Verified Construction of Static Single Assignment Form

Sebastian Ullrich   Denis Lohner

May 26, 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

1.1 Miscellaneous Lemmata

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

lemma *length-1-last-hd*: $length\ ns = 1 \Longrightarrow last\ ns = hd\ ns$
⟨*proof*⟩

lemma *not-in-butlast*[*simp*]: $[\![x \in set\ ys; x \notin set\ (butlast\ ys)]\!] \Longrightarrow x = last\ ys$
⟨*proof*⟩

lemma *in-set-butlastI*: $x \in set\ xs \Longrightarrow x \neq last\ xs \Longrightarrow x \in set\ (butlast\ xs)$
⟨*proof*⟩

lemma *butlast-strict-prefix*: $xs \neq [] \Longrightarrow strict\text{-}prefix\ (butlast\ xs)\ xs$
⟨*proof*⟩

lemma *set-tl*: $set\ (tl\ xs) \subseteq set\ xs$
⟨*proof*⟩

lemma *in-set-tlD*[*elim*]: $x \in set\ (tl\ xs) \Longrightarrow x \in set\ xs$
⟨*proof*⟩

lemma *suffix-unsnoc*:
assumes $suffix\ xs\ ys\ xs \neq []$
obtains x **where** $xs = butlast\ xs@[x]\ ys = butlast\ ys@[x]$
⟨*proof*⟩

lemma *prefix-split-first*:
assumes $x \in set\ xs$
obtains as **where** $prefix\ (as@[x])\ xs$ **and** $x \notin set\ as$
⟨*proof*⟩

lemma *in-prefix*[*elim*]:
assumes $prefix\ xs\ ys$ **and** $x \in set\ xs$

shows $x \in set\ ys$
⟨*proof*⟩

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

lemma *prefix-tl-subset*: $prefix\ xs\ ys \Longrightarrow set\ (tl\ xs) \subseteq set\ (tl\ ys)$
⟨*proof*⟩

lemma *suffix-tl-subset*: $suffix\ xs\ ys \Longrightarrow set\ (tl\ xs) \subseteq set\ (tl\ ys)$
⟨*proof*⟩

lemma *set-tl-append'*: $set\ (tl\ (xs\ @\ ys)) \subseteq set\ (tl\ xs) \cup set\ ys$
⟨*proof*⟩

lemma *last-in-tl*: $length\ xs > 1 \Longrightarrow last\ xs \in set\ (tl\ xs)$
⟨*proof*⟩

lemma *concat-join*: $xs \neq [] \Longrightarrow ys \neq [] \Longrightarrow last\ xs = hd\ ys \Longrightarrow butlast\ xs@ys = xs@tl\ ys$
⟨*proof*⟩

lemma *fold-induct*[*case-names Nil Cons*]: $P\ s \Longrightarrow (\bigwedge x\ s.\ x \in set\ xs \Longrightarrow P\ s \Longrightarrow P\ (f\ x\ s)) \Longrightarrow P\ (fold\ f\ xs\ s)$
⟨*proof*⟩

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

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

lemma *fold-union-elemI'*:
assumes $x \in xs \lor (\exists xs \in set\ xss.\ x \in xs)$
shows $x \in fold\ (\cup)\ xss\ xs$
⟨*proof*⟩

lemma *fold-union-finite*[*intro*!]:
assumes $finite\ xs\ \forall xs \in set\ xss.\ finite\ xs$
shows $finite\ (fold\ (\cup)\ xss\ xs)$
⟨*proof*⟩

lemma *in-set-zip-map*:

assumes $(x,y) \in set\ (zip\ xs\ (map\ f\ ys))$
obtains y' **where** $(x,y') \in set\ (zip\ xs\ ys)$ $f\ y' = y$
$\langle proof \rangle$

lemma *dom-comp-subset*: $g\ `\ dom\ (f \circ g) \subseteq dom\ f$
$\langle proof \rangle$

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

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

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

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

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

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

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

lemma *singleton-list-hd-last*: $length\ xs = 1 \Longrightarrow hd\ xs = last\ xs$
$\langle proof \rangle$

lemma *distinct-hd-tl*: $distinct\ xs \Longrightarrow hd\ xs \notin set\ (tl\ xs)$
$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *set-butlast-distinct*: $distinct\ xs \Longrightarrow set\ (butlast\ xs) \cap \{last\ xs\} = \{\}$
$\langle proof \rangle$

lemma *disjoint-elem*[*elim*]: $A \cap B = \{\} \Longrightarrow x \in A \Longrightarrow x \notin B$ $\langle proof \rangle$

lemma *prefix-butlastD*[*elim*]: $prefix\ xs\ (butlast\ ys) \Longrightarrow prefix\ xs\ ys$
$\langle proof \rangle$

lemma *butlast-prefix*: $prefix\ xs\ ys \Longrightarrow prefix\ (butlast\ xs)\ (butlast\ ys)$
$\langle proof \rangle$

lemma *hd-in-butlast*: $length\ xs > 1 \Longrightarrow hd\ xs \in set\ (butlast\ xs)$
$\langle proof \rangle$

lemma *nonsimple-length-gt-1*: $xs \neq [] \Longrightarrow hd\ xs \neq last\ xs \Longrightarrow length\ xs > 1$
$\langle proof \rangle$

lemma *set-hd-tl*: $xs \neq [] \Longrightarrow set\ [hd\ xs] \cup set\ (tl\ xs) = set\ xs$
$\langle proof \rangle$

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

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

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

lemma *removeAll-induct*:
assumes $\bigwedge xs.\ (\bigwedge x.\ x \in set\ xs \Longrightarrow P\ (removeAll\ x\ xs)) \Longrightarrow P\ xs$
shows $P\ xs$
$\langle proof \rangle$

lemma *The-Min*: $Ex1\ P \Longrightarrow The\ P = Min\ \{x.\ P\ x\}$
$\langle proof \rangle$

lemma *The-Max*: $Ex1\ P \Longrightarrow The\ P = Max\ \{x.\ P\ x\}$
$\langle proof \rangle$

lemma *set-sorted-list-of-set-remove* [*simp*]:
$set\ (sorted\text{-}list\text{-}of\text{-}set\ (Set.remove\ x\ A)) = Set.remove\ x\ (set\ (sorted\text{-}list\text{-}of\text{-}set\ A))$
$\langle proof \rangle$

lemma *set-minus-one*: $[\![v \neq v';\ v' \in set\ vs]\!] \Longrightarrow set\ vs - \{v'\} \subseteq \{v\} \longleftrightarrow set\ vs = \{v'\} \lor set\ vs = \{v,v'\}$
$\langle proof \rangle$

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

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

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

end

1.2 Serial Relations

A serial relation on a finite carrier induces a cycle.

theory *Serial-Rel*
imports *Main*
begin

definition $serial\text{-}on\ A\ r \longleftrightarrow (\forall x \in A.\ \exists y \in A.\ (x,y) \in r)$
lemmas $serial\text{-}onI = serial\text{-}on\text{-}def[THEN\ iffD2,\ rule\text{-}format]$
lemmas $serial\text{-}onE = serial\text{-}on\text{-}def[THEN\ iffD1,\ rule\text{-}format,\ THEN\ bexE]$

fun *iterated-serial-on* :: $'a\ set \Rightarrow 'a\ rel \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$ **where**
$iterated\text{-}serial\text{-}on\ A\ r\ x\ 0 = x$
$|\ iterated\text{-}serial\text{-}on\ A\ r\ x\ (Suc\ n) = (SOME\ y.\ y \in A \wedge (iterated\text{-}serial\text{-}on\ A\ r\ x\ n,y) \in r)$

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

lemma *iterated-serial-on-in-A*:
assumes $serial\text{-}on\ A\ r\ a \in A$
shows $iterated\text{-}serial\text{-}on\ A\ r\ a\ n \in A$
⟨*proof*⟩

lemma *iterated-serial-on-in-power*:
assumes $serial\text{-}on\ A\ r\ a \in A$
shows $(a,iterated\text{-}serial\text{-}on\ A\ r\ a\ n) \in r\ \hat{}\hat{}\ n$
⟨*proof*⟩

lemma *trancl-powerI*: $a \in R\ \hat{}\hat{}\ n \Longrightarrow n > 0 \Longrightarrow a \in R^+$
⟨*proof*⟩

theorem *serial-on-finite-cycle*:
assumes $serial\text{-}on\ A\ r\ A \neq \{\}\ finite\ A$
obtains a **where** $a \in A\ (a,a) \in r^+$

$\langle proof \rangle$

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 $\lambda P\ m\ x.$ *if* $(P\ x)$ *then None else* $m\ x$ $\langle proof \rangle$
lift-definition *map-keys* :: $('a \Rightarrow 'b) \Rightarrow ('a,'c)$ *mapping* $\Rightarrow ('b,'c)$ *mapping*
  is $\lambda f\ m\ x.$ *if* $f -` \{x\} \neq \{\}$ *then* $m$ (*THE* $k.\ f -` \{x\} = \{k\}$) *else None* $\langle proof \rangle$
lift-definition *map-values* :: $('a \Rightarrow 'b \Rightarrow 'c$ *option*$) \Rightarrow ('a,'b)$ *mapping* $\Rightarrow ('a,'c)$ *mapping*
  is $\lambda f\ m\ x.$ *Option.bind* $(m\ x)\ (f\ x)$ $\langle proof \rangle$
lift-definition *restrict-mapping* :: $('a \Rightarrow 'b) \Rightarrow 'a$ *set* $\Rightarrow ('a,\ 'b)$ *mapping*
  is $\lambda f.$ *restrict-map* (*Some* $\circ f$) $\langle proof \rangle$
lift-definition *mapping-add* :: $('a,\ 'b)$ *mapping* $\Rightarrow ('a,\ 'b)$ *mapping* $\Rightarrow ('a,\ 'b)$ *mapping*
  is $(++)$ $\langle proof \rangle$

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*)
$\langle proof \rangle$

lemma *Mapping-Mapping-lookup* [*simp*, *code-unfold*]: *Mapping.Mapping* (*Mapping.lookup* m) = m $\langle proof \rangle$
declare *Mapping.lookup.abs-eq*[*simp*]

lemma *Mapping-eq-lookup*: $m = m' \longleftrightarrow$ *Mapping.lookup* m = *Mapping.lookup* m'
$\langle proof \rangle$

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*)
  $\langle proof \rangle$

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

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* (λ*n. Mapping.update n* (*g n*)) *xs m*) *x*
= (*if* (*x* ∈ *set xs*) *then Some* (*g x*) *else Mapping.lookup m x*)
⟨*proof*⟩

lemma *mapping-eq-iff*: *m1* = *m2* ⟷ (∀ *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*∈*Mapping.keys m. f k* (*the* (*Mapping.lookup m k*)) = *None*}
⟨*proof*⟩

lemma *map-default-eq*: *Mapping.map-default k v f m* = *m* ⟷ (∃ *v. Mapping.lookup m k* = *Some v* ∧ *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* (λ*a* -. *f a*) *r*)

⟨*proof*⟩

lemma *map-keys-code*:
assumes *inj f*
shows *map-keys f* (*RBT-Mapping.Mapping t*) = *RBT.fold* (λ*x 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* (λ*x v m*. *case* (*f x v*) *of None* $\Rightarrow$ *m* | *Some v'* $\Rightarrow$ *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* $\times$ '*ed* $\times$ '*n*)

definition *getFrom* :: ('*n*, '*ed*) *edge* $\Rightarrow$ '*n* **where**
getFrom $\equiv$ *fst*
definition *getData* :: ('*n*, '*ed*) *edge* $\Rightarrow$ '*ed* **where**
getData $\equiv$ *fst* $\circ$ *snd*
definition *getTo* :: ('*n*, '*ed*) *edge* $\Rightarrow$ '*n* **where**

$getTo \equiv snd \circ snd$

lemma *get-edge-simps* [*simp*]:
$getFrom\ (f,d,t) = f$
$getData\ (f,d,t) = d$
$getTo\ (f,d,t) = t$
⟨*proof*⟩

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*]: $finite\ (edges\ G) \Longrightarrow finite\ (pred\ G\ v)$
⟨*proof*⟩

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

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

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 $pred\text{-}it\ g\ v \equiv it\text{-}to\text{-}it\ (pred\text{-}list\text{-}it\ g\ v)$
end

locale *graph-pred-it* = *graph* α *invar* + *graph-pred-it-defs pred-list-it*
for α :: $'G \Rightarrow ('V,'W)$ *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*:
$invar\ g \Longrightarrow set\text{-}iterator\ (pred\text{-}list\text{-}it\ g\ v)\ (pred\ (\alpha\ g)\ v)$
begin
lemma *pred-it-correct*:
$invar\ g \Longrightarrow set\text{-}iterator\ (pred\text{-}it\ g\ v)\ (pred\ (\alpha\ g)\ v)$
⟨*proof*⟩

lemma *pi-pred-it*[*icf-proper-iteratorI*]:
$proper\text{-}it\ (pred\text{-}it\ S\ v)\ (pred\text{-}it\ S\ v)$
⟨*proof*⟩

lemma *pred-it-proper*[*proper-it*]:
$proper\text{-}it'\ (\lambda S.\ pred\text{-}it\ S\ v)\ (\lambda S.\ pred\text{-}it\ S\ v)$
⟨*proof*⟩
end

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

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

lemma (**in** *graph-pred-it*) *pred-it-is-iterator*[*refine-transfer*]:
invar g $\Longrightarrow$ *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 λg. *foldri* (αn *g*) +
graph-pred-it-defs λg *n*. *foldri* (*inEdges'* *g n*)
for
αe :: $'g \Rightarrow ('node \times 'edgeD \times 'node)$ *set* **and**
αn :: $'g \Rightarrow 'node$ *list* **and**
invar :: $'g \Rightarrow$ *bool* **and**
inEdges' :: $'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD)$ *list*
begin

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

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

definition *successors* :: $'g \Rightarrow 'node \Rightarrow 'node$ *list* **where**
successors g m ≡ [*n* . *n* ← αn *g*, *m* ∈ *set* (*predecessors g n*)]

declare *predecessors-def* [*code*]

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

definition *path2* :: $'g \Rightarrow 'node \Rightarrow 'node$ *list* $\Rightarrow 'node \Rightarrow$ *bool* (- ⊢ -–-→-

$[51,0,0,51]$ 80) **where**
  path2 g n ns m ≡ *path g ns* ∧ *n* = *hd ns* ∧ *m* = *last ns*

 abbreviation α *g* ≡ ⦇*nodes* = *set* (αn *g*), *edges* = αe *g*⦈
end

locale *graph-path* =
 graph-path-base αe αn *invar inEdges'* +
 graph α *invar* +
 ni: *graph-nodes-it* α *invar* λg. *foldri* (αn *g*) +
 pi: *graph-pred-it* α *invar* λg *n*. *foldri* (*inEdges' g n*)
for
 αe :: *'g* ⇒ (*'node* × *'edgeD* × *'node*) *set* **and**
 αn :: *'g* ⇒ *'node list* **and**
 invar :: *'g* ⇒ *bool* **and**
 inEdges' :: *'g* ⇒ *'node* ⇒ (*'node* × *'edgeD*) *list*
begin
 lemma αn*-correct*: *invar g* ⟹ *set* (αn *g*) ⊇ *getFrom* ` $\alpha e$ *g* ∪ *getTo* ` αe *g*
  ⟨*proof*⟩

 lemma αn*-distinct*: *invar g* ⟹ *distinct* (αn *g*)
  ⟨*proof*⟩

 lemma *inEdges-correct'*:
  assumes *invar g*
  shows *set* (*inEdges g n*) = (λ(*f*,*d*). (*f*,*d*,*n*)) ` (*pred* (α *g*) *n*)
 ⟨*proof*⟩

 lemma *inEdges-correct* [*intro*!, *simp*]:
  invar g ⟹ *set* (*inEdges g n*) = {(-, -, *t*). *t* = *n*} ∩ αe *g*
  ⟨*proof*⟩

 lemma *in-set-*αn*I1* [*intro*]: ⟦*invar g*; *x* ∈ *getFrom* ` αe *g*⟧ ⟹ *x* ∈ *set* (αn *g*)
  ⟨*proof*⟩
 lemma *in-set-*αn*I2* [*intro*]: ⟦*invar g*; *x* ∈ *getTo* ` αe *g*⟧ ⟹ *x* ∈ *set* (αn *g*)
  ⟨*proof*⟩

 lemma *edge-to-node*:
  assumes *invar g* **and** *e* ∈ αe *g*
  obtains *getFrom e* ∈ *set* (αn *g*) **and** *getTo e* ∈ *set* (αn *g*)
 ⟨*proof*⟩

 lemma *inEdge-to-edge*:
  assumes *e* ∈ *set* (*inEdges g n*) **and** *invar g*
  obtains *eD n'* **where** (*n'*,*eD*,*n*) ∈ αe *g*

⟨*proof*⟩

lemma *edge-to-inEdge*:
assumes $(n,eD,m) \in \alpha e\ g$ *invar g*
obtains $(n,eD,m) \in set\ (inEdges\ g\ m)$
⟨*proof*⟩

lemma *edge-to-predecessors*:
assumes $(n,eD,m) \in \alpha e\ g$ *invar g*
obtains $n \in set\ (predecessors\ g\ m)$
⟨*proof*⟩

lemma *predecessor-is-node*[*elim*]: $[\![n \in set\ (predecessors\ g\ n'); invar\ g]\!] \Longrightarrow n \in set\ (\alpha n\ g)$
⟨*proof*⟩

lemma *successor-is-node*[*elim*]: $[\![n \in set\ (predecessors\ g\ n'); n \in set\ (\alpha n\ g); invar\ g]\!] \Longrightarrow n' \in set\ (\alpha n\ g)$
⟨*proof*⟩

lemma *successors-predecessors*[*simp*]: $n \in set\ (\alpha n\ g) \Longrightarrow n \in set\ (successors\ g\ m) \longleftrightarrow m \in set\ (predecessors\ g\ n)$
⟨*proof*⟩

lemma *path-not-Nil*[*simp, dest*]: $path\ g\ ns \Longrightarrow ns \neq []$
⟨*proof*⟩

lemma *path2-not-Nil*[*simp*]: $g \vdash n{-}ns{\rightarrow}m \Longrightarrow ns \neq []$
⟨*proof*⟩

lemma *path2-not-Nil2*[*simp*]: $\neg\ g \vdash n{-}[]{\rightarrow}m$
⟨*proof*⟩

lemma *path2-not-Nil3*[*simp*]: $g \vdash n{-}ns{\rightarrow}m \Longrightarrow length\ ns \geq 1$
⟨*proof*⟩

lemma *empty-path2*[*intro*]: $[\![n \in set\ (\alpha n\ g); invar\ g]\!] \Longrightarrow g \vdash n{-}[n]{\rightarrow}n$
⟨*proof*⟩

lemma *Cons-path2*[*intro*]: $[\![g \vdash n{-}ns{\rightarrow}m; n' \in set\ (predecessors\ g\ n)]\!] \Longrightarrow g \vdash n'{-}n'\#ns{\rightarrow}m$
⟨*proof*⟩

lemma *path2-cases*:
assumes $g \vdash n{-}ns{\rightarrow}m$
obtains (*empty-path*) $ns = [n]$ $m = n$
| (*Cons-path*) $g \vdash hd\ (tl\ ns){-}tl\ ns{\rightarrow}m$ $n \in set\ (predecessors\ g\ (hd\ (tl\ ns)))$
⟨*proof*⟩

