) = n_2› ‹(*-0-*) $\oplus$ *2* $\neq n_2'$›
  show *?case* **by** −(*erule WCFG.While-CFG.cases*,*auto*)
next
  case (*WCFG-WhileFalse b c′*)
  from ‹*while* (*b*) $c' \vdash n_2 -et_2\rightarrow n_2'$› ‹(*-0-*) = n_2› ‹(*-1-*) $\neq n_2'$›
  show *?case* **by** −(*erule WCFG.While-CFG.cases*,*auto*)
next
  case (*WCFG-WhileFalseSkip b c′*)
  from ‹*while* (*b*) $c' \vdash n_2 -et_2\rightarrow n_2'$› ‹(*-1-*) = n_2› ‹(*-Exit-*) $\neq n_2'$›
  show *?case* **by** −(*erule WCFG.While-CFG.cases*,*auto dest*:*label-incr-ge*)
next
  case (*WCFG-WhileBody c′ n et n′ b*)
  note *IH* = ‹$\bigwedge n_2\ n_2'$. ⟦$c' \vdash n_2 -et_2\rightarrow n_2'$; $n = n_2$; $n' \neq n_2'$⟧
    $\Longrightarrow \exists\, Q\ Q'.\ et = (Q)_{\surd} \wedge et_2 = (Q')_{\surd} \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg$
$Q\ s))$›
  from ‹*while* (*b*) $c' \vdash n_2 -et_2\rightarrow n_2'$› ‹$c' \vdash n -et\rightarrow n'$› ‹$n \neq$ (*-Entry-*)›
    ‹$n' \neq$ (*-Exit-*)› ‹$n \oplus$ *2* = n_2› ‹$n' \oplus$ *2* $\neq n_2'$›
  obtain *nx* **where** $c' \vdash n -et_2\rightarrow nx \wedge n' \neq nx$
    apply − **apply**(*erule WCFG.While-CFG.cases*)
    apply(*auto intro*:*WCFG.While-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 (*WCFG-WhileBodyExit c′ n et b*)
  note *IH* = ‹$\bigwedge n_2\ n_2'$. ⟦$c' \vdash n_2 -et_2\rightarrow n_2'$; $n = n_2$; (*-Exit-*) $\neq n_2'$⟧
    $\Longrightarrow \exists\, Q\ Q'.\ et = (Q)_{\surd} \wedge et_2 = (Q')_{\surd} \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg$
$Q\ s))$›
  from ‹*while* (*b*) $c' \vdash n_2 -et_2\rightarrow n_2'$› ‹$c' \vdash n -et\rightarrow$ (*-Exit-*)› ‹$n \neq$ (*-Entry-*)›

‹$n \oplus 2 = n_2$› ‹(-0-) $\neq$ n_2'›
obtain *nx* **where** $c' \vdash n -et_2\rightarrow nx \wedge$ (-*Exit*-) $\neq$ *nx*
apply − **apply**(*erule WCFG.While-CFG.cases*)
apply(*auto intro*:*WCFG.While-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*
qed

end

4.3 Instantiate CFG locale with While CFG

theory *Interpretation* **imports**
WCFG
../Basic/CFGExit
begin

4.3.1 Instatiation of the *CFG* locale

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

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

abbreviation *kind* :: *w-edge* ⇒ *state edge-kind*
where *kind e* ≡ *fst*(*snd e*)

definition *valid-edge* :: *cmd* ⇒ *w-edge* ⇒ *bool*
where *valid-edge prog a* ≡ *prog* ⊢ *sourcenode a* −*kind a*→ *targetnode a*

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

lemma *While-CFG-aux*:
CFG sourcenode targetnode (*valid-edge prog*) *Entry*
proof(*unfold-locales*)
fix *a* **assume** *valid-edge prog a* **and** *targetnode a* = (-*Entry*-)
obtain *nx et nx*′ **where** *a* = (*nx*,*et*,*nx*′) **by** (*cases a*) *auto*
with ‹*valid-edge prog a*› ‹*targetnode a* = (-*Entry*-)›
have *prog* ⊢ *nx* −*et*→ (-*Entry*-) **by**(*simp add*:*valid-edge-def*)
thus *False* **by** *fastforce*
next
fix *a a*′

assume *assms*:*valid-edge prog a valid-edge prog a'*
sourcenode a = *sourcenode a' targetnode a* = *targetnode a'*
obtain *x et y* **where** [*simp*]:*a* = (*x*,*et*,*y*) **by** (*cases a*) *auto*
obtain *x' et' y'* **where** [*simp*]:*a'* = (*x'*,*et'*,*y'*) **by** (*cases a'*) *auto*
from *assms* **have** *et* = *et'*
by(*fastforce intro*:*WCFG-edge-det simp*:*valid-edge-def*)
with ‹*sourcenode a* = *sourcenode a'*› ‹*targetnode a* = *targetnode a'*›
show *a* = *a'* **by** *simp*
qed

interpretation *While-CFG*:
CFG sourcenode targetnode kind valid-edge prog Entry
for *prog*
by(*rule While-CFG-aux*)

lemma *While-CFGExit-aux*:
CFGExit sourcenode targetnode kind (*valid-edge prog*) *Entry Exit*
proof(*unfold-locales*)
fix *a* **assume** *valid-edge prog a* **and** *sourcenode a* = (*-Exit-*)
obtain *nx et nx'* **where** *a* = (*nx*,*et*,*nx'*) **by** (*cases a*) *auto*
with ‹*valid-edge prog a*› ‹*sourcenode a* = (*-Exit-*)›
have *prog* ⊢ (*-Exit-*) −*et*→ *nx'* **by**(*simp add*:*valid-edge-def*)
thus *False* **by** *fastforce*
next
have *prog* ⊢ (*-Entry-*) −(λ*s. False*)$_\surd$→ (*-Exit-*) **by**(*rule WCFG-Entry-Exit*)
thus ∃ *a. valid-edge prog a* ∧ *sourcenode a* = (*-Entry-*) ∧
targetnode a = (*-Exit-*) ∧ *kind a* = (λ*s. False*)$_\surd$
by(*fastforce simp*:*valid-edge-def*)
qed

interpretation *While-CFGExit*:
CFGExit sourcenode targetnode kind valid-edge prog Entry Exit
for *prog*
by(*rule While-CFGExit-aux*)

end

4.4 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*

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*+)
qed

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

lemma *less-num-inner-nodes-label*:
  $l < \#{:}c \Longrightarrow \exists\, c'.\ labels\ c\ l\ c'$
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** $l = 0 \lor l = 1$ **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* $l' < \#{:}c_1$)
      case *True*
      from *IH1*[*OF this*] **obtain** c' **where** $labels\ c_1\ l'\ c'$ **by** *auto*
      with ‹$l = l' + 1$› **have** $labels\ (if\ (b)\ c_1\ else\ c_2)\ l\ c'$

```
      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
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* c_1 *l c* c_2)
note *IH* = ‹$\bigwedge c''$. *labels* c_1 *l c''* $\Longrightarrow$ *c = c''*›
from ‹*labels* c_1 *l c*› **have** *l* < #:c_1 **by**(*fastforce intro*:*label-less-num-inner-nodes*)
with ‹*labels* (c_1;;c_2) *l c''*› **obtain** *cx* **where** *c''* = *cx*;;c_2 ∧ *labels* c_1 *l cx*
by(*fastforce elim*:*labels.cases intro*:*Labels-Base*)
hence [*simp*]:*c''* = *cx*;;c_2 **and** *labels* c_1 *l cx* **by** *simp-all*
from *IH*[*OF* ‹*labels* c_1 *l cx*›] **show** *?case* **by** *simp*
next
case (*Labels-Seq2* c_2 *l c* c_1)
note *IH* = ‹$\bigwedge c''$. *labels* c_2 *l c''* $\Longrightarrow$ *c = c''*›
from ‹*labels* (c_1;;c_2) (*l* + #:c_1) *c''*› ‹*labels* c_2 *l c*› **have** *labels* c_2 *l c''*
by(*auto elim*:*labels.cases dest*:*label-less-num-inner-nodes*)
from *IH*[*OF this*] **show** *?case* **.**
next
case (*Labels-CondTrue* c_1 *l c b* c_2)
note *IH* = ‹$\bigwedge c''$. *labels* c_1 *l c''* $\Longrightarrow$ *c = c''*›
from ‹*labels* (*if* (*b*) c_1 *else* c_2) (*l* + *1*) *c''*› ‹*labels* c_1 *l c*› **have** *labels* c_1 *l c''*
by(*fastforce elim*:*labels.cases dest*:*label-less-num-inner-nodes*)
from *IH*[*OF this*] **show** *?case* **.**
next
case (*Labels-CondFalse* c_2 *l c b* c_1)
note *IH* = ‹$\bigwedge c''$. *labels* c_2 *l c''* $\Longrightarrow$ *c = c''*›
from ‹*labels* (*if* (*b*) c_1 *else* c_2) (*l* + #:c_1 + *1*) *c''*› ‹*labels* c_2 *l c*›
have *labels* c_2 *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* = ‹$\bigwedge c''$. *labels c' l c''* $\Longrightarrow$ *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*)+

end

4.5 General well-formedness of While CFG

theory *WellFormed* **imports**
Interpretation
Labels
../Basic/CFGExit-wf
../StaticIntra/CDepInstantiations

begin

4.5.1 Definition of some functions

```
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)     = {}

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

fun rhs :: cmd ⇒ vname set
where
  rhs Skip                = {}
  | rhs (V:=e)            = rhs-aux e
  | rhs (c₁;;c₂)          = rhs c₁
  | rhs (if (b) c₁ else c₂) = rhs-aux b
  | rhs (while (b) c)     = rhs-aux b


lemma rhs-interpret-eq:
  ⟦interpret b s = Some v'; ∀ V ∈ rhs-aux b. s V = s' V⟧
   ⟹ interpret b s' = Some v'
proof(induct b arbitrary:v')
  case (Val v)
  from ‹interpret (Val v) s = Some v'› have v' = v by(fastforce elim:interpret.cases)
  thus ?case by simp
next
  case (Var V)
  hence s' V = Some v' by(fastforce elim:interpret.cases)
  thus ?case by simp
next
  case (BinOp b1 bop b2)
  note IH1 = ‹⋀v'. ⟦interpret b1 s = Some v'; ∀ V ∈ rhs-aux b1. s V = s' V⟧
             ⟹ interpret b1 s' = Some v'›
  note IH2 = ‹⋀v'. ⟦interpret b2 s = Some v'; ∀ V ∈ rhs-aux b2. s V = s' V⟧
             ⟹ interpret b2 s' = Some v'›
  from ‹interpret (b1 «bop» b2) s = Some v'›
  have ∃ v₁ v₂. interpret b1 s = Some v₁ ∧ interpret b2 s = Some v₂ ∧
                binop bop v₁ v₂ = Some v'
    apply(cases interpret b1 s,simp)
    apply(cases interpret b2 s,simp)
```

by(*case-tac binop bop a aa,simp+*)
then obtain v_1 v_2 **where** *interpret b1 s = Some* v_1
and *interpret b2 s = Some* v_2 **and** *binop bop* v_1 v_2 *= Some v'* **by** *blast*
from ‹∀ *V* ∈ *rhs-aux* (*b1* «*bop*» *b2*). *s V* = *s' V*› **have** ∀ *V* ∈ *rhs-aux b1*. *s V* = *s' V*
by *simp*
from *IH1*[*OF* ‹*interpret b1 s = Some* v_1› *this*] **have** *interpret b1 s' = Some* v_1 **.**
from ‹∀ *V* ∈ *rhs-aux* (*b1* «*bop*» *b2*). *s V* = *s' V*› **have** ∀ *V* ∈ *rhs-aux b2*. *s V* = *s' V*
by *simp*
from *IH2*[*OF* ‹*interpret b2 s = Some* v_2› *this*] **have** *interpret b2 s' = Some* v_2 **.**
with ‹*interpret b1 s' = Some* v_1› ‹*binop bop* v_1 v_2 *= Some v'*› **show** *?case* **by** *simp*
qed

fun *Defs* :: *cmd* ⇒ *w-node* ⇒ *vname set*
where *Defs prog n* = {*V*. ∃ *l c*. *n* = (- *l* -) ∧ *labels prog l c* ∧ *V* ∈ *lhs c*}

fun *Uses* :: *cmd* ⇒ *w-node* ⇒ *vname set*
where *Uses prog n* = {*V*. ∃ *l c*. *n* = (- *l* -) ∧ *labels prog l c* ∧ *V* ∈ *rhs c*}

4.5.2 Lemmas about *prog* ⊢ *n* −*et*→ *n'* to show well-formed properties

lemma *WCFG-edge-no-Defs-equal*:
⟦*prog* ⊢ *n* −*et*→ *n'*; *V* ∉ *Defs prog n*⟧ ⟹ (*transfer et s*) *V* = *s V*
proof(*induct rule*:*WCFG-induct*)
case (*WCFG-LAss V' e*)
have *label*:*labels* (*V'*:=*e*) *0* (*V'*:=*e*) **and** *lhs*:*V'* ∈ *lhs* (*V'*:=*e*)
by(*auto intro*:*Labels-Base*)
hence *V'* ∈ *Defs* (*V'*:=*e*) (-*0*-) **by** *fastforce*
with ‹*V* ∉ *Defs* (*V'*:=*e*) (-*0*-)› **show** *?case* **by** *auto*
next
case (*WCFG-SeqFirst* c_1 *n et n'* c_2)
note *IH* = ‹*V* ∉ *Defs* c_1 *n* ⟹ *transfer et s V* = *s V*›
have *V* ∉ *Defs* c_1 *n*
proof
assume *V* ∈ *Defs* c_1 *n*
then obtain *c l* **where** [*simp*]:*n* = (- *l* -) **and** *labels* c_1 *l c*
and *V* ∈ *lhs c* **by** *fastforce*
from ‹*labels* c_1 *l c*› **have** *labels* (c_1;;c_2) *l* (*c*;;c_2)
by(*fastforce intro*:*Labels-Seq1*)
from ‹*V* ∈ *lhs c*› **have** *V* ∈ *lhs* (*c*;;c_2) **by** *simp*
with ‹*labels* (c_1;;c_2) *l* (*c*;;c_2)› **have** *V* ∈ *Defs* (c_1;;c_2) *n* **by** *fastforce*
with ‹*V* ∉ *Defs* (c_1;;c_2) *n*› **show** *False* **by** *fastforce*
qed
from *IH*[*OF this*] **show** *?case* **.**
next

case (*WCFG-SeqConnect* c_1 *n et* c_2)
note *IH* = ‹*V* ∉ *Defs* c_1 *n* ⟹ *transfer et s V* = *s V*›
have *V* ∉ *Defs* c_1 *n*
proof
  assume *V* ∈ *Defs* c_1 *n*
  then obtain *c l* **where** [*simp*]:*n* = (- *l* -) **and** *labels* c_1 *l c*
    and *V* ∈ *lhs c* **by** *fastforce*
  from ‹*labels* c_1 *l c*› **have** *labels* (c_1;;c_2) *l* (*c*;;c_2)
    by(*fastforce intro*:*Labels-Seq1*)
  from ‹*V* ∈ *lhs c*› **have** *V* ∈ *lhs* (*c*;;c_2) **by** *simp*
  with ‹*labels* (c_1;;c_2) *l* (*c*;;c_2)› **have** *V* ∈ *Defs* (c_1;;c_2) *n* **by** *fastforce*
  with ‹*V* ∉ *Defs* (c_1;;c_2) *n*› **show** *False* **by** *fastforce*
qed
from *IH*[*OF this*] **show** *?case* **.**
next
case (*WCFG-SeqSecond* c_2 *n et n'* c_1)
note *IH* = ‹*V* ∉ *Defs* c_2 *n* ⟹ *transfer et s V* = *s V*›
have *V* ∉ *Defs* c_2 *n*
proof
  assume *V* ∈ *Defs* c_2 *n*
  then obtain *c l* **where** [*simp*]:*n* = (- *l* -) **and** *labels* c_2 *l c*
    and *V* ∈ *lhs c* **by** *fastforce*
  from ‹*labels* c_2 *l c*› **have** *labels* (c_1;;c_2) (*l* + #:c_1) *c*
    by(*fastforce intro*:*Labels-Seq2*)
  with ‹*V* ∈ *lhs c*› **have** *V* ∈ *Defs* (c_1;;c_2) (*n* ⊕ #:c_1) **by** *fastforce*
  with ‹*V* ∉ *Defs* (c_1;;c_2) (*n* ⊕ #:c_1)› **show** *False* **by** *fastforce*
qed
from *IH*[*OF this*] **show** *?case* **.**
next
case (*WCFG-CondThen* c_1 *n et n' b* c_2)
note *IH* = ‹*V* ∉ *Defs* c_1 *n* ⟹ *transfer et s V* = *s V*›
have *V* ∉ *Defs* c_1 *n*
proof
  assume *V* ∈ *Defs* c_1 *n*
  then obtain *c l* **where** [*simp*]:*n* = (- *l* -) **and** *labels* c_1 *l c*
    and *V* ∈ *lhs c* **by** *fastforce*
  from ‹*labels* c_1 *l c*› **have** *labels* (*if* (*b*) c_1 *else* c_2) (*l* + *1*) *c*
    by(*fastforce intro*:*Labels-CondTrue*)
  with ‹*V* ∈ *lhs c*› **have** *V* ∈ *Defs* (*if* (*b*) c_1 *else* c_2) (*n* ⊕ *1*) **by** *fastforce*
  with ‹*V* ∉ *Defs* (*if* (*b*) c_1 *else* c_2) (*n* ⊕ *1*)› **show** *False* **by** *fastforce*
qed
from *IH*[*OF this*] **show** *?case* **.**
next
case (*WCFG-CondElse* c_2 *n et n' b* c_1)
note *IH* = ‹*V* ∉ *Defs* c_2 *n* ⟹ *transfer et s V* = *s V*›
have *V* ∉ *Defs* c_2 *n*
proof
  assume *V* ∈ *Defs* c_2 *n*
  then obtain *c l* **where** [*simp*]:*n* = (- *l* -) **and** *labels* c_2 *l c*

and *V* ∈ *lhs c* **by** *fastforce*
from ‹*labels* c_2 *l c*› **have** *labels* (*if* (*b*) c_1 *else* c_2) (*l* + #:c_1 + *1*) *c*
by(*fastforce intro*:*Labels-CondFalse*)
with ‹*V* ∈ *lhs c*› **have** *V* ∈ *Defs* (*if* (*b*) c_1 *else* c_2) (*n* ⊕ #:c_1 + *1*)
by(*fastforce simp*:*add.commute add.left-commute*)
with ‹*V* ∉ *Defs* (*if* (*b*) c_1 *else* c_2) (*n* ⊕ #:c_1 + *1*)› **show** *False* **by** *fastforce*
qed
from *IH*[*OF this*] **show** *?case* **.**
next
case (*WCFG-WhileBody c′ n et n′ b*)
note *IH* = ‹*V* ∉ *Defs c′ n* ⟹ *transfer et s V* = *s V*›
have *V* ∉ *Defs c′ n*
proof
assume *V* ∈ *Defs c′ n*
then obtain *c l* **where** [*simp*]:*n* = (- *l* -) **and** *labels c′ l c*
and *V* ∈ *lhs c* **by** *fastforce*
from ‹*labels c′ l c*› **have** *labels* (*while* (*b*) *c′*) (*l* + *2*) (*c*;;*while* (*b*) *c′*)
by(*fastforce intro*:*Labels-WhileBody*)
from ‹*V* ∈ *lhs c*› **have** *V* ∈ *lhs* (*c*;;*while* (*b*) *c′*) **by** *fastforce*
with ‹*labels* (*while* (*b*) *c′*) (*l* + *2*) (*c*;;*while* (*b*) *c′*)›
have *V* ∈ *Defs* (*while* (*b*) *c′*) (*n* ⊕ *2*) **by** *fastforce*
with ‹*V* ∉ *Defs* (*while* (*b*) *c′*) (*n* ⊕ *2*)› **show** *False* **by** *fastforce*
qed
from *IH*[*OF this*] **show** *?case* **.**
next
case (*WCFG-WhileBodyExit c′ n et b*)
note *IH* = ‹*V* ∉ *Defs c′ n* ⟹ *transfer et s V* = *s V*›
have *V* ∉ *Defs c′ n*
proof
assume *V* ∈ *Defs c′ n*
then obtain *c l* **where** [*simp*]:*n* = (- *l* -) **and** *labels c′ l c*
and *V* ∈ *lhs c* **by** *fastforce*
from ‹*labels c′ l c*› **have** *labels* (*while* (*b*) *c′*) (*l* + *2*) (*c*;;*while* (*b*) *c′*)
by(*fastforce intro*:*Labels-WhileBody*)
from ‹*V* ∈ *lhs c*› **have** *V* ∈ *lhs* (*c*;;*while* (*b*) *c′*) **by** *fastforce*
with ‹*labels* (*while* (*b*) *c′*) (*l* + *2*) (*c*;;*while* (*b*) *c′*)›
have *V* ∈ *Defs* (*while* (*b*) *c′*) (*n* ⊕ *2*) **by** *fastforce*
with ‹*V* ∉ *Defs* (*while* (*b*) *c′*) (*n* ⊕ *2*)› **show** *False* **by** *fastforce*
qed
from *IH*[*OF this*] **show** *?case* **.**
qed *auto*

lemma *WCFG-edge-transfer-uses-only-Uses*:
⟦*prog* ⊢ *n* −*et*→ *n′*; ∀ *V* ∈ *Uses prog n*. *s V* = *s′ V*⟧
⟹ ∀ *V* ∈ *Defs prog n*. (*transfer et s*) *V* = (*transfer et s′*) *V*
proof(*induct rule*:*WCFG-induct*)
case (*WCFG-LAss V e*)
have *Uses* (*V*:=*e*) (-*0*-) = {*V*. *V* ∈ *rhs-aux e*}

by(*fastforce elim*:*labels.cases intro*:*Labels-Base*)
with ‹∀ *V'*∈*Uses* (*V*:=*e*) (-*0*-). *s V'* = *s' V'*›
have ∀ *V'*∈*rhs-aux e*. *s V'* = *s' V'* **by** *blast*
have *Defs* (*V*:=*e*) (-*0*-) = {*V*}
by(*fastforce elim*:*labels.cases intro*:*Labels-Base*)
have *transfer* ⇑λ*s*. *s*(*V* := *interpret e s*) *s V* =
transfer ⇑λ*s*. *s*(*V* := *interpret e s*) *s' V*
proof(*cases interpret e s*)
case *None*
{ **fix** *v* **assume** *interpret e s'* = *Some v*
with ‹∀ *V'*∈*rhs-aux e*. *s V'* = *s' V'*› **have** *interpret e s* = *Some v*
by(*fastforce intro*:*rhs-interpret-eq*)
with *None* **have** *False* **by**(*fastforce split*:*if-split-asm*) }
with *None* **show** *?thesis* **by** *fastforce*
next
case (*Some v*)
hence *interpret e s* = *Some v* **by**(*fastforce split*:*if-split-asm*)
with ‹∀ *V'*∈*rhs-aux e*. *s V'* = *s' V'*›
have *interpret e s'* = *Some v* **by**(*fastforce intro*:*rhs-interpret-eq*)
with *Some* **show** *?thesis* **by** *simp*
qed
with ‹*Defs* (*V*:=*e*) (-*0*-) = {*V*}› **show** *?case* **by** *simp*
next
case (*WCFG-SeqFirst* c_1 *n et n'* c_2)
note *IH* = ‹∀ *V*∈*Uses* c_1 *n*. *s V* = *s' V*
⟹ ∀ *V*∈*Defs* c_1 *n*. *transfer et s V* = *transfer et s' V*›
from ‹∀ *V*∈*Uses* (c_1;;c_2) *n*. *s V* = *s' V*› **have** ∀ *V*∈*Uses* c_1 *n*. *s V* = *s' V*
by *auto*(*drule Labels-Seq1*[*of* - - - c_2],*erule-tac x*=*V* **in** *allE*,*auto*)
from *IH*[*OF this*] **have** ∀ *V*∈*Defs* c_1 *n*. *transfer et s V* = *transfer et s' V* **.**
with ‹c_1 ⊢ *n* −*et*→ *n'*› **show** *?case* **using** *Labels-Base*
apply *clarsimp*
apply(*erule labels.cases*,*auto dest*:*WCFG-sourcelabel-less-num-nodes*)
by(*erule-tac x*=*V* **in** *allE*,*fastforce*)
next
case (*WCFG-SeqConnect* c_1 *n et* c_2)
note *IH* = ‹∀ *V*∈*Uses* c_1 *n*. *s V* = *s' V*
⟹ ∀ *V*∈*Defs* c_1 *n*. *transfer et s V* = *transfer et s' V*›
from ‹∀ *V*∈*Uses* (c_1;;c_2) *n*. *s V* = *s' V*› **have** ∀ *V*∈*Uses* c_1 *n*. *s V* = *s' V*
by *auto*(*drule Labels-Seq1*[*of* - - - c_2],*erule-tac x*=*V* **in** *allE*,*auto*)
from *IH*[*OF this*] **have** ∀ *V*∈*Defs* c_1 *n*. *transfer et s V* = *transfer et s' V* **.**
with ‹c_1 ⊢ *n* −*et*→ (-*Exit*-)› **show** *?case* **using** *Labels-Base*
apply *clarsimp*
apply(*erule labels.cases*,*auto dest*:*WCFG-sourcelabel-less-num-nodes*)
by(*erule-tac x*=*V* **in** *allE*,*fastforce*)
next
case (*WCFG-SeqSecond* c_2 *n et n'* c_1)
note *IH* = ‹∀ *V*∈*Uses* c_2 *n*. *s V* = *s' V*
⟹ ∀ *V*∈*Defs* c_2 *n*. *transfer et s V* = *transfer et s' V*›
from ‹∀ *V*∈*Uses* (c_1;;c_2) (*n* ⊕ #:c_1). *s V* = *s' V*› **have** ∀ *V*∈*Uses* c_2 *n*. *s V* =

s′ V
  by(*auto*,*blast dest*:*Labels-Seq2*)
 from *IH*[*OF this*] **have** ∀ *V*∈*Defs* c_2 *n*. *transfer et s V* = *transfer et s′ V* **.**
 with *num-inner-nodes-gr-0*[*of* c_1] **show** *?case*
  apply *clarsimp*
  apply(*erule labels.cases*,*auto*)
  by(*cases n*,*auto dest*:*label-less-num-inner-nodes*)+
next
 case (*WCFG-CondThen* c_1 *n et n′ b* c_2)
 note *IH* = ‹∀ *V*∈*Uses* c_1 *n*. *s V* = *s′ V*
  ⟹ ∀ *V*∈*Defs* c_1 *n*. *transfer et s V* = *transfer et s′ V*›
 from ‹∀ *V*∈*Uses* (*if* (*b*) c_1 *else* c_2) (*n* ⊕ *1*). *s V* = *s′ V*›
 have ∀ *V*∈*Uses* c_1 *n*. *s V* = *s′ V* **by**(*auto*,*blast dest*:*Labels-CondTrue*)
 from *IH*[*OF this*] **have** ∀ *V*∈*Defs* c_1 *n*. *transfer et s V* = *transfer et s′ V* **.**
 with ‹c_1 ⊢ *n* −*et*→ *n′*› **show** *?case*
  apply *clarsimp*
  apply(*erule labels.cases*,*auto*)
  apply(*cases n*,*auto dest*:*label-less-num-inner-nodes*)
  by(*cases n*,*auto dest*:*WCFG-sourcelabel-less-num-nodes*)
next
 case (*WCFG-CondElse* c_2 *n et n′ b* c_1)
 note *IH* = ‹∀ *V*∈*Uses* c_2 *n*. *s V* = *s′ V*
  ⟹ ∀ *V*∈*Defs* c_2 *n*. *transfer et s V* = *transfer et s′ V*›
 from ‹∀ *V*∈*Uses* (*if* (*b*) c_1 *else* c_2) (*n* ⊕ #:c_1 + *1*). *s V* = *s′ V*›
 have ∀ *V*∈*Uses* c_2 *n*. *s V* = *s′ V*
  by *auto*(*drule Labels-CondFalse*[*of - - - b* c_1],*erule-tac x*=*V* **in** *allE*,
   auto simp:*add.assoc*)
 from *IH*[*OF this*] **have** ∀ *V*∈*Defs* c_2 *n*. *transfer et s V* = *transfer et s′ V* **.**
 with ‹c_2 ⊢ *n* −*et*→ *n′*› **show** *?case*
  apply *clarsimp*
  apply(*erule labels.cases*,*auto*)
  apply(*cases n*,*auto dest*:*label-less-num-inner-nodes*)
  by(*cases n*,*auto dest*:*WCFG-sourcelabel-less-num-nodes*)
next
 case (*WCFG-WhileBody c′ n et n′ b*)
 note *IH* = ‹∀ *V*∈*Uses c′ n*. *s V* = *s′ V*
  ⟹ ∀ *V*∈*Defs c′ n*. *transfer et s V* = *transfer et s′ V*›
 from ‹∀ *V*∈*Uses* (*while* (*b*) *c′*) (*n* ⊕ *2*). *s V* = *s′ V*› **have** ∀ *V*∈*Uses c′ n*. *s V*
= *s′ V*
  by *auto*(*drule Labels-WhileBody*[*of - - - b*],*erule-tac x*=*V* **in** *allE*,*auto*)
 from *IH*[*OF this*] **have** ∀ *V*∈*Defs c′ n*. *transfer et s V* = *transfer et s′ V* **.**
 thus *?case*
  apply *clarsimp*
  apply(*erule labels.cases*,*auto*)
  by(*cases n*,*auto dest*:*label-less-num-inner-nodes*)
next
 case (*WCFG-WhileBodyExit c′ n et b*)
 note *IH* = ‹∀ *V*∈*Uses c′ n*. *s V* = *s′ V*
  ⟹ ∀ *V*∈*Defs c′ n*. *transfer et s V* = *transfer et s′ V*›

from ‹∀ V∈*Uses* (*while* (*b*) *c'*) (*n* ⊕ *2*). *s V* = *s' V*› **have** ∀ V∈*Uses c' n*. *s V* = *s' V*
  by *auto*(*drule Labels-WhileBody*[*of - - - b*],*erule-tac x*=*V* **in** *allE*,*auto*)
 from *IH*[*OF this*] **have** ∀ V∈*Defs c' n*. *transfer et s V* = *transfer et s' V* **.**
 thus *?case*
  apply *clarsimp*
  apply(*erule labels.cases*,*auto*)
  by(*cases n*,*auto dest*:*label-less-num-inner-nodes*)
qed (*fastforce elim*:*labels.cases*)+

lemma *WCFG-edge-Uses-pred-eq*:
 ⟦*prog* ⊢ *n* −*et*→ *n'*; ∀ V ∈ *Uses prog n*. *s V* = *s' V*; *pred et s*⟧
  ⟹ *pred et s'*
proof(*induct rule*:*WCFG-induct*)
 case (*WCFG-SeqFirst* c_1 *n et n'* c_2)
 note *IH* = ‹⟦∀ V∈*Uses* c_1 *n*. *s V* = *s' V*; *pred et s*⟧ ⟹ *pred et s'*›
 from ‹∀ V∈*Uses* (c_1;; c_2) *n*. *s V* = *s' V*› **have** ∀ V∈*Uses* c_1 *n*. *s V* = *s' V*
  by *auto*(*drule Labels-Seq1*[*of - - -* c_2],*erule-tac x*=*V* **in** *allE*,*auto*)
 from *IH*[*OF this* ‹*pred et s*›] **show** *?case* **.**
next
 case (*WCFG-SeqConnect* c_1 *n et* c_2)
 note *IH* = ‹⟦∀ V∈*Uses* c_1 *n*. *s V* = *s' V*; *pred et s*⟧ ⟹ *pred et s'*›
 from ‹∀ V∈*Uses* (c_1;; c_2) *n*. *s V* = *s' V*› **have** ∀ V∈*Uses* c_1 *n*. *s V* = *s' V*
  by *auto*(*drule Labels-Seq1*[*of - - -* c_2],*erule-tac x*=*V* **in** *allE*,*auto*)
 from *IH*[*OF this* ‹*pred et s*›] **show** *?case* **.**
next
 case (*WCFG-SeqSecond* c_2 *n et n'* c_1)
 note *IH* = ‹⟦∀ V∈*Uses* c_2 *n*. *s V* = *s' V*; *pred et s*⟧ ⟹ *pred et s'*›
 from ‹∀ V∈*Uses* (c_1;; c_2) (*n* ⊕ #:c_1). *s V* = *s' V*›
 have ∀ V∈*Uses* c_2 *n*. *s V* = *s' V* **by**(*auto*,*blast dest*:*Labels-Seq2*)
 from *IH*[*OF this* ‹*pred et s*›] **show** *?case* **.**
next
 case (*WCFG-CondTrue b* c_1 c_2)
 from ‹∀ V∈*Uses* (*if* (*b*) c_1 *else* c_2) (*-0-*). *s V* = *s' V*›
 have *all*:∀ V. *labels* (*if* (*b*) c_1 *else* c_2) *0* (*if* (*b*) c_1 *else* c_2) ∧
      V ∈ *rhs* (*if* (*b*) c_1 *else* c_2) ⟶ (*s V* = *s' V*)
  by *fastforce*
 obtain *v'* **where** [*simp*]:*v'* = *true* **by** *simp*
 with ‹*pred* (λ*s*. *interpret b s* = *Some true*)$_{\surd}$ *s*›
 have *interpret b s* = *Some v'* **by** *simp*
 have *labels* (*if* (*b*) c_1 *else* c_2) *0* (*if* (*b*) c_1 *else* c_2) **by**(*rule Labels-Base*)
 with *all* **have** ∀ V ∈ *rhs-aux b*. *s V* = *s' V* **by** *simp*
 with ‹*interpret b s* = *Some v'*› **have** *interpret b s'* = *Some v'*
  by(*rule rhs-interpret-eq*)
 thus *?case* **by** *simp*
next
 case (*WCFG-CondFalse b* c_1 c_2)
 from ‹∀ V∈*Uses* (*if* (*b*) c_1 *else* c_2) (*-0-*). *s V* = *s' V*›

have *all*:∀ V. *labels* (*if* (*b*) c_1 *else* c_2) *0* (*if* (*b*) c_1 *else* c_2) ∧
$V \in$ *rhs* (*if* (*b*) c_1 *else* c_2) ⟶ ($s\ V = s'\ V$)
by *fastforce*
obtain v' **where** [*simp*]:v' = *false* **by** *simp*
with ‹*pred* (λ*s*. *interpret b s* = *Some false*)$_\surd$ *s*›
have *interpret b s* = *Some* v' **by** *simp*
have *labels* (*if* (*b*) c_1 *else* c_2) *0* (*if* (*b*) c_1 *else* c_2) **by**(*rule Labels-Base*)
with *all* **have** ∀ $V \in$ *rhs-aux b*. $s\ V = s'\ V$ **by** *simp*
with ‹*interpret b s* = *Some* v'› **have** *interpret b* s' = *Some* v'
by(*rule rhs-interpret-eq*)
thus *?case* **by** *simp*
next
case (*WCFG-CondThen* c_1 *n et* n' *b* c_2)
note *IH* = ‹⟦∀ *V*∈*Uses* c_1 *n*. $s\ V = s'\ V$; *pred et s*⟧ ⟹ *pred et* s'›
from ‹∀ *V*∈*Uses* (*if* (*b*) c_1 *else* c_2) (*n* ⊕ *1*). $s\ V = s'\ V$›
have ∀ *V*∈*Uses* c_1 *n*. $s\ V = s'\ V$ **by**(*auto*,*blast dest*:*Labels-CondTrue*)
from *IH*[*OF this* ‹*pred et s*›] **show** *?case* **.**
next
case (*WCFG-CondElse* c_2 *n et* n' *b* c_1)
note *IH* = ‹⟦∀ *V*∈*Uses* c_2 *n*. $s\ V = s'\ V$; *pred et s*⟧ ⟹ *pred et* s'›
from ‹∀ *V*∈*Uses* (*if* (*b*) c_1 *else* c_2) (*n* ⊕ #:c_1 + *1*). $s\ V = s'\ V$›
have ∀ *V*∈*Uses* c_2 *n*. $s\ V = s'\ V$
by *auto*(*drule Labels-CondFalse*[*of - - - b* c_1],*erule-tac x*=*V* **in** *allE*,
auto simp:*add.assoc*)
from *IH*[*OF this* ‹*pred et s*›] **show** *?case* **.**
next
case (*WCFG-WhileTrue b* c')
from ‹∀ *V*∈*Uses* (*while* (*b*) c') (*-0-*). $s\ V = s'\ V$›
have *all*:∀ V. *labels* (*while* (*b*) c') *0* (*while* (*b*) c') ∧
$V \in$ *rhs* (*while* (*b*) c') ⟶ ($s\ V = s'\ V$)
by *fastforce*
obtain v' **where** [*simp*]:v' = *true* **by** *simp*
with ‹*pred* (λ*s*. *interpret b s* = *Some true*)$_\surd$ *s*›
have *interpret b s* = *Some* v' **by** *simp*
have *labels* (*while* (*b*) c') *0* (*while* (*b*) c') **by**(*rule Labels-Base*)
with *all* **have** ∀ $V \in$ *rhs-aux b*. $s\ V = s'\ V$ **by** *simp*
with ‹*interpret b s* = *Some* v'› **have** *interpret b* s' = *Some* v'
by(*rule rhs-interpret-eq*)
thus *?case* **by** *simp*
next
case (*WCFG-WhileFalse b* c')
from ‹∀ *V*∈*Uses* (*while* (*b*) c') (*-0-*). $s\ V = s'\ V$›
have *all*:∀ V. *labels* (*while* (*b*) c') *0* (*while* (*b*) c') ∧
$V \in$ *rhs* (*while* (*b*) c') ⟶ ($s\ V = s'\ V$)
by *fastforce*
obtain v' **where** [*simp*]:v' = *false* **by** *simp*
with ‹*pred* (λ*s*. *interpret b s* = *Some false*)$_\surd$ *s*›
have *interpret b s* = *Some* v' **by** *simp*
have *labels* (*while* (*b*) c') *0* (*while* (*b*) c') **by**(*rule Labels-Base*)

with *all* **have** ∀ *V* ∈ *rhs-aux b*. *s V* = *s′ V* **by** *simp*
with ‹*interpret b s* = *Some v′*› **have** *interpret b s′* = *Some v′*
by(*rule rhs-interpret-eq*)
thus *?case* **by** *simp*
next
case (*WCFG-WhileBody c′ n et n′ b*)
note *IH* = ‹⟦∀ *V*∈*Uses c′ n*. *s V* = *s′ V*; *pred et s*⟧ ⟹ *pred et s′*›
from ‹∀ *V*∈*Uses* (*while* (*b*) *c′*) (*n* ⊕ *2*). *s V* = *s′ V*› **have** ∀ *V*∈*Uses c′ n*. *s V* = *s′ V*
by *auto*(*drule Labels-WhileBody*[*of - - - b*],*erule-tac x*=*V* **in** *allE*,*auto*)
from *IH*[*OF this* ‹*pred et s*›] **show** *?case* **.**
next
case (*WCFG-WhileBodyExit c′ n et b*)
note *IH* = ‹⟦∀ *V*∈*Uses c′ n*. *s V* = *s′ V*; *pred et s*⟧ ⟹ *pred et s′*›
from ‹∀ *V*∈*Uses* (*while* (*b*) *c′*) (*n* ⊕ *2*). *s V* = *s′ V*› **have** ∀ *V*∈*Uses c′ n*. *s V* = *s′ V*
by *auto*(*drule Labels-WhileBody*[*of - - - b*],*erule-tac x*=*V* **in** *allE*,*auto*)
from *IH*[*OF this* ‹*pred et s*›] **show** *?case* **.**
qed *simp-all*

interpretation *While-CFG-wf*: *CFG-wf sourcenode targetnode kind valid-edge prog Entry Defs prog Uses prog id*
for *prog*
proof(*unfold-locales*)
show *Defs prog* (*-Entry-*) = {} ∧ *Uses prog* (*-Entry-*) = {}
by(*simp add*:*Defs.simps Uses.simps*)
next
fix *a V s*
assume *valid-edge prog a* **and** *V* ∉ *Defs prog* (*sourcenode a*)
obtain *nx et nx′* **where** [*simp*]:*a* = (*nx*,*et*,*nx′*) **by**(*cases a*) *auto*
with ‹*valid-edge prog a*› **have** *prog* ⊢ *nx* −*et*→ *nx′* **by**(*simp add*:*valid-edge-def*)
with ‹*V* ∉ *Defs prog* (*sourcenode a*)› **show** *id* (*transfer* (*kind a*) *s*) *V* = *id s V*
by(*fastforce intro*: *WCFG-edge-no-Defs-equal*)
next
fix *a* **fix** *s s′*::*state*
assume *valid-edge prog a*
and ∀ *V*∈*Uses prog* (*sourcenode a*). *id s V* = *id s′ V*
obtain *nx et nx′* **where** [*simp*]:*a* = (*nx*,*et*,*nx′*) **by**(*cases a*) *auto*
with ‹*valid-edge prog a*› **have** *prog* ⊢ *nx* −*et*→ *nx′* **by**(*simp add*:*valid-edge-def*)
with ‹∀ *V*∈*Uses prog* (*sourcenode a*). *id s V* = *id s′ V*›
show ∀ *V*∈*Defs prog* (*sourcenode a*).
id (*transfer* (*kind a*) *s*) *V* = *id* (*transfer* (*kind a*) *s′*) *V*
by −(*drule WCFG-edge-transfer-uses-only-Uses*,*simp*+)
next
fix *a s s′*
assume *pred*:*pred* (*kind a*) *s* **and** *valid*:*valid-edge prog a*
and *all*:∀ *V*∈*Uses prog* (*sourcenode a*). *id s V* = *id s′ V*
obtain *nx et nx′* **where** [*simp*]:*a* = (*nx*,*et*,*nx′*) **by**(*cases a*) *auto*

with ‹*valid-edge prog a*› **have** *prog* ⊢ *nx* −*et*→ *nx*′ **by**(*simp add*:*valid-edge-def*)
with ‹*pred* (*kind a*) *s*› ‹∀ *V*∈*Uses prog* (*sourcenode a*). *id s V* = *id s*′ *V*›
show *pred* (*kind a*) *s*′ **by** −(*drule WCFG-edge-Uses-pred-eq*,*simp*+)
next
fix *a a*′
assume *valid-edge prog a* **and** *valid-edge prog a*′
and *sourcenode a* = *sourcenode a*′ **and** *targetnode a* ≠ *targetnode a*′
thus ∃ *Q Q*′. *kind a* = $(Q)_{\surd}$ ∧ *kind a*′ = $(Q')_{\surd}$ ∧
(∀ *s*. (*Q s* ⟶ ¬ *Q*′ *s*) ∧ (*Q*′ *s* ⟶ ¬ *Q s*))
by(*fastforce intro*!: *WCFG-deterministic simp*:*valid-edge-def*)
qed

lemma *While-CFGExit-wf-aux*:*CFGExit-wf sourcenode targetnode kind*
(*valid-edge prog*) *Entry* (*Defs prog*) (*Uses prog*) *id Exit*
proof(*unfold-locales*)
show *Defs prog* (-*Exit*-) = {} ∧ *Uses prog* (-*Exit*-) = {}
by(*simp add*:*Defs.simps Uses.simps*)
qed

interpretation *While-CFGExit-wf*: *CFGExit-wf sourcenode targetnode kind*
valid-edge prog Entry Defs prog Uses prog id Exit
for *prog*
by(*rule While-CFGExit-wf-aux*)

end

4.6 Lemmas for the control dependences

theory *AdditionalLemmas* **imports** *WellFormed*
begin

4.6.1 Paths to (-*Exit*-) and from (-*Entry*-) exist

abbreviation *path* :: *cmd* ⇒ *w-node* ⇒ *w-edge list* ⇒ *w-node* ⇒ *bool*
(‹- ⊢ - −-→∗ -›)
where *prog* ⊢ *n* −*as*→∗ *n*′ ≡ *CFG.path sourcenode targetnode* (*valid-edge prog*)
n as n′

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

lemma *path-SeqFirst*:
prog ⊢ *n* −*as*→∗ (- *l* -) ⟹ *prog*;;c_2 ⊢ *n* −*as*→∗ (- *l* -)
proof(*induct n as* (- *l* -) *arbitrary*:*l rule*:*While-CFG.path.induct*)
case *empty-path*
from ‹*CFG.valid-node sourcenode targetnode* (*valid-edge prog*) (- *l* -)›

```
  show ?case
    apply -
    apply(rule While-CFG.empty-path)
    apply(auto simp:While-CFG.valid-node-def valid-edge-def)
    by(case-tac b,auto dest:WCFG-SeqFirst WCFG-SeqConnect)
next
  case (Cons-path n'' as a n)
  note IH = ‹prog;; c₂ ⊢ n'' −as→* (- l -)›
  from ‹prog ⊢ n'' −as→* (- l -)› have n'' ≠ (-Exit-)
    by fastforce
  with ‹valid-edge prog a› ‹sourcenode a = n› ‹targetnode a = n''›
  have prog;;c₂ ⊢ n −kind a→ n'' by(simp add:valid-edge-def WCFG-SeqFirst)
  with IH ‹prog;;c₂ ⊢ n −kind a→ n''› ‹sourcenode a = n› ‹targetnode a = n''›
show ?case
    by(fastforce intro:While-CFG.Cons-path simp:valid-edge-def)
qed


lemma path-SeqSecond:
  ⟦prog ⊢ n −as→* n'; n ≠ (-Entry-); as ≠ []⟧
  ⟹ c₁;;prog ⊢ n ⊕ #:c₁ −as ⊕s #:c₁→* n' ⊕ #:c₁
proof(induct rule:While-CFG.path.induct)
  case (Cons-path n'' as n' a n)
  note IH = ‹ ⟦n'' ≠ (-Entry-); as ≠ []⟧
    ⟹ c₁;;prog ⊢ n'' ⊕ #:c₁ −as ⊕s #:c₁→* n' ⊕ #:c₁›
  from ‹valid-edge prog a› ‹sourcenode a = n› ‹targetnode a = n''› ‹n ≠ (-Entry-)›
  have c₁;;prog ⊢ n ⊕ #:c₁ −kind a→ n'' ⊕ #:c₁
    by(simp add:valid-edge-def WCFG-SeqSecond)
  from ‹sourcenode a = n› ‹targetnode a = n''› ‹valid-edge prog a›
  have [(n,kind a,n'')] ⊕s #:c₁ = [a] ⊕s #:c₁
    by(cases a,simp add:label-incrs-def valid-edge-def)
  show ?case
  proof(cases as = [])
    case True
   with ‹prog ⊢ n'' −as→* n'› have [simp]:n'' = n' by(auto elim:While-CFG.cases)
    with ‹c₁;;prog ⊢ n ⊕ #:c₁ −kind a→ n'' ⊕ #:c₁›
    have c₁;;prog ⊢ n ⊕ #:c₁ −[(n,kind a,n')] ⊕s #:c₁→* n' ⊕ #:c₁
      by(fastforce intro:While-CFG.Cons-path While-CFG.empty-path
                simp:label-incrs-def While-CFG.valid-node-def valid-edge-def)
    with True ‹[(n,kind a,n'')] ⊕s #:c₁ = [a] ⊕s #:c₁› show ?thesis by simp
  next
    case False
    from ‹valid-edge prog a› ‹targetnode a = n''› have n'' ≠ (-Entry-)
      by(cases n'',auto simp:valid-edge-def)
    from IH[OF this False]
    have c₁;;prog ⊢ n'' ⊕ #:c₁ −as ⊕s #:c₁→* n' ⊕ #:c₁ .
    with ‹c₁;;prog ⊢ n ⊕ #:c₁ −kind a→ n'' ⊕ #:c₁› ‹sourcenode a = n›
      ‹targetnode a = n''› ‹[(n,kind a,n'')] ⊕s #:c₁ = [a] ⊕s #:c₁› show ?thesis
      apply(cases a)
```

apply(*simp add*:*label-incrs-def*)
by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-edge-def*)
qed
qed *simp*

lemma *path-CondTrue*:
prog ⊢ (- *l* -) −*as*→∗ *n′*
⟹ *if* (*b*) *prog else* c_2 ⊢ (- *l* -) ⊕ *1* −*as* ⊕*s 1*→∗ *n′* ⊕ *1*
proof(*induct* (- *l* -) *as n′ arbitrary*:*l rule*:*While-CFG.path.induct*)
case *empty-path*
from ‹*CFG.valid-node sourcenode targetnode* (*valid-edge prog*) (- *l* -)›
WCFG-CondTrue[*of b prog* c_2]
have *CFG.valid-node sourcenode targetnode* (*valid-edge* (*if* (*b*) *prog else* c_2))
((- *l* -) ⊕ *1*)
apply(*auto simp*:*While-CFG.valid-node-def valid-edge-def*)
apply(*rotate-tac 1*,*drule WCFG-CondThen*,*simp*,*fastforce*)
apply(*case-tac a*) **apply** *auto*
apply(*rotate-tac 1*,*drule WCFG-CondThen*,*simp*,*fastforce*)
by(*rotate-tac 1*,*drule WCFG-EntryD*,*auto*)
then show *?case*
by(*fastforce intro*:*While-CFG.empty-path simp*:*label-incrs-def*)
next
case (*Cons-path n′′ as n′ a*)
note *IH* = ‹⋀*l*. *n′′* = (- *l* -)
⟹ *if* (*b*) *prog else* c_2 ⊢ (- *l* -) ⊕ *1* −*as* ⊕*s 1*→∗ *n′* ⊕ *1*›
from ‹*valid-edge prog a*› ‹*sourcenode a* = (- *l* -)› ‹*targetnode a* = *n′′*›
have *if* (*b*) *prog else* c_2 ⊢ (- *l* -) ⊕ *1* −*kind a*→ *n′′* ⊕ *1*
by −(*rule WCFG-CondThen*,*simp-all add*:*valid-edge-def*)
from ‹*sourcenode a* = (- *l* -)› ‹*targetnode a* = *n′′*› ‹*valid-edge prog a*›
have [((- *l* -),*kind a*,*n′′*)] ⊕*s 1* = [*a*] ⊕*s 1*
by(*cases a*,*simp add*:*label-incrs-def valid-edge-def*)
show *?case*
proof(*cases n′′*)
case (*Node l′*)
from *IH*[*OF this*] **have** *if* (*b*) *prog else* c_2 ⊢ (- *l′* -) ⊕ *1* −*as* ⊕*s 1*→∗ *n′* ⊕ *1* **.**
with ‹*if* (*b*) *prog else* c_2 ⊢ (- *l* -) ⊕ *1* −*kind a*→ *n′′* ⊕ *1*› *Node*
have *if* (*b*) *prog else* c_2 ⊢ (- *l* -) ⊕ *1* −((- *l* -) ⊕ *1*,*kind a*,*n′′* ⊕ *1*)#(*as* ⊕*s 1*)→∗ *n′* ⊕ *1*
by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-edge-def valid-node-def*)
with ‹[((- *l* -),*kind a*,*n′′*)] ⊕*s 1* = [*a*] ⊕*s 1*›
have *if* (*b*) *prog else* c_2 ⊢ (- *l* -) ⊕ *1* −*a*#*as* ⊕*s 1*→∗ *n′* ⊕ *1*
by(*simp add*:*label-incrs-def*)
thus *?thesis* **by** *simp*
next
case *Entry*
with ‹*valid-edge prog a*› ‹*targetnode a* = *n′′*› **have** *False* **by** *fastforce*
thus *?thesis* **by** *simp*
next

