

# 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 . . . . .	6
1.3	Mapping Extensions . . . . .	7
<b>2</b>	<b>SSA Representation</b>	<b>9</b>
2.1	Inductive Graph Paths . . . . .	9
2.2	Domination . . . . .	18
2.3	CFG . . . . .	22
2.4	SSA CFG . . . . .	24
2.5	Bundling of CFG and Equivalent SSA CFG . . . . .	31
<b>3</b>	<b>Minimality</b>	<b>33</b>
<b>4</b>	<b>SSA Construction</b>	<b>36</b>
4.1	CFG to SSA CFG . . . . .	36
4.2	Inductive Removal of Trivial Phi Functions . . . . .	44

<b>5</b>	<b>Proof of Semantic Equivalence</b>	<b>51</b>
<b>6</b>	<b>Code Generation</b>	<b>52</b>
6.1	While Combinator Extensions . . . . .	52
6.2	Code Equations for SSA Construction . . . . .	59
6.3	Locales Transfer Rules . . . . .	63
6.4	Code Equations for SSA Minimization . . . . .	73
6.5	Generic Code Extraction Based on typedefs . . . . .	82
6.5.1	Instantiation for a Simple While Language . . . . .	92

# 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
  <proof>
```

```
lemma not-in-butlast[simp]:  $\llbracket x \in \text{set } ys; x \notin \text{set } (\text{butlast } ys) \rrbracket \implies x = \text{last } ys$ 
  <proof>
```

```
lemma in-set-butlastI:  $x \in \text{set } xs \implies x \neq \text{last } xs \implies x \in \text{set } (\text{butlast } xs)$ 
  <proof>
```

```
lemma butlast-strict-prefix:  $xs \neq [] \implies \text{strict-prefix } (\text{butlast } xs) \ xs$ 
  <proof>
```

```
lemma set-tl:  $\text{set } (\text{tl } xs) \subseteq \text{set } xs$ 
  <proof>
```

```
lemma in-set-tlD[elim]:  $x \in \text{set } (\text{tl } xs) \implies x \in \text{set } xs$ 
  <proof>
```

```
lemma suffix-unsnoc:
  assumes suffix xs ys  $xs \neq []$ 
  obtains x where  $xs = \text{butlast } xs@[x]$   $ys = \text{butlast } ys@[x]$ 
  <proof>
```

```
lemma prefix-split-first:
  assumes  $x \in \text{set } xs$ 
  obtains as where  $\text{prefix } (\text{as@[x]}) \ xs$  and  $x \notin \text{set } as$ 
  <proof>
```

```
lemma in-prefix[elim]:
  assumes  $\text{prefix } xs \ ys$  and  $x \in \text{set } xs$ 
```

**shows**  $x \in \text{set } ys$   
*<proof>*

**lemma** *strict-prefix-butlast*:  
**assumes**  $\text{prefix } xs \ (\text{butlast } ys) \ ys \neq []$   
**shows**  $\text{strict-prefix } xs \ ys$   
*<proof>*

**lemma** *prefix-tl-subset*:  $\text{prefix } xs \ ys \implies \text{set } (\text{tl } xs) \subseteq \text{set } (\text{tl } ys)$   
*<proof>*

**lemma** *suffix-tl-subset*:  $\text{suffix } xs \ ys \implies \text{set } (\text{tl } xs) \subseteq \text{set } (\text{tl } ys)$   
*<proof>*

**lemma** *set-tl-append'*:  $\text{set } (\text{tl } (xs \ @ \ ys)) \subseteq \text{set } (\text{tl } xs) \cup \text{set } ys$   
*<proof>*

**lemma** *last-in-tl*:  $\text{length } xs > 1 \implies \text{last } xs \in \text{set } (\text{tl } xs)$   
*<proof>*

**lemma** *concat-join*:  $xs \neq [] \implies ys \neq [] \implies \text{last } xs = \text{hd } ys \implies \text{butlast } xs@ys = xs@tl \ ys$   
*<proof>*

**lemma** *fold-induct[case-names Nil Cons]*:  $P \ s \implies (\bigwedge x \ s. x \in \text{set } xs \implies P \ s \implies P \ (f \ x \ s)) \implies P \ (\text{fold } f \ xs \ s)$   
*<proof>*

**lemma** *fold-union-elem*:  
**assumes**  $x \in \text{fold } (\cup) \ xss \ xs$   
**obtains**  $ys$  **where**  $x \in ys \ ys \in \text{set } xss \cup \{xs\}$   
*<proof>*

**lemma** *fold-union-elemI*:  
**assumes**  $x \in ys \ ys \in \text{set } xss \cup \{xs\}$   
**shows**  $x \in \text{fold } (\cup) \ xss \ xs$   
*<proof>*

**lemma** *fold-union-elemI'*:  
**assumes**  $x \in xs \vee (\exists xs \in \text{set } xss. x \in xs)$   
**shows**  $x \in \text{fold } (\cup) \ xss \ xs$   
*<proof>*

**lemma** *fold-union-finite[intro!]*:  
**assumes**  $\text{finite } xs \ \forall xs \in \text{set } xss. \text{finite } xs$   
**shows**  $\text{finite } (\text{fold } (\cup) \ xss \ xs)$   
*<proof>*

**lemma** *in-set-zip-map*:

**assumes**  $(x,y) \in \text{set } (\text{zip } xs \ (\text{map } f \ ys))$   
**obtains**  $y'$  **where**  $(x,y') \in \text{set } (\text{zip } xs \ ys)$   $f \ y' = y$   
*<proof>*

**lemma** *dom-comp-subset*:  $g \text{ ' dom } (f \circ g) \subseteq \text{dom } f$   
*<proof>*

**lemma** *finite-dom-comp*:  
**assumes** *finite*  $(\text{dom } f)$  *inj-on*  $g \ (\text{dom } (f \circ g))$   
**shows** *finite*  $(\text{dom } (f \circ g))$   
*<proof>*

**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>*

**lemma** *set-zip-leftI*:  
**assumes**  $\text{length } xs = \text{length } ys$   
**assumes**  $y \in \text{set } ys$   
**obtains**  $x$  **where**  $(x,y) \in \text{set } (\text{zip } xs \ ys)$   
*<proof>*

**lemma** *butlast-idx*:  
**assumes**  $y \in \text{set } (\text{butlast } xs)$   
**obtains**  $i$  **where**  $xs \ ! \ i = y$   $i < \text{length } xs - 1$   
*<proof>*

**lemma** *butlast-idx'*:  
**assumes**  $xs \ ! \ i = y$   $i < \text{length } xs - 1$   $\text{length } xs > 1$   
**shows**  $y \in \text{set } (\text{butlast } xs)$   
*<proof>*

**lemma** *card-eq-1-singleton*:  
**assumes**  $\text{card } A = 1$   
**obtains**  $x$  **where**  $A = \{x\}$   
*<proof>*

**lemma** *set-take-two*:  
**assumes**  $\text{card } A \geq 2$   
**obtains**  $x \ y$  **where**  $x \in A$   $y \in A$   $x \neq y$   
*<proof>*

**lemma** *singleton-list-hd-last*:  $\text{length } xs = 1 \implies \text{hd } xs = \text{last } xs$   
*<proof>*

**lemma** *distinct-hd-tl*:  $\text{distinct } xs \implies \text{hd } xs \notin \text{set } (\text{tl } xs)$   
*<proof>*

**lemma** *set-mono-strict-prefix*:  $\text{strict-prefix } xs \ ys \implies \text{set } xs \subseteq \text{set } (\text{butlast } ys)$

*<proof>*

**lemma** *set-butlast-distinct*:  $distinct\ xs \implies set\ (butlast\ xs) \cap \{last\ xs\} = \{\}$   
*<proof>*

**lemma** *disjoint-elim*[*elim*]:  $A \cap B = \{\} \implies x \in A \implies x \notin B$  *<proof>*

**lemma** *prefix-butlastD*[*elim*]:  $prefix\ xs\ (butlast\ ys) \implies prefix\ xs\ ys$   
*<proof>*

**lemma** *butlast-prefix*:  $prefix\ xs\ ys \implies prefix\ (butlast\ xs)\ (butlast\ ys)$   
*<proof>*

**lemma** *hd-in-butlast*:  $length\ xs > 1 \implies hd\ xs \in set\ (butlast\ xs)$   
*<proof>*

**lemma** *nonsimple-length-gt-1*:  $xs \neq [] \implies hd\ xs \neq last\ xs \implies length\ xs > 1$   
*<proof>*

**lemma** *set-hd-tl*:  $xs \neq [] \implies set\ [hd\ xs] \cup set\ (tl\ xs) = set\ xs$   
*<proof>*

**lemma** *fold-update-conv*:  
 $fold\ (\lambda n\ m.\ m(n \mapsto g\ n))\ xs\ m\ x =$   
*(if*  $(x \in set\ xs)$  *then*  $Some\ (g\ x)$  *else*  $m\ x)$   
*<proof>*

**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**  $P\ xs$   
*<proof>*

**lemma** *The-Min*:  $Ex1\ P \implies The\ P = Min\ \{x.\ P\ x\}$   
*<proof>*

**lemma** *The-Max*:  $Ex1\ P \implies The\ P = Max\ \{x.\ P\ x\}$   
*<proof>*

**lemma** *set-sorted-list-of-set-remove [simp]*:  
 $set\ (sorted-list-of-set\ (Set.remove\ x\ A)) = Set.remove\ x\ (set\ (sorted-list-of-set\ A))$   
*<proof>*

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

**lemma** *set-single-hd*:  $set\ vs = \{v\} \implies hd\ vs = v$   
 ⟨proof⟩

**lemma** *set-double-filter-hd*:  $[[\ set\ vs = \{v,v'\};\ v \neq v' ]] \implies hd\ [v' \leftarrow vs . v' \neq v] = v'$   
 ⟨proof⟩

**lemma** *map-option-the*:  $x = map\ option\ f\ y \implies x \neq None \implies the\ x = f\ (the\ y)$   
 ⟨proof⟩

**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*  $A\ r\ x\ (Suc\ n) = (SOME\ y. y \in A \wedge (iterated-serial-on\ A\ r\ x\ n, y) \in r)$

**lemma** *iterated-serial-on-linear*:  $iterated-serial-on\ A\ r\ x\ (n+m) = iterated-serial-on\ A\ r\ (iterated-serial-on\ A\ r\ x\ n)\ m$   
 ⟨proof⟩

**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⟩

**lemma** *iterated-serial-on-in-power*:

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

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

⟨proof⟩

**lemma** *trancl-powerI*:  $a \in R\ \wedge\ \wedge\ n \implies n > 0 \implies a \in R^+$

⟨proof⟩

**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>*

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

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

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

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

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

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

**is**  $\lambda f. \text{restrict-map } (\text{Some} \circ f)$  *<proof>*

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

**is**  $(++)$  *<proof>*

**definition** *mmap* = *Mapping.map id*

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

**lemma** *Mapping-Mapping-lookup* [*simp*, *code-unfold*]: *Mapping.Mapping* (*Mapping.lookup* *m*) = *m* *<proof>*

**declare** *Mapping.lookup.abs-eq*[*simp*]

**lemma** *Mapping-eq-lookup*:  $m = m' \iff \text{Mapping.lookup } m = \text{Mapping.lookup } m'$  *<proof>*

**lemma** *map-of-map-if-conv*:

*map-of* (*map*  $(\lambda k. (k, f k))$  *xs*) *x* = (*if*  $(x \in \text{set } xs)$  then *Some*  $(f x)$  else None) *<proof>*

**lemma** *Mapping-lookup-map*: *Mapping.lookup* (*Mapping.map* *f g m*) *a* = *map-option* *g* (*Mapping.lookup* *m*  $(f a)$ )

*<proof>*

**lemma** *Mapping-lookup-map-default*: *Mapping.lookup* (*Mapping.map-default* *k d f m*) *k'* = (*if*  $k = k'$

then  $(\text{Some} \circ f)$  (case *Mapping.lookup* *m* *k* of None  $\Rightarrow d$  | *Some*  $x \Rightarrow x$ )

*else Mapping.lookup m k')*  
(proof)

**lemma** *Mapping-lookup-mapping-add*: *Mapping.lookup (mapping-add m1 m2) k = case-option (Mapping.lookup m1 k) Some (Mapping.lookup m2 k)*  
(proof)

**lemma** *Mapping-lookup-map-values*: *Mapping.lookup (map-values f m) k = Option.bind (Mapping.lookup m k) (f k)*  
(proof)

**lemma** *lookup-fold-update [simp]*: *Mapping.lookup (fold ( $\lambda n. Mapping.update n (g n)$ ) xs m) x = (if (x  $\in$  set xs) then Some (g x) else Mapping.lookup m x)*  
(proof)

**lemma** *mapping-eq-iff*: *m1 = m2  $\longleftrightarrow$  ( $\forall k. Mapping.lookup m1 k = Mapping.lookup m2 k$ )*  
(proof)

**lemma** *lookup-delete*: *Mapping.lookup (Mapping.delete k m) k' = (if k = k' then None else Mapping.lookup m k')*  
(proof)

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

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

**lemma** *lookup-update-cases*: *Mapping.lookup (Mapping.update k v m) k' = (if k=k' then Some v else Mapping.lookup m k')*  
(proof)

**end**

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

**imports**

*Mapping-Exts*  
*HOL-Library.RBT-Mapping*  
*HOL-Library.RBT-Set*

**begin**

**lemma** *restrict-mapping-code [code]*:  
*restrict-mapping f (RBT-Set.Set r) = RBT-Mapping.Mapping (RBT.map ( $\lambda a . f a$ ) r)*



*<proof>*

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

**assumes** *inj f*

**shows** *map-keys f (RBT-Mapping.Mapping t) = RBT.fold (λx v m. Mapping.update (f x) v m) t Mapping.empty*

*<proof>*

**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>*

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

*<proof>*

**lemma** *mmap-rbt-code* [*code*]: *mmap f (RBT-Mapping.Mapping t) = RBT-Mapping.Mapping (RBT.map (λ-. f) t)*

*<proof>*

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

*<proof>*

**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**

$getTo \equiv snd \circ snd$

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

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

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

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

$\langle proof \rangle$

Predecessors of a node.

**definition**  $pred :: ('v,'w) graph \Rightarrow 'v \Rightarrow ('v \times 'w) set$

**where**  $pred G v \equiv \{(v',w). (v',w,v) \in edges G\}$

**lemma** *pred-finite*[*simp, intro*]:  $finite (edges G) \implies finite (pred G v)$

$\langle proof \rangle$

**lemma** *pred-empty*[*simp*]:  $pred empty v = \{\}$   $\langle proof \rangle$

**lemma** (in *valid-graph*) *pred-subset*:  $pred G v \subseteq V \times UNIV$

$\langle proof \rangle$

**type-synonym**  $('V,'W,'\sigma,'G) graph-pred-it =$

$'G \Rightarrow 'V \Rightarrow ('V \times 'W, '\sigma) set-iterator$

**locale** *graph-pred-it-defs* =

**fixes** *pred-list-it* ::  $'G \Rightarrow 'V \Rightarrow ('V \times 'W, ('V \times 'W) list) set-iterator$

**begin**

**definition** *pred-it g v*  $\equiv it-to-it (pred-list-it g v)$

**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* ::  $'G \Rightarrow 'V \Rightarrow ('V \times 'W, ('V \times 'W) list) set-iterator$  +

**assumes** *pred-list-it-correct*:

$invar g \implies set-iterator (pred-list-it g v) (pred (\alpha g) v)$

**begin**

**lemma** *pred-it-correct*:

$invar g \implies set-iterator (pred-it g v) (pred (\alpha g) v)$

$\langle proof \rangle$

**lemma** *pi-pred-it*[*icf-proper-iteratorI*]:

*proper-it*  $(pred-it S v) (pred-it S v)$

$\langle proof \rangle$

**lemma** *pred-it-proper*[*proper-it*]:

*proper-it'*  $(\lambda S. pred-it S v) (\lambda S. pred-it S v)$

$\langle proof \rangle$

**end**

**record**  $('V, 'W, 'G) graph-ops = ('V, 'W, 'G) GraphSpec.graph-ops +$

*gop-pred-list-it* :: 'G ⇒ 'V ⇒ ('V × 'W, ('V × 'W) list) set-iterator

**lemma** (in *graph-pred-it*) *pred-it-is-iterator*[*refine-transfer*]:  
*invar g* ⇒ *set-iterator* (*pred-it g v*) (*pred* (α *g*) *v*)  
 ⟨*proof*⟩

**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* ≡ *gop-pred-list-it ops*  
**end**

**locale** *StdGraph* = *StdGraphDefs* + *org.StdGraph* +  
*graph-pred-it* α *invar pred-list-it*

**locale** *graph-path-base* =  
*graph-nodes-it-defs* λ*g*. *foldri* (α *n g*) +  
*graph-pred-it-defs* λ*g n*. *foldri* (*inEdges'* *g n*)  
**for**  
*αe* :: 'g ⇒ ('node × 'edgeD × 'node) set **and**  
*αn* :: 'g ⇒ 'node list **and**  
*invar* :: 'g ⇒ bool **and**  
*inEdges'* :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list  
**begin**

**definition** *inEdges* :: 'g ⇒ 'node ⇒ ('node × 'edgeD × 'node) list  
**where** *inEdges g n* ≡ *map* (λ(*f, d*). (*f, d, n*)) (*inEdges'* *g n*)

**definition** *predecessors* :: 'g ⇒ 'node ⇒ 'node list **where**  
*predecessors g n* ≡ *map getFrom* (*inEdges g n*)

**definition** *successors* :: 'g ⇒ 'node ⇒ 'node list **where**  
*successors g m* ≡ [*n* . *n* ← α *n g*, *m* ∈ *set* (*predecessors g n*)]

**declare** *predecessors-def* [*code*]

**declare** [[*inductive-internals*]]

**inductive** *path* :: 'g ⇒ 'node list ⇒ bool  
**for** *g* :: 'g  
**where**  
*empty-path*[*intro*]: [*n* ∈ *set* (α *n g*); *invar g*] ⇒ *path g* [*n*]  
 | *Cons-path*[*intro*]: [*path g ns*; *n'* ∈ *set* (*predecessors g* (*hd ns*))] ⇒ *path g*  
 (*n' # ns*)

**definition** *path2* :: 'g ⇒ 'node ⇒ 'node list ⇒ 'node ⇒ bool (- ⊢ ---->-)

[51,0,0,51] 80) **where**

$path2\ g\ n\ ns\ m \equiv path\ g\ ns \wedge n = hd\ ns \wedge m = last\ ns$

**abbreviation**  $\alpha\ g \equiv (\!| nodes = set\ (\alpha n\ g), edges = \alpha e\ g |)$   
**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** *an-correct*:  $invar\ g \Longrightarrow set\ (\alpha n\ g) \supseteq getFrom\ ' \alpha e\ g \cup getTo\ ' \alpha e\ g$   
*<proof>*

**lemma** *an-distinct*:  $invar\ g \Longrightarrow distinct\ (\alpha n\ g)$   
*<proof>*

**lemma** *inEdges-correct'*:

**assumes** *invar g*

**shows**  $set\ (inEdges\ g\ n) = (\lambda(f,d). (f,d,n))\ ' (pred\ (\alpha\ g)\ n)$

*<proof>*

**lemma** *inEdges-correct* [*intro!*, *simp*]:

$invar\ g \Longrightarrow set\ (inEdges\ g\ n) = \{(-, -, t). t = n\} \cap \alpha e\ g$

*<proof>*

**lemma** *in-set-anI1* [*intro*]:  $[[invar\ g; x \in getFrom\ ' \alpha e\ g]] \Longrightarrow x \in set\ (\alpha n\ g)$   
*<proof>*

**lemma** *in-set-anI2* [*intro*]:  $[[invar\ g; x \in getTo\ ' \alpha e\ g]] \Longrightarrow x \in set\ (\alpha n\ g)$   
*<proof>*

**lemma** *edge-to-node*:

**assumes** *invar g* **and**  $e \in \alpha e\ g$

**obtains**  $getFrom\ e \in set\ (\alpha n\ g)$  **and**  $getTo\ e \in set\ (\alpha n\ g)$

*<proof>*

**lemma** *inEdge-to-edge*:

**assumes**  $e \in set\ (inEdges\ g\ n)$  **and** *invar g*

**obtains**  $eD\ n'$  **where**  $(n',eD,n) \in \alpha e\ g$

$\langle \text{proof} \rangle$

**lemma** *edge-to-inEdge*:

**assumes**  $(n, eD, m) \in \alpha e g \text{ invar } g$

**obtains**  $(n, eD, m) \in \text{set } (\text{inEdges } g m)$

$\langle \text{proof} \rangle$

**lemma** *edge-to-predecessors*:

**assumes**  $(n, eD, m) \in \alpha e g \text{ invar } g$

**obtains**  $n \in \text{set } (\text{predecessors } g m)$

$\langle \text{proof} \rangle$

**lemma** *predecessor-is-node[elim]*:  $\llbracket n \in \text{set } (\text{predecessors } g n'); \text{ invar } g \rrbracket \implies n \in \text{set } (\alpha n g)$

$\langle \text{proof} \rangle$

**lemma** *successor-is-node[elim]*:  $\llbracket n \in \text{set } (\text{predecessors } g n'); n \in \text{set } (\alpha n g); \text{ invar } g \rrbracket \implies n' \in \text{set } (\alpha n g)$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *path-not-Nil[simp, dest]*:  $\text{path } g ns \implies ns \neq []$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *path2-cases*:

**assumes**  $g \vdash n - ns \rightarrow m$

**obtains**  $(\text{empty-path}) ns = [n] m = n$

|  $(\text{Cons-path}) g \vdash \text{hd } (tl ns) - tl ns \rightarrow m \ n \in \text{set } (\text{predecessors } g (\text{hd } (tl ns)))$

$\langle \text{proof} \rangle$

**lemma** *path2-induct*[*consumes 1, case-names empty-path Cons-path*]:  
**assumes**  $g \vdash n - ns \rightarrow m$   
**assumes** *empty*:  $invar\ g \implies P\ m\ [m]\ m$   
**assumes** *Cons*:  $\bigwedge ns\ n'\ n.\ g \vdash n - ns \rightarrow m \implies P\ n\ ns\ m \implies n' \in set\ (predecessors\ g\ n) \implies P\ n'\ (n' \# ns)\ m$   
**shows**  $P\ n\ ns\ m$   
*<proof>*

**lemma** *path-invar*[*intro*]:  $path\ g\ ns \implies invar\ g$   
*<proof>*

**lemma** *path-in- $\alpha n$* [*intro*]:  $\llbracket path\ g\ ns; n \in set\ ns \rrbracket \implies n \in set\ (\alpha n\ g)$   
*<proof>*

**lemma** *path2-in- $\alpha n$* [*elim*]:  $\llbracket g \vdash n - ns \rightarrow m; l \in set\ ns \rrbracket \implies l \in set\ (\alpha n\ g)$   
*<proof>*

**lemma** *path2-hd-in- $\alpha n$* [*elim*]:  $g \vdash n - ns \rightarrow m \implies n \in set\ (\alpha n\ g)$   
*<proof>*

**lemma** *path2-tl-in- $\alpha n$* [*elim*]:  $g \vdash n - ns \rightarrow m \implies m \in set\ (\alpha n\ g)$   
*<proof>*

**lemma** *path2-forget-hd*[*simp*]:  $g \vdash n - ns \rightarrow m \implies g \vdash hd\ ns - ns \rightarrow m$   
*<proof>*

**lemma** *path2-forget-last*[*simp*]:  $g \vdash n - ns \rightarrow m \implies g \vdash n - ns \rightarrow last\ ns$   
*<proof>*

**lemma** *path-hd*[*dest*]:  $path\ g\ (n \# ns) \implies path\ g\ [n]$   
*<proof>*

**lemma** *path-by-tail*[*intro*]:  $\llbracket path\ g\ (n \# n' \# ns); path\ g\ (n' \# ns) \rrbracket \implies path\ g\ (n' \# ms)$   
 $\implies path\ g\ (n \# n' \# ms)$   
*<proof>*

**lemma**  *$\alpha n$ -in- $\alpha nE$*  [*elim*]:  
**assumes**  $(n, e, m) \in \alpha e\ g$  **and**  $invar\ g$   
**obtains**  $n \in set\ (\alpha n\ g)$  **and**  $m \in set\ (\alpha n\ g)$   
*<proof>*

**lemma** *path-split*:  
**assumes**  $path\ g\ (ns @ m \# ns')$   
**shows**  $path\ g\ (ns @ [m])\ path\ g\ (m \# ns')$   
*<proof>*

**lemma** *path2-split*:  
**assumes**  $g \vdash n - ns @ n' \# ns' \rightarrow m$

**shows**  $g \vdash n - ns@[n'] \rightarrow n' \quad g \vdash n' - n' \# ns' \rightarrow m$   
 ⟨proof⟩

**lemma** *elem-set-implies-elem-tl-app-cons[simp]*:  $x \in \text{set } xs \implies x \in \text{set } (tl (ys@y\#xs))$   
 ⟨proof⟩

**lemma** *path2-split-ex*:

**assumes**  $g \vdash n - ns \rightarrow m \quad x \in \text{set } ns$

**obtains**  $ns_1 \ ns_2$  **where**  $g \vdash n - ns_1 \rightarrow x \quad g \vdash x - ns_2 \rightarrow m \quad ns = ns_1 @ tl \ ns_2 \quad ns = \text{butlast } ns_1 @ ns_2$   
 ⟨proof⟩

**lemma** *path2-split-ex'*:

**assumes**  $g \vdash n - ns \rightarrow m \quad x \in \text{set } ns$

**obtains**  $ns_1 \ ns_2$  **where**  $g \vdash n - ns_1 \rightarrow x \quad g \vdash x - ns_2 \rightarrow m \quad ns = \text{butlast } ns_1 @ ns_2$   
 ⟨proof⟩

**lemma** *path-snoc*:

**assumes**  $\text{path } g \ (ns@[n]) \quad n \in \text{set } (\text{predecessors } g \ m)$

**shows**  $\text{path } g \ (ns@[n,m])$

⟨proof⟩

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

**assumes**  $g \vdash n - ns \rightarrow m \quad m \in \text{set } (\text{predecessors } g \ m')$

**shows**  $g \vdash n - ns@[m'] \rightarrow m'$

⟨proof⟩

**lemma** *path-unsnoc*:

**assumes**  $\text{path } g \ ns \quad \text{length } ns \geq 2$

**obtains**  $\text{path } g \ (\text{butlast } ns) \wedge \text{last } (\text{butlast } ns) \in \text{set } (\text{predecessors } g \ (\text{last } ns))$

⟨proof⟩

**lemma** *path2-unsnoc*:

**assumes**  $g \vdash n - ns \rightarrow m \quad \text{length } ns \geq 2$

**obtains**  $g \vdash n - \text{butlast } ns \rightarrow \text{last } (\text{butlast } ns) \quad \text{last } (\text{butlast } ns) \in \text{set } (\text{predecessors } g \ m)$

⟨proof⟩

**lemma** *path2-rev-induct[consumes 1, case-names empty snoc]*:

**assumes**  $g \vdash n - ns \rightarrow m$

**assumes** *empty*:  $n \in \text{set } (\alpha n \ g) \implies P \ n \ [n] \ n$

**assumes** *snoc*:  $\bigwedge ns \ m' \ m. \ g \vdash n - ns \rightarrow m' \implies P \ n \ ns \ m' \implies m' \in \text{set } (\text{predecessors } g \ m) \implies P \ n \ (ns@[m]) \ m$

**shows**  $P \ n \ ns \ m$

⟨proof⟩

**lemma** *path2-hd[elim, dest?]*:  $g \vdash n - ns \rightarrow m \implies n = \text{hd } ns$

⟨proof⟩

**lemma** *path2-hd-in-ns*[*elim*]:  $g \vdash n - ns \rightarrow m \implies n \in \text{set } ns$   
 ⟨*proof*⟩

**lemma** *path2-last*[*elim, dest?*]:  $g \vdash n - ns \rightarrow m \implies m = \text{last } ns$   
 ⟨*proof*⟩

**lemma** *path2-last-in-ns*[*elim*]:  $g \vdash n - ns \rightarrow m \implies m \in \text{set } ns$   
 ⟨*proof*⟩

**lemma** *path-app*[*elim*]:  
**assumes** *path*  $g \ ns \ path \ g \ ms \ \text{last } ns = \text{hd } ms$   
**shows** *path*  $g \ (ns @ tl \ ms)$   
 ⟨*proof*⟩

**lemma** *path2-app*[*elim*]:  
**assumes**  $g \vdash n - ns \rightarrow m \ g \vdash m - ms \rightarrow l$   
**shows**  $g \vdash n - ns @ tl \ ms \rightarrow l$   
 ⟨*proof*⟩

**lemma** *butlast-tl*:  
**assumes**  $\text{last } xs = \text{hd } ys \ xs \neq [] \ ys \neq []$   
**shows**  $\text{butlast } xs @ ys = xs @ tl \ ys$   
 ⟨*proof*⟩

**lemma** *path2-app'*[*elim*]:  
**assumes**  $g \vdash n - ns \rightarrow m \ g \vdash m - ms \rightarrow l$   
**shows**  $g \vdash n - \text{butlast } ns @ ms \rightarrow l$   
 ⟨*proof*⟩

**lemma** *path2-nontrivial*[*elim*]:  
**assumes**  $g \vdash n - ns \rightarrow m \ n \neq m$   
**shows**  $\text{length } ns \geq 2$   
 ⟨*proof*⟩

**lemma** *simple-path2-aux*:  
**assumes**  $g \vdash n - ns \rightarrow m$   
**obtains**  $ns'$  **where**  $g \vdash n - ns' \rightarrow m \ \text{distinct } ns' \ \text{set } ns' \subseteq \text{set } ns \ \text{length } ns' \leq$   
 $\text{length } ns$   
 ⟨*proof*⟩

**lemma** *simple-path2*:  
**assumes**  $g \vdash n - ns \rightarrow m$   
**obtains**  $ns'$  **where**  $g \vdash n - ns' \rightarrow m \ \text{distinct } ns' \ \text{set } ns' \subseteq \text{set } ns \ \text{length } ns' \leq$   
 $\text{length } ns \ n \notin \text{set } (tl \ ns') \ m \notin \text{set } (\text{butlast } ns')$   
 ⟨*proof*⟩

**lemma** *simple-path2-unsnoc*:  
**assumes**  $g \vdash n - ns \rightarrow m \ n \neq m$   
**obtains**  $ns'$  **where**  $g \vdash n - ns' \rightarrow \text{last } ns' \ \text{last } ns' \in \text{set } (\text{predecessors } g \ m)$



*distinct ns' set ns'  $\subseteq$  set ns m  $\notin$  set ns'*  
 ⟨proof⟩

**lemma path2-split-first-last:**

**assumes**  $g \vdash n - ns \rightarrow m$   $x \in \text{set } ns$

**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**  $g \vdash n - ns_1 @ [x] \rightarrow x$   $x \notin \text{set } ns_1$

**and**  $g \vdash x - ns_3 \rightarrow x$

**and**  $g \vdash x - x \# ns_2 \rightarrow m$   $x \notin \text{set } ns_2$

⟨proof⟩

**lemma path2-simple-loop:**

**assumes**  $g \vdash n - ns \rightarrow n$   $n' \in \text{set } ns$

**obtains**  $ns'$  **where**  $g \vdash n - ns' \rightarrow n$   $n' \in \text{set } ns'$   $n \notin \text{set } (tl (\text{butlast } ns'))$  *set ns'*  
 $\subseteq \text{set } ns$

⟨proof⟩

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

**assumes**  $g \vdash n - ns \rightarrow m$   $\exists x \in \text{set } ns. P x$

**obtains**  $m'$   $ns'$  **where**  $g \vdash n - ns' \rightarrow m'$   $P m' \forall x \in \text{set } (\text{butlast } ns'). \neg P x$  *prefix*  
 $ns' ns$

⟨proof⟩

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

**assumes**  $g \vdash n - ns \rightarrow m$   $\exists x \in \text{set } ns. P x$

**obtains**  $n'$   $ns'$  **where**  $g \vdash n' - ns' \rightarrow m$   $P n' \forall x \in \text{set } (tl ns'). \neg P x$  *suffix ns'*  
 $ns$

⟨proof⟩

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

**assumes** 1:  $g \vdash n - ns \rightarrow m$

**assumes** 2: *prefix*  $(ns' @ [m'])$  *ns*

**shows**  $g \vdash n - ns' @ [m'] \rightarrow m'$

⟨proof⟩

**lemma path2-prefix-ex:**

**assumes**  $g \vdash n - ns \rightarrow m$   $m' \in \text{set } ns$

**obtains**  $ns'$  **where**  $g \vdash n - ns' \rightarrow m'$  *prefix ns' ns m'  $\notin$  set (butlast ns')*

⟨proof⟩

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

**assumes**  $g \vdash n - ns \rightarrow m$   $m' \in \text{set } (\text{butlast } ns)$

**obtains**  $ns'$  **where**  $g \vdash n - ns' \rightarrow m'$  *strict-prefix ns' ns m'  $\notin$  set (butlast ns')*

⟨proof⟩

**lemma path2-nontriv[elim]:**  $\llbracket g \vdash n - ns \rightarrow m; n \neq m \rrbracket \implies \text{length } ns > 1$

⟨proof⟩

