

Graph Theory

By Lars Noschinski

February 6, 2026

Abstract

This development provides a formalization of planarity based on combinatorial maps and proves that Kuratowski's theorem implies combinatorial planarity. Moreover, it contains verified implementations of programs checking certificates for planarity (i.e., a combinatorial map) or non-planarity (i.e., a Kuratowski subgraph).

The development is described in [1].

Contents

1	Combinatorial Maps	3
2	Maps and Isomorphism	10
3	Auxiliary List Lemmas	14
4	Permutations as Products of Disjoint Cycles	15
4.1	Cyclic Permutations	15
4.2	Arbitrary Permutations	18
5	List Orbits	23
5.1	Relation to <i>cyclic-on</i>	26
5.2	Permutations of a List	32
5.3	Enumerating Permutations from List Orbits	34
5.4	Lists of Permutations	35
6	Enumerating Maps	38
7	Compute Face Cycles	41
8	Kuratowski Graphs are not Combinatorially Planar	47
8.1	A concrete K5 graph	47
8.2	A concrete K33 graph	47
8.3	Generalization to arbitrary Kuratowski Graphs	48
8.3.1	Number of Face Cycles is a Graph Invariant	48

8.3.2	Combinatorial planarity is a Graph Invariant	49
8.3.3	Completeness is a Graph Invariant	50
8.3.4	Conclusion	52
9	<i>n</i>-step reachability	57
10	More	59
11	Modifying Permutations	59
12	Cyclic Permutations	61
13	Combinatorial Planarity and Subdivisions	62
14	Combinatorial Planarity and Subgraphs	73
14.1	Deleting an isolated vertex	76
14.2	Deleting an arc pair	79
14.3	Modifying <i>edge-rev</i>	107
14.4	Conclusion	109
15	Implementation of a Non-Planarity Checker	114
15.1	An abstract graph datatype	114
15.2	Code	115
16	Verification of a Non-Planarity Checker	121
16.1	Graph Basics and Implementation	121
16.2	Total Correctness	129
16.2.1	Procedure <i>is-subgraph</i>	129
16.2.2	Procedure <i>is-loop-free</i>	132
16.2.3	Procedure <i>select-nodes</i>	133
16.2.4	Procedure <i>find-endpoint</i>	134
16.2.5	Procedure <i>contract</i>	142
16.2.6	Procedure <i>is-K33</i>	148
16.2.7	Procedure <i>is-K5</i>	154
16.2.8	Soundness of the Checker	157
17	Auxilliary Lemmas for Autocorres	158
17.1	Option monad	158
18	AutoCorres setup for VCG labelling	159
18.1	Labeled VCG theorems for branching	160
18.2	Labelled VCG theorems for the option monad	161

19 Verification of a Planarity Checker	162
19.1 Implementation Types	163
19.2 Implementation	164
19.3 Verification	167
19.3.1 <i>is-map</i>	167
19.3.2 <i>isolated-nodes</i>	180
19.3.3 <i>face-cycles</i>	183

theory *Graph-Genus*

imports

HOL-Combinatorics.Permutations

Graph-Theory.Graph-Theory

begin

lemma *nat-diff-mod-right*:

fixes $a\ b\ c :: \text{nat}$

assumes $b < a$

shows $(a - b) \text{ mod } c = (a - b \text{ mod } c) \text{ mod } c$

proof –

from *assms* **have** *b-mod*: $b \text{ mod } c \leq a$

by (*metis mod-less-eq-dividend linear not-le order-trans*)

have $\text{int } ((a - b) \text{ mod } c) = (\text{int } a - \text{int } b \text{ mod } \text{int } c) \text{ mod } \text{int } c$

using *assms* **by** (*simp add: zmod-int of-nat-diff mod-simps*)

also **have** $\dots = \text{int } ((a - b \text{ mod } c) \text{ mod } c)$

using *assms b-mod* **by** (*simp add: zmod-int*)

finally **show** *?thesis* **by** *simp*

qed

lemma *inj-on-f-imageI*:

assumes $\text{inj-on } f\ S \ \bigwedge t. t \in T \implies t \subseteq S$

shows $\text{inj-on } ((\cdot) f)\ T$

using *assms* **by** (*auto simp: inj-on-image-eq-iff intro: inj-onI*)

1 Combinatorial Maps

lemma (*in bidirected-digraph*) *has-dom-arev*:

has-dom arev (*arcs G*)

using *arev-dom* **by** (*auto simp: has-dom-def*)

record *'b pre-map* =

edge-rev :: *'b* \Rightarrow *'b*

edge-succ :: *'b* \Rightarrow *'b*

definition *edge-pred* :: *'b pre-map* \Rightarrow *'b* \Rightarrow *'b* **where**

edge-pred M = *inv* (*edge-succ M*)

locale *pre-digraph-map* = *pre-digraph* + **fixes** *M* :: *'b pre-map*

locale *digraph-map* = *fin-digraph* *G*
+ *pre-digraph-map* *G* *M*
+ *bidirected-digraph* *G* *edge-rev* *M* **for** *G* *M* +
assumes *edge-succ-permutes*: *edge-succ* *M* *permutes* *arcs* *G*
assumes *edge-succ-cyclic*: $\bigwedge v. v \in \text{verts } G \implies \text{out-arcs } G v \neq \{\}$ \implies *cyclic-on*
(*edge-succ* *M*) (*out-arcs* *G* *v*)

lemma (**in** *fin-digraph*) *digraph-mapI*:
assumes *bidi*: $\bigwedge a. a \notin \text{arcs } G \implies \text{edge-rev } M a = a$
 $\bigwedge a. a \in \text{arcs } G \implies \text{edge-rev } M a \neq a$
 $\bigwedge a. a \in \text{arcs } G \implies \text{edge-rev } M (\text{edge-rev } M a) = a$
 $\bigwedge a. a \in \text{arcs } G \implies \text{tail } G (\text{edge-rev } M a) = \text{head } G a$
assumes *edge-succ-permutes*: *edge-succ* *M* *permutes* *arcs* *G*
assumes *edge-succ-cyclic*: $\bigwedge v. v \in \text{verts } G \implies \text{out-arcs } G v \neq \{\}$ \implies *cyclic-on*
(*edge-succ* *M*) (*out-arcs* *G* *v*)
shows *digraph-map* *G* *M*
using *assms* **by** *unfold-locales* *auto*

lemma (**in** *fin-digraph*) *digraph-mapI-permutes*:
assumes *bidi*: *edge-rev* *M* *permutes* *arcs* *G*
 $\bigwedge a. a \in \text{arcs } G \implies \text{edge-rev } M a \neq a$
 $\bigwedge a. a \in \text{arcs } G \implies \text{edge-rev } M (\text{edge-rev } M a) = a$
 $\bigwedge a. a \in \text{arcs } G \implies \text{tail } G (\text{edge-rev } M a) = \text{head } G a$
assumes *edge-succ-permutes*: *edge-succ* *M* *permutes* *arcs* *G*
assumes *edge-succ-cyclic*: $\bigwedge v. v \in \text{verts } G \implies \text{out-arcs } G v \neq \{\}$ \implies *cyclic-on*
(*edge-succ* *M*) (*out-arcs* *G* *v*)
shows *digraph-map* *G* *M*
proof –
interpret *bidirected-digraph* *G* *edge-rev* *M* **using** *bidi* **by** *unfold-locales* (*auto*
simp: *permutes-def*)
show *?thesis*
using *edge-succ-permutes* *edge-succ-cyclic* **by** *unfold-locales*
qed

context *digraph-map*
begin

lemma *digraph-map[intro]*: *digraph-map* *G* *M* **by** *unfold-locales*

lemma *permutation-edge-succ*: *permutation* (*edge-succ* *M*)
by (*metis* *edge-succ-permutes* *finite-arcs* *permutation-permutes*)

lemma *edge-pred-succ[simp]*: *edge-pred* *M* (*edge-succ* *M* *a*) = *a*
by (*metis* *edge-pred-def* *edge-succ-permutes* *permutes-inverses*(2))

lemma *edge-succ-pred[simp]*: *edge-succ* *M* (*edge-pred* *M* *a*) = *a*
by (*metis* *edge-pred-def* *edge-succ-permutes* *permutes-inverses*(1))

lemma *edge-pred-permutes*: *edge-pred M permutes arcs G*
unfolding *edge-pred-def* **using** *edge-succ-permutes* **by** (*rule permutes-inv*)

lemma *permutation-edge-pred*: *permutation (edge-pred M)*
by (*metis edge-pred-permutes finite-arcs permutation-permutes*)

lemma *edge-succ-eq-iff[simp]*: $\bigwedge x y. \text{edge-succ } M x = \text{edge-succ } M y \longleftrightarrow x = y$
by (*metis edge-pred-succ*)

lemma *edge-rev-in-arcs[simp]*: $\text{edge-rev } M a \in \text{arcs } G \longleftrightarrow a \in \text{arcs } G$
by (*metis arev-arev arev-permutes-arcs permutes-not-in*)

lemma *edge-succ-in-arcs[simp]*: $\text{edge-succ } M a \in \text{arcs } G \longleftrightarrow a \in \text{arcs } G$
by (*metis edge-pred-succ edge-succ-permutes permutes-not-in*)

lemma *edge-pred-in-arcs[simp]*: $\text{edge-pred } M a \in \text{arcs } G \longleftrightarrow a \in \text{arcs } G$
by (*metis edge-succ-pred edge-pred-permutes permutes-not-in*)

lemma *tail-edge-succ[simp]*: $\text{tail } G (\text{edge-succ } M a) = \text{tail } G a$
proof *cases*
assume $a \in \text{arcs } G$
then have $\text{tail } G a \in \text{verts } G$ **by** *auto*
moreover
then have $\text{out-arcs } G (\text{tail } G a) \neq \{\}$
using $\langle a \in \text{arcs } G \rangle$ **by** *auto*
ultimately
have *cyclic-on (edge-succ M) (out-arcs G (tail G a))*
by (*rule edge-succ-cyclic*)
moreover
have $a \in \text{out-arcs } G (\text{tail } G a)$
using $\langle a \in \text{arcs } G \rangle$ **by** *simp*
ultimately
have $\text{edge-succ } M a \in \text{out-arcs } G (\text{tail } G a)$
by (*rule cyclic-on-inI*)
then show *?thesis* **by** *simp*
next
assume $a \notin \text{arcs } G$ **then show** *?thesis* **using** *edge-succ-permutes* **by** (*simp*
add: permutes-not-in)
qed

lemma *tail-edge-pred[simp]*: $\text{tail } G (\text{edge-pred } M a) = \text{tail } G a$
by (*metis edge-succ-pred tail-edge-succ*)

lemma *bij-edge-succ[intro]*: *bij (edge-succ M)*
using *edge-succ-permutes* **by** (*simp add: permutes-conv-has-dom*)

lemma *edge-pred-cyclic*:
assumes $v \in \text{verts } G \text{ out-arcs } G v \neq \{\}$
shows *cyclic-on (edge-pred M) (out-arcs G v)*

proof –

obtain a **where** $orb\text{-}a\text{-}eq$: $orbit\ (edge\text{-}succ\ M)\ a = out\text{-}arcs\ G\ v$
using $edge\text{-}succ\text{-}cyclic$ [$OF\ assms$] **by** ($auto\ simp$: $cyclic\text{-}on\text{-}def$)
have $cyclic\text{-}on\ (edge\text{-}pred\ M)\ (orbit\ (edge\text{-}pred\ M)\ a)$
using $permutation\text{-}edge\text{-}pred$ **by** ($rule\ cyclic\text{-}on\text{-}orbit'$)
also have $orbit\ (edge\text{-}pred\ M)\ a = orbit\ (edge\text{-}succ\ M)\ a$
unfolding $edge\text{-}pred\text{-}def$ **using** $permutation\text{-}edge\text{-}succ$ **by** ($rule\ orbit\text{-}inv\text{-}eq$)
finally show $cyclic\text{-}on\ (edge\text{-}pred\ M)\ (out\text{-}arcs\ G\ v)$ **by** ($simp\ add$: $orb\text{-}a\text{-}eq$)
qed

definition (**in** $pre\text{-}digraph\text{-}map$) $face\text{-}cycle\text{-}succ :: 'b \Rightarrow 'b$ **where**
 $face\text{-}cycle\text{-}succ \equiv edge\text{-}succ\ M\ o\ edge\text{-}rev\ M$

definition (**in** $pre\text{-}digraph\text{-}map$) $face\text{-}cycle\text{-}pred :: 'b \Rightarrow 'b$ **where**
 $face\text{-}cycle\text{-}pred \equiv edge\text{-}rev\ M\ o\ edge\text{-}pred\ M$

lemma $face\text{-}cycle\text{-}pred\text{-}succ$ [$simp$]:
shows $face\text{-}cycle\text{-}pred\ (face\text{-}cycle\text{-}succ\ a) = a$
unfolding $face\text{-}cycle\text{-}pred\text{-}def\ face\text{-}cycle\text{-}succ\text{-}def$ **by** $simp$

lemma $face\text{-}cycle\text{-}succ\text{-}pred$ [$simp$]:
shows $face\text{-}cycle\text{-}succ\ (face\text{-}cycle\text{-}pred\ a) = a$
unfolding $face\text{-}cycle\text{-}pred\text{-}def\ face\text{-}cycle\text{-}succ\text{-}def$ **by** $simp$

lemma $tail\text{-}face\text{-}cycle\text{-}succ$: $a \in arcs\ G \implies tail\ G\ (face\text{-}cycle\text{-}succ\ a) = head\ G\ a$
by ($auto\ simp$: $face\text{-}cycle\text{-}succ\text{-}def$)

lemma $funpow\text{-}prop$:
assumes $\bigwedge x. P\ (f\ x) \longleftrightarrow P\ x$
shows $P\ ((f\ \sim^n)\ x) \longleftrightarrow P\ x$
using $assms$ **by** ($induct\ n$) $auto$

lemma $face\text{-}cycle\text{-}succ\text{-}no\text{-}arc$ [$simp$]: $a \notin arcs\ G \implies face\text{-}cycle\text{-}succ\ a = a$
by ($auto\ simp$: $face\text{-}cycle\text{-}succ\text{-}def\ permutes\text{-}not\text{-}in$ [$OF\ arev\text{-}permutes\text{-}arcs$]
 $permutes\text{-}not\text{-}in$ [$OF\ edge\text{-}succ\text{-}permutes$])

lemma $funpow\text{-}face\text{-}cycle\text{-}succ\text{-}no\text{-}arc$ [$simp$]:
assumes $a \notin arcs\ G$ **shows** $(face\text{-}cycle\text{-}succ\ \sim^n)\ a = a$
using $assms$ **by** ($induct\ n$) $auto$

lemma $funpow\text{-}face\text{-}cycle\text{-}pred\text{-}no\text{-}arc$ [$simp$]:
assumes $a \notin arcs\ G$ **shows** $(face\text{-}cycle\text{-}pred\ \sim^n)\ a = a$
using $assms$
by ($induct\ n$) ($auto\ simp$: $face\text{-}cycle\text{-}pred\text{-}def\ permutes\text{-}not\text{-}in$ [$OF\ arev\text{-}permutes\text{-}arcs$]
 $permutes\text{-}not\text{-}in$ [$OF\ edge\text{-}pred\text{-}permutes$])

lemma $face\text{-}cycle\text{-}succ\text{-}closed$ [$simp$]:
 $face\text{-}cycle\text{-}succ\ a \in arcs\ G \longleftrightarrow a \in arcs\ G$

by (*metis comp-apply edge-rev-in-arcs edge-succ-in-arcs face-cycle-succ-def*)

lemma *face-cycle-pred-closed*[*simp*]:

face-cycle-pred $a \in \text{arcs } G \longleftrightarrow a \in \text{arcs } G$

by (*metis face-cycle-succ-closed face-cycle-succ-pred*)

lemma *face-cycle-succ-permutes*:

face-cycle-succ permutes arcs G

unfolding *face-cycle-succ-def*

using *arev-permutes-arcs edge-succ-permutes* **by** (*rule permutes-compose*)

lemma *permutation-face-cycle-succ*: permutation *face-cycle-succ*

using *face-cycle-succ-permutes finite-arcs* **by** (*metis permutation-permutes*)

lemma *bij-face-cycle-succ*: bij *face-cycle-succ*

using *face-cycle-succ-permutes* **by** (*simp add: permutes-conv-has-dom*)

lemma *face-cycle-pred-permutes*:

face-cycle-pred permutes arcs G

unfolding *face-cycle-pred-def*

using *edge-pred-permutes arev-permutes-arcs* **by** (*rule permutes-compose*)

definition (*in pre-digraph-map*) *face-cycle-set* :: 'b \Rightarrow 'b set **where**

face-cycle-set $a = \text{orbit } \text{face-cycle-succ } a$

definition (*in pre-digraph-map*) *face-cycle-sets* :: 'b set set **where**

face-cycle-sets = *face-cycle-set* ' arcs G

lemma *face-cycle-set-altdef*: *face-cycle-set* $a = \{(\text{face-cycle-succ } \sim n) a \mid n. \text{True}\}$

unfolding *face-cycle-set-def*

by (*intro orbit-altdef-self-in permutation-self-in-orbit permutation-face-cycle-succ*)

lemma *face-cycle-set-self*[*simp, intro*]: $a \in \text{face-cycle-set } a$

unfolding *face-cycle-set-def* **using** *permutation-face-cycle-succ* **by** (*rule permutation-self-in-orbit*)

lemma *empty-not-in-face-cycle-sets*: $\{\} \notin \text{face-cycle-sets}$

by (*auto simp: face-cycle-sets-def*)

lemma *finite-face-cycle-set*[*simp, intro*]: finite (*face-cycle-set* a)

using *face-cycle-set-self* **unfolding** *face-cycle-set-def* **by** (*simp add: finite-orbit*)

lemma *finite-face-cycle-sets*[*simp, intro*]: finite *face-cycle-sets*

by (*auto simp: face-cycle-sets-def*)

lemma *face-cycle-set-induct*[*case-names base step, induct set: face-cycle-set*]:

assumes *consume*: $a \in \text{face-cycle-set } x$

and *ih-base*: $P x$

and *ih-step*: $\bigwedge y. y \in \text{face-cycle-set } x \implies P y \implies P (\text{face-cycle-succ } y)$
shows $P a$
using *consume* **unfolding** *face-cycle-set-def*
by *induct* (*auto simp: ih-step face-cycle-set-def[symmetric] ih-base*)

lemma *face-cycle-succ-cyclic*:
cyclic-on face-cycle-succ (face-cycle-set a)
unfolding *face-cycle-set-def* **using** *permutation-face-cycle-succ* **by** (*rule cyclic-on-orbit'*)

lemma *face-cycle-eq*:
assumes $b \in \text{face-cycle-set } a$ **shows** $\text{face-cycle-set } b = \text{face-cycle-set } a$
using *assms* **unfolding** *face-cycle-set-def*
by (*auto intro: orbit-swap orbit-trans permutation-face-cycle-succ permutation-self-in-orbit*)

lemma *face-cycle-succ-in-arcsI*: $\bigwedge a. a \in \text{arcs } G \implies \text{face-cycle-succ } a \in \text{arcs } G$
by (*auto simp: face-cycle-succ-def*)

lemma *face-cycle-succ-inI*: $\bigwedge x y. x \in \text{face-cycle-set } y \implies \text{face-cycle-succ } x \in \text{face-cycle-set } y$
by (*metis face-cycle-succ-cyclic cyclic-on-inI*)

lemma *face-cycle-succ-inD*: $\bigwedge x y. \text{face-cycle-succ } x \in \text{face-cycle-set } y \implies x \in \text{face-cycle-set } y$
by (*metis face-cycle-eq face-cycle-set-self face-cycle-succ-inI*)

lemma *face-cycle-set-parts*:
 $\text{face-cycle-set } a = \text{face-cycle-set } b \vee \text{face-cycle-set } a \cap \text{face-cycle-set } b = \{\}$
by (*metis disjoint-iff-not-equal face-cycle-eq*)

definition *fc-equiv* :: $'b \Rightarrow 'b \Rightarrow \text{bool}$ **where**
 $\text{fc-equiv } a b \equiv a \in \text{face-cycle-set } b$

lemma *reflp-fc-equiv*: *reflp fc-equiv*
by (*rule reflpI*) (*simp add: fc-equiv-def*)

lemma *symp-fc-equiv*: *symp fc-equiv*
using *face-cycle-set-parts*
by (*intro sympI*) (*auto simp: fc-equiv-def*)

lemma *transp-fc-equiv*: *transp fc-equiv*
using *face-cycle-set-parts*
by (*intro transpI*) (*auto simp: fc-equiv-def*)

lemma *equivp fc-equiv*
by (*intro equivpI reflp-fc-equiv symp-fc-equiv transp-fc-equiv*)

lemma *in-face-cycle-setD*:
assumes $y \in \text{face-cycle-set } x$ $x \in \text{arcs } G$ **shows** $y \in \text{arcs } G$

```

    using assms
  by (auto simp: face-cycle-set-def dest: permutes-orbit-subset[OF face-cycle-succ-permutes])

lemma in-face-cycle-setsD:
  assumes  $x \in \text{face-cycle-sets}$  shows  $x \subseteq \text{arcs } G$ 
  using assms by (auto simp: face-cycle-sets-def dest: in-face-cycle-setD)

end

definition (in pre-digraph) isolated-verts :: 'a set where
  isolated-verts  $\equiv \{v \in \text{verts } G. \text{out-arcs } G \ v = \{\}\}$ 

definition (in pre-digraph-map) euler-char :: int where
  euler-char  $\equiv \text{int } (\text{card } (\text{verts } G)) - \text{int } (\text{card } (\text{arcs } G) \ \text{div } 2) + \text{int } (\text{card } \text{face-cycle-sets})$ 

definition (in pre-digraph-map) euler-genus :: int where
  euler-genus  $\equiv (\text{int } (2 * \text{card } \text{sccs}) - \text{int } (\text{card } \text{isolated-verts}) - \text{euler-char}) \ \text{div } 2$ 

definition comb-planar :: ('a,'b) pre-digraph  $\Rightarrow$  bool where
  comb-planar  $G \equiv \exists M. \text{digraph-map } G \ M \wedge \text{pre-digraph-map.euler-genus } G \ M = 0$ 

Number of isolated vertices is a graph invariant

context
  fixes  $G$  hom assumes hom: pre-digraph.digraph-isomorphism  $G$  hom
begin

  interpretation wf-digraph  $G$  using hom by (auto simp: pre-digraph.digraph-isomorphism-def)

  lemma isolated-verts-app-iso[simp]:
    pre-digraph.isolated-verts (app-iso hom  $G$ ) = iso-verts hom ' isolated-verts
  using hom
  by (auto simp: pre-digraph.isolated-verts-def iso-verts-tail inj-image-mem-iff
    out-arcs-app-iso-eq)

  lemma card-isolated-verts-iso[simp]:
    card (iso-verts hom ' pre-digraph.isolated-verts  $G$ ) = card isolated-verts
  apply (rule card-image)
  using hom apply (rule digraph-isomorphism-inj-on-verts[THEN inj-on-subset])
  apply (auto simp: isolated-verts-def)
  done

end

context digraph-map begin

```

lemma *face-cycle-succ-neq*:
assumes $a \in \text{arcs } G$ $\text{tail } G \ a \neq \text{head } G \ a$ **shows** $\text{face-cycle-succ } a \neq a$
proof –
from *assms* **have** $\text{edge-rev } M \ a \in \text{arcs } G$
by (*subst edge-rev-in-arcs simp*)
then have $\text{cyclic-on } (\text{edge-succ } M) (\text{out-arcs } G (\text{tail } G (\text{edge-rev } M \ a)))$
by (*intro edge-succ-cyclic*) (*auto dest: tail-in-verts simp: out-arcs-def intro: exI*[*where x=edge-rev M a*])
then have $\text{edge-succ } M (\text{edge-rev } M \ a) \in (\text{out-arcs } G (\text{tail } G (\text{edge-rev } M \ a)))$
by (*rule cyclic-on-inI*) (*auto simp: edge-rev M a ∈ ->[simplified]*)
moreover have $\text{tail } G (\text{edge-succ } M (\text{edge-rev } M \ a)) = \text{head } G \ a$
using *assms* **by** *auto*
then have $\text{edge-succ } M (\text{edge-rev } M \ a) \neq a$ **using** *assms* **by** *metis*
ultimately show *?thesis*
using *assms* **by** (*auto simp: face-cycle-succ-def*)
qed

end

2 Maps and Isomorphism

definition (*in pre-digraph*)
 $\text{wrap-iso-arcs hom } f = \text{perm-restrict } (\text{iso-arcs hom } o \ f \ o \ \text{iso-arcs } (\text{inv-iso hom}))$
 $(\text{arcs } (\text{app-iso hom } G))$

definition (*in pre-digraph-map*) $\text{map-iso} :: ('a, 'b, 'a2, 'b2) \text{ digraph-isomorphism} \Rightarrow 'b2 \text{ pre-map}$ **where**
 $\text{map-iso } f \equiv$
 $(\ \text{edge-rev} = \text{wrap-iso-arcs } f (\text{edge-rev } M)$
 $\ , \ \text{edge-succ} = \text{wrap-iso-arcs } f (\text{edge-succ } M)$
 $\)$

lemma *funcsetI-permutes*:
assumes $f \text{ permutes } S$ **shows** $f \in S \rightarrow S$
by (*metis assms funcsetI permutes-in-image*)

context

fixes $G \ \text{hom}$ **assumes** $\text{hom}: \text{pre-digraph.digraph-isomorphism } G \ \text{hom}$
begin

interpretation $\text{wf-digraph } G$ **using** hom **by** (*auto simp: pre-digraph.digraph-isomorphism-def*)

lemma *wrap-iso-arcs-iso-arcs[simp]*:
assumes $x \in \text{arcs } G$
shows $\text{wrap-iso-arcs hom } f (\text{iso-arcs hom } x) = \text{iso-arcs hom } (f \ x)$
using *assms hom* **by** (*auto simp: wrap-iso-arcs-def perm-restrict-def*)

lemma *inj-on-wrap-iso-arcs*:
assumes $\text{dom}: \bigwedge f. f \in F \Longrightarrow \text{has-dom } f (\text{arcs } G)$

```

assumes funcset:  $F \subseteq \text{arcs } G \rightarrow \text{arcs } G$ 
shows inj-on (wrap-iso-arcs hom)  $F$ 
proof (rule inj-onI)
  fix  $f\ g$  assume  $F: f \in F\ g \in F$  and  $eq: \text{wrap-iso-arcs hom } f = \text{wrap-iso-arcs}$ 
hom } g
  { fix  $x$  assume  $x \notin \text{arcs } G$ 
    then have  $f\ x = x\ g\ x = x$  using  $F\ \text{dom}$  by (auto simp: has-dom-def)
    then have  $f\ x = g\ x$  by simp
  }
moreover
  { fix  $x$  assume  $x \in \text{arcs } G$ 
    then have  $f\ x \in \text{arcs } G\ g\ x \in \text{arcs } G$  using  $F\ \text{funcset}$  by auto
    with digraph-isomorphism-inj-on-arcs[OF hom] -
    have  $\text{iso-arcs hom } (f\ x) = \text{iso-arcs hom } (g\ x) \implies f\ x = g\ x$ 
      by (rule inj-onD)
    then have  $f\ x = g\ x$ 
      using assms hom  $\langle x \in \text{arcs } G \rangle\ eq$ 
      by (auto simp: wrap-iso-arcs-def fun-eq-iff perm-restrict-def split: if-splits)
  }
ultimately show  $f = g$  by auto
qed

```

```

lemma inj-on-wrap-iso-arcs-f:
  assumes  $A \subseteq \text{arcs } G\ f \in A \rightarrow A\ B = \text{iso-arcs hom } 'A$ 
  assumes inj-on  $f\ A$  shows inj-on (wrap-iso-arcs hom)  $f\ B$ 
proof (rule inj-onI)
  fix  $x\ y$ 
  assume in-hom-A:  $x \in B\ y \in B$ 
  and wia-eq:  $\text{wrap-iso-arcs hom } f\ x = \text{wrap-iso-arcs hom } f\ y$ 
  from in-hom-A  $\langle B = \rightarrow \rangle$  obtain  $x0$  where  $x0: x = \text{iso-arcs hom } x0\ x0 \in A$  by
auto
  from in-hom-A  $\langle B = \rightarrow \rangle$  obtain  $y0$  where  $y0: y = \text{iso-arcs hom } y0\ y0 \in A$ 
by auto
  have arcs-0:  $x0 \in \text{arcs } G\ y0 \in \text{arcs } G\ f\ x0 \in \text{arcs } G\ f\ y0 \in \text{arcs } G$ 
    using  $x0\ y0\ \langle A \subseteq \rightarrow \rangle\ \langle f \in \rightarrow \rangle$  by auto

  have ( $\text{iso-arcs hom } o\ f\ o\ \text{iso-arcs } (\text{inv-iso hom})$ )  $x = (\text{iso-arcs hom } o\ f\ o\ \text{iso-arcs}$ 
(inv-iso hom))  $y$ 
    using in-hom-A wia-eq assms(1)  $\langle B = \rightarrow \rangle$  by (auto simp: wrap-iso-arcs-def
perm-restrict-def split: if-splits)
  then show  $x = y$ 
    using hom assms digraph-isomorphism-inj-on-arcs[OF hom]  $x0\ y0\ \text{arcs-0}$ 
\langle inj-on f A \rangle\ \langle A \subseteq \rightarrow \rangle
    by (auto dest!: inj-onD)
qed

```

```

lemma wrap-iso-arcs-in-funcsetI:
  assumes  $A \subseteq \text{arcs } G\ f \in A \rightarrow A$ 
  shows  $\text{wrap-iso-arcs hom } f \in \text{iso-arcs hom } 'A \rightarrow \text{iso-arcs hom } 'A$ 

```

```

proof
  fix  $x$  assume  $x \in \text{iso-arcs hom } 'A$ 
  then obtain  $x0$  where  $x = \text{iso-arcs hom } x0$   $x0 \in A$  by blast
  then have  $f x0 \in A$  using  $\langle f \in \rightarrow \rangle$  by auto
  then show  $\text{wrap-iso-arcs hom } f x \in \text{iso-arcs hom } 'A$ 
  unfolding  $\langle x = \rightarrow \rangle$  using  $\langle x0 \in A \rangle$  assms hom by (auto simp: wrap-iso-arcs-def
perm-restrict-def)
qed

lemma wrap-iso-arcs-permutes:
  assumes  $A \subseteq \text{arcs } G$   $f$  permutes  $A$ 
  shows  $\text{wrap-iso-arcs hom } f$  permutes ( $\text{iso-arcs hom } 'A$ )
proof -
  { fix  $x$  assume  $A: x \notin \text{iso-arcs hom } 'A$ 
    have  $\text{wrap-iso-arcs hom } f x = x$ 
    proof cases
      assume  $x \in \text{iso-arcs hom } ' \text{arcs } G$ 
      then have  $\text{iso-arcs } (\text{inv-iso hom}) x \notin A$   $x \in \text{arcs } (\text{app-iso hom } G)$ 
      using  $A$  hom by (metis arcs-app-iso image-eqI pre-digraph.iso-arcs-iso-inv,
simp)
      then have  $f (\text{iso-arcs } (\text{inv-iso hom}) x) = (\text{iso-arcs } (\text{inv-iso hom}) x)$ 
      using  $\langle f \text{ permutes } A \rangle$  by (simp add: permutes-not-in)
      then show ?thesis using hom assms  $\langle x \in \text{arcs } \rightarrow \rangle$ 
      by (simp add: wrap-iso-arcs-def perm-restrict-def)
    next
      assume  $x \notin \text{iso-arcs hom } ' \text{arcs } G$ 
      then show ?thesis
      by (simp add: wrap-iso-arcs-def perm-restrict-def)
    qed
  } note not-in-id = this

  have  $f \in A \rightarrow A$  using assms by (intro funcsetI-permutes)
  have inj-on-wrap: inj-on ( $\text{wrap-iso-arcs hom } f$ ) ( $\text{iso-arcs hom } 'A$ )
  using assms  $\langle f \in A \rightarrow A \rangle$  by (intro inj-on-wrap-iso-arcs-f) (auto intro:
inj-on-subset permutes-inj)
  have woa-in-fs: wrap-iso-arcs hom  $f \in \text{iso-arcs hom } 'A \rightarrow \text{iso-arcs hom } 'A$ 
  using assms  $\langle f \in A \rightarrow A \rangle$  by (intro wrap-iso-arcs-in-funcsetI)

  { fix  $x y$  assume  $\text{wrap-iso-arcs hom } f x = \text{wrap-iso-arcs hom } f y$ 
    then have  $x = y$ 
    apply (cases  $x \in \text{iso-arcs hom } 'A$ ; cases  $y \in \text{iso-arcs hom } 'A$ )
    using woa-in-fs inj-on-wrap by (auto dest: inj-onD simp: not-in-id)
  } note uniqueD = this

  note  $\langle f \text{ permutes } A \rangle$ 
  moreover
  note not-in-id
  moreover
  { fix  $y$  have  $\exists x. \text{wrap-iso-arcs hom } f x = y$ 

```

```

proof cases
  assume  $y \in \text{iso-arcs hom } A$ 
  then obtain  $y0$  where  $y0 \in A$   $\text{iso-arcs hom } y0 = y$  by blast
    with  $\langle f \text{ permutes } A \rangle$  obtain  $x0$  where  $x0 \in A$   $f x0 = y0$  unfolding
    permutes-def by metis
  moreover
    then have  $\bigwedge x. x \in \text{arcs } G \implies \text{iso-arcs hom } x0 = \text{iso-arcs hom } x \implies x0$ 
    =  $x$ 
    using assms hom by (auto simp: digraph-isomorphism-def dest: inj-onD)
  ultimately
    have  $\text{wrap-iso-arcs hom } f (\text{iso-arcs hom } x0) = y$ 
    using  $\langle - = y \rangle$  assms hom by (auto simp: wrap-iso-arcs-def perm-restrict-def)
    then show ?thesis ..
  qed (metis not-in-id)
}
ultimately
show ?thesis unfolding permutes-def by (auto simp: dest: uniqueD)
qed

end

```

```

lemma (in digraph-map) digraph-map-isoI:
  assumes digraph-isomorphism hom shows digraph-map (app-iso hom G) (map-iso
  hom)
proof –
  interpret  $iG: \text{fin-digraph app-iso hom } G$  using assms by (rule fin-digraphI-app-iso)
  show ?thesis
  proof (rule  $iG.\text{digraph-mapI-permutes}$ )
    show  $\text{edge-rev } (\text{map-iso hom}) \text{ permutes arcs } (\text{app-iso hom } G)$ 
    using assms unfolding map-iso-def by (simp add: wrap-iso-arcs-permutes
    arev-permutes-arcs)
  next
    show  $\text{edge-succ } (\text{map-iso hom}) \text{ permutes arcs } (\text{app-iso hom } G)$ 
    using assms unfolding map-iso-def by (simp add: wrap-iso-arcs-permutes
    edge-succ-permutes)
  next
    fix  $a$  assume  $A: a \in \text{arcs } (\text{app-iso hom } G)$ 
    show  $\text{tail } (\text{app-iso hom } G) (\text{edge-rev } (\text{map-iso hom}) a) = \text{head } (\text{app-iso hom }
    G) a$ 
    using A assms
    by (cases rule: in-arcs-app-iso-cases) (auto simp: map-iso-def iso-verts-tail
    iso-verts-head)
    show  $\text{edge-rev } (\text{map-iso hom}) (\text{edge-rev } (\text{map-iso hom}) a) = a$ 
    using A assms by (cases rule: in-arcs-app-iso-cases) (auto simp: map-iso-def)
    show  $\text{edge-rev } (\text{map-iso hom}) a \neq a$ 
    using A assms by (auto simp: map-iso-def arev-neq)
  next
    fix  $v$  assume  $v \in \text{verts } (\text{app-iso hom } G)$  and oa-hom: out-arcs (app-iso hom
    G)  $v \neq \{\}$ 

```

```

then obtain v0 where v0 ∈ verts G v = iso-verts hom v0 by auto
moreover
then have oa: out-arcs G v0 ≠ {}
  using assms oa-hom by (auto simp: out-arcs-def iso-verts-tail)
ultimately
have cyclic-on-v0: cyclic-on (edge-succ M) (out-arcs G v0)
  by (intro edge-succ-cyclic)

from oa-hom obtain a where a ∈ out-arcs (app-iso hom G) v by blast
then obtain a0 where a0 ∈ arcs G a = iso-arcs hom a0 by auto
then have a0 ∈ out-arcs G v0
  using ⟨v = -⟩ ⟨v0 ∈ -⟩ ⟨a ∈ -⟩ assms by (simp add: iso-verts-tail)

show cyclic-on (edge-succ (map-iso hom)) (out-arcs (app-iso hom G) v)
proof (rule cyclic-on-singleI)
  show a ∈ out-arcs (app-iso hom G) v by fact
next
  have out-arcs (app-iso hom G) v = iso-arcs hom ‘ out-arcs G v0
    unfolding ⟨v = -⟩ by (rule out-arcs-app-iso-eq) fact+
  also have out-arcs G v0 = orbit (edge-succ M) a0
    using cyclic-on-v0 ⟨a0 ∈ out-arcs G v0⟩ unfolding cyclic-on-alldef by
simp
  also have iso-arcs hom ‘ ... = orbit (edge-succ (map-iso hom)) a
  proof -
    have ∧x. x ∈ orbit (edge-succ M) a0 ⟹ x ∈ arcs G
      using ⟨out-arcs G v0 = -⟩ by auto
    then show ?thesis using ⟨out-arcs G v0 = -⟩
      unfolding ⟨a = -⟩ using ⟨a0 ∈ out-arcs G v0⟩ assms
      by (intro orbit-inverse) (auto simp: map-iso-def)
  qed
  finally show out-arcs (app-iso hom G) v = orbit (edge-succ (map-iso hom))
a .
  qed
  qed
qed

end
theory List-Aux
imports
  List-Index.List-Index
begin

```

3 Auxiliary List Lemmas

```

lemma nth-rotate-conv-nth1-conv-nth:
  assumes m < length xs
  shows rotate1 xs ! m = xs ! (Suc m mod length xs)
  using assms
proof (induction xs arbitrary: m)

```

```

case (Cons x xs)
show ?case
proof (cases  $m < \text{length } xs$ )
  case False
  with Cons.prems have  $m = \text{length } xs$  by force
  then show ?thesis by (auto simp: nth-append)
qed (auto simp: nth-append)
qed simp

lemma nth-rotate-conv-nth:
  assumes  $m < \text{length } xs$ 
  shows  $\text{rotate } n \text{ } xs ! m = xs ! ((m + n) \bmod \text{length } xs)$ 
  using assms
proof (induction n arbitrary: m)
  case 0 then show ?case by simp
next
  case (Suc n)
  show ?case
  proof cases
    assume  $m + 1 < \text{length } xs$ 
    with Suc show ?thesis using Suc by (auto simp: nth-rotate-conv-nth1-conv-nth)
  next
    assume  $\neg(m + 1 < \text{length } xs)$ 
    with Suc have  $m + 1 = \text{length } xs$   $0 < \text{length } xs$  by auto
    moreover
    { have  $\text{Suc } (m + n) \bmod \text{length } xs = (\text{Suc } m + n) \bmod \text{length } xs$ 
      by auto
      also have  $\dots = n \bmod \text{length } xs$  using  $\langle m + 1 = \rightarrow \rangle$  by simp
      finally have  $\text{Suc } (m + n) \bmod \text{length } xs = n \bmod \text{length } xs$  . }
    ultimately
    show ?thesis by (auto simp: nth-rotate-conv-nth1-conv-nth Suc.IH)
  qed
qed

end

```

4 Permutations as Products of Disjoint Cycles

```

theory Executable-Permutations
imports
  HOL-Combinatorics.Permutations
  Graph-Theory.Auxiliary
  List-Aux
begin

```

4.1 Cyclic Permutations

```

definition list-succ :: 'a list  $\Rightarrow$  'a  $\Rightarrow$  'a where
  list-succ xs x = (if  $x \in \text{set } xs$  then  $xs ! ((\text{index } xs \text{ } x + 1) \bmod \text{length } xs)$  else x)

```

We demonstrate the functions on the following simple lemmas

list-succ [1, 2, 3] 1 = 2 *list-succ* [1, 2, 3] 2 = 3 *list-succ* [1, 2, 3] 3 = 1

lemma *list-succ-altdef*:

list-succ xs x = (let $n = \text{index } xs \ x$ in if $n + 1 = \text{length } xs$ then $xs ! 0$ else if $n + 1 < \text{length } xs$ then $xs ! (n + 1)$ else x)

using *index-le-size*[of $xs \ x$] **unfolding** *list-succ-def index-less-size-conv*[symmetric]
by (auto simp: Let-def)

lemma *list-succ-Nil*:

list-succ [] = *id*

by (simp add: list-succ-def fun-eq-iff)

lemma *list-succ-singleton*:

list-succ [x] = *list-succ []*

by (simp add: fun-eq-iff list-succ-def)

lemma *list-succ-short*:

assumes $\text{length } xs < 2$ **shows** *list-succ xs* = *id*

using *assms*

by (cases xs) (rename-tac [2] $y \ ys$, case-tac [2] ys , auto simp: list-succ-Nil list-succ-singleton)

lemma *list-succ-simps*:

$\text{index } xs \ x + 1 = \text{length } xs \implies \text{list-succ } xs \ x = xs ! 0$

$\text{index } xs \ x + 1 < \text{length } xs \implies \text{list-succ } xs \ x = xs ! (\text{index } xs \ x + 1)$

$\text{length } xs \leq \text{index } xs \ x \implies \text{list-succ } xs \ x = x$

by (auto simp: list-succ-altdef)

lemma *list-succ-not-in*:

assumes $x \notin \text{set } xs$ **shows** *list-succ xs x* = x

using *assms* **by** (auto simp: list-succ-def)

lemma *list-succ-list-succ-rev*:

assumes *distinct xs* **shows** *list-succ (rev xs)* (*list-succ xs x*) = x

proof –

{ **assume** $\text{index } xs \ x + 1 < \text{length } xs$

moreover then have $\text{length } xs - \text{Suc } (\text{Suc } (\text{length } xs - \text{Suc } (\text{Suc } (\text{index } xs \ x)))) = \text{index } xs \ x$

by *linarith*

ultimately have *?thesis* **using** *assms*

by (simp add: list-succ-def index-rev index-nth-id rev-nth)

}

moreover

{ **assume** $A: \text{index } xs \ x + 1 = \text{length } xs$

moreover

from A **have** $xs \neq []$ **by** *auto*

moreover

with A **have** $\text{last } xs = xs ! \text{index } xs \ x$ **by** (cases $\text{length } xs$) (auto simp: last-conv-nth)

```

ultimately
have ?thesis
  using assms
  by (auto simp add: list-succ-def rev-nth index-rev index-nth-id last-conv-nth)
}
moreover
{ assume A: index xs x ≥ length xs
  then have x ∉ set xs by (metis index-less less-irrefl)
  then have ?thesis by (auto simp: list-succ-def) }
ultimately show ?thesis
  by linarith
qed

lemma inj-list-succ: distinct xs ⇒ inj (list-succ xs)
  by (metis injI list-succ-list-succ-rev)

lemma inv-list-succ-eq: distinct xs ⇒ inv (list-succ xs) = list-succ (rev xs)
  by (metis distinct-rev inj-imp-inv-eq inj-list-succ list-succ-list-succ-rev)

lemma bij-list-succ: distinct xs ⇒ bij (list-succ xs)
  by (metis bij-def inj-list-succ distinct-rev list-succ-list-succ-rev surj-def)

lemma list-succ-permutes:
  assumes distinct xs shows list-succ xs permutes set xs
  using assms by (auto simp: permutes-conv-has-dom bij-list-succ has-dom-def
list-succ-def)

lemma permutation-list-succ:
  assumes distinct xs shows permutation (list-succ xs)
  using list-succ-permutes[OF assms] by (auto simp: permutation-permutes)

lemma list-succ-nth:
  assumes distinct xs n < length xs shows list-succ xs (xs ! n) = xs ! (Suc n mod
length xs)
  using assms by (auto simp: list-succ-def index-nth-id)

lemma list-succ-last[simp]:
  assumes distinct xs xs ≠ [] shows list-succ xs (last xs) = hd xs
  using assms by (auto simp: list-succ-def hd-conv-nth)

lemma list-succ-rotate1[simp]:
  assumes distinct xs shows list-succ (rotate1 xs) = list-succ xs
proof (rule ext)
fix y show list-succ (rotate1 xs) y = list-succ xs y
  using assms
proof (induct xs)
case Nil then show ?case by simp
next
case (Cons x xs)

```

```

show ?case
proof (cases x = y)
  case True
    then have index (xs @ [y]) y = length xs
      using ⟨distinct (x # xs)⟩ by (simp add: index-append)
    with True show ?thesis by (cases xs=[]) (auto simp: list-succ-def nth-append)
  next
    case False
    then show ?thesis
      apply (cases index xs y + 1 < length xs)
      apply (auto simp: list-succ-def index-append nth-append)
      by (metis Suc-lessI index-less-size-conv mod-self nth-Cons-0 nth-append
nth-append-length)
    qed
  qed
qed

```

```

lemma list-succ-rotate[simp]:
  assumes distinct xs shows list-succ (rotate n xs) = list-succ xs
  using assms by (induct n) auto

```

```

lemma list-succ-in-conv:
  list-succ xs x ∈ set xs ⟷ x ∈ set xs
unfolding list-succ-def using length-pos-if-in-set by force

```

```

lemma list-succ-in-conv1:
  assumes A ∩ set xs = {}
  shows list-succ xs x ∈ A ⟷ x ∈ A
  by (metis assms disjoint-iff-not-equal list-succ-in-conv list-succ-not-in)

```

```

lemma list-succ-commute:
  assumes set xs ∩ set ys = {}
  shows list-succ xs (list-succ ys x) = list-succ ys (list-succ xs x)
proof –
  have  $\bigwedge x. x \in \text{set } xs \implies \text{list-succ } ys \ x = x$ 
     $\bigwedge x. x \in \text{set } ys \implies \text{list-succ } xs \ x = x$ 
    using assms by (blast intro: list-succ-not-in)+
  then show ?thesis
    by (cases x ∈ set xs ∪ set ys) (auto simp: list-succ-in-conv list-succ-not-in)
qed

```

4.2 Arbitrary Permutations

```

fun lists-succ :: 'a list list ⇒ 'a ⇒ 'a where
  lists-succ [] x = x
| lists-succ (xs # xss) x = list-succ xs (lists-succ xss x)

```

```

definition distincts :: 'a list list ⇒ bool where
  distincts xss ≡ distinct xss ∧ (∀ xs ∈ set xss. distinct xs ∧ xs ≠ []) ∧ (∀ xs ∈ set

```

$xss. \forall ys \in set\ xss. xs \neq ys \longrightarrow set\ xs \cap set\ ys = \{\}$)

lemma *distincts-distinct*: $distincts\ xss \implies distinct\ xss$
by (*auto simp: distincts-def*)

lemma *distincts-Nil*[*simp*]: $distincts\ []$
by (*simp add: distincts-def*)

lemma *distincts-single*: $distincts\ [xs] \longleftrightarrow distinct\ xs \wedge xs \neq []$
by (*auto simp add: distincts-def*)

lemma *distincts-Cons*: $distincts\ (xs \# xss) \longleftrightarrow xs \neq [] \wedge distinct\ xs \wedge distincts\ xss \wedge (set\ xs \cap (\bigcup ys \in set\ xss. set\ ys)) = \{\}$ (**is** $?L \longleftrightarrow ?R$)

proof

assume $?L$ **then show** $?R$ **by** (*auto simp: distincts-def*)

next

assume $?R$

then have $distinct\ (xs \# xss)$

apply (*auto simp: disjoint-iff-not-equal distincts-distinct*)

apply (*metis length-greater-0-conv nth-mem*)

done

moreover

from $\langle ?R \rangle$ **have** $\forall xs \in set\ (xs \# xss). distinct\ xs \wedge xs \neq []$

by (*auto simp: distincts-def*)

moreover

from $\langle ?R \rangle$ **have** $\forall xs' \in set\ (xs \# xss). \forall ys \in set\ (xs \# xss). xs' \neq ys \longrightarrow set\ xs' \cap set\ ys = \{\}$

by (*simp add: distincts-def*) *blast*

ultimately show $?L$ **unfolding** *distincts-def* **by** (*intro conjI*)

qed

lemma *distincts-Cons'*: $distincts\ (xs \# xss) \longleftrightarrow xs \neq [] \wedge distinct\ xs \wedge distincts\ xss \wedge (\forall ys \in set\ xss. set\ xs \cap set\ ys = \{\})$
(**is** $?L \longleftrightarrow ?R$)
unfolding *distincts-Cons* **by** *blast*

lemma *distincts-rev*:

$distincts\ (map\ rev\ xss) \longleftrightarrow distincts\ xss$

by (*simp add: distincts-def distinct-map*)

lemma *length-distincts*:

assumes $distincts\ xss$

shows $length\ xss = card\ (set\ ' set\ xss)$

using *assms*

proof (*induct xss*)

case *Nil* **then show** $?case$ **by** *simp*

next

case (*Cons xs xss*)

then have $set\ xs \not\subseteq set\ 'set\ xss$
using $equalsOI[of\ set\ xs]$ **by** $(auto\ simp:\ distincts-Cons\ disjoint-iff-not-equal)$)
with $Cons$ **show** $?case$ **by** $(auto\ simp\ add:\ distincts-Cons)$
qed

lemma $distincts-remove1$: $distincts\ xss \implies distincts\ (remove1\ xs\ xss)$
by $(auto\ simp:\ distincts-def)$

lemma $distinct-Cons-remove1$:
 $x \in set\ xs \implies distinct\ (x \# remove1\ x\ xs) = distinct\ xs$
by $(induct\ xs)\ auto$

lemma $set-Cons-remove1$:
 $x \in set\ xs \implies set\ (x \# remove1\ x\ xs) = set\ xs$
by $(induct\ xs)\ auto$

lemma $distincts-Cons-remove1$:
 $xs \in set\ xss \implies distincts\ (xs \# remove1\ xs\ xss) = distincts\ xss$
by $(simp\ only:\ distinct-Cons-remove1\ set-Cons-remove1\ distincts-def)$

lemma $distincts-inj-on-set$:
assumes $distincts\ xss$ **shows** $inj-on\ set\ (set\ xss)$
by $(rule\ inj-onI)\ (metis\ assms\ distincts-def\ inf.idem\ set-empty)$

lemma $distincts-distinct-set$:
assumes $distincts\ xss$ **shows** $distinct\ (map\ set\ xss)$
using $assms$ **by** $(auto\ simp:\ distinct-map\ distincts-distinct\ distincts-inj-on-set)$

lemma $distincts-distinct-nth$:
assumes $distincts\ xss$ $n < length\ xss$ **shows** $distinct\ (xss\ !\ n)$
using $assms$ **by** $(auto\ simp:\ distincts-def)$

lemma $lists-succ-not-in$:
assumes $x \notin (\bigcup_{xs \in set\ xss} set\ xs)$ **shows** $lists-succ\ xss\ x = x$
using $assms$ **by** $(induct\ xss)\ (auto\ simp:\ list-succ-not-in)$

lemma $lists-succ-in-conv$:
 $lists-succ\ xss\ x \in (\bigcup_{xs \in set\ xss} set\ xs) \longleftrightarrow x \in (\bigcup_{xs \in set\ xss} set\ xs)$
by $(induct\ xss)\ (auto\ simp:\ list-succ-in-conv\ lists-succ-not-in\ list-succ-not-in)$

lemma $lists-succ-in-conv1$:
assumes $A \cap (\bigcup_{xs \in set\ xss} set\ xs) = \{\}$
shows $lists-succ\ xss\ x \in A \longleftrightarrow x \in A$
by $(metis\ Int-iff\ assms\ emptyE\ lists-succ-in-conv\ lists-succ-not-in)$

lemma $lists-succ-Cons-pf$: $lists-succ\ (xs \# xss) = list-succ\ xs\ o\ lists-succ\ xss$
by $auto$

lemma $lists-succ-Nil-pf$: $lists-succ\ [] = id$

by (*simp add: fun-eq-iff*)

lemmas *lists-succ-simps-pf = lists-succ-Cons-pf lists-succ-Nil-pf*

lemma *lists-succ-permutes*:
assumes *distincts xss*
shows *lists-succ xss permutes (∪ xs ∈ set xss. set xs)*
using *assms*
proof (*induction xss*)
case Nil then show ?case by auto
next
case (Cons xs xss)
have *list-succ xs permutes (set xs)*
using *Cons by (intro list-succ-permutes) (simp add: distincts-def)*
moreover
have *lists-succ xss permutes (∪ ys ∈ set xss. set ys)*
using *Cons by (auto simp: Cons distincts-def)*
ultimately show *lists-succ (xs # xss) permutes (∪ ys ∈ set (xs # xss). set ys)*
using *Cons by (auto simp: lists-succ-Cons-pf intro: permutes-compose permutes-subset)*
qed

lemma *bij-lists-succ: distincts xss ⇒ bij (lists-succ xss)*
by (*induct xss*) (*auto simp: lists-succ-simps-pf bij-comp bij-list-succ distincts-Cons*)

lemma *lists-succ-snoc: lists-succ (xss @ [xs]) = lists-succ xss o list-succ xs*
by (*induct xss*) *auto*

lemma *inv-lists-succ-eq*:
assumes *distincts xss*
shows *inv (lists-succ xss) = lists-succ (rev (map rev xss))*
proof –
have **: ∧f g. inv (λb. f (g b)) = inv (f o g)* **by** (*simp add: o-def*)
have ***:* *lists-succ [] = id* **by** *auto*
show *?thesis*
using *assms by (induct xss) (auto simp: * ** lists-succ-snoc lists-succ-Cons-pf o-inv-distrib inv-list-succ-eq distincts-Cons bij-list-succ bij-lists-succ)*
qed

lemma *lists-succ-remove1*:
assumes *distincts xss xs ∈ set xss*
shows *lists-succ (xs # remove1 xs xss) = lists-succ xss*
using *assms*
proof (*induct xss*)
case Nil then show ?case by simp
next
case (Cons ys xss)
show *?case*

```

proof cases
  assume  $xs = ys$  then show ?case by simp
next
  assume  $xs \neq ys$ 
  with Cons.prems have  $inter: set\ xs \cap set\ ys = \{\}$  and  $xs \in set\ xss$ 
  by (auto simp: distincts-Cons)
  have dists:
     $distincts\ (xs \# remove1\ xs\ xss)$ 
     $distincts\ (xs \# ys \# remove1\ xs\ xss)$ 
  using  $\langle distincts\ (ys \# xss) \rangle$   $\langle xs \in set\ xss \rangle$  by (auto simp: distincts-def)

  have  $list-succ\ xs \circ (list-succ\ ys \circ lists-succ\ (remove1\ xs\ xss))$ 
     $= list-succ\ ys \circ (list-succ\ xs \circ lists-succ\ (remove1\ xs\ xss))$ 
  using inter unfolding fun-eq-iff comp-def
  by (subst list-succ-commute) auto
  also have  $\dots = list-succ\ ys \circ (lists-succ\ (xs \# remove1\ xs\ xss))$ 
  using dists by (simp add: lists-succ-Cons-pf distincts-Cons)
  also have  $\dots = list-succ\ ys \circ lists-succ\ xss$ 
  using  $\langle xs \in set\ xss \rangle$   $\langle distincts\ (ys \# xss) \rangle$ 
  by (simp add: distincts-Cons Cons.hyps)
  finally
  show  $lists-succ\ (xs \# remove1\ xs\ (ys \# xss)) = lists-succ\ (ys \# xss)$ 
  using Cons dists by (auto simp: lists-succ-Cons-pf distincts-Cons)
qed
qed

lemma lists-succ-no-order:
  assumes  $distincts\ xss\ distincts\ yss\ set\ xss = set\ yss$ 
  shows  $lists-succ\ xss = lists-succ\ yss$ 
  using assms
proof (induct xss arbitrary: yss)
  case Nil then show ?case by simp
next
  case (Cons xs xss)
  have  $xs \notin set\ xss\ xs \in set\ yss$  using Cons.prems
  by (auto dest: distincts-distinct)
  have  $lists-succ\ xss = lists-succ\ (remove1\ xs\ yss)$ 
  using Cons.prems  $\langle xs \notin \cdot \rangle$ 
  by (intro Cons.hyps) (auto simp add: distincts-Cons distincts-remove1 distincts-distinct)
  then have  $lists-succ\ (xs \# xss) = lists-succ\ (xs \# remove1\ xs\ yss)$ 
  using Cons.prems  $\langle xs \in \cdot \rangle$ 
  by (simp add: lists-succ-Cons-pf distincts-Cons-remove1)
  then show ?case
  using Cons.prems  $\langle xs \in \cdot \rangle$  by (simp add: lists-succ-remove1)
qed

```

5 List Orbits

Computes the orbit of x under f

definition *orbit-list* :: ('a ⇒ 'a) ⇒ 'a ⇒ 'a list **where**
orbit-list f x ≡ *iterate* 0 (*funpow-dist1* f x) f x

partial-function (*tailrec*)

orbit-list-impl :: ('a ⇒ 'a) ⇒ 'a ⇒ 'a list ⇒ 'a ⇒ 'a list

where

orbit-list-impl f s acc x = (*let* $x' = f$ x *in* *if* $x' = s$ *then* *rev* (x # acc) *else*
orbit-list-impl f s (x # acc) x')

context notes [*simp*] = *length-minus-list-mset* **begin**

Computes the list of orbits

fun *orbits-list* :: ('a ⇒ 'a) ⇒ 'a list ⇒ 'a list list **where**

orbits-list f [] = []

| *orbits-list* f (x # xs) =

orbit-list f x # *orbits-list* f (*minus-list-mset* xs (*orbit-list* f x))

fun *orbits-list-impl* :: ('a ⇒ 'a) ⇒ 'a list ⇒ 'a list list **where**

orbits-list-impl f [] = []

| *orbits-list-impl* f (x # xs) =

(*let* $fc =$ *orbit-list-impl* f x [] *in* fc # *orbits-list-impl* f (*minus-list-mset* xs fc))

declare *orbit-list-impl.simps*[*code*]

end

abbreviation *sset* :: 'a list list ⇒ 'a set set **where**

sset xss ≡ *set* ' *set* xss

lemma *iterate-funpow-step*:

assumes f x ≠ y y ∈ *orbit* f x

shows *iterate* 0 (*funpow-dist1* f x) y = x # *iterate* 0 (*funpow-dist1* f (f x)) y
 f (f x)

proof –

from *assms* **have** A : y ∈ *orbit* f (f x) **by** (*simp* *add*: *orbit-step*)

have *iterate* 0 (*funpow-dist1* f x) y = x # *iterate* 1 (*funpow-dist1* f x) y f x
(*is* - = - # *?it*)

unfolding *iterate-def* **by** (*simp* *add*: *upt-rec*)

also **have** *?it* = *map* ($\lambda n. (f$ $\overset{\sim}{\sim}$ $n)$ x) (*map* *Suc* [$0..<$ *funpow-dist* f (f x) y])

unfolding *iterate-def* *map-Suc-upt* **by** *simp*

also **have** $\dots =$ *map* ($\lambda n. (f$ $\overset{\sim}{\sim}$ $n)$ (f x)) [$0..<$ *funpow-dist* f (f x) y]

by (*simp* *add*: *funpow-swap1*)

also **have** $\dots =$ *iterate* 0 (*funpow-dist1* f (f x)) y f (f x)

unfolding *iterate-def*

unfolding *iterate-def* **by** (*simp* *add*: *funpow-dist-step*[*OF* *assms*(1) A])

finally **show** *?thesis* .

qed

lemma *orbit-list-impl-conv*:
assumes $y \in \text{orbit } f \ x$
shows $\text{orbit-list-impl } f \ y \ \text{acc } x = \text{rev } \text{acc } @ \ \text{iterate } 0 \ (\text{funpow-dist1 } f \ x \ y) \ f \ x$
using *assms*
proof (*induct n* \equiv *funpow-dist1* $f \ x \ y$ *arbitrary: x acc*)
case (*Suc* x)

show *?case*
proof *cases*
assume $f \ x = y$
then show *?thesis* **by** (*subst orbit-list-impl.simps*) (*simp add: Let-def iterate-def funpow-dist-0*)
next
assume *not-y* : $f \ x \neq y$

have *y-in-succ*: $y \in \text{orbit } f \ (f \ x)$
by (*intro orbit-step Suc.prem*s *not-y*)

have $\text{orbit-list-impl } f \ y \ \text{acc } x = \text{orbit-list-impl } f \ y \ (x \ \# \ \text{acc}) \ (f \ x)$
using *not-y* **by** (*subst orbit-list-impl.simps*) *simp*
also have $\dots = \text{rev } (x \ \# \ \text{acc}) \ @ \ \text{iterate } 0 \ (\text{funpow-dist1 } f \ (f \ x) \ y) \ f \ (f \ x)$ (*is -*
 $=$ *?rev @ ?it*)
by (*intro Suc funpow-dist-step not-y y-in-succ*)
also have $\dots = \text{rev } \text{acc} \ @ \ \text{iterate } 0 \ (\text{funpow-dist1 } f \ x \ y) \ f \ x$
using *not-y Suc.prem*s **by** (*simp add: iterate-funpow-step*)
finally show *?thesis* .