```
    case Exit
    with ‹prog ⊢ n'' −as→* n'› have n' = (-Exit-) and as = []
      by(auto dest:While-CFGExit.path-Exit-source)
    from ‹if (b) prog else c₂ ⊢ (- l -) ⊕ 1 −kind a→ n'' ⊕ 1›
    have if (b) prog else c₂ ⊢ (- l -) ⊕ 1 −[((- l -) ⊕ 1,kind a,n'' ⊕ 1)]→* n'' ⊕ 1
      by(fastforce intro:While-CFG.Cons-path While-CFG.empty-path
                simp:While-CFG.valid-node-def valid-edge-def)
    with Exit ‹[((- l -),kind a,n'')] ⊕s 1 = [a] ⊕s 1›  ‹n' = (-Exit-)› ‹as = []›
    show ?thesis by(fastforce simp:label-incrs-def)
  qed
qed


lemma path-CondFalse:
  prog ⊢ (- l -) −as→* n'
  ⟹ if (b) c₁ else prog ⊢ (- l -) ⊕ (#:c₁ + 1) −as ⊕s (#:c₁ + 1)→* n' ⊕ (#:c₁
+ 1)
proof(induct (- l -) as n' arbitrary:l rule:While-CFG.path.induct)
  case empty-path
  from ‹CFG.valid-node sourcenode targetnode (valid-edge prog) (- l -)›
    WCFG-CondFalse[of b c₁ prog]
  have CFG.valid-node sourcenode targetnode (valid-edge (if (b) c₁ else prog))
    ((- l -) ⊕ #:c₁ + 1)
    apply(auto simp:While-CFG.valid-node-def valid-edge-def)
    apply(rotate-tac 1,drule WCFG-CondElse,simp,fastforce)
    apply(case-tac a) apply auto
     apply(rotate-tac 1,drule WCFG-CondElse,simp,fastforce)
    by(rotate-tac 1,drule WCFG-EntryD,auto)
  thus ?case by(fastforce intro:While-CFG.empty-path simp:label-incrs-def)
next
  case (Cons-path n'' as n' a)
  note IH = ‹⋀l. n'' = (- l -) ⟹ if (b) c₁ else prog ⊢ (- l -) ⊕ (#:c₁ + 1)
                                          −as ⊕s (#:c₁ + 1)→* n' ⊕ (#:c₁ + 1)›
  from ‹valid-edge prog a› ‹sourcenode a = (- l -)›  ‹targetnode a = n''›
  have if (b) c₁ else prog ⊢ (- l -) ⊕ (#:c₁ + 1) −kind a→ n'' ⊕ (#:c₁ + 1)
    by −(rule WCFG-CondElse,simp-all add:valid-edge-def)
  from ‹sourcenode a = (- l -)› ‹targetnode a = n''› ‹valid-edge prog a›
  have [((- l -),kind a,n'')] ⊕s (#:c₁ + 1) = [a] ⊕s (#:c₁ + 1)
    by(cases a,simp add:label-incrs-def valid-edge-def)
  show ?case
  proof(cases n'')
    case (Node l')
    from IH[OF this] have if (b) c₁ else prog ⊢ (- l' -) ⊕ (#:c₁ + 1)
                                        −as ⊕s (#:c₁ + 1)→* n' ⊕ (#:c₁ + 1) .
    with ‹if (b) c₁ else prog ⊢ (- l -) ⊕ (#:c₁ + 1) −kind a→ n'' ⊕ (#:c₁ + 1)›
Node
    have if (b) c₁ else prog ⊢ (- l -) ⊕ (#:c₁ + 1)
      −((- l -) ⊕ (#:c₁ + 1),kind a,n'' ⊕ (#:c₁ + 1))#(as ⊕s (#:c₁ + 1))→*
      n' ⊕ (#:c₁ + 1)
```

by(*fastforce intro*: *While-CFG.Cons-path simp*:*valid-edge-def valid-node-def*)
with ‹[((- *l* -),*kind a*,n'')] ⊕*s* (#:c_1 + *1*) = [*a*] ⊕*s* (#:c_1 + *1*)› *Node*
have *if* (*b*) c_1 *else prog* ⊢ (- *l* -) ⊕ (#:c_1 + *1*) −*a*#*as* ⊕*s* (#:c_1 + *1*)→∗
n' ⊕ (#:c_1 + *1*)
by(*simp add*:*label-incrs-def*)
thus *?thesis* **by** *simp*
next
case *Entry*
with ‹*valid-edge prog a*› ‹*targetnode a* = n''› **have** *False* **by** *fastforce*
thus *?thesis* **by** *simp*
next
case *Exit*
with ‹*prog* ⊢ n'' −*as*→∗ n'› **have** n' = (-*Exit*-) **and** *as* = []
by(*auto dest*:*While-CFGExit.path-Exit-source*)
from ‹*if* (*b*) c_1 *else prog* ⊢ (- *l* -) ⊕ (#:c_1 + *1*) −*kind a*→ n'' ⊕ (#:c_1 + *1*)›
have *if* (*b*) c_1 *else prog* ⊢ (- *l* -) ⊕ (#:c_1 + *1*)
−[((- *l* -) ⊕ (#:c_1 + *1*),*kind a*,n'' ⊕ (#:c_1 + *1*))]→∗ n'' ⊕ (#:c_1 + *1*)
by(*fastforce intro*:*While-CFG.Cons-path While-CFG.empty-path*
simp:*While-CFG.valid-node-def valid-edge-def*)
with *Exit* ‹[((- *l* -),*kind a*,n'')] ⊕*s* (#:c_1 + *1*) = [*a*] ⊕*s* (#:c_1 + *1*)› ‹n' =
(-*Exit*-)›
‹*as* = []›
show *?thesis* **by**(*fastforce simp*:*label-incrs-def*)
qed
qed

lemma *path-While*:
prog ⊢ (- *l* -) −*as*→∗ (- l' -)
⟹ *while* (*b*) *prog* ⊢ (- *l* -) ⊕ *2* −*as* ⊕*s 2*→∗ (- l' -) ⊕ *2*
proof(*induct* (- *l* -) *as* (- l' -) *arbitrary*:*l* l' *rule*:*While-CFG.path.induct*)
case *empty-path*
from ‹*CFG.valid-node sourcenode targetnode* (*valid-edge prog*) (- *l* -)›
WCFG-WhileTrue[*of b prog*]
have *CFG.valid-node sourcenode targetnode* (*valid-edge* (*while* (*b*) *prog*)) ((- *l* -)
⊕ *2*)
apply(*auto simp*:*While-CFG.valid-node-def valid-edge-def*)
apply(*case-tac ba*) **apply** *auto*
apply(*rotate-tac 1*,*drule WCFG-WhileBody*,*auto*)
apply(*rotate-tac 1*,*drule WCFG-WhileBodyExit*,*auto*)
apply(*case-tac a*) **apply** *auto*
apply(*rotate-tac 1*,*drule WCFG-WhileBody*,*auto*)
by(*rotate-tac 1*,*drule WCFG-EntryD*,*auto*)
thus *?case* **by**(*fastforce intro*:*While-CFG.empty-path simp*:*label-incrs-def*)
next
case (*Cons-path* n'' *as a*)
note *IH* = ‹⋀*l*. n'' = (- *l* -)
⟹ *while* (*b*) *prog* ⊢ (- *l* -) ⊕ *2* −*as* ⊕*s 2*→∗ (- l' -) ⊕ *2*›
from ‹*sourcenode a* = (- *l* -)› ‹*targetnode a* = n''› ‹*valid-edge prog a*›
have [((- *l* -),*kind a*,n'')] ⊕*s 2* = [*a*] ⊕*s 2*

by(*cases a*,*simp add*:*label-incrs-def valid-edge-def*)
show *?case*
proof(*cases n''*)
case (*Node l''*)
with ‹*valid-edge prog a*› ‹*sourcenode a* = (- *l* -)› ‹*targetnode a* = *n''*›
have *while* (*b*) *prog* ⊢ (- *l* -) ⊕ *2* −*kind a*→ *n''* ⊕ *2*
by −(*rule WCFG-WhileBody*,*simp-all add*:*valid-edge-def*)
from *IH*[*OF Node*]
have *while* (*b*) *prog* ⊢ (- *l''* -) ⊕ *2* −*as* ⊕*s 2*→∗ (- *l'* -) ⊕ *2* .
with ‹*while* (*b*) *prog* ⊢ (- *l* -) ⊕ *2* −*kind a*→ *n''* ⊕ *2*› *Node*
have *while* (*b*) *prog* ⊢ (- *l* -) ⊕ *2* −((- *l* -) ⊕ *2*,*kind a*,*n''* ⊕ *2*)#(*as* ⊕*s 2*)→∗
(- *l'* -) ⊕ *2*
by(*fastforce intro*: *While-CFG.Cons-path simp*:*valid-edge-def*)
with ‹[((- *l* -),*kind a*,*n''*)] ⊕*s 2* = [*a*] ⊕*s 2*› **show** *?thesis* **by**(*simp add*:*label-incrs-def*)
next
case *Entry*
with ‹*valid-edge prog a*› ‹*targetnode a* = *n''*› **have** *False* **by** *fastforce*
thus *?thesis* **by** *simp*
next
case *Exit*
with ‹*prog* ⊢ *n''* −*as*→∗ (- *l'* -)› **have** (- *l'* -) = (-*Exit*-) **and** *as* = []
by(*auto dest*: *While-CFGExit.path-Exit-source*)
then have *False* **by** *simp*
thus *?thesis* **by** *simp*
qed
qed

lemma *inner-node-Entry-Exit-path*:
l < #:*prog* ⟹ (∃ *as*. *prog* ⊢ (- *l* -) −*as*→∗ (-*Exit*-)) ∧
(∃ *as*. *prog* ⊢ (-*Entry*-) −*as*→∗ (- *l* -))
proof(*induct prog arbitrary*:*l*)
case *Skip*
from ‹*l* < #:*Skip*› **have** [*simp*]:*l* = *0* **by** *simp*
hence *Skip* ⊢ (- *l* -) −⇑*id*→ (-*Exit*-) **by**(*simp add*: *WCFG-Skip*)
hence *Skip* ⊢ (- *l* -) −[((- *l* -),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)
by (*fastforce intro*: *While-CFG.path.intros simp*: *valid-edge-def*)
have *Skip* ⊢ (-*Entry*-) −(λ*s*. *True*)$_{\surd}$→ (- *l* -) **by**(*simp add*: *WCFG-Entry*)
hence *Skip* ⊢ (-*Entry*-) −[((-*Entry*-),(λ*s*. *True*)$_{\surd}$,(- *l* -))]→∗ (- *l* -)
by(*fastforce intro*: *While-CFG.path.intros simp*:*valid-edge-def While-CFG.valid-node-def*)
with ‹*Skip* ⊢ (- *l* -) −[((- *l* -),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)› **show** *?case* **by** *fastforce*
next
case (*LAss V e*)
from ‹*l* < #:*V*:=*e*› **have** *l* = *0* ∨ *l* = *1* **by** *auto*
thus *?case*
proof
assume [*simp*]:*l* = *0*
hence *V*:=*e* ⊢ (-*Entry*-) −(λ*s*. *True*)$_{\surd}$→ (- *l* -) **by**(*simp add*: *WCFG-Entry*)

hence *V*:=*e* ⊢ (-*Entry*-) −[((-*Entry*-),(λ*s*. *True*)$_{\surd}$,(- *l* -))]→∗ (- *l* -)
by(*fastforce intro*:*While-CFG.path.intros simp*:*valid-edge-def While-CFG.valid-node-def*)
have *V*:=*e* ⊢ (-*1*-) −⇑*id*→ (-*Exit*-) **by**(*rule WCFG-LAssSkip*)
hence *V*:=*e* ⊢ (-*1*-) −[((-*1*-),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)
by(*fastforce intro*:*While-CFG.path.intros simp*:*valid-edge-def*)
with *WCFG-LAss* **have** *V*:=*e* ⊢ (- *l* -) −
[((- *l* -),⇑(λ*s*. *s*(*V*:=(*interpret e s*))),(-*1*-)),((-*1*-),⇑*id*,(-*Exit*-))]→∗
(-*Exit*-)
by(*fastforce intro*:*While-CFG.path.intros simp*:*valid-edge-def*)
with ‹*V*:=*e* ⊢ (-*Entry*-) −[((-*Entry*-),(λ*s*. *True*)$_{\surd}$,(- *l* -))]→∗ (- *l* -)›
show *?case* **by** *fastforce*
next
assume [*simp*]:*l* = *1*
hence *V*:=*e* ⊢ (- *l* -) −⇑*id*→ (-*Exit*-) **by**(*simp add*:*WCFG-LAssSkip*)
hence *V*:=*e* ⊢ (- *l* -) −[((- *l* -),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)
by(*fastforce intro*:*While-CFG.path.intros simp*:*valid-edge-def*)
have *V*:=*e* ⊢ (-*0*-) −⇑(λ*s*. *s*(*V*:=(*interpret e s*)))→ (- *l* -)
by(*simp add*:*WCFG-LAss*)
hence *V*:=*e* ⊢ (-*0*-) −[((-*0*-),⇑(λ*s*. *s*(*V*:=(*interpret e s*))),(- *l* -))]→∗ (- *l* -)
by(*fastforce intro*:*While-CFG.path.intros simp*:*valid-edge-def While-CFG.valid-node-def*)
with *WCFG-Entry*[*of V*:=*e*] **have** *V*:=*e* ⊢ (-*Entry*-) −[((-*Entry*-),(λ*s*. *True*)$_{\surd}$,(-*0*-))
,((-*0*-),⇑(λ*s*. *s*(*V*:=(*interpret e s*))),(- *l* -))]→∗ (- *l* -)
by(*fastforce intro*:*While-CFG.path.intros simp*:*valid-edge-def*)
with ‹*V*:=*e* ⊢ (- *l* -) −[((- *l* -),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)› **show** *?case* **by** *fastforce*
qed
next
case (*Seq prog1 prog2*)
note *IH1* = ‹⋀*l*. *l* < #:*prog1* ⟹
(∃ *as*. *prog1* ⊢ (- *l* -) −*as*→∗ (-*Exit*-)) ∧ (∃ *as*. *prog1* ⊢ (-*Entry*-) −*as*→∗ (- *l* -))›
note *IH2* = ‹⋀*l*. *l* < #:*prog2* ⟹
(∃ *as*. *prog2* ⊢ (- *l* -) −*as*→∗ (-*Exit*-)) ∧ (∃ *as*. *prog2* ⊢ (-*Entry*-) −*as*→∗ (- *l* -))›
show *?case*
proof(*cases l* < #:*prog1*)
case *True*
from *IH1*[*OF True*] **obtain** *as as′* **where** *prog1* ⊢ (- *l* -) −*as*→∗ (-*Exit*-)
and *prog1* ⊢ (-*Entry*-) −*as′*→∗ (- *l* -) **by** *blast*
from ‹*prog1* ⊢ (-*Entry*-) −*as′*→∗ (- *l* -)›
have *prog1*;;*prog2* ⊢ (-*Entry*-) −*as′*→∗ (- *l* -)
by(*fastforce intro*:*path-SeqFirst*)
from ‹*prog1* ⊢ (- *l* -) −*as*→∗ (-*Exit*-)›
obtain *asx ax* **where** *prog1* ⊢ (- *l* -) −*asx*@[*ax*]→∗ (-*Exit*-)
by(*induct rule*:*rev-induct*,*auto elim*:*While-CFG.path.cases*)
hence *prog1* ⊢ (- *l* -) −*asx*→∗ *sourcenode ax*
and *valid-edge prog1 ax* **and** (-*Exit*-) = *targetnode ax*
by(*auto intro*:*While-CFG.path-split-snoc*)
from ‹*prog1* ⊢ (- *l* -) −*asx*→∗ *sourcenode ax*› ‹*valid-edge prog1 ax*›
obtain *lx* **where** [*simp*]:*sourcenode ax* = (- *lx* -)
by(*cases sourcenode ax*) *auto*
with ‹*prog1* ⊢ (- *l* -) −*asx*→∗ *sourcenode ax*›

have *prog1*;;*prog2* ⊢ (- *l* -) −*asx*→∗ *sourcenode ax*
  by(*fastforce intro*:*path-SeqFirst*)
from ‹*valid-edge prog1 ax*› ‹(-*Exit*-) = *targetnode ax*›
have *prog1*;;*prog2* ⊢ *sourcenode ax* −*kind ax*→ (-*0*-) ⊕ #:*prog1*
  by(*fastforce intro*:*WCFG-SeqConnect simp*:*valid-edge-def*)
hence *prog1*;;*prog2* ⊢ *sourcenode ax* −[(*sourcenode ax*,*kind ax*,(-*0*-) ⊕ #:*prog1*)]→∗
  (-*0*-) ⊕ #:*prog1*
  by(*fastforce intro*:*While-CFG.Cons-path While-CFG.empty-path*
    simp:*While-CFG.valid-node-def valid-edge-def*)
with ‹*prog1*;;*prog2* ⊢ (- *l* -) −*asx*→∗ *sourcenode ax*›
have *prog1*;;*prog2* ⊢ (- *l* -) −*asx*@[(*sourcenode ax*,*kind ax*,(-*0*-) ⊕ #:*prog1*)]→∗
  (-*0*-) ⊕ #:*prog1*
  by(*fastforce intro*:*While-CFG.path-Append*)
from *IH2*[*of 0*] **obtain** *as''* **where** *prog2* ⊢ (- *0* -) −*as''*→∗ (-*Exit*-) **by** *blast*
hence *prog1*;;*prog2* ⊢ (-*0*-) ⊕ #:*prog1* −*as''* ⊕*s* #:*prog1*→∗ (-*Exit*-) ⊕ #:*prog1*
  by(*fastforce intro*!:*path-SeqSecond elim*:*While-CFG.path.cases*)
hence *prog1*;;*prog2* ⊢ (-*0*-) ⊕ #:*prog1* −*as''* ⊕*s* #:*prog1*→∗ (-*Exit*-)
  by *simp*
with ‹*prog1*;;*prog2* ⊢ (- *l* -) −*asx*@[(*sourcenode ax*,*kind ax*,(-*0*-) ⊕ #:*prog1*)]→∗
  (-*0*-) ⊕ #:*prog1*›
have *prog1*;;*prog2* ⊢ (- *l* -) −(*asx*@[(*sourcenode ax*,*kind ax*,(-*0*-) ⊕ #:*prog1*)])@
  (*as''* ⊕*s* #:*prog1*)→∗ (-*Exit*-)
  by(*fastforce intro*:*While-CFG.path-Append*)
with ‹*prog1*;;*prog2* ⊢ (-*Entry*-) −*as'*→∗ (- *l* -)› **show** *?thesis* **by** *blast*
next
case *False*
hence #:*prog1* ≤ *l* **by** *simp*
then obtain *l'* **where** [*simp*]:*l* = *l'* + #:*prog1* **and** *l'* = *l* − #:*prog1* **by** *simp*
from ‹*l* < #:*prog1*;; *prog2*› **have** *l'* < #:*prog2* **by** *simp*
from *IH2*[*OF this*] **obtain** *as as'* **where** *prog2* ⊢ (- *l'* -) −*as*→∗ (-*Exit*-)
  and *prog2* ⊢ (-*Entry*-) −*as'*→∗ (- *l'* -) **by** *blast*
from ‹*prog2* ⊢ (- *l'* -) −*as*→∗ (-*Exit*-)›
have *prog1*;;*prog2* ⊢ (- *l'* -) ⊕ #:*prog1* −*as* ⊕*s* #:*prog1*→∗ (-*Exit*-) ⊕ #:*prog1*
  by(*fastforce intro*!:*path-SeqSecond elim*:*While-CFG.path.cases*)
hence *prog1*;;*prog2* ⊢ (- *l* -) −*as* ⊕*s* #:*prog1*→∗ (-*Exit*-)
  by *simp*
from *IH1*[*of 0*] **obtain** *as''* **where** *prog1* ⊢ (-*0*-) −*as''*→∗ (-*Exit*-) **by** *blast*
then obtain *ax asx* **where** *prog1* ⊢ (-*0*-) −*asx*@[*ax*]→∗ (-*Exit*-)
  by(*induct rule*:*rev-induct*,*auto elim*:*While-CFG.path.cases*)
hence *prog1* ⊢ (-*0*-) −*asx*→∗ *sourcenode ax* **and** *valid-edge prog1 ax*
  and (-*Exit*-) = *targetnode ax* **by**(*auto intro*:*While-CFG.path-split-snoc*)
from *WCFG-Entry* ‹*prog1* ⊢ (-*0*-) −*asx*→∗ *sourcenode ax*›
have *prog1* ⊢ (-*Entry*-) −((-*Entry*-),(λ*s. True*)$_{\surd}$,(-*0*-))#*asx*→∗ *sourcenode ax*
  by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-edge-def valid-node-def*)
from ‹*prog1* ⊢ (-*0*-) −*asx*→∗ *sourcenode ax*› ‹*valid-edge prog1 ax*›
obtain *lx* **where** [*simp*]:*sourcenode ax* = (- *lx* -)
  by(*cases sourcenode ax*) *auto*
with ‹*prog1* ⊢ (-*Entry*-) −((-*Entry*-),(λ*s. True*)$_{\surd}$,(-*0*-))#*asx*→∗ *sourcenode ax*›
have *prog1*;;*prog2* ⊢ (-*Entry*-) −((-*Entry*-),(λ*s. True*)$_{\surd}$,(-*0*-))#*asx*→∗

sourcenode ax
by(*fastforce intro*:*path-SeqFirst*)
from ‹*prog2* ⊢ (-*Entry*-) −*as*′→∗ (- *l*′ -)› **obtain** *ax*′ *asx*′
where *prog2* ⊢ (-*Entry*-) −*ax*′#*asx*′→∗ (- *l*′ -)
by(*cases as*′,*auto elim*:*While-CFG.path.cases*)
hence (-*Entry*-) = *sourcenode ax*′ **and** *valid-edge prog2 ax*′
and *prog2* ⊢ *targetnode ax*′ −*asx*′→∗ (- *l*′ -)
by(*auto intro*:*While-CFG.path-split-Cons*)
hence *targetnode ax*′ = (-*0*-) **by**(*fastforce dest*:*WCFG-EntryD simp*:*valid-edge-def*)
from ‹*valid-edge prog1 ax*› ‹(-*Exit*-) = *targetnode ax*›
have *prog1*;;*prog2* ⊢ *sourcenode ax* −*kind ax*→ (-*0*-) ⊕ #:*prog1*
by(*fastforce intro*:*WCFG-SeqConnect simp*:*valid-edge-def*)
have ∃ *as*. *prog1*;;*prog2* ⊢ *sourcenode ax* −*as*→∗ (- *l* -)
proof(*cases asx*′ = [])
case *True*
with ‹*prog2* ⊢ *targetnode ax*′ −*asx*′→∗ (- *l*′ -)› ‹*targetnode ax*′ = (-*0*-)›
have *l*′ = *0* **by**(*auto elim*:*While-CFG.path.cases*)
with ‹*prog1*;;*prog2* ⊢ *sourcenode ax* −*kind ax*→ (-*0*-) ⊕ #:*prog1*›
have *prog1*;;*prog2* ⊢ *sourcenode ax* −[(*sourcenode ax*,*kind ax*,(- *l* -))]→∗
(- *l* -)
by(*auto intro*!:*While-CFG.path.intros*
simp:*While-CFG.valid-node-def valid-edge-def*,*blast*)
thus *?thesis* **by** *blast*
next
case *False*
with ‹*prog2* ⊢ *targetnode ax*′ −*asx*′→∗ (- *l*′ -)› ‹*targetnode ax*′ = (-*0*-)›
have *prog1*;;*prog2* ⊢ (-*0*-) ⊕ #:*prog1* −*asx*′ ⊕*s* #:*prog1*→∗ (- *l*′ -) ⊕ #:*prog1*
by(*fastforce intro*:*path-SeqSecond*)
hence *prog1*;;*prog2* ⊢ (-*0*-) ⊕ #:*prog1* −*asx*′ ⊕*s* #:*prog1*→∗ (- *l* -) **by** *simp*
with ‹*prog1*;;*prog2* ⊢ *sourcenode ax* −*kind ax*→ (-*0*-) ⊕ #:*prog1*›
have *prog1*;;*prog2* ⊢ *sourcenode ax* −(*sourcenode ax*,*kind ax*,(-*0*-) ⊕ #:*prog1*)#
(*asx*′ ⊕*s* #:*prog1*)→∗ (- *l* -)
by(*fastforce intro*: *While-CFG.Cons-path simp*:*valid-node-def valid-edge-def*)
thus *?thesis* **by** *blast*
qed
then obtain *asx*′′ **where** *prog1*;;*prog2* ⊢ *sourcenode ax* −*asx*′′→∗ (- *l* -) **by**
blast
with ‹*prog1*;;*prog2* ⊢ (-*Entry*-) −((-*Entry*-),(λ*s*. *True*)$_\surd$,(-*0*-))#*asx*→∗
sourcenode ax›
have *prog1*;;*prog2* ⊢ (-*Entry*-) −(((-*Entry*-),(λ*s*. *True*)$_\surd$,(-*0*-))#*asx*)@*asx*′′→∗
(- *l* -)
by(*rule While-CFG.path-Append*)
with ‹*prog1*;;*prog2* ⊢ (- *l* -) −*as* ⊕*s* #:*prog1*→∗ (-*Exit*-)›
show *?thesis* **by** *blast*
qed
next
case (*Cond b prog1 prog2*)
note *IH1* = ‹⋀*l*. *l* < #:*prog1* ⟹
(∃ *as*. *prog1* ⊢ (- *l* -) −*as*→∗ (-*Exit*-)) ∧ (∃ *as*. *prog1* ⊢ (-*Entry*-) −*as*→∗ (- *l* -))›

note *IH2* = ‹⋀*l*. *l* < #:*prog2* ⟹
(∃ *as*. *prog2* ⊢ (- *l* -) −*as*→∗ (-*Exit*-)) ∧ (∃ *as*. *prog2* ⊢ (-*Entry*-) −*as*→∗ (- *l* -))›
show *?case*
proof(*cases l* = *0*)
  case *True*
  from *IH1*[*of 0*] **obtain** *as* **where** *prog1* ⊢ (-*0*-) −*as*→∗ (-*Exit*-) **by** *blast*
  hence *if* (*b*) *prog1 else prog2* ⊢ (-*0*-) ⊕ *1* −*as* ⊕*s 1*→∗ (-*Exit*-) ⊕ *1*
    by(*fastforce intro*:*path-CondTrue*)
  with *WCFG-CondTrue*[*of b prog1 prog2*] **have** *if* (*b*) *prog1 else prog2* ⊢
    (-*0*-) −((-*0*-),(λ*s*. *interpret b s* = *Some true*)$_{\surd}$,(-*0*-) ⊕ *1*)#(*as* ⊕*s 1*)→∗
    (-*Exit*-) ⊕ *1*
    by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-edge-def valid-node-def*)
  with *True* **have** *if* (*b*) *prog1 else prog2* ⊢
    (- *l* -) −((-*0*-),(λ*s*. *interpret b s* = *Some true*)$_{\surd}$,(-*0*-) ⊕ *1*)#(*as* ⊕*s 1*)→∗
    (-*Exit*-) **by** *simp*
  moreover
  from *WCFG-Entry*[*of if* (*b*) *prog1 else prog2*] *True*
  have *if* (*b*) *prog1 else prog2* ⊢ (-*Entry*-) −[((-*Entry*-),(λ*s*. *True*)$_{\surd}$,(-*0*-))]→∗
                                  (- *l* -)
    by(*fastforce intro*:*While-CFG.Cons-path While-CFG.empty-path*
            simp:*While-CFG.valid-node-def valid-edge-def*)
  ultimately show *?thesis* **by** *blast*
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'* < #:*prog1*)
    case *True*
    from *IH1*[*OF this*] **obtain** *as as'* **where** *prog1* ⊢ (- *l'* -) −*as*→∗ (-*Exit*-)
      and *prog1* ⊢ (-*Entry*-) −*as'*→∗ (- *l'* -) **by** *blast*
    from ‹*prog1* ⊢ (- *l'* -) −*as*→∗ (-*Exit*-)›
    have *if* (*b*) *prog1 else prog2* ⊢ (- *l'* -) ⊕ *1* −*as* ⊕*s 1*→∗ (-*Exit*-) ⊕ *1*
      by(*fastforce intro*:*path-CondTrue*)
    hence *if* (*b*) *prog1 else prog2* ⊢ (- *l* -) −*as* ⊕*s 1*→∗ (-*Exit*-)
      by *simp*
    from ‹*prog1* ⊢ (-*Entry*-) −*as'*→∗ (- *l'* -)› **obtain** *ax asx*
      where *prog1* ⊢ (-*Entry*-) −*ax*#*asx*→∗ (- *l'* -)
      by(*cases as'*,*auto elim*:*While-CFG.cases*)
    hence (-*Entry*-) = *sourcenode ax* **and** *valid-edge prog1 ax*
      and *prog1* ⊢ *targetnode ax* −*asx*→∗ (- *l'* -)
      by(*auto intro*:*While-CFG.path-split-Cons*)
    hence *targetnode ax* = (-*0*-) **by**(*fastforce dest*:*WCFG-EntryD simp*:*valid-edge-def*)
    with ‹*prog1* ⊢ *targetnode ax* −*asx*→∗ (- *l'* -)›
    have *if* (*b*) *prog1 else prog2* ⊢ (-*0*-) ⊕ *1* −*asx* ⊕*s 1*→∗ (- *l'* -) ⊕ *1*
      by(*fastforce intro*:*path-CondTrue*)
    with *WCFG-CondTrue*[*of b prog1 prog2*]
    have *if* (*b*) *prog1 else prog2* ⊢ (-*0*-)
      −((-*0*-),(λ*s*. *interpret b s* = *Some true*)$_{\surd}$,(-*0*-) ⊕ *1*)#(*asx* ⊕*s 1*)→∗

(- *l'* -) ⊕ *1*
by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-edge-def*)
with *WCFG-Entry*[*of if* (*b*) *prog1 else prog2*]
have *if* (*b*) *prog1 else prog2* ⊢ (*-Entry-*) −((*-Entry-*),(λ*s. True*)$_\surd$,(*-0-*))#
((*-0-*),(λ*s. interpret b s* = *Some true*)$_\surd$,(*-0-*) ⊕ *1*)#(*asx* ⊕*s 1*)→∗
(- *l'* -) ⊕ *1*
by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-edge-def*)
with ‹*if* (*b*) *prog1 else prog2* ⊢ (- *l* -) −*as* ⊕*s 1*→∗ (*-Exit-*)›
show *?thesis* **by** *simp blast*
next
case *False*
hence #:*prog1* ≤ *l'* **by** *simp*
then obtain *l''* **where** [*simp*]:*l'* = *l''* + #:*prog1* **and** *l''* = *l'* − #:*prog1*
by *simp*
from ‹*l* < #:(*if* (*b*) *prog1 else prog2*)›
have *l''* < #:*prog2* **by** *simp*
from *IH2*[*OF this*] **obtain** *as as'* **where** *prog2* ⊢ (- *l''* -) −*as*→∗ (*-Exit-*)
and *prog2* ⊢ (*-Entry-*) −*as'*→∗ (- *l''* -) **by** *blast*
from ‹*prog2* ⊢ (- *l''* -) −*as*→∗ (*-Exit-*)›
have *if* (*b*) *prog1 else prog2* ⊢ (- *l''* -) ⊕ (#:*prog1* + *1*)
−*as* ⊕*s* (#:*prog1* + *1*)→∗ (*-Exit-*) ⊕ (#:*prog1* + *1*)
by(*fastforce intro*:*path-CondFalse*)
hence *if* (*b*) *prog1 else prog2* ⊢ (- *l* -) −*as* ⊕*s* (#:*prog1* + *1*)→∗ (*-Exit-*)
by(*simp add*:*add.assoc*)
from ‹*prog2* ⊢ (*-Entry-*) −*as'*→∗ (- *l''* -)› **obtain** *ax asx*
where *prog2* ⊢ (*-Entry-*) −*ax*#*asx*→∗ (- *l''* -)
by(*cases as'*,*auto elim*:*While-CFG.cases*)
hence (*-Entry-*) = *sourcenode ax* **and** *valid-edge prog2 ax*
and *prog2* ⊢ *targetnode ax* −*asx*→∗ (- *l''* -)
by(*auto intro*:*While-CFG.path-split-Cons*)
hence *targetnode ax* = (*-0-*) **by**(*fastforce dest*:*WCFG-EntryD simp*:*valid-edge-def*)
with ‹*prog2* ⊢ *targetnode ax* −*asx*→∗ (- *l''* -)›
have *if* (*b*) *prog1 else prog2* ⊢ (*-0-*) ⊕ (#:*prog1* + *1*) −*asx* ⊕*s* (#:*prog1* +
1)→∗
(- *l''* -) ⊕ (#:*prog1* + *1*)
by(*fastforce intro*:*path-CondFalse*)
with *WCFG-CondFalse*[*of b prog1 prog2*]
have *if* (*b*) *prog1 else prog2* ⊢ (*-0-*)
−((*-0-*),(λ*s. interpret b s* = *Some false*)$_\surd$,(*-0-*) ⊕ (#:*prog1* + *1*))#
(*asx* ⊕*s* (#:*prog1* + *1*))→∗ (- *l''* -) ⊕ (#:*prog1* + *1*)
by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-edge-def*)
with *WCFG-Entry*[*of if* (*b*) *prog1 else prog2*]
have *if* (*b*) *prog1 else prog2* ⊢ (*-Entry-*) −((*-Entry-*),(λ*s. True*)$_\surd$,(*-0-*))#
((*-0-*),(λ*s. interpret b s* = *Some false*)$_\surd$,(*-0-*) ⊕ (#:*prog1* + *1*))#
(*asx* ⊕*s* (#:*prog1* + *1*))→∗ (- *l''* -) ⊕ (#:*prog1* + *1*)
by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-edge-def*)
with
‹*if* (*b*) *prog1 else prog2* ⊢ (- *l* -) −*as* ⊕*s* (#:*prog1* + *1*)→∗ (*-Exit-*)›
show *?thesis* **by**(*simp add*:*add.assoc*,*blast*)

qed
qed
next
case (*While b prog′*)
note *IH* = ‹⋀*l*. *l* < #:*prog′* ⟹
(∃ *as*. *prog′* ⊢ (- *l* -) −*as*→∗ (-*Exit*-)) ∧ (∃ *as*. *prog′* ⊢ (-*Entry*-) −*as*→∗ (- *l* -))›
show *?case*
proof(*cases l* < *1*)
case *True*
from *WCFG-Entry*[*of while* (*b*) *prog′*]
have *while* (*b*) *prog′* ⊢ (-*Entry*-) −[((-*Entry*-),(λ*s*. *True*)$_{\surd}$,(-*0*-))]→∗ (-*0*-)
by(*fastforce intro*: *While-CFG.Cons-path While-CFG.empty-path*
simp: *While-CFG.valid-node-def valid-edge-def*)
from *WCFG-WhileFalseSkip*[*of b prog′*]
have *while* (*b*) *prog′* ⊢ (-*1*-) −[((-*1*-),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)
by(*fastforce intro*: *While-CFG.Cons-path While-CFG.empty-path*
simp: *valid-node-def valid-edge-def*)
with *WCFG-WhileFalse*[*of b prog′*]
have *while* (*b*) *prog′* ⊢ (-*0*-) −((-*0*-),(λ*s*. *interpret b s* = *Some false*)$_{\surd}$,(-*1*-))#
[((-*1*-),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)
by(*fastforce intro*: *While-CFG.Cons-path While-CFG.empty-path*
simp: *valid-node-def valid-edge-def*)
with ‹*while* (*b*) *prog′* ⊢ (-*Entry*-) −[((-*Entry*-),(λ*s*. *True*)$_{\surd}$,(-*0*-))]→∗ (-*0*-)› *True*
show *?thesis* **by** *simp blast*
next
case *False*
hence *1* ≤ *l* **by** *simp*
thus *?thesis*
proof(*cases l* < *2*)
case *True*
with ‹*1* ≤ *l*› **have** [*simp*]:*l* = *1* **by** *simp*
from *WCFG-WhileFalseSkip*[*of b prog′*]
have *while* (*b*) *prog′* ⊢ (-*1*-) −[((-*1*-),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)
by(*fastforce intro*: *While-CFG.Cons-path While-CFG.empty-path*
simp: *valid-node-def valid-edge-def*)
from *WCFG-WhileFalse*[*of b prog′*]
have *while* (*b*) *prog′* ⊢ (-*0*-)
−[((-*0*-),(λ*s*. *interpret b s* = *Some false*)$_{\surd}$,(-*1*-))]→∗ (-*1*-)
by(*fastforce intro*: *While-CFG.Cons-path While-CFG.empty-path*
simp: *While-CFG.valid-node-def valid-edge-def*)
with *WCFG-Entry*[*of while* (*b*) *prog′*]
have *while* (*b*) *prog′* ⊢ (-*Entry*-) −((-*Entry*-),(λ*s*. *True*)$_{\surd}$,(-*0*-))#
[((-*0*-),(λ*s*. *interpret b s* = *Some false*)$_{\surd}$,(-*1*-))]→∗ (-*1*-)
by(*fastforce intro*: *While-CFG.Cons-path simp*: *valid-node-def valid-edge-def*)
with ‹*while* (*b*) *prog′* ⊢ (-*1*-) −[((-*1*-),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)›
show *?thesis* **by** *simp blast*
next
case *False*
with ‹*1* ≤ *l*› **have** *2* ≤ *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*) *prog′*› **have** *l′* < #:*prog′* **by** *simp*
from *IH*[*OF this*] **obtain** *as as′* **where** *prog′* ⊢ (- *l′* -) −*as*→∗ (-*Exit*-)
and *prog′* ⊢ (-*Entry*-) −*as′*→∗ (- *l′* -) **by** *blast*
from ‹*prog′* ⊢ (- *l′* -) −*as*→∗ (-*Exit*-)› **obtain** *ax asx* **where**
prog′ ⊢ (- *l′* -) −*asx*@[*ax*]→∗ (-*Exit*-)
by(*induct as rule*:*rev-induct*,*auto elim*:*While-CFG.cases*)
hence *prog′* ⊢ (- *l′* -) −*asx*→∗ *sourcenode ax* **and** *valid-edge prog′ ax*
and (-*Exit*-) = *targetnode ax*
by(*auto intro*:*While-CFG.path-split-snoc*)
then obtain *lx* **where** *sourcenode ax* = (- *lx* -)
by(*cases sourcenode ax*) *auto*
with ‹*prog′* ⊢ (- *l′* -) −*asx*→∗ *sourcenode ax*›
have *while* (*b*) *prog′* ⊢ (- *l′* -) ⊕ *2* −*asx* ⊕*s 2*→∗ *sourcenode ax* ⊕ *2*
by(*fastforce intro*:*path-While simp del*:*label-incr.simps*)
from *WCFG-WhileFalseSkip*[*of b prog′*]
have *while* (*b*) *prog′* ⊢ (-*1*-) −[((-*1*-),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)
by(*fastforce intro*:*While-CFG.Cons-path While-CFG.empty-path*
simp:*valid-node-def valid-edge-def*)
with *WCFG-WhileFalse*[*of b prog′*]
have *while* (*b*) *prog′* ⊢ (-*0*-) −((-*0*-),(λ*s. interpret b s* = *Some false*)$_\surd$,(-*1*-))#
[((-*1*-),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)
by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-node-def valid-edge-def*)
with ‹*valid-edge prog′ ax*› ‹(-*Exit*-) = *targetnode ax*› ‹*sourcenode ax* = (- *lx*
-)›
have *while* (*b*) *prog′* ⊢ *sourcenode ax* ⊕ *2* −(*sourcenode ax* ⊕ *2*,*kind ax*,(-*0*-))#
((-*0*-),(λ*s. interpret b s* = *Some false*)$_\surd$,(-*1*-))#
[((-*1*-),⇑*id*,(-*Exit*-))]→∗ (-*Exit*-)
by(*fastforce intro*:*While-CFG.Cons-path dest*:*WCFG-WhileBodyExit*
simp:*valid-node-def valid-edge-def*)
with ‹*while* (*b*) *prog′* ⊢ (- *l′* -) ⊕ *2* −*asx* ⊕*s 2*→∗ *sourcenode ax* ⊕ *2*›
have *path*:*while* (*b*) *prog′* ⊢ (- *l′* -) ⊕ *2* −(*asx* ⊕*s 2*)@
((*sourcenode ax* ⊕ *2*,*kind ax*,(-*0*-))#
((-*0*-),(λ*s. interpret b s* = *Some false*)$_\surd$,(-*1*-))#
[((-*1*-),⇑*id*,(-*Exit*-))])→∗ (-*Exit*-)
by(*rule While-CFG.path-Append*)
from ‹*prog′* ⊢ (-*Entry*-) −*as′*→∗ (- *l′* -)› **obtain** *ax′ asx′*
where *prog′* ⊢ (-*Entry*-) −*ax′*#*asx′*→∗ (- *l′* -)
by(*cases as′*,*auto elim*:*While-CFG.cases*)
hence (-*Entry*-) = *sourcenode ax′* **and** *valid-edge prog′ ax′*
and *prog′* ⊢ *targetnode ax′* −*asx′*→∗ (- *l′* -)
by(*auto intro*:*While-CFG.path-split-Cons*)
hence *targetnode ax′* = (-*0*-) **by**(*fastforce dest*:*WCFG-EntryD simp*:*valid-edge-def*)
with ‹*prog′* ⊢ *targetnode ax′* −*asx′*→∗ (- *l′* -)›
have *while* (*b*) *prog′* ⊢ (-*0*-) ⊕ *2* −*asx′* ⊕*s 2*→∗ (- *l′* -) ⊕ *2*
by(*fastforce intro*:*path-While*)
with *WCFG-WhileTrue*[*of b prog′*]
have *while* (*b*) *prog′* ⊢ (-*0*-)

−((-0-),(λs. interpret b s = Some true)$_{\surd}$,(-0-) ⊕ 2)#(asx′ ⊕s 2)→*
(- l′ -) ⊕ 2
by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-node-def valid-edge-def*)
with *WCFG-Entry*[*of while* (*b*) *prog′*]
have *while* (*b*) *prog′* ⊢ (-*Entry*-) −((-*Entry*-),(λs. *True*)$_{\surd}$,(-0-))#
((-0-),(λs. *interpret b s* = *Some true*)$_{\surd}$,(-0-) ⊕ 2)#(*asx′* ⊕s 2)→*
(- l′ -) ⊕ 2
by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-node-def valid-edge-def*)
with *path* **show** *?thesis* **by** *simp blast*
qed
qed
qed

lemma *valid-node-Exit-path*:
assumes *valid-node prog n* **shows** ∃ *as*. *prog* ⊢ *n* −*as*→* (-*Exit*-)
proof(*cases n*)
case (*Node l*)
with ‹*valid-node prog n*› **have** *l* < #:*prog*
by(*fastforce dest*:*WCFG-sourcelabel-less-num-nodes WCFG-targetlabel-less-num-nodes*
simp:*valid-node-def valid-edge-def*)
with *Node* **show** *?thesis* **by**(*fastforce dest*:*inner-node-Entry-Exit-path*)
next
case *Entry*
from *WCFG-Entry-Exit*[*of prog*]
have *prog* ⊢ (-*Entry*-) −[((-*Entry*-),(λs. *False*)$_{\surd}$,(-*Exit*-))]→* (-*Exit*-)
by(*fastforce intro*:*While-CFG.Cons-path While-CFG.empty-path*
simp:*valid-node-def valid-edge-def*)
with *Entry* **show** *?thesis* **by** *blast*
next
case *Exit*
with *WCFG-Entry-Exit*[*of prog*]
have *prog* ⊢ *n* −[]→* (-*Exit*-)
by(*fastforce intro*:*While-CFG.empty-path simp*:*valid-node-def valid-edge-def*)
thus *?thesis* **by** *blast*
qed

lemma *valid-node-Entry-path*:
assumes *valid-node prog n* **shows** ∃ *as*. *prog* ⊢ (-*Entry*-) −*as*→* *n*
proof(*cases n*)
case (*Node l*)
with ‹*valid-node prog n*› **have** *l* < #:*prog*
by(*fastforce dest*:*WCFG-sourcelabel-less-num-nodes WCFG-targetlabel-less-num-nodes*
simp:*valid-node-def valid-edge-def*)
with *Node* **show** *?thesis* **by**(*fastforce dest*:*inner-node-Entry-Exit-path*)
next
case *Entry*
with *WCFG-Entry-Exit*[*of prog*]

have *prog* ⊢ (-*Entry*-) −[]→∗ *n*
  by(*fastforce intro*:*While-CFG.empty-path simp*:*valid-node-def valid-edge-def*)
thus *?thesis* **by** *blast*
next
case *Exit*
from *WCFG-Entry-Exit*[*of prog*]
have *prog* ⊢ (-*Entry*-) −[((-*Entry*-),(λ*s. False*)$_{\surd}$,(-*Exit*-))]→∗ (-*Exit*-)
  by(*fastforce intro*:*While-CFG.Cons-path While-CFG.empty-path*
    simp:*valid-node-def valid-edge-def*)
with *Exit* **show** *?thesis* **by** *blast*
qed

4.6.2 Some finiteness considerations

lemma *finite-labels*:*finite* {*l*. ∃ *c*. *labels prog l c*}
proof −
have *finite* {*l*::*nat*. *l* < #:*prog*} **by**(*fastforce intro*:*nat-seg-image-imp-finite*)
moreover have {*l*. ∃ *c*. *labels prog l c*} ⊆ {*l*::*nat*. *l* < #:*prog*}
  by(*fastforce intro*:*label-less-num-inner-nodes*)
ultimately show *?thesis* **by**(*auto intro*:*finite-subset*)
qed

lemma *finite-valid-nodes*:*finite* {*n*. *valid-node prog n*}
proof −
have {*n*. ∃ *n′ et*. *prog* ⊢ *n* −*et*→ *n′*} ⊆
  insert (-*Entry*-) ((λ*l′*. (- *l′* -)) ‘ {*l*. ∃ *c*. *labels prog l c*})
  apply *clarsimp*
  apply(*case-tac x*,*auto*)
  by(*fastforce dest*:*WCFG-sourcelabel-less-num-nodes less-num-inner-nodes-label*)
hence *finite* {*n*. ∃ *n′ et*. *prog* ⊢ *n* −*et*→ *n′*}
  by(*auto intro*:*finite-subset finite-imageI finite-labels*)
have {*n′*. ∃ *n et*. *prog* ⊢ *n* −*et*→ *n′*} ⊆
  insert (-*Exit*-) ((λ*l′*. (- *l′* -)) ‘ {*l*. ∃ *c*. *labels prog l c*})
  apply *clarsimp*
  apply(*case-tac x*,*auto*)
  by(*fastforce dest*:*WCFG-targetlabel-less-num-nodes less-num-inner-nodes-label*)
hence *finite* {*n′*. ∃ *n et*. *prog* ⊢ *n* −*et*→ *n′*}
  by(*auto intro*:*finite-subset finite-imageI finite-labels*)
have {*n*. ∃ *nx et nx′*. *prog* ⊢ *nx* −*et*→ *nx′* ∧ (*n* = *nx* ∨ *n* = *nx′*)} =
  {*n*. ∃ *n′ et*. *prog* ⊢ *n* −*et*→ *n′*} *Un* {*n′*. ∃ *n et*. *prog* ⊢ *n* −*et*→ *n′*} **by** *blast*
with ‹*finite* {*n*. ∃ *n′ et*. *prog* ⊢ *n* −*et*→ *n′*}› ‹*finite* {*n′*. ∃ *n et*. *prog* ⊢ *n* −*et*→ *n′*}›
have *finite* {*n*. ∃ *nx et nx′*. *prog* ⊢ *nx* −*et*→ *nx′* ∧ (*n* = *nx* ∨ *n* = *nx′*)}
  by *fastforce*
thus *?thesis* **by**(*simp add*:*valid-node-def valid-edge-def*)
qed

lemma *finite-successors*:

finite $\{n'.\ \exists\, a'.\ \textit{valid-edge prog } a' \wedge \textit{sourcenode } a' = n \wedge$
$\textit{targetnode } a' = n'\}$
proof −
have $\{n'.\ \exists\, a'.\ \textit{valid-edge prog } a' \wedge \textit{sourcenode } a' = n \wedge$
$\textit{targetnode } a' = n'\} \subseteq \{n.\ \textit{valid-node prog } n\}$
by(*auto simp*:*valid-edge-def valid-node-def*)
thus *?thesis* **by**(*fastforce elim*:*finite-subset intro*:*finite-valid-nodes*)
qed

end

4.7 Interpretations of the various dynamic control dependences

theory *DynamicControlDependences* **imports** *AdditionalLemmas ../Dynamic/DynPDG*
begin

interpretation *WDynStandardControlDependence*:
DynStandardControlDependencePDG sourcenode targetnode kind valid-edge prog
Entry Defs prog Uses prog id Exit
for *prog*
proof(*unfold-locales*)
fix *n* **assume** *CFG.valid-node sourcenode targetnode* (*valid-edge prog*) *n*
hence *valid-node prog n* **by**(*simp add*:*valid-node-def While-CFG.valid-node-def*)
thus $\exists\, as.\ prog \vdash$ (*-Entry-*) $-as\rightarrow*$ *n* **by**(*rule valid-node-Entry-path*)
next
fix *n* **assume** *CFG.valid-node sourcenode targetnode* (*valid-edge prog*) *n*
hence *valid-node prog n* **by**(*simp add*:*valid-node-def While-CFG.valid-node-def*)
thus $\exists\, as.\ prog \vdash n\ -as\rightarrow*$ (*-Exit-*) **by**(*rule valid-node-Exit-path*)
qed

interpretation *WDynWeakControlDependence*:
DynWeakControlDependencePDG sourcenode targetnode kind valid-edge prog
Entry Defs prog Uses prog id Exit
for *prog*
proof(*unfold-locales*)
fix *n* **assume** *CFG.valid-node sourcenode targetnode* (*valid-edge prog*) *n*
hence *valid-node prog n* **by**(*simp add*:*valid-node-def While-CFG.valid-node-def*)
show *finite* $\{n'.\ \exists\, a'.\ \textit{valid-edge prog } a' \wedge \textit{sourcenode } a' = n \wedge$
$\textit{targetnode } a' = n'\}$
by(*rule finite-successors*)
qed

end

4.8 Semantics

theory *Semantics* **imports** *Labels Com* **begin**

4.8.1 Small Step Semantics

inductive *red* :: *cmd* ∗ *state* ⇒ *cmd* ∗ *state* ⇒ *bool*
and *red′* :: *cmd* ⇒ *state* ⇒ *cmd* ⇒ *state* ⇒ *bool*
(‹((1⟨-,/-⟩) →/ (1⟨-,/-⟩))› [0,0,0,0] 81)
where
⟨c,s⟩ → ⟨c′,s′⟩ == red (c,s) (c′,s′)
| *RedLAss*:
⟨V:=e,s⟩ → ⟨Skip,s(V:=(interpret e s))⟩

| *SeqRed*:
$\langle c_1,s\rangle \rightarrow \langle c_1',s'\rangle \Longrightarrow \langle c_1;;c_2,s\rangle \rightarrow \langle c_1';;c_2,s'\rangle$

| *RedSeq*:
$\langle Skip;;c_2,s\rangle \rightarrow \langle c_2,s\rangle$

| *RedCondTrue*:
$interpret\ b\ s = Some\ true \Longrightarrow \langle if\ (b)\ c_1\ else\ c_2,s\rangle \rightarrow \langle c_1,s\rangle$

| *RedCondFalse*:
$interpret\ b\ s = Some\ false \Longrightarrow \langle if\ (b)\ c_1\ else\ c_2,s\rangle \rightarrow \langle c_2,s\rangle$

| *RedWhileTrue*:
interpret b s = Some true ⟹ ⟨while (b) c,s⟩ → ⟨c;;while (b) c,s⟩

| *RedWhileFalse*:
interpret b s = Some false ⟹ ⟨while (b) c,s⟩ → ⟨Skip,s⟩

lemmas *red-induct = red.induct[split-format (complete)]*

abbreviation *reds* ::*cmd* ⇒ *state* ⇒ *cmd* ⇒ *state* ⇒ *bool*
(‹((1⟨-,/-⟩) →∗/ (1⟨-,/-⟩))› [0,0,0,0] 81) **where**
⟨c,s⟩ →∗ ⟨c′,s′⟩ == red^{**} (c,s) (c′,s′)

4.8.2 Label Semantics

inductive *step* :: *cmd* ⇒ *cmd* ⇒ *state* ⇒ *nat* ⇒ *cmd* ⇒ *state* ⇒ *nat* ⇒ *bool*
(‹(- ⊢ (1⟨-,/-,/-⟩) ↝/ (1⟨-,/-,/-⟩))› [51,0,0,0,0,0,0] 81)
where

StepLAss:
V:=e ⊢ ⟨V:=e,s,0⟩ ↝ ⟨Skip,s(V:=(interpret e s)),1⟩

| *StepSeq*:
$[\![labels\ (c_1;;c_2)\ l\ (Skip;;c_2);\ labels\ (c_1;;c_2)\ \#{:}c_1\ \ c_2;\ l < \#{:}c_1]\!]$
$\Longrightarrow c_1;;c_2 \vdash \langle Skip;;c_2,s,l\rangle \leadsto \langle c_2,s,\#{:}c_1\rangle$

| *StepSeqWhile*:
labels $(while\ (b)\ c')\ l\ (Skip;;while\ (b)\ c')$
$\Longrightarrow$ *while* $(b)\ c' \vdash \langle Skip;;while\ (b)\ c',s,l\rangle \rightsquigarrow \langle while\ (b)\ c',s,0\rangle$

| *StepCondTrue*:
interpret $b\ s = Some\ true$
$\Longrightarrow$ *if* $(b)\ c_1\ else\ c_2 \vdash \langle if\ (b)\ c_1\ else\ c_2,s,0\rangle \rightsquigarrow \langle c_1,s,1\rangle$

| *StepCondFalse*:
interpret $b\ s = Some\ false$
$\Longrightarrow$ *if* $(b)\ c_1\ else\ c_2 \vdash \langle if\ (b)\ c_1\ else\ c_2,s,0\rangle \rightsquigarrow \langle c_2,s,\#{:}c_1 + 1\rangle$

| *StepWhileTrue*:
interpret $b\ s = Some\ true$
$\Longrightarrow$ *while* $(b)\ c \vdash \langle while\ (b)\ c,s,0\rangle \rightsquigarrow \langle c;;while\ (b)\ c,s,2\rangle$

| *StepWhileFalse*:
interpret $b\ s = Some\ false \Longrightarrow while\ (b)\ c \vdash \langle while\ (b)\ c,s,0\rangle \rightsquigarrow \langle Skip,s,1\rangle$

| *StepRecSeq1*:
$prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle$
$\Longrightarrow prog;;c_2 \vdash \langle c;;c_2,s,l\rangle \rightsquigarrow \langle c';;c_2,s',l'\rangle$

| *StepRecSeq2*:
$prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle$
$\Longrightarrow c_1;;prog \vdash \langle c,s,l + \#{:}c_1\rangle \rightsquigarrow \langle c',s',l' + \#{:}c_1\rangle$

| *StepRecCond1*:
$prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle$
$\Longrightarrow if\ (b)\ prog\ else\ c_2 \vdash \langle c,s,l + 1\rangle \rightsquigarrow \langle c',s',l' + 1\rangle$

| *StepRecCond2*:
$prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle$
$\Longrightarrow if\ (b)\ c_1\ else\ prog \vdash \langle c,s,l + \#{:}c_1 + 1\rangle \rightsquigarrow \langle c',s',l' + \#{:}c_1 + 1\rangle$

| *StepRecWhile*:
$cx \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle$
$\Longrightarrow while\ (b)\ cx \vdash \langle c;;while\ (b)\ cx,s,l + 2\rangle \rightsquigarrow \langle c';;while\ (b)\ cx,s',l' + 2\rangle$

lemma *step-label-less*:
$prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle \Longrightarrow l < \#{:}prog \wedge l' < \#{:}prog$
proof(*induct rule*:*step.induct*)
case (*StepSeq* $c_1\ c_2\ l\ s$)
from ‹*labels* $(c_1;;c_2)\ l\ (Skip;;c_2)$›
have $l < \#{:}(c_1;;\ c_2)$ **by**(*rule label-less-num-inner-nodes*)
thus *?case* **by**(*simp add*:*num-inner-nodes-gr-0*)
next

case (*StepSeqWhile b cx l s*)
from ‹*labels* (*while* (*b*) *cx*) *l* (*Skip*;;*while* (*b*) *cx*)›
have $l < \#$:(*while* (*b*) *cx*) **by**(*rule label-less-num-inner-nodes*)
thus *?case* **by** *simp*
qed (*auto intro*:*num-inner-nodes-gr-0*)

abbreviation *steps* :: *cmd* ⇒ *cmd* ⇒ *state* ⇒ *nat* ⇒ *cmd* ⇒ *state* ⇒ *nat* ⇒ *bool*
(‹(- ⊢ (*1*⟨-,/-,/-⟩) ↝∗/ (*1*⟨-,/-,/-⟩))› [*51*,*0*,*0*,*0*,*0*,*0*,*0*] *81*) **where**
prog ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c*′,*s*′,*l*′⟩ ==
(λ(*c*,*s*,*l*) (*c*′,*s*′,*l*′). *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c*′,*s*′,*l*′⟩)** (*c*,*s*,*l*) (*c*′,*s*′,*l*′)

4.8.3 Proof of bisimulation of ⟨*c*,*s*⟩ → ⟨*c*′,*s*′⟩ and *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c*′,*s*′,*l*′⟩ via *labels*

From *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c*′,*s*′,*l*′⟩ **to** ⟨*c*,*s*⟩ → ⟨*c*′,*s*′⟩

lemma *step-red*:
prog ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c*′,*s*′,*l*′⟩ ⟹ ⟨*c*,*s*⟩ → ⟨*c*′,*s*′⟩
by(*induct rule*:*step.induct*,*rule RedLAss*,*auto intro*:*red.intros*)

lemma *steps-reds*:
prog ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c*′,*s*′,*l*′⟩ ⟹ ⟨*c*,*s*⟩ →∗ ⟨*c*′,*s*′⟩
proof(*induct rule*:*converse-rtranclp-induct3*)
case *refl* **thus** *?case* **by** *simp*
next
case (*step c s l c*″ *s*″ *l*″)
then have *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c*″,*s*″,*l*″⟩
and ⟨*c*″,*s*″⟩ →∗ ⟨*c*′,*s*′⟩ **by** *simp-all*
from ‹*prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c*″,*s*″,*l*″⟩› **have** ⟨*c*,*s*⟩ → ⟨*c*″,*s*″⟩
by(*fastforce intro*:*step-red*)
with ‹⟨*c*″,*s*″⟩ →∗ ⟨*c*′,*s*′⟩› **show** *?case*
by(*fastforce intro*:*converse-rtranclp-into-rtranclp*)
qed

From ⟨*c*,*s*⟩ → ⟨*c*′,*s*′⟩ **and** *labels* **to** *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c*′,*s*′,*l*′⟩

lemma *red-step*:
⟦*labels prog l c*; ⟨*c*,*s*⟩ → ⟨*c*′,*s*′⟩⟧
⟹ ∃ *l*′. *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c*′,*s*′,*l*′⟩ ∧ *labels prog l*′ *c*′
proof(*induct arbitrary*:*c*′ *rule*:*labels.induct*)
case (*Labels-Base c*)
from ‹⟨*c*,*s*⟩ → ⟨*c*′,*s*′⟩› **show** *?case*
proof(*induct rule*:*red-induct*)
case (*RedLAss V e s*)

have $V{:=}e \vdash \langle V{:=}e,s,0\rangle \leadsto \langle Skip,s(V{:=}(interpret\ e\ s)),1\rangle$ **by**(*rule StepLAss*)
have *labels* $(V{:=}e)$ *1 Skip* **by**(*fastforce intro*:*Labels-LAss*)
with $‹V{:=}e \vdash \langle V{:=}e,s,0\rangle \leadsto \langle Skip,s(V{:=}(interpret\ e\ s)),1\rangle›$ **show** *?case* **by** *blast*
next
case $(SeqRed\ c_1\ s\ c_1{}'\ s'\ c_2)$
from $‹\exists l'.\ c_1 \vdash \langle c_1,s,0\rangle \leadsto \langle c_1{}',s',l'\rangle \wedge labels\ c_1\ l'\ c_1{}'›$
obtain l' **where** $c_1 \vdash \langle c_1,s,0\rangle \leadsto \langle c_1{}',s',l'\rangle$ **and** $labels\ c_1\ l'\ c_1{}'$ **by** *blast*
from $‹c_1 \vdash \langle c_1,s,0\rangle \leadsto \langle c_1{}',s',l'\rangle›$ **have** $c_1;;c_2 \vdash \langle c_1;;c_2,s,0\rangle \leadsto \langle c_1{}';;c_2,s',l'\rangle$
by(*rule StepRecSeq1*)
moreover
from $‹labels\ c_1\ l'\ c_1{}'›$ **have** $labels\ (c_1;;c_2)\ l'\ (c_1{}';;c_2)$ **by**(*rule Labels-Seq1*)
ultimately show *?case* **by** *blast*
next
case $(RedSeq\ c_2\ s)$
have $labels\ c_2\ 0\ c_2$ **by**(*rule Labels.Labels-Base*)
hence $labels\ (Skip;;c_2)\ (0 + \#{:}Skip)\ c_2$ **by**(*rule Labels-Seq2*)
have $labels\ (Skip;;c_2)\ 0\ (Skip;;c_2)$ **by**(*rule Labels.Labels-Base*)
with $‹labels\ (Skip;;c_2)\ (0 + \#{:}Skip)\ c_2›$
have $Skip;;c_2 \vdash \langle Skip;;c_2,s,0\rangle \leadsto \langle c_2,s,\#{:}Skip\rangle$
by(*fastforce intro*:*StepSeq*)
with $‹labels\ (Skip;;c_2)\ (0 + \#{:}Skip)\ c_2›$ **show** *?case* **by** *auto*
next
case $(RedCondTrue\ b\ s\ c_1\ c_2)$
from $‹interpret\ b\ s = Some\ true›$
have $if\ (b)\ c_1\ else\ c_2 \vdash \langle if\ (b)\ c_1\ else\ c_2,s,0\rangle \leadsto \langle c_1,s,1\rangle$
by(*rule StepCondTrue*)
have $labels\ (if\ (b)\ c_1\ else\ c_2)\ (0 + 1)\ c_1$
by(*rule Labels-CondTrue*,*rule Labels.Labels-Base*)
with $‹if\ (b)\ c_1\ else\ c_2 \vdash \langle if\ (b)\ c_1\ else\ c_2,s,0\rangle \leadsto \langle c_1,s,1\rangle›$ **show** *?case* **by** *auto*
next
case $(RedCondFalse\ b\ s\ c_1\ c_2)$
from $‹interpret\ b\ s = Some\ false›$
have $if\ (b)\ c_1\ else\ c_2 \vdash \langle if\ (b)\ c_1\ else\ c_2,s,0\rangle \leadsto \langle c_2,s,\#{:}c_1 + 1\rangle$
by(*rule StepCondFalse*)
have $labels\ (if\ (b)\ c_1\ else\ c_2)\ (0 + \#{:}c_1 + 1)\ c_2$
by(*rule Labels-CondFalse*,*rule Labels.Labels-Base*)
with $‹if\ (b)\ c_1\ else\ c_2 \vdash \langle if\ (b)\ c_1\ else\ c_2,s,0\rangle \leadsto \langle c_2,s,\#{:}c_1 + 1\rangle›$
show *?case* **by** *auto*
next
case $(RedWhileTrue\ b\ s\ c)$
from $‹interpret\ b\ s = Some\ true›$
have $while\ (b)\ c \vdash \langle while\ (b)\ c,s,0\rangle \leadsto \langle c;;\ while\ (b)\ c,s,2\rangle$
by(*rule StepWhileTrue*)
have $labels\ (while\ (b)\ c)\ (0 + 2)\ (c;;\ while\ (b)\ c)$
by(*rule Labels-WhileBody*,*rule Labels.Labels-Base*)
with $‹while\ (b)\ c \vdash \langle while\ (b)\ c,s,0\rangle \leadsto \langle c;;\ while\ (b)\ c,s,2\rangle›$
show *?case* **by**(*auto simp del*:*add-2-eq-Suc'*)
next

case (*RedWhileFalse b s c*)

from ‹*interpret b s = Some false*›

have $while\ (b)\ c \vdash \langle while\ (b)\ c,s,0\rangle \rightsquigarrow \langle Skip,s,1\rangle$

by(*rule StepWhileFalse*)

have *labels* (*while* (*b*) *c*) *1 Skip* **by**(*rule Labels-WhileExit*)

with ‹$while\ (b)\ c \vdash \langle while\ (b)\ c,s,0\rangle \rightsquigarrow \langle Skip,s,1\rangle$› **show** *?case* **by** *auto*

qed

next

case (*Labels-LAss V e*)

from ‹$\langle Skip,s\rangle \rightarrow \langle c',s'\rangle$› **have** *False* **by**(*auto elim*:*red.cases*)

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

next

case (*Labels-Seq1* c_1 *l c* c_2)

note *IH* = ‹$\bigwedge c'.\ \langle c,s\rangle \rightarrow \langle c',s'\rangle \Longrightarrow$

$\exists\, l'.\ c_1 \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle \wedge labels\ c_1\ l'\ c'$›

from ‹$\langle c;;c_2,s\rangle \rightarrow \langle c',s'\rangle$›

have $(c = Skip \wedge c' = c_2 \wedge s = s') \vee (\exists\, c''.\ c' = c'';;c_2)$

by −(*erule red.cases*,*auto*)

thus *?case*

proof

assume [*simp*]:$c = Skip \wedge c' = c_2 \wedge s = s'$

from ‹*labels* c_1 *l c*› **have** $l < \#{:}c_1$

by(*rule label-less-num-inner-nodes*[*simplified*])

have *labels* $(c_1;;c_2)\ (0 + \#{:}c_1)\ c_2$

by(*rule Labels-Seq2*,*rule Labels-Base*)

from ‹*labels* c_1 *l c*› **have** *labels* $(c_1;;\ c_2)\ l\ (Skip;;c_2)$

by(*fastforce intro*:*Labels.Labels-Seq1*)

with ‹*labels* $(c_1;;c_2)\ (0 + \#{:}c_1)\ c_2$› ‹$l < \#{:}c_1$›

have $c_1;;\ c_2 \vdash \langle Skip;;c_2,s,l\rangle \rightsquigarrow \langle c_2,s,\#{:}c_1\rangle$

by(*fastforce intro*:*StepSeq*)

with ‹*labels* $(c_1;;c_2)\ (0 + \#{:}c_1)\ c_2$› **show** *?case* **by** *auto*

next

assume $\exists\, c''.\ c' = c'';;c_2$

then obtain c'' **where** [*simp*]:$c' = c'';;c_2$ **by** *blast*

have $c_2 \neq c'';;\ c_2$

by (*induction* c_2) *auto*

with ‹$\langle c;;c_2,s\rangle \rightarrow \langle c',s'\rangle$› **have** $\langle c,s\rangle \rightarrow \langle c'',s'\rangle$

by (*auto elim*!:*red.cases*)

from *IH*[*OF this*] **obtain** l' **where** $c_1 \vdash \langle c,s,l\rangle \rightsquigarrow \langle c'',s',l'\rangle$

and *labels* $c_1\ l'\ c''$ **by** *blast*

from ‹$c_1 \vdash \langle c,s,l\rangle \rightsquigarrow \langle c'',s',l'\rangle$› **have** $c_1;;c_2 \vdash \langle c;;c_2,s,l\rangle \rightsquigarrow \langle c'';;c_2,s',l'\rangle$

by(*rule StepRecSeq1*)

from ‹*labels* $c_1\ l'\ c''$› **have** *labels* $(c_1;;c_2)\ l'\ (c'';;c_2)$

by(*rule Labels.Labels-Seq1*)

with ‹$c_1;;c_2 \vdash \langle c;;c_2,s,l\rangle \rightsquigarrow \langle c'';;c_2,s',l'\rangle$› **show** *?case* **by** *auto*

qed

next

case (*Labels-Seq2* c_2 *l c* c_1 c')

note *IH* = ‹$\bigwedge c'.\ \langle c,s\rangle \rightarrow \langle c',s'\rangle \Longrightarrow$

$\exists l'.\ c_2 \vdash \langle c,s,l\rangle \leadsto \langle c',s',l'\rangle \wedge$ *labels* $c_2\ l'\ c'$›
from *IH*[*OF* ‹$\langle c,s\rangle \rightarrow \langle c',s'\rangle$›] **obtain** l' **where** $c_2 \vdash \langle c,s,l\rangle \leadsto \langle c',s',l'\rangle$
and *labels* $c_2\ l'\ c'$ **by** *blast*
from ‹$c_2 \vdash \langle c,s,l\rangle \leadsto \langle c',s',l'\rangle$› **have** $c_1;;\ c_2 \vdash \langle c,s,l + \#{:}c_1\rangle \leadsto \langle c',s',l' + \#{:}c_1\rangle$
by(*rule StepRecSeq2*)
moreover
from ‹*labels* $c_2\ l'\ c'$› **have** *labels* $(c_1;;c_2)\ (l' + \#{:}c_1)\ c'$
by(*rule Labels.Labels-Seq2*)
ultimately show *?case* **by** *blast*
next
case (*Labels-CondTrue* $c_1\ l\ c\ b\ c_2\ c'$)
note *label* = ‹*labels* $c_1\ l\ c$› **and** *red* = ‹$\langle c,s\rangle \rightarrow \langle c',s'\rangle$›
and *IH* = ‹$\bigwedge c'.\ \langle c,s\rangle \rightarrow \langle c',s'\rangle \Longrightarrow$
$\exists l'.\ c_1 \vdash \langle c,s,l\rangle \leadsto \langle c',s',l'\rangle \wedge$ *labels* $c_1\ l'\ c'$›
from *IH*[*OF* ‹$\langle c,s\rangle \rightarrow \langle c',s'\rangle$›] **obtain** l' **where** $c_1 \vdash \langle c,s,l\rangle \leadsto \langle c',s',l'\rangle$
and *labels* $c_1\ l'\ c'$ **by** *blast*
from ‹$c_1 \vdash \langle c,s,l\rangle \leadsto \langle c',s',l'\rangle$›
have *if* $(b)\ c_1$ *else* $c_2 \vdash \langle c,s,l + 1\rangle \leadsto \langle c',s',l' + 1\rangle$
by(*rule StepRecCond1*)
moreover
from ‹*labels* $c_1\ l'\ c'$› **have** *labels* (*if* $(b)\ c_1$ *else* $c_2)\ (l' + 1)\ c'$
by(*rule Labels.Labels-CondTrue*)
ultimately show *?case* **by** *blast*
next
case (*Labels-CondFalse* $c_2\ l\ c\ b\ c_1\ c'$)
note *IH* = ‹$\bigwedge c'.\ \langle c,s\rangle \rightarrow \langle c',s'\rangle \Longrightarrow$
$\exists l'.\ c_2 \vdash \langle c,s,l\rangle \leadsto \langle c',s',l'\rangle \wedge$ *labels* $c_2\ l'\ c'$›
from *IH*[*OF* ‹$\langle c,s\rangle \rightarrow \langle c',s'\rangle$›] **obtain** l' **where** $c_2 \vdash \langle c,s,l\rangle \leadsto \langle c',s',l'\rangle$
and *labels* $c_2\ l'\ c'$ **by** *blast*
from ‹$c_2 \vdash \langle c,s,l\rangle \leadsto \langle c',s',l'\rangle$›
have *if* $(b)\ c_1$ *else* $c_2 \vdash \langle c,s,l + \#{:}c_1 + 1\rangle \leadsto \langle c',s',l' + \#{:}c_1 + 1\rangle$
by(*rule StepRecCond2*)
moreover
from ‹*labels* $c_2\ l'\ c'$› **have** *labels* (*if* $(b)\ c_1$ *else* $c_2)\ (l' + \#{:}c_1 + 1)\ c'$
by(*rule Labels.Labels-CondFalse*)
ultimately show *?case* **by** *blast*
next
case (*Labels-WhileBody* $c'\ l\ c\ b\ cx$)
note *IH* = ‹$\bigwedge c''.\ \langle c,s\rangle \rightarrow \langle c'',s'\rangle \Longrightarrow$
$\exists l'.\ c' \vdash \langle c,s,l\rangle \leadsto \langle c'',s',l'\rangle \wedge$ *labels* $c'\ l'\ c''$›
from ‹$\langle c;;$*while* $(b)\ c',s\rangle \rightarrow \langle cx,s'\rangle$›
have $(c = Skip \wedge cx =$ *while* $(b)\ c' \wedge s = s') \vee (\exists c''.\ cx = c'';;$*while* $(b)\ c')$
by −(*erule red.cases*,*auto*)
thus *?case*
proof
assume [*simp*]:$c = Skip \wedge cx =$ *while* $(b)\ c' \wedge s = s'$
have *labels* (*while* $(b)\ c')$ *0* (*while* $(b)\ c')$
by(*fastforce intro*:*Labels-Base*)
from ‹*labels* $c'\ l\ c$› **have** *labels* (*while* $(b)\ c')\ (l + 2)\ (Skip;;$*while* $(b)\ c')$

by(*fastforce intro*:*Labels.Labels-WhileBody simp del*:*add-2-eq-Suc*′)
hence *while* (*b*) *c*′ ⊢ ⟨*Skip*;;*while* (*b*) *c*′,*s*,*l* + *2*⟩ ↝ ⟨*while* (*b*) *c*′,*s*,*0*⟩
by(*rule StepSeqWhile*)
with ‹*labels* (*while* (*b*) *c*′) *0* (*while* (*b*) *c*′)› **show** *?case* **by** *simp blast*
next
assume ∃ *c*′′. *cx* = *c*′′;;*while* (*b*) *c*′
then obtain *c*′′ **where** [*simp*]:*cx* = *c*′′;;*while* (*b*) *c*′ **by** *blast*
with ‹⟨*c*;;*while* (*b*) *c*′,*s*⟩ → ⟨*cx*,*s*′⟩› **have** ⟨*c*,*s*⟩ → ⟨*c*′′,*s*′⟩
by(*auto elim*:*red.cases*)
from *IH*[*OF this*] **obtain** *l*′ **where** *c*′ ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c*′′,*s*′,*l*′⟩
and *labels c*′ *l*′ *c*′′ **by** *blast*
from ‹*c*′ ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c*′′,*s*′,*l*′⟩›
have *while* (*b*) *c*′ ⊢ ⟨*c*;;*while* (*b*) *c*′,*s*,*l* + *2*⟩ ↝ ⟨*c*′′;;*while* (*b*) *c*′,*s*′,*l*′ + *2*⟩
by(*rule StepRecWhile*)
moreover
from ‹*labels c*′ *l*′ *c*′′› **have** *labels* (*while* (*b*) *c*′) (*l*′ + *2*) (*c*′′;;*while* (*b*) *c*′)
by(*rule Labels.Labels-WhileBody*)
ultimately show *?case* **by** *simp blast*
qed
next
case (*Labels-WhileExit b c*′ *c*′′)
from ‹⟨*Skip*,*s*⟩ → ⟨*c*′′,*s*′⟩› **have** *False* **by**(*auto elim*:*red.cases*)
thus *?case* **by** *simp*
qed

lemma *reds-steps*:
⟦⟨*c*,*s*⟩ →∗ ⟨*c*′,*s*′⟩; *labels prog l c*⟧
⟹ ∃ *l*′. *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c*′,*s*′,*l*′⟩ ∧ *labels prog l*′ *c*′
proof(*induct rule*:*rtranclp-induct2*)
case *refl*
from ‹*labels prog l c*› **show** *?case* **by** *blast*
next
case (*step c*′′ *s*′′ *c*′ *s*′)
note *IH* = ‹*labels prog l c* ⟹
∃ *l*′. *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c*′′,*s*′′,*l*′⟩ ∧ *labels prog l*′ *c*′′›
from *IH*[*OF* ‹*labels prog l c*›] **obtain** *l*′′ **where** *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c*′′,*s*′′,*l*′′⟩
and *labels prog l*′′ *c*′′ **by** *blast*
from ‹*labels prog l*′′ *c*′′› ‹⟨*c*′′,*s*′′⟩ → ⟨*c*′,*s*′⟩› **obtain** *l*′
where *prog* ⊢ ⟨*c*′′,*s*′′,*l*′′⟩ ↝ ⟨*c*′,*s*′,*l*′⟩
and *labels prog l*′ *c*′ **by**(*auto dest*:*red-step*)
from ‹*prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c*′′,*s*′′,*l*′′⟩› ‹*prog* ⊢ ⟨*c*′′,*s*′′,*l*′′⟩ ↝ ⟨*c*′,*s*′,*l*′⟩›
have *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c*′,*s*′,*l*′⟩
by(*fastforce elim*:*rtranclp-trans*)
with ‹*labels prog l*′ *c*′› **show** *?case* **by** *blast*
qed

The bisimulation theorem

theorem *reds-steps-bisimulation*:
  labels prog l c $\Longrightarrow$ ($\langle c,s\rangle \rightarrow* \langle c',s'\rangle$) =
    ($\exists$ *l'*. *prog* $\vdash \langle c,s,l\rangle \rightsquigarrow* \langle c',s',l'\rangle \wedge$ *labels prog l' c'*)
  by(*fastforce intro*:*reds-steps elim*:*steps-reds*)

end

4.9 Equivalence

theory *WEquivalence* **imports** *Semantics WCFG* **begin**

4.9.1 From *prog* $\vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle$ to $c \vdash$ (- *l* -) $-et\rightarrow$ (- *l'* -) with *transfers* and *preds*

lemma *Skip-WCFG-edge-Exit*:
  $[\![$*labels prog l Skip*$]\!] \Longrightarrow$ *prog* $\vdash$ (- *l* -) $-\Uparrow id\rightarrow$ (-*Exit*-)
proof(*induct prog l Skip rule*:*labels.induct*)
  case *Labels-Base*
  show *?case* **by**(*fastforce intro*:*WCFG-Skip*)
next
  case (*Labels-LAss V e*)
  show *?case* **by**(*rule WCFG-LAssSkip*)
next
  case (*Labels-Seq2* c_2 *l* c_1)
  from ‹$c_2 \vdash$ (- *l* -) $-\Uparrow id\rightarrow$ (-*Exit*-)›
  have $c_1;;c_2 \vdash$ (- *l* -) $\oplus$ #:c_1 $-\Uparrow id\rightarrow$ (-*Exit*-) $\oplus$ #:c_1
    by(*fastforce intro*:*WCFG-SeqSecond*)
  thus *?case* **by**(*simp del*:*id-apply*)
next
  case (*Labels-CondTrue* c_1 *l b* c_2)
  from ‹$c_1 \vdash$ (- *l* -) $-\Uparrow id\rightarrow$ (-*Exit*-)›
  have *if* (*b*) c_1 *else* $c_2 \vdash$ (- *l* -) $\oplus$ *1* $-\Uparrow id\rightarrow$ (-*Exit*-) $\oplus$ *1*
    by(*fastforce intro*:*WCFG-CondThen*)
  thus *?case* **by**(*simp del*:*id-apply*)
next
  case (*Labels-CondFalse* c_2 *l b* c_1)
  from ‹$c_2 \vdash$ (- *l* -) $-\Uparrow id\rightarrow$ (-*Exit*-)›
  have *if* (*b*) c_1 *else* $c_2 \vdash$ (- *l* -) $\oplus$ (#:c_1 + *1*) $-\Uparrow id\rightarrow$ (-*Exit*-) $\oplus$ (#:c_1 + *1*)
    by(*fastforce intro*:*WCFG-CondElse*)
  thus *?case* **by**(*simp del*:*id-apply*)
next
  case (*Labels-WhileExit b c'*)
  show *?case* **by**(*rule WCFG-WhileFalseSkip*)
qed

lemma *step-WCFG-edge*:

assumes *prog* ⊢ ⟨*c*,*s*,*l*⟩ ⇝ ⟨*c*′,*s*′,*l*′⟩
obtains *et* **where** *prog* ⊢ (- *l* -) −*et*→ (- *l*′ -) **and** *transfer et s* = *s*′
and *pred et s*
proof −
from ‹*prog* ⊢ ⟨*c*,*s*,*l*⟩ ⇝ ⟨*c*′,*s*′,*l*′⟩›
have ∃ *et*. *prog* ⊢ (- *l* -) −*et*→ (- *l*′ -) ∧ *transfer et s* = *s*′ ∧ *pred et s*
proof(*induct rule*:*step.induct*)
case (*StepLAss V e s*)
have *pred* ⇑(λ*s*. *s*(*V*:=(*interpret e s*))) *s* **by** *simp*
have *V*:=*e* ⊢ (-*0*-) −⇑(λ*s*. *s*(*V*:=(*interpret e s*)))→ (-*1*-)
by(*rule WCFG-LAss*)
have *transfer* ⇑(λ*s*. *s*(*V*:=(*interpret e s*))) *s* = *s*(*V*:=(*interpret e s*)) **by** *simp*
with ‹*pred* ⇑(λ*s*. *s*(*V*:=(*interpret e s*))) *s*›
‹*V*:=*e* ⊢ (-*0*-) −⇑(λ*s*. *s*(*V*:=(*interpret e s*)))→ (-*1*-)› **show** *?case* **by** *blast*
next
case (*StepSeq* c_1 c_2 *l s*)
from ‹*labels* (c_1;;c_2) *l* (*Skip*;;c_2)› ‹*l* < #:c_1› **have** *labels* c_1 *l Skip*
by(*auto elim*:*labels.cases intro*:*Labels-Base*)
hence c_1 ⊢ (- *l* -) −⇑*id*→ (-*Exit*-)
by(*fastforce intro*:*Skip-WCFG-edge-Exit*)
hence c_1;;c_2 ⊢ (- *l* -) −⇑*id*→ (-*0*-) ⊕ #:c_1
by(*rule WCFG-SeqConnect*,*simp*)
thus *?case* **by** *auto*
next
case (*StepSeqWhile b cx l s*)
from ‹*labels* (*while* (*b*) *cx*) *l* (*Skip*;;*while* (*b*) *cx*)›
obtain *lx* **where** *labels cx lx Skip*
and [*simp*]:*l* = *lx* + *2* **by**(*auto elim*:*labels.cases*)
hence *cx* ⊢ (- *lx* -) −⇑*id*→ (-*Exit*-)
by(*fastforce intro*:*Skip-WCFG-edge-Exit*)
hence *while* (*b*) *cx* ⊢ (- *lx* -) ⊕ *2* −⇑*id*→ (-*0*-)
by(*fastforce intro*:*WCFG-WhileBodyExit*)
thus *?case* **by** *auto*
next
case (*StepCondTrue b s* c_1 c_2)
from ‹*interpret b s* = *Some true*›
have *pred* (λ*s*. *interpret b s* = *Some true*)$_{\surd}$ *s* **by** *simp*
moreover
have *if* (*b*) c_1 *else* c_2 ⊢ (-*0*-) −(λ*s*. *interpret b s* = *Some true*)$_{\surd}$→ (-*0*-) ⊕ *1*
by(*rule WCFG-CondTrue*)
moreover
have *transfer* (λ*s*. *interpret b s* = *Some true*)$_{\surd}$ *s* = *s* **by** *simp*
ultimately show *?case* **by** *auto*
next
case (*StepCondFalse b s* c_1 c_2)
from ‹*interpret b s* = *Some false*›
have *pred* (λ*s*. *interpret b s* = *Some false*)$_{\surd}$ *s* **by** *simp*
moreover
have *if* (*b*) c_1 *else* c_2 ⊢ (-*0*-) −(λ*s*. *interpret b s* = *Some false*)$_{\surd}$→

(-0-) ⊕ (#:c_1 + 1)
by(rule WCFG-CondFalse)
moreover
have transfer (λs. interpret b s = Some false)$_\surd$ s = s **by** simp
ultimately show ?case **by** auto
next
case (StepWhileTrue b s c)
from ‹interpret b s = Some true›
have pred (λs. interpret b s = Some true)$_\surd$ s **by** simp
moreover
have while (b) c ⊢ (-0-) −(λs. interpret b s = Some true)$_\surd$→ (-0-) ⊕ 2
by(rule WCFG-WhileTrue)
moreover
have transfer (λs. interpret b s = Some true)$_\surd$ s = s **by** simp
ultimately show ?case **by**(auto simp del:add-2-eq-Suc′)
next
case (StepWhileFalse b s c)
from ‹interpret b s = Some false›
have pred (λs. interpret b s = Some false)$_\surd$ s **by** simp
moreover
have while (b) c ⊢ (-0-) −(λs. interpret b s = Some false)$_\surd$→ (-1-)
by(rule WCFG-WhileFalse)
moreover
have transfer (λs. interpret b s = Some false)$_\surd$ s = s **by** simp
ultimately show ?case **by** auto
next
case (StepRecSeq1 prog c s l c′ s′ l′ c_2)
from ‹∃ et. prog ⊢ (- l -) −et→ (- l′ -) ∧ transfer et s = s′ ∧ pred et s›
obtain et **where** prog ⊢ (- l -) −et→ (- l′ -)
and transfer et s = s′ **and** pred et s **by** blast
moreover
from ‹prog ⊢ (- l -) −et→ (- l′ -)› **have** prog;;c_2 ⊢ (- l -) −et→ (- l′ -)
by(fastforce intro:WCFG-SeqFirst)
ultimately show ?case **by** blast
next
case (StepRecSeq2 prog c s l c′ s′ l′ c_1)
from ‹∃ et. prog ⊢ (- l -) −et→ (- l′ -) ∧ transfer et s = s′ ∧ pred et s›
obtain et **where** prog ⊢ (- l -) −et→ (- l′ -)
and transfer et s = s′ **and** pred et s **by** blast
moreover
from ‹prog ⊢ (- l -) −et→ (- l′ -)›
have c_1;;prog ⊢ (- l -) ⊕ #:c_1 −et→ (- l′ -) ⊕ #:c_1
by(fastforce intro:WCFG-SeqSecond)
ultimately show ?case **by** simp blast
next
case (StepRecCond1 prog c s l c′ s′ l′ b c_2)
from ‹∃ et. prog ⊢ (- l -) −et→ (- l′ -) ∧ transfer et s = s′ ∧ pred et s›
obtain et **where** prog ⊢ (- l -) −et→ (- l′ -)
and transfer et s = s′ **and** pred et s **by** blast

moreover
from ‹*prog* ⊢ (- *l* -) −*et*→ (- *l'* -)›
have *if* (*b*) *prog else* c_2 ⊢ (- *l* -) ⊕ *1* −*et*→ (- *l'* -) ⊕ *1*
by(*fastforce intro*:*WCFG-CondThen*)
ultimately show *?case* **by** *simp blast*
next
case (*StepRecCond2 prog c s l c' s' l' b* c_1)
from ‹∃ *et*. *prog* ⊢ (- *l* -) −*et*→ (- *l'* -) ∧ *transfer et s* = *s'* ∧ *pred et s*›
obtain *et* **where** *prog* ⊢ (- *l* -) −*et*→ (- *l'* -)
and *transfer et s* = *s'* **and** *pred et s* **by** *blast*
moreover
from ‹*prog* ⊢ (- *l* -) −*et*→ (- *l'* -)›
have *if* (*b*) c_1 *else prog* ⊢ (- *l* -) ⊕ (#:c_1 + *1*) −*et*→ (- *l'* -) ⊕ (#:c_1 + *1*)
by(*fastforce intro*:*WCFG-CondElse*)
ultimately show *?case* **by** *simp blast*
next
case (*StepRecWhile cx c s l c' s' l' b*)
from ‹∃ *et*. *cx* ⊢ (- *l* -) −*et*→ (- *l'* -) ∧ *transfer et s* = *s'* ∧ *pred et s*›
obtain *et* **where** *cx* ⊢ (- *l* -) −*et*→ (- *l'* -)
and *transfer et s* = *s'* **and** *pred et s* **by** *blast*
moreover
hence *while* (*b*) *cx* ⊢ (- *l* -) ⊕ *2* −*et*→ (- *l'* -) ⊕ *2*
by(*fastforce intro*:*WCFG-WhileBody*)
ultimately show *?case* **by** *simp blast*
qed
with *that* **show** *?thesis* **by** *blast*
qed

4.9.2 From *c* ⊢ (- *l* -) −*et*→ (- *l'* -) with *transfers* and *preds* to *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c'*,*s'*,*l'*⟩

