Minimal Static Single Assignment Form

Max Wagner Denis Lohner

March 17, 2025

Abstract

This formalization is an extension to [3]. In their work, the authors have shown that Braun et al.'s static single assignment (SSA) construction algorithm [1] produces minimal SSA form for input programs with a reducible control flow graph (CFG). However Braun et al. also proposed an extension to their algorithm that they claim produces minimal SSA form even for irreducible CFGs. In this formalization we support that claim by giving a mechanized proof.

As the extension of Braun et al.'s algorithm aims for removing so-called *redundant strongly connected components* (sccs) of ϕ functions, we show that this suffices to guarantee minimality according to Cytron et al. [2].

Contents

1 Minimality under Irreducible Control Flow

Braun et al. [1] provide an extension to the original construction algorithm to ensure minimality according to Cytron's definition even in the case of irreducible control flow. This extension establishes the property of being *redundant-scc-free*, i.e. the resulting graph G contains no subsets inducing a strongly connected subgraph G' via ϕ functions such that G' has less than two ϕ arguments in $G \setminus G'$. In this section we will show that a graph with this property is Cytron-minimal.

Our formalization follows the proof sketch given in [1]. We first provide a formal proof of Lemma 1 from [1] which states that every redundant set of ϕ functions contains at least one redundant SCC. A redundant set of ϕ functions is a set P of ϕ functions with $P \cup \{v\} \supseteq A$, where A is the union over all ϕ functions arguments contained in P. I.e. P references at most one SSA value (v) outside P. A redundant SCC is a redundant set that is strongly connected according to the *is-argument* relation.

Next, we show that a CFG in SSA form without redundant sets of ϕ functions is Cytron-minimal.

Finally putting those results together, we conclude that the extension to Braun et al.'s algorithm always produces minimal SSA form.

```
theory Irreducible
  imports Formal-SSA.Minimality
begin

context CFG-SSA-Transformed
begin
```

1.1 Proof of Lemma 1 from Braun et al.

To preserve readability, we won't distinguish between graph nodes and the ϕ functions contained inside such a node.

The graph induced by the ϕ network contained in the vertex set P. Note that the edges of this graph are not necessarily a subset of the edges of the input graph.

definition *induced-phi-graph g P* ≡ {(*φ*,*φ'*). *phiArg g φ φ'*} ∩ *P* × *P*

For the purposes of this section, we define a "redundant set" as a nonempty set of ϕ functions with at most one ϕ argument outside itself. A redundant SCC is defined analogously. Note that since any uses of values in a redundant set can be replaced by uses of its singular argument (without modifying program semantics), the name is adequate.

definition *redundant-set g P* ≡ *P* ≠ {} ∧ *P* ⊆ *dom* (*phi g*) ∧ (∃ *v'* ∈ *allVars g*. ∀ *φ* ∈ *P*. ∀ *φ'*. *phiArg g φ φ'* ⟶ *φ'* ∈ *P* ∪ {*v'*})
definition *redundant-scc g P scc* ≡ *redundant-set g scc* ∧ *is-scc* (*induced-phi-graph g P*) *scc*

We prove an important lemma via condensation graphs of ϕ networks, so the relevant definitions are introduced here.

definition *condensation-nodes g P* ≡ *scc-of* (*induced-phi-graph g P*) ‘ *P*
definition *condensation-edges g P* ≡ ((λ(*x*,*y*). (*scc-of* (*induced-phi-graph g P*) *x*, *scc-of* (*induced-phi-graph g P*) *y*)) ‘ (*induced-phi-graph g P*)) − *Id*

For a finite P, the condensation graph induced by P is finite and acyclic.

lemma *condensation-finite*: *finite* (*condensation-edges g P*)

The set of edges of the condensation graph, spanning at most all ϕ nodes and their arguments (both of which are finite sets), is finite itself.

```
proof −
  let ?phiEdges={(a,b). phiArg g a b}
  have finite ?phiEdges
  proof −
    let ?phiDomRan=(dom (phi g) × ⋃ (set ‘ (ran (phi g))))
    from phi-finite
    have finite ?phiDomRan by (simp add: imageE phi-finite map-dom-ran-finite)
    have ?phiEdges ⊆ ?phiDomRan
     apply (rule subst[of ∀ a ∈ ?phiEdges. a ∈ ?phiDomRan])
      apply (simp-all add: subset-eq[symmetric] phiArg-def)
     by (auto simp: ran-def)
    with ‹finite ?phiDomRan›
    show finite ?phiEdges by (rule Finite-Set.rev-finite-subset)
  qed
  hence ⋀f. finite (f ` (?phiEdges ∩ (P × P))) by auto
 thus finite (condensation-edges g P) unfolding condensation-edges-def induced-phi-graph-def
by auto
qed
```

auxiliary lemmas for acyclicity

```
lemma condensation-nodes-edges: (condensation-edges g P) ⊆ (condensation-nodes
g P × condensation-nodes g P)
unfolding condensation-edges-def condensation-nodes-def induced-phi-graph-def
by auto


lemma condensation-edge-impl-path:
assumes (a, b) ∈ (condensation-edges g P)
assumes (φ_a ∈ a)
assumes (φ_b ∈ b)
shows (φ_a, φ_b) ∈ (induced-phi-graph g P)^*
unfolding condensation-edges-def
proof −
  from assms(1)
  obtain x y where x-y-props:
    (x, y) ∈ (induced-phi-graph g P)
    a = scc-of (induced-phi-graph g P) x
    b = scc-of (induced-phi-graph g P) y
   unfolding condensation-edges-def by auto
  hence x ∈ a y ∈ b by auto
```

All that's left is to combine these paths.

```
  with assms(2) x-y-props(2)
 have (φ_a, x) ∈ (induced-phi-graph g P)^* by (meson is-scc-connected scc-of-is-scc)
  moreover with assms(3) x-y-props(3) ‹y ∈ b›
 have (y, φ_b) ∈ (induced-phi-graph g P)^* by (meson is-scc-connected scc-of-is-scc)
  ultimately
  show (φ_a, φ_b) ∈ (induced-phi-graph g P)^* using x-y-props(1)  by auto
qed


lemma path-in-condensation-impl-path:
assumes (a, b) ∈ (condensation-edges g P)^+
assumes (φ_a ∈ a)
assumes (φ_b ∈ b)
shows (φ_a, φ_b) ∈ (induced-phi-graph g P)^*
using assms
proof (induction arbitrary: φ_b rule:trancl-induct)
  fix y z φ_b
  assume (y, z) ∈ condensation-edges g P
```

hence *is-scc* (*induced-phi-graph g P*) *y* **unfolding** *condensation-edges-def* **by** *auto*
hence $\exists\varphi_y.\ \varphi_y \in y$ **using** *scc-non-empty′* **by** *auto*
then obtain φ_y **where** φ_y*-in-y*: $\varphi_y \in y$ **by** *auto*

assume φ_b*-elem*: $\varphi_b \in z$
assume $\bigwedge\varphi_b.\ \varphi_a \in a \Longrightarrow \varphi_b \in y \Longrightarrow (\varphi_a, \varphi_b) \in (\textit{induced-phi-graph } g\ P)^*$
with *assms*(*2*) φ_y*-in-y*
have φ_a*-to-*φ_y: $(\varphi_a, \varphi_y) \in (\textit{induced-phi-graph } g\ P)^*$ **using** *condensation-edge-impl-path* **by** *auto*

from φ_b*-elem* φ_y*-in-y* ‹(*y*, *z*) ∈ *condensation-edges g P*›
have $(\varphi_y, \varphi_b) \in (\textit{induced-phi-graph } g\ P)^*$ **using** *condensation-edge-impl-path* **by** *auto*
with φ_a*-to-*φ_y
show $(\varphi_a, \varphi_b) \in (\textit{induced-phi-graph } g\ P)^*$ **by** *auto*
qed (*auto intro*:*condensation-edge-impl-path*)

lemma *condensation-acyclic*: *acyclic* (*condensation-edges g P*)
proof (*rule acyclicI*, *rule allI*, *rule ccontr*, *simp*)
fix *x*