```

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  $g n m \equiv m \in set (\alpha n g) \wedge (\forall ns. g \vdash Entry\ g - ns \rightarrow m \longrightarrow n \in set\ ns)$ 

  abbreviation strict-dom  $g n m \equiv n \neq m \wedge dominates\ g\ n\ m$ 
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]:  $Entry\ g \in set (\alpha n g)$ 
assumes Entry-unreachable:  $invar\ g \implies inEdges\ g (Entry\ g) = []$ 
assumes Entry-reaches[intro]:
   $\llbracket n \in set (\alpha n g); invar\ g \rrbracket \implies \exists ns. g \vdash Entry\ g - ns \rightarrow n$ 
begin
  lemma Entry-dominates[simp,intro]:  $\llbracket invar\ g; n \in set (\alpha n g) \rrbracket \implies dominates\ g (Entry\ g)\ n$ 
  <proof>

  lemma Entry-iff-unreachable[simp]:
    assumes  $invar\ g\ n \in set (\alpha n g)$ 
    shows  $predecessors\ g\ n = [] \iff n = Entry\ g$ 
    <proof>

  lemma Entry-loop:
    assumes  $invar\ g\ g \vdash Entry\ g - ns \rightarrow Entry\ g$ 

```

**shows**  $ns = [Entry\ g]$   
 $\langle proof \rangle$

**lemma** *simple-Entry-path*:  
**assumes**  $invar\ g\ n \in set\ (\alpha n\ g)$   
**obtains**  $ns$  **where**  $g \vdash Entry\ g - ns \rightarrow n$  **and**  $n \notin set\ (butlast\ ns)$   
 $\langle proof \rangle$

**lemma** *dominatesI* [intro]:  
 $\llbracket m \in set\ (\alpha n\ g); \bigwedge ns. \llbracket g \vdash Entry\ g - ns \rightarrow m \rrbracket \implies n \in set\ ns \rrbracket \implies dominates\ g\ n\ m$   
 $\langle proof \rangle$

**lemma** *dominatesE*:  
**assumes**  $dominates\ g\ n\ m$   
**obtains**  $m \in set\ (\alpha n\ g)$  **and**  $\bigwedge ns. g \vdash Entry\ g - ns \rightarrow m \implies n \in set\ ns$   
 $\langle proof \rangle$

**lemma**[simp]:  $dominates\ g\ n\ m \implies m \in set\ (\alpha n\ g)$   $\langle proof \rangle$

**lemma**[simp]:  
**assumes**  $dominates\ g\ n\ m$  **and**[simp]:  $invar\ g$   
**shows**  $n \in set\ (\alpha n\ g)$   
 $\langle proof \rangle$

**lemma** *strict-domE*[elim]:  
**assumes**  $strict-dom\ g\ n\ m$   
**obtains**  $m \in set\ (\alpha n\ g)$  **and**  $\bigwedge ns. g \vdash Entry\ g - ns \rightarrow m \implies n \in set\ (butlast\ ns)$   
 $\langle proof \rangle$

**lemma** *dominates-refl*[intro!]:  $\llbracket invar\ g; n \in set\ (\alpha n\ g) \rrbracket \implies dominates\ g\ n\ n$   
 $\langle proof \rangle$

**lemma** *dominates-trans*:  
**assumes**  $invar\ g$   
**assumes**  $part1: dominates\ g\ n\ n'$   
**assumes**  $part2: dominates\ g\ n'\ n''$   
**shows**  $dominates\ g\ n\ n''$   
 $\langle proof \rangle$

**lemma** *dominates-antisymm*:  
**assumes**  $invar\ g$   
**assumes**  $dom1: dominates\ g\ n\ n'$   
**assumes**  $dom2: dominates\ g\ n'\ n$   
**shows**  $n = n'$   
 $\langle proof \rangle$

**lemma** *dominates-unsnoc*:

**assumes**  $[simp]: \text{invar } g \text{ and } \text{dominates } g \ n \ m \ m' \in \text{set } (\text{predecessors } g \ m) \ n \neq m$

**shows**  $\text{dominates } g \ n \ m'$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{dominates-unsnoc}'$ :

**assumes**  $[simp]: \text{invar } g \text{ and } \text{dominates } g \ n \ m \ g \vdash m' - ms \rightarrow m \ \forall x \in \text{set } (tl \ ms). \ x \neq n$

**shows**  $\text{dominates } g \ n \ m'$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{dominates-path}$ :

**assumes**  $\text{dominates } g \ n \ m \text{ and } [simp]: \text{invar } g$   
**obtains**  $ns \text{ where } g \vdash n - ns \rightarrow m$

$\langle \text{proof} \rangle$

**lemma**  $\text{dominates-antitrans}$ :

**assumes**  $[simp]: \text{invar } g \text{ and } \text{dominates } g \ n_1 \ m \ \text{dominates } g \ n_2 \ m$   
**obtains**  $(1) \text{ dominates } g \ n_1 \ n_2$

$| (2) \text{ dominates } g \ n_2 \ n_1$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{dominates-extend}$ :

**assumes**  $\text{dominates } g \ n \ m$   
**assumes**  $g \vdash m' - ms \rightarrow m \ n \notin \text{set } (tl \ ms)$   
**shows**  $\text{dominates } g \ n \ m'$

$\langle \text{proof} \rangle$

**definition**  $\text{dominators} :: 'g \Rightarrow 'node \Rightarrow 'node \text{ set } \text{ where}$

$\text{dominators } g \ n \equiv \{m \in \text{set } (\alpha n \ g). \ \text{dominates } g \ m \ n\}$

**definition**  $\text{isIdom } g \ n \ m \longleftrightarrow \text{strict-dom } g \ m \ n \wedge (\forall m' \in \text{set } (\alpha n \ g). \ \text{strict-dom } g \ m' \ n \longrightarrow \text{dominates } g \ m' \ m)$

**definition**  $\text{idom} :: 'g \Rightarrow 'node \Rightarrow 'node \text{ where}$

$\text{idom } g \ n \equiv \text{THE } m. \ \text{isIdom } g \ n \ m$

**lemma**  $\text{idom-ex}$ :

**assumes**  $[simp]: \text{invar } g \ n \in \text{set } (\alpha n \ g) \ n \neq \text{Entry } g$   
**shows**  $\exists ! m. \ \text{isIdom } g \ n \ m$

$\langle \text{proof} \rangle$

**lemma**  $\text{idom}: \llbracket \text{invar } g; \ n \in \text{set } (\alpha n \ g) - \{\text{Entry } g\} \rrbracket \Longrightarrow \text{isIdom } g \ n \ (\text{idom } g \ n)$

$\langle \text{proof} \rangle$

**lemma**  $\text{dominates-mid}$ :

**assumes**  $\text{dominates } g \ n \ x \ \text{dominates } g \ x \ m \ g \vdash n - ns \rightarrow m \text{ and } [simp]: \text{invar } g$   
**shows**  $x \in \text{set } ns$

$\langle \text{proof} \rangle$

**definition** *shortestPath* :: 'g ⇒ 'node ⇒ nat **where**  
*shortestPath* g n ≡ (LEAST l. ∃ ns. length ns = l ∧ g ⊢ Entry g-ns→n)

**lemma** *shortestPath-ex*:

**assumes** n ∈ set (αn g) *invar* g

**obtains** ns **where** g ⊢ Entry g-ns→n *distinct* ns length ns = *shortestPath* g

n

⟨*proof*⟩

**lemma**<sub>[simp]</sub>: [n ∈ set (αn g); *invar* g] ⇒ *shortestPath* g n ≠ 0

⟨*proof*⟩

**lemma** *shortestPath-upper-bound*:

**assumes** n ∈ set (αn g) *invar* g

**shows** *shortestPath* g n ≤ length (αn g)

⟨*proof*⟩

**lemma** *shortestPath-predecessor*:

**assumes** n ∈ set (αn g) − {Entry g} **and**<sub>[simp]</sub>: *invar* g

**obtains** n' **where** Suc (*shortestPath* g n') = *shortestPath* g n n' ∈ set

(*predecessors* g n)

⟨*proof*⟩

**lemma** *successor-in-αn*<sub>[simp]</sub>:

**assumes** *predecessors* g n ≠ [] **and**<sub>[simp]</sub>: *invar* g

**shows** n ∈ set (αn g)

⟨*proof*⟩

**lemma** *shortestPath-single-predecessor*:

**assumes** *predecessors* g n = [m] **and**<sub>[simp]</sub>: *invar* g

**shows** *shortestPath* g m < *shortestPath* g n

⟨*proof*⟩

**lemma** *strict-dom-shortestPath-order*:

**assumes** *strict-dom* g n m m ∈ set (αn g) *invar* g

**shows** *shortestPath* g n < *shortestPath* g m

⟨*proof*⟩

**lemma** *dominates-shortestPath-order*:

**assumes** *dominates* g n m m ∈ set (αn g) *invar* g

**shows** *shortestPath* g n ≤ *shortestPath* g m

⟨*proof*⟩

**lemma** *strict-dom-trans*:

**assumes**<sub>[simp]</sub>: *invar* g

**assumes** *strict-dom* g n m *strict-dom* g m m'

**shows** *strict-dom* g n m'

⟨*proof*⟩

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

**lemma**[simp]:  
**assumes** *EntryPath* g ns *prefix* ns' ns ns' ≠ []  
**shows** *EntryPath* g ns'  
 ⟨*proof*⟩

**lemma** *EntryPath-suffix*:  
**assumes** *EntryPath* g ns *suffix* ns' ns ns' ≠ []  
**shows** *EntryPath* g ns'  
 ⟨*proof*⟩

**lemma** *EntryPath-butlast-less-last*:  
**assumes** *EntryPath* g ns z ∈ *set* (*butlast* ns)  
**shows** *shortestPath* g z < *shortestPath* g (last ns)  
 ⟨*proof*⟩

**lemma** *EntryPath-distinct*:  
**assumes** *EntryPath* g ns  
**shows** *distinct* ns  
 ⟨*proof*⟩

**lemma** *Entry-reachesE*:  
**assumes** n ∈ *set* (αn g) **and**[simp]: *invar* g  
**obtains** ns **where** g ⊢ *Entry* g-ns→n *EntryPath* g ns  
 ⟨*proof*⟩

**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)
  ⟨proof⟩

  lemma Entry-no-predecessor[simp]: predecessors g (Entry g) = []
  ⟨proof⟩

  lemma uses-in-vars[elim, simp]: v ∈ uses g n ⇒ v ∈ vars g
  ⟨proof⟩

  lemma varsE:
    assumes v ∈ vars g
    obtains n where n ∈ set (αn g) v ∈ uses g n
  ⟨proof⟩

  lemma defs-uses-disjoint'[simp]: n ∈ set (αn g) ⇒ v ∈ defs g n ⇒ v ∈ uses
  g n ⇒ False
  ⟨proof⟩
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

```

*<proof>*

**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*

*<proof>*

**end**

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

**locale** *CFG-wf = CFG 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 +*

**assumes** *def-ass-uses: ∀ m ∈ set (αn g). ∀ v ∈ uses g m. defAss' g m v*

## 2.4 SSA CFG

**type-synonym** *('node, 'val)phis = 'node × 'val → 'val list*

**declare** *in-set-zipE[elim]*

**declare** *zip-same[simp]*

**locale** *CFG-SSA-base = CFG-base 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 ⇒ 'val::linorder set and*

*uses :: 'g ⇒ 'node ⇒ 'val set +*

**fixes** *phis :: 'g ⇒ ('node, 'val)phis*

**begin**

**definition** *phiDefs g n ≡ {v. (n,v) ∈ dom (phis g)}*

**definition***[code]: allDefs g n ≡ defs g n ∪ phiDefs g n*

**definition***[code]: phiUses g n ≡*

*∪ n' ∈ set (successors g n). ∪ v' ∈ phiDefs g n'. snd ' Set.filter (λ(n'',v). n'' = n) (set (zip (predecessors g n') (the (phis g (n',v')))))*

**definition***[code]: allUses g n ≡ uses g n ∪ phiUses g n*

**definition***[code]: allVars g ≡ ∪ n ∈ set (αn g). allDefs g n ∪ allUses g n*



**definition**  $defAss :: 'g \Rightarrow 'node \Rightarrow 'val \Rightarrow bool$  **where**  
 $defAss\ g\ m\ v \iff (\forall ns. g \vdash Entry\ g\ ns \rightarrow m \longrightarrow (\exists n \in set\ ns. v \in allDefs\ g\ n))$