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

lemma *path-invar*[*intro*]: $path\ g\ ns \Longrightarrow invar\ g$
⟨*proof*⟩

lemma *path-in-αn*[*intro*]: $[\![path\ g\ ns;\ n \in set\ ns]\!] \Longrightarrow n \in set\ (\alpha n\ g)$
⟨*proof*⟩

lemma *path2-in-αn*[*elim*]: $[\![g \vdash n{-}ns{\rightarrow}m;\ l \in set\ ns]\!] \Longrightarrow l \in set\ (\alpha n\ g)$
⟨*proof*⟩

lemma *path2-hd-in-αn*[*elim*]: $g \vdash n{-}ns{\rightarrow}m \Longrightarrow n \in set\ (\alpha n\ g)$
⟨*proof*⟩

lemma *path2-tl-in-αn*[*elim*]: $g \vdash n{-}ns{\rightarrow}m \Longrightarrow m \in set\ (\alpha n\ g)$
⟨*proof*⟩

lemma *path2-forget-hd*[*simp*]: $g \vdash n{-}ns{\rightarrow}m \Longrightarrow g \vdash hd\ ns{-}ns{\rightarrow}m$
⟨*proof*⟩

lemma *path2-forget-last*[*simp*]: $g \vdash n{-}ns{\rightarrow}m \Longrightarrow g \vdash n{-}ns{\rightarrow}last\ ns$
⟨*proof*⟩

lemma *path-hd*[*dest*]: $path\ g\ (n\#ns) \Longrightarrow path\ g\ [n]$
⟨*proof*⟩

lemma *path-by-tail*[*intro*]: $[\![path\ g\ (n\#n'\#ns);\ path\ g\ (n'\#ns) \Longrightarrow path\ g\ (n'\#ms)]\!] \Longrightarrow path\ g\ (n\#n'\#ms)$
⟨*proof*⟩

lemma *αn-in-αnE* [*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 $g \vdash n{-}ns@[n']{\to}n'$ $g \vdash n'{-}n'\#ns'{\to}m$
⟨*proof*⟩

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

lemma *path2-split-ex*:
assumes $g \vdash n{-}ns{\to}m$ $x \in set\ ns$
obtains ns_1 ns_2 **where** $g \vdash n{-}ns_1{\to}x$ $g \vdash x{-}ns_2{\to}m$ $ns = ns_1@tl\ ns_2$ $ns = butlast\ ns_1@ns_2$
⟨*proof*⟩

lemma *path2-split-ex'*:
assumes $g \vdash n{-}ns{\to}m$ $x \in set\ ns$
obtains ns_1 ns_2 **where** $g \vdash n{-}ns_1{\to}x$ $g \vdash x{-}ns_2{\to}m$ $ns = butlast\ ns_1@ns_2$
⟨*proof*⟩

lemma *path-snoc*:
assumes $path\ g\ (ns@[n])$ $n \in set\ (predecessors\ g\ m)$
shows $path\ g\ (ns@[n,m])$
⟨*proof*⟩

lemma *path2-snoc*[*elim*]:
assumes $g \vdash n{-}ns{\to}m$ $m \in set\ (predecessors\ g\ m')$
shows $g \vdash n{-}ns@[m']{\to}m'$
⟨*proof*⟩

lemma *path-unsnoc*:
assumes $path\ g\ ns$ $length\ ns \geq 2$
obtains $path\ g\ (butlast\ ns) \wedge last\ (butlast\ ns) \in set\ (predecessors\ g\ (last\ ns))$
⟨*proof*⟩

lemma *path2-unsnoc*:
assumes $g \vdash n{-}ns{\to}m$ $length\ ns \geq 2$
obtains $g \vdash n{-}butlast\ ns{\to}last\ (butlast\ ns)$ $last\ (butlast\ ns) \in set\ (predecessors\ g\ m)$
⟨*proof*⟩

lemma *path2-rev-induct*[*consumes 1*, *case-names empty snoc*]:
assumes $g \vdash n{-}ns{\to}m$
assumes *empty*: $n \in set\ (\alpha n\ g) \Longrightarrow P\ n\ [n]\ n$
assumes *snoc*: $\bigwedge ns\ m'\ m.\ g \vdash n{-}ns{\to}m' \Longrightarrow P\ n\ ns\ m' \Longrightarrow m' \in set\ (predecessors\ g\ m) \Longrightarrow P\ n\ (ns@[m])\ m$
shows $P\ n\ ns\ m$
⟨*proof*⟩

lemma *path2-hd*[*elim*, *dest?*]: $g \vdash n{-}ns{\to}m \Longrightarrow n = hd\ ns$
⟨*proof*⟩

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

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

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

lemma *path-app*[*elim*]:
assumes $path\ g\ ns\ path\ g\ ms\ last\ ns = hd\ ms$
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 $last\ xs = hd\ ys\ xs \neq []\ ys \neq []$
shows $butlast\ xs@ys = xs@tl\ ys$
⟨*proof*⟩

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

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

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

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

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

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

lemma *path2-split-first-last*:
assumes $g \vdash n{-}ns{\rightarrow}m$ $x \in set\ ns$
obtains ns_1 ns_3 ns_2 **where** $ns = ns_1@ns_3@ns_2$ $prefix\ (ns_1@[x])\ ns$ $suffix\ (x\#ns_2)\ ns$
and $g \vdash n{-}ns_1@[x]{\rightarrow}x$ $x \notin set\ ns_1$
and $g \vdash x{-}ns_3{\rightarrow}x$
and $g \vdash x{-}x\#ns_2{\rightarrow}m$ $x \notin set\ ns_2$
⟨*proof*⟩

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

lemma *path2-split-first-prop*:
assumes $g \vdash n{-}ns{\rightarrow}m$ $\exists\, x{\in}set\ ns.\ P\ x$
obtains m' ns' **where** $g \vdash n{-}ns'{\rightarrow}m'$ $P\ m'$ $\forall\, x \in set\ (butlast\ ns').\ \neg P\ x$ $prefix\ ns'\ ns$
⟨*proof*⟩

lemma *path2-split-last-prop*:
assumes $g \vdash n{-}ns{\rightarrow}m$ $\exists\, x{\in}set\ ns.\ P\ x$
obtains n' ns' **where** $g \vdash n'{-}ns'{\rightarrow}m$ $P\ n'$ $\forall\, x \in set\ (tl\ ns').\ \neg P\ x$ $suffix\ ns'\ ns$
⟨*proof*⟩

lemma *path2-prefix*[*elim*]:
assumes *1*: $g \vdash n{-}ns{\rightarrow}m$
assumes *2*: $prefix\ (ns'@[m'])\ ns$
shows $g \vdash n{-}ns'@[m']{\rightarrow}m'$
⟨*proof*⟩

lemma *path2-prefix-ex*:
assumes $g \vdash n{-}ns{\rightarrow}m$ $m' \in set\ ns$
obtains ns' **where** $g \vdash n{-}ns'{\rightarrow}m'$ $prefix\ ns'\ ns$ $m' \notin set\ (butlast\ ns')$
⟨*proof*⟩

lemma *path2-strict-prefix-ex*:
assumes $g \vdash n{-}ns{\rightarrow}m$ $m' \in set\ (butlast\ ns)$
obtains ns' **where** $g \vdash n{-}ns'{\rightarrow}m'$ $strict\text{-}prefix\ ns'\ ns$ $m' \notin set\ (butlast\ ns')$
⟨*proof*⟩

lemma *path2-nontriv*[*elim*]: $[\![g \vdash n{-}ns{\rightarrow}m;\ n \neq m]\!] \Longrightarrow length\ ns > 1$
⟨*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* αe αn *invar* *inEdges'*
for
αe :: $'g \Rightarrow ('node \times 'edgeD \times 'node)$ *set* **and**
αn :: $'g \Rightarrow 'node$ *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* αe αn *invar* *inEdges'* *Entry*
+ *graph-path* αe αn *invar* *inEdges'*
for
αe :: $'g \Rightarrow ('node \times 'edgeD \times 'node)$ *set* **and**
αn :: $'g \Rightarrow 'node$ *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* $g \Longrightarrow$ *inEdges* g (*Entry* g) = []
assumes *Entry-reaches*[*intro*]:
$[\![n \in$ *set* $(\alpha n\ g)$; *invar* $g]\!] \Longrightarrow \exists ns.\ g \vdash$ *Entry* $g{-}ns{\rightarrow}n$
begin
lemma *Entry-dominates*[*simp,intro*]: $[\![$*invar* g; $n \in$ *set* $(\alpha n\ g)]\!] \Longrightarrow$ *dominates* g (*Entry* g) n
⟨*proof*⟩

lemma *Entry-iff-unreachable*[*simp*]:
assumes *invar* g $n \in$ *set* $(\alpha n\ g)$
shows *predecessors* g n = [] $\longleftrightarrow$ 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 $invar\ g\ n \in set\ (\alpha n\ g)$
obtains ns **where** $g \vdash Entry\ g{-}ns{\rightarrow}n$ **and** $n \notin set\ (butlast\ ns)$
⟨*proof*⟩

lemma *dominatesI* [*intro*]:
$[\![m \in set\ (\alpha n\ g);\ \bigwedge ns.\ [\![g \vdash Entry\ g{-}ns{\rightarrow}m]\!] \Longrightarrow n \in set\ ns]\!] \Longrightarrow dominates\ g\ n\ m$
⟨*proof*⟩

lemma *dominatesE*:
assumes $dominates\ g\ n\ m$
obtains $m \in set\ (\alpha n\ g)$ **and** $\bigwedge ns.\ g \vdash Entry\ g{-}ns{\rightarrow}m \Longrightarrow n \in set\ ns$
⟨*proof*⟩

lemma[*simp*]: $dominates\ g\ n\ m \Longrightarrow m \in set\ (\alpha n\ g)$ ⟨*proof*⟩

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

lemma *strict-domE*[*elim*]:
assumes $strict\text{-}dom\ g\ n\ m$
obtains $m \in set\ (\alpha n\ g)$ **and** $\bigwedge ns.\ g \vdash Entry\ g{-}ns{\rightarrow}m \Longrightarrow n \in set\ (butlast\ ns)$
⟨*proof*⟩

lemma *dominates-refl*[*intro*!]: $[\![invar\ g;\ n \in set\ (\alpha n\ g)]\!] \Longrightarrow dominates\ g\ n\ n$
⟨*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** $dominates\ g\ n\ m\ m' \in set\ (predecessors\ g\ m)\ n$

$\neq m$
shows *dominates* $g\ n\ m'$
⟨*proof*⟩

lemma *dominates-unsnoc*$'$:
assumes [*simp*]: *invar* g **and** *dominates* $g\ n\ m\ g \vdash m'{-}ms{\rightarrow}m\ \forall x \in set\ (tl\ ms).\ x \neq n$
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 $g \vdash m'{-}ms{\rightarrow}m\ n \notin set\ (tl\ ms)$
shows *dominates* $g\ n\ m'$
⟨*proof*⟩

definition *dominators* :: $'g \Rightarrow 'node \Rightarrow 'node\ set$ **where**
dominators $g\ n \equiv \{m \in set\ (\alpha n\ g).\ dominates\ g\ m\ n\}$

definition *isIdom* $g\ n\ m \longleftrightarrow strict\text{-}dom\ g\ m\ n \land (\forall m' \in set\ (\alpha n\ g).\ strict\text{-}dom\ g\ m'\ n \longrightarrow dominates\ g\ m'\ m)$
definition *idom* :: $'g \Rightarrow 'node \Rightarrow 'node$ **where**
idom $g\ n \equiv THE\ m.\ isIdom\ g\ n\ m$

lemma *idom-ex*:
assumes[*simp*]: *invar* $g\ n \in set\ (\alpha n\ g)\ n \neq Entry\ g$
shows $\exists!m.\ isIdom\ g\ n\ m$
⟨*proof*⟩

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

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

definition *shortestPath* :: $'g \Rightarrow 'node \Rightarrow nat$ **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*]: $[\![n \in set\ (\alpha n\ g);\ invar\ g]\!] \Longrightarrow shortestPath\ g\ n \neq 0$
⟨*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 $strict\text{-}dom\ g\ n\ m$ $m \in set\ (\alpha n\ g)$ $invar\ g$
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 $strict\text{-}dom\ g\ n\ m$ $strict\text{-}dom\ g\ m\ m'$
shows $strict\text{-}dom\ g\ n\ m'$
⟨*proof*⟩

inductive $EntryPath :: 'g \Rightarrow 'node\ list \Rightarrow bool$ **where**
EntryPath-triv[*simp*]: $EntryPath\ g\ [n]$

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

lemma[*simp*]:
  assumes *EntryPath g ns prefix ns′ ns ns′* $\neq$ []
  shows *EntryPath g ns′*
⟨*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* (αn *g*) **and**[*simp*]: *invar g*
  obtains *ns* **where** *g* $\vdash$ *Entry g*−*ns*→*n EntryPath g ns*
⟨*proof*⟩
end

end

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

2.3 CFG

locale *CFG-base* = *graph-Entry-base* αe αn *invar inEdges′ Entry*
for
  αe :: *′g* ⇒ (*′node*::*linorder* × *′edgeD* × *′node*) *set* **and**
  αn :: *′g* ⇒ *′node list* **and**
  invar :: *′g* ⇒ *bool* **and**
  inEdges′ :: *′g* ⇒ *′node* ⇒ (*′node* × *′edgeD*) *list* **and**
  Entry :: *′g* ⇒ *′node* +
fixes *defs* :: *′g* ⇒ *′node* ⇒ *′var*::*linorder set*
fixes *uses* :: *′g* ⇒ *′node* ⇒ *′var set*
begin
  definition *vars g* ≡ *fold* (∪) (*map* (*uses g*) (αn *g*)) {}

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

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 $defs\text{-}uses\text{-}disjoint$: $n \in set\ (\alpha n\ g) \Longrightarrow defs\ g\ n \cap uses\ g\ n = \{\}$
assumes $defs\text{-}finite[simp]$: $finite\ (defs\ g\ n)$
assumes $uses\text{-}in\text{-}\alpha n$: $v \in uses\ g\ n \Longrightarrow n \in set\ (\alpha n\ g)$
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 $uses\text{-}in\text{-}vars[elim,\ simp]$: $v \in uses\ g\ n \Longrightarrow\ v \in vars\ g$
$\langle proof \rangle$

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 $defs\text{-}uses\text{-}disjoint'[simp]$: $n \in set\ (\alpha n\ g) \Longrightarrow v \in defs\ g\ n \Longrightarrow v \in uses\ g\ n \Longrightarrow False$
$\langle proof \rangle$
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* αe αn *invar inEdges' Entry defs uses*
for
αe :: $'g \Rightarrow ('node{::}linorder \times 'edgeD \times 'node)\ set$ **and**
αn :: $'g \Rightarrow 'node\ list$ **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 *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 $('node, 'val)\ phis = 'node \times 'val \rightharpoonup 'val\ list$

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

locale *CFG-SSA-base* = *CFG-base* αe αn *invar inEdges' Entry defs uses*
for
αe :: $'g \Rightarrow ('node{::}linorder \times 'edgeD \times 'node)\ set$ **and**
αn :: $'g \Rightarrow 'node\ list$ **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 :: $'g \Rightarrow 'node \Rightarrow 'val\ set$ +
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* :: $'g \Rightarrow 'node \Rightarrow 'val \Rightarrow bool$ **where**

defAss g m v $\longleftrightarrow$ ($\forall$ *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 allVars-def defAss-def*
end

locale *CFG-SSA* = *CFG* αe αn *invar inEdges' Entry defs uses* + *CFG-SSA-base* αe αn *invar inEdges' Entry defs uses phis*
for
αe :: *'g* $\Rightarrow$ (*'node*::*linorder* $\times$ *'edgeD* $\times$ *'node*) *set* **and**
αn :: *'g* $\Rightarrow$ *'node list* **and**
invar :: *'g* $\Rightarrow$ *bool* **and**
inEdges' :: *'g* $\Rightarrow$ *'node* $\Rightarrow$ (*'node* $\times$ *'edgeD*) *list* **and**
Entry::*'g* $\Rightarrow$ *'node* **and**
defs :: *'g* $\Rightarrow$ *'node* $\Rightarrow$ *'val*::*linorder set* **and**
uses :: *'g* $\Rightarrow$ *'node* $\Rightarrow$ *'val set* **and**
phis :: *'g* $\Rightarrow$ (*'node*, *'val*) *phis* +
assumes *phis-finite*: *finite* (*dom* (*phis g*))
assumes *phis-in-*αn: *phis g* (*n*,*v*) = *Some vs* $\Longrightarrow$ *n* $\in$ *set* (αn *g*)
assumes *phis-wf*:
phis g (*n*,*v*) = *Some args* $\Longrightarrow$ *length* (*predecessors g n*) = *length args*
assumes *simpleDefs-phiDefs-disjoint*:
n $\in$ *set* (αn *g*) $\Longrightarrow$ *defs g n* $\cap$ *phiDefs g n* = {}
assumes *allDefs-disjoint*:
$[\![$*n* $\in$ *set* (αn *g*); *m* $\in$ *set* (αn *g*); *n* $\neq$ *m*$]\!]$ $\Longrightarrow$ *allDefs g n* $\cap$ *allDefs g m* = {}
begin
lemma *phis-disj*:
assumes *phis g* (*n*,*v*) = *Some vs*
and *phis g* (*n'*,*v*) = *Some vs'*
shows *n* = *n'* **and** *vs* = *vs'*
$\langle$*proof*$\rangle$

lemma *allDefs-disjoint'*: $[\![$*n* $\in$ *set* (αn *g*); *m* $\in$ *set* (αn *g*); *v* $\in$ *allDefs g n*; *v* $\in$ *allDefs g m*$]\!]$ $\Longrightarrow$ *n* = *m*
$\langle$*proof*$\rangle$

lemma *phiUsesI*:
assumes *n'* $\in$ *set* (αn *g*) *phis g* (*n'*,*v'*) = *Some vs* (*n*,*v*) $\in$ *set* (*zip* (*predecessors g n'*) *vs*)
shows *v* $\in$ *phiUses g n*
$\langle$*proof*$\rangle$