Assume there is a cycle in the condensation graph.

assume *cyclic*: $(x, x) \in (\textit{condensation-edges } g\ P)^+$
have *nonrefl*: $(x, x) \notin (\textit{condensation-edges } g\ P)$ **unfolding** *condensation-edges-def* **by** *auto*

Then there must be a second SCC *b* on this path.

from *this cyclic*
obtain *b* **where** *b-on-path*: $(x, b) \in (\textit{condensation-edges } g\ P)$ $(b, x) \in (\textit{condensation-edges } g\ P)^+$
by (*meson converse-tranclE*)

hence *x* ∈ (*condensation-nodes g P*) *b* ∈ (*condensation-nodes g P*) **using** *condensation-nodes-edges* **by** *auto*
hence *nodes-are-scc*: *is-scc* (*induced-phi-graph g P*) *x is-scc* (*induced-phi-graph g P*) *b*
using *scc-of-is-scc* **unfolding** *induced-phi-graph-def condensation-nodes-def* **by** *auto*

However, the existence of this path means all nodes in *b* and *x* are mutually reachable.

have $\exists\varphi_x.\ \varphi_x \in x\ \exists\varphi_b.\ \varphi_b \in b$ **using** *nodes-are-scc scc-non-empty′ ex-in-conv* **by** *auto*
then obtain φ_x φ_b **where** φ*xb-elem*: $\varphi_x \in x$ $\varphi_b \in b$ **by** *metis*
with *nodes-are-scc*(*1*) *b-on-path path-in-condensation-impl-path condensation-edge-impl-path* φ*xb-elem*(*2*)
have $\varphi_b \in x$

```
  by − (rule is-scc-closed)
```

This however means x and b must be the same SCC, which is a contradiction to the nonreflexivity of *condensation-edges*.

```
  with nodes-are-scc φxb-elem
  have x = b using is-scc-unique[of induced-phi-graph g P] by simp
  hence (x, x) ∈ (condensation-edges g P) using b-on-path by simp
  with nonrefl
  show False by simp
qed
```

Since the condensation graph of a set is acyclic and finite, it must have a leaf.

```
lemma Ex-condensation-leaf:
assumes P ≠ {}
shows ∃ leaf. leaf ∈ (condensation-nodes g P) ∧ (∀ scc.(leaf, scc) ∉ condensation-edges g P)
proof −
  from assms obtain x where x ∈ condensation-nodes g P unfolding condensation-nodes-def by auto
  show ?thesis
  proof (rule wfE-min)
    from condensation-finite condensation-acyclic
    show wf ((condensation-edges g P)^-1) by (rule finite-acyclic-wf-converse)
  next
    fix leaf
    assume leaf-node: leaf ∈ condensation-nodes g P
    moreover
   assume leaf-is-leaf: scc ∉ condensation-nodes g P if (scc, leaf) ∈ (condensation-edges g P)^-1 for scc
    ultimately
    have leaf ∈ condensation-nodes g P ∧ (∀ scc. (leaf, scc) ∉ condensation-edges g P) using condensation-nodes-edges by blast
     thus ∃ leaf. leaf ∈ condensation-nodes g P ∧ (∀ scc. (leaf, scc) ∉ condensation-edges g P) by blast
  qed fact
qed


lemma scc-in-P:
assumes scc ∈ condensation-nodes g P
shows scc ⊆ P
proof −
  have scc ⊆ P if y-props: scc = scc-of (induced-phi-graph g P) n n ∈ P for n
  proof −
    from y-props
    show scc ⊆ P
    proof (clarsimp simp:y-props(1); case-tac n = x)
      fix x
      assume different: n ≠ x
      assume x ∈ scc-of (induced-phi-graph g P) n
    hence (n, x) ∈ (induced-phi-graph g P)^* by (metis is-scc-connected scc-of-is-scc node-in-scc-of-node)
      with different
      have (n, x) ∈ (induced-phi-graph g P)^+ by (metis rtranclD)
        then obtain z where step: (z, x) ∈ (induced-phi-graph g P) by (meson tranclE)
      from step
      show x ∈ P unfolding induced-phi-graph-def by auto
    qed simp
  qed
  from this assms(1) have x ∈ P if x-node: x ∈ scc for x
   apply −
   apply (rule imageE[of scc scc-of (induced-phi-graph g P)])
    using condensation-nodes-def x-node by blast+
  thus ?thesis by clarify
qed
```

```
lemma redundant-scc-phis:
assumes redundant-set g P scc ∈ condensation-nodes g P x ∈ scc
shows phi g x ≠ None
using assms by (meson domIff redundant-set-def scc-in-P subsetCE)
```

The following lemma will be important for the main proof of this section. If P is redundant, a leaf in the condensation graph induced by P corresponds to a strongly connected set with at most one argument, thus a redundant strongly connected set exists.

Lemma 1. Every redundant set contains a redundant SCC.

```
lemma 1:
assumes redundant-set g P
shows ∃ scc ⊆ P. redundant-scc g P scc
proof −
  from assms Ex-condensation-leaf[of P g]
  obtain leaf where leaf-props: leaf ∈ (condensation-nodes g P) ∀ scc. (leaf, scc) ∉ condensation-edges g P
   unfolding redundant-set-def by auto
  hence is-scc (induced-phi-graph g P) leaf unfolding condensation-nodes-def by auto
  moreover
  hence leaf ≠ {} by (rule scc-non-empty′)
  moreover
  have leaf ⊆ dom (phi g)
    apply (subst subset-eq, rule ballI)
    using redundant-scc-phis leaf-props(1) assms(1) by auto
  moreover
  from assms
  obtain pred where pred-props: pred ∈ allVars g ∀ φ∈P. ∀ φ′. phiArg g φ φ′ ⟶ φ′ ∈ P ∪ {pred} unfolding redundant-set-def by auto
  {
```

Any argument of a ϕ function in the leaf SCC which is *not* in the leaf SCC itself must be the unique argument of P

```
    fix φ φ'

    consider (in-P) φ' ∉ leaf ∧ φ' ∈ P | (neither) φ' ∉ leaf ∧ φ' ∉ P ∪ {pred} | φ' ∉ leaf ∧ φ' ∈ {pred} | φ' ∈ leaf by auto
    hence φ' ∈ leaf ∪ {pred} if φ ∈ leaf and phiArg g φ φ'
    proof cases
      case in-P — In this case leaf wasn't really a leaf, a contradiction
      moreover
      from in-P that leaf-props(1) scc-in-P[of leaf g P]
       have (φ, φ') ∈ induced-phi-graph g P unfolding induced-phi-graph-def by auto
      ultimately
        have (leaf, scc-of (induced-phi-graph g P) φ') ∈ condensation-edges g P unfolding condensation-edges-def
       using leaf-props(1) that ‹is-scc (induced-phi-graph g P) leaf›
       apply −
       apply clarsimp
       apply (rule conjI)
        prefer 2
        apply auto[1]
       unfolding condensation-nodes-def
         by (metis (no-types, lifting) is-scc-unique node-in-scc-of-node pair-imageI scc-of-is-scc)
      with leaf-props(2)
      show ?thesis by auto
    next
      case neither — In which case P itself wasn't redundant, a contradiction
      with that leaf-props pred-props
      have ¬redundant-set g P unfolding redundant-set-def
        by (meson rev-subsetD scc-in-P)
      with assms
      show ?thesis by auto
    qed auto — the other cases are trivial
  }
  with pred-props(1)
  have ∃ v'∈allVars g. ∀ φ∈leaf. ∀ φ'. phiArg g φ φ' ⟶ φ' ∈ leaf ∪ {v'} by auto
  ultimately
  have redundant-scc g P leaf unfolding redundant-scc-def redundant-set-def by auto
  thus ?thesis using leaf-props(1) scc-in-P by meson
qed
```