lemma *WCFG-edge-Exit-Skip*:
⟦*prog* ⊢ *n* −*et*→ (-*Exit*-); *n* ≠ (-*Entry*-)⟧
⟹ ∃ *l*. *n* = (- *l* -) ∧ *labels prog l Skip* ∧ *et* = ⇑*id*
proof(*induct prog n et* (-*Exit*-) *rule*:*WCFG-induct*)
case *WCFG-Skip* **show** *?case* **by**(*fastforce intro*:*Labels-Base*)
next
case *WCFG-LAssSkip* **show** *?case* **by**(*fastforce intro*:*Labels-LAss*)
next
case (*WCFG-SeqSecond* c_2 *n et n'* c_1)
note *IH* = ‹⟦*n'* = (-*Exit*-); *n* ≠ (-*Entry*-)⟧
⟹ ∃ *l*. *n* = (- *l* -) ∧ *labels* c_2 *l Skip* ∧ *et* = ⇑*id*›
from ‹*n'* ⊕ #:c_1 = (-*Exit*-)› **have** *n'* = (-*Exit*-) **by**(*cases n'*) *auto*
from *IH*[*OF this* ‹*n* ≠ (-*Entry*-)›] **obtain** *l* **where** [*simp*]:*n* = (- *l* -) *et* = ⇑*id*
and *labels* c_2 *l Skip* **by** *blast*
hence *labels* (c_1;;c_2) (*l* + #:c_1) *Skip* **by**(*fastforce intro*:*Labels-Seq2*)
thus *?case* **by**(*fastforce simp*:*id-def*)
next

case (*WCFG-CondThen* c_1 *n et n′ b* c_2)
note *IH* = ‹⟦*n′* = (-*Exit*-); *n* ≠ (-*Entry*-)⟧
⟹ ∃ *l*. *n* = (- *l* -) ∧ *labels* c_1 *l Skip* ∧ *et* = ⇑*id*›
from ‹*n′* ⊕ *1* = (-*Exit*-)› **have** *n′* = (-*Exit*-) **by**(*cases n′*) *auto*
from *IH*[*OF this* ‹*n* ≠ (-*Entry*-)›] **obtain** *l* **where** [*simp*]:*n* = (- *l* -) *et* = ⇑*id*
and *labels* c_1 *l Skip* **by** *blast*
hence *labels* (*if* (*b*) c_1 *else* c_2) (*l* + *1*) *Skip*
by(*fastforce intro*:*Labels-CondTrue*)
thus *?case* **by**(*fastforce simp*:*id-def*)
next
case (*WCFG-CondElse* c_2 *n et n′ b* c_1)
note *IH* = ‹⟦*n′* = (-*Exit*-); *n* ≠ (-*Entry*-)⟧
⟹ ∃ *l*. *n* = (- *l* -) ∧ *labels* c_2 *l Skip* ∧ *et* = ⇑*id*›
from ‹*n′* ⊕ #:c_1 + *1* = (-*Exit*-)› **have** *n′* = (-*Exit*-) **by**(*cases n′*) *auto*
from *IH*[*OF this* ‹*n* ≠ (-*Entry*-)›] **obtain** *l* **where** [*simp*]:*n* = (- *l* -) *et* = ⇑*id*
and *label*:*labels* c_2 *l Skip* **by** *blast*
hence *labels* (*if* (*b*) c_1 *else* c_2) (*l* + #:c_1 + *1*) *Skip*
by(*fastforce intro*:*Labels-CondFalse*)
thus *?case* **by**(*fastforce simp*:*add.assoc id-def*)
next
case *WCFG-WhileFalseSkip* **show** *?case* **by**(*fastforce intro*:*Labels-WhileExit*)
next
case (*WCFG-WhileBody c′ n et n′ b*) **thus** *?case* **by**(*cases n′*) *auto*
qed *simp-all*

lemma *WCFG-edge-step*:
⟦*prog* ⊢ (- *l* -) −*et*→ (- *l′* -); *transfer et s* = *s′*; *pred et s*⟧
⟹ ∃ *c c′*. *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c′*,*s′*,*l′*⟩ ∧ *labels prog l c* ∧ *labels prog l′ c′*
proof(*induct prog* (- *l* -) *et* (- *l′* -) *arbitrary*:*l l′ rule*:*WCFG-induct*)
case (*WCFG-LAss V e*)
from ‹*transfer* ⇑λ*s*. *s*(*V*:=(*interpret e s*)) *s* = *s′*›
have [*simp*]:*s′* = *s*(*V*:=(*interpret e s*)) **by**(*simp del*:*fun-upd-apply*)
have *labels* (*V*:=*e*) *0* (*V*:=*e*) **by**(*fastforce intro*:*Labels-Base*)
moreover
hence *labels* (*V*:=*e*) *1 Skip* **by**(*fastforce intro*:*Labels-LAss*)
ultimately show *?case*
apply(*rule-tac x*=*V*:=*e* **in** *exI*)
apply(*rule-tac x*=*Skip* **in** *exI*)
by(*fastforce intro*:*StepLAss simp del*:*fun-upd-apply*)
next
case (*WCFG-SeqFirst* c_1 *et* c_2)
note *IH* = ‹⟦*transfer et s* = *s′*; *pred et s*⟧
⟹ ∃ *c c′*. c_1 ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c′*,*s′*,*l′*⟩ ∧ *labels* c_1 *l c* ∧ *labels* c_1 *l′ c′*›
from *IH*[*OF* ‹*transfer et s* = *s′*› ‹*pred et s*›]
obtain *c c′* **where** c_1 ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c′*,*s′*,*l′*⟩
and *labels* c_1 *l c* **and** *labels* c_1 *l′ c′* **by** *blast*
from ‹c_1 ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c′*,*s′*,*l′*⟩› **have** c_1;;c_2 ⊢ ⟨*c*;;c_2,*s*,*l*⟩ ↝ ⟨*c′*;;c_2,*s′*,*l′*⟩
by(*rule StepRecSeq1*)

moreover
from ‹*labels* c_1 *l c*› **have** *labels* (c_1;;c_2) *l* (*c*;;c_2)
by(*fastforce intro*:*Labels-Seq1*)
moreover
from ‹*labels* c_1 *l′ c′*› **have** *labels* (c_1;;c_2) *l′* (*c′*;;c_2)
by(*fastforce intro*:*Labels-Seq1*)
ultimately show *?case* **by** *blast*
next
case (*WCFG-SeqConnect* c_1 *et* c_2)
from ‹c_1 ⊢ (- *l* -) −*et*→ (-*Exit*-)›
have *labels* c_1 *l Skip* **and** [*simp*]:*et* = ⇑*id*
by(*auto dest*:*WCFG-edge-Exit-Skip*)
from ‹*transfer et s* = *s′*› **have** [*simp*]:*s′* = *s* **by** *simp*
have *labels* c_2 *0* c_2 **by**(*fastforce intro*:*Labels-Base*)
hence *labels* (c_1;;c_2) #:c_1 c_2 **by**(*fastforce dest*:*Labels-Seq2*)
moreover
from ‹*labels* c_1 *l Skip*› **have** *labels* (c_1;;c_2) *l* (*Skip*;;c_2)
by(*fastforce intro*:*Labels-Seq1*)
moreover
from ‹*labels* c_1 *l Skip*› **have** *l* < #:c_1 **by**(*rule label-less-num-inner-nodes*)
ultimately
have c_1;;c_2 ⊢ ⟨*Skip*;;c_2,*s*,*l*⟩ ↝ ⟨c_2,*s*,#:c_1⟩ **by** −(*rule StepSeq*)
with ‹*labels* (c_1;;c_2) *l* (*Skip*;;c_2)›
‹*labels* (c_1;;c_2) #:c_1 c_2› ‹(-*0*-) ⊕ #:c_1 = (- *l′* -)› **show** *?case* **by** *simp blast*
next
case (*WCFG-SeqSecond* c_2 *n et n′* c_1)
note *IH* = ‹⋀*l l′*. ⟦*n* = (- *l* -); *n′* = (- *l′* -); *transfer et s* = *s′*; *pred et s*⟧
⟹ ∃ *c c′*. c_2 ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c′*,*s′*,*l′*⟩ ∧ *labels* c_2 *l c* ∧ *labels* c_2 *l′ c′*›
from ‹*n* ⊕ #:c_1 = (- *l* -)› **obtain** *lx* **where** *n* = (- *lx* -)
and [*simp*]:*l* = *lx* + #:c_1
by(*cases n*) *auto*
from ‹*n′* ⊕ #:c_1 = (- *l′* -)› **obtain** *lx′* **where** *n′* = (- *lx′* -)
and [*simp*]:*l′* = *lx′* + #:c_1
by(*cases n′*) *auto*
from *IH*[*OF* ‹*n* = (- *lx* -)› ‹*n′* = (- *lx′* -)› ‹*transfer et s* = *s′*› ‹*pred et s*›]
obtain *c c′* **where** c_2 ⊢ ⟨*c*,*s*,*lx*⟩ ↝ ⟨*c′*,*s′*,*lx′*⟩
and *labels* c_2 *lx c* **and** *labels* c_2 *lx′ c′* **by** *blast*
from ‹c_2 ⊢ ⟨*c*,*s*,*lx*⟩ ↝ ⟨*c′*,*s′*,*lx′*⟩› **have** c_1;;c_2 ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c′*,*s′*,*l′*⟩
by(*fastforce intro*:*StepRecSeq2*)
moreover
from ‹*labels* c_2 *lx c*› **have** *labels* (c_1;;c_2) *l c* **by**(*fastforce intro*:*Labels-Seq2*)
moreover
from ‹*labels* c_2 *lx′ c′*› **have** *labels* (c_1;;c_2) *l′ c′* **by**(*fastforce intro*:*Labels-Seq2*)
ultimately show *?case* **by** *blast*
next
case (*WCFG-CondTrue b* c_1 c_2)
from ‹(-*0*-) ⊕ *1* = (- *l′* -)› **have** [*simp*]:*l′* = *1* **by** *simp*
from ‹*transfer* (λ*s. interpret b s* = *Some true*)$_{\surd}$ *s* = *s′*› **have** [*simp*]:*s′* = *s* **by** *simp*

have *labels* (*if* (*b*) c_1 *else* c_2) *0* (*if* (*b*) c_1 *else* c_2)
by(*fastforce intro*:*Labels-Base*)
have *labels* c_1 *0* c_1 **by**(*fastforce intro*:*Labels-Base*)
hence *labels* (*if* (*b*) c_1 *else* c_2) *1* c_1 **by**(*fastforce dest*:*Labels-CondTrue*)
from ‹*pred* (*λs. interpret b s* = *Some true*)$_{\surd}$ *s*›
have *interpret b s* = *Some true* **by** *simp*
hence *if* (*b*) c_1 *else* c_2 ⊢ ⟨*if* (*b*) c_1 *else* c_2,*s*,*0*⟩ ⇝ ⟨c_1,*s*,*1*⟩
by(*rule StepCondTrue*)
with ‹*labels* (*if* (*b*) c_1 *else* c_2) *0* (*if* (*b*) c_1 *else* c_2)›
‹*labels* (*if* (*b*) c_1 *else* c_2) *1* c_1› **show** *?case* **by** *simp blast*
next
case (*WCFG-CondFalse b* c_1 c_2)
from ‹(-*0*-) ⊕ #:c_1 + *1* = (- *l′* -)› **have** [*simp*]:*l′* = #:c_1 + *1* **by** *simp*
from ‹*transfer* (*λs. interpret b s* = *Some false*)$_{\surd}$ *s* = *s′*› **have** [*simp*]:*s′* = *s*
by *simp*
have *labels* (*if* (*b*) c_1 *else* c_2) *0* (*if* (*b*) c_1 *else* c_2)
by(*fastforce intro*:*Labels-Base*)
have *labels* c_2 *0* c_2 **by**(*fastforce intro*:*Labels-Base*)
hence *labels* (*if* (*b*) c_1 *else* c_2) (#:c_1 + *1*) c_2 **by**(*fastforce dest*:*Labels-CondFalse*)
from ‹*pred* (*λs. interpret b s* = *Some false*)$_{\surd}$ *s*›
have *interpret b s* = *Some false* **by** *simp*
hence *if* (*b*) c_1 *else* c_2 ⊢ ⟨*if* (*b*) c_1 *else* c_2,*s*,*0*⟩ ⇝ ⟨c_2,*s*,#:c_1 + *1*⟩
by(*rule StepCondFalse*)
with ‹*labels* (*if* (*b*) c_1 *else* c_2) *0* (*if* (*b*) c_1 *else* c_2)›
‹*labels* (*if* (*b*) c_1 *else* c_2) (#:c_1 + *1*) c_2› **show** *?case* **by** *simp blast*
next
case (*WCFG-CondThen* c_1 *n et n′ b* c_2)
note *IH* = ‹⋀*l l′*. ⟦*n* = (- *l* -); *n′* = (- *l′* -); *transfer et s* = *s′*; *pred et s*⟧
⟹ ∃ *c c′*. c_1 ⊢ ⟨*c*,*s*,*l*⟩ ⇝ ⟨*c′*,*s′*,*l′*⟩ ∧ *labels* c_1 *l c* ∧ *labels* c_1 *l′ c′*›
from ‹*n* ⊕ *1* = (- *l* -)› **obtain** *lx* **where** *n* = (- *lx* -) **and** [*simp*]:*l* = *lx* + *1*
by(*cases n*) *auto*
from ‹*n′* ⊕ *1* = (- *l′* -)› **obtain** *lx′* **where** *n′* = (- *lx′* -) **and** [*simp*]:*l′* = *lx′* + *1*
by(*cases n′*) *auto*
from *IH*[*OF* ‹*n* = (- *lx* -)› ‹*n′* = (- *lx′* -)› ‹*transfer et s* = *s′*› ‹*pred et s*›]
obtain *c c′* **where** c_1 ⊢ ⟨*c*,*s*,*lx*⟩ ⇝ ⟨*c′*,*s′*,*lx′*⟩
and *labels* c_1 *lx c* **and** *labels* c_1 *lx′ c′* **by** *blast*
from ‹c_1 ⊢ ⟨*c*,*s*,*lx*⟩ ⇝ ⟨*c′*,*s′*,*lx′*⟩› **have** *if* (*b*) c_1 *else* c_2 ⊢ ⟨*c*,*s*,*l*⟩ ⇝ ⟨*c′*,*s′*,*l′*⟩
by(*fastforce intro*:*StepRecCond1*)
moreover
from ‹*labels* c_1 *lx c*› **have** *labels* (*if* (*b*) c_1 *else* c_2) *l c*
by(*fastforce intro*:*Labels-CondTrue*)
moreover
from ‹*labels* c_1 *lx′ c′*› **have** *labels* (*if* (*b*) c_1 *else* c_2) *l′ c′*
by(*fastforce intro*:*Labels-CondTrue*)
ultimately show *?case* **by** *blast*
next
case (*WCFG-CondElse* c_2 *n et n′ b* c_1)
note *IH* = ‹⋀*l l′*. ⟦*n* = (- *l* -); *n′* = (- *l′* -); *transfer et s* = *s′*; *pred et s*⟧
⟹ ∃ *c c′*. c_2 ⊢ ⟨*c*,*s*,*l*⟩ ⇝ ⟨*c′*,*s′*,*l′*⟩ ∧ *labels* c_2 *l c* ∧ *labels* c_2 *l′ c′*›

from ‹$n \oplus \#{:}c_1 + 1 = (\text{- } l \text{ -})$› **obtain** *lx* **where** $n = (\text{- } lx \text{ -})$
  and [*simp*]:$l = lx + \#{:}c_1 + 1$
  by(*cases n*) *auto*
from ‹$n' \oplus \#{:}c_1 + 1 = (\text{- } l' \text{ -})$› **obtain** *lx′* **where** $n' = (\text{- } lx' \text{ -})$
  and [*simp*]:$l' = lx' + \#{:}c_1 + 1$
  by(*cases n′*) *auto*
from *IH*[*OF* ‹$n = (\text{- } lx \text{ -})$› ‹$n' = (\text{- } lx' \text{ -})$› ‹*transfer et s = s′*› ‹*pred et s*›]
obtain *c c′* **where** $c_2 \vdash \langle c,s,lx\rangle \rightsquigarrow \langle c',s',lx'\rangle$
  and *labels* c_2 *lx c* **and** *labels* c_2 *lx′ c′* **by** *blast*
from ‹$c_2 \vdash \langle c,s,lx\rangle \rightsquigarrow \langle c',s',lx'\rangle$› **have** *if* (*b*) c_1 *else* $c_2 \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle$
  by(*fastforce intro*:*StepRecCond2*)
moreover
from ‹*labels* c_2 *lx c*› **have** *labels* (*if* (*b*) c_1 *else* c_2) *l c*
  by(*fastforce intro*:*Labels-CondFalse*)
moreover
from ‹*labels* c_2 *lx′ c′*› **have** *labels* (*if* (*b*) c_1 *else* c_2) *l′ c′*
  by(*fastforce intro*:*Labels-CondFalse*)
ultimately show *?case* **by** *blast*
next
case (*WCFG-WhileTrue b cx*)
from ‹$(\text{-}0\text{-}) \oplus 2 = (\text{- } l' \text{ -})$› **have** [*simp*]:$l' = 2$ **by** *simp*
from ‹*transfer* ($\lambda s.$ *interpret b s = Some true*)$_\surd$ *s = s′*› **have** [*simp*]:*s′ = s* **by** *simp*
have *labels* (*while* (*b*) *cx*) *0* (*while* (*b*) *cx*)
  by(*fastforce intro*:*Labels-Base*)
have *labels cx 0 cx* **by**(*fastforce intro*:*Labels-Base*)
hence *labels* (*while* (*b*) *cx*) *2* (*cx*;;*while* (*b*) *cx*)
  by(*fastforce dest*:*Labels-WhileBody*)
from ‹*pred* ($\lambda s.$ *interpret b s = Some true*)$_\surd$ *s*› **have** *interpret b s = Some true* **by** *simp*
hence *while* (*b*) $cx \vdash \langle$*while* (*b*) *cx*,*s*,*0*$\rangle \rightsquigarrow \langle$*cx*;;*while* (*b*) *cx*,*s*,*2*$\rangle$
  by(*rule StepWhileTrue*)
with ‹*labels* (*while* (*b*) *cx*) *0* (*while* (*b*) *cx*)›
  ‹*labels* (*while* (*b*) *cx*) *2* (*cx*;;*while* (*b*) *cx*)› **show** *?case* **by** *simp blast*
next
case (*WCFG-WhileFalse b cx*)
from ‹*transfer* ($\lambda s.$ *interpret b s = Some false*)$_\surd$ *s = s′*› **have** [*simp*]:*s′ = s*
  by *simp*
have *labels* (*while* (*b*) *cx*) *0* (*while* (*b*) *cx*) **by**(*fastforce intro*:*Labels-Base*)
have *labels* (*while* (*b*) *cx*) *1 Skip* **by**(*fastforce intro*:*Labels-WhileExit*)
from ‹*pred* ($\lambda s.$ *interpret b s = Some false*)$_\surd$ *s*› **have** *interpret b s = Some false*
  by *simp*
hence *while* (*b*) $cx \vdash \langle$*while* (*b*) *cx*,*s*,*0*$\rangle \rightsquigarrow \langle$*Skip*,*s*,*1*$\rangle$
  by(*rule StepWhileFalse*)
with ‹*labels* (*while* (*b*) *cx*) *0* (*while* (*b*) *cx*)› ‹*labels* (*while* (*b*) *cx*) *1 Skip*›
show *?case* **by** *simp blast*
next
case (*WCFG-WhileBody cx n et n′ b*)
note *IH* = ‹$\bigwedge l\ l'.$ ⟦$n = (\text{- } l \text{ -})$; $n' = (\text{- } l' \text{ -})$; *transfer et s = s′*; *pred et s*⟧

$\Longrightarrow \exists$ *c c'*. *cx* ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c'*,*s'*,*l'*⟩ ∧ *labels cx l c* ∧ *labels cx l' c'*›
from ‹*n* ⊕ *2* = (- *l* -)› **obtain** *lx* **where** *n* = (- *lx* -) **and** [*simp*]:*l* = *lx* + *2*
by(*cases n*) *auto*
from ‹*n'* ⊕ *2* = (- *l'* -)› **obtain** *lx'* **where** *n'* = (- *lx'* -)
and [*simp*]:*l'* = *lx'* + *2* **by**(*cases n'*) *auto*
from *IH*[*OF* ‹*n* = (- *lx* -)› ‹*n'* = (- *lx'* -)› ‹*transfer et s* = *s'*› ‹*pred et s*›]
obtain *c c'* **where** *cx* ⊢ ⟨*c*,*s*,*lx*⟩ ↝ ⟨*c'*,*s'*,*lx'*⟩
and *labels cx lx c* **and** *labels cx lx' c'* **by** *blast*
hence *while* (*b*) *cx* ⊢ ⟨*c*;;*while* (*b*) *cx*,*s*,*l*⟩ ↝ ⟨*c'*;;*while* (*b*) *cx*,*s'*,*l'*⟩
by(*fastforce intro*:*StepRecWhile*)
moreover
from ‹*labels cx lx c*› **have** *labels* (*while* (*b*) *cx*) *l* (*c*;;*while* (*b*) *cx*)
by(*fastforce intro*:*Labels-WhileBody*)
moreover
from ‹*labels cx lx' c'*› **have** *labels* (*while* (*b*) *cx*) *l'* (*c'*;;*while* (*b*) *cx*)
by(*fastforce intro*:*Labels-WhileBody*)
ultimately show *?case* **by** *blast*
next
case (*WCFG-WhileBodyExit cx n et b*)
from ‹*n* ⊕ *2* = (- *l* -)› **obtain** *lx* **where** [*simp*]:*n* = (- *lx* -) **and** [*simp*]:*l* = *lx* +
2
by(*cases n*) *auto*
from ‹*cx* ⊢ *n* −*et*→ (-*Exit*-)› **have** *labels cx lx Skip* **and** [*simp*]:*et* = ⇑*id*
by(*auto dest*:*WCFG-edge-Exit-Skip*)
from ‹*transfer et s* = *s'*› **have** [*simp*]:*s'* = *s* **by** *simp*
from ‹*labels cx lx Skip*› **have** *labels* (*while* (*b*) *cx*) *l* (*Skip*;;*while* (*b*) *cx*)
by(*fastforce intro*:*Labels-WhileBody*)
hence *while* (*b*) *cx* ⊢ ⟨*Skip*;;*while* (*b*) *cx*,*s*,*l*⟩ ↝ ⟨*while* (*b*) *cx*,*s*,*0*⟩
by(*rule StepSeqWhile*)
moreover
have *labels* (*while* (*b*) *cx*) *0* (*while* (*b*) *cx*)
by(*fastforce intro*:*Labels-Base*)
ultimately show *?case*
using ‹*labels* (*while* (*b*) *cx*) *l* (*Skip*;;*while* (*b*) *cx*)› **by** *simp blast*
qed

end

4.10 Semantic well-formedness of While CFG

theory *SemanticsWellFormed*
imports *WellFormed WEquivalence ../Basic/SemanticsCFG*
begin

4.10.1 Instatiation of the *CFG-semantics-wf* locale

fun *labels-nodes* :: *cmd* ⇒ *w-node* ⇒ *cmd* ⇒ *bool* **where**
labels-nodes prog (- *l* -) *c* = *labels prog l c*

| *labels-nodes prog* (*-Entry-*) *c* = *False*
| *labels-nodes prog* (*-Exit-*) *c* = *False*

interpretation *While-semantics-CFG-wf*: *CFG-semantics-wf*
sourcenode targetnode kind valid-edge prog Entry reds labels-nodes prog
for *prog*
proof(*unfold-locales*)
fix *n c s c′ s′ n′*
assume *labels-nodes prog n c* **and** ⟨*c*,*s*⟩ →∗ ⟨*c′*,*s′*⟩
then obtain *l l′* **where** [*simp*]:*n* = (- *l* -) **and** *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c′*,*s′*,*l′*⟩
and *labels prog l′ c′* **by**(*cases n*,*auto dest*:*reds-steps*)
from ‹*labels prog l′ c′*› **have** *l′* < #:*prog* **by**(*rule label-less-num-inner-nodes*)
from ‹*prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝∗ ⟨*c′*,*s′*,*l′*⟩›
have ∃ *as*. *CFG.path sourcenode targetnode* (*valid-edge prog*)
(- *l* -) *as* (- *l′* -) ∧
transfers (*CFG.kinds kind as*) *s* = *s′* ∧ *preds* (*CFG.kinds kind as*) *s*
proof(*induct rule*:*converse-rtranclp-induct3*)
case *refl*
from ‹*l′* < #:*prog*› **have** *valid-node prog* (- *l′* -)
by(*fastforce dest*:*less-num-nodes-edge simp*:*valid-node-def valid-edge-def*)
hence *CFG.valid-node sourcenode targetnode* (*valid-edge prog*) (- *l′* -)
by(*simp add*:*valid-node-def While-CFG.valid-node-def*)
hence *CFG.path sourcenode targetnode* (*valid-edge prog*) (- *l′* -) [] (- *l′* -)
by(*rule While-CFG.empty-path*)
thus *?case* **by**(*auto simp*:*While-CFG.kinds-def*)
next
case (*step c s l c″ s″ l″*)
from ‹(λ(*c*, *s*, *l*) (*c′*, *s′*, *l′*).
prog ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c′*,*s′*,*l′*⟩) (*c*,*s*,*l*) (*c″*,*s″*,*l″*)›
have *prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c″*,*s″*,*l″*⟩ **by** *simp*
from ‹∃ *as*. *CFG.path sourcenode targetnode* (*valid-edge prog*)
(- *l″* -) *as* (- *l′* -) ∧
transfers (*CFG.kinds kind as*) *s″* = *s′* ∧
preds (*CFG.kinds kind as*) *s″*›
obtain *as* **where** *CFG.path sourcenode targetnode* (*valid-edge prog*)
(- *l″* -) *as* (- *l′* -)
and *transfers* (*CFG.kinds kind as*) *s″* = *s′*
and *preds* (*CFG.kinds kind as*) *s″* **by** *auto*
from ‹*prog* ⊢ ⟨*c*,*s*,*l*⟩ ↝ ⟨*c″*,*s″*,*l″*⟩› **obtain** *et*
where *prog* ⊢ (- *l* -) −*et*→ (- *l″* -)
and *transfer et s* = *s″* **and** *pred et s*
by(*erule step-WCFG-edge*)
from ‹*prog* ⊢ (- *l* -) −*et*→ (- *l″* -)›
‹*CFG.path sourcenode targetnode* (*valid-edge prog*) (- *l″* -) *as* (- *l′* -)›
have *CFG.path sourcenode targetnode* (*valid-edge prog*)
(- *l* -) (((- *l* -),*et*,(- *l″* -))#*as*) (- *l′* -)
by(*fastforce intro*:*While-CFG.Cons-path simp*:*valid-edge-def*)
moreover

from ‹*transfers* (*CFG.kinds kind as*) *s''* = *s'*› ‹*transfer et s* = *s''*›
have *transfers* (*CFG.kinds kind* (((- *l* -),*et*,(- *l''* -))#*as*)) *s* = *s'*
by(*simp add*: *While-CFG.kinds-def*)
moreover from ‹*preds* (*CFG.kinds kind as*) *s''*› ‹*pred et s*› ‹*transfer et s* = *s''*›
have *preds* (*CFG.kinds kind* (((- *l* -),*et*,(- *l''* -))#*as*)) *s*
by(*simp add*: *While-CFG.kinds-def*)
ultimately show *?case* **by** *blast*
qed
with ‹*labels prog l' c'*›
show (∃ *n' as*.
CFG.path sourcenode targetnode (*valid-edge prog*) *n as n'* ∧
transfers (*CFG.kinds kind as*) *s* = *s'* ∧
preds (*CFG.kinds kind as*) *s* ∧ *labels-nodes prog n' c'*)
by(*rule-tac x*=(- *l'* -) **in** *exI*,*simp*)
qed

end

4.11 Interpretations of the various static control dependences

theory *StaticControlDependences* **imports**
AdditionalLemmas
SemanticsWellFormed
begin

lemma *WhilePostdomination-aux*:
Postdomination sourcenode targetnode kind (*valid-edge prog*) *Entry Exit*
proof(*unfold-locales*)
fix *n* **assume** *CFG.valid-node sourcenode targetnode* (*valid-edge prog*) *n*
hence *valid-node prog n* **by**(*simp add*:*valid-node-def While-CFG.valid-node-def*)
thus ∃ *as*. *prog* ⊢ (-*Entry*-) −*as*→∗ *n* **by**(*rule valid-node-Entry-path*)
next
fix *n* **assume** *CFG.valid-node sourcenode targetnode* (*valid-edge prog*) *n*
hence *valid-node prog n* **by**(*simp add*:*valid-node-def While-CFG.valid-node-def*)
thus ∃ *as*. *prog* ⊢ *n* −*as*→∗ (-*Exit*-) **by**(*rule valid-node-Exit-path*)
qed

interpretation *WhilePostdomination*:
Postdomination sourcenode targetnode kind valid-edge prog Entry Exit
by(*rule WhilePostdomination-aux*)

lemma *WhileStrongPostdomination-aux*:
StrongPostdomination sourcenode targetnode kind (*valid-edge prog*) *Entry Exit*
proof(*unfold-locales*)
fix *n* **assume** *CFG.valid-node sourcenode targetnode* (*valid-edge prog*) *n*

hence *valid-node prog n* **by**(*simp add:valid-node-def While-CFG.valid-node-def*)
show *finite* $\{n'. \exists a'.$ *valid-edge prog* $a' \wedge$ *sourcenode* $a' = n \wedge$
targetnode $a' = n'\}$
by(*rule finite-successors*)
qed

interpretation *WhileStrongPostdomination*:
StrongPostdomination sourcenode targetnode kind valid-edge prog Entry Exit
by(*rule WhileStrongPostdomination-aux*)

4.11.1 Standard Control Dependence

lemma *WStandardControlDependence-aux*:
StandardControlDependencePDG sourcenode targetnode kind (*valid-edge prog*)
Entry (*Defs prog*) (*Uses prog*) *id Exit*
by(*unfold-locales*)

interpretation *WStandardControlDependence*:
StandardControlDependencePDG sourcenode targetnode kind valid-edge prog
Entry Defs prog Uses prog id Exit
by(*rule WStandardControlDependence-aux*)

lemma *Fundamental-property-scd-aux*: *BackwardSlice-wf sourcenode targetnode kind*

(*valid-edge prog*) *Entry* (*Defs prog*) (*Uses prog*) *id*
(*WStandardControlDependence.PDG-BS-s prog*) *reds* (*labels-nodes prog*)
proof −
interpret *BackwardSlice sourcenode targetnode kind valid-edge prog Entry*
Defs prog Uses prog id
StandardControlDependencePDG.PDG-BS-s sourcenode targetnode
(*valid-edge prog*) (*Defs prog*) (*Uses prog*) *Exit*
by(*rule WStandardControlDependence.PDGBackwardSliceCorrect*)
show *?thesis* **by**(*unfold-locales*)
qed

interpretation *Fundamental-property-scd*: *BackwardSlice-wf sourcenode targetnode kind*
valid-edge prog Entry Defs prog Uses prog id
WStandardControlDependence.PDG-BS-s prog reds labels-nodes prog
by(*rule Fundamental-property-scd-aux*)

4.11.2 Weak Control Dependence

lemma *WWeakControlDependence-aux*:
WeakControlDependencePDG sourcenode targetnode kind (*valid-edge prog*)
Entry (*Defs prog*) (*Uses prog*) *id Exit*
by(*unfold-locales*)

interpretation *WWeakControlDependence*:

WeakControlDependencePDG sourcenode targetnode kind valid-edge prog
Entry Defs prog Uses prog id Exit
by(*rule WWeakControlDependence-aux*)

lemma *Fundamental-property-wcd-aux*: *BackwardSlice-wf sourcenode targetnode kind*

(*valid-edge prog*) *Entry* (*Defs prog*) (*Uses prog*) *id*
(*WWeakControlDependence.PDG-BS-w prog*) *reds* (*labels-nodes prog*)
proof −
interpret *BackwardSlice sourcenode targetnode kind valid-edge prog Entry*
Defs prog Uses prog id
WeakControlDependencePDG.PDG-BS-w sourcenode targetnode
(*valid-edge prog*) (*Defs prog*) (*Uses prog*) *Exit*
by(*rule WWeakControlDependence.WeakPDGBackwardSliceCorrect*)
show *?thesis* **by**(*unfold-locales*)
qed

interpretation *Fundamental-property-wcd*: *BackwardSlice-wf sourcenode targetnode kind*
valid-edge prog Entry Defs prog Uses prog id
WWeakControlDependence.PDG-BS-w prog reds labels-nodes prog
by(*rule Fundamental-property-wcd-aux*)

4.11.3 Weak Order Dependence

lemma *Fundamental-property-wod-aux*: *BackwardSlice-wf sourcenode targetnode kind*

(*valid-edge prog*) *Entry* (*Defs prog*) (*Uses prog*) *id*
(*While-CFG-wf.wod-backward-slice prog*) *reds* (*labels-nodes prog*)
proof −
interpret *BackwardSlice sourcenode targetnode kind valid-edge prog Entry*
Defs prog Uses prog id
CFG-wf.wod-backward-slice sourcenode targetnode (*valid-edge prog*)
(*Defs prog*) (*Uses prog*)
by(*rule While-CFG-wf.WODBackwardSliceCorrect*)
show *?thesis* **by**(*unfold-locales*)
qed

interpretation *Fundamental-property-wod*: *BackwardSlice-wf sourcenode targetnode kind*
valid-edge prog Entry Defs prog Uses prog id
While-CFG-wf.wod-backward-slice prog reds labels-nodes prog
by(*rule Fundamental-property-wod-aux*)

end

Chapter 5

A Control Flow Graph for Jinja Byte Code

5.1 Formalizing the CFG

theory *JVMCFG* **imports** *../Basic/BasicDefs Jinja.BVExample* **begin**

declare *lesub-list-impl-same-size* [*simp del*]
declare *nlistsE-length* [*simp del*]

5.1.1 Type definitions

Wellformed Programs

definition *wf-jvmprog* = {(*P*, *Phi*). $\textit{wf-jvm-prog}_{Phi}$ *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 *rep-jvmprog-jvm-prog* :: *wf-jvmprog* ⇒ *jvm-prog*
(‹-_{wf}›)
  where P_{wf} ≡ *fst*(*Rep-wf-jvmprog*(*P*))

abbreviation *rep-jvmprog-phi* :: *wf-jvmprog* ⇒ ty_P
(‹-_{Φ}›)
  where P_{Φ} ≡ *snd*(*Rep-wf-jvmprog*(*P*))

lemma *wf-jvmprog-is-wf*: $\textit{wf-jvm-prog}_{P_{\Phi}}$ (P_{wf})
using *Rep-wf-jvmprog* [*of P*]

by (*auto simp*: *wf-jvmprog-def split-beta*)

Basic Types

We consider a program to be a well-formed Jinja program, together with a given base class and a main method

type-synonym *jvmprog* = *wf-jvmprog* × *cname* × *mname*
type-synonym *callstack* = (*cname* × *mname* × *pc*) *list*

The state is modeled as heap × stack-variables × local-variables

stack and local variables are modeled as pairs of natural numbers. The first number gives the position in the call stack (i.e. the method in which the variable is used), the second the position in the method's stack or array of local variables resp.

The stack variables are numbered from bottom up (which is the reverse order of the array for the stack in Jinja's state representation), whereas local variables are identified by their position in the array of local variables of Jinja's state representation.

type-synonym *state* = *heap* × ((*nat* × *nat*) ⇒ *val*) × ((*nat* × *nat*) ⇒ *val*)

abbreviation *heap-of* :: *state* ⇒ *heap*
where
heap-of s ≡ *fst*(*s*)

abbreviation *stk-of* :: *state* ⇒ ((*nat* × *nat*) ⇒ *val*)
where
stk-of s ≡ *fst*(*snd*(*s*))

abbreviation *loc-of* :: *state* ⇒ ((*nat* × *nat*) ⇒ *val*)
where
loc-of s ≡ *snd*(*snd*(*s*))

5.1.2 Basic Definitions

State update (instruction execution)

This function models instruction execution for our state representation.

Additional parameters are the call depth of the current program point, the stack length of the current program point, the length of the stack in the underlying call frame (needed for Return), and (for Invoke) the length of the array of local variables of the invoked method.

Exception handling is not covered by this function.

fun *exec-instr* :: *instr* ⇒ *wf-jvmprog* ⇒ *state* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *state*
where
exec-instr-Load:

exec-instr (*Load n*) *P s calldepth stk-length rs ill* =
(*let* (*h*,*stk*,*loc*) = *s*
in (*h*, *stk*((*calldepth*,*stk-length*):=*loc*(*calldepth*,*n*)), *loc*))

| *exec-instr-Store*:
exec-instr (*Store n*) *P s calldepth stk-length rs ill* =
(*let* (*h*,*stk*,*loc*) = *s*
in (*h*, *stk*, *loc*((*calldepth*,*n*):=*stk*(*calldepth*,*stk-length* − *1*))))

| *exec-instr-Push*:
exec-instr (*Push v*) *P s calldepth stk-length rs ill* =
(*let* (*h*,*stk*,*loc*) = *s*
in (*h*, *stk*((*calldepth*,*stk-length*):=*v*), *loc*))

| *exec-instr-New*:
exec-instr (*New C*) *P s calldepth stk-length rs ill* =
(*let* (*h*,*stk*,*loc*) = *s*;
a = *the*(*new-Addr h*)
in (*h*(*a* ↦ (*blank* (P_{wf}) *C*)), *stk*((*calldepth*,*stk-length*):=*Addr a*), *loc*))

| *exec-instr-Getfield*:
exec-instr (*Getfield F C*) *P s calldepth stk-length rs ill* =
(*let* (*h*,*stk*,*loc*) = *s*;
a = *the-Addr* (*stk* (*calldepth*,*stk-length* − *1*));
(*D*,*fs*) = *the*(*h a*)
in (*h*, *stk*((*calldepth*,*stk-length* − *1*) := *the*(*fs*(*F*,*C*))), *loc*))

| *exec-instr-Putfield*:
exec-instr (*Putfield F C*) *P s calldepth stk-length rs ill* =
(*let* (*h*,*stk*,*loc*) = *s*;
v = *stk* (*calldepth*,*stk-length* − *1*);
a = *the-Addr* (*stk* (*calldepth*,*stk-length* − *2*));
(*D*,*fs*) = *the*(*h a*)
in (*h*(*a* ↦ (*D*,*fs*((*F*,*C*) ↦ *v*))), *stk*, *loc*))

| *exec-instr-Checkcast*:
exec-instr (*Checkcast C*) *P s calldepth stk-length rs ill* = *s*

| *exec-instr-Pop*:
exec-instr (*Pop*) *P s calldepth stk-length rs ill* = *s*

| *exec-instr-IAdd*:
exec-instr (*IAdd*) *P s calldepth stk-length rs ill* =
(*let* (*h*,*stk*,*loc*) = *s*;
i_1 = *the-Intg* (*stk* (*calldepth*, *stk-length* − *1*));
i_2 = *the-Intg* (*stk* (*calldepth*, *stk-length* − *2*))
in (*h*, *stk*((*calldepth*, *stk-length* − *2*) := *Intg* ($i_1 + i_2$)), *loc*))

| *exec-instr-IfFalse*:

exec-instr (*IfFalse b*) *P s calldepth stk-length rs ill* = *s*

| *exec-instr-CmpEq*:
exec-instr (*CmpEq*) *P s calldepth stk-length rs ill* =
(*let* (*h*,*stk*,*loc*) = *s*;
v_1 = *stk* (*calldepth*, *stk-length* − *1*);
v_2 = *stk* (*calldepth*, *stk-length* − *2*)
in (*h*, *stk*((*calldepth*, *stk-length* − *2*) := *Bool* (v_1 = v_2)), *loc*))

| *exec-instr-Goto*:
exec-instr (*Goto i*) *P s calldepth stk-length rs ill* = *s*

| *exec-instr-Throw*:
exec-instr (*Throw*) *P s calldepth stk-length rs ill* = *s*

| *exec-instr-Invoke*:
exec-instr (*Invoke M n*) *P s calldepth stk-length rs invoke-loc-length* =
(*let* (*h*,*stk*,*loc*) = *s*;
loc' = (λ(*a*,*b*). *if* (*a* ≠ *Suc calldepth* ∨ *b* ≥ *invoke-loc-length*) *then loc*(*a*,*b*) *else*
(*if* (*b* ≤ *n*) *then stk*(*calldepth*, *stk-length* − (*Suc n* − *b*)) *else*
arbitrary))
in (*h*,*stk*,*loc'*))

| *exec-instr-Return*:
exec-instr (*Return*) *P s calldepth stk-length ret-stk-length ill* =
(*if* (*calldepth* = *0*)
then s
else
(*let* (*h*,*stk*,*loc*) = *s*;
v = *stk*(*calldepth*, *stk-length* − *1*)
in (*h*,*stk*((*calldepth* − *1*, *ret-stk-length* − *1*) := *v*),*loc*))
)

length of stack and local variables

The following terms extract the stack length at a given program point from the well-typing of the given program

abbreviation *stkLength* :: *wf-jvmprog* ⇒ *cname* ⇒ *mname* ⇒ *pc* ⇒ *nat*
where
stkLength P C M pc ≡ *length* (*fst*(*the*(((P_Φ) *C M*)!*pc*)))

abbreviation *locLength* :: *wf-jvmprog* ⇒ *cname* ⇒ *mname* ⇒ *pc* ⇒ *nat*
where
locLength P C M pc ≡ *length* (*snd*(*the*(((P_Φ) *C M*)!*pc*)))

Conversion functions

This function takes a natural number n and a function f with domain *nat* and creates the array [f 0, f 1, f 2, ..., f (n - 1)].

This is used for extracting the array of local variables

abbreviation $locs :: nat \Rightarrow (nat \Rightarrow 'a) \Rightarrow 'a\ list$
where $locs\ n\ loc \equiv map\ loc\ [0..<n]$

This function takes a natural number n and a function f with domain *nat* and creates the array [f (n - 1), ..., f 1, f 0].

This is used for extracting the stack as a list

abbreviation $stks :: nat \Rightarrow (nat \Rightarrow 'a) \Rightarrow 'a\ list$
where $stks\ n\ stk \equiv map\ stk\ (rev\ [0..<n])$

This function creates a list of the arrays for local variables from the given state corresponding to the given callstack

fun $locss :: wf\text{-}jvmprog \Rightarrow callstack \Rightarrow ((nat \times nat) \Rightarrow 'a) \Rightarrow 'a\ list\ list$
where
$locss\ P\ []\ loc = []$
$|\ locss\ P\ ((C,M,pc)\#cs)\ loc =$
$(locs\ (locLength\ P\ C\ M\ pc)\ (\lambda a.\ loc\ (length\ cs,\ a)))\#(locss\ P\ cs\ loc)$

This function creates a list of the (methods') stacks from the given state corresponding to the given callstack

fun $stkss :: wf\text{-}jvmprog \Rightarrow callstack \Rightarrow ((nat \times nat) \Rightarrow 'a) \Rightarrow 'a\ list\ list$
where
$stkss\ P\ []\ stk = []$
$|\ stkss\ P\ ((C,M,pc)\#cs)\ stk =$
$(stks\ (stkLength\ P\ C\ M\ pc)\ (\lambda a.\ stk\ (length\ cs,\ a)))\#(stkss\ P\ cs\ stk)$

Given a callstack and a state, this abbreviation converts the state to Jinja's state representation

abbreviation $state\text{-}to\text{-}jvm\text{-}state :: wf\text{-}jvmprog \Rightarrow callstack \Rightarrow state \Rightarrow jvm\text{-}state$
where $state\text{-}to\text{-}jvm\text{-}state\ P\ cs\ s \equiv$
$(None,\ heap\text{-}of\ s,\ zip\ (stkss\ P\ cs\ (stk\text{-}of\ s))\ (zip\ (locss\ P\ cs\ (loc\text{-}of\ s))\ cs))$

This function extracts the call stack from a given frame stack (as it is given by Jinja's state representation)

definition $framestack\text{-}to\text{-}callstack :: frame\ list \Rightarrow callstack$
where $framestack\text{-}to\text{-}callstack\ frs \equiv map\ snd\ (map\ snd\ frs)$

State Conformance

Now we lift byte code verifier conformance to our state representation

definition $bv\text{-}conform :: wf\text{-}jvmprog \Rightarrow callstack \Rightarrow state \Rightarrow bool$
$(\langle\text{-},\text{-} \vdash_{BV} \text{-} \surd\rangle)$
where $P,cs \vdash_{BV} s \surd \equiv correct\text{-}state\ (P_{wf})\ (P_{\Phi})\ (state\text{-}to\text{-}jvm\text{-}state\ P\ cs\ s)$

Statically determine catch-block

This function is equivalent to Jinja's *find-handler* function

fun *find-handler-for* :: *wf-jvmprog* ⇒ *cname* ⇒ *callstack* ⇒ *callstack*
where
  find-handler-for P C [] = []
| *find-handler-for P C* (*c*#*cs*) = (*let* (*C′*,*M′*,*pc′*) = *c in*
    (*case match-ex-table* (P_{wf}) *C pc′* (*ex-table-of* (P_{wf}) *C′ M′*) *of*
      None ⇒ *find-handler-for P C cs*
    | *Some pc-d* ⇒ (*C′*, *M′*, *fst pc-d*)#*cs*))

5.1.3 Simplification lemmas

lemma *find-handler-decr* [*simp*]: *find-handler-for P Exc cs* ≠ *c*#*cs*
proof
  assume *find-handler-for P Exc cs* = *c*#*cs*
  hence *length cs* < *length* (*find-handler-for P Exc cs*) **by** *simp*
  thus *False* **by** (*induct cs*, *auto*)
qed

lemma *stkss-length* [*simp*]: *length* (*stkss P cs stk*) = *length cs*
  by (*induct cs*) *auto*

lemma *locss-length* [*simp*]: *length* (*locss P cs loc*) = *length cs*
  by (*induct cs*) *auto*

lemma *nth-stkss*:
  ⟦ *a* < *length cs*; *b* < *length* (*stkss P cs stk* ! (*length cs* − *Suc a*)) ⟧
  ⟹ *stkss P cs stk* ! (*length cs* − *Suc a*) !
    (*length* (*stkss P cs stk* ! (*length cs* − *Suc a*)) − *Suc b*) = *stk* (*a*,*b*)
proof (*induct cs*)
  case *Nil*
  thus *?case* **by** (*simp add*: *nth-Cons′*)
next
  case (*Cons aa cs*)
  thus *?case*
    by (*cases aa*, *auto simp add*: *nth-Cons′ rev-nth less-Suc-eq*)
qed

lemma *nth-locss*:
  ⟦ *a* < *length cs*; *b* < *length* (*locss P cs loc* ! (*length cs* − *Suc a*)) ⟧
  ⟹ *locss P cs loc* ! (*length cs* − *Suc a*) ! *b* = *loc* (*a*,*b*)
proof (*induct cs*)

case *Nil*
thus *?case* **by** (*simp add*: *nth-Cons′*)
next
case (*Cons aa cs*)
thus *?case*
by (*cases aa*, *auto simp*: *nth-Cons′ less-Suc-eq*)
qed

lemma *hd-stks* [*simp*]: $n \neq 0 \Longrightarrow hd\ (stks\ n\ stk) = stk(n - 1)$
by (*cases n*, *simp-all*)

lemma *hd-tl-stks*: $n > 1 \Longrightarrow hd\ (tl\ (stks\ n\ stk)) = stk(n - 2)$
by (*cases n*, *auto*)

lemma *stkss-purge*:
$length\ cs \leq a \Longrightarrow stkss\ P\ cs\ (stk((a,b) := c)) = stkss\ P\ cs\ stk$
by (*induct cs*, *auto*)

lemma *stkss-purge′*:
$length\ cs \leq a \Longrightarrow stkss\ P\ cs\ (\lambda s.\ if\ s = (a,\ b)\ then\ c\ else\ stk\ s) = stkss\ P\ cs\ stk$
by (*fold fun-upd-def*, *simp only*: *stkss-purge*)

lemma *locss-purge*:
$length\ cs \leq a \Longrightarrow locss\ P\ cs\ (loc((a,b) := c)) = locss\ P\ cs\ loc$
by (*induct cs*, *auto*)

lemma *locss-purge′*:
$length\ cs \leq a \Longrightarrow locss\ P\ cs\ (\lambda s.\ if\ s = (a,\ b)\ then\ c\ else\ loc\ s) = locss\ P\ cs\ loc$
by (*fold fun-upd-def*, *simp only*: *locss-purge*)

lemma *locs-pullout* [*simp*]:
$locs\ b\ (loc(n := e)) = (locs\ b\ loc)\ [n := e]$
proof (*induct b*)
case *0*
thus *?case* **by** *simp*
next
case (*Suc b*)
thus *?case*
by (*cases n − b*, *auto simp*: *list-update-append not-less-eq less-Suc-eq*)
qed

lemma *locs-pullout′* [*simp*]:
$locs\ b\ (\lambda a.\ if\ a = n\ then\ e\ else\ loc\ (c,\ a)) = (locs\ b\ (\lambda a.\ loc\ (c,\ a)))\ [n := e]$
by (*fold fun-upd-def*) *simp*

lemma *stks-pullout*:
  $n < b \Longrightarrow$ *stks b* (*stk*(n := e)) = (*stks b stk*) [$b - Suc\ n$:= e]
proof (*induct b*)
  case *0*
  thus *?case* **by** *simp*
next
  case (*Suc b*)
  thus *?case*
  proof (*cases* $b = n$)
    case *True*
    with *Suc* **show** *?thesis*
      by *auto*

  next
    case *False*
    with *Suc* **show** *?thesis*
      by (*cases* $b - n$) (*auto intro*!: *nth-equalityI simp*: *nth-list-update*)
 qed
qed

lemma *nth-tl* : *xs* ≠ [] ⟹ *tl xs* ! *n* = *xs* ! (*Suc n*)
  by (*cases xs*, *simp-all*)

lemma *f2c-Nil* [*simp*]: *framestack-to-callstack* [] = []
  by (*simp add*: *framestack-to-callstack-def*)

lemma *f2c-Cons* [*simp*]:
 framestack-to-callstack ((*stk*,*loc*,*C*,*M*,*pc*)#*frs*) = (*C*,*M*,*pc*)#(*framestack-to-callstack frs*)
  by (*simp add*: *framestack-to-callstack-def*)

lemma *f2c-length* [*simp*]:
  length (*framestack-to-callstack frs*) = *length frs*
  by (*simp add*: *framestack-to-callstack-def*)

lemma *f2c-s2jvm-id* [*simp*]:
  framestack-to-callstack
    (*snd*(*snd*(*state-to-jvm-state P cs s*))) =
  cs
  by (*cases s*, *simp add*: *framestack-to-callstack-def*)

lemma *f2c-s2jvm-id′* [*simp*]:
  framestack-to-callstack
  (*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) = *cs*
  by (*simp add*: *framestack-to-callstack-def*)

lemma *f2c-append* [*simp*]:
  framestack-to-callstack (*frs* @ *frs′*) =
  (*framestack-to-callstack frs*) @ (*framestack-to-callstack frs′*)

by (*simp add*: *framestack-to-callstack-def*)

5.1.4 CFG construction

5.1.5 Datatypes

Nodes are labeled with a callstack and an optional tuple (consisting of a callstack and a flag).

The first callstack determines the current program point (i.e. the next statement to execute). If the second parameter is not None, we are at an intermediate state, where the target of the instruction is determined (the second callstack) and the flag is set to whether an exception is thrown or not.

datatype *j-node* =
  Entry (‹'('-Entry'-')›)
| *Node callstack* (*callstack* × *bool*) *option* (‹'('- -,- '-')›)

The empty callstack indicates the exit node

abbreviation *j-node-Exit* :: *j-node* (‹'('-Exit'-')›)
where *j-node-Exit* ≡ (- [],*None* -)

An edge is a triple, consisting of two nodes and the edge kind

type-synonym *j-edge* = (*j-node* × *state edge-kind* × *j-node*)

5.1.6 CFG

The CFG is constructed by a case analysis on the instructions and their different behavior in different states. E.g. the exceptional behavior of New, if there is no more space in the heap, vs. the normal behavior.

Note: The set of edges defined by this predicate is a first approximation to the real set of edges in the CFG. We later (theory JVMInterpretation) add some well-formedness requirements to the nodes.

inductive *JVM-CFG* :: *jvmprog* ⇒ *j-node* ⇒ *state edge-kind* ⇒ *j-node* ⇒ *bool*
 (‹- ⊢ - −-→ -›)
where
 JCFG-EntryExit:
 prog ⊢ (-*Entry*-) −(λ*s*. *False*)$_{\surd}$→ (-*Exit*-)

| *JCFG-EntryStart*:
 prog = (*P*, *C0*, *Main*) ⟹ *prog* ⊢ (-*Entry*-) −(λ*s*. *True*)$_{\surd}$→ (- [(*C0*, *Main*, *0*)],*None* -)

| *JCFG-ReturnExit*:
 ⟦ *prog* = (*P*,*C0*,*Main*);
  (*instrs-of* (P_{wf}) *C M*) ! *pc* = *Return* ⟧
  ⟹ *prog* ⊢ (- [(*C*, *M*, *pc*)],*None* -) −⇑*id*→ (-*Exit*-)

| *JCFG-Straight-NoExc*:
⟦ *prog* = (*P*, *C0*, *Main*);
instrs-of (P_{wf}) *C M* ! *pc* ∈ {*Load idx*, *Store idx*, *Push val*, *Pop*, *IAdd*, *CmpEq*};
ek = ⇑(λ*s*. *exec-instr* ((*instrs-of* (P_{wf}) *C M*) ! *pc*) *P s*
(*length cs*) (*stkLength P C M pc*) *arbitrary arbitrary*) ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *Suc pc*)#*cs*,*None* -)

| *JCFG-New-Normal-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*New Cl*);
ek = (λ(*h*,*stk*,*loc*). *new-Addr h* ≠ *None*)$_{\surd}$⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*, *M*, *Suc pc*)#*cs*,*False*)⌋ -)

| *JCFG-New-Normal-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*New Cl*);
ek = ⇑(λ*s*. *exec-instr* (*New Cl*) *P s* (*length cs*) (*stkLength P C M pc*) *arbitrary arbitrary*) ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*, *M*, *Suc pc*)#*cs*, *False*)⌋ -) −*ek*→ (- (*C*, *M*, *Suc pc*)#*cs*,*None* -)

| *JCFG-New-Exc-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*New Cl*);
find-handler-for P OutOfMemory ((*C*, *M*, *pc*)#*cs*) = *cs*′;
ek = (λ(*h*,*stk*,*loc*). *new-Addr h* = *None*)$_{\surd}$ ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊(*cs*′,*True*)⌋ -)

| *JCFG-New-Exc-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*New Cl*);
find-handler-for P OutOfMemory ((*C*, *M*, *pc*)#*cs*) = (*C*′, *M*′, *pc*′)#*cs*′;
ek = ⇑(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs*′,(*stkLength P C*′ *M*′ *pc*′) − *1*) := *Addr* (*addr-of-sys-xcpt OutOfMemory*)),
loc)
) ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*′, *M*′, *pc*′)#*cs*′, *True*)⌋ -) −*ek*→ (- (*C*′, *M*′, *pc*′)#*cs*′,*None* -)

| *JCFG-New-Exc-Exit*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*New Cl*);
find-handler-for P OutOfMemory ((*C*, *M*, *pc*)#*cs*) = [] ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊([], *True*)⌋ -) −⇑*id*→ (-*Exit*-)

| *JCFG-Getfield-Normal-Pred*:

⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Getfield Fd Cl*);
ek = (λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *1*) ≠ *Null*)$_{\surd}$ ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*, *M*, *Suc pc*)#*cs*, *False*)⌋ -)

| *JCFG-Getfield-Normal-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Getfield Fd Cl*);
ek = ⇑(λ*s*. *exec-instr* (*Getfield Fd Cl*) *P s* (*length cs*) (*stkLength P C M pc*) *arbitrary arbitrary*) ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*, *M*, *Suc pc*)#*cs*, *False*)⌋ -) −*ek*→ (- (*C*, *M*, *Suc pc*)#*cs*,*None* -)

| *JCFG-Getfield-Exc-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Getfield Fd Cl*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = *cs*′;
ek = (λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null*)$_{\surd}$ ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊(*cs*′, *True*)⌋ -)

| *JCFG-Getfield-Exc-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Getfield Fd Cl*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = (*C*′, *M*′, *pc*′)#*cs*′;
ek = ⇑(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs*′,(*stkLength P C*′ *M*′ *pc*′) − *1*) := *Addr* (*addr-of-sys-xcpt NullPointer*)),
loc)
) ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*′, *M*′, *pc*′)#*cs*′, *True*)⌋ -) −*ek*→ (- (*C*′, *M*′, *pc*′)#*cs*′,*None* -)

| *JCFG-Getfield-Exc-Exit*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Getfield Fd Cl*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = [] ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊([], *True*)⌋ -) −⇑*id*→ (-*Exit*-)

| *JCFG-Putfield-Normal-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Putfield Fd Cl*);
ek = (λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *2*) ≠ *Null*)$_{\surd}$ ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*, *M*, *Suc pc*)#*cs*, *False*)⌋ -)

| *JCFG-Putfield-Normal-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);

(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Putfield Fd Cl*);
ek = ⇑(λ*s*. *exec-instr* (*Putfield Fd Cl*) *P s* (*length cs*) (*stkLength P C M pc*)
arbitrary arbitrary) ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*, *M*, *Suc pc*)#*cs*, *False*)⌋ -) −*ek*→ (- (*C*, *M*, *Suc pc*)#*cs*,*None* -)

| *JCFG-Putfield-Exc-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Putfield Fd Cl*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = *cs*′;
ek = (λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *2*) = *Null*)$_{\surd}$ ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊(*cs*′, *True*)⌋ -)

| *JCFG-Putfield-Exc-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Putfield Fd Cl*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = (*C*′, *M*′, *pc*′)#*cs*′;
ek = ⇑(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs*′,(*stkLength P C*′ *M*′ *pc*′) − *1*) := *Addr* (*addr-of-sys-xcpt NullPointer*)),
loc)
) ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*′, *M*′, *pc*′)#*cs*′, *True*)⌋ -) −*ek*→ (- (*C*′, *M*′, *pc*′)#*cs*′,*None* -)

| *JCFG-Putfield-Exc-Exit*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Putfield Fd Cl*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = [] ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊([], *True*)⌋ -) −⇑*id*→ (-*Exit*-)

| *JCFG-Checkcast-Normal-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Checkcast Cl*);
ek = (λ(*h*,*stk*,*loc*). *cast-ok* (P_{wf}) *Cl h* (*stk*(*length cs*, *stkLength P C M pc* − *Suc 0*)))$_{\surd}$ ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *Suc pc*)#*cs*,*None* -)

| *JCFG-Checkcast-Exc-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Checkcast Cl*);
find-handler-for P ClassCast ((*C*, *M*, *pc*)#*cs*) = *cs*′;
ek = (λ(*h*,*stk*,*loc*). ¬ *cast-ok* (P_{wf}) *Cl h* (*stk*(*length cs*, *stkLength P C M pc* − *Suc 0*)))$_{\surd}$ ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊(*cs*′, *True*)⌋ -)

| *JCFG-Checkcast-Exc-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);

(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Checkcast Cl*);
find-handler-for P ClassCast ((*C*, *M*, *pc*)#*cs*) = (*C′*, *M′*, *pc′*)#*cs′*;
ek = ⇑(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs′*,(*stkLength P C′ M′ pc′*) − *1*) := *Addr* (*addr-of-sys-xcpt ClassCast*)),
loc)
) ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C′*, *M′*, *pc′*)#*cs′*, *True*)⌋ -) −*ek*→ (- (*C′*, *M′*, *pc′*)#*cs′*,*None* -)

| *JCFG-Checkcast-Exc-Exit*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Checkcast Cl*);
find-handler-for P ClassCast ((*C*, *M*, *pc*)#*cs*) = [] ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊([], *True*)⌋ -) −⇑*id*→ (-*Exit*-)

| *JCFG-Invoke-Normal-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Invoke M2 n*);
cd = *length cs*;
stk-length = *stkLength P C M pc*;
ek = (λ(*h*,*stk*,*loc*).
stk(*cd*, *stk-length* − *Suc n*) ≠ *Null* ∧
fst(*method* (P_{wf}) (*cname-of h* (*the-Addr*(*stk*(*cd*, *stk-length* − *Suc n*)))) *M2*) = *D*
)$_{\surd}$ ⟧
⟹
prog ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊((*D*, *M2*, *0*)#(*C*, *M*, *pc*)#*cs*, *False*)⌋ -)

| *JCFG-Invoke-Normal-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Invoke M2 n*);
stk-length = *stkLength P C M pc*;
loc-length = *locLength P D M2 0*;
ek = ⇑(λ*s*. *exec-instr* (*Invoke M2 n*) *P s* (*length cs*) *stk-length arbitrary loc-length*)
⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*D*, *M2*, *0*)#(*C*, *M*, *pc*)#*cs*, *False*)⌋ -) −*ek*→
(- (*D*, *M2*, *0*)#(*C*, *M*, *pc*)#*cs*,*None* -)

| *JCFG-Invoke-Exc-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Invoke m2 n*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = *cs′*;
ek = (λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *Suc n*) = *Null*)$_{\surd}$ ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊(*cs′*, *True*)⌋ -)

| *JCFG-Invoke-Exc-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Invoke M2 n*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = (*C'*, *M'*, *pc'*)#*cs'*;
ek = ⇑(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs'*,(*stkLength P C' M' pc'*) − *1*) := *Addr* (*addr-of-sys-xcpt NullPointer*)),
loc)
)
⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C'*, *M'*, *pc'*)#*cs'*, *True*)⌋ -) −*ek*→ (- (*C'*, *M'*, *pc'*)#*cs'*,*None* -)

| *JCFG-Invoke-Exc-Exit*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Invoke M2 n*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = [] ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊([], *True*)⌋ -) −⇑*id*→ (-*Exit*-)

| *JCFG-Return-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = *Return*;
stk-length = *stkLength P C M pc*;
r-stk-length = *stkLength P C' M'* (*Suc pc'*);
ek = ⇑(λ*s*. *exec-instr Return P s* (*Suc* (*length cs*)) *stk-length r-stk-length arbitrary*) ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#(*C'*, *M'*, *pc'*)#*cs*,*None* -) −*ek*→ (- (*C'*, *M'*, *Suc pc'*)#*cs*,*None* -)

| *JCFG-Goto-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = *Goto idx* ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −⇑*id*→ (- (*C*, *M*, *nat* (*int pc* + *idx*))#*cs*,*None* -)

| *JCFG-IfFalse-False*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*IfFalse b*);
b ≠ *1*;
ek = (λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *1*) = *Bool False*)$_{\surd}$ ⟧
⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *nat* (*int pc* + *b*))#*cs*,*None* -)

| *JCFG-IfFalse-Next*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*IfFalse b*);
ek = (λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *1*) ≠ *Bool False* ∨ *b* = *1*)$_{\surd}$ ⟧

$\Longrightarrow$ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *Suc pc*)#*cs*,*None* -)

| *JCFG-Throw-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = *Throw*;
cd = *length cs*;
stk-length = *stkLength P C M pc*;
∃ *Exc*. *find-handler-for P Exc* ((*C*, *M*, *pc*)#*cs*) = *cs'*;
ek = (λ(*h*,*stk*,*loc*).
(*stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null* ∧
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = *cs'*) ∨
(*stk*(*length cs*, *stkLength P C M pc* − *1*) ≠ *Null* ∧
find-handler-for P (*cname-of h* (*the-Addr*(*stk*(*cd*, *stk-length* − *1*)))) ((*C*, *M*, *pc*)#*cs*) = *cs'*)
)$_{\surd}$ ⟧
$\Longrightarrow$ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊(*cs'*, *True*)⌋ -)

| *JCFG-Throw-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = *Throw*;
ek = ⇑(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs'*,(*stkLength P C' M' pc'*) − *1*) :=
if (*stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null*) *then*
Addr (*addr-of-sys-xcpt NullPointer*)
else (*stk*(*length cs*, *stkLength P C M pc* − *1*))),
loc)
) ⟧
$\Longrightarrow$ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C'*, *M'*, *pc'*)#*cs'*, *True*)⌋ -) −*ek*→ (- (*C'*, *M'*, *pc'*)#*cs'*,*None* -)

| *JCFG-Throw-Exit*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = *Throw* ⟧
$\Longrightarrow$ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊([],*True*)⌋ -) −⇑*id*→ (-*Exit*-)

5.1.7 CFG properties

lemma *JVMCFG-Exit-no-sourcenode* [*dest*]:
assumes *edge*:*prog* ⊢ (-*Exit*-) −*et*→ *n'*
shows *False*
proof −
{ fix *n*
have ⟦*prog* ⊢ *n* −*et*→ *n'*; *n* = (-*Exit*-)⟧ $\Longrightarrow$ *False*
by (*auto elim*!: *JVM-CFG.cases*)
}
with *edge* **show** *?thesis* **by** *fastforce*
qed

lemma *JVMCFG-Entry-no-targetnode* [*dest*]:
  assumes *edge*:*prog* ⊢ *n* −*et*→ (*-Entry-*)
  shows *False*
proof −
  { **fix** *n*′ **have** ⟦*prog* ⊢ *n* −*et*→ *n*′; *n*′ = (*-Entry-*)⟧ ⟹ *False*
    by (*auto elim*!: *JVM-CFG.cases*)
  }
  with *edge* **show** *?thesis* **by** *fastforce*
qed

lemma *JVMCFG-EntryD*:
  ⟦(*P*,*C*,*M*) ⊢ *n* −*et*→ *n*′; *n* = (*-Entry-*)⟧
  ⟹ (*n*′ = (*-Exit-*) ∧ *et* = (λ*s*. *False*)$_{\surd}$) ∨ (*n*′ = (*-* [(*C*,*M*,*0*)],*None* -) ∧ *et* = (λ*s*. *True*)$_{\surd}$)
by (*erule JVM-CFG.cases*) *simp-all*

declare *split-def* [*simp add*]
declare *find-handler-for.simps* [*simp del*]

lemma *JVMCFG-edge-det*:
  ⟦*prog* ⊢ *n* −*et*→ *n*′; *prog* ⊢ *n* −*et*′→ *n*′′⟧ ⟹ *et* = *et*′
  by (*erule JVM-CFG.cases*, (*erule JVM-CFG.cases*, *fastforce*+)+)

declare *split-def* [*simp del*]
declare *find-handler-for.simps* [*simp add*]

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

5.2 Instatiation of the *CFG* locale

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

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

abbreviation *kind* :: *j-edge* ⇒ *state edge-kind*
  where *kind e* ≡ *fst*(*snd e*)

The following predicates define the aforementioned well-formedness requirements for nodes. Later, *valid-callstack* will be implied by Jinja's state conformance.

fun *valid-callstack* :: *jvmprog* ⇒ *callstack* ⇒ *bool*
where
  valid-callstack prog [] = *True*
| *valid-callstack* (*P*, *C0*, *Main*) [(*C*, *M*, *pc*)] ⟷

  $C = C0 \wedge M = Main \wedge$
  (P_Φ) C M ! $pc \neq None \wedge$
  $(\exists\ T\ Ts\ mxs\ mxl\ is\ xt.\ (P_{wf}) \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}(mxs,\ mxl,\ is,\ xt)\ in\ C \wedge pc < length\ is)$
| *valid-callstack* $(P,\ C0,\ Main)$ $((C,\ M,\ pc)\#(C',\ M',\ pc')\#cs) \longleftrightarrow$
  instrs-of (P_{wf}) C' M' ! $pc' =$
    Invoke M (*locLength* P C M 0 $-$ *Suc* (*fst*(*snd*(*snd*(*snd*(*snd*(*method* (P_{wf}) C M))))))) $\wedge$
  (P_Φ) C M ! $pc \neq None \wedge$
  $(\exists\ T\ Ts\ mxs\ mxl\ is\ xt.\ (P_{wf}) \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}(mxs,\ mxl,\ is,\ xt)\ in\ C \wedge pc < length\ is) \wedge$
  valid-callstack $(P,\ C0,\ Main)$ $((C',\ M',\ pc')\#cs)$

fun *valid-node* :: *jvmprog* ⇒ *j-node* ⇒ *bool*
where
 valid-node prog (*-Entry-*) = *True*

| *valid-node prog* (*- cs,None -*) ⟷ *valid-callstack prog cs*
| *valid-node prog* (*- cs,*$\lfloor(cs',\ xf)\rfloor$ *-*) ⟷
  valid-callstack prog cs ∧ *valid-callstack prog cs'* ∧
  (∃ *Q*. *prog* ⊢ (*- cs,None -*) $-(Q)_{\surd}\rightarrow$ (*- cs,*$\lfloor(cs',\ xf)\rfloor$ *-*)) ∧
  (∃ *f*. *prog* ⊢ (*- cs,*$\lfloor(cs',\ xf)\rfloor$ *-*) $-\Uparrow f\rightarrow$ (*- cs',None -*))

fun *valid-edge* :: *jvmprog* ⇒ *j-edge* ⇒ *bool*
where
 valid-edge prog a ⟷
  (*prog* ⊢ (*sourcenode a*) −(*kind a*)→ (*targetnode a*))
  ∧ (*valid-node prog* (*sourcenode a*))
  ∧ (*valid-node prog* (*targetnode a*))

interpretation *JVM-CFG-Interpret*:
 CFG sourcenode targetnode kind valid-edge prog Entry
 for *prog*
proof (*unfold-locales*)
 fix *a*
 assume *ve*: *valid-edge prog a*
  and *trg*: *targetnode a* = (*-Entry-*)
 obtain *n et n'*
  where *a* = (*n*,*et*,*n'*)
  by (*cases a*) *fastforce*
 with *ve trg*
  have *prog* ⊢ *n* −*et*→ (*-Entry-*) **by** *simp*
 thus *False* **by** *fastforce*
next
 fix *a a'*
 assume *valid*: *valid-edge prog a*
  and *valid'*: *valid-edge prog a'*
  and *sourceeq*: *sourcenode a* = *sourcenode a'*
  and *targeteq*: *targetnode a* = *targetnode a'*

obtain *n1 et n2*
  where *a*:*a* = (*n1*, *et*, *n2*)
  by (*cases a*) *fastforce*
obtain *n1′ et′ n2′*
  where *a′*:*a′* = (*n1′*, *et′*, *n2′*)
  by (*cases a′*) *fastforce*
from *a valid a′ valid′ sourceeq targeteq*
have *et* = *et′*
  by (*fastforce elim*: *JVMCFG-edge-det*)
with *a a′ sourceeq targeteq*
show *a* = *a′*
  by *simp*
qed

interpretation *JVM-CFGExit-Interpret*:
  CFGExit sourcenode targetnode kind valid-edge prog Entry (*-Exit-*)
  for *prog*
proof(*unfold-locales*)
  fix *a*
  assume *ve*: *valid-edge prog a*
    and *src*: *sourcenode a* = (*-Exit-*)
  obtain *n et n′*
    where *a* = (*n*,*et*,*n′*)
    by (*cases a*) *fastforce*
  with *ve src*
    have *prog* ⊢ (*-Exit-*) −*et*→ *n′* **by** *simp*
  thus *False* **by** *fastforce*
next
  have *prog* ⊢ (*-Entry-*) −(λ*s*. *False*)$_{\surd}$→ (*-Exit-*)
    by (*rule JCFG-EntryExit*)
  thus ∃ *a*. *valid-edge prog a* ∧ *sourcenode a* = (*-Entry-*) ∧
          targetnode a = (*-Exit-*) ∧ *kind a* = (λ*s*. *False*)$_{\surd}$
    by *fastforce*
qed

end

Chapter 6

Standard and Weak Control Dependence

6.1 A type for well-formed programs

theory *JVMPostdomination* **imports** *JVMInterpretation ../Basic/Postdomination*
begin

For instantiating *Postdomination* every node in the CFG of a program must be reachable from the (*-Entry-*) node and there must be a path to the (*-Exit-*) node from each node.

Therefore, we restrict the set of allowed programs to those, where the CFG fulfills these requirements. This is done by defining a new type for well-formed programs. The universe of every type in Isabelle must be non-empty. That's why we first define an example program *EP* and its typing *Phi-EP*, which is a member of the carrier set of the later defined type.

Restricting the set of allowed programs in this way is reasonable, as Jinja's compiler only produces byte code programs, that are members of this type (A proof for this is current work).

definition *EP* :: *jvm-prog*
  where *EP* = (*''C''*, *Object*, [], [(*''M''*, [], *Void*, *1*::*nat*, *0*::*nat*, [*Push Unit*, *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* [])

Now we show, that *EP* is indeed a well-formed program in the sense of Jinja's byte code verifier

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*: $i < Suc\ (Suc\ 0) \Longrightarrow i = 0 \lor i = 1$
by *auto*

lemma *EP-wf*: $wf\text{-}jvm\text{-}prog_{Phi\text{-}EP}\ EP$
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)
have *classes-wf*:
∀ *c*∈*set EP*.
wf-cdecl
$(\lambda P\ C\ (M,\ Ts,\ T_r,\ mxs,\ mxl_0,\ is,\ xt).\ wt\text{-}method\ P\ C\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ (Phi\text{-}EP\ C\ M))$
EP c
proof
fix *C*
assume *C-in-EP*: *C* ∈ *set EP*
show *wf-cdecl*
$(\lambda P\ C\ (M,\ Ts,\ T_r,\ mxs,\ mxl_0,\ is,\ xt).\ wt\text{-}method\ P\ C\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ (Phi\text{-}EP\ C\ M))$
EP C
proof (*cases C* ∈ *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 [*simp*]: *C* = (*''C''*, *the* (*class EP ''C''*))
by (*auto simp*: *EP-def SystemClasses-def class-def*)
show *?thesis*
apply (*auto dest*!: *i-max-2D*
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
stk-esl-def upto-esl-def loc-sl-def SemiType.esl-def
SemiType.sup-def Err.sl-def Err.le-def err-def Listn.sl-def
Err.esl-def Opt.esl-def Product.esl-def relevant-entries-def)
apply (*fastforce simp*: *SystemClasses-def ObjectC-def*)
apply (*clarsimp simp*: *Method-def*)
apply (*cases rule*: *Methods.cases*,
(*fastforce simp*: *class-def SystemClasses-def ObjectC-def*)+)
apply (*clarsimp simp*: *Method-def*)
by (*cases rule*: *Methods.cases*,
(*fastforce simp*: *class-def SystemClasses-def ObjectC-def*)+)
qed
qed
with *distinct-EP*
show (∀ *c*∈*set EP*.
wf-cdecl
(λ*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) ∧
distinct-fst EP
by *simp*
qed

lemma [*simp*]: *Abs-wf-jvmprog* (*EP*, *Phi-EP*)$_{wf}$ = *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*]: *Abs-wf-jvmprog* (*EP*, *Phi-EP*)$_\Phi$ = *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*, *Return*] ∧
xt = [] ∧
D = ''*C*''
apply (*clarsimp simp*: *Method-def EP-def*)
apply (*erule Methods.cases*, *clarsimp simp*: *class-def SystemClasses-def ObjectC-def*)
apply (*clarsimp simp*: *class-def*)
apply (*erule Methods.cases*)
by (*fastforce simp*: *class-def SystemClasses-def ObjectC-def NullPointerC-def OutOfMemoryC-def ClassCastC-def if-split-eq1*)+

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* = (*1*, *0*, [*Push Unit*, *Return*], []) *in* ''*C*''
apply (*auto simp*: *Method-def EP-def*)
apply (*rule-tac x*=*Map.empty*(''*M*'' ↦ (([], *Void*, *1*, *0*, [*Push Unit*, *Return*], []),''*C*'')) **in** *exI*)
apply *auto*
apply (*rule Methods.intros*(*2*))
apply (*fastforce simp*: *class-def*)
apply *clarsimp*
apply (*rule Methods.intros*(*1*))
apply (*fastforce simp*: *class-def SystemClasses-def ObjectC-def*)
apply *fastforce*
by *fastforce*

lemma *instrs-of-EP-C-M* [*simp*]:
instrs-of EP ''*C*'' ''*M*'' = [*Push Unit*, *Return*]
using *C-sees-M-in-EP*

