

# Verified Construction of Static Single Assignment Form

Sebastian Ullrich      Denis Lohner

June 24, 2019

## Abstract

We define a functional variant of the static single assignment (SSA) form construction algorithm described by Braun et al. [2], which combines simplicity and efficiency. The definition is based on a general, abstract control flow graph representation using Isabelle locales. We prove that the algorithm’s output is semantically equivalent to the input according to a small-step semantics, and that it is in minimal SSA form for the common special case of reducible inputs. We then show the satisfiability of the locale assumptions by giving instantiations for a simple While language. Furthermore, we use a generic instantiation based on typedefs in order to extract ML code and replace the unverified SSA construction algorithm of the CompCertSSA project [1] with it.

## Contents

<b>1</b>	<b>Prelude</b>	<b>2</b>
1.1	Miscellaneous Lemmata . . . . .	2
1.2	Serial Relations . . . . .	8
1.3	Mapping Extensions . . . . .	9
<b>2</b>	<b>SSA Representation</b>	<b>13</b>
2.1	Inductive Graph Paths . . . . .	13
2.2	Domination . . . . .	27
2.3	CFG . . . . .	38
2.4	SSA CFG . . . . .	40
2.5	Bundling of CFG and Equivalent SSA CFG . . . . .	52
<b>3</b>	<b>Minimality</b>	<b>59</b>
<b>4</b>	<b>SSA Construction</b>	<b>83</b>
4.1	CFG to SSA CFG . . . . .	83
4.2	Inductive Removal of Trivial Phi Functions . . . . .	100

<b>5</b>	<b>Proof of Semantic Equivalence</b>	<b>120</b>
<b>6</b>	<b>Code Generation</b>	<b>124</b>
6.1	While Combinator Extensions . . . . .	124
6.2	Code Equations for SSA Construction . . . . .	135
6.3	Locales Transfer Rules . . . . .	149
6.4	Code Equations for SSA Minimization . . . . .	164
6.5	Generic Code Extraction Based on typedefs . . . . .	193
6.5.1	Instantiation for a Simple While Language . . . . .	209

## 1 Prelude

### 1.1 Miscellaneous Lemmata

```

theory FormalSSA-Misc
imports Main HOL-Library.Sublist
begin

lemma length-1-last-hd: length ns = 1  $\implies$  last ns = hd ns
  by (metis One-nat-def append.simps(1) append-butlast-last-id diff-Suc-Suc diff-zero
length-0-conv length-butlast list.sel(1) zero-neq-one)

lemma not-in-butlast[simp]:  $\llbracket x \in \text{set } ys; x \notin \text{set } (\text{butlast } ys) \rrbracket \implies x = \text{last } ys$ 
  apply (cases ys = [])
  apply simp
  apply (subst(asm) append-butlast-last-id[symmetric])
  by (simp-all del:append-butlast-last-id)

lemma in-set-butlastI:  $x \in \text{set } xs \implies x \neq \text{last } xs \implies x \in \text{set } (\text{butlast } xs)$ 
  by (metis append-butlast-last-id append-is-Nil-conv list.distinct(1) rotate1.simps(2)
set-ConsD set-rotate1 split-list)

lemma butlast-strict-prefix:  $xs \neq [] \implies \text{strict-prefix } (\text{butlast } xs) xs$ 
  by (metis append-butlast-last-id strict-prefixI')

lemma set-tl:  $\text{set } (\text{tl } xs) \subseteq \text{set } xs$ 
  by (metis set-mono-suffix suffix-tl)

lemma in-set-tlD[elim]:  $x \in \text{set } (\text{tl } xs) \implies x \in \text{set } xs$ 
  using set-tl[of xs] by auto

lemma suffix-unsnoc:
  assumes suffix xs ys  $xs \neq []$ 
  obtains x where  $xs = \text{butlast } xs@[x]$   $ys = \text{butlast } ys@[x]$ 
  by (metis append-butlast-last-id append-is-Nil-conv assms(1) assms(2) last-appendR
suffix-def)

lemma prefix-split-first:

```

**assumes**  $x \in \text{set } xs$   
**obtains**  $as$  **where**  $\text{prefix } (as@[x]) \ xs$  **and**  $x \notin \text{set } as$   
**proof** *atomize-elim*  
**from** *assms* **obtain**  $as \ bs$  **where**  $xs = as@x\#bs \wedge x \notin \text{set } as$  **by** (*atomize-elim*,  
*rule split-list-first*)  
**thus**  $\exists as. \text{prefix } (as@[x]) \ xs \wedge x \notin \text{set } as$  **by**  $-(\text{rule } exI[\text{where } x = as], \text{auto})$   
**qed**

**lemma** *in-prefix[elim]*:  
**assumes**  $\text{prefix } xs \ ys$  **and**  $x \in \text{set } xs$   
**shows**  $x \in \text{set } ys$   
**using** *assms* **by** (*auto elim!:prefixE*)

**lemma** *strict-prefix-butlast*:  
**assumes**  $\text{prefix } xs \ (\text{butlast } ys) \ ys \neq []$   
**shows** *strict-prefix*  $xs \ ys$   
**using** *assms* **unfolding** *append-butlast-last-id[symmetric]* **by** (*auto simp add:less-le*  
*butlast-strict-prefix prefix-order.le-less-trans*)

**lemma** *prefix-tl-subset*:  $\text{prefix } xs \ ys \implies \text{set } (tl \ xs) \subseteq \text{set } (tl \ ys)$   
**by** (*metis Nil-tl prefix-bot.bot.extremum prefix-def set-mono-prefix tl-append2*)

**lemma** *suffix-tl-subset*:  $\text{suffix } xs \ ys \implies \text{set } (tl \ xs) \subseteq \text{set } (tl \ ys)$   
**by** (*metis append-Nil suffix-def set-mono-suffix suffix-tl suffix-order.order-trans*  
*tl-append2*)

**lemma** *set-tl-append'*:  $\text{set } (tl \ (xs \ @ \ ys)) \subseteq \text{set } (tl \ xs) \cup \text{set } ys$   
**by** (*metis list.sel(2) order-refl set-append set-mono-suffix suffix-tl tl-append2*)

**lemma** *last-in-tl*:  $\text{length } xs > 1 \implies \text{last } xs \in \text{set } (tl \ xs)$   
**by** (*metis One-nat-def diff-Suc-Suc last-in-set last-tl length-tl less-numeral-extra(4)*  
*list.size(3) zero-less-diff*)

**lemma** *concat-join*:  $xs \neq [] \implies ys \neq [] \implies \text{last } xs = \text{hd } ys \implies \text{butlast } xs@ys =$   
 $xs@tl \ ys$   
**by** (*induction xs, auto*)

**lemma** *fold-induct[case-names Nil Cons]*:  $P \ s \implies (\bigwedge x \ s. x \in \text{set } xs \implies P \ s \implies$   
 $P \ (f \ x \ s)) \implies P \ (\text{fold } f \ xs \ s)$   
**by** (*rule fold-invariant [where Q =  $\lambda x. x \in \text{set } xs$ ]*) *simp*

**lemma** *fold-union-elem*:  
**assumes**  $x \in \text{fold } (\cup) \ xss \ xs$   
**obtains**  $ys$  **where**  $x \in ys \ ys \in \text{set } xss \cup \{xs\}$   
**using** *assms*  
**by** (*induction rule:fold-induct*) *auto*

**lemma** *fold-union-elemI*:  
**assumes**  $x \in ys \ ys \in \text{set } xss \cup \{xs\}$

**shows**  $x \in \text{fold } (\cup) \text{ } xss \text{ } xs$   
**using** *assms*  
**by** (*metis Sup-empty Sup-insert Sup-set-fold Un-insert-right UnionI ccpo-Sup-singleton fold-simps(2) list.simps(15)*)

**lemma** *fold-union-elimI'*:  
**assumes**  $x \in xs \vee (\exists xs \in \text{set } xss. x \in xs)$   
**shows**  $x \in \text{fold } (\cup) \text{ } xss \text{ } xs$   
**using** *assms*  
**using** *fold-union-elimI* **by** *fastforce*

**lemma** *fold-union-finite[intro!]*:  
**assumes**  $\text{finite } xs \ \forall xs \in \text{set } xss. \text{finite } xs$   
**shows**  $\text{finite } (\text{fold } (\cup) \text{ } xss \text{ } xs)$   
**using** *assms* **by**  $-$  (*rule fold-invariant, auto*)

**lemma** *in-set-zip-map*:  
**assumes**  $(x,y) \in \text{set } (\text{zip } xs \ (\text{map } f \text{ } ys))$   
**obtains**  $y'$  **where**  $(x,y') \in \text{set } (\text{zip } xs \text{ } ys) \ f \ y' = y$   
**proof**  $-$   
**from** *assms* **obtain**  $i$  **where**  $x = xs \ ! \ i \ y = \text{map } f \ \text{ys} \ ! \ i \ i < \text{length } xs \ i < \text{length } ys$  **by** (*auto simp:set-zip*)  
**thus** *thesis* **by**  $-$  (*rule that[of ys ! i], auto simp:set-zip*)  
**qed**

**lemma** *dom-comp-subset*:  $g \ ' \ \text{dom } (f \circ g) \subseteq \text{dom } f$   
**by** (*auto simp add:dom-def*)

**lemma** *finite-dom-comp*:  
**assumes**  $\text{finite } (\text{dom } f) \ \text{inj-on } g \ (\text{dom } (f \circ g))$   
**shows**  $\text{finite } (\text{dom } (f \circ g))$   
**proof** (*rule finite-imageD*)  
**have**  $g \ ' \ \text{dom } (f \circ g) \subseteq \text{dom } f$  **by** *auto*  
**with** *assms(1)* **show**  $\text{finite } (g \ ' \ \text{dom } (f \circ g))$  **by**  $-$  (*rule finite-subset*)  
**qed** (*simp add:assms(2)*)

**lemma** *the1-list*:  $\exists!x \in \text{set } xs. P \ x \implies (\text{THE } x. x \in \text{set } xs \wedge P \ x) = \text{hd } (\text{filter } P \ xs)$   
**proof** (*induction xs*)  
**case** (*Cons y xs*)  
**let**  $?Q = \lambda xs \ x. x \in \text{set } xs \wedge P \ x$   
**from** *Cons.prem*s **obtain**  $x$  **where**  $x: ?Q \ (y\#xs) \ x$  **by** *auto*  
**have**  $x = \text{hd } (\text{filter } P \ (y\#xs))$   
**proof** (*cases x=y*)  
**case** *True*  
**with**  $x$  **show** *?thesis* **by** *auto*  
**next**  
**case** *False*  
**with** *Cons.prem*s  $x$  **have**  $1: \exists!x. x \in \text{set } xs \wedge P \ x$  **by** *auto*

**hence** (*THE*  $x. x \in \text{set } xs \wedge P x = x$  **using**  $x$  *False* **by** – (rule *the1-equality*, *auto*)  
**also from** 1 **have** (*THE*  $x. x \in \text{set } xs \wedge P x = \text{hd } (\text{filter } P \text{ } xs)$  **by** (rule *Cons.IH*)  
**finally show** *?thesis* **using** *False*  $x$  *Cons.prem*s **by** *auto*  
**qed**  
**thus** *?case* **using**  $x$  **by** – (rule *the1-equality*[*OF Cons.prem*s], *auto*)  
**qed** *auto*

**lemma** *set-zip-leftI*:  
**assumes**  $\text{length } xs = \text{length } ys$   
**assumes**  $y \in \text{set } ys$   
**obtains**  $x$  **where**  $(x,y) \in \text{set } (\text{zip } xs \text{ } ys)$   
**proof** –  
**from** *assms*(2) **obtain**  $i$  **where**  $y = ys ! i$   $i < \text{length } ys$  **by** (*metis in-set-conv-nth*)  
**with** *assms*(1) **show** *thesis* **by** – (rule *that*[*of xs ! i*], *auto simp add:set-zip*)  
**qed**

**lemma** *butlast-idx*:  
**assumes**  $y \in \text{set } (\text{butlast } xs)$   
**obtains**  $i$  **where**  $xs ! i = y$   $i < \text{length } xs - 1$   
**apply** *atomize-elim*  
**using** *assms* **proof** (*induction xs arbitrary:y*)  
**case** (*Cons*  $x$   $xs$ )  
**from** *Cons.prem*s **have**[*simp*]:  $xs \neq []$  **by** (*simp split:if-split-asm*)  
**show** *?case*  
**proof** (*cases*  $y = x$ )  
**case** *True*  
**show** *?thesis* **by** (rule *exI*[**where**  $x=0$ ], *simp-all add:True*)  
**next**  
**case** *False*  
**with** *Cons.prem*s **have**  $y \in \text{set } (\text{butlast } xs)$  **by** *simp*  
**from** *Cons.IH*[*OF this*] **obtain**  $i$  **where**  $y = xs ! i$  **and**  $i < \text{length } xs - 1$  **by** *auto*  
**thus** *?thesis* **by** – (rule *exI*[**where**  $x=\text{Suc } i$ ], *simp*)  
**qed**  
**qed** *simp*

**lemma** *butlast-idx'*:  
**assumes**  $xs ! i = y$   $i < \text{length } xs - 1$   $\text{length } xs > 1$   
**shows**  $y \in \text{set } (\text{butlast } xs)$   
**using** *assms* **proof** (*induction xs arbitrary:i*)  
**case** (*Cons*  $x$   $xs$ )  
**show** *?case*  
**proof** (*cases*  $i$ )  
**case** 0  
**with** *Cons.prem*s(1,3) **show** *?thesis* **by** *simp*  
**next**  
**case** (*Suc*  $j$ )

**with** *Cons.prem*s(1)[*symmetric*] *Cons.prem*s(2,3) **have**  $y \in \text{set } (\text{butlast } xs)$   
**by** – (*rule Cons.IH*, *auto*)  
**with** *Cons.prem*s(3) **show** *?thesis* **by** *simp*  
**qed**  
**qed** *simp*

**lemma** *card-eq-1-singleton*:  
**assumes**  $\text{card } A = 1$   
**obtains**  $x$  **where**  $A = \{x\}$   
**using** *assms[simplified]* **by** – (*drule card-eq-SucD*, *auto*)

**lemma** *set-take-two*:  
**assumes**  $\text{card } A \geq 2$   
**obtains**  $x y$  **where**  $x \in A$   $y \in A$   $x \neq y$   
**proof** –  
**from** *assms* **obtain**  $k$  **where**  $\text{card } A = \text{Suc } (\text{Suc } k)$   
**by** (*auto simp: le-iff-add*)  
**from** *card-eq-SucD[OF this]* **obtain**  $x B$  **where**  $x: A = \text{insert } x B$   $x \notin B$   $\text{card } B = \text{Suc } k$  **by** *auto*  
**from** *card-eq-SucD[OF this(3)]* **obtain**  $y$  **where**  $y: y \in B$  **by** *auto*  
**from**  $x y$  **show** *?thesis* **by** – (*rule that[of x y]*, *auto*)  
**qed**

**lemma** *singleton-list-hd-last*:  $\text{length } xs = 1 \implies \text{hd } xs = \text{last } xs$   
**by** (*metis One-nat-def impossible-Cons last.simps length-0-conv less-nat-zero-code list.sel(1) nat-less-le neq-Nil-conv not-less-eq-eq*)

**lemma** *distinct-hd-tl*:  $\text{distinct } xs \implies \text{hd } xs \notin \text{set } (\text{tl } xs)$   
**by** (*metis distinct.simps(2) hd-Cons-tl in-set-member list.sel(2) member-rec(2)*)

**lemma** *set-mono-strict-prefix*:  $\text{strict-prefix } xs \ ys \implies \text{set } xs \subseteq \text{set } (\text{butlast } ys)$   
**by** (*metis append-butlast-last-id strict-prefixE strict-prefix-simps(1) prefix-snoc set-mono-prefix*)

**lemma** *set-butlast-distinct*:  $\text{distinct } xs \implies \text{set } (\text{butlast } xs) \cap \{\text{last } xs\} = \{\}$   
**by** (*metis append-butlast-last-id butlast.simps(1) distinct-append inf-bot-right inf-commute list.set(1) set-simps(2)*)

**lemma** *disjoint-elem[elim]*:  $A \cap B = \{\} \implies x \in A \implies x \notin B$  **by** *auto*

**lemma** *prefix-butlastD[elim]*:  $\text{prefix } xs (\text{butlast } ys) \implies \text{prefix } xs \ ys$   
**using** *strict-prefix-butlast* **by** *fastforce*

**lemma** *butlast-prefix*:  $\text{prefix } xs \ ys \implies \text{prefix } (\text{butlast } xs) (\text{butlast } ys)$   
**by** (*induction xs ys rule: list-induct2'*; *auto*)

**lemma** *hd-in-butlast*:  $\text{length } xs > 1 \implies \text{hd } xs \in \text{set } (\text{butlast } xs)$   
**by** (*metis butlast.simps(2) dual-order.strict-iff-order hd-Cons-tl hd-in-set length-greater-0-conv length-tl less-le-trans list.distinct(1) list.sel(1) neq0-conv zero-less-diff*)

**lemma** *nonsimple-length-gt-1*:  $xs \neq [] \implies hd\ xs \neq last\ xs \implies length\ xs > 1$   
**by** (*metis length-0-conv less-one nat-neq-iff singleton-list-hd-last*)

**lemma** *set-hd-tl*:  $xs \neq [] \implies set\ [hd\ xs] \cup set\ (tl\ xs) = set\ xs$   
**by** (*metis inf-sup-aci(5) rotate1-hd-tl set-append set-rotate1*)

**lemma** *fold-update-conv*:  
 $fold\ (\lambda n\ m.\ m(n \mapsto g\ n))\ xs\ m\ x =$   
*(if (x ∈ set xs) then Some (g x) else m x)*  
**by** (*induction xs arbitrary: m*) *auto*

**lemmas** *removeAll-le = length-removeAll-less-eq*

**lemmas** *removeAll-less [intro] = length-removeAll-less*

**lemma** *removeAll-induct*:  
**assumes**  $\bigwedge xs.\ (\bigwedge x.\ x \in set\ xs \implies P\ (removeAll\ x\ xs)) \implies P\ xs$   
**shows**  $P\ xs$   
**by** (*induct xs rule:length-induct, rule:assms*) *auto*

**lemma** *The-Min*:  $Ex1\ P \implies The\ P = Min\ \{x.\ P\ x\}$   
**apply** (*rule the-equality*)  
**apply** (*metis (mono-tags) Min.infinite Min-in Min-singleton all-not-in-conv finite-subset insert-iff mem-Collect-eq subsetI*)  
**by** (*metis (erased, hide-lams) Least-Min Least-equality Set.set-insert ex-in-conv finite.emptyI finite-insert insert-iff mem-Collect-eq order-refl*)

**lemma** *The-Max*:  $Ex1\ P \implies The\ P = Max\ \{x.\ P\ x\}$   
**apply** (*rule the-equality*)  
**apply** (*metis (mono-tags) Max.infinite Max-in Max-singleton all-not-in-conv finite-subset insert-iff mem-Collect-eq subsetI*)  
**by** (*metis Max-singleton Min-singleton Nitpick.Ex1-unfold The-Min the-equality*)

**lemma** *set-sorted-list-of-set-remove [simp]*:  
 $set\ (sorted-list-of-set\ (Set.remove\ x\ A)) = Set.remove\ x\ (set\ (sorted-list-of-set\ A))$   
**unfolding** *Set.remove-def*  
**by** (*cases finite A; simp*)

**lemma** *set-minus-one*:  $\llbracket v \neq v'; v' \in set\ vs \rrbracket \implies set\ vs - \{v'\} \subseteq \{v\} \iff set\ vs = \{v'\} \vee set\ vs = \{v, v'\}$   
**by** *auto*

**lemma** *set-single-hd*:  $set\ vs = \{v\} \implies hd\ vs = v$   
**by** (*induction vs; auto*)

**lemma** *set-double-filter-hd*:  $\llbracket set\ vs = \{v, v'\}; v \neq v' \rrbracket \implies hd\ [v' \leftarrow vs . v' \neq v] = v'$   
**apply** (*induction vs*)

```

apply simp
apply auto
apply (case-tac  $v \in \text{set } vs$ )
  prefer 2
  apply (subgoal-tac  $\text{set } vs = \{v\}$ )
    prefer 2
    apply fastforce
  apply (clarsimp simp: set-single-hd)
by fastforce

```

```

lemma map-option-the:  $x = \text{map-option } f \ y \implies x \neq \text{None} \implies \text{the } x = f \ (\text{the } y)$ 
  by (auto simp: map-option-case split: option.split prod.splits)

```

**end**

## 1.2 Serial Relations

A serial relation on a finite carrier induces a cycle.

```

theory Serial-Rel
imports Main
begin

```

```

definition serial-on  $A \ r \longleftrightarrow (\forall x \in A. \exists y \in A. (x,y) \in r)$ 

```

```

lemmas serial-onI = serial-on-def[THEN iffD2, rule-format]

```

```

lemmas serial-onE = serial-on-def[THEN iffD1, rule-format, THEN bexE]

```

```

fun iterated-serial-on :: 'a set  $\Rightarrow$  'a rel  $\Rightarrow$  'a  $\Rightarrow$  nat  $\Rightarrow$  'a where

```

```

  iterated-serial-on  $A \ r \ x \ 0 = x$ 
| iterated-serial-on  $A \ r \ x \ (\text{Suc } n) = (\text{SOME } y. y \in A \wedge (\text{iterated-serial-on } A \ r \ x \ n, y) \in r)$ 

```

```

lemma iterated-serial-on-linear:  $\text{iterated-serial-on } A \ r \ x \ (n+m) = \text{iterated-serial-on } A \ r \ (\text{iterated-serial-on } A \ r \ x \ n) \ m$ 

```

```

by (induction  $m$ ) auto

```

```

lemma iterated-serial-on-in-A:

```

```

  assumes serial-on  $A \ r \ a \in A$ 

```

```

  shows iterated-serial-on  $A \ r \ a \ n \in A$ 

```

```

proof (induct  $n$ )

```

```

  case (Suc  $n$ )

```

```

  thus ?case

```

```

  using assms(1, 2) by (subst iterated-serial-on.simps(2)) (rule someI2-ex, auto
elim: serial-onE)

```

```

qed (simp add:assms(2))

```

```

lemma iterated-serial-on-in-power:

```

```

  assumes serial-on  $A \ r \ a \in A$ 

```

```

  shows  $(a, \text{iterated-serial-on } A \ r \ a \ n) \in r \wedge \wedge n$ 

```

```

proof (induct  $n$ )

```



**case** (*Suc n*)  
**moreover obtain** *b* **where** (*iterated-serial-on A r a n, b*)  $\in r$  *b*  $\in A$   
**using** *iterated-serial-on-in-A[OF assms, of n]* *assms(1)* **by** – (*rule serial-onE*)  
**ultimately show** *?case* **by** (*subst iterated-serial-on.simps(2)*) (*rule someI2-ex, auto*)  
**qed** *simp*

**lemma** *trancl-powerI*:  $a \in R \wedge n > 0 \implies a \in R^+$   
**by** (*auto simp:trancl-power*)

**theorem** *serial-on-finite-cycle*:  
**assumes** *serial-on A r A  $\neq \{\}$  finite A*  
**obtains** *a* **where**  $a \in A$  (*a, a*)  $\in r^+$   
**proof** –  
**from** *assms(2)* **obtain** *a* **where**  $a \in A$  **by** *auto*  
**let** *?f = iterated-serial-on A r a*  
**from** *a* **have** *range ?f  $\subseteq A$*  **using** *assms(1)* **by** (*auto intro: iterated-serial-on-in-A*)  
**with** *assms(3)* **have**  $\exists m \in UNIV. \neg \text{finite } \{n \in UNIV. ?f n = ?f m\}$   
**by** – (*rule pigeonhole-infinite, auto simp: finite-subset*)  
**then obtain** *n m* **where**  $?f m = ?f n$  **and** [*simp*]:  $n < m$   
**by** (*metis (mono-tags, lifting) finite-nat-set-iff-bounded mem-Collect-eq not-less-eq*)  
**hence**  $?f n = \text{iterated-serial-on } A \text{ r } (?f n) (m - n)$   
**using** *iterated-serial-on-linear[of A r a n m - n]* **by** (*auto simp:less-imp-le-nat*)  
**also have**  $(?f n, \text{iterated-serial-on } A \text{ r } (?f n) (m - n)) \in r \wedge (m - n)$   
**by** (*rule iterated-serial-on-in-power[OF assms(1)], rule iterated-serial-on-in-A[OF assms(1) a]*)  
**finally show** *?thesis*  
**by** – (*rule that[of ?f n], rule iterated-serial-on-in-A[OF assms(1) a], rule trancl-powerI, auto*)  
**qed**

**end**

### 1.3 Mapping Extensions

Some lifted definition on mapping and efficient implementations.

**theory** *Mapping-Exts*  
**imports** *HOL-Library.Mapping FormalSSA-Misc*  
**begin**

**lift-definition** *mapping-delete-all* ::  $('a \Rightarrow \text{bool}) \Rightarrow ('a, 'b) \text{ mapping} \Rightarrow ('a, 'b) \text{ mapping}$

**is**  $\lambda P m x. \text{if } (P x) \text{ then None else } m x$  .

**lift-definition** *map-keys* ::  $('a \Rightarrow 'b) \Rightarrow ('a, 'c) \text{ mapping} \Rightarrow ('b, 'c) \text{ mapping}$

**is**  $\lambda f m x. \text{if } f - \{x\} \neq \{\} \text{ then } m (\text{THE } k. f - \{x\} = \{k\}) \text{ else None}$  .

**lift-definition** *map-values* ::  $('a \Rightarrow 'b \Rightarrow 'c \text{ option}) \Rightarrow ('a, 'b) \text{ mapping} \Rightarrow ('a, 'c) \text{ mapping}$

**is**  $\lambda f m x. \text{Option.bind } (m x) (f x)$  .

**lift-definition** *restrict-mapping* ::  $('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow ('a, 'b) \text{ mapping}$

**is**  $\lambda f. \text{restrict-map } (\text{Some} \circ f)$  .  
**lift-definition**  $\text{mapping-add} :: ('a, 'b) \text{ mapping} \Rightarrow ('a, 'b) \text{ mapping} \Rightarrow ('a, 'b) \text{ mapping}$   
**is**  $(++)$  .

**definition**  $\text{mmap} = \text{Mapping.map id}$

**lemma**  $\text{lookup-map-keys}: \text{Mapping.lookup } (\text{map-keys } f \ m) \ x = (\text{if } f \ -' \ \{x\} \neq \{\})$   
 $\text{then } \text{Mapping.lookup } m \ (\text{THE } k. \ f \ -' \ \{x\} = \{k\}) \ \text{else } \text{None}$   
**apply**  $\text{transfer ..}$

**lemma**  $\text{Mapping-Mapping-lookup} [\text{simp}, \text{code-unfold}]: \text{Mapping.Mapping } (\text{Mapping.lookup } m) = m$  **by**  $\text{transfer simp}$   
**declare**  $\text{Mapping.lookup.abs-eq}[\text{simp}]$

**lemma**  $\text{Mapping-eq-lookup}: m = m' \longleftrightarrow \text{Mapping.lookup } m = \text{Mapping.lookup } m'$   
**by**  $\text{transfer simp}$

**lemma**  $\text{map-of-map-if-conv}$ :  
 $\text{map-of } (\text{map } (\lambda k. (k, f \ k)) \ xs) \ x = (\text{if } (x \in \text{set } xs) \ \text{then } \text{Some } (f \ x) \ \text{else } \text{None})$   
**by**  $(\text{clarsimp simp: map-of-map-restrict})$

**lemma**  $\text{Mapping-lookup-map}: \text{Mapping.lookup } (\text{Mapping.map } f \ g \ m) \ a = \text{map-option } g \ (\text{Mapping.lookup } m \ (f \ a))$   
**by**  $\text{transfer simp}$

**lemma**  $\text{Mapping-lookup-map-default}: \text{Mapping.lookup } (\text{Mapping.map-default } k \ d \ f \ m) \ k' = (\text{if } k = k')$   
 $\text{then } (\text{Some } \circ f) \ (\text{case } \text{Mapping.lookup } m \ k \ \text{of } \text{None} \Rightarrow d \ | \ \text{Some } x \Rightarrow x)$   
 $\text{else } \text{Mapping.lookup } m \ k'$   
**unfolding**  $\text{Mapping.map-default-def } \text{Mapping.default-def}$   
**by**  $\text{transfer auto}$

**lemma**  $\text{Mapping-lookup-mapping-add}: \text{Mapping.lookup } (\text{mapping-add } m1 \ m2) \ k =$   
 $\text{case-option } (\text{Mapping.lookup } m1 \ k) \ \text{Some } (\text{Mapping.lookup } m2 \ k)$   
**by**  $\text{transfer (simp add: map-add-def)}$

**lemma**  $\text{Mapping-lookup-map-values}: \text{Mapping.lookup } (\text{map-values } f \ m) \ k =$   
 $\text{Option.bind } (\text{Mapping.lookup } m \ k) \ (f \ k)$   
**by**  $\text{transfer simp}$

**lemma**  $\text{lookup-fold-update} [\text{simp}]: \text{Mapping.lookup } (\text{fold } (\lambda n. \ \text{Mapping.update } n \ (g \ n)) \ xs \ m) \ x$   
 $= (\text{if } (x \in \text{set } xs) \ \text{then } \text{Some } (g \ x) \ \text{else } \text{Mapping.lookup } m \ x)$   
**by**  $\text{transfer (rule fold-update-conv)}$

**lemma**  $\text{mapping-eq-iff}: m1 = m2 \longleftrightarrow (\forall k. \ \text{Mapping.lookup } m1 \ k = \text{Mapping.lookup } m2 \ k)$   
**by**  $\text{transfer auto}$

**lemma** *lookup-delete*:  $\text{Mapping.lookup } (\text{Mapping.delete } k \ m) \ k' = (\text{if } k = k' \ \text{then } \text{None} \ \text{else } \text{Mapping.lookup } m \ k')$

**by** *transfer auto*

**lemma** *keys-map-values*:  $\text{Mapping.keys } (\text{map-values } f \ m) = \text{Mapping.keys } m - \{k \in \text{Mapping.keys } m. \ f \ k \ (\text{the } (\text{Mapping.lookup } m \ k)) = \text{None}\}$

**by** *transfer (auto simp add: bind-eq-Some-conv)*

**lemma** *map-default-eq*:  $\text{Mapping.map-default } k \ v \ f \ m = m \longleftrightarrow (\exists v. \ \text{Mapping.lookup } m \ k = \text{Some } v \wedge f \ v = v)$

**apply** (*clarsimp simp: Mapping.map-default-def Mapping.default-def*)

**by** *transfer' (auto simp: fun-eq-iff split: if-splits)*

**lemma** *lookup-update-cases*:  $\text{Mapping.lookup } (\text{Mapping.update } k \ v \ m) \ k' = (\text{if } k=k' \ \text{then } \text{Some } v \ \text{else } \text{Mapping.lookup } m \ k')$

**by** (*cases k=k', simp-all add: Mapping.lookup-update Mapping.lookup-update-neq*)

**end**

**theory** *RBT-Mapping-Exts*

**imports**

*Mapping-Exts*

*HOL-Library.RBT-Mapping*

*HOL-Library.RBT-Set*

**begin**

**lemma** *restrict-mapping-code* [*code*]:

$\text{restrict-mapping } f \ (\text{RBT-Set.Set } r) = \text{RBT-Mapping.Mapping } (\text{RBT.map } (\lambda a \ -. \ f \ a) \ r)$

**by** *transfer (auto simp: RBT-Set.Set-def restrict-map-def)*

**lemma** *map-keys-code*:

**assumes** *inj f*

**shows**  $\text{map-keys } f \ (\text{RBT-Mapping.Mapping } t) = \text{RBT.fold } (\lambda x \ v \ m. \ \text{Mapping.update } (f \ x) \ v \ m) \ t \ \text{Mapping.empty}$

**proof**–

**have**[*simp*]:  $\bigwedge x. \ \{y. \ f \ y = f \ x\} = \{x\}$

**using** *assms* **by** (*auto simp: inj-on-def*)

**have**[*simp*]:  $\text{distinct } (\text{map fst } (\text{rev } (\text{RBT.entries } t)))$

**apply** (*subst rev-map[symmetric]*)

**apply** (*subst distinct-rev*)

**apply** (*rule distinct-entries*)

**done**

{  
  **fix** *k v*

```

fix xs :: ('a × 'c) list
assume asm: distinct (map fst xs)
hence
  (k, v) ∈ set xs ⇒ Some v = foldr (λ(x, v) m. m(f x ↦ v)) xs Map.empty
(f k)
  k ∉ fst ` set xs ⇒ None = foldr (λ(x, v) m. m(f x ↦ v)) xs Map.empty (f
k)
  ∧x. x ∉ f ` UNIV ⇒ None = foldr (λ(x, v) m. m(f x ↦ v)) xs Map.empty
x
  by (induction xs) (auto simp: image-def dest!: injD[OF assms])
}
note a = this[unfolded foldr-conv-fold, where xs3=rev -, simplified]

```

```

show ?thesis
unfolding RBT.fold-fold
apply (transfer fixing: t f)
apply (rule ext)
apply (auto simp: vimage-def)
apply (rename-tac x)
apply (case-tac RBT.lookup t x)
apply (auto simp: lookup-in-tree[symmetric] intro!: a(2))[1]
apply (auto dest!: lookup-in-tree[THEN iffD1] intro!: a(1))[1]
apply (rule a(3); auto)
done
qed

```

```

lemma map-values-code [code]:
  map-values f (RBT-Mapping.Mapping t) = RBT.fold (λx v m. case (f x v) of
None ⇒ m | Some v' ⇒ Mapping.update x v' m) t Mapping.empty
proof -
  {fix xs m
  assume distinct (map fst (xs::('a × 'c) list))
  hence fold (λp m. case f (fst p) (snd p) of None ⇒ m | Some v' ⇒ m(fst p
↦ v')) xs m
  = m ++ (λx. Option.bind (map-of xs x) (f x))
  by (induction xs arbitrary: m) (auto intro: rev-image-eqI split: bind-split op-
tion.splits simp: map-add-def fun-eq-iff)
  }
note bind-map-of-fold = this
show ?thesis
unfolding RBT.fold-fold
apply (transfer fixing: t f)
apply (simp add: split-def)
apply (rule bind-map-of-fold [of RBT.entries t Map.empty, simplified, symmet-
ric])
  using RBT.distinct-entries distinct-map by auto
qed

```

```

lemma [code-unfold]: set (RBT.keys t) = RBT-Set.Set (RBT.map (λ- . ()) t)

```

**by** (*auto simp: RBT-Set.Set-def RBT.keys-def-alt RBT.lookup-in-tree elim: rev-image-eqI*)

**lemma** *mmap-rbt-code* [*code*]: *mmap f (RBT-Mapping.Mapping t) = RBT-Mapping.Mapping (RBT.map (λ-. f) t)*  
**unfolding** *mmap-def* **by** *transfer auto*

**lemma** *mapping-add-code* [*code*]: *mapping-add (RBT-Mapping.Mapping t1) (RBT-Mapping.Mapping t2) = RBT-Mapping.Mapping (RBT.union t1 t2)*  
**by** *transfer (simp add: lookup-union)*

**end**

## 2 SSA Representation

### 2.1 Inductive Graph Paths

We extend the Graph framework with inductively defined paths. We adopt the convention of separating locale definitions into assumption-less base locales.

**theory** *Graph-path imports*

*FormalSSA-Misc*

*Dijkstra-Shortest-Path.GraphSpec*

*CAVA-Automata.Digraph-Basic*

**begin**

**hide-const** *Omega-Words-Fun.prefix Omega-Words-Fun.suffix*

**type-synonym** (*'n, 'ed*) *edge = ('n × 'ed × 'n)*

**definition** *getFrom* :: (*'n, 'ed*) *edge* ⇒ *'n* **where**  
*getFrom* ≡ *fst*

**definition** *getData* :: (*'n, 'ed*) *edge* ⇒ *'ed* **where**  
*getData* ≡ *fst* ∘ *snd*

**definition** *getTo* :: (*'n, 'ed*) *edge* ⇒ *'n* **where**  
*getTo* ≡ *snd* ∘ *snd*

**lemma** *get-edge-simps* [*simp*]:

*getFrom* (*f,d,t*) = *f*

*getData* (*f,d,t*) = *d*

*getTo* (*f,d,t*) = *t*

**by** (*simp-all add: getFrom-def getData-def getTo-def*)

Predecessors of a node.

**definition** *pred* :: (*'v, 'w*) *graph* ⇒ *'v* ⇒ (*'v × 'w*) *set*  
**where** *pred G v* ≡ {(*v',w*). (*v',w,v*) ∈ *edges G*}

**lemma** *pred-finite* [*simp, intro*]: *finite (edges G) ⇒ finite (pred G v)*  
**unfolding** *pred-def*

by (rule finite-subset[**where**  $B=(\lambda(v,w,v'). (v,w))\text{'edges } G]$ ] force+

**lemma** *pred-empty*[simp]: *pred empty*  $v = \{\}$  **unfolding** *empty-def* *pred-def* **by** *auto*

**lemma** (in *valid-graph*) *pred-subset*: *pred*  $G v \subseteq V \times UNIV$   
**unfolding** *pred-def* **using** *E-valid*  
**by** (*force*)

**type-synonym** ( $'V, 'W, 'σ, 'G$ ) *graph-pred-it* =  
 $'G \Rightarrow 'V \Rightarrow ('V \times 'W, 'σ)$  *set-iterator*

**locale** *graph-pred-it-defs* =  
**fixes** *pred-list-it* ::  $'G \Rightarrow 'V \Rightarrow ('V \times 'W, ('V \times 'W) \text{ list})$  *set-iterator*  
**begin**  
**definition** *pred-it*  $g v \equiv it\text{-to-it } (pred\text{-list-it } g v)$   
**end**

**locale** *graph-pred-it* = *graph*  $\alpha$  *invar* + *graph-pred-it-defs* *pred-list-it*  
**for**  $\alpha :: 'G \Rightarrow ('V, 'W)$  *graph* **and** *invar* **and**  
*pred-list-it* ::  $'G \Rightarrow 'V \Rightarrow ('V \times 'W, ('V \times 'W) \text{ list})$  *set-iterator* +  
**assumes** *pred-list-it-correct*:  
 $invar\ g \Longrightarrow set\text{-iterator } (pred\text{-list-it } g v) (pred (\alpha\ g) v)$

**begin**  
**lemma** *pred-it-correct*:  
 $invar\ g \Longrightarrow set\text{-iterator } (pred\text{-it } g v) (pred (\alpha\ g) v)$   
**unfolding** *pred-it-def*  
**apply** (*rule it-to-it-correct*)  
**by** (*rule pred-list-it-correct*)

**lemma** *pi-pred-it*[*icf-proper-iteratorI*]:  
 $proper\text{-it } (pred\text{-it } S v) (pred\text{-it } S v)$   
**unfolding** *pred-it-def*  
**by** (*intro icf-proper-iteratorI*)

**lemma** *pred-it-proper*[*proper-it*]:  
 $proper\text{-it}' (\lambda S. pred\text{-it } S v) (\lambda S. pred\text{-it } S v)$   
**apply** (*rule proper-it'I*)  
**by** (*rule pi-pred-it*)

**end**

**record** ( $'V, 'W, 'G$ ) *graph-ops* = ( $'V, 'W, 'G$ ) *GraphSpec.graph-ops* +  
*gop-pred-list-it* ::  $'G \Rightarrow 'V \Rightarrow ('V \times 'W, ('V \times 'W) \text{ list})$  *set-iterator*

**lemma** (in *graph-pred-it*) *pred-it-is-iterator*[*refine-transfer*]:  
 $invar\ g \Longrightarrow set\text{-iterator } (pred\text{-it } g v) (pred (\alpha\ g) v)$   
**by** (*rule pred-it-correct*)

**locale** *StdGraphDefs* = *GraphSpec.StdGraphDefs* *ops*

```

+ graph-pred-it-defs gop-pred-list-it ops
for ops :: ('V,'W,'G,'m) graph-ops-scheme
begin
  abbreviation pred-list-it where pred-list-it  $\equiv$  gop-pred-list-it ops
end

locale StdGraph = StdGraphDefs + org.StdGraph +
  graph-pred-it  $\alpha$  invar pred-list-it

locale graph-path-base =
  graph-nodes-it-defs  $\lambda g$ . foldri ( $\alpha n$  g) +
  graph-pred-it-defs  $\lambda g$  n. foldri (inEdges' g n)
for
   $\alpha e$  :: 'g  $\Rightarrow$  ('node  $\times$  'edgeD  $\times$  'node) set and
   $\alpha n$  :: 'g  $\Rightarrow$  'node list and
  invar :: 'g  $\Rightarrow$  bool and
  inEdges' :: 'g  $\Rightarrow$  'node  $\Rightarrow$  ('node  $\times$  'edgeD) list
begin

  definition inEdges :: 'g  $\Rightarrow$  'node  $\Rightarrow$  ('node  $\times$  'edgeD  $\times$  'node) list
  where inEdges g n  $\equiv$  map ( $\lambda(f,d).$  (f,d,n)) (inEdges' g n)

  definition predecessors :: 'g  $\Rightarrow$  'node  $\Rightarrow$  'node list where
    predecessors g n  $\equiv$  map getFrom (inEdges g n)

  definition successors :: 'g  $\Rightarrow$  'node  $\Rightarrow$  'node list where
    successors g m  $\equiv$  [n . n  $\leftarrow$   $\alpha n$  g, m  $\in$  set (predecessors g n)]

  declare predecessors-def [code]

  declare [[inductive-internals]]
  inductive path :: 'g  $\Rightarrow$  'node list  $\Rightarrow$  bool
    for g :: 'g
  where
    empty-path[intro]: [n  $\in$  set ( $\alpha n$  g); invar g]  $\Longrightarrow$  path g [n]
    | Cons-path[intro]: [path g ns; n'  $\in$  set (predecessors g (hd ns))]  $\Longrightarrow$  path g
    (n'#ns)

  definition path2 :: 'g  $\Rightarrow$  'node  $\Rightarrow$  'node list  $\Rightarrow$  'node  $\Rightarrow$  bool (-  $\vdash$  ---- $\rightarrow$ -
  [51,0,0,51] 80) where
    path2 g n ns m  $\equiv$  path g ns  $\wedge$  n = hd ns  $\wedge$  m = last ns

  abbreviation  $\alpha$  g  $\equiv$  ( $\setminus$ nodes = set ( $\alpha n$  g), edges =  $\alpha e$  g)
end

locale graph-path =

```

```

graph-path-base  $\alpha e \alpha n$  invar inEdges' +
graph  $\alpha$  invar +
ni: graph-nodes-it  $\alpha$  invar  $\lambda g$ . foldri ( $\alpha n g$ ) +
pi: graph-pred-it  $\alpha$  invar  $\lambda g n$ . foldri (inEdges'  $g n$ )
for
 $\alpha e :: 'g \Rightarrow ('node \times 'edgeD \times 'node)$  set and
 $\alpha n :: 'g \Rightarrow 'node$  list and
invar :: ' $g \Rightarrow bool$  and
inEdges' :: ' $g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)$  list
begin
lemma  $\alpha n$ -correct: invar  $g \implies set (\alpha n g) \supseteq getFrom \text{' } \alpha e g \cup getTo \text{' } \alpha e g$ 
  by (frule valid) (auto dest: valid-graph.E-validD)

lemma  $\alpha n$ -distinct: invar  $g \implies distinct (\alpha n g)$ 
  by (frule ni.nodes-list-it-correct)
  (metis foldri-cons-id iterate-to-list-correct iterate-to-list-def)

lemma inEdges-correct':
  assumes invar  $g$ 
  shows  $set (inEdges g n) = (\lambda(f,d). (f,d,n)) \text{' } (pred (\alpha g) n)$ 
proof -
  from iterate-to-list-correct [OF pi.pred-list-it-correct [OF assms], of  $n$ ]
  show ?thesis
  by (auto intro: rev-image-eqI simp: iterate-to-list-def pred-def inEdges-def)
qed

lemma inEdges-correct [intro!, simp]:
  invar  $g \implies set (inEdges g n) = \{(-, -, t). t = n\} \cap \alpha e g$ 
  by (auto simp: inEdges-correct' pred-def)

lemma in-set- $\alpha n I1$  [intro]:  $[[invar g; x \in getFrom \text{' } \alpha e g] \implies x \in set (\alpha n g)$ 
  using  $\alpha n$ -correct by blast
lemma in-set- $\alpha n I2$  [intro]:  $[[invar g; x \in getTo \text{' } \alpha e g] \implies x \in set (\alpha n g)$ 
  using  $\alpha n$ -correct by blast

lemma edge-to-node:
  assumes invar  $g$  and  $e \in \alpha e g$ 
  obtains  $getFrom e \in set (\alpha n g)$  and  $getTo e \in set (\alpha n g)$ 
using assms(2)  $\alpha n$ -correct [OF  $\langle invar g \rangle$ ]
  by (cases e) (auto 4 3 intro: rev-image-eqI)

lemma inEdge-to-edge:
  assumes  $e \in set (inEdges g n)$  and invar  $g$ 
  obtains  $eD n'$  where  $(n', eD, n) \in \alpha e g$ 
  using assms by auto

```



**lemma** *edge-to-inEdge*:  
**assumes**  $(n, eD, m) \in \alpha e\ g\ \text{invar}\ g$   
**obtains**  $(n, eD, m) \in \text{set}\ (\text{inEdges}\ g\ m)$   
**using** *assms* **by** *auto*

**lemma** *edge-to-predecessors*:  
**assumes**  $(n, eD, m) \in \alpha e\ g\ \text{invar}\ g$   
**obtains**  $n \in \text{set}\ (\text{predecessors}\ g\ m)$   
**proof** *atomize-elim*  
**from** *assms* **have**  $(n, eD, m) \in \text{set}\ (\text{inEdges}\ g\ m)$  **by** (*rule edge-to-inEdge*)  
**thus**  $n \in \text{set}\ (\text{predecessors}\ g\ m)$  **unfolding** *predecessors-def* **by** (*metis get-edge-simps(1)*  
*image-eqI set-map*)  
**qed**

**lemma** *predecessor-is-node[elim]*:  $\llbracket n \in \text{set}\ (\text{predecessors}\ g\ n') ; \text{invar}\ g \rrbracket \implies n \in \text{set}\ (\alpha n\ g)$   
**unfolding** *predecessors-def* **by** (*fastforce intro: rev-image-eqI simp: getTo-def getFrom-def*)

**lemma** *successor-is-node[elim]*:  $\llbracket n \in \text{set}\ (\text{predecessors}\ g\ n') ; n \in \text{set}\ (\alpha n\ g) ; \text{invar}\ g \rrbracket \implies n' \in \text{set}\ (\alpha n\ g)$   
**unfolding** *predecessors-def* **by** (*fastforce intro: rev-image-eqI*)

**lemma** *successors-predecessors[simp]*:  $n \in \text{set}\ (\alpha n\ g) \implies n \in \text{set}\ (\text{successors}\ g\ m) \longleftrightarrow m \in \text{set}\ (\text{predecessors}\ g\ n)$   
**by** (*auto simp: successors-def predecessors-def*)

**lemma** *path-not-Nil[simp, dest]*:  $\text{path}\ g\ ns \implies ns \neq []$   
**by** (*erule path.cases*) *auto*

**lemma** *path2-not-Nil[simp]*:  $g \vdash n - ns \rightarrow m \implies ns \neq []$   
**unfolding** *path2-def* **by** *auto*

**lemma** *path2-not-Nil2[simp]*:  $\neg g \vdash n - [] \rightarrow m$   
**unfolding** *path2-def* **by** *auto*

**lemma** *path2-not-Nil3[simp]*:  $g \vdash n - ns \rightarrow m \implies \text{length}\ ns \geq 1$   
**by** (*cases ns, auto*)

**lemma** *empty-path2[intro]*:  $\llbracket n \in \text{set}\ (\alpha n\ g) ; \text{invar}\ g \rrbracket \implies g \vdash n - [n] \rightarrow n$   
**unfolding** *path2-def* **by** *auto*

**lemma** *Cons-path2[intro]*:  $\llbracket g \vdash n - ns \rightarrow m ; n' \in \text{set}\ (\text{predecessors}\ g\ n) \rrbracket \implies g \vdash n' - n' \# ns \rightarrow m$   
**unfolding** *path2-def* **by** *auto*

**lemma** *path2-cases*:

```

assumes  $g \vdash n - ns \rightarrow m$ 
obtains (empty-path)  $ns = [n] \ m = n$ 
  | (Cons-path)  $g \vdash hd \ (tl \ ns) - tl \ ns \rightarrow m \ n \in \text{set} \ (\text{predecessors } g \ (hd \ (tl \ ns)))$ 
proof -
from assms have 1:  $path \ g \ ns \ hd \ ns = n \ last \ ns = m$  by (auto simp: path2-def)
from this(1) show thesis
proof cases
  case (empty-path  $n$ )
  with 1 show thesis by - (rule that(1), auto)
next
  case (Cons-path  $ns \ n'$ )
  with 1 show thesis by - (rule that(2), auto simp: path2-def)
qed
qed

```

```

lemma path2-induct[consumes 1, case-names empty-path Cons-path]:
  assumes  $g \vdash n - ns \rightarrow m$ 
  assumes empty:  $invar \ g \Longrightarrow P \ m \ [m] \ m$ 
  assumes Cons:  $\bigwedge ns \ n' \ n. \ g \vdash n - ns \rightarrow m \Longrightarrow P \ n \ ns \ m \Longrightarrow n' \in \text{set} \ (\text{predecessors } g \ n) \Longrightarrow P \ n' \ (n' \# \ ns) \ m$ 
  shows  $P \ n \ ns \ m$ 
using assms(1)
unfolding path2-def
apply -
proof (erule conjE, induction arbitrary: n rule:path.induct)
  case empty-path
  with empty show ?case by simp
next
  case (Cons-path  $ns \ n' \ n''$ )
  hence[simp]:  $n'' = n'$  by simp
  from Cons-path Cons show ?case unfolding path2-def by auto
qed

```

```

lemma path-invar[intro]:  $path \ g \ ns \Longrightarrow invar \ g$ 
by (induction rule:path.induct)

```

```

lemma path-in- $\alpha n$ [intro]:  $\llbracket path \ g \ ns; n \in \text{set} \ ns \rrbracket \Longrightarrow n \in \text{set} \ (\alpha n \ g)$ 
by (induct ns arbitrary: n rule:path.induct) auto

```

```

lemma path2-in- $\alpha n$ [elim]:  $\llbracket g \vdash n - ns \rightarrow m; l \in \text{set} \ ns \rrbracket \Longrightarrow l \in \text{set} \ (\alpha n \ g)$ 
unfolding path2-def by auto

```

```

lemma path2-hd-in- $\alpha n$ [elim]:  $g \vdash n - ns \rightarrow m \Longrightarrow n \in \text{set} \ (\alpha n \ g)$ 
unfolding path2-def by auto

```

```

lemma path2-tl-in- $\alpha n$ [elim]:  $g \vdash n - ns \rightarrow m \Longrightarrow m \in \text{set} \ (\alpha n \ g)$ 
unfolding path2-def by auto

```

```

lemma path2-forget-hd[simp]:  $g \vdash n - ns \rightarrow m \Longrightarrow g \vdash hd \ ns - ns \rightarrow m$ 

```

**unfolding** *path2-def* **by** *simp*

**lemma** *path2-forget-last*[*simp*]:  $g \vdash n - ns \rightarrow m \implies g \vdash n - ns \rightarrow \text{last } ns$   
**unfolding** *path2-def* **by** *simp*

**lemma** *path-hd*[*dest*]:  $\text{path } g (n \# ns) \implies \text{path } g [n]$   
**by** (*rule empty-path*, *auto elim:path.cases*)

**lemma** *path-by-tail*[*intro*]:  $\llbracket \text{path } g (n \# n' \# ns); \text{path } g (n' \# ns) \implies \text{path } g (n' \# ms) \rrbracket$   
 $\implies \text{path } g (n \# n' \# ms)$   
**by** (*rule path.cases*) *auto*

**lemma**  *$\alpha n$ -in- $\alpha n E$*  [*elim*]:  
  **assumes**  $(n, e, m) \in \alpha e g$  **and** *invar*  $g$   
  **obtains**  $n \in \text{set } (\alpha n g)$  **and**  $m \in \text{set } (\alpha n g)$   
  **using** *assms*  
  **by** (*auto elim: edge-to-node*)

**lemma** *path-split*:  
  **assumes**  $\text{path } g (ns @ m \# ns')$   
  **shows**  $\text{path } g (ns @ [m]) \text{ path } g (m \# ns')$   
**proof** –  
  **from** *assms* **show**  $\text{path } g (ns @ [m])$   
  **proof** (*induct ns*)  
    **case** (*Cons n ns*)  
    **thus** ?*case* **by** (*cases ns*) *auto*  
  **qed** *auto*  
  **from** *assms* **show**  $\text{path } g (m \# ns')$   
  **proof** (*induct ns*)  
    **case** (*Cons n ns*)  
    **thus** ?*case* **by** (*auto elim:path.cases*)  
  **qed** *auto*  
**qed**

**lemma** *path2-split*:  
  **assumes**  $g \vdash n - ns @ n' \# ns' \rightarrow m$   
  **shows**  $g \vdash n - ns @ [n'] \rightarrow n' g \vdash n' - n' \# ns' \rightarrow m$   
**using** *assms* **unfolding** *path2-def* **by** (*auto intro:path-split iff:hd-append*)

**lemma** *elem-set-implies-elem-tl-app-cons*[*simp*]:  $x \in \text{set } xs \implies x \in \text{set } (\text{tl } (ys @ y \# xs))$   
**by** (*induction ys arbitrary: y; auto*)

**lemma** *path2-split-ex*:  
  **assumes**  $g \vdash n - ns \rightarrow m$   $x \in \text{set } ns$   
  **obtains**  $ns_1 ns_2$  **where**  $g \vdash n - ns_1 \rightarrow x g \vdash x - ns_2 \rightarrow m$   $ns = ns_1 @ \text{tl } ns_2$   $ns = \text{butlast } ns_1 @ ns_2$   
  **proof** –  
  **from** *assms*(2) **obtain**  $ns_1 ns_2$  **where**  $ns = ns_1 @ x \# ns_2$  **by** *atomize-elim* (*rule split-list*)

**with** *assms*[*simplified this*] **show** *thesis*  
**by**  $-(\text{rule } \text{that}, \text{auto } \text{dest}:\text{path2-split}(1) \text{ path2-split}(2) \text{ intro}:\text{suffixI})$   
**qed**

**lemma** *path2-split-ex'*:  
**assumes**  $g \vdash n - ns \rightarrow m \ x \in \text{set } ns$   
**obtains**  $ns_1 \ ns_2$  **where**  $g \vdash n - ns_1 \rightarrow x \ g \vdash x - ns_2 \rightarrow m \ ns = \text{butlast } ns_1 @ ns_2$   
**using** *assms* **by** (*rule path2-split-ex*)

**lemma** *path-snoc*:  
**assumes**  $\text{path } g \ (ns@[n]) \ n \in \text{set } (\text{predecessors } g \ m)$   
**shows**  $\text{path } g \ (ns@[n,m])$   
**using** *assms*(1) **proof** (*induction ns*)  
**case** *Nil*  
**hence**  $1: n \in \text{set } (\alpha n \ g) \ \text{invar } g$  **by** *auto*  
**with** *assms*(2) **have**  $m \in \text{set } (\alpha n \ g)$  **by** *auto*  
**with** 1 **have**  $\text{path } g \ [m]$  **by** *auto*  
**with** *assms*(2) **show** *?case* **by** *auto*  
**next**  
**case** (*Cons l ns*)  
**hence**  $1: \text{path } g \ (ns \ @ \ [n]) \ \wedge \ l \in \text{set } (\text{predecessors } g \ (\text{hd } (ns@[n])))$  **by**  $-(\text{cases } g \ (l \ \# \ ns) \ @ \ [n] \ \text{rule}:\text{path.cases}, \ \text{auto})$   
**hence**  $\text{path } g \ (ns \ @ \ [n,m])$  **by** (*auto intro:Cons.IH*)  
**with** 1 **have**  $\text{path } g \ (l \ \# \ ns \ @ \ [n,m])$  **by**  $-(\text{rule } \text{Cons-path}, \ \text{assumption}, \ \text{cases } ns, \ \text{auto})$   
**thus** *?case* **by** *simp*  
**qed**

**lemma** *path2-snoc[elim]*:  
**assumes**  $g \vdash n - ns \rightarrow m \ m \in \text{set } (\text{predecessors } g \ m')$   
**shows**  $g \vdash n - ns @ [m'] \rightarrow m'$   
**proof**–  
**from** *assms*(1) **have**  $1: ns \neq []$  **by** *auto*  
  
**have**  $\text{path } g \ ((\text{butlast } ns) \ @ \ [\text{last } ns, \ m'])$   
**using** *assms* **unfolding** *path2-def* **by**  $-(\text{rule } \text{path-snoc}, \ \text{auto})$   
**hence**  $\text{path } g \ ((\text{butlast } ns \ @ \ [\text{last } ns]) \ @ \ [m'])$  **by** *simp*  
**with** 1 **have**  $\text{path } g \ (ns \ @ \ [m'])$  **by** *simp*  
**thus** *?thesis*  
**using** *assms* **unfolding** *path2-def* **by** *auto*  
**qed**

**lemma** *path-unsnoc*:  
**assumes**  $\text{path } g \ ns \ \text{length } ns \geq 2$   
**obtains**  $\text{path } g \ (\text{butlast } ns) \ \wedge \ \text{last } (\text{butlast } ns) \in \text{set } (\text{predecessors } g \ (\text{last } ns))$   
**using** *assms*  
**proof** (*atomize-elim, induction ns*)  
**case** (*Cons-path ns n*)  
**show** *?case*

```

proof (cases 2 ≤ length ns)
  case True
    hence [simp]: hd (butlast ns) = hd ns by (cases ns, auto)
    have 0: n#butlast ns = butlast (n#ns) using True by auto
    have 1: n ∈ set (predecessors g (hd (butlast ns))) using Cons-path by simp
    from True have path g (butlast ns) using Cons-path by auto
    hence path g (n#butlast ns) using 1 by auto
    hence path g (butlast (n#ns)) using 0 by simp
    moreover
      from Cons-path True have last (butlast ns) ∈ set (predecessors g (last ns))
by simp
    hence last (butlast (n # ns)) ∈ set (predecessors g (last (n # ns)))
      using True by (cases ns, auto)
    ultimately show ?thesis by auto
  next
    case False
    thus ?thesis
    proof (cases ns)
      case Nil
      thus ?thesis using Cons-path by -(rule ccontr, auto elim:path.cases)
    next
      case (Cons n' ns')
      hence [simp]: ns = [n'] using False by (cases ns', auto)
      have path g [n,n'] using Cons-path by auto
      thus ?thesis using Cons-path by auto
    qed
  qed
qed auto

```

```

lemma path2-unsnoc:
  assumes g ⊢ n-ns→m length ns ≥ 2
  obtains g ⊢ n-butlast ns→last (butlast ns) last (butlast ns) ∈ set (predecessors
g m)
  using assms unfolding path2-def by (metis append-butlast-last-id hd-append2
path-not-Nil path-unsnoc)

```

```

lemma path2-rev-induct[consumes 1, case-names empty snoc]:
  assumes g ⊢ n-ns→m
  assumes empty: n ∈ set (αn g) ⇒ P n [n] n
  assumes snoc: ∧ns m' m. g ⊢ n-ns→m' ⇒ P n ns m' ⇒ m' ∈ set
(predecessors g m) ⇒ P n (ns@[m]) m
  shows P n ns m
  using assms(1) proof (induction arbitrary:m rule:length-induct)
    case (1 ns)
    show ?case
  proof (cases length ns ≥ 2)
    case False
    thus ?thesis
  proof (cases ns)

```

```

    case Nil
    thus ?thesis using 1(2) by auto
  next
    case (Cons n' ns')
    with False have ns' = [] by (cases ns', auto)
    with Cons 1(2) have n' = n m = n unfolding path2-def by auto
    with Cons ⟨ns' = []⟩ 1(2) show ?thesis by (auto intro:empty)
  qed
next
  case True
  let ?ns' = butlast ns
  let ?m' = last ?ns'
  from 1(2) have m: m = last ns unfolding path2-def by auto
  from True 1(2) obtain ns': g ⊢ n-?ns'→?m' ?m' ∈ set (predecessors g m)
by -(rule path2-unsnoc)
  with True 1.IH have P n ?ns' ?m' by auto
  with ns' have P n (?ns'@[m]) m by (auto intro!: snoc)
  with m 1(2) show ?thesis by auto
qed
qed

lemma path2-hd[elim, dest?]: g ⊢ n-ns→m ⇒ n = hd ns
unfolding path2-def by simp

lemma path2-hd-in-ns[elim]: g ⊢ n-ns→m ⇒ n ∈ set ns
unfolding path2-def by auto

lemma path2-last[elim, dest?]: g ⊢ n-ns→m ⇒ m = last ns
unfolding path2-def by simp

lemma path2-last-in-ns[elim]: g ⊢ n-ns→m ⇒ m ∈ set ns
unfolding path2-def by auto

lemma path-app[elim]:
  assumes path g ns path g ms last ns = hd ms
  shows path g (ns@tl ms)
using assms by (induction ns rule:path.induct) auto

lemma path2-app[elim]:
  assumes g ⊢ n-ns→m g ⊢ m-ms→l
  shows g ⊢ n-ns@tl ms→l
proof-
  have last (ns @ tl ms) = last ms using assms
  unfolding path2-def
  proof (cases tl ms)
    case Nil
    hence ms = [m] using assms(2) unfolding path2-def by (cases ms, auto)
    thus ?thesis using assms(1) unfolding path2-def by auto
  next

```

```

    case (Cons m' ms')
    from this[symmetric] have ms = hd ms#m'#ms' using assms(2) by auto
    thus ?thesis using assms unfolding path2-def by auto
qed
with assms show ?thesis
  unfolding path2-def by auto
qed

lemma butlast-tl:
  assumes last xs = hd ys xs ≠ [] ys ≠ []
  shows butlast xs@ys = xs@tl ys
  by (metis append.simps(1) append.simps(2) append-assoc append-butlast-last-id
  assms(1) assms(2) assms(3) list.collapse)

lemma path2-app'[elim]:
  assumes g ⊢ n-ns→m g ⊢ m-ms→l
  shows g ⊢ n-butlast ns@ms→l
  proof-
    have butlast ns@ms = ns@tl ms using assms by - (rule butlast-tl, auto
    dest:path2-hd path2-last)
    moreover from assms have g ⊢ n-ns@tl ms→l by (rule path2-app)
    ultimately show ?thesis by simp
  qed

lemma path2-nontrivial[elim]:
  assumes g ⊢ n-ns→m n ≠ m
  shows length ns ≥ 2
  using assms
  by (metis Suc-1 le-antisym length-1-last-hd not-less-eq-eq path2-hd path2-last
  path2-not-Nil3)

lemma simple-path2-aux:
  assumes g ⊢ n-ns→m
  obtains ns' where g ⊢ n-ns'→m distinct ns' set ns' ⊆ set ns length ns' ≤
  length ns
  apply atomize-elim
  using assms proof (induction rule:path2-induct)
    case empty-path
    with assms show ?case by - (rule exI[of - [m]], auto)
  next
    case (Cons-path ns n n')
    then obtain ns' where ns': g ⊢ n'-ns'→m distinct ns' set ns' ⊆ set ns length
    ns' ≤ length ns by auto
    show ?case
    proof (cases n ∈ set ns')
      case False
      with ns' Cons-path(2) show ?thesis by - (rule exI[where x=n#ns'], auto)
    next
      case True

```

**with  $ns'$  obtain  $ns'_1 ns'_2$  where  $split: ns' = ns'_1 @ n \# ns'_2$   $n \notin set\ ns'_2$  by**  
 $-(atomize-elim, rule\ split-list-last)$   
**with  $ns'$  have  $g \vdash n - n \# ns'_2 \rightarrow m$  by**  $-(rule\ path2-split, simp)$   
**with  $split\ ns'$  show  $?thesis$  by**  $-(rule\ exI[where\ x = n \# ns'_2], auto)$   
**qed**  
**qed**

**lemma  $simple-path2$ :**  
**assumes  $g \vdash n - ns \rightarrow m$**   
**obtains  $ns'$  where  $g \vdash n - ns' \rightarrow m$   $distinct\ ns'$   $set\ ns' \subseteq set\ ns$   $length\ ns' \leq$**   
 $length\ ns$   $n \notin set\ (tl\ ns')$   $m \notin set\ (butlast\ ns')$   
**using  $assms$**   
**apply  $(rule\ simple-path2-aux)$**   
**by  $(metis\ append-butlast-last-id\ distinct.simps(2)\ distinct1-rotate\ hd-Cons-tl\ path2-hd$**   
 $path2-last\ path2-not-Nil\ rotate1.simps(2))$

**lemma  $simple-path2-unsnoc$ :**  
**assumes  $g \vdash n - ns \rightarrow m$   $n \neq m$**   
**obtains  $ns'$  where  $g \vdash n - ns' \rightarrow last\ ns'$   $last\ ns' \in set\ (predecessors\ g\ m)$**   
 $distinct\ ns'$   $set\ ns' \subseteq set\ ns$   $m \notin set\ ns'$   
**proof-**  
**obtain  $ns'$  where  $1: g \vdash n - ns' \rightarrow m$   $distinct\ ns'$   $set\ ns' \subseteq set\ ns$   $m \notin set$**   
 $(butlast\ ns')$  **by  $(rule\ simple-path2[OF\ assms(1)])$**   
**with  $assms(2)$  obtain  $2: g \vdash n - butlast\ ns' \rightarrow last\ (butlast\ ns')$   $last\ (butlast$**   
 $ns') \in set\ (predecessors\ g\ m)$  **by**  $-(rule\ path2-unsnoc, auto)$   
**show  $thesis$**   
**proof  $(rule\ that[of\ butlast\ ns'])$**   
**from  $1(3)$  show  $set\ (butlast\ ns') \subseteq set\ ns$  by  $(metis\ in-set-butlastD\ subsetI$**   
 $subset-trans)$   
**qed  $(auto\ simp:1\ 2\ distinct-butlast)$**   
**qed**

**lemma  $path2-split-first-last$ :**  
**assumes  $g \vdash n - ns \rightarrow m$   $x \in set\ ns$**   
**obtains  $ns_1\ ns_3\ ns_2$  where  $ns = ns_1 @ ns_3 @ ns_2$   $prefix\ (ns_1 @ [x])\ ns\ suffix$**   
 $(x \# ns_2)\ ns$   
**and  $g \vdash n - ns_1 @ [x] \rightarrow x$   $x \notin set\ ns_1$**   
**and  $g \vdash x - ns_3 \rightarrow x$**   
**and  $g \vdash x - x \# ns_2 \rightarrow m$   $x \notin set\ ns_2$**   
**proof-**  
**from  $assms(2)$  obtain  $ns_1\ ns'$  where  $1: ns = ns_1 @ x \# ns'$   $x \notin set\ ns_1$  by**  
 $(atomize-elim, rule\ split-list-first)$   
**from  $assms(1)[unfolded\ 1(1)]$  have  $2: g \vdash n - ns_1 @ [x] \rightarrow x$   $g \vdash x - x \# ns' \rightarrow m$**   
**by**  $-(erule\ path2-split, erule\ path2-split)$   
**obtain  $ns_3\ ns_2$  where  $3: x \# ns' = ns_3 @ x \# ns_2$   $x \notin set\ ns_2$  by  $(atomize-elim,$**   
 $rule\ split-list-last, simp)$   
**from  $2(2)[unfolded\ 3(1)]$  have  $4: g \vdash x - ns_3 @ [x] \rightarrow x$   $g \vdash x - x \# ns_2 \rightarrow m$  by**  $-($   
 $erule\ path2-split, erule\ path2-split)$   
**show  $thesis$**



```

proof (rule that[OF - - - 2(1) 1(2) 4 3(2)])
  show  $ns = ns_1 @ (ns_3 @ [x]) @ ns_2$  using 1(1) 3(1) by simp
  show prefix  $(ns_1@[x]) ns$  using 1 by auto
  show suffix  $(x\#ns_2) ns$  using 1 3 by (metis Sublist.suffix-def suffix-order.order-trans)
qed
qed

lemma path2-simple-loop:
  assumes  $g \vdash n - ns \rightarrow n \ n' \in \text{set } ns$ 
  obtains  $ns'$  where  $g \vdash n - ns' \rightarrow n \ n' \in \text{set } ns' \ n \notin \text{set } (tl (butlast ns')) \ \text{set } ns' \subseteq \text{set } ns$ 
  using assms proof (induction length ns arbitrary: ns rule: nat-less-induct)
  case 1
  let  $?ns' = tl (butlast ns)$ 
  show ?case
  proof (cases  $n \in \text{set } ?ns'$ )
  case False
  with 1.prem1(2,3) show ?thesis by - (rule 1.prem1(1), auto)
  next
  case True
  hence 2: length ns > 1 by (cases ns, auto)
  with 1.prem1(2) obtain  $m$  where  $g \vdash n - butlast ns \rightarrow m \ m \in \text{set } (predecessors g n)$  by - (rule path2-unsnoc, auto)
  with True obtain  $m'$  where  $g \vdash m' - ?ns' \rightarrow m \ n \in \text{set } (predecessors g m')$  by - (erule path2-cases, auto)
  with True obtain  $ns_1 \ ns_2$  where split:  $g \vdash m' - ns_1 \rightarrow n \ g \vdash n - ns_2 \rightarrow m \ ?ns' = ns_1 @ tl \ ns_2 \ ?ns' = butlast ns_1 @ ns_2$ 
  by - (rule path2-split-ex)
  have  $ns = butlast ns @ [n]$  using 2 1.prem1(2) by (auto simp: path2-def)
  moreover have  $butlast ns = n \# tl (butlast ns)$  using 2 m(1) by (auto simp: path2-def)
  ultimately have split':  $ns = n \# ns_1 @ tl \ ns_2 @ [n] \ ns = n \# butlast ns_1 @ ns_2 @ [n]$ 
using split(3,4) by auto
  show ?thesis
  proof (cases  $n' \in \text{set } (n \# ns_1)$ )
  case True
  show ?thesis
  proof (rule 1.hyps[rule-format, of -  $n \# ns_1$ ])
  show length  $(n \# ns_1) < \text{length } ns$  using split'(1) by auto
  show  $n' \in \text{set } (n \# ns_1)$  by (rule True)
  qed (auto intro: split(1) m'(2) intro!: 1.prem1(1) simp: split'(1))
  next
  case False
  from False split'(1) 1.prem1(3) have 5:  $n' \in \text{set } (ns_2 @ [n])$  by auto
  show ?thesis
  proof (rule 1.hyps[rule-format, of -  $ns_2 @ [n]$ ])
  show length  $(ns_2 @ [n]) < \text{length } ns$  using split'(2) by auto
  show  $n' \in \text{set } (ns_2 @ [n])$  by (rule 5)
  show  $g \vdash n - ns_2 @ [n] \rightarrow n$  using split(2) m(2) by (rule path2-snoc)

```

qed (auto intro!: 1.premis(1) simp: split'(2))  
 qed  
 qed  
 qed

**lemma path2-split-first-prop:**

assumes  $g \vdash n - ns \rightarrow m \exists x \in \text{set } ns. P x$   
 obtains  $m' ns'$  where  $g \vdash n - ns' \rightarrow m' P m' \forall x \in \text{set } (\text{butlast } ns'). \neg P x \text{ prefix } ns' ns$   
**proof**–  
 obtain  $ns'' n' ns'$  where  $1: ns = ns'' @ n' \# ns' P n' \forall x \in \text{set } ns''. \neg P x$  by  
 – (rule split-list-first-propE[OF assms(2)])  
 with  $assms(1)$  have  $g \vdash n - ns'' @ [n'] \rightarrow n'$  by – (rule path2-split(1), auto)  
 with 1 show thesis by – (rule that, auto)  
 qed

**lemma path2-split-last-prop:**

assumes  $g \vdash n - ns \rightarrow m \exists x \in \text{set } ns. P x$   
 obtains  $n' ns'$  where  $g \vdash n' - ns' \rightarrow m P n' \forall x \in \text{set } (\text{tl } ns'). \neg P x \text{ suffix } ns' ns$   
**proof**–  
 obtain  $ns'' n' ns'$  where  $1: ns = ns'' @ n' \# ns' P n' \forall x \in \text{set } ns'. \neg P x$  by –  
 (rule split-list-last-propE[OF assms(2)])  
 with  $assms(1)$  have  $g \vdash n' - n' \# ns' \rightarrow m$  by – (rule path2-split(2), auto)  
 with 1 show thesis by – (rule that, auto simp: Sublist.suffix-def)  
 qed

**lemma path2-prefix[elim]:**

assumes 1:  $g \vdash n - ns \rightarrow m$   
 assumes 2:  $\text{prefix } (ns' @ [m']) ns$   
 shows  $g \vdash n - ns' @ [m'] \rightarrow m'$   
 using  $assms$  by – (erule prefixE, rule path2-split, simp)

**lemma path2-prefix-ex:**

assumes  $g \vdash n - ns \rightarrow m m' \in \text{set } ns$   
 obtains  $ns'$  where  $g \vdash n - ns' \rightarrow m' \text{ prefix } ns' ns m' \notin \text{set } (\text{butlast } ns')$   
**proof**–  
 from  $assms(2)$  obtain  $ns'$  where  $\text{prefix } (ns' @ [m']) ns m' \notin \text{set } ns'$  by (rule prefix-split-first)  
 with  $assms(1)$  show thesis by – (rule that, auto)  
 qed

**lemma path2-strict-prefix-ex:**

assumes  $g \vdash n - ns \rightarrow m m' \in \text{set } (\text{butlast } ns)$   
 obtains  $ns'$  where  $g \vdash n - ns' \rightarrow m' \text{ strict-prefix } ns' ns m' \notin \text{set } (\text{butlast } ns')$   
**proof**–  
 from  $assms(2)$  obtain  $ns'$  where  $ns': \text{prefix } (ns' @ [m']) (\text{butlast } ns) m' \notin \text{set } ns'$  by (rule prefix-split-first)  
 hence  $\text{strict-prefix } (ns' @ [m']) ns$  using  $assms$  by – (rule strict-prefix-butlast,

```

auto)
  with assms(1) ns'(2) show thesis by - (rule that, auto)
qed

lemma path2-nontriv[elim]:  $\llbracket g \vdash n \text{-} ns \rightarrow m; n \neq m \rrbracket \implies \text{length } ns > 1$ 
  by (metis hd-Cons-tl last-appendR last-snoc length-greater-0-conv length-tl path2-def
path-not-Nil zero-less-diff)

declare path-not-Nil [simp del]
declare path2-not-Nil [simp del]
declare path2-not-Nil3 [simp del]
end

```

## 2.2 Domination

We fix an entry node per graph and use it to define node domination.

```

locale graph-Entry-base = graph-path-base  $\alpha e \ \alpha n$  invar inEdges'
for
   $\alpha e :: 'g \Rightarrow ('node \times 'edgeD \times 'node)$  set and
   $\alpha n :: 'g \Rightarrow 'node$  list and
  invar :: 'g  $\Rightarrow$  bool and
  inEdges' :: 'g  $\Rightarrow 'node \Rightarrow ('node \times 'edgeD)$  list
+
fixes Entry :: 'g  $\Rightarrow 'node$ 
begin
  definition dominates :: 'g  $\Rightarrow 'node \Rightarrow 'node \Rightarrow$  bool where
    dominates g n m  $\equiv m \in \text{set } (\alpha n \ g) \wedge (\forall ns. g \vdash \text{Entry } g \text{-} ns \rightarrow m \longrightarrow n \in \text{set } ns)$ 
  abbreviation strict-dom g n m  $\equiv n \neq m \wedge \text{dominates } g \ n \ m$ 
end

locale graph-Entry = graph-Entry-base  $\alpha e \ \alpha n$  invar inEdges' Entry
+ graph-path  $\alpha e \ \alpha n$  invar inEdges'
for
   $\alpha e :: 'g \Rightarrow ('node \times 'edgeD \times 'node)$  set and
   $\alpha n :: 'g \Rightarrow 'node$  list and
  invar :: 'g  $\Rightarrow$  bool and
  inEdges' :: 'g  $\Rightarrow 'node \Rightarrow ('node \times 'edgeD)$  list and
  Entry :: 'g  $\Rightarrow 'node$ 
+
assumes Entry-in-graph[simp]:  $\text{Entry } g \in \text{set } (\alpha n \ g)$ 
assumes Entry-unreachable:  $\text{invar } g \implies \text{inEdges } g \ (\text{Entry } g) = []$ 
assumes Entry-reaches[intro]:
   $\llbracket n \in \text{set } (\alpha n \ g); \text{invar } g \rrbracket \implies \exists ns. g \vdash \text{Entry } g \text{-} ns \rightarrow n$ 
begin
  lemma Entry-dominates[simp,intro]:  $\llbracket \text{invar } g; n \in \text{set } (\alpha n \ g) \rrbracket \implies \text{dominates } g \ (\text{Entry } g) \ n$ 
  unfolding dominates-def by auto
end

```

**lemma** *Entry-iff-unreachable*[simp]:  
**assumes** *invar g n ∈ set (αn g)*  
**shows** *predecessors g n = [] ↔ n = Entry g*  
**proof** (*rule, rule ccontr*)  
**assume** *predecessors g n = [] n ≠ Entry g*  
**with** *Entry-reaches[OF assms(2,1)]* **show** *False* **by** (*auto elim:simple-path2-unsnoc*)  
**qed** (*auto simp:assms Entry-unreachable predecessors-def*)

**lemma** *Entry-loop*:  
**assumes** *invar g g ⊢ Entry g-ns→Entry g*  
**shows** *ns=[Entry g]*  
**proof** (*cases length ns ≥ 2*)  
**case** *True*  
**with** *assms* **have** *last (butlast ns) ∈ set (predecessors g (Entry g))* **by** - (*rule path2-unsnoc*)  
**with** *Entry-unreachable[OF assms(1)]* **have** *False* **by** (*simp add:predecessors-def*)  
**thus** *?thesis ..*  
**next**  
**case** *False*  
**with** *assms* **show** *?thesis*  
**by** (*metis Suc-leI hd-Cons-tl impossible-Cons le-less length-greater-0-conv numeral-2-eq-2 path2-hd path2-not-Nil*)  
**qed**

