

Towards Certified Slicing

Daniel Wasserrab

May 24, 2012

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 $\exists x \in \text{set } xs. P x$

obtains $zs\ x'\ ys$ **where** $xs = zs @ x' \# ys$ **and** $P\ x'$ **and** $\forall z \in \text{set } zs. \neg P\ z$

proof(*atomize-elim*)

from $\langle \exists x \in \text{set } xs. P\ x \rangle$

show $\exists zs\ x'\ ys. xs = zs @ x' \# ys \wedge P\ x' \wedge (\forall z \in \text{set } zs. \neg P\ z)$

proof(*induct xs*)

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

next

case (*Cons* $x'\ xs'$)

note $IH = \langle \exists a \in \text{set } xs'. P\ a \rangle$

$\implies \exists zs\ x'\ ys. xs' = zs @ x' \# ys \wedge P\ x' \wedge (\forall z \in \text{set } zs. \neg P\ z)$

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  $\langle \exists y \in set\ (x' \# xs'). P\ y \rangle$  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  $\langle \forall z \in set\ zs. \neg P\ z \rangle$  False have  $\forall z \in set\ (x' \# zs). \neg P\ z$  by simp
    with  $\langle xs' = zs @ y \# ys \rangle$   $\langle P\ y \rangle$  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  $\langle \exists x \in set\ xs. P\ x \rangle$ 
  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  $IH = \langle \exists a \in set\ xs'. P\ a \implies \exists ys\ x'\ zs. xs' = ys @ x' \# zs \wedge P\ x' \wedge (\forall z \in set\ zs. \neg P\ z) \rangle$ 
    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  $\langle \exists y \in set\ (x' \# xs'). P\ y \rangle$  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  $\langle \forall zs. xs \neq ys @ zs \rangle$ 
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  $IH = \langle \bigwedge xs. \forall zs. xs \neq ys' @ zs \implies$ 
 $\exists j\ zs. xs = take\ j\ ys' @ zs \wedge j < length\ ys' \wedge$ 
 $(\forall k. j < k \implies (\forall zs'. xs \neq take\ k\ ys' @ zs')) \rangle$ 
show ?case
proof(cases xs)
case Nil thus ?thesis by simp
next
case (Cons x' xs')
with  $\langle \forall zs. xs \neq (y' \# ys') @ zs \rangle$  have  $x' \neq y' \vee (\forall zs. xs' \neq ys' @ zs)$ 
by simp
show ?thesis
proof(cases x' = y')
case True
with  $\langle x' \neq y' \vee (\forall zs. xs' \neq ys' @ zs) \rangle$  have  $\forall zs. xs' \neq ys' @ zs$  by simp
from  $IH[OF\ this]$  have  $\exists j\ zs. xs' = take\ j\ ys' @ zs \wedge j < length\ ys' \wedge$ 
 $(\forall k. j < k \implies (\forall zs'. xs' \neq take\ k\ ys' @ zs'))$  .
then obtain  $j\ zs$  where  $xs' = take\ j\ ys' @ zs$ 
and  $j < length\ ys'$ 
and  $all-sub:\forall k. j < k \implies (\forall zs'. xs' \neq take\ k\ ys' @ zs')$ 
by blast
from  $\langle xs' = take\ j\ ys' @ zs \rangle$  True
have  $(x' \# xs') = take\ (Suc\ j)\ (y' \# ys') @ zs$ 
by simp
from all-sub True have all-imp: $\forall k. j < k \implies$ 
 $(\forall zs'. (x' \# xs') \neq take\ (Suc\ k)\ (y' \# ys') @ zs')$ 
by auto
{ fix l assume (Suc j) < l
then obtain k where [simp]:l = Suc k by(cases l) auto
with  $\langle (Suc\ j) < l \rangle$  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  $\langle (x' \# xs') = take\ (Suc\ j)\ (y' \# ys') @ zs \rangle$   $\langle j < length\ ys' \rangle$  Cons
show ?thesis by (metis Suc-length-conv less-Suc-eq-0-disj)

```

```

next
  case False
  with Cons have  $\forall i \, zs'. \, i > 0 \longrightarrow xs \neq take \, i \, (y' \# ys') @ zs'$ 
    by auto(case-tac i,auto)
  moreover
  have  $\exists zs. \, xs = take \, 0 \, (y' \# ys') @ zs$  by simp
  ultimately show ?thesis by(rule-tac x=0 in exI,auto)
qed
qed
qed
end

```

Chapter 1

The Framework

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

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

1.1.1 Edge kinds

datatype 'state edge-kind = Update 'state $\Rightarrow$ 'state $\quad (\uparrow-)$
| Predicate 'state $\Rightarrow$ bool $\quad (('-')_{\checkmark})$

1.1.2 Transfer and predicate functions

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

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

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

fun preds :: 'state edge-kind list $\Rightarrow$ 'state $\Rightarrow$ bool

where $\text{preds } [] \ s = \text{True}$
 $| \text{preds } (e \# es) \ s = (\text{pred } e \ s \wedge \text{preds } es \ (\text{transfer } e \ s))$

lemma *transfers-split*:
 $(\text{transfers } (ets @ ets') \ s) = (\text{transfers } ets' \ (\text{transfers } ets \ s))$
by(*induct ets arbitrary:s*) *auto*

lemma *preds-split*:
 $(\text{preds } (ets @ ets') \ s) = (\text{preds } ets \ s \wedge \text{preds } ets' \ (\text{transfers } ets \ s))$
by(*induct ets arbitrary:s*) *auto*

lemma *transfers-id-no-influence*:
 $\text{transfers } [et \leftarrow ets. et \neq \uparrow id] \ s = \text{transfers } ets \ s$
by(*induct ets arbitrary:s,auto*)

lemma *preds-True-no-influence*:
 $\text{preds } [et \leftarrow ets. et \neq (\lambda s. \text{True})_{\checkmark}] \ s = \text{preds } ets \ s$
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 $\Rightarrow$ 'node
fixes *targetnode* :: 'edge $\Rightarrow$ 'node
fixes *kind* :: 'edge $\Rightarrow$ 'state edge-kind
fixes *valid-edge* :: 'edge $\Rightarrow$ bool
fixes *Entry*::'node ('('Entry'-'))
assumes *Entry-target* [*dest*]: $\llbracket \text{valid-edge } a; \text{targetnode } a = (-\text{Entry}-) \rrbracket \Longrightarrow \text{False}$
and *edge-det*:
 $\llbracket \text{valid-edge } a; \text{valid-edge } a'; \text{sourcenode } a = \text{sourcenode } a';$
 $\text{targetnode } a = \text{targetnode } a' \rrbracket \Longrightarrow a = a'$

begin

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

lemma [*simp*]: $\text{valid-edge } a \Longrightarrow \text{valid-node } (\text{sourcenode } a)$

by(*fastforce simp:valid-node-def*)

lemma [*simp*]: *valid-edge a* $\implies$ *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*
 (- $\dashrightarrow^*$ - [*51,0,0*] 80)

where

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

| *Cons-path*:

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

lemma *path-valid-node*:

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

using $\langle n -as\rightarrow^* n' \rangle$

by(*induct rule:path.induct,auto*)

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

by(*fastforce elim:path.cases*)

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

by(*induct rule:path.induct*) *auto*

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

by(*fastforce intro:Cons-path empty-path*)

lemma *path-Entry-target* [*dest*]:

assumes *n* -*as* $\rightarrow^*$ (-*Entry*-)

shows *n* = (-*Entry*-) **and** *as* = []

using $\langle n -as\rightarrow^* (-Entry-) \rangle$

proof(*induct n as n' $\equiv$ (-Entry-) rule:path.induct*)

case (*Cons-path n'' as a n*)

from $\langle \text{targetnode } a = n'' \rangle \langle \text{valid-edge } a \rangle \langle n'' = (-Entry-) \rangle$ **have** *False*

by -(*rule Entry-target,simp-all*)

{ **case** 1

with $\langle \text{False} \rangle$ **show** ?*case* ..

next

case 2

with $\langle \text{False} \rangle$ **show** ?*case* ..

}

qed *simp-all*

lemma *path-Append*: $\llbracket n - as \rightarrow^* n''; n'' - as' \rightarrow^* n' \rrbracket$
 $\implies n - as @ as' \rightarrow^* n'$
by(*induct rule: path.induct, auto intro: Cons-path*)

lemma *path-split*:
assumes $n - as @ a \# as' \rightarrow^* n'$
shows $n - as \rightarrow^* \text{sourcenode } a$ **and** *valid-edge* a **and** $\text{targetnode } a - as' \rightarrow^* n'$
using $\langle n - as @ a \# as' \rightarrow^* n' \rangle$
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* = $\langle \bigwedge n. n - asx @ a \# as' \rightarrow^* n' \implies n - asx \rightarrow^* \text{sourcenode } a \rangle$
note *IH2* = $\langle \bigwedge n. n - asx @ a \# as' \rightarrow^* n' \implies \text{valid-edge } a \rangle$
note *IH3* = $\langle \bigwedge n. n - asx @ a \# as' \rightarrow^* n' \implies \text{targetnode } a - as' \rightarrow^* n' \rangle$
{ case 1
hence $\text{sourcenode } ax = n$ **and** $\text{targetnode } ax - asx @ a \# as' \rightarrow^* n'$ **and** *valid-edge* ax
by(*auto elim:path.cases*)
from *IH1*[*OF* $\langle \text{targetnode } ax - asx @ a \# as' \rightarrow^* n' \rangle$]
have $\text{targetnode } ax - asx \rightarrow^* \text{sourcenode } a$.
with $\langle \text{sourcenode } ax = n \rangle$ $\langle \text{valid-edge } ax \rangle$ **show** ?*case* **by**(*fastforce intro: Cons-path*)
next
case 2 **hence** $\text{targetnode } ax - asx @ a \# as' \rightarrow^* n'$ **by**(*auto elim:path.cases*)
from *IH2*[*OF this*] **show** ?*case* .
next
case 3 **hence** $\text{targetnode } ax - asx @ a \# as' \rightarrow^* n'$ **by**(*auto elim:path.cases*)
from *IH3*[*OF this*] **show** ?*case* .
}
qed

lemma *path-split-Cons*:
assumes $n - as \rightarrow^* n'$ **and** $as \neq []$
obtains $a' as'$ **where** $as = a' \# as'$ **and** $n = \text{sourcenode } a'$
and *valid-edge* a' **and** $\text{targetnode } a' - as' \rightarrow^* n'$
proof –
from $\langle as \neq [] \rangle$ **obtain** $a' as'$ **where** $as = a' \# as'$ **by**(*cases as*) *auto*
with $\langle n - as \rightarrow^* n' \rangle$ **have** $n - [] @ a' \# as' \rightarrow^* n'$ **by** *simp*

hence $n - [] \rightarrow^* \text{sourcenode } a' \text{ and valid-edge } a' \text{ and targetnode } a' - as' \rightarrow^* n'$
 by (rule path-split)+
 from $\langle n - [] \rightarrow^* \text{sourcenode } a' \rangle$ have $n = \text{sourcenode } a'$ by fast
 with $\langle as = a' \# as' \rangle \langle \text{valid-edge } a' \rangle \langle \text{targetnode } a' - as' \rightarrow^* n' \rangle$ that show ?thesis
 by fastforce
 qed

lemma path-split-snoc:

assumes $n - as \rightarrow^* n'$ and $as \neq []$
 obtains $a' as'$ where $as = as' @ [a']$ and $n - as' \rightarrow^* \text{sourcenode } a'$
 and valid-edge a' and $n' = \text{targetnode } a'$
 proof -
 from $\langle as \neq [] \rangle$ obtain $a' as'$ where $as = as' @ [a']$ by (cases as rule: rev-cases)
 auto
 with $\langle n - as \rightarrow^* n' \rangle$ have $n - as' @ a' \# [] \rightarrow^* n'$ by simp
 hence $n - as' \rightarrow^* \text{sourcenode } a' \text{ and valid-edge } a' \text{ and targetnode } a' - [] \rightarrow^* n'$
 by (rule path-split)+
 from $\langle \text{targetnode } a' - [] \rightarrow^* n' \rangle$ have $n' = \text{targetnode } a'$ by fast
 with $\langle as = as' @ [a'] \rangle \langle \text{valid-edge } a' \rangle \langle n - as' \rightarrow^* \text{sourcenode } a' \rangle$ that show ?thesis
 by fastforce
 qed

lemma path-split-second:

assumes $n - as @ a \# as' \rightarrow^* n'$ shows $\text{sourcenode } a - a \# as' \rightarrow^* n'$
 proof -
 from $\langle n - as @ a \# as' \rightarrow^* n' \rangle$ have valid-edge a and $\text{targetnode } a - as' \rightarrow^* n'$
 by (auto intro: path-split)
 thus ?thesis by (fastforce intro: Cons-path)
 qed

lemma path-Entry-Cons:

assumes $(-Entry-) - as \rightarrow^* n'$ and $n' \neq (-Entry-)$
 obtains $n a$ where $\text{sourcenode } a = (-Entry-)$ and $\text{targetnode } a = n$
 and $n - tl as \rightarrow^* n'$ and valid-edge a and $a = hd as$
 proof -
 from $\langle (-Entry-) - as \rightarrow^* n' \rangle \langle n' \neq (-Entry-) \rangle$ have $as \neq []$
 by (cases as, auto elim: path.cases)
 with $\langle (-Entry-) - as \rightarrow^* n' \rangle$ obtain $a' as'$ where $as = a' \# as'$
 and $(-Entry-) = \text{sourcenode } a' \text{ and valid-edge } a' \text{ and targetnode } a' - as' \rightarrow^* n'$
 by (erule path-split-Cons)
 with that show ?thesis by fastforce
 qed

lemma path-det:

```

   $\llbracket n - as \rightarrow^* n'; n - as \rightarrow^* n'' \rrbracket \implies n' = n''$ 
proof(induct as arbitrary:n)
  case Nil thus ?case by(auto elim:path.cases)
next
  case (Cons a' as')
  note IH =  $\langle \bigwedge n. \llbracket n - as' \rightarrow^* n'; n - as' \rightarrow^* n'' \rrbracket \implies n' = n'' \rangle$ 
  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  $\langle$ targetnode a' - as'  $\rightarrow^* n' \rangle$  this] show ?thesis .
qed

```

definition

```

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

```

definition

```

kinds :: 'edge list  $\Rightarrow$  'state edge-kind list
where kinds xs  $\equiv$  map kind xs

```

definition

```

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

```

lemma path-sourcenode:

```

 $\llbracket n - as \rightarrow^* n'; as \neq [] \rrbracket \implies \text{hd } (\text{sourcenodes } as) = n$ 
by(fastforce elim:path-split-Cons simp:sourcenodes-def)

```

lemma path-targetnode:

```

 $\llbracket n - as \rightarrow^* n'; as \neq [] \rrbracket \implies \text{last } (\text{targetnodes } as) = n'$ 
by(fastforce elim:path-split-snoc simp:targetnodes-def)

```

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

```

 $\llbracket n - as \rightarrow^* n'; as \neq [] \rrbracket \implies$ 
  sourcenodes as = n#(butlast (targetnodes as))
proof(induct as arbitrary:n)
  case Nil thus ?case by simp
next
  case (Cons a' as')
  note IH =  $\langle \bigwedge n. \llbracket n - as' \rightarrow^* n'; as' \neq [] \rrbracket$ 
     $\implies \text{sourcenodes } as' = n\#(\text{butlast } (\text{targetnodes } as')) \rangle$ 
  from  $\langle n - a' \# as' \rightarrow^* n' \rangle$  have n = sourcenode a' and targetnode a' - as'  $\rightarrow^* n'$ 

```

```

    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)))⟩ ⟨sourcnode a = n⟩
    show ?case by(simp add:sourcenodes-def)
  next
    case 2
    with ⟨(-Entry-) ∉ set (sourcenodes(tl(a#as)))⟩ ⟨sourcnode 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 sourcnode targetnode kind valid-edge Entry
  for sourcnode :: '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; sourcnode a = (-Exit-)⟧ ⟹ False
  and Entry-Exit-edge: ∃ a. valid-edge a ∧ sourcnode a = (-Entry-) ∧
    targetnode a = (-Exit-) ∧ kind a = (λs. False)✓

```

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-) $\implies$ 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* $\Rightarrow$ bool'
where *inner-node-def*:
inner-node n $\equiv$ valid-node n $\wedge$ n $\neq$ (-Entry-) $\wedge$ n $\neq$ (-Exit-)

lemma *inner-is-valid*:
inner-node n $\implies$ valid-node n
by(*simp add:inner-node-def valid-node-def*)

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

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

lemma [*simp*]: $\llbracket$ *valid-edge a; targetnode a $\neq$ (-Exit-)* $\rrbracket$
 $\implies$ *inner-node (targetnode a)*
by(*simp add:inner-node-def,rule ccontr,simp,erule Entry-target*)

lemma [*simp*]: $\llbracket$ *valid-edge a; sourcenode a $\neq$ (-Entry-)* $\rrbracket$
 $\implies$ *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]:
   $\llbracket \text{valid-node } n; n = (-\text{Entry-}) \implies Q; n = (-\text{Exit-}) \implies Q; \\ \text{inner-node } n \implies Q \rrbracket \implies Q$ 
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  $(-\text{Exit-}) -as \rightarrow^* n'$  shows  $n' = (-\text{Exit-})$  and  $as = []$ 
  using  $\langle (-\text{Exit-}) -as \rightarrow^* n' \rangle$ 
proof(induct n  $\equiv$  (-Exit-) as n' rule:path.induct)
  case (Cons-path n'' as n' a)
  from  $\langle \text{sourcenode } a = (-\text{Exit-}) \rangle \langle \text{valid-edge } a \rangle$  have False
    by  $-(\text{rule Exit-source,simp-all})$ 
  { case 1 with  $\langle \text{False} \rangle$  show ?case ..
  next
    case 2 with  $\langle \text{False} \rangle$  show ?case ..
  }
qed simp-all

```

```

lemma Exit-no-sourcenode[dest]:
  assumes  $\text{isin}:(-\text{Exit-}) \in \text{set } (\text{sourcenodes } as)$  and  $\text{path}:n -as \rightarrow^* n'$ 
  shows False
proof -
  from isin obtain  $ns' ns''$  where  $\text{sourcenodes } as = ns' @ (-\text{Exit-}) \# ns''$ 
  by(auto dest:split-list simp:sourcenodes-def)
  then obtain  $as' as'' a$  where  $as = as' @ a \# as''$ 
  and  $\text{source:sourcenode } a = (-\text{Exit-})$ 
  by(fastforce elim:map-append-append-maps simp:sourcenodes-def)
  with path have  $\text{valid-edge } a$  by(fastforce dest:path-split)
  with source show ?thesis by  $-(\text{erule Exit-source})$ 
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  $\Rightarrow$  'node and targetnode :: 'edge  $\Rightarrow$  'node
and kind :: 'edge  $\Rightarrow$  'state edge-kind and valid-edge :: 'edge  $\Rightarrow$  bool
and Entry :: 'node (('(-Entry'-')) and Exit :: 'node (('(-Exit'-')) +
assumes Entry-path:valid-node  $n \implies \exists as. (-Entry-) -as \rightarrow^* n$ 
and Exit-path:valid-node  $n \implies \exists as. n -as \rightarrow^* (-Exit-)$ 

```

begin

```

definition postdominate :: 'node  $\Rightarrow$  'node  $\Rightarrow$  bool (- postdominates - [51,0])
where postdominate-def:n' postdominates  $n \equiv$ 
  (valid-node  $n \wedge$  valid-node  $n' \wedge$ 
  (( $\forall as. n -as \rightarrow^* (-Exit-) \longrightarrow n' \in \text{set}(\text{sourcenodes } as)$ )))

```

lemma postdominate-implies-path:

```

  assumes n' postdominates  $n$  obtains  $as$  where  $n -as \rightarrow^* n'$ 
proof(atomize-elim)
  from  $\langle n' \text{ postdominates } n \rangle$  have valid-node  $n$ 
    and all: $\forall as. n -as \rightarrow^* (-Exit-) \longrightarrow n' \in \text{set}(\text{sourcenodes } as)$ 
    by(auto simp:postdominate-def)
  from  $\langle \text{valid-node } n \rangle$  obtain  $as$  where  $n -as \rightarrow^* (-Exit-)$  by(auto dest:Exit-path)
  with all have  $n' \in \text{set}(\text{sourcenodes } as)$  by simp
  then obtain  $ns\ ns'$  where  $\text{sourcenodes } as = ns @ n' \# ns'$  by(auto dest:split-list)
  then obtain  $as' a\ as''$  where  $\text{sourcenodes } as' = ns$ 
    and sourcenode  $a = n'$  and  $as = as' @ a \# as''$ 
    by(fastforce elim:map-append-append-maps simp:sourcenodes-def)
  from  $\langle n -as \rightarrow^* (-Exit-) \rangle$   $\langle as = as' @ a \# as'' \rangle$  have  $n -as' \rightarrow^* \text{sourcenode } a$ 
    by(fastforce dest:path-split)
  with  $\langle \text{sourcenode } a = n' \rangle$  show  $\exists as. n -as \rightarrow^* n'$  by blast
qed

```

lemma postdominate-refl:

```

  assumes valid:valid-node  $n$  and notExit: $n \neq (-Exit-)$ 
  shows  $n$  postdominates  $n$ 
using valid
proof(induct rule:valid-node-cases)
  case Entry
  { fix  $as$  assume  $\text{path}:(-Entry-) -as \rightarrow^* (-Exit-)$ 
    hence notempty: $as \neq []$ 
    apply - apply(erule path.cases)
    by (drule sym,simp,drule Exit-noteq-Entry,auto)
  with path have  $\text{hd}(\text{sourcenodes } as) = (-Entry-)$ 
    by(fastforce intro:path-sourcenode)
  with notempty have  $(-Entry-) \in \text{set}(\text{sourcenodes } as)$ 
    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  $\exists as. n -as \rightarrow^* (-Exit-)$ )
case True
{ fix as' assume path': $n -as' \rightarrow^* (-Exit-)$ 
with inner have notempty: $as' \neq []$ 
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'  $\neq []$  by(simp add:sourcenodes-def)
with hd[THEN sym] have  $n \in \text{set (sourcenodes as')}$  by simp }
hence  $\forall as. n -as \rightarrow^* (-Exit-) \rightarrow n \in \text{set (sourcenodes as)}$  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: $n -as \rightarrow^* (-Exit-)$ 
with pd1 have  $n'' \in \text{set (sourcenodes as)}$  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  $n -as'@a\#as'' \rightarrow^* (-Exit-)$  by simp
with source have path': $n'' -a\#as'' \rightarrow^* (-Exit-)$ 
by(fastforce dest:path-split-second)
with pd2 have  $n' \in \text{set (sourcenodes (a\#as''))}$ 
by(auto simp:postdominate-def)
with as have  $n' \in \text{set (sourcenodes as)}$  by(auto simp:sourcenodes-def) }
with valid valid' show ?thesis by(simp add:postdominate-def)
qed

```

```

lemma postdominate-antisym:

```


assumes $pd1:n' \text{ postdominates } n$ and $pd2:n \text{ postdominates } n'$
 shows $n = n'$
proof –
 from $pd1$ have $valid:valid\text{-}node\ n$ and $valid':valid\text{-}node\ n'$
 by($auto\ simp:postdominate\text{-}def$)
 from $valid$ obtain as where $path1:n - as \rightarrow^* (-Exit-)$ by($fastforce\ dest:Exit\text{-}path$)
 from $valid'$ obtain as' where $path2:n' - as' \rightarrow^* (-Exit-)$ by($fastforce\ dest:Exit\text{-}path$)
 from $pd1\ path1$ have $\exists nx \in set(sourcenodes\ as). nx = n'$
 by($simp\ add:postdominate\text{-}def$)
 then obtain $ns\ ns'$ where $sources:sourcenodes\ as = ns@n'\#ns'$
 and $all:\forall nx \in set\ ns'. nx \neq n'$
 by($fastforce\ elim!:rightmost\text{-}element\text{-}property$)
 from $sources$ obtain $asx\ a\ asx'$ where $ns':ns' = sourcenodes\ asx'$
 and $as:as = asx@a\#asx'$ and $source:sourcenode\ a = n'$
 by($fastforce\ elim:map\text{-}append\text{-}append\text{-}maps\ simp:sourcenodes\text{-}def$)
 from $path1\ as$ have $n - asx@a\#asx' \rightarrow^* (-Exit-)$ by $simp$
 with $source$ have $n' - a\#asx' \rightarrow^* (-Exit-)$ by($fastforce\ dest:path\text{-}split\text{-}second$)
 with $pd2$ have $n \in set(sourcenodes\ (a\#asx'))$ by($simp\ add:postdominate\text{-}def$)
 with $source$ have $n = n' \vee n \in set(sourcenodes\ asx')$ by($simp\ add:sourcenodes\text{-}def$)
 thus ?thesis
proof
 assume $n = n'$ thus ?thesis .
next
 assume $n \in set(sourcenodes\ asx')$
 then obtain $nsx'\ nsx''$ where $sourcenodes\ asx' = nsx'@n\#nsx''$
 by($auto\ dest:split\text{-}list$)
 then obtain $asi\ asi'\ a'$ where $asx':asx' = asi@a'\#asi'$
 and $source':sourcenode\ a' = n$
 by($fastforce\ elim:map\text{-}append\text{-}append\text{-}maps\ simp:sourcenodes\text{-}def$)
 with $path1\ as$ have $n - (asx@a\#asi)@a'\#asi' \rightarrow^* (-Exit-)$ by $simp$
 with $source'$ have $n - a'\#asi' \rightarrow^* (-Exit-)$ by($fastforce\ dest:path\text{-}split\text{-}second$)
 with $pd1$ have $n' \in set(sourcenodes\ (a'\#asi'))$ by($auto\ simp:postdominate\text{-}def$)
 with $source'$ have $n' = n \vee n' \in set(sourcenodes\ asi')$
 by($simp\ add:sourcenodes\text{-}def$)
 thus ?thesis
proof
 assume $n' = n$ thus ?thesis by($rule\ sym$)
next
 assume $n' \in set(sourcenodes\ asi')$
 with $asx'\ ns'$ have $n' \in set\ ns'$ by($simp\ add:sourcenodes\text{-}def$)
 with all have $False$ by $blast$
 thus ?thesis by $simp$
 qed
 qed
 qed

lemma *postdominate-path-branch*:

assumes $n - as \rightarrow^* n''$ and $n' \text{ postdominates } n''$ and $\neg n' \text{ postdominates } n$

obtains $a \text{ as}' \text{ as}''$ **where** $as = as'@a\#as''$ **and** *valid-edge* a
and $\neg n'$ *postdominates* (*sourcenode* a) **and** n' *postdominates* (*targetnode* a)
proof(*atomize-elim*)
from *assms*
show $\exists as' a as''. as = as'@a\#as'' \wedge \text{valid-edge } a \wedge$
 $\neg n' \text{ postdominates } (\text{sourcenode } a) \wedge n' \text{ postdominates } (\text{targetnode } a)$
proof(*induct rule:path.induct*)
case (*Cons-path* $n'' as \text{ nx } a \text{ n}$)
note $IH = \langle \llbracket n' \text{ postdominates } \text{nx}; \neg n' \text{ postdominates } n'' \rrbracket$
 $\implies \exists as' a as''. as = as'@a\#as'' \wedge \text{valid-edge } a \wedge$
 $\neg n' \text{ postdominates } \text{sourcenode } a \wedge n' \text{ postdominates } \text{targetnode } a \rangle$
show ?*case*
proof(*cases* $n' \text{ postdominates } n''$)
case *True*
with $\langle \neg n' \text{ postdominates } n \rangle \langle \text{sourcenode } a = n \rangle \langle \text{targetnode } a = n'' \rangle$
 $\langle \text{valid-edge } a \rangle$
show ?*thesis* **by** *blast*
next
case *False*
from $IH[OF \langle n' \text{ postdominates } \text{nx} \rangle \text{ this}]$ **show** ?*thesis*
by *clarsimp*(*rule-tac* $x=a\#as'$ **in** *exI*,*clarsimp*)
qed
qed *simp*
qed

lemma *Exit-no-postdominator*:
 $(\neg \text{Exit}) \text{ postdominates } n \implies \text{False}$
by(*fastforce dest:Exit-path simp:postdominate-def*)

lemma *postdominate-path-targetnode*:
assumes $n' \text{ postdominates } n$ **and** $n - as \rightarrow^* n''$ **and** $n' \notin \text{set}(\text{sourcenodes } as)$
shows $n' \text{ postdominates } n''$
proof –
from $\langle n' \text{ postdominates } n \rangle$ **have** *valid-node* n **and** *valid-node* n'
and $\text{all}:\forall as''. n - as'' \rightarrow^* (\neg \text{Exit}) \longrightarrow n' \in \text{set}(\text{sourcenodes } as'')$
by(*simp-all add:postdominate-def*)
from $\langle n - as \rightarrow^* n'' \rangle$ **have** *valid-node* n'' **by**(*fastforce dest:path-valid-node*)
have $\forall as''. n'' - as'' \rightarrow^* (\neg \text{Exit}) \longrightarrow n' \in \text{set}(\text{sourcenodes } as'')$
proof(*rule ccontr*)
assume $\neg (\forall as''. n'' - as'' \rightarrow^* (\neg \text{Exit}) \longrightarrow n' \in \text{set}(\text{sourcenodes } as''))$
then obtain as'' **where** $n'' - as'' \rightarrow^* (\neg \text{Exit})$
and $n' \notin \text{set}(\text{sourcenodes } as'')$ **by** *blast*
from $\langle n - as \rightarrow^* n'' \rangle \langle n'' - as'' \rightarrow^* (\neg \text{Exit}) \rangle$ **have** $n - as@as'' \rightarrow^* (\neg \text{Exit})$
by(*rule path-Append*)
with $\langle n' \notin \text{set}(\text{sourcenodes } as) \rangle \langle n' \notin \text{set}(\text{sourcenodes } as'') \rangle$
have $n' \notin \text{set}(\text{sourcenodes } (as@as''))$
by(*simp add:sourcenodes-def*)

```

with  $\langle n - as @ as'' \rightarrow^* (-Exit-) \rangle \langle n' \text{ postdominates } n \rangle$  show False
by(simp add:postdominate-def)
qed
with  $\langle \text{valid-node } n' \rangle \langle \text{valid-node } n'' \rangle$  show ?thesis by(simp add:postdominate-def)
qed

```

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

```

assumes  $\neg n \text{ postdominates } (\text{sourcenode } a)$  and valid-node n and valid-edge a
obtains ax where sourcenode a = sourcenode ax and valid-edge ax
and  $\neg n \text{ postdominates } \text{targetnode } ax$ 
proof(atomize-elim)
show  $\exists ax. \text{sourcenode } a = \text{sourcenode } ax \wedge \text{valid-edge } ax \wedge$ 
 $\neg n \text{ postdominates } \text{targetnode } ax$ 
proof -
from assms obtain asx
where sourcenode a - asx  $\rightarrow^* (-Exit-)$ 
and  $n \notin \text{set}(\text{sourcenodes } asx)$  by(auto simp:postdominate-def)
from  $\langle \text{sourcenode } a - asx \rightarrow^* (-Exit-) \rangle \langle \text{valid-edge } a \rangle$ 
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)
apply simp-all
by fastforce
with  $\langle \text{sourcenode } a - asx \rightarrow^* (-Exit-) \rangle$  have sourcenode a - [] @ ax # asx'  $\rightarrow^*$ 
 $(-Exit-)$ 
by simp
hence valid-edge ax and sourcenode a = sourcenode ax
and targetnode ax - asx'  $\rightarrow^* (-Exit-)$ 
by(fastforce dest:path-split) +
with  $\langle n \notin \text{set}(\text{sourcenodes } asx) \rangle$  have  $\neg n \text{ postdominates } \text{targetnode } ax$ 
by(fastforce simp:postdominate-def sourcenodes-def)
with  $\langle \text{sourcenode } a = \text{sourcenode } ax \rangle \langle \text{valid-edge } ax \rangle$  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  $\langle \text{inner-node } n \rangle$  have valid-node n by(rule inner-is-valid)
then obtain as where  $(-Entry-) - as \rightarrow^* n$  by(fastforce dest:Entry-path)
show  $\exists a. \text{valid-edge } a \wedge \text{inner-node } (\text{targetnode } a) \wedge \text{sourcenode } a = (-Entry-)$ 
proof(cases as = [])
case True

```

```

with  $\langle \text{inner-node } n \rangle \langle (-\text{Entry}) -as \rightarrow^* n \rangle$  have False
  by (fastforce simp:inner-node-def)
thus ?thesis by simp
next
case False
with  $\langle (-\text{Entry}) -as \rightarrow^* n \rangle$  obtain  $a' as'$  where  $as = a' \# as'$ 
  and  $(-\text{Entry}) = \text{sourcenode } a'$  and  $\text{valid-edge } a'$ 
  and  $\text{targetnode } a' -as' \rightarrow^* n$ 
  by  $-(\text{erule path-split-Cons})$ 
from  $\langle \text{valid-edge } a' \rangle$  have  $\text{valid-node } (\text{targetnode } a')$  by simp
thus ?thesis
proof (cases targetnode a' rule:valid-node-cases)
  case Entry
  from  $\langle \text{valid-edge } a' \rangle$  this have False by (rule Entry-target)
  thus ?thesis by simp
next
case Exit
with  $\langle \text{targetnode } a' -as' \rightarrow^* n \rangle \langle \text{inner-node } n \rangle$ 
have False by simp (drule path-Exit-source, auto simp:inner-node-def)
thus ?thesis by simp
next
case inner
with  $\langle \text{valid-edge } a' \rangle \langle (-\text{Entry}) = \text{sourcenode } a' \rangle$  show ?thesis by simp blast
qed
qed
qed

```

```

lemma inner-node-Exit-edge:
  assumes inner-node n
  obtains  $a$  where  $\text{valid-edge } a$  and  $\text{inner-node } (\text{sourcenode } a)$ 
  and  $\text{targetnode } a = (-\text{Exit})$ 
proof (atomize-elim)
  from  $\langle \text{inner-node } n \rangle$  have  $\text{valid-node } n$  by (rule inner-is-valid)
  then obtain  $as$  where  $n -as \rightarrow^* (-\text{Exit})$  by (fastforce dest:Exit-path)
  show  $\exists a. \text{valid-edge } a \wedge \text{inner-node } (\text{sourcenode } a) \wedge \text{targetnode } a = (-\text{Exit})$ 
  proof (cases as = [])
    case True
    with  $\langle \text{inner-node } n \rangle \langle n -as \rightarrow^* (-\text{Exit}) \rangle$  have False by fastforce
    thus ?thesis by simp
  next
  case False
  with  $\langle n -as \rightarrow^* (-\text{Exit}) \rangle$  obtain  $a' as'$  where  $as = as' @ [a']$ 
    and  $n -as' \rightarrow^* \text{sourcenode } a'$  and  $\text{valid-edge } a'$ 
    and  $(-\text{Exit}) = \text{targetnode } a'$  by  $-(\text{erule path-split-snoc})$ 
  from  $\langle \text{valid-edge } a' \rangle$  have  $\text{valid-node } (\text{sourcenode } a')$  by simp
  thus ?thesis
  proof (cases sourcenode a' rule:valid-node-cases)
    case Entry

```

```

with  $\langle n - as' \rightarrow^* sourcenode\ a' \rangle \langle inner-node\ n \rangle$ 
have False by simp (drule path-Entry-target, auto simp: inner-node-def)
thus ?thesis by simp
next
case Exit
from  $\langle valid-edge\ a' \rangle$  this have False by (rule Exit-source)
thus ?thesis by simp
next
case inner
with  $\langle valid-edge\ a' \rangle \langle (-Exit-) = targetnode\ a' \rangle$  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 ::  $'edge \Rightarrow 'node$  and targetnode ::  $'edge \Rightarrow 'node$ 
  and kind ::  $'edge \Rightarrow 'state\ edge-kind$  and valid-edge ::  $'edge \Rightarrow bool$ 
  and Entry ::  $'node \Rightarrow ('(-Entry'-))$  and Exit ::  $'node \Rightarrow ('(-Exit'-))$  +
  assumes successor-set-finite: valid-node  $n \implies$ 
    finite  $\{n'. \exists a'. valid-edge\ a' \wedge sourcenode\ a' = n \wedge targetnode\ a' = n'\}$ 

```

begin

```

definition strong-postdominate ::  $'node \Rightarrow 'node \Rightarrow bool$ 
  (- strongly-postdominates - [51, 0])
where strong-postdominate-def: n' strongly-postdominates n  $\equiv$ 
  (n' postdominates n  $\wedge$ 
   ( $\exists k \geq 1. \forall as\ nx. n - as \rightarrow^* nx \wedge$ 
     $length\ as \geq k \longrightarrow n' \in set(sourcenodes\ as)))$ 

```

lemma *strong-postdominate-prop-smaller-path:*

```

assumes all:  $\forall as\ nx. n - as \rightarrow^* nx \wedge length\ as \geq k \longrightarrow n' \in set(sourcenodes\ as)$ 
and  $n - as \rightarrow^* n''$  and  $length\ as \geq k$ 
obtains  $as'\ as''$  where  $n - as' \rightarrow^* n'$  and  $length\ as' < k$  and  $n' - as'' \rightarrow^* n''$ 
and  $as = as' @ as''$ 

```

proof (*atomize-elim*)

```

show  $\exists as'\ as''. n - as' \rightarrow^* n' \wedge length\ as' < k \wedge n' - as'' \rightarrow^* n'' \wedge as = as' @ as''$ 

```

proof (*rule ccontr*)

```

assume  $\neg (\exists as'\ as''. n - as' \rightarrow^* n' \wedge length\ as' < k \wedge n' - as'' \rightarrow^* n'' \wedge$ 
   $as = as' @ as'')$ 

```

hence $\text{all}': \forall as' as''. n - as' \rightarrow^* n' \wedge n' - as'' \rightarrow^* n'' \wedge as = as' @ as''$
 $\rightarrow \text{length } as' \geq k$ **by** *fastforce*
 from $\text{all } \langle n - as \rightarrow^* n' \rangle \langle \text{length } as \geq k \rangle$ **have** $\exists nx \in \text{set}(\text{sourcenodes } as). nx = n'$
 by *fastforce*
 then **obtain** $ns \ ns'$ **where** $\text{sourcenodes } as = ns @ n' \# ns'$
 and $\forall nx \in \text{set } ns. nx \neq n'$
 by (*fastforce elim!:split-list-first-propE*)
 then **obtain** $asx \ a \ asx'$ **where** $[simp]: ns = \text{sourcenodes } asx$
 and $[simp]: as = asx @ a \# asx'$ **and** $\text{sourcenode } a = n'$
 by (*fastforce elim:map-append-append-maps simp:sourcenodes-def*)
 from $\langle n - as \rightarrow^* n' \rangle$ **have** $n - asx @ a \# asx' \rightarrow^* n''$ **by** *simp*
 with $\langle \text{sourcenode } a = n' \rangle$ **have** $n - asx \rightarrow^* n'$ **and** *valid-edge* a
 and *targetnode* $a - asx' \rightarrow^* n''$ **by** (*fastforce dest:path-split*) +
 with $\langle \text{sourcenode } a = n' \rangle$ **have** $n' - a \# asx' \rightarrow^* n''$ **by** (*fastforce intro:Cons-path*)
 with $\langle n - asx \rightarrow^* n' \rangle$ **all'** **have** $\text{length } asx \geq k$ **by** *simp*
 with $\langle n - asx \rightarrow^* n' \rangle$ **all** **have** $n' \in \text{set}(\text{sourcenodes } asx)$ **by** *fastforce*
 with $\langle \forall nx \in \text{set } ns. nx \neq n' \rangle$ **show** *False* **by** *fastforce*
 qed
 qed

lemma *strong-postdominate-refl*:
 assumes *valid-node* n **and** $n \neq (-\text{Exit})$
 shows n *strongly-postdominates* n
proof –
 from *assms* **have** n *postdominates* n **by** (*rule postdominate-refl*)
 { **fix** $as \ nx$ **assume** $n - as \rightarrow^* nx$ **and** $\text{length } as \geq 1$
 then obtain $a' \ as'$ **where** $[simp]: as = a' \# as'$ **by** (*cases as*) *auto*
 with $\langle n - as \rightarrow^* nx \rangle$ **have** $n - [] @ a' \# as' \rightarrow^* nx$ **by** *simp*
 hence $n = \text{sourcenode } a'$ **by** (*fastforce dest:path-split*)
 hence $n \in \text{set}(\text{sourcenodes } as)$ **by** (*simp add:sourcenodes-def*) }
 hence $\forall as \ nx. n - as \rightarrow^* nx \wedge \text{length } as \geq 1 \rightarrow n \in \text{set}(\text{sourcenodes } as)$
 by *auto*
 hence $\exists k \geq 1. \forall as \ nx. n - as \rightarrow^* nx \wedge \text{length } as \geq k \rightarrow n \in \text{set}(\text{sourcenodes } as)$
 by *blast*
 with $\langle n \text{ postdominates } n \rangle$ **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 $\langle n'' \text{ strongly-postdominates } n \rangle$ **have** n'' *postdominates* n
 and *paths1*: $\exists k \geq 1. \forall as \ nx. n - as \rightarrow^* nx \wedge \text{length } as \geq k$
 $\rightarrow n'' \in \text{set}(\text{sourcenodes } as)$

by(*auto simp only:strong-postdominate-def*)
from *paths1* **obtain** *k1*
where *all1*: $\forall as\ nx. n -as\rightarrow^* nx \wedge \text{length } as \geq k1 \longrightarrow n'' \in \text{set}(\text{sourcenodes } as)$
and $k1 \geq 1$ **by** *blast*
from $\langle n' \text{ strongly-postdominates } n'' \rangle$ **have** $n' \text{ postdominates } n''$
and *paths2*: $\exists k \geq 1. \forall as\ nx. n'' -as\rightarrow^* nx \wedge \text{length } as \geq k \longrightarrow n' \in \text{set}(\text{sourcenodes } as)$
by(*auto simp only:strong-postdominate-def*)
from *paths2* **obtain** *k2*
where *all2*: $\forall as\ nx. n'' -as\rightarrow^* nx \wedge \text{length } as \geq k2 \longrightarrow n' \in \text{set}(\text{sourcenodes } as)$
and $k2 \geq 1$ **by** *blast*
from $\langle n'' \text{ postdominates } n \rangle \langle n' \text{ postdominates } n'' \rangle$
have $n' \text{ postdominates } n$ **by**(*rule postdominate-trans*)
{ fix *as nx* **assume** $n -as\rightarrow^* nx$ **and** $\text{length } as \geq k1 + k2$
hence $\text{length } as \geq k1$ **by** *fastforce*
with $\langle n -as\rightarrow^* nx \rangle$ **all1** **obtain** *asx asx'* **where** $n -asx\rightarrow^* n''$
and $\text{length } asx < k1$ **and** $n'' -asx'\rightarrow^* nx$
and [*simp*]: $as = asx@asx'$ **by** $-(\text{erule strong-postdominate-prop-smaller-path})$
with $\langle \text{length } as \geq k1 + k2 \rangle$ **have** $\text{length } asx' \geq k2$ **by** *fastforce*
with $\langle n'' -asx'\rightarrow^* nx \rangle$ **all2** **have** $n' \in \text{set}(\text{sourcenodes } asx')$ **by** *fastforce*
hence $n' \in \text{set}(\text{sourcenodes } as)$ **by**(*simp add:sourcenodes-def*) **}**
with $\langle k1 \geq 1 \rangle \langle k2 \geq 1 \rangle$ **have** $\exists k \geq 1. \forall as\ nx. n -as\rightarrow^* nx \wedge \text{length } as \geq k \longrightarrow n' \in \text{set}(\text{sourcenodes } as)$
by(*rule-tac x=k1 + k2 in exI,auto*)
with $\langle n' \text{ postdominates } n \rangle$ **show** *?thesis* **by**(*simp add:strong-postdominate-def*)
qed

lemma *strong-postdominate-antisym*:

$\llbracket n' \text{ strongly-postdominates } n; n \text{ strongly-postdominates } n' \rrbracket \implies n = n'$
by(*fastforce intro:postdominate-antisym simp:strong-postdominate-def*)

lemma *strong-postdominate-path-branch*:

assumes $n -as\rightarrow^* n''$ **and** $n' \text{ strongly-postdominates } n''$
and $\neg n' \text{ strongly-postdominates } n$
obtains $a \text{ as}' \text{ as}''$ **where** $as = as'@a\#as''$ **and** *valid-edge* *a*
and $\neg n' \text{ strongly-postdominates } (\text{sourcenode } a)$
and $n' \text{ strongly-postdominates } (\text{targetnode } a)$
proof(*atomize-elim*)
from *assms*
show $\exists as' a as''. as = as'@a\#as'' \wedge \text{valid-edge } a \wedge$
 $\neg n' \text{ strongly-postdominates } (\text{sourcenode } a) \wedge$
 $n' \text{ strongly-postdominates } (\text{targetnode } a)$
proof(*induct rule:path.induct*)
case (*Cons-path* $n'' \text{ as } nx \text{ a } n$)
note *IH* = $\llbracket n' \text{ strongly-postdominates } nx; \neg n' \text{ strongly-postdominates } n'' \rrbracket$

```

 $\implies \exists as' a as''. as = as'@a\#as'' \wedge \text{valid-edge } a \wedge$ 
 $\neg n' \text{ strongly-postdominates sourcenode } a \wedge$ 
 $n' \text{ strongly-postdominates targetnode } a$ 
show ?case
proof(cases  $n' \text{ strongly-postdominates } n''$ )
  case True
    with  $\langle \neg n' \text{ strongly-postdominates } n \rangle \langle \text{sourcenode } a = n \rangle \langle \text{targetnode } a = n'' \rangle$ 
       $\langle \text{valid-edge } a \rangle$ 
    show ?thesis by blast
  next
    case False
    from IH[OF  $\langle n' \text{ strongly-postdominates } nx \rangle$  this] show ?thesis
    by clarsimp(rule-tac  $x=a\#as'$  in exI,clarsimp)
  qed
qed simp
qed

```

lemma Exit-no-strong-postdominator:
 $\llbracket (-\text{Exit-}) \text{ strongly-postdominates } n; n -as \rightarrow^* (-\text{Exit-}) \rrbracket \implies \text{False}$
by(fastforce intro:Exit-no-postdominator path-valid-node simp:strong-postdominate-def)

lemma strong-postdominate-path-targetnode:
assumes $n' \text{ strongly-postdominates } n$ **and** $n -as \rightarrow^* n''$
and $n' \notin \text{set}(\text{sourcenodes } as)$
shows $n' \text{ strongly-postdominates } n''$
proof -
from $\langle n' \text{ strongly-postdominates } n \rangle$ **have** $n' \text{ postdominates } n$
and $\exists k \geq 1. \forall as \ nx. n -as \rightarrow^* nx \wedge \text{length } as \geq k$
 $\longrightarrow n' \in \text{set}(\text{sourcenodes } as)$
by(auto simp only:strong-postdominate-def)
then obtain k **where** $k \geq 1$
and $\text{paths}:\forall as \ nx. n -as \rightarrow^* nx \wedge \text{length } as \geq k$
 $\longrightarrow n' \in \text{set}(\text{sourcenodes } as)$ **by** auto
from $\langle n' \text{ postdominates } n \rangle \langle n -as \rightarrow^* n'' \rangle \langle n' \notin \text{set}(\text{sourcenodes } as) \rangle$
have $n' \text{ postdominates } n''$
by(rule postdominate-path-targetnode)
{ fix $as' \ nx$ **assume** $n'' -as' \rightarrow^* nx$ **and** $\text{length } as' \geq k$
with $\langle n -as \rightarrow^* n'' \rangle$ **have** $n -as@as' \rightarrow^* nx$ **and** $\text{length } (as@as') \geq k$
by(auto intro:path-Append)
with paths **have** $n' \in \text{set}(\text{sourcenodes } (as@as'))$ **by** fastforce
with $\langle n' \notin \text{set}(\text{sourcenodes } as) \rangle$ **have** $n' \in \text{set}(\text{sourcenodes } as')$
by(fastforce simp:sourcenodes-def) **}**
with $\langle k \geq 1 \rangle$ **have** $\exists k \geq 1. \forall as' \ nx. n'' -as' \rightarrow^* nx \wedge \text{length } as' \geq k$
 $\longrightarrow n' \in \text{set}(\text{sourcenodes } as')$ **by** auto
with $\langle n' \text{ postdominates } n'' \rangle$ **show** ?thesis **by**(simp add:strong-postdominate-def)
qed

lemma *not-strong-postdominate-successor-set*:

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

```

    by  $-(erule\ path.cases, auto)$ 
  with  $allImp$  have  $targetnode\ ax \in N$  by  $fastforce$ 
  with  $all$  have  $n\ strongly-postdominates\ (targetnode\ ax)$ 
    by  $auto$ 
  then obtain  $k'$  where  $k':k' = (SOME\ i.\ i \geq 1 \wedge$ 
     $(\forall as\ nx.\ targetnode\ ax -as \rightarrow^* nx \wedge length\ as \geq i$ 
       $\longrightarrow n \in set(sourcenodes\ as)))$  by  $simp$ 
  with  $\langle n\ strongly-postdominates\ (targetnode\ ax) \rangle$ 
  have  $k' \geq 1 \wedge (\forall as\ nx.\ targetnode\ ax -as \rightarrow^* nx \wedge length\ as \geq k'$ 
     $\longrightarrow n \in set(sourcenodes\ as))$ 
    by  $(auto\ elim!:someI-ex\ simp:strong-postdominate-def)$ 
  hence  $k' \geq 1$ 
    and  $spdAll:\forall as\ nx.\ targetnode\ ax -as \rightarrow^* nx \wedge length\ as \geq k'$ 
       $\longrightarrow n \in set(sourcenodes\ as)$ 
    by  $simp-all$ 
  from  $\langle targetnode\ ax \in N \rangle\ k'$  have  $k' \in ?N'$  by  $blast$ 
  moreover
  from  $this\ \langle targetnode\ ax \in N \rangle$  have  $?N' \neq \{\}$  by  $auto$ 
  ultimately have  $k' \leq Max\ ?N'$  using  $\langle finite\ ?N' \rangle$  by  $(fastforce\ intro:Max-ge)$ 
  hence  $k' \leq k$  by  $simp$ 
  with  $\langle targetnode\ ax -as' \rightarrow^* nx \rangle\ \langle length\ as \geq k + 1 \rangle\ spdAll$ 
  have  $n \in set(sourcenodes\ as')$ 
    by  $fastforce$ 
  with  $\langle n \notin set(sourcenodes\ as) \rangle$  show  $False$  by  $(simp\ add:sourcenodes-def)$ 
qed
qed
qed

```

lemma *not-strong-postdominate-predecessor-successor*:

```

  assumes  $\neg n\ strongly-postdominates\ (sourcenode\ a)$ 
  and  $valid-node\ n$  and  $valid-edge\ a$ 
  obtains  $a'$  where  $valid-edge\ a'$  and  $sourcenode\ a' = sourcenode\ a$ 
  and  $\neg n\ strongly-postdominates\ (targetnode\ a')$ 
proof( $atomize-elim$ )
  show  $\exists a'.\ valid-edge\ a' \wedge sourcenode\ a' = sourcenode\ a \wedge$ 
     $\neg n\ strongly-postdominates\ (targetnode\ a')$ 
  proof( $rule\ ccontr$ )
    assume  $\neg (\exists a'.\ valid-edge\ a' \wedge sourcenode\ a' = sourcenode\ a \wedge$ 
       $\neg n\ strongly-postdominates\ targetnode\ a')$ 
    hence  $all:\forall a'.\ valid-edge\ a' \wedge sourcenode\ a' = sourcenode\ a \longrightarrow$ 
       $n\ strongly-postdominates\ (targetnode\ a')$  by  $auto$ 
    let  $?N = \{n'.\ \exists a'.\ sourcenode\ a' = sourcenode\ a \wedge valid-edge\ a' \wedge$ 
       $targetnode\ a' = n'\}$ 
    from  $all$  have  $\forall nx \in ?N.\ \exists a'.\ valid-edge\ a' \wedge sourcenode\ a' = sourcenode\ a$ 
       $\wedge$ 
       $targetnode\ a' = nx \wedge n\ strongly-postdominates\ nx$ 
    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  $\Rightarrow$  'node and targetnode :: 'edge  $\Rightarrow$  'node
  and kind :: 'edge  $\Rightarrow$  'state edge-kind and valid-edge :: 'edge  $\Rightarrow$  bool
  and Entry :: 'node ('(-Entry-')) +
  fixes Def::'node  $\Rightarrow$  'var set
  fixes Use::'node  $\Rightarrow$  'var set
  fixes state-val::'state  $\Rightarrow$  'var  $\Rightarrow$  'val
  assumes Entry-empty:Def (-Entry-) = {}  $\wedge$  Use (-Entry-) = {}
  and CFG-edge-no-Def-equal:
     $\llbracket \text{valid-edge } a; V \notin \text{Def } (\text{sourcenode } a); \text{pred } (\text{kind } a) \text{ } s \rrbracket$ 
     $\implies \text{state-val } (\text{transfer } (\text{kind } a) \text{ } s) \text{ } V = \text{state-val } s \text{ } V$ 
  and CFG-edge-transfer-uses-only-Use:
     $\llbracket \text{valid-edge } a; \forall V \in \text{Use } (\text{sourcenode } a). \text{state-val } s \text{ } V = \text{state-val } s' \text{ } V;$ 
     $\text{pred } (\text{kind } a) \text{ } s; \text{pred } (\text{kind } a) \text{ } s' \rrbracket$ 
     $\implies \forall V \in \text{Def } (\text{sourcenode } a). \text{state-val } (\text{transfer } (\text{kind } a) \text{ } s) \text{ } V =$ 
     $\text{state-val } (\text{transfer } (\text{kind } a) \text{ } s') \text{ } V$ 
  and CFG-edge-Uses-pred-equal:
     $\llbracket \text{valid-edge } a; \text{pred } (\text{kind } a) \text{ } s;$ 
     $\forall V \in \text{Use } (\text{sourcenode } a). \text{state-val } s \text{ } V = \text{state-val } s' \text{ } V \rrbracket$ 
     $\implies \text{pred } (\text{kind } a) \text{ } s'$ 
  and deterministic: $\llbracket \text{valid-edge } a; \text{valid-edge } a'; \text{sourcenode } a = \text{sourcenode } a';$ 
     $\text{targetnode } a \neq \text{targetnode } a' \rrbracket$ 
     $\implies \exists Q \text{ } Q'. \text{kind } a = (Q)_{\surd} \wedge \text{kind } a' = (Q')_{\surd} \wedge$ 
     $(\forall s. (Q \text{ } s \longrightarrow \neg Q' \text{ } s) \wedge (Q' \text{ } s \longrightarrow \neg Q \text{ } s))$ 

begin

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

```

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

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

$\llbracket n - as \rightarrow^* n'; \forall n \in set \ (sourcenodes \ as). \ V \notin Def \ n; preds \ (kinds \ as) \ s \rrbracket$
 $\implies state-val \ (transfers \ (kinds \ as) \ s) \ V = state-val \ s \ V$
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 $IH = \langle \bigwedge s. \llbracket \forall n \in set \ (sourcenodes \ as). \ V \notin Def \ n; preds \ (kinds \ as) \ s \rrbracket \implies$
 $state-val \ (transfers \ (kinds \ as) \ s) \ V = state-val \ s \ V \rangle$
 from $\langle preds \ (kinds \ (a \# as)) \ s \rangle$ **have** $pred \ (kind \ a) \ s$
 and $preds \ (kinds \ as) \ (transfer \ (kind \ a) \ s)$ **by**(*simp-all add:kinds-def*)
 from $\langle \forall n \in set \ (sourcenodes \ (a \# as)). \ V \notin Def \ n \rangle$
 have $noDef: V \notin Def \ (sourcenode \ a)$
 and $all: \forall n \in set \ (sourcenodes \ as). \ V \notin Def \ n$
 by(*auto simp:sourcenodes-def*)
 from $\langle valid-edge \ a \rangle$ $noDef \ \langle pred \ (kind \ a) \ s \rangle$
 have $state-val \ (transfer \ (kind \ a) \ s) \ V = state-val \ s \ V$
 by(*rule CFG-edge-no-Def-equal*)
 with $IH[OF \ all \ \langle preds \ (kinds \ as) \ (transfer \ (kind \ a) \ s) \rangle]$ **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* $\Rightarrow$ *'node* **and** *targetnode* :: *'edge* $\Rightarrow$ *'node*
and *kind* :: *'edge* $\Rightarrow$ *'state edge-kind* **and** *valid-edge* :: *'edge* $\Rightarrow$ *bool*
and *Entry* :: *'node* $(('(-Entry'-))$ **and** *Def* :: *'node* $\Rightarrow$ *'var set*
and *Use* :: *'node* $\Rightarrow$ *'var set* **and** *state-val* :: *'state* $\Rightarrow$ *'var* $\Rightarrow$ *'val*
and *Exit* :: *'node* $(('(-Exit'-))$ +
assumes *Exit-empty:Def* $(-Exit-) = \{\}$ $\wedge$ *Use* $(-Exit-) = \{\}$

begin

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

lemma *Exit-Def-empty* [dest!]: $V \in Def \ (-Exit-) \implies False$
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 $\Rightarrow$ 'node **and** *targetnode* :: 'edge $\Rightarrow$ 'node
and *kind* :: 'edge $\Rightarrow$ 'state *edge-kind* **and** *valid-edge* :: 'edge $\Rightarrow$ bool
and *Entry* :: 'node ('('Entry'-')) +
fixes *sem*::'com $\Rightarrow$ 'state $\Rightarrow$ 'com $\Rightarrow$ 'state $\Rightarrow$ bool
 $((1 \langle -,/- \rangle) \Rightarrow / (1 \langle -,/- \rangle)) [0,0,0,0] 81)$
fixes *identifies*::'node $\Rightarrow$ 'com $\Rightarrow$ bool ($- \triangleq - [51,0] 80$)
assumes *fundamental-property*:
 $\llbracket n \triangleq c; \langle c,s \rangle \Rightarrow \langle c',s' \rangle \rrbracket \implies$
 $\exists n' as. n -as \rightarrow^* n' \wedge \text{transfers } (kinds \ as) \ s = s' \wedge \text{preds } (kinds \ as) \ s \wedge$
 $n' \triangleq c'$

end

1.6 Dynamic data dependence

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

context *CFG-wf* **begin**

definition *dyn-data-dependence* ::
'node $\Rightarrow$ 'var $\Rightarrow$ 'node $\Rightarrow$ 'edge list $\Rightarrow$ bool ($- \text{influences } - \text{ in } - \text{ via } - [51,0,0]$)
where $n \text{ influences } V \text{ in } n' \text{ via } as \equiv$
 $((V \in Def \ n) \wedge (V \in Use \ n') \wedge (n -as \rightarrow^* n') \wedge$
 $(\exists a' as'. (as = a' \# as') \wedge (\forall n'' \in set \ (sourcenodes \ as'). V \notin Def \ n''))))$

lemma *dyn-influence-Cons-source*:
 $n \text{ influences } V \text{ in } n' \text{ via } a \# as \implies \text{sourcenode } a = n$
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 $n \notin \text{set } (\text{sourcenodes } as)$
proof(rule *ccontr*)
assume $\neg n \notin \text{set } (\text{sourcenodes } as)$
hence $n \in \text{set } (\text{sourcenodes } as)$ **by** *simp*
from $\langle n \text{ influences } V \text{ in } n' \text{ via } a\#as \rangle$ **have** $\forall n'' \in \text{set } (\text{sourcenodes } as). V \notin \text{Def } n''$
and $V \in \text{Def } n$ **by**(*simp-all add:dyn-data-dependence-def*)
from $\langle \forall n'' \in \text{set } (\text{sourcenodes } as). V \notin \text{Def } n'' \rangle$
 $\langle n \in \text{set } (\text{sourcenodes } as) \rangle$ **have** $V \notin \text{Def } n$ **by** *simp*
with $\langle V \in \text{Def } n \rangle$ **show** *False* **by** *simp*
qed

lemma *dyn-influence-only-first-edge*:
assumes n influences V in n' via $a\#as$ **and** $\text{preds } (\text{kinds } (a\#as)) s$
shows $\text{state-val } (\text{transfers } (\text{kinds } (a\#as)) s) V =$
 $\text{state-val } (\text{transfer } (\text{kind } a) s) V$
proof –
from $\langle \text{preds } (\text{kinds } (a\#as)) s \rangle$ **have** $\text{preds } (\text{kinds } as) (\text{transfer } (\text{kind } a) s)$
by(*simp add:kinds-def*)
from $\langle n \text{ influences } V \text{ in } n' \text{ via } a\#as \rangle$ **have** $n - a\#as \rightarrow^* n'$
and $\forall n'' \in \text{set } (\text{sourcenodes } as). V \notin \text{Def } n''$
by(*simp-all add:dyn-data-dependence-def*)
from $\langle n - a\#as \rightarrow^* n' \rangle$ **have** $n = \text{sourcenode } a$ **and** $\text{targetnode } a - as \rightarrow^* n'$
by(*auto elim:path-split-Cons*)
from $\langle n \text{ influences } V \text{ in } n' \text{ via } a\#as \rangle$ $\langle n = \text{sourcenode } a \rangle$
have $\text{sourcenode } a \notin \text{set } (\text{sourcenodes } as)$
by(*fastforce intro!:dyn-influence-source-notin-tl-edges*)
{ fix n'' **assume** $n'' \in \text{set } (\text{sourcenodes } as)$
with $\langle \text{sourcenode } a \notin \text{set } (\text{sourcenodes } as) \rangle$ $\langle n = \text{sourcenode } a \rangle$
have $n'' \neq n$ **by**(*fastforce simp:sourcenodes-def*)
with $\langle \forall n'' \in \text{set } (\text{sourcenodes } as). V \notin \text{Def } n'' \rangle$ $\langle n'' \in \text{set } (\text{sourcenodes } as) \rangle$
have $V \notin \text{Def } n''$ **by**(*auto simp:sourcenodes-def*) **}**
hence $\forall n'' \in \text{set } (\text{sourcenodes } as). V \notin \text{Def } n''$ **by** *simp*
with $\langle \text{targetnode } a - as \rightarrow^* n' \rangle$ $\langle \text{preds } (\text{kinds } as) (\text{transfer } (\text{kind } a) s) \rangle$
have $\text{state-val } (\text{transfers } (\text{kinds } as) (\text{transfer } (\text{kind } a) s)) V =$
 $\text{state-val } (\text{transfer } (\text{kind } a) s) V$
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 $\Rightarrow$ 'node $\Rightarrow$ 'edge list $\Rightarrow$ bool
 (- controls_s - via - [51,0,0])
where *dyn-standard-control-dependence-def*: *n* controls_s *n'* via *as* $\equiv$
 $(\exists a \ a' \ as'. (as = a\#as') \wedge (n' \notin \text{set}(\text{sourcenodes } as)) \wedge (n - as \rightarrow^* n') \wedge$
 $(n' \text{ postdominates } (\text{targetnode } a)) \wedge$
 $(\text{valid-edge } a') \wedge (\text{sourcenode } a' = n) \wedge$
 $(\neg n' \text{ postdominates } (\text{targetnode } a'))))$

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* $\rightarrow^*$ (-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* - $\llcorner$ *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

lemma *dyn-standard-control-dependence-def-variant*:

n controls_s *n'* via *as* = $((n - as \rightarrow^* n') \wedge (n \neq n') \wedge$
 $(\neg n' \text{ postdominates } n) \wedge (n' \notin \text{set}(\text{sourcenodes } as)) \wedge$
 $(\forall n'' \in \text{set}(\text{targetnodes } as). n' \text{ postdominates } n''))$

proof

assume $(n - as \rightarrow^* n') \wedge (n \neq n') \wedge (\neg n' \text{ postdominates } n) \wedge$
 $(n' \notin \text{set}(\text{sourcenodes } as)) \wedge$
 $(\forall n'' \in \text{set}(\text{targetnodes } as). n' \text{ postdominates } n'')$
hence *path*:*n* - *as* $\rightarrow^*$ *n'* **and** *noteq*:*n* $\neq$ *n'*
and *not-pd*: $\neg n' \text{ postdominates } n$
and *notin*: $n' \notin \text{set}(\text{sourcenodes } as)$
and *all*: $\forall n'' \in \text{set}(\text{targetnodes } as). n' \text{ postdominates } n''$
by *auto*
have *notExit*:*n* $\neq$ (-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  $n$  controlss  $n'$  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  $path2:targetnode$   $ax - asx \rightarrow^* n'$ 
by(auto intro:path-split-Cons)
show ?thesis
proof(cases  $\exists as'. n - as' \rightarrow^* (-Exit-)$ )
  case True
  with not-pd valid valid' obtain  $as'$  where  $path':n - as' \rightarrow^* (-Exit-)$ 
  and not-isin: $n' \notin$  set(sourcenodes  $as'$ )
  by(auto simp:postdominate-def)
  have  $as' \neq []$ 
  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  $n - [] @ a' \# as'' \rightarrow^* (-Exit-)$  by simp
  hence source': $n =$  sourcenode  $a'$ 
  and valid-edge:valid-edge  $a'$ 
  and  $path2':targetnode$   $a' - as'' \rightarrow^* (-Exit-)$ 
  by(fastforce dest:path-split)+
  from  $path2'$  not-isin  $as'$  valid'
  have  $\neg n'$  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  $n$  controlss  $n'$  via  $as$ 
then obtain  $a$   $nx$   $a'$   $nx'$   $as'$  where notin: $n' \notin$  set(sourcenodes  $as$ )
and  $path:n - as \rightarrow^* n'$  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 ≠ 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:∀ as. n -as→* (-Exit-) → n' ∈ set (sourcenodes as)
    by(auto simp:postdominate-def)
  from not-pd valid' valid-target' obtain as'
    where targetnode a' -as'→* (-Exit-)
    by(auto simp:postdominate-def)
  with source' valid-edge have path':n -a'#as'→* (-Exit-)
    by(fastforce intro:Cons-path)
  with all have n' ∈ set (sourcenodes (a'#as')) by blast
  with source' have n' = n ∨ n' ∈ set (sourcenodes as')
    by(fastforce simp:sourcenodes-def)
  hence False
proof
  assume n' = n
  with noteq show ?thesis by simp
next
  assume isin:n' ∈ set (sourcenodes as')
  from path' have path2:targetnode a' -as'→* (-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''→* (-Exit-)
    and notin:n' ∉ set (sourcenodes as'')
    by(auto simp:postdominate-def)
  from valid-edge path'' have sourcenode a' -a'#as''→* (-Exit-)
    by(fastforce intro:Cons-path)
  with all source' have n' ∈ set (sourcenodes ([a']@as'')) by simp
  with source' have n' = n ∨ n' ∈ set (sourcenodes as'')
    by(auto simp:sourcenodes-def)

```

```

thus ?thesis
proof
  assume  $n' = n$ 
  with noteq show ?thesis by simp
next
  assume  $n' \in \text{set } (\text{sourcenodes } as'')$ 
  with notin show ?thesis by simp
qed
qed
qed }
hence not-pd':  $\neg n'$  postdominates  $n$  by blast
{ fix  $n''$  assume  $n'' \in \text{set } (\text{targetnodes } as)$ 
  with  $as$  source have  $n'' = \text{targetnode } a \vee n'' \in \text{set } (\text{targetnodes } as')$ 
  by(auto simp:targetnodes-def)
  hence  $n'$  postdominates  $n''$ 
proof
  assume  $n'' = \text{targetnode } a$ 
  with pd show ?thesis by simp
next
  assume  $isin:n'' \in \text{set } (\text{targetnodes } as')$ 
  hence  $\exists ni \in \text{set } (\text{targetnodes } as'). ni = n''$  by simp
  then obtain  $ns\ ns'$  where  $\text{targets:targetnodes } as' = ns@n''\#ns'$ 
    and  $\text{all-noteq}:\forall ni \in \text{set } ns'. ni \neq n''$ 
    by(fastforce elim!:rightmost-element-property)
  from targets obtain  $xs\ ax\ ys$  where  $ys:ns' = \text{targetnodes } ys$ 
    and  $as':as' = xs@ax\#ys$  and  $\text{target'}:\text{targetnode } ax = n''$ 
    by(fastforce elim:map-append-append-maps simp:targetnodes-def)
  from  $\text{all-noteq } ys$  have  $\text{notin-target}:n'' \notin \text{set } (\text{targetnodes } ys)$ 
    by auto
  from path  $as$  have  $n - []@a\#as' \rightarrow^* n'$  by simp
  hence  $\text{targetnode } a - as' \rightarrow^* n'$ 
    by(fastforce dest:path-split)
  with  $isin$  have  $\text{path'}:\text{targetnode } a - as' \rightarrow^* n'$ 
    by(fastforce split:split-if-asm simp:targetnodes-def)
  with  $as'$  target'' have  $\text{path1}:\text{targetnode } a - xs \rightarrow^* \text{sourcenode } ax$ 
    and  $\text{valid-edge'}:\text{valid-edge } ax$ 
    and  $\text{path2}:n'' - ys \rightarrow^* n'$ 
    by(auto intro:path-split)
  from  $\text{valid-edge'}$  have  $\text{sourcenode } ax - [ax] \rightarrow^* \text{targetnode } ax$  by(rule path-edge)
  with  $\text{path1 } \text{target''}$  have  $\text{path-n'}:\text{targetnode } a - xs@[ax] \rightarrow^* n''$ 
    by(fastforce intro:path-Append)
  from notin  $as\ as'$  have  $\text{notin'}:n' \notin \text{set } (\text{sourcenodes } (xs@[ax]))$ 
    by(simp add:sourcenodes-def)
  show ?thesis
proof(rule ccontr)
  assume  $\neg n'$  postdominates  $n''$ 
  with  $\text{valid' } \text{target'' } \text{valid-edge'}$  obtain  $asx'$ 
    where  $\text{Exit-path}:n'' - asx' \rightarrow^* (-\text{Exit-})$ 
    and  $\text{notin'':}n' \notin \text{set } (\text{sourcenodes } asx')$  by(auto simp:postdominate-def)

```

```

    from path-n'' Exit-path
    have Exit-path':targetnode a  $\rightarrow$  (xs@[ax])@asx' $\rightarrow$ * (-Exit-)
    by(fastforce intro:path-Append)
    from notin' notin'' have n'  $\notin$  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  $\rightarrow$  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-)  $\rightarrow$  as@a#as' $\rightarrow$ * n
  and Exit-path:n  $\rightarrow$  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: $\neg$  n postdominates (targetnode a')
  and last: $\forall$  ax ax'. ax  $\in$  set as'  $\wedge$  sourcenode ax = sourcenode ax'  $\wedge$ 
    valid-edge ax'  $\rightarrow$  n postdominates targetnode ax'
  shows n' controlss n via a#as'
proof -
  from path source have path-n'n:n'  $\rightarrow$  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': $\neg$  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  $\rightarrow$  as' $\rightarrow$ * n
      by(fastforce dest:path-split)
      from Exit-path path-nxn have  $\exists$  as. targetnode a  $\rightarrow$  as $\rightarrow$ * (-Exit-)
      by(fastforce dest:path-Append)
    end
  end

```

```

with not-pd' inner-node valid-edge obtain asx
  where path-Exit:targetnode a -asx→* (-Exit-)
  and notin:n ∉ set (sourcenodes asx)
  by(auto simp:postdominate-def inner-is-valid)
show ?thesis
proof(cases ∃ 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  $n' - (a\#as'1)@a1\#as'2 \rightarrow^* n$  by simp
  hence  $n' - a\#as'1 \rightarrow^*$  sourcenode  $a1$  and valid-edge1:valid-edge  $a1$ 
    by(fastforce elim:path-split)+
  hence path1:targetnode  $a - as'1 \rightarrow^*$  sourcenode  $a1$ 
    by(fastforce intro:path-split-Cons)
  from path-Exit  $asx$ 
  have targetnode  $a - as'1 \rightarrow^*$  sourcenode  $a2$ 
    and valid-edge2:valid-edge  $a2$ 
    and path2:targetnode  $a2 - asx'1 \rightarrow^*$  (-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  $n \notin \text{set}(\text{sourcenodes } asx'1)$ 
    by(simp add:sourcenodes-def)
  with path2 have not-pd'2: $\neg n$  postdominates targetnode  $a2$ 
    by(cases  $asx'1 = []$ ,auto simp:postdominate-def)
  from  $as'$  have  $a1 \in \text{set } as'$  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  $n'$  controlss  $n$  via as
proof(atomize-elim)
  from inner-node obtain  $as'$  where pathExit: $n - as' \rightarrow^*$  (-Exit-)
    by(fastforce dest:inner-is-valid Exit-path)
  from inner-node obtain as where pathEntry:(-Entry-)  $-as \rightarrow^* n$ 
    by(fastforce dest:inner-is-valid Entry-path)
  with inner-node have notEmpty: $as \neq []$ 
    by(auto elim:path.cases simp:inner-node-def)
  have  $\exists a asx. (-Entry-) -a\#asx \rightarrow^* n \wedge n \notin \text{set}(\text{sourcenodes } (a\#asx))$ 
  proof(cases  $n \in \text{set}(\text{sourcenodes } as)$ )
    case True
    hence  $\exists n'' \in \text{set}(\text{sourcenodes } as). n = n''$  by simp
    then obtain  $ns' ns''$  where nodes:sourcenodes  $as = ns'@n\#ns''$ 
      and notin: $\forall n'' \in \text{set } ns'. n \neq n''$ 
      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  $(-Entry-) - xs @ a' \# ys \rightarrow^* n$  by simp
hence path2: $(-Entry-) - xs \rightarrow^*$  sourcenode a'
  by (fastforce dest: path-split)
show ?thesis
proof (cases xs = [])
  case True
    with path2 have  $(-Entry-) = \text{sourcenode } a'$  by (auto elim: path.cases)
    with pathEntry source notEmpty have  $(-Entry-) - as \rightarrow^* (-Entry-) \wedge as \neq []$ 
      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': $(-Entry-) - a \# asx \rightarrow^* n$ 
  and notin: n  $\notin \text{set } (\text{sourcenodes } (a \# asx))$  by blast
show  $\exists n' as. n' \text{ controls}_s n \text{ via } as$ 
proof (cases  $\forall a' a''. a' \in \text{set } asx \wedge \text{sourcenode } a' = \text{sourcenode } a'' \wedge$ 
   $\text{valid-edge } a'' \rightarrow n \text{ postdominates targetnode } a''$ )
  case True
    from inner-node have not-pd:  $\neg n \text{ postdominates } (-Exit-)$ 
      by (fastforce intro: empty-path simp: postdominate-def sourcenodes-def)
    from pathEntry' have path': $(-Entry-) - [] @ a \# asx \rightarrow^* n$  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 controlss n via  $a \# asx$ 
      by  $\neg(\text{erule } \text{which-node-dyn-standard-control-dependence-source}, \text{auto})$ 
    thus ?thesis by blast
  next
    case False
    hence  $\exists a' \in \text{set } asx. \exists a''. \text{sourcenode } a' = \text{sourcenode } a'' \wedge \text{valid-edge } a'' \wedge$ 
       $\neg n \text{ postdominates targetnode } a''$ 
      by fastforce
    then obtain ax asx' asx'' where  $asx = asx' @ ax \# asx'' \wedge$ 
       $(\exists a''. \text{sourcenode } ax = \text{sourcenode } a'' \wedge \text{valid-edge } a'' \wedge$ 
       $\neg n \text{ postdominates targetnode } a'') \wedge$ 
       $(\forall z \in \text{set } asx''. \neg (\exists a''. \text{sourcenode } z = \text{sourcenode } a'' \wedge \text{valid-edge } a'' \wedge$ 
       $\neg n \text{ postdominates targetnode } a''))$ 
      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 ∉ set (sourcenodes (ax#asx''))
by(auto simp:sourcenodes-def)
from as' pathEntry' have (-Entry-) -(a#asx')@ax#asx''→* n by simp
with inner-node not-pd notin' eq last pathExit valid-edge
have sourcenode ax controlss 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 ⇒ 'node ⇒ 'edge list ⇒ bool
(- weakly controls - via - [51,0,0])
where dyn-weak-control-dependence-def:n weakly controls n' via as ≡
(∃ a a' as'. (as = a#as') ∧ (n' ∉ set(sourcenodes as)) ∧ (n -as→* n') ∧
(n' strongly-postdominates (targetnode a)) ∧
(valid-edge a') ∧ (sourcenode a' = n) ∧
(¬ 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→* (-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'→* (-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  $\Rightarrow$  'node and targetnode :: 'edge  $\Rightarrow$  'node
  and kind :: 'edge  $\Rightarrow$  'state edge-kind and valid-edge :: 'edge  $\Rightarrow$  bool
  and Entry :: 'node  $\Rightarrow$  ('(-Entry'-)) and Def :: 'node  $\Rightarrow$  'var set
  and Use :: 'node  $\Rightarrow$  'var set and state-val :: 'state  $\Rightarrow$  'var  $\Rightarrow$  'val
  and Exit :: 'node  $\Rightarrow$  ('(-Exit'-)) +
  fixes dyn-control-dependence :: 'node  $\Rightarrow$  'node  $\Rightarrow$  'edge list  $\Rightarrow$  bool
  (- controls - via - [51,0,0])
  assumes Exit-not-dyn-control-dependent: n controls n' via as  $\implies$  n'  $\neq$  (-Exit-)
  assumes dyn-control-dependence-path:
    n controls n' via as  $\implies$  n -as $\rightarrow^*$  n'  $\wedge$  as  $\neq$  []
```

begin

```
inductive cdep-edge :: 'node  $\Rightarrow$  'edge list  $\Rightarrow$  'node  $\Rightarrow$  bool
  (-  $\dashrightarrow_{cd}$  - [51,0,0] 80)
  and ddep-edge :: 'node  $\Rightarrow$  'var  $\Rightarrow$  'edge list  $\Rightarrow$  'node  $\Rightarrow$  bool
  (- -'{-}' $\dashrightarrow_{dd}$  - [51,0,0,0] 80)
  and DynPDG-edge :: 'node  $\Rightarrow$  'var option  $\Rightarrow$  'edge list  $\Rightarrow$  'node  $\Rightarrow$  bool
```

where

— Syntax

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

— Rules

$| DynPDG-cdep-edge:$
 $n\ controls\ n'\ via\ as \implies n - as \rightarrow_{cd} n'$

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

inductive $DynPDG-path :: 'node \Rightarrow 'edge\ list \Rightarrow 'node \Rightarrow bool$
 $(- \dashrightarrow_{d*} - [51, 0, 0] 80)$

where $DynPDG-path-Nil:$

$valid-node\ n \implies n - [] \rightarrow_{d*} n$

$| DynPDG-path-Append-cdep:$
 $\llbracket n - as \rightarrow_{d*} n''; n'' - as' \rightarrow_{cd} n' \rrbracket \implies n - as @ as' \rightarrow_{d*} n'$

$| DynPDG-path-Append-ddep:$
 $\llbracket n - as \rightarrow_{d*} n''; n'' - \{V\} as' \rightarrow_{dd} n' \rrbracket \implies n - as @ as' \rightarrow_{d*} n'$

lemma $DynPDG-empty-path-eq-nodes: n - [] \rightarrow_{d*} n' \implies n = n'$

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 $DynPDG-path-cdep: n - as \rightarrow_{cd} n' \implies n - as \rightarrow_{d*} n'$

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 $DynPDG-path-ddep: n - \{V\} as \rightarrow_{dd} n' \implies n - as \rightarrow_{d*} n'$

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 $DynPDG-path-Append:$

$\llbracket n'' - as' \rightarrow_{d*} n'; n - as \rightarrow_{d*} n' \rrbracket \implies n - as @ as' \rightarrow_{d*} n'$

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*: $\llbracket n -as \rightarrow_d^* n'; n' = (-Exit-) \rrbracket \implies n = (-Exit-)$
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*:
 $\llbracket n -as \rightarrow_d^* n'; \neg inner-node\ n' \rrbracket \implies n = n'$
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 $\langle n'' -as' \rightarrow_{cd} n' \rangle \neg inner-node\ n'$ **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 $\langle n'' -\{V\} as' \rightarrow_{dd} n' \rangle \neg inner-node\ n'$ **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 $\langle n -as \rightarrow_{cd} n' \rangle$
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 $\langle n -\{V\} as \rightarrow_{dd} n' \rangle$
by(*auto elim:DynPDG-edge.cases simp:dyn-data-dependence-def*)

lemma *DynPDG-path-CFG-path*:
 $n -as \rightarrow_d^* n' \implies n -as \rightarrow^* n'$
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 $\langle n'' - as' \rightarrow_{cd} n' \rangle$ **have** $n'' - as' \rightarrow_* n'$
by(rule *DynPDG-cdep-edge-CFG-path*(1))
with $\langle n - as \rightarrow_* n' \rangle$ **show** ?case **by**(rule *path-Append*)
next
case (*DynPDG-path-Append-ddep* n as n'' V as' n')
from $\langle n'' - \{V\} as' \rightarrow_{dd} n' \rangle$ **have** $n'' - as' \rightarrow_* n'$
by(rule *DynPDG-ddep-edge-CFG-path*(1))
with $\langle n - as \rightarrow_* n' \rangle$ **show** ?case **by**(rule *path-Append*)
qed

lemma *DynPDG-path-split*:

$n - as \rightarrow_d^* n' \implies$
 $(as = [] \wedge n = n') \vee$
 $(\exists n'' asx asx'. (n - asx \rightarrow_{cd} n'') \wedge (n'' - asx' \rightarrow_d^* n') \wedge$
 $(as = asx @ asx')) \vee$
 $(\exists n'' V asx asx'. (n - \{V\} asx \rightarrow_{dd} n'') \wedge (n'' - asx' \rightarrow_d^* n') \wedge$
 $(as = asx @ asx'))$
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 $IH = \langle as = [] \wedge n = n'' \vee$
 $(\exists nx asx asx'. n - asx \rightarrow_{cd} nx \wedge nx - asx' \rightarrow_d^* n'' \wedge as = asx @ asx') \vee$
 $(\exists nx V asx asx'. n - \{V\} asx \rightarrow_{dd} nx \wedge nx - asx' \rightarrow_d^* n'' \wedge as = asx @ asx') \rangle$
from IH **show** ?case
proof
assume $as = [] \wedge n = n''$
with $\langle n'' - as' \rightarrow_{cd} n' \rangle$ **have** *valid-node* n'
by(*fastforce intro:path-valid-node*(2) *DynPDG-path-CFG-path*
DynPDG-path-cdep)
with $\langle as = [] \wedge n = n'' \rangle$ $\langle n'' - as' \rightarrow_{cd} n' \rangle$
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 $\langle n'' - as' \rightarrow_{cd} n' \rangle$ **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 $\langle n - asx \rightarrow_{cd} nx \rangle$ $\langle as = asx @ asx' \rangle$
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 $\langle n'' - as' \rightarrow_{cd} n' \rangle$ 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 $\langle n'' - \{V\} as' \rightarrow_{dd} n' \rangle$ **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 $\langle n - \{V'\} asx \rightarrow_{dd} nx \rangle$ $\langle as = asx @ asx' \rangle$
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
qed

lemma DynPDG-path-rev-cases [consumes 1,
case-names DynPDG-path-Nil DynPDG-path-cdep-Append DynPDG-path-ddep-Append]:
 $\llbracket n - as \rightarrow_{d*} n'; \llbracket as = []; n = n' \rrbracket \implies Q;$
 $\bigwedge n'' \ asx \ asx'. \llbracket n - asx \rightarrow_{cd} n''; n'' - asx' \rightarrow_{d*} n';$
 $as = asx @ asx \rrbracket \implies Q;$
 $\bigwedge V \ n'' \ asx \ asx'. \llbracket n - \{V\} asx \rightarrow_{dd} n''; n'' - asx' \rightarrow_{d*} n';$
 $as = asx @ asx \rrbracket \implies Q]$
 $\implies Q$
by (blast dest: DynPDG-path-split)

lemma DynPDG-ddep-edge-no-shorter-ddep-edge:
assumes $ddep: n - \{V\} as \rightarrow_{dd} n'$
shows $\forall as' \ a \ as''. \ tl \ as = as' @ a \# as'' \longrightarrow \neg \text{sourcenode } a - \{V\} a \# as'' \rightarrow_{dd} n'$
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 $\text{notin}: n \notin \text{set}(\text{sourcenodes } asx)$
by (auto dest: dyn-influence-source-notin-tl-edges simp: dyn-data-dependence-def)
from influence as
have imp: $\forall nx \in \text{set}(\text{sourcenodes } asx). \ V \notin \text{Def } nx$
by (auto simp: dyn-data-dependence-def)
{ fix $as' \ a' \ as''$
assume eq: $tl \ as = as' @ a' \# as''$
and ddep': $\text{sourcenode } a' - \{V\} a' \# as'' \rightarrow_{dd} n'$
from eq as **notin** **have** noteq: $\text{sourcenode } a' \neq n$ **by** (auto simp: sourcenodes-def)
from ddep' **have** $V \in \text{Def}(\text{sourcenode } a')$

