

Verified Construction of Static Single Assignment Form

Sebastian Ullrich Denis Lohner

May 14, 2024

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

1	Prelude	2
1.1	Miscellaneous Lemmata	2
1.2	Serial Relations	6
1.3	Mapping Extensions	7
2	SSA Representation	9
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
3	Minimality	33
4	SSA Construction	36
4.1	CFG to SSA CFG	36
4.2	Inductive Removal of Trivial Phi Functions	44

5	Proof of Semantic Equivalence	51
6	Code Generation	52
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  $\text{prefix } (as@[x]) \ xs$  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 $ys \ \text{where } x \in ys \ ys \in \text{set } xss \cup \{xs\}$
<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** $(x,y') \in \text{set } (\text{zip } xs \ ys)$ $f \ y' = y$
 $\langle \text{proof} \rangle$

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

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

lemma *the1-list*: $\exists ! x \in \text{set } xs. P \ x \implies (\text{THE } x. x \in \text{set } xs \wedge P \ x) = \text{hd } (\text{filter } P \ xs)$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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 ∈ 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 $(a, iterated-serial-on\ A\ r\ a\ n) \in r \overset{\sim}{\sim} n$

⟨proof⟩

lemma *trancl-powerI*: $a \in R \overset{\sim}{\sim} n \implies n > 0 \implies a \in R^+$

⟨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 bool) \Rightarrow ('a,'b) *mapping* \Rightarrow ('a,'b) *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) *mapping* \Rightarrow ('b,'c) *mapping*

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

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

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

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

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

lift-definition *mapping-add* :: ('a, 'b) *mapping* \Rightarrow ('a, 'b) *mapping* \Rightarrow ('a, 'b) *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 *Mapping.lookup* *m* = *Mapping.lookup* *m'* *<proof>*

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

map-of (*map* ($\lambda k. (k, f k)$) *xs*) *x* = (*if* (*x* \in *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 (*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 o 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* α *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 $\lambda g. \text{foldri } (\alpha n \ g) +$
graph-pred-it-defs $\lambda g \ n. \text{foldri } (\text{inEdges}' \ g \ n)$
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* ($\lambda(f,d). (f,d,n)$) (*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* ← $\alpha n \ g$, *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* ($\alpha n \ g$); *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 $\alpha\ g \equiv (\text{nodes} = set\ (\alpha n\ g),\ edges = \alpha e\ g)$
end

locale *graph-path* =

graph-path-base $\alpha e\ \alpha n\ invar\ inEdges'$ +

graph $\alpha\ invar$ +

ni: *graph-nodes-it* $\alpha\ invar\ \lambda g.\ foldr\ (\alpha n\ g) +$

pi: *graph-pred-it* $\alpha\ invar\ \lambda g\ n.\ foldr\ (inEdges'\ g\ n)$

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 *αn-correct*: $invar\ g \Longrightarrow set\ (\alpha n\ g) \supseteq getFrom\ ' \alpha e\ g \cup getTo\ ' \alpha e\ g$
<proof>

lemma *αn-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-αnI1* [*intro*]: $\llbracket invar\ g; x \in getFrom\ ' \alpha e\ g \rrbracket \Longrightarrow x \in set\ (\alpha n\ g)$
<proof>

lemma *in-set-αnI2* [*intro*]: $\llbracket invar\ g; x \in getTo\ ' \alpha e\ g \rrbracket \Longrightarrow x \in set\ (\alpha n\ g)$
<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- αn* [*intro*]: $\llbracket path\ g\ ns; n \in set\ ns \rrbracket \implies n \in set\ (\alpha n\ g)$
<proof>

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

lemma *path2-tl-in- αn* [*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 *αn -in- $\alpha n E$* [*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-elim-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** $g \vdash n - ns' \rightarrow m \ \text{distinct } ns' \ \text{set } ns' \subseteq \text{set } ns \ \text{length } ns' \leq$
length ns
(*proof*)

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

$ns' \text{ set } ns' \subseteq \text{set } ns \ m \notin \text{set } ns'$
 ⟨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** $g \vdash n - ns' \rightarrow n \ n' \in \text{set } ns' \ n \notin \text{set } (tl \ (butlast \ ns'))$ *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** $g \vdash n - ns' \rightarrow m' \ P \ m' \ \forall x \in \text{set } (butlast \ ns'). \ \neg P \ x$ *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 $2: \text{prefix } (ns' @ [m']) \ ns$

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

⟨proof⟩

lemma *path2-strict-prefix-ex*:

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

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

⟨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]:
   $[[n \in set\ (\alpha n\ g); invar\ g]] \implies \exists ns. g \vdash Entry\ g\ ns \rightarrow n$ 
begin
  lemma Entry-dominates[simp,intro]:  $[[invar\ g; n \in set\ (\alpha n\ g)]] \implies dominates\ g\ (Entry\ g)\ n$ 
  <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  $ns = [Entry\ g]$ 

```

<proof>

lemma *simple-Entry-path*:

assumes *invar g n ∈ set (αn g)*

obtains *ns where g ⊢ Entry g-ns→n and n ∉ set (butlast ns)*

<proof>

lemma *dominatesI [intro]*:

$\llbracket m \in \text{set } (\alpha n g); \bigwedge ns. \llbracket g \vdash \text{Entry } g-ns \rightarrow m \rrbracket \implies n \in \text{set } ns \rrbracket \implies \text{dominates } g$
n m

<proof>

lemma *dominatesE*:

assumes *dominates g n m*

obtains *m ∈ set (αn g) and $\bigwedge ns. g \vdash \text{Entry } g-ns \rightarrow m \implies n \in \text{set } ns$*

<proof>

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

lemma*[simp]*:

assumes *dominates g n m* **and***[simp]*: *invar g*

shows *n ∈ set (αn g)*

<proof>

lemma *strict-domE[elim]*:

assumes *strict-dom g n m*

obtains *m ∈ set (αn g) and $\bigwedge ns. g \vdash \text{Entry } g-ns \rightarrow m \implies n \in \text{set } (\text{butlast } ns)$*

<proof>

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

<proof>

lemma *dominates-trans*:

assumes *invar g*

assumes *part1: dominates g n n'*

assumes *part2: dominates g n' n''*

shows *dominates g n n''*

<proof>

lemma *dominates-antisymm*:

assumes *invar g*

assumes *dom1: dominates g n n'*

assumes *dom2: dominates g n' n*

shows *n = n'*

<proof>

lemma *dominates-unsnoc*:

assumes *[simp]: invar g* **and** *dominates g n m m' ∈ set (predecessors g m) n*

$\neq m$

shows *dominates* $g\ n\ m'$

<proof>

lemma *dominates-unsnoc'*:

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

shows *dominates* $g\ n\ m'$

<proof>

lemma *dominates-path*:

assumes *dominates* $g\ n\ m$ **and** [*simp*]: *invar* g

obtains ns **where** $g \vdash n - ns \rightarrow m$

<proof>

lemma *dominates-antitrans*:

assumes [*simp*]: *invar* g **and** *dominates* $g\ n_1\ m$ *dominates* $g\ n_2\ m$

obtains (1) *dominates* $g\ n_1\ n_2$

| (2) *dominates* $g\ n_2\ n_1$

<proof>

lemma *dominates-extend*:

assumes *dominates* $g\ n\ m$

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

shows *dominates* $g\ n\ m'$

<proof>

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

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

definition *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 *idom* :: $'g \Rightarrow 'node \Rightarrow 'node$ **where**

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

lemma *idom-ex*:

assumes [*simp*]: *invar* $g\ n \in \text{set } (\alpha n\ g)\ n \neq \text{Entry } g$

shows $\exists! m. \text{isIdom } g\ n\ m$

<proof>

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

<proof>

lemma *dominates-mid*:

assumes *dominates* $g\ n\ x$ *dominates* $g\ x\ m\ g \vdash n - ns \rightarrow m$ **and** [*simp*]: *invar* g

shows $x \in \text{set } ns$

<proof>

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

$shortestPath\ g\ n \equiv (LEAST\ l.\ \exists\ ns.\ length\ ns = l \wedge g \vdash Entry\ g - ns \rightarrow n)$

lemma *shortestPath-ex*:

assumes $n \in set\ (\alpha n\ g)$ *invar* g

obtains ns **where** $g \vdash Entry\ g - ns \rightarrow n$ *distinct* ns $length\ ns = shortestPath\ g\ n$
<proof>

lemma_[simp]: $\llbracket n \in set\ (\alpha n\ g);\ invar\ g \rrbracket \implies shortestPath\ g\ n \neq 0$
<proof>

lemma *shortestPath-upper-bound*:

assumes $n \in set\ (\alpha n\ g)$ *invar* g

shows $shortestPath\ g\ n \leq length\ (\alpha n\ g)$
<proof>

lemma *shortestPath-predecessor*:

assumes $n \in set\ (\alpha n\ g) - \{Entry\ g\}$ **and**_[simp]: *invar* g

obtains n' **where** $Suc\ (shortestPath\ g\ n') = shortestPath\ g\ n$ $n' \in set$
(predecessors $g\ n)$
<proof>

lemma *successor-in- αn* _[simp]:

assumes $predecessors\ g\ n \neq []$ **and**_[simp]: *invar* g

shows $n \in set\ (\alpha n\ g)$
<proof>

lemma *shortestPath-single-predecessor*:

assumes $predecessors\ g\ n = [m]$ **and**_[simp]: *invar* g

shows $shortestPath\ g\ m < shortestPath\ g\ n$
<proof>

lemma *strict-dom-shortestPath-order*:

assumes *strict-dom* $g\ n\ m$ $m \in set\ (\alpha n\ g)$ *invar* g

shows $shortestPath\ g\ n < shortestPath\ g\ m$
<proof>

lemma *dominates-shortestPath-order*:

assumes *dominates* $g\ n\ m$ $m \in set\ (\alpha n\ g)$ *invar* g

shows $shortestPath\ g\ n \leq shortestPath\ g\ m$
<proof>

lemma *strict-dom-trans*:

assumes_[simp]: *invar* g

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

shows *strict-dom* $g\ n\ m'$
<proof>

inductive *EntryPath* :: ' $g \Rightarrow 'node\ list \Rightarrow bool$ **where**

*EntryPath-triv*_[simp]: *EntryPath* $g\ [n]$

| *EntryPath-snoc*[intro]: $EntryPath\ g\ ns \implies shortestPath\ g\ m = Suc\ (shortestPath\ g\ (last\ ns)) \implies EntryPath\ g\ (ns@[m])$

lemma[simp]:
assumes $EntryPath\ g\ ns\ prefix\ ns'\ ns\ ns' \neq []$
shows $EntryPath\ g\ ns'$
 <proof>

lemma *EntryPath-suffix*:
assumes $EntryPath\ g\ ns\ suffix\ ns'\ ns\ ns' \neq []$
shows $EntryPath\ g\ ns'$
 <proof>

lemma *EntryPath-butlast-less-last*:
assumes $EntryPath\ g\ ns\ z \in set\ (butlast\ ns)$
shows $shortestPath\ g\ z < shortestPath\ g\ (last\ ns)$
 <proof>

lemma *EntryPath-distinct*:
assumes $EntryPath\ g\ ns$
shows $distinct\ ns$
 <proof>

lemma *Entry-reachesE*:
assumes $n \in set\ (\alpha n\ g)$ **and**[simp]: $invar\ g$
obtains ns **where** $g \vdash Entry\ g - ns \rightarrow n$ $EntryPath\ g\ ns$
 <proof>

end

end

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

2.3 CFG

locale *CFG-base* = *graph-Entry-base* $\alpha e\ \alpha n\ invar\ inEdges'\ Entry$
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**

$Entry :: 'g \Rightarrow 'node +$

fixes $defs :: 'g \Rightarrow 'node \Rightarrow 'var::linorder\ set$

fixes $uses :: 'g \Rightarrow 'node \Rightarrow 'var\ set$

begin

definition $vars\ g \equiv fold\ (\cup)\ (map\ (uses\ g)\ (\alpha n\ g))\ \{\}$

definition $defAss' :: 'g \Rightarrow 'node \Rightarrow 'var \Rightarrow bool$ **where**
 $defAss' g m v \leftrightarrow (\forall ns. g \vdash Entry\ g - ns \rightarrow m \longrightarrow (\exists n \in set\ ns. v \in defs\ g\ n))$

definition $defAss'Uses :: 'g \Rightarrow bool$ **where**
 $defAss'Uses\ g \equiv \forall m \in set\ (\alpha n\ g). \forall v \in uses\ g\ m. defAss'\ g\ m\ v$

end

locale $CFG = CFG\text{-}base\ \alpha e\ \alpha n\ invar\ inEdges'\ Entry\ defs\ uses$
 $+ graph\text{-}Entry\ \alpha e\ \alpha n\ invar\ inEdges'\ Entry$

for
 $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)\ set$ **and**
 $\alpha n :: 'g \Rightarrow 'node\ list$ **and**
 $invar :: 'g \Rightarrow bool$ **and**
 $inEdges' :: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)\ list$ **and**
 $Entry :: 'g \Rightarrow 'node$ **and**
 $defs :: 'g \Rightarrow 'node \Rightarrow 'var::linorder\ set$ **and**
 $uses :: 'g \Rightarrow 'node \Rightarrow 'var\ set +$

assumes $defs\text{-}uses\text{-}disjoint: n \in set\ (\alpha n\ g) \implies defs\ g\ n \cap uses\ g\ n = \{\}$
assumes $defs\text{-}finite[simp]: finite\ (defs\ g\ n)$
assumes $uses\text{-}in\text{-}\alpha n: v \in uses\ g\ n \implies n \in set\ (\alpha n\ g)$
assumes $uses\text{-}finite[simp, intro!]: finite\ (uses\ g\ n)$
assumes $invar[intro!]: invar\ g$

begin
lemma $vars\text{-}finite[simp]: finite\ (vars\ g)$
 $\langle proof \rangle$

lemma $Entry\text{-}no\text{-}predecessor[simp]: predecessors\ g\ (Entry\ g) = []$
 $\langle proof \rangle$

lemma $uses\text{-}in\text{-}vars[elim, simp]: v \in uses\ g\ n \implies v \in vars\ g$
 $\langle proof \rangle$

lemma $varsE:$
assumes $v \in vars\ g$
obtains n **where** $n \in set\ (\alpha n\ g) \ v \in uses\ g\ n$
 $\langle proof \rangle$

lemma $defs\text{-}uses\text{-}disjoint'[simp]: n \in set\ (\alpha n\ g) \implies v \in defs\ g\ n \implies v \in uses\ g\ n \implies False$
 $\langle proof \rangle$

end

context CFG

begin
lemma $defAss'E:$
assumes $defAss'\ g\ m\ v\ g \vdash Entry\ g - ns \rightarrow m$
obtains n **where** $n \in set\ ns\ v \in defs\ g\ n$
 $\langle proof \rangle$

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

lemma $defAss'-extend$:

assumes $defAss' g m v$

assumes $g \vdash n - ns \rightarrow m \ \forall n \in set (tl ns). v \notin defs g n$

shows $defAss' g n v$

<proof>

end

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

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

for

$\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node) \ set \ \mathbf{and}$

$\alpha n :: 'g \Rightarrow 'node \ list \ \mathbf{and}$

$invar :: 'g \Rightarrow bool \ \mathbf{and}$

$inEdges' :: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD) \ list \ \mathbf{and}$

$Entry :: 'g \Rightarrow 'node \ \mathbf{and}$

$defs :: 'g \Rightarrow 'node \Rightarrow 'var::linorder \ set \ \mathbf{and}$

$uses :: 'g \Rightarrow 'node \Rightarrow 'var \ set \ +$

assumes $def-ass-uses: \ \forall m \in set (\alpha n \ g). \ \forall v \in uses \ g \ m. \ defAss' \ g \ m \ v$

2.4 SSA CFG

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

declare $in-set-zipE$ [elim]

declare $zip-same$ [simp]

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

for

$\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node) \ set \ \mathbf{and}$

$\alpha n :: 'g \Rightarrow 'node \ list \ \mathbf{and}$

$invar :: 'g \Rightarrow bool \ \mathbf{and}$

$inEdges' :: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD) \ list \ \mathbf{and}$

$Entry :: 'g \Rightarrow 'node \ \mathbf{and}$

$defs :: 'g \Rightarrow 'node \Rightarrow 'val::linorder \ set \ \mathbf{and}$

$uses :: 'g \Rightarrow 'node \Rightarrow 'val \ set \ +$

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

begin

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

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

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

$\bigcup n' \in set (successors \ g \ n). \bigcup v' \in phiDefs \ g \ n'. \ snd \ ' \ Set.filter \ (\lambda(n'', v). n'' = n) (set (zip (predecessors \ g \ n') (the (phis \ g \ (n', v')))))$

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

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

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

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

lemmas *CFG-SSA-defs* = *phiDefs-def allDefs-def phiUses-def allUses-def all-Vars-def defAss-def*
end

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

for

$\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)\ set\ \mathbf{and}$

$\alpha n :: 'g \Rightarrow 'node\ list\ \mathbf{and}$

invar :: $'g \Rightarrow bool\ \mathbf{and}$

inEdges' :: $'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)\ list\ \mathbf{and}$

Entry:: $'g \Rightarrow 'node\ \mathbf{and}$

defs :: $'g \Rightarrow 'node \Rightarrow 'val::linorder\ set\ \mathbf{and}$

uses :: $'g \Rightarrow 'node \Rightarrow 'val\ set\ \mathbf{and}$

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

assumes *phis-finite*: *finite* (*dom* (*phis g*))

assumes *phis-in- αn* : *phis g* (*n,v*) = *Some vs* $\implies n \in set\ (\alpha n\ g)$

assumes *phis-wf*:

$phis\ g\ (n,v) = Some\ args \implies length\ (predecessors\ g\ n) = length\ args$

assumes *simpleDefs-phiDefs-disjoint*:

$n \in set\ (\alpha n\ g) \implies defs\ g\ n \cap phiDefs\ g\ n = \{\}$

assumes *allDefs-disjoint*:

$\llbracket n \in set\ (\alpha n\ g); m \in set\ (\alpha n\ g); n \neq m \rrbracket \implies allDefs\ g\ n \cap allDefs\ g\ m = \{\}$

begin

lemma *phis-disj*:

assumes *phis g* (*n,v*) = *Some vs*

and *phis g* (*n',v*) = *Some vs'*

shows $n = n'\ \mathbf{and}\ vs = vs'$

<proof>

lemma *allDefs-disjoint'*: $\llbracket n \in set\ (\alpha n\ g); m \in set\ (\alpha n\ g); v \in allDefs\ g\ n; v \in allDefs\ g\ m \rrbracket \implies n = m$

<proof>

lemma *phiUsesI*:

assumes $n' \in set\ (\alpha n\ g)\ phis\ g\ (n',v') = Some\ vs\ (n,v) \in set\ (zip\ (predecessors\ g\ n')\ vs)$

shows $v \in phiUses\ g\ n$

<proof>

lemma *phiUsesE*:

assumes $v \in phiUses\ g\ n$

obtains $n'\ v'\ vs\ \mathbf{where}\ n' \in set\ (successors\ g\ n)\ (n,v) \in set\ (zip\ (predecessors\ g\ n')\ vs)\ phis\ g\ (n',v') = Some\ vs$

<proof>

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 $g \vdash n - ns \rightarrow m \ \forall n \in \text{set } (tl \ ns). \ v \notin \text{allDefs } g \ n$
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* αn *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**
 $\text{invar} :: 'g \Rightarrow \text{bool}$ **and**
 $\text{inEdges}' :: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD) \text{ list}$ **and**
 $\text{Entry} :: 'g \Rightarrow 'node$ **and**
 $\text{defs} :: 'g \Rightarrow 'node \Rightarrow 'val::\text{linorder} \text{ set}$ **and**
 $\text{uses} :: 'g \Rightarrow 'node \Rightarrow 'val \text{ set}$ **and**
 $\text{phis} :: 'g \Rightarrow ('node, 'val) \text{ phis}$

begin

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

definition *defNode* :: 'g ⇒ '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**

$phis :: 'g \Rightarrow ('node, 'val) phis +$
assumes $allUses-def-ass: \llbracket v \in allUses\ g\ n; n \in set\ (\alpha n\ g) \rrbracket \Longrightarrow defAss\ g\ n\ v$
assumes $Entry-no-phis[simp]: phis\ g\ (Entry\ g, v) = None$
begin
lemma $allVars-in-allDefs: v \in allVars\ g \Longrightarrow \exists n \in set\ (\alpha n\ g). v \in allDefs\ g\ n$
 $\langle proof \rangle$

lemma $phiDefs-Entry-empty[simp]: phiDefs\ g\ (Entry\ g) = \{\}$
 $\langle proof \rangle$

lemma $phi-Entry-empty[simp]: defNode\ g\ v = Entry\ g \Longrightarrow phi\ g\ v = None$
 $\langle proof \rangle$

lemma $defNode-ex1:$
assumes $v \in allVars\ g$
shows $\exists! n. n \in set\ (\alpha n\ g) \wedge v \in allDefs\ g\ n$
 $\langle proof \rangle$

lemma $defNode-def: v \in allVars\ g \Longrightarrow defNode\ g\ v = (THE\ n. n \in set\ (\alpha n\ g) \wedge v \in allDefs\ g\ n)$
 $\langle proof \rangle$

lemma $defNode[simp]:$
assumes $v \in allVars\ g$
shows $(defNode\ g\ v) \in set\ (\alpha n\ g) \wedge v \in allDefs\ g\ (defNode\ g\ v)$
 $\langle proof \rangle$

lemma $defNode-eq[intro]:$
assumes $n \in set\ (\alpha n\ g) \wedge v \in allDefs\ g\ n$
shows $defNode\ g\ v = n$
 $\langle proof \rangle$

lemma $defNode-cases[consumes 1]:$
assumes $v \in allVars\ g$
obtains $(simpleDef)\ v \in defs\ g\ (defNode\ g\ v)$
 $\quad | (phi)\ \quad phi\ g\ v \neq None$
 $\langle proof \rangle$

lemma $phi-phiDefs[simp]: phi\ g\ v = Some\ vs \Longrightarrow v \in phiDefs\ g\ (defNode\ g\ v)$
 $\langle proof \rangle$

lemma $simpleDef-not-phi:$
assumes $n \in set\ (\alpha n\ g) \wedge v \in defs\ g\ n$
shows $phi\ g\ v = None$
 $\langle proof \rangle$

lemma $phi-wf: phi\ g\ v = Some\ vs \Longrightarrow length\ (predecessors\ g\ (defNode\ g\ v)) = length\ vs$
 $\langle proof \rangle$

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 $g \vdash \text{defNode } g \ v \text{-ns} \rightarrow n$ $\text{defNode } g \ v \notin \text{set } (\text{tl } ns)$ $v \in \text{allUses } g \ n \ n' \in \text{set } ns$

shows *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 *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** $g \vdash \text{defNode } g \ q \text{-ns} \rightarrow m$ $\text{EntryPath } g \ ns \ q \in \text{phiUses } g \ m$

$m \in \text{set } (\text{predecessors } g \text{ (defNode } g \text{ } p))$
 ⟨proof⟩

lemma *phiArg-path-ex*:

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

lemma *phiArg-tranclp-path-ex*:

assumes $r^{++} \text{ } p \text{ } q \text{ } p \in \text{allVars } g$ **and**[*simp*]: $\bigwedge p \text{ } q. r \text{ } p \text{ } q \implies \text{phiArg } g \text{ } p \text{ } q$
obtains ns **where** $g \vdash \text{defNode } g \text{ } q - ns \rightarrow \text{defNode } g \text{ } p$ $\text{length } ns > 1$
 $\forall n \in \text{set } (\text{butlast } ns). \exists p \text{ } q \text{ } m \text{ } ns'. r \text{ } p \text{ } q \wedge g \vdash \text{defNode } g \text{ } q - ns' \rightarrow m \wedge (\text{defNode } g \text{ } q) \notin \text{set } (\text{tl } ns') \wedge q \in \text{phiUses } g \text{ } m \wedge m \in \text{set } (\text{predecessors } g \text{ (defNode } g \text{ } p)) \wedge$
 $n \in \text{set } ns' \wedge \text{set } ns' \subseteq \text{set } ns \wedge \text{defNode } g \text{ } p \in \text{set } ns$
 ⟨proof⟩

lemma *non-dominated-predecessor*:

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

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

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

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

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

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

⟨proof⟩

lemma *phi-no-closed-loop*:

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

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

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

declare *phiArg-def* [*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 inEdges'*
Entry oldDefs oldUses defs uses phis var
+ *old: CFG-wf* $\alpha e \alpha n$ *invar inEdges' Entry oldDefs oldUses* + *CFG-SSA-wf* αe
 αn *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*: $n \in set\ (\alpha n\ g) \implies oldUses\ g\ n = var\ g\ 'uses\ g\ n$
assumes *conventional*:

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

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

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

begin

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

<proof>

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

<proof>

lemma *phiArg-same-var*: $\text{phiArg } g \ p \ q \implies \text{var } g \ q = \text{var } g \ p$
<proof>

lemma *oldDef-defAss*:

assumes $v \in \text{allUses } g \ n \ g \vdash \text{Entry } g \text{-ns} \rightarrow n$

obtains m **where** $m \in \text{set } ns \ \text{var } g \ v \in \text{oldDefs } g \ m$

<proof>

lemma *allDef-path-from-simpleDef*:

assumes*[simp]*: $v \in \text{allVars } g$

obtains $n \ ns$ **where** $g \vdash n \text{-ns} \rightarrow \text{defNode } g \ v \ \text{old.EntryPath } g \ ns \ \text{var } g \ v \in \text{oldDefs } g \ n$

<proof>

lemma *defNode-var-disjoint*:

assumes $p \in \text{allVars } g \ q \in \text{allVars } g \ p \neq q \ \text{defNode } g \ p = \text{defNode } g \ q$

shows $\text{var } g \ p \neq \text{var } g \ q$

<proof>

lemma *phiArg-distinct-nodes*:

assumes $\text{phiArg } g \ p \ q \ p \neq q$ **and***[simp]*: $p \in \text{allVars } g$

shows $\text{defNode } g \ p \neq \text{defNode } g \ q$

<proof>

lemma *phiArgs-def-distinct*:

assumes $\text{phiArg } g \ p \ q \ \text{phiArg } g \ p \ r \ q \neq r \ p \in \text{allVars } g$

shows $\text{defNode } g \ q \neq \text{defNode } g \ r$

<proof>

lemma *defNode-not-on-defUse-path*:

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

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

shows $\text{defNode } g \ q \notin \text{set } ns$

<proof>

lemma *defUse-paths-disjoint*:

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

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

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

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

<proof>

lemma *oldDefsI*: $v \in \text{defs } g \ n \implies \text{var } g \ v \in \text{oldDefs } g \ n$ *<proof>*

lemma *simpleDefs-phiDefs-var-disjoint*:

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

shows $\text{var } g \ v \notin \text{oldDefs } g \ n$

<proof>

lemma *liveVal-use-path*:

assumes *liveVal g v*

obtains *ns m* **where** $g \vdash \text{defNode } g \ v - ns \rightarrow m \ \text{var } g \ v \in \text{oldUses } g \ m$

$\wedge x. x \in \text{set } (tl \ ns) \implies \text{var } g \ v \notin \text{oldDefs } g \ x$

<proof>

end

end

3 Minimality

We show that every reducible CFG without trivial ϕ 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* $\equiv g \vdash x - xs \rightarrow z \wedge g \vdash y - ys \rightarrow z \wedge \text{length } xs > 1 \wedge \text{length } ys > 1 \wedge x \neq y \wedge$

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

Simplified definition

definition *pathsConverge' g x xs y ys z* $\equiv g \vdash x - xs \rightarrow z \wedge g \vdash y - ys \rightarrow z \wedge \text{length } xs > 1 \wedge \text{length } ys > 1 \wedge x \neq y \wedge$

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

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

<proof>

lemma *pathsConvergeI*:

assumes $g \vdash x - xs \rightarrow z \ g \vdash y - ys \rightarrow z \ \text{length } xs > 1 \ \text{length } ys > 1 \ \text{set } (\text{butlast } xs) \cap \text{set } (\text{butlast } ys) = \{\}$

shows $\text{pathsConverge } g \ x \ xs \ y \ ys \ z$

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

context *CFG-SSA-Transformed*

begin

A ϕ 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 ϕ 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 ϕ 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 $necessaryPhi\ g\ (var\ g\ v)\ n\ g \vdash n - ns \rightarrow m\ v \in allUses\ g\ m \wedge x.\ x \in set\ (tl\ ns) \implies v \notin allDefs\ g\ x \wedge v \notin defs\ g\ n$

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 ϕ 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 ϕ 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 ϕ 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 $phiCount\ g = card\ ((\lambda(n,v). (n, var\ g\ v))\ ' dom\ (phis\ g))$

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 $\text{length } (\text{predecessors } g \ n) \neq 1$
 $\langle \text{proof} \rangle$

abbreviation *unvisitedPath* :: $'node \ list \Rightarrow 'node \ list \Rightarrow \text{bool}$ **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 $\text{unvisitedPath } (\text{removeAll } (\text{last } ns) \ un) \ (\text{butlast } ns)$
 $\langle \text{proof} \rangle$

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 $n \in \text{phiDefNodes-aux } g \ v \ un \ m$
 $\langle \text{proof} \rangle$

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

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

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

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 $n \in \text{phiDefNodes } g \ v$
 $\langle \text{proof} \rangle$

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

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

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

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

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* $\alpha e \ \alpha n$ *invar inEdges' Entry defs' uses' 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 $g \vdash n - ns \rightarrow n$ *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 $\text{var} \ g \ v' = \text{var} \ g \ v$

<proof>

lemma *allUse-simpleUse*:

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

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

<proof>

lemma *defs'*: $v \in \text{defs}' \ g \ n \longleftrightarrow \text{var } g \ v \in \text{defs } g \ n \wedge \text{defKind } v = \text{SimpleDef} \wedge \text{defNode } v = n$

<proof>

lemma *use-implies-allDef*:

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

shows $v \in \text{braun-ssa.allDefs } g \ (\text{defNode } v)$

<proof>

lemma *allUse-defNode-in- αn [simp]*:

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

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

<proof>

lemma *allUse-implies-allDef*:

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

shows $v \in \text{braun-ssa.allDefs } g \ (\text{defNode } v)$

<proof>

lemma *conventional*:

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

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

shows $\text{var } g \ v' \neq \text{var } g \ v$

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