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

lemma *defs-in-allDefs*[*simp*]: $v \in$ *defs g n* $\Longrightarrow v \in$ *allDefs g n* ⟨*proof*⟩
lemma *phiDefs-in-allDefs*[*simp*, *elim*]: $v \in$ *phiDefs g n* $\Longrightarrow v \in$ *allDefs g n* ⟨*proof*⟩
lemma *uses-in-allUses*[*simp*]: $v \in$ *uses g n* $\Longrightarrow v \in$ *allUses g n* ⟨*proof*⟩
lemma *phiUses-in-allUses*[*simp*]: $v \in$ *phiUses g n* $\Longrightarrow v \in$ *allUses g n* ⟨*proof*⟩
lemma *allDefs-in-allVars*[*simp*, *intro*]: $[\![v \in$ *allDefs g n*; $n \in$ *set* $(\alpha n\ g)]\!] \Longrightarrow v \in$ *allVars g* ⟨*proof*⟩
lemma *allUses-in-allVars*[*simp*, *intro*]: $[\![v \in$ *allUses g n*; $n \in$ *set* $(\alpha n\ g)]\!] \Longrightarrow v \in$ *allVars g* ⟨*proof*⟩

lemma *phiDefs-finite*[*simp*]: *finite* (*phiDefs g n*)
⟨*proof*⟩

lemma *phiUses-finite*[*simp*]:
assumes $n \in$ *set* $(\alpha n\ g)$
shows *finite* (*phiUses g n*)
⟨*proof*⟩

lemma *allDefs-finite*[*simp*]: $n \in$ *set* $(\alpha n\ g) \Longrightarrow$ *finite* (*allDefs g n*) ⟨*proof*⟩
lemma *allUses-finite*[*simp*]: $n \in$ *set* $(\alpha n\ g) \Longrightarrow$ *finite* (*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 $g \vdash n{-}ns{\rightarrow}m\ \forall n \in$ *set* (*tl ns*). $v \notin$ *allDefs g n*
shows *defAss g n v*
⟨*proof*⟩

lemma *defAss-dominating*:
assumes[*simp*]: $n \in$ *set* $(\alpha n\ g)$
shows *defAss g n v* $\longleftrightarrow$ $(\exists m \in$ *set* $(\alpha n\ g)$. *dominates g m n* $\wedge\ v \in$ *allDefs g m*)
⟨*proof*⟩
end

locale *CFG-SSA-wf-base* = *CFG-SSA-base* αe αn *invar inEdges' Entry defs uses phis*
for
αe :: *'g* ⇒ (*'node*::*linorder* × *'edgeD* × *'node*) *set* **and**
αn :: *'g* ⇒ *'node list* **and**
invar :: *'g* ⇒ *bool* **and**
inEdges' :: *'g* ⇒ *'node* ⇒ (*'node* × *'edgeD*) *list* **and**
Entry::*'g* ⇒ *'node* **and**
defs :: *'g* ⇒ *'node* ⇒ *'val*::*linorder set* **and**
uses :: *'g* ⇒ *'node* ⇒ *'val set* **and**
phis :: *'g* ⇒ (*'node*, *'val*) *phis*

begin

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

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

abbreviation $def\text{-}dominates\ g\ v'\ v \equiv dominates\ g\ (defNode\ g\ v')\ (defNode\ g\ v)$
abbreviation $strict\text{-}def\text{-}dom\ g\ v'\ v \equiv defNode\ g\ v' \neq defNode\ g\ v \wedge def\text{-}dominates\ g\ v'\ v$

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

definition$[code]$: $isTrivialPhi\ g\ v\ v' \longleftrightarrow v' \neq v \wedge$
$(case\ phi\ g\ v\ of$
$Some\ vs \Rightarrow set\ vs = \{v, v'\} \vee set\ vs = \{v'\}$
$\mid None \Rightarrow False)$
definition$[code]$: $trivial\ g\ v \equiv \exists\, v' \in allVars\ g.\ isTrivialPhi\ g\ v\ v'$
definition$[code]$: $redundant\ g \equiv \exists\, v \in allVars\ g.\ trivial\ g\ v$

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

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

declare $[[inductive\text{-}internals]]$
inductive $liveVal :: 'g \Rightarrow 'val \Rightarrow bool$
for $g :: 'g$
where
$liveSimple\text{: } [\![n \in set\ (\alpha n\ g);\ val \in uses\ g\ n]\!] \Longrightarrow liveVal\ g\ val$
$\mid livePhi\text{: } [\![liveVal\ g\ v;\ phiArg\ g\ v\ v']\!] \Longrightarrow liveVal\ g\ v'$

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

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

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

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

lemma *phiDefs-Entry-empty*[*simp*]: $phiDefs\ g\ (Entry\ g) = \{\}$
⟨*proof*⟩

lemma *phi-Entry-empty*[*simp*]: $defNode\ g\ v = Entry\ g \Longrightarrow phi\ g\ v = None$
⟨*proof*⟩

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

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

lemma *defNode*[*simp*]:
assumes $v \in allVars\ g$
shows $(defNode\ g\ v) \in set\ (\alpha n\ g)\ v \in allDefs\ g\ (defNode\ g\ v)$
⟨*proof*⟩

lemma *defNode-eq*[*intro*]:
assumes $n \in set\ (\alpha n\ g)\ v \in allDefs\ g\ n$
shows $defNode\ g\ v = n$
⟨*proof*⟩

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

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

lemma *simpleDef-not-phi*:
assumes $n \in set\ (\alpha n\ g)\ v \in defs\ g\ n$
shows $phi\ g\ v = None$
⟨*proof*⟩

lemma *phi-wf*: $phi\ g\ v = Some\ vs \Longrightarrow length\ (predecessors\ g\ (defNode\ g\ v)) = length\ vs$
⟨*proof*⟩

lemma *phi-finite*: *finite* (*dom* (*phi* *g*))
⟨*proof*⟩

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

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

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

lemma *phiArg-in-allVars*[*simp*]:
assumes $phiArg\ g\ v\ v'$
shows $v' \in allVars\ g$
⟨*proof*⟩

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

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

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

lemma *allUses-dominated*:
assumes $v \in allUses\ g\ n$ $n \in set\ (\alpha n\ g)$
shows $dominates\ g\ (defNode\ g\ v)\ n$
⟨*proof*⟩

lemma *phiArg-path-ex'*:
assumes $phiArg\ g\ p\ q$ **and**[*simp*]: $p \in allVars\ g$
obtains $ns\ m$ **where** $g \vdash defNode\ g\ q{-}ns{\rightarrow}m$ $EntryPath\ g\ ns$ $q \in phiUses\ g\ m$

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

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

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

lemma *non-dominated-predecessor*:
assumes $n \in set\ (\alpha n\ g)$ $n \neq Entry\ g$
obtains *m* **where** $m \in set\ (predecessors\ g\ n)$ $\neg dominates\ g\ n\ m$
⟨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*]: $liveVal\ g\ v \Longrightarrow v \in allVars\ g$
⟨proof⟩

lemma *phi-no-closed-loop*:
assumes[*simp*]: $p \in allVars\ g$ **and** $phi\ g\ p = Some\ vs$
shows $set\ vs \neq \{p\}$
⟨proof⟩

lemma *phis-phi*: $phis\ g\ (n,\ v) = Some\ vs \Longrightarrow phi\ g\ v = Some\ vs$
⟨proof⟩

lemma *trivial-phi*: $trivial\ g\ v \Longrightarrow phi\ g\ v \neq None$
⟨proof⟩

lemma *trivial-finite*: $finite\ \{v.\ trivial\ g\ v\}$
⟨proof⟩

lemma *trivial-in-allVars*: $trivial\ g\ v \Longrightarrow v \in allVars\ g$
⟨proof⟩

declare *phiArg-def* [*simp del*]
end

2.5 Bundling of CFG and Equivalent SSA CFG

locale *CFG-SSA-Transformed-base* = *old*: *CFG-base* αe αn *invar* *inEdges'* *Entry* *oldDefs* *oldUses* + *CFG-SSA-wf-base* αe αn *invar* *inEdges'* *Entry* *defs* *uses* *phis*
for
  αe :: $'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)$ *set* **and**
  αn :: $'g \Rightarrow 'node$ *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* αe αn *invar* *inEdges'* *Entry* *oldDefs* *oldUses* *defs* *uses* *phis* *var*
  + *old*: *CFG-wf* αe αn *invar* *inEdges'* *Entry* *oldDefs* *oldUses* + *CFG-SSA-wf* αe αn *invar* *inEdges'* *Entry* *defs* *uses* *phis*
for
  αe :: $'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)$ *set* **and**
  αn :: $'g \Rightarrow 'node$ *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* (αn *g*) $\Longrightarrow$ *oldUses g n* = *var g* ‘ *uses g n*
  assumes *conventional*:
$[\![g \vdash n{-}ns{\rightarrow}m;\ n \notin set\ (tl\ ns);\ v \in allDefs\ g\ n;\ v \in allUses\ g\ m;\ x \in set\ (tl\ ns);\ v' \in allDefs\ g\ x]\!] \Longrightarrow var\ g\ v' \neq var\ g\ v$
  assumes *phis-same-var*[*elim*]: $phis\ g\ (n,v) = Some\ vs \Longrightarrow v' \in set\ vs \Longrightarrow var\ g\ v' = var\ g\ v$
  assumes *allDefs-var-disjoint*: $[\![n \in set\ (\alpha n\ g);\ v \in allDefs\ g\ n;\ v' \in allDefs\ g\ n;\ v \neq v']\!] \Longrightarrow var\ g\ v' \neq var\ g\ v$
begin
  lemma *conventional'*: $[\![g \vdash n{-}ns{\rightarrow}m;\ n \notin set\ (tl\ ns);\ v \in allDefs\ g\ n;\ v \in allUses\ g\ m;\ v' \in allDefs\ g\ x;\ var\ g\ v' = var\ g\ v]\!] \Longrightarrow x \notin set\ (tl\ ns)$
    ⟨*proof*⟩

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

$\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

$\langle proof \rangle$

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

Simplified definition

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

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

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

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*: $old.pathsConverge\ g\ n\ ns\ m\ ms\ z \Longrightarrow v \in oldDefs\ g\ n \Longrightarrow v \in oldDefs\ g\ m \Longrightarrow necessaryPhi\ g\ v\ z$
⟨*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 $\neg(def\text{-}dominates\ g\ q\ p \wedge def\text{-}dominates\ g\ r\ p)$
⟨*proof*⟩

lemma *convergence-prop*:
assumes $necessaryPhi\ g\ (var\ g\ v)\ n\ g \vdash n{-}ns{\rightarrow}m\ v \in allUses\ g\ m\ \bigwedge x.\ x \in set\ (tl\ ns) \Longrightarrow v \notin allDefs\ g\ x\ v \notin defs\ g\ n$
shows $phis\ g\ (n,v) \neq None$
⟨*proof*⟩

lemma *convergence-prop'*:
assumes $necessaryPhi\ g\ v\ n\ g \vdash n{-}ns{\rightarrow}m\ v \in var\ g\ `\ allUses\ g\ m\ \bigwedge x.\ x \in set\ ns \Longrightarrow v \notin oldDefs\ g\ x$
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 *ununnecessaryPhis-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 $g \vdash n_1{-}ns_1{\to}$*defNode g r* **and**
var g r $\in$ *oldDefs g* n_2 $g \vdash n_2{-}ns_2{\to}$*defNode g r* **and**
set (*butlast* ns_1) $\cap$ *set* (*butlast* ns_2) = {}
⟨*proof*⟩

lemma *ununnecessaryPhis-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* $g \vdash n{-}ns{\to}$*defNode g r* **and** $V \in$ *oldDefs g m* $g \vdash m{-}ms{\to}$*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* αe αn *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 $x < y \longleftrightarrow x = SimpleDef \wedge y = PhiDef$
definition *less-eq-Def* (x :: *Def*) $y \longleftrightarrow x = y \vee x < y$
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* ⇒ *'node* **where** *defNode v* ≡ *fst* (*snd v*)
abbreviation *defKind* :: (*'node*, *'var*) *ssaVal* ⇒ *Def* **where** *defKind v* ≡ *snd* (*snd v*)

declare *var-def*[*simp*]

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

⟨*proof*⟩
termination ⟨*proof*⟩
declare *lookupDef.simps* [*code*]

definition *defs'* :: *'g* ⇒ *'node* ⇒ (*'node*, *'var*) *ssaVal set* **where**
  defs' g n ≡ (*λv.* (*v*,*n*,*SimpleDef*)) ‘ *defs g n*
definition *uses'* :: *'g* ⇒ *'node* ⇒ (*'node*, *'var*) *ssaVal set* **where**
  uses' g n ≡ *lookupDef g n* ‘ *uses g n*
definition *phis'* :: *'g* ⇒ (*'node*, (*'node*, *'var*) *ssaVal*) *phis* **where**
  phis' ≡ *λg* (*n*,(*v*,*m*,*def*)).
    if m = *n* ∧ *n* ∈ *phiDefNodes g v* ∧ *v* ∈ *vars g* ∧ *def* = *PhiDef then*
      Some [*lookupDef g m v* . *m* ← *predecessors g n*]
    else None
declare *uses'-def* [*code*] *defs'-def* [*code*] *phis'-def* [*code*]

abbreviation *lookupDefNode g n v* ≡ *defNode* (*lookupDef g n v*)
declare *lookupDef.simps* [*simp del*]
declare *phiDefNodes-aux.simps* [*simp del*]

lemma *phiDefNodes-aux-cases*:
  obtains (*nonrec*) *phiDefNodes-aux g v unvisited n* = {} (*n* ∉ *set unvisited* ∨ *v* ∈ *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 ∈ *set unvisited v* ∉ *defs g n*
⟨*proof*⟩

lemma *phiDefNode-aux-is-join-node*:
  assumes *n* ∈ *phiDefNodes-aux g v un m*
  shows *length* (*predecessors g n*) ≠ *1*
⟨*proof*⟩

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

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

lemma *unvisitedPath-removeLast*:
assumes $\mathit{unvisitedPath}\ \mathit{un}\ \mathit{ns}\ \mathit{length}\ \mathit{ns} \geq \mathit{2}$
shows $\mathit{unvisitedPath}\ (\mathit{removeAll}\ (\mathit{last}\ \mathit{ns})\ \mathit{un})\ (\mathit{butlast}\ \mathit{ns})$
⟨*proof*⟩

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

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

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

lemma *phiDefNodes-αn*[*simp*]: $n \in \mathit{phiDefNodes}\ g\ v \Longrightarrow n \in \mathit{set}\ (\alpha n\ g)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *lookupDef-induct''*[*consumes 2*, *case-names SimpleDef PhiDef rec*]:
assumes *lookupDef g n v* = *val* $n \in$ *set* (αn *g*)
⟦$v \in$ *defs g* (*defNode val*); *defKind val* = *SimpleDef*⟧ $\Longrightarrow$ *P* (*defNode val*)
⟦$v \notin$ *defs g* (*defNode val*); *length* (*predecessors g* (*defNode val*)) $\neq$ *1*; *defKind val* = *PhiDef*⟧ $\Longrightarrow$ *P* (*defNode val*)
$\bigwedge$*n m*. ⟦$v \notin$ *defs g n*; *predecessors g n* = [*m*]; $m \in$ *set* (αn *g*); *lookupDef g n v* = *lookupDef g m v*; *P m*⟧ $\Longrightarrow$ *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* (αn *g*) $\Longrightarrow$ *defs' g n* $\cap$ *uses' g n* = {}
⟨*proof*⟩

lemma *allDefs'-disjoint*: $n \in$ *set* (αn *g*) $\Longrightarrow$ $m \in$ *set* (αn *g*) $\Longrightarrow$ $n \neq m$
$\Longrightarrow$ (*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* $\Longrightarrow$ *length* (*predecessors g n*) = *length args*
⟨*proof*⟩

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

lemma *oldDefs-correct*: $defs\ g\ n = var\ g\ `\ defs'\ g\ n$
⟨*proof*⟩

lemma *oldUses-correct*: $n \in set\ (\alpha n\ g) \Longrightarrow uses\ g\ n = var\ g\ `\ uses'\ g\ n$
⟨*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 $m \in set\ (\alpha n\ g)$
obtains *ns* **where** $g \vdash lookupDefNode\ g\ m\ v{-}ns{\rightarrow}m\ (\forall x \in set\ (tl\ ns).\ v \notin defs\ g\ x)$
⟨*proof*⟩

lemma *lookupDef-path-conventional*:
assumes $g \vdash n{-}ns{\rightarrow}m\ \ n = lookupDefNode\ g\ m\ v\ \ n \notin set\ (tl\ ns)\ \ x \in set\ (tl\ ns)\ \ v' \in braun\text{-}ssa.allDefs\ g\ x$
shows $var\ g\ v' \neq v$
⟨*proof*⟩

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

lemma *phis′-fst*:
assumes $phis'\ g\ (n,v) = Some\ vs\ \ v' \in set\ vs$
shows $var\ g\ v' = var\ g\ v$