1.2 Proof of Minimality

We inductively define the reachable-set of a ϕ function as all ϕ functions reachable from a given node via an unbroken chain of ϕ argument edges to unnecessary ϕ functions.

inductive-set $\mathit{reachable} :: {'g} \Rightarrow {'val} \Rightarrow {'val}\ \mathit{set}$

for $g :: {'g}$ **and** $\varphi :: {'val}$

where refl: $\mathit{unnecessaryPhi}\ g\ \varphi \Longrightarrow \varphi \in \mathit{reachable}\ g\ \varphi$

$|\ \mathit{step}$: $\varphi' \in \mathit{reachable}\ g\ \varphi \Longrightarrow \mathit{phiArg}\ g\ \varphi'\ \varphi'' \Longrightarrow \mathit{unnecessaryPhi}\ g\ \varphi'' \Longrightarrow \varphi'' \in \mathit{reachable}\ g\ \varphi$

lemma *reachable-props*:

assumes $\varphi' \in \mathit{reachable}\ g\ \varphi$

shows $(\mathit{phiArg}\ g)^{**}\ \varphi\ \varphi'$ **and** $\mathit{unnecessaryPhi}\ g\ \varphi'$

using *assms*

by $(\mathit{induction}\ \varphi'\ \mathit{rule}{:}\ \mathit{reachable.induct})\ \mathit{auto}$

We call the transitive arguments of a ϕ function not in its reachable-set the "true arguments" of this ϕ function.

definition $[\mathit{simp}]$: $\mathit{trueArgs}\ g\ \varphi \equiv \{\varphi'.\ \varphi' \notin \mathit{reachable}\ g\ \varphi\} \cap \{\varphi'.\ \exists\varphi'' \in \mathit{reachable}\ g\ \varphi.\ \mathit{phiArg}\ g\ \varphi''\ \varphi'\}$

lemma *preds-finite*: $\mathit{finite}\ (\mathit{trueArgs}\ g\ \varphi)$

proof $(\mathit{rule}\ \mathit{ccontr})$

assume $\mathit{infinite}\ (\mathit{trueArgs}\ g\ \varphi)$

hence a: $\mathit{infinite}\ \{\varphi'.\ \exists\varphi'' \in \mathit{reachable}\ g\ \varphi.\ \mathit{phiArg}\ g\ \varphi''\ \varphi'\}$ **by** *auto*

have *phiarg-set*: $\{\varphi'.\ \exists\varphi.\ \mathit{phiArg}\ g\ \varphi\ \varphi'\} = \bigcup\ (\mathit{set}\ `\{b.\ \exists a.\ \mathit{phi}\ g\ a = \mathit{Some}\ b\})$
unfolding *phiArg-def* **by** *auto*

If the true arguments of a ϕ function are infinite in number, there must be an infinite number of ϕ functions...

have $\mathit{infinite}\ \{\varphi'.\ \exists\varphi.\ \mathit{phiArg}\ g\ \varphi\ \varphi'\}$

by $(\mathit{rule}\ \mathit{infinite\text{-}super}[\mathit{of}\ \{\varphi'.\ \exists\varphi'' \in \mathit{reachable}\ g\ \varphi.\ \mathit{phiArg}\ g\ \varphi''\ \varphi'\}])\ (\mathit{auto}\ \mathit{simp}{:}\ a)$

with *phiarg-set*

have $\mathit{infinite}\ (\mathit{ran}\ (\mathit{phi}\ g))$ **unfolding** *ran-def phiArg-def* **by** *clarsimp*

Which cannot be.

thus *False* **by** $(\mathit{simp}\ \mathit{add}{:}\mathit{phi\text{-}finite}\ \mathit{map\text{-}dom\text{-}ran\text{-}finite})$

qed

Any unnecessary ϕ with less than 2 true arguments induces with $\mathit{reachable}\ g\ \varphi$ a redundant set itself.

lemma *few-preds-redundant*:

assumes $\mathit{card}\ (\mathit{trueArgs}\ g\ \varphi) < 2\ \mathit{unnecessaryPhi}\ g\ \varphi$

shows $\mathit{redundant\text{-}set}\ g\ (\mathit{reachable}\ g\ \varphi)$

unfolding *redundant-set-def*

proof $(\mathit{intro}\ \mathit{conjI})$

from *assms*

show $\mathit{reachable}\ g\ \varphi \neq \{\}$

using $\mathit{empty\text{-}iff}\ \mathit{reachable.intros}(1)$ **by** *auto*

next

from $\mathit{assms}(2)$

```
  show reachable g φ ⊆ dom (phi g)
    by (metis domIff reachable.cases subsetI unnecessaryPhi-def)
next
  from assms(1)
  consider (single) card (trueArgs g φ) = 1 | (empty) card (trueArgs g φ) = 0 by
force
  thus ∃pred∈allVars g. ∀φ'∈reachable g φ. ∀φ''. phiArg g φ' φ'' ⟶ φ'' ∈ reachable
g φ ∪ {pred}
  proof cases
    case single
   then obtain pred where pred-prop: trueArgs g φ = {pred} using card-eq-1-singleton
by blast
    hence pred ∈ allVars g by (auto intro: Int-Collect phiArg-in-allVars)
    moreover
    from pred-prop
   have ∀φ'∈reachable g φ. ∀φ''. phiArg g φ' φ'' ⟶ φ'' ∈ reachable g φ ∪ {pred}
by auto
    ultimately
    show ?thesis by auto
  next
    case empty
    from allDefs-in-allVars[of - g defNode g φ] assms
   have phi-var: φ ∈ allVars g unfolding unnecessaryPhi-def phiDefs-def allDefs-def
defNode-def phi-def trueArgs-def
      by (clarsimp simp: domIff phis-in-αn)
    from empty assms(1)
     have no-preds: trueArgs g φ = {} by (subst card-0-eq[OF preds-finite, symmetric])
auto
    show ?thesis
    proof (rule bexI, rule ballI, rule allI, rule impI)
      fix φ' φ''
      assume phis-props: φ' ∈ reachable g φ phiArg g φ' φ''
      with no-preds
      have φ'' ∈ reachable g φ
      unfolding trueArgs-def
      proof −
        from phis-props
        have φ'' ∈ {φ'. ∃φ''∈reachable g φ. phiArg g φ'' φ'} by auto
        with phis-props no-preds
        show φ'' ∈ reachable g φ unfolding trueArgs-def by auto
      qed
      thus φ'' ∈ reachable g φ ∪ {φ} by simp
    qed (auto simp: phi-var)
  qed
qed


lemma phiArg-trancl-same-var:
assumes (phiArg g)⁺⁺ φ n
```

shows *var g* φ = *var g n*
using *assms*
apply (*induction rule*: *tranclp-induct*)
  apply (*rule phiArg-same-var*[*symmetric*])
  apply *simp*
 using *phiArg-same-var* **by** *auto*

The following path extension lemma will be used a number of times in the inner induction of the main proof. Basically, the idea is to extend a path ending in a ϕ argument to the corresponding ϕ function while preserving disjointness to a second path.