```

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

locale CFG-Construct-wf = CFG-Construct  $\alpha e \ \alpha n \ \text{invar} \ \text{inEdges}' \ \text{Entry} \ \text{defs} \ \text{uses}$ 
+ CFG-wf  $\alpha e \ \alpha n \ \text{invar} \ \text{inEdges}' \ \text{Entry} \ \text{defs} \ \text{uses}$ 
for
   $\alpha e :: 'g \Rightarrow ('node::\text{linorder} \times 'edgeD \times 'node) \ \text{set} \ \mathbf{and}$ 
   $\alpha n :: 'g \Rightarrow 'node \ \text{list} \ \mathbf{and}$ 
   $\text{invar} :: 'g \Rightarrow \text{bool} \ \mathbf{and}$ 
   $\text{inEdges}' :: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD) \ \text{list} \ \mathbf{and}$ 
   $\text{Entry} :: 'g \Rightarrow 'node \ \mathbf{and}$ 
   $\text{defs} :: 'g \Rightarrow 'node \Rightarrow 'var::\text{linorder} \ \text{set} \ \mathbf{and}$ 
   $\text{uses} :: 'g \Rightarrow 'node \Rightarrow 'var \ \text{set}$ 
begin
  lemma def-ass-allUses-aux:
    assumes  $g \vdash \text{Entry } g \text{--ns} \rightarrow n$ 
    shows  $\text{lookupDefNode } g \ n \ (\text{var } g \ v) \in \text{set } ns$ 
     $\langle \text{proof} \rangle$ 

  lemma def-ass-allUses:
    assumes  $v \in \text{braun-ssa.allUses } g \ n \ n \in \text{set } (\alpha n \ g)$ 
    shows  $\text{braun-ssa.defAss } g \ n \ v$ 
     $\langle \text{proof} \rangle$ 

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

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

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

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

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

  lemma PhiDef-defNode-eq:

```

```

assumes  $n \in \text{set } (\alpha n \ g) \ n \in \text{phiDefNodes } g \ v \ v \in \text{vars } g$ 
shows  $\text{braun-ssa.defNode } g \ (v, n, \text{PhiDef}) = n$ 
<proof>

lemma phiDefNodes-aux-pruned-aux:
assumes  $n \in \text{phiDefNodes-aux } g \ v \ (\alpha n \ g) \ nUse \ v \in \text{uses } g \ nUse \ g \vdash n - ns \rightarrow m$ 
 $g \vdash m - ms \rightarrow nUse \ \text{braun-ssa.liveVal } g \ (\text{lookupDef } g \ m \ v) \ \forall n \in \text{set } (ns @ ms). \ v \notin$ 
 $\text{defs } g \ n$ 
shows  $\text{braun-ssa.liveVal } g \ (v, n, \text{PhiDef})$ 
<proof>

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  $\text{braun-ssa.liveVal } g \ (v, m, \text{PhiDef})$ 
<proof>

theorem phis'-pruned:  $\text{braun-ssa.pruned } g$ 
<proof>

declare var-def [simp del]

declare no-disjoint-cycle [simp del]
declare lookupDef-looksUp [simp del]

declare lookupDef.simps [code]
declare phiDefNodes-aux.simps [code]
declare phiDefNodes-def [code]
declare defs'-def [code]
declare uses'-def [code]
declare phis'-def [code]
declare predecessors-def [code]
end

end

```

4.2 Inductive Removal of Trivial Phi Functions

```

theory Construct-SSA-notriv
imports SSA-CFG Minimality HOL-Library.While-Combinator
begin

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

```

```

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*)

<proof>

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

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

<proof>

lemma *trivialPhi-strict-dom*:

assumes *[simp]*: $v \in \text{allVars } g$ **and** *triv*: *isTrivialPhi g v v'*

shows *strict-def-dom g v' v*

<proof>

lemma *isTrivialPhi-asymmetric*:

assumes *isTrivialPhi g a b*

and *isTrivialPhi g b a*

shows *False*

<proof>

lemma *substitution[intro]*: *redundant g* \implies *isTrivialPhi g* (*chooseNext g*) (*substitution g*)

<proof>

lemma *trivialPhi-in-allVars[simp]*:

assumes *isTrivialPhi g v v'* **and** *[simp]*: $v \in \text{allVars } g$

shows $v' \in \text{allVars } g$

<proof>

lemma *substitution-in-allVars[simp]*:

assumes *redundant g*

shows *substitution g* \in *allVars g*

<proof>

lemma *defs-uses-disjoint-inv*:

assumes *[simp]*: $n \in \text{set } (\alpha n \ g)$ *redundant g*

shows $\text{defs } g \ n \cap \text{uses}' \ g \ n = \{\}$

<proof>

end

context *CFG-SSA-wf*

begin

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

for $g :: 'g$

where

liveSimple': $\llbracket n \in \text{set } (\alpha n \ g); \text{val} \in \text{uses } g \ n \rrbracket \implies \text{liveVal}' \ g \ [\text{val}]$

| *livePhi'*: $\llbracket \text{liveVal}' \ g \ (v \# \text{vs}); \text{phiArg } g \ v \ v' \rrbracket \implies \text{liveVal}' \ g \ (v' \# v \# \text{vs})$