⟨*proof*⟩

lemma *allUse-simpleUse*:
assumes $v \in$ *braun-ssa.allUses* $g\ m\ m \in$ *set* $(\alpha n\ g)$
obtains *ms* m' **where** $g \vdash m{-}ms{\rightarrow}m'$ *var* $g\ v \in$ *uses* $g\ m'\ \forall x \in$ *set* (*tl ms*). *var* $g\ v \notin$ *defs* $g\ x$
⟨*proof*⟩

lemma *defs'*: $v \in$ *defs'* $g\ n \longleftrightarrow$ *var* $g\ v \in$ *defs* $g\ n \wedge$ *defKind* $v =$ *SimpleDef* $\wedge$ *defNode* $v = n$
⟨*proof*⟩

lemma *use-implies-allDef*:
assumes *lookupDef* $g\ m$ (*var* $g\ v$) $= v\ \ m \in$ *set* $(\alpha n\ g)$ *var* $g\ v \in$ *uses* $g\ m'\ g \vdash m{-}ms{\rightarrow}m'\ \forall x \in$ *set* (*tl ms*). *var* $g\ v \notin$ *defs* $g\ x$
shows $v \in$ *braun-ssa.allDefs* g (*defNode* v)
⟨*proof*⟩

lemma *allUse-defNode-in-*αn[*simp*]:
assumes $v \in$ *braun-ssa.allUses* $g\ m\ m \in$ *set* $(\alpha n\ g)$
shows *defNode* $v \in$ *set* $(\alpha n\ g)$
⟨*proof*⟩

lemma *allUse-implies-allDef*:
assumes $v \in$ *braun-ssa.allUses* $g\ m\ m \in$ *set* $(\alpha n\ g)$
shows $v \in$ *braun-ssa.allDefs* g (*defNode* v)
⟨*proof*⟩

lemma *conventional*:
assumes $g \vdash n{-}ns{\rightarrow}m\ n \notin$ *set* (*tl ns*) $v \in$ *braun-ssa.allDefs* $g\ n\ v \in$ *braun-ssa.allUses* $g\ m$
$x \in$ *set* (*tl ns*) $v' \in$ *braun-ssa.allDefs* $g\ x$
shows *var* $g\ v' \neq$ *var* $g\ v$
⟨*proof*⟩

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

lemma *allDefs-var-disjoint*: $[\![n \in$ *set* $(\alpha n\ g)$; $v \in$ *braun-ssa.allDefs* $g\ n$; $v' \in$ *braun-ssa.allDefs* $g\ n$; $v \neq v']\!] \Longrightarrow$ *var* $g\ v' \neq$ *var* $g\ v$
⟨*proof*⟩

lemma[*simp*]: $n \in$ *set* $(\alpha n\ g) \Longrightarrow v \in$ *defs* $g\ n \Longrightarrow$ *lookupDefNode* $g\ n\ v = n$
⟨*proof*⟩

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

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

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

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

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

lemma *braun-ssa-CFG-SSA-wf*:
CFG-SSA-wf αe αn *invar inEdges′ Entry defs′ uses′ phis′*
⟨*proof*⟩

sublocale *braun-ssa*: *CFG-SSA-wf* αe αn *invar inEdges′ Entry defs′ uses′ phis′*
⟨*proof*⟩

lemma *braun-ssa-CFG-SSA-Transformed*:
CFG-SSA-Transformed αe αn *invar inEdges′ Entry defs uses defs′ uses′ phis′ var*
⟨*proof*⟩

sublocale *braun-ssa*: *CFG-SSA-Transformed* αe αn *invar inEdges′ Entry defs uses defs′ uses′ phis′ var*
⟨*proof*⟩

lemma *PhiDef-defNode-eq*:

assumes $n \in$ *set* (αn g) $n \in$ *phiDefNodes* g v $v \in$ *vars* g
shows *braun-ssa.defNode* g (v,n,*PhiDef*) $= n$
⟨*proof*⟩

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

lemma *phiDefNodes-aux-pruned*:
assumes $m \in$ *phiDefNodes-aux* g v (αn g) n $n \in$ *set* (αn g) $v \in$ *uses* g n
shows *braun-ssa.liveVal* g (v, m, *PhiDef*)
⟨*proof*⟩

theorem *phis′-pruned*: *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* αe αn *invar inEdges′ Entry oldDefs oldUses defs uses phis var*
for
αe :: $'g \Rightarrow$ ($'node$::*linorder* $\times$ $'edgeD$ $\times$ $'node$) *set* **and**
αn :: $'g \Rightarrow$ $'node$ *list* **and**
invar :: $'g \Rightarrow$ *bool* **and**
inEdges′ :: $'g \Rightarrow$ $'node \Rightarrow$ ($'node$ $\times$ $'edgeD$) *list* **and**
Entry::$'g \Rightarrow$ $'node$ **and**

oldDefs :: *'g* ⇒ *'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 inEdges' 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 oldDefs 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* $g \Longrightarrow$ *chooseNext* $g \in$ *allVars* $g \wedge$ *trivial* g (*chooseNext* g)
⟨*proof*⟩

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

lemma *isTrivialPhi-det*: *trivial* g $v \Longrightarrow \exists! v'.$ *isTrivialPhi* g v v'
⟨*proof*⟩

lemma *trivialPhi-strict-dom*:
assumes[*simp*]: $v \in$ *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* $g \Longrightarrow$ *isTrivialPhi* g (*chooseNext* g) (*substitution* g)
⟨*proof*⟩

lemma *trivialPhi-in-allVars*[*simp*]:
assumes *isTrivialPhi* g v v' **and**[*simp*]: $v \in$ *allVars* g
shows $v' \in$ *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*]: $n \in$ *set* (αn g) *redundant* g
shows *defs* g $n \cap$ *uses'* g $n = \{\}$
⟨*proof*⟩
end

context *CFG-SSA-wf*
begin
inductive *liveVal'* :: *'g* $\Rightarrow$ *'val list* $\Rightarrow$ *bool*
for g :: *'g*
where
liveSimple': $[\![n \in$ *set* $(\alpha n\ g)$; *val* $\in$ *uses* g $n]\!] \Longrightarrow$ *liveVal'* g [*val*]
| *livePhi'*: $[\![$*liveVal'* g $(v\#vs)$; *phiArg* g v $v']\!] \Longrightarrow$ *liveVal'* g $(v'\#v\#vs)$

lemma *liveVal'-suffix*:

assumes *liveVal'* g *vs suffix vs' vs vs'* $\neq$ []
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 $n \in$ *set* (αn g)
shows *defs* g n ∩ *step.phiDefs* g n = {}
⟨*proof*⟩

lemma *allDefs-disjoint-inv*:
assumes $n \in$ *set* (αn g) $m \in$ *set* (αn g) $n \neq m$
shows *step.allDefs* g n ∩ *step.allDefs* g m = {}
⟨*proof*⟩

lemma *phis-finite-inv*:
  shows *finite* (*dom* (*phis′ g*))
⟨*proof*⟩

lemma *phis-wf-inv*:
  assumes *phis′ g* (*n*, *v*) = *Some args*
  shows *length* (*old.predecessors g n*) = *length args*
⟨*proof*⟩

sublocale *step*: *CFG-SSA αe αn invar inEdges′ Entry defs u-g p-g*
⟨*proof*⟩

lemma *allUses-narrows*:
  assumes *n* ∈ *set* (*αn g*)
  shows *step.allUses g n* ⊆ *substNext g* ‘ *allUses g n*
⟨*proof*⟩

lemma *allDefs-narrows*[*simp*]: *v* ∈ *step.allDefs g n* ⟹ *v* ∈ *allDefs g n*
  ⟨*proof*⟩

lemma *allUses-def-ass-inv*:
  assumes *v′* ∈ *step.allUses g n n* ∈ *set* (*αn g*)
  shows *step.defAss g n v′*
⟨*proof*⟩

lemma *Entry-no-phis-inv*: *phis′ g* (*Entry g*, *v*) = *None*
⟨*proof*⟩

sublocale *step*: *CFG-SSA-wf αe αn invar inEdges′ Entry defs u-g p-g*
⟨*proof*⟩

lemma *chooseNext-eliminated*: *chooseNext g* ∉ *step.allDefs g* (*defNode g* (*chooseNext g*))
⟨*proof*⟩

lemma *oldUses-inv*:
  assumes *n* ∈ *set* (*αn g*)
  shows *oldUses g n* = *var g* ‘ *u-g g n*
⟨*proof*⟩

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*
⟨*proof*⟩

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

$\langle proof \rangle$

lemma *phis-same-var-inv*:
assumes $phis'\ g\ (n,v) = Some\ vs\ v' \in set\ vs$
shows $var\ g\ v' = var\ g\ v$
$\langle proof \rangle$

lemma *allDefs-var-disjoint-inv*: $[\![n \in set\ (\alpha n\ g);\ v \in step.allDefs\ g\ n;\ v' \in step.allDefs\ g\ n;\ v \neq v']\!] \Longrightarrow var\ g\ v' \neq var\ g\ v$
$\langle proof \rangle$

lemma *step-CFG-SSA-Transformed-notriv*: *CFG-SSA-Transformed-notriv* αe αn *invar inEdges′ Entry oldDefs oldUses defs u-g p-g var chooseNext-all*
$\langle proof \rangle$

sublocale *step*: *CFG-SSA-Transformed-notriv* αe αn *invar inEdges′ Entry oldDefs oldUses defs u-g p-g var chooseNext-all*
$\langle proof \rangle$

lemma *step-defNode*: $v \in allVars\ g \Longrightarrow v \neq chooseNext\ g \Longrightarrow step.defNode\ g\ v = defNode\ g\ v$
$\langle proof \rangle$

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

lemma *liveVal′-inv*:
assumes $liveVal'\ g\ (v\#vs)\ v \neq chooseNext\ g$
obtains vs' **where** $step.liveVal'\ g\ (v\#vs')$
$\langle proof \rangle$

lemma *liveVal-inv*:
assumes $liveVal\ g\ v\ v \neq chooseNext\ g$
shows $step.liveVal\ g\ v$
$\langle proof \rangle$

lemma *pruned-inv*:
assumes $pruned\ g$
shows $step.pruned\ g$
$\langle proof \rangle$
end

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

interpretation *uninst*: *CFG-SSA-Transformed-notriv-base* αe αn *invar inEdges'* *Entry oldDefs oldUses defs u p var chooseNext-all*
for *u* **and** *p*
⟨*proof*⟩

definition *cond g* ≡ λ(*u*,*p*). *uninst.redundant* (*uses*(*g*:=*u*)) (*phis*(*g*:=*p*)) *g*
definition *step g* ≡ λ(*u*,*p*). (*uninst.uses'* (*uses*(*g*:=*u*)) (*phis*(*g*:=*p*)) *g*,
uninst.phis' (*uses*(*g*:=*u*)) (*phis*(*g*:=*p*)) *g*)
definition[*code*]: *substAll g* ≡ *while* (*cond g*) (*step g*) (*uses g*,*phis g*)

definition[*code*]: *uses'-all g* ≡ *fst* (*substAll g*)
definition[*code*]: *phis'-all g* ≡ *snd* (*substAll g*)

lemma *uninst-allVars-simps* [*simp*]:
uninst.allVars u (λ-. *p g*) *g* = *uninst.allVars u p g*
uninst.allVars (λ-. *u g*) *p g* = *uninst.allVars u p g*
uninst.allVars (*uses*(*g*:=*u g*)) *p g* = *uninst.allVars u p g*
uninst.allVars u (*phis*(*g*:=*p g*)) *g* = *uninst.allVars u p g*
⟨*proof*⟩

lemma *uninst-trivial-simps* [*simp*]:
uninst.trivial u (λ-. *p g*) *g* = *uninst.trivial u p g*
uninst.trivial (λ-. *u g*) *p g* = *uninst.trivial u p g*
uninst.trivial (*uses*(*g*:=*u g*)) *p g* = *uninst.trivial u p g*
uninst.trivial u (*phis*(*g*:=*p g*)) *g* = *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 *card* (*dom* (*phis' g*)) < *card* (*dom* (*phis g*))
⟨*proof*⟩

lemma *step-preserves-inst*:
assumes *inst' g* (*u*,*p*)
and *CFG-SSA-wf-base.redundant* αn *inEdges' defs* (*uses*(*g*:=*u*)) (*phis*(*g*:=*p*)) *g*
shows *inst' g* (*step g* (*u*,*p*))
⟨*proof*⟩

lemma *substAll*:
assumes *P* (*uses g*, *phis g*)
assumes $\bigwedge x.\ P\ x \Longrightarrow inst'\ g\ x \Longrightarrow cond\ g\ x \Longrightarrow P\ (step\ g\ x)$

assumes $\bigwedge x.\ P\ x \Longrightarrow inst'\ g\ x \Longrightarrow \neg cond\ g\ x \Longrightarrow Q\ (fst\ x)\ (snd\ x)$
shows $inst\ g\ (uses'\text{-}all\ g)\ (phis'\text{-}all\ g)\ Q\ (uses'\text{-}all\ g)\ (phis'\text{-}all\ g)$
⟨*proof*⟩

sublocale *notriv*: *CFG-SSA-Transformed* αe αn *invar inEdges′ Entry oldDefs oldUses defs uses′-all phis′-all*
⟨*proof*⟩

theorem *not-redundant*: $\neg$*notriv.redundant g*
⟨*proof*⟩

corollary *minimal*: *old.reducible g* $\Longrightarrow$ *notriv.cytronMinimal g*
⟨*proof*⟩

theorem *pruned-invariant*:
assumes *pruned g*
shows *notriv.pruned g*
⟨*proof*⟩
end

end

5 Proof of Semantic Equivalence

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

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

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* :: $'g \Rightarrow 'node\ list \Rightarrow ('node, 'var)\ state \Rightarrow bool$ (- ⊢ -⇓- [*50*, *50*, *50*] *50*)
where
$g \vdash Entry\ g{-}ns{\rightarrow}last\ ns \Longrightarrow g \vdash ns{\Downarrow}(fold\ (step\ g)\ ns\ Map.empty)$

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 $\Rightarrow$ *s (phiParams ! i)*
| *None* $\Rightarrow$ *s v*

inductive *ssaBS* :: $'g \Rightarrow$ *'node list* $\Rightarrow$ *('node, 'val) state* $\Rightarrow$ *bool* (- $\vdash$ -$\Downarrow_s$ - [50, 50, 50] 50)
for
g :: *'g*
where
empty: $g \vdash$ [*Entry g*]$\Downarrow_s$(*ssaStep g* (*Entry g*) *0 Map.empty*)
| *snoc*: $[\![g \vdash ns\Downarrow_s s$; *last ns* = *old.predecessors g m* ! *i*; $m \in$ *set* (αn *g*); $i <$ *length* (*old.predecessors g m*)$]\!] \Longrightarrow$
$g \vdash$ (*ns*@[*m*])$\Downarrow_s$(*ssaStep g m i s*)

lemma *ssaBS-I*:
assumes $g \vdash$ *Entry g*$-ns\rightarrow n$
obtains *s* **where** $g \vdash ns\Downarrow_s s$
⟨*proof*⟩

lemma *ssaBS-nonempty*[*simp*]: $\neg$ ($g \vdash$ []$\Downarrow_s s$)
⟨*proof*⟩

lemma *ssaBS-hd*[*simp*]: $g \vdash ns\Downarrow_s s \Longrightarrow$ *hd ns* = *Entry g*
⟨*proof*⟩

lemma *equiv-aux*:
assumes $g \vdash ns\Downarrow s$ $g \vdash ns\Downarrow_s s'$ $g \vdash$ *last ns*$-ms\rightarrow m$ $v \in$ *allUses g m* $\forall n \in$ *set* (*tl ms*). *var g v* $\notin$ *var g* ‘ *allDefs g n*
shows *s* (*var g v*) = *s' v*
⟨*proof*⟩

theorem *equiv*:
assumes $g \vdash ns\Downarrow s$ $g \vdash ns\Downarrow_s s'$ $v \in$ *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** $\bigwedge s.\ [\![I\ s;\ b\ s]\!] \Longrightarrow I\ (c\ s)$
  and $\bigwedge s.\ [\![I\ s;\ b\ s]\!] \Longrightarrow (c\ s,\ s\ ) \in r$
  shows *False*
  ⟨*proof*⟩

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

lemma *while-option-sim*:
  assumes *start*: $R\ (Some\ s1)\ (Some\ s2)$
  and *cond*: $\bigwedge s1\ s2.\ [\![R\ (Some\ s1)\ (Some\ s2);\ I\ s1\ ]\!] \Longrightarrow b1\ s1 = b2\ s2$
  and *step* : $\bigwedge s1\ s2.\ [\![R\ (Some\ s1)\ (Some\ s2);\ I\ s1;\ b1\ s1\ ]\!] \Longrightarrow R\ (Some\ (c1\ s1))\ (Some\ (c2\ s2))$
  and *diverge*: *R None None*
  and *inv-start*: $I\ s1$
  and *inv-step*: $\bigwedge s1.\ [\![I\ s1;\ b1\ s1\ ]\!] \Longrightarrow I\ (c1\ s1)$
  shows R (*while-option* $b1\ c1\ s1$) (*while-option* $b2\ c2\ s2$)
⟨*proof*⟩

end

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

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

lemma *Union-of-alt-def*: *Union-of* $f\ A = (\bigcup x \in A.\ f\ x)$
  ⟨*proof*⟩

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

context *CFG-base* **begin**
  definition *addN* :: $'g \Rightarrow 'node \Rightarrow ('var, 'node\ set)$ *mapping* $\Rightarrow ('var, 'node\ set)$ *mapping*
   where *addN* $g\ n \equiv$ *fold* ($\lambda v.$ *Mapping.map-default* $v\ \{\}$ (*insert* n)) (*sorted-list-of-set* (*uses* $g\ n$))
  definition *addN'* $g\ n =$ *fold* ($\lambda v\ m.\ m(v \mapsto$ *case-option* $\{n\}$ (*insert* n) $(m\ v)$)) (*sorted-list-of-set* (*uses* $g\ n$))

lemma *addN-transfer* [*transfer-rule*]:
rel-fun (=) (*rel-fun* (=) (*rel-fun* (*pcr-mapping* (=) (=)) (*pcr-mapping* (=) (=)))) *addN′ addN*
⟨*proof*⟩

definition *useNodes-of g* = *fold* (*addN g*) (*αn 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*]: ⋀*n*. *finite* (*uses g n*)
shows *Mapping.lookup* (*useNodes-of g*) *v* =
(*if* (∃ *n* ∈ *set* (*αn g*). *v* ∈ *uses g n*) *then Some* {*n* ∈ *set* (*αn g*). *v* ∈ *uses g n*} *else None*)
⟨*proof*⟩
end

context *CFG* **begin**
lift-definition *useNodes-of′* :: *′g* ⇒ (*′var*, *′node set*) *mapping*
is *λg v*. *if* (∃ *n* ∈ *set* (*αn g*). *v* ∈ *uses g n*) *then Some* {*n* ∈ *set* (*αn g*). *v* ∈ *uses g n*} *else None* ⟨*proof*⟩