qed
qed

lemma *orbit-list-conv-impl*:
assumes $x \in \text{orbit } f \ x$
shows $\text{orbit-list } f \ x = \text{orbit-list-impl } f \ x \ [] \ x$
unfolding *orbit-list-impl-conv*[*OF assms*] *orbit-list-def* **by** *simp*

lemma *set-orbit-list*:
assumes $x \in \text{orbit } f \ x$
shows $\text{set } (\text{orbit-list } f \ x) = \text{orbit } f \ x$
by (*simp add: orbit-list-def orbit-conv-funpow-dist1*[*OF assms*] *set-iterate*)

lemma *set-orbit-list'*:
assumes *permutation* f **shows** $\text{set } (\text{orbit-list } f \ x) = \text{orbit } f \ x$
using *assms* **by** (*simp add: permutation-self-in-orbit set-orbit-list*)

lemma *distinct-orbit-list*:
assumes $x \in \text{orbit } f \ x$
shows *distinct* $(\text{orbit-list } f \ x)$
by (*simp del: upt-Suc add: orbit-list-def iterate-def distinct-map inj-on-funpow-dist1*[*OF*])

assms])

lemma *distinct-orbit-list'*:

assumes *permutation f* **shows** *distinct (orbit-list f x)*

using *assms* **by** (*simp add: permutation-self-in-orbit distinct-orbit-list*)

lemma *orbits-list-conv-impl*:

assumes *permutation f*

shows *orbits-list f xs = orbits-list-impl f xs*

proof (*induct length xs arbitrary: xs rule: less-induct*)

case less **show** *?case*

using *assms* **by** (*cases xs (auto simp: assms less less-Suc-eq-le length-minus-list-mset orbit-list-conv-impl permutation-self-in-orbit Let-def)*)

qed

lemma *orbit-list-not-nil[*simp*]*: *orbit-list f x \neq []*

by (*simp add: orbit-list-def*)

lemma *sset-orbits-list*:

assumes *permutation f* **shows** *sset (orbits-list f xs) = (orbit f) ' set xs*

proof (*induct length xs arbitrary: xs rule: less-induct*)

case less

show *?case*

proof (*cases xs*)

case Nil **then show** *?thesis* **by** *simp*

next

case (*Cons x' xs'*)

let *?xs'' = minus-list-mset xs' (orbit-list f x')*

have *A: sset (orbits-list f ?xs'') = orbit f ' set ?xs''*

using *Cons* **by** (*simp add: less-Suc-eq-le length-minus-list-mset less.hyps*)

have *B: set (orbit-list f x') = orbit f x'*

by (*rule set-orbit-list (simp add: permutation-self-in-orbit assms)*)

have *orbit f ' set (minus-list-mset xs' (orbit-list f x')) \subseteq orbit f ' set xs'*

using *minus-list-mset-subset[*of xs'*]* **by** *auto*

moreover

have *orbit f ' set xs' - {orbit f x'} \subseteq (orbit f ' set (minus-list-mset xs' (orbit-list f x')))* (*is ?L \subseteq ?R*)

proof

fix *A* **assume** *A \in ?L*

then obtain *y* **where** *A = orbit f y y \in set xs'* **by** *auto*

have *A \neq orbit f x'* **using** *$\langle A \in ?L \rangle$* **by** *auto*

from *$\langle A = - \rangle \langle A \neq - \rangle$* **have** *y \notin orbit f x'*

by (*meson assms cyclic-on-orbit orbit-cyclic-eq3 permutation-permutes*)

with *$\langle y \in - \rangle$* **have** *y \in set (minus-list-mset xs' (orbit-list f x'))*

using *minus-list-set-subset-minus-list-mset*

by (*fastforce simp: set-orbit-list permutation-self-in-orbit assms*)

then show *A \in ?R* **using** *$\langle A = - \rangle$* **by** *auto*

qed

ultimately
 show *?thesis* by (auto simp: A B Cons)
 qed
 qed

5.1 Relation to *cyclic-on*

lemma *list-succ-orbit-list*:
 assumes $s \in \text{orbit } f \ s \ \wedge \ x. \ x \notin \text{orbit } f \ s \implies f \ x = x$
 shows $\text{list-succ } (\text{orbit-list } f \ s) = f$
proof –
 have $\text{distinct } (\text{orbit-list } f \ s) \ \wedge \ x. \ x \notin \text{set } (\text{orbit-list } f \ s) \implies x = f \ x$
 using *assms* by (simp-all add: *distinct-orbit-list set-orbit-list*)
moreover
 have $\wedge i. i < \text{length } (\text{orbit-list } f \ s) \implies \text{orbit-list } f \ s \ ! \ (\text{Suc } i \ \text{mod } \text{length } (\text{orbit-list } f \ s)) = f \ (\text{orbit-list } f \ s \ ! \ i)$
 using *funpow-dist1-prop*[*OF* $\langle s \in \text{orbit } f \ s \rangle$] by (auto simp: *orbit-list-def funpow-mod-eq*)
 ultimately show *?thesis*
 by (auto simp: *list-succ-def fun-eq-iff*)
 qed

lemma *list-succ-funpow-conv*:
 assumes $A: \text{distinct } xs \ x \in \text{set } xs$
 shows $(\text{list-succ } xs \ \overset{\sim}{\sim} \ n) \ x = xs \ ! \ ((\text{index } xs \ x + n) \ \text{mod } \text{length } xs)$
proof –
 have $xs \neq []$ using *assms* by auto
 then show *?thesis*
 by (induct n) (auto simp: *hd-conv-nth A index-nth-id list-succ-def mod-simps*)
 qed

lemma *orbit-list-succ*:
 assumes $\text{distinct } xs \ x \in \text{set } xs$
 shows $\text{orbit } (\text{list-succ } xs) \ x = \text{set } xs$
proof (*intro set-eqI iffI*)
 fix y assume $y \in \text{orbit } (\text{list-succ } xs) \ x$
 then show $y \in \text{set } xs$
 by (induct (auto simp: *list-succ-in-conv* $\langle x \in \text{set } xs \rangle$)
 next
 fix y assume $y \in \text{set } xs$
moreover
 { fix $i \ j$ have $i < \text{length } xs \implies j < \text{length } xs \implies \exists n. xs \ ! \ j = xs \ ! \ ((i + n) \ \text{mod } \text{length } xs)$
 using *assms* by (auto simp: *exI*[**where** $x=j + (\text{length } xs - i)$])
 }
 ultimately
 show $y \in \text{orbit } (\text{list-succ } xs) \ x$
 using *assms* by (auto simp: *orbit-altdef-permutation permutation-list-succ list-succ-funpow-conv index-nth-id in-set-conv-nth*)

qed

lemma *cyclic-on-list-succ*:

assumes $\text{distinct } xs \text{ } xs \neq []$ **shows** $\text{cyclic-on } (\text{list-succ } xs) \text{ } (\text{set } xs)$
using *assms last-in-set* **by** (*auto simp: cyclic-on-def orbit-list-succ*)

lemma *obtain-orbit-list-func*:

assumes $s \in \text{orbit } f \text{ } s \wedge x. x \notin \text{orbit } f \text{ } s \implies f \text{ } x = x$
obtains xs **where** $f = \text{list-succ } xs \text{ } \text{set } xs = \text{orbit } f \text{ } s \text{ } \text{distinct } xs \text{ } \text{hd } xs = s$

proof –

{ **from** *assms* **have** $f = \text{list-succ } (\text{orbit-list } f \text{ } s)$ **by** (*simp add: list-succ-orbit-list*)

moreover

have $\text{set } (\text{orbit-list } f \text{ } s) = \text{orbit } f \text{ } s \text{ } \text{distinct } (\text{orbit-list } f \text{ } s)$

by (*auto simp: set-orbit-list distinct-orbit-list assms*)

moreover have $\text{hd } (\text{orbit-list } f \text{ } s) = s$

by (*simp add: orbit-list-def iterate-def hd-map del: upt-Suc*)

ultimately have $\exists xs. f = \text{list-succ } xs \wedge \text{set } xs = \text{orbit } f \text{ } s \wedge \text{distinct } xs \wedge \text{hd } xs = s$ **by** *blast*

} **then show** *?thesis* **by** (*metis that*)

qed

lemma *cyclic-on-obtain-list-succ*:

assumes $\text{cyclic-on } f \text{ } S \wedge x. x \notin S \implies f \text{ } x = x$

obtains xs **where** $f = \text{list-succ } xs \text{ } \text{set } xs = S \text{ } \text{distinct } xs$

proof –

from *assms* **obtain** s **where** $s: s \in \text{orbit } f \text{ } s \wedge x. x \notin \text{orbit } f \text{ } s \implies f \text{ } x = x \text{ } S = \text{orbit } f \text{ } s$

by (*auto simp: cyclic-on-def*)

then show *?thesis* **by** (*metis that obtain-orbit-list-func*)

qed

lemma *cyclic-on-obtain-list-succ'*:

assumes $\text{cyclic-on } f \text{ } S \text{ } f \text{ permutes } S$

obtains xs **where** $f = \text{list-succ } xs \text{ } \text{set } xs = S \text{ } \text{distinct } xs$

using *assms* **unfolding** *permutes-def* **by** (*metis cyclic-on-obtain-list-succ*)

lemma *list-succ-unique*:

assumes $s \in \text{orbit } f \text{ } s \wedge x. x \notin \text{orbit } f \text{ } s \implies f \text{ } x = x$

shows $\exists! xs. f = \text{list-succ } xs \wedge \text{distinct } xs \wedge \text{hd } xs = s \wedge \text{set } xs = \text{orbit } f \text{ } s$

proof –

from *assms* **obtain** xs **where** $xs: f = \text{list-succ } xs \text{ } \text{distinct } xs \text{ } \text{hd } xs = s \text{ } \text{set } xs = \text{orbit } f \text{ } s$

by (*rule obtain-orbit-list-func*)

moreover

{ **fix** zs

assume $A: f = \text{list-succ } zs \text{ } \text{distinct } zs \text{ } \text{hd } zs = s \text{ } \text{set } zs = \text{orbit } f \text{ } s$

then have $zs \neq []$ **using** $\langle s \in \text{orbit } f \text{ } s \rangle$ **by** *auto*

from $\langle \text{distinct } xs \rangle \langle \text{distinct } zs \rangle \langle \text{set } xs = \text{orbit } f \text{ } s \rangle \langle \text{set } zs = \text{orbit } f \text{ } s \rangle$

have $\text{len: length } xs = \text{length } zs$ **by** (*metis distinct-card*)

```

{ fix n assume n < length xs
  then have zs ! n = xs ! n
  proof (induct n)
    case 0 with A xs ⟨zs ≠ []⟩ show ?case by (simp add: hd-conv-nth
nth-rotate-conv-nth)
  next
    case (Suc n)
    then have list-succ zs (zs ! n) = list-succ xs (xs ! n)
      using ⟨f = list-succ xs⟩ ⟨f = list-succ zs⟩ by simp
    with ⟨Suc n < -⟩ show ?case
      by (simp add: list-succ-nth len ⟨distinct xs⟩ ⟨distinct zs⟩)
    qed }
  then have zs = xs by (metis len nth-equalityI) }
ultimately show ?thesis by metis
qed

```

lemma *distincts-orbits-list*:

```

assumes distinct as permutation f
shows distincts (orbits-list f as)
using assms(1)
proof (induct length as arbitrary: as rule: less-induct)
  case less
  show ?case
  proof (cases as)
    case Nil then show ?thesis by simp
  next
    case (Cons a as')
    let ?as' = fold remove1 (orbit-list f a) as'
    from Cons less.prem1 have A: distincts (orbits-list f (minus-list-mset as'
(orbit-list f a)))
      by (intro less) (auto simp: distinct-minus-list-mset length-minus-list-mset
less-Suc-eq-le)

    have B: set (orbit-list f a) ∩ ∪ (sset (orbits-list f (minus-list-mset as' (orbit-list
f a)))) = {}
    proof -
      have orbit f a ∩ set (minus-list-mset as' (orbit-list f a)) = {}
      using assms less.prem1 Cons by (simp add: set-minus-list-mset-distinct
set-orbit-list')
      then have orbit f a ∩ ∪ (orbit f ' set (minus-list-mset as' (orbit-list f a))) =
{}
      by auto (metis assms(2) cyclic-on-orbit disjoint-iff-not-equal permuta-
tion-self-in-orbit[OF assms(2)] orbit-cyclic-eq3 permutation-permutes)
      then show ?thesis using assms
      by (auto simp: set-orbit-list' sset-orbits-list disjoint-iff-not-equal)
    qed
  show ?thesis
  using A B assms by (auto simp: distincts-Cons Cons distinct-orbit-list')

```

qed
qed

lemma *cyclic-on-lists-succ'*:

assumes *distincts xss*

shows $A \in \text{sset } xss \implies \text{cyclic-on } (\text{lists-succ } xss) A$

using *assms*

proof (*induction xss arbitrary: A*)

case *Nil* **then show** *?case* **by** *auto*

next

case (*Cons xs xss A*)

then have *inter: set xs \cap ($\bigcup_{ys \in \text{set } xss. \text{set } ys}$) = {}* **by** (*auto simp: distincts-Cons*)

note *pcp[OF - - inter] = permutes-comp-preserves-cyclic1 permutes-comp-preserves-cyclic2*

from *Cons* **show** *cyclic-on (lists-succ (xs # xss)) A*

by (*cases A = set xs*)

(*auto intro: pcp simp: cyclic-on-list-succ list-succ-permutes lists-succ-permutes lists-succ-Cons-pf distincts-Cons*)

qed

lemma *cyclic-on-lists-succ*:

assumes *distincts xss*

shows $\bigwedge xs. xs \in \text{set } xss \implies \text{cyclic-on } (\text{lists-succ } xss) (\text{set } xs)$

using *assms* **by** (*auto intro: cyclic-on-lists-succ'*)

lemma *permutes-as-lists-succ*:

assumes *distincts xss*

assumes *ls-eq: $\bigwedge xs. xs \in \text{set } xss \implies \text{list-succ } xs = \text{perm-restrict } f (\text{set } xs)$*

assumes *f permutes ($\bigcup (\text{sset } xss)$)*

shows $f = \text{lists-succ } xss$

using *assms*

proof (*induct xss arbitrary: f*)

case *Nil* **then show** *?case* **by** *simp*

next

case (*Cons xs xss*)

let *?sets = $\lambda xss. \bigcup_{ys \in \text{set } xss. \text{set } ys}$*

have *xs: distinct xs xs \neq []* **using** *Cons* **by** (*auto simp: distincts-Cons*)

have *f-xs: perm-restrict f (set xs) = list-succ xs*

using *Cons* **by** *simp*

have *co-xs: cyclic-on (perm-restrict f (set xs)) (set xs)*

unfolding *f-xs* **using** *xs* **by** (*rule cyclic-on-list-succ*)

have *perm-xs: perm-restrict f (set xs) permutes set xs*

unfolding *f-xs* **using** *<distinct xs>* **by** (*rule list-succ-permutes*)

have *perm-xss*: *perm-restrict* *f* (?sets *xss*) *permutes* (?sets *xss*)
proof –
have *perm-restrict* *f* (?sets (*xs* # *xss*) – *set xs*) *permutes* (?sets (*xs* # *xss*) –
set xs)
using *Cons co-xs* **by** (*intro perm-restrict-diff-cyclic*) (*auto simp: cyclic-on-perm-restrict*)
also have ?sets (*xs* # *xss*) – *set xs* = ?sets *xss*
using *Cons* **by** (*auto simp: distincts-Cons*)
finally show ?thesis .
qed

have *f-xss*: *perm-restrict* *f* (?sets *xss*) = *lists-succ* *xss*
proof –
have *: $\bigwedge xs. xs \in \text{set } xss \implies ((\bigcup x \in \text{set } xss. \text{set } x) \cap \text{set } xs) = \text{set } xs$
by *blast*
with *perm-xss* *Cons.prem*s **show** ?thesis
by (*intro Cons.hyps*) (*auto simp: distincts-Cons perm-restrict-perm-restrict **)
qed

from *Cons.prem*s **show** *f* = *lists-succ* (*xs* # *xss*)
by (*simp add: lists-succ-Cons-pf distincts-Cons f-xss[symmetric]*
perm-restrict-union perm-xs perm-xss)
qed

lemma *cyclic-on-obtain-lists-succ*:

assumes
permutes: *f permutes* *S* **and**
S: $S = \bigcup (\text{sset } \text{css})$ **and**
dists: *distincts* *css* **and**
cyclic: $\bigwedge cs. cs \in \text{set } \text{css} \implies \text{cyclic-on } f (\text{set } cs)$
obtains *xss* **where** *f* = *lists-succ* *xss* *distincts* *xss* *map set xss* = *map set css*
map hd xss = *map hd css*
proof –
let ?*fc* = $\lambda cs. \text{perm-restrict } f (\text{set } cs)$
define *some-list* **where** *some-list* *cs* = (*SOME* *xs*. ?*fc* *cs* = *list-succ* *xs* \wedge *set xs*
= *set cs* \wedge *distinct* *xs* \wedge *hd xs* = *hd cs*) **for** *cs*
{ fix *cs* **assume** *cs* \in *set css*
then have *cyclic-on* (?*fc* *cs*) (*set cs*) $\bigwedge x. x \notin \text{set } cs \implies ?fc$ *cs* *x* = *x* *hd cs* \in
set cs
using *cyclic* *dists* **by** (*auto simp add: cyclic-on-perm-restrict perm-restrict-def*
distincts-def)
then have *hd cs* \in *orbit* (?*fc* *cs*) (*hd cs*) $\bigwedge x. x \notin \text{orbit}$ (?*fc* *cs*) (*hd cs*) \implies
?*fc* *cs* *x* = *x* *hd cs* \in *set cs* *set cs* = *orbit* (?*fc* *cs*) (*hd cs*)
by (*auto simp: cyclic-on-alldef*)
then have $\exists xs. ?fc$ *cs* = *list-succ* *xs* \wedge *set xs* = *set cs* \wedge *distinct* *xs* \wedge *hd xs* =
hd cs
by (*metis obtain-orbit-list-func*)
then have ?*fc* *cs* = *list-succ* (*some-list* *cs*) \wedge *set* (*some-list* *cs*) = *set cs* \wedge
distinct (*some-list* *cs*) \wedge *hd* (*some-list* *cs*) = *hd cs*
unfolding *some-list-def* **by** (*rule someI-ex*)

```

    then have ?fc cs = list-succ (some-list cs) set (some-list cs) = set cs distinct
  (some-list cs) hd (some-list cs) = hd cs
    by auto
  } note sl-cs = this

  have  $\bigwedge cs. cs \in \text{set } css \implies cs \neq []$  using dists by (auto simp: distincts-def)
  then have some-list-ne:  $\bigwedge cs. cs \in \text{set } css \implies \text{some-list } cs \neq []$ 
    by (metis set-empty sl-cs(2))

  have set: map set (map some-list css) = map set css map hd (map some-list css)
= map hd css
    using sl-cs(2,4) by (auto simp add: map-idI)

  have distincts: distincts (map some-list css)
  proof -
    have c-dist:  $\bigwedge xs\ ys. \llbracket xs \in \text{set } css; ys \in \text{set } css; xs \neq ys \rrbracket \implies \text{set } xs \cap \text{set } ys = \{\}$ 
      using dists by (auto simp: distincts-def)

    have distinct (map some-list css)
    proof -
      have inj-on some-list (set css)
        using sl-cs(2) c-dist by (intro inj-onI) (metis inf.idem set-empty)
      with ‹distincts css› show ?thesis
        by (auto simp: distincts-distinct distinct-map)
    qed
    moreover
    have  $\forall xs \in \text{set } (map \text{some-list } css). \text{distinct } xs \wedge xs \neq []$ 
      using sl-cs(3) some-list-ne by auto
    moreover
    from c-dist have  $(\forall xs \in \text{set } (map \text{some-list } css). \forall ys \in \text{set } (map \text{some-list } css).$ 
 $xs \neq ys \longrightarrow \text{set } xs \cap \text{set } ys = \{\})$ 
      using sl-cs(2) by auto
    ultimately
    show ?thesis by (simp add: distincts-def)
  qed

  have f: f = lists-succ (map some-list css)
    using distincts
  proof (rule permutes-as-lists-succ)
    fix xs assume xs  $\in$  set (map some-list css)
    then show list-succ xs = perm-restrict f (set xs)
      using sl-cs(1) sl-cs(2) by auto
  next
    have S =  $(\bigcup xs \in \text{set } (map \text{some-list } css). \text{set } xs)$ 
      using S sl-cs(2) by auto
    with permutes show f permutes  $\bigcup (sset (map \text{some-list } css))$ 
      by simp
  qed

```

```

  from f distincts set show ?thesis ..
qed

```

5.2 Permutations of a List

```

lemma length-remove1-less:
  assumes x ∈ set xs shows length (remove1 x xs) < length xs
proof -
  from assms have 0 < length xs by auto
  with assms show ?thesis by (auto simp: length-remove1)
qed
context notes [simp] = length-remove1-less begin
fun permutations :: 'a list ⇒ 'a list list where
  permutations-Nil: permutations [] = [[]]
| permutations-Cons:
  permutations xs = [y # ys. y <- xs, ys <- permutations (remove1 y xs)]
end

declare permutations-Cons[simp del]

```

The function above returns all permutations of a list. The function below computes only those which yield distinct cyclic permutation functions (cf. *list-succ*).

```

fun cyc-permutations :: 'a list ⇒ 'a list list where
  cyc-permutations [] = [[]]
| cyc-permutations (x # xs) = map (Cons x) (permutations xs)

```

```

lemma nil-in-permutations[simp]: [] ∈ set (permutations xs) ⟷ xs = []
  by (induct xs) (auto simp: permutations-Cons)

```

```

lemma permutations-not-nil:
  assumes xs ≠ []
  shows permutations xs = concat (map (λx. map ((#) x) (permutations (remove1 x xs)))) xs)
  using assms by (cases xs) (auto simp: permutations-Cons)

```

```

lemma set-permutations-step:
  assumes xs ≠ []
  shows set (permutations xs) = (⋃ x ∈ set xs. Cons x ` set (permutations (remove1 x xs)))
  using assms by (cases xs) (auto simp: permutations-Cons)

```

```

lemma in-set-permutations:
  assumes distinct xs
  shows ys ∈ set (permutations xs) ⟷ distinct ys ∧ set xs = set ys (is ?L xs ys
  ⟷ ?R xs ys)
  using assms

```

```

proof (induct length xs arbitrary: xs ys)
  case 0 then show ?case by auto
next
  case (Suc n)
  then have xs ≠ [] by auto

  show ?case
  proof
    assume ?L xs ys
    then obtain y ys' where ys = y # ys' y ∈ set xs ys' ∈ set (permutations
(remove1 (hd ys) xs))
    using ⟨xs ≠ []⟩ by (auto simp: permutations-not-nil)
    moreover
    then have ?R (remove1 y xs) ys'
    using Suc.prem1 Suc.hyps(2) by (intro Suc.hyps(1)[THEN iffD1]) (auto simp:
length-remove1)
    ultimately show ?R xs ys
    using Suc by auto
  next
    assume ?R xs ys
    with ⟨xs ≠ []⟩ obtain y ys' where ys = y # ys' y ∈ set xs by (cases ys) auto
    moreover
    then have ys' ∈ set (permutations (remove1 y xs))
    using Suc ⟨?R xs ys⟩ by (intro Suc.hyps(1)[THEN iffD2]) (auto simp:
length-remove1)
    ultimately
    show ?L xs ys
    using ⟨xs ≠ []⟩ by (auto simp: permutations-not-nil)
  qed
qed

```

lemma in-set-cyc-permutations:

```

assumes distinct xs
shows ys ∈ set (cyc-permutations xs) ⟷ distinct ys ∧ set xs = set ys ∧ hd ys
= hd xs (is ?L xs ys ⟷ ?R xs ys)
proof (cases xs)
  case (Cons x xs) with assms show ?thesis
  by (cases ys) (auto simp: in-set-permutations intro!: imageI)
qed auto

```

lemma in-set-cyc-permutations-obtain:

```

assumes distinct xs distinct ys set xs = set ys
obtains n where rotate n ys ∈ set (cyc-permutations xs)
proof (cases xs)
  case Nil with assms have rotate 0 ys ∈ set (cyc-permutations xs) by auto
then show ?thesis ..
next
  case (Cons x xs')
  let ?ys' = rotate (index ys x) ys

```

```

have  $ys \neq [] \ x \in \text{set } ys$ 
  using Cons assms by auto
then have  $\text{distinct } ?ys' \wedge \text{set } xs = \text{set } ?ys' \wedge \text{hd } ?ys' = \text{hd } xs$ 
  using assms Cons by (auto simp add: hd-rotate-conv-nth)
with  $\langle \text{distinct } xs \rangle$  have  $?ys' \in \text{set } (\text{cyc-permutations } xs)$ 
  by (rule in-set-cyc-permutations[THEN iffD2])
then show ?thesis ..
qed

```

lemma *list-succ-set-cyc-permutations:*

```

assumes  $\text{distinct } xs \ xs \neq []$ 
shows  $\text{list-succ } \langle \text{set } (\text{cyc-permutations } xs) = \{f. f \text{ permutes set } xs \wedge \text{cyclic-on } f$ 
 $(\text{set } xs)\} \text{ (is } ?L = ?R)$ 
proof (intro set-eqI iffI)
  fix  $f$  assume  $f \in ?L$ 
  moreover have  $\bigwedge ys. \text{set } xs = \text{set } ys \implies xs \neq [] \implies ys \neq []$  by auto
  ultimately show  $f \in ?R$ 
  using assms by (auto simp: in-set-cyc-permutations list-succ-permutes cyclic-on-list-succ)
next
  fix  $f$  assume  $f \in ?R$ 
  then obtain  $ys$  where  $ys: \text{list-succ } ys = f \ \text{distinct } ys \ \text{set } ys = \text{set } xs$ 
    by (auto elim: cyclic-on-obtain-list-succ')
  moreover
  with  $\langle \text{distinct } xs \rangle$  obtain  $n$  where  $\text{rotate } n \ ys \in \text{set } (\text{cyc-permutations } xs)$ 
    by (auto elim: in-set-cyc-permutations-obtain)
  then have  $\text{list-succ } (\text{rotate } n \ ys) \in ?L$  by simp
  ultimately
  show  $f \in ?L$  by simp
qed

```

5.3 Enumerating Permutations from List Orbits

definition *cyc-permutationss* :: $'a \text{ list list} \Rightarrow 'a \text{ list list list}$ **where**
cyc-permutationss = *product-lists o map cyc-permutations*

lemma *cyc-permutationss-Nil[simp]: cyc-permutationss [] = [[]]*
 by (*auto simp: cyc-permutationss-def*)

lemma *in-set-cyc-permutationss:*

```

assumes distincts  $xss$ 
shows  $yss \in \text{set } (\text{cyc-permutationss } xss) \iff \text{distincts } yss \wedge \text{map set } xss = \text{map}$ 
 $\text{set } yss \wedge \text{map hd } xss = \text{map hd } yss$ 
proof -
  { assume  $A: \text{list-all2 } (\lambda x \ ys. x \in \text{set } ys) \ yss \ (\text{map } \text{cyc-permutations } xss)$ 
    then have  $\text{length } yss = \text{length } xss$  by (auto simp: list-all2-lengthD)
    then have  $\bigcup (\text{sset } xss) = \bigcup (\text{sset } yss) \ \text{distincts } yss \ \text{map set } xss = \text{map set } yss$ 
       $\text{map hd } xss = \text{map hd } yss$ 
      using  $A$  assms
      by (induct yss xss rule: list-induct2) (auto simp: distincts-Cons in-set-cyc-permutations)
  }

```

```

} note X = this
{ assume A: distincts yss map set xss = map set yss map hd xss = map hd yss
  then have length yss = length xss by (auto dest: map-eq-imp-length-eq)
  then have list-all2 (λx ys. x ∈ set ys) yss (map cyc-permutations xss)
    using A assms
    by (induct yss xss rule: list-induct2) (auto simp: distincts-Cons in-set-cyc-permutations)
} note Y = this
show ?thesis
  unfolding cyc-permutationss-def
  by (auto simp: product-lists-set intro: X Y)
qed

```

```

lemma lists-succ-set-cyc-permutationss:
  assumes distincts xss
  shows lists-succ ' set (cyc-permutationss xss) = {f. f permutes  $\bigcup$ (sset xss)  $\wedge$ 
    ( $\forall c \in$  sset xss. cyclic-on f c)} (is ?L = ?R)
  using assms
proof (intro set-eqI iffI)
  fix f assume f ∈ ?L
  then obtain yss where yss ∈ set (cyc-permutationss xss) f = lists-succ yss by
    (rule imageE)
  moreover
  from ⟨yss ∈  $\rightarrow$  assms have set (map set xss) = set (map set yss)
    by (auto simp: in-set-cyc-permutationss)
  then have sset xss = sset yss by simp
  ultimately
  show f ∈ ?R
    using assms
  by (auto simp: in-set-cyc-permutationss cyclic-on-lists-succ') (metis lists-succ-permutes)
next
  fix f assume f ∈ ?R
  then have f permutes  $\bigcup$ (sset xss)  $\wedge$  cs. cs ∈ set xss  $\implies$  cyclic-on f (set cs)
    by auto
  from this(1) refl assms this(2)
  obtain yss where f = lists-succ yss distincts yss map set yss = map set xss map
    hd yss = map hd xss
  by (rule cyclic-on-obtain-lists-succ)
  with assms show f ∈ ?L by (auto intro!: imageI simp: in-set-cyc-permutationss)
qed

```

5.4 Lists of Permutations

definition *permutationss* :: 'a list list \Rightarrow 'a list list list **where**
permutationss = product-lists o map permutations

lemma *permutationss-Nil*[simp]: *permutationss* [] = [[]]
 by (auto simp: permutationss-def)

lemma *permutationss-Cons*:

$permutationss (xs \# xss) = concat (map (\lambda ys. map (Cons ys) (permutationss xss)) (permutations xs))$

by (*auto simp: permutationss-def*)

lemma *in-set-permutationss*:

assumes *distincts xss*

shows $yss \in set (permutationss xss) \longleftrightarrow distincts yss \wedge map set xss = map set yss$

proof –

{ **assume** *A*: *list-all2* ($\lambda x ys. x \in set ys$) *yss* (*map permutationss xss*)

then have $length yss = length xss$ **by** (*auto simp: list-all2-lengthD*)

then have $\bigcup (sset xss) = \bigcup (sset yss)$ *distincts yss* $map set xss = map set yss$

using *A* *assms*

by (*induct yss xss rule: list-induct2*) (*auto simp: distincts-Cons in-set-permutationss*)

} **note** *X = this*

{ **assume** *A*: *distincts yss* $map set xss = map set yss$

then have $length yss = length xss$ **by** (*auto dest: map-eq-imp-length-eq*)

then have *list-all2* ($\lambda x ys. x \in set ys$) *yss* (*map permutationss xss*)

using *A* *assms*

by (*induct yss xss rule: list-induct2*) (*auto simp: in-set-permutationss distincts-Cons*)

} **note** *Y = this*

show *?thesis*

unfolding *permutationss-def*

by (*auto simp: product-lists-set intro: X Y*)

qed

lemma *set-permutationss*:

assumes *distincts xss*

shows $set (permutationss xss) = \{yss. distincts yss \wedge map set xss = map set yss\}$

using *in-set-permutationss[OF assms]* **by** *blast*

lemma *permutationss-complete*:

assumes *distincts xss* *distincts yss* $xss \neq []$

and $set ' set xss = set ' set yss$

shows $set yss \in set ' set (permutationss xss)$

proof –

have $length xss = length yss$

using *assms* **by** (*simp add: length-distincts*)

from $\langle sset xss = - \rangle$

have $\exists yss'. set yss' = set yss \wedge map set yss' = map set xss$

using *assms(1-2)*

proof (*induct xss arbitrary: yss*)

case *Nil* **then show** *?case* **by** *simp*

next

case (*Cons xs xss*)

from $\langle sset (xs \# xss) = sset yss \rangle$

obtain *ys* **where** $ys \in set yss$ $set ys = set xs$

```

    by auto (metis imageE insertI1)
  with ‹distincts yss› have set ys  $\notin$  sset (remove1 ys yss)
    by (fastforce simp: distincts-def)
  moreover
  from ‹distincts (xs # xss)› have set xs  $\notin$  sset xss
    by (fastforce simp: distincts-def)
  ultimately have sset xss = sset (remove1 ys yss)
    using ‹distincts yss› ‹sset (xs # xss) = sset yss›
    apply (auto simp: distincts-distinct ‹set ys = set xs›[symmetric])
  apply (smt Diff-insert-absorb ‹ys  $\in$  set yss› image-insert insert-Diff rev-image-eqI)
    by (metis ‹ys  $\in$  set yss› image-eqI insert-Diff insert-iff)
  then obtain yss' where set yss' = set (remove1 ys yss)  $\wedge$  map set yss' = map
set xss
    using Cons by atomize-elim (auto simp: distincts-Cons distincts-remove1)
  then have set (ys # yss') = set yss  $\wedge$  map set (ys # yss') = map set (xs #
xss)
    using ys set-remove1-eq ‹distincts yss› by (auto simp: distincts-distinct)
  then show ?case ..
qed
then obtain yss' where set yss' = set yss map set yss' = map set xss by blast
then have distincts yss' using ‹distincts xss› ‹distincts yss›
  unfolding distincts-def
  by simp-all (metis ‹length xss = length yss› card-distinct distinct-card length-map)
then have set yss'  $\in$  set ' set (permutations xss)
  using ‹distincts xss› ‹map set yss' = -›
  by (auto simp: set-permutations)
then show ?thesis using ‹set yss' = -› by auto
qed

lemma permutations-complete:
  assumes distinct xs distinct ys set xs = set ys
  shows ys  $\in$  set (permutations xs)
  using assms
proof (induct length xs arbitrary: xs ys)
  case 0 then show ?case by simp
next
  case (Suc n)
  from Suc.hyps have xs  $\neq$  [] by auto
  then obtain y ys' where [simp]: ys = y # ys' y  $\in$  set xs using Suc.prem by
(cases ys) auto
  have ys'  $\in$  set (permutations (remove1 y xs))
    using Suc.prem ‹Suc n = -› by (intro Suc.hyps) (simp-all add: length-remove1
)
  then show ?case using ‹xs  $\neq$  []› by (auto simp: set-permutations-step)
qed

end
theory Digraph-Map-Impl

```

```

imports
  Graph-Genus
  Executable-Permutations
  Transitive-Closure.Transitive-Closure-Impl
begin

```

6 Enumerating Maps

definition *grouped-by-fst* :: ('a × 'b) list ⇒ ('a × 'b) list list **where**
grouped-by-fst xs = map (λu. filter (λx. fst x = u) xs) (remdups (map fst xs))

fun *grouped-out-arcs* :: 'a list × ('a × 'a) list ⇒ ('a × 'a) list list **where**
grouped-out-arcs (vs,as) = grouped-by-fst as

definition *all-maps-list* :: ('a list × ('a × 'a) list) ⇒ ('a × 'a) list list list **where**
all-maps-list G-list = (cyc-permutationss o grouped-out-arcs) G-list

definition *list-digraph-ext* ext G-list ≡ (| pverts = set (fst G-list), parcs = set (snd G-list), ... = ext |)

abbreviation *list-digraph* ≡ *list-digraph-ext* ()

code-datatype *list-digraph-ext*

lemma *list-digraph-simps*:
pverts (list-digraph G-list) = set (fst G-list)
parcs (list-digraph G-list) = set (snd G-list)
by (auto simp: list-digraph-ext-def)

lemma *union-grouped-by-fst*:
 $(\bigcup xs \in \text{set } (\text{grouped-by-fst } ys). \text{set } xs) = \text{set } ys$
by (auto simp: grouped-by-fst-def)

lemma *union-grouped-out-arcs*:
 $(\bigcup xs \in \text{set } (\text{grouped-out-arcs } G\text{-list}). \text{set } xs) = \text{set } (\text{snd } G\text{-list})$
by (cases G-list) (simp add: union-grouped-by-fst)

lemma *nil-not-in-grouped-out-arcs*: [] ∉ set (grouped-out-arcs G-list)
apply (cases G-list) **apply** (auto simp: grouped-by-fst-def)
by (metis (mono-tags) filter-empty-conv fst-conv)

lemma *set-grouped-out-arcs*:
assumes pair-wf-digraph (list-digraph G-list)
shows set ' set (grouped-out-arcs G-list) = {out-arcs (list-digraph G-list) v | v.
v ∈ pverts (list-digraph G-list) ∧ out-arcs (list-digraph G-list) v ≠ {}} }
(is ?L = ?R)

proof –

```

interpret pair-wf-digraph list-digraph G-list by fact
define vs where vs = remdups (map fst (snd G-list))
have set vs = {v. out-arcs (list-digraph G-list) v ≠ {}}
  by (auto simp: out-arcs-def list-digraph-ext-def vs-def intro: rev-image-eqI )
then have vs: set vs = {v ∈ pverts (list-digraph G-list). out-arcs (list-digraph
G-list) v ≠ {}}
  by (auto dest: in-arcsD1)
have goa: grouped-out-arcs G-list = map (λu. filter (λx. fst x = u) (snd G-list))
vs
  by (cases G-list) (auto simp: grouped-by-fst-def vs-def)
have filter: set ∘ (λu. filter (λx. fst x = u) (snd G-list)) = out-arcs (list-digraph
G-list)
  by (rule ext) (auto simp: list-digraph-ext-def)

have set (map set (grouped-out-arcs G-list)) = ?R by (auto simp add: goa filter
vs)
then show ?thesis by simp
qed

```

```

lemma distincts-grouped-by-fst:
  assumes distinct xs shows distincts (grouped-by-fst xs)
proof –
  have list-eq-setD:  $\bigwedge xs\ ys. xs = ys \implies set\ xs = set\ ys$  by auto
  have inj: inj-on (λu. filter (λx. fst x = u) xs) (fst ‘ set xs)
    by (rule inj-onI) (drule list-eq-setD, auto)
  with assms show ?thesis
  by (auto simp: grouped-by-fst-def distincts-def distinct-map filter-empty-conv)
qed

```

```

lemma distincts-grouped-arcs:
  assumes distinct (snd G-list) shows distincts (grouped-out-arcs G-list)
  using assms by (cases G-list) (simp add: distincts-grouped-by-fst)

```

```

lemma distincts-in-all-maps-list:
  distinct (snd X)  $\implies xss \in set\ (all-maps-list\ X) \implies distincts\ xss$ 
  by (simp add: all-maps-list-def distincts-grouped-arcs in-set-cyc-permutationss)

```

```

definition to-map :: ('a × 'a) set  $\Rightarrow$  ('a × 'a  $\Rightarrow$  'a × 'a)  $\Rightarrow$  ('a × 'a) pre-map
where
  to-map A f = (| edge-rev = swap-in A, edge-succ = f |)

```

```

abbreviation to-map' as xss  $\equiv$  to-map (set as) (lists-succ xss)

```

```

definition all-maps :: 'a pair-pre-digraph  $\Rightarrow$  ('a × 'a) pre-map set where
  all-maps G  $\equiv$  to-map (arcs G) ‘ {f. f permutes arcs G  $\wedge$  ( $\forall v \in verts\ G. out-arcs\ G\ v \neq \{\}$ )  $\longrightarrow$  cyclic-on f (out-arcs G v)}

```

definition *maps-all-maps-list* :: ('a list × ('a × 'a) list) ⇒ ('a × 'a) pre-map list
where
maps-all-maps-list G-list = map (to-map (set (snd G-list)) o lists-succ) (all-maps-list G-list)

lemma (in *pair-graph*) *all-maps-correct*:
shows *all-maps* G = {M. *digraph-map* G M}
proof (intro *set-eqI iffI*)
fix M **assume** A: M ∈ *all-maps* G
then have [*simp*]: *edge-rev* M = *swap-in* (arcs G) *edge-succ* M *permutes* *parcs* G
by (auto *simp: all-maps-def to-map-def*)

have *digraph-map* G M
proof (rule *digraph-mapI*)
fix a **assume** a ∉ *parcs* G **then show** *edge-rev* M a = a **by** (auto *simp: swap-in-def*)
next
fix a **assume** a ∈ *parcs* G
then show *edge-rev* M (edge-rev M a) = a *fst* (edge-rev M a) = *snd* a *edge-rev* M a ≠ a
by (*case-tac* [!] a) (auto *intro: arcs-symmetric simp: swap-in-def dest: no-loops*)
next
show *edge-succ* M *permutes* *parcs* G **by** *simp*
next
fix v **assume** v ∈ *pverts* G *out-arcs* (*with-proj* G) v ≠ {}
then show *cyclic-on* (*edge-succ* M) (*out-arcs* (*with-proj* G) v)
using A **unfolding** *all-maps-def* **by** (auto *simp: to-map-def*)
qed
then show M ∈ {M. *digraph-map* G M} **by** *simp*
next
fix M **assume** A: M ∈ {M. *digraph-map* G M}
then interpret M: *digraph-map* G M **by** *simp*
from A **have** ∧x. *fst* (edge-rev M x) = *fst* (*swap-in* (arcs G) x)
∧x. *snd* (edge-rev M x) = *snd* (*swap-in* (arcs G) x)
using M.*tail-arev* M.*head-arev* **by** (auto *simp: fun-eq-iff swap-in-def M.arev-eq*)
then have *edge-rev* M = *swap-in* (arcs G)
by (*metis prod.collapse fun-eq-iff*)
then show M ∈ *all-maps* G
using M.*edge-succ-permutes* M.*edge-succ-cyclic*
unfolding *all-maps-def*
by (auto *simp: to-map-def intro!: image-eqI*[**where** x=*edge-succ* M])
qed

lemma *set-maps-all-maps-list*:
assumes *pair-wf-digraph* (*list-digraph* G-list) *distinct* (snd G-list)
shows *all-maps* (*list-digraph* G-list) = *set* (*maps-all-maps-list* G-list)

proof –

let $?G = \text{list-digraph } G\text{-list}$

```

{ fix  $f$ 
  have  $(\forall x \in \text{set } (\text{grouped-out-arcs } G\text{-list}). \text{cyclic-on } f (\text{set } x))$ 
     $\longleftrightarrow (\forall x \in \text{set } ' \text{set } (\text{grouped-out-arcs } G\text{-list}). \text{cyclic-on } f x) (\text{is } ?all1 = -)$ 
  by simp
  also have  $\dots \longleftrightarrow (\forall v \in \text{pverts } ?G. \text{out-arcs } ?G v \neq \{\}) \longrightarrow \text{cyclic-on } f (\text{out-arcs } ?G v)$ 
  (is - = ?all2)
  using  $\text{assms}$  by  $(\text{auto simp: set-grouped-out-arcs})$ 
  finally have  $?all1 = ?all2$  .
} note  $\text{all-eq} = \text{this}$ 

```

```

have  $\text{lists-succ } ' \text{set } (\text{all-maps-list } G\text{-list})$ 
   $= \{f. f \text{ permutes arcs } ?G \wedge (\forall v \in \text{pverts } ?G. \text{out-arcs } ?G v \neq \{\}) \longrightarrow \text{cyclic-on } f (\text{out-arcs } ?G v)\}$ 

```

unfolding all-maps-list-def **using** assms all-eq

by $(\text{simp add: lists-succ-set-cyc-permutationss distincts-grouped-arcs union-grouped-out-arcs list-digraph-simps})$

```

then have  $*$ :  $\text{lists-succ } ' \text{set } (\text{all-maps-list } G\text{-list}) = \{f. f \text{ permutes set } (\text{snd } G\text{-list}) \wedge (\forall v \in \text{set } (\text{fst } G\text{-list}). \text{out-arcs } (\text{with-proj } (\text{pverts} = \text{set } (\text{fst } G\text{-list}), \text{parcs} = \text{set } (\text{snd } G\text{-list}))) v \neq \{\}) \longrightarrow \text{cyclic-on } f (\text{out-arcs } (\text{with-proj } (\text{pverts} = \text{set } (\text{fst } G\text{-list}), \text{parcs} = \text{set } (\text{snd } G\text{-list}))) v)\}$ 

```

by $(\text{auto simp add: maps-all-maps-list-def all-maps-def list-digraph-simps list-digraph-ext-def})$

```

then have  $**$ :  $\bigwedge f. \neg (f \text{ permutes set } (\text{snd } G\text{-list}) \wedge (\forall a. a \in \text{set } (\text{fst } G\text{-list}) \longrightarrow \text{out-arcs } (\text{with-proj } (\text{pverts} = \text{set } (\text{fst } G\text{-list}), \text{parcs} = \text{set } (\text{snd } G\text{-list}))) a \neq \{\}) \longrightarrow \text{cyclic-on } f (\text{out-arcs } (\text{with-proj } (\text{pverts} = \text{set } (\text{fst } G\text{-list}), \text{parcs} = \text{set } (\text{snd } G\text{-list}))) a)) \vee f \in \text{lists-succ } ' \text{set } (\text{all-maps-list } G\text{-list})$ 

```

by force

from $*$ **show** $?thesis$

by $(\text{auto simp add: maps-all-maps-list-def all-maps-def list-digraph-simps list-digraph-ext-def})$
 $(\text{use } ** \text{ in blast})$

qed

7 Compute Face Cycles

definition $\text{lists-fc-succ} :: ('a \times 'a) \text{ list list} \Rightarrow ('a \times 'a) \Rightarrow ('a \times 'a) \text{ where}$

$\text{lists-fc-succ } xss = (\text{let } sxss = \bigcup (\text{sset } xss) \text{ in } (\lambda x. \text{lists-succ } xss (\text{swap-in } sxss x)))$

locale $\text{lists-digraph-map} =$

fixes $G\text{-list} :: 'b \text{ list} \times ('b \times 'b) \text{ list}$

and $xss :: ('b \times 'b) \text{ list list}$

assumes $\text{digraph-map: digraph-map } (\text{list-digraph } G\text{-list}) (\text{to-map}' (\text{snd } G\text{-list}) xss)$

assumes $\text{no-loops: } \bigwedge a. a \in \text{parcs } (\text{list-digraph } G\text{-list}) \Longrightarrow \text{fst } a \neq \text{snd } a$

assumes $\text{distincts-xss: distincts } xss$

assumes $\text{parcs-xss: parcs } (\text{list-digraph } G\text{-list}) = \bigcup (\text{sset } xss)$

begin

abbreviation *(input)* $G \equiv \text{list-digraph } G\text{-list}$
abbreviation *(input)* $M \equiv \text{to-map}' (\text{snd } G\text{-list}) \text{ } xss$

lemma *edge-rev-simps*:
assumes $(u,v) \in \text{parcs } G$ **shows** $\text{edge-rev } M (u,v) = (v,u)$
using *assms*
unfolding *to-map-def list-digraph-ext-def* **by** *(auto simp: swap-in-def to-map-def)*

end

sublocale *lists-digraph-map* \subseteq *digraph-map* $G M$ **by** *(rule local.digraph-map)*

sublocale *lists-digraph-map* \subseteq *pair-graph* G
proof
fix e **assume** $e \in \text{parcs } G$
then have $e \in \text{arcs } G$ **by** *simp*
then have $\text{head } G e \in \text{verts } G$ $\text{tail } G e \in \text{verts } G$ **by** *(blast dest: wellformed)+*
then show $\text{fst } e \in \text{pverts } G$ $\text{snd } e \in \text{pverts } G$ **by** *auto*
next
fix e **assume** $e \in \text{parcs } G$ **then show** $\text{fst } e \neq \text{snd } e$ **using** *no-loops* **by** *simp*
next
show *finite (pverts G)* *finite (parcs G)*
unfolding *list-digraph-ext-def* **by** *simp-all*
next
{ fix $u v$ **assume** $(u,v) \in \text{parcs } G$
then have $\text{edge-rev } M (u,v) \in \text{parcs } G$
using *edge-rev-in-arcs* **by** *simp*
then have $(v,u) \in \text{parcs } G$ **using** $\langle (u,v) \in - \rangle$ **by** *(simp add: edge-rev-simps)* **}**
then show *symmetric G*
unfolding *symmetric-def* **by** *(auto intro: symI)*

qed

context *lists-digraph-map* **begin**

definition *lists-fcs* \equiv *orbits-list (lists-fc-succ xss)*

lemma *M-simps*:
 $\text{edge-succ } M = \text{lists-succ } xss$
unfolding *to-map-def* **by** *(cases G-list) auto*

lemma *lists-fc-succ-permutes*: *lists-fc-succ xss permutes* $(\bigcup (\text{sset } xss))$
proof –
have $\forall (u,v) \in \bigcup (\text{sset } xss). (v,u) \in \bigcup (\text{sset } xss)$
using *sym-arcs* **unfolding** *parcs-xss[symmetric]* *symmetric-def* **by** *(auto elim: symE)*
then have *swap-in* $(\bigcup (\text{sset } xss))$ *permutes* $\bigcup (\text{sset } xss)$
using *distincts-xss*
apply *(auto simp: permutes-def split: if-splits)*
unfolding *swap-in-def*

```

apply (simp-all split: if-splits prod.splits)
apply metis+
done
moreover
have lists-succ xss permutes ( $\bigcup$  (sset xss))
  using lists-succ-permutes[OF distincts-xss] by simp
moreover
have lists-fc-succ xss = lists-succ xss o swap-in ( $\bigcup$  (sset xss))
  by (simp add: fun-eq-iff lists-fc-succ-def)
ultimately
show ?thesis by (metis permutes-compose)
qed

lemma permutation-lists-fc-succ[intro, simp]: permutation (lists-fc-succ xss)
  using lists-fc-succ-permutes by (auto simp: permutation-permutes)

lemma face-cycle-succ-conv: face-cycle-succ = lists-fc-succ xss
  using parcs-xss unfolding face-cycle-succ-def
  by (simp add: fun-eq-iff to-map-def lists-fc-succ-def swap-in-def list-digraph-ext-def)

lemma sset-lists-fcs:
  sset (lists-fcs as) = {face-cycle-set a | a. a ∈ set as}
  by (auto simp: lists-fcs-def sset-orbits-list face-cycle-set-def face-cycle-succ-conv)

lemma distincts-lists-fcs: distinct as  $\implies$  distincts (lists-fcs as)
  by (simp add: lists-fcs-def distincts-orbits-list)

lemma face-cycle-set-ss: a ∈ parcs G  $\implies$  face-cycle-set a  $\subseteq$  parcs G
  using in-face-cycle-setD with-proj-simps(2) by blast

lemma face-cycle-succ-neq:
  assumes a ∈ parcs G shows face-cycle-succ a  $\neq$  a
  using assms no-loops by (intro face-cycle-succ-neq) auto

lemma card-face-cycle-sets-conv:
  shows card (pre-digraph-map.face-cycle-sets G M) = length (lists-fcs (remdups (snd G-list)))
proof –
  interpret digraph-map G M by (rule digraph-map)

  have face-cycle-sets = {face-cycle-set a | a. a ∈ parcs G}
  by (auto simp: face-cycle-sets-def)
  also have  $\dots = \text{sset (lists-fcs (remdups (snd G-list)))}$ 
  unfolding sset-lists-fcs by (simp add: list-digraph-simps)
  also have card  $\dots = \text{length (lists-fcs (remdups (snd G-list)))}$ 
  by (simp add: card-image distincts-inj-on-set distinct-card distincts-distinct distincts-lists-fcs)
  finally show ?thesis .
qed

```

end

definition *gen-succ* $\equiv \lambda as\ xs. [b. (a,b) <- as, a \in set\ xs]$

interpretation *RTLl*: *set-access-gen set* $\lambda x\ xs. x \in set\ xs \ [] \ \lambda xs\ ys. remdups\ (xs \ @ \ ys)$ *gen-succ*

by *standard* (*auto simp: gen-succ-def*)

hide-const (**open**) *gen-succ*

It would suffice to check that $set\ (RTLl.rtrancl-i\ A\ [u]) = set\ V$. We don't do this here, since it makes the proof more complicated (and is not necessary for the graphs we care about)

definition *sccs-verts-impl* $:: 'a\ list \times ('a \times 'a)\ list \Rightarrow 'a\ set\ set$ **where**
sccs-verts-impl $G \equiv set\ ' (\lambda x. RTLl.rtrancl-i\ (snd\ G)\ [x])\ ' set\ (fst\ G)$

definition *isolated-verts-impl* $:: 'a\ list \times ('a \times 'a)\ list \Rightarrow 'a\ list$ **where**
isolated-verts-impl $G = [v \leftarrow (fst\ G). \neg(\exists e \in set\ (snd\ G). fst\ e = v)]$

definition *pair-graph-impl* $:: 'a\ list \times ('a \times 'a)\ list \Rightarrow bool$ **where**
pair-graph-impl $G \equiv case\ G\ of\ (V,A) \Rightarrow (\forall (u,v) \in set\ A. u \neq v \wedge u \in set\ V \wedge v \in set\ V \wedge (v,u) \in set\ A)$

definition *genus-impl* $:: 'a\ list \times ('a \times 'a)\ list \Rightarrow ('a \times 'a)\ list\ list \Rightarrow int$ **where**
genus-impl $G\ M \equiv case\ G\ of\ (V,A) \Rightarrow$
 $(int\ (2*card\ (sccs-verts-impl\ G)) - int\ (length\ (isolated-verts-impl\ G))$
 $- (int\ (length\ V) - int\ (length\ A)\ div\ 2$
 $+ int\ (length\ (orbits-list-impl\ (lists-fc-succ\ M)\ A))))\ div\ 2$

definition *comb-planar-impl* $:: 'a\ list \times ('a \times 'a)\ list \Rightarrow bool$ **where**
comb-planar-impl $G \equiv case\ G\ of\ (V,A) \Rightarrow$
 $let\ i = int\ (2*card\ (sccs-verts-impl\ G)) - int\ (length\ (isolated-verts-impl\ G))$
 $- int\ (length\ V) + int\ (length\ A)\ div\ 2$
 $in\ (\exists M \in set\ (all-maps-list\ G). (i - int\ (length\ (orbits-list-impl\ (lists-fc-succ\ M)\ A))))\ div\ 2 = 0)$

lemma *sccs-verts-impl-correct*:

assumes *pair-pseudo-graph* (*list-digraph* G)

shows *pre-digraph.sccs-verts* (*list-digraph* G) = *sccs-verts-impl* G

proof –

interpret *pair-pseudo-graph* *list-digraph* G **by** *fact*

{ **fix** u **assume** $u \in set\ (fst\ G)$

then **have** $\bigwedge x. (u, x) \in (set\ (snd\ G))^* \Longrightarrow x \in set\ (fst\ G)$

by (*metis in-arcsD2 list-digraph-simps rtrancl.cases*)

then **have** $set\ (RTLl.rtrancl-i\ (snd\ G)\ [u]) = \{v. u \rightarrow^* list-digraph\ G\ v\}$

unfolding *RTLl.rtrancl-impl* *reachable-conv* **by** (*auto simp: list-digraph-simps*
 $\langle u \in \cdot \rangle$)

```

also have ... = scc-of u
  unfolding scc-of-def by (auto intro: symmetric-reachable')
  finally have scc-of u = set (RTL.I.rtrancl-i (snd G) [u]) by simp
}
then have pre-digraph.sccs-verts (list-digraph G) = set ' (λx. RTL.I.rtrancl-i
(snd G) [x]) ' set (fst G)
  unfolding sccs-verts-conv-scc-of list-digraph-simps
  by (force intro: rev-image-eqI)
then show ?thesis unfolding sccs-verts-impl-def by simp
qed

lemma isolated-verts-impl-correct:
  pre-digraph.isolated-verts (list-digraph G) = set (isolated-verts-impl G)
  by (auto simp: pre-digraph.isolated-verts-def isolated-verts-impl-def list-digraph-simps
out-arcs-def)

lemma pair-graph-impl-correct[code]:
  pair-graph (list-digraph G) = pair-graph-impl G (is ?L = ?R)
  unfolding pair-graph-def pair-digraph-def pair-fin-digraph-def pair-wf-digraph-def
pair-fin-digraph-axioms-def pair-loopfree-digraph-def pair-loopfree-digraph-axioms-def
pair-sym-digraph-def pair-sym-digraph-axioms-def pair-pseudo-graph-def
pair-graph-impl-def
  by (auto simp: pair-graph-impl-def list-digraph-simps symmetric-def intro: symI
dest: symD split: prod.splits)

lemma genus-impl-correct:
  assumes dist-V: distinct (fst G) and dist-A: distinct (snd G)
  assumes lists-digraph-map G M
  shows pre-digraph-map.euler-genus (list-digraph G) (to-map' (snd G) M) =
genus-impl G M
proof –
  interpret lists-digraph-map G M by fact
  obtain V A where G-eq: G = (V,A) by (cases G)
  moreover
  have distinct (isolated-verts-impl G)
    using dist-V by (auto simp: isolated-verts-impl-def )
  moreover
  have faces: card face-cycle-sets = length (orbits-list-impl (lists-fc-succ M) (snd
G))
    using dist-A
    by (simp add: card-face-cycle-sets-conv lists-fcs-def orbits-list-conv-impl dis-
tinct-remdups-id)
  ultimately show ?thesis
    using pair-pseudo-graph dist-V dist-A
    unfolding euler-genus-def euler-char-def genus-impl-def card-sccs-verts[symmetric]

  by (simp add: sccs-verts-impl-correct isolated-verts-impl-correct
distinct-card list-digraph-simps zdiv-int)
qed

```

```

lemma elems-all-maps-list:
  assumes  $M \in \text{set } (\text{all-maps-list } G)$  distinct (snd G)
  shows  $\bigcup (\text{sset } M) = \text{set } (\text{snd } G)$ 
  using assms
  by (simp add: all-maps-list-def in-set-cyc-permutationss distincts-grouped-arcs
union-grouped-out-arcs[symmetric])
  (metis set-map)

lemma comb-planar-impl-altdef: comb-planar-impl  $G = (\exists M \in \text{set } (\text{all-maps-list } G).
genus-impl  $G M = 0)$ 
  unfolding comb-planar-impl-def Let-def genus-impl-def by (cases G) (simp add: algebra-simps)

lemma comb-planar-impl-correct:
  assumes pair-graph (list-digraph G)
  assumes dist-V: distinct (fst G) and dist-A: distinct (snd G)
  shows comb-planar (list-digraph G) = comb-planar-impl G (is ?L = ?R)
proof –
  interpret  $G$ : pair-graph list-digraph G by fact
  let  $?G = \text{list-digraph } G$ 
  have  $*$ : all-maps (list-digraph G) = set (maps-all-maps-list G)
  by (rule set-maps-all-maps-list) (unfold-locales, simp add: dist-A)

obtain  $V A$  where  $G = (V, A)$  by (cases G)

{ fix  $M$  assume  $M \in \text{set } (\text{all-maps-list } G)$ 
  have digraph-map (list-digraph G) (to-map' (snd G) M)
  using  $\langle M \in \cdot \rangle$  G.all-maps-correct by (auto simp: * maps-all-maps-list-def)
  then interpret  $G$ : digraph-map list-digraph G to-map' (snd G) M .

  have distincts M using  $\langle M \in \cdot \rangle$ 
  using dist-A distincts-in-all-maps-list by blast

  have lists-digraph-map G M
  using elems-all-maps-list[OF \langle M \in \cdot \rangle \langle distinct (snd G) \rangle]
  apply unfold-locales
  by (auto intro: \langle distincts M \rangle dest: G.adj-not-same) (auto simp: list-digraph-simps)
} note ldm = this

have comb-planar ?G = (\exists M \in \{M. digraph-map ?G M\}. pre-digraph-map.euler-genus
?G M = 0)
  unfolding comb-planar-def by simp
also have  $\dots = (\exists M \in \text{set } (\text{all-maps-list } G). pre-digraph-map.euler-genus (list-digraph
G)
  (to-map (set (snd G)) (lists-succ M) = 0)
  unfolding comb-planar-def comb-planar-impl-def Let-def G.all-maps-correct[symmetric]
set-maps-all-maps-list[OF G.pair-wf-digraph dist-A] maps-all-maps-list-def by
simp$$ 
```

also have $\dots = (\exists M \in \text{set } (\text{all-maps-list } G). \text{genus-impl } G \ M = 0)$
using *ldm assms* **by** (*simp add: genus-impl-correct*)
also have $\dots = \text{comb-planar-impl } G$
unfolding *comb-planar-impl-def genus-impl-def Let-def* **by** (*simp add: $\langle G = (V, A) \rangle$ algebra-simps*)
finally show *?thesis* .
qed

end
theory *Planar-Complete*
imports
Digraph-Map-Impl
begin

8 Kuratowski Graphs are not Combinatorially Planar

8.1 A concrete K5 graph

definition *c-K5-list* $\equiv ([0..4], [(x,y). x <- [0..4], y <- [0..4], x \neq y])$

abbreviation *c-K5* $:: \text{int pair-pre-digraph where}$
c-K5 $\equiv \text{list-digraph } c\text{-K5-list}$

lemma *c-K5-not-comb-planar*: $\neg \text{comb-planar } c\text{-K5}$
by (*subst comb-planar-impl-correct*) *eval+*

lemma *pverts-c-K5*: $pverts \ c\text{-K5} = \{0..4\}$
by (*simp add: c-K5-list-def list-digraph-ext-def*)

lemma *parcs-c-K5*: $parcs \ c\text{-K5} = \{(u,v). u \in \{0..4\} \wedge v \in \{0..4\} \wedge u \neq v\}$
by (*auto simp: c-K5-list-def list-digraph-ext-def*)

lemmas *c-K5-simps* = *pverts-c-K5 parcs-c-K5*

lemma *complete-c-K5*: $K_5 \ c\text{-K5}$

proof –

interpret *K5*: *pair-graph c-K5* **by** *eval*
show *?thesis* **unfolding** *complete-digraph-def* **by** (*auto simp: c-K5-simps*)
qed