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 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)
and
g :: 'g +
assumes redundant[simp]: redundant g
begin
abbreviation u-g ≡ uses(g:=uses' g)
abbreviation p-g ≡ phis(g:=phis' g)

sublocale step: CFG-SSA-Transformed-notriv-base αe αn invar inEdges' Entry
oldDefs oldUses defs u-g p-g var chooseNext-all ⟨proof⟩

lemma simpleDefs-phiDefs-disjoint-inv:
assumes n ∈ set (αn g)
shows defs g n ∩ step.phiDefs g n = {}
⟨proof⟩

lemma allDefs-disjoint-inv:
assumes n ∈ set (αn g) m ∈ set (αn g) n ≠ m
shows step.allDefs g n ∩ step.allDefs g m = {}
⟨proof⟩

```

lemma *phis-finite-inv*:
shows *finite (dom (phis' g))*
 \langle *proof* \rangle

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

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

lemma *allUses-narrows*:
assumes *n ∈ set (αn g)*
shows *step.allUses g n ⊆ substNext g ' allUses g n*
 \langle *proof* \rangle

lemma *allDefs-narrows[simp]*: *v ∈ step.allDefs g n ⇒ v ∈ allDefs g n*
 \langle *proof* \rangle

lemma *allUses-def-ass-inv*:
assumes *v' ∈ step.allUses g n n ∈ set (αn g)*
shows *step.defAss g n v'*
 \langle *proof* \rangle

lemma *Entry-no-phis-inv*: *phis' g (Entry g, v) = None*
 \langle *proof* \rangle

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

lemma *chooseNext-eliminated*: *chooseNext g ∉ step.allDefs g (defNode g (chooseNext g))*
 \langle *proof* \rangle

lemma *oldUses-inv*:
assumes *n ∈ set (αn g)*
shows *oldUses g n = var g ' u-g g n*
 \langle *proof* \rangle

lemma *conventional-inv*:
assumes *g ⊢ n-ns→m n ∉ set (tl ns) v ∈ step.allDefs g n v ∈ step.allUses g*
m x ∈ set (tl ns) v' ∈ step.allDefs g x
shows *var g v' ≠ var g v*
 \langle *proof* \rangle

lemma[*simp*]: *var g (substNext g v) = var g v*

<proof>

lemma *phis-same-var-inv*:

assumes *phis'* *g* (*n,v*) = *Some vs v' ∈ set vs*

shows *var g v' = var g v*

<proof>

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

<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 oldDefs oldUses defs u-g p-g var chooseNext-all*

<proof>

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

<proof>

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

<proof>

lemma *liveVal'-inv*:

assumes *liveVal'* *g* (*v#vs*) $v \neq \text{chooseNext } g$

obtains *vs'* **where** *step.liveVal'* *g* (*v#vs'*)

<proof>

lemma *liveVal-inv*:

assumes *liveVal* *g* *v* $v \neq \text{chooseNext } g$

shows *step.liveVal* *g* *v*

<proof>

lemma *pruned-inv*:

assumes *pruned* *g*

shows *step.pruned* *g*

<proof>

end

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

begin

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

abbreviation *inst'* *g* $\equiv \lambda(u,p). \text{inst } g \ u \ p$

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 αn inEdges' defs ($\text{uses}(g:=u)$) ($\text{phis}(g:=p)$)

g

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

⟨proof⟩

lemma *substAll*:

assumes $P (\text{uses } g, \text{phis } g)$
assumes $\bigwedge x. P x \implies \text{inst}' g x \implies \text{cond } g x \implies P (\text{step } g x)$

assumes $\bigwedge x. P x \implies inst' g x \implies \neg cond g x \implies Q (fst x) (snd x)$
shows $inst g (uses'-all g) (phis'-all g) Q (uses'-all g) (phis'-all g)$
 $\langle proof \rangle$

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

theorem *not-redundant*: $\neg notriv.redundant g$
 $\langle proof \rangle$

corollary *minimal*: $old.reducible g \implies notriv.cytronMinimal g$
 $\langle proof \rangle$

theorem *pruned-invariant*:
assumes *pruned g*
shows *notriv.pruned g*
 $\langle proof \rangle$

end

end

5 Proof of Semantic Equivalence

theory *SSA-Semantics* **imports** *Construct-SSA* **begin**

type-synonym $('node, 'var) state = 'var \rightarrow 'node$

context *CFG-SSA-Transformed*

begin

declare *invar*[*intro!*]

definition *step* ::

$'g \Rightarrow 'node \Rightarrow ('node, 'var) state \Rightarrow ('node, 'var) state$

where

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

inductive *bs* :: $'g \Rightarrow 'node list \Rightarrow ('node, 'var) state \Rightarrow bool$ ($- \vdash - \Downarrow - [50, 50, 50] 50$)

where

$g \vdash Entry g - ns \rightarrow last ns \implies g \vdash ns \Downarrow (fold (step g) ns Map.empty)$

definition *ssaStep* ::

$'g \Rightarrow 'node \Rightarrow nat \Rightarrow ('node, 'val) state \Rightarrow ('node, 'val) state$

where

$ssaStep g m i s v \equiv$

$if v \in defs g m then$