lemma *phiArg-disjoint-paths-extend*:
assumes *var g r* = *V* **and** *var g s* = *V* **and** *r* $\in$ *allVars g* **and** *s* $\in$ *allVars g*
and *V* $\in$ *oldDefs g n* **and** *V* $\in$ *oldDefs g m*
and *g* $\vdash$ *n*$-$*ns*$\rightarrow$*defNode g r* **and** *g* $\vdash$ *m*$-$*ms*$\rightarrow$*defNode g s*
and *set ns* $\cap$ *set ms* = {}
and *phiArg g* φ_r *r*
obtains *ns'*
where *g* $\vdash$ *n*$-$*ns*@*ns'*$\rightarrow$*defNode g* φ_r
and *set* (*butlast* (*ns*@*ns'*)) $\cap$ *set ms* = {}
proof (*cases r* = φ_r)
  case (*True*)

If the node to extend the path to is already the endpoint, the lemma is trivial.

  with *assms*(*7*,*8*,*9*) *in-set-butlastD*
  have *g* $\vdash$ *n*$-$*ns*@[]$\rightarrow$*defNode g* φ_r *set* (*butlast* (*ns*@[])) $\cap$ *set ms* = {}
    by *simp-all fastforce*
  with *that* **show** *?thesis* **.**
next
  case *False*

It suffices to obtain any path from r to φ_r. However, since we'll need the corresponding predecessor of φ_r later, we must do this as follows:

  from *assms*(*10*)
  have φ_r $\in$ *allVars g* **unfolding** *phiArg-def*
    by (*metis allDefs-in-allVars phiDefs-in-allDefs phi-def phi-phiDefs phis-in-*α*n*)
  with *assms*(*10*)
  obtain *rs' pred*$_{\varphi r}$ **where** *rs'-props*: *g* $\vdash$ *defNode g r*$-$*rs'*$\rightarrow$ *pred*$_{\varphi r}$ *old.EntryPath g rs' r* $\in$ *phiUses g pred*$_{\varphi r}$ *pred*$_{\varphi r}$ $\in$ *set* (*old.predecessors g* (*defNode g* φ_r))
   by (*rule phiArg-path-ex'*)

  define *rs* **where** *rs* = *rs'*@[*defNode g* φ_r]
  from *rs'-props*(*2*,*1*) *old.EntryPath-distinct old.path2-hd*
  have *rs'-loopfree*: *defNode g r* $\notin$ *set* (*tl rs'*) **by** (*simp add*: *Misc.distinct-hd-tl*)

  from *False assms* **have** *defNode g* φ_r $\neq$ *defNode g r*
   apply $-$
   apply (*rule phiArg-distinct-nodes*)
     apply (*auto intro*:*phiArg-in-allVars*)[*2*]

unfolding *phiArg-def* **by** (*metis allDefs-in-allVars phiDefs-in-allDefs phi-def phi-phiDefs phis-in-αn*)

from *rs′-props*
have *rs-props*: *g* ⊢ *defNode g r*—*rs*→ *defNode g* φ_r *length rs* > *1 defNode g r* ∉ *set* (*tl rs*)
apply (*subgoal-tac defNode g r* = *hd rs′*)
prefer *2* **using** *rs′-props*(*1*)
apply (*rule old.path2-hd*)
using *old.path2-snoc old.path2-def rs′-props*(*1*) *rs-def rs′-loopfree* ‹*defNode g* φ_r ≠ *defNode g r*› **by** *auto*

show *thesis*
proof (*cases set* (*butlast rs*) ∩ *set ms* = {})
case *inter-empty*: *True*

If the intersection of these is empty, *tl rs* is already the extension we're looking for

show *thesis*
proof (*rule that*)
show *set* (*butlast* (*ns* @ *tl rs*)) ∩ *set ms* = {}
proof (*rule ccontr*, *simp only*: *ex-in-conv*[*symmetric*])
assume ∃ *x*. *x* ∈ *set* (*butlast* (*ns* @ *tl rs*)) ∩ *set ms*
then obtain *x* **where** *x-props*: *x* ∈ *set* (*butlast* (*ns* @ *tl rs*)) *x* ∈ *set ms* **by** *auto*
with *rs-props*(*2*)
consider (*in-ns*) *x* ∈ *set ns* | (*in-rs*) *x* ∈ *set* (*butlast* (*tl rs*)) **by** (*metis Un-iff butlast-append in-set-butlastD set-append*)
thus *False*
apply (*cases*)
using *x-props*(*2*) *assms*(*9*)
apply (*simp add*: *disjoint-elem*)
by (*metis x-props*(*2*) *inter-empty in-set-tlD List.butlast-tl disjoint-iff-not-equal*)
qed
qed (*auto intro*:*assms*(*7*) *rs-props*(*1*) *old.path2-app*)
next
case *inter-ex*: *False*

If the intersection is nonempty, there must be a first point of intersection *i*.

from *inter-ex assms*(*7*,*8*) *rs-props*
obtain *i ri* **where** *ri-props*: *g* ⊢ *defNode g r*—*ri*→*i i* ∈ *set ms* ∀ *n* ∈ *set* (*butlast ri*). *n* ∉ *set ms prefix ri rs*
apply −
apply (*rule old.path2-split-first-prop*[*of g defNode g r rs defNode g* φ_r, **where** *P*=λ*m*. *m* ∈ *set ms*])
apply *blast*
apply (*metis disjoint-iff-not-equal in-set-butlastD*)
by *blast*
with *assms*(*8*) *old.path2-prefix-ex*

obtain *ms'* **where** *ms'-props*: $g \vdash m\ {-}ms'{\rightarrow}\ i$ *prefix ms' ms* $i \notin$ *set* (*butlast ms'*) **by** *blast*

We proceed by case distinction:

- if $i = \mathit{defNode}\ g\ \varphi_r$, the path *ri* is already the path extension we're looking for
- Otherwise, the fact that i is on the path from ϕ argument to the ϕ itself leads to a contradiction. However, we still need to distinguish the cases of whether $m = i$

consider (*ri-is-valid*) $i = \mathit{defNode}\ g\ \varphi_r$ | (*m-i-same*) $i \neq \mathit{defNode}\ g\ \varphi_r\ \ m = i$ | (*m-i-differ*) $i \neq \mathit{defNode}\ g\ \varphi_r\ \ m \neq i$ **by** *auto*

thus *thesis*
proof (*cases*)
  case *ri-is-valid*

ri is a valid path extension.

  with *assms*(*7*) *ri-props*(*1*)
  have $g \vdash n\ {-}ns@(tl\ ri){\rightarrow}\ \mathit{defNode}\ g\ \varphi_r$ **by** *auto*

  moreover
  have *set* (*butlast* (*ns*@(*tl ri*))) $\cap$ *set ms* = {}
  proof (*rule ccontr*)
    assume *contr*: *set* (*butlast* (*ns* @ *tl ri*)) $\cap$ *set ms* $\neq$ {}
    from *this*
    obtain *x* **where** *x-props*: $x \in$ *set* (*butlast* (*ns* @ *tl ri*)) $x \in$ *set ms* **by** *auto*
    with *assms*(*9*) **have** $x \notin$ *set ns* **by** *auto*
    with *x-props* ‹$g \vdash n{-}ns\ @\ tl\ ri{\rightarrow}\mathit{defNode}\ g\ \varphi_r$› ‹$\mathit{defNode}\ g\ \varphi_r \neq \mathit{defNode}\ g\ r$› *assms*(*7*)
    have $x \in$ *set* (*butlast* (*tl ri*))
     by (*metis Un-iff append-Nil2 butlast-append old.path2-last set-append*)
    with *x-props*(*2*) *ri-props*(*3*)
    show *False* **by** (*metis FormalSSA-Misc.in-set-tlD List.butlast-tl*)
  qed
  ultimately
  show *thesis* **by** (*rule that*)
next
  case *m-i-same*