**lemmas**  $CFG\text{-}SSA\text{-}defs = phiDefs\text{-}def\ allDefs\text{-}def\ phiUses\text{-}def\ allUses\text{-}def\ allVars\text{-}def\ defAss\text{-}def$   
**end**

**locale**  $CFG\text{-}SSA = CFG\ \alpha e\ \alpha n\ invar\ inEdges'\ Entry\ defs\ uses + CFG\text{-}SSA\text{-}base$   
 $\alpha e\ \alpha n\ invar\ inEdges'\ Entry\ defs\ uses\ phis$   
**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**  $phis\text{-}finite: finite\ (dom\ (phis\ g))$   
**assumes**  $phis\text{-}in\text{-}\alpha n: phis\ g\ (n, v) = Some\ vs \implies n \in set\ (\alpha n\ g)$   
**assumes**  $phis\text{-}wf:$   
 $phis\ g\ (n, v) = Some\ args \implies length\ (predecessors\ g\ n) = length\ args$   
**assumes**  $simpleDefs\text{-}phiDefs\text{-}disjoint:$   
 $n \in set\ (\alpha n\ g) \implies defs\ g\ n \cap phiDefs\ g\ n = \{\}$   
**assumes**  $allDefs\text{-}disjoint:$   
 $\llbracket n \in set\ (\alpha n\ g); m \in set\ (\alpha n\ g); n \neq m \rrbracket \implies allDefs\ g\ n \cap allDefs\ g\ m = \{\}$

**begin**

**lemma**  $phis\text{-}disj:$   
**assumes**  $phis\ g\ (n, v) = Some\ vs$   
**and**  $phis\ g\ (n', v) = Some\ vs'$   
**shows**  $n = n'$  **and**  $vs = vs'$   
 $\langle proof \rangle$

**lemma**  $allDefs\text{-}disjoint': \llbracket n \in set\ (\alpha n\ g); m \in set\ (\alpha n\ g); v \in allDefs\ g\ n; v \in allDefs\ g\ m \rrbracket \implies n = m$   
 $\langle proof \rangle$

**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**  $v \in phiUses\ g\ n$   
 $\langle proof \rangle$

**lemma**  $phiUsesE:$   
**assumes**  $v \in phiUses\ g\ n$   
**obtains**  $n'\ v'\ vs$  **where**  $n' \in set\ (successors\ g\ n)\ (n, v) \in set\ (zip\ (predecessors\ g\ n')\ vs)\ phis\ g\ (n', v') = Some\ vs$   
 $\langle proof \rangle$

**lemma** *defs-in-allDefs*[simp]:  $v \in \text{defs } g \ n \implies v \in \text{allDefs } g \ n$  *<proof>*  
**lemma** *phiDefs-in-allDefs*[simp, elim]:  $v \in \text{phiDefs } g \ n \implies v \in \text{allDefs } g \ n$  *<proof>*  
**lemma** *uses-in-allUses*[simp]:  $v \in \text{uses } g \ n \implies v \in \text{allUses } g \ n$  *<proof>*  
**lemma** *phiUses-in-allUses*[simp]:  $v \in \text{phiUses } g \ n \implies v \in \text{allUses } g \ n$  *<proof>*  
**lemma** *allDefs-in-allVars*[simp, intro]:  $\llbracket v \in \text{allDefs } g \ n; n \in \text{set } (\alpha n \ g) \rrbracket \implies v \in \text{allVars } g$  *<proof>*  
**lemma** *allUses-in-allVars*[simp, intro]:  $\llbracket v \in \text{allUses } g \ n; n \in \text{set } (\alpha n \ g) \rrbracket \implies v \in \text{allVars } g$  *<proof>*

**lemma** *phiDefs-finite*[simp]: *finite* (*phiDefs* *g n*) *<proof>*

**lemma** *phiUses-finite*[simp]:  
**assumes**  $n \in \text{set } (\alpha n \ g)$   
**shows** *finite* (*phiUses* *g n*) *<proof>*

**lemma** *allDefs-finite*[simp]:  $n \in \text{set } (\alpha n \ g) \implies \text{finite } (\text{allDefs } g \ n)$  *<proof>*  
**lemma** *allUses-finite*[simp]:  $n \in \text{set } (\alpha n \ g) \implies \text{finite } (\text{allUses } g \ n)$  *<proof>*  
**lemma** *allVars-finite*[simp]: *finite* (*allVars* *g*) *<proof>*

**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**  $g \vdash n - ns \rightarrow m \ \forall n \in \text{set } (\text{tl } ns). v \notin \text{allDefs } g \ n$   
**shows** *defAss* *g n v* *<proof>*

**lemma** *defAss-dominating*:  
**assumes**[simp]:  $n \in \text{set } (\alpha n \ g)$   
**shows** *defAss* *g n v*  $\longleftrightarrow (\exists m \in \text{set } (\alpha n \ g). \text{dominates } g \ m \ n \wedge v \in \text{allDefs } g \ m)$  *<proof>*

**end**

**locale** *CFG-SSA-wf-base* = *CFG-SSA-base* *ae*  $\alpha n$  *invar* *inEdges'* *Entry* *defs* *uses* *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*  $:: 'g \Rightarrow \text{bool}$  **and**  
*inEdges'*  $:: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD) \text{ list}$  **and**  
*Entry*  $:: 'g \Rightarrow 'node$  **and**  
*defs*  $:: 'g \Rightarrow 'node \Rightarrow 'val :: \text{linorder} \text{ set}$  **and**  
*uses*  $:: 'g \Rightarrow 'node \Rightarrow 'val \text{ set}$  **and**

*phis* :: 'g ⇒ ('node, 'val) *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 ⇒ 'val ⇒ 'node **where**  
*defNode-code* [*code*]: *defNode* g v ≡ hd [n ← αn g. v ∈ *allDefs* g n]

**abbreviation** *def-dominates* g v' v ≡ *dominates* g (*defNode* g v') (*defNode* g v)

**abbreviation** *strict-def-dom* g v' v ≡ *defNode* g v' ≠ *defNode* g v ∧ *def-dominates* g v' v

**definition** *phi* g v = *phis* g (*defNode* g v, v)

**definition**[*simp*]: *phiArg* g v v' ≡ ∃ vs. *phi* g v = *Some* vs ∧ v' ∈ *set* vs

**definition**[*code*]: *isTrivialPhi* g v v' ↔ v' ≠ v ∧  
(case *phi* g v of  
*Some* vs ⇒ *set* vs = {v, v'} ∨ *set* vs = {v'}  
| *None* ⇒ *False*)

**definition**[*code*]: *trivial* g v ≡ ∃ v' ∈ *allVars* g. *isTrivialPhi* g v v'

**definition**[*code*]: *redundant* g ≡ ∃ v ∈ *allVars* g. *trivial* g v

**definition** *defAssUses* g ≡ ∀ n ∈ *set* (αn g). ∀ v ∈ *allUses* g n. *defAss* g n v

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

**declare** [[*inductive-internals*]]

**inductive** *liveVal* :: 'g ⇒ 'val ⇒ *bool*

**for** g :: 'g

**where**

*liveSimple*: [[n ∈ *set* (αn g); val ∈ *uses* g n]] ⇒ *liveVal* g val

| *livePhi*: [[*liveVal* g v; *phiArg* g v v']] ⇒ *liveVal* g v'

**definition** *pruned* g = (∀ n ∈ *set* (αn g). ∀ val. val ∈ *phiDefs* g n → *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* αe αn *invar* *inEdges'* *Entry* *defs* *uses* *phis* +  
*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**

*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:  $v \in \text{allVars } g \implies \exists n \in \text{set } (\alpha n \ g). \ v \in \text{allDefs } g \ n$ 
  <proof>

lemma phiDefs-Entry-empty[simp]:  $\text{phiDefs } g \ (\text{Entry } g) = \{\}$ 
  <proof>

lemma phi-Entry-empty[simp]:  $\text{defNode } g \ v = \text{Entry } g \implies \text{phi } g \ v = \text{None}$ 
  <proof>

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>

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

lemma defNode[simp]:
  assumes  $v \in \text{allVars } g$ 
  shows  $(\text{defNode } g \ v) \in \text{set } (\alpha n \ g) \wedge v \in \text{allDefs } g \ (\text{defNode } g \ v)$ 
  <proof>

lemma defNode-eq[intro]:
  assumes  $n \in \text{set } (\alpha n \ g) \wedge v \in \text{allDefs } g \ n$ 
  shows  $\text{defNode } g \ v = n$ 
  <proof>

lemma defNode-cases[consumes 1]:
  assumes  $v \in \text{allVars } g$ 
  obtains  $(\text{simpleDef}) \ v \in \text{defs } g \ (\text{defNode } g \ v)$ 
  |  $(\text{phi}) \quad \text{phi } g \ v \neq \text{None}$ 
  <proof>

lemma phi-phiDefs[simp]:  $\text{phi } g \ v = \text{Some } vs \implies v \in \text{phiDefs } g \ (\text{defNode } g \ v)$ 
  <proof>

lemma simpleDef-not-phi:
  assumes  $n \in \text{set } (\alpha n \ g) \wedge v \in \text{defs } g \ n$ 
  shows  $\text{phi } g \ v = \text{None}$ 
  <proof>

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

```

*<proof>*

**lemma** *phi-finite*: *finite (dom (phi g))*  
*<proof>*

**lemma** *phiUses-exI*:  
  **assumes**  $m \in \text{set } (\text{predecessors } g \ n)$   $\text{phis } g \ (n,v) = \text{Some } vs$   $n \in \text{set } (\alpha n \ g)$   
  **obtains**  $v'$  **where**  $v' \in \text{phiUses } g \ m$   $v' \in \text{set } vs$   
*<proof>*

**lemma** *phiArg-exI*:  
  **assumes**  $m \in \text{set } (\text{predecessors } g \ (\text{defNode } g \ v))$   $\text{phi } g \ v \neq \text{None}$  **and**[*simp*]:  $v \in \text{allVars } g$   
  **obtains**  $v'$  **where**  $v' \in \text{phiUses } g \ m$   $\text{phiArg } g \ v \ v'$   
*<proof>*

**lemma** *phiUses-exI'*:  
  **assumes**  $\text{phiArg } g \ p \ q$  **and**[*simp*]:  $p \in \text{allVars } g$   
  **obtains**  $m$  **where**  $q \in \text{phiUses } g \ m$   $m \in \text{set } (\text{predecessors } g \ (\text{defNode } g \ p))$   
*<proof>*

**lemma** *phiArg-in-allVars*[*simp*]:  
  **assumes**  $\text{phiArg } g \ v \ v'$   
  **shows**  $v' \in \text{allVars } g$   
*<proof>*

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

**lemma** *defUse-path-ex*:  
  **assumes**  $v \in \text{allUses } g \ m$   $m \in \text{set } (\alpha n \ g)$   
  **obtains**  $ns$  **where**  $g \vdash \text{defNode } g \ v \text{-ns} \rightarrow m$   $\text{EntryPath } g \ ns$   
*<proof>*

**lemma** *defUse-path-dominated*:  
  **assumes**  $g \vdash \text{defNode } g \ v \text{-ns} \rightarrow n$   $\text{defNode } g \ v \notin \text{set } (\text{tl } ns)$   $v \in \text{allUses } g \ n \ n'$   
  **shows**  $\text{dominates } g \ (\text{defNode } g \ v) \ n'$   
*<proof>*

**lemma** *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>*

**lemma** *phiArg-path-ex'*:  
  **assumes**  $\text{phiArg } g \ p \ q$  **and**[*simp*]:  $p \in \text{allVars } g$

**obtains**  $ns\ m$  **where**  $g \vdash \text{defNode } g\ q - ns \rightarrow m \text{ EntryPath } g\ ns\ q \in \text{phiUses } g$   
 $m\ m \in \text{set } (\text{predecessors } g\ (\text{defNode } g\ p))$   
 $\langle \text{proof} \rangle$

**lemma** *phiArg-path-ex*:

**assumes**  $\text{phiArg } g\ p\ q$  **and**  $[\text{simp}]$ :  $p \in \text{allVars } g$   
**obtains**  $ns$  **where**  $g \vdash \text{defNode } g\ q - ns \rightarrow \text{defNode } g\ p \text{ length } ns > 1$   
 $\langle \text{proof} \rangle$

**lemma** *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**  $g \vdash \text{defNode } g\ q - ns \rightarrow \text{defNode } g\ p \text{ length } ns > 1$   
 $\forall n \in \text{set } (\text{butlast } ns). \exists p\ q\ m\ ns'. r\ p\ q \wedge g \vdash \text{defNode } g\ q - ns' \rightarrow m \wedge (\text{defNode } g\ q) \notin \text{set } (\text{tl } ns') \wedge q \in \text{phiUses } g\ m \wedge m \in \text{set } (\text{predecessors } g\ (\text{defNode } g\ p)) \wedge$   
 $n \in \text{set } ns' \wedge \text{set } ns' \subseteq \text{set } ns \wedge \text{defNode } g\ p \in \text{set } ns$   
 $\langle \text{proof} \rangle$

**lemma** *non-dominated-predecessor*:

**assumes**  $n \in \text{set } (\alpha n\ g)\ n \neq \text{Entry } g$   
**obtains**  $m$  **where**  $m \in \text{set } (\text{predecessors } g\ n) \neg \text{dominates } g\ n\ m$   
 $\langle \text{proof} \rangle$

**lemmas**  $\text{dominates-trans}'[\text{trans}, \text{elim}] = \text{dominates-trans}[\text{OF invar}]$

**lemmas**  $\text{strict-dom-trans}'[\text{trans}, \text{elim}] = \text{strict-dom-trans}[\text{OF invar}]$

**lemmas**  $\text{dominates-refl}'[\text{simp}] = \text{dominates-refl}[\text{OF invar}]$

**lemmas**  $\text{dominates-antisymm}'[\text{dest}] = \text{dominates-antisymm}[\text{OF invar}]$

**lemma** *liveVal-in-allVars*  $[\text{simp}]$ :  $\text{liveVal } g\ v \implies v \in \text{allVars } g$

$\langle \text{proof} \rangle$

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

**assumes**  $[\text{simp}]$ :  $p \in \text{allVars } g$  **and**  $\text{phi } g\ p = \text{Some } vs$   
**shows**  $\text{set } vs \neq \{p\}$   
 $\langle \text{proof} \rangle$

**lemma** *phis-phi*:  $\text{phis } g\ (n, v) = \text{Some } vs \implies \text{phi } g\ v = \text{Some } vs$

$\langle \text{proof} \rangle$

**lemma** *trivial-phi*:  $\text{trivial } g\ v \implies \text{phi } g\ v \neq \text{None}$

$\langle \text{proof} \rangle$

**lemma** *trivial-finite*:  $\text{finite } \{v. \text{trivial } g\ v\}$

$\langle \text{proof} \rangle$

**lemma** *trivial-in-allVars*:  $\text{trivial } g\ v \implies v \in \text{allVars } g$

$\langle \text{proof} \rangle$

**declare** *phiArg-def*  $[\text{simp del}]$

**end**

## 2.5 Bundling of CFG and Equivalent SSA CFG

**locale** *CFG-SSA-Transformed-base* = *old: CFG-base*  $\alpha e \alpha n$  *invar inEdges' Entry*  
*oldDefs oldUses* + *CFG-SSA-wf-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**  
*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** +  
**fixes** *var*  $:: 'g \Rightarrow 'val \Rightarrow 'var$

**locale** *CFG-SSA-Transformed* = *CFG-SSA-Transformed-base*  $\alpha e \alpha n$  *invar in-*  
*Edges' Entry oldDefs oldUses defs uses phis var*  
+ *old: CFG-wf*  $\alpha e \alpha n$  *invar inEdges' Entry oldDefs oldUses* + *CFG-SSA-wf*  $\alpha e$   
 $\alpha n$  *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**  
*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*  $:: 'g \Rightarrow 'val \Rightarrow 'var$  +

**assumes** *oldDefs-def*:  $oldDefs\ g\ n = var\ g\ 'defs\ g\ n$   
**assumes** *oldUses-def*:  $n \in set\ (\alpha n\ g) \Longrightarrow oldUses\ g\ n = var\ g\ 'uses\ g\ n$   
**assumes** *conventional*:

$\llbracket g \vdash n-ns \rightarrow m; n \notin set\ (tl\ ns); v \in allDefs\ g\ n; v \in allUses\ g\ m; x \in set\ (tl\ ns); v' \in allDefs\ g\ x \rrbracket \Longrightarrow var\ g\ v' \neq var\ g\ v$

**assumes** *phis-same-var[elim]*:  $phis\ g\ (n, v) = Some\ vs \Longrightarrow v' \in set\ vs \Longrightarrow var\ g\ v' = var\ g\ v$

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

**begin**

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

*<proof>*

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

$\langle proof \rangle$

**lemma** *phiArg-same-var*:  $phiArg\ g\ p\ q \implies var\ g\ q = var\ g\ p$   
 $\langle proof \rangle$

**lemma** *oldDef-defAss*:  
 **assumes**  $v \in allUses\ g\ n\ g \vdash Entry\ g\ -ns \rightarrow n$   
 **obtains**  $m$  **where**  $m \in set\ ns\ var\ g\ v \in oldDefs\ g\ m$   
 $\langle proof \rangle$

**lemma** *allDef-path-from-simpleDef*:  
 **assumes** $[simp]$ :  $v \in allVars\ g$   
 **obtains**  $n\ ns$  **where**  $g \vdash n - ns \rightarrow defNode\ g\ v\ old.EntryPath\ g\ ns\ var\ g\ v \in oldDefs\ g\ n$   
 $\langle proof \rangle$

**lemma** *defNode-var-disjoint*:  
 **assumes**  $p \in allVars\ g\ q \in allVars\ g\ p \neq q\ defNode\ g\ p = defNode\ g\ q$   
 **shows**  $var\ g\ p \neq var\ g\ q$   
 $\langle proof \rangle$

**lemma** *phiArg-distinct-nodes*:  
 **assumes**  $phiArg\ g\ p\ q\ p \neq q$  **and** $[simp]$ :  $p \in allVars\ g$   
 **shows**  $defNode\ g\ p \neq defNode\ g\ q$   
 $\langle proof \rangle$

**lemma** *phiArgs-def-distinct*:  
 **assumes**  $phiArg\ g\ p\ q\ phiArg\ g\ p\ r\ q \neq r\ p \in allVars\ g$   
 **shows**  $defNode\ g\ q \neq defNode\ g\ r$   
 $\langle proof \rangle$

**lemma** *defNode-not-on-defUse-path*:  
 **assumes**  $p: g \vdash defNode\ g\ p - ns \rightarrow n\ defNode\ g\ p \notin set\ (tl\ ns)\ p \in allUses\ g\ n$   
 **assumes** $[simp]$ :  $q \in allVars\ g\ p \neq q\ var\ g\ p = var\ g\ q$   
 **shows**  $defNode\ g\ q \notin set\ ns$   
 $\langle proof \rangle$

**lemma** *defUse-paths-disjoint*:  
 **assumes**  $p: g \vdash defNode\ g\ p - ns \rightarrow n\ defNode\ g\ p \notin set\ (tl\ ns)\ p \in allUses\ g\ n$   
 **assumes**  $q: g \vdash defNode\ g\ q - ms \rightarrow m\ defNode\ g\ q \notin set\ (tl\ ms)\ q \in allUses\ g\ m$   
 **assumes** $[simp]$ :  $p \neq q\ var\ g\ p = var\ g\ q$   
 **shows**  $set\ ns \cap set\ ms = \{\}$   
 $\langle proof \rangle$

**lemma** *oldDefsI*:  $v \in defs\ g\ n \implies var\ g\ v \in oldDefs\ g\ n$   $\langle proof \rangle$

**lemma** *simpleDefs-phiDefs-var-disjoint*:  
 **assumes**  $v \in phiDefs\ g\ n\ n \in set\ (\alpha n\ g)$



```

shows  $var\ g\ v \notin oldDefs\ g\ n$ 
⟨proof⟩

lemma liveVal-use-path:
assumes liveVal  $g\ v$ 
obtains  $ns\ m$  where  $g \vdash defNode\ g\ v - ns \rightarrow m$   $var\ g\ v \in oldUses\ g\ m$ 
 $\bigwedge x. x \in set\ (tl\ ns) \implies var\ g\ v \notin oldDefs\ g\ x$ 
⟨proof⟩
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  $g\ x\ xs\ y\ ys\ z \equiv g \vdash x - xs \rightarrow z \wedge g \vdash y - ys \rightarrow z \wedge length\ xs > 1 \wedge length\ ys > 1 \wedge x \neq y \wedge$ 
 $(\forall j \in \{0..<length\ xs\}. \forall k \in \{0..<length\ ys\}. xs\ !\ j = ys\ !\ k \implies j = length\ xs - 1 \vee k = length\ ys - 1)$ 

```

Simplified definition

```

definition pathsConverge'  $g\ x\ xs\ y\ ys\ z \equiv g \vdash x - xs \rightarrow z \wedge g \vdash y - ys \rightarrow z \wedge length\ xs > 1 \wedge length\ ys > 1 \wedge x \neq y \wedge$ 
 $set\ (butlast\ xs) \cap set\ (butlast\ ys) = \{\}$ 

```

```

lemma pathsConverge'[simp]:  $pathsConverge\ g\ x\ xs\ y\ ys\ z \iff pathsConverge'\ g\ x\ xs\ y\ ys\ z$ 
⟨proof⟩

```