```

    Some m
  else
    case phis g (m,v) of
      Some phiParams => s (phiParams ! i)
    | None => s v

```

inductive *ssaBS* :: 'g => 'node list => ('node, 'val) state => bool (- ⊢ -↓_s - [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 ⊢ []↓_ss)
 ⟨proof⟩

lemma *ssaBS-hd[simp]*: g ⊢ ns↓_ss ==> 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** $\bigwedge s. \llbracket I s; b s \rrbracket \Longrightarrow I (c s)$
and $\bigwedge s. \llbracket I s; b s \rrbracket \Longrightarrow (c s, s) \in r$
shows *False*
 $\langle \text{proof} \rangle$

lemma *while-option-NoneD*:
assumes *while-option b c s = None*
and *wf r* **and** $\bigwedge s. b s \Longrightarrow (c s, s) \in r$
shows *False*
 $\langle \text{proof} \rangle$

lemma *while-option-sim*:
assumes *start: R (Some s1) (Some s2)*
and *cond: $\bigwedge s1 s2. \llbracket R (Some s1) (Some s2); I s1 \rrbracket \Longrightarrow b1 s1 = b2 s2$*
and *step : $\bigwedge s1 s2. \llbracket R (Some s1) (Some s2); I s1; b1 s1 \rrbracket \Longrightarrow R (Some (c1 s1)) (Some (c2 s2))$*
and *diverge: R None None*
and *inv-start: I s1*
and *inv-step: $\bigwedge s1. \llbracket I s1; b1 s1 \rrbracket \Longrightarrow I (c1 s1)$*
shows *R (while-option b1 c1 s1) (while-option b2 c2 s2)*
 $\langle \text{proof} \rangle$

end

theory *SSA-CFG-code* **imports**
SSA-CFG
Mapping-Exts
HOL-Library.Product-Lexorder
begin

definition *Union-of* :: $('a \Rightarrow 'b \text{ set}) \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set}$
where *Union-of f A* $\equiv \bigcup (f ` A)$

lemma *Union-of-alt-def*: *Union-of f A = $(\bigcup x \in A. f x)$*
 $\langle \text{proof} \rangle$

type-synonym $('node, 'val) \text{phis-code} = ('node \times 'val, 'val \text{ list}) \text{ mapping}$

context *CFG-base* **begin**

definition *addN* :: $'g \Rightarrow 'node \Rightarrow ('var, 'node \text{ set}) \text{ mapping} \Rightarrow ('var, 'node \text{ set}) \text{ mapping}$
where *addN g n* $\equiv \text{fold } (\lambda v. \text{Mapping.map-default } v \ \{\} \ (insert \ n)) \ (\text{sorted-list-of-set } (uses \ g \ n))$

definition *addN'* $g \ n = \text{fold } (\lambda v \ m. \ m(v \mapsto \text{case-option } \{n\} \ (insert \ n) \ (m \ v))) \ (\text{sorted-list-of-set } (uses \ g \ n))$

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 $Mapping.lookup (useNodes\text{-}of\ g)\ v =$
 (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'*: $useNodes\text{-}of' = useNodes\text{-}of$
 ⟨*proof*⟩

declare *useNodes-of'.transfer* [*unfolded useNodes-of', transfer-rule*]

lemma *lookup-useNodes-of'*: $Mapping.lookup (useNodes\text{-}of\ g)\ v =$
 (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 $phis\text{-}addN\ g\ n = fold (\lambda v. Mapping.map\text{-}default\ v\ \{\}) (insert\ n)) (case\text{-}option$
 $\ []\ id\ (phis\ g\ n))$

definition *phidefNodes* **where** [*code*]:

$phidefNodes\ g = fold (\lambda(n,v). Mapping.update\ v\ n) (sorted\text{-}list\text{-}of\text{-}set (dom (phis$
 $g))) Mapping.empty$

lemma *keys-phidefNodes*:

assumes $finite (dom (phis\ g))$

shows $Mapping.keys (phidefNodes\ g) = snd\ \text{'}\ dom (phis\ g)$

⟨*proof*⟩

definition *phiNodes-of* :: $'g \Rightarrow ('val, ('node \times 'val)\ set)\ mapping$

where $\text{phiNodes-of } g = \text{fold } (\text{phis-addN } g) (\text{sorted-list-of-set } (\text{dom } (\text{phis } g)))$
 Mapping.empty

lemma $\text{lookup-phiNodes-of}$:
assumes $[\text{simp}]$: $\text{finite } (\text{dom } (\text{phis } g))$
shows $\text{Mapping.lookup } (\text{phiNodes-of } g) v =$
 $(\text{if } (\exists n \in \text{dom } (\text{phis } g). v \in \text{set } (\text{the } (\text{phis } g n)))) \text{ then } \text{Some } \{n \in \text{dom } (\text{phis } g). v \in \text{set } (\text{the } (\text{phis } g n))\} \text{ else } \text{None})$
 $\langle \text{proof} \rangle$

lemmas $\text{phiNodes-of-code} = \text{phiNodes-of-def } [\text{unfolded } \text{phis-addN-def } [\text{abs-def}]]$
declare $\text{phiNodes-of-code } [\text{code}]$

lemma $\text{phis-transfer } [\text{transfer-rule}]$:
includes lifting-syntax
shows $((=) \implies \text{pcr-mapping } (=) (=)) \text{ phis } (\lambda g. \text{Mapping.Mapping } (\text{phis } g))$
 $\langle \text{proof} \rangle$

end

context CFG-SSA begin
declare $\text{lookup-phiNodes-of } [\text{OF } \text{phis-finite}, \text{simp}]$
declare $\text{keys-phidefNodes } [\text{OF } \text{phis-finite}, \text{simp}]$
end

locale $\text{CFG-SSA-ext-base} = \text{CFG-SSA-base } \alpha e \alpha n \text{ invar } \text{inEdges' Entry defs uses phis}$

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

begin

abbreviation $\text{cache } g f \equiv \text{Mapping.tabulate } (\alpha n g) f$

lemma $\text{lookup-cache}[\text{simp}]$: $n \in \text{set } (\alpha n g) \implies \text{Mapping.lookup } (\text{cache } g f) n = \text{Some } (f n)$
 $\langle \text{proof} \rangle$

lemma $\text{lookup-cacheD } [\text{dest}]$: $\text{Mapping.lookup } (\text{cache } g f) x = \text{Some } y \implies y = f x$
 $\langle \text{proof} \rangle$

lemma $\text{lookup-cache-usesD}$: $\text{Mapping.lookup } (\text{cache } g (\text{uses } g)) n = \text{Some } vs \implies vs = \text{uses } g n$
 $\langle \text{proof} \rangle$

end

definition $[simp]$: $usesOf\ m\ n \equiv case-option\ \{\}\ id\ (Mapping.lookup\ m\ n)$

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

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

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

begin

lemma $usesOf-cache[abs-def, simp]$: $usesOf\ (cache\ g\ (uses\ g))\ n = uses\ g\ n$

$\langle proof \rangle$

end

locale $CFG-SSA-base-code = CFG-SSA-ext-base\ \alpha e\ \alpha n\ invar\ inEdges'\ Entry\ defs\ usesOf\ \circ\ uses\ \lambda g.\ Mapping.lookup\ (phis\ g)$

for $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)\ set$

and $\alpha n :: 'g \Rightarrow 'node\ list$

and $invar :: 'g \Rightarrow bool$

and $inEdges' :: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)\ list$

and $Entry :: 'g \Rightarrow 'node$

and $defs :: 'g \Rightarrow 'node \Rightarrow 'val::linorder\ set$

and $uses :: 'g \Rightarrow ('node, 'val)\ set\ mapping$

and $phis :: 'g \Rightarrow ('node, 'val)\ phis-code$

begin

declare $phis-transfer\ [simplified, transfer-rule]$

lemma $phiDefs-code\ [code]$:

$phiDefs\ g\ n = snd\ 'Set.filter\ (\lambda(n',v).\ n' = n)\ (Mapping.keys\ (phis\ g))$

$\langle proof \rangle$

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

+ $CFG-SSA-ext\ \alpha e\ \alpha n\ invar\ inEdges'\ Entry\ defs\ usesOf\ \circ\ uses\ \lambda g.\ Mapping.lookup\ (phis\ g)$

for $\alpha e :: 'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)\ set$

and $\alpha n :: 'g \Rightarrow 'node\ list$

and $invar :: 'g \Rightarrow bool$

and *inEdges'* :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list
and *Entry* :: 'g ⇒ 'node
and *defs* :: 'g ⇒ 'node ⇒ 'val::linorder set
and *uses* :: 'g ⇒ ('node, 'val set) mapping
and *phis* :: 'g ⇒ ('node, 'val) phis-code

definition *the-trivial* v vs = (case (foldl (λ(good,v') w. if w = v then (good,v')
else case v' of Some v' ⇒ (good ∧ w = v', Some v')
| None ⇒ (good, Some w))
(True, None) vs)
of (False, -) ⇒ None | (True,v) ⇒ v)

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

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

lemma *the-trivial-conv*:
shows *the-trivial* v vs = (if ∃ v' ∈ set vs. v' ≠ v ∧ set vs - {v'} ⊆ {v} then
Some (THE v'. v' ∈ set vs ∧ v' ≠ v ∧ set vs - {v'} ⊆ {v}) else None)
⟨proof⟩

lemma *the-trivial-SomeE*:
assumes *the-trivial* v vs = Some v'
obtains v ≠ v' **and** set vs = {v'} | v ≠ v' **and** set vs = {v,v'}
⟨proof⟩

locale *CFG-SSA-wf-base-code* = *CFG-SSA-base-code* αe αn invar *inEdges'* *Entry*
defs *uses* *phis*

+ *CFG-SSA-wf-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

definition [code]:

trivial-code (v::'val) vs = (*the-trivial* v vs ≠ None)

definition[code]: *trivial-phis* g = *Set.filter* (λ(n,v). *trivial-code* v (*the* (*Mapping.lookup* (*phis* g) (n,v)))) (*Mapping.keys* (*phis* g))

definition [code]: *redundant-code* $g = (\text{trivial-phis } g \neq \{\})$
end

locale *CFG-SSA-wf-code* = *CFG-SSA-code* $\alpha e \alpha n$ *invar inEdges' Entry defs uses phis*
+ *CFG-SSA-wf-base-code* $\alpha e \alpha n$ *invar inEdges' Entry defs uses phis*
+ *CFG-SSA-wf* $\alpha e \alpha n$ *invar inEdges' Entry defs usesOf* \circ *uses* $\lambda g. \text{Mapping.lookup}$
(*phis* g)
for $\alpha e :: 'g \Rightarrow ('node::\text{linorder} \times 'edgeD \times 'node)$ **set**
and $\alpha n :: 'g \Rightarrow 'node$ **list**
and *invar* $:: 'g \Rightarrow \text{bool}$
and *inEdges'* $:: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)$ **list**
and *Entry* $:: 'g \Rightarrow 'node$
and *defs* $:: 'g \Rightarrow 'node \Rightarrow 'val::\text{linorder}$ **set**
and *uses* $:: 'g \Rightarrow ('node, 'val)$ **set** *mapping*
and *phis* $:: 'g \Rightarrow ('node, 'val)$ *phis-code*

begin
lemma *trivial-code*:
 $\text{phi } g \ v = \text{Some } vs \implies \text{trivial } g \ v = \text{trivial-code } v \ vs$
 $\langle \text{proof} \rangle$

lemma *trivial-phis*:
 $\text{trivial-phis } g = \{(n,v). \text{Mapping.lookup } (\text{phis } g) \ (n,v) \neq \text{None} \wedge \text{trivial } g \ v\}$
 $\langle \text{proof} \rangle$

lemma *redundant-code*:
 $\text{redundant } g = \text{redundant-code } g$
 $\langle \text{proof} \rangle$

lemma *trivial-code-mapI*:
 $\llbracket \text{trivial-code } v \ vs; f \ ' \ (\text{set } vs - \{v\}) \neq \{v\}; f \ v = v \rrbracket \implies \text{trivial-code } v \ (\text{map } f \ vs)$
 $\langle \text{proof} \rangle$

lemma *trivial-code-map-conv*:
 $f \ v = v \implies \text{trivial-code } v \ (\text{map } f \ vs) \longleftrightarrow (\exists v' \in \text{set } vs. f \ v' \neq v \wedge (f \ ' \ \text{set } vs) - \{f \ v'\} \subseteq \{v\})$
 $\langle \text{proof} \rangle$

end

locale *CFG-SSA-Transformed-code* = *ssa: CFG-SSA-wf-code* $\alpha e \alpha n$ *invar inEdges' Entry defs uses phis*
+
CFG-SSA-Transformed $\alpha e \alpha n$ *invar inEdges' Entry oldDefs oldUses defs usesOf*
 \circ *uses* $\lambda g. \text{Mapping.lookup}$ (*phis* g) *var*

for
 $\alpha e :: 'g \Rightarrow ('node::\text{linorder} \times 'edgeD \times 'node)$ **set** **and**
 $\alpha n :: 'g \Rightarrow 'node$ **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 *readVariableRecursive.simps*[simp del] *readArgs.simps*[simp del]

lemma *fst-readVariableRecursive*:
assumes $n \in \text{set } (\alpha n g)$

shows $\text{fst } (\text{readVariableRecursive } g \ v \ n \ \text{phis}) = \text{lookupDef } g \ n \ v$
 ⟨proof⟩

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
 ⟨proof⟩

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

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

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

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

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 $n \in \text{phiDefNodes-aux } g \ v \ un \ m'$
 ⟨proof⟩

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

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

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

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

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

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

lemma *from-sparse-inj*: *inj from-sparse*
<proof>

declare *uses'-phis'-def*[*unfolded aux-2-def*[*abs-def*] *aux-1-def*, *code*]

lift-definition *phis'-code* :: $'g \Rightarrow ('node, ('node, 'var)\ ssaVal)\ phis-code$ **is** *phis'*
<proof>

lemma *foldr-prod*: *foldr* ($\lambda x\ y. (f1\ x\ (fst\ y),\ f2\ x\ (snd\ y))$) *xs* $y = (foldr\ f1\ xs$
 $(fst\ y),\ foldr\ f2\ xs\ (snd\ y))$
<proof>

lemma *foldr-aux-1*:

assumes $set\ us \subseteq uses\ g\ n$ *Mapping.lookup* $u\ n = None$ *foldr* (*aux-1* $g\ n$) *us*
 $(u,p) = (u',p')$ (**is** *foldr* $?f\ -\ =\ -$)
assumes *finite* (*Mapping.keys* p) $\wedge n\ v. (n,v) \in Mapping.keys\ p \implies length$
(*predecessors* $g\ n$) $\neq 1 \wedge v. Mapping.lookup\ p\ (Entry\ g,v) \in \{None,\ Some\ []\}$
shows *lookupDef* $g\ n\ 'set\ us = lookup-multimap\ u'\ n \wedge m. m \neq n \implies Map-$
ping.lookup $u'\ m = Mapping.lookup\ u\ m$
 $\wedge m\ v. (if\ m \in phiDefNodes-aux\ g\ v\ [n \leftarrow \alpha n\ g. (n,v) \notin Mapping.keys\ p]\ n \wedge$
 $v \in set\ us\ then$
Some (*map* ($\lambda m. lookupDef\ g\ m\ v$) (*predecessors* $g\ m$)) *else*
(*Mapping.lookup* $p\ (m,v)$) = *Mapping.lookup* $p'\ (m,v)$
<proof>

lemma *foldr-aux-2*:

assumes $set\ ns \subseteq set\ (\alpha n\ g)$ *distinct* *ns* *foldr* (*aux-2* g) *ns* *init-state* = (u',p')
shows $\wedge n. n \in set\ ns \implies uses'\ g\ n = lookup-multimap\ u'\ n \wedge n. n \notin set\ ns$
 $\implies Mapping.lookup\ u'\ n = None$
 $\wedge m\ v. (if\ \exists n \in set\ ns. m \in phiDefNodes-aux\ g\ v\ (\alpha n\ g)\ n \wedge v \in uses\ g\ n$
then
Some (*map* ($\lambda m. lookupDef\ g\ m\ v$) (*predecessors* $g\ m$)) *else*
None) = *Mapping.lookup* $p'\ (m,v)$
<proof>

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

lemma *fst-uses'-phis'-in- αn* : $Mapping.keys (fst (uses'-phis' g)) \subseteq set (\alpha n g)$
 ⟨*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*]: $right-total R \implies ((R \implies (=)) \implies (\implies)) All All$
 ⟨*proof*⟩

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

lemma *weak-conj-transfer* [*transfer-rule*]:
 $((\implies) \implies (\implies) \implies (\implies)) (\wedge) (\wedge)$
 ⟨*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 $(\implies) (graph-path \alpha e \alpha n invar inEdges) (graph-path \alpha e2 \alpha n2 invar2 inEdges2)$

⟨*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*]: $(G \implies (=)) \textit{invar1 invar2}$

and [*transfer-rule*]: $(G \implies (=)) \textit{inEdges1 inEdges2}$

and [*transfer-rule*]: $(G \implies (=)) \textit{Entry1 Entry2}$

shows (\longrightarrow) (*graph-Entry* $\alpha e1 \alpha n1 \textit{invar1 inEdges1 Entry1}$) (*graph-Entry* $\alpha e2 \alpha n2 \textit{invar2 inEdges2 Entry2}$)

⟨*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*]: $(G \implies (=)) \textit{invar invar2}$

and [*transfer-rule*]: $(G \implies (=)) \textit{inEdges' inEdges2}$

and [*transfer-rule*]: $(G \implies (=)) \textit{Entry Entry2}$

shows $(G \implies (=))$ *dominates* (*graph-Entry-base.dominates* $\alpha n2 \textit{invar2 inEdges2 Entry2}$)

⟨*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*]: $(G \implies (=)) \textit{invar invar2}$

and [*transfer-rule*]: $(G \implies (=)) \textit{inEdges' inEdges2}$

and [*transfer-rule*]: ($G \text{ ===> } (=)$) *Entry Entry2*
shows ($G \text{ ===> } (=)$) *shortestPath (graph-Entry.shortestPath $\alpha n2$ invar2 inEdges2 Entry2)*
 <*proof*>

lemma *dominators-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: ($G \text{ ===> } (=)$) $\alpha e \alpha e2$
and [*transfer-rule*]: ($G \text{ ===> } (=)$) $\alpha n \alpha n2$
and [*transfer-rule*]: ($G \text{ ===> } (=)$) *invar invar2*
and [*transfer-rule*]: ($G \text{ ===> } (=)$) *inEdges' inEdges2*
and [*transfer-rule*]: ($G \text{ ===> } (=)$) *Entry Entry2*
shows ($G \text{ ===> } (=)$) *dominators (graph-Entry.dominators $\alpha n2$ invar2 inEdges2 Entry2)*
 <*proof*>

lemma *isIdom-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: ($G \text{ ===> } (=)$) $\alpha e \alpha e2$
and [*transfer-rule*]: ($G \text{ ===> } (=)$) $\alpha n \alpha n2$
and [*transfer-rule*]: ($G \text{ ===> } (=)$) *invar invar2*
and [*transfer-rule*]: ($G \text{ ===> } (=)$) *inEdges' inEdges2*
and [*transfer-rule*]: ($G \text{ ===> } (=)$) *Entry Entry2*
shows ($G \text{ ===> } (=)$) *isIdom (graph-Entry.isIdom $\alpha n2$ invar2 inEdges2 Entry2)*
 <*proof*>

lemma *idom-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: ($G \text{ ===> } (=)$) $\alpha e \alpha e2$
and [*transfer-rule*]: ($G \text{ ===> } (=)$) $\alpha n \alpha n2$
and [*transfer-rule*]: ($G \text{ ===> } (=)$) *invar invar2*
and [*transfer-rule*]: ($G \text{ ===> } (=)$) *inEdges' inEdges2*
and [*transfer-rule*]: ($G \text{ ===> } (=)$) *Entry Entry2*
shows ($G \text{ ===> } (=)$) *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]: ( $G \implies (=)$ )  $\alpha e1 \alpha e2$ 
and [transfer-rule]: ( $G \implies (=)$ )  $\alpha n1 \alpha n2$ 
and [transfer-rule]: ( $G \implies (=)$ )  $invar1 invar2$ 
and [transfer-rule]: ( $G \implies (=)$ )  $inEdges1 inEdges2$ 
and [transfer-rule]: ( $G \implies (=)$ )  $Entry1 Entry2$ 
and [transfer-rule]: ( $G \implies (=)$ )  $defs1 defs2$ 
and [transfer-rule]: ( $G \implies (=)$ )  $uses1 uses2$ 
shows  $SSA-CFG.CFG \alpha e1 \alpha n1 invar1 inEdges1 Entry1 defs1 uses1$ 
   $\rightarrow SSA-CFG.CFG \alpha e2 \alpha n2 invar2 inEdges2 Entry2 defs2 uses2$ 
<proof>