```
apply (simp add: method-def)
apply (rule theI2)
  apply fastforce
 apply (clarsimp dest!: method-in-EP-is-M)
by (clarsimp dest!: method-in-EP-is-M)
```

```
lemma valid-node-in-EP-D:
  valid-node (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M'') n
  ⟹ n ∈ {(-Entry-), (- [(''C'', ''M'', 0)],None -), (- [(''C'', ''M'', 1)],None -),
(-Exit-)}
proof −
  assume vn: valid-node (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M'') n
  show ?thesis
  proof (cases n)
    case Entry
    thus ?thesis
      by simp
  next
    case [simp]: (Node cs opt)
    show ?thesis
    proof (cases opt)
      case [simp]: None
      from vn
      show ?thesis
        apply (cases cs)
         apply simp
        apply (case-tac list)
         apply clarsimp
         apply (drule method-in-EP-is-M)
         apply clarsimp
        apply clarsimp
        apply (drule method-in-EP-is-M)
        apply clarsimp
        apply (case-tac lista)
         apply clarsimp
         apply (drule method-in-EP-is-M)
         apply clarsimp
         apply (case-tac ba, clarsimp, clarsimp)
        apply clarsimp
        apply (drule method-in-EP-is-M)
        apply clarsimp
        by (case-tac ba, clarsimp, clarsimp)
    next
      case [simp]: (Some f)
      obtain cs'' xf where [simp]: f = (cs'', xf)
        by (cases f, fastforce)
      from vn
```

show *?thesis*
apply (*cases cs*)
apply *clarsimp*
apply (*erule JVM-CFG.cases*, *clarsimp*+)
apply (*case-tac list*)
apply *clarsimp*
apply (*frule method-in-EP-is-M*)
apply (*case-tac b*)
apply (*erule JVM-CFG.cases*, *clarsimp*+)
apply (*erule JVM-CFG.cases*, *clarsimp*+)
apply (*frule method-in-EP-is-M*)
apply (*case-tac b*)
apply (*erule JVM-CFG.cases*, *clarsimp*+)
by (*erule JVM-CFG.cases*, *clarsimp*+)
qed
qed
qed

lemma *EP-C-M-0-valid* [*simp*]:
JVM-CFG-Interpret.valid-node (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), ''*C*'', ''*M*'')
(- [(''*C*'', ''*M*'', *0*)],*None* -)
proof −
have *valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), ''*C*'', ''*M*'')
((-*Entry*-), (λ*s. True*)$_{\surd}$, (- [(''*C*'', ''*M*'', *0*)],*None* -))
apply (*auto simp*: *Phi-EP-def*)
by *rule auto*
thus *?thesis*
by (*fastforce simp*: *JVM-CFG-Interpret.valid-node-def*)
qed

lemma *EP-C-M-Suc-0-valid* [*simp*]:
JVM-CFG-Interpret.valid-node (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), ''*C*'', ''*M*'')
(- [(''*C*'', ''*M*'', *Suc 0*)],*None* -)
proof −
have *valid-edge* (*Abs-wf-jvmprog* (*EP*, *Phi-EP*), ''*C*'', ''*M*'')
((- [(''*C*'', ''*M*'', *Suc 0*)],*None* -), ⇑*id*, (-*Exit*-))
apply (*auto simp*: *Phi-EP-def*)
by *rule auto*
thus *?thesis*
by (*fastforce simp*: *JVM-CFG-Interpret.valid-node-def*)
qed

definition
cfg-wf-prog =
{*P*. (∀ *n. valid-node P n* ⟶
(∃ *as. JVM-CFG-Interpret.path P* (-*Entry*-) *as n*) ∧
(∃ *as. JVM-CFG-Interpret.path P n as* (-*Exit*-)))}