If $m = i$, we have, with m, a variable definition on the path from a ϕ function to its argument. This constitutes a contradiction to the conventional property.

  note *rs'-props*(*1*) *rs'-loopfree*
  moreover have $r \in$ *allDefs g* (*defNode g r*) **by** (*simp add*: *assms*(*3*))
  moreover from *rs'-props*(*3*) **have** $r \in$ *allUses g* $\mathit{pred}_{\varphi r}$ **unfolding** *allUses-def* **by** *simp*

  moreover

from *rs-props*(*1*) *m-i-same rs-def ri-props*(*1*,*2*,*4*) ‹*defNode g* $\varphi_r \neq$ *defNode g r*› *assms*(*7*,*9*)
have $m \in$ *set* (*tl* rs')
by (*metis disjoint-elem hd-append in-hd-or-tl-conv in-prefix list.sel*(*1*) *old.path2-hd old.path2-last old.path2-last-in-ns prefix-snoc*)

moreover
from *assms*(*6*) **obtain** def_m **where** $def_m \in$ *allDefs g m var g* $def_m = V$ **unfolding** *oldDefs-def* **using** *defs-in-allDefs* **by** *blast*

ultimately
have *var g* $def_m \neq$ *var g r* **by** $-$ (*rule conventional*, *simp-all*)
with ‹*var g* $def_m = V$› *assms*(*1*)
have *False* **by** *simp*
thus *?thesis* **by** *simp*

next
case *m-i-differ*

If $m \neq i$, i constitutes a proper path convergence point.

have *old.pathsConverge g m* ms' *n* (*ns* @ *tl ri*) *i*
proof (*rule old.pathsConvergeI*)
show $1 <$ *length* ms' **using** *m-i-differ* ms'*-props old.path2-nontriv* **by** *blast*
next
show $1 <$ *length* (*ns* @ *tl ri*)
using *ri-props old.path2-nontriv assms*(*9*) **by** (*metis assms*(*7*) *disjoint-elem old.path2-app old.path2-hd-in-ns*)
next
show *set* (*butlast* ms') $\cap$ *set* (*butlast* (*ns* @ *tl ri*)) $= \{\}$
proof (*rule ccontr*)
assume *set* (*butlast* ms') $\cap$ *set* (*butlast* (*ns* @ *tl ri*)) $\neq \{\}$
then obtain i' **where** i'*-props*: $i' \in$ *set* (*butlast* ms') $i' \in$ *set* (*butlast* (*ns* @ *tl ri*)) **by** *auto*
with ms'*-props*(*2*)
have i'*-not-in-ms*: $i' \in$ *set* (*butlast ms*) **by** (*metis in-set-butlast-appendI prefixE*)

with *assms*(*9*)
show *False*
proof (*cases* $i' \notin$ *set ns*)
case *True*
with i'*-props*(*2*)
have $i' \in$ *set* (*butlast* (*tl ri*))
by (*metis Un-iff butlast-append in-set-butlastD set-append*)
hence $i' \in$ *set* (*butlast ri*) **by** (*simp add*:*in-set-tlD List.butlast-tl*)
with i'*-not-in-ms ri-props*(*3*)
show *False* **by** (*auto dest*:*in-set-butlastD*)
qed (*meson disjoint-elem in-set-butlastD*)
qed
qed (*auto intro*: *assms*(*7*) *ri-props*(*1*) *old.path2-app* ms'*-props*(*1*))

At this intersection of paths we can find a ϕ function.

```
from this assms(6,5)
have necessaryPhi g V i by (rule necessaryPhiI)
```

Before we can conclude that there is indeed a ϕ at i, we have to prove a couple of technicalities...

```
moreover
from m-i-differ ri-props(1,4) rs-def old.path2-last prefix-snoc
have ri-rs'-prefix: prefix ri rs' by fastforce
then obtain rs'-rest where rs'-rest-prop: rs' = ri@rs'-rest using prefixE by auto
from old.path2-last[OF ri-props(1)] last-snoc[of - i] obtain tmp where ri = tmp@[i]
  apply (subgoal-tac ri ≠ [])
   prefer 2
   using ri-props(1) apply (simp add: old.path2-not-Nil)
  apply (rule-tac that)
  using append-butlast-last-id[symmetric] by auto
with rs'-rest-prop have rs'-rest-def: rs' = tmp@i#rs'-rest by auto
with rs'-props(1) have g ⊢ i −i#rs'-rest→ pred_φr
 by (simp add:old.path2-split)
moreover
note ‹var g r = V› [simp]
from rs'-props(3)
have r ∈ allUses g pred_φr unfolding allUses-def by simp

moreover
from ‹defNode g r ∉ set (tl rs')› rs'-rest-def
have defNode g r ∉ set rs'-rest by auto
with ‹g ⊢ i −i#rs'-rest→ pred_φr›
have ⋀x. x ∈ set rs'-rest ⟹ r ∉ allDefs g x
 by (metis defNode-eq list.distinct(1) list.sel(3) list.set-cases old.path2-cases old.path2-in-αn)

moreover
from assms(7,9) ‹g ⊢ i −i#rs'-rest→ pred_φr› ri-props(2)
have r ∉ defs g i
by (metis defNode-eq defs-in-allDefs disjoint-elem old.path2-hd-in-αn old.path2-last-in-ns)
ultimately
```

The convergence property gives us that there is a ϕ in the last node fulfilling *necessaryPhi* on a path to a use of r without a definition of r. Thus i bears a ϕ function for the value of r.

```
have ∃ y. phis g (i, r) = Some y
by (rule convergence-prop [where g=g and n=i and v=r and ns=i#rs'-rest, simplified])
moreover

from ‹g ⊢ n−ns→defNode g r› have defNode g r ∈ set ns by auto
```

```
   with ‹set ns ∩ set ms = {}› ‹i ∈ set ms› have i ≠ defNode g r by auto
   moreover

   from ms'-props(1) have i ∈ set (αn g) by auto
   moreover

   have defNode g r ∈ set (αn g) by (simp add: assms(3))
```

However, we now have two definitions of *r*: one in *i*, and one in *defNode g r*, which we know to be distinct. This is a contradiction to the *allDefs-disjoint*-property.

```
   ultimately have False
    using allDefs-disjoint [where g=g and n=i and m=defNode g r]
    unfolding allDefs-def phiDefs-def
    apply clarsimp
    apply (erule-tac c=r in equalityCE)
    using phi-def phis-phi by auto
   thus ?thesis by simp
  qed
 qed
qed


lemma reachable-same-var:
assumes φ' ∈ reachable g φ
shows var g φ = var g φ'
using assms by (metis Nitpick.rtranclp-unfold phiArg-trancl-same-var reachable-props(1))


lemma φ-node-no-defs:
assumes unnecessaryPhi g φ φ ∈ allVars g var g φ ∈ oldDefs g n
shows defNode g φ ≠ n
using assms simpleDefs-phiDefs-var-disjoint defNode(1) not-None-eq phi-phiDefs
 unfolding unnecessaryPhi-def by auto


lemma defNode-differ-aux:
assumes φs ∈ reachable g φ φ ∈ allVars g s ∈ allVars g φs ≠ s var g φ = var g s
shows defNode g φs ≠ defNode g s unfolding reachable-def
proof (rule ccontr)
 assume ¬ defNode g φs ≠ defNode g s
 hence eq: defNode g φs = defNode g s by simp
 from assms(1)
 have vars-eq: var g φ = var g φs
  apply –
  apply (cases φ = φs)
  apply simp
  apply (rule phiArg-trancl-same-var)
  apply (drule reachable-props)
  unfolding reachable-def by (meson IntD1 mem-Collect-eq rtranclpD)
  have φ_s-in-allVars: φ_s ∈ allVars g unfolding reachable-def
  proof (cases φ = φ_s)
    case False
    with assms(1)
    obtain φ' where phiArg g φ' φ_s by (metis rtranclp.cases reachable-props(1))
    thus φ_s ∈ allVars g by (rule phiArg-in-allVars)
  next
    case eq: True
    with assms(2)
    show φ_s ∈ allVars g by (subst eq[symmetric])
  qed

  from eq φ_s-in-allVars assms(3,4)
  have var g φ_s ≠ var g s by − (rule defNode-var-disjoint)
  with vars-eq assms(5)
  show False by auto
qed
```