**lemma** *simple-Entry-path*:  
**assumes** *invar g n ∈ set (αn g)*  
**obtains** *ns* **where** *g ⊢ Entry g-ns→n* **and** *n ∉ set (butlast ns)*  
**proof** -  
**from** *assms* **obtain** *ns* **where** *p: g ⊢ Entry g-ns→n* **by** - (*atomize-elim, rule Entry-reaches*)  
**with** *p* **obtain** *ns'* **where** *g ⊢ Entry g-ns'→n* *n ∉ set (butlast ns')* **by** - (*rule path2-split-first-last, auto*)  
**thus** *?thesis* **by** (*rule that*)  
**qed**

**lemma** *dominatesI* [*intro*]:  
 $\llbracket m \in \text{set } (\alpha n g); \bigwedge ns. \llbracket g \vdash \text{Entry } g-ns \rightarrow m \rrbracket \implies n \in \text{set } ns \rrbracket \implies \text{dominates } g n m$   
**unfolding** *dominates-def* **by** *simp*

**lemma** *dominatesE*:  
**assumes** *dominates g n m*  
**obtains** *m ∈ set (αn g)* **and**  $\bigwedge ns. g \vdash \text{Entry } g-ns \rightarrow m \implies n \in \text{set } ns$   
**using** *assms* **unfolding** *dominates-def* **by** *auto*

**lemma**[*simp*]: *dominates g n m ⟹ m ∈ set (αn g)* **by** (*rule dominatesE*)

**lemma**[*simp*]:

```

assumes dominates g n m and[simp]: invar g
shows  $n \in \text{set } (\alpha n g)$ 
proof–
  from assms obtain ns where  $g \vdash \text{Entry } g\text{-}ns \rightarrow m$  by atomize-elim (rule
Entry-reaches, auto)
  with assms show ?thesis by (auto elim!:dominatesE)
qed

lemma strict-domE[elim]:
  assumes strict-dom g n m
  obtains  $m \in \text{set } (\alpha n g)$  and  $\bigwedge ns. g \vdash \text{Entry } g\text{-}ns \rightarrow m \implies n \in \text{set } (\text{butlast } ns)$ 
  using assms by (metis dominates-def path2-def path-not-Nil rotate1.simps(2)
set-ConsD set-rotate1 snoc-eq-iff-butlast)

lemma dominates-refl[intro!]:  $[[\text{invar } g; n \in \text{set } (\alpha n g)]] \implies \text{dominates } g n n$ 
by auto

lemma dominates-trans:
  assumes invar g
  assumes part1: dominates g n n'
  assumes part2: dominates g n' n''
  shows dominates g n n''
proof
  from part2 show  $n'' \in \text{set } (\alpha n g)$  by auto

  fix ns :: 'node list
  assume  $p: g \vdash \text{Entry } g\text{-}ns \rightarrow n''$ 
  with part2 have  $n' \in \text{set } ns$  by – (erule dominatesE, auto)
  then obtain as where prefix: prefix (as@[n']) ns by (auto intro:prefix-split-first)
  with p have  $g \vdash \text{Entry } g\text{-}(as@[n']) \rightarrow n''$  by auto
  with part1 have  $n \in \text{set } (as@[n'])$  unfolding dominates-def by auto
  with prefix show  $n \in \text{set } ns$  by auto
qed

lemma dominates-antisymm:
  assumes invar g
  assumes dom1: dominates g n n'
  assumes dom2: dominates g n' n
  shows  $n = n'$ 
proof (rule ccontr)
  assume  $n \neq n'$ 
  from dom2 have  $n \in \text{set } (\alpha n g)$  by auto
  with (invar g) obtain ns where  $p: g \vdash \text{Entry } g\text{-}ns \rightarrow n$  and  $n \notin \text{set } (\text{butlast } ns)$ 
  by (rule simple-Entry-path)
  with dom2 have  $n' \in \text{set } ns$  by – (erule dominatesE, auto)
  then obtain as where prefix: prefix (as@[n']) ns by (auto intro:prefix-split-first)
  with p have  $g \vdash \text{Entry } g\text{-}as@[n'] \rightarrow n'$  by (rule path2-prefix)

```

**with**  $dom1$  **have**  $n \in set (as@[n'])$  **unfolding** *dominates-def* **by** *auto*  
**with**  $\langle n \neq n' \rangle$  **have**  $n \in set as$  **by** *auto*  
**with**  $\langle prefix (as@[n']) ns \rangle$  **have**  $n \in set (butlast ns)$  **by**  $-(erule prefixE, auto$   
*iff:butlast-append)*  
**with**  $\langle n \notin set (butlast ns) \rangle$  **show** *False..*  
**qed**

**lemma** *dominates-unsnoc*:  
**assumes**  $[simp]: invar g$  **and** *dominates*  $g n m m' \in set (predecessors g m) n$   
 $\neq m$   
**shows** *dominates*  $g n m'$   
**proof**  
**show**  $m' \in set (\alpha n g)$  **using** *assms* **by** *auto*  
**next**  
**fix**  $ns$   
**assume**  $g \vdash Entry g-ns \rightarrow m'$   
**with** *assms*(3) **have**  $g \vdash Entry g-ns@[m] \rightarrow m$  **by** *auto*  
**with** *assms*(2,4) **show**  $n \in set ns$  **by**  $(auto elim!:dominatesE)$   
**qed**

**lemma** *dominates-unsnoc'*:  
**assumes**  $[simp]: invar g$  **and** *dominates*  $g n m g \vdash m'-ms \rightarrow m \forall x \in set (tl$   
 $ms). x \neq n$   
**shows** *dominates*  $g n m'$   
**using** *assms*(3,4) **proof** (*induction rule:path2-induct*)  
**case** *empty-path*  
**show** *?case* **by**  $(rule assms(2))$   
**next**  
**case**  $(Cons-path ms m'' m')$   
**from** *Cons-path*(4) **have** *dominates*  $g n m'$   
**by**  $(simp add: Cons-path.IH in-set-tlD)$   
**moreover from** *Cons-path*(1) **have**  $m' \in set ms$  **by** *auto*  
**hence**  $m' \neq n$  **using** *Cons-path*(4) **by** *simp*  
**ultimately show** *?case* **using** *Cons-path*(2) **by**  $-(rule dominates-unsnoc,$   
*auto)*  
**qed**

**lemma** *dominates-path*:  
**assumes** *dominates*  $g n m$  **and**  $[simp]: invar g$   
**obtains**  $ns$  **where**  $g \vdash n-ns \rightarrow m$   
**proof** *atomize-elim*  
**from** *assms* **obtain**  $ns$  **where**  $ns: g \vdash Entry g-ns \rightarrow m$  **by** *atomize-elim* (*rule*  
*Entry-reaches, auto*)  
**with** *assms* **have**  $n \in set ns$  **by**  $-(erule dominatesE)$   
**with**  $ns$  **show**  $\exists ns. g \vdash n-ns \rightarrow m$  **by**  $-(rule path2-split-ex, auto)$   
**qed**

**lemma** *dominates-antitrans*:  
**assumes**  $[simp]: invar g$  **and** *dominates*  $g n_1 m$  *dominates*  $g n_2 m$

```

obtains (1) dominates  $g\ n_1\ n_2$ 
          | (2) dominates  $g\ n_2\ n_1$ 
proof (cases dominates  $g\ n_1\ n_2$ )
  case False
  show thesis
  proof (rule 2, rule dominatesI)
    show  $n_1 \in \text{set } (\alpha n\ g)$  using assms(2) by simp
  next
  fix ns
  assume asm:  $g \vdash \text{Entry } g\text{-}ns \rightarrow n_1$ 
  from assms(2) obtain  $ns_2$  where  $g \vdash n_1\text{-}ns_2 \rightarrow m$  by (rule dominates-path,
simp)
  then obtain  $ns_2'$  where  $ns_2'$ :  $g \vdash n_1\text{-}ns_2' \rightarrow m$   $n_1 \notin \text{set } (tl\ ns_2')$   $\text{set } ns_2' \subseteq$ 
set  $ns_2$  by (rule simple-path2)
  with asm have  $g \vdash \text{Entry } g\text{-}ns @ tl\ ns_2' \rightarrow m$  by auto
  with assms(3) have  $n_2 \in \text{set } (ns @ tl\ ns_2')$  by - (erule dominatesE)
  moreover have  $n_2 \notin \text{set } (tl\ ns_2')$ 
  proof
    assume  $n_2 \in \text{set } (tl\ ns_2')$ 
    with  $ns_2'(1,2)$  obtain  $ns_3$  where  $ns_3$ :  $g \vdash n_2\text{-}ns_3 \rightarrow m$   $n_1 \notin \text{set } (tl\ ns_3)$ 
      by - (erule path2-split-ex, auto simp: path2-not-Nil)
    have dominates  $g\ n_1\ n_2$ 
    proof
      show  $n_2 \in \text{set } (\alpha n\ g)$  using assms(3) by simp
    next
    fix  $ns'$ 
    assume  $ns'$ :  $g \vdash \text{Entry } g\text{-}ns' \rightarrow n_2$ 
    with  $ns_3(1)$  have  $g \vdash \text{Entry } g\text{-}ns' @ tl\ ns_3 \rightarrow m$  by auto
    with assms(2) have  $n_1 \in \text{set } (ns' @ tl\ ns_3)$  by - (erule dominatesE)
    with  $ns_3(2)$  show  $n_1 \in \text{set } ns'$  by simp
    qed
  with False show False ..
  qed
  ultimately show  $n_2 \in \text{set } ns$  by simp
qed
qed

```

**lemma** *dominates-extend*:

```

assumes dominates  $g\ n\ m$ 
assumes  $g \vdash m'\text{-}ms \rightarrow m$   $n \notin \text{set } (tl\ ms)$ 
shows dominates  $g\ n\ m'$ 
proof (rule dominatesI)
  show  $m' \in \text{set } (\alpha n\ g)$  using assms(2) by auto
next
fix  $ms'$ 
assume  $g \vdash \text{Entry } g\text{-}ms' \rightarrow m'$ 
with assms(2) have  $g \vdash \text{Entry } g\text{-}ms' @ tl\ ms \rightarrow m$  by auto
with assms(1) have  $n \in \text{set } (ms' @ tl\ ms)$  by - (erule dominatesE)
with assms(3) show  $n \in \text{set } ms'$  by auto

```

qed

**definition** *dominators* :: 'g ⇒ 'node ⇒ 'node set **where**  
  *dominators* g n ≡ {m ∈ set (αn g). *dominates* g m n}

**definition** *isIdom* g n m ⇔ *strict-dom* g m n ∧ (∀ m' ∈ set (αn g). *strict-dom* g m' n → *dominates* g m' m)

**definition** *idom* :: 'g ⇒ 'node ⇒ 'node **where**  
  *idom* g n ≡ THE m. *isIdom* g n m

**lemma** *idom-ex*:

**assumes** [*simp*]: *invar* g n ∈ set (αn g) n ≠ *Entry* g

**shows** ∃!m. *isIdom* g n m

**proof** (*rule ex-ex1I*)

**let** ?A = λm. {m' ∈ set (αn g). *strict-dom* g m' n ∧ *strict-dom* g m m'}

**have** 1: ∧A m. *finite* A ⇒ A = ?A m ⇒ *strict-dom* g m n ⇒ ∃ m'. *isIdom* g n m'

**proof** –

**fix** A m

**show** *finite* A ⇒ A = ?A m ⇒ *strict-dom* g m n ⇒ ∃ m'. *isIdom* g n m'

**proof** (*induction arbitrary:m rule:finite-psubset-induct*)

**case** (*psubset* A m)

**show** ?*case*

**proof** (*cases* A = {})

**case** *True*

          { **fix** m'

**assume** *asm*: *strict-dom* g m' n **and** [*simp*]: m' ∈ set (αn g)

**with** *True* *psubset.prem*s(1) **have** ¬(*strict-dom* g m m') **by** *auto*

**hence** *dominates* g m' m **using** *dominates-antitrans*[of g m' n m] *asm*  
            *psubset.prem*s(2) **by** *fastforce*

          }

**thus** ?*thesis* **using** *psubset.prem*s(2) **by** – (*rule exI*[of - m], *auto*  
*simp:isIdom-def*)

**next**

**case** *False*

**then obtain** m' **where** m' ∈ A **by** *auto*

**with** *psubset.prem*s(1) **have** m': m' ∈ set (αn g) *strict-dom* g m' n  
*strict-dom* g m m' **by** *auto*

**have** ?A m' ⊂ ?A m

**proof**

**show** ?A m' ≠ ?A m **using** m' **by** *auto*

**show** ?A m' ⊆ ?A m **using** m' *dominates-antisymm*[of g m m']  
*dominates-trans*[of g m] **by** *auto*

**qed**

**thus** ?*thesis* **by** (*rule* *psubset.IH*[of - m', *simplified* *psubset.prem*s(1)],  
*simp-all add: m'*)

**qed**

**qed**



```

qed
show  $\exists m. \text{isIdom } g \ n \ m$  by (rule 1[of ?A (Entry g)], auto)
next
fix m m'
assume isIdom g n m isIdom g n m'
thus m = m' by - (rule dominates-antisymm[of g], auto simp:isIdom-def)
qed

lemma idom:  $[[\text{invar } g; n \in \text{set } (\alpha n \ g) - \{\text{Entry } g\}]] \implies \text{isIdom } g \ n \ (\text{idom } g \ n)$ 
unfolding idom-def by (rule theI', rule idom-ex, auto)

lemma dominates-mid:
  assumes dominates g n x dominates g x m g  $\vdash n - ns \rightarrow m$  and[simp]: invar g
  shows  $x \in \text{set } ns$ 
using assms
proof (cases n = x)
  case False
  from assms(1) obtain ns0 where ns0: g  $\vdash \text{Entry } g - ns_0 \rightarrow n$   $n \notin \text{set } (\text{butlast } ns_0)$  by - (rule simple-Entry-path, auto)
  with assms(3) have g  $\vdash \text{Entry } g - \text{butlast } ns_0 @ ns \rightarrow m$  by auto
  with assms(2) have  $x \in \text{set } (\text{butlast } ns_0 @ ns)$  by (auto elim!:dominatesE)
  moreover have  $x \notin \text{set } (\text{butlast } ns_0)$ 
  proof
    assume asm:  $x \in \text{set } (\text{butlast } ns_0)$ 
    with ns0 obtain ns0' where ns0': g  $\vdash \text{Entry } g - ns_0' \rightarrow x$   $n \notin \text{set } (\text{butlast } ns_0')$ 
    by - (erule path2-split-ex, auto dest:in-set-butlastD simp: butlast-append split: if-split-asm)
    show False by (metis False assms(1) ns0' strict-domE)
  qed
  ultimately show ?thesis by simp
qed auto

definition shortestPath :: 'g  $\Rightarrow$  'node  $\Rightarrow$  nat where
  shortestPath g n  $\equiv$  (LEAST l.  $\exists ns. \text{length } ns = l \wedge g \vdash \text{Entry } g - ns \rightarrow n$ )

lemma shortestPath-ex:
  assumes n  $\in \text{set } (\alpha n \ g)$  invar g
  obtains ns where g  $\vdash \text{Entry } g - ns \rightarrow n$  distinct ns length ns = shortestPath g n
proof -
  from assms obtain ns where g  $\vdash \text{Entry } g - ns \rightarrow n$  by - (atomize-elim, rule Entry-reaches)
  then obtain sns where sns: length sns = shortestPath g n g  $\vdash \text{Entry } g - sns \rightarrow n$ 
  unfolding shortestPath-def
  by - (atomize-elim, rule LeastI, auto)
  then obtain sns' where sns': length sns'  $\leq$  shortestPath g n g  $\vdash \text{Entry } g - sns' \rightarrow n$  distinct sns' by - (rule simple-path2, auto)
  moreover from sns'(2) have shortestPath g n  $\leq$  length sns' unfolding

```

*shortestPath-def* **by** – (rule *Least-le*, auto)  
**ultimately show thesis by** –(rule *that*, auto)  
**qed**

**lemma**[*simp*]:  $\llbracket n \in \text{set } (\alpha n g); \text{invar } g \rrbracket \implies \text{shortestPath } g n \neq 0$   
**by** (metis *length-0-conv path2-not-Nil2 shortestPath-ex*)

**lemma** *shortestPath-upper-bound*:  
**assumes**  $n \in \text{set } (\alpha n g)$  *invar*  $g$   
**shows**  $\text{shortestPath } g n \leq \text{length } (\alpha n g)$   
**proof** –  
**from** *assms* **obtain**  $ns$  **where**  $ns: g \vdash \text{Entry } g - ns \rightarrow n$   $\text{length } ns = \text{shortestPath } g n$  *distinct*  $ns$  **by** (rule *shortestPath-ex*)  
**hence**  $\text{shortestPath } g n = \text{length } ns$  **by** *simp*  
**also have**  $\dots = \text{card } (\text{set } ns)$  **using**  $ns(3)$  **by** (rule *distinct-card[symmetric]*)  
**also have**  $\dots \leq \text{card } (\text{set } (\alpha n g))$  **using**  $ns(1)$  **by** – (rule *card-mono*, auto)  
**also have**  $\dots \leq \text{length } (\alpha n g)$  **by** (rule *card-length*)  
**finally show** *?thesis* .  
**qed**

**lemma** *shortestPath-predecessor*:  
**assumes**  $n \in \text{set } (\alpha n g) - \{\text{Entry } g\}$  **and**[*simp*]: *invar*  $g$   
**obtains**  $n'$  **where**  $\text{Suc } (\text{shortestPath } g n') = \text{shortestPath } g n$   $n' \in \text{set } (\text{predecessors } g n)$   
**proof** –  
**from** *assms* **obtain**  $sns$  **where**  $sns: \text{length } sns = \text{shortestPath } g n$   $g \vdash \text{Entry } g - sns \rightarrow n$   
**by** – (rule *shortestPath-ex*, auto)  
**let**  $?n' = \text{last } (\text{butlast } sns)$   
**from** *assms*(1)  $sns(2)$  **have**  $1: \text{length } sns \geq 2$  **by** auto  
**hence** *prefix*:  $g \vdash \text{Entry } g - \text{butlast } sns \rightarrow \text{last } (\text{butlast } sns) \wedge \text{last } (\text{butlast } sns) \in \text{set } (\text{predecessors } g n)$   
**using**  $sns$  **by** –(rule *path2-unsnoc*, auto)  
**hence**  $\text{shortestPath } g ?n' \leq \text{length } (\text{butlast } sns)$   
**unfolding** *shortestPath-def* **by** –(rule *Least-le*, rule *exI*[**where**  $x = \text{butlast } sns$ ], *simp*)  
**with**  $1$   $sns(1)$  **have**  $2: \text{shortestPath } g ?n' < \text{shortestPath } g n$  **by** auto  
**{ assume** *asm*:  $\text{Suc } (\text{shortestPath } g ?n') \neq \text{shortestPath } g n$   
**obtain**  $sns'$  **where**  $sns': g \vdash \text{Entry } g - sns' \rightarrow ?n'$   $\text{length } sns' = \text{shortestPath } g ?n'$   
**using** *prefix* **by** – (rule *shortestPath-ex*, auto)  
**hence**[*simp*]:  $g \vdash \text{Entry } g - sns'@[n] \rightarrow n$  **using** *prefix* **by** auto  
**from** *asm*  $2$  **have**  $\text{Suc } (\text{shortestPath } g ?n') < \text{shortestPath } g n$  **by** auto  
**from** *this*[*unfolded* *shortestPath-def*, *THEN* *not-less-Least*, *folded* *shortestPath-def*, *simplified*, *THEN* *spec*[*of* -  $sns'@[n]$ ]]  
**have** *False* **using**  $sns'(2)$  **by** auto  
**}**  
**with** *prefix* **show** *thesis* **by** – (rule *that*, auto)  
**qed**

**lemma** *successor-in- $\alpha n$* [simp]:  
**assumes** *predecessors*  $g\ n \neq []$  **and**[simp]: *invar*  $g$   
**shows**  $n \in \text{set } (\alpha n\ g)$   
**proof**–  
**from** *assms*(1) **obtain**  $m$  **where**  $m \in \text{set } (\text{predecessors } g\ n)$  **by** (*cases predecessors*  $g\ n$ , *auto*)  
**with** *assms*(1) **obtain**  $m'\ e$  **where**  $(m', e, n) \in \alpha e\ g$  **using** *inEdges-correct*[*of*  $g\ n$ , *THEN arg-cong*[**where**  $f = (\cdot)$  *getTo*]]  
**by** (*auto simp: predecessors-def simp del: inEdges-correct*)  
**with** *assms*(1) **show** *?thesis*  
**by** (*auto simp: predecessors-def*)  
**qed**

**lemma** *shortestPath-single-predecessor*:  
**assumes** *predecessors*  $g\ n = [m]$  **and**[simp]: *invar*  $g$   
**shows** *shortestPath*  $g\ m < \text{shortestPath } g\ n$   
**proof**–  
**from** *assms*(1) **have**  $n \in \text{set } (\alpha n\ g) - \{\text{Entry } g\}$   
**by** (*auto simp: predecessors-def Entry-unreachable*)  
**thus** *?thesis* **by** (*rule shortestPath-predecessor, auto simp: assms*(1))  
**qed**

**lemma** *strict-dom-shortestPath-order*:  
**assumes** *strict-dom*  $g\ n\ m\ m \in \text{set } (\alpha n\ g)$  *invar*  $g$   
**shows** *shortestPath*  $g\ n < \text{shortestPath } g\ m$   
**proof**–  
**from** *assms*(2,3) **obtain**  $sns$  **where**  $sns: g \vdash \text{Entry } g - sns \rightarrow m$  *length*  $sns = \text{shortestPath } g\ m$   
**by** (*rule shortestPath-ex*)  
**with** *assms*(1) *sns*(1) **obtain**  $sns'$  **where**  $sns': g \vdash \text{Entry } g - sns' \rightarrow n$  *prefix*  $sns'\ sns$  **by**  $-(\text{erule path2-prefix-ex, auto elim: dominatesE})$   
**hence** *shortestPath*  $g\ n \leq \text{length } sns'$   
**unfolding** *shortestPath-def* **by**  $-(\text{rule Least-le, auto})$   
**also have** *length*  $sns' < \text{length } sns$   
**proof**–  
**from** *assms*(1) *sns*(1) *sns'*(1) **have**  $sns' \neq sns$  **by**  $-(\text{drule path2-last, drule path2-last, auto})$   
**with** *sns'*(2) **have** *strict-prefix*  $sns'\ sns$  **by** *auto*  
**thus** *?thesis* **by** (*rule prefix-length-less*)  
**qed**  
**finally show** *?thesis* **by** (*simp add: sns*(2))  
**qed**

**lemma** *dominates-shortestPath-order*:  
**assumes** *dominates*  $g\ n\ m\ m \in \text{set } (\alpha n\ g)$  *invar*  $g$   
**shows** *shortestPath*  $g\ n \leq \text{shortestPath } g\ m$   
**using** *assms* **by** (*cases*  $n = m$ , *auto intro: strict-dom-shortestPath-order*[*THEN less-imp-le*])

```

lemma strict-dom-trans:
  assumes [simp]: invar g
  assumes strict-dom g n m strict-dom g m m'
  shows strict-dom g n m'
proof (rule, rule notI)
  assume  $n = m'$ 
  moreover from assms(3) have  $m' \in \text{set } (\alpha n g)$  by auto
  ultimately have dominates g m' n by auto
  with assms(2) have dominates g m' m by  $-$  (rule dominates-trans, auto)
  with assms(3) show False by  $-$  (erule conjE, drule dominates-antisymm[OF
assms(1)], auto)
next
  from assms show dominates g n m' by  $-$  (rule dominates-trans, auto)
qed

```

```

inductive EntryPath :: 'g  $\Rightarrow$  'node list  $\Rightarrow$  bool where
  EntryPath-triv [simp]: EntryPath g [n]
  | EntryPath-snoc [intro]: EntryPath g ns  $\Longrightarrow$  shortestPath g m = Suc (shortestPath
g (last ns))  $\Longrightarrow$  EntryPath g (ns@[m])

```

```

lemma [simp]:
  assumes EntryPath g ns prefix ns' ns ns'  $\neq$  []
  shows EntryPath g ns'
using assms proof induction
  case (EntryPath-triv ns n)
  thus ?case by (cases ns', auto)
qed auto

```

```

lemma EntryPath-suffix:
  assumes EntryPath g ns suffix ns' ns ns'  $\neq$  []
  shows EntryPath g ns'
using assms proof (induction arbitrary: ns')
  case EntryPath-triv
  thus ?case
  by (metis EntryPath.EntryPath-triv append-Nil append-is-Nil-conv list.sel(3)
Sublist.suffix-def tl-append2)
next
  case (EntryPath-snoc g ns m)
  from EntryPath-snoc.prems obtain ns'' where [simp]:  $ns' = ns''@[m]$ 
  by  $-$  (erule suffix-unsnoc, auto)
  show ?case
  proof (cases ns'' = [])
  case True
  thus ?thesis by auto
next
  case False
  from EntryPath-snoc.prems(1) have suffix ns'' ns by (auto simp: Sub-
list.suffix-def)

```

```

    with False have last ns'' = last ns by (auto simp: Sublist.suffix-def)
  moreover from False have EntryPath g ns'' using EntryPath-snoc.prem(1)
    by - (rule EntryPath-snoc.IH, auto simp: Sublist.suffix-def)
  ultimately show ?thesis using EntryPath-snoc.hyps(2)
    by - (simp, rule EntryPath.EntryPath-snoc, simp-all)
qed
qed

```

```

lemma EntryPath-butlast-less-last:
  assumes EntryPath g ns z ∈ set (butlast ns)
  shows shortestPath g z < shortestPath g (last ns)
using assms proof (induction)
  case (EntryPath-snoc g ns m)
  thus ?case by (cases z ∈ set (butlast ns), auto dest: not-in-butlast)
qed simp

```

```

lemma EntryPath-distinct:
  assumes EntryPath g ns
  shows distinct ns
using assms
proof (induction)
  case (EntryPath-snoc g ns m)
  from this consider (non-distinct) m ∈ set ns | distinct (ns @ [m]) by auto
  thus distinct (ns @ [m])
  proof (cases)
    case non-distinct
    have EntryPath g (ns @ [m]) using EntryPath-snoc by (intro Entry-
Path.intros(2))
    with non-distinct
    have False
    using EntryPath-butlast-less-last butlast-snoc last-snoc less-not-refl by force
    thus ?thesis by simp
  qed
qed simp

```

```

lemma Entry-reachesE:
  assumes n ∈ set (αn g) and[simp]: invar g
  obtains ns where g ⊢ Entry g-ns→n EntryPath g ns
using assms(1) proof (induction shortestPath g n arbitrary:n)
  case 0
  hence False by simp
  thus ?case..
next
  case (Suc l)
  note Suc.prem(2)[simp]
  show ?case
  proof (cases n = Entry g)
    case True
    thus ?thesis by - (rule Suc.prem(1), auto)

```

```

next
  case False
  then obtain n' where n': shortestPath g n' = l n' ∈ set (predecessors g n)
    using Suc.hyps(2)[symmetric] by - (rule shortestPath-predecessor, auto)
  moreover {
    fix ns
    assume asm: g ⊢ Entry g-ns→n' EntryPath g ns
    hence thesis using n' Suc.hyps(2) path2-last[OF asm(1)]
      by - (rule Suc.prem(1)[of ns@[n]], auto)
  }
  ultimately show thesis by - (rule Suc.hyps(1), auto)
qed
qed
end
end
end

```

```

theory SSA-CFG
imports Graph-path HOL-Library.Sublist
begin

```

## 2.3 CFG

```

locale CFG-base = graph-Entry-base αe αn invar inEdges' Entry
for
  αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set and
  αn :: 'g ⇒ 'node list and
  invar :: 'g ⇒ bool and
  inEdges' :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list and
  Entry :: 'g ⇒ 'node +
fixes defs :: 'g ⇒ 'node ⇒ 'var::linorder set
fixes uses :: 'g ⇒ 'node ⇒ 'var set
begin
  definition vars g ≡ fold (∪) (map (uses g) (αn g)) {}
  definition defAss' :: 'g ⇒ 'node ⇒ 'var ⇒ bool where
    defAss' g m v ↔ (∀ ns. g ⊢ Entry g-ns→m → (∃ n ∈ set ns. v ∈ defs g n))

  definition defAss'Uses :: 'g ⇒ bool where
    defAss'Uses g ≡ ∀ m ∈ set (αn g). ∀ v ∈ uses g m. defAss' g m v
end

```

```

locale CFG = CFG-base αe αn invar inEdges' Entry defs uses
+ graph-Entry αe αn invar inEdges' Entry
for
  αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set and
  αn :: 'g ⇒ 'node list and
  invar :: 'g ⇒ bool and
  inEdges' :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list and

```

```

Entry :: 'g ⇒ 'node and
defs :: 'g ⇒ 'node ⇒ 'var::linorder set and
uses :: 'g ⇒ 'node ⇒ 'var set +
assumes defs-uses-disjoint: n ∈ set (αn g) ⇒ defs g n ∩ uses g n = {}
assumes defs-finite[simp]: finite (defs g n)
assumes uses-in-αn: v ∈ uses g n ⇒ n ∈ set (αn g)
assumes uses-finite[simp, intro!]: finite (uses g n)
assumes invar[intro!]: invar g
begin
  lemma vars-finite[simp]: finite (vars g)
  by (auto simp:vars-def)

  lemma Entry-no-predecessor[simp]: predecessors g (Entry g) = []
  using Entry-unreachable
  by (auto simp:predecessors-def)

  lemma uses-in-vars[elim, simp]: v ∈ uses g n ⇒ v ∈ vars g
  by (auto simp add:vars-def uses-in-αn intro!: fold-union-elemI)

  lemma varsE:
    assumes v ∈ vars g
    obtains n where n ∈ set (αn g) v ∈ uses g n
    using assms by (auto simp:vars-def elim!:fold-union-elem)

  lemma defs-uses-disjoint'[simp]: n ∈ set (αn g) ⇒ v ∈ defs g n ⇒ v ∈ uses
g n ⇒ False
  using defs-uses-disjoint by auto
end

context CFG
begin
  lemma defAss'E:
    assumes defAss' g m v g ⊢ Entry g-ns→m
    obtains n where n ∈ set ns v ∈ defs g n
    using assms unfolding defAss'-def by auto

  lemmas defAss'I = defAss'-def[THEN iffD2, rule-format]

  lemma defAss'-extend:
    assumes defAss' g m v
    assumes g ⊢ n-ns→m ∀ n ∈ set (tl ns). v ∉ defs g n
    shows defAss' g n v
  unfolding defAss'-def proof (rule allI, rule impI)
    fix ns'
    assume g ⊢ Entry g-ns'→n
    with assms(2) have g ⊢ Entry g-ns'@tl ns→m by auto
    with assms(1) obtain n' where n': n' ∈ set (ns'@tl ns) v ∈ defs g n' by
-(erule defAss'E)
    with assms(3) have n' ∉ set (tl ns) by auto

```

```

    with n' show  $\exists n \in \text{set } ns'. v \in \text{defs } g \ n$  by auto
  qed
end

```

A CFG is well-formed if it satisfies definite assignment.

```

locale CFG-wf = CFG  $\alpha e \ \alpha n$  invar inEdges' Entry defs uses
for
   $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)$  set and
   $\alpha n :: 'g \Rightarrow 'node$  list and
  invar  $:: 'g \Rightarrow \text{bool}$  and
  inEdges'  $:: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)$  list and
  Entry  $:: 'g \Rightarrow 'node$  and
  defs  $:: 'g \Rightarrow 'node \Rightarrow 'var::linorder$  set and
  uses  $:: 'g \Rightarrow 'node \Rightarrow 'var$  set +
assumes def-ass-uses:  $\forall m \in \text{set } (\alpha n \ g). \forall v \in \text{uses } g \ m. \text{defAss}' \ g \ m \ v$ 

```

## 2.4 SSA CFG

```

type-synonym ('node, 'val) phis = 'node  $\times$  'val  $\rightarrow$  'val list

```

```

declare in-set-zipE[elim]
declare zip-same[simp]

```

```

locale CFG-SSA-base = CFG-base  $\alpha e \ \alpha n$  invar inEdges' Entry defs uses
for

```

```

   $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)$  set and
   $\alpha n :: 'g \Rightarrow 'node$  list and
  invar  $:: 'g \Rightarrow \text{bool}$  and
  inEdges'  $:: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)$  list and
  Entry  $:: 'g \Rightarrow 'node$  and
  defs  $:: 'g \Rightarrow 'node \Rightarrow 'val::linorder$  set and
  uses  $:: 'g \Rightarrow 'node \Rightarrow 'val$  set +

```

```

fixes phis  $:: 'g \Rightarrow ('node, 'val)$  phis

```

```

begin

```

```

  definition phiDefs  $g \ n \equiv \{v. (n, v) \in \text{dom } (\text{phis } g)\}$ 

```

```

  definition[code]: allDefs  $g \ n \equiv \text{defs } g \ n \cup \text{phiDefs } g \ n$ 

```

```

  definition[code]: phiUses  $g \ n \equiv$ 

```

```

 $\bigcup n' \in \text{set } (\text{successors } g \ n). \bigcup v' \in \text{phiDefs } g \ n'. \text{snd } \text{'Set.filter } (\lambda(n'', v). n''$ 

```

```

 $= n) (\text{set } (\text{zip } (\text{predecessors } g \ n') (\text{the } (\text{phis } g \ (n', v')))))$ 

```

```

  definition[code]: allUses  $g \ n \equiv \text{uses } g \ n \cup \text{phiUses } g \ n$ 

```

```

  definition[code]: allVars  $g \equiv \bigcup n \in \text{set } (\alpha n \ g). \text{allDefs } g \ n \cup \text{allUses } g \ n$ 

```

```

  definition defAss  $:: 'g \Rightarrow 'node \Rightarrow 'val \Rightarrow \text{bool}$  where

```

```

 $\text{defAss } g \ m \ v \iff (\forall ns. g \vdash \text{Entry } g \text{-ns} \rightarrow m \longrightarrow (\exists n \in \text{set } ns. v \in \text{allDefs } g$ 

```

```

 $n))$ 

```

```

  lemmas CFG-SSA-defs = phiDefs-def allDefs-def phiUses-def allUses-def allVars-def
defAss-def

```

```

end

```



**locale** *CFG-SSA* = *CFG*  $\alpha e$   $\alpha n$  *invar* *inEdges'* *Entry* *defs* *uses* + *CFG-SSA-base*  
 $\alpha e$   $\alpha n$  *invar* *inEdges'* *Entry* *defs* *uses* *phis*  
**for**  
 $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)$  **set** **and**  
 $\alpha n :: 'g \Rightarrow 'node$  **list** **and**  
*invar*  $:: 'g \Rightarrow bool$  **and**  
*inEdges'*  $:: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)$  **list** **and**  
*Entry*  $:: 'g \Rightarrow 'node$  **and**  
*defs*  $:: 'g \Rightarrow 'node \Rightarrow 'val::linorder$  **set** **and**  
*uses*  $:: 'g \Rightarrow 'node \Rightarrow 'val$  **set** **and**  
*phis*  $:: 'g \Rightarrow ('node, 'val)$  *phis* +  
**assumes** *phis-finite*: *finite* (*dom* (*phis* *g*))  
**assumes** *phis-in- $\alpha n$* : *phis* *g* (*n*,*v*) = *Some* *vs*  $\implies n \in set$  ( $\alpha n$  *g*)  
**assumes** *phis-wf*:  
*phis* *g* (*n*,*v*) = *Some* *args*  $\implies length$  (*predecessors* *g* *n*) = *length* *args*  
**assumes** *simpleDefs-phiDefs-disjoint*:  
 $n \in set$  ( $\alpha n$  *g*)  $\implies defs$  *g* *n*  $\cap phiDefs$  *g* *n* = {}  
**assumes** *allDefs-disjoint*:  
 $\llbracket n \in set$  ( $\alpha n$  *g*);  $m \in set$  ( $\alpha n$  *g*);  $n \neq m$   $\rrbracket \implies allDefs$  *g* *n*  $\cap allDefs$  *g* *m* = {}  
**begin**  
**lemma** *phis-disj*:  
**assumes** *phis* *g* (*n*,*v*) = *Some* *vs*  
**and** *phis* *g* (*n'*,*v*) = *Some* *vs'*  
**shows**  $n = n'$  **and**  $vs = vs'$   
**proof** –  
**from** *assms* **have**  $n \in set$  ( $\alpha n$  *g*) **and**  $n' \in set$  ( $\alpha n$  *g*)  
**by** (*auto* *dest*: *phis-in- $\alpha n$* )  
**from** *allDefs-disjoint* [*OF* *this*] *assms* **show**  $n = n'$   
**by** (*auto* *simp*: *allDefs-def* *phiDefs-def*)  
**with** *assms* **show**  $vs = vs'$  **by** *simp*  
**qed**  
  
**lemma** *allDefs-disjoint'*:  $\llbracket n \in set$  ( $\alpha n$  *g*);  $m \in set$  ( $\alpha n$  *g*);  $v \in allDefs$  *g* *n*;  $v \in allDefs$  *g* *m*  $\rrbracket \implies n = m$   
**using** *allDefs-disjoint* **by** *auto*  
  
**lemma** *phiUsesI*:  
**assumes**  $n' \in set$  ( $\alpha n$  *g*) *phis* *g* (*n'*,*v'*) = *Some* *vs* (*n*,*v*)  $\in set$  (*zip* (*predecessors* *g* *n'*) *vs*)  
**shows**  $v \in phiUses$  *g* *n*  
**proof** –  
**from** *assms*(3) **have**  $n \in set$  (*predecessors* *g* *n'*) **by** *auto*  
**hence** 1:  $n' \in set$  (*successors* *g* *n*) **using** *assms*(1) **by** *simp*  
**from** *assms*(2) **have** 2:  $v' \in phiDefs$  *g* *n'* **by** (*auto* *simp* *add*:*phiDefs-def*)  
**from** *assms*(2) **have** 3: *the* (*phis* *g* (*n'*,*v'*)) = *vs* **by** *simp*  
**show** ?*thesis* **unfolding** *phiUses-def* **by** (*rule* *UN-I*[*OF* 1], *rule* *UN-I*[*OF* 2],  
*auto* *simp*:*image-def* *Set.filter-def* *assms*(3) 3)  
**qed**

**lemma** *phiUsesE*:  
**assumes**  $v \in \text{phiUses } g \ n$   
**obtains**  $n' \ v' \ vs$  **where**  $n' \in \text{set } (\text{successors } g \ n)$   $(n, v) \in \text{set } (\text{zip } (\text{predecessors } g \ n') \ vs)$   $\text{phis } g \ (n', v') = \text{Some } vs$   
**proof** –  
**from** *assms*(1) **obtain**  $n' \ v' \ \text{where}$   $n' \in \text{set } (\text{successors } g \ n)$   $v' \in \text{phiDefs } g \ n'$   
 $v \in \text{snd } \text{'Set.filter } (\lambda(n'', v). n'' = n) (\text{set } (\text{zip } (\text{predecessors } g \ n') \ (\text{the } (\text{phis } g \ (n', v')))))$  **by** (*auto simp:phiUses-def*)  
**thus** *?thesis* **by** – (*rule that[of n' the (phis g (n',v')) v']*, *auto simp:phiDefs-def*)  
**qed**

**lemma** *defs-in-allDefs[simp]*:  $v \in \text{defs } g \ n \implies v \in \text{allDefs } g \ n$  **by** (*simp add:allDefs-def*)

**lemma** *phiDefs-in-allDefs[simp, elim]*:  $v \in \text{phiDefs } g \ n \implies v \in \text{allDefs } g \ n$  **by** (*simp add:allDefs-def*)

**lemma** *uses-in-allUses[simp]*:  $v \in \text{uses } g \ n \implies v \in \text{allUses } g \ n$  **by** (*simp add:allUses-def*)

**lemma** *phiUses-in-allUses[simp]*:  $v \in \text{phiUses } g \ n \implies v \in \text{allUses } g \ n$  **by** (*simp add:allUses-def*)

**lemma** *allDefs-in-allVars[simp, intro]*:  $\llbracket v \in \text{allDefs } g \ n; n \in \text{set } (\alpha n \ g) \rrbracket \implies v \in \text{allVars } g$  **by** (*auto simp:allVars-def*)

**lemma** *allUses-in-allVars[simp, intro]*:  $\llbracket v \in \text{allUses } g \ n; n \in \text{set } (\alpha n \ g) \rrbracket \implies v \in \text{allVars } g$  **by** (*auto simp:allVars-def*)

**lemma** *phiDefs-finite[simp]*: *finite* (*phiDefs*  $g \ n$ )

**unfolding** *phiDefs-def*

**proof** (*rule finite-surj[where f=snd]*, *rule phis-finite[where g=g]*)

**have**  $\bigwedge x \ y. \text{phis } g \ (n, x) = \text{Some } y \implies x \in \text{snd } \text{'dom } (\text{phis } g)$  **by** (*metis domI imageI snd-conv*)

**thus**  $\{v. (n, v) \in \text{dom } (\text{phis } g)\} \subseteq \text{snd } \text{'dom } (\text{phis } g)$  **by** *auto*

**qed**

**lemma** *phiUses-finite[simp]*:

**assumes**  $n \in \text{set } (\alpha n \ g)$

**shows** *finite* (*phiUses*  $g \ n$ )

**by** (*auto simp:phiUses-def Set.filter-def*)

**lemma** *allDefs-finite[simp]*:  $n \in \text{set } (\alpha n \ g) \implies \text{finite } (\text{allDefs } g \ n)$  **by** (*auto simp add:allDefs-def*)

**lemma** *allUses-finite[simp]*:  $n \in \text{set } (\alpha n \ g) \implies \text{finite } (\text{allUses } g \ n)$  **by** (*auto simp add:allUses-def*)

**lemma** *allVars-finite[simp]*: *finite* (*allVars*  $g$ ) **by** (*auto simp add:allVars-def*)

**lemmas** *defAssI* = *defAss-def[THEN iffD2, rule-format]*

**lemmas** *defAssD* = *defAss-def[THEN iffD1, rule-format]*

**lemma** *defAss-extend*:

**assumes** *defAss*  $g \ m \ v$

```

    assumes  $g \vdash n - ns \rightarrow m \ \forall n \in \text{set } (tl \ ns). \ v \notin \text{allDefs } g \ n$ 
    shows  $\text{defAss } g \ n \ v$ 
  unfolding  $\text{defAss-def}$  proof (rule  $\text{allI}$ , rule  $\text{impI}$ )
    fix  $ns'$ 
    assume  $g \vdash \text{Entry } g - ns' \rightarrow n$ 
    with  $\text{assms}(2)$  have  $g \vdash \text{Entry } g - ns' @ tl \ ns \rightarrow m$  by  $\text{auto}$ 
    with  $\text{assms}(1)$  obtain  $n'$  where  $n': n' \in \text{set } (ns' @ tl \ ns) \ v \in \text{allDefs } g \ n'$  by
    (auto dest: $\text{defAssD}$ )
    with  $\text{assms}(3)$  have  $n' \notin \text{set } (tl \ ns)$  by  $\text{auto}$ 
    with  $n'$  show  $\exists n \in \text{set } ns'. \ v \in \text{allDefs } g \ n$  by  $\text{auto}$ 
  qed

lemma  $\text{defAss-dominating}$ :
  assumes[ $\text{simp}$ ]:  $n \in \text{set } (\alpha n \ g)$ 
  shows  $\text{defAss } g \ n \ v \iff (\exists m \in \text{set } (\alpha n \ g). \ \text{dominates } g \ m \ n \wedge v \in \text{allDefs } g \ m)$ 
  proof
    assume  $\text{asm}: \text{defAss } g \ n \ v$ 
    obtain  $ns$  where  $ns: g \vdash \text{Entry } g - ns \rightarrow n$  by ( $\text{atomize}$ ,  $\text{auto}$ )
    from  $\text{defAssD}[OF \ \text{asm} \ \text{this}]$  obtain  $m$  where  $m: m \in \text{set } ns \ v \in \text{allDefs } g \ m$ 
  by  $\text{auto}$ 
    have  $\text{dominates } g \ m \ n$ 
  proof
    fix  $ns'$ 
    assume  $ns': g \vdash \text{Entry } g - ns' \rightarrow n$ 
    from  $\text{defAssD}[OF \ \text{asm} \ \text{this}]$  obtain  $m'$  where  $m': m' \in \text{set } ns' \ v \in \text{allDefs } g \ m'$  by  $\text{auto}$ 
    with  $m \ ns \ ns'$  have  $m' = m$  by  $-$  (rule  $\text{allDefs-disjoint'}$ ,  $\text{auto}$ )
    with  $m'$  show  $m \in \text{set } ns'$  by  $\text{simp}$ 
  qed  $\text{simp}$ 
    with  $m \ ns$  show  $\exists m \in \text{set } (\alpha n \ g). \ \text{dominates } g \ m \ n \wedge v \in \text{allDefs } g \ m$  by  $\text{auto}$ 
  next
    assume  $\exists m \in \text{set } (\alpha n \ g). \ \text{dominates } g \ m \ n \wedge v \in \text{allDefs } g \ m$ 
    then obtain  $m$  where[ $\text{simp}$ ]:  $m \in \text{set } (\alpha n \ g)$  and  $m: \text{dominates } g \ m \ n \ v \in \text{allDefs } g \ m$  by  $\text{auto}$ 
    show  $\text{defAss } g \ n \ v$ 
  proof (rule  $\text{defAssI}$ )
    fix  $ns$ 
    assume  $g \vdash \text{Entry } g - ns \rightarrow n$ 
    with  $m(1)$  have  $m \in \text{set } ns$  by  $-$  (rule  $\text{dominates-mid}$ ,  $\text{auto}$ )
    with  $m(2)$  show  $\exists n \in \text{set } ns. \ v \in \text{allDefs } g \ n$  by  $\text{auto}$ 
  qed
  qed
end

locale  $\text{CFG-SSA-wf-base} = \text{CFG-SSA-base } \alpha e \ \alpha n \ \text{invar} \ \text{inEdges}' \ \text{Entry} \ \text{defs} \ \text{uses}$ 
   $\text{phis}$ 
for
   $\alpha e :: 'g \Rightarrow ('node :: \text{linorder} \times 'edgeD \times 'node) \ \text{set}$  and

```

```

 $\alpha n :: 'g \Rightarrow 'node \text{ list and}$ 
 $invar :: 'g \Rightarrow bool \text{ and}$ 
 $inEdges' :: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD) \text{ list and}$ 
 $Entry :: 'g \Rightarrow 'node \text{ and}$ 
 $defs :: 'g \Rightarrow 'node \Rightarrow 'val :: \text{linorder set and}$ 
 $uses :: 'g \Rightarrow 'node \Rightarrow 'val \text{ set and}$ 
 $phis :: 'g \Rightarrow ('node, 'val) \text{ phis}$ 
begin

```

Using the SSA properties, we can map every value to its unique defining node and remove the *'node* parameter of the *phis* map.

```

definition defNode :: 'g  $\Rightarrow$  'val  $\Rightarrow$  'node where
  defNode-code [code]:  $defNode\ g\ v \equiv hd\ [n \leftarrow \alpha n\ g.\ v \in allDefs\ g\ n]$ 

```

```

abbreviation def-dominates  $g\ v'\ v \equiv dominates\ g\ (defNode\ g\ v')$  ( $defNode\ g\ v$ )

```

```

abbreviation strict-def-dom  $g\ v'\ v \equiv defNode\ g\ v' \neq defNode\ g\ v \wedge def-dominates\ g\ v'\ v$ 

```

```

definition phi  $g\ v = phis\ g\ (defNode\ g\ v, v)$ 

```

```

definition[simp]: phiArg  $g\ v\ v' \equiv \exists vs.\ phi\ g\ v = Some\ vs \wedge v' \in set\ vs$ 

```

```

definition[code]: isTrivialPhi  $g\ v\ v' \longleftrightarrow v' \neq v \wedge$ 

```

```

  (case phi g v of
     $Some\ vs \Rightarrow set\ vs = \{v, v'\} \vee set\ vs = \{v'\}$ 
  |  $None \Rightarrow False$ )

```

```

definition[code]: trivial  $g\ v \equiv \exists v' \in allVars\ g.\ isTrivialPhi\ g\ v\ v'$ 

```

```

definition[code]: redundant  $g \equiv \exists v \in allVars\ g.\ trivial\ g\ v$ 

```

```

definition defAssUses  $g \equiv \forall n \in set\ (\alpha n\ g).\ \forall v \in allUses\ g\ n.\ defAss\ g\ n\ v$ 

```

'liveness' of an SSA value is defined inductively starting from simple uses so that a circle of  $\phi$  functions is not considered live.

```

declare [[inductive-internals]]

```

```

inductive liveVal :: 'g  $\Rightarrow$  'val  $\Rightarrow$  bool

```

```

  for  $g :: 'g$ 

```

```

where

```

```

  liveSimple:  $[n \in set\ (\alpha n\ g); val \in uses\ g\ n] \Longrightarrow liveVal\ g\ val$ 

```

```

  | livePhi:  $[liveVal\ g\ v; phiArg\ g\ v\ v'] \Longrightarrow liveVal\ g\ v'$ 

```

```

definition pruned  $g = (\forall n \in set\ (\alpha n\ g).\ \forall val.\ val \in phiDefs\ g\ n \longrightarrow liveVal\ g\ val)$ 

```

```

lemmas CFG-SSA-wf-defs = CFG-SSA-defs defNode-code phi-def isTrivialPhi-def
trivial-def redundant-def liveVal-def pruned-def

```

```

end

```

```

locale CFG-SSA-wf = CFG-SSA  $\alpha e\ \alpha n\ invar\ inEdges'\ Entry\ defs\ uses\ phis +$ 
CFG-SSA-wf-base  $\alpha e\ \alpha n\ invar\ inEdges'\ Entry\ defs\ uses\ phis$ 

```

```

for

```

```

αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set and
αn :: 'g ⇒ 'node list and
invar :: 'g ⇒ bool and
inEdges' :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list and
Entry::'g ⇒ 'node and
defs :: 'g ⇒ 'node ⇒ 'val::linorder set and
uses :: 'g ⇒ 'node ⇒ 'val set and
phis :: 'g ⇒ ('node, 'val) phis +
assumes allUses-def-ass:  $\llbracket v \in \text{allUses } g \ n; \ n \in \text{set } (\alpha n \ g) \rrbracket \implies \text{defAss } g \ n \ v$ 
assumes Entry-no-phis[simp]:  $\text{phis } g \ (\text{Entry } g, v) = \text{None}$ 
begin
lemma allVars-in-allDefs:  $v \in \text{allVars } g \implies \exists n \in \text{set } (\alpha n \ g). \ v \in \text{allDefs } g \ n$ 
  unfolding allVars-def
  apply auto
  apply (drule(1) allUses-def-ass)
  apply (clarsimp simp: defAss-def)
  apply (drule Entry-reaches)
  apply auto[1]
  by fastforce

lemma phiDefs-Entry-empty[simp]:  $\text{phiDefs } g \ (\text{Entry } g) = \{\}$ 
  by (auto simp: phiDefs-def)

lemma phi-Entry-empty[simp]:  $\text{defNode } g \ v = \text{Entry } g \implies \text{phi } g \ v = \text{None}$ 
  by (simp add:phi-def)

lemma defNode-ex1:
  assumes  $v \in \text{allVars } g$ 
  shows  $\exists! n. \ n \in \text{set } (\alpha n \ g) \wedge v \in \text{allDefs } g \ n$ 
  proof (rule ex-ex1I)
    show  $\exists n. \ n \in \text{set } (\alpha n \ g) \wedge v \in \text{allDefs } g \ n$ 
  proof-
    from assms(1) obtain n where  $n: \ n \in \text{set } (\alpha n \ g) \wedge v \in \text{allDefs } g \ n \vee v \in \text{allUses } g \ n$  by (auto simp:allVars-def)
    thus ?thesis
    proof (cases  $v \in \text{allUses } g \ n$ )
      case True
        from n(1) obtain ns where  $ns: \ g \vdash \text{Entry } g \text{-}ns \rightarrow n$  by (atomize-elim, rule Entry-reaches, auto)
        with allUses-def-ass[OF True n(1)] obtain m where  $m: \ m \in \text{set } ns \wedge v \in \text{allDefs } g \ m$  by - (drule defAssD, auto)
        from ns this(1) have  $m \in \text{set } (\alpha n \ g)$  by (rule path2-in-αn)
        with n(1) m show ?thesis by auto
      case False
        qed auto
    qed
  show  $\bigwedge n \ m. \ n \in \text{set } (\alpha n \ g) \wedge v \in \text{allDefs } g \ n \implies m \in \text{set } (\alpha n \ g) \wedge v \in \text{allDefs } g \ m \implies n = m$  using allDefs-disjoint by auto
  qed

```

**lemma** *defNode-def*:  $v \in \text{allVars } g \implies \text{defNode } g \ v = (\text{THE } n. n \in \text{set } (\alpha n \ g))$   
 $\wedge v \in \text{allDefs } g \ n)$

**unfolding** *defNode-code* **by** (rule *the1-list[symmetric]*, rule *defNode-ex1*)

**lemma** *defNode[simp]*:

**assumes**  $v \in \text{allVars } g$

**shows**  $(\text{defNode } g \ v) \in \text{set } (\alpha n \ g) \ v \in \text{allDefs } g \ (\text{defNode } g \ v)$

**apply** (*atomize(full)*)

**unfolding** *defNode-def* [*OF assms*] **using** *assms*

**by** – (rule *theI'*, rule *defNode-ex1*)

**lemma** *defNode-eq[intro]*:

**assumes**  $n \in \text{set } (\alpha n \ g) \ v \in \text{allDefs } g \ n$

**shows**  $\text{defNode } g \ v = n$

**apply** (*subst defNode-def*, rule *allDefs-in-allVars* [*OF assms(2) assms(1)*])

**by** (rule *the1-equality*, rule *defNode-ex1*, rule *allDefs-in-allVars* [**where**  $n=n$ ],  
*simp-all add:assms*)

**lemma** *defNode-cases[consumes 1]*:

**assumes**  $v \in \text{allVars } g$

**obtains** (*simpleDef*)  $v \in \text{defs } g \ (\text{defNode } g \ v)$

| (*phi*)  $\text{phi } g \ v \neq \text{None}$

**proof** (*cases v \in defs g (defNode g v)*)

**case** *True*

**thus** *thesis* **by** (rule *simpleDef*)

**next**

**case** *False*

**with** *assms* [*THEN defNode(2)*] **show** *thesis*

**by** – (rule *phi*, *auto simp: allDefs-def phiDefs-def phi-def*)

**qed**

**lemma** *phi-phiDefs[simp]*:  $\text{phi } g \ v = \text{Some } vs \implies v \in \text{phiDefs } g \ (\text{defNode } g \ v)$   
**by** (*auto simp: phiDefs-def phi-def*)

**lemma** *simpleDef-not-phi*:

**assumes**  $n \in \text{set } (\alpha n \ g) \ v \in \text{defs } g \ n$

**shows**  $\text{phi } g \ v = \text{None}$

**proof** –

**from** *assms* **have**  $\text{defNode } g \ v = n$  **by** *auto*

**with** *assms* **show** *?thesis* **using** *simpleDefs-phiDefs-disjoint* **by** (*auto simp:*  
*phi-def phiDefs-def*)

**qed**

**lemma** *phi-wf*:  $\text{phi } g \ v = \text{Some } vs \implies \text{length } (\text{predecessors } g \ (\text{defNode } g \ v)) =$   
 $\text{length } vs$

**by** (rule *phis-wf*) (*simp add: phi-def*)

**lemma** *phi-finite*:  $\text{finite } (\text{dom } (\text{phi } g))$

**proof** –

```

let ?f = λv. (defNode g v,v)
have phi g = phis g ∘ ?f by (auto simp add:phi-def)
moreover have inj ?f by (auto intro:injI)
hence finite (dom (phis g ∘ ?f)) by - (rule finite-dom-comp, auto simp
add:phis-finite inj-on-def)
ultimately show ?thesis by simp
qed

```

```

lemma phiUses-exI:
assumes m ∈ set (predecessors g n) phis g (n,v) = Some vs n ∈ set (αn g)
obtains v' where v' ∈ phiUses g m v' ∈ set vs
proof-
from assms(1) obtain i where i: m = predecessors g n ! i i < length
(predecessors g n) by (metis in-set-conv-nth)
with assms(2) phis-wf have[simp]: i < length vs by (auto simp add:phi-def)
from i assms(2,3) have vs ! i ∈ phiUses g m by - (rule phiUsesI, auto simp
add:phiUses-def phi-def set-zip)
thus thesis by (rule that) (auto simp add:i(2) phis-wf)
qed

```

```

lemma phiArg-exI:
assumes m ∈ set (predecessors g (defNode g v)) phi g v ≠ None and[simp]: v
∈ allVars g
obtains v' where v' ∈ phiUses g m phiArg g v v'
proof-
from assms(2) obtain vs where phi g v = Some vs by auto
with assms(1) show thesis
by - (rule phiUses-exI, auto intro!:that simp: phi-def)
qed

```

```

lemma phiUses-exI':
assumes phiArg g p q and[simp]: p ∈ allVars g
obtains m where q ∈ phiUses g m m ∈ set (predecessors g (defNode g p))
proof-
let ?n = defNode g p
from assms(1) obtain i vs where vs: phi g p = Some vs and i: q = vs ! i i
< length vs by (metis in-set-conv-nth phiArg-def)
with phis-wf have[simp]: i < length (predecessors g ?n) by (auto simp
add:phi-def)
from vs i have q ∈ phiUses g (predecessors g ?n ! i) by - (rule phiUsesI,
auto simp add:phiUses-def phi-def set-zip)
thus thesis by (rule that) (auto simp add:i(2) phis-wf)
qed

```

```

lemma phiArg-in-allVars[simp]:
assumes phiArg g v v'
shows v' ∈ allVars g
proof-
let ?n = defNode g v

```

**from** *assms*(1) **obtain** *vs* **where** *vs*:  $\text{phi } g \ v = \text{Some } vs \ v' \in \text{set } vs$  **by** *auto*  
**then obtain** *m* **where** *m*:  $(m, v') \in \text{set } (\text{zip } (\text{predecessors } g \ ?n) \ vs)$  **by**  $-$  (rule *set-  
zip-leftI*, rule *phi-wf*)  
**from** *vs*(1) **have** *n*:  $?n \in \text{set } (\alpha n \ g)$  **by** (*simp add: phi-def phis-in- $\alpha n$* )  
**with** *m* **have**[*simp*]:  $m \in \text{set } (\alpha n \ g)$  **by** *auto*  
**from** *n m vs* **have**  $v' \in \text{phiUses } g \ m$  **by**  $-$  (rule *phiUsesI*, *simp-all add:phi-def*)  
**thus** *?thesis* **by**  $-$  (rule *allUses-in-allVars*, *auto simp:allUses-def*)  
**qed**

**lemma** *defAss-defNode*:

**assumes**  $\text{defAss } g \ m \ v \ v \in \text{allVars } g \ g \vdash \text{Entry } g - ns \rightarrow m$   
**shows**  $\text{defNode } g \ v \in \text{set } ns$

**proof** $-$

**from** *assms* **obtain** *n* **where** *n*:  $n \in \text{set } ns \ v \in \text{allDefs } g \ n$  **by** (*auto  
simp:defAss-def*)  
**with** *assms*(3) **have**  $n = \text{defNode } g \ v$  **by**  $-$  (rule *defNode-eq[symmetric]*, *auto*)  
**with** *n* **show**  $\text{defNode } g \ v \in \text{set } ns$  **by** (*simp add:defAss-def*)  
**qed**

**lemma** *defUse-path-ex*:

**assumes**  $v \in \text{allUses } g \ m \ m \in \text{set } (\alpha n \ g)$   
**obtains** *ns* **where**  $g \vdash \text{defNode } g \ v - ns \rightarrow m \ \text{EntryPath } g \ ns$

**proof** $-$

**from** *assms* **have**  $\text{defAss } g \ m \ v$  **by**  $-$  (rule *allUses-def-ass*, *auto*)  
**moreover from** *assms* **obtain** *ns* **where**  $g \vdash \text{Entry } g - ns \rightarrow m \ \text{EntryPath } g \ ns$   
**by**  $-$  (*atomize-elim*, rule *Entry-reachesE*, *auto*)  
**ultimately have**  $\text{defNode } g \ v \in \text{set } ns$  **using** *assms*(1)  
**by**  $-$  (rule *defAss-defNode*, *auto*)  
**with** *ns*(1) **obtain** *ns'* **where**  $g \vdash \text{defNode } g \ v - ns' \rightarrow m \ \text{suffix } ns' \ ns$   
**by** (rule *path2-split-ex'*, *auto simp: Sublist.suffix-def*)  
**thus** *thesis* **using** *ns*(2)  
**by**  $-$  (rule *that*, *assumption*, rule *EntryPath-suffix*, *auto*)  
**qed**

**lemma** *defUse-path-dominated*:

**assumes**  $g \vdash \text{defNode } g \ v - ns \rightarrow n \ \text{defNode } g \ v \notin \text{set } (\text{tl } ns) \ v \in \text{allUses } g \ n \ n' \in \text{set } ns$   
**shows** *dominates*  $g \ (\text{defNode } g \ v) \ n'$   
**proof** (rule *dominatesI*)  
**fix** *es*  
**assume** *asm*:  $g \vdash \text{Entry } g - es \rightarrow n'$   
**from** *assms*(1,4) **obtain** *ns'* **where**  $g \vdash n' - ns' \rightarrow n \ \text{suffix } ns' \ ns$   
**by**  $-$  (rule *path2-split-ex*, *auto simp: Sublist.suffix-def*)  
**from** *assms* **have**  $\text{defAss } g \ n \ v$  **by**  $-$  (rule *allUses-def-ass*, *auto*)  
**with** *asm ns'*(1) *assms*(3) **have**  $\text{defNode } g \ v \in \text{set } (es @ \text{tl } ns')$  **by**  $-$  (rule *defAss-defNode*, *auto*)  
**with** *suffix-tl-subset[OF ns'*(2)] *assms*(2) **show**  $\text{defNode } g \ v \in \text{set } es$  **by** *auto*  
**next**



**show**  $n' \in \text{set } (\alpha n \ g)$  **using**  $\text{assms}(1,4)$  **by**  $\text{auto}$   
**qed**

**lemma**  $\text{allUses-dominated}$ :  
**assumes**  $v \in \text{allUses } g \ n \ n \in \text{set } (\alpha n \ g)$   
**shows**  $\text{dominates } g \ (\text{defNode } g \ v) \ n$   
**proof**–  
**from**  $\text{assms}$  **obtain**  $ns$  **where**  $g \vdash \text{defNode } g \ v \text{--} ns \rightarrow n \ \text{defNode } g \ v \notin \text{set } (tl \ ns)$   
**by** –  $(\text{rule } \text{defUse-path-ex}, \text{auto } \text{elim}: \text{simple-path2})$   
**with**  $\text{assms}(1)$  **show**  $?thesis$  **by** –  $(\text{rule } \text{defUse-path-dominated}, \text{auto})$   
**qed**

**lemma**  $\text{phiArg-path-ex}'$ :  
**assumes**  $\text{phiArg } g \ p \ q$  **and** $[\text{simp}]$ :  $p \in \text{allVars } g$   
**obtains**  $ns \ m$  **where**  $g \vdash \text{defNode } g \ q \text{--} ns \rightarrow m \ \text{EntryPath } g \ ns \ q \in \text{phiUses } g \ m \ m \in \text{set } (\text{predecessors } g \ (\text{defNode } g \ p))$   
**proof**–  
**from**  $\text{assms}$  **obtain**  $m$  **where**  $m: q \in \text{phiUses } g \ m \ m \in \text{set } (\text{predecessors } g \ (\text{defNode } g \ p))$   
**by**  $(\text{rule } \text{phiUses-exI}')$   
**then obtain**  $ns$  **where**  $g \vdash \text{defNode } g \ q \text{--} ns \rightarrow m \ \text{EntryPath } g \ ns$  **by** –  $(\text{rule } \text{defUse-path-ex}, \text{auto})$   
**with**  $m$  **show**  $thesis$  **by** –  $(\text{rule } \text{that})$   
**qed**

**lemma**  $\text{phiArg-path-ex}$ :  
**assumes**  $\text{phiArg } g \ p \ q$  **and** $[\text{simp}]$ :  $p \in \text{allVars } g$   
**obtains**  $ns$  **where**  $g \vdash \text{defNode } g \ q \text{--} ns \rightarrow \text{defNode } g \ p \ \text{length } ns > 1$   
**by**  $(\text{rule } \text{phiArg-path-ex}'[\text{OF } \text{assms}], \text{rule}, \text{auto})$

**lemma**  $\text{phiArg-tranclp-path-ex}$ :  
**assumes**  $r^{++} \ p \ q \ p \in \text{allVars } g$  **and** $[\text{simp}]$ :  $\bigwedge p \ q. r \ p \ q \implies \text{phiArg } g \ p \ q$   
**obtains**  $ns$  **where**  $g \vdash \text{defNode } g \ q \text{--} ns \rightarrow \text{defNode } g \ p \ \text{length } ns > 1$   
 $\forall n \in \text{set } (\text{butlast } ns). \exists p \ q \ m \ ns'. r \ p \ q \wedge g \vdash \text{defNode } g \ q \text{--} ns' \rightarrow m \wedge (\text{defNode } g \ q) \notin \text{set } (tl \ ns') \wedge q \in \text{phiUses } g \ m \wedge m \in \text{set } (\text{predecessors } g \ (\text{defNode } g \ p)) \wedge n \in \text{set } ns' \wedge \text{set } ns' \subseteq \text{set } ns \wedge \text{defNode } g \ p \in \text{set } ns$   
**using**  $\text{assms}(1,2)$  **proof**  $(\text{induction rule}: \text{converse-tranclp-induct})$   
**case**  $(\text{base } p)$   
**from**  $\text{base.hyps } \text{base.prem}(2)$  **obtain**  $ns' \ m$  **where**  $g \vdash \text{defNode } g \ q \text{--} ns' \rightarrow m \ \text{defNode } g \ q \notin \text{set } (tl \ ns') \ m \in \text{set } (\text{predecessors } g \ (\text{defNode } g \ p)) \ q \in \text{phiUses } g \ m$   
**by** –  $(\text{rule } \text{phiArg-path-ex}', \text{rule } \text{assms}(3), \text{auto } \text{intro}: \text{simple-path2})$   
**hence**  $ns: g \vdash \text{defNode } g \ q \text{--} ns' @ [\text{defNode } g \ p] \rightarrow \text{defNode } g \ p \ \text{length } (ns' @ [\text{defNode } g \ p]) > 1$  **by**  $\text{auto}$

**show**  $?case$   
**proof**  $(\text{rule } \text{base.prem}(1)[\text{OF } ns, \text{rule-format}], \text{rule } \text{exI}, \text{rule } \text{exI}, \text{rule } \text{exI}, \text{rule } \text{exI})$   
**fix**  $n$

```

assume  $n \in \text{set } (\text{butlast } (ns' @ [\text{defNode } g \ p]))$ 
with  $\text{base.hyps } ns'$ 
show  $r \ p \ q \wedge$ 
 $g \vdash \text{defNode } g \ q - ns' \rightarrow m \wedge$ 
 $\text{defNode } g \ q \notin \text{set } (\text{tl } ns') \wedge$ 
 $q \in \text{phiUses } g \ m \wedge$ 
 $m \in \text{set } (\text{predecessors } g \ (\text{defNode } g \ p)) \wedge n \in \text{set } ns' \wedge \text{set } ns' \subseteq \text{set } (ns'$ 
 $@ [\text{defNode } g \ p]) \wedge \text{defNode } g \ p \in \text{set } (ns' @ [\text{defNode } g \ p])$ 
by auto
qed
next
case  $(\text{step } p \ p')$ 
from  $\text{step.premis}(2) \ \text{step.hyps}(1)$  obtain  $ns'_2 \ m$  where  $ns'_2: g \vdash \text{defNode } g$ 
 $p' - ns'_2 \rightarrow m \ m \in \text{set } (\text{predecessors } g \ (\text{defNode } g \ p)) \ \text{defNode } g \ p' \notin \text{set } (\text{tl } ns'_2) \ p'$ 
 $\in \text{phiUses } g \ m$ 
by  $-(\text{rule } \text{phiArg-path-ex}', \text{rule } \text{assms}(3), \text{auto } \text{intro: simple-path2})$ 
then obtain  $ns_2$  where  $ns_2: g \vdash \text{defNode } g \ p' - ns_2 \rightarrow \text{defNode } g \ p \ \text{length } ns_2$ 
 $> 1 \ ns_2 = ns'_2 @ [\text{defNode } g \ p]$  by  $(\text{atomize-elim}, \text{auto})$ 

show thesis
proof  $(\text{rule } \text{step.IH})$ 
fix  $ns$ 
assume  $ns: g \vdash \text{defNode } g \ q - ns \rightarrow \text{defNode } g \ p' \ 1 < \text{length } ns$ 
assume  $IH: \forall n \in \text{set } (\text{butlast } ns).$ 
 $\exists p \ q \ m \ ns'.$ 
 $r \ p \ q \wedge$ 
 $g \vdash \text{defNode } g \ q - ns' \rightarrow m \wedge$ 
 $\text{defNode } g \ q \notin \text{set } (\text{tl } ns') \wedge$ 
 $q \in \text{phiUses } g \ m \wedge m \in \text{set } (\text{predecessors } g \ (\text{defNode } g \ p)) \wedge n \in \text{set}$ 
 $ns' \wedge \text{set } ns' \subseteq \text{set } ns \wedge \text{defNode } g \ p \in \text{set } ns$ 

let  $?path = ns @ \text{tl } ns_2$ 
have  $ns - ns_2: g \vdash \text{defNode } g \ q - ?path \rightarrow \text{defNode } g \ p \ 1 < \text{length } ?path$  using
 $ns \ ns_2(1,2)$  by auto
show thesis
proof  $(\text{rule } \text{step.premis}(1)[\text{OF } ns - ns_2, \text{rule-format}])$ 
fix  $n$ 
assume  $n: n \in \text{set } (\text{butlast } ?path)$ 
show  $\exists p \ q \ m \ ns' a.$ 
 $r \ p \ q \wedge$ 
 $g \vdash \text{defNode } g \ q - ns' a \rightarrow m \wedge$ 
 $\text{defNode } g \ q \notin \text{set } (\text{tl } ns' a) \wedge$ 
 $q \in \text{phiUses } g \ m \wedge m \in \text{set } (\text{predecessors } g \ (\text{defNode } g \ p)) \wedge n \in \text{set } ns' a$ 
 $\wedge \text{set } ns' a \subseteq \text{set } ?path \wedge \text{defNode } g \ p \in \text{set } ?path$ 
proof  $(\text{cases } n \in \text{set } (\text{butlast } ns))$ 
case True
with  $IH$  obtain  $p \ q \ m \ ns'$  where
 $r \ p \ q \wedge$ 
 $g \vdash \text{defNode } g \ q - ns' \rightarrow m \wedge$ 

```

```

      defNode g q ∉ set (tl ns') ∧
      q ∈ phiUses g m ∧ m ∈ set (predecessors g (defNode g p)) ∧ n ∈ set
ns' ∧ set ns' ⊆ set ns ∧ defNode g p ∈ set ns by auto
      thus ?thesis by - (rule exI, rule exI, rule exI, rule exI, auto)
    next
      case False
      from ns ns2 have 1: ?path = butlast ns@ns2
      by - (rule concat-join[symmetric], auto simp: path2-def)
      from ns2(1) n False 1 have n ∈ set (butlast ns2) by (auto simp:
butlast-append path2-not-Nil)
      with step.hyps ns'2 ns2(3) show ?thesis
      by - (subst 1, rule exI[where x=p], rule exI[where x=p], rule exI,
rule exI, auto simp: path2-not-Nil)
    qed
  qed
next
  show p' ∈ allVars g using step.premis(2) step.hyps(1)[THEN assms(3)] by
auto
  qed
qed

```

**lemma** non-dominated-predecessor:

```

  assumes n ∈ set (αn g) n ≠ Entry g
  obtains m where m ∈ set (predecessors g n) ¬dominates g n m
proof-
  obtain ns where g ⊢ Entry g-ns→n
  by (atomize-elim, rule Entry-reaches, auto simp add:assms(1))
  then obtain ns' where ns': g ⊢ Entry g-ns'→n n ∉ set (butlast ns')
  by (rule simple-path2)
  let ?m = last (butlast ns')
  from ns'(1) assms(2) obtain m: g ⊢ Entry g-butlast ns'→?m ?m ∈ set
(predecessors g n)
  by - (rule path2-unsnoc, auto)
  with m(1) ns'(2) show thesis
  by - (rule that, auto elim:dominatesE)
qed

```

**lemmas** dominates-trans'[trans, elim] = dominates-trans[OF invar]

**lemmas** strict-dom-trans'[trans, elim] = strict-dom-trans[OF invar]

**lemmas** dominates-refl'[simp] = dominates-refl[OF invar]

**lemmas** dominates-antisymm'[dest] = dominates-antisymm[OF invar]

**lemma** liveVal-in-allVars[simp]: liveVal g v ⇒ v ∈ allVars g

**by** (induction rule: liveVal.induct, auto intro!: allUses-in-allVars)

**lemma** phi-no-closed-loop:

**assumes**[simp]: p ∈ allVars g **and** phi g p = Some vs

**shows** set vs ≠ {p}

**proof** (cases defNode g p = Entry g)

```

    case True
    with assms(2) show ?thesis by auto
next
case False
show ?thesis
proof
  assume[simp]: set vs = {p}
  let ?n = defNode g p
  obtain ns where ns: g ⊢ Entry g-ns→?n ?n ∉ set (butlast ns) by (rule
simple-Entry-path, auto)
  let ?m = last (butlast ns)
  from ns False obtain m: g ⊢ Entry g-butlast ns→?m ?m ∈ set (predecessors
g ?n)
  by - (rule path2-unsnoc, auto)
  hence p ∈ phiUses g ?m using assms(2) by - (rule phiUses-exI, auto
simp:phi-def)
  hence defAss g ?m p using m by - (rule allUses-def-ass, auto)
  then obtain l where l: l ∈ set (butlast ns) p ∈ allDefs g l using m by -
(drule defAssD, auto)
  with assms(2) m have l = ?n by - (rule allDefs-disjoint', auto)
  with ns l m show False by auto
qed
qed

```

```

lemma phis-phi: phis g (n, v) = Some vs ⇒ phi g v = Some vs
unfolding phi-def
apply (subst defNode-eq)
by (auto simp: allDefs-def phi-def phiDefs-def intro: phis-in-αn)

```

```

lemma trivial-phi: trivial g v ⇒ phi g v ≠ None
by (auto simp: trivial-def isTrivialPhi-def split: option.splits)

```

```

lemma trivial-finite: finite {v. trivial g v}
by (rule finite-subset[OF - phi-finite]) (auto dest: trivial-phi)

```

```

lemma trivial-in-allVars: trivial g v ⇒ v ∈ allVars g
by (drule trivial-phi, auto simp: allDefs-def phiDefs-def image-def phi-def intro:
phis-in-αn intro!: allDefs-in-allVars)

```

```

declare phiArg-def [simp del]
end

```

## 2.5 Bundling of CFG and Equivalent SSA CFG

```

locale CFG-SSA-Transformed-base = old: CFG-base αe αn invar inEdges' Entry
oldDefs oldUses + CFG-SSA-wf-base αe αn invar inEdges' Entry defs uses phis
for
  αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set and
  αn :: 'g ⇒ 'node list and

```

*invar* :: 'g ⇒ bool **and**  
*inEdges'* :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list **and**  
*Entry*::'g ⇒ 'node **and**  
*oldDefs* :: 'g ⇒ 'node ⇒ 'var::linorder set **and**  
*oldUses* :: 'g ⇒ 'node ⇒ 'val set **and**  
*defs* :: 'g ⇒ 'node ⇒ 'val::linorder set **and**  
*uses* :: 'g ⇒ 'node ⇒ 'val set **and**  
*phis* :: 'g ⇒ ('node, 'val) phis +  
**fixes** var :: 'g ⇒ 'val ⇒ 'var

**locale** *CFG-SSA-Transformed* = *CFG-SSA-Transformed-base* αe αn *invar inEdges' Entry oldDefs oldUses defs uses phis var*  
 + *old*: *CFG-wf* αe αn *invar inEdges' Entry oldDefs oldUses* + *CFG-SSA-wf* αe  
 αn *invar inEdges' Entry defs uses phis*

**for**

αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set **and**  
 αn :: 'g ⇒ 'node list **and**  
*invar* :: 'g ⇒ bool **and**  
*inEdges'* :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list **and**  
*Entry*::'g ⇒ 'node **and**  
*oldDefs* :: 'g ⇒ 'node ⇒ 'var::linorder set **and**  
*oldUses* :: 'g ⇒ 'node ⇒ 'val set **and**  
*defs* :: 'g ⇒ 'node ⇒ 'val::linorder set **and**  
*uses* :: 'g ⇒ 'node ⇒ 'val set **and**  
*phis* :: 'g ⇒ ('node, 'val) phis **and**  
 var :: 'g ⇒ 'val ⇒ 'var +  
**assumes** *oldDefs-def*: *oldDefs* g n = var g ' *defs* g n  
**assumes** *oldUses-def*: n ∈ set (αn g) ⇒ *oldUses* g n = var g ' *uses* g n  
**assumes** *conventional*:

$\llbracket g \vdash n - ns \rightarrow m; n \notin \text{set } (tl \ ns); v \in \text{allDefs } g \ n; v \in \text{allUses } g \ m; x \in \text{set } (tl \ ns); v' \in \text{allDefs } g \ x \rrbracket \Longrightarrow \text{var } g \ v' \neq \text{var } g \ v$   
**assumes** *phis-same-var[elim]*: *phis* g (n, v) = Some vs ⇒ v' ∈ set vs ⇒ var g v' = var g v

**assumes** *allDefs-var-disjoint*:  $\llbracket n \in \text{set } (\alpha n \ g); v \in \text{allDefs } g \ n; v' \in \text{allDefs } g \ n; v \neq v' \rrbracket \Longrightarrow \text{var } g \ v' \neq \text{var } g \ v$

**begin**

**lemma** *conventional'*:  $\llbracket g \vdash n - ns \rightarrow m; n \notin \text{set } (tl \ ns); v \in \text{allDefs } g \ n; v \in \text{allUses } g \ m; v' \in \text{allDefs } g \ x; \text{var } g \ v' = \text{var } g \ v \rrbracket \Longrightarrow x \notin \text{set } (tl \ ns)$   
**using** *conventional* **by** *auto*