8.2 A concrete K33 graph

definition *c-K33-list* $\equiv ([0..5], [(x,y). x <- [0..5], y <- [0..5], \text{even } x \longleftrightarrow \text{odd } y])$

abbreviation *c-K33* $:: \text{int pair-pre-digraph where}$
c-K33 $\equiv \text{list-digraph } c\text{-K33-list}$

lemma *c-K33-not-comb-planar*: \neg comb-planar *c-K33*
 by (subst comb-planar-impl-correct) eval+

lemma *complete-c-K33*: $K_{3,3}$ *c-K33*

proof –

interpret *K33*: pair-graph *c-K33* by eval

show ?thesis

unfolding *complete-bipartite-digraph-def*

apply (intro conjI)

apply *unfold-locales*

apply (rule exI[of - {0,2,4}])

apply (rule exI[of - {1,3,5}])

unfolding *c-K33-list-def list-digraph-simps with-proj-simps*

apply eval

done

qed

8.3 Generalization to arbitrary Kuratowski Graphs

8.3.1 Number of Face Cycles is a Graph Invariant

lemma (in *digraph-map*) *wrap-wrap-iso*:

assumes *hom*: digraph-isomorphism *hom*

assumes *f*: $f \in \text{arcs } G \rightarrow \text{arcs } G$ **and** *g*: $g \in \text{arcs } G \rightarrow \text{arcs } G$

shows *wrap-iso-arcs hom f* (*wrap-iso-arcs hom g x*) = *wrap-iso-arcs hom (f o g)*

x

proof –

have $\bigwedge x. x \in \text{arcs } G \implies g x \in \text{arcs } G$ **using** *g* **by** auto

with *hom f* **show** ?thesis

by (cases $x \in \text{iso-arcs hom } \text{' arcs } G$) (auto simp: *wrap-iso-arcs-def perm-restrict-simps*)

qed

lemma (in *digraph-map*) *face-cycle-succ-iso*:

assumes *hom*: digraph-isomorphism *hom* $x \in \text{iso-arcs hom } \text{' arcs } G$

shows *pre-digraph-map.face-cycle-succ* (*map-iso hom*) *x* = *wrap-iso-arcs hom*

face-cycle-succ x

using *assms* **by** (*simp add: pre-digraph-map.face-cycle-succ-def map-iso-def wrap-wrap-iso*)

lemma (in *digraph-map*) *face-cycle-set-iso*:

assumes *hom*: digraph-isomorphism *hom* $x \in \text{iso-arcs hom } \text{' arcs } G$

shows *pre-digraph-map.face-cycle-set* (*map-iso hom*) *x* = *iso-arcs hom* ' *face-cycle-set*
 (*iso-arcs (inv-iso hom) x*)

proof –

have *: $\bigwedge x y. x \in \text{orbit face-cycle-succ } y \implies y \in \text{arcs } G \implies x \in \text{arcs } G$

$\bigwedge x. x \in \text{arcs } G \implies x \in \text{orbit face-cycle-succ } x$

using *face-cycle-set-def* **by** (auto simp: *in-face-cycle-setD*)

show ?thesis

using *assms* **unfolding** *pre-digraph-map.face-cycle-set-def*

by (subst orbit-inverse [where $g' = \text{pre-digraph-map.face-cycle-succ (map-iso hom)}$])

(*auto simp: * face-cycle-succ-iso*)
qed

lemma (in *digraph-map*) *face-cycle-sets-iso*:
assumes *hom: digraph-isomorphism hom*
shows *pre-digraph-map.face-cycle-sets (app-iso hom G) (map-iso hom) = (λx. iso-arcs hom ' x) ' face-cycle-sets*
using *assms by (auto simp: pre-digraph-map.face-cycle-sets-def face-cycle-set-iso) (auto simp: face-cycle-set-iso intro: rev-image-eqI)*

lemma (in *digraph-map*) *card-face-cycle-sets-iso*:
assumes *hom: digraph-isomorphism hom*
shows *card (pre-digraph-map.face-cycle-sets (app-iso hom G) (map-iso hom)) = card face-cycle-sets*
proof –
have *inj-on ((') (iso-arcs hom)) face-cycle-sets*
by (*rule inj-on-f-imageI digraph-isomorphism-inj-on-arcs hom in-face-cycle-setsD*) +
then show *?thesis using hom by (simp add: face-cycle-sets-iso card-image)*
qed

8.3.2 Combinatorial planarity is a Graph Invariant

lemma (in *digraph-map*) *euler-char-iso*:
assumes *digraph-isomorphism hom*
shows *pre-digraph-map.euler-char (app-iso hom G) (map-iso hom) = euler-char*
using *assms by (auto simp: pre-digraph-map.euler-char-def card-face-cycle-sets-iso)*

lemma (in *digraph-map*) *euler-genus-iso*:
assumes *digraph-isomorphism hom*
shows *pre-digraph-map.euler-genus (app-iso hom G) (map-iso hom) = euler-genus*
using *assms by (auto simp: pre-digraph-map.euler-genus-def euler-char-iso)*

lemma (in *wf-digraph*) *comb-planar-iso*:
assumes *digraph-isomorphism hom*
shows *comb-planar (app-iso hom G) ⟷ comb-planar G*

proof
assume *comb-planar G*
then obtain *M* **where** *digraph-map G M pre-digraph-map.euler-genus G M = 0*
by (*auto simp: comb-planar-def*)
then have *digraph-map (app-iso hom G) (pre-digraph-map.map-iso G M hom) pre-digraph-map.euler-genus (app-iso hom G) (pre-digraph-map.map-iso G M hom) = 0*
using *assms by (auto intro: digraph-map.digraph-map-isoI simp: digraph-map.euler-genus-iso)*
then show *comb-planar (app-iso hom G)*
by (*metis comb-planar-def*)
next
let *?G = app-iso hom G*
assume *comb-planar ?G*

```

then obtain  $M$  where  $\text{digraph-map } ?G M$ 
   $\text{pre-digraph-map.euler-genus } ?G M = 0$ 
  by (auto simp: comb-planar-def)
moreover
have  $\text{pre-digraph.digraph-isomorphism } ?G (\text{inv-iso hom})$ 
  using assms by (rule digraph-isomorphism-invI)
ultimately
have  $\text{digraph-map } (\text{app-iso } (\text{inv-iso hom}) ?G) (\text{pre-digraph-map.map-iso } ?G M$ 
   $(\text{inv-iso hom}))$ 
   $\text{pre-digraph-map.euler-genus } (\text{app-iso } (\text{inv-iso hom}) ?G) (\text{pre-digraph-map.map-iso}$ 
   $?G M (\text{inv-iso hom})) = 0$ 
  using assms by (auto intro: digraph-map.digraph-map-isoI simp only: di-
  graph-map.euler-genus-iso)
  then show comb-planar  $G$ 
  using assms by (auto simp: comb-planar-def)
qed

```

8.3.3 Completeness is a Graph Invariant

```

lemma (in loopfree-digraph) loopfree-digraphI-app-iso:
  assumes  $\text{digraph-isomorphism hom}$ 
  shows loopfree-digraph ( $\text{app-iso hom } G$ )
proof –
  interpret  $iG: \text{wf-digraph app-iso hom } G$  using assms by (rule wf-digraphI-app-iso)
  show thesis
  using assms digraph-isomorphism-inj-on-verts[OF assms]
  by unfold-locales (auto simp: iso-verts-tail iso-verts-head dest: inj-onD no-loops)
qed

```

```

lemma (in nomulti-digraph) nomulti-digraphI-app-iso:
  assumes  $\text{digraph-isomorphism hom}$ 
  shows nomulti-digraph ( $\text{app-iso hom } G$ )
proof –
  interpret  $iG: \text{wf-digraph app-iso hom } G$  using assms by (rule wf-digraphI-app-iso)
  show thesis
  using assms
  by unfold-locales (auto simp: iso-verts-tail iso-verts-head arc-to-ends-def no-multi-arcs
  dest: inj-onD)
qed

```

```

lemma (in pre-digraph) symmetricI-app-iso:
  assumes  $\text{digraph-isomorphism hom}$ 
  assumes symmetric  $G$ 
  shows symmetric ( $\text{app-iso hom } G$ )
proof –
  let  $?G = \text{app-iso hom } G$ 
  have sym (arcs-ends  $?G$ )
  proof (rule symI)
    fix  $u v$  assume  $u \rightarrow ?G v$ 

```

then obtain a where $a: a \in \text{arcs } ?G \text{ tail } ?G a = u \text{ head } ?G a = v$ by *auto*
then obtain $a0$ where $a0: a0 \in \text{arcs } G a = \text{iso-arcs hom } a0$ by *auto*
with $\langle \text{symmetric } G \rangle$ obtain $b0$ where $b0 \in \text{arcs } G \text{ tail } G b0 = \text{head } G a0$
head $G b0 = \text{tail } G a0$
by (*auto simp: symmetric-def arcs-ends-conv elim: symE*)
moreover
define b where $b = \text{iso-arcs hom } b0$
ultimately
have $b \in \text{iso-arcs hom } \langle \text{arcs } G \text{ tail } ?G b = v \text{ head } ?G b = u$
using $a a0$ *assms* by (*auto simp: iso-verts-tail iso-verts-head*)
then show $v \rightarrow ?G u$ by (*auto simp: arcs-ends-conv*)
qed
then show $?thesis$ by (*simp add: symmetric-def*)
qed

lemma (in *sym-digraph*) *sym-digraphI-app-iso*:
assumes *digraph-isomorphism hom*
shows *sym-digraph (app-iso hom G)*
proof –
interpret $iG: \text{wf-digraph app-iso hom } G$ using *assms* by (*rule wf-digraphI-app-iso*)
show $?thesis$ using *assms* by (*unfold-locales (intro symmetricI-app-iso sym-arcs)*)

qed

lemma (in *graph*) *graphI-app-iso*:
assumes *digraph-isomorphism hom*
shows *graph (app-iso hom G)*
proof –
interpret $iG: \text{fin-digraph app-iso hom } G$
using *assms* by (*rule fin-digraphI-app-iso*)
interpret $iG: \text{loopfree-digraph app-iso hom } G$
using *assms* by (*rule loopfree-digraphI-app-iso*)
interpret $iG: \text{nomulti-digraph app-iso hom } G$
using *assms* by (*rule nomulti-digraphI-app-iso*)
interpret $iG: \text{sym-digraph app-iso hom } G$
using *assms* by (*rule sym-digraphI-app-iso*)
show $?thesis$ by (*intro-locales*)
qed

lemma (in *wf-digraph*) *graph-app-iso-eq*:
assumes *digraph-isomorphism hom*
shows *graph (app-iso hom G) \longleftrightarrow graph G*
using *assms* by (*metis app-iso-inv digraph-isomorphism-invI graph.graphI-app-iso*)

lemma (in *pre-digraph*) *arcs-ends-iso*:
assumes *digraph-isomorphism hom*
shows *arcs-ends (app-iso hom G) = $(\lambda(u,v). (\text{iso-verts hom } u, \text{iso-verts hom } v))$*
 $\langle \text{arcs-ends } G$
using *assms*

by (*auto simp: arcs-ends-conv image-image iso-verts-tail iso-verts-head cong: image-cong*)

lemma *inj-onI-pair*:

assumes *inj-on* $f S T \subseteq S \times S$

shows *inj-on* $(\lambda(u,v). (f u, f v)) T$

using *assms* **by** (*intro inj-onI*) (*auto dest: inj-onD*)

lemma (*in wf-digraph*) *complete-digraph-iso*:

assumes *digraph-isomorphism* *hom*

shows $K_n (app\text{-}iso\ hom\ G) \longleftrightarrow K_n G$ (**is** $?L \longleftrightarrow ?R$)

proof

assume $?L$

then interpret iG : *graph app-iso hom G* **by** (*simp add: complete-digraph-def*)

{ have $\{(u, v). u \in iso\text{-}verts\ hom\ 'verts\ G \wedge v \in iso\text{-}verts\ hom\ 'verts\ G \wedge u \neq v\}$

$= (\lambda(u,v). (iso\text{-}verts\ hom\ u, iso\text{-}verts\ hom\ v))\ ' \{(u,v). u \in verts\ G \wedge v \in verts\ G \wedge iso\text{-}verts\ hom\ u \neq iso\text{-}verts\ hom\ v\}$ (**is** $?L = -$)

by *auto*

also have $\dots = (\lambda(u,v). (iso\text{-}verts\ hom\ u, iso\text{-}verts\ hom\ v))\ ' \{(u,v). u \in verts\ G \wedge v \in verts\ G \wedge u \neq v\}$

using *digraph-isomorphism-inj-on-verts[OF assms]* **by** (*auto dest: inj-onD*)

finally have $?L = \dots$

} note $X = this$

{ fix A **assume** $A: A \subseteq verts\ G \times verts\ G$

then have *inj-on* $(\lambda(u, v). (iso\text{-}verts\ hom\ u, iso\text{-}verts\ hom\ v)) A$

using A *digraph-isomorphism-inj-on-verts[OF assms]* **by** (*intro inj-onI-pair*)

} note $Y = this$

have $(arcs\text{-}ends\ G \cup \{(u, v). u \in verts\ G \wedge v \in verts\ G \wedge u \neq v\}) \subseteq verts\ G \times verts\ G$

by *auto*

note $Y' = Y[OF\ this]$

show $?R$ **using** *assms* $\langle ?L \rangle$

by (*simp add: complete-digraph-def X arcs-ends-iso graph-app-iso-eq inj-on-Un-image-eq-iff Y'*)

next

assume $?R$ **then show** $?L$ **using** *assms*

by (*fastforce simp add: complete-digraph-def arcs-ends-iso graph-app-iso-eq*)

qed

8.3.4 Conclusion

definition (*in pre-digraph*)

mk-iso $:: ('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'd) \Rightarrow ('a, 'b, 'c, 'd)$ *digraph-isomorphism*

where

mk-iso $fv\ fa \equiv (\!| iso\text{-}verts = fv, iso\text{-}arcs = fa,$

$iso\text{-}head = fv\ o\ head\ G\ o\ the\text{-}inv\text{-}into\ (arcs\ G)\ fa,$

$iso\text{-}tail = fv \circ tail\ G \circ the\text{-}inv\text{-}into\ (arcs\ G)\ fa\ \})$

lemma (in *pre-digraph*) *mk-iso-simps*[*simp*]:

iso-verts (*mk-iso* *fv* *fa*) = *fv*

iso-arcs (*mk-iso* *fv* *fa*) = *fa*

by (*auto simp: mk-iso-def*)

lemma (in *wf-digraph*) *digraph-isomorphism-mk-iso*:

assumes *inj-on* *fv* (*verts* *G*) *inj-on* *fa* (*arcs* *G*)

shows *digraph-isomorphism* (*mk-iso* *fv* *fa*)

using *assms* **by** (*auto simp: digraph-isomorphism-def mk-iso-def the-inv-into-f-f wf-digraph*)

definition *pairself* *f* $\equiv \lambda x. case\ x\ of\ (u,v) \Rightarrow (f\ u, f\ v)$

lemma *inj-on-pairself*:

assumes *inj-on* *f* *S* **and** $T \subseteq S \times S$

shows *inj-on* (*pairself* *f*) *T*

using *assms* **unfolding** *pairself-def* **by** (*rule inj-onI-pair*)

definition

mk-iso-nomulti $:: ('a,'b)\ pre\text{-}digraph \Rightarrow ('c,'d)\ pre\text{-}digraph \Rightarrow ('a \Rightarrow 'c) \Rightarrow ('a, 'b, 'c, 'd)\ digraph\text{-}isomorphism$

where

mk-iso-nomulti *G* *H* *fv* $\equiv \{$

iso-verts = *fv*,

iso-arcs = *the-inv-into* (*arcs* *H*) (*arc-to-ends* *H*) *o* *pairself* *fv* *o* *arc-to-ends* *G*,

iso-head = *head* *H*,

iso-tail = *tail* *H*

$\}$

lemma (in *pre-digraph*) *mk-iso-simps-nomulti*[*simp*]:

iso-verts (*mk-iso-nomulti* *G* *H* *fv*) = *fv*

iso-head (*mk-iso-nomulti* *G* *H* *fv*) = *head* *H*

iso-tail (*mk-iso-nomulti* *G* *H* *fv*) = *tail* *H*

by (*auto simp: mk-iso-nomulti-def*)

lemma (in *nomulti-digraph*)

assumes *nomulti-digraph* *H*

assumes *fv: inj-on* *fv* (*verts* *G*) *verts* *H* = *fv* ' *verts* *G* **and** *arcs-ends: arcs-ends* *H* = *pairself* *fv* ' *arcs-ends* *G*

shows *digraph-isomorphism-mk-iso-nomulti: digraph-isomorphism* (*mk-iso-nomulti* *G* *H* *fv*) (**is** *?t-multi*)

and *ap-iso-mk-iso-nomulti-eq: app-iso* (*mk-iso-nomulti* *G* *H* *fv*) *G* = *H* (**is** *?t-app*)

and *digraph-iso-mk-iso-nomulti: digraph-iso* *G* *H* (**is** *?t-iso*)

using *assms*

proof –

interpret *H: nomulti-digraph* *H* **by** *fact*

```

let ?fa = iso-arcs (mk-iso-nomulti G H fv)

have fa: bij-betw ?fa (arcs G) (arcs H)
proof -
  have bij-betw (arc-to-ends G) (arcs G) (arcs-ends G)
    by (auto simp: bij-betw-def inj-on-arc-to-ends arcs-ends-def)
  also have bij-betw (pairself fv) (arcs-ends G) (arcs-ends H)
    using arcs-ends by (auto simp: bij-betw-def arcs-ends-def arc-to-ends-def intro:
fv inj-on-pairself)
  also (bij-betw-trans) have bij-betw (the-inv-into (arcs H) (arc-to-ends H))
(arcs-ends H) (arcs H)
    by (auto simp: bij-betw-def the-inv-into-into H.inj-on-arc-to-ends arcs-ends-def
inj-on-the-inv-into)
  finally (bij-betw-trans) show ?thesis
    by (simp add: mk-iso-nomulti-def o-assoc)
qed
moreover
{ fix a assume a ∈ arcs G
  then have pairself fv (arc-to-ends G a) ∈ arcs-ends H
    using arcs-ends by (auto simp: arcs-ends-def)
  then obtain b where (pairself fv (arc-to-ends G a)) = arc-to-ends H b b ∈
arcs H
    by (auto simp: arcs-ends-def)
  then have fv (tail G a) = tail H (?fa a) fv (head G a) = head H (?fa a)
    by (auto simp: mk-iso-nomulti-def the-inv-into-f-f H.inj-on-arc-to-ends)
    (auto simp: pairself-def arc-to-ends-def)
}
ultimately
show ?t-multi ?t-app using fv by (auto simp: digraph-isomorphism-def bij-betw-def
wf-digraph)
then show ?t-iso by (auto simp: digraph-iso-def)
qed

lemma complete-digraph-are-iso:
  assumes  $K_n$  G  $K_n$  H shows digraph-iso G H
proof -
  interpret G: graph G using assms by (simp add: complete-digraph-def)
  interpret H: graph H using assms by (simp add: complete-digraph-def)

  from assms have card (verts G) = n card (verts H) = n
    by (auto simp: complete-digraph-def)
  with G.finite-verts H.finite-verts obtain fv where bij-betw fv (verts G) (verts
H)
    by (metis finite-same-card-bij)
  then have fv: inj-on fv (verts G) verts H = fv ‘ verts G by (auto simp:
bij-betw-def)

  have arcs-ends H = {(u,v). u ∈ verts H ∧ v ∈ verts H ∧ u ≠ v}
    using ⟨ $K_n$  H⟩ by (auto simp: complete-digraph-def)

```

also have $\dots = \text{pairself } fv \text{ ' } \{(u,v). u \in \text{verts } G \wedge v \in \text{verts } G \wedge u \neq v\}$ (**is** $?L = ?R$)
proof (*intro set-eqI iffI*)
fix x **assume** $x \in ?L$
then have $\text{fst } x \in fv \text{ ' } \text{verts } G$ $\text{snd } x \in fv \text{ ' } \text{verts } G$ $\text{fst } x \neq \text{snd } x$
using fv **by** *auto*
then obtain $u v$ **where** $\text{fst } x = fv \ u$ $\text{snd } x = fv \ v$ $u \in \text{verts } G$ $v \in \text{verts } G$ **by** *auto*
then have $(\text{fst } x, \text{snd } x) \in ?R$ **using** $\langle x \in ?L \rangle$ **by** (*auto simp: pairself-def*)
then show $x \in ?R$ **by** *auto*
next
fix x **assume** $x \in ?R$ **then show** $x \in ?L$
using fv **by** (*auto simp: pairself-def dest: inj-onD*)
qed
also have $\dots = \text{pairself } fv \text{ ' } \text{arcs-ends } G$
using $\langle K_n \ G \rangle$ **by** (*auto simp: complete-digraph-def*)
finally have $\text{arcs-ends: arcs-ends } H = \text{pairself } fv \text{ ' } \text{arcs-ends } G$.

show $?thesis$ **using** $H.\text{nomulti-digraph } fv \ \text{arcs-ends}$ **by** (*rule G.digraph-iso-mk-iso-nomulti*)
qed

lemma *pairself-image-prod*:
 $\text{pairself } f \text{ ' } (A \times B) = f \text{ ' } A \times f \text{ ' } B$
by (*auto simp: pairself-def*)

lemma *complete-bipartite-digraph-are-iso*:
assumes $K_{m,n} \ G \ K_{m,n} \ H$ **shows** *digraph-iso* $G \ H$
proof –
interpret G : *graph* G **using** *assms* **by** (*simp add: complete-bipartite-digraph-def*)
interpret H : *graph* H **using** *assms* **by** (*simp add: complete-bipartite-digraph-def*)

from *assms* **obtain** $GU \ GV$ **where** $G\text{-parts: } \text{verts } G = GU \cup GV$ $GU \cap GV = \{\}$
 $\text{card } GU = m$ $\text{card } GV = n$ $\text{arcs-ends } G = GU \times GV \cup GV \times GU$
by (*auto simp: complete-bipartite-digraph-def*)
from *assms* **obtain** $HU \ HV$ **where** $H\text{-parts: } \text{verts } H = HU \cup HV$ $HU \cap HV = \{\}$
 $\text{card } HU = m$ $\text{card } HV = n$ $\text{arcs-ends } H = HU \times HV \cup HV \times HU$
by (*auto simp: complete-bipartite-digraph-def*)

have $\text{fin: } \text{finite } GU$ $\text{finite } GV$ $\text{finite } HU$ $\text{finite } HV$
using $G\text{-parts}$ $H\text{-parts}$ $G.\text{finite-verts}$ $H.\text{finite-verts}$ **by** *simp-all*

obtain $fv\text{-}U$ **where** $fv\text{-}U$: *bij-betw* $fv\text{-}U \ GU \ HU$
using $\langle \text{card } GU = \rangle$ $\langle \text{card } HU = \rangle$ $\langle \text{finite } GU \rangle$ $\langle \text{finite } HU \rangle$ **by** (*metis finite-same-card-bij*)
obtain $fv\text{-}V$ **where** $fv\text{-}V$: *bij-betw* $fv\text{-}V \ GV \ HV$
using $\langle \text{card } GV = \rangle$ $\langle \text{card } HV = \rangle$ $\langle \text{finite } GV \rangle$ $\langle \text{finite } HV \rangle$ **by** (*metis finite-same-card-bij*)

define fv **where** $fv\ x = (if\ x \in GU\ then\ fv-U\ x\ else\ fv-V\ x)$ **for** x
have $\bigwedge x. x \in GV \implies x \notin GU$ **using** $\langle GU \cap GV = \{\} \rangle$ **by** *blast*
then have $bij-fv-UV: bij-betw\ fv\ GU\ HU\ bij-betw\ fv\ GV\ HV$
using $fv-U\ fv-V$ **by** (*auto simp: fv-def cong: bij-betw-cong*)
then have $bij-betw\ fv\ (verts\ G)\ (verts\ H)$
unfolding $\langle verts\ G = - \rangle\ \langle verts\ H = - \rangle$ **using** $\langle HU \cap - = \{\} \rangle$ **by** (*rule*
bij-betw-combine)
then have $fv: inj-on\ fv\ (verts\ G)\ (verts\ H = fv\ `verts\ G)$ **by** (*auto simp:*
bij-betw-def)

have $arcs-ends\ H = HU \times HV \cup HV \times HU$
using $\langle K_{m,n}\ H \rangle\ H-parts$ **by** (*auto simp: complete-digraph-def*)
also have $\dots = pairself\ fv\ ` (GU \times GV \cup GV \times GU)$ (**is** $?L = ?R$)
proof (*intro set-eqI iffI*)
fix x **assume** $x \in ?L$
then have $(fst\ x \in fv\ `GU \wedge snd\ x \in fv\ `GV) \vee (fst\ x \in fv\ `GV \wedge snd\ x \in$
 $fv\ `GU)$
using $bij-fv-UV$ **by** (*auto simp: bij-betw-def*)
then show $x \in ?R$
by (*cases x*) (*auto simp: pairself-image-prod image-Un*)
next
fix x **assume** $x \in ?R$ **then show** $x \in ?L$
using $bij-fv-UV$ **by** (*auto simp: pairself-image-prod image-Un bij-betw-def*)
qed
also have $\dots = pairself\ fv\ `arcs-ends\ G$
using $\langle K_{m,n}\ G \rangle\ G-parts$ **by** (*auto simp: complete-bipartite-digraph-def*)
finally have $arcs-ends: arcs-ends\ H = pairself\ fv\ `arcs-ends\ G$.

show $?thesis$ **using** $H.nomulti-digraph\ fv\ arcs-ends$ **by** (*rule G.digraph-iso-mk-iso-nomulti*)
qed

lemma $K5-not-comb-planar:$

assumes $K_5\ G$ **shows** $\neg comb-planar\ G$
proof –
interpret $graph\ G$ **using** $assms$ **by** (*auto simp: complete-digraph-def*)
have $digraph-iso\ G\ c-K5$
using $assms\ complete-c-K5$ **by** (*rule complete-digraph-are-iso*)
then obtain hom **where** $hom: digraph-isomorphism\ hom\ app-iso\ hom\ G = c-K5$
by (*auto simp: digraph-iso-def*)
then show $?thesis$ **using** $c-K5-not-comb-planar\ comb-planar-iso$ **by** *fastforce*
qed

lemma $K33-not-comb-planar:$

assumes $K_{3,3}\ G$ **shows** $\neg comb-planar\ G$
proof –
interpret $graph\ G$ **using** $assms$ **by** (*auto simp: complete-bipartite-digraph-def*)
have $digraph-iso\ G\ c-K33$
using $assms\ complete-c-K33$ **by** (*rule complete-bipartite-digraph-are-iso*)

```

then obtain hom where hom: digraph-isomorphism hom app-iso hom G = c-K33
  by (auto simp: digraph-iso-def)
then show ?thesis using c-K33-not-comb-planar comb-planar-iso by fastforce
qed

end

```

9 *n*-step reachability

```

theory Reachablen

```

```

imports

```

```

  Graph-Theory.Graph-Theory

```

```

begin

```

```

inductive

```

```

  ntrancl-onp :: 'a set  $\Rightarrow$  'a rel  $\Rightarrow$  nat  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  bool
```

```

  for F :: 'a set and r :: 'a rel
```

```

where

```

```

  ntrancl-on-0:  $a = b \Longrightarrow a \in F \Longrightarrow \text{ntrancl-onp } F \ r \ 0 \ a \ b$ 
```

```

  | ntrancl-on-Suc:  $(a,b) \in r \Longrightarrow \text{ntrancl-onp } F \ r \ n \ b \ c \Longrightarrow a \in F \Longrightarrow \text{ntrancl-onp } F \ r \ (\text{Suc } n) \ a \ c$ 
```

```

lemma ntrancl-onpD-rtrancl-on:

```

```

  assumes ntrancl-onp F r n a b shows  $(a,b) \in \text{rtrancl-on } F \ r$ 
```

```

  using assms by induct (auto intro: converse-rtrancl-on-into-rtrancl-on)
```

```

lemma rtrancl-onE-ntrancl-onp:

```

```

  assumes  $(a,b) \in \text{rtrancl-on } F \ r$  obtains n where ntrancl-onp F r n a b
```

```

proof atomize-elim
```

```

  from assms show  $\exists n. \text{ntrancl-onp } F \ r \ n \ a \ b$ 
```

```

  proof induct
```

```

    case base
```

```

    then have ntrancl-onp F r 0 b b by (auto intro: ntrancl-onp.intros)
```

```

    then show ?case ..
```

```

  next
```

```

    case (step a c)
```

```

    from  $\langle \exists n. \_ \rangle$  obtain n where ntrancl-onp F r n c b ..
```

```

    with  $\langle (a,c) \in r \rangle$  have ntrancl-onp F r (Suc n) a b using  $\langle a \in F \rangle$  by (rule ntrancl-onp.intros)
```

```

    then show ?case ..
```

```

  qed
```

```

qed

```

```

lemma rtrancl-on-conv-ntrancl-onp:  $(a,b) \in \text{rtrancl-on } F \ r \longleftrightarrow (\exists n. \text{ntrancl-onp } F \ r \ n \ a \ b)$ 
```

```

  by (metis ntrancl-onpD-rtrancl-on rtrancl-onE-ntrancl-onp)
```

```

definition nreachable :: 'a,'b) pre-digraph  $\Rightarrow$  'a  $\Rightarrow$  nat  $\Rightarrow$  'a  $\Rightarrow$  bool ( $\langle \_ \rightarrow \_ \rangle$ )

```

[100,100] 40) **where**
 $nreachable\ G\ u\ n\ v \equiv ntrancl\text{-}onp\ (verts\ G)\ (arcs\text{-}ends\ G)\ n\ u\ v$

context *wf-digraph* **begin**

lemma *reachableE-nreachable*:
assumes $u \rightarrow^* v$ **obtains** n **where** $u \rightarrow^n v$
using *assms* **by** (*auto simp: reachable-def nreachable-def elim: rtrancl-onE-ntrancl-onp*)

lemma *converse-nreachable-cases*[*cases pred: nreachable*]:
assumes $u \rightarrow^n v$
obtains (*ntrancl-on-0*) $u = v\ n = 0\ u \in verts\ G$
| (*ntrancl-on-Suc*) $w\ m$ **where** $u \rightarrow w\ n = Suc\ m\ w \rightarrow^m v$
using *assms* **unfolding** *nreachable-def* **by** *cases auto*

lemma *converse-nreachable-induct*[*consumes 1, case-names base step, induct pred: reachable*]:
assumes *major*: $u \rightarrow^n_G v$
and *cases*: $v \in verts\ G \implies P\ 0\ v$
 $\bigwedge n\ x\ y. \llbracket x \rightarrow_G y; y \rightarrow^n_G v; P\ n\ y \rrbracket \implies P\ (Suc\ n)\ x$
shows $P\ n\ u$
using *assms* **unfolding** *nreachable-def* **by** *induct auto*

lemma *converse-nreachable-induct-less*[*consumes 1, case-names base step, induct pred: reachable*]:
assumes *major*: $u \rightarrow^n_G v$
and *cases*: $v \in verts\ G \implies P\ 0\ v$
 $\bigwedge n\ x\ y. \llbracket x \rightarrow_G y; y \rightarrow^n_G v; \bigwedge z\ m. m \leq n \implies (z \rightarrow^m_G v) \implies P\ m\ z \rrbracket \implies$
 $P\ (Suc\ n)\ x$
shows $P\ n\ u$
proof –
have $\bigwedge q\ u. q \leq n \implies (u \rightarrow^q_G v) \implies P\ q\ u$
proof (*induction n arbitrary: u rule: less-induct*)
case (*less n*)
show *?case*
proof (*cases q*)
case 0 **with** *less* **show** *?thesis* **by** (*auto intro: cases elim: converse-nreachable-cases*)
next
case (*Suc q'*)
with $\langle u \rightarrow^q v \rangle$ **obtain** w **where** $u \rightarrow w\ w \rightarrow^{q'} v$ **by** (*auto elim: converse-nreachable-cases*)
then **show** *?thesis*
unfolding $\langle q = \rightarrow \rangle$ **using** *Suc less* **by** (*auto intro!: less.IH cases*)
qed
qed
with *major* **show** *?thesis* **by** *auto*
qed

end

```

end
theory Permutations-2
imports
  HOL-Combinatorics.Permutations
  Graph-Theory.Auxiliary
  Executable-Permutations
begin

```

10 More

abbreviation *funswapid* :: 'a ⇒ 'a ⇒ 'a ⇒ 'a (**infix** <⇒_F> 90) **where**
x ⇒_F *y* ≡ *transpose x y*

lemma *in-funswapid-image-iff*: $x \in (a \Rightarrow_F b) \wedge S \longleftrightarrow (a \Rightarrow_F b) x \in S$
by (*fact in-transpose-image-iff*)

lemma *bij-swap-compose*: $\text{bij } (x \Rightarrow_F y \circ f) \longleftrightarrow \text{bij } f$
by (*metis UNIV-I bij-betw-comp-iff2 bij-betw-id bij-swap-iff subsetI*)

lemma *bij-eq-iff*:
assumes *bij f* **shows** $f x = f y \longleftrightarrow x = y$
using *assms* **by** (*auto simp add: bij-iff*)

lemma *swap-swap-id[simp]*: $(x \Rightarrow_F y) ((x \Rightarrow_F y) z) = z$
by (*fact transpose-involutory*)

11 Modifying Permutations

definition *perm-swap* :: 'a ⇒ 'a ⇒ ('a ⇒ 'a) ⇒ ('a ⇒ 'a) **where**
perm-swap x y f ≡ $x \Rightarrow_F y \circ f \circ x \Rightarrow_F y$

definition *perm-rem* :: 'a ⇒ ('a ⇒ 'a) ⇒ ('a ⇒ 'a) **where**
perm-rem x f ≡ *if f x ≠ x then x ⇒_F f x o f else f*

An example:

perm-rem 2 (list-succ [1, 2, 3, 4]) x = list-succ [1, 3, 4] x

lemma *perm-swap-id[simp]*: *perm-swap a b id = id*
by (*auto simp: perm-swap-def*)

lemma *perm-rem-permutes*:
assumes *f permutes S ∪ {x}*
shows *perm-rem x f permutes S*
using *assms* **by** (*auto simp: permutes-def perm-rem-def*) (*metis transpose-def*)+

lemma *perm-rem-same*:
assumes *bij f f y = y* **shows** *perm-rem x f y = f y*
using *assms* **by** (*auto simp: perm-rem-def bij-iff transpose-def*)

lemma *perm-rem-simps*:

assumes *bij f*

shows

$x = y \implies \text{perm-rem } x \ f \ y = x$

$f \ y = x \implies \text{perm-rem } x \ f \ y = f \ x$

$y \neq x \implies f \ y \neq x \implies \text{perm-rem } x \ f \ y = f \ y$

using *assms* **by** (*auto simp: perm-rem-def transpose-def bij-iff*)

lemma *bij-perm-rem[simp]*: $\text{bij } (\text{perm-rem } x \ f) \longleftrightarrow \text{bij } f$

by (*simp add: perm-rem-def bij-swap-compose*)

lemma *perm-rem-conv*: $\bigwedge f \ x \ y. \text{bij } f \implies \text{perm-rem } x \ f \ y = ($

if $x = y$ *then* x

else if $f \ y = x$ *then* $f \ (f \ y)$

else $f \ y$)

by (*auto simp: perm-rem-simps*)

lemma *perm-rem-commutes*:

assumes *bij f* **shows** $\text{perm-rem } a \ (\text{perm-rem } b \ f) = \text{perm-rem } b \ (\text{perm-rem } a \ f)$

proof –

have *bij-simp*: $\bigwedge x \ y. f \ x = f \ y \longleftrightarrow x = y$

using *assms* **by** (*auto simp: bij-iff*)

show *?thesis* **using** *assms* **by** (*auto simp: perm-rem-conv bij-simp fun-eq-iff*)

qed

lemma *perm-rem-id[simp]*: $\text{perm-rem } a \ \text{id} = \text{id}$

by (*simp add: perm-rem-def*)

lemma *perm-swap-comp*: $\text{perm-swap } a \ b \ (f \circ g) \ x = \text{perm-swap } a \ b \ f \ (\text{perm-swap } a \ b \ g \ x)$

by (*auto simp: perm-swap-def*)

lemma *bij-perm-swap-iff[simp]*: $\text{bij } (\text{perm-swap } a \ b \ f) \longleftrightarrow \text{bij } f$

by (*simp add: bij-swap-compose bij-swap-iff perm-swap-def*)

lemma *funpow-perm-swap*: $\text{perm-swap } a \ b \ f \ \sim^n = \text{perm-swap } a \ b \ (f \ \sim^n)$

by (*induct n*) (*auto simp: perm-swap-def fun-eq-iff*)

lemma *orbit-perm-swap*: $\text{orbit } (\text{perm-swap } a \ b \ f) \ x = (a \ \rightleftharpoons_F b) \ \text{' } \text{orbit } f \ ((a \ \rightleftharpoons_F b) \ x)$

by (*auto simp: orbit-altdef funpow-perm-swap*) (*auto simp: perm-swap-def*)

lemma *has-dom-perm-swap*: $\text{has-dom } (\text{perm-swap } a \ b \ f) \ S = \text{has-dom } f \ ((a \ \rightleftharpoons_F b) \ \text{' } S)$

by (*auto simp: has-dom-def perm-swap-def inj-image-mem-iff*) (*metis image-iff swap-swap-id*)

lemma *perm-restrict-dom-subset*:

```

assumes has-dom f A shows perm-restrict f A = f
proof –
  from assms have  $\bigwedge x. x \notin A \implies f x = x$  by (auto simp: has-dom-def)
  then show ?thesis by (auto simp: perm-restrict-def fun-eq-iff)
qed

lemma perm-swap-permutes2:
  assumes f permutes ((x  $\equiv_F$  y) ‘ S)
  shows perm-swap x y f permutes S
  using assms
  by (auto simp: perm-swap-def permutes-conv-has-dom has-dom-perm-swap [unfolded perm-swap-def] intro!: bij-comp)

```

12 Cyclic Permutations

```

lemma cyclic-on-perm-swap:
  assumes cyclic-on f S shows cyclic-on (perm-swap x y f) ((x  $\equiv_F$  y) ‘ S)
  using assms by (rule cyclic-on-image) (auto simp: perm-swap-def)

lemma orbit-perm-rem:
  assumes bij f x  $\neq$  y shows orbit (perm-rem y f) x = orbit f x - {y} (is ?L = ?R)
proof (intro set-eqI iffI)
  fix z assume z  $\in$  ?L
  then show z  $\in$  ?R
    using assms by induct (auto simp: perm-rem-conv bij-iff intro: orbit.intros)
next
  fix z assume A: z  $\in$  ?R
  { assume z  $\in$  orbit f x
    then have (z  $\neq$  y  $\longrightarrow$  z  $\in$  ?L)  $\wedge$  (z = y  $\longrightarrow$  f z  $\in$  ?L)
    proof induct
      case base with assms show ?case by (auto intro: orbit-eqI(1) simp: perm-rem-conv)
    next
      case (step z) then show ?case
        using assms by (cases y = z) (auto intro: orbit-eqI simp: perm-rem-conv)
    qed
  } with A show z  $\in$  ?L by auto
qed

```

```

lemma orbit-perm-rem-eq:
  assumes bij f shows orbit (perm-rem y f) x = (if x = y then {y} else orbit f x - {y})
  using assms by (simp add: orbit-eq-singleton-iff orbit-perm-rem perm-rem-simps)

```

```

lemma cyclic-on-perm-rem:
  assumes cyclic-on f S bij f S  $\neq$  {x} shows cyclic-on (perm-rem x f) (S - {x})
  using assms [unfolded cyclic-on-alldef] by (simp add: cyclic-on-def orbit-perm-rem-eq)
auto

```

```

end
theory Planar-Subdivision
imports
  Graph-Genus
  Reachablen
  Permutations-2
begin

```

13 Combinatorial Planarity and Subdivisions

```

locale subdiv1-contr = subdiv-step +
  fixes HM
  assumes H-map: digraph-map H HM
  assumes edge-rev-conv: edge-rev HM = rev-H

sublocale subdiv1-contr  $\subseteq$  H: digraph-map H HM
  rewrites edge-rev HM = rev-H by (intro H-map edge-rev-conv)+

sublocale subdiv1-contr  $\subseteq$  G: fin-digraph G
  by unfold-locales (auto simp: arcs-G verts-G)

```

```

context subdiv1-contr begin

```

```

definition GM :: 'b pre-map where
  GM  $\equiv$ 
  (| edge-rev = rev-G
    , edge-succ = perm-swap uw uv (perm-swap vw vu (fold perm-rem [wu, wv]
(edge-succ HM)))
  |)

```

```

lemma edge-rev-GM: edge-rev GM = rev-G
  by (simp add: GM-def)

```

```

lemma edge-succ-GM: edge-succ GM = perm-swap uw uv (perm-swap vw (rev-G
uv) (fold perm-rem [wu, wv] (edge-succ HM)))
  by (simp add: GM-def)

```

```

lemma rev-H-eq-rev-G:
  assumes  $x \in \text{arcs } G - \{uv, vu\}$  shows rev-H x = rev-G x
proof -
  have perm-restrict rev-H (arcs G) = perm-restrict rev-G (arcs H)
    using subdiv-step by (auto simp: subdivision-step-def)
  with assms show ?thesis
  unfolding arcs-H by (auto simp: perm-restrict-def fun-eq-iff split: if-splits)
qed

```

```

lemma edge-succ-permutes: edge-succ GM permutes arcs G
proof -

```

```

have arcs  $H \subseteq (vw \Rightarrow_F \text{rev-}G \text{ } uv) \text{ ' } (uw \Rightarrow_F uv) \text{ ' arcs } G \cup \{wv\} \cup \{wu\}$ 
using subdiv-distinct-arcs in-arcs-G
by (auto simp: arcs-H in-funswapid-image-iff transpose-def split: if-splits)
then have perm-swap  $uw \ uv$  (perm-swap  $vw$  (rev-G  $uv$ ) (perm-rem ( $wv$ )
(perm-rem ( $wu$ ) (edge-succ  $HM$ )))) permutes arcs  $G$ 
by (blast intro: perm-rem-permutes perm-swap-permutes2 permutes-subset
H.edge-succ-permutes)
then show ?thesis by (auto simp: edge-succ-GM)
qed

```

lemma out-arcs-empty:

assumes $x \in \text{verts } G$

shows $\text{out-arcs } G \ x = \{\}$ \longleftrightarrow $\text{out-arcs } H \ x = \{\}$

proof

assume $A: \text{out-arcs } H \ x = \{\}$

have tail-eqI: $\bigwedge a. \text{tail } H \ a = \text{tail } G \ a$ **by** (simp only: tail-eq)

{ **fix** a **assume** $a \in \text{out-arcs } G \ x$

moreover have $a \in \text{arcs } H \Longrightarrow a \neq uv$ $a \in \text{arcs } H \Longrightarrow a \neq vu$

using not-in-arcs-H **by** auto

ultimately have $(uw \Rightarrow_F uv) ((vw \Rightarrow_F vu) \ a) \in \text{out-arcs } H \ x$

using subdiv-distinct-arcs in-arcs-H not-in-arcs-H

by (auto simp: arcs-G intro: tail-eqI)

}

then show $\text{out-arcs } G \ x = \{\}$

using A **by** (auto simp del: in-out-arcs-conv)

next

assume $A: \text{out-arcs } G \ x = \{\}$

have tail-eqI: $\bigwedge a. \text{tail } H \ a = \text{tail } G \ a$ **by** (simp only: tail-eq)

{ **fix** a **assume** $a \in \text{out-arcs } H \ x$

moreover have $x \neq w$ **using** assms not-in-verts-G **by** blast

ultimately have $(uw \Rightarrow_F uv) ((vw \Rightarrow_F vu) \ a) \in \text{out-arcs } G \ x$

using subdiv-distinct-arcs in-arcs-G not-in-arcs-G

by (auto simp: arcs-H) (auto simp: transpose-def intro: tail-eqI[symmetric])

}

then show $\text{out-arcs } H \ x = \{\}$

using A **by** (auto simp del: in-out-arcs-conv)

qed

lemma cyclic-on-edge-succ:

assumes $x \in \text{verts } G$ $\text{out-arcs } G \ x \neq \{\}$

shows cyclic-on (edge-succ GM) ($\text{out-arcs } G \ x$)

proof –

have oa-Gx: $\text{out-arcs } G \ x = (uw \Rightarrow_F uv) \text{ ' } (vw \Rightarrow_F vu) \text{ ' } (\text{out-arcs } H \ x - \{wv\} - \{wu\})$

using subdiv-distinct-arcs not-in-arcs-G in-arcs-G

by (auto simp: in-funswapid-image-iff arcs-H transpose-def tail-eq[symmetric] split: if-splits)

```

have cyclic-on (perm-swap uw uv (perm-swap vw (rev-G w) (perm-rem (w)
(perm-rem (wu) (edge-succ HM)))))) (out-arcs G x)
  unfolding oa-Gx
proof (intro cyclic-on-perm-swap cyclic-on-perm-rem)
  show cyclic-on (edge-succ HM) (out-arcs H x)
  using assms by (auto simp: out-arcs-empty verts-H intro: H.edge-succ-cyclic)
  show bij (edge-succ HM) by (simp add: H.bij-edge-succ)
  show bij (perm-rem (wu) (edge-succ HM)) by (simp add: H.bij-edge-succ)

have  $x \neq w$  using assms not-in-verts-G by auto
then have  $wu \notin \text{out-arcs } H \ x \ wv \notin \text{out-arcs } H \ x$ 
  by (auto simp: arc-to-ends-def)
then show  $\text{out-arcs } H \ x - \{wu\} \neq \{wv\} \ \text{out-arcs } H \ x \neq \{wu\}$ 
  by blast+
qed
then show ?thesis by (simp add: edge-succ-GM)
qed

```

```

lemma digraph-map-GM:
  shows digraph-map G GM
  by unfold-locales (auto simp: edge-rev-GM G.arev-dom edge-succ-permutes
cyclic-on-edge-succ verts-G)

```

end

```

sublocale subdiv1-contr  $\subseteq$  GM: digraph-map G GM by (rule digraph-map-GM)

```

```

context subdiv1-contr begin

```

```

lemma reachableGD:
  assumes  $x \rightarrow^*_G y$  shows  $x \rightarrow^*_H y$ 
  using assms
proof induct
  case base then show ?case by (auto simp: verts-H)
next
  case (step x z)
  moreover
  have  $u \rightarrow^*_H v \ v \rightarrow^*_H u$  using adj-with-w by auto
  moreover
  { assume A:  $(x,z) \neq (u,v) \ (x,z) \neq (v,u)$ 
    from  $\langle x \rightarrow_G z \rangle$  obtain a where  $a \in \text{arcs } G \ \text{tail } G \ a = x \ \text{head } G \ a = z$ 
    by auto
    with A have  $a \in \text{arcs } H \ \text{arc-to-ends } H \ a = (x,z) \ \text{tail } H \ a = x \ \text{head } G \ a = z$ 
    by (auto simp: arcs-H tail-eq head-eq arc-to-ends-def fun-eq-iff)
    then have  $x \rightarrow_H z$  by (auto simp: arcs-ends-def intro: rev-image-eqI)
  }
  ultimately

```

show *?case* **by** (*auto intro: H.reachable-trans*)
qed

definition *proj-verts-H* :: 'a \Rightarrow 'a **where**
proj-verts-H $x \equiv$ if $x = w$ then u else x

lemma *proj-verts-H-in-G*: $x \in \text{verts } H \implies \text{proj-verts-H } x \in \text{verts } G$
using *in-verts-G* **by** (*auto simp: proj-verts-H-def verts-H*)

lemma *dominatesHD*:
assumes $x \rightarrow_H y$ **shows** $\text{proj-verts-H } x \rightarrow^*_G \text{proj-verts-H } y$
proof –
have *X1*: $\bigwedge a. (w, y) = \text{arc-to-ends } G \ a \implies a \notin \text{arcs } G$
by (*metis G.adj-in-verts(1) G.dominatesI not-in-verts-G*)
have *X2*: $\bigwedge a. (x, w) = \text{arc-to-ends } G \ a \implies a \notin \text{arcs } G$
by (*metis G.adj-in-verts(2) G.dominatesI not-in-verts-G*)
show *?thesis*
using *assms subdiv-ate-H-rev subdiv-ate in-verts-G*
by (*auto simp: arcs-ends-def arcs-H arc-to-ends-eq proj-verts-H-def G-reach*
dest: X1 X2)
qed

lemma *reachableHD*:
assumes $\text{reach}: x \rightarrow^*_H y$ **shows** $\text{proj-verts-H } x \rightarrow^*_G \text{proj-verts-H } y$
using *assms* **by** *induct (blast intro: proj-verts-H-in-G G.reachable-trans dominatesHD)+*

lemma *H-reach-conv*: $\bigwedge x y. x \rightarrow^*_H y \iff \text{proj-verts-H } x \rightarrow^*_G \text{proj-verts-H } y$
using *w-reach* **by** (*auto simp: reachableHD*)
(auto simp: proj-verts-H-def verts-H split: if-splits dest: reachableGD intro: H.reachable-trans)

lemma *sccs-eq*: $G.\text{sccs-verts} = (\cdot) \text{proj-verts-H } \text{' } H.\text{sccs-verts}$ (**is** $?L = ?R$)
proof (*intro set-eqI iffI*)
fix S **assume** $S \in ?L$
then have $w \notin S$ **using** *G.sccs-verts-subsets not-in-verts-G* **by** *blast*
then have *S-eq*: $\text{proj-verts-H } \text{' } \text{proj-verts-H } - \text{' } S = S$
by (*auto simp: proj-verts-H-def intro: range-eqI*)
then have $\text{proj-verts-H } - \text{' } S \neq \{\}$ **using** $\langle S \in ?L \rangle$ **by** *safe (auto simp: G.sccs-verts-def)*
with $\langle S \in ?L \rangle$ **have** $\text{proj-verts-H } - \text{' } S \in H.\text{sccs-verts}$
by (*auto simp: G.in-sccs-verts-conv-reachable H.in-sccs-verts-conv-reachable H-reach-conv*)
then have $\text{proj-verts-H } \text{' } \text{proj-verts-H } - \text{' } S \in (\cdot) \text{proj-verts-H } \text{' } H.\text{sccs-verts}$
by (*rule imageI*)
then show $S \in ?R$ **by** (*simp only: S-eq*)
next
fix S **assume** $S \in ?R$
have *X*: $\bigwedge v x. v \notin \text{proj-verts-H } \text{' } x \implies v = w \vee (\exists y. v = \text{proj-verts-H } y \wedge y$

$\notin x$)
by (*auto simp: proj-verts-H-def split: if-splits*)
from $\langle S \in ?R \rangle$ **show** $S \in ?L$
using *not-in-verts-G* **by** (*fastforce simp: G.reachable-in-verts G.in-sccs-verts-conv-reachable*
H.in-sccs-verts-conv-reachable H-reach-conv dest: X)
qed

lemma *inj-on-proj-verts-H: inj-on ((\cdot) proj-verts-H) (pre-digraph.sccs-verts H)*
proof (*rule inj-onI*)
fix $S T$ **assume** $A: S \in H.sccs-verts T \in H.sccs-verts$ *proj-verts-H* $\cdot S =$
proj-verts-H $\cdot T$
have $\bigwedge x. w \notin x \implies \text{proj-verts-H } \cdot x = x$ **by** (*auto simp: proj-verts-H-def*)
with A **have** $S \neq T \implies S \cap T \neq \{\}$
by (*metis H.in-sccs-verts-conv-reachable Int-iff empty-iff image-eqI proj-verts-H-def*
w-reach(1,2))
then show $S = T$ **using** *H.sccs-verts-disjoint[OF A(1,2)]* **by** *metis*
qed

lemma *card-sccs-verts: card G.sccs-verts = card H.sccs-verts*
unfolding *sccs-eq* **by** (*intro card-image inj-on-proj-verts-H*)

lemma *card-sccs-eq: card G.sccs = card H.sccs*
using *card-sccs-verts G.inj-on-verts-sccs H.inj-on-verts-sccs*
by (*auto simp: G.sccs-verts-conv H.sccs-verts-conv card-image*)

lemma *isolated-verts-eq: G.isolated-verts = H.isolated-verts*
by (*auto simp: G.isolated-verts-def H.isolated-verts-def verts-H out-arcs-w dest:*
out-arcs-empty)

lemma *card-verts: card (verts H) = card (verts G) + 1*
unfolding *verts-H* **using** *not-in-verts-G* **by** *auto*

lemma *card-arcs: card (arcs H) = card (arcs G) + 2*
unfolding *arcs-H* **using** *not-in-arcs-G subdiv-distinct-arcs in-arcs-G* **by** (*auto*
simp: card-insert-if)

lemma *edge-succ-wu: edge-succ HM wu = wv*
using *out-arcs-w out-degree-w edge-succ-permutes H.edge-succ-cyclic[of w]*
by (*auto elim: eq-on-cyclic-on-iff1[where x=wu] simp: verts-H out-degree-def*)

lemma *edge-succ-wv: edge-succ HM wv = wu*
using *out-arcs-w out-degree-w edge-succ-permutes H.edge-succ-cyclic[of w]*
by (*auto elim: eq-on-cyclic-on-iff1[where x=wv] simp: verts-H out-degree-def*)

lemmas *edge-succ-w = edge-succ-wu edge-succ-wv*

lemma *H-face-cycle-succ:*
H.face-cycle-succ uw = wv
H.face-cycle-succ wv = wu

unfolding H .face-cycle-succ-def **by** (auto simp: edge-succ-w)

lemma H -edge-succ-tail-eqD:

assumes edge-succ HM $a = b$ **shows** tail H $a =$ tail H b

using assms H .tail-edge-succ[of a] **by** auto

lemma YYY:

$(wu \Rightarrow_F vw) (edge-succ HM vw) = (edge-succ HM vw)$

$(wu \Rightarrow_F vw) (edge-succ HM uw) = (edge-succ HM uw)$

using H .edge-succ-cyclic[of w] subdiv-distinct-verts0

by (auto simp: Transposition.transpose-def dest: H -edge-succ-tail-eqD)

Project arcs of H to corresponding arcs of G

definition proj-arcs- H :: $'b \Rightarrow 'b$ **where**

proj-arcs- H $x \equiv$

if $x = uw \vee x = vw$ then uw

else if $x = vw \vee x = wu$ then vu

else x

Project arcs of G to corresponding arcs of H

definition proj-arcs- G :: $'b \Rightarrow 'b$ **where**

proj-arcs- G $x \equiv$

if $x = uw$ then uw

else if $x = vu$ then vw

else x

lemma proj-arcs- H -simps[simp]:

proj-arcs- H $uw = uw$

proj-arcs- H $wv = uv$

proj-arcs- H $vw = vu$

proj-arcs- H $wu = vu$

$x \notin \{uw, vw, wu, wv\} \implies$ proj-arcs- H $x = x$

$a \in$ arcs $G \implies$ proj-arcs- H $a = a$

using subdiv-distinct-arcs not-in-arcs- G **by** (auto simp: proj-arcs- H -def)

lemma proj-arcs- H -in-arcs- G : $a \in$ arcs $H \implies$ proj-arcs- H $a \in$ arcs G

using subdiv-distinct-arcs in-arcs- G **by** (auto simp: proj-arcs- H -def arcs- H)

lemma proj-arcs-eq-swap:

assumes $a \notin \{uv, vu, wu, wv\}$

shows proj-arcs- H $a = (uw \Rightarrow_F uv \circ vw \Rightarrow_F vu) a$

using assms subdiv-distinct-arcs **by** (cases $a \in \{uw, vw\}$) auto

lemma proj-arcs- G -simps:

proj-arcs- G $uv = uw$

proj-arcs- G $vu = vw$

$a \notin \{uv, vu\} \implies$ proj-arcs- G $a = a$

using subdiv-distinct-arcs not-in-arcs- G **by** (auto simp: swap-id-eq proj-arcs- G -def)

lemma *proj-arcs-G-in-arcs-H*:
assumes $a \in \text{arcs } G$ **shows** $\text{proj-arcs-G } a \in \text{arcs } H$
using *assms subdiv-distinct-arcs* **by** (*auto simp: proj-arcs-G-def arcs-H*)

lemma *proj-arcs-HG*: $a \in \text{arcs } G \implies \text{proj-arcs-H } (\text{proj-arcs-G } a) = a$
by (*auto simp: proj-arcs-G-def*)

lemma *fcs-proj-arcs-GH*:
assumes $a \in \text{arcs } H$ **shows** $H.\text{face-cycle-set } (\text{proj-arcs-G } (\text{proj-arcs-H } a)) = H.\text{face-cycle-set } a$
proof –
have $H.\text{face-cycle-set } vw = H.\text{face-cycle-set } wu \ H.\text{face-cycle-set } uw = H.\text{face-cycle-set } wv$
unfolding *H.face-cycle-set-def* **by** (*auto simp add: H-face-cycle-succ[symmetric] self-in-orbit-step H.permutation-face-cycle-succ permutation-self-in-orbit*)
then show *?thesis*
using *assms not-in-arcs-H* **by** (*cases a \in \{uv, vu, uw, wu, vw, wv\}*) (*auto simp: proj-arcs-G-simps*)
qed

lemma *H-face-cycle-succ-neq-uv*:
 $a \notin \{uv, vu\} \implies H.\text{face-cycle-succ } a \notin \{uv, vu\}$
using *not-in-arcs-H* **by** (*cases a \in \text{arcs } H*) (*auto dest: H.face-cycle-succ-in-arcsI*)

lemma *face-cycle-succ-choose-inter*:
 $\{H.\text{face-cycle-succ } uw, H.\text{face-cycle-succ } vw, H.\text{face-cycle-succ } wu, H.\text{face-cycle-succ } wv\} \cap \{uv, vu\} = \{\}$
using *subdiv-distinct-arcs H-face-cycle-succ-neq-uv* **by** *safe (simp-all,metis+)*

lemma *face-cycle-succ-choose-neq*:
 $H.\text{face-cycle-succ } wu \notin \{wu, wv\}$
 $H.\text{face-cycle-succ } wv \notin \{wu, wv\}$
using *subdiv-distinct-verts0 in-arcs-H*
by (*auto simp del: H.edge-rev-in-arcs dest: H.tail-face-cycle-succ*)

lemma *H-face-cycle-succ-G-not-in*:
assumes $a \in \text{arcs } G$ **shows** $H.\text{face-cycle-succ } a \notin \{wu, wv\}$
proof (*cases a \in \{uv, vu\}*)
case *True* **with** *assms* **show** *?thesis* **using** *subdiv-distinct-arcs* **by** (*auto simp: arcs-H*)
next
case *False* **with** *assms* **have** $a \in \text{arcs } H$ **by** (*auto simp: arcs-H*)
from *assms* **have** $\text{head } H \ a \neq w$ **by** (*auto simp: head-eq verts-G arcs-H dest: G.head-in-verts*)
then show *?thesis* **using** *H.tail-face-cycle-succ[OF \langle a \in \text{arcs } H \rangle]* **by** *auto*
qed

lemma
face-cycle-succ-uv: $GM.\text{face-cycle-succ } uv = \text{proj-arcs-H } (H.\text{face-cycle-succ } wv)$

and

face-cycle-succ-vu: $GM.face-cycle-succ\ vu = proj-arcs-H\ (H.face-cycle-succ\ wu)$
unfolding $GM.face-cycle-succ-def\ edge-rev-GM\ edge-succ-GM$
using *face-cycle-succ-choose-neq face-cycle-succ-choose-inter subdiv-distinct-arcs*
apply (*auto simp: fun-eq-iff perm-swap-def*)
apply (*auto simp: perm-rem-def edge-succ-w H.face-cycle-succ-def YYY proj-arcs-H-def*)
done

lemma *face-cycle-succ-not-w*:

assumes $a \in arcs\ G\ a \notin \{w, vu\}$

shows $GM.face-cycle-succ\ a = proj-arcs-H\ (H.face-cycle-succ\ a)$

proof –

have $GM.face-cycle-succ\ a = (uw \Rightarrow_F w) ((vw \Rightarrow_F rev-G\ w) (perm-rem\ (wv)$
 $(perm-rem\ (wu)\ (edge-succ\ HM)) (((vw \Rightarrow_F vu)\ ((uw \Rightarrow_F w)\ (rev-G\ a))))))$
by (*simp add: GM.face-cycle-succ-def perm-swap-def edge-succ-GM edge-rev-GM*)
also have $(vw \Rightarrow_F vu)\ ((uw \Rightarrow_F w)\ (rev-G\ a)) = rev-G\ a$
using *assms not-in-arcs-G* **by** (*auto simp: transpose-def G.arev-eq-iff*)
also have $perm-rem\ (wv)\ (perm-rem\ (wu)\ (edge-succ\ HM))\ (rev-G\ a) =$
 $edge-succ\ HM\ (rev-G\ a)$

proof –

have $*$: $\bigwedge a. tail\ H\ a \neq w \implies (wu \Rightarrow_F w)\ a = a$ **by** (*auto simp: transpose-def*)

from *assms* **have** $head\ H\ a \neq w\ tail\ H\ (rev-G\ a) = head\ H\ a$

by (*auto simp: tail-eq head-eq verts-G dest: G.head-in-verts*)

then have $((wu \Rightarrow_F w)\ (edge-succ\ HM\ (rev-G\ a))) = edge-succ\ HM\ (rev-G$