```

context *CFG-base* **begin**

context includes *lifting-syntax*
begin

```

lemma vars-transfer [transfer-rule]:
assumes [transfer-rule]: right-total  $G$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $\alpha e \alpha e2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $\alpha n \alpha n2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $invar invar2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $inEdges' inEdges2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $Entry Entry2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $defs defs2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $uses uses2$ 
shows ( $G \implies (=)$ )  $vars (CFG-base.vars \alpha n2 uses2)$ 
<proof>

```

```

lemma defAss'-transfer [transfer-rule]:
assumes [transfer-rule]: right-total  $G$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $\alpha e \alpha e2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $\alpha n \alpha n2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $invar invar2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $inEdges' inEdges2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $Entry Entry2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $defs defs2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $uses uses2$ 
shows ( $G \implies (=)$ )  $defAss' (CFG-base.defAss' \alpha n2 invar2 inEdges2 Entry2$ 
 $defs2)$ 
<proof>

```

```

lemma defAss'Uses-transfer [transfer-rule]:
assumes [transfer-rule]: right-total  $G$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $\alpha e \alpha e2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $\alpha n \alpha n2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $invar invar2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $inEdges' inEdges2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $Entry Entry2$ 
  and [transfer-rule]: ( $G \implies (=)$ )  $defs defs2$ 

```

and [*transfer-rule*]: ($G \implies (=)$) *uses uses2*
shows ($G \implies (=)$) *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*]: ($G \implies (=)$) $\alpha e1 \alpha e2$
and [*transfer-rule*]: ($G \implies (=)$) $\alpha n1 \alpha n2$
and [*transfer-rule*]: ($G \implies (=)$) *invar1 invar2*
and [*transfer-rule*]: ($G \implies (=)$) *inEdges1 inEdges2*
and [*transfer-rule*]: ($G \implies (=)$) *Entry1 Entry2*
and [*transfer-rule*]: ($G \implies (=)$) *defs1 defs2*
and [*transfer-rule*]: ($G \implies (=)$) *uses1 uses2*
shows *CFG-Construct* $\alpha e1 \alpha n1$ *invar1 inEdges1 Entry1 defs1 uses1*
 \longrightarrow *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*]: ($G \implies (=)$) $\alpha e1 \alpha e2$
and [*transfer-rule*]: ($G \implies (=)$) $\alpha n1 \alpha n2$
and [*transfer-rule*]: ($G \implies (=)$) *invar1 invar2*
and [*transfer-rule*]: ($G \implies (=)$) *inEdges1 inEdges2*
and [*transfer-rule*]: ($G \implies (=)$) *Entry1 Entry2*
and [*transfer-rule*]: ($G \implies (=)$) *defs1 defs2*
and [*transfer-rule*]: ($G \implies (=)$) *uses1 uses2*
shows *CFG-Construct-linorder* $\alpha e1 \alpha n1$ *invar1 inEdges1 Entry1 defs1 uses1*
 \longrightarrow *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*]: $(G \text{ ==== } \Rightarrow (=)) \alpha e \alpha e2$
and [*transfer-rule*]: $(G \text{ ==== } \Rightarrow (=)) \alpha n \alpha n2$
and [*transfer-rule*]: $(G \text{ ==== } \Rightarrow (=)) \textit{invar} \textit{invar}2$
and [*transfer-rule*]: $(G \text{ ==== } \Rightarrow (=)) \textit{inEdges}' \textit{inEdges}2$
and [*transfer-rule*]: $(G \text{ ==== } \Rightarrow (=)) \textit{Entry} \textit{Entry}2$
and [*transfer-rule*]: $(G \text{ ==== } \Rightarrow (=)) \textit{defs} \textit{defs}2$
and [*transfer-rule*]: $(G \text{ ==== } \Rightarrow (=)) \textit{uses} \textit{uses}2$
shows $(G \text{ ==== } \Rightarrow (=)) \textit{phiDefNodes-aux} (\textit{CFG-Construct.phiDefNodes-aux} \textit{inEdges}2 \textit{defs}2)$
<proof>

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

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

lemma *defs'-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: $(G \text{ ==== } \Rightarrow (=)) \alpha e \alpha e2$
and [*transfer-rule*]: $(G \text{ ==== } \Rightarrow (=)) \alpha n \alpha n2$
and [*transfer-rule*]: $(G \text{ ==== } \Rightarrow (=)) \textit{invar} \textit{invar}2$
and [*transfer-rule*]: $(G \text{ ==== } \Rightarrow (=)) \textit{inEdges}' \textit{inEdges}2$
and [*transfer-rule*]: $(G \text{ ==== } \Rightarrow (=)) \textit{Entry} \textit{Entry}2$

```

    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 ==> (=))  $\alpha n \alpha n2$ 

```

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  $\implies$  (=)) αe αe2
    and [transfer-rule]: (G  $\implies$  (=)) αn αn2
    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  $\implies$  (=)) 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  $\implies$  (=)) αe αe2
    and [transfer-rule]: (G  $\implies$  (=)) αn αn2
    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  $\implies$  (=)) 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  $\implies$  (=)) αe αe2
    and [transfer-rule]: (G  $\implies$  (=)) αn αn2
    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 ==> (=)) 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* $x \equiv (\lambda-. x)$

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

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

definition [*code*]: *substNext-code* $g \ next \equiv \lambda v. \text{if } v = snd \ next \text{ then substitution-code } g \ next \text{ else } v$

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 λg . Mapping.lookup (phis g) var λ 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 (λn . Mapping.update n (Set.insert next' (Set.remove (snd next) (the
(Mapping.lookup (uses g) n))))
(sorted-list-of-set (case-option {}) id (Mapping.lookup nodes-of-uses (snd
next))))
(uses g)

definition *nodes-of-uses'* :: *'g $\Rightarrow 'node \times 'val \Rightarrow 'val \Rightarrow 'val$ 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' {} (λns . 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'* :: $'g \Rightarrow 'node \times 'val \Rightarrow 'val \Rightarrow ('val, ('node \times 'val) \text{ set})$
mapping $\Rightarrow ('val, ('node \times 'val) \text{ set})$ *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* :: $'g \Rightarrow ('val, 'a \text{ set})$ *mapping* $\Rightarrow ('val, 'a \text{ set})$
mapping $\Rightarrow \text{bool}$ ($- \vdash - \approx_{\varphi} - 50$)

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 nou₁ v*) = *case-option* { } *id* (*Mapping.lookup nou₂ v*)
end

locale *CFG-SSA-Transformed-notriv-linorder* = *CFG-SSA-Transformed-notriv-base*
 αe αn *invar inEdges' Entry oldDefs oldUses defs uses phis var chooseNext-all*

+ *CFG-SSA-Transformed-notriv* αe αn *invar inEdges' Entry oldDefs oldUses*
defs uses phis var chooseNext-all

for

αe :: $'g \Rightarrow ('node::\text{linorder} \times 'edgeD \times 'node) \text{ set}$ **and**

αn :: $'g \Rightarrow 'node \text{ list}$ **and**

invar :: $'g \Rightarrow \text{bool}$ **and**

inEdges' :: $'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD) \text{ list}$ **and**

Entry:: $'g \Rightarrow 'node$ **and**

oldDefs :: $'g \Rightarrow 'node \Rightarrow 'var::\text{linorder} \text{ set}$ **and**