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

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

lemma *lookup-useNodes-of*: *Mapping.lookup* (*useNodes-of g*) *v* =
(*if* (∃ *n* ∈ *set* (*αn g*). *v* ∈ *uses g n*) *then Some* {*n* ∈ *set* (*αn g*). *v* ∈ *uses g n*} *else None*)
⟨*proof*⟩

end

context *CFG-SSA-base* **begin**
definition *phis-addN*
where *phis-addN g n* = *fold* (*λv*. *Mapping.map-default v* {} (*insert n*)) (*case-option* [] *id* (*phis g n*))

definition *phidefNodes* **where** [*code*]:
phidefNodes g = *fold* (*λ*(*n*,*v*). *Mapping.update v n*) (*sorted-list-of-set* (*dom* (*phis g*))) *Mapping.empty*

lemma *keys-phidefNodes*:
assumes *finite* (*dom* (*phis g*))
shows *Mapping.keys* (*phidefNodes g*) = *snd* ‘ *dom* (*phis g*)
⟨*proof*⟩

definition *phiNodes-of* :: *′g* ⇒ (*′val*, (*′node* × *′val*) *set*) *mapping*

where *phiNodes-of g* = *fold* (*phis-addN g*) (*sorted-list-of-set* (*dom* (*phis g*))) *Mapping.empty*

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

lemmas *phiNodes-of-code* = *phiNodes-of-def* [*unfolded phis-addN-def* [*abs-def*]]
declare *phiNodes-of-code* [*code*]

lemma *phis-transfer* [*transfer-rule*]:
includes *lifting-syntax*
shows ((=) ===> *pcr-mapping* (=) (=)) *phis* (λ*g*. *Mapping.Mapping* (*phis g*))
⟨*proof*⟩

end

context *CFG-SSA* **begin**
declare *lookup-phiNodes-of* [*OF phis-finite*, *simp*]
declare *keys-phidefNodes* [*OF phis-finite*, *simp*]
end

locale *CFG-SSA-ext-base* = *CFG-SSA-base* αe αn *invar inEdges' Entry defs uses phis*
for αe :: *'g* $\Rightarrow$ (*'node*::*linorder* $\times$ *'edgeD* $\times$ *'node*) *set*
and αn :: *'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
abbreviation *cache g f* $\equiv$ *Mapping.tabulate* (αn *g*) *f*

lemma *lookup-cache*[*simp*]: *n* $\in$ *set* (αn *g*) $\Longrightarrow$ *Mapping.lookup* (*cache g f*) *n* = *Some* (*f n*)
⟨*proof*⟩

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

lemma *lookup-cache-usesD*: *Mapping.lookup* (*cache g* (*uses g*)) *n* = *Some vs* $\Longrightarrow$ *vs* = *uses g n*
⟨*proof*⟩

end

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

locale *CFG-SSA-ext* = *CFG-SSA-ext-base* αe αn *invar inEdges' Entry defs uses phis*
+ *CFG-SSA* αe αn *invar inEdges' Entry defs uses phis*
for αe :: *'g* $\Rightarrow$ (*'node*::*linorder* $\times$ *'edgeD* $\times$ *'node*) *set*
and αn :: *'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*
⟨*proof*⟩
end

locale *CFG-SSA-base-code* = *CFG-SSA-ext-base* αe αn *invar inEdges' Entry defs usesOf* $\circ$ *uses* λg. *Mapping.lookup* (*phis g*)
for αe :: *'g* $\Rightarrow$ (*'node*::*linorder* $\times$ *'edgeD* $\times$ *'node*) *set*
and αn :: *'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* (λ(*n'*,*v*). *n'* = *n*) (*Mapping.keys* (*phis g*))
⟨*proof*⟩

lemmas *phiUses-code* [*code*] = *phiUses-def* [*folded Union-of-alt-def*]
declare *allUses-def* [*code*]
lemmas *allVars-code* [*code*] = *allVars-def* [*folded Union-of-alt-def*]
end

locale *CFG-SSA-code* = *CFG-SSA-base-code* αe αn *invar inEdges' Entry defs uses phis*
+ *CFG-SSA-ext* αe αn *invar inEdges' Entry defs usesOf* $\circ$ *uses* λg. *Mapping.lookup* (*phis g*)
for αe :: *'g* $\Rightarrow$ (*'node*::*linorder* $\times$ *'edgeD* $\times$ *'node*) *set*
and αn :: *'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* α*e* α*n invar inEdges' Entry defs uses phis*
+ *CFG-SSA-wf-base* α*e* α*n invar inEdges' Entry defs usesOf* ∘ *uses* λ*g*. *Mapping.lookup* (*phis g*)
for α*e* :: *'g* ⇒ (*'node::linorder* × *'edgeD* × *'node*) *set*
and α*n* :: *'g* ⇒ *'node list*
and *invar* :: *'g* ⇒ *bool*
and *inEdges'* :: *'g* ⇒ *'node* ⇒ (*'node* × *'edgeD*) *list*
and *Entry* :: *'g* ⇒ *'node*
and *defs* :: *'g* ⇒ *'node* ⇒ *'val::linorder set*
and *uses* :: *'g* ⇒ (*'node*, *'val set*) *mapping*
and *phis* :: *'g* ⇒ (*'node*, *'val*) *phis-code*
begin
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* = (*trivial-phis g* ≠ {})
end

locale *CFG-SSA-wf-code* = *CFG-SSA-code αe αn invar inEdges′ Entry defs uses phis*
+ *CFG-SSA-wf-base-code αe αn invar inEdges′ Entry defs uses phis*
+ *CFG-SSA-wf 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
lemma *trivial-code*:
phi g v = *Some vs* ⟹ *trivial g v* = *trivial-code v vs*
⟨*proof*⟩

lemma *trivial-phis*:
trivial-phis g = {(*n*,*v*). *Mapping.lookup* (*phis g*) (*n*,*v*) ≠ *None* ∧ *trivial g v*}
⟨*proof*⟩

lemma *redundant-code*:
redundant g = *redundant-code g*
⟨*proof*⟩

lemma *trivial-code-mapI*:
⟦ *trivial-code v vs*; *f* ‘ (*set vs* − {*v*}) ≠ {*v*} ; *f v* = *v* ⟧ ⟹ *trivial-code v* (*map f vs*)
⟨*proof*⟩

lemma *trivial-code-map-conv*:
f v = *v* ⟹ *trivial-code v* (*map f vs*) ⟷ (∃ *v′*∈*set vs*. *f v′* ≠ *v* ∧ (*f* ‘ *set vs*) − {*f v′*} ⊆ {*v*})
⟨*proof*⟩

end

locale *CFG-SSA-Transformed-code* = *ssa*: *CFG-SSA-wf-code αe αn invar inEdges′ Entry defs uses phis*
+
CFG-SSA-Transformed αe αn invar inEdges′ Entry oldDefs oldUses defs usesOf ∘ *uses λg. Mapping.lookup* (*phis g*) *var*
for
αe :: ′*g* ⇒ (′*node*::*linorder* × ′*edgeD* × ′*node*) *set* **and**
αn :: ′*g* ⇒ ′*node list* **and**

invar :: *'g* ⇒ *bool* **and**
inEdges' :: *'g* ⇒ *'node* ⇒ (*'node* × *'edgeD*) *list* **and**
Entry::*'g* ⇒ *'node* **and**
oldDefs :: *'g* ⇒ *'node* ⇒ *'var*::*linorder set* **and**
oldUses :: *'g* ⇒ *'node* ⇒ *'var set* **and**
defs :: *'g* ⇒ *'node* ⇒ *'val*::*linorder set* **and**
uses :: *'g* ⇒ (*'node*, *'val set*) *mapping* **and**
phis :: *'g* ⇒ (*'node*, *'val*) *phis-code* **and**
var :: *'g* ⇒ *'val* ⇒ *'var*
+
assumes *dom-uses-in-graph*: *Mapping.keys* (*uses g*) ⊆ *set* (*αn g*)

end

6.2 Code Equations for SSA Construction

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

definition[*code*]: *lookup-multimap m k* ≡ (*case-option* {} *id* (*Mapping.lookup m k*))

locale *CFG-Construct-linorder* = *CFG-Construct-wf αe αn invar inEdges' Entry defs uses*
for
αe :: *'g* ⇒ (*'node*::*linorder* × *'edgeD* × *'node*) *set* **and**
αn :: *'g* ⇒ *'node list* **and**
invar :: *'g* ⇒ *bool* **and**
inEdges' :: *'g* ⇒ *'node* ⇒ (*'node* × *'edgeD*) *list* **and**
Entry::*'g* ⇒ *'node* **and**
defs :: *'g* ⇒ *'node* ⇒ (*'var*::*linorder*) *set* **and**
uses :: *'g* ⇒ *'node* ⇒ *'var set*
begin
type-synonym (*'n*, *'v*) *sparse-phis* = (*'n* × *'v*, (*'n*, *'v*) *ssaVal list*) *mapping*

function *readVariableRecursive* :: *'g* ⇒ *'var* ⇒ *'node* ⇒ (*'node*, *'var*) *sparse-phis* ⇒ ((*'node*, *'var*) *ssaVal* × (*'node*, *'var*) *sparse-phis*)
and *readArgs* :: *'g* ⇒ *'var* ⇒ *'node* ⇒ (*'node*, *'var*) *sparse-phis* ⇒ *'node list* ⇒ (*'node*, *'var*) *sparse-phis* × (*'node*, *'var*) *ssaVal list*
where[*code*]: *readVariableRecursive g v n phis* = (*if v* ∈ *defs g n then* ((*v*,*n*,*SimpleDef*), *phis*)
else case predecessors g n of
[] ⇒ ((*v*,*n*,*PhiDef*), *Mapping.update* (*n*,*v*) [] *phis*)
| [*m*] ⇒ *readVariableRecursive g v m phis*
| *ms* ⇒ (*case Mapping.lookup phis* (*n*,*v*) *of*

Some - ⇒ $((v,n,PhiDef),phis)$
| *None* ⇒
let $phis = Mapping.update\ (n,v)\ []\ phis$ *in*
let $(phis,args) = readArgs\ g\ v\ n\ phis\ ms$ *in*
$((v,n,PhiDef),\ Mapping.update\ (n,v)\ args\ phis)$
))
| $readArgs\ g\ v\ n\ phis\ [] = (phis,[])$
| $readArgs\ g\ v\ n\ phis\ (m\#ms) = ($
let $(phis,args) = readArgs\ g\ v\ n\ phis\ ms$ *in*
let $(v,phis) = readVariableRecursive\ g\ v\ m\ phis$ *in*
$(phis,v\#args))$
⟨*proof*⟩

lemma *length-filter-less2*:
assumes $x \in set\ xs\ \neg P\ x\ Q\ x\ \bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $length\ (filter\ P\ xs) < length\ (filter\ Q\ xs)$
⟨*proof*⟩

lemma *length-filter-le2*:
assumes $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $length\ (filter\ P\ xs) \leq length\ (filter\ Q\ xs)$
⟨*proof*⟩

abbreviation $phis\text{-}measure\ g\ v\ phis \equiv length\ [n \leftarrow \alpha n\ g.\ Mapping.lookup\ phis\ (n,v) = None]$

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

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

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

termination
⟨*proof*⟩

declare *readVariableRecursive.simps*[*simp del*] *readArgs.simps*[*simp del*]

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

shows *fst* (*readVariableRecursive g v n phis*) = *lookupDef g n v*
⟨*proof*⟩

definition *phis′-aux g v ns* (*phis*:: (*′node*,*′var*) *sparse-phis*) ≡ *Mapping.Mapping* (λ(*m*,v_2).
(*if* v_2=*v* ∧ *m* ∈ ⋃(*phiDefNodes-aux g v* [*n* ← *αn g*. (*n*,*v*) ∉ *Mapping.keys phis*] ‘ *ns*) ∧ *v* ∈ *vars g then Some* (*map* (*λm. lookupDef g m v*) (*predecessors g m*)) *else* (*Mapping.lookup phis* (*m*,v_2))))

lemma *phis′-aux-keys-super*: *Mapping.keys* (*phis′-aux g v ns phis*) ⊇ *Mapping.keys phis*
⟨*proof*⟩

lemma *phiDefNodes-aux-in-unvisited*:
shows *phiDefNodes-aux g v un n* ⊆ *set un*
⟨*proof*⟩

lemma *phiDefNodes-aux-unvisited-monotonic*:
assumes *set un* ⊆ *set un′*
shows *phiDefNodes-aux g v un n* ⊆ *phiDefNodes-aux g v un′ n*
⟨*proof*⟩

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

lemma *phis′-aux-finite*:
assumes *finite* (*Mapping.keys phis*)
shows *finite* (*Mapping.keys* (*phis′-aux g v ns phis*))
⟨*proof*⟩

lemma *phiDefNodes-aux-redirect*:
assumes *asm*: *g* ⊢ *n*−*ns*→*m* ∀ *n* ∈ *set ns*. *v* ∉ *defs g n length* (*predecessors g n*) ≠ *1 unvisitedPath un ns*
assumes *n′*: *n′* ∈ *set ns n′* ∈ *phiDefNodes-aux g v un m′ m′* ∈ *set* (*αn g*)
shows *n* ∈ *phiDefNodes-aux g v un m′*
⟨*proof*⟩

lemma *snd-readVariableRecursive*:
assumes *v* ∈ *vars g n* ∈ *set* (*αn g*) *finite* (*Mapping.keys phis*)
⋀*n*. (*n*,*v*) ∈ *Mapping.keys phis* ⟹ *length* (*predecessors g n*) ≠ *1 Mapping.lookup phis* (*Entry g*,*v*) ∈ {*None*, *Some* []}
shows
phis′-aux g v {*n*} *phis* = *snd* (*readVariableRecursive g v n phis*)
set ms ⊆ *set* (*αn g*) ⟹ (*phis′-aux g v* (*set ms*) *phis*, *map* (*λm. lookupDef g m v*) *ms*) = *readArgs g v n phis ms*
⟨*proof*⟩

definition *aux-1 g n* = (λ*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* $\equiv$ λ(*n*,*v*). (*n*,(*v*,*n*,*PhiDef*))
definition *uses'-phis' g* = (
let (*u*,*p*) = *foldr* (*aux-2 g*) (α*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* (λ*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*) $\bigwedge$*n v*. (*n*,*v*) $\in$ *Mapping.keys p* $\Longrightarrow$ *length* (*predecessors g n*) $\neq$ *1* $\bigwedge$*v*. *Mapping.lookup p* (*Entry g*,*v*) $\in$ {*None*, *Some* []}
shows *lookupDef g n ` set us* = *lookup-multimap u' n* $\bigwedge$*m*. *m* $\neq$ *n* $\Longrightarrow$ *Mapping.lookup u' m* = *Mapping.lookup u m*
$\bigwedge$*m v*. (*if m* $\in$ *phiDefNodes-aux g v* [*n* $\leftarrow$ α*n g*. (*n*,*v*) $\notin$ *Mapping.keys p*] *n* $\wedge$ *v* $\in$ *set us then*
Some (*map* (λ*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* (α*n g*) *distinct ns foldr* (*aux-2 g*) *ns init-state* = (*u'*,*p'*)
shows $\bigwedge$*n*. *n* $\in$ *set ns* $\Longrightarrow$ *uses' g n* = *lookup-multimap u' n* $\bigwedge$*n*. *n* $\notin$ *set ns* $\Longrightarrow$ *Mapping.lookup u' n* = *None*
$\bigwedge$*m v*. (*if* $\exists$*n* $\in$ *set ns*. *m* $\in$ *phiDefNodes-aux g v* (α*n g*) *n* $\wedge$ *v* $\in$ *uses g n then*
Some (*map* (λ*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*)) ⊆ *set* (*αn 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* ⟹ ((*R* ===> (=)) ===> (⟶)) *All All*
⟨*proof*⟩

lemma *weak-imp-transfer* [*transfer-rule*]:
((=) ===> (=) ===> (⟶)) (⟶) (⟶)
⟨*proof*⟩

lemma *weak-conj-transfer* [*transfer-rule*]:
((⟶) ===> (⟶) ===> (⟶)) (∧) (∧)
⟨*proof*⟩

lemma *graph-path-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: (*G* ===> (=)) *αe αe2*
and [*transfer-rule*]: (*G* ===> (=)) *αn αn2*
and [*transfer-rule*]: (*G* ===> (=)) *invar invar2*
and [*transfer-rule*]: (*G* ===> (=)) *inEdges inEdges2*
shows (⟶) (*graph-path αe αn invar inEdges*) (*graph-path αe2 αn2 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*]: $(A ===> (=))$ *αe αe2*
    and [*transfer-rule*]: $(A ===> (=))$ *αn αn2*
    and [*transfer-rule*]: $(A ===> (=))$ *invar invar2*
    and [*transfer-rule*]: $(A ===> (=))$ *inEdges′ inEdges2*
  shows $(A ===> (=))$ *inEdges* (*graph-path-base.inEdges inEdges2*)
⟨*proof*⟩

lemma *predecessors-transfer* [*transfer-rule*]:
  assumes [*transfer-rule*]: *right-total A*
    and [*transfer-rule*]: $(A ===> (=))$ *αe αe2*
    and [*transfer-rule*]: $(A ===> (=))$ *αn αn2*
    and [*transfer-rule*]: $(A ===> (=))$ *invar invar2*
    and [*transfer-rule*]: $(A ===> (=))$ *inEdges′ inEdges2*
  shows $(A ===> (=))$ *predecessors* (*graph-path-base.predecessors inEdges2*)
⟨*proof*⟩

lemma *successors-transfer* [*transfer-rule*]:
  assumes [*transfer-rule*]: *right-total A*
    and [*transfer-rule*]: $(A ===> (=))$ *αe αe2*
    and [*transfer-rule*]: $(A ===> (=))$ *αn αn2*
    and [*transfer-rule*]: $(A ===> (=))$ *invar invar2*
    and [*transfer-rule*]: $(A ===> (=))$ *inEdges′ inEdges2*
  shows $(A ===> (=))$ *successors* (*graph-path-base.successors αn2 inEdges2*)
⟨*proof*⟩

lemma *path-transfer* [*transfer-rule*]:
  assumes [*transfer-rule*]: *right-total A*
    and [*transfer-rule*]: $(A ===> (=))$ *αe αe2*
    and [*transfer-rule*]: $(A ===> (=))$ *αn αn2*
    and [*transfer-rule*]: $(A ===> (=))$ *invar invar2*
    and [*transfer-rule*]: $(A ===> (=))$ *inEdges′ inEdges2*
  shows $(A ===> (=))$ *path* (*graph-path-base.path αn2 invar2 inEdges2*)
⟨*proof*⟩

lemma *path2-transfer* [*transfer-rule*]:
  assumes [*transfer-rule*]: *right-total A*
    and [*transfer-rule*]: $(A ===> (=))$ *αe αe2*
    and [*transfer-rule*]: $(A ===> (=))$ *αn αn2*
    and [*transfer-rule*]: $(A ===> (=))$ *invar invar2*
    and [*transfer-rule*]: $(A ===> (=))$ *inEdges′ inEdges2*
  shows $(A ===> (=))$ *path2* (*graph-path-base.path2 αn2 invar2 inEdges2*)
⟨*proof*⟩