```

    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→* n' and Uses:V ∈ Use n' and preds:preds (kinds as) s
  and no-dep:∀ as' a as''. as = as'@a#as'' → ¬ sourcenode a -{V}a#as''→dd
  n'
  shows state-val (transfers (kinds as) s) V = state-val s V
proof -
  { fix n''
    assume inset:n'' ∈ set (sourcenodes as) and Defs:V ∈ Def n''
    hence ∃ nx ∈ set (sourcenodes as). V ∈ Def nx by auto
    then obtain nx ns' ns'' where nodes:sourcenodes as = ns'@nx#ns''
      and Defs':V ∈ Def nx and notDef:∀ nx' ∈ set ns''. V ∉ 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''→* n'
      by(fastforce dest:path-split-second)
    from notDef as'' source
    have ∀ n'' ∈ set (sourcenodes as''). V ∉ 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 ∉ set (sourcenodes as'') by(rule dyn-influence-source-notin-tl-edges)
    with influence source
    have ∃ asx a'. sourcenode a' -{V}a'#asx→dd n' ∧ sourcenode a' = nx ∧
      (∃ 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 ∀ n ∈ set (sourcenodes as). V ∉ 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→dd n'; preds (kinds (a#as)) s⟧ ⟹
    state-val (transfers (kinds (a#as)) s) V = state-val (transfer (kind a) s) V
  apply -
  apply(erule DynPDG-edge.cases)
  apply auto
  apply(frul 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** $\langle preds\ (kinds\ as)\ s \rangle$
and $\langle preds\ (kinds\ as)\ s' \rangle$ **and** $state-val\ s\ V = state-val\ s'\ V$
and $state-val\ (transfers\ (kinds\ as)\ s)\ V \neq$
 $state-val\ (transfers\ (kinds\ as)\ s')\ V$
obtains $as'\ a\ as''$ **where** $as = as' @ a \# as''$
and $sourcenode\ a - \{V\} a \# as'' \rightarrow_{dd}\ n'$
and $state-val\ (transfers\ (kinds\ as)\ s)\ V =$
 $state-val\ (transfers\ (kinds\ (as' @ [a]))\ s)\ V$
and $state-val\ (transfers\ (kinds\ as)\ s')\ V =$
 $state-val\ (transfers\ (kinds\ (as' @ [a]))\ s')\ V$
proof(*atomize-elim*)
show $\exists as'\ a\ as''. as = as' @ a \# as'' \wedge$
 $sourcenode\ a - \{V\} a \# as'' \rightarrow_{dd}\ n' \wedge$
 $state-val\ (transfers\ (kinds\ as)\ s)\ V =$
 $state-val\ (transfers\ (kinds\ (as' @ [a]))\ s)\ V \wedge$
 $state-val\ (transfers\ (kinds\ as)\ s')\ V =$
 $state-val\ (transfers\ (kinds\ (as' @ [a]))\ s')\ V$
proof(*cases* $\forall as'\ a\ as''. as = as' @ a \# as'' \longrightarrow$
 $\neg sourcenode\ a - \{V\} a \# as'' \rightarrow_{dd}\ n'$)
case *True*
with $\langle n - as \rightarrow^* n' \rangle$ $\langle V \in Use\ n' \rangle$ $\langle preds\ (kinds\ as)\ s \rangle$ $\langle preds\ (kinds\ as)\ s' \rangle$
have $state-val\ (transfers\ (kinds\ as)\ s)\ V = state-val\ s\ V$
and $state-val\ (transfers\ (kinds\ as)\ s')\ V = state-val\ s'\ V$
by(*auto intro:no-ddep-same-state*)
with $\langle state-val\ s\ V = state-val\ s'\ V \rangle$
 $\langle state-val\ (transfers\ (kinds\ as)\ s)\ V \neq state-val\ (transfers\ (kinds\ as)\ s')\ V \rangle$
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 $\langle preds\ (kinds\ as)\ s \rangle$ **have** $\langle preds\ (kinds\ (a \# as''))\ (transfers\ (kinds\ as')\ s) \rangle$
by(*simp add:kinds-def preds-split*)
with $\langle sourcenode\ a - \{V\} a \# as'' \rightarrow_{dd}\ n' \rangle$ **have** *all*:
 $state-val\ (transfers\ (kinds\ (a \# as''))\ (transfers\ (kinds\ as')\ s))\ V =$
 $state-val\ (transfer\ (kind\ a)\ (transfers\ (kinds\ as')\ s))\ V$
by(*auto dest!:DynPDG-ddep-edge-only-first-edge*)
from $\langle preds\ (kinds\ as)\ s' \rangle$
have $\langle preds\ (kinds\ (a \# as''))\ (transfers\ (kinds\ as')\ s') \rangle$
by(*simp add:kinds-def preds-split*)
with $\langle sourcenode\ a - \{V\} a \# as'' \rightarrow_{dd}\ n' \rangle$ **have** *all'*:
 $state-val\ (transfers\ (kinds\ (a \# as''))\ (transfers\ (kinds\ as')\ s'))\ V =$
 $state-val\ (transfer\ (kind\ a)\ (transfers\ (kinds\ as')\ s'))\ V$
by(*auto dest!:DynPDG-ddep-edge-only-first-edge*)


```

hence eq:  $\wedge s. \text{transfers } (\text{kinds } as) s =$ 
  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 controlss n' via as
  show n' ≠ (-Exit-)
  proof
    assume n' = (-Exit-)
    with ⟨n controlss n' via as⟩ show False
    by(fastforce intro:Exit-not-dyn-standard-control-dependent)
  qed
qed
next
  fix n n' as assume n controlss 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 $\Rightarrow$ 'node **and** targetnode :: 'edge $\Rightarrow$ 'node
and kind :: 'edge $\Rightarrow$ 'state edge-kind **and** valid-edge :: 'edge $\Rightarrow$ bool
and Entry :: 'node ('('Exit-')) **and** Def :: 'node $\Rightarrow$ 'var set
and Use :: 'node $\Rightarrow$ 'var set **and** state-val :: 'state $\Rightarrow$ 'var $\Rightarrow$ '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 $\Rightarrow$ 'node $\Rightarrow$ 'edge list $\Rightarrow$ bool

where *empty-control-dependence n n' as* $\equiv$ *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  $n - as \rightarrow^* n' \wedge as \neq []$  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*:

$V \in Use\ n' \implies (V, [], []) \in dependent-live-vars\ n'$

| *dep-vars-Cons-cdep*:

$\llbracket V \in Use\ (sourcenode\ a); sourcenode\ a - a\#as' \rightarrow_{cd}\ n''; n'' - as'' \rightarrow_d^* n \rrbracket$
 $\implies (V, [], a\#as'@as'') \in dependent-live-vars\ n'$

| *dep-vars-Cons-ddep*:

$\llbracket (V, as', as) \in dependent-live-vars\ n'; V' \in Use\ (sourcenode\ a);$
 $n' = last(targetnodes\ (a\#as));$
 $sourcenode\ a - \{V\}a\#as' \rightarrow_{dd}\ last(targetnodes\ (a\#as')) \rrbracket$
 $\implies (V', [], a\#as) \in dependent-live-vars\ n'$

| *dep-vars-Cons-keep*:

$\llbracket (V, as', as) \in dependent-live-vars\ n'; n' = last(targetnodes\ (a\#as));$
 $\neg sourcenode\ a - \{V\}a\#as' \rightarrow_{dd}\ last(targetnodes\ (a\#as')) \rrbracket$
 $\implies (V, a\#as', a\#as) \in dependent-live-vars\ n'$

lemma *dependent-live-vars-fst-prefix-snd*:

$(V, as', as) \in dependent-live-vars\ n' \implies \exists as''. as'@as'' = as$

by (induct rule: *dependent-live-vars.induct*, simp-all)

lemma *dependent-live-vars-Exit-empty* [*dest*]:

$(V, as', as) \in dependent-live-vars\ (-Exit-) \implies False$

proof (induct rule: *dependent-live-vars.induct*)

case (*dep-vars-Cons-cdep* *V a as' n'' as''*)

from $\langle n'' - as'' \rightarrow_d^* (-Exit-) \rangle$ **have** $n'' = (-Exit-)$

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

with $\langle sourcenode\ a - a\#as' \rightarrow_{cd}\ n'' \rangle$ **have** $sourcenode\ a - a\#as' \rightarrow_d^* (-Exit-)$

by(fastforce intro:DynPDG-path-cdep)
 hence sourcenode $a = (-Exit-)$ by(fastforce intro:DynPDG-path-Exit)
 with $\langle V \in Use \text{ (sourcenode } a) \rangle$ show False by simp(erule Exit-Use-empty)
 qed auto

lemma dependent-live-vars-lastnode:
 $\llbracket (V, as', as) \in dependent-live-vars \ n'; as \neq [] \rrbracket$
 $\implies n' = last(targetnodes \ as)$
proof(induct rule:dependent-live-vars.induct)
 case (dep-vars-Cons-cdep $V \ a \ as' \ n'' \ as''$)
 from $\langle sourcenode \ a - a \# as' \rightarrow_{cd} n'' \rangle$ have sourcenode $a - a \# as' \rightarrow^* n''$
 by(rule DynPDG-cdep-edge-CFG-path(1))
 from $\langle n'' - as'' \rightarrow_d^* n' \rangle$ have $n'' - as'' \rightarrow^* n'$ by(rule DynPDG-path-CFG-path)
 show ?case
proof(cases $as'' = []$)
 case True
 with $\langle n'' - as'' \rightarrow^* n' \rangle$ have $n'' = n'$ by (auto elim: DynPDG.dependent-live-vars.cases)
 with $\langle sourcenode \ a - a \# as' \rightarrow^* n'' \rangle$ True
 show ?thesis by(fastforce intro:path-targetnode[THEN sym])
 next
 case False
 with $\langle n'' - as'' \rightarrow^* n' \rangle$ have $n' = last(targetnodes \ as'')$
 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:
 $\llbracket (V, as', as) \in dependent-live-vars \ n'; n - as \rightarrow^* n' \rrbracket$
 $\implies \exists \ nx \ as''. as = as' @ as'' \wedge n - as' \rightarrow^* nx \wedge nx - as'' \rightarrow_d^* n' \wedge V \in Use \ nx$
 $\wedge$
 $(\forall \ n'' \in set \ (sourcenodes \ as'). V \notin Def \ n'')$

proof(induct arbitrary:n rule:dependent-live-vars.induct)
 case (dep-vars-Use V)
 from $\langle n - [] \rightarrow^* n' \rangle$ have valid-node n' by(rule path-valid-node(2))
 hence $n' - [] \rightarrow_d^* n'$ by(rule DynPDG-path-Nil)
 with $\langle V \in Use \ n' \rangle \langle n - [] \rightarrow^* n' \rangle$ show ?case
 by(auto simp:sourcenodes-def)
 next
 case (dep-vars-Cons-cdep $V \ a \ as' \ n'' \ as'' \ n$)
 from $\langle n - a \# as' @ as'' \rightarrow^* n' \rangle$ have sourcenode $a = n$
 by(auto elim:path.cases)
 from $\langle sourcenode \ a - a \# as' \rightarrow_{cd} n'' \rangle$ 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 $\langle sourcenode \ a - a \# as' \rightarrow_{cd} n'' \rangle$ have sourcenode $a - a \# as' \rightarrow_d^* n''$

by(rule DynPDG-path-cdep)
 with $\langle n'' - as'' \rightarrow_d^* n' \rangle$ have sourcenode $a - (a \# as') @ as'' \rightarrow_d^* n'$
 by(rule DynPDG-path-Append)
 with $\langle sourcenode a - [] \rightarrow^* sourcenode a \rangle \langle V \in Use (sourcenode a) \rangle \langle sourcenode$
 $a = n \rangle$
 show ?case by(auto simp:sourcenodes-def)
 next
 case(dep-vars-Cons-ddep $V as' as V' a n$)
 note ddep = $\langle sourcenode a - \{V\} a \# as' \rightarrow_{dd} last (targetnodes (a \# as')) \rangle$
 note IH = $\langle \bigwedge n. n - as \rightarrow^* n' \Rightarrow \exists nx as''. as = as' @ as'' \wedge n - as' \rightarrow^* nx \wedge nx - as'' \rightarrow_d^* n' \wedge$
 $V \in Use nx \wedge (\forall n'' \in set (sourcenodes as')). V \notin Def n'' \rangle$
 from $\langle n - a \# as \rightarrow^* n' \rangle$ have $n - [] @ a \# as \rightarrow^* n'$ by simp
 hence $n = sourcenode a$ and targetnode $a - as \rightarrow^* n'$ and valid-edge a
 by(fastforce dest:path-split)+
 hence $n - [] \rightarrow^* n$
 by(fastforce intro:empty-path simp:valid-node-def)
 from IH[OF $\langle targetnode a - as \rightarrow^* n' \rangle$]
 have $\exists nx as''. as = as' @ as'' \wedge targetnode a - as' \rightarrow^* nx \wedge nx - as'' \rightarrow_d^* n' \wedge$
 $V \in Use nx \wedge (\forall n'' \in set (sourcenodes as')). V \notin Def n''$.
 then obtain $nx'' as''$ where targetnode $a - as' \rightarrow^* nx''$
 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 $\langle targetnode a - as' \rightarrow^* nx'' \rangle$ have $nx'' = targetnode a$
 by(auto elim:path.cases)
 with $\langle nx'' - as'' \rightarrow_d^* n' \rangle$ 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 $\langle targetnode a - as' \rightarrow^* nx'' \rangle$ have $last (targetnodes as') = nx''$
 by(fastforce intro:path-targetnode)
 with $\langle last (targetnodes (a \# as')) = last (targetnodes as') \rangle \langle nx'' - as'' \rightarrow_d^* n' \rangle$
 show ?thesis by simp
 qed
 with ddep $\langle as = as' @ as'' \rangle$ have sourcenode $a - a \# as \rightarrow_d^* n'$
 by(fastforce dest:DynPDG-path-ddep DynPDG-path-Append)
 with $\langle V' \in Use (sourcenode a) \rangle \langle n = sourcenode a \rangle \langle n - [] \rightarrow^* n \rangle$
 show ?case by(auto simp:sourcenodes-def)
 next
 case (dep-vars-Cons-keep $V as' as a n$)
 note no-ddep = $\langle \neg sourcenode a - \{V\} a \# as' \rightarrow_{dd} last (targetnodes (a \# as')) \rangle$
 note IH = $\langle \bigwedge n. n - as \rightarrow^* n' \Rightarrow \exists nx as''. (as = as' @ as'') \wedge (n - as' \rightarrow^* nx) \wedge (nx - as'' \rightarrow_d^* n') \wedge$
 $V \in Use nx \wedge (\forall n'' \in set (sourcenodes as')). V \notin Def n'' \rangle$
 from $\langle n - a \# as \rightarrow^* n' \rangle$ have $n = sourcenode a$ and valid-edge a
 and targetnode $a - as \rightarrow^* n'$ by(auto elim:path-split-Cons)

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

lemma *dependent-live-vars-dependent-edge*:

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

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 $\llbracket \forall as' \ a \ as''. as = as'@a \# as'' \longrightarrow \neg sourcenode\ a - \{V\} a \# as'' \rightarrow_{dd} n';$
 $as \neq [] \longrightarrow n' = last(targetnodes\ as) \rrbracket$
 $\implies (V, as, as) \in dependent-live-vars\ n'$
proof(*induct as*)
case *Nil*
from $\langle V \in Use\ n' \rangle$ **show** *?case* **by**(*rule dep-vars-Use*)
next
case (*Cons ax asx*)
note $lastnode = \langle ax \# asx \neq [] \longrightarrow n' = last(targetnodes\ (ax \# asx)) \rangle$
note $IH = \llbracket \forall as' \ a \ as''. asx = as'@a \# as'' \longrightarrow$
 $\neg sourcenode\ a - \{V\} a \# as'' \rightarrow_{dd} n';$
 $asx \neq [] \longrightarrow n' = last(targetnodes\ asx) \rrbracket$
 $\implies (V, asx, asx) \in dependent-live-vars\ n'$
from $\langle \forall as' \ a \ as''. ax \# asx = as'@a \# as'' \longrightarrow \neg sourcenode\ a - \{V\} a \# as'' \rightarrow_{dd}$
 $n' \rangle$
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 $\langle V \in Use\ n' \rangle$ **have** $(V, [], []) \in dependent-live-vars\ n'$ **by**(*rule dep-vars-Use*)
with $\langle \neg sourcenode\ ax - \{V\} ax \# asx \rightarrow_{dd} n' \rangle$ *True lastnode*
have $(V, [ax], [ax]) \in dependent-live-vars\ n'$
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** $(V, asx, asx) \in dependent-live-vars\ n'$.
with $\langle \neg sourcenode\ ax - \{V\} ax \# asx \rightarrow_{dd} n' \rangle$ *lastnode*
show *?thesis* **by**(*fastforce intro:dep-vars-Cons-keep*)
qed
qed

lemma *dependent-live-vars-same-pathsD*:
 $\llbracket (V, as, as) \in dependent-live-vars\ n'; as \neq [] \longrightarrow n' = last(targetnodes\ as) \rrbracket$
 $\implies V \in Use\ n' \wedge (\forall as' \ a \ as''. as = as'@a \# as'' \longrightarrow$

$\neg \text{sourcenode } a - \{V\} a \# as'' \rightarrow_{dd} n'$

proof(*induct as*)

case *Nil*

have $(V, [], []) \in \text{dependent-live-vars } n'$ **by** *fact*

thus *?case*

by(*fastforce elim:dependent-live-vars.cases simp:targetnodes-def sourcenodes-def*)

next

case (*Cons ax asx*)

note $IH = \langle \llbracket (V, asx, asx) \in \text{dependent-live-vars } n';$
 $asx \neq [] \longrightarrow n' = \text{last } (\text{targetnodes } asx) \rrbracket$
 $\implies V \in \text{Use } n' \wedge (\forall as' a as''. asx = as' @ a \# as'' \longrightarrow$
 $\neg \text{sourcenode } a - \{V\} a \# as'' \rightarrow_{dd} n') \rangle$

from $\langle (V, ax \# asx, ax \# asx) \in \text{dependent-live-vars } n' \rangle$

have $(V, asx, asx) \in \text{dependent-live-vars } n'$

and $\neg \text{sourcenode } ax - \{V\} ax \# asx \rightarrow_{dd} \text{last}(\text{targetnodes } (ax \# asx))$

by(*auto elim:dependent-live-vars.cases*)

from $\langle ax \# asx \neq [] \longrightarrow n' = \text{last } (\text{targetnodes } (ax \# asx)) \rangle$

have $n' = \text{last } (\text{targetnodes } (ax \# asx))$ **by** *simp*

show *?case*

proof(*cases asx = []*)

case *True*

with $\langle (V, asx, asx) \in \text{dependent-live-vars } n' \rangle$ **have** $V \in \text{Use } n'$

by(*fastforce elim:dependent-live-vars.cases*)

from $\langle \neg \text{sourcenode } ax - \{V\} ax \# asx \rightarrow_{dd} \text{last}(\text{targetnodes } (ax \# asx)) \rangle$

True $\langle n' = \text{last } (\text{targetnodes } (ax \# asx)) \rangle$

have $\forall as' a as''. ax \# asx = as' @ a \# as'' \longrightarrow \neg \text{sourcenode } a - \{V\} a \# as'' \rightarrow_{dd} n'$

by *auto(case-tac as', auto)*

with $\langle V \in \text{Use } n' \rangle$ **show** *?thesis* **by** *simp*

next

case *False*

with $\langle n' = \text{last } (\text{targetnodes } (ax \# asx)) \rangle$

have $asx \neq [] \longrightarrow n' = \text{last } (\text{targetnodes } asx)$

by(*simp add:targetnodes-def*)

from $IH[OF \langle (V, asx, asx) \in \text{dependent-live-vars } n' \rangle \text{ this}]$

have $V \in \text{Use } n' \wedge (\forall as' a as''. asx = as' @ a \# as'' \longrightarrow$
 $\neg \text{sourcenode } a - \{V\} a \# as'' \rightarrow_{dd} n') .$

with $\langle \neg \text{sourcenode } ax - \{V\} ax \# asx \rightarrow_{dd} \text{last}(\text{targetnodes } (ax \# asx)) \rangle$

$\langle n' = \text{last } (\text{targetnodes } (ax \# asx)) \rangle$ **have** $V \in \text{Use } n'$

and $\forall as' a as''. ax \# asx = as' @ a \# as'' \longrightarrow$
 $\neg \text{sourcenode } a - \{V\} a \# as'' \rightarrow_{dd} n'$

by *auto(case-tac as', auto)*

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

qed

qed

lemma *dependent-live-vars-same-paths:*
 $as \neq [] \longrightarrow n' = \text{last}(\text{targetnodes } as) \implies$

$(V, as, as) \in \text{dependent-live-vars } n' =$
 $(V \in \text{Use } n' \wedge (\forall as' a as''. as = as' @ a \# as'' \longrightarrow$
 $\neg \text{sourcenode } a - \{V\} a \# as'' \rightarrow_{dd} n'))$
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** $V' \in \text{Use } n''$
shows $(V', [], as) \in \text{dependent-live-vars } n'$
proof(*cases as*)
 case *Nil*
 with $\langle n'' - as \rightarrow_{cd} n' \rangle$ **show** ?thesis
 by(*fastforce elim:DynPDG-edge.cases dest:dyn-control-dependence-path*)
next
 case (*Cons ax asx*)
 with $\langle n'' - as \rightarrow_{cd} n' \rangle$ **have** *sourcenode ax = n''*
 by(*auto dest:DynPDG-cdep-edge-CFG-path elim:path.cases*)
 from $\langle n'' - as \rightarrow_{cd} n' \rangle$ **have** *valid-node n'*
 by(*fastforce intro:path-valid-node(2) DynPDG-cdep-edge-CFG-path(1)*)
 from *Cons* $\langle n'' - as \rightarrow_{cd} n' \rangle$ **have** *last(targetnodes as) = n'*
 by(*fastforce intro:path-targetnode dest:DynPDG-cdep-edge-CFG-path*)
 with *Cons* $\langle n'' - as \rightarrow_{cd} n' \rangle \langle V' \in \text{Use } n'' \rangle \langle \text{sourcenode ax} = n'' \rangle \langle \text{valid-node } n' \rangle$
 have $(V', [], ax \# asx @ []) \in \text{dependent-live-vars } n'$
 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** $V' \in \text{Use } n''$
shows $(V', [], as) \in \text{dependent-live-vars } n'$
proof(*cases as*)
 case *Nil*
 with $\langle n'' - \{V\} as \rightarrow_{dd} n' \rangle$ **show** ?thesis
 by(*fastforce elim:DynPDG-edge.cases simp:dyn-data-dependence-def*)
next
 case (*Cons ax asx*)
 with $\langle n'' - \{V\} as \rightarrow_{dd} n' \rangle$ **have** *sourcenode ax = n''*
 by(*auto dest:DynPDG-ddep-edge-CFG-path elim:path.cases*)
 from *Cons* $\langle n'' - \{V\} as \rightarrow_{dd} n' \rangle$ **have** *last(targetnodes as) = n'*
 by(*fastforce intro:path-targetnode elim:DynPDG-ddep-edge-CFG-path(1)*)
 from *Cons* $\langle n'' - \{V\} as \rightarrow_{dd} n' \rangle$ **have** *all: $\forall as' a as''. asx = as' @ a \# as'' \longrightarrow$*
 $\neg \text{sourcenode } a - \{V\} a \# as'' \rightarrow_{dd} n'$
 by(*fastforce dest:DynPDG-ddep-edge-no-shorter-ddep-edge*)
 from $\langle n'' - \{V\} as \rightarrow_{dd} n' \rangle$ **have** $V \in \text{Use } n'$
 by(*auto elim!:DynPDG-edge.cases simp:dyn-data-dependence-def*)
 from *Cons* $\langle n'' - \{V\} as \rightarrow_{dd} n' \rangle$ **have** $as \neq [] \longrightarrow n' = \text{last(targetnodes as)}$
 by(*fastforce dest:DynPDG-ddep-edge-CFG-path path-targetnode*)
 with *Cons* **have** $asx \neq [] \longrightarrow n' = \text{last(targetnodes asx)}$

```

    by(fastforce simp:targetnodes-def)
  with all ⟨V ∈ Use n'⟩ have (V,asx,asx) ∈ dependent-live-vars n'
    by -(rule dependent-live-vars-same-pathsI)
  with ⟨V' ∈ Use n''⟩ ⟨n'' - {V} as →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 →dd n'' and as' ≠ []
  and (V,as'',as') ∈ dependent-live-vars n'
  shows (V,as@as'',as@as') ∈ dependent-live-vars n'

```

proof -

```

  from ⟨n - {V} a#as →dd n''⟩ have ∀ n'' ∈ set (sourcenodes as). V ∉ Def n''
    by(auto elim:DynPDG-edge.cases simp:dyn-data-dependence-def)
  with ⟨(V,as'',as') ∈ 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') ∈ dependent-live-vars n';
      ∀ n'' ∈ set (sourcenodes asx). V ∉ Def n'']
      ⇒ (V, asx@as'',asx@as') ∈ dependent-live-vars n'⟩
    from ⟨∀ n'' ∈ set (sourcenodes (ax#asx)). V ∉ Def n''⟩
    have ∀ n'' ∈ set (sourcenodes asx). V ∉ Def n''

```

```

      by(auto simp:sourcenodes-def)
    from IH[OF ⟨(V,as'',as') ∈ dependent-live-vars n'⟩ this]
    have (V,asx@as'',asx@as') ∈ dependent-live-vars n'.
    from ⟨as' ≠ []⟩ ⟨(V,as'',as') ∈ dependent-live-vars n'⟩
    have n' = last(targetnodes as')

```

```

      by(fastforce intro:dependent-live-vars-lastnode)
    with ⟨as' ≠ []⟩ have n' = last(targetnodes (ax#asx@as'))
      by(fastforce simp:targetnodes-def)

```

```

  have ¬ sourcenode ax - {V} ax#asx@as'' →dd last(targetnodes (ax#asx@as'))
  proof

```

```

    assume sourcenode ax - {V} ax#asx@as'' →dd last(targetnodes (ax#asx@as'))
    hence sourcenode ax - {V} ax#asx@as'' →dd last(targetnodes (ax#asx@as'))
      by simp
    with ⟨∀ n'' ∈ set (sourcenodes (ax#asx)). V ∉ Def n''⟩
    show False

```

```

      by(fastforce elim:DynPDG-edge.cases
        simp:dyn-data-dependence-def sourcenodes-def)

```

qed

```

  with ⟨(V,asx@as'',asx@as') ∈ 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 $(V', as', as) \in \text{dependent-live-vars } n''$
 shows $(V', as', as @ as'') \in \text{dependent-live-vars } n'$
proof –
 from $\langle n'' - as'' \rightarrow_{cd} n' \rangle$ have $as'' \neq []$
 by (fastforce elim: DynPDG-edge.cases dest: dyn-control-dependence-path)
 with $\langle n'' - as'' \rightarrow_{cd} n' \rangle$ have $\text{last}(\text{targetnodes } as'') = n'$
 by (fastforce intro: path-targetnode elim: DynPDG-cdep-edge-CFG-path(1))
 from $\langle (V', as', as) \in \text{dependent-live-vars } n'' \rangle$ show ?thesis
proof (induct rule: dependent-live-vars.induct)
 case (dep-vars-Use V')
 from $\langle V' \in \text{Use } n'' \rangle \langle n'' - as'' \rightarrow_{cd} n' \rangle \langle \text{last}(\text{targetnodes } as'') = n' \rangle$ 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 $\langle n'' - as'' \rightarrow_{cd} n' \rangle$ 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 (rule DynPDG-path-Append)
 with $\langle V \in \text{Use } (\text{sourcenode } a) \rangle \langle (\text{sourcenode } a) - a \# as' \rightarrow_{cd} nx \rangle$
 show ?case by (fastforce intro: dependent-live-vars.dep-vars-Cons-cdep)
 next
 case (dep-vars-Cons-ddep $V as' as V' a$)
 from $\langle as'' \neq [] \rangle \langle \text{last}(\text{targetnodes } as'') = n' \rangle$
 have $n' = \text{last}(\text{targetnodes } ((a \# as) @ as''))$
 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 $\langle as'' \neq [] \rangle \langle \text{last}(\text{targetnodes } as'') = n' \rangle$
 have $n' = \text{last}(\text{targetnodes } ((a \# as) @ as''))$
 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 $(V', as', as) \in \text{dependent-live-vars } n''$
 shows $(V', as', as @ as'') \in \text{dependent-live-vars } n'$
proof –
 from $\langle n'' - \{V\} as'' \rightarrow_{dd} n' \rangle$ have $as'' \neq []$
 by (rule DynPDG-ddep-edge-CFG-path(2))

with $\langle n'' - \{V\} as'' \rightarrow_{dd} n' \rangle$ **have** $last(targetnodes\ as'') = n'$
by (*fastforce* *intro: path-targetnode elim: DynPDG-ddep-edge-CFG-path(1)*)
from $\langle n'' - \{V\} as'' \rightarrow_{dd} n' \rangle$ **have** $notExit: n' \neq (-Exit)$
by (*fastforce* *elim: DynPDG-edge.cases simp: dyn-data-dependence-def*)
from $\langle (V', as', as) \in dependent-live-vars\ n' \rangle$ **show** *?thesis*
proof (*induct rule: dependent-live-vars.induct*)
case (*dep-vars-Use V'*)
from $\langle V' \in Use\ n'' \rangle \langle n'' - \{V\} as'' \rightarrow_{dd} n' \rangle \langle last(targetnodes\ as'') = n' \rangle$ **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 $\langle n'' - \{V\} as'' \rightarrow_{dd} n' \rangle$ **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 $-(rule\ DynPDG-path-Append)$
with $\langle V' \in Use\ (sourcenode\ a) \rangle \langle sourcenode\ a - a \# as' \rightarrow_{cd} nx \rangle$ $notExit$
show *?case* **by** (*fastforce* *intro: dependent-live-vars.dep-vars-Cons-cdep*)
next
case (*dep-vars-Cons-ddep V as' as V' a*)
from $\langle as'' \neq [] \rangle \langle last(targetnodes\ as'') = n' \rangle$
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 $\langle as'' \neq [] \rangle \langle last(targetnodes\ as'') = n' \rangle$
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:*
 $\llbracket n'' - as'' \rightarrow_{d*} n'; (V', as', as) \in dependent-live-vars\ n' \rrbracket$
 $\implies (V', as', as @ as'') \in dependent-live-vars\ n'$
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 $IH = \langle (V', as', as) \in dependent-live-vars\ n \implies$
 $(V', as', as @ asx) \in dependent-live-vars\ n'' \rangle$
from $IH[OF\ \langle (V', as', as) \in dependent-live-vars\ n \rangle]$
have $(V', as', as @ asx) \in dependent-live-vars\ n''$
with $\langle n'' - asx' \rightarrow_{cd} n' \rangle$ **have** $(V', as', (as @ asx) @ asx') \in dependent-live-vars\ n'$
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 =  $\langle (V', as', as) \in \text{dependent-live-vars } n \implies$ 
     $(V', as', as @ asx) \in \text{dependent-live-vars } n'' \rangle$ 
  from IH[OF  $\langle (V', as', as) \in \text{dependent-live-vars } n \rangle$ ]
  have  $(V', as', as @ asx) \in \text{dependent-live-vars } n''$  .
  with  $\langle n'' - \{V\} asx' \rightarrow_{dd} n' \rangle$  have  $(V', as', (as @ asx) @ asx') \in \text{dependent-live-vars}$ 
   $n'$ 
  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-legs :: bit-vector  $\Rightarrow$  bit-vector  $\Rightarrow$  bool ( $- \preceq_b -$  99)
  where bv-Nils: []  $\preceq_b$  [] = True
    | bv-Cons:  $(x \# xs) \preceq_b (y \# ys) = ((x \longrightarrow y) \wedge xs \preceq_b ys)$ 
    | bv-rest:  $xs \preceq_b ys = \text{False}$ 

```

2.3.1 Some basic properties

lemma *bv-length*: $xs \preceq_b ys \implies \text{length } xs = \text{length } ys$
by(*induct rule:bv-legs.induct*)*auto*

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

lemma *bv-legs-AppendI*:
 $\llbracket xs \preceq_b ys; xs' \preceq_b ys' \rrbracket \implies (xs @ xs') \preceq_b (ys @ ys')$
by(*induct xs ys rule:bv-legs.induct, auto*)

lemma *bv-legs-AppendD*:
 $\llbracket (xs @ xs') \preceq_b (ys @ ys'); \text{length } xs = \text{length } ys \rrbracket$
 $\implies xs \preceq_b ys \wedge xs' \preceq_b ys'$
by(*induct xs ys rule:bv-legs.induct, auto*)

```

lemma bv-leqs-eq:
   $xs \preceq_b ys = ((\forall i < \text{length } xs. xs ! i \longrightarrow ys ! i) \wedge \text{length } xs = \text{length } ys)$ 
proof(induct xs ys rule:bv-leqs.induct)
  case ( $2\ x\ xs\ y\ ys$ )
  note  $eq = \langle xs \preceq_b ys =$ 
     $((\forall i < \text{length } xs. xs ! i \longrightarrow ys ! i) \wedge \text{length } xs = \text{length } ys) \rangle$ 
  show ?case
  proof
    assume  $leqs: x \# xs \preceq_b y \# ys$ 
    with  $eq$  have  $x \longrightarrow y$  and  $\forall i < \text{length } xs. xs ! i \longrightarrow ys ! i$ 
    and  $\text{length } xs = \text{length } ys$  by simp-all
    from  $\langle x \longrightarrow y \rangle$  have  $(x \# xs) ! 0 \longrightarrow (y \# ys) ! 0$  by simp
    { fix  $i$  assume  $i > 0$  and  $i < \text{length } (x \# xs)$ 
      then obtain  $j$  where  $i = \text{Suc } j$  and  $j < \text{length } xs$  by(cases i) auto
      with  $\langle \forall i < \text{length } xs. xs ! i \longrightarrow ys ! i \rangle$ 
      have  $(x \# xs) ! i \longrightarrow (y \# ys) ! i$  by auto }
    hence  $\forall i < \text{length } (x \# xs). i > 0 \longrightarrow (x \# xs) ! i \longrightarrow (y \# ys) ! i$  by simp
    with  $\langle (x \# xs) ! 0 \longrightarrow (y \# ys) ! 0 \rangle$   $\langle \text{length } xs = \text{length } ys \rangle$ 
    show  $(\forall i < \text{length } (x \# xs). (x \# xs) ! i \longrightarrow (y \# ys) ! i) \wedge$ 
       $\text{length } (x \# xs) = \text{length } (y \# ys)$ 
    by clarsimp(case-tac i>0,auto)
  next
    assume  $(\forall i < \text{length } (x \# xs). (x \# xs) ! i \longrightarrow (y \# ys) ! i) \wedge$ 
       $\text{length } (x \# xs) = \text{length } (y \# ys)$ 
    hence  $\forall i < \text{length } (x \# xs). (x \# xs) ! i \longrightarrow (y \# ys) ! i$ 
    and  $\text{length } (x \# xs) = \text{length } (y \# ys)$  by simp-all
    from  $\langle \forall i < \text{length } (x \# xs). (x \# xs) ! i \longrightarrow (y \# ys) ! i \rangle$ 
    have  $\forall i < \text{length } xs. xs ! i \longrightarrow ys ! i$ 
    by clarsimp(erule-tac x=Suc i in allE,auto)
    with  $eq$   $\langle \text{length } (x \# xs) = \text{length } (y \# ys) \rangle$  have  $xs \preceq_b ys$  by simp
    from  $\langle \forall i < \text{length } (x \# xs). (x \# xs) ! i \longrightarrow (y \# ys) ! i \rangle$ 
    have  $x \longrightarrow y$  by(erule-tac x=0 in allE) simp
    with  $\langle xs \preceq_b ys \rangle$  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*:
 $\text{replicate } (\text{length } xs) \text{ False } \preceq_b xs$
by(*induct xs*) *auto*

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

lemma *bv-leqs-refl*: $xs \preceq_b xs$

```

by(induct xs) auto

lemma bv-legs-trans:  $\llbracket xs \preceq_b ys; ys \preceq_b zs \rrbracket \implies xs \preceq_b zs$ 
proof(induct xs ys arbitrary:zs rule:bv-legs.induct)
  case (2 x xs y ys)
  note IH =  $\langle \bigwedge zs. \llbracket xs \preceq_b ys; ys \preceq_b zs \rrbracket \implies xs \preceq_b zs \rangle$ 
  from  $\langle (x \# xs) \preceq_b (y \# ys) \rangle$  have  $xs \preceq_b ys$  and  $x \longrightarrow y$  by simp-all
  from  $\langle (y \# ys) \preceq_b zs \rangle$  obtain  $z \ zs'$  where  $zs = z \# zs'$  by (cases zs) auto
  with  $\langle (y \# ys) \preceq_b zs \rangle$  have  $ys \preceq_b zs'$  and  $y \longrightarrow z$  by simp-all
  from IH[OF  $\langle xs \preceq_b ys \rangle \langle ys \preceq_b zs' \rangle$ ] have  $xs \preceq_b zs'$ .
  with  $\langle x \longrightarrow y \rangle \langle y \longrightarrow z \rangle \langle zs = z \# zs' \rangle$  show ?case by simp
qed simp-all

lemma bv-legs-antisym:  $\llbracket xs \preceq_b ys; ys \preceq_b xs \rrbracket \implies xs = ys$ 
by(induct xs ys rule:bv-legs.induct) auto

definition bv-less :: bit-vector  $\Rightarrow$  bit-vector  $\Rightarrow$  bool (- <b - 99)
  where  $xs <_b ys \equiv xs \preceq_b ys \wedge xs \neq ys$ 

interpretation order bv-legs bv-less
by(unfold-locales,
  auto intro:bv-legs-refl bv-legs-trans bv-legs-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  $\Rightarrow$  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  $\exists a' as'. as = a'#as' \wedge slice-path as' = xs$ 
  by(induct as) auto
qed

```

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

```

fun select-edge-kinds :: 'edge list  $\Rightarrow$  bit-vector  $\Rightarrow$  '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  $\uparrow f \Rightarrow \uparrow id \mid (Q)_{\surd} \Rightarrow (\lambda s. True)_{\surd}$ ))#select-edge-kinds as
  bs

```

```

definition slice-kinds :: 'edge list  $\Rightarrow$  '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-legs-information-same-Uses:
   $\llbracket n - as \rightarrow^* n'; bs \preceq_b bs'; slice-path as = bs;$ 
  select-edge-kinds as bs = es; select-edge-kinds as bs' = es';
   $\forall V xs. (V, xs, as) \in dependent-live-vars n' \longrightarrow state-val s V = state-val s' V;$ 
  preds es' s  $\llbracket$ 
 $\implies (\forall V \in Use n'. state-val (transfers es s) V =$ 
  state-val (transfers es' s') V  $\wedge$  preds es s
proof(induct bs bs' arbitrary:as es es' n s s' rule:bv-legs.induct)
  case 1
  from  $\langle slice-path as = [] \rangle$  have as = [] by(cases as) auto
  with  $\langle select-edge-kinds as [] = es \rangle \langle select-edge-kinds as [] = es' \rangle$ 
  have es = [] and es' = [] by simp-all
  { fix V assume  $V \in Use n'$ 
    hence  $(V, [], []) \in dependent-live-vars n'$  by(rule dep-vars-Use)
    with  $\langle \forall V xs. (V, xs, as) \in dependent-live-vars n' \longrightarrow$ 
      state-val s V = state-val s' V  $\rangle \langle V \in Use n' \rangle \langle as = [] \rangle$ 
    have state-val s V = state-val s' V by blast }
  with  $\langle es = [] \rangle \langle es' = [] \rangle$  show ?case by simp
next
  case (2 x xs y ys)
  note all =  $\langle \forall V xs. (V, xs, as) \in dependent-live-vars n' \longrightarrow$ 

```


$state-val\ s\ V = state-val\ s'\ V$
note $IH = \langle \bigwedge as\ es\ es'\ n\ s\ s'. \llbracket n - as \rightarrow * n'; xs \preceq_b ys; slice-path\ as = xs;$
 $select-edge-kinds\ as\ xs = es; select-edge-kinds\ as\ ys = es';$
 $\forall V\ xs. (V, xs, as) \in dependent-live-vars\ n' \longrightarrow$
 $state-val\ s\ V = state-val\ s'\ V;$
 $preds\ es'\ s' \rrbracket$
 $\implies (\forall V \in Use\ n'. state-val\ (transfers\ es\ s)\ V =$
 $state-val\ (transfers\ es'\ s')\ V) \wedge preds\ es\ s \rangle$
from $\langle x \# xs \preceq_b y \# ys \rangle$ **have** $x \longrightarrow y$ **and** $xs \preceq_b ys$ **by** *simp-all*
from $\langle slice-path\ as = x \# xs \rangle$ **obtain** $a' as'$ **where** $as = a' \# as'$
and $slice-path\ as' = xs$ **by** (*erule slice-path-right-Cons*)
from $\langle as = a' \# as' \rangle \langle select-edge-kinds\ as\ (x \# xs) = es \rangle$
obtain $ex\ esx$ **where** $es = ex \# esx$
and $ex:ex = (if\ x\ then\ kind\ a'$
 $else\ (case\ kind\ a'\ of\ \uparrow f \Rightarrow \uparrow id \mid (Q)_{\checkmark} \Rightarrow (\lambda s. True)_{\checkmark}))$
and $select-edge-kinds\ as'\ xs = esx$ **by** *auto*
from $\langle as = a' \# as' \rangle \langle select-edge-kinds\ as\ (y \# ys) = es' \rangle$ **obtain** $ex'\ esx'$
where $es' = ex' \# esx'$
and $ex':ex' = (if\ y\ then\ kind\ a'$
 $else\ (case\ kind\ a'\ of\ \uparrow f \Rightarrow \uparrow id \mid (Q)_{\checkmark} \Rightarrow (\lambda s. True)_{\checkmark}))$
and $select-edge-kinds\ as'\ ys = esx'$ **by** *auto*
from $\langle n - as \rightarrow * n' \rangle \langle as = a' \# as' \rangle$ **have** $[simp]:n = sourcenode\ a'$
and $valid-edge\ a'$ **and** $targetnode\ a' - as' \rightarrow * n'$
by (*auto elim:path-split-Cons*)
from $\langle n - as \rightarrow * n' \rangle \langle as = a' \# as' \rangle$ **have** $last(targetnodes\ as) = n'$
by (*fastforce intro:path-targetnode*)
from $\langle preds\ es'\ s' \rangle \langle es' = ex' \# esx' \rangle$ **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 $\langle slice-path\ as' = xs \rangle \langle xs \preceq_b ys \rangle$
have $[simp]:xs = [] \wedge ys = []$ **by** *auto(cases ys, auto) +*
with $\langle select-edge-kinds\ as'\ xs = esx \rangle \langle select-edge-kinds\ as'\ ys = esx' \rangle$
have $[simp]:esx = []$ **and** $[simp]:esx' = []$ **by** *simp-all*
from *True* $\langle targetnode\ a' - as' \rightarrow * n' \rangle$
have $[simp]:n' = targetnode\ a'$ **by** (*auto elim:path.cases*)
show *?thesis*
proof (*cases x*)
case *True*
with $\langle x \longrightarrow y \rangle$ $ex\ ex'$ **have** $[simp]:ex = kind\ a' \wedge ex' = kind\ a'$ **by** *simp*
have $pred\ ex\ s$
proof (*cases ex*)
case (*Predicate Q*)
with $ex\ ex'\ True\ \langle x \longrightarrow y \rangle$ **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  $V' \in Use\ n$ 
    with  $\langle True \rangle \langle valid-edge\ a' \rangle$ 
    have  $(V', [], a' \# [] @ []) \in dependent-live-vars\ n'$ 
      by(fastforce intro:dep-vars-Cons-cdep DynPDG-path-Nil
        simp:targetnodes-def)
    with all  $\langle as = a' \# as' \rangle$  have state-val  $s\ V' = state-val\ s'\ V'$ 
      by fastforce }
  with  $\langle pred\ ex'\ s' \rangle \langle valid-edge\ a' \rangle$ 
  show ?thesis by(fastforce elim:CFG-edge-Uses-pred-equal)
next
  case False
  from ex True Predicate have  $kind\ a' = (Q)_{\checkmark}$  by(auto split:split-if-asm)
  from  $\langle True \rangle \langle slice-path\ as = x \# xs \rangle \langle as = a' \# as' \rangle$  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  $\langle [a'] = asx @ asx' \rangle$ 
  have  $(asx = [a'] \wedge asx' = []) \vee (asx = [] \wedge asx' = [a'])$ 
    by (cases asx) auto
  hence False
proof
  assume  $asx = [a'] \wedge asx' = []$ 
  with  $\langle n'' - asx' \rightarrow_{cd} nx' \rangle$  show False
  by(fastforce elim:DynPDG-edge.cases dest:dyn-control-dependence-path)
next
  assume  $asx = [] \wedge asx' = [a']$ 
  with  $\langle nx - asx \rightarrow_{d*} n'' \rangle$  have  $nx = n''$  and  $asx' = [a']$ 
    by(auto intro:DynPDG-empty-path-eq-nodes)
  with  $\langle n = nx \rangle \langle n' = nx' \rangle \langle n'' - asx' \rightarrow_{cd} nx' \rangle$  show False by simp
qed
thus ?thesis by simp
next
  case (DynPDG-path-Append-ddep  $nx\ asx\ n''\ V\ asx'\ nx'$ )
  from  $\langle [a'] = asx @ asx' \rangle$ 
  have  $(asx = [a'] \wedge asx' = []) \vee (asx = [] \wedge asx' = [a'])$ 
    by (cases asx) auto
  thus ?case
proof
  assume  $asx = [a'] \wedge asx' = []$ 
  with  $\langle n'' - \{V\} asx' \rightarrow_{dd} nx' \rangle$  have False
    by(fastforce elim:DynPDG-edge.cases simp:dyn-data-dependence-def)

```

```

    thus ?thesis by simp
  next
    assume  $asx = [] \wedge asx' = [a']$ 
    with  $\langle nx - asx \rightarrow_d^* n'' \rangle$  have  $nx = n''$ 
    by (simp add: DynPDG-empty-path-eq-nodes)
    { fix  $V'$  assume  $V' \in Use\ n$ 
      from  $\langle n'' - \{V\} asx' \rightarrow_{dd} nx' \rangle$   $\langle asx = [] \wedge asx' = [a'] \rangle$   $\langle n' = nx' \rangle$ 
      have  $(V, [], []) \in dependent-live-vars\ n'$ 
      by (fastforce intro: dep-vars-Use elim: DynPDG-edge.cases
          simp: dyn-data-dependence-def)
      with  $\langle V' \in Use\ n \rangle$   $\langle n'' - \{V\} asx' \rightarrow_{dd} nx' \rangle$   $\langle asx = [] \wedge asx' = [a'] \rangle$ 
       $\langle n = nx \rangle$   $\langle nx = n'' \rangle$   $\langle n' = nx' \rangle$ 
      have  $(V', [], [a']) \in dependent-live-vars\ n'$ 
      by (auto elim: dep-vars-Cons-ddep simp: targetnodes-def)
      with all  $\langle as = a' \# as' \rangle$  have  $state-val\ s\ V' = state-val\ s'\ V'$ 
      by fastforce }
    with  $\langle pred\ ex'\ s' \rangle$   $\langle valid-edge\ a' \rangle$   $ex\ ex'\ True$   $\langle x \longrightarrow y \rangle$  show ?thesis
    by (fastforce elim: CFG-edge-Uses-pred-equal)
  qed
qed
qed
qed simp
{ fix  $V$  assume  $V \in Use\ n'$ 
  from  $\langle V \in Use\ n' \rangle$  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  $\bigwedge V. V \in Use\ n \implies state-val\ s\ V = state-val\ s'\ V$ 
    proof -
      fix  $V'$  assume  $V' \in Use\ n$ 
      with  $\langle (V, [], []) \in dependent-live-vars\ n' \rangle$  True
      have  $(V', [], [a']) \in dependent-live-vars\ n'$ 
      by (fastforce intro: dep-vars-Cons-ddep simp: targetnodes-def)
      with all  $\langle as = a' \# as' \rangle$  show  $state-val\ s\ V' = state-val\ s'\ V'$  by auto
    qed
  qed
  with  $\langle valid-edge\ a' \rangle$   $\langle pred\ ex'\ s' \rangle$   $\langle pred\ ex\ s \rangle$ 
  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  $\langle V \in Def\ n \rangle$  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  $\langle last(targetnodes\ as) = n' \rangle$   $\langle as = a' \# as' \rangle$ 

```

```

  ⟨(V, [], []) ∈ dependent-live-vars n'⟩
  have (V, [a'], [a']) ∈ dependent-live-vars n'
    by (fastforce intro: dep-vars-Cons-keep)
  from ⟨(V, [a'], [a']) ∈ 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 ∈ 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)✓ ⇒ (λs. True)✓)
  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'] →a* 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  $\forall V \in \text{Use } n'. \text{state-val (transfer ex s) } V =$ 
    state-val (transfer ex' s') V by simp
  with  $\langle as = a' \# as' \rangle \langle es = ex \# esx \rangle \langle es' = ex' \# esx' \rangle \langle \text{pred ex s} \rangle$ 
  show ?thesis by simp
next
case False
  with ex' have cases-y: ex' = (case kind a' of  $\uparrow f \Rightarrow \uparrow id \mid (Q)_{\checkmark} \Rightarrow (\lambda s.$ 
    True) $_{\checkmark}$ )
    by simp
  with cases-x have [simp]: ex = ex' by (cases kind a') auto
  { fix V assume  $V \in \text{Use } n'$ 
    from  $\langle V \in \text{Use } n' \rangle$  have  $(V, [], []) \in \text{dependent-live-vars } n'$ 
      by (rule dep-vars-Use)
    from  $\langle \text{slice-path } as = x \# xs \rangle \langle as = a' \# as' \rangle \langle \neg x \rangle$ 
    have  $\neg n - [a^{\uparrow}] \rightarrow_{d*} n'$  by (simp add: targetnodes-def)
    hence no-dep:  $\neg n - \{V\} [a^{\uparrow}] \rightarrow_{dd} n'$  by (fastforce dest: DynPDG-path-ddep)
    with  $\langle \text{last (targetnodes } as) = n' \rangle \langle as = a' \# as' \rangle$ 
     $\langle (V, [], []) \in \text{dependent-live-vars } n' \rangle$ 
    have  $(V, [a^{\uparrow}], [a^{\uparrow}]) \in \text{dependent-live-vars } n'$ 
      by (fastforce intro: dep-vars-Cons-keep)
    with all  $\langle as = a' \# as' \rangle$  have state-val s V = state-val s' V by auto }
  with  $\langle as = a' \# as' \rangle$  cases-x  $\langle es = ex \# esx \rangle \langle es' = ex' \# esx' \rangle \langle \text{pred ex s} \rangle$ 
  show ?thesis by (cases kind a', auto)
qed
qed
next
case False
show ?thesis
proof (cases  $\forall V xs. (V, xs, as') \in \text{dependent-live-vars } n' \longrightarrow$ 
  state-val (transfer ex s) V = state-val (transfer ex' s') V)
case True
  hence imp':  $\forall V xs. (V, xs, as') \in \text{dependent-live-vars } n' \longrightarrow$ 
    state-val (transfer ex s) V = state-val (transfer ex' s') V .
  from IH[OF  $\langle \text{targetnode } a' - as' \rightarrow^* n' \rangle \langle xs \preceq_b ys \rangle \langle \text{slice-path } as' = xs \rangle$ 
     $\langle \text{select-edge-kinds } as' xs = esx \rangle \langle \text{select-edge-kinds } as' ys = esx' \rangle$ 
    this  $\langle \text{preds } esx' (\text{transfer ex' s'}) \rangle]$ 
  have all':  $\forall V \in \text{Use } n'. \text{state-val (transfers esx (transfer ex s)) } V =$ 
    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  $\langle \text{slice-path } as = x \# xs \rangle \langle as = a' \# as' \rangle \langle \text{last (targetnodes } as) = n' \rangle$  ex
    have ex =  $(\lambda s. \text{True})_{\checkmark} \vee n - a' \# as' \rightarrow_{d*} n'$ 
      by (cases kind a', auto split: split-if-asm)
    thus ?thesis

```

```

proof
  assume  $ex = (\lambda s. \text{True})_{\surd}$  thus ?thesis by simp
next
  assume  $n - a' \# as' \rightarrow_d^* n'$ 
  with  $\langle \text{slice-path } as = x \# xs \rangle \langle as = a' \# as' \rangle \langle \text{last}(\text{targetnodes } as) = n' \rangle$   $ex$ 
  have [simp]:  $ex = \text{kind } a'$  by clarsimp
  with  $\langle x \rightarrow y \rangle$   $ex \ ex'$  have [simp]:  $ex' = ex$  by (cases  $x$ ) auto
  from  $\langle n - a' \# as' \rightarrow_d^* n' \rangle$  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  $\langle n - asx \rightarrow_{cd} n'' \rangle$  have  $asx \neq []$ 
  by (auto elim: DynPDG-edge.cases dest: dyn-control-dependence-path)
  with  $\langle n - asx \rightarrow_{cd} n'' \rangle \langle n'' - asx' \rightarrow_d^* n' \rangle \langle a' \# as' = asx @ asx' \rangle$ 
  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  $V \in \text{Use } n$ 
    with  $\langle \text{last}(\text{targetnodes } as) = n' \rangle \langle as = a' \# as' \rangle$ 
    have  $(V, [], as) \in \text{dependent-live-vars } n'$ 
    by (fastforce intro: dep-vars-Cons-cdep)
    with  $all$  have state-val  $s \ V = \text{state-val } s' \ V$  by blast }
  with  $\langle \text{valid-edge } a' \rangle \langle \text{pred } ex' \ s' \rangle$ 
  show ?thesis by (fastforce elim: CFG-edge-Uses-pred-equal)
next
  case (DynPDG-path-ddep-Append  $V \ n'' \ asx \ asx'$ )
  from  $\langle n - \{V\} asx \rightarrow_{dd} n'' \rangle$  obtain  $ai \ ais$  where  $asx = ai \# ais$ 
  by (cases  $asx$ ) (auto dest: DynPDG-ddep-edge-CFG-path)
  with  $\langle n - \{V\} asx \rightarrow_{dd} n'' \rangle$  have sourcenode  $ai = n$ 
  by (fastforce dest: DynPDG-ddep-edge-CFG-path elim: path.cases)
  from  $\langle n - \{V\} asx \rightarrow_{dd} n'' \rangle \langle asx = ai \# ais \rangle$ 
  have  $\text{last}(\text{targetnodes } asx) = n''$ 
  by (fastforce intro: path-targetnode dest: DynPDG-ddep-edge-CFG-path)
  { fix  $V'$  assume  $V' \in \text{Use } n$ 
    from  $\langle n - \{V\} asx \rightarrow_{dd} n'' \rangle$  have  $(V, [], []) \in \text{dependent-live-vars } n''$ 
    by (fastforce elim: DynPDG-edge.cases dep-vars-Use
      simp: dyn-data-dependence-def)
    with  $\langle n'' - asx' \rightarrow_d^* n' \rangle$  have  $(V, [], [] @ asx') \in \text{dependent-live-vars } n'$ 
    by (rule dependent-live-vars-dep-dependent-live-vars)
    have  $(V', [], as) \in \text{dependent-live-vars } n'$ 
    proof (cases  $asx' = []$ )
    case True
    with  $\langle n'' - asx' \rightarrow_d^* n' \rangle$  have  $n'' = n'$ 
    by (fastforce intro: DynPDG-empty-path-eq-nodes)
    with  $\langle n - \{V\} asx \rightarrow_{dd} n'' \rangle \langle V' \in \text{Use } n \rangle$  True  $\langle as = a' \# as' \rangle$ 
     $\langle a' \# as' = asx @ asx' \rangle$ 
  }

```

```

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

```

```

    ⟨valid-edge a'⟩ ⟨pred ex' s'⟩ ⟨pred ex s⟩
    CFG-edge-transfer-uses-only-Use[of a' s s']
obtain V'' where V'' ∈ Use n
    and state-val s V'' ≠ 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'→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'→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'→d* n'

```



```

    by  $\neg$ (drule dependent-live-vars-dependent-edge,
      auto dest: DynPDG-path-ddep DynPDG-path-Append)
  with False have False by simp }
hence  $V' \notin \text{Def}$  (sourcenode  $a'$ ) by fastforce
from False  $\langle \text{slice-path } as = x \# xs \rangle \langle as = a' \# as' \rangle$ 
   $\langle \text{last}(\text{targetnodes } as) = n' \rangle \langle as' \neq [] \rangle$ 
have  $\neg x$  by (auto simp: targetnodes-def)
with  $ex$  have cases:  $ex = (\text{case kind } a' \text{ of } \uparrow f \Rightarrow \uparrow id \mid (Q)_{\checkmark} \Rightarrow (\lambda s. \text{True})_{\checkmark})$ 
  by simp
have state-val  $s \ V' \neq \text{state-val } s' \ V'$ 
proof (cases  $y$ )
  case True
  with  $ex'$  have [simp]:  $ex' = \text{kind } a' \text{ by simp}$ 
  from  $\langle V' \notin \text{Def} \text{ (sourcenode } a') \rangle \langle \text{valid-edge } a' \rangle \langle \text{pred } ex' \ s' \rangle$ 
  have states-eq: state-val (transfer (kind  $a'$ )  $s'$ )  $V' = \text{state-val } s' \ V'$ 
    by (fastforce intro: CFG-edge-no-Def-equal)
  from cases have state-val  $s \ V' = \text{state-val } (\text{transfer } ex \ s) \ V'$ 
    by (cases kind  $a'$ ) auto
  with states-eq
     $\langle \text{state-val } (\text{transfer } ex \ s) \ V' \neq \text{state-val } (\text{transfer } ex' \ s') \ V' \rangle$ 
  show ?thesis by simp
next
  case False
  with  $ex'$  have  $ex' = (\text{case kind } a' \text{ of } \uparrow f \Rightarrow \uparrow id \mid (Q)_{\checkmark} \Rightarrow (\lambda s. \text{True})_{\checkmark})$ 
    by simp
  with cases have state-val  $s \ V' = \text{state-val } (\text{transfer } ex \ s) \ V'$ 
    and state-val  $s' \ V' = \text{state-val } (\text{transfer } ex' \ s') \ V'$ 
    by (cases kind  $a'$ , auto) +
  with  $\langle \text{state-val } (\text{transfer } ex \ s) \ V' \neq \text{state-val } (\text{transfer } ex' \ s') \ V' \rangle$ 
  show ?thesis by simp
qed
from  $\langle V' \notin \text{Def} \text{ (sourcenode } a') \rangle$ 
have  $\neg n - \{V'\} a' \# xs' \rightarrow_{dd} \text{last}(\text{targetnodes } (a' \# xs'))$ 
  by (auto elim: DynPDG-edge.cases simp: dyn-data-dependence-def)
with  $\langle (V', xs', as') \in \text{dependent-live-vars } n' \rangle \langle \text{last}(\text{targetnodes } as) = n' \rangle$ 
   $\langle as = a' \# as' \rangle$ 
have  $(V', a' \# xs', a' \# as') \in \text{dependent-live-vars } n'$ 
  by (fastforce intro: dep-vars-Cons-keep)
with  $\langle as = a' \# as' \rangle$  all have state-val  $s \ V' = \text{state-val } s' \ V'$  by auto
with  $\langle \text{state-val } s \ V' \neq \text{state-val } s' \ V' \rangle$  have False by simp
thus ?thesis by simp
qed
qed
qed
qed simp-all

```

theorem fundamental-property-of-path-slicing:
 assumes $n - as \rightarrow^* n'$ and preds (kinds as) s

shows $(\forall V \in \text{Use } n'. \text{state-val } (\text{transfers } (\text{slice-kinds } as) s) V =$
 $\text{state-val } (\text{transfers } (\text{kinds } as) s) V)$
and $\text{preds } (\text{slice-kinds } as) s$
proof –
have $\text{length } as = \text{length } (\text{slice-path } as)$ **by** $(\text{simp add:slice-path-length})$
hence $\text{slice-path } as \preceq_b \text{ replicate } (\text{length } as) \text{ True}$
by $(\text{simp add:maximal-element})$
have $\text{select-edge-kinds } as (\text{replicate } (\text{length } as) \text{ True}) = \text{kinds } as$
by $(\text{rule select-edge-kinds-max-bv})$
with $\langle n - as \rightarrow^* n' \rangle \langle \text{slice-path } as \preceq_b \text{ replicate } (\text{length } as) \text{ True} \rangle$
 $\langle \text{preds } (\text{kinds } as) s \rangle$
have $(\forall V \in \text{Use } n'. \text{state-val } (\text{transfers } (\text{slice-kinds } as) s) V =$
 $\text{state-val } (\text{transfers } (\text{kinds } as) s) V) \wedge \text{preds } (\text{slice-kinds } as) s$
by $-(\text{rule slice-path-legs-information-same-Uses, simp-all add:slice-kinds-def})$
thus $\forall V \in \text{Use } n'. \text{state-val } (\text{transfers } (\text{slice-kinds } as) s) V =$
 $\text{state-val } (\text{transfers } (\text{kinds } as) s) V$ **and** $\text{preds } (\text{slice-kinds } as) s$
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 $\text{sourcenode} :: 'edge \Rightarrow 'node$ **and** $\text{targetnode} :: 'edge \Rightarrow 'node$
and $\text{kind} :: 'edge \Rightarrow 'state \text{ edge-kind}$ **and** $\text{valid-edge} :: 'edge \Rightarrow \text{bool}$
and $\text{Entry} :: 'node \Rightarrow ('(-\text{Entry}'-))$ **and** $\text{Def} :: 'node \Rightarrow 'var \text{ set}$
and $\text{Use} :: 'node \Rightarrow 'var \text{ set}$ **and** $\text{state-val} :: 'state \Rightarrow 'var \Rightarrow 'val$
and $\text{dyn-control-dependence} :: 'node \Rightarrow 'node \Rightarrow 'edge \text{ list} \Rightarrow \text{bool}$
 $(- \text{ controls - via - } [51, 0, 0] 1000)$
and $\text{Exit} :: 'node \Rightarrow ('(-\text{Exit}'-))$
and $\text{sem} :: 'com \Rightarrow 'state \Rightarrow 'com \Rightarrow 'state \Rightarrow \text{bool}$
 $((1 \langle -,/- \rangle) \Rightarrow / (1 \langle -,/- \rangle)) [0,0,0,0] 81)$
and $\text{identifies} :: 'node \Rightarrow 'com \Rightarrow \text{bool} (- \triangleq - [51, 0] 80)$

begin

theorem *fundamental-property-of-path-slicing-semantically:*

assumes $n \triangleq c$ **and** $\langle c, s \rangle \Rightarrow \langle c', s' \rangle$
obtains n' **as** **where** $n - as \rightarrow^* n'$ **and** $\text{preds } (\text{slice-kinds } as) s$
and $n' \triangleq c'$
and $\forall V \in \text{Use } n'. \text{state-val } (\text{transfers } (\text{slice-kinds } as) s) V =$
 $\text{state-val } s' V$

proof (atomize-elim)

from $\langle n \triangleq c \rangle \langle \langle c, s \rangle \Rightarrow \langle c', s' \rangle \rangle$ **obtain** n' **as** **where** $n - as \rightarrow^* n'$

```

and transfers (kinds as) s = s'
and preds (kinds as) s
and n'  $\triangleq$  c'
by(fastforce dest:fundamental-property)
with  $\langle n - as \rightarrow^* n' \rangle \langle preds \text{ (kinds as) } s \rangle$ 
have  $\forall V \in Use\ n'.\ state-val \text{ (transfers (slice-kinds as) } s) \ V =$ 
 $state-val \text{ (transfers (kinds as) } s) \ V$  and preds (slice-kinds as) s
by  $-(rule\ fundamental-property-of-path-slicing, simp-all)+$ 
with  $\langle transfers \text{ (kinds as) } s = s' \rangle$  have  $\forall V \in Use\ n'.$ 
 $state-val \text{ (transfers (slice-kinds as) } s) \ V =$ 
 $state-val\ s' \ V$  by simp
with  $\langle n - as \rightarrow^* n' \rangle \langle preds \text{ (slice-kinds as) } s \rangle \langle n' \triangleq c' \rangle$ 
show  $\exists as\ n'.\ n - as \rightarrow^* n' \wedge preds \text{ (slice-kinds as) } s \wedge n' \triangleq c' \wedge$ 
 $(\forall V \in Use\ n'.\ state-val \text{ (transfers (slice-kinds as) } s) \ V = state-val\ s' \ V)$ 
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*:

$\llbracket n - as \rightarrow^* n'; \forall nx \in set(sourcenodes\ as). nx \notin S; n' \in S \rrbracket \Longrightarrow n' \in obs\ n\ S$

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 $\langle n' \in obs\ n\ S \rangle$

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  $\langle \text{valid-node } n \rangle$  have  $n - [] \rightarrow^* n$  by (rule empty-path)
  with  $\langle n \in S \rangle$  have  $n \in \text{obs } n \ S$  by (fastforce elim:obs-elem simp:sourcenodes-def)
  { fix  $n'$  assume  $n' \in \text{obs } n \ S$ 
    have  $n' = n$ 
    proof (rule ccontr)
      assume  $n' \neq n$ 
      from  $\langle n' \in \text{obs } n \ S \rangle$  obtain  $as$  where  $n - as \rightarrow^* n'$ 
        and  $\forall nx \in \text{set}(\text{sourcenodes } as). nx \notin S$ 
        and  $n' \in S$  by (erule obsE)
      from  $\langle n - as \rightarrow^* n' \rangle \langle \forall nx \in \text{set}(\text{sourcenodes } as). nx \notin S \rangle \langle n' \neq n \rangle \langle n \in S \rangle$ 
      show False
      proof (induct rule:path.induct)
        case (Cons-path  $n''$   $as$   $n'$   $a$   $n$ )
        from  $\langle \forall nx \in \text{set}(\text{sourcenodes } (a \# as)). nx \notin S \rangle \langle \text{sourcenode } a = n \rangle$ 
        have  $n \notin S$  by (simp add:sourcenodes-def)
        with  $\langle n \in S \rangle$  show False by simp
      qed simp
    qed }
  with  $\langle n \in \text{obs } n \ S \rangle$  show ?thesis by fastforce
qed

```

```

lemma in-obs-valid:
  assumes  $n' \in \text{obs } n \ S$  shows valid-node  $n$  and valid-node  $n'$ 
  using  $\langle n' \in \text{obs } n \ S \rangle$ 
  by (auto elim:obsE intro:path-valid-node)

```

```

lemma edge-obs-subset:
  assumes valid-edge  $a$  and sourcenode  $a \notin S$ 
  shows  $\text{obs } (\text{targetnode } a) \ S \subseteq \text{obs } (\text{sourcenode } a) \ S$ 
proof
  fix  $n$  assume  $n \in \text{obs } (\text{targetnode } a) \ S$ 
  then obtain  $as$  where  $\text{targetnode } a - as \rightarrow^* n$ 
    and  $\text{all:} \forall nx \in \text{set}(\text{sourcenodes } as). nx \notin S$  and  $n \in S$  by (erule obsE)
  from  $\langle \text{valid-edge } a \rangle \langle \text{targetnode } a - as \rightarrow^* n \rangle$ 
  have sourcenode  $a - a \# as \rightarrow^* n$  by (fastforce intro:Cons-path)
  moreover
  from  $\text{all } \langle \text{sourcenode } a \notin S \rangle$  have  $\forall nx \in \text{set}(\text{sourcenodes } (a \# as)). nx \notin S$ 
    by (simp add:sourcenodes-def)
  ultimately show  $n \in \text{obs } (\text{sourcenode } a) \ S$  using  $\langle n \in S \rangle$ 
    by (rule obs-elem)
qed

```

```

lemma path-obs-subset:
   $\llbracket n - as \rightarrow^* n'; \forall n' \in \text{set}(\text{sourcenodes } as). n' \notin S \rrbracket$ 
   $\implies \text{obs } n' \ S \subseteq \text{obs } n \ S$ 

```

```

proof(induct rule:path.induct)
  case (Cons-path  $n''$  as  $n'$   $a$   $n$ )
  note  $IH = \langle \forall n' \in \text{set } (\text{sourcenodes } as). n' \notin S \implies \text{obs } n' S \subseteq \text{obs } n'' S \rangle$ 
  from  $\langle \forall n' \in \text{set } (\text{sourcenodes } (a \# as)). n' \notin S \rangle$ 
  have  $\text{all} : \forall n' \in \text{set } (\text{sourcenodes } as). n' \notin S$  and  $\text{sourcenode } a \notin S$ 
    by(simp-all add:sourcenodes-def)
  from  $IH[OF \text{ all}]$  have  $\text{obs } n' S \subseteq \text{obs } n'' S$  .
  from  $\langle \text{valid-edge } a \rangle \langle \text{targetnode } a = n'' \rangle \langle \text{sourcenode } a = n \rangle \langle \text{sourcenode } a \notin S \rangle$ 
  have  $\text{obs } n'' S \subseteq \text{obs } n S$  by(fastforce dest:edge-obs-subset)
  with  $\langle \text{obs } n' S \subseteq \text{obs } n'' S \rangle$  show  $?case$  by fastforce
qed simp

```

```

lemma path-ex-obs:
  assumes  $n - as \rightarrow^* n'$  and  $n' \in S$ 
  obtains  $m$  where  $m \in \text{obs } n S$ 
proof(atomize-elim)
  show  $\exists m. m \in \text{obs } n S$ 
  proof(cases  $\forall nx \in \text{set}(\text{sourcenodes } as). nx \notin S$ )
    case True
    with  $\langle n - as \rightarrow^* n' \rangle \langle n' \in S \rangle$  have  $n' \in \text{obs } n S$  by  $-(\text{rule } \text{obs-elem})$ 
    thus  $?thesis$  by fastforce
  next
    case False
    hence  $\exists nx \in \text{set}(\text{sourcenodes } as). nx \in S$  by fastforce
    then obtain  $nx \ ns \ ns'$  where  $\text{sourcenodes } as = ns @ nx \# ns'$ 
      and  $nx \in S$  and  $\forall n' \in \text{set } ns. n' \notin S$ 
      by(fastforce elim!:split-list-first-propE)
    from  $\langle \text{sourcenodes } as = ns @ nx \# ns' \rangle$  obtain  $as' \ a \ as''$ 
      where  $ns = \text{sourcenodes } as'$ 
      and  $as = as' @ a \# as''$  and  $\text{sourcenode } a = nx$ 
      by(fastforce elim:map-append-append-maps simp:sourcenodes-def)
    with  $\langle n - as \rightarrow^* n' \rangle$  have  $n - as' \rightarrow^* nx$  by(fastforce dest:path-split)
    with  $\langle nx \in S \rangle \langle \forall n' \in \text{set } ns. n' \notin S \rangle \langle ns = \text{sourcenodes } as' \rangle$  have  $nx \in \text{obs } n$ 
     $S$ 
      by(fastforce intro:obs-elem)
    thus  $?thesis$  by fastforce
  qed
qed
end
end

```

Chapter 3

Static Intraprocedural Slicing

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 has to fulfill.

3.1 Distance of Paths

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

```
context CFG begin
```

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

```
where distanceI:
```

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

```
lemma every-path-distance:
```

```
  assumes  $n - as \rightarrow^* n'$ 
```

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

lemma *distance-det*:

$\llbracket \text{distance } n \ n' \ x; \text{distance } n \ n' \ x' \rrbracket \implies x = x'$
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 $\exists! a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{distance } (\text{targetnode } a') \ n' \ x \wedge$
 $\text{valid-edge } a' \wedge$
 $\text{targetnode } a' = (\text{SOME } nx. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$

$distance\ (targetnode\ a')\ n'\ x \wedge$
 $valid-edge\ a' \wedge targetnode\ a' = nx$

proof(rule *ex-ex1I*)

show $\exists a'.\ sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ n'\ x \wedge valid-edge\ a' \wedge$
 $targetnode\ a' = (SOME\ nx.\ \exists a'.\ sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ n'\ x \wedge$
 $valid-edge\ a' \wedge targetnode\ a' = nx)$

proof –

have $(\exists a'.\ sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ n'\ x \wedge valid-edge\ a' \wedge$
 $targetnode\ a' = (SOME\ nx.\ \exists a'.\ sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ n'\ x \wedge$
 $valid-edge\ a' \wedge targetnode\ a' = nx)) =$
 $(\exists nx.\ \exists a'.\ sourcenode\ a = sourcenode\ a' \wedge distance\ (targetnode\ a')\ n'\ x \wedge$
 $valid-edge\ a' \wedge targetnode\ a' = nx)$

apply(unfold *some-eq-ex*[of $\lambda nx.\ \exists a'.\ sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ n'\ x \wedge valid-edge\ a' \wedge targetnode\ a' = nx]$)

by *simp*

also have ... **using** *valid dist by blast*

finally show *?thesis* .

qed

next

fix $a'\ ax$

assume $sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ n'\ x \wedge valid-edge\ a' \wedge$
 $targetnode\ a' = (SOME\ nx.\ \exists a'.\ sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ n'\ x \wedge$
 $valid-edge\ a' \wedge targetnode\ a' = nx)$

and $sourcenode\ a = sourcenode\ ax \wedge$
 $distance\ (targetnode\ ax)\ n'\ x \wedge valid-edge\ ax \wedge$
 $targetnode\ ax = (SOME\ nx.\ \exists a'.\ sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ n'\ x \wedge$
 $valid-edge\ a' \wedge targetnode\ a' = nx)$

thus $a' = ax$ **by**(*fastforce intro!:edge-det*)

qed

lemma *distance-successor-distance*:

assumes $distance\ n\ n'\ x$ **and** $x \neq 0$

obtains a **where** $valid-edge\ a$ **and** $n = sourcenode\ a$

and $distance\ (targetnode\ a)\ n'\ (x - 1)$

and $targetnode\ a = (SOME\ nx.\ \exists a'.\ sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ n'\ (x - 1) \wedge$
 $valid-edge\ a' \wedge targetnode\ a' = nx)$

proof –

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 $\langle distance\ n\ n'\ x \rangle$ **obtain as where** $n - as \rightarrow^* n'$ **and** $x = length\ as$
and $\forall as'. n - as' \rightarrow^* n' \longrightarrow x \leq length\ as'$
by(auto elim:distance.cases)
thus False **using** imp
proof(induct rule:path.induct)
case (empty-path n)
from $\langle x = length\ [] \rangle \langle x \neq 0 \rangle$ **show** False **by** simp
next
case (Cons-path n'' as n')
note imp = $\langle \forall a. valid-edge\ a \wedge n = sourcenode\ a \wedge$
 $targetnode\ a = (SOME\ nx. \exists a'. sourcenode\ a = sourcenode\ a' \wedge$
 $distance\ (targetnode\ a')\ n'\ (x - 1) \wedge$
 $valid-edge\ a' \wedge targetnode\ a' = nx)$
 $\longrightarrow \neg distance\ (targetnode\ a)\ n'\ (x - 1) \rangle$
note all = $\langle \forall as'. n - as' \rightarrow^* n' \longrightarrow x \leq length\ as' \rangle$
from $\langle n'' - as \rightarrow^* n' \rangle$ **obtain y where** $distance\ n''\ n'\ y$
and $y \leq length\ as$ **by**(erule every-path-distance)
from $\langle distance\ n''\ n'\ y \rangle$ **obtain as' where** $n'' - as' \rightarrow^* n'$
and $y = length\ as'$
by(auto elim:distance.cases)
show False
proof(cases y < length as)
case True
from $\langle valid-edge\ a \rangle \langle sourcenode\ a = n \rangle \langle targetnode\ a = n'' \rangle \langle n'' - as' \rightarrow^* n' \rangle$
have $n - a\#as' \rightarrow^* n'$ **by** $\neg(rule\ path.Cons-path)$
with all **have** $x \leq length\ (a\#as')$ **by** blast
with $\langle x = length\ (a\#as) \rangle$ True $\langle y = length\ as' \rangle$ **show** False **by** simp
next
case False
with $\langle y \leq length\ as \rangle \langle x = length\ (a\#as) \rangle$ **have** $y = x - 1$ **by** simp
from $\langle targetnode\ a = n'' \rangle \langle distance\ n''\ n'\ y \rangle$
have $distance\ (targetnode\ a)\ n'\ y$ **by** simp
with $\langle valid-edge\ a \rangle$
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 ∧
valid-edge a' ∧ targetnode a' = nx)
  by(auto dest:only-one-SOME-dist-edge)
with imp ⟨sourcenode a = n⟩ ⟨y = x - 1⟩ show False by fastforce
qed
qed
qed
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  $\Rightarrow$  'var set
and Use :: 'node  $\Rightarrow$  'var set and state-val :: 'state  $\Rightarrow$  'var  $\Rightarrow$  'val +
fixes backward-slice :: 'node set  $\Rightarrow$  'node set
assumes valid-nodes: $n \in \text{backward-slice } S \implies \text{valid-node } n$ 
and ref: $\llbracket \text{valid-node } n; n \in S \rrbracket \implies n \in \text{backward-slice } S$ 
and dd-closed: $\llbracket n' \in \text{backward-slice } S; n \text{ influences } V \text{ in } n' \rrbracket$ 
 $\implies n \in \text{backward-slice } S$ 
and obs-finite:finite (obs n (backward-slice S))
and obs-singleton:card (obs n (backward-slice S))  $\leq 1$ 

begin

lemma slice-n-in-obs:
   $n \in \text{backward-slice } S \implies \text{obs } n (\text{backward-slice } S) = \{n\}$ 
by(fastforce intro!:n-in-obs dest:valid-nodes)

lemma obs-singleton-disj:
   $(\exists m. \text{obs } n (\text{backward-slice } S) = \{m\}) \vee \text{obs } n (\text{backward-slice } S) = \{\}$ 
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  $\langle \text{finite}(\text{obs } n (\text{backward-slice } S)) \rangle$  have obs n (backward-slice S) =  $\{\}$ 
    by simp
    thus ?thesis by simp
  next
    case False
    have card(obs n (backward-slice S))  $\leq 1$  by(rule obs-singleton)
    with False have card(obs n (backward-slice S)) = 1
    by simp
    hence  $\exists m. \text{obs } n (\text{backward-slice } S) = \{m\}$  by(fastforce dest:card-eq-SucD)
    thus ?thesis by simp
  qed
qed

lemma obs-singleton-element:
  assumes  $m \in \text{obs } n (\text{backward-slice } S)$  shows obs n (backward-slice S) =  $\{m\}$ 
proof -
  have  $(\exists m. \text{obs } n (\text{backward-slice } S) = \{m\}) \vee \text{obs } n (\text{backward-slice } S) = \{\}$ 
  by(rule obs-singleton-disj)
  with  $\langle m \in \text{obs } n (\text{backward-slice } S) \rangle$  show ?thesis by fastforce
qed

lemma obs-the-element:
   $m \in \text{obs } n (\text{backward-slice } S) \implies (\text{THE } m. m \in \text{obs } n (\text{backward-slice } S)) = m$ 
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* $:: 'node\ set \Rightarrow 'edge \Rightarrow 'state\ edge\text{-}kind$

where *slice-kind* S $a = (\text{let } S' = \text{backward-slice } S; n = \text{sourcenode } a \text{ in}$
(if *sourcenode* $a \in S'$ *then* *kind* a
else *(case* *kind* a *of* $\uparrow f \Rightarrow \uparrow id \mid (Q)_{\checkmark} \Rightarrow$
(if *obs* *(sourcenode* $a)$ $S' = \{\}$ *then*
(let $nx = (\text{SOME } n'. \exists a'. n = \text{sourcenode } a' \wedge \text{valid-edge } a' \wedge \text{targetnode } a'$
 $= n')$
in *(if* *(targetnode* $a = nx)$ *then* $(\lambda s. \text{True})_{\checkmark}$ *else* $(\lambda s. \text{False})_{\checkmark})$
else *(let* $m = \text{THE } m. m \in \text{obs } n \ S'$ *in*
(if $(\exists x. \text{distance } (\text{targetnode } a) \ m \ x \wedge \text{distance } n \ m \ (x + 1) \wedge$
 $(\text{targetnode } a = (\text{SOME } nx'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{distance } (\text{targetnode } a') \ m \ x \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = nx'))$
then $(\lambda s. \text{True})_{\checkmark}$ *else* $(\lambda s. \text{False})_{\checkmark}$
 $)$
 $)$
 $)$
 $)$

definition

slice-kinds $:: 'node\ set \Rightarrow 'edge\ list \Rightarrow 'state\ edge\text{-}kind\ list$

where *slice-kinds* S $as \equiv \text{map } (\text{slice-kind } S) \ as$

lemma *slice-kind-in-slice*:

sourcenode $a \in \text{backward-slice } S \implies \text{slice-kind } S \ a = \text{kind } a$

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

lemma *slice-kind-Upd*:

$\llbracket \text{sourcenode } a \notin \text{backward-slice } S; \text{kind } a = \uparrow f \rrbracket \implies \text{slice-kind } S \ a = \uparrow id$

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

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

$\llbracket \text{sourcenode } a \notin \text{backward-slice } S; \text{kind } a = (Q)_{\checkmark};$

$\text{obs } (\text{sourcenode } a) (\text{backward-slice } S) = \{\};$

$\text{targetnode } a = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a'$

$\wedge$

$\text{targetnode } a' = n') \rrbracket$

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

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

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

$\llbracket \text{sourcenode } a \notin \text{backward-slice } S; \text{kind } a = (Q)_{\checkmark};$

$obs \text{ (sourcenode } a) \text{ (backward-slice } S) = \{\};$
 $targetnode \ a \neq (SOME \ n'. \exists \ a'. \ sourcenode \ a = sourcenode \ a' \wedge \text{valid-edge } a')$
 $\wedge$
 $targetnode \ a' = n']$
 $\implies slice\text{-kind } S \ a = (\lambda s. \text{False})_{\checkmark}$
by(simp add:slice-kind-def)

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

assumes $sourcenode \ a \notin \text{backward-slice } S$ **and** $kind \ a = (Q)_{\checkmark}$
and $m \in obs \text{ (sourcenode } a) \text{ (backward-slice } S)$
and $distance \text{ (targetnode } a) \ m \ x \ distance \text{ (sourcenode } a) \ m \ (x + 1)$
and $targetnode \ a = (SOME \ n'. \exists \ a'. \ sourcenode \ a = sourcenode \ a' \wedge$
 $distance \text{ (targetnode } a') \ m \ x \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = n')$
shows $slice\text{-kind } S \ a = (\lambda s. \text{True})_{\checkmark}$
proof –
from $\langle m \in obs \text{ (sourcenode } a) \text{ (backward-slice } S) \rangle$
have $m = (THE \ m. \ m \in obs \text{ (sourcenode } a) \text{ (backward-slice } S))$
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 $sourcenode \ a \notin \text{backward-slice } S$ **and** $kind \ a = (Q)_{\checkmark}$
and $m \in obs \text{ (sourcenode } a) \text{ (backward-slice } S)$
and $distance \text{ (targetnode } a) \ m \ x \ distance \text{ (sourcenode } a) \ m \ (x + 1)$
and $targetnode \ a \neq (SOME \ nx'. \exists \ a'. \ sourcenode \ a = sourcenode \ a' \wedge$
 $distance \text{ (targetnode } a') \ m \ x \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = nx')$
shows $slice\text{-kind } S \ a = (\lambda s. \text{False})_{\checkmark}$
proof –
from $\langle m \in obs \text{ (sourcenode } a) \text{ (backward-slice } S) \rangle$
have $m = (THE \ m. \ m \in obs \text{ (sourcenode } a) \text{ (backward-slice } S))$
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 $sourcenode \ a \notin \text{backward-slice } S$ **and** $kind \ a = (Q)_{\checkmark}$
and $in\text{-obs}: m \in obs \text{ (sourcenode } a) \text{ (backward-slice } S)$
and $dist: distance \text{ (sourcenode } a) \ m \ (x + 1)$
 $\neg distance \text{ (targetnode } a) \ m \ x$
shows $slice\text{-kind } S \ a = (\lambda s. \text{False})_{\checkmark}$
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  $\neg (\exists x. \text{distance } (\text{targetnode } a) \ m \ x \wedge$ 
   $\text{distance } (\text{sourcenode } a) \ m \ (x + 1))$ 
  by(fastforce dest:distance-det)
with  $\langle \text{sourcenode } a \notin \text{backward-slice } S \rangle \langle \text{kind } a = (Q)_{\checkmark} \rangle$  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)✓ and sourcenode a ∉ backward-slice S
  obtains Q' where slice-kind S a = (Q')✓ and Q' = (λs. False) ∨ Q' = (λs. True)
proof(atomize-elim)
  show  $\exists Q'. \text{slice-kind } S \ a = (Q')_{\checkmark} \wedge (Q' = (\lambda s. \text{False}) \vee Q' = (\lambda s. \text{True}))$ 
  proof(cases obs (sourcenode a) (backward-slice S) = {})
    case True
      show ?thesis
      proof(cases targetnode a = (SOME n'. ∃ a'. sourcenode a = sourcenode a' ∧
        valid-edge a' ∧ targetnode a' = n'))
        case True
          with  $\langle \text{sourcenode } a \notin \text{backward-slice } S \rangle \langle \text{kind } a = (Q)_{\checkmark} \rangle$ 
             $\langle \text{obs } (\text{sourcenode } a) \ (\text{backward-slice } S) = \{\} \rangle$ 
          have slice-kind S a = (λs. True)✓ by(rule slice-kind-Pred-empty-obs-SOME)
          thus ?thesis by simp
        next
          case False
            with  $\langle \text{sourcenode } a \notin \text{backward-slice } S \rangle \langle \text{kind } a = (Q)_{\checkmark} \rangle$ 
               $\langle \text{obs } (\text{sourcenode } a) \ (\text{backward-slice } S) = \{\} \rangle$ 
            have slice-kind S a = (λs. False)✓
              by(rule slice-kind-Pred-empty-obs-not-SOME)
            thus ?thesis by simp
          qed
        next
          case False
            then obtain m where m ∈ obs (sourcenode a) (backward-slice S) by blast
            show ?thesis
            proof(cases ∃ x. distance (targetnode a) m x ∧
              distance (sourcenode a) m (x + 1))
              case True
                then obtain x where distance (targetnode a) m x
                  and distance (sourcenode a) m (x + 1) by blast
                show ?thesis
                proof(cases targetnode a = (SOME n'. ∃ a'. sourcenode a = sourcenode a' ∧
                  distance (targetnode a') m x ∧
                  valid-edge a' ∧ targetnode a' = n'))
                  case True
                    with  $\langle \text{sourcenode } a \notin \text{backward-slice } S \rangle \langle \text{kind } a = (Q)_{\checkmark} \rangle$ 

```

```

      ⟨m ∈ obs (sourcenode a) (backward-slice S)⟩
      ⟨distance (targetnode a) m x⟩ ⟨distance (sourcenode a) m (x + 1)⟩
    have slice-kind S a = (λs. True)✓
      by(rule slice-kind-Pred-obs-nearer-SOME)
    thus ?thesis by simp
  next
    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)⟩
    have slice-kind S a = (λs. False)✓
      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)✓⟩ False
    ⟨m ∈ obs (sourcenode a) (backward-slice S)⟩
  have slice-kind S a = (λs. False)✓
    by(fastforce simp:slice-kind-def Let-def)
  thus ?thesis by simp
qed
qed
qed
qed

```

lemma *only-one-SOME-edge*:

assumes *valid-edge a*

shows $\exists! a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a' \wedge$
 $\text{targetnode } a' = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = n')$

proof(rule *ex-ex1I*)

show $\exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a' \wedge$
 $\text{targetnode } a' = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = n')$

proof –

have $(\exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a' \wedge$
 $\text{targetnode } a' = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = n')) =$
 $(\exists n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a' \wedge \text{targetnode } a' = n')$
apply(*unfold some-eq-ex*[of $\lambda n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = n'$])

by *simp*

also have ... **using** *valid-edge a* **by** *blast*

finally show ?thesis .

qed

next

fix $a' \ ax$

assume $\text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a' \wedge$

$\text{targetnode } a' = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = n')$

and $\text{sourcenode } a = \text{sourcenode } ax \wedge \text{valid-edge } ax \wedge$

$\text{targetnode } ax = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = n')$

thus $a' = ax$ **by**(*fastforce intro!:edge-det*)

qed

lemma *slice-kind-only-one-True-edge*:

assumes $\text{sourcenode } a = \text{sourcenode } a'$ **and** $\text{targetnode } a \neq \text{targetnode } a'$

and $\text{valid-edge } a$ **and** $\text{valid-edge } a'$ **and** $\text{slice-kind } S \ a = (\lambda s. \text{True})_{\checkmark}$

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

proof –

from *assms* **obtain** $Q \ Q'$ **where** $\text{kind } a = (Q)_{\checkmark}$

and $\text{kind } a' = (Q')_{\checkmark}$ **and** $\text{det}:\forall s. (Q \ s \longrightarrow \neg Q' \ s) \wedge (Q' \ s \longrightarrow \neg Q \ s)$

by(*auto dest:deterministic*)

from $\langle \text{valid-edge } a \rangle$ **have** $\text{ex1}:\exists! a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a'$

$\wedge$

$\text{targetnode } a' = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = n')$

by(*rule only-one-SOME-edge*)

show *?thesis*

proof(*cases sourcenode a ∈ backward-slice S*)

case *True*

with $\langle \text{slice-kind } S \ a = (\lambda s. \text{True})_{\checkmark} \rangle$ $\langle \text{kind } a = (Q)_{\checkmark} \rangle$ **have** $Q = (\lambda s. \text{True})_{\checkmark}$

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

with *det* **have** $Q' = (\lambda s. \text{False})_{\checkmark}$ **by**(*simp add:fun-eq-iff*)

with *True* $\langle \text{kind } a' = (Q')_{\checkmark} \rangle$ $\langle \text{sourcenode } a = \text{sourcenode } a' \rangle$ **show** *?thesis*

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

next

case *False*

hence $\text{sourcenode } a \notin \text{backward-slice } S$ **by** *simp*

thus *?thesis*

proof(*cases obs (sourcenode a) (backward-slice S) = {}*)

case *True*

with $\langle \text{sourcenode } a \notin \text{backward-slice } S \rangle$ $\langle \text{slice-kind } S \ a = (\lambda s. \text{True})_{\checkmark} \rangle$

$\langle \text{kind } a = (Q)_{\checkmark} \rangle$

have $\text{target}:\text{targetnode } a = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = n')$

by(*auto simp:slice-kind-def Let-def fun-eq-iff split:split-if-asm*)

have $\text{targetnode } a' \neq (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = n')$

proof(*rule ccontr*)

assume $\neg \text{targetnode } a' \neq (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$

$\wedge$

$$\text{valid-edge } a' \wedge \text{targetnode } a' = n'$$
hence $\text{targetnode } a' = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a' \wedge \text{targetnode } a' = n')$

by *simp*
with $\langle \text{ex1 target } \langle \text{sourcenode } a = \text{sourcenode } a' \rangle \langle \text{valid-edge } a \rangle \langle \text{valid-edge } a' \rangle \text{ have } a = a' \text{ by blast} \rangle$
with $\langle \text{targetnode } a \neq \text{targetnode } a' \rangle$ **show** *False* **by** *simp*
qed

with $\langle \text{sourcenode } a \notin \text{backward-slice } S \rangle \text{ True } \langle \text{kind } a' = (Q')_{\checkmark} \rangle$
 $\langle \text{sourcenode } a = \text{sourcenode } a' \rangle$ **show** *?thesis*
by $(\text{auto simp:slice-kind-def Let-def fun-eq-iff split:split-if-asm})$

next

case *False*
hence $\text{obs } (\text{sourcenode } a) (\text{backward-slice } S) \neq \{\}$.
then obtain m **where** $m \in \text{obs } (\text{sourcenode } a) (\text{backward-slice } S)$ **by** *auto*
hence $m = (\text{THE } m. m \in \text{obs } (\text{sourcenode } a) (\text{backward-slice } S))$
by $(\text{auto dest:obs-the-element})$
with $\langle \text{sourcenode } a \notin \text{backward-slice } S \rangle$
 $\langle \text{obs } (\text{sourcenode } a) (\text{backward-slice } S) \neq \{\} \rangle$
 $\langle \text{slice-kind } S \ a = (\lambda s. \text{True})_{\checkmark} \rangle \langle \text{kind } a = (Q)_{\checkmark} \rangle$
obtain $x \ x'$ **where** $\text{distance } (\text{targetnode } a) \ m \ x$
 $\text{distance } (\text{sourcenode } a) \ m \ (x + 1)$
and $\text{target:targetnode } a = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{distance } (\text{targetnode } a') \ m \ x \wedge \text{valid-edge } a' \wedge \text{targetnode } a' = n')$

by $(\text{auto simp:slice-kind-def Let-def fun-eq-iff split:split-if-asm})$
show *?thesis*

proof $(\text{cases } \text{distance } (\text{targetnode } a') \ m \ x)$
case *False*
with $\langle \text{sourcenode } a \notin \text{backward-slice } S \rangle \langle \text{kind } a' = (Q')_{\checkmark} \rangle$
 $\langle m \in \text{obs } (\text{sourcenode } a) (\text{backward-slice } S) \rangle$
 $\langle \text{distance } (\text{targetnode } a) \ m \ x \rangle \langle \text{distance } (\text{sourcenode } a) \ m \ (x + 1) \rangle$
 $\langle \text{sourcenode } a = \text{sourcenode } a' \rangle$ **show** *?thesis*
by $(\text{fastforce intro:slice-kind-Pred-obs-not-nearer})$

next

case *True*
from $\langle \text{valid-edge } a \rangle \langle \text{distance } (\text{targetnode } a) \ m \ x \rangle$
 $\langle \text{distance } (\text{sourcenode } a) \ m \ (x + 1) \rangle$
have $\text{ex1:} \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{distance } (\text{targetnode } a') \ m \ x \wedge \text{valid-edge } a' \wedge \text{targetnode } a' = (\text{SOME } nx. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{distance } (\text{targetnode } a') \ m \ x \wedge \text{valid-edge } a' \wedge \text{targetnode } a' = nx)$

by $(\text{fastforce intro!:only-one-SOME-dist-edge})$
have $\text{targetnode } a' \neq (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{distance } (\text{targetnode } a') \ m \ x \wedge \text{valid-edge } a' \wedge \text{targetnode } a' = n')$

proof (rule ccontr)
assume $\neg \text{targetnode } a' \neq (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a')$

^

$$\text{distance } (\text{targetnode } a') \ m \ x \wedge$$

$$\text{valid-edge } a' \wedge \text{targetnode } a' = n'$$
hence $\text{targetnode } a' = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$

$$\text{distance } (\text{targetnode } a') \ m \ x \wedge$$

$$\text{valid-edge } a' \wedge \text{targetnode } a' = n')$$
by *simp*
with *ex1 target* $\langle \text{sourcenode } a = \text{sourcenode } a' \rangle$
 $\langle \text{valid-edge } a \rangle \langle \text{valid-edge } a' \rangle$
 $\langle \text{distance } (\text{targetnode } a) \ m \ x \rangle \langle \text{distance } (\text{sourcenode } a) \ m \ (x + 1) \rangle$
have $a = a'$ **by** *auto*
with $\langle \text{targetnode } a \neq \text{targetnode } a' \rangle$ **show** *False* **by** *simp*
qed
with $\langle \text{sourcenode } a \notin \text{backward-slice } S \rangle$
 $\langle \text{kind } a' = (Q')_{\checkmark} \rangle \langle m \in \text{obs } (\text{sourcenode } a) \ (\text{backward-slice } S) \rangle$
 $\langle \text{distance } (\text{targetnode } a) \ m \ x \rangle \langle \text{distance } (\text{sourcenode } a) \ m \ (x + 1) \rangle$
 $\text{True } \langle \text{sourcenode } a = \text{sourcenode } a' \rangle$ **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 $\text{sourcenode } a = \text{sourcenode } a'$ **and** $\text{targetnode } a \neq \text{targetnode } a'$
obtains $Q \ Q'$ **where** $\text{slice-kind } S \ a = (Q)_{\checkmark}$ **and** $\text{slice-kind } S \ a' = (Q')_{\checkmark}$
and $\forall s. (Q \ s \longrightarrow \neg Q' \ s) \wedge (Q' \ s \longrightarrow \neg Q \ s)$
proof (*atomize-elim*)
from *assms* **obtain** $Q \ Q'$
where $\text{kind } a = (Q)_{\checkmark}$ **and** $\text{kind } a' = (Q')_{\checkmark}$
and $\text{det}:\forall s. (Q \ s \longrightarrow \neg Q' \ s) \wedge (Q' \ s \longrightarrow \neg Q \ s)$
by (*auto dest:deterministic*)
from $\langle \text{valid-edge } a \rangle$ **have** *ex1*: $\exists ! a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a'$
^

$$\text{targetnode } a' = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$$

$$\text{valid-edge } a' \wedge \text{targetnode } a' = n')$$
by (*rule only-one-SOME-edge*)
show $\exists Q \ Q'. \text{slice-kind } S \ a = (Q)_{\checkmark} \wedge \text{slice-kind } S \ a' = (Q')_{\checkmark} \wedge$

$$(\forall s. (Q \ s \longrightarrow \neg Q' \ s) \wedge (Q' \ s \longrightarrow \neg Q \ s))$$
proof (*cases sourcenode a ∈ backward-slice S*)
case *True*
with $\langle \text{kind } a = (Q)_{\checkmark} \rangle$ **have** $\text{slice-kind } S \ a = (Q)_{\checkmark}$
by (*simp add:slice-kind-def Let-def*)
from *True* $\langle \text{kind } a' = (Q')_{\checkmark} \rangle \langle \text{sourcenode } a = \text{sourcenode } a' \rangle$
have $\text{slice-kind } S \ a' = (Q')_{\checkmark}$
by (*simp add:slice-kind-def Let-def*)
with $\langle \text{slice-kind } S \ a = (Q)_{\checkmark} \rangle$ **det** **show** *?thesis* **by** *blast*

```

next
  case False
  with  $\langle \text{kind } a = (Q)_{\checkmark} \rangle$ 
  have  $\text{slice-kind } S \ a = (\lambda s. \text{True})_{\checkmark} \vee \text{slice-kind } S \ a = (\lambda s. \text{False})_{\checkmark}$ 
    by(simp add:slice-kind-def Let-def)
  thus ?thesis
  proof
    assume  $\text{true:slice-kind } S \ a = (\lambda s. \text{True})_{\checkmark}$ 
    with  $\langle \text{sourcenode } a = \text{sourcenode } a' \rangle \langle \text{targetnode } a \neq \text{targetnode } a' \rangle$ 
       $\langle \text{valid-edge } a \rangle \langle \text{valid-edge } a' \rangle$ 
    have  $\text{slice-kind } S \ a' = (\lambda s. \text{False})_{\checkmark}$ 
      by(rule slice-kind-only-one-True-edge)
    with true show ?thesis by simp
  next
    assume  $\text{false:slice-kind } S \ a = (\lambda s. \text{False})_{\checkmark}$ 
    from False  $\langle \text{kind } a' = (Q')_{\checkmark} \rangle \langle \text{sourcenode } a = \text{sourcenode } a' \rangle$ 
    have  $\text{slice-kind } S \ a' = (\lambda s. \text{True})_{\checkmark} \vee \text{slice-kind } S \ a' = (\lambda s. \text{False})_{\checkmark}$ 
      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* ::

$$'node \text{ set} \Rightarrow ('edge \Rightarrow 'state \text{ edge-kind}) \Rightarrow 'node \Rightarrow 'state \Rightarrow 'edge \Rightarrow 'node \Rightarrow 'state \Rightarrow \text{bool } (-, - \vdash '(-, -) \dashrightarrow_{\tau} '(-, -)) \ [51, 50, 0, 0, 50, 0, 0] \ 51)$$

where *silent-moveI*:

$$\llbracket \text{pred } (f \ a) \ s; \text{transfer } (f \ a) \ s = s'; \text{sourcenode } a \notin \text{backward-slice } S; \text{valid-edge } a \rrbracket \\ \Rightarrow S, f \vdash (\text{sourcenode } a, s) \dashrightarrow_{\tau} (\text{targetnode } a, s')$$

inductive *silent-moves* ::

$$'node \text{ set} \Rightarrow ('edge \Rightarrow 'state \text{ edge-kind}) \Rightarrow 'node \Rightarrow 'state \Rightarrow 'edge \text{ list} \Rightarrow 'node \Rightarrow 'state \Rightarrow \text{bool } (-, - \vdash '(-, -) \dashrightarrow_{\tau} '(-, -)) \ [51, 50, 0, 0, 50, 0, 0] \ 51)$$

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

| *silent-moves-Cons*:

$$\llbracket S, f \vdash (n, s) \dashrightarrow_{\tau} (n', s'); S, f \vdash (n', s') =_{as \Rightarrow_{\tau}} (n'', s'') \rrbracket \\ \Rightarrow S, f \vdash (n, s) =_{a \# as \Rightarrow_{\tau}} (n'', s'')$$

lemma *silent-moves-obs-slice*:

$$\llbracket S, f \vdash (n, s) =_{as \Rightarrow_{\tau}} (n', s'); nx \in \text{obs } n' (\text{backward-slice } S) \rrbracket \\ \Rightarrow nx \in \text{obs } n (\text{backward-slice } S)$$

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 $\langle nx \in \text{obs } n'' \text{ (backward-slice } S) \rangle$
 $\langle nx \in \text{obs } n'' \text{ (backward-slice } S) \implies nx \in \text{obs } n' \text{ (backward-slice } S) \rangle$
have $\text{obs}: nx \in \text{obs } n' \text{ (backward-slice } S)$ **by** *simp*
 from $\langle S, f \vdash (n, s) -a \rightarrow_{\tau} (n', s') \rangle$
have $n = \text{sourcenode } a$ **and** $n' = \text{targetnode } a$ **and** *valid-edge* a
and $n \notin \text{(backward-slice } S)$
by(*auto elim:silent-move.cases*)
hence $\text{obs } n' \text{ (backward-slice } S) \subseteq \text{obs } n \text{ (backward-slice } S)$
by *simp*(*rule edge-obs-subset, simp+*)
with *obs* **show** ?case **by** *blast*
qed

lemma *silent-moves-preds-transfers-path*:
 $\llbracket S, f \vdash (n, s) =_{as \Rightarrow_{\tau}} (n', s') ; \text{valid-node } n \rrbracket$
 $\implies \text{preds } (\text{map } f \text{ as}) s \wedge \text{transfers } (\text{map } f \text{ as}) s = s' \wedge n -as \rightarrow^* n'$
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 $IH = \langle \text{valid-node } n' \implies$
 $\text{preds } (\text{map } f \text{ as}) s' \wedge \text{transfers } (\text{map } f \text{ as}) s' = s'' \wedge n' -as \rightarrow^* n'' \rangle$
from $\langle S, f \vdash (n, s) -a \rightarrow_{\tau} (n', s') \rangle$ **have** *pred* ($f a$) s **and** *transfer* ($f a$) $s = s'$
and $n = \text{sourcenode } a$ **and** $n' = \text{targetnode } a$ **and** *valid-edge* a
by(*auto elim:silent-move.cases*)
from $\langle n' = \text{targetnode } a \rangle \langle \text{valid-edge } a \rangle$ **have** *valid-node* n' **by** *simp*
from $IH[OF \text{ this}]$ **have** $\text{preds } (\text{map } f \text{ as}) s'$ **and** $\text{transfers } (\text{map } f \text{ as}) s' = s''$
and $n' -as \rightarrow^* n''$ **by** *simp-all*
from $\langle n = \text{sourcenode } a \rangle \langle n' = \text{targetnode } a \rangle \langle \text{valid-edge } a \rangle \langle n' -as \rightarrow^* n'' \rangle$
have $n -a \# as \rightarrow^* n''$ **by**(*fastforce intro:Cons-path*)
with $\langle \text{pred } (f a) s \rangle \langle \text{preds } (\text{map } f \text{ as}) s' \rangle \langle \text{transfer } (f a) s = s' \rangle$
 $\langle \text{transfers } (\text{map } f \text{ as}) s' = s'' \rangle$ **show** ?case **by** *simp*
qed

lemma *obs-silent-moves*:
assumes $\text{obs } n \text{ (backward-slice } S) = \{n'\}$
obtains *as* **where** $S, \text{slice-kind } S \vdash (n, s) =_{as \Rightarrow_{\tau}} (n', s)$
proof(*atomize-elim*)
from $\langle \text{obs } n \text{ (backward-slice } S) = \{n'\} \rangle$
have $n' \in \text{obs } n \text{ (backward-slice } S)$ **by** *simp*
then obtain *as* **where** $n -as \rightarrow^* n'$
and $\forall nx \in \text{set}(\text{sourcenodes } as). nx \notin \text{(backward-slice } S)$
and $n' \in \text{(backward-slice } S)$ **by**(*erule obsE*)
from $\langle n -as \rightarrow^* n' \rangle$ **obtain** x **where** *distance* $n n' x$ **and** $x \leq \text{length } as$

```

  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 ⇒τ (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) = [] ⇒τ (n', s) by(fastforce intro:silent-moves-Nil)
  thus ∃ as. S, slice-kind S ⊢ (n, s) = as ⇒τ (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 ⇒τ (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: $S, \text{slice-kind } S \vdash (\text{targetnode } a, \text{transfer } (\text{slice-kind } S \ a) \ s) = asx' \Rightarrow_{\tau}$
 $(n', \text{transfer } (\text{slice-kind } S \ a) \ s)$ **by** *blast*
have $\text{pred } (\text{slice-kind } S \ a) \ s \wedge \text{transfer } (\text{slice-kind } S \ a) \ s = s$
proof(*cases kind a*)
case (*Update f*)
with $\langle n \notin \text{backward-slice } S \rangle \langle n = \text{sourcenode } a \rangle$ **have** $\text{slice-kind } S \ a = \uparrow id$
by(*fastforce intro:slice-kind-Upd*)
thus *?thesis* **by** *simp*
next
case (*Predicate Q*)
with $\langle n \notin \text{backward-slice } S \rangle \langle n = \text{sourcenode } a \rangle$
 $\langle n' \in \text{obs } n \ (\text{backward-slice } S) \rangle \langle \text{distance } (\text{targetnode } a) \ n' \ x \rangle$
 $\langle \text{distance } n \ n' \ (\text{Suc } x) \rangle \text{target}$
have $\text{slice-kind } S \ a = (\lambda s. \text{True})_{\checkmark}$
by(*fastforce intro:slice-kind-Pred-obs-nearer-SOME*)
thus *?thesis* **by** *simp*
qed
hence $\text{pred } (\text{slice-kind } S \ a) \ s$ **and** $\text{transfer } (\text{slice-kind } S \ a) \ s = s$
by *simp-all*
with $\langle n \notin \text{backward-slice } S \rangle \langle n = \text{sourcenode } a \rangle \langle \text{valid-edge } a \rangle$
have $S, \text{slice-kind } S \vdash (\text{sourcenode } a, s) -a \rightarrow_{\tau}$
 $(\text{targetnode } a, \text{transfer } (\text{slice-kind } S \ a) \ s)$
by(*fastforce intro:silent-moveI*)
with *moves* $\langle \text{transfer } (\text{slice-kind } S \ a) \ s = s \rangle \langle n = \text{sourcenode } a \rangle$
have $S, \text{slice-kind } S \vdash (n, s) = a \# asx' \Rightarrow_{\tau} (n', s)$
by(*fastforce intro:silent-moves-Cons*)
thus $\exists as. S, \text{slice-kind } S \vdash (n, s) = as \Rightarrow_{\tau} (n', s)$ **by** *blast*
qed
qed

inductive *observable-move* ::