Theorem 1. A graph which does not contain any redundant set is minimal according to Cytron et al.'s definition of minimality.

```
theorem no-redundant-set-minimal:
assumes no-redundant-set: ¬(∃ P. redundant-set g P)
shows cytronMinimal g
proof (rule ccontr)
  assume ¬cytronMinimal g
```

Assume the graph is not Cytron-minimal. Thus there is a ϕ function which does not sit at the convergence point of multiple liveness intervals.

```
  then obtain φ where φ-props: unnecessaryPhi g φ φ ∈ allVars g φ ∈ reachable g φ
    using cytronMinimal-def unnecessaryPhi-def reachable-def unnecessaryPhi-def reachable.intros by auto
```

We consider the reachable-set of φ. If φ has less than two true arguments, we know it to be a redundant set, a contradiction. Otherwise, we know there to be at least two paths from different definitions leading into the reachable-set of φ.

```
  consider (nontrivial) card (trueArgs g φ) ≥ 2 | (trivial) card (trueArgs g φ) < 2 using linorder-not-le by auto
  thus False
  proof cases
    case trivial
```

If there are less than 2 true arguments of this set, the set is trivially redundant (see *few-preds-redundant*).

```
    from this φ-props(1)
    have redundant-set g (reachable g φ) by (rule few-preds-redundant)
    with no-redundant-set
    show False by simp
  next
    case nontrivial
```

If there are two or more necessary arguments, there must be disjoint paths from Defs to two of these ϕ functions.

```
then obtain r s φr φs where assign-nodes-props:
  r ≠ s φr ∈ reachable g φ φs ∈ reachable g φ
  ¬ unnecessaryPhi g r ¬ unnecessaryPhi g s
  r ∈ {n. (phiArg g)** φ n} s ∈ {n. (phiArg g)** φ n}
  phiArg g φr r phiArg g φs s
 apply simp
 apply (rule set-take-two[OF nontrivial])
 apply simp
by (meson reachable.intros(2) reachable-props(1) rtranclp-tranclp-tranclp tranclp.r-into-trancl tranclp-into-rtranclp)
moreover from assign-nodes-props
have φ-r-s-uneq: φ ≠ r φ ≠ s using φ-props by auto
moreover
from assign-nodes-props this
have r-s-in-tranclp: (phiArg g)++ φ r (phiArg g)++ φ s
by (meson mem-Collect-eq rtranclpD) (meson assign-nodes-props(7) φ-r-s-uneq(2) mem-Collect-eq rtranclpD)
from this
obtain V where V-props: var g r = V var g s = V var g φ = V by (metis phiArg-trancl-same-var)
moreover
from r-s-in-tranclp
 have r-s-allVars: r ∈ allVars g s ∈ allVars g by (metis phiArg-in-allVars tranclp.cases)+
moreover
from V-props defNode-var-disjoint r-s-allVars assign-nodes-props(1)
have r-s-defNode-distinct: defNode g r ≠ defNode g s by auto
ultimately
obtain n ns m ms where r-s-path-props: V ∈ oldDefs g n g ⊢ n−ns→defNode g r V ∈ oldDefs g m g ⊢ m−ms→defNode g s
   set ns ∩ set ms = {} by (auto intro: ununnecessaryPhis-disjoint-paths[of g r s])

have n-m-distinct: n ≠ m
proof (rule ccontr)
  assume n-m: ¬ n ≠ m
  with r-s-path-props(2) old.path2-hd-in-ns
  have n ∈ set ns by blast
  moreover
  from n-m r-s-path-props(4) old.path2-hd-in-ns
  have n ∈ set ms by blast
  ultimately
  show False using r-s-path-props(5) by auto
qed
```

These paths can be extended into paths reaching ϕ functions in our set.

```
from V-props r-s-allVars r-s-path-props assign-nodes-props
```

```
  obtain rs where rs-props: g ⊢ n −ns@rs→ defNode g φ_r set (butlast (ns@rs))
∩ set ms = {}
  using phiArg-disjoint-paths-extend by blast
```

(In fact, we can prove that *set* (*ns* @ *rs*) ∩ *set ms* = {}, which we need for the next path extension.)

```
  have defNode g φ_r ∉ set ms
  proof (rule ccontr)
    assume φ_r-in-ms: ¬ defNode g φ_r ∉ set ms
    from this r-s-path-props(4)
    obtain ms' where ms'-props: g ⊢ m −ms'→ defNode g φ_r prefix ms' ms by
−(rule old.path2-prefix-ex[of g m ms defNode g s defNode g φ_r], auto)

    have old.pathsConverge g n (ns@rs) m ms' (defNode g φ_r)
    proof (rule old.pathsConvergeI)
      show set (butlast (ns @ rs)) ∩ set (butlast ms') = {}
      proof (rule ccontr)
        assume set (butlast (ns @ rs)) ∩ set (butlast ms') ≠ {}
        then obtain c where c-props: c ∈ set (butlast (ns@rs)) c ∈ set (butlast
ms') by auto
        from this(2) ms'-props(2)
        have c ∈ set ms by (simp add: in-prefix in-set-butlastD)
        with c-props(1) rs-props(2)
        show False by auto
      qed
    next
      have m-n-φ_r-differ: n ≠ defNode g φ_r m ≠ defNode g φ_r
        using assign-nodes-props(2,3,4,5) V-props r-s-path-props φ_r-in-ms
        apply fastforce
      using V-props(1) φ_r-in-ms assign-nodes-props(8) old.path2-in-αn phiArg-def
phiArg-same-var r-s-path-props(3,4) simpleDefs-phiDefs-var-disjoint
        by auto
      with ms'-props(1)
      show 1 < length ms' using old.path2-nontriv by simp
      from m-n-φ_r-differ rs-props(1)
      show 1 < length (ns@rs) using old.path2-nontriv by blast
    qed (auto intro: rs-props set-mono-prefix ms'-props)
    with V-props r-s-path-props
   have necessaryPhi' g φ_r unfolding necessaryPhi-def using assign-nodes-props(8)
phiArg-same-var by auto
    with reachable-props(2)[OF assign-nodes-props(2)]
    show False unfolding unnecessaryPhi-def by simp
  qed

  with rs-props
  have aux: set ms ∩ set (ns @ rs) = {}
    by (metis disjoint-iff-not-equal not-in-butlast old.path2-last)

  have φ_r-V: var g φ_r = V
    using V-props(1) assign-nodes-props(8) phiArg-same-var by auto
```

have φ_r*-allVars*: $\varphi_r \in \mathit{allVars}\ g$
by (*meson phiArg-def assign-nodes-props*(*8*) *allDefs-in-allVars old.path2-tl-in-*α*n phiDefs-in-allDefs phi-phiDefs rs-props*)