**lemma** *conventional''*:  $\llbracket g \vdash \text{defNode } g \ v - ns \rightarrow m; \text{defNode } g \ v \notin \text{set } (tl \ ns); v \in \text{allUses } g \ m; \text{var } g \ v' = \text{var } g \ v; v \in \text{allVars } g; v' \in \text{allVars } g \rrbracket \Longrightarrow \text{defNode } g \ v' \notin \text{set } (tl \ ns)$

**by** (rule *conventional'* [where v=v and v'=v], *auto*)

**lemma** *phiArg-same-var*: *phiArg* g p q ⇒ var g q = var g p  
**by** (*metis phiArg-def phi-def phis-same-var*)

**lemma** *oldDef-defAss*:

```

assumes  $v \in \text{allUses } g \ n \ g \vdash \text{Entry } g - ns \rightarrow n$ 
obtains  $m$  where  $m \in \text{set } ns \ \text{var } g \ v \in \text{oldDefs } g \ m$ 
using assms proof (induction ns arbitrary: v n rule: length-induct)
  case (1 ns)
    from 1.prem(2-) have 2:  $\text{defNode } g \ v \in \text{set } ns$ 
      by - (rule defAss-defNode, rule allUses-def-ass, auto)
    let  $?V = \text{defNode } g \ v$ 
    from 1.prem(2,3) have[simp]:  $v \in \text{allVars } g$  by auto
    thus ?case
  proof (cases v rule: defNode-cases)
    case simpleDef
      with 2 show thesis by - (rule 1.prem(1), auto simp: oldDefs-def)
  next
    case phi
      then obtain  $vs$  where  $vs: \text{phi } g \ v = \text{Some } vs$  by auto
      from 1.prem(3) 2 obtain  $ns'$  where  $ns': g \vdash \text{Entry } g - ns' \rightarrow ?V \ \text{prefix } ns'$ 
ns
        by (rule old.path2-split-ex, auto)
      let  $?V' = \text{last } (\text{butlast } ns')$ 
      from  $ns'$  phi have nontriv:  $\text{length } ns' \geq 2$ 
        by - (rule old.path2-nontrivial, auto)
      hence 3:  $g \vdash \text{Entry } g - \text{butlast } ns' \rightarrow ?V' \ ?V' \in \text{set } (\text{old.predecessors } g \ ?V)$ 
        using  $ns'(1)$  by (auto intro: old.path2-unsnoc)
      with phi vs obtain  $v'$  where  $v': v' \in \text{phiUses } g \ ?V' \ \text{var } g \ v' = \text{var } g \ v$ 
        by - (rule phiArg-exI, auto simp: phi-def phi-same-var phiArg-def)
      show thesis
      proof (rule 1.IH[rule-format])
        show  $\text{length } (\text{butlast } ns') < \text{length } ns$  using  $ns'$  by (cases ns', auto simp: old.path2-not-Nil2 dest: prefix-length-le)
        show  $v' \in \text{allUses } g \ ?V'$  using  $v'(1)$  by simp
      next
        fix  $n$ 
        assume  $n \in \text{set } (\text{butlast } ns') \ \text{var } g \ v' \in \text{oldDefs } g \ n$ 
        thus thesis
        using  $ns'(2)$ [THEN set-mono-prefix]  $v'(2)$  by - (rule 1.prem(1)[of n], auto dest: in-set-butlastD)
      qed (rule 3(1))
    qed
  qed

lemma allDef-path-from-simpleDef:
  assumes[simp]:  $v \in \text{allVars } g$ 
  obtains  $n \ ns$  where  $g \vdash n - ns \rightarrow \text{defNode } g \ v \ \text{old.EntryPath } g \ ns \ \text{var } g \ v \in \text{oldDefs } g \ n$ 
proof -
  let  $?V = \text{defNode } g \ v$ 
  from assms obtain  $ns$  where  $ns: g \vdash \text{Entry } g - ns \rightarrow ?V \ \text{old.EntryPath } g \ ns$ 
    by - (rule old.Entry-reachesE, auto)
  from assms show thesis

```

```

proof (cases v rule: defNode-cases)
  case simpleDef
  thus thesis by - (rule that, auto simp: oldDefs-def)
next
case phi
then obtain vs where vs: phi g v = Some vs by auto
let ?V' = last (butlast ns)
from ns phi have nontriv: length ns ≥ 2
  by - (rule old.path2-nontrivial, auto)
hence ∃: g ⊢ Entry g-butlast ns → ?V' ?V' ∈ set (old.predecessors g ?V)
  using ns(1) by (auto intro: old.path2-unsnoc)
with phi vs obtain v' where v': v' ∈ phiUses g ?V' var g v' = var g v
  by - (rule phiArg-exI, auto simp: phi-def phi-same-var phiArg-def)
with ∃(1) obtain n where n: n ∈ set (butlast ns) var g v' ∈ oldDefs g n
  by - (rule oldDef-defAss[of v' g], auto)
with ns obtain ns' where g ⊢ n - ns' → ?V suffix ns' ns
  by - (rule old.path2-split-ex'[OF ns(1)], auto intro: in-set-butlastD simp:
Sublist.suffix-def)
with n(2) v'(2) ns(2) show thesis
  by - (rule that, assumption, erule old.EntryPath-suffix, auto)
qed
qed

```

```

lemma defNode-var-disjoint:
  assumes p ∈ allVars g q ∈ allVars g p ≠ q defNode g p = defNode g q
  shows var g p ≠ var g q
proof -
  have q ∈ allDefs g (defNode g p) using assms(2) assms(4) by (auto)
  thus ?thesis using assms(1-3)
  by - (rule allDefs-var-disjoint[of defNode g p g], auto)
qed

```

```

lemma phiArg-distinct-nodes:
  assumes phiArg g p q p ≠ q and[simp]: p ∈ allVars g
  shows defNode g p ≠ defNode g q
proof
  have p ∈ allDefs g (defNode g p) by simp
  moreover assume defNode g p = defNode g q
  ultimately have var g p ≠ var g q using assms
  by - (rule defNode-var-disjoint, auto)
  moreover
  from assms(1) have var g q = var g p by (rule phiArg-same-var)
  ultimately show False by simp
qed

```

```

lemma phiArgs-def-distinct:
  assumes phiArg g p q phiArg g p r q ≠ r p ∈ allVars g
  shows defNode g q ≠ defNode g r
proof (rule)

```

**assume**  $\text{defNode } g \ q = \text{defNode } g \ r$   
**hence**  $\text{var } g \ q \neq \text{var } g \ r$  **using**  $\text{assms}$  **by**  $-(\text{rule } \text{defNode-var-disjoint}, \text{auto})$   
**thus**  $\text{False}$  **using**  $\text{phiArg-same-var}[\text{OF } \text{assms}(1)] \ \text{phiArg-same-var}[\text{OF } \text{assms}(2)]$   
**by**  $\text{simp}$   
**qed**

**lemma**  $\text{defNode-not-on-defUse-path}$ :

**assumes**  $p: g \vdash \text{defNode } g \ p - \text{ns} \rightarrow n \ \text{defNode } g \ p \notin \text{set } (\text{tl } \text{ns}) \ p \in \text{allUses } g \ n$   
**assumes** $[\text{simp}]$ :  $q \in \text{allVars } g \ p \neq q \ \text{var } g \ p = \text{var } g \ q$   
**shows**  $\text{defNode } g \ q \notin \text{set } \text{ns}$

**proof**–

**let**  $?P = \text{defNode } g \ p$   
**let**  $?Q = \text{defNode } g \ q$

**have** $[\text{simp}]$ :  $p \in \text{allVars } g$  **using**  $p(1,3)$  **by**  $\text{auto}$

**have**  $?P \neq ?Q$  **using**  $\text{defNode-var-disjoint}[\text{of } p \ q]$  **by**  $\text{auto}$

**moreover** **have**  $?Q \notin \text{set } (\text{tl } \text{ns})$  **using**  $p(2,3)$

**by**  $-(\text{rule } \text{conventional}[\text{OF } p(1), \text{of } p \ q], \text{auto})$

**ultimately show**  $?thesis$  **using**  $p(1)$  **by**  $(\text{cases } \text{ns}, \text{auto } \text{simp}: \text{old.path2-def})$

**qed**

**lemma**  $\text{defUse-paths-disjoint}$ :

**assumes**  $p: g \vdash \text{defNode } g \ p - \text{ns} \rightarrow n \ \text{defNode } g \ p \notin \text{set } (\text{tl } \text{ns}) \ p \in \text{allUses } g \ n$   
**assumes**  $q: g \vdash \text{defNode } g \ q - \text{ms} \rightarrow m \ \text{defNode } g \ q \notin \text{set } (\text{tl } \text{ms}) \ q \in \text{allUses } g$

$m$

**assumes** $[\text{simp}]$ :  $p \neq q \ \text{var } g \ p = \text{var } g \ q$

**shows**  $\text{set } \text{ns} \cap \text{set } \text{ms} = \{\}$

**proof**  $(\text{rule } \text{equals}0I)$

**fix**  $y$

**assume**  $y: y \in \text{set } \text{ns} \cap \text{set } \text{ms}$

{

**fix**  $p \ \text{ns} \ n$

**assume**  $p: g \vdash \text{defNode } g \ p - \text{ns} \rightarrow n \ \text{defNode } g \ p \notin \text{set } (\text{tl } \text{ns}) \ p \in \text{allUses } g \ n$

**assume**  $y: y \in \text{set } \text{ns}$

**from**  $p(1,3)$  **have**  $\text{dom}: \text{old.dominates } g \ (\text{defNode } g \ p) \ n$  **by**  $-(\text{rule } \text{allUses-dominated}, \text{auto})$

**moreover**

**obtain**  $\text{ns}'$  **where**  $g \vdash y - \text{ns}' \rightarrow n \ \text{suffix } \text{ns}' \ \text{ns}$

**by**  $(\text{rule } \text{old.path2-split-first-last}[\text{OF } p(1) \ y], \text{auto})$

**ultimately have**  $\text{old.dominates } g \ (\text{defNode } g \ p) \ y$  **using**  $\text{suffix-tl-subset}[\text{of } \text{ns}' \ \text{ns}] \ p(2)$

**by**  $-(\text{rule } \text{old.dominates-extend}[\text{where } \text{ms}=\text{ns}'], \text{auto})$

}

**with**  $\text{assms } y$  **have**  $\text{dom}: \text{old.dominates } g \ (\text{defNode } g \ p) \ y \ \text{old.dominates } g \ (\text{defNode } g \ q) \ y$  **by**  $\text{auto}$

{

**fix**  $p \ \text{ns} \ n \ q \ \text{ms} \ m$



```

let ?P = defNode g p
let ?Q = defNode g q

assume p: g ⊢ defNode g p-ns→n defNode g p ∉ set (tl ns) p ∈ allUses g n
old.dominates g ?P y y ∈ set ns
assume q: g ⊢ defNode g q-ms→m defNode g q ∉ set (tl ms) q ∈ allUses g
m old.dominates g ?Q y y ∈ set ms
assume[simp]: p ≠ q var g p = var g q
assume dom: old.dominates g ?P ?Q
then obtain pqs where pqs: g ⊢ ?P-pqs→?Q ?P ∉ set (tl pqs) by (rule
old.dominates-path, auto intro: old.simple-path2)
from p obtain ns2 where ns2: g ⊢ y-ns2→n suffix ns2 ns by - (rule
old.path2-split-first-last, auto)
from q obtain ms1 where ms1: g ⊢ ?Q-ms1→y prefix ms1 ms by - (rule
old.path2-split-first-last, auto)
have var g q ≠ var g p
proof (rule conventional[OF - - p(3)])
let ?path = (pqs@tl ms1)@tl ns2
show g ⊢ ?P-?path→n using pqs ms1 ns2
by (auto simp del:append-assoc intro:old.path2-app)
have ?P ∉ set (tl ns2) using p(2) ns2(2)[THEN suffix-tl-subset, THEN
subsetD] by auto
moreover
have[simp]: q ∈ allVars g p ∈ allVars g using p q by auto
have ?P ∉ set (tl ms) using q
by - (rule conventional'[where v'=p and v=q], auto)
hence ?P ∉ set (tl ms1) using ms1(2)[simplified, THEN prefix-tl-subset]
by auto
ultimately
show ?P ∉ set (tl ?path) using pqs(2)
by - (rule notI, auto dest: subsetD[OF set-tl-append])
show p ∈ allDefs g (defNode g p) by auto
have ?P ≠ ?Q using defNode-var-disjoint[of p g q] by auto
hence 1: length pqs > 1 using pqs by - (rule old.path2-nontriv)
hence ?Q ∈ set (tl pqs) using pqs unfolding old.path2-def by (auto
intro:last-in-tl)
moreover from 1 have pqs ≠ [] by auto
ultimately show ?Q ∈ set (tl ?path) by simp
show q ∈ allDefs g ?Q by simp
qed
hence False by simp
}
from this[OF p - - q] this[OF q - - p] y dom show False
by - (rule old.dominates-antitrans[OF - dom], auto)
qed

lemma oldDefsI: v ∈ defs g n ⇒ var g v ∈ oldDefs g n by (simp add:
oldDefs-def)

```

```

lemma simpleDefs-phiDefs-var-disjoint:
  assumes  $v \in \text{phiDefs } g \ n \ n \in \text{set } (\alpha n \ g)$ 
  shows  $\text{var } g \ v \notin \text{oldDefs } g \ n$ 
proof
  from assms have[simp]:  $v \in \text{allVars } g$  by auto
  assume  $\text{var } g \ v \in \text{oldDefs } g \ n$ 
  then obtain  $v''$  where  $v'': v'' \in \text{defs } g \ n \ \text{var } g \ v'' = \text{var } g \ v$ 
    by (auto simp: oldDefs-def)
  from this(1) assms have  $v'' \neq v$ 
    using simpleDefs-phiDefs-disjoint[of n g] by (auto simp: phiArg-def)
  with  $v''$  assms show False
    using allDefs-var-disjoint[of n g v'' v] by auto
qed

lemma liveVal-use-path:
  assumes liveVal  $g \ v$ 
  obtains  $ns \ m$  where  $g \vdash \text{defNode } g \ v \text{-} ns \rightarrow m \ \text{var } g \ v \in \text{oldUses } g \ m$ 
     $\bigwedge x. x \in \text{set } (tl \ ns) \implies \text{var } g \ v \notin \text{oldDefs } g \ x$ 
using assms proof (induction)
  case (liveSimple  $m \ v$ )
  from liveSimple.hyps have[simp]:  $v \in \text{allVars } g$ 
    by - (rule allUses-in-allVars, auto)
  from liveSimple.hyps obtain  $ns$  where  $ns: g \vdash \text{defNode } g \ v \text{-} ns \rightarrow m \ \text{defNode}$ 
 $g \ v \notin \text{set } (tl \ ns)$ 
    by - (rule defUse-path-ex, auto intro!: uses-in-allUses elim: old.simple-path2)
  from this(1) show thesis
proof (rule liveSimple.prems)
  show  $\text{var } g \ v \in \text{oldUses } g \ m$  using liveSimple.hyps by (auto simp: oldUses-def)
  {
    fix  $x$ 
    assume asm:  $x \in \text{set } (tl \ ns) \ \text{var } g \ v \in \text{oldDefs } g \ x$ 
    then obtain  $v'$  where  $v' \in \text{defs } g \ x \ \text{var } g \ v' = \text{var } g \ v$ 
      by (auto simp: oldDefs-def)
    with asm liveSimple.hyps have False
      by - (rule conventional[OF ns, of v x v', THEN notE], auto)
  }
  thus  $\bigwedge x. x \in \text{set } (tl \ ns) \implies \text{var } g \ v \notin \text{oldDefs } g \ x$  by auto
qed
next
  case (livePhi  $v \ v'$ )
  from livePhi.hyps have[simp]:  $v \in \text{allVars } g \ v' \in \text{allVars } g \ \text{var } g \ v' = \text{var } g \ v$ 
    by (auto intro: phiArg-same-var)
  show thesis
proof (rule livePhi.IH)
  fix  $ns \ m$ 
  assume asm:  $g \vdash \text{defNode } g \ v \text{-} ns \rightarrow m \ \text{var } g \ v \in \text{oldUses } g \ m$ 
     $\bigwedge x. x \in \text{set } (tl \ ns) \implies \text{var } g \ v \notin \text{oldDefs } g \ x$ 
  from livePhi.hyps(2) obtain  $ns' \ m'$  where  $ns': g \vdash \text{defNode } g \ v' \text{-} ns' \rightarrow m'$ 
 $v' \in \text{phiUses } g \ m'$ 

```

```

    m' ∈ set (old.predecessors g (defNode g v)) defNode g v' ∉ set (tl ns')
  by (rule phiArg-path-ex', auto elim: old.simple-path2)
show thesis
proof (rule livePhi.premis)
  show g ⊢ defNode g v'-(ns'@[defNode g v])@tl ns→m
  apply (rule old.path2-app)
  apply (rule old.path2-snoc[OF ns'(1,3)])
  by (rule asm(1))
  show var g v' ∈ oldUses g m using asm(2) by simp
  {
    fix x
    assume asm: x ∈ set (tl ns') var g v ∈ oldDefs g x
    then obtain v'' where v'' ∈ defs g x var g v'' = var g v
      by (auto simp: oldDefs-def)
    with asm ns'(2) have False
      by - (rule conventional[OF ns'(1,4), of v' x v'', THEN notE], auto)
  }
  then show ∧x. x ∈ set (tl ((ns'@[defNode g v])@tl ns)) ⇒ var g v' ∉
oldDefs g x
  using simpleDefs-phiDefs-var-disjoint[of v g defNode g v] livePhi.hyps(2)
  by (auto dest!: set-tl-append'[THEN subsetD] asm(3) simp: phiArg-def)
qed
qed
qed
end
end
end

```

### 3 Minimality

We show that every reducible CFG without trivial  $\phi$  functions is minimal, recreating the proof in [2]. The original proof is inlined as prose text.

```

theory Minimality
imports SSA-CFG Serial-Rel
begin

```

```

context graph-path
begin

```

Cytron's definition of path convergence

```

definition pathsConverge g x xs y ys z ≡ g ⊢ x-xs→z ∧ g ⊢ y-ys→z ∧ length
xs > 1 ∧ length ys > 1 ∧ x ≠ y ∧
(∀ j ∈ {0..< length xs}. ∀ k ∈ {0..< length ys}. xs ! j = ys ! k → j = length
xs - 1 ∨ k = length ys - 1)

```

Simplified definition

```

definition pathsConverge' g x xs y ys z ≡ g ⊢ x-xs→z ∧ g ⊢ y-ys→z ∧ length
xs > 1 ∧ length ys > 1 ∧ x ≠ y ∧

```

$set (butlast\ xs) \cap set (butlast\ ys) = \{\}$

**lemma** *pathsConverge'[simp]*:  $pathsConverge\ g\ x\ xs\ y\ ys\ z \longleftrightarrow pathsConverge'\ g\ x\ xs\ y\ ys\ z$

**proof**–

**have**  $(\forall j \in \{0..<length\ xs\}. \forall k \in \{0..<length\ ys\}. xs\ !\ j = ys\ !\ k \longrightarrow j = length\ xs - 1 \vee k = length\ ys - 1)$

$\longleftrightarrow (\forall x' \in set (butlast\ xs). \forall y' \in set (butlast\ ys). x' \neq y')$

**proof**

**assume**  $1: \forall j \in \{0..<length\ xs\}. \forall k \in \{0..<length\ ys\}. xs\ !\ j = ys\ !\ k \longrightarrow j = length\ xs - 1 \vee k = length\ ys - 1$

**show**  $\forall x' \in set (butlast\ xs). \forall y' \in set (butlast\ ys). x' \neq y'$

**proof** (*rule, rule, rule*)

**fix**  $x'\ y'$

**assume**  $2: x' \in set (butlast\ xs)\ y' \in set (butlast\ ys)$  **and**[*simp*]:  $x' = y'$

**from**  $2(1)$  **obtain**  $j$  **where**  $j: xs\ !\ j = x'\ j < length\ xs - 1$  **by** (*rule butlast-idx*)

**moreover from**  $j$  **have**  $j < length\ xs$  **by** *simp*

**moreover from**  $2(2)$  **obtain**  $k: ys\ !\ k = y'\ k < length\ ys - 1$  **by** (*rule butlast-idx*)

**moreover from**  $k$  **have**  $k < length\ ys$  **by** *simp*

**ultimately show** *False* **using**  $1[THEN\ bspec[where\ x=j], THEN\ bspec[where\ x=k]]$  **by** *auto*

**qed**

**next**

**assume**  $1: \forall x' \in set (butlast\ xs). \forall y' \in set (butlast\ ys). x' \neq y'$

**show**  $\forall j \in \{0..<length\ xs\}. \forall k \in \{0..<length\ ys\}. xs\ !\ j = ys\ !\ k \longrightarrow j = length\ xs - 1 \vee k = length\ ys - 1$

**proof** (*rule, rule, rule, simp*)

**fix**  $j\ k$

**assume**  $2: j < length\ xs\ k < length\ ys\ xs\ !\ j = ys\ !\ k$

**show**  $j = length\ xs - Suc\ 0 \vee k = length\ ys - Suc\ 0$

**proof** (*rule ccontr, simp*)

**assume**  $3: j \neq length\ xs - Suc\ 0 \wedge k \neq length\ ys - Suc\ 0$

**let**  $?x' = xs\ !\ j$

**let**  $?y' = ys\ !\ k$

**from**  $2(1)$   $3$  **have**  $?x' \in set (butlast\ xs)$  **by** – (*rule butlast-idx', auto*)

**moreover from**  $2(2)$   $3$  **have**  $?y' \in set (butlast\ ys)$  **by** – (*rule butlast-idx', auto*)

**ultimately have**  $?x' \neq ?y'$  **using**  $1$  **by** *simp*

**with**  $2(3)$  **show** *False* **by** *simp*

**qed**

**qed**

**qed**

**thus** *?thesis* **by** (*auto simp:pathsConverge-def pathsConverge'-def*)

**qed**

**lemma** *pathsConvergeI*:

**assumes**  $g \vdash x \rightarrow xs \rightarrow z\ g \vdash y \rightarrow ys \rightarrow z\ length\ xs > 1\ length\ ys > 1\ set (butlast$

$xs) \cap \text{set } (\text{butlast } ys) = \{\}$   
**shows** *pathsConverge*  $g\ x\ xs\ y\ ys\ z$   
**proof**–  
**from** *assms* **have**  $x \neq y$   
**by** (*metis append-is-Nil-conv disjoint-iff-not-equal length-butlast list.collapse list.distinct(1) nth-Cons-0 nth-butlast nth-mem path2-def split-list zero-less-diff*)  
**with** *assms* **show** *?thesis* **by** (*simp add:pathsConverge'-def*)  
**qed**  
**end**

A (control) flow graph  $G$  is reducible iff for each cycle  $C$  of  $G$  there is a node of  $C$  that dominates all other nodes in  $C$ .

**definition** (in *graph-Entry*) *reducible*  $g \equiv \forall n\ ns. g \vdash n - ns \rightarrow n \longrightarrow (\exists m \in \text{set } ns. \forall n \in \text{set } ns. \text{dominates } g\ m\ n)$

**context** *CFG-SSA-Transformed*  
**begin**

A  $\phi$  function for variable  $v$  is necessary in block  $Z$  iff two non-null paths  $X \rightarrow^+ Z$  and  $Y \rightarrow^+ Z$  converge at a block  $Z$ , such that the blocks  $X$  and  $Y$  contain assignments to  $v$ .

**definition** *necessaryPhi*  $g\ v\ z \equiv \exists n\ ns\ m\ ms. \text{old.pathsConverge } g\ n\ ns\ m\ ms\ z \wedge v \in \text{oldDefs } g\ n \wedge v \in \text{oldDefs } g\ m$

**abbreviation** *necessaryPhi'*  $g\ val \equiv \text{necessaryPhi } g\ (\text{var } g\ val)\ (\text{defNode } g\ val)$

**definition** *unnecessaryPhi*  $g\ val \equiv \text{phi } g\ val \neq \text{None} \wedge \neg \text{necessaryPhi}'\ g\ val$

**lemma** *necessaryPhiI*:  $\text{old.pathsConverge } g\ n\ ns\ m\ ms\ z \implies v \in \text{oldDefs } g\ n \implies v \in \text{oldDefs } g\ m \implies \text{necessaryPhi } g\ v\ z$

**by** (*auto simp: necessaryPhi-def*)

A program with only necessary  $\phi$  functions is in minimal SSA form.

**definition** *cytronMinimal*  $g \equiv \forall v \in \text{allVars } g. \text{phi } g\ v \neq \text{None} \longrightarrow \text{necessaryPhi}'\ g\ v$

Let  $p$  be a  $\phi$  function in a block  $P$ . Furthermore, let  $q$  in a block  $Q$  and  $r$  in a block  $R$  be two operands of  $p$ , such that  $p$ ,  $q$  and  $r$  are pairwise distinct. Then at least one of  $Q$  and  $R$  does not dominate  $P$ .

**lemma** 2:

**assumes** *phiArg*  $g\ p\ q\ phiArg\ g\ p\ r$  *distinct*  $[p, q, r]$  **and**[*simp*]:  $p \in \text{allVars } g$   
**shows**  $\neg(\text{def-dominates } g\ q\ p \wedge \text{def-dominates } g\ r\ p)$

**proof** (*rule, erule conjE*)

Proof. Assume that  $Q$  and  $R$  dominate  $P$ , i.e., every path from the start block to  $P$  contains  $Q$  and  $R$ .

**assume** *asm*:  $\text{def-dominates } g\ q\ p \wedge \text{def-dominates } g\ r\ p$

Since immediate dominance forms a tree,  $Q$  dominates  $R$  or  $R$  dominates  $Q$ .

**hence**  $\text{def-dominates } g\ q\ r \vee \text{def-dominates } g\ r\ q$

**by**  $-\text{(rule old.dominates-antitrans[of } g \text{ defNode } g \text{ } q \text{ defNode } g \text{ } p \text{ defNode } g \text{ } r], \text{ auto})$

**moreover**

{

Without loss of generality, let Q dominate R.

**fix**  $q \ r$

**assume**  $assms: \text{phiArg } g \ p \ q \ \text{phiArg } g \ p \ r \ \text{distinct } [p, q, r]$

**assume**  $asm: \text{def-dominates } g \ q \ p \ \text{def-dominates } g \ r \ p$

**assume**  $wlog: \text{def-dominates } g \ q \ r$

**have** $[simp]: \text{var } g \ q = \text{var } g \ r$  **using**  $\text{phiArg-same-var}[OF \ assms(1)] \ \text{phiArg-same-var}[OF \ assms(2)]$  **by**  $simp$

Furthermore, let S be the corresponding predecessor block of P where p is using q.

**obtain**  $S$  **where**  $S: q \in \text{phiUses } g \ S \ S \in \text{set } (\text{old.predecessors } g \ (\text{defNode } g \ p))$  **by**  $(\text{rule } \text{phiUses-exI}[OF \ assms(1)], \ \text{simp})$

Then there is a path from the start block crossing Q then R and S.

**have**  $\text{defNode } g \ p \neq \text{defNode } g \ q$  **using**  $assms(1,3)$

**by**  $-\text{(rule } \text{phiArg-distinct-nodes}, \ \text{auto})$

**with**  $S$  **have**  $\text{old.dominates } g \ (\text{defNode } g \ q) \ S$

**by**  $-\text{(rule } \text{allUses-dominated}, \ \text{auto})$

**then obtain**  $ns$  **where**  $ns: g \vdash \text{defNode } g \ q \rightarrow ns \rightarrow S \ \text{distinct } ns$

**by**  $(\text{rule } \text{old.dominates-path}, \ \text{auto } \text{elim: } \text{old.simple-path2})$

**moreover have**  $\text{defNode } g \ r \in \text{set } (\text{tl } ns)$

**proof** $-\text{}$

**have**  $\text{defNode } g \ r \neq \text{defNode } g \ q$  **using**  $assms$

**by**  $-\text{(rule } \text{phiArgs-def-distinct}, \ \text{auto})$

**hence**  $\text{hd } ns \neq \text{defNode } g \ r$  **using**  $ns$  **by**  $(\text{auto } \text{simp:old.path2-def})$

**moreover**

**have**  $\text{defNode } g \ p \neq \text{defNode } g \ r$  **using**  $assms(2,3)$

**by**  $-\text{(rule } \text{phiArg-distinct-nodes}, \ \text{auto})$

**with**  $S(2)$  **have**  $\text{old.dominates } g \ (\text{defNode } g \ r) \ S$

**by**  $-\text{(rule } \text{old.dominates-unsnoc}[\text{where } m = \text{defNode } g \ p], \ \text{auto } \text{simp:wlog } \text{asm } \text{assms})$

**with**  $wlog$  **have**  $\text{defNode } g \ r \in \text{set } ns$  **using**  $ns(1)$

**by**  $(\text{rule } \text{old.dominates-mid}, \ \text{auto})$

**ultimately**

**show**  $?thesis$  **by**  $(\text{metis } \text{append-Nil } \text{in-set-conv-decomp } \text{list.sel}(1) \ \text{tl-append2})$

**qed**

This violates the SSA property.

**moreover have**  $q \in \text{allDefs } g \ (\text{defNode } g \ q)$  **using**  $assms \ S(1)$  **by**  $simp$

**moreover have**  $r \in \text{allDefs } g \ (\text{defNode } g \ r)$  **using**  $assms \ S(1)$  **by**  $simp$

**ultimately have**  $\text{var } g \ r \neq \text{var } g \ q$  **using**  $S(1)$

**by**  $-\text{(rule } \text{conventional}, \ \text{auto } \text{simp:old.path2-def } \text{distinct-hd-tl})$

**hence**  $False$  **by**  $simp$

}

**ultimately show *False* using *assms asm* by *auto***  
**qed**

**lemma *convergence-prop*:**

**assumes *necessaryPhi g (var g v) n g*  $\vdash$  *n-ns $\rightarrow$ m v*  $\in$  *allUses g m*  $\wedge$  *x. x*  $\in$  *set (tl ns)*  $\implies$  *v*  $\notin$  *allDefs g x* *v*  $\notin$  *defs g n***  
**shows *phis g (n,v)*  $\neq$  *None***

**proof**

**from *assms(2, 3)* have *v*  $\in$  *allVars g* by *auto***  
**hence *1: v*  $\in$  *allDefs g (defNode g v)* by (*rule defNode*)**

**assume *phis g (n,v)* = *None***  
**with *assms(5)* have *2: v*  $\notin$  *allDefs g n***  
**by (*auto simp:allDefs-def phiDefs-def*)**

**from *assms(1)* obtain *a as b bs v<sub>a</sub> v<sub>b</sub>* where**

***a: v<sub>a</sub>*  $\in$  *defs g a var g v<sub>a</sub>* = *var g v* and**  
***b: v<sub>b</sub>*  $\in$  *defs g b var g v<sub>b</sub>* = *var g v***

**and *conv: g*  $\vdash$  *a-as $\rightarrow$ n g*  $\vdash$  *b-bs $\rightarrow$ n g*  $1 <$  *length as*  $1 <$  *length bs* *a*  $\neq$  *b* *set (butlast as)*  $\cap$  *set (butlast bs)* =  $\{\}$**

**by (*auto simp:necessaryPhi-def old.pathsConverge'-def oldDefs-def*)**

**have *old.dominates g (defNode g v) m* using *assms(2,3)***

**by - (*rule allUses-dominated, auto*)**

**hence *dom: old.dominates g (defNode g v) n* using *assms(2,4) 1***

**by - (*rule old.dominates-unsnoc', auto*)**

**hence *old.strict-dom g (defNode g v) n* using *1 2* by *auto***

**{**

**fix *v<sub>a</sub> a as v<sub>b</sub> b bs***

**assume *a: v<sub>a</sub>*  $\in$  *defs g a var g v<sub>a</sub>* = *var g v***

**assume *as: g*  $\vdash$  *a-as $\rightarrow$ n g* *length as*  $>$  *1***

**assume *b: v<sub>b</sub>*  $\in$  *defs g b var g v<sub>b</sub>* = *var g v***

**assume *bs: g*  $\vdash$  *b-bs $\rightarrow$ n g***

**assume *conv: a*  $\neq$  *b* *set (butlast as)*  $\cap$  *set (butlast bs)* =  $\{\}$**

**have *3: defNode g v*  $\neq$  *a***

**proof**

**assume *contr: defNode g v* = *a***

**have *a*  $\in$  *set (butlast as)* using *as* by (*auto simp:old.path2-def intro:hd-in-butlast*)**

**hence *a*  $\notin$  *set (butlast bs)* using *conv(2)* by *auto***

**moreover**

**have *a*  $\neq$  *n* using *1 2 contr* by *auto***

**hence *a*  $\neq$  *last bs* using *bs* by (*auto simp:old.path2-def*)**

**ultimately have *4: a*  $\notin$  *set bs***

**by - (*subst append-butlast-last-id[symmetric]*, *rule old.path2-not-Nil[OF bs]*, *auto*)**

**have *v*  $\neq$  *v<sub>a</sub>***

```

proof
  assume asm:  $v = v_a$ 
  have  $v \neq v_b$ 
  proof
    assume  $v = v_b$ 
    with asm[symmetric] b(1) have  $v_a \in \text{allDefs } g \ b$  by simp
    with asm have  $a = b$  using as bs a(1) by  $\text{-- (rule allDefs-disjoint',$ 
auto)
    with conv(1) show False by simp
  qed
obtain ebs where  $ebs: g \vdash \text{Entry } g \text{--} ebs \rightarrow b$ 
  using bs by (atomize, auto)
hence  $g \vdash \text{Entry } g \text{--} \text{butlast } ebs @ bs \rightarrow n$  using bs by auto
hence  $5: a \in \text{set } (\text{butlast } ebs @ bs)$ 
  by  $\text{-- (rule old.dominatesE[OF dom[simplified contr]])}$ 
show False
proof (cases a \in set (butlast ebs))
  case True
    hence  $a \in \text{set } ebs$  by (rule in-set-butlastD)
    with ebs obtain abs where  $abs: g \vdash a \text{--} abs \rightarrow b$   $a \notin \text{set } (tl \ abs)$ 
    by (rule old.path2-split-first-last, auto)
    let  $?path = (abs @ tl \ bs) @ tl \ ns$ 
    have  $\text{var } g \ v_b \neq \text{var } g \ v_a$ 
    proof (rule conventional)
      show  $g \vdash a \text{--} ?path \rightarrow m$  using abs(1) bs assms(2)
      by  $\text{-- (rule old.path2-app, rule old.path2-app)}$ 
      have  $a \notin \text{set } (tl \ bs)$  using 4 by (auto simp: in-set-tlD)
      moreover have  $a \notin \text{set } (tl \ ns)$  using 1 2 contr assms(4) by auto
      ultimately show  $a \notin \text{set } (tl \ ?path)$  using abs conv(2)
      by  $\text{-- (subst tl-append2, auto simp: old.path2-not-Nil)}$ 
      show  $v_a \in \text{allUses } g \ m$  using asm assms(3) by simp
      have  $b \in \text{set } (tl \ abs)$  using abs(1) conv(1)
      by (auto simp: old.path2-def intro!: last-in-tl nonsimple-length-gt-1)
      thus  $b \in \text{set } (tl \ ?path)$  using abs(1) by (simp add: old.path2-not-Nil)
    qed (simp-all add: a b)
    thus False using a b by simp
  next
    case False
    with 4 5 show False by simp
  qed
qed
hence  $\text{var } g \ v \neq \text{var } g \ v_a$  using a as 1 contr by  $\text{-- (rule allDefs-var-disjoint,$ 
auto)
  with a(2) show False by simp
qed
obtain eas where  $eas: g \vdash \text{Entry } g \text{--} eas \rightarrow a$ 
  using as by (atomize, auto)
hence  $g \vdash \text{Entry } g \text{--} eas @ tl \ as \rightarrow n$  using as by auto
hence 4: defNode  $g \ v \in \text{set } (eas @ tl \ as)$  by  $\text{-- (rule old.dominatesE[OF dom])}$ 

```



```

have defNode g v ∈ set (tl as)
proof (rule ccontr)
  assume asm: defNode g v ∉ set (tl as)
  with 4 have defNode g v ∈ set eas by simp
  then obtain eas' where eas': g ⊢ defNode g v - defNode g v # eas' → a
  defNode g v ∉ set eas' using eas
  by - (rule old.path2-split-first-last)
  let ?path = ((defNode g v # eas') @ tl as) @ tl ns
  have var g va ≠ var g v
  proof (rule conventional)
    show g ⊢ defNode g v - ?path → m using eas' as assms(2)
    by (auto simp del:append-Cons append-assoc intro: old.path2-app)
    show a ∈ set (tl ?path) using eas' 3 by (auto simp: old.path2-def)
    show defNode g v ∉ set (tl ?path) using assms(4) 1 eas'(2) asm by auto
  qed (simp-all add: 1 assms(3) a(1))
  with a(2) show False by simp
qed
moreover have defNode g v ≠ n using 1 2 by auto
ultimately have defNode g v ∈ set (butlast as) using as subsetD[OF set-tl,
of defNode g v as]
  by - (rule in-set-butlastI, auto simp: old.path2-def)
}
note def-in-as = this
from def-in-as[OF a conv(1,3) b conv(2)] def-in-as[OF b conv(2,4) a conv(1)]
conv(5,6) show False by auto
qed

lemma convergence-prop':
  assumes necessaryPhi g v n g ⊢ n - ns → m v ∈ var g ' allUses g m ∧ x. x ∈
  set ns ⇒ v ∉ oldDefs g x
  obtains val where var g val = v phis g (n, val) ≠ None
  using assms proof (induction length ns arbitrary: ns m rule: less-induct)
  case less
  from less.prem(4) obtain val where val: var g val = v val ∈ allUses g m by
  auto
  show ?thesis
  proof (cases ∃ m' ∈ set (tl ns). v ∈ var g ' phiDefs g m')
  case False
  with less.prem(5) have ∧ x. x ∈ set (tl ns) ⇒ val ∉ allDefs g x
  by (auto simp: allDefs-def val(1)[symmetric] oldDefs-def dest: in-set-tlD)
  moreover from less.prem(3,5) have val ∉ defs g n
  by (auto simp: oldDefs-def val(1)[symmetric] dest: old.path2-hd-in-ns)
  ultimately show ?thesis
  using less.prem
  by - (rule that[OF val(1)], rule convergence-prop, auto simp: val)
next
  case True
  with less.prem(3) obtain ns' m' where m': g ⊢ n - ns' → m' v ∈ var g '

```

```

phiDefs g m' prefix ns' ns
  by - (erule old.path2-split-first-prop[where P= $\lambda m. v \in \text{var } g \text{ ' } \text{phiDefs } g$ 
m], auto dest: in-set-tlD)
  show ?thesis
  proof (cases m' = n)
    case True
      with m'(2) show ?thesis by (auto simp: phiDefs-def intro: that)
    next
      case False
        with m'(1) obtain m'' where m'':  $g \vdash n\text{-butlast } ns' \rightarrow m'' \text{ } m'' \in \text{set}$ 
(old.predecessors g m')
          by - (rule old.path2-unsnoc, auto)
        show ?thesis
        proof (rule less.hyps[of butlast ns', OF -])
          show length (butlast ns') < length ns
            using m''(1) m'(3) by (cases length ns', auto dest: prefix-length-le)

          from m'(2) obtain val vs where vs:  $\text{phis } g (m', \text{val}) = \text{Some } vs \text{ var } g \text{ val}$ 
= v
            by (auto simp: phiDefs-def)
          with m'' obtain val' where val'  $\in \text{phiUses } g m'' \text{ val}' \in \text{set } vs$ 
            by - (rule phiUses-exI, auto simp: phiDefs-def)
          with vs have val'  $\in \text{allUses } g m'' \text{ var } g \text{ val}' = v$  by auto
          then show  $v \in \text{var } g \text{ ' } \text{allUses } g m''$  by auto

          from m'(3) show  $\bigwedge x. x \in \text{set } (\text{butlast } ns') \implies v \notin \text{oldDefs } g x$ 
            by - (rule less.prem5, auto elim: in-set-butlastD)
          qed (auto intro: less.prem5(1,2) m''(1))
        qed
      qed
    qed
  qed

```

```

lemma nontrivialE:
  assumes  $\neg \text{trivial } g p \text{ phi } g p \neq \text{None}$  and[simp]:  $p \in \text{allVars } g$ 
  obtains r s where  $\text{phiArg } g p r \text{ phiArg } g p s \text{ distinct } [p, r, s]$ 
  proof -
    from assms(2) obtain vs where vs:  $\text{phi } g p = \text{Some } vs$  by auto
    have card (set vs - {p})  $\geq 2$ 
    proof -
      have card (set vs)  $\neq 0$  using Entry-no-phis[of g p] phi-wf[OF vs] vs by (auto
simp:phi-def invar)
      moreover have set vs  $\neq \{p\}$  using vs by - (rule phi-no-closed-loop, auto)
      ultimately have card (set vs - {p})  $\neq 0$ 
      by (metis List.finite-set card-0-eq insert-Diff-single insert-absorb remove.All-id
set-removeAll)
      moreover have card (set vs - {p})  $\neq 1$ 
      proof
        assume card (set vs - {p}) = 1
        then obtain q where  $q: \{q\} = \text{set } vs - \{p\}$  by - (erule card-eq-1-singleton,

```

```

auto)
  hence isTrivialPhi g p q using vs by (auto simp:isTrivialPhi-def split:option.split)
  moreover have phiArg g p q using q vs unfolding phiArg-def by auto
  ultimately show False using assms(1) by (auto simp:trivial-def)
qed
ultimately show ?thesis by arith
qed
then obtain r s where rs: r ≠ s r ∈ set vs - {p} s ∈ set vs - {p} by (rule
set-take-two)
thus ?thesis using vs by - (rule that[of r s], auto simp: phiArg-def)
qed

lemma paths-converge-prefix:
  assumes g ⊢ x-x's→z g ⊢ y-y's→z x ≠ y length xs > 1 length ys > 1 x ∉ set
  (butlast ys) y ∉ set (butlast xs)
  obtains xs' ys' z' where old.pathsConverge g x xs' y ys' z' prefix xs' xs prefix
  ys' ys
  using assms proof (induction length xs arbitrary:xs ys z rule:nat-less-induct)
  case 1
  from 1.prem(3,4) have 2: x ≠ y by (auto simp:old.path2-def)
  show thesis
  proof (cases set (butlast xs) ∩ set (butlast ys) = {})
  case True
  with 1.prem(2-) have old.pathsConverge g x xs y ys z by (auto simp add:
  old.pathsConverge'-def)
  thus thesis by (rule 1.prem(1), simp-all)
  next
  case False
  then obtain xs' z' where xs': g ⊢ x-x's'→z' prefix xs' (butlast xs) z' ∈ set
  (butlast ys) ∀ a ∈ set (butlast xs'). a ∉ set (butlast ys)
  using 1.prem(2,5) by - (rule old.path2-split-first-prop[of g x butlast xs -
  λa. a ∈ set (butlast ys)], auto elim: old.path2-unsnoc)
  from xs'(3) 1.prem(3) obtain ys' where ys': g ⊢ y-y's'→z' strict-prefix
  ys' ys
  by - (rule old.path2-strict-prefix-ex)
  show ?thesis
  proof (rule 1.hyps[rule-format, OF - - xs'(1) ys'(1) assms(3)])
  show length xs' < length xs using xs'(2) xs'(1)
  by - (rule prefix-length-less, rule strict-prefix-butlast, auto)
  from 1.prem(1) prefix-order.dual-order.strict-implies-order prefix-order.dual-order.trans
  prefix-butlastD xs'(2) ys'(2)
  show ∧xs'' ys'' z''. old.pathsConverge g x xs'' y ys'' z'' ⇒ prefix xs'' xs'
  ⇒ prefix ys'' ys' ⇒ thesis
  by blast
  show length xs' > 1
  proof-
  have length xs' ≠ 0 using xs' by auto
  moreover {
  assume length xs' = 1

```

```

    with  $xs'(1,3)$  have  $x \in \text{set } (\text{butlast } ys)$ 
  by (auto simp: old.path2-def simp del: One-nat-def dest!: singleton-list-hd-last)
    with 1.premis(7) have False ..
  }
  ultimately show ?thesis by arith
qed

show length  $ys' > 1$ 
proof-
  have length  $ys' \neq 0$  using  $ys'$  by auto
  moreover {
    assume length  $ys' = 1$ 
    with  $ys'(1)$   $xs'(1,2)$  have  $y \in \text{set } (\text{butlast } xs)$ 
      by (auto simp: old.path2-def old.path-not-Nil simp del: One-nat-def
dest!: singleton-list-hd-last)
    with 1.premis(8) have False ..
  }
  ultimately show ?thesis by arith
qed

show  $y \notin \text{set } (\text{butlast } xs')$ 
  using  $xs'(2)$  1.premis(8)
  by (metis in-prefix in-set-butlastD)
show  $x \notin \text{set } (\text{butlast } ys')$ 
  by (metis 1.premis(7) in-set-butlast-appendI strict-prefixE'  $ys'(2)$ )
qed simp
qed
qed

lemma unnecessaryPhi-disjoint-paths-aux:
  assumes  $\neg \text{unnecessaryPhi } g r$  and [simp]:  $r \in \text{allVars } g$ 
  obtains  $n_1 ns_1 n_2 ns_2$  where
    var  $g r \in \text{oldDefs } g n_1 g \vdash n_1 - ns_1 \rightarrow \text{defNode } g r$  and
    var  $g r \in \text{oldDefs } g n_2 g \vdash n_2 - ns_2 \rightarrow \text{defNode } g r$  and
    set  $(\text{butlast } ns_1) \cap \text{set } (\text{butlast } ns_2) = \{\}$ 
  proof (cases phi g r)
  case None
    thus thesis by - (rule that[of defNode g r - defNode g r], auto intro!: oldDefsI
intro: defNode-cases[of r g])
  next
  case Some
    with asms that show ?thesis by (auto simp: unnecessaryPhi-def necessaryPhi-def
old.pathsConverge'-def)
  qed

lemma unnecessaryPhi-disjoint-paths:
  assumes  $\neg \text{unnecessaryPhi } g r \neg \text{unnecessaryPhi } g s$ 

  and  $rs: \text{defNode } g r \neq \text{defNode } g s$ 

```

**and**[simp]:  $r \in \text{allVars } g \ s \in \text{allVars } g \ \text{var } g \ r = V \ \text{var } g \ s = V$   
**obtains**  $n \ ns \ m \ ms$  **where**  $V \in \text{oldDefs } g \ n \ g \vdash n - ns \rightarrow \text{defNode } g \ r$  **and**  $V \in \text{oldDefs } g \ m \ g \vdash m - ms \rightarrow \text{defNode } g \ s$   
**and**  $\text{set } ns \cap \text{set } ms = \{\}$   
**proof**–  
**obtain**  $n_1 \ ns_1 \ n_2 \ ns_2$  **where**  
 $ns_1: V \in \text{oldDefs } g \ n_1 \ g \vdash n_1 - ns_1 \rightarrow \text{defNode } g \ r \ \text{defNode } g \ r \notin \text{set } (\text{butlast } ns_1)$  **and**  
 $ns_2: V \in \text{oldDefs } g \ n_2 \ g \vdash n_2 - ns_2 \rightarrow \text{defNode } g \ r \ \text{defNode } g \ r \notin \text{set } (\text{butlast } ns_2)$  **and**  
 $ns: \text{set } (\text{butlast } ns_1) \cap \text{set } (\text{butlast } ns_2) = \{\}$   
**proof**–  
**from** *assms* **obtain**  $n_1 \ ns_1 \ n_2 \ ns_2$  **where**  
 $ns_1: V \in \text{oldDefs } g \ n_1 \ g \vdash n_1 - ns_1 \rightarrow \text{defNode } g \ r$  **and**  
 $ns_2: V \in \text{oldDefs } g \ n_2 \ g \vdash n_2 - ns_2 \rightarrow \text{defNode } g \ r$  **and**  
 $ns: \text{set } (\text{butlast } ns_1) \cap \text{set } (\text{butlast } ns_2) = \{\}$   
**by** – (*rule ununnecessaryPhis-disjoint-paths-aux, auto*)  
  
**from**  $ns_1$  **obtain**  $ns_1'$  **where**  $ns_1': g \vdash n_1 - ns_1' \rightarrow \text{defNode } g \ r \ \text{defNode } g \ r \notin \text{set } (\text{butlast } ns_1')$  *distinct*  $ns_1'$   $\text{set } ns_1' \subseteq \text{set } ns_1$   
**by** (*auto elim: old.simple-path2*)  
**from**  $ns_2$  **obtain**  $ns_2'$  **where**  $ns_2': g \vdash n_2 - ns_2' \rightarrow \text{defNode } g \ r \ \text{defNode } g \ r \notin \text{set } (\text{butlast } ns_2')$  *distinct*  $ns_2'$   $\text{set } ns_2' \subseteq \text{set } ns_2$   
**by** (*auto elim: old.simple-path2*)  
  
**have**  $\text{set } (\text{butlast } ns_1') \cap \text{set } (\text{butlast } ns_2') = \{\}$   
**proof** (*rule equalsOI*)  
**fix**  $x$   
**assume**  $1: x \in \text{set } (\text{butlast } ns_1') \cap \text{set } (\text{butlast } ns_2')$   
**with** *set-butlast-distinct*[*OF*  $ns_1'(3)$ ]  $ns_1'(1)$  **have**  $2: x \neq \text{defNode } g \ r$  **by**  
(*auto simp: old.path2-def*)  
**with**  $1 \ ns_1'(4) \ ns_2'(4) \ ns_1(2) \ ns_2(2)$  **have**  $x \in \text{set } (\text{butlast } ns_1) \ x \in \text{set } (\text{butlast } ns_2)$   
**by** – (*auto intro!: in-set-butlastI elim: in-set-butlastD simp: old.path2-def*)  
**with**  $ns$  **show** *False* **by** *auto*  
**qed**  
  
**thus** *thesis* **by** (*rule that*[*OF*  $ns_1(1) \ ns_1'(1,2) \ ns_2(1) \ ns_2'(1,2)$ ])  
**qed**  
  
**obtain**  $m \ ms$  **where**  $ms: V \in \text{oldDefs } g \ m \ g \vdash m - ms \rightarrow \text{defNode } g \ s \ \text{defNode } g \ r \notin \text{set } ms$   
**proof**–  
**from** *assms*(2) **obtain**  $m_1 \ ms_1 \ m_2 \ ms_2$  **where**  
 $ms_1: V \in \text{oldDefs } g \ m_1 \ g \vdash m_1 - ms_1 \rightarrow \text{defNode } g \ s$  **and**  
 $ms_2: V \in \text{oldDefs } g \ m_2 \ g \vdash m_2 - ms_2 \rightarrow \text{defNode } g \ s$  **and**  
 $ms: \text{set } (\text{butlast } ms_1) \cap \text{set } (\text{butlast } ms_2) = \{\}$   
**by** – (*rule ununnecessaryPhis-disjoint-paths-aux, auto*)  
**show** *thesis*

```

proof (cases defNode g r ∈ set ms1)
  case False
  with ms1 show thesis by (rule that)
next
  case True
  have defNode g r ∉ set ms2
  proof
    assume defNode g r ∈ set ms2
    moreover note (defNode g r ≠ defNode g s)
    ultimately have defNode g r ∈ set (butlast ms1) defNode g r ∈ set (butlast
ms2) using True ms1(2) ms2(2)
      by (auto simp:old.path2-def intro:in-set-butlastI)
    with ms show False by auto
  qed
  with ms2 show thesis by (rule that)
qed
qed

show ?thesis
proof (cases (set ns1 ∪ set ns2) ∩ set ms = {})
  case True
  with ns1 ms show ?thesis by – (rule that, auto)
next
  case False
  then obtain m' ms' where ms': g ⊢ m'–ms'→defNode g s m' ∈ set ns1 ∪
set ns2 set (tl ms') ∩ (set ns1 ∪ set ns2) = {} suffix ms' ms
    by – (rule old.path2-split-last-prop[OF ms(2), of λx. x ∈ set ns1 ∪ set ns2],
auto)
  from this(4) ms(3) have 2: defNode g r ∉ set ms'
    by (auto dest: set-mono-suffix)
  {
    fix n1 ns1 n2 ns2
    assume 4: m' ∈ set ns1
    assume ns1: V ∈ oldDefs g n1 g ⊢ n1–ns1→defNode g r defNode g r ∉ set
(butlast ns1)
    assume ns2: V ∈ oldDefs g n2 g ⊢ n2–ns2→defNode g r defNode g r ∉ set
(butlast ns2)
    assume ns: set (butlast ns1) ∩ set (butlast ns2) = {}
    assume ms': g ⊢ m'–ms'→defNode g s set (tl ms') ∩ (set ns1 ∪ set ns2)
= {}
    have m' ∈ set (butlast ns1)
    proof–
      from ms'(1) have m' ∈ set ms' by auto
      with 2 have defNode g r ≠ m' by auto
    with 4 ns1(2) show ?thesis by – (rule in-set-butlastI, auto simp:old.path2-def)
    qed
    with ns1(2) obtain ns1' where ns1': g ⊢ n1–ns1'→m' m' ∉ set (butlast
ns1') strict-prefix ns1' ns1
      by – (rule old.path2-strict-prefix-ex)
  }

```

```

have thesis
proof (rule that[OF  $ns_2(1,2)$ ,  $ns_1(1)$ , of  $ns_1'@tl\ ms'$ ])
  show  $g \vdash n_1 - ns_1' @ tl\ ms' \rightarrow defNode\ g\ s$  using  $ns_1'(1)\ ms'(1)$  by auto
  show  $set\ ns_2 \cap set\ (ns_1' @ tl\ ms') = \{\}$ 
  proof (rule equalsOI)
    fix  $x$ 
    assume  $x \in set\ ns_2 \cap set\ (ns_1' @ tl\ ms')$ 
    show False
    proof (cases  $x \in set\ ns_1'$ )
      case True
      hence  $\_4: x \in set\ (butlast\ ns_1)$  using  $ns_1'(\_3)$  by (auto dest:set-mono-strict-prefix)
      with  $ns_1(\_3)$  have  $x \neq defNode\ g\ r$  by auto
      with  $ns_2(\_2)\ x$  have  $x \in set\ (butlast\ ns_2)$ 
      by - (rule in-set-butlastI, auto simp:old.path2-def)
      with  $\_4\ ns$  show False by auto
      next
      case False
      with  $x$  have  $x \in set\ (tl\ ms')$  by simp
      with  $x\ ms'(\_2)$  show False by auto
    qed
  qed
}
note  $\_4 = this$ 
show ?thesis
proof (cases  $m' \in set\ ns_1$ )
  case True
  thus ?thesis using  $ns_1\ ns_2\ ns\ ms'(1,\_3)$  by (rule  $\_4$ )
next
  case False
  with  $ms'(\_2)$  have  $m' \in set\ ns_2$  by simp
  thus ?thesis using  $ns\ ms'(1,\_3)$  by - (rule  $\_4[OF - ns_2\ ns_1]$ , auto)
qed
qed

```

Lemma 3. If a  $\phi$  function  $p$  in a block  $P$  for a variable  $v$  is unnecessary, but non-trivial, then it has an operand  $q$  in a block  $Q$ , such that  $q$  is an unnecessary  $\phi$  function and  $Q$  does not dominate  $P$ .

**lemma 3:**

```

assumes unnecessaryPhi  $g\ p$   $\neg trivial\ g\ p$  and[simp]:  $p \in allVars\ g$ 
obtains  $q$  where phiArg  $g\ p\ q$  unnecessaryPhi  $g\ q$   $\neg def-dominates\ g\ q\ p$ 
proof -
note unnecessaryPhi-def[simp]
let  $?P = defNode\ g\ p$ 

```

The node  $p$  must have at least two different operands  $r$  and  $s$ , which are not  $p$  itself. Otherwise,  $p$  is trivial.

```

from assms obtain  $r\ s$  where  $rs: phiArg\ g\ p\ r\ phiArg\ g\ p\ s\ distinct\ [p,\ r,\ s]$ 

```

by  $-\text{(rule nontrivialE, auto)}$   
**hence**[simp]:  $\text{var } g \ r = \text{var } g \ p \ \text{var } g \ s = \text{var } g \ p \ r \in \text{allVars } g \ s \in \text{allVars } g$   
 by  $(\text{simp-all add:phiArg-same-var})$

They can either be:

- The result of a direct assignment to v.
- The result of a necessary  $\phi$  function r' . This however means that r' was reachable by at least two different direct assignments to v. So there is a path from a direct assignment of v to p.
- Another unnecessary  $\phi$  function.

**let** ?R = defNode g r  
**let** ?S = defNode g s

**have**[simp]: ?R  $\neq$  ?S **using** rs **by**  $-\text{(rule phiArgs-def-distinct, auto)}$

**have** one-unnec: unnecessaryPhi g r  $\vee$  unnecessaryPhi g s  
**proof** (rule ccontr, simp only: de-Morgan-disj not-not)

Assume neither r in a block R nor s in a block S is an unnecessary  $\phi$  function.

**assume** asm:  $\neg \text{unnecessaryPhi } g \ r \wedge \neg \text{unnecessaryPhi } g \ s$

Then a path from an assignment to v in a block n crosses R and a path from an assignment to v in a block m crosses S.

AMENDMENT: ...so that the paths are disjoint!

**obtain** n ns m ms **where** ns:  $\text{var } g \ p \in \text{oldDefs } g \ n \ g \vdash n - ns \rightarrow ?R \ n \notin \text{set } (tl \ ns)$

**and** ms:  $\text{var } g \ p \in \text{oldDefs } g \ m \ g \vdash m - ms \rightarrow \text{defNode } g \ s \ m \notin \text{set } (tl \ ms)$

**and** ns-ms:  $\text{set } ns \cap \text{set } ms = \{\}$

**proof**–

**obtain** n ns m ms **where** ns:  $\text{var } g \ p \in \text{oldDefs } g \ n \ g \vdash n - ns \rightarrow ?R$  **and** ms:  $\text{var } g \ p \in \text{oldDefs } g \ m \ g \vdash m - ms \rightarrow ?S$

**and** ns-ms:  $\text{set } ns \cap \text{set } ms = \{\}$

**using** asm[THEN conjunct1] asm[THEN conjunct2] **by** (rule ununnecessaryPhis-disjoint-paths, auto)

**moreover from** ns **obtain** ns' **where**  $g \vdash n - ns' \rightarrow ?R \ n \notin \text{set } (tl \ ns') \ \text{set } ns' \subseteq \text{set } ns$

**by** (auto intro: old.simple-path2)

**moreover from** ms **obtain** ms' **where**  $g \vdash m - ms' \rightarrow ?S \ m \notin \text{set } (tl \ ms') \ \text{set } ms' \subseteq \text{set } ms$

**by** (auto intro: old.simple-path2)

**ultimately show** thesis **by**  $-\text{(rule that[of n ns' m ms'], auto)}$

**qed**

**from** ns(1) ms(1) **obtain** v v' **where** v:  $v \in \text{defs } g \ n$  **and** v':  $v' \in \text{defs } g \ m$   
**and**[simp]:  $\text{var } g \ v = \text{var } g \ p \ \text{var } g \ v' = \text{var } g \ p$   
**by** (auto simp:oldDefs-def)



They converge at P or earlier.

```

obtain ns' n' where ns': g ⊢ ?R-ns'→n' r ∈ phiUses g n' n' ∈ set
(old.predecessors g ?P) ?R ∉ set (tl ns')
by (rule phiArg-path-ex'[OF rs(1)], auto elim: old.simple-path2)
obtain ms' m' where ms': g ⊢ ?S-ms'→m' s ∈ phiUses g m' m' ∈ set
(old.predecessors g ?P) ?S ∉ set (tl ms')
by (rule phiArg-path-ex'[OF rs(2)], auto elim: old.simple-path2)

```

```

let ?left = (ns@tl ns')@[?P]
let ?right = (ms@tl ms')@[?P]

```

```

obtain ns'' ms'' z where z: old.pathsConverge g n ns'' m ms'' z prefix ns''
?left prefix ms'' ?right

```

```

proof (rule paths-converge-prefix)
show n ≠ m using ns ms ns-ms by auto

```

```

show g ⊢ n-?left→?P using ns ns'
by - (rule old.path2-snoc, rule old.path2-app)
show length ?left > 1 using ns by auto
show g ⊢ m-?right→?P using ms ms'
by - (rule old.path2-snoc, rule old.path2-app)
show length ?right > 1 using ms by auto

```

```

have n ∉ set ms using ns-ms ns by auto
moreover have n ∉ set (tl ms') using v rs(2) ms'(2) asm
by - (rule conventional'[OF ms'(1,4), of s v], auto)
ultimately show n ∉ set (butlast ?right)
by (auto simp del:append-assoc)

```

```

have m ∉ set ns using ns-ms ms by auto
moreover have m ∉ set (tl ns') using v' rs(1) ns'(2) asm
by - (rule conventional'[OF ns'(1,4), of r v'], auto)
ultimately show m ∉ set (butlast ?left)
by (auto simp del:append-assoc)

```

qed

```

from this(1) ns(1) ms(1) have necessary: necessaryPhi g (var g p) z by
(rule necessaryPhiI)

```

```

show False
proof (cases z = ?P)

```

Convergence at P is not possible because p is unnecessary.

```

case True
thus False using assms(1) necessary by simp
next

```

An earlier convergence would imply a necessary  $\phi$  function at this point, which violates the SSA property.

```

case False

```

```

from  $z(1)$  have  $z \in \text{set } ns'' \cap \text{set } ms''$  by (auto simp: old.pathsConverge'-def)
with False have  $z \in \text{set } (ns@tl \ ns') \cap \text{set } (ms@tl \ ms')$ 
using  $z(2,3)$ [THEN set-mono-prefix] by (auto elim:set-mono-prefix)
hence  $z\text{-on}: z \in \text{set } (tl \ ns') \cup \text{set } (tl \ ms')$  using  $ns\text{-}ms$  by auto

{
  fix  $r \ ns' \ n'$ 
  let  $?R = \text{defNode } g \ r$ 
  assume  $ns': g \vdash ?R\text{-}ns' \rightarrow n' \ r \in \text{phiUses } g \ n' \ n' \in \text{set } (\text{old.predecessors}$ 
 $g \ (?P)) \ ?R \notin \text{set } (tl \ ns')$ 
  assume  $rs: \text{var } g \ r = \text{var } g \ p$ 
  have  $z \notin \text{set } (tl \ ns')$ 
  proof
    assume  $asm: z \in \text{set } (tl \ ns')$ 
    obtain  $zs$  where  $zs: g \vdash z\text{-}zs \rightarrow n' \ \text{set } (tl \ zs) \subseteq \text{set } (tl \ ns')$  using  $asm$ 
      by - (rule old.path2-split-ex[OF ns'(1)], auto simp: old.path2-not-Nil
    elim: subsetD[OF set-tl])

    have  $\text{phis } g \ (z, r) \neq \text{None}$ 
    proof (rule convergence-prop[OF necessary[simplified rs[symmetric]]])
       $zs(1)$ 
      show  $r \in \text{allUses } g \ n'$  using  $ns'(2)$  by auto
      show  $r \notin \text{defs } g \ z$ 
      proof
        assume  $r \in \text{defs } g \ z$ 
        hence  $?R = z$  using  $zs$  by - (rule defNode-eq, auto)
        thus False using  $ns'(4)$   $asm$  by auto
      qed
    next
      fix  $x$ 
      assume  $x \in \text{set } (tl \ zs)$ 
      moreover have  $?R \notin \text{set } (tl \ zs)$  using  $ns'(4)$   $zs(2)$  by auto
      ultimately show  $r \notin \text{allDefs } g \ x$ 
      by (metis defNode-eq old.path2-in- $\alpha n$  set-tl subset-code(1)  $zs(1)$ )
    qed
    hence  $?R = z$  using  $zs(1)$  by - (rule defNode-eq, auto simp: allDefs-def
  phiDefs-def)
    thus False using  $ns'(4)$   $asm$  by auto
    qed
  }
note  $z\text{-not-on} = \text{this}$ 

  have  $z \notin \text{set } (tl \ ns')$  by (rule  $z\text{-not-on}$ [OF ns'], simp)
  moreover have  $z \notin \text{set } (tl \ ms')$  by (rule  $z\text{-not-on}$ [OF ms'], simp)
  ultimately show False using  $z\text{-on}$  by simp
qed
qed

```

So  $r$  or  $s$  must be an unnecessary  $\phi$  function. Without loss of generality, let this be  $r$ .

```

{
  fix r s
  assume r: unnecessaryPhi g r and[simp]: var g r = var g p
  assume[simp]: var g s = var g p
  assume rs: phiArg g p r phiArg g p s distinct [p, r, s]
  let ?R = defNode g r
  let ?S = defNode g s

  have[simp]: ?R ≠ ?S using rs by - (rule phiArgs-def-distinct, auto)
  have[simp]: s ∈ allVars g using rs by auto

  have thesis
  proof (cases old.dominates g ?R ?P)
    case False

```

If R does not dominate P, then r is the sought-after q.

```

  thus thesis using r rs(1) by - (rule that)
next
case True

```

So let R dominate P. Due to Lemma 2, S does not dominate P.

```

  hence 4: ¬old.dominates g ?S ?P using 2[OF rs] by simp

```

Employing the SSA property,  $r \neq p$  yields  $R \neq P$ .

```

  have ?R ≠ ?P
  proof (rule notI, rule allDefs-var-disjoint[of ?R g p r, simplified])
    show r ∈ allDefs g (defNode g r) using rs(1) by auto
    show p ≠ r using rs(3) by auto
  qed auto

```

Thus, R strictly dominates P.

```

  hence old.strict-dom g ?R ?P using True by simp

```

This implies that R dominates all predecessors of P, which contain the uses of p, especially the predecessor S' that contains the use of s.

```

  moreover obtain ss' S' where ss': g ⊢ ?S-ss'→S'
    and S': s ∈ phiUses g S' S' ∈ set (old.predecessors g ?P)
    by (rule phiArg-path-ex'[OF rs(2)], simp)
  ultimately have 5: old.dominates g ?R S' by - (rule old.dominates-unsnoc,
auto)

```

Due to the SSA property, there is a path from S to S' that does not contain R.

```

  from ss' obtain ss' where ss': g ⊢ ?S-ss'→S' ?S ∉ set (tl ss') by (rule
old.simple-path2)
  hence ?R ∉ set (tl ss') using rs(1,2) S'(1)
  by - (rule conventional'[where v=s and v'=r], auto simp del: phiArg-def)

```

Employing R dominates S' this yields R dominates S.

```

  hence dom: old.dominates g ?R ?S using 5 ss' by - (rule old.dominates-extend)

```

Now assume that s is necessary.

**have** *unnecessaryPhi* *g s*  
**proof** (*rule ccontr*)  
**assume** *s*:  $\neg$ *unnecessaryPhi* *g s*

Let X contain the most recent definition of v on a path from the start block to R.

**from** *rs(1)* **obtain** *X xs* **where** *xs*:  $g \vdash X - xs \rightarrow ?R$  *var g r*  $\in$  *oldDefs g*  
*X old.EntryPath g xs*  
**by**  $-$  (*rule allDef-path-from-simpleDef*[*of r g*], *auto simp del: phiArg-def*)  
**then obtain** *X xs* **where** *xs*:  $g \vdash X - xs \rightarrow ?R$  *var g r*  $\in$  *oldDefs g*  $X \forall x$   
 $\in$  *set (tl xs)*. *var g r*  $\notin$  *oldDefs g* *x old.EntryPath g xs*  
**by**  $-$  (*rule old.path2-split-last-prop*[*OF xs(1)*], *of*  $\lambda x$ . *var g r*  $\in$  *oldDefs*  
*g x*], *auto dest: old.EntryPath-suffix*)  
**then obtain** *x* **where** *x*:  $x \in$  *defs g X* *var g x = var g r* **by** (*auto simp:*  
*oldDefs-def old.path2-def*)  
**hence**[*simp*]:  $X = \text{defNode } g \ x$  **using** *xs* **by**  $-$  (*rule defNode-eq*[*symmetric*],  
*auto*)  
**from** *xs* **obtain** *xs* **where** *xs*:  $g \vdash X - xs \rightarrow ?R$   $X \notin$  *set (tl xs)* *old.EntryPath*  
*g xs*  
**by**  $-$  (*rule old.simple-path2*, *auto dest: old.EntryPath-suffix*)

By Definition 2 there are two definitions of v that render s necessary. Since R dominates S, the SSA property yields that one of these definitions is contained in a block Y on a path  $R \rightarrow^+ S$ .

**obtain** *Y ys ys'* **where** *Y*: *var g s*  $\in$  *oldDefs g Y*  
**and** *ys*:  $g \vdash Y - ys \rightarrow ?S$   $?R \notin$  *set ys*  
**and** *ys'*:  $g \vdash ?R - ys' \rightarrow Y$   $?R \notin$  *set (tl ys')*  
**proof** (*cases phi g s*)  
**case** *None*  
**hence**  $s \in$  *defs g ?S* **by**  $-$  (*rule defNode-cases*[*of s g*], *auto*)  
**moreover obtain** *ns* **where**  $g \vdash ?R - ns \rightarrow ?S$   $?R \notin$  *set (tl ns)* **using**  
*dom*  
**by**  $-$  (*rule old.dominates-path*, *auto intro: old.simple-path2*)  
**ultimately show thesis** **by**  $-$  (*rule that*[*where ys1=[?S]*], *auto dest:*  
*oldDefsI*)  
**next**  
**case** *Some*  
**with** *s* **obtain** *Y<sub>1</sub> ys<sub>1</sub> Y<sub>2</sub> ys<sub>2</sub>* **where** *var g s*  $\in$  *oldDefs g*  $Y_1$   $g \vdash$   
 $Y_1 - ys_1 \rightarrow ?S$   
**and** *var g s*  $\in$  *oldDefs g*  $Y_2$   $g \vdash Y_2 - ys_2 \rightarrow ?S$   
**and** *ys*: *set (butlast ys<sub>1</sub>)*  $\cap$  *set (butlast ys<sub>2</sub>)* = {}  $Y_1 \neq Y_2$   
**by** (*auto simp:necessaryPhi-def old.pathsConverge'-def*)  
**moreover from** *ys(1)* **have**  $?R \notin$  *set (butlast ys<sub>1</sub>)*  $\vee$   $?R \notin$  *set (butlast*  
*ys<sub>2</sub>)* **by** *auto*  
**ultimately obtain** *Y ys* **where** *ys*: *var g s*  $\in$  *oldDefs g*  $Y$   $g \vdash Y - ys \rightarrow ?S$   
 $?R \notin$  *set (butlast ys)* **by** *auto*  
**obtain** *es* **where** *es*:  $g \vdash \text{Entry } g - es \rightarrow Y$  **using** *ys(2)*  
**by**  $-$  (*atomize-elim*, *rule old.Entry-reaches*, *auto*)  
**have**  $?R \in$  *set (butlast es@ys)* **using** *old.path2-app'*[*OF es ys(2)*] **by**  $-$   
(*rule old.dominatesE*[*OF dom*])

**moreover have**  $?R \neq \text{last } ys$  **using** *old.path2-last*[*OF ys*(2), *symmetric*]  
**by** *simp*  
**hence**  $1: ?R \notin \text{set } ys$  **using** *ys*(3) **by** (*auto dest: in-set-butlastI*)  
**ultimately have**  $?R \in \text{set } (\text{butlast } es)$  **by** *auto*  
**then obtain**  $ys'$  **where**  $g \vdash ?R - ys' \rightarrow Y$   $?R \notin \text{set } (\text{tl } ys')$  **using** *es*  
**by**  $-$  (*rule old.path2-split-ex, assumption, rule in-set-butlastD, auto*  
*intro: old.simple-path2*)  
**thus thesis using** *ys*(1,2) 1 **by**  $-$  (*rule that*[*of Y ys ys'*], *auto*)  
**qed**

**from**  $Y$  **obtain**  $y$  **where**  $y: y \in \text{defs } g \ Y \ \text{var } g \ y = \text{var } g \ s$  **by** (*auto*  
*simp: oldDefs-def*)  
**hence**[*simp*]:  $Y = \text{defNode } g \ y$  **using** *ys* **by**  $-$  (*rule defNode-eq*[*symmetric*],  
*auto*)

**obtain**  $rr' \ R'$  **where**  $g \vdash ?R - rr' \rightarrow R'$   
**and**  $R': r \in \text{phiUses } g \ R' \ R' \in \text{set } (\text{old.predecessors } g \ ?P)$   
**by** (*rule phiArg-path-ex'*[*OF rs*(1)], *simp*)  
**then obtain**  $rr'$  **where**  $rr': g \vdash ?R - rr' \rightarrow R' \ ?R \notin \text{set } (\text{tl } rr')$  **by**  $-$  (*rule*  
*old.simple-path2*)  
**with**  $R'$  **obtain**  $rr$  **where**  $rr: g \vdash ?R - rr \rightarrow ?P$  **and**[*simp*]:  $rr = rr' @$   
 $[?P]$  **by** (*auto intro: old.path2-snoc*)  
**from**  $ss' \ S'$  **obtain**  $ss$  **where**  $ss: g \vdash ?S - ss \rightarrow ?P$  **and**[*simp*]:  $ss = ss' @$   
 $[?P]$  **by** (*auto intro: old.path2-snoc*)

Thus, there are paths  $X \rightarrow^+ P$  and  $Y \rightarrow^+ P$  rendering p necessary. Since this is a contradiction, s is unnecessary and the sought-after q.

**have** *old.pathsConverge*  $g \ X \ (\text{butlast } xs @ rr) \ Y \ (ys @ \text{tl } ss) \ ?P$   
**proof** (*rule old.pathsConvergeI*)  
**show**  $g \vdash X - \text{butlast } xs @ rr \rightarrow ?P$  **using** *xs rr* **by** *auto*  
**show**  $g \vdash Y - ys @ \text{tl } ss \rightarrow ?P$  **using** *ys ss* **by** *auto*

{  
**assume**  $X = ?P$   
**moreover have**  $p \in \text{phiDefs } g \ ?P$  **using** *assms*(1) **by** (*auto*  
*simp: phiDefs-def phi-def*)  
**ultimately have** *False* **using** *simpleDefs-phiDefs-disjoint*[*of X g*]  
*allDefs-var-disjoint*[*of X g*]  $x$  **by** (*cases x = p, auto*)  
}  
**thus**  $\text{length } (\text{butlast } xs @ rr) > 1$  **using** *xs rr* **by**  $-$  (*rule old.path2-nontriv,*  
*auto*)

{  
**assume**  $Y = ?P$   
**moreover have**  $p \in \text{phiDefs } g \ ?P$  **using** *assms*(1) **by** (*auto*  
*simp: phiDefs-def phi-def*)  
**ultimately have** *False* **using** *simpleDefs-phiDefs-disjoint*[*of Y g*]  
*allDefs-var-disjoint*[*of Y g*]  $y$  **by** (*cases y = p, auto*)  
}  
**thus**  $\text{length } (ys @ \text{tl } ss) > 1$  **using** *ys ss* **by**  $-$  (*rule old.path2-nontriv,*

*auto*)  
**show**  $\text{set}(\text{butlast}(\text{butlast } xs \text{ @ } rr)) \cap \text{set}(\text{butlast}(ys \text{ @ } tl \text{ } ss)) = \{\}$   
**proof** (*rule equals0I*)  
**fix**  $z$   
**assume**  $z \in \text{set}(\text{butlast}(\text{butlast } xs @ rr)) \cap \text{set}(\text{butlast}(ys @ tl \text{ } ss))$   
**moreover** {  
**assume** *asm*:  $z \in \text{set}(\text{butlast } xs) \wedge z \in \text{set } ys$   
**have**  $\text{old.shortestPath } g \ z < \text{old.shortestPath } g \ ?R$  **using** *asm*(1)  
*xs*(3)  
**by** – (*subst old.path2-last[OF xs(1)]*, *rule old.EntryPath-butlast-less-last*)  
**moreover**  
**from** *ys asm*(2) **obtain**  $ys'$  **where**  $g \vdash z - ys' \rightarrow ?S \text{ suffix } ys' \ ys$   
**by** – (*rule old.path2-split-ex*, *auto simp: Sublist.suffix-def*)  
**have**  $\text{old.dominates } g \ ?R \ z$  **using** *ys*(2) *set-tl[of ys] suffix-tl-subset[OF*  
*ys'(2)]*  
**by** – (*rule old.dominates-extend[OF dom ys'(1)]*, *auto*)  
**hence**  $\text{old.shortestPath } g \ ?R \leq \text{old.shortestPath } g \ z$   
**by** (*rule old.dominates-shortestPath-order*, *auto*)  
**ultimately have** *False* **by** *simp*  
**}**  
**moreover** {  
**assume** *asm*:  $z \in \text{set}(\text{butlast } xs) \wedge z \in \text{set}(tl \text{ } ss')$   
**have**  $\text{old.shortestPath } g \ z < \text{old.shortestPath } g \ ?R$  **using** *asm*(1)  
*xs*(3)  
**by** – (*subst old.path2-last[OF xs(1)]*, *rule old.EntryPath-butlast-less-last*)  
**moreover**  
**from** *asm*(2) **obtain**  $ss_2$  **where**  $g \vdash z - ss_2 \rightarrow S' \text{ set}(tl \text{ } ss_2) \subseteq$   
 $\text{set}(tl \text{ } ss')$   
**using** *set-tl[of ss']* **by** – (*rule old.path2-split-ex[OF ss'(1)]*, *auto*  
*simp: old.path2-not-Nil dest: in-set-butlastD*)  
**from** *S'(1) ss'(1)* **have**  $\text{old.dominates } g \ ?S \ S'$  **by** – (*rule*  
*allUses-dominated*, *auto*)  
**hence**  $\text{old.dominates } g \ ?S \ z$  **using** *ss'(2) ss\_2(2)*  
**by** – (*rule old.dominates-extend[OF - ss\_2(1)]*, *auto*)  
**with** *dom* **have**  $\text{old.dominates } g \ ?R \ z$  **by** *auto*  
**hence**  $\text{old.shortestPath } g \ ?R \leq \text{old.shortestPath } g \ z$   
**by** – (*rule old.dominates-shortestPath-order*, *auto*)  
**ultimately have** *False* **by** *simp*  
**}**  
**moreover**  
**have**  $?R \neq Y$  **using** *ys* **by** (*auto simp: old.path2-def*)  
**with** *ys'(1)* **have**  $1: \text{length } ys' > 1$  **by** (*rule old.path2-nontriv*)  
**{**  
**assume** *asm*:  $z \in \text{set } rr' \wedge z \in \text{set } ys$   
**then obtain**  $ys_1$  **where**  $g \vdash Y - ys_1 \rightarrow z \text{ prefix } ys_1 \ ys$   
**by** – (*rule old.path2-split-ex[OF ys(1)]*, *auto*)  
**from** *asm* **obtain**  $rr_2$  **where**  $g \vdash z - rr_2 \rightarrow R' \text{ set}(tl \text{ } rr_2) \subseteq \text{set}$   
*(tl rr')*  
**by** – (*rule old.path2-split-ex[OF rr'(1)]*, *auto simp: old.path2-not-Nil*)

```

let ?path = ys'@tl (ys1@tl rr2)
have var g y ≠ var g r
proof (rule conventional)
show g ⊢ ?R-?path→R' using ys' ys1 rr2 by (intro old.path2-app)
  show r ∈ allDefs g ?R using rs by auto
  show r ∈ allUses g R' using R' by auto

thus Y ∈ set (tl ?path) using ys'(1) 1
  by (auto simp:old.path2-def old.path2-not-Nil intro:last-in-tl)
show y ∈ allDefs g Y using y by simp
show defNode g r ∉ set (tl ?path)
  using ys' ys1(1) ys(2) rr2(2) rr'(2) prefix-tl-subset[OF ys1(2)]
set-tl[of ys] by (auto simp: old.path2-not-Nil)
qed
hence False using y by simp
}
moreover {
  assume asm: z ∈ set rr' z ∈ set (tl ss')
  then obtain ss'_1 where ss'_1: g ⊢ ?S-ss'_1→z prefix ss'_1 ss' using
ss'

  by - (rule old.path2-split-ex[OF ss'(1), of z], auto)
  from asm obtain rr'_2 where rr'_2: g ⊢ z-rr'_2→R' suffix rr'_2 rr'
  using rr' by - (rule old.path2-split-ex, auto simp: Sublist.suffix-def)
  let ?path = butlast ys'@(ys@tl (ss'_1@tl rr'_2))
  have var g s ≠ var g r
  proof (rule conventional)
    show g ⊢ ?R-?path→R' using ys' ys ss'_1 rr'_2(1) by (intro
old.path2-app old.path2-app')
    show r ∈ allDefs g ?R using rs by auto
    show r ∈ allUses g R' using R' by auto
    from 1 have[simp]: butlast ys' ≠ [] by (cases ys', auto)
    show ?S ∈ set (tl ?path) using ys(1) by auto
    show s ∈ allDefs g ?S using rs(2) by auto
    have ?R ∉ set (tl ss')
      using rs S'(1) by - (rule conventional''[OF ss'], auto)
    thus defNode g r ∉ set (tl ?path)
    using ys(1) ss'_1(1) suffix-tl-subset[OF rr'_2(2)] ys'(2) ys(2) rr'(2)
prefix-tl-subset[OF ss'_1(2)]
    by (auto simp: List.butlast-tl[symmetric] old.path2-not-Nil dest:
in-set-butlastD)
  qed
  hence False using y by simp
}
ultimately show False using rr'(1) ss'(1)
  by (auto simp del: append-assoc simp: append-assoc[symmetric]
old.path2-not-Nil dest: in-set-tlD)
qed
qed
hence necessaryPhi' g p using xs oldDefsI[OF x(1)] x(2) oldDefsI[OF

```

```

y(1)] y(2)
  by - (rule necessaryPhiI[where v=var g p], assumption, auto simp:old.path2-def)
  with assms(1) show False by auto
qed
thus ?thesis using rs(2) 4 by - (rule that)
qed
}
from one-unnec this[of r s] this[of s r] rs show thesis by auto
qed

```

Theorem 1. A program in SSA form with a reducible CFG G without any trivial  $\phi$  functions is in minimal SSA form.