oldUses :: $'g \Rightarrow 'node \Rightarrow 'var \text{ set}$ **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 = 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' Entry
  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

```

$\langle \text{proof} \rangle$

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

for u p
 $\langle \text{proof} \rangle$

lemma *phis'-code-correct*:

assumes *ssa.redundant* g

shows *phis' g = Mapping.lookup (phis'-code g (chooseNext' g))*

$\langle \text{proof} \rangle$

lemma *redundant-ign[simp]*: *uninst-code.ssa.redundant-code (const p) g = uninst-code.ssa.redundant-code (phis(g:=p)) g*

$\langle \text{proof} \rangle$

lemma *uses'-ign[simp]*: *uninst-code.uses'-codem (const u) g = uninst-code.uses'-codem (u-g g u) g*

$\langle \text{proof} \rangle$

lemma *phis'-ign[simp]*: *uninst-code.phis'-code (const p) g = uninst-code.phis'-code (phis(g:=p)) g*

$\langle \text{proof} \rangle$

lemma *phis'm-ign[simp]*: *uninst-code.phis'-codem (const p) g = uninst-code.phis'-codem (phis(g:=p)) g*

$\langle \text{proof} \rangle$

lemma *set-sorted-list-of-set-phis-dom [simp]*:

set (sorted-list-of-set $\{x \in \text{dom} (\text{Mapping.lookup} (\text{phis } g)). P x\} = \{x \in \text{dom} (\text{Mapping.lookup} (\text{phis } g)). P x\}$)

$\langle \text{proof} \rangle$

lemma *phis'-codem-correct*:

assumes $g \vdash \text{nodes-of-phis} \approx_{\varphi} (\text{ssa.phiNodes-of } g)$ **and** $\text{next} \in \text{Mapping.keys} (\text{phis } g)$

shows *phis'-codem g next (substitution-code g next) nodes-of-phis = phis'-code g next*

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *uses'-codem-correct*:

assumes $g \vdash \text{nodes-of-uses} \approx_{\varphi} \text{ssa.useNodes-of } g$ **and** $\text{next} \in \text{Mapping.keys} (\text{phis } g)$

shows *usesOf (uses'-codem g next (substitution-code g next) nodes-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 *dom* (*Mapping.lookup* (*phis'-codem g next next' nodes-of-phis*)) = *dom* (*Mapping.lookup* (*phis g*)) \cup (*case-option* {} *id* (*Mapping.lookup nodes-of-phis (snd next)*)) - {*next*}
<proof>

lemma *dom-phis'-code* [*simp*]:

shows *dom* (*Mapping.lookup* (*phis'-code g next*)) = *dom* (*Mapping.lookup* (*phis g*)) - {*v. snd v = snd next*}
<proof>

lemma *nodes-of-phis-finite* [*simplified*]:

assumes $g \vdash \text{nodes-of-phis} \approx_{\varphi} \text{ssa.phiNodes-of } g$ **and** *Mapping.lookup nodes-of-phis v = 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 *Mapping.lookup* (*phis'-codem g next next' nodes-of-phis*) *next* = *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 *Mapping.lookup* (*phis'-codem g next (substitution-code g next) nodes-of-phis*)
 $\varphi =$
map-option (*map* (*substNext-code g next*)) (*Mapping.lookup* (*phis g*) φ)
<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*

$g \text{ next})) \cup (\text{case-option } \{\} (\text{Set.remove next}) (\text{Mapping.lookup nodes-of-phis (snd next)}))$
 ⟨proof⟩

lemma *lookup-nodes-of-phis'-not-subst:*

$v \neq \text{substitution-code } g \text{ next} \implies$

$\text{Mapping.lookup (nodes-of-phis' } g \text{ next (substitution-code } g \text{ next) nodes-of-phis) } v$
 $= (\text{if } v = \text{snd next} \text{ then None else } \text{Mapping.lookup nodes-of-phis } v)$
 ⟨proof⟩

lemma *lookup-phis'-code:*

$\text{Mapping.lookup (phis'-code } g \text{ next) } v = (\text{if } \text{snd } v = \text{snd next} \text{ then None else}$
 $\text{map-option (map (substNext-code } g \text{ next)) (Mapping.lookup (phis } g) v))$
 ⟨proof⟩

lemma *phi-equiv-mappingE':*

assumes $g \vdash m_1 \approx_\varphi \text{ssa.phiNodes-of } g$

and $\text{Mapping.lookup (phis } g) x = \text{Some } vs$ **and** $b \in \text{set } vs$ **and** $b \in \text{snd '}$
 $\text{Mapping.keys (phis } g)$

obtains $\text{Mapping.lookup } m_1 b = \text{Some } \{n \in \text{Mapping.keys (phis } g). b \in \text{set}$
 $(\text{the (Mapping.lookup (phis } g) n))\}$

⟨proof⟩

lemma *phi-equiv-mappingE:*

assumes $g \vdash m_1 \approx_\varphi \text{ssa.phiNodes-of } g$ **and** $b \in \text{Mapping.keys (phis } g)$

and $\text{Mapping.lookup (phis } g) x = \text{Some } vs$ **and** $\text{snd } b \in \text{set } vs$

obtains ns **where** $\text{Mapping.lookup } m_1 (\text{snd } b) = \text{Some } \{n \in \text{Mapping.keys}$
 $(\text{phis } g). \text{snd } b \in \text{set (the (Mapping.lookup (phis } g) n))\}$

⟨proof⟩

lemma *phi-equiv-mappingE2':*

assumes $g \vdash m_1 \approx_\varphi \text{ssa.phiNodes-of } g$

and $b \in \text{snd ' Mapping.keys (phis } g)$

and $\forall \varphi \in \text{Mapping.keys (phis } g). b \notin \text{set (the (Mapping.lookup (phis } g) \varphi))$

shows $\text{Mapping.lookup } m_1 b = \text{None} \vee \text{Mapping.lookup } m_1 b = \text{Some } \{\}$

⟨proof⟩

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

⟨proof⟩

lemma *keys-phis'-codem':*

assumes $g \vdash \text{nodes-of-phis} \approx_\varphi \text{ssa.phiNodes-of } g$ **and** $\text{next} \in \text{Mapping.keys}$
 $(\text{phis } g)$

shows $\text{Mapping.keys (phis'-codem } g \text{ next next' nodes-of-phis) = \text{Mapping.keys}$
 $(\text{phis } g) - \{\text{next}\}$

⟨proof⟩

lemma *triv-phis'-correct:*

assumes $g \vdash \text{nodes-of-phis} \approx_{\varphi} \text{ssa.phiNodes-of } g$ **and** $\text{next} \in \text{Mapping.keys}$
 $(\text{phis } g)$ **and** $\text{ssa.trivial } g (\text{snd next})$

shows $\text{uninst-code.triv-phis}' (\text{const } (\text{phis}'\text{-codem } g \text{ next } (\text{substitution-code } g \text{ next})$
 $\text{nodes-of-phis})) g \text{ next } (\text{ssa.trivial-phis } g) \text{ nodes-of-phis} = \text{uninst-code.ssa.trivial-phis}$
 $(\text{const } (\text{phis}'\text{-codem } g \text{ next } (\text{substitution-code } g \text{ next}) \text{ nodes-of-phis})) g$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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 $u g x \equiv u\text{-g } g (\text{fst } x)$

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

lemma *usesOf-upd* [simp]: $(usesOf \circ u \ g \ s1)(g := usesOf \ us) = usesOf \circ u \ g \ g$
us
 ⟨proof⟩

lemma *keys-uses'-codem* [simp]: $Mapping.keys \ (uses'-codem \ g \ next \ (substitution-code \ g \ next) \ (ssa.useNodes-of \ g)) = Mapping.keys \ (uses \ g)$
 ⟨proof⟩

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

lemma *triv-phis-base* [simp]: $uninst-code.ssa.trivial-phis \ (const \ (phis \ g)) \ g = ssa.trivial-phis \ g$
 ⟨proof⟩

lemma *useNodes-of-base* [simp]: $uninst-code.ssa.useNodes-of \ (const \ (uses \ g)) \ g = ssa.useNodes-of \ g$
 ⟨proof⟩

lemma *phiNodes-of-base* [simp]: $uninst-code.ssa.phiNodes-of \ (const \ (phis \ g)) \ g = ssa.phiNodes-of \ g$
 ⟨proof⟩

lemma *phi-equiv-mapping-refl* [simp]: $uninst-code.phi-equiv-mapping \ ph \ g \ m \ m$
 ⟨proof⟩

lemma *substAll-efficient-code* [code]:
 $substAll \ g = map-prod \ usesOf \ Mapping.lookup \ (fst \ (substAll-efficient \ g))$
 ⟨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* =

$gen-\alpha e :: ('node, 'edge) \text{ edge set}$
 $gen-\alpha n :: 'node \text{ list}$
 $gen-inEdges :: 'node \Rightarrow ('node, 'edge) \text{ edge list}$
 $gen-Entry :: 'node$
 $gen-defs :: 'node \Rightarrow 'var \text{ set}$
 $gen-uses :: 'node \Rightarrow 'var \text{ set}$

abbreviation $trivial-gen-cfg \text{ ext} \equiv gen-cfg-ext \{ \} [undefined] (const []) undefined (const \{ \}) (const \{ \}) \text{ ext}$

abbreviation $(input) \text{ ign } f \ g \ (-::unit) \equiv f \ g$

lemma $set-iterator-foldri-Nil$ [*simp, intro!*]: $set-iterator (foldri [] \{ \})$
 $\langle proof \rangle$

lemma $set-iterator-foldri-one$ [*simp, intro!*]: $set-iterator (foldri [a] \{ a \})$
 $\langle proof \rangle$

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

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

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

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

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

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

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

lift-definition $gen-wf-Entry :: ('node::linorder, 'var::linorder, 'edge) \ gen-cfg-wf \Rightarrow 'node \text{ is } gen-Entry \langle proof \rangle$

lift-definition $gen-wf-defs :: ('node::linorder, 'var::linorder, 'edge) \ gen-cfg-wf \Rightarrow 'node \Rightarrow 'var \text{ set} \text{ is } gen-defs \langle proof \rangle$

lift-definition $gen-wf-uses :: ('node::linorder, 'var::linorder, 'edge) \ gen-cfg-wf \Rightarrow 'node \Rightarrow 'var \text{ set} \text{ is } gen-uses \langle proof \rangle$

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

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

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-αe.transfer*
gen-wf-αn.transfer gen-wf-invar-trans gen-wf-inEdges'-transfer, transfer-rule]

declare *CFG-base.defAss'-transfer*[*OF gen-cfg-wf.right-total gen-wf-αe.transfer gen-wf-αn.transfer*
gen-wf-invar-trans gen-wf-inEdges'-transfer, transfer-rule]

global-interpretation *gen-wf: CFG-Construct-linorder gen-wf-αe gen-wf-αn 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 ⇒ 'val set

gen-ssa-uses :: ('node, 'val set) mapping

gen-phis :: ('node, 'val) phis-code

gen-var :: 'val ⇒ 'var

typedef (*'node, 'var, 'edge, 'val*) *gen-ssa-cfg-wf = {g :: ('node::linorder, 'var::linorder,*
'edge, 'val::linorder) gen-ssa-cfg.

CFG-SSA-Transformed-code (ign gen-αe g) (ign gen-αn g) (const True) (ign
gen-inEdges' g) (ign gen-Entry g) (ign gen-defs g) (ign gen-uses g) (ign gen-ssa-defs
g) (ign gen-ssa-uses g) (ign gen-phis g) (ign gen-var g)}

⟨proof⟩

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

lift-definition *gen-ssa-wf-αn :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)*

gen-ssa-cfg-wf ⇒ 'node list is gen-αn ⟨proof⟩

lift-definition *gen-ssa-wf-αe :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)*

gen-ssa-cfg-wf ⇒ ('node, 'edge) edge set is gen-αe ⟨proof⟩

lift-definition *gen-ssa-wf-inEdges :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)*

gen-ssa-cfg-wf ⇒ 'node ⇒ ('node, 'edge) edge list is gen-inEdges ⟨proof⟩

lift-definition *gen-ssa-wf-Entry :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)*

gen-ssa-cfg-wf ⇒ 'node is gen-Entry ⟨proof⟩

lift-definition *gen-ssa-wf-defs :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)*

gen-ssa-cfg-wf \Rightarrow 'node \Rightarrow 'var set **is** *gen-defs* <proof>
lift-definition *gen-ssa-wf-uses* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)
gen-ssa-cfg-wf \Rightarrow 'node \Rightarrow 'var set **is** *gen-uses* <proof>
lift-definition *gen-ssa-wf-ssa-defs* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)
gen-ssa-cfg-wf \Rightarrow 'node \Rightarrow 'val set **is** *gen-ssa-defs* <proof>
lift-definition *gen-ssa-wf-ssa-uses* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)
gen-ssa-cfg-wf \Rightarrow ('node, 'val set) mapping **is** *gen-ssa-uses* <proof>
lift-definition *gen-ssa-wf-phis* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)
gen-ssa-cfg-wf \Rightarrow ('node, 'val) *phis-code* **is** *gen-phis* <proof>
lift-definition *gen-ssa-wf-var* :: ('node::linorder, 'var::linorder, 'edge, 'val::linorder)
gen-ssa-cfg-wf \Rightarrow 'val \Rightarrow 'var **is** *gen-var* <proof>

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

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

global-interpretation *uninst*: *CFG-SSA-wf-base-code gen-ssa-wf- αe gen-ssa-wf- αn gen-wf-invar gen-ssa-wf-inEdges' gen-ssa-wf-Entry gen-ssa-wf-ssa-defs* $u\ p$

for u **and** p

defines

uninst-predecessors = *uninst.predecessors*
and *uninst-successors* = *uninst.successors*
and *uninst-phiDefs* = *uninst.phiDefs*
and *uninst-phiUses* = *uninst.phiUses*
and *uninst-allDefs* = *uninst.allDefs*
and *uninst-allUses* = *uninst.allUses*
and *uninst-allVars* = *uninst.allVars*
and *uninst-isTrivialPhi* = *uninst.isTrivialPhi*
and *uninst-trivial* = *uninst.trivial-code*
and *uninst-redundant* = *uninst.redundant-code*
and *uninst-phi* = *uninst.phi*
and *uninst-defNode* = *uninst.defNode*
and *uninst-trivial-phis* = *uninst.trivial-phis*
and *uninst-phidefNodes* = *uninst.phidefNodes*
and *uninst-useNodes-of* = *uninst.useNodes-of*
and *uninst-phiNodes-of* = *uninst.phiNodes-of*
 <proof>

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

lemma *gen-ssa-wf-invar-trans*: *rel-fun cr-gen-ssa-cfg-wf (=) gen-wf-invar gen-wf-invar*
 <proof>

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- α e.transfer gen-ssa-wf- α n.transfer gen-ssa-wf-invar-trans gen-ssa-wf-inEdges'-transfer, transfer-rule*]

declare *CFG-SSA-base-code.CFG-SSA-base-code-transfer-rules*[*OF gen-ssa-cfg-wf.right-total gen-ssa-wf- α e.transfer gen-ssa-wf- α n.transfer gen-ssa-wf-invar-trans gen-ssa-wf-inEdges'-transfer gen-ssa-wf-Entry.transfer gen-ssa-wf-ssa-defs.transfer gen-ssa-wf-ssa-uses.transfer gen-ssa-wf-phis.transfer, transfer-rule*]

lemma *path2-ign[simp]*: *graph-path-base.path2 (ign gen- α n g) gen-wf-invar (ign gen-inEdges' g) g' n ns m \longleftrightarrow graph-path-base.path2 gen- α n 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-phis g)) ga n = CFG-SSA-base.allDefs gen-ssa-defs (λ g. Mapping.lookup (gen-phis g)) g n*
 <proof>

lemma *successors-ign[simp]*: *graph-path-base.successors (ign gen- α n g) (ign gen-inEdges' g) ga n = graph-path-base.successors gen- α n 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-phis g)) ga = CFG-SSA-base.phiDefs (λ g. Mapping.lookup (gen-phis g)) g*
 <proof>

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

lemma *allUses-ign[simp]*: *CFG-SSA-base.allUses (ign gen- α n g) (ign gen-inEdges' g) (usesOf \circ ign gen-ssa-uses g) (ign Mapping.lookup (gen-phis g)) ga m*
 = *CFG-SSA-base.allUses gen- α n gen-inEdges' (usesOf \circ gen-ssa-uses) (λ g. Mapping.lookup (gen-phis g)) g m*
 <proof>

lemma *defAss'-ign[simp]*: *CFG-base.defAss' (ign gen- α n g) gen-wf-invar (ign gen-inEdges' g) (ign gen-Entry g) (ign gen-defs g) ga*
 = *CFG-base.defAss' gen- α n 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- α e gen-ssa-wf- α n gen-wf-invar gen-ssa-wf-inEdges' gen-ssa-wf-Entry gen-ssa-wf-defs gen-ssa-wf-uses gen-ssa-wf-ssa-defs gen-ssa-wf-ssa-uses gen-ssa-wf-phis gen-ssa-wf-var uninstant-chooseNext*

defines

gen-ssa-wf-notriv-substAll = gen-ssa-wf-notriv.substAll and
gen-ssa-wf-notriv-substAll-efficient = gen-ssa-wf-notriv.substAll-efficient

<proof>

global-interpretation *uninst-code: CFG-SSA-Transformed-notriv-base-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 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- αe -wf-extend [simp]:*

gen- αe (gen-cfg-wf-extend gen-cfg-wf ($\!|$ gen-ssa-defs = d, gen-ssa-uses = u, gen-phis = p, gen-var = v))
= gen-wf- αe gen-cfg-wf
<proof>

lemma *gen- αn -wf-extend [simp]:*

gen- αn (gen-cfg-wf-extend gen-cfg-wf ($\!|$ gen-ssa-defs = d, gen-ssa-uses = u, gen-phis = p, gen-var = v))
= gen-wf- αn gen-cfg-wf
<proof>

lemma *gen-inEdges-wf-extend [simp]:*

gen-inEdges (gen-cfg-wf-extend gen-cfg-wf ($\!|$ 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 ($\!|$ 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 ($\!|$ 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* ($\langle \text{gen-ssa-defs} = d, \text{gen-ssa-uses} = u, \text{gen-phs} = p, \text{gen-var} = v \rangle$))
= *gen-wf-uses* *gen-cfg-wf*
<proof>

lemma *gen-ssa-defs-wf-extend* [*simp*]:

gen-ssa-defs (*gen-cfg-wf-extend* *gen-cfg-wf* ($\langle \text{gen-ssa-defs} = d, \text{gen-ssa-uses} = u, \text{gen-phs} = p, \text{gen-var} = v \rangle$))
= *d*
<proof>

lemma *gen-ssa-uses-wf-extend* [*simp*]:

gen-ssa-uses (*gen-cfg-wf-extend* *gen-cfg-wf* ($\langle \text{gen-ssa-defs} = d, \text{gen-ssa-uses} = u, \text{gen-phs} = p, \text{gen-var} = v \rangle$))
= *u*
<proof>

lemma *gen-phs-wf-extend* [*simp*]:

gen-phs (*gen-cfg-wf-extend* *gen-cfg-wf* ($\langle \text{gen-ssa-defs} = d, \text{gen-ssa-uses} = u, \text{gen-phs} = p, \text{gen-var} = v \rangle$))
= *p*
<proof>

lemma *gen-var-wf-extend* [*simp*]:

gen-var (*gen-cfg-wf-extend* *gen-cfg-wf* ($\langle \text{gen-ssa-defs} = d, \text{gen-ssa-uses} = u, \text{gen-phs} = p, \text{gen-var} = v \rangle$))
= *v*
<proof>

lemma *CFG-SSA-Transformed-codeI*:

assumes *CFG-SSA-Transformed* $\alpha e \alpha n$ *invar inEdges Entry oldDefs oldUses defs*
($\lambda g. \text{lookup-multimap } (\text{uses } g) (\lambda g. \text{Mapping.lookup } (\text{phs } g)) \text{ var}$)
and $\bigwedge g. \text{Mapping.keys } (\text{uses } g) \subseteq \text{set } (\alpha n \ g)$
shows *CFG-SSA-Transformed-code* $\alpha e \alpha n$ *invar inEdges Entry oldDefs oldUses*
defs uses phs 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.phs'* *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*: (∧ *a b*. [∃ *a* ∈ set (*x # xs*); *b* ∈ set (*x # xs*)] ⇒ *f a* ∘ *f b* = *f b* ∘ *f a*)
⇒ *fold f (x # xs)* = *f x* ∘ (*fold f xs*)
<proof>

lemma *Union-of-code* [*code*]: *Union-of f (RBT-Set.Set r)* = *RBT.fold (λ a -. (∪ (f a)) r {})*
<proof>

definition[*code*]: *disjoint xs ys* = (*xs* ∩ *ys* = {})

definition *gen-ssa-wf-notriv-substAll'* = *fst* ∘ *gen-ssa-wf-notriv-substAll-efficient*

definition *fold-set f A* ≡ *fold f (sorted-list-of-set A)*
declare *fold-set-def* [*symmetric, code-unfold*]