```
typedef cfg-wf-prog = cfg-wf-prog
  unfolding cfg-wf-prog-def
proof
  let ?prog = ((Abs-wf-jvmprog (EP, Phi-EP)), ''C'', ''M'')
  let ?edge0 = ((-Entry-), (λs. False)√, (-Exit-))
  let ?edge1 = ((-Entry-), (λs. True)√, (- [(''C'', ''M'', 0)],None -))
  let ?edge2 = ((- [(''C'', ''M'', 0)],None -),
               ⇑(λ(h, stk, loc). (h, stk((0, 0) := Unit), loc)),
               (- [(''C'', ''M'', 1)],None -))
  let ?edge3 = ((- [(''C'', ''M'', 1)],None -), ⇑id, (-Exit-))
  show ?prog ∈ {P. ∀ n. valid-node P n ⟶
               (∃ as. CFG.path sourcenode targetnode (valid-edge P) (-Entry-) as n)
∧
               (∃ as. CFG.path sourcenode targetnode (valid-edge P) n as (-Exit-))}
  proof (auto dest!: valid-node-in-EP-D)
    have JVM-CFG-Interpret.path ?prog (-Entry-) [] (-Entry-)
      by (simp add: JVM-CFG-Interpret.path.empty-path)
    thus ∃ as. JVM-CFG-Interpret.path ?prog (-Entry-) as (-Entry-)
      by fastforce
  next
    have JVM-CFG-Interpret.path ?prog (-Entry-) [?edge0] (-Exit-)
      by rule (auto intro: JCFG-EntryExit JVM-CFG-Interpret.path.empty-path)
    thus ∃ as. JVM-CFG-Interpret.path ?prog (-Entry-) as (-Exit-)
      by fastforce
  next
   have JVM-CFG-Interpret.path ?prog (-Entry-) [?edge1] (- [(''C'', ''M'', 0)],None
-)
    by rule (auto intro: JCFG-EntryStart simp: JVM-CFG-Interpret.path.empty-path
Phi-EP-def)
   thus ∃ as. JVM-CFG-Interpret.path ?prog (-Entry-) as (- [(''C'', ''M'', 0)],None
-)
      by fastforce
  next
     have JVM-CFG-Interpret.path ?prog (- [(''C'', ''M'', 0)],None -) [?edge2,
?edge3] (-Exit-)
      apply rule
         apply rule
            apply (auto simp: JVM-CFG-Interpret.path.empty-path Phi-EP-def)
       apply (rule JCFG-ReturnExit, auto)
      by (rule JCFG-Straight-NoExc, auto simp: Phi-EP-def)
   thus ∃ as. JVM-CFG-Interpret.path ?prog (- [(''C'', ''M'', 0)],None -) as (-Exit-)
      by fastforce
  next
   have JVM-CFG-Interpret.path ?prog (-Entry-) [?edge1, ?edge2] (- [(''C'', ''M'',
1)],None -)
      apply rule
         apply rule
            apply (auto simp: JVM-CFG-Interpret.path.empty-path Phi-EP-def)
       apply (rule JCFG-Straight-NoExc, auto simp: Phi-EP-def)
```

by (*rule JCFG-EntryStart*, *auto*)
thus ∃ *as*. *JVM-CFG-Interpret.path ?prog* (*-Entry-*) *as* (*-* [(*''C''*, *''M''*, *Suc 0*)],*None -*)
by *fastforce*
next
have *JVM-CFG-Interpret.path ?prog* (*-* [(*''C''*, *''M''*, *Suc 0*)],*None -*) [*?edge3*] (*-Exit-*)
apply *rule*
apply (*auto simp*: *JVM-CFG-Interpret.path.empty-path Phi-EP-def*)
by (*rule JCFG-ReturnExit*, *auto*)
thus ∃ *as*. *JVM-CFG-Interpret.path ?prog* (*-* [(*''C''*, *''M''*, *Suc 0*)],*None -*) *as* (*-Exit-*)
by *fastforce*
next
have *JVM-CFG-Interpret.path ?prog* (*-Entry-*) [*?edge0*] (*-Exit-*)
by *rule* (*auto intro*: *JCFG-EntryExit JVM-CFG-Interpret.path.empty-path*)
thus ∃ *as*. *JVM-CFG-Interpret.path ?prog* (*-Entry-*) *as* (*-Exit-*)
by *fastforce*
next
have *JVM-CFG-Interpret.path ?prog* (*-Exit-*) [] (*-Exit-*)
by (*simp add*: *JVM-CFG-Interpret.path.empty-path*)
thus ∃ *as*. *JVM-CFG-Interpret.path ?prog* (*-Exit-*) *as* (*-Exit-*)
by *fastforce*
qed
qed

abbreviation *lift-to-cfg-wf-prog* :: (*jvmprog* ⇒ *'a*) ⇒ (*cfg-wf-prog* ⇒ *'a*)
(‹$-_{CFG}$›)
where f_{CFG} ≡ (λ*P*. *f* (*Rep-cfg-wf-prog P*))

6.2 Interpretation of the *Postdomination* locale

interpretation *JVM-CFG-Postdomination*:
Postdomination sourcenode targetnode kind $valid\text{-}edge_{CFG}$ *prog Entry* (*-Exit-*)
for *prog*
proof(*unfold-locales*)
fix *n*
assume *vn*: *CFG.valid-node sourcenode targetnode* ($valid\text{-}edge_{CFG}$ *prog*) *n*
have *prog-is-cfg-wf-prog*: *Rep-cfg-wf-prog prog* ∈ *cfg-wf-prog*
by (*rule Rep-cfg-wf-prog*)
obtain *P C0 Main* **where** [*simp*]: *Rep-cfg-wf-prog prog* = (*P*,*C0*,*Main*)
by (*cases Rep-cfg-wf-prog prog*, *fastforce*)
from *prog-is-cfg-wf-prog* **have** (*P*, *C0*, *Main*) ∈ *cfg-wf-prog*
by *simp*
hence *valid-node* (*P*, *C0*, *Main*) *n* ⟶
(∃ *as*. *CFG.path sourcenode targetnode* (*valid-edge* (*P*, *C0*, *Main*)) (*-Entry-*) *as* *n*)
by (*fastforce simp*: *cfg-wf-prog-def*)

moreover from *vn* **have** *valid-node* (*P*, *C0*, *Main*) *n*
by (*auto simp*: *JVM-CFG-Interpret.valid-node-def*)
ultimately
show ∃ *as*. *CFG.path sourcenode targetnode* (*valid-edge*$_{CFG}$ *prog*) (*-Entry-*) *as n*
by *simp*
next
fix *n*
assume *vn*: *CFG.valid-node sourcenode targetnode* (*valid-edge*$_{CFG}$ *prog*) *n*
have *prog-is-cfg-wf-prog*: *Rep-cfg-wf-prog prog* ∈ *cfg-wf-prog*
by (*rule Rep-cfg-wf-prog*)
obtain *P C0 Main* **where** [*simp*]: *Rep-cfg-wf-prog prog* = (*P*,*C0*,*Main*)
by (*cases Rep-cfg-wf-prog prog*, *fastforce*)
from *prog-is-cfg-wf-prog* **have** (*P*, *C0*, *Main*) ∈ *cfg-wf-prog*
by *simp*
hence *valid-node* (*P*, *C0*, *Main*) *n* ⟶
(∃ *as*. *CFG.path sourcenode targetnode* (*valid-edge* (*P*, *C0*, *Main*)) *n as* (*-Exit-*))
by (*fastforce simp*: *cfg-wf-prog-def*)
moreover from *vn* **have** *valid-node* (*P*, *C0*, *Main*) *n*
by (*auto simp*: *JVM-CFG-Interpret.valid-node-def*)
ultimately
show ∃ *as*. *CFG.path sourcenode targetnode* (*valid-edge*$_{CFG}$ *prog*) *n as* (*-Exit-*)
by *simp*
qed

6.3 Interpretation of the *StrongPostdomination* locale

6.3.1 Some helpfull lemmas

lemma *find-handler-for-tl-eq*:
find-handler-for P Exc cs = (*C*,*M*,*pcx*)#*cs'* ⟹ ∃ *cs''* *pc*. *cs* = *cs''* @ [(*C*,*M*,*pc*)] @ *cs'*
by (*induct cs*, *auto*)

lemma *valid-callstack-tl*:
valid-callstack prog ((*C*,*M*,*pc*)#*cs*) ⟹ *valid-callstack prog cs*
by (*cases prog*, *cases cs*, *auto*)

lemma *find-handler-Throw-Invoke-pc-in-range*:
⟦*cs* = (*C'*,*M'*,*pc'*)#*cs'*; *valid-callstack* (*P*,*C0*,*Main*) *cs*;
instrs-of (P_{wf}) *C' M'* ! *pc'* = *Throw* ∨ (∃ *M''* *n''*. *instrs-of* (P_{wf}) *C' M'* ! *pc'* = *Invoke M'' n''*);
find-handler-for P Exc cs = (*C*,*M*,*pc*)#*cs''* ⟧
⟹ *pc* < *length* (*instrs-of* (P_{wf}) *C M*)
proof (*induct cs arbitrary*: *C' M' pc' cs'*)
case *Nil*
thus *?case* **by** *simp*
next

case (*Cons a cs*)
hence [*simp*]: $a = (C',M',pc')$ **and** [*simp*]: $cs = cs'$ **by** *simp-all*
note *IH* = ‹⋀*C' M' pc' cs'*.
⟦*cs* = (*C'*, *M'*, *pc'*) # *cs'*; *valid-callstack* (*P*, *C0*, *Main*) *cs*;
instrs-of P_{wf} *C' M'* ! *pc'* = *Throw* ∨
(∃ *M'' n''*. *instrs-of* P_{wf} *C' M'* ! *pc'* = *Invoke M'' n''*);
find-handler-for P Exc cs = (*C*, *M*, *pc*) # *cs''*⟧
⟹ *pc* < *length* (*instrs-of* P_{wf} *C M*)›
note *throw* = ‹*instrs-of* P_{wf} *C' M'* ! *pc'* = *Throw* ∨ (∃ *M'' n''*. *instrs-of* P_{wf} *C' M'* ! *pc'* = *Invoke M'' n''*)›
note *fhf* = ‹*find-handler-for P Exc* (*a* # *cs*) = (*C*, *M*, *pc*) # *cs''*›
note *v-cs-a-cs* = ‹*valid-callstack* (*P*, *C0*, *Main*) (*a* # *cs*)›
show *?case*
proof (*cases match-ex-table* (P_{wf}) *Exc pc'* (*ex-table-of* (P_{wf}) *C' M'*))
case *None*
with *fhf* **have** *fhf-tl*: *find-handler-for P Exc cs* = (*C*,*M*,*pc*)#*cs''*
by *simp*
from *v-cs-a-cs* **have** *valid-callstack* (*P*, *C0*, *Main*) *cs*
by (*auto dest*: *valid-callstack-tl*)
from *v-cs-a-cs*
have *cs* ≠ [] ⟶ (*let* (*C*,*M*,*pc*) = *hd cs in* ∃ *n*. *instrs-of* (P_{wf}) *C M* ! *pc* = *Invoke M' n*)
by (*cases cs'*, *auto*)
with *IH None fhf-tl* ‹*valid-callstack* (*P*, *C0*, *Main*) *cs*›
show *?thesis*
by (*cases cs*) *fastforce+*
next
case (*Some xte*)
with *fhf* **have** [*simp*]: $C' = C$ **and** [*simp*]: $M' = M$ **by** *simp-all*
from *v-cs-a-cs fhf Some*
obtain *Ts T mxs mxl is xt* **where** *wt-class*:
(P_{wf}) ⊢ *C sees M*: *Ts*→*T* = (*mxs*, *mxl*, *is*, *xt*) *in C* ∧
pc' < *length is* ∧ (P_{Φ}) *C M* ! *pc'* ≠ *None*
by (*cases cs*) *fastforce+*
with *wf-jvmprog-is-wf* [*of P*]
have *wt-instr*:(P_{wf}),*T*,*mxs*,*length is*,*xt* ⊢ *is* ! *pc'*,*pc'* :: (P_{Φ}) *C M*
by (*fastforce dest*!: *wt-jvm-prog-impl-wt-instr*)
from *Some fhf* **obtain** *f t D d* **where** (*f*,*t*,*D*,*pc*,*d*)∈ *set* (*ex-table-of* (P_{wf}) *C M*) ∧
matches-ex-entry (P_{wf}) *Exc pc'* (*f*,*t*,*D*,*pc*,*d*)
by (*cases xte*, *fastforce dest*: *match-ex-table-SomeD*)
with *wt-instr throw wt-class*
show *?thesis*
by (*fastforce simp*: *relevant-entries-def is-relevant-entry-def matches-ex-entry-def*)
qed
qed

6.3.2 Every node has only finitely many successors

lemma *successor-set-finite*:
  JVM-CFG-Interpret.valid-node prog n
  $\Longrightarrow$ *finite* $\{n'. \exists a'.$ *valid-edge prog* $a' \wedge$ *sourcenode* $a' = n \wedge$
            targetnode $a' = n'\}$
proof –
  assume *valid-node*: *JVM-CFG-Interpret.valid-node prog n*
  obtain *P C0 Main* **where** [*simp*]: *prog* = (*P*, *C0*, *Main*)
    by (*cases prog*, *fastforce*)
  note *P-wf* = *wf-jvmprog-is-wf* [*of P*]
  show *?thesis*
  proof (*cases n*)
    case *Entry*
    thus *?thesis*
      by (*rule-tac B*={(-*Exit*-), (- [(*C0*, *Main*, *0*)],*None* -)} **in** *finite-subset*,
        auto dest: *JVMCFG-EntryD*)
  next
    case [*simp*]: (*Node cs x*)
    show *?thesis*
    proof (*cases cs*)
      case *Nil*
      thus *?thesis*
        by (*rule-tac B*={} **in** *finite-subset*,
          auto elim: *JVM-CFG.cases*)
    next
      case [*simp*]: (*Cons a cs'*)
      obtain *C M pc* **where** [*simp*]: *a* = (*C*,*M*,*pc*) **by** (*cases a*, *fastforce*)
      have *finite-classes*: *finite* {*C*. *is-class* (P_{wf}) *C*}
        by (*rule finite-is-class*)
      from *valid-node* **have** *is-class* (P_{wf}) *C*
        apply (*auto simp*: *JVM-CFG-Interpret.valid-node-def*)
         apply (*cases x*, *auto*)
          apply (*cases cs'*, *auto dest*!: *sees-method-is-class*)
         apply (*cases cs'*, *auto dest*!: *sees-method-is-class*)
        apply (*cases cs'*, *auto dest*!: *sees-method-is-class*)
         apply (*cases x*, *auto dest*!: *sees-method-is-class*)
        by (*cases x*, *auto dest*!: *sees-method-is-class*)
      show *?thesis*
      proof (*cases instrs-of* (P_{wf}) *C M* ! *pc*)
        case (*Load nat*)
        with *valid-node*
        show *?thesis*
          apply *auto*
          apply (*rule-tac B*={(- (*C*,*M*,*Suc pc*)#*cs'*,*x* -)} **in** *finite-subset*)
           by (*auto elim*: *JVM-CFG.cases*)
      next
        case (*Store nat*)
        with *valid-node*

show *?thesis*
apply *auto*
apply (*rule-tac B*={(- (*C*,*M*,*Suc pc*)#*cs'*,*x* -)} **in** *finite-subset*)
by (*auto elim*: *JVM-CFG.cases*)
next
case (*Push val*)
with *valid-node*
show *?thesis*
apply *auto*
apply (*rule-tac B*={(- (*C*,*M*,*Suc pc*)#*cs'*,*x* -)} **in** *finite-subset*)
by (*auto elim*: *JVM-CFG.cases*)
next
case (*New C'*)
with *valid-node*
show *?thesis*
apply *auto*
apply (*rule-tac B*={(- (*C*,*M*,*pc*)#*cs'*,⌊((*C*,*M*,*Suc pc*)#*cs'*,*False*)⌋ -),
(- (*C*,*M*,*pc*)#*cs'*,⌊(*find-handler-for P OutOfMemory* ((*C*,*M*,*pc*)#*cs'*),*True*)⌋
-),
(- *fst*(*the*(*x*)),*None* -)} **in** *finite-subset*)
apply (*rule subsetI*)
apply (*clarsimp simp del*: *find-handler-for*.*simps*)
by (*erule JVM-CFG.cases*, *simp-all del*: *find-handler-for*.*simps*)
next
case (*Getfield Fd C'*)
with *valid-node*
show *?thesis*
apply *auto*
apply (*rule-tac B*={(- (*C*,*M*,*pc*)#*cs'*,⌊((*C*,*M*,*Suc pc*)#*cs'*,*False*)⌋ -),
(- (*C*,*M*,*pc*)#*cs'*,⌊(*find-handler-for P NullPointer* ((*C*,*M*,*pc*)#*cs'*),*True*)⌋
-),
(- *fst*(*the*(*x*)),*None* -)} **in** *finite-subset*)
apply (*rule subsetI*)
apply (*clarsimp simp del*: *find-handler-for*.*simps*)
by (*erule JVM-CFG.cases*, *simp-all del*: *find-handler-for*.*simps*)
next
case (*Putfield Fd C'*)
with *valid-node*
show *?thesis*
apply *auto*
apply (*rule-tac B*={(- (*C*,*M*,*pc*)#*cs'*,⌊((*C*,*M*,*Suc pc*)#*cs'*,*False*)⌋ -),
(- (*C*,*M*,*pc*)#*cs'*,⌊(*find-handler-for P NullPointer* ((*C*,*M*,*pc*)#*cs'*),*True*)⌋
-),
(- *fst*(*the*(*x*)),*None* -)} **in** *finite-subset*)
apply (*rule subsetI*)
apply (*clarsimp simp del*: *find-handler-for*.*simps*)
by (*erule JVM-CFG.cases*, *simp-all del*: *find-handler-for*.*simps*)
next
case (*Checkcast C'*)

with *valid-node*
show *?thesis*
apply *auto*
apply (*rule-tac B*={(- (*C*,*M*,*Suc pc*)#*cs*′,*None* -),
(- (*C*,*M*,*pc*)#*cs*′,⌊(*find-handler-for P ClassCast* ((*C*,*M*,*pc*)#*cs*′),*True*)⌋
-),
(- *fst*(*the*(*x*)),*None* -)} **in** *finite-subset*)
apply (*rule subsetI*)
apply (*clarsimp simp del*: *find-handler-for.simps*)
by (*erule JVM-CFG.cases*, *simp-all del*: *find-handler-for.simps*)
next
case (*Invoke M′ n′*)
with *finite-classes valid-node*
show *?thesis*
apply *auto*
apply (*rule-tac B*=
{*n*′. (∃ *D*. *is-class* (P_{wf}) *D* ∧ *n*′ = (- (*C*,*M*,*pc*)#*cs*′,⌊((*D*,*M*′,*0*)#(*C*,*M*,*pc*)#*cs*′,*False*)⌋
-))}
∪ {(- (*C*,*M*,*pc*)#*cs*′,⌊(*find-handler-for P NullPointer* ((*C*,*M*,*pc*)#*cs*′),*True*)⌋
-),
(- *fst*(*the*(*x*)),*None* -)}
in *finite-subset*)
apply (*rule subsetI*)
apply (*clarsimp simp del*: *find-handler-for.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *find-handler-for.simps*)
apply (*clarsimp simp del*: *find-handler-for.simps*)
apply (*drule sees-method-is-class*)
by (*clarsimp simp del*: *find-handler-for.simps*)
next
case *Return*
with *valid-node*
show *?thesis*
apply *auto*
apply (*rule-tac B*=
{(- (*fst*(*hd*(*cs*′)),*fst*(*snd*(*hd*(*cs*′))),*Suc*(*snd*(*snd*(*hd*(*cs*′)))))#(*tl cs*′),*None*
-),
(-*Exit*-)} **in** *finite-subset*)
apply (*rule subsetI*)
apply (*clarsimp simp del*: *find-handler-for.simps*)
by (*erule JVM-CFG.cases*, *simp-all del*: *find-handler-for.simps*)
next
case *Pop*
with *valid-node*
show *?thesis*
apply *auto*
apply (*rule-tac B*={(- (*C*,*M*,*Suc pc*)#*cs*′,*None* -)} **in** *finite-subset*)
by (*auto elim*: *JVM-CFG.cases*)
next
case *IAdd*

```
      with valid-node
      show ?thesis
        apply auto
        apply (rule-tac B={(- (C,M,Suc pc)#cs',None -)} in finite-subset)
         by (auto elim: JVM-CFG.cases)
    next
      case (Goto i)
      with valid-node
      show ?thesis
        apply auto
      apply (rule-tac B={(- (C,M,nat (int pc + i))#cs',None -)} in finite-subset)
         by (auto elim: JVM-CFG.cases)
    next
      case CmpEq
      with valid-node
      show ?thesis
        apply auto
        apply (rule-tac B={(- (C,M,Suc pc)#cs',None -)} in finite-subset)
         by (auto elim: JVM-CFG.cases)
    next
      case (IfFalse i)
      with valid-node
      show ?thesis
        apply auto
        apply (rule-tac B={(- (C,M,Suc pc)#cs',None -),
          (- (C,M,nat (int pc + i))#cs',None -)} in finite-subset)
         by (auto elim: JVM-CFG.cases)
    next
      case Throw
      have finite {(l,pc'). l < Suc (length cs') ∧
        pc' < (∑ i≤(length cs'). (length (instrs-of (P_wf) (fst (((C, M, pc) # cs')
! i))
        (fst (snd (((C, M, pc) # cs') ! i))))))}
        (is finite ?f1)
        by (auto intro: finite-cartesian-product bounded-nat-set-is-finite)
      hence f-1: finite {(l,pc'). l < length ((C, M, pc) # cs') ∧
      pc' < length (instrs-of (P_wf) (fst(((C,M,pc)#cs')!l)) (fst(snd(((C,M,pc)#cs')!l))))}
        apply (rule-tac B=?f1 in finite-subset)
         apply clarsimp
         apply (rule less-le-trans)
          defer
          apply (rule-tac A={a} in sum-mono2)
            by simp-all
      from valid-node Throw
      show ?thesis
        apply auto
        apply (rule-tac B=
          {n'. ∃ Cx Mx pc' h cs'' pcx. (C,M,pc)#cs' = cs''@[(Cx,Mx,pcx)]@h ∧
            pc' < length (instrs-of (P_wf) Cx Mx) ∧
```

```
  n′ = (- (C,M,pc)#cs′,⌊((Cx,Mx,pc′)#h,True)⌋ -)}
 ∪ {(- fst(the(x)),None -), (-Exit-), (- (C,M,pc)#cs′,⌊([],True)⌋ -)}
 in finite-subset)
apply (rule subsetI)
apply clarsimp
apply (erule JVM-CFG.cases, simp-all del: find-handler-for.simps)
apply (clarsimp simp del: find-handler-for.simps)
apply (case-tac find-handler-for P Exc ((C,M,pc)#cs′), simp)
apply (clarsimp simp del: find-handler-for.simps)
apply (erule impE)
 apply (case-tac list, fastforce, fastforce)
apply (frule find-handler-for-tl-eq)
apply (clarsimp simp del: find-handler-for.simps)
apply (erule-tac x=list in allE)
apply (clarsimp simp del: find-handler-for.simps)
apply (subgoal-tac
  finite (
     (λ(Cx,Mx,pc′,h,cs′′,pcx). (- (C, M, pc) # cs′,⌊((Cx, Mx, pc′) # h,
True)⌋ -)) ‘
    {(Cx,Mx,pc′,h,cs′′,pcx). (C, M, pc) # cs′ = cs′′ @ (Cx, Mx, pcx) # h
∧
     pc′ < length (instrs-of P_wf Cx Mx)}))
apply (case-tac ((λ(Cx, Mx, pc′, h, cs′′, pcx).
  (- (C, M, pc) # cs′,⌊((Cx, Mx, pc′) # h, True)⌋ -)) ‘
  {(Cx, Mx, pc′, h, cs′′, pcx).
    (C, M, pc) # cs′ = cs′′ @ (Cx, Mx, pcx) # h ∧
    pc′ < length (instrs-of (P_wf) Cx Mx)}) =
  {n′. ∃ Cx Mx pc′ h.
     (∃ cs′′ pcx. (C, M, pc) # cs′ = cs′′ @ (Cx, Mx, pcx) # h) ∧
     pc′ < length (instrs-of (P_wf) Cx Mx) ∧
     n′ = (- (C, M, pc) # cs′,⌊((Cx, Mx, pc′) # h, True)⌋ -)})
 apply clarsimp
apply (erule notE)
apply (rule equalityI)
 apply clarsimp
apply clarsimp
apply (rule-tac x=(Cx,Mx,pc′,h,cs′′,pcx) in image-eqI)
 apply clarsimp
apply clarsimp
apply (rule finite-imageI)
apply (subgoal-tac finite (
  (λ(l, pc′). (fst(((C, M, pc)#cs′) ! l),
              fst(snd(((C, M, pc)#cs′) ! l)),
              pc′,
              drop l cs′,
              take l ((C, M, pc)#cs′),
              snd(snd(((C, M, pc)#cs′) ! l))
             )
  ) ‘ {(l, pc′). l < length ((C,M,pc)#cs′) ∧
```

pc′ < *length* (*instrs-of* (P_{wf}) (*fst*(((*C*, *M*, *pc*)#*cs′*) ! *l*))
(*fst*(*snd*(((*C*, *M*, *pc*)#*cs′*) ! *l*))))}))
apply (*case-tac* ((λ(*l*, *pc′*).
(*fst* (((*C*, *M*, *pc*) # *cs′*) ! *l*),
fst (*snd* (((*C*, *M*, *pc*) # *cs′*) ! *l*)),
pc′,
drop l cs′,
take l ((*C*, *M*, *pc*) # *cs′*),
snd (*snd* (((*C*, *M*, *pc*) # *cs′*) ! *l*))
)) ‘ {(*l*, *pc′*). *l* < *length* ((*C*,*M*,*pc*)#*cs′*) ∧
pc′ < *length* (*instrs-of* (P_{wf}) (*fst* (((*C*, *M*, *pc*) # *cs′*) ! *l*))
(*fst* (*snd* (((*C*, *M*, *pc*) # *cs′*) ! *l*))))})
= {(*Cx*, *Mx*, *pc′*, *h*, *cs′′*, *pcx*).
(*C*, *M*, *pc*) # *cs′* = *cs′′* @ (*Cx*, *Mx*, *pcx*) # *h* ∧
pc′ < *length* (*instrs-of* (P_{wf}) *Cx Mx*)})
apply *clarsimp*
apply (*erule notE*)
apply (*rule equalityI*)
apply *clarsimp*
apply (*rule id-take-nth-drop* [*of* - (*C*,*M*,*pc*)#*cs′*, *simplified*])
apply *simp*
apply *clarsimp*
apply (*rule-tac x*=(*length ad*,*ab*) **in** *image-eqI*)
apply *clarsimp*
apply (*case-tac ad*, *clarsimp*, *clarsimp*)
apply *clarsimp*
apply (*case-tac ad*, *clarsimp*, *clarsimp*)
apply (*rule finite-imageI*)
by (*rule f-1*)
qed
qed
qed
qed

6.3.3 Interpretation of the locale

interpretation *JVM-CFG-StrongPostdomination*:
StrongPostdomination sourcenode targetnode kind valid-edge$_{CFG}$ *prog Entry* (*-Exit-*)
for *prog*
proof(*unfold-locales*)
fix *n*
assume *vn*: *CFG.valid-node sourcenode targetnode* (*valid-edge*$_{CFG}$ *prog*) *n*
thus *finite* {*n′*. ∃ *a′*. *valid-edge*$_{CFG}$ *prog a′* ∧ *sourcenode a′* = *n* ∧ *targetnode a′* = *n′*}
by (*rule successor-set-finite*)
qed

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

6.4 Instantiation of the *CFG-wf* locale

6.4.1 Variables and Values

datatype *jinja-var* = *HeapVar addr* | *Stk nat nat* | *Loc nat nat*
datatype *jinja-val* = *Object obj option* | *Primitive val*

fun *state-val* :: *state* ⇒ *jinja-var* ⇒ *jinja-val*
where
state-val (*h*, *stk*, *loc*) (*HeapVar a*) = *Object* (*h a*)
| *state-val* (*h*, *stk*, *loc*) (*Stk cd idx*) = *Primitive* (*stk* (*cd*, *idx*))
| *state-val* (*h*, *stk*, *loc*) (*Loc cd idx*) = *Primitive* (*loc* (*cd*, *idx*))

6.4.2 The *Def* and *Use* sets

inductive-set *Def* :: *wf-jvmprog* ⇒ *j-node* ⇒ *jinja-var set*
for *P* :: *wf-jvmprog*
and *n* :: *j-node*
where
Def-Load:
⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
instrs-of (P_{wf}) *C M* ! *pc* = *Load idx*;
cd = *length cs*;
i = *stkLength P C M pc*⟧
⟹ *Stk cd i* ∈ *Def P n*

| *Def-Store*:
⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
instrs-of (P_{wf}) *C M* ! *pc* = *Store idx*;
cd = *length cs* ⟧
⟹ *Loc cd idx* ∈ *Def P n*

| *Def-Push*:
⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
instrs-of (P_{wf}) *C M* ! *pc* = *Push v*;
cd = *length cs*;
i = *stkLength P C M pc* ⟧
⟹ *Stk cd i* ∈ *Def P n*

| *Def-New-Normal-Stk*:
⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,⌊(*cs*′,*False*)⌋ -);
instrs-of (P_{wf}) *C M* ! *pc* = *New Cl*;
cd = *length cs*;
i = *stkLength P C M pc* ⟧
⟹ *Stk cd i* ∈ *Def P n*

| *Def-New-Normal-Heap*:
⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,⌊(*cs*′,*False*)⌋ -);
instrs-of (P_{wf}) *C M* ! *pc* = *New Cl* ⟧
⟹ *HeapVar a* ∈ *Def P n*

| *Def-Exc-Stk*:
$[\![$ $n = (\text{-}\ (C,\ M,\ pc)\#cs, \lfloor (cs',True) \rfloor\ \text{-})$;
$cs' \neq []$;
$cd = length\ cs' - 1$;
$(C',M',pc') = hd\ cs'$;
$i = stkLength\ P\ C'\ M'\ pc' - 1]\!]$
$\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

| *Def-Getfield-Stk*:
$[\![$ $n = (\text{-}\ (C,\ M,\ pc)\#cs, \lfloor (cs',False) \rfloor\ \text{-})$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Getfield\ Fd\ Cl$;
$cd = length\ cs$;
$i = stkLength\ P\ C\ M\ pc - 1\ ]\!]$
$\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

| *Def-Putfield-Heap*:
$[\![$ $n = (\text{-}\ (C,\ M,\ pc)\#cs, \lfloor (cs',False) \rfloor\ \text{-})$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Putfield\ Fd\ Cl\ ]\!]$
$\Longrightarrow HeapVar\ a \in Def\ P\ n$

| *Def-Invoke-Loc*:
$[\![$ $n = (\text{-}\ (C,\ M,\ pc)\#cs, \lfloor (cs',False) \rfloor\ \text{-})$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Invoke\ M'\ n'$;
$cs' \neq []$;
$hd\ cs' = (C',M',0)$;
$i < locLength\ P\ C'\ M'\ 0$;
$cd = Suc\ (length\ cs)\ ]\!]$
$\Longrightarrow Loc\ cd\ i \in Def\ P\ n$

| *Def-Return-Stk*:
$[\![$ $n = (\text{-}\ (C,\ M,\ pc)\#(D,M',pc')\#cs,None\ \text{-})$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Return$;
$cd = length\ cs$;
$i = stkLength\ P\ D\ M'\ (Suc\ pc') - 1\ ]\!]$
$\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

| *Def-IAdd-Stk*:
$[\![$ $n = (\text{-}\ (C,\ M,\ pc)\#cs,None\ \text{-})$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = IAdd$;
$cd = length\ cs$;
$i = stkLength\ P\ C\ M\ pc - 2\ ]\!]$
$\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

| *Def-CmpEq-Stk*:
$[\![$ $n = (\text{-}\ (C,\ M,\ pc)\#cs,None\ \text{-})$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = CmpEq$;
$cd = length\ cs$;
$i = stkLength\ P\ C\ M\ pc - 2\ ]\!]$

$\Longrightarrow$ *Stk cd i* $\in$ *Def P n*

inductive-set *Use* :: *wf-jvmprog* $\Rightarrow$ *j-node* $\Rightarrow$ *jinja-var set*
for *P* :: *wf-jvmprog*
and *n* :: *j-node*
where
Use-Load:
$[\![$ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
instrs-of (P_{wf}) *C M* ! *pc* = *Load idx*;
cd = *length cs* $]\!]$
$\Longrightarrow$ (*Loc cd idx*) $\in$ *Use P n*

| *Use-Store*:
$[\![$ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
instrs-of (P_{wf}) *C M* ! *pc* = *Store idx*;
cd = *length cs*;
Suc i = (*stkLength P C M pc*) $]\!]$
$\Longrightarrow$ (*Stk cd i*) $\in$ *Use P n*

| *Use-New*:
$[\![$ *n* = (- (*C*, *M*, *pc*)#*cs*,*x* -);
x = *None* $\vee$ *x* = $\lfloor$(*cs'*,*False*)$\rfloor$;
instrs-of (P_{wf}) *C M* ! *pc* = *New Cl* $]\!]$
$\Longrightarrow$ *HeapVar a* $\in$ *Use P n*

| *Use-Getfield-Stk*:
$[\![$ *n* = (- (*C*, *M*, *pc*)#*cs*,*x* -);
x = *None* $\vee$ *x* = $\lfloor$(*cs'*,*False*)$\rfloor$;
instrs-of (P_{wf}) *C M* ! *pc* = *Getfield Fd Cl*;
cd = *length cs*;
Suc i = *stkLength P C M pc* $]\!]$
$\Longrightarrow$ *Stk cd i* $\in$ *Use P n*

| *Use-Getfield-Heap*:
$[\![$ *n* = (- (*C*, *M*, *pc*)#*cs*,$\lfloor$(*cs'*,*False*)$\rfloor$ -);
instrs-of (P_{wf}) *C M* ! *pc* = *Getfield Fd Cl* $]\!]$
$\Longrightarrow$ *HeapVar a* $\in$ *Use P n*

| *Use-Putfield-Stk-Pred*:
$[\![$ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
instrs-of (P_{wf}) *C M* ! *pc* = *Putfield Fd Cl*;
cd = *length cs*;
i = *stkLength P C M pc* − *2* $]\!]$
$\Longrightarrow$ *Stk cd i* $\in$ *Use P n*

| *Use-Putfield-Stk-Update*:
$[\![$ *n* = (- (*C*, *M*, *pc*)#*cs*,$\lfloor$(*cs'*,*False*)$\rfloor$ -);
instrs-of (P_{wf}) *C M* ! *pc* = *Putfield Fd Cl*;
cd = *length cs*;

$i = stkLength\ P\ C\ M\ pc - 2 \lor i = stkLength\ P\ C\ M\ pc - 1$ ⟧
$\Longrightarrow Stk\ cd\ i \in Use\ P\ n$

| *Use-Putfield-Heap*:
⟦ $n = (\_\ (C, M, pc)\#cs,\lfloor(cs',False)\rfloor\ \_)$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Putfield\ Fd\ Cl$ ⟧
$\Longrightarrow HeapVar\ a \in Use\ P\ n$

| *Use-Checkcast-Stk*:
⟦ $n = (\_\ (C, M, pc)\#cs,x\ \_)$;
$x = None \lor x = \lfloor(cs',False)\rfloor$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Checkcast\ Cl$;
$cd = length\ cs$;
$i = stkLength\ P\ C\ M\ pc - Suc\ 0$ ⟧
$\Longrightarrow Stk\ cd\ i \in Use\ P\ n$

| *Use-Checkcast-Heap*:
⟦ $n = (\_\ (C, M, pc)\#cs,None\ \_)$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Checkcast\ Cl$ ⟧
$\Longrightarrow HeapVar\ a \in Use\ P\ n$

| *Use-Invoke-Stk-Pred*:
⟦ $n = (\_\ (C, M, pc)\#cs,None\ \_)$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Invoke\ M'\ n'$;
$cd = length\ cs$;
$i = stkLength\ P\ C\ M\ pc - Suc\ n'$ ⟧
$\Longrightarrow Stk\ cd\ i \in Use\ P\ n$

| *Use-Invoke-Heap-Pred*:
⟦ $n = (\_\ (C, M, pc)\#cs,None\ \_)$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Invoke\ M'\ n'$ ⟧
$\Longrightarrow HeapVar\ a \in Use\ P\ n$

| *Use-Invoke-Stk-Update*:
⟦ $n = (\_\ (C, M, pc)\#cs,\lfloor(cs',False)\rfloor\ \_)$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Invoke\ M'\ n'$;
$cd = length\ cs$;
$i < stkLength\ P\ C\ M\ pc$;
$i \geq stkLength\ P\ C\ M\ pc - Suc\ n'$ ⟧
$\Longrightarrow Stk\ cd\ i \in Use\ P\ n$

| *Use-Return-Stk*:
⟦ $n = (\_\ (C, M, pc)\#(D,M',pc')\#cs,None\ \_)$;
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Return$;
$cd = Suc\ (length\ cs)$;
$i = stkLength\ P\ C\ M\ pc - 1$ ⟧
$\Longrightarrow Stk\ cd\ i \in Use\ P\ n$

| *Use-IAdd-Stk*:

⟦ n = (- (C, M, pc)#cs,None -);
instrs-of (P_{wf}) C M ! pc = IAdd;
cd = length cs;
i = stkLength P C M pc − 1 ∨ i = stkLength P C M pc − 2 ⟧
⟹ Stk cd i ∈ Use P n

| *Use-IfFalse-Stk*:
⟦ n = (- (C, M, pc)#cs,None -);
instrs-of (P_{wf}) C M ! pc = (IfFalse b);
cd = length cs;
i = stkLength P C M pc − 1 ⟧
⟹ Stk cd i ∈ Use P n

| *Use-CmpEq-Stk*:
⟦ n = (- (C, M, pc)#cs,None -);
instrs-of (P_{wf}) C M ! pc = CmpEq;
cd = length cs;
i = stkLength P C M pc − 1 ∨ i = stkLength P C M pc − 2 ⟧
⟹ Stk cd i ∈ Use P n

| *Use-Throw-Stk*:
⟦ n = (- (C, M, pc)#cs,x -);
x = None ∨ x = ⌊(cs′,True)⌋;
instrs-of (P_{wf}) C M ! pc = Throw;
cd = length cs;
i = stkLength P C M pc − 1 ⟧
⟹ Stk cd i ∈ Use P n

| *Use-Throw-Heap*:
⟦ n = (- (C, M, pc)#cs,x -);
x = None ∨ x = ⌊(cs′,True)⌋;
instrs-of (P_{wf}) C M ! pc = Throw ⟧
⟹ HeapVar a ∈ Use P n

declare *correct-state-def* [*simp del*]

lemma *edge-transfer-uses-only-Use*:
⟦*valid-edge* (P,C0,Main) a; ∀ V ∈ Use P (*sourcenode* a). *state-val* s V = *state-val* s′ V⟧
⟹ ∀ V ∈ Def P (*sourcenode* a). *state-val* (*BasicDefs.transfer* (*kind* a) s) V =
state-val (*BasicDefs.transfer* (*kind* a) s′) V
proof
fix V
assume *ve*: *valid-edge* (P, C0, Main) a
and *use-eq*: ∀ V∈Use P (*sourcenode* a). *state-val* s V = *state-val* s′ V
and *v-in-def*: V ∈ Def P (*sourcenode* a)
obtain h stk loc **where** [*simp*]: s = (h,stk,loc) **by** (*cases* s, *fastforce*)
obtain h′ stk′ loc′ **where** [*simp*]: s′ = (h′,stk′,loc′) **by** (*cases* s′, *fastforce*)
note P-wf = *wf-jvmprog-is-wf* [*of* P]

from *ve*
have *ex-edge*: (*P*,*C0*,*Main*) ⊢ (*sourcenode a*)−*kind a*→(*targetnode a*)
and *vn*: *valid-node* (*P*,*C0*,*Main*) (*sourcenode a*)
by *simp-all*
show *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val* (*transfer* (*kind a*) *s'*) *V*
proof (*cases sourcenode a*)
case [*simp*]: (*Node cs x*)
from *vn ex-edge* **have** *cs* ≠ []
by (*cases x*, *auto elim*: *JVM-CFG.cases*)
then obtain *C M pc cs'* **where** [*simp*]: *cs* = (*C*, *M*, *pc*)#*cs'* **by** (*cases cs*, *fastforce*+)
with *vn* **obtain** *ST LT* **where** *wt*: ((P_{Φ}) *C M* ! *pc*) = ⌊(*ST*,*LT*)⌋
by (*cases cs'*, (*cases x*, *auto*)+)
show *?thesis*
proof (*cases instrs-of* (P_{wf}) *C M* ! *pc*)
case [*simp*]: (*Load n*)
from *ex-edge* **have** [*simp*]: *x* = *None*
by (*auto elim*!: *JVM-CFG.cases*)
hence *Loc* (*length cs'*) *n* ∈ *Use P* (*sourcenode a*)
by (*auto intro*!: *Use-Load*)
with *use-eq* **have** *state-val s* (*Loc* (*length cs'*) *n*) = *state-val s'* (*Loc* (*length cs'*) *n*)
by (*simp del*: *state-val.simps*)
with *v-in-def ex-edge* **show** *?thesis*
by (*auto elim*!: *Def.cases*
elim: *JVM-CFG.cases*
simp: *split-beta*)
next
case [*simp*]: (*Store n*)
from *ex-edge* **have** [*simp*]:*x* = *None*
by (*auto elim*!: *JVM-CFG.cases*)
have *ST* ≠ []
proof −
from *vn*
obtain *Ts T mxs mxl is xt*
where *C-sees-M*: P_{wf} ⊢ *C sees M*: *Ts*→*T* = (*mxs*, *mxl*, *is*, *xt*) *in C*
by (*cases cs'*, *auto*)
with *vn*
have *pc* < *length is*
by (*cases cs'*, *auto dest*: *sees-method-fun*)
from *P-wf C-sees-M*
have *wt-method* (P_{wf}) *C Ts T mxs mxl is xt* (P_{Φ} *C M*)
by (*auto dest*: *sees-wf-mdecl simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
with *Store C-sees-M wt* ‹*pc* < *length is*›
show *?thesis*
by (*fastforce simp*: *wt-method-def*)
qed
then obtain *ST1 STr* **where** [*simp*]: *ST* = *ST1*#*STr*
by (*cases ST*, *fastforce*+)

from *wt*
have *Stk* (*length cs′*) (*length ST − 1*) ∈ *Use P* (*sourcenode a*)
(**is** *?stk-top* ∈ *?Use-src*)
by −(*rule Use-Store*, *fastforce*+)
with *use-eq* **have** *state-val s ?stk-top* = *state-val s′ ?stk-top*
by (*simp del*: *state-val.simps*)
with *v-in-def ex-edge wt* **show** *?thesis*
by (*auto elim*!: *Def.cases*
elim: *JVM-CFG.cases*
simp: *split-beta*)
next
case [*simp*]: (*Push val*)
from *ex-edge* **have** *x* = *None*
by (*auto elim*!: *JVM-CFG.cases*)
with *v-in-def ex-edge* **show** *?thesis*
by (*auto elim*!: *Def.cases*
elim: *JVM-CFG.cases*)
next
case [*simp*]: (*New Cl*)
show *?thesis*
proof (*cases x*)
case *None*
with *v-in-def* **have** *False*
by (*auto elim*: *Def.cases*)
thus *?thesis* **by** *simp*
next
case (*Some x′*)
then obtain *cs′′ xf* **where** [*simp*]: *x* = ⌊(*cs′′*,*xf*)⌋
by (*cases x′*, *fastforce*)
have ¬ *xf* ⟶ (∀ *addr*. *HeapVar addr* ∈ *Use P* (*sourcenode a*))
by (*fastforce intro*: *Use-New*)
with *use-eq*
have ¬ *xf* ⟶ (∀ *addr*. *state-val s* (*HeapVar addr*) = *state-val s′* (*HeapVar*
addr))
by (*simp del*: *state-val.simps*)
hence ¬ *xf* ⟶ *h* = *h′*
by (*auto intro*: *ext*)
with *v-in-def ex-edge* **show** *?thesis*
by (*auto elim*!: *Def.cases*
elim: *JVM-CFG.cases*)
qed
next
case [*simp*]: (*Getfield Fd Cl*)
show *?thesis*
proof (*cases x*)
case *None*
with *v-in-def* **have** *False*
by (*auto elim*: *Def.cases*)
thus *?thesis* **by** *simp*

next
case (*Some x′*)
then obtain *cs′′ xf* **where** [*simp*]: $x = \lfloor(cs'',xf)\rfloor$
by (*cases x′*, *fastforce*)
have $ST \neq []$
proof −
from *vn* **obtain** *T Ts mxs mxl is xt*
where *sees-M*: $(P_{wf}) \vdash C$ *sees M*:*Ts*→*T* = (*mxs*,*mxl*,*is*,*xt*) *in C*
by (*cases cs′*, *auto*)
with *vn*
have *pc* < *length is*
by (*cases cs′*, *auto dest*: *sees-method-fun*)
from *P-wf sees-M* **have** *wt-method* (P_{wf}) *C Ts T mxs mxl is xt* $(P_\Phi$ *C M*)
by (*auto dest*: *sees-wf-mdecl simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
with *Getfield sees-M wt* ‹*pc* < *length is*› **show** *?thesis*
by (*fastforce simp*: *wt-method-def*)
qed
then obtain *ST1 STr* **where** [*simp*]: *ST* = *ST1#STr* **by** (*cases ST*, *fastforce*)
from *wt*
have ¬ *xf* ⟶ (*Stk* (*length cs′*) (*length ST* − *1*) ∈ *Use P* (*sourcenode a*))
(**is** *?xf* ⟶ *?stk-top* ∈ *?Use-src*)
by (*auto intro*!: *Use-Getfield-Stk*)
with *use-eq*
have *stk-top-eq*: ¬ *xf* ⟶ *state-val s ?stk-top* = *state-val s′ ?stk-top*
by (*simp del*: *state-val.simps*)
have ¬ *xf* ⟶ (∀ *addr*. *HeapVar addr* ∈ *Use P* (*sourcenode a*))
by (*auto intro*!: *Use-Getfield-Heap*)
with *use-eq*
have ¬ *xf* ⟶ (∀ *addr*. *state-val s* (*HeapVar addr*) = *state-val s′* (*HeapVar addr*))
by (*simp del*: *state-val.simps*)
hence ¬ *xf* ⟶ *h* = *h′*
by (*auto intro*: *ext*)
with *ex-edge v-in-def stk-top-eq wt*
show *?thesis*
by (*auto elim*!: *Def.cases*
elim: *JVM-CFG.cases*
simp: *split-beta*)
qed
next
case [*simp*]: (*Putfield Fd Cl*)
show *?thesis*
proof (*cases x*)
case *None*
with *v-in-def* **have** *False*
by (*auto elim*: *Def.cases*)
thus *?thesis* **by** *simp*

next
case (*Some x'*)
then obtain *cs'' xf* **where** [*simp*]: $x = \lfloor(cs'',xf)\rfloor$
by (*cases x'*, *fastforce*)
have *length ST* > *1*
proof −
from *vn* **obtain** *T Ts mxs mxl is xt*
where *sees-M*: $(P_{wf}) \vdash C$ *sees M*:*Ts*→*T* = (*mxs*,*mxl*,*is*,*xt*) *in C*
by (*cases cs'*, *auto*)
with *vn*
have *pc* < *length is*
by (*cases cs'*, *auto dest*: *sees-method-fun*)
from *P-wf sees-M* **have** *wt-method* (P_{wf}) *C Ts T mxs mxl is xt* $(P_{\Phi}$ *C M*)
by (*auto dest*: *sees-wf-mdecl simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
with *Putfield sees-M* ‹*pc* < *length is*› *wt* **show** *?thesis*
by (*fastforce simp*: *wt-method-def*)
qed
then obtain *ST1 STr'* **where** *ST* = *ST1*#*STr'* ∧ *length STr'* > *0*
by (*cases ST*, *fastforce*+)
then obtain *ST2 STr* **where** [*simp*]: *ST* = *ST1*#*ST2*#*STr*
by (*cases STr'*, *fastforce*+)
from *wt*
have ¬ *xf* ⟶ (*Stk* (*length cs'*) (*length ST* − *1*) ∈ *Use P* (*sourcenode a*))
(**is** *?xf* ⟶ *?stk-top* ∈ *?Use-src*)
by (*fastforce intro*: *Use-Putfield-Stk-Update*)
with *use-eq* **have** *stk-top*:(¬ *xf*) ⟶ *state-val s ?stk-top* = *state-val s' ?stk-top*
by (*simp del*: *state-val.simps*)
from *wt*
have ¬ *xf* ⟶ (*Stk* (*length cs'*) (*length ST* − *2*) ∈ *Use P* (*sourcenode a*))
(**is** *?xf* ⟶ *?stk-nxt* ∈ *?Use-src*)
by (*fastforce intro*: *Use-Putfield-Stk-Update*)
with *use-eq*
have *stk-nxt*:(¬ *xf*) ⟶ *state-val s ?stk-nxt* = *state-val s' ?stk-nxt*
by (*simp del*: *state-val.simps*)
have ¬ *xf* ⟶ (∀ *addr*. *HeapVar addr* ∈ *Use P* (*sourcenode a*))
by (*fastforce intro*: *Use-Putfield-Heap*)
with *use-eq*
have ¬ *xf* ⟶ (∀ *addr*. *state-val s* (*HeapVar addr*) = *state-val s'* (*HeapVar addr*))
by (*simp del*: *state-val.simps*)
hence ¬ *xf* ⟶ *h* = *h'*
by (*auto intro*: *ext*)
with *ex-edge v-in-def stk-top stk-nxt wt* **show** *?thesis*
by (*auto elim*!: *Def.cases*
elim: *JVM-CFG.cases*
simp: *split-beta*)
qed

```
next
  case [simp]: (Checkcast Cl)
  show ?thesis
  proof (cases x)
    case None
    with v-in-def have False
      by (auto elim: Def.cases)
    thus ?thesis by simp
  next
    case (Some x')
    with ex-edge obtain cs''
      where x = ⌊(cs'',True)⌋
      by (auto elim!: JVM-CFG.cases)
    with v-in-def ex-edge show ?thesis
      by (auto elim!: Def.cases
               elim: JVM-CFG.cases)
  qed
next
  case [simp]: (Invoke M' n')
  show ?thesis
  proof (cases x)
    case None
    with v-in-def have False
      by (auto elim: Def.cases)
    thus ?thesis by simp
  next
    case (Some x')
    then obtain cs'' xf where [simp]: x = ⌊(cs'',xf)⌋
      by (cases x', fastforce)
    show ?thesis
    proof (cases xf)
      case True
      with v-in-def ex-edge show ?thesis
        by (auto elim!: Def.cases
                 elim: JVM-CFG.cases)
    next
      case [simp]: False
      have length ST > n'
      proof -
        from vn obtain T Ts mxs mxl is xt
          where sees-M: (P_wf) ⊢ C sees M:Ts→T = (mxs,mxl,is,xt) in C
          by (cases cs', auto)
        with vn
        have pc < length is
          by (cases cs', auto dest: sees-method-fun)
        from P-wf sees-M have wt-method (P_wf) C Ts T mxs mxl is xt (P_Φ C
M)
          by (auto dest: sees-wf-mdecl simp: wf-jvm-prog-phi-def wf-mdecl-def)
        with Invoke sees-M ‹pc < length is› wt show ?thesis
```

        by (*fastforce simp*: *wt-method-def*)
      qed
      moreover obtain *STn* **where** *STn* = *take n′ ST* **by** *fastforce*
      moreover obtain *STs* **where** *STs* = *ST* ! *n′* **by** *fastforce*
      moreover obtain *STr* **where** *STr* = *drop* (*Suc n′*) *ST* **by** *fastforce*
      ultimately have [*simp*]: *ST* = *STn*@*STs*#*STr* ∧ *length STn* = *n′*
        by (*auto simp*: *id-take-nth-drop*)
      from *wt*
        have ∀ *i*. *i* ≤ *n′* ⟶ *Stk* (*length cs′*) (*length ST* − *Suc i*) ∈ *Use P*
(*sourcenode a*)
        by (*fastforce intro*: *Use-Invoke-Stk-Update*)
      with *use-eq*
      have
        ∀ *i*. *i* ≤ *n′* ⟶ *state-val s* (*Stk* (*length cs′*) (*length ST* − *Suc i*)) =
                state-val s′ (*Stk* (*length cs′*) (*length ST* − *Suc i*))
        by (*simp del*: *state-val.simps*)
      hence *stk-eq*:
        ∀ *i*. *i* ≤ *n′* ⟶ *state-val s* (*Stk* (*length cs′*) (*i* + *length STr*)) =
                state-val s′ (*Stk* (*length cs′*) (*i* + *length STr*))
        by (*clarsimp*, *erule-tac x*=*n′* − *i* **in** *allE*, *auto simp*: *add.commute*)
      from *ex-edge* **obtain** *C′*
        where *trg*: *targetnode a* = (- (*C′*,*M′*,*0*)#(*C*, *M*, *pc*)#*cs′*,*None* -)
        by (*fastforce elim*: *JVM-CFG.cases*)
      with *ex-edge stk-eq v-in-def wt*
      show *?thesis*
        by (*auto elim*!: *Def.cases*) (*erule JVM-CFG.cases*, *auto simp*: *split-beta*
add.commute)
    qed
  qed
next
  case [*simp*]: *Return*
  show *?thesis*
  proof (*cases x*)
    case [*simp*]: *None*
    show *?thesis*
    proof (*cases cs′*)
      case *Nil*
      with *v-in-def* **show** *?thesis*
        by (*auto elim*!: *Def.cases*)
    next
      case (*Cons aa list*)
      then obtain *C′ M′ pc′ cs″* **where** [*simp*]: *cs′* = (*C′*,*M′*,*pc′*)#*cs″*
        by (*cases aa*, *fastforce*)
      from *wt*
      have *Stk* (*length cs′*) (*length ST* − *1*) ∈ *Use P* (*sourcenode a*)
        by (*fastforce intro*: *Use-Return-Stk*)
      with *use-eq*
      have *state-val s* (*Stk* (*length cs′*) (*length ST* − *1*)) =
          state-val s′ (*Stk* (*length cs′*) (*length ST* − *1*))