$a)$

by (*intro **) *auto*

then show *?thesis* **by** (*auto simp: perm-rem-def edge-succ-w*)

qed

also have $edge-succ\ HM\ (rev-G\ a) = H.face-cycle-succ\ a$

using *assms* **unfolding** $H.face-cycle-succ-def$ **by** (*simp add: rev-H-eq-rev-G*)

also have $(uw \Rightarrow_F w)\ ((vw \Rightarrow_F rev-G\ w)\ (H.face-cycle-succ\ a)) = proj-arcs-H$
 $(H.face-cycle-succ\ a)$

proof –

from *assms* **have** $a \in arcs\ H$ **by** (*auto simp: arcs-H*)

then have *fcs-not-in*: $H.face-cycle-succ\ a \notin \{w, vu, wv, vw\}$

using *assms H-face-cycle-succ-G-not-in in-arcs-G not-in-arcs-H*

by (*auto simp del: G.arev-in-arcs dest: H.face-cycle-succ-closed[THEN*
 $iffD2]$)

then show *?thesis* **by** (*auto simp add: proj-arcs-eq-swap*)

qed

finally show *?thesis* .

qed

lemmas $G.face-cycle-succ = face-cycle-succ-w\ face-cycle-succ-vu\ face-cycle-succ-not-w$

lemma *in-G-fcs-in-H-fcs*:

assumes $a \in arcs\ G$

assumes $x \in GM.face-cycle-set\ a$

shows $x \in proj-arcs-H\ ' H.face-cycle-set\ (proj-arcs-G\ a)$

```

using ⟨ $x \in \cdot$ ⟩
proof induct
  case base show ?case
    by (rule rev-image-eqI[where  $x = \text{proj-arcs-}G\ a$ ]) (auto simp: ⟨ $a \in \text{arcs } G$ ⟩
proj-arcs- $G$ -def)
  next
    case (step  $b$ )
    { fix  $x$  assume  $x \in H.\text{face-cycle-set } (\text{proj-arcs-}G\ a)$ 
      then have  $x \in \text{arcs } H$ 
      using ⟨ $a \in \text{arcs } G$ ⟩ by (auto dest:  $H.\text{in-face-cycle-setD}$  simp:  $\text{proj-arcs-}G\ \text{in-arcs-}H$ )
      moreover
      then have  $x \notin \{uv, vu\} \implies x \notin \{uw, wu, vw, wv\} \implies x \in \text{arcs } G$ 
        using ⟨ $a \in \text{arcs } G$ ⟩ by (auto simp:  $\text{arcs-}H$  dest:  $H.\text{in-face-cycle-setD}$ )
      ultimately
      have  $GM.\text{face-cycle-succ } (\text{proj-arcs-}H\ x) \in \{\text{proj-arcs-}H\ (H.\text{face-cycle-succ } x),$ 
         $\text{proj-arcs-}H\ (H.\text{face-cycle-succ } (H.\text{face-cycle-succ } x))\}$ 
      by (cases  $x \in \{uw, vw, wu, wv\}$ ) (auto simp:  $G.\text{face-cycle-succ } H.\text{face-cycle-succ}$ )
    }
    moreover
    have  $b \in \text{arcs } G$ 
    using step(1) ⟨ $a \in \text{arcs } G$ ⟩ by (simp add:  $GM.\text{in-face-cycle-setD } GM.\text{face-cycle-set-def}$ )
    ultimately
    show ?case using ⟨ $b \in \text{arcs } G$ ⟩ step(2) by (auto intro:  $H.\text{face-cycle-succ-inI}$ )
qed

```

```

lemma in- $H$ -fcs-in- $G$ -fcs:
  assumes  $a \in \text{arcs } H$ 
  assumes  $x \in H.\text{face-cycle-set } a$ 
  shows  $x \in \text{proj-arcs-}H \iff GM.\text{face-cycle-set } (\text{proj-arcs-}H\ a)$ 
  using ⟨ $x \in \cdot$ ⟩
proof induct
  case base then show ?case by auto
  next
    case (step  $y$ )
    then have  $y \in \text{arcs } H$  using ⟨ $a \in \text{arcs } H$ ⟩ by (auto dest:  $H.\text{in-face-cycle-setD}$ )
    moreover then have  $y \notin \{uv, vu\}$  by (fastforce simp:  $\text{arcs-}H$ )
    ultimately have  $\text{proj-arcs-}H\ (H.\text{face-cycle-succ } y) = GM.\text{face-cycle-succ } (\text{proj-arcs-}H\ y)$ 
       $\vee \text{proj-arcs-}H\ (H.\text{face-cycle-succ } y) = \text{proj-arcs-}H\ y$ 
    by (cases  $y \in \{uw, vw, wv, wu\}$ ) (auto simp:  $H.\text{face-cycle-succ } G.\text{face-cycle-succ } \text{arcs-}G$ )
    with step show ?case by (auto intro:  $GM.\text{face-cycle-succ-inI}$ )
qed

```

```

lemma  $G$ -fcs-eq:
  assumes  $a \in \text{arcs } G$ 
  shows  $GM.\text{face-cycle-set } a = \text{proj-arcs-}H \iff H.\text{face-cycle-set } (\text{proj-arcs-}G\ a)$  (is
? $L = ?R$ )

```

using *assms* **by** (*auto dest: in-H-fcs-in-G-fcs[rotated] in-G-fcs-in-H-fcs[rotated]*
simp: proj-arcs-G-in-arcs-H proj-arcs-HG)

lemma *H-fcs-eq*:

assumes $a \in \text{arcs } H$

shows $\text{proj-arcs-}H \text{ ' } H.\text{face-cycle-set } a = GM.\text{face-cycle-set } (\text{proj-arcs-}H \ a)$

using *assms* **by** (*auto dest: in-H-fcs-in-G-fcs[rotated] in-G-fcs-in-H-fcs[rotated]*
simp: proj-arcs-H-in-arcs-G fcs-proj-arcs-GH)

lemma *face-cycle-sets*:

shows $GM.\text{face-cycle-sets} = (\cdot) \text{ proj-arcs-}H \text{ ' } H.\text{face-cycle-sets}$ (**is** ?*L* = ?*R*)

unfolding *GM.face-cycle-sets-def H.face-cycle-sets-def*

by (*blast intro!: H-fcs-eq G-fcs-eq proj-arcs-G-in-arcs-H proj-arcs-H-in-arcs-G*)

lemma *inj-on-proj-arcs-H*: *inj-on* ((\cdot) *proj-arcs-H*) *H.face-cycle-sets*

proof (*rule inj-onI*)

fix *A B* **assume** *fcs*: $A \in H.\text{face-cycle-sets}$ $B \in H.\text{face-cycle-sets}$

and *pa-eq*: $\text{proj-arcs-}H \text{ ' } A = \text{proj-arcs-}H \text{ ' } B$

have *xw-iff-wy*:

$\bigwedge X. X \in H.\text{face-cycle-sets} \implies uw \in X \longleftrightarrow vw \in X$

$\bigwedge X. X \in H.\text{face-cycle-sets} \implies vw \in X \longleftrightarrow wu \in X$

using *H-face-cycle-succ* **by** (*auto simp: H.face-cycle-sets-def dest: H.face-cycle-succ-inI*
intro: H.face-cycle-succ-inD)

have *not-in-A*: $uv \notin A$ $vu \notin A$ **and** *not-in-B*: $vu \notin B$ $wv \notin B$

using *fcs not-in-arcs-H* **by** (*auto dest: H.in-face-cycle-setsD*)

have $A = \text{proj-arcs-}H - \{ (\text{proj-arcs-}H \text{ ' } A) - \{uv, vu\} \}$

using *subdiv-distinct-arcs not-in-A* **by** (*auto simp: proj-arcs-H-def xw-iff-wy[OF*
fcs(1)] split: if-splits)

also have $\dots = \text{proj-arcs-}H - \{ (\text{proj-arcs-}H \text{ ' } B) - \{uv, vu\} \}$ **by** (*simp add:*
pa-eq)

also have $\dots = B$

using *subdiv-distinct-arcs not-in-B* **by** (*auto simp: proj-arcs-H-def xw-iff-wy[OF*
fcs(2)] split: if-splits)

finally show $A = B$.

qed

lemma *card-face-cycle-sets*: $\text{card } GM.\text{face-cycle-sets} = \text{card } H.\text{face-cycle-sets}$

unfolding *face-cycle-sets* **using** *inj-on-proj-arcs-H* **by** (*rule card-image*)

lemma *euler-char-eq*: $GM.\text{euler-char} = H.\text{euler-char}$

by (*auto simp: GM.euler-char-def H.euler-char-def card-verts card-arcs card-face-cycle-sets*)

lemma *euler-genus-eq*: $GM.\text{euler-genus} = H.\text{euler-genus}$

by (*auto simp: GM.euler-genus-def H.euler-genus-def euler-char-eq card-sccs-eq*
isolated-verts-eq)

end

lemma *subdivision-genus-same-rev*:

assumes *subdivision* (G , *rev-G*) (H , *edge-rev HM*) *digraph-map H HM pre-digraph-map.euler-genus H HM = m*

shows $\exists GM. \text{digraph-map } G \text{ } GM \wedge \text{pre-digraph-map.euler-genus } G \text{ } GM = m \wedge \text{edge-rev } GM = \text{rev-}G$

proof –

from *assms* **show** *?thesis*

proof (*induction rev-H \equiv edge-rev HM arbitrary: HM*)

case *base* **then show** *?case* **by** *auto*

next

case (*divide I rev-I H u v w uv uw vw*)

then interpret *subdiv-step I rev-I H edge-rev HM u v w uv uw vw*

by *unfold-locales simp*

interpret $H: \text{digraph-map } H \text{ } HM$ **using** $\langle \text{digraph-map } H \text{ } HM \rangle$.

interpret $IH: \text{subdiv1-contr } I \text{ } rev-I \text{ } H \text{ } edge-rev \text{ } HM \text{ } u \text{ } v \text{ } w \text{ } uv \text{ } uw \text{ } vw \text{ } HM$

by *unfold-locales simp*

have $eulerI: IH.GM.euler-genus = m$ **by** (*auto simp: IH.euler-genus-eq divide*)

with $- IH.digraph-map-GM$ **show** *?case* **by** (*rule divide*) (*simp add: IH.edge-rev-GM*)

qed

qed

lemma *subdivision-genus*:

assumes *subdivision* (G , *rev-G*) (H , *rev-H*) *digraph-map H HM pre-digraph-map.euler-genus H HM = m*

shows $\exists GM. \text{digraph-map } G \text{ } GM \wedge \text{pre-digraph-map.euler-genus } G \text{ } GM = m$

proof –

interpret $H: \text{digraph-map } H \text{ } HM$ **by** *fact*

show *?thesis*

using *subdivision-genus-same-rev subdivision-choose-rev assms H.bidirected-digraph*

by *metis*

qed

lemma *subdivision-comb-planar*:

assumes *subdivision* (G , *rev-G*) (H , *rev-H*) *comb-planar H* **shows** *comb-planar G*

using *assms unfolding comb-planar-def* **by** (*metis subdivision-genus*)

end

theory *Planar-Subgraph*

imports

Graph-Genus

Permutations-2

HOL-Library.FuncSet

HOL-Library.Simps-Case-Conv

begin

14 Combinatorial Planarity and Subgraphs

lemma *out-arcs-emptyD-dominates*:

assumes *out-arcs* G $x = \{\}$ **shows** $\neg x \rightarrow_G y$
using *assms* **by** (*auto simp: out-arcs-def*)

lemma (**in** *wf-digraph*) *reachable-refl-iff*: $u \rightarrow^* u \iff u \in \text{verts } G$
by (*auto simp: reachable-in-verts*)

context *digraph-map* **begin**

lemma *face-cycle-set-succ[simp]*: $\text{face-cycle-set } (\text{face-cycle-succ } a) = \text{face-cycle-set } a$
by (*metis face-cycle-eq face-cycle-set-self face-cycle-succ-inD*)

lemma *face-cycle-succ-funpow-in[simp]*:
 $(\text{face-cycle-succ } \tilde{n}) a \in \text{arcs } G \iff a \in \text{arcs } G$
by (*induct n*) *auto*

lemma *segment-face-cycle-x-x-eq*:
 $\text{segment } \text{face-cycle-succ } x \ x = \text{face-cycle-set } x - \{x\}$
unfolding *face-cycle-set-def* **using** *face-cycle-succ-permutes finite-arcs permutation-permutes*
by (*intro segment-x-x-eq*) *blast*

lemma *fcs-x-eq-x*: $\text{face-cycle-succ } x = x \iff \text{face-cycle-set } x = \{x\}$ (**is** $?L \iff ?R$)
unfolding *face-cycle-set-def orbit-eq-singleton-iff* ..

end

lemma (**in** *bidirected-digraph*) *bidirected-digraph-del-arc*:

$\text{bidirected-digraph } (\text{pre-digraph.del-arc } (\text{pre-digraph.del-arc } G (\text{arev } a)) a) (\text{perm-restrict arev } (\text{arcs } G - \{a, \text{arev } a\}))$

proof *unfold-locales*

fix b **assume** A : $b \in \text{arcs } (\text{pre-digraph.del-arc } (\text{del-arc } (\text{arev } a)) a)$

have $\text{arev } b \neq b \implies b \neq \text{arev } a \implies b \neq a \implies \text{perm-restrict arev } (\text{arcs } G - \{a, \text{arev } a\}) (\text{arev } b) = b$

using *bij-arev arev-dom* **by** (*subst perm-restrict-simps*) (*auto simp: bij-iff*)

then show $\text{perm-restrict arev } (\text{arcs } G - \{a, \text{arev } a\}) (\text{perm-restrict arev } (\text{arcs } G - \{a, \text{arev } a\}) b) = b$

using A

by (*auto simp: pre-digraph.del-arc-simps perm-restrict-simps arev-dom*)

qed (*auto simp: pre-digraph.del-arc-simps perm-restrict-simps arev-dom*)

lemma (**in** *bidirected-digraph*) *bidirected-digraph-del-vert*: $\text{bidirected-digraph } (\text{del-vert } u) (\text{perm-restrict arev } (\text{del-vert } u))$

by *unfold-locales* (*auto simp: del-vert-simps perm-restrict-simps arev-dom*)

lemma (in *pre-digraph*) *ends-del-arc*: $\text{arc-to-ends } (\text{del-arc } u) = \text{arc-to-ends } G$
by (*simp add: arc-to-ends-def fun-eq-iff*)

lemma (in *pre-digraph*) *dominates-arcsD*:
assumes $v \rightarrow_{\text{del-arc } u} w$ **shows** $v \rightarrow_G w$
using *assms* **by** (*auto simp: arcs-ends-def ends-del-arc*)

lemma (in *wf-digraph*) *reachable-del-arcD*:
assumes $v \rightarrow^*_{\text{del-arc } u} w$ **shows** $v \rightarrow^*_G w$
proof –
interpret *H*: *wf-digraph del-arc u* **by** (*rule wf-digraph-del-arc*)
from *assms* **show** *?thesis*
by (*induct*) (*auto dest: dominates-arcsD intro: adj-reachable-trans*)
qed

lemma (in *fin-digraph*) *finite-isolated-verts[intro!]*: *finite isolated-verts*
by (*auto simp: isolated-verts-def*)

lemma (in *wf-digraph*) *isolated-verts-in-sccs*:
assumes $u \in \text{isolated-verts}$ **shows** $\{u\} \in \text{sccs-verts}$
proof –
have $v = u$ **if** $u \rightarrow^*_G v$ **for** v
using *that* *assms* **by** *induct* (*auto simp: arcs-ends-def arc-to-ends-def isolated-verts-def*)
with *assms* **show** *?thesis* **by** (*auto simp: sccs-verts-def isolated-verts-def*)
qed

lemma (in *digraph-map*) *in-face-cycle-sets*:
 $a \in \text{arcs } G \implies \text{face-cycle-set } a \in \text{face-cycle-sets}$
by (*auto simp: face-cycle-sets-def*)

lemma (in *digraph-map*) *heads-face-cycle-set*:
assumes $a \in \text{arcs } G$
shows $\text{head } G \text{ ` face-cycle-set } a = \text{tail } G \text{ ` face-cycle-set } a$ (**is** $?L = ?R$)
proof (*intro set-eqI iffI*)
fix u **assume** $u \in ?L$
then obtain b **where** $b \in \text{face-cycle-set } a$ $\text{head } G \text{ } b = u$ **by** *blast*
then have $\text{face-cycle-succ } b \in \text{face-cycle-set } a$ $\text{tail } G \text{ } (\text{face-cycle-succ } b) = u$
using *assms* **by** (*auto simp: tail-face-cycle-succ face-cycle-succ-inI in-face-cycle-setD*)
then show $u \in ?R$ **by** *auto*
next
fix u **assume** $u \in ?R$
then obtain b **where** $b \in \text{face-cycle-set } a$ $\text{tail } G \text{ } b = u$ **by** *blast*
moreover
then obtain c **where** $b = \text{face-cycle-succ } c$ **by** (*metis face-cycle-succ-pred*)
ultimately
have $c \in \text{face-cycle-set } a$ $\text{head } G \text{ } c = u$
by (*auto dest: face-cycle-succ-inD*) (*metis assms face-cycle-succ-no-arc in-face-cycle-setD*)

tail-face-cycle-succ)
then show $u \in ?L$ **by** *auto*
qed

lemma (**in** *pre-digraph*) *casI-nth*:
assumes $p \neq []$ $u = \text{tail } G \text{ (hd } p)$ $v = \text{head } G \text{ (last } p) \wedge i. \text{Suc } i < \text{length } p \implies$
 $\text{head } G \text{ (p ! } i) = \text{tail } G \text{ (p ! Suc } i)$
shows $\text{cas } u \text{ } p \text{ } v$
using *assms*
proof (*induct p arbitrary: u*)
case Nil then show *?case* **by** *simp*
next
case (*Cons a p*)
have $\text{cas } (\text{head } G \text{ } a) \text{ } p \text{ } v$
proof (*cases p = []*)
case False then show *?thesis*
using *Cons.prem1-3* *Cons.prem4[of 0]* *Cons.prem4[of Suc i for i]*
by (*intro Cons*) (*simp-all add: hd-conv-nth*)
qed (*simp add: Cons*)
with Cons show *?case* **by** *simp*
qed

lemma (**in** *digraph-map*) *obtain-trail-in-fcs*:
assumes $a \in \text{arcs } G$ $a0 \in \text{face-cycle-set } a$ $an \in \text{face-cycle-set } a$
obtains p **where** $\text{trail } (\text{tail } G \text{ } a0) \text{ } p \text{ (head } G \text{ } an) \text{ } p \neq []$ $\text{hd } p = a0$ $\text{last } p = an$
 $\text{set } p \subseteq \text{face-cycle-set } a$
proof –
have *fcs-a*: $\text{face-cycle-set } a = \text{orbit } \text{face-cycle-succ } a0$
using *assms face-cycle-eq* **by** (*simp add: face-cycle-set-def*)
have $a0 = (\text{face-cycle-succ } \sim 0) \text{ } a0$ **by** *simp*
have $an = (\text{face-cycle-succ } \sim \text{funpow-dist } \text{face-cycle-succ } a0 \text{ } an) \text{ } a0$
using *assms* **by** (*simp add: fcs-a funpow-dist-prop*)

define p **where** $p = \text{map } (\lambda n. (\text{face-cycle-succ } \sim n) \text{ } a0) [0..<\text{Suc } (\text{funpow-dist } \text{face-cycle-succ } a0 \text{ } an)]$
have *p-nth*: $\bigwedge i. i < \text{length } p \implies p ! i = (\text{face-cycle-succ } \sim i) \text{ } a0$
by (*auto simp: p-def simp del: upt-Suc*)

have *P2*: $p \neq []$ **by** (*simp add: p-def*)
have *P3*: $\text{hd } p = a0$ **using** $\langle a0 = \rightarrow \rangle$ **by** (*auto simp: p-def hd-map simp del: upt-Suc*)
have *P4*: $\text{last } p = an$ **using** $\langle an = \rightarrow \rangle$ **by** (*simp add: p-def*)
have *P5*: $\text{set } p \subseteq \text{face-cycle-set } a$
unfolding *p-def fcs-a orbit-altdef-permutation[OF permutation-face-cycle-succ]*
by *auto*

have *P1*: $\text{trail } (\text{tail } G \text{ } a0) \text{ } p \text{ (head } G \text{ } an)$
proof –
have *distinct p*

proof –
have $an \in \text{orbit face-cycle-succ } a0$ **using** assms **by** (simp add: fcs-a)
then have $\text{inj-on } (\lambda n. (\text{face-cycle-succ } \sim n) a0) \{0..\text{funpow-dist face-cycle-succ } a0\}$
by $(\text{rule inj-on-funpow-dist})$
also have $\{0..\text{funpow-dist face-cycle-succ } a0\} = (\text{set } [0..\text{Suc } (\text{funpow-dist face-cycle-succ } a0)])$
by auto
finally have $\text{inj-on } (\lambda n. (\text{face-cycle-succ } \sim n) a0) (\text{set } [0..\text{Suc } (\text{funpow-dist face-cycle-succ } a0)])$.
then show $\text{distinct } p$ **by** $(\text{simp add: distinct-map p-def})$
qed
moreover
have $a0 \in \text{arcs } G$ **by** $(\text{metis assms}(1-2) \text{ in-face-cycle-setD})$
then have $\text{tail } G a0 \in \text{verts } G$ **by** simp
moreover
have $\text{set } p \subseteq \text{arcs } G$ **using** $P5$
by $(\text{metis assms}(1) \text{ in-face-cycle-setD subset-code}(1))$
moreover
then have $\bigwedge i. \text{Suc } i < \text{length } p \implies p ! \text{Suc } i \in \text{arcs } G$ **by** auto
then have $\bigwedge i. \text{Suc } i < \text{length } p \implies \text{head } G (p ! i) = \text{tail } G (p ! \text{Suc } i)$
by $(\text{auto simp: p-nth tail-face-cycle-succ})$
ultimately
show $?thesis$
using $P2 P3 P4$ **unfolding** $\text{trail-def awalk-def}$ **by** $(\text{auto intro: casI-nth})$
qed

from $P1 P2 P3 P4 P5$ **show** $?thesis ..$
qed

lemma (in digraph-map) $\text{obtain-trail-in-fcs'}$:
assumes $a \in \text{arcs } G$ $u \in \text{tail } G$ $\text{face-cycle-set } a \in \text{tail } G$ $\text{face-cycle-set } a$
obtains p **where** $\text{trail } u p \in \text{set } p \subseteq \text{face-cycle-set } a$
proof –
from assms **obtain** $a0$ **where** $\text{tail } G a0 = u$ $a0 \in \text{face-cycle-set } a$ **by** auto
moreover
from assms **obtain** an **where** $\text{head } G an = v$ $an \in \text{face-cycle-set } a$
by $(\text{auto simp: heads-face-cycle-set[symmetric]})$
ultimately obtain p **where** $\text{trail } u p \in \text{set } p \subseteq \text{face-cycle-set } a$
using $\langle a \in \text{arcs } G \rangle$ **by** $(\text{metis obtain-trail-in-fcs})$
then show $?thesis ..$
qed

14.1 Deleting an isolated vertex

locale $\text{del-vert-props} = \text{digraph-map} +$
fixes u
assumes $u\text{-in: } u \in \text{verts } G$
assumes $u\text{-isolated: } \text{out-arcs } G u = \{\}$

```

begin

lemma u-isolated-in: in-arcs G u = {}
  using u-isolated by (simp add: in-arcs-eq)

lemma arcs-dv: arcs (del-vert u) = arcs G
  using u-isolated u-isolated-in by (auto simp: del-vert-simps)

lemma out-arcs-dv: out-arcs (del-vert u) = out-arcs G
  by (auto simp: fun-eq-iff arcs-dv tail-del-vert)

lemma digraph-map-del-vert:
  shows digraph-map (del-vert u) M
proof -
  have perm-restrict (edge-rev M) (arcs (del-vert u)) = edge-rev M
    using has-dom-arev arcs-dv by (auto simp: perm-restrict-dom-subset)
  then interpret H: bidirected-digraph del-vert u edge-rev M
    using bidirected-digraph-del-vert[of u] by simp
  show ?thesis
  by unfold-locales (auto simp: arcs-dv edge-succ-permutes out-arcs-dv edge-succ-cyclic
verts-del-vert)
qed

end

sublocale del-vert-props  $\subseteq$  H: digraph-map del-vert u M by (rule digraph-map-del-vert)

context del-vert-props begin

lemma card-verts-dv: card (verts G) = Suc (card (verts (del-vert u)))
  by (auto simp: verts-del-vert) (rule card.remove, auto simp: u-in)

lemma card-arcs-dv: card (arcs (del-vert u)) = card (arcs G)
  using u-isolated by (auto simp add: arcs-dv in-arcs-eq)

lemma isolated-verts-dv: H.isolated-verts = isolated-verts - {u}
  by (auto simp: isolated-verts-def H.isolated-verts-def verts-del-vert out-arcs-dv)

lemma u-in-isolated-verts: u  $\in$  isolated-verts
  using u-in u-isolated by (auto simp: isolated-verts-def)

lemma card-isolated-verts-dv: card isolated-verts = Suc (card H.isolated-verts)
  by (simp add: isolated-verts-dv) (rule card.remove, auto simp: u-in-isolated-verts)

lemma face-cycles-dv: H.face-cycle-sets = face-cycle-sets
  unfolding H.face-cycle-sets-def face-cycle-sets-def arcs-dv ..

lemma euler-char-dv: euler-char = 1 + H.euler-char

```

by (auto simp: euler-char-def H.euler-char-def card-arcs-dv card-verts-dv face-cycles-dv)

lemma *adj-dv*: $v \rightarrow_{del\text{-}vert\ u} w \iff v \rightarrow_G w$
by (auto simp: arcs-ends-def arcs-dv ends-del-vert)

lemma *reachable-del-vertD*:
assumes $v \rightarrow^*_{del\text{-}vert\ u} w$ **shows** $v \rightarrow^*_G w$
using *assms* by induct (auto simp: verts-del-vert adj-dv intro: adj-reachable-trans)

lemma *reachable-del-vertI*:
assumes $v \rightarrow^*_G w$ $u \neq v \vee u \neq w$ **shows** $v \rightarrow^*_{del\text{-}vert\ u} w$
using *assms*
proof induct
case (step $x\ y$)
from $\langle x \rightarrow_G y \rangle$ **obtain** a **where** $a \in arcs\ G$ $head\ G\ a = y$ **by** *auto*
then have $a \in in\text{-}arcs\ G\ y$ **by** *auto*
then have $y \neq u$ **using** *u-isolated in-arcs-eq[of u]* **by** *auto*
with step show ?case **by** (auto simp: adj-dv intro: H.adj-reachable-trans)
qed (auto simp: verts-del-vert)

lemma *G-reach-conv*: $v \rightarrow^*_G w \iff v \rightarrow^*_{del\text{-}vert\ u} w \vee (v = u \wedge w = u)$
by (auto dest: reachable-del-vertI reachable-del-vertD intro: u-in)

lemma *sccs-verts-dv*: $H.sccs\text{-}verts = sccs\text{-}verts - \{\{u\}\}$ (is ?L = ?R)
proof –
have $*\bigwedge S\ x.\ S \in sccs\text{-}verts \implies S \notin H.sccs\text{-}verts \implies x \in S \implies x = u$
by (simp add: H.in-sccs-verts-conv-reachable in-sccs-verts-conv-reachable
G-reach-conv)
(meson H.reachable-trans)
show ?thesis
by (auto dest: *) (auto simp: H.in-sccs-verts-conv-reachable in-sccs-verts-conv-reachable
G-reach-conv H.reachable-refl-iff verts-del-vert)
qed

lemma *card-sccs-verts-dv*: $card\ sccs\text{-}verts = Suc\ (card\ H.sccs\text{-}verts)$
unfolding *sccs-verts-dv*
by (rule card.remove) (auto simp: isolated-verts-in-sccs u-in-isolated-verts fi-
nite-sccs-verts)

lemma *card-sccs-dv*: $card\ sccs = Suc\ (card\ H.sccs)$
using *card-sccs-verts-dv* by (simp add: card-sccs-verts H.card-sccs-verts)

lemma *euler-genus-eq*: $H.euler\text{-}genus = euler\text{-}genus$
by (auto simp: pre-digraph-map.euler-genus-def card-sccs-dv card-isolated-verts-dv
euler-char-dv)

end

14.2 Deleting an arc pair

locale *bidel-arc* = *G*: *digraph-map* +
fixes *a*
assumes *a-in*: $a \in \text{arcs } G$

begin

abbreviation $a' \equiv \text{edge-rev } M \ a$

definition $H :: ('a, 'b) \text{ pre-digraph where}$
 $H \equiv \text{pre-digraph.del-arc } (\text{pre-digraph.del-arc } G \ a') \ a$

definition $HM :: 'b \text{ pre-map where}$

$HM =$
 $($ $\text{edge-rev} = \text{perm-restrict } (\text{edge-rev } M) \ (\text{arcs } G - \{a, a'\})$
 $;$ $\text{edge-succ} = \text{perm-rem } a \ (\text{perm-rem } a' \ (\text{edge-succ } M))$
 $)$

lemma

verts-H: $\text{verts } H = \text{verts } G$ **and**
arcs-H: $\text{arcs } H = \text{arcs } G - \{a, a'\}$ **and**
tail-H: $\text{tail } H = \text{tail } G$ **and**
head-H: $\text{head } H = \text{head } G$ **and**
ends-H: $\text{arc-to-ends } H = \text{arc-to-ends } G$ **and**
arcs-in: $\{a, a'\} \subseteq \text{arcs } G$ **and**
ends-in: $\{\text{tail } G \ a, \text{head } G \ a\} \subseteq \text{verts } G$
by (*auto simp*: *H-def pre-digraph.del-arc-simps a-in arc-to-ends-def*)

lemma *cyclic-on-edge-succ*:

assumes $x \in \text{verts } H \ \text{out-arcs } H \ x \neq \{\}$
shows *cyclic-on* (*edge-succ* *HM*) (*out-arcs* *H* *x*)

proof –

have *oa-H*: $\text{out-arcs } H \ x = (\text{out-arcs } G \ x - \{a'\}) - \{a\}$ **by** (*auto simp*: *arcs-H tail-H*)

have *cyclic-on* (*perm-rem* *a* (*perm-rem* *a'* (*edge-succ* *M*))) (*out-arcs* *G* *x* – $\{a'\}$ – $\{a\}$)

using *assms*

by (*intro cyclic-on-perm-rem G.edge-succ-cyclic*) (*auto simp*: *oa-H G.bij-edge-succ G.edge-succ-cyclic*)

then show *?thesis* **by** (*simp add*: *HM-def oa-H*)

qed

lemma *digraph-map*: *digraph-map* *H* *HM*

proof –

interpret *fin-digraph* *H* **unfolding** *H-def*

by (*rule fin-digraph.fin-digraph-del-arc*) (*rule G.fin-digraph-del-arc*)

interpret *bidirected-digraph* *H* *edge-rev* *HM* **unfolding** *H-def*

using *G.bidirected-digraph-del-arc[of a]* **by** (*auto simp*: *HM-def*)

have $*$: $\text{insert } a' \ (\text{insert } a \ (\text{arcs } H)) = \text{arcs } G$ **using** *a-in* **by** (*auto simp*:

```

arcs-H)
  have edge-succ HM permutes arcs H
  unfolding HM-def by (auto simp: * intro!: perm-rem-permutes G.edge-succ-permutes)
  moreover
  { fix v assume v ∈ verts H out-arcs H v ≠ {}
  then have cyclic-on (edge-succ HM) (out-arcs H v) by (rule cyclic-on-edge-succ)
  }
  ultimately
  show ?thesis by unfold-locales
qed

```

```

lemma rev-H: bidel-arc.H G M a' = H (is ?t1)
  and rev-HM: bidel-arc.HM G M a' = HM (is ?t2)
proof -
  interpret rev: bidel-arc G M a' using a-in by unfold-locales simp
  show ?t1
  by (rule pre-digraph.equality) (auto simp: rev.verts-H verts-H rev.arcs-H arcs-H
    rev.tail-H tail-H rev.head-H head-H)
  show ?t2 using G.edge-succ-permutes
  by (intro pre-map.equality) (auto simp: HM-def rev.HM-def insert-commute
    perm-rem-commutes permutes-conv-has-dom)
qed

```

end

sublocale bidel-arc \subseteq H: digraph-map H HM by (rule digraph-map)

context bidel-arc begin

```

lemma a-neq-a': a ≠ a'
  by (metis G.arev-neq a-in)

```

```

lemma
  arcs-G: arcs G = insert a (insert a' (arcs H)) and
  arcs-not-in: {a,a'} ∩ arcs H = {}
  using arcs-in by (auto simp: arcs-H)

```

```

lemma card-arcs-da: card (arcs G) = 2 + card (arcs H)
  using arcs-G arcs-not-in a-neq-a' by (auto simp: card-insert-if)

```

```

lemma cas-da: H.cas = G.cas

```

```

proof -
  { fix u p v have H.cas u p v = G.cas u p v
    by (induct p arbitrary: u) (simp-all add: tail-H head-H)
  } then show ?thesis by (simp add: fun-eq-iff)
qed

```

```

lemma reachable-daD:
  assumes v →*H w shows v →*G w

```

apply (rule $G.reachable-del-arcD$)
apply (rule $wf-digraph.reachable-del-arcD$)
apply (rule $G.wf-digraph-del-arc$)
using $assms$ **unfolding** $H-def$ **by** $assumption$

lemma $not-G-isolated-a$: $\{tail\ G\ a,\ head\ G\ a\} \cap G.isolated-verts = \{\}$
using $a-in\ G.in-arcs-eq[of\ head\ G\ a]$ **by** (auto simp: $G.isolated-verts-def$)

lemma $isolated-other-da$:
assumes $u \notin \{tail\ G\ a,\ head\ G\ a\}$ **shows** $u \in H.isolated-verts \iff u \in G.isolated-verts$
using $assms$ **by** (auto simp: $pre-digraph.isolated-verts-def\ verts-H\ arcs-H\ tail-H\ out-arcs-def$)

lemma $isolated-da-pre$: $H.isolated-verts = G.isolated-verts \cup$
 $(if\ tail\ G\ a \in H.isolated-verts\ then\ \{tail\ G\ a\}\ else\ \{\}) \cup$
 $(if\ head\ G\ a \in H.isolated-verts\ then\ \{head\ G\ a\}\ else\ \{\})$ (**is** $?L = ?R$)

proof (intro $set-eqI\ iffI$)
fix x **assume** $x \in ?L$ **then show** $x \in ?R$
by (cases $x \in \{tail\ G\ a,\ head\ G\ a\}$) (auto simp: $isolated-other-da$)
next
fix x **assume** $x \in ?R$ **then show** $x \in ?L$ **using** $not-G-isolated-a$
by (cases $x \in \{tail\ G\ a,\ head\ G\ a\}$) (auto simp: $isolated-other-da\ split: if-splits$)
qed

lemma $card-isolated-verts-da0$:
 $card\ H.isolated-verts = card\ G.isolated-verts + card\ (\{tail\ G\ a,\ head\ G\ a\} \cap H.isolated-verts)$
using $not-G-isolated-a$ **by** (subst $isolated-da-pre$) (auto simp: $card-insert-if\ G.finite-isolated-verts$)

lemma $segments-neq$:
assumes $segment\ G.face-cycle-succ\ a'\ a \neq \{\} \vee segment\ G.face-cycle-succ\ a\ a' \neq \{\}$
shows $segment\ G.face-cycle-succ\ a\ a' \neq segment\ G.face-cycle-succ\ a'\ a$
proof –
have $bij-fcs: bij\ G.face-cycle-succ$
using $G.face-cycle-succ-permutes$ **by** (auto simp: $permutes-conv-has-dom$)
show $?thesis$ **using** $segment-disj[OF\ a-neq-a'\ bij-fcs]$ $assms$ **by** auto
qed

lemma $H-fcs-eq-G-fcs$:
assumes $b \in arcs\ G\ \{b, G.face-cycle-succ\ b\} \cap \{a, a'\} = \{\}$
shows $H.face-cycle-succ\ b = G.face-cycle-succ\ b$
proof –
have $edge-rev\ M\ b \notin \{a, a'\}$
using $assms$ **by** auto (metis $G.arev-arev$)
then show $?thesis$
using $assms$ **unfolding** $G.face-cycle-succ-def\ H.face-cycle-succ-def$

by (auto simp: HM-def perm-restrict-simps perm-rem-simps G.bij-edge-succ)
qed

lemma *face-cycle-set-other-da*:

assumes $\{a, a'\} \cap G.\text{face-cycle-set } b = \{\}$ $b \in \text{arcs } G$

shows $H.\text{face-cycle-set } b = G.\text{face-cycle-set } b$

proof –

have $\bigwedge s. s \in G.\text{face-cycle-set } b \implies b \in \text{arcs } G \implies a \notin G.\text{face-cycle-set } b \implies a' \notin G.\text{face-cycle-set } b$

$\implies \text{pre-digraph-map.face-cycle-succ } HM \ s = G.\text{face-cycle-succ } s$

by (subst H-fcs-eq-G-fcs) (auto simp: G.in-face-cycle-setD G.face-cycle-succ-inI)

then show *?thesis*

using *assms unfolding pre-digraph-map.face-cycle-set-def*

by (intro orbit-cong) (auto simp add: pre-digraph-map.face-cycle-set-def[symmetric])

qed

lemma *in-face-cycle-set-other*:

assumes $S \in G.\text{face-cycle-sets } \{a, a'\} \cap S = \{\}$

shows $S \in H.\text{face-cycle-sets}$

proof –

from *assms obtain b where* $S = G.\text{face-cycle-set } b$ $b \in \text{arcs } G$

by (auto simp: G.face-cycle-sets-def)

with *assms have* $S = H.\text{face-cycle-set } b$ **by** (simp add: face-cycle-set-other-da)

moreover

with *assms have* $b \in \text{arcs } H$ **using** $\langle b \in \text{arcs } G \rangle$ **by** (auto simp: arcs-H)

ultimately show *?thesis* **by** (auto simp: H.face-cycle-sets-def)

qed

lemma *H-fcs-in-G-fcs*:

assumes $b \in \text{arcs } H - (G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a')$

shows $H.\text{face-cycle-set } b \in G.\text{face-cycle-sets} - \{G.\text{face-cycle-set } a, G.\text{face-cycle-set } a'\}$

proof –

have $H.\text{face-cycle-set } b = G.\text{face-cycle-set } b$

using *assms by* (intro face-cycle-set-other-da) (auto simp: arcs-H G.face-cycle-eq)

moreover have $G.\text{face-cycle-set } b \notin \{G.\text{face-cycle-set } a, G.\text{face-cycle-set } a'\}$

$b \in \text{arcs } G$

using $G.\text{face-cycle-eq}$ *assms by* (auto simp: arcs-H)

ultimately show *?thesis* **by** (auto simp: G.face-cycle-sets-def)

qed

lemma *face-cycle-sets-da0*:

$H.\text{face-cycle-sets} = G.\text{face-cycle-sets} - \{G.\text{face-cycle-set } a, G.\text{face-cycle-set } a'\}$

$\cup H.\text{face-cycle-set } ((G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a') - \{a, a'\})$ (is

?L = ?R)

proof (intro set-eqI iffI)

fix S **assume** $S \in ?L$

then obtain b **where** $S = H.\text{face-cycle-set } b$ $b \in \text{arcs } H$ **by** (auto simp: H.face-cycle-sets-def)

```

then show  $S \in ?R$ 
using arcs-not-in H-fcs-in-G-fcs by (cases  $b \in G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a'$ )
auto
next
fix  $S$  assume  $S \in ?R$ 
show  $S \in ?L$ 
proof (cases  $S \in G.\text{face-cycle-sets} - \{G.\text{face-cycle-set } a, G.\text{face-cycle-set } a'\}$ )
  case True
    then have  $S \cap \{a, a'\} = \{\}$  using G.face-cycle-set-parts by (auto simp:
G.face-cycle-sets-def)
    with True show ?thesis by (intro in-face-cycle-set-other) auto
  next
  case False
    then have  $S \in H.\text{face-cycle-set } ((G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a') - \{a, a'\})$ 
    using  $\langle S \in ?R \rangle$  by blast
    moreover have  $(G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a') - \{a, a'\} \subseteq \text{arcs } H$ 
    using a-in by (auto simp: arcs-H dest: G.in-face-cycle-setD)
    ultimately show ?thesis by (auto simp: H.face-cycle-sets-def)
qed
qed

```

```

lemma card-fcs-aa'-le:  $\text{card } \{G.\text{face-cycle-set } a, G.\text{face-cycle-set } a'\} \leq \text{card } G.\text{face-cycle-sets}$ 
using a-in by (intro card-mono) (auto simp: G.face-cycle-sets-def)

```

```

lemma card-face-cycle-sets-da0:
   $\text{card } H.\text{face-cycle-sets} = \text{card } G.\text{face-cycle-sets} - \text{card } \{G.\text{face-cycle-set } a, G.\text{face-cycle-set } a'\}$ 
   $+ \text{card } (H.\text{face-cycle-set } ((G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a') - \{a, a'\}))$ 
proof -
  have face-cycle-sets-inter:
     $(G.\text{face-cycle-sets} - \{G.\text{face-cycle-set } a, G.\text{face-cycle-set } a'\}) \cap H.\text{face-cycle-set } ((G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a') - \{a, a'\}) = \{\}$ 
    (is ?L  $\cap$  ?R = -)
  proof -
  define  $L R P$ 
    where  $L = ?L$  and  $R = ?R$  and  $P x \longleftrightarrow x \cap (G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a') = \{\}$ 
  for  $x$ 
  then have  $\bigwedge x. x \in L \implies P x \wedge x \in R \implies \neg P x$ 
  using G.face-cycle-set-parts by (auto simp: G.face-cycle-sets-def)
  then have  $L \cap R = \{\}$  by blast
  then show ?thesis unfolding L-def R-def .
qed
then show ?thesis using arcs-G
by (simp add: card-Diff-subset[symmetric] card-Un-disjoint[symmetric] G.in-face-cycle-sets face-cycle-sets-da0)
qed

```

end

locale *bidel-arc-same-face* = *bidel-arc* +
assumes *same-face*: $G.\text{face-cycle-set } a' = G.\text{face-cycle-set } a$
begin
lemma *a-in-o*: $a \in \text{orbit } G.\text{face-cycle-succ } a'$
unfolding $G.\text{face-cycle-set-def}[\text{symmetric}]$ **by** (*simp add: same-face*)

lemma *segment-a'-a-in*: $\text{segment } G.\text{face-cycle-succ } a' a \subseteq \text{arcs } H$ (**is** $?seg \subseteq -$)
proof –
have $?seg \subseteq G.\text{face-cycle-set } a'$ **by** (*auto simp: G.face-cycle-set-def segmentD-orbit*)
moreover have $G.\text{face-cycle-set } a' \subseteq \text{arcs } G$ **by** (*auto simp: G.face-cycle-set-altdef a-in*)
ultimately show *?thesis* **using** *a-in-o* **by** (*auto simp: arcs-H a-in not-in-segment1 not-in-segment2*)
qed

lemma *segment-a'-a-neD*:
assumes $\text{segment } G.\text{face-cycle-succ } a' a \neq \{\}$
shows $\text{segment } G.\text{face-cycle-succ } a' a \in H.\text{face-cycle-sets}$ (**is** $?seg \in -$)
proof –
let $?b = G.\text{face-cycle-succ } a'$

have *fcs-a-neq-a'*: $G.\text{face-cycle-succ } a' \neq a$ **by** (*metis assms segment1-empty*)

have *in-aG*: $\bigwedge x. x \in \text{segment } G.\text{face-cycle-succ } a' a \implies x \in \text{arcs } G - \{a, a'\}$
using *not-in-segment1 not-in-segment2 segment-a'-a-in* **by** (*auto simp: arcs-H*)

{ fix x **assume** $A: x \in \text{segment } G.\text{face-cycle-succ } a' a$ **and** $B: G.\text{face-cycle-succ } x \neq a$
from A **have** $G.\text{face-cycle-succ } x \neq a'$
proof *induct*
case *base* **then show** *?case*
by (*metis a-neq-a' G.face-cycle-set-self not-in-segment1 G.face-cycle-set-def same-face segment.intros*)
next
case *step* **then show** *?case* **by** (*metis a-in-o a-neq-a' not-in-segment1 segment.step*)
qed
with $A B$ **have** $\{x, G.\text{face-cycle-succ } x\} \cap \{a, a'\} = \{\}$
using *not-in-segment1[OF a-in-o] not-in-segment2[of a G.face-cycle-succ a']*
by *safe*
with *in-aG* **have** $H.\text{face-cycle-succ } x = G.\text{face-cycle-succ } x$ **by** (*intro H-fcs-eq-G-fcs*)
(*auto intro: A*)
} note *fcs-x-eq = this*

{ fix x **assume** $A: x \in \text{segment } G.\text{face-cycle-succ } a' a$ **and** $B: G.\text{face-cycle-succ } x \neq a$

$x = a$
have $G.\text{face-cycle-succ } a \neq a$ **using** $B \text{ in-}aG[OF A] G.\text{bij-face-cycle-succ}$ **by**
(auto simp: bij-eq-iff)
then have $\text{edge-succ } M a \neq \text{edge-rev } M a$
by *(metis a-in-o G.arev-arev comp-apply G.face-cycle-succ-def not-in-segmentI segment.base)*
then have $H.\text{face-cycle-succ } x = G.\text{face-cycle-succ } a'$
using $\text{in-}aG[OF A] B G.\text{bij-edge-succ}$ **unfolding** $H.\text{face-cycle-succ-def}$
 $G.\text{face-cycle-succ-def}$
by *(auto simp: HM-def perm-restrict-simps perm-rem-conv G.arev-eq-iff)*
} note $\text{fcs-last-}x\text{-eq} = \text{this}$

have $\text{segment } G.\text{face-cycle-succ } a' a = H.\text{face-cycle-set } ?b$
proof *(intro set-eqI iffI)*
fix x **assume** $x \in \text{segment } G.\text{face-cycle-succ } a' a$
then show $x \in H.\text{face-cycle-set } ?b$
proof *induct*
case base **then show** $?case$ **by** *auto*
next
case *(step x)* **then show** $?case$ **by** *(subst fcs-x-eq[symmetric]) (auto simp: H.face-cycle-succ-inI)*
qed
next
fix x **assume** $A: x \in H.\text{face-cycle-set } ?b$
then show $x \in \text{segment } G.\text{face-cycle-succ } a' a$
proof *induct*
case base **then show** $?case$ **by** *(intro segment.base fcs-a-neq-a')*
next
case *(step x)* **then show** $?case$ **using** fcs-a-neq-a'
by *(cases G.face-cycle-succ x = a) (auto simp: fcs-last-x-eq fcs-x-eq intro: segment.intros)*
qed
qed
then show $?thesis$ **using** $\text{segment-a'-}a\text{-in}$ **by** *(auto simp: H.face-cycle-sets-def)*
qed

lemma segment-a-a'-neD :
assumes $\text{segment } G.\text{face-cycle-succ } a a' \neq \{\}$
shows $\text{segment } G.\text{face-cycle-succ } a a' \in H.\text{face-cycle-sets}$
proof –
interpret $\text{rev: bidel-arc-same-face } G M a'$
using $a\text{-in same-face}$ **by** $\text{unfold-locales simp-all}$
from assms **show** $?thesis$ **using** $\text{rev.segment-a'-}a\text{-neD}$ **by** *(simp add: rev-H rev-HM)*
qed

lemma $H\text{-fcs-full}$:
assumes $SS \subseteq H.\text{face-cycle-sets}$ **shows** $H.\text{face-cycle-set } ' (\bigcup SS) = SS$
proof –

```

{ fix x assume x ∈ ∪ SS
  then obtain S where S ∈ SS x ∈ S S ∈ H.face-cycle-sets using assms by
auto
  then have H.face-cycle-set x = S
    using H.face-cycle-set-parts by (auto simp: H.face-cycle-sets-def)
  then have H.face-cycle-set x ∈ SS using ⟨S ∈ SS⟩ by auto
}
moreover
{ fix S assume S ∈ SS
  then obtain x where x ∈ arcs H S = H.face-cycle-set x x ∈ S
    using assms by (auto simp: H.face-cycle-sets-def)
  then have S ∈ H.face-cycle-set ‘ ∪ SS
    using ⟨S ∈ SS⟩ by auto
}
ultimately show ?thesis by auto
qed

```

lemma *card-fcs-gt-0*: $0 < \text{card } G.\text{face-cycle-sets}$
using *a-in* **by** (auto simp: card-gt-0-iff dest: G.in-face-cycle-sets)

lemma *card-face-cycle-sets-da'*:
 $\text{card } H.\text{face-cycle-sets} = \text{card } G.\text{face-cycle-sets} - 1$
 $+ \text{card } (\{\text{segment } G.\text{face-cycle-succ } a' a', \text{segment } G.\text{face-cycle-succ } a' a, \{\}\})$
 $- \{\{\}\}$
proof –
 have $G.\text{face-cycle-set } a$
 $= \{a, a'\} \cup \text{segment } G.\text{face-cycle-succ } a' a' \cup \text{segment } G.\text{face-cycle-succ } a' a$
using *a-neq-a' same-face* **by** (intro cyclic-split-segment) (auto simp: G.face-cycle-succ-cyclic)
then have *: $G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a' - \{a, a'\} = \text{segment } G.\text{face-cycle-succ } a' a' \cup \text{segment } G.\text{face-cycle-succ } a' a$
by (auto simp: same-face G.face-cycle-set-def[symmetric] not-in-segment1 not-in-segment2)

have **: $H.\text{face-cycle-set } ‘ (G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a' - \{a, a'\})$
 $= (\text{if } \text{segment } G.\text{face-cycle-succ } a' a' \neq \{\} \text{ then } \{\text{segment } G.\text{face-cycle-succ } a' a', \text{segment } G.\text{face-cycle-succ } a' a\} \text{ else } \{\})$
 $\cup (\text{if } \text{segment } G.\text{face-cycle-succ } a' a \neq \{\} \text{ then } \{\text{segment } G.\text{face-cycle-succ } a' a\} \text{ else } \{\})$
unfolding *
using *H-fcs-full*[of $\{\text{segment } G.\text{face-cycle-succ } a' a', \text{segment } G.\text{face-cycle-succ } a' a\}$]
using *H-fcs-full*[of $\{\text{segment } G.\text{face-cycle-succ } a' a'\}$]
using *H-fcs-full*[of $\{\text{segment } G.\text{face-cycle-succ } a' a\}$]
by (auto simp add: segment-a-a'-neD segment-a'-a-neD)
show ?thesis
unfolding *card-face-cycle-sets-da0* ** **by** (simp add: same-face card-insert-if)
qed

end

locale *bidel-arc-diff-face* = *bidel-arc* +
 assumes *diff-face*: $G.\text{face-cycle-set } a' \neq G.\text{face-cycle-set } a$
begin

definition *S* :: 'b set **where**

$S \equiv \text{segment } G.\text{face-cycle-succ } a \ a \cup \text{segment } G.\text{face-cycle-succ } a' \ a'$

lemma *diff-face-not-in*: $a \notin G.\text{face-cycle-set } a' \ a' \notin G.\text{face-cycle-set } a$
using *diff-face G.face-cycle-eq* **by** *auto*

lemma *H-fcs-eq-for-a*:

assumes $b \in \text{arcs } H \cap G.\text{face-cycle-set } a$
shows $H.\text{face-cycle-set } b = S$ (**is** ?*L* = ?*R*)

proof (*intro set-eqI iffI*)

fix *c* **assume** $c \in ?L$

then have $c \in \text{arcs } H$ **using** *assms* **by** (*auto dest: H.in-face-cycle-setD*)

moreover

have $c \in G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a'$

proof (*rule ccontr*)

assume *A*: $\neg ?thesis$

then have $G.\text{face-cycle-set } c \cap (G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a') =$

{}

using *G.face-cycle-set-parts* **by** (*auto simp: arcs-H*)

also then have $G.\text{face-cycle-set } c = H.\text{face-cycle-set } c$

using $\langle c \in \text{arcs } H \rangle$ **by** (*subst face-cycle-set-other-da*) (*auto simp: arcs-H*)

also have $\dots = H.\text{face-cycle-set } b$

using $\langle c \in ?L \rangle$ **using** *H.face-cycle-set-parts* **by** *auto*

finally show *False* **using** *assms* **by** *auto*

qed

ultimately show $c \in ?R$ **unfolding** *S-def arcs-H G.segment-face-cycle-x-x-eq*

by *auto*

next

fix *x* **assume** $x \in ?R$

from *assms* **have** $a \neq b$ **by** (*auto simp: arcs-H*)

from *assms* **have** *b-in*: $b \in \text{segment } G.\text{face-cycle-succ } a \ a$

using *G.segment-face-cycle-x-x-eq* **by** (*auto simp: arcs-H*)

have *fcs-a-neq-a*: $G.\text{face-cycle-succ } a \neq a$

using *assms* $\langle a \neq b \rangle$ **by** (*auto simp add: G.segment-face-cycle-x-x-eq G.fcs-x-eq-x*)

have *split-seg*: $\text{segment } G.\text{face-cycle-succ } a \ a = \text{segment } G.\text{face-cycle-succ } a \ b$
 $\cup \{b\}$

$\cup \text{segment } G.\text{face-cycle-succ } b \ a$

using *b-in* **by** (*intro segment-split*)

have *a-in-orb-a*: $a \in \text{orbit } G.\text{face-cycle-succ } a$ **by** (*simp add: G.face-cycle-set-def[symmetric]*)

```

define c where c = inv G.face-cycle-succ a
have c-succ: G.face-cycle-succ c = a unfolding c-def
  by (meson bij-inv-eq-iff permutation-bijective G.permutation-face-cycle-succ)
have c-in-aa: c ∈ segment G.face-cycle-succ a a
  unfolding G.segment-face-cycle-x-x-eq c-def using fcs-a-neq-a c-succ c-def
by force
have c-in: c ∈ {b} ∪ segment G.face-cycle-succ b a
  using split-seg b-in c-succ c-in-aa
  by (auto dest: not-in-segment1[OF segmentD-orbit] intro: segment.intros)
from c-in-aa have c ∈ arcs H unfolding G.segment-face-cycle-x-x-eq
  using arcs-in c-succ diff-face by (auto simp: arcs-H G.face-cycle-eq[of a])

have b-in-L: b ∈ ?L by auto
moreover
{ fix x assume x ∈ segment G.face-cycle-succ b a then have x ∈ ?L
  proof induct
    case base then show ?case
      using assms diff-face-not-in(2) by (subst H-fcs-eq-G-fcs[symmetric])
      (auto simp: arcs-H intro: H.face-cycle-succ-inI G.face-cycle-succ-inI)
    next
      case (step x)
      have G.face-cycle-succ x ∉ G.face-cycle-set a ⇒ b ∈ G.face-cycle-set a
⇒ False
      using step(1) by (metis G.face-cycle-eq G.face-cycle-succ-inI pre-digraph-map.face-cycle-set-def
segmentD-orbit)
      moreover
      have x ∈ arcs G
        using step assms H.in-face-cycle-setD arcs-H by auto
      moreover
      then have (G.face-cycle-succ x ∉ G.face-cycle-set a ⇒ b ∈ G.face-cycle-set
a ⇒ False) ⇒ {x, G.face-cycle-succ x} ∩ {a, a'} = {}
        using step(2,3) assms diff-face-not-in(2) H.in-face-cycle-setD arcs-H by
safe auto
      ultimately show ?case using step
        by (subst H-fcs-eq-G-fcs[symmetric]) (auto intro: H.face-cycle-succ-inI)
    qed
  } note sba-in-L = this
moreover
{ fix x assume A: x ∈ segment G.face-cycle-succ a' a' then have x ∈ ?L
  proof -
    from c-in have c ∈ ?L using b-in-L sba-in-L by blast

    have G.face-cycle-succ a' ≠ a'
      using A by (auto simp add: G.segment-face-cycle-x-x-eq G.fcs-x-eq-x)
    then have *: G.face-cycle-succ a' = H.face-cycle-succ c
      using a-neq-a' c-succ ⟨c ∈ arcs H⟩ unfolding G.face-cycle-succ-def
H.face-cycle-succ-def arcs-H
      by (auto simp: HM-def perm-restrict-simps perm-rem-conv G.bij-edge-succ

```

```

G.arev-eq-iff)

  from A have x ∈ H.face-cycle-set c
  proof induct
    case base then show ?case by (simp add: * H.face-cycle-succ-inI)
  next
    case (step x)
    have x ∈ arcs G
    using ⟨c ∈ arcs H⟩ step.hyps(2) by (auto simp: arcs-H dest: H.in-face-cycle-setD)
    moreover
    have G.face-cycle-succ x ≠ a' ⇒ {x, G.face-cycle-succ x} ∩ {a, a'} =
  {
    using step(1) diff-face-not-in(1) G.face-cycle-succ-inI G.segment-face-cycle-x-x-eq
    by (auto simp: not-in-segment2)
    ultimately
    show ?case using step by (subst H-fcs-eq-G-fcs[symmetric]) (auto intro:
H.face-cycle-succ-inI)
  }
  qed
  also have H.face-cycle-set c = ?L
  using ⟨c ∈ ?L⟩ H.face-cycle-set-parts by auto
  finally show ?thesis .
  qed
} note sa'a'-in-L = this
moreover
{ assume A: x ∈ segment G.face-cycle-succ a b

  obtain d where d ∈ ?L and d-succ: H.face-cycle-succ d = G.face-cycle-succ
a
  proof (cases G.face-cycle-succ a' = a')
    case True
    from c-in have c ∈ ?L using b-in-L sba-in-L by blast
    moreover
    have H.face-cycle-succ c = G.face-cycle-succ a
    using fcs-a-neq-a c-succ a-neq-a' True ⟨c ∈ arcs H⟩
    unfolding G.face-cycle-succ-def H.face-cycle-succ-def arcs-H
    by (auto simp: HM-def perm-restrict-simps arcs-H perm-rem-conv G.bij-edge-succ
G.arev-eq-iff)
    ultimately show ?thesis ..
  next
    case False

    define d where d = inv G.face-cycle-succ a'
    have d-succ: G.face-cycle-succ d = a' unfolding d-def
    by (meson bij-inv-eq-iff permutation-bijective G.permutation-face-cycle-succ)
    have *: d ∈ ?L
    using sa'a'-in-L False
    by (metis DiffI d-succ empty-iff G.face-cycle-set-self G.face-cycle-set-succ in-
sert-iff G.permutation-face-cycle-succ pre-digraph-map.face-cycle-set-def segment-x-x-eq)
    then have d ∈ arcs H using assms by (auto dest: H.in-face-cycle-setD)

```

```

    have  $H.\text{face-cycle-succ } d = G.\text{face-cycle-succ } a$ 
      using  $\text{fcs-a-neq-a } a\text{-neq-a}' \langle d \in \text{arcs } H \rangle d\text{-succ}$ 
      unfolding  $G.\text{face-cycle-succ-def } H.\text{face-cycle-succ-def } \text{arcs-H}$ 
    by (auto simp:  $\text{HM-def perm-restrict-simps } \text{arcs-H perm-rem-conv } G.\text{bij-edge-succ}$ 
 $G.\text{arev-eq-iff}$ )
      with * show ?thesis ..
  qed
  then have  $d \in \text{arcs } H$  using  $\text{assms}$ 
    by - (drule  $H.\text{in-face-cycle-setD}$ , auto)

  from  $A$  have  $x \in H.\text{face-cycle-set } d$ 
  proof induct
  case base then show ?case by (simp add:  $d\text{-succ[symmetric]} H.\text{face-cycle-succ-inI}$ )
  next
    case (step  $x$ )
    moreover
    have  $x \in \text{arcs } G$ 
      using  $\langle d \in \text{arcs } H \rangle \text{arcs-H digraph-map.in-face-cycle-setD } \text{step.hyps}(2)$  by
  fastforce
    moreover
    have  $\{x, G.\text{face-cycle-succ } x\} \cap \{a, a'\} = \{\}$ 
      proof -
        have  $a \neq x$  using  $\text{step}(2) H.\text{in-face-cycle-setD } \langle d \in \text{arcs } H \rangle \text{arcs-not-in}$ 
  by blast
          moreover
          have  $a \neq G.\text{face-cycle-succ } x$ 
            by (metis  $b\text{-in not-in-segment1 } \text{segment.step } \text{segmentD-orbit } \text{step}(1)$ )
          moreover
          have  $a' \neq x$   $a' \neq G.\text{face-cycle-succ } x$ 
            using  $\text{step}(1) \text{diff-face-not-in}(2)$  by (auto simp:  $G.\text{face-cycle-set-def}$ 
  dest!:  $\text{segmentD-orbit intro: orbit.step}$ )
          ultimately
          show ?thesis by auto
        qed
      ultimately
      show ?case using  $\text{step}$ 
        by (subst  $H.\text{fcs-eq-G-fcs[symmetric]}$ ) (auto intro:  $H.\text{face-cycle-succ-inI}$ )
    qed
    also have  $H.\text{face-cycle-set } d = ?L$ 
      using  $\langle d \in ?L \rangle H.\text{face-cycle-set-parts}$  by auto
    finally have  $x \in ?L$  .
  }
  ultimately show  $x \in ?L$ 
    using  $\langle x \in ?R \rangle \text{unfolding } S\text{-def split-seg}$  by blast
  qed

  lemma  $\text{HJ-fcs-eq-for-a}'$ :
    assumes  $b \in \text{arcs } H \cap G.\text{face-cycle-set } a'$ 
    shows  $H.\text{face-cycle-set } b = S$ 