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

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

lemma *dominates-transfer* [*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: $(G ===> (=))$ αe $\alpha e2$
and [*transfer-rule*]: $(G ===> (=))$ αn $\alpha n2$
and [*transfer-rule*]: $(G ===> (=))$ *invar invar2*
and [*transfer-rule*]: $(G ===> (=))$ *inEdges' inEdges2*
and [*transfer-rule*]: $(G ===> (=))$ *Entry Entry2*
shows $(G ===> (=))$ *dominates* (*graph-Entry-base.dominates* $\alpha n2$ *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*]: $(G ===> (=))$ αe $\alpha e2$
and [*transfer-rule*]: $(G ===> (=))$ αn $\alpha n2$
and [*transfer-rule*]: $(G ===> (=))$ *invar invar2*
and [*transfer-rule*]: $(G ===> (=))$ *inEdges' inEdges2*

and [*transfer-rule*]: (G ===> (=)) *Entry Entry2*
shows (G ===> (=)) *shortestPath* (*graph-Entry.shortestPath* α*n2 invar2 inEdges2 Entry2*)
⟨*proof*⟩

lemma *dominators-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: (G ===> (=)) α*e* α*e2*
and [*transfer-rule*]: (G ===> (=)) α*n* α*n2*
and [*transfer-rule*]: (G ===> (=)) *invar invar2*
and [*transfer-rule*]: (G ===> (=)) *inEdges′ inEdges2*
and [*transfer-rule*]: (G ===> (=)) *Entry Entry2*
shows (G ===> (=)) *dominators* (*graph-Entry.dominators* α*n2 invar2 inEdges2 Entry2*)
⟨*proof*⟩

lemma *isIdom-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: (G ===> (=)) α*e* α*e2*
and [*transfer-rule*]: (G ===> (=)) α*n* α*n2*
and [*transfer-rule*]: (G ===> (=)) *invar invar2*
and [*transfer-rule*]: (G ===> (=)) *inEdges′ inEdges2*
and [*transfer-rule*]: (G ===> (=)) *Entry Entry2*
shows (G ===> (=)) *isIdom* (*graph-Entry.isIdom* α*n2 invar2 inEdges2 Entry2*)
⟨*proof*⟩

lemma *idom-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: (G ===> (=)) α*e* α*e2*
and [*transfer-rule*]: (G ===> (=)) α*n* α*n2*
and [*transfer-rule*]: (G ===> (=)) *invar invar2*
and [*transfer-rule*]: (G ===> (=)) *inEdges′ inEdges2*
and [*transfer-rule*]: (G ===> (=)) *Entry Entry2*
shows (G ===> (=)) *idom* (*graph-Entry.idom* α*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 ===> (=)) α*e1* α*e2*
and [*transfer-rule*]: (G ===> (=)) α*n1* α*n2*
and [*transfer-rule*]: (G ===> (=)) *invar1 invar2*
and [*transfer-rule*]: (G ===> (=)) *inEdges1 inEdges2*
and [*transfer-rule*]: (G ===> (=)) *Entry1 Entry2*
and [*transfer-rule*]: (G ===> (=)) *defs1 defs2*
and [*transfer-rule*]: (G ===> (=)) *uses1 uses2*
shows *SSA-CFG.CFG* α*e1* α*n1 invar1 inEdges1 Entry1 defs1 uses1*
$\longrightarrow$ *SSA-CFG.CFG* α*e2* α*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 ===> (=)) α*e* α*e2*
and [*transfer-rule*]: (G ===> (=)) α*n* α*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 ===> (=)) *vars* (*CFG-base.vars* α*n2 uses2*)
⟨*proof*⟩

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

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

and [*transfer-rule*]: $(G ===> (=))$ *uses uses2*
shows $(G ===> (=))$ *defAss′Uses* (*CFG-base.defAss′Uses* α*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*]: $(G ===> (=))$ α*e1* α*e2*
and [*transfer-rule*]: $(G ===> (=))$ α*n1* α*n2*
and [*transfer-rule*]: $(G ===> (=))$ *invar1 invar2*
and [*transfer-rule*]: $(G ===> (=))$ *inEdges1 inEdges2*
and [*transfer-rule*]: $(G ===> (=))$ *Entry1 Entry2*
and [*transfer-rule*]: $(G ===> (=))$ *defs1 defs2*
and [*transfer-rule*]: $(G ===> (=))$ *uses1 uses2*
shows *CFG-Construct* α*e1* α*n1 invar1 inEdges1 Entry1 defs1 uses1*
$\longrightarrow$ *CFG-Construct* α*e2* α*n2 invar2 inEdges2 Entry2 defs2 uses2*
⟨*proof*⟩

lemma *CFG-Construct-linorder-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: $(G ===> (=))$ α*e1* α*e2*
and [*transfer-rule*]: $(G ===> (=))$ α*n1* α*n2*
and [*transfer-rule*]: $(G ===> (=))$ *invar1 invar2*
and [*transfer-rule*]: $(G ===> (=))$ *inEdges1 inEdges2*
and [*transfer-rule*]: $(G ===> (=))$ *Entry1 Entry2*
and [*transfer-rule*]: $(G ===> (=))$ *defs1 defs2*
and [*transfer-rule*]: $(G ===> (=))$ *uses1 uses2*
shows *CFG-Construct-linorder* α*e1* α*n1 invar1 inEdges1 Entry1 defs1 uses1*
$\longrightarrow$ *CFG-Construct-linorder* α*e2* α*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* ===> (=)) *αe αe2*
    and [*transfer-rule*]: (*G* ===> (=)) *αn αn2*
    and [*transfer-rule*]: (*G* ===> (=)) *invar invar2*
    and [*transfer-rule*]: (*G* ===> (=)) *inEdges′ inEdges2*
    and [*transfer-rule*]: (*G* ===> (=)) *Entry Entry2*
    and [*transfer-rule*]: (*G* ===> (=)) *defs defs2*
    and [*transfer-rule*]: (*G* ===> (=)) *uses uses2*
  shows (*G* ===> (=)) *phiDefNodes-aux* (*CFG-Construct.phiDefNodes-aux inEdges2 defs2*)
⟨*proof*⟩

lemma *phiDefNodes-transfer* [*transfer-rule*]:
  assumes [*transfer-rule*]: *right-total G*
    and [*transfer-rule*]: (*G* ===> (=)) *αe αe2*
    and [*transfer-rule*]: (*G* ===> (=)) *αn αn2*
    and [*transfer-rule*]: (*G* ===> (=)) *invar invar2*
    and [*transfer-rule*]: (*G* ===> (=)) *inEdges′ inEdges2*
    and [*transfer-rule*]: (*G* ===> (=)) *Entry Entry2*
    and [*transfer-rule*]: (*G* ===> (=)) *defs defs2*
    and [*transfer-rule*]: (*G* ===> (=)) *uses uses2*
  shows (*G* ===> (=)) *phiDefNodes* (*CFG-Construct.phiDefNodes αn2 inEdges2 defs2 uses2*)
⟨*proof*⟩

lemma *lookupDef-transfer* [*transfer-rule*]:
  assumes [*transfer-rule*]: *right-total G*
    and [*transfer-rule*]: (*G* ===> (=)) *αe αe2*
    and [*transfer-rule*]: (*G* ===> (=)) *αn αn2*
    and [*transfer-rule*]: (*G* ===> (=)) *invar invar2*
    and [*transfer-rule*]: (*G* ===> (=)) *inEdges′ inEdges2*
    and [*transfer-rule*]: (*G* ===> (=)) *Entry Entry2*
    and [*transfer-rule*]: (*G* ===> (=)) *defs defs2*
    and [*transfer-rule*]: (*G* ===> (=)) *uses uses2*
  shows (*G* ===> (=)) *lookupDef* (*CFG-Construct.lookupDef αn2 inEdges2 defs2*)
⟨*proof*⟩

lemma *defs′-transfer* [*transfer-rule*]:
  assumes [*transfer-rule*]: *right-total G*
    and [*transfer-rule*]: (*G* ===> (=)) *αe αe2*
    and [*transfer-rule*]: (*G* ===> (=)) *αn αn2*
    and [*transfer-rule*]: (*G* ===> (=)) *invar invar2*
    and [*transfer-rule*]: (*G* ===> (=)) *inEdges′ inEdges2*
    and [*transfer-rule*]: (*G* ===> (=)) *Entry Entry2*

  and [*transfer-rule*]: (G ===> (=)) *defs defs2*
  and [*transfer-rule*]: (G ===> (=)) *uses uses2*
shows (G ===> (=)) *defs′* (*CFG-Construct.defs′ defs2*)
⟨*proof*⟩

lemma *uses′-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
  and [*transfer-rule*]: (G ===> (=)) αe $\alpha e2$
  and [*transfer-rule*]: (G ===> (=)) αn $\alpha n2$
  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 ===> (=)) αe $\alpha e2$
  and [*transfer-rule*]: (G ===> (=)) αn $\alpha n2$
  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 ===> (=)) αe $\alpha e2$
  and [*transfer-rule*]: (G ===> (=)) αn $\alpha n2$

and [*transfer-rule*]: $(G ===> (=))$ *invar invar2*
and [*transfer-rule*]: $(G ===> (=))$ *inEdges' inEdges2*
and [*transfer-rule*]: $(G ===> (=))$ *Entry Entry2*
and [*transfer-rule*]: $(G ===> (=))$ *defs defs2*
and [*transfer-rule*]: $(G ===> (=))$ *uses uses2*
and [*transfer-rule*]: $(G ===> (=))$ *phis phis2*
shows $(G ===> (=))$ *phiDefs* (*CFG-SSA-base.phiDefs phis2*)
⟨*proof*⟩

lemma *allDefs-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: $(G ===> (=))$ αe $\alpha e2$
and [*transfer-rule*]: $(G ===> (=))$ αn $\alpha n2$
and [*transfer-rule*]: $(G ===> (=))$ *invar invar2*
and [*transfer-rule*]: $(G ===> (=))$ *inEdges' inEdges2*
and [*transfer-rule*]: $(G ===> (=))$ *Entry Entry2*
and [*transfer-rule*]: $(G ===> (=))$ *defs* (*defs2*::'*a* ⇒ '*node* ⇒ '*val set*)
and [*transfer-rule*]: $(G ===> (=))$ *uses* (*uses2*::'*a* ⇒ '*node* ⇒ '*val set*)
and [*transfer-rule*]: $(G ===> (=))$ *phis phis2*
shows $(G ===> (=))$ *allDefs* (*CFG-SSA-base.allDefs defs2 phis2*)
⟨*proof*⟩

lemma *phiUses-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: $(G ===> (=))$ αe $\alpha e2$
and [*transfer-rule*]: $(G ===> (=))$ αn $\alpha n2$
and [*transfer-rule*]: $(G ===> (=))$ *invar invar2*
and [*transfer-rule*]: $(G ===> (=))$ *inEdges' inEdges2*
and [*transfer-rule*]: $(G ===> (=))$ *Entry Entry2*
and [*transfer-rule*]: $(G ===> (=))$ *defs defs2*
and [*transfer-rule*]: $(G ===> (=))$ *uses uses2*
and [*transfer-rule*]: $(G ===> (=))$ *phis phis2*
shows $(G ===> (=))$ *phiUses* (*CFG-SSA-base.phiUses* $\alpha n2$ *inEdges2 phis2*)
⟨*proof*⟩

lemma *allUses-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *right-total G*
and [*transfer-rule*]: $(G ===> (=))$ αe $\alpha e2$
and [*transfer-rule*]: $(G ===> (=))$ αn $\alpha n2$
and [*transfer-rule*]: $(G ===> (=))$ *invar invar2*
and [*transfer-rule*]: $(G ===> (=))$ *inEdges' inEdges2*
and [*transfer-rule*]: $(G ===> (=))$ *Entry Entry2*
and [*transfer-rule*]: $(G ===> (=))$ *defs defs2*
and [*transfer-rule*]: $(G ===> (=))$ *uses uses2*
and [*transfer-rule*]: $(G ===> (=))$ *phis phis2*
shows $(G ===> (=))$ *allUses* (*CFG-SSA-base.allUses* $\alpha n2$ *inEdges2 uses2 phis2*)
⟨*proof*⟩

lemma *allVars-transfer* [*transfer-rule*]:
  assumes [*transfer-rule*]: *right-total G*
    and [*transfer-rule*]: (*G* ===> (=)) *αe αe2*
    and [*transfer-rule*]: (*G* ===> (=)) *αn αn2*
    and [*transfer-rule*]: (*G* ===> (=)) *invar invar2*
    and [*transfer-rule*]: (*G* ===> (=)) *inEdges′ inEdges2*
    and [*transfer-rule*]: (*G* ===> (=)) *Entry Entry2*
    and [*transfer-rule*]: (*G* ===> (=)) *defs defs2*
    and [*transfer-rule*]: (*G* ===> (=)) *uses uses2*
    and [*transfer-rule*]: (*G* ===> (=)) *phis phis2*
  shows (*G* ===> (=)) *allVars* (*CFG-SSA-base.allVars αn2 inEdges2 defs2 uses2 phis2*)
⟨*proof*⟩

lemma *defAss-transfer* [*transfer-rule*]:
  assumes [*transfer-rule*]: *right-total G*
    and [*transfer-rule*]: (*G* ===> (=)) *αe αe2*
    and [*transfer-rule*]: (*G* ===> (=)) *αn αn2*
    and [*transfer-rule*]: (*G* ===> (=)) *invar invar2*
    and [*transfer-rule*]: (*G* ===> (=)) *inEdges′ inEdges2*
    and [*transfer-rule*]: (*G* ===> (=)) *Entry Entry2*
    and [*transfer-rule*]: (*G* ===> (=)) *defs defs2*
    and [*transfer-rule*]: (*G* ===> (=)) *uses uses2*
    and [*transfer-rule*]: (*G* ===> (=)) *phis phis2*
  shows (*G* ===> (=)) *defAss* (*CFG-SSA-base.defAss αn2 invar2 inEdges2 Entry2 defs2 phis2*)
⟨*proof*⟩

lemmas *CFG-SSA-base-transfer-rules* =
  phiDefs-transfer
  allDefs-transfer
  phiUses-transfer
  allUses-transfer
  allVars-transfer
  defAss-transfer
end

end

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

lemma *CFG-SSA-base-code-transfer-rules* [*transfer-rule*]:
  includes *lifting-syntax*
  assumes [*transfer-rule*]: *right-total G*
    and [*transfer-rule*]: (*G* ===> (=)) *αe αe2*
    and [*transfer-rule*]: (*G* ===> (=)) *αn αn2*
    and [*transfer-rule*]: (*G* ===> (=)) *invar invar2*
    and [*transfer-rule*]: (*G* ===> (=)) *inEdges′ inEdges2*
    and [*transfer-rule*]: (*G* ===> (=)) *Entry Entry2*

and [*transfer-rule*]: (*G* ===> (=)) *defs defs2*
and [*transfer-rule*]: (*G* ===> (=)) *uses uses2*
and [*transfer-rule*]: (*G* ===> (=)) *phis phis2*
shows (*G* ===> (=)) *phiDefs* (*CFG-SSA-base.phiDefs* (λ*g*. *Mapping.lookup* (*phis2 g*)))
(*G* ===> (=)) *allDefs* (*CFG-SSA-base.allDefs defs2* (λ*g*. *Mapping.lookup* (*phis2 g*)))
(*G* ===> (=)) *phiUses* (*CFG-SSA-base.phiUses* α*n2 inEdges2* (λ*g*. *Mapping.lookup* (*phis2 g*)))
(*G* ===> (=)) *allUses* (*CFG-SSA-base.allUses* α*n2 inEdges2* (*usesOf* ∘ *uses2*) (λ*g*. *Mapping.lookup* (*phis2 g*)))
(*G* ===> (=)) *defAss* (*CFG-SSA-base.defAss* α*n2 invar2 inEdges2 Entry2 defs2* (λ*g*. *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* ===> (=)) α*e1* α*e2*
and [*transfer-rule*]: (*G* ===> (=)) α*n1* α*n2*
and [*transfer-rule*]: (*G* ===> (=)) *invar1 invar2*
and [*transfer-rule*]: (*G* ===> (=)) *inEdges1 inEdges2*
and [*transfer-rule*]: (*G* ===> (=)) *Entry1 Entry2*
and [*transfer-rule*]: (*G* ===> (=)) *defs1 defs2*
and [*transfer-rule*]: (*G* ===> (=)) *uses1 uses2*
and [*transfer-rule*]: (*G* ===> (=)) *phis1 phis2*
shows *CFG-SSA* α*e1* α*n1 invar1 inEdges1 Entry1 defs1 uses1 phis1*
⟶ *CFG-SSA* α*e2* α*n2 invar2 inEdges2 Entry2 defs2 uses2 phis2*
⟨*proof*⟩

end

6.4 Code Equations for SSA Minimization

theory *Construct-SSA-notriv-code* **imports**
SSA-CFG-code
Construct-SSA-notriv
While-Combinator-Exts
begin

abbreviation (*input*) *const x* ≡ (λ-. *x*)

context *CFG-SSA-Transformed-notriv-base* **begin**
definition [*code*]: *substitution-code g next* = *the* (*the-trivial* (*snd next*) (*the* (*phis g next*)))
definition [*code*]: *substNext-code g next* ≡ λ*v*. *if v* = *snd next then substitution-code g next else v*

definition [*code*]: *uses'-code g next n* $\equiv$ *substNext-code g next* ` *uses g n*

lemma *substNext-code-alt-def*:
substNext-code g next = *id*(*snd next* := *substitution-code g next*)
$\langle$*proof*$\rangle$
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* αe αn *invar inEdges' Entry defs uses phis* +
CFG-SSA-Transformed-notriv-base αe αn *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
αe :: *'g* $\Rightarrow$ (*'node*::*linorder* $\times$ *'edgeD* $\times$ *'node*) *set* **and**
αn :: *'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'* {} (λ*ns*. *ns* $\cup$ *users*) (*Mapping.delete* (*snd next*) *nodes-of-uses*)
else Mapping.delete (*snd next*) *nodes-of-uses*)

definition [*code*]: *phis'-code g next* ≡ *map-values* (*λ*(*n*,*v*) *vs*. *if v* = *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 (*λn*. *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* ⇒ *'node* × *'val* ⇒ *'val* ⇒ (*'val*, (*'node* × *'val*) *set*) *mapping* ⇒ (*'val*, (*'node* × *'val*) *set*) *mapping*
where [*code*]: *nodes-of-phis' g next next' nodes-of-phis* =
(*let old-phis* = *Set.remove next* (*case-option* {} *id* (*Mapping.lookup nodes-of-phis* (*snd next*)));
nop = *Mapping.delete* (*snd next*) *nodes-of-phis*
in
Mapping.map-default next' {} (*λns*. (*Set.remove next ns*) ∪ *old-phis*) *nop*)

definition [*code*]: *triv-phis' g next triv-phis nodes-of-phis*
= (*Set.remove next triv-phis*) ∪ (*Set.filter* (*λn*. *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* ⇒ (*'val*, *'a set*) *mapping* ⇒ (*'val*, *'a set*) *mapping* ⇒ *bool* (- ⊢ - $\approx_\varphi$ - *50*)
where $g \vdash nou_1 \approx_\varphi nou_2$ ≡ ∀ *v* ∈ *Mapping.keys* (*ssa.phidefNodes g*). *case-option* {} *id* (*Mapping.lookup* nou_1 *v*) = *case-option* {} *id* (*Mapping.lookup* nou_2 *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* ⇒ (*'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**