```

lemma pathsConvergeI:
assumes  $g \vdash x - xs \rightarrow z\ g \vdash y - ys \rightarrow z\ length\ xs > 1\ length\ ys > 1\ set\ (butlast\ xs) \cap set\ (butlast\ ys) = \{\}$ 
shows  $pathsConverge\ g\ x\ xs\ y\ ys\ z$ 
⟨proof⟩
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  $g \equiv \forall n\ ns. g \vdash n - ns \rightarrow n \implies (\exists m \in set\ ns. \forall n \in set\ ns. dominates\ g\ m\ n)$ 

```

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

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

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

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

*<proof>*

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

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

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]:  $p \in allVars\ g$

**shows**  $\neg(def\ dominates\ g\ q\ p \wedge def\ dominates\ g\ r\ p)$

*<proof>*

**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>*

**lemma** *convergence-prop'*:

**assumes**  $necessaryPhi\ g\ v\ n\ g \vdash n - ns \rightarrow m\ v \in var\ g\ 'allUses\ g\ m \wedge x.\ x \in set\ ns \implies v \notin oldDefs\ g\ x$

**obtains**  $val$  **where**  $var\ g\ val = v\ phis\ g\ (n,val) \neq None$

*<proof>*

**lemma** *nontrivialE*:

**assumes**  $\neg trivial\ g\ p\ phi\ g\ p \neq None$  **and**[simp]:  $p \in allVars\ g$

**obtains**  $r\ s$  **where**  $phiArg\ g\ p\ r\ phiArg\ g\ p\ s\ distinct\ [p,\ r,\ s]$

*<proof>*

**lemma** *paths-converge-prefix*:

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

**obtains**  $xs' ys' z'$  **where**  $old.pathsConverge\ g\ x\ xs'\ y\ ys'\ z'\ prefix\ xs'\ xs\ prefix\ ys'\ ys$   
 ⟨proof⟩

**lemma** *unnecessaryPhi-disjoint-paths-aux*:  
**assumes**  $\neg unnecessaryPhi\ g\ r$  **and**[simp]:  $r \in allVars\ g$   
**obtains**  $n_1\ ns_1\ n_2\ ns_2$  **where**  
 $var\ g\ r \in oldDefs\ g\ n_1\ g \vdash n_1 - ns_1 \rightarrow defNode\ g\ r$  **and**  
 $var\ g\ r \in oldDefs\ g\ n_2\ g \vdash n_2 - ns_2 \rightarrow defNode\ g\ r$  **and**  
 $set\ (butlast\ ns_1) \cap set\ (butlast\ ns_2) = \{\}$   
 ⟨proof⟩

**lemma** *unnecessaryPhi-disjoint-paths*:  
**assumes**  $\neg unnecessaryPhi\ g\ r$   $\neg unnecessaryPhi\ g\ s$   
**and**  $rs: defNode\ g\ r \neq defNode\ g\ s$   
**and**[simp]:  $r \in allVars\ g\ s \in allVars\ g\ var\ g\ r = V\ var\ g\ s = V$   
**obtains**  $n\ ns\ m\ ms$  **where**  $V \in oldDefs\ g\ n\ g \vdash n - ns \rightarrow defNode\ g\ r$  **and**  $V \in oldDefs\ g\ m\ g \vdash m - ms \rightarrow defNode\ g\ s$   
**and**  $set\ ns \cap set\ ms = \{\}$   
 ⟨proof⟩

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⟩

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**  $cytronMinimal\ g$   
 ⟨proof⟩  
**end**

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

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

**lemma** *phiCount*:  $phiCount\ g = card\ (dom\ (phis\ g))$   
 ⟨proof⟩

**theorem** *phi-count-minimal*:  
**assumes**  $cytronMinimal\ g\ pruned\ g$   
**assumes**  $CFG-SSA-Transformed\ \alpha e\ \alpha n\ invar\ inEdges'\ Entry\ oldDefs\ oldUses\ defs'\ uses'\ phis'\ var'$

```

    shows card (dom (phis g)) ≤ card (dom (phis' g))
  ⟨proof⟩
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 × 'node × Def

instantiation Def :: linorder
begin
  definition x < y ↔ x = SimpleDef ∧ y = PhiDef
  definition less-eq-Def (x :: Def) y ↔ x = y ∨ x < y
  instance ⟨proof⟩
end

locale CFG-Construct = CFG 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
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**  $defNode :: ('node, 'var) ssaVal \Rightarrow 'node$  **where**  $defNode v \equiv fst (snd v)$

**abbreviation**  $defKind :: ('node, 'var) ssaVal \Rightarrow Def$  **where**  $defKind v \equiv snd (snd v)$

**declare**  $var-def[simp]$

**function**  $lookupDef :: 'g \Rightarrow 'node \Rightarrow 'var \Rightarrow ('node, 'var) ssaVal$  **where**

$lookupDef g n v =$   
   if  $n \notin set (\alpha n g)$  then  $undefined$   
   else if  $v \in defs g n$  then  $(v, n, SimpleDef)$   
   else case  $predecessors g n$  of  
      $[m] \Rightarrow lookupDef g m v$   
      $| - \Rightarrow (v, n, PhiDef)$   
 )

$\langle proof \rangle$

**termination**  $\langle proof \rangle$

**declare**  $lookupDef.simps [code]$

**definition**  $defs' :: 'g \Rightarrow 'node \Rightarrow ('node, 'var) ssaVal set$  **where**

$defs' g n \equiv (\lambda v. (v, n, SimpleDef)) \text{ ` } defs g n$

**definition**  $uses' :: 'g \Rightarrow 'node \Rightarrow ('node, 'var) ssaVal set$  **where**

$uses' g n \equiv lookupDef g n \text{ ` } uses g n$

**definition**  $phis' :: 'g \Rightarrow ('node, ('node, 'var) ssaVal) phis$  **where**

$phis' \equiv \lambda g (n, (v, m, def)).$

  if  $m = n \wedge n \in phiDefNodes g v \wedge v \in vars g \wedge def = PhiDef$  then

    Some  $[lookupDef g m v . m \leftarrow predecessors g n]$

  else None

**declare**  $uses'-def [code]$   $defs'-def [code]$   $phis'-def [code]$

**abbreviation**  $lookupDefNode g n v \equiv defNode (lookupDef g n v)$

**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  $\{n\}$ )

$n \in set unvisited \vee n \notin defs g n$

$\langle proof \rangle$

**lemma**  $phiDefNode-aux-is-join-node:$

**assumes**  $n \in phiDefNodes-aux g v un m$

**shows** length (predecessors g n)  $\neq 1$

$\langle proof \rangle$

**lemma** *phiDefNode-is-join-node*:  
**assumes**  $n \in \text{phiDefNodes } g \ v$   
**shows**  $\text{length } (\text{predecessors } g \ n) \neq 1$   
*<proof>*

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

**lemma** *unvisitedPath-removeLast*:  
**assumes**  $\text{unvisitedPath } un \ ns \ \text{length } ns \geq 2$   
**shows**  $\text{unvisitedPath } (\text{removeAll } (\text{last } ns) \ un) \ (\text{butlast } ns)$   
*<proof>*

**lemma** *phiDefNodes-auxI*:  
**assumes**  $g \vdash n - ns \rightarrow m \ \text{unvisitedPath } un \ ns \ \forall n \in \text{set } ns. v \notin \text{defs } g \ n \ \text{length } (\text{predecessors } g \ n) \neq 1$   
**shows**  $n \in \text{phiDefNodes-aux } g \ v \ un \ m$   
*<proof>*

**lemma** *phiDefNodes-auxE*:  
**assumes**  $n \in \text{phiDefNodes-aux } g \ v \ un \ m \ m \in \text{set } (\alpha n \ g)$   
**obtains**  $ns$  **where**  $g \vdash n - ns \rightarrow m \ \forall n \in \text{set } ns. v \notin \text{defs } g \ n \ \text{length } (\text{predecessors } g \ n) \neq 1 \ \text{unvisitedPath } un \ ns$   
*<proof>*

**lemma** *phiDefNodesE*:  
**assumes**  $n \in \text{phiDefNodes } g \ v$   
**obtains**  $ns \ m$  **where**  $g \vdash n - ns \rightarrow m \ \forall n \in \text{set } ns. v \notin \text{defs } g \ n \ v \in \text{uses } g \ m$   
*<proof>*

**lemma** *phiDefNodes- $\alpha n$ [simp]*:  $n \in \text{phiDefNodes } g \ v \implies n \in \text{set } (\alpha n \ g)$   
*<proof>*

**lemma** *phiDefNodesI*:  
**assumes**  $g \vdash n - ns \rightarrow m \ v \in \text{uses } g \ m \ \forall n \in \text{set } ns. v \notin \text{defs } g \ n \ \text{length } (\text{predecessors } g \ n) \neq 1$   
**shows**  $n \in \text{phiDefNodes } g \ v$   
*<proof>*

**lemma** *lookupDef-cases[consumes 1]*:  
**assumes**  $n \in \text{set } (\alpha n \ g)$   
**obtains**  $(\text{SimpleDef}) \ v \in \text{defs } g \ n \ \text{lookupDef } g \ n \ v = (v, n, \text{SimpleDef})$   
 $\quad \mid (\text{PhiDef}) \quad v \notin \text{defs } g \ n \ \text{length } (\text{predecessors } g \ n) \neq 1 \ \text{lookupDef } g \ n \ v$   
 $= (v, n, \text{PhiDef})$   
 $\quad \mid (\text{rec}) \ m \ \text{where } v \notin \text{defs } g \ n \ \text{predecessors } g \ n = [m] \ m \in \text{set } (\alpha n \ g)$   
 $\text{lookupDef } g \ n \ v = \text{lookupDef } g \ m \ v$   
*<proof>*

**lemma** *lookupDef-cases'[consumes 1]*:

**assumes**  $n \in \text{set } (\alpha n g)$   
**obtains**  $(\text{SimpleDef}) v \in \text{defs } g n \text{ defNode } (\text{lookupDef } g n v) = n \text{ defKind}$   
 $(\text{lookupDef } g n v) = \text{SimpleDef}$   
 $| (\text{PhiDef}) v \notin \text{defs } g n \text{ length } (\text{predecessors } g n) \neq 1 \text{ lookupDefNode } g$   
 $n v = n \text{ defKind } (\text{lookupDef } g n v) = \text{PhiDef}$   
 $| (\text{rec}) m \textbf{ where } v \notin \text{defs } g n \text{ predecessors } g n = [m] m \in \text{set } (\alpha n g)$   
 $\text{lookupDef } g n v = \text{lookupDef } g m v$   
 $\langle \text{proof} \rangle$

**lemma lookupDefE:**  
**assumes**  $\text{lookupDef } g n v = v' n \in \text{set } (\alpha n g)$   
**obtains**  $(\text{SimpleDef}) v \in \text{defs } g n v' = (v, n, \text{SimpleDef})$   
 $| (\text{PhiDef}) v \notin \text{defs } g n \text{ length } (\text{predecessors } g n) \neq 1 v' = (v, n, \text{PhiDef})$   
 $| (\text{rec}) m \textbf{ where } v \notin \text{defs } g n \text{ predecessors } g n = [m] m \in \text{set } (\alpha n g) v' =$   
 $\text{lookupDef } g m v$   
 $\langle \text{proof} \rangle$

**lemma lookupDef-induct[consumes 1, case-names SimpleDef PhiDef rec]:**  
**assumes**  $n \in \text{set } (\alpha n g)$   
 $\bigwedge n. \llbracket n \in \text{set } (\alpha n g); v \in \text{defs } g n; \text{lookupDef } g n v = (v, n, \text{SimpleDef}) \rrbracket$   
 $\implies P n$   
 $\bigwedge n. \llbracket n \in \text{set } (\alpha n g); v \notin \text{defs } g n; \text{length } (\text{predecessors } g n) \neq 1;$   
 $\text{lookupDef } g n v = (v, n, \text{PhiDef}) \rrbracket \implies P n$   
 $\bigwedge n m. \llbracket v \notin \text{defs } g n; \text{predecessors } g n = [m]; m \in \text{set } (\alpha n g); \text{lookupDef}$   
 $g n v = \text{lookupDef } g m v; P m \rrbracket \implies P n$   
**shows**  $P n$   
 $\langle \text{proof} \rangle$

**lemma lookupDef-induct'[consumes 2, case-names SimpleDef PhiDef rec]:**  
**assumes**  $n \in \text{set } (\alpha n g) \text{ lookupDef } g n v = (v, n', \text{def})$   
 $\llbracket v \in \text{defs } g n'; \text{def} = \text{SimpleDef} \rrbracket \implies P n'$   
 $\llbracket v \notin \text{defs } g n'; \text{length } (\text{predecessors } g n') \neq 1; \text{def} = \text{PhiDef} \rrbracket \implies P n'$   
 $\bigwedge n m. \llbracket v \notin \text{defs } g n; \text{predecessors } g n = [m]; m \in \text{set } (\alpha n g); \text{lookupDef}$   
 $g n v = \text{lookupDef } g m v; P m \rrbracket \implies P n$   
**shows**  $P n$   
 $\langle \text{proof} \rangle$

**lemma lookupDef-looksup[simp]:**  
**assumes**  $\text{lookupDef } g n v = (v', n', \text{def}) n \in \text{set } (\alpha n g)$   
**shows**  $v' = v$   
 $\langle \text{proof} \rangle$

**lemma lookupDef-looksup':**  
**assumes**  $(v', n', \text{def}) = \text{lookupDef } g n v n \in \text{set } (\alpha n g)$   
**shows**  $v' = v$   
 $\langle \text{proof} \rangle$

**lemma lookupDef-looksup'':**  
**assumes**  $n \in \text{set } (\alpha n g)$

**obtains**  $n'$  **def where**  $lookupDef\ g\ n\ v = (v, n', def)$   
 ⟨proof⟩

**lemma**  $lookupDef\ fst[simp]$ :  $n \in set\ (\alpha n\ g) \implies fst\ (lookupDef\ g\ n\ v) = v$   
 ⟨proof⟩

**lemma**  $lookupDef\ to\ \alpha n$ :  
**assumes**  $lookupDef\ g\ n\ v = (v', n', def)\ n \in set\ (\alpha n\ g)$   
**shows**  $n' \in set\ (\alpha n\ g)$   
 ⟨proof⟩

**lemma**  $lookupDef\ to\ \alpha n'[simp]$ :  
**assumes**  $lookupDef\ g\ n\ v = val\ n \in set\ (\alpha n\ g)$   
**shows**  $defNode\ val \in set\ (\alpha n\ g)$   
 ⟨proof⟩

**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**  $P\ n$   
 ⟨proof⟩

**lemma**  $defs'\ finite$ :  $finite\ (defs'\ g\ n)$   
 ⟨proof⟩

**lemma**  $uses'\ finite$ :  $finite\ (uses'\ g\ n)$   
 ⟨proof⟩

**lemma**  $defs'\ uses'\ disjoint$ :  $n \in set\ (\alpha n\ g) \implies defs'\ g\ n \cap uses'\ g\ n = \{\}$   
 ⟨proof⟩

**lemma**  $allDefs'\ disjoint$ :  $n \in set\ (\alpha n\ g) \implies m \in set\ (\alpha n\ g) \implies n \neq m$   
 $\implies (defs'\ g\ n \cup \{v. (n, v) \in dom\ (phis'\ g)\}) \cap (defs'\ g\ m \cup \{v. (m, v) \in dom\ (phis'\ g)\}) = \{\}$   
 ⟨proof⟩

**lemma**  $phiDefNodes\ aux\ finite$ :  $finite\ (phiDefNodes\ aux\ g\ v\ un\ m)$   
 ⟨proof⟩

**lemma**  $phis'\ finite$ :  $finite\ (dom\ (phis'\ g))$   
 ⟨proof⟩

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



```

⟨proof⟩

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

lemma oldDefs-correct:  $\text{defs} \ g \ n = \text{var} \ g \ ' \ \text{defs}' \ g \ n$ 
⟨proof⟩

lemma oldUses-correct:  $n \in \text{set } (\alpha n \ g) \implies \text{uses} \ g \ n = \text{var} \ g \ ' \ \text{uses}' \ g \ n$ 
⟨proof⟩

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

sublocale braun-ssa: CFG-SSA  $\alpha e \ \alpha n \ \text{invar} \ \text{inEdges}' \ \text{Entry} \ \text{defs}' \ \text{uses}' \ \text{phis}'$ 
⟨proof⟩
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  $g \vdash n - ns \rightarrow n \ \text{distinct} \ ns$ 
    shows  $ns = [n]$ 
  ⟨proof⟩

  lemma lookupDef-path:
    assumes  $m \in \text{set } (\alpha n \ g)$ 
    obtains  $ns$  where  $g \vdash \text{lookupDefNode} \ g \ m \ v - ns \rightarrow m \ (\forall x \in \text{set} \ (tl \ ns). v \notin \text{defs} \ g \ x)$ 
  ⟨proof⟩

  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  $\text{var} \ g \ v' \neq v$ 
  ⟨proof⟩

  lemma allUse-lookupDef:
    assumes  $v \in \text{braun-ssa.allUses} \ g \ m \ m \in \text{set} \ (\alpha n \ g)$ 
    shows  $\text{lookupDef} \ g \ m \ (\text{var} \ g \ v) = v$ 
  ⟨proof⟩

  lemma phis'-fst:
    assumes  $\text{phis}' \ g \ (n, v) = \text{Some} \ vs \ v' \in \text{set} \ vs$ 

```

**shows**  $var\ g\ v' = var\ g\ v$   
 ⟨proof⟩

**lemma** *allUse-simpleUse*:

**assumes**  $v \in braun-ssa.allUses\ g\ m\ m \in set\ (\alpha n\ g)$

**obtains**  $ms\ m'$  **where**  $g \vdash m - ms \rightarrow m'$   $var\ g\ v \in uses\ g\ m' \forall x \in set\ (tl\ ms).$   
 $var\ g\ v \notin defs\ g\ x$

⟨proof⟩

**lemma** *defs'*:  $v \in defs'\ g\ n \iff var\ g\ v \in defs\ g\ n \wedge defKind\ v = SimpleDef \wedge defNode\ v = n$

⟨proof⟩

**lemma** *use-implies-allDef*:

**assumes**  $lookupDef\ g\ m\ (var\ g\ v) = v\ m \in set\ (\alpha n\ g)\ var\ g\ v \in uses\ g\ m' g \vdash m - ms \rightarrow m' \forall x \in set\ (tl\ ms).$   $var\ g\ v \notin defs\ g\ x$

**shows**  $v \in braun-ssa.allDefs\ g\ (defNode\ v)$

⟨proof⟩

**lemma** *allUse-defNode-in- $\alpha n$ [simp]*:

**assumes**  $v \in braun-ssa.allUses\ g\ m\ m \in set\ (\alpha n\ g)$

**shows**  $defNode\ v \in set\ (\alpha n\ g)$

⟨proof⟩

**lemma** *allUse-implies-allDef*:

**assumes**  $v \in braun-ssa.allUses\ g\ m\ m \in set\ (\alpha n\ g)$

**shows**  $v \in braun-ssa.allDefs\ g\ (defNode\ v)$

⟨proof⟩

**lemma** *conventional*:

**assumes**  $g \vdash n - ns \rightarrow m\ n \notin set\ (tl\ ns)\ v \in braun-ssa.allDefs\ g\ n\ v \in braun-ssa.allUses\ g\ m$

$x \in set\ (tl\ ns)\ v' \in braun-ssa.allDefs\ g\ x$

**shows**  $var\ g\ v' \neq var\ g\ v$

⟨proof⟩

**lemma** *allDefs-var-disjoint-aux*:  $n \in set\ (\alpha n\ g) \implies v \in defs\ g\ n \implies n \notin phiDefNodes\ g\ v$

⟨proof⟩

**lemma** *allDefs-var-disjoint*:  $\llbracket n \in set\ (\alpha n\ g); v \in braun-ssa.allDefs\ g\ n; v' \in braun-ssa.allDefs\ g\ n; v \neq v' \rrbracket \implies var\ g\ v' \neq var\ g\ v$

⟨proof⟩

**lemma**[*simp*]:  $n \in set\ (\alpha n\ g) \implies v \in defs\ g\ n \implies lookupDefNode\ g\ n\ v = n$

⟨proof⟩

**lemma**[*simp*]:  $n \in set\ (\alpha n\ g) \implies length\ (predecessors\ g\ n) \neq 1 \implies lookupDefNode\ g\ n\ v = n$

```

⟨proof⟩

lemma lookupDef-idem[simp]:
  assumes  $n \in \text{set } (\alpha n g)$ 
  shows  $\text{lookupDef } g (\text{lookupDefNode } g n v) v = \text{lookupDef } g n v$ 
  ⟨proof⟩
end

locale CFG-Construct-wf = CFG-Construct  $\alpha e \alpha n \text{invar inEdges' Entry defs uses}$ 
+ CFG-wf  $\alpha e \alpha n \text{invar inEdges' Entry defs uses}$ 
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⟩

  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⟩

  lemma Empty-no-phis:
    shows  $\text{phis}' g (\text{Entry } g, v) = \text{None}$ 
    ⟨proof⟩

  lemma braun-ssa-CFG-SSA-wf:
     $\text{CFG-SSA-wf } \alpha e \alpha n \text{invar inEdges' Entry defs' uses' phis}'$ 
    ⟨proof⟩

  sublocale braun-ssa: CFG-SSA-wf  $\alpha e \alpha n \text{invar inEdges' Entry defs' uses' phis}'$ 
  ⟨proof⟩

  lemma braun-ssa-CFG-SSA-Transformed:
     $\text{CFG-SSA-Transformed } \alpha e \alpha n \text{invar inEdges' Entry defs uses defs' uses' phis}'$ 
  var
  ⟨proof⟩

  sublocale braun-ssa: CFG-SSA-Transformed  $\alpha e \alpha n \text{invar inEdges' Entry defs}$ 
  uses defs' uses' phis' var
  ⟨proof⟩

```

**lemma** *PhiDef-defNode-eq*:  
**assumes**  $n \in \text{set } (\alpha n \ g) \ n \in \text{phiDefNodes } g \ v \ v \in \text{vars } g$   
**shows**  $\text{braun-ssa.defNode } g \ (v, n, \text{PhiDef}) = n$   
 $\langle \text{proof} \rangle$

**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**  $\text{braun-ssa.liveVal } g \ (v, n, \text{PhiDef})$   
 $\langle \text{proof} \rangle$

**lemma** *phiDefNodes-aux-pruned*:  
**assumes**  $m \in \text{phiDefNodes-aux } g \ v \ (\alpha n \ g) \ n \ n \in \text{set } (\alpha n \ g) \ v \in \text{uses } g \ n$   
**shows**  $\text{braun-ssa.liveVal } g \ (v, m, \text{PhiDef})$   
 $\langle \text{proof} \rangle$

**theorem** *phis'-pruned*:  $\text{braun-ssa.pruned } g$   
 $\langle \text{proof} \rangle$

**declare** *var-def* [*simp del*]

**declare** *no-disjoint-cycle* [*simp del*]  
**declare** *lookupDef-lookup* [*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*  $\alpha e \ \alpha n$   
 $\text{invar } \text{inEdges}' \ \text{Entry} \ \text{oldDefs} \ \text{oldUses} \ \text{defs} \ \text{uses} \ \text{phis} \ \text{var}$

**for**

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

```

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 ≡ case x of (n,v) ⇒ 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 αe αn invar in-
Edges' Entry oldDefs oldUses defs uses phis var
+ CFG-SSA-Transformed-notriv-base αe αn invar inEdges' Entry oldDefs oldUses
defs uses phis var chooseNext-all
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 and
  phis :: 'g ⇒ ('node, 'val) phis and
  var :: 'g ⇒ 'val ⇒ 'var and
  chooseNext-all :: ('node ⇒ 'val set) ⇒ ('node, 'val) phis ⇒ 'g ⇒ ('node × 'val)
+
assumes chooseNext-all: CFG-SSA-Transformed αe αn invar inEdges' Entry old-
Defs oldUses defs u p var ⇒
  CFG-SSA-wf-base.redundant αn inEdges' defs u p g ⇒
  chooseNext-all (u g) (p g) g ∈ dom (p g) ∧
  CFG-SSA-wf-base.trivial αn inEdges' defs u p g (snd (chooseNext-all (u g) (p g)
g))
begin
  lemma chooseNext':redundant g ⇒ chooseNext' g ∈ dom (phis g) ∧ trivial g
(chooseNext g)
  ⟨proof⟩