$'node \text{ set} \Rightarrow ('edge \Rightarrow 'state \text{ edge-kind}) \Rightarrow 'node \Rightarrow 'state \Rightarrow 'edge \Rightarrow$
 $'node \Rightarrow 'state \Rightarrow \text{bool } (-, - \vdash '(-, -) \dashrightarrow '(-, -) \ [51, 50, 0, 0, 50, 0, 0] \ 51)$

where *observable-moveI*:

$\llbracket \text{pred } (f \ a) \ s; \text{transfer } (f \ a) \ s = s'; \text{sourcenode } a \in \text{backward-slice } S;$
 $\text{valid-edge } a \rrbracket$
 $\Rightarrow S, f \vdash (\text{sourcenode } a, s) -a \rightarrow (\text{targetnode } a, s')$

inductive *observable-moves* ::

$'node \text{ set} \Rightarrow ('edge \Rightarrow 'state \text{ edge-kind}) \Rightarrow 'node \Rightarrow 'state \Rightarrow 'edge \text{ list} \Rightarrow$
 $'node \Rightarrow 'state \Rightarrow \text{bool } (-, - \vdash '(-, -) \dashrightarrow '(-, -) \ [51, 50, 0, 0, 50, 0, 0] \ 51)$

where *observable-moves-snoc*:

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

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

lemma *silent-move-observable-moves*:
 $\llbracket S, f \vdash (n'', s'') = as \Rightarrow (n', s'); S, f \vdash (n, s) -a \rightarrow_\tau (n'', s'') \rrbracket$
 $\Rightarrow S, f \vdash (n, s) = a \# as \Rightarrow (n', s')$
proof(*induct rule:observable-moves.induct*)
case (*observable-moves-snoc* $S f nx sx as n' s' a' n'' s''$)
from $\langle S, f \vdash (n, s) -a \rightarrow_\tau (nx, sx) \rangle \langle S, f \vdash (nx, sx) = as \Rightarrow_\tau (n', s') \rangle$
have $S, f \vdash (n, s) = a \# as \Rightarrow_\tau (n', s')$ **by**(*rule silent-moves-Cons*)
with $\langle S, f \vdash (n', s') -a' \rightarrow (n'', s'') \rangle$
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*:
 $S, f \vdash (n, s) = as \Rightarrow (n', s')$
 $\Rightarrow preds (map f as) s \wedge transfers (map f as) s = s' \wedge n -as \rightarrow^* n'$
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 $\langle S, f \vdash (n, s) = as \Rightarrow_\tau (n', s') \rangle$ **have** $n = n'$ **and** $s = s'$
by(*auto elim:silent-moves.cases*)
with $\langle S, f \vdash (n', s') -a \rightarrow (n'', s'') \rangle$ **show** *?thesis*
by(*fastforce elim:observable-move.cases*)
next
case (*Cons* $a' as'$)
with $\langle S, f \vdash (n, s) = as \Rightarrow_\tau (n', s') \rangle$ **show** *?thesis*
by(*fastforce elim:silent-moves.cases silent-move.cases*)
qed
with $\langle S, f \vdash (n, s) = as \Rightarrow_\tau (n', s') \rangle$
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 $\langle S, f \vdash (n', s') -a \rightarrow (n'', s'') \rangle$ **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 $\langle n' = sourcenode a \rangle \langle n'' = targetnode a \rangle \langle valid-edge a \rangle$
have $n' -[a] \rightarrow^* n''$ **by**(*fastforce intro:path.intros*)
with $\langle n -as \rightarrow^* n' \rangle$ **have** $n -as @ [a] \rightarrow^* n''$ **by**(*rule path-Append*)
with $\langle preds (map f as) s \rangle \langle pred (f a) s' \rangle \langle transfer (f a) s' = s'' \rangle$

```

  <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  $\Rightarrow$  'node  $\Rightarrow$  'var set (rv -)
for S :: 'node set and n :: 'node

```

```

where rvI:
   $\llbracket n -as \rightarrow^* n'; n' \in \text{backward-slice } S; V \in \text{Use } n';$ 
   $\forall nx \in \text{set}(\text{sourcenodes } as). V \notin \text{Def } nx \rrbracket$ 
 $\implies V \in \text{rv } S \ n$ 

```

```

lemma rvE:
  assumes rv:  $V \in \text{rv } S \ n$ 
  obtains as n' where  $n -as \rightarrow^* n'$  and  $n' \in \text{backward-slice } S$  and  $V \in \text{Use } n'$ 
  and  $\forall nx \in \text{set}(\text{sourcenodes } as). V \notin \text{Def } nx$ 
using rv
by(atomize-elim, auto elim!:relevant-vars.cases)

```

```

lemma eq-obs-in-rv:
  assumes obs-eq:  $\text{obs } n (\text{backward-slice } S) = \text{obs } n' (\text{backward-slice } S)$ 
  and  $x \in \text{rv } S \ n$  shows  $x \in \text{rv } S \ n'$ 
proof -
  from  $\langle x \in \text{rv } S \ n \rangle$  obtain as m
    where  $n -as \rightarrow^* m$  and  $m \in \text{backward-slice } S$  and  $x \in \text{Use } m$ 
    and  $\forall nx \in \text{set}(\text{sourcenodes } as). x \notin \text{Def } nx$ 
    by(erule rvE)
  from  $\langle n -as \rightarrow^* m \rangle$  have valid-node m by(fastforce dest:path-valid-node)
  from  $\langle n -as \rightarrow^* m \rangle \langle m \in \text{backward-slice } S \rangle$ 
  have  $\exists nx \ as' \ as''. nx \in \text{obs } n (\text{backward-slice } S) \wedge n -as' \rightarrow^* nx \wedge$ 
     $nx -as'' \rightarrow^* m \wedge as = as' @ as''$ 
  proof(cases  $\forall nx \in \text{set}(\text{sourcenodes } as). nx \notin \text{backward-slice } S$ )
  case True
    with  $\langle n -as \rightarrow^* m \rangle \langle m \in \text{backward-slice } S \rangle$  have  $m \in \text{obs } n (\text{backward-slice } S)$ 
    by -(rule obs-elem)
    with  $\langle n -as \rightarrow^* m \rangle \langle \text{valid-node } m \rangle$  show ?thesis by(blast intro:empty-path)
  next
  case False
    hence  $\exists nx \in \text{set}(\text{sourcenodes } as). nx \in \text{backward-slice } S$  by simp
    then obtain nx' ns ns' where  $\text{sourcenodes } as = ns @ nx' \# ns'$ 
    and  $nx' \in \text{backward-slice } S$ 
    and  $\forall x \in \text{set } ns. x \notin \text{backward-slice } S$ 
    by(fastforce elim!:split-list-first-propE)
    from  $\langle \text{sourcenodes } as = ns @ nx' \# ns' \rangle$ 

```



```

obtain  $as' a' as''$  where  $ns = \text{sourcenodes } as'$ 
  and  $as = as' @ a' \# as''$  and  $\text{sourcenode } a' = nx'$ 
  by( $\text{fastforce elim:map-append-append-maps simp:sourcenodes-def}$ )
from  $\langle n - as \rightarrow^* m \rangle \langle as = as' @ a' \# as'' \rangle \langle \text{sourcenode } a' = nx' \rangle$ 
have  $n - as' \rightarrow^* nx'$  and  $\text{valid-edge } a'$  and  $\text{targetnode } a' - as'' \rightarrow^* m$ 
  by( $\text{fastforce dest:path-split}$ ) +
with  $\langle \text{sourcenode } a' = nx' \rangle$  have  $nx' - a' \# as'' \rightarrow^* m$  by( $\text{fastforce intro:Cons-path}$ )
from  $\langle n - as' \rightarrow^* nx' \rangle \langle nx' \in \text{backward-slice } S \rangle$ 
   $\langle \forall x \in \text{set } ns. x \notin \text{backward-slice } S \rangle \langle ns = \text{sourcenodes } as' \rangle$ 
have  $nx' \in \text{obs } n (\text{backward-slice } S)$ 
  by( $\text{fastforce intro:obs-elem}$ )
with  $\langle n - as' \rightarrow^* nx' \rangle \langle nx' - a' \# as'' \rightarrow^* m \rangle \langle as = as' @ a' \# as'' \rangle$  show  $?thesis$ 
by blast
qed
then obtain  $nx as' as''$  where  $nx \in \text{obs } n (\text{backward-slice } S)$ 
  and  $n - as' \rightarrow^* nx$  and  $nx - as'' \rightarrow^* m$  and  $as = as' @ as''$ 
  by blast
from  $\langle nx \in \text{obs } n (\text{backward-slice } S) \rangle \text{obs-eq}$ 
have  $nx \in \text{obs } n' (\text{backward-slice } S)$  by auto
then obtain  $asx$  where  $n' - asx \rightarrow^* nx$ 
  and  $\forall ni \in \text{set}(\text{sourcenodes } asx). ni \notin \text{backward-slice } S$ 
  and  $nx \in \text{backward-slice } S$ 
  by( $\text{erule obsE}$ )
from  $\langle as = as' @ as'' \rangle \langle \forall nx \in \text{set}(\text{sourcenodes } as). x \notin \text{Def } nx \rangle$ 
have  $\forall ni \in \text{set}(\text{sourcenodes } as''). x \notin \text{Def } ni$ 
  by( $\text{auto simp:sourcenodes-def}$ )
from  $\langle \forall ni \in \text{set}(\text{sourcenodes } asx). ni \notin \text{backward-slice } S \rangle \langle n' - asx \rightarrow^* nx \rangle$ 
have  $\forall ni \in \text{set}(\text{sourcenodes } asx). x \notin \text{Def } ni$ 
proof( $\text{induct asx arbitrary:n'}$ )
  case Nil thus  $?case$  by( $\text{simp add:sourcenodes-def}$ )
next
  case ( $\text{Cons } ax' asx'$ )
  note  $IH = \langle \bigwedge n'. [\forall ni \in \text{set}(\text{sourcenodes } asx'). ni \notin \text{backward-slice } S;$ 
     $n' - asx' \rightarrow^* nx] \implies \forall ni \in \text{set}(\text{sourcenodes } asx'). x \notin \text{Def } ni \rangle$ 
  from  $\langle n' - ax' \# asx' \rightarrow^* nx \rangle$  have  $n' - [] @ ax' \# asx' \rightarrow^* nx$  by simp
  hence  $\text{targetnode } ax' - asx' \rightarrow^* nx$  and  $n' = \text{sourcenode } ax'$ 
    by( $\text{fastforce dest:path-split}$ ) +
  from  $\langle \forall ni \in \text{set}(\text{sourcenodes } (ax' \# asx')). ni \notin \text{backward-slice } S \rangle$ 
  have  $\text{all} : \forall ni \in \text{set}(\text{sourcenodes } asx'). ni \notin \text{backward-slice } S$ 
    and  $\text{sourcenode } ax' \notin \text{backward-slice } S$ 
    by( $\text{auto simp:sourcenodes-def}$ )
  from  $IH[OF \text{ all } (\text{targetnode } ax' - asx' \rightarrow^* nx)]$ 
  have  $\forall ni \in \text{set}(\text{sourcenodes } asx'). x \notin \text{Def } ni$  .
  with  $\langle \forall ni \in \text{set}(\text{sourcenodes } as''). x \notin \text{Def } ni \rangle$ 
  have  $\forall ni \in \text{set}(\text{sourcenodes } (asx' @ as'')). x \notin \text{Def } ni$ 
    by( $\text{auto simp:sourcenodes-def}$ )
  from  $\langle n' - ax' \# asx' \rightarrow^* nx \rangle \langle nx - as'' \rightarrow^* m \rangle$  have  $n' - (ax' \# asx') @ as'' \rightarrow^* m$ 
    by-( $\text{rule path-Append}$ )

```

hence $n' - ax' \# asx' @ as'' \rightarrow^* m$ **by** *simp*
 have $x \notin \text{Def } (\text{sourcenode } ax')$
proof
 assume $x \in \text{Def } (\text{sourcenode } ax')$
 with $\langle x \in \text{Use } m \rangle \langle \forall ni \in \text{set } (\text{sourcenodes } (asx' @ as'')). x \notin \text{Def } ni \rangle$
 $\langle n' - ax' \# asx' @ as'' \rightarrow^* m \rangle \langle n' = \text{sourcenode } ax' \rangle$
 have n' influences x in m
 by *(auto simp: data-dependence-def)*
 with $\langle m \in \text{backward-slice } S \rangle \text{dd-closed}$ have $n' \in \text{backward-slice } S$
 by *(auto simp: dd-closed)*
 with $\langle n' = \text{sourcenode } ax' \rangle \langle \text{sourcenode } ax' \notin \text{backward-slice } S \rangle$
 show *False* **by** *simp*
qed
 with $\langle \forall ni \in \text{set } (\text{sourcenodes } (asx' @ as'')). x \notin \text{Def } ni \rangle$
 show ?case **by** *(simp add: sourcenodes-def)*
qed
 with $\langle \forall ni \in \text{set } (\text{sourcenodes } as''). x \notin \text{Def } ni \rangle$
 have $\forall ni \in \text{set } (\text{sourcenodes } (asx @ as'')). x \notin \text{Def } ni$
 by *(auto simp: sourcenodes-def)*
 from $\langle n' - asx \rightarrow^* nx \rangle \langle nx - as'' \rightarrow^* m \rangle$ have $n' - asx @ as'' \rightarrow^* m$ **by** *(rule path-Append)*
 with $\langle m \in \text{backward-slice } S \rangle \langle x \in \text{Use } m \rangle$
 $\langle \forall ni \in \text{set } (\text{sourcenodes } (asx @ as'')). x \notin \text{Def } ni \rangle$ show $x \in \text{rv } S \ n'$ **by** *-(rule rvI)*
qed

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

fixes $S :: 'node \text{ set}$
 assumes *valid-node* n **and** *valid-node* n'
 and *obs-eq*: $\text{obs } n (\text{backward-slice } S) = \text{obs } n' (\text{backward-slice } S)$
 shows $\text{rv } S \ n = \text{rv } S \ n'$
proof
 show $\text{rv } S \ n \subseteq \text{rv } S \ n'$
proof
 fix x assume $x \in \text{rv } S \ n$
 with $\langle \text{valid-node } n \rangle \text{obs-eq}$ show $x \in \text{rv } S \ n'$ **by** *-(rule eq-obs-in-rv)*
qed
next
 show $\text{rv } S \ n' \subseteq \text{rv } S \ n$
proof
 fix x assume $x \in \text{rv } S \ n'$
 with $\langle \text{valid-node } n' \rangle \text{obs-eq}$ [THEN *sym*] show $x \in \text{rv } S \ n$ **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 $\forall V \in \text{rv } S \ n. \text{state-val } s \ V = \text{state-val } s' \ V$

and $\text{preds } (\text{slice-kinds } S \ (a\#as)) \ s$ and $\text{preds } (\text{slice-kinds } S \ (a\#asx)) \ s'$
 shows $\forall V \in \text{rv } S \ n''. \text{state-val } (\text{transfer } (\text{slice-kind } S \ a) \ s) \ V =$
 $\text{state-val } (\text{transfer } (\text{slice-kind } S \ a) \ s') \ V$

proof

fix V **assume** $V \in \text{rv } S \ n''$

show $\text{state-val } (\text{transfer } (\text{slice-kind } S \ a) \ s) \ V =$
 $\text{state-val } (\text{transfer } (\text{slice-kind } S \ a) \ s') \ V$

proof($\text{cases } V \in \text{Def } n$)

case True

show $?thesis$

proof($\text{cases } \text{sourcenode } a \in \text{backward-slice } S$)

case True

hence $\text{slice-kind } S \ a = \text{kind } a$ **by**($\text{rule slice-kind-in-slice}$)

with $\langle \text{preds } (\text{slice-kinds } S \ (a\#as)) \ s \rangle$ **have** $\text{pred } (\text{kind } a) \ s$

by($\text{simp add:slice-kinds-def}$)

from $\langle \text{slice-kind } S \ a = \text{kind } a \rangle \langle \text{preds } (\text{slice-kinds } S \ (a\#asx)) \ s' \rangle$

have $\text{pred } (\text{kind } a) \ s'$

by($\text{simp add:slice-kinds-def}$)

from $\langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = n \rangle$ **have** $n \rightarrow^* n$

by($\text{fastforce intro:empty-path}$)

with $\text{True } \langle \text{sourcenode } a = n \rangle$ **have** $\forall V \in \text{Use } n. V \in \text{rv } S \ n$

by($\text{fastforce intro:rvI simp:sourcenodes-def}$)

with $\langle \forall V \in \text{rv } S \ n. \text{state-val } s \ V = \text{state-val } s' \ V \rangle \langle \text{sourcenode } a = n \rangle$

have $\forall V \in \text{Use } (\text{sourcenode } a). \text{state-val } s \ V = \text{state-val } s' \ V$ **by** blast

from $\langle \text{valid-edge } a \rangle$ **this** $\langle \text{pred } (\text{kind } a) \ s \rangle \langle \text{pred } (\text{kind } a) \ s' \rangle$

have $\forall V \in \text{Def } (\text{sourcenode } a). \text{state-val } (\text{transfer } (\text{kind } a) \ s) \ V =$
 $\text{state-val } (\text{transfer } (\text{kind } a) \ s') \ V$

by($\text{rule CFG-edge-transfer-uses-only-Use}$)

with $\langle V \in \text{Def } n \rangle \langle \text{sourcenode } a = n \rangle \langle \text{slice-kind } S \ a = \text{kind } a \rangle$

show $?thesis$ **by** simp

next

case False

from $\langle V \in \text{rv } S \ n'' \rangle$ **obtain** $xs \ nx$ **where** $n'' \rightarrow^* xs \rightarrow^* nx$

and $nx \in \text{backward-slice } S$ **and** $V \in \text{Use } nx$

and $\forall nx' \in \text{set}(\text{sourcenodes } xs). V \notin \text{Def } nx'$ **by**(erule rvE)

from $\langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = n \rangle \langle \text{targetnode } a = n'' \rangle$

$\langle n'' \rightarrow^* xs \rightarrow^* nx \rangle$

have $n \rightarrow^* a\#xs \rightarrow^* nx$ **by** $\text{-(rule path.Cons-path)}$

with $\langle V \in \text{Def } n \rangle \langle V \in \text{Use } nx \rangle \langle \forall nx' \in \text{set}(\text{sourcenodes } xs). V \notin \text{Def } nx' \rangle$

have n **influences** V **in** nx **by**($\text{fastforce simp:data-dependence-def}$)

with $\langle nx \in \text{backward-slice } S \rangle$ **have** $n \in \text{backward-slice } S$

by(rule dd-closed)

with $\langle \text{sourcenode } a = n \rangle$ False **have** False **by** simp

thus $?thesis$ **by** simp

qed

next

case False

from $\langle V \in \text{rv } S \ n'' \rangle$ **obtain** $xs \ nx$ **where** $n'' \rightarrow^* xs \rightarrow^* nx$

and $nx \in \text{backward-slice } S$ **and** $V \in \text{Use } nx$

```

    and  $\forall nx' \in \text{set}(\text{sourcenodes } xs). V \notin \text{Def } nx'$  by (erule rvE)
  from  $\langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = n \rangle \langle \text{targetnode } a = n'' \rangle \langle n'' - xs \rightarrow^* nx \rangle$ 
  have  $n - a \# xs \rightarrow^* nx$  by  $-(\text{rule path.Cons-path})$ 
  from  $\text{False} \langle \forall nx' \in \text{set}(\text{sourcenodes } xs). V \notin \text{Def } nx' \rangle \langle \text{sourcenode } a = n \rangle$ 
  have  $\forall nx' \in \text{set}(\text{sourcenodes } (a \# xs)). V \notin \text{Def } nx'$ 
    by (simp add: sourcenodes-def)
  with  $\langle n - a \# xs \rightarrow^* nx \rangle \langle nx \in \text{backward-slice } S \rangle \langle V \in \text{Use } nx \rangle$ 
  have  $V \in \text{rv } S \ n$  by (rule rvI)
  show ?thesis
proof (cases kind a)
  case (Predicate Q)
  show ?thesis
proof (cases sourcenode a  $\in$  backward-slice S)
  case True
  with Predicate have slice-kind S a = (Q)✓
    by (simp add: slice-kind-in-slice)
  with  $\langle \forall V \in \text{rv } S \ n. \text{state-val } s \ V = \text{state-val } s' \ V \rangle \langle V \in \text{rv } S \ n \rangle$ 
  show ?thesis by simp
next
  case False
  with Predicate obtain Q' where slice-kind S a = (Q')✓
    by  $-(\text{erule kind-Predicate-notin-slice-slice-kind-Predicate})$ 
  with  $\langle \forall V \in \text{rv } S \ n. \text{state-val } s \ V = \text{state-val } s' \ V \rangle \langle V \in \text{rv } S \ n \rangle$ 
  show ?thesis by simp
qed
next
case (Update f)
show ?thesis
proof (cases sourcenode a  $\in$  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  $\langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = n \rangle \langle V \notin \text{Def } n \rangle$ 
  have  $\text{state-val } (\text{transfer } (\text{kind } a) \ s) \ V = \text{state-val } s \ V$ 
    by (fastforce intro: CFG-edge-no-Def-equal)
  from Update have pred (kind a) s' by simp
  with  $\langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = n \rangle \langle V \notin \text{Def } n \rangle$ 
  have  $\text{state-val } (\text{transfer } (\text{kind } a) \ s') \ V = \text{state-val } s' \ V$ 
    by (fastforce intro: CFG-edge-no-Def-equal)
  with  $\langle \forall V \in \text{rv } S \ n. \text{state-val } s \ V = \text{state-val } s' \ V \rangle \langle V \in \text{rv } S \ n \rangle$ 
     $\langle \text{state-val } (\text{transfer } (\text{kind } a) \ s) \ V = \text{state-val } s \ V \rangle$ 
     $\langle \text{slice-kind } S \ a = \text{kind } a \rangle$ 
  show ?thesis by fastforce
next
  case False
  with Update have slice-kind S a =  $\uparrow \text{id}$  by  $-(\text{rule slice-kind-Upd})$ 
  with  $\langle \forall V \in \text{rv } S \ n. \text{state-val } s \ V = \text{state-val } s' \ V \rangle \langle V \in \text{rv } S \ n \rangle$ 
  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 $\forall V \in rv\ S\ n. \text{state-val } s\ V = \text{state-val } s'\ V$
shows *False*

proof –

from $\langle \text{valid-edge } a \rangle \langle \text{valid-edge } ax \rangle \langle \text{sourcenode } a = n \rangle \langle \text{sourcenode } ax = n \rangle$
 $\langle \text{targetnode } a = n'' \rangle \langle \text{targetnode } ax \neq n'' \rangle$
obtain $Q\ Q'$ **where** $\text{kind } a = (Q)_{\checkmark}$ **and** $\text{kind } ax = (Q')_{\checkmark}$
and $\forall s. (Q\ s \longrightarrow \neg Q'\ s) \wedge (Q'\ s \longrightarrow \neg Q\ s)$
by (*auto dest:deterministic*)
from $\langle \text{valid-edge } a \rangle \langle \text{valid-edge } ax \rangle \langle \text{sourcenode } a = n \rangle \langle \text{sourcenode } ax = n \rangle$
 $\langle \text{targetnode } a = n'' \rangle \langle \text{targetnode } ax \neq n'' \rangle$
obtain $P\ P'$ **where** $\text{slice-kind } S\ a = (P)_{\checkmark}$
and $\text{slice-kind } S\ ax = (P')_{\checkmark}$
and $\forall s. (P\ s \longrightarrow \neg P'\ s) \wedge (P'\ s \longrightarrow \neg P\ s)$
by –(*erule slice-deterministic,auto*)
show ?thesis

proof (*cases sourcenode a ∈ backward-slice S*)

case *True*

hence $\text{slice-kind } S\ a = \text{kind } a$ **by** (*rule slice-kind-in-slice*)

with $\langle \text{preds (slice-kinds } S\ (a\#as))\ s \rangle \langle \text{kind } a = (Q)_{\checkmark} \rangle$

$\langle \text{slice-kind } S\ a = (P)_{\checkmark} \rangle$ **have** $\text{pred (kind } a)\ s$

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

from *True* $\langle \text{sourcenode } a = n \rangle \langle \text{sourcenode } ax = n \rangle$

have $\text{slice-kind } S\ ax = \text{kind } ax$ **by** (*fastforce simp:slice-kind-in-slice*)

with $\langle \text{preds (slice-kinds } S\ (ax\#asx))\ s' \rangle \langle \text{kind } ax = (Q')_{\checkmark} \rangle$

$\langle \text{slice-kind } S\ ax = (P')_{\checkmark} \rangle$ **have** $\text{pred (kind } ax)\ s'$

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

with $\langle \text{kind } ax = (Q')_{\checkmark} \rangle$ **have** $Q'\ s'$ **by** *simp*

from $\langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = n \rangle$ **have** $n \dashv\!\!\rightarrow^* n$

by (*fastforce intro:empty-path*)

with *True* $\langle \text{sourcenode } a = n \rangle$ **have** $\forall V \in \text{Use } n. V \in rv\ S\ n$

by (*fastforce intro:rvI simp:sourcenodes-def*)

with $\langle \forall V \in rv\ S\ n. \text{state-val } s\ V = \text{state-val } s'\ V \rangle \langle \text{sourcenode } a = n \rangle$

have $\forall V \in \text{Use (sourcenode } a). \text{state-val } s\ V = \text{state-val } s'\ V$ **by** *blast*

with $\langle \text{valid-edge } a \rangle \langle \text{pred (kind } a)\ s \rangle$ **have** $\text{pred (kind } a)\ s'$

by (*rule CFG-edge-Uses-pred-equal*)

with $\langle \text{kind } a = (Q)_{\checkmark} \rangle$ **have** $Q\ s'$ **by** *simp*

with $\langle Q'\ s' \rangle \langle \forall s. (Q\ s \longrightarrow \neg Q'\ s) \wedge (Q'\ s \longrightarrow \neg Q\ s) \rangle$ **have** *False* **by** *simp*

thus ?thesis **by** *simp*

```

next
  case False
  with  $\langle \text{kind } a = (Q)_{\checkmark} \rangle \langle \text{slice-kind } S \ a = (P)_{\checkmark} \rangle$ 
  have  $P = (\lambda s. \text{False}) \vee P = (\lambda s. \text{True})$ 
    by(fastforce elim:kind-Predicate-notin-slice-slice-kind-Predicate)
  with  $\langle \text{slice-kind } S \ a = (P)_{\checkmark} \rangle \langle \text{preds } (\text{slice-kinds } S \ (a \# as)) \ s \rangle$ 
  have  $P = (\lambda s. \text{True})$  by(fastforce simp:slice-kinds-def)
  from  $\langle \text{kind } ax = (Q')_{\checkmark} \rangle \langle \text{slice-kind } S \ ax = (P')_{\checkmark} \rangle$ 
     $\langle \text{sourcenode } a = n \rangle \langle \text{sourcenode } ax = n \rangle \text{False}$ 
  have  $P' = (\lambda s. \text{False}) \vee P' = (\lambda s. \text{True})$ 
    by(fastforce elim:kind-Predicate-notin-slice-slice-kind-Predicate)
  with  $\langle \text{slice-kind } S \ ax = (P')_{\checkmark} \rangle \langle \text{preds } (\text{slice-kinds } S \ (ax \# asx)) \ s' \rangle$ 
  have  $P' = (\lambda s. \text{True})$  by(fastforce simp:slice-kinds-def)
  with  $\langle P = (\lambda s. \text{True}) \rangle \langle \forall s. (P \ s \longrightarrow \neg P' \ s) \wedge (P' \ s \longrightarrow \neg P \ s) \rangle$ 
  have False by blast
  thus ?thesis by simp
qed
qed

```

3.3.4 The set WS

inductive-set $WS :: 'node \text{ set} \Rightarrow (('node \times 'state) \times ('node \times 'state)) \text{ set}$
for $S :: 'node \text{ set}$
where $WSI: \llbracket \text{obs } n \ (\text{backward-slice } S) = \text{obs } n' \ (\text{backward-slice } S);$
 $\forall V \in \text{rv } S \ n. \text{state-val } s \ V = \text{state-val } s' \ V;$
 $\text{valid-node } n; \text{valid-node } n' \rrbracket$
 $\implies ((n,s),(n',s')) \in WS \ S$

lemma WSD :

$((n,s),(n',s')) \in WS \ S$
 $\implies \text{obs } n \ (\text{backward-slice } S) = \text{obs } n' \ (\text{backward-slice } S) \wedge$
 $(\forall V \in \text{rv } S \ n. \text{state-val } s \ V = \text{state-val } s' \ V) \wedge$
 $\text{valid-node } n \wedge \text{valid-node } n'$
by(auto elim:WS.cases)

lemma $WS\text{-silent-move}$:

assumes $((n_1,s_1),(n_2,s_2)) \in WS \ S$ **and** $S, \text{kind} \vdash (n_1,s_1) -a \rightarrow_{\tau} (n_1',s_1')$
and $\text{obs } n_1' \ (\text{backward-slice } S) \neq \{\}$ **shows** $((n_1',s_1'),(n_2,s_2)) \in WS \ S$
proof –
from $\langle ((n_1,s_1),(n_2,s_2)) \in WS \ S \rangle$ **have** $\text{valid-node } n_1$ **and** $\text{valid-node } n_2$
by(auto dest:WSD)
from $\langle S, \text{kind} \vdash (n_1,s_1) -a \rightarrow_{\tau} (n_1',s_1') \rangle$ **have** $\text{sourcenode } a = n_1$
and $\text{targetnode } a = n_1'$ **and** $\text{transfer } (\text{kind } a) \ s_1 = s_1'$
and $n_1 \notin \text{backward-slice } S$ **and** $\text{valid-edge } a$ **and** $\text{pred } (\text{kind } a) \ s_1$
by(auto elim:silent-move.cases)
from $\langle \text{targetnode } a = n_1' \rangle \langle \text{valid-edge } a \rangle$ **have** $\text{valid-node } n_1'$
by(auto simp:valid-node-def)

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

with $\langle \text{obs } n_1' \text{ (backward-slice } S) = \text{obs } n_2 \text{ (backward-slice } S) \rangle$
 $\langle \text{valid-node } n_1' \rangle \langle \text{valid-node } n_2 \rangle$ **show** *?thesis* **by** (*fastforce intro: WSI*)
qed

lemma *WS-silent-moves*:

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

lemma *WS-observable-move*:

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


```

with  $\langle n_1 \in (\text{backward-slice } S) \rangle$  have  $\text{pred } (\text{slice-kind } S \ a) \ s_2$ 
  by (simp add:slice-kind-in-slice)
from  $\langle n_1 \in (\text{backward-slice } S) \rangle$  obtain  $s_2'$ 
  where  $\text{transfer } (\text{slice-kind } S \ a) \ s_2 = s_2'$ 
  by (simp add:slice-kind-in-slice)
with  $\langle \text{pred } (\text{slice-kind } S \ a) \ s_2 \rangle \langle n_1 \in (\text{backward-slice } S) \rangle \langle \text{valid-edge } a \rangle$ 
have  $S, \text{slice-kind } S \vdash (n_1, s_2) -a \rightarrow (n_1', s_2')$ 
  by (fastforce intro:observable-moveI)
from  $\langle \text{obs } n_2 \ (\text{backward-slice } S) = \{n_1\} \rangle$ 
obtain as where  $S, \text{slice-kind } S \vdash (n_2, s_2) =_{as \Rightarrow \tau} (n_1, s_2)$ 
  by (erule obs-silent-moves)
with  $\langle S, \text{slice-kind } S \vdash (n_1, s_2) -a \rightarrow (n_1', s_2') \rangle$ 
have  $S, \text{slice-kind } S \vdash (n_2, s_2) =_{as @ [a] \Rightarrow} (n_1', s_2')$ 
  by  $-(\text{rule observable-moves-snoc})$ 
have  $\forall V \in \text{rv } S \ n_1'. \text{state-val } s_1' \ V = \text{state-val } s_2' \ V$ 
proof
  fix  $V$  assume  $\text{rv}: V \in \text{rv } S \ n_1'$ 
  show  $\text{state-val } s_1' \ V = \text{state-val } s_2' \ V$ 
  proof (cases V ∈ Def n1)
    case True
    thus ?thesis
    proof (cases kind a)
      case (Update f)
      with  $\langle \text{transfer } (\text{kind } a) \ s_1 = s_1' \rangle$  have  $s_1' = f \ s_1$  by simp
      from Update[THEN sym]  $\langle n_1 \in (\text{backward-slice } S) \rangle$ 
      have  $\text{slice-kind } S \ a = \uparrow f$ 
      by (fastforce intro:slice-kind-in-slice)
      with  $\langle \text{transfer } (\text{slice-kind } S \ a) \ s_2 = s_2' \rangle$  have  $s_2' = f \ s_2$  by simp
      from  $\langle \text{valid-edge } a \rangle \langle \forall V \in \text{Use } n_1. \text{state-val } s_1 \ V = \text{state-val } s_2 \ V \rangle$ 
      True Update  $\langle s_1' = f \ s_1 \rangle \langle s_2' = f \ s_2 \rangle$  show ?thesis
      by (fastforce dest:CFG-edge-transfer-uses-only-Use)
    next
    case (Predicate Q)
    with  $\langle \text{transfer } (\text{kind } a) \ s_1 = s_1' \rangle$  have  $s_1' = s_1$  by simp
    from Predicate[THEN sym]  $\langle n_1 \in (\text{backward-slice } S) \rangle$ 
    have  $\text{slice-kind } S \ a = (Q)_{\checkmark}$ 
    by (fastforce intro:slice-kind-in-slice)
    with  $\langle \text{transfer } (\text{slice-kind } S \ a) \ s_2 = s_2' \rangle$  have  $s_2' = s_2$  by simp
    with  $\langle \text{valid-edge } a \rangle \langle \forall V \in \text{Use } n_1. \text{state-val } s_1 \ V = \text{state-val } s_2 \ V \rangle$ 
    True Predicate  $\langle s_1' = s_1 \rangle \langle \text{pred } (\text{kind } a) \ s_1 \rangle \langle \text{pred } (\text{kind } a) \ s_2 \rangle$ 
    show ?thesis by (auto dest:CFG-edge-transfer-uses-only-Use)
  qed
next
  case False
  with  $\langle \text{valid-edge } a \rangle \langle \text{transfer } (\text{kind } a) \ s_1 = s_1' \rangle [THEN \text{sym}]$ 
   $\langle \text{pred } (\text{kind } a) \ s_1 \rangle \langle \text{pred } (\text{kind } a) \ s_2 \rangle$ 
  have  $\text{state-val } s_1' \ V = \text{state-val } s_1 \ V$ 
  by (fastforce intro:CFG-edge-no-Def-equal)
  have  $\text{state-val } s_2' \ V = \text{state-val } s_2 \ V$ 

```

```

proof(cases kind a)
  case (Update f)
  with  $\langle n_1 \in (\text{backward-slice } S) \rangle$  have slice-kind  $S \ a = \text{kind } a$ 
    by(fastforce intro:slice-kind-in-slice)
  with  $\langle \text{valid-edge } a \rangle \langle \text{transfer } (\text{slice-kind } S \ a) \ s_2 = s_2' [THEN \ \text{sym}]$ 
     $\text{False } \langle \text{pred } (\text{kind } a) \ s_2 \rangle$ 
  show ?thesis by(fastforce intro:CFG-edge-no-Def-equal)
next
  case (Predicate Q)
  with  $\langle \text{transfer } (\text{slice-kind } S \ a) \ s_2 = s_2' \rangle$  have  $s_2 = s_2'$ 
    by(cases slice-kind  $S \ a$ ,
      auto split:split-if-asm simp:slice-kind-def Let-def)
  thus ?thesis by simp
qed
from rv obtain as' nx where  $n_1' - as' \rightarrow^* nx$ 
  and  $nx \in (\text{backward-slice } S)$ 
  and  $V \in \text{Use } nx$  and  $\forall nx \in \text{set}(\text{sourcenodes } as'). \ V \notin \text{Def } nx$ 
  by(erule rvE)
from  $\langle \forall nx \in \text{set}(\text{sourcenodes } as'). \ V \notin \text{Def } nx \rangle$  False
have  $\forall nx \in \text{set}(\text{sourcenodes } (a \# as')). \ V \notin \text{Def } nx$ 
  by(auto simp:sourcenodes-def)
from  $\langle \text{valid-edge } a \rangle \langle n_1' - as' \rightarrow^* nx \rangle$  have  $n_1 - a \# as' \rightarrow^* nx$ 
  by(fastforce intro:Cons-path)
with  $\langle nx \in (\text{backward-slice } S) \rangle \langle V \in \text{Use } nx \rangle$ 
   $\langle \forall nx \in \text{set}(\text{sourcenodes } (a \# as')). \ V \notin \text{Def } nx \rangle$ 
have  $V \in rv \ S \ n_1$  by  $-(\text{rule } rvI)$ 
with  $\langle \forall V \in rv \ S \ n_1. \ \text{state-val } s_1 \ V = \text{state-val } s_2 \ V \rangle$ 
   $\langle \text{state-val } s_1' \ V = \text{state-val } s_1 \ V \rangle \langle \text{state-val } s_2' \ V = \text{state-val } s_2 \ V \rangle$ 
show ?thesis by fastforce
qed
qed
with  $\langle \text{valid-node } n_1' \rangle$  have  $((n_1', s_1'), (n_1', s_2')) \in WS \ S$  by(fastforce intro:WSI)
with  $\langle S, \text{slice-kind } S \vdash (n_2, s_2) = as@[a] \Rightarrow (n_1', s_2') \rangle$ 
   $\langle \text{transfer } (\text{slice-kind } S \ a) \ s_2 = s_2' \rangle$ 
show  $\exists as. ((n_1', s_1'), (n_1', \text{transfer } (\text{slice-kind } S \ a) \ s_2)) \in WS \ S \wedge$ 
   $S, \text{slice-kind } S \vdash (n_2, s_2) = as@[a] \Rightarrow (n_1', \text{transfer } (\text{slice-kind } S \ a) \ s_2)$ 
  by blast
qed

```

definition is-weak-sim ::

$((('node \times 'state) \times ('node \times 'state)) \ \text{set} \Rightarrow 'node \ \text{set} \Rightarrow \text{bool})$

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

$\forall n_1 \ s_1 \ n_2 \ s_2 \ n_1' \ s_1' \ as. ((n_1, s_1), (n_2, s_2)) \in R \wedge S, \text{kind} \vdash (n_1, s_1) = as \Rightarrow (n_1', s_1')$
 $\longrightarrow (\exists n_2' \ s_2' \ as'. ((n_1', s_1'), (n_2', s_2')) \in R \wedge$
 $S, \text{slice-kind } S \vdash (n_2, s_2) = as' \Rightarrow (n_2', s_2'))$

lemma *WS-weak-sim*:

assumes $((n_1, s_1), (n_2, s_2)) \in WS\ S$
and $S, kind \vdash (n_1, s_1) = as \Rightarrow (n_1', s_1')$
shows $((n_1', s_1'), (n_1', transfer\ (slice-kind\ S\ (last\ as))\ s_2)) \in WS\ S \wedge$
 $(\exists as'. S, slice-kind\ S \vdash (n_2, s_2) = as' @ [last\ as] \Rightarrow$
 $(n_1', transfer\ (slice-kind\ S\ (last\ as))\ s_2))$

proof –

from $\langle S, kind \vdash (n_1, s_1) = as \Rightarrow (n_1', s_1') \rangle$ **obtain** $a' as' n' s'$
where $S, kind \vdash (n_1, s_1) = as' \Rightarrow_{\tau} (n', s')$
and $S, kind \vdash (n', s') - a' \rightarrow (n_1', s_1')$ **and** $as = as' @ [a']$
by $(fastforce\ elim:observable-moves.cases)$
from $\langle S, kind \vdash (n', s') - a' \rightarrow (n_1', s_1') \rangle$ **have** $obs\ n' (backward-slice\ S) = \{n'\}$
by $(fastforce\ elim:observable-move.cases\ intro!:n-in-obs)$
hence $obs\ n' (backward-slice\ S) \neq \{\}$ **by** *fast*
with $\langle S, kind \vdash (n_1, s_1) = as' \Rightarrow_{\tau} (n', s') \rangle$ $\langle ((n_1, s_1), (n_2, s_2)) \in WS\ S \rangle$
have $((n', s'), (n_2, s_2)) \in WS\ S$
by $-(rule\ WS-silent-moves, simp+)$
with $\langle S, kind \vdash (n', s') - a' \rightarrow (n_1', s_1') \rangle$ **obtain** asx
where $((n_1', s_1'), (n_1', transfer\ (slice-kind\ S\ a')\ s_2)) \in WS\ S$
and $S, slice-kind\ S \vdash (n_2, s_2) = asx @ [a'] \Rightarrow$
 $(n_1', transfer\ (slice-kind\ S\ a')\ s_2)$
by $(fastforce\ elim:WS-observable-move)$
with $\langle as = as' @ [a'] \rangle$ **show**
 $((n_1', s_1'), (n_1', transfer\ (slice-kind\ S\ (last\ as))\ s_2)) \in WS\ S \wedge$
 $(\exists as'. S, slice-kind\ S \vdash (n_2, s_2) = as' @ [last\ as] \Rightarrow$
 $(n_1', transfer\ (slice-kind\ S\ (last\ as))\ s_2))$ **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 $n - as \rightarrow^* n'$ and transitive closure of $S, f \vdash (n, s) = as \Rightarrow_{\tau} (n', s')$

inductive *trans-observable-moves* ::

$'node\ set \Rightarrow ('edge \Rightarrow 'state\ edge-kind) \Rightarrow 'node \Rightarrow 'state \Rightarrow 'edge\ list \Rightarrow$
 $'node \Rightarrow 'state \Rightarrow bool\ (_, - \vdash '(-, -) = \Rightarrow^* '(-, -)\ [51, 50, 0, 0, 50, 0, 0]\ 51)$

where *tom-Nil*:

$S, f \vdash (n, s) = [] \Rightarrow^* (n, s)$

| *tom-Cons*:

$\llbracket S, f \vdash (n, s) = as \Rightarrow (n', s'); S, f \vdash (n', s') = as' \Rightarrow^* (n'', s'') \rrbracket$
 $\Rightarrow S, f \vdash (n, s) = (last\ as) \# as' \Rightarrow^* (n'', s'')$

definition *slice-edges* :: $'node\ set \Rightarrow 'edge\ list \Rightarrow 'edge\ list$

where $\text{slice-edges } S \text{ as} \equiv [a \leftarrow \text{as}. \text{sourcenode } a \in \text{backward-slice } S]$

lemma *silent-moves-no-slice-edges*:

$S, f \vdash (n, s) = \text{as} \Rightarrow_{\tau} (n', s') \implies \text{slice-edges } S \text{ as} = []$

by(*induct rule:silent-moves.induct, auto elim:silent-move.cases simp:slice-edges-def*)

lemma *observable-moves-last-slice-edges*:

$S, f \vdash (n, s) = \text{as} \Rightarrow (n', s') \implies \text{slice-edges } S \text{ as} = [\text{last as}]$

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*:

$\text{slice-edges } S \text{ as} = []$

$\implies \forall nx \in \text{set}(\text{sourcenodes as}). nx \notin (\text{backward-slice } S)$

proof(*induct as*)

case *Nil* **thus** ?*case* **by**(*simp add:slice-edges-def sourcenodes-def*)

next

case (*Cons a' as'*)

note $IH = \langle \text{slice-edges } S \text{ as}' = [] \implies$

$\forall nx \in \text{set}(\text{sourcenodes as}'). nx \notin \text{backward-slice } S \rangle$

from $\langle \text{slice-edges } S (a' \# \text{as}') = [] \rangle$ **have** $\text{slice-edges } S \text{ as}' = []$

and $\text{sourcenode } a' \notin \text{backward-slice } S$

by(*auto simp:slice-edges-def split:split-if-asm*)

from $IH[OF \langle \text{slice-edges } S \text{ as}' = [] \rangle \langle \text{sourcenode } a' \notin \text{backward-slice } S \rangle]$

show ?*case* **by**(*simp add:sourcenodes-def*)

qed

lemma *sliced-path-determ*:

$\llbracket n - \text{as} \rightarrow^* n'; n - \text{as}' \rightarrow^* n'; \text{slice-edges } S \text{ as} = \text{slice-edges } S \text{ as}';$

$\text{preds}(\text{slice-kinds } S \text{ as}) s; \text{preds}(\text{slice-kinds } S \text{ as}') s'; n' \in S;$

$\forall V \in \text{rv } S n. \text{state-val } s V = \text{state-val } s' V \rrbracket \implies \text{as} = \text{as}'$

proof(*induct arbitrary:as' s s' rule:path.induct*)

case (*empty-path n*)

from $\langle \text{slice-edges } S [] = \text{slice-edges } S \text{ as}' \rangle$

have $\forall nx \in \text{set}(\text{sourcenodes as}'). nx \notin (\text{backward-slice } S)$

by(*fastforce intro!:slice-edges-no-nodes-in-slice simp:slice-edges-def*)

with $\langle n - \text{as}' \rightarrow^* n \rangle$ **show** ?*case*

proof(*induct nx $\equiv$ n as' nx' $\equiv$ n rule:path.induct*)

case (*Cons-path n'' as a*)

from $\langle \text{valid-node } n \rangle \langle n \in S \rangle$ **have** $n \in \text{backward-slice } S$ **by**(*rule refl*)

with $\langle \forall nx \in \text{set}(\text{sourcenodes } (a \# \text{as})). nx \notin \text{backward-slice } S \rangle$

$\langle \text{sourcenode } a = n \rangle$

have *False* **by**(*simp add:sourcenodes-def*)

```

    thus ?case by simp
qed simp
next
case (Cons-path n'' as n' a n)
note IH = ⟨ $\bigwedge as' s s'. \llbracket n'' - as' \rightarrow^* n'; \text{slice-edges } S \text{ as} = \text{slice-edges } S \text{ as}';$ 
  preds (slice-kinds S as) s; preds (slice-kinds S as') s';  $n' \in S;$ 
 $\forall V \in rv \ S \ n''. \text{state-val } s \ V = \text{state-val } s' \ V \rrbracket \implies as = as'$ ⟩
show ?case
proof(cases as')
  case Nil
  with ⟨ $n - as' \rightarrow^* n'$ ⟩ have  $n = n'$  by fastforce
  from Nil ⟨slice-edges S (a#as) = slice-edges S as'⟩ ⟨sourcenode a = n⟩
  have  $n \notin \text{backward-slice } S$  by(fastforce simp:slice-edges-def)
  from ⟨valid-edge a⟩ ⟨sourcenode a = n⟩ ⟨ $n = n'$ ⟩ ⟨ $n' \in S$ ⟩
  have  $n \in \text{backward-slice } S$  by(fastforce intro:refl)
  with ⟨ $n = n'$ ⟩ ⟨ $n \notin \text{backward-slice } S$ ⟩ have False by simp
  thus ?thesis by simp
next
case (Cons ax asx)
  with ⟨ $n - as' \rightarrow^* n'$ ⟩ have  $n = \text{sourcenode ax}$  and valid-edge ax
  and targetnode ax  $-asx \rightarrow^* n'$  by(auto elim:path-split-Cons)
  show ?thesis
  proof(cases targetnode ax = n'')
    case True
    with ⟨targetnode ax  $-asx \rightarrow^* n'$ ⟩ have  $n'' - asx \rightarrow^* n'$  by simp
    from ⟨valid-edge ax⟩ ⟨valid-edge a⟩ ⟨ $n = \text{sourcenode ax}$ ⟩ ⟨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
    ⟨ $n = \text{sourcenode ax}$ ⟩ ⟨sourcenode a = n⟩
    have slice-edges S as = slice-edges S asx
    by(cases  $n \in \text{backward-slice } S$ )(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 ⟨ $\forall V \in rv \ S \ n. \text{state-val } s \ V = \text{state-val } s' \ V$ ⟩
    have  $\forall V \in rv \ S \ n''. \text{state-val } (\text{transfer } (\text{slice-kind } S \ a) \ s) \ V =$ 
      state-val (transfer (slice-kind S a) s') V
    by -(rule rv-edge-slice-kinds,auto)
    from IH[OF ⟨ $n'' - asx \rightarrow^* n'$ ⟩ ⟨slice-edges S as = slice-edges S asx⟩
    preds1 preds2 ⟨ $n' \in S$ ⟩ this] Cons ⟨ $ax = a$ ⟩ show ?thesis by simp
  next
  case False
  with ⟨valid-edge a⟩ ⟨valid-edge ax⟩ ⟨sourcenode a = n⟩ ⟨ $n = \text{sourcenode ax}$ ⟩
  ⟨targetnode a = n''⟩ ⟨preds (slice-kinds S (a#as)) s⟩

```

$\langle \text{preds } (\text{slice-kinds } S \text{ as}') s' \rangle \text{ Cons}$
 $\langle \forall V \in \text{rv } S \text{ n. state-val } s \text{ V} = \text{state-val } s' \text{ V} \rangle$
have *False* **by** $\neg(\text{erule rv-branching-edges-slice-kinds-False, auto})$
thus *?thesis* **by** *simp*
qed
qed
qed

lemma *path-trans-observable-moves*:

assumes $n - \text{as} \rightarrow^* n'$ **and** $\text{preds } (\text{kinds as}) s$ **and** $\text{transfers } (\text{kinds as}) s = s'$
obtains $n'' s'' \text{ as}' \text{ as}''$ **where** $S, \text{kind} \vdash (n, s) = \text{slice-edges } S \text{ as} \Rightarrow^* (n'', s'')$
and $S, \text{kind} \vdash (n'', s'') = \text{as}' \Rightarrow_\tau (n', s')$
and $\text{slice-edges } S \text{ as} = \text{slice-edges } S \text{ as}''$ **and** $n - \text{as}'' @ \text{as}' \rightarrow^* n'$
proof (*atomize-elim*)
from $\langle n - \text{as} \rightarrow^* n' \rangle \langle \text{preds } (\text{kinds as}) s \rangle \langle \text{transfers } (\text{kinds as}) s = s' \rangle$
show $\exists n'' s'' \text{ as}' \text{ as}''.$
 $S, \text{kind} \vdash (n, s) = \text{slice-edges } S \text{ as} \Rightarrow^* (n'', s'') \wedge$
 $S, \text{kind} \vdash (n'', s'') = \text{as}' \Rightarrow_\tau (n', s') \wedge \text{slice-edges } S \text{ as} = \text{slice-edges } S \text{ as}'' \wedge$
 $n - \text{as}'' @ \text{as}' \rightarrow^* n'$
proof (*induct arbitrary:s rule:path.induct*)
case (*empty-path n*)
from $\langle \text{transfers } (\text{kinds } []) s = s' \rangle$ **have** $s = s'$ **by** (*simp add:kinds-def*)
have $S, \text{kind} \vdash (n, s) = [] \Rightarrow^* (n, s)$ **by** (*rule tom-Nil*)
have $S, \text{kind} \vdash (n, s) = [] \Rightarrow_\tau (n, s)$ **by** (*rule silent-moves-Nil*)
with $\langle S, \text{kind} \vdash (n, s) = [] \Rightarrow^* (n, s) \rangle \langle s = s' \rangle \langle \text{valid-node } n \rangle$
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 $IH = \langle \bigwedge s. [\text{preds } (\text{kinds as}) s; \text{transfers } (\text{kinds as}) s = s']$
 $\implies \exists nx s'' \text{ as}' \text{ as}'' . S, \text{kind} \vdash (n'', s) = \text{slice-edges } S \text{ as} \Rightarrow^* (nx, s'') \wedge$
 $S, \text{kind} \vdash (nx, s'') = \text{as}' \Rightarrow_\tau (n', s') \wedge$
 $\text{slice-edges } S \text{ as} = \text{slice-edges } S \text{ as}'' \wedge n'' - \text{as}'' @ \text{as}' \rightarrow^* n' \rangle$
from $\langle \text{preds } (\text{kinds } (a \# \text{as})) s \rangle \langle \text{transfers } (\text{kinds } (a \# \text{as})) s = s' \rangle$
have $\text{preds } (\text{kinds as}) (\text{transfer } (\text{kind } a) s)$
 $\text{transfers } (\text{kinds as}) (\text{transfer } (\text{kind } a) s) = s'$ **by** (*simp-all add:kinds-def*)
from $IH[OF \text{ this}]$ **obtain** $nx \text{ sx asx asx}'$
where $S, \text{kind} \vdash (n'', \text{transfer } (\text{kind } a) s) = \text{slice-edges } S \text{ as} \Rightarrow^* (nx, \text{sx})$
and $S, \text{kind} \vdash (nx, \text{sx}) = \text{asx} \Rightarrow_\tau (n', s')$
and $\text{slice-edges } S \text{ as} = \text{slice-edges } S \text{ asx}'$
and $n'' - \text{asx}' @ \text{asx} \rightarrow^* n'$
by *clarsimp*
from $\langle \text{preds } (\text{kinds } (a \# \text{as})) s \rangle$ **have** $\text{pred } (\text{kind } a) s$ **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⇒τ (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→τ (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→τ (n'',transfer (kind a) s)⟩
    ⟨S,kind ⊢ (nx,sx) =asx⇒τ (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⇒τ (n',s')⟩
    ⟨S,kind ⊢ (n,s) -a→τ (ni,transfer (kind a) s)⟩
    ⟨si = transfer (kind a) s⟩
  have S,kind ⊢ (n,s) =a#asx⇒τ (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⇒τ (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→τ (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→τ (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'') =as⇒τ (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
qed

```

lemma *WS-weak-sim-trans*:

```

assumes ((n1,s1),(n2,s2)) ∈ WS S
and S,kind ⊢ (n1,s1) =as⇒* (n1',s1') and as ≠ []
shows ((n1',s1'),(n1',transfers (slice-kinds S as) s2)) ∈ WS S ∧
  S,slice-kind S ⊢ (n2,s2) =as⇒* (n1',transfers (slice-kinds S as) s2)

```

proof –

```

obtain f where f = kind by simp
with ⟨S,kind ⊢ (n1,s1) =as⇒* (n1',s1')⟩
have S,f ⊢ (n1,s1) =as⇒* (n1',s1') by simp
from ⟨S,f ⊢ (n1,s1) =as⇒* (n1',s1')⟩ ⟨((n1,s1),(n2,s2)) ∈ WS S⟩ ⟨as ≠ []⟩ ⟨f =
kind⟩
show ((n1',s1'),(n1',transfers (slice-kinds S as) s2)) ∈ WS S ∧
  S,slice-kind S ⊢ (n2,s2) =as⇒* (n1',transfers (slice-kinds S as) s2)
proof(induct arbitrary:n2 s2 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 =  $\langle \bigwedge n_2 s_2. \llbracket ((n', s'), (n_2, s_2)) \in WS\ S; as' \neq []; f = kind \rrbracket$ 
 $\implies ((n'', s''), (n'', transfers\ (slice-kinds\ S\ as')\ s_2)) \in WS\ S \wedge$ 
 $S, slice-kind\ S \vdash (n_2, s_2) = as' \Rightarrow^* (n'', transfers\ (slice-kinds\ S\ as')\ s_2) \rangle$ 
from  $\langle S, f \vdash (n, s) = as \Rightarrow (n', s') \rangle$ 
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  $\langle S, f \vdash (nx, sx) - ax \rightarrow (n', s') \rangle$  have obs nx (backward-slice S) = {nx}
by (fastforce intro!: n-in-obs elim: observable-move.cases)
with  $\langle S, f \vdash (n, s) = asx \Rightarrow_\tau (nx, sx) \rangle \langle ((n, s), (n_2, s_2)) \in WS\ S \rangle \langle f = kind \rangle$ 
have  $\langle (nx, sx), (n_2, s_2) \rangle \in WS\ S$  by (fastforce intro: WS-silent-moves)
with  $\langle S, f \vdash (nx, sx) - ax \rightarrow (n', s') \rangle \langle f = kind \rangle$ 
obtain asx' where  $\langle (n', s'), (n', transfer\ (slice-kind\ S\ ax)\ s_2) \rangle \in WS\ S$ 
and  $S, slice-kind\ S \vdash (n_2, s_2) = asx' @ [ax] \Rightarrow$ 
 $(n', transfer\ (slice-kind\ S\ ax)\ s_2)$ 
by (fastforce elim: WS-observable-move)
show ?case
proof (cases  $as' = []$ )
case True
with  $\langle S, f \vdash (n', s') = as' \Rightarrow^* (n'', s'') \rangle$  have  $n' = n'' \wedge s' = s''$ 
by (fastforce elim: trans-observable-moves.cases dest: observable-move-notempty)
from  $\langle S, slice-kind\ S \vdash (n_2, s_2) = asx' @ [ax] \Rightarrow$ 
 $(n', transfer\ (slice-kind\ S\ ax)\ s_2) \rangle$ 
have  $S, slice-kind\ S \vdash (n_2, s_2) = (last\ (asx' @ [ax])) \# [] \Rightarrow^*$ 
 $(n', transfer\ (slice-kind\ S\ ax)\ s_2)$ 
by (fastforce intro: trans-observable-moves.intros)
with  $\langle ((n', s'), (n', transfer\ (slice-kind\ S\ ax)\ s_2)) \in WS\ S \rangle \langle as = asx @ [ax] \rangle$ 
 $\langle n' = n'' \wedge s' = s'' \rangle$  True
show ?thesis by (fastforce simp: slice-kinds-def)
next
case False
from IH [OF  $\langle ((n', s'), (n', transfer\ (slice-kind\ S\ ax)\ s_2)) \in WS\ S \rangle$  this
 $\langle f = kind \rangle$ ]
have  $\langle ((n'', s''), (n'', transfers\ (slice-kinds\ S\ as')\ s_2)) \in WS\ S$ 
 $(transfer\ (slice-kind\ S\ ax)\ s_2)) \rangle \in WS\ S$ 
and  $S, slice-kind\ S \vdash (n', transfer\ (slice-kind\ S\ ax)\ s_2)$ 
 $= as' \Rightarrow^* (n'', transfers\ (slice-kinds\ S\ as')\ s_2)$ 
 $(transfer\ (slice-kind\ S\ ax)\ s_2))$  by simp-all
with  $\langle S, slice-kind\ S \vdash (n_2, s_2) = asx' @ [ax] \Rightarrow$ 
 $(n', transfer\ (slice-kind\ S\ ax)\ s_2) \rangle$ 
have  $S, slice-kind\ S \vdash (n_2, s_2) = (last\ (asx' @ [ax])) \# as' \Rightarrow^*$ 
 $(n'', transfers\ (slice-kinds\ S\ as')\ (transfer\ (slice-kind\ S\ ax)\ s_2))$ 
by (fastforce intro: trans-observable-moves.tom-Cons)
with  $\langle ((n'', s''), (n'', transfers\ (slice-kinds\ S\ as')\ s_2)) \in WS\ S \rangle$ 
 $(transfer\ (slice-kind\ S\ ax)\ s_2)) \rangle \in WS\ S$  False  $\langle as = asx @ [ax] \rangle$ 
show ?thesis by (fastforce simp: slice-kinds-def)

```

qed
qed
qed

lemma *transfers-slice-kinds-slice-edges*:
 $\text{transfers } (\text{slice-kinds } S \ (\text{slice-edges } S \ as)) \ s = \text{transfers } (\text{slice-kinds } S \ as) \ s$
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 = $\langle \bigwedge s. \text{transfers } (\text{slice-kinds } S \ (\text{slice-edges } S \ as')) \ s =$
 $\text{transfers } (\text{slice-kinds } S \ as') \ s \rangle$
show ?case
proof(*cases sourcenode a' ∈ backward-slice S*)
 case True
 hence eq: $\text{transfers } (\text{slice-kinds } S \ (\text{slice-edges } S \ (a' \# as')) \ s$
 $= \text{transfers } (\text{slice-kinds } S \ (\text{slice-edges } S \ as')) \ s$
 $(\text{transfer } (\text{slice-kind } S \ a') \ s)$
by(*simp add:slice-edges-def slice-kinds-def*)
 have $\text{transfers } (\text{slice-kinds } S \ (a' \# as')) \ s$
 $= \text{transfers } (\text{slice-kinds } S \ as') \ (\text{transfer } (\text{slice-kind } S \ a') \ s)$
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: $\text{transfers } (\text{slice-kinds } S \ (\text{slice-edges } S \ (a' \# as')) \ s$
 $= \text{transfers } (\text{slice-kinds } S \ (\text{slice-edges } S \ as')) \ s$
by(*simp add:slice-edges-def slice-kinds-def*)
from False **have** $\text{transfer } (\text{slice-kind } S \ a') \ s = s$
by(*cases kind a', auto simp:slice-kind-def Let-def*)
 hence $\text{transfers } (\text{slice-kinds } S \ (a' \# as')) \ s$
 $= \text{transfers } (\text{slice-kinds } S \ as') \ s$
by(*simp add:slice-kinds-def*)
 with eq IH[*of s*] **show** ?thesis **by** *simp*
 qed
 qed

lemma *trans-observable-moves-preds*:
 assumes $S, f \vdash (n, s) = as \Rightarrow^* (n', s')$ **and** *valid-node n*
 obtains *as'* **where** $\text{preds } (\text{map } f \ as') \ s$ **and** $\text{slice-edges } S \ as' = as$
and $n - as' \rightarrow^* n'$
proof(*atomize-elim*)
from $\langle S, f \vdash (n, s) = as \Rightarrow^* (n', s') \rangle \langle \text{valid-node } n \rangle$
show $\exists as'. \text{preds } (\text{map } f \ as') \ s \wedge \text{slice-edges } S \ as' = as \wedge n - as' \rightarrow^* n'$
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 →* n'
  by(fastforce dest:observable-moves-preds-transfers-path)+
from ⟨n - as →* n'⟩ have valid-node n' by(fastforce dest:path-valid-node)
from ⟨S, f ⊢ (n, s) = as ⇒ (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 →* n''
  by blast
from ⟨n - as →* n'⟩ ⟨n' - asx →* n''⟩ have n - as@asx →* 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 →* 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 →* n' and slice-edges S as = [] and n' ∈ backward-slice S
obtains as' where n - as' →* n' and preds (slice-kinds S as') s
  and slice-edges S as' = []
proof(atomize-elim)
from ⟨slice-edges S as = []⟩
have ∀ nx ∈ set(sourcenodes as). nx ∉ (backward-slice S)
  by(rule slice-edges-no-nodes-in-slice)
with ⟨n - as →* n'⟩ ⟨n' ∈ backward-slice S⟩ have n' ∈ obs n (backward-slice S)
  by -(rule obs-elem)
hence obs n (backward-slice S) = {n'} by(rule obs-singleton-element)
from ⟨n - as →* n'⟩ have valid-node n and valid-node n'
  by(fastforce dest:path-valid-node)+
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⟩ ⟨obs n (backward-slice S) = {n'}⟩
show ∃ as'. n - as' →* n' ∧ preds (slice-kinds S as') s ∧
  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 = ⟨ $\bigwedge n. \llbracket \text{distance } n \ n' \ x; \text{ obs } n \ (\text{backward-slice } S) = \{n'\} \rrbracket$ 
 $\implies \exists as'. n - as' \rightarrow^* n' \wedge \text{preds } (\text{slice-kinds } S \ as') \ s \wedge$ 
 $\text{slice-edges } S \ as' = [] \rangle$ 
from ⟨distance  $n \ n' \ (Suc \ x)$ ⟩ obtain a
  where valid-edge a and  $n = \text{sourcenode } a$ 
  and distance (targetnode a)  $n' \ x$ 
  and target:targetnode a = (SOME nx.  $\exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$ 
    distance (targetnode a')  $n' \ x \wedge$ 
    valid-edge a'  $\wedge \text{targetnode } a' = nx$ )
  by(auto elim:distance-successor-distance)
have  $n \notin \text{backward-slice } S$ 
proof
  assume  $n \in \text{backward-slice } S$ 
  from ⟨valid-edge a⟩ ⟨ $n = \text{sourcenode } a$ ⟩ have valid-node n by simp
  with ⟨ $n \in \text{backward-slice } S$ ⟩ 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  $n - [] \rightarrow^* n'$  by (fastforce intro:empty-path)
  with ⟨distance  $n \ n' \ (Suc \ x)$ ⟩ show False
    by (fastforce elim:distance.cases)
qed
from ⟨distance (targetnode a)  $n' \ x$ ⟩ ⟨ $n' \in \text{backward-slice } S$ ⟩
obtain m where  $m \in \text{obs } (\text{targetnode } a) \ (\text{backward-slice } S)$ 
  by (fastforce elim:distance.cases path-ex-obs)
from ⟨valid-edge a⟩ ⟨ $n \notin \text{backward-slice } S$ ⟩ ⟨ $n = \text{sourcenode } a$ ⟩
have obs (targetnode a) (backward-slice S)  $\subseteq$ 
  obs (sourcenode a) (backward-slice S)
  by -(rule edge-obs-subset, auto)
with ⟨ $m \in \text{obs } (\text{targetnode } a) \ (\text{backward-slice } S)$ ⟩ ⟨ $n = \text{sourcenode } a$ ⟩
  ⟨obs n (backward-slice S) = {n'}⟩
have  $n' \in \text{obs } (\text{targetnode } a) \ (\text{backward-slice } S)$  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  $- as \rightarrow^* n'$  and preds (slice-kinds S as) s
  and slice-edges S as = [] by blast
from ⟨targetnode a  $- as \rightarrow^* n'$ ⟩ ⟨valid-edge a⟩ ⟨ $n = \text{sourcenode } a$ ⟩
have  $n - a \# as \rightarrow^* n'$  by (fastforce intro:Cons-path)
from ⟨slice-edges S as = []⟩ ⟨ $n \notin \text{backward-slice } S$ ⟩ ⟨ $n = \text{sourcenode } a$ ⟩
have slice-edges S (a # as) = [] by (simp add:slice-edges-def)
show ?case
proof (cases kind a)
  case (Update f)
  with ⟨ $n \notin \text{backward-slice } S$ ⟩ ⟨ $n = \text{sourcenode } a$ ⟩ have slice-kind S a =  $\uparrow id$ 
    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 $\langle \text{preds (slice-kinds } S \text{ as)} \ s \rangle$ have preds (slice-kinds S ($a \# \text{as}$)) s
 by (simp add: slice-kinds-def)
 with $\langle n - a \# \text{as} \rightarrow^* n' \rangle$ $\langle \text{slice-edges } S \text{ (} a \# \text{as)} = [] \rangle$ show ?thesis
 by blast
 next
 case (Predicate Q)
 with $\langle n \notin \text{backward-slice } S \rangle$ $\langle n = \text{sourcenode } a \rangle$ $\langle \text{distance } n \ n' \ (\text{Suc } x) \rangle$
 $\langle \text{obs } n \ (\text{backward-slice } S) = \{n'\} \rangle$ $\langle \text{distance (targetnode } a) \ n' \ x \rangle$
 $\langle \text{targetnode } a = (\text{SOME } nx. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{distance (targetnode } a') \ n' \ x \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = nx) \rangle$
 have slice-kind S $a = (\lambda s. \text{True})_{\checkmark}$
 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 $\langle \text{preds (slice-kinds } S \text{ as)} \ s \rangle$ have preds (slice-kinds S ($a \# \text{as}$)) s
 by (simp add: slice-kinds-def)
 with $\langle n - a \# \text{as} \rightarrow^* n' \rangle$ $\langle \text{slice-edges } S \text{ (} a \# \text{as)} = [] \rangle$ show ?thesis by blast
 qed
 qed
 qed

theorem fundamental-property-of-static-slicing:

assumes path: $n - \text{as} \rightarrow^* n'$ and preds: preds (kinds as) s and $n' \in S$
 obtains as' where preds (slice-kinds S as') s
 and $(\forall V \in \text{Use } n'. \text{state-val (transfers (slice-kinds } S \text{ as}') \ s) \ V =$
 $\text{state-val (transfers (kinds as) \ s) \ V})$
 and slice-edges S $\text{as} = \text{slice-edges } S \ \text{as}'$ and $n - \text{as}' \rightarrow^* n'$
 proof (atomize-elim)
 from path preds obtain $n'' \ s'' \ \text{as}' \ \text{as}''$
 where $S, \text{kind} \vdash (n, s) = \text{slice-edges } S \ \text{as} \Rightarrow^* (n'', s'')$
 and $S, \text{kind} \vdash (n'', s'') = \text{as}' \Rightarrow_{\tau} (n', \text{transfers (kinds as) } s)$
 and slice-edges $S \ \text{as} = \text{slice-edges } S \ \text{as}''$
 and $n - \text{as}'' @ \text{as}' \rightarrow^* n'$
 by $-(\text{erule-tac } S = S \text{ in path-trans-observable-moves, auto})$
 from path have valid-node n and valid-node n'
 by (fastforce dest: path-valid-node)+
 from $\langle \text{valid-node } n \rangle$ have $((n, s), (n, s)) \in \text{WS } S$ by (fastforce intro: WSI)
 from $\langle \text{valid-node } n' \rangle$ $\langle n' \in S \rangle$ have obs n' (backward-slice S) $= \{n'\}$
 by (fastforce intro!: n-in-obs refl)
 from $\langle \text{valid-node } n' \rangle$ have $n' - [] \rightarrow^* n'$ by (fastforce intro: empty-path)
 with $\langle \text{valid-node } n' \rangle$ $\langle n' \in S \rangle$ have $\forall V \in \text{Use } n'. V \in \text{rv } S \ n'$
 by (fastforce intro: rvI refl simp: sourcenodes-def)
 show $\exists \text{as}'. \text{preds (slice-kinds } S \text{ as}') \ s \wedge$
 $(\forall V \in \text{Use } n'. \text{state-val (transfers (slice-kinds } S \text{ as}') \ s) \ V =$
 $\text{state-val (transfers (kinds as) \ s) \ V}) \wedge$

$\text{slice-edges } S \text{ as} = \text{slice-edges } S \text{ as}' \wedge n - \text{as}' \rightarrow^* n'$
proof(*cases slice-edges* $S \text{ as} = []$)
case *True*
hence *preds* (*slice-kinds* $S []$) s **and** *slice-edges* $S [] = \text{slice-edges } S \text{ as}$
by(*simp-all add:slice-kinds-def slice-edges-def*)
from $\langle S, \text{kind} \vdash (n, s) = \text{slice-edges } S \text{ as} \Rightarrow^* (n'', s'') \rangle$ *True*
have $n = n''$ **and** $s = s''$
by(*fastforce elim:trans-observable-moves.cases*) +
with $\langle S, \text{kind} \vdash (n'', s'') = \text{as}' \Rightarrow_{\tau} (n', \text{transfers } (\text{kinds } \text{as}) s) \rangle$
have $S, \text{kind} \vdash (n, s) = \text{as}' \Rightarrow_{\tau} (n', \text{transfers } (\text{kinds } \text{as}) s)$ **by** *simp*
with $\langle \text{valid-node } n \rangle$ **have** $n - \text{as}' \rightarrow^* n'$
by(*fastforce dest:silent-moves-preds-transfers-path*)
from $\langle S, \text{kind} \vdash (n, s) = \text{as}' \Rightarrow_{\tau} (n', \text{transfers } (\text{kinds } \text{as}) s) \rangle$
have *slice-edges* $S \text{ as}' = []$ **by**(*fastforce dest:silent-moves-no-slice-edges*)
with $\langle n - \text{as}' \rightarrow^* n' \rangle$ $\langle \text{valid-node } n' \rangle$ $\langle n' \in S \rangle$ **obtain** asx
where $n - \text{asx} \rightarrow^* n'$ **and** *preds* (*slice-kinds* $S \text{ asx}$) s
and *slice-edges* $S \text{ asx} = []$
by $-(\text{erule exists-sliced-path-preds, auto intro: refl})$
from $\langle S, \text{kind} \vdash (n, s) = \text{as}' \Rightarrow_{\tau} (n', \text{transfers } (\text{kinds } \text{as}) s) \rangle$
 $\langle ((n, s), (n, s)) \in \text{WS } S \rangle$ $\langle \text{obs } n' (\text{backward-slice } S) = \{n'\} \rangle$
have $((n', \text{transfers } (\text{kinds } \text{as}) s), (n, s)) \in \text{WS } S$
by(*fastforce intro:WS-silent-moves*)
with *True* **have** $\forall V \in \text{rv } S \text{ n'}. \text{state-val } (\text{transfers } (\text{kinds } \text{as}) s) V =$
 $\text{state-val } (\text{transfers } (\text{slice-kinds } S (\text{slice-edges } S \text{ as})) s) V$
by(*fastforce dest:WSD simp:slice-edges-def slice-kinds-def*)
with $\langle \forall V \in \text{Use } n'. V \in \text{rv } S \text{ n' } \rangle$
have $\forall V \in \text{Use } n'. \text{state-val } (\text{transfers } (\text{kinds } \text{as}) s) V =$
 $\text{state-val } (\text{transfers } (\text{slice-kinds } S (\text{slice-edges } S \text{ as})) s) V$ **by** *simp*
with $\langle \text{slice-edges } S \text{ asx} = [] \rangle$ $\langle \text{slice-edges } S [] = \text{slice-edges } S \text{ as} \rangle$
have $\forall V \in \text{Use } n'. \text{state-val } (\text{transfers } (\text{kinds } \text{as}) s) V =$
 $\text{state-val } (\text{transfers } (\text{slice-kinds } S (\text{slice-edges } S \text{ asx})) s) V$
by(*simp add:slice-edges-def*)
hence $\forall V \in \text{Use } n'. \text{state-val } (\text{transfers } (\text{kinds } \text{as}) s) V =$
 $\text{state-val } (\text{transfers } (\text{slice-kinds } S \text{ asx}) s) V$
by(*simp add:transfers-slice-kinds-slice-edges*)
with $\langle n - \text{asx} \rightarrow^* n' \rangle$ $\langle \text{preds } (\text{slice-kinds } S \text{ asx}) s \rangle$
 $\langle \text{slice-edges } S \text{ asx} = [] \rangle$ $\langle \text{slice-edges } S [] = \text{slice-edges } S \text{ as} \rangle$
show *?thesis*
by(*rule-tac x=asx in exI, simp add:slice-edges-def*)
next
case *False*
with $\langle S, \text{kind} \vdash (n, s) = \text{slice-edges } S \text{ as} \Rightarrow^* (n'', s'') \rangle$ $\langle ((n, s), (n, s)) \in \text{WS } S \rangle$
have $((n'', s''), (n'', \text{transfers } (\text{slice-kinds } S (\text{slice-edges } S \text{ as})) s)) \in \text{WS } S$
 $S, \text{slice-kind } S \vdash (n, s) = \text{slice-edges } S \text{ as} \Rightarrow^*$
 $(n'', \text{transfers } (\text{slice-kinds } S (\text{slice-edges } S \text{ as})) s)$
by(*fastforce dest:WS-weak-sim-trans*) +
from $\langle S, \text{slice-kind } S \vdash (n, s) = \text{slice-edges } S \text{ as} \Rightarrow^*$
 $(n'', \text{transfers } (\text{slice-kinds } S (\text{slice-edges } S \text{ as})) s) \rangle$
 $\langle \text{valid-node } n \rangle$

```

obtain  $asx$  where  $preds$  ( $slice\text{-}kinds\ S\ asx$ )  $s$ 
  and  $slice\text{-}edges\ S\ asx = slice\text{-}edges\ S\ as$ 
  and  $n - asx \rightarrow^* n''$ 
  by( $fastforce\ elim:trans\text{-}observable\text{-}moves\text{-}preds\ simp:slice\text{-}kinds\text{-}def$ )
from  $\langle n - asx \rightarrow^* n'' \rangle$  have  $valid\text{-}node\ n''$  by( $fastforce\ dest:path\text{-}valid\text{-}node$ )
with  $\langle S, kind \vdash (n'', s'') = as' \Rightarrow_\tau (n', transfers\ (kinds\ as)\ s) \rangle$ 
have  $n'' - as' \rightarrow^* n'$ 
  by( $fastforce\ dest:silent\text{-}moves\text{-}preds\text{-}transfers\text{-}path$ )
from  $\langle S, kind \vdash (n'', s'') = as' \Rightarrow_\tau (n', transfers\ (kinds\ as)\ s) \rangle$ 
have  $slice\text{-}edges\ S\ as' = []$  by( $fastforce\ dest:silent\text{-}moves\text{-}no\text{-}slice\text{-}edges$ )
with  $\langle n'' - as' \rightarrow^* n' \rangle$   $\langle valid\text{-}node\ n' \rangle$   $\langle n' \in S \rangle$  obtain  $asx'$ 
  where  $n'' - asx' \rightarrow^* n'$  and  $slice\text{-}edges\ S\ asx' = []$ 
  and  $preds$  ( $slice\text{-}kinds\ S\ asx'$ ) ( $transfers\ (slice\text{-}kinds\ S\ asx)\ s$ )
  by  $-(erule\ exists\text{-}sliced\text{-}path\text{-}preds, auto\ intro: refl)$ 
from  $\langle n - asx \rightarrow^* n'' \rangle$   $\langle n'' - asx' \rightarrow^* n' \rangle$  have  $n - asx @ asx' \rightarrow^* n'$ 
  by( $rule\ path\text{-}Append$ )
from  $\langle slice\text{-}edges\ S\ asx = slice\text{-}edges\ S\ as \rangle$   $\langle slice\text{-}edges\ S\ asx' = [] \rangle$ 
have  $slice\text{-}edges\ S\ as = slice\text{-}edges\ S\ (asx @ asx')$ 
  by( $auto\ simp:slice\text{-}edges\text{-}def$ )
from  $\langle preds\ (slice\text{-}kinds\ S\ asx')\ (transfers\ (slice\text{-}kinds\ S\ asx)\ s) \rangle$ 
   $\langle preds\ (slice\text{-}kinds\ S\ asx)\ s \rangle$ 
have  $preds\ (slice\text{-}kinds\ S\ (asx @ asx'))\ s$ 
  by( $simp\ add:slice\text{-}kinds\text{-}def\ preds\text{-}split$ )
from  $\langle obs\ n'\ (backward\text{-}slice\ S) = \{n'\} \rangle$ 
   $\langle S, kind \vdash (n'', s'') = as' \Rightarrow_\tau (n', transfers\ (kinds\ as)\ s) \rangle$ 
   $\langle ((n'', s''), (n'', transfers\ (slice\text{-}kinds\ S\ (slice\text{-}edges\ S\ as))\ s)) \in WS\ S \rangle$ 
have  $((n', transfers\ (kinds\ as)\ s),$ 
   $(n'', transfers\ (slice\text{-}kinds\ S\ (slice\text{-}edges\ S\ as))\ s)) \in WS\ S$ 
  by( $fastforce\ intro:WS\text{-}silent\text{-}moves$ )
hence  $\forall V \in rv\ S\ n'.\ state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V =$ 
   $state\text{-}val\ (transfers\ (slice\text{-}kinds\ S\ (slice\text{-}edges\ S\ as))\ s)\ V$ 
  by( $fastforce\ dest:WSD$ )
with  $\langle \forall V \in Use\ n'.\ V \in rv\ S\ n' \rangle$   $\langle slice\text{-}edges\ S\ asx = slice\text{-}edges\ S\ as \rangle$ 
have  $\forall V \in Use\ n'.\ state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V =$ 
   $state\text{-}val\ (transfers\ (slice\text{-}kinds\ S\ (slice\text{-}edges\ S\ asx))\ s)\ V$ 
  by  $fastforce$ 
with  $\langle slice\text{-}edges\ S\ asx' = [] \rangle$ 
have  $\forall V \in Use\ n'.\ state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V =$ 
   $state\text{-}val\ (transfers\ (slice\text{-}kinds\ S\ (slice\text{-}edges\ S\ (asx @ asx')))\ s)\ V$ 
  by( $auto\ simp:slice\text{-}edges\text{-}def$ )
hence  $\forall V \in Use\ n'.\ state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V =$ 
   $state\text{-}val\ (transfers\ (slice\text{-}kinds\ S\ (asx @ asx'))\ s)\ V$ 
  by( $simp\ add:transfers\text{-}slice\text{-}kinds\text{-}slice\text{-}edges$ )
with  $\langle preds\ (slice\text{-}kinds\ S\ (asx @ asx'))\ s \rangle$   $\langle n - asx @ asx' \rightarrow^* n' \rangle$ 
   $\langle slice\text{-}edges\ S\ as = slice\text{-}edges\ S\ (asx @ asx') \rangle$ 
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  $\Rightarrow$  'node and targetnode :: 'edge  $\Rightarrow$  'node
  and kind :: 'edge  $\Rightarrow$  'state edge-kind and valid-edge :: 'edge  $\Rightarrow$  bool
  and Entry :: 'node ('('Entry'-')) and Def :: 'node  $\Rightarrow$  'var set
  and Use :: 'node  $\Rightarrow$  'var set and state-val :: 'state  $\Rightarrow$  'var  $\Rightarrow$  'val
  and backward-slice :: 'node set  $\Rightarrow$  'node set
  and sem :: 'com  $\Rightarrow$  'state  $\Rightarrow$  'com  $\Rightarrow$  'state  $\Rightarrow$  bool
  (((1<-,->-))  $\Rightarrow$  / (1<-,->-)) [0,0,0,0] 81)
  and identifies :: 'node  $\Rightarrow$  'com  $\Rightarrow$  bool (-  $\triangleq$  - [51, 0] 80)

begin

theorem fundamental-property-of-path-slicing-semantically:
  assumes  $n \triangleq c$  and  $\langle c, s \rangle \Rightarrow \langle c', s' \rangle$ 
  obtains  $n'$  as where  $n - as \rightarrow^* n'$  and  $\text{preds } (\text{slice-kinds } \{n'\} \text{ as}) s$  and  $n' \triangleq c'$ 
  and  $\forall V \in \text{Use } n'. \text{state-val } (\text{transfers } (\text{slice-kinds } \{n'\} \text{ as}) s) V = \text{state-val } s' V$ 
proof(atomize-elim)
  from  $\langle n \triangleq c \rangle \langle \langle c, s \rangle \Rightarrow \langle c', s' \rangle \rangle$  obtain  $n'$  as where  $n - as \rightarrow^* n'$ 
  and  $\text{transfers } (\text{kinds as}) s = s'$  and  $\text{preds } (\text{kinds as}) s$  and  $n' \triangleq c'$ 
  by(fastforce dest:fundamental-property)
  from  $\langle n - as \rightarrow^* n' \rangle \langle \text{preds } (\text{kinds as}) s \rangle$  obtain  $as'$ 
  where  $\text{preds } (\text{slice-kinds } \{n'\} \text{ as}') s$ 
  and  $\text{vals}:\forall V \in \text{Use } n'. \text{state-val } (\text{transfers } (\text{slice-kinds } \{n'\} \text{ as}') s) V =$ 
   $\text{state-val } (\text{transfers } (\text{kinds as}) s) V$  and  $n - as' \rightarrow^* n'$ 
  by -(erule fundamental-property-of-static-slicing,auto)
  from  $\langle \text{transfers } (\text{kinds as}) s = s' \rangle \text{vals have } \forall V \in \text{Use } n'. \text{state-val } (\text{transfers } (\text{slice-kinds } \{n'\} \text{ as}') s) V = \text{state-val } s' V$ 
  by simp
  with  $\langle \text{preds } (\text{slice-kinds } \{n'\} \text{ as}') s \rangle \langle n - as' \rightarrow^* n' \rangle \langle n' \triangleq c' \rangle$ 
  show  $\exists as n'. n - as \rightarrow^* n' \wedge \text{preds } (\text{slice-kinds } \{n'\} \text{ as}) s \wedge n' \triangleq c' \wedge$ 
   $(\forall V \in \text{Use } n'. \text{state-val } (\text{transfers } (\text{slice-kinds } \{n'\} \text{ as}) s) V = \text{state-val } s' V)$ 
  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 $n' =$

($\exists a a' as. (n' \notin \text{set}(\text{sourcenodes } (a \# as))) \wedge (n - a \# as \rightarrow^* n') \wedge$
 $(n' \text{ postdominates } (\text{targetnode } a)) \wedge$
 $(\text{valid-edge } a') \wedge (\text{sourcenode } a' = n) \wedge$
 $(\neg n' \text{ postdominates } (\text{targetnode } a'))$)

by(*auto simp: standard-control-dependences-eq dyn-standard-control-dependence-def*)

lemma *Exit-not-standard-control-dependent*:

n controls_s (-Exit-) $\implies$ False

by(*auto simp: standard-control-dependences-eq*

intro: Exit-not-dyn-standard-control-dependent)

lemma *standard-control-dependence-def-variant*:

n controls_s $n' = (\exists as. (n - as \rightarrow^* n') \wedge (n \neq n') \wedge$
 $(\neg n' \text{ postdominates } n) \wedge (n' \notin \text{set}(\text{sourcenodes } as)) \wedge$
 $(\forall n'' \in \text{set}(\text{targetnodes } as). n' \text{ postdominates } n''))$

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 $\Rightarrow$ 'node $\Rightarrow$ bool

(- *weakly controls* - [51,0])

where *weak-control-dependences-eq*:

n *weakly controls* $n' \equiv \exists as. n$ *weakly controls* n' *via* as

lemma

weak-control-dependence-def: n *weakly controls* $n' =$

$(\exists a\ a'\ as. (n' \notin \text{set}(\text{sourcenodes } (a\#as)))) \wedge (n - a\#as \rightarrow^* n') \wedge$

$(n' \text{ strongly-postdominates } (\text{targetnode } a)) \wedge$

$(\text{valid-edge } a') \wedge (\text{sourcenode } a' = n) \wedge$

$(\neg n' \text{ strongly-postdominates } (\text{targetnode } a'))$

by(*auto simp: weak-control-dependences-eq dyn-weak-control-dependence-def*)

lemma *Exit-not-weak-control-dependent*:

n *weakly controls* (-Exit-) $\implies$ 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 $\Rightarrow$ 'node **and** *targetnode* :: 'edge $\Rightarrow$ 'node

and *kind* :: 'edge $\Rightarrow$ 'state *edge-kind* **and** *valid-edge* :: 'edge $\Rightarrow$ bool

and *Entry* :: 'node ('(-Entry'-)) **and** *Def* :: 'node $\Rightarrow$ 'var set
and *Use* :: 'node $\Rightarrow$ 'var set **and** *state-val* :: 'state $\Rightarrow$ 'var $\Rightarrow$ 'val
and *Exit* :: 'node ('(-Exit'-)) +
fixes *control-dependence* :: 'node $\Rightarrow$ 'node $\Rightarrow$ bool
(- *controls* - [51,0])
assumes *Exit-not-control-dependent*: *n controls n'* $\implies$ *n' $\neq$ (-Exit-)*
assumes *control-dependence-path*:
n controls n'
 $\implies \exists$ *as*. *CFG.path sourcenode targetnode valid-edge n as n' $\wedge$ as $\neq$ []*

begin

inductive *cdep-edge* :: 'node $\Rightarrow$ 'node $\Rightarrow$ bool
(- $\longrightarrow_{cd}$ - [51,0] 80)
and *ddep-edge* :: 'node $\Rightarrow$ 'var $\Rightarrow$ 'node $\Rightarrow$ bool
(- $\dashrightarrow_{dd}$ - [51,0,0] 80)
and *PDG-edge* :: 'node $\Rightarrow$ 'var option $\Rightarrow$ 'node $\Rightarrow$ 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' $\implies$ *n $\longrightarrow_{cd}$ n'*

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

inductive *PDG-path* :: 'node $\Rightarrow$ 'node $\Rightarrow$ bool
(- $\longrightarrow_{d*}$ - [51,0] 80)

where *PDG-path-Nil*:

valid-node n $\implies$ n $\longrightarrow_{d}$ n*

| *PDG-path-Append-cdep*:
 $\llbracket n \longrightarrow_{d*} n''; n'' \longrightarrow_{cd} n' \rrbracket \implies n \longrightarrow_{d*} n'$

| *PDG-path-Append-ddep*:
 $\llbracket n \longrightarrow_{d*} n''; n'' - V \rightarrow_{dd} n' \rrbracket \implies n \longrightarrow_{d*} n'$

lemma *PDG-path-cdep*: *n $\longrightarrow_{cd}$ n' $\implies$ 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:  $n - V \rightarrow_{dd} n' \implies n \rightarrow_{d*} n'$ 
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:
 $\llbracket n'' \rightarrow_{d*} n'; n \rightarrow_{d*} n'' \rrbracket \implies n \rightarrow_{d*} n'$ 
by (induct rule: PDG-path.induct, auto intro: PDG-path.intros)

```

```

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

```

```

lemma PDG-path-CFG-path:
assumes  $n \rightarrow_{d*} n'$  obtains  $as$  where  $n - as \rightarrow^* n'$ 
proof (atomize-elim)
from  $\langle n \rightarrow_{d*} n' \rangle$  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  $\langle n'' \rightarrow_{cd} n' \rangle$  obtain  $as$  where  $n'' - as \rightarrow^* n'$ 
by (fastforce elim: PDG-cdep-edge-CFG-path)
with  $\langle \exists as. n - as \rightarrow^* n'' \rangle$  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  $\langle n'' - V \rightarrow_{dd} n' \rangle$  obtain  $as$  where  $n'' - as \rightarrow^* n'$ 
by (fastforce elim: PDG-ddep-edge-CFG-path)
with  $\langle \exists as. n - as \rightarrow^* n'' \rangle$  obtain  $as'$  where  $n - as' @ as \rightarrow^* n'$ 
by (auto dest: path-Append)
thus ?case by blast
qed
qed

```

```

lemma PDG-path-Exit:  $\llbracket n \rightarrow_{d*} n'; n' = (-Exit-) \rrbracket \implies n = (-Exit-)$ 
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*:
 $\llbracket n \longrightarrow_d^* n'; \neg \text{inner-node } n' \rrbracket \implies n = n'$
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 $\langle n'' \longrightarrow_{cd} n' \rangle \langle \neg \text{inner-node } n' \rangle$ **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 $\langle n'' - V \longrightarrow_{dd} n' \rangle \langle \neg \text{inner-node } n' \rangle$ **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* $\Rightarrow$ *'node set*
 where *PDG-BS* *S* $\equiv \{n'. \exists n. n' \longrightarrow_d^* n \wedge n \in S \wedge \text{valid-node } n\}$

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

lemma *Exit-PDG-BS*: $n \in \text{PDG-BS } \{(-\text{Exit-})\} \implies n = (-\text{Exit-})$
 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  $\Rightarrow$  'node and targetnode :: 'edge  $\Rightarrow$  'node
  and kind :: 'edge  $\Rightarrow$  'state edge-kind and valid-edge :: 'edge  $\Rightarrow$  bool
  and Entry :: 'node ('('Entry'-')) and Def :: 'node  $\Rightarrow$  'var set
  and Use :: 'node  $\Rightarrow$  'var set and state-val :: 'state  $\Rightarrow$  'var  $\Rightarrow$  '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 controlss n'
  show n'  $\neq$  (-Exit-)
  proof
    assume n' = (-Exit-)
    with  $\langle n \text{ controls}_s n' \rangle$  show False
    by(fastforce intro:Exit-not-standard-control-dependent)
  qed
next
  fix n n' assume n controlss n'
  thus  $\exists as. n -as \rightarrow^* n' \wedge as \neq []$ 
  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  $\Rightarrow$  'node and targetnode :: 'edge  $\Rightarrow$  'node
  and kind :: 'edge  $\Rightarrow$  'state edge-kind and valid-edge :: 'edge  $\Rightarrow$  bool
  and Entry :: 'node ('('Entry'-')) and Def :: 'node  $\Rightarrow$  'var set
  and Use :: 'node  $\Rightarrow$  'var set and state-val :: 'state  $\Rightarrow$  'var  $\Rightarrow$  '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 ∃ as. n -as→* n' ∧ as ≠ []
    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 ⇒ 'node ⇒ 'node ⇒ bool
 (- →_{wod} -, -)
where *wod-def*: $n \rightarrow_{wod} n_1, n_2 \equiv ((n_1 \neq n_2) \wedge$
 $(\exists as. (n -as \rightarrow^* n_1) \wedge (n_2 \notin \text{set}(\text{sourcenodes } as))) \wedge$
 $(\exists as. (n -as \rightarrow^* n_2) \wedge (n_1 \notin \text{set}(\text{sourcenodes } as))) \wedge$
 $(\exists a. (\text{valid-edge } a) \wedge (n = \text{sourcenode } a) \wedge$
 $((\exists as. (\text{targetnode } a -as \rightarrow^* n_1) \wedge$
 $(\forall as'. (\text{targetnode } a -as' \rightarrow^* n_2) \longrightarrow n_1 \in \text{set}(\text{sourcenodes } as')))) \vee$
 $(\exists as. (\text{targetnode } a -as \rightarrow^* n_2) \wedge$
 $(\forall as'. (\text{targetnode } a -as' \rightarrow^* n_1) \longrightarrow n_2 \in \text{set}(\text{sourcenodes } as')))))))$

inductive-set (in *CFG-wf*) *wod-backward-slice* :: 'node set ⇒ 'node set

for *S* :: 'node set

where *refl*: $\llbracket \text{valid-node } n; n \in S \rrbracket \Longrightarrow n \in \text{wod-backward-slice } S$

| *cd-closed*:

$\llbracket n' \rightarrow_{wod} n_1, n_2; n_1 \in \text{wod-backward-slice } S; n_2 \in \text{wod-backward-slice } S \rrbracket$
 $\Longrightarrow n' \in \text{wod-backward-slice } S$

| $dd\text{-closed}:\llbracket n' \text{ influences } V \text{ in } n''; n'' \in \text{wod-backward-slice } S \rrbracket$
 $\implies n' \in \text{wod-backward-slice } S$

lemma (in *CFG-wf*)
 $\text{wod-backward-slice-valid-node}: n \in \text{wod-backward-slice } S \implies \text{valid-node } n$
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*: $(\text{-Exit-}) \in \text{obs } n' (PDG\text{-}BS \ S) \implies (\text{-Exit-}) \in S$
by(*auto elim:obsE PDG-path-CFG-path simp:PDG-BS-def split:split-if-asm*)

abbreviation *PDG-path'* :: $'node \Rightarrow 'node \Rightarrow \text{bool}$ ($\text{-} \longrightarrow_d^* \text{-}$ [51,0] 80)
where $n \longrightarrow_d^* n' \equiv PDG.PDG\text{-}path \ \text{sourcenode} \ \text{targetnode} \ \text{valid-edge} \ \text{Def} \ \text{Use}$
 $\text{standard-control-dependence } n \ n'$

lemma *cd-closed*:
 $\llbracket n' \in PDG\text{-}BS \ S; n \ \text{controls}_s \ n' \rrbracket \implies n \in PDG\text{-}BS \ S$
by(*simp add:PDG-BS-def*)(*blast dest:PDG-cdep-edge PDG-path-Append PDG-path-cdep*)

lemma *obs-postdominate*:
assumes $n \in \text{obs } n' (PDG\text{-}BS \ S)$ **and** $n \neq (\text{-Exit-})$ **shows** $n \ \text{postdominates } n'$
proof(*rule ccontr*)
assume $\neg n \ \text{postdominates } n'$
from $\langle n \in \text{obs } n' (PDG\text{-}BS \ S) \rangle$ **have** *valid-node* n **by**(*fastforce dest:in-obs-valid*)
with $\langle n \in \text{obs } n' (PDG\text{-}BS \ S) \rangle \langle n \neq (\text{-Exit-}) \rangle$ **have** $n \ \text{postdominates } n$
by(*fastforce intro:postdominate-refl*)
from $\langle n \in \text{obs } n' (PDG\text{-}BS \ S) \rangle$ **obtain** *as* **where** $n' \text{-as} \rightarrow^* n$
and $\forall n' \in \text{set}(\text{sourcenodes } as). n' \notin (PDG\text{-}BS \ S)$
and $n \in (PDG\text{-}BS \ S)$ **by**(*erule obsE*)
from $\langle n \ \text{postdominates } n \rangle \langle \neg n \ \text{postdominates } n' \rangle \langle n' \text{-as} \rightarrow^* n \rangle$
obtain $as' \ a \ as''$ **where** $[simp]: as = as' @ a \# as''$ **and** *valid-edge* a

and $\neg n \text{ postdominates } (\text{sourcenode } a)$ **and** $n \text{ postdominates } (\text{targetnode } a)$
by $\neg(\text{erule postdominate-path-branch})$
from $\langle \neg n \text{ postdominates } (\text{sourcenode } a) \rangle \langle \text{valid-edge } a \rangle \langle \text{valid-node } n \rangle$
obtain asx **where** $\text{sourcenode } a - asx \rightarrow^* (-Exit-)$
and $n \notin \text{set}(\text{sourcenodes } asx)$ **by** $(\text{auto simp: postdominate-def})$
from $\langle \text{sourcenode } a - asx \rightarrow^* (-Exit-) \rangle \langle \text{valid-edge } a \rangle$
obtain $ax \ asx'$ **where** $[simp]: asx = ax \# asx'$
apply $\neg \text{apply}(\text{erule path.cases})$
apply $(\text{drule-tac } s = (-Exit-) \text{ in sym})$
apply simp
apply $(\text{drule Exit-source})$
apply simp-all
by fastforce
with $\langle \text{sourcenode } a - asx \rightarrow^* (-Exit-) \rangle$ **have** $\text{sourcenode } a - [] @ ax \# asx' \rightarrow^* (-Exit-)$

by simp
hence $\text{valid-edge } ax$ **and** $[simp]: \text{sourcenode } a = \text{sourcenode } ax$
and $\text{targetnode } ax - asx' \rightarrow^* (-Exit-)$
by $(\text{fastforce dest: path-split}) +$
with $\langle n \notin \text{set}(\text{sourcenodes } asx) \rangle$ **have** $\neg n \text{ postdominates } \text{targetnode } ax$
by $(\text{fastforce simp: postdominate-def sourcenodes-def})$
from $\langle n \in \text{obs } n' (PDG-BS S) \rangle \langle \forall n' \in \text{set}(\text{sourcenodes } as). n' \notin (PDG-BS S) \rangle$
have $n \notin \text{set}(\text{sourcenodes } (a \# as''))$
by $(\text{fastforce elim: obs.cases simp: sourcenodes-def})$
from $\langle n' - as \rightarrow^* n \rangle$ **have** $\text{sourcenode } a - a \# as'' \rightarrow^* n$
by $(\text{fastforce dest: path-split-second})$
with $\langle n \text{ postdominates } (\text{targetnode } a) \rangle \langle \neg n \text{ postdominates } \text{targetnode } ax \rangle$
 $\langle \text{valid-edge } ax \rangle \langle n \notin \text{set}(\text{sourcenodes } (a \# as'')) \rangle$
have $\text{sourcenode } a \text{ controls}_s n$ **by** $(\text{fastforce simp: standard-control-dependence-def})$
with $\langle n \in \text{obs } n' (PDG-BS S) \rangle$ **have** $\text{sourcenode } a \in (PDG-BS S)$
by $(\text{fastforce intro: cd-closed PDG-cdep-edge elim: obs.cases})$
with $\langle \forall n' \in \text{set}(\text{sourcenodes } as). n' \notin (PDG-BS S) \rangle$
show False **by** $(\text{simp add: sourcenodes-def})$
qed

lemma $\text{obs-singleton}: (\exists m. \text{obs } n (PDG-BS S) = \{m\}) \vee \text{obs } n (PDG-BS S) = \{\}$
proof (rule ccontr)
assume $\neg ((\exists m. \text{obs } n (PDG-BS S) = \{m\}) \vee \text{obs } n (PDG-BS S) = \{\})$
hence $\exists nx \ nx'. nx \in \text{obs } n (PDG-BS S) \wedge nx' \in \text{obs } n (PDG-BS S) \wedge$
 $nx \neq nx'$ **by** auto
then obtain $nx \ nx'$ **where** $nx \in \text{obs } n (PDG-BS S)$ **and** $nx' \in \text{obs } n (PDG-BS S)$
and $nx \neq nx'$ **by** auto
from $\langle nx \in \text{obs } n (PDG-BS S) \rangle$ **obtain** as **where** $n - as \rightarrow^* nx$
and $\forall n' \in \text{set}(\text{sourcenodes } as). n' \notin (PDG-BS S)$ **and** $nx \in (PDG-BS S)$
by (erule obsE)
from $\langle n - as \rightarrow^* nx \rangle$ **have** $\text{valid-node } nx$ **by** $(\text{fastforce dest: path-valid-node})$