```

declare fold-set-def
  [where  $A=RBT\text{-}Set.Set$   $r$  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  $c_1$   $c \Longrightarrow$  subcmd ( $c_1;;c_2$ )  $c$ 

```

```

| sub-Seq2: subcmd  $c_2$   $c \Longrightarrow$  subcmd ( $c_1;;c_2$ )  $c$ 

```

```

| sub-If1: subcmd  $c_1$   $c \Longrightarrow$  subcmd (if ( $b$ )  $c_1$  else  $c_2$ )  $c$ 

```

```

| sub-If2: subcmd  $c_2$   $c \Longrightarrow$  subcmd (if ( $b$ )  $c_1$  else  $c_2$ )  $c$ 

```

```

| 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 ( $e_1$  «  $-$  »  $e_2$ ) = max (maxVnameLen-aux  $e_1$ ) (maxVnameLen-aux  $e_2$ )

```

```

fun maxVnameLen :: cmd  $\Rightarrow$  nat where

```

```

  maxVnameLen Skip = 0

```

```

| maxVnameLen ( $V:=e$ ) = max (length  $V$ ) (maxVnameLen-aux  $e$ )

```

```

| maxVnameLen ( $c_1;;c_2$ ) = max (maxVnameLen  $c_1$ ) (maxVnameLen  $c_2$ )

```

```

| maxVnameLen (if ( $b$ )  $c_1$  else  $c_2$ ) = max (maxVnameLen  $c_1$ ) (max (maxVnameLen-aux  $b$ ) (maxVnameLen  $c_2$ ))

```

```

| 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**

$transform\text{-}aux\ \text{-}\ Skip = Skip$
 $| transform\text{-}aux\ V'\ (V:=e) =$
 $\quad (if\ V \in rhs\ (V:=e)\ then\ V':=e;;\ V:=Var\ V'$
 $\quad \quad else\ V:=e)$
 $| transform\text{-}aux\ V'\ (c1;;c2) = transform\text{-}aux\ V'\ c1;;\ transform\text{-}aux\ V'\ c2$
 $| transform\text{-}aux\ V'\ (if\ (b)\ c1\ else\ c2) =$
 $\quad (if\ (b)\ transform\text{-}aux\ V'\ c1\ else\ transform\text{-}aux\ V'\ c2)$
 $| transform\text{-}aux\ V'\ (while\ (b)\ c) = (while\ (b)\ transform\text{-}aux\ V'\ c)$

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 $leftmost\text{-}labels$: $labels\ c\ n\ c' \Longrightarrow subcmd\ c\ (leftmostCmd\ c')$
 $\langle proof \rangle$

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 *lhs* *c'* \cap

rhs *c'* = $\{\}$

<proof>

corollary *Defs-Uses-transform-disjoint* [*simp*]: *Defs* (*transform* *c*) $n \cap$ *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**

(-Entry-) $\leq x = \text{True}$

| *(- n -)* $\leq x = (\text{case } x \text{ of}$

(-Entry-) $\implies \text{False}$

| *(- m -)* $\implies n \leq m$

| *(-Exit-)* $\implies \text{True}$)

| *(-Exit-)* $\leq x = (x = (-Exit-))$

fun *less-w-node* **where**

(-Entry-) $< x = (x \neq (-Entry-))$

| *(- n -)* $< x = (\text{case } x \text{ of}$

(-Entry-) $\implies \text{False}$

| *(- m -)* $\implies n < m$

| *(-Exit-)* $\implies \text{True}$)

| *(-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* $\{(l,c). \text{labels prog } l \ c\}$
 ⟨*proof*⟩

lemma *finite-Defs* [*simp, intro!*]: *finite* (*Defs* *c* *n*)
 ⟨*proof*⟩

lemma *finite-Uses* [*simp, intro!*]: *finite* (*Uses* *c* *n*)
 ⟨*proof*⟩

definition *while-cfg- α e* *c* = *Collect* (*valid-edge* (*transform* *c*))

definition *while-cfg- α n* *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* \wedge *set ls* = $\{(sourcenode$ *e, kind* *e)| e. valid-edge* (*transform* *c*) *e* \wedge *targetnode* *e* = *t* $\}$)

definition *while-cfg-Entry* *c* = (*-Entry-*)

definition *while-cfg-defs* *c* = (*Defs* (*transform* *c*))(*-Entry-*) := $\{v. \exists n. v \in \text{Uses}$ (*transform* *c*) *n* $\}$

definition *while-cfg-uses* *c* = *Uses* (*transform* *c*)

abbreviation *while-cfg-inEdges* *c* *t* \equiv *map* ($\lambda(f,d). (f,d,t)$) (*while-cfg-inEdges'* *c* *t*)

lemmas *while-cfg-defs* = *while-cfg- α e-def* *while-cfg- α n-def*

while-cfg-invar-def *while-cfg-inEdges'-def*

while-cfg-Entry-def *while-cfg-defs-def*

while-cfg-uses-def

interpretation *while*: *graph-path* *while-cfg- α e* *while-cfg- α n* *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* $g \equiv$ (\mid
 gen- αe = *while-cfg- αe* g ,
 gen- αn = *while-cfg- αn* 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
 \mid)

lemma *while-path-graph-pathD: While-CFG.path* (*transform c*) $n \text{ es } m \implies \text{while.path2}$
 $c n (n\#\text{map targetnode es}) m$
<proof>

lemma *Uses-Entry [simp]: Uses c (-Entry-) = {}*
<proof>

lemma *in-Uses-valid-node: V \in Uses c n \implies valid-node c n*
<proof>

lemma *while-cfg-CFG-wf-impl:*
 SSA-CFG.CFG-wf ($\lambda-. \text{gen-}\alpha e (\text{gen-while-cfg cmd})$) ($\lambda-. \text{gen-}\alpha n (\text{gen-while-cfg}$
 cmd))
 ($\lambda-. \text{while-cfg-invar cmd}$) ($\lambda-. \text{gen-inEdges}' (\text{gen-while-cfg cmd})$)
 ($\lambda-. \text{gen-Entry} (\text{gen-while-cfg cmd})$) ($\lambda-. \text{gen-defs} (\text{gen-while-cfg cmd})$)
 ($\lambda-. \text{gen-uses} (\text{gen-while-cfg cmd})$)
<proof>

lift-definition *gen-while-cfg-wf* :: *cmd* \Rightarrow (*w-node, vname, state edge-kind*) *gen-cfg-wf*
 is *gen-while-cfg*
<proof>

definition *build-ssa cmd = gen-ssa-wf-notriv-substAll* (*gen-ssa-cfg-wf* (*gen-while-cfg-wf*
cmd))

end

References

- [1] G. Barthe, D. Demange, and D. Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, Mar. 2014.
- [2] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. Simple and efficient construction of static single assignment form. In R. Jhala and K. Bosschere, editors, *Compiler Construction*, volume 7791 of *Lecture Notes in Computer Science*, pages 102–122. Springer Berlin Heidelberg, 2013.