```

**lemma** *chooseNext*: *redundant g*  $\implies$  *chooseNext g*  $\in$  *allVars g*  $\wedge$  *trivial g*  
 (*chooseNext g*)  
 $\langle$ *proof* $\rangle$

**lemmas** *chooseNext-in-allVars*[*simp*] = *chooseNext*[*THEN conjunct1*]

**lemma** *isTrivialPhi-det*: *trivial g v*  $\implies$   $\exists !v'$ . *isTrivialPhi g v v'*  
 $\langle$ *proof* $\rangle$

**lemma** *trivialPhi-strict-dom*:  
**assumes**[*simp*]: *v*  $\in$  *allVars g* **and** *triv*: *isTrivialPhi g v v'*  
**shows** *strict-def-dom g v' v*  
 $\langle$ *proof* $\rangle$

**lemma** *isTrivialPhi-asymmetric*:  
**assumes** *isTrivialPhi g a b*  
**and** *isTrivialPhi g b a*  
**shows** *False*  
 $\langle$ *proof* $\rangle$

**lemma** *substitution*[*intro*]: *redundant g*  $\implies$  *isTrivialPhi g* (*chooseNext g*) (*substitution g*)  
 $\langle$ *proof* $\rangle$

**lemma** *trivialPhi-in-allVars*[*simp*]:  
**assumes** *isTrivialPhi g v v'* **and**[*simp*]: *v*  $\in$  *allVars g*  
**shows** *v'*  $\in$  *allVars g*  
 $\langle$ *proof* $\rangle$

**lemma** *substitution-in-allVars*[*simp*]:  
**assumes** *redundant g*  
**shows** *substitution g*  $\in$  *allVars g*  
 $\langle$ *proof* $\rangle$

**lemma** *defs-uses-disjoint-inv*:  
**assumes**[*simp*]: *n*  $\in$  *set* ( $\alpha n$  *g*) *redundant g*  
**shows** *defs g n*  $\cap$  *uses' g n* =  $\{\}$   
 $\langle$ *proof* $\rangle$

**end**

**context** *CFG-SSA-wf*

**begin**

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

**for** *g* :: *'g*

**where**

*liveSimple'*:  $\llbracket n \in \text{set } (\alpha n \text{ } g); \text{val} \in \text{uses } g \text{ } n \rrbracket \implies \text{liveVal}' \text{ } g \text{ } [\text{val}]$   
 | *livePhi'*:  $\llbracket \text{liveVal}' \text{ } g \text{ } (v \# vs); \text{phiArg } g \text{ } v \text{ } v' \rrbracket \implies \text{liveVal}' \text{ } g \text{ } (v' \# v \# vs)$

```

lemma liveVal'-suffix:
  assumes liveVal' g vs suffix vs' vs vs' ≠ []
  shows liveVal' g vs'
  ⟨proof⟩

lemma liveVal'I:
  assumes liveVal g v
  obtains vs where liveVal' g (v#vs)
  ⟨proof⟩

lemma liveVal'D:
  assumes liveVal' g vs vs = v#vs'
  shows liveVal g v
  ⟨proof⟩
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
   $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)$  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  $\Rightarrow 'g \Rightarrow ('node \times 'val)$ 
and
   $g :: 'g +$ 
assumes redundant[simp]: redundant g
begin
  abbreviation  $u\text{-}g \equiv uses(g:=uses' g)$ 
  abbreviation  $p\text{-}g \equiv phis(g:=phis' g)$ 

  sublocale step: CFG-SSA-Transformed-notriv-base  $\alpha e \alpha n$  invar inEdges' Entry
oldDefs oldUses defs u-g p-g var chooseNext-all ⟨proof⟩

lemma simpleDefs-phiDefs-disjoint-inv:
  assumes  $n \in set (\alpha n g)$ 
  shows  $defs g n \cap step.phiDefs g n = \{\}$ 
  ⟨proof⟩

lemma allDefs-disjoint-inv:
  assumes  $n \in set (\alpha n g)$   $m \in set (\alpha n g)$   $n \neq m$ 
  shows  $step.allDefs g n \cap step.allDefs g m = \{\}$ 

```

*<proof>*

**lemma** *phis-finite-inv*:  
  **shows** *finite (dom (phis' g))*  
*<proof>*

**lemma** *phis-wf-inv*:  
  **assumes** *phis' g (n, v) = Some args*  
  **shows** *length (old.predecessors g n) = length args*  
*<proof>*

**sublocale** *step: CFG-SSA  $\alpha e \alpha n$  invar inEdges' Entry defs u-g p-g*  
*<proof>*

**lemma** *allUses-narrows*:  
  **assumes** *n  $\in$  set ( $\alpha n$  g)*  
  **shows** *step.allUses g n  $\subseteq$  substNext g ' allUses g n*  
*<proof>*

**lemma** *allDefs-narrows[simp]*: *v  $\in$  step.allDefs g n  $\implies$  v  $\in$  allDefs g n*  
*<proof>*

**lemma** *allUses-def-ass-inv*:  
  **assumes** *v'  $\in$  step.allUses g n n  $\in$  set ( $\alpha n$  g)*  
  **shows** *step.defAss g n v'*  
*<proof>*

**lemma** *Entry-no-phis-inv*: *phis' g (Entry g, v) = None*  
*<proof>*

**sublocale** *step: CFG-SSA-wf  $\alpha e \alpha n$  invar inEdges' Entry defs u-g p-g*  
*<proof>*

**lemma** *chooseNext-eliminated*: *chooseNext g  $\notin$  step.allDefs g (defNode g (chooseNext g))*  
*<proof>*

**lemma** *oldUses-inv*:  
  **assumes** *n  $\in$  set ( $\alpha n$  g)*  
  **shows** *oldUses g n = var g ' u-g g n*  
*<proof>*

**lemma** *conventional-inv*:  
  **assumes** *g  $\vdash$  n  $\rightarrow$  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>*



**lemma**<sub>[simp]</sub>:  $\text{var } g \text{ (substNext } g \text{ } v) = \text{var } g \text{ } v$   
 ⟨proof⟩

**lemma** *phis-same-var-inv*:  
**assumes**  $\text{phis}' g \text{ (} n, v) = \text{Some } vs \text{ } v' \in \text{set } vs$   
**shows**  $\text{var } g \text{ } v' = \text{var } g \text{ } v$   
 ⟨proof⟩

**lemma** *allDefs-var-disjoint-inv*:  $\llbracket n \in \text{set } (\alpha n \text{ } g); v \in \text{step.allDefs } g \text{ } n; v' \in \text{step.allDefs } g \text{ } n; v \neq v' \rrbracket \implies \text{var } g \text{ } v' \neq \text{var } g \text{ } v$   
 ⟨proof⟩

**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*  
 ⟨proof⟩

**sublocale** *step*: *CFG-SSA-Transformed-notriv*  $\alpha e \alpha n$  *invar inEdges' Entry old-Defs oldUses defs u-g p-g var chooseNext-all*  
 ⟨proof⟩

**lemma** *step-defNode*:  $v \in \text{allVars } g \implies v \neq \text{chooseNext } g \implies \text{step.defNode } g \text{ } v = \text{defNode } g \text{ } v$   
 ⟨proof⟩

**lemma** *step-phi*:  $v \in \text{allVars } g \implies v \neq \text{chooseNext } g \implies \text{step.phi } g \text{ } v = \text{map-option (map (substNext } g)) (\text{phi } g \text{ } v)$   
 ⟨proof⟩

**lemma** *liveVal'-inv*:  
**assumes**  $\text{liveVal}' g \text{ (} v \# vs) v \neq \text{chooseNext } g$   
**obtains**  $vs'$  **where**  $\text{step.liveVal}' g \text{ (} v \# vs')$   
 ⟨proof⟩

**lemma** *liveVal-inv*:  
**assumes**  $\text{liveVal } g \text{ } v v \neq \text{chooseNext } g$   
**shows**  $\text{step.liveVal } g \text{ } v$   
 ⟨proof⟩

**lemma** *pruned-inv*:  
**assumes** *pruned*  $g$   
**shows**  $\text{step.pruned } g$   
 ⟨proof⟩

**end**

**context** *CFG-SSA-Transformed-notriv-base*  
**begin**  
**abbreviation**  $\text{inst } g \text{ } u \text{ } p \equiv \text{CFG-SSA-Transformed-notriv } \alpha e \alpha n \text{ invar inEdges' Entry oldDefs oldUses defs (uses(g:=u)) (phis(g:=p)) var chooseNext-all}$   
**abbreviation**  $\text{inst}' g \equiv \lambda(u,p). \text{inst } g \text{ } u \text{ } p$

**interpretation** *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  
 <proof>

**definition** *cond*  $g \equiv \lambda(u,p). \text{uninst.redundant } (\text{uses}(g:=u)) (\text{phis}(g:=p)) g$

**definition** *step*  $g \equiv \lambda(u,p). (\text{uninst.uses}' (\text{uses}(g:=u)) (\text{phis}(g:=p)) g,$   
 $\text{uninst.phis}' (\text{uses}(g:=u)) (\text{phis}(g:=p)) g)$

**definition**[code]: *substAll*  $g \equiv \text{while } (\text{cond } g) (\text{step } g) (\text{uses } g, \text{phis } g)$

**definition**[code]: *uses'-all*  $g \equiv \text{fst } (\text{substAll } g)$

**definition**[code]: *phis'-all*  $g \equiv \text{snd } (\text{substAll } g)$

**lemma** *uninst-allVars-simps* [simp]:

$\text{uninst.allVars } u (\lambda-. p g) g = \text{uninst.allVars } u p g$   
 $\text{uninst.allVars } (\lambda-. u g) p g = \text{uninst.allVars } u p g$   
 $\text{uninst.allVars } (\text{uses}(g:=u g)) p g = \text{uninst.allVars } u p g$   
 $\text{uninst.allVars } u (\text{phis}(g:=p g)) g = \text{uninst.allVars } u p g$   
 <proof>

**lemma** *uninst-trivial-simps* [simp]:

$\text{uninst.trivial } u (\lambda-. p g) g = \text{uninst.trivial } u p g$   
 $\text{uninst.trivial } (\lambda-. u g) p g = \text{uninst.trivial } u p g$   
 $\text{uninst.trivial } (\text{uses}(g:=u g)) p g = \text{uninst.trivial } u p g$   
 $\text{uninst.trivial } u (\text{phis}(g:=p g)) g = \text{uninst.trivial } u p g$   
 <proof>

**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**  $\text{card } (\text{dom } (\text{phis}' g)) < \text{card } (\text{dom } (\text{phis } g))$   
 <proof>

**lemma** *step-preserves-inst*:

**assumes** *inst'*  $g (u,p)$   
**and** CFG-SSA-wf-base.*redundant*  $\alpha n$  inEdges' defs (*uses*( $g:=u$ )) (*phis*( $g:=p$ ))

$g$

**shows** *inst'*  $g (\text{step } g (u,p))$

<proof>

**lemma** *substAll*:

**assumes**  $P (\text{uses } g, \text{phis } g)$

**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>

**sublocale** *notriv*: *CFG-SSA-Transformed*  $\alpha e \alpha n$  *invar inEdges' Entry oldDefs*  
*oldUses defs uses'-all phis'-all*  
 <proof>

**theorem** *not-redundant*:  $\neg notriv.redundant g$   
 <proof>

**corollary** *minimal*:  $old.reducible g \implies notriv.cytronMinimal g$   
 <proof>

**theorem** *pruned-invariant*:  
**assumes** *pruned g*  
**shows** *notriv.pruned g*  
 <proof>

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**

$step g m s v \equiv \text{if } v \in oldDefs g m \text{ then } Some m \text{ else } s v$

**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**

$ssaStep g m i s v \equiv$

```

if v ∈ defs g m then
  Some m
else
  casephis g (m,v) of
    Some phiParams ⇒ s (phiParams ! i)
  | None ⇒ s v

```

**inductive** *ssaBS* :: 'g ⇒ 'node list ⇒ ('node, 'val) state ⇒ bool (- ⊢ -↓<sub>s</sub> - [50, 50, 50] 50)

```

for
  g :: 'g
where
  empty: g ⊢ [Entry g]↓s(ssaStep g (Entry g) 0 Map.empty)
  | snoc: [[g ⊢ ns↓ss; last ns = old.predecessors g m ! i; m ∈ set (αn g); i < length
(old.predecessors g m)]] ⇒
    g ⊢ (ns@[m])↓s(ssaStep g m i s)

```

**lemma** *ssaBS-I*:

```

assumes g ⊢ Entry g - ns → n
obtains s where g ⊢ ns↓ss
⟨proof⟩

```

**lemma** *ssaBS-nonempty[simp]*: ¬ (g ⊢ []↓<sub>s</sub>s)  
⟨proof⟩

**lemma** *ssaBS-hd[simp]*: g ⊢ ns↓<sub>s</sub>s ⇒ hd ns = Entry g  
⟨proof⟩

**lemma** *equiv-aux*:

```

assumes g ⊢ ns↓s g ⊢ ns↓ss' g ⊢ last ns - ms → m v ∈ allUses g m ∀ n ∈ set
(tl ms). var g v ∉ var g ' allDefs g n
shows s (var g v) = s' v
⟨proof⟩

```

**theorem** *equiv*:

```

assumes g ⊢ ns↓s g ⊢ ns↓ss' v ∈ uses g (last ns)
shows s (var g v) = s' v
⟨proof⟩

```

**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
  <proof>

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
  <proof>

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>

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 = ( $\bigcup x \in A. f x$ )
  <proof>

type-synonym ('node, 'val)phis-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  $\equiv \text{fold } (\lambda v. \text{Mapping.map-default } v \ \{ \} \ (\text{insert } n)) \ (\text{sorted-list-of-set } (\text{uses } g \ n))$ 
  definition addN' g n = fold ( $\lambda v m. m(v \mapsto \text{case-option } \{n\} \ (\text{insert } n) \ (m \ v))$ )
  (sorted-list-of-set (uses g n))

```

**lemma** *addN-transfer* [*transfer-rule*]:  
 $rel\text{-}fun (=) (rel\text{-}fun (=) (rel\text{-}fun (pcr\text{-}mapping (=) (=)) (pcr\text{-}mapping (=) (=))))$   
*addN' addN*  
 ⟨*proof*⟩

**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*⟩  
**end**

**context** *CFG* **begin**

**lift-definition** *useNodes-of'* :: '*g*  $\Rightarrow$  ('*var*, '*node set*) *mapping*

**is**  $\lambda g\ v. if (\exists n \in set (\alpha n\ g). v \in uses\ g\ n) then Some \{n \in set (\alpha n\ g). v \in uses\ g\ n\} else None$  ⟨*proof*⟩

**lemma** *useNodes-of'*: *useNodes-of' = useNodes-of*  
 ⟨*proof*⟩

**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*)  
 ⟨*proof*⟩

**end**

**context** *CFG-SSA-base* **begin**

**definition** *phis-addN*

**where** *phis-addN g n* = *fold* ( $\lambda v. Mapping.map\text{-}default\ v\ \{\}\ (insert\ n)$ ) (*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*⟩

**definition** *phiNodes-of* :: 'g ⇒ ('val, ('node × '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 (∃ n ∈ dom (*phis* g). v ∈ set (the (*phis* g n))) then Some {n ∈ dom (*phis* g). v ∈ set (the (*phis* g n))} else None)  
 ⟨*proof*⟩

**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))  
 ⟨*proof*⟩

**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)  
 ⟨*proof*⟩

**lemma** *lookup-cacheD* [*dest*]: *Mapping.lookup* (*cache* g f) x = Some y ⇒ y = f x  
 ⟨*proof*⟩

**lemma** *lookup-cache-usesD*: *Mapping.lookup* (*cache* g (*uses* g)) n = Some vs ⇒ vs = *uses* g n

```

  <proof>
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
  <proof>
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 phis-transfer [simplified, transfer-rule]

  lemma phiDefs-code [code]:
  phiDefs g n = snd ' Set.filter (λ(n',v). n' = n) (Mapping.keys (phis g))
  <proof>

  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*  $v$   $vs = (\text{case } (\text{foldl } (\lambda(\text{good}, v') w. \text{if } w = v \text{ then } (\text{good}, v') \text{ else case } v' \text{ of } \text{Some } v' \Rightarrow (\text{good} \wedge w = v', \text{Some } v') \mid \text{None} \Rightarrow (\text{good}, \text{Some } w)) (\text{True}, \text{None}) vs)$   
*of* (*False*, -) ⇒ *None* | (*True*,  $v$ ) ⇒  $v$ )

**lemma** *the-trivial-Nil* [*simp*]: *the-trivial*  $x$  [] = *None*  
 ⟨*proof*⟩

**lemma** *the-trivialI*:  
**assumes** *set*  $vs \subseteq \{v, v'\}$   
**and**  $v' \neq v$   
**shows** *the-trivial*  $v$   $vs = (\text{if } \text{set } vs \subseteq \{v\} \text{ then } \text{None} \text{ else } \text{Some } v')$   
 ⟨*proof*⟩

**lemma** *the-trivial-conv*:  
**shows** *the-trivial*  $v$   $vs = (\text{if } \exists v' \in \text{set } vs. v' \neq v \wedge \text{set } vs - \{v'\} \subseteq \{v\} \text{ then } \text{Some } (\text{THE } v'. v' \in \text{set } vs \wedge v' \neq v \wedge \text{set } vs - \{v'\} \subseteq \{v\}) \text{ else } \text{None})$   
 ⟨*proof*⟩

**lemma** *the-trivial-SomeE*:  
**assumes** *the-trivial*  $v$   $vs = \text{Some } v'$   
**obtains**  $v \neq v'$  **and**  $\text{set } vs = \{v'\} \mid v \neq v'$  **and**  $\text{set } vs = \{v, v'\}$   
 ⟨*proof*⟩

**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* ◦ *uses*  $\lambda g. \text{Mapping.lookup } (\text{phis } g)$

```

for  $\alpha e$  :: 'g ⇒ ('node::linorder × 'edgeD × 'node) set
and  $\alpha n$  :: '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**

**definition** [*code*]:

*trivial-code*  $(v::'val)$   $vs = (\text{the-trivial } v$   $vs \neq \text{None})$

**definition**[*code*]: *trivial-phis*  $g = \text{Set.filter } (\lambda(n, v). \text{trivial-code } v \text{ (the } (\text{Mapping.lookup}$

(*phis g*) (*n,v*))) (*Mapping.keys (phis g)*)  
**definition** [*code*]: *redundant-code g* = (*trivial-phis g* ≠ {})  
**end**

**locale** *CFG-SSA-wf-code* = *CFG-SSA-code* α*e* α*n* *invar inEdges'* *Entry defs uses phis*

+ *CFG-SSA-wf-base-code* α*e* α*n* *invar inEdges'* *Entry defs uses phis*  
+ *CFG-SSA-wf* α*e* α*n* *invar inEdges'* *Entry defs usesOf* ∘ *uses* λ*g*. *Mapping.lookup (phis g)*

**for** α*e* :: '*g* ⇒ ('*node*::*linorder* × '*edgeD* × '*node*) *set*  
**and** α*n* :: '*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**

**lemma** *trivial-code*:

*phi g v* = *Some vs* ⇒ *trivial g v* = *trivial-code v vs*  
<*proof*>

**lemma** *trivial-phis*:

*trivial-phis g* = {(*n,v*). *Mapping.lookup (phis g) (n,v)* ≠ *None* ∧ *trivial g v*}  
<*proof*>

**lemma** *redundant-code*:

*redundant g* = *redundant-code g*  
<*proof*>

**lemma** *trivial-code-mapI*:

[[ *trivial-code v vs*; *f* ' (*set vs* - {*v*}) ≠ {*v*} ; *f v* = *v* ]] ⇒ *trivial-code v (map f vs)*  
<*proof*>

**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*})  
<*proof*>

**end**

**locale** *CFG-SSA-Transformed-code* = *ssa*: *CFG-SSA-wf-code* α*e* α*n* *invar inEdges'* *Entry defs uses phis*

+

*CFG-SSA-Transformed* α*e* α*n* *invar inEdges'* *Entry oldDefs oldUses defs usesOf*  
∘ *uses* λ*g*. *Mapping.lookup (phis g) var*

**for**

α*e* :: '*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))
⟨proof⟩

```

**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⟩

**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⟩

**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*:  $\text{phis-measure } g v (\text{Mapping.update } k a p) \leq \text{phis-measure } g v p$   
⟨proof⟩

**lemma** *phis-measure-update-le'*:  $\text{phis-measure } g v p \leq \text{phis-measure } g v (\text{Mapping.update } k [] phis) \implies \text{phis-measure } g v (\text{Mapping.update } k a p) \leq \text{phis-measure } g v phis$   
⟨proof⟩

**lemma** *readArgs-phis-le*:  
 $\text{readVariableRecursive-readArgs-dom } (\text{Inl } (g, v, n, phis)) \implies (val,p) = \text{readVariableRecursive } g v n phis \implies \text{phis-measure } g v p \leq \text{phis-measure } g v phis$   
 $\text{readVariableRecursive-readArgs-dom } (\text{Inr } (g, v, n, phis, ms)) \implies (p,u) = \text{readArgs } g v n phis ms \implies \text{phis-measure } g v p \leq \text{phis-measure } g v phis$   
⟨proof⟩

**termination**  
⟨proof⟩

**declare**  $\text{readVariableRecursive.simps[simp del]} \text{ readArgs.simps[simp del]}$

**lemma** *fst-readVariableRecursive*:

**assumes**  $n \in \text{set } (\alpha n \ g)$   
**shows**  $\text{fst } (\text{readVariableRecursive } g \ v \ n \ \text{phis}) = \text{lookupDef } g \ n \ v$   
 $\langle \text{proof} \rangle$

**definition**  $\text{phis}'\text{-aux } g \ v \ ns \ (\text{phis}:: ('node, 'var) \ \text{sparse-phis}) \equiv \text{Mapping.Mapping}$   
 $(\lambda(m, v_2).$   
*(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  $\text{Some } (\text{map } (\lambda m. \text{lookupDef } g \ m \ v) (\text{predecessors } g \ m))$  else  $(\text{Mapping.lookup phis } (m, v_2))$ ))*