from *V-props*(*2*) φ_r*-V r-s-allVars*(*2*) φ_r*-allVars r-s-path-props*(*3*) *r-s-path-props*(*1*) *r-s-path-props*(*4*) *rs-props*(*1*) *aux assign-nodes-props*(*9*)
obtain *ss* **where** *ss-props*: $g \vdash m\ {-}ms@ss{\rightarrow}\ \mathit{defNode}\ g\ \varphi_s$ $\mathit{set}\ (\mathit{butlast}\ (ms@ss)) \cap \mathit{set}\ (\mathit{butlast}\ (ns@rs)) = \{\}$
by (*rule phiArg-disjoint-paths-extend*) (*metis disjoint-iff-not-equal in-set-butlastD*)

define p_m **where** $p_m = ms@ss$
define p_n **where** $p_n = ns@rs$

have *ind-props*: $g \vdash m\ {-}p_m{\rightarrow}\ \mathit{defNode}\ g\ \varphi_s$ $g \vdash n\ {-}p_n{\rightarrow}\ \mathit{defNode}\ g\ \varphi_r$ $\mathit{set}\ (\mathit{butlast}\ p_m) \cap \mathit{set}\ (\mathit{butlast}\ p_n) = \{\}$
using *rs-props*(*1*) *ss-props* p_m*-def* p_n*-def* **by** *auto*

The following case will occur twice in the induction, with swapped identifiers, so we're proving it outside. Basically, if the paths p_m and p_n intersect, the first such intersection point must be a ϕ function in *reachable* $g\ \varphi$, yielding the path convergence we seek.

have *path-crossing-yields-convergence*:
$\exists\,\varphi_z \in \mathit{reachable}\ g\ \varphi.\ \exists\, ns\ ms.\ \mathit{old.pathsConverge}\ g\ n\ ns\ m\ ms\ (\mathit{defNode}\ g\ \varphi_z)$
if $\varphi_r \in \mathit{reachable}\ g\ \varphi$ **and** $\varphi_s \in \mathit{reachable}\ g\ \varphi$ **and** $g \vdash n\ {-}p_n{\rightarrow}\ \mathit{defNode}\ g\ \varphi_r$
and $g \vdash m\ {-}p_m{\rightarrow}\ \mathit{defNode}\ g\ \varphi_s$ **and** $\mathit{set}\ (\mathit{butlast}\ p_m) \cap \mathit{set}\ (\mathit{butlast}\ p_n) = \{\}$
and $\mathit{set}\ p_m \cap \mathit{set}\ p_n \neq \{\}$
for $\varphi_r\ \varphi_s\ p_m\ p_n$
proof –
from *that*(*6*) *split-list-first-propE*
obtain $p_m 1\ n_z\ p_m 2$ **where** n_z*-props*: $n_z \in \mathit{set}\ p_n$ $p_m = p_m 1\ @\ n_z\ \#\ p_m 2$ $\forall\, n \in \mathit{set}\ p_m 1.\ n \notin \mathit{set}\ p_n$
by (*auto intro*: *split-list-first-propE*)

with *that*(*3*,*4*)
obtain $p_n{}'$ **where** $p_n{}'$*-props*: $g \vdash n{-}p_n{}'{\rightarrow}n_z$ $g \vdash m{-}p_m 1@[n_z]{\rightarrow}n_z$ $\mathit{prefix}\ p_n{}'\ p_n$ $n_z \notin \mathit{set}\ (\mathit{butlast}\ p_n{}')$
by (*meson old.path2-prefix-ex old.path2-split*(*1*))

from *V-props*(*3*) *reachable-same-var*[*OF that*(*1*)] *reachable-same-var*[*OF that*(*2*)]
have *phis-V*: $\mathit{var}\ g\ \varphi_r = V$ $\mathit{var}\ g\ \varphi_s = V$ **by** *simp-all*
from *reachable-props*(*1*) *that*(*1*,*2*) φ*-props*(*2*) *phiArg-in-allVars*
have *phis-allVars*: $\varphi_r \in \mathit{allVars}\ g$ $\varphi_s \in \mathit{allVars}\ g$ **by** (*metis rtranclp.cases*)+

Various inequalities for proving paths aren't trivial.

have $n \neq \mathit{defNode}\ g\ \varphi_r$ $m \neq \mathit{defNode}\ g\ \varphi_r$
using φ*-node-no-defs phis-V*(*1*) *phis-allVars*(*1*) *r-s-path-props*(*1*,*3*) *reachable-props*(*2*) *that*(*1*) **by** *blast*+

from *φ-node-no-defs reachable-props(2) that(2) r-s-path-props(1,3) phis-V(2) that phis-allVars*

have $m \neq$ *defNode g* φ_s $n \neq$ *defNode g* φ_s **by** *blast+*

With this scenario, since *set* (*butlast* p_n) $\cap$ *set* (*butlast* p_m) $= \{\}$, one of the paths p_n and p_m must end somewhere within the other, however this means the ϕ function in that node must either be φ or φ_r

from *assms* n_z*-props*

consider (p_n*-ends-in-*p_m) $n_z =$ *defNode g* φ_s | (p_m*-ends-in-*p_n) $n_z =$ *defNode g* φ_r

proof (*cases* $n_z =$ *last* p_n)

case *True*

with ‹$g \vdash n -p_n\rightarrow$ *defNode g* φ_r›

have $n_z =$ *defNode g* φ_r **using** *old.path2-last* **by** *auto*

with *that*(*2*) **show** *?thesis*.

next

case *False*

from n_z*-props*(*2*)

have $n_z \in$ *set* p_m **by** *simp*

with *False* n_z*-props*(*1*) ‹*set* (*butlast* p_m) $\cap$ *set* (*butlast* p_n) $= \{\}$› ‹$g \vdash m -p_m\rightarrow$ *defNode g* φ_s›

have $n_z =$ *defNode g* φ_s **by** (*metis disjoint-elem not-in-butlast old.path2-last*)

with *that*(*1*) **show** *?thesis*.

qed

thus $\exists\,\varphi_z \in$ *reachable g* φ. $\exists$ *ns ms. old.pathsConverge g n ns m ms* (*defNode g* φ_z)

proof (*cases*)

case p_n*-ends-in-*p_m

have *old.pathsConverge g n* $p_n{}'$ *m* p_m (*defNode g* φ_s)

proof (*rule old.pathsConvergeI*)

from p_n*-ends-in-*p_m $p_n{}'$*-props*(*1*) **show** $g \vdash n{-}p_n{}'{\rightarrow}$*defNode g* φ_s **by** *simp*

from ‹$n \neq$ *defNode g* φ_s› p_n*-ends-in-*p_m $p_n{}'$*-props*(*1*) *old.path2-nontriv* **show** $1 <$ *length* $p_n{}'$ **by** *auto*

from *that*(*4*) **show** $g \vdash m -p_m\rightarrow$ *defNode g* φ_s.