$defs$ :: $'g \Rightarrow 'node \Rightarrow 'val::linorder\ set$ **and**
$uses$ :: $'g \Rightarrow 'node \Rightarrow 'val\ set$ **and**
$phis$ :: $'g \Rightarrow ('node, 'val)\ phis$ **and**
var :: $'g \Rightarrow 'val \Rightarrow 'var$ **and**
$chooseNext\text{-}all$ :: $('node \Rightarrow 'val\ set) \Rightarrow ('node, 'val)\ phis \Rightarrow 'g \Rightarrow ('node \times 'val)$
begin
lemma *isTrivial-the-trivial*: $\llbracket\ phi\ g\ v = Some\ vs;\ isTrivialPhi\ g\ v\ v'\ \rrbracket \Longrightarrow the\text{-}trivial\ v\ vs = Some\ v'$
⟨*proof*⟩

lemma *the-trivial-THE-isTrivial*: $\llbracket\ phi\ g\ v = Some\ vs;\ trivial\ g\ v\ \rrbracket \Longrightarrow the\text{-}trivial\ v\ vs = Some\ (The\ (isTrivialPhi\ g\ v))$
⟨*proof*⟩

lemma *substitution-code-correct*:
assumes *redundant g*
shows $substitution\ g = substitution\text{-}code\ g\ (chooseNext'\ g)$
⟨*proof*⟩

lemma *substNext-code-correct*:
assumes *redundant g*
shows $substNext\ g = substNext\text{-}code\ g\ (chooseNext'\ g)$
⟨*proof*⟩

lemma *uses'-code-correct*:
assumes *redundant g*
shows $uses'\ g = uses'\text{-}code\ g\ (chooseNext'\ g)$
⟨*proof*⟩

end

context *CFG-SSA-Transformed-notriv-linorder*
begin
lemma *substAll-terminates*: $while\text{-}option\ (cond\ g)\ (step\ g)\ (uses\ g,\ phis\ g) \neq None$
⟨*proof*⟩
end

locale *CFG-SSA-Transformed-notriv-linorder-code* =
$CFG\text{-}SSA\text{-}Transformed\text{-}code\ \alpha e\ \alpha n\ invar\ inEdges'\ Entry\ oldDefs\ oldUses\ defs\ uses\ phis\ var$
$+\ CFG\text{-}SSA\text{-}Transformed\text{-}notriv\text{-}base\text{-}code\ \alpha e\ \alpha n\ invar\ inEdges'\ Entry\ oldDefs\ oldUses\ defs\ uses\ phis\ var\ chooseNext\text{-}all$
$+\ CFG\text{-}SSA\text{-}Transformed\text{-}notriv\text{-}linorder\ \alpha e\ \alpha n\ invar\ inEdges'\ Entry\ oldDefs\ oldUses\ defs\ usesOf \circ uses\ \lambda g.\ Mapping.lookup\ (phis\ g)\ var$
$\lambda uses\ phis.\ chooseNext\text{-}all\ uses\ (Mapping.Mapping\ phis)$
for
αe :: $'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)\ set$ **and**
αn :: $'g \Rightarrow 'node\ list$ **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* **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 invar inEdges' Entry oldDefs oldUses defs u p var chooseNext-all*
for *u p*

⟨*proof*⟩

interpretation *uninst*: *CFG-SSA-Transformed-notriv-base* αe αn *invar inEdges′ Entry oldDefs oldUses defs u p var cN*
for *u p*
⟨*proof*⟩

lemma *phis′-code-correct*:
assumes *ssa.redundant g*
shows *phis′ g* = *Mapping.lookup* (*phis′-code g* (*chooseNext′ g*))
⟨*proof*⟩

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

lemma *uses′-ign*[*simp*]: *uninst-code.uses′-codem* (*const u*) *g* = *uninst-code.uses′-codem* (*u-g g u*) *g*
⟨*proof*⟩

lemma *phis′-ign*[*simp*]: *uninst-code.phis′-code* (*const p*) *g* = *uninst-code.phis′-code* (*phis*(*g*:=*p*)) *g*
⟨*proof*⟩

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

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

lemma *phis′-codem-correct*:
assumes *g* ⊢ *nodes-of-phis* $\approx_\varphi$ (*ssa.phiNodes-of g*) **and** *next* ∈ *Mapping.keys* (*phis g*)
shows *phis′-codem g next* (*substitution-code g next*) *nodes-of-phis* = *phis′-code g next*
⟨*proof*⟩

lemma *uses-transfer* [*transfer-rule*]: (*rel-fun* (=) (*pcr-mapping* (=) (=))) (λ*g n*. *Mapping.lookup* (*uses g*) *n*) *uses*
⟨*proof*⟩

lemma *uses′-codem-correct*:
assumes *g* ⊢ *nodes-of-uses* $\approx_\varphi$ *ssa.useNodes-of g* **and** *next* ∈ *Mapping.keys* (*phis g*)
shows *usesOf* (*uses′-codem g next* (*substitution-code g next*) *nodes-of-uses*) = *uses′-code g next*

$\langle proof \rangle$

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*
$\langle proof \rangle$

lemma *cN-transfer* [*transfer-rule*]: (*rel-fun* (=) (*rel-fun* (*pcr-mapping* (=) (=)) (=))) *cN chooseNext-all*
$\langle proof \rangle$

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

lemma *dom-phis'-codem*:
assumes $\bigwedge$*ns*. *Mapping.lookup nodes-of-phis* (*snd next*) = *Some ns* $\Longrightarrow$ *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*}
$\langle proof \rangle$

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

lemma *nodes-of-phis-finite* [*simplified*]:
assumes *g* $\vdash$ *nodes-of-phis* $\approx_\varphi$ *ssa.phiNodes-of g* **and** *Mapping.lookup nodes-of-phis v* = *Some ns* **and** *v* $\in$ *Mapping.keys* (*ssa.phidefNodes g*)
shows *finite ns*
$\langle proof \rangle$

lemma *lookup-phis'-codem-next*:
assumes $\bigwedge$*ns*. *Mapping.lookup nodes-of-phis* (*snd next*) = *Some ns* $\Longrightarrow$ *finite ns*
shows *Mapping.lookup* (*phis'-codem g next next' nodes-of-phis*) *next* = *None*
$\langle proof \rangle$

lemma *lookup-phis'-codem-other*:
assumes *g* $\vdash$ *nodes-of-phis* $\approx_\varphi$ (*ssa.phiNodes-of g*)
and *next* $\in$ *Mapping.keys* (*phis g*) **and** *next* $\neq \varphi$
shows *Mapping.lookup* (*phis'-codem g next* (*substitution-code g next*) *nodes-of-phis*) φ =
map-option (*map* (*substNext-code g next*)) (*Mapping.lookup* (*phis g*) φ)
$\langle proof \rangle$

lemma *lookup-nodes-of-phis'-subst* [*simp*]:
Mapping.lookup (*nodes-of-phis' g next* (*substitution-code g next*) *nodes-of-phis*) (*substitution-code g next*) =
Some ((*case-option* {} (*Set.remove next*) (*Mapping.lookup nodes-of-phis* (*substitution-code*

g next))) ∪ (*case-option* {} (*Set.remove next*) (*Mapping.lookup nodes-of-phis* (*snd next*))))
⟨*proof*⟩

lemma *lookup-nodes-of-phis′-not-subst*:
$v \neq$ *substitution-code g next* ⟹
Mapping.lookup (*nodes-of-phis′ g next* (*substitution-code g next*) *nodes-of-phis*) *v* = (*if v* = *snd next then None else Mapping.lookup nodes-of-phis v*)
⟨*proof*⟩

lemma *lookup-phis′-code*:
Mapping.lookup (*phis′-code g next*) *v* = (*if snd v* = *snd next then None else map-option* (*map* (*substNext-code g next*)) (*Mapping.lookup* (*phis g*) *v*))
⟨*proof*⟩

lemma *phi-equiv-mappingE′*:
assumes $g \vdash m_1 \approx_\varphi$ *ssa.phiNodes-of g*
and *Mapping.lookup* (*phis g*) *x* = *Some vs* **and** *b* ∈ *set vs* **and** *b* ∈ *snd* ‘ *Mapping.keys* (*phis g*)
obtains *Mapping.lookup* m_1 *b* = *Some* {*n* ∈ *Mapping.keys* (*phis g*). *b* ∈ *set* (*the* (*Mapping.lookup* (*phis g*) *n*))}
⟨*proof*⟩

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

lemma *phi-equiv-mappingE2′*:
assumes $g \vdash m_1 \approx_\varphi$ *ssa.phiNodes-of g*
and *b* ∈ *snd* ‘ *Mapping.keys* (*phis g*)
and ∀ φ ∈ *Mapping.keys* (*phis g*). *b* ∉ *set* (*the* (*Mapping.lookup* (*phis g*) φ))
shows *Mapping.lookup* m_1 *b* = *None* ∨ *Mapping.lookup* m_1 *b* = *Some* {}
⟨*proof*⟩

lemma *keys-phis′-codem* [*simp*]: *Mapping.keys* (*phis′-codem g next next′* (*ssa.phiNodes-of g*)) = *Mapping.keys* (*phis g*) − {*next*}
⟨*proof*⟩

lemma *keys-phis′-codem′*:
assumes *g* ⊢ *nodes-of-phis* $\approx_\varphi$ *ssa.phiNodes-of g* **and** *next* ∈ *Mapping.keys* (*phis g*)
shows *Mapping.keys* (*phis′-codem g next next′ nodes-of-phis*) = *Mapping.keys* (*phis g*) − {*next*}
⟨*proof*⟩

lemma *triv-phis′-correct*:

assumes $g \vdash$ *nodes-of-phis* $\approx_\varphi$ *ssa.phiNodes-of g* **and** *next* $\in$ *Mapping.keys* (*phis g*) **and** *ssa.trivial g* (*snd next*)
shows *uninst-code.triv-phis′* (*const* (*phis′-codem g next* (*substitution-code g next*) *nodes-of-phis*)) *g next* (*ssa.trivial-phis g*) *nodes-of-phis* = *uninst-code.ssa.trivial-phis* (*const* (*phis′-codem g next* (*substitution-code g next*) *nodes-of-phis*)) *g*
⟨*proof*⟩

lemma *nodes-of-phis′-correct*:
assumes $g \vdash$ *nodes-of-phis* $\approx_\varphi$ *ssa.phiNodes-of g*
and *next* $\in$ *Mapping.keys* (*phis g*) **and** *ssa.trivial g* (*snd next*)
shows $g \vdash$ (*nodes-of-phis′ g next* (*substitution-code g next*) *nodes-of-phis*) $\approx_\varphi$ (*uninst-code.ssa.phiNodes-of* (*const* (*phis′-codem g next* (*substitution-code g next*) *nodes-of-phis*)) *g*)
⟨*proof*⟩

lemma *nodes-of-uses′-correct*:
assumes $g \vdash$ *nodes-of-uses* $\approx_\varphi$ *ssa.useNodes-of g*
and *next* $\in$ *Mapping.keys* (*phis g*) **and** *ssa.trivial g* (*snd next*)
shows $g \vdash$ (*nodes-of-uses′ g next* (*substitution-code g next*) (*Mapping.keys* (*ssa.phidefNodes g*)) *nodes-of-uses*) $\approx_\varphi$ (*uninst-code.ssa.useNodes-of* (*const* (*uses′-codem g next* (*substitution-code g next*) *nodes-of-uses*)) *g*)
⟨*proof*⟩

definition[*code*]: *substAll-efficient g* ≡
let phiVals = *Mapping.keys* (*ssa.phidefNodes g*);
u = *uses g*;
p = *phis g*;
tp = *ssa.trivial-phis g*;
nou = *ssa.useNodes-of g*;
nop = *ssa.phiNodes-of g*
in
while
(λ((*u*,*p*),*triv-phis*,*nodes-of-uses*,*nodes-of-phis*). ¬ *Set.is-empty triv-phis*)
(λ((*u*,*p*),*triv-phis*,*nodes-of-uses*,*nodes-of-phis*). *let*
next = *Max triv-phis*;
next′ = *uninst-code.substitution-code* (*const p*) *g next*;
(*u′*,*p′*) = *uninst-code.step-codem* (*const u*) (*const p*) *g next next′ nodes-of-uses nodes-of-phis*;
tp′ = *uninst-code.triv-phis′* (*const p′*) *g next triv-phis nodes-of-phis*;
nou′ = *uninst-code.nodes-of-uses′ g next next′ phiVals nodes-of-uses*;
nop′ = *uninst-code.nodes-of-phis′ g next next′ nodes-of-phis*
in ((*u′*, *p′*), *tp′*, *nou′*, *nop′*))
((*u*, *p*), *tp*, *nou*, *nop*)

abbreviation *u-c x* ≡ *const* (*usesOf* (*fst x*))
abbreviation *p-c x* ≡ *const* (*Mapping.lookup* (*snd x*))
abbreviation *u g x* ≡ *u-g g* (*fst x*)
abbreviation *p g x* ≡ *p-g g* (*snd x*)

lemma *usesOf-upd* [*simp*]: (*usesOf* ∘ *u g s1*)(*g* := *usesOf us*) = *usesOf* ∘ *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′*: ⟦ *g* ⊢ *nodes-of-uses* $\approx_\varphi$ *ssa.useNodes-of g*; *next* ∈ *Mapping.keys* (*phis g*) ⟧
⟹ *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-αe :: (*'node*, *'edge*) *edge set*
gen-αn :: *'node list*
gen-inEdges :: *'node* ⇒ (*'node*, *'edge*) *edge list*
gen-Entry :: *'node*
gen-defs :: *'node* ⇒ *'var set*
gen-uses :: *'node* ⇒ *'var set*

abbreviation *trivial-gen-cfg ext* ≡ *gen-cfg-ext* {} [*undefined*] (*const* []) *undefined* (*const* {}) (*const* {}) *ext*
abbreviation (*input*) *ign f g* (-::*unit*) ≡ *f g*

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

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

abbreviation *gen-inEdges' g n* ≡ *map* (λ(*f*,*d*,*t*). (*f*,*d*)) (*gen-inEdges g n*)

lemma *gen-cfg-inhabited*: *let g* = *trivial-gen-cfg ext in CFG-wf* (*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*)
⟨*proof*⟩

typedef (*'node*, *'var*, *'edge*) *gen-cfg-wf* = {*g* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*) *gen-cfg*.
CFG-wf (*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*)}
⟨*proof*⟩