**lemma**  $\text{phis}'\text{-aux-keys-super}$ :  $\text{Mapping.keys } (\text{phis}'\text{-aux } g \ v \ ns \ \text{phis}) \supseteq \text{Mapping.keys phis}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{phiDefNodes-aux-in-unvisited}$ :  
**shows**  $\text{phiDefNodes-aux } g \ v \ un \ n \subseteq \text{set } un$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{phiDefNodes-aux-unvisited-monotonic}$ :  
**assumes**  $\text{set } un \subseteq \text{set } un'$   
**shows**  $\text{phiDefNodes-aux } g \ v \ un \ n \subseteq \text{phiDefNodes-aux } g \ v \ un' \ n$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{phiDefNodes-aux-single-pred}$ :  
**assumes**  $\text{predecessors } g \ n = [m]$   
**shows**  $\text{phiDefNodes-aux } g \ v \ (\text{removeAll } n \ un) \ m = \text{phiDefNodes-aux } g \ v \ un \ m$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{phis}'\text{-aux-finite}$ :  
**assumes**  $\text{finite } (\text{Mapping.keys phis})$   
**shows**  $\text{finite } (\text{Mapping.keys } (\text{phis}'\text{-aux } g \ v \ ns \ \text{phis}))$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{phiDefNodes-aux-redirect}$ :  
**assumes**  $\text{asm}: g \vdash n - ns \rightarrow m \ \forall n \in \text{set } ns. v \notin \text{defs } g \ n \ \text{length } (\text{predecessors } g \ n) \neq 1 \ \text{unvisitedPath } un \ ns$   
**assumes**  $n': n' \in \text{set } ns \ n' \in \text{phiDefNodes-aux } g \ v \ un \ m' \ m' \in \text{set } (\alpha n \ g)$   
**shows**  $n \in \text{phiDefNodes-aux } g \ v \ un \ m'$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{snd-readVariableRecursive}$ :  
**assumes**  $v \in \text{vars } g \ n \in \text{set } (\alpha n \ g) \ \text{finite } (\text{Mapping.keys phis})$   
 $\bigwedge n. (n, v) \in \text{Mapping.keys phis} \implies \text{length } (\text{predecessors } g \ n) \neq 1 \ \text{Mapping.lookup phis } (\text{Entry } g, v) \in \{\text{None}, \text{Some } []\}$   
**shows**  
 $\text{phis}'\text{-aux } g \ v \ \{n\} \ \text{phis} = \text{snd } (\text{readVariableRecursive } g \ v \ n \ \text{phis})$   
 $\text{set } ms \subseteq \text{set } (\alpha n \ g) \implies (\text{phis}'\text{-aux } g \ v \ (\text{set } ms) \ \text{phis}, \text{map } (\lambda m. \text{lookupDef } g \ m \ v) \ ms) = \text{readArgs } g \ v \ n \ \text{phis } ms$   
 $\langle \text{proof} \rangle$

**definition**  $aux-1\ g\ n = (\lambda v\ (uses,phis).$   
 $\text{let } (use,phis') = \text{readVariableRecursive } g\ v\ n\ phis\ \text{in}$   
 $(\text{Mapping.update } n\ (\text{insert use } (\text{lookup-multimap } uses\ n)))\ uses,\ phis')$   
 $)$

**definition**  $aux-2\ g\ n = \text{foldr } (aux-1\ g\ n)\ (\text{sorted-list-of-set } (uses\ g\ n))$

**abbreviation**  $init-state \equiv (\text{Mapping.empty}, \text{Mapping.empty})$

**abbreviation**  $from-sparse \equiv \lambda(n,v). (n,(v,n,PhiDef))$

**definition**  $uses'-phis'\ g = ($   
 $\text{let } (u,p) = \text{foldr } (aux-2\ g)\ (\alpha n\ g)\ init-state\ \text{in}$   
 $(u,\ \text{map-keys } from-sparse\ p)$   
 $)$

**lemma**  $from-sparse-inj: inj\ from-sparse$   
 $\langle proof \rangle$

**declare**  $uses'-phis'-def[\text{unfolded } aux-2-def[\text{abs-def}]\ aux-1-def,\ code]$

**lift-definition**  $phis'-code :: 'g \Rightarrow ('node, ('node, 'var)\ ssaVal)\ phis-code\ \text{is } phis'$   
 $\langle proof \rangle$

**lemma**  $foldr-prod: \text{foldr } (\lambda x\ y. (f1\ x\ (fst\ y),\ f2\ x\ (snd\ y)))\ xs\ y = (\text{foldr } f1\ xs$   
 $(fst\ y),\ \text{foldr } f2\ xs\ (snd\ y))$   
 $\langle proof \rangle$

**lemma**  $foldr-aux-1:$

**assumes**  $set\ us \subseteq uses\ g\ n$   $\text{Mapping.lookup } u\ n = None$   $\text{foldr } (aux-1\ g\ n)\ us$   
 $(u,p) = (u',p')$  **(is**  $\text{foldr } ?f\ -\ - = -)$

**assumes**  $finite\ (\text{Mapping.keys } p) \wedge n\ v. (n,v) \in \text{Mapping.keys } p \implies \text{length}$   
 $(\text{predecessors } g\ n) \neq 1 \wedge v. \text{Mapping.lookup } p\ (\text{Entry } g,v) \in \{None,\ \text{Some } []\}$

**shows**  $\text{lookupDef } g\ n\ 'set\ us = \text{lookup-multimap } u'\ n \wedge m. m \neq n \implies \text{Map-}$   
 $\text{ping.lookup } u'\ m = \text{Mapping.lookup } u\ m$

$\wedge m\ v. (\text{if } m \in \text{phiDefNodes-aux } g\ v\ [n \leftarrow \alpha n\ g. (n,v) \notin \text{Mapping.keys } p]\ n$   
 $\wedge v \in \text{set } us\ \text{then}$

$\text{Some } (\text{map } (\lambda m. \text{lookupDef } g\ m\ v)\ (\text{predecessors } g\ m))\ \text{else}$

$(\text{Mapping.lookup } p\ (m,v))) = \text{Mapping.lookup } p'\ (m,v)$

$\langle proof \rangle$

**lemma**  $foldr-aux-2:$

**assumes**  $set\ ns \subseteq \text{set } (\alpha n\ g)$   $\text{distinct } ns$   $\text{foldr } (aux-2\ g)\ ns\ init-state = (u',p')$

**shows**  $\wedge n. n \in \text{set } ns \implies uses'\ g\ n = \text{lookup-multimap } u'\ n \wedge n. n \notin \text{set } ns$   
 $\implies \text{Mapping.lookup } u'\ n = None$

$\wedge m\ v. (\text{if } \exists n \in \text{set } ns. m \in \text{phiDefNodes-aux } g\ v\ (\alpha n\ g)\ n \wedge v \in uses\ g\ n$   
 $\text{then}$

$\text{Some } (\text{map } (\lambda m. \text{lookupDef } g\ m\ v)\ (\text{predecessors } g\ m))\ \text{else}$

$None) = \text{Mapping.lookup } p'\ (m,v)$

$\langle proof \rangle$

**lemma** *fst-uses'-phis'*:  $uses' g = lookup-multimap (fst (uses'-phis' g))$   
 ⟨proof⟩

**lemma** *fst-uses'-phis'-in- $\alpha n$* :  $Mapping.keys (fst (uses'-phis' g)) \subseteq set (\alpha n g)$   
 ⟨proof⟩

**lemma** *snd-uses'-phis'*:  $phis'-code g = snd (uses'-phis' g)$   
 ⟨proof⟩

**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  
 ⟨proof⟩

**lemma** *weak-imp-transfer* [*transfer-rule*]:  
 $((=) \implies (=) \implies (\longrightarrow)) (\longrightarrow) (\longrightarrow)$   
 ⟨proof⟩

**lemma** *weak-conj-transfer* [*transfer-rule*]:  
 $((\longrightarrow) \implies (\longrightarrow) \implies (\longrightarrow)) (\wedge) (\wedge)$   
 ⟨proof⟩

**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**  $(\longrightarrow) (graph-path \alpha e \alpha n invar inEdges) (graph-path \alpha e2 \alpha n2 invar2 inEdges2)$

⟨proof⟩

**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*]:  $(A \implies (=)) \text{invar invar2}$   
    **and** [*transfer-rule*]:  $(A \implies (=)) \text{inEdges}' \text{inEdges2}$   
  **shows**  $(A \implies (=)) \text{inEdges} (\text{graph-path-base.inEdges inEdges2})$   
(*proof*)

**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*]:  $(A \implies (=)) \text{invar invar2}$   
    **and** [*transfer-rule*]:  $(A \implies (=)) \text{inEdges}' \text{inEdges2}$   
  **shows**  $(A \implies (=)) \text{predecessors} (\text{graph-path-base.predecessors inEdges2})$   
(*proof*)

**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*]:  $(A \implies (=)) \text{invar invar2}$   
    **and** [*transfer-rule*]:  $(A \implies (=)) \text{inEdges}' \text{inEdges2}$   
  **shows**  $(A \implies (=)) \text{successors} (\text{graph-path-base.successors } \alpha n2 \text{ inEdges2})$   
(*proof*)

**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*]:  $(A \implies (=)) \text{invar invar2}$   
    **and** [*transfer-rule*]:  $(A \implies (=)) \text{inEdges}' \text{inEdges2}$   
  **shows**  $(A \implies (=)) \text{path} (\text{graph-path-base.path } \alpha n2 \text{ invar2 inEdges2})$   
(*proof*)

**lemma** *path2-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*]:  $(A \implies (=)) \text{invar invar2}$   
    **and** [*transfer-rule*]:  $(A \implies (=)) \text{inEdges}' \text{inEdges2}$   
  **shows**  $(A \implies (=)) \text{path2} (\text{graph-path-base.path2 } \alpha n2 \text{ invar2 inEdges2})$   
(*proof*)



**lemma** *weak-Ex-transfer* [*transfer-rule*]:  $((=) \implies (\longrightarrow)) \implies (\longrightarrow)$  *Ex Ex*  
 ⟨*proof*⟩

**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*]:  $(G \implies (=)) \alpha e1 \alpha e2$   
**and** [*transfer-rule*]:  $(G \implies (=)) \alpha n1 \alpha n2$   
**and** [*transfer-rule*]:  $(G \implies (=)) \textit{invar1} \textit{invar2}$   
**and** [*transfer-rule*]:  $(G \implies (=)) \textit{inEdges1} \textit{inEdges2}$   
**and** [*transfer-rule*]:  $(G \implies (=)) \textit{Entry1} \textit{Entry2}$   
**shows**  $(\longrightarrow) (\textit{graph-Entry} \alpha e1 \alpha n1 \textit{invar1} \textit{inEdges1} \textit{Entry1}) (\textit{graph-Entry} \alpha e2 \alpha n2 \textit{invar2} \textit{inEdges2} \textit{Entry2})$   
 ⟨*proof*⟩

**context** *graph-Entry-base* **begin**

**lemma** *dominates-transfer* [*transfer-rule*]:  
**includes** *lifting-syntax*  
**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*]:  $(G \implies (=)) \textit{invar} \textit{invar2}$   
**and** [*transfer-rule*]:  $(G \implies (=)) \textit{inEdges}' \textit{inEdges2}$   
**and** [*transfer-rule*]:  $(G \implies (=)) \textit{Entry} \textit{Entry2}$   
**shows**  $(G \implies (=)) \textit{dominates} (\textit{graph-Entry-base.dominates} \alpha n2 \textit{invar2} \textit{inEdges2} \textit{Entry2})$   
 ⟨*proof*⟩

**end**

**context** *graph-Entry* **begin**

**context** **includes** *lifting-syntax*  
**begin**

**lemma** *shortestPath-transfer* [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *right-total G*  
**and** [*transfer-rule*]:  $(G \implies (=)) \alpha e \alpha e2$   
**and** [*transfer-rule*]:  $(G \implies (=)) \alpha n \alpha n2$   
**and** [*transfer-rule*]:  $(G \implies (=)) \textit{invar} \textit{invar2}$

```

    and [transfer-rule]: (G ==> (=)) inEdges' inEdges2
    and [transfer-rule]: (G ==> (=)) Entry Entry2
    shows (G ==> (=)) shortestPath (graph-Entry.shortestPath  $\alpha n2$  invar2 in-
Edges2 Entry2)
<proof>

```

```

lemma dominators-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 ==> (=)) dominators (graph-Entry.dominators  $\alpha n2$  invar2 in-
Edges2 Entry2)
<proof>

```

```

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>

```

```

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>

```

```

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*]: ( $G \text{ ===> } (=)$ )  $\alpha e1 \ \alpha e2$   
**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*  
**shows** *SSA-CFG.CFG*  $\alpha e1 \ \alpha n1 \ \textit{invar1} \ \textit{inEdges1} \ \textit{Entry1} \ \textit{defs1} \ \textit{uses1}$   
 $\rightarrow$  *SSA-CFG.CFG*  $\alpha e2 \ \alpha n2 \ \textit{invar2} \ \textit{inEdges2} \ \textit{Entry2} \ \textit{defs2} \ \textit{uses2}$   
*<proof>*

**context** *CFG-base* **begin**

**context includes** *lifting-syntax*  
**begin**

**lemma** *vars-transfer* [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *right-total G*  
**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ )  $\alpha e \ \alpha e2$   
**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*  
**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *defs defs2*  
**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *uses uses2*  
**shows** ( $G \text{ ===> } (=)$ ) *vars* (*CFG-base.vars*  $\alpha n2 \ \textit{uses2}$ )  
*<proof>*

**lemma** *defAss'-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*  
**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *defs defs2*  
**and** [*transfer-rule*]: ( $G \text{ ===> } (=)$ ) *uses uses2*  
**shows** ( $G \text{ ===> } (=)$ ) *defAss'* (*CFG-base.defAss'*  $\alpha n2 \ \textit{invar2} \ \textit{inEdges2} \ \textit{Entry2} \ \textit{defs2}$ )  
*<proof>*

**lemma** *defAss'Uses-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*

```

    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>

```

```

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
  <proof>

```

```

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>

```

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*]:  $(G \text{ ==== } > (=)) \alpha e \alpha e2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \alpha n \alpha n2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{invar} \textit{invar}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{inEdges}' \textit{inEdges}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{Entry} \textit{Entry}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{defs} \textit{defs}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{uses} \textit{uses}2$   
**shows**  $(G \text{ ==== } > (=)) \textit{phiDefNodes-aux} (\textit{CFG-Construct.phiDefNodes-aux} \textit{inEdges}2 \textit{defs}2)$   
*<proof>*

**lemma** *phiDefNodes-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*]:  $(G \text{ ==== } > (=)) \textit{invar} \textit{invar}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{inEdges}' \textit{inEdges}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{Entry} \textit{Entry}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{defs} \textit{defs}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{uses} \textit{uses}2$   
**shows**  $(G \text{ ==== } > (=)) \textit{phiDefNodes} (\textit{CFG-Construct.phiDefNodes} \alpha n2 \textit{inEdges}2 \textit{defs}2 \textit{uses}2)$   
*<proof>*

**lemma** *lookupDef-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*]:  $(G \text{ ==== } > (=)) \textit{invar} \textit{invar}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{inEdges}' \textit{inEdges}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{Entry} \textit{Entry}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{defs} \textit{defs}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{uses} \textit{uses}2$   
**shows**  $(G \text{ ==== } > (=)) \textit{lookupDef} (\textit{CFG-Construct.lookupDef} \alpha n2 \textit{inEdges}2 \textit{defs}2)$   
*<proof>*

**lemma** *defs'-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*]:  $(G \text{ ==== } > (=)) \textit{invar} \textit{invar}2$   
**and** [*transfer-rule*]:  $(G \text{ ==== } > (=)) \textit{inEdges}' \textit{inEdges}2$

```

    and [transfer-rule]: (G ==> (=)) Entry Entry2
    and [transfer-rule]: (G ==> (=)) defs defs2
    and [transfer-rule]: (G ==> (=)) uses uses2
  shows (G ==> (=)) defs' (CFG-Construct.defs' defs2)
<proof>

```

```

lemma 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 ==> (=)) uses' (CFG-Construct.uses'  $\alpha n2$  inEdges2 defs2 uses2)
<proof>

```

```

lemma phis'-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 ==> (=)) phis' (CFG-Construct.phis'  $\alpha n2$  inEdges2 defs2 uses2)
<proof>

```

```

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 ==> (=))  $\alpha e \alpha e2$ 

```

**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*  
**and** [*transfer-rule*]: ( $G \implies (=)$ ) *phis phis2*  
**shows** ( $G \implies (=)$ ) *phiDefs (CFG-SSA-base.phiDefs phis2)*  
*<proof>*

**lemma** *allDefs-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::'a  $\Rightarrow$  'node  $\Rightarrow$  'val set)*  
**and** [*transfer-rule*]: ( $G \implies (=)$ ) *uses (uses2::'a  $\Rightarrow$  'node  $\Rightarrow$  'val set)*  
**and** [*transfer-rule*]: ( $G \implies (=)$ ) *phis phis2*  
**shows** ( $G \implies (=)$ ) *allDefs (CFG-SSA-base.allDefs defs2 phis2)*  
*<proof>*

**lemma** *phiUses-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*  
**and** [*transfer-rule*]: ( $G \implies (=)$ ) *phis phis2*  
**shows** ( $G \implies (=)$ ) *phiUses (CFG-SSA-base.phiUses  $\alpha n2$  inEdges2 phis2)*  
*<proof>*

**lemma** *allUses-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*  
**and** [*transfer-rule*]: ( $G \implies (=)$ ) *phis phis2*  
**shows** ( $G \implies (=)$ ) *allUses (CFG-SSA-base.allUses  $\alpha n2$  inEdges2 uses2*  
*phis2)*  
*<proof>*

```

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>

```

```

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 αn2 invar2 inEdges2 En-
try2 defs2 phis2)
  <proof>

```

```

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 ==> (=)) 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 ==> (=)) phiDefs (CFG-SSA-base.phiDefs (λg. Mapping.lookup
(phis2 g)))
(G ==> (=)) allDefs (CFG-SSA-base.allDefs defs2 (λg. Mapping.lookup
(phis2 g)))
(G ==> (=)) phiUses (CFG-SSA-base.phiUses αn2 inEdges2 (λg. Map-
ping.lookup (phis2 g)))
(G ==> (=)) allUses (CFG-SSA-base.allUses αn2 inEdges2 (usesOf ∘
uses2) (λg. Mapping.lookup (phis2 g)))
(G ==> (=)) defAss (CFG-SSA-base.defAss αn2 invar2 inEdges2 Entry2
defs2 (λg. Mapping.lookup (phis2 g)))
⟨proof⟩

```

**end**

**lemma** *CFG-SSA-transfer* [transfer-rule]:

```

includes lifting-syntax
assumes [transfer-rule]: right-total G
and [transfer-rule]: (G ==> (=)) αe1 αe2
and [transfer-rule]: (G ==> (=)) αn1 αn2
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
and [transfer-rule]: (G ==> (=))phis1 phis2
shows CFG-SSA αe1 αn1 invar1 inEdges1 Entry1 defs1 uses1 phis1
→ CFG-SSA αe2 αn2 invar2 inEdges2 Entry2 defs2 uses2 phis2
⟨proof⟩

```

**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 ≡ (λ-. x)

**context** *CFG-SSA-Transformed-notriv-base* **begin**

**definition** [*code*]: *substNext-code* g next = *the* (*the-trivial* (*snd* next) (*the* (*phis* g next)))

**definition** [*code*]: *substNext-code* g next ≡ λv. *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)*  
 ⟨*proof*⟩

**end**

**type-synonym** (*'g, 'node, 'val*) *chooseNext-code* = (*'node*  $\Rightarrow$  *'val set*)  $\Rightarrow$  (*'node, 'val*) *phis-code*  $\Rightarrow$  *'g*  $\Rightarrow$  (*'node*  $\times$  *'val*)

**locale** *CFG-SSA-Transformed-notriv-base-code* =  
*ssa:CFG-SSA-wf-base-code*  $\alpha e \alpha n$  *invar inEdges' Entry defs uses phis* +  
*CFG-SSA-Transformed-notriv-base*  $\alpha e \alpha n$  *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*)

**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*

**begin**  
**definition** [code]: *cond-code g = ssa.redundant-code g*

**definition** *uses'-codem* :: *'g*  $\Rightarrow$  *'node*  $\times$  *'val*  $\Rightarrow$  *'val*  $\Rightarrow$  (*'val, 'node set*) *mapping*  
 $\Rightarrow$  (*'node, 'val set*) *mapping*  
**where** [code]: *uses'-codem g next next' nodes-of-uses =*  
*fold ( $\lambda n$ . *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*  $\Rightarrow$  *'node*  $\times$  *'val*  $\Rightarrow$  *'val*  $\Rightarrow$  *'val set*  $\Rightarrow$  (*'val, 'node set*) *mapping*  $\Rightarrow$  (*'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'  $\in$  phiVals) then Mapping.map-default next' {} ( $\lambda ns$ . *ns*  $\cup$  *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 ⇒ '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 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'
  ⟨proof⟩

  lemma the-trivial-THE-isTrivial: [ phi g v = Some vs; trivial g v ] ⇒ the-trivial
v vs = Some (The (isTrivialPhi g v))
  ⟨proof⟩

  lemma substitution-code-correct:
  assumes redundant g
  shows substitution g = substitution-code g (chooseNext' g)
  ⟨proof⟩

  lemma substNext-code-correct:
  assumes redundant g
  shows substNext g = substNext-code g (chooseNext' g)
  ⟨proof⟩

  lemma uses'-code-correct:
  assumes redundant g
  shows uses' g = uses'-code g (chooseNext' g)
  ⟨proof⟩

end

context CFG-SSA-Transformed-notriv-linorder
begin
  lemma substAll-terminates: while-option (cond g) (step g) (uses g, phis g) ≠
None
  ⟨proof⟩
end

locale CFG-SSA-Transformed-notriv-linorder-code =
  CFG-SSA-Transformed-code αe αn invar inEdges' Entry oldDefs oldUses defs
uses phis var
+ CFG-SSA-Transformed-notriv-base-code αe αn invar inEdges' Entry oldDefs
oldUses defs uses phis var chooseNext-all
+ CFG-SSA-Transformed-notriv-linorder 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
+
assumes chooseNext-all-code:
  CFG-SSA-Transformed-code αe αn invar inEdges' Entry oldDefs oldUses defs u
  p var ⇒
  CFG-SSA-wf-base-code.redundant-code p g ⇒
  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 αe αn invar inEdges' En-
  try oldDefs oldUses defs uses phis var chooseNext-all
+
  CFG-SSA-step αe αn invar inEdges' Entry oldDefs oldUses defs usesOf ∘ uses
  λg. Mapping.lookup (phis g) var λuses phis. chooseNext-all uses (Mapping.Mapping
  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
  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 and
  g :: 'g

context CFG-SSA-Transformed-notriv-linorder-code
begin
  abbreviation u-g g u ≡ uses(g:=u)
  abbreviation p-g g p ≡ phis(g:=p)
  abbreviation cN ≡ (λuses phis. chooseNext-all uses (Mapping.Mapping phis))

  interpretation uninst-code: CFG-SSA-Transformed-notriv-base-code αe αn in-
  var inEdges' Entry oldDefs oldUses defs u p var chooseNext-all