```

```

proof –
  interpret rev: bidel-arc-diff-face G M a'
  using arcs-in diff-face by unfold-locales simp-all
  show ?thesis using rev.H-fcs-eq-for-a assms by (auto simp: rev-H rev-HM S-def
rev.S-def)
qed

lemma card-face-cycle-sets-da':
  card H.face-cycle-sets = card G.face-cycle-sets – card {G.face-cycle-set a,
G.face-cycle-set a'} + (if S = {} then 0 else 1)
proof –
  have S = G.face-cycle-set a ∪ G.face-cycle-set a' – {a, a'}
  unfolding S-def using diff-face-not-in
  by (auto simp: segment-x-x-eq G.permutation-face-cycle-succ G.face-cycle-set-def)
moreover
  { fix x assume x ∈ S
    then have x ∈ arcs H ∩ (G.face-cycle-set a ∪ G.face-cycle-set a' – {a, a'})
    unfolding ⟨S = ⟩ arcs-H using a-in by (auto intro: G.in-face-cycle-setD)
    then have H.face-cycle-set x = S using H-fcs-eq-for-a HJ-fcs-eq-for-a' by
blast
  }
  then have H.face-cycle-set ‘ S = (if S = {} then {} else {S})
  by auto
  ultimately show ?thesis by (simp add: card-face-cycle-sets-da0)
qed

end

locale bidel-arc-biconnected = bidel-arc +
  assumes reach-a: tail G a →*H head G a
begin

lemma reach-a': tail G a' →*H head G a'
  using reach-a a-in by (simp add: symmetric-reachable H.sym-arcs)

lemma
  tail-a': tail G a' = head G a and
  head-a': head G a' = tail G a
  using a-in by simp-all

lemma reachable-daI:
  assumes v →*G w shows v →*H w
proof –
  have *:  $\bigwedge v w. v \rightarrow_G w \implies v \rightarrow^*_H w$ 
  using reach-a reach-a' by (auto simp: arcs-ends-def ends-H arcs-G arc-to-ends-def
tail-a')
  show ?thesis using assms by induct (auto simp: verts-H intro: * H.reachable-trans)
qed

```

lemma *reachable-da*: $v \rightarrow^* H w \iff v \rightarrow^* G w$
by (*metis reachable-daD reachable-daI*)

lemma *sccs-verts-da*: $H.sccs-verts = G.sccs-verts$
by (*auto simp: G.in-sccs-verts-conv-reachable H.in-sccs-verts-conv-reachable reachable-da*)

lemma *card-sccs-da*: $\text{card } H.sccs = \text{card } G.sccs$
by (*simp add: G.card-sccs-verts[symmetric] H.card-sccs-verts[symmetric] sccs-verts-da*)

end

locale *bidel-arc-not-biconnected* = *bidel-arc* +
assumes *not-reach-a*: $\neg \text{tail } G a \rightarrow^* H \text{head } G a$
begin

lemma *H-awalkI*: $G.\text{awalk } u p v \implies \{a, a'\} \cap \text{set } p = \{\} \implies H.\text{awalk } u p v$
by (*auto simp: pre-digraph.apath-def pre-digraph.awalk-def verts-H arcs-H cas-da*)

lemma *tail-neq-head*: $\text{tail } G a \neq \text{head } G a$
using *not-reach-a a-in* **by** (*metis H.reachable-refl G.head-in-verts verts-H*)

lemma *scc-of-tail-neq-head*: $H.scc\text{-of } (\text{tail } G a) \neq H.scc\text{-of } (\text{head } G a)$

proof –

have $\text{tail } G a \in H.scc\text{-of } (\text{tail } G a)$ $\text{head } G a \in H.scc\text{-of } (\text{head } G a)$

using *ends-in* **by** (*auto simp: H.in-scc-of-self verts-H*)

with *not-reach-a* **show** *?thesis* **by** (*auto simp: H.scc-of-def*)

qed

lemma *scc-of-G-tail*:

assumes $u \in G.scc\text{-of } (\text{tail } G a)$

shows $H.scc\text{-of } u = H.scc\text{-of } (\text{tail } G a) \vee H.scc\text{-of } u = H.scc\text{-of } (\text{head } G a)$

proof –

from *assms* **have** $u \rightarrow^* G \text{tail } G a$ **by** (*auto simp: G.scc-of-def*)

then obtain p **where** $p: G.\text{apath } u p (\text{tail } G a)$ **by** (*auto simp: G.reachable-apath*)

show *?thesis*

proof (*cases head G a ∈ set (G.awalk-verts u p)*)

case *True*

with p **obtain** $p' q$ **where** $p = p' @ q$ $G.\text{awalk } (\text{head } G a) q (\text{tail } G a)$

and $p': G.\text{awalk } u p' (\text{head } G a)$

unfolding $G.\text{apath-def}$ **by** (*metis G.awalk-decomp*)

moreover

then have $\text{tail } G a \in \text{set } (\text{tl } (G.\text{awalk-verts } (\text{head } G a) q))$

using *tail-neq-head*

apply (*cases q*)

apply (*simp add: G.awalk-Nil-iff*)

```

    apply (simp add: G.awalk-Cons-iff)
    by (metis G.awalkE G.awalk-verts-non-Nil last-in-set)
  ultimately
  have tail G a  $\notin$  set (G.awalk-verts u p^)
    using G.apath-decomp-disjoint[OF p, of p' q tail G a] by auto
  with p' have {a,a'}  $\cap$  set p' = {}
  by (auto simp: G.set-awalk-verts G.apath-def) (metis a-in imageI G.head-arev)
  with p' show ?thesis unfolding G.apath-def by (metis H.scc-ofI-awalk
H.scc-of-eq H-awalkI)
next
case False
with p have {a,a'}  $\cap$  set p = {}
by (auto simp: G.set-awalk-verts G.apath-def) (metis a-in imageI G.tail-arev)
with p show ?thesis unfolding G.apath-def by (metis H.scc-ofI-awalk
H.scc-of-eq H-awalkI)
qed
qed

lemma scc-of-other:
  assumes u  $\notin$  G.scc-of (tail G a)
  shows H.scc-of u = G.scc-of u
  using assms
proof (intro set-eqI iffI)
  fix v assume v  $\in$  H.scc-of u then show v  $\in$  G.scc-of u
    by (auto simp: H.scc-of-def G.scc-of-def intro: reachable-daD)
next
  fix v assume v  $\in$  G.scc-of u
  then obtain p where p: G.awalk u p v by (auto simp: G.scc-of-def G.reachable-awalk)
  moreover
  have {a,a'}  $\cap$  set p = {}
  proof -
    have  $\neg u \rightarrow^* G$  tail G a using assms by (metis G.scc-ofI-reachable)
    then have  $\bigwedge p. \neg G$ .awalk u p (tail G a) by (metis G.reachable-awalk)
    then have tail G a  $\notin$  set (G.awalk-verts u p)
      using p by (auto dest: G.awalk-decomp)
    with p show ?thesis
  by (auto simp: G.set-awalk-verts G.apath-def) (metis a-in imageI G.head-arev)
  qed
  ultimately have H.awalk u p v by (rule H-awalkI)
  then show v  $\in$  H.scc-of u by (metis H.scc-ofI-reachable' H.reachable-awalk)
qed

lemma scc-of-tail-inter:
  tail G a  $\in$  G.scc-of (tail G a)  $\cap$  H.scc-of (tail G a)
  using ends-in by (auto simp: G.in-scc-of-self H.in-scc-of-self verts-H)

lemma scc-of-head-inter:
  head G a  $\in$  G.scc-of (tail G a)  $\cap$  H.scc-of (head G a)
proof -

```

have $\text{tail } G \ a \rightarrow_G \text{head } G \ a \ \text{head } G \ a \rightarrow_G \text{tail } G \ a$
by (*metis a-in G.in-arcs-imp-in-arcs-ends*) (*metis a-in G.graph-symmetric G.in-arcs-imp-in-arcs-ends*)
then have $\text{tail } G \ a \rightarrow^*_G \text{head } G \ a \ \text{head } G \ a \rightarrow^*_G \text{tail } G \ a$ **by** *auto*
then show *?thesis using ends-in by (auto simp: G.scc-of-def H.in-scc-of-self verts-H)*
qed

lemma *G-scc-of-tail-not-in: G.scc-of (tail G a) \notin H.sccs-verts*
proof
assume $A: G.\text{scc-of } (\text{tail } G \ a) \in H.\text{sccs-verts}$
from A *scc-of-tail-inter* **have** $G.\text{scc-of } (\text{tail } G \ a) = H.\text{scc-of } (\text{tail } G \ a)$
by (*metis H.scc-of-in-sccs-verts H.sccs-verts-disjoint a-in empty-iff G.tail-in-verts verts-H*)
moreover
from A *scc-of-head-inter* **have** $G.\text{scc-of } (\text{tail } G \ a) = H.\text{scc-of } (\text{head } G \ a)$
by (*metis H.scc-of-in-sccs-verts H.sccs-verts-disjoint a-in empty-iff G.head-in-verts verts-H*)
ultimately show *False using scc-of-tail-neq-head by blast*
qed

lemma *H-scc-of-a-not-in:*
 $H.\text{scc-of } (\text{tail } G \ a) \notin G.\text{sccs-verts} \ H.\text{scc-of } (\text{head } G \ a) \notin G.\text{sccs-verts}$
proof *safe*
assume $H.\text{scc-of } (\text{tail } G \ a) \in G.\text{sccs-verts}$
with *scc-of-tail-inter* **have** $G.\text{scc-of } (\text{tail } G \ a) = H.\text{scc-of } (\text{tail } G \ a)$
by (*metis G.scc-of-in-sccs-verts G.sccs-verts-disjoint a-in empty-iff G.tail-in-verts*)
with *G-scc-of-tail-not-in* **show** *False*
using *ends-in by (auto simp: H.scc-of-in-sccs-verts verts-H)*
next
assume $H.\text{scc-of } (\text{head } G \ a) \in G.\text{sccs-verts}$
with *scc-of-head-inter* **have** $G.\text{scc-of } (\text{tail } G \ a) = H.\text{scc-of } (\text{head } G \ a)$
by (*metis G.scc-of-in-sccs-verts G.sccs-verts-disjoint a-in empty-iff G.tail-in-verts*)
with *G-scc-of-tail-not-in* **show** *False*
using *ends-in by (auto simp: H.scc-of-in-sccs-verts verts-H)*
qed

lemma *scc-verts-da:*
 $H.\text{sccs-verts} = (G.\text{sccs-verts} - \{G.\text{scc-of } (\text{tail } G \ a)\}) \cup \{H.\text{scc-of } (\text{tail } G \ a), H.\text{scc-of } (\text{head } G \ a)\}$ (*is ?L = ?R*)
proof (*intro set-eqI iffI*)
fix S **assume** $S \in ?L$
then obtain u **where** $u \in \text{verts } G \ S = H.\text{scc-of } u$ **by** (*auto simp: verts-H H.sccs-verts-conv-scc-of*)
moreover
then have $G.\text{scc-of } (\text{tail } G \ a) \neq H.\text{scc-of } u$ **using** $\langle S \in ?L \rangle$ *G-scc-of-tail-not-in*
by *auto*
ultimately show $S \in ?R$
unfolding *G.sccs-verts-conv-scc-of*

```

    by (cases u ∈ G.scc-of (tail G a)) (auto dest: scc-of-G-tail scc-of-other)
  next
  fix S assume S ∈ ?R
  show S ∈ ?L
  proof (cases S ∈ G.sccs-verts)
    case True
      with ⟨S ∈ ?R⟩ obtain u where u: u ∈ verts G S = G.scc-of u and S ≠
        G.scc-of (tail G a)
        using H.scc-of-a-not-in by (auto simp: G.sccs-verts-conv-scc-of)
        then have G.scc-of u ∩ G.scc-of (tail G a) = {}
        using ends-in by (intro G.sccs-verts-disjoint) (auto simp: G.scc-of-in-sccs-verts)
        then have u ∉ G.scc-of (tail G a)
        using u by (auto dest: G.in-scc-of-self)
        with u show ?thesis using scc-of-other
        by (auto simp: H.sccs-verts-conv-scc-of verts-H G.sccs-verts-conv-scc-of)
    next
  case False with ⟨S ∈ ?R⟩ ends-in show ?thesis by (auto simp: H.sccs-verts-conv-scc-of
verts-H)
  qed
qed

```

```

lemma card-sccs-da: card H.sccs = Suc (card G.sccs)
  using H.scc-of-a-not-in ends-in
  unfolding G.card-sccs-verts[symmetric] H.card-sccs-verts[symmetric] scc-verts-da
  by (simp add: card-insert-if G.finite-sccs-verts scc-of-tail-neq-head card-Suc-Diff1
    G.scc-of-in-sccs-verts del: card-Diff-insert)

```

end

sublocale bidel-arc-not-biconnected \subseteq bidel-arc-same-face

proof

note a-in

moreover from a-in have head G a ∈ tail G ‘ G.face-cycle-set a

by (simp add: G.heads-face-cycle-set[symmetric])

moreover have tail G a ∈ tail G ‘ G.face-cycle-set a by simp

ultimately obtain p where p: G.trail (head G a) p (tail G a) set p \subseteq G.face-cycle-set a

by (rule G.obtain-trail-in-fcs')

define p' where p' = G.awalk-to-apath p

from p have p': G.apath (head G a) p' (tail G a) set p' \subseteq G.face-cycle-set a

by (auto simp: G.trail-def p'-def dest: G.apath-awalk-to-apath G.awalk-to-apath-subset)

then have set p' \subseteq arcs G

using a-in by (blast dest: G.in-face-cycle-setD)

have \neg set p' \subseteq arcs H

proof

assume set p' \subseteq arcs H

then have H.awalk (head G a) p' (tail G a)

using p' by (auto simp: G.apath-def arcs-H intro: H-awalkI)

```

then show False using not-reach-a by (metis H.symmetric-reachable' H.reachable-awalk)
qed
then have  $set\ p' \cap \{a, a'\} \neq \{\}$  using  $\langle set\ p' \subseteq arcs\ G \rangle$  by (auto simp: arcs-H)
moreover
have  $a \notin set\ p'$ 
proof
  assume  $a \in set\ p'$ 
  then have  $head\ G\ a \in set\ (tl\ (G.awalk-verts\ (head\ G\ a)\ p'))$ 
    using  $\langle G.apath - p' - \rangle$ 
    by (cases p') (auto simp: G.set-awalk-verts G.apath-def G.awalk-Cons-iff,
metis imageI)
  moreover
  have  $head\ G\ a \notin set\ (tl\ (G.awalk-verts\ (head\ G\ a)\ p'))$ 
    using  $\langle G.apath - p' - \rangle$  by (cases p') (auto simp: G.apath-def)
  ultimately show False by contradiction
qed
ultimately
have  $a' \in G.face-cycle-set\ a$  using  $p'(2)$  by auto
then show  $G.face-cycle-set\ a' = G.face-cycle-set\ a$  using  $G.face-cycle-set-parts$ 
by auto
qed

```

```

locale bidel-arc-tail-conn = bidel-arc +
  assumes conn-tail:  $tail\ G\ a \notin H.isolated-verts$ 

```

```

locale bidel-arc-head-conn = bidel-arc +
  assumes conn-head:  $head\ G\ a \notin H.isolated-verts$ 

```

```

locale bidel-arc-tail-isolated = bidel-arc +
  assumes isolated-tail:  $tail\ G\ a \in H.isolated-verts$ 

```

```

locale bidel-arc-head-isolated = bidel-arc +
  assumes isolated-head:  $head\ G\ a \in H.isolated-verts$ 
begin

```

```

  lemma G-edge-succ-a'-no-loop:
    assumes no-loop-a:  $head\ G\ a \neq tail\ G\ a$  shows  $G-edge-succ-a'$ :  $edge-succ\ M$ 
 $a' = a'$  (is ?t2)
  proof -
    have  $*$ :  $out-arcs\ G\ (tail\ G\ a') = \{a'\}$ 
      using a-in isolated-head no-loop-a
      by (auto simp: H.isolated-verts-def verts-H out-arcs-def arcs-H tail-H)
    obtain  $edge-succ\ M\ a' \in \{a'\}$ 
      using  $G.edge-succ-cyclic[of\ tail\ G\ a']$ 
      apply (rule eq-on-cyclic-on-iff1 [where x=a'])
      using  $*$  a-in a-neq-a' no-loop-a by simp-all
    then show ?thesis by auto
qed

```

lemma *G-face-cycle-succ-a-no-loop*:
assumes *no-loop-a*: $\text{head } G \ a \neq \text{tail } G \ a$ **shows** $G.\text{face-cycle-succ } a = a'$
using *assms* **by** (*auto simp: G.face-cycle-succ-def G-edge-succ-a'-no-loop*)

end

locale *bidel-arc-same-face-tail-conn* = *bidel-arc-same-face* + *bidel-arc-tail-conn*
begin

definition *a-neighbor* :: 'b **where**
 $a\text{-neighbor} \equiv \text{SOME } b. G.\text{face-cycle-succ } b = a$

lemma *face-cycle-succ-a-neighbor*: $G.\text{face-cycle-succ } a\text{-neighbor} = a$
proof –
have $\exists b. G.\text{face-cycle-succ } b = a$ **by** (*metis G.face-cycle-succ-pred*)
then show *?thesis* **unfolding** *a-neighbor-def* **by** (*rule someI-ex*)
qed

lemma *a-neighbor-in*: $a\text{-neighbor} \in \text{arcs } G$
using *a-in* **by** (*metis face-cycle-succ-a-neighbor G.face-cycle-succ-closed*)

lemma *a-neighbor-neq-a*: $a\text{-neighbor} \neq a$
proof
assume $a\text{-neighbor} = a$
then have $G.\text{face-cycle-set } a = \{a\}$ **using** *face-cycle-succ-a-neighbor* **by** (*simp*
add: G.fcs-x-eq-x)
with *a-neq-a'* *same-face G.face-cycle-set-self[of a]* **show** *False* **by** *simp*
qed

lemma *a-neighbor-neq-a'*: $a\text{-neighbor} \neq a'$
proof
assume *A*: $a\text{-neighbor} = a'$

have *a-in-oa*: $a \in \text{out-arcs } G \ (\text{tail } G \ a)$ **using** *a-in* **by** *auto*
have *cyc*: *cyclic-on (edge-succ M) (out-arcs G (tail G a))*
using *a-in* **by** (*intro G.edge-succ-cyclic auto*)

from *A* **have** $G.\text{face-cycle-succ } a' = a$ **by** (*metis face-cycle-succ-a-neighbor*)
then have $\text{edge-succ } M \ a = a$ **by** (*auto simp: G.face-cycle-succ-def*)
then have $\text{card } (\text{out-arcs } G \ (\text{tail } G \ a)) = 1$
using *cyc a-in* **by** (*auto elim: eq-on-cyclic-on-iff1*)
then have $\text{out-arcs } G \ (\text{tail } G \ a) = \{a\}$
using *a-in-oa* **by** (*auto simp del: in-out-arcs-conv dest: card-eq-SucD*)
then show *False* **using** *conn-tail a-in*
by (*auto simp: H.isolated-verts-def arcs-H tail-H verts-H out-arcs-def*)

qed

lemma *edge-rev-a-neighbor-neq*: $\text{edge-rev } M \ a \ \text{neighbor} \neq a'$
by (*metis a-neighbor-neq-a G.arev-arev*)

lemma *edge-succ-a-neighbor-neq*: $\text{edge-succ } M \ a \neq a'$

proof

assume *edge-succ* $M \ a = a'$

then have $G.\text{face-cycle-set } a' = \{a'\}$

using *face-cycle-succ-a-neighbor*

by *auto* (*metis G.arev-arev-raw G.face-cycle-succ-def G.fcs-x-eq-x a-in comp-apply singletonD*)

with *a-neighbor-neq-a' same-face G.face-cycle-set-self*[of *a*] **show** *False* by *auto*

qed

lemma *H-face-cycle-succ-a-neighbor*: $H.\text{face-cycle-succ } a\text{-neighbor} = G.\text{face-cycle-succ } a'$

using *face-cycle-succ-a-neighbor edge-succ-a-neighbor edge-rev-a-neighbor-neq a-neighbor-neq-a a-neighbor-neq-a' a-neighbor-in*

unfolding *H.face-cycle-succ-def G.face-cycle-succ-def*

by (*auto simp: HM-def perm-restrict-simps perm-rem-conv G.bij-edge-succ*)

lemma *H-fcs-a-neighbor*: $H.\text{face-cycle-set } a\text{-neighbor} = \text{segment } G.\text{face-cycle-succ } a' \ a$
(*is ?L = ?R*)

proof –

{ **fix** *n* **assume** $A: 0 < n \ n < \text{funpow-dist1 } G.\text{face-cycle-succ } a' \ a$

then have $*$: $(G.\text{face-cycle-succ } \sim n) \ a' \in \text{segment } G.\text{face-cycle-succ } a' \ a$

using *a-in-o* by (*auto simp: segment-altdef*)

then have $(G.\text{face-cycle-succ } \sim n) \ a' \notin \{a, a'\}$ $(G.\text{face-cycle-succ } \sim n) \ a' \in \text{arcs } G$

using *not-in-segment1*[*OF a-in-o*] *not-in-segment2*[of *a G.face-cycle-succ a'*]

by (*auto simp: segment-altdef a-in-o*)

} **note** $X = \text{this}$

{ **fix** *n* **assume** $0 < n \ n < \text{funpow-dist1 } G.\text{face-cycle-succ } a' \ a$

then have $(H.\text{face-cycle-succ } \sim n) \ a\text{-neighbor} = (G.\text{face-cycle-succ } \sim n) \ a'$

proof (*induct n*)

case 0 **then show** *?case* by *simp*

next

case (*Suc n*)

show *?case*

proof (*cases n=0*)

case *True* **then show** *?thesis* by (*simp add: H-face-cycle-succ-a-neighbor*)

next

case *False*

then have $(H.\text{face-cycle-succ } \sim n) \ a\text{-neighbor} = (G.\text{face-cycle-succ } \sim n) \ a'$

using *Suc* by *simp*

then show *?thesis*

using $X[\text{of } \text{Suc } n] \ X[\text{of } n] \ \text{False } \text{Suc}$ by (*simp add: H-fcs-eq-G-fcs*)

```

    qed
  qed
} note Y = this

have fcs-a'-neq-a: G.face-cycle-succ a' ≠ a
by (metis (no-types) a-neighbor-neq-a' G.face-cycle-pred-succ face-cycle-succ-a-neighbor)

show ?thesis
proof (intro set-eqI iffI)
  fix b assume b ∈ ?L

  define m where m = funpow-dist1 G.face-cycle-succ a' a - 1

  have b-in0: b ∈ orbit H.face-cycle-succ (a-neighbor)
    using ⟨b ∈ ?L⟩ by (simp add: H.face-cycle-set-def[symmetric])

  have 0 < m
  by (auto simp: m-def) (metis a-neighbor-neq-a' G.face-cycle-pred-succ G.face-cycle-set-def
    G.face-cycle-set-self G.face-cycle-set-succ face-cycle-succ-a-neighbor fun-
pow-dist-0-eq neq0-conv
  same-face)
  then have pos-dist: 0 < funpow-dist1 H.face-cycle-succ a-neighbor b
    by (simp add: m-def)

  have *: (G.face-cycle-succ  $\widehat{\widehat{}}$  Suc m) a' = a
    using a-in-o by (simp add: m-def funpow-dist1-prop del: funpow.simps)
  have (H.face-cycle-succ  $\widehat{\widehat{}}$  m) a-neighbor = a-neighbor
  proof -
    have a = G.face-cycle-succ ((H.face-cycle-succ  $\widehat{\widehat{}}$  m) a-neighbor)
      using * ⟨0 < m⟩ by (simp add: Y m-def)
  then show ?thesis using face-cycle-succ-a-neighbor by (metis G.face-cycle-pred-succ)
  qed
  then have funpow-dist1 H.face-cycle-succ a-neighbor b ≤ m
    using ⟨0 < m⟩ b-in0 by (intro funpow-dist1-le-self) simp-all
  also have ... < funpow-dist1 G.face-cycle-succ a' a unfolding m-def by
simp
  finally have dist-less: funpow-dist1 H.face-cycle-succ a-neighbor b
    < funpow-dist1 G.face-cycle-succ a' a .
  have b = (H.face-cycle-succ  $\widehat{\widehat{}}$  funpow-dist1 H.face-cycle-succ a-neighbor b)
a-neighbor
    using b-in0 by (simp add: funpow-dist1-prop del: funpow.simps)
  also have ... = (G.face-cycle-succ  $\widehat{\widehat{}}$  funpow-dist1 H.face-cycle-succ a-neighbor
b) a'
    using pos-dist dist-less by (rule Y)
  also have ... ∈ ?R using pos-dist dist-less by (simp add: segment-altdef
a-in-o del: funpow.simps)
  finally show b ∈ ?R .
next
fix b assume b ∈ ?R

```

```

    then show  $b \in ?L$ 
      using  $Y$ 
      by (auto simp: segment-altdef a-in-o  $H$ .face-cycle-set-altdef  $Suc$ -le-eq) metis
    qed
  qed
end

```

```

locale bidel-arc-isolated-loop =
  bidel-arc-biconnected + bidel-arc-tail-isolated
begin

```

```

  lemma loop-a[simp]:  $head\ G\ a = tail\ G\ a$ 
    using isolated-tail reach-a by (auto simp:  $H$ .isolated-verts-def
      elim:  $H$ .converse-reachable-cases dest: out-arcs-emptyD-dominates)

```

```

end

```

```

sublocale bidel-arc-isolated-loop  $\subseteq$  bidel-arc-head-isolated
  using isolated-tail loop-a by unfold-locales simp

```

```

context bidel-arc-isolated-loop begin

```

The edges a and a' form a loop on an otherwise isolated vertex

```

lemma card-isolated-verts-da:  $card\ H.isolated-verts = Suc\ (card\ G.isolated-verts)$ 
  by (simp add: card-isolated-verts-da0 isolated-tail)

```

```

lemma
  G-edge-succ-a[simp]:  $edge-succ\ M\ a = a'$  (is  $?t1$ ) and
  G-edge-succ-a'[simp]:  $edge-succ\ M\ a' = a$  (is  $?t2$ )

```

```

proof –

```

```

  have *:  $out-arcs\ G\ (tail\ G\ a) = \{a, a'\}$ 
    using a-in isolated-tail
    by (auto simp:  $H.isolated-verts-def\ verts-H\ out-arcs-def\ arcs-H\ tail-H$ )

```

```

  obtain  $edge-succ\ M\ a' \in \{a, a'\}$   $edge-succ\ M\ a' \neq a'$ 
    using G.edge-succ-cyclic[of tail G a']
    apply (rule eq-on-cyclic-on-iff1 [where  $x=a'$ ])
    using * a-in a-neq-a' loop-a by auto

```

```

  moreover

```

```

  obtain  $edge-succ\ M\ a \in \{a, a'\}$   $edge-succ\ M\ a \neq a$ 
    using G.edge-succ-cyclic[of tail G a]
    apply (rule eq-on-cyclic-on-iff1 [where  $x=a$ ])
    using * a-in a-neq-a' loop-a by auto

```

```

  ultimately show  $?t1\ ?t2$  by auto

```

```

qed

```

lemma
G.face-cycle-succ-*a*[simp]: *G*.face-cycle-succ *a* = *a* **and**
G.face-cycle-succ-*a'*[simp]: *G*.face-cycle-succ *a'* = *a'*
by (auto simp: *G*.face-cycle-succ-def)

lemma
G.face-cycle-set-*a*[simp]: *G*.face-cycle-set *a* = {*a*} **and**
G.face-cycle-set-*a'*[simp]: *G*.face-cycle-set *a'* = {*a'*}
unfolding *G*.fcs-x-eq-x[symmetric] **by** simp-all

end

sublocale *bidel-arc-isolated-loop* \subseteq *bidel-arc-diff-face*
using *a-neq-a'* **by** unfold-locales auto

context *bidel-arc-isolated-loop* **begin**

lemma *card-face-cycle-sets-da*: *card G*.face-cycle-sets = *Suc (Suc (card H*.face-cycle-sets))
unfolding *card-face-cycle-sets-da'* **using** *diff-face card-fcs-aa'-le*
by (auto simp: *card-insert-if S-def G.segment-face-cycle-x-x-eq*)

lemma *euler-genus-da*: *H.euler-genus* = *G.euler-genus*
unfolding *G.euler-genus-def H.euler-genus-def G.euler-char-def H.euler-char-def*
by (simp add: *card-isolated-verts-da verts-H card-arcs-da card-face-cycle-sets-da card-sccs-da*)

end

locale *bidel-arc-two-isolated* =
bidel-arc-not-biconnected + *bidel-arc-tail-isolated* + *bidel-arc-head-isolated*
begin

tail G a and *head G a* form an SCC with *a* and *a'* as the only arcs.

lemma *no-loop-a*: *head G a* \neq *tail G a*
using *not-reach-a a-in* **by** (auto simp: *verts-H*)

lemma *card-isolated-verts-da*: *card H.isolated-verts* = *Suc (Suc (card G.isolated-verts))*
using *no-loop-a isolated-tail isolated-head* **by** (simp add: *card-isolated-verts-da0 card-insert-if*)

lemma *G-edge-succ-a'*[simp]: *edge-succ M a'* = *a'*
using *G-edge-succ-a'-no-loop no-loop-a* **by** simp

lemma *G-edge-succ-a*[simp]: *edge-succ M a* = *a*
proof –
have *: *out-arcs G (tail G a)* = {*a*}
using *a-in isolated-tail isolated-head no-loop-a*
by (auto simp: *H.isolated-verts-def verts-H out-arcs-def arcs-H tail-H*)

```

obtain edge-succ  $M a \in \{a\}$ 
  using G.edge-succ-cyclic[of tail G a]
  apply (rule eq-on-cyclic-on-iff1 [where  $x=a$ ])
  using * a-in a-neq-a' no-loop-a by simp-all
  then show ?thesis by auto
qed

lemma
  G.face-cycle-succ-a[simp]: G.face-cycle-succ  $a = a'$  and
  G.face-cycle-succ-a'[simp]: G.face-cycle-succ  $a' = a$ 
  by (auto simp: G.face-cycle-succ-def)

lemma
  G.face-cycle-set-a[simp]: G.face-cycle-set  $a = \{a, a'\}$  (is ?t1) and
  G.face-cycle-set-a'[simp]: G.face-cycle-set  $a' = \{a, a'\}$  (is ?t2)
proof -
  { fix  $n$  have (G.face-cycle-succ  $\overset{\sim}{\sim} n$ )  $a \in \{a, a'\}$  (G.face-cycle-succ  $\overset{\sim}{\sim} n$ )  $a' \in$ 
   $\{a, a'\}$ 
  by (induct n auto)
  }
  then
  show ?t1 ?t2 by (auto simp: G.face-cycle-set-altdef intro: exI [where  $x=0$ ]
exI [where  $x=1$ ])
qed

lemma card-face-cycle-sets-da: card G.face-cycle-sets = Suc (card H.face-cycle-sets)
  unfolding card-face-cycle-sets-da0 using card-fcs-aa'-le by simp

lemma euler-genus-da: H.euler-genus = G.euler-genus
  unfolding G.euler-genus-def H.euler-genus-def G.euler-char-def H.euler-char-def
  by (simp add: card-isolated-verts-da verts-H card-arcs-da card-face-cycle-sets-da
card-sccs-da)

end

locale bidel-arc-tail-not-isol = bidel-arc-not-biconnected +
  bidel-arc-tail-conn

sublocale bidel-arc-tail-not-isol  $\subseteq$  bidel-arc-same-face-tail-conn
  by unfold-locales

locale bidel-arc-only-tail-not-isol = bidel-arc-tail-not-isol +
  bidel-arc-head-isolated

context bidel-arc-only-tail-not-isol
begin

lemma card-isolated-verts-da: card H.isolated-verts = Suc (card G.isolated-verts)
  using isolated-head conn-tail by (simp add: card-isolated-verts-da0)

```

lemma *segment-a'-a-ne*: *segment G.face-cycle-succ a' a ≠ {}*
unfolding *H-fcs-a-neigh[symmetric]* **by** *auto*

lemma *segment-a-a'-e*: *segment G.face-cycle-succ a a' = {}*
proof –
have *a' = G.face-cycle-succ a* **using** *tail-neq-head*
by (*simp add: G-face-cycle-succ-a-no-loop*)
then show *?thesis* **by** (*auto simp: segment1-empty*)
qed

lemma *card-face-cycle-sets-da*: *card H.face-cycle-sets = card G.face-cycle-sets*
unfolding *card-face-cycle-sets-da'* **using** *segment-a'-a-ne segment-a-a'-e card-fcs-gt-0*
by (*simp add: card-insert-if*)

lemma *euler-genus-da*: *H.euler-genus = G.euler-genus*
unfolding *G.euler-genus-def H.euler-genus-def G.euler-char-def H.euler-char-def*
by (*simp add: card-isolated-verts-da verts-H card-arcs-da card-face-cycle-sets-da card-sccs-da*)

end

locale *bidel-arc-only-head-not-isol = bidel-arc-not-biconnected +*
bidel-arc-head-conn +
bidel-arc-tail-isolated
begin

interpretation *rev: bidel-arc G M a'*
using *a-in* **by** *unfold-locales simp*

interpretation *rev: bidel-arc-only-tail-not-isol G M a'*
using *a-in not-reach-a*
by *unfold-locales (auto simp: rev-H isolated-tail conn-head dest: H.symmetric-reachable')*

lemma *euler-genus-da*: *H.euler-genus = G.euler-genus*
using *rev.euler-genus-da* **by** (*simp add: rev-H rev-HM*)

end

locale *bidel-arc-two-not-isol = bidel-arc-tail-not-isol +*
bidel-arc-head-conn
begin

lemma *isolated-verts-da*: *H.isolated-verts = G.isolated-verts*
using *conn-head conn-tail* **by** (*subst isolated-da-pre*) *simp*

lemma *segment-a'-a-ne'*: *segment G.face-cycle-succ a' a ≠ {}*
unfolding *H-fcs-a-neigh[symmetric]* **by** *auto*

interpretation *rev: bidel-arc-tail-not-isol G M a'*
using *arcs-in not-reach-a rev-H conn-head*
by *unfold-locales (auto dest: H.symmetric-reachable')*

lemma *segment-a-a'-ne': segment G.face-cycle-succ a a' ≠ {}*
using *rev.H-fcs-a-neigh[symmetric] rev-H rev-HM* **by** *auto*

lemma *card-face-cycle-sets-da: card H.face-cycle-sets = Suc (card G.face-cycle-sets)*
unfolding *card-face-cycle-sets-da'* **using** *segment-a'-a-ne' segment-a-a'-ne'*
card-fcs-gt-0
by *(simp add: segments-neq card-insert-if)*

lemma *euler-genus-da: H.euler-genus = G.euler-genus*
unfolding *G.euler-genus-def H.euler-genus-def G.euler-char-def H.euler-char-def*
by *(simp add: isolated-verts-da verts-H card-arcs-da card-face-cycle-sets-da*
card-sccs-da)

end

locale *bidel-arc-biconnected-non-triv = bidel-arc-biconnected +*
bidel-arc-tail-conn

sublocale *bidel-arc-biconnected-non-triv ⊆ bidel-arc-head-conn*
by *unfold-locales (metis (mono-tags) G.in-sccs-verts-conv-reachable G.symmetric-reachable'*
H.isolated-verts-in-sccs conn-tail empty-iff insert-iff reach-a reachable-daD sccs-verts-da)

context *bidel-arc-biconnected-non-triv* **begin**

lemma *isolated-verts-da: H.isolated-verts = G.isolated-verts*
using *conn-head conn-tail* **by** *(subst isolated-da-pre) simp*

end

locale *bidel-arc-biconnected-same = bidel-arc-biconnected-non-triv +*
bidel-arc-same-face

sublocale *bidel-arc-biconnected-same ⊆ bidel-arc-same-face-tail-conn*
by *unfold-locales*

context *bidel-arc-biconnected-same* **begin**

interpretation *rev: bidel-arc-same-face-tail-conn G M a'*
using *arcs-in conn-head* **by** *unfold-locales (auto simp: same-face rev-H)*

lemma *card-face-cycle-sets-da: Suc (card H.face-cycle-sets) ≥ (card G.face-cycle-sets)*
unfolding *card-face-cycle-sets-da'* **using** *card-fcs-gt-0* **by** *linarith*

lemma *euler-genus-da: H.euler-genus ≤ G.euler-genus*

```

using card-face-cycle-sets-da
unfolding G.euler-genus-def H.euler-genus-def G.euler-char-def H.euler-char-def
by (simp add: isolated-verts-da verts-H card-arcs-da card-sccs-da )

end

locale bidel-arc-biconnected-diff = bidel-arc-biconnected-non-triv +
  bidel-arc-diff-face
begin

lemma fcs-not-triv: G.face-cycle-set a ≠ {a} ∨ G.face-cycle-set a' ≠ {a'}
proof (rule ccontr)
  assume ¬?thesis
  then have G.face-cycle-succ a = a G.face-cycle-succ a' = a'
    by (auto simp: G.fcs-x-eq-x)
  then have *: edge-succ M a = a' edge-succ M a' = a
    by (auto simp: G.face-cycle-succ-def)
  then have (edge-succ M  $\hat{\hat{}}$  2) a = a by (auto simp: eval-nat-numeral)
  { fix n
    have (edge-succ M  $\hat{\hat{}}$  2) a = a by (auto simp: * eval-nat-numeral)
    then have (edge-succ M  $\hat{\hat{}}$  n) a = (edge-succ M  $\hat{\hat{}}$  (n mod 2)) a
      by (auto simp: funpow-mod-eq)
    moreover have n mod 2 = 0 ∨ n mod 2 = 1 by auto
    ultimately have (edge-succ M  $\hat{\hat{}}$  n) a ∈ {a, a'} by (auto simp: *)
  }
  then have orbit (edge-succ M) a = {a, a'}
  by (auto simp: orbit-altdef-permutation[OF G.permutation-edge-succ] exI[where
x=0] exI[where x=1] *)

  have out-arcs G (tail G a) ⊆ {a, a'}
  proof -
    have cyclic-on (edge-succ M) (out-arcs G (tail G a))
      using arcs-in by (intro G.edge-succ-cyclic) auto
    then have orbit (edge-succ M) a = out-arcs G (tail G a)
      using arcs-in by (intro orbit-cyclic-eq3) auto
    then show ?thesis using ⟨orbit - - = {-, -}⟩ by auto
  qed
  then have out-arcs H (tail G a) = {} by (auto simp: arcs-H tail-H)
  then have tail G a ∈ H.isolated-verts using arcs-in by (simp add: H.isolated-verts-def
verts-H)
  then show False using conn-tail by contradiction
  qed

lemma S-ne: S ≠ {}
  using fcs-not-triv by (auto simp: S-def G.segment-face-cycle-x-x-eq)

lemma card-face-cycle-sets-da: card G.face-cycle-sets = Suc (card H.face-cycle-sets)

```

unfolding *card-face-cycle-sets-da'* **using** *S-ne diff-face card-fcs-aa'-le* **by** *simp*

lemma *euler-genus-da*: $H.euler-genus = G.euler-genus$

unfolding *G.euler-genus-def H.euler-genus-def G.euler-char-def H.euler-char-def*

by (*simp add: isolated-verts-da verts-H card-arcs-da card-sccs-da card-face-cycle-sets-da*)

end

context *bidel-arc* **begin**

lemma *euler-genus-da*: $H.euler-genus \leq G.euler-genus$

proof –

let *?biconnected* = *tail G a* \rightarrow^*_H *head G a*

let *?isol-tail* = *tail G a* $\in H.isolated-verts$

let *?isol-head* = *head G a* $\in H.isolated-verts$

let *?same-face* = *G.face-cycle-set a'* = *G.face-cycle-set a*

{ **assume** *?biconnected ?isol-tail*

then interpret *EG: bidel-arc-isolated-loop* **by** *unfold-locales*

have *?thesis* **by** (*simp add: EG.euler-genus-da*)

}

moreover

{ **assume** *?biconnected* $\neg ?isol-tail$ *?same-face*

then interpret *EG: bidel-arc-biconnected-same* **by** *unfold-locales*

have *?thesis* **by** (*simp add: EG.euler-genus-da*)

}

moreover

{ **assume** *?biconnected* $\neg ?isol-tail$ $\neg ?same-face$

then interpret *EG: bidel-arc-biconnected-diff* **by** *unfold-locales*

have *?thesis* **by** (*simp add: EG.euler-genus-da*)

}

moreover

{ **assume** $\neg ?biconnected$ *?isol-tail ?isol-head*

then interpret *EG: bidel-arc-two-isolated* **by** *unfold-locales*

have *?thesis* **by** (*simp add: EG.euler-genus-da*)

}

moreover

{ **assume** $\neg ?biconnected$ $\neg ?isol-tail$ *?isol-head*

then interpret *EG: bidel-arc-only-tail-not-isol* **by** *unfold-locales*

have *?thesis* **by** (*simp add: EG.euler-genus-da*)

}

moreover

{ **assume** $\neg ?biconnected$ *?isol-tail* $\neg ?isol-head$

then interpret *EG: bidel-arc-only-head-not-isol* **by** *unfold-locales*

have *?thesis* **by** (*simp add: EG.euler-genus-da*)

}

moreover

{ **assume** $\neg ?biconnected$ $\neg ?isol-tail$ $\neg ?isol-head$

```

    then interpret EG: bidel-arc-two-not-isol by unfold-locales
    have ?thesis by (simp add: EG.euler-genus-da)
  }
  ultimately show ?thesis by satx
qed
end

```

14.3 Modifying *edge-rev*

definition (in *pre-digraph-map*) *rev-swap* :: 'b \Rightarrow 'b \Rightarrow 'b *pre-map* **where**
rev-swap a b = (\lrcorner *edge-rev* = *perm-swap* a b (*edge-rev* M), *edge-succ* = *perm-swap* a b (*edge-succ* M) \lrcorner)

context *digraph-map* **begin**

lemma *digraph-map-rev-swap*:

assumes *arc-to-ends* G a = *arc-to-ends* G b {a,b} \subseteq *arcs* G

shows *digraph-map* G (*rev-swap* a b)

proof

let ?M' = *rev-swap* a b

have *tail-swap*: $\bigwedge x. \text{tail } G ((a \rightleftharpoons_F b) x) = \text{tail } G x$

using *assms* **by** (*case-tac* $x \in \{a,b\}$) (*auto simp: arc-to-ends-def*)

have *swap-in-arcs*: $\bigwedge x. (a \rightleftharpoons_F b) x \in \text{arcs } G \longleftrightarrow x \in \text{arcs } G$

using *assms* **by** (*case-tac* $x \in \{a,b\}$) *auto*

have *es-perm*: *edge-succ* ?M' *permutes arcs* G

using *assms* *edge-succ-permutes* **unfolding** *permutes-conv-has-dom*
by (*auto simp: rev-swap-def has-dom-perm-swap*)

{

fix x **show** $(x \in \text{arcs } G) = (\text{edge-rev } (\text{rev-swap } a b) x \neq x)$

using *assms*(2)

by (*cases* $x \in \{a,b\}$) (*auto simp: rev-swap-def perm-swap-def arev-dom*

Transposition.transpose-def split: if-splits)

next

fix x **assume** $x \in \text{arcs } G$ **then show** *edge-rev* ?M' (*edge-rev* ?M' x) = x

by (*auto simp: rev-swap-def perm-swap-comp[symmetric]*)

next

fix x **assume** $x \in \text{arcs } G$ **then show** *tail* G (*edge-rev* ?M' x) = *head* G x

using *assms* **by** (*case-tac* $x \in \{a,b\}$) (*auto simp: rev-swap-def perm-swap-def*

tail-swap arc-to-ends-def)

next

show *edge-succ* ?M' *permutes arcs* G **by fact**

next

fix v **assume** A: $v \in \text{verts } G$ *out-arcs* G v $\neq \{\}$

then obtain c **where** $c \in \text{out-arcs } G v$ **by** *blast*

have *inj* (*perm-swap* a b (*edge-succ* M)) **by** (*simp add: bij-is-inj bij-edge-succ*)

```

have out-arcs G v = (a  $\Rightarrow_F$  b) ‘ out-arcs G v
  by (auto simp: tail-swap swap-swap-id swap-in-arcs intro: image-eqI[where
x=(a  $\Rightarrow_F$  b) y for y])
  also have (a  $\Rightarrow_F$  b) ‘ out-arcs G v = (a  $\Rightarrow_F$  b) ‘ orbit (edge-succ M) ((a
 $\Rightarrow_F$  b) c)
    using edge-succ-cyclic using A <c  $\in$  ->
    by (intro arg-cong[where f=(‘) (a  $\Rightarrow_F$  b)])
      (intro orbit-cyclic-eq3[symmetric], auto simp: swap-in-arcs tail-swap)
  also have ... = orbit (edge-succ ?M') c
    by (simp add: orbit-perm-swap rev-swap-def)
  finally have oa-orb: out-arcs G v = orbit (edge-succ ?M') c .

show cyclic-on (edge-succ ?M') (out-arcs G v)
  unfolding oa-orb using es-perm finite-arcs by (rule cyclic-on-orbit)
}
qed

```

lemma euler-genus-rev-swap:

```

assumes arc-to-ends G a = arc-to-ends G b {a,b}  $\subseteq$  arcs G
shows pre-digraph-map.euler-genus G (rev-swap a b) = euler-genus

```

proof –

```

let ?M' = rev-swap a b

```

```

interpret G': digraph-map G ?M' using assms by (rule digraph-map-rev-swap)

```

```

have swap-in-arcs:  $\bigwedge x. (a \Rightarrow_F b) x \in \text{arcs } G \longleftrightarrow x \in \text{arcs } G$ 
  using assms by (case-tac x  $\in$  {a,b}) auto

```

```

have G'-fcs: G'.face-cycle-succ = perm-swap a b face-cycle-succ
  unfolding G'.face-cycle-succ-def face-cycle-succ-def
  by (auto simp: fun-eq-iff rev-swap-def perm-swap-comp)

```

```

have  $\bigwedge x. G'.\text{face-cycle-set } x = (a \Rightarrow_F b) \text{ ‘ face-cycle-set } ((a \Rightarrow_F b) x)$ 
  by (auto simp: face-cycle-set-def G'.face-cycle-set-def orbit-perm-swap G'-fcs
imageI)

```

```

then have G'.face-cycle-sets = ( $\lambda S. (a \Rightarrow_F b) \text{ ‘ } S$ ) ‘ face-cycle-sets
  by (auto simp: pre-digraph-map.face-cycle-sets-def swap-in-arcs)
  (metis swap-swap-id image-eqI swap-in-arcs)

```

```

then have card G'.face-cycle-sets = card (( $\lambda S. (a \Rightarrow_F b) \text{ ‘ } S$ ) ‘ face-cycle-sets)
by simp

```

```

also have ... = card face-cycle-sets

```

```

  by (rule card-image) (rule inj-on-f-imageI[where S=UNIV], auto)

```

finally

```

show pre-digraph-map.euler-genus G ?M' = euler-genus

```

```

  unfolding pre-digraph-map.euler-genus-def pre-digraph-map.euler-char-def by

```

simp

qed

end

14.4 Conclusion

lemma *bidirected-subgraph-obtain*:

assumes *sg*: *subgraph H G arcs H* \neq *arcs G*

assumes *fin*: *finite (arcs G)*

assumes *bidir*: \exists *rev*. *bidirected-digraph G rev* \exists *rev*. *bidirected-digraph H rev*

obtains *a a'* **where** $\{a, a'\} \subseteq \text{arcs } G - \text{arcs } H$ $a' \neq a$

tail G a' = head G a *head G a' = tail G a*

proof –

obtain *a* **where** $a: a \in \text{arcs } G - \text{arcs } H$ **using** *sg* **by** *blast*

obtain *rev-G rev-H* **where** *rev*: *bidirected-digraph G rev-G* *bidirected-digraph H rev-H*

using *bidir* **by** *blast*

interpret *G*: *bidirected-digraph G rev-G* **by** (*rule rev*)

interpret *H*: *bidirected-digraph H rev-H* **by** (*rule rev*)

have *sg-props*: *arcs H* \subseteq *arcs G* *tail H = tail G* *head H = head G*

using *sg* **by** (*auto simp: subgraph-def compatible-def*)

{ **fix** *w1 w2* **assume** *A*: *tail G a = w1* *head G a = w2*

have *in-arcs H w1* \cap *out-arcs H w2 = rev-H* ' (*out-arcs H w1* \cap *in-arcs H w2*)

(**is** *?Sh = -*)

unfolding *H.in-arcs-eq* **by** (*simp add: image-Int image-image H.inj-on-arev*)

then have *card (in-arcs H w1* \cap *out-arcs H w2) = card (out-arcs H w1* \cap *in-arcs H w2)*

by (*metis card-image H.arev-arev inj-on-inverseI*)

also have $\dots <$ *card (out-arcs G w1* \cap *in-arcs G w2)* (**is** *card ?Sh1 <* *card ?Sg1*)

proof (*rule psubset-card-mono*)

show *finite ?Sg1* **using** *fin* **by** (*auto simp: out-arcs-def*)

show *?Sh1* \subset *?Sg1* **using** *A a sg-props* **by** *auto*

qed

also have *?Sg1 = rev-G* ' (*in-arcs G w1* \cap *out-arcs G w2)* (**is** $- = -$ ' *?Sg*)

unfolding *G.in-arcs-eq* **by** (*simp add: image-Int image-image G.inj-on-arev*)

also have *card* $\dots =$ *card ?Sg*

by (*metis card-image G.arev-arev inj-on-inverseI*)

finally have *card-less*: *card ?Sh <* *card ?Sg* .

have *S-ss*: *?Sh* \subseteq *?Sg* **using** *sg-props* **by** *auto*

have *?thesis*

proof (*cases w1 = w2*)

case *True*

have *card (?Sh - {a}) = card ?Sh*

using *a* **by** (*intro arg-cong[where f=card]*) *auto*

also have $\dots <$ *card ?Sg - 1*

proof –

from *True* **have** *even (card ?Sg)* *even (card ?Sh)*

by (*auto simp: G.even-card-loops H.even-card-loops*)

then show *?thesis* **using** *card-less*
by *simp (metis Suc-pred even-Suc le-neq-implies-less lessE less-Suc-eq-le zero-less-Suc)*
qed
also have $\dots = \text{card } (?Sg - \{a\})$
using *fin a A True by (auto simp: out-arcs-def card-Diff-singleton)*
finally have *card-diff-a-less: card (?Sh - {a}) < card (?Sg - {a}) .*
moreover
from *S-ss* **have** $?Sh - \{a\} \subseteq ?Sg - \{a\}$ **using** *S-ss* **by** *blast*
ultimately have $?Sh - \{a\} \subset ?Sg - \{a\}$
by *(intro card-psubset) auto*
then obtain a' **where** $a' \in (?Sg - \{a\}) - ?Sh$ **by** *blast*
then have $\{a, a'\} \subseteq \text{arcs } G - \text{arcs } H$ $a' \neq a$ $\text{tail } G$ $a' = \text{head } G$ a $\text{head } G$
 $a' = \text{tail } G$ a
using *A a sg-props* **by** *auto*
then show *?thesis ..*
next
case *False*
from *card-less S-ss* **have** $?Sh \subset ?Sg$ **by** *auto*
then obtain a' **where** $a' \in ?Sg - ?Sh$ **by** *blast*
then have $\{a, a'\} \subseteq \text{arcs } G - \text{arcs } H$ $a' \neq a$ $\text{tail } G$ $a' = \text{head } G$ a $\text{head } G$
 $a' = \text{tail } G$ a
using *A a sg-props False* **by** *auto*
then show *?thesis ..*
qed
}
then show *?thesis* **by** *simp*
qed

lemma *subgraph-euler-genus-le:*

assumes *G: subgraph H G digraph-map G GM and H: \exists rev. bidirected-digraph H rev*

obtains *HM* **where** *digraph-map H HM pre-digraph-map.euler-genus H HM \leq pre-digraph-map.euler-genus G GM*

proof –

let $?d = \lambda G. \text{card } (\text{arcs } G) + \text{card } (\text{verts } G) - \text{card } (\text{arcs } H) - \text{card } (\text{verts } H)$

from *H* **obtain** *rev-H* **where** *bidirected-digraph H rev-H* **by** *blast*

then interpret *H: bidirected-digraph H rev-H .*

from *G*

have \exists *HM. digraph-map H HM \wedge pre-digraph-map.euler-genus H HM \leq pre-digraph-map.euler-genus G GM*

proof *(induct ?d G arbitrary: G GM rule: less-induct)*

case *less*

from *less* **interpret** *G: digraph-map G GM* **by** –

have *H-ss: arcs H \subseteq arcs G verts H \subseteq verts G* **using** $\langle \text{subgraph } H \ G \rangle$ **by** *auto*
then have *card-le: card (arcs H) \leq card (arcs G) card (verts H) \leq card (verts*

G)

```

by (auto intro: card-mono)

have ends: tail H = tail G head H = head G
using ⟨subgraph H G⟩ by (auto simp: compatible-def)

show ?case
proof (cases ?d G = 0)
  case True
    then have card (arcs H) = card (arcs G) card (verts H) = card (verts G)
      using card-le by linarith+
    then have arcs H = arcs G verts H = verts G
      using H-ss by (auto simp: card-subset-eq)
    then have H = G using ⟨subgraph H G⟩ by (auto simp: compatible-def)
      then have digraph-map H GM ∧ pre-digraph-map.euler-genus H GM ≤
G.euler-genus by auto
    then show ?thesis ..
  next
    case False
      then have H-ne: (arcs G - arcs H) ≠ {} ∨ (verts G - verts H) ≠ {}
        using H-ss card-le by auto

      { assume A: arcs G - arcs H ≠ {}
        then obtain a a' where aa': {a, a'} ⊆ arcs G - arcs H a' ≠ a tail G a'
= head G a head G a' = tail G a
          using H-ss ⟨subgraph H G⟩ by (auto intro: bidirected-subgraph-obtain)
        let ?GM' = G.rev-swap (edge-rev GM a) a'

        interpret G': digraph-map G ?GM'
          using aa' by (intro G.digraph-map-rev-swap) (auto simp: arc-to-ends-def)
        interpret G': bidel-arc G ?GM' a
          using aa' by unfold-locales simp

        have edge-rev GM a ≠ a
          using aa' by (intro G.arev-neq) auto
        then have er-a: edge-rev ?GM' a = a'
          using ⟨a' ≠ a⟩ by (auto simp: G.rev-swap-def perm-swap-def swap-id-eq
dest: G.arev-neq)
        then have sg: subgraph H G'.H
          using H-ss aa' by (intro subgraphI) (auto simp: G'.verts-H G'.arcs-H
G'.tail-H G'.head-H ends compatible-def intro: H.wf-digraph G'.H.wf-digraph)

        have card {a,a'} ≤ card (arcs G)
          using aa' by (intro card-mono) auto
        then obtain HM where HM: digraph-map H HM pre-digraph-map.euler-genus
H HM ≤ G'.H.euler-genus
          using aa' False by atomize-elim (rule less, auto simp: G'.verts-H G'.arcs-H
card-insert-if sg er-a)

```

```

    have  $G'.H.euler-genus \leq G'.euler-genus$  by (rule  $G'.euler-genus-da$ )
    also have  $G'.euler-genus = G.euler-genus$ 
      using  $aa'$  by (auto simp:  $G.euler-genus-rev-swap$   $arc-to-ends-def$ )
    finally have  $?thesis$  using  $HM$  by auto
  }
  moreover
  { assume  $A: arcs\ G - arcs\ H = \{\}$ 
    then have  $A': verts\ G - verts\ H \neq \{\}$  and  $arcs-H: arcs\ H = arcs\ G$  using
  H-ss  $H-ne$  by auto
    then obtain  $v$  where  $v: v \in verts\ G - verts\ H$  by auto
    have  $card-lt: card\ (verts\ H) < card\ (verts\ G)$ 
      using  $A' H-ss$  by (intro  $psubset-card-mono$ ) auto

    have  $out-arcs\ G\ v = out-arcs\ H\ v$  using  $A H-ss$  by (auto simp:  $ends$ )
    then interpret  $G: del-vert-props\ G\ GM\ v$ 
      using  $v$  by  $unfold-locales$  auto

    have  $?d\ (G.del-vert\ v) < ?d\ G$ 
      using  $card-lt$  by (simp add:  $arcs-H\ G.arcs-dv\ G.card-verts-dv$ )
    moreover
    have  $subgraph\ H\ (G.del-vert\ v)$ 
      using  $H-ss\ v$  by (auto simp:  $subgraph-def\ arcs-H\ G.arcs-dv\ G.verts-del-vert$ 
  H.wf-digraph
       $G.H.wf-digraph\ compatible-def\ G.tail-del-vert\ G.head-del-vert\ ends$ )
    moreover
    have  $bidirected-digraph\ (G.del-vert\ v)\ (edge-rev\ GM)$ 
      using  $G.arev-dom$  by (intro  $G.H.bidirected-digraphI$ ) (auto simp:  $G.arcs-dv$ )
    ultimately
    have  $?thesis$  unfolding  $G.euler-genus-eq[symmetric]$  by (intro  $less$ ) auto
  }
  ultimately show  $?thesis$  by blast
qed
qed
then obtain  $HM$  where  $digraph-map\ H\ HM\ pre-digraph-map.euler-genus\ H\ HM$ 
 $\leq pre-digraph-map.euler-genus\ G\ GM$ 
  by  $atomize-elim$ 
then show  $?thesis$  ..
qed

lemma (in  $digraph-map$ )  $nonneg-euler-genus: 0 \leq euler-genus$ 
proof -
  define  $H$  where  $H = (\ |verts = \{\}, arcs = \{\}, tail = tail\ G, head = head\ G\ |)$ 
  then have  $H-simps: verts\ H = \{\} arcs\ H = \{\} tail\ H = tail\ G head\ H = head\ G$ 
    by (simp-all add:  $H-def$ )

  interpret  $H: bidirected-digraph\ H\ id$ 
  by  $unfold-locales$  (auto simp:  $H-def$ )
  have  $wf-digraph\ H\ wf-digraph\ G$  by  $unfold-locales$ 

```

```

then have subgraph  $H$   $G$  by (intro subgraphI) (auto simp: H-def compatible-def)
then obtain  $HM$  where digraph-map  $H$   $HM$  pre-digraph-map.euler-genus  $H$   $HM$ 
 $\leq$  euler-genus
  by (rule subgraph-euler-genus-le) auto
then interpret  $H$ : digraph-map  $H$   $HM$  by –

have  $H.sccs = \{\}$ 
proof –
  { fix  $x$  assume  $*$ :  $x \in H.sccs-verts$ 
    then have  $x = \{\}$  by (auto dest: H.sccs-verts-subsets simp: H-simps)
    with  $*$  have False by (auto simp: H.in-sccs-verts-conv-reachable)
  } then show ?thesis by (auto simp: H.sccs-verts-conv)
qed
then have  $H.euler-genus = 0$ 
  by (auto simp: H.euler-genus-def H.euler-char-def H.isolated-verts-def H.face-cycle-sets-def
H-simps)
  then show ?thesis using  $\langle H.euler-genus \leq \rightarrow \rangle$  by simp
qed

lemma subgraph-comb-planar:
  assumes subgraph  $G$   $H$  comb-planar  $H \exists rev. bidirected-digraph$   $G$   $rev$  shows
comb-planar  $G$ 
proof –
  from  $\langle comb-planar\ H \rangle$  obtain  $HM$  where digraph-map  $H$   $HM$  and  $H-genus$ :
pre-digraph-map.euler-genus  $H$   $HM = 0$ 
  unfolding comb-planar-def by metis

  obtain  $GM$  where  $G$ : digraph-map  $G$   $GM$  pre-digraph-map.euler-genus  $G$   $GM$ 
 $\leq$  pre-digraph-map.euler-genus  $H$   $HM$ 
  using assms(1)  $\langle digraph-map\ H\ HM \rangle$  assms(3) by (rule subgraph-euler-genus-le)
  interpret  $G$ : digraph-map  $G$   $GM$  by fact

  show ?thesis using  $G$   $H-genus$   $G.nonneg-euler-genus$  unfolding comb-planar-def
by auto
qed

end
theory Kuratowski-Combinatorial
imports
  Planar-Complete
  Planar-Subdivision
  Planar-Subgraph
begin

theorem comb-planar-compat:
  assumes comb-planar  $G$ 
  shows kuratowski-planar  $G$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 

```

```

then obtain  $G0$   $rev-G0$   $K$   $rev-K$  where  $sub: subgraph\ G0\ G\ subdivision\ (K,$ 
 $rev-K)\ (G0, rev-G0)$ 
  and  $is-kur: K_{3,3}\ K \vee K_5\ K$ 
  unfolding  $kuratowski-planar-def$  by  $auto$ 

have  $comb-planar\ K$  using  $sub\ assms$ 
  by  $(metis\ subgraph-comb-planar\ subdivision-comb-planar\ subdivision-bidir)$ 
moreover
have  $\neg comb-planar\ K$  using  $is-kur$  by  $(metis\ K5-not-comb-planar\ K33-not-comb-planar)$ 
ultimately
show  $False$  by  $contradiction$ 
qed