with ‹$m \neq$ *defNode g* φ_s› *old.path2-nontriv* **show** $1 <$ *length* p_m **by** *simp*

from *that* $p_n{}'$*-props*(*3*) **show** *set* (*butlast* $p_n{}'$) $\cap$ *set* (*butlast* p_m) $= \{\}$

by (*meson butlast-prefix disjointI disjoint-elem in-prefix*)

qed

with *that*(*1*,*2*,*3*) **show** *?thesis* **by** (*auto intro:reachable.intros*(*2*))

next

case p_m*-ends-in-*p_n

have *old.pathsConverge g n* $p_n{}'$ *m* (p_m1@[n_z]) (*defNode g* φ_r)

proof (*rule old.pathsConvergeI*)

from p_m*-ends-in-*p_n $p_n{}'$*-props*(*1*,*2*) **show** $g \vdash n{-}p_n{}'{\rightarrow}$*defNode g* φ_r $g \vdash m{-}p_m1$ @ [n_z]$\rightarrow$*defNode g* φ_r **by** *simp-all*

with ‹$n \neq$ *defNode g* φ_r› ‹$m \neq$ *defNode g* φ_r› **show** $1 <$ *length* $p_n{}'$ $1 <$ *length* (p_m1 @ [n_z])

using *old.path2-nontriv*[*of g m* p_m1 @ [n_z]] *old.path2-nontriv*[*of g n*] **by**

```
simp-all
          from n_z-props p_n'-props(3) show set (butlast p_n') ∩ set (butlast (p_m1 @ [n_z])) = {}
            using butlast-snoc disjointI in-prefix in-set-butlastD by fastforce
        qed
        with that(1) show ?thesis by (auto intro:reachable.intros)
      qed
    qed
```

Since the reachable-set was built starting at a single ϕ, these paths must at some point converge *within reachable g* φ.

```
    from assign-nodes-props(3,2) ind-props V-props(3) φ_r-V φ_r-allVars
    have ∃φ_z ∈ reachable g φ. ∃ns ms. old.pathsConverge g n ns m ms (defNode g φ_z)
    proof (induction arbitrary: p_m p_n rule: reachable.induct)
      case refl
```

In the induction basis, we know that $\varphi = \varphi_s$, and a path to φ_r must be obtained – for this we need a second induction.

```
      from refl.prems refl.hyps show ?case
      proof (induction arbitrary: p_m p_n rule: reachable.induct)
        case refl
```

The first case, in which $\varphi_r = \varphi_s = \varphi$, is trivial – φ suffices.

```
        have old.pathsConverge g n p_n m p_m (defNode g φ)
        proof (rule old.pathsConvergeI)
          show 1 < length p_n 1 < length p_m
             using refl V-props simpleDefs-phiDefs-var-disjoint unfolding unnecessaryPhi-def
            by (metis domD domIff old.path2-hd-in-αn old.path2-nontriv phi-phiDefs r-s-path-props(1) r-s-path-props(3))+
           show g ⊢ n−p_n→defNode g φ g ⊢ m−p_m→defNode g φ set (butlast p_n) ∩ set (butlast p_m) = {}
          using refl by auto
        qed
        with ‹φ ∈ reachable g φ› show ?case by auto
      next
        case (step φ' φ_r)
```

In this case we have that $\varphi = \varphi_s$ and need to acquire a path going to φ_r, however with the aux. lemma we have, we still need that p_n and p_m are disjoint.

```
        thus ?case
        proof (cases set p_n ∩ set p_m = {})
          case paths-cross: False
          with step reachable.intros
           show ?thesis using path-crossing-yields-convergence[of φ_r φ p_n p_m] by (metis disjointI disjoint-elem)
        next
          case True
```

If the paths are intersection-free, we can apply our path extension lemma to obtain the path needed.

```
          from step(9,8,10) ‹φ ∈ allVars g› r-s-path-props(1,3) step(6,5) True step(2)
        obtain ns where g ⊢ n −p_n@ns→ defNode g φ' set (butlast (p_n@ns)) ∩ set p_m = {} by (rule phiArg-disjoint-paths-extend)

        from this(2) have set (butlast p_m) ∩ set (butlast (p_n @ ns)) = {}
          using in-set-butlastD by fastforce
        moreover
        from phiArg-same-var step.hyps(2) step.prems(5) have var g φ' = V
          by auto
        moreover
        have φ' ∈ allVars g
          by (metis φ-props(2) phiArg-in-allVars reachable.cases step.hyps(1))
        ultimately
       show ∃ φ_z∈reachable g φ. ∃ ns ms. old.pathsConverge g n ns m ms (defNode g φ_z)
          using step.prems(1) φ-props V-props ‹g ⊢ n −p_n@ns→ defNode g φ'›
          by −(rule step.IH; blast)
      qed
    qed
  next
    case (step φ' φ_s)
```

With the induction basis handled, we can finally move on to the induction proper.

```
    show ?thesis
    proof (cases set p_m ∩ set p_n = {})
      case True
     have φ_s-V: var g φ_s = V using step(1,2,3,9) reachable-same-var by (simp add: phiArg-same-var)
      from step(2) have φ_s-allVars: φ_s ∈ allVars g by (rule phiArg-in-allVars)

        obtain p_m' where tmp: g ⊢ m −p_m@p_m'→ defNode g φ' set (butlast (p_m@p_m')) ∩ set (butlast p_n) = {}
       by (rule phiArg-disjoint-paths-extend[of g φ_s V φ_r m n p_m p_n φ'])
      (metis φ_s-V φ_s-allVars step r-s-path-props(1,3) True disjoint-iff-not-equal in-set-butlastD)+

      from step(5) this(1) step(7) this(2) step(9) step(10) step(11)
      show ?thesis by (rule step.IH[of p_m@p_m' p_n])
    next
      case paths-cross: False
      with step reachable.intros
       show ?thesis using path-crossing-yields-convergence[of φ_r φ_s p_n p_m] by blast
    qed
  qed
```

then obtain φ_z *ns ms* **where** $\varphi_z \in$ *reachable g* φ **and** *old.pathsConverge g n ns m ms* (*defNode g* φ_z)
by *blast*
moreover
with *reachable-props* **have** *var g* $\varphi_z = V$ **by** (*metis V-props*(*3*) *phiArg-trancl-same-var rtranclpD*)
ultimately have *necessaryPhi′ g* φ_z **using** *r-s-path-props*
unfolding *necessaryPhi-def* **by** *blast*
moreover with ‹$\varphi_z \in$ *reachable g* φ› **have** *unnecessaryPhi g* φ_z **by** −(*rule reachable-props*)
ultimately show *False* **unfolding** *unnecessaryPhi-def* **by** *blast*
qed
qed

Together with lemma 1, we thus have that a CFG without redundant SCCs is cytron-minimal, proving that the property established by Braun et al.'s algorithm suffices.

corollary *no-redundant-SCC-minimal*:
assumes ¬(∃ *P scc. redundant-scc g P scc*)
shows *cytronMinimal g*
using *assms 1 no-redundant-set-minimal* **by** *blast*

Finally, to conclude, we'll show that the above theorem is indeed a stronger assertion about a graph than the lack of trivial ϕ functions. Intuitively, this is because a set containing only a trivial ϕ function is a redundant set.

corollary
assumes ¬(∃ *P. redundant-set g P*)
shows ¬*redundant g*
proof −
have *redundant g* ⟹ ∃ *P. redundant-set g P*
proof −
assume *redundant g*
then obtain φ **where** *phi g* $\varphi \neq$ *None trivial g* φ
unfolding *redundant-def redundant-set-def dom-def phiArg-def trivial-def isTrivialPhi-def*
by (*clarsimp split*: *option.splits*) *fastforce*
hence *redundant-set g* $\{\varphi\}$
unfolding *redundant-set-def dom-def phiArg-def trivial-def isTrivialPhi-def*
by *auto*
thus *?thesis* **by** *auto*
qed
with *assms* **show** *?thesis* **by** *auto*
qed

end

end

References

[1] 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.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[3] S. Ullrich and D. Lohner. Verified construction of static single assignment form. *Archive of Formal Proofs*, Feb. 2016. http://isa-afp.org/entries/Formal_SSA.shtml, Formal proof development.