```
        by (simp del: state-val.simps)
      with v-in-def ex-edge wt show ?thesis
        by (auto elim!: Def.cases
                 elim: JVM-CFG.cases
                 simp: split-beta)
    qed
  next
    case (Some x')
    with ex-edge show ?thesis
      by (auto elim: JVM-CFG.cases)
  qed
next
  case Pop
  with v-in-def ex-edge show ?thesis
    by (auto elim!: Def.cases elim: JVM-CFG.cases)
next
  case [simp]: IAdd
  show ?thesis
  proof (cases x)
    case [simp]: None
    from wt
    have Stk (length cs') (length ST − 1) ∈ Use P (sourcenode a)
      (is ?stk-top ∈ ?Use)
      by (auto intro!: Use-IAdd-Stk)
    with use-eq
    have stk-top:state-val s ?stk-top = state-val s' ?stk-top
      by (simp del: state-val.simps)
    from wt
    have Stk (length cs') (length ST − 2) ∈ Use P (sourcenode a)
      (is ?stk-snd ∈ ?Use)
      by (auto intro!: Use-IAdd-Stk)
    with use-eq
    have stk-snd:state-val s ?stk-snd = state-val s' ?stk-snd
      by (simp del: state-val.simps)
    with v-in-def ex-edge stk-top wt show ?thesis
      by (auto elim!: Def.cases
               elim: JVM-CFG.cases
               simp: split-beta)
  next
    case (Some x')
    with ex-edge show ?thesis
      by (auto elim: JVM-CFG.cases)
  qed
next
  case [simp]: (IfFalse b)
  show ?thesis
  proof (cases x)
    case [simp]: None
    from wt
```

have *Stk* (*length cs'*) (*length ST* − *1*) ∈ *Use P* (*sourcenode a*)
(**is** *?stk-top* ∈ *?Use*)
by (*auto intro*!: *Use-IfFalse-Stk*)
with *use-eq*
have *stk-top*:*state-val s ?stk-top* = *state-val s' ?stk-top*
by (*simp del*: *state-val.simps*)
with *v-in-def ex-edge wt* **show** *?thesis*
by (*auto elim*!: *Def.cases*)
next
case (*Some x'*)
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
qed
next
case [*simp*]: *CmpEq*
show *?thesis*
proof (*cases x*)
case [*simp*]: *None*
have *Stk* (*length cs'*) (*stkLength P C M pc* − *1*) ∈ *Use P* (*sourcenode a*)
(**is** *?stk-top* ∈ *?Use*)
by (*auto intro*!: *Use-CmpEq-Stk*)
with *use-eq*
have *stk-top*:*state-val s ?stk-top* = *state-val s' ?stk-top*
by (*simp del*: *state-val.simps*)
have *Stk* (*length cs'*) (*stkLength P C M pc* − *2*) ∈ *Use P* (*sourcenode a*)
(**is** *?stk-snd* ∈ *?Use*)
by (*auto intro*!: *Use-CmpEq-Stk*)
with *use-eq*
have *stk-snd*:*state-val s ?stk-snd* = *state-val s' ?stk-snd*
by (*simp del*: *state-val.simps*)
with *v-in-def ex-edge stk-top wt* **show** *?thesis*
by (*auto elim*!: *Def.cases*
elim: *JVM-CFG.cases*)
next
case (*Some x'*)
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
qed
next
case (*Goto i*)
with *ex-edge v-in-def* **show** *?thesis*
by (*auto elim*!: *Def.cases*
elim: *JVM-CFG.cases*)
next
case [*simp*]: *Throw*
show *?thesis*
proof (*cases x*)
case [*simp*]: *None*
have *Stk* (*length cs'*) (*stkLength P C M pc* − *1*) ∈ *Use P* (*sourcenode a*)

(**is** *?stk-top* ∈ *?Use*)
by (*auto intro*!: *Use-Throw-Stk*)
with *use-eq*
have *stk-top*:*state-val s ?stk-top* = *state-val s′ ?stk-top*
by (*simp del*: *state-val.simps*)
with *v-in-def* **show** *?thesis*
by (*auto elim*!: *Def.cases*)
next
case (*Some x′*)
then obtain *cs′′ xf* **where** [*simp*]: *x* = ⌊(*cs′′*,*xf*)⌋
by (*cases x′*, *fastforce*)
hence *xf* ⟶ *Stk* (*length cs′*) (*stkLength P C M pc* − *1*) ∈ *Use P* (*sourcenode a*)
(**is** *xf* ⟶ *?stk-top* ∈ *?Use*)
by (*fastforce intro*: *Use-Throw-Stk*)
with *use-eq*
have *stk-top*:*xf* ⟶ *state-val s ?stk-top* = *state-val s′ ?stk-top*
by (*simp del*: *state-val.simps*)
with *v-in-def ex-edge* **show** *?thesis*
by (*auto elim*!: *Def.cases*
elim: *JVM-CFG.cases*)
qed
qed
next
case *Entry*
with *vn v-in-def* **show** *?thesis*
by −(*erule Def.cases*, *auto*)
qed
qed

lemma *CFG-edge-Uses-pred-equal*:
⟦ *valid-edge* (*P*,*C0*,*Main*) *a*;
pred (*kind a*) *s*;
∀ *V* ∈ *Use P* (*sourcenode a*). *state-val s V* = *state-val s′ V*⟧
⟹ *pred* (*kind a*) *s′*
proof −
assume *ve*: *valid-edge* (*P*,*C0*,*Main*) *a*
and *pred*: *pred* (*kind a*) *s*
and *use-eq*: ∀ *V*∈*Use P* (*sourcenode a*). *state-val s V* = *state-val s′ V*
obtain *h stk loc* **where** [*simp*]: *s* = (*h*,*stk*,*loc*) **by** (*cases s*, *blast*)
obtain *h′ stk′ loc′* **where** [*simp*]: *s′* = (*h′*,*stk′*,*loc′*) **by** (*cases s′*, *blast*)
from *ve*
have *vn*: *valid-node* (*P*,*C0*,*Main*) (*sourcenode a*)
and *ex-edge*: (*P*,*C0*,*Main*) ⊢ (*sourcenode a*)−*kind a*→(*targetnode a*)
by *simp-all*
note *P-wf* = *wf-jvmprog-is-wf* [*of P*]
show *pred* (*kind a*) *s′*
proof (*cases sourcenode a*)
case [*simp*]: (*Node cs x*)

from *ve* **have** $cs \neq []$
by (*cases x*, *auto elim*: *JVM-CFG.cases*)
then obtain *C M pc cs'* **where** [*simp*]: $cs = (C, M, pc)\#cs'$ **by** (*cases cs*, *fastforce*+)
from *vn* **obtain** *ST LT* **where** *wt*: $((P_\Phi)\ C\ M\ !\ pc) = \lfloor(ST,LT)\rfloor$
by (*cases cs'*, (*cases x*, *auto*)+)
show *?thesis*
proof (*cases instrs-of* (P_{wf}) *C M* ! *pc*)
case (*Load nat*)
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
next
case (*Store nat*)
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
next
case (*Push val*)
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
next
case [*simp*]: (*New Cl*)
show *?thesis*
proof (*cases x*)
case *None*
hence ∀ *addr*. *HeapVar addr* ∈ *Use P* (*sourcenode a*)
by (*auto intro*!: *Use-New*)
with *use-eq* **have** ∀ *addr*. *state-val s* (*HeapVar addr*) = *state-val s'* (*HeapVar addr*)
by (*simp del*: *state-val.simps*)
hence $h = h'$
by (*auto intro*: *ext*)
with *ex-edge pred* **show** *?thesis*
by (*auto elim*!: *JVM-CFG.cases*)
next
case (*Some x'*)
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
qed
next
case [*simp*]: (*Getfield Fd Cl*)
have $ST \neq []$
proof −
from *vn* **obtain** *T Ts mxs mxl is xt*
where *sees-M*: $(P_{wf}) \vdash C$ *sees* $M{:}Ts \rightarrow T = (mxs,mxl,is,xt)$ *in C*
by (*cases cs'*, (*cases x*, *auto*)+)
with *vn*
have *pc* < *length is*
by (*cases cs'*, (*cases x*, *auto dest*: *sees-method-fun*)+)
from *P-wf sees-M* **have** *wt-method* (P_{wf}) *C Ts T mxs mxl is xt* $(P_\Phi\ C\ M)$

by (*auto dest*: *sees-wf-mdecl simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
with *Getfield wt sees-M* ‹*pc* < *length is*› **show** *?thesis*
by (*fastforce simp*: *wt-method-def*)
qed
then obtain *ST1 STr* **where** [*simp*]: *ST* = *ST1#STr* **by** (*cases ST*, *fastforce+*)
show *?thesis*
proof (*cases x*)
case [*simp*]: *None*
from *wt*
have *Stk* (*length cs'*) (*length ST* − *1*) ∈ *Use P* (*sourcenode a*)
(**is** *?stk-top* ∈ *?Use*)
by (*fastforce intro*: *Use-Getfield-Stk*)
with *use-eq* **have** *state-val s ?stk-top* = *state-val s' ?stk-top*
by (*simp del*: *state-val.simps*)
with *ex-edge pred wt* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
next
case (*Some x'*)
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
qed
next
case [*simp*]: (*Putfield Fd Cl*)
have *length ST* > *1*
proof −
from *vn* **obtain** *T Ts mxs mxl is xt*
where *sees-M*: (P_{wf}) ⊢ *C sees M*:*Ts*→*T* = (*mxs*,*mxl*,*is*,*xt*) *in C*
by (*cases cs'*, (*cases x*, *auto*)+)
with *vn*
have *pc* < *length is*
by (*cases cs'*, (*cases x*, *auto dest*: *sees-method-fun*)+)
from *P-wf sees-M* **have** *wt-method* (P_{wf}) *C Ts T mxs mxl is xt* (P_Φ *C M*)
by (*auto dest*: *sees-wf-mdecl simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
with *Putfield wt sees-M* ‹*pc* < *length is*› **show** *?thesis*
by (*fastforce simp*: *wt-method-def*)
qed
then obtain *ST1 STr'* **where** *ST* = *ST1#STr'* ∧ *STr'* ≠ [] **by** (*cases ST*, *fastforce+*)
then obtain *ST2 STr* **where** [*simp*]: *ST* = *ST1#ST2#STr* **by** (*cases STr'*, *fastforce+*)
show *?thesis*
proof (*cases x*)
case [*simp*]: *None*
with *wt*
have *Stk* (*length cs'*) (*length ST* − *2*) ∈ *Use P* (*sourcenode a*)
(**is** *?stk-top* ∈ *?Use*)
by (*fastforce intro*: *Use-Putfield-Stk-Pred*)
with *use-eq* **have** *state-val s ?stk-top* = *state-val s' ?stk-top*

by (*simp del*: *state-val.simps*)
with *ex-edge pred wt* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
next
case (*Some x'*)
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
qed
next
case [*simp*]: (*Checkcast Cl*)
have $ST \neq []$
proof −
from *vn* **obtain** *T Ts mxs mxl is xt*
where *sees-M*: $(P_{wf}) \vdash C$ *sees* $M{:}Ts{\rightarrow}T = (mxs,mxl,is,xt)$ *in C*
by (*cases cs'*, (*cases x*, *auto*)+)
with *vn*
have *pc* < *length is*
by (*cases cs'*, (*cases x*, *auto dest*: *sees-method-fun*)+)
from *P-wf sees-M* **have** *wt-method* (P_{wf}) *C Ts T mxs mxl is xt* $(P_{\Phi}$ *C M*)
by (*auto dest*: *sees-wf-mdecl simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
with *Checkcast wt sees-M* ‹*pc* < *length is*› **show** *?thesis*
by (*fastforce simp*: *wt-method-def*)
qed
then obtain *ST1 STr* **where** [*simp*]: *ST* = *ST1#STr* **by** (*cases ST*, *fastforce*+)
show *?thesis*
proof (*cases x*)
case [*simp*]: *None*
from *wt*
have *Stk* (*length cs'*) (*stkLength P C M pc* − *Suc 0*) ∈ *Use P* (*sourcenode a*)
(**is** *?stk-top* ∈ *?Use*)
by (*fastforce intro*: *Use-Checkcast-Stk*)
with *use-eq*
have *stk-top*: *state-val s ?stk-top* = *state-val s' ?stk-top*
by (*simp del*: *state-val.simps*)
have ∀ *addr*. *HeapVar addr* ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Checkcast-Heap*)
with *use-eq*
have ∀ *addr*. *state-val s* (*HeapVar addr*) = *state-val s'* (*HeapVar addr*)
by (*simp del*: *state-val.simps*)
hence *h* = *h'*
by (*auto intro*: *ext*)
with *ex-edge stk-top pred wt* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
next
case (*Some x'*)
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)

qed
next
case [*simp*]: (*Invoke M′ n′*)
have *length ST* > *n′*
proof −
from *vn* **obtain** *T Ts mxs mxl is xt*
where *sees-M*: (P_{wf}) ⊢ *C sees M*:*Ts*→*T* = (*mxs*,*mxl*,*is*,*xt*) *in C*
by (*cases cs′*, (*cases x*, *auto*)+)
with *vn*
have *pc* < *length is*
by (*cases cs′*, (*cases x*, *auto dest*: *sees-method-fun*)+)
from *P-wf sees-M* **have** *wt-method* (P_{wf}) *C Ts T mxs mxl is xt* (P_Φ *C M*)
by (*auto dest*: *sees-wf-mdecl simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
with *Invoke wt sees-M* ‹*pc* < *length is*› **show** *?thesis*
by (*fastforce simp*: *wt-method-def*)
qed
moreover obtain *STn* **where** *STn* = *take n′ ST* **by** *fastforce*
moreover obtain *STs* **where** *STs* = *ST* ! *n′* **by** *fastforce*
moreover obtain *STr* **where** *STr* = *drop* (*Suc n′*) *ST* **by** *fastforce*
ultimately have [*simp*]: *ST* = *STn*@*STs*#*STr* ∧ *length STn* = *n′*
by (*auto simp*: *id-take-nth-drop*)
show *?thesis*
proof (*cases x*)
case [*simp*]: *None*
with *wt*
have *Stk* (*length cs′*) (*stkLength P C M pc* − *Suc n′*) ∈ *Use P* (*sourcenode a*)
(**is** *?stk-top* ∈ *?Use*)
by (*fastforce intro*: *Use-Invoke-Stk-Pred*)
with *use-eq*
have *stk-top*: *state-val s ?stk-top* = *state-val s′ ?stk-top*
by (*simp del*: *state-val.simps*)
have ∀ *addr*. *HeapVar addr* ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Invoke-Heap-Pred*)
with *use-eq*
have ∀ *addr*. *state-val s* (*HeapVar addr*) = *state-val s′* (*HeapVar addr*)
by (*simp del*: *state-val.simps*)
hence *h* = *h′*
by (*auto intro*: *ext*)
with *ex-edge stk-top pred wt* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
next
case (*Some x′*)
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
qed
next
case *Return*
with *ex-edge* **show** *?thesis*

by (*auto elim*: *JVM-CFG.cases*)
next
  case *Pop*
  with *ex-edge* **show** *?thesis*
    by (*auto elim*: *JVM-CFG.cases*)
next
  case *IAdd*
  with *ex-edge* **show** *?thesis*
    by (*auto elim*: *JVM-CFG.cases*)
next
  case [*simp*]: (*IfFalse b*)
  show *?thesis*
  proof (*cases x*)
    case [*simp*]: *None*
    have *Stk* (*length cs'*) (*stkLength P C M pc* − *1*) ∈ *Use P* (*sourcenode a*)
      (**is** *?stk-top* ∈ *?Use*)
      by (*fastforce intro*: *Use-IfFalse-Stk*)
    with *use-eq*
    have *state-val s ?stk-top* = *state-val s' ?stk-top*
      by (*simp del*: *state-val.simps*)
    with *ex-edge pred wt* **show** *?thesis*
      by (*auto elim*: *JVM-CFG.cases*)
  next
    case (*Some x'*)
    with *ex-edge* **show** *?thesis*
      by (*auto elim*: *JVM-CFG.cases*)
  qed
next
  case *CmpEq*
  with *ex-edge* **show** *?thesis*
    by (*auto elim*: *JVM-CFG.cases*)
next
  case (*Goto i*)
  with *ex-edge* **show** *?thesis*
    by (*auto elim*: *JVM-CFG.cases*)
next
  case [*simp*]: *Throw*
  have *ST* ≠ []
  proof −
    from *vn* **obtain** *T Ts mxs mxl is xt*
      where *sees-M*: (P_{wf}) ⊢ *C sees M*:*Ts*→*T* = (*mxs*,*mxl*,*is*,*xt*) *in C*
      by (*cases cs'*, (*cases x*, *auto*)+)
    with *vn*
    have *pc* < *length is*
      by (*cases cs'*, (*cases x*, *auto dest*: *sees-method-fun*)+)
    from *P-wf sees-M* **have** *wt-method* (P_{wf}) *C Ts T mxs mxl is xt* (P_{Φ} *C M*)
      by (*auto dest*: *sees-wf-mdecl simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
    with *Throw wt sees-M* ‹*pc* < *length is*› **show** *?thesis*
      by (*fastforce simp*: *wt-method-def*)

qed
then obtain *ST1 STr* **where** [*simp*]: *ST* = *ST1#STr* **by** (*cases ST*, *fastforce+*)
show *?thesis*
proof (*cases x*)
case [*simp*]: *None*
from *wt*
have *Stk* (*length cs'*) (*stkLength P C M pc* − *1*) ∈ *Use P* (*sourcenode a*)
(**is** *?stk-top* ∈ *?Use*)
by (*fastforce intro*: *Use-Throw-Stk*)
with *use-eq*
have *stk-top*: *state-val s ?stk-top* = *state-val s' ?stk-top*
by (*simp del*: *state-val.simps*)
have ∀ *addr*. *HeapVar addr* ∈ *Use P* (*sourcenode a*)
by (*fastforce intro*: *Use-Throw-Heap*)
with *use-eq*
have ∀ *addr*. *state-val s* (*HeapVar addr*) = *state-val s'* (*HeapVar addr*)
by (*simp del*: *state-val.simps*)
hence *h* = *h'*
by (*auto intro*: *ext*)
with *ex-edge pred stk-top wt* **show** *?thesis*
by (*auto elim!*: *JVM-CFG.cases*)
next
case (*Some x'*)
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
qed
qed
next
case *Entry*
with *ex-edge pred* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
qed
qed

lemma *edge-no-Def-equal*:
⟦ *valid-edge* (*P*, *C0*, *Main*) *a*;
V ∉ *Def P* (*sourcenode a*) ⟧
⟹ *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*
proof −
assume *ve*:*valid-edge* (*P*, *C0*, *Main*) *a*
and *v-not-def*: *V* ∉ *Def P* (*sourcenode a*)
obtain *h stk loc* **where** [*simp*]: (*s*::*state*) = (*h*, *stk*, *loc*) **by** (*cases s*, *blast*)
from *ve* **have** *vn*: *valid-node* (*P*, *C0*, *Main*) (*sourcenode a*)
and *ex-edge*: (*P*, *C0*, *Main*) ⊢ (*sourcenode a*)−*kind a*→(*targetnode a*)
by *simp-all*
show *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*
proof (*cases sourcenode a*)

```
  case [simp]: (Node cs x)
  with ve have cs ≠ []
    by (cases x, auto elim: JVM-CFG.cases)
   then obtain C M pc cs' where [simp]: cs = (C, M, pc)#cs' by (cases cs,
fastforce+)
  with vn obtain ST LT where wt: ((P_Φ) C M ! pc) = ⌊(ST,LT)⌋
    by (cases cs', (cases x, auto)+)
  show ?thesis
  proof (cases instrs-of (P_wf) C M ! pc)
    case [simp]: (Load nat)
    from ex-edge have x = None
      by (auto elim: JVM-CFG.cases)
    with v-not-def have V ≠ Stk (length cs') (stkLength P C M pc)
      by (auto intro!: Def-Load)
    with ex-edge show ?thesis
      by (auto elim!: JVM-CFG.cases, cases V, auto)
  next
    case [simp]: (Store nat)
    with ex-edge have x = None
      by (auto elim: JVM-CFG.cases)
    with v-not-def have V ≠ Loc (length cs') nat
      by (auto intro!: Def-Store)
    with ex-edge show ?thesis
      by (auto elim!: JVM-CFG.cases, cases V, auto)
  next
    case [simp]: (Push val)
    with ex-edge have x = None
      by (auto elim: JVM-CFG.cases)
    with v-not-def have V ≠ Stk (length cs') (stkLength P C M pc)
      by (auto intro!: Def-Push)
    with ex-edge show ?thesis
      by (auto elim!: JVM-CFG.cases, cases V, auto)
  next
    case [simp]: (New Cl)
    show ?thesis
    proof (cases x)
      case None
      with ex-edge show ?thesis
        by (auto elim: JVM-CFG.cases)
    next
      case (Some x')
      then obtain cs'' xf where [simp]: x = ⌊(cs'',xf)⌋
        by (cases x', fastforce)
      with ex-edge v-not-def show ?thesis
        apply (auto elim!: JVM-CFG.cases)
         apply (cases V, auto intro!: Def-New-Normal-Stk Def-New-Normal-Heap)
          by (cases V, auto intro!: Def-Exc-Stk)+
   qed
  next
```

case [*simp*]: (*Getfield F Cl*)
show *?thesis*
proof (*cases x*)
case *None*
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
next
case (*Some x'*)
then obtain *cs'' xf* **where** [*simp*]: $x = \lfloor(cs'',xf)\rfloor$
by (*cases x'*, *fastforce*)
with *ex-edge v-not-def* **show** *?thesis*
apply (*auto elim*!: *JVM-CFG.cases simp*: *split-beta*)
apply (*cases V*, *auto intro*!: *Def-Getfield-Stk*)
by (*cases V*, *auto intro*!: *Def-Exc-Stk*)+
qed
next
case [*simp*]: (*Putfield Fd Cl*)
show *?thesis*
proof (*cases x*)
case *None*
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
next
case (*Some x'*)
then obtain *cs'' xf* **where** [*simp*]: $x = \lfloor(cs'',xf)\rfloor$
by (*cases x'*, *fastforce*)
with *ex-edge v-not-def* **show** *?thesis*
apply (*auto elim*!: *JVM-CFG.cases simp*: *split-beta*)
apply (*cases V*, *auto intro*!: *Def-Putfield-Heap*)
by (*cases V*, *auto intro*!: *Def-Exc-Stk*)+
qed
next
case [*simp*]: (*Checkcast Cl*)
show *?thesis*
proof (*cases x*)
case *None*
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
next
case (*Some x'*)
then obtain *cs'' xf* **where** [*simp*]: $x = \lfloor(cs'',xf)\rfloor$
by (*cases x'*, *fastforce*)
with *ex-edge v-not-def* **show** *?thesis*
apply (*auto elim*!: *JVM-CFG.cases*)
by (*cases V*, *auto intro*!: *Def-Exc-Stk*)+
qed
next
case [*simp*]: (*Invoke M' n'*)
show *?thesis*

proof (*cases x*)
  case *None*
  with *ex-edge* **show** *?thesis*
    by (*auto elim*: *JVM-CFG.cases*)
next
  case (*Some x'*)
  then obtain *cs'' xf* **where** [*simp*]: $x = \lfloor(cs'',xf)\rfloor$
    by (*cases x'*, *fastforce*)
  from *ex-edge v-not-def* **show** *?thesis*
    apply (*auto elim*!: *JVM-CFG.cases*)
      apply (*cases V*, *auto intro*!: *Def-Invoke-Loc*)
     by (*cases V*, *auto intro*!: *Def-Exc-Stk*)+
qed
next
  case *Return*
  with *ex-edge v-not-def* **show** *?thesis*
    apply (*auto elim*!: *JVM-CFG.cases*)
    by (*cases V*, *auto intro*!: *Def-Return-Stk*)
next
  case *Pop*
  with *ex-edge* **show** *?thesis*
    by (*auto elim*: *JVM-CFG.cases*)
next
  case *IAdd*
  with *ex-edge v-not-def* **show** *?thesis*
    apply (*auto elim*!: *JVM-CFG.cases*)
    by (*cases V*, *auto intro*!: *Def-IAdd-Stk*)
next
  case (*IfFalse b*)
  with *ex-edge* **show** *?thesis*
    by (*auto elim*: *JVM-CFG.cases*)
next
  case *CmpEq*
  with *ex-edge v-not-def* **show** *?thesis*
    apply (*auto elim*!: *JVM-CFG.cases*)
    by (*cases V*, *auto intro*!: *Def-CmpEq-Stk*)
next
  case (*Goto i*)
  with *ex-edge* **show** *?thesis*
    by (*auto elim*: *JVM-CFG.cases*)
next
  case [*simp*]: *Throw*
  show *?thesis*
  proof (*cases x*)
    case *None*
    with *ex-edge* **show** *?thesis*
      by (*auto elim*: *JVM-CFG.cases*)
  next
    case (*Some x'*)

then obtain *cs''* *xf* **where** [*simp*]: $x = \lfloor(cs'',xf)\rfloor$
by (*cases x'*, *fastforce*)
from *ex-edge v-not-def* **show** *?thesis*
apply (*auto elim*!: *JVM-CFG.cases*)
by (*cases V*, *auto intro*!: *Def-Exc-Stk*)+
qed
qed
next
case *Entry*
with *ex-edge* **show** *?thesis*
by (*auto elim*: *JVM-CFG.cases*)
qed
qed

interpretation *JVM-CFG-wf*: *CFG-wf*
sourcenode targetnode kind valid-edge prog (*-Entry-*)
Def (*fst prog*) *Use* (*fst prog*) *state-val*
for *prog*
proof (*unfold-locales*)
show *Def* (*fst prog*) (*-Entry-*) = {} ∧ *Use* (*fst prog*) (*-Entry-*) = {}
by (*auto elim*: *Def.cases Use.cases*)
next
fix *a V s*
assume *ve*:*valid-edge prog a*
and *v-not-def*: $V \notin$ *Def* (*fst prog*) (*sourcenode a*)
thus *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*
by −(*cases prog*,
rule edge-no-Def-equal [*of fst prog fst* (*snd prog*) *snd* (*snd prog*)], *auto*)
next
fix *a s s'*
assume *ve*: *valid-edge prog a*
and *use-eq*: ∀ *V*∈*Use* (*fst prog*) (*sourcenode a*). *state-val s V* = *state-val s' V*
thus ∀ *V*∈*Def* (*fst prog*) (*sourcenode a*).
state-val (*transfer* (*kind a*) *s*) *V* = *state-val* (*transfer* (*kind a*) *s'*) *V*
by −(*cases prog*,
rule edge-transfer-uses-only-Use [*of fst prog fst*(*snd prog*) *snd*(*snd prog*)], *auto*)
next
fix *a s s'*
assume *ve*: *valid-edge prog a*
and *pred*: *pred* (*kind a*) *s*
and *use-eq*: ∀ *V*∈*Use* (*fst prog*) (*sourcenode a*). *state-val s V* = *state-val s' V*
thus *pred* (*kind a*) *s'*
by −(*cases prog*,
rule CFG-edge-Uses-pred-equal [*of fst prog fst*(*snd prog*) *snd*(*snd prog*)], *auto*)
next
fix *a a'*
assume *ve-a*: *valid-edge prog a*
and *ve-a'*: *valid-edge prog a'*
and *src-eq*: *sourcenode a* = *sourcenode a'*

and *trg-neq*: *targetnode a* ≠ *targetnode a'*
hence *prog* ⊢ (*sourcenode a*)−*kind a*→(*targetnode a*)
and *prog* ⊢ (*sourcenode a'*)−*kind a'*→(*targetnode a'*)
by *simp-all*
with *src-eq trg-neq*
show ∃ *Q Q'*. *kind a* = $(Q)_\surd$ ∧ *kind a'* = $(Q')_\surd$ ∧ (∀ *s*. (*Q s* ⟶ ¬ *Q' s*) ∧ (*Q' s* ⟶ ¬ *Q s*))
apply (*cases prog*, *auto*)
apply (*erule JVM-CFG.cases*, *erule-tac* [!] *JVM-CFG.cases*)

by *simp-all*
qed

interpretation *JVM-CFGExit-wf*: *CFGExit-wf*
sourcenode targetnode kind valid-edge prog (*-Entry-*)
Def (*fst prog*) *Use* (*fst prog*) *state-val* (*-Exit-*)
proof
show *Def* (*fst prog*) (*-Exit-*) = {} ∧ *Use* (*fst prog*) (*-Exit-*) = {}
by(*fastforce elim*:*Def.cases Use.cases*)
qed

end

6.5 Instantiating the control dependences

theory *JVMControlDependences* **imports**
JVMPostdomination
JVMCFG-wf
../Dynamic/DynPDG
../StaticIntra/CDepInstantiations
begin

6.5.1 Dynamic dependences

interpretation *JVMDynStandardControlDependence*:
DynStandardControlDependencePDG sourcenode targetnode kind
$valid\text{-}edge_{CFG}$ *prog* (*-Entry-*) *Def* (fst_{CFG} *prog*) *Use* (fst_{CFG} *prog*)
state-val (*-Exit-*) **..**

interpretation *JVMDynWeakControlDependence*:
DynWeakControlDependencePDG sourcenode targetnode kind
$valid\text{-}edge_{CFG}$ *prog* (*-Entry-*) *Def* (fst_{CFG} *prog*) *Use* (fst_{CFG} *prog*)
state-val (*-Exit-*) **..**

6.5.2 Static dependences

interpretation *JVMStandardControlDependence*:
StandardControlDependencePDG sourcenode targetnode kind

valid-edge$_{CFG}$ *prog* (*-Entry-*) *Def* (*fst*$_{CFG}$ *prog*) *Use* (*fst*$_{CFG}$ *prog*)
state-val (*-Exit-*) **..**

interpretation *JVMWeakControlDependence*:
WeakControlDependencePDG sourcenode targetnode kind
valid-edge$_{CFG}$ *prog* (*-Entry-*) *Def* (*fst*$_{CFG}$ *prog*) *Use* (*fst*$_{CFG}$ *prog*)
state-val (*-Exit-*) **..**

end

Chapter 7

Equivalence of the CFG and Jinja

theory *SemanticsWF* **imports** *JVMInterpretation ../Basic/SemanticsCFG* **begin**

declare *rev-nth* [*simp add*]

7.1 State updates

The following abbreviations update the stack and the local variables (in the representation as used in the CFG) according to a *frame list* as it is used in Jinja's state representation.

abbreviation *update-stk* :: $((nat \times nat) \Rightarrow val) \Rightarrow (frame\ list) \Rightarrow ((nat \times nat) \Rightarrow val)$
where
  update-stk stk frs ≡ (λ(*a*, *b*).
    if length frs ≤ *a then stk* (*a*, *b*)
      else let xs = *fst* (*frs* ! (*length frs* − *Suc a*))
        in if length xs ≤ *b then stk* (*a*, *b*) *else xs* ! (*length xs* − *Suc b*))

abbreviation *update-loc* :: $((nat \times nat) \Rightarrow val) \Rightarrow (frame\ list) \Rightarrow ((nat \times nat) \Rightarrow val)$
where
  update-loc loc frs ≡ (λ(*a*, *b*).
    if length frs ≤ *a then loc* (*a*, *b*)
      else let xs = *fst* (*snd* (*frs* ! (*length frs* − *Suc a*)))
        in if length xs ≤ *b then loc* (*a*, *b*) *else xs* ! *b*)

7.1.1 Some simplification lemmas

lemma *update-loc-s2jvm* [*simp*]:
  update-loc loc (*snd*(*snd*(*state-to-jvm-state P cs* (*h*,*stk*,*loc*)))) = *loc*
  by (*auto intro*!: *ext simp*: *nth-locss*)

lemma *update-stk-s2jvm* [*simp*]:
update-stk stk (*snd*(*snd*(*state-to-jvm-state P cs* (*h*,*stk*,*loc*)))) = *stk*
by (*auto intro*!: *ext simp*: *nth-stkss*)

lemma *update-loc-s2jvm′* [*simp*]:
update-loc loc (*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) = *loc*
by (*auto intro*!: *ext simp*: *nth-locss*)

lemma *update-stk-s2jvm′* [*simp*]:
update-stk stk (*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) = *stk*
by (*auto intro*!: *ext simp*: *nth-stkss*)

lemma *find-handler-find-handler-forD*:
find-handler (P_{wf}) *a h frs* = (*xp′*,*h′*,*frs′*)
⟹ *find-handler-for P* (*cname-of h a*) (*framestack-to-callstack frs*) =
framestack-to-callstack frs′
by (*induct frs*, *auto*)

lemma *find-handler-nonempty-frs* [*simp*]:
(*find-handler P a h frs* ≠ (*None*, *h′*, []))
by (*induct frs*, *auto*)

lemma *find-handler-heap-eqD*:
find-handler P a h frs = (*xp*, *h′*, *frs′*) ⟹ *h′* = *h*
by (*induct frs*, *auto*)

lemma *find-handler-frs-decrD*:
find-handler P a h frs = (*xp*, *h′*, *frs′*) ⟹ *length frs′* ≤ *length frs*
by (*induct frs*, *auto*)

lemma *find-handler-decrD* [*dest*]:
find-handler P a h frs = (*xp*, *h′*, *f*#*frs*) ⟹ *False*
by (*drule find-handler-frs-decrD*, *simp*)

lemma *find-handler-decrD′* [*dest*]:
⟦ *find-handler P a h frs* = (*xp*,*h′*,*f*#*frs′*); *length frs* = *length frs′* ⟧ ⟹ *False*
by (*drule find-handler-frs-decrD*, *simp*)

lemma *Suc-minus-Suc-Suc* [*simp*]:
b < *n* − *1* ⟹ *Suc* (*n* − *Suc* (*Suc b*)) = *n* − *Suc b*
by *simp*

lemma *find-handler-loc-fun-eq′*:
find-handler (P_{wf}) *a h*
(*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) =
(*xf*, *h′*, *frs*)
⟹ *update-loc loc frs* = *loc*
proof

fix *x*
obtain *a′ b′* **where** *x*: *x* = (*a′*::*nat*,*b′*::*nat*) **by** *fastforce*
assume *find-handler*: *find-handler* (P_{wf}) *a h*
(*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) =
(*xf*, *h′*, *frs*)
thus *update-loc loc frs x* = *loc x*
proof (*induct cs*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons aa cs′*)
then obtain *C M pc* **where** *step-case*: *find-handler* (P_{wf}) *a h*
(*zip* (*stkss P* ((*C*,*M*,*pc*) # *cs′*) *stk*)
(*zip* (*locss P* ((*C*,*M*,*pc*) # *cs′*) *loc*) ((*C*,*M*,*pc*) # *cs′*))) =
(*xf*, *h′*, *frs*)
by (*cases aa*, *clarsimp*)
note *IH* = ‹*find-handler* (P_{wf}) *a h*
(*zip* (*stkss P cs′ stk*) (*zip* (*locss P cs′ loc*) *cs′*)) =
(*xf*, *h′*, *frs*) ⟹
update-loc loc frs x = *loc x*›
show *?thesis*
proof (*cases match-ex-table* (P_{wf}) (*cname-of h a*) *pc* (*ex-table-of* (P_{wf}) *C M*))
case *None*
with *step-case IH* **show** *?thesis*
by *simp*
next
case (*Some e*)
with *step-case x*
show *?thesis*
by (*cases length cs′* = *a′*,
auto simp: *nth-Cons′ nth-locss*)
qed
qed
qed

lemma *find-handler-loc-fun-eq*:
find-handler (P_{wf}) *a h* (*snd*(*snd*(*state-to-jvm-state P cs* (*h*,*stk*,*loc*)))) = (*xf*,*h′*,*frs*)
⟹ *update-loc loc frs* = *loc*
by (*simp add*: *find-handler-loc-fun-eq′*)

lemma *find-handler-stk-fun-eq′*:
⟦*find-handler* (P_{wf}) *a h*
(*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) =
(*None*, *h′*, *frs*);
cd = *length frs* − *1*;
i = *length* (*fst*(*hd*(*frs*))) − *1* ⟧
⟹ *update-stk stk frs* = *stk*((*cd*, *i*) := *Addr a*)
proof
fix *x*