```

```

end
theory  $Simpl-Anno$  imports  $Simpl.Vcg$  begin

```

```

definition  $named-loop\ name = UNIV$ 

```

```

lemma  $annotate-named-loop-inv:$ 
   $whileAnno\ b\ (named-loop\ name)\ V\ c = whileAnno\ b\ I\ V\ c$ 
by  $(simp\ add: whileAnno-def)$ 

```

```

lemma  $annotate-named-loop-inv-fix:$ 
   $whileAnno\ b\ (named-loop\ name)\ V\ c = whileAnnoFix\ b\ I\ (\lambda-. V)\ (\lambda-. c)$ 
by  $(simp\ add: whileAnno-def\ whileAnnoFix-def)$ 

```

```

lemma  $annotate-named-loop-var:$ 
   $whileAnno\ b\ (named-loop\ name)\ V'\ c = whileAnno\ b\ I\ V\ c$ 
by  $(simp\ add: whileAnno-def)$ 

```

```

lemma  $annotate-named-loop-var-fix:$ 
   $whileAnno\ b\ (named-loop\ name)\ V'\ c = whileAnnoFix\ b\ I\ (\lambda-. V)\ (\lambda-. c)$ 
by  $(simp\ add: whileAnno-def\ whileAnnoFix-def)$ 

```

```

end

```

15 Implementation of a Non-Planarity Checker

```

theory  $Check-Non-Planarity-Impl$ 
imports
   $Simpl.Vcg$ 
   $Simpl-Anno$ 
   $Graph-Theory.Graph-Theory$ 
begin

```

15.1 An abstract graph datatype

```

type-synonym  $ig-vertex = nat$ 

```

```

type-synonym ig-edge = ig-vertex × ig-vertex

typedef IGraph = {(vs :: ig-vertex list, es :: ig-edge list). distinct vs}
  by auto

definition ig-verts :: IGraph ⇒ ig-vertex list where
  ig-verts G ≡ fst (Rep-IGraph G)

definition ig-arcs :: IGraph ⇒ ig-edge list where
  ig-arcs G ≡ snd (Rep-IGraph G)

definition ig-verts-cnt :: IGraph ⇒ nat
  where ig-verts-cnt G ≡ length (ig-verts G)

definition ig-arcs-cnt :: IGraph ⇒ nat
  where ig-arcs-cnt G ≡ length (ig-arcs G)

declare ig-verts-cnt-def[simp]
declare ig-arcs-cnt-def[simp]

definition IGraph-inv :: IGraph ⇒ bool where
  IGraph-inv G ≡ (∀ e ∈ set (ig-arcs G). fst e ∈ set (ig-verts G) ∧ snd e ∈ set (ig-verts G))

definition ig-empty :: IGraph where
  ig-empty ≡ Abs-IGraph ([], [])

definition ig-add-v :: IGraph ⇒ ig-vertex ⇒ IGraph where
  ig-add-v G v = (if v ∈ set (ig-verts G) then G else Abs-IGraph (ig-verts G @ [v], ig-arcs G))

definition ig-add-e :: IGraph ⇒ ig-vertex ⇒ ig-vertex ⇒ IGraph where
  ig-add-e G u v ≡ Abs-IGraph (ig-verts G, ig-arcs G @ [(u,v)])

definition ig-in-out-arcs :: IGraph ⇒ ig-vertex ⇒ ig-edge list where
  ig-in-out-arcs G u ≡ filter (λe. fst e = u ∨ snd e = u) (ig-arcs G)

definition ig-opposite :: IGraph ⇒ ig-edge ⇒ ig-vertex ⇒ ig-vertex where
  ig-opposite G e u = (if fst e = u then snd e else fst e)

definition ig-neighbors :: IGraph ⇒ ig-vertex ⇒ ig-vertex set where
  ig-neighbors G u ≡ {v ∈ set (ig-verts G). (u,v) ∈ set (ig-arcs G) ∨ (v,u) ∈ set (ig-arcs G)}

```

15.2 Code

```

procedures is-subgraph (G :: IGraph, H :: IGraph | R :: bool)
  where
    i :: nat

```

```

v :: ig-vertex
ends :: ig-edge
in
  TRY
    'i ::= 0 ;;
    WHILE 'i < ig-verts-cnt 'G INV named-loop "verts"
    DO
      'v ::= ig-verts 'G ! 'i ;;
      IF 'v ∉ set (ig-verts 'H) THEN
        RAISE 'R ::= False
      FI ;;
      'i ::= 'i + 1
    OD ;;

    'i ::= 0 ;;
    WHILE 'i < ig-arcs-cnt 'G INV named-loop "arcs"
    DO
      'ends ::= ig-arcs 'G ! 'i ;;
      IF 'ends ∉ set (ig-arcs 'H) ∧ (snd 'ends, fst 'ends) ∉ set (ig-arcs 'H)
    THEN
      RAISE 'R ::= False
      FI ;;
      IF fst 'ends ∉ set (ig-verts 'G) ∨ snd 'ends ∉ set (ig-verts 'G) THEN
        RAISE 'R ::= False
      FI ;;
      'i ::= 'i + 1
    OD ;;
    'R ::= True
  CATCH SKIP END

```

procedures *is-loopfree* (*G* :: *IGraph* | *R* :: *bool*)

where

```

i :: nat
ends :: ig-edge
edge-map :: ig-edge ⇒ bool

```

in

```

  TRY
    'i ::= 0 ;;
    WHILE 'i < ig-arcs-cnt 'G INV named-loop "loop"
    DO
      'ends ::= ig-arcs 'G ! 'i ;;
      IF fst 'ends = snd 'ends THEN
        RAISE 'R ::= False
      FI ;;
      'i ::= 'i + 1
    OD ;;
    'R ::= True
  CATCH SKIP END

```

procedures *select-nodes* (*G* :: *IGraph* | *R* :: *IGraph*)

where

i :: *nat*

v :: *ig-vertex*

in

'R ::= *ig-empty* ;;

'i ::= 0 ;;

WHILE *'i* < *ig-verts-cnt 'G*

INV *named-loop "loop"*

DO

'v ::= *ig-verts 'G ! 'i* ;;

IF 2 < *card (ig-neighbors 'G 'v)* **THEN**

'R ::= *ig-add-v 'R 'v*

FI ;;

'i ::= *'i + 1*

OD

procedures *find-endpoint* (*G* :: *IGraph*, *H* :: *IGraph*, *v-tail* :: *ig-vertex*, *v-next* :: *ig-vertex* | *R* :: *ig-vertex option*)

where

found :: *bool*

i :: *nat*

len :: *nat*

io-arcs :: *ig-edge list*

v0 :: *ig-vertex*

v1 :: *ig-vertex*

vt :: *ig-vertex*

in

TRY

IF *'v-tail* = *'v-next* **THEN RAISE** *'R* ::= *None* **FI** ;;

'v0 ::= *'v-tail* ;;

'v1 ::= *'v-next* ;;

'len ::= 1 ;;

WHILE *'v1* ∉ *set (ig-verts 'H)*

INV *named-loop "path"*

DO

'io-arcs ::= *ig-in-out-arcs 'G 'v1* ;;

'i ::= 0 ;;

'found ::= *False* ;;

WHILE *'found* = *False* ∧ *'i* < *length 'io-arcs*

INV *named-loop "arcs"*

DO

'vt ::= *ig-opposite 'G ('io-arcs ! 'i) 'v1* ;;

IF *'vt* ≠ *'v0* **THEN**

```

        'found ::= True ;;
        'v0 ::= 'v1 ;;
        'v1 ::= 'vt
    FI ;;
    'i ::= 'i + 1
OD ;;
'len ::= 'len + 1 ;;
IF ¬ 'found THEN RAISE 'R ::= None FI
OD ;;
IF 'v1 = 'v-tail THEN RAISE 'R ::= None FI ;;
'R ::= Some 'v1
CATCH SKIP END

```

procedures *contract* (*G* :: *IGraph*, *H* :: *IGraph* | *R* :: *IGraph*)

where

```

i :: nat
j :: nat
u :: ig-vertex
v :: ig-vertex
vo :: ig-vertex option
io-arcs :: ig-edge list

```

in

```

'i ::= 0 ;;
WHILE 'i < ig-verts-cnt 'H
INV named-loop "iter-nodes"
DO
    'u ::= ig-verts 'H ! 'i ;;
    'io-arcs ::= ig-in-out-arcs 'G 'u ;;

    'j ::= 0 ;;
    WHILE 'j < length 'io-arcs
    INV named-loop "iter-adj"
    DO
        'v ::= ig-opposite 'G ('io-arcs ! 'j) 'u ;;
        'vo ::= CALL find-endpoint('G, 'H, 'u, 'v) ;;
        IF 'vo ≠ None THEN
            'H ::= ig-add-e 'H 'u (the 'vo)
        FI ;;
        'j ::= 'j + 1
    OD ;;
    'i ::= 'i + 1
OD ;;
'R ::= 'H

```

procedures *is-K33* (*G* :: *IGraph* | *R* :: *bool*)

where

```

i :: nat

```

```

j :: nat
u :: ig-vertex
v :: ig-vertex
blue :: ig-vertex ⇒ bool
blue-cnt :: nat
io-arcs :: ig-edge list
in
TRY
  IF ig-verts-cnt 'G ≠ 6 THEN RAISE 'R ::= False FI ;;
  'blue ::= (λ-. False) ;;

  'u ::= ig-verts 'G ! 0 ;;
  'i ::= 0 ;;
  'io-arcs ::= ig-in-out-arcs 'G 'u ;;

  WHILE 'i < length 'io-arcs INV named-loop "colorize"
  DO
    'v ::= ig-opposite 'G ('io-arcs ! 'i) 'u ;;
    'blue ::= 'blue('v := True) ;;
    'i ::= 'i + 1
  OD ;;

  'blue-cnt ::= 0 ;;
  'i ::= 0 ;;
  WHILE 'i < ig-verts-cnt 'G INV named-loop "component-size"
  DO
    IF 'blue (ig-verts 'G ! 'i) THEN 'blue-cnt ::= 'blue-cnt + 1 FI ;;
    'i ::= 'i + 1
  OD ;;
  IF 'blue-cnt ≠ 3 THEN RAISE 'R ::= False FI ;;

  'i ::= 0 ;;
  WHILE 'i < ig-verts-cnt 'G INV named-loop "connected-outer"
  DO
    'u ::= ig-verts 'G ! 'i ;;
    'j ::= 0 ;;
    WHILE 'j < ig-verts-cnt 'G INV named-loop "connected-inner"
    DO
      'v ::= ig-verts 'G ! 'j ;;
      IF ¬(('blue 'u = 'blue 'v) ↔ ('u, 'v) ∉ set (ig-arcs 'G)) THEN RAISE
'R ::= False FI ;;
      'j ::= 'j + 1
    OD ;;
    'i ::= 'i + 1
  OD ;;
  'R ::= True
CATCH SKIP END

```

```

procedures is-K5 (G :: IGraph | R :: bool)
  where
    i :: nat
    j :: nat
    u :: ig-vertex
  in
    TRY
      IF ig-verts-cnt 'G ≠ 5 THEN RAISE 'R ::= False FI ;;
      'i ::= 0 ;;
      WHILE 'i < 5 INV named-loop "outer-loop"
        DO
          'u ::= ig-verts 'G ! 'i ;;
          'j ::= 0 ;;
          WHILE 'j < 5 INV named-loop "inner-loop"
            DO
              IF ¬('i ≠ 'j ↔ ('u, ig-verts 'G ! 'j) ∈ set (ig-arcs 'G))
                THEN
                  RAISE 'R ::= False
                FI ;;
              'j ::= 'j + 1
            OD ;;
          'i ::= 'i + 1
        OD ;;
      'R ::= True
    CATCH SKIP END

```

```

procedures check-kuratowski (G :: IGraph, K :: IGraph | R :: bool)
  where
    H :: IGraph
  in
    TRY
      'R ::= CALL is-subgraph('K, 'G) ;;
      IF ¬'R THEN RAISE 'R ::= False FI ;;
      'R ::= CALL is-loopfree('K) ;;
      IF ¬'R THEN RAISE 'R ::= False FI ;;
      'H ::= CALL select-nodes('K) ;;
      'H ::= CALL contract('K, 'H) ;;
      'R ::= CALL is-K5('H) ;;
      IF 'R THEN RAISE 'R ::= True FI ;;
      'R ::= CALL is-K33('H)
    CATCH SKIP END

```

end

16 Verification of a Non-Planarity Checker

theory *Check-Non-Planarity-Verification* **imports**

Check-Non-Planarity-Impl

../Planarity/Kuratowski-Combinatorial

HOL-Library.Rewrite

HOL-Eisbach.Eisbach

begin

16.1 Graph Basics and Implementation

context *pre-digraph* **begin**

lemma *cas-nonempty-ends*:

assumes $p \neq []$ *cas* $u\ p\ v$ *cas* $u'\ p\ v'$

shows $u = u'\ v = v'$

using *assms* **apply** (*metis cas-simp*)

using *assms* **by** (*metis append-Nil2 cas.simps(1) cas-append-iff cas-simp*)

lemma *awalk-nonempty-ends*:

assumes $p \neq []$ *awalk* $u\ p\ v$ *awalk* $u'\ p\ v'$

shows $u = u'\ v = v'$

using *assms* **by** (*auto simp: awalk-def intro: cas-nonempty-ends*)

end

lemma (*in pair-graph*) *verts2-awalk-distinct*:

assumes $V: \text{verts3 } G \subseteq V\ V \subseteq \text{pverts } G\ u \in V$

assumes $p: \text{awalk } u\ p\ v\ \text{set } (\text{inner-verts } p) \cap V = \{\}$ *progressing* p

shows *distinct* (*inner-verts* p)

using p

proof (*induct* p *arbitrary: v* *rule: rev-induct*)

case *Nil* **then show** *?case* **by** *auto*

next

case (*snoc* $e\ es$)

have *distinct* (*inner-verts* es)

apply (*rule snoc.hyps*)

using *snoc.prem* **apply** (*auto dest: progressing-appendD1*)

apply (*metis (opaque-lifting, no-types) disjoint-iff-not-equal in-set-inner-verts-appendI-l*)

done

show *?case*

proof (*rule ccontr*)

assume $A: \neg ?thesis$

then obtain $es'\ e'$ **where** $es = es' @ [e']\ es \neq []$

by (*cases* es *rule: rev-cases*) *auto*

have $fst\ e \in \text{set } (\text{inner-verts } es)$

using A $\langle \text{distinct } (\text{inner-verts } es) \rangle$ $\langle es \neq [] \rangle$

by (*auto simp: inner-verts-def*)

moreover

```

have fst e' ≠ fst e snd e' = fst e
  using ⟨es = es' @ [e']⟩ snoc.prem1
  by (auto simp: awalk-Cons-iff dest: no-loops)
ultimately
obtain es'' e'' where es' = es'' @ [e'']
  by (cases es' rule: rev-cases) (auto simp: ⟨es = es' @ [e']⟩ inner-verts-def)
then have fst e'' ≠ fst e
  using ⟨snd e' = fst e⟩[symmetric] snoc.prem1,3 unfolding ⟨es = -⟩
  by (simp add: ⟨es = -⟩ awalk-Cons-iff progressing-append-iff progressing-Cons)

have fst e' ∈ set (inner-verts es)
  using ⟨es = es' @ [e']⟩ ⟨es' = es'' @ [e'']⟩
  by (cases es'') (auto simp: inner-verts-def)

have fst e ∈ set (inner-verts es')
  using ⟨es = es' @ [e']⟩ ⟨fst e ∈ set (inner-verts es)⟩ ⟨fst e' ≠ fst e⟩
  by (cases es') (auto simp: inner-verts-def)
then obtain q e'2 e'3 r where Z: es' = q @ [e'2, e'3] @ r snd e'2 = fst e fst
e'3 = fst e
proof -
  obtain e'3' where e'3' ∈ set (tl es') fst e'3' = fst e
    using ⟨fst e ∈ set (inner-verts es')⟩
    by (cases es') (auto simp: inner-verts-def)
  then obtain q r where tl es' = q @ e'3' # r
    by (metis split-list)
  then have F2: snd (last (hd es' # q)) = fst e
    using ⟨es = es' @ [e']⟩ snoc.prem1 ⟨fst e'3' = fst e⟩
    apply (cases es')
    apply (case-tac [2] q rule: rev-cases)
    apply auto
  done
  then have es' = (butlast (hd es' # q)) @ [last (hd es' # q), e'3'] @ r
    using ⟨tl es' = q @ e'3' # r⟩ by (cases es') auto
  then show ?thesis using F2 ⟨fst e'3' = fst e⟩ by fact
qed
then have fst e'2 ≠ snd e'3
  using snoc.prem3 unfolding ⟨es = -⟩
  by (simp add: progressing-append-iff progressing-Cons)
moreover
from Z have B: fst e'2 = u ∨ fst e'2 ∈ set (inner-verts es')
  using ⟨es = es' @ [e']⟩ snoc.prem1
  by (cases q) (auto simp: inner-verts-def)
then have fst e'2 ≠ fst e'
proof
  assume fst e'2 = u
  then have fst e'2 ∉ set (inner-verts es)
    using V ⟨es = es' @ [e']⟩ snoc.prem2
    by (cases es') (auto simp: inner-verts-def)
  moreover

```

```

have  $\text{fst } e' \in \text{set } (\text{inner-verts } es)$ 
  using  $\langle es = es' @ [e'] \rangle \langle es' = es'' @ [e''] \rangle$ 
  by  $(\text{cases } es'')$   $(\text{auto simp: inner-verts-def})$ 
ultimately show  $?thesis$  by auto
next
assume  $\text{fst } e'2 \in \text{set } (\text{inner-verts } es')$ 
moreover
have  $\text{fst } e' \in \text{set } (\text{inner-verts } es)$ 
  using  $\langle es = es' @ [e'] \rangle \langle es' = es'' @ [e''] \rangle$ 
  by  $(\text{cases } es'')$   $(\text{auto simp: inner-verts-def})$ 
ultimately
show  $?thesis$ 
  using  $\langle \text{distinct } (\text{inner-verts } es) \rangle$  unfolding  $\langle es = es' @ [e'] \rangle$ 
  by  $(\text{cases } es')$   $(\text{fastforce simp: inner-verts-def})+$ 
qed
moreover
have  $\text{snd } e'3 \neq \text{fst } e'$ 
proof  $(\text{rule notI, cases})$ 
  assume  $r = []$   $\text{snd } e'3 = \text{fst } e'$ 
  then show  $\text{False}$  using  $Z \langle es = es' @ [e'] \rangle \text{snoc.prem}(3) \langle \text{snd } e' = \text{fst } e \rangle$ 
  by  $(\text{simp add: progressing-append-iff progressing-Cons})$ 
next
assume  $A: r \neq []$   $\text{snd } e'3 = \text{fst } e'$ 
then obtain  $r0$   $rs$  where  $r = r0 \# rs$  by  $(\text{cases } r)$  auto
then have  $\text{snd } e'3 = \text{fst } r0$ 
  using  $Z \langle es = es' @ [e'] \rangle \text{snoc.prem}(1)$ 
  by  $(\text{auto simp: awalk-Cons-iff})$ 
with  $A$  have  $\text{fst } r0 = \text{fst } e'$  by auto
have  $\neg \text{distinct } (\text{inner-verts } es)$ 
  by  $(\text{cases } q)$   $(\text{auto simp add: } Z(1) \langle es = es' @ [e'] \rangle$ 
   $\langle r = r0 \# rs \rangle \langle \text{fst } r0 = \text{fst } e' \rangle \text{inner-verts-def})$ 
then show  $\text{False}$  using  $\langle \text{distinct } (\text{inner-verts } es) \rangle$  by auto
qed
ultimately
have  $\text{card-to-fst-e: card } \{e'2, (\text{snd } e'3, \text{fst } e'3), e'\} = 3$ 
  by  $(\text{auto simp: card-insert-if})$ 
moreover
have  $e'3 \in \text{parcs } G$ 
  using  $Z$  using  $\text{snoc.prem}(1) \langle es = es' @ [e'] \rangle$ 
  by  $(\text{auto intro: arcs-symmetric})$ 
then have  $(\text{snd } e'3, \text{fst } e'3) \in \text{parcs } G$ 
  by  $(\text{auto intro: arcs-symmetric})$ 
then have  $\{e'2, (\text{snd } e'3, \text{fst } e'3), e'\} \subseteq \{ed \in \text{parcs } G. \text{snd } ed = \text{fst } e\}$ 
  using  $\text{snoc.prem}(1) \langle es = es' @ [e'] \rangle Z$  by auto
moreover
have  $\text{fst } e \in \text{pverts } G$  using  $\text{snoc.prem}(1)$  by auto
then have  $\text{card-to-fst-e-abs: card } \{ed \in \text{parcs } G. \text{snd } ed = \text{fst } e\} \leq 2$ 
  using  $\langle \text{fst } e \in \text{set } (\text{inner-verts } es) \rangle V \text{snoc.prem}(2)$ 
  unfolding  $\text{verts3-def in-degree-def}$ 

```

```

    by (cases es) (auto simp: inner-verts-def in-arcs-def)
  ultimately
  have {e'2, (snd e'3, fst e'3), e'} = {ed ∈ parcs G. snd ed = fst e}
    by (intro card-seteq) auto
  then show False
    using card-to-fst-e card-to-fst-e-abs by auto
qed
qed

```

```

lemma (in wf-digraph) inner-verts-conv':
  assumes awalk u p v 2 ≤ length p shows inner-verts p = awalk-verts (head G
(hd p)) (butlast (tl p))
  using assms
  apply (cases p)
  apply (auto simp: awalk-Cons-iff; fail)
  apply (match premises in p = - # as for as ⇒ ⟨cases as rule: rev-cases⟩)
  apply (auto simp: inner-verts-def awalk-verts-conv)
  done

```

```

lemma verts3-in-verts:
  assumes x ∈ verts3 G shows x ∈ verts G
  using assms unfolding verts3-def by auto

```

```

lemma (in pair-graph) deg2-awalk-is-iapath:
  assumes V: verts3 G ⊆ V V ⊆ pverts G
  assumes p: awalk u p v set (inner-verts p) ∩ V = {} progressing p
  assumes in-V: u ∈ V v ∈ V
  assumes u ≠ v
  shows gen-iapath V u p v
proof (cases p)
  case Nil then show ?thesis using p(1) in-V ⟨u ≠ v⟩ by (auto simp: apath-def
gen-iapath-def)
next
  case (Cons p0 ps)
  then have ev-p: awalk-verts u p = u # butlast (tl (awalk-verts u p)) @ [v]
    using p(1) by (cases p) auto

  have u ∉ set (inner-verts p) v ∉ set (inner-verts p)
    using p(2) in-V by auto
  with verts2-awalk-distinct[OF V in-V(1) p] have distinct (awalk-verts u p)
    using p(1) ⟨u ≠ v⟩ by (subst ev-p) (auto simp: inner-verts-conv[of p u]
verts3-def)
  then show ?thesis using p(1-2) in-V ⟨u ≠ v⟩ by (auto simp: apath-def
gen-iapath-def)
qed

```

```

lemma (in pair-graph) inner-verts-min-degree:
  assumes walk-p: awalk u p v and progress: progressing p

```

and $w-p$: $w \in \text{set } (\text{inner-verts } p)$
shows $2 \leq \text{in-degree } G w$

proof –

from $w-p$ **have** $2 \leq \text{length } p$ **using** *not-le* **by** *fastforce*
moreover
then obtain $e1$ es $e2$ **where** $p\text{-decomp}$: $p = e1 \# es @ [e2]$
by (*metis One-nat-def Suc-1 Suc-eq-plus1 le0 list.size(3) list.size(4) neq-Nil-conv not-less-eq-eq rev-cases*)
ultimately
have $w-es$: $w \in \text{set } (\text{awalk-verts } (\text{snd } e1) es)$
using $\text{walk-}p$ $w-p$ **by** (*auto simp: apath-def inner-verts-conv'*)

have walk-es : $\text{awalk } (\text{snd } e1) es (\text{fst } e2)$
using $\text{walk-}p$ **by** (*auto simp: p-decomp awalk-simps*)
obtain q r **where** $es\text{-decomp}$: $es = q @ r$ $\text{awalk } (\text{snd } e1) q w$ $\text{awalk } w r (\text{fst } e2)$
using $\text{awalk-decomp}[OF \text{walk-es } w-es]$ **by** *auto*

define xs x y ys
where $xs = \text{butlast } (e1 \# q)$ **and** $x = \text{last } (e1 \# q)$
and $y = \text{hd } (r @ [e2])$ **and** $ys = \text{tl } (r @ [e2])$
then have $p = xs @ x \# y \# ys$
by (*auto simp: p-decomp es-decomp*)
moreover
have $\text{awalk } u (e1 \# q) w$ $\text{awalk } w (r @ [e2]) v$
using $\text{walk-}p$ $es\text{-decomp}$ $p\text{-decomp}$ **by** (*auto simp: awalk-Cons-iff*)
then have $\text{inc-}w$: $\text{snd } x = w$ $\text{fst } y = w$
unfolding $x\text{-def}$ $y\text{-def}$
apply –
apply (*auto simp: awalk-Cons-iff awalk-verts-conv; fail*)
apply (*cases r*)
apply *auto*
done

ultimately have $\text{fst } x \neq \text{snd } y$
using progress **by** (*auto simp: progressing-append-iff progressing-Cons*)

have $x \in \text{parcs } G$ $y \in \text{parcs } G$
using $\text{walk-}p$ $\langle p = xs @ x \# y \# ys \rangle$ **by** *auto*
then have $\{x, (\text{snd } y, w)\} \subseteq \{e \in \text{parcs } G. \text{snd } e = w\}$
using $\text{inc-}w$ **by** *auto (metis arcs-symmetric surjective-pairing)*
then have $\text{card } \{x, (\text{snd } y, w)\} \leq \text{in-degree } G w$
unfolding in-degree-def **by** (*intro card-mono*) *auto*
then show $?thesis$ **using** $\langle \text{fst } x \neq \text{snd } y \rangle$ $\text{inc-}w$
by (*auto simp: card-insert-if split: if-split-asm*)

qed

lemma (*in pair-pseudo-graph*) $\text{gen-iapath-same}2E$:
assumes $\text{verts}3$ $G \subseteq V$ $V \subseteq \text{pverts } G$
and $\text{gen-iapath } V u p$ $\text{gen-iapath } V w q$ x
and $e \in \text{set } p$ $e \in \text{set } q$

obtains $p = q$
using *assms same-gen-iapath-by-common-arc* **by** *metis*

definition $mk\text{-graph}' :: IGraph \Rightarrow ig\text{-vertex pair-pre-digraph}$ **where**
 $mk\text{-graph}' IG \equiv (\!| pverts = set (ig\text{-verts } IG), parcs = set (ig\text{-arcs } IG)\!|)$

definition $mk\text{-graph} :: IGraph \Rightarrow ig\text{-vertex pair-pre-digraph}$ **where**
 $mk\text{-graph } IG \equiv mk\text{-symmetric } (mk\text{-graph}' IG)$

lemma $verts\text{-mkg}'$: $pverts (mk\text{-graph}' G) = set (ig\text{-verts } G)$
unfolding $mk\text{-graph}'\text{-def}$ **by** *simp*

lemma $arcs\text{-mkg}'$: $parcs (mk\text{-graph}' G) = set (ig\text{-arcs } G)$
unfolding $mk\text{-graph}'\text{-def}$ **by** *simp*

lemmas $mkg'\text{-simps} = verts\text{-mkg}' arcs\text{-mkg}'$

lemma $verts\text{-mkg}$: $pverts (mk\text{-graph } G) = set (ig\text{-verts } G)$
unfolding $mk\text{-graph}\text{-def}$ **by** (*simp add: mkg'\text{-simps}*)

lemma $parcs\text{-mk-symmetric-symcl}$: $parcs (mk\text{-symmetric } G) = (arcs\text{-ends } G)^s$
by (*auto simp: parcs\text{-mk-symmetric symcl-def arcs\text{-ends-conv}*)

lemma $arcs\text{-mkg}$: $parcs (mk\text{-graph } G) = (set (ig\text{-arcs } G))^s$
unfolding $mk\text{-graph}\text{-def}$ $parcs\text{-mk-symmetric-symcl}$ **by** (*simp add: arcs\text{-mkg}'*)

lemmas $mkg\text{-simps} = verts\text{-mkg } arcs\text{-mkg}$

definition $iadj :: IGraph \Rightarrow ig\text{-vertex} \Rightarrow ig\text{-vertex} \Rightarrow bool$ **where**
 $iadj G u v \equiv (u,v) \in set (ig\text{-arcs } G) \vee (v,u) \in set (ig\text{-arcs } G)$

definition $loop\text{-free } G \equiv (\forall e \in parcs G. fst e \neq snd e)$

lemma $ig\text{-opposite-simps}$:
 $ig\text{-opposite } G (u,v) u = v$ $ig\text{-opposite } G (v,u) u = v$
unfolding $ig\text{-opposite}\text{-def}$ **by** *auto*

lemma $distinct\text{-ig-verts}$:
 $distinct (ig\text{-verts } G)$
by (*cases G*) (*auto simp: ig\text{-verts}\text{-def Abs-IGraph-inverse*)

lemma $set\text{-ig-arcs-verts}$:
assumes $IGraph\text{-inv } G (u,v) \in set (ig\text{-arcs } G)$ **shows** $u \in set (ig\text{-verts } G) v \in set (ig\text{-verts } G)$

using *assms* **unfolding** *IGraph-inv-def*
 by (*auto simp: mkg'-simps dest: all-nth-imp-all-set*)

lemma *IGraph-inv-conv*:

$IGraph\text{-}inv\ G \longleftrightarrow pair\text{-}fin\text{-}digraph\ (mk\text{-}graph'\ G)$

proof –

{ **assume** $\forall e \in set\ (ig\text{-}arcs\ G). fst\ e \in set\ (ig\text{-}verts\ G) \wedge snd\ e \in set\ (ig\text{-}verts\ G)$
then have $pair\text{-}fin\text{-}digraph\ (mk\text{-}graph'\ G)$
by *unfold-locales (auto simp: mkg'-simps)* }

moreover

{ **assume** $pair\text{-}fin\text{-}digraph\ (mk\text{-}graph'\ G)$
then interpret $pair\text{-}fin\text{-}digraph\ mk\text{-}graph'\ G$.
have $\forall e \in set\ (ig\text{-}arcs\ G). fst\ e \in set\ (ig\text{-}verts\ G) \wedge snd\ e \in set\ (ig\text{-}verts\ G)$
using *tail-in-verts head-in-verts*
by (*fastforce simp: mkg'-simps in-set-conv-nth*) }

ultimately

show *?thesis unfolding IGraph-inv-def by blast*

qed

lemma *IGraph-inv-conv'*:

$IGraph\text{-}inv\ G \longleftrightarrow pair\text{-}pseudo\text{-}graph\ (mk\text{-}graph\ G)$

unfolding *IGraph-inv-conv*

proof

assume $pair\text{-}fin\text{-}digraph\ (mk\text{-}graph'\ G)$

interpret *ppd: pair-fin-digraph mk-graph' G by fact*

interpret *pd: pair-fin-digraph mk-graph G*

unfolding *mk-graph-def ..*

show $pair\text{-}pseudo\text{-}graph\ (mk\text{-}graph\ G)$

by *unfold-locales (auto simp: mk-graph-def symmetric-mk-symmetric)*

next

assume *A: pair-pseudo-graph (mk-graph G)*

interpret *ppg: pair-pseudo-graph mk-graph G by fact*

show $pair\text{-}fin\text{-}digraph\ (mk\text{-}graph'\ G)$

using *ppg.wellformed'*

by *unfold-locales (auto simp: mkg-simps mkg'-simps symcl-def, auto)*

qed

lemma *iadj-io-edge*:

assumes $u \in set\ (ig\text{-}verts\ G) \ e \in set\ (ig\text{-}in\text{-}out\text{-}arcs\ G\ u)$

shows $iadj\ G\ u\ (ig\text{-}opposite\ G\ e\ u)$

proof –

from *assms* **obtain** *v* **where** $e = (u, v) \vee e = (v, u) \ e \in set\ (ig\text{-}arcs\ G)$

unfolding *ig-in-out-arcs-def* **by** (*cases e*) *auto*

then have $*$: $ig\text{-}opposite\ G\ e\ u = v$ **by** *safe (auto simp: ig-opposite-def)*

show *?thesis using e unfolding iadj-def * by auto*

qed

lemma *All-set-ig-verts*: $(\forall v \in set\ (ig\text{-}verts\ G). P\ v) \longleftrightarrow (\forall i < ig\text{-}verts\text{-}cnt\ G. P)$

(*ig-verts* G ! i)

by (*metis in-set-conv-nth ig-verts-cnt-def*)

lemma *IGraph-imp-ppd-mkg'*:

assumes *IGraph-inv* G **shows** *pair-fin-digraph* (*mk-graph'* G)

using *assms unfolding IGraph-inv-conv* **by** *auto*

lemma *finite-symcl-iff*: *finite* (R^s) \longleftrightarrow *finite* R

unfolding *symcl-def* **by** *blast*

lemma (**in** *pair-fin-digraph*) *pair-pseudo-graphI-mk-symmetric*:

pair-pseudo-graph (*mk-symmetric* G)

by *unfold-locales*

(*auto simp: parcs-mk-symmetric symmetric-mk-symmetric wellformed'*)

lemma *IGraph-imp-ppg-mkg*:

assumes *IGraph-inv* G **shows** *pair-pseudo-graph* (*mk-graph* G)

using *assms unfolding mk-graph-def*

by (*intro pair-fin-digraph.pair-pseudo-graphI-mk-symmetric IGraph-imp-ppd-mkg'*)

lemma *IGraph-lf-imp-pg-mkg*:

assumes *IGraph-inv* G *loop-free* (*mk-graph* G) **shows** *pair-graph* (*mk-graph* G)

proof –

interpret *ppg*: *pair-pseudo-graph* *mk-graph* G

using *assms(1)* **by** (*rule IGraph-imp-ppg-mkg*)

show *pair-graph* (*mk-graph* G)

using *assms* **by** *unfold-locales (auto simp: loop-free-def)*

qed

lemma *set-ig-arcs-imp-verts*:

assumes $(u,v) \in \text{set } (ig\text{-arcs } G)$ *IGraph-inv* G **shows** $u \in \text{set } (ig\text{-verts } G)$ $v \in \text{set } (ig\text{-verts } G)$

proof –

interpret *pair-pseudo-graph* *mk-graph* G

using *assms* **by** (*auto intro: IGraph-imp-ppg-mkg*)

from *assms* **have** $(u,v) \in \text{parcs } (mk\text{-graph } G)$ **by** (*simp add: mkg-simps symcl-def*)

then have $u \in \text{pverts } (mk\text{-graph } G)$ $v \in \text{pverts } (mk\text{-graph } G)$ **by** (*auto dest: wellformed'*)

then show $u \in \text{set } (ig\text{-verts } G)$ $v \in \text{set } (ig\text{-verts } G)$ **by** (*auto simp: mkg-simps*)

qed

lemma *iadj-imp-verts*:

assumes *iadj* G u v *IGraph-inv* G **shows** $u \in \text{set } (ig\text{-verts } G)$ $v \in \text{set } (ig\text{-verts } G)$

using *assms unfolding iadj-def* **by** (*auto dest: set-ig-arcs-imp-verts*)

lemma *card-ig-neighbors-indegree*:

assumes *IGraph-inv* G

shows $\text{card } (ig\text{-neighbors } G$ $u) = \text{in-degree } (mk\text{-graph } G)$ u

proof –

have $inj2$: $inj\text{-}on$ $(\lambda e. ig\text{-}opposite\ G\ e\ u)$ $\{e \in parcs\ (mk\text{-}graph\ G). snd\ e = u\}$
unfolding $ig\text{-}opposite\text{-}def$ **by** $(rule\ inj\text{-}onI)$ $(fastforce\ split: if\text{-}split\text{-}asm)$

have $ig\text{-}neighbors\ G\ u = (\lambda e. ig\text{-}opposite\ G\ e\ u)$ ‘ $\{e \in parcs\ (mk\text{-}graph\ G). snd\ e = u\}$

using $assms$ **unfolding** $ig\text{-}neighbors\text{-}def$

by $(auto\ simp: ig\text{-}opposite\text{-}simps\ symcl\text{-}def\ mkg\text{-}simps\ set\text{-}ig\text{-}arcs\text{-}verts\ intro!:$
 $rev\text{-}image\text{-}eqI)$

then have $card\ (ig\text{-}neighbors\ G\ u) = card\ ((\lambda e. ig\text{-}opposite\ G\ e\ u)$ ‘ $\{e \in parcs$
 $(mk\text{-}graph\ G). snd\ e = u\}$)

by $simp$

also have $\dots = in\text{-}degree\ (mk\text{-}graph\ G)\ u$

unfolding $in\text{-}degree\text{-}def\ in\text{-}arcs\text{-}def\ with\text{-}proj\text{-}simps$

using $inj2$ **by** $(rule\ card\text{-}image)$

finally show $?thesis$.

qed

lemma $iadjD$:

assumes $iadj\ G\ u\ v$

shows $\exists e \in set\ (ig\text{-}in\text{-}out\text{-}arcs\ G\ u). (e = (u,v) \vee e = (v,u))$

proof –

from $assms$ **obtain** e **where** $e \in set\ (ig\text{-}arcs\ G)$ $e = (u,v) \vee e = (v,u)$

unfolding $iadj\text{-}def$ **by** $auto$

then show $?thesis$ **unfolding** $ig\text{-}in\text{-}out\text{-}arcs\text{-}def$ **by** $auto$

qed

lemma

$ig\text{-}verts\text{-}empty[simp]$: $ig\text{-}verts\ ig\text{-}empty = []$ **and**

$ig\text{-}verts\text{-}add\text{-}e[simp]$: $ig\text{-}verts\ (ig\text{-}add\text{-}e\ G\ u\ v) = ig\text{-}verts\ G$ **and**

$ig\text{-}verts\text{-}add\text{-}v[simp]$: $ig\text{-}verts\ (ig\text{-}add\text{-}v\ G\ v) = ig\text{-}verts\ G$ @ $(if\ v \in set\ (ig\text{-}verts$
 $G)$ then $[]$ else $[v])$)

unfolding $ig\text{-}verts\text{-}def\ ig\text{-}empty\text{-}def\ ig\text{-}add\text{-}e\text{-}def\ ig\text{-}add\text{-}v\text{-}def$

by $(auto\ simp: Abs\text{-}IGraph\text{-}inverse\ distinct\text{-}ig\text{-}verts[simplified\ ig\text{-}verts\text{-}def])$

lemma

$ig\text{-}arcs\text{-}empty[simp]$: $ig\text{-}arcs\ ig\text{-}empty = []$ **and**

$ig\text{-}arcs\text{-}add\text{-}e[simp]$: $ig\text{-}arcs\ (ig\text{-}add\text{-}e\ G\ u\ v) = ig\text{-}arcs\ G$ @ $[(u,v)]$ **and**

$ig\text{-}arcs\text{-}add\text{-}v[simp]$: $ig\text{-}arcs\ (ig\text{-}add\text{-}v\ G\ v) = ig\text{-}arcs\ G$

unfolding $ig\text{-}arcs\text{-}def\ ig\text{-}empty\text{-}def\ ig\text{-}add\text{-}e\text{-}def\ ig\text{-}add\text{-}v\text{-}def$

by $(auto\ simp: Abs\text{-}IGraph\text{-}inverse\ distinct\text{-}ig\text{-}verts)$

16.2 Total Correctness

16.2.1 Procedure $is\text{-}subgraph$

definition $is\text{-}subgraph\text{-}verts\text{-}inv :: IGraph \Rightarrow IGraph \Rightarrow nat \Rightarrow bool$ **where**

$is\text{-}subgraph\text{-}verts\text{-}inv\ G\ H\ i \equiv set\ (take\ i\ (ig\text{-}verts\ G)) \subseteq set\ (ig\text{-}verts\ H)$

definition $is\text{-}subgraph\text{-}arcs\text{-}inv :: IGraph \Rightarrow IGraph \Rightarrow nat \Rightarrow bool$ **where**

is-subgraph-arcs-inv $G H i \equiv \forall j < i. \text{let } (u,v) = \text{ig-arcs } G ! j \text{ in}$
 $((u,v) \in \text{set } (\text{ig-arcs } H) \vee (v,u) \in \text{set } (\text{ig-arcs } H))$
 $\wedge u \in \text{set } (\text{ig-verts } G) \wedge v \in \text{set } (\text{ig-verts } G)$

lemma *is-subgraph-verts-0*: *is-subgraph-verts-inv* $G H 0$
unfolding *is-subgraph-verts-inv-def* **by** *auto*

lemma *is-subgraph-verts-step*:
assumes *is-subgraph-verts-inv* $G H i$ *ig-verts* $G ! i \in \text{set } (\text{ig-verts } H)$
assumes $i < \text{length } (\text{ig-verts } G)$
shows *is-subgraph-verts-inv* $G H (\text{Suc } i)$
using *assms* **by** (*auto simp: is-subgraph-verts-inv-def take-Suc-conv-app-nth*)

lemma *is-subgraph-verts-last*:
is-subgraph-verts-inv $G H (\text{length } (\text{ig-verts } G)) \longleftrightarrow \text{pverts } (\text{mk-graph } G) \subseteq \text{pverts}$
 $(\text{mk-graph } H)$
apply (*auto simp: is-subgraph-verts-inv-def mkg-simps*)
done

lemma *is-subgraph-arcs-0*: *is-subgraph-arcs-inv* $G H 0$
unfolding *is-subgraph-arcs-inv-def* **by** *auto*

lemma *is-subgraph-arcs-step*:
assumes *is-subgraph-arcs-inv* $G H i$
 $e \in \text{set } (\text{ig-arcs } H) \vee (\text{snd } e, \text{fst } e) \in \text{set } (\text{ig-arcs } H)$
 $\text{fst } e \in \text{set } (\text{ig-verts } G) \text{ snd } e \in \text{set } (\text{ig-verts } G)$
assumes $e = \text{ig-arcs } G ! i$
assumes $i < \text{length } (\text{ig-arcs } G)$
shows *is-subgraph-arcs-inv* $G H (\text{Suc } i)$
using *assms* **by** (*auto simp: is-subgraph-arcs-inv-def less-Suc-eq*)

lemma *wellformed-pseudo-graph-mkg*:
shows *pair-wf-digraph* $(\text{mk-graph } G) = \text{pair-pseudo-graph}(\text{mk-graph } G)$ (**is** $?L =$
 $?R$)

proof
assume $?R$
then interpret *ppg*: *pair-pseudo-graph* $\text{mk-graph } G$.
show $?L$ **by** *unfold-locales*

next
assume $?L$
moreover have *symmetric* $(\text{mk-graph } G)$
unfolding *mk-graph-def* **by** (*simp add: symmetric-mk-symmetric*)
ultimately show $?R$
unfolding *pair-wf-digraph-def*
by *unfold-locales (auto simp: mkg-simps finite-symcl-iff)*

qed

lemma *is-subgraph-arcs-last*:
is-subgraph-arcs-inv $G H (\text{length } (\text{ig-arcs } G)) \longleftrightarrow \text{parcs } (\text{mk-graph } G) \subseteq \text{parcs}$

$(mk\text{-graph } H) \wedge pair\text{-pseudo-graph } (mk\text{-graph } G)$
proof –
have $is\text{-subgraph-arcs-inv } G H (length (ig\text{-arcs } G))$
 $= (\forall (u,v) \in set (ig\text{-arcs } G). ((u,v) \in set (ig\text{-arcs } H) \vee (v,u) \in set (ig\text{-arcs } H)))$
 $\wedge u \in set (ig\text{-verts } G) \wedge v \in set (ig\text{-verts } G))$
unfolding $is\text{-subgraph-arcs-inv-def}$
by $(metis (lifting, no-types) all\text{-nth-imp-all-set nth-mem})$
also have $\dots \longleftrightarrow parcs (mk\text{-graph } G) \subseteq parcs (mk\text{-graph } H) \wedge pair\text{-pseudo-graph } (mk\text{-graph } G)$
unfolding $wellformed\text{-pseudo-graph-mkg[symmetric]}$
by $(auto simp: mkg\text{-simps pair-wf-digraph-def symcl-def})$
finally show $?thesis$.
qed

lemma $is\text{-subgraph-verts-arcs-last}$:
assumes $is\text{-subgraph-verts-inv } G H (ig\text{-verts-cnt } G)$
assumes $is\text{-subgraph-arcs-inv } G H (ig\text{-arcs-cnt } G)$
assumes $I\text{Graph-inv } H$
shows $subgraph (mk\text{-graph } G) (mk\text{-graph } H) (\text{is } ?T1)$
 $pair\text{-pseudo-graph } (mk\text{-graph } G) (\text{is } ?T2)$
proof –
interpret $ppg: pair\text{-pseudo-graph } mk\text{-graph } G$
using $assms$ **by** $(simp add: is\text{-subgraph-arcs-last})$
interpret $ppgH: pair\text{-pseudo-graph } mk\text{-graph } H$ **using** $assms$ **by** $(intro I\text{Graph-imp-ppg-mkg})$
have $wf\text{-digraph } (with\text{-proj } (mk\text{-graph } G))$ **by** $unfold\text{-locales}$
with $assms$ **show** $?T1 ?T2$
by $(auto simp: is\text{-subgraph-verts-last is\text{-subgraph-arcs-last subgraph-def ppgH.wf-digraph})$
qed

lemma $is\text{-subgraph-false}$:
assumes $subgraph (mk\text{-graph } G) (mk\text{-graph } H)$
obtains $\forall i < length (ig\text{-verts } G). ig\text{-verts } G ! i \in set (ig\text{-verts } H)$
 $\forall i < length (ig\text{-arcs } G). let (u,v) = ig\text{-arcs } G ! i in$
 $((u,v) \in set (ig\text{-arcs } H) \vee (v,u) \in set (ig\text{-arcs } H))$
 $\wedge u \in set (ig\text{-verts } G) \wedge v \in set (ig\text{-verts } G)$
proof
from $assms$
show $\forall i < length (ig\text{-verts } G). ig\text{-verts } G ! i \in set (ig\text{-verts } H)$
unfolding $subgraph-def$ **by** $(auto simp: mkg\text{-simps})$
next
from $assms$ **have** $is\text{-subgraph-arcs-inv } G H (length (ig\text{-arcs } G))$
unfolding $is\text{-subgraph-arcs-last subgraph-def wellformed\text{-pseudo-graph-mkg[symmetric]}$
by $(auto simp: wf\text{-digraph-wp-iff})$
then show $\forall i < length (ig\text{-arcs } G). let (u,v) = ig\text{-arcs } G ! i in$
 $((u,v) \in set (ig\text{-arcs } H) \vee (v,u) \in set (ig\text{-arcs } H))$
 $\wedge u \in set (ig\text{-verts } G) \wedge v \in set (ig\text{-verts } G)$
by $(auto simp: is\text{-subgraph-arcs-inv-def})$
qed

```

lemma (in is-subgraph-impl) is-subgraph-spec:
   $\forall \sigma. \Gamma \vdash_t \{ \sigma. IGraph\text{-inv } 'H \} 'R := PROC \text{ is-subgraph}('G, 'H) \{ \{ 'G = \sigma G$ 
 $\wedge 'H = \sigma H \wedge 'R = (\text{subgraph } (\text{mk-graph } 'G) (\text{mk-graph } 'H) \wedge IGraph\text{-inv } 'G) \}$ 
  apply (vcg-step spec=none)
  apply (rewrite
    at whileAnno - (named-loop "verts") - -
    in for ( $\sigma$ )
    to whileAnno -
     $\{ \text{is-subgraph-verts-inv } 'G 'H 'i \wedge 'G = \sigma G \wedge 'H = \sigma H \wedge 'i \leq \text{ig-verts-cnt}$ 
 $'G$ 
     $\wedge IGraph\text{-inv } 'H \}$ 
    (MEASURE ig-verts-cnt 'G - 'i)
    -
    annotate-named-loop-var)
  apply (rewrite
    at whileAnno - (named-loop "arcs") - -
    in for ( $\sigma$ )
    to whileAnno -
     $\{ \text{is-subgraph-arcs-inv } 'G 'H 'i \wedge 'G = \sigma G \wedge 'H = \sigma H \wedge 'i \leq \text{ig-arcs-cnt}$ 
 $'G$ 
     $\wedge \text{is-subgraph-verts-inv } 'G 'H (\text{length } (\text{ig-verts } 'G)) \wedge IGraph\text{-inv } 'H \}$ 
    (MEASURE ig-arcs-cnt 'G - 'i)
    -
    annotate-named-loop-var)
  apply vcg
  apply (fastforce simp: is-subgraph-verts-0)
  apply (fastforce simp: is-subgraph-verts-step elim: is-subgraph-false)
  apply (fastforce simp: is-subgraph-arcs-0 not-less)
  apply (auto simp: is-subgraph-arcs-step elim!: is-subgraph-false; fastforce)
  apply (fastforce simp: IGraph-inv-conv' is-subgraph-verts-arcs-last)
  done

```

16.2.2 Procedure *is-loop-free*

definition *is-loopfree-inv* $G k \equiv \forall j < k. \text{fst } (\text{ig-arcs } G ! j) \neq \text{snd } (\text{ig-arcs } G ! j)$

lemma *is-loopfree-0*:
is-loopfree-inv $G 0$
 by (auto simp: is-loopfree-inv-def)

lemma *is-loopfree-step1*:
assumes *is-loopfree-inv* $G n$
assumes $\text{fst } (\text{ig-arcs } G ! n) \neq \text{snd } (\text{ig-arcs } G ! n)$
assumes $n < \text{ig-arcs-cnt } G$
shows *is-loopfree-inv* $G (\text{Suc } n)$
using *assms* **unfolding** *is-loopfree-inv-def*
by (auto intro: less-SucI elim: less-SucE)

```

lemma is-loopfree-step2:
  assumes loop-free (mk-graph G)
  assumes  $n < \text{ig-arcs-cnt } G$ 
  shows  $\text{fst } (\text{ig-arcs } G ! n) \neq \text{snd } (\text{ig-arcs } G ! n)$ 
  using assms unfolding is-loopfree-inv-def loop-free-def
  by (auto simp: mkg-simps symcl-def)

lemma is-loopfree-last:
  assumes is-loopfree-inv G (ig-arcs-cnt G)
  shows loop-free (mk-graph G)
  using assms apply (auto simp: is-loopfree-inv-def loop-free-def mkg-simps in-set-conv-nth
symcl-def)
  apply (metis fst-eqD snd-eqD)+
  done

lemma (in is-loopfree-impl) is-loopfree-spec:
   $\forall \sigma. \Gamma \vdash_t \{ \sigma. \text{IGraph-inv } 'G \} 'R ::= \text{PROC } \text{is-loopfree}('G) \{ 'G = {}^\sigma G \wedge 'R$ 
   $= \text{loop-free } (\text{mk-graph } 'G) \}$ 
  apply (vcg-step spec=none)
  apply (rewrite
    at whileAnno - (named-loop "loop") - -
    in for ( $\sigma$ )
    to whileAnno -
     $\{ \text{is-loopfree-inv } 'G 'i \wedge 'G = {}^\sigma G \wedge 'i \leq \text{ig-arcs-cnt } 'G \}$ 
    (MEASURE ig-arcs-cnt 'G - 'i)
    -
    annotate-named-loop-var)
  apply vcg
  apply (fastforce simp: is-loopfree-0)
  apply (fastforce intro: is-loopfree-step1 dest: is-loopfree-step2)
  apply (fastforce simp: is-loopfree-last)
  done

```

16.2.3 Procedure *select-nodes*

definition *select-nodes-inv* :: *IGraph* \Rightarrow *IGraph* \Rightarrow *nat* \Rightarrow *bool* **where**
select-nodes-inv *G H i* $\equiv \text{set } (\text{ig-verts } H) = \{v \in \text{set } (\text{take } i (\text{ig-verts } G)). \text{card}$
 $(\text{ig-neighbors } G v) \geq 3\} \wedge \text{IGraph-inv } H$

```

lemma select-nodes-inv-step:
  fixes G H i
  defines  $v \equiv \text{ig-verts } G ! i$ 
  assumes G-inv: IGraph-inv G
  assumes sni-inv: select-nodes-inv G H i
  assumes less:  $i < \text{ig-verts-cnt } G$ 
  assumes H':  $H' = (\text{if } 3 \leq \text{card } (\text{ig-neighbors } G v) \text{ then } \text{ig-add-v } H v \text{ else } H)$ 
  shows select-nodes-inv G H' (Suc i)
proof -
  have *: IGraph-inv H' using sni-inv H'

```

unfolding *IGraph-inv-def select-nodes-inv-def* **by** *auto*
have *take-Suc-i*: $\text{take } (Suc\ i) \ (ig\text{-verts } G) = \text{take } i \ (ig\text{-verts } G) \ @ \ [v]$
using *less unfolding v-def* **by** *(auto simp: take-Suc-conv-app-nth)*
have *X*: $v \notin \text{set } (\text{take } i \ (ig\text{-verts } G))$
using *G-inv less distinct-ig-verts* **unfolding** *v-def IGraph-inv-conv*
by *(auto simp: distinct-conv-nth in-set-conv-nth)*

show *?thesis*
unfolding *select-nodes-inv-def* **using** *X sni-inv*
by *(simp only: *) (auto simp: take-Suc-i select-nodes-inv-def H')*
qed

definition *select-nodes-prop* :: $IGraph \Rightarrow IGraph \Rightarrow \text{bool}$ **where**
select-nodes-prop $G\ H \equiv \text{pverts } (\text{mk-graph } H) = \text{verts3 } (\text{mk-graph } G)$

lemma *(in select-nodes-impl) select-nodes-spec*:

$\forall \sigma. \Gamma \vdash_t \{ \sigma. IGraph\text{-inv } 'G \} \ 'R ::= \text{PROC } \text{select-nodes}('G)$
 $\{ \text{select-nodes-prop } ^\sigma G \ 'R \wedge IGraph\text{-inv } 'R \wedge \text{set } (ig\text{-arcs } 'R) = \{ \} \}$
apply *vcg-step*
apply *(rewrite*
at whileAnno - (named-loop "loop") - -
in for *(σ)*
to whileAnno -
 $\{ \text{select-nodes-inv } 'G \ 'R \ 'i \wedge \ 'i \leq \text{ig-verts-cnt } 'G \wedge \ 'G = ^\sigma G \wedge IGraph\text{-inv}$
 $\ 'G \wedge \text{set } (ig\text{-arcs } 'R) = \{ \} \}$
 $(\text{MEASURE } \text{ig-verts-cnt } 'G - \ 'i)$
 $-$
annotate-named-loop-var)
apply *vcg*
apply *(fastforce simp: select-nodes-inv-def IGraph-inv-def mkg'-simps)*
apply *(fastforce simp add: select-nodes-inv-step)*
apply *(fastforce simp add: select-nodes-inv-def select-nodes-prop-def card-ig-neighbors-indegree*
verts3-def mkg-simps)
done

16.2.4 Procedure *find-endpoint*

definition *find-endpoint-path-inv* **where**

find-endpoint-path-inv $G\ H\ \text{len } u\ v\ w\ x \equiv$
 $\exists p. \text{pre-digraph.awalk } (\text{mk-graph } G) \ u\ p\ x \wedge \text{length } p = \text{len} \wedge$
 $\text{hd } p = (u, v) \wedge \text{last } p = (w, x) \wedge$
 $\text{set } (\text{pre-digraph.inner-verts } (\text{mk-graph } G) \ p) \cap \text{set } (ig\text{-verts } H) = \{ \} \wedge$
progressing p

definition *find-endpoint-arcs-inv* **where**

find-endpoint-arcs-inv $G\ \text{found } k\ v0\ v1\ v0'\ v1' \equiv$
 $(\text{found} \longrightarrow (\exists i < k. v1' = \text{ig-opposite } G \ (ig\text{-in-out-arcs } G \ v1 \ ! \ i) \ v1 \wedge v0' =$
 $v1 \wedge v0 \neq v1')) \wedge$
 $(\neg \text{found} \longrightarrow (\forall i < k. v0 = \text{ig-opposite } G \ (ig\text{-in-out-arcs } G \ v1 \ ! \ i) \ v1) \wedge v0 =$

$v0' \wedge v1 = v1'$)

lemma *find-endpoint-path-first*:

assumes *iadj* G u v $u \neq v$ *IGraph-inv* G

shows *find-endpoint-path-inv* G H (*Suc* 0) u v u v

proof –

interpret *ppg*: *pair-pseudo-graph mk-graph* G

using *assms* **by** (*auto intro*: *IGraph-imp-ppg-mkg*)

have $(u,v) \in$ *parcs* (*mk-graph* G)

using *assms* **by** (*auto simp*: *iadj-def mkg-simps symcl-def*)

then have *ppg.awalk* u $[(u,v)]$ v *length* $[(u,v)] =$ *Suc* 0 *hd* $[(u,v)] = (u,v)$ *last* $[(u,v)] = (u,v)$ *progressing* $[(u,v)]$

using *assms* **by** (*auto simp*: *ppg.awalk-simps iadj-imp-verts mkg-simps progressing-Cons*)

moreover

have *set* (*ppg.inner-verts* $[(u,v)]$) \cap *set* (*ig-verts* H) = $\{\}$

by (*auto simp*: *ppg.inner-verts-def*)

ultimately

show *?thesis unfolding find-endpoint-path-inv-def* **by** *blast*

qed

lemma *find-endpoint-arcs-0*:

find-endpoint-arcs-inv G *False* 0 $v0$ $v1$ $v0$ $v1$

unfolding *find-endpoint-arcs-inv-def* **by** *auto*

lemma *find-endpoint-path-lastE*:

assumes *find-endpoint-path-inv* G H *len* u v w x

assumes *ig*: *IGraph-inv* G **and** *lf*: *loop-free* (*mk-graph* G)

assumes *snp*: *select-nodes-prop* G H

assumes 0 < *len*

assumes *u*: $u \in$ *set* (*ig-verts* H)

obtains *p* **where** *pre-digraph.awalk* (*mk-graph* G) u $((u,v) \# p)$ x

and *progressing* $((u,v) \# p)$

and *set* (*pre-digraph.inner-verts* (*mk-graph* G) $((u,v) \# p)$) \cap *set* (*ig-verts* H)