```

with  $\langle nx \in (PDG-BS\ S) \rangle$  have  $obs\ nx\ (PDG-BS\ S) = \{nx\}$  by  $-(rule\ n-in-obs)$ 
with  $\langle n - as \rightarrow^* nx \rangle$   $\langle nx \in obs\ n\ (PDG-BS\ S) \rangle$   $\langle nx' \in obs\ n\ (PDG-BS\ S) \rangle$   $\langle nx \neq nx' \rangle$ 
have  $as \neq []$  by  $(fastforce\ elim:path.cases)$ 
with  $\langle n - as \rightarrow^* nx \rangle$   $\langle nx \in obs\ n\ (PDG-BS\ S) \rangle$   $\langle nx' \in obs\ n\ (PDG-BS\ S) \rangle$ 
 $\langle nx \neq nx' \rangle$   $\langle obs\ nx\ (PDG-BS\ S) = \{nx\} \rangle$   $\langle \forall n' \in set(sourcenodes\ as). n' \notin (PDG-BS\ S) \rangle$ 
have  $\exists a\ as'\ as''. n - as' \rightarrow^* sourcenode\ a \wedge targetnode\ a - as'' \rightarrow^* nx \wedge$ 
 $valid-edge\ a \wedge as = as' @ a \# as'' \wedge$ 
 $obs\ (targetnode\ a)\ (PDG-BS\ S) = \{nx\} \wedge$ 
 $(\neg (\exists m. obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{m\} \vee$ 
 $obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{\}))$ 
proof  $(induct\ arbitrary:nx'\ rule:path.induct)$ 
case  $(Cons-path\ n''\ as\ n')$ 
note  $[simp] = \langle sourcenode\ a = n \rangle [THEN\ sym] \langle targetnode\ a = n' \rangle [THEN\ sym]$ 
note  $more-than-one = \langle n' \in obs\ n\ (PDG-BS\ S) \rangle \langle nx' \in obs\ n\ (PDG-BS\ S) \rangle$ 
 $\langle n' \neq nx' \rangle$ 
note  $IH = \langle \bigwedge nx'. \llbracket n' \in obs\ n''\ (PDG-BS\ S); nx' \in obs\ n''\ (PDG-BS\ S); n' \neq nx' \rrbracket$ 
 $obs\ n'\ (PDG-BS\ S) = \{n'\}; \forall n' \in set\ (sourcenodes\ as). n' \notin (PDG-BS\ S); as \neq []$ 
 $\implies \exists a\ as'\ as''. n'' - as' \rightarrow^* sourcenode\ a \wedge targetnode\ a - as'' \rightarrow^* n' \wedge$ 
 $valid-edge\ a \wedge as = as' @ a \# as'' \wedge obs\ (targetnode\ a)\ (PDG-BS\ S) = \{n'\} \wedge$ 
 $(\neg (\exists m. obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{m\} \vee$ 
 $obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{\}))$ 
from  $\langle \forall n' \in set\ (sourcenodes\ (a \# as)). n' \notin (PDG-BS\ S) \rangle$ 
have  $\forall n' \in set\ (sourcenodes\ as). n' \notin (PDG-BS\ S)$  and  $sourcenode\ a \notin (PDG-BS\ S)$ 
by  $(simp-all\ add:sourcenodes-def)$ 
show  $?case$ 
proof  $(cases\ as = [])$ 
case  $True$ 
with  $\langle n'' - as \rightarrow^* n' \rangle$  have  $[simp]: n' = n''$  by  $(fastforce\ elim:path.cases)$ 
from  $more-than-one$ 
have  $\neg (\exists m. obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{m\} \vee$ 
 $obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{\})$ 
by  $auto$ 
with  $\langle obs\ n'\ (PDG-BS\ S) = \{n'\} \rangle$   $True\ \langle valid-edge\ a \rangle$  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  $as \neq []$  .
from  $\langle n'' - as \rightarrow^* n' \rangle$   $\langle \forall n' \in set\ (sourcenodes\ as). n' \notin (PDG-BS\ S) \rangle$ 
have  $obs\ n'\ (PDG-BS\ S) \subseteq obs\ n''\ (PDG-BS\ S)$  by  $(rule\ path-obs-subset)$ 
show  $?thesis$ 
proof  $(cases\ obs\ n'\ (PDG-BS\ S) = obs\ n''\ (PDG-BS\ S))$ 

```

```

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

```

```

with  $\langle \text{obs}(\text{targetnode } a) (PDG\text{-}BS\ S) = \{nx\} \rangle$ 
have  $nx \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S)$  by simp
with more-than-one obtain  $m$  where  $m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S)$ 
and  $nx \neq m$  by auto
from  $\langle m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$ 
have valid-node  $m$  by (fastforce dest:in-obs-valid)
from  $\langle \text{obs}(\text{targetnode } a) (PDG\text{-}BS\ S) = \{nx\} \rangle$  have valid-node  $nx$ 
by (fastforce dest:in-obs-valid)
show False
proof (cases m postdominates (sourcenode a))
  case True
    with  $\langle nx \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$   $\langle m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$ 
    have  $m \text{ postdominates } nx$ 
    by (fastforce intro:postdominate-path-targetnode elim:obs.cases)
    with  $\langle nx \neq m \rangle$  have  $\neg nx \text{ postdominates } m$  by (fastforce dest:postdominate-antisym)
    have  $(\text{-Exit-}) \rightarrow^* (\text{-Exit-})$  by (fastforce intro:empty-path)
    with  $\langle m \text{ postdominates } nx \rangle$  have  $nx \neq (\text{-Exit-})$ 
    by (fastforce simp:postdominate-def sourcenodes-def)
    have  $\neg nx \text{ postdominates } (\text{sourcenode } a)$ 
    proof (rule ccontr)
      assume  $\neg \neg nx \text{ postdominates } \text{sourcenode } a$ 
      hence  $nx \text{ postdominates } \text{sourcenode } a$  by simp
      from  $\langle m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$   $\langle nx \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$ 
      obtain  $asx'$  where  $\text{sourcenode } a \rightarrow^* asx' m$  and  $nx \notin \text{set}(\text{sourcenodes } asx')$ 
      by (fastforce elim:obs.cases)
      with  $\langle nx \text{ postdominates } \text{sourcenode } a \rangle$  have  $nx \text{ postdominates } m$ 
      by (rule postdominate-path-targetnode)
      with  $\langle \neg nx \text{ postdominates } m \rangle$  show False by simp
    qed
    with  $\langle nx \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$   $\langle \text{valid-node } nx \rangle$   $\langle nx \neq (\text{-Exit-}) \rangle$ 
    show False by (fastforce dest:obs-postdominate)
  next
    case False
    show False
    proof (cases m = Exit)
      case True
        from  $\langle m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$   $\langle nx \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$ 
        obtain  $xs$  where  $\text{sourcenode } a \rightarrow^* xs m$  and  $nx \notin \text{set}(\text{sourcenodes } xs)$ 
        by (fastforce elim:obsE)
        obtain  $x' xs'$  where  $[simp]: xs = x' \# xs'$ 
        proof (cases xs)
          case Nil
            with  $\langle \text{sourcenode } a \rightarrow^* xs m \rangle$  have  $[simp]: \text{sourcenode } a = m$  by fastforce
            with  $\langle m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$ 
            have  $m \in (PDG\text{-}BS\ S)$  by (fastforce elim:obsE)
            with  $\langle \text{valid-node } m \rangle$  have  $\text{obs } m (PDG\text{-}BS\ S) = \{m\}$ 

```

```

    by(rule n-in-obs)
  with  $\langle nx \in \text{obs}(\text{sourcenode } a) \text{ (PDG-BS } S) \rangle \langle nx \neq m \rangle$  have False
    by fastforce
  thus ?thesis by simp
qed blast
from  $\langle \text{sourcenode } a - xs \rightarrow^* m \rangle$  have  $\text{sourcenode } a = \text{sourcenode } x'$ 
  and  $\text{valid-edge } x' \text{ and targetnode } x' - xs' \rightarrow^* m$ 
  by(auto elim:path-split-Cons)
from  $\langle \text{targetnode } x' - xs' \rightarrow^* m \rangle \langle nx \notin \text{set}(\text{sourcenodes } xs) \rangle \langle \text{valid-edge } x' \rangle$ 
   $\langle \text{valid-node } m \rangle$  True
have  $\neg nx \text{ postdominates } (\text{targetnode } x')$ 
  by(fastforce simp:postdominate-def sourcenodes-def)
from  $\langle nx \neq m \rangle$  True have  $nx \neq (-Exit-)$  by simp
with  $\langle \text{obs}(\text{targetnode } a) \text{ (PDG-BS } S) = \{nx\} \rangle$ 
have  $nx \text{ postdominates } (\text{targetnode } a)$ 
  by(fastforce intro:obs-postdominate)
from  $\langle \text{obs}(\text{targetnode } a) \text{ (PDG-BS } S) = \{nx\} \rangle$ 
obtain ys where  $\text{targetnode } a - ys \rightarrow^* nx$ 
  and  $\forall nx' \in \text{set}(\text{sourcenodes } ys). nx' \notin (\text{PDG-BS } S)$ 
  and  $nx \in (\text{PDG-BS } S)$  by(fastforce elim:obsE)
hence  $nx \notin \text{set}(\text{sourcenodes } ys)$  by fastforce
have  $\text{sourcenode } a \neq nx$ 
proof
  assume  $\text{sourcenode } a = nx$ 
  from  $\langle nx \in \text{obs}(\text{sourcenode } a) \text{ (PDG-BS } S) \rangle$ 
  have  $nx \in (\text{PDG-BS } S)$  by  $\neg(\text{erule obsE})$ 
  with  $\langle \text{valid-node } nx \rangle$  have  $\text{obs } nx \text{ (PDG-BS } S) = \{nx\}$  by  $\neg(\text{erule n-in-obs})$ 
  with  $\langle \text{sourcenode } a = nx \rangle \langle m \in \text{obs}(\text{sourcenode } a) \text{ (PDG-BS } S) \rangle$ 
   $\langle nx \neq m \rangle$  show False by fastforce
qed
with  $\langle nx \notin \text{set}(\text{sourcenodes } ys) \rangle$  have  $nx \notin \text{set}(\text{sourcenodes } (a \# ys))$ 
  by(fastforce simp:sourcenodes-def)
from  $\langle \text{valid-edge } a \rangle \langle \text{targetnode } a - ys \rightarrow^* nx \rangle$ 
have  $\text{sourcenode } a - a \# ys \rightarrow^* nx$  by(fastforce intro:Cons-path)
from  $\langle \text{sourcenode } a - a \# ys \rightarrow^* nx \rangle \langle nx \notin \text{set}(\text{sourcenodes } (a \# ys)) \rangle$ 
   $\langle nx \text{ postdominates } (\text{targetnode } a) \rangle \langle \text{valid-edge } x' \rangle$ 
   $\langle \neg nx \text{ postdominates } (\text{targetnode } x') \rangle \langle \text{sourcenode } a = \text{sourcenode } x' \rangle$ 
have  $(\text{sourcenode } a) \text{ controls}_s nx$ 
  by(fastforce simp:standard-control-dependence-def)
with  $\langle nx \in (\text{PDG-BS } S) \rangle$  have  $\text{sourcenode } a \in (\text{PDG-BS } S)$ 
  by(rule cd-closed)
with  $\langle \text{valid-edge } a \rangle$  have  $\text{obs}(\text{sourcenode } a) \text{ (PDG-BS } S) = \{\text{sourcenode } a\}$ 
  by(fastforce intro!:n-in-obs)
with  $\langle m \in \text{obs}(\text{sourcenode } a) \text{ (PDG-BS } S) \rangle$ 
   $\langle nx \in \text{obs}(\text{sourcenode } a) \text{ (PDG-BS } S) \rangle \langle nx \neq m \rangle$ 
show False by simp
next
case False
with  $\langle m \in \text{obs}(\text{sourcenode } a) \text{ (PDG-BS } S) \rangle \langle \text{valid-node } m \rangle$ 

```

```

      ⟨¬ 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 ∈ 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-scd, OF PDG.PDG-ddep-edge[OF
PDG-scd]])
      dest:PDG-path-Append simp:PDG-BS-def split:split-if-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

```

3.8.2 Weak control dependence

context WeakControlDependencePDG **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:split-if-asm)

lemma cd-closed:

[[n' ∈ PDG-BS S; n weakly controls n']] ⇒ n ∈ 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 $n \in \text{obs } n' \text{ (PDG-BS } S)$ **and** $n \neq (-\text{Exit-})$
shows $n \text{ strongly-postdominates } n'$
proof(*rule ccontr*)
assume $\neg n \text{ strongly-postdominates } n'$
from $\langle n \in \text{obs } n' \text{ (PDG-BS } S) \rangle$ **have** *valid-node* n **by**(*fastforce dest:in-obs-valid*)
with $\langle n \in \text{obs } n' \text{ (PDG-BS } S) \rangle \langle n \neq (-\text{Exit-}) \rangle$ **have** $n \text{ strongly-postdominates } n$
by(*fastforce intro:strong-postdominate-refl*)
from $\langle n \in \text{obs } n' \text{ (PDG-BS } S) \rangle$ **obtain** as **where** $n' - as \rightarrow^* n$
and $\forall n' \in \text{set}(\text{sourcenodes } as). n' \notin \text{(PDG-BS } S)$
and $n \in \text{(PDG-BS } S)$ **by**(*erule obsE*)
from $\langle n \text{ strongly-postdominates } n \rangle \langle \neg n \text{ strongly-postdominates } n' \rangle \langle n' - as \rightarrow^* n \rangle$
obtain $as' a as''$ **where** $[simp]: as = as' @ a \# as''$ **and** *valid-edge* a
and $\neg n \text{ strongly-postdominates (sourcenode } a)$ **and**
 $n \text{ strongly-postdominates (targetnode } a)$
by $-(erule \text{ strong-postdominate-path-branch})$
from $\langle n \in \text{obs } n' \text{ (PDG-BS } S) \rangle \langle \forall n' \in \text{set}(\text{sourcenodes } as). n' \notin \text{(PDG-BS } S) \rangle$
have $n \notin \text{set}(\text{sourcenodes } (a \# as''))$
by(*fastforce elim:obs.cases simp:sourcenodes-def*)
from $\langle n' - as \rightarrow^* n \rangle$ **have** *sourcenode* $a - a \# as'' \rightarrow^* n$
by(*fastforce dest:path-split-second*)
from $\langle \neg n \text{ strongly-postdominates (sourcenode } a) \rangle \langle \text{valid-edge } a \rangle \langle \text{valid-node } n \rangle$
obtain a' **where** *sourcenode* $a' = \text{sourcenode } a$
and *valid-edge* a' **and** $\neg n \text{ strongly-postdominates (targetnode } a')$
by(*fastforce elim:not-strong-postdominate-predecessor-successor*)
with $\langle n \text{ strongly-postdominates (targetnode } a) \rangle \langle n \notin \text{set}(\text{sourcenodes } (a \# as'')) \rangle$
 $\langle \text{sourcenode } a - a \# as'' \rightarrow^* n \rangle$
have *sourcenode* a *weakly controls* n
by(*fastforce simp:weak-control-dependence-def*)
with $\langle n \in \text{obs } n' \text{ (PDG-BS } S) \rangle$ **have** *sourcenode* $a \in \text{(PDG-BS } S)$
by(*fastforce intro:cd-closed PDG-cdep-edge elim:obs.cases*)
with $\langle \forall n' \in \text{set}(\text{sourcenodes } as). n' \notin \text{(PDG-BS } S) \rangle$
show *False* **by**(*simp add:sourcenodes-def*)
qed

lemma *obs-singleton*: $(\exists m. \text{obs } n \text{ (PDG-BS } S) = \{m\}) \vee \text{obs } n \text{ (PDG-BS } S) = \{\}$
proof(*rule ccontr*)
assume $\neg ((\exists m. \text{obs } n \text{ (PDG-BS } S) = \{m\}) \vee \text{obs } n \text{ (PDG-BS } S) = \{\})$
hence $\exists nx \ nx'. nx \in \text{obs } n \text{ (PDG-BS } S) \wedge nx' \in \text{obs } n \text{ (PDG-BS } S) \wedge$
 $nx \neq nx'$ **by** *auto*
then obtain $nx \ nx'$ **where** $nx \in \text{obs } n \text{ (PDG-BS } S)$ **and** $nx' \in \text{obs } n \text{ (PDG-BS } S)$
and $nx \neq nx'$ **by** *auto*
from $\langle nx \in \text{obs } n \text{ (PDG-BS } S) \rangle$ **obtain** as **where** $n - as \rightarrow^* nx$
and $\forall n' \in \text{set}(\text{sourcenodes } as). n' \notin \text{(PDG-BS } S)$ **and** $nx \in \text{(PDG-BS } S)$

```

    by(erule obsE)
  from  $\langle n - as \rightarrow^* nx \rangle$  have valid-node nx by(fastforce dest:path-valid-node)
  with  $\langle nx \in (PDG-BS S) \rangle$  have obs nx (PDG-BS S) = {nx} by -(rule n-in-obs)
  with  $\langle n - as \rightarrow^* nx \rangle \langle nx \in obs\ n\ (PDG-BS\ S) \rangle \langle nx' \in obs\ n\ (PDG-BS\ S) \rangle \langle nx \neq nx' \rangle$ 
  have  $as \neq []$  by(fastforce elim:path.cases)
  with  $\langle n - as \rightarrow^* nx \rangle \langle nx \in obs\ n\ (PDG-BS\ S) \rangle \langle nx' \in obs\ n\ (PDG-BS\ S) \rangle$ 
     $\langle nx \neq nx' \rangle \langle obs\ nx\ (PDG-BS\ S) = \{nx\} \rangle \langle \forall n' \in set(sourcenodes\ as). n' \notin (PDG-BS\ S) \rangle$ 
  have  $\exists a\ as'\ as''. n - as' \rightarrow^* sourcenode\ a \wedge targetnode\ a - as'' \rightarrow^* nx \wedge$ 
     $valid-edge\ a \wedge as = as' @ a \# as'' \wedge$ 
     $obs\ (targetnode\ a)\ (PDG-BS\ S) = \{nx\} \wedge$ 
     $(\neg (\exists m. obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{m\} \vee$ 
     $obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{\}))$ 
  proof(induct arbitrary:nx' rule:path.induct)
  case (Cons-path n'' as n' a n)
  note [simp] =  $\langle sourcenode\ a = n \rangle [THEN\ sym] \langle targetnode\ a = n' \rangle [THEN\ sym]$ 
  note more-than-one =  $\langle n' \in obs\ n\ (PDG-BS\ S) \rangle \langle nx' \in obs\ n\ (PDG-BS\ S) \rangle$ 
   $\langle n' \neq nx' \rangle$ 
  note IH =  $\langle \bigwedge nx'. \llbracket n' \in obs\ n''\ (PDG-BS\ S); nx' \in obs\ n''\ (PDG-BS\ S); n' \neq nx' \rrbracket$ 
     $obs\ n'\ (PDG-BS\ S) = \{n'\}; \forall n' \in set\ (sourcenodes\ as). n' \notin (PDG-BS\ S); as \neq []$ 
     $\implies \exists a\ as'\ as''. n'' - as' \rightarrow^* sourcenode\ a \wedge targetnode\ a - as'' \rightarrow^* n' \wedge$ 
     $valid-edge\ a \wedge as = as' @ a \# as'' \wedge obs\ (targetnode\ a)\ (PDG-BS\ S) = \{n'\} \wedge$ 
     $(\neg (\exists m. obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{m\} \vee$ 
     $obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{\}))$ 
  from  $\langle \forall n' \in set\ (sourcenodes\ (a \# as)). n' \notin (PDG-BS\ S) \rangle$ 
  have  $\forall n' \in set\ (sourcenodes\ as). n' \notin (PDG-BS\ S)$  and  $sourcenode\ a \notin (PDG-BS\ S)$ 
  by(simp-all add:sourcenodes-def)
  show ?case
  proof(cases as = [])
  case True
  with  $\langle n'' - as \rightarrow^* n' \rangle$  have [simp]:  $n' = n''$  by(fastforce elim:path.cases)
  from more-than-one
  have  $\neg (\exists m. obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{m\} \vee$ 
     $obs\ (sourcenode\ a)\ (PDG-BS\ S) = \{\})$ 
  by auto
  with  $\langle obs\ n'\ (PDG-BS\ S) = \{n'\} \rangle$  True  $\langle valid-edge\ a \rangle$  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  $as \neq []$  .
  from  $\langle n'' - as \rightarrow^* n' \rangle \langle \forall n' \in set\ (sourcenodes\ as). n' \notin (PDG-BS\ S) \rangle$ 
  have  $obs\ n'\ (PDG-BS\ S) \subseteq obs\ n''\ (PDG-BS\ S)$  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) = {n}
  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  $\text{obs}(\text{targetnode } a) (PDG\text{-}BS\ S) \subseteq \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S)$ 
  by (rule edge-obs-subset)
with  $\langle \text{obs}(\text{targetnode } a) (PDG\text{-}BS\ S) = \{nx\} \rangle$ 
have  $nx \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S)$  by simp
with more-than-one obtain  $m$  where  $m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S)$ 
  and  $nx \neq m$  by auto
from  $\langle m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$ 
have valid-node  $m$  by (fastforce dest:in-obs-valid)
from  $\langle \text{obs}(\text{targetnode } a) (PDG\text{-}BS\ S) = \{nx\} \rangle$  have valid-node  $nx$ 
  by (fastforce dest:in-obs-valid)
show False
proof (cases  $m$  strongly-postdominates (sourcenode  $a$ ))
  case True
    with  $\langle nx \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle \langle m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$ 
      have  $m$  strongly-postdominates  $nx$ 
        by (fastforce intro:strong-postdominate-path-targetnode elim:obs.cases)
      with  $\langle nx \neq m \rangle$  have  $\neg nx$  strongly-postdominates  $m$ 
        by (fastforce dest:strong-postdominate-antisym)
      have  $(\text{-Exit-}) - \square \rightarrow^* (\text{-Exit-})$  by (fastforce intro:empty-path)
      with  $\langle m$  strongly-postdominates  $nx \rangle$  have  $nx \neq (\text{-Exit-})$ 
        by (fastforce simp:strong-postdominate-def sourcenodes-def postdominate-def)
      have  $\neg nx$  strongly-postdominates (sourcenode  $a$ )
        proof (rule ccontr)
          assume  $\neg \neg nx$  strongly-postdominates sourcenode  $a$ 
          hence  $nx$  strongly-postdominates sourcenode  $a$  by simp
          from  $\langle m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle \langle nx \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$ 
            obtain  $asx'$  where sourcenode  $a - asx' \rightarrow^* m$  and  $nx \notin \text{set}(\text{sourcenodes } asx')$ 
              by (fastforce elim:obs.cases)
            with  $\langle nx$  strongly-postdominates sourcenode  $a \rangle$  have  $nx$  strongly-postdominates
               $m$ 
                by (rule strong-postdominate-path-targetnode)
              with  $\langle \neg nx$  strongly-postdominates  $m \rangle$  show False by simp
          qed
      with  $\langle nx \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle \langle \text{valid-node } nx \rangle \langle nx \neq (\text{-Exit-}) \rangle$ 
        show False by (fastforce dest:obs-strong-postdominate)
    next
      case False
      show False
      proof (cases  $m = \text{Exit}$ )
        case True
          from  $\langle m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle \langle nx \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle$ 
            obtain  $xs$  where sourcenode  $a - xs \rightarrow^* m$  and  $nx \notin \text{set}(\text{sourcenodes } xs)$ 
              by (fastforce elim:obsE)
            obtain  $x' xs'$  where  $[simp]: xs = x' \# xs'$ 
              proof (cases  $xs$ )
                case Nil

```

with $\langle \text{sourcenode } a - xs \rightarrow^* m \rangle$ **have** $[simp]: \text{sourcenode } a = m$ **by** *fastforce*
with $\langle m \in \text{obs } (\text{sourcenode } a) \text{ (PDG-BS } S) \rangle$
have $m \in (\text{PDG-BS } S)$ **by** $(\text{fastforce elim:obsE})$
with $\langle \text{valid-node } m \rangle$ **have** $\text{obs } m \text{ (PDG-BS } S) = \{m\}$
by (rule n-in-obs)
with $\langle nx \in \text{obs } (\text{sourcenode } a) \text{ (PDG-BS } S) \rangle$ $\langle nx \neq m \rangle$ **have** *False*
by *fastforce*
thus *?thesis* **by** *simp*
qed blast
from $\langle \text{sourcenode } a - xs \rightarrow^* m \rangle$ **have** $\text{sourcenode } a = \text{sourcenode } x'$
and $\text{valid-edge } x' \text{ and targetnode } x' - xs' \rightarrow^* m$
by $(\text{auto elim:path-split-Cons})$
from $\langle \text{targetnode } x' - xs' \rightarrow^* m \rangle$ $\langle nx \notin \text{set}(\text{sourcenodes } xs) \rangle$ $\langle \text{valid-edge } x' \rangle$
 $\langle \text{valid-node } m \rangle$ *True*
have $\neg nx \text{ strongly-postdominates } (\text{targetnode } x')$
by $(\text{fastforce simp:strong-postdominate-def postdominate-def sourcenodes-def})$
from $\langle nx \neq m \rangle$ *True* **have** $nx \neq (-\text{Exit-})$ **by** *simp*
with $\langle \text{obs } (\text{targetnode } a) \text{ (PDG-BS } S) = \{nx\} \rangle$
have $nx \text{ strongly-postdominates } (\text{targetnode } a)$
by $(\text{fastforce intro:obs-strong-postdominate})$
from $\langle \text{obs } (\text{targetnode } a) \text{ (PDG-BS } S) = \{nx\} \rangle$
obtain *ys* **where** $\text{targetnode } a - ys \rightarrow^* nx$
and $\forall nx' \in \text{set}(\text{sourcenodes } ys). nx' \notin (\text{PDG-BS } S)$
and $nx \in (\text{PDG-BS } S)$ **by** $(\text{fastforce elim:obsE})$
hence $nx \notin \text{set}(\text{sourcenodes } ys)$ **by** *fastforce*
have $\text{sourcenode } a \neq nx$
proof
assume $\text{sourcenode } a = nx$
from $\langle nx \in \text{obs } (\text{sourcenode } a) \text{ (PDG-BS } S) \rangle$
have $nx \in (\text{PDG-BS } S)$ **by** (erule obsE)
with $\langle \text{valid-node } nx \rangle$ **have** $\text{obs } nx \text{ (PDG-BS } S) = \{nx\}$ **by** (erule n-in-obs)
with $\langle \text{sourcenode } a = nx \rangle$ $\langle m \in \text{obs } (\text{sourcenode } a) \text{ (PDG-BS } S) \rangle$
 $\langle nx \neq m \rangle$ **show** *False* **by** *fastforce*
qed
with $\langle nx \notin \text{set}(\text{sourcenodes } ys) \rangle$ **have** $nx \notin \text{set}(\text{sourcenodes } (a \# ys))$
by $(\text{fastforce simp:sourcenodes-def})$
from $\langle \text{valid-edge } a \rangle$ $\langle \text{targetnode } a - ys \rightarrow^* nx \rangle$
have $\text{sourcenode } a - a \# ys \rightarrow^* nx$ **by** $(\text{fastforce intro:Cons-path})$
from $\langle \text{sourcenode } a - a \# ys \rightarrow^* nx \rangle$ $\langle nx \notin \text{set}(\text{sourcenodes } (a \# ys)) \rangle$
 $\langle nx \text{ strongly-postdominates } (\text{targetnode } a) \rangle$ $\langle \text{valid-edge } x' \rangle$
 $\langle \neg nx \text{ strongly-postdominates } (\text{targetnode } x') \rangle$ $\langle \text{sourcenode } a = \text{sourcenode } x' \rangle$
have $(\text{sourcenode } a) \text{ weakly controls } nx$
by $(\text{fastforce simp:weak-control-dependence-def})$
with $\langle nx \in (\text{PDG-BS } S) \rangle$ **have** $\text{sourcenode } a \in (\text{PDG-BS } S)$
by (rule cd-closed)
with $\langle \text{valid-edge } a \rangle$ **have** $\text{obs } (\text{sourcenode } a) \text{ (PDG-BS } S) = \{\text{sourcenode } a\}$
by $(\text{fastforce intro!:n-in-obs})$
with $\langle m \in \text{obs } (\text{sourcenode } a) \text{ (PDG-BS } S) \rangle$

```

       $\langle nx \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle \langle nx \neq m \rangle$ 
    show False by simp
  next
    case False
    with  $\langle m \in \text{obs}(\text{sourcenode } a) (PDG\text{-}BS\ S) \rangle \langle \text{valid-node } m \rangle$ 
       $\langle \neg m \text{ strongly-postdominates sourcenode } a \rangle$ 
    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  $n \in PDG\text{-}BS\ S$ 
  thus valid-node  $n$  by (rule PDG-BS-valid-node)
next
  fix  $n\ S$  assume valid-node  $n$  and  $n \in S$ 
  thus  $n \in PDG\text{-}BS\ S$  by (fastforce intro:PDG-path-Nil simp:PDG-BS-def)
next
  fix  $n'\ S\ n\ V$  assume  $n' \in PDG\text{-}BS\ S$  and  $n$  influences  $V$  in  $n'$ 
  thus  $n \in PDG\text{-}BS\ S$ 
    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:split-if-asm)
next
  fix  $n\ S$ 
  have  $(\exists m. \text{obs } n (PDG\text{-}BS\ S) = \{m\}) \vee \text{obs } n (PDG\text{-}BS\ S) = \{\}$ 
    by (rule obs-singleton)
  thus finite (obs  $n (PDG\text{-}BS\ S)$ ) by fastforce
next
  fix  $n\ S$ 
  have  $(\exists m. \text{obs } n (PDG\text{-}BS\ S) = \{m\}) \vee \text{obs } n (PDG\text{-}BS\ S) = \{\}$ 
    by (rule obs-singleton)
  thus card (obs  $n (PDG\text{-}BS\ S)$ )  $\leq 1$  by fastforce
qed
end

```

3.8.3 Weak order dependence

context *CFG-wf* begin

lemma *obs-singleton*:

shows $(\exists m. \text{obs } n (\text{wod-backward-slice } S) = \{m\}) \vee$
 $\text{obs } n (\text{wod-backward-slice } S) = \{\}$

proof(*rule ccontr*)
let $?WOD-BS = \text{wod-backward-slice } S$
assume $\neg ((\exists m. \text{obs } n \text{ ?WOD-BS} = \{m\}) \vee \text{obs } n \text{ ?WOD-BS} = \{\})$
hence $\exists nx \ nx'. nx \in \text{obs } n \text{ ?WOD-BS} \wedge nx' \in \text{obs } n \text{ ?WOD-BS} \wedge$
 $nx \neq nx' \text{ by auto}$
then obtain $nx \ nx'$ **where** $nx \in \text{obs } n \text{ ?WOD-BS}$ **and** $nx' \in \text{obs } n \text{ ?WOD-BS}$
and $nx \neq nx' \text{ by auto}$
from $\langle nx \in \text{obs } n \text{ ?WOD-BS} \rangle$ **obtain** as **where** $n - as \rightarrow^* nx$
and $\forall n' \in \text{set}(\text{sourcenodes } as). n' \notin \text{?WOD-BS}$ **and** $nx \in \text{?WOD-BS}$
by(*erule obsE*)
from $\langle n - as \rightarrow^* nx \rangle$ **have** *valid-node* nx **by**(*fastforce dest:path-valid-node*)
with $\langle nx \in \text{?WOD-BS} \rangle$ **have** $\text{obs } nx \text{ ?WOD-BS} = \{nx\}$ **by** $\neg(\text{rule } n\text{-in-obs})$
with $\langle n - as \rightarrow^* nx \rangle \langle nx \in \text{obs } n \text{ ?WOD-BS} \rangle \langle nx' \in \text{obs } n \text{ ?WOD-BS} \rangle \langle nx \neq nx' \rangle$

have $as \neq []$ **by**(*fastforce elim:path.cases*)
with $\langle n - as \rightarrow^* nx \rangle \langle nx \in \text{obs } n \text{ ?WOD-BS} \rangle \langle nx' \in \text{obs } n \text{ ?WOD-BS} \rangle \langle nx \neq nx' \rangle$

 $\langle \text{obs } nx \text{ ?WOD-BS} = \{nx\} \rangle \langle \forall n' \in \text{set}(\text{sourcenodes } as). n' \notin \text{?WOD-BS} \rangle$
have $\exists a \ as' \ as''. n - as' \rightarrow^* \text{sourcenode } a \wedge \text{targetnode } a - as'' \rightarrow^* nx \wedge$
 $\text{valid-edge } a \wedge as = as' @ a \# as'' \wedge$
 $\text{obs } (\text{targetnode } a) \text{ ?WOD-BS} = \{nx\} \wedge$
 $(\neg (\exists m. \text{obs } (\text{sourcenode } a) \text{ ?WOD-BS} = \{m\}) \vee$
 $\text{obs } (\text{sourcenode } a) \text{ ?WOD-BS} = \{\}))$
proof(*induct arbitrary:nx' rule:path.induct*)
case (*Cons-path* n'' as' a n)
note [*simp*] = $\langle \text{sourcenode } a = n \rangle [THEN \text{sym}] \langle \text{targetnode } a = n'' \rangle [THEN \text{sym}]$
note *more-than-one* = $\langle n' \in \text{obs } n \text{ (?WOD-BS)} \rangle \langle nx' \in \text{obs } n \text{ (?WOD-BS)} \rangle$
 $\langle n' \neq nx' \rangle$
note *IH* = $\langle \bigwedge nx'. \llbracket n' \in \text{obs } n'' \text{ (?WOD-BS)}; nx' \in \text{obs } n'' \text{ (?WOD-BS)}; n' \neq$
 $nx' \rrbracket;$
 $\text{obs } n' \text{ (?WOD-BS)} = \{n'\}; \forall n' \in \text{set } (\text{sourcenodes } as). n' \notin \text{(?WOD-BS)}; as$
 $\neq [] \rrbracket$
 $\implies \exists a \ as' \ as''. n'' - as' \rightarrow^* \text{sourcenode } a \wedge \text{targetnode } a - as'' \rightarrow^* n' \wedge$
 $\text{valid-edge } a \wedge as = as' @ a \# as'' \wedge \text{obs } (\text{targetnode } a) \text{ (?WOD-BS)} = \{n'\} \wedge$
 $(\neg (\exists m. \text{obs } (\text{sourcenode } a) \text{ (?WOD-BS)} = \{m\}) \vee$
 $\text{obs } (\text{sourcenode } a) \text{ (?WOD-BS)} = \{\})) \rangle$
from $\langle \forall n' \in \text{set } (\text{sourcenodes } (a \# as)). n' \notin \text{(?WOD-BS)} \rangle$
have $\forall n' \in \text{set } (\text{sourcenodes } as). n' \notin \text{(?WOD-BS)}$ **and** $\text{sourcenode } a \notin \text{(?WOD-BS)}$
by(*simp-all add:sourcenodes-def*)
show *?case*
proof(*cases as = []*)
case *True*
with $\langle n'' - as \rightarrow^* n' \rangle$ **have** [*simp*]: $n' = n''$ **by**(*fastforce elim:path.cases*)
from *more-than-one*
have $\neg (\exists m. \text{obs } (\text{sourcenode } a) \text{ (?WOD-BS)} = \{m\}) \vee$
 $\text{obs } (\text{sourcenode } a) \text{ (?WOD-BS)} = \{\}$
by *auto*
with $\langle \text{obs } n' \text{ (?WOD-BS)} = \{n'\} \rangle \text{True } \langle \text{valid-edge } a \rangle$ **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  $as \neq []$  .
from  $\langle n'' - as \rightarrow^* n' \rangle \langle \forall n' \in \text{set } (\text{sourcenodes } as). n' \notin (?WOD-BS) \rangle$ 
have  $\text{obs } n' (?WOD-BS) \subseteq \text{obs } n'' (?WOD-BS)$  by(rule path-obs-subset)
show ?thesis
proof(cases  $\text{obs } n' (?WOD-BS) = \text{obs } n'' (?WOD-BS)$ )
  case True
  with  $\langle n'' - as \rightarrow^* n' \rangle \langle \text{valid-edge } a \rangle \langle \text{obs } n' (?WOD-BS) = \{n'\} \rangle \text{ more-than-one}$ 
  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  $\langle \text{obs } n' (?WOD-BS) \subseteq \text{obs } n'' (?WOD-BS) \rangle$ 
  have  $\text{obs } n' (?WOD-BS) \subset \text{obs } n'' (?WOD-BS)$  by simp
  with  $\langle \text{obs } n' (?WOD-BS) = \{n'\} \rangle$  obtain  $ni$  where  $n' \in \text{obs } n'' (?WOD-BS)$ 
    and  $ni \in \text{obs } n'' (?WOD-BS)$  and  $n' \neq ni$  by auto
  from IH[OF this  $\langle \text{obs } n' (?WOD-BS) = \{n'\} \rangle$ 
     $\langle \forall n' \in \text{set } (\text{sourcenodes } as). n' \notin (?WOD-BS) \rangle \langle as \neq [] \rangle$ ] obtain  $a' as' as''$ 
    where  $n'' - as' \rightarrow^* \text{sourcenode } a'$  and  $\text{targetnode } a' - as'' \rightarrow^* n'$ 
    and  $\text{valid-edge } a'$  and  $[simp]: as = as' \# a' \# as''$ 
    and  $\text{obs } (\text{targetnode } a') (?WOD-BS) = \{n'\}$ 
    and  $\text{more-than-one}' : \neg (\exists m. \text{obs } (\text{sourcenode } a') (?WOD-BS) = \{m\} \vee$ 
       $\text{obs } (\text{sourcenode } a') (?WOD-BS) = \{\})$ 
    by blast
  from  $\langle n'' - as' \rightarrow^* \text{sourcenode } a' \rangle \langle \text{valid-edge } a' \rangle$ 
  have  $n - a \# as' \rightarrow^* \text{sourcenode } a'$  by(fastforce intro:path.Cons-path)
  with  $\langle \text{targetnode } a' - as'' \rightarrow^* n' \rangle \langle \text{obs } (\text{targetnode } a') (?WOD-BS) = \{n'\} \rangle$ 
     $\text{more-than-one}' \langle \text{valid-edge } a' \rangle$  show ?thesis
    apply(rule-tac x=a' in exI)
    apply(rule-tac x=a#a' in exI)
    apply(rule-tac x=as'' in exI)
    by fastforce
qed
qed
qed simp
then obtain  $a as' as''$  where  $\text{valid-edge } a$ 
  and  $\text{obs } (\text{targetnode } a) (?WOD-BS) = \{nx\}$ 
  and  $\text{more-than-one}' : \neg (\exists m. \text{obs } (\text{sourcenode } a) (?WOD-BS) = \{m\} \vee$ 
     $\text{obs } (\text{sourcenode } a) (?WOD-BS) = \{\})$ 
  by blast
have  $\text{sourcenode } a \notin (?WOD-BS)$ 
proof(rule ccontr)

```

```

assume  $\neg \text{sourcenode } a \notin ?WOD\text{-}BS$ 
hence  $\text{sourcenode } a \in ?WOD\text{-}BS$  by simp
with  $\langle \text{valid-edge } a \rangle$  have  $\text{obs } (\text{sourcenode } a) (?WOD\text{-}BS) = \{\text{sourcenode } a\}$ 
by (fastforce intro!:n-in-obs)
with more-than-one show False by simp
qed
with  $\langle \text{valid-edge } a \rangle$ 
have  $\text{obs } (\text{targetnode } a) (?WOD\text{-}BS) \subseteq \text{obs } (\text{sourcenode } a) (?WOD\text{-}BS)$ 
by (rule edge-obs-subset)
with  $\langle \text{obs } (\text{targetnode } a) (?WOD\text{-}BS) = \{nx\} \rangle$ 
have  $nx \in \text{obs } (\text{sourcenode } a) (?WOD\text{-}BS)$  by simp
with more-than-one obtain  $m$  where  $m \in \text{obs } (\text{sourcenode } a) (?WOD\text{-}BS)$ 
and  $nx \neq m$  by auto
with  $\langle nx \in \text{obs } (\text{sourcenode } a) (?WOD\text{-}BS) \rangle$  obtain  $as2$ 
where  $\text{sourcenode } a - as2 \rightarrow^* m$  and  $nx \notin \text{set}(\text{sourcenodes } as2)$ 
by (fastforce elim:obsE)
from  $\langle nx \in \text{obs } (\text{sourcenode } a) (?WOD\text{-}BS) \rangle \langle m \in \text{obs } (\text{sourcenode } a) (?WOD\text{-}BS) \rangle$ 

obtain  $as1$  where  $\text{sourcenode } a - as1 \rightarrow^* nx$  and  $m \notin \text{set}(\text{sourcenodes } as1)$ 
by (fastforce elim:obsE)
from  $\langle \text{obs } (\text{targetnode } a) (?WOD\text{-}BS) = \{nx\} \rangle$  obtain  $asx$ 
where  $\text{targetnode } a - asx \rightarrow^* nx$  by (fastforce elim:obsE)
have  $\forall asx'. \text{targetnode } a - asx' \rightarrow^* m \longrightarrow nx \in \text{set}(\text{sourcenodes } asx')$ 
proof (rule ccontr)
assume  $\neg (\forall asx'. \text{targetnode } a - asx' \rightarrow^* m \longrightarrow nx \in \text{set}(\text{sourcenodes } asx'))$ 
then obtain  $asx'$  where  $\text{targetnode } a - asx' \rightarrow^* m$  and  $nx \notin \text{set}(\text{sourcenodes } asx')$ 
by blast
show False
proof (cases  $\forall nx \in \text{set}(\text{sourcenodes } asx'). nx \notin ?WOD\text{-}BS$ )
case True
with  $\langle \text{targetnode } a - asx' \rightarrow^* m \rangle \langle m \in \text{obs } (\text{sourcenode } a) (?WOD\text{-}BS) \rangle$ 
have  $m \in \text{obs } (\text{targetnode } a) ?WOD\text{-}BS$  by (fastforce intro:obs-elem elim:obsE)
with  $\langle nx \neq m \rangle \langle \text{obs } (\text{targetnode } a) (?WOD\text{-}BS) = \{nx\} \rangle$  show False by simp
next
case False
hence  $\exists nx \in \text{set}(\text{sourcenodes } asx'). nx \in ?WOD\text{-}BS$  by blast
then obtain  $nx' ns ns'$  where  $\text{sourcenodes } asx' = ns @ nx' \# ns'$  and  $nx' \in ?WOD\text{-}BS$ 
and  $\forall nx \in \text{set } ns. nx \notin ?WOD\text{-}BS$  by (fastforce elim!:split-list-first-propE)
from  $\langle \text{sourcenodes } asx' = ns @ nx' \# ns' \rangle$  obtain  $ax ai ai'$ 
where  $[simp]: asx' = ai @ ax \# ai' \quad ns = \text{sourcenodes } ai \quad nx' = \text{sourcenode } ax$ 
by (fastforce elim:map-append-append-maps simp:sourcenodes-def)
from  $\langle \text{targetnode } a - asx' \rightarrow^* m \rangle$  have  $\text{targetnode } a - ai \rightarrow^* \text{sourcenode } ax$ 
by (fastforce dest:path-split)
with  $\langle nx' \in ?WOD\text{-}BS \rangle \langle \forall nx \in \text{set } ns. nx \notin ?WOD\text{-}BS \rangle$ 
have  $nx' \in \text{obs } (\text{targetnode } a) ?WOD\text{-}BS$  by (fastforce intro:obs-elem)
with  $\langle \text{obs } (\text{targetnode } a) (?WOD\text{-}BS) = \{nx\} \rangle$  have  $nx' = nx$  by simp
with  $\langle nx \notin \text{set}(\text{sourcenodes } asx') \rangle$  show False by (simp add:sourcenodes-def)

```

```

qed
qed
with  $\langle nx \neq m \rangle \langle \text{sourcenode } a - as1 \rightarrow^* nx \rangle \langle m \notin \text{set}(\text{sourcenodes } as1) \rangle$ 
 $\langle \text{sourcenode } a - as2 \rightarrow^* m \rangle \langle nx \notin \text{set}(\text{sourcenodes } as2) \rangle \langle \text{valid-edge } a \rangle$ 
 $\langle \text{targetnode } a - asx \rightarrow^* nx \rangle$ 
have  $\text{sourcenode } a \rightarrow_{\text{wod}} nx, m$  by (simp add:wod-def,blast)
with  $\langle nx \in \text{obs}(\text{sourcenode } a) \rangle \langle ?WOD-BS \rangle \langle m \in \text{obs}(\text{sourcenode } a) \rangle \langle ?WOD-BS \rangle$ 

have  $\text{sourcenode } a \in ?WOD-BS$  by (fastforce elim:cd-closed elim:obsE)
with  $\langle \text{sourcenode } a \notin ?WOD-BS \rangle$  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** $n \in \text{wod-backward-slice } S$

thus $\text{valid-node } n$ **by** (*rule wod-backward-slice-valid-node*)

next

fix $n S$ **assume** $\text{valid-node } n$ **and** $n \in S$

thus $n \in \text{wod-backward-slice } S$ **by** (*rule refl*)

next

fix $n' S n V$ **assume** $n' \in \text{wod-backward-slice } S$ n *influences* V *in* n'

thus $n \in \text{wod-backward-slice } S$

by $\neg(\text{rule dd-closed})$

next

fix $n S$

have $(\exists m. \text{obs } n (\text{wod-backward-slice } S) = \{m\}) \vee$

$\text{obs } n (\text{wod-backward-slice } S) = \{\}$

by (*rule obs-singleton*)

thus $\text{finite } (\text{obs } n (\text{wod-backward-slice } S))$ **by** *fastforce*

next

fix $n S$

have $(\exists m. \text{obs } n (\text{wod-backward-slice } S) = \{m\}) \vee \text{obs } n (\text{wod-backward-slice } S)$

$= \{\}$

by (*rule obs-singleton*)

thus $\text{card } (\text{obs } n (\text{wod-backward-slice } S)) \leq 1$ **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 $n \text{ controls}_s n'$ **shows** $n \longrightarrow_{\text{wod}} n', (-\text{Exit-})$

proof –

from $\langle n \text{ controls}_s n' \rangle$ **obtain** $as \ a \ a' \ as'$ **where** $as = a \# as'$

and $n' \notin \text{set}(\text{sourcenodes } as)$ **and** $n - as \rightarrow^* n'$

and $n' \text{ postdominates } (\text{targetnode } a)$

and *valid-edge* a' **and** *sourcenode* $a' = n$

and $\neg n' \text{ postdominates } (\text{targetnode } a')$

by (*auto simp: standard-control-dependence-def*)

from $\langle n - as \rightarrow^* n' \rangle \langle as = a \# as' \rangle$ **have** *sourcenode* $a = n$ **by** (*auto elim: path.cases*)

from $\langle n - as \rightarrow^* n' \rangle \langle as = a \# as' \rangle \langle n' \notin \text{set}(\text{sourcenodes } as) \rangle$ **have** $n \neq n'$

by (*induct rule: path.induct, auto simp: sourcenodes-def*)

from $\langle n - as \rightarrow^* n' \rangle \langle as = a \# as' \rangle$ **have** *valid-edge* a

by (*auto elim: path.cases*)

from $\langle n \text{ controls}_s n' \rangle$ **have** $n' \neq (-\text{Exit-})$

by (*fastforce dest: Exit-not-standard-control-dependent*)

from $\langle n - as \rightarrow^* n' \rangle$ **have** $(-\text{Exit-}) \notin \text{set}(\text{sourcenodes } as)$ **by** *fastforce*

from $\langle n - as \rightarrow^* n' \rangle$ **have** *valid-node* n **and** *valid-node* n'

by (*auto dest: path-valid-node*)

with $\langle \neg n' \text{ postdominates } (\text{targetnode } a') \rangle \langle \text{valid-edge } a' \rangle$

obtain asx **where** $\text{targetnode } a' - asx \rightarrow^* (-\text{Exit-})$ **and** $n' \notin \text{set}(\text{sourcenodes } asx)$

by (*auto simp: postdominate-def*)

with $\langle \text{valid-edge } a' \rangle \langle \text{sourcenode } a' = n \rangle$ **have** $n - a' \# asx \rightarrow^* (-\text{Exit-})$

by (*fastforce intro: Cons-path*)

with $\langle n \neq n' \rangle \langle \text{sourcenode } a' = n \rangle \langle n' \notin \text{set}(\text{sourcenodes } asx) \rangle$

have $n' \notin \text{set}(\text{sourcenodes } (a' \# asx))$ **by** (*simp add: sourcenodes-def*)

from $\langle n' \text{ postdominates } (\text{targetnode } a) \rangle$

obtain asx' **where** $\text{targetnode } a - asx' \rightarrow^* n'$ **by** (*erule postdominate-implies-path*)

from $\langle n' \text{ postdominates } (\text{targetnode } a) \rangle$

have $\forall as'. \text{targetnode } a - as' \rightarrow^* (-\text{Exit-}) \longrightarrow n' \in \text{set}(\text{sourcenodes } as')$

by (*auto simp: postdominate-def*)

with $\langle n' \neq (-\text{Exit-}) \rangle \langle n - as \rightarrow^* n' \rangle \langle (-\text{Exit-}) \notin \text{set}(\text{sourcenodes } as) \rangle$

$\langle n - a' \# asx \rightarrow^* (-\text{Exit-}) \rangle \langle n' \notin \text{set}(\text{sourcenodes } (a' \# asx)) \rangle$

$\langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = n \rangle \langle \text{targetnode } a - asx' \rightarrow^* n' \rangle$

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 v₁ v₂* = *Some(Bool(v₁ = v₂))*
| *binop And (Bool b₁) (Bool b₂)* = *Some(Bool(b₁ ∧ b₂))*
| *binop Less (Intg i₁) (Intg i₂)* = *Some(Bool(i₁ < i₂))*
| *binop Add (Intg i₁) (Intg i₂)* = *Some(Intg(i₁ + i₂))*

```

| binop Sub (Intg i1) (Intg i2) = Some(Intg(i1 - i2))
| binop bop v1 v2 = None

```

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
| #:(c1;;c2) = #:c1 + #:c2
| #:(if (b) c1 else c2) = #:c1 + #:c2 + 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 (e1⋖bop⋗e2) s =
  (case interpret e1 s of None ⇒ None
   | Some v1 ⇒ (case interpret e2 s of None ⇒ None
                  | Some v2 ⇒ (
                      case binop bop v1 v2 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* $\Rightarrow$ *nat* $\Rightarrow$ *w-node* (- $\oplus$ - 60)
where (- *l* -) $\oplus$ *i* = (- *l* + *i* -)
 | (- *Entry* -) $\oplus$ *i* = (- *Entry* -)
 | (- *Exit* -) $\oplus$ *i* = (- *Exit* -)

lemma *Exit-label-incr* [*dest*]: (- *Exit* -) = *n* $\oplus$ *i* $\Longrightarrow$ *n* = (- *Exit* -)
by(*cases n, auto*)

lemma *label-incr-Exit* [*dest*]: *n* $\oplus$ *i* = (- *Exit* -) $\Longrightarrow$ *n* = (- *Exit* -)
by(*cases n, auto*)

lemma *Entry-label-incr* [*dest*]: (- *Entry* -) = *n* $\oplus$ *i* $\Longrightarrow$ *n* = (- *Entry* -)
by(*cases n, auto*)

lemma *label-incr-Entry* [*dest*]: *n* $\oplus$ *i* = (- *Entry* -) $\Longrightarrow$ *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* $\oplus$ *i* $\Longrightarrow$ *n* = (- (*l* - *i*) -)
by(*cases n, auto*)

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

lemma *label-incr-0* [*dest*]:
 $\llbracket (-0-) = n \oplus i; i > 0 \rrbracket \Longrightarrow \text{False}$
by(*cases n*) *auto*

lemma *label-incr-0-rev* [*dest*]:
 $\llbracket n \oplus i = (-0-); i > 0 \rrbracket \Longrightarrow \text{False}$
by(*cases n*) *auto*

4.2.2 CFG edges

type-synonym $w\text{-edge} = (w\text{-node} \times \text{state edge-kind} \times w\text{-node})$

inductive $\text{While-CFG} :: \text{cmd} \Rightarrow w\text{-node} \Rightarrow \text{state edge-kind} \Rightarrow w\text{-node} \Rightarrow \text{bool}$
 $(- \vdash - \dashrightarrow -)$

where

WCFG-Entry-Exit :

$\text{prog} \vdash (-\text{Entry-}) -(\lambda s. \text{False})_{\sqrt{\rightarrow}} (-\text{Exit-})$

| WCFG-Entry :

$\text{prog} \vdash (-\text{Entry-}) -(\lambda s. \text{True})_{\sqrt{\rightarrow}} (-0-)$

| WCFG-Skip :

$\text{Skip} \vdash (-0-) -\uparrow id \rightarrow (-\text{Exit-})$

| WCFG-LAss :

$V := e \vdash (-0-) -\uparrow (\lambda s. s(V := (\text{interpret } e \ s))) \rightarrow (-1-)$

| WCFG-LAssSkip :

$V := e \vdash (-1-) -\uparrow id \rightarrow (-\text{Exit-})$

| WCFG-SeqFirst :

$\llbracket c_1 \vdash n -et\rightarrow n'; n' \neq (-\text{Exit-}) \rrbracket \implies c_1;;c_2 \vdash n -et\rightarrow n'$

| WCFG-SeqConnect :

$\llbracket c_1 \vdash n -et\rightarrow (-\text{Exit-}); n \neq (-\text{Entry-}) \rrbracket \implies c_1;;c_2 \vdash n -et\rightarrow (-0-) \oplus \# : c_1$

| WCFG-SeqSecond :

$\llbracket c_2 \vdash n -et\rightarrow n'; n \neq (-\text{Entry-}) \rrbracket \implies c_1;;c_2 \vdash n \oplus \# : c_1 -et\rightarrow n' \oplus \# : c_1$

| WCFG-CondTrue :

$\text{if } (b) \ c_1 \text{ else } c_2 \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some true})_{\sqrt{\rightarrow}} (-0-) \oplus 1$

| WCFG-CondFalse :

$\text{if } (b) \ c_1 \text{ else } c_2 \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\sqrt{\rightarrow}} (-0-) \oplus (\# : c_1 + 1)$

| WCFG-CondThen :

$\llbracket c_1 \vdash n -et\rightarrow n'; n \neq (-\text{Entry-}) \rrbracket \implies \text{if } (b) \ c_1 \text{ else } c_2 \vdash n \oplus 1 -et\rightarrow n' \oplus 1$

| WCFG-CondElse :

$\llbracket c_2 \vdash n -et\rightarrow n'; n \neq (-\text{Entry-}) \rrbracket$
 $\implies \text{if } (b) \ c_1 \text{ else } c_2 \vdash n \oplus (\# : c_1 + 1) -et\rightarrow n' \oplus (\# : c_1 + 1)$

| WCFG-WhileTrue :

$\text{while } (b) \ c' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some true})_{\sqrt{\rightarrow}} (-0-) \oplus 2$

| WCFG-WhileFalse :

$while\ (b)\ c' \vdash (-0-) \rightarrow (\lambda s. interpret\ b\ s = Some\ false)_{\checkmark} \rightarrow (-1-)$

| *WCFG-WhileFalseSkip*:

$while\ (b)\ c' \vdash (-1-) \rightarrow \uparrow id \rightarrow (-Exit-)$

| *WCFG-WhileBody*:

$\llbracket c' \vdash n \rightarrow n'; n \neq (-Entry-); n' \neq (-Exit-) \rrbracket$

$\implies while\ (b)\ c' \vdash n \oplus 2 \rightarrow n' \oplus 2$

| *WCFG-WhileBodyExit*:

$\llbracket c' \vdash n \rightarrow (-Exit-); n \neq (-Entry-) \rrbracket \implies while\ (b)\ c' \vdash n \oplus 2 \rightarrow (-0-)$

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

lemmas *WCFG-elim*s = *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*]:

$prog \vdash (-Exit-) \rightarrow n' \implies False$

by(*induct prog n* $\equiv (-Exit-)$ *et n'* *rule:WCFG-induct,auto*)

lemma *WCFG-Entry-no-targetnode* [*dest*]:

$prog \vdash n \rightarrow (-Entry-) \implies False$

by(*induct prog n et n'* $\equiv (-Entry-)$ *rule:WCFG-induct,auto*)

lemma *WCFG-sourcelabel-less-num-nodes*:

$prog \vdash (-l-) \rightarrow n' \implies l < \# : prog$

proof(*induct prog (-l-) et n'* *arbitrary:l rule:WCFG-induct*)

case (*WCFG-SeqFirst* *c*₁ *et n'* *c*₂)

from $\langle l < \# : c_1 \rangle$ **show** *?case by simp*

next

case (*WCFG-SeqConnect* *c*₁ *et c*₂)

from $\langle l < \# : c_1 \rangle$ **show** *?case by simp*

next

case (*WCFG-SeqSecond* *c*₂ *n et n'* *c*₁)

note $IH = \langle \bigwedge l. n = (-l-) \implies l < \# : c_2 \rangle$

from $\langle n \oplus \# : c_1 = (-l-) \rangle$ **obtain** *l'* **where** $n = (-l'-)$ **by**(*cases n*) *auto*

from $IH[OF\ this]$ **have** $l' < \# : c_2$.

with $\langle n \oplus \# : c_1 = (-l-) \rangle \langle n = (-l'-) \rangle$ **show** *?case by simp*

next

case (*WCFG-CondThen* *c*₁ *n et n'* *b c*₂)

note $IH = \langle \bigwedge l. n = (-l-) \implies l < \# : c_1 \rangle$

from $\langle n \oplus 1 = (-l-) \rangle$ **obtain** *l'* **where** $n = (-l'-)$ **by**(*cases n*) *auto*

from $IH[OF\ this]$ **have** $l' < \# : c_1$.

with $\langle n \oplus 1 = (-l-) \rangle \langle n = (-l'-) \rangle$ **show** *?case by simp*

next

```

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

```

lemma *WCFG-targetlabel-less-num-nodes*:

```

prog  $\vdash n -et\rightarrow (- l -) \implies l < \# : prog$ 
proof(induct prog n et (- l -) arbitrary:l rule:WCFG-induct)
case (WCFG-SeqFirst  $c_1$   $n$  et  $c_2$ )
from  $\langle l < \# : c_1 \rangle$  show ?case by simp
next
case (WCFG-SeqSecond  $c_2$   $n$  et  $n'$   $c_1$ )
note  $IH = \langle \bigwedge l. n' = (- l -) \implies l < \# : c_2 \rangle$ 
from  $\langle n' \oplus \# : c_1 = (- l -) \rangle$  obtain  $l'$  where  $n' = (- l' -)$  by(cases  $n'$ ) auto
from  $IH[OF \text{ this}]$  have  $l' < \# : c_2$  .
with  $\langle n' \oplus \# : c_1 = (- l -) \rangle \langle n' = (- l' -) \rangle$  show ?case by simp
next
case (WCFG-CondThen  $c_1$   $n$  et  $n'$   $b$   $c_2$ )
note  $IH = \langle \bigwedge l. n' = (- l -) \implies l < \# : c_1 \rangle$ 
from  $\langle n' \oplus 1 = (- l -) \rangle$  obtain  $l'$  where  $n' = (- l' -)$  by(cases  $n'$ ) auto
from  $IH[OF \text{ this}]$  have  $l' < \# : c_1$  .
with  $\langle n' \oplus 1 = (- l -) \rangle \langle n' = (- l' -) \rangle$  show ?case by simp
next
case (WCFG-CondElse  $c_2$   $n$  et  $n'$   $b$   $c_1$ )
note  $IH = \langle \bigwedge l. n' = (- l -) \implies l < \# : c_2 \rangle$ 
from  $\langle n' \oplus (\# : c_1 + 1) = (- l -) \rangle$  obtain  $l'$  where  $n' = (- l' -)$  by(cases  $n'$ ) auto
from  $IH[OF \text{ this}]$  have  $l' < \# : c_2$  .
with  $\langle n' \oplus (\# : c_1 + 1) = (- l -) \rangle \langle n' = (- l' -) \rangle$  show ?case by simp
next
case (WCFG-WhileBody  $c'$   $n$  et  $n'$   $b$ )
note  $IH = \langle \bigwedge l. n' = (- l -) \implies l < \# : c' \rangle$ 
from  $\langle n' \oplus 2 = (- l -) \rangle$  obtain  $l'$  where  $n' = (- l' -)$  by(cases  $n'$ ) auto
from  $IH[OF \text{ this}]$  have  $l' < \# : c'$  .
with  $\langle n' \oplus 2 = (- l -) \rangle \langle n' = (- l' -) \rangle$  show ?case by simp

```

qed (*auto simp:num-inner-nodes-gr-0*)

lemma *WCFG-EntryD*:

$prog \vdash (-Entry-) -et \rightarrow n'$
 $\implies (n' = (-Exit-) \wedge et = (\lambda s. False)_{\checkmark}) \vee (n' = (-0-) \wedge et = (\lambda s. True)_{\checkmark})$
by(*induct prog n $\equiv$ (-Entry-) et n' rule:WCFG-induct,auto*)

lemma *WCFG-edge-det*:

$\llbracket prog \vdash n -et \rightarrow n'; prog \vdash n -et' \rightarrow n' \rrbracket \implies et = et'$
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-elim*s)
next
 case *WCFG-LAss* **thus** ?case **by**(*fastforce elim:WCFG-elim*s)
next
 case *WCFG-LAssSkip* **thus** ?case **by**(*fastforce elim:WCFG-elim*s)
next
 case (*WCFG-SeqFirst* c_1 n et n' c_2)
 note $IH = \langle c_1 \vdash n -et' \rightarrow n' \implies et = et' \rangle$
 from $\langle c_1 \vdash n -et \rightarrow n' \rangle \langle n' \neq (-Exit-) \rangle$ **obtain** l **where** $n' = (-l-)$
 by (*cases n'*) *auto*
 with $\langle c_1 \vdash n -et \rightarrow n' \rangle$ **have** $l < \# : c_1$
 by(*fastforce intro:WCFG-targetlabel-less-num-nodes*)
 with $\langle c_1;;c_2 \vdash n -et' \rightarrow n' \rangle \langle n' = (-l-) \rangle$ **have** $c_1 \vdash n -et' \rightarrow n'$
 by(*fastforce elim:WCFG-elim*s *intro:WCFG-intros dest:label-incr-ge*)
 from $IH[OF\ this]$ **show** ?case .
next
 case (*WCFG-SeqConnect* c_1 n et c_2)
 note $IH = \langle c_1 \vdash n -et' \rightarrow (-Exit-) \implies et = et' \rangle$
 from $\langle c_1 \vdash n -et \rightarrow (-Exit-) \rangle \langle n \neq (-Entry-) \rangle$ **obtain** l **where** $n = (-l-)$
 by (*cases n*) *auto*
 with $\langle c_1 \vdash n -et \rightarrow (-Exit-) \rangle$ **have** $l < \# : c_1$
 by(*fastforce intro:WCFG-sourcelabel-less-num-nodes*)
 with $\langle c_1;;c_2 \vdash n -et' \rightarrow (-0-) \oplus \# : c_1 \rangle \langle n = (-l-) \rangle$ **have** $c_1 \vdash n -et' \rightarrow (-Exit-)$
 by(*fastforce elim:WCFG-elim*s *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 $IH = \langle c_2 \vdash n -et' \rightarrow n' \implies et = et' \rangle$
 from $\langle c_2 \vdash n -et \rightarrow n' \rangle \langle n \neq (-Entry-) \rangle$ **obtain** l **where** $n = (-l-)$
 by (*cases n*) *auto*
 with $\langle c_2 \vdash n -et \rightarrow n' \rangle$ **have** $l < \# : c_2$
 by(*fastforce intro:WCFG-sourcelabel-less-num-nodes*)
 with $\langle c_1;;c_2 \vdash n \oplus \# : c_1 -et' \rightarrow n' \oplus \# : c_1 \rangle \langle n = (-l-) \rangle$ **have** $c_2 \vdash n -et' \rightarrow n'$


```

    by  $-(erule\ WCFG-elim, (fastforce\ dest: WCFG-sourcelabel-less-num-nodes\ label-incr-ge\ dest!: label-incr-inj) +)$ 
  from  $IH[OF\ this]$  show ?case .
next
  case  $WCFG-CondTrue$  thus ?case by  $(fastforce\ elim: WCFG-elim)$ 
next
  case  $WCFG-CondFalse$  thus ?case by  $(fastforce\ elim: WCFG-elim)$ 
next
  case  $(WCFG-CondThen\ c_1\ n\ et\ n'\ b\ c_2)$ 
  note  $IH = \langle c_1 \vdash n -et' \rightarrow n' \implies et = et' \rangle$ 
  from  $\langle c_1 \vdash n -et \rightarrow n' \rangle \langle n \neq (-Entry-) \rangle$  obtain  $l$  where  $n = (-\ l\ -)$ 
  by  $(cases\ n)\ auto$ 
  with  $\langle c_1 \vdash n -et \rightarrow n' \rangle$  have  $l < \#:c_1$ 
  by  $(fastforce\ intro: WCFG-sourcelabel-less-num-nodes)$ 
  with  $\langle if\ (b)\ c_1\ else\ c_2 \vdash n \oplus 1 -et' \rightarrow n' \oplus 1 \rangle \langle n = (-\ l\ -) \rangle$ 
  have  $c_1 \vdash n -et' \rightarrow n'$ 
  by  $-(erule\ WCFG-elim, (fastforce\ dest: label-incr-ge\ label-incr-inj) +)$ 
  from  $IH[OF\ this]$  show ?case .
next
  case  $(WCFG-CondElse\ c_2\ n\ et\ n'\ b\ c_1)$ 
  note  $IH = \langle c_2 \vdash n -et' \rightarrow n' \implies et = et' \rangle$ 
  from  $\langle c_2 \vdash n -et \rightarrow n' \rangle \langle n \neq (-Entry-) \rangle$  obtain  $l$  where  $n = (-\ l\ -)$ 
  by  $(cases\ n)\ auto$ 
  with  $\langle c_2 \vdash n -et \rightarrow n' \rangle$  have  $l < \#:c_2$ 
  by  $(fastforce\ intro: WCFG-sourcelabel-less-num-nodes)$ 
  with  $\langle if\ (b)\ c_1\ else\ c_2 \vdash n \oplus (\#:c_1 + 1) -et' \rightarrow n' \oplus (\#:c_1 + 1) \rangle \langle n = (-\ l\ -) \rangle$ 
  have  $c_2 \vdash n -et' \rightarrow n'$ 
  by  $-(erule\ WCFG-elim, (fastforce\ dest: WCFG-sourcelabel-less-num-nodes\ label-incr-inj\ label-incr-ge\ label-incr-simp-rev) +)$ 
  from  $IH[OF\ this]$  show ?case .
next
  case  $WCFG-WhileTrue$  thus ?case by  $(fastforce\ elim: WCFG-elim)$ 
next
  case  $WCFG-WhileFalse$  thus ?case by  $(fastforce\ elim: WCFG-elim)$ 
next
  case  $WCFG-WhileFalseSkip$  thus ?case by  $(fastforce\ elim: WCFG-elim)$ 
next
  case  $(WCFG-WhileBody\ c'\ n\ et\ n'\ b)$ 
  note  $IH = \langle c' \vdash n -et' \rightarrow n' \implies et = et' \rangle$ 
  from  $\langle c' \vdash n -et \rightarrow n' \rangle \langle n \neq (-Entry-) \rangle$  obtain  $l$  where  $n = (-\ l\ -)$ 
  by  $(cases\ n)\ auto$ 
  moreover
  with  $\langle c' \vdash n -et \rightarrow n' \rangle$  have  $l < \#:c'$ 
  by  $(fastforce\ intro: WCFG-sourcelabel-less-num-nodes)$ 
  moreover
  from  $\langle c' \vdash n -et \rightarrow n' \rangle \langle n' \neq (-Exit-) \rangle$  obtain  $l'$  where  $n' = (-\ l'\ -)$ 
  by  $(cases\ n')\ auto$ 
  moreover
  with  $\langle c' \vdash n -et \rightarrow n' \rangle$  have  $l' < \#:c'$ 

```

```

    by(fastforce intro:WCFG-targetlabel-less-num-nodes)
  ultimately have  $c' \vdash n - et' \rightarrow n'$  using  $\langle \text{while } (b) \ c' \vdash n \oplus 2 - et' \rightarrow n' \oplus 2 \rangle$ 
    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 =  $\langle c' \vdash n - et' \rightarrow (-Exit-) \implies et = et' \rangle$ 
  from  $\langle c' \vdash n - et \rightarrow (-Exit-) \rangle \langle n \neq (-Entry-) \rangle$  obtain  $l$  where  $n = (-l-)$ 
    by (cases  $n$ ) auto
  with  $\langle c' \vdash n - et \rightarrow (-Exit-) \rangle$  have  $l < \# : c'$ 
    by(fastforce intro:WCFG-sourcelabel-less-num-nodes)
  with  $\langle \text{while } (b) \ c' \vdash n \oplus 2 - et' \rightarrow (-0-) \rangle \langle n = (-l-) \rangle$ 
  have  $c' \vdash n - et' \rightarrow (-Exit-)$ 
    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  $prog \vdash (-l-) - et \rightarrow (-Exit-)$ 
proof -
  have  $\exists l \ et. \ l < \# : prog \wedge prog \vdash (-l-) - et \rightarrow (-Exit-)$ 
  proof(induct  $prog$ )
    case Skip
    have  $0 < \# : Skip$  by simp
    moreover have  $Skip \vdash (-0-) - \uparrow id \rightarrow (-Exit-)$  by(rule WCFG-Skip)
    ultimately show ?case by blast
  next
    case (LAss  $V \ e$ )
    have  $1 < \# : (V := e)$  by simp
    moreover have  $V := e \vdash (-1-) - \uparrow id \rightarrow (-Exit-)$  by(rule WCFG-LAssSkip)
    ultimately show ?case by blast
  next
    case (Seq  $prog1 \ prog2$ )
    from  $\langle \exists l \ et. \ l < \# : prog2 \wedge prog2 \vdash (-l-) - et \rightarrow (-Exit-) \rangle$ 
    obtain  $l \ et$  where  $l < \# : prog2$  and  $prog2 \vdash (-l-) - et \rightarrow (-Exit-)$ 
      by blast
    from  $\langle prog2 \vdash (-l-) - et \rightarrow (-Exit-) \rangle$ 
    have  $prog1 ;; prog2 \vdash (-l-) \oplus \# : prog1 - et \rightarrow (-Exit-) \oplus \# : prog1$ 
      by(fastforce intro:WCFG-SeqSecond)
    with  $\langle l < \# : prog2 \rangle$  show ?case by(rule-tac  $x=l + \# : prog1$  in  $exI, auto$ )
  next
    case (Cond  $b \ prog1 \ prog2$ )
    from  $\langle \exists l \ et. \ l < \# : prog1 \wedge prog1 \vdash (-l-) - et \rightarrow (-Exit-) \rangle$ 
    obtain  $l \ et$  where  $l < \# : prog1$  and  $prog1 \vdash (-l-) - et \rightarrow (-Exit-)$ 
      by blast
    from  $\langle prog1 \vdash (-l-) - et \rightarrow (-Exit-) \rangle$ 
    have  $\text{if } (b) \ prog1 \ \text{else } prog2 \vdash (-l-) \oplus 1 - et \rightarrow (-Exit-) \oplus 1$ 
      by(fastforce intro:WCFG-CondThen)
    with  $\langle l < \# : prog1 \rangle$  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' ⊢ (- l -) -↑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  $IH1 = \langle \bigwedge l. l < \# : prog1 \implies$ 
 $\exists n \text{ et. } prog1 \vdash n - et \rightarrow (-l-) \vee prog1 \vdash (-l-) - et \rightarrow n \rangle$ 
note  $IH2 = \langle \bigwedge l. l < \# : prog2 \implies$ 
 $\exists n \text{ et. } prog2 \vdash n - et \rightarrow (-l-) \vee prog2 \vdash (-l-) - et \rightarrow n \rangle$ 
show  $?case$ 
proof( $cases \ l = 0$ )
  case True
    have  $if \ (b) \ prog1 \ \text{else} \ prog2 \vdash (-Entry-) - (\lambda s. \ True)_{\sqrt{\rightarrow}} (-0-)$ 
    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  $n \text{ et}$ 
        where  $prog1 \vdash n - et \rightarrow (-l'-) \vee prog1 \vdash (-l'-) - et \rightarrow n$  by blast
        thus  $?thesis$ 
        proof
          assume  $edge : prog1 \vdash n - et \rightarrow (-l'-)$ 
          show  $?thesis$ 
          proof( $cases \ n = (-Entry-)$ )
            case True
              with  $edge$  have  $l' = 0$  by(auto dest: WCFG-EntryD)
              have  $if \ (b) \ prog1 \ \text{else} \ prog2 \vdash (-0-) - (\lambda s. \ interpret \ b \ s = \text{Some } true)_{\sqrt{\rightarrow}} (-0-) \oplus 1$ 
              by(rule WCFG-CondTrue)
              with  $\langle l' = 0 \rangle \langle l = l' + 1 \rangle$  show  $?thesis$  by simp blast
            case False
              with  $edge$  have  $if \ (b) \ prog1 \ \text{else} \ prog2 \vdash n \oplus 1 - et \rightarrow (-l'-) \oplus 1$ 
              by(fastforce intro: WCFG-CondThen)
              with  $\langle l = l' + 1 \rangle$  show  $?thesis$  by simp blast
          qed
        next
          assume  $prog1 \vdash (-l'-) - et \rightarrow n$ 
          hence  $if \ (b) \ prog1 \ \text{else} \ prog2 \vdash (-l'-) \oplus 1 - et \rightarrow n \oplus 1$ 
          by(fastforce intro: WCFG-CondThen)
          with  $\langle l = l' + 1 \rangle$  show  $?thesis$  by simp blast
        qed
      next
        case False
        hence  $\# : prog1 \leq l'$  by simp
        then obtain  $l''$  where  $l' = l'' + \# : prog1$  and  $l'' = l' - \# : prog1$ 
        by simp
        from  $\langle l' = l'' + \# : prog1 \rangle \langle l = l' + 1 \rangle \langle l < \# : (if \ (b) \ prog1 \ \text{else} \ prog2) \rangle$ 
        have  $l'' < \# : prog2$  by simp

```

```

from IH2[OF this] obtain  $n \text{ et}$ 
  where  $\text{prog2} \vdash n -\text{et} \rightarrow (-l'' -) \vee \text{prog2} \vdash (-l'' -) -\text{et} \rightarrow n$  by blast
thus ?thesis
proof
  assume  $\text{prog2} \vdash n -\text{et} \rightarrow (-l'' -)$ 
  show ?thesis
  proof(cases  $n = (-\text{Entry-})$ )
    case True
    with  $\langle \text{prog2} \vdash n -\text{et} \rightarrow (-l'' -) \rangle$  have  $l'' = 0$  by(auto dest:WCFG-EntryD)
    have if (b)  $\text{prog1}$  else  $\text{prog2} \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow$ 
       $(-0-) \oplus (\#:\text{prog1} + 1)$ 
      by(rule WCFG-CondFalse)
    with  $\langle l'' = 0 \rangle \langle l' = l'' + \#:\text{prog1} \rangle \langle l = l' + 1 \rangle$  show ?thesis by simp blast
  next
    case False
    with  $\langle \text{prog2} \vdash n -\text{et} \rightarrow (-l'' -) \rangle$ 
    have if (b)  $\text{prog1}$  else  $\text{prog2} \vdash n \oplus (\#:\text{prog1} + 1) -\text{et} \rightarrow$ 
       $(-l'' -) \oplus (\#:\text{prog1} + 1)$ 
      by(fastforce intro:WCFG-CondElse)
    with  $\langle l = l' + 1 \rangle \langle l' = l'' + \#:\text{prog1} \rangle$  show ?thesis by simp blast
  qed
next
  assume  $\text{prog2} \vdash (-l'' -) -\text{et} \rightarrow n$ 
  hence if (b)  $\text{prog1}$  else  $\text{prog2} \vdash (-l'' -) \oplus (\#:\text{prog1} + 1) -\text{et} \rightarrow$ 
     $n \oplus (\#:\text{prog1} + 1)$ 
    by(fastforce intro:WCFG-CondElse)
  with  $\langle l = l' + 1 \rangle \langle l' = l'' + \#:\text{prog1} \rangle$  show ?thesis by simp blast
qed
qed
qed
next
  case (While b  $\text{prog}'$ )
  note  $IH = \langle \bigwedge l. l < \#:\text{prog}' \rangle$ 
     $\implies \exists n \text{ et. } \text{prog}' \vdash n -\text{et} \rightarrow (-l -) \vee \text{prog}' \vdash (-l -) -\text{et} \rightarrow n$ 
  show ?case
  proof(cases  $l < 1$ )
    case True
    have while (b)  $\text{prog}' \vdash (-\text{Entry-}) -(\lambda s. \text{True})_{\checkmark} \rightarrow (-0-)$  by(rule WCFG-Entry)
    with True show ?thesis by simp blast
  next
    case False
    hence  $1 \leq l$  by simp
    thus ?thesis
    proof(cases  $l < 2$ )
      case True
      with  $\langle 1 \leq l \rangle$  have  $l = 1$  by simp
      have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
        by(rule WCFG-WhileFalse)
      with  $\langle l = 1 \rangle$  show ?thesis by simp blast
    next
      case False
      with  $\langle 1 \leq l \rangle$  have  $2 \leq l$  by simp
      thus ?thesis
      proof(cases  $l < 3$ )
        case True
        with  $\langle 2 \leq l \rangle$  have  $l = 2$  by simp
        have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
          by(rule WCFG-WhileFalse)
        with  $\langle l = 2 \rangle$  show ?thesis by simp blast
      next
        case False
        with  $\langle 2 \leq l \rangle$  have  $3 \leq l$  by simp
        thus ?thesis
        proof(cases  $l < 4$ )
          case True
          with  $\langle 3 \leq l \rangle$  have  $l = 3$  by simp
          have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
            by(rule WCFG-WhileFalse)
          with  $\langle l = 3 \rangle$  show ?thesis by simp blast
        next
          case False
          with  $\langle 3 \leq l \rangle$  have  $4 \leq l$  by simp
          thus ?thesis
          proof(cases  $l < 5$ )
            case True
            with  $\langle 4 \leq l \rangle$  have  $l = 4$  by simp
            have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
              by(rule WCFG-WhileFalse)
            with  $\langle l = 4 \rangle$  show ?thesis by simp blast
          next
            case False
            with  $\langle 4 \leq l \rangle$  have  $5 \leq l$  by simp
            thus ?thesis
            proof(cases  $l < 6$ )
              case True
              with  $\langle 5 \leq l \rangle$  have  $l = 5$  by simp
              have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                by(rule WCFG-WhileFalse)
              with  $\langle l = 5 \rangle$  show ?thesis by simp blast
            next
              case False
              with  $\langle 5 \leq l \rangle$  have  $6 \leq l$  by simp
              thus ?thesis
              proof(cases  $l < 7$ )
                case True
                with  $\langle 6 \leq l \rangle$  have  $l = 6$  by simp
                have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                  by(rule WCFG-WhileFalse)
                with  $\langle l = 6 \rangle$  show ?thesis by simp blast
              next
                case False
                with  $\langle 6 \leq l \rangle$  have  $7 \leq l$  by simp
                thus ?thesis
                proof(cases  $l < 8$ )
                  case True
                  with  $\langle 7 \leq l \rangle$  have  $l = 7$  by simp
                  have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                    by(rule WCFG-WhileFalse)
                  with  $\langle l = 7 \rangle$  show ?thesis by simp blast
                next
                  case False
                  with  $\langle 7 \leq l \rangle$  have  $8 \leq l$  by simp
                  thus ?thesis
                  proof(cases  $l < 9$ )
                    case True
                    with  $\langle 8 \leq l \rangle$  have  $l = 8$  by simp
                    have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                      by(rule WCFG-WhileFalse)
                    with  $\langle l = 8 \rangle$  show ?thesis by simp blast
                  next
                    case False
                    with  $\langle 8 \leq l \rangle$  have  $9 \leq l$  by simp
                    thus ?thesis
                    proof(cases  $l < 10$ )
                      case True
                      with  $\langle 9 \leq l \rangle$  have  $l = 9$  by simp
                      have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                        by(rule WCFG-WhileFalse)
                      with  $\langle l = 9 \rangle$  show ?thesis by simp blast
                    next
                      case False
                      with  $\langle 9 \leq l \rangle$  have  $10 \leq l$  by simp
                      thus ?thesis
                      proof(cases  $l < 11$ )
                        case True
                        with  $\langle 10 \leq l \rangle$  have  $l = 10$  by simp
                        have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                          by(rule WCFG-WhileFalse)
                        with  $\langle l = 10 \rangle$  show ?thesis by simp blast
                      next
                        case False
                        with  $\langle 10 \leq l \rangle$  have  $11 \leq l$  by simp
                        thus ?thesis
                        proof(cases  $l < 12$ )
                          case True
                          with  $\langle 11 \leq l \rangle$  have  $l = 11$  by simp
                          have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                            by(rule WCFG-WhileFalse)
                          with  $\langle l = 11 \rangle$  show ?thesis by simp blast
                        next
                          case False
                          with  $\langle 11 \leq l \rangle$  have  $12 \leq l$  by simp
                          thus ?thesis
                          proof(cases  $l < 13$ )
                            case True
                            with  $\langle 12 \leq l \rangle$  have  $l = 12$  by simp
                            have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                              by(rule WCFG-WhileFalse)
                            with  $\langle l = 12 \rangle$  show ?thesis by simp blast
                          next
                            case False
                            with  $\langle 12 \leq l \rangle$  have  $13 \leq l$  by simp
                            thus ?thesis
                            proof(cases  $l < 14$ )
                              case True
                              with  $\langle 13 \leq l \rangle$  have  $l = 13$  by simp
                              have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                                by(rule WCFG-WhileFalse)
                              with  $\langle l = 13 \rangle$  show ?thesis by simp blast
                            next
                              case False
                              with  $\langle 13 \leq l \rangle$  have  $14 \leq l$  by simp
                              thus ?thesis
                              proof(cases  $l < 15$ )
                                case True
                                with  $\langle 14 \leq l \rangle$  have  $l = 14$  by simp
                                have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                                  by(rule WCFG-WhileFalse)
                                with  $\langle l = 14 \rangle$  show ?thesis by simp blast
                              next
                                case False
                                with  $\langle 14 \leq l \rangle$  have  $15 \leq l$  by simp
                                thus ?thesis
                                proof(cases  $l < 16$ )
                                  case True
                                  with  $\langle 15 \leq l \rangle$  have  $l = 15$  by simp
                                  have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                                    by(rule WCFG-WhileFalse)
                                  with  $\langle l = 15 \rangle$  show ?thesis by simp blast
                                next
                                  case False
                                  with  $\langle 15 \leq l \rangle$  have  $16 \leq l$  by simp
                                  thus ?thesis
                                  proof(cases  $l < 17$ )
                                    case True
                                    with  $\langle 16 \leq l \rangle$  have  $l = 16$  by simp
                                    have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                                      by(rule WCFG-WhileFalse)
                                    with  $\langle l = 16 \rangle$  show ?thesis by simp blast
                                  next
                                    case False
                                    with  $\langle 16 \leq l \rangle$  have  $17 \leq l$  by simp
                                    thus ?thesis
                                    proof(cases  $l < 18$ )
                                      case True
                                      with  $\langle 17 \leq l \rangle$  have  $l = 17$  by simp
                                      have while (b)  $\text{prog}' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark} \rightarrow (-1-)$ 
                                        by(rule WCFG-WhileFalse)
                                      with  $\langle l = 17 \rangle$  show ?thesis by simp blast
                                    next
                                      case False
                                      with  $\langle 17 \leq l \rangle$  have  $18 \leq l$  by simp
                                      thus
```

```

next
  case False
  with  $\langle 1 \leq l \rangle$  have  $2 \leq l$  by simp
  then obtain  $l'$  where  $l = l' + 2$  and  $l' = l - 2$ 
    by(simp del:add-2-eq-Suc')
  from  $\langle l = l' + 2 \rangle \langle l < \# : \text{while } (b) \text{ prog}' \rangle$  have  $l' < \# : \text{prog}'$  by simp
  from IH[OF this] obtain  $n$  et
    where  $\text{prog}' \vdash n - \text{et} \rightarrow (-l' -) \vee \text{prog}' \vdash (-l' -) - \text{et} \rightarrow n$  by blast
  thus ?thesis
proof
  assume  $\text{prog}' \vdash n - \text{et} \rightarrow (-l' -)$ 
  show ?thesis
proof(cases  $n = (-\text{Entry}-)$ )
  case True
  with  $\langle \text{prog}' \vdash n - \text{et} \rightarrow (-l' -) \rangle$  have  $l' = 0$  by(auto dest:WCFG-EntryD)
  have  $\text{while } (b) \text{ prog}' \vdash (-0-) - (\lambda s. \text{interpret } b \text{ } s = \text{Some true})_{\checkmark} \rightarrow$ 
     $(-0-) \oplus 2$ 
    by(rule WCFG-WhileTrue)
  with  $\langle l' = 0 \rangle \langle l = l' + 2 \rangle$  show ?thesis by simp blast
next
  case False
  with  $\langle \text{prog}' \vdash n - \text{et} \rightarrow (-l' -) \rangle$ 
  have  $\text{while } (b) \text{ prog}' \vdash n \oplus 2 - \text{et} \rightarrow (-l' -) \oplus 2$ 
    by(fastforce intro:WCFG-WhileBody)
  with  $\langle l = l' + 2 \rangle$  show ?thesis by simp blast
qed
next
  assume  $\text{prog}' \vdash (-l' -) - \text{et} \rightarrow n$ 
  show ?thesis
proof(cases  $n = (-\text{Exit}-)$ )
  case True
  with  $\langle \text{prog}' \vdash (-l' -) - \text{et} \rightarrow n \rangle$ 
  have  $\text{while } (b) \text{ prog}' \vdash (-l' -) \oplus 2 - \text{et} \rightarrow (-0-)$ 
    by(fastforce intro:WCFG-WhileBodyExit)
  with  $\langle l = l' + 2 \rangle$  show ?thesis by simp blast
next
  case False
  with  $\langle \text{prog}' \vdash (-l' -) - \text{et} \rightarrow n \rangle$ 
  have  $\text{while } (b) \text{ prog}' \vdash (-l' -) \oplus 2 - \text{et} \rightarrow n \oplus 2$ 
    by(fastforce intro:WCFG-WhileBody)
  with  $\langle l = l' + 2 \rangle$  show ?thesis by simp blast
qed
qed
qed
qed
qed

```

lemma *WCFG-deterministic*:

$\llbracket \text{prog} \vdash n_1 - \text{et}_1 \rightarrow n_1'; \text{prog} \vdash n_2 - \text{et}_2 \rightarrow n_2'; n_1 = n_2; n_1' \neq n_2' \rrbracket$
 $\implies \exists Q Q'. \text{et}_1 = (Q)_{\checkmark} \wedge \text{et}_2 = (Q')_{\checkmark} \wedge (\forall s. (Q s \longrightarrow \neg Q' s) \wedge (Q' s \longrightarrow \neg Q s))$
proof(*induct arbitrary: $n_2 \ n_2'$ rule: WCFG-induct*)
case (*WCFG-Entry-Exit prog*)
from $\langle \text{prog} \vdash n_2 - \text{et}_2 \rightarrow n_2' \rangle \langle (-\text{Entry-}) = n_2 \rangle \langle (-\text{Exit-}) \neq n_2' \rangle$
have $\text{et}_2 = (\lambda s. \text{True})_{\checkmark}$ **by** (*fastforce dest: WCFG-EntryD*)
thus *?case by simp*
next
case (*WCFG-Entry prog*)
from $\langle \text{prog} \vdash n_2 - \text{et}_2 \rightarrow n_2' \rangle \langle (-\text{Entry-}) = n_2 \rangle \langle (-0-) \neq n_2' \rangle$
have $\text{et}_2 = (\lambda s. \text{False})_{\checkmark}$ **by** (*fastforce dest: WCFG-EntryD*)
thus *?case by simp*
next
case *WCFG-Skip*
from $\langle \text{Skip} \vdash n_2 - \text{et}_2 \rightarrow n_2' \rangle \langle (-0-) = n_2 \rangle \langle (-\text{Exit-}) \neq n_2' \rangle$
have *False* **by** (*fastforce elim: WCFG.While-CFG.cases*)
thus *?case by simp*
next
case (*WCFG-LAss V e*)
from $\langle V := e \vdash n_2 - \text{et}_2 \rightarrow n_2' \rangle \langle (-0-) = n_2 \rangle \langle (-1-) \neq n_2' \rangle$
have *False* **by** $\neg(\text{erule } \text{WCFG.While-CFG.cases, auto})$
thus *?case by simp*
next
case (*WCFG-LAssSkip V e*)
from $\langle V := e \vdash n_2 - \text{et}_2 \rightarrow n_2' \rangle \langle (-1-) = n_2 \rangle \langle (-\text{Exit-}) \neq n_2' \rangle$
have *False* **by** $\neg(\text{erule } \text{WCFG.While-CFG.cases, auto})$
thus *?case by simp*
next
case (*WCFG-SeqFirst $c_1 \ n \ \text{et} \ n' \ c_2$*)
note $\text{IH} = \langle \bigwedge n_2 \ n_2'. \llbracket c_1 \vdash n_2 - \text{et}_2 \rightarrow n_2'; n = n_2; n' \neq n_2' \rrbracket$
 $\implies \exists Q Q'. \text{et} = (Q)_{\checkmark} \wedge \text{et}_2 = (Q')_{\checkmark} \wedge (\forall s. (Q s \longrightarrow \neg Q' s) \wedge (Q' s \longrightarrow \neg Q s)) \rangle$
from $\langle c_1; c_2 \vdash n_2 - \text{et}_2 \rightarrow n_2' \rangle \langle c_1 \vdash n - \text{et} \rightarrow n' \rangle \langle n = n_2 \rangle \langle n' \neq n_2' \rangle$
have $c_1 \vdash n_2 - \text{et}_2 \rightarrow n_2' \vee (c_1 \vdash n_2 - \text{et}_2 \rightarrow (-\text{Exit-}) \wedge n_2' = (-0-) \oplus \# : c_1)$
apply $\neg$ **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 $c_1 \vdash n_2 - \text{et}_2 \rightarrow n_2'$
from $\text{IH}[\text{OF this } \langle n = n_2 \rangle \langle n' \neq n_2' \rangle]$ **show** *?case* .
next
assume $c_1 \vdash n_2 - \text{et}_2 \rightarrow (-\text{Exit-}) \wedge n_2' = (-0-) \oplus \# : c_1$
hence $\text{edge} : c_1 \vdash n_2 - \text{et}_2 \rightarrow (-\text{Exit-})$ **and** $n_2' : n_2' = (-0-) \oplus \# : c_1$ **by** *simp-all*
from $\text{IH}[\text{OF edge } \langle n = n_2 \rangle \langle n' \neq (-\text{Exit-}) \rangle]$ **show** *?case* .
qed
next
case (*WCFG-SeqConnect $c_1 \ n \ \text{et} \ c_2$*)