**theorem** *reducible-nonredundant-imp-minimal*:

**assumes** *old.reducible g*  $\neg$ *redundant g*

**shows** *cytronMinimal g*

**unfolding** *cytronMinimal-def*

**proof** (*rule, rule*)

Proof. Assume G is not in minimal SSA form and contains no trivial  $\phi$  functions. We choose an unnecessary  $\phi$  function p.

**fix** p

**assume**[*simp*]: p  $\in$  *allVars g* **and** *phi*: *phi g p*  $\neq$  None

**show** *necessaryPhi' g p*

**proof** (*rule ccontr*)

**assume**  $\neg$ *necessaryPhi' g p*

**with** *phi* **have** *asm*: *unnecessaryPhi g p* **by** (*simp add: unnecessaryPhi-def*)

**let** ?A = {p. p  $\in$  *allVars g*  $\wedge$  *unnecessaryPhi g p*}

**let** ?r =  $\lambda p q. p \in ?A \wedge q \in ?A \wedge$  *phiArg g p q*  $\wedge$   $\neg$ *def-dominates g q p*

**let** ?r' = {(p,q). ?r p q}

**note** *phiArg-def[simp del]*

Due to Lemma 3, p has an operand q, which is unnecessary and does not dominate p. By induction q has an unnecessary  $\phi$  function as operand as well and so on. Since the program only has a finite number of operations, there must be a cycle when following the q chain.

**obtain** q **where** q: (q,q)  $\in$  ?r<sup>+</sup> q  $\in$  ?A

**proof** (*rule serial-on-finite-cycle*)

**show** *serial-on ?A ?r'*

**proof** (*rule serial-onI*)

**fix** x

**assume** x  $\in$  ?A

**then obtain** y **where** *unnecessaryPhi g y phiArg g x y*  $\neg$ *def-dominates g*

y x

**using** *assms(2)* **by** - (*rule 3, auto simp: redundant-def*)

**thus**  $\exists y \in ?A. (x,y) \in ?r'$  **using** (x  $\in$  ?A) **by** - (*rule bexI[where x=y]*,

*auto*)

**qed**

**show** ?A  $\neq$  {} **using** *asm* **by** (*auto intro!: exI*)

**qed** *auto*



A cycle in the  $\phi$  functions implies a cycle in G.

**then obtain**  $ns$  **where**  $ns: g \vdash \text{defNode } g \ q - ns \rightarrow \text{defNode } g \ q \ \text{length } ns > 1$   
 $\forall n \in \text{set } (\text{butlast } ns). \exists p \ q \ m \ ns'. ?r \ p \ q \wedge g \vdash \text{defNode } g \ q - ns' \rightarrow m$   
 $\wedge (\text{defNode } g \ q) \notin \text{set } (\text{tl } ns') \wedge q \in \text{phiUses } g \ m \wedge m \in \text{set } (\text{old.predecessors } g$   
 $(\text{defNode } g \ p)) \wedge n \in \text{set } ns' \wedge \text{set } ns' \subseteq \text{set } ns \wedge \text{defNode } g \ p \in \text{set } ns$   
**by**  $- (\text{rule } \text{phiArg-tranclp-path-ex}[\text{where } r=?r], \text{auto simp: tranclp-unfold})$

As G is reducible, the control flow cycle contains one entry block, which dominates all other blocks in the cycle.

**obtain**  $n$  **where**  $n: n \in \text{set } ns \ \forall m \in \text{set } ns. \text{old.dominates } g \ n \ m$   
**using**  $\text{assms}(1)[\text{unfolded } \text{old.reducible-def}, \text{rule-format}, \text{OF } ns(1)]$  **by**  $\text{auto}$

Without loss of generality, let  $q$  be in the entry block, which means it dominates  $p$ .

**have**  $n \in \text{set } (\text{butlast } ns)$   
**proof**  $(\text{cases } n = \text{last } ns)$   
**case**  $\text{False}$   
**with**  $n(1)$  **show**  $?thesis$  **by**  $(\text{rule } \text{in-set-butlastI})$   
**next**  
**case**  $\text{True}$   
**with**  $ns(1)$  **have**  $n = \text{hd } ns$  **by**  $(\text{auto simp: old.path2-def})$   
**with**  $ns(2)$  **show**  $?thesis$  **by**  $- (\text{auto intro: hd-in-butlast})$   
**qed**  
**then obtain**  $p \ q \ ns' \ m$  **where**  $ns': ?r \ p \ q \ g \vdash \text{defNode } g \ q - ns' \rightarrow m \ \text{defNode } g \ q \notin \text{set } (\text{tl } ns')$   
 $q \in \text{phiUses } g \ m \ m \in \text{set } (\text{old.predecessors } g \ (\text{defNode } g \ p)) \ n \in \text{set } ns'$   
 $\text{set } ns' \subseteq \text{set } ns \ \text{defNode } g \ p \in \text{set } ns$   
**by**  $- (\text{drule } ns(3)[\text{THEN } \text{bspec}], \text{auto})$   
**hence**  $\text{old.dominates } g \ (\text{defNode } g \ q) \ n$  **by**  $- (\text{rule } \text{defUse-path-dominated}, \text{auto})$   
**moreover from**  $ns' \ n(2)$  **have**  $n\text{-dom}: \text{old.dominates } g \ n \ (\text{defNode } g \ q)$   
 $\text{old.dominates } g \ n \ (\text{defNode } g \ p)$  **by**  $- (\text{auto elim!: bspec})$   
**ultimately have**  $\text{defNode } g \ q = n$  **by**  $\text{auto}$

Therefore, our assumption is wrong and G is either in minimal SSA form or there exist trivial  $\phi$  functions.

**with**  $ns'(1) \ n\text{-dom}(2)$  **show**  $\text{False}$  **by**  $\text{auto}$   
**qed**  
**qed**  
**end**

**context**  $\text{CFG-SSA-Transformed}$

**begin**

**definition**  $\text{phiCount } g = \text{card } ((\lambda(n,v). (n, \text{var } g \ v)) \text{ ` } \text{dom } (\text{phis } g))$

**lemma**  $\text{phiCount}: \text{phiCount } g = \text{card } (\text{dom } (\text{phis } g))$

**proof**  $-$

**have**  $1: v = v'$

**if**  $\text{asm}: \text{phis } g \ (n, v) \neq \text{None} \ \text{phis } g \ (n, v') \neq \text{None} \ \text{var } g \ v = \text{var } g \ v'$

**for**  $n \ v \ v'$

**proof** (*rule ccontr*)  
**from** *asm* **have**[*simp*]:  $v \in \text{allDefs } g \ n \ v' \in \text{allDefs } g \ n$  **by** (*auto simp: phiDefs-def allDefs-def*)  
**from** *asm* **have**[*simp*]:  $n \in \text{set } (\alpha n \ g)$  **by** - (*auto simp: phis-in- $\alpha n$* )  
**assume**  $v \neq v'$   
**with** *asm* **show** *False*  
**by** - (*rule allDefs-var-disjoint[of n g v v', THEN notE], auto*)  
**qed**

**show** *?thesis*  
**unfolding** *phiCount-def*  
**apply** (*rule card-image*)  
**apply** (*rule inj-onI*)  
**by** (*auto intro!: 1*)  
**qed**

**theorem** *phi-count-minimal*:  
**assumes** *cytronMinimal g pruned g*  
**assumes** *CFG-SSA-Transformed  $\alpha e \ \alpha n \ \text{invar } \text{inEdges}' \ \text{Entry } \text{oldDefs } \text{oldUses}$*   
*defs' uses' phis' var'*  
**shows**  $\text{card } (\text{dom } (\text{phis } g)) \leq \text{card } (\text{dom } (\text{phis}' g))$   
**proof** -  
**interpret** *other: CFG-SSA-Transformed  $\alpha e \ \alpha n \ \text{invar } \text{inEdges}' \ \text{Entry } \text{oldDefs}$*   
*oldUses defs' uses' phis' var'*  
**by** (*rule assms(3)*)  
**{**  
**fix**  $n \ v$   
**assume** *asm: phis g (n,v)  $\neq$  None*  
**from** *asm* **have**[*simp*]:  $v \in \text{phiDefs } g \ n \ v \in \text{allDefs } g \ n$  **by** (*auto simp: phiDefs-def allDefs-def*)  
**from** *asm* **have**[*simp*]:  $\text{defNode } g \ v = n \ n \in \text{set } (\alpha n \ g)$  **by** - (*auto simp: phis-in- $\alpha n$* )  
**from** *asm* **have** *liveVal g v*  
**by** - (*rule (pruned g)[unfolded pruned-def, THEN bspec, of n, rule-format]; simp*)  
**then obtain**  $ns \ m$  **where**  $ns: g \vdash n - ns \rightarrow m \ \text{var } g \ v \in \text{oldUses } g \ m \ \wedge x. x \in \text{set } (\text{tl } ns) \implies \text{var } g \ v \notin \text{oldDefs } g \ x$   
**by** (*rule liveVal-use-path, simp*)  
**have**  $\exists v'. \text{phis}' g \ (n, v') \neq \text{None} \wedge \text{var } g \ v = \text{var}' g \ v'$   
**proof** (*rule other.convergence-prop[OF - ns(1)]*)  
**from** *asm* **show** *necessaryPhi g (var g v) n*  
**by** - (*rule (cytronMinimal g)[unfolded cytronMinimal-def, THEN bspec, of v, simplified, rule-format], auto simp: cytronMinimal-def phi-def, auto intro: allDefs-in-allVars[where n=n]*)  
**with**  $ns(1,2)$  **show**  $\text{var } g \ v \in \text{var}' g \ \text{other.allUses } g \ m$   
**by** (*subst(asm) other.oldUses-def, auto simp: image-def allUses-def other.oldUses-def intro!: bexI*)  
**have**  $\text{var } g \ v \notin \text{oldDefs } g \ n$

```

    by (rule simpleDefs-phiDefs-var-disjoint, auto)
  then show  $\bigwedge x. x \in \text{set } ns \implies \text{var } g \ v \notin \text{oldDefs } g \ x$ 
    using ns(1) by (case-tac x = hd ns, auto dest: ns(3) not-hd-in-tl dest:
old.path2-hd)
  qed auto
}
note 1 = this

have phiCount g  $\leq$  other.phiCount g
unfolding phiCount-def other.phiCount-def
apply (rule card-mono)
apply (rule finite-imageI)
apply (rule other.phis-finite)
by (auto simp: dom-def image-def simp del: not-None-eq intro!: 1)

thus ?thesis by (simp add: phiCount other.phiCount)
qed
end

end

```

## 4 SSA Construction

### 4.1 CFG to SSA CFG

```

theory Construct-SSA imports SSA-CFG
  HOL-Library.While-Combinator
  HOL-Library.Product-Lexorder
begin

datatype Def = SimpleDef | PhiDef
type-synonym ('node, 'var) ssaVal = 'var  $\times$  'node  $\times$  Def

instantiation Def :: linorder
begin
  definition x < y  $\longleftrightarrow$  x = SimpleDef  $\wedge$  y = PhiDef
  definition less-eq-Def (x :: Def) y  $\longleftrightarrow$  x = y  $\vee$  x < y
  instance by intro-classes (metis Def.distinct(1) less-Def-def less-eq-Def-def Def.exhaust)+
end

locale CFG-Construct = CFG  $\alpha e \ \alpha n$  invar inEdges' Entry defs uses
for
   $\alpha e$  :: 'g  $\Rightarrow$  ('node::linorder  $\times$  'edgeD  $\times$  'node) set and
   $\alpha n$  :: 'g  $\Rightarrow$  'node list and
  invar :: 'g  $\Rightarrow$  bool and
  inEdges' :: 'g  $\Rightarrow$  'node  $\Rightarrow$  ('node  $\times$  'edgeD) list and
  Entry::'g  $\Rightarrow$  'node and
  defs :: 'g  $\Rightarrow$  'node  $\Rightarrow$  'var::linorder set and
  uses :: 'g  $\Rightarrow$  'node  $\Rightarrow$  'var set

```

**begin**

```
fun phiDefNodes-aux :: 'g ⇒ 'var ⇒ 'node list ⇒ 'node ⇒ 'node set where
  phiDefNodes-aux g v unvisited n =
    if n ∉ set unvisited ∨ v ∈ defs g n then {}
    else fold (∪)
      [phiDefNodes-aux g v (removeAll n unvisited) m . m ← predecessors g n]
      (if length (predecessors g n) ≠ 1 then {n} else {})
```

```
definition phiDefNodes :: 'g ⇒ 'var ⇒ 'node set where
  phiDefNodes g v ≡ fold (∪)
    [phiDefNodes-aux g v (αn g) n . n ← αn g, v ∈ uses g n]
    {}
```

```
definition var :: 'g ⇒ ('node, 'var) ssaVal ⇒ 'var where var g ≡ fst
```

```
abbreviation defNode :: ('node, 'var) ssaVal ⇒ 'node where defNode v ≡ fst
(snd v)
```

```
abbreviation defKind :: ('node, 'var) ssaVal ⇒ Def where defKind v ≡ snd
(snd v)
```

```
declare var-def [simp]
```

```
function lookupDef :: 'g ⇒ 'node ⇒ 'var ⇒ ('node, 'var) ssaVal where
  lookupDef g n v =
    if n ∉ set (αn g) then undefined
    else if v ∈ defs g n then (v,n,SimpleDef)
    else case predecessors g n of
      [m] ⇒ lookupDef g m v
      | - ⇒ (v,n,PhiDef)
```

**by** auto

```
termination by (relation measure (λ(g,n,-). shortestPath g n)) (auto intro:shortestPath-predecessor)
```

```
declare lookupDef.simps [code]
```

```
definition defs' :: 'g ⇒ 'node ⇒ ('node, 'var) ssaVal set where
```

```
  defs' g n ≡ (λv. (v,n,SimpleDef)) ` defs g n
```

```
definition uses' :: 'g ⇒ 'node ⇒ ('node, 'var) ssaVal set where
```

```
  uses' g n ≡ lookupDef g n ` uses g n
```

```
definition phis' :: 'g ⇒ ('node, ('node, 'var) ssaVal) phis where
```

```
  phis' ≡ λg (n,(v,m,def)).
```

```
    if m = n ∧ n ∈ phiDefNodes g v ∧ v ∈ vars g ∧ def = PhiDef then
```

```
      Some [lookupDef g m v . m ← predecessors g n]
```

```
    else None
```

```
declare uses'-def [code] defs'-def [code] phis'-def [code]
```

```
abbreviation lookupDefNode g n v ≡ defNode (lookupDef g n v)
```

```
declare lookupDef.simps [simp del]
```

```
declare phiDefNodes-aux.simps [simp del]
```

**lemma** *phiDefNodes-aux-cases*:  
**obtains** (*nonrec*) *phiDefNodes-aux g v unvisited n = {}* (*n*  $\notin$  *set unvisited*  $\vee$  *v*  $\in$  *defs g n*)  
| (*rec*) *phiDefNodes-aux g v unvisited n = fold union (map (phiDefNodes-aux g v (removeAll n unvisited)) (predecessors g n))*  
(*if length (predecessors g n) = 1 then {} else {n}*)  
*n*  $\in$  *set unvisited* *v*  $\notin$  *defs g n*  
**proof** (*cases n*  $\in$  *set unvisited*  $\wedge$  *v*  $\notin$  *defs g n*)  
**case** *True*  
**thus** *?thesis using rec by (simp add:phiDefNodes-aux.simps)*  
**next**  
**case** *False*  
**thus** *?thesis using nonrec by (simp add:phiDefNodes-aux.simps)*  
**qed**

**lemma** *phiDefNode-aux-is-join-node*:  
**assumes** *n*  $\in$  *phiDefNodes-aux g v un m*  
**shows** *length (predecessors g n)  $\neq$  1*  
**using** *assms proof (induction un arbitrary: m rule:removeAll-induct)*  
**case** (*1 un m*)  
**thus** *?case*  
**proof** (*cases un v g m rule:phiDefNodes-aux-cases*)  
**case** *rec*  
**with 1 show** *?thesis by (fastforce elim!:fold-union-elem split:if-split-asm)*  
**qed auto**  
**qed**

**lemma** *phiDefNode-is-join-node*:  
**assumes** *n*  $\in$  *phiDefNodes g v*  
**shows** *length (predecessors g n)  $\neq$  1*  
**using** *assms unfolding phiDefNodes-def*  
**by** (*auto elim!:fold-union-elem dest!:phiDefNode-aux-is-join-node*)

**abbreviation** *unvisitedPath* :: *'node list*  $\Rightarrow$  *'node list*  $\Rightarrow$  *bool* **where**  
*unvisitedPath un ns*  $\equiv$  *distinct ns*  $\wedge$  *set ns*  $\subseteq$  *set un*

**lemma** *unvisitedPath-removeLast*:  
**assumes** *unvisitedPath un ns length ns  $\geq$  2*  
**shows** *unvisitedPath (removeAll (last ns) un) (butlast ns)*  
**proof** –  
**let** *?n = last ns*  
**let** *?ns' = butlast ns*  
**let** *?un' = removeAll ?n un*  
**let** *?n' = last ?ns'*  
**from** *assms(2) have [simp]: ?n = ns ! (length ns - 1) by -(rule last-conv-nth, auto)*  
**from** *assms(1) have distinct ?ns' by -(rule distinct-butlast, simp)*  
**moreover**

```

have set ?ns'  $\subseteq$  set ?un'
proof
  fix n
  assume assm:  $n \in \text{set } ?ns'$ 
  then obtain i where  $n = ?ns' ! i$   $i < \text{length } ?ns'$  by (auto simp add:in-set-conv-nth)
  hence  $i: n = ns ! i$   $i < \text{length } ns - 1$  by (auto simp add:nth-butlast)
  with assms have  $1: n \neq ?n$  by (auto iff:nth-eq-iff-index-eq)
  from i assms(1) have  $n \in \text{set } un$  by auto
  with  $\langle n \in \text{set } ?ns' \rangle$  assms(1) 1 show  $n \in \text{set } ?un'$  by auto
qed
ultimately show ?thesis by simp
qed

lemma phiDefNodes-auxI:
  assumes  $g \vdash n - ns \rightarrow m$  unvisitedPath un ns  $\forall n \in \text{set } ns. v \notin \text{defs } g \ n$   $\text{length}$ 
  (predecessors g n)  $\neq 1$ 
  shows  $n \in \text{phiDefNodes-aux } g \ v \ un \ m$ 
  using assms(1,2,3) proof (induction un arbitrary: m ns rule:removeAll-induct)
  case (1 un)
  show ?case
  proof (cases un v g m rule:phiDefNodes-aux-cases)
  case nonrec
  from 1.prem(1) have  $m \in \text{set } ns$  unfolding path2-def by auto
  with nonrec show ?thesis using 1.prem(2,3) by auto
  next
  case rec
  show ?thesis
  proof (cases  $n = m$ )
  case True
  thus ?thesis using rec assms(4) by  $-(\text{subst } \text{rec}(1), \text{rule } \text{fold-union-elemI}[\text{of}$ 
   $-\{m\}], \text{auto})$ 
  next
  case False
  let ?ns' = butlast ns
  let ?m' = last ?ns'
  from 1.prem(1) have [simp]:  $m = \text{last } ns$  unfolding path2-def by simp
  with 1(2) False have ns':  $g \vdash n - ?ns' \rightarrow ?m'$   $?m' \in \text{set } (\text{predecessors } g \ m)$ 
  by (auto intro: path2-unsnoc)

  have  $n \in \text{phiDefNodes-aux } g \ v$  (removeAll m un) ?m'
  using rec(2) ns'
  apply-
  proof (rule 1.IH)
  from 1.prem(1) False have  $\text{length } ns \geq 2$  by (auto simp del:  $\langle m = \text{last}$ 
   $ns \rangle$ )
  with 1.prem(2) show unvisitedPath (removeAll m un) ?ns' by (subst
   $\langle m = \text{last } ns \rangle, \text{rule } \text{unvisitedPath-removeLast}$ )
  from 1.prem(3) show  $\forall n \in \text{set } ?ns'. v \notin \text{defs } g \ n$  by (auto in-
  tro:in-set-butlastD)

```

**qed**  
**with**  $ns'(2)$  **show**  $?thesis$  **by**  $-(subst\ rec, rule\ fold\ union\ elemI, auto)$   
**qed**  
**qed**  
**qed**

**lemma**  $phiDefNodes\ auxE$ :  
**assumes**  $n \in phiDefNodes\ aux\ g\ v\ un\ m\ m \in set\ (\alpha n\ g)$   
**obtains**  $ns$  **where**  $g \vdash n - ns \rightarrow m \ \forall n \in set\ ns. v \notin defs\ g\ n\ length\ (predecessors\ g\ n) \neq 1\ unvisitedPath\ un\ ns$   
**using**  $assms$  **proof**  $(atomize\ elim, induction\ un\ arbitrary:m\ rule:removeAll\ induct)$   
**case**  $(1\ un)$   
**show**  $?case$   
**proof**  $(cases\ un\ v\ g\ m\ rule:phiDefNodes\ aux\ cases)$   
**case**  $nonrec$   
**thus**  $?thesis$  **using**  $1.prem$ s **by**  $simp$   
**next**  
**case**  $rec$   
**show**  $?thesis$   
**proof**  $(cases\ n \in (if\ length\ (predecessors\ g\ m) = 1\ then\ \{\}\ else\ \{m\}))$   
**case**  $True$   
**hence**  $n = m$  **by**  $(simp\ split:if\ split\ asm)$   
**thus**  $?thesis$  **using**  $1.prem$ s $(2)$   $rec\ True$  **by**  $auto$   
**next**  
**case**  $False$   
**with**  $rec\ 1.prem$ s $(1)$  **obtain**  $m'$  **where**  $m': n \in phiDefNodes\ aux\ g\ v\ (removeAll\ m\ un)\ m'\ m' \in set\ (predecessors\ g\ m)$   
**by**  $(auto\ elim!:fold\ union\ elem)$   
**with**  $1.prem$ s $(2)$  **have**  $m' \in set\ (\alpha n\ g)$  **by**  $auto$   
**with**  $1.IH[of\ m\ m']\ m'\ rec$  **obtain**  $ns$  **where**  $g \vdash n - ns \rightarrow m' \ \forall n \in set\ ns. v \notin defs\ g\ n\ length\ (predecessors\ g\ n) \neq 1\ unvisitedPath\ (removeAll\ m\ un)\ ns$  **by**  $auto$   
**thus**  $?thesis$  **using**  $m'\ rec$  **by**  $-(rule\ exI, auto)$   
**qed**  
**qed**  
**qed**

**lemma**  $phiDefNodesE$ :  
**assumes**  $n \in phiDefNodes\ g\ v$   
**obtains**  $ns\ m$  **where**  $g \vdash n - ns \rightarrow m \ \forall n \in set\ ns. v \notin defs\ g\ n\ v \in uses\ g\ m$   
**using**  $assms$   
**by**  $(auto\ elim!:phiDefNodes\ auxE\ elim!:fold\ union\ elem\ simp:phiDefNodes\ def)$

**lemma**  $phiDefNodes\ \alpha n[simp]$ :  $n \in phiDefNodes\ g\ v \implies n \in set\ (\alpha n\ g)$   
**by**  $(erule\ phiDefNodesE, auto)$

**lemma**  $phiDefNodesI$ :  
**assumes**  $g \vdash n - ns \rightarrow m\ v \in uses\ g\ m \ \forall n \in set\ ns. v \notin defs\ g\ n\ length\ (predecessors\ g\ n) \neq 1$

**shows**  $n \in \text{phiDefNodes } g \ v$   
**proof**–  
**from**  $\text{assms}(1)$  **have**  $m \in \text{set } (\alpha n \ g)$  **by**  $(\text{rule path2-in-}\alpha n, \text{ auto})$   
**from**  $\text{assms}$  **obtain**  $ns'$  **where**  $g \vdash n - ns' \rightarrow m$   $\text{distinct } ns' \ \forall n \in \text{set } ns'. \ v \notin$   
 $\text{defs } g \ n$  **by**  $-(\text{rule simple-path2}, \text{ auto})$   
**with**  $\text{assms}(4)$  **have**  $1: n \in \text{phiDefNodes-aux } g \ v \ (\alpha n \ g) \ m$  **by**  $-(\text{rule phiDefNodes-auxI},$   
 $\text{auto intro:path2-in-}\alpha n)$   
**thus**  $?thesis$  **using**  $\text{assms}(2)$   $\langle m \in \text{set } (\alpha n \ g) \rangle$   
**unfolding**  $\text{phiDefNodes-def}$   
**by**  $-(\text{rule fold-union-elemI}, \text{ auto})$   
**qed**

**lemma**  $\text{lookupDef-cases}[consumes \ 1]$ :  
**assumes**  $n \in \text{set } (\alpha n \ g)$   
**obtains**  $(\text{SimpleDef}) \ v \in \text{defs } g \ n$   $\text{lookupDef } g \ n \ v = (v, n, \text{SimpleDef})$   
 $\quad | (\text{PhiDef}) \quad v \notin \text{defs } g \ n$   $\text{length } (\text{predecessors } g \ n) \neq 1$   $\text{lookupDef } g \ n \ v$   
 $= (v, n, \text{PhiDef})$   
 $\quad | (\text{rec}) \ m$  **where**  $v \notin \text{defs } g \ n$   $\text{predecessors } g \ n = [m]$   $m \in \text{set } (\alpha n \ g)$   
 $\text{lookupDef } g \ n \ v = \text{lookupDef } g \ m \ v$   
**proof**  $(\text{cases } v \in \text{defs } g \ n)$   
**case**  $\text{True}$   
**thus**  $thesis$  **using**  $\text{assms SimpleDef}$  **by**  $(\text{simp add:lookupDef.simps})$   
**next**  
**case**  $\text{False}$   
**thus**  $thesis$   
**proof**  $(\text{cases } \text{length } (\text{predecessors } g \ n) = 1)$   
**case**  $\text{True}$   
**then obtain**  $m$  **where**  $m: \text{predecessors } g \ n = [m]$  **by**  $(\text{cases } \text{predecessors } g$   
 $n, \text{ auto})$   
**hence**  $m \in \text{set } (\text{predecessors } g \ n)$  **by**  $\text{simp}$   
**thus**  $thesis$  **using**  $\text{False rec assms } m$  **by**  $-(\text{subst}(asm) \text{lookupDef.simps}, \text{ drule}$   
 $\text{predecessor-is-node}, \text{ auto})$   
**next**  
**case**  $\text{False}$   
**thus**  $thesis$  **using**  $\langle v \notin \text{defs } g \ n \rangle \text{assms}$  **by**  $-(\text{rule PhiDef}, \text{ assumption},$   
 $\text{assumption}, \text{ subst lookupDef.simps}, \text{ auto split:list.split})$   
**qed**  
**qed**

**lemma**  $\text{lookupDef-cases}'[consumes \ 1]$ :  
**assumes**  $n \in \text{set } (\alpha n \ g)$   
**obtains**  $(\text{SimpleDef}) \ v \in \text{defs } g \ n$   $\text{defNode } (\text{lookupDef } g \ n \ v) = n$   $\text{defKind}$   
 $(\text{lookupDef } g \ n \ v) = \text{SimpleDef}$   
 $\quad | (\text{PhiDef}) \quad v \notin \text{defs } g \ n$   $\text{length } (\text{predecessors } g \ n) \neq 1$   $\text{lookupDefNode } g$   
 $n \ v = n$   $\text{defKind } (\text{lookupDef } g \ n \ v) = \text{PhiDef}$   
 $\quad | (\text{rec}) \ m$  **where**  $v \notin \text{defs } g \ n$   $\text{predecessors } g \ n = [m]$   $m \in \text{set } (\alpha n \ g)$   
 $\text{lookupDef } g \ n \ v = \text{lookupDef } g \ m \ v$   
**using**  $\text{assms}$   
**by**  $(\text{rule lookupDef-cases}[of } n \ g \ v]) \text{simp-all}$



**lemma** *lookupDefE*:  
**assumes**  $lookupDef\ g\ n\ v = v' \ n \in set\ (\alpha n\ g)$   
**obtains** (*SimpleDef*)  $v \in defs\ g\ n\ v' = (v, n, SimpleDef)$   
| (*PhiDef*)  $v \notin defs\ g\ n\ length\ (predecessors\ g\ n) \neq 1\ v' = (v, n, PhiDef)$   
| (*rec*)  $m$  **where**  $v \notin defs\ g\ n\ predecessors\ g\ n = [m]\ m \in set\ (\alpha n\ g)\ v' =$   
 $lookupDef\ g\ m\ v$   
**using** *assms* **by**  $-(atomize-elim, cases\ rule:lookupDef-cases[of\ n\ g\ v], auto)$

**lemma** *lookupDef-induct*[*consumes 1, case-names SimpleDef PhiDef rec*]:  
**assumes**  $n \in set\ (\alpha n\ g)$   
 $\bigwedge n. \llbracket n \in set\ (\alpha n\ g); v \in defs\ g\ n; lookupDef\ g\ n\ v = (v, n, SimpleDef) \rrbracket$   
 $\implies P\ n$   
 $\bigwedge n. \llbracket n \in set\ (\alpha n\ g); v \notin defs\ g\ n; length\ (predecessors\ g\ n) \neq 1;$   
 $lookupDef\ g\ n\ v = (v, n, PhiDef) \rrbracket \implies P\ n$   
 $\bigwedge n\ m. \llbracket v \notin defs\ g\ n; predecessors\ g\ n = [m]; m \in set\ (\alpha n\ g); lookupDef$   
 $g\ n\ v = lookupDef\ g\ m\ v; P\ m \rrbracket \implies P\ n$   
**shows**  $P\ n$   
**apply** (*induct rule:lookupDef.induct*[**where**  $P = \lambda g' n v'. g'=g \wedge v'=v \wedge n \in$   
 $set\ (\alpha n\ g) \longrightarrow P\ n, of\ g\ v\ n, simplified\ (no-asm), THEN\ mp$ ])  
**apply** *clarsimp*  
**apply** (*rule-tac v=v and n=n in lookupDef-cases; auto intro: assms lookupDef-cases*)  
**by** (*rule assms(1)*)

**lemma** *lookupDef-induct'*[*consumes 2, case-names SimpleDef PhiDef rec*]:  
**assumes**  $n \in set\ (\alpha n\ g)\ lookupDef\ g\ n\ v = (v, n', def)$   
 $\llbracket v \in defs\ g\ n'; def = SimpleDef \rrbracket \implies P\ n'$   
 $\llbracket v \notin defs\ g\ n'; length\ (predecessors\ g\ n') \neq 1; def = PhiDef \rrbracket \implies P\ n'$   
 $\bigwedge n\ m. \llbracket v \notin defs\ g\ n; predecessors\ g\ n = [m]; m \in set\ (\alpha n\ g); lookupDef$   
 $g\ n\ v = lookupDef\ g\ m\ v; P\ m \rrbracket \implies P\ n$   
**shows**  $P\ n$   
**using** *assms(1,2)*  
**proof** (*induction rule:lookupDef-induct*[**where**  $v=v$ ])  
**case** (*SimpleDef n*)  
**with** *assms(2)* **have**  $n = n'\ def = SimpleDef$  **by** *auto*  
**with** *SimpleDef assms(3)* **show** *?case* **by** *simp*  
**next**  
**case** (*PhiDef n*)  
**with** *assms(2)* **have**  $n = n'\ def = PhiDef$  **by** *auto*  
**with** *PhiDef assms(4)* **show** *?case* **by** *simp*  
**qed** (*rule assms(5), simp-all*)

**lemma** *lookupDef-lookup[simp]*:  
**assumes**  $lookupDef\ g\ n\ v = (v', n', def)\ n \in set\ (\alpha n\ g)$   
**shows**  $v' = v$   
**using** *assms(1) assms(2)* **by** (*induction rule:lookupDef.induct*) (*auto elim:lookupDefE*)

**lemma** *lookupDef-lookup'*:  
**assumes**  $(v', n', def) = lookupDef\ g\ n\ v\ n \in set\ (\alpha n\ g)$

**shows**  $v' = v$   
**using**  $assms(1)[symmetric]$   $assms(2)$  **by** (rule  $lookupDef-looksup$ )

**lemma**  $lookupDef-looksup''$ :  
**assumes**  $n \in set(\alpha n g)$   
**obtains**  $n' def$  **where**  $lookupDef g n v = (v, n', def)$   
**apply**  $atomize-elim$   
**using**  $assms$  **by** (induction rule: $lookupDef.induct$ ) (cases rule: $lookupDef-cases$ ,  $auto$ )

**lemma**  $lookupDef-fst[simp]$ :  $n \in set(\alpha n g) \implies fst(lookupDef g n v) = v$   
**by** ( $metis$   $fst-conv$   $lookupDef-looksup''$ )

**lemma**  $lookupDef-to-\alpha n$ :  
**assumes**  $lookupDef g n v = (v', n', def)$   $n \in set(\alpha n g)$   
**shows**  $n' \in set(\alpha n g)$   
**using**  $assms(2, 1)$   
**by** (induction rule: $lookupDef.induct[of n g v]$ )  $simp-all$

**lemma**  $lookupDef-to-\alpha n'[simp]$ :  
**assumes**  $lookupDef g n v = val$   $n \in set(\alpha n g)$   
**shows**  $defNode val \in set(\alpha n g)$   
**using**  $assms$  **by** (cases  $val$ ) ( $auto$   $simp:lookupDef-to-\alpha n$ )

**lemma**  $lookupDef-induct''[consumes 2, case-names SimpleDef PhiDef rec]$ :  
**assumes**  $lookupDef g n v = val$   $n \in set(\alpha n g)$   
 $\llbracket v \in defs g (defNode val); defKind val = SimpleDef \rrbracket \implies P (defNode val)$   
 $\llbracket v \notin defs g (defNode val); length(predecessors g (defNode val)) \neq 1; defKind val = PhiDef \rrbracket \implies P (defNode val)$   
 $\bigwedge n m. \llbracket v \notin defs g n; predecessors g n = [m]; m \in set(\alpha n g); lookupDef g n v = lookupDef g m v; P m \rrbracket \implies P n$   
**shows**  $P n$   
**using**  $assms$   
**apply** (cases  $val$ )  
**apply** ( $simp$ )  
**apply** (erule  $lookupDef-induct'$ )  
**using**  $assms(2)$  **by**  $auto$

**lemma**  $defs'-finite$ :  $finite(defs' g n)$   
**unfolding**  $defs'-def$  **using**  $defs'-finite$   
**by**  $simp$

**lemma**  $uses'-finite$ :  $finite(uses' g n)$   
**unfolding**  $uses'-def$  **using**  $uses'-finite$   
**by**  $simp$

**lemma**  $defs'-uses'-disjoint$ :  $n \in set(\alpha n g) \implies defs' g n \cap uses' g n = \{\}$   
**unfolding**  $defs'-def$   $uses'-def$  **using**  $defs'-uses'-disjoint$

**by** (*auto dest:lookupDef-looksup'*)

**lemma** *allDefs'-disjoint*:  $n \in \text{set } (\alpha n \ g) \implies m \in \text{set } (\alpha n \ g) \implies n \neq m$   
 $\implies (\text{defs}' \ g \ n \cup \{v. (n, v) \in \text{dom } (\text{phis}' \ g)\}) \cap (\text{defs}' \ g \ m \cup \{v. (m, v) \in \text{dom } (\text{phis}' \ g)\}) = \{\}$   
**by** (*auto split:if-split-asm simp: defs'-def phis'-def*)

**lemma** *phiDefNodes-aux-finite*: *finite* (*phiDefNodes-aux* *g v un m*)  
**proof** (*induction un arbitrary:m rule:removeAll-induct*)  
  **case** (*1 un*)  
  **thus** *?case* **by** (*cases un v g m rule:phiDefNodes-aux-cases*) *auto*  
**qed**

**lemma** *phis'-finite*: *finite* (*dom* (*phis'* *g*))

**proof**–

**let** *?super* = *set* ( $\alpha n \ g$ )  $\times$  *vars* *g*  $\times$  *set* ( $\alpha n \ g$ )  $\times$  {*PhiDef*}  
  **have** *finite ?super* **by** *auto*  
  **thus** *?thesis*  
  **by** – (*rule finite-subset[of - ?super], auto simp:phis'-def split:if-split-asm*)  
**qed**

**lemma** *phis'-wf*: *phis'* *g* (*n, v*) = *Some args*  $\implies$  *length* (*predecessors* *g n*) = *length args*

**unfolding** *phis'-def predecessors-def* **by** (*auto split:prod.splits if-split-asm*)

**lemma** *simpleDefs-phiDefs-disjoint*:  $n \in \text{set } (\alpha n \ g) \implies \text{defs}' \ g \ n \cap \{v. (n, v) \in \text{dom } (\text{phis}' \ g)\} = \{\}$

**unfolding** *phis'-def defs'-def* **by** *auto*

**lemma** *oldDefs-correct*: *defs* *g n* = *var* *g* ' *defs'* *g n*  
**by** (*simp add:defs'-def image-image*)

**lemma** *oldUses-correct*:  $n \in \text{set } (\alpha n \ g) \implies \text{uses} \ g \ n = \text{var} \ g \ ' \ \text{uses}' \ g \ n$   
**by** (*simp add:uses'-def image-image*)

**lemmas** *base-SSA-defs* = *CFG-SSA-base.CFG-SSA-defs*

**sublocale** *braun-ssa*: *CFG-SSA*  $\alpha e \ \alpha n$  *invar inEdges' Entry defs' uses' phis'*  
**apply** *unfold-locales*

**apply** (*rule defs'-uses'-disjoint, simp-all*)  
  **apply** (*rule defs'-finite*)  
  **apply** (*auto simp add: uses'-def uses-in- $\alpha n$* )[*1*]  
  **apply** (*rule uses'-finite*)  
  **apply** (*rule invar*)  
  **apply** (*rule phis'-finite*)  
  **apply** (*auto simp: phis'-def split: if-split-asm*)[*1*]  
  **apply** (*rule phis'-wf, simp-all add: base-SSA-defs*)  
  **apply** (*erule simpleDefs-phiDefs-disjoint*)  
  **apply** (*erule allDefs'-disjoint, simp, simp*)

```

done
end

declare (in CFG) invar[rule del]
declare (in CFG) Entry-no-predecessor[simp del]
context CFG-Construct
begin
  declare invar[intro!]
  declare Entry-no-predecessor[simp]

  lemma no-disjoint-cycle[simp]:
    assumes  $g \vdash n - ns \rightarrow n$  distinct ns
    shows  $ns = [n]$ 
  using assms unfolding path2-def
  by (metis distinct.simps(2) hd-Cons-tl last-in-set last-tl path-not-Nil)

  lemma lookupDef-path:
    assumes  $m \in \text{set } (\alpha n g)$ 
    obtains ns where  $g \vdash \text{lookupDefNode } g m v - ns \rightarrow m$  ( $\forall x \in \text{set } (tl ns). v \notin$ 
    defs g x)
  apply atomize-elim
  using assms proof (induction rule:lookupDef-induct[of m g v])
    case (SimpleDef n)
    thus ?case by  $-(rule \text{exI}[of - [n]], auto)$ 
  next
    case (PhiDef n)
    thus ?case by  $-(rule \text{exI}[of - [n]], auto)$ 
  next
    case (rec m m')
    then obtain ns where  $g \vdash \text{lookupDefNode } g m v - ns \rightarrow m' \forall x \in \text{set } (tl ns). v$ 
     $\notin$  defs g x by auto
    with rec.hyps(1,2) show ?case by  $-(rule \text{exI}[of - ns@[m]], auto simp:$ 
    path2-not-Nil)
  qed

  lemma lookupDef-path-conventional:
    assumes  $g \vdash n - ns \rightarrow m$   $n = \text{lookupDefNode } g m v n \notin \text{set } (tl ns)$   $x \in \text{set } (tl$ 
    ns)  $v' \in \text{braun-ssa.allDefs } g x$ 
    shows  $\text{var } g v' \neq v$ 
  using assms(1-4) proof (induction rule:path2-rev-induct)
    case empty
    from empty.prem(3) have False by simp
    thus ?case ..
  next
    case (snoc ns m m')
    note snoc.prem(1)[simp]
    from snoc.hyps have  $p: g \vdash n - ns@[m'] \rightarrow m'$  by auto
    hence  $m' \in \text{set } (\alpha n g)$  by auto
    thus ?thesis

```

```

proof (cases rule:lookupDef-cases'[of m' g v])
  case SimpleDef
  with snoc.premis(2,3) have False by (simp add:tl-append split:list.split-asm)
  thus ?thesis ..
next
  case PhiDef
  with snoc.premis(2,3) have False by (simp add:tl-append split:list.split-asm)
  thus ?thesis ..
next
  case (rec m2)
  from this(2) snoc.hyps(2) have[simp]: m2 = m by simp
  show ?thesis
  proof (cases x ∈ set (tl ns))
    case True
    with rec(4) snoc.premis(2) show ?thesis by - (rule snoc.IH, simp-all
add:tl-append split:list.split-asm)
    next
    case False
    with snoc.premis(3) have[simp]: x = m' by (simp add:tl-append split:list.split-asm)

    show ?thesis
    proof (cases v' ∈ defs' g x)
      case True
      with rec(1) show ?thesis by (auto simp add:defs'-def)
    next
      case False
      with assms(5) have v' ∈ braun-ssa.phiDefs g m' by (simp add:braun-ssa.allDefs-def)
      hence m' ∈ phiDefNodes g (fst v')
      unfolding braun-ssa.phiDefs-def by (auto simp add:phis'-def split:prod.split-asm
if-split-asm)
      with rec(2) show ?thesis by (auto dest:phiDefNode-is-join-node)
    qed
  qed
qed
qed

```

```

lemma allUse-lookupDef:
  assumes v ∈ braun-ssa.allUses g m m ∈ set (αn g)
  shows lookupDef g m (var g v) = v
  proof (cases v ∈ uses' g m)
    case True
    then obtain v' where v': v = lookupDef g m v' v' ∈ uses g m by (auto simp
add:uses'-def)
    with assms(2) have var g v = v' unfolding var-def by (metis lookupDef-fst)
    with v' show ?thesis by simp
  next
  case False
  with assms(1) obtain m' v' vs where (m,v) ∈ set (zip (predecessors g m')
vs) phis' g (m', v') = Some vs

```

by (auto simp add:braun-ssa.allUses-def elim:braun-ssa.phiUsesE)  
 hence  $l: v = \text{lookupDef } g \ m \ (\text{var } g \ v')$  by (auto simp add:phis'-def split:prod.split-asm  
 if-split-asm elim:in-set-zip-map)  
 with *assms*(2) have  $\text{var } g \ v = \text{var } g \ v'$  unfolding *var-def* by (metis *lookupDef-fst*)  
 with *l* show ?thesis by simp  
 qed

**lemma** *phis'-fst*:  
 assumes  $\text{phis}' \ g \ (n, v) = \text{Some } vs \ v' \in \text{set } vs$   
 shows  $\text{var } g \ v' = \text{var } g \ v$   
 using *assms* by (auto intro!:*lookupDef-fst dest*!:*phiDefNodes- $\alpha$ n simp add:phis'-def split:prod.split-asm if-split-asm*)

**lemma** *allUse-simpleUse*:  
 assumes  $v \in \text{braun-ssa.allUses } g \ m \ m \in \text{set } (\alpha n \ g)$   
 obtains  $ms \ m'$  where  $g \vdash m - ms \rightarrow m' \ \text{var } g \ v \in \text{uses } g \ m' \ \forall x \in \text{set } (\text{tl } ms).$   
 $\text{var } g \ v \notin \text{defs } g \ x$   
**proof** (cases  $v \in \text{uses}' \ g \ m$ )  
 case *True*  
 then obtain  $v'$  where  $v': v = \text{lookupDef } g \ m \ v' \ v' \in \text{uses } g \ m$  by (auto simp  
*add:uses'-def*)  
 with *assms*(2) have  $\text{var } g \ v = v'$  unfolding *var-def* by (metis *lookupDef-fst*)  
 with  $v'$  *assms*(2) show ?thesis by - (rule *that*, auto)  
 next  
 case *False*  
 with *assms*(1) obtain  $m' \ v' \ vs$  where  $\text{phi}: (m, v) \in \text{set } (\text{zip } (\text{predecessors } g \ m') \ vs)$   
 $\text{phis}' \ g \ (m', v') = \text{Some } vs$   
 by (auto simp add:braun-ssa.allUses-def elim:braun-ssa.phiUsesE)  
 hence  $m': m' \in \text{phiDefNodes } g \ (\text{var } g \ v')$  by (auto simp add:phis'-def split:prod.split-asm  
*if-split-asm*)  
 from *phi* have [*simp*]:  $\text{var } g \ v = \text{var } g \ v'$  by - (rule *phis'-fst*, auto)  
 from  $m'$  obtain  $m'' \ ms$  where  $g \vdash m' - ms \rightarrow m'' \ \forall x \in \text{set } ms. \ \text{var } g \ v' \notin \text{defs}$   
 $g \ x \ \text{var } g \ v' \in \text{uses } g \ m''$  by (erule *phiDefNodesE*)  
 with *phi*(1) show ?thesis by - (rule *that*[*of m#ms m''*], auto simp *del:var-def*)  
 qed

**lemma** *defs'*:  $v \in \text{defs}' \ g \ n \iff \text{var } g \ v \in \text{defs } g \ n \wedge \text{defKind } v = \text{SimpleDef} \wedge \text{defNode } v = n$   
 by (cases *v*, auto simp *add:defs'-def*)

**lemma** *use-implies-allDef*:  
 assumes  $\text{lookupDef } g \ m \ (\text{var } g \ v) = v \ m \in \text{set } (\alpha n \ g) \ \text{var } g \ v \in \text{uses } g \ m' \ g \vdash m - ms \rightarrow m' \ \forall x \in \text{set } (\text{tl } ms). \ \text{var } g \ v \notin \text{defs } g \ x$   
 shows  $v \in \text{braun-ssa.allDefs } g \ (\text{defNode } v)$   
 using *assms* **proof** (*induction arbitrary:ms rule:lookupDef-induct'*)  
 case *SimpleDef*  
 hence  $v \in \text{defs}' \ g \ (\text{defNode } v)$  by (*simp add:defs'*)  
 thus ?case by (*simp add:braun-ssa.allDefs-def*)  
 next

**case** *PhiDef*  
**from** *PhiDef.prem*s(1,2) **have** *vars*:  $\text{var } g \ v \in \text{vars } g$  **by** *auto*  
**from** *PhiDef.hyps*(1) *PhiDef.prem*s(2,3) **have**  $\forall n \in \text{set } ms. \text{var } g \ v \notin \text{defs } g$   
*n* **by** (*metis hd-Cons-tl path2-def path2-not-Nil set-ConsD*)  
**with** *PhiDef* **have** *defNode*  $v \in \text{phiDefNodes } g$  ( $\text{var } g \ v$ ) **by** – (*rule phiDefNodesI*)  
**with** *PhiDef.hyps*(3) *vars* **have**  $v \in \text{braun-ssa.phiDefs } g$  (*defNode*  $v$ ) **un-**  
**folding** *braun-ssa.phiDefs-def* **by** (*auto simp add: phis'-def split:prod.split*)  
**thus** ?*case* **by** (*simp add:braun-ssa.allDefs-def*)  
**next**  
**case** (*rec n m*)  
**from** *rec.hyps*(1) *rec.prem*s(2,3) **have**  $\forall n \in \text{set } ms. \text{var } g \ v \notin \text{defs } g \ n$  **by**  
(*metis hd-Cons-tl path2-def path2-not-Nil set-ConsD*)  
**with** *rec* **show** ?*case* **by** – (*rule rec.IH[of m#ms], auto*)  
**qed**

**lemma** *allUse-defNode-in- $\alpha$ n[simp]*:  
**assumes**  $v \in \text{braun-ssa.allUses } g \ m \ m \in \text{set } (\alpha \ n \ g)$   
**shows** *defNode*  $v \in \text{set } (\alpha \ n \ g)$   
**proof**–  
**let** ?*n* = *defNode* (*lookupDef*  $g \ m$  ( $\text{var } g \ v$ ))  
**from** *assms*(1,2) **have** *l*: *lookupDef*  $g \ m$  ( $\text{var } g \ v$ ) =  $v$  **by** (*rule allUse-lookupDef*)  
**from** *assms* **obtain** *ns* **where** *ns*:  $g \vdash ?n - ns \rightarrow m$  **by** – (*rule lookupDef-path, auto*)  
**with** *l* **show** ?*thesis* **by** *auto*  
**qed**

**lemma** *allUse-implies-allDef*:  
**assumes**  $v \in \text{braun-ssa.allUses } g \ m \ m \in \text{set } (\alpha \ n \ g)$   
**shows**  $v \in \text{braun-ssa.allDefs } g$  (*defNode*  $v$ )  
**proof**–  
**let** ?*n* = *defNode* (*lookupDef*  $g \ m$  ( $\text{var } g \ v$ ))  
**from** *assms*(1,2) **have** *l*: *lookupDef*  $g \ m$  ( $\text{var } g \ v$ ) =  $v$  **by** (*rule allUse-lookupDef*)  
**from** *assms* **obtain** *ns* **where** *ns*:  $g \vdash ?n - ns \rightarrow m \ \forall x \in \text{set } (tl \ ns). \text{var } g \ v \notin$   
*defs*  $g \ x$  **by** – (*rule lookupDef-path, auto*)  
**from** *assms* **obtain** *ms*  $m'$  **where**  $g \vdash m - ms \rightarrow m' \ \text{var } g \ v \in \text{uses } g \ m' \ \forall x \in$   
*set* (*tl* *ms*).  $\text{var } g \ v \notin \text{defs } g \ x$  **by** – (*rule allUse-simpleUse*)  
**hence**  $v \in \text{braun-ssa.allDefs } g$  (*defNode*  $v$ ) **using** *ns* *assms*(2) *l* **by** – (*rule use-implies-allDef, auto*)  
**with** *assms*(2) *l* **show** ?*thesis* **by** *simp*  
**qed**

**lemma** *conventional*:  
**assumes**  $g \vdash n - ns \rightarrow m \ n \notin \text{set } (tl \ ns) \ v \in \text{braun-ssa.allDefs } g \ n \ v \in \text{braun-ssa.allUses}$   
 $g \ m$   
 $x \in \text{set } (tl \ ns) \ v' \in \text{braun-ssa.allDefs } g \ x$   
**shows**  $\text{var } g \ v' \neq \text{var } g \ v$   
**proof**–  
**from** *assms*(1) **have**[*simp*]:  $m \in \text{set } (\alpha \ n \ g)$  **by** *auto*

**from** *assms*(4) **have**[*simp*]: *lookupDef g m (var g v) = v* **by** – (*rule allUse-lookupDef, auto*)

**from** *assms*(1,4) **have**  $v \in \text{braun-ssa.allDefs } g \text{ (defNode } v)$  **by** – (*rule allUse-implies-allDef, auto*)

**with** *assms*(1,3,4) *braun-ssa.allDefs-disjoint*[of  $n \ g \ \text{defNode } v$ ] **have**[*simp*]:  $\text{defNode } v = n$  **by** – (*rule braun-ssa.allDefs-disjoint', auto*)

**from** *assms* **show** *?thesis* **by** – (*rule lookupDef-path-conventional*[**where**  $m=m$ ], *simp-all add:uses'-def del:var-def*)  
**qed**

**lemma** *allDefs-var-disjoint-aux*:  $n \in \text{set } (\alpha n \ g) \implies v \in \text{defs } g \ n \implies n \notin \text{phiDefNodes } g \ v$   
**by** (*auto elim!:phiDefNodesE dest:path2-hd-in-ns*)

**lemma** *allDefs-var-disjoint*:  $\llbracket n \in \text{set } (\alpha n \ g); v \in \text{braun-ssa.allDefs } g \ n; v' \in \text{braun-ssa.allDefs } g \ n; v \neq v' \rrbracket \implies \text{var } g \ v' \neq \text{var } g \ v$   
**unfolding** *braun-ssa.allDefs-def braun-ssa.phiDefs-def*  
**by** (*auto simp: defs'-def phis'-def allDefs-var-disjoint-aux split:prod.splits if-split-asm*)

**lemma**[*simp*]:  $n \in \text{set } (\alpha n \ g) \implies v \in \text{defs } g \ n \implies \text{lookupDefNode } g \ n \ v = n$   
**by** (*cases rule:lookupDef-cases*[of  $n \ g \ v$ ]) *simp-all*

**lemma**[*simp*]:  $n \in \text{set } (\alpha n \ g) \implies \text{length } (\text{predecessors } g \ n) \neq 1 \implies \text{lookupDefNode } g \ n \ v = n$   
**by** (*cases rule:lookupDef-cases*[of  $n \ g \ v$ ]) *simp-all*

**lemma** *lookupDef-idem*[*simp*]:  
**assumes**  $n \in \text{set } (\alpha n \ g)$   
**shows**  $\text{lookupDef } g \ (\text{lookupDefNode } g \ n \ v) \ v = \text{lookupDef } g \ n \ v$   
**using** *assms* **by** (*induction rule:lookupDef-induct''*[of  $g \ n \ v$ , *OF refl*]) (*simp-all add:assms*)  
**end**

**locale** *CFG-Construct-wf* = *CFG-Construct*  $\alpha e \ \alpha n \ \text{invar } \text{inEdges}' \ \text{Entry } \text{defs } \text{uses}$   
+ *CFG-wf*  $\alpha e \ \alpha n \ \text{invar } \text{inEdges}' \ \text{Entry } \text{defs } \text{uses}$

**for**  
 $\alpha e :: 'g \Rightarrow ('node::\text{linorder} \times 'edgeD \times 'node) \ \text{set}$  **and**  
 $\alpha n :: 'g \Rightarrow 'node \ \text{list}$  **and**  
 $\text{invar} :: 'g \Rightarrow \text{bool}$  **and**  
 $\text{inEdges}' :: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD) \ \text{list}$  **and**  
 $\text{Entry} :: 'g \Rightarrow 'node$  **and**  
 $\text{defs} :: 'g \Rightarrow 'node \Rightarrow 'var::\text{linorder} \ \text{set}$  **and**  
 $\text{uses} :: 'g \Rightarrow 'node \Rightarrow 'var \ \text{set}$

**begin**  
**lemma** *def-ass-allUses-aux*:  
**assumes**  $g \vdash \text{Entry } g - ns \rightarrow n$   
**shows**  $\text{lookupDefNode } g \ n \ (\text{var } g \ v) \in \text{set } ns$



```

proof–
  from assms have[simp]:  $n \in \text{set } (\alpha n \ g)$  by auto
  thus ?thesis using assms
  proof (induction arbitrary:ns rule:lookupDef-induct''[of g n var g v, OF refl,
consumes 1])
    case ( $\exists m \ m' \ ns$ )
      show ?case
      proof (cases length ns  $\geq 2$ )
        case False
          with  $\exists.prem$ s have  $m = \text{Entry } g$  by (metis path2-nontrivial)
          with  $\exists.hyps(2)$  have False by simp
          thus ?thesis ..
        next
          case True
          with  $\exists.prem$ s have  $g \vdash \text{Entry } g - \text{butlast } ns \rightarrow m'$ 
          by (rule path2-unsnoc) (simp add: $\exists.hyps(2)$ )
          with  $\exists.hyps \ \exists.IH$ [of butlast ns] show ?thesis by (simp add:in-set-butlastD)
        qed
      qed auto
    qed

```

```

lemma def-ass-allUses:
  assumes  $v \in \text{braun-ssa.allUses } g \ n \ n \in \text{set } (\alpha n \ g)$ 
  shows  $\text{braun-ssa.defAss } g \ n \ v$ 
  proof (rule braun-ssa.defAssI)
    fix ns
    assume asm:  $g \vdash \text{Entry } g - ns \rightarrow n$ 
    let  $?m = \text{lookupDefNode } g \ n \ (\text{var } g \ v)$ 
    from asm have  $?m \in \text{set } ns$  by (rule def-ass-allUses-aux)
    moreover from assms allUse-lookupDef have  $?m = \text{defNode } v$  by simp
    moreover from assms allUse-implies-allDef have  $v \in \text{braun-ssa.allDefs } g$ 
    (defNode v) by simp
    ultimately show  $\exists n \in \text{set } ns. v \in \text{braun-ssa.allDefs } g \ n$  by auto
  qed

```

```

lemma Empty-no-phis:
  shows  $\text{phis}' \ g \ (\text{Entry } g, v) = \text{None}$ 
  proof–
    have  $\bigwedge v. \text{Entry } g \notin \text{phiDefNodes } g \ v$ 
    proof (rule, rule phiDefNodesE, assumption)
      fix  $v \ ns \ m$ 
      assume asm:  $g \vdash \text{Entry } g - ns \rightarrow m \ \forall n \in \text{set } ns. v \notin \text{defs } g \ n \ v \in \text{uses } g \ m$ 
      hence  $m \in \text{set } (\alpha n \ g)$  by auto
      from def-ass-uses[of g, THEN bspec[OF - this], THEN bspec[OF - asm( $\exists$ )]]
      asm
      show False by (auto elim!:defAss'E)
    qed
    thus ?thesis by (auto simp:phis'-def split:prod.split)
  qed

```

```

lemma braun-ssa-CFG-SSA-wf:
  CFG-SSA-wf  $\alpha e \alpha n$  invar inEdges' Entry defs' uses' phis'
apply unfold-locales
apply (erule def-ass-allUses, assumption)
apply (rule Empty-no-phis)
done

sublocale braun-ssa: CFG-SSA-wf  $\alpha e \alpha n$  invar inEdges' Entry defs' uses' phis'
by (rule braun-ssa-CFG-SSA-wf)

lemma braun-ssa-CFG-SSA-Transformed:
  CFG-SSA-Transformed  $\alpha e \alpha n$  invar inEdges' Entry defs uses defs' uses' phis'
var
apply unfold-locales
  apply (rule oldDefs-correct)
  apply (erule oldUses-correct)
  apply (erule conventional, simp, simp, simp, simp, simp)
  apply (erule phis'-fst, simp)
apply (erule allDefs-var-disjoint, simp, simp, simp)
done

sublocale braun-ssa: CFG-SSA-Transformed  $\alpha e \alpha n$  invar inEdges' Entry defs
uses defs' uses' phis' var
by (rule braun-ssa-CFG-SSA-Transformed)

lemma PhiDef-defNode-eq:
  assumes  $n \in \text{set } (\alpha n g) \ n \in \text{phiDefNodes } g \ v \ v \in \text{vars } g$ 
  shows braun-ssa.defNode  $g \ (v, n, \text{PhiDef}) = n$ 
  using assms by - (rule braun-ssa.defNode-eq, rule assms(1), subst braun-ssa.allDefs-def,
subst braun-ssa.phiDefs-def, auto simp: phis'-def)

lemma phiDefNodes-aux-pruned-aux:
  assumes  $n \in \text{phiDefNodes-aux } g \ v \ (\alpha n g) \ n \text{Use } v \in \text{uses } g \ n \text{Use } g \vdash n - ns \rightarrow m$ 
 $g \vdash m - ms \rightarrow n \text{Use } \text{braun-ssa.liveVal } g \ (\text{lookupDef } g \ m \ v) \ \forall n \in \text{set } (ns @ ms). \ v \notin$ 
 $\text{defs } g \ n$ 
  shows braun-ssa.liveVal  $g \ (v, n, \text{PhiDef})$ 
  using assms(3-) proof (induction  $n \ ns \ m$  arbitrary:  $ms$  rule: path2-rev-induct)
  case empty
  with assms(1) have lookupDef  $g \ n \ v = (v, n, \text{PhiDef})$ 
  by -(drule phiDefNode-aux-is-join-node, cases rule: lookupDef-cases, auto)
  with empty.prem(2) show ?case by simp
next
  case (snoc  $ns \ m \ m'$ )
  from snoc.prem have  $m' \in \text{set } (\alpha n g)$  by auto
  thus ?case
  proof (cases rule: lookupDef-cases[where  $v=v$ ])
  case SimpleDef
  with snoc.prem(3) have False by simp

```

```

thus ?thesis..
next
have step: braun-ssa.liveVal g (lookupDef g m v)  $\implies$  ?thesis
proof (rule snoc.IH)
  from snoc.prems(1) snoc.hyps(2) show  $g \vdash m - m \# ms \rightarrow n$  Use by auto
  from snoc.prems(3) snoc.hyps(1) show  $\forall n \in \text{set}(ns @ m \# ms). v \notin \text{defs}$ 
g n by auto
  qed
  {
    case rec
    from snoc.hyps(2) rec(2) have[simp]: predecessors g m' = [m] by auto
    with rec snoc.prems(2) have braun-ssa.liveVal g (lookupDef g m v) by auto
    with step show ?thesis.
  }
  next
  case PhiDef
  with snoc assms(2) have phiDefNode:  $m' \in \text{phiDefNodes } g v$  by  $\text{-(rule } \text{phiDefNodesI, auto)}$ 
  from assms(2,4) have vars:  $v \in \text{vars } g$  by auto
  have braun-ssa.liveVal g (lookupDef g m v)
  proof (rule braun-ssa.livePhi)
  from PhiDef(3) snoc.prems(2) show braun-ssa.liveVal g (v, m', PhiDef)
by simp
  from phiDefNode snoc.hyps(2) vars show braun-ssa.phiArg g (v, m', PhiDef)
(lookupDef g m v)
  by (subst braun-ssa.phiArg-def, subst braun-ssa.phi-def, subst PhiDef-defNode-eq,
auto simp: phi'-def)
  qed
  thus ?thesis by (rule step)
}
qed
qed

lemma phiDefNodes-aux-pruned:
  assumes  $m \in \text{phiDefNodes-aux } g v$  ( $\alpha n$  g)  $n \in \text{set } (\alpha n$  g)  $v \in \text{uses } g n$ 
  shows braun-ssa.liveVal g (v, m, PhiDef)
proof -
  from assms(1,2) obtain ns where  $ns: g \vdash m - ns \rightarrow n \forall n \in \text{set } ns. v \notin \text{defs}$ 
g n by (rule phiDefNodes-auxE)
  hence  $v \notin \text{defs } g n$  by (auto dest:path2-last simp: path2-not-Nil)
  with ns assms(1,3) show ?thesis
apply -
proof (rule phiDefNodes-aux-pruned-aux)
  from assms(2,3) show braun-ssa.liveVal g (lookupDef g n v) by  $\text{-(rule } \text{braun-ssa.liveSimple, auto simp add: uses'-def)}$ 
  qed auto
qed

theorem phi'-pruned: braun-ssa.pruned g
unfolding braun-ssa.pruned-def braun-ssa.phiDefs-def

```

```

apply (subst phis'-def)
by (auto split:prod.splits if-split-asm simp add:phiDefNodes-def elim!:fold-union-elem
phiDefNodes-aux-pruned)

declare var-def [simp del]

declare no-disjoint-cycle [simp del]
declare lookupDef-looksup [simp del]

declare lookupDef.simps [code]
declare phiDefNodes-aux.simps [code]
declare phiDefNodes-def [code]
declare defs'-def [code]
declare uses'-def [code]
declare phis'-def [code]
declare predecessors-def [code]
end

end

```

## 4.2 Inductive Removal of Trivial Phi Functions

```

theory Construct-SSA-notriv
imports SSA-CFG Minimality HOL-Library.While-Combinator
begin

locale CFG-SSA-Transformed-notriv-base = CFG-SSA-Transformed-base ae an
invar inEdges' Entry oldDefs oldUses defs uses phis var
for
  ae :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set and
  an :: 'g ⇒ 'node list and
  invar :: 'g ⇒ bool and
  inEdges' :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list and
  Entry::'g ⇒ 'node and
  oldDefs :: 'g ⇒ 'node ⇒ 'var::linorder set and
  oldUses :: 'g ⇒ 'node ⇒ 'var set and
  defs :: 'g ⇒ 'node ⇒ 'val::linorder set and
  uses :: 'g ⇒ 'node ⇒ 'val set and
  phis :: 'g ⇒ ('node, 'val) phis and
  var :: 'g ⇒ 'val ⇒ 'var +
fixes chooseNext-all :: ('node ⇒ 'val set) ⇒ ('node, 'val) phis ⇒ 'g ⇒ ('node ×
'val)
begin
  abbreviation chooseNext g ≡ snd (chooseNext-all (uses g) (phis g) g)
  abbreviation chooseNext' g ≡ chooseNext-all (uses g) (phis g) g

  definition substitution g ≡ THE v'. isTrivialPhi g (chooseNext g) v'
  definition substNext g ≡ λv. if v = chooseNext g then substitution g else v
  definition[simp]: uses' g n ≡ substNext g ' uses g n

```

```

definition[simp]: phis' g x  $\equiv$  case x of (n,v)  $\Rightarrow$  if v = chooseNext g
  then None
  else map-option (map (substNext g)) (phis g (n,v))
end

locale CFG-SSA-Transformed-notriv = CFG-SSA-Transformed  $\alpha e \alpha n$  invar in-
Edges' Entry oldDefs oldUses defs uses phis var
+ CFG-SSA-Transformed-notriv-base  $\alpha e \alpha n$  invar inEdges' Entry oldDefs oldUses
defs uses phis var chooseNext-all
for
 $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)$  set and
 $\alpha n :: 'g \Rightarrow 'node$  list and
invar :: 'g  $\Rightarrow$  bool and
inEdges' :: 'g  $\Rightarrow 'node \Rightarrow ('node \times 'edgeD)$  list and
Entry :: 'g  $\Rightarrow 'node$  and
oldDefs :: 'g  $\Rightarrow 'node \Rightarrow 'var::linorder$  set and
oldUses :: 'g  $\Rightarrow 'node \Rightarrow 'var$  set and
defs :: 'g  $\Rightarrow 'node \Rightarrow 'val::linorder$  set and
uses :: 'g  $\Rightarrow 'node \Rightarrow 'val$  set and
phis :: 'g  $\Rightarrow ('node, 'val)$  phis and
var :: 'g  $\Rightarrow 'val \Rightarrow 'var$  and
chooseNext-all :: ('node  $\Rightarrow 'val$  set)  $\Rightarrow ('node, 'val)$  phis  $\Rightarrow 'g \Rightarrow ('node \times 'val)$ 
+
assumes chooseNext-all: CFG-SSA-Transformed  $\alpha e \alpha n$  invar inEdges' Entry old-
Defs oldUses defs u p var  $\Longrightarrow$ 
CFG-SSA-wf-base.redundant  $\alpha n$  inEdges' defs u p g  $\Longrightarrow$ 
chooseNext-all (u g) (p g) g  $\in$  dom (p g)  $\wedge$ 
CFG-SSA-wf-base.trivial  $\alpha n$  inEdges' defs u p g (snd (chooseNext-all (u g) (p g)
g))
begin
lemma chooseNext':redundant g  $\Longrightarrow$  chooseNext' g  $\in$  dom (phis g)  $\wedge$  trivial g
(chooseNext g)
by (rule chooseNext-all, unfold-locales)

lemma chooseNext: redundant g  $\Longrightarrow$  chooseNext g  $\in$  allVars g  $\wedge$  trivial g
(chooseNext g)
by (drule chooseNext', auto simp: trivial-in-allVars)

lemmas chooseNext-in-allVars[simp] = chooseNext[THEN conjunct1]

lemma isTrivialPhi-det: trivial g v  $\Longrightarrow$   $\exists !v'. isTrivialPhi g v v'$ 
proof(rule ex-ex1I)
fix v' v''
assume isTrivialPhi g v v' isTrivialPhi g v v''
from this[unfolded isTrivialPhi-def, THEN conjunct2] show v' = v'' by (auto
simp:isTrivialPhi-def doubleton-eq-iff split:option.splits)
qed (auto simp: trivial-def)

lemma trivialPhi-strict-dom:

```

```

assumes[simp]:  $v \in \text{allVars } g$  and  $\text{triv}: \text{isTrivialPhi } g \ v \ v'$ 
shows  $\text{strict-def-dom } g \ v' \ v$ 
proof
  let  $?n = \text{defNode } g \ v$ 
  let  $?n' = \text{defNode } g \ v'$ 
  from  $\text{triv}$  obtain  $vs$  where  $vs: \text{phi } g \ v = \text{Some } vs \ (\text{set } vs = \{v'\} \vee \text{set } vs = \{v, v'\})$  by  $(\text{auto } \text{simp}: \text{isTrivialPhi-def } \text{split}: \text{option.splits})$ 
  hence  $?n \neq \text{Entry } g$  by  $\text{auto}$ 

  have  $\text{other-preds-dominated}: \bigwedge m. m \in \text{set } (\text{old.predecessors } g \ ?n) \implies v' \notin \text{phiUses } g \ m \implies \text{old.dominates } g \ ?n \ m$ 
  proof-
    fix  $m$ 
    assume  $m: m \in \text{set } (\text{old.predecessors } g \ ?n) \ v' \notin \text{phiUses } g \ m$ 
    hence[simp]:  $m \in \text{set } (\alpha n \ g)$  by  $\text{auto}$ 
    show  $\text{old.dominates } g \ ?n \ m$ 
    proof  $(\text{cases } v \in \text{phiUses } g \ m)$ 
      case  $\text{True}$ 
      hence  $v \in \text{allUses } g \ m$  by  $\text{simp}$ 
      thus  $?thesis$  by  $(\text{rule } \text{allUses-dominated}) \ \text{simp-all}$ 
    next
      case  $\text{False}$ 
      with  $vs$  have  $v' \in \text{phiUses } g \ m$  by  $-(\text{rule } \text{phiUses-exI}[\text{OF } m(1)], \text{auto } \text{simp}: \text{phi-def})$ 
      with  $m(2)$  show  $?thesis$  by  $\text{simp}$ 
    qed
  qed

  show  $?n' \neq ?n$ 
  proof  $(\text{rule } \text{notI})$ 
    assume  $\text{asm}: ?n' = ?n$ 
    have  $\bigwedge m. m \in \text{set } (\text{old.predecessors } g \ ?n) \implies v' \in \text{phiUses } g \ m \implies \text{old.dominates } g \ ?n \ m$ 
    proof-
      fix  $m$ 
      assume  $m \in \text{set } (\text{old.predecessors } g \ ?n) \ v' \in \text{phiUses } g \ m$ 
      hence  $\text{old.dominates } g \ ?n' \ m$  by  $-(\text{rule } \text{allUses-dominated}, \text{auto})$ 
      thus  $?thesis \ m$  by  $(\text{simp } \text{add}: \text{asm})$ 
    qed
    with  $\text{non-dominated-predecessor}[\text{of } ?n \ g] \ \text{other-preds-dominated } (\ ?n \neq \text{Entry } g)$  show  $\text{False}$  by  $\text{auto}$ 
  qed

  show  $\text{old.dominates } g \ ?n' \ ?n$ 
  proof
    fix  $ns$ 
    assume  $\text{asm}: g \vdash \text{Entry } g - ns \rightarrow ?n$ 
    from  $(?n \neq \text{Entry } g)$  obtain  $m \ ns'$ 
      where  $ns': g \vdash \text{Entry } g - ns' \rightarrow m \ m \in \text{set } (\text{old.predecessors } g \ ?n) \ ?n \notin \text{set}$ 

```

```

ns' set ns' ⊆ set ns
  by - (rule old.simple-path2-unsnoc[OF asm], auto)
  hence[simp]: m ∈ set (αn g) by auto
  from ns' have ¬old.dominates g ?n m by (auto elim:old.dominatesE)
  with other-preds-dominated[of m] ns'(2) have v' ∈ phiUses g m by auto
  hence old.dominates g ?n' m by - (rule allUses-dominated, auto)
  with ns'(1) have ?n' ∈ set ns' by - (erule old.dominatesE)
  with ns'(4) show ?n' ∈ set ns by auto
qed auto
qed

```

```

lemma isTrivialPhi-asymmetric:
assumes isTrivialPhi g a b
  and isTrivialPhi g b a
shows False
using assms
proof -
  from ⟨isTrivialPhi g a b⟩
  have b ∈ allVars g
    unfolding isTrivialPhi-def
  by (fastforce intro!: phiArg-in-allVars simp: phiArg-def split: option.splits)
  from ⟨isTrivialPhi g b a⟩
  have a ∈ allVars g
    unfolding isTrivialPhi-def
  by (fastforce intro!: phiArg-in-allVars simp: phiArg-def split: option.splits)
  from trivialPhi-strict-dom [OF ⟨a ∈ allVars g⟩ assms(1)]
    trivialPhi-strict-dom [OF ⟨b ∈ allVars g⟩ assms(2)]
  show ?thesis by blast
qed

```

```

lemma substitution[intro]: redundant g ⇒ isTrivialPhi g (chooseNext g) (substitution
g)
  unfolding substitution-def by (rule theI', rule isTrivialPhi-det, simp add:
chooseNext)

```

```

lemma trivialPhi-in-allVars[simp]:
  assumes isTrivialPhi g v v' and[simp]: v ∈ allVars g
  shows v' ∈ allVars g
proof -
  from assms(1) have phiArg g v v'
  unfolding phiArg-def
  by (auto simp: isTrivialPhi-def split: option.splits)
  thus v' ∈ allVars g by - (rule phiArg-in-allVars, auto)
qed

```

```

lemma substitution-in-allVars[simp]:
  assumes redundant g
  shows substitution g ∈ allVars g
using assms by - (rule trivialPhi-in-allVars, auto)

```

```

lemma defs-uses-disjoint-inv:
  assumes [simp]:  $n \in \text{set } (\alpha n \ g) \text{ redundant } g$ 
  shows  $\text{defs } g \ n \cap \text{uses}' \ g \ n = \{\}$ 
proof (rule equals0I)
  fix  $v'$ 
  assume asm:  $v' \in \text{defs } g \ n \cap \text{uses}' \ g \ n$ 
  then obtain  $v$  where  $v \in \text{uses } g \ n \ v' = \text{substNext } g \ v$  and  $v': v' \in \text{defs } g$ 
  by auto
  show False
  proof (cases v = chooseNext g)
    case False
    thus ?thesis using  $v \ v'$  defs-uses-disjoint[of n g] by (auto simp:substNext-def
split:if-split-asm)
  next
    case [simp]: True
    from  $v'$  have n-defNode:  $n = \text{defNode } g \ v'$  by - (rule defNode-eq[symmetric],
auto)
    from  $v(1)$  have [simp]:  $v \in \text{allVars } g$  by - (rule allUses-in-allVars[where
 $n=n$ ], auto)
    let  $?n' = \text{defNode } g \ v$ 
    have old.strict-dom  $g \ n \ ?n'$ 
    by (simp only:n-defNode v(2), rule trivialPhi-strict-dom, auto simp:substNext-def)
    moreover from  $v(1)$  have old.dominates  $g \ ?n' \ n$  by - (rule allUses-dominated,
auto)
    ultimately show False by auto
  qed
qed
end

```

**context** *CFG-SSA-wf*

**begin**

**inductive** *liveVal'* ::  $'g \Rightarrow 'val \ \text{list} \Rightarrow \text{bool}$

**for**  $g :: 'g$

**where**

*liveSimple'*:  $\llbracket n \in \text{set } (\alpha n \ g); \text{val} \in \text{uses } g \ n \rrbracket \Longrightarrow \text{liveVal}' \ g \ [\text{val}]$

| *livePhi'*:  $\llbracket \text{liveVal}' \ g \ (v\#\text{vs}); \text{phiArg } g \ v \ v' \rrbracket \Longrightarrow \text{liveVal}' \ g \ (v'\#\text{v}\#\text{vs})$

**lemma** *liveVal'-suffix*:

**assumes**  $\text{liveVal}' \ g \ \text{vs} \ \text{suffix} \ \text{vs}' \ \text{vs} \ \text{vs}' \neq []$

**shows**  $\text{liveVal}' \ g \ \text{vs}'$

**using** *assms* **proof** *induction*

**case** (*liveSimple' n v*)

**from** *liveSimple'.prems* **have**  $\text{vs}' = [v]$

**by** (*metis append-Nil butlast.simps(2) suffixI suffix-order.antisym suffix-unsnoc*)

**with** *liveSimple'.hyps* **show** *?case* **by** (*auto intro: liveVal'.liveSimple'*)

**next**

**case** (*livePhi' v vs v'*)

**show** *?case*



```

proof (cases vs' = v' # v # vs)
  case True
  with livePhi' show ?thesis by - (auto intro: liveVal'.livePhi')
next
  case False
  with livePhi'.prems have suffix vs' (v#vs)
    by (metis list.sel(3) self-append-conv2 suffixI suffix-take tl-append2)
  with livePhi'.prems(2) show ?thesis by - (rule livePhi'.IH)
qed
qed

lemma liveVal'I:
  assumes liveVal g v
  obtains vs where liveVal' g (v#vs)
using assms proof induction
  case (liveSimple n v)
  thus thesis by - (rule liveSimple(3), rule liveSimple')
next
  case (livePhi v v')
  show thesis
  proof (rule livePhi.IH)
    fix vs
    assume asm: liveVal' g (v#vs)
    show thesis
    proof (cases v' ∈ set (v#vs))
      case False
      with livePhi.hyps asm show thesis by - (rule livePhi.prems, rule livePhi')
    next
      case True
      then obtain vs' where suffix (v'#vs') (v#vs)
      by - (drule split-list-last, auto simp: Sublist.suffix-def)
      with asm show thesis by - (rule livePhi.prems, rule liveVal'-suffix,
simp-all)
    qed
  qed
qed

lemma liveVal'D:
  assumes liveVal' g vs vs = v#vs'
  shows liveVal g v
using assms proof (induction arbitrary: v vs')
  case (liveSimple' n vs)
  thus ?case by - (rule liveSimple, auto)
next
  case (livePhi' v2 vs v')
  thus ?case by - (rule livePhi, auto)
qed
end

```

**locale** *CFG-SSA-step* = *CFG-SSA-Transformed-notriv*  $\alpha e$   $\alpha n$  *invar* *inEdges'* *Entry* *oldDefs* *oldUses* *defs* *uses* *phis* *var* *chooseNext-all*  
**for**  
 $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node) \text{ set and}$   
 $\alpha n :: 'g \Rightarrow 'node \text{ list and}$   
*invar*  $:: 'g \Rightarrow \text{bool and}$   
*inEdges'*  $:: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD) \text{ list and}$   
*Entry*  $:: 'g \Rightarrow 'node \text{ and}$   
*oldDefs*  $:: 'g \Rightarrow 'node \Rightarrow 'var::linorder \text{ set and}$   
*oldUses*  $:: 'g \Rightarrow 'node \Rightarrow 'var \text{ set and}$   
*defs*  $:: 'g \Rightarrow 'node \Rightarrow 'val::linorder \text{ set and}$   
*uses*  $:: 'g \Rightarrow 'node \Rightarrow 'val \text{ set and}$   
*phis*  $:: 'g \Rightarrow ('node, 'val) \text{ phis and}$   
*var*  $:: 'g \Rightarrow 'val \Rightarrow 'var \text{ and}$   
*chooseNext-all*  $:: ('node \Rightarrow 'val \text{ set}) \Rightarrow ('node, 'val) \text{ phis} \Rightarrow 'g \Rightarrow ('node \times 'val)$   
**and**  
 $g :: 'g +$   
**assumes** *redundant[simp]*: *redundant g*  
**begin**  
**abbreviation**  $u\text{-}g \equiv \text{uses}(g := \text{uses}' g)$   
**abbreviation**  $p\text{-}g \equiv \text{phis}(g := \text{phis}' g)$   
  
**sublocale** *step*: *CFG-SSA-Transformed-notriv-base*  $\alpha e$   $\alpha n$  *invar* *inEdges'* *Entry* *oldDefs* *oldUses* *defs*  $u\text{-}g$   $p\text{-}g$  *var* *chooseNext-all* .  
  