= $\{\}$

and *len* \leq *ig-verts-cnt* G

proof –

from *ig* **and** *lf* **interpret** *pair-graph mk-graph* G

by (*rule* *IGraph-lf-imp-pg-mkg*)

have [*simp*]: *verts3* (*mk-graph* G) = *set* (*ig-verts* H)

using *assms* **unfolding** *select-nodes-prop-def* **by** (*auto simp*: *mkg-simps*)

from *assms* **obtain** *q* **where** *awalk* u q x *length* $q =$ *len* *hd* $q = (u,v)$

and *iv*: *set* (*inner-verts* q) \cap *verts3* (*mk-graph* G) = $\{\}$

and *prg*: *progressing* q

unfolding *find-endpoint-path-inv-def* **by** *auto*

moreover then obtain $q0$ qs **where** $q = q0 \# qs$ **using** $\langle 0 < len \rangle$ **by** (*cases* q) *auto*

moreover

have *len* \leq *ig-verts-cnt* G

proof –
have *ev-q*: $awalk\text{-}verts\ u\ q = u \# inner\text{-}verts\ q @ [x]$
unfolding *inner-verts-conv*[of *q u*] **using** $q \langle q = q0 \# qs \rangle$ **by** *auto*
then have *len-ev*: $length\ (awalk\text{-}verts\ u\ q) = 2 + length\ (inner\text{-}verts\ q)$
by *auto*

have *set-av*: $set\ (awalk\text{-}verts\ u\ q) \subseteq pverts\ (mk\text{-}graph\ G)$
using *q(1)* **by** *auto*

from *snp u* **have** $u \in verts3\ (mk\text{-}graph\ G)$ **by** *simp*
moreover
with - - **have** *distinct* (*inner-verts q*)
using *q(1) iv prg* **by** (*rule verts2-awalk-distinct*) (*auto simp: verts3-def*)
ultimately
have *distinct* ($u \# inner\text{-}verts\ q$) **using** *iv* **by** *auto*
moreover
have $set\ (u \# inner\text{-}verts\ q) \subseteq pverts\ (mk\text{-}graph\ G)$
using *ev-q set-av* **by** *auto*
ultimately
have $length\ (u \# inner\text{-}verts\ q) \leq card\ (pverts\ (mk\text{-}graph\ G))$
by (*metis card-mono distinct-card finite-set verts-mkg*)
then have $length\ (awalk\text{-}verts\ u\ q) \leq 1 + card\ (pverts\ (mk\text{-}graph\ G))$
by (*simp add: len-ev*)
then have $length\ q \leq card\ (pverts\ (mk\text{-}graph\ G))$
by (*auto simp: length-awalk-verts*)
also have $\dots \leq ig\text{-}verts\text{-}cnt\ G$ **by** (*auto simp: mkg-simps card-length*)
finally show *?thesis* **by** (*simp add: q*)
qed
ultimately show *?thesis* **by** (*intro that*) *auto*
qed

lemma *find-endpoint-path-last1*:
assumes *find-endpoint-path-inv G H len u v w x*
assumes *ig: IGraph-inv G and lf: loop-free (mk-graph G)*
assumes *snp: select-nodes-prop G H*
assumes $0 < len$
assumes *mem*: $u \in set\ (ig\text{-}verts\ H)\ x \in set\ (ig\text{-}verts\ H)\ u \neq x$
shows $\exists p. pre\text{-}digraph.\text{iapath}\ (mk\text{-}graph\ G)\ u\ ((u,v) \# p)\ x$
proof –
from *ig and lf* **interpret** *pair-graph mk-graph G*
by (*rule IGraph-lf-imp-pg-mkg*)
have [*simp*]: $verts3\ (mk\text{-}graph\ G) = set\ (ig\text{-}verts\ H)$
 $\bigwedge x. x \in set\ (ig\text{-}verts\ H) \implies x \in pverts\ (mk\text{-}graph\ G)$
using *assms* **unfolding** *select-nodes-prop-def* **by** (*auto simp: mkg-simps verts3-def*)
show *?thesis*
apply (*rule find-endpoint-path-lastE[OF assms(1–5) mem(1)]*)
by (*drule deg2-awalk-is-iapath[rotated 2]*) (*auto simp: mem*)
qed

lemma *find-endpoint-path-last2D*:
assumes *path*: *find-endpoint-path-inv* G H *len* u v w u
assumes *ig*: *IGraph-inv* G **and** *lf*: *loop-free* (*mk-graph* G)
assumes *snp*: *select-nodes-prop* G H
assumes $0 < \text{len}$
assumes *mem*: $u \in \text{set } (\text{ig-verts } H)$
assumes *iapath*: *pre-digraph.iapath* (*mk-graph* G) u $((u,v) \# p)$ x
shows *False*
proof –
from *ig* **and** *lf* **interpret** *pair-graph* *mk-graph* G
by (*rule* *IGraph-lf-imp-pg-mkg*)
have [*simp*]: $\text{verts3 } (\text{mk-graph } G) = \text{set } (\text{ig-verts } H)$
using *assms* **unfolding** *select-nodes-prop-def* **by** (*auto simp: mkg-simps*)
have V : $\text{verts3 } (\text{mk-graph } G) \subseteq \text{verts3 } (\text{mk-graph } G) \text{verts3 } (\text{mk-graph } G) \subseteq$
pverts (*mk-graph* G)
using *verts3-in-verts* [**where** $G = \text{mk-graph } G$] **by** *auto*

obtain q **where** *walk-q*: *awalk* u $((u, v) \# q)$ u **and**
progress-q: *progressing* $((u, v) \# q)$ **and**
iv-q: $\text{set } (\text{inner-verts } ((u, v) \# q)) \cap \text{verts3 } (\text{mk-graph } G) = \{\}$
by (*rule* *find-endpoint-path-lastE* [*OF* *path ig lf snp <0 < len> mem*]) *auto*

from *iapath* **have** *walk-p*: *awalk* u $((u,v) \# p)$ x **and**
iv-p: $\text{set } (\text{inner-verts } ((u, v) \# p)) \cap \text{verts3 } (\text{mk-graph } G) = \{\}$ **and**
uv-verts3: $u \in \text{verts3 } (\text{mk-graph } G) \ x \in \text{verts3 } (\text{mk-graph } G)$
unfolding *gen-iapath-def* *apath-def* **by** *auto*
from *iapath* **have** *progress-p*: *progressing* $((u,v) \# p)$
unfolding *gen-iapath-def* **by** (*auto intro: apath-imp-progressing*)

from V *walk-q* *walk-p* *progress-q* *progress-p* *iv-q* *iv-p*
have $(u,v) \# q = (u,v) \# p$
apply (*rule* *same-awalk-by-common-arc* [**where** $e = (u,v)$])
using *uv-verts3*
apply *auto*
done
then show *False*
by (*metis* *iapath* *apath-nonempty-ends* *gen-iapath-def* *awalk-nonempty-ends*(2)
walk-p *walk-q*)
qed

lemma *find-endpoint-arcs-last*:
assumes *arcs*: *find-endpoint-arcs-inv* G *False* (*length* (*ig-in-out-arcs* G $v1$)) $v0$
 $v1$ $v0a$ $v1a$
assumes *path*: *find-endpoint-path-inv* G H *len* $v\text{-tail}$ $v\text{-next}$ $v0$ $v1$
assumes *ig*: *IGraph-inv* G **and** *lf*: *loop-free* (*mk-graph* G)
assumes *snp*: *select-nodes-prop* G H
assumes *mem*: $v\text{-tail} \in \text{set } (\text{ig-verts } H)$
assumes $0 < \text{len}$
shows $\neg \text{pre-digraph.iapath } (\text{mk-graph } G) \ v\text{-tail } ((v\text{-tail}, v\text{-next}) \# p) \ x$

proof

let $\neg ?A = ?thesis$
assume $?A$

interpret *pair-graph mk-graph G using ig lf by (rule IGraph-lf-imp-pg-mkg)*

have $v3G\text{-eq}: \text{verts3} (\text{mk-graph } G) = \text{set} (\text{ig-verts } H)$
using *assms unfolding select-nodes-prop-def by (auto simp: mkg-simps)*

If no extending edge was found (as implied by *find-endpoint-arcs-inv G False (length (ig-in-out-arcs G v1)) v0 v1 v0a v1a*), the last vertex of the walk computed (as implied by *find-endpoint-path-inv G H len v-tail v-next v0 v1*) is of degree 1. Hence we consider all vertices except the degree-2 nodes.

define V **where** $V = \{v \in \text{pverts} (\text{mk-graph } G). \text{in-degree} (\text{mk-graph } G) v \neq 2\}$

have $V: \text{verts3} (\text{mk-graph } G) \subseteq V \ V \subseteq \text{pverts} (\text{mk-graph } G)$
unfolding *verts3-def V-def by auto*

from $\langle ?A \rangle$ **have** *walk-p: awalk v-tail ((v-tail, v-next) # p) x and*
progress-p: progressing ((v-tail, v-next) # p)
by *(auto simp: gen-iapath-def apath-def intro: apath-imp-progressing)*
have *iapath-V-p: gen-iapath V v-tail ((v-tail, v-next) # p) x*

proof –

{ fix u **assume** $A: u \in \text{set} (\text{inner-verts} ((v\text{-tail}, v\text{-next}) \# p))$
then have $u \in \text{pverts} (\text{mk-graph } G)$ **using** $\langle ?A \rangle$

by *(auto 2 4 simp: set-inner-verts gen-iapath-def apath-Cons-iff dest: awalkI-apath)*

with $A \langle ?A \rangle$ *inner-verts-min-degree[OF walk-p progress-p A]* **have** $u \notin V$
unfolding *gen-iapath-def verts3-def V-def by auto }*

with $\langle ?A \rangle$ V **show** $?thesis$ **by** *(auto simp: gen-iapath-def)*

qed

have *arcs-p: (v-tail, v-next) ∈ set ((v-tail, v-next) # p)*
unfolding *gen-iapath-def apath-def by auto*

have *id-x: 2 < in-degree (mk-graph G) x*
using $\langle ?A \rangle$ **unfolding** *gen-iapath-def verts3-def by auto*

from *arcs* **have** *edge-no-pr: $\bigwedge e. e \in \text{set} (\text{ig-in-out-arcs } G v1) \implies$*
 $v0 = \text{ig-opposite } G e v1$ **and** $v0 = v0a v1 = v1a$
by *(auto simp: find-endpoint-arcs-inv-def in-set-conv-nth)*

have $\{e \in \text{parcs} (\text{mk-graph } G). \text{snd } e = v1\} \subseteq \{(v0, v1)\}$ **(is** $?L \subseteq ?R$ **)**

proof

fix e **assume** $e \in ?L$

then have $\text{fst } e \neq \text{snd } e$ **by** *(auto dest: no-loops)*

moreover

from $\langle e \in ?L \rangle$ **have** $e \in \text{set} (\text{ig-in-out-arcs } G v1) \vee (\text{snd } e, \text{fst } e) \in \text{set}$

```

(ig-in-out-arcs G v1)
  by (auto simp: mkg-simps ig-in-out-arcs-def symcl-def)
  then have v0 = ig-opposite G e v1  $\vee$  v0 = ig-opposite G (snd e, fst e) v1
  by (auto intro: edge-no-pr)
  ultimately show e  $\in$  ?R using  $\langle e \in ?L \rangle$  by (auto simp: ig-opposite-def)
qed
then have id-v1: in-degree (mk-graph G) v1  $\leq$  card {(v0,v1)}
  unfolding in-degree-def in-arcs-def by (intro card-mono) auto

from path obtain q where walk-q: awalk v-tail q v1 and
  q-props: length q = len hd q = (v-tail, v-next) and
  iv-q': set (inner-verts q)  $\cap$  verts3 (mk-graph G) = {} and
  progress-q: progressing q
  by (auto simp: find-endpoint-path-inv-def v3G-eq)
then have v1  $\in$  pverts (mk-graph G)
  by (metis awalk-last-in-verts)
then have v1  $\in$  V using id-v1 unfolding V-def by auto

{ fix u assume A: u  $\in$  set (inner-verts q)
  then have u  $\in$  set (pawalk-verts v-tail q) using walk-q
  by (auto simp: inner-verts-conv[where u=v-tail] awalk-def dest: in-set-butlastD
list-set-tl)
  then have u  $\in$  pverts (mk-graph G) using walk-q by auto
  with A iv-q' inner-verts-min-degree[OF walk-q progress-q A] have u  $\notin$  V
  unfolding verts3-def V-def by auto }
then have iv-q: set (inner-verts q)  $\cap$  V = {} by auto

have arcs-q: (v-tail, v-next)  $\in$  set q
  using q-props  $\langle 0 < len \rangle$  by (cases q) auto

have neq: v-tail  $\neq$  v1
  using find-endpoint-path-last2D[OF - ig lf snp  $\langle 0 < len \rangle$   $\langle v\text{-tail} \in \rightarrow \langle ?A \rangle$ ] path
by auto

have in-V: v-tail  $\in$  V using iapath-V-p unfolding gen-iapath-def by auto
have iapath-V-q: gen-iapath V v-tail q v1
  using V walk-q iv-q progress-q in-V  $\langle v1 \in V \rangle$  neq by (rule deg2-awalk-is-iapath)

have ((v-tail, v-next)  $\#$  p) = q
  using V iapath-V-p iapath-V-q arcs-p arcs-q
  by (rule same-gen-iapath-by-common-arc)
then have v1 = x using walk-p walk-q by auto
then show False using id-v1 id-x by auto
qed

lemma find-endpoint-arcs-step1E:
  assumes find-endpoint-arcs-inv G False k v0 v1 v0' v1'
  assumes ig-opposite G (ig-in-out-arcs G v1 ! k) v1'  $\neq$  v0'
  obtains v0 = v0' v1 = v1' find-endpoint-arcs-inv G True (Suc k) v0 v1 v1

```

(*ig-opposite* G (*ig-in-out-arcs* G $v1$! k) $v1$)
using *assms* **unfolding** *find-endpoint-arcs-inv-def*
by (*auto intro: less-SucI elim: less-SucE*)

lemma *find-endpoint-arcs-step2E*:

assumes *find-endpoint-arcs-inv* G *False* k $v0$ $v1$ $v0'$ $v1'$
assumes *ig-opposite* G (*ig-in-out-arcs* G $v1$! k) $v1' = v0'$
obtains $v0 = v0'$ $v1 = v1'$ *find-endpoint-arcs-inv* G *False* (*Suc* k) $v0$ $v1$ $v0$ $v1$
using *assms* **unfolding** *find-endpoint-arcs-inv-def*
by (*auto intro: less-SucI elim: less-SucE*)

lemma *find-endpoint-path-step*:

assumes *path: find-endpoint-path-inv* G H *len* u v w x **and** $0 < len$
assumes *arcs: find-endpoint-arcs-inv* G *True* k w x w' x'
 $k \leq length$ (*ig-in-out-arcs* G x)
assumes *ig: IGraph-inv* G
assumes *not-end: x* \notin *set* (*ig-verts* H)
shows *find-endpoint-path-inv* G H (*Suc* len) u v w' x'

proof –

interpret *pg: pair-pseudo-graph mk-graph* G
using *ig* **by** (*auto intro: IGraph-imp-ppg-mkg*)
from *path* **obtain** p **where** *awalk: pg.awalk* u p x **and**
 $p: length$ $p = len$ *hd* $p = (u, v)$ *last* $p = (w, x)$ **and**
 $iv: set$ (*pg.inner-verts* p) \cap *set* (*ig-verts* H) = $\{\}$ **and**
progress: progressing p
by (*auto simp: find-endpoint-path-inv-def*)

define p' **where** $p' = p @ [(x, x')]$

from *path* **have** $x \in set$ (*ig-verts* G)
by (*metis awalk pg.awalk-last-in-verts verts-mkg*)

with *arcs* **have** *iadj* G x x' $x = w'$ $w \neq x'$

using $\langle x \in set$ (*ig-verts* G) \rangle **unfolding** *find-endpoint-arcs-inv-def*
by (*auto intro: iadj-io-edge*)

then **have** $(x, x') \in parcs$ (*mk-graph* G) $x' \in set$ (*ig-verts* G)

using *ig* **unfolding** *iadj-def* **by** (*auto simp: mkg-simps set-ig-arcs-imp-verts symcl-def*)

then **have** *pg.awalk* u p' x'

unfolding *p'-def* **using** *awalk* **by** (*auto simp: pg.awalk-simps mkg-simps*)

moreover

have *length* $p' = Suc$ len *hd* $p' = (u, v)$ *last* $p' = (w', x')$

using $\langle x = w' \rangle$ $\langle 0 < len \rangle$ p **by** (*auto simp: p'-def*)

moreover

have *set* (*pg.inner-verts* p') \cap *set* (*ig-verts* H) = $\{\}$

using *iv not-end* p $\langle 0 < len \rangle$ **unfolding** *p'-def* **by** (*auto simp: pg.inner-verts-def*)

moreover

$\{$ **fix** ys y z zs **have** $p' \neq ys @ [(y, z), (z, y)] @ zs$

proof

```

let  $\neg ?A = ?thesis$ 
assume  $?A$ 
from progress have  $\bigwedge zs. p \neq ys @ (y,z) \# (z,y) \# zs$ 
  by (auto simp: progressing-append-iff progressing-Cons)
with  $\langle ?A \rangle$  have  $zs = []$  unfolding  $p'$ -def by (cases zs rule: rev-cases) auto
  then show False using  $\langle ?A \rangle$  using  $\langle w \neq x' \rangle \langle last\ p = (w,x) \rangle$  unfolding
p'-def by auto
  qed }
then have progressing p' by (auto simp: progressing-def)
ultimately show  $?thesis$  unfolding find-endpoint-path-inv-def by blast
qed

lemma no-loop-path:
  assumes  $u = v$  and ig: IGraph-inv G
  shows  $\neg (\exists p\ w. pre-digraph.iapath (mk-graph\ G)\ u\ ((u, v) \# p)\ w)$ 
proof -
  interpret ppg: pair-pseudo-graph mk-graph G
  using ig by (rule IGraph-imp-ppg-mkg)
  from  $\langle u = v \rangle$  show  $?thesis$ 
  by (auto simp: ppg.gen-iapath-def ppg.apath-Cons-iff)
  (metis hd-in-set ppg.awalk-verts-non-Nil ppg.awhd-of-awalk pre-digraph.awalkI-apath)
qed

lemma (in find-endpoint-impl) find-endpoint-spec:
   $\forall \sigma. \Gamma \vdash_t \{ \sigma. select-nodes-prop\ 'G\ 'H \wedge loop-free (mk-graph\ 'G) \wedge 'v-tail \in set$ 
  (ig-verts 'H)  $\wedge iadj\ 'G\ 'v-tail\ 'v-next \wedge IGraph-inv\ 'G \}$ 
   $'R := PROC\ find-endpoint('G, 'H, 'v-tail, 'v-next)$ 
   $\{ case\ 'R\ of\ None \Rightarrow \neg (\exists p\ w. pre-digraph.iapath (mk-graph\ \sigma\ G)\ \sigma\ v-tail\ ((\sigma\ v-tail,$ 
   $\sigma\ v-next) \# p)\ w)$ 
   $| Some\ w \Rightarrow (\exists p. pre-digraph.iapath (mk-graph\ \sigma\ G)\ \sigma\ v-tail\ ((\sigma\ v-tail, \sigma\ v-next)$ 
   $\# p)\ w) \}$ 
  apply vcg-step
  apply (rewrite
    at whileAnno - (named-loop "path") - -
    in for ( $\sigma$ )
    to whileAnno -
     $\{ find-endpoint-path-inv\ 'G\ 'H\ 'len\ 'v-tail\ 'v-next\ v0\ v1$ 
     $\wedge 'v-tail = \sigma\ v-tail \wedge 'v-next = \sigma\ v-next \wedge 'G = \sigma\ G \wedge 'H = \sigma\ H$ 
     $\wedge 0 < 'len$ 
     $\wedge 'v-tail \in set (ig-verts\ 'H) \wedge select-nodes-prop\ 'G\ 'H \wedge IGraph-inv\ 'G$ 
   $\wedge loop-free (mk-graph\ 'G) \}$ 
    (MEASURE Suc (ig-verts-cnt 'G) - 'len)
    -
    annotate-named-loop-var)
  apply (rewrite
    at whileAnno - (named-loop "arcs") - -
    in for ( $\sigma$ )
    to whileAnnoFix -
    ( $\lambda(v0,v1,len). \{ find-endpoint-arcs-inv\ 'G\ 'found\ 'i\ v0\ v1\ 'v0\ 'v1$ 

```

```

    ∧ 'i ≤ length (ig-in-out-arcs 'G v1) ∧ 'io-arcs = ig-in-out-arcs 'G v1
    ∧ 'v-tail = σv-tail ∧ 'v-next = σv-next ∧ 'G = σG ∧ 'H = σH
    ∧ 'len = len
    ∧ 'v-tail ∈ set (ig-verts 'H) ∧ select-nodes-prop 'G 'H ∧ IGraph-inv 'G
  })
  (λ-. (MEASURE length 'io-arcs - 'i))
  -
  annotate-named-loop-var-fix)
apply vcg
  apply (fastforce simp: find-endpoint-path-first no-loop-path)
  apply (match premises in find-endpoint-path-inv - - - - v0 v1 for v0 v1
    ⇒ ⟨rule exI[where x=v0], rule exI[where x=v1]⟩)
  apply (fastforce simp: find-endpoint-arcs-last find-endpoint-arcs-0 find-endpoint-path-step
    elim: find-endpoint-path-lastE)
  apply (fastforce elim: find-endpoint-arcs-step1E find-endpoint-arcs-step2E)
  apply (fastforce dest: find-endpoint-path-last1 find-endpoint-path-last2D)
done

```

16.2.5 Procedure *contract*

definition *contract-iter-nodes-inv* **where**

```

contract-iter-nodes-inv G H k ≡
  set (ig-arcs H) = (⋃ i<k. {(u,v). u = (ig-verts H ! i) ∧ (∃ p. pre-digraph.iapath
    (mk-graph G) u p v)})

```

definition *contract-iter-adj-inv* :: IGraph ⇒ IGraph ⇒ IGraph ⇒ nat ⇒ nat ⇒ bool **where**

```

contract-iter-adj-inv G H0 H u l ≡ (set (ig-arcs H) - ({u} × UNIV) = set
  (ig-arcs H0)) ∧
  ig-verts H = ig-verts H0 ∧
  (∀ v. (u,v) ∈ set (ig-arcs H) ⟷
    ((∃ j<l. ∃ p. pre-digraph.iapath (mk-graph G) u ((u, ig-opposite G (ig-in-out-arcs
    G u ! j) u) # p) v)))

```

lemma *contract-iter-adj-invE*:

```

assumes contract-iter-adj-inv G H0 H u l
obtains set (ig-arcs H) - ({u} × UNIV) = set (ig-arcs H0) ig-verts H = ig-verts
  H0
  ∧ v. (u,v) ∈ set (ig-arcs H) ⟷ ((∃ j<l. ∃ p. pre-digraph.iapath (mk-graph G)
  u ((u, ig-opposite G (ig-in-out-arcs G u ! j) u) # p) v))
using assms unfolding contract-iter-adj-inv-def by auto

```

lemma *contract-iter-adj-inv-def'*:

```

contract-iter-adj-inv G H0 H u l ⟷ (
  set (ig-arcs H) - ({u} × UNIV) = set (ig-arcs H0)) ∧ ig-verts H = ig-verts
  H0 ∧
  (∀ v. ((∃ j<l. ∃ p. pre-digraph.iapath (mk-graph G) u ((u, ig-opposite G (ig-in-out-arcs
  G u ! j) u) # p) v) ⟶ (u,v) ∈ set (ig-arcs H)) ∧
  ((u,v) ∈ set (ig-arcs H) ⟶ ((∃ j<l. ∃ p. pre-digraph.iapath (mk-graph G) u

```

((*u*, *ig-opposite* *G* (*ig-in-out-arcs* *G* *u* ! *j*) *u*) # *p*) *v*)))
unfolding *contract-iter-adj-inv-def* **by** *metis*

lemma *select-nodes-prop-add-e[simp]*:
select-nodes-prop *G* (*ig-add-e* *H* *u* *v*) = *select-nodes-prop* *G* *H*
by (*simp* *add*: *select-nodes-prop-def* *mkg-simps*)

lemma *contract-iter-adj-inv-step1*:
assumes *pair-pseudo-graph* (*mk-graph* *G*)
assumes *ciai*: *contract-iter-adj-inv* *G* *H0* *H* *u* *l*
assumes *iapath*: *pre-digraph.iapath* (*mk-graph* *G*) *u* ((*u*, *ig-opposite* *G* (*ig-in-out-arcs* *G* *u* ! *l*) *u*) # *p*) *v*
shows *contract-iter-adj-inv* *G* *H0* (*ig-add-e* *H* *u* *w*) *u* (*Suc* *l*)

proof –

interpret *pair-pseudo-graph* *mk-graph* *G* **by** *fact*
{ **fix** *v* *j* **assume** *: *j* < *Suc* *l* ∃ *p*. *iapath* *u* ((*u*, *ig-opposite* *G* (*ig-in-out-arcs* *G* *u* ! *j*) *u*) # *p*) *v*
then **have** (*u*, *v*) ∈ *set* (*ig-arcs* (*ig-add-e* *H* *u* *w*))
proof (*cases* *j* < *l*)

case *True* **with** * *ciai* **show** ?*thesis*
by (*auto simp*: *contract-iter-adj-inv-def*)[]

next

case *False* **with** * **have** *j* = *l* **by** *arith*
with *(*2*) **obtain** *q* **where** **: *iapath* *u* ((*u*, *ig-opposite* *G* (*ig-in-out-arcs* *G* *u* ! *l*) *u*) # *q*) *v*
by *metis*

with *iapath* **have** *p* = *q*
using *verts3-in-verts*[**where** *G*=*mk-graph* *G*]
by (*auto elim*: *gen-iapath-same2E*[*rotated* 2])

with ** *iapath* **have** *v* = *w*

by (*auto simp*: *pre-digraph.gen-iapath-def* *pre-digraph.apath-def* *elim*: *pre-digraph.awalk-nonempty-ends*[*rotated* 2])
then **show** ?*thesis* **by** *simp*

qed }

moreover

{ **fix** *v* **assume** *: (*u*,*v*) ∈ *set* (*ig-arcs* (*ig-add-e* *H* *u* *w*))
have (∃ *j*<*Suc* *l*. ∃ *p*. *gen-iapath* (*verts3* (*mk-graph* *G*)) *u* ((*u*, *ig-opposite* *G* (*ig-in-out-arcs* *G* *u* ! *j*) *u*) # *p*) *v*)
proof *cases*

assume *v* = *w* **then** **show** ?*thesis* **using** *iapath* **by** *auto*

next

assume *v* ≠ *w* **then** **show** ?*thesis* **using** *ciai* *

unfolding *contract-iter-adj-inv-def* **by** (*auto intro*: *less-SucI*)

qed }

moreover

have *set* (*ig-arcs* (*ig-add-e* *H* *u* *w*)) – (*{u}* × *UNIV*) = *set* (*ig-arcs* *H0*)
ig-verts (*ig-add-e* *H* *u* *w*) = *ig-verts* *H0*

using *ciai* **unfolding** *contract-iter-adj-inv-def* **by** *auto*

ultimately

show ?*thesis* **unfolding** *contract-iter-adj-inv-def* **by** *metis*

qed

lemma *contract-iter-adj-inv-step2*:

assumes *ciai*: *contract-iter-adj-inv* G $H0$ H u l
assumes *iapath*: $\bigwedge p w. \neg \text{pre-digraph.iapath } (\text{mk-graph } G) u ((u, \text{ig-opposite } G$
 $(\text{ig-in-out-arcs } G u ! l) u) \# p) w$
shows *contract-iter-adj-inv* G $H0$ H u $(\text{Suc } l)$
proof –
 { **fix** $v j$ **assume** $*$: $j < \text{Suc } l \exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) u ((u,$
 $\text{ig-opposite } G (\text{ig-in-out-arcs } G u ! j) u) \# p) v$
 then have $(u, v) \in \text{set } (\text{ig-arcs } H)$
 proof (*cases* $j < l$)
 case *True* **with** $*$ *ciai* **show** *?thesis*
 by (*auto simp: contract-iter-adj-inv-def*)
 next
 case *False* **with** $*$ **have** $j = l$ **by** *auto*
 with $*$ **show** *?thesis* **using** *iapath* **by** *metis*
 qed }
 moreover
 { **fix** v **assume** $*$: $(u, v) \in \text{set } (\text{ig-arcs } H)$
 then have $(\exists j < \text{Suc } l. \exists p. \text{pre-digraph.gen-iapath } (\text{mk-graph } G) (\text{verts3 } (\text{mk-graph}$
 $G)) u ((u, \text{ig-opposite } G (\text{ig-in-out-arcs } G u ! j) u) \# p) v)$
 using *ciai* **unfolding** *contract-iter-adj-inv-def* **by** (*auto intro: less-SucI*) }
 moreover
 have $\text{set } (\text{ig-arcs } H) - (\{u\} \times \text{UNIV}) = \text{set } (\text{ig-arcs } H0) \text{ ig-verts } H = \text{ig-verts}$
 $H0$
 using *ciai* **unfolding** *contract-iter-adj-inv-def* **by** (*auto simp:*)
 ultimately
 show *?thesis* **unfolding** *contract-iter-adj-inv-def* **by** *metis*
qed

definition *contract-iter-adj-prop* **where**

contract-iter-adj-prop G $H0$ H $u \equiv \text{ig-verts } H = \text{ig-verts } H0$
 $\wedge \text{set } (\text{ig-arcs } H) = \text{set } (\text{ig-arcs } H0) \cup (\{u\} \times \{v. \exists p. \text{pre-digraph.iapath}$
 $(\text{mk-graph } G) u p v\})$

lemma *contract-iter-adj-propI*:

assumes *nodes*: *contract-iter-nodes-inv* G H i
assumes *ciai*: *contract-iter-adj-inv* G H H' u $(\text{length } (\text{ig-in-out-arcs } G u))$
assumes u : $u = \text{ig-verts } H ! i$
shows *contract-iter-adj-prop* G H H' u
proof –
 have $\text{ig-verts } H' = \text{ig-verts } H$
 using *ciai* **unfolding** *contract-iter-adj-inv-def* **by** *auto*
 moreover
 have $\text{set } (\text{ig-arcs } H') \subseteq \text{set } (\text{ig-arcs } H) \cup (\{u\} \times \{v. \exists p. \text{pre-digraph.iapath}$
 $(\text{mk-graph } G) u p v\})$

```

    using ciai unfolding contract-iter-adj-inv-def by auto
  moreover
  { fix v p assume path: pre-digraph.iapath (mk-graph G) u p v
  then obtain e es where p = e # es by (cases p) (auto simp: pre-digraph.gen-iapath-def)
  then have  $e \in \text{parcs } (\text{mk-graph } G)$  using path
  by (auto simp: pre-digraph.gen-iapath-def pre-digraph.apath-def pre-digraph.awalk-def)
  moreover
  then obtain w where  $e = (u, w)$  using  $\langle p = e \# es \rangle$  path
  by (cases e) (auto simp: pre-digraph.gen-iapath-def pre-digraph.apath-def
pre-digraph.awalk-def pre-digraph.cas.simps)
  ultimately
  have  $(u, w) \in \text{set } (\text{ig-arcs } G) \vee (w, u) \in \text{set } (\text{ig-arcs } G)$ 
  unfolding mk-graph-def by (auto simp: parcs-mk-symmetric mkg'-simps)
  then obtain e' where  $H1: e' = (u, w) \vee e' = (w, u)$  and  $e' \in \text{set } (\text{ig-arcs } G)$ 
  by auto
  then have  $e' \in \text{set } (\text{ig-in-out-arcs } G \ u)$ 
  unfolding ig-in-out-arcs-def by auto
  then obtain k where  $H2: \text{ig-in-out-arcs } G \ u \ ! \ k = e' \ k < \text{length } (\text{ig-in-out-arcs } G \ u)$ 
  by (auto simp: in-set-conv-nth)
  have opp-e': ig-opposite G e' u = w using H1 unfolding ig-opposite-def by
auto
  have  $(u, v) \in \text{set } (\text{ig-arcs } H')$ 
  using ciai unfolding contract-iter-adj-inv-def'
  apply safe
  apply (erule allE[where  $x=v$ ])
  apply safe
  apply (erule notE)
  apply (rule exI[where  $x=k$ ])
  apply (simp add: H2 opp-e')
  using path  $\langle e = (u, w) \rangle \langle p = e \# es \rangle$  by auto }
  then have  $\text{set } (\text{ig-arcs } H) \cup (\{u\} \times \{v. \exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) \ u \ p \ v\}) \subseteq \text{set } (\text{ig-arcs } H')$ 
  using ciai unfolding contract-iter-adj-inv-def by auto
  ultimately
  show ?thesis unfolding contract-iter-adj-prop-def by blast
qed

```

lemma *contract-iter-nodes-inv-step*:

```

  assumes nodes: contract-iter-nodes-inv G H i
  assumes adj: contract-iter-adj-inv G H H' (ig-verts H ! i) (length (ig-in-out-arcs G (ig-verts H ! i)))
  assumes snp: select-nodes-prop G H
  shows contract-iter-nodes-inv G H' (Suc i)
proof -
  have ciap: contract-iter-adj-prop G H H' (ig-verts H ! i)
  using nodes adj by (rule contract-iter-adj-propI) simp
  then have ie-H':  $\text{set } (\text{ig-arcs } H') = \text{set } (\text{ig-arcs } H) \cup \{(u, v). u = \text{ig-verts } H' \ ! \ i \wedge (\exists p. \text{pre-digraph.gen-iapath } (\text{mk-graph } G) (\text{verts3 } (\text{mk-graph } G)) \ u \ p \ v)\}$ 

```

and [*simp*]: $ig\text{-verts } H' = ig\text{-verts } H$
unfolding *contract-iter-adj-prop-def* **by** *auto*
have *ie-H*: $set (ig\text{-arcs } H) = (\bigcup j < i. \{(u, v). u = ig\text{-verts } H' ! j \wedge (\exists p. pre\text{-digraph.gen-iapath } (mk\text{-graph } G) (verts3 (mk\text{-graph } G)) u p v)\})$
using *nodes* **unfolding** *contract-iter-nodes-inv-def* **by** *simp*

have *: $\bigwedge S k. (\bigcup i < Suc k. S i) = (\bigcup i < k. S i) \cup S k$
by (*metis UN-insert lessThan-Suc sup-commute*)

show *?thesis* **by** (*simp only: contract-iter-nodes-inv-def ie-H ie-H' **)
qed

lemma *contract-iter-nodes-0*:
assumes $set (ig\text{-arcs } H) = \{\}$ **shows** *contract-iter-nodes-inv* $G H 0$
using *assms* **unfolding** *contract-iter-nodes-inv-def* **by** *simp*

lemma *contract-iter-adj-0*:
assumes *nodes*: *contract-iter-nodes-inv* $G H i$
assumes *i*: $i < ig\text{-verts-cnt } H$
shows *contract-iter-adj-inv* $G H H (ig\text{-verts } H ! i) 0$
using *assms* *distinct-ig-verts*
unfolding *contract-iter-adj-inv-def* *contract-iter-nodes-inv-def*
by (*auto simp: distinct-conv-nth*)

lemma *snp-vertexes*:
assumes *select-nodes-prop* $G H u \in set (ig\text{-verts } H)$ **shows** $u \in set (ig\text{-verts } G)$
using *assms* **unfolding** *select-nodes-prop-def* **by** (*auto simp: verts3-def mkg-simps*)

lemma *igraph-ig-add-eI*:
assumes *IGraph-inv* G
assumes $u \in set (ig\text{-verts } G) v \in set (ig\text{-verts } G)$
shows *IGraph-inv* (*ig-add-e* $G u v$)
using *assms* **unfolding** *IGraph-inv-def* **by** *auto*

lemma *snp-iapath-ends-in*:
assumes *select-nodes-prop* $G H$
assumes *pre-digraph.iapath* (*mk-graph* G) $u p v$
shows $u \in set (ig\text{-verts } H) v \in set (ig\text{-verts } H)$
using *assms* **unfolding** *pre-digraph.gen-iapath-def* *select-nodes-prop-def* *verts3-def*
by (*auto simp: mkg-simps*)

lemma *contract-iter-nodes-last*:
assumes *nodes*: *contract-iter-nodes-inv* $G H (ig\text{-verts-cnt } H)$
assumes *snp*: *select-nodes-prop* $G H$
assumes *igraph*: *IGraph-inv* G
shows *mk-graph'* $H = contr\text{-graph } (mk\text{-graph } G)$ (**is** *?t1*)
and *symmetric* (*mk-graph'* H) (**is** *?t2*)

proof –
interpret *ppg-mkG*: *pair-pseudo-graph* *mk-graph* G

```

    using igraph by (rule IGraph-imp-ppg-mkg)
  { fix u v p assume pre-digraph.iapath (mk-graph G) u p v
    then have  $\exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) \ v \ p \ u$ 
      using ppg-mkG.gen-iapath-rev-path[where u=u and v=v, symmetric] by auto
    }
  then have ie-sym:  $\bigwedge u \ v. (\exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) \ u \ p \ v) \longleftrightarrow (\exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) \ v \ p \ u)$ 
    by auto

  from nodes have set (ig-arcs H) =  $\{(u, v). u \in \text{set } (\text{ig-verts } H) \wedge (\exists p. \text{pre-digraph.gen-iapath } (\text{mk-graph } G) \ (\text{verts3 } (\text{mk-graph } G)) \ u \ p \ v))\}$ 
    unfolding contract-iter-nodes-inv-def by (auto simp: in-set-conv-nth)
  then have *: set (ig-arcs H) =  $\{(u, v). (\exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) \ u \ p \ v)\}$ 
    using snp by (auto simp: snp-iapath-ends-in(1))
  then have **: set (ig-arcs H) =  $(\lambda(a, b). (b, a)) \ ' \ \{(u, v). (\exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) \ u \ p \ v)\}$ 
    using ie-sym by fastforce

  have sym: symmetric (mk-graph' H)
    unfolding symmetric-conv by (auto simp: mkg'-simps * ie-sym)

  have pverts (mk-graph' H) = verts3 (mk-graph G)
    using snp unfolding select-nodes-prop-def by (simp add: mkg-simps mkg'-simps)
  moreover
  have parcs (mk-graph' H) =  $\{(u, v). (\exists p. \text{ppg-mkG.iapath } u \ p \ v)\}$ 
    using * by (auto simp: mkg-simps mkg'-simps)
  ultimately show ?t1 ?t2
    using snp sym unfolding gen-contr-graph-def select-nodes-prop-def by auto
qed

lemma (in contract-impl) contract-spec:
   $\forall \sigma. \Gamma \vdash_t \{\sigma. \text{select-nodes-prop } 'G \ 'H \wedge \text{IGraph-inv } 'G \wedge \text{loop-free } (\text{mk-graph } 'G) \wedge \text{IGraph-inv } 'H \wedge \text{set } (\text{ig-arcs } 'H) = \{\}\}$ 
   $'R ::= \text{PROC } \text{contract}('G, 'H)$ 
 $\{\ 'G = \sigma G \wedge \text{mk-graph}' \ 'R = \text{contr-graph } (\text{mk-graph } 'G) \wedge \text{symmetric } (\text{mk-graph}' \ 'R) \wedge \text{IGraph-inv } 'R \}$ 
  apply vcg-step
  apply (rewrite
    at whileAnno - (named-loop "iter-nodes") - -
  in for ( $\sigma$ )
  to whileAnno -
     $\{\text{contract-iter-nodes-inv } 'G \ 'H \ 'i$ 
       $\wedge \text{select-nodes-prop } 'G \ 'H \wedge 'i \leq \text{ig-verts-cnt } 'H \wedge \text{IGraph-inv } 'G \wedge$ 
loop-free (mk-graph 'G)
       $\wedge \text{IGraph-inv } 'H \wedge 'G = \sigma G \}$ 
    (MEASURE ig-verts-cnt 'H - 'i)
    -
  annotate-named-loop-var)

```

```

apply (rewrite
  at whileAnno - (named-loop "iter-adj") - -
  in for ( $\sigma$ )
  to whileAnnoFix -
    ( $\lambda(H, u, i). \{ \text{contract-iter-adj-inv } 'G \ H \ 'H \ u \ 'j$ 
       $\wedge \text{select-nodes-prop } 'G \ 'H \wedge 'u = u \wedge 'j \leq \text{length} (\text{ig-in-out-arcs } 'G \ 'u)$ 
 $\wedge 'io\text{-arcs} = \text{ig-in-out-arcs } 'G \ 'u$ 
       $\wedge u \in \text{set} (\text{ig-verts } 'H) \wedge \text{IGraph-inv } 'G \wedge \text{loop-free} (\text{mk-graph } 'G) \wedge$ 
 $\text{IGraph-inv } 'H \wedge 'G = \sigma \ G \wedge 'i = i \}$ )
    ( $\lambda\cdot. (\text{MEASURE length } 'io\text{-arcs} - 'j)$ )
    -
  annotate-named-loop-var-fix)
apply vcg
  apply (fastforce simp: contract-iter-nodes-0)
  apply (match premises in select-nodes-prop - H for H  $\Rightarrow$   $\langle \text{rule exI[where } x=H \rangle$ )
  apply (fastforce simp: contract-iter-adj-0 contract-iter-nodes-inv-step elim: contract-iter-adj-invE)
  apply (fastforce simp: contract-iter-adj-inv-step2 contract-iter-adj-inv-step1
    IGraph-imp-ppg-mkg igrph-ig-add-eI snp-iapath-ends-in iadj-io-edge snp-vertexes)
  apply (fastforce simp: not-less intro: contract-iter-nodes-last)
done

```

16.2.6 Procedure is-K33

definition *is-K33-colorize-inv* :: IGraph \Rightarrow ig-vertex \Rightarrow nat \Rightarrow (ig-vertex \Rightarrow bool) \Rightarrow bool **where**

is-K33-colorize-inv G u k blue $\equiv \forall v \in \text{set} (\text{ig-verts } G). \text{blue } v \longleftrightarrow$
 $(\exists i < k. v = \text{ig-opposite } G (\text{ig-in-out-arcs } G \ u \ ! \ i) \ u)$

definition *is-K33-component-size-inv* :: IGraph \Rightarrow nat \Rightarrow (ig-vertex \Rightarrow bool) \Rightarrow nat \Rightarrow bool **where**

is-K33-component-size-inv G k blue cnt $\equiv \text{cnt} = \text{card} \{i. i < k \wedge \text{blue} (\text{ig-verts } G \ ! \ i)\}$

definition *is-K33-outer-inv* :: IGraph \Rightarrow nat \Rightarrow (ig-vertex \Rightarrow bool) \Rightarrow bool **where**
is-K33-outer-inv G k blue $\equiv \forall i < k. \forall v \in \text{set} (\text{ig-verts } G).$

blue (ig-verts G ! i) = blue v \longleftrightarrow (ig-verts G ! i, v) \notin set (ig-arcs G)

definition *is-K33-inner-inv* :: IGraph \Rightarrow nat \Rightarrow nat \Rightarrow (ig-vertex \Rightarrow bool) \Rightarrow bool **where**

is-K33-inner-inv G k l blue $\equiv \forall j < l.$

blue (ig-verts G ! k) = blue (ig-verts G ! j) \longleftrightarrow (ig-verts G ! k, ig-verts G ! j) \notin set (ig-arcs G)

lemma *is-K33-colorize-0*: *is-K33-colorize-inv* G u 0 ($\lambda\cdot. \text{False}$)

unfolding *is-K33-colorize-inv-def* **by** auto

lemma *is-K33-component-size-0*: *is-K33-component-size-inv* G 0 blue 0

unfolding *is-K33-component-size-inv-def* **by** *auto*

lemma *is-K33-outer-0*: *is-K33-outer-inv G 0 blue*
unfolding *is-K33-outer-inv-def* **by** *auto*

lemma *is-K33-inner-0*: *is-K33-inner-inv G k 0 blue*
unfolding *is-K33-inner-inv-def* **by** *auto*

lemma *is-K33-colorize-last*:
assumes $u \in \text{set } (ig\text{-verts } G)$
shows *is-K33-colorize-inv G u (length (ig-in-out-arcs G u)) blue*
 $= (\forall v \in \text{set } (ig\text{-verts } G). \text{blue } v \longleftrightarrow iadj\ G\ u\ v) \text{ (is } ?L = ?R)$

proof –
{ **fix** v
have $(\exists i < \text{length } (ig\text{-in-out-arcs } G\ u). v = ig\text{-opposite } G\ (ig\text{-in-out-arcs } G\ u\ i)\ u)$
 $\longleftrightarrow (\exists e \in \text{set } (ig\text{-in-out-arcs } G\ u). v = ig\text{-opposite } G\ e\ u) \text{ (is } ?A \longleftrightarrow -)$
by *auto (auto simp: in-set-conv-nth)*
also have $\dots \longleftrightarrow iadj\ G\ u\ v$
using *assms* **by** *(force simp: iadj-io-edge ig-opposite-simps dest: iadjD)*
finally have $?A \longleftrightarrow iadj\ G\ u\ v . \}$
then show *?thesis unfolding is-K33-colorize-inv-def* **by** *auto*

qed

lemma *is-K33-component-size-last*:
assumes $k = ig\text{-verts-cnt } G$
shows *is-K33-component-size-inv G k blue cnt* $\longleftrightarrow \text{card } \{u \in \text{set } (ig\text{-verts } G). \text{blue } u\} = cnt$

proof –
have $*: \{u \in \text{set } (ig\text{-verts } G). \text{blue } u\} = (\lambda n. ig\text{-verts } G\ !\ n) \text{ ' } \{i. i < ig\text{-verts-cnt } G \wedge \text{blue } (ig\text{-verts } G\ !\ i)\}$
by *(auto simp: in-set-conv-nth)*
have *inj-on* $(\lambda n. ig\text{-verts } G\ !\ n) \{i. i < ig\text{-verts-cnt } G \wedge \text{blue } (ig\text{-verts } G\ !\ i)\}$
using *distinct-ig-verts* **by** *(auto simp: nth-eq-iff-index-eq intro: inj-onI)*
with *assms* **show** *?thesis*
unfolding $* \text{ is-K33-component-size-inv-def}$
by *(auto intro: card-image)*

qed

lemma *is-K33-outer-last*:
is-K33-outer-inv G (ig-verts-cnt G) blue $\longleftrightarrow (\forall u \in \text{set } (ig\text{-verts } G). \forall v \in \text{set } (ig\text{-verts } G). \text{blue } u = \text{blue } v \longleftrightarrow (u, v) \notin \text{set } (ig\text{-arcs } G))$
unfolding *is-K33-outer-inv-def* **by** *(simp add: All-set-ig-verts)*

lemma *is-K33-inner-last*:
is-K33-inner-inv G k (ig-verts-cnt G) blue $\longleftrightarrow (\forall v \in \text{set } (ig\text{-verts } G). \text{blue } (ig\text{-verts } G\ !\ k) = \text{blue } v \longleftrightarrow (ig\text{-verts } G\ !\ k, v) \notin \text{set } (ig\text{-arcs } G))$
unfolding *is-K33-inner-inv-def* **by** *(simp add: All-set-ig-verts)*

lemma *is-K33-colorize-step*:
fixes $G\ u\ i\ blue$
assumes *colorize: is-K33-colorize-inv* $G\ u\ k\ blue$
shows *is-K33-colorize-inv* $G\ u\ (Suc\ k)$ (*blue (ig-opposite* $G\ (ig-in-out-arcs\ G\ u\ !\ k)\ u := True$)
using *assms* **by** (*auto simp: is-K33-colorize-inv-def elim: less-SucE intro: less-SucI*)

lemma *is-K33-component-size-step1*:
assumes *comp:is-K33-component-size-inv* $G\ k\ blue\ blue-cnt$
assumes *blue: blue (ig-verts* $G\ !\ k)$
shows *is-K33-component-size-inv* $G\ (Suc\ k)\ blue\ (Suc\ blue-cnt)$
proof –
have $\{i.\ i < Suc\ k \wedge blue\ (ig-verts\ G\ !\ i)\}$
 $= insert\ k\ \{i.\ i < k \wedge blue\ (ig-verts\ G\ !\ i)\}$
using *blue* **by** *auto*
with *comp* **show** *?thesis*
unfolding *is-K33-component-size-inv-def* **by** *auto*
qed

lemma *is-K33-component-size-step2*:
assumes *comp:is-K33-component-size-inv* $G\ k\ blue\ blue-cnt$
assumes *blue: ¬blue (ig-verts* $G\ !\ k)$
shows *is-K33-component-size-inv* $G\ (Suc\ k)\ blue\ blue-cnt$
proof –
have $\{i.\ i < Suc\ k \wedge blue\ (ig-verts\ G\ !\ i)\} = \{i.\ i < k \wedge blue\ (ig-verts\ G\ !\ i)\}$
using *blue* **by** (*auto elim: less-SucE*)
with *comp* **show** *?thesis*
unfolding *is-K33-component-size-inv-def* **by** *auto*
qed

lemma *is-K33-outer-step*:
assumes *is-K33-outer-inv* $G\ i\ blue$
assumes *is-K33-inner-inv* $G\ i\ (ig-verts-cnt\ G)\ blue$
shows *is-K33-outer-inv* $G\ (Suc\ i)\ blue$
using *assms* **unfolding** *is-K33-outer-inv-def is-K33-inner-last*
by (*auto intro: less-SucI elim: less-SucE*)

lemma *is-K33-inner-step*:
assumes *is-K33-inner-inv* $G\ i\ j\ blue$
assumes (*blue (ig-verts* $G\ !\ i) = blue\ (ig-verts\ G\ !\ j)$) $\longleftrightarrow (ig-verts\ G\ !\ i,\ ig-verts\ G\ !\ j) \notin set\ (ig-arcs\ G)$
shows *is-K33-inner-inv* $G\ i\ (Suc\ j)\ blue$
using *assms* **by** (*auto simp: is-K33-inner-inv-def elim: less-SucE*)

lemma *K33-mkg'I*:
fixes $G\ col\ cnt$
defines $u \equiv ig-verts\ G\ !\ 0$
assumes *ig: IGraph-inv* G

assumes *iv-cnt*: $ig\text{-verts-cnt } G = 6$ **and** *c1-cnt*: $cnt = 3$
assumes *colorize*: $is\text{-}K33\text{-colorize-inv } G \ u \ (length \ (ig\text{-in-out-arcs } G \ u)) \ blue$
assumes *comp*: $is\text{-}K33\text{-component-size-inv } G \ (ig\text{-verts-cnt } G) \ blue \ cnt$
assumes *outer*: $is\text{-}K33\text{-outer-inv } G \ (ig\text{-verts-cnt } G) \ blue$
shows $K_{3,3} \ (mk\text{-graph}' \ G)$
proof –
have $u \in set \ (ig\text{-verts } G)$ **unfolding** *u-def* **using** *iv-cnt* **by** *auto*
then have $(\forall v \in set \ (ig\text{-verts } G). \ blue \ v \longleftrightarrow \ iadj \ G \ u \ v)$
using *colorize* **by** $(rule \ is\text{-}K33\text{-colorize-last} [THEN \ iffD1])$

define $U \ V$ **where** $U = \{u \in set \ (ig\text{-verts } G). \ \neg blue \ u\}$ **and** $V = \{v \in set \ (ig\text{-verts } G). \ blue \ v\}$
then have *UV-set*: $U \subseteq set \ (ig\text{-verts } G) \ V \subseteq set \ (ig\text{-verts } G) \ U \cup V = set \ (ig\text{-verts } G) \ U \cap V = \{\}$
and *fin-UV*: $finite \ U \ finite \ V$ **by** *auto*

have *card-verts*: $card \ (set \ (ig\text{-verts } G)) = 6$
using *iv-cnt* *distinct-ig-verts* **by** $(simp \ add: \ distinct\text{-card})$

from *ig comp c1-cnt* **have** $card \ V = 3$ **by** $(simp \ add: \ is\text{-}K33\text{-component-size-last} \ V\text{-def})$
moreover have $card \ (U \cup V) = 6$ **using** *UV-set* *distinct-ig-verts* *iv-cnt*
by $(auto \ simp: \ distinct\text{-card})$
ultimately have $card \ U = 3$
by $(simp \ add: \ card\text{-}Un\text{-disjoint} [OF \ fin\text{-}UV \ UV\text{-set}(4)])$
note $cards = \langle card \ V = 3 \rangle \langle card \ U = 3 \rangle \ card\text{-verts}$

from *is-K33-outer-last* $[THEN \ iffD1, \ OF \ outer]$
have $(\forall u \in U. \ \forall v \in V. \ (u, v) \in set \ (ig\text{-arcs } G) \wedge (v, u) \in set \ (ig\text{-arcs } G))$
 $\wedge (\forall u \in U. \ \forall u' \in U. \ (u, u') \notin set \ (ig\text{-arcs } G))$
 $\wedge (\forall v \in V. \ \forall v' \in V. \ (v, v') \notin set \ (ig\text{-arcs } G))$
unfolding *U-def* *V-def* **by** *auto*
then have $U \times V \subseteq set \ (ig\text{-arcs } G) \ V \times U \subseteq set \ (ig\text{-arcs } G)$
 $U \times U \cap set \ (ig\text{-arcs } G) = \{\} \ V \times V \cap set \ (ig\text{-arcs } G) = \{\}$
by *auto*
moreover have $set \ (ig\text{-arcs } G) \subseteq (U \cup V) \times (U \cup V)$
unfolding $\langle U \cup V = \rightarrow \rangle$ **by** $(auto \ simp: \ ig \ set\text{-}ig\text{-arcs}\text{-verts})$
ultimately
have *conn*: $set \ (ig\text{-arcs } G) = U \times V \cup V \times U$
by *blast*

interpret *ppg-mkg'*: *pair-fin-digraph* *mk-graph'* G
using *ig* **by** $(auto \ intro: \ IGraph\text{-imp}\text{-ppd}\text{-mkg}')$

show *?thesis*
unfolding *complete-bipartite-digraph-pair-def* *mkg'-simps*
using *cards* *UV-set* *conn* **by** *simp* *metis*
qed

lemma *K33-mkg'E*:
assumes *K33*: $K_{3,3}$ (*mk-graph' G*)
assumes *ig*: *IGraph-inv G*
assumes *colorize*: *is-K33-colorize-inv G u* (*length (ig-in-out-arcs G u)*) *blue*
and *u*: $u \in \text{set } (ig\text{-verts } G)$
obtains *is-K33-component-size-inv G* (*ig-verts-cnt G*) *blue 3*
is-K33-outer-inv G (*ig-verts-cnt G*) *blue*
proof –
from *K33* **obtain** *U V* **where**
verts-G: $\text{set } (ig\text{-verts } G) = U \cup V$ **and**
arcs-G: $\text{set } (ig\text{-arcs } G) = U \times V \cup V \times U$ **and**
disj-UV: $U \cap V = \{\}$ **and**
card: $\text{card } U = 3 \text{ card } V = 3$
unfolding *complete-bipartite-digraph-pair-def mkg'-simps* **by** *auto*

from *colorize u* **have** *col-adj*: $\bigwedge v. v \in \text{set } (ig\text{-verts } G) \implies \text{blue } v \longleftrightarrow \text{iadj } G u$
v
using *is-K33-colorize-last* **by** *auto*

have *iadj-conv*: $\bigwedge u v. \text{iadj } G u v \longleftrightarrow (u,v) \in U \times V \cup V \times U$
unfolding *iadj-def arcs-G* **by** *auto*

{ **assume** $u \in U$
then have $V = \{v \in \text{set } (ig\text{-verts } G). \text{blue } v\}$
using *disj-UV* **by** (*auto simp: iadj-conv verts-G col-adj*)
then have *is-K33-component-size-inv G* (*ig-verts-cnt G*) *blue 3*
using *ig card* **by** (*simp add: is-K33-component-size-last*)
moreover
have $\bigwedge v. v \in U \cup V \implies \text{blue } v \longleftrightarrow v \in V$
using $\langle u \in U \rangle$ *disj-UV* **by** (*auto simp: verts-G col-adj iadj-conv*)
then have *is-K33-outer-inv G* (*ig-verts-cnt G*) *blue*
using $\langle U \cap V = \{\} \rangle$ **by** (*subst is-K33-outer-last*) (*auto simp: arcs-G verts-G*)
)
ultimately have *?thesis* **by** (*rule that*) }
moreover
{ **assume** $u \in V$
then have $U = \{v \in \text{set } (ig\text{-verts } G). \text{blue } v\}$
using *disj-UV* **by** (*auto simp: iadj-conv verts-G col-adj*)
then have *is-K33-component-size-inv G* (*ig-verts-cnt G*) *blue 3*
using *ig card* **by** (*simp add: is-K33-component-size-last*)
moreover
have $\bigwedge v. v \in U \cup V \implies \text{blue } v \longleftrightarrow v \in U$
using $\langle u \in V \rangle$ *disj-UV* **by** (*auto simp: verts-G col-adj iadj-conv*)
then have *is-K33-outer-inv G* (*ig-verts-cnt G*) *blue*
using $\langle U \cap V = \{\} \rangle$ **by** (*subst is-K33-outer-last*) (*auto simp: arcs-G verts-G*)
)
ultimately have *?thesis* **by** (*rule that*) }
ultimately show *?thesis* **using** *verts-G u* **by** *blast*
qed

lemma *K33-card*:

assumes $K_{3,3}$ (*mk-graph' G*) **shows** *ig-verts-cnt G = 6*

proof –

from *assms* **have** *card (verts (mk-graph' G)) = 6*

unfolding *complete-bipartite-digraph-pair-def* **by** (*auto simp: card-Un-disjoint*)

then show *?thesis*

using *distinct-ig-verts* **by** (*auto simp: mkg'-simps distinct-card*)

qed

abbreviation (*input*) *is-K33-colorize-inv-last* :: *IGraph* \Rightarrow (*ig-vertex* \Rightarrow *bool*) \Rightarrow *bool* **where**

is-K33-colorize-inv-last G blue \equiv *is-K33-colorize-inv G (ig-verts G ! 0) (length (ig-in-out-arcs G (ig-verts G ! 0))) blue*

abbreviation (*input*) *is-K33-component-size-inv-last* :: *IGraph* \Rightarrow (*ig-vertex* \Rightarrow *bool*) \Rightarrow *bool* **where**

is-K33-component-size-inv-last G blue \equiv *is-K33-component-size-inv G (ig-verts-cnt G) blue 3*

lemma *is-K33-outerD*:

assumes *is-K33-outer-inv G (ig-verts-cnt G) blue*

assumes $i < \text{ig-verts-cnt } G$ $j < \text{ig-verts-cnt } G$

shows (*blue (ig-verts G ! i) = blue (ig-verts G ! j)*) \longleftrightarrow (*ig-verts G ! i, ig-verts G ! j*) \notin *set (ig-arcs G)*

using *assms* **unfolding** *is-K33-outer-last* **by** *auto*

lemma (**in** *is-K33-impl*) *is-K33-spec*:

$\forall \sigma. \Gamma \vdash_t \{ \sigma. \text{IGraph-inv } 'G \wedge \text{symmetric (mk-graph' } 'G) \}$

$'R ::= \text{PROC is-K33}('G)$

$\{ 'G = \sigma G \wedge 'R = K_{3,3}(\text{mk-graph' } 'G) \}$

apply *vcg-step*

apply (*rewrite*

at whileAnno - (named-loop "colorize") - -

in for (σ)

to whileAnno -

$\{ \text{is-K33-colorize-inv } 'G 'u 'i 'blue \wedge 'i \leq \text{length } 'io\text{-arcs}$

$\wedge 'io\text{-arcs} = \text{ig-in-out-arcs } 'G 'u \wedge 'u = \text{ig-verts } 'G ! 0 \wedge 'G = \sigma G \wedge$

IGraph-inv } 'G

$\wedge 'u = \text{ig-verts } 'G ! 0 \wedge \text{ig-verts-cnt } 'G = 6 \}$

(*MEASURE length } 'io-arcs - } 'i*)

-

annotate-named-loop-var)

apply (*rewrite*

at whileAnno - (named-loop "component-size") - -

in for (σ)

to whileAnnoFix -

($\lambda \text{blue. } \{ \text{is-K33-component-size-inv } 'G 'i 'blue 'blue\text{-cnt}$

$\wedge 'i \leq \text{ig-verts-cnt } 'G \wedge 'blue = \text{blue} \wedge 'G = \sigma G \wedge \text{IGraph-inv } 'G$

```

    ∧ ig-verts-cnt ' G = 6 ∧ is-K33-colorize-inv-last ' G ' blue } }
    (λ-. (MEASURE ig-verts-cnt ' G - ' i))
  -
  annotate-named-loop-var-fix)
apply (rewrite
  at whileAnno - (named-loop "connected-outer") - -
  in for (σ)
  to whileAnnoFix -
    (λblue. { is-K33-outer-inv ' G ' i ' blue ∧ ' i ≤ ig-verts-cnt ' G
      ∧ ' blue = blue ∧ ' G = σ G ∧ IGraph-inv ' G
      ∧ ig-verts-cnt ' G = 6 ∧ is-K33-colorize-inv-last ' G ' blue ∧ is-K33-component-size-inv-last
' G ' blue } }
    (λ-. (MEASURE ig-verts-cnt ' G - ' i))
  -
  annotate-named-loop-var-fix)
apply (rewrite
  at whileAnno - (named-loop "connected-inner") - -
  in for (σ)
  to whileAnnoFix -
    (λ(i,blue). { is-K33-inner-inv ' G ' i ' j ' blue ∧ ' j ≤ ig-verts-cnt ' G
      ∧ ' i = i ∧ ' i < ig-verts-cnt ' G ∧ ' blue = blue ∧ ' G = σ G ∧ IGraph-inv
' G ∧ ' u = ig-verts ' G ! ' i
      ∧ ig-verts-cnt ' G = 6 ∧ is-K33-colorize-inv-last ' G ' blue ∧ is-K33-component-size-inv-last
' G ' blue } }
    (λ-. (MEASURE ig-verts-cnt ' G - ' j))
  -
  annotate-named-loop-var-fix)
apply vcg
  apply (fastforce simp: is-K33-colorize-0 is-K33-component-size-0 is-K33-outer-0
is-K33-component-size-last
    elim: K33-mkg'E dest: K33-card intro: K33-mkg'I)
  apply (fastforce simp add: is-K33-colorize-step)
  apply (fastforce simp: is-K33-colorize-0 is-K33-component-size-0 is-K33-outer-0
is-K33-component-size-last
    elim: K33-mkg'E intro: K33-mkg'I)
  apply (fastforce simp: is-K33-component-size-step1 is-K33-component-size-step2)
  apply (fastforce simp: is-K33-inner-0 is-K33-outer-step)
  apply (simp only: simp-thms)
  apply (intro conjI allI impI notI)
  apply (fastforce elim: K33-mkg'E dest: is-K33-outerD)
  apply (fastforce elim: K33-mkg'E dest: is-K33-outerD)
  apply (simp add: is-K33-inner-step; fail)
apply linarith
done

```

16.2.7 Procedure *is-K5*

definition

is-K5-outer-inv G $k \equiv \forall i < k. \forall v \in \text{set } (ig\text{-verts } G). ig\text{-verts } G ! i \neq v$