```
  obtain a' b' where x: x = (a'::nat,b'::nat) by fastforce
  assume find-handler: find-handler (P_wf) a h
    (zip (stkss P cs stk) (zip (locss P cs loc) cs)) =
    (None, h', frs)
    and calldepth: cd = length frs − 1
    and idx: i = length (fst (hd frs)) − 1
  from find-handler have frs ≠ []
    by clarsimp
  then obtain stk' loc' C' M' pc' frs' where frs: frs = (stk',loc',C',M',pc')#frs'
    by (cases frs, fastforce+)
  from find-handler
  show update-stk stk frs x = (stk((cd, i) := Addr a)) x
  proof (induct cs)
    case Nil
    thus ?case by simp
  next
    case (Cons aa cs')
    then obtain C M pc where step-case: find-handler (P_wf) a h
      (zip (stkss P ((C,M,pc) # cs') stk)
      (zip (locss P ((C,M,pc) # cs') loc) ((C,M,pc) # cs'))) =
      (None, h', frs)
      by (cases aa, clarsimp)
    note IH = ‹find-handler (P_wf) a h
      (zip (stkss P cs' stk) (zip (locss P cs' loc) cs')) =
      (None, h', frs) ⟹
      update-stk stk frs x = (stk((cd, i) := Addr a)) x›
    show ?thesis
    proof (cases match-ex-table (P_wf) (cname-of h a) pc (ex-table-of (P_wf) C M))
      case None
      with step-case IH show ?thesis
        by simp
    next
      case (Some e)
      show ?thesis
      proof (cases a' = length cs')
        case True
        with Some step-case frs calldepth idx x
        show ?thesis
          by (fastforce simp: nth-Cons')
      next
        case False
        with Some step-case frs calldepth idx x
        show ?thesis
          by (fastforce simp: nth-Cons' nth-stkss)
      qed
    qed
  qed
qed
```

lemma *find-handler-stk-fun-eq*:
find-handler (P_{wf}) *a h* (*snd*(*snd*(*state-to-jvm-state P cs* (*h*,*stk*,*loc*)))) = (*None*,*h′*,*frs*)
⟹ *update-stk stk frs* = *stk*((*length frs* − *1*, *length* (*fst*(*hd*(*frs*))) − *1*) := *Addr a*)
by (*simp add*: *find-handler-stk-fun-eq′*)

lemma *f2c-emptyD* [*dest*]:
framestack-to-callstack frs = [] ⟹ *frs* = []
by (*simp add*: *framestack-to-callstack-def*)

lemma *f2c-emptyD′* [*dest*]:
[] = *framestack-to-callstack frs* ⟹ *frs* = []
by (*simp add*: *framestack-to-callstack-def*)

lemma *correct-state-imp-valid-callstack*:
⟦ *P*,*cs* $\vdash_{BV}$ *s* √; *fst* (*last cs*) = *C0*; *fst*(*snd* (*last cs*)) = *Main* ⟧
⟹ *valid-callstack* (*P*,*C0*,*Main*) *cs*
proof (*cases cs rule*: *rev-cases*)
case *Nil*
thus *?thesis* **by** *simp*
next
case (*snoc cs′ y*)
assume *bv-correct*: *P*,*cs* $\vdash_{BV}$ *s* √
and *last-C*: *fst* (*last cs*) = *C0*
and *last-M*: *fst*(*snd* (*last cs*)) = *Main*
with *snoc* **obtain** *pcX* **where** [*simp*]: *cs* = *cs′*@[(*C0*,*Main*,*pcX*)]
by (*cases last cs*, *fastforce*)
obtain *h stk loc* **where** [*simp*]: *s* = (*h*,*stk*,*loc*)
by (*cases s*, *fastforce*)
from *bv-correct* **show** *?thesis*
proof (*cases snd*(*snd*(*state-to-jvm-state P cs s*)))
case *Nil*
thus *?thesis*
by (*cases cs′*, *auto*)
next
case [*simp*]: (*Cons a frs′*)
obtain *stk′ loc′ C M pc* **where** [*simp*]: *a* = (*stk′*, *loc′*, *C*, *M*, *pc*) **by** (*cases a*, *fastforce*)
from *Cons bv-correct* **show** *?thesis*
apply *clarsimp*
proof (*induct cs′ arbitrary*: *stk′ loc′ C M pc frs′*)
case *Nil*
thus *?case* **by** (*fastforce simp*: *bv-conform-def*)
next
case (*Cons a′ cs″*)
then have [*simp*]: *a′* = (*C*,*M*,*pc*)
by (*cases a′*, *fastforce*)
from *Cons* **obtain** *T Ts mxs mxl is xt*
where *sees-M*: (P_{wf}) ⊢ *C sees M*:*Ts*→*T* = (*mxs*,*mxl*,*is*,*xt*) *in C*

by (*clarsimp simp*: *bv-conform-def correct-state-def*)
with *Cons*
have *pc* < *length is*
by (*auto dest*: *sees-method-fun*
simp: *bv-conform-def*)
from *wf-jvmprog-is-wf* [*of P*] *sees-M*
have *wt-method* (P_{wf}) *C Ts T mxs mxl is xt* (P_Φ *C M*)
by (*auto dest*: *sees-wf-mdecl simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
with ‹*pc* < *length is*› *sees-M*
have *length Ts* = *locLength P C M 0* − *Suc mxl*
by (*auto dest*!: *list-all2-lengthD*
simp: *wt-method-def wt-start-def*)
with *Cons sees-M* **show** *?case*
by (*cases cs''*,
(*fastforce dest*: *sees-method-fun simp*: *bv-conform-def*)+)
qed
qed
qed

declare *correct-state-def* [*simp del*]

lemma *bool-sym*: *Bool* (*a* = *b*) = *Bool* (*b* = *a*)
by *auto*

lemma *find-handler-exec-correct*:
⟦(P_{wf}),(P_Φ) ⊢ *state-to-jvm-state P cs* (*h*,*stk*,*loc*) √;
(P_{wf}),(P_Φ) ⊢ *find-handler* (P_{wf}) *a h*
(*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) √;
find-handler-for P (*cname-of h a*) *cs* = (*C'*, *M'*, *pc'*) # *cs'*
⟧ ⟹
(P_{wf}),(P_Φ) ⊢ (*None*, *h*,
(*stks* (*stkLength P C' M' pc'*)
(λ*a'*. (*stk*((*length cs'*, *stkLength P C' M' pc'* − *Suc 0*) := *Addr a*)) (*length cs'*, *a'*)),
locs (*locLength P C' M' pc'*) (λ*a*. *loc* (*length cs'*, *a*)), *C'*, *M'*, *pc'*) #
zip (*stkss P cs' stk*) (*zip* (*locss P cs' loc*) *cs'*)) √
proof (*induct cs*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons aa cs*)
note *state-correct* = ‹P_{wf},P_Φ ⊢ *state-to-jvm-state P* (*aa* # *cs*) (*h*, *stk*, *loc*) √›
note *IH* = ‹⟦P_{wf},P_Φ ⊢ *state-to-jvm-state P cs* (*h*, *stk*, *loc*) √;
P_{wf},P_Φ ⊢ *find-handler* P_{wf} *a h* (*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) √;
find-handler-for P (*cname-of h a*) *cs* = (*C'*, *M'*, *pc'*) # *cs'*⟧
⟹ P_{wf},P_Φ ⊢ (*None*, *h*,
(*stks* (*stkLength P C' M' pc'*)
(λ*a'*. (*stk*((*length cs'*, *stkLength P C' M' pc'* − *Suc 0*) := *Addr*

a))
(length cs′, a′)),
locs (locLength P C′ M′ pc′) (λa. loc (length cs′, a)), C′, M′, pc′) #
zip (stkss P cs′ stk) (zip (locss P cs′ loc) cs′)) $\surd$›
note *trg-state-correct* = ‹P_{wf},P_Φ ⊢ *find-handler* P_{wf} *a h*
(zip (stkss P (aa # cs) stk)
(zip (locss P (aa # cs) loc) (aa # cs))) $\surd$›
note *fhf* = ‹*find-handler-for P (cname-of h a) (aa # cs) = (C′, M′, pc′) # cs′*›
obtain *C M pc* **where** [*simp*]: *aa = (C,M,pc)* **by** (*cases aa, fastforce*)
note *P-wf = wf-jvmprog-is-wf [of P]*
from *state-correct*
have *cs-state-correct*: P_{wf},P_Φ ⊢ *state-to-jvm-state P cs (h, stk, loc)* $\surd$
apply (*auto simp: correct-state-def*)
apply (*cases zip (stkss P cs stk) (zip (locss P cs loc) cs)*)
by *fastforce+*
show *?thesis*
proof (*cases match-ex-table* (P_{wf}) (*cname-of h a*) *pc* (*ex-table-of* (P_{wf}) *C M*))
case *None*
with *trg-state-correct fhf cs-state-correct IH* **show** *?thesis*
by *clarsimp*
next
case (*Some xte*)
with *IH trg-state-correct fhf state-correct* **show** *?thesis*
apply (*cases stkLength P C′ M′ (fst xte), auto*)
apply (*clarsimp simp: correct-state-def*)
apply (*auto simp: correct-state-def*)
apply (*rule-tac x=Ts* **in** *exI*)
apply (*rule-tac x=T* **in** *exI*)
apply (*rule-tac x=mxs* **in** *exI*)
apply (*rule-tac* $x=mxl_0$ **in** *exI*)
apply (*rule-tac x=is* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x=xt* **in** *exI*)
apply *clarsimp*
apply *clarsimp*
apply (*drule sees-method-fun, fastforce, clarsimp*)
apply (*auto simp: list-all2-Cons1*)
apply (*rule list-all2-all-nthI*)
apply *clarsimp*
apply *clarsimp*
apply (*frule-tac ys=zs* **in** *list-all2-lengthD*)
apply *clarsimp*
apply (*drule-tac p=n* **and** *ys=zs* **in** *list-all2-nthD*)
apply *clarsimp*
apply *clarsimp*
apply (*case-tac length aa − Suc (length aa − snd xte + n) = length zs − Suc n*)
apply *clarsimp*

apply *clarsimp*
apply (*rule list-all2-all-nthI*)
apply *clarsimp*
apply (*frule-tac p=n* **and** *ys=b* **in** *list-all2-nthD*)
apply (*clarsimp dest*!: *list-all2-lengthD*)
by (*clarsimp dest*!: *list-all2-lengthD*)
qed
qed

lemma *locs-rev-stks*:
$x \geq z \Longrightarrow$
locs z
(λb.
if $z < b$ *then loc* (*Suc y*, *b*)
else if $b \leq z$
then stk (y, $x + b - Suc\ z$)
else arbitrary)
@ [*stk* (y, $x - Suc\ 0$)]
=
stk (y, $x - Suc\ (z)$)
rev (*take z* (*stks x* (λa. *stk*(y, a))))
apply (*rule nth-equalityI*)
apply (*simp*)
apply (*auto simp*: *nth-append nth-Cons′ less-Suc-eq min.absorb2 max.absorb2*)
done

lemma *locs-invoke-purge*:
(z::*nat*) $> c \Longrightarrow$
locs l
(λb. *if* $z = c \longrightarrow Q\ b$ *then loc* (c, b) *else u b*) =
locs l (λa. *loc* (c, a))
by (*induct l*, *auto*)

lemma *nth-rev-equalityI*:
⟦*length xs* = *length ys*; ∀ *i*<*length xs*. *xs* ! (*length xs* − *Suc i*) = *ys* ! (*length ys* − *Suc i*)⟧
⟹ *xs* = *ys*
proof (*induct xs ys rule*: *list-induct2*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons x xs y ys*)
hence ∀ *i*<*length ys*. *xs* ! (*length ys* − *Suc i*) = *ys* ! (*length ys* − *Suc i*)
apply *auto*
apply (*erule-tac x=i* **in** *allE*)
by (*auto simp*: *nth-Cons′*)
with *Cons* **show** *?case*
by (*auto simp*: *nth-Cons*)

qed

lemma *length-locss*:
  $i < length\ cs$
  $\Longrightarrow length\ (locss\ P\ cs\ loc\ !\ (length\ cs - Suc\ i)) =$
  $locLength\ P\ (fst(cs\ !\ (length\ cs - Suc\ i)))$
  $(fst(snd(cs\ !\ (length\ cs - Suc\ i))))$
  $(snd(snd(cs\ !\ (length\ cs - Suc\ i))))$
apply (*induct cs*, *auto*)
apply (*case-tac* $i = length\ cs$)
by (*auto simp*: *nth-Cons'*)

lemma *locss-invoke-purge*:
  $z > length\ cs \Longrightarrow$
  $locss\ P\ cs$
  $(\lambda(a,\ b).\ if\ (a = z \longrightarrow Q\ b)$
  $then\ loc\ (a,\ b)$
  $else\ u\ b)$
  $= locss\ P\ cs\ loc$
  by (*induct cs*, *auto simp*: *locs-invoke-purge* [*simplified*])

lemma *stks-purge'*:
  $d \geq b \Longrightarrow stks\ b\ (\lambda x.\ if\ x = d\ then\ e\ else\ stk\ x) = stks\ b\ stk$
  by *simp*

7.1.2 Byte code verifier conformance

Here we prove state conformance invariant under *transfer* for our CFG. Therefore, we must assume, that the predicate of a potential preceding predicate-edge holds for every update-edge.

theorem *bv-invariant*:
  $[\![$ *valid-edge* $(P,C0,Main)\ a$;
  sourcenode $a = (\text{-}\ (C,M,pc)\#cs,x\ \text{-})$;
  targetnode $a = (\text{-}\ (C',M',pc')\#cs',x'\ \text{-})$;
  pred $(kind\ a)\ s$;
  $x \neq None \longrightarrow (\exists\ a\text{-}pred.$
  sourcenode $a\text{-}pred = (\text{-}\ (C,M,pc)\#cs,None\ \text{-}) \wedge$
  targetnode $a\text{-}pred = sourcenode\ a \wedge$
  valid-edge $(P,C0,Main)\ a\text{-}pred \wedge$
  pred $(kind\ a\text{-}pred)\ s$
  );
  $P,((C,M,pc)\#cs) \vdash_{BV} s\ \surd\ ]\!]$
  $\Longrightarrow P,((C',M',pc')\#cs') \vdash_{BV} transfer\ (kind\ a)\ s\ \surd$
proof −
  assume *ve*: *valid-edge* $(P,\ C0,\ Main)\ a$
  and *src* [*simp*]: *sourcenode* $a = (\text{-}\ (C,M,pc)\#cs,x\ \text{-})$
  and *trg* [*simp*]: *targetnode* $a = (\text{-}\ (C',M',pc')\#cs',x'\ \text{-})$
  and *pred-s*: *pred* $(kind\ a)\ s$
  and *a-pred*: $x \neq None \longrightarrow (\exists\ a\text{-}pred.$

sourcenode a-pred = (- (*C*,*M*,*pc*)#*cs*,*None* -) ∧
targetnode a-pred = *sourcenode a* ∧
valid-edge (*P*,*C0*,*Main*) *a-pred* ∧
pred (*kind a-pred*) *s*
)
and *state-correct*: *P*,((*C*,*M*,*pc*)#*cs*) $\vdash_{BV}$ *s* $\surd$
obtain *h stk loc* **where** *s* [*simp*]: *s* = (*h*,*stk*,*loc*) **by** (*cases s*, *fastforce*)
note *P-wf* = *wf-jvmprog-is-wf* [*of P*]
from *ve* **obtain** *Ts T mxs mxl is xt*
where *sees-M*: (P_{wf}) ⊢ *C sees M*:*Ts*→*T* = (*mxs*,*mxl*,*is*,*xt*) *in C*
and *pc* < *length is*
and *reachable*: P_Φ *C M* ! *pc* ≠ *None*
by (*cases x*) (*cases cs*, *auto*)+
from *P-wf sees-M*
have *wt-method*: *wt-method* (P_{wf}) *C Ts T mxs mxl is xt* (P_Φ *C M*)
by (*auto dest*: *sees-wf-mdecl simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
with *sees-M* ‹*pc* < *length is*› *reachable*
have *applicable*: app_i ((*is* ! *pc*),(P_{wf}),*pc*,*mxs*,*T*,(*the*(P_Φ *C M* ! *pc*)))
by (*auto simp*: *wt-method-def*)
from *state-correct ve P-wf*
have *trg-state-correct*:
(P_{wf}),(P_Φ) ⊢ *the* (*JVMExec.exec* ((P_{wf}), *state-to-jvm-state P* ((*C*,*M*,*pc*)#*cs*) *s*)) $\surd$
apply *simp*
apply (*drule BV-correct-1*)
apply (*fastforce simp*: *bv-conform-def*)
apply (*simp add*: *exec-1-iff*)
apply (*cases instrs-of* (P_{wf}) *C M* ! *pc*)
apply (*simp-all add*: *split-beta*)
done
from *reachable* **obtain** *ST LT* **where** *reachable*: (P_Φ) *C M* ! *pc* = ⌊(*ST*, *LT*)⌋
by *fastforce*
with *wt-method sees-M* ‹*pc* < *length is*›
have *stk-loc-succs*:
∀ *pc*′ ∈ *set* (*succs* (*is* ! *pc*) (*ST*, *LT*) *pc*).
stkLength P C M pc′ = *length* (*fst* (eff_i (*is* ! *pc*, (P_{wf}), *ST*, *LT*))) ∧
locLength P C M pc′ = *length* (*snd* (eff_i (*is* ! *pc*, (P_{wf}), *ST*, *LT*)))
unfolding *wt-method-def* **apply** (*cases is* ! *pc*)
using [[*simproc del*: *list-to-set-comprehension*]]
apply (*cases is* ! *pc*)
apply (*tactic* ‹*PARALLEL-ALLGOALS*
(*Clasimp.fast-force-tac* (@{*context*} *addSDs* @{*thms list-all2-lengthD*}))›)
done
have [*simp*]: ∃ *x*. *x* **by** *auto*
have [*simp*]: *Ex Not* **by** *auto*
show *?thesis*
proof (*cases instrs-of* (P_{wf}) *C M* ! *pc*)
case (*Invoke m n*)
from *state-correct* **have** *preallocated h*

by (*clarsimp simp*: *bv-conform-def correct-state-def hconf-def*)
from *Invoke applicable sees-M* **have** *stkLength P C M pc* > *n*
by (*cases the* (P_{Φ} *C M* ! *pc*)) *auto*
show *?thesis*
proof (*cases x*)
case [*simp*]: *None*
with *ve Invoke* **obtain** *Q* **where** *kind*: *kind a* = $(Q)_{\surd}$
by (*auto elim*!: *JVM-CFG.cases*)
with *ve Invoke* **have** (*C'*,*M'*,*pc'*)#*cs'* = (*C*,*M*,*pc*)#*cs*
by (*auto elim*!: *JVM-CFG.cases*)
with *state-correct kind* **show** *?thesis*
by *simp*
next
case [*simp*]: (*Some aa*)
with *ve Invoke* **obtain** *xf* **where** [*simp*]: *aa* = ((*C'*,*M'*,*pc'*)#*cs'* , *xf*)
by (*auto elim*!: *JVM-CFG.cases*)
from *ve Invoke* **obtain** *f* **where** *kind*: *kind a* = ⇑*f*
apply −
apply *clarsimp*
apply (*erule JVM-CFG.cases*)
apply *auto*
done
show *?thesis*
proof (*cases xf*)
case [*simp*]: *True*
with *a-pred Invoke* **have** *stk-n*: *stk* (*length cs*, *stkLength P C M pc* − *Suc n*) = *Null*
apply *auto*
apply (*erule JVM-CFG.cases*)
apply *simp-all*
done
from *ve Invoke kind*
have [*simp*]: *f* = (λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs'*,(*stkLength P C' M' pc'*) − *1*) := *Addr* (*addr-of-sys-xcpt NullPointer*)),
loc))
apply −
apply *clarsimp*
apply (*erule JVM-CFG.cases*)
apply *auto*
done
from *ve Invoke*
have *find-handler-for P NullPointer* ((*C*,*M*,*pc*)#*cs*) = (*C'*,*M'*,*pc'*)#*cs'*
apply −
apply *clarsimp*
apply (*erule JVM-CFG.cases*)
apply *auto*
done

with *Invoke state-correct kind stk-n trg-state-correct applicable sees-M* ‹*preallocated h*›
show *?thesis*
apply (*cases the* (P_Φ *C M* ! *pc*),
auto simp: *bv-conform-def stkss-purge*
simp del: *find-handler-for.simps exec.simps* app_i*.simps fun-upd-apply*)
apply (*rule-tac cs*=(*C*,*M*,*pc*)#*cs* **in** *find-handler-exec-correct*)
apply *fastforce*
apply (*fastforce simp*: *split-beta split*: *if-split-asm*)
apply *fastforce*
done
next
case [*simp*]: *False*
from *a-pred Invoke*
have [*simp*]: $m = M'$
by −(*clarsimp*, *erule JVM-CFG.cases*, *auto*)
from *a-pred Invoke*
have [*simp*]: $pc' = 0$
by −(*clarsimp*, *erule JVM-CFG.cases*, *auto*)
from *ve Invoke*
have [*simp*]: *cs'* = (*C*,*M*,*pc*)#*cs*
by −(*clarsimp*, *erule JVM-CFG.cases*, *auto*)
from *ve Invoke kind*
have [*simp*]:
f = (λ*s*. *exec-instr* (*Invoke m n*) *P s* (*length cs*) (*stkLength P C M pc*)
arbitrary (*locLength P C' M' 0*))
by −(*clarsimp*, *erule JVM-CFG.cases*, *auto*)
from *state-correct* **obtain** *ST LT* **where** [*simp*]:
(P_Φ) *C M* ! *pc* = ⌊(*ST*,*LT*)⌋
by (*auto simp*: *bv-conform-def correct-state-def*)
from *a-pred Invoke*
have [*simp*]:
fst (*method* (P_{wf})
(*cname-of h* (*the-Addr* (*stk* (*length cs*, *length ST* − *Suc n*)))) *M'*) = *C'*
by −(*clarsimp*, *erule JVM-CFG.cases*, *auto*)
from *a-pred Invoke*
have [*simp*]: *stk* (*length cs*, *length ST* − *Suc n*) ≠ *Null*
by −(*clarsimp*, *erule JVM-CFG.cases*, *auto*)
from *state-correct applicable sees-M Invoke*
have [*simp*]: *ST* ! *n* ≠ *NT*
apply (*auto simp*: *correct-state-def bv-conform-def*)
apply (*drule-tac p*=*n* **and** *ys*=*ST* **in** *list-all2-nthD*)
apply *simp*
by *clarsimp*
from *applicable Invoke sees-M*
have *length ST* > *n*
by *auto*
with *trg-state-correct Invoke*
have [*simp*]: *stkLength P C' M' 0* = *0*

by (*auto simp*: *split-beta correct-state-def*
split: *if-split-asm*)
from *trg-state-correct Invoke* ‹*length ST* > *n*›
have *locLength P C′ M′ 0* =
Suc n + *fst*(*snd*(*snd*(*snd*(*snd*(*method* (P_{wf})
(*cname-of h* (*the-Addr* (*stk* (*length cs*, *length ST* − *Suc n*)))) *M′*)))))
by (*auto simp*: *split-beta correct-state-def*
dest!: *list-all2-lengthD*
split: *if-split-asm*)
with *Invoke state-correct trg-state-correct* ‹*length ST* > *n*›
have *JVMExec.exec* (P_{wf}, *state-to-jvm-state P* ((*C*, *M*, *pc*) # *cs*) *s*)
=
⌊(*None*, *h*,
(*stks* (*stkLength P C′ M′ pc′*) (λ*a*. *stk* (*Suc* (*length cs*), *a*)),
locs (*locLength P C′ M′ pc′*)
(λ*a′*. (λ(*a*, *b*).
if a = *Suc* (*length cs*) ⟶ *locLength P C′ M′ 0* ≤ *b then loc*
(*a*, *b*)
else if b ≤ *n then stk* (*length cs*, *length ST* − (*Suc n* − *b*))
else arbitrary) (*Suc* (*length cs*), *a′*)),
C′, *M′*, *pc′*) #
(*stks* (*length ST*) (λ*a*. *stk* (*length cs*, *a*)),
locs (*length LT*) (λ*a*. *loc* (*length cs*, *a*)), *C*, *M*, *pc*) #
zip (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*))⌋
apply (*auto simp*: *split-beta bv-conform-def*)
apply (*rule nth-equalityI*)
apply *simp*
apply (*cases ST*,
auto simp: *nth-Cons′ nth-append min.absorb1 min.absorb2*)
apply (*rule nth-equalityI*)
apply *simp*
by (*auto simp*: *rev-nth nth-Cons′ nth-append min-def*)
with *Invoke state-correct kind trg-state-correct applicable sees-M*
show *?thesis*
apply (*cases the* (P_{Φ} *C M* ! *pc*),
auto simp: *bv-conform-def stkss-purge rev-nth*
simp del: *find-handler-for.simps exec.simps* app_i*.simps*)
apply(*subst locss-invoke-purge*, *simp*)
by *simp*
qed
qed
next
case (*Load nat*)
with *stk-loc-succs sees-M reachable*
have *stkLength P C M* (*Suc pc*) = *Suc* (*stkLength P C M pc*)
and *locLength P C M* (*Suc pc*) = *locLength P C M pc*
by *simp-all*
with *state-correct ve P-wf applicable sees-M Load trg-state-correct*
show *?thesis*

apply *auto*
apply (*erule JVM-CFG.cases*, *simp-all*)
by (*auto simp*: *bv-conform-def stkss-purge stks-purge'*)
next
case (*Store nat*)
with *stk-loc-succs sees-M reachable applicable*
have *stkLength P C M* (*Suc pc*) = *stkLength P C M pc* − *1*
and *locLength P C M* (*Suc pc*) = *locLength P C M pc*
by *auto*
with *state-correct ve P-wf applicable sees-M Store trg-state-correct*
show *?thesis*
apply *auto*
apply (*erule JVM-CFG.cases*, *simp-all*)
by (*auto simp*: *bv-conform-def locss-purge*)
next
case (*Push val*)
with *stk-loc-succs sees-M reachable applicable*
have *stkLength P C M* (*Suc pc*) = *Suc* (*stkLength P C M pc*)
and *locLength P C M* (*Suc pc*) = *locLength P C M pc*
by *auto*
with *state-correct ve P-wf applicable sees-M Push trg-state-correct*
show *?thesis*
apply *auto*
apply (*erule JVM-CFG.cases*, *simp-all*)
by (*auto simp*: *bv-conform-def stks-purge' stkss-purge*)
next
case *Pop*
with *stk-loc-succs sees-M reachable applicable*
have *stkLength P C M* (*Suc pc*) = *stkLength P C M pc* − *1*
and *locLength P C M* (*Suc pc*) = *locLength P C M pc*
by *auto*
with *state-correct ve P-wf applicable sees-M Pop trg-state-correct*
show *?thesis*
apply *auto*
apply (*erule JVM-CFG.cases*, *simp-all*)
by (*auto simp*: *bv-conform-def*)
next
case *IAdd*
with *stk-loc-succs sees-M reachable applicable*
have *stkLength P C M* (*Suc pc*) = *stkLength P C M pc* − *1*
and *locLength P C M* (*Suc pc*) = *locLength P C M pc*
by *auto*
with *state-correct ve P-wf applicable sees-M IAdd trg-state-correct*
show *?thesis*
apply *auto*
apply (*erule JVM-CFG.cases*, *simp-all*)
by (*auto simp*: *bv-conform-def stks-purge' stkss-purge add.commute*)
next
case *CmpEq*

with *stk-loc-succs sees-M reachable applicable*
have *stkLength P C M* (*Suc pc*) = *stkLength P C M pc* − *1*
and *locLength P C M* (*Suc pc*) = *locLength P C M pc*
by *auto*
with *state-correct ve P-wf applicable sees-M CmpEq trg-state-correct*
show *?thesis*
apply *auto*
apply (*erule JVM-CFG.cases*, *simp-all*)
apply (*auto simp*: *bv-conform-def stks-purge′ stkss-purge bool-sym*)
apply (*erule JVM-CFG.cases*, *simp-all*)
by (*auto simp*: *bv-conform-def stks-purge′ stkss-purge bool-sym*)
next
case (*Goto b*)
with *stk-loc-succs sees-M reachable applicable*
have *stkLength P C M* (*nat* (*int pc* + *b*)) = *stkLength P C M pc*
and *locLength P C M* (*nat* (*int pc* + *b*)) = *locLength P C M pc*
by *auto*
with *state-correct ve P-wf applicable sees-M Goto trg-state-correct*
show *?thesis*
apply *auto*
by (*erule JVM-CFG.cases*, *simp-all add*: *bv-conform-def*)
next
case (*IfFalse b*)
have *nat-int-pc-conv*: *nat* (*int pc* + *1*) = *pc* + *1*
by (*cases pc*) *auto*
from *IfFalse stk-loc-succs sees-M reachable applicable*
have *stkLength P C M* (*Suc pc*) = *stkLength P C M pc* − *1*
and *stkLength P C M* (*nat* (*int pc* + *b*)) = *stkLength P C M pc* − *1*
and *locLength P C M* (*Suc pc*) = *locLength P C M pc*
and *locLength P C M* (*nat* (*int pc* + *b*)) = *locLength P C M pc*
by *auto*
with *state-correct ve P-wf applicable sees-M IfFalse pred-s nat-int-pc-conv trg-state-correct*
show *?thesis*
apply *auto*
apply (*erule JVM-CFG.cases*, *simp-all*)
by (*auto simp*: *bv-conform-def split*: *if-split-asm*)
next
case *Return*
with *ve* **obtain** *Ts′ T′ mxs′ mxl′ is′ xt′*
where *sees-M′*: (P_{wf}) ⊢ *C′ sees M′*:*Ts′*→*T′* = (*mxs′*,*mxl′*,*is′*,*xt′*) *in C′*
and (*pc′* − *1*) < *length is′*
and *reachable′*: P_{Φ} *C′ M′* ! (*pc′* − *1*) ≠ *None*
apply *auto*
apply (*erule JVM-CFG.cases*, *auto*)
by (*cases cs′*, *auto*)
with *Return ve wt-method sees-M applicable*
have *is′* ! (*pc′* − *1*) = *Invoke M* (*length Ts*)
apply *auto*

apply (*erule JVM-CFG.cases*, *auto*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
by (*auto dest*!: *list-all2-lengthD simp*: *wt-method-def wt-start-def*)
from *P-wf sees-M′*
have *wt-method* (P_{wf}) *C′ Ts′ T′ mxs′ mxl′ is′ xt′* (P_{Φ} *C′ M′*)
by (*auto dest*: *sees-wf-mdecl simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
with *ve Return* ‹*pc′* − *1* < *length is′*› *reachable′ sees-M state-correct*
have *stkLength P C′ M′ pc′* = *stkLength P C′ M′* (*pc′* − *1*) − *length Ts*
using [[*simproc del*: *list-to-set-comprehension*]]
apply *auto*
apply (*erule JVM-CFG.cases*, *auto*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
using *sees-M′*
apply *hypsubst-thin*
apply (*auto simp*: *wt-method-def*)
apply (*erule-tac x*=*pc′* **in** *allE*)
apply (*auto simp*: *bv-conform-def correct-state-def not-less-eq less-Suc-eq*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
apply (*auto simp*: *wt-start-def*)
apply (*auto dest*!: *list-all2-lengthD*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
by *auto*
from ‹*wt-method* (P_{wf}) *C′ Ts′ T′ mxs′ mxl′ is′ xt′* (P_{Φ} *C′ M′*)›
‹(*pc′* − *1*) < *length is′*› ‹P_{Φ} *C′ M′* ! (*pc′* − *1*) ≠ *None*›
‹*is′* ! (*pc′* − *1*) = *Invoke M* (*length Ts*)›
have *stkLength P C′ M′* (*pc′* − *1*) > *0*
by (*fastforce simp*: *wt-method-def*)
then obtain *ST′ STr′* **where** [*simp*]: *fst* (*the* (P_{Φ} *C′ M′* ! (*pc′* − *1*))) = *ST′*#*STr′*
by (*cases fst* (*the* (P_{Φ} *C′ M′* ! (*pc′* − *1*))), *fastforce*+)
from *wt-method*
have *locLength P C M 0* = *Suc* (*length Ts*) + *mxl*
by (*auto dest*!: *list-all2-lengthD*
simp: *wt-method-def wt-start-def*)
from ‹*wt-method* (P_{wf}) *C′ Ts′ T′ mxs′ mxl′ is′ xt′* (P_{Φ} *C′ M′*)›
ve Return ‹*pc′* − *1* < *length is′*› *reachable′ sees-M state-correct*
have *locLength P C′ M′* (*pc′* − *1*) = *locLength P C′ M′ pc′*
using [[*simproc del*: *list-to-set-comprehension*]]
apply *auto*
apply (*erule JVM-CFG.cases*, *auto*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
using *sees-M′*
apply *hypsubst-thin*
apply (*auto simp*: *wt-method-def*)
apply (*erule-tac x*=*pc′* **in** *allE*)
apply (*auto simp*: *wt-start-def*)
apply (*clarsimp simp*: *bv-conform-def correct-state-def*)

apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
by (*auto dest*!: *list-all2-lengthD*)
with ‹*stkLength P C′ M′ pc′* = *stkLength P C′ M′* (*pc′* − *1*) − *length Ts*›
Return state-correct ve P-wf applicable sees-M trg-state-correct sees-M′
‹*fst* (*the* (P_Φ *C′ M′* ! (*pc′* − *1*))) = *ST′*#*STr′*› ‹*is′* ! (*pc′* − *1*) = *Invoke M* (*length Ts*)›
‹*locLength P C M 0* = *Suc* (*length Ts*) + *mxl*›
show *?thesis*
apply (*auto simp*: *bv-conform-def*)
apply (*erule JVM-CFG.cases*, *auto simp*: *stkss-purge locss-purge*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
apply (*auto simp*: *correct-state-def*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
apply (*drule sees-method-fun*, *fastforce*, *clarsimp*)
apply (*rule-tac x*=*Ts′* **in** *exI*)
apply (*rule-tac x*=*T′* **in** *exI*)
apply (*rule-tac x*=*mxs′* **in** *exI*)
apply (*rule-tac x*=*mxl′* **in** *exI*)
apply (*rule-tac x*=*is′* **in** *exI*)
apply *clarsimp*
apply (*rule conjI*)
apply (*rule-tac x*=*xt′* **in** *exI*)
apply *clarsimp*
apply (*rule list-all2-all-nthI*)
apply *clarsimp*
apply *clarsimp*
apply (*auto simp*: *rev-nth list-all2-Cons1*)
apply (*case-tac n*, *auto simp*: *list-all2-Cons1*)
apply (*case-tac n*, *auto simp*: *list-all2-Cons1*)
apply (*drule-tac p*=*nat* **and** *ys*=*zs* **in** *list-all2-nthD2*)
apply *clarsimp*
by *auto*
next
case (*New Cl*)
from *state-correct* **have** *preallocated h*
by (*clarsimp simp*: *bv-conform-def correct-state-def hconf-def*)
from *New stk-loc-succs sees-M reachable applicable*
have *stkLength P C M* (*Suc pc*) = *Suc* (*stkLength P C M pc*)
and *locLength P C M* (*Suc pc*) = *locLength P C M pc*
by *auto*
with *New state-correct ve sees-M trg-state-correct applicable a-pred* ‹*preallocated h*›
show *?thesis*
apply (*clarsimp simp del*: *exec.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)

apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
defer
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*simp add*: *bv-conform-def stkss-purge del*: *exec.simps find-handler-for.simps*)
apply (*rule-tac cs*=(*C*,*M*,*pc*)#*cs* **in** *find-handler-exec-correct* [*simplified*])
apply *fastforce*
apply *fastforce*
apply *clarsimp*
by (*auto simp*: *split-beta bv-conform-def stks-purge′ stkss-purge*
simp del: *find-handler-for.simps*)
next
case (*Getfield Fd Cl*)
from *state-correct* **have** *preallocated h*
by (*clarsimp simp*: *bv-conform-def correct-state-def hconf-def*)
from *Getfield stk-loc-succs sees-M reachable applicable*
have *stkLength P C M* (*Suc pc*) = *stkLength P C M pc*
and *locLength P C M* (*Suc pc*) = *locLength P C M pc*
by *auto*
with *Getfield state-correct ve sees-M trg-state-correct applicable a-pred* ‹*preallocated h*›
show *?thesis*
apply (*clarsimp simp del*: *exec.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
defer
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*simp add*: *bv-conform-def stkss-purge del*: *exec.simps find-handler-for.simps*)
apply (*rule-tac cs*=(*C*,*M*,*pc*)#*cs* **in** *find-handler-exec-correct* [*simplified*])
apply *fastforce*
apply (*fastforce simp*: *split-beta*)
apply *clarsimp*
by (*auto simp*: *split-beta bv-conform-def stks-purge′ stkss-purge*
simp del: *find-handler-for.simps*)
next
case (*Putfield Fd Cl*)
from *state-correct* **have** *preallocated h*
by (*clarsimp simp*: *bv-conform-def correct-state-def hconf-def*)
from *Putfield stk-loc-succs sees-M reachable applicable*
have *stkLength P C M* (*Suc pc*) = *stkLength P C M pc* − *2*
and *locLength P C M* (*Suc pc*) = *locLength P C M pc*
by *auto*
with *Putfield state-correct ve sees-M trg-state-correct applicable a-pred* ‹*preallocated h*›

show *?thesis*
apply (*clarsimp simp del*: *exec.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
defer
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*simp add*: *bv-conform-def stkss-purge del*: *exec.simps find-handler-for.simps*)
apply (*rule-tac cs*=(*C*,*M*,*pc*)#*cs* **in** *find-handler-exec-correct* [*simplified*])
apply *fastforce*
apply (*fastforce simp*: *split-beta*)
apply *clarsimp*
by (*auto simp*: *split-beta bv-conform-def stks-purge′ stkss-purge*
simp del: *find-handler-for.simps*)
next
case (*Checkcast Cl*)
from *state-correct* **have** *preallocated h*
by (*clarsimp simp*: *bv-conform-def correct-state-def hconf-def*)
from *Checkcast stk-loc-succs sees-M reachable applicable*
have *stkLength P C M* (*Suc pc*) = *stkLength P C M pc*
and *locLength P C M* (*Suc pc*) = *locLength P C M pc*
by *auto*
with *Checkcast state-correct ve sees-M*
trg-state-correct applicable a-pred pred-s ‹*preallocated h*›
show *?thesis*
apply (*clarsimp simp del*: *exec.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
defer
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*simp add*: *bv-conform-def stkss-purge del*: *exec.simps find-handler-for.simps*)
apply (*rule-tac cs*=(*C*,*M*,*pc*)#*cs* **in** *find-handler-exec-correct* [*simplified*])
apply *fastforce*
apply (*fastforce simp*: *split-beta*)
apply *clarsimp*
by (*auto simp*: *split-beta bv-conform-def*
simp del: *find-handler-for.simps*)
next
case *Throw*
from *state-correct* **have** *preallocated h*
by (*clarsimp simp*: *bv-conform-def correct-state-def hconf-def*)
from *Throw applicable state-correct sees-M* **obtain** *a*
where *stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null* ∨
stk(*length cs*, *stkLength P C M pc* − *1*) = *Addr a*

by (*cases stk*(*length cs*, *stkLength P C M pc* − *1*),
auto simp: *is-refT-def bv-conform-def correct-state-def conf-def*)
with *Throw state-correct ve trg-state-correct a-pred applicable sees-M* ‹*preallocated h*›
show *?thesis*
apply (*clarsimp simp del*: *exec.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp del*: *exec.simps find-handler-for.simps*)
apply (*erule JVM-CFG.cases*, *simp-all del*: *exec.simps find-handler-for.simps*)
apply (*clarsimp simp*: *bv-conform-def simp del*: *exec.simps find-handler-for.simps*)
apply (*rule conjI*)
apply (*clarsimp simp*: *stkss-purge simp del*: *exec.simps find-handler-for.simps*)
apply (*rule-tac cs*=(*C*,*M*,*pc*)#*cs* **in** *find-handler-exec-correct* [*simplified*])
apply *fastforce*
apply (*simp add*: *hd-stks*)
apply *simp*
apply (*clarsimp simp*: *stkss-purge simp del*: *exec.simps find-handler-for.simps*)
apply (*simp del*: *find-handler-for.simps exec.simps cong*: *if-cong*)
apply (*rule-tac cs*=(*C*,*M*,*pc*)#*cs* **in** *find-handler-exec-correct* [*simplified*])
apply *fastforce*
apply (*simp add*: *hd-stks*)
by *simp*
qed
qed

7.2 CFG simulates Jinja's semantics

7.2.1 Definitions

The following predicate defines the semantics of Jinja lifted to our state representation. Thereby, we require the state to be byte code verifier conform; otherwise the step in the semantics is undefined.

The predicate *valid-callstack* is actually an implication of the byte code verifier conformance. But we list it explicitly for convenience.

inductive *sem* :: *jvmprog* ⇒ *callstack* ⇒ *state* ⇒ *callstack* ⇒ *state* ⇒ *bool*
(‹- ⊢ ⟨-,-⟩ ⇒ ⟨-,-⟩›)
where *Step*:
⟦ *prog* = (*P*,*C0*,*Main*);
$P,cs \vdash_{BV} s \ \surd$;
valid-callstack prog cs;
JVMExec.exec ((P_{wf}), *state-to-jvm-state P cs s*) = ⌊(*None*,*h′*,*frs′*)⌋;
cs′ = *framestack-to-callstack frs′*;
s = (*h*, *stk*, *loc*);
s′ = (*h′*, *update-stk stk frs′*, *update-loc loc frs′*) ⟧
⟹ *prog* ⊢ ⟨*cs*,*s*⟩ ⇒ ⟨*cs′*,*s′*⟩

abbreviation *identifies* :: *j-node* ⇒ *callstack* ⇒ *bool*
where *identifies n cs* ≡ (*n* = (- *cs*,*None* -))

7.2.2 Some more simplification lemmas

lemma *valid-callstack-tl*:
  $valid\text{-}callstack\ prog\ ((C,M,pc)\#cs) \Longrightarrow valid\text{-}callstack\ prog\ cs$
  by (*cases prog*, *cases cs*, *auto*)

lemma *stkss-cong* [*cong*]:
  $[\![\ P = P';$
  $cs = cs';$
  $\bigwedge a\ b.\ [\![\ a < length\ cs;$
    $b < stkLength\ P\ (fst(cs\ !\ (length\ cs - Suc\ a)))$
      $(fst(snd(cs\ !\ (length\ cs - Suc\ a))))$
      $(snd(snd(cs\ !\ (length\ cs - Suc\ a))))\ ]\!]$
    $\Longrightarrow stk\ (a,\ b) = stk'\ (a,\ b)\ ]\!]$
  $\Longrightarrow stkss\ P\ cs\ stk = stkss\ P'\ cs'\ stk'$
  by (*auto*, *hypsubst-thin*, *induct cs'*,
    auto intro!: *nth-equalityI simp*: *nth-Cons'*)

lemma *locss-cong* [*cong*]:
  $[\![\ P = P';$
  $cs = cs';$
  $\bigwedge a\ b.\ [\![\ a < length\ cs;$
    $b < locLength\ P\ (fst(cs\ !\ (length\ cs - Suc\ a)))$
      $(fst(snd(cs\ !\ (length\ cs - Suc\ a))))$
      $(snd(snd(cs\ !\ (length\ cs - Suc\ a))))\ ]\!]$
    $\Longrightarrow loc\ (a,\ b) = loc'\ (a,\ b)\ ]\!]$
  $\Longrightarrow locss\ P\ cs\ loc = locss\ P'\ cs'\ loc'$
  by (*auto*, *hypsubst-thin*, *induct cs'*,
    auto intro!: *nth-equalityI simp*: *nth-Cons'*)

lemma *hd-tl-equalityI*:
  $[\![\ length\ xs = length\ ys;\ hd\ xs = hd\ ys;\ tl\ xs = tl\ ys\ ]\!] \Longrightarrow xs = ys$
apply (*induct xs arbitrary*: *ys*)
 apply *simp*
by (*case-tac ys*, *auto*)

lemma *stkLength-is-length-stk*:
  $P_{wf},P_{\Phi} \vdash (None,\ h,\ (stk,\ loc,\ C,\ M,\ pc)\ \#\ frs')\ \surd \Longrightarrow stkLength\ P\ C\ M\ pc =$
$length\ stk$
  by (*auto dest*!: *list-all2-lengthD simp*: *correct-state-def*)

lemma *locLength-is-length-loc*:
  $P_{wf},P_{\Phi} \vdash (None,\ h,\ (stk,\ loc,\ C,\ M,\ pc)\ \#\ frs')\ \surd \Longrightarrow locLength\ P\ C\ M\ pc =$
$length\ loc$
  by (*auto dest*!: *list-all2-lengthD simp*: *correct-state-def*)

lemma *correct-state-frs-tlD*:
  $(P_{wf}),(P_{\Phi}) \vdash (None,\ h,\ a\ \#\ frs')\ \surd \Longrightarrow (P_{wf}),(P_{\Phi}) \vdash (None,\ h,\ frs')\ \surd$
  by (*cases frs'*, (*fastforce simp*: *correct-state-def*)+)

lemma *update-stk-Cons* [*simp*]:
stkss P (*framestack-to-callstack frs'*) (*update-stk stk* ((*stk'*, *loc'*, *C'*, *M'*, *pc'*) # *frs'*)) =
stkss P (*framestack-to-callstack frs'*) (*update-stk stk frs'*)
apply (*induct frs' arbitrary*: *stk' loc' C' M' pc'*)
apply *clarsimp*
apply (*simp only*: *f2c-Nil*)
apply *clarsimp*
apply *clarsimp*
apply (*simp only*: *f2c-Cons*)
apply *clarsimp*
apply (*rule stkss-cong*)
by (*fastforce simp*: *nth-Cons'*)+

lemma *update-loc-Cons* [*simp*]:
locss P (*framestack-to-callstack frs'*) (*update-loc loc* ((*stk'*, *loc'*, *C'*, *M'*, *pc'*) # *frs'*)) =
locss P (*framestack-to-callstack frs'*) (*update-loc loc frs'*)
apply (*induct frs' arbitrary*: *stk' loc' C' M' pc'*)
apply *clarsimp*
apply (*simp only*: *f2c-Nil*)
apply *clarsimp*
apply *clarsimp*
apply (*simp only*: *f2c-Cons*)
apply *clarsimp*
apply (*rule locss-cong*)
by (*fastforce simp*: *nth-Cons'*)+

lemma *s2j-id*:
$(P_{wf}),(P_\Phi) \vdash$ (*None*,*h'*,*frs'*) $\surd$
$\Longrightarrow$ *state-to-jvm-state P* (*framestack-to-callstack frs'*)
(*h*, *update-stk stk frs'*, *update-loc loc frs'*) = (*None*, *h*, *frs'*)
apply (*induct frs'*)
apply *simp*
apply *simp*
apply (*rule hd-tl-equalityI*)
apply *simp*
apply *simp*
apply *clarsimp*
apply (*simp only*: *f2c-Cons fst-conv snd-conv*)
apply *clarsimp*
apply (*rule conjI*)
apply (*rule nth-equalityI*)
apply (*simp add*: *stkLength-is-length-stk*)
apply (*clarsimp simp*: *stkLength-is-length-stk*)
apply (*case-tac a*, *simp-all*)
apply (*rule nth-equalityI*)
apply (*simp add*: *locLength-is-length-loc*)
apply (*clarsimp simp*: *locLength-is-length-loc*)

apply (*drule correct-state-frs-tlD*)
apply *simp*
apply *clarsimp*
apply (*simp only*: *f2c-Cons fst-conv snd-conv*)
by *clarsimp*

lemma *find-handler-last-cs-eqD*:
⟦ *find-handler* P_{wf} *a h frs* = (*None*, *h'*, *frs'*);
last frs = (*stk*,*loc*,*C*,*M*,*pc*);
last frs' = (*stk'*,*loc'*,*C'*,*M'*,*pc'*) ⟧
⟹ *C* = *C'* ∧ *M* = *M'*
by (*induct frs*, *auto split*: *if-split-asm*)

lemma *exec-last-frs-eq-class*:
⟦ *JVMExec.exec* (P_{wf}, *None*, *h*, *frs*) = ⌊(*None*, *h'*, *frs'*)⌋;
last frs = (*stk*, *loc*, *C*, *M*, *pc*);
last frs' = (*stk'*, *loc'*, *C'*, *M'*, *pc'*);
frs ≠ [];
frs' ≠ [] ⟧
⟹ *C* = *C'*
apply (*cases frs*, *auto split*: *if-split-asm*)
apply (*cases instrs-of* P_{wf} *C M* ! *pc*, *auto simp*: *split-beta*)
apply (*case-tac instrs-of* P_{wf} *ab ac* ! *b*, *auto simp*: *split-beta*)
apply (*case-tac list*, *auto*)
apply (*case-tac lista*, *auto*)
apply (*drule find-handler-last-cs-eqD*)
apply *fastforce*
apply *fastforce*
by *simp*

lemma *exec-last-frs-eq-method*:
⟦ *JVMExec.exec* (P_{wf}, *None*, *h*, *frs*) = ⌊(*None*, *h'*, *frs'*)⌋;
last frs = (*stk*, *loc*, *C*, *M*, *pc*);
last frs' = (*stk'*, *loc'*, *C'*, *M'*, *pc'*);
frs ≠ [];
frs' ≠ [] ⟧
⟹ *M* = *M'*
apply (*cases frs*, *auto split*: *if-split-asm*)
apply (*cases instrs-of* P_{wf} *C M* ! *pc*, *auto simp*: *split-beta*)
apply (*case-tac instrs-of* P_{wf} *ab ac* ! *b*, *auto simp*: *split-beta*)
apply (*case-tac list*, *auto*)
apply (*case-tac lista*, *auto*)
apply (*drule find-handler-last-cs-eqD*)
apply *fastforce*
apply *fastforce*
by *simp*

lemma *valid-callstack-append-last-class*:

valid-callstack (*P*,*C0*,*Main*) (*cs*@[(*C*,*M*,*pc*)]) ⟹ *C* = *C0*
by (*induct cs*, *auto dest*: *valid-callstack-tl*)

lemma *valid-callstack-append-last-method*:
valid-callstack (*P*,*C0*,*Main*) (*cs*@[(*C*,*M*,*pc*)]) ⟹ *M* = *Main*
by (*induct cs*, *auto dest*: *valid-callstack-tl*)

lemma *zip-stkss-locss-append-single* [*simp*]:
zip (*stkss P* (*cs* @ [(*C*, *M*, *pc*)]) *stk*)
(*zip* (*locss P* (*cs* @ [(*C*, *M*, *pc*)]) *loc*) (*cs* @ [(*C*, *M*, *pc*)]))
= (*zip* (*stkss P* (*cs* @ [(*C*, *M*, *pc*)]) *stk*) (*zip* (*locss P* (*cs* @ [(*C*, *M*, *pc*)]) *loc*) *cs*))
@ [(*stks* (*stkLength P C M pc*) (λ*a*. *stk* (*0*, *a*)),
locs (*locLength P C M pc*) (λ*a*. *loc* (*0*, *a*)), *C*, *M*, *pc*)]
by (*induct cs*, *auto*)

7.2.3 Interpretation of the *CFG-semantics-wf* locale

interpretation *JVM-semantics-CFG-wf*:
CFG-semantics-wf sourcenode targetnode kind valid-edge prog (-*Entry*-)
sem prog identifies
for *prog*
proof(*unfold-locales*)
fix *n c s c′ s′*
assume *sem-step*:*prog* ⊢ ⟨*c*,*s*⟩ ⇒ ⟨*c′*,*s′*⟩
and *identifies n c*
obtain *P C0 M0*
where *prog* [*simp*]:*prog* = (*P*,*C0*,*M0*)
by (*cases prog*,*fastforce*)
obtain *h stk loc*
where *s* [*simp*]: *s* = (*h*,*stk*,*loc*)
by (*cases s*, *fastforce*)
obtain *h′ stk′ loc′*
where *s′* [*simp*]: *s′* = (*h′*,*stk′*,*loc′*)
by (*cases s′*, *fastforce*)
from *sem-step s s′ prog* **obtain** *C M pc cs C′ M′ pc′ cs′*
where *c* [*simp*]: *c* = (*C*,*M*,*pc*)#*cs*
by (*cases c*, *auto elim*: *sem.cases simp*: *bv-conform-def*)
with *sem-step prog* **obtain** *ST LT*
where *wt* [*simp*]: (P_{Φ}) *C M* ! *pc* = ⌊(*ST*,*LT*)⌋
by (*auto elim*!: *sem.cases*, *cases cs*, *fastforce*+)
note *P-wf* = *wf-jvmprog-is-wf* [*of P*]
from *sem-step prog* **obtain** *frs′*
where *jvm-exec*: *JVMExec.exec* ((P_{wf}), *state-to-jvm-state P c s*) = ⌊(*None*,*h′*,*frs′*)⌋
by (*auto elim*!: *sem.cases*)
with *sem-step prog s s′*
have *loc′*: *loc′* = *update-loc loc frs′*
and *stk′*: *stk′* = *update-stk stk frs′*
by (*auto elim*!: *sem.cases*)

from *sem-step s prog*
have *state-wf*: $P, c \vdash_{BV} (h, stk, loc) \sqrt{}$
  by (*auto elim*!: *sem.cases*)
hence *state-correct*: $(P_{wf}),(P_\Phi) \vdash$ *state-to-jvm-state P c* $(h, stk, loc) \sqrt{}$
  by (*simp add*: *bv-conform-def*)
with *P-wf jvm-exec s*
have *trg-state-correct*: $(P_{wf}),(P_\Phi) \vdash (None, h', frs') \sqrt{}$
  by −(*rule BV-correct-1*, (*fastforce simp*: *exec-1-iff*)+)
from *sem-step c s prog* **have** *prealloc*: *preallocated h*
  by (*auto elim*: *sem.cases*
      simp: *bv-conform-def correct-state-def hconf-def*)
from *state-correct* **obtain** *Ts T mxs mxl is xt*
  where *sees-M*: $(P_{wf}) \vdash C$ *sees* $M{:}Ts \rightarrow T = (mxs, mxl, is, xt)$ *in C*
  by (*clarsimp simp*: *bv-conform-def correct-state-def*)
with *state-correct*
have $pc <$ *length is*
  by (*auto dest*: *sees-method-fun*
      simp: *bv-conform-def correct-state-def*)
with *P-wf sees-M* **have**
  applicable: $app_i(is\ !\ pc,\ (P_{wf}),\ pc,\ mxs,\ T,\ ST,\ LT)$
  by (*fastforce dest*!: *sees-wf-mdecl*
        simp: *wf-jvm-prog-phi-def wf-mdecl-def wt-method-def*)
from *sem-step*
have *v-cs*: *valid-callstack prog c*
  by (*auto elim*: *sem.cases*)
then obtain *pcL* **where** *last-c*: *last c* = $(C0, M0, pcL)$
  apply *clarsimp*
  apply (*induct cs arbitrary*: *C M pc*, *simp*)
  by *fastforce*
from *sees-M P-wf* ‹$pc <$ *length is*›
have *wt-instrs*: $P_{wf}, T, mxs, length\ is, xt \vdash is\ !\ pc, pc :: (P_\Phi)\ C\ M$
  by −(*drule wt-jvm-prog-impl-wt-instr*, *fastforce*+)
with *applicable*
have *effect*: $\forall\, succ \in set\ (succs\ (is\ !\ pc)\ (ST, LT)\ pc).$
  $(P_{wf}) \vdash \lfloor eff_i(is\ !\ pc,\ (P_{wf}),\ ST,\ LT) \rfloor \leq' (P_\Phi)\ C\ M\ !\ succ \land succ < length\ is$
  apply *clarsimp*
  apply (*erule-tac* $x=(succ,\ \lfloor eff_i(is\ !\ pc,\ (P_{wf}),\ ST,\ LT) \rfloor\ )$ **in** *ballE*)
   by (*erule-tac* $x=(succ,\ \lfloor eff_i(is\ !\ pc,\ (P_{wf}),\ ST,\ LT) \rfloor\ )$ **in** *ballE*, *clarsimp*+)
 with *P-wf sees-M last-c v-cs*
 have *v-cs-succ*:
 $\forall\, succ \in set\ (succs\ (is\ !\ pc)\ (ST, LT)\ pc).$ *valid-callstack* $(P, C0, M0)\ ((C, M, succ)\#cs)$
  by −(*rule ballI*,
    erule-tac $x=succ$ **in** *ballE*,
    auto,
    induct cs,
    fastforce+)
 from *trg-state-correct v-cs jvm-exec*
 have *v-cs-f2c-frs'*:
  valid-callstack $(P, C0, M0)$ (*framestack-to-callstack frs'*)

apply (*cases frs' rule*: *rev-cases*, *simp*)
apply (*rule-tac s*=(*h'*, *update-stk stk frs'*, *update-loc loc frs'*)
in *correct-state-imp-valid-callstack*)
apply (*simp only*: *bv-conform-def s2j-id*)
apply (*auto dest*!: *f2c-emptyD simp del*: *exec.simps*)
apply (*cases cs rule*: *rev-cases*)
apply (*clarsimp simp del*: *exec.simps*)
apply (*drule exec-last-frs-eq-class*, *fastforce*+)
apply (*clarsimp simp del*: *exec.simps*)
apply (*simp only*: *append-Cons* [*symmetric*])
apply (*frule valid-callstack-append-last-class*)
apply (*frule valid-callstack-append-last-method*)
apply (*clarsimp simp del*: *exec.simps*)
apply (*drule exec-last-frs-eq-class*, *fastforce*+)
apply (*cases cs rule*: *rev-cases*)
apply (*clarsimp simp del*: *exec.simps*)
apply (*drule exec-last-frs-eq-method*, *fastforce*+)
apply (*clarsimp simp del*: *exec.simps*)
apply (*simp only*: *append-Cons* [*symmetric*])
apply (*frule valid-callstack-append-last-method*)
apply (*clarsimp simp del*: *exec.simps*)
by (*drule exec-last-frs-eq-method*, *fastforce*+)
show ∃ *n' as*.
CFG.path sourcenode targetnode (*valid-edge prog*) *n as n'* ∧
transfers (*CFG.kinds kind as*) *s* = *s'* ∧
preds (*CFG.kinds kind as*) *s* ∧ *identifies n' c'*
proof
show ∃ *as*. *CFG.path sourcenode targetnode* (*valid-edge prog*) *n as* (- *c'*,*None* -)
∧
transfers (*CFG.kinds kind as*) *s* = *s'* ∧
preds (*CFG.kinds kind as*) *s* ∧
identifies (- *c'*,*None* -) *c'*
proof (*cases* (*instrs-of* (P_{wf}) *C M*)!*pc*)
case (*Load nat*)
with *sem-step s s' c prog*
have *c'*: *c'* = (*C*,*M*,*pc*+*1*)#*cs*
by (*auto elim*!: *sem.cases*)
from *applicable sees-M Load*
have *nat* < *length LT*
by *simp*
from *sees-M Load* **have** *Suc pc* ∈ *set* (*succs* (*is* ! *pc*) (*ST*,*LT*) *pc*)
by *simp*
with *prog sem-step Load v-cs-succ*
have *v-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
⇑(λ*s*. *exec-instr* (*instrs-of* (P_{wf}) *C M* ! *pc*) *P s* (*length cs*) (*stkLength P C*
M pc) *0 0*),
(- (*C*,*M*,*Suc pc*)#*cs*,*None* -))
(**is** *valid-edge prog ?e1*)
by (*auto elim*!: *sem.cases intro*: *JCFG-Straight-NoExc*)

with ‹*identifies n c*› *c c'* **have** *JVM-CFG-Interpret.path prog n* [*?e1*] (-*c'*,*None* -)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Load jvm-exec loc' stk' c c' s s' prog wt* ‹*nat* < *length LT*›
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s* = *s'*
by (*auto intro*!: *ext*
simp: *JVM-CFG-Interpret.kinds-def*
nth-stkss nth-locss nth-Cons' nth-tl
not-less-eq-eq Suc-le-eq)
moreover have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
next
case (*Store nat*)
with *sem-step s s' c prog*
have *c'*: *c'* = (*C*,*M*,*pc*+*1*)#*cs*
by (*auto elim*!: *sem.cases*)
from *applicable Store sees-M*
have *length ST* > *0* ∧ *nat* < *length LT*
by *clarsimp*
then obtain *ST1 STr* **where** [*simp*]: *ST* = *ST1*#*STr* **by** (*cases ST*, *fastforce*+)
from *sees-M Store* **have** *Suc pc* ∈ *set* (*succs* (*is* ! *pc*) (*ST*, *LT*) *pc*)
by *simp*
with *prog sem-step Store v-cs-succ*
have *v-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
⇑(λ*s*. *exec-instr* (*instrs-of* (P_{wf}) *C M* ! *pc*) *P s* (*length cs*) (*stkLength P C M pc*) *0 0*),
(- (*C*,*M*,*Suc pc*)#*cs*,*None* -))
(**is** *valid-edge prog ?e1*)
by (*fastforce elim*: *sem.cases intro*: *JCFG-Straight-NoExc*)
with ‹*identifies n c*› *c c'* **have** *JVM-CFG-Interpret.path prog n* [*?e1*] (-*c'*,*None* -)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Store jvm-exec stk' loc' c c' s s' prog wt*
‹*length ST* > *0* ∧ *nat* < *length LT*›
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s* = *s'*
by (*auto intro*!: *ext*
simp: *JVM-CFG-Interpret.kinds-def*
nth-stkss nth-locss nth-Cons' nth-tl
not-less-eq-eq hd-stks)
moreover have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)

ultimately show *?thesis* **by** *fastforce*
next
case (*Push val*)
with *sem-step s s′ c prog*
have *c′*: *c′* = (*C*,*M*,*pc*+*1*)#*cs*
by (*auto elim*!: *sem.cases*)
from *sees-M Push* **have** *Suc pc* ∈ *set* (*succs* (*is* ! *pc*) (*ST*, *LT*) *pc*)
by *simp*
with *prog sem-step Push v-cs-succ*
have *v-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
⇑(λ*s*. *exec-instr* (*instrs-of* (P_{wf}) *C M* ! *pc*) *P s* (*length cs*) (*stkLength P C M pc*) *0 0*),
(- (*C*,*M*,*Suc pc*)#*cs*,*None* -))
(**is** *valid-edge prog ?e1*)
by (*fastforce elim*: *sem.cases intro*: *JCFG-Straight-NoExc*)
with ‹*identifies n c*› *c c′* **have** *JVM-CFG-Interpret.path prog n* [*?e1*] (-*c′*,*None* -)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Push jvm-exec stk′ loc′ c c′ s s′ prog wt*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s* = *s′*
by (*auto intro*!: *ext*
simp: *JVM-CFG-Interpret.kinds-def*
nth-stkss nth-locss nth-Cons′ nth-tl
not-less-eq-eq)
moreover have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
next
case (*New Cl*)
show *?thesis*
proof (*cases new-Addr h*)
case *None*
with *New sem-step s s′ c prog prealloc*
have *c′*: *c′* = *find-handler-for P OutOfMemory c*
by (*fastforce elim*!: *sem.cases*
dest: *find-handler-find-handler-forD*)
with *jvm-exec New None prealloc*
have *f2c-frs′-c′*: *framestack-to-callstack frs′* = *c′*
by (*auto dest*!: *find-handler-find-handler-forD*)
with *New c′ v-cs v-cs-f2c-frs′*
have *v-pred-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
(λ(*h*,*stk*,*loc*). *new-Addr h* = *None*)$_{\surd}$,
(- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*True*)⌋ -))
(**is** *valid-edge prog ?e1*)
apply *auto*
apply (*rule JCFG-New-Exc-Pred*, *fastforce*+)

apply (*rule-tac x*=(λ(*h*, *stk*, *loc*). *new-Addr h* = *None*) **in** *exI*)
apply (*rule JCFG-New-Exc-Pred*, *fastforce*+)
apply (*cases find-handler-for P OutOfMemory cs*)
apply (*rule exI*)
apply *clarsimp*
apply (*rule JCFG-New-Exc-Exit*, *fastforce*+)
apply *clarsimp*
apply (*rule-tac x*=λ(*h*, *stk*, *loc*).
(*h*, *stk*((*length list*, *stkLength P a aa b* − *Suc 0*) :=
Addr (*addr-of-sys-xcpt OutOfMemory*)),
loc) **in** *exI*)
apply (*rule JCFG-New-Exc-Update*, *fastforce*+)
apply (*rule JCFG-New-Exc-Pred*, *fastforce*+)
apply (*rule exI*)
apply (*rule JCFG-New-Exc-Pred*, *fastforce*+)
apply (*rule exI*)
by (*rule JCFG-New-Exc-Update*, *fastforce*+)
show *?thesis*
proof (*cases c′*)
case *Nil*
with *prog sem-step New c*
have *v-exec-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊([],*True*)⌋ -),
⇑*id*,
(-*Exit*-))
(**is** *valid-edge prog ?e2*)
by (*fastforce elim*: *sem.cases intro*: *JCFG-New-Exc-Exit*)
with *v-pred-edge* ‹*identifies n c*› *c c′ Nil*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (-*Exit*-)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Nil None New sem-step c c′ s s′ prog*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s′*
by (*auto elim*!: *sem.cases simp*: *JVM-CFG-Interpret.kinds-def*)
moreover from *None s* **have** *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*])
s
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **using** *Nil* **by** *fastforce*
next
case (*Cons a cs′*)
then obtain *C′ M′ pc′* **where** *Cons*: *c′* = (*C′*,*M′*,*pc′*)#*cs′* **by** (*cases a*,
fastforce)
from *jvm-exec c s None New*
have *update-loc loc frs′* = *loc*
by −(*rule find-handler-loc-fun-eq′* [*of P - h* (*C*,*M*,*pc*)#*cs stk loc*], *simp*)
with *loc′* **have** *loc′* = *loc*
by *simp*

from *c Cons s s′ sem-step jvm-exec prog*
have (*C′*,*M′*,*pc′*)#*cs′* = *framestack-to-callstack frs′*
by (*auto elim*!: *sem.cases*)
moreover obtain *stk″ loc″ frs″* **where** *frs′*: *frs′* = (*stk″*,*loc″*,*C′*,*M′*,*pc′*)#*frs″*
and *cs′*: *cs′* = *framestack-to-callstack frs″* **using** *calculation*
by (*cases frs′*, *fastforce*+)
ultimately
have *update-stk stk frs′* =
stk((*length cs′*,*stkLength P C′ M′ pc′* − *Suc 0*) := *Addr* (*addr-of-sys-xcpt OutOfMemory*))
using *c s c′ None Cons prog New trg-state-correct wt jvm-exec prealloc stk′*
by −(*rule find-handler-stk-fun-eq′* [*of P - h* (*C*,*M*,*pc*)#*cs - loc h′*],
auto dest!: *list-all2-lengthD*
simp: *hd-stks split-beta framestack-to-callstack-def*
correct-state-def)
with *stk′* **have** *stk′*:
stk′ =
stk((*length cs′*,*stkLength P C′ M′ pc′* − *Suc 0*) := *Addr* (*addr-of-sys-xcpt OutOfMemory*))
by *simp*
from *New Cons v-cs-f2c-frs′ v-cs f2c-frs′-c′*
have *v-exec-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*True*)⌋ -),
⇑(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs′*,(*stkLength P C′ M′ pc′*) − *1*) :=
Addr (*addr-of-sys-xcpt OutOfMemory*)),
loc)
),
(- *c′*,*None* -))
(**is** *valid-edge prog ?e2*)
apply *auto*
apply (*rule JCFG-New-Exc-Update*)
apply *fastforce*
apply *fastforce*
using *Cons c′* **apply** *simp*
apply *simp*
using *v-pred-edge c′ Cons* **apply** *clarsimp*
using *v-pred-edge c′ Cons* **apply** *clarsimp*
done
with *v-pred-edge* ‹*identifies n c*› *c c′ Nil*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (- *c′*,*None* -)
by −(*rule JVM-CFG-Interpret.path.Cons-path*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*+)
moreover from *New c c′ s s′ loc′ stk′* ‹*loc′* = *loc*› *prog jvm-exec None*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s′*
by (*auto dest*: *find-handler-heap-eqD*

```
              simp: JVM-CFG-Interpret.kinds-def)
      moreover from None s
      have preds (JVM-CFG-Interpret.kinds [?e1,?e2]) s
        by (simp add: JVM-CFG-Interpret.kinds-def)
      ultimately show ?thesis by fastforce
    qed
  next
    case (Some obj)
    with New sem-step s s' c prog prealloc
    have c': c' = (C,M,Suc pc)#cs
      by (auto elim!: sem.cases)
    with New jvm-exec Some
    have f2c-frs'-c': framestack-to-callstack frs' = c'
      by auto
    with New c' v-cs v-cs-f2c-frs'
    have v-pred-edge: valid-edge prog ((- (C,M,pc)#cs,None -),
      (λ(h,stk,loc). new-Addr h ≠ None)√,
      (- (C,M,pc)#cs,⌊(c',False)⌋ -))
      (is valid-edge prog ?e1)
      apply auto
        apply (fastforce intro!: JCFG-New-Normal-Pred)
       apply (rule exI)
       apply (fastforce intro!: JCFG-New-Normal-Pred)
      apply (rule exI)
      by (fastforce intro!: JCFG-New-Normal-Update)
    from New sees-M have Suc pc ∈ set (succs (is ! pc) (ST, LT) pc)
      by simp
    with prog New c' sem-step prog v-cs-succ
    have v-exec-edge:valid-edge prog ((- (C,M,pc)#cs,⌊(c',False)⌋ -),
      ⇑(λs. exec-instr (instrs-of (P_wf) C M ! pc) P s (length cs) (stkLength P
C M pc) 0 0),
      (- (C,M,Suc pc)#cs,None -))
      (is valid-edge prog ?e2)
   by (auto elim!: sem.cases intro: JCFG-New-Normal-Update JCFG-New-Normal-Pred)
    with v-pred-edge ‹identifies n c› c c'
    have JVM-CFG-Interpret.path prog n [?e1,?e2] (- c',None -)
      by −(simp,
        rule JVM-CFG-Interpret.path.Cons-path,
        rule JVM-CFG-Interpret.path.Cons-path,
        rule JVM-CFG-Interpret.path.empty-path,
        auto simp: JVM-CFG-Interpret.valid-node-def, fastforce)
    moreover from New jvm-exec loc' stk' c c' s s' prog Some
    have transfers (JVM-CFG-Interpret.kinds [?e1,?e2]) s = s'
      by (auto intro!: ext
              simp: JVM-CFG-Interpret.kinds-def
                 nth-stkss nth-locss nth-Cons'
                 not-less-eq-eq hd-stks)
    moreover from Some s
    have preds (JVM-CFG-Interpret.kinds [?e1,?e2]) s
```

by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
qed
next
case (*Getfield Fd Cl*)
with *applicable sees-M*
have *length ST* > *0*
by *clarsimp*
then obtain *ST1 STr* **where** *ST* [*simp*]: *ST* = *ST1#STr* **by** (*cases ST*, *fastforce+*)
show *?thesis*
proof (*cases stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null*)
case *True*
with *Getfield sem-step s s′ c prog prealloc wt*
have *c′*: *c′* = *find-handler-for P NullPointer c*
by (*cases the* (*h* (*the-Addr Null*)),
auto elim!: *sem.cases*
dest!: *find-handler-find-handler-forD*
simp: *hd-stks*)
with *Getfield True jvm-exec prealloc*
have *framestack-to-callstack frs′* = *c′*
by (*auto simp*: *split-beta dest*!: *find-handler-find-handler-forD*)
with *Getfield prog c′ v-cs v-cs-f2c-frs′*
have *v-pred-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
(λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null*)$_{\surd}$,
(- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*True*)⌋ -))
(**is** *valid-edge prog ?e1*)
apply (*auto simp del*: *find-handler-for.simps*)
apply (*fastforce intro*!: *JCFG-Getfield-Exc-Pred*)
apply (*fastforce intro*!: *JCFG-Getfield-Exc-Pred*)
apply *auto*
apply (*cases find-handler-for P NullPointer cs*)
apply (*fastforce intro*!: *JCFG-Getfield-Exc-Exit*)
apply (*fastforce intro*!: *JCFG-Getfield-Exc-Update*)
apply (*fastforce intro*!: *JCFG-Getfield-Exc-Update*)
done
show *?thesis*
proof (*cases c′*)
case *Nil*
with *Getfield c prog c′ v-pred-edge*
have *v-exec-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊([],*True*)⌋ -),
⇑*id*,
(-*Exit*-))
(**is** *valid-edge prog ?e2*)
by (*fastforce intro*: *JCFG-Getfield-Exc-Exit*)
with *v-pred-edge* ‹*identifies n c*› *c c′ Nil*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (-*Exit*-)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,

rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Nil True Getfield sem-step c c′ s s′ prog wt* ‹*length ST* > *0*›
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s′*
by (*auto elim*!: *sem.cases*
simp: *hd-stks split-beta JVM-CFG-Interpret.kinds-def*)
moreover from *True s*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **using** *Nil* **by** *fastforce*
next
case (*Cons a cs′*)
then obtain *C′ M′ pc′* **where** *Cons*: *c′* = (*C′*,*M′*,*pc′*)#*cs′* **by** (*cases a*, *fastforce*)
from *jvm-exec c s True Getfield wt ST*
have *update-loc loc frs′* = *loc*
by −(*rule find-handler-loc-fun-eq′* [*of P - h* (*C*,*M*,*pc*)#*cs stk loc*],
auto simp: *split-beta hd-stks*)
with *loc′* **have** *loc′* = *loc*
by *simp*
from *c Cons s s′ sem-step jvm-exec prog*
have *cs′-f2c-frs′*: (*C′*,*M′*,*pc′*)#*cs′* = *framestack-to-callstack frs′*
by (*auto elim*!: *sem.cases*)
moreover obtain *stk″ loc″ frs″* **where** *frs′* = (*stk″*,*loc″*,*C′*,*M′*,*pc′*)#*frs″*
and *cs′* = *framestack-to-callstack frs″* **using** *calculation*
by (*cases frs′*, *fastforce*+)
ultimately
have *update-stk stk frs′* =
stk((*length cs′*,*stkLength P C′ M′ pc′* − *Suc 0*) := *Addr* (*addr-of-sys-xcpt NullPointer*))
using *c s c′ True Cons prog Getfield trg-state-correct wt ST jvm-exec prealloc stk′*
by −(*rule find-handler-stk-fun-eq′* [*of P - h* (*C*,*M*,*pc*)#*cs - loc h′*],
auto dest!: *list-all2-lengthD*
simp: *hd-stks split-beta framestack-to-callstack-def*
correct-state-def)
with *stk′* **have** *stk′*:
stk′ =
stk((*length cs′*,*stkLength P C′ M′ pc′* − *Suc 0*) := *Addr* (*addr-of-sys-xcpt NullPointer*))
by *simp*
from *prog Cons Getfield c′ v-cs v-cs-f2c-frs′ jvm-exec*
have *v-exec-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*True*)⌋ -),
⇑(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs′*,(*stkLength P C′ M′ pc′*) − *1*) :=
Addr (*addr-of-sys-xcpt NullPointer*)),