```

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

```

```

next
  case (WCFG-CondElse c2 n et n' b c1)
  note IH =  $\langle \bigwedge n_2 \ n_2'. \llbracket c_2 \vdash n_2 - et_2 \rightarrow n_2'; n = n_2; n' \neq n_2' \rrbracket$ 
     $\implies \exists Q \ Q'. et = (Q)_{\checkmark} \wedge et_2 = (Q')_{\checkmark} \wedge (\forall s. (Q \ s \longrightarrow \neg Q' \ s) \wedge (Q' \ s \longrightarrow \neg Q \ s)) \rangle$ 
  from  $\langle \text{if } (b) \ c_1 \text{ else } c_2 \vdash n_2 - et_2 \rightarrow n_2' \rangle \langle c_2 \vdash n - et \rightarrow n' \rangle \langle n \neq (-\text{Entry-}) \rangle$ 
     $\langle n \oplus \# : c_1 + 1 = n_2 \rangle \langle n' \oplus \# : c_1 + 1 \neq n_2' \rangle$ 
  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  $\langle \text{while } (b) \ c' \vdash n_2 - et_2 \rightarrow n_2' \rangle \langle (-0-) = n_2 \rangle \langle (-0-) \oplus 2 \neq n_2' \rangle$ 
  show ?case by - (erule WCFG.While-CFG.cases, auto)
next
  case (WCFG-WhileFalse b c')
  from  $\langle \text{while } (b) \ c' \vdash n_2 - et_2 \rightarrow n_2' \rangle \langle (-0-) = n_2 \rangle \langle (-1-) \neq n_2' \rangle$ 
  show ?case by - (erule WCFG.While-CFG.cases, auto)
next
  case (WCFG-WhileFalseSkip b c')
  from  $\langle \text{while } (b) \ c' \vdash n_2 - et_2 \rightarrow n_2' \rangle \langle (-1-) = n_2 \rangle \langle (-\text{Exit-}) \neq n_2' \rangle$ 
  show ?case by - (erule WCFG.While-CFG.cases, auto dest: label-incr-ge)
next
  case (WCFG-WhileBody c' n et n' b)
  note IH =  $\langle \bigwedge n_2 \ n_2'. \llbracket c' \vdash n_2 - et_2 \rightarrow n_2'; n = n_2; n' \neq n_2' \rrbracket$ 
     $\implies \exists Q \ Q'. et = (Q)_{\checkmark} \wedge et_2 = (Q')_{\checkmark} \wedge (\forall s. (Q \ s \longrightarrow \neg Q' \ s) \wedge (Q' \ s \longrightarrow \neg Q \ s)) \rangle$ 
  from  $\langle \text{while } (b) \ c' \vdash n_2 - et_2 \rightarrow n_2' \rangle \langle c' \vdash n - et \rightarrow n' \rangle \langle n \neq (-\text{Entry-}) \rangle$ 
     $\langle n' \neq (-\text{Exit-}) \rangle \langle n \oplus 2 = n_2 \rangle \langle n' \oplus 2 \neq n_2' \rangle$ 
  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 =  $\langle \bigwedge n_2 \ n_2'. \llbracket c' \vdash n_2 - et_2 \rightarrow n_2'; n = n_2; (-\text{Exit-}) \neq n_2' \rrbracket$ 
     $\implies \exists Q \ Q'. et = (Q)_{\checkmark} \wedge et_2 = (Q')_{\checkmark} \wedge (\forall s. (Q \ s \longrightarrow \neg Q' \ s) \wedge (Q' \ s \longrightarrow \neg Q \ s)) \rangle$ 
  from  $\langle \text{while } (b) \ c' \vdash n_2 - et_2 \rightarrow n_2' \rangle \langle c' \vdash n - et \rightarrow (-\text{Exit-}) \rangle \langle n \neq (-\text{Entry-}) \rangle$ 
     $\langle n \oplus 2 = n_2 \rangle \langle (-0-) \neq n_2' \rangle$ 

```

```

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  $sourcenode :: w-edge \Rightarrow w-node$ 
where  $sourcenode\ e \equiv fst\ e$ 

```

```

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

```

```

abbreviation  $kind :: w-edge \Rightarrow state\ edge-kind$ 
where  $kind\ e \equiv fst(snd\ e)$ 

```

```

definition  $valid-edge :: cmd \Rightarrow w-edge \Rightarrow bool$ 
where  $valid-edge\ prog\ a \equiv prog \vdash sourcenode\ a - kind\ a \rightarrow targetnode\ a$ 

```

```

definition  $valid-node :: cmd \Rightarrow w-node \Rightarrow bool$ 
where  $valid-node\ prog\ n \equiv$ 
   $(\exists a. valid-edge\ prog\ a \wedge (n = sourcenode\ a \vee n = targetnode\ a))$ 

```

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  $\langle valid-edge\ prog\ a \rangle \langle targetnode\ a = (-Entry-) \rangle$ 
  have  $prog \vdash nx - et \rightarrow (-Entry-)$  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  $\langle \text{sourcenode } a = \text{sourcenode } a' \rangle \langle \text{targetnode } a = \text{targetnode } a' \rangle$ 
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  $\langle \text{valid-edge prog } a \rangle \langle \text{sourcenode } a = (-Exit-) \rangle$ 
  have prog  $\vdash (-Exit-) -et\rightarrow nx'$  by(simp add:valid-edge-def)
  thus False by fastforce
next
  have prog  $\vdash (-Entry-) -(\lambda s. \text{False})_{\checkmark}\rightarrow (-Exit-)$  by(rule WCFG-Entry-Exit)
  thus  $\exists a. \text{valid-edge prog } a \wedge \text{sourcenode } a = (-Entry-) \wedge$ 
     $\text{targetnode } a = (-Exit-) \wedge \text{kind } a = (\lambda s. \text{False})_{\checkmark}$ 
    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  $\Rightarrow$  nat  $\Rightarrow$  cmd  $\Rightarrow$  bool
where

```

Labels-Base:

$labels\ c\ 0\ c$

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

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

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

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

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

| *Labels-WhileBody:*
 $labels\ c'\ l\ c \implies labels\ (while(b)\ c')\ (l + 2)\ (c;;while(b)\ c')$

| *Labels-WhileExit:*
 $labels\ (while(b)\ c')\ 1\ Skip$

lemma *label-less-num-inner-nodes:*
 $labels\ c\ l\ c' \implies l < \# : c$

proof(*induct c arbitrary:l c'*)
 case *Skip*
 from $\langle labels\ Skip\ l\ c' \rangle$ **show** ?case **by**(*fastforce elim:labels.cases*)
next
 case (*LAss V e*)
 from $\langle labels\ (V := e)\ l\ c' \rangle$ **show** ?case **by**(*fastforce elim:labels.cases*)
next
 case (*Seq c₁ c₂*)
 note $IH1 = \langle \bigwedge l\ c'.\ labels\ c_1\ l\ c' \implies l < \# : c_1 \rangle$
 note $IH2 = \langle \bigwedge l\ c'.\ labels\ c_2\ l\ c' \implies l < \# : c_2 \rangle$
 from $\langle labels\ (c_1;;c_2)\ l\ c' \rangle$ $IH1\ IH2$ **show** ?case
 by *simp(erule labels.cases,auto,force)*
next
 case (*Cond b c₁ c₂*)
 note $IH1 = \langle \bigwedge l\ c'.\ labels\ c_1\ l\ c' \implies l < \# : c_1 \rangle$
 note $IH2 = \langle \bigwedge l\ c'.\ labels\ c_2\ l\ c' \implies l < \# : c_2 \rangle$
 from $\langle labels\ (if\ (b)\ c_1\ else\ c_2)\ l\ c' \rangle$ $IH1\ IH2$ **show** ?case
 by *simp(erule labels.cases,auto,force)*
next
 case (*While b c*)
 note $IH = \langle \bigwedge l\ c'.\ labels\ c\ l\ c' \implies l < \# : c \rangle$
 from $\langle labels\ (while\ (b)\ c)\ l\ c' \rangle$ IH **show** ?case
 by *simp(erule labels.cases,fastforce+)*
qed

```

declare One-nat-def [simp del]

lemma less-num-inner-nodes-label:
   $l < \# : c \implies \exists c'. \text{labels } c \ l \ c'$ 
proof(induct c arbitrary:l)
  case Skip
  from  $\langle l < \# : \text{Skip} \rangle$  have  $l = 0$  by simp
  thus ?case by(fastforce intro:Labels-Base)
next
  case (LAss V e)
  from  $\langle l < \# : (V := e) \rangle$  have  $l = 0 \vee l = 1$  by auto
  thus ?case by(auto intro:Labels-Base Labels-LAss)
next
  case (Seq c1 c2)
  note IH1 =  $\langle \bigwedge l. l < \# : c_1 \implies \exists c'. \text{labels } c_1 \ l \ c' \rangle$ 
  note IH2 =  $\langle \bigwedge l. l < \# : c_2 \implies \exists c'. \text{labels } c_2 \ l \ c' \rangle$ 
  show ?case
  proof(cases  $l < \# : c_1$ )
    case True
    from IH1[OF this] obtain c' where labels c1 l c' by auto
    hence labels (c1;;c2) l (c';;c2) 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  $\langle l = l' + \# : c_1 \rangle \langle l < \# : c_1;;c_2 \rangle$  have  $l' < \# : c_2$  by simp
    from IH2[OF this] obtain c' where labels c2 l' c' by auto
    with  $\langle l = l' + \# : c_1 \rangle$  have labels (c1;;c2) l c' by(fastforce intro:Labels-Seq2)
    thus ?thesis by auto
  qed
next
  case (Cond b c1 c2)
  note IH1 =  $\langle \bigwedge l. l < \# : c_1 \implies \exists c'. \text{labels } c_1 \ l \ c' \rangle$ 
  note IH2 =  $\langle \bigwedge l. l < \# : c_2 \implies \exists c'. \text{labels } c_2 \ l \ c' \rangle$ 
  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 c1 l' c' by auto

```

```

    with  $\langle l = l' + 1 \rangle$  have labels (if (b)  $c_1$  else  $c_2$ )  $l$   $c'$ 
      by(fastforce dest:Labels-CondTrue)
    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  $\langle l' = l'' + \# : c_1 \rangle \langle l = l' + 1 \rangle \langle l < \# : \text{if (b) } c_1 \text{ else } c_2 \rangle$ 
    have  $l'' < \# : c_2$  by simp
    from IH2[OF this] obtain  $c'$  where labels  $c_2$   $l''$   $c'$  by auto
    with  $\langle l' = l'' + \# : c_1 \rangle \langle l = l' + 1 \rangle$  have labels (if (b)  $c_1$  else  $c_2$ )  $l$   $c'$ 
      by(fastforce dest:Labels-CondFalse)
    thus ?thesis by auto
  qed
qed
next
  case (While b  $c'$ )
  note IH =  $\langle \bigwedge l. l < \# : c' \implies \exists c''. \text{labels } c' l c'' \rangle$ 
  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  $\langle \neg l < 1 \rangle$  have  $l = 1$  by simp
      thus ?thesis by(fastforce intro:Labels-WhileExit)
    next
      case False
      with  $\langle \neg l < 1 \rangle$  have  $2 \leq l$  by simp
      then obtain  $l'$  where  $l = l' + 2$  and  $l' = l - 2$ 
        by(simp del:add-2-eq-Suc')
      from  $\langle l = l' + 2 \rangle \langle l < \# : \text{while (b) } c' \rangle$  have  $l' < \# : c'$  by simp
      from IH[OF this] obtain  $c''$  where labels  $c' l' c''$  by auto
      with  $\langle l = l' + 2 \rangle$  have labels (while (b)  $c'$ )  $l$  ( $c''$ ;;while (b)  $c'$ )
        by(fastforce dest:Labels-WhileBody)
      thus ?thesis by auto
    qed
  qed
qed
qed

```

lemma labels-det:
 labels $c l c' \implies (\bigwedge c''. \text{labels } c l c'' \implies c' = c'')$
 proof(induct rule: labels.induct)

```

    case (Labels-Base c c'')
    from ⟨labels c 0 c'⟩ obtain l where labels c l c'' and l = 0 by auto
    thus ?case by(induct rule: labels.induct,auto)
next
    case (Labels-Seq1 c1 l c c2)
    note IH = ⟨∧c''. labels c1 l c'' ⇒ c = c''⟩
    from ⟨labels c1 l c⟩ have l < #:c1 by(fastforce intro:label-less-num-inner-nodes)
    with ⟨labels (c1;;c2) l c'⟩ obtain cx where c'' = cx;;c2 ∧ labels c1 l cx
    by(fastforce elim:labels.cases intro:Labels-Base)
    hence [simp]:c'' = cx;;c2 and labels c1 l cx by simp-all
    from IH[OF ⟨labels c1 l cx⟩] show ?case by simp
next
    case (Labels-Seq2 c2 l c c1)
    note IH = ⟨∧c''. labels c2 l c'' ⇒ c = c''⟩
    from ⟨labels (c1;;c2) (l + #:c1) c'⟩ ⟨labels c2 l c⟩ have labels c2 l c''
    by(auto elim:labels.cases dest:label-less-num-inner-nodes)
    from IH[OF this] show ?case .
next
    case (Labels-CondTrue c1 l c b c2)
    note IH = ⟨∧c''. labels c1 l c'' ⇒ c = c''⟩
    from ⟨labels (if (b) c1 else c2) (l + 1) c'⟩ ⟨labels c1 l c⟩ have labels c1 l c''
    by(fastforce elim:labels.cases dest:label-less-num-inner-nodes)
    from IH[OF this] show ?case .
next
    case (Labels-CondFalse c2 l c b c1)
    note IH = ⟨∧c''. labels c2 l c'' ⇒ c = c''⟩
    from ⟨labels (if (b) c1 else c2) (l + #:c1 + 1) c'⟩ ⟨labels c2 l c⟩
    have labels c2 l c''
    by(fastforce elim:labels.cases dest:label-less-num-inner-nodes)
    from IH[OF this] show ?case .
next
    case (Labels-WhileBody c' l c b)
    note IH = ⟨∧c''. labels c' l c'' ⇒ c = c''⟩
    from ⟨labels (while (b) c') (l + 2) c'⟩ ⟨labels c' l c⟩
    obtain cx where c'' = cx;;while (b) c' ∧ labels c' l cx
    by -(erule labels.cases,auto)
    hence [simp]:c'' = cx;;while (b) c' and labels c' l cx by simp-all
    from IH[OF ⟨labels c' l cx⟩] show ?case by simp
qed (fastforce elim:labels.cases)+

```

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  $\Rightarrow$  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  $\Rightarrow$  vname set
where

```

```

  rhs-aux (Val v)        = {}
| rhs-aux (Var V)        = {V}
| rhs-aux (e1 <<bop>> e2) = (rhs-aux e1  $\cup$  rhs-aux e2)

```

```

fun rhs :: cmd  $\Rightarrow$  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';  $\forall V \in$  rhs-aux b. s V = s' V]]
 $\impl$  interpret b s' = Some v'

```

proof(induct b arbitrary:v')

case (Val v)

from $\langle$ interpret (Val v) s = Some v' $\rangle$ **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 = $\langle \bigwedge v'. \llbracket \text{interpret } b1 \text{ s} = \text{Some } v'; \forall V \in \text{rhs-aux } b1. \text{s } V = \text{s}' V \rrbracket$
 $\impl \text{interpret } b1 \text{ s}' = \text{Some } v' \rangle$

note IH2 = $\langle \bigwedge v'. \llbracket \text{interpret } b2 \text{ s} = \text{Some } v'; \forall V \in \text{rhs-aux } b2. \text{s } V = \text{s}' V \rrbracket$
 $\impl \text{interpret } b2 \text{ s}' = \text{Some } v' \rangle$

from $\langle$ interpret (b1 <<bop>> b2) s = Some v' $\rangle$

have $\exists v_1 v_2. \text{interpret } b1 \text{ s} = \text{Some } v_1 \wedge \text{interpret } b2 \text{ s} = \text{Some } v_2 \wedge$
 $\text{binop } bop \text{ } v_1 \text{ } v_2 = \text{Some } v'$

```

    apply(cases interpret b1 s, simp)
    apply(cases interpret b2 s, simp)
    by(case-tac binop bop a aa, simp+)
  then obtain v1 v2 where interpret b1 s = Some v1
    and interpret b2 s = Some v2 and binop bop v1 v2 = 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 v1⟩ this] have interpret b1 s' = Some v1 .
  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 v2⟩ this] have interpret b2 s' = Some v2 .
  with ⟨interpret b1 s' = Some v1⟩ ⟨binop bop v1 v2 = 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 $\text{prog} \vdash n \text{ --et--> } n'$ 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 c1 n et n' c2)
  note IH = ⟨V ∉ Defs c1 n ⟹ transfer et s V = s V⟩
  have V ∉ Defs c1 n
  proof
    assume V ∈ Defs c1 n
    then obtain c l where [simp]: n = (- l -) and labels c1 l c
      and V ∈ lhs c by fastforce
    from ⟨labels c1 l c⟩ have labels (c1;;c2) l (c;;c2)
      by(fastforce intro: Labels-Seq1)
    from ⟨V ∈ lhs c⟩ have V ∈ lhs (c;;c2) by simp
    with ⟨labels (c1;;c2) l (c;;c2)⟩ have V ∈ Defs (c1;;c2) n by fastforce
    with ⟨V ∉ Defs (c1;;c2) n⟩ show False by fastforce
  qed
qed

```

```

    from IH[OF this] show ?case .
next
  case (WCFG-SeqConnect c1 n et c2)
  note IH = ⟨V ∉ Defs c1 n ⟹ transfer et s V = s V⟩
  have V ∉ Defs c1 n
  proof
    assume V ∈ Defs c1 n
    then obtain c l where [simp]: n = (- l -) and labels c1 l c
      and V ∈ lhs c by fastforce
    from ⟨labels c1 l c⟩ have labels (c1;;c2) l (c;;c2)
      by (fastforce intro:Labels-Seq1)
    from ⟨V ∈ lhs c⟩ have V ∈ lhs (c;;c2) by simp
    with ⟨labels (c1;;c2) l (c;;c2)⟩ have V ∈ Defs (c1;;c2) n by fastforce
    with ⟨V ∉ Defs (c1;;c2) n⟩ show False by fastforce
  qed
  from IH[OF this] show ?case .
next
  case (WCFG-SeqSecond c2 n et n' c1)
  note IH = ⟨V ∉ Defs c2 n ⟹ transfer et s V = s V⟩
  have V ∉ Defs c2 n
  proof
    assume V ∈ Defs c2 n
    then obtain c l where [simp]: n = (- l -) and labels c2 l c
      and V ∈ lhs c by fastforce
    from ⟨labels c2 l c⟩ have labels (c1;;c2) (l + #:c1) c
      by (fastforce intro:Labels-Seq2)
    with ⟨V ∈ lhs c⟩ have V ∈ Defs (c1;;c2) (n ⊕ #:c1) by fastforce
    with ⟨V ∉ Defs (c1;;c2) (n ⊕ #:c1)⟩ show False by fastforce
  qed
  from IH[OF this] show ?case .
next
  case (WCFG-CondThen c1 n et n' b c2)
  note IH = ⟨V ∉ Defs c1 n ⟹ transfer et s V = s V⟩
  have V ∉ Defs c1 n
  proof
    assume V ∈ Defs c1 n
    then obtain c l where [simp]: n = (- l -) and labels c1 l c
      and V ∈ lhs c by fastforce
    from ⟨labels c1 l c⟩ have labels (if (b) c1 else c2) (l + 1) c
      by (fastforce intro:Labels-CondTrue)
    with ⟨V ∈ lhs c⟩ have V ∈ Defs (if (b) c1 else c2) (n ⊕ 1) by fastforce
    with ⟨V ∉ Defs (if (b) c1 else c2) (n ⊕ 1)⟩ show False by fastforce
  qed
  from IH[OF this] show ?case .
next
  case (WCFG-CondElse c2 n et n' b c1)
  note IH = ⟨V ∉ Defs c2 n ⟹ transfer et s V = s V⟩
  have V ∉ Defs c2 n
  proof

```

```

assume  $V \in \text{Defs } c_2 \ n$ 
then obtain  $c \ l$  where  $[\text{simp}]:n = (- \ l \ -)$  and  $\text{labels } c_2 \ l \ c$ 
and  $V \in \text{lhs } c$  by fastforce
from  $\langle \text{labels } c_2 \ l \ c \rangle$  have  $\text{labels } (\text{if } (b) \ c_1 \ \text{else } c_2) \ (l + \# : c_1 + 1) \ c$ 
by  $(\text{fastforce } \text{intro} : \text{Labels-CondFalse})$ 
with  $\langle V \in \text{lhs } c \rangle$  have  $V \in \text{Defs } (\text{if } (b) \ c_1 \ \text{else } c_2) \ (n \oplus \# : c_1 + 1)$ 
by  $(\text{fastforce } \text{simp} : \text{nat-add-commute } \text{nat-add-left-commute})$ 
with  $\langle V \notin \text{Defs } (\text{if } (b) \ c_1 \ \text{else } c_2) \ (n \oplus \# : c_1 + 1) \rangle$  show False by fastforce
qed
from  $\text{IH}[\text{OF } \text{this}]$  show ?case .
next
case  $(\text{WCFG-WhileBody } c' \ n \ \text{et } n' \ b)$ 
note  $\text{IH} = \langle V \notin \text{Defs } c' \ n \implies \text{transfer et } s \ V = s \ V \rangle$ 
have  $V \notin \text{Defs } c' \ n$ 
proof
assume  $V \in \text{Defs } c' \ n$ 
then obtain  $c \ l$  where  $[\text{simp}]:n = (- \ l \ -)$  and  $\text{labels } c' \ l \ c$ 
and  $V \in \text{lhs } c$  by fastforce
from  $\langle \text{labels } c' \ l \ c \rangle$  have  $\text{labels } (\text{while } (b) \ c') \ (l + 2) \ (c;;\text{while } (b) \ c')$ 
by  $(\text{fastforce } \text{intro} : \text{Labels-WhileBody})$ 
from  $\langle V \in \text{lhs } c \rangle$  have  $V \in \text{lhs } (c;;\text{while } (b) \ c')$  by fastforce
with  $\langle \text{labels } (\text{while } (b) \ c') \ (l + 2) \ (c;;\text{while } (b) \ c') \rangle$ 
have  $V \in \text{Defs } (\text{while } (b) \ c') \ (n \oplus 2)$  by fastforce
with  $\langle V \notin \text{Defs } (\text{while } (b) \ c') \ (n \oplus 2) \rangle$  show False by fastforce
qed
from  $\text{IH}[\text{OF } \text{this}]$  show ?case .
next
case  $(\text{WCFG-WhileBodyExit } c' \ n \ \text{et } b)$ 
note  $\text{IH} = \langle V \notin \text{Defs } c' \ n \implies \text{transfer et } s \ V = s \ V \rangle$ 
have  $V \notin \text{Defs } c' \ n$ 
proof
assume  $V \in \text{Defs } c' \ n$ 
then obtain  $c \ l$  where  $[\text{simp}]:n = (- \ l \ -)$  and  $\text{labels } c' \ l \ c$ 
and  $V \in \text{lhs } c$  by fastforce
from  $\langle \text{labels } c' \ l \ c \rangle$  have  $\text{labels } (\text{while } (b) \ c') \ (l + 2) \ (c;;\text{while } (b) \ c')$ 
by  $(\text{fastforce } \text{intro} : \text{Labels-WhileBody})$ 
from  $\langle V \in \text{lhs } c \rangle$  have  $V \in \text{lhs } (c;;\text{while } (b) \ c')$  by fastforce
with  $\langle \text{labels } (\text{while } (b) \ c') \ (l + 2) \ (c;;\text{while } (b) \ c') \rangle$ 
have  $V \in \text{Defs } (\text{while } (b) \ c') \ (n \oplus 2)$  by fastforce
with  $\langle V \notin \text{Defs } (\text{while } (b) \ c') \ (n \oplus 2) \rangle$  show False by fastforce
qed
from  $\text{IH}[\text{OF } \text{this}]$  show ?case .
qed auto

```

lemma *WCFG-edge-transfer-uses-only-Uses*:

$\llbracket \text{prog} \vdash n \text{ --et--} n'; \forall V \in \text{Uses } \text{prog } n. \ s \ V = s' \ V \rrbracket$

$\implies \forall V \in \text{Defs } \text{prog } n. \ (\text{transfer et } s) \ V = (\text{transfer et } s') \ V$

proof $(\text{induct rule} : \text{WCFG-induct})$

```

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 ↑λs. s(V := interpret e s) s V =
  transfer ↑λs. 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:split-if-asm) }
  with None show ?thesis by fastforce
next
  case (Some v)
  hence interpret e s = Some v by(fastforce split:split-if-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 c1 n et n' c2)
  note IH = ⟨∀ V∈Uses c1 n. s V = s' V⟩
    ⇒ ∀ V∈Defs c1 n. transfer et s V = transfer et s' V
  from ⟨∀ V∈Uses (c1;;c2) n. s V = s' V⟩ have ∀ V∈Uses c1 n. s V = s' V
    by auto(drule Labels-Seq1[of - - - c2],erule-tac x=V in allE,auto)
  from IH[OF this] have ∀ V∈Defs c1 n. transfer et s V = transfer et s' V .
  with ⟨c1 ⊢ 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 c1 n et c2)
  note IH = ⟨∀ V∈Uses c1 n. s V = s' V⟩
    ⇒ ∀ V∈Defs c1 n. transfer et s V = transfer et s' V
  from ⟨∀ V∈Uses (c1;;c2) n. s V = s' V⟩ have ∀ V∈Uses c1 n. s V = s' V
    by auto(drule Labels-Seq1[of - - - c2],erule-tac x=V in allE,auto)
  from IH[OF this] have ∀ V∈Defs c1 n. transfer et s V = transfer et s' V .
  with ⟨c1 ⊢ 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 c2 n et n' c1)
  note IH = ⟨∀ V∈Uses c2 n. s V = s' V⟩

```

```

     $\Rightarrow \forall V \in \text{Defs } c_2 \ n. \text{ transfer et } s \ V = \text{ transfer et } s' \ V$ 
from  $\langle \forall V \in \text{Uses } (c_1;;c_2) \ (n \oplus \#;c_1). \ s \ V = s' \ V \rangle$  have  $\forall V \in \text{Uses } c_2 \ n. \ s \ V =$ 
 $s' \ V$ 
    by(auto,blast dest:Labels-Seq2)
from IH[OF this] have  $\forall V \in \text{Defs } c_2 \ n. \text{ transfer et } s \ V = \text{ transfer et } s' \ V$  .
with num-inner-nodes-gr-0[of c1] show ?case
    apply clarsimp
    apply(erule labels.cases,auto)
    by(cases n,auto dest:label-less-num-inner-nodes)+
next
case (WCFG-CondThen c1 n et n' b c2)
note IH =  $\langle \forall V \in \text{Uses } c_1 \ n. \ s \ V = s' \ V \rangle$ 
     $\Rightarrow \forall V \in \text{Defs } c_1 \ n. \text{ transfer et } s \ V = \text{ transfer et } s' \ V$ 
from  $\langle \forall V \in \text{Uses } (\text{if } (b) \ c_1 \text{ else } c_2) \ (n \oplus 1). \ s \ V = s' \ V \rangle$ 
have  $\forall V \in \text{Uses } c_1 \ n. \ s \ V = s' \ V$  by(auto,blast dest:Labels-CondTrue)
from IH[OF this] have  $\forall V \in \text{Defs } c_1 \ n. \text{ transfer et } s \ V = \text{ transfer et } s' \ V$  .
with  $\langle c_1 \vdash n - \text{et} \rightarrow n' \rangle$  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 c2 n et n' b c1)
note IH =  $\langle \forall V \in \text{Uses } c_2 \ n. \ s \ V = s' \ V \rangle$ 
     $\Rightarrow \forall V \in \text{Defs } c_2 \ n. \text{ transfer et } s \ V = \text{ transfer et } s' \ V$ 
from  $\langle \forall V \in \text{Uses } (\text{if } (b) \ c_1 \text{ else } c_2) \ (n \oplus \#;c_1 + 1). \ s \ V = s' \ V \rangle$ 
have  $\forall V \in \text{Uses } c_2 \ n. \ s \ V = s' \ V$ 
    by auto(drule Labels-CondFalse[of - - - b c1],erule-tac x=V in allE,
    auto simp:nat-add-assoc)
from IH[OF this] have  $\forall V \in \text{Defs } c_2 \ n. \text{ transfer et } s \ V = \text{ transfer et } s' \ V$  .
with  $\langle c_2 \vdash n - \text{et} \rightarrow n' \rangle$  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 =  $\langle \forall V \in \text{Uses } c' \ n. \ s \ V = s' \ V \rangle$ 
     $\Rightarrow \forall V \in \text{Defs } c' \ n. \text{ transfer et } s \ V = \text{ transfer et } s' \ V$ 
from  $\langle \forall V \in \text{Uses } (\text{while } (b) \ c') \ (n \oplus 2). \ s \ V = s' \ V \rangle$  have  $\forall V \in \text{Uses } c' \ n. \ s \ V$ 
 $= s' \ V$ 
    by auto(drule Labels-WhileBody[of - - - b],erule-tac x=V in allE,auto)
from IH[OF this] have  $\forall V \in \text{Defs } c' \ n. \text{ transfer et } s \ V = \text{ transfer et } s' \ V$  .
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  $IH = \langle \forall V \in \text{Uses } c' n. s V = s' V \rangle$ 
 $\implies \forall V \in \text{Defs } c' n. \text{transfer et } s V = \text{transfer et } s' V \rangle$ 
from  $\langle \forall V \in \text{Uses } (\text{while } (b) c') (n \oplus 2). s V = s' V \rangle$  have  $\forall V \in \text{Uses } c' n. s V = s' V$ 
by auto(drule Labels-WhileBody[of - - b],erule-tac  $x = V$  in allE,auto)
from  $IH[OF \text{ this}]$  have  $\forall V \in \text{Defs } c' n. \text{transfer et } s V = \text{transfer et } s' V$  .
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*:

```

 $\llbracket \text{prog} \vdash n - \text{et} \rightarrow n'; \forall V \in \text{Uses prog } n. s V = s' V; \text{pred et } s \rrbracket$ 
 $\implies \text{pred et } s'$ 
proof(induct rule:WCFG-induct)
case (WCFG-SeqFirst  $c_1 n \text{ et } n' c_2$ )
note  $IH = \langle \llbracket \forall V \in \text{Uses } c_1 n. s V = s' V; \text{pred et } s \rrbracket \implies \text{pred et } s' \rangle$ 
from  $\langle \forall V \in \text{Uses } (c_1;; c_2) n. s V = s' V \rangle$  have  $\forall V \in \text{Uses } c_1 n. s V = s' V$ 
by auto(drule Labels-Seq1[of - -  $c_2$ ],erule-tac  $x = V$  in allE,auto)
from  $IH[OF \text{ this } \langle \text{pred et } s \rangle]$  show ?case .
next
case (WCFG-SeqConnect  $c_1 n \text{ et } c_2$ )
note  $IH = \langle \llbracket \forall V \in \text{Uses } c_1 n. s V = s' V; \text{pred et } s \rrbracket \implies \text{pred et } s' \rangle$ 
from  $\langle \forall V \in \text{Uses } (c_1;; c_2) n. s V = s' V \rangle$  have  $\forall V \in \text{Uses } c_1 n. s V = s' V$ 
by auto(drule Labels-Seq1[of - -  $c_2$ ],erule-tac  $x = V$  in allE,auto)
from  $IH[OF \text{ this } \langle \text{pred et } s \rangle]$  show ?case .
next
case (WCFG-SeqSecond  $c_2 n \text{ et } n' c_1$ )
note  $IH = \langle \llbracket \forall V \in \text{Uses } c_2 n. s V = s' V; \text{pred et } s \rrbracket \implies \text{pred et } s' \rangle$ 
from  $\langle \forall V \in \text{Uses } (c_1;; c_2) (n \oplus \# : c_1). s V = s' V \rangle$ 
have  $\forall V \in \text{Uses } c_2 n. s V = s' V$  by(auto,blast dest:Labels-Seq2)
from  $IH[OF \text{ this } \langle \text{pred et } s \rangle]$  show ?case .
next
case (WCFG-CondTrue  $b c_1 c_2$ )
from  $\langle \forall V \in \text{Uses } (\text{if } (b) c_1 \text{ else } c_2) (-0-). s V = s' V \rangle$ 
have  $\text{all:} \forall V. \text{labels } (\text{if } (b) c_1 \text{ else } c_2) 0 (\text{if } (b) c_1 \text{ else } c_2) \wedge$ 
 $V \in \text{rhs } (\text{if } (b) c_1 \text{ else } c_2) \longrightarrow (s V = s' V)$ 
by fastforce
obtain  $v'$  where [simp]: $v' = \text{true}$  by simp
with  $\langle \text{pred } (\lambda s. \text{interpret } b s = \text{Some true}) \checkmark s \rangle$ 
have  $\text{interpret } b s = \text{Some } v'$  by simp
have  $\text{labels } (\text{if } (b) c_1 \text{ else } c_2) 0 (\text{if } (b) c_1 \text{ else } c_2)$  by(rule Labels-Base)
with all have  $\forall V \in \text{rhs-aux } b. s V = s' V$  by simp
with  $\langle \text{interpret } b s = \text{Some } v' \rangle$  have  $\text{interpret } b s' = \text{Some } v'$ 
by(rule rhs-interpret-eq)
thus ?case by simp
next

```

```

case (WCFG-CondFalse  $b \ c_1 \ c_2$ )
from  $\langle \forall V \in \text{Uses} \ (if \ (b) \ c_1 \ else \ c_2) \ (-0-). \ s \ V = s' \ V \rangle$ 
have  $all: \forall V. \text{labels} \ (if \ (b) \ c_1 \ else \ c_2) \ 0 \ (if \ (b) \ c_1 \ else \ c_2) \wedge$ 
 $V \in \text{rhs} \ (if \ (b) \ c_1 \ else \ c_2) \longrightarrow (s \ V = s' \ V)$ 
by fastforce
obtain  $v'$  where  $[simp]: v' = false$  by simp
with  $\langle pred \ (\lambda s. \text{interpret } b \ s = \text{Some } false) \rangle_{\surd} \ s \rangle$ 
have  $\text{interpret } b \ s = \text{Some } v'$  by simp
have  $\text{labels} \ (if \ (b) \ c_1 \ else \ c_2) \ 0 \ (if \ (b) \ c_1 \ else \ c_2)$  by (rule Labels-Base)
with  $all \ \text{have} \ \forall V \in \text{rhs-aux } b. \ s \ V = s' \ V$  by simp
with  $\langle \text{interpret } b \ s = \text{Some } v' \rangle$  have  $\text{interpret } b \ s' = \text{Some } v'$ 
by (rule rhs-interpret-eq)
thus  $?case$  by simp
next
case (WCFG-CondThen  $c_1 \ n \ et \ n' \ b \ c_2$ )
note  $IH = \llbracket \forall V \in \text{Uses} \ c_1 \ n. \ s \ V = s' \ V; \ pred \ et \ s \rrbracket \Longrightarrow pred \ et \ s'$ 
from  $\langle \forall V \in \text{Uses} \ (if \ (b) \ c_1 \ else \ c_2) \ (n \oplus 1). \ s \ V = s' \ V \rangle$ 
have  $\forall V \in \text{Uses} \ c_1 \ n. \ s \ V = s' \ V$  by (auto,blast dest:Labels-CondTrue)
from  $IH[OF \ this \ \langle pred \ et \ s \rangle]$  show  $?case$  .
next
case (WCFG-CondElse  $c_2 \ n \ et \ n' \ b \ c_1$ )
note  $IH = \llbracket \forall V \in \text{Uses} \ c_2 \ n. \ s \ V = s' \ V; \ pred \ et \ s \rrbracket \Longrightarrow pred \ et \ s'$ 
from  $\langle \forall V \in \text{Uses} \ (if \ (b) \ c_1 \ else \ c_2) \ (n \oplus \# : c_1 + 1). \ s \ V = s' \ V \rangle$ 
have  $\forall V \in \text{Uses} \ c_2 \ n. \ s \ V = s' \ V$ 
by (auto(drule Labels-CondFalse[of - - - b c1],erule-tac x=V in allE,
auto simp:nat-add-assoc))
from  $IH[OF \ this \ \langle pred \ et \ s \rangle]$  show  $?case$  .
next
case (WCFG-WhileTrue  $b \ c'$ )
from  $\langle \forall V \in \text{Uses} \ (while \ (b) \ c') \ (-0-). \ s \ V = s' \ V \rangle$ 
have  $all: \forall V. \text{labels} \ (while \ (b) \ c') \ 0 \ (while \ (b) \ c') \wedge$ 
 $V \in \text{rhs} \ (while \ (b) \ c') \longrightarrow (s \ V = s' \ V)$ 
by fastforce
obtain  $v'$  where  $[simp]: v' = true$  by simp
with  $\langle pred \ (\lambda s. \text{interpret } b \ s = \text{Some } true) \rangle_{\surd} \ s \rangle$ 
have  $\text{interpret } b \ s = \text{Some } v'$  by simp
have  $\text{labels} \ (while \ (b) \ c') \ 0 \ (while \ (b) \ c')$  by (rule Labels-Base)
with  $all \ \text{have} \ \forall V \in \text{rhs-aux } b. \ s \ V = s' \ V$  by simp
with  $\langle \text{interpret } b \ s = \text{Some } v' \rangle$  have  $\text{interpret } b \ s' = \text{Some } v'$ 
by (rule rhs-interpret-eq)
thus  $?case$  by simp
next
case (WCFG-WhileFalse  $b \ c'$ )
from  $\langle \forall V \in \text{Uses} \ (while \ (b) \ c') \ (-0-). \ s \ V = s' \ V \rangle$ 
have  $all: \forall V. \text{labels} \ (while \ (b) \ c') \ 0 \ (while \ (b) \ c') \wedge$ 
 $V \in \text{rhs} \ (while \ (b) \ c') \longrightarrow (s \ V = s' \ V)$ 
by fastforce
obtain  $v'$  where  $[simp]: v' = false$  by simp
with  $\langle pred \ (\lambda s. \text{interpret } b \ s = \text{Some } false) \rangle_{\surd} \ s \rangle$ 

```



```

have interpret  $b\ s = \text{Some } v'$  by simp
have labels  $(\text{while } (b)\ c')\ 0\ (\text{while } (b)\ c')$  by (rule Labels-Base)
with all have  $\forall V \in \text{rhs-aux } b. s\ V = s'\ V$  by simp
with  $\langle \text{interpret } b\ s = \text{Some } v' \rangle$  have interpret  $b\ s' = \text{Some } v'$ 
  by (rule rhs-interpret-eq)
thus ?case by simp
next
  case (WCFG-WhileBody  $c'\ n\ \text{et } n'\ b$ )
  note  $IH = \langle \llbracket \forall V \in \text{Uses } c'\ n. s\ V = s'\ V; \text{pred } \text{et } s \rrbracket \implies \text{pred } \text{et } s' \rangle$ 
  from  $\langle \forall V \in \text{Uses } (\text{while } (b)\ c')\ (n \oplus 2). s\ V = s'\ V \rangle$  have  $\forall V \in \text{Uses } c'\ n. s\ V$ 
   $= s'\ V$ 
  by auto (drule Labels-WhileBody [of - - - b], erule-tac  $x = V$  in allE, auto)
  from  $IH$  [OF this  $\langle \text{pred } \text{et } s \rangle$ ] show ?case .
next
  case (WCFG-WhileBodyExit  $c'\ n\ \text{et } b$ )
  note  $IH = \langle \llbracket \forall V \in \text{Uses } c'\ n. s\ V = s'\ V; \text{pred } \text{et } s \rrbracket \implies \text{pred } \text{et } s' \rangle$ 
  from  $\langle \forall V \in \text{Uses } (\text{while } (b)\ c')\ (n \oplus 2). s\ V = s'\ V \rangle$  have  $\forall V \in \text{Uses } c'\ n. s\ V$ 
   $= s'\ V$ 
  by auto (drule Labels-WhileBody [of - - - b], erule-tac  $x = V$  in allE, auto)
  from  $IH$  [OF this  $\langle \text{pred } \text{et } s \rangle$ ] 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  $\text{Defs } \text{prog } (-\text{Entry}-) = \{\} \wedge \text{Uses } \text{prog } (-\text{Entry}-) = \{\}$ 
  by (simp add: Defs.simps Uses.simps)
next
  fix  $a\ V\ s$ 
  assume valid-edge prog a and  $V \notin \text{Defs } \text{prog } (\text{sourcenode } a)$ 
  obtain  $nx\ \text{et } nx'$  where  $[simp]: a = (nx, \text{et}, nx')$  by (cases a) auto
  with  $\langle \text{valid-edge prog } a \rangle$  have  $\text{prog} \vdash nx \text{ --et--> } nx'$  by (simp add: valid-edge-def)
  with  $\langle V \notin \text{Defs } \text{prog } (\text{sourcenode } a) \rangle$  show  $\text{id } (\text{transfer } (\text{kind } a)\ s)\ V = \text{id } s\ V$ 
  by (fastforce intro: WCFG-edge-no-Defs-equal)
next
  fix  $a\ \text{fix } s\ s'::\text{state}$ 
  assume valid-edge prog a
  and  $\forall V \in \text{Uses } \text{prog } (\text{sourcenode } a). \text{id } s\ V = \text{id } s'\ V$ 
  obtain  $nx\ \text{et } nx'$  where  $[simp]: a = (nx, \text{et}, nx')$  by (cases a) auto
  with  $\langle \text{valid-edge prog } a \rangle$  have  $\text{prog} \vdash nx \text{ --et--> } nx'$  by (simp add: valid-edge-def)
  with  $\langle \forall V \in \text{Uses } \text{prog } (\text{sourcenode } a). \text{id } s\ V = \text{id } s'\ V \rangle$ 
  show  $\forall V \in \text{Defs } \text{prog } (\text{sourcenode } a).$ 
     $\text{id } (\text{transfer } (\text{kind } a)\ s)\ V = \text{id } (\text{transfer } (\text{kind } a)\ s')\ V$ 
  by  $-(\text{drule } \text{WCFG-edge-transfer-uses-only-Uses, simp+})$ 
next
  fix  $a\ s\ s'$ 
  assume  $\text{pred}:\text{pred } (\text{kind } a)\ s$  and valid:valid-edge prog a

```

and $\text{all}:\forall V \in \text{Uses prog (sourcenode a)}. \text{id } s \ V = \text{id } s' \ V$
obtain $nx \text{ et } nx'$ **where** $[simp]:a = (nx, et, nx')$ **by** $(\text{cases } a) \text{ auto}$
with $\langle \text{valid-edge prog a} \rangle$ **have** $\text{prog} \vdash nx - et \rightarrow nx'$ **by** $(\text{simp add:valid-edge-def})$
with $\langle \text{pred (kind a) s} \rangle \langle \forall V \in \text{Uses prog (sourcenode a)}. \text{id } s \ V = \text{id } s' \ V \rangle$
show pred (kind a) s' **by** $-(\text{drule WCFG-edge-Uses-pred-eq, simp+})$
next
fix $a \ a'$
assume valid-edge prog a **and** $\text{valid-edge prog a'}$
and $\text{sourcenode a} = \text{sourcenode a'}$ **and** $\text{targetnode a} \neq \text{targetnode a'}$
thus $\exists Q \ Q'. \text{kind a} = (Q)_{\checkmark} \wedge \text{kind a'} = (Q')_{\checkmark} \wedge$
 $(\forall s. (Q \ s \longrightarrow \neg Q' \ s) \wedge (Q' \ s \longrightarrow \neg Q \ s))$
by $(\text{fastforce intro!: WCFG-deterministic simp:valid-edge-def})$
qed

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

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

end

4.6 Lemmas for the control dependences

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

4.6.1 Paths to $(-\text{Exit-})$ and from $(-\text{Entry-})$ exist

abbreviation $\text{path} :: \text{cmd} \Rightarrow \text{w-node} \Rightarrow \text{w-edge list} \Rightarrow \text{w-node} \Rightarrow \text{bool}$
 $(- \vdash - \dashrightarrow* -)$
where $\text{prog} \vdash n - as \rightarrow* n' \equiv \text{CFG.path sourcenode targetnode (valid-edge prog)}$

$n \text{ as } n'$

definition $\text{label-incrs} :: \text{w-edge list} \Rightarrow \text{nat} \Rightarrow \text{w-edge list}$ $(- \oplus s - 60)$
where $as \oplus s i \equiv \text{map } (\lambda(n, et, n'). (n \oplus i, et, n' \oplus i)) \ as$

lemma *path-SeqFirst:*

```

  prog ⊢ n -as→* (- l -) ⇒ prog;;c₂ ⊢ 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 $c_1;;prog \vdash n'' \oplus \# : c_1 - as \oplus s \# : c_1 \rightarrow * n' \oplus \# : c_1$.
with $\langle c_1;;prog \vdash n \oplus \# : c_1 - kind \ a \rightarrow n'' \oplus \# : c_1 \rangle \langle sourcenode \ a = n \rangle$
 $\langle targetnode \ a = n'' \rangle \langle [(n, kind \ a, n'')] \oplus s \# : c_1 = [a] \oplus s \# : c_1 \rangle$ **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*:

$prog \vdash (- \ l \ -) - as \rightarrow * n'$
 $\implies$ *if* (*b*) *prog* *else* $c_2 \vdash (- \ l \ -) \oplus 1 - as \oplus s 1 \rightarrow * n' \oplus 1$
proof(*induct* $(- \ l \ -)$ *as* n' *arbitrary:l rule:While-CFG.path.induct*)
case *empty-path*
from $\langle CFG.valid-node \ sourcenode \ targetnode \ (valid-edge \ prog) \ (- \ l \ -) \rangle$
 $WCFG-CondTrue[of \ b \ prog \ c_2]$
have $CFG.valid-node \ sourcenode \ targetnode \ (valid-edge \ (if \ (b) \ prog \ else \ c_2))$
 $((- \ l \ -) \oplus 1)$
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 $IH = \langle \bigwedge l. n'' = (- \ l \ -) \rangle$
 $\implies$ *if* (*b*) *prog* *else* $c_2 \vdash (- \ l \ -) \oplus 1 - as \oplus s 1 \rightarrow * n' \oplus 1$
from $\langle valid-edge \ prog \ a \rangle \langle sourcenode \ a = (- \ l \ -) \rangle \langle targetnode \ a = n'' \rangle$
have *if* (*b*) *prog* *else* $c_2 \vdash (- \ l \ -) \oplus 1 - kind \ a \rightarrow n'' \oplus 1$
by $-(rule \ WCFG-CondThen,simp-all \ add:valid-edge-def)$
from $\langle sourcenode \ a = (- \ l \ -) \rangle \langle targetnode \ a = n'' \rangle \langle valid-edge \ prog \ a \rangle$
have $[((- \ l \ -), kind \ a, n'')] \oplus s 1 = [a] \oplus s 1$
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* $c_2 \vdash (- \ l' \ -) \oplus 1 - as \oplus s 1 \rightarrow * n' \oplus 1$.
with $\langle if \ (b) \ prog \ else \ c_2 \vdash (- \ l \ -) \oplus 1 - kind \ a \rightarrow n'' \oplus 1 \rangle$ *Node*
have *if* (*b*) *prog* *else* $c_2 \vdash (- \ l \ -) \oplus 1 - ((- \ l \ -) \oplus 1, kind \ a, n'' \oplus 1) \# (as \oplus s 1) \rightarrow * n' \oplus 1$
by(*fastforce intro: While-CFG.Cons-path simp:valid-edge-def valid-node-def*)
with $\langle [((- \ l \ -), kind \ a, n'')] \oplus s 1 = [a] \oplus s 1 \rangle$
have *if* (*b*) *prog* *else* $c_2 \vdash (- \ l \ -) \oplus 1 - a \# as \oplus s 1 \rightarrow * n' \oplus 1$
by(*simp add:label-incrs-def*)
thus *?thesis* **by** *simp*
next

```

case Entry
with  $\langle \text{valid-edge } \text{prog } a \rangle \langle \text{targetnode } a = n'' \rangle$  have False by fastforce
thus ?thesis by simp
next
case Exit
with  $\langle \text{prog} \vdash n'' - \text{as} \rightarrow^* n' \rangle$  have  $n' = (-\text{Exit-})$  and  $\text{as} = []$ 
by (auto dest: While-CFGExit.path-Exit-source)
from  $\langle \text{if } (b) \text{ prog else } c_2 \vdash (-l-) \oplus 1 - \text{kind } a \rightarrow n'' \oplus 1 \rangle$ 
have  $\langle \text{if } (b) \text{ prog else } c_2 \vdash (-l-) \oplus 1 - [((-l-) \oplus 1, \text{kind } a, n'' \oplus 1)] \rightarrow^* n'' \oplus 1 \rangle$ 
by (fastforce intro: While-CFG.Cons-path While-CFG.empty-path
simp: While-CFG.valid-node-def valid-edge-def)
with Exit  $\langle [((-l-), \text{kind } a, n'')] \oplus s \ 1 = [a] \oplus s \ 1 \rangle$   $\langle n' = (-\text{Exit-}) \rangle$   $\langle \text{as} = [] \rangle$ 
show ?thesis by (fastforce simp: label-incrs-def)
qed
qed

```

lemma *path-CondFalse*:

```

 $\text{prog} \vdash (-l-) - \text{as} \rightarrow^* n'$ 
 $\implies \text{if } (b) \ c_1 \text{ else } \text{prog} \vdash (-l-) \oplus (\# : c_1 + 1) - \text{as} \oplus s \ (\# : c_1 + 1) \rightarrow^* n' \oplus (\# : c_1 + 1)$ 

```

proof (*induct* $(-l-)$ *as* n' *arbitrary:l rule: While-CFG.path.induct*)

```

case empty-path
from  $\langle \text{CFG.valid-node sourcenode targetnode } (\text{valid-edge prog}) (-l-) \rangle$ 
WCFG-CondFalse [of b c1 prog]
have  $\langle \text{CFG.valid-node sourcenode targetnode } (\text{valid-edge } (\text{if } (b) \ c_1 \text{ else } \text{prog}))$ 
 $((-l-) \oplus \# : c_1 + 1)$ 
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  $IH = \langle \bigwedge l. n'' = (-l-) \implies \text{if } (b) \ c_1 \text{ else } \text{prog} \vdash (-l-) \oplus (\# : c_1 + 1)$ 
 $- \text{as} \oplus s \ (\# : c_1 + 1) \rightarrow^* n' \oplus (\# : c_1 + 1) \rangle$ 

```

from $\langle \text{valid-edge prog } a \rangle \langle \text{sourcenode } a = (-l-) \rangle \langle \text{targetnode } a = n'' \rangle$

have $\langle \text{if } (b) \ c_1 \text{ else } \text{prog} \vdash (-l-) \oplus (\# : c_1 + 1) - \text{kind } a \rightarrow n'' \oplus (\# : c_1 + 1) \rangle$

by $-(\text{rule } \text{WCFG-CondElse, simp-all add: valid-edge-def})$

from $\langle \text{sourcenode } a = (-l-) \rangle \langle \text{targetnode } a = n'' \rangle \langle \text{valid-edge prog } a \rangle$

have $\langle [((-l-), \text{kind } a, n'')] \oplus s \ (\# : c_1 + 1) = [a] \oplus s \ (\# : c_1 + 1) \rangle$

by (*cases a, simp add: label-incrs-def valid-edge-def*)

show *?case*

proof (*cases* n'')

case (*Node* l')

from $IH[OF \text{ this}]$ **have** $\langle \text{if } (b) \ c_1 \text{ else } \text{prog} \vdash (-l'-) \oplus (\# : c_1 + 1)$

$- \text{as} \oplus s \ (\# : c_1 + 1) \rightarrow^* n' \oplus (\# : c_1 + 1) \rangle$

with $\langle \text{if } (b) \ c_1 \text{ else } \text{prog} \vdash (-l-) \oplus (\# : c_1 + 1) - \text{kind } a \rightarrow n'' \oplus (\# : c_1 + 1) \rangle$

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 (#:c1 + 1) = [a] ⊕ s (#:c1 + 1)⟩ Node
have if (b) c1 else prog ⊢ (- l -) ⊕ (#:c1 + 1) - a # as ⊕ s (#:c1 + 1) →*
  n' ⊕ (#:c1 + 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-CFG.Exit.path-Exit-source)
from ⟨if (b) c1 else prog ⊢ (- l -) ⊕ (#:c1 + 1) - kind a → n'' ⊕ (#:c1 + 1)⟩
have if (b) c1 else prog ⊢ (- l -) ⊕ (#:c1 + 1)
  -[((- l -) ⊕ (#:c1 + 1), kind a, n'' ⊕ (#:c1 + 1))] →* n'' ⊕ (#:c1 + 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 (#:c1 + 1) = [a] ⊕ s (#:c1 + 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  $IH = \langle \bigwedge l. n'' = (- l -) \Rightarrow \text{while } (b) \text{ prog} \vdash (- l -) \oplus 2 -as \oplus s 2 \rightarrow^* (- l' -) \oplus 2 \rangle$ 
from  $\langle \text{sourcenode } a = (- l -) \rangle \langle \text{targetnode } a = n'' \rangle \langle \text{valid-edge prog } a \rangle$ 
have  $[((- l -), \text{kind } a, n'')] \oplus s 2 = [a] \oplus s 2$ 
by  $(\text{cases } a, \text{simp add: label-incrs-def valid-edge-def})$ 
show  $?case$ 
proof  $(\text{cases } n'')$ 
  case  $(\text{Node } l'')$ 
    with  $\langle \text{valid-edge prog } a \rangle \langle \text{sourcenode } a = (- l -) \rangle \langle \text{targetnode } a = n'' \rangle$ 
    have  $\text{while } (b) \text{ prog} \vdash (- l -) \oplus 2 -\text{kind } a \rightarrow n'' \oplus 2$ 
    by  $-(\text{rule } WCFG\text{-WhileBody}, \text{simp-all add: valid-edge-def})$ 
    from  $IH[OF \text{ Node}]$ 
    have  $\text{while } (b) \text{ prog} \vdash (- l'' -) \oplus 2 -as \oplus s 2 \rightarrow^* (- l' -) \oplus 2 .$ 
    with  $\langle \text{while } (b) \text{ prog} \vdash (- l -) \oplus 2 -\text{kind } a \rightarrow n'' \oplus 2 \rangle \text{ Node}$ 
    have  $\text{while } (b) \text{ prog} \vdash (- l -) \oplus 2 -((- l -) \oplus 2, \text{kind } a, n'' \oplus 2) \# (as \oplus s 2) \rightarrow^*$ 
     $(- l' -) \oplus 2$ 
    by  $(\text{fastforce intro: While-CFG.Cons-path simp: valid-edge-def})$ 
    with  $\langle [((- l -), \text{kind } a, n'')] \oplus s 2 = [a] \oplus s 2 \rangle$  show  $?thesis$  by  $(\text{simp add: label-incrs-def})$ 
  next
    case  $\text{Entry}$ 
    with  $\langle \text{valid-edge prog } a \rangle \langle \text{targetnode } a = n'' \rangle$  have  $\text{False}$  by  $\text{fastforce}$ 
    thus  $?thesis$  by  $\text{simp}$ 
  next
    case  $\text{Exit}$ 
    with  $\langle \text{prog} \vdash n'' -as \rightarrow^* (- l' -) \rangle$  have  $(- l' -) = (- \text{Exit} -)$  and  $as = []$ 
    by  $(\text{auto dest: While-CFG.Exit.path-Exit-source})$ 
    then have  $\text{False}$  by  $\text{simp}$ 
    thus  $?thesis$  by  $\text{simp}$ 
qed
qed

```

lemma *inner-node-Entry-Exit-path*:

$l < \# : \text{prog} \Rightarrow (\exists as. \text{prog} \vdash (- l -) -as \rightarrow^* (- \text{Exit} -)) \wedge$
 $(\exists as. \text{prog} \vdash (- \text{Entry} -) -as \rightarrow^* (- l -))$

proof $(\text{induct prog arbitrary: } l)$

```

case  $\text{Skip}$ 
from  $\langle l < \# : \text{Skip} \rangle$  have  $[\text{simp}]: l = 0$  by  $\text{simp}$ 
hence  $\text{Skip} \vdash (- l -) -\uparrow id \rightarrow (- \text{Exit} -)$  by  $(\text{simp add: WCFG-Skip})$ 
hence  $\text{Skip} \vdash (- l -) -[((- l -), \uparrow id, (- \text{Exit} -))] \rightarrow^* (- \text{Exit} -)$ 
by  $(\text{fastforce intro: While-CFG.path.intros simp: valid-edge-def})$ 
have  $\text{Skip} \vdash (- \text{Entry} -) -(\lambda s. \text{True})_{\sqrt{\phantom{x}}} \rightarrow (- l -)$  by  $(\text{simp add: WCFG-Entry})$ 
hence  $\text{Skip} \vdash (- \text{Entry} -) -[((- \text{Entry} -), (\lambda s. \text{True})_{\sqrt{\phantom{x}}}, (- l -))] \rightarrow^* (- l -)$ 
by  $(\text{fastforce intro: While-CFG.path.intros simp: valid-edge-def While-CFG.valid-node-def})$ 
with  $\langle \text{Skip} \vdash (- l -) -[((- l -), \uparrow id, (- \text{Exit} -))] \rightarrow^* (- \text{Exit} -) \rangle$  show  $?case$  by  $\text{fastforce}$ 
next
case  $(\text{LAss } V e)$ 
from  $\langle l < \# : V := e \rangle$  have  $l = 0 \vee l = 1$  by  $\text{auto}$ 

```

```

thus ?case
proof
  assume [simp]:l = 0
  hence V:=e ⊢ (-Entry-) - (λs. True)✓→ (- l -) by (simp add: WCFG-Entry)
  hence V:=e ⊢ (-Entry-) - [((-Entry-), (λs. True)✓, (- 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 -), ↑(λs. 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-), (λs. True)✓, (- 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-) - ↑(λs. s(V:= (interpret e s)))→ (- l -)
  by (simp add: WCFG-LAss)
  hence V:=e ⊢ (-0-) - [((-0-), ↑(λs. 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-), (λs. True)✓, (-0-)]
    , [((-0-), ↑(λs. 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 $\langle \text{prog1} \vdash (-l-) -asx \rightarrow^* \text{sourcenode } ax \rangle \langle \text{valid-edge prog1 } ax \rangle$
obtain lx **where** $[simp]: \text{sourcenode } ax = (-lx-)$
by $(\text{cases sourcenode } ax) \text{ auto}$
with $\langle \text{prog1} \vdash (-l-) -asx \rightarrow^* \text{sourcenode } ax \rangle$
have $\text{prog1};;\text{prog2} \vdash (-l-) -asx \rightarrow^* \text{sourcenode } ax$
by $(\text{fastforce intro: path-SeqFirst})$
from $\langle \text{valid-edge prog1 } ax \rangle \langle (-Exit-) = \text{targetnode } ax \rangle$
have $\text{prog1};;\text{prog2} \vdash \text{sourcenode } ax -\text{kind } ax \rightarrow (-0-) \oplus \#:\text{prog1}$
by $(\text{fastforce intro: WCFG-SeqConnect simp: valid-edge-def})$
hence $\text{prog1};;\text{prog2} \vdash \text{sourcenode } ax -[(\text{sourcenode } ax, \text{kind } ax, (-0-) \oplus \#:\text{prog1})] \rightarrow^*$
 $(-0-) \oplus \#:\text{prog1}$
by $(\text{fastforce intro: While-CFG.Cons-path While-CFG.empty-path}$
 $\text{simp: While-CFG.valid-node-def valid-edge-def})$
with $\langle \text{prog1};;\text{prog2} \vdash (-l-) -asx \rightarrow^* \text{sourcenode } ax \rangle$
have $\text{prog1};;\text{prog2} \vdash (-l-) -asx @[(\text{sourcenode } ax, \text{kind } ax, (-0-) \oplus \#:\text{prog1})] \rightarrow^*$
 $(-0-) \oplus \#:\text{prog1}$
by $(\text{fastforce intro: While-CFG.path-Append})$
from $IH2[\text{of } 0]$ **obtain** as'' **where** $\text{prog2} \vdash (-0-) -as'' \rightarrow^* (-Exit-)$ **by** *blast*
hence $\text{prog1};;\text{prog2} \vdash (-0-) \oplus \#:\text{prog1} -as'' \oplus s \#:\text{prog1} \rightarrow^* (-Exit-) \oplus \#:\text{prog1}$
by $(\text{fastforce intro!: path-SeqSecond elim: While-CFG.path.cases})$
hence $\text{prog1};;\text{prog2} \vdash (-0-) \oplus \#:\text{prog1} -as'' \oplus s \#:\text{prog1} \rightarrow^* (-Exit-)$
by *simp*
with $\langle \text{prog1};;\text{prog2} \vdash (-l-) -asx @[(\text{sourcenode } ax, \text{kind } ax, (-0-) \oplus \#:\text{prog1})] \rightarrow^*$
 $(-0-) \oplus \#:\text{prog1})$
have $\text{prog1};;\text{prog2} \vdash (-l-) - (asx @[(\text{sourcenode } ax, \text{kind } ax, (-0-) \oplus \#:\text{prog1})]) @$
 $(as'' \oplus s \#:\text{prog1}) \rightarrow^* (-Exit-)$
by $(\text{fastforce intro: While-CFG.path-Append})$
with $\langle \text{prog1};;\text{prog2} \vdash (-Entry-) -as' \rightarrow^* (-l-) \rangle$ **show** *?thesis* **by** *blast*
next
case *False*
hence $\#:\text{prog1} \leq l$ **by** *simp*
then obtain l' **where** $[simp]: l = l' + \#:\text{prog1}$ **and** $l' = l - \#:\text{prog1}$ **by** *simp*
from $\langle l < \#:\text{prog1};;\text{prog2} \rangle$ **have** $l' < \#:\text{prog2}$ **by** *simp*
from $IH2[\text{OF this}]$ **obtain** as' **where** $\text{prog2} \vdash (-l'-) -as' \rightarrow^* (-Exit-)$
and $\text{prog2} \vdash (-Entry-) -as' \rightarrow^* (-l'-)$ **by** *blast*
from $\langle \text{prog2} \vdash (-l'-) -as' \rightarrow^* (-Exit-) \rangle$
have $\text{prog1};;\text{prog2} \vdash (-l'-) \oplus \#:\text{prog1} -as \oplus s \#:\text{prog1} \rightarrow^* (-Exit-) \oplus \#:\text{prog1}$
by $(\text{fastforce intro!: path-SeqSecond elim: While-CFG.path.cases})$
hence $\text{prog1};;\text{prog2} \vdash (-l-) -as \oplus s \#:\text{prog1} \rightarrow^* (-Exit-)$
by *simp*
from $IH1[\text{of } 0]$ **obtain** as'' **where** $\text{prog1} \vdash (-0-) -as'' \rightarrow^* (-Exit-)$ **by** *blast*
then obtain ax asx **where** $\text{prog1} \vdash (-0-) -asx @[ax] \rightarrow^* (-Exit-)$
by $(\text{induct rule: rev-induct, auto elim: While-CFG.path.cases})$
hence $\text{prog1} \vdash (-0-) -asx \rightarrow^* \text{sourcenode } ax$ **and** $\text{valid-edge prog1 } ax$
and $(-Exit-) = \text{targetnode } ax$ **by** $(\text{auto intro: While-CFG.path-split-snoc})$
from $WCFG\text{-Entry } \langle \text{prog1} \vdash (-0-) -asx \rightarrow^* \text{sourcenode } ax \rangle$
have $\text{prog1} \vdash (-Entry-) -((-Entry-), (\lambda s. \text{True}), (-0-)) \#asx \rightarrow^* \text{sourcenode } ax$
by $(\text{fastforce intro: While-CFG.Cons-path simp: valid-edge-def valid-node-def})$
from $\langle \text{prog1} \vdash (-0-) -asx \rightarrow^* \text{sourcenode } ax \rangle \langle \text{valid-edge prog1 } ax \rangle$

```

obtain  $lx$  where  $[simp]:sourcenode\ ax = (-\ lx\ -)$ 
  by  $(cases\ sourcenode\ ax)\ auto$ 
with  $\langle prog1 \vdash (-Entry-) - ((-Entry-), (\lambda s. True)_{\checkmark}, (-0-)) \# asx \rightarrow^* sourcenode\ ax \rangle$ 
have  $prog1;;prog2 \vdash (-Entry-) - ((-Entry-), (\lambda s. True)_{\checkmark}, (-0-)) \# asx \rightarrow^*$ 
   $sourcenode\ ax$ 
  by  $(fastforce\ intro:path-SeqFirst)$ 
from  $\langle prog2 \vdash (-Entry-) - as' \rightarrow^* (-\ l'\ -) \rangle$  obtain  $ax'\ asx'$ 
  where  $prog2 \vdash (-Entry-) - ax' \# asx' \rightarrow^* (-\ l'\ -)$ 
  by  $(cases\ as', auto\ elim: While-CFG.path.cases)$ 
hence  $(-Entry-) = sourcenode\ ax'$  and  $valid-edge\ prog2\ ax'$ 
  and  $prog2 \vdash targetnode\ ax' - asx' \rightarrow^* (-\ l'\ -)$ 
  by  $(auto\ intro: While-CFG.path-split-Cons)$ 
hence  $targetnode\ ax' = (-0-)$  by  $(fastforce\ dest: WCFG-EntryD\ simp: valid-edge-def)$ 
from  $\langle valid-edge\ prog1\ ax \rangle \langle (-Exit-) = targetnode\ ax \rangle$ 
have  $prog1;;prog2 \vdash sourcenode\ ax - kind\ ax \rightarrow (-0-) \oplus \# : prog1$ 
  by  $(fastforce\ intro: WCFG-SeqConnect\ simp: valid-edge-def)$ 
have  $\exists as. prog1;;prog2 \vdash sourcenode\ ax - as \rightarrow^* (-\ l\ -)$ 
proof  $(cases\ asx' = [])$ 
  case  $True$ 
    with  $\langle prog2 \vdash targetnode\ ax' - asx' \rightarrow^* (-\ l'\ -) \rangle \langle targetnode\ ax' = (-0-) \rangle$ 
    have  $l' = 0$  by  $(auto\ elim: While-CFG.path.cases)$ 
    with  $\langle prog1;;prog2 \vdash sourcenode\ ax - kind\ ax \rightarrow (-0-) \oplus \# : prog1 \rangle$ 
    have  $prog1;;prog2 \vdash sourcenode\ ax - [(sourcenode\ ax, kind\ ax, (-\ l\ -))] \rightarrow^*$ 
       $(-\ l\ -)$ 
    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  $\langle prog2 \vdash targetnode\ ax' - asx' \rightarrow^* (-\ l'\ -) \rangle \langle targetnode\ ax' = (-0-) \rangle$ 
    have  $prog1;;prog2 \vdash (-0-) \oplus \# : prog1 - asx' \oplus s \# : prog1 \rightarrow^* (-\ l'\ -) \oplus \# : prog1$ 
      by  $(fastforce\ intro:path-SeqSecond)$ 
    hence  $prog1;;prog2 \vdash (-0-) \oplus \# : prog1 - asx' \oplus s \# : prog1 \rightarrow^* (-\ l\ -)$  by  $simp$ 
    with  $\langle prog1;;prog2 \vdash sourcenode\ ax - kind\ ax \rightarrow (-0-) \oplus \# : prog1 \rangle$ 
    have  $prog1;;prog2 \vdash sourcenode\ ax - (sourcenode\ ax, kind\ ax, (-0-) \oplus \# : prog1) \#$ 
       $(asx' \oplus s \# : prog1) \rightarrow^* (-\ l\ -)$ 
    by  $(fastforce\ intro: While-CFG.Cons-path\ simp: valid-node-def\ valid-edge-def)$ 
    thus  $?thesis$  by  $blast$ 
  qed
then obtain  $asx''$  where  $prog1;;prog2 \vdash sourcenode\ ax - asx'' \rightarrow^* (-\ l\ -)$  by
 $blast$ 
with  $\langle prog1;;prog2 \vdash (-Entry-) - ((-Entry-), (\lambda s. True)_{\checkmark}, (-0-)) \# asx \rightarrow^*$ 
   $sourcenode\ ax \rangle$ 
have  $prog1;;prog2 \vdash (-Entry-) - (((-Entry-), (\lambda s. True)_{\checkmark}, (-0-)) \# asx) @ asx'' \rightarrow^*$ 
   $(-\ l\ -)$ 
  by  $(rule\ While-CFG.path-Append)$ 
with  $\langle prog1;;prog2 \vdash (-\ l\ -) - as \oplus s \# : prog1 \rightarrow^* (-Exit-) \rangle$ 
show  $?thesis$  by  $blast$ 

```

```

qed
next
case (Cond b prog1 prog2)
note IH1 =  $\langle \bigwedge l. l < \# : \text{prog1} \implies$ 
 $(\exists as. \text{prog1} \vdash (-l-) -as \rightarrow^* (-Exit-)) \wedge (\exists as. \text{prog1} \vdash (-Entry-) -as \rightarrow^* (-l-)) \rangle$ 
note IH2 =  $\langle \bigwedge l. l < \# : \text{prog2} \implies$ 
 $(\exists as. \text{prog2} \vdash (-l-) -as \rightarrow^* (-Exit-)) \wedge (\exists as. \text{prog2} \vdash (-Entry-) -as \rightarrow^* (-l-)) \rangle$ 
show ?case
proof(cases l = 0)
case True
from IH1[of 0] obtain as where prog1  $\vdash (-0-) -as \rightarrow^* (-Exit-)$  by blast
hence if (b) prog1 else prog2  $\vdash (-0-) \oplus 1 -as \oplus s 1 \rightarrow^* (-Exit-) \oplus 1$ 
by(fastforce intro:path-CondTrue)
with WCFG-CondTrue[of b prog1 prog2] have if (b) prog1 else prog2  $\vdash$ 
 $(-0-) -((-0-), (\lambda s. \text{interpret } b \ s = \text{Some true})_{\checkmark}, (-0-) \oplus 1) \# (as \oplus s 1) \rightarrow^*$ 
 $(-Exit-) \oplus 1$ 
by(fastforce intro:While-CFG.Cons-path simp:valid-edge-def valid-node-def)
with True have if (b) prog1 else prog2  $\vdash$ 
 $(-l-) -((-0-), (\lambda s. \text{interpret } b \ s = \text{Some true})_{\checkmark}, (-0-) \oplus 1) \# (as \oplus s 1) \rightarrow^*$ 
 $(-Exit-)$  by simp
moreover
from WCFG-Entry[of if (b) prog1 else prog2] True
have if (b) prog1 else prog2  $\vdash (-Entry-) -[((-Entry-), (\lambda s. \text{True})_{\checkmark}, (-0-))] \rightarrow^*$ 
 $(-l-)$ 
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  $\vdash (-l'-) -as \rightarrow^* (-Exit-)$ 
and prog1  $\vdash (-Entry-) -as' \rightarrow^* (-l'-)$  by blast
from  $\langle \text{prog1} \vdash (-l'-) -as \rightarrow^* (-Exit-) \rangle$ 
have if (b) prog1 else prog2  $\vdash (-l'-) \oplus 1 -as \oplus s 1 \rightarrow^* (-Exit-) \oplus 1$ 
by(fastforce intro:path-CondTrue)
hence if (b) prog1 else prog2  $\vdash (-l-) -as \oplus s 1 \rightarrow^* (-Exit-)$ 
by simp
from  $\langle \text{prog1} \vdash (-Entry-) -as' \rightarrow^* (-l'-) \rangle$  obtain ax asx
where prog1  $\vdash (-Entry-) -ax \# asx \rightarrow^* (-l'-)$ 
by(cases as', auto elim:While-CFG.cases)
hence  $(-Entry-) = \text{sourcenode } ax$  and valid-edge prog1 ax
and prog1  $\vdash \text{targetnode } ax -asx \rightarrow^* (-l'-)$ 
by(auto intro:While-CFG.path-split-Cons)
hence targetnode ax =  $(-0-)$  by(fastforce dest:WCFG-EntryD simp:valid-edge-def)
with  $\langle \text{prog1} \vdash \text{targetnode } ax -asx \rightarrow^* (-l'-) \rangle$ 

```

```

have if (b) prog1 else prog2  $\vdash (-0-) \oplus 1 -asx \oplus s 1 \rightarrow^* (-l' -) \oplus 1$ 
  by (fastforce intro:path-CondTrue)
with WCFG-CondTrue[of b prog1 prog2]
have if (b) prog1 else prog2  $\vdash (-0-)$ 
   $-((-0-), (\lambda s. \text{interpret } b \ s = \text{Some true})_{\checkmark}, (-0-) \oplus 1) \# (asx \oplus s 1) \rightarrow^*$ 
   $(-l' -) \oplus 1$ 
  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  $\vdash (-\text{Entry-}) -((-Entry-), (\lambda s. \text{True})_{\checkmark}, (-0-)) \#$ 
   $((-0-), (\lambda s. \text{interpret } b \ s = \text{Some true})_{\checkmark}, (-0-) \oplus 1) \# (asx \oplus s 1) \rightarrow^*$ 
   $(-l' -) \oplus 1$ 
  by (fastforce intro:While-CFG.Cons-path simp:valid-edge-def)
with (if (b) prog1 else prog2  $\vdash (-l -) -as \oplus s 1 \rightarrow^* (-\text{Exit-})$ )
show ?thesis by simp blast
next
case False
hence  $\# : \text{prog1} \leq l'$  by simp
then obtain  $l''$  where  $[simp]: l' = l'' + \# : \text{prog1}$  and  $l'' = l' - \# : \text{prog1}$ 
  by simp
from  $\langle l < \# : (\text{if } (b) \text{ prog1 else prog2}) \rangle$ 
have  $l'' < \# : \text{prog2}$  by simp
from IH2[OF this] obtain  $as \ as'$  where  $\text{prog2} \vdash (-l'' -) -as \rightarrow^* (-\text{Exit-})$ 
  and  $\text{prog2} \vdash (-\text{Entry-}) -as' \rightarrow^* (-l'' -)$  by blast
from  $\langle \text{prog2} \vdash (-l'' -) -as \rightarrow^* (-\text{Exit-}) \rangle$ 
have if (b) prog1 else prog2  $\vdash (-l'' -) \oplus (\# : \text{prog1} + 1)$ 
   $-as \oplus s (\# : \text{prog1} + 1) \rightarrow^* (-\text{Exit-}) \oplus (\# : \text{prog1} + 1)$ 
  by (fastforce intro:path-CondFalse)
hence if (b) prog1 else prog2  $\vdash (-l -) -as \oplus s (\# : \text{prog1} + 1) \rightarrow^* (-\text{Exit-})$ 
  by (simp add:nat-add-assoc)
from  $\langle \text{prog2} \vdash (-\text{Entry-}) -as' \rightarrow^* (-l'' -) \rangle$  obtain  $ax \ asx$ 
  where  $\text{prog2} \vdash (-\text{Entry-}) -ax \# asx \rightarrow^* (-l'' -)$ 
  by (cases  $as'$ , auto elim:While-CFG.cases)
hence  $(-\text{Entry-}) = \text{sourcenode } ax$  and  $\text{valid-edge } \text{prog2 } ax$ 
  and  $\text{prog2} \vdash \text{targetnode } ax -asx \rightarrow^* (-l'' -)$ 
  by (auto intro:While-CFG.path-split-Cons)
hence  $\text{targetnode } ax = (-0-)$  by (fastforce dest:WCFG-EntryD simp:valid-edge-def)
with  $\langle \text{prog2} \vdash \text{targetnode } ax -asx \rightarrow^* (-l'' -) \rangle$ 
have if (b) prog1 else prog2  $\vdash (-0-) \oplus (\# : \text{prog1} + 1) -asx \oplus s (\# : \text{prog1} +$ 
   $1) \rightarrow^*$ 
   $(-l'' -) \oplus (\# : \text{prog1} + 1)$ 
  by (fastforce intro:path-CondFalse)
with WCFG-CondFalse[of b prog1 prog2]
have if (b) prog1 else prog2  $\vdash (-0-)$ 
   $-((-0-), (\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark}, (-0-) \oplus (\# : \text{prog1} + 1)) \#$ 
   $(asx \oplus s (\# : \text{prog1} + 1)) \rightarrow^* (-l'' -) \oplus (\# : \text{prog1} + 1)$ 
  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  $\vdash (-\text{Entry-}) -((-Entry-), (\lambda s. \text{True})_{\checkmark}, (-0-)) \#$ 
   $((-0-), (\lambda s. \text{interpret } b \ s = \text{Some false})_{\checkmark}, (-0-) \oplus (\# : \text{prog1} + 1)) \#$ 

```

```

    (asx  $\oplus$  s (#:prog1 + 1))  $\rightarrow^*$  (- l'' -)  $\oplus$  (#:prog1 + 1)
  by (fastforce intro: While-CFG.Cons-path simp: valid-edge-def)
with
   $\langle$ if (b) prog1 else prog2  $\vdash$  (- l -)  $\rightarrow$  as  $\oplus$  s (#:prog1 + 1)  $\rightarrow^*$  (-Exit-) $\rangle$ 
show ?thesis by (simp add: nat-add-assoc, blast)
qed
qed
next
case (While b prog')
note IH =  $\langle \bigwedge l. l < \# : \text{prog}' \implies$ 
  ( $\exists$  as.  $\text{prog}' \vdash$  (- l -)  $\rightarrow$  as  $\rightarrow^*$  (-Exit-))  $\wedge$  ( $\exists$  as.  $\text{prog}' \vdash$  (-Entry-)  $\rightarrow$  as  $\rightarrow^*$  (- l -)) $\rangle$ 
show ?case
proof (cases l < 1)
  case True
  from WCFG-Entry[of while (b) prog']
  have while (b) prog'  $\vdash$  (-Entry-)  $\rightarrow$  [((-Entry-), ( $\lambda s. \text{True}$ ) $\surd$ , (-0-))]  $\rightarrow^*$  (-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'  $\vdash$  (-1-)  $\rightarrow$  [((-1-), ( $\uparrow id$ , (-Exit-))]  $\rightarrow^*$  (-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'  $\vdash$  (-0-)  $\rightarrow$  [((-0-), ( $\lambda s. \text{interpret } b \ s = \text{Some false}$ ) $\surd$ , (-1-))#
    [((-1-), ( $\uparrow id$ , (-Exit-))]  $\rightarrow^*$  (-Exit-)
  by (fastforce intro: While-CFG.Cons-path While-CFG.empty-path
    simp: valid-node-def valid-edge-def)
  with  $\langle$ while (b) prog'  $\vdash$  (-Entry-)  $\rightarrow$  [((-Entry-), ( $\lambda s. \text{True}$ ) $\surd$ , (-0-))]  $\rightarrow^*$  (-0-) $\rangle$  True
  show ?thesis by simp blast
next
case False
hence 1  $\leq$  l by simp
thus ?thesis
proof (cases l < 2)
  case True
  with 1  $\leq$  l have [simp]: l = 1 by simp
  from WCFG-WhileFalseSkip[of b prog']
  have while (b) prog'  $\vdash$  (-1-)  $\rightarrow$  [((-1-), ( $\uparrow id$ , (-Exit-))]  $\rightarrow^*$  (-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'  $\vdash$  (-0-)
   $\rightarrow$  [((-0-), ( $\lambda s. \text{interpret } b \ s = \text{Some false}$ ) $\surd$ , (-1-))]  $\rightarrow^*$  (-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'  $\vdash$  (-Entry-)  $\rightarrow$  [((-Entry-), ( $\lambda s. \text{True}$ ) $\surd$ , (-0-))#
    [((-0-), ( $\lambda s. \text{interpret } b \ s = \text{Some false}$ ) $\surd$ , (-1-))]  $\rightarrow^*$  (-1-)
  by (fastforce intro: While-CFG.Cons-path simp: valid-node-def valid-edge-def)

```

with $\langle \text{while } (b) \text{ prog}' \vdash (-1-) - [((-1-), \uparrow \text{id}, (-\text{Exit-}))] \rightarrow^* (-\text{Exit-}) \rangle$
show $?thesis$ **by** *simp blast*
next
case *False*
with $\langle 1 \leq l \rangle$ **have** $2 \leq l$ **by** *simp*
then obtain l' **where** $[simp]: l = l' + 2$ **and** $l' = l - 2$
by $(simp \text{ del: add-2-eq-Suc})$
from $\langle l < \#; \text{while } (b) \text{ prog}' \rangle$ **have** $l' < \#; \text{prog}'$ **by** *simp*
from $IH[OF \text{ this}]$ **obtain** $as \text{ as}'$ **where** $\text{prog}' \vdash (-l' -) - as \rightarrow^* (-\text{Exit-})$
and $\text{prog}' \vdash (-\text{Entry-}) - as' \rightarrow^* (-l' -)$ **by** *blast*
from $\langle \text{prog}' \vdash (-l' -) - as \rightarrow^* (-\text{Exit-}) \rangle$ **obtain** $ax \text{ asx}$ **where**
 $\text{prog}' \vdash (-l' -) - asx @ [ax] \rightarrow^* (-\text{Exit-})$
by $(induct \text{ as rule: rev-induct, auto elim: While-CFG.cases})$
hence $\text{prog}' \vdash (-l' -) - asx \rightarrow^* \text{sourcenode } ax$ **and** $\text{valid-edge } \text{prog}' \text{ } ax$
and $(-\text{Exit-}) = \text{targetnode } ax$
by $(auto \text{ intro: While-CFG.path-split-snoc})$
then obtain lx **where** $\text{sourcenode } ax = (-lx -)$
by $(cases \text{ sourcenode } ax) \text{ auto}$
with $\langle \text{prog}' \vdash (-l' -) - asx \rightarrow^* \text{sourcenode } ax \rangle$
have $\text{while } (b) \text{ prog}' \vdash (-l' -) \oplus 2 - asx \oplus s 2 \rightarrow^* \text{sourcenode } ax \oplus 2$
by $(fastforce \text{ intro: path-While simp del: label-incr.simps})$
from $WCFG\text{-}WhileFalseSkip[of \text{ b prog}']$
have $\text{while } (b) \text{ prog}' \vdash (-1-) - [((-1-), \uparrow \text{id}, (-\text{Exit-}))] \rightarrow^* (-\text{Exit-})$
by $(fastforce \text{ intro: While-CFG.Cons-path While-CFG.empty-path simp: valid-node-def valid-edge-def})$
with $WCFG\text{-}WhileFalse[of \text{ b prog}']$
have $\text{while } (b) \text{ prog}' \vdash (-0-) - ((-0-), (\lambda s. \text{interpret } b \text{ } s = \text{Some false})_{\checkmark}, (-1-)) \#$
 $[((-1-), \uparrow \text{id}, (-\text{Exit-}))] \rightarrow^* (-\text{Exit-})$
by $(fastforce \text{ intro: While-CFG.Cons-path simp: valid-node-def valid-edge-def})$
with $\langle \text{valid-edge } \text{prog}' \text{ } ax \rangle \langle (-\text{Exit-}) = \text{targetnode } ax \rangle \langle \text{sourcenode } ax = (-lx -) \rangle$
have $\text{while } (b) \text{ prog}' \vdash \text{sourcenode } ax \oplus 2 - (\text{sourcenode } ax \oplus 2, \text{kind } ax, (-0-)) \#$
 $((-0-), (\lambda s. \text{interpret } b \text{ } s = \text{Some false})_{\checkmark}, (-1-)) \#$
 $[((-1-), \uparrow \text{id}, (-\text{Exit-}))] \rightarrow^* (-\text{Exit-})$
by $(fastforce \text{ intro: While-CFG.Cons-path dest: WCFG-WhileBodyExit simp: valid-node-def valid-edge-def})$
with $\langle \text{while } (b) \text{ prog}' \vdash (-l' -) \oplus 2 - asx \oplus s 2 \rightarrow^* \text{sourcenode } ax \oplus 2 \rangle$
have $\text{path: while } (b) \text{ prog}' \vdash (-l' -) \oplus 2 - (asx \oplus s 2) @$
 $((\text{sourcenode } ax \oplus 2, \text{kind } ax, (-0-)) \#$
 $((-0-), (\lambda s. \text{interpret } b \text{ } s = \text{Some false})_{\checkmark}, (-1-)) \#$
 $[((-1-), \uparrow \text{id}, (-\text{Exit-}))]) \rightarrow^* (-\text{Exit-})$
by $(rule \text{ While-CFG.path-Append})$
from $\langle \text{prog}' \vdash (-\text{Entry-}) - as' \rightarrow^* (-l' -) \rangle$ **obtain** $ax' \text{ asx}'$
where $\text{prog}' \vdash (-\text{Entry-}) - ax' \# asx' \rightarrow^* (-l' -)$
by $(cases \text{ as}', auto \text{ elim: While-CFG.cases})$
hence $(-\text{Entry-}) = \text{sourcenode } ax'$ **and** $\text{valid-edge } \text{prog}' \text{ } ax'$
and $\text{prog}' \vdash \text{targetnode } ax' - asx' \rightarrow^* (-l' -)$
by $(auto \text{ intro: While-CFG.path-split-Cons})$
hence $\text{targetnode } ax' = (-0-) \text{ by } (fastforce \text{ dest: WCFG-EntryD simp: valid-edge-def})$

```

with  $\langle \text{prog}' \vdash \text{targetnode } ax' - asx' \rightarrow^* (- l' -) \rangle$ 
have  $\text{while } (b) \text{ prog}' \vdash (-0-) \oplus 2 - asx' \oplus s 2 \rightarrow^* (- l' -) \oplus 2$ 
  by (fastforce intro:path-While)
with WCFG-WhileTrue[of b prog']
have  $\text{while } (b) \text{ prog}' \vdash (-0-)$ 
   $-(((-0-), (\lambda s. \text{interpret } b \ s = \text{Some true})_{\checkmark}, (-0-) \oplus 2) \# (asx' \oplus s 2) \rightarrow^*$ 
   $(- l' -) \oplus 2$ 
  by (fastforce intro:While-CFG.Cons-path simp:valid-node-def valid-edge-def)
with WCFG-Entry[of while (b) prog']
have  $\text{while } (b) \text{ prog}' \vdash (-\text{Entry-}) -(((-\text{Entry-}), (\lambda s. \text{True})_{\checkmark}, (-0-)) \#$ 
   $((-0-), (\lambda s. \text{interpret } b \ s = \text{Some true})_{\checkmark}, (-0-) \oplus 2) \# (asx' \oplus s 2) \rightarrow^*$ 
   $(- l' -) \oplus 2$ 
  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  $\exists as. \text{prog} \vdash n - as \rightarrow^* (-\text{Exit-})$ 
proof (cases n)
  case (Node l)
    with  $\langle \text{valid-node prog } n \rangle$  have  $l < \#:\text{prog}$ 
    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  $\text{prog} \vdash (-\text{Entry-}) -((( -\text{Entry-}), (\lambda s. \text{False})_{\checkmark}, (-\text{Exit-})) \rightarrow^* (-\text{Exit-})$ 
    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  $\text{prog} \vdash n - [] \rightarrow^* (-\text{Exit-})$ 
    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  $\exists as. \text{prog} \vdash (-\text{Entry-}) - as \rightarrow^* n$ 
proof (cases n)
  case (Node l)
    with  $\langle \text{valid-node prog } n \rangle$  have  $l < \#:\text{prog}$ 
    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-), (λs. False)✓, (-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-) ((λl'. (- 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-) ((λl'. (- 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'.  $\exists a'. \text{valid-edge prog } a' \wedge \text{sourcenode } a' = n \wedge$ 
         targetnode  $a' = n'$ }
proof -
  have {n'.  $\exists a'. \text{valid-edge prog } a' \wedge \text{sourcenode } a' = n \wedge$ 
        targetnode  $a' = n'$ }  $\subseteq \{n. \text{valid-node prog } n\}$ 
  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  $\exists as. \text{prog} \vdash (-\text{Entry-}) -as \rightarrow^* n$  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  $\exists as. \text{prog} \vdash n -as \rightarrow^* (-\text{Exit-})$  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'.  $\exists a'. \text{valid-edge prog } a' \wedge \text{sourcenode } a' = n \wedge$ 
               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* $\Rightarrow$ *cmd* * *state* $\Rightarrow$ *bool*

and *red'* :: *cmd* $\Rightarrow$ *state* $\Rightarrow$ *cmd* $\Rightarrow$ *state* $\Rightarrow$ *bool*

(($(1 \langle -, / - \rangle) \rightarrow / (1 \langle -, / - \rangle)$) [0,0,0,0] 81)

where

$\langle c, s \rangle \rightarrow \langle c', s' \rangle == \text{red } (c, s) (c', s')$

| *RedLAss*:

$\langle V := e, s \rangle \rightarrow \langle \text{Skip}, s(V := (\text{interpret } e \ s)) \rangle$

| *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*:

$\langle \text{Skip};; c_2, s \rangle \rightarrow \langle c_2, s \rangle$

| *RedCondTrue*:

$\text{interpret } b \ s = \text{Some true} \Longrightarrow \langle \text{if } (b) \ c_1 \ \text{else } c_2, s \rangle \rightarrow \langle c_1, s \rangle$

| *RedCondFalse*:

$\text{interpret } b \ s = \text{Some false} \Longrightarrow \langle \text{if } (b) \ c_1 \ \text{else } c_2, s \rangle \rightarrow \langle c_2, s \rangle$

| *RedWhileTrue*:

$\text{interpret } b \ s = \text{Some true} \Longrightarrow \langle \text{while } (b) \ c, s \rangle \rightarrow \langle c;; \text{while } (b) \ c, s \rangle$

| *RedWhileFalse*:

$\text{interpret } b \ s = \text{Some false} \Longrightarrow \langle \text{while } (b) \ c, s \rangle \rightarrow \langle \text{Skip}, s \rangle$

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

abbreviation *reds* :: *cmd* $\Rightarrow$ *state* $\Rightarrow$ *cmd* $\Rightarrow$ *state* $\Rightarrow$ *bool*

(($(1 \langle -, / - \rangle) \rightarrow * / (1 \langle -, / - \rangle)$) [0,0,0,0] 81) **where**

$\langle c, s \rangle \rightarrow * \langle c', s' \rangle == \text{red}^{**} (c, s) (c', s')$

4.8.2 Label Semantics

inductive *step* :: *cmd* $\Rightarrow$ *cmd* $\Rightarrow$ *state* $\Rightarrow$ *nat* $\Rightarrow$ *cmd* $\Rightarrow$ *state* $\Rightarrow$ *nat* $\Rightarrow$ *bool*

(($(- \vdash (1 \langle -, / - \rangle)) \rightsquigarrow / (1 \langle -, / - \rangle)$) [51,0,0,0,0,0,0] 81)

where

StepLAss:

$V := e \vdash \langle V := e, s, 0 \rangle \rightsquigarrow \langle \text{Skip}, s(V := (\text{interpret } e \ s)), 1 \rangle$

| *StepSeq*:
 $\llbracket \text{labels } (c_1;;c_2) \ l \ (Skip;;c_2); \text{labels } (c_1;;c_2) \ \# : c_1 \ c_2; \ l < \# : c_1 \rrbracket$
 $\implies c_1;;c_2 \vdash \langle Skip;;c_2,s,l \rangle \rightsquigarrow \langle c_2,s,\# : c_1 \rangle$

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

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

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

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

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

| *StepRecSeq1*:
 $\text{prog} \vdash \langle c,s,l \rangle \rightsquigarrow \langle c',s',l' \rangle$
 $\implies \text{prog};;c_2 \vdash \langle c;;c_2,s,l \rangle \rightsquigarrow \langle c';;c_2,s',l' \rangle$

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

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

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

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

lemma *step-label-less*:

$\text{prog} \vdash \langle c,s,l \rangle \rightsquigarrow \langle c',s',l' \rangle \implies l < \# : \text{prog} \wedge l' < \# : \text{prog}$

proof(*induct rule:step.induct*)

case (*StepSeq* $c_1 \ c_2 \ l \ s$)

```

from  $\langle \text{labels } (c_1;;c_2) \ l \ (Skip;;c_2) \rangle$ 
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  $\langle \text{labels } (while \ (b) \ cx) \ l \ (Skip;;while \ (b) \ cx) \rangle$ 
  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 $\text{steps} :: \text{cmd} \Rightarrow \text{cmd} \Rightarrow \text{state} \Rightarrow \text{nat} \Rightarrow \text{cmd} \Rightarrow \text{state} \Rightarrow \text{nat} \Rightarrow \text{bool}$
 $((- \vdash (1 \langle -, / -, / - \rangle) \rightsquigarrow^* / (1 \langle -, / -, / - \rangle))) [51, 0, 0, 0, 0, 0, 0] \ 81)$ **where**
 $\text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle ==$
 $(\lambda(c, s, l) \ (c', s', l'). \text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle)^{**} \ (c, s, l) \ (c', s', l')$

4.8.3 Proof of bisimulation of $\langle c, s \rangle \rightarrow \langle c', s' \rangle$ and $\text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle$ **via** labels

From $\text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle$ **to** $\langle c, s \rangle \rightarrow \langle c', s' \rangle$

lemma *step-red*:

$\text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \implies \langle c, s \rangle \rightarrow \langle c', s' \rangle$
by(induct rule:step.induct, rule RedLAss, auto intro:red.intros)

lemma *steps-reds*:

$\text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle \implies \langle c, s \rangle \rightarrow^* \langle c', s' \rangle$
proof(induct rule:converse-rtrancpl-induct3)
case refl **thus** ?case **by** simp
next
case (step c s l c'' s'' l'')
then **have** $\text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow \langle c'', s'', l'' \rangle$
and $\langle c'', s'' \rangle \rightarrow^* \langle c', s' \rangle$ **by** simp-all
from $\langle \text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow \langle c'', s'', l'' \rangle \rangle$ **have** $\langle c, s \rangle \rightarrow \langle c'', s'' \rangle$
by(fastforce intro:step-red)
with $\langle \langle c'', s'' \rangle \rightarrow^* \langle c', s' \rangle \rangle$ **show** ?case
by(fastforce intro:converse-rtrancpl-into-rtrancpl)
qed

From $\langle c, s \rangle \rightarrow \langle c', s' \rangle$ **and** labels **to** $\text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle$

lemma *red-step*:

$\llbracket \text{labels } \text{prog } l \ c; \langle c, s \rangle \rightarrow \langle c', s' \rangle \rrbracket$
 $\implies \exists l'. \text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \wedge \text{labels } \text{prog } l' \ c'$
proof(induct arbitrary:c' rule:labels.induct)