**lemma** *simpleDefs-phiDefs-disjoint-inv*:  
**assumes**  $n \in \text{set } (\alpha n g)$   
**shows**  $\text{defs } g n \cap \text{step.phiDefs } g n = \{\}$   
**using** *simpleDefs-phiDefs-disjoint[OF assms]*  
**by** (*auto simp: phiDefs-def step.phiDefs-def dom-def split:option.splits*)  
  
**lemma** *allDefs-disjoint-inv*:  
**assumes**  $n \in \text{set } (\alpha n g)$   $m \in \text{set } (\alpha n g)$   $n \neq m$   
**shows**  $\text{step.allDefs } g n \cap \text{step.allDefs } g m = \{\}$   
**using** *allDefs-disjoint[OF assms]*  
**by** (*auto simp: CFG-SSA-defs step.CFG-SSA-defs dom-def split:option.splits*)  
  
**lemma** *phis-finite-inv*:  
**shows** *finite* (*dom* (*phis'*  $g$ ))  
**using** *phis-finite[of g]* **by** - (*rule finite-subset, auto split:if-split-asm*)  
  
**lemma** *phis-wf-inv*:  
**assumes**  $\text{phis}' g (n, v) = \text{Some args}$   
**shows**  $\text{length } (\text{old.predecessors } g n) = \text{length args}$   
**using** *phis-wf[of g]* *assms* **by** (*auto split:if-split-asm*)  
  
**sublocale** *step*: *CFG-SSA*  $\alpha e$   $\alpha n$  *invar* *inEdges'* *Entry* *defs*  $u\text{-}g$   $p\text{-}g$   
**apply** *unfold-locales*

```

apply (rename-tac g')
apply (case-tac g'=g)
  apply (simp add:defs-uses-disjoint-inv[simplified])
  apply (simp add:defs-uses-disjoint)
apply (rule defs-finite)
apply (auto simp: uses-in- $\alpha n$  split: if-split-asm)[1]
apply (rename-tac g' n)
apply (case-tac g'=g)
  apply simp
  apply simp
apply (rule invar)
apply (rename-tac g')
apply (case-tac g'=g)
  apply (simp add:phis-finite-inv)
  apply (simp add:phis-finite)
apply (auto simp: phis-in- $\alpha n$  split: if-split-asm)[1]
apply (rename-tac g' n v args)
apply (case-tac g'=g)
  apply (simp add:phis-wf-inv)
  apply (simp add:phis-wf)
apply (rename-tac g')
apply (case-tac g'=g)
  apply (simp add: simpleDefs-phiDefs-disjoint-inv)
apply (simp add: simpleDefs-phiDefs-disjoint[unfolded CFG-SSA-defs] step.CFG-SSA-defs)
)
apply (rename-tac g' m)
apply (case-tac g'=g)
  apply (simp add: allDefs-disjoint-inv)
apply (simp add: allDefs-disjoint[unfolded CFG-SSA-defs] step.CFG-SSA-defs)
done

```

```

lemma allUses-narrows:
  assumes  $n \in \text{set } (\alpha n g)$ 
  shows  $\text{step.allUses } g n \subseteq \text{substNext } g \text{ 'allUses } g n$ 
proof–
  have  $\bigwedge n' v' z b. \text{phis } g (n', v') = \text{Some } z \implies (n, b) \in \text{set } (\text{zip } (\text{old.predecessors } g n') z) \implies b \notin \text{phiUses } g n \implies b \in \text{uses } g n$ 
  proof–
    fix  $n' v' z b$ 
    assume  $(n, b) \in \text{set } (\text{zip } (\text{old.predecessors } g n') (z :: \text{'val list}))$ 
    with assms(1) have  $n' \in \text{set } (\alpha n g)$  by auto
    thus  $\text{phis } g (n', v') = \text{Some } z \implies (n, b) \in \text{set } (\text{zip } (\text{old.predecessors } g n') z) \implies b \notin \text{phiUses } g n \implies b \in \text{uses } g n$  by (auto intro:phiUsesI)
  qed
  thus ?thesis by (auto simp:step.allUses-def allUses-def zip-map2 intro!:imageI elim!:step.phiUsesE phiUsesE split:if-split-asm)
qed

```

```

lemma allDefs-narrows[simp]:  $v \in \text{step.allDefs } g n \implies v \in \text{allDefs } g n$ 

```

by (auto simp: step.allDefs-def step.phiDefs-def phiDefs-def allDefs-def split:if-split-asm)

**lemma** *allUses-def-ass-inv*:  
 assumes  $v' \in \text{step.allUses } g \ n \ n \in \text{set } (\alpha n \ g)$   
 shows  $\text{step.defAss } g \ n \ v'$   
**proof** (rule *step.defAssI*)  
 fix  $ns$   
 assume  $asm: g \vdash \text{Entry } g - ns \rightarrow n$

from *assms* obtain  $v$  where  $v': v' = \text{substNext } g \ v$  and  $[simp]: v \in \text{allUses } g$   
 $n$

using *allUses-narrows* by auto  
 with *assms*(2) have  $[simp]: v \in \text{allVars } g$  by – (rule *allUses-in-allVars*)  
 have  $[simp]: v' \in \text{allVars } g$  by (simp add: *substNext-def v'*)  
 let  $?n_v = \text{defNode } g \ v$   
 let  $?n_{v'} = \text{defNode } g \ v'$   
 from *assms*(2) *asm* have  $1: ?n_v \in \text{set } ns$  using *allUses-def-ass*[of  $v \ g \ n$ ] by  
 (simp add: *defAss-defNode*)  
 then obtain  $ns_v$  where  $ns_v: \text{prefix } (ns_v @ [?n_v]) \ ns$  by (rule *prefix-split-first*)  
 with *asm* have  $2: g \vdash \text{Entry } g - ns_v @ [?n_v] \rightarrow ?n_v$  by auto  
 show  $\exists n \in \text{set } ns. v' \in \text{step.allDefs } g \ n$   
**proof** (cases  $v = \text{chooseNext } g$ )  
 case *True*  
 hence *dom*: *strict-def-dom*  $g \ v' \ v$  using *substitution*[of  $g$ ] by – (rule  
*trivialPhi-strict-dom*, *simp-all* add: *substNext-def v'*)  
 hence  $[simp]: v' \neq v$  by auto  
 have  $v' \in \text{allDefs } g \ ?n_{v'}$  by *simp*  
 hence  $v' \in \text{step.allDefs } g \ ?n_{v'}$  **unfolding** *step.allDefs-def step.phiDefs-def*  
*allDefs-def phiDefs-def* by (auto simp: *True[symmetric]*)  
 moreover have  $?n_{v'} \in \text{set } ns$   
**proof**–  
 from *dom* have *def-dominates*  $g \ v' \ v$  by auto  
 hence  $?n_{v'} \in \text{set } (ns_v @ [?n_v])$  using  $2$  by – (erule *old.dominatesE*)  
 with  $ns_v$  show *thesis* by auto  
 qed  
 ultimately show *thesis* by auto  
 next  
 case  $[simp]: \text{False}$   
 have  $[simp]: v' = v$  by (simp add: *v' substNext-def*)  
 have  $v \in \text{allDefs } g \ ?n_v$  by *simp*  
 thus *thesis* by – (rule *bexI*[of -  $?n_v$ ], auto simp: *allDefs-def step.allDefs-def*  
*step.phiDefs-def 1 phiDefs-def*)  
 qed  
 qed

**lemma** *Entry-no-phis-inv*:  $\text{phis}' \ g \ (\text{Entry } g, v) = \text{None}$   
 by (simp add: *Entry-no-phis*)

**sublocale** *step*: *CFG-SSA-wf*  $\alpha e \ \alpha n \ \text{invar } \text{inEdges}' \ \text{Entry } \text{defs } u-g \ p-g$

```

apply unfold-locales
apply (rename-tac g' n)
apply (case-tac g'=g)
  apply (simp add:allUses-def-ass-inv)
apply (simp add:allUses-def-ass[unfolded CFG-SSA-defs, simplified] step.CFG-SSA-defs)
apply (rename-tac g' v)
apply (case-tac g'=g)
  apply (simp add:Entry-no-phis-inv)
  apply (simp)
done

```

**lemma** *chooseNext-eliminated*: *chooseNext g*  $\notin$  *step.allDefs g* (*defNode g* (*chooseNext g*))

```

proof–
  let ?v = chooseNext g
  let ?n = defNode g ?v
  from chooseNext[OF redundant] have ?v  $\in$  phiDefs g ?n ?n  $\in$  set ( $\alpha n$  g)
    by (auto simp: trivial-def isTrivialPhi-def phiDefs-def phi-def split: option.splits)
  hence ?v  $\notin$  defs g ?n using simpleDefs-phiDefs-disjoint[of ?n g] by auto
  thus ?thesis by (auto simp:step.allDefs-def step.phiDefs-def)
qed

```

**lemma** *oldUses-inv*:

```

  assumes n  $\in$  set ( $\alpha n$  g)
  shows oldUses g n = var g ' u-g g n
proof–
  have var g (substitution g) = var g (chooseNext g) using substitution[of g]
    by – (rule phiArg-same-var, auto simp: isTrivialPhi-def phiArg-def split: option.splits)
  thus ?thesis using assms by (auto simp: substNext-def oldUses-def image-Un)
qed

```

**lemma** *conventional-inv*:

```

  assumes g  $\vdash$  n–ns→m n  $\notin$  set (tl ns) v  $\in$  step.allDefs g n v  $\in$  step.allUses g
  m x  $\in$  set (tl ns) v'  $\in$  step.allDefs g x
  shows var g v'  $\neq$  var g v
proof–
  from assms(1,3) have[simp]: n = defNode g v v  $\in$  allDefs g n by – (rule defNode-eq[symmetric], auto)
  from assms(1) have[simp]: m  $\in$  set ( $\alpha n$  g) by auto
  from assms(4) obtain v0 where v0: v = substNext g v0 v0  $\in$  allUses g m
using allUses-narrows[of m] by auto
  hence[simp]: v0  $\in$  allVars g using assms(1) by auto
  let ?n0 = defNode g v0
  show ?thesis
proof (cases v0 = chooseNext g)
  case False
    with v0 have v = v0 by (simp add:substNext-def split:if-split-asm)

```

```

with assms  $v_0$  show ?thesis by – (rule conventional, auto)
next
case True
  hence dom: strict-def-dom  $g$   $v$   $v_0$  using substitution[of  $g$ ] by – (rule
trivialPhi-strict-dom, simp-all add:substNext-def  $v_0$ )
  from  $v_0(2)$  have old.dominates  $g$   $?n_0$   $m$  using assms(1) by – (rule
allUses-dominated, auto)
  with assms(1) dom have  $?n_0 \in \text{set } ns$  by – (rule old.dominates-mid, auto)
  with assms(1) obtain  $ns_1$   $ns_3$   $ns_2$  where
     $ns$ :  $ns = ns_1 @ ns_3 @ ns_2$  and
     $ns_1$ :  $g \vdash n - ns_1 @ [?n_0] \rightarrow ?n_0$   $?n_0 \notin \text{set } ns_1$  and
     $ns_3$ :  $g \vdash ?n_0 - ns_3 \rightarrow ?n_0$  and
     $ns_2$ :  $g \vdash ?n_0 - ?n_0 \# ns_2 \rightarrow m$   $?n_0 \notin \text{set } ns_2$  by (rule old.path2-split-first-last)
  have[simp]:  $ns_1 \neq []$ 
  proof
    assume  $ns_1 = []$ 
    hence  $?n_0 = n$  hd  $ns = n$  using assms(1)  $ns_3$  by (rule auto simp:ns old.path2-def)
    thus False by (metis  $\langle n = \text{defNode } g \ v \rangle$  dom)
  qed
  hence length ( $ns_1 @ [?n_0]$ )  $\geq 2$  by (cases  $ns_1$ , auto)
  with  $ns_1$  have 1:  $g \vdash n - ns_1 \rightarrow \text{last } ns_1$   $\text{last } ns_1 \in \text{set } (\text{old.predecessors } g \ ?n_0)$ 
by – (rule old.path2-unsnoc, simp, simp, erule old.path2-unsnoc, auto)
  from  $\langle v_0 = \text{chooseNext } g \rangle$   $v_0$  have triv: isTrivialPhi  $g$   $v_0$   $v$  using substitution
[of  $g$ ] by (rule auto simp:substNext-def)
  then obtain vs where vs:  $\text{phi } g \ v_0 = \text{Some } vs$   $\text{set } vs = \{v_0, v\} \vee \text{set } vs = \{v\}$ 
by (rule auto simp:isTrivialPhi-def split:option.splits)
  hence[simp]:  $\text{var } g \ v_0 = \text{var } g \ v$  by – (rule phiArg-same-var[symmetric], auto
simp:phiArg-def)
  have[simp]:  $v \in \text{phiUses } g \ (\text{last } ns_1)$ 
  proof–
    from vs  $ns_1$  1 have  $v \in \text{phiUses } g \ (\text{last } ns_1) \vee v_0 \in \text{phiUses } g \ (\text{last } ns_1)$ 
by – (rule phiUses-exI[of  $\text{last } ns_1$   $g$   $?n_0$   $v_0$  vs], auto simp:phi-def)
    moreover have  $v_0 \notin \text{phiUses } g \ (\text{last } ns_1)$ 
    proof
      assume asm:  $v_0 \in \text{phiUses } g \ (\text{last } ns_1)$ 
      from True have  $\text{last } ns_1 \in \text{set } ns_1$  by – (rule last-in-set, auto)
      hence  $\text{last } ns_1 \in \text{set } (\alpha n \ g)$  by – (rule old.path2-in-alpha[OF  $ns_1(1)$ ], auto)
      with asm  $ns_1$  have old.dominates  $g$   $?n_0$  ( $\text{last } ns_1$ ) by – (rule allUses-dominated,
auto)
      moreover have strict-def-dom  $g$   $v$   $v_0$  using triv by – (rule trivialPhi-strict-dom,
auto)
      ultimately have  $?n_0 \in \text{set } ns_1$  using 1(1) by – (rule old.dominates-mid,
auto)
      with  $ns_1(2)$  show False ..
    qed
    ultimately show ?thesis by simp
  qed
show ?thesis

```

```

proof (cases x ∈ set (tl ns1))
  case True
    thus ?thesis using assms(2,3,6) by - (rule conventional[where x=x, OF
1(1)], auto simp:ns)
  next
    case False
    show ?thesis
    proof (cases var g v' = var g v0)
      case [simp]: True
        {
          assume asm: x ∈ set ns3
          with assms(6)[THEN allDefs-narrows] have[simp]: x = defNode g v'
            using ns3 by - (rule defNode-eq[symmetric], auto)
          {
            assume v' = v0
            hence False using assms(6) (v0 = chooseNext g) simpleDefs-phiDefs-disjoint[of
x g] vs(1)
              by (auto simp: step.allDefs-def step.phiDefs-def)
          }
          moreover {
            assume v' ≠ v0
            hence x ≠ ?n0 using allDefs-var-disjoint[OF - assms(6)[THEN
allDefs-narrows], of v0]
              by auto
            from ns3 asm ns obtain ns3 where ns3: g ⊢ ?n0-ns3→?n0 ?n0 ∉
set (tl (butlast ns3)) x ∈ set ns3 set ns3 ⊆ set (tl ns)
              by - (rule old.path2-simple-loop, auto)
            with (x ≠ ?n0) have length ns3 > 1
              by (metis empty-iff graph-path-base.path2-def hd-Cons-tl insert-iff
length-greater-0-conv length-tl list.set(1) list.set(2) zero-less-diff)
            with ns3 obtain ns' m where ns': g ⊢ ?n0-ns'→m m ∈ set
(old.predecessors g ?n0) ns' = butlast ns3
              by - (rule old.path2-unsnoc, auto)
            with vs ns3 have v ∈ phiUses g m ∨ v0 ∈ phiUses g m
              by - (rule phiUses-exI[of m g ?n0 v0 vs], auto simp:phi-def)
            moreover {
              assume v ∈ phiUses g m
              have var g v0 ≠ var g v
              proof (rule conventional)
                show g ⊢ n-ns1 @ ns'→m using old.path2-app'[OF ns1(1) ns'(1)]
by simp
              have n ∉ set (tl ns1) using ns assms(2) by auto
              moreover have n ∉ set ns' using ns'(3) ns3(4) assms(2) by (auto
dest: in-set-butlastD)
              ultimately show n ∉ set (tl (ns1 @ ns')) by simp
              show v ∈ allDefs g n using (v ∈ allDefs g n) .
              show ?n0 ∈ set (tl (ns1 @ ns')) using ns'(1) by (auto simp:
old.path2-def)
              qed (auto simp: (v ∈ phiUses g m))
            }
          }
        }

```

```

    hence False by simp
  }
  moreover {
    assume  $v_0 \in \text{phiUses } g \ m$ 
    moreover from  $ns_3(1,3) \langle x \neq ?n_0 \rangle \langle \text{length } ns_3 > 1 \rangle$  have  $x \in \text{set}$ 
    (tl (butlast ns3))
    by (cases ns3, auto simp: old.path2-def intro: in-set-butlastI)
    ultimately have  $\text{var } g \ v' \neq \text{var } g \ v_0$ 
    using assms(6)[THEN allDefs-narrows] ns3(2,3) ns'(3) by - (rule
    conventional[OF ns'(1)], auto)
    hence False by simp
  }
  ultimately have False by auto
}
ultimately have False by auto
}
moreover {
  assume asm: x ∉ set ns3
  have  $\text{var } g \ v' \neq \text{var } g \ v_0$ 
  proof (cases x = ?n0)
    case True
    moreover have  $v_0 \notin \text{step.allDefs } g \ ?n_0$  by (auto simp:⟨v0 = chooseNext
    g⟩ chooseNext-eliminated)
    ultimately show ?thesis using assms(6) vs(1) by - (rule allDefs-var-disjoint[of
    x g], auto)
  next
    case False
    with  $\langle x \notin \text{set } (\text{tl } ns_1) \rangle$  assms(5) asm have  $x \in \text{set } ns_2$  by (auto
    simp:ns)
    thus ?thesis using assms(2,6) v0(2) ns2(2) by - (rule conventional[OF
    ns2(1), where x=x], auto simp:ns)
  qed
}
ultimately show ?thesis by auto
qed auto
qed
qed
qed

```

```

lemma[simp]:  $\text{var } g \ (\text{substNext } g \ v) = \text{var } g \ v$ 
  using substitution[OF redundant]
  by (auto simp:substNext-def isTrivialPhi-def phi-def split:option.splits)

```

```

lemma phis-same-var-inv:
  assumes  $\text{phis}' \ g \ (n, v) = \text{Some } vs \ v' \in \text{set } vs$ 
  shows  $\text{var } g \ v' = \text{var } g \ v$ 
  proof -
    from assms obtain  $vs_0 \ v_0$  where  $1: \text{phis } g \ (n, v) = \text{Some } vs_0 \ v_0 \in \text{set } vs_0 \ v'$ 
    = substNext } g \ v_0 by (auto split:if-split-asm)
  
```



**hence**  $\text{var } g \ v_0 = \text{var } g \ v$  **by** *auto*  
**with 1 show** *?thesis* **by** *auto*  
**qed**

**lemma** *allDefs-var-disjoint-inv*:  $\llbracket n \in \text{set } (\alpha n \ g); v \in \text{step.allDefs } g \ n; v' \in \text{step.allDefs } g \ n; v \neq v' \rrbracket \implies \text{var } g \ v' \neq \text{var } g \ v$   
**using** *allDefs-var-disjoint*  
**by** (*auto simp: step.allDefs-def*)

**lemma** *step-CFG-SSA-Transformed-notriv*: *CFG-SSA-Transformed-notriv*  $\alpha e \ \alpha n$   
*invar inEdges' Entry oldDefs oldUses defs u-g p-g var chooseNext-all*  
**apply** *unfold-locales*  
**apply** (*rule oldDefs-def*)  
**apply** (*rename-tac g'*)  
**apply** (*case-tac g'=g*)  
**apply** (*simp add: oldUses-inv*)  
**apply** (*simp add: oldUses-def*)  
**apply** (*rename-tac g' n ns m v x v'*)  
**apply** (*case-tac g'=g*)  
**apply** (*simp add: conventional-inv*)  
**apply** (*simp add: conventional[unfolded CFG-SSA-defs, simplified] step.CFG-SSA-defs*)  
**apply** (*rename-tac g' n v vs v'*)  
**apply** (*case-tac g'=g*)  
**apply** (*simp add: phis-same-var-inv*)  
**apply** (*simp add: phis-same-var*)  
**apply** (*rename-tac g' v v'*)  
**apply** (*case-tac g'=g*)  
**apply** (*simp add: allDefs-var-disjoint-inv*)  
**apply** (*simp add: allDefs-var-disjoint[unfolded allDefs-def phiDefs-def, simplified]*)  
*step.allDefs-def step.phiDefs-def*  
**by** (*rule chooseNext-all*)

**sublocale** *step*: *CFG-SSA-Transformed-notriv*  $\alpha e \ \alpha n$  *invar inEdges' Entry old-*  
*Defs oldUses defs u-g p-g var chooseNext-all*  
**by** (*rule step-CFG-SSA-Transformed-notriv*)

**lemma** *step-defNode*:  $v \in \text{allVars } g \implies v \neq \text{chooseNext } g \implies \text{step.defNode } g \ v$   
 $= \text{defNode } g \ v$   
**by** (*auto simp: step.CFG-SSA-wf-defs dom-def CFG-SSA-wf-defs*)

**lemma** *step-phi*:  $v \in \text{allVars } g \implies v \neq \text{chooseNext } g \implies \text{step.phi } g \ v =$   
 $\text{map-option } (\text{map } (\text{substNext } g)) \ (\text{phi } g \ v)$   
**by** (*auto simp: step.phi-def step-defNode phi-def*)

**lemma** *liveVal'-inv*:  
**assumes** *liveVal' g (v#vs) v*  $\neq \text{chooseNext } g$   
**obtains** *vs'* **where** *step.liveVal' g (v#vs')*  
**using** *assms* **proof** (*induction length vs arbitrary: v vs rule: nat-less-induct*)  
**case** (*1 vs v*)

```

from 1.prems(2) show thesis
proof cases
  case (liveSimple' n)
    with 1.prems(3) show thesis by - (rule 1.prems(1), rule step.liveSimple',
    auto simp: substNext-def)
  next
    case (livePhi' v' vs')
      from this(2) have[simp]: v' ∈ allVars g by - (drule liveVal'D, rule, rule
      liveVal-in-allVars)
      show thesis
      proof (cases chooseNext g = v')
        case False
          show thesis
          proof (rule 1.hyps[rule-format, of length vs' vs' v'])
            fix vs'_2
            assume asm: step.liveVal' g (v'#vs'_2)
            have step.phiArg g v' v using livePhi'(3) False 1.prems(3) by (auto simp:
            step.phiArg-def phiArg-def step-phi substNext-def)
            thus thesis by - (rule 1.prems(1), rule step.livePhi', rule asm)
            qed (auto simp: livePhi' False[symmetric])
        next
          case [simp]: True
            with 1.prems(3) have[simp]: v ≠ v' by simp
            from True have trivial g v' using chooseNext[OF redundant] by auto
            with ⟨phiArg g v' v⟩ have isTrivialPhi g v' v by (auto simp: phiArg-def
            trivial-def isTrivialPhi-def)
            hence[simp]: substitution g = v unfolding substitution-def
            by - (rule the1-equality, auto intro!: isTrivialPhi-det[unfolded trivial-def])

            obtain vs'_2 where vs'_2: suffix (v'#vs'_2) (v'#vs') v' ∉ set vs'_2
            using split-list-last[of v' v'#vs'] by (auto simp: Sublist.suffix-def)
            with ⟨liveVal' g (v'#vs')⟩ have liveVal' g (v'#vs'_2) by - (rule liveVal'-suffix,
            simp-all)
            thus thesis
            proof (cases rule: liveVal'.cases)
              case (liveSimple' n)
                hence v ∈ uses' g n by (auto simp: substNext-def)
                with liveSimple' show thesis by - (rule 1.prems(1), rule step.liveSimple',
                auto)
              next
                case (livePhi' v'' vs'')
                  from this(2) have[simp]: v'' ∈ allVars g by - (drule liveVal'D, rule, rule
                  liveVal-in-allVars)
                  from vs'_2(2) livePhi'(1) have[simp]: v'' ≠ v' by auto
                  show thesis
                  proof (rule 1.hyps[rule-format, of length vs'' vs'' v'])
                    show length vs'' < length vs using ⟨vs = v'#vs'⟩ livePhi'(1) vs'_2(1)[THEN
                    suffix-ConsD2]
                    by (auto simp: Sublist.suffix-def)

```

```

next
  fix vs''2
  assume asm: step.liveVal' g (v''#vs''2)
  from livePhi' ⟨phiArg g v' v⟩ have step.phiArg g v'' v by (auto simp:
phiArg-def step.phiArg-def step-phi substNext-def)
  thus thesis by - (rule 1.premis(1), rule step.livePhi', rule asm)
qed (auto simp: livePhi'(2))
qed
qed
qed
qed

```

**lemma** *liveVal-inv*:

```

  assumes liveVal g v v ≠ chooseNext g
  shows step.liveVal g v
  apply (rule liveVal'I[OF assms(1)])
  apply (erule liveVal'-inv[OF - assms(2)])
  apply (rule step.liveVal'D)
  by simp-all

```

**lemma** *pruned-inv*:

```

  assumes pruned g
  shows step.pruned g
  proof (rule step.pruned-def[THEN iffD2, rule-format])
    fix n v
    assume v ∈ step.phiDefs g n and[simp]: n ∈ set (αn g)
    hence v ∈ phiDefs g n v ≠ chooseNext g by (auto simp: step.CFG-SSA-defs
CFG-SSA-defs split: if-split-asm)
    hence liveVal g v using assms by (auto simp: pruned-def)
    thus step.liveVal g v using ⟨v ≠ chooseNext g⟩ by (rule liveVal-inv)
  qed
end

```

**context** *CFG-SSA-Transformed-notriv-base*

**begin**

```

  abbreviation inst g u p ≡ CFG-SSA-Transformed-notriv αe αn invar inEdges'
Entry oldDefs oldUses defs (uses(g:=u)) (phis(g:=p)) var chooseNext-all
  abbreviation inst' g ≡ λ(u,p). inst g u p

```

**interpretation** *uninst*: *CFG-SSA-Transformed-notriv-base* αe αn invar inEdges'  
Entry oldDefs oldUses defs u p var chooseNext-all

```

  for u and p
  by unfold-locales

```

**definition** *cond* g ≡ λ(u,p). *uninst.redundant* (uses(g:=u)) (phis(g:=p)) g

**definition** *step* g ≡ λ(u,p). (*uninst.uses'* (uses(g:=u)) (phis(g:=p)) g,  
*uninst.phis'* (uses(g:=u)) (phis(g:=p)) g)

**definition**[code]: *substAll* g ≡ *while* (cond g) (step g) (uses g,phis g)

```

definition[code]: uses'-all g ≡ fst (substAll g)
definition[code]: phis'-all g ≡ snd (substAll g)

lemma uninstant-allVars-simps [simp]:
  uninstant.allVars u (λ-. p g) g = uninstant.allVars u p g
  uninstant.allVars (λ-. u g) p g = uninstant.allVars u p g
  uninstant.allVars (uses(g:=u g)) p g = uninstant.allVars u p g
  uninstant.allVars u (phis(g:=p g)) g = uninstant.allVars u p g
  unfolding uninstant.allVars-def uninstant.allDefs-def uninstant.allUses-def uninstant.phisDefs-def
  uninstant.phisUses-def
  by simp-all

lemma uninstant-trivial-simps [simp]:
  uninstant.trivial u (λ-. p g) g = uninstant.trivial u p g
  uninstant.trivial (λ-. u g) p g = uninstant.trivial u p g
  uninstant.trivial (uses(g:=u g)) p g = uninstant.trivial u p g
  uninstant.trivial u (phis(g:=p g)) g = uninstant.trivial u p g
  unfolding uninstant.trivial-def [abs-def] uninstant.isTrivialPhi-def uninstant.phi-def
  uninstant.defNode-code
  uninstant.allDefs-def uninstant.phisDefs-def
  by simp-all

end

context CFG-SSA-Transformed-notriv
begin
  declare fun-upd-apply[simp del] fun-upd-same[simp]

  lemma substAll-wf:
    assumes[simp]: redundant g
    shows card (dom (phis' g)) < card (dom (phis g))
    proof (rule psubset-card-mono)
      let ?v = chooseNext g
      from chooseNext[of g] obtain n where (n,?v) ∈ dom (phis g) by (auto simp:
  trivial-def isTrivialPhi-def phi-def split:option.splits)
      moreover have (n,?v) ∉ dom (phis' g) by auto
      ultimately have dom (phis' g) ≠ dom (phis g) by auto
      thus dom (phis' g) ⊂ dom (phis g) by (auto split:if-split-asm)
    qed (rule phis-finite)

  lemma step-preserves-inst:
    assumes inst' g (u,p)
    and CFG-SSA-wf-base.redundant αn inEdges' defs (uses(g:=u)) (phis(g:=p))
  g
    shows inst' g (step g (u,p))
    proof –
      from assms(1) interpret i: CFG-SSA-Transformed-notriv αe αn invar in-
  Edges' Entry oldDefs oldUses defs uses(g:=u) phis(g:=p) var

```

```

by simp

from assms(2) interpret step: CFG-SSA-step  $\alpha e \alpha n$  invar inEdges' Entry
oldDefs oldUses defs uses( $g:=u$ ) phis( $g:=p$ ) var chooseNext-all
by unfold-locates

show ?thesis using step.step-CFG-SSA-Transformed-notriv[simplified] by (simp
add: step-def)
qed

lemma substAll:
  assumes  $P$  (uses  $g$ , phis  $g$ )
  assumes  $\bigwedge x. P\ x \implies inst'\ g\ x \implies cond\ g\ x \implies P\ (step\ g\ x)$ 
  assumes  $\bigwedge x. P\ x \implies inst'\ g\ x \implies \neg cond\ g\ x \implies Q\ (fst\ x)\ (snd\ x)$ 
  shows  $inst\ g\ (uses'-all\ g)\ (phis'-all\ g)\ Q\ (uses'-all\ g)\ (phis'-all\ g)$ 
proof-
  note uses'-def[simp del]
  note phis'-def[simp del]
  have 2:  $\bigwedge f\ x. f\ x = f\ (fst\ x, snd\ x)$  by simp

  have  $inst'\ g\ (substAll\ g) \wedge Q\ (uses'-all\ g)\ (phis'-all\ g)$  unfolding substAll-def
uses'-all-def phis'-all-def
  apply (rule while-rule[where  $P=\lambda x. inst'\ g\ x \wedge P\ x$ ])
    apply (rule conjI)
    apply (simp, unfold-locates)
    apply (rule assms(1))
    apply (rule conjI)
    apply (clarsimp simp: cond-def step-def)
    apply (rule step-preserves-inst [unfolded step-def, simplified], assumption+)
  proof-
    show wf  $\{(y,x). (inst'\ g\ x \wedge cond\ g\ x) \wedge y = step\ g\ x\}$ 
    apply (rule wf-if-measure[where  $f=\lambda(u,p). card\ (dom\ p)$ ])
    apply (clarsimp simp: cond-def step-def split:prod.split)
  proof-
    fix  $u\ p$ 
    assume  $inst\ g\ u\ p$ 
    then interpret  $i$ : CFG-SSA-Transformed-notriv  $\alpha e \alpha n$  invar inEdges'
Entry oldDefs oldUses defs uses( $g:=u$ ) phis( $g:=p$ ) by simp
    assume  $i.redundant\ g$ 
    thus  $card\ (dom\ (i.phis'\ g)) < card\ (dom\ p)$  by (rule  $i.substAll-wf$ [of  $g$ ,
simplified])
    qed
  qed (auto intro: assms(2,3))
  thus  $inst\ g\ (uses'-all\ g)\ (phis'-all\ g)\ Q\ (uses'-all\ g)\ (phis'-all\ g)$ 
  by (auto simp: uses'-all-def phis'-all-def)
qed

```

**sublocale** *notriv*: CFG-SSA-Transformed  $\alpha e \alpha n$  *invar* *inEdges'* *Entry* *oldDefs*

*oldUses defs uses'-all phis'-all*  
**proof**–  
**interpret** *ssa: CFG-SSA  $\alpha e \alpha n$  invar inEdges' Entry defs uses'-all phis'-all*  
**proof**  
**fix** *g*  
**interpret** *i: CFG-SSA-Transformed-notriv  $\alpha e \alpha n$  invar inEdges' Entry old-Defs oldUses defs uses( $g:=uses'-all g$ ) phis( $g:=phis'-all g$ ) var*  
**by** (*rule substAll, auto*)  
**interpret** *uninst: CFG-SSA-Transformed-notriv-base  $\alpha e \alpha n$  invar inEdges' Entry oldDefs oldUses defs u p var chooseNext-all*  
**for** *u and p*  
**by** *unfold-locales*  
  
**fix** *n v args m*  
**show** *finite (defs g n) by (rule defs-finite)*  
**show**  *$v \in uses'-all g n \implies n \in set (\alpha n g)$  by (rule i.uses-in- $\alpha n$ [of - g, simplified])*  
**show** *finite (uses'-all g n) by (rule i.uses-finite[of g, simplified])*  
**show** *invar g by (rule invar)*  
**show** *finite (dom (phis'-all g)) by (rule i.phis-finite[of g, simplified])*  
**show**  *$phis'-all g (n, v) = Some args \implies n \in set (\alpha n g)$  using i.phis-in- $\alpha n$ [of g] by simp*  
**show**  *$phis'-all g (n, v) = Some args \implies length (old.predecessors g n) = length args$  using i.phis-wf[of g] by simp*  
**show**  *$n \in set (\alpha n g) \implies defs g n \cap uninst.phiDefs phis'-all g n = \{\}$  using i.simpleDefs-phiDefs-disjoint[of n g] by (simp add: uninst.CFG-SSA-defs)*  
**show**  *$n \in set (\alpha n g) \implies m \in set (\alpha n g) \implies n \neq m \implies uninst.allDefs phis'-all g n \cap uninst.allDefs phis'-all g m = \{\}$*   
**using** *i.allDefs-disjoint[of n g] by (simp add: uninst.CFG-SSA-defs)*  
**show**  *$n \in set (\alpha n g) \implies defs g n \cap uses'-all g n = \{\}$  using i.defs-uses-disjoint[of n g] by simp*  
**qed**  
**interpret** *uninst: CFG-SSA-Transformed-notriv-base  $\alpha e \alpha n$  invar inEdges' Entry oldDefs oldUses defs u p var chooseNext-all*  
**for** *u and p*  
**by** *unfold-locales*  
  
**show** *CFG-SSA-Transformed  $\alpha e \alpha n$  invar inEdges' Entry oldDefs oldUses defs uses'-all phis'-all var*  
**proof**  
**fix** *g n v ns m x v' vs*  
**interpret** *i: CFG-SSA-Transformed-notriv  $\alpha e \alpha n$  invar inEdges' Entry old-Defs oldUses defs uses( $g:=uses'-all g$ ) phis( $g:=phis'-all g$ ) var*  
**by** (*rule substAll, auto*)  
**show** *oldDefs g n = var g ' defs g n by (rule oldDefs-def)*  
**show**  *$n \in set (\alpha n g) \implies oldUses g n = var g ' uses'-all g n$  using i.oldUses-def[of n g] by simp*  
**show**  *$v \in ssa.allUses g n \implies n \in set (\alpha n g) \implies ssa.defAss g n v$  using i.allUses-def-ass[of v g n] by (simp add: uninst.CFG-SSA-defs)*

**show** *old.path2*  $g\ n\ ns\ m \implies n \notin \text{set } (tl\ ns) \implies v \in \text{ssa.allDefs } g\ n \implies v \in \text{ssa.allUses } g\ m \implies x \in \text{set } (tl\ ns) \implies v' \in \text{ssa.allDefs } g\ x \implies \text{var } g\ v' \neq \text{var } g\ v$  **using** *i.conventional*[of  $g\ n\ ns\ m\ v\ x\ v'$ ] **by** (*simp add: uninstantiated.CFG-SSA-defs*)  
**show** *phis'-all*  $g\ (n, v) = \text{Some } vs \implies v' \in \text{set } vs \implies \text{var } g\ v' = \text{var } g\ v$   
**using** *i.phis-same-var*[of  $g\ n\ v$ ] **by** *simp*  
**show**  $n \in \text{set } (\alpha n\ g) \implies v \in \text{ssa.allDefs } g\ n \implies v' \in \text{ssa.allDefs } g\ n \implies v \neq v' \implies \text{var } g\ v' \neq \text{var } g\ v$  **using** *i.allDefs-var-disjoint* **by** (*simp add: uninstantiated.CFG-SSA-defs*)  
**show** *phis'-all*  $g\ (\text{Entry } g, v) = \text{None}$  **using** *i.Entry-no-phis*[of  $g\ v$ ] **by** *simp*  
**qed**  
**qed**

**theorem** *not-redundant*:  $\neg \text{notriv.redundant } g$   
**proof**–

**interpret** *uninst*: *CFG-SSA-Transformed-notriv-base*  $\alpha e\ \alpha n\ \text{invar } \text{inEdges}'$   
*Entry oldDefs oldUses defs u p var chooseNext-all*  
**for**  $u$  **and**  $p$   
**by** *unfold-locales*

**have**  $1: \bigwedge u\ p. \text{uninst.redundant } (\text{uses}(g:=u\ g))\ (\text{phis}(g:=p\ g))\ g \longleftrightarrow \text{uninst.redundant } u\ p\ g$   
**by** (*simp add: uninstantiated.CFG-SSA-wf-defs*)  
**show** *?thesis*  
**by** (*rule substAll*(2)[**where**  $Q = \lambda u\ p. \neg \text{uninst.redundant } (\text{uses}(g:=u))\ (\text{phis}(g:=p))$ ]  
 $g$  **and**  $P = \lambda \cdot. \text{True}$  **and**  $g = g$ , *simplified cond-def substAll-def 1*], *auto*)  
**qed**

**corollary** *minimal*:  $\text{old.reducible } g \implies \text{notriv.cytronMinimal } g$   
**by** (*erule notriv.reducible-nonredundant-imp-minimal, rule not-redundant*)

**theorem** *pruned-invariant*:

**assumes** *pruned*  $g$   
**shows** *notriv.pruned*  $g$

**proof**–

**{**

**fix**  $u\ p$

**assume** *inst*  $g\ u\ p$

**then interpret** *i*: *CFG-SSA-Transformed-notriv*  $\alpha e\ \alpha n\ \text{invar } \text{inEdges}'$  *Entry oldDefs oldUses defs uses(g:=u) phis(g:=p) var chooseNext-all*  
**by** *simp*

**assume** *i.redundant*  $g$

**then interpret** *i*: *CFG-SSA-step*  $\alpha e\ \alpha n\ \text{invar } \text{inEdges}'$  *Entry oldDefs oldUses defs uses(g:=u) phis(g:=p) var chooseNext-all*  $g$   
**by** *unfold-locales*

**interpret** *uninst*: *CFG-SSA-Transformed-notriv-base*  $\alpha e\ \alpha n\ \text{invar } \text{inEdges}'$   
*Entry oldDefs oldUses defs u p var chooseNext-all*  
**for**  $u$  **and**  $p$

```

    by unfold-locales

    assume i.pruned g
    hence uninst.pruned (uses(g:=i.uses' g)) (phis(g:=i.phis' g)) g
      by (rule i.pruned-inv[simplified])
  }
  note 1 = this

  interpret uninst: CFG-SSA-Transformed-notriv-base  $\alpha e \alpha n$  invar inEdges'
  Entry oldDefs oldUses defs u p var chooseNext-all
  for u and p
  by unfold-locales

  have 2:  $\bigwedge u u' p p' g. uninst.pruned (u'(g:=u g)) (p'(g:=p g)) g \longleftrightarrow uninst.pruned$ 
u p g
    by (clarsimp simp: uninst.CFG-SSA-wf-defs)

  from 1 assms show ?thesis
  by - (rule substAll(2)[where P= $\lambda(u,p). uninst.pruned (uses(g:=u)) (phis(g:=p))$ 
g and Q= $\lambda u p. uninst.pruned (uses(g:=u)) (phis(g:=p)) g$  and  $g=g$ , simplified
2],
      auto simp: cond-def step-def)
  qed
end

end

```

## 5 Proof of Semantic Equivalence

**theory** *SSA-Semantics* **imports** *Construct-SSA* **begin**

**type-synonym** (*'node, 'var*) *state* = *'var  $\rightarrow$  'node*

**context** *CFG-SSA-Transformed*

**begin**

**declare** *invar[intro!]*

**definition** *step* ::

*'g  $\Rightarrow$  'node  $\Rightarrow$  ('node, 'var) state  $\Rightarrow$  ('node, 'var) state*

**where**

*step g m s v  $\equiv$  if  $v \in oldDefs g m$  then *Some m* else *s v**

**inductive** *bs* :: *'g  $\Rightarrow$  'node list  $\Rightarrow$  ('node, 'var) state  $\Rightarrow$  bool* (*-  $\vdash$  - $\Downarrow$ - [50, 50,*

*50] 50)*

**where**

*g  $\vdash$  Entry g-ns $\rightarrow$ last ns  $\implies$  g  $\vdash$  ns $\Downarrow$ (fold (step g) ns Map.empty)*

**definition** *ssaStep* ::



$'g \Rightarrow 'node \Rightarrow nat \Rightarrow ('node, 'val) state \Rightarrow ('node, 'val) state$   
**where**  
 $ssaStep\ g\ m\ i\ s\ v \equiv$   
 $\text{if } v \in \text{defs } g\ m \text{ then}$   
 $\text{Some } m$   
 $\text{else}$   
 $\text{case } \text{phis } g\ (m, v) \text{ of}$   
 $\text{Some } \text{phiParams} \Rightarrow s\ (\text{phiParams } !\ i)$   
 $| \text{None} \Rightarrow s\ v$

**inductive**  $ssaBS :: 'g \Rightarrow 'node\ list \Rightarrow ('node, 'val) state \Rightarrow bool$  ( $-\vdash -\Downarrow_s - [50, 50, 50] 50$ )  
**for**  
 $g :: 'g$   
**where**  
 $\text{empty: } g \vdash [Entry\ g] \Downarrow_s (ssaStep\ g\ (Entry\ g)\ 0\ Map.empty)$   
 $| \text{snoc: } \llbracket g \vdash ns \Downarrow_s s; \text{last } ns = \text{old.predecessors } g\ m\ !\ i; m \in \text{set } (\alpha n\ g); i < \text{length}$   
 $(\text{old.predecessors } g\ m) \rrbracket \Longrightarrow$   
 $g \vdash (ns @ [m]) \Downarrow_s (ssaStep\ g\ m\ i\ s)$

**lemma**  $ssaBS-I$ :  
**assumes**  $g \vdash Entry\ g - ns \rightarrow n$   
**obtains**  $s$  **where**  $g \vdash ns \Downarrow_s s$   
**using**  $assms$   
**proof** ( $\text{atomize-elim, induction rule: old.path2-rev-induct}$ )  
**case** ( $\text{snoc } ns\ m'\ m$ )  
**then obtain**  $s$  **where**  $s: g \vdash ns \Downarrow_s s$  **by**  $\text{auto}$   
**from**  $\text{snoc.hyps}(2)$  **obtain**  $i$  **where**  $m' = \text{old.predecessors } g\ m\ !\ i$   $i < \text{length}$   
 $(\text{old.predecessors } g\ m)$  **by** ( $\text{auto simp: in-set-conv-nth}$ )  
**with**  $\text{snoc.hyps } \text{snoc.prem } s$  **show**  $?case$  **by**  $-(\text{rule } \text{exI, erule } ssaBS.\text{snoc, auto}$   
 $\text{dest: old.path2-last})$   
**qed** ( $\text{auto intro: } ssaBS.\text{empty}$ )

**lemma**  $ssaBS\text{-nonempty}[simp]$ :  $\neg (g \vdash [] \Downarrow_s s)$   
**by** ( $\text{rule } \text{notI, cases rule: } ssaBS.\text{cases, auto}$ )

**lemma**  $ssaBS\text{-hd}[simp]$ :  $g \vdash ns \Downarrow_s s \Longrightarrow \text{hd } ns = Entry\ g$   
**by** ( $\text{induction rule: } ssaBS.\text{induct, auto simp: hd-append}$ )

**lemma**  $\text{equiv-aux}$ :  
**assumes**  $g \vdash ns \Downarrow_s g \vdash ns \Downarrow_s s' \vdash \text{last } ns - ms \rightarrow m\ v \in \text{allUses } g\ m\ \forall n \in \text{set}$   
 $(\text{tl } ms). \text{var } g\ v \notin \text{var } g\ ' \text{allDefs } g\ n$   
**shows**  $s(\text{var } g\ v) = s' v$   
**using**  $assms(2)$   $assms(1, 3-)$  **proof** ( $\text{induction arbitrary: } v\ s\ ms\ m$ )  
**case**  $\text{empty}$   
**have**  $v \in \text{defs } g\ (Entry\ g)$   
**proof**—  
**from**  $\text{empty.prem}(2, 3)$  **have**  $\text{defAss } g\ m\ v$  **by**  $-(\text{rule } \text{allUses-def-ass, auto})$

```

with empty.prems(2) obtain  $n$  where  $n: n \in \text{set } ms \ v \in \text{allDefs } g \ n$  by –
(drule defAssD, auto)
with empty.prems(4) have  $n \notin \text{set } (tl \ ms)$  by auto
with empty.prems(2)  $n$  have  $n = \text{Entry } g$  by (cases ms, auto dest: old.path2-hd)
with  $n(2)$  show ?thesis by (auto simp: allDefs-def)
qed
with empty.prems(1) show ?case
by – (erule bs.cases, auto simp: step-def ssaStep-def oldDefs-def split: option.split)
next
case (snoc ns s' n i)
from snoc.prems(2) have[simp]:  $n \in \text{set } (\alpha n \ g) \ m \in \text{set } (\alpha n \ g)$  by auto
from snoc.prems(2,3) have[simp]:  $v \in \text{allVars } g$  by – (rule allUses-in-allVars,
auto)
from snoc.hyps(4) have[simp]:  $n \neq \text{Entry } g$  by (auto simp: Entry-no-predecessor)

show ?case
proof (cases var g v \in var g ' allDefs g n)
case True

have[simp]: defNode g v = n (is ?nv = -)
proof–
from True obtain  $v'$  where  $v': v' \in \text{allDefs } g \ n \ \text{var } g \ v' = \text{var } g \ v$  by auto
from snoc.prems(3) have defAss g m v by – (rule allUses-def-ass, auto)
moreover from snoc.prems(1) obtain  $ns'$  where  $ns': g \vdash \text{Entry } g - ns' \rightarrow n$ 
set ns' \subseteq set (ns@[n]) distinct ns'
by (auto elim!: bs.cases intro: old.simple-path2)
ultimately have  $?n_v \in \text{set } (ns' @ tl \ ms)$ 
using snoc.prems(2) by – (drule defAss-defNode, auto elim!: bs.cases
dest: old.path2-app)
moreover {
let  $?n'' = \text{last } (butlast \ ns')$ 
assume asm: ?nv \in set (butlast ns')
with  $ns'(1,3)$  have[simp]:  $?n_v \neq n$  by (cases ns' rule: rev-cases, auto
dest!: old.path2-last)
from  $ns'(1)$  have  $\text{length } ns' \geq 2$  by auto
with  $ns'$  have  $bns': g \vdash \text{Entry } g - butlast \ ns' \rightarrow ?n'' \ ?n'' \in \text{set } (\text{old.predecessors}$ 
g n)
by (auto elim: old.path2-unsnoc)
with asm obtain  $ns''$  where  $ns'': g \vdash ?n_v - ns'' \rightarrow ?n''$  suffix ns'' (butlast
ns') ?nv \notin set (tl ns'')
by – (rule old.path2-split-first-last, auto)
with  $bns' \ \text{snoc.prem$ s(2) have  $g \vdash ?n_v - (ns''@[n]) @ tl \ ms \rightarrow m$  by – (rule
old.path2-app, auto)
hence defNode g v' \notin set (tl (ns''@[n]) @ tl ms)
using  $v' \ \text{snoc.prem$ s(3,4)  $bns'(2) \ ns''(1,3)$ 
by – (rule conventional''[of g v - m], auto intro!: old.path2-app simp:
old.path2-not-Nil)
with  $ns' \ ns''(1) \ v'(1)$  have False by (auto simp: old.path2-not-Nil)

```

```

    }
    ultimately show ?thesis using snoc.premns(4) ns'(1) by (cases ns' rule:
rev-cases, auto dest: old.path2-last)
  qed
  from ⟨v ∈ allVars g⟩ show ?thesis
  proof (cases rule: defNode-cases)
    case simpleDef
    thus ?thesis using snoc.premns(1) by - (erule bs.cases, auto simp: step-def
ssaStep-def oldDefs-def)
  next
    case phi
    {
      fix v'
      assume asm: v' ∈ defs g n var g v = var g v'
      with phi have v' = v using allDefs-var-disjoint[of n g v' v]
        by (cases, auto dest!: phi-phiDefs)
      with asm(1) phi have False using simpleDefs-phiDefs-disjoint[of n g]
        by (auto dest!: phi-phiDefs)
    }
    note 1 = this
    {
      fix vs
      assume asm: g ⊢ Entry g-ns @ [n]→n phis g (n, v) = Some vs var g v
∉ var g ' defs g n
      let ?n' = last ns
      from asm(1) have length ns ≥ 1 by (cases ns, auto simp: old.path2-def)
      hence g ⊢ Entry g-ns→?n'
        by - (rule old.path2-unsnoc[OF asm(1)], auto)
      moreover have vs ! i ∈ phiUses g ?n' using snoc.hyps(2,4) phis-wf[OF
asm(2)]
        by - (rule phiUsesI[OF - asm(2)], auto simp: set-zip)
      ultimately have fold (step g) ns Map.empty (var g (vs ! i)) = s' (vs ! i)
      by - (rule snoc.IH[where ms1=[?n'] and m1=?n'], auto intro!: bs.intros)
      hence fold (step g) ns Map.empty (var g v) = s' (vs ! i) using
phis-same-var[OF asm(2), of vs ! i] snoc.hyps(4) phis-wf[OF asm(2)]
        by auto
    }
    thus ?thesis using phi snoc.premns(1)
      by - (erule bs.cases, auto dest!: 1 simp: step-def ssaStep-def oldDefs-def
phi-def)
  qed
  next
  case False
  hence phis g (n, v) = None by (auto simp: allDefs-def phiDefs-def)
  moreover have fold (step g) ns Map.empty (var g v) = s' v
  proof-
    from snoc.hyps(1) have length ns ≥ 1 by (cases ns, auto)
    moreover from snoc.premns(2,4) False have ∀ n ∈ set ms. var g v ∉ var
g ' allDefs g n

```

```

    by (cases ms, auto simp: phiDefs-def dest: old.path2-hd)
  ultimately show ?thesis
    using snoc.prem1(1,2,3) by - (rule snoc.IH[where ms1=last ns#ms],
  auto elim!: bs.cases intro!: bs.intros elim: old.path2-unsnoc intro!: old.Cons-path2)
  qed
  ultimately show ?thesis
    using snoc.prem1(1) False by - (erule bs.cases, auto simp: step-def
  ssaStep-def oldDefs-def)
  qed
  qed

theorem equiv:
  assumes  $g \vdash ns \Downarrow s$   $g \vdash ns \Downarrow_s s' v \in \text{uses } g$  (last ns)
  shows  $s \text{ (var } g \text{ v)} = s' v$ 
  using assms by - (rule equiv-aux[where ms=[last ns]], auto elim!: bs.cases)
end

end

```

## 6 Code Generation

### 6.1 While Combinator Extensions

```

theory While-Combinator-Exts imports
  HOL-Library.While-Combinator
begin
lemma while-option-None-invD:
  assumes while-option b c s = None and wf r
  and I s and  $\bigwedge s. \llbracket I s; b s \rrbracket \implies I (c s)$ 
  and  $\bigwedge s. \llbracket I s; b s \rrbracket \implies (c s, s) \in r$ 
  shows False
  using assms
  by - (drule wf-rel-while-option-Some [of r I b c], auto)

lemma while-option-NoneD:
  assumes while-option b c s = None
  and wf r and  $\bigwedge s. b s \implies (c s, s) \in r$ 
  shows False
  using assms
  by (blast intro: while-option-None-invD)

lemma while-option-sim:
  assumes start:  $R$  (Some s1) (Some s2)
  and cond:  $\bigwedge s1 s2. \llbracket R$  (Some s1) (Some s2); I s1  $\rrbracket \implies b1 s1 = b2 s2$ 
  and step :  $\bigwedge s1 s2. \llbracket R$  (Some s1) (Some s2); I s1; b1 s1  $\rrbracket \implies R$  (Some (c1
  s1)) (Some (c2 s2))
  and diverge:  $R$  None None
  and inv-start: I s1
  and inv-step:  $\bigwedge s1. \llbracket I s1; b1 s1 \rrbracket \implies I (c1 s1)$ 

```

```

shows  $R$  (while-option  $b1$   $c1$   $s1$ ) (while-option  $b2$   $c2$   $s2$ )
proof -
{ fix  $k$ 
  assume  $\forall k' < k. b1 ((c1 \wedge\wedge k') s1)$ 
  with start cond step inv-start inv-step
  have  $b1 ((c1 \wedge\wedge k) s1) = b2 ((c2 \wedge\wedge k) s2)$  and  $I ((c1 \wedge\wedge k) s1)$ 
    and  $R (Some ((c1 \wedge\wedge k) s1)) (Some ((c2 \wedge\wedge k) s2))$ 
    by (induction  $k$ ) auto
}
moreover
{ fix  $k$ 
  assume  $\neg b1 ((c1 \wedge\wedge k) s1)$ 
  hence  $\forall k' < LEAST\ k. \neg b1 ((c1 \wedge\wedge k) s1). b1 ((c1 \wedge\wedge k') s1)$ 
    by (metis (lifting) not-less-Least)
}
moreover
{ fix  $k$ 
  assume  $\neg b2 ((c2 \wedge\wedge k) s2)$ 
  hence  $\forall k' < LEAST\ k. \neg b2 ((c2 \wedge\wedge k) s2). b2 ((c2 \wedge\wedge k') s2)$ 
    by (metis (lifting) not-less-Least)
}
moreover
{
  assume  $\exists k. \neg b1 ((c1 \wedge\wedge k) s1)$ 
    and  $\exists k. \neg b2 ((c2 \wedge\wedge k) s2)$ 
  hence not-cond-Least:  $\neg b1 ((c1 \wedge\wedge (LEAST\ k. \neg b1 ((c1 \wedge\wedge k) s1))) s1)$ 
     $\neg b2 ((c2 \wedge\wedge (LEAST\ k. \neg b2 ((c2 \wedge\wedge k) s2))) s2)$ 
    by  $\neg$ (drule LeastI-ex, assumption) $+$ 
  { fix  $k$ 
    assume  $\forall k' < k. b1 ((c1 \wedge\wedge k') s1)$ 
    with calculation(1) dual-order.strict-trans
    have  $\forall k' < k. b2 ((c2 \wedge\wedge k') s2)$ 
      by blast
  }
  hence  $(LEAST\ k'. \neg b1 ((c1 \wedge\wedge k') s1)) = (LEAST\ k'. \neg b2 ((c2 \wedge\wedge k') s2))$ 
    by (metis (no-types, lifting) not-cond-Least calculation(1,4,5) less-linear)
  with calculation(3,4)
  have  $R (Some ((c1 \wedge\wedge (LEAST\ k. \neg b1 ((c1 \wedge\wedge k) s1))) s1))$ 
    ( $Some ((c2 \wedge\wedge (LEAST\ k. \neg b2 ((c2 \wedge\wedge k) s2))) s2)$ )
    by auto
}
ultimately show ?thesis using diverge
  unfolding while-option-def
  apply (split if-split)
  apply (rule conjI)
  apply (split if-split)
  apply metis
  apply (split if-split)
  by (metis (lifting) LeastI-ex)

```

qed

end

**theory** *SSA-CFG-code* **imports**  
  *SSA-CFG*  
  *Mapping-Exts*  
  *HOL-Library.Product-Lexorder*  
**begin**

**definition** *Union-of* :: ('a  $\Rightarrow$  'b set)  $\Rightarrow$  'a set  $\Rightarrow$  'b set  
  **where** *Union-of* f A  $\equiv$   $\bigcup$ (f ` A)

**lemma** *Union-of-alt-def*: *Union-of* f A = ( $\bigcup$  x  $\in$  A. f x)  
  **unfolding** *Union-of-def* **by** *simp*

**type-synonym** ('node, 'val) *this-code* = ('node  $\times$  'val, 'val list) *mapping*

**context** *CFG-base* **begin**

**definition** *addN* :: 'g  $\Rightarrow$  'node  $\Rightarrow$  ('var, 'node set) *mapping*  $\Rightarrow$  ('var, 'node set)  
  *mapping*

**where** *addN* g n  $\equiv$  fold ( $\lambda$ v. *Mapping.map-default* v {} (*insert* n)) (*sorted-list-of-set*  
  (*uses* g n))

**definition** *addN'* g n = fold ( $\lambda$ v m. m(v  $\mapsto$  *case-option* {n} (*insert* n) (m v)))  
  (*sorted-list-of-set* (*uses* g n))

**lemma** *addN-transfer* [*transfer-rule*]:  
  *rel-fun* (=) (*rel-fun* (=) (*rel-fun* (*pcr-mapping* (=) (=)) (*pcr-mapping* (=) (=))))  
  *addN'* *addN*

**unfolding** *addN-def* [*abs-def*] *addN'-def* [*abs-def*]

*Mapping.map-default-def* [*abs-def*] *Mapping.default-def*

**apply** (*auto simp: mapping.pcr-cr-eq rel-fun-def cr-mapping-def*)

**apply** *transfer*

**apply** (*rule fold-cong*)

**apply** *simp*

**apply** *simp*

**apply** (*intro ext*)

**by** *auto*

**definition** *useNodes-of* g = fold (*addN* g) ( $\alpha$ n g) *Mapping.empty*

**lemmas** *useNodes-of-code* = *useNodes-of-def* [*unfolded addN-def* [*abs-def*]]

**declare** *useNodes-of-code* [*code*]

**lemma** *lookup-useNodes-of'*:

**assumes** [*simp*]:  $\bigwedge$ n. *finite* (*uses* g n)

**shows** *Mapping.lookup* (*useNodes-of* g) v =

  (*if* ( $\exists$ n  $\in$  set ( $\alpha$ n g). v  $\in$  *uses* g n) *then Some* {n  $\in$  set ( $\alpha$ n g). v  $\in$  *uses* g n}  
  *else None*)

```

proof –
  { fix m n xs v
    have Mapping.lookup (fold (λv. Mapping.map-default (v::'var) {}) (insert
      (n::'node))) xs m) v =
      (case Mapping.lookup m v of None ⇒ (if v ∈ set xs then Some {n} else
        None)
      | Some N ⇒ (if v ∈ set xs then Some (insert n N) else Some N))
    by (induction xs arbitrary: m) (auto simp: Mapping-lookup-map-default split:
      option.splits)
  }
  note addN-conv = this [of n sorted-list-of-set (uses g n) for g n, folded addN-def,
    simplified]
  { fix xs m v
    have Mapping.lookup (fold (addN g) xs m) v = (case Mapping.lookup m v of
      None ⇒ if (∃ n∈set xs. v ∈ uses g n) then Some {n∈set xs. v ∈ uses g n} else
        None
      | Some N ⇒ Some ({n∈set xs. v ∈ uses g n} ∪ N))
    by (induction xs arbitrary: m) (auto split: option.splits simp: addN-conv)
  }
  note this [of αn g Mapping.empty, simp]
  show ?thesis unfolding useNodes-of-def
  by (auto split: option.splits simp: lookup-empty)
qed
end

```

**context** CFG **begin**

```

lift-definition useNodes-of' :: 'g ⇒ ('var, 'node set) mapping
is λg v. if (∃ n ∈ set (αn g). v ∈ uses g n) then Some {n ∈ set (αn g). v ∈ uses
  g n} else None .

```

**lemma** useNodes-of': useNodes-of' = useNodes-of

```

proof
  fix g
  { fix m n xs
    have fold (λv m. m(v::'var ↦ case m v of None ⇒ {n::'node} | Some x ⇒
      insert n x)) xs m =
      (λv. case m v of None ⇒ (if v ∈ set xs then Some {n} else None)
      | Some N ⇒ (if v ∈ set xs then Some (insert n N) else Some N))
    by (induction xs arbitrary: m)(auto split: option.splits)
  }
  note addN'-conv = this [of n sorted-list-of-set (uses g n) for g n, folded
    addN'-def, simplified]
  { fix xs m
    have fold (addN' g) xs m = (λv. case m v of None ⇒ if (∃ n∈set xs. v ∈
      uses g n) then Some {n∈set xs. v ∈ uses g n} else None
      | Some N ⇒ Some ({n∈set xs. v ∈ uses g n} ∪ N))
    by (induction xs arbitrary: m) (auto 4 4 split: option.splits if-splits simp:
      addN'-conv intro!: ext)
  }

```

```

note this [of  $\alpha n$   $g$   $Map.empty$ ,  $simp$ ]
show  $useNodes-of' g = useNodes-of g$ 
unfolding  $mmap-def$   $useNodes-of-def$ 
by ( $transfer$   $fixing: g$ )  $auto$ 
qed

declare  $useNodes-of'.transfer$  [ $unfolded$   $useNodes-of'$ ,  $transfer-rule$ ]

lemma  $lookup-useNodes-of$ :  $Mapping.lookup (useNodes-of g) v =$ 
  ( $if (\exists n \in set (\alpha n g). v \in uses g n)$  then  $Some \{n \in set (\alpha n g). v \in uses g n\}$ 
  else  $None$ )
by  $clarsimp (transfer')$ ;  $auto$ 

end

context  $CFG-SSA-base$  begin
  definition  $phis-addN$ 
  where  $phis-addN g n = fold (\lambda v. Mapping.map-default v \{\} (insert n)) (case-option$ 
   $\ [] id (phis g n))$ 

  definition  $phidefNodes$  where [ $code$ ]:
     $phidefNodes g = fold (\lambda(n,v). Mapping.update v n) (sorted-list-of-set (dom (phis$ 
     $g))) Mapping.empty$ 

  lemma  $keys-phidefNodes$ :
    assumes  $finite (dom (phis g))$ 
    shows  $Mapping.keys (phidefNodes g) = snd ' dom (phis g)$ 
  proof –
    { fix  $xs m x$ 
      have  $fold (\lambda(a,b) m. m(b \mapsto a)) (xs::('node \times 'val) list) m x = (if x \in snd$ 
      ' $set xs$  then  $(Some \circ fst) (last [(b,a) \leftarrow xs. a = x])$  else  $m x$ )
      by ( $induction xs$   $arbitrary: m$ ) ( $auto split: if-splits simp: filter-empty-conv$ 
       $intro: rev-image-eqI$ )
    }
    from  $this$  [ $of sorted-list-of-set (dom (phis g)) Map.empty$ ]  $assms$ 
    show  $?thesis$ 
    unfolding  $phidefNodes-def$   $keys-dom-lookup$ 
    by ( $transfer$   $fixing: g phis$ ) ( $auto simp: dom-def intro: rev-image-eqI$ )
  qed

  definition  $phiNodes-of :: 'g \Rightarrow ('val, ('node \times 'val) set) mapping$ 
  where  $phiNodes-of g = fold (phis-addN g) (sorted-list-of-set (dom (phis g)))$ 
   $Mapping.empty$ 

  lemma  $lookup-phiNodes-of$ :
    assumes [ $simp$ ]:  $finite (dom (phis g))$ 
    shows  $Mapping.lookup (phiNodes-of g) v =$ 
      ( $if (\exists n \in dom (phis g). v \in set (the (phis g n)))$  then  $Some \{n \in dom (phis$ 
       $g). v \in set (the (phis g n))\}$  else  $None$ )

```



```

proof –
{
  fix m n xs v
  have Mapping.lookup (fold (λv. Mapping.map-default v {}) (insert (n::'node ×
'val))) xs (m::('val, ('node × 'val) set) mapping) v =
  (case Mapping.lookup m v of None ⇒ (if v ∈ set xs then Some {n} else None)
  | Some N ⇒ (if v ∈ set xs then Some (insert n N) else Some N))
  by (induction xs arbitrary: m) (auto simp: Mapping.lookup-map-default split:
option.splits)
}
note phis-addN-conv = this [of n case-option [] id (phis g n) for n, folded
phis-addN-def]
{
  fix xs m v
  have Mapping.lookup (fold (phis-addN g) xs m) v =
  (case Mapping.lookup m v of None ⇒ if (∃ n ∈ set xs. v ∈ set (case-option []
id (phis g n))) then Some {n ∈ set xs. v ∈ set (case-option [] id (phis g n))} else
None
  | Some N ⇒ Some ({n ∈ set xs. v ∈ set (case-option [] id (phis g n))} ∪
N))
  by (induction xs arbitrary: m) (auto simp: phis-addN-conv split: option.splits
if-splits)+
}
note this [of sorted-list-of-set (dom (phis g)), simp]
show ?thesis
  unfolding phiNodes-of-def
by (force split: option.splits simp: lookup-empty)
qed

lemmas phiNodes-of-code = phiNodes-of-def [unfolded phis-addN-def [abs-def]]
declare phiNodes-of-code [code]

lemma phis-transfer [transfer-rule]:
includes lifting-syntax
shows ((=) == => pcr-mapping (=) (=)) phis (λg. Mapping.Mapping (phis g))
by (auto simp: mapping.pcr-cr-eq rel-fun-def cr-mapping-def Mapping.Mapping-inverse)

end

context CFG-SSA begin
  declare lookup-phiNodes-of [OF phis-finite, simp]
  declare keys-phidefNodes [OF phis-finite, simp]
end

locale CFG-SSA-ext-base = CFG-SSA-base αe αn invar inEdges' Entry defs uses
phis
  for αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set
  and αn :: 'g ⇒ 'node list
  and invar :: 'g ⇒ bool

```

```

    and inEdges' :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list
    and Entry :: 'g ⇒ 'node
    and defs :: 'g ⇒ 'node ⇒ 'val::linorder set
    and uses :: 'g ⇒ 'node ⇒ 'val set
    and phis :: 'g ⇒ ('node, 'val) phis
begin
  abbreviation cache g f ≡ Mapping.tabulate (αn g) f

  lemma lookup-cache[simp]: n ∈ set (αn g) ⇒ Mapping.lookup (cache g f) n =
  Some (f n)
  by transfer (auto simp: Map.map-of-map-restrict)

  lemma lookup-cacheD [dest]: Mapping.lookup (cache g f) x = Some y ⇒ y =
  f x
  by transfer (auto simp: Map.map-of-map-restrict restrict-map-def split: if-splits)

  lemma lookup-cache-usesD: Mapping.lookup (cache g (uses g)) n = Some vs ⇒
  vs = uses g n
  by blast
end

definition[simp]: usesOf m n ≡ case-option {} id (Mapping.lookup m n)

locale CFG-SSA-ext = CFG-SSA-ext-base αe αn invar inEdges' Entry defs uses
phis
+ CFG-SSA αe αn invar inEdges' Entry defs uses phis
for αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set
and αn :: 'g ⇒ 'node list
and invar :: 'g ⇒ bool
and inEdges' :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list
and Entry :: 'g ⇒ 'node
and defs :: 'g ⇒ 'node ⇒ 'val::linorder set
and uses :: 'g ⇒ 'node ⇒ 'val set
and phis :: 'g ⇒ ('node, 'val) phis
begin
  lemma usesOf-cache[abs-def, simp]: usesOf (cache g (uses g)) n = uses g n
  by (auto simp: uses-in-αn dest: lookup-cache-usesD split: option.split)
end

locale CFG-SSA-base-code = CFG-SSA-ext-base αe αn invar inEdges' Entry defs
usesOf ∘ uses λg. Mapping.lookup (phis g)
for αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set
and αn :: 'g ⇒ 'node list
and invar :: 'g ⇒ bool
and inEdges' :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list
and Entry :: 'g ⇒ 'node
and defs :: 'g ⇒ 'node ⇒ 'val::linorder set
and uses :: 'g ⇒ ('node, 'val set) mapping
and phis :: 'g ⇒ ('node, 'val) phis-code

```

```

begin
  declare phi-transfer [simplified, transfer-rule]

  lemma phiDefs-code [code]:
    phiDefs g n = snd ' Set.filter (λ(n',v). n' = n) (Mapping.keys (phis g))
      unfolding phiDefs-def
      by transfer (auto 4 3 intro: rev-image-eqI simp: Set.filter-def)

  lemmas phiUses-code [code] = phiUses-def [folded Union-of-alt-def]
  declare allUses-def [code]
  lemmas allVars-code [code] = allVars-def [folded Union-of-alt-def]
end

locale CFG-SSA-code = CFG-SSA-base-code αe αn invar inEdges' Entry defs
uses phis
+ CFG-SSA-ext αe αn invar inEdges' Entry defs usesOf ∘ uses λg. Map-
ping.lookup (phis g)
for αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set
and αn :: 'g ⇒ 'node list
and invar :: 'g ⇒ bool
and inEdges' :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list
and Entry :: 'g ⇒ 'node
and defs :: 'g ⇒ 'node ⇒ 'val::linorder set
and uses :: 'g ⇒ ('node, 'val set) mapping
and phis :: 'g ⇒ ('node, 'val) phis-code

definition the-trivial v vs = (case (foldl (λ(good,v') w. if w = v then (good,v')
  else case v' of Some v' ⇒ (good ∧ w = v', Some v')
  | None ⇒ (good, Some w))
  (True, None) vs)
of (False, -) ⇒ None | (True,v) ⇒ v)

lemma the-trivial-Nil [simp]: the-trivial x [] = None
unfolding the-trivial-def by simp

lemma the-trivialI:
  assumes set vs ⊆ {v, v'}
  and v' ≠ v
  shows the-trivial v vs = (if set vs ⊆ {v} then None else Some v')
proof -
  { fix vx
    have [ set vs ⊆ {v, v'}; v' ≠ v; vx ∈ {None, Some v'} ]
    ⇒ (case foldl (λ(good, v') w.
      if w = v then (good, v')
      else case v' of None ⇒ (good, Some w) | Some v' ⇒ (good ∧ w =
v', Some v'))
      (True, vx) vs of
      (True, x) ⇒ x | (False, x) ⇒ None) = (if set vs ⊆ {v} then vx else Some v')
  }

```

```

    by (induction vs arbitrary: vx; case-tac vx; auto)
  }
  with assms show ?thesis unfolding the-trivial-def by simp
qed

lemma the-trivial-conv:
  shows the-trivial v vs = (if  $\exists v' \in \text{set } vs. v' \neq v \wedge \text{set } vs - \{v'\} \subseteq \{v\}$  then
  Some (THE v'. v'  $\in \text{set } vs \wedge v' \neq v \wedge \text{set } vs - \{v'\} \subseteq \{v\}$ ) else None)
proof -
  { fix b a vs
    have a  $\neq v$ 
       $\implies \text{foldl } (\lambda(\text{good}, v') w.
        \text{if } w = v \text{ then } (\text{good}, v')
        \text{ else case } v' \text{ of None } \implies (\text{good}, \text{Some } w) \mid \text{Some } v' \implies (\text{good} \wedge w
= v', \text{Some } v'))
        (b, \text{Some } a) vs =
        (b \wedge \text{set } vs \subseteq \{v, a\}, \text{Some } a)$ 
    by (induction vs arbitrary: b; clarsimp)
  }
  note this[simp]
  { fix b vx
    have  $\llbracket vx \in \text{insert None } (\text{Some } ' \text{set } vs); \text{case-option True } (\lambda vx. vx \neq v) vx \rrbracket$ 
       $\implies \text{foldl } (\lambda(\text{good}, v') w.
        \text{if } w = v \text{ then } (\text{good}, v')
        \text{ else case } v' \text{ of None } \implies (\text{good}, \text{Some } w) \mid \text{Some } v' \implies (\text{good} \wedge w =
v', \text{Some } v'))
        (b, vx) vs = (b \wedge (\text{case } vx \text{ of Some } w \implies \text{set } vs \subseteq \{v, w\} \mid \text{None} \implies \exists w. \text{set }
vs \subseteq \{v, w\}),
        (\text{case } vx \text{ of Some } w \implies \text{Some } w \mid \text{None} \implies \text{if } (\exists v' \in \text{set } vs. v' \neq v) \text{ then Some }
(\text{hd } (\text{filter } (\lambda v'. v' \neq v) vs)) \text{ else None}))$ 
    by (induction vs arbitrary: b vx; auto)
  }
  hence the-trivial v vs = (if  $\exists v' \in \text{set } vs. v' \neq v \wedge \text{set } vs - \{v'\} \subseteq \{v\}$  then
  Some (hd (filter ( $\lambda v'. v' \neq v$ ) vs)) else None)
  unfolding the-trivial-def by (auto split: bool.splits)
  thus ?thesis
  apply (auto split: if-splits)
  apply (rule the-equality [THEN sym])
  by (thin-tac P for P, (induction vs; auto))+
qed

```

```

lemma the-trivial-SomeE:
  assumes the-trivial v vs = Some v'
  obtains v  $\neq v'$  and set vs = {v'}  $\mid v \neq v'$  and set vs = {v, v'}
using assms
apply atomize-elim
apply (subst(asm) the-trivial-conv)
apply (split if-splits; simp)
by (subgoal-tac (THE v'. v'  $\in \text{set } vs \wedge v' \neq v \wedge \text{set } vs - \{v'\} \subseteq \{v\}$ ) = hd (filter