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

lift-definition *gen-wf-αn* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*) *gen-cfg-wf* ⇒ *'node list* **is** *gen-αn* ⟨*proof*⟩
lift-definition *gen-wf-αe* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*) *gen-cfg-wf* ⇒ (*'node*, *'edge*) *edge set* **is** *gen-αe* ⟨*proof*⟩
lift-definition *gen-wf-inEdges* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*) *gen-cfg-wf* ⇒ *'node* ⇒ (*'node*, *'edge*) *edge list* **is** *gen-inEdges* ⟨*proof*⟩
lift-definition *gen-wf-Entry* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*) *gen-cfg-wf* ⇒ *'node* **is** *gen-Entry* ⟨*proof*⟩
lift-definition *gen-wf-defs* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*) *gen-cfg-wf* ⇒ *'node* ⇒ *'var set* **is** *gen-defs* ⟨*proof*⟩
lift-definition *gen-wf-uses* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*) *gen-cfg-wf* ⇒ *'node* ⇒ *'var set* **is** *gen-uses* ⟨*proof*⟩

abbreviation *gen-wf-invar* ≡ *const True*
abbreviation *gen-wf-inEdges' g n* ≡ *map* (λ(*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 ⇒ *'node* ⇒ *'var set* **is** *gen-defs* ⟨*proof*⟩
lift-definition *gen-ssa-wf-uses* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*, *'val*::*linorder*) *gen-ssa-cfg-wf* ⇒ *'node* ⇒ *'var set* **is** *gen-uses* ⟨*proof*⟩
lift-definition *gen-ssa-wf-ssa-defs* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*, *'val*::*linorder*) *gen-ssa-cfg-wf* ⇒ *'node* ⇒ *'val set* **is** *gen-ssa-defs* ⟨*proof*⟩
lift-definition *gen-ssa-wf-ssa-uses* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*, *'val*::*linorder*) *gen-ssa-cfg-wf* ⇒ (*'node*, *'val set*) *mapping* **is** *gen-ssa-uses* ⟨*proof*⟩
lift-definition *gen-ssa-wf-phis* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*, *'val*::*linorder*) *gen-ssa-cfg-wf* ⇒ (*'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* ⇒ *'val* ⇒ *'var* **is** *gen-var* ⟨*proof*⟩

abbreviation *gen-ssa-wf-inEdges' g n* ≡ *map* (λ(*f*,*d*,*t*). (*f*,*d*)) (*gen-ssa-wf-inEdges g n*)

lemma *gen-ssa-wf-inEdges'-transfer* [*transfer-rule*]: *rel-fun cr-gen-ssa-cfg-wf* (=) *gen-inEdges' gen-ssa-wf-inEdges'*
⟨*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* ≡ *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-αe.transfer gen-ssa-wf-αn.transfer gen-ssa-wf-invar-trans gen-ssa-wf-inEdges′-transfer*, *transfer-rule*]
declare *CFG-SSA-base-code.CFG-SSA-base-code-transfer-rules*[*OF gen-ssa-cfg-wf.right-total gen-ssa-wf-αe.transfer gen-ssa-wf-αn.transfer gen-ssa-wf-invar-trans gen-ssa-wf-inEdges′-transfer gen-ssa-wf-Entry.transfer gen-ssa-wf-ssa-defs.transfer gen-ssa-wf-ssa-uses.transfer gen-ssa-wf-phis.transfer*, *transfer-rule*]

lemma *path2-ign*[*simp*]: *graph-path-base.path2* (*ign gen-αn g*) *gen-wf-invar* (*ign gen-inEdges′ g*) *g′ n ns m* ⟷ *graph-path-base.path2 gen-αn gen-wf-invar gen-inEdges′ g n ns m*
⟨*proof*⟩
lemma *allDefs-ign*[*simp*]: *CFG-SSA-base.allDefs* (*ign gen-ssa-defs g*) (*ign Mapping.lookup* (*gen-phis g*)) *ga n* = *CFG-SSA-base.allDefs gen-ssa-defs* (*λg. Mapping.lookup* (*gen-phis g*)) *g n*
⟨*proof*⟩
lemma *successors-ign*[*simp*]: *graph-path-base.successors* (*ign gen-αn g*) (*ign gen-inEdges′ g*) *ga n* = *graph-path-base.successors gen-αn gen-inEdges′ g n*
⟨*proof*⟩
lemma *predecessors-ign*[*simp*]: *graph-path-base.predecessors* (*ign gen-inEdges′ g*) *ga n* = *graph-path-base.predecessors gen-inEdges′ g n*
⟨*proof*⟩
lemma *phiDefs-ign*[*simp*]: *CFG-SSA-base.phiDefs* (*ign Mapping.lookup* (*gen-phis g*)) *ga* = *CFG-SSA-base.phiDefs* (*λg. Mapping.lookup* (*gen-phis g*)) *g*
⟨*proof*⟩
lemma *defAss-ign*[*simp*]: *CFG-SSA-base.defAss* (*ign gen-αn g*) *gen-wf-invar* (*ign gen-inEdges′ g*) (*ign gen-Entry g*) (*ign gen-ssa-defs g*) (*ign Mapping.lookup* (*gen-phis g*)) *ga*
= *CFG-SSA-base.defAss gen-αn gen-wf-invar gen-inEdges′ gen-Entry gen-ssa-defs* (*λg. Mapping.lookup* (*gen-phis g*)) *g*
⟨*proof*⟩
lemma *allUses-ign*[*simp*]: *CFG-SSA-base.allUses* (*ign gen-αn g*) (*ign gen-inEdges′ g*) (*usesOf* ∘ *ign gen-ssa-uses g*) (*ign Mapping.lookup* (*gen-phis g*)) *ga m*
= *CFG-SSA-base.allUses gen-αn gen-inEdges′* (*usesOf* ∘ *gen-ssa-uses*) (*λg. Mapping.lookup* (*gen-phis g*)) *g m*
⟨*proof*⟩
lemma *defAss′-ign*[*simp*]: *CFG-base.defAss′* (*ign gen-αn g*) *gen-wf-invar* (*ign gen-inEdges′ g*) (*ign gen-Entry g*) (*ign gen-defs g*) *ga*
= *CFG-base.defAss′ gen-αn gen-wf-invar gen-inEdges′ gen-Entry gen-defs g*
⟨*proof*⟩

global-interpretation *gen-ssa-wf-notriv*: *CFG-SSA-Transformed-notriv-linorder-code gen-ssa-wf-αe gen-ssa-wf-αn gen-wf-invar gen-ssa-wf-inEdges′ gen-ssa-wf-Entry gen-ssa-wf-defs gen-ssa-wf-uses gen-ssa-wf-ssa-defs gen-ssa-wf-ssa-uses gen-ssa-wf-phis gen-ssa-wf-var uninst-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*

$\langle proof \rangle$

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′*
$\langle proof \rangle$

lift-definition *gen-cfg-wf-extend* :: (*′a*::*linorder*, *′b*::*linorder*, *′c*) *gen-cfg-wf* ⇒ *′d* ⇒ (*′a*, *′b*, *′c*, *′d*) *gen-cfg-scheme*
is *gen-cfg.extend* $\langle proof \rangle$

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*
$\langle proof \rangle$

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*
$\langle proof \rangle$

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*
$\langle proof \rangle$

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*
$\langle proof \rangle$

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* (|*gen-ssa-defs* = *d*, *gen-ssa-uses* = *u*, *gen-phis* = *p*, *gen-var* = *v*|))
= *gen-wf-uses gen-cfg-wf*
⟨*proof*⟩

lemma *gen-ssa-defs-wf-extend* [*simp*]:
gen-ssa-defs (*gen-cfg-wf-extend gen-cfg-wf* (|*gen-ssa-defs* = *d*, *gen-ssa-uses* = *u*, *gen-phis* = *p*, *gen-var* = *v*|))
= *d*
⟨*proof*⟩

lemma *gen-ssa-uses-wf-extend* [*simp*]:
gen-ssa-uses (*gen-cfg-wf-extend gen-cfg-wf* (|*gen-ssa-defs* = *d*, *gen-ssa-uses* = *u*, *gen-phis* = *p*, *gen-var* = *v*|))
= *u*
⟨*proof*⟩

lemma *gen-phis-wf-extend* [*simp*]:
gen-phis (*gen-cfg-wf-extend gen-cfg-wf* (|*gen-ssa-defs* = *d*, *gen-ssa-uses* = *u*, *gen-phis* = *p*, *gen-var* = *v*|))
= *p*
⟨*proof*⟩

lemma *gen-var-wf-extend* [*simp*]:
gen-var (*gen-cfg-wf-extend gen-cfg-wf* (|*gen-ssa-defs* = *d*, *gen-ssa-uses* = *u*, *gen-phis* = *p*, *gen-var* = *v*|))
= *v*
⟨*proof*⟩

lemma *CFG-SSA-Transformed-codeI*:
assumes *CFG-SSA-Transformed* αe αn *invar inEdges Entry oldDefs oldUses defs* ($\lambda g.$ *lookup-multimap* (*uses g*)) ($\lambda g.$ *Mapping.lookup* (*phis g*)) *var*
and $\bigwedge g.$ *Mapping.keys* (*uses g*) $\subseteq$ *set* (αn *g*)
shows *CFG-SSA-Transformed-code* αe αn *invar inEdges Entry oldDefs oldUses defs uses phis var*
⟨*proof*⟩

lemma *CFG-SSA-Transformed-ign*:
CFG-SSA-Transformed (*ign gen-wf-*αe *gen-cfg-wf*) (*ign gen-wf-*αn *gen-cfg-wf*) *gen-wf-invar*
(*const* (*gen-wf-inEdges′ gen-cfg-wf*)) (*ign gen-wf-Entry gen-cfg-wf*) (*ign gen-wf-defs gen-cfg-wf*)
(*ign gen-wf-uses gen-cfg-wf*) (*ign gen-wf-defs′ gen-cfg-wf*) (*ign gen-wf.uses′ gen-cfg-wf*)
(*ign gen-wf.phis′ gen-cfg-wf*)
(*ign gen-wf-var gen-cfg-wf*)

$\langle proof \rangle$

lift-definition *gen-ssa-cfg-wf* :: (*'node*::*linorder*, *'var*::*linorder*, *'edge*) *gen-cfg-wf*
$\Rightarrow$ (*'node*, *'var*, *'edge* , (*'node*,*'var*) *ssaVal*) *gen-ssa-cfg-wf*
is λg. *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*
⦈
$\langle proof \rangle$

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* $\equiv$ *map-keys gen-wf.from-sparse*

declare *map-keys-code*[*OF gen-wf.from-sparse-inj*, *folded map-keys-from-sparse-def*, *code*]
declare *map-keys-from-sparse-def*[*symmetric*, *code-unfold*]

lemma *fold-Cons-commute*: $(\bigwedge a\ b.\ [\![a \in set\ (x\ \#\ xs);\ b \in set\ (x\ \#\ xs)]\!] \Longrightarrow f\ a \circ f\ b = f\ b \circ f\ a)$
$\Longrightarrow fold\ f\ (x\ \#\ xs) = f\ x \circ (fold\ f\ xs)$
$\langle proof \rangle$

lemma *Union-of-code* [*code*]: $Union\text{-}of\ f\ (RBT\text{-}Set.Set\ r) = RBT.fold\ (\lambda a\ \text{-}.\ (\cup)\ (f\ a))\ r\ \{\}$
$\langle proof \rangle$

definition[*code*]: $disjoint\ xs\ ys = (xs \cap ys = \{\})$

definition $gen\text{-}ssa\text{-}wf\text{-}notriv\text{-}substAll' = fst \circ gen\text{-}ssa\text{-}wf\text{-}notriv\text{-}substAll\text{-}efficient$

definition $fold\text{-}set\ f\ A \equiv fold\ f\ (sorted\text{-}list\text{-}of\text{-}set\ A)$
declare *fold-set-def* [*symmetric*, *code-unfold*]

declare *fold-set-def*
[**where** *A=RBT-Set.Set r* **for** *r*,
unfolded sorted-list-set fold-keys-def-alt [*symmetric,abs-def*] *fold-keys-def* [*abs-def*], *code*]

declare *graph-path-base.inEdges-def* [*code*]

end

theory *Generic-Extract* **imports**
Generic-Interpretation
begin

export-code open
set sorted-list-of-set disjoint RBT.fold
gen-ssa-cfg-wf gen-wf-var gen-ssa-wf-notriv-substAll′
in *OCaml* **module-name** *BraunSSA*

end

theory *Disjoin-Transform* **imports**
Slicing.AdditionalLemmas
begin

inductive *subcmd* :: *cmd* ⇒ *cmd* ⇒ *bool* **where**
sub-Skip: *subcmd c Skip*
| *sub-Base*: *subcmd c c*
| *sub-Seq1*: *subcmd c1 c* ⟹ *subcmd* (*c1*;;*c2*) *c*
| *sub-Seq2*: *subcmd c2 c* ⟹ *subcmd* (*c1*;;*c2*) *c*
| *sub-If1*: *subcmd c1 c* ⟹ *subcmd* (*if* (*b*) *c1 else c2*) *c*
| *sub-If2*:*subcmd c2 c* ⟹ *subcmd* (*if* (*b*) *c1 else c2*) *c*
| *sub-While*: *subcmd c′ c* ⟹ *subcmd* (*while* (*b*) *c′*) *c*

fun *maxVnameLen-aux* :: *expr* ⇒ *nat* **where**
maxVnameLen-aux (*Val -*) = *0*
| *maxVnameLen-aux* (*Var V*) = *length V*
| *maxVnameLen-aux* (*e1* « - » *e2*) = *max* (*maxVnameLen-aux e1*) (*maxVnameLen-aux e2*)

fun *maxVnameLen* :: *cmd* ⇒ *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* ⇒ *vname* **where** *tempName c* ≡ *replicate* (*Suc* (*maxVnameLen c*)) (*CHR ''a''*)

inductive *newname* :: *cmd* ⇒ *vname* ⇒ *bool* **where**
  newname Skip V
| $V \notin \{V'\} \cup$ *rhs-aux e* ⟹ *newname* (*V′:=e*) *V*
| ⟦*newname c1 V*; *newname c2 V*⟧ ⟹ *newname* (*c1;;c2*) *V*
| ⟦*newname c1 V*; *newname c2 V*; $V \notin$ *rhs-aux b*⟧ ⟹ *newname* (*if* (*b*) *c1 else c2*) *V*
| ⟦*newname c V*; $V \notin$ *rhs-aux b*⟧ ⟹ *newname* (*while* (*b*) *c*) *V*

lemma *maxVnameLen-aux-newname*: *length V* > *maxVnameLen-aux e* ⟹ $V \notin$ *rhs-aux e*
⟨*proof*⟩

lemma *maxVnameLen-newname*: *length V* > *maxVnameLen c* ⟹ *newname c V*
⟨*proof*⟩

lemma *tempname-newname*[*intro*]: *newname c* (*tempName c*)
  ⟨*proof*⟩

fun *transform-aux* :: *vname* ⇒ *cmd* ⇒ *cmd* **where**
  transform-aux - Skip = *Skip*
| *transform-aux V′* (*V:=e*) =
    (*if* $V \in$ *rhs* (*V:=e*) *then V′:=e;; V:=Var V′*
     else V:=e)
| *transform-aux V′* (c_1;;c_2) = *transform-aux V′* c_1;; *transform-aux V′* c_2
| *transform-aux V′* (*if* (*b*) c_1 *else* c_2) =
    (*if* (*b*) *transform-aux V′* c_1 *else transform-aux V′* c_2)
| *transform-aux V′* (*while* (*b*) *c*) = (*while* (*b*) *transform-aux V′ c*)

abbreviation *transform* :: *cmd* ⇒ *cmd* **where**
  transform c ≡ *transform-aux* (*tempName c*) *c*

fun *leftmostCmd* :: *cmd* ⇒ *cmd* **where**
  leftmostCmd (*c1;;c2*) = *leftmostCmd c1*
| *leftmostCmd c* = *c*

lemma *leftmost-lhs*[*simp*]: *lhs* (*leftmostCmd c*) = *lhs c*
⟨*proof*⟩

lemma *leftmost-rhs*[*simp*]: *rhs* (*leftmostCmd c*) = *rhs c*
⟨*proof*⟩

lemma *leftmost-subcmd*[*intro*]: *subcmd c* (*leftmostCmd c*)
⟨*proof*⟩

lemma *leftmost-labels*: *labels c n c′* ⟹ *subcmd c* (*leftmostCmd c′*)
⟨*proof*⟩

theorem *transform-disjoint*:
  assumes *subcmd* (*transform-aux temp c*) (*V*:=*e*) *newname c temp*
  shows $V \notin$ *rhs-aux e*
⟨*proof*⟩

lemma *transform-disjoint'*: *subcmd* (*transform c*) (*leftmostCmd c'*) $\Longrightarrow$ *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* = *True*
| (- *n* -) $\leq$ *x* = (*case x of*
    (-*Entry*-) $\Rightarrow$ *False*
  | (- *m* -) $\Rightarrow$ $n \leq m$
  | (-*Exit*-) $\Rightarrow$ *True*)
| (-*Exit*-) $\leq$ *x* = (*x* = (-*Exit*-))

fun *less-w-node* **where**
  (-*Entry*-) < *x* = (*x* $\neq$ (-*Entry*-))
| (- *n* -) < *x* = (*case x of*
    (-*Entry*-) $\Rightarrow$ *False*
  | (- *m* -) $\Rightarrow$ $n < m$
  | (-*Exit*-) $\Rightarrow$ *True*)
| (-*Exit*-) < *x* = *False*

instance ⟨*proof*⟩
end

instance *w-node* :: *linorder* ⟨*proof*⟩

declare *Defs.simps* [*simp del*]
declare *Uses.simps* [*simp del*]
declare *Let-def* [*simp*]

declare *finite-valid-nodes* [*simp*, *intro*!]

lemma *finite-valid-edge* [*simp*, *intro*!]: *finite* (*Collect* (*valid-edge c*))
⟨*proof*⟩

lemma *uses-expr-finite*: *finite* (*rhs-aux e*)
⟨*proof*⟩

lemma *uses-cmd-finite*: *finite* (*rhs c*)
⟨*proof*⟩

lemma *defs-cmd-finite*: *finite* (*lhs c*)
⟨*proof*⟩

lemma *finite-labels′*: *finite* {(*l*,*c*). *labels prog l c*}
⟨*proof*⟩

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

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

definition *while-cfg-*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* ∧ *set ls* = {(*sourcenode e*, *kind e*)| *e. valid-edge* (*transform c*) *e* ∧ *targetnode e* = *t*})
definition *while-cfg-Entry c* = (*-Entry-*)
definition *while-cfg-defs c* = (*Defs* (*transform c*))((*-Entry-*) := {*v*. ∃ *n*. *v* ∈ *Uses* (*transform c*) *n*})
definition *while-cfg-uses c* = *Uses* (*transform c*)

abbreviation *while-cfg-inEdges c t* ≡ *map* (λ(*f*,*d*). (*f*,*d*,*t*)) (*while-cfg-inEdges′ c t*)

lemmas *while-cfg-defs* = *while-cfg-*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* (λ*x y*. *x* = *c*)

$\langle proof \rangle$

lemma *const-transfer*: *rel-fun* $(\lambda x\ y.\ x = c)$ $(=)$ f $(\lambda$-. $f\ c)$
$\langle proof \rangle$

interpretation *while-ign*: *graph-path* λ-. *while-cfg-αe cmd* λ-. *while-cfg-αn cmd* λ-. *while-cfg-invar cmd* λ-. *while-cfg-inEdges' cmd*
$\langle proof \rangle$

definition *gen-while-cfg g* $\equiv$ (|
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*
|)

lemma *while-path-graph-pathD*: *While-CFG.path* (*transform c*) *n es m* $\Longrightarrow$ *while.path2 c n* (*n#map targetnode es*) *m*
$\langle proof \rangle$

lemma *Uses-Entry* [*simp*]: *Uses c* (*-Entry-*) = {}
$\langle proof \rangle$

lemma *in-Uses-valid-node*: $V \in$ *Uses c n* $\Longrightarrow$ *valid-node c n*
$\langle proof \rangle$

lemma *while-cfg-CFG-wf-impl*:
SSA-CFG.CFG-wf (λ-. *gen-αe* (*gen-while-cfg cmd*)) (λ-. *gen-αn* (*gen-while-cfg cmd*))
(λ-. *while-cfg-invar cmd*) (λ-. *gen-inEdges'* (*gen-while-cfg cmd*))
(λ-. *gen-Entry* (*gen-while-cfg cmd*)) (λ-. *gen-defs* (*gen-while-cfg cmd*))
(λ-. *gen-uses* (*gen-while-cfg cmd*))
$\langle proof \rangle$

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

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.