```

**for**  $u\ p$   
 $\langle proof \rangle$

**interpretation**  $uninst$ : *CFG-SSA-Transformed-notriv-base*  $\alpha e\ \alpha n\ invar\ inEdges'$   
*Entry oldDefs oldUses defs u p var cN*

**for**  $u\ p$   
 $\langle proof \rangle$

**lemma**  $phis'$ -code-correct:  
**assumes**  $ssa.redundant\ g$   
**shows**  $phis'\ g = Mapping.lookup\ (phis'\text{-code}\ g\ (chooseNext'\ g))$   
 $\langle proof \rangle$

**lemma**  $redundant\ ign[simp]$ :  $uninst\text{-code}.ssa.redundant\text{-code}\ (const\ p)\ g = uninst\text{-code}.ssa.redundant\text{-code}\ (phis(g:=p))\ g$   
 $\langle proof \rangle$

**lemma**  $uses'\text{-ign}[simp]$ :  $uninst\text{-code}.uses'\text{-codem}\ (const\ u)\ g = uninst\text{-code}.uses'\text{-codem}\ (u-g\ g\ u)\ g$   
 $\langle proof \rangle$

**lemma**  $phis'\text{-ign}[simp]$ :  $uninst\text{-code}.phis'\text{-code}\ (const\ p)\ g = uninst\text{-code}.phis'\text{-code}\ (phis(g:=p))\ g$   
 $\langle proof \rangle$

**lemma**  $phis'\text{-m-ign}[simp]$ :  $uninst\text{-code}.phis'\text{-codem}\ (const\ p)\ g = uninst\text{-code}.phis'\text{-codem}\ (phis(g:=p))\ g$   
 $\langle proof \rangle$

**lemma**  $set\text{-sorted-list-of-set-phis-dom}[simp]$ :  
 $set\ (sorted\text{-list-of-set}\ \{x \in dom\ (Mapping.lookup\ (phis\ g)).\ P\ x\}) = \{x \in dom\ (Mapping.lookup\ (phis\ g)).\ P\ x\}$   
 $\langle proof \rangle$

**lemma**  $phis'\text{-codem-correct}$ :  
**assumes**  $g \vdash nodes\text{-of-phis} \approx_{\varphi} (ssa.phiNodes\text{-of}\ g)$  **and**  $next \in Mapping.keys\ (phis\ g)$   
**shows**  $phis'\text{-codem}\ g\ next\ (substitution\text{-code}\ g\ next)\ nodes\text{-of-phis} = phis'\text{-code}\ g\ next$   
 $\langle proof \rangle$

**lemma**  $uses\text{-transfer}[transfer\text{-rule}]$ :  $(rel\text{-fun}\ (=)\ (pcr\text{-mapping}\ (=)\ (=)))\ (\lambda g\ n.\ Mapping.lookup\ (uses\ g)\ n)\ uses$   
 $\langle proof \rangle$

**lemma**  $uses'\text{-codem-correct}$ :  
**assumes**  $g \vdash nodes\text{-of-uses} \approx_{\varphi} ssa.useNodes\text{-of}\ g$  **and**  $next \in Mapping.keys\ (phis\ g)$   
**shows**  $usesOf\ (uses'\text{-codem}\ g\ next\ (substitution\text{-code}\ g\ next)\ nodes\text{-of-uses}) =$

*uses'-code g next*  
 ⟨proof⟩

**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  
 ⟨proof⟩

**lemma** *cN-transfer* [transfer-rule]: (rel-fun (=) (rel-fun (pcr-mapping (=) (=)) (=))) cN chooseNext-all  
 ⟨proof⟩

**lemma** *usesOf-transfer* [transfer-rule]: (rel-fun (pcr-mapping (=) (=)) (=)) (λm x. case-option {} id (m x)) usesOf  
 ⟨proof⟩

**lemma** *dom-phis'-codem*:  
**assumes**  $\bigwedge ns. \text{Mapping.lookup nodes-of-phis (snd next) = Some ns} \implies \text{finite ns}$   
**shows**  $\text{dom (Mapping.lookup (phis'-codem g next next' nodes-of-phis))} = \text{dom (Mapping.lookup (phis g))} \cup (\text{case-option } \{\} \text{ id (Mapping.lookup nodes-of-phis (snd next))}) - \{\text{next}\}$   
 ⟨proof⟩

**lemma** *dom-phis'-code* [simp]:  
**shows**  $\text{dom (Mapping.lookup (phis'-code g next))} = \text{dom (Mapping.lookup (phis g))} - \{v. \text{snd } v = \text{snd next}\}$   
 ⟨proof⟩

**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*  
 ⟨proof⟩

**lemma** *lookup-phis'-codem-next*:  
**assumes**  $\bigwedge ns. \text{Mapping.lookup nodes-of-phis (snd next) = Some ns} \implies \text{finite ns}$   
**shows**  $\text{Mapping.lookup (phis'-codem g next next' nodes-of-phis) next} = \text{None}$   
 ⟨proof⟩

**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⟩

**lemma** *lookup-nodes-of-phis'-subst* [simp]:

$Mapping.lookup (nodes-of-phis' g next (substitution-code g next) nodes-of-phis)$   
 $(substitution-code g next) =$   
 $Some ((case-option \{\} (Set.remove next) (Mapping.lookup nodes-of-phis (substitution-code$   
 $g next))) \cup (case-option \{\} (Set.remove next) (Mapping.lookup nodes-of-phis (snd$   
 $next))))$   
 $\langle proof \rangle$

**lemma lookup-nodes-of-phis'-not-subst:**  
 $v \neq substitution-code g next \implies$   
 $Mapping.lookup (nodes-of-phis' g next (substitution-code g next) nodes-of-phis) v$   
 $= (if v = snd next then None else Mapping.lookup nodes-of-phis v)$   
 $\langle proof \rangle$

**lemma lookup-phis'-code:**  
 $Mapping.lookup (phis'-code g next) v = (if snd v = snd next then None else$   
 $map-option (map (substNext-code g next)) (Mapping.lookup (phis g) v))$   
 $\langle proof \rangle$

**lemma phi-equiv-mappingE':**  
**assumes**  $g \vdash m_1 \approx_\varphi ssa.phiNodes-of g$   
**and**  $Mapping.lookup (phis g) x = Some vs$  **and**  $b \in set vs$  **and**  $b \in snd \text{ `}$   
 $Mapping.keys (phis g)$   
**obtains**  $Mapping.lookup m_1 b = Some \{n \in Mapping.keys (phis g). b \in set$   
 $(the (Mapping.lookup (phis g) n))\}$   
 $\langle proof \rangle$

**lemma phi-equiv-mappingE:**  
**assumes**  $g \vdash m_1 \approx_\varphi ssa.phiNodes-of g$  **and**  $b \in Mapping.keys (phis g)$   
**and**  $Mapping.lookup (phis g) x = Some vs$  **and**  $snd b \in set vs$   
**obtains**  $ns$  **where**  $Mapping.lookup m_1 (snd b) = Some \{n \in Mapping.keys$   
 $(phis g). snd b \in set (the (Mapping.lookup (phis g) n))\}$   
 $\langle proof \rangle$

**lemma phi-equiv-mappingE2':**  
**assumes**  $g \vdash m_1 \approx_\varphi ssa.phiNodes-of g$   
**and**  $b \in snd \text{ ` } Mapping.keys (phis g)$   
**and**  $\forall \varphi \in Mapping.keys (phis g). b \notin set (the (Mapping.lookup (phis g) \varphi))$   
**shows**  $Mapping.lookup m_1 b = None \vee Mapping.lookup m_1 b = Some \{\}$   
 $\langle proof \rangle$

**lemma keys-phis'-codem [simp]:**  $Mapping.keys (phis'-codem g next next' (ssa.phiNodes-of$   
 $g)) = Mapping.keys (phis g) - \{next\}$   
 $\langle proof \rangle$

**lemma keys-phis'-codem':**  
**assumes**  $g \vdash nodes-of-phis \approx_\varphi ssa.phiNodes-of g$  **and**  $next \in Mapping.keys$   
 $(phis g)$   
**shows**  $Mapping.keys (phis'-codem g next next' nodes-of-phis) = Mapping.keys$   
 $(phis g) - \{next\}$



*<proof>*

**lemma** *triv-phis'-correct:*

**assumes**  $g \vdash \text{nodes-of-phis} \approx_{\varphi} \text{ssa.phiNodes-of } g$  **and**  $\text{next} \in \text{Mapping.keys}$   
(*phis*  $g$ ) **and**  $\text{ssa.trivial } g$  (*snd*  $\text{next}$ )  
**shows**  $\text{uninst-code.triv-phis}' (\text{const } (\text{phis}'\text{-codem } g \text{ next } (\text{substitution-code } g \text{ next}) \text{ nodes-of-phis})) g \text{ next } (\text{ssa.trivial-phis } g) \text{ nodes-of-phis} = \text{uninst-code.ssa.trivial-phis}$   
( $\text{const } (\text{phis}'\text{-codem } g \text{ next } (\text{substitution-code } g \text{ next}) \text{ nodes-of-phis})$ )  $g$   
*<proof>*

**lemma** *nodes-of-phis'-correct:*

**assumes**  $g \vdash \text{nodes-of-phis} \approx_{\varphi} \text{ssa.phiNodes-of } g$   
**and**  $\text{next} \in \text{Mapping.keys}$  (*phis*  $g$ ) **and**  $\text{ssa.trivial } g$  (*snd*  $\text{next}$ )  
**shows**  $g \vdash (\text{nodes-of-phis}' g \text{ next } (\text{substitution-code } g \text{ next}) \text{ nodes-of-phis}) \approx_{\varphi}$   
( $\text{uninst-code.ssa.phiNodes-of } (\text{const } (\text{phis}'\text{-codem } g \text{ next } (\text{substitution-code } g \text{ next}) \text{ nodes-of-phis})) g$ )  
*<proof>*

**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**  $\text{ssa.trivial } g$  (*snd*  $\text{next}$ )  
**shows**  $g \vdash (\text{nodes-of-uses}' g \text{ next } (\text{substitution-code } g \text{ next}) (\text{Mapping.keys } (\text{ssa.phidefNodes}$   
 $g)) \text{ nodes-of-uses}) \approx_{\varphi} (\text{uninst-code.ssa.useNodes-of } (\text{const } (\text{uses}'\text{-codem } g \text{ next}$   
 $(\text{substitution-code } g \text{ next}) \text{ nodes-of-uses})) g$ )  
*<proof>*

**definition**[*code*]: *substAll-efficient*  $g \equiv$

*let*  $\text{phiVals} = \text{Mapping.keys } (\text{ssa.phidefNodes } g)$ ;  
 $u = \text{uses } g$ ;  
 $p = \text{phis } g$ ;  
 $tp = \text{ssa.trivial-phis } g$ ;  
 $\text{nou} = \text{ssa.useNodes-of } g$ ;  
 $\text{nop} = \text{ssa.phiNodes-of } g$   
*in*  
*while*  
( $\lambda((u,p), \text{triv-phis}, \text{nodes-of-uses}, \text{nodes-of-phis}). \neg \text{Set.is-empty triv-phis}$ )  
( $\lambda((u,p), \text{triv-phis}, \text{nodes-of-uses}, \text{nodes-of-phis}). \text{let}$   
 $\text{next} = \text{Max triv-phis}$ ;  
 $\text{next}' = \text{uninst-code.substitution-code } (\text{const } p) g \text{ next}$ ;  
( $u', p'$ ) =  $\text{uninst-code.step-codem } (\text{const } u) (\text{const } p) g \text{ next next}' \text{ nodes-of-uses}$   
 $\text{nodes-of-phis}$ ;  
 $tp' = \text{uninst-code.triv-phis}' (\text{const } p') g \text{ next triv-phis nodes-of-phis}$ ;  
 $\text{nou}' = \text{uninst-code.nodes-of-uses}' g \text{ next next}' \text{ phiVals nodes-of-uses}$ ;  
 $\text{nop}' = \text{uninst-code.nodes-of-phis}' g \text{ next next}' \text{ nodes-of-phis}$   
*in* (( $u', p'$ ),  $tp'$ ,  $\text{nou}'$ ,  $\text{nop}'$ ))  
( $(u, p), tp, \text{nou}, \text{nop}$ )

**abbreviation**  $u\text{-}c \ x \equiv \text{const } (\text{usesOf } (\text{fst } x))$

**abbreviation**  $p\text{-}c \ x \equiv \text{const } (\text{Mapping.lookup } (\text{snd } x))$

**abbreviation**  $u\ g\ x \equiv u\text{-}g\ g\ (fst\ x)$   
**abbreviation**  $p\ g\ x \equiv p\text{-}g\ g\ (snd\ x)$

**lemma** *usesOf-upd* [simp]:  $(usesOf \circ u\ g\ s1)(g := usesOf\ us) = usesOf \circ u\text{-}g\ g\ us$   
 <proof>

**lemma** *keys-uses'-codem* [simp]:  $Mapping.keys\ (uses'\text{-codem}\ g\ next\ (substitution\text{-code}\ g\ next)\ (ssa.useNodes\text{-of}\ g)) = Mapping.keys\ (uses\ g)$   
 <proof>

**lemma** *keys-uses'-codem'*:  $\llbracket g \vdash nodes\text{-of}\text{-uses} \approx_{\varphi} ssa.useNodes\text{-of}\ g; next \in Mapping.keys\ (phis\ g) \rrbracket$   
 $\implies Mapping.keys\ (uses'\text{-codem}\ g\ next\ (substitution\text{-code}\ g\ next)\ nodes\text{-of}\text{-uses})$   
 $= Mapping.keys\ (uses\ g)$   
 <proof>

**lemma** *triv-phis-base* [simp]:  $uninst\text{-code}.ssa.trivial\text{-phis}\ (const\ (phis\ g))\ g = ssa.trivial\text{-phis}\ g$   
 <proof>

**lemma** *useNodes-of-base* [simp]:  $uninst\text{-code}.ssa.useNodes\text{-of}\ (const\ (uses\ g))\ g = ssa.useNodes\text{-of}\ g$   
 <proof>

**lemma** *phiNodes-of-base* [simp]:  $uninst\text{-code}.ssa.phiNodes\text{-of}\ (const\ (phis\ g))\ g = ssa.phiNodes\text{-of}\ g$   
 <proof>

**lemma** *phi-equiv-mapping-refl* [simp]:  $uninst\text{-code}.phi\text{-equiv}\text{-mapping}\ ph\ g\ m\ m$   
 <proof>

**lemma** *substAll-efficient-code* [code]:  
 $substAll\ g = map\text{-prod}\ usesOf\ Mapping.lookup\ (fst\ (substAll\text{-efficient}\ g))$   
 <proof>

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- $\alpha e$  :: ('node, 'edge) edge set  
 gen- $\alpha n$  :: '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  
(const {}) (const {}) ext

**abbreviation** (input) ign f g (-::unit)  $\equiv$  f g

**lemma** set-iterator-foldri-Nil [simp, intro!]: set-iterator (foldri [] ) {}  
 <proof>

**lemma** set-iterator-foldri-one [simp, intro!]: set-iterator (foldri [a] ) {a}  
 <proof>

**abbreviation** gen-inEdges' g n  $\equiv$  map ( $\lambda(f,d,t). (f,d)$ ) (gen-inEdges g n)

**lemma** gen-cfg-inhabited: let g = trivial-gen-cfg ext in CFG-wf (ign gen- $\alpha e$  g)  
(ign gen- $\alpha n$  g) (const True) (ign gen-inEdges' g) (ign gen-Entry g) (ign gen-defs  
g) (ign gen-uses g)  
<proof>

**typedef** ('node, 'var, 'edge) gen-cfg-wf = {g :: ('node::linorder, 'var::linorder,  
'edge) gen-cfg.  
 CFG-wf (ign gen- $\alpha e$  g) (ign gen- $\alpha n$  g) (const True) (ign gen-inEdges' g) (ign  
gen-Entry g) (ign gen-defs g) (ign gen-uses g)}  
<proof>

**setup-lifting** type-definition-gen-cfg-wf

**lift-definition** gen-wf- $\alpha n$  :: ('node::linorder, 'var::linorder, 'edge) gen-cfg-wf  $\Rightarrow$   
'node list **is** gen- $\alpha n$  <proof>

**lift-definition** gen-wf- $\alpha e$  :: ('node::linorder, 'var::linorder, 'edge) gen-cfg-wf  $\Rightarrow$   
('node, 'edge) edge set **is** gen- $\alpha e$  <proof>

**lift-definition** gen-wf-inEdges :: ('node::linorder, 'var::linorder, 'edge) gen-cfg-wf  
 $\Rightarrow$  'node  $\Rightarrow$  ('node, 'edge) edge list **is** gen-inEdges <proof>

**lift-definition** gen-wf-Entry :: ('node::linorder, 'var::linorder, 'edge) gen-cfg-wf  
 $\Rightarrow$  'node **is** gen-Entry <proof>

**lift-definition** gen-wf-defs :: ('node::linorder, 'var::linorder, 'edge) gen-cfg-wf  $\Rightarrow$   
'node  $\Rightarrow$  'var set **is** gen-defs <proof>

**lift-definition** gen-wf-uses :: ('node::linorder, 'var::linorder, 'edge) gen-cfg-wf  $\Rightarrow$   
'node  $\Rightarrow$  'var set **is** gen-uses <proof>

**abbreviation** gen-wf-invar  $\equiv$  const True

**abbreviation** *gen-wf-inEdges'* *g n*  $\equiv \text{map } (\lambda(f,d,t). (f,d)) (gen-wf-inEdges\ g\ n)$

**lemma** *gen-wf-inEdges'-transfer* [*transfer-rule*]: *rel-fun cr-gen-cfg-wf* (=) *gen-inEdges'*  
*gen-wf-inEdges'*  
{*proof*}

**lemma** *gen-wf-invar-trans*: *rel-fun cr-gen-cfg-wf* (=) *gen-wf-invar gen-wf-invar*  
{*proof*}

**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'*

{*proof*}

**record** (*'node, 'var, 'edge, 'val*) *gen-ssa-cfg* = (*'node, 'var, 'edge*) *gen-cfg* +  
*gen-ssa-defs* :: *'node*  $\Rightarrow$  *'val set*  
*gen-ssa-uses* :: (*'node, 'val set*) *mapping*  
*gen-phis* :: (*'node, 'val*) *phis-code*  
*gen-var* :: *'val*  $\Rightarrow$  *'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- $\alpha$ e g*) (*ign gen- $\alpha$ n 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*)}

{*proof*}

**setup-lifting** *type-definition-gen-ssa-cfg-wf*

**lift-definition** *gen-ssa-wf- $\alpha$ n* :: (*'node::linorder, 'var::linorder, 'edge, 'val::linorder*)  
*gen-ssa-cfg-wf*  $\Rightarrow$  *'node list* **is** *gen- $\alpha$ n* {*proof*}

**lift-definition** *gen-ssa-wf- $\alpha$ e* :: (*'node::linorder, 'var::linorder, 'edge, 'val::linorder*)  
*gen-ssa-cfg-wf*  $\Rightarrow$  (*'node, 'edge*) *edge set* **is** *gen- $\alpha$ e* {*proof*}

**lift-definition** *gen-ssa-wf-inEdges* :: (*'node::linorder, 'var::linorder, 'edge, 'val::linorder*)  
*gen-ssa-cfg-wf*  $\Rightarrow$  *'node*  $\Rightarrow$  (*'node, 'edge*) *edge list* **is** *gen-inEdges* {*proof*}

**lift-definition**  $gen\text{-}ssa\text{-}wf\text{-}Entry :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)$   
 $gen\text{-}ssa\text{-}cfg\text{-}wf \Rightarrow 'node \text{ is } gen\text{-}Entry \langle proof \rangle$   
**lift-definition**  $gen\text{-}ssa\text{-}wf\text{-}defs :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)$   
 $gen\text{-}ssa\text{-}cfg\text{-}wf \Rightarrow 'node \Rightarrow 'var \text{ set is } gen\text{-}defs \langle proof \rangle$   
**lift-definition**  $gen\text{-}ssa\text{-}wf\text{-}uses :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)$   
 $gen\text{-}ssa\text{-}cfg\text{-}wf \Rightarrow 'node \Rightarrow 'var \text{ set is } gen\text{-}uses \langle proof \rangle$   
**lift-definition**  $gen\text{-}ssa\text{-}wf\text{-}ssa\text{-}defs :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)$   
 $gen\text{-}ssa\text{-}cfg\text{-}wf \Rightarrow 'node \Rightarrow 'val \text{ set is } gen\text{-}ssa\text{-}defs \langle proof \rangle$   
**lift-definition**  $gen\text{-}ssa\text{-}wf\text{-}ssa\text{-}uses :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)$   
 $gen\text{-}ssa\text{-}cfg\text{-}wf \Rightarrow ('node, 'val \text{ set}) \text{ mapping is } gen\text{-}ssa\text{-}uses \langle proof \rangle$   
**lift-definition**  $gen\text{-}ssa\text{-}wf\text{-}phis :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)$   
 $gen\text{-}ssa\text{-}cfg\text{-}wf \Rightarrow ('node, 'val) \text{ phis-code is } gen\text{-}phis \langle proof \rangle$   
**lift-definition**  $gen\text{-}ssa\text{-}wf\text{-}var :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)$   
 $gen\text{-}ssa\text{-}cfg\text{-}wf \Rightarrow 'val \Rightarrow 'var \text{ is } gen\text{-}var \langle proof \rangle$

**abbreviation**  $gen\text{-}ssa\text{-}wf\text{-}inEdges' g n \equiv map (\lambda(f,d,t). (f,d)) (gen\text{-}ssa\text{-}wf\text{-}inEdges g n)$

**lemma**  $gen\text{-}ssa\text{-}wf\text{-}inEdges'\text{-}transfer [transfer\text{-}rule]: rel\text{-}fun cr\text{-}gen\text{-}ssa\text{-}cfg\text{-}wf (=)$   
 $gen\text{-}inEdges' gen\text{-}ssa\text{-}wf\text{-}inEdges'$   
 $\langle proof \rangle$