```

```

( $\lambda v'. v' \neq v$ ) vs))
  (fastforce simp: set-double-filter-hd set-single-hd set-minus-one)+

locale CFG-SSA-wf-base-code = CFG-SSA-base-code  $\alpha e$   $\alpha n$  invar inEdges' Entry
defs uses phis
+ CFG-SSA-wf-base  $\alpha e$   $\alpha n$  invar inEdges' Entry defs usesOf  $\circ$  uses  $\lambda g$ . Map-
ping.lookup (phis g)
  for  $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)$  set
  and  $\alpha n :: 'g \Rightarrow 'node$  list
  and invar  $:: 'g \Rightarrow bool$ 
  and inEdges'  $:: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)$  list
  and Entry  $:: 'g \Rightarrow 'node$ 
  and defs  $:: 'g \Rightarrow 'node \Rightarrow 'val::linorder$  set
  and uses  $:: 'g \Rightarrow ('node, 'val$  set) mapping
  and phis  $:: 'g \Rightarrow ('node, 'val)$  phis-code
begin
  definition [code]:
    trivial-code ( $v::'val$ ) vs = (the-trivial  $v$  vs  $\neq None$ )
  definition[code]: trivial-phis  $g = Set.filter (\lambda(n,v). trivial-code v (the (Mapping.lookup$ 
(phis g) ( $n,v$ )))) (Mapping.keys (phis g))
```

**definition** [code]: *redundant-code*  $g = (trivial-phis g \neq \{\})$

**end**

**locale** CFG-SSA-wf-code = CFG-SSA-code  $\alpha e$   $\alpha n$  *invar inEdges' Entry* *defs uses* *phis*

+ CFG-SSA-wf-base-code  $\alpha e$   $\alpha n$  *invar inEdges' Entry* *defs uses phis*

+ CFG-SSA-wf  $\alpha e$   $\alpha n$  *invar inEdges' Entry* *defs usesOf*  $\circ$  *uses*  $\lambda g$ . *Map-*
*ping.lookup (phis g)*

**for**  $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)$  *set*

**and**  $\alpha n :: 'g \Rightarrow 'node$  *list*

**and** *invar*  $:: 'g \Rightarrow bool$

**and** *inEdges'*  $:: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)$  *list*

**and** *Entry*  $:: 'g \Rightarrow 'node$

**and** *defs*  $:: 'g \Rightarrow 'node \Rightarrow 'val::linorder$  *set*

**and** *uses*  $:: 'g \Rightarrow ('node, 'val$  *set)* *mapping*

**and** *phis*  $:: 'g \Rightarrow ('node, 'val)$  *phis-code*

**begin**

**lemma** *trivial-code*:

*phi g v = Some vs*  $\implies$  *trivial g v = trivial-code v vs*

**unfolding** *trivial-def trivial-code-def*

**apply** (*auto split: option.splits simp: isTrivialPhi-def*)

**apply** (*clarsimp simp: the-trivial-conv split: if-splits*)

**apply** (*clarsimp simp: the-trivial-conv split: if-splits*)

**apply** (*erule the-trivial-SomeE*)

**apply** *simp*

**apply** (*rule phiArg-in-allVars; auto simp: phiArg-def*)

**apply** (*rename-tac v'*)

**apply** (*rule-tac x=v' in bexI*)

**apply** *simp*

```

apply (rule phiArg-in-allVars; auto simp: phiArg-def)
done

```

```

lemma trivial-phis:

```

```

  trivial-phis g = {(n,v). Mapping.lookup (phis g) (n,v) ≠ None ∧ trivial g v}
unfolding trivial-phis-def Set.filter-def
apply (auto simp add: phi-def keys-dom-lookup)
apply (subst trivial-code)
apply (auto simp: image-def trivial-in-allVars phis-phi)
apply (frule trivial-phi)
apply (auto simp add: trivial-code phi-def[symmetric] phis-phi)
done

```

```

lemma redundant-code:

```

```

  redundant g = redundant-code g
unfolding redundant-def redundant-code-def trivial-phis[of g]
apply (auto simp: image-def trivial-in-allVars)
apply (frule trivial-phi)
apply (auto simp: phi-def)
done

```

```

lemma trivial-code-mapI:

```

```

  [| trivial-code v vs; f ' (set vs - {v}) ≠ {v} ; f v = v |] ⇒ trivial-code v (map f
vs)
unfolding trivial-code-def the-trivial-conv
  by (auto split: if-splits)

```

```

lemma trivial-code-map-conv:

```

```

  f v = v ⇒ trivial-code v (map f vs) ⇔ (∃ v' ∈ set vs. f v' ≠ v ∧ (f ' set vs)
- {f v'} ⊆ {v})
unfolding trivial-code-def the-trivial-conv
  by auto

```

```

end

```

```

locale CFG-SSA-Transformed-code = ssa: CFG-SSA-wf-code αe αn invar inEdges'
Entry defs uses phis

```

```

+

```

```

  CFG-SSA-Transformed αe αn invar inEdges' Entry oldDefs oldUses defs usesOf
○ uses λg. Mapping.lookup (phis g) var

```

```

for

```

```

  αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set and
  αn :: 'g ⇒ 'node list and
  invar :: 'g ⇒ bool and
  inEdges' :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list and
  Entry::'g ⇒ 'node and
  oldDefs :: 'g ⇒ 'node ⇒ 'var::linorder set and
  oldUses :: 'g ⇒ 'node ⇒ 'var set and
  defs :: 'g ⇒ 'node ⇒ 'val::linorder set and

```

```

    uses :: 'g ⇒ ('node, 'val set) mapping and
    phis :: 'g ⇒ ('node, 'val) phis-code and
    var :: 'g ⇒ 'val ⇒ 'var
+
assumes dom-uses-in-graph: Mapping.keys (uses g) ⊆ set (αn g)

end

```

## 6.2 Code Equations for SSA Construction

```

theory Construct-SSA-code imports

```

```

  SSA-CFG-code

```

```

  Construct-SSA

```

```

  Mapping-Exts

```

```

  HOL-Library.Product-Lexorder

```

```

begin

```

```

definition[code]: lookup-multimap m k ≡ (case-option {} id (Mapping.lookup m
k))

```

```

locale CFG-Construct-linorder = CFG-Construct-wf αe αn invar inEdges' Entry
defs uses

```

```

for

```

```

  αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set and

```

```

  αn :: 'g ⇒ 'node list and

```

```

  invar :: 'g ⇒ bool and

```

```

  inEdges' :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list and

```

```

  Entry :: 'g ⇒ 'node and

```

```

  defs :: 'g ⇒ 'node ⇒ ('var::linorder) set and

```

```

  uses :: 'g ⇒ 'node ⇒ 'var set

```

```

begin

```

```

  type-synonym ('n, 'v) sparse-phis = ('n × 'v, ('n, 'v) ssaVal list) mapping

```

```

  function readVariableRecursive :: 'g ⇒ 'var ⇒ 'node ⇒ ('node, 'var) sparse-phis
⇒ (('node, 'var) ssaVal × ('node, 'var) sparse-phis)

```

```

    and readArgs :: 'g ⇒ 'var ⇒ 'node ⇒ ('node, 'var) sparse-phis ⇒ 'node list
⇒ ('node, 'var) sparse-phis × ('node, 'var) ssaVal list

```

```

  where[code]: readVariableRecursive g v n phis = (if v ∈ defs g n then ((v,n,SimpleDef),
phis)

```

```

    else case predecessors g n of

```

```

      [] ⇒ ((v,n,PhiDef), Mapping.update (n,v) [] phis)

```

```

      | [m] ⇒ readVariableRecursive g v m phis

```

```

      | ms ⇒ (case Mapping.lookup phis (n,v) of

```

```

          Some - ⇒ ((v,n,PhiDef),phis)

```

```

          | None ⇒

```

```

            let phis = Mapping.update (n,v) [] phis in

```

```

            let (phis,args) = readArgs g v n phis ms in

```

```

            ((v,n,PhiDef), Mapping.update (n,v) args phis)

```

```

    ))

```

```

| readArgs g v n phis [] = (phis,[])
| readArgs g v n phis (m#ms) = (
  let (phis,args) = readArgs g v n phis ms in
  let (v,phis) = readVariableRecursive g v m phis in
  (phis,v#args))
by pat-completeness auto

```

**lemma** *length-filter-less2*:

```

  assumes  $x \in \text{set } xs \neg P x Q x \wedge x. P x \implies Q x$ 
  shows  $\text{length } (\text{filter } P xs) < \text{length } (\text{filter } Q xs)$ 
proof–
  have  $\wedge x. (Q x \wedge P x) = P x$ 
  using assms(4) by auto
  hence  $\text{filter } P xs = \text{filter } P (\text{filter } Q xs)$ 
  by auto
  also have  $\text{length } (...) < \text{length } (\text{filter } Q xs)$ 
  using assms(1-3) by – (rule length-filter-less, auto)
  finally show ?thesis .
qed

```

**lemma** *length-filter-le2*:

```

  assumes  $\wedge x. P x \implies Q x$ 
  shows  $\text{length } (\text{filter } P xs) \leq \text{length } (\text{filter } Q xs)$ 
proof–
  have  $\wedge x. (Q x \wedge P x) = P x$ 
  using assms by auto
  hence  $\text{filter } P xs = \text{filter } P (\text{filter } Q xs)$ 
  by auto
  also have  $\text{length } (...) \leq \text{length } (\text{filter } Q xs)$ 
  by – (rule length-filter-le)
  finally show ?thesis .
qed

```

**abbreviation** *phis-measure*  $g v phis \equiv \text{length } [n \leftarrow \alpha n g. \text{Mapping.lookup } phis (n,v) = \text{None}]$

**lemma** *phis-measure-update-le*:  $\text{phis-measure } g v (\text{Mapping.update } k a p) \leq \text{phis-measure } g v p$

```

apply (rule length-filter-le2)
apply (case-tac k = (x, v))
apply (auto simp: lookup-update lookup-update-neq)
done

```

**lemma** *phis-measure-update-le'*:  $\text{phis-measure } g v p \leq \text{phis-measure } g v (\text{Mapping.update } k [] phis) \implies \text{phis-measure } g v (\text{Mapping.update } k a p) \leq \text{phis-measure } g v phis$

```

apply (rule le-trans, rule phis-measure-update-le)
apply (rule le-trans, assumption, rule phis-measure-update-le)
done

```



```

lemma readArgs-phis-le:
  readVariableRecursive-readArgs-dom (Inl (g, v, n, phis))  $\implies$  (val,p) = read-
VariableRecursive g v n phis  $\implies$  phis-measure g v p  $\leq$  phis-measure g v phis
  readVariableRecursive-readArgs-dom (Inr (g, v, n, phis, ms))  $\implies$  (p,u) =
readArgs g v n phis ms  $\implies$  phis-measure g v p  $\leq$  phis-measure g v phis
  proof (induction arbitrary: val p and p u rule: readVariableRecursive-readArgs.pinduct)
    case (1 g v n phis)
      show ?case
      using 1.IH(1,2) 1.prem
      apply (auto simp: readVariableRecursive.psimps Let-def phis-measure-update-le
split: if-split-asm list.splits option.splits prod.splits)
      apply (subgoal-tac phis-measure g v x1  $\leq$  phis-measure g v (Mapping.update
(n,v) [] phis))
      defer
      apply (rule 1.IH(3))
      apply (auto simp: phis-measure-update-le')
      done
    next
      case (3 g v n m ms phis)
      from 3.IH(1) 3.prem show ?case
      apply (auto simp: readArgs.psimps split: prod.splits)
      apply (rule le-trans)
      apply (rule 3.IH(3))
      apply auto
      apply (rule 3.IH(2))
      apply auto
      done
    qed (auto simp: readArgs.psimps split: prod.splits)

termination
apply (relation measures [
   $\lambda$ args. let (g,v,phis) = case args of Inl((g,v,n,phis))  $\Rightarrow$  (g,v,phis) | Inr((g,v,n,phis,ms))
 $\Rightarrow$  (g,v,phis) in
    phis-measure g v phis,
   $\lambda$ args. case args of Inl(-)  $\Rightarrow$  0 | Inr((g,v,n,phis,ms))  $\Rightarrow$  length ms,
   $\lambda$ args. let (g,n) = case args of Inl((g,v,n,phis))  $\Rightarrow$  (g,n) | Inr((g,v,n,ms,phis))
 $\Rightarrow$  (g,n) in
    shortestPath g n
  ])
apply (auto intro: shortestPath-single-predecessor)[2]
apply clarsimp
apply (rule-tac x=n in length-filter-less2)
  apply (rule successor-in- $\alpha$ n; auto)
  apply (auto simp: lookup-update)[2]
apply (case-tac x=n; auto simp: lookup-update-neq)
apply (auto dest: readArgs-phis-le)
done

```

```

declare readVariableRecursive.simps[simp del] readArgs.simps[simp del]

lemma fst-readVariableRecursive:
  assumes  $n \in \text{set } (\alpha n \ g)$ 
  shows  $\text{fst } (\text{readVariableRecursive } g \ v \ n \ \text{phis}) = \text{lookupDef } g \ n \ v$ 
using assms
apply (induction rule: lookupDef-induct[where  $v=v$ ])
  apply (simp add: readVariableRecursive.simps)
  apply (simp add: readVariableRecursive.simps; auto simp: split-def Let-def split: list.split option.split)
  apply (auto simp add: readVariableRecursive.simps)
done

definition phis'-aux  $g \ v \ ns$  (phis:: ('node, 'var) sparse-phis)  $\equiv \text{Mapping.Mapping}$ 
( $\lambda(m, v_2).$ 
  (if  $v_2=v \wedge m \in \bigcup(\text{phiDefNodes-aux } g \ v \ [n \leftarrow \alpha n \ g. (n, v) \notin \text{Mapping.keys phis}] \ 'ns) \wedge v \in \text{vars } g$  then Some (map ( $\lambda m. \text{lookupDef } g \ m \ v$ ) (predecessors  $g \ m$ )) else (Mapping.lookup phis ( $m, v_2$ ))))))

lemma phis'-aux-keys-super:  $\text{Mapping.keys } (\text{phis'-aux } g \ v \ ns \ \text{phis}) \supseteq \text{Mapping.keys phis}$ 
by (auto simp: keys-dom-lookup phis'-aux-def)

lemma phiDefNodes-aux-in-unvisited:
  shows  $\text{phiDefNodes-aux } g \ v \ un \ n \subseteq \text{set } un$ 
proof (induction un arbitrary: n rule:removeAll-induct)
  case ( $1 \ un$ )
  show ?case
  apply (simp only: phiDefNodes-aux.simps)
  apply (auto elim!: fold-union-elem)
  apply (rename-tac m n')
  apply (drule-tac x2=n and n2=n' in 1)
  apply auto[1]
  apply (rename-tac m n')
  apply (drule-tac x2=n and n2=n' in 1)
  apply auto
done
qed

lemma phiDefNodes-aux-unvisited-monotonic:
  assumes  $\text{set } un \subseteq \text{set } un'$ 
  shows  $\text{phiDefNodes-aux } g \ v \ un \ n \subseteq \text{phiDefNodes-aux } g \ v \ un' \ n$ 
using assms proof (induction un arbitrary: un' n rule:removeAll-induct)
  case ( $1 \ un$ )
  {
    fix  $m \ A$ 
    assume  $n \in \text{set } un$ 
    hence  $a: \bigwedge m. \text{phiDefNodes-aux } g \ v \ (\text{removeAll } n \ un) \ m \subseteq \text{phiDefNodes-aux}$ 
 $g \ v \ (\text{removeAll } n \ un') \ m$ 
  }

```

```

apply (rule 1)
using 1(2)
by auto

assume  $m \in \text{fold } (\cup) (\text{map } (\text{phiDefNodes-aux } g \ v \ (\text{removeAll } n \ un)) (\text{predecessors } g \ n)) \ A$ 
hence  $m \in \text{fold } (\cup) (\text{map } (\text{phiDefNodes-aux } g \ v \ (\text{removeAll } n \ un')) (\text{predecessors } g \ n)) \ A$ 
apply (rule fold-union-elem)
apply (rule fold-union-elemI')
apply (auto simp: image-def dest: a[THEN subsetD])
done
}
with 1(2) show ?case
apply (subst(1 2) phiDefNodes-aux.simps)
by auto
qed

lemma phiDefNodes-aux-single-pred:
assumes predecessors g n = [m]
shows phiDefNodes-aux g v (removeAll n un) m = phiDefNodes-aux g v un m
proof-
{
fix n' ns
assume asm:  $g \vdash n' - ns \rightarrow m$  distinct ns length (predecessors g n')  $\neq 1$   $n \in \text{set } ns$ 
then obtain ns1 ns2 where split:  $g \vdash n' - ns_1 \rightarrow n \ g \vdash n - ns_2 \rightarrow m$  ns = butlast ns1 @ ns2
by - (rule path2-split-ex)
with ⟨distinct ns⟩ have  $m \notin \text{set } (\text{butlast } ns_1)$ 
by (auto dest: path2-last-in-ns)
from split(1,2) have False
apply-
apply (frule path2-unsnoc)
apply (erule path2-nontrivial)
using assms asm(3) ⟨ $m \notin \text{set } (\text{butlast } ns_1)$ ⟩
apply (auto dest: path2-not-Nil)
done
}
with assms show ?thesis
apply-
apply rule
apply (rule phiDefNodes-aux-unvisited-monotonic; auto)
apply (rule subsetI)
apply (rename-tac n')
apply (erule phiDefNodes-auxE)
apply (rule predecessor-is-node[where n'=n]; auto)
apply (rule phiDefNodes-auxI; auto)
done

```

qed

**lemma** *phis'-aux-finite*:

**assumes** *finite* (*Mapping.keys phis*)

**shows** *finite* (*Mapping.keys (phis'-aux g v ns phis)*)

**proof**–

**have**  $a: \bigwedge n. \text{phiDefNodes-aux } g \ v \ [n \leftarrow \alpha n \ g \ . \ (n, v) \notin \text{dom } (\text{Mapping.lookup } \text{phis})]$   $n \subseteq (\text{set } (\alpha n \ g))$

**by** (*rule subset-trans, rule phiDefNodes-aux-in-unvisited, auto*)

**have** *Mapping.keys (phis'-aux g v ns phis)*  $\subseteq \text{set } (\alpha n \ g) \times \text{vars } g \cup \text{Mapping.keys } \text{phis}$

**by** (*auto simp: phis'-aux-def keys-dom-lookup split: if-split-asm dest: subsetD[OF a]*)

**thus** *?thesis* **by** (*rule finite-subset, auto intro: assms*)

qed

**lemma** *phiDefNodes-aux-redirect*:

**assumes** *asm*:  $g \vdash n - ns \rightarrow m \ \forall n \in \text{set } ns. v \notin \text{defs } g \ n \ \text{length } (\text{predecessors } g \ n) \neq 1 \ \text{unvisitedPath } un \ ns$

**assumes**  $n': n' \in \text{set } ns \ n' \in \text{phiDefNodes-aux } g \ v \ un \ m' \ m' \in \text{set } (\alpha n \ g)$

**shows**  $n \in \text{phiDefNodes-aux } g \ v \ un \ m'$

**proof**–

**from** *asm*(1) *n'*(1) **obtain**  $ns_1$  **where**  $ns_1: g \vdash n - ns_1 \rightarrow n' \ \text{set } ns_1 \subseteq \text{set } ns$

**by** (*rule path2-split-ex, simp*)

**from** *n'*(2–3) **obtain**  $ns'$  **where**  $ns': g \vdash n' - ns' \rightarrow m' \ \forall n \in \text{set } ns'. v \notin \text{defs } g \ n \ \text{length } (\text{predecessors } g \ n') \neq 1$

*unvisitedPath } un } ns'*

**by** (*rule phiDefNodes-auxE*)

**from**  $ns_1$ (1) *ns'*(1) **obtain**  $ms$  **where**  $ms: g \vdash n - ms \rightarrow m' \ \text{distinct } ms \ \text{set } ms \subseteq \text{set } ns_1 \cup \text{set } (\text{tl } ns')$

**by** – (*drule path2-app, auto elim: simple-path2*)

**show** *?thesis*

**using**  $ms$ (1)

**apply** (*rule phiDefNodes-auxI*)

**using**  $ms$   $asm$ (4)  $ns_1$ (2) *ns'*(4)

**apply** *clarsimp*

**apply** (*rename-tac x*)

**apply** (*case-tac x*  $\in \text{set } ns_1$ )

**apply** (*drule-tac A=set ns and c=x in subsetD; auto*)

**apply** (*drule-tac A=set ns' and c=x in subsetD; auto*)

**using**  $asm$ (2–3)  $ns_1$ (2) *ns'*(2)  $ms$ (3)

**apply** (*auto dest!: bspec*)

**done**

qed

**lemma** *snd-readVariableRecursive*:

```

assumes  $v \in \text{vars } g \ n \in \text{set } (\alpha n \ g) \ \text{finite } (\text{Mapping.keys } \text{phis})$ 
 $\wedge n. (n, v) \in \text{Mapping.keys } \text{phis} \implies \text{length } (\text{predecessors } g \ n) \neq 1 \ \text{Mapping.lookup } \text{phis } (\text{Entry } g, v) \in \{\text{None}, \text{Some } []\}$ 
shows
 $\text{phis}'\text{-aux } g \ v \ \{n\} \ \text{phis} = \text{snd } (\text{readVariableRecursive } g \ v \ n \ \text{phis})$ 
 $\text{set } ms \subseteq \text{set } (\alpha n \ g) \implies (\text{phis}'\text{-aux } g \ v \ (\text{set } ms) \ \text{phis}, \text{map } (\lambda m. \text{lookupDef } g \ m \ v) \ ms) = \text{readArgs } g \ v \ n \ \text{phis } ms$ 
using assms proof (induction  $g \ v \ n \ \text{phis}$  and  $g \ v \ n \ \text{phis } ms$  rule: readVariableRecursive-readArgs.induct)
case ( $1 \ g \ v \ n \ \text{phis}$ )
note  $1.\text{prems}(1-3)[\text{simp}]$ 
note  $\text{phis}\text{-wf} = 1.\text{prems}(4)[\text{rule-format}]$ 

from  $1.\text{prems}(5)$  have  $a: (\text{Entry } g, v) \in \text{Mapping.keys } \text{phis} \implies \text{Mapping.lookup } \text{phis } (\text{Entry } g, v) = \text{Some } []$ 
by (auto simp: keys-dom-lookup)

have IH1:  $\wedge m. v \notin \text{defs } g \ n \implies \text{predecessors } g \ n = [m] \implies \text{phis}'\text{-aux } g \ v \ \{m\} \ \text{phis} = \text{snd } (\text{readVariableRecursive } g \ v \ m \ \text{phis})$ 
apply (rule  $1.\text{IH}[\text{rule-format}]$ )
apply auto[4]
apply (rule-tac  $n'=n$  in predecessor-is-node; auto)
using  $1.\text{prems}(5)$ 
apply (auto dest: phis-wf)
done

{
fix  $m_1 \ m_2 :: 'node$ 
fix  $ms' :: 'node \ \text{list}$ 
let  $?ms = m_1 \# m_2 \# ms'$ 
let  $?phis' = \text{Mapping.update } (n, v) \ [] \ \text{phis}$ 
assume asm:  $v \notin \text{defs } g \ n \ \text{predecessors } g \ n = ?ms \ \text{Mapping.lookup } \text{phis } (n, v) = \text{None}$ 
moreover have  $\text{set } ?ms \subseteq \text{set } (\alpha n \ g)$ 
by (rule subsetI, rule predecessor-is-node[of - g n]; auto simp: asm(2))
ultimately have  $\text{readArgs } g \ v \ n \ ?phis' \ ?ms = (\text{phis}'\text{-aux } g \ v \ (\text{set } ?ms) \ ?phis', \text{map } (\lambda m. \text{lookupDef } g \ m \ v) \ ?ms)$ 
using  $1.\text{prems}(5)$ 
by - (rule  $1.\text{IH}(2)[\text{symmetric}, \text{rule-format}]; \text{auto dest: phis-wf simp: lookup-update-cases}$ )
}
note IH2 = this

note foldr-Cons[simp del] fold-Cons[simp del] list.map(2)[simp del] set-simps(2)[simp del]

have  $c: \wedge f \ x. \bigcup (f \ ' \ \{x\}) = f \ x$  by auto

show ?case
unfolding phis'-aux-def c
apply (subst readVariableRecursive.simps)

```

```

apply (subst phiDefNodes-aux.simps[abs-def])
apply (cases predecessors g n)
apply (auto simp: a Mapping-eq-lookup lookup-update-cases Entry-iff-unreachable[OF
invar] split: list.split intro!: ext)[1]
apply (rename-tac m1 ms)
apply (case-tac ms)
apply (subst Mapping-eq-lookup)
apply (intro ext)
apply (auto simp: fold-Cons list.map(2))[1]
  apply (auto dest: phis-wf)[1]
  apply (subst IH1[symmetric], assumption, assumption)
  apply (auto simp: phis'-aux-def)[1]
  apply (drule rev-subsetD, rule phiDefNodes-aux-unvisited-monotonic[where
un'=[n←αn g . (n, v) ∉ Mapping.keys phis]]; auto)
  apply (subst IH1[symmetric], assumption, assumption)
  apply (auto simp: phis'-aux-def)[1]
  apply (subst IH1[symmetric], assumption, assumption)
  apply (auto simp: phis'-aux-def phiDefNodes-aux-single-pred)[1]
apply (auto simp: Mapping-eq-lookup lookup-update-cases intro!: ext)
  apply (auto simp: keys-dom-lookup)[1]
  apply (auto split: option.split prod.split)[1]
  apply (subst(asm) IH2, assumption, assumption, assumption)
  apply (erule fold-union-elem)
  apply (auto simp: lookup-update-cases phis'-aux-def[abs-def])[1]
  apply (drule rev-subsetD, rule phiDefNodes-aux-unvisited-monotonic[where
un'=[n'←αn g . n' ≠ n ∧ (n', v) ∉ Mapping.keys phis]]; auto)
  apply (drule rev-subsetD, rule phiDefNodes-aux-unvisited-monotonic[where
un'=[n'←αn g . n' ≠ n ∧ (n', v) ∉ Mapping.keys phis]]; auto)
  apply (rename-tac m)
  apply (erule-tac x=m in ballE)
  apply (drule rev-subsetD, rule phiDefNodes-aux-unvisited-monotonic[where
un'=[n'←αn g . n' ≠ n ∧ (n', v) ∉ Mapping.keys phis]]; auto)
  apply auto[1]
  apply (subst(asm) IH2, assumption, assumption)
  apply (auto simp: keys-dom-lookup)[2]
  apply (auto split: option.split prod.split)[1]
  apply (subst(asm) IH2, assumption, assumption, assumption)
  apply (auto simp: lookup-update-neq phis'-aux-def)[1]
apply (auto split: option.splits prod.splits)[1]
apply (subst(asm) IH2, assumption, assumption, assumption)
  apply (auto simp: lookup-update-cases phis'-aux-def removeAll-filter-not-eq
image-def split: if-split-asm)[1]
  apply (cut-tac fold-union-elemI)
  apply auto[3]
  apply (cut-tac fold-union-elemI)
  apply auto[1]
  apply assumption
  apply (subgoal-tac [x←αn g . x ≠ n ∧ (x, v) ∉ Mapping.keys phis] = [x←αn
g . (x, v) ∉ Mapping.keys phis ∧ n ≠ x])

```

```

    apply auto[1]
    apply (rule arg-cong2[where f=filter])
    apply auto[2]
    apply (cut-tac fold-union-elemI)
    apply auto[1]
    apply assumption
    apply (subgoal-tac [x←αn g . x ≠ n ∧ (x, v) ∉ Mapping.keys phis] = [x←αn
g . (x, v) ∉ Mapping.keys phis ∧ n ≠ x])
    apply auto[1]
    apply (rule arg-cong2[where f=filter])
    apply auto[2]
    apply (cut-tac fold-union-elemI)
    apply auto[1]
    apply assumption
    apply (subgoal-tac [x←αn g . x ≠ n ∧ (x, v) ∉ Mapping.keys phis] = [x←αn
g . (x, v) ∉ Mapping.keys phis ∧ n ≠ x])
    apply auto[1]
    apply (rule arg-cong2[where f=filter])
    apply auto[2]
    done
next
case (∃ g v n phis m ms)
note 3.premis(2-4)[simp]
from 3.premis(1) have[simp]: m ∈ set (αn g) by auto

from 3 have IH1: readArgs g v n phis ms = (phis'-aux g v (set ms) phis, map
(λm. lookupDef g m v) ms)
by auto

have IH2: phis'-aux g v {m} (phis'-aux g v (set ms) phis) = snd (readVariableRecursive
g v m (phis'-aux g v (set ms) phis))
apply (rule 3.IH(2))
    apply (auto simp: IH1 intro: phis'-aux-finite)[5]
    apply (simp add: phis'-aux-def keys-dom-lookup dom-def split: if-split-asm)
    apply safe
    apply (erule phiDefNodes-auxE)
    using 3.premis(1,5)
    apply (auto simp: keys-dom-lookup)[3]
using 3.premis(6)
apply (auto simp: phis'-aux-def split: if-split-asm)
done

have a: phiDefNodes-aux g v [n←αn g . (n, v) ∉ Mapping.keys (phis'-aux g v
(set ms) phis)] m ⊆ phiDefNodes-aux g v [n←αn g . (n, v) ∉ Mapping.keys phis]
m
    apply (rule phiDefNodes-aux-unvisited-monotonic)
    by (auto dest: phis'-aux-keys-super[THEN subsetD])

{

```

```

fix n
  assume m: n ∈ phiDefNodes-aux g v [n←αn g . (n, v) ∉ Mapping.keys phis]
m and
  ms: ∀ x ∈ set ms. n ∉ phiDefNodes-aux g v [n←αn g . (n, v) ∉ Mapping.keys
phis] x

  have n ∈ phiDefNodes-aux g v [n←αn g . (n, v) ∉ Mapping.keys (phis'-aux
g v (set ms) phis)] m
  using m
  apply–
  apply (erule phiDefNodes-auxE, simp)
  apply (rule phiDefNodes-auxI)
    apply (auto simp: phis'-aux-def keys-dom-lookup split: if-split-asm)[3]
    apply (drule phiDefNodes-aux-redirect)
      using 3.prem1
      apply auto[6]
    apply (rule ms[THEN ballE]; auto simp: keys-dom-lookup)
  apply auto
  done
}
note b = this

show ?case
unfolding readArgs.simps phis'-aux-def
unfolding IH1
apply (simp add: split-def Let-def IH2[symmetric])
apply (subst phis'-aux-def)
apply (subst(2) phis'-aux-def)
apply (auto simp: Mapping-eq-lookup fst-readVariableRecursive split: prod.splits
intro!: ext dest!: a[THEN subsetD] b)
done
qed (auto simp: readArgs.simps phis'-aux-def)

definition aux-1 g n = (λv (uses,phis).
  let (use,phis') = readVariableRecursive g v n phis in
  (Mapping.update n (insert use (lookup-multimap uses n)) uses, phis')
)

definition aux-2 g n = foldr (aux-1 g n) (sorted-list-of-set (uses g n))

abbreviation init-state ≡ (Mapping.empty, Mapping.empty)
abbreviation from-sparse ≡ λ(n,v). (n,(v,n,PhiDef))
definition uses'-phis' g = (
  let (u,p) = foldr (aux-2 g) (αn g) init-state in
  (u, map-keys from-sparse p)
)

lemma from-sparse-inj: inj from-sparse
by (rule injI, auto)

```



```

declare uses'-phis'-def[unfolded aux-2-def[abs-def] aux-1-def, code]

lift-definition phis'-code :: 'g  $\Rightarrow$  ('node, ('node, 'var) ssaVal) phis-code is phis'
.

lemma foldr-prod: foldr ( $\lambda x y. (f1 x (fst y), f2 x (snd y))$ ) xs y = (foldr f1 xs
(fst y), foldr f2 xs (snd y))
by (induction xs, auto)

lemma foldr-aux-1:
assumes set us  $\subseteq$  uses g n Mapping.lookup u n = None foldr (aux-1 g n) us
(u,p) = (u',p') (is foldr ?f - - = -)
assumes finite (Mapping.keys p)  $\wedge$  n v. (n,v)  $\in$  Mapping.keys p  $\implies$  length
(predecessors g n)  $\neq$  1  $\wedge$  v. Mapping.lookup p (Entry g,v)  $\in$  {None, Some []}
shows lookupDef g n ' set us = lookup-multimap u' n  $\wedge$  m. m  $\neq$  n  $\implies$  Map-
ping.lookup u' m = Mapping.lookup u m
 $\wedge$  m v. (if m  $\in$  phiDefNodes-aux g v [n  $\leftarrow$   $\alpha$ n g. (n,v)  $\notin$  Mapping.keys p] n
 $\wedge$  v  $\in$  set us then
  Some (map ( $\lambda m. lookupDef g m v$ ) (predecessors g m)) else
  (Mapping.lookup p (m,v))) = Mapping.lookup p' (m,v)
using assms proof (induction us arbitrary: u' p')
case (Cons v us)
let ?u = fst (foldr ?f us (u,p))
let ?p = snd (foldr ?f us (u,p))
{
  case 1
  have n  $\in$  set ( $\alpha$ n g) using 1(1) uses-in- $\alpha$ n by auto
  hence lookupDef g n v = fst (readVariableRecursive g v n ?p)
  by (rule fst-readVariableRecursive[symmetric])
  moreover have lookupDef g n ' set us = lookup-multimap ?u n
  using 1 by - (rule Cons(1)[of ?u ?p], auto)
  ultimately show ?case
  using 1(3) by (auto simp: aux-1-def split-def Let-def lookup-multimap-def
lookup-update split: option.splits)
next
  case 2
  have Mapping.lookup ?u m = Mapping.lookup u m
  using 2 by - (rule Cons(2)[of - ?u ?p], auto)
  thus ?case
  using 2 by (auto simp: aux-1-def split-def Let-def lookup-multimap-def
lookup-update-neq split: option.splits)
next
  case (3 m v' u' p')
  from 3(1) have[simp]:  $\wedge v. v \in$  set us  $\implies v \in$  vars g
  by auto

  from 3 have IH:  $\wedge m v'. (if m \in$  phiDefNodes-aux g v' [n  $\leftarrow$   $\alpha$ n g. (n,v')  $\notin$ 
Mapping.keys p] n  $\wedge$  v'  $\in$  set us then

```

```

    Some (map (λm. lookupDef g m v') (predecessors g m)) else
    (Mapping.lookup p (m,v')) = Mapping.lookup ?p (m,v')
  by - (rule Cons(3)[of ?u ?p], auto)

  have rVV: phis'-aux g v {n} ?p = snd (readVariableRecursive g v n ?p)
  apply (rule snd-readVariableRecursive(1))
    using 3
    apply (auto simp: uses-in-αn)[2]
    apply (rule finite-subset[where B=set (αn g) × vars g ∪ Mapping.keys p])
    apply (auto simp: keys-dom-lookup IH[symmetric] split: if-split-asm dest!:
  phiDefNodes-aux-in-unvisited[THEN subsetD])[1]
    apply (simp add: 3(4))[1]
    using 3(5-6)
    apply (auto simp: keys-dom-lookup dom-def IH[symmetric] split: if-split-asm
  dest!: phiDefNode-aux-is-join-node)
  done

  have a: m ∈ phiDefNodes-aux g v [n←αn g . (n, v) ∉ Mapping.keys ?p] n
  ⇒ m ∈ phiDefNodes-aux g v [n←αn g . (n, v) ∉ Mapping.keys p] n
  apply (erule rev-subsetD)
  apply (rule phiDefNodes-aux-unvisited-monotonic)
  by (auto simp: IH[symmetric] keys-dom-lookup split: if-split-asm)

  have b: v ∉ set us ⇒ [n←αn g . (n, v) ∉ Mapping.keys ?p] = [n←αn g .
  (n, v) ∉ Mapping.keys p]
  by (rule arg-cong2[where f=filter], auto simp: keys-dom-lookup IH[symmetric])

  from 3 show ?case
  unfolding aux-1-def
  unfolding foldr.foldr-Cons
  unfolding aux-1-def[symmetric]
  by (auto simp: Let-def split-def IH[symmetric] rVV[symmetric] phis'-aux-def
  b dest: a uses-in-vars split: if-split-asm)
  }
  qed (auto simp: lookup-multimap-def)

  lemma foldr-aux-2:
  assumes set ns ⊆ set (αn g) distinct ns foldr (aux-2 g) ns init-state = (u',p')
  shows ∧n. n ∈ set ns ⇒ uses' g n = lookup-multimap u' n ∧n. n ∉ set ns
  ⇒ Mapping.lookup u' n = None
  ∧m v. (if ∃n ∈ set ns. m ∈ phiDefNodes-aux g v (αn g) n ∧ v ∈ uses g n
  then
    Some (map (λm. lookupDef g m v) (predecessors g m)) else
    None) = Mapping.lookup p' (m,v)
  using assms proof (induction ns arbitrary: u' p')
  case (Cons n ns)
  let ?u = fst (foldr (aux-2 g) ns init-state)
  let ?p = snd (foldr (aux-2 g) ns init-state)

```

```

fix m u' p'
assume asm: set (n#ns)  $\subseteq$  set ( $\alpha n$  g) distinct (n#ns) foldr (aux-2 g) (n#ns)
init-state = (u', p')
hence IH:
   $\bigwedge n. n \in \text{set } ns \implies \text{uses}' g n = \text{lookup-multimap } ?u n$ 
   $\bigwedge n. n \notin \text{set } ns \implies \text{Mapping.lookup } ?u n = \text{None}$ 
   $\bigwedge m v. (\text{if } \exists n \in \text{set } ns. m \in \text{phiDefNodes-aux } g v (\alpha n g) n \wedge v \in \text{uses } g n$ 
then
  Some (map ( $\lambda m. \text{lookupDef } g m v$ ) (predecessors g m)) else
  None) = Mapping.lookup ?p (m,v)
apply -
  apply (rule Cons.IH(1)[where p'2=?p]; auto; fail)
  apply (rule Cons.IH(2)[where p'2=?p]; auto; fail)
by (rule Cons.IH(3)[where u'2=?u], auto)

with this[of n] asm(2) have a': Mapping.lookup ?u n = None by simp
moreover have finite (Mapping.keys ?p)
by (rule finite-subset[where B=set ( $\alpha n$  g)  $\times$  vars g]) (auto simp: keys-dom-lookup
IH[symmetric] split: if-split-asm dest!: phiDefNodes-aux-in-unvisited[THEN sub-
setD])
moreover have  $\bigwedge n v. (n,v) \in \text{Mapping.keys } ?p \implies \text{length } (\text{predecessors } g n) \neq 1$ 
by (auto simp: keys-dom-lookup dom-def IH[symmetric] split: if-split-asm
dest!: phiDefNode-aux-is-join-node)
moreover have  $\bigwedge v. \text{Mapping.lookup } ?p (\text{Entry } g, v) \in \{\text{None}, \text{Some } []\}$ 
by (auto simp: IH[symmetric])
ultimately have aux-2: lookupDef g n 'uses g n = lookup-multimap u' n  $\bigwedge m. m \neq n \implies \text{Mapping.lookup } u' m = \text{Mapping.lookup } ?u m$ 
 $\bigwedge m v. (\text{if } m \in \text{phiDefNodes-aux } g v [n \leftarrow \alpha n g. (n,v) \notin \text{Mapping.keys } ?p] n \wedge v \in \text{uses } g n$  then
  Some (map ( $\lambda m. \text{lookupDef } g m v$ ) (predecessors g m)) else
  (Mapping.lookup ?p (m,v))) = Mapping.lookup p' (m,v)
apply-
  apply (rule foldr-aux-1(1)[of sorted-list-of-set (uses g n) g n ?u ?p u' p',
simplified]; simp add: aux-2-def[symmetric] asm(3)[simplified]; fail)
  apply (rule foldr-aux-1(2)[of sorted-list-of-set (uses g n) g n ?u ?p u' p',
simplified]; simp add: aux-2-def[symmetric] asm(3)[simplified]; fail)
  apply (rule foldr-aux-1(3)[of sorted-list-of-set (uses g n) g n ?u ?p u' p',
simplified]; simp add: aux-2-def[symmetric] asm(3)[simplified]; fail)
done

{
assume 1: m  $\in$  set (n#ns)
show uses' g m = lookup-multimap u' m
apply (cases m = n)
apply (simp add: uses'-def aux-2)
using 1 asm(2)
apply (auto simp: IH(1) lookup-multimap-def aux-2(2))
done
}

```

```

next
  assume 2:  $m \notin \text{set } (n\#\text{ns})$ 
  thus  $\text{Mapping.lookup } u' m = \text{None}$ 
    by (simp add: aux-2(2) IH(2))
next
  fix v
  show (if  $\exists n \in \text{set } (n\#\text{ns}). m \in \text{phiDefNodes-aux } g v (\alpha n g) n \wedge v \in \text{uses } g$ 
  n then
    Some (map ( $\lambda m. \text{lookupDef } g m v$ ) (predecessors g m)) else
    None) =  $\text{Mapping.lookup } p' (m,v)$ 
  apply (auto simp: aux-2(3)[symmetric] IH(3)[symmetric] keys-dom-lookup
  dom-def)
  apply (erule phiDefNodes-auxE)
  apply (erule uses-in- $\alpha n$ )
  apply (rule phiDefNodes-auxI)
  apply auto[4]
  apply (drule phiDefNodes-aux-redirect; auto simp: uses-in- $\alpha n$ ; fail)
  apply (drule rev-subsetD)
  apply (rule phiDefNodes-aux-unvisited-monotonic)
  apply auto
  done
}
qed (auto simp: lookup-empty)

```

```

lemma fst-uses'-phis':  $\text{uses}' g = \text{lookup-multimap } (\text{fst } (\text{uses}'\text{-phis}' g))$ 
apply (rule ext)
apply (simp add: uses'-phis'-def Let-def split-def)
apply (case-tac  $x \in \text{set } (\alpha n g)$ )
  apply (rule foldr-aux-2(1)[OF - - surjective-pairing]; auto simp: lookup-empty
  intro:  $\alpha n$ -distinct; fail)
  unfolding lookup-multimap-def
  apply (subst foldr-aux-2(2)[OF - - surjective-pairing]; auto simp: lookup-empty
  uses-in- $\alpha n$  uses'-def intro:  $\alpha n$ -distinct)
done

```

```

lemma fst-uses'-phis'-in- $\alpha n$ :  $\text{Mapping.keys } (\text{fst } (\text{uses}'\text{-phis}' g)) \subseteq \text{set } (\alpha n g)$ 
apply (rule subsetI)
apply (rule ccontr)
apply (simp add: uses'-phis'-def Let-def split-def keys-dom-lookup dom-def)
apply (subst(asm) foldr-aux-2(2)[OF - - surjective-pairing]; auto intro:  $\alpha n$ -distinct)
done

```

```

lemma snd-uses'-phis':  $\text{phis}'\text{-code } g = \text{snd } (\text{uses}'\text{-phis}' g)$ 
proof -
  have a:  $\bigwedge n v. (\text{THE } k. (\lambda p. (\text{fst } p, \text{snd } p, \text{fst } p, \text{PhiDef})) - \{ (n, v, n, \text{PhiDef}) \} = \{k\}) = (n,v)$ 
  by (rule the1-equality) (auto simp: vimage-def)
  show ?thesis
  apply (subst Mapping-eq-lookup)

```

```

apply transfer
apply (simp add: phi'-def uses'-phi'-def Let-def split-def)
apply (auto simp: lookup-map-keys a intro!: ext)
subgoal by (auto simp: vimage-def)[1]
subgoal
  apply (subst foldr-aux-2(3)[OF - - surjective-pairing, symmetric])
  by (auto simp: phiDefNodes-def vimage-def elim!: fold-union-elem intro!:
 $\alpha$ n-distinct split: if-split-asm)

subgoal
  apply (subst(asm) foldr-aux-2(3)[OF - - surjective-pairing, symmetric])
  by (auto simp: phiDefNodes-def vimage-def elim!: fold-union-elem intro!:
 $\alpha$ n-distinct split: if-split-asm)

subgoal
  apply (subst(asm) foldr-aux-2(3)[OF - - surjective-pairing, symmetric])
  by (auto simp: phiDefNodes-def vimage-def elim!: fold-union-elem intro!:
 $\alpha$ n-distinct fold-union-elemI split: if-split-asm)
done
qed
end

end

```

### 6.3 Locales Transfer Rules

```

theory SSA-Transfer-Rules imports
  SSA-CFG
  Construct-SSA-code
begin

context includes lifting-syntax
begin

lemmas weak-All-transfer1 [transfer-rule] = iffD1 [OF right-total-alt-def2]
lemma weak-All-transfer2 [transfer-rule]: right-total R  $\implies$  ((R  $\implies$  > (=))  $\implies$  >
( $\longrightarrow$ ) All All
  by (auto 4 4 elim: right-totalE rel-funE)

lemma weak-imp-transfer [transfer-rule]:
  ((=)  $\implies$  > (=)  $\implies$  > ( $\longrightarrow$ )) ( $\longrightarrow$ ) ( $\longrightarrow$ )
  by auto

lemma weak-conj-transfer [transfer-rule]:
  (( $\longrightarrow$ )  $\implies$  > ( $\longrightarrow$ )  $\implies$  > ( $\longrightarrow$ )) ( $\wedge$ ) ( $\wedge$ )
  by auto

lemma graph-path-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G

```

```

and [transfer-rule]: ( $G \implies (=)$ )  $\alpha e \alpha e2$ 
and [transfer-rule]: ( $G \implies (=)$ )  $\alpha n \alpha n2$ 
and [transfer-rule]: ( $G \implies (=)$ ) invar invar2
and [transfer-rule]: ( $G \implies (=)$ ) inEdges inEdges2
shows ( $\longrightarrow$ ) (graph-path  $\alpha e \alpha n$  invar inEdges) (graph-path  $\alpha e2 \alpha n2$  invar2
inEdges2)
unfolding graph-path-def [abs-def] graph-def valid-graph-def graph-nodes-it-def
graph-pred-it-def
graph-nodes-it-axioms-def graph-pred-it-axioms-def set-iterator-def set-iterator-genord-def

foldri-def
using assms(2-5)
apply clarsimp
apply safe
apply (rule-tac y=g in right-totalE [OF assms(1)]; (erule(1) rel-funE)+;
auto)
apply (rule-tac y=g in right-totalE [OF assms(1)]; (erule(1) rel-funE)+;
force)
apply (rule-tac y=g in right-totalE [OF assms(1)]; (erule(1) rel-funE)+;
force)
apply (rule-tac y=g in right-totalE [OF assms(1)]; (erule(1) rel-funE)+;
force)
apply (rule-tac y=g in right-totalE [OF assms(1)]; (erule(1) rel-funE)+;
force)
apply (rule-tac y=g in right-totalE [OF assms(1)]; (erule(1) rel-funE)+;
force)
apply (rule-tac y=g in right-totalE [OF assms(1)]; (erule(1) rel-funE)+)
apply (erule-tac x=x in allE)+
apply clarsimp
apply (rule-tac y=g in right-totalE [OF assms(1)]; (erule(1) rel-funE)+;
force)
apply (rule-tac y=g in right-totalE [OF assms(1)]; (erule(1) rel-funE)+; force)
apply (rule-tac y=g in right-totalE [OF assms(1)]; (erule(1) rel-funE)+; force)
apply (rule-tac y=g in right-totalE [OF assms(1)]; (erule(1) rel-funE)+; force)
done

end

context graph-path-base begin

context includes lifting-syntax
begin

lemma inEdges-transfer [transfer-rule]:
assumes [transfer-rule]: right-total A
and [transfer-rule]: ( $A \implies (=)$ )  $\alpha e \alpha e2$ 
and [transfer-rule]: ( $A \implies (=)$ )  $\alpha n \alpha n2$ 
and [transfer-rule]: ( $A \implies (=)$ ) invar invar2

```

**and** [*transfer-rule*]: ( $A \implies (=)$ ) *inEdges'* *inEdges2*  
**shows** ( $A \implies (=)$ ) *inEdges* (*graph-path-base.inEdges inEdges2*)  
**proof** –  
**interpret** *gp2*: *graph-path-base*  $\alpha e2$   $\alpha n2$  *invar2 inEdges2* .  
**show** ?*thesis*  
**unfolding** *gp2.inEdges-def* [*abs-def*] *inEdges-def* [*abs-def*]  
**by** *transfer-prover*  
**qed**

**lemma** *predecessors-transfer* [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *right-total* *A*  
**and** [*transfer-rule*]: ( $A \implies (=)$ )  $\alpha e$   $\alpha e2$   
**and** [*transfer-rule*]: ( $A \implies (=)$ )  $\alpha n$   $\alpha n2$   
**and** [*transfer-rule*]: ( $A \implies (=)$ ) *invar invar2*  
**and** [*transfer-rule*]: ( $A \implies (=)$ ) *inEdges'* *inEdges2*  
**shows** ( $A \implies (=)$ ) *predecessors* (*graph-path-base.predecessors inEdges2*)  
**proof** –  
**interpret** *gp2*: *graph-path-base*  $\alpha e2$   $\alpha n2$  *invar2 inEdges2* .  
**show** ?*thesis*  
**unfolding** *gp2.predecessors-def* [*abs-def*] *predecessors-def* [*abs-def*]  
**by** *transfer-prover*  
**qed**

**lemma** *successors-transfer* [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *right-total* *A*  
**and** [*transfer-rule*]: ( $A \implies (=)$ )  $\alpha e$   $\alpha e2$   
**and** [*transfer-rule*]: ( $A \implies (=)$ )  $\alpha n$   $\alpha n2$   
**and** [*transfer-rule*]: ( $A \implies (=)$ ) *invar invar2*  
**and** [*transfer-rule*]: ( $A \implies (=)$ ) *inEdges'* *inEdges2*  
**shows** ( $A \implies (=)$ ) *successors* (*graph-path-base.successors*  $\alpha n2$  *inEdges2*)  
**proof** –  
**interpret** *gp2*: *graph-path-base*  $\alpha e2$   $\alpha n2$  *invar2 inEdges2* .  
**show** ?*thesis*  
**unfolding** *gp2.successors-def* [*abs-def*] *successors-def* [*abs-def*]  
**by** *transfer-prover*  
**qed**

**lemma** *path-transfer* [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *right-total* *A*  
**and** [*transfer-rule*]: ( $A \implies (=)$ )  $\alpha e$   $\alpha e2$   
**and** [*transfer-rule*]: ( $A \implies (=)$ )  $\alpha n$   $\alpha n2$   
**and** [*transfer-rule*]: ( $A \implies (=)$ ) *invar invar2*  
**and** [*transfer-rule*]: ( $A \implies (=)$ ) *inEdges'* *inEdges2*  
**shows** ( $A \implies (=)$ ) *path* (*graph-path-base.path*  $\alpha n2$  *invar2 inEdges2*)  
**proof** –  
**interpret** *gp2*: *graph-path-base*  $\alpha e2$   $\alpha n2$  *invar2 inEdges2* .  
**show** ?*thesis*  
**unfolding** *gp2.path-def* *path-def*  
**by** *transfer-prover*

qed

```
lemma path2-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total A
    and [transfer-rule]: (A ===> (=))  $\alpha e \alpha e2$ 
    and [transfer-rule]: (A ===> (=))  $\alpha n \alpha n2$ 
    and [transfer-rule]: (A ===> (=)) invar invar2
    and [transfer-rule]: (A ===> (=)) inEdges' inEdges2
  shows (A ===> (=)) path2 (graph-path-base.path2  $\alpha n2$  invar2 inEdges2)
proof -
  interpret gp2: graph-path-base  $\alpha e2 \alpha n2$  invar2 inEdges2 .
  show ?thesis
    unfolding gp2.path2-def [abs-def] path2-def [abs-def]
  by transfer-prover
qed
```

```
lemma weak-Ex-transfer [transfer-rule]: (((=) ===> ( $\longrightarrow$ )) ===> ( $\longrightarrow$ )) Ex Ex
  by (auto elim: rel-funE)
```

```
lemmas transfer-rules = inEdges-transfer predecessors-transfer successors-transfer
path-transfer path2-transfer
```

end

end

```
lemma graph-Entry-transfer [transfer-rule]:
  includes lifting-syntax
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G ===> (=))  $\alpha e1 \alpha e2$ 
    and [transfer-rule]: (G ===> (=))  $\alpha n1 \alpha n2$ 
    and [transfer-rule]: (G ===> (=)) invar1 invar2
    and [transfer-rule]: (G ===> (=)) inEdges1 inEdges2
    and [transfer-rule]: (G ===> (=)) Entry1 Entry2
  shows ( $\longrightarrow$ ) (graph-Entry  $\alpha e1 \alpha n1$  invar1 inEdges1 Entry1) (graph-Entry  $\alpha e2$ 
 $\alpha n2$  invar2 inEdges2 Entry2)
proof -
  {
    assume a: graph-path  $\alpha e1 \alpha n1$  invar1 inEdges1  $\wedge$  graph-Entry-axioms  $\alpha n1$ 
    invar1 inEdges1 Entry1
    then interpret graph-path  $\alpha e1 \alpha n1$  invar1 inEdges1 by simp
    have ?thesis
      unfolding graph-Entry-def [abs-def] graph-Entry-axioms-def
    by transfer-prover
  }
  thus ?thesis
    unfolding graph-Entry-def [abs-def] by simp
qed
```



**context** *graph-Entry-base* **begin**

**lemma** *dominates-transfer* [*transfer-rule*]:

**includes** *lifting-syntax*

**assumes** [*transfer-rule*]: *right-total* *G*

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ )  $\alpha e \alpha e2$

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ )  $\alpha n \alpha n2$

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *invar invar2*

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *inEdges' inEdges2*

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *Entry Entry2*

**shows** ( $G \text{ ===> } (=)$ ) *dominates* (*graph-Entry-base.dominates*  $\alpha n2$  *invar2 inEdges2 Entry2*)

**proof** –

**interpret** *gE2*: *graph-Entry-base*  $\alpha e2 \alpha n2$  *invar2 inEdges2 Entry2* .

**show** *?thesis*

**unfolding** *dominates-def* [*abs-def*] *gE2.dominates-def* [*abs-def*]

**by** *transfer-prover*

**qed**

**end**

**context** *graph-Entry* **begin**

**context includes** *lifting-syntax*

**begin**

**lemma** *shortestPath-transfer* [*transfer-rule*]:

**assumes** [*transfer-rule*]: *right-total* *G*

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ )  $\alpha e \alpha e2$

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ )  $\alpha n \alpha n2$

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *invar invar2*

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *inEdges' inEdges2*

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *Entry Entry2*

**shows** ( $G \text{ ===> } (=)$ ) *shortestPath* (*graph-Entry.shortestPath*  $\alpha n2$  *invar2 inEdges2 Entry2*)

**proof** –

**interpret** *gE2*: *graph-Entry*  $\alpha e2 \alpha n2$  *invar2 inEdges2 Entry2*

**by** *transfer' unfold-locales*

**show** *?thesis*

**unfolding** *shortestPath-def* [*abs-def*] *gE2.shortestPath-def* [*abs-def*]

**by** *transfer-prover*

**qed**

**lemma** *dominators-transfer* [*transfer-rule*]:

**assumes** [*transfer-rule*]: *right-total* *G*

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ )  $\alpha e \alpha e2$

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ )  $\alpha n \alpha n2$

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *invar invar2*

**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *inEdges' inEdges2*

```

    and [transfer-rule]: (G ==> (=)) Entry Entry2
  shows (G ==> (=)) dominators (graph-Entry.dominators  $\alpha$ n2 invar2 inEdges2 Entry2)
proof -
  interpret gE2: graph-Entry  $\alpha$ e2  $\alpha$ n2 invar2 inEdges2 Entry2
  by transfer' unfold-locales
  show ?thesis
  unfolding dominators-def [abs-def] gE2.dominators-def [abs-def]
  by transfer-prover
qed

```

```

lemma isIdom-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
  and [transfer-rule]: (G ==> (=))  $\alpha$ e  $\alpha$ e2
  and [transfer-rule]: (G ==> (=))  $\alpha$ n  $\alpha$ n2
  and [transfer-rule]: (G ==> (=)) invar invar2
  and [transfer-rule]: (G ==> (=)) inEdges' inEdges2
  and [transfer-rule]: (G ==> (=)) Entry Entry2
  shows (G ==> (=)) isIdom (graph-Entry.isIdom  $\alpha$ n2 invar2 inEdges2 Entry2)
proof -
  interpret gE2: graph-Entry  $\alpha$ e2  $\alpha$ n2 invar2 inEdges2 Entry2
  by transfer' unfold-locales
  show ?thesis
  unfolding isIdom-def [abs-def] gE2.isIdom-def [abs-def]
  by transfer-prover
qed

```

```

lemma idom-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
  and [transfer-rule]: (G ==> (=))  $\alpha$ e  $\alpha$ e2
  and [transfer-rule]: (G ==> (=))  $\alpha$ n  $\alpha$ n2
  and [transfer-rule]: (G ==> (=)) invar invar2
  and [transfer-rule]: (G ==> (=)) inEdges' inEdges2
  and [transfer-rule]: (G ==> (=)) Entry Entry2
  shows (G ==> (=)) idom (graph-Entry.idom  $\alpha$ n2 invar2 inEdges2 Entry2)
proof -
  interpret gE2: graph-Entry  $\alpha$ e2  $\alpha$ n2 invar2 inEdges2 Entry2
  by transfer' unfold-locales
  show ?thesis
  unfolding idom-def [abs-def] gE2.idom-def [abs-def]
  by transfer-prover
qed

```

```

lemmas graph-Entry-transfer =
  dominates-transfer
  shortestPath-transfer
  dominators-transfer
  isIdom-transfer
  idom-transfer

```

end

end

**lemma** *CFG-transfer* [*transfer-rule*]:  
  **includes** *lifting-syntax*  
  **assumes** [*transfer-rule*]: *right-total G*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *αe1 αe2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *αn1 αn2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *invar1 invar2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *inEdges1 inEdges2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *Entry1 Entry2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *defs1 defs2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *uses1 uses2*  
  **shows** *SSA-CFG.CFG αe1 αn1 invar1 inEdges1 Entry1 defs1 uses1*  
     $\rightarrow$  *SSA-CFG.CFG αe2 αn2 invar2 inEdges2 Entry2 defs2 uses2*  
  **unfolding** *SSA-CFG.CFG-def* [*abs-def*] *CFG-axioms-def* [*abs-def*]  
  **by** *transfer-prover*

**context** *CFG-base* **begin**

**context includes** *lifting-syntax*  
**begin**

**lemma** *vars-transfer* [*transfer-rule*]:  
  **assumes** [*transfer-rule*]: *right-total G*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *αe αe2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *αn αn2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *invar invar2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *inEdges' inEdges2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *Entry Entry2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *defs defs2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *uses uses2*  
  **shows** (*G*  $\text{====}$   $\Rightarrow$  (=)) *vars (CFG-base.vars αn2 uses2)*  
**proof** –  
  **interpret** *CFG-base2: CFG-base αe2 αn2 invar2 inEdges2 Entry2 defs2 uses2* .  
  **show** *?thesis*  
    **unfolding** *vars-def* [*abs-def*] *CFG-base2.vars-def* [*abs-def*]  
    **by** *transfer-prover*  
**qed**

**lemma** *defAss'-transfer* [*transfer-rule*]:  
  **assumes** [*transfer-rule*]: *right-total G*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *αe αe2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *αn αn2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *invar invar2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *inEdges' inEdges2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *Entry Entry2*  
    **and** [*transfer-rule*]: (*G*  $\text{====}$   $\Rightarrow$  (=)) *defs defs2*

```

    and [transfer-rule]: (G ==> (=)) uses uses2
  shows (G ==> (=)) defAss' (CFG-base.defAss'  $\alpha n2$  invar2 inEdges2 Entry2
  defs2)
proof -
  interpret CFG2: CFG-base  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2 .
  show ?thesis
    unfolding defAss'-def [abs-def] CFG2.defAss'-def [abs-def]
    by transfer-prover
qed

```

```

lemma defAss'Uses-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G ==> (=))  $\alpha e$   $\alpha e2$ 
    and [transfer-rule]: (G ==> (=))  $\alpha n$   $\alpha n2$ 
    and [transfer-rule]: (G ==> (=)) invar invar2
    and [transfer-rule]: (G ==> (=)) inEdges' inEdges2
    and [transfer-rule]: (G ==> (=)) Entry Entry2
    and [transfer-rule]: (G ==> (=)) defs defs2
    and [transfer-rule]: (G ==> (=)) uses uses2
  shows (G ==> (=)) defAss'Uses (CFG-base.defAss'Uses  $\alpha n2$  invar2 inEdges2
  Entry2 defs2 uses2)
proof -
  interpret CFG2: CFG-base  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2 .
  show ?thesis
    unfolding defAss'Uses-def [abs-def] CFG2.defAss'Uses-def [abs-def]
    by transfer-prover
qed

```

```

lemmas CFG-transfers =
  vars-transfer
  defAss'-transfer
  defAss'Uses-transfer

```

**end**

**end**

```

context includes lifting-syntax
begin

```

```

lemma CFG-Construct-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G ==> (=))  $\alpha e1$   $\alpha e2$ 
    and [transfer-rule]: (G ==> (=))  $\alpha n1$   $\alpha n2$ 
    and [transfer-rule]: (G ==> (=)) invar1 invar2
    and [transfer-rule]: (G ==> (=)) inEdges1 inEdges2
    and [transfer-rule]: (G ==> (=)) Entry1 Entry2

```

```

    and [transfer-rule]: (G ==> (=)) defs1 defs2
    and [transfer-rule]: (G ==> (=)) uses1 uses2
  shows CFG-Construct  $\alpha e1$   $\alpha n1$  invar1 inEdges1 Entry1 defs1 uses1
     $\rightarrow$  CFG-Construct  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2
  unfolding CFG-Construct-def [abs-def] by transfer-prover

lemma CFG-Construct-linorder-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G ==> (=))  $\alpha e1$   $\alpha e2$ 
    and [transfer-rule]: (G ==> (=))  $\alpha n1$   $\alpha n2$ 
    and [transfer-rule]: (G ==> (=)) invar1 invar2
    and [transfer-rule]: (G ==> (=)) inEdges1 inEdges2
    and [transfer-rule]: (G ==> (=)) Entry1 Entry2
    and [transfer-rule]: (G ==> (=)) defs1 defs2
    and [transfer-rule]: (G ==> (=)) uses1 uses2
  shows CFG-Construct-linorder  $\alpha e1$   $\alpha n1$  invar1 inEdges1 Entry1 defs1 uses1
     $\rightarrow$  CFG-Construct-linorder  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2
proof -
  {
    assume CFG-Construct-linorder  $\alpha e1$   $\alpha n1$  invar1 inEdges1 Entry1 defs1 uses1
    then interpret CFG-Construct-linorder  $\alpha e1$   $\alpha n1$  invar1 inEdges1 Entry1 defs1
    uses1 .

    have ?thesis
      unfolding CFG-Construct-linorder-def CFG-Construct-wf-def CFG-wf-def CFG-wf-axioms-def
        by transfer-prover
  }
  thus ?thesis by simp
qed

end

context CFG-Construct begin

context includes lifting-syntax
begin

lemma phiDefNodes-aux-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G ==> (=))  $\alpha e$   $\alpha e2$ 
    and [transfer-rule]: (G ==> (=))  $\alpha n$   $\alpha n2$ 
    and [transfer-rule]: (G ==> (=)) invar invar2
    and [transfer-rule]: (G ==> (=)) inEdges' inEdges2
    and [transfer-rule]: (G ==> (=)) Entry Entry2
    and [transfer-rule]: (G ==> (=)) defs defs2
    and [transfer-rule]: (G ==> (=)) uses uses2
  shows (G ==> (=)) phiDefNodes-aux (CFG-Construct.phiDefNodes-aux inEdges2 defs2)
proof -

```

```

interpret i: CFG-Construct  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2
  by transfer' unfold-locales
{ fix g1 g2 v unvisited n
  assume G g1 g2
  with assms have inEdges2 g2 = inEdges' g1 and defs2 g2 = defs g1
    by (auto elim: rel-funE)
  hence phiDefNodes-aux g1 v unvisited n = CFG-Construct.phiDefNodes-aux
inEdges2 defs2 g2 v unvisited n
  apply (induction g1 v unvisited n rule: phiDefNodes-aux.induct)
  apply (subst phiDefNodes-aux.simps)
  apply (subst i.phiDefNodes-aux.simps)
  apply (subgoal-tac i.predecessors g2 n = predecessors g n)
  prefer 2 apply (clarsimp simp: i.predecessors-def predecessors-def i.inEdges-def
inEdges-def)
  by (simp cong: if-cong arg-cong2 [where f=fold (∪)] map-cong)
}
thus ?thesis by blast
qed

```

```

lemma phiDefNodes-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G  $\implies$  (=))  $\alpha e$   $\alpha e2$ 
    and [transfer-rule]: (G  $\implies$  (=))  $\alpha n$   $\alpha n2$ 
    and [transfer-rule]: (G  $\implies$  (=)) invar invar2
    and [transfer-rule]: (G  $\implies$  (=)) inEdges' inEdges2
    and [transfer-rule]: (G  $\implies$  (=)) Entry Entry2
    and [transfer-rule]: (G  $\implies$  (=)) defs defs2
    and [transfer-rule]: (G  $\implies$  (=)) uses uses2
  shows (G  $\implies$  (=)) phiDefNodes (CFG-Construct.phiDefNodes  $\alpha n2$  inEdges2
defs2 uses2)
proof –
  interpret i: CFG-Construct  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2
    by transfer' unfold-locales
  show ?thesis
    unfolding phiDefNodes-def [abs-def] i.phiDefNodes-def [abs-def]
    by transfer-prover
qed

```

```

lemma lookupDef-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G  $\implies$  (=))  $\alpha e$   $\alpha e2$ 
    and [transfer-rule]: (G  $\implies$  (=))  $\alpha n$   $\alpha n2$ 
    and [transfer-rule]: (G  $\implies$  (=)) invar invar2
    and [transfer-rule]: (G  $\implies$  (=)) inEdges' inEdges2
    and [transfer-rule]: (G  $\implies$  (=)) Entry Entry2
    and [transfer-rule]: (G  $\implies$  (=)) defs defs2
    and [transfer-rule]: (G  $\implies$  (=)) uses uses2
  shows (G  $\implies$  (=)) lookupDef (CFG-Construct.lookupDef  $\alpha n2$  inEdges2
defs2)

```

**proof** –  
**interpret**  $i$ : *CFG-Construct*  $\alpha e2$   $\alpha n2$  *invar2* *inEdges2* *Entry2* *defs2* *uses2*  
  by *transfer'* *unfold-locales*  
{ **fix**  $g$   $g2$   $n$   $v$   
  **assume**  $G$   $g$   $g2$   
  **with** *assms* **have**  $\alpha n2$   $g2 = \alpha n$   $g$  **and** *inEdges2*  $g2 = inEdges'$   $g$  **and** *defs2*  $g2$   
= *defs*  $g$   
  **by** (*auto elim: rel-funE*)  
  **hence** *lookupDef*  $g$   $n$   $v = i.lookupDef$   $g2$   $n$   $v$   
  **apply** (*induction*  $g$   $n$   $v$  *rule: lookupDef.induct*)  
  **apply** (*subst lookupDef.simps*)  
  **apply** (*subst i.lookupDef.simps*)  
  **apply** (*subgoal-tac i.predecessors*  $g2$   $n = predecessors$   $g$   $n$ )  
  **prefer** 2 **apply** (*clarsimp simp: i.predecessors-def predecessors-def i.inEdges-def*  
*inEdges-def*)  
  **by** (*simp cong: if-cong list.case-cong*)  
}
**thus** ?thesis **by** *blast*  
**qed**

**lemma** *defs'-transfer* [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *right-total*  $G$   
  **and** [*transfer-rule*]: ( $G \implies (=)$ )  $\alpha e$   $\alpha e2$   
  **and** [*transfer-rule*]: ( $G \implies (=)$ )  $\alpha n$   $\alpha n2$   
  **and** [*transfer-rule*]: ( $G \implies (=)$ ) *invar* *invar2*  
  **and** [*transfer-rule*]: ( $G \implies (=)$ ) *inEdges'* *inEdges2*  
  **and** [*transfer-rule*]: ( $G \implies (=)$ ) *Entry* *Entry2*  
  **and** [*transfer-rule*]: ( $G \implies (=)$ ) *defs* *defs2*  
  **and** [*transfer-rule*]: ( $G \implies (=)$ ) *uses* *uses2*  
**shows** ( $G \implies (=)$ ) *defs'* (*CFG-Construct.defs'* *defs2*)

**proof** –  
**interpret**  $i$ : *CFG-Construct*  $\alpha e2$   $\alpha n2$  *invar2* *inEdges2* *Entry2* *defs2* *uses2*  
  by *transfer'* *unfold-locales*  
**show** ?thesis  
  **unfolding** *defs'-def* [*abs-def*] *i.defs'-def* [*abs-def*]  
  **by** *transfer-prover*  
**qed**

**lemma** *uses'-transfer* [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *right-total*  $G$   
  **and** [*transfer-rule*]: ( $G \implies (=)$ )  $\alpha e$   $\alpha e2$   
  **and** [*transfer-rule*]: ( $G \implies (=)$ )  $\alpha n$   $\alpha n2$   
  **and** [*transfer-rule*]: ( $G \implies (=)$ ) *invar* *invar2*  
  **and** [*transfer-rule*]: ( $G \implies (=)$ ) *inEdges'* *inEdges2*  
  **and** [*transfer-rule*]: ( $G \implies (=)$ ) *Entry* *Entry2*  
  **and** [*transfer-rule*]: ( $G \implies (=)$ ) *defs* *defs2*  
  **and** [*transfer-rule*]: ( $G \implies (=)$ ) *uses* *uses2*  
**shows** ( $G \implies (=)$ ) *uses'* (*CFG-Construct.uses'*  $\alpha n2$  *inEdges2* *defs2* *uses2*)  
**proof** –

```

interpret i: CFG-Construct  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2
  by transfer' unfold-locales
show ?thesis
  unfolding uses'-def [abs-def] i.uses'-def [abs-def]
  by transfer-prover
qed

lemma phis'-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
  and [transfer-rule]: (G  $\implies$  (=))  $\alpha e$   $\alpha e2$ 
  and [transfer-rule]: (G  $\implies$  (=))  $\alpha n$   $\alpha n2$ 
  and [transfer-rule]: (G  $\implies$  (=)) invar invar2
  and [transfer-rule]: (G  $\implies$  (=)) inEdges' inEdges2
  and [transfer-rule]: (G  $\implies$  (=)) Entry Entry2
  and [transfer-rule]: (G  $\implies$  (=)) defs defs2
  and [transfer-rule]: (G  $\implies$  (=)) uses uses2
  shows (G  $\implies$  (=)) phis' (CFG-Construct.phis'  $\alpha n2$  inEdges2 defs2 uses2)
proof –
  interpret i: CFG-Construct  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2
  by transfer' unfold-locales
  show ?thesis
  unfolding phis'-def [abs-def] i.phis'-def [abs-def]
  by transfer-prover
qed

lemmas CFG-Construct-transfer-rules =
  phiDefNodes-aux-transfer
  phiDefNodes-transfer
  lookupDef-transfer
  defs'-transfer
  uses'-transfer
  phis'-transfer
end

end

context CFG-SSA-base begin

context includes lifting-syntax
begin

lemma phiDefs-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
  and [transfer-rule]: (G  $\implies$  (=))  $\alpha e$   $\alpha e2$ 
  and [transfer-rule]: (G  $\implies$  (=))  $\alpha n$   $\alpha n2$ 
  and [transfer-rule]: (G  $\implies$  (=)) invar invar2
  and [transfer-rule]: (G  $\implies$  (=)) inEdges' inEdges2
  and [transfer-rule]: (G  $\implies$  (=)) Entry Entry2
  and [transfer-rule]: (G  $\implies$  (=)) defs defs2

```



```

    and [transfer-rule]: (G ==> (=)) uses uses2
    and [transfer-rule]: (G ==> (=))phis phis2
  shows (G ==> (=)) phiDefs (CFG-SSA-base.phiDefs phis2)
proof -
  interpret i: CFG-SSA-base  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2 phis2
.
  show ?thesis
    unfolding phiDefs-def [abs-def] i.phiDefs-def [abs-def]
    by transfer-prover
qed

lemma allDefs-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G ==> (=))  $\alpha e$   $\alpha e2$ 
    and [transfer-rule]: (G ==> (=))  $\alpha n$   $\alpha n2$ 
    and [transfer-rule]: (G ==> (=)) invar invar2
    and [transfer-rule]: (G ==> (=)) inEdges' inEdges2
    and [transfer-rule]: (G ==> (=)) Entry Entry2
    and [transfer-rule]: (G ==> (=)) defs (defs2::'a  $\Rightarrow$  'node  $\Rightarrow$  'val set)
    and [transfer-rule]: (G ==> (=)) uses (uses2::'a  $\Rightarrow$  'node  $\Rightarrow$  'val set)
    and [transfer-rule]: (G ==> (=))phis phis2
  shows (G ==> (=)) allDefs (CFG-SSA-base.allDefs defs2 phis2)
proof -
  interpret i: CFG-SSA-base  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2 phis2
.
  show ?thesis
    unfolding allDefs-def [abs-def] i.allDefs-def [abs-def]
    by transfer-prover
qed

lemma phiUses-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G ==> (=))  $\alpha e$   $\alpha e2$ 
    and [transfer-rule]: (G ==> (=))  $\alpha n$   $\alpha n2$ 
    and [transfer-rule]: (G ==> (=)) invar invar2
    and [transfer-rule]: (G ==> (=)) inEdges' inEdges2
    and [transfer-rule]: (G ==> (=)) Entry Entry2
    and [transfer-rule]: (G ==> (=)) defs defs2
    and [transfer-rule]: (G ==> (=)) uses uses2
    and [transfer-rule]: (G ==> (=))phis phis2
  shows (G ==> (=)) phiUses (CFG-SSA-base.phiUses  $\alpha n2$  inEdges2 phis2)
proof -
  interpret i: CFG-SSA-base  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2 phis2
.
  show ?thesis
    unfolding phiUses-def [abs-def] i.phiUses-def [abs-def]
    by transfer-prover
qed

```

```

lemma allUses-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G ==> (=)) αe αe2
    and [transfer-rule]: (G ==> (=)) αn αn2
    and [transfer-rule]: (G ==> (=)) invar invar2
    and [transfer-rule]: (G ==> (=)) inEdges' inEdges2
    and [transfer-rule]: (G ==> (=)) Entry Entry2
    and [transfer-rule]: (G ==> (=)) defs defs2
    and [transfer-rule]: (G ==> (=)) uses uses2
    and [transfer-rule]: (G ==> (=)) phis phis2
  shows (G ==> (=)) allUses (CFG-SSA-base.allUses αn2 inEdges2 uses2
phis2)
proof –
  interpret i: CFG-SSA-base αe2 αn2 invar2 inEdges2 Entry2 defs2 uses2 phis2
  .
  show ?thesis
    unfolding allUses-def [abs-def] i.allUses-def [abs-def]
    by transfer-prover
qed

```

```

lemma allVars-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G ==> (=)) αe αe2
    and [transfer-rule]: (G ==> (=)) αn αn2
    and [transfer-rule]: (G ==> (=)) invar invar2
    and [transfer-rule]: (G ==> (=)) inEdges' inEdges2
    and [transfer-rule]: (G ==> (=)) Entry Entry2
    and [transfer-rule]: (G ==> (=)) defs defs2
    and [transfer-rule]: (G ==> (=)) uses uses2
    and [transfer-rule]: (G ==> (=)) phis phis2
  shows (G ==> (=)) allVars (CFG-SSA-base.allVars αn2 inEdges2 defs2 uses2
phis2)
proof –
  interpret i: CFG-SSA-base αe2 αn2 invar2 inEdges2 Entry2 defs2 uses2 phis2
  .
  show ?thesis
    unfolding allVars-def [abs-def] i.allVars-def [abs-def]
    by transfer-prover
qed

```

```

lemma defAss-transfer [transfer-rule]:
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G ==> (=)) αe αe2
    and [transfer-rule]: (G ==> (=)) αn αn2
    and [transfer-rule]: (G ==> (=)) invar invar2
    and [transfer-rule]: (G ==> (=)) inEdges' inEdges2
    and [transfer-rule]: (G ==> (=)) Entry Entry2
    and [transfer-rule]: (G ==> (=)) defs defs2
    and [transfer-rule]: (G ==> (=)) uses uses2

```

```

    and [transfer-rule]: (G ==> (=)) phis phis2
  shows (G ==> (=)) defAss (CFG-SSA-base.defAss  $\alpha n2$  invar2 inEdges2 Entry2
  defs2 phis2)
proof -
  interpret i: CFG-SSA-base  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2 phis2
  .
  show ?thesis
    unfolding defAss-def [abs-def] i.defAss-def [abs-def]
    by transfer-prover
qed

lemmas CFG-SSA-base-transfer-rules =
  phiDefs-transfer
  allDefs-transfer
  phiUses-transfer
  allUses-transfer
  allVars-transfer
  defAss-transfer
end

end

context CFG-SSA-base-code begin

lemma CFG-SSA-base-code-transfer-rules [transfer-rule]:
  includes lifting-syntax
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G ==> (=))  $\alpha e$   $\alpha e2$ 
    and [transfer-rule]: (G ==> (=))  $\alpha n$   $\alpha n2$ 
    and [transfer-rule]: (G ==> (=)) invar invar2
    and [transfer-rule]: (G ==> (=)) inEdges' inEdges2
    and [transfer-rule]: (G ==> (=)) Entry Entry2
    and [transfer-rule]: (G ==> (=)) defs defs2
    and [transfer-rule]: (G ==> (=)) uses uses2
    and [transfer-rule]: (G ==> (=)) phis phis2
  shows (G ==> (=)) phiDefs (CFG-SSA-base.phiDefs ( $\lambda g$ . Mapping.lookup
  (phis2 g)))
    (G ==> (=)) allDefs (CFG-SSA-base.allDefs defs2 ( $\lambda g$ . Mapping.lookup
  (phis2 g)))
    (G ==> (=)) phiUses (CFG-SSA-base.phiUses  $\alpha n2$  inEdges2 ( $\lambda g$ . Map-
  ping.lookup (phis2 g)))
    (G ==> (=)) allUses (CFG-SSA-base.allUses  $\alpha n2$  inEdges2 (usesOf  $\circ$ 
  uses2) ( $\lambda g$ . Mapping.lookup (phis2 g)))
    (G ==> (=)) defAss (CFG-SSA-base.defAss  $\alpha n2$  invar2 inEdges2 Entry2
  defs2 ( $\lambda g$ . Mapping.lookup (phis2 g)))
  apply (simp add: CFG-SSA-base.CFG-SSA-defs[abs-def], transfer-prover)
  apply (simp add: CFG-SSA-base.CFG-SSA-defs[abs-def], transfer-prover)
  apply (simp add: CFG-SSA-base.CFG-SSA-defs[abs-def], transfer-prover)
  apply (simp add: CFG-SSA-base.CFG-SSA-defs[abs-def], transfer-prover)