```

case (Labels-Base c)
from  $\langle c, s \rangle \rightarrow \langle c', s' \rangle$  show ?case
proof(induct rule:red-induct)
  case (RedLAss V e s)
  have  $V := e \vdash \langle V := e, s, 0 \rangle \rightsquigarrow \langle \text{Skip}, s(V := (\text{interpret } e \ s)), 1 \rangle$  by(rule StepLAss)
  have labels (V:=e) 1 Skip by(fastforce intro:Labels-LAss)
  with  $\langle V := e \vdash \langle V := e, s, 0 \rangle \rightsquigarrow \langle \text{Skip}, s(V := (\text{interpret } e \ s)), 1 \rangle$  show ?case by
blast
next
case (SeqRed c1 s c1' s' c2)
from  $\exists l'. c_1 \vdash \langle c_1, s, 0 \rangle \rightsquigarrow \langle c_1', s', l' \rangle \wedge \text{labels } c_1 \ l' \ c_1'$ 
obtain l' where  $c_1 \vdash \langle c_1, s, 0 \rangle \rightsquigarrow \langle c_1', s', l' \rangle$  and labels c1 l' c1' by blast
from  $\langle c_1 \vdash \langle c_1, s, 0 \rangle \rightsquigarrow \langle c_1', s', l' \rangle$  have  $c_1;;c_2 \vdash \langle c_1;;c_2, s, 0 \rangle \rightsquigarrow \langle c_1';c_2, s', l' \rangle$ 
by(rule StepRecSeq1)
moreover
from  $\langle \text{labels } c_1 \ l' \ c_1' \rangle$  have labels (c1::c2) l' (c1::c2) by(rule Labels-Seq1)
ultimately show ?case by blast
next
case (RedSeq c2 s)
have labels c2 0 c2 by(rule Labels.Labels-Base)
hence labels (Skip;;c2) (0 + #:Skip) c2 by(rule Labels-Seq2)
have labels (Skip;;c2) 0 (Skip;;c2) by(rule Labels.Labels-Base)
with  $\langle \text{labels } (Skip;;c_2) \ (0 + \#:Skip) \ c_2 \rangle$ 
have  $\text{Skip};;c_2 \vdash \langle \text{Skip};;c_2, s, 0 \rangle \rightsquigarrow \langle c_2, s, \#:Skip \rangle$ 
by(fastforce intro:StepSeq)
with  $\langle \text{labels } (Skip;;c_2) \ (0 + \#:Skip) \ c_2 \rangle$  show ?case by auto
next
case (RedCondTrue b s c1 c2)
from  $\langle \text{interpret } b \ s = \text{Some true} \rangle$ 
have  $\text{if } (b) \ c_1 \ \text{else } c_2 \vdash \langle \text{if } (b) \ c_1 \ \text{else } c_2, s, 0 \rangle \rightsquigarrow \langle c_1, s, 1 \rangle$ 
by(rule StepCondTrue)
have labels (if (b) c1 else c2) (0 + 1) c1
by(rule Labels-CondTrue, rule Labels.Labels-Base)
with  $\langle \text{if } (b) \ c_1 \ \text{else } c_2 \vdash \langle \text{if } (b) \ c_1 \ \text{else } c_2, s, 0 \rangle \rightsquigarrow \langle c_1, s, 1 \rangle$  show ?case by
auto
next
case (RedCondFalse b s c1 c2)
from  $\langle \text{interpret } b \ s = \text{Some false} \rangle$ 
have  $\text{if } (b) \ c_1 \ \text{else } c_2 \vdash \langle \text{if } (b) \ c_1 \ \text{else } c_2, s, 0 \rangle \rightsquigarrow \langle c_2, s, \#:c_1 + 1 \rangle$ 
by(rule StepCondFalse)
have labels (if (b) c1 else c2) (0 + #:c1 + 1) c2
by(rule Labels-CondFalse, rule Labels.Labels-Base)
with  $\langle \text{if } (b) \ c_1 \ \text{else } c_2 \vdash \langle \text{if } (b) \ c_1 \ \text{else } c_2, s, 0 \rangle \rightsquigarrow \langle c_2, s, \#:c_1 + 1 \rangle$ 
show ?case by auto
next
case (RedWhileTrue b s c)
from  $\langle \text{interpret } b \ s = \text{Some true} \rangle$ 
have  $\text{while } (b) \ c \vdash \langle \text{while } (b) \ c, s, 0 \rangle \rightsquigarrow \langle c;; \text{while } (b) \ c, s, 2 \rangle$ 
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 ⊢ ⟨while (b) c,s,0⟩ ∼ ⟨c;; while (b) c,s,2⟩)
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 ⊢ ⟨while (b) c,s,0⟩ ∼ ⟨Skip,s,1⟩
  by(rule StepWhileFalse)
have labels (while (b) c) 1 Skip by(rule Labels-WhileExit)
with (while (b) c ⊢ ⟨while (b) c,s,0⟩ ∼ ⟨Skip,s,1⟩) show ?case by auto
qed
next
case (Labels-LAss V e)
from ⟨⟨Skip,s⟩ → ⟨c',s'⟩⟩ have False by(auto elim:red.cases)
thus ?case by simp
next
case (Labels-Seq1 c1 l c c2)
note IH = ⟨∧ c'. ⟨c,s⟩ → ⟨c',s'⟩ ⟹
  ∃ l'. c1 ⊢ ⟨c,s,l⟩ ∼ ⟨c',s',l'⟩ ∧ labels c1 l' c'⟩
from ⟨⟨c;;c2,s⟩ → ⟨c',s'⟩⟩
have (c = Skip ∧ c' = c2 ∧ s = s') ∨ (∃ c''. c' = c'';;c2)
  by -(erule red.cases,auto)
thus ?case
proof
  assume [simp]: c = Skip ∧ c' = c2 ∧ s = s'
  from ⟨labels c1 l c⟩ have l < #:c1
    by(rule label-less-num-inner-nodes[simplified])
  have labels (c1;;c2) (0 + #:c1) c2
    by(rule Labels-Seq2,rule Labels-Base)
  from ⟨labels c1 l c⟩ have labels (c1;; c2) l (Skip;;c2)
    by(fastforce intro:Labels.Labels-Seq1)
  with ⟨labels (c1;;c2) (0 + #:c1) c2⟩ ⟨l < #:c1⟩
  have c1;; c2 ⊢ ⟨Skip;;c2,s,l⟩ ∼ ⟨c2,s, #:c1⟩
    by(fastforce intro:StepSeq)
  with ⟨labels (c1;;c2) (0 + #:c1) c2⟩ show ?case by auto
next
  assume ∃ c''. c' = c'';;c2
  then obtain c'' where [simp]: c' = c'';;c2 by blast
  with ⟨⟨c;;c2,s⟩ → ⟨c',s'⟩⟩ have ⟨c,s⟩ → ⟨c'',s'⟩
    by(auto elim!:red.cases,induct c2,auto)
  from IH[OF this] obtain l' where c1 ⊢ ⟨c,s,l⟩ ∼ ⟨c'',s',l'⟩
    and labels c1 l' c'' by blast
  from ⟨c1 ⊢ ⟨c,s,l⟩ ∼ ⟨c'',s',l'⟩⟩ have c1;;c2 ⊢ ⟨c;;c2,s,l⟩ ∼ ⟨c'';;c2,s',l'⟩
    by(rule StepRecSeq1)
  from ⟨labels c1 l' c''⟩ have labels (c1;;c2) l' (c'';;c2)
    by(rule Labels.Labels-Seq1)
  with ⟨c1;;c2 ⊢ ⟨c;;c2,s,l⟩ ∼ ⟨c'';;c2,s',l'⟩⟩ show ?case by auto
qed

```

```

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

```

```

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*:

$labels\ prog\ l\ c \implies (\langle c, s \rangle \rightarrow^* \langle c', s' \rangle) =$
 $(\exists l'.\ prog \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle \wedge labels\ prog\ l'\ c')$
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**
 $c \vdash (-\ l\ -) \rightarrow_{et} (-\ l'\ -)$ **with transfers and preds**

lemma *Skip-WCFG-edge-Exit*:

$\llbracket labels\ prog\ l\ Skip \rrbracket \implies prog \vdash (-\ l\ -) \rightarrow_{id} (-Exit-)$

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 c₂ l c₁*)

from $\langle c_2 \vdash (-\ l\ -) \rightarrow_{id} (-Exit-) \rangle$

have $c_1;;c_2 \vdash (-\ l\ -) \oplus \# : c_1 \rightarrow_{id} (-Exit-) \oplus \# : c_1$

by(*fastforce intro:WCFG-SeqSecond*)

thus *?case* **by**(*simp del:id-apply*)

next

case (*Labels-CondTrue c₁ l b c₂*)

from $\langle c_1 \vdash (-\ l\ -) \rightarrow_{id} (-Exit-) \rangle$

have *if* (*b*) c_1 *else* $c_2 \vdash (-\ l\ -) \oplus 1 \rightarrow_{id} (-Exit-) \oplus 1$

by(*fastforce intro:WCFG-CondThen*)

thus *?case* **by**(*simp del:id-apply*)

next

case (*Labels-CondFalse c₂ l b c₁*)

from $\langle c_2 \vdash (-\ l\ -) \rightarrow_{id} (-Exit-) \rangle$

have *if* (*b*) c_1 *else* $c_2 \vdash (-\ l\ -) \oplus (\# : c_1 + 1) \rightarrow_{id} (-Exit-) \oplus (\# : c_1 + 1)$

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 $\text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle$

obtains *et* **where** $\text{prog} \vdash (-l-) \rightarrow \text{et} \rightarrow (-l'-)$ **and** *transfer* $\text{et } s = s'$

and *pred* $\text{et } s$

proof –

from $\langle \text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \rangle$

have $\exists \text{et}. \text{prog} \vdash (-l-) \rightarrow \text{et} \rightarrow (-l'-) \wedge \text{transfer } \text{et } s = s' \wedge \text{pred } \text{et } s$

proof(*induct rule:step.induct*)

case (*StepLAss* $V \ e \ s$)

have $\text{pred } \uparrow(\lambda s. s(V := (\text{interpret } e \ s))) \ s$ **by** *simp*

have $V := e \vdash (-0-) \rightarrow \uparrow(\lambda s. s(V := (\text{interpret } e \ s))) \rightarrow (-1-)$

by(*rule WCFG-LAss*)

have $\text{transfer } \uparrow(\lambda s. s(V := (\text{interpret } e \ s))) \ s = s(V := (\text{interpret } e \ s))$ **by** *simp*

with $\langle \text{pred } \uparrow(\lambda s. s(V := (\text{interpret } e \ s))) \ s \rangle$

$\langle V := e \vdash (-0-) \rightarrow \uparrow(\lambda s. s(V := (\text{interpret } e \ s))) \rightarrow (-1-) \rangle$ **show** *?case* **by** *blast*

next

case (*StepSeq* $c_1 \ c_2 \ l \ s$)

from $\langle \text{labels } (c_1;;c_2) \ l \ (\text{Skip};;c_2) \rangle \langle l < \# : c_1 \rangle$ **have** $\text{labels } c_1 \ l \ \text{Skip}$

by(*auto elim:labels.cases intro:Labels-Base*)

hence $c_1 \vdash (-l-) \rightarrow \uparrow \text{id} \rightarrow (-\text{Exit}-)$

by(*fastforce intro:Skip-WCFG-edge-Exit*)

hence $c_1;;c_2 \vdash (-l-) \rightarrow \uparrow \text{id} \rightarrow (-0-) \oplus \# : c_1$

by(*rule WCFG-SeqConnect,simp*)

thus *?case* **by** *auto*

next

case (*StepSeqWhile* $b \ cx \ l \ s$)

from $\langle \text{labels } (\text{while } (b) \ cx) \ l \ (\text{Skip};;\text{while } (b) \ cx) \rangle$

obtain lx **where** $\text{labels } cx \ lx \ \text{Skip}$

and $[simp]: l = lx + 2$ **by**(*auto elim:labels.cases*)

hence $cx \vdash (-lx-) \rightarrow \uparrow \text{id} \rightarrow (-\text{Exit}-)$

by(*fastforce intro:Skip-WCFG-edge-Exit*)

hence $\text{while } (b) \ cx \vdash (-lx-) \oplus 2 \rightarrow \uparrow \text{id} \rightarrow (-0-)$

by(*fastforce intro:WCFG-WhileBodyExit*)

thus *?case* **by** *auto*

next

case (*StepCondTrue* $b \ s \ c_1 \ c_2$)

from $\langle \text{interpret } b \ s = \text{Some true} \rangle$

have $\text{pred } (\lambda s. \text{interpret } b \ s = \text{Some true})_{\surd} \ s$ **by** *simp*

moreover

have $\text{if } (b) \ c_1 \ \text{else } c_2 \vdash (-0-) \rightarrow (\lambda s. \text{interpret } b \ s = \text{Some true})_{\surd} \rightarrow (-0-) \oplus 1$

by(*rule WCFG-CondTrue*)

moreover

have $\text{transfer } (\lambda s. \text{interpret } b \ s = \text{Some true})_{\surd} \ s = s$ **by** *simp*

ultimately **show** *?case* **by** *auto*

next

case (*StepCondFalse* $b \ s \ c_1 \ c_2$)

from $\langle \text{interpret } b \ s = \text{Some false} \rangle$

have $\text{pred } (\lambda s. \text{interpret } b \ s = \text{Some false})_{\surd} \ s$ **by** *simp*

moreover

```

have if (b) c1 else c2 ⊢ (-0-) → (λs. interpret b s = Some false)✓ →
      (-0-) ⊕ (#:c1 + 1)
  by(rule WCFG-CondFalse)
moreover
have transfer (λs. interpret b s = Some false)✓ 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)✓ s by simp
moreover
have while (b) c ⊢ (-0-) → (λs. interpret b s = Some true)✓ → (-0-) ⊕ 2
  by(rule WCFG-WhileTrue)
moreover
have transfer (λs. interpret b s = Some true)✓ 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)✓ s by simp
moreover
have while (b) c ⊢ (-0-) → (λs. interpret b s = Some false)✓ → (-1-)
  by(rule WCFG-WhileFalse)
moreover
have transfer (λs. interpret b s = Some false)✓ s = s by simp
ultimately show ?case by auto
next
case (StepRecSeq1 prog c s l c' s' l' c2)
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;;c2 ⊢ (- l -) → et → (- l' -)
  by(fastforce intro:WCFG-SeqFirst)
ultimately show ?case by blast
next
case (StepRecSeq2 prog c s l c' s' l' c1)
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 c1;;prog ⊢ (- l -) ⊕ #:c1 → et → (- l' -) ⊕ #:c1
  by(fastforce intro:WCFG-SeqSecond)
ultimately show ?case by simp blast
next
case (StepRecCond1 prog c s l c' s' l' b c2)
from ⟨∃ et. prog ⊢ (- l -) → et → (- l' -) ∧ transfer et s = s' ∧ pred et s⟩
obtain et where prog ⊢ (- l -) → et → (- l' -)

```

and $\text{transfer } et \ s = s' \text{ and } \text{pred } et \ s \text{ by } \text{blast}$
 moreover
 from $\langle \text{prog} \vdash (-l-) \rightarrow et \rightarrow (-l'-) \rangle$
 have if $(b) \text{ prog else } c_2 \vdash (-l-) \oplus 1 \rightarrow et \rightarrow (-l'-) \oplus 1$
 by $(\text{fastforce intro:WCFG-CondThen})$
 ultimately show $?case \text{ by } \text{simp blast}$
 next
 case $(\text{StepRecCond2 prog } c \ s \ l \ c' \ s' \ l' \ b \ c_1)$
 from $\langle \exists et. \text{ prog} \vdash (-l-) \rightarrow et \rightarrow (-l'-) \wedge \text{transfer } et \ s = s' \wedge \text{pred } et \ s \rangle$
 obtain et where $\text{prog} \vdash (-l-) \rightarrow et \rightarrow (-l'-)$
 and $\text{transfer } et \ s = s' \text{ and } \text{pred } et \ s \text{ by } \text{blast}$
 moreover
 from $\langle \text{prog} \vdash (-l-) \rightarrow et \rightarrow (-l'-) \rangle$
 have if $(b) \ c_1 \text{ else } \text{prog} \vdash (-l-) \oplus (\# : c_1 + 1) \rightarrow et \rightarrow (-l'-) \oplus (\# : c_1 + 1)$
 by $(\text{fastforce intro:WCFG-CondElse})$
 ultimately show $?case \text{ by } \text{simp blast}$
 next
 case $(\text{StepRecWhile } cx \ c \ s \ l \ c' \ s' \ l' \ b)$
 from $\langle \exists et. \ cx \vdash (-l-) \rightarrow et \rightarrow (-l'-) \wedge \text{transfer } et \ s = s' \wedge \text{pred } et \ s \rangle$
 obtain et where $\text{cx} \vdash (-l-) \rightarrow et \rightarrow (-l'-)$
 and $\text{transfer } et \ s = s' \text{ and } \text{pred } et \ s \text{ by } \text{blast}$
 moreover
 hence while $(b) \ cx \vdash (-l-) \oplus 2 \rightarrow et \rightarrow (-l'-) \oplus 2$
 by $(\text{fastforce intro:WCFG-WhileBody})$
 ultimately show $?case \text{ by } \text{simp blast}$
 qed
 with that show $?thesis \text{ by } \text{blast}$
 qed

4.9.2 From $c \vdash (-l-) \rightarrow et \rightarrow (-l'-)$ with transfers and preds to $\text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle$

lemma *WCFG-edge-Exit-Skip*:

$\llbracket \text{prog} \vdash n \rightarrow et \rightarrow (-Exit-); n \neq (-Entry-) \rrbracket$
 $\implies \exists l. n = (-l-) \wedge \text{labels prog } l \text{ Skip} \wedge et = \uparrow id$

proof $(\text{induct prog } n \ et \ (-Exit-) \text{ rule:WCFG-induct})$

case *WCFG-Skip* show $?case \text{ by } (\text{fastforce intro:Labels-Base})$

next

case *WCFG-LAssSkip* show $?case \text{ by } (\text{fastforce intro:Labels-LAss})$

next

case $(\text{WCFG-SeqSecond } c_2 \ n \ et \ n' \ c_1)$

note $IH = \llbracket n' = (-Exit-); n' \neq (-Entry-) \rrbracket$

$\implies \exists l. n = (-l-) \wedge \text{labels } c_2 \ l \text{ Skip} \wedge et = \uparrow id$

from $\langle n' \oplus \# : c_1 = (-Exit-) \rangle$ **have** $n' = (-Exit-)$ **by** $(\text{cases } n') \text{ auto}$

from $IH[OF \text{ this } \langle n \neq (-Entry-) \rangle]$ **obtain** l **where** $[\text{simp}]: n = (-l-) \ et = \uparrow id$

and $\text{labels } c_2 \ l \text{ Skip} \text{ by } \text{blast}$

hence $\text{labels } (c_1;;c_2) \ (l + \# : c_1) \text{ Skip} \text{ by } (\text{fastforce intro:Labels-Seq2})$

thus $?case \text{ by } (\text{fastforce simp:id-def})$

```

next
  case (WCFG-CondThen  $c_1$   $n$   $et$   $n'$   $b$   $c_2$ )
  note  $IH = \langle \llbracket n' = (-Exit-); n \neq (-Entry-) \rrbracket \rangle$ 
     $\implies \exists l. n = (-l-) \wedge \text{labels } c_1 \ l \text{ Skip} \wedge et = \uparrow id$ 
  from  $\langle n' \oplus 1 = (-Exit-) \rangle$  have  $n' = (-Exit-)$  by (cases  $n'$ ) auto
  from  $IH[OF \text{ this } \langle n \neq (-Entry-) \rangle]$  obtain  $l$  where  $[simp]: n = (-l-) \ et = \uparrow id$ 
    and labels  $c_1 \ l \text{ Skip}$  by blast
  hence labels (if (b)  $c_1$  else  $c_2$ )  $(l + 1) \text{ Skip}$ 
    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  $IH = \langle \llbracket n' = (-Exit-); n \neq (-Entry-) \rrbracket \rangle$ 
     $\implies \exists l. n = (-l-) \wedge \text{labels } c_2 \ l \text{ Skip} \wedge et = \uparrow id$ 
  from  $\langle n' \oplus \# : c_1 + 1 = (-Exit-) \rangle$  have  $n' = (-Exit-)$  by (cases  $n'$ ) auto
  from  $IH[OF \text{ this } \langle n \neq (-Entry-) \rangle]$  obtain  $l$  where  $[simp]: n = (-l-) \ et = \uparrow id$ 
    and label: labels  $c_2 \ l \text{ Skip}$  by blast
  hence labels (if (b)  $c_1$  else  $c_2$ )  $(l + \# : c_1 + 1) \text{ Skip}$ 
    by (fastforce intro: Labels-CondFalse)
  thus ?case by (fastforce simp: nat-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*:

```

 $\llbracket prog \vdash (-l-) -et \rightarrow (-l'-); \text{transfer } et \ s = s'; \text{pred } et \ s \rrbracket$ 
 $\implies \exists c \ c'. \text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \wedge \text{labels } prog \ l \ c \wedge \text{labels } prog \ l' \ c'$ 
proof (induct prog  $(-l-)$   $et$   $(-l'-)$  arbitrary:  $l \ l'$  rule: WCFG-induct)
  case (WCFG-LAss  $V \ e$ )
  from  $\langle \text{transfer } \uparrow \lambda s. s(V := (\text{interpret } e \ s)) \ s = s' \rangle$ 
  have  $[simp]: s' = s(V := (\text{interpret } e \ s))$  by (simp del: fun-upd-apply)
  have labels  $(V := e) \ 0 \ (V := e)$  by (fastforce intro: Labels-Base)
  moreover
  hence labels  $(V := e) \ 1 \text{ Skip}$  by (fastforce intro: Labels-LAss)
  ultimately show ?case
    apply (rule-tac  $x = V := e$  in  $exI$ )
    apply (rule-tac  $x = \text{Skip}$  in  $exI$ )
    by (fastforce intro: StepLAss simp del: fun-upd-apply)
next
  case (WCFG-SeqFirst  $c_1$   $et$   $c_2$ )
  note  $IH = \langle \llbracket \text{transfer } et \ s = s'; \text{pred } et \ s \rrbracket \rangle$ 
     $\implies \exists c \ c'. c_1 \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \wedge \text{labels } c_1 \ l \ c \wedge \text{labels } c_1 \ l' \ c'$ 
  from  $IH[OF \langle \text{transfer } et \ s = s' \rangle \langle \text{pred } et \ s \rangle]$ 
  obtain  $c \ c'$  where  $c_1 \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle$ 
    and labels  $c_1 \ l \ c$  and labels  $c_1 \ l' \ c'$  by blast
  from  $\langle c_1 \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \rangle$  have  $c_1;; c_2 \vdash \langle c;; c_2, s, l \rangle \rightsquigarrow \langle c';; c_2, s', l' \rangle$ 

```

```

    by(rule StepRecSeq1)
  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 ⟨ $c_1 \vdash (- l -) \multimap et \rightarrow (- Exit -)$ ⟩
  have labels  $c_1$   $l$  Skip and  $[simp]:et = \uparrow id$ 
    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  $c_1;;c_2 \vdash \langle Skip;;c_2,s,l \rangle \rightsquigarrow \langle c_2,s,\#:$  $c_1 \rangle$  by  $\neg$ (rule StepSeq)
  with ⟨labels  $(c_1;;c_2)$   $l$  (Skip;; $c_2$ )⟩
    ⟨labels  $(c_1;;c_2)$   $\#:$  $c_1$   $c_2$ ⟩  $\langle (-0-) \oplus \#:$  $c_1 = (- l' -) \rangle$  show ?case by simp blast
next
case (WCFG-SeqSecond  $c_2$   $n$   $et$   $n'$   $c_1$ )
  note  $IH = \langle \bigwedge l l'. \llbracket n = (- l -); n' = (- l' -); transfer\ et\ s = s'; pred\ et\ s \rrbracket$ 
     $\implies \exists c c'. c_2 \vdash \langle c,s,l \rangle \rightsquigarrow \langle c',s',l' \rangle \wedge labels\ c_2\ l\ c \wedge labels\ c_2\ l'\ c'$ 
  from  $\langle n \oplus \#:$  $c_1 = (- l -) \rangle$  obtain  $lx$  where  $n = (- lx -)$ 
    and  $[simp]:l = lx + \#:$  $c_1$ 
    by(cases  $n$ ) auto
  from  $\langle n' \oplus \#:$  $c_1 = (- l' -) \rangle$  obtain  $lx'$  where  $n' = (- lx' -)$ 
    and  $[simp]:l' = lx' + \#:$  $c_1$ 
    by(cases  $n'$ ) auto
  from  $IH[OF \langle n = (- lx -) \rangle \langle n' = (- lx' -) \rangle \langle transfer\ et\ s = s' \rangle \langle pred\ et\ s \rangle]$ 
  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  $\langle c_2 \vdash \langle c,s,lx \rangle \rightsquigarrow \langle c',s',lx' \rangle \rangle$  have  $c_1;;c_2 \vdash \langle c,s,l \rangle \rightsquigarrow \langle c',s',l' \rangle$ 
    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  $\langle (-0-) \oplus 1 = (- l' -) \rangle$  have  $[simp]:l' = 1$  by simp
  from ⟨transfer  $(\lambda s. interpret\ b\ s = Some\ true) \surd s = s' \rangle$  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 $\langle \text{pred } (\lambda s. \text{interpret } b \ s = \text{Some true})_{\surd} \ s \rangle$
have interpret $b \ s = \text{Some true}$ **by** simp
hence if (b) c_1 else $c_2 \vdash \langle \text{if } (b) \ c_1 \text{ else } c_2, s, 0 \rangle \rightsquigarrow \langle c_1, s, 1 \rangle$
by(rule StepCondTrue)
with $\langle \text{labels } (\text{if } (b) \ c_1 \text{ else } c_2) \ 0 \ (\text{if } (b) \ c_1 \text{ else } c_2) \rangle$
 $\langle \text{labels } (\text{if } (b) \ c_1 \text{ else } c_2) \ 1 \ c_1 \rangle$ **show** ?case **by** simp blast
next
case (WCFG-CondFalse b c_1 c_2)
from $\langle (-0-) \oplus \# : c_1 + 1 = (-l' -) \rangle$ **have** [simp]: $l' = \# : c_1 + 1$ **by** simp
from $\langle \text{transfer } (\lambda s. \text{interpret } b \ s = \text{Some false})_{\surd} \ s = s' \rangle$ **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 $\langle \text{pred } (\lambda s. \text{interpret } b \ s = \text{Some false})_{\surd} \ s \rangle$
have interpret $b \ s = \text{Some false}$ **by** simp
hence if (b) c_1 else $c_2 \vdash \langle \text{if } (b) \ c_1 \text{ else } c_2, s, 0 \rangle \rightsquigarrow \langle c_2, s, \# : c_1 + 1 \rangle$
by(rule StepCondFalse)
with $\langle \text{labels } (\text{if } (b) \ c_1 \text{ else } c_2) \ 0 \ (\text{if } (b) \ c_1 \text{ else } c_2) \rangle$
 $\langle \text{labels } (\text{if } (b) \ c_1 \text{ else } c_2) \ (\# : c_1 + 1) \ c_2 \rangle$ **show** ?case **by** simp blast
next
case (WCFG-CondThen c_1 n et n' b c_2)
note IH = $\langle \bigwedge l \ l'. \llbracket n = (-l -); n' = (-l' -); \text{transfer et } s = s'; \text{pred et } s \rrbracket$
 $\implies \exists c \ c'. \ c_1 \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \wedge \text{labels } c_1 \ l \ c \wedge \text{labels } c_1 \ l' \ c' \rangle$
from $\langle n \oplus 1 = (-l -) \rangle$ **obtain** lx **where** $n = (-lx -)$ **and** [simp]: $l = lx + 1$
by(cases n) auto
from $\langle n' \oplus 1 = (-l' -) \rangle$ **obtain** lx' **where** $n' = (-lx' -)$ **and** [simp]: $l' = lx' + 1$
by(cases n') auto
from IH[OF $\langle n = (-lx -) \rangle \langle n' = (-lx' -) \rangle \langle \text{transfer et } s = s' \rangle \langle \text{pred et } s \rangle$]
obtain $c \ c'$ **where** $c_1 \vdash \langle c, s, lx \rangle \rightsquigarrow \langle c', s', lx' \rangle$
and labels $c_1 \ lx \ c$ **and** labels $c_1 \ lx' \ c'$ **by** blast
from $\langle c_1 \vdash \langle c, s, lx \rangle \rightsquigarrow \langle c', s', lx' \rangle \rangle$ **have** if (b) c_1 else $c_2 \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle$
by(fastforce intro:StepRecCond1)
moreover
from $\langle \text{labels } c_1 \ lx \ c \rangle$ **have** labels (if (b) c_1 else c_2) $l \ c$
by(fastforce intro:Labels-CondTrue)
moreover
from $\langle \text{labels } c_1 \ lx' \ c' \rangle$ **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 = $\langle \bigwedge l \ l'. \llbracket n = (-l -); n' = (-l' -); \text{transfer et } s = s'; \text{pred et } s \rrbracket$

```

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

```



```

note  $IH = \langle \bigwedge l l'. \llbracket n = (- l -); n' = (- l' -); \text{transfer et } s = s'; \text{pred et } s \rrbracket$ 
 $\implies \exists c c'. cx \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \wedge \text{labels } cx \ l \ c \wedge \text{labels } cx \ l' \ c'$ 
from  $\langle n \oplus 2 = (- l -) \rangle$  obtain  $lx$  where  $n = (- lx -)$  and  $[simp]: l = lx + 2$ 
by(cases  $n$ ) auto
from  $\langle n' \oplus 2 = (- l' -) \rangle$  obtain  $lx'$  where  $n' = (- lx' -)$ 
and  $[simp]: l' = lx' + 2$  by(cases  $n'$ ) auto
from  $IH[OF \langle n = (- lx -) \rangle \langle n' = (- lx' -) \rangle \langle \text{transfer et } s = s' \rangle \langle \text{pred et } s \rangle]$ 
obtain  $c \ c'$  where  $cx \vdash \langle c, s, lx \rangle \rightsquigarrow \langle c', s', lx' \rangle$ 
and  $\text{labels } cx \ lx \ c$  and  $\text{labels } cx \ lx' \ c'$  by blast
hence  $\text{while } (b) \ cx \vdash \langle c;; \text{while } (b) \ cx, s, l \rangle \rightsquigarrow \langle c';; \text{while } (b) \ cx, s', l' \rangle$ 
by(fastforce intro:StepRecWhile)
moreover
from  $\langle \text{labels } cx \ lx \ c \rangle$  have  $\text{labels } (\text{while } (b) \ cx) \ l \ (c;; \text{while } (b) \ cx)$ 
by(fastforce intro:Labels-WhileBody)
moreover
from  $\langle \text{labels } cx \ lx' \ c' \rangle$  have  $\text{labels } (\text{while } (b) \ cx) \ l' \ (c';; \text{while } (b) \ cx)$ 
by(fastforce intro:Labels-WhileBody)
ultimately show ?case by blast
next
case (WCFG-WhileBodyExit  $cx \ n \ \text{et } b$ )
from  $\langle n \oplus 2 = (- l -) \rangle$  obtain  $lx$  where  $[simp]: n = (- lx -)$  and  $[simp]: l = lx + 2$ 
by(cases  $n$ ) auto
from  $\langle cx \vdash n - \text{et} \rightarrow (- \text{Exit} -) \rangle$  have  $\text{labels } cx \ lx \ \text{Skip}$  and  $[simp]: \text{et} = \uparrow id$ 
by(auto dest:WCFG-edge-Exit-Skip)
from  $\langle \text{transfer et } s = s' \rangle$  have  $[simp]: s' = s$  by simp
from  $\langle \text{labels } cx \ lx \ \text{Skip} \rangle$  have  $\text{labels } (\text{while } (b) \ cx) \ l \ (\text{Skip};; \text{while } (b) \ cx)$ 
by(fastforce intro:Labels-WhileBody)
hence  $\text{while } (b) \ cx \vdash \langle \text{Skip};; \text{while } (b) \ cx, s, l \rangle \rightsquigarrow \langle \text{while } (b) \ cx, s, 0 \rangle$ 
by(rule StepSeqWhile)
moreover
have  $\text{labels } (\text{while } (b) \ cx) \ 0 \ (\text{while } (b) \ cx)$ 
by(fastforce intro:Labels-Base)
ultimately show ?case
using  $\langle \text{labels } (\text{while } (b) \ cx) \ l \ (\text{Skip};; \text{while } (b) \ cx) \rangle$  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* $\Rightarrow$ *w-node* $\Rightarrow$ *cmd* $\Rightarrow$ *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** $\langle c, s \rangle \rightarrow^* \langle c', s' \rangle$

then obtain *l l'* **where** [*simp*]: *n* = (- *l* -) **and** *prog* $\vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle$

and *labels* *prog* *l' c'* **by**(*cases* *n*, *auto* *dest:reds-steps*)

from $\langle \text{labels } \text{prog } l' c' \rangle$ **have** *l' < #:prog* **by**(*rule* *label-less-num-inner-nodes*)

from $\langle \text{prog } \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle \rangle$

have $\exists \text{as. } \text{CFG.path } \text{sourcenode } \text{targetnode } (\text{valid-edge } \text{prog})$

(- *l* -) *as* (- *l'* -) $\wedge$

transfers (*CFG.kinds* *kind* *as*) *s* = *s'* $\wedge$ *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'* -) $\square$ (- *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 $\langle \lambda(c, s, l) (c', s', l') \rangle$

prog $\vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle$ (*c, s, l*) (*c'', s'', l''*)

have *prog* $\vdash \langle c, s, l \rangle \rightsquigarrow \langle c'', s'', l'' \rangle$ **by** *simp*

from $\langle \exists \text{as. } \text{CFG.path } \text{sourcenode } \text{targetnode } (\text{valid-edge } \text{prog})$

(- *l''* -) *as* (- *l'* -) $\wedge$

transfers (*CFG.kinds* *kind* *as*) *s''* = *s'* $\wedge$

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 $\langle \text{prog } \vdash \langle c, s, l \rangle \rightsquigarrow \langle c'', s'', l'' \rangle \rangle$ **obtain** *et*

where *prog* $\vdash (- l -) \rightarrow \text{et} \rightarrow (- l'' -)$

and *transfer* *et* *s* = *s''* **and** *pred* *et* *s*

by(*erule* *step-WCFG-edge*)

from $\langle \text{prog } \vdash (- l -) \rightarrow \text{et} \rightarrow (- l'' -) \rangle$

$\langle \text{CFG.path } \text{sourcenode } \text{targetnode } (\text{valid-edge } \text{prog}) (- l'' -) \text{ as } (- l' -) \rangle$

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 $\exists as. \text{prog} \vdash (-\text{Entry-}) -as \rightarrow^* n$ **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 $\exists as. \text{prog} \vdash n -as \rightarrow^* (-\text{Exit-})$ **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

4.12 Slicing guarantees IFC Noninterference

```
theory NonInterferenceIntra imports
  ../Slicing/StaticIntra/Slice
  ../Slicing/Basic/CFGExit-wf
begin
```

4.12.1 Assumptions of this Approach

Classical IFC noninterference, a special case of a noninterference definition using partial equivalence relations (per) [?], partitions the variables (i.e. locations) into security levels. Usually, only levels for secret or high, written H , and public or low, written L , variables are used. Basically, a program that is noninterferent has to fulfil one basic property: executing the program in two different initial states that may differ in the values of their H -variables yields two final states that again only differ in the values of their H -variables; thus the values of the H -variables did not influence those of the L -variables.

Every per-based approach makes certain assumptions: (i) all H -variables are defined at the beginning of the program, (ii) all L -variables are observed (or used in our terms) at the end and (iii) every variable is either H or L . This security label is fixed for a variable and can not be altered during a program run. Thus, we have to extend the prerequisites of the slicing framework in [?] accordingly in a new locale:

```
locale NonInterferenceIntraGraph =
  BackwardSlice sourcenode targetnode kind valid-edge Entry Def Use state-val
  backward-slice +
  CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for sourcenode :: 'edge  $\Rightarrow$  'node and targetnode :: 'edge  $\Rightarrow$  'node
and kind :: 'edge  $\Rightarrow$  'state edge-kind and valid-edge :: 'edge  $\Rightarrow$  bool
and Entry :: 'node ('('Entry'-')) and Def :: 'node  $\Rightarrow$  'var set
and Use :: 'node  $\Rightarrow$  'var set and state-val :: 'state  $\Rightarrow$  'var  $\Rightarrow$  'val
and backward-slice :: 'node set  $\Rightarrow$  'node set
and Exit :: 'node ('('Exit'-')) +
fixes H :: 'var set
fixes L :: 'var set
fixes High :: 'node ('('High'-'))
fixes Low :: 'node ('('Low'-'))
assumes Entry-edge-Exit-or-High:
   $\llbracket \text{valid-edge } a; \text{sourcenode } a = (-\text{Entry-}) \rrbracket$ 
   $\implies \text{targetnode } a = (-\text{Exit-}) \vee \text{targetnode } a = (-\text{High-})$ 
and High-target-Entry-edge:
   $\exists a. \text{valid-edge } a \wedge \text{sourcenode } a = (-\text{Entry-}) \wedge \text{targetnode } a = (-\text{High-}) \wedge$ 
   $\text{kind } a = (\lambda s. \text{True})_{\vee}$ 
and Entry-predecessor-of-High:
```

$\llbracket \text{valid-edge } a; \text{targetnode } a = (-\text{High-}) \rrbracket \implies \text{sourcenode } a = (-\text{Entry-})$
and *Exit-edge-Entry-or-Low*: $\llbracket \text{valid-edge } a; \text{targetnode } a = (-\text{Exit-}) \rrbracket$
 $\implies \text{sourcenode } a = (-\text{Entry-}) \vee \text{sourcenode } a = (-\text{Low-})$
and *Low-source-Exit-edge*:
 $\exists a. \text{valid-edge } a \wedge \text{sourcenode } a = (-\text{Low-}) \wedge \text{targetnode } a = (-\text{Exit-}) \wedge$
 $\text{kind } a = (\lambda s. \text{True})_{\checkmark}$
and *Exit-successor-of-Low*:
 $\llbracket \text{valid-edge } a; \text{sourcenode } a = (-\text{Low-}) \rrbracket \implies \text{targetnode } a = (-\text{Exit-})$
and *DefHigh*: $\text{Def } (-\text{High-}) = H$
and *UseHigh*: $\text{Use } (-\text{High-}) = H$
and *UseLow*: $\text{Use } (-\text{Low-}) = L$
and *HighLowDistinct*: $H \cap L = \{\}$
and *HighLowUNIV*: $H \cup L = \text{UNIV}$

begin

lemma *Low-neq-Exit*: **assumes** $L \neq \{\}$ **shows** $(-\text{Low-}) \neq (-\text{Exit-})$

proof

assume $(-\text{Low-}) = (-\text{Exit-})$
have $\text{Use } (-\text{Exit-}) = \{\}$ **by** *fastforce*
with $\text{UseLow } \langle L \neq \{\} \rangle \langle (-\text{Low-}) = (-\text{Exit-}) \rangle$ **show** *False* **by** *simp*
qed

lemma *Entry-path-High-path*:

assumes $(-\text{Entry-}) -as \rightarrow^* n$ **and** *inner-node* n
obtains $a' \text{ as}'$ **where** $as = a' \# as'$ **and** $(-\text{High-}) -as' \rightarrow^* n$
and $\text{kind } a' = (\lambda s. \text{True})_{\checkmark}$
proof(*atomize-elim*)
from $\langle (-\text{Entry-}) -as \rightarrow^* n \rangle \langle \text{inner-node } n \rangle$
show $\exists a' \text{ as}'. as = a' \# as' \wedge (-\text{High-}) -as' \rightarrow^* n \wedge \text{kind } a' = (\lambda s. \text{True})_{\checkmark}$
proof(*induct* $n' \equiv (-\text{Entry-}) \text{ as } n$ *rule: path.induct*)
case (*Cons-path* $n'' \text{ as } n' a$)
from $\langle n'' -as \rightarrow^* n' \rangle \langle \text{inner-node } n' \rangle$ **have** $n'' \neq (-\text{Exit-})$
by(*fastforce simp: inner-node-def*)
with $\langle \text{valid-edge } a \rangle \langle \text{targetnode } a = n' \rangle \langle \text{sourcenode } a = (-\text{Entry-}) \rangle$
have $n'' = (-\text{High-})$ **by** $-(\text{drule } \text{Entry-edge-Exit-or-High, auto})$
from *High-target-Entry-edge*
obtain a' **where** *valid-edge* a' **and** *sourcenode* $a' = (-\text{Entry-})$
and *targetnode* $a' = (-\text{High-})$ **and** $\text{kind } a' = (\lambda s. \text{True})_{\checkmark}$
by *blast*
with $\langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = (-\text{Entry-}) \rangle \langle \text{targetnode } a = n'' \rangle$
 $\langle n'' = (-\text{High-}) \rangle$
have $a = a'$ **by**(*auto dest: edge-det*)
with $\langle n'' -as \rightarrow^* n' \rangle \langle n'' = (-\text{High-}) \rangle \langle \text{kind } a' = (\lambda s. \text{True})_{\checkmark} \rangle$ **show** *?case* **by**
blast
qed *fastforce*
qed

```

lemma Exit-path-Low-path:
  assumes  $n - as \rightarrow^* (-Exit-)$  and inner-node  $n$ 
  obtains  $a' as'$  where  $as = as'@[a]$  and  $n - as' \rightarrow^* (-Low-)$ 
  and  $kind\ a' = (\lambda s. True)_\checkmark$ 
proof(atomize-elim)
  from  $\langle n - as \rightarrow^* (-Exit-) \rangle$ 
  show  $\exists as' a'. as = as'@[a] \wedge n - as' \rightarrow^* (-Low-) \wedge kind\ a' = (\lambda s. True)_\checkmark$ 
  proof(induct as rule:rev-induct)
    case Nil
    with  $\langle inner-node\ n \rangle$  show ?case by fastforce
  next
    case (snoc  $a' as'$ )
    from  $\langle n - as'@[a] \rightarrow^* (-Exit-) \rangle$ 
    have  $n - as' \rightarrow^* sourcenode\ a'$  and valid-edge  $a'$  and targetnode  $a' = (-Exit-)$ 
    by(auto elim:path-split-snoc)
    { assume sourcenode  $a' = (-Entry-)$ 
      with  $\langle n - as' \rightarrow^* sourcenode\ a' \rangle$  have  $n = (-Entry-)$ 
      by(blast intro!:path-Entry-target)
      with  $\langle inner-node\ n \rangle$  have False by(simp add:inner-node-def) }
    with  $\langle valid-edge\ a' \rangle \langle targetnode\ a' = (-Exit-) \rangle$  have sourcenode  $a' = (-Low-)$ 
    by(blast dest!:Exit-edge-Entry-or-Low)
    from Low-source-Exit-edge
    obtain  $ax$  where valid-edge  $ax$  and sourcenode  $ax = (-Low-)$ 
    and targetnode  $ax = (-Exit-)$  and  $kind\ ax = (\lambda s. True)_\checkmark$ 
    by blast
    with  $\langle valid-edge\ a' \rangle \langle targetnode\ a' = (-Exit-) \rangle \langle sourcenode\ a' = (-Low-) \rangle$ 
    have  $a' = ax$  by(fastforce intro:edge-det)
    with  $\langle n - as' \rightarrow^* sourcenode\ a' \rangle \langle sourcenode\ a' = (-Low-) \rangle \langle kind\ ax = (\lambda s. True)_\checkmark \rangle$ 
    show ?case by blast
  qed
qed

```

```

lemma not-Low-High:  $V \notin L \implies V \in H$ 
  using HighLowUNIV
  by fastforce

```

```

lemma not-High-Low:  $V \notin H \implies V \in L$ 
  using HighLowUNIV
  by fastforce

```

4.12.2 Low Equivalence

In classical noninterference, an external observer can only see public values, in our case the L -variables. If two states agree in the values of all L -variables, these states are indistinguishable for him. *Low equivalence* groups those states in an equivalence class using the relation $\approx_L$:

definition *lowEquivalence* :: 'state $\Rightarrow$ 'state $\Rightarrow$ bool (**infixl** $\approx_L$ 50)
where $s \approx_L s' \equiv \forall V \in L. \text{state-val } s \ V = \text{state-val } s' \ V$

The following lemmas connect low equivalent states with relevant variables as necessary in the correctness proof for slicing.

lemma *relevant-vars-Entry*:

assumes $V \in \text{rv } S$ **(-Entry-)** **and** $(\text{-High-}) \notin \text{backward-slice } S$
shows $V \in L$

proof –

from $\langle V \in \text{rv } S \text{ (-Entry-)} \rangle$ **obtain** $as \ n'$ **where** $(\text{-Entry-}) -as \rightarrow^* n'$
and $n' \in \text{backward-slice } S$ **and** $V \in \text{Use } n'$
and $\forall nx \in \text{set}(\text{sourcenodes } as). V \notin \text{Def } nx$ **by** (erule rvE)
from $\langle (\text{-Entry-}) -as \rightarrow^* n' \rangle$ **have** *valid-node* n' **by** (rule path-valid-node)
thus ?thesis

proof (cases n' rule:valid-node-cases)

case *Entry*

with $\langle V \in \text{Use } n' \rangle$ **have** *False* **by** (simp add:Entry-empty)
thus ?thesis **by** simp

next

case *Exit*

with $\langle V \in \text{Use } n' \rangle$ **have** *False* **by** (simp add:Exit-empty)
thus ?thesis **by** simp

next

case *inner*

with $\langle (\text{-Entry-}) -as \rightarrow^* n' \rangle$ **obtain** $a' \ as'$ **where** $as = a' \# as'$
and $(\text{-High-}) -as' \rightarrow^* n'$ **by** $\neg$ (erule Entry-path-High-path)
from $\langle (\text{-Entry-}) -as \rightarrow^* n' \rangle$ $\langle as = a' \# as' \rangle$
have *sourcenode* $a' = (\text{-Entry-})$ **by** (fastforce elim:path.cases)
show ?thesis

proof (cases $as' = []$)

case *True*

with $\langle (\text{-High-}) -as' \rightarrow^* n' \rangle$ **have** $n' = (\text{-High-})$ **by** fastforce
with $\langle n' \in \text{backward-slice } S \rangle$ $\langle (\text{-High-}) \notin \text{backward-slice } S \rangle$
have *False* **by** simp
thus ?thesis **by** simp

next

case *False*

with $\langle (\text{-High-}) -as' \rightarrow^* n' \rangle$ **have** $\text{hd } (\text{sourcenodes } as') = (\text{-High-})$
by (rule path-sourcenode)

from *False* **have** $\text{hd } (\text{sourcenodes } as') \in \text{set } (\text{sourcenodes } as')$
by (fastforce intro:hd-in-set simp:sourcenodes-def)

with $\langle as = a' \# as' \rangle$ **have** $\text{hd } (\text{sourcenodes } as') \in \text{set } (\text{sourcenodes } as)$
by (simp add:sourcenodes-def)

with $\langle \text{hd } (\text{sourcenodes } as') = (\text{-High-}) \rangle$ $\langle \forall nx \in \text{set}(\text{sourcenodes } as). V \notin \text{Def}$

$nx \rangle$

have $V \notin \text{Def } (\text{-High-})$ **by** fastforce
hence $V \notin H$ **by** (simp add:DefHigh)
thus ?thesis **by** (rule not-High-Low)

qed

qed
qed

lemma *lowEquivalence-relevant-nodes-Entry*:
assumes $s \approx_L s'$ **and** $(-High-) \notin \text{backward-slice } S$
shows $\forall V \in \text{rv } S \ (-Entry-). \text{state-val } s \ V = \text{state-val } s' \ V$
proof
fix V **assume** $V \in \text{rv } S \ (-Entry-)$
with $\langle(-High-) \notin \text{backward-slice } S\rangle$ **have** $V \in L$ **by** $-(\text{rule relevant-vars-Entry})$
with $\langle s \approx_L s' \rangle$ **show** $\text{state-val } s \ V = \text{state-val } s' \ V$ **by** $(\text{simp add:lowEquivalence-def})$
qed

lemma *rv-Low-Use-Low*:
assumes $(-Low-) \in S$
shows $\llbracket n -as \rightarrow^* (-Low-); n -as' \rightarrow^* (-Low-);$
 $\forall V \in \text{rv } S \ n. \text{state-val } s \ V = \text{state-val } s' \ V;$
 $\text{preds } (\text{slice-kinds } S \ as) \ s; \text{preds } (\text{slice-kinds } S \ as') \ s \rrbracket$
 $\implies \forall V \in \text{Use } (-Low-). \text{state-val } (\text{transfers } (\text{slice-kinds } S \ as) \ s) \ V =$
 $\text{state-val } (\text{transfers } (\text{slice-kinds } S \ as') \ s') \ V$
proof $(\text{induct } n \text{ as } n \equiv (-Low-) \text{ arbitrary:as' } s \ s' \text{ rule:path.induct})$
case *empty-path*
{ fix V **assume** $V \in \text{Use } (-Low-)$
moreover
from $\langle \text{valid-node } (-Low-) \rangle$ **have** $(-Low-) -[] \rightarrow^* (-Low-)$
by $(\text{fastforce intro:path.empty-path})$
moreover
from $\langle \text{valid-node } (-Low-) \rangle \langle (-Low-) \in S \rangle$ **have** $(-Low-) \in \text{backward-slice } S$
by $(\text{fastforce intro:refl})$
ultimately have $V \in \text{rv } S \ (-Low-)$
by $(\text{fastforce intro:rvI simp:sourcenodes-def})$ **}**
hence $\forall V \in \text{Use } (-Low-). V \in \text{rv } S \ (-Low-)$ **by** *simp*
show *?case*
proof $(\text{cases } L = \{\})$
case *True* **with** *UseLow* **show** *?thesis* **by** *simp*
next
case *False*
from $\langle (-Low-) -as' \rightarrow^* (-Low-) \rangle$ **have** $as' = []$
proof $(\text{induct } n \equiv (-Low-) \text{ as' } n' \equiv (-Low-) \text{ rule:path.induct})$
case $(\text{Cons-path } n'' \text{ as } a)$
from $\langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = (-Low-) \rangle$
have $\text{targetnode } a = (-Exit-) \text{ by } -(\text{rule Exit-successor-of-Low,simp+})$
with $\langle \text{targetnode } a = n'' \rangle \langle n'' -as \rightarrow^* (-Low-) \rangle$
have $(-Low-) = (-Exit-) \text{ by } -(\text{rule path-Exit-source,fastforce})$
with *False* **have** *False* **by** $-(\text{drule Low-neq-Exit,simp})$
thus *?case* **by** *simp*

```

qed simp
with ⟨ $\forall V \in Use \text{ (-Low-)}. V \in rv S \text{ (-Low-)}$ ⟩
  ⟨ $\forall V \in rv S \text{ (-Low-)}. state-val s V = state-val s' V$ ⟩
show ?thesis by(auto simp:slice-kinds-def)
qed
next
case (Cons-path n'' as a n)
note IH = ⟨ $\bigwedge as' s s'. \llbracket n'' - as' \rightarrow * \text{ (-Low-)}$ ⟩;
   $\forall V \in rv S n''. state-val s V = state-val s' V$ ;
  preds (slice-kinds S as) s; preds (slice-kinds S as') s'⟩
 $\Rightarrow \forall V \in Use \text{ (-Low-)}. state-val (transfers (slice-kinds S as) s) V =$ 
   $state-val (transfers (slice-kinds S as') s') V$ 
show ?case
proof(cases L = {})
  case True with UseLow show ?thesis by simp
next
  case False
  show ?thesis
  proof(cases as')
    case Nil
    with ⟨ $n - as' \rightarrow * \text{ (-Low-)}$ ⟩ have n = (-Low-) by fastforce
    with ⟨valid-edge a⟩ ⟨sourcenode a = n⟩ have targetnode a = (-Exit-)
      by -(rule Exit-successor-of-Low,simp+)
    from Low-source-Exit-edge obtain ax where valid-edge ax
      and sourcenode ax = (-Low-) and targetnode ax = (-Exit-)
      and kind ax = ( $\lambda s. True$ )✓ by blast
    from ⟨valid-edge a⟩ ⟨sourcenode a = n⟩ ⟨n = (-Low-)⟩ ⟨targetnode a = (-Exit-)⟩
      ⟨valid-edge ax⟩ ⟨sourcenode ax = (-Low-)⟩ ⟨targetnode ax = (-Exit-)⟩
    have a = ax by(fastforce dest:edge-det)
    with ⟨kind ax = ( $\lambda s. True$ )✓⟩ have kind a = ( $\lambda s. True$ )✓ by simp
    with ⟨targetnode a = (-Exit-)⟩ ⟨targetnode a = n'⟩ ⟨ $n'' - as \rightarrow * \text{ (-Low-)}$ ⟩
    have (-Low-) = (-Exit-) by -(rule path-Exit-source,auto)
    with False have False by -(drule Low-neq-Exit,simp)
    thus ?thesis by simp
  next
    case (Cons ax asx)
    with ⟨ $n - as' \rightarrow * \text{ (-Low-)}$ ⟩ have n = sourcenode ax and valid-edge ax
      and targetnode ax - asx  $\rightarrow * \text{ (-Low-)}$  by(auto elim:path-split-Cons)
    show ?thesis
    proof(cases targetnode ax = n'')
      case True
      with ⟨targetnode ax - asx  $\rightarrow * \text{ (-Low-)}$ ⟩ have n'' - asx  $\rightarrow * \text{ (-Low-)}$  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 ⟨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→* (-Low-)⟩ this preds1 preds2]
    Cons ⟨ax = a⟩ show ?thesis by(simp add:slice-kinds-def)
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 -(rule rv-branching-edges-slice-kinds-False,auto)
thus ?thesis by simp
qed
qed
qed
qed

```

4.12.3 The Correctness Proofs

In the following, we present two correctness proofs that slicing guarantees IFC noninterference. In both theorems, $(-High-) \notin \text{backward-slice } S$, where $(-Low-) \in S$, makes sure that no high variable (which are all defined in $(-High-)$) can influence a low variable (which are all used in $(-Low-)$).

First, a theorem regarding $(-Entry-) -as \rightarrow^* (-Exit-)$ paths in the control flow graph (CFG), which agree to a complete program execution:

lemma *nonInterference-path-to-Low*:

assumes $s \approx_L s'$ **and** $(-High-) \notin \text{backward-slice } S$ **and** $(-Low-) \in S$
and $(-Entry-) -as \rightarrow^* (-Low-)$ **and** $\text{preds (kinds as) } s$
and $(-Entry-) -as' \rightarrow^* (-Low-)$ **and** $\text{preds (kinds as') } s'$
shows $\text{transfers (kinds as) } s \approx_L \text{transfers (kinds as') } s'$

proof –

from $\langle(-Entry-) -as \rightarrow^* (-Low-)\rangle \langle\text{preds (kinds as) } s\rangle \langle(-Low-) \in S\rangle$
obtain asx **where** $\text{preds (slice-kinds } S \text{ asx) } s$
and $\forall V \in \text{Use } (-Low-). \text{state-val}(\text{transfers (slice-kinds } S \text{ asx) } s) V =$
 $\text{state-val}(\text{transfers (kinds as) } s) V$
and $\text{slice-edges } S \text{ as} = \text{slice-edges } S \text{ asx}$
and $(-Entry-) -asx \rightarrow^* (-Low-)$ **by** (rule *fundamental-property-of-static-slicing*)
from $\langle(-Entry-) -as' \rightarrow^* (-Low-)\rangle \langle\text{preds (kinds as') } s'\rangle \langle(-Low-) \in S\rangle$
obtain asx' **where** $\text{preds (slice-kinds } S \text{ asx') } s'$
and $\forall V \in \text{Use } (-Low-). \text{state-val}(\text{transfers (slice-kinds } S \text{ asx') } s') V =$
 $\text{state-val}(\text{transfers (kinds as') } s') V$
and $\text{slice-edges } S \text{ as'} = \text{slice-edges } S \text{ asx'}$

and $\langle (-\text{Entry-}) - \text{asx}' \rightarrow * (-\text{Low-}) \rangle$ **by** (erule fundamental-property-of-static-slicing)
from $\langle s \approx_L s' \rangle \langle (-\text{High-}) \notin \text{backward-slice } S \rangle$
have $\forall V \in \text{rv } S \langle (-\text{Entry-}). \text{state-val } s \ V = \text{state-val } s' \ V$
by (rule lowEquivalence-relevant-nodes-Entry)
with $\langle (-\text{Entry-}) - \text{asx} \rightarrow * (-\text{Low-}) \rangle \langle (-\text{Entry-}) - \text{asx}' \rightarrow * (-\text{Low-}) \rangle \langle (-\text{Low-}) \in S \rangle$
 $\langle \text{preds } (\text{slice-kinds } S \ \text{asx}) \ s \rangle \langle \text{preds } (\text{slice-kinds } S \ \text{asx}') \ s' \rangle$
have $\forall V \in \text{Use } (-\text{Low-}). \text{state-val } (\text{transfers } (\text{slice-kinds } S \ \text{asx}) \ s) \ V =$
 $\text{state-val } (\text{transfers } (\text{slice-kinds } S \ \text{asx}') \ s') \ V$
by $\neg(\text{rule rv-Low-Use-Low, auto})$
with $\langle \forall V \in \text{Use } (-\text{Low-}). \text{state-val } (\text{transfers } (\text{slice-kinds } S \ \text{asx}) \ s) \ V =$
 $\text{state-val } (\text{transfers } (\text{kinds } \text{as}) \ s) \ V \rangle$
 $\langle \forall V \in \text{Use } (-\text{Low-}). \text{state-val } (\text{transfers } (\text{slice-kinds } S \ \text{asx}') \ s') \ V =$
 $\text{state-val } (\text{transfers } (\text{kinds } \text{as}') \ s') \ V \rangle$
show ?thesis **by** (auto simp: lowEquivalence-def UseLow)
qed

theorem nonInterference-path:

assumes $s \approx_L s'$ **and** $\langle (-\text{High-}) \notin \text{backward-slice } S \rangle$ **and** $\langle (-\text{Low-}) \in S \rangle$
and $\langle (-\text{Entry-}) - \text{as} \rightarrow * (-\text{Exit-}) \rangle$ **and** $\langle \text{preds } (\text{kinds } \text{as}) \ s \rangle$
and $\langle (-\text{Entry-}) - \text{as}' \rightarrow * (-\text{Exit-}) \rangle$ **and** $\langle \text{preds } (\text{kinds } \text{as}') \ s' \rangle$
shows $\text{transfers } (\text{kinds } \text{as}) \ s \approx_L \text{transfers } (\text{kinds } \text{as}') \ s'$
proof –
from $\langle (-\text{Entry-}) - \text{as} \rightarrow * (-\text{Exit-}) \rangle$ **obtain** $x \ xs$ **where** $\text{as} = x \# xs$
and $\langle (-\text{Entry-}) = \text{sourcenode } x \ \text{and } \text{valid-edge } x$
and $\text{targetnode } x - \text{xs} \rightarrow * (-\text{Exit-}) \rangle$
apply (cases $\text{as} = []$)
apply (simp, drule empty-path-nodes, drule Entry-noteq-Exit, simp)
by (erule path-split-Cons)
from $\langle \text{valid-edge } x \rangle$ **have** $\text{valid-node } (\text{targetnode } x)$ **by** simp
hence $\text{inner-node } (\text{targetnode } x)$
proof (cases rule: valid-node-cases)
case Entry
with $\langle \text{valid-edge } x \rangle$ **have** False **by** (rule Entry-target)
thus ?thesis **by** simp
next
case Exit
with $\langle \text{targetnode } x - \text{xs} \rightarrow * (-\text{Exit-}) \rangle$ **have** $xs = []$
by $\neg(\text{rule path-Exit-source, simp})$
from Entry-Exit-edge **obtain** z **where** $\text{valid-edge } z$
and $\text{sourcenode } z = (-\text{Entry-})$ **and** $\text{targetnode } z = (-\text{Exit-})$
and $\text{kind } z = (\lambda s. \text{False})_{\checkmark}$ **by** blast
from $\langle \text{valid-edge } x \rangle \langle \text{valid-edge } z \rangle \langle (-\text{Entry-}) = \text{sourcenode } x \rangle$
 $\langle \text{sourcenode } z = (-\text{Entry-}) \rangle \text{Exit} \langle \text{targetnode } z = (-\text{Exit-}) \rangle$
have $x = z$ **by** (fastforce intro: edge-det)
with $\langle \text{preds } (\text{kinds } \text{as}) \ s \rangle \langle \text{as} = x \# xs \rangle \langle xs = [] \rangle \langle \text{kind } z = (\lambda s. \text{False})_{\checkmark} \rangle$
have False **by** (simp add: kinds-def)
thus ?thesis **by** simp
qed simp

with $\langle \text{targetnode } x \rightarrow^* (-\text{Exit-}) \rangle$ **obtain** $x' \text{ } xs'$ **where** $xs = xs'@[x]$
and $\text{targetnode } x \rightarrow^* (-\text{Low-})$ **and** $\text{kind } x' = (\lambda s. \text{True})_{\checkmark}$
by(*fastforce elim:Exit-path-Low-path*)
with $\langle (-\text{Entry-}) = \text{sourcenode } x \rangle \langle \text{valid-edge } x \rangle$
have $\langle (-\text{Entry-}) \rightarrow^* (-\text{Low-}) \rangle$ **by**(*fastforce intro:Cons-path*)
from $\langle as = x\#xs \rangle \langle xs = xs'@[x] \rangle$ **have** $as = (x\#xs')@[x]$ **by** *simp*
with $\langle \text{preds } (\text{kinds } as) \ s \rangle$ **have** $\text{preds } (\text{kinds } (x\#xs')) \ s$
by(*simp add:kinds-def preds-split*)
from $\langle (-\text{Entry-}) \rightarrow^* (-\text{Exit-}) \rangle$ **obtain** $y \text{ } ys$ **where** $as' = y\#ys$
and $\langle (-\text{Entry-}) = \text{sourcenode } y \rangle$ **and** $\text{valid-edge } y$
and $\text{targetnode } y \rightarrow^* (-\text{Exit-})$
apply(*cases as' = []*)
apply(*simp, drule empty-path-nodes, drule Entry-noteq-Exit, simp*)
by(*erule path-split-Cons*)
from $\langle \text{valid-edge } y \rangle$ **have** $\text{valid-node } (\text{targetnode } y)$ **by** *simp*
hence $\text{inner-node } (\text{targetnode } y)$
proof(*cases rule:valid-node-cases*)
case *Entry*
with $\langle \text{valid-edge } y \rangle$ **have** *False* **by**(*rule Entry-target*)
thus *?thesis* **by** *simp*
next
case *Exit*
with $\langle \text{targetnode } y \rightarrow^* (-\text{Exit-}) \rangle$ **have** $ys = []$
by $\neg(\text{rule path-Exit-source, simp})$
from *Entry-Exit-edge* **obtain** z **where** $\text{valid-edge } z$
and $\text{sourcenode } z = (-\text{Entry-})$ **and** $\text{targetnode } z = (-\text{Exit-})$
and $\text{kind } z = (\lambda s. \text{False})_{\checkmark}$ **by** *blast*
from $\langle \text{valid-edge } y \rangle \langle \text{valid-edge } z \rangle \langle (-\text{Entry-}) = \text{sourcenode } y \rangle$
 $\langle \text{sourcenode } z = (-\text{Entry-}) \rangle$ *Exit* $\langle \text{targetnode } z = (-\text{Exit-}) \rangle$
have $y = z$ **by**(*fastforce intro:edge-det*)
with $\langle \text{preds } (\text{kinds } as') \ s' \rangle \langle as' = y\#ys \rangle \langle ys = [] \rangle \langle \text{kind } z = (\lambda s. \text{False})_{\checkmark} \rangle$
have *False* **by**(*simp add:kinds-def*)
thus *?thesis* **by** *simp*
qed *simp*
with $\langle \text{targetnode } y \rightarrow^* (-\text{Exit-}) \rangle$ **obtain** $y' \text{ } ys'$ **where** $ys = ys'@[y]$
and $\text{targetnode } y \rightarrow^* (-\text{Low-})$ **and** $\text{kind } y' = (\lambda s. \text{True})_{\checkmark}$
by(*fastforce elim:Exit-path-Low-path*)
with $\langle (-\text{Entry-}) = \text{sourcenode } y \rangle \langle \text{valid-edge } y \rangle$
have $\langle (-\text{Entry-}) \rightarrow^* (-\text{Low-}) \rangle$ **by**(*fastforce intro:Cons-path*)
from $\langle as' = y\#ys \rangle \langle ys = ys'@[y] \rangle$ **have** $as' = (y\#ys')@[y]$ **by** *simp*
with $\langle \text{preds } (\text{kinds } as') \ s' \rangle$ **have** $\text{preds } (\text{kinds } (y\#ys')) \ s'$
by(*simp add:kinds-def preds-split*)
from $\langle s \approx_L s' \rangle \langle (-\text{High-}) \notin \text{backward-slice } S \rangle \langle (-\text{Low-}) \in S \rangle$
 $\langle (-\text{Entry-}) \rightarrow^* (-\text{Low-}) \rangle \langle \text{preds } (\text{kinds } (x\#xs')) \ s \rangle$
 $\langle (-\text{Entry-}) \rightarrow^* (-\text{Low-}) \rangle \langle \text{preds } (\text{kinds } (y\#ys')) \ s' \rangle$
have $\text{transfers } (\text{kinds } (x\#xs')) \ s \approx_L \text{transfers } (\text{kinds } (y\#ys')) \ s'$
by(*rule nonInterference-path-to-Low*)
with $\langle as = x\#xs \rangle \langle xs = xs'@[x] \rangle \langle \text{kind } x' = (\lambda s. \text{True})_{\checkmark} \rangle$
 $\langle as' = y\#ys \rangle \langle ys = ys'@[y] \rangle \langle \text{kind } y' = (\lambda s. \text{True})_{\checkmark} \rangle$

show *?thesis* **by**(*simp add:kinds-def transfers-split*)
qed

end

The second theorem assumes that we have a operational semantics, whose evaluations are written $\langle c, s \rangle \Rightarrow \langle c', s' \rangle$ and which conforms to the CFG. The correctness theorem then states that if no high variable influenced a low variable and the initial states were low equivalent, the resulting states are again low equivalent:

locale *NonInterferenceIntra* =
NonInterferenceIntraGraph sourcenode targetnode kind valid-edge Entry
Def Use state-val backward-slice Exit H L High Low +
BackwardSlice-wf sourcenode targetnode kind valid-edge Entry Def Use state-val
backward-slice sem identifies
for *sourcenode* :: 'edge $\Rightarrow$ 'node **and** *targetnode* :: 'edge $\Rightarrow$ 'node
and *kind* :: 'edge $\Rightarrow$ 'state edge-kind **and** *valid-edge* :: 'edge $\Rightarrow$ bool
and *Entry* :: 'node ('('Entry'-')) **and** *Def* :: 'node $\Rightarrow$ 'var set
and *Use* :: 'node $\Rightarrow$ 'var set **and** *state-val* :: 'state $\Rightarrow$ 'var $\Rightarrow$ 'val
and *backward-slice* :: 'node set $\Rightarrow$ 'node set
and *sem* :: 'com $\Rightarrow$ 'state $\Rightarrow$ 'com $\Rightarrow$ 'state $\Rightarrow$ bool
 (((1 $\langle$ -,/- $\rangle$) $\Rightarrow$ / (1 $\langle$ -,/- $\rangle$)) [0,0,0,0] 81)
and *identifies* :: 'node $\Rightarrow$ 'com $\Rightarrow$ bool (- $\triangleq$ - [51, 0] 80)
and *Exit* :: 'node ('('Exit'-'))
and *H* :: 'var set **and** *L* :: 'var set
and *High* :: 'node ('('High'-')) **and** *Low* :: 'node ('('Low'-')) +
fixes *final* :: 'com $\Rightarrow$ bool
assumes *final-edge-Low*: $\llbracket \text{final } c; n \triangleq c \rrbracket$
 $\Rightarrow \exists a. \text{valid-edge } a \wedge \text{sourcenode } a = n \wedge \text{targetnode } a = (-\text{Low-}) \wedge \text{kind } a =$
 $\uparrow \text{id}$
begin

The following theorem needs the explicit edge from $(-\text{High-})$ to n . An approach using a *init* predicate for initial statements, being reachable from $(-\text{High-})$ via a $(\lambda s. \text{True})_{\vee}$ edge, does not work as the same statement could be identified by several nodes, some initial, some not. E.g., in the program **while** (**True**) **Skip**; **Skip** two nodes identify this initial statement: the initial node and the node within the loop (because of loop unrolling).

theorem *nonInterference*:

assumes $s_1 \approx_L s_2$ **and** $(-\text{High-}) \notin \text{backward-slice } S$ **and** $(-\text{Low-}) \in S$
and *valid-edge* a **and** *sourcenode* $a = (-\text{High-})$ **and** *targetnode* $a = n$
and *kind* $a = (\lambda s. \text{True})_{\vee}$ **and** $n \triangleq c$ **and** *final* c'
and $\langle c, s_1 \rangle \Rightarrow \langle c', s_1' \rangle$ **and** $\langle c, s_2 \rangle \Rightarrow \langle c', s_2' \rangle$
shows $s_1' \approx_L s_2'$

proof –

from *High-target-Entry-edge* **obtain** ax **where** *valid-edge* ax
and *sourcenode* $ax = (-\text{Entry-})$ **and** *targetnode* $ax = (-\text{High-})$

and $\text{kind } ax = (\lambda s. \text{True})_{\checkmark}$ **by** *blast*
from $\langle n \triangleq c \rangle \langle \langle c, s_1 \rangle \Rightarrow \langle c', s_1' \rangle \rangle$
obtain $n_1 \text{ } as_1$ **where** $n - as_1 \rightarrow^* n_1$ **and** *transfers* (*kinds* as_1) $s_1 = s_1'$
and *preds* (*kinds* as_1) s_1 **and** $n_1 \triangleq c'$
by (*fastforce dest:fundamental-property*)
from $\langle n - as_1 \rightarrow^* n_1 \rangle \langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = (-\text{High-}) \rangle \langle \text{targetnode } a = n \rangle$
have $(-\text{High-}) - a \# as_1 \rightarrow^* n_1$ **by** (*rule Cons-path*)
from $\langle \text{final } c' \rangle \langle n_1 \triangleq c' \rangle$
obtain a_1 **where** *valid-edge* a_1 **and** *sourcenode* $a_1 = n_1$
and *targetnode* $a_1 = (-\text{Low-})$ **and** *kind* $a_1 = \uparrow id$ **by** (*fastforce dest:final-edge-Low*)
hence $n_1 - [a_1] \rightarrow^* (-\text{Low-})$ **by** (*fastforce intro:path-edge*)
with $\langle (-\text{High-}) - a \# as_1 \rightarrow^* n_1 \rangle$ **have** $(-\text{High-}) - (a \# as_1) @ [a_1] \rightarrow^* (-\text{Low-})$
by (*rule path-Append*)
with $\langle \text{valid-edge } ax \rangle \langle \text{sourcenode } ax = (-\text{Entry-}) \rangle \langle \text{targetnode } ax = (-\text{High-}) \rangle$
have $(-\text{Entry-}) - ax \# ((a \# as_1) @ [a_1]) \rightarrow^* (-\text{Low-})$ **by** $-(\text{rule Cons-path})$
from $\langle \text{kind } ax = (\lambda s. \text{True})_{\checkmark} \rangle \langle \text{kind } a = (\lambda s. \text{True})_{\checkmark} \rangle \langle \text{preds } (\text{kinds } as_1) s_1 \rangle$
 $\langle \text{kind } a_1 = \uparrow id \rangle$ **have** *preds* (*kinds* $(ax \# ((a \# as_1) @ [a_1]))$) s_1
by (*simp add:kinds-def preds-split*)
from $\langle n \triangleq c \rangle \langle \langle c, s_2 \rangle \Rightarrow \langle c', s_2' \rangle \rangle$
obtain $n_2 \text{ } as_2$ **where** $n - as_2 \rightarrow^* n_2$ **and** *transfers* (*kinds* as_2) $s_2 = s_2'$
and *preds* (*kinds* as_2) s_2 **and** $n_2 \triangleq c'$
by (*fastforce dest:fundamental-property*)
from $\langle n - as_2 \rightarrow^* n_2 \rangle \langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = (-\text{High-}) \rangle \langle \text{targetnode } a = n \rangle$
have $(-\text{High-}) - a \# as_2 \rightarrow^* n_2$ **by** (*rule Cons-path*)
from $\langle \text{final } c' \rangle \langle n_2 \triangleq c' \rangle$
obtain a_2 **where** *valid-edge* a_2 **and** *sourcenode* $a_2 = n_2$
and *targetnode* $a_2 = (-\text{Low-})$ **and** *kind* $a_2 = \uparrow id$ **by** (*fastforce dest:final-edge-Low*)
hence $n_2 - [a_2] \rightarrow^* (-\text{Low-})$ **by** (*fastforce intro:path-edge*)
with $\langle (-\text{High-}) - a \# as_2 \rightarrow^* n_2 \rangle$ **have** $(-\text{High-}) - (a \# as_2) @ [a_2] \rightarrow^* (-\text{Low-})$
by (*rule path-Append*)
with $\langle \text{valid-edge } ax \rangle \langle \text{sourcenode } ax = (-\text{Entry-}) \rangle \langle \text{targetnode } ax = (-\text{High-}) \rangle$
have $(-\text{Entry-}) - ax \# ((a \# as_2) @ [a_2]) \rightarrow^* (-\text{Low-})$ **by** $-(\text{rule Cons-path})$
from $\langle \text{kind } ax = (\lambda s. \text{True})_{\checkmark} \rangle \langle \text{kind } a = (\lambda s. \text{True})_{\checkmark} \rangle \langle \text{preds } (\text{kinds } as_2) s_2 \rangle$
 $\langle \text{kind } a_2 = \uparrow id \rangle$ **have** *preds* (*kinds* $(ax \# ((a \# as_2) @ [a_2]))$) s_2
by (*simp add:kinds-def preds-split*)
from $\langle s_1 \approx_L s_2 \rangle \langle (-\text{High-}) \notin \text{backward-slice } S \rangle \langle (-\text{Low-}) \in S \rangle$
 $\langle (-\text{Entry-}) - ax \# ((a \# as_1) @ [a_1]) \rightarrow^* (-\text{Low-}) \rangle \langle \text{preds } (\text{kinds } (ax \# ((a \# as_1) @ [a_1])))$
 $s_1 \rangle$
 $\langle (-\text{Entry-}) - ax \# ((a \# as_2) @ [a_2]) \rightarrow^* (-\text{Low-}) \rangle \langle \text{preds } (\text{kinds } (ax \# ((a \# as_2) @ [a_2])))$
 $s_2 \rangle$
have *transfers* (*kinds* $(ax \# ((a \# as_1) @ [a_1]))$) $s_1 \approx_L$
transfers (*kinds* $(ax \# ((a \# as_2) @ [a_2]))$) s_2
by (*rule nonInterference-path-to-Low*)
with $\langle \text{kind } ax = (\lambda s. \text{True})_{\checkmark} \rangle \langle \text{kind } a = (\lambda s. \text{True})_{\checkmark} \rangle \langle \text{kind } a_1 = \uparrow id \rangle \langle \text{kind } a_2$
 $= \uparrow id \rangle$
 $\langle \text{transfers } (\text{kinds } as_1) s_1 = s_1' \rangle \langle \text{transfers } (\text{kinds } as_2) s_2 = s_2' \rangle$
show *?thesis* **by** (*simp add:kinds-def transfers-split*)
qed

end

end

4.13 Framework Graph Lifting for Noninterference

theory *LiftingIntra*

imports *NonInterferenceIntra* ../Slicing/StaticIntra/CDepInstantiations

begin

In this section, we show how a valid CFG from the slicing framework in [?] can be lifted to fulfil all properties of the *NonInterferenceIntraGraph* locale. Basically, we redefine the hitherto existing *Entry* and *Exit* nodes as new *High* and *Low* nodes, and introduce two new nodes *NewEntry* and *NewExit*. Then, we have to lift all functions to operate on this new graph.

4.13.1 Liftings

The datatypes

datatype *'node LDCFG-node* = *Node 'node*

| *NewEntry*

| *NewExit*

type-synonym (*'edge, 'node, 'state*) *LDCFG-edge* =

'node LDCFG-node × (*'state edge-kind*) × *'node LDCFG-node*

Lifting *valid-edge*

inductive *lift-valid-edge* :: (*'edge* ⇒ *bool*) ⇒ (*'edge* ⇒ *'node*) ⇒ (*'edge* ⇒ *'node*)

⇒

(*'edge* ⇒ *'state edge-kind*) ⇒ *'node* ⇒ *'node* ⇒ (*'edge, 'node, 'state*) *LDCFG-edge*

⇒

bool

for *valid-edge*::*'edge* ⇒ *bool* **and** *src*::*'edge* ⇒ *'node* **and** *trg*::*'edge* ⇒ *'node*

and *knd*::*'edge* ⇒ *'state edge-kind* **and** *E*::*'node* **and** *X*::*'node*

where *lve-edge*:

[[*valid-edge a*; *src a* ≠ *E* ∨ *trg a* ≠ *X*;

e = (*Node (src a), knd a, Node (trg a)*)]

⇒ *lift-valid-edge valid-edge src trg knd E X e*

| *lve-Entry-edge*:

e = (*NewEntry, (λs. True)*, *Node E*)

⇒ *lift-valid-edge valid-edge src trg knd E X e*

| *lve-Exit-edge*:
 $e = (\text{Node } X, (\lambda s. \text{True})_{\checkmark}, \text{NewExit})$
 $\implies \text{lift-valid-edge valid-edge src trg kno } E X e$

| *lve-Entry-Exit-edge*:
 $e = (\text{NewEntry}, (\lambda s. \text{False})_{\checkmark}, \text{NewExit})$
 $\implies \text{lift-valid-edge valid-edge src trg kno } E X e$

lemma [*simp*]: $\neg \text{lift-valid-edge valid-edge src trg kno } E X (\text{Node } E, \text{et}, \text{Node } X)$
by (*auto elim: lift-valid-edge.cases*)

Lifting Def and Use sets

inductive-set *lift-Def-set* :: $('node \Rightarrow 'var \text{ set}) \Rightarrow 'node \Rightarrow 'node \Rightarrow$
 $'var \text{ set} \Rightarrow 'var \text{ set} \Rightarrow ('node \text{ LDCFG-node} \times 'var) \text{ set}$
for *Def*:: $('node \Rightarrow 'var \text{ set})$ **and** *E*:: $'node$ **and** *X*:: $'node$
and *H*:: $'var \text{ set}$ **and** *L*:: $'var \text{ set}$

where *lift-Def-node*:

$V \in \text{Def } n \implies (\text{Node } n, V) \in \text{lift-Def-set Def } E X H L$

| *lift-Def-High*:

$V \in H \implies (\text{Node } E, V) \in \text{lift-Def-set Def } E X H L$

abbreviation *lift-Def* :: $('node \Rightarrow 'var \text{ set}) \Rightarrow 'node \Rightarrow 'node \Rightarrow$
 $'var \text{ set} \Rightarrow 'var \text{ set} \Rightarrow 'node \text{ LDCFG-node} \Rightarrow 'var \text{ set}$
where *lift-Def Def E X H L n* $\equiv \{ V. (n, V) \in \text{lift-Def-set Def } E X H L \}$

inductive-set *lift-Use-set* :: $('node \Rightarrow 'var \text{ set}) \Rightarrow 'node \Rightarrow 'node \Rightarrow$
 $'var \text{ set} \Rightarrow 'var \text{ set} \Rightarrow ('node \text{ LDCFG-node} \times 'var) \text{ set}$
for *Use*:: $'node \Rightarrow 'var \text{ set}$ **and** *E*:: $'node$ **and** *X*:: $'node$
and *H*:: $'var \text{ set}$ **and** *L*:: $'var \text{ set}$

where

lift-Use-node:

$V \in \text{Use } n \implies (\text{Node } n, V) \in \text{lift-Use-set Use } E X H L$

| *lift-Use-High*:

$V \in H \implies (\text{Node } E, V) \in \text{lift-Use-set Use } E X H L$

| *lift-Use-Low*:

$V \in L \implies (\text{Node } X, V) \in \text{lift-Use-set Use } E X H L$

abbreviation *lift-Use* :: $('node \Rightarrow 'var \text{ set}) \Rightarrow 'node \Rightarrow 'node \Rightarrow$
 $'var \text{ set} \Rightarrow 'var \text{ set} \Rightarrow 'node \text{ LDCFG-node} \Rightarrow 'var \text{ set}$

where $\text{lift-Use } Use \ E \ X \ H \ L \ n \equiv \{V. (n, V) \in \text{lift-Use-set } Use \ E \ X \ H \ L\}$

4.13.2 The lifting lemmas

Lifting the basic locales

abbreviation $\text{src} :: ('edge, 'node, 'state) \text{LDCFG-edge} \Rightarrow 'node \text{LDCFG-node}$
 where $\text{src } a \equiv \text{fst } a$

abbreviation $\text{trg} :: ('edge, 'node, 'state) \text{LDCFG-edge} \Rightarrow 'node \text{LDCFG-node}$
 where $\text{trg } a \equiv \text{snd}(\text{snd } a)$

definition $\text{knd} :: ('edge, 'node, 'state) \text{LDCFG-edge} \Rightarrow 'state \text{edge-kind}$
 where $\text{knd } a \equiv \text{fst}(\text{snd } a)$

lemma *lift-CFG*:

assumes $\text{wf} : \text{CFGExit-wf } \text{sourcenode } \text{targetnode } \text{kind } \text{valid-edge } \text{Entry } \text{Def } \text{Use}$
 $\text{state-val } \text{Exit}$

shows $\text{CFG } \text{src } \text{trg}$

$(\text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit}) \text{NewEntry}$

proof –

interpret $\text{CFGExit-wf } \text{sourcenode } \text{targetnode } \text{kind } \text{valid-edge } \text{Entry } \text{Def } \text{Use}$
 $\text{state-val } \text{Exit}$

by $(\text{rule } \text{wf})$

show $?thesis$

proof

fix a **assume** $\text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit } a$

and $\text{trg } a = \text{NewEntry}$

thus False **by** $(\text{fastforce } \text{elim} : \text{lift-valid-edge.cases})$

next

fix $a \ a'$

assume $\text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit } a$

and $\text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit } a'$

and $\text{src } a = \text{src } a' \text{ and } \text{trg } a = \text{trg } a'$

thus $a = a'$

proof $(\text{induct } \text{rule} : \text{lift-valid-edge.induct})$

case lve-edge **thus** $?case$ **by** $-(\text{erule } \text{lift-valid-edge.cases}, \text{auto } \text{dest} : \text{edge-det})$

qed $(\text{auto } \text{elim} : \text{lift-valid-edge.cases})$

qed

qed

lemma *lift-CFG-wf*:

assumes $\text{wf} : \text{CFGExit-wf } \text{sourcenode } \text{targetnode } \text{kind } \text{valid-edge } \text{Entry } \text{Def } \text{Use}$
 $\text{state-val } \text{Exit}$

shows $\text{CFG-wf } \text{src } \text{trg } \text{knd}$

$(\text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit}) \text{NewEntry}$

$(\text{lift-Def } \text{Def } \text{Entry } \text{Exit } H \ L) (\text{lift-Use } \text{Use } \text{Entry } \text{Exit } H \ L) \text{state-val}$

proof –

```

interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
           state-val Exit
  by(rule wf)
interpret CFG:CFG src trg kend
           lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
  by(fastforce intro:lift-CFG wf)
show ?thesis
proof
  show lift-Def Def Entry Exit H L NewEntry = {}  $\wedge$ 
        lift-Use Use Entry Exit H L NewEntry = {}
    by(fastforce elim:lift-Use-set.cases lift-Def-set.cases)
next
  fix a V s
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and V  $\notin$  lift-Def Def Entry Exit H L (src a) and pred (kend a) s
  thus state-val (transfer (kend a) s) V = state-val s V
  proof(induct rule:lift-valid-edge.induct)
    case lve-edge
    thus ?case by(fastforce intro:CFG-edge-no-Def-equal dest:lift-Def-node[of -
Def]
      simp:knd-def)
    qed(auto simp:knd-def)
next
  fix a s s'
  assume assms:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
a
     $\forall V \in \text{lift-Use Use Entry Exit H L (src a)}. \text{state-val } s \text{ } V = \text{state-val } s' \text{ } V$ 
     $\text{pred (kend a) } s \text{ } \text{pred (kend a) } s'$ 
  show  $\forall V \in \text{lift-Def Def Entry Exit H L (src a)}.$ 
     $\text{state-val (transfer (kend a) } s) \text{ } V = \text{state-val (transfer (kend a) } s') \text{ } V$ 
  proof
    fix V assume V  $\in$  lift-Def Def Entry Exit H L (src a)
    with assms
    show state-val (transfer (kend a) s) V = state-val (transfer (kend a) s') V
    proof(induct rule:lift-valid-edge.induct)
      case (lve-edge a e)
      show ?case
      proof (cases Node (sourcenode a) = Node Entry)
        case True
        hence sourcenode a = Entry by simp
        from Entry-Exit-edge obtain a' where valid-edge a'
          and sourcenode a' = Entry and targetnode a' = Exit
          and kind a' = ( $\lambda s. \text{False}$ ) $\checkmark$  by blast
        have  $\exists Q. \text{kind } a = (Q)\checkmark$ 
        proof(cases targetnode a = Exit)
          case True
          with  $\langle \text{valid-edge } a \rangle \langle \text{valid-edge } a' \rangle \langle \text{sourcenode } a = \text{Entry} \rangle$ 
             $\langle \text{sourcenode } a' = \text{Entry} \rangle \langle \text{targetnode } a' = \text{Exit} \rangle$ 
          have a = a' by(fastforce dest:edge-det)

```

```

    with  $\langle \text{kind } a' = (\lambda s. \text{False})_{\checkmark} \rangle$  show ?thesis by simp
next
  case False
  with  $\langle \text{valid-edge } a \rangle \langle \text{valid-edge } a' \rangle \langle \text{sourcenode } a = \text{Entry} \rangle$ 
     $\langle \text{sourcenode } a' = \text{Entry} \rangle \langle \text{targetnode } a' = \text{Exit} \rangle$ 
  show ?thesis by(auto dest:deterministic)
qed
from True  $\langle V \in \text{lift-Def Def Entry Exit H L (src } e) \rangle \text{Entry-empty}$ 
   $\langle e = (\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)) \rangle$ 
have  $V \in H$  by(fastforce elim:lift-Def-set.cases)
from True  $\langle e = (\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)) \rangle$ 
   $\langle \text{sourcenode } a \neq \text{Entry} \vee \text{targetnode } a \neq \text{Exit} \rangle$ 
have  $\forall V \in H. V \in \text{lift-Use Use Entry Exit H L (src } e)$ 
  by(fastforce intro:lift-Use-High)
with  $\langle \forall V \in \text{lift-Use Use Entry Exit H L (src } e). \text{state-val } s \ V = \text{state-val } s' \ V \rangle \langle V \in H \rangle$ 
have  $\text{state-val } s \ V = \text{state-val } s' \ V$  by simp
with  $\langle e = (\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)) \rangle$ 
   $\langle \exists Q. \text{kind } a = (Q)_{\checkmark} \rangle$ 
show ?thesis by(fastforce simp:knd-def)
next
  case False
  { fix  $V'$  assume  $V' \in \text{Use (sourcenode } a)$ 
    with  $\langle e = (\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)) \rangle$ 
    have  $V' \in \text{lift-Use Use Entry Exit H L (src } e)$ 
    by(fastforce intro:lift-Use-node)
  }
  with  $\langle \forall V \in \text{lift-Use Use Entry Exit H L (src } e). \text{state-val } s \ V = \text{state-val } s' \ V \rangle$ 
  have  $\forall V \in \text{Use (sourcenode } a). \text{state-val } s \ V = \text{state-val } s' \ V$ 
  by fastforce
  from  $\langle \text{valid-edge } a \rangle \text{this} \langle \text{pred (knd } e) \ s \rangle \langle \text{pred (knd } e) \ s' \rangle$ 
     $\langle e = (\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)) \rangle$ 
  have  $\forall V \in \text{Def (sourcenode } a). \text{state-val (transfer (kind } a) \ s) \ V =$ 
     $\text{state-val (transfer (kind } a) \ s') \ V$ 
  by -(erule CFG-edge-transfer-uses-only-Use,auto simp:knd-def)
  from  $\langle V \in \text{lift-Def Def Entry Exit H L (src } e) \rangle \text{False}$ 
     $\langle e = (\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)) \rangle$ 
  have  $V \in \text{Def (sourcenode } a)$  by(fastforce elim:lift-Def-set.cases)
  with  $\langle \forall V \in \text{Def (sourcenode } a). \text{state-val (transfer (kind } a) \ s) \ V =$ 
     $\text{state-val (transfer (kind } a) \ s') \ V} \rangle$ 
     $\langle e = (\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)) \rangle$ 
  show ?thesis by(simp add:knd-def)
qed
next
  case (lve-Entry-edge e)
  from  $\langle V \in \text{lift-Def Def Entry Exit H L (src } e) \rangle$ 
     $\langle e = (\text{NewEntry}, (\lambda s. \text{True})_{\checkmark}, \text{Node Entry}) \rangle$ 
  have False by(fastforce elim:lift-Def-set.cases)

```

```

      thus ?case by simp
    next
      case (lve-Exit-edge e)
      from ⟨V ∈ lift-Def Def Entry Exit H L (src e)⟩
        ⟨e = (Node Exit, (λs. True)✓, NewExit)⟩
      have False
      by (fastforce elim:lift-Def-set.cases intro!:Entry-noteq-Exit simp:Exit-empty)
      thus ?case by simp
    qed (simp add:knd-def)
  qed
next
  fix a s s'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and pred (knd a) s
  and ∀ V ∈ lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
  thus pred (knd a) s'
  by (induct rule:lift-valid-edge.induct,
      auto elim!:CFG-edge-Uses-pred-equal dest:lift-Use-node simp:knd-def)
next
  fix a a'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
  and src a = src a' and trg a ≠ trg a'
  thus ∃ Q Q'. knd a = (Q)✓ ∧ knd a' = (Q')✓ ∧
    (∀ s. (Q s ⟶ ¬ Q' s) ∧ (Q' s ⟶ ¬ Q s))
  proof (induct rule:lift-valid-edge.induct)
    case (lve-edge a e)
    from ⟨lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'⟩
      ⟨valid-edge a⟩ ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
      ⟨src e = src a'⟩ ⟨trg e ≠ trg a'⟩
    show ?case
    proof (induct rule:lift-valid-edge.induct)
      case lve-edge thus ?case by (auto dest:deterministic simp:knd-def)
    next
      case (lve-Exit-edge e')
      from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
        ⟨e' = (Node Exit, (λs. True)✓, NewExit)⟩ ⟨src e = src e'⟩
      have sourcenode a = Exit by simp
      with ⟨valid-edge a⟩ have False by (rule Exit-source)
      thus ?case by simp
    qed auto
  qed (fastforce elim:lift-valid-edge.cases simp:knd-def)+
qed
qed

```

lemma *lift-CFGExit*:

assumes *wf:CFGExit-wf* *sourcenode targetnode kind valid-edge Entry Def Use*
state-val Exit

shows *CFGExit src trg knđ*
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry NewExit
proof –
interpret *CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use*
state-val Exit
by(*rule wf*)
interpret *CFG:CFG src trg knđ*
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
by(*fastforce intro:lift-CFG wf*)
show ?thesis
proof
fix *a* **assume** *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *src a = NewExit*
thus *False* **by**(*fastforce elim:lift-valid-edge.cases*)
next
from *lve-Entry-Exit-edge*
show $\exists a. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \wedge$
 $\text{src } a = \text{NewEntry} \wedge \text{trg } a = \text{NewExit} \wedge \text{knđ } a = (\lambda s. \text{False})_{\checkmark}$
by(*fastforce simp:knđ-def*)
qed
qed

lemma *lift-CFGExit-wf*:
assumes *wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use*
state-val Exit
shows *CFGExit-wf src trg knđ*
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
proof –
interpret *CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use*
state-val Exit
by(*rule wf*)
interpret *CFGExit:CFGExit src trg knđ*
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
NewEntry NewExit
by(*fastforce intro:lift-CFGExit wf*)
interpret *CFG-wf:CFG-wf src trg knđ*
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
NewEntry lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L state-val
by(*fastforce intro:lift-CFG-wf wf*)
show ?thesis
proof
show *lift-Def Def Entry Exit H L NewExit = {}* $\wedge$
lift-Use Use Entry Exit H L NewExit = {}
by(*fastforce elim:lift-Use-set.cases lift-Def-set.cases*)
qed
qed

Lifting *wod-backward-slice*

lemma *lift-wod-backward-slice*:

fixes *valid-edge* **and** *sourcenode* **and** *targetnode* **and** *kind* **and** *Entry* **and** *Exit*
and *Def* **and** *Use* **and** *H* **and** *L*

defines *lve:lve* $\equiv$ *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*

and *lDef:lDef* $\equiv$ *lift-Def Def Entry Exit H L*

and *lUse:lUse* $\equiv$ *lift-Use Use Entry Exit H L*

assumes *wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use*
state-val Exit

and $H \cap L = \{\}$ **and** $H \cup L = \text{UNIV}$

shows *NonInterferenceIntraGraph src trg knl lve NewEntry lDef lUse state-val*
(CFG-wf.wod-backward-slice src trg lve lDef lUse)
NewExit H L (Node Entry) (Node Exit)

proof –

interpret *CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use*
state-val Exit

by(*rule wf*)

interpret *CFGExit-wf*:

CFGExit-wf src trg knl lve NewEntry lDef lUse state-val NewExit

by(*fastforce intro:lift-CFGExit-wf wf simp:lve lDef lUse*)

from *wf lve* **have** *CFG:CFG src trg lve NewEntry*

by(*fastforce intro:lift-CFG*)

from *wf lve lDef lUse* **have** *CFG-wf:CFG-wf src trg knl lve NewEntry*
lDef lUse state-val

by(*fastforce intro:lift-CFG-wf*)

show *?thesis*

proof

fix *n S*

assume $n \in \text{CFG-wf.wod-backward-slice src trg lve lDef lUse } S$

with *CFG-wf* **show** *CFG.valid-node src trg lve n*

by –(*rule CFG-wf.wod-backward-slice-valid-node*)

next

fix *n S* **assume** *CFG.valid-node src trg lve n* **and** $n \in S$

with *CFG-wf* **show** $n \in \text{CFG-wf.wod-backward-slice src trg lve lDef lUse } S$

by –(*rule CFG-wf.refl*)

next

fix *n' S n V*

assume $n' \in \text{CFG-wf.wod-backward-slice src trg lve lDef lUse } S$

and *CFG-wf.data-dependence src trg lve lDef lUse n V n'*

with *CFG-wf* **show** $n \in \text{CFG-wf.wod-backward-slice src trg lve lDef lUse } S$

by –(*rule CFG-wf.dd-closed*)

next

fix *n S*

from *CFG-wf*

have $(\exists m. (\text{CFG.obs src trg lve } n$

$(\text{CFG-wf.wod-backward-slice src trg lve lDef lUse } S)) = \{m\}) \vee$

$\text{CFG.obs src trg lve } n (\text{CFG-wf.wod-backward-slice src trg lve lDef lUse } S) =$

$\{\}$

by(*rule CFG-wf.obs-singleton*)


```

thus finite
  (CFG.obs src trg lve n (CFG-wf.wod-backward-slice src trg lve lDef lUse S))
  by fastforce
next
  fix n S
  from CFG-wf
  have ( $\exists m. (CFG.obs\ src\ trg\ lve\ n$ 
    (CFG-wf.wod-backward-slice src trg lve lDef lUse S)) =  $\{m\}$ )  $\vee$ 
    CFG.obs src trg lve n (CFG-wf.wod-backward-slice src trg lve lDef lUse S) =
  {}
    by(rule CFG-wf.obs-singleton)
  thus card (CFG.obs src trg lve n
    (CFG-wf.wod-backward-slice src trg lve lDef lUse S))  $\leq 1$ 
    by fastforce
next
  fix a assume lve a and src a = NewEntry
  with lve show trg a = NewExit  $\vee$  trg a = Node Entry
    by(fastforce elim:lift-valid-edge.cases)
next
  from lve-Entry-edge lve
  show  $\exists a. lve\ a \wedge src\ a = NewEntry \wedge trg\ a = Node\ Entry \wedge kno\ a = (\lambda s. True)_{\checkmark}$ 
  by(fastforce simp:knd-def)
next
  fix a assume lve a and trg a = Node Entry
  with lve show src a = NewEntry by(fastforce elim:lift-valid-edge.cases)
next
  fix a assume lve a and trg a = NewExit
  with lve show src a = NewEntry  $\vee$  src a = Node Exit
    by(fastforce elim:lift-valid-edge.cases)
next
  from lve-Exit-edge lve
  show  $\exists a. lve\ a \wedge src\ a = Node\ Exit \wedge trg\ a = NewExit \wedge kno\ a = (\lambda s. True)_{\checkmark}$ 
  by(fastforce simp:knd-def)
next
  fix a assume lve a and src a = Node Exit
  with lve show trg a = NewExit by(fastforce elim:lift-valid-edge.cases)
next
  from lDef show lDef (Node Entry) = H
    by(fastforce elim:lift-Def-set.cases intro:lift-Def-High)
next
  from Entry-noteq-Exit lUse show lUse (Node Entry) = H
    by(fastforce elim:lift-Use-set.cases intro:lift-Use-High)
next
  from Entry-noteq-Exit lUse show lUse (Node Exit) = L
    by(fastforce elim:lift-Use-set.cases intro:lift-Use-Low)
next
  from  $\langle H \cap L = \{\} \rangle$  show  $H \cap L = \{\}$  .
next

```

from $\langle H \cup L = UNIV \rangle$ **show** $H \cup L = UNIV$.
qed
qed

Lifting PDG-BS with standard-control-dependence

lemma *lift-Postdomination*:

assumes $wf:CFGExit\text{-}wf$ $sourcenode$ $targetnode$ $kind$ $valid\text{-}edge$ $Entry$ Def Use
 $state\text{-}val$ $Exit$
and $pd:Postdomination$ $sourcenode$ $targetnode$ $kind$ $valid\text{-}edge$ $Entry$ $Exit$
and $inner:CFGExit.inner\text{-}node$ $sourcenode$ $targetnode$ $valid\text{-}edge$ $Entry$ $Exit$ nx
shows *Postdomination* src trg knd
 $(lift\text{-}valid\text{-}edge\ valid\text{-}edge\ sourcenode\ targetnode\ kind\ Entry\ Exit)\ NewEntry\ NewExit$
proof –
interpret *Postdomination* $sourcenode$ $targetnode$ $kind$ $valid\text{-}edge$ $Entry$ $Exit$
by(*rule* pd)
interpret $CFGExit\text{-}wf:CFGExit\text{-}wf$ src trg knd
 $lift\text{-}valid\text{-}edge\ valid\text{-}edge\ sourcenode\ targetnode\ kind\ Entry\ Exit\ NewEntry$
 $lift\text{-}Def\ Def\ Entry\ Exit\ H\ L\ lift\text{-}Use\ Use\ Entry\ Exit\ H\ L\ state\text{-}val\ NewExit$
by(*fastforce* *intro*: $lift\text{-}CFGExit\text{-}wf\ wf$)
from wf **have** $CFG:CFG$ src trg
 $(lift\text{-}valid\text{-}edge\ valid\text{-}edge\ sourcenode\ targetnode\ kind\ Entry\ Exit)\ NewEntry$
by(*rule* $lift\text{-}CFG$)
show *?thesis*
proof
fix n **assume** $CFG.valid\text{-}node\ src\ trg$
 $(lift\text{-}valid\text{-}edge\ valid\text{-}edge\ sourcenode\ targetnode\ kind\ Entry\ Exit)\ n$
show $\exists as. CFG.path\ src\ trg$
 $(lift\text{-}valid\text{-}edge\ valid\text{-}edge\ sourcenode\ targetnode\ kind\ Entry\ Exit)$
 $NewEntry\ as\ n$
proof(*cases* n)
case $NewEntry$
have $lift\text{-}valid\text{-}edge\ valid\text{-}edge\ sourcenode\ targetnode\ kind\ Entry\ Exit$
 $(NewEntry,(\lambda s. False)_{\checkmark},NewExit)$ **by**(*fastforce* *intro*: $lve\text{-}Entry\text{-}Exit\text{-}edge$)
with $NewEntry$ **have** $CFG.path\ src\ trg$
 $(lift\text{-}valid\text{-}edge\ valid\text{-}edge\ sourcenode\ targetnode\ kind\ Entry\ Exit)$
 $NewEntry\ []\ n$
by(*fastforce* *intro*: $CFG.empty\text{-}path[OF\ CFG]$ *simp*: $CFG.valid\text{-}node\text{-}def[OF\ CFG]$)
thus *?thesis* **by** *blast*
next
case $NewExit$
have $lift\text{-}valid\text{-}edge\ valid\text{-}edge\ sourcenode\ targetnode\ kind\ Entry\ Exit$
 $(NewEntry,(\lambda s. False)_{\checkmark},NewExit)$ **by**(*fastforce* *intro*: $lve\text{-}Entry\text{-}Exit\text{-}edge$)
with $NewExit$ **have** $CFG.path\ src\ trg$
 $(lift\text{-}valid\text{-}edge\ valid\text{-}edge\ sourcenode\ targetnode\ kind\ Entry\ Exit)$
 $NewEntry\ [(NewEntry,(\lambda s. False)_{\checkmark},NewExit)]\ n$
by(*fastforce* *intro*: $CFG.Cons\text{-}path[OF\ CFG]$ $CFG.empty\text{-}path[OF\ CFG]$
 $simp:CFG.valid\text{-}node\text{-}def[OF\ CFG]$)