**global-interpretation**  $uninst: CFG\text{-}SSA\text{-}wf\text{-}base\text{-}code gen\text{-}ssa\text{-}wf\text{-}\alpha e gen\text{-}ssa\text{-}wf\text{-}\alpha n$   
 $gen\text{-}wf\text{-}invar gen\text{-}ssa\text{-}wf\text{-}inEdges' gen\text{-}ssa\text{-}wf\text{-}Entry gen\text{-}ssa\text{-}wf\text{-}ssa\text{-}defs u p$   
**for**  $u$  **and**  $p$   
**defines**  
 $uninst\text{-}predecessors = uninst.predecessors$   
**and**  $uninst\text{-}successors = uninst.successors$   
**and**  $uninst\text{-}phiDefs = uninst.phiDefs$   
**and**  $uninst\text{-}phiUses = uninst.phiUses$   
**and**  $uninst\text{-}allDefs = uninst.allDefs$   
**and**  $uninst\text{-}allUses = uninst.allUses$   
**and**  $uninst\text{-}allVars = uninst.allVars$   
**and**  $uninst\text{-}isTrivialPhi = uninst.isTrivialPhi$   
**and**  $uninst\text{-}trivial = uninst.trivial\text{-}code$   
**and**  $uninst\text{-}redundant = uninst.redundant\text{-}code$   
**and**  $uninst\text{-}phi = uninst.phi$   
**and**  $uninst\text{-}defNode = uninst.defNode$   
**and**  $uninst\text{-}trivial\text{-}phis = uninst.trivial\text{-}phis$   
**and**  $uninst\text{-}phidefNodes = uninst.phidefNodes$   
**and**  $uninst\text{-}useNodes\text{-}of = uninst.useNodes\text{-}of$   
**and**  $uninst\text{-}phinodes\text{-}of = uninst.phinodes\text{-}of$   
 $\langle proof \rangle$

**definition**  $uninst\text{-}chooseNext u p g \equiv Max (uninst\text{-}trivial\text{-}phis (const p) g)$

**lemma**  $gen\text{-}ssa\text{-}wf\text{-}invar\text{-}trans: rel\text{-}fun cr\text{-}gen\text{-}ssa\text{-}cfg\text{-}wf (=) gen\text{-}wf\text{-}invar gen\text{-}wf\text{-}invar$   
 $\langle proof \rangle$

**declare** *graph-path-base.transfer-rules*[*OF gen-ssa-cfg-wf.right-total gen-ssa-wf-αe.transfer gen-ssa-wf-αn.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-αe.transfer gen-ssa-wf-αn.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-αe.transfer gen-ssa-wf-αn.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-αn g) gen-wf-invar (ign gen-inEdges' g) g' n ns m*  $\longleftrightarrow$  *graph-path-base.path2 gen-αn gen-wf-invar gen-inEdges' g n ns m*  
 <proof>

**lemma** *allDefs-ign[simp]*: *CFG-SSA-base.allDefs (ign gen-ssa-defs g) (ign Mapping.lookup (gen-phs g)) ga n = CFG-SSA-base.allDefs gen-ssa-defs (λg. Mapping.lookup (gen-phs g)) g n*  
 <proof>

**lemma** *successors-ign[simp]*: *graph-path-base.successors (ign gen-αn g) (ign gen-inEdges' g) ga n = graph-path-base.successors gen-αn gen-inEdges' g n*  
 <proof>

**lemma** *predecessors-ign[simp]*: *graph-path-base.predecessors (ign gen-inEdges' g) ga n = graph-path-base.predecessors gen-inEdges' g n*  
 <proof>

**lemma** *phiDefs-ign[simp]*: *CFG-SSA-base.phiDefs (ign Mapping.lookup (gen-phs g)) ga = CFG-SSA-base.phiDefs (λg. Mapping.lookup (gen-phs g)) g*  
 <proof>

**lemma** *defAss-ign[simp]*: *CFG-SSA-base.defAss (ign gen-αn g) gen-wf-invar (ign gen-inEdges' g) (ign gen-Entry g) (ign gen-ssa-defs g) (ign Mapping.lookup (gen-phs g)) ga*  
 = *CFG-SSA-base.defAss gen-αn gen-wf-invar gen-inEdges' gen-Entry gen-ssa-defs (λg. Mapping.lookup (gen-phs g)) g*  
 <proof>

**lemma** *allUses-ign[simp]*: *CFG-SSA-base.allUses (ign gen-αn g) (ign gen-inEdges' g) (usesOf ∘ ign gen-ssa-uses g) (ign Mapping.lookup (gen-phs g)) ga m*  
 = *CFG-SSA-base.allUses gen-αn gen-inEdges' (usesOf ∘ gen-ssa-uses) (λg. Mapping.lookup (gen-phs g)) g m*  
 <proof>

**lemma** *defAss'-ign[simp]*: *CFG-base.defAss' (ign gen-αn g) gen-wf-invar (ign gen-inEdges' g) (ign gen-Entry g) (ign gen-defs g) ga*  
 = *CFG-base.defAss' gen-αn gen-wf-invar gen-inEdges' gen-Entry gen-defs g*  
 <proof>

**global-interpretation** *gen-ssa-wf-notriv*: *CFG-SSA-Transformed-notriv-linorder-code gen-ssa-wf-αe gen-ssa-wf-αn 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-phs 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*  
 ⟨proof⟩

**global-interpretation** *uninst-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 uninst-chooseNext*  
**for** *u* **and** *p*

**defines**

*uninst-code-step-code* = *uninst-code.step-codem* **and**  
*uninst-code-phis'* = *uninst-code.phis'-codem* **and**  
*uninst-code-uses'* = *uninst-code.uses'-codem* **and**  
*uninst-code-substNext* = *uninst-code.substNext-code* **and**  
*uninst-code-substitution* = *uninst-code.substitution-code* **and**  
*uninst-code-triv-phis'* = *uninst-code.triv-phis'* **and**  
*uninst-code-nodes-of-uses'* = *uninst-code.nodes-of-uses'* **and**  
*uninst-code-nodes-of-phis'* = *uninst-code.nodes-of-phis'*

⟨proof⟩

**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* ⟨proof⟩

**lemma** *gen- $\alpha$ e-wf-extend* [simp]:

*gen- $\alpha$ e* (*gen-cfg-wf-extend gen-cfg-wf* ( $\downarrow$ *gen-ssa-defs* = *d*, *gen-ssa-uses* = *u*, *gen-phis*  
 = *p*, *gen-var* = *v*))  
 = *gen-wf- $\alpha$ e gen-cfg-wf*  
 ⟨proof⟩

**lemma** *gen- $\alpha$ n-wf-extend* [simp]:

*gen- $\alpha$ n* (*gen-cfg-wf-extend gen-cfg-wf* ( $\downarrow$ *gen-ssa-defs* = *d*, *gen-ssa-uses* = *u*, *gen-phis*  
 = *p*, *gen-var* = *v*))  
 = *gen-wf- $\alpha$ n gen-cfg-wf*  
 ⟨proof⟩

**lemma** *gen-inEdges-wf-extend* [simp]:

*gen-inEdges* (*gen-cfg-wf-extend gen-cfg-wf* ( $\downarrow$ *gen-ssa-defs* = *d*, *gen-ssa-uses* = *u*,  
*gen-phis* = *p*, *gen-var* = *v*))  
 = *gen-wf-inEdges gen-cfg-wf*  
 ⟨proof⟩

**lemma** *gen-Entry-wf-extend* [simp]:

*gen-Entry* (*gen-cfg-wf-extend gen-cfg-wf* ( $\downarrow$ *gen-ssa-defs* = *d*, *gen-ssa-uses* = *u*,  
*gen-phis* = *p*, *gen-var* = *v*))  
 = *gen-wf-Entry gen-cfg-wf*  
 ⟨proof⟩

**lemma** *gen-defs-wf-extend* [simp]:

*gen-defs* (*gen-cfg-wf-extend gen-cfg-wf* ( $\downarrow$ *gen-ssa-defs* = *d*, *gen-ssa-uses* = *u*,  
*gen-phis* = *p*, *gen-var* = *v*))

= *gen-wf-defs gen-cfg-wf*  
 ⟨*proof*⟩

**lemma** *gen-uses-wf-extend [simp]*:

*gen-uses (gen-cfg-wf-extend gen-cfg-wf (λgen-ssa-defs = d, gen-ssa-uses = u,  
 gen-phis = p, gen-var = v))*  
 = *gen-wf-uses gen-cfg-wf*  
 ⟨*proof*⟩

**lemma** *gen-ssa-defs-wf-extend [simp]*:

*gen-ssa-defs (gen-cfg-wf-extend gen-cfg-wf (λgen-ssa-defs = d, gen-ssa-uses = u,  
 gen-phis = p, gen-var = v))*  
 = *d*  
 ⟨*proof*⟩

**lemma** *gen-ssa-uses-wf-extend [simp]*:

*gen-ssa-uses (gen-cfg-wf-extend gen-cfg-wf (λgen-ssa-defs = d, gen-ssa-uses = u,  
 gen-phis = p, gen-var = v))*  
 = *u*  
 ⟨*proof*⟩

**lemma** *gen-phis-wf-extend [simp]*:

*gen-phis (gen-cfg-wf-extend gen-cfg-wf (λgen-ssa-defs = d, gen-ssa-uses = u,  
 gen-phis = p, gen-var = v))*  
 = *p*  
 ⟨*proof*⟩

**lemma** *gen-var-wf-extend [simp]*:

*gen-var (gen-cfg-wf-extend gen-cfg-wf (λgen-ssa-defs = d, gen-ssa-uses = u, gen-phis  
 = p, gen-var = v))*  
 = *v*  
 ⟨*proof*⟩

**lemma** *CFG-SSA-Transformed-codeI*:

**assumes** *CFG-SSA-Transformed αe αn invar inEdges Entry oldDefs oldUses*  
*defs (λg. lookup-multimap (uses g)) (λg. Mapping.lookup (phis g)) var*  
**and**  $\bigwedge g. \text{Mapping.keys } (uses\ g) \subseteq \text{set } (\alpha n\ g)$   
**shows** *CFG-SSA-Transformed-code αe αn invar inEdges Entry oldDefs oldUses*  
*defs uses phis var*  
 ⟨*proof*⟩

**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*)  
 ⟨*proof*⟩

**lift-definition** *gen-ssa-cfg-wf* :: ('node::linorder, 'var::linorder, 'edge) *gen-cfg-wf*  
 ⇒ ('node, 'var, 'edge, ('node,'var) *ssaVal*) *gen-ssa-cfg-wf*  
 is λ*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*  
 )  
 ⟨*proof*⟩

**declare** *uninst.defNode-code*[*abs-def, code*] *uninst.allVars-code*[*abs-def, code*] *uninst.allUses-def*[*abs-def, code*] *uninst.allDefs-def*[*abs-def, code*]  
*uninst.phiUses-code*[*abs-def, code*] *uninst.phi-def*[*abs-def, code*] *uninst.redundant-code-def*[*abs-def, code*]  
**declare** *uninst-code.uses'-code-def*[*abs-def, code*] *uninst-code.substNext-code-def*[*abs-def, code*] *uninst-code.substitution-code-def*[*abs-def, folded uninst-phi-def, code*]  
**declare** *uninst-code.phis'-code-def*[*folded uninst-code-substNext-def, code*] *uninst-code.step-code-def*[*folded uninst-code.uses'-code-def uninst-code.phis'-code-def, code*]  
*uninst-code.cond-code-def*[*folded uninst-redundant-def, code*]  
**declare** *gen-ssa-wf-notriv.substAll-efficient-def*  
 [*folded uninst-code-nodes-of-phis'-def uninst-code-nodes-of-uses'-def uninst-code-triv-phis'-def uninst-code-substitution-def uninst-code-step-code-def uninst-code-phis'-def uninst-code-uses'-def uninst-trivial-phis-def uninst-phiDefNodes-def uninst-useNodes-of-def uninst-phiNodes-of-def, code*]  
**declare** *keys-dom-lookup* [*symmetric, code-unfold*]

**definition** *map-keys-from-sparse* ≡ *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*: ( $\bigwedge a b. \llbracket a \in \text{set } (x \# xs); b \in \text{set } (x \# xs) \rrbracket \implies f a \circ f b = f b \circ f a$ )  
 ⇒  $\text{fold } f (x \# xs) = f x \circ (\text{fold } f xs)$   
 ⟨*proof*⟩

**lemma** *Union-of-code* [*code*]:  $\text{Union-of } f (RBT\text{-Set.Set } r) = RBT.\text{fold } (\lambda a -. (\cup (f a)) r \{\})$   
 ⟨*proof*⟩

**definition**[*code*]:  $\text{disjoint } xs \ ys = (xs \cap ys = \{\})$

**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  $\Rightarrow$  cmd  $\Rightarrow$  bool where
  sub-Skip: subcmd c Skip
| sub-Base: subcmd c c
| sub-Seq1: subcmd c1 c  $\Longrightarrow$  subcmd (c1;;c2) c
| sub-Seq2: subcmd c2 c  $\Longrightarrow$  subcmd (c1;;c2) c
| sub-If1: subcmd c1 c  $\Longrightarrow$  subcmd (if (b) c1 else c2) c
| sub-If2: subcmd c2 c  $\Longrightarrow$  subcmd (if (b) c1 else c2) c
| sub-While: subcmd c' c  $\Longrightarrow$  subcmd (while (b) c') c

fun maxVnameLen-aux :: expr  $\Rightarrow$  nat where
  maxVnameLen-aux (Val -) = 0
| maxVnameLen-aux (Var V) = length V
| maxVnameLen-aux (e1 « - » e2) = max (maxVnameLen-aux e1) (maxVnameLen-aux e2)

fun maxVnameLen :: cmd  $\Rightarrow$  nat where
  maxVnameLen Skip = 0
| maxVnameLen (V:=e) = max (length V) (maxVnameLen-aux e)
| maxVnameLen (c1;;c2) = max (maxVnameLen c1) (maxVnameLen c2)
| maxVnameLen (if (b) c1 else c2) = max (maxVnameLen c1) (max (maxVnameLen-aux b) (maxVnameLen c2))
| maxVnameLen (while (b) c) = max (maxVnameLen c) (maxVnameLen-aux b)

```

**definition**  $tempName :: cmd \Rightarrow vname$  **where**  $tempName\ c \equiv replicate\ (Suc\ (maxVnameLen\ c))\ (CHR\ 'a')$

**inductive**  $newname :: cmd \Rightarrow vname \Rightarrow bool$  **where**  
 $newname\ Skip\ V$   
 $| V \notin \{V'\} \cup rhs\text{-aux}\ e \Longrightarrow newname\ (V' := e)\ V$   
 $| \llbracket newname\ c1\ V; newname\ c2\ V \rrbracket \Longrightarrow newname\ (c1;;c2)\ V$   
 $| \llbracket newname\ c1\ V; newname\ c2\ V; V \notin rhs\text{-aux}\ b \rrbracket \Longrightarrow newname\ (if\ (b)\ c1\ else\ c2)\ V$   
 $| \llbracket newname\ c\ V; V \notin rhs\text{-aux}\ b \rrbracket \Longrightarrow newname\ (while\ (b)\ c)\ V$

**lemma**  $maxVnameLen\text{-aux}\text{-newname}$ :  $length\ V > maxVnameLen\text{-aux}\ e \Longrightarrow V \notin rhs\text{-aux}\ e$   
 $\langle proof \rangle$

**lemma**  $maxVnameLen\text{-newname}$ :  $length\ V > maxVnameLen\ c \Longrightarrow newname\ c\ V$   
 $\langle proof \rangle$

**lemma**  $tempname\text{-newname}$ [intro]:  $newname\ c\ (tempName\ c)$   
 $\langle proof \rangle$

**fun**  $transform\text{-aux} :: vname \Rightarrow cmd \Rightarrow cmd$  **where**  
 $transform\text{-aux}\ -\ Skip = Skip$   
 $| transform\text{-aux}\ V'\ (V := e) =$   
 $\quad (if\ V \in rhs\ (V := e)\ then\ V' := e;; V := Var\ V'$   
 $\quad \quad else\ V := e)$   
 $| transform\text{-aux}\ V'\ (c1;;c2) = transform\text{-aux}\ V'\ c1;; transform\text{-aux}\ V'\ c2$   
 $| transform\text{-aux}\ V'\ (if\ (b)\ c1\ else\ c2) =$   
 $\quad (if\ (b)\ transform\text{-aux}\ V'\ c1\ else\ transform\text{-aux}\ V'\ c2)$   
 $| transform\text{-aux}\ V'\ (while\ (b)\ c) = (while\ (b)\ transform\text{-aux}\ V'\ c)$

**abbreviation**  $transform :: cmd \Rightarrow cmd$  **where**  
 $transform\ c \equiv transform\text{-aux}\ (tempName\ c)\ c$

**fun**  $leftmostCmd :: cmd \Rightarrow cmd$  **where**  
 $leftmostCmd\ (c1;;c2) = leftmostCmd\ c1$   
 $| leftmostCmd\ c = c$

**lemma**  $leftmost\text{-lhs}$ [simp]:  $lhs\ (leftmostCmd\ c) = lhs\ c$   
 $\langle proof \rangle$

**lemma**  $leftmost\text{-rhs}$ [simp]:  $rhs\ (leftmostCmd\ c) = rhs\ c$   
 $\langle proof \rangle$

**lemma**  $leftmost\text{-subcmd}$ [intro]:  $subcmd\ c\ (leftmostCmd\ c)$   
 $\langle proof \rangle$

**lemma**  $leftmost\text{-labels}$ :  $labels\ c\ n\ c' \Longrightarrow subcmd\ c\ (leftmostCmd\ c')$

*<proof>*

**theorem** *transform-disjoint*:

**assumes** *subcmd* (*transform-aux temp c*) (*V:=e*) *newname c temp*

**shows**  $V \notin \text{rhs-aux } e$

*<proof>*

**lemma** *transform-disjoint'*: *subcmd* (*transform c*) (*leftmostCmd c'*)  $\implies \text{lhs } c' \cap \text{rhs } c' = \{\}$

*<proof>*

**corollary** *Defs-Uses-transform-disjoint [simp]*: *Defs* (*transform c*)  $n \cap \text{Uses}$  (*transform c*)  $n = \{\}$

*<proof>*

**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**

$(\text{-Entry-}) \leq x = \text{True}$

|  $(\text{- } n \text{-}) \leq x = (\text{case } x \text{ of}$

$(\text{-Entry-}) \implies \text{False}$

  |  $(\text{- } m \text{-}) \implies n \leq m$

  |  $(\text{-Exit-}) \implies \text{True}$ )

|  $(\text{-Exit-}) \leq x = (x = (\text{-Exit-}))$

**fun** *less-w-node* **where**

$(\text{-Entry-}) < x = (x \neq (\text{-Entry-}))$

|  $(\text{- } n \text{-}) < x = (\text{case } x \text{ of}$

$(\text{-Entry-}) \implies \text{False}$

  |  $(\text{- } m \text{-}) \implies n < m$

  |  $(\text{-Exit-}) \implies \text{True}$ )

|  $(\text{-Exit-}) < x = \text{False}$

**instance** *<proof>*

**end**

**instance** *w-node* :: *linorder* *<proof>*

**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))*  
 ⟨*proof*⟩  
  
**lemma** *uses-expr-finite*: *finite (rhs-aux e)*  
 ⟨*proof*⟩  
  
**lemma** *uses-cmd-finite*: *finite (rhs c)*  
 ⟨*proof*⟩  
  
**lemma** *defs-cmd-finite*: *finite (lhs c)*  
 ⟨*proof*⟩  
  
**lemma** *finite-labels'*: *finite {(l,c). labels prog l c}*  
 ⟨*proof*⟩  
  
**lemma** *finite-Defs* [*simp, intro!*]: *finite (Defs c n)*  
 ⟨*proof*⟩  
  
**lemma** *finite-Uses* [*simp, intro!*]: *finite (Uses c n)*  
 ⟨*proof*⟩  
  
**definition** *while-cfg-αe* *c* = *Collect (valid-edge (transform c))*  
**definition** *while-cfg-αn* *c* = *sorted-list-of-set (Collect (valid-node (transform c)))*  
**definition** *while-cfg-invar* *c* = *True*  
**definition** *while-cfg-inEdges'* *c t* = *(SOME ls. distinct ls ∧ set ls = {(sourcenode e, kind e) | e. valid-edge (transform c) e ∧ targetnode e = t})*  
**definition** *while-cfg-Entry* *c* = *(-Entry-)*  
**definition** *while-cfg-defs* *c* = *(Defs (transform c))((-Entry-) := {v. ∃ n. v ∈ Uses (transform c) n})*  
**definition** *while-cfg-uses* *c* = *Uses (transform c)*  
  
**abbreviation** *while-cfg-inEdges* *c t* ≡ *map (λ(f,d). (f,d,t)) (while-cfg-inEdges' c t)*  
  
**lemmas** *while-cfg-defs = while-cfg-αe-def while-cfg-αn-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-αe while-cfg-αn while-cfg-invar while-cfg-inEdges'*  
 ⟨*proof*⟩

**lemma** *right-total-const*: *right-total* ( $\lambda x y. x = c$ )  
 ⟨*proof*⟩

**lemma** *const-transfer*: *rel-fun* ( $\lambda x y. x = c$ ) (=) *f* ( $\lambda-. f c$ )  
 ⟨*proof*⟩

**interpretation** *while-ign*: *graph-path*  $\lambda-. \text{while-cfg-}\alpha e \text{ cmd}$   $\lambda-. \text{while-cfg-}\alpha n \text{ cmd}$   
 $\lambda-. \text{while-cfg-invar cmd}$   $\lambda-. \text{while-cfg-inEdges}' \text{ cmd}$   
 ⟨*proof*⟩

**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*  $m \implies \text{while.path2}$   
 $c n (n \# \text{map targetnode es}) m$   
 ⟨*proof*⟩

**lemma** *Uses-Entry [simp]*: *Uses*  $c (-\text{Entry}) = \{\}$   
 ⟨*proof*⟩

**lemma** *in-Uses-valid-node*:  $V \in \text{Uses } c n \implies \text{valid-node } c n$   
 ⟨*proof*⟩

**lemma** *while-cfg-CFG-wf-impl*:  
   *SSA-CFG.CFG-wf* ( $\lambda-. \text{gen-}\alpha e (\text{gen-while-cfg } \text{cmd})$ ) ( $\lambda-. \text{gen-}\alpha n (\text{gen-while-cfg}$   
    $\text{cmd})$ )  
     ( $\lambda-. \text{while-cfg-invar cmd}$ ) ( $\lambda-. \text{gen-inEdges}' (\text{gen-while-cfg } \text{cmd})$ )  
     ( $\lambda-. \text{gen-Entry } (\text{gen-while-cfg } \text{cmd})$ ) ( $\lambda-. \text{gen-defs } (\text{gen-while-cfg } \text{cmd})$ )  
     ( $\lambda-. \text{gen-uses } (\text{gen-while-cfg } \text{cmd})$ )  
 ⟨*proof*⟩

**lift-definition** *gen-while-cfg-wf* ::  $\text{cmd} \Rightarrow (w\text{-node}, v\text{name}, \text{state } \text{edge-kind}) \text{ gen-cfg-wf}$   
**is** *gen-while-cfg*  
 ⟨*proof*⟩

**definition** *build-ssa*  $\text{cmd} = \text{gen-ssa-wf-notriv-substAll} (\text{gen-ssa-cfg-wf } (\text{gen-while-cfg-wf}$   
 $\text{cmd}))$

**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.