```

```

apply (simp add: CFG-SSA-base.CFG-SSA-defs[abs-def], transfer-prover)
done

end

lemma CFG-SSA-transfer [transfer-rule]:
  includes lifting-syntax
  assumes [transfer-rule]: right-total G
    and [transfer-rule]: (G  $\text{====}$   $\>$  (=))  $\alpha e1$   $\alpha e2$ 
    and [transfer-rule]: (G  $\text{====}$   $\>$  (=))  $\alpha n1$   $\alpha n2$ 
    and [transfer-rule]: (G  $\text{====}$   $\>$  (=)) invar1 invar2
    and [transfer-rule]: (G  $\text{====}$   $\>$  (=)) inEdges1 inEdges2
    and [transfer-rule]: (G  $\text{====}$   $\>$  (=)) Entry1 Entry2
    and [transfer-rule]: (G  $\text{====}$   $\>$  (=)) defs1 defs2
    and [transfer-rule]: (G  $\text{====}$   $\>$  (=)) uses1 uses2
    and [transfer-rule]: (G  $\text{====}$   $\>$  (=)) phis1 phis2
  shows CFG-SSA  $\alpha e1$   $\alpha n1$  invar1 inEdges1 Entry1 defs1 uses1 phis1
     $\longrightarrow$  CFG-SSA  $\alpha e2$   $\alpha n2$  invar2 inEdges2 Entry2 defs2 uses2 phis2
  proof -
  {
    assume CFG-SSA  $\alpha e1$   $\alpha n1$  invar1 inEdges1 Entry1 defs1 uses1 phis1
    then interpret CFG-SSA  $\alpha e1$   $\alpha n1$  invar1 inEdges1 Entry1 defs1 uses1 phis1
    .
  }

  have ?thesis
  unfolding CFG-SSA-def [abs-def] CFG-SSA-axioms-def
  by transfer-prover
  }
  thus ?thesis by simp
qed

end

```

## 6.4 Code Equations for SSA Minimization

```

theory Construct-SSA-notriv-code imports
  SSA-CFG-code
  Construct-SSA-notriv
  While-Combinator-Exts
begin

abbreviation (input) const x  $\equiv$  ( $\lambda$ -. x)

context CFG-SSA-Transformed-notriv-base begin
  definition [code]: substitution-code g next = the (the-trivial (snd next) (the (phis g next)))
  definition [code]: substNext-code g next  $\equiv$   $\lambda v$ . if v = snd next then substitution-code g next else v
  definition [code]: uses'-code g next n  $\equiv$  substNext-code g next ' uses g n

```

```

lemma substNext-code-alt-def:
  substNext-code g next = id(snd next := substitution-code g next)
  unfolding substNext-code-def by auto
end

type-synonym ('g, 'node, 'val) chooseNext-code = ('node ⇒ 'val set) ⇒ ('node,
'val) phis-code ⇒ 'g ⇒ ('node × 'val)

locale CFG-SSA-Transformed-notriv-base-code =
  ssa:CFG-SSA-wf-base-code αe αn invar inEdges' Entry defs uses phis +
  CFG-SSA-Transformed-notriv-base αe αn invar inEdges' Entry oldDefs oldUses
defs usesOf ◦ uses λg. Mapping.lookup (phis g) var λuses phis. chooseNext-all uses
(Mapping.Mapping phis)
for
  αe :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set and
  αn :: 'g ⇒ 'node list and
  invar :: 'g ⇒ bool and
  inEdges' :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list and
  Entry::'g ⇒ 'node and
  oldDefs :: 'g ⇒ 'node ⇒ 'var::linorder set and
  oldUses :: 'g ⇒ 'node ⇒ 'var set and
  defs :: 'g ⇒ 'node ⇒ 'val::linorder set and
  uses :: 'g ⇒ ('node, 'val set) mapping and
  phis :: 'g ⇒ ('node, 'val) phis-code and
  var :: 'g ⇒ 'val ⇒ 'var and
  chooseNext-all :: ('g, 'node, 'val) chooseNext-code
begin
  definition [code]: cond-code g = ssa.redundant-code g

  definition uses'-codem :: 'g ⇒ 'node × 'val ⇒ 'val ⇒ ('val, 'node set) mapping
  ⇒ ('node, 'val set) mapping
  where [code]: uses'-codem g next next' nodes-of-uses =
    fold (λn. Mapping.update n (Set.insert next' (Set.remove (snd next) (the
    (Mapping.lookup (uses g) n)))))
    (sorted-list-of-set (case-option {} id (Mapping.lookup nodes-of-uses (snd
    next)))))
    (uses g)

  definition nodes-of-uses' :: 'g ⇒ 'node × 'val ⇒ 'val ⇒ 'val set ⇒ ('val, 'node
set) mapping ⇒ ('val, 'node set) mapping
  where [code]: nodes-of-uses' g next next' phiVals nodes-of-uses =
    (let users = case-option {} id (Mapping.lookup nodes-of-uses (snd next)))
    in
    if (next' ∈ phiVals) then Mapping.map-default next' {} (λns. ns ∪ users)
    (Mapping.delete (snd next) nodes-of-uses)
    else Mapping.delete (snd next) nodes-of-uses)

```

**definition** [code]: *phis'-code g next*  $\equiv$  *map-values* ( $\lambda(n,v)$  vs. if  $v = \text{snd next}$  then *None* else *Some* (*map* (*substNext-code g next*) vs)) (*phis g*)

**definition** [code]: *phis'-codem g next next' nodes-of-phis* =  
*fold* ( $\lambda n.$  *Mapping.update*  $n$  (*List.map* (*id*(*snd next* := *next'*)) (*the* (*Mapping.lookup* (*phis g*)  $n$ ))))  
(*sorted-list-of-set* (*case-option* {} (*Set.remove next*) (*Mapping.lookup nodes-of-phis* (*snd next*))))  
(*Mapping.delete next* (*phis g*))

**definition** *nodes-of-phis'* ::  $'g \Rightarrow 'node \times 'val \Rightarrow 'val \Rightarrow ('val, ('node \times 'val)$   
*set) mapping*  $\Rightarrow ('val, ('node \times 'val)$  *set) mapping*  
**where** [code]: *nodes-of-phis' g next next' nodes-of-phis* =  
(*let* *old-phis* = *Set.remove next* (*case-option* {} *id* (*Mapping.lookup nodes-of-phis* (*snd next*))));  
*nop* = *Mapping.delete* (*snd next*) *nodes-of-phis*  
*in*  
*Mapping.map-default next' {}* ( $\lambda ns.$  (*Set.remove next ns*)  $\cup$  *old-phis*) *nop*)

**definition** [code]: *triv-phis' g next triv-phis nodes-of-phis*  
= (*Set.remove next triv-phis*)  $\cup$  (*Set.filter* ( $\lambda n.$  *ssa.trivial-code* (*snd n*) (*the* (*Mapping.lookup* (*phis g*)  $n$ )))) (*case-option* {} (*Set.remove next*) (*Mapping.lookup nodes-of-phis* (*snd next*))))

**definition** [code]: *step-code g* = (*let next* = *chooseNext' g* *in* (*uses'-code g next*, *phis'-code g next*))

**definition** [code]: *step-codem g next next' nodes-of-uses nodes-of-phis* = (*uses'-codem g next next' nodes-of-uses*, *phis'-codem g next next' nodes-of-phis*)

**definition** *phi-equiv-mapping* ::  $'g \Rightarrow ('val, 'a$  *set) mapping*  $\Rightarrow ('val, 'a$  *set)*  
*mapping*  $\Rightarrow$  *bool* ( $- \vdash - \approx_{\varphi} - 50$ )

**where**  $g \vdash \text{nou}_1 \approx_{\varphi} \text{nou}_2 \equiv \forall v \in \text{Mapping.keys} (\text{ssa.phidefNodes } g).$  *case-option* {} *id* (*Mapping.lookup nou<sub>1</sub> v*) = *case-option* {} *id* (*Mapping.lookup nou<sub>2</sub> v*)  
**end**

**locale** *CFG-SSA-Transformed-notriv-linorder* = *CFG-SSA-Transformed-notriv-base*  
 $\alpha e$   $\alpha n$  *invar inEdges' Entry oldDefs oldUses defs uses phis var chooseNext-all*  
+ *CFG-SSA-Transformed-notriv*  $\alpha e$   $\alpha n$  *invar inEdges' Entry oldDefs oldUses*  
*defs uses phis var chooseNext-all*

**for**

$\alpha e$  ::  $'g \Rightarrow ('node::\text{linorder} \times 'edgeD \times 'node)$  *set* **and**  
 $\alpha n$  ::  $'g \Rightarrow 'node$  *list* **and**  
*invar* ::  $'g \Rightarrow$  *bool* **and**  
*inEdges'* ::  $'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)$  *list* **and**  
*Entry*:: $'g \Rightarrow 'node$  **and**  
*oldDefs* ::  $'g \Rightarrow 'node \Rightarrow 'var::\text{linorder}$  *set* **and**  
*oldUses* ::  $'g \Rightarrow 'node \Rightarrow 'var$  *set* **and**  
*defs* ::  $'g \Rightarrow 'node \Rightarrow 'val::\text{linorder}$  *set* **and**

```

uses :: 'g ⇒ 'node ⇒ 'val set and
phis :: 'g ⇒ ('node, 'val) phis and
var :: 'g ⇒ 'val ⇒ 'var and
chooseNext-all :: ('node ⇒ 'val set) ⇒ ('node, 'val) phis ⇒ 'g ⇒ ('node × 'val)
begin
  lemma isTrivial-the-trivial: [ phi g v = Some vs; isTrivialPhi g v v' ] ⇒
the-trivial v vs = Some v'
  by (subst the-trivialI [of vs v v']) (auto simp: isTrivialPhi-def)

  lemma the-trivial-THE-isTrivial: [ phi g v = Some vs; trivial g v ] ⇒ the-trivial
v vs = Some (The (isTrivialPhi g v))
  apply (frule isTrivialPhi-det)
  apply clarsimp
  apply (frule(1) isTrivial-the-trivial)
  by (auto dest: isTrivial-the-trivial intro!: the-equality intro: sym)

  lemma substitution-code-correct:
  assumes redundant g
  shows substitution g = substitution-code g (chooseNext' g)
  proof -
  from substitution [OF assms] have phi g (chooseNext g) ≠ None
  unfolding isTrivialPhi-def by (clarsimp split: option.splits)
  then obtain vs where phi g (chooseNext g) = Some vs by blast
  with isTrivial-the-trivial [OF this substitution [OF assms]] chooseNext'[OF
assms]
  show ?thesis unfolding substitution-code-def by (auto simp: phis-phi[of g fst
(chooseNext' g)])
  qed

  lemma substNext-code-correct:
  assumes redundant g
  shows substNext g = substNext-code g (chooseNext' g)
  unfolding substNext-def [abs-def] substNext-code-def
  by (auto simp: substitution-code-correct [OF assms])

  lemma uses'-code-correct:
  assumes redundant g
  shows uses' g = uses'-code g (chooseNext' g)
  unfolding uses'-def [abs-def] uses'-code-def [abs-def]
  by (auto simp: substNext-code-correct [OF assms])
end

context CFG-SSA-Transformed-notriv-linorder
begin
  lemma substAll-terminates: while-option (cond g) (step g) (uses g, phis g) ≠
None
  apply (rule notI)
  apply (rule while-option-None-invD [where I=inst' g and r={(y,x). (inst' g x

```

```

 $\wedge$  cond g x)  $\wedge$  y = step g x}], assumption)
  apply (rule wf-if-measure[where f= $\lambda(u,p). \text{card} (\text{dom } p)$ ])
  defer
  apply simp
  apply unfold-locales
  apply (case-tac s)
  apply (simp add: step-def cond-def)
  apply (rule step-preserves-inst [unfolded step-def, simplified], assumption+)
  apply (simp add: step-def cond-def)
  apply (clarsimp simp: cond-def step-def split:prod.split)
  proof-
    fix u p
    assume CFG-SSA-Transformed-notriv  $\alpha e \alpha n$  invar inEdges' Entry oldDefs
oldUses defs (uses(g:=u)) (phis(g:=p)) var chooseNext-all
    then interpret i: CFG-SSA-Transformed-notriv  $\alpha e \alpha n$  invar inEdges' Entry
oldDefs oldUses defs uses(g:=u) phis(g:=p) var chooseNext-all .
    assume i.redundant g
    thus card (dom (i.phis' g)) < card (dom p) by (rule i.substAll-wf[of g, sim-
plified])
  qed
end

```

```

locale CFG-SSA-Transformed-notriv-linorder-code =
  CFG-SSA-Transformed-code  $\alpha e \alpha n$  invar inEdges' Entry oldDefs oldUses defs
uses phis var
+ CFG-SSA-Transformed-notriv-base-code  $\alpha e \alpha n$  invar inEdges' Entry oldDefs
oldUses defs uses phis var chooseNext-all
+ CFG-SSA-Transformed-notriv-linorder  $\alpha e \alpha n$  invar inEdges' Entry oldDefs oldUses
defs usesOf  $\circ$  uses  $\lambda g. \text{Mapping.lookup} (\text{phis } g) \text{ var}$ 
   $\lambda \text{uses phis. chooseNext-all uses} (\text{Mapping.Mapping phis})$ 
for
 $\alpha e :: 'g \Rightarrow ('node::\text{linorder} \times 'edgeD \times 'node) \text{ set}$  and
 $\alpha n :: 'g \Rightarrow 'node \text{ list}$  and
invar :: 'g  $\Rightarrow$  bool and
inEdges' :: 'g  $\Rightarrow 'node \Rightarrow ('node \times 'edgeD) \text{ list}$  and
Entry :: 'g  $\Rightarrow 'node$  and
oldDefs :: 'g  $\Rightarrow 'node \Rightarrow 'var::\text{linorder} \text{ set}$  and
oldUses :: 'g  $\Rightarrow 'node \Rightarrow 'var \text{ set}$  and
defs :: 'g  $\Rightarrow 'node \Rightarrow 'val::\text{linorder} \text{ set}$  and
uses :: 'g  $\Rightarrow ('node, 'val \text{ set}) \text{ mapping}$  and
phis :: 'g  $\Rightarrow ('node, 'val) \text{ phis-code}$  and
var :: 'g  $\Rightarrow 'val \Rightarrow 'var$  and
chooseNext-all :: ('g, 'node, 'val) chooseNext-code
+
assumes chooseNext-all-code:
  CFG-SSA-Transformed-code  $\alpha e \alpha n$  invar inEdges' Entry oldDefs oldUses defs u
p var  $\Longrightarrow$ 
  CFG-SSA-wf-base-code.redundant-code p g  $\Longrightarrow$ 
  chooseNext-all (usesOf (u g)) (p g) g = Max (CFG-SSA-wf-base-code.trivial-phis

```



$p\ g)$

```
locale CFG-SSA-step-code =  
  step-code: CFG-SSA-Transformed-notriv-linorder-code  $\alpha e\ \alpha n$  invar inEdges' Entry oldDefs oldUses defs uses phis var chooseNext-all  
+  
  CFG-SSA-step  $\alpha e\ \alpha n$  invar inEdges' Entry oldDefs oldUses defs usesOf  $\circ$  uses  
 $\lambda g.$  Mapping.lookup (phis  $g$ ) var  $\lambda$ uses phis. chooseNext-all uses (Mapping.Mapping  
phis)  $g$   
for  
   $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)$  set and  
   $\alpha n :: 'g \Rightarrow 'node$  list and  
  invar ::  $'g \Rightarrow bool$  and  
  inEdges' ::  $'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)$  list and  
  Entry ::  $'g \Rightarrow 'node$  and  
  oldDefs ::  $'g \Rightarrow 'node \Rightarrow 'var::linorder$  set and  
  oldUses ::  $'g \Rightarrow 'node \Rightarrow 'var$  set and  
  defs ::  $'g \Rightarrow 'node \Rightarrow 'val::linorder$  set and  
  uses ::  $'g \Rightarrow ('node, 'val)$  set mapping and  
  phis ::  $'g \Rightarrow ('node, 'val)$  phis-code and  
  var ::  $'g \Rightarrow 'val \Rightarrow 'var$  and  
  chooseNext-all :: ( $'g, 'node, 'val$ ) chooseNext-code and  
   $g :: 'g$ 
```

**context** CFG-SSA-Transformed-notriv-linorder-code

**begin**

**abbreviation**  $u\text{-}g\ g\ u \equiv uses(g:=u)$

**abbreviation**  $p\text{-}g\ g\ p \equiv phis(g:=p)$

**abbreviation**  $cN \equiv (\lambda uses\ phis.\ chooseNext-all\ uses\ (Mapping.Mapping\ phis))$

**interpretation**  $uninst\text{-}code$ : CFG-SSA-Transformed-notriv-base-code  $\alpha e\ \alpha n$  invar inEdges' Entry oldDefs oldUses defs  $u\ p$  var chooseNext-all

**for**  $u\ p$

**by** *unfold-locales*

**interpretation**  $uninst$ : CFG-SSA-Transformed-notriv-base  $\alpha e\ \alpha n$  invar inEdges' Entry oldDefs oldUses defs  $u\ p$  var  $cN$

**for**  $u\ p$

**by** *unfold-locales*

**lemma** *phis'-code-correct*:

**assumes** *ssa.redundant*  $g$

**shows**  $phis'\ g = Mapping.lookup\ (phis'\text{-}code\ g\ (chooseNext'\ g))$

**unfolding** *phis'-def* [*abs-def*] *phis'-code-def* [*abs-def*]

**by** (*auto simp*: *Mapping-lookup-map-values substNext-code-correct* [*OF* *assms*])

*split*: *if-split Option.bind-split*)

**lemma** *redundant-ign[simp]*:  $uninst\text{-}code.ssa.redundant\text{-}code\ (const\ p)\ g = uninst\text{-}code.ssa.redundant\text{-}code\ (phis(g:=p))\ g$

**unfolding** *uninst-code.ssa.redundant-code-def uninst-code.ssa.trivial-code-def [abs-def]*  
*CFG-SSA-wf-base.CFG-SSA-wf-defs uninst-code.ssa.trivial-phis-def*

**unfolding** *fun-upd-same*

..

**lemma** *uses'-ign [simp]: uninst-code.uses'-codem (const u) g = uninst-code.uses'-codem (u-g g u) g*

**unfolding** *uninst-code.uses'-codem-def [abs-def] uninst.substNext-code-def uninst.substitution-code-def uninst-code.ssa.trivial-code-def [abs-def] CFG-SSA-wf-base.CFG-SSA-wf-defs*

*uninst.uses'-code-def [abs-def]*

**by** *simp*

**lemma** *phis'-ign [simp]: uninst-code.phis'-code (const p) g = uninst-code.phis'-code (phis(g:=p)) g*

**unfolding** *uninst-code.phis'-code-def [abs-def] uninst.substNext-code-def uninst.substitution-code-def uninst-code.ssa.trivial-code-def [abs-def] CFG-SSA-wf-base.CFG-SSA-wf-defs*

**unfolding** *fun-upd-same*

..

**lemma** *phis'm-ign [simp]: uninst-code.phis'-codem (const p) g = uninst-code.phis'-codem (phis(g:=p)) g*

**unfolding** *uninst-code.phis'-codem-def [abs-def] uninst.substNext-code-def uninst.substitution-code-def uninst-code.ssa.trivial-code-def [abs-def] CFG-SSA-wf-base.CFG-SSA-wf-defs*

**unfolding** *fun-upd-same*

..

**lemma** *set-sorted-list-of-set-phis-dom [simp]:*

*set (sorted-list-of-set {x ∈ dom (Mapping.lookup (phis g)). P x}) = {x ∈ dom (Mapping.lookup (phis g)). P x}*

**apply** *(subst set-sorted-list-of-set)*

**by** *(rule finite-subset [OF - ssa.phis-finite [of g]]) auto*

**lemma** *phis'-codem-correct:*

**assumes** *g ⊢ nodes-of-phis ≈<sub>φ</sub> (ssa.phiNodes-of g) and next ∈ Mapping.keys (phis g)*

**shows** *phis'-codem g next (substitution-code g next) nodes-of-phis = phis'-code g next*

**proof** –

**from** *assms*

**have** *phis'-code g next = mmap (map (substNext-code g next)) (Mapping.delete next (phis g))*

**unfolding** *phis'-code-def mmap-def phi-equiv-mapping-def*

**apply** *(subst mapping-eq-iff)*

**by** *(auto simp: Mapping-lookup-map-values Mapping-lookup-map Option.bind-def map-option-case lookup-delete keys-dom-lookup*

*dest: ssa.phis-disj [where n=fst next and v=snd next, simplified] split: option.splits)*

**also from** *assms have ... = phis'-codem g next (substitution-code g next)*

*nodes-of-phis*  
**unfolding** *phis'-codem-def mmap-def ssa.lookup-phiNodes-of* [*OF ssa.phis-finite*]  
*phi-equiv-mapping-def*  
**apply** (*subst mapping-eg-iff*)  
**apply** (*simp add: Mapping-lookup-map lookup-delete map-option-case*)  
**by** (*erule-tac x=next in ballE*)  
(*force intro!: map-idI*  
*simp: substNext-code-def keys-dom-lookup fun-upd-apply*  
*split: option.splits if-splits*)  
**finally show** *?thesis ..*  
**qed**

**lemma** *uses-transfer* [*transfer-rule*]: (*rel-fun (=) (pcr-mapping (=) (=))*) ( $\lambda g n.$   
*Mapping.lookup (uses g) n*) *uses*  
**by** (*auto simp: mapping.pcr-cr-eg cr-mapping-def Mapping.lookup.rep-eg*)

**lemma** *uses'-codem-correct*:  
**assumes**  $g \vdash \text{nodes-of-uses} \approx_{\varphi} \text{ssa.useNodes-of } g$  **and**  $\text{next} \in \text{Mapping.keys } (phis$   
 $g)$   
**shows** *usesOf (uses'-codem g next (substitution-code g next) nodes-of-uses) =*  
*uses'-code g next*  
**using** *dom-uses-in-graph [of g] assms*  
**unfolding** *uses'-codem-def uses'-code-def [abs-def]*  
**apply** (*clarsimp simp: mmap-def Mapping.replace-def [abs-def] phi-equiv-mapping-def*  
*intro!: ext*)  
**apply** (*transfer' fixing: g*)  
**apply** (*subgoal-tac  $\wedge b.$  finite*  
 $\{n. (\exists y. \text{Mapping.lookup } (uses\ g)\ n = \text{Some } y) \wedge$   
 $(\forall x2. \text{Mapping.lookup } (uses\ g)\ n = \text{Some } x2 \longrightarrow n \in \text{set } (\alpha n\ g) \wedge$   
 $b \in x2)\}$ )  
**prefer** 2  
**apply** (*rule finite-subset [where B=set (\alpha n g)]*)  
**apply** (*clarsimp simp: Mapping.keys-dom-lookup*)  
**apply** *simp*  
**by** (*auto simp: map-of-map-restrict restrict-map-def substNext-code-def fold-update-conv*  
*Mapping.keys-dom-lookup*  
*split: option.splits*)

**lemma** *step-ign*[*simp*]: *uninst-code.step-codem (const u) (const p) g = uninst-code.step-codem*  
 $(u-g\ g\ u)$  (*phis(g:=p)*)  $g$   
**by** (*rule ext*)  
(*simp add: uninst-code.step-codem-def Let-def*)

**lemma** *cN-transfer* [*transfer-rule*]: (*rel-fun (=) (rel-fun (pcr-mapping (=) (=))*  
 $(=))$ ) *cN chooseNext-all*  
**by** (*auto simp: rel-fun-def mapping.pcr-cr-eg cr-mapping-def mapping.rep-inverse*)

**lemma** *usesOf-transfer* [*transfer-rule*]: (*rel-fun (pcr-mapping (=) (=)) (=)*) ( $\lambda m$   
 $x.$  *case-option {} id (m x)*) *usesOf*  
**by** (*auto simp: rel-fun-def mapping.pcr-cr-eg cr-mapping-def Mapping.lookup.rep-eg*)

```

lemma dom-phis'-codem:
  assumes  $\bigwedge ns. \text{Mapping.lookup nodes-of-phis (snd next) = Some ns} \implies \text{finite } ns$ 
  shows  $\text{dom (Mapping.lookup (phis'-codem g next next' nodes-of-phis))} = \text{dom (Mapping.lookup (phis g))} \cup (\text{case-option } \{\} \text{ id (Mapping.lookup nodes-of-phis (snd next))}) - \{\text{next}\}$ 
  using assms unfolding phis'-codem-def
  by (auto split: if-splits option.splits simp: lookup-delete)

lemma dom-phis'-code [simp]:
  shows  $\text{dom (Mapping.lookup (phis'-code g next))} = \text{dom (Mapping.lookup (phis g))} - \{v. \text{snd } v = \text{snd next}\}$ 
  by (auto simp: phis'-code-def Mapping-lookup-map-values bind-eq-Some-conv)

lemma nodes-of-phis-finite [simplified]:
  assumes  $g \vdash \text{nodes-of-phis} \approx_{\varphi} \text{ssa.phiNodes-of } g$  and  $\text{Mapping.lookup nodes-of-phis } v = \text{Some } ns$  and  $v \in \text{Mapping.keys (ssa.phidefNodes } g)$ 
  shows finite ns
  using assms unfolding phi-equiv-mapping-def
  by (erule-tac x=v in ballE) (auto intro: finite-subset [OF - ssa.phis-finite [of g]])

lemma lookup-phis'-codem-next:
  assumes  $\bigwedge ns. \text{Mapping.lookup nodes-of-phis (snd next) = Some ns} \implies \text{finite } ns$ 
  shows  $\text{Mapping.lookup (phis'-codem g next next' nodes-of-phis) next} = \text{None}$ 
  using assms unfolding phis'-codem-def
  by (auto simp: Set.remove-def lookup-delete split: option.splits)

lemma lookup-phis'-codem-other:
  assumes  $g \vdash \text{nodes-of-phis} \approx_{\varphi} (\text{ssa.phiNodes-of } g)$ 
  and  $\text{next} \in \text{Mapping.keys (phis } g)$  and  $\text{next} \neq \varphi$ 
  shows  $\text{Mapping.lookup (phis'-codem g next (substitution-code g next) nodes-of-phis)} \varphi =$ 
     $\text{map-option (map (substNext-code g next)) (Mapping.lookup (phis } g) \varphi)$ 
  proof (cases snd next  $\neq$  snd  $\varphi$ )
  case True
    with assms(1,2) show ?thesis
    unfolding phis'-codem-correct [OF assms(1,2)] phis'-code-def
    using assms(3)
    by (auto intro!: map-idI [symmetric] simp: Mapping-lookup-map-values substNext-code-def lookup-delete map-option-case split: option.splits prod.splits)
  next
  case False
    hence  $\text{snd next} = \text{snd } \varphi$  by simp
    with assms(3) have  $\text{fst next} \neq \text{fst } \varphi$  by (cases next, cases  $\varphi$ ) auto
    with assms(2) False have [simp]:  $\text{Mapping.lookup (phis } g) \varphi = \text{None}$ 
    by (cases  $\varphi$ , cases next) (fastforce simp: keys-dom-lookup dest: ssa.phis-disj)

```

**from**  $\langle \text{fst next} \neq \text{fst } \varphi \rangle \langle \text{snd next} = \text{snd } \varphi \rangle$  **show** *?thesis*  
**unfolding** *phis'-codem-correct* [*OF assms(1,2)*] *phis'-code-def*  
**by** (*auto simp: Mapping-lookup-map-values lookup-delete map-option-case substNext-code-def split: option.splits*)  
**qed**

**lemma** *lookup-nodes-of-phis'-subst* [*simp*]:  
 $\text{Mapping.lookup (nodes-of-phis' } g \text{ next (substitution-code } g \text{ next) nodes-of-phis)}$   
 $(\text{substitution-code } g \text{ next}) =$   
 $\text{Some } ((\text{case-option } \{ \} (\text{Set.remove next}) (\text{Mapping.lookup nodes-of-phis (substitution-code}$   
 $g \text{ next}))) \cup (\text{case-option } \{ \} (\text{Set.remove next}) (\text{Mapping.lookup nodes-of-phis (snd}$   
 $\text{next}))))$   
**unfolding** *nodes-of-phis'-def*  
**by** (*clarsimp simp: Mapping-lookup-map-default Set.remove-def lookup-delete split: option.splits*)

**lemma** *lookup-nodes-of-phis'-not-subst*:  
 $v \neq \text{substitution-code } g \text{ next} \implies$   
 $\text{Mapping.lookup (nodes-of-phis' } g \text{ next (substitution-code } g \text{ next) nodes-of-phis) } v$   
 $= (\text{if } v = \text{snd next} \text{ then None else Mapping.lookup nodes-of-phis } v)$   
**unfolding** *nodes-of-phis'-def*  
**by** (*clarsimp simp: Mapping-lookup-map-default lookup-delete*)

**lemma** *lookup-phis'-code*:  
 $\text{Mapping.lookup (phis'-code } g \text{ next) } v = (\text{if } \text{snd } v = \text{snd next} \text{ then None else}$   
 $\text{map-option (map (substNext-code } g \text{ next)) (Mapping.lookup (phis } g) v))$   
**unfolding** *phis'-code-def*  
**by** (*auto simp: Mapping-lookup-map-values bind-eq-None-conv map-conv-bind-option comp-def split: prod.splits*)

**lemma** *phi-equiv-mappingE'*:  
**assumes**  $g \vdash m_1 \approx_{\varphi} \text{ssa.phiNodes-of } g$   
**and**  $\text{Mapping.lookup (phis } g) x = \text{Some } vs$  **and**  $b \in \text{set } vs$  **and**  $b \in \text{snd } \text{'}$   
 $\text{Mapping.keys (phis } g)$   
**obtains**  $\text{Mapping.lookup } m_1 \text{ } b = \text{Some } \{n \in \text{Mapping.keys (phis } g). b \in \text{set}$   
 $(\text{the (Mapping.lookup (phis } g) n))\}$   
**using** *assms unfolding phi-equiv-mapping-def*  
**apply** (*auto split: option.splits if-splits*)  
**apply** (*clarsimp simp: keys-dom-lookup*)  
**apply** (*rename-tac n \varphi-args*)  
**apply** (*erule-tac x=(n,b) in ballE*)  
**prefer** 2 **apply** *auto[1]*  
**by** (*cases x) force*

**lemma** *phi-equiv-mappingE*:  
**assumes**  $g \vdash m_1 \approx_{\varphi} \text{ssa.phiNodes-of } g$  **and**  $b \in \text{Mapping.keys (phis } g)$   
**and**  $\text{Mapping.lookup (phis } g) x = \text{Some } vs$  **and**  $\text{snd } b \in \text{set } vs$   
**obtains**  $ns$  **where**  $\text{Mapping.lookup } m_1 (\text{snd } b) = \text{Some } \{n \in \text{Mapping.keys}$   
 $(\text{phis } g). \text{snd } b \in \text{set (the (Mapping.lookup (phis } g) n))\}$

**proof** –  
**from** *assms*(2) **have** *snd*  $b \in \text{snd } \text{Mapping.keys } (\text{phis } g)$  **by** *simp*  
**with** *assms*(1,3,4) **show** *?thesis*  
**by** (*rule phi-equiv-mappingE'*) (*rule that*)  
**qed**

**lemma** *phi-equiv-mappingE2'*:  
**assumes**  $g \vdash m_1 \approx_\varphi \text{ssa.phiNodes-of } g$   
**and**  $b \in \text{snd } \text{Mapping.keys } (\text{phis } g)$   
**and**  $\forall \varphi \in \text{Mapping.keys } (\text{phis } g). b \notin \text{set } (\text{the } (\text{Mapping.lookup } (\text{phis } g) \varphi))$   
**shows**  $\text{Mapping.lookup } m_1 \ b = \text{None} \vee \text{Mapping.lookup } m_1 \ b = \text{Some } \{\}$   
**using** *assms* **unfolding** *phi-equiv-mapping-def*  
**apply** (*auto split: option.splits if-splits*)  
**apply** (*clarsimp simp: keys-dom-lookup*)  
**apply** (*rename-tac n phi-args*)  
**apply** (*erule-tac x=(n,b) in ballE*)  
**prefer** 2 **apply** *auto*[1]  
**by** (*cases Mapping.lookup m<sub>1</sub> b; auto*)

**lemma** *keys-phis'-codem [simp]: Mapping.keys (phis'-codem g next next' (ssa.phiNodes-of g)) = Mapping.keys (phis g) - {next}*  
**unfolding** *phis'-codem-def*  
**by** (*auto simp: keys-dom-lookup fun-upd-apply lookup-delete split: option.splits if-splits*)

**lemma** *keys-phis'-codem'*:  
**assumes**  $g \vdash \text{nodes-of-phis} \approx_\varphi \text{ssa.phiNodes-of } g$  **and**  $\text{next} \in \text{Mapping.keys } (\text{phis } g)$   
**shows**  $\text{Mapping.keys } (\text{phis'-codem } g \ \text{next} \ \text{next}' \ \text{nodes-of-phis}) = \text{Mapping.keys } (\text{phis } g) - \{\text{next}\}$   
**using** *assms* **unfolding** *phis'-codem-def phi-equiv-mapping-def ssa.keys-phiDefNodes [OF ssa.phis-finite]*  
**by** (*force split: option.splits if-splits simp: fold-update-conv fun-upd-apply keys-dom-lookup lookup-delete*)

**lemma** *triv-phis'-correct*:  
**assumes**  $g \vdash \text{nodes-of-phis} \approx_\varphi \text{ssa.phiNodes-of } g$  **and**  $\text{next} \in \text{Mapping.keys } (\text{phis } g)$  **and** *ssa.trivial g (snd next)*  
**shows**  $\text{uninst-code.triv-phis}' (\text{const } (\text{phis'-codem } g \ \text{next} \ (\text{substitution-code } g \ \text{next}) \ \text{nodes-of-phis})) \ g \ \text{next} \ (\text{ssa.trivial-phis } g) \ \text{nodes-of-phis} = \text{uninst-code.ssa.trivial-phis} (\text{const } (\text{phis'-codem } g \ \text{next} \ (\text{substitution-code } g \ \text{next}) \ \text{nodes-of-phis})) \ g$   
**proof** (*rule set-eqI*)  
**note** *keys-phis'-codem'* [*OF assms*(1,2), *simp*]  
**fix**  $\varphi$

**from** *assms*(2,3) *ssa.phis-in- $\alpha$ n* [*of g fst next snd next*]  
**have** *ssa.redundant g*  
**unfolding** *ssa.redundant-def ssa.allVars-def ssa.allDefs-def ssa.phiDefs-def*  
**by** (*cases next*) (*auto simp: keys-dom-lookup*)

**then interpret** *step*: *CFG-SSA-step-code*  $\alpha e \alpha n$  *invar inEdges' Entry oldDefs oldUses defs uses this var chooseNext-all*  
**by** *unfold-locales*

**let**  $?u-g = \lambda g. \text{Mapping.Mapping } (\lambda n. \text{if } \text{step.u-g } g \ n = \{\} \text{ then None else Some } (\text{step.u-g } g \ n))$   
**let**  $?p-g = \lambda g. \text{Mapping.Mapping } (\text{step.p-g } g)$

**have** *u-g-is-u-g*:  $\text{usesOf} \circ ?u-g = \text{step.u-g}$   
**by** (*subst usesOf-def [abs-def]*) (*intro ext; auto*)  
**have** *p-g-is-p-g*:  $(\lambda g. \text{Mapping.lookup } (?p-g \ g)) = \text{step.p-g}$  **by** *simp*

**interpret** *step*: *CFG-SSA-wf-code*  $\alpha e \alpha n$  *invar inEdges' Entry defs*  $\lambda g. \text{Mapping.Mapping } (\lambda n. \text{if } \text{step.u-g } g \ n = \{\} \text{ then None else Some } (\text{step.u-g } g \ n)) \lambda g. \text{Mapping.Mapping } (\text{step.p-g } g)$   
**apply** (*intro CFG-SSA-wf-code.intro CFG-SSA-code.intro*)  
**unfolding** *u-g-is-u-g p-g-is-p-g* **by** *intro-locales*

**show**  $\varphi \in \text{uninst-code.triv-phis}' (\text{const } (\text{phis'-codem } g \ \text{next } (\text{substitution-code } g \ \text{next}) \ \text{nodes-of-phis})) \ g \ \text{next} \ (\text{ssa.trivial-phis } g) \ \text{nodes-of-phis} \longleftrightarrow \varphi \in \text{uninst-code.ssa.trivial-phis} (\text{const } (\text{phis'-codem } g \ \text{next } (\text{substitution-code } g \ \text{next}) \ \text{nodes-of-phis})) \ g$   
**proof** (*cases*  $\varphi = \text{next}$ )  
**case** *True*  
**hence**  $\varphi \notin \text{uninst-code.triv-phis}' (\text{const } (\text{phis'-codem } g \ \text{next } (\text{substitution-code } g \ \text{next}) \ \text{nodes-of-phis})) \ g \ \text{next} \ (\text{ssa.trivial-phis } g) \ \text{nodes-of-phis}$   
**unfolding** *uninst-code.triv-phis'-def* **by** (*auto split: option.splits*)  
**moreover**  
**from** *True* **have**  $\varphi \notin \text{Mapping.keys } (\text{phis'-codem } g \ \text{next } (\text{substitution-code } g \ \text{next}) \ \text{nodes-of-phis})$   
**unfolding** *phis'-codem-def*  
**by** (*transfer fixing: nodes-of-phis next*) (*auto simp: fold-update-conv split: if-splits option.splits*)  
**hence**  $\varphi \notin \text{uninst-code.ssa.trivial-phis} (\text{const } (\text{phis'-codem } g \ \text{next } (\text{substitution-code } g \ \text{next}) \ \text{nodes-of-phis})) \ g$   
**unfolding** *uninst-code.ssa.trivial-phis-def* **by** *simp*  
**ultimately show** *?thesis* **by** *simp*

**next**  
**case** *False*  
**show** *?thesis*  
**proof** (*cases*  $\text{Mapping.lookup } \text{nodes-of-phis} \ (\text{snd } \text{next})$ )  
**case** *None*  
**hence**  $\text{uninst-code.triv-phis}' (\text{const } (\text{phis'-codem } g \ \text{next } (\text{substitution-code } g \ \text{next}) \ \text{nodes-of-phis})) \ g \ \text{next} \ (\text{ssa.trivial-phis } g) \ \text{nodes-of-phis} = \text{ssa.trivial-phis } g - \{\text{next}\}$   
**unfolding** *uninst-code.triv-phis'-def* **by** *auto*  
**moreover from** *None*  
**have**  $\text{uninst-code.ssa.trivial-phis} (\text{const } (\text{phis'-codem } g \ \text{next } (\text{substitution-code } g \ \text{next}) \ \text{nodes-of-phis})) \ g = \text{ssa.trivial-phis } g - \{\text{next}\}$

```

    unfolding phis'-codem-def uninst-code.ssa.trivial-phis-def by (auto simp:
lookup-delete)
    ultimately show ?thesis by simp
  next
  case [simp]: (Some nodes)
    from assms(2) have snd next ∈ snd ‘ dom (Mapping.lookup (phis g)) by
(auto simp: keys-dom-lookup)
    with assms(1) Some have finite nodes by (rule nodes-of-phis-finite)
    hence [simp]: set (sorted-list-of-set nodes) = nodes by simp
    obtain  $\varphi$ -node  $\varphi$ -val where [simp]:  $\varphi$  = ( $\varphi$ -node,  $\varphi$ -val) by (cases  $\varphi$ ) auto
    show ?thesis
    proof (cases  $\varphi$  ∈ nodes)
      case False
        with  $\langle \varphi \neq next \rangle$  have  $\varphi$  ∈ uninst-code.triv-phis' (const (phis'-codem g next
(substitution-code g next) nodes-of-phis)) g next (ssa.trivial-phis g) nodes-of-phis
 $\longleftrightarrow \varphi$  ∈ ssa.trivial-phis g
          unfolding uninst-code.triv-phis'-def by simp
          moreover

          from False  $\langle \varphi \neq next \rangle$  have ...  $\longleftrightarrow \varphi$  ∈ uninst-code.ssa.trivial-phis (const
(phis'-codem g next) (substitution-code g next) nodes-of-phis)) g
            unfolding phis'-codem-def uninst-code.ssa.trivial-phis-def
            by (auto simp add: keys-dom-lookup dom-def lookup-delete)
            ultimately show ?thesis by simp
          next
          case True
            with assms(1,2) have  $\varphi$  ∈ Mapping.keys (phis g)
            unfolding phi-equiv-mapping-def apply clarsimp
            apply (clarsimp simp: keys-dom-lookup)
            by (erule-tac x=next in ballE) (auto split: option.splits if-splits)

            then obtain  $\varphi$ -args where [simp]: Mapping.lookup (phis g) ( $\varphi$ -node,
 $\varphi$ -val) = Some  $\varphi$ -args
            unfolding keys-dom-lookup by auto
            hence [simp]: ssa.phi g  $\varphi$ -val = Some  $\varphi$ -args
            by (rule ssa.phis-phi)

            from True  $\langle \varphi \neq next \rangle$  have  $\varphi$  ∈ uninst-code.triv-phis' (const (phis'-codem g
next) (substitution-code g next) nodes-of-phis)) g next (ssa.trivial-phis g) nodes-of-phis
 $\longleftrightarrow$ 
               $\varphi$  ∈ ssa.trivial-phis g ∨ ssa.trivial-code (snd  $\varphi$ ) (the (Mapping.lookup
(phis'-codem g next) (substitution-code g next) nodes-of-phis)  $\varphi$ ))
              unfolding uninst-code.triv-phis'-def by simp
              moreover

              from  $\langle \varphi \neq next \rangle$   $\langle \varphi$  ∈ Mapping.keys (phis g)  $\langle next$  ∈ Mapping.keys (phis
g)  $\rangle$ 
                have [simp]:  $\varphi$ -val  $\neq$  snd next
                unfolding keys-dom-lookup

```



```

    by (cases next, cases  $\varphi$ ) (auto dest: ssa.phis-disj)

show ?thesis
proof (cases  $\varphi \in \text{ssa.trivial-phis } g$ )
  case True
  hence ssa.trivial-code  $\varphi$ -val  $\varphi$ -args
    unfolding ssa.trivial-phis-def by clarsimp
  hence ssa.trivial-code  $\varphi$ -val (map (substNext-code g next)  $\varphi$ -args)
    apply (rule ssa.trivial-code-mapI)
    prefer 2
    apply (clarsimp simp: substNext-code-def)
    apply (clarsimp simp: substNext-code-def substitution-code-def)
    apply (erule-tac c= $\varphi$ -val in equalityCE)
    prefer 2 apply simp
  apply clarsimp
  apply (subgoal-tac ssa.isTrivialPhi g  $\varphi$ -val (snd next))
  apply (subgoal-tac ssa.isTrivialPhi g (snd next)  $\varphi$ -val)
    apply (blast dest: isTrivialPhi-asymmetric)
  using assms(3)  $\langle \text{next} \in \text{Mapping.keys } (\text{phis } g) \rangle$ 
  apply (clarsimp simp: ssa.trivial-def keys-dom-lookup)
  apply (frule isTrivial-the-trivial [rotated 1, where v=snd next])
  apply -
  apply (rule ssa.phis-phi [where n=fst next])
  apply simp
  apply simp
  apply (thin-tac  $\varphi$ -val = v for v)
  using  $\langle \text{ssa.trivial-code } \varphi$ -val  $\varphi$ -args  $\rangle$ 
  apply (clarsimp simp: ssa.trivial-code-def)
  by (erule the-trivial-SomeE) (auto simp: ssa.isTrivialPhi-def)
  with calculation True  $\langle \varphi \neq \text{next} \rangle$   $\langle \varphi \in \text{nodes} \rangle$  show ?thesis
    unfolding uninstant-code.ssa.trivial-phis-def phis'-codem-def
    by (clarsimp simp: keys-dom-lookup substNext-code-alt-def)
  next
  case False
  with calculation  $\langle \varphi \neq \text{next} \rangle$   $\langle \varphi \in \text{Mapping.keys } (\text{phis } g) \rangle$  True show
?thesis
    unfolding phis'-codem-def uninstant-code.ssa.trivial-phis-def
    by (auto simp: keys-dom-lookup triv-phis'-def ssa.trivial-code-def)
  qed
  qed
  qed
  qed
  qed

```

**lemma nodes-of-phis'-correct:**  
**assumes**  $g \vdash \text{nodes-of-phis} \approx_{\varphi} \text{ssa.phiNodes-of } g$   
**and**  $\text{next} \in \text{Mapping.keys } (\text{phis } g)$  **and**  $\text{ssa.trivial } g$  (snd next)  
**shows**  $g \vdash (\text{nodes-of-phis}' g \text{ next } (\text{substitution-code } g \text{ next}) \text{ nodes-of-phis}) \approx_{\varphi}$   
 $(\text{uninstant-code.ssa.phiNodes-of } (\text{const } (\text{phis}'\text{-codem } g \text{ next } (\text{substitution-code } g \text{ next})))$

```

nodes-of-phis)) g)
  proof -
    from assms(2) obtain next-args where lookup-next [simp]: Mapping.lookup
      (phis g) next = Some next-args
    unfolding keys-dom-lookup by auto
    hence phi-next [simp]: ssa.phi g (snd next) = Some next-args
    by -(rule ssa.phis-phi [where n=fst next], simp)
    from assms(3) have in-next-args:  $\bigwedge v. v \in \text{set next-args} \implies v = \text{snd next} \vee$ 
       $v = \text{substitution-code } g \text{ next}$ 
    unfolding ssa.trivial-def substitution-code-def
    apply clarsimp
    apply (subst(asm) isTrivial-the-trivial)
    apply (rule ssa.phis-phi [where g=g and n=fst next])
    apply simp
    apply assumption
    by (auto simp: ssa.isTrivialPhi-def split: option.splits)
    from assms(2) have [dest!]:  $\bigwedge x \text{ vs}. \text{Mapping.lookup } (phis \ g) \ (x, \text{snd next}) =$ 
       $\text{Some vs} \implies x = \text{fst next} \wedge \text{vs} = \text{next-args}$ 
    by (auto simp add: keys-dom-lookup dest: ssa.phis-disj [where n'=fst next])
    show ?thesis
    apply (simp only: phi-equiv-mapping-def)
    apply (subgoal-tac finite (dom (Mapping.lookup (phis'-codem g next (substitution-code
      g next) nodes-of-phis))))
    prefer 2
    apply (subst dom-phis'-codem)
    apply (rule nodes-of-phis-finite [OF assms(1)], assumption)
    using assms(2) [simplified keys-dom-lookup]
    apply clarsimp
    apply (clarsimp simp: ssa.phis-finite split: option.splits)
    apply (rule nodes-of-phis-finite [OF assms(1)], assumption)
    using assms(2) [simplified keys-dom-lookup]
    apply clarsimp
    apply (simp-all only: phis'-codem-correct [OF assms(1,2)])
    apply (intro ballI)
    apply (rename-tac v)
    apply (subst(asm) ssa.keys-phidefNodes [OF ssa.phis-finite])
    apply (subst uninst-code.ssa.lookup-phiNodes-of, assumption)
    apply (subst lookup-phis'-code)+
    apply (subst substNext-code-def)+
    apply (subst dom-phis'-code)+
    apply (cases  $\exists \varphi \in \text{Mapping.keys } (phis \ g). \text{snd next} \in \text{set } (the \ (Mapping.lookup$ 
      (phis g)  $\varphi))$ )
    apply (erule bexE)
    apply (subst(asm) keys-dom-lookup)
    apply (drule domD)
    apply (erule exE)
    apply (rule phi-equiv-mappingE [OF assms(1,2)], assumption)
    apply clarsimp
    apply (cases substitution-code g next  $\in \text{snd ' Mapping.keys } (phis \ g)$ )

```

```

apply (cases  $\exists \varphi' \in \text{Mapping.keys } (\text{phis } g)$ . substitution-code  $g \text{ next} \in \text{set } (\text{the } (\text{Mapping.lookup } (\text{phis } g) \varphi'))$ )
  apply (erule bexE)
  apply (subst(asm) keys-dom-lookup)+
  apply (drule domD)
  apply (erule exE)
  apply (rule-tac  $x=\varphi'$  in phi-equiv-mappingE' [OF assms(1)], assumption)
    apply simp
    apply (simp add: keys-dom-lookup)
  apply (case-tac  $v = \text{substitution-code } g \text{ next}$ )
  apply (simp only:)
  apply (subst lookup-nodes-of-phis'-subst)
  apply (simp add: lookup-phis'-code)
  apply (auto 4 4 intro: rev-image-eqI
    simp: keys-dom-lookup map-option-case substNext-code-def split: option.splits)[1]
  apply (subst lookup-nodes-of-phis'-not-subst, assumption)
  apply (case-tac  $\exists \varphi_v \in \text{Mapping.keys } (\text{phis } g)$ .  $v \in \text{set } (\text{the } (\text{Mapping.lookup } (\text{phis } g) \varphi_v))$ )
    apply (erule bexE)
    apply (simp add: keys-dom-lookup)
    apply (drule domD)
    apply (erule exE)
    apply (rule-tac  $x=\varphi_v$  in phi-equiv-mappingE' [OF assms(1)], assumption)
      apply simp
      apply (clarsimp simp: keys-dom-lookup)
      apply (clarsimp simp: keys-dom-lookup)
      apply (rename-tac n v  $\varphi$ -args n' v' n''  $\varphi$ -args'  $\varphi$ -args'')
      apply (auto dest: in-next-args)[1]
      apply (erule-tac  $x=(n,v)$  in ballE)
      prefer 2 apply (auto dest: in-next-args)[1]
      apply auto[1]
      using phi-equiv-mappingE2' [OF assms(1), rotated 1]
      apply (erule-tac  $x=v$  in meta-allE)
      apply (erule meta-impE)
      apply clarsimp
      apply (auto simp: keys-dom-lookup)[1]
      apply force
      apply force
      using phi-equiv-mappingE2' [OF assms(1), rotated 1]
      apply (erule-tac  $x=\text{substitution-code } g \text{ next}$  in meta-allE)
      apply (erule meta-impE)
      apply clarsimp
      apply (erule meta-impE)
      apply assumption
      apply (case-tac  $v = \text{substitution-code } g \text{ next}$ )
      apply (auto simp: keys-dom-lookup)[1]
        apply force
        apply force

```

```

    apply force
    apply force
    apply force
    apply force
    apply (subst lookup-nodes-of-phis'-not-subst, assumption)
    apply (case-tac  $\exists \varphi_v \in \text{Mapping.keys } (\text{phis } g). v \in \text{set } (\text{the } (\text{Mapping.lookup } (\text{phis } g) \varphi_v))$ )
    apply (erule bexE)
    apply (simp add: keys-dom-lookup)
    apply (drule domD)
    apply (erule exE)
    apply (rule-tac  $x=\varphi_v$  in phi-equiv-mappingE' [OF assms(1)], assumption)
    apply simp
    apply (clarsimp simp: keys-dom-lookup)
    apply (auto simp: keys-dom-lookup dest: in-next-args)[1]
    apply (force dest: in-next-args)[1]
    apply (force dest: in-next-args)[1]
    using phi-equiv-mappingE2' [OF assms(1), rotated 1]
    apply (erule-tac  $x=v$  in meta-allE)
    apply (erule meta-impE)
    applyclarsimp
    apply (auto simp: keys-dom-lookup)[1]
    apply force
    apply force
    apply force
    apply force
    apply (case-tac  $v = \text{substitution-code } g \text{ next}$ )
    apply (auto simp: keys-dom-lookup)[1]
    apply (subst lookup-nodes-of-phis'-not-subst, assumption)
    apply (case-tac  $\exists \varphi_v \in \text{Mapping.keys } (\text{phis } g). v \in \text{set } (\text{the } (\text{Mapping.lookup } (\text{phis } g) \varphi_v))$ )
    apply (erule bexE)
    apply (simp add: keys-dom-lookup)
    apply (drule domD)
    apply (erule exE)
    apply (rule-tac  $x=\varphi_v$  in phi-equiv-mappingE' [OF assms(1)], assumption)
    apply simp
    apply (clarsimp simp: keys-dom-lookup)
    apply (auto simp: keys-dom-lookup dest: in-next-args)[1]
    apply (force dest: in-next-args)[1]
    using phi-equiv-mappingE2' [OF assms(1), rotated 1]
    apply (erule-tac  $x=v$  in meta-allE)
    apply (erule meta-impE)
    applyclarsimp
    apply (auto simp: keys-dom-lookup)[1]
    apply force
    apply force
    using phi-equiv-mappingE2' [OF assms(1), rotated 1]
    apply (erule-tac  $x=\text{snd next}$  in meta-allE)

```

```

apply (erule meta-impE)
apply clarsimp
apply (erule meta-impE)
using assms(2)
apply clarsimp
apply (subgoal-tac {n ∈ Mapping.keys (phis g). snd next ∈ set (the (Mapping.lookup
(phis g) n))} = {}))
prefer 2
apply auto[1]
apply (cases substitution-code g next ∈ snd ‘ Mapping.keys (phis g))
apply (cases ∃ φ' ∈ Mapping.keys (phis g). substitution-code g next ∈ set (the
(Mapping.lookup (phis g) φ'))))
apply (erule bexE)
apply (subst(asm) keys-dom-lookup)+
apply (drule domD)
apply (erule exE)
apply (rule-tac x=φ' in phi-equiv-mappingE' [OF assms(1)], assumption)
apply simp
apply (simp add: keys-dom-lookup)
apply (case-tac v = substitution-code g next)
apply (auto simp: keys-dom-lookup; force)[1]
apply (subst lookup-nodes-of-phis'-not-subst, assumption)
apply (case-tac ∃ φ_v ∈ Mapping.keys (phis g). v ∈ set (the (Mapping.lookup
(phis g) φ_v)))
apply (erule bexE)
apply (simp add: keys-dom-lookup)
apply (drule domD)
apply (erule exE)
apply (rule-tac x=φ_v in phi-equiv-mappingE' [OF assms(1)], assumption)
apply simp
apply (clarsimp simp: keys-dom-lookup)
apply (auto simp: keys-dom-lookup dest: in-next-args)[1]
apply (force dest: in-next-args)[1]
apply (force dest: in-next-args)[1]
using phi-equiv-mappingE2' [OF assms(1), rotated 1]
apply (erule-tac x=v in meta-allE)
apply (erule meta-impE)
apply clarsimp
apply (auto simp: keys-dom-lookup; force)[1]
using phi-equiv-mappingE2' [OF assms(1), rotated 1]
apply (erule-tac x=substitution-code g next in meta-allE)
apply (erule meta-impE)
apply clarsimp
apply (erule meta-impE)
apply assumption
apply (case-tac v = substitution-code g next)
apply (auto simp: keys-dom-lookup; force)[1]
apply (subst lookup-nodes-of-phis'-not-subst, assumption)
apply (case-tac ∃ φ_v ∈ Mapping.keys (phis g). v ∈ set (the (Mapping.lookup

```

```

(phis g  $\varphi_v$ ))
  apply (erule bexE)
  apply (simp add: keys-dom-lookup)
  apply (erule domD)
  apply (erule exE)
  apply (rule-tac  $x=\varphi_v$  in phi-equiv-mappingE' [OF assms(1)], assumption)
    apply simp
    apply (clarsimp simp: keys-dom-lookup)
  apply (auto simp: keys-dom-lookup dest: in-next-args; force dest: in-next-args)[1]
  using phi-equiv-mappingE2' [OF assms(1), rotated 1]
  apply (erule-tac  $x=v$  in meta-allE)
  apply (erule meta-impE)
  apply clarsimp
  apply (erule meta-impE)
  apply (clarsimp simp: keys-dom-lookup)
  apply (auto simp: keys-dom-lookup; force)[1]
  apply (case-tac  $v = \text{substitution-code } g \text{ next}$ )
  apply (auto simp: keys-dom-lookup)[1]
  apply (subst lookup-nodes-of-phis'-not-subst, assumption)
  apply (case-tac  $\exists \varphi_v \in \text{Mapping.keys } (phis\ g). v \in \text{set } (the\ (Mapping.lookup$ 
(phis g  $\varphi_v$ )))
    apply (erule bexE)
    apply (simp add: keys-dom-lookup)
    apply (erule domD)
    apply (erule exE)
    apply (rule-tac  $x=\varphi_v$  in phi-equiv-mappingE' [OF assms(1)], assumption)
      apply simp
      apply (clarsimp simp: keys-dom-lookup)
    apply (auto simp: keys-dom-lookup dest: in-next-args; force dest: in-next-args)[1]
    using phi-equiv-mappingE2' [OF assms(1), rotated 1]
    apply (erule-tac  $x=v$  in meta-allE)
    apply (erule meta-impE)
    apply clarsimp
    apply (erule meta-impE)
    apply (clarsimp simp: keys-dom-lookup)
    apply (auto simp: keys-dom-lookup; force)[1]
  done
qed

```

**lemma** *nodes-of-uses'-correct*:

```

assumes  $g \vdash \text{nodes-of-uses} \approx_{\varphi} \text{ssa.useNodes-of } g$ 
and  $\text{next} \in \text{Mapping.keys } (phis\ g)$  and ssa.trivial  $g$  (snd next)
shows  $g \vdash (\text{nodes-of-uses}'\ g \text{ next } (\text{substitution-code } g \text{ next}) (\text{Mapping.keys } (ssa.\text{phidefNodes}$ 
 $g)) \text{ nodes-of-uses}) \approx_{\varphi} (\text{uninst-code.ssa.useNodes-of } (\text{const } (\text{uses'-codem } g \text{ next}$ 
 $(\text{substitution-code } g \text{ next}) \text{ nodes-of-uses}))\ g)$ 
proof –
from assms(2,3) ssa.phis-in- $\alpha$ n [of g fst next snd next]
have ssa.redundant g
unfolding ssa.redundant-def ssa.allVars-def ssa.allDefs-def ssa.phiDefs-def

```

```

    by (cases next) (auto simp: keys-dom-lookup)

  then interpret step: CFG-SSA-step-code  $\alpha e \alpha n$  invar inEdges' Entry oldDefs
    oldUses defs usesphis var chooseNext-all
    by unfold-locales

  from assms(2,3) obtain next-args v where lookup-next [simp]: Mapping.lookup
    (phis g) next = Some next-args
    and ssa.isTrivialPhi g (snd next) v
    unfolding keys-dom-lookup ssa.trivial-def by auto
    hence phi-next [simp]: ssa.phi g (snd next) = Some next-args
    by  $\neg$ (rule ssa.phis-phi [where n=fst next], simp)
    hence the-trivial-next-args [simp]: the-trivial (snd next) next-args = Some v
  using  $\langle$ ssa.isTrivialPhi g (snd next)  $\rangle$  v
    by (rule isTrivial-the-trivial)

  from assms(3) have in-next-args:  $\bigwedge v. v \in \text{set next-args} \implies v = \text{snd next} \vee$ 
     $v = \text{substitution-code } g \text{ next}$ 
    unfolding ssa.trivial-def substitution-code-def
    apply (clarsimp simp del: the-trivial-next-args)
    apply (subst(asm) isTrivial-the-trivial)
    apply (rule ssa.phis-phi [where g=g and n=fst next])
    apply simp
    apply assumption
    by (auto simp: ssa.isTrivialPhi-def split: option.splits)

  from  $\langle$ ssa.isTrivialPhi g (snd next)  $\rangle$  v
  have triv-phi-is-v [dest!]:  $\bigwedge v'. \text{ssa.isTrivialPhi } g \text{ (snd next) } v' \implies v' = v$ 
    using isTrivialPhi-det [OF assms(3)] by auto

  from  $\langle$ ssa.isTrivialPhi g (snd next)  $\rangle$  v have [simp]:  $v \neq \text{snd next}$  unfolding
    ssa.isTrivialPhi-def by simp

  from assms(2) have [dest!]:  $\bigwedge x \text{ vs. Mapping.lookup (phis g) (x, \text{snd next}) =}$ 
     $\text{Some vs} \implies x = \text{fst next} \wedge \text{vs} = \text{next-args}$ 
    by (auto simp add: keys-dom-lookup dest: ssa.phis-disj [where n'=fst next])

  have [simp]: (CFG-base.useNodes-of  $\alpha n$ 
    (const
      (CFG-SSA-Transformed-notriv-base.uses'  $\alpha n$  defs (usesOf  $\circ$  uses)
      ( $\lambda g. \text{Mapping.lookup (phis g) } cN \text{ } g$ ))
      g)
    = (CFG-base.useNodes-of  $\alpha n$ 
      ((usesOf  $\circ$  uses)
      (g := CFG-SSA-Transformed-notriv-base.uses'  $\alpha n$  defs (usesOf  $\circ$  uses)
      ( $\lambda g. \text{Mapping.lookup (phis g) } cN \text{ } g$ )) g)
  unfolding uninst.useNodes-of-def uninst.addN-def [abs-def]
  by auto

```

```

have substNext-idem [simp]:  $\bigwedge v. \text{substNext } g (\text{substNext } g v) = \text{substNext } g v$ 
  unfolding substNext-def by (auto split: if-splits)

from assms(1)
have nodes-of-uses-eq-NoneD [elim-format, elim]:  $\bigwedge v n \text{ args. } \llbracket \text{Mapping.lookup } \text{nodes-of-uses } v = \text{None}; \text{Mapping.lookup } (\text{phis } g) (n, v) = \text{Some } \text{args} \rrbracket$ 
   $\implies (\forall n \in \text{set } (\alpha n \ g). \forall vs. \text{Mapping.lookup } (\text{uses } g) n = \text{Some } vs \longrightarrow v \notin vs)$ 
  unfolding phi-equiv-mapping-def
  apply (clarsimp simp: ssa.lookup-useNodes-of split: option.splits if-splits)
  by (erule-tac x=(n,v) in ballE) auto

from assms(1)
have nodes-of-uses-eq-SomeD [elim-format, elim]:  $\bigwedge v \text{ nodes } n \text{ args. } \llbracket \text{Mapping.lookup } \text{nodes-of-uses } v = \text{Some } \text{nodes}; \text{Mapping.lookup } (\text{phis } g) (n, v) = \text{Some } \text{args} \rrbracket$ 
   $\implies \text{nodes} = \{n \in \text{set } (\alpha n \ g). \exists vs. \text{Mapping.lookup } (\text{uses } g) n = \text{Some } vs \wedge v \in vs\}$ 
  unfolding phi-equiv-mapping-def
  apply (clarsimp simp: ssa.lookup-useNodes-of split: option.splits if-splits)
  by (erule-tac x=(n,v) in ballE) auto

show ?thesis
unfolding phi-equiv-mapping-def nodes-of-uses'-def substitution-code-def Let-def
  ssa.keys-phidefNodes [OF ssa.phis-finite]
  apply (subst o-def [where g=const g for g])
  apply (subst uses'-codem-correct [OF assms(1,2), unfolded substitution-code-def])
  apply (subst uninstant.lookup-useNodes-of)
  apply (clarsimp simp: uses'-code-def split: option.splits)
  apply (rule finite-imageI)
  using ssa.uses-finite [of g]
  apply (fastforce split: option.splits)[1]
  apply (cases v  $\in$  snd ' dom (Mapping.lookup (phis g)))
  prefer 2
  apply (force intro: rev-image-eqI simp: lookup-delete uninstant-code.uses'-code-def
  substNext-code-def substitution-code-def split: option.splits)[1]
  apply (clarsimp simp: Mapping.lookup-map-default lookup-delete uses'-code-def
  substNext-code-def substitution-code-def)
  apply (rename-tac n n' v' phi-args phi-args')
  apply safe
  apply (auto elim: nodes-of-uses-eq-SomeD [where n=fst next]
  nodes-of-uses-eq-NoneD [where n=fst next] simp:
  phi-equiv-mapping-def split: option.splits)[13]

  using assms(1)
  apply (simp add: phi-equiv-mapping-def)
  apply (erule-tac x=(n,v) in ballE)
  prefer 2 apply auto[1]

```



```

apply (auto simp: ssa.lookup-useNodes-of split: option.splits)[1]

apply (auto elim: nodes-of-uses-eq-SomeD [where n=fst next]
  nodes-of-uses-eq-NoneD [where n=fst next] split: option.splits)[1]

using assms(1)
apply (simp add: phi-equiv-mapping-def)
apply (erule-tac x=(n,v) in ballE)
prefer 2 apply auto[1]
apply (auto simp: ssa.lookup-useNodes-of split: option.splits)[1]

apply (auto 4 3 elim: nodes-of-uses-eq-SomeD [where n=fst next] split:
option.splits)[4]

using assms(1)
apply (simp add: phi-equiv-mapping-def)
apply (erule-tac x=(n',v') in ballE)
prefer 2 apply auto[1]
apply (auto simp: ssa.lookup-useNodes-of split: option.splits)[1]

by (auto split: option.splits)[1]
qed

definition[code]: substAll-efficient g ≡
  let phiVals = Mapping.keys (ssa.phidefNodes g);
    u = uses g;
    p =phis g;
    tp = ssa.trivial-phis g;
    nou = ssa.useNodes-of g;
    nop = ssa.phiNodes-of g
  in
  while
  (λ((u,p),triv-phis,nodes-of-uses,nodes-of-phis). ¬ Set.is-empty triv-phis)
  (λ((u,p),triv-phis,nodes-of-uses,nodes-of-phis). let
    next = Max triv-phis;
    next' = uninst-code.substitution-code (const p) g next;
    (u',p') = uninst-code.step-codem (const u) (const p) g next next' nodes-of-uses
nodes-of-phis;
    tp' = uninst-code.triv-phis' (const p') g next triv-phis nodes-of-phis;
    nou' = uninst-code.nodes-of-uses' g next next' phiVals nodes-of-uses;
    nop' = uninst-code.nodes-of-phis' g next next' nodes-of-phis
  in ((u', p'), tp', nou', nop'))
  ((u, p), tp, nou, nop)

abbreviation u-c x ≡ const (usesOf (fst x))
abbreviation p-c x ≡ const (Mapping.lookup (snd x))
abbreviation u g x ≡ u-g g (fst x)
abbreviation p g x ≡ p-g g (snd x)

```

**lemma** *usesOf-upd* [simp]: (*usesOf*  $\circ$  *u g s1*)(*g* := *usesOf us*) = *usesOf*  $\circ$  *u-g g us*

**by** (*auto simp: fun-upd-apply usesOf-def [abs-def] split: option.splits if-splits*)

**lemma** *keys-uses'-codem* [simp]: *Mapping.keys* (*uses'-codem g next* (*substitution-code g next*) (*ssa.useNodes-of g*)) = *Mapping.keys* (*uses g*)

**unfolding** *uses'-codem-def*

**apply** (*transfer fixing: g*)

**apply** (*auto split: option.splits if-splits simp: fold-update-conv*)

**by** (*subst(asm) sorted-list-of-set*) (*auto intro: finite-subset [OF - finite-set]*)

**lemma** *keys-uses'-codem'*:  $\llbracket g \vdash \text{nodes-of-uses} \approx_{\varphi} \text{ssa.useNodes-of } g; \text{next} \in \text{Mapping.keys (phis } g) \rrbracket$

$\implies \text{Mapping.keys (uses'-codem } g \text{ next (substitution-code } g \text{ next) nodes-of-uses)}$   
 $= \text{Mapping.keys (uses } g)$

**unfolding** *uses'-codem-def*

**apply** (*clarsimp simp: keys-dom-lookup split: if-splits option.splits*)

**apply** (*auto simp: phi-equiv-mapping-def*)

**by** (*erule-tac x=next in ballE*) (*auto simp: ssa.lookup-useNodes-of split: if-splits option.splits*)

**lemma** *triv-phis-base* [simp]: *uninst-code.ssa.trivial-phis* (*const (phis g)*) *g* = *ssa.trivial-phis g*

**unfolding** *uninst-code.ssa.trivial-phis-def ..*

**lemma** *useNodes-of-base* [simp]: *uninst-code.ssa.useNodes-of* (*const (uses g)*) *g* = *ssa.useNodes-of g*

**unfolding** *uninst-code.ssa.useNodes-of-def uninst-code.ssa.addN-def [abs-def]*  
*mmap-def Mapping.map-default-def [abs-def] Mapping.default-def*

**unfolding** *usesOf-def [abs-def]*

**by** *transfer auto*

**lemma** *phiNodes-of-base* [simp]: *uninst-code.ssa.phiNodes-of* (*const (phis g)*) *g* = *ssa.phiNodes-of g*

**unfolding** *uninst-code.ssa.phiNodes-of-def uninst-code.ssa.phis-addN-def [abs-def]*  
*mmap-def Mapping.map-default-def [abs-def] Mapping.default-def*

**by** *transfer auto*

**lemma** *phi-equiv-mapping-refl* [simp]: *uninst-code.phi-equiv-mapping* *ph g m m*

**unfolding** *uninst-code.phi-equiv-mapping-def* **by** *simp*

**lemma** *substAll-efficient-code* [code]:

*substAll g* = *map-prod usesOf Mapping.lookup* (*fst (substAll-efficient g)*)

**unfolding** *substAll-efficient-def while-def substAll-def Let-def*

**apply** –

**apply** (*rule map-option-the [OF - substAll-terminates]*)

**proof** (*rule while-option-sim*) **where**

*R* =  $\lambda x y. y = \text{map-option } (\lambda a. \text{map-prod usesOf Mapping.lookup (fst (f a))})$

*x* **and**

*I* =  $\lambda((u,p), \text{triv-phis}, \text{nodes-of-uses}, \text{phis-of-nodes}). \text{Mapping.keys } u \subseteq \text{set } (\alpha n$

```

g)  $\wedge$  Mapping.keys  $p \subseteq$  Mapping.keys (phis g)
   $\wedge$  CFG-SSA-Transformed-notriv-linorder-code  $\alpha e \alpha n$  invar inEdges' Entry
oldDefs oldUses defs (uses( $g:=u$ )) (phis( $g:=p$ )) var chooseNext-all
   $\wedge$  triv-phis = uninstant-code.ssa.trivial-phis (const p) g
   $\wedge$  uninstant-code.phi-equiv-mapping (const p) g nodes-of-uses (uninstant-code.ssa.useNodes-of
(const u) g)
   $\wedge$  uninstant-code.phi-equiv-mapping (const p) g phis-of-nodes (uninstant-code.ssa.phiNodes-of
(const p) g)
  for f
    , simplified], simp-all add: split-def dom-uses-in-graph Set.is-empty-def)
  show CFG-SSA-Transformed-notriv-linorder-code  $\alpha e \alpha n$  invar inEdges' Entry
oldDefs oldUses defs uses phis var
  chooseNext-all
  by unfold-locales
next
fix s1
  assume Mapping.keys (fst (fst s1))  $\subseteq$  set ( $\alpha n$  g)  $\wedge$  Mapping.keys (snd (fst
s1))  $\subseteq$  Mapping.keys (phis g)
   $\wedge$  CFG-SSA-Transformed-notriv-linorder-code  $\alpha e \alpha n$  invar inEdges' Entry
oldDefs oldUses defs (u g (fst s1)) (p g (fst s1)) var chooseNext-all
   $\wedge$  fst (snd s1) = uninstant-code.ssa.trivial-phis (const (snd (fst s1))) g
   $\wedge$  uninstant-code.phi-equiv-mapping (const (snd (fst s1))) g (fst (snd (snd s1)))
(uninstant-code.ssa.useNodes-of (const (fst (fst s1))) g)
   $\wedge$  uninstant-code.phi-equiv-mapping (const (snd (fst s1))) g (snd (snd (snd s1)))
(uninstant-code.ssa.phiNodes-of (const (snd (fst s1))) g)
  then obtain s1-uses s1-phis s1-triv-phis s1-nodes-of-uses s1-phi-nodes-of where
[simp]: s1 = ((s1-uses, s1-phis), s1-triv-phis, s1-nodes-of-uses, s1-phi-nodes-of)
  and Mapping.keys s1-uses  $\subseteq$  set ( $\alpha n$  g)
  and Mapping.keys s1-phis  $\subseteq$  Mapping.keys (phis g)
  and CFG-SSA-Transformed-notriv-linorder-code  $\alpha e \alpha n$  invar inEdges' Entry
oldDefs oldUses defs (u-g g s1-uses) (p-g g s1-phis) var chooseNext-all
  and [simp]: s1-triv-phis = uninstant-code.ssa.trivial-phis (const s1-phis) g
  and nou-equiv: uninstant-code.phi-equiv-mapping (const s1-phis) g s1-nodes-of-uses
(uninstant-code.ssa.useNodes-of (const s1-uses) g)
  and pno-equiv: uninstant-code.phi-equiv-mapping (const s1-phis) g s1-phi-nodes-of
(uninstant-code.ssa.phiNodes-of (const s1-phis) g)
  by (cases s1; auto)
  from this(4) interpret i: CFG-SSA-Transformed-notriv-linorder-code  $\alpha e \alpha n$ 
invar inEdges' Entry oldDefs oldUses defs u-g g s1-uses p-g g s1-phis var chooseNext-all
.
let ?s2 = map-prod usesOf Mapping.lookup (fst s1)
have [simp]: uninstant-code.ssa.trivial-phis (const s1-phis) g  $\neq$  {}  $\longleftrightarrow$  cond g ?s2
  unfolding uninstant-code.ssa.redundant-code-def [symmetric]
  by (clarsimp simp add: cond-def i.ssa.redundant-code [simplified, symmetric])
CFG-SSA-wf-base.CFG-SSA-wf-defs)
thus uninstant-code.ssa.trivial-phis (const (snd (fst s1))) g  $\neq$  {}  $\longleftrightarrow$  cond g ?s2
  by simp
{
  assume uninstant-code.ssa.trivial-phis (const (snd (fst s1))) g  $\neq$  {}

```

**hence** *red*: *uninst.redundant* (*usesOf*  $\circ$  *u-g g s1-uses*) ( $\lambda g'. \text{Mapping.lookup}$   
*(p-g g s1-phis g')*) *g*  
**by** (*simp add: cond-def uninst.CFG-SSA-wf-defs*)  
**then interpret** *step: CFG-SSA-step-code*  $\alpha e \alpha n$  *invar inEdges' Entry oldDefs*  
*oldUses defs*  
*u-g g s1-uses p-g g s1-phis var chooseNext-all g*  
**by** *unfold-locales simp*  
**from** *step.step-CFG-SSA-Transformed-notriv[simplified]*  
**interpret** *step-step: CFG-SSA-Transformed-notriv*  $\alpha e \alpha n$  *invar inEdges'*  
*Entry oldDefs oldUses defs*  
*(usesOf  $\circ$  u-g g s1-uses)(g := uninst.uses' (usesOf  $\circ$  u-g g s1-uses) ( $\lambda g'.$*   
*Mapping.lookup (p-g g s1-phis g')*) *g*)  
*( $\lambda g'. \text{Mapping.lookup (p-g g s1-phis g')}$ )(g := uninst.phis' (usesOf  $\circ$  u-g g*  
*s1-uses) ( $\lambda g'. \text{Mapping.lookup (p-g g s1-phis g')}$ ) g)*  
*var i.cN .*

**interpret** *step-step: CFG-SSA-ext*  $\alpha e \alpha n$  *invar inEdges' Entry defs*  
*(usesOf  $\circ$  u-g g s1-uses)(g := uninst.uses' (usesOf  $\circ$  (u-g g s1-uses)) ( $\lambda g'.$*   
*Mapping.lookup (p-g g s1-phis g')*) *g*)  
*( $\lambda g'. \text{Mapping.lookup (p-g g s1-phis g')}$ )(g := uninst.phis' (usesOf  $\circ$  u-g g*  
*s1-uses) ( $\lambda g'. \text{Mapping.lookup (p-g g s1-phis g')}$ ) g)*  
 ..