```
            loc)
          ),
          (- c',None -))
          (is valid-edge prog ?e2)
          apply (auto simp del: exec.simps find-handler-for.simps)
              apply (rule JCFG-Getfield-Exc-Update, fastforce+)
             apply (simp only: cs'-f2c-frs')
            apply (fastforce intro!: JCFG-Getfield-Exc-Pred)
           apply (fastforce intro!: JCFG-Getfield-Exc-Update)
          by (simp only: cs'-f2c-frs')
        with v-pred-edge ‹identifies n c› c c' Nil
        have JVM-CFG-Interpret.path prog n [?e1,?e2] (- c',None -)
          by −(rule JVM-CFG-Interpret.path.Cons-path,
            rule JVM-CFG-Interpret.path.Cons-path,
            rule JVM-CFG-Interpret.path.empty-path,
            auto simp: JVM-CFG-Interpret.valid-node-def, fastforce+)
        moreover from Getfield c c' s s' loc' stk' prog True jvm-exec
          ‹loc' = loc› wt ST
        have transfers (JVM-CFG-Interpret.kinds [?e1,?e2]) s = s'
          by (auto dest: find-handler-heap-eqD
                  simp: JVM-CFG-Interpret.kinds-def split-beta hd-stks)
        moreover from True s
        have preds (JVM-CFG-Interpret.kinds [?e1,?e2]) s
          by (simp add: JVM-CFG-Interpret.kinds-def)
        ultimately show ?thesis by fastforce
      qed
    next
      case False
      with Getfield sem-step s s' c prog prealloc wt ‹length ST > 0›
      have c': c' = (C,M,Suc pc)#cs
        by (auto elim!: sem.cases
                simp: split-beta hd-stks)
      with False Getfield jvm-exec prealloc
      have framestack-to-callstack frs' = c'
        by (auto dest!: find-handler-find-handler-forD simp: split-beta)
      with Getfield c' v-cs v-cs-f2c-frs'
      have v-pred-edge: valid-edge prog ((- (C,M,pc)#cs,None -),
        (λ(h,stk,loc). stk(length cs, stkLength P C M pc − 1) ≠ Null)√,
        (- (C,M,pc)#cs,⌊(c',False)⌋ -))
        (is valid-edge prog ?e1)
        apply auto
          apply (fastforce intro: JCFG-Getfield-Normal-Pred)
         apply (fastforce intro: JCFG-Getfield-Normal-Pred)
        by (fastforce intro: JCFG-Getfield-Normal-Update)
      with prog c' Getfield v-cs-succ sees-M
      have v-exec-edge:valid-edge prog ((- (C,M,pc)#cs,⌊(c',False)⌋ -),
        ⇑(λs. exec-instr (instrs-of (P_wf) C M ! pc) P s (length cs) (stkLength P
C M pc) 0 0),
        (- (C,M,Suc pc)#cs,None -))
```

(**is** *valid-edge prog ?e2*)
by (*fastforce intro*: *JCFG-Getfield-Normal-Update*)
with *v-pred-edge* ‹*identifies n c*› *c c′*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (- *c′*,*None* -)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Getfield jvm-exec stk′ loc′ c c′ s s′ prog False wt ST*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s′*
by (*auto intro*!: *ext*
simp: *nth-stkss nth-locss nth-tl nth-Cons′ hd-stks*
not-less-eq-eq split-beta JVM-CFG-Interpret.kinds-def)
moreover from *False s*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
qed
next
case (*Putfield Fd Cl*)
with *applicable sees-M*
have *length ST* > *1*
by *clarsimp*
then obtain *ST1 STr′* **where** *ST* = *ST1*#*STr′*
by (*cases ST*, *fastforce*+)
with ‹*length ST* > *1*› **obtain** *ST2 STr*
where *ST*: *ST* = *ST1*#*ST2*#*STr*
by (*cases STr′*, *fastforce*+)
show *?thesis*
proof (*cases stk*(*length cs*, *stkLength P C M pc* − *2*) = *Null*)
case *True*
with *Putfield sem-step s s′ c prog prealloc wt* ‹*length ST* > *1*›
have *c′*: *c′* = *find-handler-for P NullPointer c*
by (*auto elim*!: *sem.cases*
dest!: *find-handler-find-handler-forD*
simp: *hd-tl-stks split-beta*)
with *Putfield jvm-exec True prealloc* ‹*length ST* > *1*› *wt*
have *framestack-to-callstack frs′* = *c′*
by (*auto dest*!: *find-handler-find-handler-forD simp*: *split-beta hd-tl-stks*)
with *Putfield c′ v-cs v-cs-f2c-frs′*
have *v-pred-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
(λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *2*) = *Null*)$_{\surd}$,
(- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*True*)⌋ -))
(**is** *valid-edge prog ?e1*)
apply (*auto simp del*: *find-handler-for.simps*)
apply (*fastforce intro*: *JCFG-Putfield-Exc-Pred*)
apply (*fastforce intro*: *JCFG-Putfield-Exc-Pred*)
apply (*cases find-handler-for P NullPointer* ((*C*, *M*, *pc*)#*cs*))

apply (*fastforce intro*: *JCFG-Putfield-Exc-Exit*)
by (*fastforce intro*: *JCFG-Putfield-Exc-Update*)
show *?thesis*
proof (*cases c′*)
case *Nil*
with *Putfield c prog c′ v-pred-edge*
have *v-exec-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊([],*True*)⌋ -),
⇑*id*,
(-*Exit*-))
(**is** *valid-edge prog ?e2*)
by (*fastforce intro*: *JCFG-Putfield-Exc-Exit*)
with *v-pred-edge* ‹*identifies n c*› *c c′ Nil*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (-*Exit*-)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Nil True Putfield sem-step c c′ s s′ prog wt ST*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s′*
by (*auto elim*!: *sem.cases*
simp: *split-beta JVM-CFG-Interpret.kinds-def*)
moreover from *True s*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **using** *Nil* **by** *fastforce*
next
case (*Cons a cs′*)
then obtain *C′ M′ pc′* **where** *Cons*: *c′* = (*C′*,*M′*,*pc′*)#*cs′* **by** (*cases a*, *fastforce*)
from *jvm-exec c s True Putfield ST wt*
have *update-loc loc frs′* = *loc*
by −(*rule find-handler-loc-fun-eq′* [*of P - h* (*C*,*M*,*pc*)#*cs stk loc*],
auto simp: *split-beta hd-tl-stks if-split-eq1*)
with *sem-step s s′ c prog jvm-exec*
have *loc′*:*loc′* = *loc*
by (*clarsimp elim*!: *sem.cases*)
from *c Cons s s′ sem-step jvm-exec prog*
have *stk′* = *update-stk stk frs′*
and *cs′-f2c-frs′*: (*C′*,*M′*,*pc′*)#*cs′* = *framestack-to-callstack frs′*
by (*auto elim*!: *sem.cases*)
moreover obtain *stk′′ loc′′ frs′′* **where** *frs′* = (*stk′′*,*loc′′*,*C′*,*M′*,*pc′*)#*frs′′*
and *cs′* = *framestack-to-callstack frs′′* **using** *calculation*
by (*cases frs′*, *fastforce*+)
ultimately
have *stk′*:
update-stk stk frs′ =
stk((*length cs′*,*stkLength P C′ M′ pc′* − *Suc 0*) := *Addr* (*addr-of-sys-xcpt NullPointer*))

```
      using c s Cons True prog Putfield ST wt trg-state-correct jvm-exec
      by −(rule find-handler-stk-fun-eq′ [of P - h (C,M,pc)#cs - loc h′],
        auto dest!: list-all2-lengthD
              simp: hd-stks hd-tl-stks split-beta framestack-to-callstack-def
                    correct-state-def)
    from Cons Putfield c prog c′ v-pred-edge v-cs-f2c-frs′ jvm-exec
    have v-exec-edge:valid-edge prog ((- (C,M,pc)#cs,⌊(c′,True)⌋ -),
      ⇑(λ(h,stk,loc).
        (h, stk((length cs′,(stkLength P C′ M′ pc′) − 1) :=
              Addr (addr-of-sys-xcpt NullPointer)), loc) ),
      (- c′,None -))
      (is valid-edge prog ?e2)
      by (auto intro!: JCFG-Putfield-Exc-Update)
    with v-pred-edge ‹identifies n c› c c′ Nil
    have JVM-CFG-Interpret.path prog n [?e1,?e2] (- c′,None -)
      by −(rule JVM-CFG-Interpret.path.Cons-path,
        rule JVM-CFG-Interpret.path.Cons-path,
        rule JVM-CFG-Interpret.path.empty-path,
        auto simp: JVM-CFG-Interpret.valid-node-def, fastforce+)
   moreover from True Putfield c c′ s s′ loc′ stk′ ‹stk′ = update-stk stk frs′›
                jvm-exec wt ST
    have transfers (JVM-CFG-Interpret.kinds [?e1,?e2]) s = s′
      by (auto dest: find-handler-heap-eqD
               simp: JVM-CFG-Interpret.kinds-def hd-tl-stks split-beta)
    moreover from True s
    have preds (JVM-CFG-Interpret.kinds [?e1,?e2]) s
      by (simp add: JVM-CFG-Interpret.kinds-def)
    ultimately show ?thesis by fastforce
  qed
next
  case False
  with Putfield sem-step s s′ c prog prealloc wt ‹length ST > 1›
  have c′: c′ = (C,M,Suc pc)#cs
    by (auto elim!: sem.cases
            simp: hd-tl-stks split-beta)
  with Putfield False jvm-exec ‹length ST > 1› wt
  have framestack-to-callstack frs′ = c′
    by (auto simp: split-beta hd-tl-stks)
  with Putfield c′ v-cs v-cs-f2c-frs′
  have v-pred-edge: valid-edge prog ((- (C,M,pc)#cs,None -),
    (λ(h,stk,loc). stk(length cs, stkLength P C M pc − 2) ≠ Null)√,
    (- (C,M,pc)#cs,⌊(c′,False)⌋ -))
    (is valid-edge prog ?e1)
    apply auto
      apply (fastforce intro: JCFG-Putfield-Normal-Pred)
     apply (fastforce intro: JCFG-Putfield-Normal-Pred)
    by (fastforce intro: JCFG-Putfield-Normal-Update)
  with prog Putfield c′ v-cs-succ sees-M
  have v-exec-edge:valid-edge prog ((- (C,M,pc)#cs,⌊(c′,False)⌋ -),
```

```
        ⇑(λs. exec-instr (instrs-of (P_wf) C M ! pc) P s (length cs) (stkLength P
C M pc) 0 0),
        (- (C,M,Suc pc)#cs,None -))
        (is valid-edge prog ?e2)
        by (fastforce intro: JCFG-Putfield-Normal-Update)
      with v-pred-edge ‹identifies n c› c c'
      have JVM-CFG-Interpret.path prog n [?e1,?e2] (- c',None -)
        by −(simp,
          rule JVM-CFG-Interpret.path.Cons-path,
          rule JVM-CFG-Interpret.path.Cons-path,
          rule JVM-CFG-Interpret.path.empty-path,
          auto simp: JVM-CFG-Interpret.valid-node-def, fastforce)
      moreover from Putfield jvm-exec stk' loc' c c' s s' prog False wt ST
      have transfers (JVM-CFG-Interpret.kinds [?e1,?e2]) s = s'
        by (auto intro!: ext
                simp: JVM-CFG-Interpret.kinds-def split-beta
                      nth-stkss  nth-locss nth-Cons'
                     not-less-eq-eq)
      moreover from False s
      have preds (JVM-CFG-Interpret.kinds [?e1,?e2]) s
        by (simp add: JVM-CFG-Interpret.kinds-def)
      ultimately show ?thesis by fastforce
    qed
  next
    case (Checkcast Cl)
    with applicable sees-M
    have length ST > 0
      by clarsimp
   then obtain ST1 STr where ST: ST = ST1#STr by (cases ST, fastforce+)
    show ?thesis
    proof (cases ¬ cast-ok (P_wf) Cl h (stk(length cs,length ST − Suc 0)))
      case True
      with Checkcast sem-step s s' c prog prealloc wt ‹length ST > 0›
      have c': c' = find-handler-for P ClassCast c
        by (auto elim!: sem.cases
                dest!: find-handler-find-handler-forD
                 simp: hd-stks split-beta)
      with jvm-exec Checkcast True prealloc ‹length ST > 0› wt
      have framestack-to-callstack frs' = c'
        by (auto dest!: find-handler-find-handler-forD simp: hd-stks)
      with Checkcast c' v-cs v-cs-f2c-frs'
      have v-pred-edge:valid-edge prog ((- (C,M,pc)#cs,None -),
        (λ(h,stk,loc). ¬ cast-ok (P_wf) Cl h (stk(length cs, stkLength P C M pc −
Suc 0)))√,
        (- (C,M,pc)#cs,⌊(c',True)⌋ -))
        (is valid-edge prog ?e1)
        apply (auto simp del: find-handler-for.simps)
         apply (fastforce intro: JCFG-Checkcast-Exc-Pred)
        apply (fastforce intro: JCFG-Checkcast-Exc-Pred)
```

apply (*cases find-handler-for P ClassCast* ((*C*,*M*,*pc*)#*cs*))
apply (*fastforce intro*: *JCFG-Checkcast-Exc-Exit*)
by (*fastforce intro*: *JCFG-Checkcast-Exc-Update*)
show *?thesis*
proof (*cases c'*)
case *Nil*
with *Checkcast c prog c' v-pred-edge*
have *v-exec-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊([],*True*)⌋ -),
⇑*id*,
(-*Exit*-))
(**is** *valid-edge prog ?e2*)
by (*fastforce intro*: *JCFG-Checkcast-Exc-Exit*)
with *v-pred-edge* ‹*identifies n c*› *c c' Nil*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (-*Exit*-)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Nil True Checkcast sem-step c c' s s' prog wt* ‹*length ST* > *0*›
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s'*
by (*auto elim*!: *sem.cases*
simp: *hd-stks split-beta JVM-CFG-Interpret.kinds-def*)
moreover from *True s wt*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **using** *Nil* **by** *fastforce*
next
case (*Cons a cs'*)
then obtain *C' M' pc'* **where** *Cons*: *c'* = (*C'*,*M'*,*pc'*)#*cs'* **by** (*cases a*, *fastforce*)
from *jvm-exec c s True Checkcast ST wt*
have *loc''*: *update-loc loc frs'* = *loc*
by −(*rule find-handler-loc-fun-eq'* [*of P - h* (*C*,*M*,*pc*)#*cs stk loc*],
auto simp: *split-beta hd-tl-stks if-split-eq1*)
from *c Cons s s' sem-step jvm-exec prog*
have *stk'* = *update-stk stk frs'*
and [*simp*]: *framestack-to-callstack frs'* = (*C'*, *M'*, *pc'*)#*cs'*
by (*auto elim*!: *sem.cases*)
moreover obtain *stk'' loc'' frs''* **where** *frs'* = (*stk''*,*loc''*,*C'*,*M'*,*pc'*)#*frs''*
and *cs'* = *framestack-to-callstack frs''* **using** *calculation*
by (*cases frs'*, *fastforce*+)
ultimately
have *stk''*:
update-stk stk frs' =
stk((*length cs'*,*stkLength P C' M' pc'* − *Suc 0*) := *Addr* (*addr-of-sys-xcpt ClassCast*))
using *c s Cons True prog Checkcast ST wt trg-state-correct jvm-exec*

by −(*rule find-handler-stk-fun-eq′* [*of P - h* (*C*,*M*,*pc*)#*cs - loc h′*],
auto dest!: *list-all2-lengthD*
simp: *hd-stks hd-tl-stks split-beta framestack-to-callstack-def*
correct-state-def)
from *prog Checkcast Cons c c′ v-pred-edge v-cs-f2c-frs′*
have *v-exec-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*True*)⌋ -),
⇑(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs′*,(*stkLength P C′ M′ pc′*) − *1*) :=
Addr (*addr-of-sys-xcpt ClassCast*)),
loc)
),
(- *c′*,*None* -))
(**is** *valid-edge prog ?e2*)
by (*auto intro*!: *JCFG-Checkcast-Exc-Update*)
with *v-pred-edge* ‹*identifies n c*› *c c′ Nil*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (- *c′*,*None* -)
by −(*rule JVM-CFG-Interpret.path.Cons-path*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*+)
moreover from *True Checkcast c s s′ loc′ stk′ loc″ stk″*
prog wt ST jvm-exec
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s′*
by (*auto dest*: *find-handler-heap-eqD*
simp: *JVM-CFG-Interpret.kinds-def split-beta*)
moreover from *True s wt*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
qed
next
case *False*
with *Checkcast sem-step s s′ c prog prealloc wt* ‹*length ST* > *0*›
have *c′*: *c′* = (*C*,*M*,*Suc pc*)#*cs*
by (*auto elim*!: *sem.cases*
simp: *hd-stks split-beta*)
with *prog Checkcast sem-step c s v-cs-succ sees-M*
have *v-pred-edge*: *valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
(λ(*h*,*stk*,*loc*). *cast-ok* (P_{wf}) *Cl h* (*stk*(*length cs*, *stkLength P C M pc* − *Suc*
0)))$_{\surd}$,
(- (*C*,*M*,*Suc pc*)#*cs*,*None* -))
(**is** *valid-edge prog ?e1*)
by (*auto intro*!: *JCFG-Checkcast-Normal-Pred elim*: *sem.cases*)
with ‹*identifies n c*› *c c′*
have *JVM-CFG-Interpret.path prog n* [*?e1*] (- *c′*,*None* -)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,

auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Checkcast jvm-exec stk′ loc′ c s s′ prog False wt ST*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s* = *s′*
by (*auto elim*!: *sem.cases*
intro!: *ext*
simp: *split-beta hd-stks JVM-CFG-Interpret.kinds-def*
nth-stkss nth-locss nth-Cons′
not-less-eq-eq)
moreover from *False s wt*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
qed
next
case (*Invoke M′ n′*)
with *applicable sees-M*
have *length ST* > *n′*
by *clarsimp*
moreover obtain *STn* **where** *STn* = *take n′ ST* **by** *fastforce*
moreover obtain *STs* **where** *STs* = *ST* ! *n′* **by** *fastforce*
moreover obtain *STr* **where** *STr* = *drop* (*Suc n′*) *ST* **by** *fastforce*
ultimately have *ST*: *ST* = *STn*@*STs*#*STr* ∧ *length STn* = *n′*
by (*auto simp*: *id-take-nth-drop*)
with *jvm-exec c s Invoke wt*
have *h* = *h′*
by (*auto dest*: *find-handler-heap-eqD*
simp: *split-beta nth-Cons′ if-split-eq1*)
show *?thesis*
proof (*cases stk*(*length cs*, *stkLength P C M pc* − *Suc n′*) = *Null*)
case *True*
with *Invoke sem-step prog prealloc wt ST*
have *c′*: *c′* = *find-handler-for P NullPointer c*
apply (*auto elim*!: *sem.cases*
simp: *split-beta nth-Cons′ ST*
split: *if-split-asm*)
by (*auto dest*!: *find-handler-find-handler-forD*)
with *jvm-exec True Invoke wt ST prealloc*
have *framestack-to-callstack frs′* = *c′*
by (*auto dest*!: *find-handler-find-handler-forD*
simp: *split-beta nth-Cons′ if-split-eq1*)
with *Invoke c′ v-cs v-cs-f2c-frs′*
have *v-pred-edge*: *valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
(λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *Suc n′*) = *Null*)$_{\surd}$,
(- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*True*)⌋ -))
(**is** *valid-edge prog ?e1*)
apply (*auto simp del*: *find-handler-for.simps*)
apply (*fastforce intro*: *JCFG-Invoke-Exc-Pred*)
apply (*fastforce intro*: *JCFG-Invoke-Exc-Pred*)
apply (*cases find-handler-for P NullPointer* ((*C*, *M*, *pc*) # *cs*))

apply (*fastforce intro*: *JCFG-Invoke-Exc-Exit*)
by (*fastforce intro*: *JCFG-Invoke-Exc-Update*)
show *?thesis*
proof (*cases c′*)
case *Nil*
with *prog Invoke c c′ v-pred-edge*
have *v-exec-edge*: *valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊([],*True*)⌋ -),
⇑*id*,
(-*Exit*-))
(**is** *valid-edge prog ?e2*)
by (*fastforce intro*: *JCFG-Invoke-Exc-Exit*)
with *v-pred-edge* ‹*identifies n c*› *c c′ Nil*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (- *c′*,*None* -)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Invoke jvm-exec stk′ loc′ c c′ s s′*
prog True wt ST prealloc Nil ‹*h* = *h′*›
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s′*
by (*auto dest*!: *find-handler-find-handler-forD*
simp: *split-beta JVM-CFG-Interpret.kinds-def*
nth-Cons′ if-split-eq1 framestack-to-callstack-def)
moreover from *s True*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
next
case (*Cons a cs′*)
then obtain *C′ M′ pc′* **where** *Cons*: *c′* = (*C′*,*M′*,*pc′*)#*cs′*
by (*cases a*, *fastforce*)
from *jvm-exec c s True Invoke ST wt*
have *loc′′*: *update-loc loc frs′* = *loc*
by −(*rule find-handler-loc-fun-eq′* [*of P - h* (*C*,*M*,*pc*)#*cs stk loc*],
auto simp: *split-beta if-split-eq1 nth-Cons′*)
from *c Cons s s′ sem-step jvm-exec prog*
have *stk′* = *update-stk stk frs′*
and [*simp*]: *framestack-to-callstack frs′* = (*C′*,*M′*,*pc′*)#*cs′*
by (*auto elim*!: *sem.cases*)
moreover obtain *stk′′ loc′′ frs′′* **where** *frs′* = (*stk′′*,*loc′′*,*C′*,*M′*,*pc′*)#*frs′′*
and *cs′* = *framestack-to-callstack frs′′* **using** *calculation*
by (*cases frs′*, *fastforce*+)
ultimately
have *stk′′*:
update-stk stk frs′ =
stk((*length cs′*,*stkLength P C′ M′ pc′* − *Suc 0*) := *Addr* (*addr-of-sys-xcpt*
NullPointer))
using *c s Cons True prog Invoke ST wt trg-state-correct jvm-exec*

by −(*rule find-handler-stk-fun-eq′* [*of P - h* (*C*,*M*,*pc*)#*cs* - *loc h′*],
auto dest!: *list-all2-lengthD*
simp: *nth-Cons′ split-beta correct-state-def if-split-eq1*)
from *Cons Invoke c prog c′ v-pred-edge v-cs-f2c-frs′*
have *v-exec-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*True*)⌋ -),
⇑(λ(*h*,*stk*,*loc*).
(*h*, *stk*((*length cs′*,(*stkLength P C′ M′ pc′*) − *1*) :=
Addr (*addr-of-sys-xcpt NullPointer*)), *loc*)),
(- *c′*,*None* -))
(**is** *valid-edge prog ?e2*)
by (*auto intro*!: *JCFG-Invoke-Exc-Update*)
with *v-pred-edge* ‹*identifies n c*› *c c′ Nil*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (- *c′*,*None* -)
by −(*rule JVM-CFG-Interpret.path.Cons-path*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*+)
moreover from *Cons True Invoke jvm-exec c c′ s s′ loc′ stk′ loc″ stk″*
prog wt ST ‹*h* = *h′*›
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s′*
by (*auto simp*: *JVM-CFG-Interpret.kinds-def split-beta*)
moreover from *True s*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
qed
next
case *False*
obtain *D* **where** *D*:
D = *fst* (*method* P_{wf} (*cname-of h* (*the-Addr* (*stk* (*length cs*, *length ST* − *Suc n′*)))) *M′*)
by *simp*
from *c wt s state-correct*
have (P_{wf}),*h* ⊢ *stks* (*length ST*) (λ*a*. *stk* (*length cs*, *a*)) [:≤] *ST*
by (*clarsimp simp*: *bv-conform-def correct-state-def*)
with *False ST wt*
have *STs* ≠ *NT*
apply −
apply (*drule-tac p*=*n′* **in** *list-all2-nthD*)
apply *simp*
apply (*auto simp*: *nth-Cons′ split*: *if-split-asm*)
apply *hypsubst-thin*
by (*induct STn*, *auto simp*: *nth-Cons′ split*: *if-split-asm*)
with *applicable ST Invoke sees-M*
obtain *D′* **where** *D′*: *STs* = *Class D′*
by (*clarsimp simp*: *nth-append*)
from *Invoke c s jvm-exec False wt ST D*
obtain *loc″* **where** *frs′*: *frs′* = ([],*loc″*,*D*,*M′*,*0*)#(*snd*(*snd*(*state-to-jvm-state P c s*)))

by (*auto simp*: *split-beta if-split-eq1 nth-Cons′ ST*)
with *trg-state-correct*
obtain *Ts′ T′ mb′* **where** *D-sees-M′*: $(P_{wf}) \vdash D$ *sees M′*:*Ts′*→*T′* = *mb′ in D*
by (*auto simp*: *correct-state-def*)
from *state-correct c s wt ST D′*
have *stk-wt*: P_{wf},*h* ⊢ *stk* (*length cs*, *length STn* + *length STr*) #
stks (*length STn* + *length STr*) (λ*a*. *stk* (*length cs*, *a*)) [:≤] *STn* @ *Class D′* # *STr*
by (*auto simp*: *correct-state-def*)
have (*stk* (*length cs*, *length STn* + *length STr*) #
stks (*length STn* + *length STr*) (λ*a*. *stk* (*length cs*, *a*))) ! *length STn* =
stk (*length cs*, *length STr*)
by (*auto simp*: *nth-Cons′ ST*)
with *stk-wt*
have P_{wf},*h* ⊢ *stk* (*length cs*, *length STr*) :≤ *Class D′*
by (*drule-tac P*=*conf* (P_{wf}) *h* **and** *p*=*length STn* **in** *list-all2-nthD*,
auto simp: *nth-append*)
with *False ST wt*
have *subD′*: $(P_{wf}) \vdash$ (*cname-of h* (*the-Addr* (*stk* (*length cs*, *length ST* − *Suc n′*)))) ⪯* *D′*
by (*cases stk* (*length cs*, *length STr*), *auto simp*: *conf-def*)
from *trg-state-correct frs′ D-sees-M′ Invoke s c*
have *length Ts′* = *n′*
by (*auto dest*: *sees-method-fun simp*: *correct-state-def*)
with *c trg-state-correct wt ST D-sees-M′ D P-wf frs′ subD′ D′*
obtain *Ts T mxs mxl is xt*
where *stk-sees-M′*:
$(P_{wf}) \vdash$ (*cname-of h* (*the-Addr* (*stk* (*length cs*, *length ST* − *Suc n′*))))
sees M′:*Ts*→*T* = (*mxs*,*mxl*,*is*,*xt*) *in D*
by (*auto dest*: *sees-method-fun*
dest!: *sees-method-mono*
simp: *correct-state-def split-beta nth-append wf-jvm-prog-phi-def*
simp del: *ST*)
with *c s False jvm-exec Invoke frs′ wt* ‹*length ST* > *n′*›
have *loc′′*:
loc′′ = *stk* (*length cs*, *length ST* − *Suc n′*) #
rev (*take n′* (*stks* (*length ST*) (λ*a*. *stk*(*length cs*, *a*)))) @
replicate mxl arbitrary
by (*auto simp*: *split-beta if-split-eq1 simp del*: *ST*)
with *trg-state-correct frs′ Invoke wt* ‹*length ST* > *n′*›
have *locLength-trg*:
locLength P D M′ 0 = *n′* + *Suc mxl*
by (*auto dest*: *list-all2-lengthD simp*: *correct-state-def*)
from *stk′ frs′ c s*
have *stk′* = *stk*
by (*auto intro*!: *ext*
simp: *nth-stkss nth-Cons′ not-less-eq-eq Suc-le-eq*
simp del: *ST*)

from *loc′ frs′ c s loc′′ wt ST*
have *upd-loc′*: *loc′* = (λ(*a*, *b*).
  if a = *Suc* (*length cs*) ⟶ *Suc* (*n′* + *mxl*) ≤ *b then loc* (*a*, *b*)
  else if b ≤ *n′ then stk* (*length cs*, *Suc* (*n′* + *length STr*) − (*Suc n′* − *b*))
    else arbitrary)
  by (*auto intro*!: *ext*
      simp: *nth-locss nth-Cons′ nth-append rev-nth*
        not-less-eq-eq Suc-le-eq less-Suc-eq add.commute
        min.absorb1 min.absorb2 max.absorb1 max.absorb2)
from *frs′ jvm-exec sem-step prog*
have *c′*: *c′* = (*D*,*M′*,*0*)#*c*
  by (*auto elim*!: *sem.cases*)
from *frs′*
have *framestack-to-callstack frs′* = (*D*, *M′*, *0*) # (*C*, *M*, *pc*) # *cs*
  by *simp*
with *Invoke c′ v-cs v-cs-f2c-frs′*
have *v-pred-edge*: *valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
  (λ(*h*,*stk′*,*loc*).
    stk′(*length cs*, *stkLength P C M pc* − *Suc n′*) ≠ *Null* ∧
    fst(*method* (P_{wf})
      (*cname-of h* (*the-Addr*(*stk′*(*length cs*, *stkLength P C M pc* − *Suc*
n′)))) *M′*
) = *D*
)$_{\surd}$,
  (- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*False*)⌋ -))
  (**is** *valid-edge prog ?e1*)
  apply *auto*
   apply (*fastforce intro*: *JCFG-Invoke-Normal-Pred*)
  apply (*fastforce intro*: *JCFG-Invoke-Normal-Pred*)
  apply (*rule exI*)
  by (*fastforce intro*: *JCFG-Invoke-Normal-Update*)
with *Invoke v-cs-f2c-frs′ c′ v-cs*
have *v-exec-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*False*)⌋ -),
  ⇑(λ*s*.
    exec-instr (*instrs-of* (P_{wf}) *C M* ! *pc*) *P s*
      (*length cs*) (*stkLength P C M pc*) *0* (*locLength P D M′ 0*)
  ),
  (- (*D*,*M′*,*0*)#*c*,*None* -))
  (**is** *valid-edge prog ?e2*)
  by (*fastforce intro*!: *JCFG-Invoke-Normal-Update*
      simp del: *exec.simps valid-callstack.simps*)
with *v-pred-edge* ‹*identifies n c*› *c c′ locLength-trg*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (- *c′*,*None* -)
  by −(*simp*,
    rule JVM-CFG-Interpret.path.Cons-path,
    rule JVM-CFG-Interpret.path.Cons-path,
    rule JVM-CFG-Interpret.path.empty-path,
    auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *s s′* ‹*h* = *h′*› ‹*stk′* = *stk*› *upd-loc′*

locLength-trg stk-sees-M′ Invoke c wt ST
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s′*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
moreover from *False s D wt* **have** *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
qed
next
case *Return*
with *applicable sees-M*
have *length ST* > *0*
by *clarsimp*
then obtain *ST1 STr* **where** *ST*: *ST* = *ST1#STr* **by** (*cases ST*, *fastforce+*)
show *?thesis*
proof (*cases cs*)
case *Nil*
with *sem-step s s′ c prog Return*
have *c′*: *c′* = [] ∧ *C* = *C0* ∧ *M* = *M0*
by (*auto elim*!: *sem.cases*)
with *prog sem-step Return Nil c*
have *v-edge*: *valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
⇑*id*,
(-*Exit*-))
(**is** *valid-edge prog ?e1*)
by (*fastforce intro*: *JCFG-ReturnExit elim*: *sem.cases*)
with ‹*identifies n c*› *c c′* **have** *JVM-CFG-Interpret.path prog n* [*?e1*] (-*c′*,*None* -)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Return sem-step c c′ s s′ prog wt Nil* ‹*length ST* > *0*›
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s* = *s′*
by (*auto elim*!: *sem.cases simp*: *JVM-CFG-Interpret.kinds-def*)
moreover have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
next
case (*Cons a cs′*)
with *c* **obtain** *D M′ pc′* **where** *c*: *c* = (*C*,*M*,*pc*)#(*D*,*M′*,*pc′*)#*cs′* **by** (*cases a*, *fastforce*)
with *prog sem-step Return*
have *c′*: *c′* = (*D*,*M′*,*Suc pc′*)#*cs′*
by (*auto elim*!: *sem.cases*)
from *c s jvm-exec Return*
have *h* = *h′*
by (*auto simp*: *split-beta*)
from *c s jvm-exec loc′ Return*

```
      have loc′ = loc
        by (auto intro!: ext
                simp: split-beta not-less-eq-eq Suc-le-eq not-less-eq less-Suc-eq-le
                      nth-locss hd-stks nth-Cons′)
      from c s jvm-exec stk′ Return ST wt trg-state-correct
      have stk-upd:
        stk′ =
        stk((length cs′, stkLength P D M′ (Suc pc′) − 1) :=
          stk(Suc (length cs′), length ST − 1))
        by (auto intro!: ext
               dest!: list-all2-lengthD
                simp: split-beta not-less-eq-eq Suc-le-eq
                      nth-stkss hd-stks nth-Cons′ correct-state-def)
      from jvm-exec Return c′ c
      have framestack-to-callstack frs′ = c′
        by auto
      with Return v-cs v-cs-f2c-frs′ c′ c
      have v-edge: valid-edge prog ((- (C,M,pc)#(D,M′,pc′)#cs′,None -),
        ⇑(λs. exec-instr Return P s
           (Suc (length cs′)) (stkLength P C M pc) (stkLength P D M′ (Suc pc′))
0),
        (- (D,M′,Suc pc′)#cs′,None -))
        (is valid-edge prog ?e1)
        by (fastforce intro: JCFG-Return-Update)
      with ‹identifies n c› c c′
      have JVM-CFG-Interpret.path prog n [?e1] (- c′,None -)
        by −(simp,
          rule JVM-CFG-Interpret.path.Cons-path,
          rule JVM-CFG-Interpret.path.empty-path,
          auto simp: JVM-CFG-Interpret.valid-node-def, fastforce)
      moreover from stk′ loc′ s s′ ‹h = h′› ‹loc′ = loc› stk-upd wt
      have transfers (JVM-CFG-Interpret.kinds [?e1]) s = s′
        by (simp add: JVM-CFG-Interpret.kinds-def)
      moreover have preds (JVM-CFG-Interpret.kinds [?e1]) s
        by (simp add: JVM-CFG-Interpret.kinds-def)
      ultimately show ?thesis by fastforce
    qed
  next
    case Pop
    with sem-step s s′ c prog
    have c′: c′ = (C,M,pc+1)#cs
      by (auto elim!: sem.cases)
    from Pop sees-M applicable
    have ST ≠ []
      by clarsimp
    then obtain ST1 STr where ST: ST = ST1#STr
      by (cases ST, fastforce+)
    with c′ jvm-exec Pop
    have framestack-to-callstack frs′ = c′
```

by *auto*
with *Pop v-cs v-cs-f2c-frs′ c′*
have *v-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
⇑(λ*s*. *exec-instr* (*instrs-of* (P_{wf}) *C M* ! *pc*) *P s* (*length cs*) (*stkLength P C M pc*) *0 0*),
(- (*C*,*M*,*Suc pc*)#*cs*,*None* -))
(**is** *valid-edge prog ?e1*)
by (*fastforce intro*: *JCFG-Straight-NoExc*)
with ‹*identifies n c*› *c c′* **have** *JVM-CFG-Interpret.path prog n* [*?e1*] (- *c′*,*None* -)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Pop jvm-exec s s′ stk′ loc′ c wt ST*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s* = *s′*
by (*auto intro*!: *ext*
simp: *nth-stkss nth-locss nth-Cons′ nth-tl*
not-less-eq-eq Suc-le-eq JVM-CFG-Interpret.kinds-def)
moreover have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
next
case *IAdd*
with *sem-step s s′ c prog*
have *c′*: *c′* = (*C*,*M*,*pc*+*1*)#*cs*
by (*auto elim*!: *sem.cases*)
from *IAdd applicable sees-M*
have *length ST* > *1*
by *clarsimp*
then obtain *ST1 STr′* **where** *ST* = *ST1*#*STr′* **by** (*cases ST*, *fastforce*+)
with ‹*length ST* > *1*› **obtain** *ST2 STr*
where *ST*: *ST* = *ST1*#*ST2*#*STr* **by** (*cases STr′*, *fastforce*+)
from *c′ jvm-exec IAdd*
have *framestack-to-callstack frs′* = *c′*
by *auto*
with *IAdd c′ v-cs v-cs-f2c-frs′*
have *v-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
⇑(λ*s*. *exec-instr* (*instrs-of* (P_{wf}) *C M* ! *pc*) *P s* (*length cs*) (*stkLength P C M pc*) *0 0*),
(- (*C*,*M*,*Suc pc*)#*cs*,*None* -))
(**is** *valid-edge prog ?e1*)
by (*fastforce intro*: *JCFG-Straight-NoExc*)
with ‹*identifies n c*› *c c′*
have *JVM-CFG-Interpret.path prog n* [*?e1*] (- *c′*,*None* -)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)

moreover from *IAdd jvm-exec c s s' stk' loc' wt ST*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s* = *s'*
by (*auto intro*!: *ext*
simp: *nth-stkss nth-locss nth-Cons' nth-tl*
hd-stks hd-tl-stks
not-less-eq-eq Suc-le-eq JVM-CFG-Interpret.kinds-def)
moreover have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
next
case (*IfFalse b*)
with *applicable sees-M*
have *ST* ≠ []
by *clarsimp*
then obtain *ST1 STr* **where** *ST* [*simp*]: *ST* = *ST1*#*STr* **by** (*cases ST*, *fastforce*+)
show *?thesis*
proof (*cases stk* (*length cs*, *stkLength P C M pc* − *1*) = *Bool False* ∧ *b* ≠ *1*)
case *True*
with *sem-step s s' c prog IfFalse wt ST*
have *c'*: *c'* = (*C*,*M*,*nat* (*int pc* + *b*))#*cs*
by (*auto elim*!: *sem.cases*
simp: *hd-stks*)
with *jvm-exec IfFalse True*
have *framestack-to-callstack frs'* = *c'*
by *auto*
with *c' IfFalse True v-cs v-cs-f2c-frs'*
have *v-edge*: *valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
(λ(*h*,*stk*,*loc*). *stk* (*length cs*, *stkLength P C M pc* − *1*) = *Bool False*)$_\surd$,
(- (*C*,*M*,*nat* (*int pc* + *b*))#*cs*,*None* -))
(**is** *valid-edge prog ?e1*)
by (*fastforce intro*: *JCFG-IfFalse-False*)
with ‹*identifies n c*› *c c'* **have** *JVM-CFG-Interpret.path prog n* [*?e1*] (-*c'*,*None* -)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *IfFalse True jvm-exec c s s' stk' loc' wt ST*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s* = *s'*
by (*auto intro*!: *ext*
simp: *hd-stks nth-stkss nth-locss nth-Cons' nth-tl*
JVM-CFG-Interpret.kinds-def not-less-eq-eq)
moreover from *True s*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
next
case *False*

have *nat* (*int pc* + *1*) = *Suc pc*
  by (*cases pc*, *auto*)
with *False sem-step s s′ c prog IfFalse wt ST*
have *c′*: *c′* = (*C*,*M*,*Suc pc*)#*cs*
  by (*auto elim*!: *sem.cases simp*: *hd-stks*)
with *jvm-exec IfFalse False*
have *framestack-to-callstack frs′* = *c′*
  by *auto*
with *c′ IfFalse False v-cs v-cs-f2c-frs′*
have *v-edge*: *valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
  (λ(*h*,*stk*,*loc*). *stk* (*length cs*, *stkLength P C M pc* − *1*) ≠ *Bool False* ∨ *b* =
1)$_{\surd}$,
  (- (*C*,*M*,*Suc pc*)#*cs*,*None* -))
  (**is** *valid-edge prog ?e1*)
  by (*fastforce intro*: *JCFG-IfFalse-Next*)
with ‹*identifies n c*› *c c′*
have *JVM-CFG-Interpret.path prog n* [*?e1*] (- *c′*,*None* -)
  by −(*simp*,
    rule JVM-CFG-Interpret.path.Cons-path,
    rule JVM-CFG-Interpret.path.empty-path,
    auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *IfFalse False jvm-exec c s s′ stk′ loc′ wt ST*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s* = *s′*
  by (*auto intro*!: *ext*
        simp: *hd-stks nth-stkss nth-locss nth-Cons′ nth-tl*
          JVM-CFG-Interpret.kinds-def not-less-eq-eq)
moreover from *False s*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s*
  by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
qed
next
case (*Goto i*)
with *sem-step s s′ c prog*
have *c′*: *c′* = (*C*,*M*,*nat* (*int pc* + *i*))#*cs*
  by (*auto elim*!: *sem.cases*)
with *jvm-exec Goto*
have *framestack-to-callstack frs′* = *c′*
  by *auto*
with *c′ Goto v-cs v-cs-f2c-frs′*
have *v-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
  ⇑*id*,
  (- (*C*,*M*,*nat* (*int pc* + *i*))#*cs*,*None* -))
  (**is** *valid-edge prog ?e1*)
  by (*fastforce intro*: *JCFG-Goto-Update*)
with ‹*identifies n c*› *c c′*
have *JVM-CFG-Interpret.path prog n* [*?e1*] (- *c′*,*None* -)
  by −(*simp*,
    rule JVM-CFG-Interpret.path.Cons-path,

rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *Goto jvm-exec c s s′ stk′ loc′*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s* = *s′*
by (*auto intro*!: *ext*
simp: *nth-stkss nth-locss nth-Cons′*
JVM-CFG-Interpret.kinds-def not-less-eq-eq)
moreover have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
next
case *CmpEq*
with *sem-step s s′ c prog*
have *c′*: *c′* = (*C*,*M*,*Suc pc*)#*cs*
by (*auto elim*!: *sem.cases*)
from *CmpEq applicable sees-M*
have *length ST* > *1*
by *clarsimp*
then obtain *ST1 STr′* **where** *ST* = *ST1*#*STr′* **by** (*cases ST*, *fastforce*+)
with ‹*length ST* > *1*› **obtain** *ST2 STr*
where *ST*: *ST* = *ST1*#*ST2*#*STr* **by** (*cases STr′*, *fastforce*+)
from *c′ CmpEq jvm-exec*
have *framestack-to-callstack frs′* = *c′*
by *auto*
with *c′ CmpEq v-cs v-cs-f2c-frs′*
have *v-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
⇑(λ*s*. *exec-instr* (*instrs-of* (P_{wf}) *C M* ! *pc*) *P s* (*length cs*) (*stkLength P C M pc*) *0 0*),
(- (*C*,*M*,*Suc pc*)#*cs*,*None* -))
(**is** *valid-edge prog ?e1*)
by (*fastforce intro*: *JCFG-Straight-NoExc*)
with ‹*identifies n c*› *c c′*
have *JVM-CFG-Interpret.path prog n* [*?e1*] (- *c′*,*None* -)
by −(*simp*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
moreover from *CmpEq jvm-exec c s s′ stk′ loc′ wt ST*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s* = *s′*
by (*auto intro*!: *ext*
simp: *nth-stkss nth-locss nth-Cons′ nth-tl*
hd-stks hd-tl-stks
not-less-eq-eq JVM-CFG-Interpret.kinds-def)
moreover have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
next
case *Throw*
with *sees-M applicable*

have *ST* ≠ []
  by *clarsimp*
then obtain *ST1 STr* **where** *ST*: *ST* = *ST1*#*STr* **by** (*cases ST*, *fastforce*+)
from *jvm-exec sem-step*
have *f2c-frs′-eq-c′*: *framestack-to-callstack frs′* = *c′*
  by (*auto elim*: *sem.cases*)
show *?thesis*
proof (*cases stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null*)
  case *True*
  with *sem-step Throw s s′ c prog wt ST prealloc*
  have *c′*:*c′* = *find-handler-for P NullPointer c*
    by (*fastforce elim*!: *sem.cases*
          dest: *find-handler-find-handler-forD*
          simp: *hd-stks*)
  with *Throw v-cs v-cs-f2c-frs′ f2c-frs′-eq-c′ prealloc*
  have *v-pred-edge*: *valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
    (λ(*h*,*stk*,*loc*).
      (*stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null* ∧
       find-handler-for P NullPointer ((*C*,*M*,*pc*)#*cs*) = *c′*) ∨
      (*stk*(*length cs*, *stkLength P C M pc* − *1*) ≠ *Null* ∧
       find-handler-for P (*cname-of h* (*the-Addr*(*stk*(*length cs*, *stkLength P C*
M pc − *1*))))
          ((*C*,*M*,*pc*)#*cs*) = *c′*)
)$_{\surd}$,
    (- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*True*)⌋ -))
  (**is** *valid-edge prog ?e1*)
  apply (*auto simp del*: *find-handler-for.simps*)
    apply (*fastforce intro*: *JCFG-Throw-Pred*)
   apply (*fastforce intro*: *JCFG-Throw-Pred*)
  apply (*cases find-handler-for P NullPointer* ((*C*, *M*, *pc*) # *cs*))
   apply (*fastforce intro*: *JCFG-Throw-Exit*)
  by (*fastforce intro*: *JCFG-Throw-Update*)
  show *?thesis*
  proof (*cases c′*)
    case *Nil*
    with *prog Throw c c′ sem-step v-pred-edge*
    have *v-exec-edge*: *valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊([],*True*)⌋ -),
      ⇑*id*,
      (-*Exit*-))
      (**is** *valid-edge prog ?e2*)
      by (*auto intro*: *JCFG-Throw-Exit*)
    with *v-pred-edge* ‹*identifies n c*› *c c′ Nil*
    have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (- *c′*,*None* -)
      by −(*simp*,
        rule JVM-CFG-Interpret.path.Cons-path,
        rule JVM-CFG-Interpret.path.Cons-path,
        rule JVM-CFG-Interpret.path.empty-path,
        auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*)
    moreover from *Throw jvm-exec c c′ s s′ stk′ loc′*

True Nil wt ST trg-state-correct prealloc
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s'*
by (*cases frs'*,
auto dest: *find-handler-find-handler-forD*
simp: *JVM-CFG-Interpret.kinds-def split-beta correct-state-def*)
moreover from *True s wt c' c* **have** *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
next
case (*Cons a cs'*)
then obtain *C' M' pc'*
where *Cons*: *c'* = (*C'*,*M'*,*pc'*)#*cs'* **by** (*cases a*, *fastforce*)
with *jvm-exec s loc' c True Throw wt ST*
have *loc'* = *loc*
by (*auto intro*!: *ext*
simp: *find-handler-loc-fun-eq'*
not-less-eq-eq nth-Cons' nth-locss)
from *c Cons s s' sem-step jvm-exec prog*
have *stk'* = *update-stk stk frs'*
and (*C'*,*M'*,*pc'*)#*cs'* = *framestack-to-callstack frs'*
by (*auto elim*!: *sem.cases*)
moreover obtain *stk'' loc'' frs''* **where** *frs'* = (*stk''*,*loc''*,*C'*,*M'*,*pc'*)#*frs''*
and *cs'* = *framestack-to-callstack frs''* **using** *calculation*
by (*cases frs'*, *fastforce+*)
ultimately
have *stk''*:
update-stk stk frs' =
stk((*length cs'*,*stkLength P C' M' pc'* − *Suc 0*) := *Addr* (*addr-of-sys-xcpt NullPointer*))
using *c s Cons True prog Throw ST wt trg-state-correct jvm-exec*
by −(*rule find-handler-stk-fun-eq'* [*of P - h* (*C*,*M*,*pc*)#*cs - loc h'*],
auto dest!: *list-all2-lengthD*
simp: *nth-Cons' split-beta correct-state-def if-split-eq1*)
from ‹(*C'*,*M'*,*pc'*)#*cs'* = *framestack-to-callstack frs'*› *Cons*
have *framestack-to-callstack frs'* = *c'*
by *simp*
with *Cons Throw v-cs v-cs-f2c-frs' v-pred-edge*
have *v-exec-edge*:
valid-edge prog ((- (*C*,*M*,*pc*)#*cs*,⌊(*c'*,*True*)⌋ -),
⇑(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs'*,*stkLength P C' M' pc'* − *1*) :=
if (*stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null*)
then Addr (*addr-of-sys-xcpt NullPointer*)
else (*stk*(*length cs*, *stkLength P C M pc* − *1*))),
loc)
),
(- *c'*,*None* -))

(**is** *valid-edge prog ?e2*)
by (*auto intro*!: *JCFG-Throw-Update*)
with *v-pred-edge* ‹*identifies n c*› *c c′ True prog*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (- *c′*,*None* -)
by −(*rule JVM-CFG-Interpret.path.Cons-path*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*+)
moreover from *Cons True Throw jvm-exec c c′ s s′* ‹*loc′* = *loc*› *stk′ stk″*
wt ST
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s′*
by (*auto dest*: *find-handler-heap-eqD simp*: *JVM-CFG-Interpret.kinds-def*)
moreover from *True s wt c c′*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
qed
next
case *False*
with *sem-step Throw s s′ c prog wt ST prealloc*
have *c′*:
c′ = *find-handler-for P*
(*cname-of h* (*the-Addr*(*stk*(*length cs*, *stkLength P C M pc* − *1*)))) *c*
by (*fastforce elim*!: *sem.cases*
dest: *find-handler-find-handler-forD*
simp: *hd-stks*)
with *Throw v-cs v-cs-f2c-frs′ f2c-frs′-eq-c′*
have *v-pred-edge*: *valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,*None* -),
(λ(*h*,*stk*,*loc*).
(*stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null* ∧
find-handler-for P NullPointer ((*C*,*M*,*pc*)#*cs*) = *c′*) ∨
(*stk*(*length cs*, *stkLength P C M pc* − *1*) ≠ *Null* ∧
find-handler-for P (*cname-of h* (*the-Addr*(*stk*(*length cs*, *stkLength P C*
M pc − *1*))))
((*C*,*M*,*pc*)#*cs*) = *c′*)
)$_\surd$,
(- (*C*,*M*,*pc*)#*cs*,⌊(*c′*,*True*)⌋ -))
(**is** *valid-edge prog ?e1*)
apply (*auto simp del*: *find-handler-for.simps*)
apply (*fastforce intro*: *JCFG-Throw-Pred*)
apply (*fastforce intro*: *JCFG-Throw-Pred*)
apply (*cases find-handler-for P*
(*cname-of h* (*the-Addr*(*stk*(*length cs*, *stkLength P C M pc* − *1*))))
((*C*,*M*,*pc*)#*cs*))
apply (*fastforce intro*: *JCFG-Throw-Exit*)
by (*fastforce intro*: *JCFG-Throw-Update*)
show *?thesis*
proof (*cases c′*)
case *Nil*

with *prog Throw c c′ v-pred-edge*
have *v-exec-edge*: *valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊([],*True*)⌋ -),
⇑*id*,
(-*Exit*-))
(**is** *valid-edge prog ?e2*)
by (*auto intro*!: *JCFG-Throw-Exit*)
with *v-pred-edge* ‹*identifies n c*› *c c′ Nil*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (- *c′*,*None* -)
by −(*rule JVM-CFG-Interpret.path.Cons-path*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*+)
moreover from *Throw jvm-exec c c′ s s′ False Nil trg-state-correct wt ST*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s′*
by (*cases frs′*,
auto dest: *find-handler-find-handler-forD*
simp: *JVM-CFG-Interpret.kinds-def correct-state-def*)
moreover from *False s wt c′ c*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
next
case (*Cons a cs′*)
then obtain *C′ M′ pc′*
where *Cons*: *c′* = (*C′*,*M′*,*pc′*)#*cs′* **by** (*cases a*, *fastforce*)
with *jvm-exec s loc′ c Throw wt ST*
have *loc′* = *loc*
by (*auto intro*!: *ext*
simp: *find-handler-loc-fun-eq′*
not-less-eq-eq nth-Cons′ nth-locss)
from *c Cons s s′ sem-step jvm-exec prog*
have *stk′* = *update-stk stk frs′*
and (*C′*,*M′*,*pc′*)#*cs′* = *framestack-to-callstack frs′*
by (*auto elim*!: *sem.cases*)
moreover obtain *stk′′ loc′′ frs′′* **where** *frs′* = (*stk′′*,*loc′′*,*C′*,*M′*,*pc′*)#*frs′′*
and *cs′* = *framestack-to-callstack frs′′* **using** *calculation*
by (*cases frs′*, *fastforce*+)
ultimately
have *stk′′*:
update-stk stk frs′ =
stk((*length cs′*,*stkLength P C′ M′ pc′* − *Suc 0*) :=
Addr (*the-Addr* (*stk*((*length cs*, *stkLength P C M pc* − *Suc 0*)))))
using *c s Cons False prog Throw ST wt trg-state-correct jvm-exec*
by −(*rule find-handler-stk-fun-eq′* [*of P - h* (*C*,*M*,*pc*)#*cs - loc h*′],
auto dest!: *list-all2-lengthD*
simp: *nth-Cons′ split-beta correct-state-def if-split-eq1*)
from *applicable False Throw ST sees-M*
have *is-refT ST1*
by *clarsimp*

with *state-correct wt ST c False*
have *addr-the-addr-stk-eq*:
Addr(*the-Addr*(*stk*(*length cs*, *length STr*))) = *stk*(*length cs*, *length STr*)
by (*cases stk* (*length cs*, *length STr*),
auto simp: *correct-state-def is-refT-def conf-def*)
from ‹(C',M',pc')#cs' = *framestack-to-callstack frs'*› *Cons*
have *framestack-to-callstack frs'* = c'
by *simp*
with *Cons Throw v-cs v-cs-f2c-frs' v-pred-edge*
have *v-exec-edge*:*valid-edge prog* ((- (*C*,*M*,*pc*)#*cs*,⌊(c',*True*)⌋ -),
⇑(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs'*,*stkLength P C' M' pc'* − *1*) :=
if (*stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null*)
then Addr (*addr-of-sys-xcpt NullPointer*)
else (*stk*(*length cs*, *stkLength P C M pc* − *1*))),
loc)),
(- c',*None* -))
(**is** *valid-edge prog ?e2*)
by (*auto intro*!: *JCFG-Throw-Update*)
with *v-pred-edge* ‹*identifies n c*› *c c' Nil*
have *JVM-CFG-Interpret.path prog n* [*?e1*,*?e2*] (- c',*None* -)
by −(*rule JVM-CFG-Interpret.path.Cons-path*,
rule JVM-CFG-Interpret.path.Cons-path,
rule JVM-CFG-Interpret.path.empty-path,
auto simp: *JVM-CFG-Interpret.valid-node-def*, *fastforce*+)
moreover from *Cons False Throw jvm-exec c c' s s' loc' stk'*
addr-the-addr-stk-eq prog wt ST ‹*loc'* = *loc*› *stk''*
have *transfers* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s* = *s'*
by (*auto dest*: *find-handler-heap-eqD*
simp: *JVM-CFG-Interpret.kinds-def*)
moreover from *False s wt c c'*
have *preds* (*JVM-CFG-Interpret.kinds* [*?e1*,*?e2*]) *s*
by (*simp add*: *JVM-CFG-Interpret.kinds-def*)
ultimately show *?thesis* **by** *fastforce*
qed
qed
qed
qed
qed

end
theory *Slicing*
imports
Basic/Postdomination
Basic/CFGExit-wf
Basic/SemanticsCFG
Dynamic/DynSlice

StaticIntra/CDepInstantiations
StaticIntra/ControlDependenceRelations
While/DynamicControlDependences
While/StaticControlDependences
JinjaVM/JVMControlDependences
JinjaVM/SemanticsWF
begin

end

Bibliography

[1] Daniel Wasserrab and Denis Lohner and Gregor Snelting On PDG-Based Noninterference and its Modular Proof In *Proc. of PLAS'09*, pages 31–44. ACM, 2009.

[2] Daniel Wasserrab and Andreas Lochbihler. Formalizing a framework for dynamic slicing of program dependence graphs in Isabelle/HOL. In *Proc. of TPHOLS'08*, pages 294–309. Springer-Verlag, 2008.

[3] Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.