```

thus ?thesis by blast
next
case (Node m)
with Entry-Exit-edge ⟨CFG.valid-node src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n⟩
have valid-node m
  by(auto elim:lift-valid-edge.cases
    simp:CFG.valid-node-def[OF CFG] valid-node-def)
thus ?thesis
proof(cases m rule:valid-node-cases)
  case Entry
  have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
    (NewEntry,(λs. True)√,Node Entry) by(fastforce intro:lve-Entry-edge)
  with Entry Node have CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    NewEntry [(NewEntry,(λs. True)√,Node Entry)] n
    by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
      simp:CFG.valid-node-def[OF CFG])
  thus ?thesis by blast
next
case Exit
  from inner obtain ax where valid-edge ax and inner-node (sourcenode
ax)
    and targetnode ax = Exit by(erule inner-node-Exit-edge)
  hence lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
    (Node (sourcenode ax),kind ax,Node Exit)
    by(auto intro:lift-valid-edge.lve-edge simp:inner-node-def)
  hence path:CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node (sourcenode ax)) [(Node (sourcenode ax),kind ax,Node Exit)]
    (Node Exit)
    by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
      simp:CFG.valid-node-def[OF CFG])
  have edge:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
    (NewEntry,(λs. True)√,Node Entry) by(fastforce intro:lve-Entry-edge)
  from ⟨inner-node (sourcenode ax)⟩ have valid-node (sourcenode ax)
    by(rule inner-is-valid)
  then obtain asx where Entry -asx→* sourcenode ax
    by(fastforce dest:Entry-path)
  from this ⟨valid-edge ax⟩ have ∃ es. CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node Entry) es (Node (sourcenode ax))
  proof(induct asx arbitrary:ax rule:rev-induct)
    case Nil
    from ⟨Entry -[]→* sourcenode ax⟩ have sourcenode ax = Entry by
fastforce
    hence CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node Entry) [] (Node (sourcenode ax))

```

```

    apply simp apply (rule CFG.empty-path[OF CFG])
    by (auto intro:lve-Entry-edge simp:CFG.valid-node-def[OF CFG])
  thus ?case by blast
next
case (snoc x xs)
note IH = ⟨ $\wedge ax. \llbracket \text{Entry} -xs \rightarrow^* \text{sourcenode } ax; \text{valid-edge } ax \rrbracket \implies$ 
 $\exists es. \text{CFG.path src trg}$ 
 $(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit})$ 
 $(\text{Node Entry}) es (\text{Node (sourcenode } ax)) \rangle$ 
from ⟨ $\text{Entry} -xs@[x] \rightarrow^* \text{sourcenode } ax$ ⟩
have  $\text{Entry} -xs \rightarrow^* \text{sourcenode } x$  and  $\text{valid-edge } x$ 
and  $\text{targetnode } x = \text{sourcenode } ax$  by (auto elim:path-split-snoc)
{ assume  $\text{targetnode } x = \text{Exit}$ 
  with ⟨ $\text{valid-edge } ax$ ⟩ ⟨ $\text{targetnode } x = \text{sourcenode } ax$ ⟩
  have False by  $\neg(\text{rule Exit-source, simp+})$  }
hence  $\text{targetnode } x \neq \text{Exit}$  by clarsimp
with ⟨ $\text{valid-edge } x$ ⟩ ⟨ $\text{targetnode } x = \text{sourcenode } ax$ ⟩ [THEN sym]
have  $\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}$ 
 $(\text{Node (sourcenode } x), \text{kind } x, \text{Node (sourcenode } ax))$ 
by (fastforce intro:lift-valid-edge.lve-edge)
hence  $\text{path:CFG.path src trg}$ 
 $(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit})$ 
 $(\text{Node (sourcenode } x)) [(\text{Node (sourcenode } x), \text{kind } x, \text{Node (sourcenode } ax))]$ 
 $(\text{Node (sourcenode } ax))$ 
by (fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
simp:CFG.valid-node-def[OF CFG])
from IH[OF ⟨ $\text{Entry} -xs \rightarrow^* \text{sourcenode } x$ ⟩ ⟨ $\text{valid-edge } x$ ⟩] obtain es
where  $\text{CFG.path src trg}$ 
 $(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit})$ 
 $(\text{Node Entry}) es (\text{Node (sourcenode } x))$  by blast
with path have  $\text{CFG.path src trg}$ 
 $(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit})$ 
 $(\text{Node Entry}) (es@[ (\text{Node (sourcenode } x), \text{kind } x, \text{Node (sourcenode } ax)) ])$ 
 $(\text{Node (sourcenode } ax))$ 
by  $\neg(\text{rule CFG.path-Append[OF CFG]})$ 
thus ?case by blast
qed
then obtain es where  $\text{CFG.path src trg}$ 
 $(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit})$ 
 $(\text{Node Entry}) es (\text{Node (sourcenode } ax))$  by blast
with path have  $\text{CFG.path src trg}$ 
 $(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit})$ 
 $(\text{Node Entry}) (es@[ (\text{Node (sourcenode } ax), \text{kind } ax, \text{Node Exit}) ])$   $(\text{Node Exit})$ 
by  $\neg(\text{rule CFG.path-Append[OF CFG]})$ 
with edge have  $\text{CFG.path src trg}$ 
 $(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit})$ 
 $\text{NewEntry } ((\text{NewEntry}, (\lambda s. \text{True}), \surd, \text{Node Entry}) \#$ 

```

```

      (es@ [(Node (sourcenode ax),kind ax,Node Exit)])) (Node Exit)
    by(fastforce intro:CFG.Cons-path[OF CFG])
  with Node Exit show ?thesis by fastforce
next
case inner
from ⟨valid-node m⟩ obtain as where Entry -as→* m
  by(fastforce dest:Entry-path)
with inner have ∃ es. CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) es (Node m)
proof(induct arbitrary:m rule:rev-induct)
case Nil
from ⟨Entry -[]→* m⟩
have m = Entry by fastforce
with lve-Entry-edge have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) [] (Node m)
by(fastforce intro:CFG.empty-path[OF CFG] simp:CFG.valid-node-def[OF
CFG])
thus ?case by blast
next
case (snoc x xs)
note IH = ⟨∧m. ⟦inner-node m; Entry -xs→* m⟧
  ⇒ ∃ es. CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) es (Node m)⟩
from ⟨Entry -xs@[x]→* m⟩ have Entry -xs→* sourcenode x
  and valid-edge x and m = targetnode x by(auto elim:path-split-snoc)
with ⟨inner-node m⟩
have edge:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (Node (sourcenode x),kind x,Node m)
  by(fastforce intro:lve-edge simp:inner-node-def)
hence path:CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node (sourcenode x)) [(Node (sourcenode x),kind x,Node m)] (Node m)
  by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
    simp:CFG.valid-node-def[OF CFG])
from ⟨valid-edge x⟩ have valid-node (sourcenode x) by simp
thus ?case
proof(cases sourcenode x rule:valid-node-cases)
case Entry
with edge have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) [(Node Entry,kind x,Node m)] (Node m)
  apply - apply(rule CFG.Cons-path[OF CFG])
  apply(rule CFG.empty-path[OF CFG])
  by(auto simp:CFG.valid-node-def[OF CFG])
thus ?thesis by blast
next

```

```

    case Exit
    with (valid-edge x) have False by(rule Exit-source)
    thus ?thesis by simp
next
case inner
from IH[OF this (Entry -xs→* sourcenode x)] obtain es
  where CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node Entry) es (Node (sourcenode x)) by blast
with path have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) (es@[ (Node (sourcenode x),kind x,Node m)]) (Node m)
  by -(rule CFG.path-Append[OF CFG])
thus ?thesis by blast
qed
qed
then obtain es where path:CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) es (Node m) by blast
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (NewEntry,(λs. True)√,Node Entry) by(fastforce intro:lve-Entry-edge)
from this path Node have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  NewEntry ((NewEntry,(λs. True)√,Node Entry)#es) n
  by(fastforce intro:CFG.Cons-path[OF CFG])
thus ?thesis by blast
qed
qed
next
fix n assume CFG.valid-node src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n
show ∃ as. CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  n as NewExit
proof(cases n)
case NewEntry
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (NewEntry,(λs. False)√,NewExit) by(fastforce intro:lve-Entry-Exit-edge)
with NewEntry have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  n [(NewEntry,(λs. False)√,NewExit)] NewExit
  by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
    simp:CFG.valid-node-def[OF CFG])
thus ?thesis by blast
next
case NewExit
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (NewEntry,(λs. False)√,NewExit) by(fastforce intro:lve-Entry-Exit-edge)
with NewExit have CFG.path src trg

```

```

    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    n [] NewExit
    by(fastforce intro:CFG.empty-path[OF CFG] simp:CFG.valid-node-def[OF
CFG])
  thus ?thesis by blast
next
case (Node m)
with Entry-Exit-edge ⟨CFG.valid-node src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n⟩
have valid-node m
  by(auto elim:lift-valid-edge.cases
    simp:CFG.valid-node-def[OF CFG] valid-node-def)
thus ?thesis
proof(cases m rule:valid-node-cases)
  case Entry
  from inner obtain ax where valid-edge ax and inner-node (targetnode ax)
    and sourcenode ax = Entry by(erule inner-node-Entry-edge)
  hence edge:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
    (Node Entry,kind ax,Node (targetnode ax))
    by(auto intro:lift-valid-edge.lve-edge simp:inner-node-def)
  have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
    (Node Exit,(λs. True)√,NewExit) by(fastforce intro:lve-Exit-edge)
  hence path:CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node Exit) [(Node Exit,(λs. True)√,NewExit)] (NewExit)
    by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
      simp:CFG.valid-node-def[OF CFG])
  from ⟨inner-node (targetnode ax)⟩ have valid-node (targetnode ax)
    by(rule inner-is-valid)
  then obtain asx where targetnode ax - asx →* Exit by(fastforce dest:Exit-path)
  from this ⟨valid-edge ax⟩ have ∃ es. CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node (targetnode ax)) es (Node Exit)
  proof(induct asx arbitrary:ax)
    case Nil
    from ⟨targetnode ax - [] →* Exit⟩ have targetnode ax = Exit by fastforce
    hence CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node (targetnode ax)) [] (Node Exit)
      apply simp apply(rule CFG.empty-path[OF CFG])
      by(auto intro:lve-Exit-edge simp:CFG.valid-node-def[OF CFG])
    thus ?case by blast
  next
  case (Cons x xs)
  note IH = ⟨∧ ax. [targetnode ax - xs →* Exit; valid-edge ax] ⇒
    ∃ es. CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node (targetnode ax)) es (Node Exit)⟩
  from ⟨targetnode ax - x # xs →* Exit⟩

```

```

have targetnode  $x \rightarrow^* \text{Exit}$  and valid-edge  $x$ 
  and sourcenode  $x = \text{targetnode } ax$  by (auto elim:path-split-Cons)
{ assume sourcenode  $x = \text{Entry}$ 
  with  $\langle \text{valid-edge } ax \rangle \langle \text{sourcenode } x = \text{targetnode } ax \rangle$ 
  have  $\text{False}$  by  $\neg(\text{rule Entry-target, simp+})$  }
hence sourcenode  $x \neq \text{Entry}$  by clarsimp
with  $\langle \text{valid-edge } x \rangle \langle \text{sourcenode } x = \text{targetnode } ax \rangle$  [THEN sym]
have edge:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (Node (targetnode  $ax$ ), kind  $x$ , Node (targetnode  $x$ ))
  by (fastforce intro:lift-valid-edge.lve-edge)
from IH[OF  $\langle \text{targetnode } x \rightarrow^* \text{Exit} \rangle \langle \text{valid-edge } x \rangle$ ] obtain  $es$ 
  where CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node (targetnode  $x$ ))  $es$  (Node Exit) by blast
with edge have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node (targetnode  $ax$ ))
  ((Node (targetnode  $ax$ ), kind  $x$ , Node (targetnode  $x$ )) #  $es$ ) (Node Exit)
  by (fastforce intro:CFG.Cons-path[OF CFG])
thus ?case by blast
qed
then obtain  $es$  where CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node (targetnode  $ax$ ))  $es$  (Node Exit) by blast
with edge have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) ((Node Entry, kind  $ax$ , Node (targetnode  $ax$ )) #  $es$ ) (Node
Exit)
  by (fastforce intro:CFG.Cons-path[OF CFG])
with path have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) (((Node Entry, kind  $ax$ , Node (targetnode  $ax$ )) #  $es$ ) @
  [(Node Exit, ( $\lambda s. \text{True}$ ) $_{\checkmark}$ , NewExit)]) NewExit
  by  $\neg(\text{rule CFG.path-Append[OF CFG]})$ 
with Node Entry show ?thesis by fastforce
next
case Exit
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (Node Exit, ( $\lambda s. \text{True}$ ) $_{\checkmark}$ , NewExit) by (fastforce intro:lve-Exit-edge)
with Exit Node have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
   $n$  [(Node Exit, ( $\lambda s. \text{True}$ ) $_{\checkmark}$ , NewExit)] NewExit
  by (fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
simp:CFG.valid-node-def[OF CFG])
thus ?thesis by blast
next
case inner
from  $\langle \text{valid-node } m \rangle$  obtain  $as$  where  $m \rightarrow^* \text{Exit}$ 
by (fastforce dest:Exit-path)

```



```

with inner have  $\exists es. CFG.path\ src\ trg$ 
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node m) es (Node Exit)
proof(induct as arbitrary:m)
  case Nil
  from  $\langle m - [] \rightarrow^* Exit \rangle$ 
  have  $m = Exit$  by fastforce
  with lve-Exit-edge have  $CFG.path\ src\ trg$ 
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node m)  $[]$  (Node Exit)
  by(fastforce intro:CFG.empty-path[OF CFG] simp:CFG.valid-node-def[OF
CFG])
  thus ?case by blast
next
  case (Cons x xs)
  note  $IH = \langle \bigwedge m. \llbracket inner-node\ m; m - xs \rightarrow^* Exit \rrbracket$ 
     $\implies \exists es. CFG.path\ src\ trg$ 
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node m) es (Node Exit)  $\rangle$ 
  from  $\langle m - x \# xs \rightarrow^* Exit \rangle$  have  $targetnode\ x - xs \rightarrow^* Exit$ 
    and valid-edge x and  $m = sourcenode\ x$  by(auto elim:path-split-Cons)
  with (inner-node m)
  have edge:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
    (Node m,kind x,Node (targetnode x))
    by(fastforce intro:lve-edge simp:inner-node-def)
  from (valid-edge x) have valid-node (targetnode x) by simp
  thus ?case
  proof(cases targetnode x rule:valid-node-cases)
    case Entry
    with (valid-edge x) have False by(rule Entry-target)
    thus ?thesis by simp
  next
    case Exit
    with edge have  $CFG.path\ src\ trg$ 
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node m)  $[(Node\ m,kind\ x,Node\ Exit)]$  (Node Exit)
    apply – apply(rule CFG.Cons-path[OF CFG])
    apply(rule CFG.empty-path[OF CFG])
    by(auto simp:CFG.valid-node-def[OF CFG])
    thus ?thesis by blast
  next
    case inner
    from  $IH[OF\ this\ \langle targetnode\ x - xs \rightarrow^* Exit \rangle]$  obtain es
      where  $CFG.path\ src\ trg$ 
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node (targetnode x)) es (Node Exit) by blast
    with edge have  $CFG.path\ src\ trg$ 
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node m)  $((Node\ m,kind\ x,Node\ (targetnode\ x))\#es)$  (Node Exit)

```

```

      by(fastforce intro:CFG.Cons-path[OF CFG])
    thus ?thesis by blast
  qed
qed
then obtain es where path:CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node m) es (Node Exit) by blast
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (Node Exit, (λs. True)√, NewExit) by(fastforce intro:lve-Exit-edge)
hence CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Exit) [(Node Exit, (λs. True)√, NewExit)] NewExit
  by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
    simp:CFG.valid-node-def[OF CFG])
with path Node have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  n (es@[(Node Exit, (λs. True)√, NewExit)]) NewExit
  by(fastforce intro:CFG.path-Append[OF CFG])
thus ?thesis by blast
qed
qed
qed
qed

```

lemma *lift-PDG-scd*:

```

  assumes PDG:PDG sourcenode targetnode kind valid-edge Entry Def Use state-val
  Exit
  (Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit)
  and pd:Postdomination sourcenode targetnode kind valid-edge Entry Exit
  and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
  shows PDG src trg kno
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
  (Postdomination.standard-control-dependence src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit)
proof –
  interpret PDG sourcenode targetnode kind valid-edge Entry Def Use state-val
  Exit
    Postdomination.standard-control-dependence sourcenode targetnode
    valid-edge Exit
    by(rule PDG)
  have wf:CFGExit.wf sourcenode targetnode kind valid-edge Entry Def Use
    state-val Exit by(unfold-locales)
  from wf pd inner have pd':Postdomination src trg kno
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    NewEntry NewExit
    by(rule lift-Postdomination)
  from wf have CFG:CFG src trg

```

```

    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  by(rule lift-CFG)
from wf have CFG-wf:CFG-wf src trg kno
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val
  by(rule lift-CFG-wf)
from wf have CFGExit:CFGExit src trg kno
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  NewEntry NewExit
  by(rule lift-CFGExit)
from wf have CFGExit-wf:CFGExit-wf src trg kno
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
  by(rule lift-CFGExit-wf)
show ?thesis
proof
  fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and trg a = NewEntry
  with CFG show False by(rule CFG.Entry-target)
next
  fix a a'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
    and src a = src a' and trg a = trg a'
  with CFG show a = a' by(rule CFG.edge-det)
next
  from CFG-wf
  show lift-Def Def Entry Exit H L NewEntry = {} ∧
    lift-Use Use Entry Exit H L NewEntry = {}
    by(rule CFG-wf.Entry-empty)
next
  fix a V s
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and V ∉ lift-Def Def Entry Exit H L (src a) and pred (kno a) s
  with CFG-wf show state-val (transfer (kno a) s) V = state-val s V
    by(rule CFG-wf.CFG-edge-no-Def-equal)
next
  fix a s s'
  assume assms:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
a
    ∀ V ∈ lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
    pred (kno a) s pred (kno a) s'
  with CFG-wf show ∀ V ∈ lift-Def Def Entry Exit H L (src a).
    state-val (transfer (kno a) s) V = state-val (transfer (kno a) s') V
    by(rule CFG-wf.CFG-edge-transfer-uses-only-Use)
next
  fix a s s'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and pred (kno a) s

```

```

    and  $\forall V \in \text{lift-Use Use Entry Exit } H \ L \ (\text{src } a). \text{ state-val } s \ V = \text{state-val } s' \ V$ 
  with CFG-wf show pred (knd a) s' by(rule CFG-wf.CFG-edge-Uses-pred-equal)
next
  fix a a'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
    and src a = src a' and trg a  $\neq$  trg a'
  with CFG-wf show  $\exists Q \ Q'. \text{knd } a = (Q)_{\checkmark} \wedge \text{knd } a' = (Q')_{\checkmark} \wedge$ 
     $(\forall s. (Q \ s \longrightarrow \neg Q' \ s) \wedge (Q' \ s \longrightarrow \neg Q \ s))$ 
    by(rule CFG-wf.deterministic)
next
  fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and src a = NewExit
  with CFGExit show False by(rule CFGExit.Exit-source)
next
  from CFGExit
  show  $\exists a. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \wedge$ 
     $\text{src } a = \text{NewEntry} \wedge \text{trg } a = \text{NewExit} \wedge \text{knd } a = (\lambda s. \text{False})_{\checkmark}$ 
    by(rule CFGExit.Entry-Exit-edge)
next
  from CFGExit-wf
  show lift-Def Def Entry Exit H L NewExit = {}  $\wedge$ 
    lift-Use Use Entry Exit H L NewExit = {}
    by(rule CFGExit-wf.Exit-empty)
next
  fix n n'
  assume scd:Postdomination.standard-control-dependence src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n
  n'
  show n'  $\neq$  NewExit
  proof(rule ccontr)
    assume  $\neg n' \neq \text{NewExit}$ 
    hence n' = NewExit by simp
    with scd pd' show False
    by(fastforce intro:Postdomination.Exit-not-standard-control-dependent)
  qed
next
  fix n n'
  assume Postdomination.standard-control-dependence src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n
  n'
  thus  $\exists as. \text{CFG.path src trg}$ 
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
     $n \text{ as } n' \wedge as \neq []$ 
    by(fastforce simp:Postdomination.standard-control-dependence-def[OF pd'])
  qed
qed

```

lemma *lift-PDG-standard-backward-slice*:

fixes *valid-edge* **and** *sourcenode* **and** *targetnode* **and** *kind* **and** *Entry* **and** *Exit*
and *Def* **and** *Use* **and** *H* **and** *L*

defines *lve*:*lve* $\equiv$ *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
and *lDef*:*lDef* $\equiv$ *lift-Def Def Entry Exit H L*
and *lUse*:*lUse* $\equiv$ *lift-Use Use Entry Exit H L*

assumes *PDG*:*PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
(Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit)
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
and $H \cap L = \{\}$ **and** $H \cup L = \text{UNIV}$

shows *NonInterferenceIntraGraph src trg knl lve NewEntry lDef lUse state-val*
(PDG.PDG-BS src trg lve lDef lUse
(Postdomination.standard-control-dependence src trg lve NewExit))
NewExit H L (Node Entry) (Node Exit)

proof –

interpret *PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
Postdomination.standard-control-dependence sourcenode targetnode
valid-edge Exit

by(*rule PDG*)

have *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use*
state-val Exit **by**(*unfold-locales*)

interpret *wf'*:*CFGExit-wf src trg knl lve NewEntry lDef lUse state-val NewExit*
by(*fastforce intro:lift-CFGExit-wf wf simp:lve lDef lUse*)

from *PDG pd inner lve lDef lUse* **have** *PDG'*:*PDG src trg knl*
lve NewEntry lDef lUse state-val NewExit
(Postdomination.standard-control-dependence src trg lve NewExit)
by(*fastforce intro:lift-PDG-scd*)

from *wf pd inner* **have** *pd'*:*Postdomination src trg knl*
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry NewExit
by(*rule lift-Postdomination*)

from *wf lve* **have** *CFG*:*CFG src trg lve NewEntry*
by(*fastforce intro:lift-CFG*)

from *wf lve lDef lUse*

have *CFG-wf*:*CFG-wf src trg knl lve NewEntry lDef lUse state-val*
by(*fastforce intro:lift-CFG-wf*)

from *wf lve* **have** *CFGExit*:*CFGExit src trg knl lve NewEntry NewExit*
by(*fastforce intro:lift-CFGExit*)

from *wf lve lDef lUse*

have *CFGExit-wf*:*CFGExit-wf src trg knl lve NewEntry lDef lUse state-val NewExit*
by(*fastforce intro:lift-CFGExit-wf*)

show *?thesis*

proof

```

fix  $n \ S$ 
assume  $n \in PDG.PDG-BS \ src \ trg \ lve \ lDef \ lUse$ 
  ( $Postdomination.standard-control-dependence \ src \ trg \ lve \ NewExit$ )  $S$ 
with  $PDG'$  show  $CFG.valid-node \ src \ trg \ lve \ n$ 
  by( $rule \ PDG.PDG-BS-valid-node$ )
next
  fix  $n \ S$  assume  $CFG.valid-node \ src \ trg \ lve \ n$  and  $n \in S$ 
  thus  $n \in PDG.PDG-BS \ src \ trg \ lve \ lDef \ lUse$ 
    ( $Postdomination.standard-control-dependence \ src \ trg \ lve \ NewExit$ )  $S$ 
    by( $fastforce \ intro:PDG.PDG-path-Nil[OF \ PDG'] \ simp:PDG.PDG-BS-def[OF$ 
 $PDG']$ )
  next
    fix  $n' \ S \ n \ V$ 
    assume  $n' \in PDG.PDG-BS \ src \ trg \ lve \ lDef \ lUse$ 
      ( $Postdomination.standard-control-dependence \ src \ trg \ lve \ NewExit$ )  $S$ 
      and  $CFG.wf.data-dependence \ src \ trg \ lve \ lDef \ lUse \ n \ V \ n'$ 
    thus  $n \in PDG.PDG-BS \ src \ trg \ lve \ lDef \ lUse$ 
      ( $Postdomination.standard-control-dependence \ src \ trg \ lve \ NewExit$ )  $S$ 
      by( $fastforce \ intro:PDG.PDG-path-Append[OF \ PDG'] \ PDG.PDG-path-ddep[OF$ 
 $PDG']$ 
         $PDG.PDG-ddep-edge[OF \ PDG'] \ simp:PDG.PDG-BS-def[OF$ 
 $PDG']$ 
         $split:split-if-asm$ )
    next
      fix  $n \ S$ 
      interpret  $PDGx:PDG \ src \ trg \ knd \ lve \ NewEntry \ lDef \ lUse \ state-val \ NewExit$ 
         $Postdomination.standard-control-dependence \ src \ trg \ lve \ NewExit$ 
      by( $rule \ PDG'$ )
      interpret  $pdx:Postdomination \ src \ trg \ knd \ lve \ NewEntry \ NewExit$ 
        by( $fastforce \ intro:pd' \ simp:lve$ )
      have  $scd:StandardControlDependencePDG \ src \ trg \ knd \ lve \ NewEntry$ 
         $lDef \ lUse \ state-val \ NewExit$  by( $unfold-locales$ )
      from  $StandardControlDependencePDG.obs-singleton[OF \ scd]$ 
      have ( $\exists m. \ CFG.obs \ src \ trg \ lve \ n$ 
        ( $PDG.PDG-BS \ src \ trg \ lve \ lDef \ lUse$ 
          ( $Postdomination.standard-control-dependence \ src \ trg \ lve \ NewExit$ )  $S$ ) =  $\{m\}$ )
 $\vee$ 
        ( $CFG.obs \ src \ trg \ lve \ n$ 
          ( $PDG.PDG-BS \ src \ trg \ lve \ lDef \ lUse$ 
            ( $Postdomination.standard-control-dependence \ src \ trg \ lve \ NewExit$ )  $S$ ) =  $\{\}$ )
        by( $fastforce \ simp:StandardControlDependencePDG.PDG-BS-s-def[OF \ scd]$ )
      thus  $finite \ (CFG.obs \ src \ trg \ lve \ n$ 
        ( $PDG.PDG-BS \ src \ trg \ lve \ lDef \ lUse$ 
          ( $Postdomination.standard-control-dependence \ src \ trg \ lve \ NewExit$ )  $S$ ))
        by  $fastforce$ 
    next
      fix  $n \ S$ 
      interpret  $PDGx:PDG \ src \ trg \ knd \ lve \ NewEntry \ lDef \ lUse \ state-val \ NewExit$ 
         $Postdomination.standard-control-dependence \ src \ trg \ lve \ NewExit$ 

```

```

    by(rule PDG')
  interpret pdx:Postdomination src trg kno lve NewEntry NewExit
    by(fastforce intro:pd' simp:lve)
  have scd:StandardControlDependencePDG src trg kno lve NewEntry
    lDef lUse state-val NewExit by(unfold-locales)
  from StandardControlDependencePDG.obs-singleton[OF scd]
  have (∃ m. CFG.obs src trg lve n
    (PDG.PDG-BS src trg lve lDef lUse
    (Postdomination.standard-control-dependence src trg lve NewExit) S) = {m})
  ∨
    CFG.obs src trg lve n
    (PDG.PDG-BS src trg lve lDef lUse
    (Postdomination.standard-control-dependence src trg lve NewExit) S) = {}
  by(fastforce simp:StandardControlDependencePDG.PDG-BS-s-def[OF scd])
  thus card (CFG.obs src trg lve n
    (PDG.PDG-BS src trg lve lDef lUse
    (Postdomination.standard-control-dependence src trg lve NewExit) S)) ≤ 1
  by fastforce
next
  fix a assume lve a and src a = NewEntry
  with lve show trg a = NewExit ∨ trg a = Node Entry
    by(fastforce elim:lift-valid-edge.cases)
next
  from lve-Entry-edge lve
  show ∃ a. lve a ∧ src a = NewEntry ∧ trg a = Node Entry ∧ kno a = (λs.
True)√
    by(fastforce simp:kno-def)
next
  fix a assume lve a and trg a = Node Entry
  with lve show src a = NewEntry by(fastforce elim:lift-valid-edge.cases)
next
  fix a assume lve a and trg a = NewExit
  with lve show src a = NewEntry ∨ src a = Node Exit
    by(fastforce elim:lift-valid-edge.cases)
next
  from lve-Exit-edge lve
  show ∃ a. lve a ∧ src a = Node Exit ∧ trg a = NewExit ∧ kno a = (λs. True)√
    by(fastforce simp:kno-def)
next
  fix a assume lve a and src a = Node Exit
  with lve show trg a = NewExit by(fastforce elim:lift-valid-edge.cases)
next
  from lDef show lDef (Node Entry) = H
    by(fastforce elim:lift-Def-set.cases intro:lift-Def-High)
next
  from Entry-noteq-Exit lUse show lUse (Node Entry) = H
    by(fastforce elim:lift-Use-set.cases intro:lift-Use-High)
next
  from Entry-noteq-Exit lUse show lUse (Node Exit) = L

```

```

    by(fastforce elim:lift-Use-set.cases intro:lift-Use-Low)
  next
    from  $\langle H \cap L = \{\} \rangle$  show  $H \cap L = \{\}$  .
  next
    from  $\langle H \cup L = UNIV \rangle$  show  $H \cup L = UNIV$  .
qed
qed

```

Lifting PDG-BS with weak-control-dependence

lemma *lift-StrongPostdomination*:

```

  assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
          state-val Exit
  and spd:StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit
  and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
  shows StrongPostdomination src trg kno
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry NewExit
proof -
  interpret StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit
  by(rule spd)
  have pd:Postdomination sourcenode targetnode kind valid-edge Entry Exit
  by(unfold-locals)
  interpret pd':Postdomination src trg kno
  lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  NewEntry NewExit
  by(fastforce intro:wf inner lift-Postdomination pd)
  interpret CFGExit-wf:CFGExit-wf src trg kno
  lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
  lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L state-val NewExit
  by(fastforce intro:lift-CFGExit-wf wf)
  from wf have CFG:CFG src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  by(rule lift-CFG)
  show ?thesis
proof
  fix n assume CFG.valid-node src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n
  show finite
  {n'.  $\exists a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a' \wedge$ 
    src a' = n  $\wedge$  trg a' = n'}
  proof(cases n)
    case NewEntry
    hence {n'.  $\exists a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind}$ 
      Entry Exit a'  $\wedge$  src a' = n  $\wedge$  trg a' = n'} = {NewExit,Node
Entry}
    by(auto elim:lift-valid-edge.cases intro:lift-valid-edge.intros)
    thus ?thesis by simp
  next
    case NewExit

```


hence $\{n'. \exists a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind}$
 $\text{Entry Exit } a' \wedge \text{src } a' = n \wedge \text{trg } a' = n'\} = \{\}$
by *fastforce*
thus *?thesis* **by** *simp*
next
case $(\text{Node } m)$
with *Entry-Exit-edge* $(\text{CFG.valid-node src trg}$
 $(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}) n)$
have *valid-node* m
by $(\text{auto elim:lift-valid-edge.cases}$
 $\text{simp:CFG.valid-node-def[OF CFG] valid-node-def})$
hence *finite* $\{m'. \exists a'. \text{valid-edge } a' \wedge \text{sourcenode } a' = m \wedge \text{targetnode } a' =$
 $m'\}$
by $(\text{rule successor-set-finite})$
have $\{m'. \exists a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind}$
 $\text{Entry Exit } a' \wedge \text{src } a' = \text{Node } m \wedge \text{trg } a' = \text{Node } m'\} \subseteq$
 $\{m'. \exists a'. \text{valid-edge } a' \wedge \text{sourcenode } a' = m \wedge \text{targetnode } a' = m'\}$
by $(\text{fastforce elim:lift-valid-edge.cases})$
with $\{ \text{finite } \{m'. \exists a'. \text{valid-edge } a' \wedge \text{sourcenode } a' = m \wedge \text{targetnode } a' =$
 $m'\} \}$
have *finite* $\{m'. \exists a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind}$
 $\text{Entry Exit } a' \wedge \text{src } a' = \text{Node } m \wedge \text{trg } a' = \text{Node } m'\}$
by $-(\text{rule finite-subset})$
hence *finite* $(\text{Node } ' \{m'. \exists a'. \text{lift-valid-edge valid-edge sourcenode}$
 $\text{targetnode kind Entry Exit } a' \wedge \text{src } a' = \text{Node } m \wedge \text{trg } a' = \text{Node } m'\})$
by *fastforce*
hence *fin:finite* $((\text{Node } ' \{m'. \exists a'. \text{lift-valid-edge valid-edge sourcenode}$
 $\text{targetnode kind Entry Exit } a' \wedge \text{src } a' = \text{Node } m \wedge \text{trg } a' = \text{Node } m'\}) \cup$
 $\{\text{NewEntry, NewExit}\})$ **by** *fastforce*
with *Node* **have** $\{n'. \exists a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind}$
 $\text{Entry Exit } a' \wedge \text{src } a' = n \wedge \text{trg } a' = n'\} \subseteq$
 $(\text{Node } ' \{m'. \exists a'. \text{lift-valid-edge valid-edge sourcenode}$
 $\text{targetnode kind Entry Exit } a' \wedge \text{src } a' = \text{Node } m \wedge \text{trg } a' = \text{Node } m'\}) \cup$
 $\{\text{NewEntry, NewExit}\})$ **by** *auto* $(\text{case-tac } x, \text{auto})$
with *fin* **show** *?thesis* **by** $-(\text{rule finite-subset})$
qed
qed
qed

lemma *lift-PDG-wcd*:

assumes *PDG:PDG sourcenode targetnode kind valid-edge Entry Def Use state-val*
Exit
 $(\text{StrongPostdomination.weak-control-dependence sourcenode targetnode}$
 $\text{valid-edge Exit})$

```

and spd:StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit
and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
shows PDG src trg kn
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
(StrongPostdomination.weak-control-dependence src trg
 (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit)
proof –
  interpret PDG sourcenode targetnode kind valid-edge Entry Def Use state-val
Exit
    StrongPostdomination.weak-control-dependence sourcenode targetnode
      valid-edge Exit
    by(rule PDG)
  have wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
    state-val Exit by(unfold-locales)
  from wf spd inner have spd':StrongPostdomination src trg kn
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    NewEntry NewExit
    by(rule lift-StrongPostdomination)
  from wf have CFG:CFG src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
    by(rule lift-CFG)
  from wf have CFG-wf:CFG-wf src trg kn
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
    (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val
    by(rule lift-CFG-wf)
  from wf have CFGExit:CFGExit src trg kn
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    NewEntry NewExit
    by(rule lift-CFGExit)
  from wf have CFGExit-wf:CFGExit-wf src trg kn
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
    (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
    by(rule lift-CFGExit-wf)
  show ?thesis
  proof
    fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and trg a = NewEntry
      with CFG show False by(rule CFG.Entry-target)
  next
    fix a a'
    assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
      and src a = src a' and trg a = trg a'
      with CFG show a = a' by(rule CFG.edge-det)
  next
    from CFG-wf
    show lift-Def Def Entry Exit H L NewEntry = {}  $\wedge$ 
      lift-Use Use Entry Exit H L NewEntry = {}

```

```

    by(rule CFG-wf.Entry-empty)
  next
    fix a V s
    assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and  $V \notin \text{lift-Def Def Entry Exit H L (src a)}$  and  $\text{pred (knd a) s}$ 
    with CFG-wf show state-val (transfer (knd a) s) V = state-val s V
      by(rule CFG-wf.CFG-edge-no-Def-equal)
  next
    fix a s s'
    assume assms:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
a
       $\forall V \in \text{lift-Use Use Entry Exit H L (src a)}. \text{state-val s V} = \text{state-val s' V}$ 
       $\text{pred (knd a) s} \text{ pred (knd a) s'}$ 
    with CFG-wf show  $\forall V \in \text{lift-Def Def Entry Exit H L (src a)}$ .
      state-val (transfer (knd a) s) V = state-val (transfer (knd a) s') V
      by(rule CFG-wf.CFG-edge-transfer-uses-only-Use)
  next
    fix a s s'
    assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and  $\text{pred (knd a) s}$ 
      and  $\forall V \in \text{lift-Use Use Entry Exit H L (src a)}. \text{state-val s V} = \text{state-val s' V}$ 
    with CFG-wf show  $\text{pred (knd a) s'}$  by(rule CFG-wf.CFG-edge-Uses-pred-equal)
  next
    fix a a'
    assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
      and  $\text{src a} = \text{src a'}$  and  $\text{trg a} \neq \text{trg a'}$ 
    with CFG-wf show  $\exists Q Q'. \text{knd a} = (Q)_{\checkmark} \wedge \text{knd a'} = (Q')_{\checkmark} \wedge$ 
       $(\forall s. (Q s \longrightarrow \neg Q' s) \wedge (Q' s \longrightarrow \neg Q s))$ 
      by(rule CFG-wf.deterministic)
  next
    fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and  $\text{src a} = \text{NewExit}$ 
    with CFGEExit show False by(rule CFGEExit.Exit-source)
  next
    from CFGEExit
    show  $\exists a. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a} \wedge$ 
       $\text{src a} = \text{NewEntry} \wedge \text{trg a} = \text{NewExit} \wedge \text{knd a} = (\lambda s. \text{False})_{\checkmark}$ 
      by(rule CFGEExit.Entry-Exit-edge)
  next
    from CFGEExit-wf
    show lift-Def Def Entry Exit H L NewExit = {}  $\wedge$ 
      lift-Use Use Entry Exit H L NewExit = {}
      by(rule CFGEExit-wf.Exit-empty)
  next
    fix n n'
    assume wcd:StrongPostdomination.weak-control-dependence src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n
n'

```

```

show  $n' \neq \text{NewExit}$ 
proof(rule ccontr)
  assume  $\neg n' \neq \text{NewExit}$ 
  hence  $n' = \text{NewExit}$  by simp
  with wcd spd' show False
    by(fastforce intro:StrongPostdomination.Exit-not-weak-control-dependent)
qed
next
fix  $n\ n'$ 
assume StrongPostdomination.weak-control-dependence src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit  $n$ 
 $n'$ 
  thus  $\exists as. \text{CFG.path src trg}$ 
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
     $n\ as\ n' \wedge as \neq \square$ 
  by(fastforce simp:StrongPostdomination.weak-control-dependence-def[OF spd'])
qed
qed

```

lemma *lift-PDG-weak-backward-slice*:

```

fixes valid-edge and sourcenode and targetnode and kind and Entry and Exit
and Def and Use and H and L
defines lve:lve  $\equiv$  lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
and lDef:lDef  $\equiv$  lift-Def Def Entry Exit H L
and lUse:lUse  $\equiv$  lift-Use Use Entry Exit H L
assumes PDG:PDG sourcenode targetnode kind valid-edge Entry Def Use state-val
Exit
  (StrongPostdomination.weak-control-dependence sourcenode targetnode
valid-edge Exit)
and spd:StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit
and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
and  $H \cap L = \{\}$  and  $H \cup L = \text{UNIV}$ 
shows NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val
  (PDG.PDG-BS src trg lve lDef lUse
    (StrongPostdomination.weak-control-dependence src trg lve NewExit))
    NewExit H L (Node Entry) (Node Exit)

```

proof –

```

interpret PDG sourcenode targetnode kind valid-edge Entry Def Use state-val
Exit
  StrongPostdomination.weak-control-dependence sourcenode targetnode
    valid-edge Exit
by(rule PDG)
have wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
  state-val Exit by(unfold-locales)
interpret wf':CFGExit-wf src trg knd lve NewEntry lDef lUse state-val NewExit
by(fastforce intro:lift-CFGExit-wf wf simp:lve lDef lUse)

```

```

from PDG spd inner lve lDef lUse have PDG':PDG src trg kn  

lve NewEntry lDef lUse state-val NewExit  

(StrongPostdomination.weak-control-dependence src trg lve NewExit)  

by(fastforce intro:lift-PDG-wcd)  

from wf spd inner have spd':StrongPostdomination src trg kn  

(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)  

NewEntry NewExit  

by(rule lift-StrongPostdomination)  

from wf lve have CFG:CFG src trg lve NewEntry  

by(fastforce intro:lift-CFG)  

from wf lve lDef lUse  

have CFG-wf:CFG-wf src trg kn lve NewEntry lDef lUse state-val  

by(fastforce intro:lift-CFG-wf)  

from wf lve have CFGExit:CFGExit src trg kn lve NewEntry NewExit  

by(fastforce intro:lift-CFGExit)  

from wf lve lDef lUse  

have CFGExit-wf:CFGExit-wf src trg kn lve NewEntry lDef lUse state-val  

NewExit  

by(fastforce intro:lift-CFGExit-wf)  

show ?thesis  

proof  

  fix n S  

  assume n ∈ PDG.PDG-BS src trg lve lDef lUse  

(StrongPostdomination.weak-control-dependence src trg lve NewExit) S  

  with PDG' show CFG.valid-node src trg lve n  

  by(rule PDG.PDG-BS-valid-node)  

next  

  fix n S assume CFG.valid-node src trg lve n and n ∈ S  

  thus n ∈ PDG.PDG-BS src trg lve lDef lUse  

(StrongPostdomination.weak-control-dependence src trg lve NewExit) S  

  by(fastforce intro:PDG.PDG-path-Nil[OF PDG'] simp:PDG.PDG-BS-def[OF  

PDG'])  

next  

  fix n' S n V  

  assume n' ∈ PDG.PDG-BS src trg lve lDef lUse  

(StrongPostdomination.weak-control-dependence src trg lve NewExit) S  

and CFG-wf.data-dependence src trg lve lDef lUse n V n'  

thus n ∈ PDG.PDG-BS src trg lve lDef lUse  

(StrongPostdomination.weak-control-dependence src trg lve NewExit) S  

by(fastforce intro:PDG.PDG-path-Append[OF PDG'] PDG.PDG-path-ddep[OF  

PDG']  

PDG.PDG-ddep-edge[OF PDG'] simp:PDG.PDG-BS-def[OF  

PDG']  

split:split-if-asm)  

next  

  fix n S  

interpret PDGx:PDG src trg kn lve NewEntry lDef lUse state-val NewExit  

StrongPostdomination.weak-control-dependence src trg lve NewExit  

by(rule PDG')

```

```

interpret spd:StrongPostdomination src trg knd lve NewEntry NewExit
  by(fastforce intro:spd' simp:lve)
have wcd:WeakControlDependencePDG src trg knd lve NewEntry
  lDef lUse state-val NewExit by(unfold-locales)
from WeakControlDependencePDG.obs-singleton[OF wcd]
have ( $\exists m. \text{CFG.obs src trg lve } n$ 
  (PDG.PDG-BS src trg lve lDef lUse
    (StrongPostdomination.weak-control-dependence src trg lve NewExit) S) =
{m})  $\vee$ 
  CFG.obs src trg lve n
  (PDG.PDG-BS src trg lve lDef lUse
    (StrongPostdomination.weak-control-dependence src trg lve NewExit) S) =
{}
by(fastforce simp:WeakControlDependencePDG.PDG-BS-w-def[OF wcd])
thus finite (CFG.obs src trg lve n
  (PDG.PDG-BS src trg lve lDef lUse
    (StrongPostdomination.weak-control-dependence src trg lve NewExit) S))
by fastforce
next
fix n S
interpret PDGx:PDG src trg knd lve NewEntry lDef lUse state-val NewExit
  StrongPostdomination.weak-control-dependence src trg lve NewExit
by(rule PDG')
interpret spdx:StrongPostdomination src trg knd lve NewEntry NewExit
by(fastforce intro:spd' simp:lve)
have wcd:WeakControlDependencePDG src trg knd lve NewEntry
  lDef lUse state-val NewExit by(unfold-locales)
from WeakControlDependencePDG.obs-singleton[OF wcd]
have ( $\exists m. \text{CFG.obs src trg lve } n$ 
  (PDG.PDG-BS src trg lve lDef lUse
    (StrongPostdomination.weak-control-dependence src trg lve NewExit) S) =
{m})  $\vee$ 
  CFG.obs src trg lve n
  (PDG.PDG-BS src trg lve lDef lUse
    (StrongPostdomination.weak-control-dependence src trg lve NewExit) S) =
{}
by(fastforce simp:WeakControlDependencePDG.PDG-BS-w-def[OF wcd])
thus card (CFG.obs src trg lve n
  (PDG.PDG-BS src trg lve lDef lUse
    (StrongPostdomination.weak-control-dependence src trg lve NewExit) S))  $\leq$ 
1
by fastforce
next
fix a assume lve a and src a = NewEntry
with lve show trg a = NewExit  $\vee$  trg a = Node Entry
by(fastforce elim:lift-valid-edge.cases)
next
from lve-Entry-edge lve
show  $\exists a. \text{lve } a \wedge \text{src } a = \text{NewEntry} \wedge \text{trg } a = \text{Node Entry} \wedge \text{knd } a = (\lambda s.$ 

```

```

True)✓
  by(fastforce simp:knd-def)
next
  fix a assume lve a and trg a = Node Entry
  with lve show src a = NewEntry by(fastforce elim:lift-valid-edge.cases)
next
  fix a assume lve a and trg a = NewExit
  with lve show src a = NewEntry  $\vee$  src a = Node Exit
  by(fastforce elim:lift-valid-edge.cases)
next
  from lve-Exit-edge lve
  show  $\exists a. lve\ a \wedge src\ a = Node\ Exit \wedge trg\ a = NewExit \wedge knd\ a = (\lambda s. True)$ ✓
  by(fastforce simp:knd-def)
next
  fix a assume lve a and src a = Node Exit
  with lve show trg a = NewExit by(fastforce elim:lift-valid-edge.cases)
next
  from lDef show lDef (Node Entry) = H
  by(fastforce elim:lift-Def-set.cases intro:lift-Def-High)
next
  from Entry-noteq-Exit lUse show lUse (Node Entry) = H
  by(fastforce elim:lift-Use-set.cases intro:lift-Use-High)
next
  from Entry-noteq-Exit lUse show lUse (Node Exit) = L
  by(fastforce elim:lift-Use-set.cases intro:lift-Use-Low)
next
  from  $\langle H \cap L = \{\} \rangle$  show  $H \cap L = \{\}$  .
next
  from  $\langle H \cup L = UNIV \rangle$  show  $H \cup L = UNIV$  .
qed
qed

end

```

4.14 Information Flow for While

```

theory NonInterferenceWhile imports
  SemanticsWellFormed
  StaticControlDependences
  ../.. / InformationFlowSlicing / LiftingIntra
begin

locale SecurityTypes =

```

```

fixes  $H :: \text{vname set}$ 
fixes  $L :: \text{vname set}$ 
assumes  $\text{HighLowDistinct}: H \cap L = \{\}$ 
and  $\text{HighLowUNIV}: H \cup L = \text{UNIV}$ 
begin

```

4.14.1 Lifting labels-nodes and Defining final

```

fun  $\text{labels-LDCFG-nodes} :: \text{cmd} \Rightarrow \text{w-node LDCFG-node} \Rightarrow \text{cmd} \Rightarrow \text{bool}$ 
  where  $\text{labels-LDCFG-nodes prog (Node } n) c = \text{labels-nodes prog } n \ c$ 
  |  $\text{labels-LDCFG-nodes prog } n \ c = \text{False}$ 

```

```

lemmas  $\text{WCFG-path-induct}[\text{consumes } 1, \text{case-names empty-path Cons-path}]$ 
  =  $\text{CFG.path.induct}[OF \text{While-CFG-aux}]$ 

```

lemma *lift-valid-node*:

```

assumes  $\text{CFG.valid-node sourcenode targetnode (valid-edge prog) } n$ 
shows  $\text{CFG.valid-node src trg}$ 
  ( $\text{lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-)}$ )
  ( $\text{Node } n$ )

```

proof –

```

from  $\langle \text{CFG.valid-node sourcenode targetnode (valid-edge prog) } n \rangle$ 
obtain  $a$  where  $\text{valid-edge prog } a$  and  $n = \text{sourcenode } a \vee n = \text{targetnode } a$ 
  by ( $\text{fastforce simp: While-CFG.valid-node-def}$ )
from  $\langle n = \text{sourcenode } a \vee n = \text{targetnode } a \rangle$ 
show  $?thesis$ 
proof
  assume  $n = \text{sourcenode } a$ 
  show  $?thesis$ 
  proof ( $\text{cases sourcenode } a = \text{Entry}$ )
    case  $\text{True}$ 
    have  $\text{lift-valid-edge (valid-edge prog) sourcenode targetnode kind Entry Exit}$ 
      ( $\text{NewEntry}, (\lambda s. \text{True}) \cdot \checkmark, \text{Node Entry}$ )
    by ( $\text{fastforce intro: lve-Entry-edge}$ )
    with  $\text{While-CFGExit-wf-aux}[of \text{prog}] \langle n = \text{sourcenode } a \rangle \text{True}$  show  $?thesis$ 
    by ( $\text{fastforce simp: CFG.valid-node-def}[OF \text{lift-CFG}]$ )
  next
    case  $\text{False}$ 
    with  $\langle \text{valid-edge prog } a \rangle \langle n = \text{sourcenode } a \vee n = \text{targetnode } a \rangle$ 
    have  $\text{lift-valid-edge (valid-edge prog) sourcenode targetnode kind Entry Exit}$ 
      ( $\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)$ )
    by ( $\text{fastforce intro: lve-edge}$ )
    with  $\text{While-CFGExit-wf-aux}[of \text{prog}] \langle n = \text{sourcenode } a \rangle$  show  $?thesis$ 
    by ( $\text{fastforce simp: CFG.valid-node-def}[OF \text{lift-CFG}]$ )

```

qed

next


```

assume  $n = \text{targetnode } a$ 
show  $?thesis$ 
proof( $\text{cases } \text{targetnode } a = \text{Exit}$ )
  case  $\text{True}$ 
    have  $\text{lift-valid-edge } (\text{valid-edge } \text{prog}) \text{ sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit}$ 
       $(\text{Node } \text{Exit}, (\lambda s. \text{True})_{\checkmark}, \text{NewExit})$ 
    by( $\text{fastforce intro:lve-Exit-edge}$ )
    with  $\text{While-CFGExit-wf-aux}[\text{of } \text{prog}] \langle n = \text{targetnode } a \rangle \text{True}$  show  $?thesis$ 
    by( $\text{fastforce simp:CFG.valid-node-def[OF lift-CFG]}$ )
  next
    case  $\text{False}$ 
    with  $\langle \text{valid-edge } \text{prog } a \rangle \langle n = \text{sourcenode } a \vee n = \text{targetnode } a \rangle$ 
    have  $\text{lift-valid-edge } (\text{valid-edge } \text{prog}) \text{ sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit}$ 
       $(\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a))$ 
    by( $\text{fastforce intro:lve-edge}$ )
    with  $\text{While-CFGExit-wf-aux}[\text{of } \text{prog}] \langle n = \text{targetnode } a \rangle$  show  $?thesis$ 
    by( $\text{fastforce simp:CFG.valid-node-def[OF lift-CFG]}$ )
  qed
qed
qed

```

lemma *lifted-CFG-fund-prop*:

```

assumes  $\text{labels-LDCFG-nodes } \text{prog } n \text{ c}$  and  $\langle c, s \rangle \rightarrow^* \langle c', s' \rangle$ 
shows  $\exists n' \text{ as. } \text{CFG.path } \text{src } \text{trg}$ 
 $(\text{lift-valid-edge } (\text{valid-edge } \text{prog}) \text{ sourcenode } \text{targetnode } \text{kind } (-\text{Entry-}) (-\text{Exit-}))$ 
 $n \text{ as } n' \wedge \text{transfers } (\text{CFG.kinds } \text{knd } \text{as}) s = s' \wedge$ 
 $\text{preds } (\text{CFG.kinds } \text{knd } \text{as}) s \wedge \text{labels-LDCFG-nodes } \text{prog } n' \text{ c'}$ 
proof –
  from  $\langle \text{labels-LDCFG-nodes } \text{prog } n \text{ c} \rangle$  obtain  $n_x$  where  $n = \text{Node } n_x$ 
  and  $\text{labels-nodes } \text{prog } n_x \text{ c}$  by( $\text{cases } n$ ) auto
  from  $\langle \text{labels-nodes } \text{prog } n_x \text{ c} \rangle \langle c, s \rangle \rightarrow^* \langle c', s' \rangle$ 
  obtain  $n'$  as where  $\text{prog } \vdash n_x -\text{as} \rightarrow^* n'$  and  $\text{transfers } (\text{CFG.kinds } \text{kind } \text{as}) s$ 
 $= s'$ 
  and  $\text{preds } (\text{CFG.kinds } \text{kind } \text{as}) s$  and  $\text{labels-nodes } \text{prog } n' \text{ c'}$ 
  by( $\text{auto dest:While-semantics-CFG-wf.fundamental-property}$ )
  from  $\langle \text{labels-nodes } \text{prog } n' \text{ c'} \rangle$  have  $\text{labels-LDCFG-nodes } \text{prog } (\text{Node } n') \text{ c'}$ 
  by simp
  from  $\langle \text{prog } \vdash n_x -\text{as} \rightarrow^* n' \rangle \langle \text{transfers } (\text{CFG.kinds } \text{kind } \text{as}) s = s' \rangle$ 
   $\langle \text{preds } (\text{CFG.kinds } \text{kind } \text{as}) s \rangle \langle n = \text{Node } n_x \rangle$ 
   $\langle \text{labels-nodes } \text{prog } n_x \text{ c} \rangle \langle \text{labels-nodes } \text{prog } n' \text{ c'} \rangle$ 
  have  $\exists \text{es. } \text{CFG.path } \text{src } \text{trg}$ 
   $(\text{lift-valid-edge } (\text{valid-edge } \text{prog}) \text{ sourcenode } \text{targetnode } \text{kind } (-\text{Entry-}) (-\text{Exit-}))$ 
   $(\text{Node } n_x) \text{ es } (\text{Node } n') \wedge \text{transfers } (\text{CFG.kinds } \text{knd } \text{es}) s = s' \wedge$ 
   $\text{preds } (\text{CFG.kinds } \text{knd } \text{es}) s$ 
  proof(induct arbitrary:n s c rule:WCFG-path-induct)
  case (empty-path  $n \text{ } n_x$ )
  from  $\langle \text{CFG.valid-node } \text{sourcenode } \text{targetnode } (\text{valid-edge } \text{prog}) \text{ } n \rangle$ 

```

```

have valid-node:CFG.valid-node src trg
  (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
  (Node n)
  by(rule lift-valid-node)
have CFG.kinds kind
  ( $\llbracket :: (w\text{-node } LDCFG\text{-node} \times state \text{ edge-kind} \times w\text{-node } LDCFG\text{-node}) \text{ list} \rrbracket =$ 
 $\llbracket$ 
    by(simp add:CFG.kinds-def[OF lift-CFG[OF While-CFGExit-wf-aux]])
with  $\langle transfers (CFG.kinds kind \llbracket \rrbracket) s = s' \rangle$   $\langle preds (CFG.kinds kind \llbracket \rrbracket) s \rangle$ 
    valid-node
show ?case
    by(fastforce intro:CFG.empty-path[OF lift-CFG[OF While-CFGExit-wf-aux]])

    simp: While-CFG.kinds-def)
next
case (Cons-path n'' as n' a nx)
note IH =  $\langle \bigwedge n s c. \llbracket transfers (CFG.kinds kind as) s = s';$ 
  preds (CFG.kinds kind as) s; n = LDCFG-node.Node n'';
  labels-nodes prog n'' c; labels-nodes prog n' c  $\rrbracket$ 
 $\implies \exists es. CFG.path src trg$ 
  (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
  (LDCFG-node.Node n'') es (LDCFG-node.Node n')  $\wedge$ 
  transfers (CFG.kinds kind es) s = s' \wedge preds (CFG.kinds kind es) s
from  $\langle transfers (CFG.kinds kind (a \# as)) s = s' \rangle$ 
have transfers (CFG.kinds kind as) (transfer (kind a) s) = s'
  by(simp add:While-CFG.kinds-def)
from  $\langle preds (CFG.kinds kind (a \# as)) s \rangle$ 
have preds (CFG.kinds kind as) (transfer (kind a) s)
  and pred (kind a) s by(simp-all add:While-CFG.kinds-def)
show ?case
proof(cases sourcenode a = (-Entry-))
  case True
  with  $\langle sourcenode a = nx \rangle$   $\langle labels-nodes prog nx c \rangle$  have False by simp
  thus ?thesis by simp
next
case False
with  $\langle valid-edge prog a \rangle$ 
have edge:lift-valid-edge (valid-edge prog) sourcenode targetnode kind
  Entry Exit (Node (sourcenode a),kind a,Node (targetnode a))
  by(fastforce intro:lve-edge)
from  $\langle prog \vdash n'' -as \rightarrow^* n' \rangle$ 
have CFG.valid-node sourcenode targetnode (valid-edge prog) n''
  by(rule While-CFG.path-valid-node)
then obtain c'' where labels-nodes prog n'' c''
proof(cases rule:While-CFGExit.valid-node-cases)
  case Entry
  with  $\langle targetnode a = n'' \rangle$   $\langle valid-edge prog a \rangle$  have False by fastforce
  thus ?thesis by simp
next

```

```

case Exit
with  $\langle \text{prog} \vdash n'' - \text{as} \rightarrow^* n' \rangle$  have  $n' = (-\text{Exit}-)$  by fastforce
with  $\langle \text{labels-nodes prog } n' \text{ } c' \rangle$  have False by fastforce
thus ?thesis by simp
next
case inner
then obtain  $l''$  where  $[simp]: n'' = (- l'' -)$  by  $(\text{cases } n'')$  auto
with  $\langle \text{valid-edge prog } a \rangle \langle \text{targetnode } a = n'' \rangle$  have  $l'' < \#:\text{prog}$ 
by  $(\text{fastforce intro: } WCFG\text{-targetlabel-less-num-nodes simp: valid-edge-def})$ 
then obtain  $c''$  where  $\text{labels prog } l'' \text{ } c''$ 
by  $(\text{fastforce dest: less-num-inner-nodes-label})$ 
with that show ?thesis by fastforce
qed
from  $IH[OF \langle \text{transfers } (CFG.kinds \text{ kind } \text{as}) (\text{transfer } (\text{kind } a) \text{ } s) = s' \rangle$ 
 $\langle \text{preds } (CFG.kinds \text{ kind } \text{as}) (\text{transfer } (\text{kind } a) \text{ } s) \rangle - \text{this}$ 
 $\langle \text{labels-nodes prog } n' \text{ } c' \rangle]$ 
obtain es where  $CFG.\text{path src trg}$ 
 $(\text{lift-valid-edge } (\text{valid-edge prog}) \text{ sourcenode targetnode kind}$ 
 $(-\text{Entry}-) (-\text{Exit}-)) (LDCFG\text{-node.Node } n'') \text{ } es (LDCFG\text{-node.Node } n')$ 
and  $\text{transfers } (CFG.kinds \text{ kind } \text{es}) (\text{transfer } (\text{kind } a) \text{ } s) = s'$ 
and  $\text{preds } (CFG.kinds \text{ kind } \text{es}) (\text{transfer } (\text{kind } a) \text{ } s)$  by blast
with  $\langle \text{targetnode } a = n'' \rangle \langle \text{sourcenode } a = nx \rangle \text{ edge}$ 
have  $\text{path: } CFG.\text{path src trg}$ 
 $(\text{lift-valid-edge } (\text{valid-edge prog}) \text{ sourcenode targetnode}$ 
 $\text{kind } (-\text{Entry}-) (-\text{Exit}-))$ 
 $(LDCFG\text{-node.Node } nx) ((\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode}$ 
 $a)) \# es)$ 
 $(LDCFG\text{-node.Node } n')$ 
by  $(\text{fastforce intro: } CFG.\text{Cons-path}[OF \text{ lift-CFG}[OF \text{ While-CFGExit-wf-aux}]])$ 
from  $\text{edge}$  have  $\text{knd } ((\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a))) =$ 
 $\text{kind } a$ 
by  $(\text{simp add: knd-def})$ 
with  $\langle \text{transfers } (CFG.kinds \text{ kind } \text{es}) (\text{transfer } (\text{kind } a) \text{ } s) = s' \rangle$ 
 $\langle \text{preds } (CFG.kinds \text{ kind } \text{es}) (\text{transfer } (\text{kind } a) \text{ } s) \rangle \langle \text{pred } (\text{kind } a) \text{ } s \rangle$ 
have  $\text{transfers}$ 
 $(CFG.kinds \text{ kind } ((\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a))) \# es)) \text{ } s$ 
 $= s'$ 
and  $\text{preds}$ 
 $(CFG.kinds \text{ kind } ((\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a))) \# es)) \text{ } s$ 
by  $(\text{auto simp: } CFG.\text{kinds-def}[OF \text{ lift-CFG}[OF \text{ While-CFGExit-wf-aux}]])$ 
with path show ?thesis by blast
qed
qed
with  $\langle n = \text{Node } nx \rangle \langle \text{labels-LDCFG-nodes prog } (\text{Node } n') \text{ } c' \rangle$ 
show ?thesis by fastforce
qed

```

```

fun final :: cmd ⇒ bool
  where final Skip = True
        | final c = False

lemma final-edge:
  labels-nodes prog n Skip ⇒ prog ⊢ n -↑id→ (-Exit-)
proof(induct prog arbitrary:n)
  case Skip
  from ⟨labels-nodes Skip n Skip⟩ have n = (- 0 -)
    by(cases n)(auto elim:labels.cases)
  thus ?case by(fastforce intro:WCFG-Skip)
next
  case (LAss V e)
  from ⟨labels-nodes (V:=e) n Skip⟩ have n = (- 1 -)
    by(cases n)(auto elim:labels.cases)
  thus ?case by(fastforce intro:WCFG-LAssSkip)
next
  case (Seq c1 c2)
  note IH2 = ⟨∧n. labels-nodes c2 n Skip ⇒ c2 ⊢ n -↑id→ (-Exit-)⟩
  from ⟨labels-nodes (c1;; c2) n Skip⟩ obtain l where n = (- l -)
    and l ≥ #:c1 and labels-nodes c2 (- l - #:c1 -) Skip
    by(cases n)(auto elim:labels.cases)
  from IH2[OF ⟨labels-nodes c2 (- l - #:c1 -) Skip⟩]
  have c2 ⊢ (- l - #:c1 -) -↑id→ (-Exit-) .
  with ⟨l ≥ #:c1⟩ have c1;;c2 ⊢ (- l - #:c1 -) ⊕ #:c1 -↑id→ (-Exit-) ⊕ #:c1
    by(fastforce intro:WCFG-SeqSecond)
  with ⟨n = (- l -)⟩ ⟨l ≥ #:c1⟩ show ?case by(simp add:id-def)
next
  case (Cond b c1 c2)
  note IH1 = ⟨∧n. labels-nodes c1 n Skip ⇒ c1 ⊢ n -↑id→ (-Exit-)⟩
  note IH2 = ⟨∧n. labels-nodes c2 n Skip ⇒ c2 ⊢ n -↑id→ (-Exit-)⟩
  from ⟨labels-nodes (if (b) c1 else c2) n Skip⟩
  obtain l where n = (- l -) and disj:(l ≥ 1 ∧ labels-nodes c1 (- l - 1 -) Skip) ∨
    (l ≥ #:c1 + 1 ∧ labels-nodes c2 (- l - #:c1 - 1 -) Skip)
    by(cases n) (fastforce elim:labels.cases)+
  from disj show ?case
proof
  assume 1 ≤ l ∧ labels-nodes c1 (- l - 1 -) Skip
  hence 1 ≤ l and labels-nodes c1 (- l - 1 -) Skip by simp-all
  from IH1[OF ⟨labels-nodes c1 (- l - 1 -) Skip⟩]
  have c1 ⊢ (- l - 1 -) -↑id→ (-Exit-) .
  with ⟨1 ≤ l⟩ have if (b) c1 else c2 ⊢ (- l - 1 -) ⊕ 1 -↑id→ (-Exit-) ⊕ 1
    by(fastforce intro:WCFG-CondThen)
  with ⟨n = (- l -)⟩ ⟨1 ≤ l⟩ show ?case by(simp add:id-def)
next
  assume #:c1 + 1 ≤ l ∧ labels-nodes c2 (- l - #:c1 - 1 -) Skip
  hence #:c1 + 1 ≤ l and labels-nodes c2 (- l - #:c1 - 1 -) Skip by simp-all
  from IH2[OF ⟨labels-nodes c2 (- l - #:c1 - 1 -) Skip⟩]

```

```

have  $c_2 \vdash (-l - \# : c_1 - 1 -) \dashv\!\!\rightarrow (-Exit-) .$ 
with  $\langle \# : c_1 + 1 \leq l \rangle$  have if (b)  $c_1$  else  $c_2 \vdash (-l - \# : c_1 - 1 -) \oplus (\# : c_1 + 1)$ 
 $\dashv\!\!\rightarrow (-Exit-) \oplus (\# : c_1 + 1)$ 
by (fastforce intro: WCFG-CondElse)
with  $\langle n = (-l -) \rangle \langle \# : c_1 + 1 \leq l \rangle$  show ?case by (simp add: id-def)
qed
next
case (While b c)
from  $\langle labels-nodes (while (b) c) n Skip \rangle$  have  $n = (-1 -)$ 
by (cases n) (auto elim: labels.cases)
thus ?case by (fastforce intro: WCFG-WhileFalseSkip)
qed

```

4.14.2 Semantic Non-Interference for Weak Order Dependence

lemmas *WODNonInterferenceGraph =*
lift-wod-backward-slice [*OF While-CFGExit-wf-aux HighLowDistinct HighLowU-NIV*]

lemma *WODNonInterference:*

```

NonInterferenceIntra src trg kno
  (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
   (-Entry-) (-Exit-))
  NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
  (lift-Use (Uses prog) (-Entry-) (-Exit-) H L id
   (CFG-wf.wod-backward-slice src trg
    (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
     (-Entry-) (-Exit-))
    (lift-Def (Defs prog) (-Entry-) (-Exit-) H L
     (lift-Use (Uses prog) (-Entry-) (-Exit-) H L
      reds (labels-LDCFG-nodes prog)
      NewExit H L (LDCFG-node.Node (-Entry-)) (LDCFG-node.Node (-Exit-)) final

```

proof –

```

interpret NonInterferenceIntraGraph src trg kno
  lift-valid-edge (valid-edge prog) sourcenode targetnode kind
  (-Entry-) (-Exit-)
  NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
  lift-Use (Uses prog) (-Entry-) (-Exit-) H L id
  CFG-wf.wod-backward-slice src trg
  (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
   (-Entry-) (-Exit-))
  (lift-Def (Defs prog) (-Entry-) (-Exit-) H L
   (lift-Use (Uses prog) (-Entry-) (-Exit-) H L
    NewExit H L LDCFG-node.Node (-Entry-) LDCFG-node.Node (-Exit-)
    by (rule WODNonInterferenceGraph)
  interpret BackwardSlice-wf src trg kno
  lift-valid-edge (valid-edge prog) sourcenode targetnode kind

```

```

    (-Entry-) (-Exit-)
  NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
  lift-Use (Uses prog) (-Entry-) (-Exit-) H L id
  CFG-wf.wod-backward-slice src trg
    (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
      (-Entry-) (-Exit-))
    (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
    (lift-Use (Uses prog) (-Entry-) (-Exit-) H L) reds labels-LDCFG-nodes prog
proof(unfold-locales)
  fix n c s c' s'
  assume labels-LDCFG-nodes prog n c and  $\langle c, s \rangle \rightarrow^* \langle c', s' \rangle$ 
  thus  $\exists n'$  as. CFG.path src trg
    (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
    n as n'  $\wedge$  transfers (CFG.kinds kno as) s = s'  $\wedge$ 
    preds (CFG.kinds kno as) s  $\wedge$  labels-LDCFG-nodes prog n' c'
    by(rule lifted-CFG-fund-prop)
qed
show ?thesis
proof(unfold-locales)
  fix c n
  assume final c and labels-LDCFG-nodes prog n c
  from  $\langle$ final c $\rangle$  have [simp]: c = Skip by(cases c) auto
  from  $\langle$ labels-LDCFG-nodes prog n c $\rangle$  obtain nx where [simp]: n = Node nx
    and labels-nodes prog nx Skip by(cases n) auto
  from  $\langle$ labels-nodes prog nx Skip $\rangle$  have prog  $\vdash$  nx  $\rightarrow$ id  $\rightarrow$  (-Exit-)
    by(rule final-edge)
  then obtain a where valid-edge prog a and sourcenode a = nx
    and kind a =  $\rightarrow$ id and targetnode a = (-Exit-)
    by(auto simp:valid-edge-def)
  with  $\langle$ labels-nodes prog nx Skip $\rangle$ 
  show  $\exists a$ . lift-valid-edge (valid-edge prog) sourcenode targetnode
    kind (-Entry-) (-Exit-) a  $\wedge$ 
    src a = n  $\wedge$  trg a = LDCFG-node.Node (-Exit-)  $\wedge$  kno a =  $\rightarrow$ id
    by(rule-tac x=(Node nx, $\rightarrow$ id,Node (-Exit-)) in exI)
    (auto intro!:lve-edge simp:kno-def valid-edge-def)
qed
qed

```

4.14.3 Semantic Non-Interference for Standard Control Dependence

```

lemma inner-node-exists: $\exists n$ . CFGExit.inner-node sourcenode targetnode
  (valid-edge prog) (-Entry-) (-Exit-) n
proof –
  have prog  $\vdash$  (-Entry-)  $\rightarrow$ ( $\lambda s$ . True) $\rightarrow$  (-0-) by(rule WCFG-Entry)
  hence CFG.valid-node sourcenode targetnode (valid-edge prog) (-0-)
    by(auto simp:While-CFG.valid-node-def valid-edge-def)
  thus ?thesis by(auto simp:While-CFGExit.inner-node-def)
qed

```

lemmas *SCDNonInterferenceGraph* =
lift-PDG-standard-backward-slice[*OF WStandardControlDependence.PDG-scd*
WhilePostdomination-aux - HighLowDistinct HighLowUNIV]

lemma *SCDNonInterference*:

NonInterferenceIntra src trg knđ
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-))
NewEntry (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
(lift-Use (Uses prog) (-Entry-) (-Exit-) H L) id
(PDG.PDG-BS src trg
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-))
(lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
(lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
(Postdomination.standard-control-dependence src trg
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-)) NewExit))
reds (labels-LDCFG-nodes prog)
NewExit H L (LDCFG-node.Node (-Entry-)) (LDCFG-node.Node (-Exit-)) final

proof –

from *inner-node-exists* **obtain** *n* **where** *CFGExit.inner-node sourcenode targetnode*

(valid-edge prog) (-Entry-) (-Exit-) n **by** *blast*

then interpret *NonInterferenceIntraGraph src trg knđ*

lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-)

NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L

lift-Use (Uses prog) (-Entry-) (-Exit-) H L id

PDG.PDG-BS src trg

(lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-))

(lift-Def (Defs prog) (-Entry-) (-Exit-) H L)

(lift-Use (Uses prog) (-Entry-) (-Exit-) H L)

(Postdomination.standard-control-dependence src trg

(lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-)) NewExit)

NewExit H L LDCFG-node.Node (-Entry-) LDCFG-node.Node (-Exit-)

by(*fastforce intro:SCDNonInterferenceGraph*)

interpret *BackwardSlice-wf src trg knđ*

lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-)

NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L

lift-Use (Uses prog) (-Entry-) (-Exit-) H L id

PDG.PDG-BS src trg

(lift-valid-edge (valid-edge prog) sourcenode targetnode kind

```

      (-Entry-) (-Exit-)
    (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
    (lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
    (Postdomination.standard-control-dependence src trg
      (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
        (-Entry-) (-Exit-)) NewExit) reds labels-LDCFG-nodes prog
  proof(unfold-locales)
    fix n c s c' s'
    assume labels-LDCFG-nodes prog n c and ⟨c,s⟩ →* ⟨c',s'⟩
    thus ∃ n' as. CFG.path src trg
      (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
      n as n' ∧ transfers (CFG.kinds kno as) s = s' ∧
      preds (CFG.kinds kno as) s ∧ labels-LDCFG-nodes prog n' c'
    by(rule lifted-CFG-fund-prop)
  qed
show ?thesis
proof(unfold-locales)
  fix c n
  assume final c and labels-LDCFG-nodes prog n c
  from ⟨final c⟩ have [simp]: c = Skip by(cases c) auto
  from ⟨labels-LDCFG-nodes prog n c⟩ obtain nx where [simp]: n = Node nx
    and labels-nodes prog nx Skip by(cases n) auto
  from ⟨labels-nodes prog nx Skip⟩ have prog ⊢ nx -↑id→ (-Exit-)
    by(rule final-edge)
  then obtain a where valid-edge prog a and sourcenode a = nx
    and kind a = ↑id and targetnode a = (-Exit-)
    by(auto simp:valid-edge-def)
  with ⟨labels-nodes prog nx Skip⟩
  show ∃ a. lift-valid-edge (valid-edge prog) sourcenode targetnode
    kind (-Entry-) (-Exit-) a ∧
    src a = n ∧ trg a = LDCFG-node.Node (-Exit-) ∧ kno a = ↑id
  by(rule-tac x=(Node nx,↑id,Node (-Exit-)) in exI)
    (auto intro!:lve-edge simp:kno-def valid-edge-def)
  qed
qed

```

4.14.4 Semantic Non-Interference for Weak Control Dependence

lemmas *WCDNonInterferenceGraph* =
lift-PDG-weak-backward-slice[*OF WWeakControlDependence.PDG-wcd*
WhileStrongPostdomination-aux - HighLowDistinct HighLowUNIV]

lemma *WCDNonInterference*:
NonInterferenceIntra src trg kno
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-))
NewEntry (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)

(lift-Use (Uses prog) (-Entry-) (-Exit-) H L) id
 (PDG.PDG-BS src trg
 (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
 (-Entry-) (-Exit-))
 (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
 (lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
 (StrongPostdomination.weak-control-dependence src trg
 (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
 (-Entry-) (-Exit-)) NewExit))
 reds (labels-LDCFG-nodes prog)
 NewExit H L (LDCFG-node.Node (-Entry-)) (LDCFG-node.Node (-Exit-)) final
proof –
from inner-node-exists **obtain** n **where** CFGEExit.inner-node sourcenode targetnode
 (valid-edge prog) (-Entry-) (-Exit-) n **by** blast
then interpret NonInterferenceIntraGraph src trg kn
 lift-valid-edge (valid-edge prog) sourcenode targetnode kind
 (-Entry-) (-Exit-)
 NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
 lift-Use (Uses prog) (-Entry-) (-Exit-) H L id
 PDG.PDG-BS src trg
 (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
 (-Entry-) (-Exit-))
 (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
 (lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
 (StrongPostdomination.weak-control-dependence src trg
 (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
 (-Entry-) (-Exit-)) NewExit)
 NewExit H L LDCFG-node.Node (-Entry-) LDCFG-node.Node (-Exit-)
by(fastforce intro: WCDNonInterferenceGraph)
interpret BackwardSlice-wf src trg kn
 lift-valid-edge (valid-edge prog) sourcenode targetnode kind
 (-Entry-) (-Exit-)
 NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
 lift-Use (Uses prog) (-Entry-) (-Exit-) H L id
 PDG.PDG-BS src trg
 (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
 (-Entry-) (-Exit-))
 (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
 (lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
 (StrongPostdomination.weak-control-dependence src trg
 (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
 (-Entry-) (-Exit-)) NewExit) reds labels-LDCFG-nodes prog
proof(unfold-locales)
fix n c s c' s'
assume labels-LDCFG-nodes prog n c **and** $\langle c, s \rangle \rightarrow^* \langle c', s' \rangle$
thus $\exists n'$ as. CFG.path src trg
 (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
 n as n' $\wedge$ transfers (CFG.kinds kn as) $s = s' \wedge$

```

    preds (CFG.kinds knd as) s  $\wedge$  labels-LDCFG-nodes prog n' c'
  by(rule lifted-CFG-fund-prop)
qed
show ?thesis
proof(unfold-locales)
  fix c n
  assume final c and labels-LDCFG-nodes prog n c
  from ⟨final c⟩ have [simp]: c = Skip by(cases c) auto
  from ⟨labels-LDCFG-nodes prog n c⟩ obtain nx where [simp]: n = Node nx
    and labels-nodes prog nx Skip by(cases n) auto
  from ⟨labels-nodes prog nx Skip⟩ have prog  $\vdash$  nx  $\rightarrow$ id  $\rightarrow$  (-Exit-)
    by(rule final-edge)
  then obtain a where valid-edge prog a and sourcenode a = nx
    and kind a =  $\rightarrow$ id and targetnode a = (-Exit-)
    by(auto simp:valid-edge-def)
  with ⟨labels-nodes prog nx Skip⟩
  show  $\exists$  a. lift-valid-edge (valid-edge prog) sourcenode targetnode
    kind (-Entry-) (-Exit-) a  $\wedge$ 
    src a = n  $\wedge$  trg a = LDCFG-node.Node (-Exit-)  $\wedge$  knd a =  $\rightarrow$ id
  by(rule-tac x=(Node nx, $\rightarrow$ id,Node (-Exit-)) in exI)
    (auto intro!:lve-edge simp:knd-def valid-edge-def)
qed
qed
end
end

```

Chapter 5

A Control Flow Graph for Jinja Byte Code

This work was done by Denis Lohner (denis.lohner@kit.edu).

5.1 Formalizing the CFG

```
theory JVMCFG imports ../Basic/BasicDefs ../Jinja/BV/BVExample begin
```

```
declare lesub-list-impl-same-size [simp del]
declare listE-length [simp del]
```

5.1.1 Type definitions

Wellformed Programs

```
definition wf-jvmprog = {(P, Phi). wf-jvm-prog_Phi P}
```

```
typedef (open) 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  $P_{wf} \equiv fst(Rep\text{-}wf\text{-}jvmprog(P))$ 
```

```
abbreviation rep-jvmprog-phi :: wf-jvmprog ⇒  $ty_P$ 
```

```
(-Φ)
```

```
  where  $P_{\Phi} \equiv snd(Rep\text{-}wf\text{-}jvmprog(P))$ 
```

lemma $wf\text{-}jvmprog\text{-}is\text{-}wf: wf\text{-}jvm\text{-}prog\ P_{\Phi} (P_{wf})$
using $Rep\text{-}wf\text{-}jvmprog$ [of P]
by ($auto simp: wf\text{-}jvmprog\text{-}def split\text{-}beta$)

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 $jvmprog = wf\text{-}jvmprog \times cname \times mname$
type-synonym $callstack = (cname \times mname \times pc) list$

The state is modeled as $heap \times stack\text{-}variables \times local\text{-}variables$
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 $state = heap \times ((nat \times nat) \Rightarrow val) \times ((nat \times nat) \Rightarrow val)$

abbreviation $heap\text{-}of :: state \Rightarrow heap$
where
 $heap\text{-}of\ s \equiv fst(s)$

abbreviation $stk\text{-}of :: state \Rightarrow ((nat \times nat) \Rightarrow val)$
where
 $stk\text{-}of\ s \equiv fst(snd(s))$

abbreviation $loc\text{-}of :: state \Rightarrow ((nat \times nat) \Rightarrow val)$
where
 $loc\text{-}of\ s \equiv snd(snd(s))$

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 $exec\text{-}instr :: instr \Rightarrow wf\text{-}jvmprog \Rightarrow state \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow state$

where

exec-instr-Load:

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

| *exec-instr-Store*:

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

| *exec-instr-Push*:

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

| *exec-instr-New*:

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

| *exec-instr-Getfield*:

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

| *exec-instr-Putfield*:

exec-instr (*Putfield F C*) *P s callddepth stk-length rs ill* =
 (let (*h,stk,loc*) = *s*;
 v = *stk* (*callddepth,stk-length* - 1);
 a = *the-Addr* (*stk* (*callddepth,stk-length* - 2));
 (*D,fs*) = *the*(*h a*)
 in (*h*(*a* $\mapsto$ (*D,fs*((*F,C*) $\mapsto$ *v*))), *stk, loc*))

| *exec-instr-Checkcast*:

exec-instr (*Checkcast C*) *P s callddepth stk-length rs ill* = *s*

| *exec-instr-Pop*:

exec-instr (*Pop*) *P s callddepth stk-length rs ill* = *s*

| *exec-instr-IAAdd*:

exec-instr (*IAAdd*) *P s callddepth stk-length rs ill* =
 (let (*h,stk,loc*) = *s*;
 *i*₁ = *the-Intg* (*stk* (*callddepth, stk-length* - 1));
 *i*₂ = *the-Intg* (*stk* (*callddepth, stk-length* - 2))
 in (*h, stk*((*callddepth, stk-length* - 2) := *Intg* (*i*₁ + *i*₂)), *loc*))

| *exec-instr-IfFalse*:
exec-instr (*IfFalse* *b*) *P s callddepth stk-length rs ill* = *s*

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

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

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

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

(if (*b* ≤ *n*) then *stk*(*callddepth*, *stk-length* - (*Suc n* - *b*)) else

arbitrary))
 in (*h,stk,loc'*))

| *exec-instr-Return*:
exec-instr (*Return*) *P s callddepth stk-length ret-stk-length ill* =
 (if (*callddepth* = 0)
 then *s*
 else
 (let (*h,stk,loc*) = *s*;
 v = *stk*(*callddepth*, *stk-length* - 1)
 in (*h,stk*((*callddepth* - 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 $[f\ 0, f\ 1, f\ 2, \dots, f\ (n - 1)]$.

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 $[f\ (n - 1), \dots, f\ 1, f\ 0]$.

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 $locss :: wf-jvmprog \Rightarrow callstack \Rightarrow ((nat \times nat) \Rightarrow 'a) \Rightarrow 'a\ list\ list$
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 $stkss :: wf-jvmprog \Rightarrow callstack \Rightarrow ((nat \times nat) \Rightarrow 'a) \Rightarrow 'a\ list\ list$
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 $state-to-jvm-state :: wf-jvmprog \Rightarrow callstack \Rightarrow state \Rightarrow jvm-state$
where $state-to-jvm-state\ P\ cs\ s \equiv$
 $(None,\ heap-of\ s,\ zip\ (stkss\ P\ cs\ (stk-of\ s))\ (zip\ (locss\ P\ cs\ (loc-of\ s))\ cs))$

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

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

State Conformance

Now we lift byte code verifier conformance to our state representation

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

Statically determine catch-block

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

```
fun find-handler-for :: wf-jvmprog  $\Rightarrow$  cname  $\Rightarrow$  callstack  $\Rightarrow$  callstack
where
  find-handler-for P C [] = []
| find-handler-for P C (c#cs) = (let (C',M',pc') = c in
  (case match-ex-table (Pwf) C pc' (ex-table-of (Pwf) C' M') of
    None  $\Rightarrow$  find-handler-for P C cs
  | Some pc-d  $\Rightarrow$  (C', M', fst pc-d)#cs))
```

5.1.3 Simplification lemmas

```
lemma find-handler-decr [simp]: find-handler-for P Exc cs  $\neq$  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)) ]
   $\implies$  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)) ]
   $\implies$  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]:  $n \neq 0 \implies \text{hd } (\text{stks } n \text{ stk}) = \text{stk}(n - 1)$ 
  by (cases n, simp-all)

lemma hd-tl-stks:  $n > 1 \implies \text{hd } (\text{tl } (\text{stks } n \text{ stk})) = \text{stk}(n - 2)$ 
  by (cases n, auto)

lemma stkss-purge:
   $\text{length } cs \leq a \implies \text{stkss } P \text{ cs } (\text{stk}((a,b) := c)) = \text{stkss } P \text{ cs } \text{stk}$ 
  by (induct cs, auto)

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

lemma locss-purge:
   $\text{length } cs \leq a \implies \text{locss } P \text{ cs } (\text{loc}((a,b) := c)) = \text{locss } P \text{ cs } \text{loc}$ 
  by (induct cs, auto)

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

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

```

```

lemma stks-pullout:
   $n < b \implies \text{stks } b \ (\text{stk}(n := e)) = \text{stks } b \ \text{stk} \ [b - \text{Suc } n := e]$ 
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 :  $xs \neq [] \implies \text{tl } xs ! n = xs ! (\text{Suc } n)$ 
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-) -(λs. False)√→ (-Exit-)

| *JCFG-EntryStart*:
prog = (*P*, *C0*, *Main*) ⇒ *prog* ⊢ (-Entry-) -(λs. True)√→ (- [(*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*:

$$\begin{aligned}
& \llbracket \text{prog} = (P, C0, \text{Main}); \\
& \text{instrs-of } (P_{wf}) \ C \ M \ ! \ pc \in \{\text{Load idx}, \text{Store idx}, \text{Push val}, \text{Pop}, \text{IAdd}, \text{CmpEq}\}; \\
& ek = \uparrow(\lambda s. \text{exec-instr } ((\text{instrs-of } (P_{wf}) \ C \ M) \ ! \ pc) \ P \ s \\
& \quad (\text{length } cs) \ (\text{stkLength } P \ C \ M \ pc) \ \text{arbitrary arbitrary}) \rrbracket \\
& \implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None } -) -ek \rightarrow (- (C, M, \text{Suc } pc) \# cs, \text{None } -)
\end{aligned}$$

| *JCFG-New-Normal-Pred*:

$$\begin{aligned}
& \llbracket \text{prog} = (P, C0, \text{Main}); \\
& (\text{instrs-of } (P_{wf}) \ C \ M) \ ! \ pc = (\text{New } Cl); \\
& ek = (\lambda(h, \text{stk}, \text{loc}). \text{new-Addr } h \neq \text{None})_{\checkmark} \rrbracket \\
& \implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None } -) -ek \rightarrow (- (C, M, pc) \# cs, [((C, M, \text{Suc } pc) \# cs, \text{False})] -)
\end{aligned}$$

| *JCFG-New-Normal-Update*:

$$\begin{aligned}
& \llbracket \text{prog} = (P, C0, \text{Main}); \\
& (\text{instrs-of } (P_{wf}) \ C \ M) \ ! \ pc = (\text{New } Cl); \\
& ek = \uparrow(\lambda s. \text{exec-instr } (\text{New } Cl) \ P \ s \ (\text{length } cs) \ (\text{stkLength } P \ C \ M \ pc) \ \text{arbitrary arbitrary}) \rrbracket \\
& \implies \text{prog} \vdash (- (C, M, pc) \# cs, [((C, M, \text{Suc } pc) \# cs, \text{False})] -) -ek \rightarrow (- (C, M, \text{Suc } pc) \# cs, \text{None } -)
\end{aligned}$$

| *JCFG-New-Exc-Pred*:

$$\begin{aligned}
& \llbracket \text{prog} = (P, C0, \text{Main}); \\
& (\text{instrs-of } (P_{wf}) \ C \ M) \ ! \ pc = (\text{New } Cl); \\
& \text{find-handler-for } P \ \text{OutOfMemory } ((C, M, pc) \# cs) = cs'; \\
& ek = (\lambda(h, \text{stk}, \text{loc}). \text{new-Addr } h = \text{None})_{\checkmark} \rrbracket \\
& \implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None } -) -ek \rightarrow (- (C, M, pc) \# cs, [(cs', \text{True})] -)
\end{aligned}$$

| *JCFG-New-Exc-Update*:

$$\begin{aligned}
& \llbracket \text{prog} = (P, C0, \text{Main}); \\
& (\text{instrs-of } (P_{wf}) \ C \ M) \ ! \ pc = (\text{New } Cl); \\
& \text{find-handler-for } P \ \text{OutOfMemory } ((C, M, pc) \# cs) = (C', M', pc') \# cs'; \\
& ek = \uparrow(\lambda(h, \text{stk}, \text{loc}). \\
& \quad (h, \\
& \quad \quad \text{stk}((\text{length } cs', (\text{stkLength } P \ C' \ M' \ pc') - 1) := \text{Addr } (\text{addr-of-sys-xcpt } \\
& \quad \quad \text{OutOfMemory})), \\
& \quad \quad \text{loc}) \\
& \quad) \rrbracket \\
& \implies \text{prog} \vdash (- (C, M, pc) \# cs, [((C', M', pc') \# cs', \text{True})] -) -ek \rightarrow (- (C', M', pc') \# cs', \text{None } -)
\end{aligned}$$

| *JCFG-New-Exc-Exit*:

$$\begin{aligned}
& \llbracket \text{prog} = (P, C0, \text{Main}); \\
& (\text{instrs-of } (P_{wf}) \ C \ M) \ ! \ pc = (\text{New } Cl); \\
& \text{find-handler-for } P \ \text{OutOfMemory } ((C, M, pc) \# cs) = [] \rrbracket \\
& \implies \text{prog} \vdash (- (C, M, pc) \# cs, [([], \text{True})] -) -\uparrow id \rightarrow (-\text{Exit}-)
\end{aligned}$$

| *JCFG-Getfield-Normal-Pred*:

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = (\text{Getfield Fd Cl});$
 $ek = (\lambda(h, stk, loc). \text{stk}(\text{length } cs, \text{stkLength } P C M pc - 1) \neq \text{Null})_{\checkmark} \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None } -) -ek \rightarrow (- (C, M, pc) \# cs, \llbracket (C, M, \text{Suc } pc) \# cs, \text{False} \rrbracket -)$

| JCFG-Getfield-Normal-Update:

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = (\text{Getfield Fd Cl});$
 $ek = \uparrow(\lambda s. \text{exec-instr } (\text{Getfield Fd Cl}) P s (\text{length } cs) (\text{stkLength } P C M pc)$
 $\text{arbitrary arbitrary}) \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, \llbracket (C, M, \text{Suc } pc) \# cs, \text{False} \rrbracket -) -ek \rightarrow (- (C,$
 $M, \text{Suc } pc) \# cs, \text{None } -)$

| JCFG-Getfield-Exc-Pred:

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = (\text{Getfield Fd Cl});$
 $\text{find-handler-for } P \text{ NullPointer } ((C, M, pc) \# cs) = cs';$
 $ek = (\lambda(h, stk, loc). \text{stk}(\text{length } cs, \text{stkLength } P C M pc - 1) = \text{Null})_{\checkmark} \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None } -) -ek \rightarrow (- (C, M, pc) \# cs, \llbracket cs', \text{True} \rrbracket -)$

| JCFG-Getfield-Exc-Update:

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = (\text{Getfield Fd Cl});$
 $\text{find-handler-for } P \text{ NullPointer } ((C, M, pc) \# cs) = (C', M', pc') \# cs';$
 $ek = \uparrow(\lambda(h, stk, loc).$
 $(h,$
 $\text{stk}((\text{length } cs', (\text{stkLength } P C' M' pc') - 1) := \text{Addr } (\text{addr-of-sys-xcpt}$
 $\text{NullPointer})),$
 $\text{loc})$
 $\rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, \llbracket (C', M', pc') \# cs', \text{True} \rrbracket -) -ek \rightarrow (- (C', M',$
 $pc') \# cs', \text{None } -)$

| JCFG-Getfield-Exc-Exit:

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = (\text{Getfield Fd Cl});$
 $\text{find-handler-for } P \text{ NullPointer } ((C, M, pc) \# cs) = [] \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, \llbracket [], \text{True} \rrbracket -) -\uparrow id \rightarrow (-\text{Exit}-)$

| JCFG-Putfield-Normal-Pred:

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = (\text{Putfield Fd Cl});$
 $ek = (\lambda(h, stk, loc). \text{stk}(\text{length } cs, \text{stkLength } P C M pc - 2) \neq \text{Null})_{\checkmark} \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None } -) -ek \rightarrow (- (C, M, pc) \# cs, \llbracket (C, M, \text{Suc } pc) \# cs, \text{False} \rrbracket -)$

| JCFG-Putfield-Normal-Update:

$\llbracket \text{prog} = (P, C0, \text{Main});$

$(instrs\text{-}of\ (P_{wf})\ C\ M)!\ pc = (Putfield\ Fd\ Cl);$
 $ek = \uparrow(\lambda s. exec\text{-}instr\ (Putfield\ Fd\ Cl)\ P\ s\ (length\ cs)\ (stkLength\ P\ C\ M\ pc)$
 $\quad\quad\quad arbitrary\ arbitrary)\ \mathbb{I}$
 $\implies prog \vdash (-\ (C, M, pc)\#cs, [((C, M, Suc\ pc)\#cs, False)]\ -) -ek \rightarrow (-\ (C,$
 $M, Suc\ pc)\#cs, None\ -)$

$| JCFG\text{-}Putfield\text{-}Exc\text{-}Pred:$
 $\mathbb{I}\ prog = (P, C0, Main);$
 $(instrs\text{-}of\ (P_{wf})\ C\ M)!\ pc = (Putfield\ Fd\ Cl);$
 $find\text{-}handler\text{-}for\ P\ NullPointer\ ((C, M, pc)\#cs) = cs';$
 $ek = (\lambda(h, stk, loc).\ stk(length\ cs, stkLength\ P\ C\ M\ pc - 2) = Null)_{\checkmark}\ \mathbb{I}$
 $\implies prog \vdash (-\ (C, M, pc)\#cs, None\ -) -ek \rightarrow (-\ (C, M, pc)\#cs, [(cs', True)]\ -)$