**from** (*Mapping.keys s1-uses*  $\subseteq$  *set* ( $\alpha n$  *g*))  
**have** *keys-u-g: Mapping.keys (u-g g s1-uses g)  $\subseteq$  set ( $\alpha n$  g)*  
**by** *clarsimp*

**have** *Max (CFG-SSA-wf-base-code.trivial-phis (p-g g s1-phis) g) = chooseNext-all*  
*(usesOf (u-g g s1-uses g)) (p-g g s1-phis g) g*  
**apply** (*rule chooseNext-all-code [where u=u-g g s1-uses, symmetric]*)  
**by** *unfold-locales (simp add: i.ssa.redundant-code [symmetric])*

**hence** [*simp*]: *Max (CFG-SSA-wf-base-code.trivial-phis (const s1-phis) g) =*  
*chooseNext-all (usesOf s1-uses) s1-phis g*  
**by** (*simp add: uninst-code.ssa.trivial-phis-def*)

**have** [*simp*]: *chooseNext-all (usesOf s1-uses) s1-phis g  $\in$  Mapping.keys s1-phis*  
**using** *i.chooseNext' [of g]*  
**by** (*clarsimp simp: Mapping.keys-dom-lookup*)

**have** [*simp*]: *uninst-code.ssa.useNodes-of (const s1-uses) g = uninst-code.ssa.useNodes-of*  
*(u-g g s1-uses) g*  
**unfolding** *uninst-code.ssa.useNodes-of-def*  
**unfolding** *uninst-code.ssa.addN-def [abs-def]*  
**by** *simp*

**have** [*simp*]: *uninst-code.ssa.phiNodes-of (const s1-phis) g = uninst-code.ssa.phiNodes-of*  
*(p-g g s1-phis) g*  
**unfolding** *uninst-code.ssa.phiNodes-of-def*

```

unfolding uninst-code.ssa.phis-addN-def [abs-def]
by simp

from (Mapping.keys s1-phis  $\subseteq$  Mapping.keys (phis g))
have finite (Mapping.keys s1-phis)
by (rule finite-subset) (auto simp: keys-dom-lookup intro: ssa.phis-finite)

hence [simp]: uninst-code.phi-equiv-mapping (const s1-phis) g = uninst-code.phi-equiv-mapping
(p-g g s1-phis) g
apply (intro ext)
apply (clarsimp simp: uninst-code.phi-equiv-mapping-def)
apply (subst uninst.keys-phidefNodes)
apply (simp add: keys-dom-lookup)
by clarsimp

have uses-conv: (usesOf  $\circ$ 
  u-g g
  (CFG-SSA-Transformed-notriv-base-code.uses'-codem (u-g g s1-uses)
   g (chooseNext-all (usesOf s1-uses) s1-phis g)
   (uninst-code.substitution-code (p-g g s1-phis) g (chooseNext-all (usesOf
s1-uses) s1-phis g)
   s1-nodes-of-uses))
  = ((usesOf  $\circ$  u-g g s1-uses)
   (g := CFG-SSA-Transformed-notriv-base.uses'  $\alpha$ n defs (usesOf  $\circ$  u-g g
s1-uses) ( $\lambda$ ga. Mapping.lookup (p-g g s1-phis ga))
   i.cN g))
unfolding i.uses'-code-correct [OF red]
apply (subst i.uses'-codem-correct [symmetric, where nodes-of-uses=s1-nodes-of-uses])
apply (rule nou-equiv [simplified])
apply auto[1]
by (auto simp: fun-upd-apply)

have phis-conv: ( $\lambda$ ga. Mapping.lookup
  (p-g g (CFG-SSA-Transformed-notriv-base-code.phis'-codem (p-g g
s1-phis) g
   (chooseNext-all (usesOf s1-uses) s1-phis g)
   (uninst-code.substitution-code (p-g g s1-phis) g (chooseNext-all
(usesOf s1-uses) s1-phis g))
   (CFG-SSA-base.phiNodes-of ( $\lambda$ ga. Mapping.lookup (p-g g
s1-phis ga)) g))
  =
  ( $\lambda$ ga. Mapping.lookup (p-g g s1-phis ga))
  (g := CFG-SSA-Transformed-notriv-base.phis'  $\alpha$ n defs (usesOf  $\circ$  u-g g
s1-uses) ( $\lambda$ ga. Mapping.lookup (p-g g s1-phis ga))
   i.cN g))
apply (subst i.phis'-code-correct [OF red])
apply (subst i.phis'-codem-correct [symmetric])
by (auto simp: fun-upd-apply)

```

```

have [simp]: uninst-code.substitution-code (const s1-phis) g = uninst-code.substitution-code
(p-g g s1-phis) g
  by (intro ext) (clarsimp simp: uninst-code.substitution-code-def)

  let ?next = Max (uninst-code.ssa.trivial-phis (const (snd (fst s1))) g)
  let ?u' = fst (uninst-code.step-codem (u g (fst s1)) (p g (fst s1)) g ?next
(uninst-code.substitution-code (const (snd (fst s1))) g ?next) (fst (snd (snd s1)))
(snd (snd (snd s1))))))
  let ?p' = snd (uninst-code.step-codem (u g (fst s1)) (p g (fst s1)) g ?next
(uninst-code.substitution-code (const (snd (fst s1))) g ?next) (fst (snd (snd s1)))
(snd (snd (snd s1))))))

  show step-s2: step g ?s2 = map-prod usesOf Mapping.lookup (uninst-code.step-codem
(u g (fst s1)) (p g (fst s1)) g
  ?next (uninst-code.substitution-code (const (snd (fst s1))) g ?next)
  (fst (snd (snd s1))) (snd (snd (snd s1))))))
  unfolding uninst-code.step-codem-def uninst.step-def split-def map-prod-def
Let-def
  apply (auto simp: map-prod-def Let-def step-step.usesOf-cache[of g, simplified]
i.phis'-codem-correct [OF pno-equiv [simplified]]
i.phis'-code-correct[simplified, OF red, simplified, symmetric]
i.uses'-codem-correct [OF nou-equiv [simplified]]
i.uses'-code-correct [OF red, symmetric, simplified])
  apply (subst uninst.uses'-def [abs-def])+
  apply (clarsimp simp: uninst.substNext-def uninst.substitution-def CFG-SSA-wf-base.CFG-SSA-wf-defs)
  apply (subst uninst.phis'-def [abs-def])+
  by (clarsimp simp: uninst.substNext-def [abs-def] uninst.substitution-def
CFG-SSA-wf-base.CFG-SSA-wf-defs cong: if-cong option.case-cong)

  have [simplified, simp]:
    uninst-code.phis'-codem (p g (fst s1)) g ?next (uninst-code.substitution-code
(const (snd (fst s1))) g ?next) s1-phi-nodes-of
    = uninst-code.phis'-codem (p g (fst s1)) g ?next (uninst-code.substitution-code
(const (snd (fst s1))) g ?next) (uninst-code.ssa.phiNodes-of (const (snd (fst s1)))
g))
    by (auto simp: i.phis'-codem-correct [OF phi-equiv-mapping-refl] i.phis'-codem-correct
[OF pno-equiv [simplified]])

  have [simplified, simp]:
    usesOf (uninst-code.uses'-codem (u g (fst s1)) g ?next (uninst-code.substitution-code
(const (snd (fst s1))) g ?next) s1-nodes-of-uses)
    = usesOf (uninst-code.uses'-codem (u g (fst s1)) g ?next (uninst-code.substitution-code
(const (snd (fst s1))) g ?next) (uninst-code.ssa.useNodes-of (const (fst (fst s1)))
g))
    by (auto simp: i.uses'-codem-correct [OF phi-equiv-mapping-refl] i.uses'-codem-correct
[OF nou-equiv [simplified]])

  from step-s2[symmetric] step.step-CFG-SSA-Transformed-notriv ⟨Mapping.keys
s1-uses ⊆ set (αn g)⟩

```

```

    ⟨Mapping.keys s1-phis ⊆ Mapping.keys (phis g)⟩
  have Mapping.keys ?u' ⊆ set (αn g) ∧
    Mapping.keys ?p' ⊆ Mapping.keys (phis g) ∧
    CFG-SSA-Transformed-notriv-linorder-code αe αn invar inEdges' Entry
oldDefs oldUses defs
  (u g (uninst-code.step-codem (u g (fst s1)) (p g (fst s1)) g ?next
(uninst-code.substitution-code (const (snd (fst s1)))) g ?next) s1-nodes-of-uses s1-phi-nodes-of))
  (p g (uninst-code.step-codem (u g (fst s1)) (p g (fst s1)) g ?next
(uninst-code.substitution-code (const (snd (fst s1)))) g ?next) s1-nodes-of-uses s1-phi-nodes-of))
  var chooseNext-all
  unfolding CFG-SSA-Transformed-notriv-linorder-code-def
    CFG-SSA-Transformed-notriv-linorder-def
    CFG-SSA-Transformed-code-def
    CFG-SSA-wf-code-def CFG-SSA-code-def
  apply (clarsimp simp: map-prod-def split-def uninst-code.step-codem-def
Let-def uses-conv phis-conv)
  apply (rule conjI)
  prefer 2
  apply (rule conjI)
  prefer 2
  apply auto[1]
  apply unfold-locales
  apply (rename-tac g')
  apply (case-tac g ≠ g')
  by (auto intro!: dom-uses-in-graph chooseNext-all-code
    simp: fun-upd-apply
      i.keys-phis'-codem' [OF pno-equiv [simplified], of ?next, simplified]
fun-upd-apply, simplified]
      i.keys-uses'-codem' [OF nou-equiv [simplified], of ?next, simplified]
fun-upd-apply, simplified])
  moreover

  have [simp]: uninst-code.ssa.trivial-phis (p-g g s1-phis) g = uninst-code.ssa.trivial-phis
(const s1-phis) g
  unfolding uninst-code.ssa.trivial-phis-def uninst-code.ssa.trivial-code-def
  by clarsimp

  from i.triv-phis'-correct [of g snd (snd (snd s1)) ?next] i.chooseNext' [of g]
  have uninst-code.triv-phis' (const ?p') g ?next s1-triv-phis (snd (snd (snd
s1)))
  = uninst-code.ssa.trivial-phis (const ?p') g
  by (auto intro: pno-equiv [simplified] simp: uninst-code.step-codem-def)
  moreover

  from ⟨Mapping.keys s1-phis ⊆ Mapping.keys (phis g)⟩ ssa.phis-finite
  have finite (dom (Mapping.lookup s1-phis))
  by (auto intro: finite-subset simp: keys-dom-lookup)
  hence phi-equiv-mapping-p'I [simplified]:
  ∧m1 m2. uninst-code.phi-equiv-mapping (const s1-phis) g m1 m2 ⇒ uninst-code.phi-equiv-mapping

```

```

(const ?p') g m1 m2
  unfolding uninstd-code.phi-equiv-mapping-def
  apply clarsimp
  apply (subst(asm) uninstd-code.keys-phidefNodes)
  apply simp
  apply (subst(asm) uninstd-code.keys-phidefNodes)
  apply (simp add: uninstd-code.step-codem-def keys-dom-lookup [symmetric])
  by (clarsimp simp: uninstd-code.step-codem-def keys-dom-lookup [symmetric])
fastforce

  have ?next ∈ Mapping.keys s1-phis by auto
  with ⟨Mapping.keys s1-phis ⊆ Mapping.keys (phis g)⟩ nou-equiv i.chooseNext'
[ of g ]
  have uninstd-code.nodes-of-uses' g ?next (uninstd-code.substitution-code (const
(snd (fst s1))) g ?next) (snd ' dom (Mapping.lookup (phis g))) (fst (snd (snd s1)))
= uninstd-code.nodes-of-uses' g ?next (uninstd-code.substitution-code (const
(snd (fst s1))) g ?next) (snd ' dom (Mapping.lookup s1-phis)) (fst (snd (snd s1)))
  unfolding uninstd-code.nodes-of-uses'-def
  apply -
  apply (erule meta-impE)
  apply auto[1]
  apply (auto simp: Let-def uninstd-code.substitution-code-def keys-dom-lookup
uninstd-code.ssa.trivial-def)
  apply (drule i.isTrivial-the-trivial [rotated 1])
  apply (rule i.ssa.phis-phi [where n=fst ?next])
  apply simp
  apply clarsimp
  apply (drule i.ssa.allVars-in-allDefs)
  apply clarsimp
  apply (drule ssa.phis-phi)
  apply clarsimp
  apply (clarsimp simp: uninstd-code.ssa.allVars-def uninstd-code.ssa.allDefs-def
uninstd-code.ssa.allUses-def uninstd-code.ssa.phiDefs-def)
  apply (erule disjE)
  apply (drule(1) ssa.simpleDef-not-phi)
  apply simp
  by (auto intro: rev-image-eqI)

ultimately
show Mapping.keys ?u' ⊆ set (αn g) ∧
Mapping.keys ?p' ⊆ Mapping.keys (phis g) ∧
CFG-SSA-Transformed-notriv-linorder-code αe αn invar inEdges' Entry
oldDefs oldUses defs
  (u g (uninstd-code.step-codem (u g (fst s1)) (p g (fst s1)) g (Max
(uninstd-code.ssa.trivial-phs (const (snd (fst s1))) g)) (uninstd-code.substitution-code
(const (snd (fst s1))) g ?next) (fst (snd (snd s1))) (snd (snd (snd s1))))))
  (p g (uninstd-code.step-codem (u g (fst s1)) (p g (fst s1)) g (Max
(uninstd-code.ssa.trivial-phs (const (snd (fst s1))) g)) (uninstd-code.substitution-code
(const (snd (fst s1))) g ?next) (fst (snd (snd s1))) (snd (snd (snd s1))))))

```



```

      var chooseNext-all  $\wedge$ 
      uninstant-code.trivial-phis' (const ?p') g ?next (uninstant-code.ssa.trivial-phis (const
(snd (fst s1))) g) (snd (snd (snd s1)))
      = uninstant-code.ssa.trivial-phis (const ?p') g  $\wedge$ 
      uninstant-code.phi-equiv-mapping (const ?p') g (uninstant-code.nodes-of-uses'
g ?next (uninstant-code.substitution-code (const (snd (fst s1))) g ?next) (snd ' dom
(Mapping.lookup (phis g))) (fst (snd (snd s1)))) (uninstant-code.ssa.useNodes-of (const
?u') g)  $\wedge$ 
      uninstant-code.phi-equiv-mapping (const ?p') g (uninstant-code.nodes-of-phis' g
?next (uninstant-code.substitution-code (const (snd (fst s1))) g ?next) (snd (snd (snd
s1)))) (uninstant-code.ssa.phiNodes-of (const ?p') g)
      using i.nodes-of-uses'-correct [of g s1-nodes-of-uses ?next, OF nou-equiv
[simplified]]
      i.chooseNext' [of g]
      i.nodes-of-phis'-correct [of g s1-phi-nodes-of ?next, OF pno-equiv [simplified]]
      apply simp
      apply (rule conjI)
      apply (rule phi-equiv-mapping-p'I)
      apply (clarsimp simp: uninstant-code.step-codem-def)
      apply (rule phi-equiv-mapping-p'I)
      by (clarsimp simp: uninstant-code.step-codem-def)
    }
  qed

end

end

```

## 6.5 Generic Code Extraction Based on typedefs

**theory** *Generic-Interpretation*

**imports**

*Construct-SSA-code*

*Construct-SSA-notriv-code*

*RBT-Mapping-Exts*

*SSA-Transfer-Rules*

*HOL-Library.RBT-Set*

*HOL-Library.Code-Target-Numeral*

**begin**

```

record ('node, 'var, 'edge) gen-cfg =
  gen-ae :: ('node, 'edge) edge set
  gen-an :: 'node list
  gen-inEdges :: 'node  $\Rightarrow$  ('node, 'edge) edge list
  gen-Entry :: 'node
  gen-defs :: 'node  $\Rightarrow$  'var set
  gen-uses :: 'node  $\Rightarrow$  'var set

```

**abbreviation** *trivial-gen-cfg ext*  $\equiv$  *gen-cfg-ext* {} [undefined] (const []) undefined

$(\text{const } \{\}) (\text{const } \{\}) \text{ ext}$   
**abbreviation**  $(\text{input}) \text{ ign } f g (-::\text{unit}) \equiv f g$

**lemma** *set-iterator-foldri-Nil* [*simp, intro!*]: *set-iterator* (*foldri* []) {}  
**by** (*rule set-iterator-I; simp add: foldri-def*)

**lemma** *set-iterator-foldri-one* [*simp, intro!*]: *set-iterator* (*foldri* [a]) {a}  
**by** (*rule set-iterator-I; simp add: foldri-def*)

**abbreviation**  $\text{gen-inEdges}' g n \equiv \text{map } (\lambda(f,d,t). (f,d)) (\text{gen-inEdges } g n)$

**lemma** *gen-cfg-inhabited*: *let*  $g = \text{trivial-gen-cfg ext in CFG-wf (ign gen-}\alpha e g)$   
 $(\text{ign gen-}\alpha n g) (\text{const True}) (\text{ign gen-inEdges}' g) (\text{ign gen-Entry } g) (\text{ign gen-defs } g)$   
 $(\text{ign gen-uses } g)$   
**apply** *auto*  
**apply** *unfold-locales*  
**by** (*auto simp: gen-cfg.defs graph-path-base.path2-def pred-def graph-path-base.inEdges-def*  
*intro!: graph-path-base.path.intros(1) exI*)

**typedef** (*'node, 'var, 'edge*) *gen-cfg-wf* = { $g :: ('node::\text{linorder}, 'var::\text{linorder}, 'edge) \text{ gen-cfg.}$   
 $\text{CFG-wf (ign gen-}\alpha e g) (\text{ign gen-}\alpha n g) (\text{const True}) (\text{ign gen-inEdges}' g) (\text{ign gen-Entry } g)$   
 $(\text{ign gen-defs } g) (\text{ign gen-uses } g)\}$   
**by** (*rule exI[where x=trivial-gen-cfg undefined]*) (*simp add: gen-cfg-inhabited[simplified]*)

**setup-lifting** *type-definition-gen-cfg-wf*

**lift-definition**  $\text{gen-wf-}\alpha n :: ('node::\text{linorder}, 'var::\text{linorder}, 'edge) \text{ gen-cfg-wf} \Rightarrow 'node \text{ list is } \text{gen-}\alpha n .$   
**lift-definition**  $\text{gen-wf-}\alpha e :: ('node::\text{linorder}, 'var::\text{linorder}, 'edge) \text{ gen-cfg-wf} \Rightarrow ('node, 'edge) \text{ edge set is } \text{gen-}\alpha e .$   
**lift-definition**  $\text{gen-wf-inEdges} :: ('node::\text{linorder}, 'var::\text{linorder}, 'edge) \text{ gen-cfg-wf} \Rightarrow 'node \Rightarrow ('node, 'edge) \text{ edge list is } \text{gen-inEdges} .$   
**lift-definition**  $\text{gen-wf-Entry} :: ('node::\text{linorder}, 'var::\text{linorder}, 'edge) \text{ gen-cfg-wf} \Rightarrow 'node \text{ is } \text{gen-Entry} .$   
**lift-definition**  $\text{gen-wf-defs} :: ('node::\text{linorder}, 'var::\text{linorder}, 'edge) \text{ gen-cfg-wf} \Rightarrow 'node \Rightarrow 'var \text{ set is } \text{gen-defs} .$   
**lift-definition**  $\text{gen-wf-uses} :: ('node::\text{linorder}, 'var::\text{linorder}, 'edge) \text{ gen-cfg-wf} \Rightarrow 'node \Rightarrow 'var \text{ set is } \text{gen-uses} .$

**abbreviation**  $\text{gen-wf-invar} \equiv \text{const True}$   
**abbreviation**  $\text{gen-wf-inEdges}' g n \equiv \text{map } (\lambda(f,d,t). (f,d)) (\text{gen-wf-inEdges } g n)$

**lemma** *gen-wf-inEdges'-transfer* [*transfer-rule*]: *rel-fun*  $\text{cr-gen-cfg-wf} (=) \text{gen-inEdges}' \text{gen-wf-inEdges}'$   
**using** *gen-wf-inEdges.transfer*  
**apply** (*auto simp: rel-fun-def cr-gen-cfg-wf-def*)  
**by** (*erule-tac x=y in allE*) *simp*

**lemma** *gen-wf-invar-trans: rel-fun cr-gen-cfg-wf (=) gen-wf-invar gen-wf-invar*  
**by** *auto*

**declare** *graph-path-base.transfer-rules*[*OF gen-cfg-wf.right-total gen-wf- $\alpha$ e.transfer*  
*gen-wf- $\alpha$ n.transfer gen-wf-invar-trans gen-wf-inEdges'-transfer, transfer-rule*]  
**declare** *CFG-base.defAss'-transfer*[*OF gen-cfg-wf.right-total gen-wf- $\alpha$ e.transfer gen-wf- $\alpha$ n.transfer*  
*gen-wf-invar-trans gen-wf-inEdges'-transfer, transfer-rule*]

**global-interpretation** *gen-wf: CFG-Construct-linorder gen-wf- $\alpha$ e gen-wf- $\alpha$ n gen-wf-invar*  
*gen-wf-inEdges' gen-wf-Entry gen-wf-defs gen-wf-uses*

**defines**

*gen-wf-predecessors = gen-wf.predecessors and*  
*gen-wf-successors = gen-wf.successors and*  
*gen-wf-defs' = gen-wf.defs' and*  
*gen-wf-vars = gen-wf.vars and*  
*gen-wf-var = gen-wf.var and*  
*gen-wf-readVariableRecursive = gen-wf.readVariableRecursive and*  
*gen-wf-readArgs = gen-wf.readArgs and*  
*gen-wf-uses'-phis' = gen-wf.uses'-phis'*

**apply** *unfold-locales*

**apply** (*transfer, simp add: CFG-Construct-wf-def CFG-wf-def CFG-def*  
*graph-Entry-def graph-path-def graph-Entry-axioms-def*)

**apply** (*transfer, simp add: CFG-Construct-wf-def CFG-wf-def CFG-def*  
*graph-Entry-def graph-path-def graph-def*)

**apply** (*transfer, simp add: CFG-Construct-wf-def CFG-wf-def CFG-def*  
*graph-Entry-def graph-path-def graph-def valid-graph-def*)

**apply** (*transfer, simp add: CFG-Construct-wf-def CFG-wf-def CFG-def*  
*graph-Entry-def graph-path-def graph-Entry-axioms-def graph-def valid-graph-def*)

**apply** *simp*

**apply** (*rule set-iterator-foldri-correct*)

**apply** (*transfer, clarsimp simp add: CFG-Construct-wf-def CFG-wf-def*  
*CFG-def graph-Entry-def*)

**apply** (*drule graph-path. $\alpha$ n-distinct; simp*)

**apply** (*transfer, clarsimp simp: CFG-Construct-wf-def CFG-wf-def CFG-def*  
*graph-Entry-def graph-path-def graph-pred-it-def graph-pred-it-axioms-def*)

**apply** (*transfer, clarsimp simp: CFG-Construct-wf-def CFG-wf-def CFG-def*  
*graph-Entry-def graph-Entry-axioms-def*)

**apply** (*transfer, clarsimp simp: CFG-Construct-wf-def CFG-wf-def CFG-def*  
*graph-Entry-def graph-Entry-axioms-def graph-path-base.inEdges-def*)

**apply** (*transfer, clarsimp simp: CFG-Construct-wf-def CFG-wf-def CFG-def*  
*graph-Entry-def graph-Entry-axioms-def graph-path-base.path2-def graph-path-base.path-def*  
*graph-path-base.predecessors-def graph-path-base.inEdges-def*)

**apply** (*transfer, simp only: CFG-Construct-wf-def CFG-wf-def CFG-def CFG-axioms-def*)

**apply** (*transfer, simp only: CFG-Construct-wf-def CFG-wf-def CFG-def CFG-axioms-def*)

**apply** (*transfer, simp only: CFG-Construct-wf-def CFG-wf-def CFG-def CFG-axioms-def*)

**apply** (*transfer, simp only: CFG-Construct-wf-def CFG-wf-def CFG-def CFG-axioms-def*)

**apply** *simp*

**by** (*transfer, clarsimp simp: CFG-Construct-wf-def CFG-wf-def CFG-wf-axioms-def*)

*CFG-base.defAss'-def [abs-def]*  
*graph-path-base.path2-def graph-path-base.path-def graph-path-base.predecessors-def*  
*graph-path-base.inEdges-def)*

**record** ('node, 'var, 'edge, 'val) *gen-ssa-cfg* = ('node, 'var, 'edge) *gen-cfg* +  
*gen-ssa-defs* :: 'node ⇒ 'val set  
*gen-ssa-uses* :: ('node, 'val set) mapping  
*gen-phis* :: ('node, 'val) phis-code  
*gen-var* :: 'val ⇒ 'var

**typedef** ('node, 'var, 'edge, 'val) *gen-ssa-cfg-wf* = {g :: ('node::linorder, 'var::linorder,  
'edge, 'val::linorder) *gen-ssa-cfg*.

*CFG-SSA-Transformed-code (ign gen-αe g) (ign gen-αn g) (const True) (ign*  
*gen-inEdges' g) (ign gen-Entry g) (ign gen-defs g) (ign gen-uses g) (ign gen-ssa-defs*  
*g) (ign gen-ssa-uses g) (ign gen-phis g) (ign gen-var g)}*

**apply** (rule *exI*[**where** *x = trivial-gen-cfg* (| *gen-ssa-defs = const {}*, *gen-ssa-uses*  
= *Mapping.empty*, *gen-phis = Mapping.empty*, *gen-var = undefined*, ... = *unde-*  
*fined* |)])

**apply** *auto*

**apply** *unfold-locales*

**by** (*auto simp: gen-cfg.defs graph-path-base.path2-def dom-def Mapping.lookup-empty*

*CFG-SSA-base.CFG-SSA-defs pred-def graph-path-base.inEdges-def intro!: graph-path-base.path.intros(1)*  
*exI*)

**setup-lifting** *type-definition-gen-ssa-cfg-wf*

**lift-definition** *gen-ssa-wf-αn* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)  
*gen-ssa-cfg-wf* ⇒ 'node list **is** *gen-αn* .

**lift-definition** *gen-ssa-wf-αe* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)  
*gen-ssa-cfg-wf* ⇒ ('node, 'edge) edge set **is** *gen-αe* .

**lift-definition** *gen-ssa-wf-inEdges* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)  
*gen-ssa-cfg-wf* ⇒ 'node ⇒ ('node, 'edge) edge list **is** *gen-inEdges* .

**lift-definition** *gen-ssa-wf-Entry* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)  
*gen-ssa-cfg-wf* ⇒ 'node **is** *gen-Entry* .

**lift-definition** *gen-ssa-wf-defs* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)  
*gen-ssa-cfg-wf* ⇒ 'node ⇒ 'var set **is** *gen-defs* .

**lift-definition** *gen-ssa-wf-uses* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)  
*gen-ssa-cfg-wf* ⇒ 'node ⇒ 'var set **is** *gen-uses* .

**lift-definition** *gen-ssa-wf-ssa-defs* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)  
*gen-ssa-cfg-wf* ⇒ 'node ⇒ 'val set **is** *gen-ssa-defs* .

**lift-definition** *gen-ssa-wf-ssa-uses* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)  
*gen-ssa-cfg-wf* ⇒ ('node, 'val set) mapping **is** *gen-ssa-uses* .

**lift-definition** *gen-ssa-wf-phis* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)  
*gen-ssa-cfg-wf* ⇒ ('node, 'val) phis-code **is** *gen-phis* .

**lift-definition** *gen-ssa-wf-var* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)  
*gen-ssa-cfg-wf* ⇒ 'val ⇒ 'var **is** *gen-var* .

**abbreviation** *gen-ssa-wf-inEdges' g n* ≡ *map* (λ(*f, d, t*). (*f, d*)) (*gen-ssa-wf-inEdges*  
*g n*)

**lemma** *gen-ssa-wf-inEdges'-transfer* [*transfer-rule*]: *rel-fun cr-gen-ssa-cfg-wf (=)*  
*gen-inEdges' gen-ssa-wf-inEdges'*  
**using** *gen-ssa-wf-inEdges.transfer*  
**apply** (*auto simp: rel-fun-def cr-gen-cfg-wf-def*)  
**by** (*erule-tac x=x in allE*) *simp*

**global-interpretation** *uninst: CFG-SSA-wf-base-code gen-ssa-wf- $\alpha$ e gen-ssa-wf- $\alpha$ n*  
*gen-wf-invar gen-ssa-wf-inEdges' gen-ssa-wf-Entry gen-ssa-wf-ssa-defs u p*  
**for** *u* **and** *p*  
**defines**  
*uninst-predecessors = uninst.predecessors*  
**and** *uninst-successors = uninst.successors*  
**and** *uninst-phiDefs = uninst.phiDefs*  
**and** *uninst-phiUses = uninst.phiUses*  
**and** *uninst-allDefs = uninst.allDefs*  
**and** *uninst-allUses = uninst.allUses*  
**and** *uninst-allVars = uninst.allVars*  
**and** *uninst-isTrivialPhi = uninst.isTrivialPhi*  
**and** *uninst-trivial = uninst.trivial-code*  
**and** *uninst-redundant = uninst.redundant-code*  
**and** *uninst-phi = uninst.phi*  
**and** *uninst-defNode = uninst.defNode*  
**and** *uninst-trivial-phis = uninst.trivial-phis*  
**and** *uninst-phidefNodes = uninst.phidefNodes*  
**and** *uninst-useNodes-of = uninst.useNodes-of*  
**and** *uninst-phiNodes-of = uninst.phiNodes-of*  
**.**

**definition** *uninst-chooseNext u p g*  $\equiv$  *Max (uninst-trivial-phis (const p) g)*

**lemma** *gen-ssa-wf-invar-trans: rel-fun cr-gen-ssa-cfg-wf (=) gen-wf-invar gen-wf-invar*  
**by** *auto*

**declare** *graph-path-base.transfer-rules*[*OF gen-ssa-cfg-wf.right-total gen-ssa-wf- $\alpha$ e.transfer*  
*gen-ssa-wf- $\alpha$ n.transfer gen-ssa-wf-invar-trans gen-ssa-wf-inEdges'-transfer, transfer-rule*]  
**declare** *CFG-base.defAss'-transfer*[*OF gen-ssa-cfg-wf.right-total gen-ssa-wf- $\alpha$ e.transfer*  
*gen-ssa-wf- $\alpha$ n.transfer gen-ssa-wf-invar-trans gen-ssa-wf-inEdges'-transfer, transfer-rule*]  
**declare** *CFG-SSA-base-code.CFG-SSA-base-code-transfer-rules*[*OF gen-ssa-cfg-wf.right-total*  
*gen-ssa-wf- $\alpha$ e.transfer gen-ssa-wf- $\alpha$ n.transfer gen-ssa-wf-invar-trans gen-ssa-wf-inEdges'-transfer*  
*gen-ssa-wf-Entry.transfer gen-ssa-wf-ssa-defs.transfer gen-ssa-wf-ssa-uses.transfer*  
*gen-ssa-wf-phis.transfer, transfer-rule*]

**lemma** *path2-ign[simp]: graph-path-base.path2 (ign gen- $\alpha$ n g) gen-wf-invar (ign*  
*gen-inEdges' g) g' n ns m  $\longleftrightarrow$  graph-path-base.path2 gen- $\alpha$ n gen-wf-invar gen-inEdges'*  
*g n ns m*  
**by** (*simp add: graph-path-base.path2-def graph-path-base.path-def graph-path-base.predecessors-def*  
*graph-path-base.inEdges-def*)

**lemma** *allDefs-ign[simp]*: *CFG-SSA-base.allDefs* (*ign gen-ssa-defs g*) (*ign Mapping.lookup (gen-phis g)*) *ga n = CFG-SSA-base.allDefs gen-ssa-defs* ( $\lambda g. Mapping.lookup (gen-phis g)$ ) *g n*  
**by** (*simp add: CFG-SSA-base.CFG-SSA-defs*)  
**lemma** *successors-ign[simp]*: *graph-path-base.successors* (*ign gen- $\alpha n$  g*) (*ign gen-inEdges'* *g*) *ga n = graph-path-base.successors gen- $\alpha n$  gen-inEdges'* *g n*  
**by** (*simp add: graph-path-base.successors-def graph-path-base.predecessors-def graph-path-base.inEdges-def*)  
**lemma** *predecessors-ign[simp]*: *graph-path-base.predecessors* (*ign gen-inEdges'* *g*) *ga n = graph-path-base.predecessors gen-inEdges'* *g n*  
**by** (*simp add: graph-path-base.predecessors-def graph-path-base.inEdges-def*)  
**lemma** *phiDefs-ign[simp]*: *CFG-SSA-base.phiDefs* (*ign Mapping.lookup (gen-phis g)*) *ga = CFG-SSA-base.phiDefs* ( $\lambda g. Mapping.lookup (gen-phis g)$ ) *g*  
**by** (*simp add: CFG-SSA-base.phiDefs-def [abs-def]*)  
**lemma** *defAss-ign[simp]*: *CFG-SSA-base.defAss* (*ign gen- $\alpha n$  g*) *gen-wf-invar* (*ign gen-inEdges'* *g*) (*ign gen-Entry g*) (*ign gen-ssa-defs g*) (*ign Mapping.lookup (gen-phis g)*) *ga*  
= *CFG-SSA-base.defAss gen- $\alpha n$  gen-wf-invar gen-inEdges' gen-Entry gen-ssa-defs* ( $\lambda g. Mapping.lookup (gen-phis g)$ ) *g*  
**by** (*simp add: CFG-SSA-base.defAss-def [abs-def]*)  
**lemma** *allUses-ign[simp]*: *CFG-SSA-base.allUses* (*ign gen- $\alpha n$  g*) (*ign gen-inEdges'* *g*) (*usesOf  $\circ$  ign gen-ssa-uses g*) (*ign Mapping.lookup (gen-phis g)*) *ga m*  
= *CFG-SSA-base.allUses gen- $\alpha n$  gen-inEdges' (usesOf  $\circ$  gen-ssa-uses)* ( $\lambda g. Mapping.lookup (gen-phis g)$ ) *g m*  
**by** (*simp add: CFG-SSA-base.CFG-SSA-defs*)  
**lemma** *defAss'-ign[simp]*: *CFG-base.defAss'* (*ign gen- $\alpha n$  g*) *gen-wf-invar* (*ign gen-inEdges'* *g*) (*ign gen-Entry g*) (*ign gen-defs g*) *ga*  
= *CFG-base.defAss' gen- $\alpha n$  gen-wf-invar gen-inEdges' gen-Entry gen-defs g*  
**by** (*simp add: CFG-base.defAss'-def [abs-def]*)

**global-interpretation** *gen-ssa-wf-notriv*: *CFG-SSA-Transformed-notriv-linorder-code gen-ssa-wf- $\alpha e$  gen-ssa-wf- $\alpha n$  gen-wf-invar gen-ssa-wf-inEdges' gen-ssa-wf-Entry gen-ssa-wf-defs gen-ssa-wf-uses gen-ssa-wf-ssa-defs gen-ssa-wf-ssa-uses gen-ssa-wf-phis gen-ssa-wf-var uninstant-chooseNext*  
**defines**  
*gen-ssa-wf-notriv-substAll = gen-ssa-wf-notriv.substAll* **and**  
*gen-ssa-wf-notriv-substAll-efficient = gen-ssa-wf-notriv.substAll-efficient*  
**apply** *unfold-locales*

**apply** *simp*  
**apply** (*transfer, clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def CFG-wf-def CFG-def graph-Entry-def graph-path-def graph-def*)  
**apply** (*transfer, clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def CFG-wf-def CFG-def graph-Entry-def graph-path-def graph-def valid-graph-def*)  
**apply** (*transfer, clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def CFG-wf-def CFG-def graph-Entry-def graph-path-def graph-def valid-graph-def*)  
**apply** (*transfer, clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def CFG-wf-def CFG-def graph-Entry-def graph-path-def graph-nodes-it-def graph-nodes-it-axioms-def*)

**apply** (*transfer, clarsimp simp: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-wf-def CFG-def graph-Entry-def graph-path-def graph-pred-it-def  
graph-pred-it-axioms-def*)

**apply** (*transfer, simp only: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-def CFG-wf-def CFG-def  
graph-Entry-def graph-Entry-axioms-def*)

**apply** (*transfer, simp only: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-def CFG-wf-def CFG-def  
graph-Entry-def graph-Entry-axioms-def graph-path-base.inEdges-def*)

**apply** (*transfer, simp only: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-def CFG-wf-def CFG-def  
graph-Entry-def graph-Entry-axioms-def graph-path-base.path2-def  
graph-path-base.path-def  
graph-path-base.predecessors-def graph-path-base.inEdges-def*)

**apply** (*transfer, clarsimp simp: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-def CFG-wf-def CFG-def  
CFG-axioms-def*)

**apply** (*transfer, simp only: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-def CFG-wf-def CFG-def  
CFG-axioms-def*)

**apply** (*transfer, clarsimp simp: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-def CFG-wf-def CFG-def  
CFG-axioms-def*)

**apply** (*transfer, clarsimp simp: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-def CFG-wf-def CFG-def  
CFG-axioms-def*)

**apply simp**

**subgoal by transfer** (*simp add: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-wf-def CFG-def CFG-axioms-def  
CFG-SSA-def CFG-SSA-axioms-def*)

**apply** (*transfer; force simp: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-def CFG-SSA-axioms-def*)

**apply** (*transfer; simp add: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-def CFG-SSA-axioms-def  
graph-path-base.predecessors-def graph-path-base.inEdges-def*)

**apply** (*transfer; clarsimp simp: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-def CFG-SSA-axioms-def*)

**apply** (*transfer; clarsimp simp: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-def CFG-SSA-axioms-def*)

**apply** (*transfer; clarsimp simp: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-wf-axioms-def CFG-SSA-base.defAss-def*)

**apply** (*transfer; clarsimp simp: CFG-SSA-Transformed-code-def  
CFG-SSA-Transformed-def CFG-SSA-wf-def CFG-SSA-wf-axioms-def*)

**apply** (*transfer; clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def  
CFG-SSA-wf-def CFG-wf-def CFG-def CFG-axioms-def*)

**apply** (*transfer; clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def  
CFG-SSA-wf-def CFG-wf-def CFG-def CFG-axioms-def*)

**apply** (*transfer; clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def  
CFG-SSA-wf-def CFG-wf-def CFG-def CFG-axioms-def*)

```

apply (transfer; clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def
CFG-SSA-wf-def CFG-wf-def CFG-def CFG-axioms-def)
apply (transfer; clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def
CFG-SSA-wf-def CFG-wf-def CFG-wf-axioms-def)
apply (transfer; clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def
CFG-SSA-Transformed-axioms-def)
apply (transfer; clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def
CFG-SSA-Transformed-axioms-def)
apply (transfer; clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def
CFG-SSA-Transformed-axioms-def)
apply (transfer; clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-def
CFG-SSA-Transformed-axioms-def)
proof -
  fix u p g
  assume CFG-SSA-Transformed gen-ssa-wf- $\alpha$ e gen-ssa-wf- $\alpha$ n gen-wf-invar gen-ssa-wf-inEdges'
gen-ssa-wf-Entry gen-ssa-wf-defs gen-ssa-wf-uses gen-ssa-wf-ssa-defs (u::('a, 'b, 'c,
'd) gen-ssa-cfg-wf  $\Rightarrow$  'a  $\Rightarrow$  'd set) p gen-ssa-wf-var
  then interpret i: CFG-SSA-Transformed gen-ssa-wf- $\alpha$ e gen-ssa-wf- $\alpha$ n gen-wf-invar
gen-ssa-wf-inEdges' gen-ssa-wf-Entry gen-ssa-wf-defs gen-ssa-wf-uses gen-ssa-wf-ssa-defs
u p gen-ssa-wf-var .
  obtain u' where [simp]: usesOf  $\circ$  u' = u
  apply (erule-tac x= $\lambda$ g. Mapping.Mapping ( $\lambda$ n. if u g n = {} then None else
Some (u g n)) in meta-allE)
  by (erule meta-impE) (auto 4 4 simp: o-def usesOf-def [abs-def] split: op-
tion.splits if-splits)
  interpret code: CFG-SSA-wf-code gen-ssa-wf- $\alpha$ e gen-ssa-wf- $\alpha$ n gen-wf-invar gen-ssa-wf-inEdges'
gen-ssa-wf-Entry gen-ssa-wf-ssa-defs u'  $\lambda$ g. Mapping.Mapping (p g)
  unfolding CFG-SSA-wf-code-def CFG-SSA-code-def
  apply simp-all
  apply (rule conjI)
  by unfold-locales

  have aux: uninstant-trivial-phis (const (Mapping.Mapping (p g))) g = uninstant-trivial-phis
( $\lambda$ g. (Mapping.Mapping (p g))) g
  by (simp add: uninstant-trivial-phis-def[abs-def])

  assume red: i.redundant g
  let ?cN = uninstant-chooseNext (u g) (Mapping.Mapping (p g)) g

  show ?cN  $\in$  dom (p g)  $\wedge$  i.trivial g (snd ?cN)
  unfolding uninstant-chooseNext-def aux
  unfolding uninstant-trivial-phis-def code.trivial-phis
  apply (rule CollectD[where a=Max -])
  apply (rule subsetD[OF - Max-in])
  apply auto[1]
  apply (rule finite-subset[OF - i.phis-finite])
  using red

```



```

  apply (auto simp: i.redundant-def[abs-def])
  apply (frule code.trivial-phi[simplified])
  apply (auto simp: i.phi-def)
done
next
  fix g
  show Mapping.keys (gen-ssa-wf-ssa-uses (g::('a, 'b, 'c, 'd) gen-ssa-cfg-wf))  $\subseteq$ 
  set (gen-ssa-wf- $\alpha$ n g)
  by transfer (clarsimp simp: CFG-SSA-Transformed-code-def CFG-SSA-Transformed-code-axioms-def)
qed (auto simp: uninstant-chooseNext-def uninstant-trivial-phis-def CFG-SSA-wf-base-code.trivial-phis-def)

```

**global-interpretation** *uninstant-code: CFG-SSA-Transformed-notriv-base-code gen-ssa-wf- $\alpha$ e gen-ssa-wf- $\alpha$ n gen-wf-invar gen-ssa-wf-inEdges' gen-ssa-wf-Entry gen-ssa-wf-defs gen-ssa-wf-uses gen-ssa-wf-ssa-defs u p gen-ssa-wf-var uninstant-chooseNext*  
**for** *u* **and** *p*  
**defines**

```

  uninstant-code-step-code = uninstant-code.step-codem and
  uninstant-code-phis' = uninstant-code.phis'-codem and
  uninstant-code-uses' = uninstant-code.uses'-codem and
  uninstant-code-substNext = uninstant-code.substNext-code and
  uninstant-code-substitution = uninstant-code.substitution-code and
  uninstant-code-triv-phis' = uninstant-code.triv-phis' and
  uninstant-code-nodes-of-uses' = uninstant-code.nodes-of-uses' and
  uninstant-code-nodes-of-phis' = uninstant-code.nodes-of-phis'
.

```

**lift-definition** *gen-cfg-wf-extend :: ('a::linorder, 'b::linorder, 'c) gen-cfg-wf  $\Rightarrow$  'd  $\Rightarrow$  ('a, 'b, 'c, 'd) gen-cfg-scheme*  
**is** *gen-cfg.extend .*

**lemma** *gen- $\alpha$ e-wf-extend [simp]:*  
*gen- $\alpha$ e (gen-cfg-wf-extend gen-cfg-wf ( $\lfloor$ gen-ssa-defs = d, gen-ssa-uses = u, gen-phis = p, gen-var = v $\rfloor$ ))*  
*= gen-wf- $\alpha$ e gen-cfg-wf*  
**by** (*simp add: gen-cfg-wf-extend-def gen-cfg.defs gen-wf- $\alpha$ e-def*)

**lemma** *gen- $\alpha$ n-wf-extend [simp]:*  
*gen- $\alpha$ n (gen-cfg-wf-extend gen-cfg-wf ( $\lfloor$ gen-ssa-defs = d, gen-ssa-uses = u, gen-phis = p, gen-var = v $\rfloor$ ))*  
*= gen-wf- $\alpha$ n gen-cfg-wf*  
**by** (*simp add: gen-cfg-wf-extend-def gen-cfg.defs gen-wf- $\alpha$ n-def*)

**lemma** *gen-inEdges-wf-extend [simp]:*  
*gen-inEdges (gen-cfg-wf-extend gen-cfg-wf ( $\lfloor$ gen-ssa-defs = d, gen-ssa-uses = u, gen-phis = p, gen-var = v $\rfloor$ ))*  
*= gen-wf-inEdges gen-cfg-wf*  
**by** (*simp add: gen-cfg-wf-extend-def gen-cfg.defs gen-wf-inEdges-def*)

**lemma** *gen-Entry-wf-extend [simp]:*

$gen-Entry$  ( $gen-cfg-wf-extend$   $gen-cfg-wf$  ( $\setminus gen-ssa-defs = d$ ,  $gen-ssa-uses = u$ ,  
 $gen-phis = p$ ,  $gen-var = v$ ))  
 $= gen-wf-Entry$   $gen-cfg-wf$   
**by** ( $simp$   $add: gen-cfg-wf-extend-def$   $gen-cfg.defs$   $gen-wf-Entry-def$ )

**lemma**  $gen-defs-wf-extend$  [ $simp$ ]:  
 $gen-defs$  ( $gen-cfg-wf-extend$   $gen-cfg-wf$  ( $\setminus gen-ssa-defs = d$ ,  $gen-ssa-uses = u$ ,  
 $gen-phis = p$ ,  $gen-var = v$ ))  
 $= gen-wf-defs$   $gen-cfg-wf$   
**by** ( $simp$   $add: gen-cfg-wf-extend-def$   $gen-cfg.defs$   $gen-wf-defs-def$ )

**lemma**  $gen-uses-wf-extend$  [ $simp$ ]:  
 $gen-uses$  ( $gen-cfg-wf-extend$   $gen-cfg-wf$  ( $\setminus gen-ssa-defs = d$ ,  $gen-ssa-uses = u$ ,  
 $gen-phis = p$ ,  $gen-var = v$ ))  
 $= gen-wf-uses$   $gen-cfg-wf$   
**by** ( $simp$   $add: gen-cfg-wf-extend-def$   $gen-cfg.defs$   $gen-wf-uses-def$ )

**lemma**  $gen-ssa-defs-wf-extend$  [ $simp$ ]:  
 $gen-ssa-defs$  ( $gen-cfg-wf-extend$   $gen-cfg-wf$  ( $\setminus gen-ssa-defs = d$ ,  $gen-ssa-uses = u$ ,  
 $gen-phis = p$ ,  $gen-var = v$ ))  
 $= d$   
**by** ( $simp$   $add: gen-cfg-wf-extend-def$   $gen-cfg.defs$ )

**lemma**  $gen-ssa-uses-wf-extend$  [ $simp$ ]:  
 $gen-ssa-uses$  ( $gen-cfg-wf-extend$   $gen-cfg-wf$  ( $\setminus gen-ssa-defs = d$ ,  $gen-ssa-uses = u$ ,  
 $gen-phis = p$ ,  $gen-var = v$ ))  
 $= u$   
**by** ( $simp$   $add: gen-cfg-wf-extend-def$   $gen-cfg.defs$ )

**lemma**  $gen-phis-wf-extend$  [ $simp$ ]:  
 $gen-phis$  ( $gen-cfg-wf-extend$   $gen-cfg-wf$  ( $\setminus gen-ssa-defs = d$ ,  $gen-ssa-uses = u$ ,  
 $gen-phis = p$ ,  $gen-var = v$ ))  
 $= p$   
**by** ( $simp$   $add: gen-cfg-wf-extend-def$   $gen-cfg.defs$ )

**lemma**  $gen-var-wf-extend$  [ $simp$ ]:  
 $gen-var$  ( $gen-cfg-wf-extend$   $gen-cfg-wf$  ( $\setminus gen-ssa-defs = d$ ,  $gen-ssa-uses = u$ ,  $gen-phis$   
 $= p$ ,  $gen-var = v$ ))  
 $= v$   
**by** ( $simp$   $add: gen-cfg-wf-extend-def$   $gen-cfg.defs$ )

**lemma**  $CFG-SSA-Transformed-codeI$ :  
**assumes**  $CFG-SSA-Transformed$   $\alpha e$   $\alpha n$   $invar$   $inEdges$   $Entry$   $oldDefs$   $oldUses$   
 $defs$  ( $\lambda g. lookup-multimap$  ( $uses$   $g$ )) ( $\lambda g. Mapping.lookup$  ( $phis$   $g$ ))  $var$   
**and**  $\bigwedge g. Mapping.keys$  ( $uses$   $g$ )  $\subseteq set$  ( $\alpha n$   $g$ )  
**shows**  $CFG-SSA-Transformed-code$   $\alpha e$   $\alpha n$   $invar$   $inEdges$   $Entry$   $oldDefs$   $oldUses$   
 $defs$   $uses$   $phis$   $var$   
**proof** –  
**interpret**  $CFG-SSA-Transformed$   $\alpha e$   $\alpha n$   $invar$   $inEdges$   $Entry$   $oldDefs$   $oldUses$

```

defns λg. lookup-multimap (uses g) λg. Mapping.lookup (phis g) var
  by fact
  have [simp]: usesOf = lookup-multimap
    by (intro ext) (clarsimp simp: lookup-multimap-def)
  from assms
  show ?thesis
  apply unfold-locales
    apply (auto intro!: defs-uses-disjoint)[1]
    apply (rule defs-finite)
    apply (rule uses-in-αn)
    apply simp
    apply (clarsimp split: option.splits)
    apply (rule invar)
    apply (rule phis-finite)
    apply (rule phis-in-αn; simp)
    apply (rule phis-wf; simp)
    apply (rule simpleDefs-phiDefs-disjoint; simp)
    apply (rule allDefs-disjoint; simp)
    apply (rule allUses-def-ass; simp add: comp-def)
    apply (rule Entry-no-phis)
    apply (rule oldDefs-def)
    apply (auto intro!: oldUses-def)[1]
    apply (rule conventional; simp add: comp-def)
    apply (rule phis-same-var; simp)
    apply (rule allDefs-var-disjoint; simp)
  by auto
qed

lemma CFG-SSA-Transformed-ign:
  CFG-SSA-Transformed (ign gen-wf-αe gen-cfg-wf) (ign gen-wf-αn gen-cfg-wf)
  gen-wf-invar
    (const (gen-wf-inEdges' gen-cfg-wf)) (ign gen-wf-Entry gen-cfg-wf) (ign
  gen-wf-defs gen-cfg-wf)
    (ign gen-wf-uses gen-cfg-wf) (ign gen-wf-defs' gen-cfg-wf) (ign gen-wf.uses'
  gen-cfg-wf)
    (ign gen-wf.phis' gen-cfg-wf)
    (ign gen-wf-var gen-cfg-wf)
  unfolding CFG-SSA-Transformed-def CFG-wf-def CFG-def CFG-wf-axioms-def
  graph-Entry-def graph-path-def graph-Entry-axioms-def
  CFG-axioms-def CFG-SSA-wf-def CFG-SSA-def CFG-SSA-axioms-def
  CFG-SSA-wf-axioms-def CFG-SSA-Transformed-axioms-def
  graph-def graph-nodes-it-def graph-pred-it-def
  graph-nodes-it-axioms-def graph-pred-it-axioms-def
  apply (clarsimp simp: gen-wf.Entry-unreachable gen-wf.defs-uses-disjoint [where
  g=gen-cfg-wf]
  gen-wf.uses-in-αn
  gen-wf.braun-ssa.uses-in-αn gen-wf.phis'-finite
  gen-wf.αn-distinct
  gen-wf.valid gen-wf.finite [simplified]

```

```

    gen-wf.ni.nodes-list-it-correct [simplified]
    gen-wf.pi.pred-list-it-correct [simplified])
apply (intro conjI)
    using gen-wf.Entry-unreachable [of gen-cfg-wf]
    apply (auto simp: graph-path-base.inEdges-def)[1]
    using gen-wf.Entry-reaches
    apply (fastforce cong del: imp-cong simp: graph-path-base.path2-def
graph-path-base.path-def graph-path-base.predecessors-def graph-path-base.inEdges-def)[1]
    using gen-wf.def-ass-uses [of gen-cfg-wf]
    apply (auto simp: CFG-base.defAss'-def graph-path-base.path2-def
graph-path-base.path-def graph-path-base.predecessors-def graph-path-base.inEdges-def)[1]
    using gen-wf.Entry-unreachable [of gen-cfg-wf]
    apply (auto simp: graph-path-base.inEdges-def)[1]
    using gen-wf.Entry-reaches
    apply (fastforce cong del: imp-cong simp: graph-path-base.path2-def
graph-path-base.path-def graph-path-base.predecessors-def gen-wf.defs-uses-disjoint
graph-path-base.inEdges-def)[1]
    apply (auto dest: gen-wf.defs'-uses'-disjoint [where g=gen-cfg-wf])[1]
    apply (auto dest: gen-wf.braun-ssa.phis-in- $\alpha$ n)[1]
    apply (auto dest: gen-wf.phis'-wf simp: graph-path-base.predecessors-def
gen-wf-predecessors-def graph-path-base.inEdges-def)[1]
    apply (fastforce dest: gen-wf.braun-ssa.simpleDefs-phiDefs-disjoint simp:
CFG-SSA-base.phiDefs-def dom-def)[1]
    using gen-wf.braun-ssa.allDefs-disjoint[where g=gen-cfg-wf]
    apply (clarsimp simp: CFG-SSA-base.CFG-SSA-defs)
    apply clarsimp
    apply (drule gen-wf.braun-ssa.allUses-def-ass [where g=gen-cfg-wf, rotated
1])
    apply (auto simp: CFG-SSA-wf-base.CFG-SSA-wf-defs CFG-SSA-wf-base.defAssUses-def
graph-path-base.path2-def graph-path-base.path-def graph-path-base.predecessors-def
graph-path-base.successors-def graph-path-base.inEdges-def)[2]
    apply (clarsimp simp: gen-wf.oldDefs-correct)
    apply (clarsimp simp: gen-wf.oldUses-correct)
    apply (intro allI impI gen-wf.conventional; auto simp: graph-path-base.path2-def
graph-path-base.path-def graph-path-base.predecessors-def graph-path-base.successors-def
CFG-SSA-base.CFG-SSA-defs graph-path-base.inEdges-def)
    apply (intro allI impI gen-wf.phis'-fst; assumption)
by (intro allI impI gen-wf.allDefs-var-disjoint; auto simp: CFG-SSA-base.CFG-SSA-defs)

lift-definition gen-ssa-cfg-wf :: ('node::linorder, 'var::linorder, 'edge) gen-cfg-wf
 $\Rightarrow$  ('node, 'var, 'edge, ('node,'var) ssaVal) gen-ssa-cfg-wf
is  $\lambda$ g. let (uses,phis) = gen-wf-uses'-phis' g in (gen-cfg-wf-extend g)(
    gen-ssa-defs = gen-wf-defs' g,
    gen-ssa-uses = uses,
    gen-phis = phis,
    gen-var = gen-wf-var g
)
apply (simp add: Let-def gen-wf-uses'-phis'-def split-beta)
apply (subst CFG-Construct-linorder.snd-uses'-phis'[symmetric, of gen-wf- $\alpha$ e -

```

```

gen-wf-invar - gen-wf-Entry))
  apply unfold-locales[1]
apply (rule CFG-SSA-Transformed-codeI)
  apply (subst CFG-Construct-linorder.fst-uses'-phis'[symmetric, of gen-wf- $\alpha$ e -
gen-wf-invar - gen-wf-Entry))
  apply unfold-locales[1]
  apply transfer
  apply (rule CFG-SSA-Transformed-ign)
apply (rule CFG-Construct-linorder.fst-uses'-phis'-in- $\alpha$ n)
by unfold-locales

declare uninstantiated.defNode-code[abs-def, code] uninstantiated.allVars-code[abs-def, code] uninstantiated.allUses-def[abs-def,
code] uninstantiated.allDefs-def[abs-def, code]
  uninstantiated.phiUses-code[abs-def, code] uninstantiated.phi-def[abs-def, code] uninstantiated.redundant-code-def[abs-def,
code]
declare uninstantiated.code.uses'-code-def[abs-def, code] uninstantiated.code.substNext-code-def[abs-def,
code] uninstantiated.code.substitution-code-def[abs-def, folded uninstantiated.phi-def, code]
declare uninstantiated.code.phis'-code-def[folded uninstantiated.code.substNext-def, code] uninstantiated.code.step-code-def[folded
uninstantiated.code.uses'-code-def uninstantiated.code.phis'-code-def, code]
  uninstantiated.code.cond-code-def[folded uninstantiated.redundant-def, code]
declare gen-ssa-wf-notriv.substAll-efficient-def
  [folded uninstantiated.code.nodes-of-phis'-def uninstantiated.code.nodes-of-uses'-def uninstantiated.code-triv-phis'-def
  uninstantiated.code.substitution-def
  uninstantiated.code.step-code-def uninstantiated.code.phis'-def uninstantiated.code.uses'-def uninstantiated.trivial-phis-def
  uninstantiated.phidefNodes-def uninstantiated.useNodes-of-def uninstantiated.phiNodes-of-def, code]
declare keys-dom-lookup [symmetric, code-unfold]

definition map-keys-from-sparse  $\equiv$  map-keys gen-wf.from-sparse

declare map-keys-code[OF gen-wf.from-sparse-inj, folded map-keys-from-sparse-def,
code]
declare map-keys-from-sparse-def[symmetric, code-unfold]

lemma fold-Cons-commute: ( $\bigwedge a b. \llbracket a \in \text{set } (x \# xs); b \in \text{set } (x \# xs) \rrbracket \implies f a$ 
 $\circ f b = f b \circ f a$ )
 $\implies \text{fold } f (x \# xs) = f x \circ (\text{fold } f xs)$ 
by (simp add: fold-commute)

lemma Union-of-code [code]: Union-of  $f$  (RBT-Set.Set  $r$ ) = RBT.fold ( $\lambda a -. (\cup$ 
 $(f a)) r \}$ )
proof -
{ fix  $xs$ 
  have ( $\bigcup x \in \{x. (x, ()) \in \text{set } xs\}. f x$ ) = fold ( $\lambda(a,-). (\cup) (f a)$ )  $xs \}$ 
  apply (induction  $xs$ )
  apply simp
  by (subst fold-Cons-commute) auto
}
note Union-fold = this
show ?thesis

```

**unfolding** *Union-of-def*  
**by** (*clarsimp simp: RBT-Set.Set-def RBT.fold-fold RBT.lookup-in-tree*) (*rule Union-fold [simplified]*)  
**qed**

**definition**[*code*]: *disjoint xs ys = (xs ∩ ys = {})*

**definition** *gen-ssa-wf-notriv-substAll' = fst ∘ gen-ssa-wf-notriv-substAll-efficient*

**definition** *fold-set f A ≡ fold f (sorted-list-of-set A)*

**declare** *fold-set-def [symmetric, code-unfold]*

**declare** *fold-set-def*

[**where** *A=RBT-Set.Set r for r,*

*unfolded sorted-list-set fold-keys-def-alt [symmetric,abs-def] fold-keys-def [abs-def],*  
*code]*

**declare** *graph-path-base.inEdges-def [code]*

**end**

**theory** *Generic-Extract imports*

*Generic-Interpretation*

**begin**

**export-code open**

*set sorted-list-of-set disjoint RBT.fold*

*gen-ssa-cfg-wf gen-wf-var gen-ssa-wf-notriv-substAll'*

**in** *OCaml module-name BraunSSA*

**end**

**theory** *Disjoin-Transform imports*

*Slicing.AdditionalLemmas*

**begin**

**inductive** *subcmd :: cmd ⇒ cmd ⇒ bool where*

*sub-Skip: subcmd c Skip*

| *sub-Base: subcmd c c*

| *sub-Seq1: subcmd c1 c ⇒ subcmd (c1;;c2) c*

| *sub-Seq2: subcmd c2 c ⇒ subcmd (c1;;c2) c*

| *sub-If1: subcmd c1 c ⇒ subcmd (if (b) c1 else c2) c*

| *sub-If2: subcmd c2 c ⇒ subcmd (if (b) c1 else c2) c*

| *sub-While: subcmd c' c ⇒ subcmd (while (b) c') c*

**fun** *maxVnameLen-aux :: expr ⇒ nat where*

*maxVnameLen-aux (Val -) = 0*

| *maxVnameLen-aux (Var V) = length V*

|  $\text{maxVnameLen-aux } (e1 \ll - \gg e2) = \text{max } (\text{maxVnameLen-aux } e1) (\text{maxVnameLen-aux } e2)$

**fun**  $\text{maxVnameLen} :: \text{cmd} \Rightarrow \text{nat}$  **where**  
 $\text{maxVnameLen Skip} = 0$   
|  $\text{maxVnameLen } (V := e) = \text{max } (\text{length } V) (\text{maxVnameLen-aux } e)$   
|  $\text{maxVnameLen } (c1 ;; c2) = \text{max } (\text{maxVnameLen } c1) (\text{maxVnameLen } c2)$   
|  $\text{maxVnameLen } (\text{if } (b) c1 \text{ else } c2) = \text{max } (\text{maxVnameLen } c1) (\text{max } (\text{maxVnameLen-aux } b) (\text{maxVnameLen } c2))$   
|  $\text{maxVnameLen } (\text{while } (b) c) = \text{max } (\text{maxVnameLen } c) (\text{maxVnameLen-aux } b)$

**definition**  $\text{tempName} :: \text{cmd} \Rightarrow \text{vname}$  **where**  $\text{tempName } c \equiv \text{replicate } (\text{Suc } (\text{maxVnameLen } c)) (\text{CHR } "a")$

**inductive**  $\text{newname} :: \text{cmd} \Rightarrow \text{vname} \Rightarrow \text{bool}$  **where**  
 $\text{newname Skip } V$   
|  $V \notin \{V'\} \cup \text{rhs-aux } e \Longrightarrow \text{newname } (V' := e) V$   
|  $\llbracket \text{newname } c1 V ; \text{newname } c2 V \rrbracket \Longrightarrow \text{newname } (c1 ;; c2) V$   
|  $\llbracket \text{newname } c1 V ; \text{newname } c2 V ; V \notin \text{rhs-aux } b \rrbracket \Longrightarrow \text{newname } (\text{if } (b) c1 \text{ else } c2) V$   
|  $\llbracket \text{newname } c V ; V \notin \text{rhs-aux } b \rrbracket \Longrightarrow \text{newname } (\text{while } (b) c) V$

**lemma**  $\text{maxVnameLen-aux-newname}$ :  $\text{length } V > \text{maxVnameLen-aux } e \Longrightarrow V \notin \text{rhs-aux } e$   
**by** ( $\text{induction } e$ )  $\text{auto}$

**lemma**  $\text{maxVnameLen-newname}$ :  $\text{length } V > \text{maxVnameLen } c \Longrightarrow \text{newname } c V$   
**by** ( $\text{induction } c$ ) ( $\text{auto intro:newname.intros dest:maxVnameLen-aux-newname}$ )

**lemma**  $\text{tempname-newname[intro]}$ :  $\text{newname } c (\text{tempName } c)$   
**by** ( $\text{rule maxVnameLen-newname}$ ) ( $\text{simp add: tempName-def}$ )

**fun**  $\text{transform-aux} :: \text{vname} \Rightarrow \text{cmd} \Rightarrow \text{cmd}$  **where**  
 $\text{transform-aux - Skip} = \text{Skip}$   
|  $\text{transform-aux } V' (V := e) =$   
 $\quad (\text{if } V \in \text{rhs } (V := e) \text{ then } V' := e ;; V := \text{Var } V'$   
 $\quad \text{else } V := e)$   
|  $\text{transform-aux } V' (c1 ;; c2) = \text{transform-aux } V' c1 ;; \text{transform-aux } V' c2$   
|  $\text{transform-aux } V' (\text{if } (b) c1 \text{ else } c2) =$   
 $\quad (\text{if } (b) \text{transform-aux } V' c1 \text{ else } \text{transform-aux } V' c2)$   
|  $\text{transform-aux } V' (\text{while } (b) c) = (\text{while } (b) \text{transform-aux } V' c)$

**abbreviation**  $\text{transform} :: \text{cmd} \Rightarrow \text{cmd}$  **where**  
 $\text{transform } c \equiv \text{transform-aux } (\text{tempName } c) c$

**fun**  $\text{leftmostCmd} :: \text{cmd} \Rightarrow \text{cmd}$  **where**  
 $\text{leftmostCmd } (c1 ;; c2) = \text{leftmostCmd } c1$   
|  $\text{leftmostCmd } c = c$

**lemma** *leftmost-lhs*[simp]:  $lhs (leftmostCmd\ c) = lhs\ c$   
**by** (*induction c*) *auto*

**lemma** *leftmost-rhs*[simp]:  $rhs (leftmostCmd\ c) = rhs\ c$   
**by** (*induction c*) *auto*

**lemma** *leftmost-subcmd*[intro]:  $subcmd\ c (leftmostCmd\ c)$   
**by** (*induction c*) (*auto intro:subcmd.intros*)

**lemma** *leftmost-labels*:  $labels\ c\ n\ c' \implies subcmd\ c (leftmostCmd\ c')$   
**by** (*induction rule:labels.induct*) (*auto intro:subcmd.intros*)

**theorem** *transform-disjoint*:  
**assumes**  $subcmd (transform\ aux\ temp\ c) (V:=e)\ newname\ c\ temp$   
**shows**  $V \notin rhs\ aux\ e$   
**using** *assms* **proof** (*induction c rule:transform-aux.induct*)  
**case** (*3 V c1 c2*)  
**from** *3.prem*s(1) **show** ?*case*  
**apply** *simp*  
**proof** (*cases (transform\ aux\ temp\ c1;; transform\ aux\ temp\ c2) (V:=e) rule:subcmd.cases*)  
**case** *sub-Seq2*  
**with** *3.prem*s(2) **show** ?*thesis* **by**  $-(rule\ 3.IH(1), auto\ elim:newname.cases)$   
**next**  
**case** *sub-If1*  
**with** *3.prem*s(2) **show** ?*thesis* **by**  $-(rule\ 3.IH(2), auto\ elim:newname.cases)$   
**qed** *auto*  
**next**  
**case** (*4 V b c1 c2*)  
**from** *4.prem*s(1) **show** ?*case*  
**apply** *simp*  
**proof** (*cases (if (b) transform\ aux\ temp\ c1 else transform\ aux\ temp\ c2) (V:=e) rule:subcmd.cases*)  
**case** *sub-If2*  
**with** *4.prem*s(2) **show** ?*thesis* **by**  $-(rule\ 4.IH(1), auto\ elim:newname.cases)$   
**next**  
**case** *sub-While*  
**with** *4.prem*s(2) **show** ?*thesis* **by**  $-(rule\ 4.IH(2), auto\ elim:newname.cases)$   
**qed** *auto*  
**next**  
**case** *5*  
**from** *5.prem*s **show** ?*case* **by**  $-(rule\ 5.IH, auto\ elim:subcmd.cases\ newname.cases)$   
**qed** (*auto elim!:subcmd.cases\ newname.cases split:if-split-asm*)

**lemma** *transform-disjoint'*:  $subcmd (transform\ c) (leftmostCmd\ c') \implies lhs\ c' \cap rhs\ c' = \{\}$   
**by** (*induction c'*) (*auto dest:transform-disjoint*)

**corollary** *Defs-Uses-transform-disjoint* [simp]:  $Defs (transform\ c)\ n \cap Uses (transform\ c)\ n = \{\}$



```

    by (auto dest: leftmost-labels transform-disjoint' labels-det)

end

6.5.1 Instantiation for a Simple While Language

theory WhileGraphSSA imports
  Generic-Interpretation
  Disjoin-Transform
  HOL-Library.List-Lexorder
  HOL-Library.Char-ord
begin

instantiation w-node :: ord
begin

fun less-eq-w-node where
  (-Entry-) ≤ x = True
| (- n -) ≤ x = (case x of
  (-Entry-) ⇒ False
  | (- m -) ⇒ n ≤ m
  | (-Exit-) ⇒ True)
| (-Exit-) ≤ x = (x = (-Exit-))

fun less-w-node where
  (-Entry-) < x = (x ≠ (-Entry-))
| (- n -) < x = (case x of
  (-Entry-) ⇒ False
  | (- m -) ⇒ n < m
  | (-Exit-) ⇒ True)
| (-Exit-) < x = False

instance ..
end

instance w-node :: linorder proof
  fix x y z :: w-node

  show x ≤ x by (cases x) auto
  show x ≤ y ∨ y ≤ x by (cases x) (cases y, auto)+
  show x < y ⟷ x ≤ y ∧ ¬ y ≤ x by (cases x) (cases y, auto)+

  assume x ≤ y and y ≤ z
  thus x ≤ z by (cases x, cases y, cases z) auto

  assume x ≤ y and y ≤ x
  thus x = y by (cases x) (cases y, auto)+
qed

```

```

declare Defs.simps [simp del]
declare Uses.simps [simp del]
declare Let-def [simp]

declare finite-valid-nodes [simp, intro!]

lemma finite-valid-edge [simp, intro!]: finite (Collect (valid-edge c))
  unfolding valid-edge-def [abs-def]
apply (rule inj-on-finite [where  $f = \lambda(f,d,t). (f,t)$  and  $B = \text{Collect } (\text{valid-node } c)$ 
 $\times \text{Collect } (\text{valid-node } c)$ ])
  apply (rule inj-onI)
  apply (auto intro: WCFG-edge-det)[1]
apply (force simp: valid-node-def valid-edge-def)[1]
by auto

lemma uses-expr-finite: finite (rhs-aux e)
  by (induction e) auto

lemma uses-cmd-finite: finite (rhs c)
  by (induction c) (auto intro: uses-expr-finite)

lemma defs-cmd-finite: finite (lhs c)
  by (induction c) auto

lemma finite-labels': finite  $\{(l,c). \text{labels prog } l\ c\}$ 
proof –
  have  $\{l. \exists c. \text{labels prog } l\ c\} = \text{fst } ' \{(l,c). \text{labels prog } l\ c\}$ 
    by auto
  with finite-labels [of prog] labels-det [of prog] show ?thesis
    by (auto 4 4 intro: inj-onI dest: finite-imageD)
qed

lemma finite-Defs [simp, intro!]: finite (Defs c n)
  unfolding Defs.simps
apply clarsimp
apply (rule-tac  $B = \bigcup (\text{lhs } ' \text{snd } ' \{(l,c'). \text{labels } c\ l\ c'\})$  in finite-subset)
  apply fastforce
apply (rule finite-Union)
  apply (rule finite-imageI)+
  apply (rule finite-labels')
by (clarsimp simp: defs-cmd-finite)

lemma finite-Uses [simp, intro!]: finite (Uses c n)
  unfolding Uses.simps
apply clarsimp
apply (rule-tac  $B = \bigcup (\text{rhs } ' \text{snd } ' \{(l,c'). \text{labels } c\ l\ c'\})$  in finite-subset)
  apply fastforce
apply (rule finite-Union)
  apply (rule finite-imageI)+

```

**apply** (*rule finite-labels'*)  
**by** (*clarsimp simp: uses-cmd-finite*)

**definition** *while-cfg- $\alpha$ e*  $c = \text{Collect } (\text{valid-edge } (\text{transform } c))$

**definition** *while-cfg- $\alpha$ n*  $c = \text{sorted-list-of-set } (\text{Collect } (\text{valid-node } (\text{transform } c)))$

**definition** *while-cfg-invar*  $c = \text{True}$

**definition** *while-cfg-inEdges'*  $c \ t = (\text{SOME } ls. \text{distinct } ls \wedge \text{set } ls = \{( \text{sourcenode } e, \text{kind } e) \mid e. \text{valid-edge } (\text{transform } c) \ e \wedge \text{targetnode } e = t\})$

**definition** *while-cfg-Entry*  $c = (-\text{Entry-})$

**definition** *while-cfg-defs*  $c = (\text{Defs } (\text{transform } c))((-\text{Entry-}) := \{v. \exists n. v \in \text{Uses } (\text{transform } c) \ n\})$

**definition** *while-cfg-uses*  $c = \text{Uses } (\text{transform } c)$

**abbreviation** *while-cfg-inEdges*  $c \ t \equiv \text{map } (\lambda(f,d). (f,d,t)) (\text{while-cfg-inEdges}' \ c \ t)$

**lemmas** *while-cfg-defs* = *while-cfg- $\alpha$ e-def* *while-cfg- $\alpha$ n-def*  
*while-cfg-invar-def* *while-cfg-inEdges'-def*  
*while-cfg-Entry-def* *while-cfg-defs-def*  
*while-cfg-uses-def*

**interpretation** *while: graph-path* *while-cfg- $\alpha$ e* *while-cfg- $\alpha$ n* *while-cfg-invar* *while-cfg-inEdges'*

**apply** *unfold-locales*

**apply** (*simp-all add: while-cfg-defs*)

**apply** (*force simp: valid-node-def*)[1]

**apply** (*force simp: valid-node-def*)[1]

**apply** (*rule set-iterator-I*)

**prefer 3 apply** (*simp add: foldri-def*)

**apply** *simp*

**apply** *simp*

**apply** (*clarsimp simp: Graph-path.pred-def*)

**apply** (*subgoal-tac finite*  $\{(v', w). \text{valid-edge } (\text{transform } g) (v', w, v)\}$ )

**apply** (*drule finite-distinct-list*)

**apply** *clarsimp*

**apply** (*rule-tac a=xs in someI2*)

**apply** *simp*

**apply** *clarsimp*

**apply** (*metis set-iterator-foldri-correct*)

**apply** (*rule-tac f= $\lambda(f,d,t). (f,d)$  in finite-surj* [*OF finite-valid-edge*])

**by** (*auto intro: rev-image-eqI*)

**lemma** *right-total-const: right-total*  $(\lambda x \ y. x = c)$

**by** (*rule right-totalI*) *simp*

**lemma** *const-transfer: rel-fun*  $(\lambda x \ y. x = c) (=) f \ (\lambda-. f \ c)$

**by** (*clarsimp simp: rel-fun-def*)

**interpretation** *while-ign: graph-path*  $\lambda-. \text{while-cfg-}\alpha\text{e cmd}$   $\lambda-. \text{while-cfg-}\alpha\text{n cmd}$   
 $\lambda-. \text{while-cfg-invar cmd}$   $\lambda-. \text{while-cfg-inEdges}' \text{ cmd}$

**by** (rule graph-path-transfer [OF right-total-const const-transfer const-transfer const-transfer const-transfer, rule-format])

unfold-locales

**definition** *gen-while-cfg*  $g \equiv ()$   
*gen- $\alpha e$*  = *while-cfg- $\alpha e$*   $g$ ,  
*gen- $\alpha n$*  = *while-cfg- $\alpha n$*   $g$ ,  
*gen-inEdges* = *while-cfg-inEdges*  $g$ ,  
*gen-Entry* = *while-cfg-Entry*  $g$ ,  
*gen-defs* = *while-cfg-defs*  $g$  ,  
*gen-uses* = *while-cfg-uses*  $g$   
 $()$

**lemma** *while-path-graph-pathD*: *While-CFG.path* (transform  $c$ )  $n$  *es*  $m \implies$  *while.path2*  $c$   $n$  ( $n \# \text{map}$  targetnode *es*)  $m$

**unfolding** *while.path2-def*

**apply** (*induction*  $n$  *es*  $m$  rule: *While-CFG.path.induct*)

**apply** *clarsimp*

**apply** (rule *while.path.intros*)

**apply** (*auto simp*: *while-cfg-defs valid-node-def While-CFG.valid-node-def*)[1]

**apply** (*simp add*: *while-cfg-defs*)

**apply** *clarsimp*

**apply** (rule *while.path.intros*)

**apply** *assumption*

**apply** (*clarsimp simp*: *while.predecessors-def*)

**apply** (*rename-tac*  $n$  *ed*  $m$ )

**apply** (*rule-tac*  $x=(n,ed,m)$  **in** *image-eqI*)

**apply** *simp*

**apply** (*clarsimp simp*: *while.inEdges-def*)

**apply** (*rule-tac*  $x=(n,ed)$  **in** *image-eqI*)

**apply** *simp*

**apply** (*clarsimp simp*: *while-cfg-inEdges'-def*)

**apply** (*subgoal-tac* *finite*  $\{(aa, a). \text{valid-edge} (transform\ c)\ (aa, a, m)\}$ )

**prefer** 2

**apply** (*rule-tac*  $f=\lambda(f,d,t). (f,d)$  **in** *finite-surj* [OF *finite-valid-edge*])

**apply** (*auto intro*: *rev-image-eqI*)[1]

**apply** (*drule* *finite-distinct-list*)

**apply** *clarsimp*

**by** (*rule-tac*  $a=xs$  **in** *someI2*; *simp*)

**lemma** *Uses-Entry* [*simp*]: *Uses*  $c$  (-Entry-) = {}

**unfolding** *Uses.simps* **by** *auto*

**lemma** *in-Uses-valid-node*:  $V \in \text{Uses } c\ n \implies \text{valid-node } c\ n$

**by** (*auto dest!*: *label-less-num-inner-nodes less-num-nodes-edge*

*simp*: *Uses.simps valid-node-def valid-edge-def*)

**lemma** *while-cfg-CFG-wf-impl*:

*SSA-CFG.CFG-wf* ( $\lambda-. \text{gen-}\alpha e$  (*gen-while-cfg cmd*)) ( $\lambda-. \text{gen-}\alpha n$  (*gen-while-cfg*

```

cmd))
  (λ-. while-cfg-invar cmd) (λ-. gen-inEdges' (gen-while-cfg cmd))
  (λ-. gen-Entry (gen-while-cfg cmd)) (λ-. gen-defs (gen-while-cfg cmd))
  (λ-. gen-uses (gen-while-cfg cmd))
apply (simp add: gen-while-cfg-def o-def split-beta)
unfolding SSA-CFG.CFG-wf-def
apply (rule conjI)
apply (rule CFG-transfer [OF right-total-const const-transfer const-transfer const-transfer
const-transfer const-transfer const-transfer const-transfer, rule-format])
apply unfold-locales[1]
  apply (auto simp: while-cfg-defs valid-node-def valid-edge-def intro: While-CFG.intros)[1]
  apply (clarsimp simp: while.inEdges-def)
  apply (clarsimp simp: while-cfg-defs valid-edge-def)
  apply (subgoal-tac {(aa, a). transform g ⊢ aa -a → (-Entry-)} = {})
  apply clarsimp
  apply (rule-tac a=[] in someI2; simp)
  apply auto[1]
  apply (subst(asm) while-cfg-αn-def)
  apply simp
  apply (drule valid-node-Entry-path)
  apply clarsimp
  apply (drule while-path-graph-pathD)
  apply (auto simp: while-cfg-Entry-def)[1]
  apply (clarsimp simp: while-cfg-defs)
  apply (clarsimp simp: while-cfg-defs)
  apply (subgoal-tac {v. ∃ n. v ∈ Uses (transform g) n} = (⋃ n ∈ Collect
(valid-node (transform g)). Uses (transform g) n))
  apply simp
  apply (auto dest: in-Uses-valid-node)[1]
  apply (auto dest!: label-less-num-inner-nodes less-num-nodes-edge
simp: Uses.simps valid-node-def valid-edge-def while-cfg-defs)[1]
  apply (clarsimp simp: while-cfg-defs)
  apply (clarsimp simp: while-cfg-defs)
apply (clarsimp simp: SSA-CFG.CFG-wf-axioms-def CFG-base.defAss'-def)
apply (rule-tac x=(-Entry-) in bexI)
  apply (auto simp: while-cfg-defs)[1]
by (auto elim: graph-path-base.path.cases simp: graph-path-base.path2-def while-cfg-Entry-def)

lift-definition gen-while-cfg-wf :: cmd ⇒ (w-node, vname, state edge-kind) gen-cfg-wf
  is gen-while-cfg
using while-cfg-CFG-wf-impl
  by (auto simp: gen-while-cfg-def o-def split-beta while-cfg-invar-def)

definition build-ssa cmd = gen-ssa-wf-notriv-substAll (gen-ssa-cfg-wf (gen-while-cfg-wf
cmd))

end

```

## References

- [1] G. Barthe, D. Demange, and D. Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, Mar. 2014.
- [2] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. Simple and efficient construction of static single assignment form. In R. Jhala and K. Bosschere, editors, *Compiler Construction*, volume 7791 of *Lecture Notes in Computer Science*, pages 102–122. Springer Berlin Heidelberg, 2013.