$| JCFG\text{-}Putfield\text{-}Exc\text{-}Update:$
 $\mathbb{I}\ prog = (P, C0, Main);$
 $(instrs\text{-}of\ (P_{wf})\ C\ M)!\ pc = (Putfield\ Fd\ Cl);$
 $find\text{-}handler\text{-}for\ P\ NullPointer\ ((C, M, pc)\#cs) = (C', M', pc')\#cs';$
 $ek = \uparrow(\lambda(h, stk, loc).$
 $\quad(h,$
 $\quad\quad\quad stk((length\ cs', (stkLength\ P\ C'\ M'\ pc') - 1) := Addr\ (addr\text{-}of\text{-}sys\text{-}xcp$
 $NullPointer))),$
 $\quad\quad\quad loc)$
 $\quad\quad\quad)\ \mathbb{I}$
 $\implies prog \vdash (-\ (C, M, pc)\#cs, [((C', M', pc')\#cs', True)]\ -) -ek \rightarrow (-\ (C', M',$
 $pc')\#cs', None\ -)$

$| JCFG\text{-}Putfield\text{-}Exc\text{-}Exit:$
 $\mathbb{I}\ prog = (P, C0, Main);$
 $(instrs\text{-}of\ (P_{wf})\ C\ M)!\ pc = (Putfield\ Fd\ Cl);$
 $find\text{-}handler\text{-}for\ P\ NullPointer\ ((C, M, pc)\#cs) = []\ \mathbb{I}$
 $\implies prog \vdash (-\ (C, M, pc)\#cs, [([], True)]\ -) -\uparrow id \rightarrow (-Exit\text{-})$

$| JCFG\text{-}Checkcast\text{-}Normal\text{-}Pred:$
 $\mathbb{I}\ prog = (P, C0, Main);$
 $(instrs\text{-}of\ (P_{wf})\ C\ M)!\ pc = (Checkcast\ Cl);$
 $ek = (\lambda(h, stk, loc).\ cast\text{-}ok\ (P_{wf})\ Cl\ h\ (stk(length\ cs, stkLength\ P\ C\ M\ pc -$
 $Suc\ 0)))_{\checkmark}\ \mathbb{I}$
 $\implies prog \vdash (-\ (C, M, pc)\#cs, None\ -) -ek \rightarrow (-\ (C, M, Suc\ pc)\#cs, None\ -)$

$| JCFG\text{-}Checkcast\text{-}Exc\text{-}Pred:$
 $\mathbb{I}\ prog = (P, C0, Main);$
 $(instrs\text{-}of\ (P_{wf})\ C\ M)!\ pc = (Checkcast\ Cl);$
 $find\text{-}handler\text{-}for\ P\ ClassCast\ ((C, M, pc)\#cs) = cs';$
 $ek = (\lambda(h, stk, loc).\ \neg\ cast\text{-}ok\ (P_{wf})\ Cl\ h\ (stk(length\ cs, stkLength\ P\ C\ M\ pc -$
 $Suc\ 0)))_{\checkmark}\ \mathbb{I}$
 $\implies prog \vdash (-\ (C, M, pc)\#cs, None\ -) -ek \rightarrow (-\ (C, M, pc)\#cs, [(cs', True)]\ -)$

$| JCFG\text{-}Checkcast\text{-}Exc\text{-}Update:$
 $\mathbb{I}\ prog = (P, C0, Main);$

$(instrs\text{-}of\ (P_{wf})\ C\ M) !\ pc = (Checkcast\ Cl);$
 $find\text{-}handler\text{-}for\ P\ ClassCast\ ((C, M, pc)\#cs) = (C', M', pc')\#cs';$
 $ek = \uparrow(\lambda(h, stk, loc).$
 $(h,$
 $stk((length\ cs', (stkLength\ P\ C'\ M'\ pc') - 1) := Addr\ (addr\text{-}of\ sys\text{-}xcp$
 $ClassCast)),$
 $loc)$
 $)\]$
 $\implies prog \vdash (-\ (C, M, pc)\#cs, [(C', M', pc')\#cs', True]) - ek \rightarrow (-\ (C', M',$
 $pc')\#cs', None -)$

$| JCFG\text{-}Checkcast\text{-}Exc\text{-}Exit:$
 $\llbracket prog = (P, C0, Main);$
 $(instrs\text{-}of\ (P_{wf})\ C\ M) !\ pc = (Checkcast\ Cl);$
 $find\text{-}handler\text{-}for\ P\ ClassCast\ ((C, M, pc)\#cs) = []\]$
 $\implies prog \vdash (-\ (C, M, pc)\#cs, [([], True)]) - \uparrow id \rightarrow (-Exit-)$

$| JCFG\text{-}Invoke\text{-}Normal\text{-}Pred:$
 $\llbracket prog = (P, C0, Main);$
 $(instrs\text{-}of\ (P_{wf})\ C\ M) !\ pc = (Invoke\ M2\ n);$
 $cd = length\ cs;$
 $stk\text{-}length = stkLength\ P\ C\ M\ pc;$
 $ek = (\lambda(h, stk, loc).$
 $stk(cd, stk\text{-}length - Suc\ n) \neq Null \wedge$
 $fst(method\ (P_{wf})\ (cname\text{-}of\ h\ (the\ Addr(stk(cd, stk\text{-}length - Suc\ n))))\ M2)$
 $= D$
 $)_{\checkmark}\]$
 $\implies$
 $prog \vdash (-\ (C, M, pc)\#cs, None -) - ek \rightarrow (-\ (C, M, pc)\#cs, [(D, M2, 0)\#(C,$
 $M, pc)\#cs, False]) -)$

$| JCFG\text{-}Invoke\text{-}Normal\text{-}Update:$
 $\llbracket prog = (P, C0, Main);$
 $(instrs\text{-}of\ (P_{wf})\ C\ M) !\ pc = (Invoke\ M2\ n);$
 $stk\text{-}length = stkLength\ P\ C\ M\ pc;$
 $loc\text{-}length = locLength\ P\ D\ M2\ 0;$
 $ek = \uparrow(\lambda s. exec\text{-}instr\ (Invoke\ M2\ n)\ P\ s\ (length\ cs)\ stk\text{-}length\ arbitrary$
 $loc\text{-}length)$
 $\]$
 $\implies prog \vdash (-\ (C, M, pc)\#cs, [(D, M2, 0)\#(C, M, pc)\#cs, False]) - ek \rightarrow$
 $(-\ (D, M2, 0)\#(C, M, pc)\#cs, None -)$

$| JCFG\text{-}Invoke\text{-}Exc\text{-}Pred:$
 $\llbracket prog = (P, C0, Main);$
 $(instrs\text{-}of\ (P_{wf})\ C\ M) !\ pc = (Invoke\ m2\ n);$
 $find\text{-}handler\text{-}for\ P\ NullPointer\ ((C, M, pc)\#cs) = cs';$
 $ek = (\lambda(h, stk, loc). stk(length\ cs, stkLength\ P\ C\ M\ pc - Suc\ n) = Null)_{\checkmark}\]$
 $\implies prog \vdash (-\ (C, M, pc)\#cs, None -) - ek \rightarrow (-\ (C, M, pc)\#cs, [(cs', True)]) -)$

| *JCFG-Invoke-Exc-Update*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad (\text{instrs-of } (P_{wf}) \ C \ M) ! \text{pc} = (\text{Invoke } M2 \ n); \\ & \quad \text{find-handler-for } P \ \text{NullPointer} \ ((C, M, \text{pc})\#cs) = (C', M', \text{pc}')\#cs'; \\ & \quad \text{ek} = \uparrow(\lambda(h, \text{stk}, \text{loc}). \\ & \quad \quad (h, \\ & \quad \quad \quad \text{stk}((\text{length } cs', (\text{stkLength } P \ C' \ M' \ \text{pc}') - 1) := \text{Addr } (\text{addr-of-sys-xcpt} \\ & \quad \quad \quad \text{NullPointer})), \\ & \quad \quad \quad \text{loc}) \\ & \quad \quad) \\ & \quad \rrbracket \\ & \implies \text{prog} \vdash (- (C, M, \text{pc})\#cs, [((C', M', \text{pc}')\#cs', \text{True})] \ -) - \text{ek} \rightarrow (- (C', M', \\ & \text{pc}')\#cs', \text{None} \ -) \end{aligned}$$

| *JCFG-Invoke-Exc-Exit*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad (\text{instrs-of } (P_{wf}) \ C \ M) ! \text{pc} = (\text{Invoke } M2 \ n); \\ & \quad \text{find-handler-for } P \ \text{NullPointer} \ ((C, M, \text{pc})\#cs) = [] \rrbracket \\ & \implies \text{prog} \vdash (- (C, M, \text{pc})\#cs, [([], \text{True})] \ -) - \uparrow id \rightarrow (- \text{Exit-}) \end{aligned}$$

| *JCFG-Return-Update*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad (\text{instrs-of } (P_{wf}) \ C \ M) ! \text{pc} = \text{Return}; \\ & \quad \text{stk-length} = \text{stkLength } P \ C \ M \ \text{pc}; \\ & \quad r\text{-stk-length} = \text{stkLength } P \ C' \ M' \ (\text{Suc } \text{pc}'); \\ & \quad \text{ek} = \uparrow(\lambda s. \text{exec-instr } \text{Return } P \ s \ (\text{Suc } (\text{length } cs)) \ \text{stk-length } r\text{-stk-length } \text{arbitrary}) \rrbracket \\ & \implies \text{prog} \vdash (- (C, M, \text{pc})\#(C', M', \text{pc}')\#cs, \text{None} \ -) - \text{ek} \rightarrow (- (C', M', \text{Suc} \\ & \text{pc}')\#cs, \text{None} \ -) \end{aligned}$$

| *JCFG-Goto-Update*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad (\text{instrs-of } (P_{wf}) \ C \ M) ! \text{pc} = \text{Goto } \text{idx} \rrbracket \\ & \implies \text{prog} \vdash (- (C, M, \text{pc})\#cs, \text{None} \ -) - \uparrow id \rightarrow (- (C, M, \text{nat } (\text{int } \text{pc} + \\ & \text{idx}))\#cs, \text{None} \ -) \end{aligned}$$

| *JCFG-IfFalse-False*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad (\text{instrs-of } (P_{wf}) \ C \ M) ! \text{pc} = (\text{IfFalse } b); \\ & \quad b \neq 1; \\ & \quad \text{ek} = (\lambda(h, \text{stk}, \text{loc}). \text{stk}(\text{length } cs, \text{stkLength } P \ C \ M \ \text{pc} - 1) = \text{Bool False})_{\checkmark} \rrbracket \\ & \implies \text{prog} \vdash (- (C, M, \text{pc})\#cs, \text{None} \ -) - \text{ek} \rightarrow (- (C, M, \text{nat } (\text{int } \text{pc} + b))\#cs, \text{None} \\ & \ -) \end{aligned}$$

| *JCFG-IfFalse-Next*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad (\text{instrs-of } (P_{wf}) \ C \ M) ! \text{pc} = (\text{IfFalse } b); \\ & \quad \text{ek} = (\lambda(h, \text{stk}, \text{loc}). \text{stk}(\text{length } cs, \text{stkLength } P \ C \ M \ \text{pc} - 1) \neq \text{Bool False} \vee b \\ & \quad = 1)_{\checkmark} \rrbracket \end{aligned}$$

$\implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None} -) -ek \rightarrow (- (C, M, \text{Suc } pc) \# cs, \text{None} -)$

| JCFG-Throw-Pred:
 $\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = \text{Throw};$
 $cd = \text{length } cs;$
 $\text{stk-length} = \text{stkLength } P C M pc;$
 $\exists \text{Exc. find-handler-for } P \text{ Exc } ((C, M, pc) \# cs) = cs';$
 $ek = (\lambda(h, \text{stk}, \text{loc}).$
 $(\text{stk}(\text{length } cs, \text{stkLength } P C M pc - 1) = \text{Null} \wedge$
 $\text{find-handler-for } P \text{ NullPointer } ((C, M, pc) \# cs) = cs') \vee$
 $(\text{stk}(\text{length } cs, \text{stkLength } P C M pc - 1) \neq \text{Null} \wedge$
 $\text{find-handler-for } P (\text{cname-of } h (\text{the-Addr}(\text{stk}(cd, \text{stk-length} - 1)))) ((C,$
 $M, pc) \# cs) = cs')$
 $\rangle \vee \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None} -) -ek \rightarrow (- (C, M, pc) \# cs, [(cs', \text{True})] -)$

| JCFG-Throw-Update:
 $\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = \text{Throw};$
 $ek = \uparrow(\lambda(h, \text{stk}, \text{loc}).$
 $(h,$
 $\text{stk}((\text{length } cs', (\text{stkLength } P C' M' pc') - 1) :=$
 $\text{if } (\text{stk}(\text{length } cs, \text{stkLength } P C M pc - 1) = \text{Null}) \text{ then}$
 $\text{Addr } (\text{addr-of-sys-xcpt } \text{NullPointer})$
 $\text{else } (\text{stk}(\text{length } cs, \text{stkLength } P C M pc - 1))),$
 $\text{loc})$
 $\rangle \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, [(C', M', pc') \# cs', \text{True}] -) -ek \rightarrow (- (C', M',$
 $pc') \# cs', \text{None} -)$

| JCFG-Throw-Exit:
 $\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = \text{Throw} \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, [([], \text{True})] -) -\uparrow id \rightarrow (-\text{Exit}-)$

5.1.7 CFG properties

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

assumes $\text{edge:prog} \vdash (-\text{Exit}-) -et \rightarrow n'$

shows *False*

proof –

{ **fix** n

have $\llbracket \text{prog} \vdash n -et \rightarrow n'; n = (-\text{Exit}-) \rrbracket \implies \text{False}$

by $(\text{auto elim! : JVM-CFG.cases})$

}

with *edge* **show** *?thesis* **by** *fastforce*

qed

```

lemma JVMCFG-Entry-no-targetnode [dest]:
  assumes edge:prog  $\vdash n -et\rightarrow (-Entry-)$ 
  shows False
proof -
  { fix n' have  $\llbracket prog \vdash n -et\rightarrow n'; n' = (-Entry-) \rrbracket \implies False$ 
    by (auto elim!: JVM-CFG.cases)
  }
  with edge show ?thesis by fastforce
qed

lemma JVMCFG-EntryD:
   $\llbracket (P,C,M) \vdash n -et\rightarrow n'; n = (-Entry-) \rrbracket$ 
   $\implies (n' = (-Exit-) \wedge et = (\lambda s. False)_{\checkmark}) \vee (n' = (-[(C,M,0)], None -) \wedge et =$ 
   $(\lambda s. True)_{\checkmark})$ 
by (erule JVM-CFG.cases) simp-all

declare split-def [simp add]
declare find-handler-for.simps [simp del]

lemma JVMCFG-edge-det:
   $\llbracket prog \vdash n -et\rightarrow n'; prog \vdash n -et'\rightarrow n' \rrbracket \implies et = et'$ 
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  $\Rightarrow$  j-node
  where sourcenode e  $\equiv$  fst e

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

abbreviation kind :: j-edge  $\Rightarrow$  state edge-kind
  where kind e  $\equiv$  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  $\Rightarrow$  callstack  $\Rightarrow$  bool
where

```

```

    valid-callstack prog [] = True
| valid-callstack (P, C0, Main) [(C, M, pc)]  $\longleftrightarrow$ 
    C = C0  $\wedge$  M = Main  $\wedge$ 
    (PΦ) C M ! pc  $\neq$  None  $\wedge$ 
    ( $\exists T Ts \text{ m}\acute{x}s \text{ m}\acute{x}l \text{ is } xt. (P_{wf}) \vdash C \text{ sees } M:Ts \rightarrow T = (\text{m}\acute{x}s, \text{m}\acute{x}l, \text{is}, xt) \text{ in } C \wedge pc$ 
    < length is)
| valid-callstack (P, C0, Main) ((C, M, pc)#(C', M', pc')#cs)  $\longleftrightarrow$ 
    instrs-of (Pwf) C' M' ! pc' =
    Invoke M (locLength P C M 0 - Suc (fst(snd(snd(snd(snd(method (Pwf) C
    M)))))) )  $\wedge$ 
    (PΦ) C M ! pc  $\neq$  None  $\wedge$ 
    ( $\exists T Ts \text{ m}\acute{x}s \text{ m}\acute{x}l \text{ is } xt. (P_{wf}) \vdash C \text{ sees } M:Ts \rightarrow T = (\text{m}\acute{x}s, \text{m}\acute{x}l, \text{is}, xt) \text{ in } C \wedge pc$ 
    < length is)  $\wedge$ 
    valid-callstack (P, C0, Main) ((C', M', pc')#cs)

```

fun valid-node :: jmpprog $\Rightarrow$ j-node $\Rightarrow$ bool

where

valid-node prog (-Entry-) = True

```

| valid-node prog (- cs, None -)  $\longleftrightarrow$  valid-callstack prog cs
| valid-node prog (- cs, [(cs', xf)] -)  $\longleftrightarrow$ 
    valid-callstack prog cs  $\wedge$  valid-callstack prog cs'  $\wedge$ 
    ( $\exists Q. \text{prog} \vdash (- cs, None -) - (Q)_{\checkmark} \rightarrow (- cs, [(cs', xf)] -) \wedge$ 
    ( $\exists f. \text{prog} \vdash (- cs, [(cs', xf)] -) - \uparrow f \rightarrow (- cs', None -)$ )

```

fun valid-edge :: jmpprog $\Rightarrow$ j-edge $\Rightarrow$ bool

where

```

    valid-edge prog a  $\longleftrightarrow$ 
    (prog  $\vdash$  (sourcenode a) - (kind a)  $\rightarrow$  (targetnode a))
     $\wedge$  (valid-node prog (sourcenode a))
     $\wedge$  (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 $\vdash$ n - et $\rightarrow$ (-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-) -(λs. False)√→ (-Exit-)
    by (rule JCFG-EntryExit)
  thus ∃ a. valid-edge prog a ∧ sourcenode a = (-Entry-) ∧
    targetnode a = (-Exit-) ∧ kind a = (λs. False)√
    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* :: *typ*
 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:  $i < \text{Suc } (\text{Suc } 0) \implies i = 0 \vee i = 1$ 
    by auto

lemma EP-wf: wf-jvm-prog  $\text{Phi-EP EP}$ 
    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:
         $\forall c \in \text{set EP.}$ 
        wf-cdecl
         $(\lambda P C (M, Ts, T_r, mxs, mxl_0, is, xt). \text{wt-method } P C Ts T_r mxs mxl_0 \text{ is}$ 
         $xt (\text{Phi-EP } C M))$ 
        EP c
    proof
    fix C
    assume C-in-EP:  $C \in \text{set EP}$ 
    show wf-cdecl
         $(\lambda P C (M, Ts, T_r, mxs, mxl_0, is, xt). \text{wt-method } P C Ts T_r mxs mxl_0 \text{ is}$ 
         $xt (\text{Phi-EP } C M))$ 
        EP C
    proof (cases  $C \in \text{set SystemClasses}$ )
    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]:  $C = ("C", \text{the } (\text{class EP } "C"))$ 
    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  $(\forall c \in \text{set } EP.$ 
  wf-cdecl
   $(\lambda P C (M, Ts, T_r, mxs, mxl_0, is, xt). \text{wt-method } P C Ts T_r mxs mxl_0 is xt$ 
   $(\text{Phi-EP } C M))$ 
   $EP c) \wedge$ 
  distinct-fst EP
  by simp
qed

lemma [simp]: Abs-wf-jvmprog (EP, Phi-EP)wf = EP
proof (cases (EP, Phi-EP)  $\in$  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)Φ = Phi-EP
proof (cases (EP, Phi-EP)  $\in$  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*:

$EP \vdash C \text{ sees } M: Ts \rightarrow T = (m\mathit{xs}, m\mathit{xl}, is, xt) \text{ in } D$
 $\implies C = "C" \wedge$
 $M = "M" \wedge$
 $Ts = [] \wedge$
 $T = \text{Void} \wedge$
 $m\mathit{xs} = 1 \wedge$
 $m\mathit{xl} = 0 \wedge$
 $is = [\text{Push Unit}, \text{Return}] \wedge$
 $xt = [] \wedge$
 $D = "C"$

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 split-if-eq1) $+$

lemma [*simp*]:

$\exists T Ts m\mathit{xs} m\mathit{xl} is. (\exists xt. EP \vdash "C" \text{ sees } "M": Ts \rightarrow T = (m\mathit{xs}, m\mathit{xl}, is, xt) \text{ in } "C") \wedge is \neq []$

using *EP-wf*

by (*fastforce dest: mdecl-visible simp: wf-jvm-prog-phi-def EP-def*)

lemma [*simp*]:

$\exists T Ts m\mathit{xs} m\mathit{xl} is. (\exists xt. EP \vdash "C" \text{ sees } "M": Ts \rightarrow T = (m\mathit{xs}, m\mathit{xl}, is, xt) \text{ in } "C") \wedge$

$\text{Suc } 0 < \text{length } is$

using *EP-wf*

by (*fastforce dest: mdecl-visible simp: wf-jvm-prog-phi-def EP-def*)

lemma *C-sees-M-in-EP* [*simp*]:

$EP \vdash "C" \text{ sees } "M": [] \rightarrow \text{Void} = (1, 0, [\text{Push Unit}, \text{Return}], []) \text{ in } "C"$

apply (*auto simp: Method-def EP-def*)

apply (*rule-tac x=Map.empty("M" $\mapsto$ (([], 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*]:

$\text{instrs-of } EP "C" "M" = [\text{Push Unit}, \text{Return}]$

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
   $\implies n \in \{(-\text{Entry-}), (- [(\text{"C"}, \text{"M"}, 0)], \text{None } -), (- [(\text{"C"}, \text{"M"}, 1)], \text{None } -),$ 
   $(-\text{Exit-})\}$ 
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 (Node cs opt) [simp]
  show ?thesis
  proof (cases opt)
    case None [simp]
    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 (Some f) [simp]
  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-), (λs. True)✓, (- [("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 (open) 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  $\exists$  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  $\exists$  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  $\exists$  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  $\exists$  as. JVM-CFG-Interpret.path ?prog (-Exit-) as (-Exit-)
  by fastforce
qed
qed

```

abbreviation *lift-to-cfg-wf-prog* :: (*jvmprog* $\Rightarrow$ 'a) $\Rightarrow$ (*cfg-wf-prog* $\Rightarrow$ 'a)
 ($\neg$ CFG)
 where $f_{CFG} \equiv (\lambda P. f \text{ (Rep-cfg-wf-prog } P))$

6.2 Interpretation of the *Postdomination* locale

interpretation *JVM-CFG-Postdomination*:

```

  Postdomination 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
  have prog-is-cfg-wf-prog: Rep-cfg-wf-prog prog  $\in$  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)  $\in$  cfg-wf-prog
  by simp
  hence valid-node (P, C0, Main) n  $\longrightarrow$ 

```

```

  (∃ 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, pc) # 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 = ⟨ $\bigwedge C' M' pc' cs'.$ 
     $\llbracket cs = (C', M', pc') \# cs'; \text{valid-callstack } (P, C0, \text{Main}) \ cs;$ 
     $\text{instrs-of } P_{wf} \ C' \ M' \ ! \ pc' = \text{Throw} \vee$ 
     $(\exists M'' n''. \text{instrs-of } P_{wf} \ C' \ M' \ ! \ pc' = \text{Invoke } M'' \ n'');$ 
     $\text{find-handler-for } P \ \text{Exc } cs = (C, M, pc) \# cs' \rrbracket$ 
     $\implies pc < \text{length } (\text{instrs-of } P_{wf} \ C \ M) \rangle$ 
  note throw = ⟨ $\text{instrs-of } P_{wf} \ C' \ M' \ ! \ pc' = \text{Throw} \vee (\exists M'' n''. \text{instrs-of } P_{wf} \ C'$ 
     $M' \ ! \ pc' = \text{Invoke } M'' \ n'') \rangle$ 
  note fhf = ⟨ $\text{find-handler-for } P \ \text{Exc } (a \# cs) = (C, M, pc) \# cs''$ ⟩
  note v-cs-a-cs = ⟨ $\text{valid-callstack } (P, C0, \text{Main}) \ (a \# cs)$ ⟩
  show ?case
  proof (cases match-ex-table (Pwf) Exc pc' (ex-table-of (Pwf) C' M'))
    case None
    with fhf have fhf-tl:  $\text{find-handler-for } P \ \text{Exc } cs = (C, M, pc) \# cs''$ 
    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 ≠ []  $\longrightarrow (\text{let } (C, M, pc) = \text{hd } cs \text{ in } \exists n. \text{instrs-of } (P_{wf}) \ C \ M \ ! \ pc =$ 
     $\text{Invoke } M' \ n)$ 
    by (cases cs', auto)
    with IH None fhf-tl ⟨ $\text{valid-callstack } (P, C0, \text{Main}) \ cs$ ⟩
    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:
     $(P_{wf}) \vdash C \text{ sees } M: Ts \rightarrow T = (mxs, mxl, is, xt) \text{ in } C \wedge$ 
     $pc' < \text{length } is \wedge (P_{\Phi}) \ C \ M \ ! \ pc' \neq \text{None}$ 
    by (cases cs) fastforce+
  with wf-jvmprog-is-wf [of P]
  have wt-instr:  $(P_{wf}), T, mxs, \text{length } is, xt \vdash is \ ! \ pc', pc' :: (P_{\Phi}) \ C \ M$ 
    by (fastforce dest!: wt-jvm-prog-impl-wt-instr)
  from Some fhf obtain f t D d where  $(f, t, D, pc, d) \in \text{set } (\text{ex-table-of } (P_{wf}) \ C$ 
     $M) \wedge$ 
     $\text{matches-ex-entry } (P_{wf}) \ \text{Exc } pc' \ (f, t, D, pc, d)$ 
    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
 $\implies \text{finite } \{n'. \exists a'. \text{valid-edge prog } a' \wedge \text{sourcenode } a' = n \wedge$
 $\text{targetnode } a' = n'\}$

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-jumpprog-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 (*Node cs x*) [*simp*]

show ?thesis

proof (*cases cs*)

case *Nil*

thus ?thesis

by (*rule-tac B={}* **in** *finite-subset*,
auto elim: JVM-CFG.cases)

next

case (*Cons a cs'*) [*simp*]

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 setsum-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 fastforce
  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 Pwf 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 (Pwf Cx Mx))} =
    {n'. ∃ Cx Mx pc' h.
      (∃ cs'' pcx. (C, M, pc) # cs' = cs'' @ (Cx, Mx, pcx) # h) ∧
      pc' < length (instrs-of (Pwf 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 (Pwf) (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 (Pwf) (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 (Pwf) 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-edgeCFG prog Entry (-Exit-)
  for prog
proof(unfold-locales)
  fix n
  assume vn: CFG.valid-node sourcenode targetnode (valid-edgeCFG prog) n
  thus finite {n'. ∃ a'. valid-edgeCFG 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  $\Rightarrow$  jinja-var  $\Rightarrow$  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  $\Rightarrow$  j-node  $\Rightarrow$  jinja-var set
```

```
  for P :: wf-jvmprog
```

```
  and n :: j-node
```

```
where
```

```
  Def-Load:
```

```
   $\llbracket n = (- (C, M, pc) \# cs, None -);$   

  instrs-of (Pwf) C M ! pc = Load idx;  

  cd = length cs;  

  i = stkLength P C M pc  $\rrbracket$   

 $\implies$  Stk cd i  $\in$  Def P n
```

```
| Def-Store:
```

```
   $\llbracket n = (- (C, M, pc) \# cs, None -);$   

  instrs-of (Pwf) C M ! pc = Store idx;  

  cd = length cs  $\rrbracket$   

 $\implies$  Loc cd idx  $\in$  Def P n
```

```
| Def-Push:
```

```
   $\llbracket n = (- (C, M, pc) \# cs, None -);$   

  instrs-of (Pwf) C M ! pc = Push v;  

  cd = length cs;  

  i = stkLength P C M pc  $\rrbracket$   

 $\implies$  Stk cd i  $\in$  Def P n
```

```
| Def-New-Normal-Stk:
```

```
   $\llbracket n = (- (C, M, pc) \# cs, [(cs', False)] -);$   

  instrs-of (Pwf) C M ! pc = New Cl;  

  cd = length cs;  

  i = stkLength P C M pc  $\rrbracket$   

 $\implies$  Stk cd i  $\in$  Def P n
```

| *Def-New-Normal-Heap*:
 $\llbracket n = (- (C, M, pc) \# cs, [(cs', False)] -);$
instrs-of $(P_{wf}) C M ! pc = New Cl \rrbracket$
 $\implies HeapVar a \in Def P n$

| *Def-Exc-Stk*:
 $\llbracket n = (- (C, M, pc) \# cs, [(cs', True)] -);$
 $cs' \neq [];$
 $cd = length\ cs' - 1;$
 $(C', M', pc') = hd\ cs';$
 $i = stkLength\ P\ C'\ M'\ pc' - 1 \rrbracket$
 $\implies Stk\ cd\ i \in Def\ P\ n$

| *Def-Getfield-Stk*:
 $\llbracket n = (- (C, M, pc) \# cs, [(cs', False)] -);$
instrs-of $(P_{wf}) C M ! pc = Getfield\ Fd\ Cl;$
 $cd = length\ cs;$
 $i = stkLength\ P\ C\ M\ pc - 1 \rrbracket$
 $\implies Stk\ cd\ i \in Def\ P\ n$

| *Def-Putfield-Heap*:
 $\llbracket n = (- (C, M, pc) \# cs, [(cs', False)] -);$
instrs-of $(P_{wf}) C M ! pc = Putfield\ Fd\ Cl \rrbracket$
 $\implies HeapVar a \in Def\ P\ n$

| *Def-Invoke-Loc*:
 $\llbracket n = (- (C, M, pc) \# cs, [(cs', False)] -);$
instrs-of $(P_{wf}) C M ! pc = Invoke\ M'\ n';$
 $cs' \neq [];$
 $hd\ cs' = (C', M', 0);$
 $i < locLength\ P\ C'\ M'\ 0;$
 $cd = Suc\ (length\ cs) \rrbracket$
 $\implies Loc\ cd\ i \in Def\ P\ n$

| *Def-Return-Stk*:
 $\llbracket n = (- (C, M, pc) \# (D, M', pc') \# cs, None -);$
instrs-of $(P_{wf}) C M ! pc = Return;$
 $cd = length\ cs;$
 $i = stkLength\ P\ D\ M'\ (Suc\ pc') - 1 \rrbracket$
 $\implies Stk\ cd\ i \in Def\ P\ n$

| *Def-IAdd-Stk*:
 $\llbracket n = (- (C, M, pc) \# cs, None -);$
instrs-of $(P_{wf}) C M ! pc = IAdd;$
 $cd = length\ cs;$
 $i = stkLength\ P\ C\ M\ pc - 2 \rrbracket$
 $\implies Stk\ cd\ i \in Def\ P\ n$

| *Def-CmpEq-Stk*:
 $\llbracket n = (- (C, M, pc) \# cs, None \ -);$
instrs-of (P_{wf}) $C\ M\ !\ pc = CmpEq$;
 $cd = length\ cs$;
 $i = stkLength\ P\ C\ M\ pc - 2 \rrbracket$
 $\implies Stk\ cd\ i \in Def\ P\ n$

inductive-set *Use* :: *wf-jvmprog* $\Rightarrow$ *j-node* $\Rightarrow$ *jinja-var set*
for $P :: wf-jvmprog$
and $n :: j-node$
where
Use-Load:
 $\llbracket n = (- (C, M, pc) \# cs, None \ -);$
instrs-of (P_{wf}) $C\ M\ !\ pc = Load\ idx$;
 $cd = length\ cs \rrbracket$
 $\implies (Loc\ cd\ idx) \in Use\ P\ n$

| *Use-Store*:
 $\llbracket n = (- (C, M, pc) \# cs, None \ -);$
instrs-of (P_{wf}) $C\ M\ !\ pc = Store\ idx$;
 $cd = length\ cs$;
 $Suc\ i = (stkLength\ P\ C\ M\ pc) \rrbracket$
 $\implies (Stk\ cd\ i) \in Use\ P\ n$

| *Use-New*:
 $\llbracket n = (- (C, M, pc) \# cs, x \ -);$
 $x = None \vee x = \lfloor (cs', False) \rfloor$;
instrs-of (P_{wf}) $C\ M\ !\ pc = New\ Cl \rrbracket$
 $\implies HeapVar\ a \in Use\ P\ n$

| *Use-Getfield-Stk*:
 $\llbracket n = (- (C, M, pc) \# cs, x \ -);$
 $x = None \vee x = \lfloor (cs', False) \rfloor$;
instrs-of (P_{wf}) $C\ M\ !\ pc = Getfield\ Fd\ Cl$;
 $cd = length\ cs$;
 $Suc\ i = stkLength\ P\ C\ M\ pc \rrbracket$
 $\implies Stk\ cd\ i \in Use\ P\ n$

| *Use-Getfield-Heap*:
 $\llbracket n = (- (C, M, pc) \# cs, \lfloor (cs', False) \rfloor \ -);$
instrs-of (P_{wf}) $C\ M\ !\ pc = Getfield\ Fd\ Cl \rrbracket$
 $\implies HeapVar\ a \in Use\ P\ n$

| *Use-Putfield-Stk-Pred*:
 $\llbracket n = (- (C, M, pc) \# cs, None \ -);$
instrs-of (P_{wf}) $C\ M\ !\ pc = Putfield\ Fd\ Cl$;
 $cd = length\ cs$;
 $i = stkLength\ P\ C\ M\ pc - 2 \rrbracket$
 $\implies Stk\ cd\ i \in Use\ P\ n$

| *Use-Putfield-Stk-Update*:
 $\llbracket n = (- (C, M, pc) \# cs, \lfloor (cs', False) \rfloor -);$
instrs-of (P_{wf}) $C M ! pc = \text{Putfield Fd Cl};$
 $cd = \text{length } cs;$
 $i = \text{stkLength } P C M pc - 2 \vee i = \text{stkLength } P C M pc - 1 \rrbracket$
 $\implies \text{Stk } cd \ i \in \text{Use } P \ n$

| *Use-Putfield-Heap*:
 $\llbracket n = (- (C, M, pc) \# cs, \lfloor (cs', False) \rfloor -);$
instrs-of (P_{wf}) $C M ! pc = \text{Putfield Fd Cl} \rrbracket$
 $\implies \text{HeapVar } a \in \text{Use } P \ n$

| *Use-Checkcast-Stk*:
 $\llbracket n = (- (C, M, pc) \# cs, x -);$
 $x = \text{None} \vee x = \lfloor (cs', False) \rfloor;$
instrs-of (P_{wf}) $C M ! pc = \text{Checkcast Cl};$
 $cd = \text{length } cs;$
 $i = \text{stkLength } P C M pc - \text{Suc } 0 \rrbracket$
 $\implies \text{Stk } cd \ i \in \text{Use } P \ n$

| *Use-Checkcast-Heap*:
 $\llbracket n = (- (C, M, pc) \# cs, \text{None} -);$
instrs-of (P_{wf}) $C M ! pc = \text{Checkcast Cl} \rrbracket$
 $\implies \text{HeapVar } a \in \text{Use } P \ n$

| *Use-Invoke-Stk-Pred*:
 $\llbracket n = (- (C, M, pc) \# cs, \text{None} -);$
instrs-of (P_{wf}) $C M ! pc = \text{Invoke } M' \ n';$
 $cd = \text{length } cs;$
 $i = \text{stkLength } P C M pc - \text{Suc } n' \rrbracket$
 $\implies \text{Stk } cd \ i \in \text{Use } P \ n$

| *Use-Invoke-Heap-Pred*:
 $\llbracket n = (- (C, M, pc) \# cs, \text{None} -);$
instrs-of (P_{wf}) $C M ! pc = \text{Invoke } M' \ n' \rrbracket$
 $\implies \text{HeapVar } a \in \text{Use } P \ n$

| *Use-Invoke-Stk-Update*:
 $\llbracket n = (- (C, M, pc) \# cs, \lfloor (cs', False) \rfloor -);$
instrs-of (P_{wf}) $C M ! pc = \text{Invoke } M' \ n';$
 $cd = \text{length } cs;$
 $i < \text{stkLength } P C M pc;$
 $i \geq \text{stkLength } P C M pc - \text{Suc } n' \rrbracket$
 $\implies \text{Stk } cd \ i \in \text{Use } P \ n$

| *Use-Return-Stk*:
 $\llbracket n = (- (C, M, pc) \# (D, M', pc') \# cs, \text{None} -);$
instrs-of (P_{wf}) $C M ! pc = \text{Return};$

$cd = \text{Suc } (\text{length } cs);$
 $i = \text{stkLength } P \ C \ M \ pc - 1 \]$
 $\implies \text{Stk } cd \ i \in \text{Use } P \ n$

$| \text{Use-IAdd-Stk:}$
 $\llbracket n = (- (C, M, pc) \# cs, \text{None } -);$
 $\text{instrs-of } (P_{wf}) \ C \ M \ ! \ pc = \text{IAdd};$
 $cd = \text{length } cs;$
 $i = \text{stkLength } P \ C \ M \ pc - 1 \ \vee \ i = \text{stkLength } P \ C \ M \ pc - 2 \]$
 $\implies \text{Stk } cd \ i \in \text{Use } P \ n$

$| \text{Use-IfFalse-Stk:}$
 $\llbracket n = (- (C, M, pc) \# cs, \text{None } -);$
 $\text{instrs-of } (P_{wf}) \ C \ M \ ! \ pc = (\text{IfFalse } b);$
 $cd = \text{length } cs;$
 $i = \text{stkLength } P \ C \ M \ pc - 1 \]$
 $\implies \text{Stk } cd \ i \in \text{Use } P \ n$

$| \text{Use-CmpEq-Stk:}$
 $\llbracket n = (- (C, M, pc) \# cs, \text{None } -);$
 $\text{instrs-of } (P_{wf}) \ C \ M \ ! \ pc = \text{CmpEq};$
 $cd = \text{length } cs;$
 $i = \text{stkLength } P \ C \ M \ pc - 1 \ \vee \ i = \text{stkLength } P \ C \ M \ pc - 2 \]$
 $\implies \text{Stk } cd \ i \in \text{Use } P \ n$

$| \text{Use-Throw-Stk:}$
 $\llbracket n = (- (C, M, pc) \# cs, x -);$
 $x = \text{None} \ \vee \ x = \lfloor (cs', \text{True}) \rfloor;$
 $\text{instrs-of } (P_{wf}) \ C \ M \ ! \ pc = \text{Throw};$
 $cd = \text{length } cs;$
 $i = \text{stkLength } P \ C \ M \ pc - 1 \]$
 $\implies \text{Stk } cd \ i \in \text{Use } P \ n$

$| \text{Use-Throw-Heap:}$
 $\llbracket n = (- (C, M, pc) \# cs, x -);$
 $x = \text{None} \ \vee \ x = \lfloor (cs', \text{True}) \rfloor;$
 $\text{instrs-of } (P_{wf}) \ C \ M \ ! \ pc = \text{Throw} \]$
 $\implies \text{HeapVar } a \in \text{Use } P \ n$

declare *correct-state-def* [simp del]

lemma *edge-transfer-uses-only-Use:*

$\llbracket \text{valid-edge } (P, C0, \text{Main}) \ a; \forall V \in \text{Use } P \ (\text{sourcenode } a). \ \text{state-val } s \ V = \text{state-val } s' \ V \rrbracket$
 $\implies \forall V \in \text{Def } P \ (\text{sourcenode } a). \ \text{state-val } (\text{BasicDefs.transfer } (\text{kind } a) \ s) \ V =$
 $\text{state-val } (\text{BasicDefs.transfer } (\text{kind } a) \ s') \ V$

proof

fix V

assume $ve: \text{valid-edge } (P, C0, \text{Main}) \ a$

and *use-eq*: $\forall V \in \text{Use } P \text{ (sourcenode } a). \text{ state-val } s \ V = \text{ state-val } s' \ V$
and *v-in-def*: $V \in \text{Def } P \text{ (sourcenode } a)$
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 $P\text{-wf} = \text{wf-jvmprog-is-wf [of } P]$
from *ve*
have *ex-edge*: $(P, C0, \text{Main}) \vdash (\text{sourcenode } a) \text{--kind } a \rightarrow (\text{targetnode } a)$
and *vn*: *valid-node* $(P, C0, \text{Main})$ (*sourcenode a*)
by *simp-all*
show $\text{state-val } (\text{transfer } (\text{kind } a) \ s) \ V = \text{state-val } (\text{transfer } (\text{kind } a) \ s') \ V$
proof (*cases sourcenode a*)
case (*Node cs x*) $[simp]$
from *vn ex-edge* **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+*)
with vn obtain $ST \ LT$ **where** $wt: ((P_\Phi) \ C \ M \ ! \ pc) = [(ST, LT)]$
by (*cases cs', (cases x, auto)+*)
show *?thesis*
proof (*cases instrs-of (P_{wf}) C M ! pc*)
case (*Load n*) $[simp]$
from *ex-edge* **have** $[simp]: x = \text{None}$
by (*auto elim!: JVM-CFG.cases*)
hence $\text{Loc } (\text{length } cs') \ n \in \text{Use } P \text{ (sourcenode } a)$
by (*auto intro!: Use-Load*)
with use-eq **have** $\text{state-val } s \ (\text{Loc } (\text{length } cs') \ n) = \text{state-val } s' \ (\text{Loc } (\text{length } cs') \ n)$
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 (*Store n*) $[simp]$
from *ex-edge* **have** $[simp]: x = \text{None}$
by (*auto elim!: JVM-CFG.cases*)
have $ST \neq []$
proof –
from *vn*
obtain $Ts \ T \ mxs \ mxl \ is \ xt$
where $C\text{-sees-}M: P_{wf} \vdash C \text{ sees } M: Ts \rightarrow T = (mxs, mxl, is, xt) \text{ in } C$
by (*cases cs', auto*)
with vn
have $pc < \text{length } is$
by (*cases cs', auto dest: sees-method-fun*)
from $P\text{-wf } C\text{-sees-}M$
have $wt\text{-method } (P_{wf}) \ C \ Ts \ T \ mxs \ mxl \ is \ xt \ (P_\Phi \ C \ M)$
by (*auto dest: sees-wf-mdecl simp: wf-jvm-prog-phi-def wf-mdecl-def*)
with $\text{Store } C\text{-sees-}M \ wt \ (pc < \text{length } is)$

```

    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 (Push val) [simp]
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 (New Cl) [simp]
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 (Getfield Fd Cl) [simp]
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 ST ≠ []
  proof -
    from vn obtain T Ts mxs mxl is xt
      where sees-M: (Pwf) ⊢ 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 (Pwf) C Ts T mxs mxl is xt (PΦ 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 (sourcnode 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 (sourcnode 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 (Putfield Fd Cl) [simp]
  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 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', 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 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 (Checkcast Cl) [simp]
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 (Invoke M' n') [simp]
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 False [simp]
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

```

M)
 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)
 by (auto dest: sees-wf-mdecl simp: wf-jvm-prog-phi-def wf-mdecl-def)
 with Invoke sees- M $\langle pc < length\ is \rangle$ 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]: $ST = STn @ STs \# STr \wedge length\ STn = n'$
 by (auto simp: id-take-nth-drop)
 from wt
 have $\forall i. i \leq n' \longrightarrow Stk\ (length\ cs')\ (length\ ST - Suc\ i) \in Use\ P$
 (sourcenode a)
 by (fastforce intro: Use-Invoke-Stk-Update)
 with use-eq
 have
 $\forall i. i \leq n' \longrightarrow state-val\ s\ (Stk\ (length\ cs')\ (length\ ST - Suc\ i)) =$
 $state-val\ s'\ (Stk\ (length\ cs')\ (length\ ST - Suc\ i))$
 by (simp del: state-val.simps)
 hence stk-eq:
 $\forall i. i \leq n' \longrightarrow state-val\ s\ (Stk\ (length\ cs')\ (i + length\ STr)) =$
 $state-val\ s'\ (Stk\ (length\ cs')\ (i + length\ STr))$
 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 Return [simp]
 show ?thesis
 proof (cases x)
 case None [simp]
 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  $Stk \ (length \ cs') \ (length \ ST - 1) \in Use \ P \ (sourcnode \ a)$ 
      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 IAdd [simp]
  show ?thesis
  proof (cases  $x$ )
    case None [simp]
    from wt
    have  $Stk \ (length \ cs') \ (length \ ST - 1) \in Use \ P \ (sourcnode \ a)$ 
      (is ?stk-top  $\in ?Use$ )
      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  $Stk \ (length \ cs') \ (length \ ST - 2) \in Use \ P \ (sourcnode \ a)$ 
      (is ?stk-snd  $\in ?Use$ )
      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 (IfFalse b) [simp]
show ?thesis
proof (cases x)
  case None [simp]
  from wt
  have Stk (length cs') (length ST - 1) ∈ Use P (sourcnode 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 CmpEq [simp]
show ?thesis
proof (cases x)
  case None [simp]
  have Stk (length cs') (stkLength P C M pc - 1) ∈ Use P (sourcnode 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 (sourcnode 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 Throw [simp]
show ?thesis
proof (cases x)
  case None [simp]
  have Stk (length cs') (stkLength P C M pc - 1)  $\in$  Use P (sourcenode a)
    (is ?stk-top  $\in$  ?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  $\longrightarrow$  Stk (length cs') (stkLength P C M pc - 1)  $\in$  Use P (sourcenode
a)
    (is xf  $\longrightarrow$  ?stk-top  $\in$  ?Use)
    by (fastforce intro: Use-Throw-Stk)
  with use-eq
  have stk-top:xf  $\longrightarrow$  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  $\neg$ (erule Def.cases, auto)
qed
qed

```

lemma *CFG-edge-Uses-pred-equal*:

```

 $\llbracket$  valid-edge (P,C0,Main) a;
pred (kind a) s;
 $\forall V \in \text{Use } P \text{ (sourcenode } a).$  state-val s V = state-val s' V
 $\implies$  pred (kind a) s'

```

proof –

```

assume ve: valid-edge (P,C0,Main) a
and pred: pred (kind a) s
and use-eq:  $\forall V \in \text{Use } P \text{ (sourcenode } a).$  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)  $\vdash$  (sourcenode a)  $\neg$  kind a  $\rightarrow$  (targetnode a)

```

```

    by simp-all
  note  $P\text{-wf} = \text{wf-jump-prog-is-wf}$  [of  $P$ ]
  show  $\text{pred}(\text{kind } a) s'$ 
  proof (cases sourcenode  $a$ )
    case (Node  $cs\ x$ ) [simp]
    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  $ex\text{-edge}$  show ?thesis
    by (auto elim: JVM-CFG.cases)
  next
    case (Store  $nat$ )
    with  $ex\text{-edge}$  show ?thesis
    by (auto elim: JVM-CFG.cases)
  next
    case (Push  $val$ )
    with  $ex\text{-edge}$  show ?thesis
    by (auto elim: JVM-CFG.cases)
  next
    case (New  $Cl$ ) [simp]
    show ?thesis
  proof (cases  $x$ )
    case None
    hence  $\forall \text{addr}. \text{HeapVar } \text{addr} \in \text{Use } P\ (\text{sourcenode } a)$ 
    by (auto intro!: Use-New)
    with  $use\text{-eq}$  have  $\forall \text{addr}. \text{state-val } s\ (\text{HeapVar } \text{addr}) = \text{state-val } s'\ (\text{HeapVar } \text{addr})$ 
    by (simp del: state-val.simps)
    hence  $h = h'$ 
    by (auto intro: ext)
    with  $ex\text{-edge } \text{pred}$  show ?thesis
    by (auto elim!: JVM-CFG.cases)
  next
    case (Some  $x'$ )
    with  $ex\text{-edge}$  show ?thesis
    by (auto elim: JVM-CFG.cases)
  qed
  next
    case (Getfield  $Fd\ Cl$ ) [simp]
    have  $ST \neq []$ 
  proof -
    from  $vn$  obtain  $T\ Ts\ mxs\ mxl\ is\ xt$ 
    where  $\text{sees-}M: (P_{wf}) \vdash C\ \text{sees } M: Ts \rightarrow T = (mxs, mxl, is, xt)\ \text{in } C$ 

```

```

      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 (Pwf) C Ts T mxs mxl is xt (PΦ 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, fast-
force+)
  show ?thesis
  proof (cases x)
    case None [simp]
    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 (Putfield Fd Cl) [simp]
have length ST > 1
proof -
  from vn obtain T Ts mxs mxl is xt
  where sees-M: (Pwf) ⊢ 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 (Pwf) C Ts T mxs mxl is xt (PΦ 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 None [simp]

```

```

with wt
have Stk (length cs') (length ST - 2)  $\in$  Use P (sourcenode a)
  (is ?stk-top  $\in$  ?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 (Checkcast Cl) [simp]
have ST  $\neq$  []
proof -
  from vn obtain T Ts mxs mxl is xt
    where sees-M: (Pwf)  $\vdash$  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 (Pwf) C Ts T mxs mxl is xt (PΦ 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, fast-
force+)
show ?thesis
proof (cases x)
  case None [simp]
  from wt
  have Stk (length cs') (stkLength P C M pc - Suc 0)  $\in$  Use P (sourcenode
a)
    (is ?stk-top  $\in$  ?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  $\forall$  addr. HeapVar addr  $\in$  Use P (sourcenode a)
    by (fastforce intro: Use-Checkcast-Heap)
  with use-eq
  have  $\forall$  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 (Invoke M' n') [simp]
  have length ST > n'
  proof -
    from vn obtain T Ts mxs mxl is xt
      where sees-M: (Pwf) ⊢ 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 (Pwf) C Ts T mxs mxl is xt (PΦ 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 None [simp]
    with wt
    have Stk (length cs') (stkLength P C M pc - Suc n') ∈ Use P (sourcnode
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 (sourcnode 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 (IfFalse b) [simp]
    show ?thesis
    proof (cases x)
      case None [simp]
      have Stk (length cs') (stkLength P C M pc - 1) ∈ Use P (sourcename 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 Throw [simp]
    have ST ≠ []
    proof -
      from vn obtain T Ts mxs mxl is xt
        where sees-M: (Pwf) ⊢ 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 (Pwf) C Ts T mxs msl is xt (PΦ 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, fast-
force+)
show ?thesis
proof (cases x)
  case None [simp]
  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)  $\vdash$  (sourcenode a) $\text{--kind } a \rightarrow$ (targetnode a)
  by simp-all
show state-val (transfer (kind a) s) V = state-val s V
proof (cases sourcenode a)
  case (Node cs x) [simp]
  with 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+)
  with 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 (Pwf) C M ! pc)
  case (Load nat) [simp]
  from ex-edge have x = None
    by (auto elim: JVM-CFG.cases)
  with v-not-def have V  $\neq$  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 (Store nat) [simp]
  with ex-edge have x = None
    by (auto elim: JVM-CFG.cases)
  with v-not-def have V  $\neq$  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 (Push val) [simp]
  with ex-edge have x = None
    by (auto elim: JVM-CFG.cases)
  with v-not-def have V  $\neq$  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 (New Cl) [simp]
  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)
    apply (cases V, auto intro!: Def-New-Normal-Stk Def-New-Normal-Heap)
    by (cases V, auto intro!: Def-Exc-Stk)+
qed
next
case (Getfield F Cl) [simp]
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 simp: split-beta)
  apply (cases V, auto intro!: Def-Getfield-Stk)
  by (cases V, auto intro!: Def-Exc-Stk)+
qed
next
case (Putfield Fd Cl) [simp]
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 simp: split-beta)
  apply (cases V, auto intro!: Def-Putfield-Heap)
  by (cases V, auto intro!: Def-Exc-Stk)+
qed
next
case (Checkcast Cl) [simp]
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)

```

```

      by (cases V, auto intro!: Def-Exc-Stk)+
    qed
  next
    case (Invoke M' n') [simp]
    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)
      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 Throw [simp]
    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)
    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-) = {}  $\wedge$  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:  $V \notin \text{Def } (fst \text{ prog}) (\text{sourcenode } a)$ 
  thus state-val (transfer (kind a) s) V = state-val s V
  by  $\neg$ (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:  $\forall V \in \text{Use } (fst \text{ prog}) (\text{sourcenode } a). \text{state-val } s \ V = \text{state-val } s' \ V$ 
  thus  $\forall V \in \text{Def } (fst \text{ prog}) (\text{sourcenode } a).$ 
    state-val (transfer (kind a) s) V = state-val (transfer (kind a) s') V
  by  $\neg$ (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:  $\forall V \in \text{Use } (fst \text{ prog}) (\text{sourcenode } a). \text{state-val } s \ V = \text{state-val } s' \ V$ 
  thus pred (kind a) s'
  by  $\neg$ (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)✓ ∧ kind a' = (Q')✓ ∧ (∀ 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-edge_{CFG} prog (-Entry-) Def (fst_{CFG} prog) Use (fst_{CFG} prog)
 state-val (-Exit-) ..

interpretation JVMDynWeakControlDependence:

DynWeakControlDependencePDG sourcenode targetnode kind
 valid-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 $\equiv (\lambda(a, b).$
 if $length\ frs \leq a$ *then* *stk* (a, b)
 else let *xs* = *fst* (*frs* ! ($length\ frs - Suc\ a$))
 in if $length\ xs \leq b$ *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 $\equiv (\lambda(a, b).$
 if $length\ frs \leq a$ *then* *loc* (a, b)
 else let *xs* = *fst* (*snd* (*frs* ! ($length\ frs - Suc\ a$)))
 in if $length\ xs \leq b$ *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]:
 $\text{update-stk } stk \text{ (snd(snd(state-to-jvm-state } P \text{ cs (h,stk,loc))))} = stk$
by (auto intro!: ext simp: nth-stkss)

lemma *update-loc-s2jvm'* [simp]:
 $\text{update-loc } loc \text{ (zip (stkss } P \text{ cs stk) (zip (locss } P \text{ cs loc) cs))} = loc$
by (auto intro!: ext simp: nth-locss)

lemma *update-stk-s2jvm'* [simp]:
 $\text{update-stk } stk \text{ (zip (stkss } P \text{ cs stk) (zip (locss } P \text{ cs loc) cs))} = stk$
by (auto intro!: ext simp: nth-stkss)

lemma *find-handler-find-handler-forD*:
 $\text{find-handler } (P_{wf}) \text{ a h frs} = (xp', h', frs')$
 $\implies \text{find-handler-for } P \text{ (cname-of h a) (framestack-to-callstack frs)} =$
 $\text{framestack-to-callstack frs'}$
by (induct frs, auto)

lemma *find-handler-nonempty-frs* [simp]:
 $(\text{find-handler } P \text{ a h frs} \neq (\text{None}, h', []))$
by (induct frs, auto)

lemma *find-handler-heap-eqD*:
 $\text{find-handler } P \text{ a h frs} = (xp, h', frs') \implies h' = h$
by (induct frs, auto)

lemma *find-handler-frs-decrD*:
 $\text{find-handler } P \text{ a h frs} = (xp, h', frs') \implies \text{length frs'} \leq \text{length frs}$
by (induct frs, auto)

lemma *find-handler-decrD* [dest]:
 $\text{find-handler } P \text{ a h frs} = (xp, h', f \# frs) \implies \text{False}$
by (drule find-handler-frs-decrD, simp)

lemma *find-handler-decrD'* [dest]:
 $\llbracket \text{find-handler } P \text{ a h frs} = (xp, h', f \# frs'); \text{length frs} = \text{length frs'} \rrbracket \implies \text{False}$
by (drule find-handler-frs-decrD, simp)

lemma *Suc-minus-Suc-Suc* [simp]:
 $b < n - 1 \implies \text{Suc } (n - \text{Suc } (\text{Suc } b)) = n - \text{Suc } b$
by simp

lemma *find-handler-loc-fun-eq'*:
 $\text{find-handler } (P_{wf}) \text{ a h}$
 $\text{(zip (stkss } P \text{ cs stk) (zip (locss } P \text{ cs loc) cs))} =$
 (xf, h', frs)
 $\implies \text{update-loc } loc \text{ frs} = loc$

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  $IH = \langle \text{find-handler } (P_{wf}) a h$ 
    (zip (stkss  $P cs' stk$ ) (zip (locss  $P cs' loc$ )  $cs'$ )) =
    ( $xf, h', frs$ )  $\implies$ 
    update-loc  $loc frs x = loc x \rangle$ 
  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  $P cs (h, stk, loc)$ ))) = ( $xf, h', frs$ )
   $\implies$  update-loc  $loc frs = loc$ 
  by (simp add: find-handler-loc-fun-eq')

lemma find-handler-stk-fun-eq':
   $\llbracket \text{find-handler } (P_{wf}) a h$ 
    (zip (stkss  $P cs stk$ ) (zip (locss  $P cs loc$ )  $cs$ )) =
    (None, h', frs);
   $cd = \text{length } frs - 1;$ 
   $i = \text{length } (\text{fst}(\text{hd}(frs))) - 1 \rrbracket$ 
   $\implies \text{update-stk } stk frs = stk((cd, i) := \text{Addr } a)$ 
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:  $cd = \text{length } frs - 1$ 
  and idx:  $i = \text{length } (fst (hd frs)) - 1$ 
from find-handler have  $frs \neq []$ 
  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  $IH = \langle \text{find-handler } (P_{wf}) a h$ 
    (zip (stkss  $P cs' stk$ ) (zip (locss  $P cs' loc$ )  $cs'$ )) =
    (None,  $h'$ ,  $frs$ )  $\implies$ 
    update-stk  $stk frs x = (stk((cd, i) := Addr a)) x \rangle$ 
  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  $a' = \text{length } cs'$ )
    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*:
 $\text{find-handler } (P_{wf}) \ a \ h \ (\text{snd}(\text{snd}(\text{state-to-jvm-state } P \ cs \ (h, \text{stk}, \text{loc})))) = (\text{None}, h', \text{frs})$
 $\implies \text{update-stk } \text{stk} \ \text{frs} = \text{stk}((\text{length } \text{frs} - 1, \text{length } (\text{fst}(\text{hd}(\text{frs}))) - 1) := \text{Addr } a)$
by (*simp add: find-handler-stk-fun-eq*)

lemma *f2c-emptyD [dest]*:
 $\text{framestack-to-callstack } \text{frs} = [] \implies \text{frs} = []$
by (*simp add: framestack-to-callstack-def*)

lemma *f2c-emptyD' [dest]*:
 $[] = \text{framestack-to-callstack } \text{frs} \implies \text{frs} = []$
by (*simp add: framestack-to-callstack-def*)

lemma *correct-state-imp-valid-callstack*:
 $\llbracket P, cs \vdash_{BV} s \ \checkmark; \text{fst } (\text{last } cs) = C0; \text{fst}(\text{snd } (\text{last } cs)) = \text{Main} \rrbracket$
 $\implies \text{valid-callstack } (P, C0, \text{Main}) \ cs$
proof (*cases cs rule: rev-cases*)
case *Nil*
thus *?thesis* **by** *simp*
next
case (*snoc cs' y*)
assume *bv-correct*: $P, cs \vdash_{BV} s \ \checkmark$
and *last-C*: $\text{fst } (\text{last } cs) = C0$
and *last-M*: $\text{fst}(\text{snd } (\text{last } cs)) = \text{Main}$
with *snoc* **obtain** *pcX* **where** [*simp*]: $cs = cs' @ [(C0, \text{Main}, pcX)]$
by (*cases last cs, fastforce*)
obtain *h stk loc* **where** [*simp*]: $s = (h, \text{stk}, \text{loc})$
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 (*Cons a frs'*) [*simp*]
obtain *stk' loc' C M pc* **where** [*simp*]: $a = (\text{stk}', \text{loc}', C, M, pc)$ **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*: $(P_{wf}) \vdash C \text{ sees } M: Ts \rightarrow T = (mxs, mxl, is, xt) \text{ in } C$

```

    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 (Pwf) C Ts T mxs mxl is xt (PΦ 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:
  ⟦(Pwf),(PΦ) ⊢ state-to-jvm-state P cs (h,stk,loc) √;
  (Pwf),(PΦ) ⊢ find-handler (Pwf) 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'
  ⟧ ⇒
  (Pwf),(PΦ) ⊢ (None, h,
    (stks (stkLength P C' M' pc')
      (λa'. (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')) √
proof (induct cs)
  case Nil
  thus ?case by simp
next
  case (Cons aa cs)
  note state-correct = ⟨Pwf,PΦ ⊢ state-to-jvm-state P (aa # cs) (h, stk, loc) √⟩
  note IH = ⟨⟦Pwf,PΦ ⊢ state-to-jvm-state P cs (h, stk, loc) √;
    Pwf,PΦ ⊢ find-handler Pwf 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'⟧
    ⇒ Pwf,PΦ ⊢ (None, h,
      (stks (stkLength P C' M' pc')
        (λa'. (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')) √
note trg-state-correct = ⟨Pwf, PΦ ⊢ find-handler Pwf a h
      (zip (stkss P (aa # cs) stk)
      (zip (locss P (aa # cs) loc) (aa # cs))) √
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: Pwf, PΦ ⊢ state-to-jvm-state P cs (h, stk, loc) √
  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 (Pwf) (cname-of h a) pc (ex-table-of (Pwf) 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=mxl0 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:
   $x \geq z \implies$ 
  locs z
  ( $\lambda b.$ 
    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 ( $\lambda a.$  stk(y, a))))
  apply (rule nth-equalityI)
  apply (simp)
  apply (auto simp: nth-append nth-Cons' less-Suc-eq min-max.inf-absorb2 min-max.sup-absorb2)
done

lemma locs-invoke-purge:
  ( $z::nat$ ) > c  $\implies$ 
  locs l
  ( $\lambda b.$  if  $z = c \longrightarrow Q\ b$  then loc (c, b) else u b) =
  locs l ( $\lambda a.$  loc (c, a))
  by (induct l, auto)

lemma nth-rev-equalityI:
   $\llbracket \text{length } xs = \text{length } ys; \forall i < \text{length } xs. xs ! (\text{length } xs - \text{Suc } i) = ys ! (\text{length } ys - \text{Suc } i) \rrbracket$ 
 $\implies xs = ys$ 
proof (induct xs ys rule: list-induct2)
  case Nil
  thus ?case by simp
next
  case (Cons x xs y ys)
  hence  $\forall i < \text{length } ys. xs ! (\text{length } ys - \text{Suc } i) = ys ! (\text{length } ys - \text{Suc } i)$ 
  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*:

$i < \text{length } cs$
 $\implies \text{length } (\text{locss } P \text{ } cs \text{ } loc \text{ } ! (\text{length } cs - \text{Suc } i)) =$
 $\text{locLength } P \text{ } (\text{fst}(cs \text{ } ! (\text{length } cs - \text{Suc } i)))$
 $\quad (\text{fst}(\text{snd}(cs \text{ } ! (\text{length } cs - \text{Suc } i))))$
 $\quad (\text{snd}(\text{snd}(cs \text{ } ! (\text{length } cs - \text{Suc } i))))$

apply (*induct cs, auto*)

apply (*case-tac i = length cs*)

by (*auto simp: nth-Cons'*)

lemma *locss-invoke-purge*:

$z > \text{length } cs \implies$
 $\text{locss } P \text{ } cs$
 $\quad (\lambda(a, b). \text{ if } (a = z \longrightarrow Q \text{ } b)$
 $\quad \text{ then } loc \text{ } (a, b)$
 $\quad \text{ else } u \text{ } b)$
 $= \text{locss } P \text{ } cs \text{ } loc$
by (*induct cs, auto simp: locs-invoke-purge [simplified]*)

lemma *stks-purge'*:

$d \geq b \implies \text{stks } b \text{ } (\lambda x. \text{ if } x = d \text{ then } e \text{ else } stk \text{ } x) = \text{stks } b \text{ } stk$
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*:

$\llbracket \text{ valid-edge } (P, C0, \text{Main}) \text{ } a;$
 $\text{ sourcenode } a = (- (C, M, pc) \# cs, x -);$
 $\text{ targetnode } a = (- (C', M', pc') \# cs', x' -);$
 $\text{ pred } (\text{kind } a) \text{ } s;$
 $x \neq \text{None} \longrightarrow (\exists a\text{-pred}.$
 $\quad \text{ sourcenode } a\text{-pred} = (- (C, M, pc) \# cs, \text{None} -) \wedge$
 $\quad \text{ targetnode } a\text{-pred} = \text{ sourcenode } a \wedge$
 $\quad \text{ valid-edge } (P, C0, \text{Main}) \text{ } a\text{-pred} \wedge$
 $\quad \text{ pred } (\text{kind } a\text{-pred}) \text{ } s$
 $\quad \text{ });$
 $P, ((C, M, pc) \# cs) \vdash_{BV} s \text{ } \checkmark \rrbracket$
 $\implies P, ((C', M', pc') \# cs') \vdash_{BV} \text{ transfer } (\text{kind } a) \text{ } s \text{ } \checkmark$

proof –

assume *ve*: *valid-edge* (*P*, *C0*, *Main*) *a*

and *src* [*simp*]: *sourcenode* *a* = $(- (C, M, pc) \# cs, x -)$

and *trg* [*simp*]: *targetnode* *a* = $(- (C', M', pc') \# cs', x' -)$

and *pred-s*: *pred* (*kind* *a*) *s*

and *a-pred*: $x \neq \text{None} \longrightarrow (\exists a\text{-pred}.$

```

    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) ⊢BV s √
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: (Pwf) ⊢ C sees M:Ts→T = (mxs,mxl,is,xt) in C
  and pc < length is
  and reachable: PΦ C M ! pc ≠ None
  by (cases x, (cases cs, auto)+)
from P-wf sees-M
have wt-method: wt-method (Pwf) C Ts T mxs mxl is xt (PΦ 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: appi ((is ! pc),(Pwf),pc,mxs,T,(the(PΦ C M ! pc)))
  by (auto simp: wt-method-def)
from state-correct ve P-wf
have trg-state-correct:
  (Pwf),(PΦ) ⊢ the (JVMEexec.exec ((Pwf), state-to-jvm-state P ((C,M,pc)#cs)
s)) √
  apply simp
  apply (drule BV-correct-1)
  apply (fastforce simp: bv-conform-def)
  apply (simp add: exec-1-iff)
  by (cases instrs-of (Pwf) C M ! pc, simp-all add: split-beta)
from reachable obtain ST LT where reachable: (PΦ) 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 (effi (is ! pc, (Pwf), ST, LT))) ∧
  locLength P C M pc' = length (snd (effi (is ! pc, (Pwf), ST, LT)))
  unfolding wt-method-def
  using [[simplproc del: list-to-set-comprehension]] by (cases is ! pc, (fastforce dest!:
list-all2-lengthD)+)
have [simp]: ∃ x. x by auto
have [simp]: Ex Not by auto
show ?thesis
proof (cases instrs-of (Pwf) 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 (PΦ C M ! pc), auto)
  show ?thesis
proof (cases x)

```



```

case None [simp]
with ve Invoke obtain Q where kind: kind a = (Q)✓
  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 (Some aa) [simp]
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
  by  $-(clarsimp, erule\ JVM-CFG.cases, auto)$ 
show ?thesis
proof (cases xf)
  case True [simp]
    with a-pred Invoke have stk-n: stk (length cs, stkLength P C M pc - Suc
n) = Null
      apply auto
      by (erule JVM-CFG.cases, simp-all)
    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))
      by  $-(clarsimp, erule\ JVM-CFG.cases, auto)$ 
    from ve Invoke
    have find-handler-for P NullPointer ((C, M, pc) \# cs) = (C', M', pc') \# cs'
      by  $-(clarsimp, erule\ JVM-CFG.cases, auto)$ 
    with Invoke state-correct kind stk-n trg-state-correct applicable sees-M
       $\langle preallocated\ h \rangle$ 
    show ?thesis
      apply (cases the (PΦ C M ! pc),
        auto simp: bv-conform-def stkss-purge
        simp del: find-handler-for.simps exec.simps appi.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: split-if-asm)
      by fastforce
  next
  case False [simp]
  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]:
   $f = (\lambda s. exec\_instr\ (Invoke\ m\ n)\ P\ s\ (length\ cs)\ (stkLength\ P\ C\ M\ pc)$ 
     $arbitrary\ (locLength\ P\ C'\ M'\ 0))$ 
  by  $-(clarsimp, erule JVM-CFG.cases, auto)$ 
from  $state-correct$  obtain  $ST\ LT$  where [simp]:
   $(P_{\Phi})\ C\ M\ !\ pc = \lfloor (ST, LT) \rfloor$ 
  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]:  $stk\ (length\ cs,\ length\ ST - Suc\ n) \neq Null$ 
  by  $-(clarsimp, erule JVM-CFG.cases, auto)$ 
from  $state-correct\ applicable\ sees-M\ Invoke$ 
have [simp]:  $ST\ !\ n \neq NT$ 
  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  $(auto\ simp: split-beta\ correct-state-def$ 
     $split: split-if-asm)$ 
from  $trg-state-correct\ Invoke\ \langle length\ ST > n \rangle$ 
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  $(auto\ simp: split-beta\ correct-state-def$ 
     $dest!: list-all2-lengthD$ 
     $split: split-if-asm)$ 
with  $Invoke\ state-correct\ trg-state-correct\ \langle length\ ST > n \rangle$ 
have  $JVMExec.exec\ (P_{wf}, state-to-jvm-state\ P\ ((C, M, pc) \# cs)\ s)$ 
   $=$ 
   $\lfloor (None, h,$ 
     $(stks\ (stkLength\ P\ C'\ M'\ pc')\ (\lambda a. stk\ (Suc\ (length\ cs), a)),$ 
     $locs\ (locLength\ P\ C'\ M'\ pc')$ 
     $(\lambda a'. (\lambda(a, b).$ 
       $if\ a = Suc\ (length\ cs) \longrightarrow locLength\ P\ C'\ M'\ 0 \leq b\ then\ loc$ 
       $(a, b)$ 
       $else\ if\ b \leq n\ then\ stk\ (length\ cs,\ length\ ST - (Suc\ n - b))$ 

```

```

      else arbitrary) (Suc (length cs), a'),
      C', M', pc') #
      (stks (length ST) (λa. stk (length cs, a)),
       locs (length LT) (λa. 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-max.inf-absorb1 min-max.inf-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 (PΦ C M ! pc),
    auto simp: bv-conform-def stkss-purge rev-nth
    simp del: find-handler-for.simps exec.simps appi.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: split-if-asm)
next
case Return
with ve obtain Ts' T' mxs' mxl' is' xt'
  where sees-M': (Pwf) ⊢ C' sees M': Ts' → T' = (mxs', mxl', is', xt') in C'
  and (pc' - 1) < length is'
  and reachable': PΦ 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 (Pwf) C' Ts' T' mxs' mxl' is' xt' (PΦ 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 [[simpproc del: list-to-set-comprehension]]
  apply auto
  apply (erule JVM-CFG.cases, auto)
  apply (drule sees-method-fun, fastforce, clarsimp)
  using sees-M'
  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  $\langle wt\text{-}method\ (P_{wf})\ C'\ Ts'\ T'\ mxs'\ mxl'\ is'\ xt'\ (P_{\Phi}\ C'\ M') \rangle$ 
 $\langle (pc' - 1) < length\ is' \rangle \langle P_{\Phi}\ C'\ M'!\ (pc' - 1) \neq None \rangle$ 
 $\langle is'!\ (pc' - 1) = Invoke\ M\ (length\ Ts) \rangle$ 
have  $stkLength\ P\ C'\ M'\ (pc' - 1) > 0$ 
by (fastforce simp: wt-method-def)
then obtain  $ST'\ STr'$  where  $[simp]:\ fst\ (the\ (P_{\Phi}\ C'\ M'!\ (pc' - 1))) =$ 
 $ST'\#STr'$ 
by (cases  $fst\ (the\ (P_{\Phi}\ C'\ M'!\ (pc' - 1)))$ , 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  $\langle wt\text{-}method\ (P_{wf})\ C'\ Ts'\ T'\ mxs'\ mxl'\ is'\ xt'\ (P_{\Phi}\ C'\ M') \rangle$ 
 $ve\ Return\ \langle pc' - 1 < length\ is' \rangle\ reachable'\ sees\text{-}M\ state\text{-}correct$ 
have  $locLength\ P\ C'\ M'\ (pc' - 1) = locLength\ P\ C'\ M'\ pc'$ 
using  $[[simp\ proc\ del:\ list\text{-}to\text{-}set\text{-}comprehension]]$ 
apply auto
apply (erule JVM-CFG.cases, auto)
apply (drule sees-method-fun, fastforce, clarsimp)
using sees-M'
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  $\langle stkLength\ P\ C'\ M'\ pc' = stkLength\ P\ C'\ M'\ (pc' - 1) - length\ Ts \rangle$ 
 $Return\ state\text{-}correct\ ve\ P\text{-}wf\ applicable\ sees\text{-}M\ trg\text{-}state\text{-}correct\ sees\text{-}M'$ 
 $\langle fst\ (the\ (P_{\Phi}\ C'\ M'!\ (pc' - 1))) = ST'\#STr' \rangle \langle is'!\ (pc' - 1) = Invoke\ M$ 
 $(length\ Ts) \rangle$ 
 $\langle locLength\ P\ C\ M\ 0 = Suc\ (length\ Ts) + mxl \rangle$ 
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* $\Rightarrow$ *callstack* $\Rightarrow$ *state* $\Rightarrow$ *callstack* $\Rightarrow$ *state* $\Rightarrow$ *bool*
 $(- \vdash \langle -, - \rangle \Rightarrow \langle -, - \rangle)$
where *Step*:
 $\llbracket \text{prog} = (P, C0, \text{Main});$
 $P, cs \vdash_{BV} s \checkmark;$
 $\text{valid-callstack prog cs};$
 $JVMExec.exec ((P_{wf}), \text{state-to-jvm-state } P \text{ cs } s) = \lfloor (None, h', frs') \rfloor;$
 $cs' = \text{framestack-to-callstack frs'};$
 $s = (h, stk, loc);$
 $s' = (h', \text{update-stk stk frs'}, \text{update-loc loc frs'}) \rrbracket$
 $\implies \text{prog} \vdash \langle cs, s \rangle \Rightarrow \langle cs', s' \rangle$

abbreviation *identifies* :: *j-node* $\Rightarrow$ *callstack* $\Rightarrow$ *bool*
where *identifies* *n cs* $\equiv (n = (- \text{ cs}, None -))$

7.2.2 Some more simplification lemmas

lemma *valid-callstack-tl*:
 $\text{valid-callstack prog } ((C, M, pc) \# cs) \implies \text{valid-callstack prog cs}$
by (*cases prog*, *cases cs*, *auto*)

lemma *stkss-cong* [*cong*]:
 $\llbracket P = P';$
 $cs = cs';$
 $\bigwedge a b. \llbracket a < \text{length } cs;$
 $b < \text{stkLength } P (\text{fst}(cs ! (\text{length } cs - \text{Suc } a)))$
 $(\text{fst}(\text{snd}(cs ! (\text{length } cs - \text{Suc } a))))$
 $(\text{snd}(\text{snd}(cs ! (\text{length } cs - \text{Suc } a)))) \rrbracket$
 $\implies \text{stk } (a, b) = \text{stk}' (a, b) \rrbracket$
 $\implies \text{stkss } P \text{ cs } \text{stk} = \text{stkss } P' \text{ cs}' \text{stk}'$
by (*auto*, *induct cs'*,
 $\text{auto intro! : nth-equalityI simp: nth-Cons'}$)

lemma *locss-cong* [*cong*]:

$\llbracket P = P';$
 $cs = cs';$
 $\bigwedge a b. \llbracket a < \text{length } cs;$
 $\quad b < \text{locLength } P \text{ (fst}(cs ! (\text{length } cs - \text{Suc } a)))$
 $\quad \quad (\text{fst}(\text{snd}(cs ! (\text{length } cs - \text{Suc } a))))$
 $\quad \quad (\text{snd}(\text{snd}(cs ! (\text{length } cs - \text{Suc } a)))) \rrbracket$
 $\implies \text{loc } (a, b) = \text{loc}' (a, b) \rrbracket$
 $\implies \text{locss } P \text{ } cs \text{ } loc = \text{locss } P' \text{ } cs' \text{ } loc'$
by (*auto*, *induct cs'*,
auto intro!:: nth-equalityI simp: nth-Cons')

lemma *hd-tl-equalityI*:

$\llbracket \text{length } xs = \text{length } ys; \text{hd } xs = \text{hd } ys; \text{tl } xs = \text{tl } ys \rrbracket \implies xs = ys$
apply (*induct xs arbitrary: ys*)
apply *simp*
by (*case-tac ys, auto*)

lemma *stkLength-is-length-stk*:

$P_{wf}, P_{\Phi} \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}') \checkmark \implies \text{stkLength } P \text{ } C \text{ } M \text{ } pc =$
 $\text{length } \text{stk}$
by (*auto dest!:: list-all2-lengthD simp: correct-state-def*)

lemma *locLength-is-length-loc*:

$P_{wf}, P_{\Phi} \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}') \checkmark \implies \text{locLength } P \text{ } C \text{ } M \text{ } pc =$
 $\text{length } \text{loc}$
by (*auto dest!:: list-all2-lengthD simp: correct-state-def*)

lemma *correct-state-frs-tlD*:

$(P_{wf}), (P_{\Phi}) \vdash (\text{None}, h, a \# \text{frs}') \checkmark \implies (P_{wf}), (P_{\Phi}) \vdash (\text{None}, h, \text{frs}') \checkmark$
by (*cases frs', (fastforce simp: correct-state-def)+*)

lemma *update-stk-Cons* [*simp*]:

$\text{stkss } P \text{ (framestack-to-callstack } \text{frs}') \text{ (update-stk } \text{stk} \text{ ((stk}', \text{loc}', C', M', pc') \#$
 $\text{frs}')) =$
 $\text{stkss } P \text{ (framestack-to-callstack } \text{frs}') \text{ (update-stk } \text{stk} \text{ } \text{frs}')$
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*]:

$\text{locss } P \text{ (framestack-to-callstack } \text{frs}') \text{ (update-loc } \text{loc} \text{ ((stk}', \text{loc}', C', M', pc') \#$
 $\text{frs}')) =$

```

    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:
  (Pwf), (P $\Phi$ )  $\vdash$  (None, h', frs')  $\checkmark$ 
 $\implies$  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:
   $\llbracket$  find-handler Pwf a h frs = (None, h', frs');
  last frs = (stk, loc, C, M, pc);
  last frs' = (stk', loc', C', M', pc')  $\rrbracket$ 
 $\implies$  C = C'  $\wedge$  M = M'
  by (induct frs, auto split: split-if-asm)

lemma exec-last-frs-eq-class:
   $\llbracket$  JVMExec.exec (Pwf, None, h, frs) =  $\lfloor$  (None, h', frs')  $\rfloor$ ;
  last frs = (stk, loc, C, M, pc);

```

$last\ frs' = (stk', loc', C', M', pc');$
 $frs \neq [];$
 $frs' \neq []$
 $\implies C = C'$
apply (cases frs, auto split: split-if-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:
 $\llbracket JVMExec.exec\ (P_{wf}, None, h, frs) = \llbracket (None, h', frs') \rrbracket;$
 $last\ frs = (stk, loc, C, M, pc);$
 $last\ frs' = (stk', loc', C', M', pc');$
 $frs \neq [];$
 $frs' \neq []$
 $\implies M = M'$
apply (cases frs, auto split: split-if-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:
 $valid-callstack\ (P, C0, Main)\ (cs @ [(C, M, pc)]) \implies C = C0$
by (induct cs, auto dest: valid-callstack-tl)

lemma valid-callstack-append-last-method:
 $valid-callstack\ (P, C0, Main)\ (cs @ [(C, M, pc)]) \implies M = Main$
by (induct cs, auto dest: valid-callstack-tl)

lemma zip-stkss-locss-append-single [simp]:
 $zip\ (stkss\ P\ (cs\ @\ [(C, M, pc)])\ stk)$
 $\quad (zip\ (locss\ P\ (cs\ @\ [(C, M, pc)])\ loc)\ (cs\ @\ [(C, M, pc)]))$
 $=\ (zip\ (stkss\ P\ (cs\ @\ [(C, M, pc)])\ stk)\ (zip\ (locss\ P\ (cs\ @\ [(C, M, pc)])\ loc)$
 $\quad cs))$
 $\quad @\ [(stks\ (stkLength\ P\ C\ M\ pc)\ (\lambda a. stk\ (0, a)),$
 $\quad locs\ (locLength\ P\ C\ M\ pc)\ (\lambda a. loc\ (0, a)), C, M, pc]$
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 $\text{sem-step:prog} \vdash \langle c, s \rangle \Rightarrow \langle c', s' \rangle$

and identifies $n\ c$

obtain $P\ C0\ M0$

where $\text{prog} [\text{simp}]: \text{prog} = (P, C0, M0)$

by (*cases prog, fastforce*)

obtain $h\ stk\ loc$

where $s [\text{simp}]: s = (h, stk, loc)$

by (*cases s, fastforce*)

obtain $h'\ stk'\ loc'$

where $s' [\text{simp}]: s' = (h', stk', loc')$

by (*cases s', fastforce*)

from *sem-step* $s\ s'\ \text{prog}$ obtain $C\ M\ pc\ cs\ C'\ M'\ pc'\ cs'$

where $c [\text{simp}]: c = (C, M, pc) \# cs$

by (*cases c, auto elim: sem.cases simp: bv-conform-def*)

with *sem-step* prog obtain $ST\ LT$

where $wt [\text{simp}]: (P_\Phi)\ C\ M\ !\ pc = \lfloor (ST, LT) \rfloor$

by (*auto elim!: sem.cases, cases cs, fastforce+*)

note $P\text{-wf} = \text{wf-jvmprog-is-wf} [\text{of } P]$

from *sem-step* prog obtain frs'

where $\text{jvm-exec: JVMExec.exec} ((P_{\text{wf}}), \text{state-to-jvm-state } P\ c\ s) = \lfloor (\text{None}, h', \text{frs}') \rfloor$

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

with *sem-step* prog $s\ s'$

have $\text{loc}': \text{loc}' = \text{update-loc } \text{loc} \ \text{frs}'$

and $\text{stk}': \text{stk}' = \text{update-stk } \text{stk} \ \text{frs}'$

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

from *sem-step* $s\ \text{prog}$

have $\text{state-wf}: P, c \vdash_{BV} (h, stk, loc) \ \checkmark$

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

hence $\text{state-correct}: (P_{\text{wf}}), (P_\Phi) \vdash \text{state-to-jvm-state } P\ c\ (h, stk, loc) \ \checkmark$

by (*simp add: bv-conform-def*)

with $P\text{-wf jvm-exec } s$

have $\text{trg-state-correct}: (P_{\text{wf}}), (P_\Phi) \vdash (\text{None}, h', \text{frs}') \ \checkmark$

by $-(\text{rule } BV\text{-correct-1}, (\text{fastforce simp: exec-1-iff})+)$

from *sem-step* $c\ s\ \text{prog}$ have $\text{prealloc: preallocated } h$

by (*auto elim: sem.cases*)

simp: bv-conform-def correct-state-def hconf-def)

from *state-correct* obtain $Ts\ T\ m\!xs\ m\!xl\ is\ xt$

where $\text{sees-M}: (P_{\text{wf}}) \vdash C\ \text{sees } M: Ts \rightarrow T = (m\!xs, m\!xl, is, xt) \text{ in } C$

by (*clarsimp simp: bv-conform-def correct-state-def*)

with *state-correct*

have $pc < \text{length } is$

```

    by (auto dest: sees-method-fun
        simp: bv-conform-def correct-state-def)
with  $P$ -wf sees- $M$  have
    applicable:  $\text{app}_i(is ! pc, (P_{wf}), pc, m\acute{x}s, T, ST, LT)$ 
    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  $\langle pc < \text{length } is \rangle$ 
have wt-instrs:  $P_{wf}, T, m\acute{x}s, \text{length } is, xt \vdash is ! pc, pc :: (P_{\Phi}) C M$ 
    by  $-(\text{drule wt-jvm-prog-impl-wt-instr, fastforce+})$ 
with applicable
have effect:  $\forall succ \in \text{set } (\text{succs } (is ! pc) (ST, LT) pc).$ 
 $(P_{wf}) \vdash \lfloor \text{eff}_i(is ! pc, (P_{wf}), ST, LT) \rfloor \leq' (P_{\Phi}) C M ! succ \wedge succ < \text{length } is$ 
    apply clarsimp
    apply (erule-tac  $x=(succ, \lfloor \text{eff}_i(is ! pc, (P_{wf}), ST, LT) \rfloor)$  in ballE)
    by (erule-tac  $x=(succ, \lfloor \text{eff}_i(is ! pc, (P_{wf}), ST, LT) \rfloor)$  in ballE, clarsimp+)
with  $P$ -wf sees- $M$  last- $c$   $v$ -cs
have  $v$ -cs-succ:
 $\forall succ \in \text{set } (\text{succs } (is ! pc) (ST, LT) pc). \text{valid-callstack } (P, C0, M0) ((C, M, succ) \# cs)$ 
    by  $-(\text{rule ballI,}$ 
        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  $s=(h', \text{update-stk stk frs}', \text{update-loc loc frs'})$ 
        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  $\exists n' \text{ as.}$ 
    CFG.path sourcenode targetnode (valid-edge prog) n as  $n' \wedge$ 
    transfers (CFG.kinds kind as)  $s = s' \wedge$ 
    preds (CFG.kinds kind as)  $s \wedge$  identifies  $n' c'$ 
proof
    show  $\exists \text{ as. CFG.path sourcenode targetnode (valid-edge prog) n as } (- c', \text{None}$ 
 $-) \wedge$ 
    transfers (CFG.kinds kind as)  $s = s' \wedge$ 
    preds (CFG.kinds kind as)  $s \wedge$ 
    identifies  $(- c', \text{None } -) c'$ 
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  $\in$  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 -),
     $\uparrow(\lambda s. \text{exec-instr (instrs-of } (P_{wf} \ C \ M \ ! \ pc) \ P \ s \ (\text{length } cs) \ (\text{stkLength } P \ C$ 
 $M \ pc) \ 0 \ 0),$ 
     $(- (C, M, \text{Suc } pc) \# cs, \text{None } -))$ 
    (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]  $(- c', \text{None}$ 
 $-)$ 
    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  $\langle \text{nat} < \text{length } LT \rangle$ 
    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, fast-
force+)
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 -),
  ↑(λs. exec-instr (instrs-of (Pwf) 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 -),
  ↑(λs. exec-instr (instrs-of (Pwf) 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)√,
    (- (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  $v-exec-edge:valid-edge\ prog\ ((- (C,M,pc)\#cs, [(c', True)]\ -),$ 
   $\uparrow(\lambda(h, stk, loc).$ 
     $(h,$ 
       $stk((length\ cs', (stkLength\ P\ C'\ M'\ pc') - 1) :=$ 
         $Addr\ (addr-of-sys-xcpt\ OutOfMemory)),$ 
       $loc)$ 
     $),$ 
     $(- c', None\ -))$ 
  (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 (Pwf) 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'
  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)✓,
  (- (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  $-(\text{rule } \textit{find-handler-loc-fun-eq'} [\textit{of } P - h (C, M, pc) \# cs \textit{ stk loc}],$ 
     $\textit{auto simp: split-beta hd-stks})$ 
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  $(\textit{auto elim!: sem.cases})$ 
moreover obtain stk'' loc'' frs'' where frs' = (stk'', loc'', C', M', pc') \# frs''
  and cs' = framestack-to-callstack frs'' using calculation
  by  $(\textit{cases frs', fastforce+})$ 
ultimately
have update-stk stk frs' =
   $\textit{stk}((\textit{length cs'}, \textit{stkLength } P \textit{ C' M' pc' - Suc } 0) := \textit{Addr} (\textit{addr-of-sys-xcpt}$ 
NullPointer))
  using c s c' True Cons prog Getfield trg-state-correct wt ST jvm-exec
prealloc stk'
  by  $-(\text{rule } \textit{find-handler-stk-fun-eq'} [\textit{of } P - h (C, M, pc) \# cs - \textit{loc h}],$ 
     $\textit{auto dest!: list-all2-lengthD}$ 
     $\textit{simp: hd-stks split-beta framestack-to-callstack-def}$ 
     $\textit{correct-state-def})$ 
with stk' have stk':
   $\textit{stk}' =$ 
   $\textit{stk}((\textit{length cs'}, \textit{stkLength } P \textit{ C' M' pc' - Suc } 0) := \textit{Addr} (\textit{addr-of-sys-xcpt}$ 
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)] -),
   $\uparrow(\lambda(h, \textit{stk}, \textit{loc}).$ 
   $(h,$ 
   $\textit{stk}((\textit{length cs'}, (\textit{stkLength } P \textit{ C' M' pc' - Suc } 0) - 1) :=$ 
   $\textit{Addr} (\textit{addr-of-sys-xcpt NullPointer})),$ 
   $\textit{loc})$ 
   $),$ 
   $(- \textit{c'}, \textit{None} -))$ 
  (is valid-edge prog ?e2)
apply  $(\textit{auto simp del: exec.simps find-handler-for.simps})$ 
  apply  $(\textit{rule JCFG-Getfield-Exc-Update, fastforce+})$ 
  apply  $(\textit{simp only: cs'-f2c-frs'})$ 
  apply  $(\textit{fastforce intro!: JCFG-Getfield-Exc-Pred})$ 
  apply  $(\textit{fastforce intro!: JCFG-Getfield-Exc-Update})$ 
  by  $(\textit{simp only: cs'-f2c-frs'})$ 
with v-pred-edge (identifies n c) c c' Nil
have JVM-CFG-Interpret.path prog n [?e1, ?e2] (- c', None -)
  by  $-(\text{rule } \textit{JVM-CFG-Interpret.path.Cons-path},$ 
     $\textit{rule JVM-CFG-Interpret.path.Cons-path},$ 
     $\textit{rule JVM-CFG-Interpret.path.empty-path},$ 
     $\textit{auto simp: JVM-CFG-Interpret.valid-node-def, fastforce+})$ 

```

```

moreover from Getfield  $c\ c'\ s\ s'\ loc'\ stk'\ prog\ True\ jvm-exec$ 
   $\langle loc' = loc \rangle\ wt\ ST$ 
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  $s\ s'\ c\ prog\ prealloc\ wt\ \langle length\ ST > 0 \rangle$ 
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  $c'\ v\text{-}cs\ v\text{-}cs\text{-}f2c\text{-}frs'$ 
have v-pred-edge: valid-edge  $prog\ ((- (C, M, pc) \# cs, None\ -),$ 
   $(\lambda(h, stk, loc).\ stk(length\ cs,\ stkLength\ P\ C\ M\ pc - 1) \neq Null) \checkmark,$ 
   $(- (C, M, pc) \# cs, [(c', False)]\ -))$ 
  (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  $prog\ c'\ Getfield\ v\text{-}cs\text{-}succ\ sees\text{-}M$ 
have v-exec-edge: valid-edge  $prog\ ((- (C, M, pc) \# cs, [(c', False)]\ -),$ 
   $\uparrow(\lambda s.\ exec\text{-}instr\ (instrs\text{-}of\ (P_{wf}\ C\ M\ !\ pc)\ P\ s\ (length\ cs)\ (stkLength\ P$ 
   $C\ M\ pc)\ 0\ 0),$ 
   $(- (C, M, Suc\ pc) \# cs, None\ -))$ 
  (is valid-edge  $prog\ ?e2$ )
  by (fastforce intro: JCFG-Getfield-Normal-Update)
with v-pred-edge  $\langle identifies\ n\ c \rangle\ c\ c'$ 
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)√,
  (- (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 split-if-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 (Pwf) 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  $\neg$  cast-ok (Pwf) Cl h (stk(length cs, length ST - Suc 0)))
  case True
  with Checkcast sem-step s s' c prog prealloc wt  $\langle$ length ST > 0 $\rangle$ 
  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  $\langle$ length ST > 0 $\rangle$  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 -),
    ( $\lambda$ (h, stk, loc).  $\neg$  cast-ok (Pwf) Cl h (stk(length cs, stkLength P C M pc -
Suc 0)))) $\vee$ ,
    ((- (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)] -),
     $\uparrow$ id,
    (-Exit-))
    (is valid-edge prog ?e2)
  by (fastforce intro: JCFG-Checkcast-Exc-Exit)
  with v-pred-edge  $\langle$ identifies n c $\rangle$  c c' Nil
  have JVM-CFG-Interpret.path prog n [?e1, ?e2] (-Exit-)
    by  $\neg$ (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 split-if-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'':
  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 (Pwf) Cl h (stk(length cs, stkLength P C M pc - Suc
0)))), ✓,
  (- (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' split-if-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: split-if-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' split-if-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 )✓,
  (- (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' split-if-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 split-if-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 split-if-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 Pwf (cname-of h (the-Addr (stk (length cs, length ST -
Suc n')))) M')
  by simp
from c wt s state-correct
have (Pwf), h ⊢ stks (length ST) (λa. 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: split-if-asm)
  by (induct STn, auto simp: nth-Cons' split: split-if-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 split-if-eq1 nth-Cons' ST)
with trg-state-correct
obtain Ts' T' mb' where D-sees-M': (Pwf) ⊢ 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: Pwf, h ⊢ stk (length cs, length STn + length STr) #
  stks (length STn + length STr) (λa. 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) (λa. stk (length cs, a))) ! length STn =
  stk (length cs, length STr)
  by (auto simp: nth-Cons' ST)
with stk-wt
have Pwf, h ⊢ stk (length cs, length STr) :≤ Class D'
  by (drule-tac P=conf (Pwf) h and p=length STn in list-all2-nthD,

```

```

      auto simp: nth-append)
with False ST wt
have subD': (Pwf) ⊢ (cname-of h (the-Addr (stk (length cs, length ST -
Suc n')))) ≤* D'
  by (cases stk (length cs, length ST), 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':
    (Pwf) ⊢ (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) (λa. stk (length cs, a)))) @
    replicate mxl arbitrary
  by (auto simp: split-beta split-if-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 ST) - (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-max.inf-absorb1 min-max.inf-absorb2 min-max.sup-absorb1
      min-max.sup-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 (Pwf)
      (cname-of h (the-Addr(stk'(length cs, stkLength P C M pc - Suc
n'))))) M'
    ) = D
  )✓,
  (- (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)] -),
  ↑(λs.
    exec-instr (instrs-of (Pwf) 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 -),
   $\uparrow(\lambda s. \text{exec-instr Return } P \ s$ 
     $(\text{Suc } (\text{length } cs')) (\text{stkLength } P \ C \ M \ pc) (\text{stkLength } P \ D \ M' (\text{Suc } pc'))$ 
0),
   $(- (D,M',\text{Suc } pc')\#cs',\text{None } -))$ 
  (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  $-(\text{simp},$ 
    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 framstack-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 -),
   $\uparrow(\lambda s. \text{exec-instr } (\text{instrs-of } (P_{wf}) \ C \ M \ ! \ pc) \ P \ s \ (\text{length } cs) (\text{stkLength } P \ C$ 
M pc) 0 0),
   $(- (C,M,\text{Suc } pc)\#cs,\text{None } -))$ 
  (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  $-(\text{simp},$ 
    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 frametack-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 -),
  ↑(λs. exec-instr (instrs-of (Pwf) 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)√,
    (- (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)√,
    (- (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 -),
  ↑(λs. exec-instr (instrs-of (Pwf) 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-stks 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')
)∨,
(- (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 split-if-eq1)
from  $\langle (C', M', pc') \# cs' = framestack-to-callstack\ frs' \rangle\ Cons$ 
have  $framestack-to-callstack\ frs' = c'$ 
      by simp
with  $Cons\ Throw\ v-cs\ v-cs-f2c-frs'\ v-pred-edge$ 
have v-exec-edge:
       $valid-edge\ prog\ ((- (C, M, pc) \# cs, [(c', True)] -),$ 
       $\uparrow(\lambda(h, stk, loc).$ 
       $(h,$ 
       $stk((length\ cs', stkLength\ P\ C'\ M'\ pc' - 1) :=$ 
       $if\ (stk(length\ cs, stkLength\ P\ C\ M\ pc - 1) = Null)$ 
       $then\ Addr\ (addr-of-sys-xcpt\ NullPointer)$ 
       $else\ (stk(length\ cs, stkLength\ P\ C\ M\ pc - 1)))),$ 
       $loc)$ 
       $),$ 
       $(- c', None -))$ 
      (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  $\wedge$ 
      find-handler-for P NullPointer ((C,M,pc)#cs) = c')  $\vee$ 
    (stk(length cs, stkLength P C M pc - 1)  $\neq$  Null  $\wedge$ 
      find-handler-for P (cname-of h (the-Addr(stk(length cs, stkLength P C
M pc - 1))))
    ((C,M,pc)#cs) = c')
  ) $\vee$ ,
  (- (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)] -),
     $\uparrow id$ ,
    (-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  $\neg$ (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 split-if-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/NonInterferenceWhile
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.