$\longleftrightarrow (ig\text{-verts } G ! i, v) \in \text{set } (ig\text{-arcs } G)$

definition

$is\text{-}K5\text{-inner-inv } G k l \equiv \forall j < l. ig\text{-verts } G ! k \neq ig\text{-verts } G ! j$
 $\longleftrightarrow (ig\text{-verts } G ! k, ig\text{-verts } G ! j) \in \text{set } (ig\text{-arcs } G)$

lemma *K5-card*:

assumes $K_5 (mk\text{-graph}' G)$ **shows** $ig\text{-verts-cnt } G = 5$
using *assms distinct-ig-verts unfolding complete-digraph-pair-def*
by (*auto simp add: mkg'-simps distinct-card*)

lemma *is-K5-inner-0*: $is\text{-}K5\text{-inner-inv } G k 0$

unfolding *is-K5-inner-inv-def* **by** *auto*

lemma *is-K5-inner-last*:

assumes $l = ig\text{-verts-cnt } G$
shows $is\text{-}K5\text{-inner-inv } G k l \longleftrightarrow (\forall v \in \text{set } (ig\text{-verts } G). ig\text{-verts } G ! k \neq v$
 $\longleftrightarrow (ig\text{-verts } G ! k, v) \in \text{set } (ig\text{-arcs } G))$

proof –

have $\bigwedge v. v \in \text{set } (ig\text{-verts } G) \implies \exists j < ig\text{-verts-cnt } G. ig\text{-verts } G ! j = v$
by (*auto simp: in-set-conv-nth*)

then show *?thesis* **using** *assms unfolding is-K5-inner-inv-def*
by *auto metis*

qed

lemma *is-K5-outer-step*:

assumes $is\text{-}K5\text{-outer-inv } G k$
assumes $is\text{-}K5\text{-inner-inv } G k (ig\text{-verts-cnt } G)$
shows $is\text{-}K5\text{-outer-inv } G (Suc k)$
using *assms unfolding is-K5-outer-inv-def*
by (*auto simp: is-K5-inner-last elim: less-SucE*)

lemma *is-K5-outer-last*:

assumes $is\text{-}K5\text{-outer-inv } G (ig\text{-verts-cnt } G)$
assumes $I\text{Graph-inv } G ig\text{-verts-cnt } G = 5$ *symmetric* ($mk\text{-graph}' G$)
shows $K_5 (mk\text{-graph}' G)$

proof –

interpret *ppg-mkg'*: *pair-fin-digraph* $mk\text{-graph}' G$
using *assms(2)* **by** (*auto intro: IGraph-imp-ppd-mkg'*)

have $\bigwedge u v. (u, v) \in \text{set } (ig\text{-arcs } G) \implies u \neq v$

using *assms(1,2) unfolding is-K5-outer-inv-def ig-verts-cnt-def*
by (*metis in-set-conv-nth set-ig-arcs-verts(2)*)

then interpret *ppg-mkg'*: *pair-graph* ($mk\text{-graph}' G$)

using *assms(4)* **by** *unfold-locales (auto simp: mkg'-simps arc-to-ends-def)*

have $\bigwedge a b. a \in p\text{verts } (mk\text{-graph}' G) \implies$

$b \in p\text{verts } (mk\text{-graph}' G) \implies a \neq b \implies (a, b) \in p\text{arcs } (mk\text{-graph}' G)$

using *assms(1) unfolding is-K5-outer-inv-def mkg'-simps*

by (*metis in-set-conv-nth ig-verts-cnt-def*)

moreover

```

have card (pverts (mk-graph' G)) = 5
  using ⟨ig-verts-cnt G = 5⟩ distinct-ig-verts by (auto simp: mkg'-simps distinct-card)
  ultimately
  show ?thesis
    unfolding complete-digraph-pair-def
    by (auto dest: ppg-mkg'.in-arcsD1 ppg-mkg'.in-arcsD2 ppg-mkg'.no-loops')
qed

```

lemma *is-K5-inner-step*:

```

assumes is-K5-inner-inv G k l
assumes k < ig-verts-cnt G
assumes k ≠ l ⟷ (ig-verts G ! k, ig-verts G ! l) ∈ set (ig-arcs G)
shows is-K5-inner-inv G k (Suc l)
using assms distinct-ig-verts unfolding is-K5-inner-inv-def
apply (auto elim: less-SucE)
by (metis (opaque-lifting, no-types) Suc-lessD less-SucE less-trans-Suc linorder-neqE-nat nth-eq-iff-index-eq)

```

lemma *iK5E*:

```

assumes K5 (mk-graph' G)
obtains ig-verts-cnt G = 5 ⟦i < ig-verts-cnt G; j < ig-verts-cnt G⟧ ⟹ i ≠ j
  ⟷ (ig-verts G ! i, ig-verts G ! j) ∈ set (ig-arcs G)
proof
  show ig-verts-cnt G = 5
    i < ig-verts-cnt G ⟹ j < ig-verts-cnt G ⟹
      (i ≠ j) = ((ig-verts G ! i, ig-verts G ! j) ∈ set (ig-arcs G))
    using assms distinct-ig-verts
  by (auto simp: complete-digraph-pair-def mkg'-simps distinct-card nth-eq-iff-index-eq)
qed

```

lemma (in *is-K5-impl*) *is-K5-spec*:

```

∀σ. Γ ⊢t {σ. IGraph-inv 'G ∧ symmetric (mk-graph' 'G)}
  'R ::= PROC is-K5('G)
  { 'G = σ G ∧ 'R = K5(mk-graph' 'G) }
apply vcg-step
apply (rewrite
  at whileAnno - (named-loop "outer-loop") - -
  in for (σ)
  to whileAnno -
  { is-K5-outer-inv 'G 'i ∧ 'i ≤ 5 ∧ IGraph-inv 'G ∧ symmetric (mk-graph'
'G) ∧ 'G = σ G ∧ ig-verts-cnt 'G = 5 }
  (MEASURE 5 - 'i)
  -
  annotate-named-loop-var)
apply (rewrite
  at whileAnno - (named-loop "inner-loop") - -
  in for (σ)

```

```

to whileAnnoFix -
  (λi. { is-K5-inner-inv 'G 'i 'j
    ∧ 'j ≤ 5 ∧ 'i < 5 ∧ IGraph-inv 'G ∧ symmetric (mk-graph' 'G) ∧ 'G =
σ G ∧ 'i = i
    ∧ ig-verts-cnt 'G = 5 ∧ 'u = ig-verts 'G ! 'i})
  (λ-. (MEASURE 5 - 'j))
  -
  annotate-named-loop-var-fix)
apply vcg
  apply (fastforce simp: is-K5-outer-inv-def intro: K5-card)
  apply (fastforce simp add: is-K5-inner-0 is-K5-outer-step)
  apply (fastforce simp: is-K5-inner-step elim: iK5E)
  apply (fastforce simp: is-K5-outer-last)
done

```

16.2.8 Soundness of the Checker

lemma *planar-theorem*:

```

assumes pair-pseudo-graph G pair-pseudo-graph K
and subgraph K G
and  $K_{3,3}$  (contr-graph K)  $\vee$   $K_5$  (contr-graph K)
shows  $\neg$ kuratowski-planar G
using assms
by (auto dest: pair-pseudo-graph.kuratowski-contr)

```

definition *witness* :: 'a pair-pre-digraph \Rightarrow 'a pair-pre-digraph \Rightarrow bool **where**

```

witness G K  $\equiv$  loop-free K  $\wedge$  pair-pseudo-graph K  $\wedge$  subgraph K G
   $\wedge$  ( $K_{3,3}$  (contr-graph K)  $\vee$   $K_5$  (contr-graph K))

```

lemma *witness* (mk-graph G) (mk-graph K) \longleftrightarrow pair-pre-digraph.certify (mk-graph G) (mk-graph K) \wedge loop-free (mk-graph K)

by (auto simp: witness-def pair-pre-digraph.certify-def Let-def wf-digraph-wp-iff wellformed-pseudo-graph-mkg)

lemma *pwd-imp-ppg-mkg*:

```

assumes pair-wf-digraph (mk-graph G)
shows pair-pseudo-graph (mk-graph G)

```

proof –

```

interpret pair-wf-digraph mk-graph G by fact
show ?thesis

```

```

apply unfold-locales

```

```

apply (auto simp: mkg-simps finite-symcl-iff)

```

```

apply (auto simp: mk-graph-def symmetric-mk-symmetric)

```

```

done

```

qed

theorem (in *check-kuratowski-impl*) *check-kuratowski-spec*:

```

 $\forall \sigma. \Gamma \vdash_t \{ \sigma. \text{pair-wf-digraph (mk-graph 'G)} \}$ 

```

```

    'R := PROC check-kuratowski('G, 'K)
  } } 'G =  $\sigma$  G  $\wedge$  'K =  $\sigma$  K  $\wedge$  'R  $\longleftrightarrow$  witness (mk-graph 'G) (mk-graph 'K) } }
  by vcg (auto simp: witness-def IGraph-inv-conv' pwd-imp-ppg-mkg)

```

```

lemma check-kuratowski-correct:
  assumes pair-pseudo-graph G
  assumes witness G K
  shows  $\neg$ kuratowski-planar G
  using assms
  by (intro planar-theorem[where K=K]) (auto simp: witness-def)

```

```

lemma check-kuratowski-correct-comb:
  assumes pair-pseudo-graph G
  assumes witness G K
  shows  $\neg$ comb-planar G
  using assms by (metis check-kuratowski-correct comb-planar-compat)

```

```

lemma check-kuratowski-complete:
  assumes pair-pseudo-graph G pair-pseudo-graph K loop-free K
  assumes subgraph K G
  assumes subdivision-pair H K  $K_{3,3}$  H  $\vee$   $K_5$  H
  shows witness G K
  using assms by (auto simp: witness-def intro: K33-contractedI K5-contractedI)

```

```

end
theory AutoCorres-Misc imports
  ../l4v/lib/OptionMonadWP
begin

```

17 Auxilliary Lemmas for Autocorres

17.1 Option monad

```

definition owhile-inv :: ('a  $\Rightarrow$  's  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\Rightarrow$  ('s,'a) lookup)  $\Rightarrow$  'a  $\Rightarrow$  ('a  $\Rightarrow$ 
's  $\Rightarrow$  bool)  $\Rightarrow$  'a rel  $\Rightarrow$  ('s,'a) lookup where
  owhile-inv c b a I R  $\equiv$  owhile c b a

```

```

lemma owhile-unfold: owhile C B r s = ocondition (C r) (B r |>> owhile C B)
(oreturn r) s
by (auto simp: ocondition-def obind-def oreturn-def owhile-def option-while-simps
split: option.split)

```

```

lemma ovalidNF-owhile:
  assumes  $\bigwedge$ s. P r s  $\Longrightarrow$  I r s
  and  $\bigwedge$ r s. ovalidNF ( $\lambda$ s'. I r s'  $\wedge$  C r s'  $\wedge$  s' = s) (B r) ( $\lambda$ r' s'. I r' s'  $\wedge$  (r',
r)  $\in$  R)
  and wf R
  and  $\bigwedge$ r s. I r s  $\Longrightarrow$   $\neg$  C r s  $\Longrightarrow$  Q r s
  shows ovalidNF (P r) (OptionMonad.owhile C B r) Q

```

```

unfolding ovalidNF-def
proof (intro allI impI)
  fix s assume P r s
  then have I r s by fact
  moreover note  $\langle \text{wf } R \rangle$ 
  moreover have  $\bigwedge r r'. I r s \implies C r s \implies B r s = \text{Some } r' \implies (r', r) \in R$ 
    using assms(2) unfolding ovalidNF-def by fastforce
  moreover have  $\bigwedge r r'. I r s \implies C r s \implies B r s = \text{Some } r' \implies I r' s$ 
    using assms(2) unfolding ovalidNF-def by blast
  moreover have  $\bigwedge r. I r s \implies C r s \implies B r s = \text{None} \implies$ 
     $\text{None} \neq \text{None} \wedge (\forall r'. \text{None} = \text{Some } r' \longrightarrow Q r' s)$ 
    using assms(2) unfolding ovalidNF-def by blast
  moreover have  $\bigwedge r. I r s \implies \neg C r s \implies \text{Some } r \neq \text{None} \wedge (\forall r'. \text{Some } r =$ 
Some } r' \longrightarrow Q r' s)
    using assms(4) unfolding ovalidNF-def by blast
  ultimately
  show  $\text{owhile } C B r s \neq \text{None} \wedge (\forall r'. \text{owhile } C B r s = \text{Some } r' \longrightarrow Q r' s)$ 
    by (rule owhile-rule[where I=I])
qed

```

```

lemma ovalidNF-owhile-inv[wp]:
  assumes  $\bigwedge r s. \text{ovalidNF } (\lambda s'. I r s' \wedge C r s' \wedge s' = s) (B r) (\lambda r' s'. I r' s' \wedge$ 
(r', r) \in R)
    and wf R
    and  $\bigwedge r s. I r s \implies \neg C r s \implies Q r s$ 
  shows ovalidNF (I r) (owhile-inv C B r I R) Q
  unfolding owhile-inv-def using - assms by (rule ovalidNF-owhile)

```

```

end
theory Setup-AutoCorres
imports
  Case-Labeling.Case-Labeling
  HOL-Eisbach.Eisbach
  AutoCorres-Misc
begin

```

18 AutoCorres setup for VCG labelling

Theorem collections for the VCG

ML-file $\langle \dots / \text{Case-Labeling} / \text{util.ML} \rangle$

```

ML  $\langle$ 
  fun vsg-tac nt-rules nt-comb ctxt =
    let
      val rules = Named-Theorems.get ctxt nt-rules
      val comb = Named-Theorems.get ctxt nt-comb

```

```

    in REPEAT-ALL-NEW-FWD ( resolve-tac ctxt rules ORELSE' (resolve-tac
    ctxt comb THEN' resolve-tac ctxt rules)) end
  >

```

```

named-theorems vcg-l
named-theorems vcg-l-comb
named-theorems vcg-elim
named-theorems vcg-simp

```

```

method-setup vcg-l = <
  Scan.succeed (fn ctxt => SIMPLE-METHOD (FIRSTGOAL (vcg-tac @ {named-theorems
  vcg-l} @ {named-theorems vcg-l-comb} ctxt)))
>

```

```

method vcg-l' = (vcg-l; (elim vcg-elim)?; (unfold vcg-simp)?)

```

```

method vcg-casify = (rule Initial-Label, vcg-l', casify)

```

18.1 Labeled VCG theorems for branching

```

definition BRANCH P ≡ P

```

```

named-theorems branch-l
named-theorems branch-l-comb

```

```

context begin
  interpretation Labeling-Syntax .

```

```

lemma DC-if[branch-l]:
  fixes ct defines ct' ≡ λpos name. (name, pos, []) # ct
  assumes a ⇒ C⟨Suc inp, ct' inp "then", outp': b⟩
  assumes ¬a ⇒ C⟨Suc outp', ct' outp' "else", outp: c⟩
  shows C⟨inp, ct, outp: BRANCH (if a then b else c)⟩
  using assms(2-) unfolding LABEL-simps BRANCH-def by auto

```

```

lemma DC-final:
  assumes V⟨("g", inp, []), ct: a⟩
  shows C⟨inp, ct, Suc inp: a⟩
  using assms unfolding LABEL-simps BRANCH-def by auto

```

```

end

```

```

method-setup branch-l = <
  Scan.succeed (fn ctxt => SIMPLE-METHOD (FIRSTGOAL (vcg-tac @ {named-theorems
  branch-l} @ {named-theorems branch-l-comb} ctxt)))
>

```

```

method branch-casify = ((rule Initial-Label, branch-l; (rule DC-final)?), casify)

```

18.2 Labelled VCG theorems for the option monad

definition

$lpred\text{-}conj :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ (**infixr** $\langle land \rangle$ 35)

where

$lpred\text{-}conj P Q \equiv \lambda x. P x \wedge Q x$

context begin

interpretation *Labeling-Syntax* .

lemma *ovalidNF-obind-K-bind* [vcg-l]:

assumes $CTXT (Suc OC1) CT OC (ovalidNF R g Q)$

and $CTXT IC CT OC1 (ovalidNF P f (\lambda-. R))$

shows $CTXT IC CT OC (ovalidNF P (f |>> K\text{-}bind g) Q)$

using *assms unfolding LABEL-simps* **by** *wp*

lemma *L-ovalidNF-obind-oreturn* [vcg-l]:

assumes $CTXT IC CT OC (ovalidNF P (g x) Q)$

shows $CTXT IC CT OC (ovalidNF P (oreturn x |>> g) Q)$

using *assms* **by** (*simp add: LABEL-simps*)

lemma *L-ovalidNF-obind* [vcg-l]:

assumes $\bigwedge r. CTXT (Suc OC1) ("bind", Suc OC1, [VAR r]) \# CT OC$
 $(ovalidNF (R r) (g r) Q)$

and $CTXT IC CT OC1 (ovalidNF P f R)$

shows $CTXT IC CT OC (ovalidNF P (f |>> (\lambda r. g r)) Q)$

using *assms unfolding LABEL-simps* **by** *wp*

lemma *ovalidNF-K-bind* [vcg-l]:

assumes $CTXT IC CT OC (ovalidNF P f Q)$

shows $CTXT IC CT OC (ovalidNF P (K\text{-}bind f x) Q)$

using *assms* **by** *simp*

lemma *L-ovalidNF-prod-case* [vcg-l]:

assumes $\bigwedge x y. SPLIT v (x, y) \Longrightarrow CTXT IC CT OC (ovalidNF (P x y) (B x y) Q)$

shows $CTXT IC CT OC (ovalidNF (case v of (x, y) \Rightarrow P x y) (case v of (x, y) \Rightarrow B x y) Q)$

using *assms unfolding LABEL-simps* **by** (*auto simp: ovalidNF-def*)

lemma *L-ovalidNF-oreturn-NF* [vcg-l]:

shows $CTXT IC CT IC (ovalidNF (P x) (oreturn x) P)$

unfolding *LABEL-simps* **by** *wp*

lemma *L-ovalidNF-owhile-inv* [vcg-l]:

fixes $CT IC$

defines $CT' \equiv \lambda r. ("while", IC, [VAR r]) \# CT$

assumes $\bigwedge r s. CTXT IC ("invariant", IC, [VAR s]) \# CT' r OC$
 $(ovalidNF$

$(BIND "loop\text{-}inv" IC (I r) land$

```

    BIND "loop-cond" IC (C r) land
    BIND "loop-var" IC ( $\lambda s'. s' = s$ )
  (B r)
  ( $\lambda r'. \text{BIND "inv" IC (I r')} \text{ land } \text{BIND "var" IC } (\lambda-. (r', r) \in R)$ )
  and  $\bigwedge r. \text{VC ("wf", OC, [])} (CT' r) (wf R)$ 
  and  $\bigwedge r s. I r s \implies \neg C r s \implies$ 
    VC ("postcondition", Suc OC, [VAR s]) (CT' r) (Q r s)
  shows CTXT IC CT (Suc OC) (ovalidNF (I r) (owhile-inv C B r I R) Q)
  using assms unfolding LABEL-simps lpred-conj-def by wp auto

```

```

lemma L-ovalidNF-wp-comb2[vcg-l-comb]:
  assumes CTXT IC CT OC (ovalidNF P f Q)
  and  $\bigwedge s. P' s \implies \text{VC ("weaken", IC, [VAR s]) CT (P s)}$ 
  shows CTXT IC CT OC (ovalidNF P' f Q)
  using assms unfolding LABEL-simps by (rule ovalidNF-wp-comb2)

```

```

lemma L-condition-NF-wp[vcg-l]:
  fixes CT IC
  defines CT'  $\equiv$  ("if", IC, []) # CT
  assumes CTXT IC (("then", IC, []) # CT') OC1 (ovalidNF L l Q)
  and CTXT (Suc OC1) (("else", Suc OC1, []) # CT') OC (ovalidNF R r Q)
  shows CTXT IC CT OC (ovalidNF ( $\lambda s. \text{BRANCH (if C s then L s else R s)}$ ))
  (ocondition C l r) Q)
  using assms unfolding LABEL-simps BRANCH-def by wp

```

```

lemma L-ogets-NF-wp[vcg-l]: CTXT IC CT IC (ovalidNF ( $\lambda s. P (f s) s$ ) (ogets
f) P)
  unfolding LABEL-simps by wp

```

```

lemma elim-land[vcg-elim]:
  assumes (P land Q) s obtains P s Q s
  using assms by (auto simp: lpred-conj-def)

```

```

lemma simp-bind[vcg-simp]: BIND ct n P s  $\longleftrightarrow$  BIND ct n (P s)
  by (auto simp: LABEL-simps)

```

```

lemma simp-land[vcg-simp]: (P land Q) s  $\longleftrightarrow$  P s  $\wedge$  Q s
  by (auto simp: lpred-conj-def)

```

end

end

19 Verification of a Planarity Checker

```

theory Check-Planarity-Verification
imports
  ../Planarity/Graph-Genus
  Setup-AutoCorres
  HOL-Library.Rewrite

```

begin

19.1 Implementation Types

type-synonym $IVert = nat$

type-synonym $IEdge = IVert \times IVert$

type-synonym $IGraph = IVert\ list \times IEdge\ list$

abbreviation (*input*) $ig\text{-edges} :: IGraph \Rightarrow IEdge\ list$ **where**
 $ig\text{-edges}\ G \equiv snd\ G$

abbreviation (*input*) $ig\text{-verts} :: IGraph \Rightarrow IVert\ list$ **where**
 $ig\text{-verts}\ G \equiv fst\ G$

definition $ig\text{-tail} :: IGraph \Rightarrow nat \Rightarrow IVert$ **where**
 $ig\text{-tail}\ IG\ a = fst\ (ig\text{-edges}\ IG\ !\ a)$

definition $ig\text{-head} :: IGraph \Rightarrow nat \Rightarrow IVert$ **where**
 $ig\text{-head}\ IG\ a = snd\ (ig\text{-edges}\ IG\ !\ a)$

type-synonym $IMap = (nat \Rightarrow nat) \times (nat \Rightarrow nat) \times (nat \Rightarrow nat)$

definition $im\text{-rev} :: IMap \Rightarrow (nat \Rightarrow nat)$ **where**
 $im\text{-rev}\ iM = fst\ iM$

definition $im\text{-succ} :: IMap \Rightarrow (nat \Rightarrow nat)$ **where**
 $im\text{-succ}\ iM = fst\ (snd\ iM)$

definition $im\text{-pred} :: IMap \Rightarrow (nat \Rightarrow nat)$ **where**
 $im\text{-pred}\ iM = snd\ (snd\ iM)$

definition $mk\text{-graph} :: IGraph \Rightarrow (IVert, nat)$ *pre-digraph* **where**
 $mk\text{-graph}\ IG \equiv (\$
 $verts = set\ (ig\text{-verts}\ IG),$
 $arcs = \{0..<\ length\ (ig\text{-edges}\ IG)\},$
 $tail = ig\text{-tail}\ IG,$
 $head = ig\text{-head}\ IG$
 $\)$

lemma $mkg\text{-simps}$:

$verts\ (mk\text{-graph}\ IG) = set\ (ig\text{-verts}\ IG)$

$tail\ (mk\text{-graph}\ IG) = ig\text{-tail}\ IG$

$head\ (mk\text{-graph}\ IG) = ig\text{-head}\ IG$

by (*auto simp: mk-graph-def*)

lemma $arcs\text{-mkg}$: $arcs\ (mk\text{-graph}\ IG) = \{0..<\ length\ (ig\text{-edges}\ IG)\}$
by (*auto simp: mk-graph-def*)

lemma *arc-to-ends-mkg*: $\text{arc-to-ends } (mk\text{-graph } iG) \ a = \text{ig-edges } iG \ ! \ a$
by (*auto simp: arc-to-ends-def mkg-simps ig-tail-def ig-head-def*)

definition *mk-map* :: $(-, \text{nat}) \text{ pre-digraph} \Rightarrow \text{IMap} \Rightarrow \text{nat pre-map}$ **where**
mk-map $G \ iM \equiv \langle$
 $\text{edge-rev} = \text{perm-restrict } (im\text{-rev } iM) \ (\text{arcs } G),$
 $\text{edge-succ} = \text{perm-restrict } (im\text{-succ } iM) \ (\text{arcs } G)$
 \rangle

lemma *mkm-simps*:
 $\text{edge-rev } (mk\text{-map } G \ iM) = \text{perm-restrict } (im\text{-rev } iM) \ (\text{arcs } G)$
 $\text{edge-succ } (mk\text{-map } G \ iM) = \text{perm-restrict } (im\text{-succ } iM) \ (\text{arcs } G)$
by (*auto simp: mk-map-def*)

lemma *es-eq-im*: $a \in \text{arcs } (mk\text{-graph } iG) \implies \text{edge-succ } (mk\text{-map } (mk\text{-graph } iG) \ iM) \ a = im\text{-succ } iM \ a$
by (*auto simp: mkm-simps arcs-mkg perm-restrict-simps*)

19.2 Implementation

definition *is-map* $iG \ iM \equiv$
 $DO \ ecnt \leftarrow \text{oreturn } (\text{length } (\text{snd } iG));$
 $vcnt \leftarrow \text{oreturn } (\text{length } (\text{fst } iG));$
 $(i, \text{revOk}) \leftarrow \text{owhile}$
 $(\lambda(i, ok) \ s. \ i < ecnt \wedge ok)$
 $(\lambda(i, ok).$
 $\quad DO$
 $\quad \quad j \leftarrow \text{oreturn } (im\text{-rev } iM \ i);$
 $\quad \quad \text{revIn} \leftarrow \text{oreturn } (j < \text{length } (ig\text{-edges } iG));$
 $\quad \quad \text{revNeg} \leftarrow \text{oreturn } (j \neq i);$
 $\quad \quad \text{revRevs} \leftarrow \text{oreturn } (ig\text{-edges } iG \ ! \ j = \text{prod.swap } (ig\text{-edges } iG \ ! \ i));$
 $\quad \quad \text{invol} \leftarrow \text{oreturn } (im\text{-rev } iM \ j = i);$
 $\quad \quad \text{oreturn } (i + 1, \text{revIn} \wedge \text{revNeg} \wedge \text{revRevs} \wedge \text{invol})$
 $\quad OD)$
 $(0, \text{True});$
 $(i, \text{succPerm}) \leftarrow \text{owhile}$
 $(\lambda(i, ok) \ s. \ i < ecnt \wedge ok)$
 $(\lambda(i, ok).$
 $\quad DO$
 $\quad \quad j \leftarrow \text{oreturn } (im\text{-succ } iM \ i);$
 $\quad \quad \text{succIn} \leftarrow \text{oreturn } (j < \text{length } (ig\text{-edges } iG));$
 $\quad \quad \text{succEnd} \leftarrow \text{oreturn } (ig\text{-tail } iG \ i = ig\text{-tail } iG \ j);$
 $\quad \quad \text{isPerm} \leftarrow \text{oreturn } (im\text{-pred } iM \ j = i);$
 $\quad \quad \text{oreturn } (i + 1, \text{succIn} \wedge \text{succEnd} \wedge \text{isPerm})$
 $\quad OD)$
 $(0, \text{True});$
 $(i, \text{succOrbits}, V, A) \leftarrow \text{owhile}$
 $(\lambda(i, ok, V, A) \ s. \ i < ecnt \wedge \text{succPerm} \wedge ok)$
 $(\lambda(i, ok, V, A).$

```

DO
(x, V, A) ← ocondition (λ-. ig-tail iG i ∈ V)
(oreturn (i ∈ A, V, A))
(DO
(A', j) ← owhile
(λ(A', j) s. j ∉ A')
(λ(A', j). DO
A' ← oreturn (insert j A');
j ← oreturn (im-succ iM j);
oreturn (A', j)
OD)
({}, i);
V ← oreturn (insert (ig-tail iG j) V);
oreturn (True, V, A ∪ A')
OD);
oreturn (i + 1, x, V, A)
OD)
(0, True, {}, {});
oreturn (revOk ∧ succPerm ∧ succOrbits)
OD

```

definition *isolated-nodes* :: IGraph ⇒ - ⇒ nat option where
isolated-nodes iG ≡

```

DO ecnt ← oreturn (length (snd iG));
vcnt ← oreturn (length (fst iG));
(i, nz) ←
owhile
(λ(i, nz) a. i < vcnt)
(λ(i, nz).
DO v ← oreturn (fst iG ! i);
j ← oreturn 0;
ret ← ocondition (λs. j < ecnt) (oreturn (ig-tail iG j ≠ v)) (oreturn
False);
ret ← ocondition (λs. ret) (oreturn (ig-head iG j ≠ v)) (oreturn ret);
(j, -) ←
owhile
(λ(j, cond) a. cond)
(λ(j, cond).
DO j ← oreturn (j + 1);
cond ← ocondition (λs. j < ecnt) (oreturn (ig-tail iG j ≠ v))
(oreturn False);
cond ← ocondition (λs. cond) (oreturn (ig-head iG j ≠ v)) (oreturn
cond);
oreturn (j, cond)
OD)
(j, ret);
nz ← oreturn (if j = ecnt then nz + 1 else nz);
oreturn (i + 1, nz)

```

```

    OD)
  (0, 0);
  oreturn nz
OD

```

definition *face-cycles* :: *IGraph* \Rightarrow *nat pre-map* \Rightarrow - \Rightarrow *nat option* **where**
face-cycles *iG* *iM* \equiv

```

  DO ecnt  $\leftarrow$  oreturn (length (snd iG));
  (edge-info, c, i)  $\leftarrow$ 
  owhile
  ( $\lambda$ (edge-info, c, i) s. i < ecnt)
  ( $\lambda$ (edge-info, c, i).
  DO (edge-info, c)  $\leftarrow$ 
  ocondition ( $\lambda$ s. i  $\notin$  edge-info)
  (DO j  $\leftarrow$  oreturn i;
  edge-info  $\leftarrow$  oreturn (insert j edge-info);
  ret'  $\leftarrow$  oreturn (pre-digraph-map.face-cycle-succ iM j);
  (edge-info, j)  $\leftarrow$ 
  owhile
  ( $\lambda$ (edge-info, j) s. i  $\neq$  j)
  ( $\lambda$ (edge-info, j).
  oreturn (insert j edge-info, pre-digraph-map.face-cycle-succ iM
j)))
  (edge-info, ret');
  oreturn (edge-info, c + 1)
  OD)
  (oreturn (edge-info, c));
  oreturn (edge-info, c, i + 1)
  OD)
  ({} , 0, 0);
  oreturn c
OD

```

definition *euler-genus* *iG* *iM* *c* \equiv

```

  DO n  $\leftarrow$  oreturn (length (ig-edges iG));
  m  $\leftarrow$  oreturn (length (ig-verts iG));
  nz  $\leftarrow$  isolated-nodes iG;
  fc  $\leftarrow$  face-cycles iG iM;
  oreturn ((int n div 2 + 2 * int c - int m - int nz - int fc) div 2)
OD

```

definition *certify* *iG* *iM* *c* \equiv

```

  DO
  map  $\leftarrow$  is-map iG iM;
  ocondition ( $\lambda$ -. map)
  (DO
  gen  $\leftarrow$  euler-genus iG (mk-map (mk-graph iG) iM) c;
  oreturn (gen = 0)
  OD)

```

(oreturn False)
 OD

19.3 Verification

context begin

interpretation Labeling-Syntax .

lemma trivial-label: $P \implies CTXT IC CT OC P$

unfolding LABEL-simps .

end

lemma ovalidNF-wp:

assumes ovalidNF $P c (\lambda r s. r = x)$

shows ovalidNF $(\lambda s. Q x s \wedge P s) c Q$

using assms unfolding ovalidNF-def by auto

19.3.1 is-map

definition is-map-rev-ok-inv $iG iM k ok \equiv ok \longleftrightarrow (\forall i < k.$

$im\text{-}rev\ iM\ i < length\ (ig\text{-}edges\ iG)$

$\wedge ig\text{-}edges\ iG\ !\ im\text{-}rev\ iM\ i = prod.swap\ (ig\text{-}edges\ iG\ !\ i)$

$\wedge im\text{-}rev\ iM\ i \neq i$

$\wedge im\text{-}rev\ iM\ (im\text{-}rev\ iM\ i) = i)$

definition is-map-succ-perm-inv $iG iM k ok \equiv ok \longleftrightarrow (\forall i < k.$

$im\text{-}succ\ iM\ i < length\ (ig\text{-}edges\ iG)$

$\wedge ig\text{-}tail\ iG\ (im\text{-}succ\ iM\ i) = ig\text{-}tail\ iG\ i$

$\wedge im\text{-}pred\ iM\ (im\text{-}succ\ iM\ i) = i)$

definition is-map-succ-orbits-inv $iG iM k ok V A \equiv$

$A = (\bigcup i < (if\ ok\ then\ k\ else\ k - 1). orbit\ (im\text{-}succ\ iM)\ i) \wedge$

$V = \{ig\text{-}tail\ iG\ i \mid i. i < (if\ ok\ then\ k\ else\ k - 1)\} \wedge$

$ok = (\forall i < k. \forall j < k. ig\text{-}tail\ iG\ i = ig\text{-}tail\ iG\ j \longrightarrow j \in orbit\ (im\text{-}succ\ iM)\ i)$

definition is-map-succ-orbits-inner-inv $iG iM i j A' \equiv$

$A' = (if\ i = j \wedge i \notin A' \ then\ \{\} \ else\ \{i\} \cup segment\ (im\text{-}succ\ iM)\ i\ j)$

$\wedge j \in orbit\ (im\text{-}succ\ iM)\ i$

definition is-map-final $iG k ok \equiv (ok \longrightarrow k = length\ (ig\text{-}edges\ iG)) \wedge k \leq length\ (ig\text{-}edges\ iG)$

lemma bij-betwI-finite-dom:

assumes finite $A f \in A \rightarrow A \wedge a. a \in A \implies g\ (f\ a) = a$

shows bij-betw $f\ A\ A$

proof –

have inj-on $f\ A$ by (metis assms(3) inj-onI)

moreover
then have $f \text{ ' } A = A$ **by** (*metis Pi-iff assms(1-2) endo-inj-surj image-subsetI*)
ultimately show *?thesis unfolding bij-betw-def by simp*
qed

lemma *permutesI-finite-dom*:
assumes *finite A*
assumes $f \in A \rightarrow A$
assumes $\bigwedge a. a \notin A \implies f a = a$
assumes $\bigwedge a. a \in A \implies g (f a) = a$
shows *f permutes A*
using *assms by (intro bij-imp-permutes bij-betwI-finite-dom)*

lemma *orbit-ss*:
assumes $f \in A \rightarrow A$ $a \in A$
shows $\text{orbit } f a \subseteq A$
proof –
{ fix x assume $x \in \text{orbit } f a$ **then have** $x \in A$ **using** *assms by induct auto* **}**
then show *?thesis by blast*
qed

lemma *segment-eq-orbit*:
assumes $y \notin \text{orbit } f x$ **shows** $\text{segment } f x y = \text{orbit } f x$
proof (*intro set-eqI iffI*)
fix z assume $z \in \text{segment } f x y$ **then show** $z \in \text{orbit } f x$ **by** (*rule segmentD-orbit*)
next
fix z assume $z \in \text{orbit } f x$ **then show** $z \in \text{segment } f x y$
using *assms by induct (auto intro: segment.intros orbit-eqI elim: orbit.cases)*
qed

lemma *funpow-in-funcset*:
assumes $x \in A$ $f \in A \rightarrow A$ **shows** $(f \text{ ^^ } n) x \in A$
using *assms by (induct n) auto*

lemma *funpow-eq-funcset*:
assumes $x \in A$ $f \in A \rightarrow A$ $\bigwedge y. y \in A \implies f y = g y$
shows $(f \text{ ^^ } n) x = (g \text{ ^^ } n) x$
using *assms by (induct n) (auto, metis funpow-in-funcset)*

lemma *funpow-dist1-eq-funcset*:
assumes $y \in \text{orbit } f x$ $x \in A$ $f \in A \rightarrow A$ $\bigwedge y. y \in A \implies f y = g y$
shows $\text{funpow-dist1 } f x y = \text{funpow-dist1 } g x y$
proof –
have $y = (f \text{ ^^ } \text{funpow-dist1 } f x y) x$ **by** (*metis assms(1) funpow-dist1-prop*)
also have $\dots = (g \text{ ^^ } \text{funpow-dist1 } f x y) x$ **by** (*metis assms(2-) funpow-eq-funcset*)
finally have $*$: $y = (g \text{ ^^ } \text{funpow-dist1 } f x y) x$.
then have $(g \text{ ^^ } \text{funpow-dist1 } g x y) x = y$ **by** (*metis funpow-dist1-prop1 zero-less-Suc*)
with $*$ **have** $gf: \text{funpow-dist1 } g x y \leq \text{funpow-dist1 } f x y$

```

    by (metis funpow-dist1-least not-le zero-less-Suc)

  have (f  $\sim$  funpow-dist1 g x y) x = y
  using ⟨(g  $\sim$  funpow-dist1 g x y) x = y⟩ by (metis assms(2-) funpow-eq-funcset)
  then have fg: funpow-dist1 f x y  $\leq$  funpow-dist1 g x y
  using ⟨y = (f  $\sim$  -) x⟩ by (metis funpow-dist1-least not-le zero-less-Suc)

  from gf fg show ?thesis by simp
qed

lemma segment-cong0:
  assumes x  $\in$  A f  $\in$  A  $\rightarrow$  A  $\wedge$  y. y  $\in$  A  $\implies$  f y = g y shows segment f x y =
  segment g x y
  proof (cases y  $\in$  orbit f x)
    case True
      moreover
      from assms have orbit f x = orbit g x by (rule orbit-cong0)
      moreover
      have (f  $\sim$  n) x = (g  $\sim$  n) x  $\wedge$  (f  $\sim$  n) x  $\in$  A for n
        by (induct n rule: nat.induct) (insert assms, auto)
      ultimately show ?thesis
        using True by (auto simp: segment-altdef funpow-dist1-eq-funcset[OF - assms])
    next
      case False
      moreover from assms have orbit f x = orbit g x by (rule orbit-cong0)
      ultimately show ?thesis by (simp add: segment-eq-orbit)
  qed

lemma rev-ok-final:
  assumes wf-iG: wf-digraph (mk-graph iG)
  assumes rev: is-map-rev-ok-inv iG iM rev-i rev-ok is-map-final iG rev-i rev-ok
  shows rev-ok  $\longleftrightarrow$  bidirected-digraph (mk-graph iG) (edge-rev (mk-map (mk-graph
  iG) iM)) (is ?L  $\longleftrightarrow$  ?R)
  proof
    assume rev-ok
    interpret wf-digraph mk-graph iG by (rule wf-iG)
    have rev-inv-sep:
       $\wedge$  i. i < length (ig-edges iG)  $\implies$  im-rev iM i < length (ig-edges iG)
       $\wedge$  i. i < length (ig-edges iG)  $\implies$  ig-edges iG ! im-rev iM i = prod.swap
      (ig-edges iG ! i)
       $\wedge$  i. i < length (ig-edges iG)  $\implies$  im-rev iM i  $\neq$  i
       $\wedge$  i. i < length (ig-edges iG)  $\implies$  im-rev iM (im-rev iM i) = i
    using rev ⟨rev-ok⟩ by (auto simp: is-map-rev-ok-inv-def is-map-final-def)
    moreover
    { fix i assume i < length (ig-edges iG)
      then have ig-tail iG (im-rev iM i) = ig-head iG i
        using rev-inv-sep(2) by (cases ig-edges iG ! i) (auto simp: ig-head-def
        ig-tail-def)
      }
  }

```

ultimately show $?R$
using wf **by** $unfold-locales (auto simp: mkg-simps arcs-mkg mkm-simps perm-restrict-def)$
next
assume $?R$
let $?rev = perm-restrict (im-rev iM) (arcs (mk-graph iG))$
interpret $bidirected-digraph mk-graph iG perm-restrict (im-rev iM) (arcs (mk-graph iG))$
using $\langle ?R \rangle$ **by** $(simp add: mkm-simps mkg-simps)$
have $\bigwedge a. a \in arcs (mk-graph iG) \implies ?rev a \in arcs (mk-graph iG)$
 $\bigwedge a. a \in arcs (mk-graph iG) \implies$
 $arc-to-ends (mk-graph iG) (?rev a) = prod.swap (arc-to-ends (mk-graph iG)$
 $a)$
 $\bigwedge a. a \in arcs (mk-graph iG) \implies ?rev a \neq a$
 $\bigwedge a. a \in arcs (mk-graph iG) \implies ?rev (?rev a) = a$
by $(auto simp: arev-dom)$
then show $rev-ok$
using rev **unfolding** $is-map-rev-ok-inv-def is-map-final-def$
by $(simp add: perm-restrict-simps arcs-mkg arc-to-ends-mkg)$
qed

locale $is-map-postcondition0 =$
fixes $iG iM rev-ok succ-i succ-ok$
assumes $succ-perm: is-map-succ-perm-inv iG iM succ-i succ-ok is-map-final iG$
 $succ-i succ-ok$
begin

lemma $succ-ok-tail-eq:$
 $succ-ok \implies i < length (ig-edges iG) \implies ig-tail iG (im-succ iM i) = ig-tail iG$
 i
using $succ-perm$ **unfolding** $is-map-succ-perm-inv-def is-map-final-def$ **by** $auto$

lemma $succ-ok-imp-pred:$
 $succ-ok \implies i < length (ig-edges iG) \implies im-pred iM (im-succ iM i) = i$
using $succ-perm$ **unfolding** $is-map-succ-perm-inv-def is-map-final-def$ **by** $auto$

lemma $succ-ok-imp-permutes:$
assumes $succ-ok$
shows $edge-succ (mk-map (mk-graph iG) iM)$ $permutes arcs (mk-graph iG)$
proof –
from $assms$ **have** $\forall a \in arcs (mk-graph iG). edge-succ (mk-map (mk-graph iG)$
 $iM) a \in arcs (mk-graph iG)$
using $succ-perm$ **unfolding** $is-map-succ-perm-inv-def is-map-final-def$
by $(auto simp: mkg-simps mkm-simps arcs-mkg perm-restrict-def)$
with $succ-ok-imp-pred[OF assms]$ **show** $?thesis$
by – $(rule permutesI-finite-dom[where g=im-pred iM], auto simp: perm-restrict-simps$
 $mkm-simps arcs-mkg)$
qed

lemma $es-A2A: succ-ok \implies edge-succ (mk-map (mk-graph iG) iM) \in arcs$

```

(mk-graph iG) → arcs (mk-graph iG)
  using succ-ok-imp-permutes by (auto dest: permutes-in-image)

lemma im-succ-le-length: succ-ok ⇒ i < length (ig-edges iG) ⇒ im-succ iM i
< length (ig-edges iG)
  using is-map-final-def is-map-succ-perm-inv-def succ-perm(1) succ-perm(2) by
auto

lemma orbit-es-eq-im:
  succ-ok ⇒ a ∈ arcs (mk-graph iG) ⇒ orbit (edge-succ (mk-map (mk-graph
iG) iM)) a = orbit (im-succ iM) a
  using - es-A2A es-eq-im by (rule orbit-cong0)

lemma segment-es-eq-im:
  succ-ok ⇒ a ∈ arcs (mk-graph iG) ⇒ segment (edge-succ (mk-map (mk-graph
iG) iM)) a b = segment (im-succ iM) a b
  using - es-A2A es-eq-im by (rule segment-cong0)

lemma in-orbit-im-succE:
  assumes j ∈ orbit (im-succ iM) i succ-ok i < length (ig-edges iG)
  obtains ig-tail iG j = ig-tail iG i j < length (ig-edges iG)
  using assms es-A2A by induct (force simp add: succ-ok-tail-eq es-eq-im arcs-mkg)+

lemma self-in-orbit-im-succ:
  assumes succ-ok i < length (ig-edges iG) shows i ∈ orbit (im-succ iM) i
proof -
  have i ∈ orbit (edge-succ (mk-map (mk-graph iG) iM)) i
  using assms succ-ok-imp-permutes
  by (intro permutation-self-in-orbit) (auto simp: permutation-permutes arcs-mkg)
  with assms show ?thesis by (simp add: orbit-es-eq-im arcs-mkg)
qed

end

locale is-map-postcondition = is-map-postcondition0 +
  fixes so-i so-ok V A
  assumes rev: rev-ok ⟷ bidirected-digraph (mk-graph iG) (edge-rev (mk-map
(mk-graph iG) iM))
  assumes succ-orbits: is-map-succ-orbits-inv iG iM so-i so-ok V A succ-ok ⟶
is-map-final iG so-i so-ok
begin

lemma ok-imp-digraph:
  assumes rev-ok succ-ok so-ok
  shows digraph-map (mk-graph iG) (mk-map (mk-graph iG) iM)
proof -
  interpret bidirected-digraph mk-graph iG edge-rev (mk-map (mk-graph iG) iM)
  using ⟨rev-ok⟩ by (simp add: rev)

```

```

from ⟨succ-ok⟩ have perm: edge-succ (mk-map (mk-graph iG) iM) permutes
arcs (mk-graph iG)
by (simp add: succ-ok-imp-permutes)
from ⟨succ-ok⟩ have ig-tail:  $\bigwedge a. a \in \text{arcs } (mk\text{-graph } iG) \implies \text{ig-tail } iG (im\text{-succ } iM a) = \text{ig-tail } iG a$ 
by (simp-all add: succ-ok-tail-eq arcs-mkg)

{ fix v assume v ∈ verts (mk-graph iG) out-arcs (mk-graph iG) v ≠ {}
then obtain a where a: a ∈ arcs (mk-graph iG) tail (mk-graph iG) a = v
by auto metis
then have out-arcs (mk-graph iG) v = {b ∈ arcs (mk-graph iG). ig-tail iG a
= ig-tail iG b}
by (auto simp: mkg-simps)
also have ... ⊆ orbit (im-succ iM) a
proof –
have (∀ i < length (snd iG). ∀ j < length (snd iG).
ig-tail iG i = ig-tail iG j → j ∈ orbit (im-succ iM) i)
using ⟨succ-ok⟩ ⟨so-ok⟩ succ-orbits unfolding is-map-succ-orbits-inv-def
is-map-final-def by metis
with a show ?thesis by (auto simp: arcs-mkg)
qed
finally have out-arcs (mk-graph iG) v ⊆ orbit (im-succ iM) a .
moreover
have orbit (im-succ iM) a ⊆ out-arcs (mk-graph iG) v
proof –
{ fix x assume x ∈ orbit (im-succ iM) a then have tail (mk-graph iG) x
= v
using a ig-tail
apply induct
apply (auto simp: mkg-simps intro: orbit.intros)
by (metis ⟨succ-ok⟩ contra-subsetD orbit-es-eq-im permutes-orbit-subset
perm)
} moreover
have orbit (im-succ iM) a ⊆ arcs (mk-graph iG)
using - a(1) apply (rule orbit-ss)
using assms arcs-mkg is-map-final-def is-map-succ-perm-inv-def succ-perm(1)
succ-perm(2) by auto
ultimately
show ?thesis by auto
qed
ultimately
have out-arcs (mk-graph iG) v = orbit (edge-succ (mk-map (mk-graph iG)
iM)) a
using ⟨succ-ok⟩ a by (auto simp: orbit-es-eq-im)
then
have cyclic-on (edge-succ (mk-map (mk-graph iG) iM)) (out-arcs (mk-graph
iG) v)
unfolding cyclic-on-def using a by force

```

```

}
with perm show ?thesis
  using ⟨rev-ok⟩ by unfold-locales (auto simp: mkg-simps arcs-mkg)
qed

lemma digraph-imp-ok:
  assumes dm: digraph-map (mk-graph iG) (mk-map (mk-graph iG) iM)
  assumes pred:  $\bigwedge i. i < \text{length } (ig\text{-edges } iG) \implies im\text{-pred } iM (im\text{-succ } iM i) = i$ 
  obtains rev-ok succ-ok so-ok
proof
  interpret dm: digraph-map mk-graph iG mk-map (mk-graph iG) iM by (fact dm)

  show rev-ok unfolding rev by unfold-locales

  show succ-ok
  proof -
    { fix i assume i ∈ arcs (mk-graph iG)
      then have
        edge-succ (mk-map (mk-graph iG) iM) i ∈ arcs (mk-graph iG)
        tail (mk-graph iG) (edge-succ (mk-map (mk-graph iG) iM) i) = tail
(mk-graph iG) i
        by auto
      then have
        im-succ iM i < length (snd iG)
        ig-tail iG (im-succ iM i) = ig-tail iG i
        unfolding es-eq-im[OF ⟨i ∈ arcs ⟩] by (auto simp: arcs-mkg mkg-simps)
    }
    then have (∀ i < length (ig-edges iG).
      im-succ iM i < length (snd iG) ∧
      ig-tail iG (im-succ iM i) = ig-tail iG i ∧ im-pred iM (im-succ iM i) = i)
      using pred by (auto simp: arcs-mkg es-eq-im)
    with succ-perm show ?thesis
    unfolding is-map-succ-perm-inv-def is-map-final-def by simp
  qed

  show so-ok
  proof -
    { fix i j assume i < length (ig-edges iG) j < length (ig-edges iG) ig-tail iG
      i = ig-tail iG j
      then have A: i ∈ arcs (mk-graph iG) j ∈ arcs (mk-graph iG) tail (mk-graph
      iG) i = tail (mk-graph iG) j
        by (auto simp: mkg-simps arcs-mkg)
      then have cyclic-on (edge-succ (mk-map (mk-graph iG) iM)) (out-arcs
      (mk-graph iG) (tail (mk-graph iG) i))
        by (auto intro!: dm.edge-succ-cyclic)
      then have orbit (edge-succ (mk-map (mk-graph iG) iM)) i = out-arcs
      (mk-graph iG) (ig-tail iG i)
        by (simp add: ⟨i ∈ arcs (mk-graph iG)⟩ mkg-simps orbit-cyclic-eq3)
    }
  }

```

```

    then have  $j \in \text{orbit } (\text{edge-succ } (\text{mk-map } (\text{mk-graph } iG) iM)) i$  using  $A$  by
    (simp add: mkg-simps)
    also have  $\text{orbit } (\text{edge-succ } (\text{mk-map } (\text{mk-graph } iG) iM)) i = \text{orbit } (\text{im-succ } iM) i$ 
      using  $\langle i \in \text{arcs } \rightarrow \rangle$ 
      by (rule orbit-cong0) (fastforce, simp add: es-eq-im)
    finally have  $j \in \text{orbit } (\text{im-succ } iM) i$  .
  }
  then show ?thesis
    using succ-orbits unfolding is-map-succ-orbits-inv-def is-map-final-def
    by safe (simp-all only:  $\langle \text{succ-ok} \rangle$  simp-thms)
qed
qed

```

end

```

lemma all-less-Suc-eq:  $(\forall x < \text{Suc } n. P x) \longleftrightarrow (\forall x < n. P x) \wedge P n$ 
  by (auto elim: less-SucE)

```

lemma in-orbit-imp-in-segment:

```

  assumes  $y \in \text{orbit } f x$   $x \neq y$  bij  $f$  shows  $y \in \text{segment } f x (f y)$ 
  using assms
proof induct
  case base then show ?case by (auto intro: segment.intros simp: bij-iff)
next
  case (step y)
  show ?case
  proof (cases  $x = y$ )
    case True then show ?thesis using step by (auto intro: segment.intros simp:
    bij-iff)
  next
    case False
    with step have  $f y \neq f (f y)$  by (metis bij-is-inj inv-f-f not-in-segment2)
    then show ?thesis using step False
    by (auto intro: segment.intros segment-step-2 bij-is-inj)
  qed
qed
qed

```

lemma ovalidNF-is-map:

```

  ovalidNF  $(\lambda s. \text{distinct } (\text{ig-verts } iG) \wedge \text{wf-digraph } (\text{mk-graph } iG))$ 
  (is-map  $iG iM$ )
   $(\lambda r s. r \longleftrightarrow \text{digraph-map } (\text{mk-graph } iG) (\text{mk-map } (\text{mk-graph } iG) iM) \wedge (\forall i <$ 
  length  $(\text{ig-edges } iG). \text{im-pred } iM (\text{im-succ } iM i) = i)$ )

```

unfolding is-map-def

```

apply (rewrite
  in oreturn (length (ig-edges iG)) |>> (lecnt.  $\sqcap$ )
  to owhile-inv - - -

```

```

    (λ(i, ok) s. is-map-rev-ok-inv iG iM i ok
     ∧ i ≤ ecnt ∧ wf-digraph (mk-graph iG))
    (measure (λ(i, ok). ecnt - i))
    owhile-inv-def[symmetric] )
apply (rewrite
  in owhile-inv - - - - |>> (λ(rev-i, rev-ok). ⊞)
  in oreturn (length (ig-edges iG)) |>> (λecnt. ⊞)
  to owhile-inv - - -
    (λ(i, ok) s. is-map-succ-perm-inv iG iM i ok
     ∧ rev-ok = bidirected-digraph (mk-graph iG) (edge-rev (mk-map (mk-graph
iG) iM))
     ∧ i ≤ ecnt ∧ wf-digraph (mk-graph iG))
    (measure (λ(i, ok). ecnt - i))
    owhile-inv-def[symmetric] )
apply (rewrite
  in owhile-inv - - - - |>> (λ(succ-i, succ-ok). ⊞)
  in owhile-inv - - - - |>> (λ(rev-i, rev-ok). ⊞)
  in oreturn (length (ig-edges iG)) |>> (λecnt. ⊞)
  to owhile-inv - - -
    (λ(i, ok, V, A) s. is-map-succ-orbits-inv iG iM i ok V A
     ∧ rev-ok = bidirected-digraph (mk-graph iG) (edge-rev (mk-map (mk-graph
iG) iM))
     ∧ is-map-succ-perm-inv iG iM succ-i succ-ok ∧ is-map-final iG succ-i succ-ok
     ∧ i ≤ ecnt ∧ wf-digraph (mk-graph iG))
    (measure (λ(i, ok, V, A). ecnt - i))
    owhile-inv-def[symmetric] )
apply (rewrite
  in owhile-inv - (λ(i, ok, V, A). ⊞) - - -
  in owhile-inv - - - - |>> (λ(succ-i, succ-ok). ⊞)
  in owhile-inv - - - - |>> (λ(rev-i, rev-ok). ⊞)
  in oreturn (length (ig-edges iG)) |>> (λecnt. ⊞)
  to owhile-inv - - -
    (λ(A', j) s. is-map-succ-orbits-inner-inv iG iM i j A'
     ∧ ig-tail iG i ∉ V ∧ succ-ok ∧ ok ∧ is-map-succ-orbits-inv iG iM i ok V A
     ∧ rev-ok = bidirected-digraph (mk-graph iG) (edge-rev (mk-map (mk-graph
iG) iM))
     ∧ is-map-succ-perm-inv iG iM succ-i succ-ok ∧ is-map-final iG succ-i succ-ok
     ∧ i < ecnt ∧ wf-digraph (mk-graph iG))
    (measure (λ(A', j). length (ig-edges iG) - card A'))
    owhile-inv-def[symmetric] )
proof vcg-casify
  let ?es = edge-succ (mk-map (mk-graph iG) iM)

  { case weaken then show ?case by (auto simp: is-map-rev-ok-inv-def)
  }
  { case (while i ok)
    { case invariant
      case weaken then show ?case by (auto simp: is-map-rev-ok-inv-def elim:
less-SucE)
    }
  }

```

```

}
{ case wf show ?case by auto
}
{ case postcondition
  then have ok  $\longleftrightarrow$  bidirected-digraph (mk-graph iG) (edge-rev (mk-map
(mk-graph iG) iM))
  by (intro rev-ok-final) (auto simp: is-map-final-def)
  with postcondition show ?case by (auto simp: is-map-succ-perm-inv-def)
}
}
case (bind - rev-ok)
{ case (while i ok)
  { case invariant case weaken
  then show ?case by (auto simp: is-map-succ-perm-inv-def elim: less-SucE)
  }
  { case wf show ?case by auto
  }
  { case postcondition
  then show ?case by (auto simp: is-map-final-def is-map-succ-orbits-inv-def)
  }
}
}
case (bind succ-i succ-ok)
{ case (while i ok V A)
  { case invariant
  { case weaken
  then interpret pc0: is-map-postcondition0 iG iM rev-ok succ-i succ-ok
  by unfold-locales auto
  from weaken.loop-cond have i < length (ig-edges iG) succ-ok ok by auto
  with weaken.loop-inv have
    V: V = {ig-tail iG k |k. k < i} and
    A: A = ( $\bigcup$  k<i. orbit (im-succ iM) k)
  by (simp-all add: is-map-succ-orbits-inv-def)
  show ?case
  proof branch-casify
    case then case g
    have V': V = {ig-tail iG ia |ia. ia < (if i  $\in$  A then Suc i else Suc i - 1)}
    using g  $\langle$  V =  $\rightarrow$  by (auto elim: less-SucE)

    have is-map-succ-orbits-inv iG iM (Suc i) (i  $\in$  A) V A
    proof (cases i  $\in$  A)
      case True
      obtain j where j: j < i i  $\in$  orbit (im-succ iM) j
      using True  $\langle$  A =  $\rightarrow$  by auto
      have i-in-less-i:  $\exists x \in \{..<i\}$ . i  $\in$  orbit (im-succ iM) x
      using True  $\langle$  A =  $\rightarrow$  by auto
      have A': A = ( $\bigcup$  i<if True then Suc i else Suc i - 1. orbit (im-succ
iM) i)
      using True unfolding  $\langle$  A =  $\rightarrow$  by (auto 4 3 intro: orbit-trans elim:
less-SucE)

```

```

have X:  $\forall k < i. \forall l < i. ig\text{-tail } iG\ k = ig\text{-tail } iG\ l \longrightarrow l \in orbit\ (im\text{-succ } iM)\ k$ 
  using weaken unfolding is-map-succ-orbits-inv-def by metis
  moreover
  { fix j assume j:  $j < i\ ig\text{-tail } iG\ j = ig\text{-tail } iG\ i$ 
    from i-in-less-i obtain k where k:  $k < i\ i \in orbit\ (im\text{-succ } iM)\ k$  by
      auto
    then have  $ig\text{-tail } iG\ k = ig\text{-tail } iG\ i$ 
      using  $\langle succ\text{-ok} \rangle \langle i < \rightarrow \rangle$  by (auto elim: pc0.in-orbit-im-succE)
    then have  $k \in orbit\ (im\text{-succ } iM)\ j$ 
      using j  $\langle ig\text{-tail } iG\ k = \rightarrow k\ X$  by auto
    then have  $i \in orbit\ (im\text{-succ } iM)\ j$  using k by (auto intro: orbit-trans)
  }
ultimately
  have  $\forall k < Suc\ i. \forall l < Suc\ i. ig\text{-tail } iG\ k = ig\text{-tail } iG\ l \longrightarrow l \in orbit\ (im\text{-succ } iM)\ k$ 
    unfolding all-less-Suc-eq using  $\langle i < \rightarrow \rangle \langle succ\text{-ok} \rangle$ 
    by (auto intro: orbit-swap pc0.self-in-orbit-im-succ)
    with True show ?thesis
    by (simp only: A' V' simp-thms is-map-succ-orbits-inv-def)
next
  case False

  from V g obtain j where j:  $j < i\ ig\text{-tail } iG\ j = ig\text{-tail } iG\ i$  by auto
  with False show ?thesis
    by (auto 0 3 simp: is-map-succ-orbits-inv-def V' A intro: exI[where
x=j] exI[where x=i])
  qed
  then show ?case using weaken by auto
next
  case else case g
  have is-map-succ-orbits-inner-inv iG iM i i {}
  unfolding is-map-succ-orbits-inner-inv-def
  using  $\langle succ\text{-ok} \rangle \langle i < \rightarrow \rangle$  by (auto simp: pc0.self-in-orbit-im-succ)
  with g weaken show ?case by blast
qed
}
{ case if case else case (while A' i')
  { case invariant case weaken
    then interpret pc0: is-map-postcondition0 iG iM rev-ok succ-i succ-ok
      by unfold-locales auto
    have  $succ\text{-ok } i < length\ (ig\text{-edges } iG)\ i' \in orbit\ (im\text{-succ } iM)\ i$ 
      using weaken by (auto simp: is-map-succ-orbits-inner-inv-def)
    have  $i' < length\ (ig\text{-edges } iG)$ 
      using  $\langle i' \in \rightarrow \rangle \langle succ\text{-ok} \rangle \langle i < \rightarrow \rangle$  by (rule pc0.in-orbit-im-succE)

    { assume  $i' \in orbit\ (im\text{-succ } iM)\ i\ i \neq i'$ 
      then have  $i' \in orbit\ (?es)\ i$ 
    }
  }
}

```

```

    by (subst pc0.orbit-es-eq-im) (auto simp add: ⟨succ-ok⟩ ⟨i < -⟩ arcs-mkg)
    then have  $i' \in \text{segment } (?es) i (?es i')$ 
      using ⟨ $i \neq i'$ ⟩ pc0.succ-ok-imp-permutes ⟨succ-ok⟩
      by (intro in-orbit-imp-in-segment) (auto simp: permutes-conv-has-dom)
    then have  $i' \in \text{segment } (im\text{-succ } iM) i (im\text{-succ } iM i')$ 
      by (subst pc0.segment-es-eq-im[symmetric] es-eq-im[symmetric];
          auto simp add: ⟨succ-ok⟩ ⟨i < -⟩ ⟨i' < -⟩ arcs-mkg)+
  } note X = this

  { fix x assume  $x \in \text{segment } (im\text{-succ } iM) i i' i \neq i'$ 
    then have  $x \in \text{segment } (?es) i i'$ 
      by (subst pc0.segment-es-eq-im) (auto simp add: ⟨succ-ok⟩ ⟨i < -⟩ ⟨i'
< -⟩ arcs-mkg)
    then have  $x \in \text{segment } (?es) i (?es i')$ 
      using ⟨ $i \neq i'$ ⟩ pc0.succ-ok-imp-permutes ⟨succ-ok⟩
      by (auto simp: permutes-conv-has-dom bij-is-inj intro: segment-step-2)
    then have  $x \in \text{segment } (im\text{-succ } iM) i (im\text{-succ } iM i')$ 
      by (subst pc0.segment-es-eq-im[symmetric] es-eq-im[symmetric];
          auto simp add: ⟨succ-ok⟩ ⟨i < -⟩ ⟨i' < -⟩ arcs-mkg)+
  } note Y = this

  have Z: is-map-succ-orbits-inner-inv iG iM i (im-succ iM i') (insert i' A')
    using weaken unfolding is-map-succ-orbits-inner-inv-def
    by (auto dest: segment-step-2D X Y simp: orbit.intros segment1-empty
split: if-splits)

  have  $A' \subseteq \text{orbit } (im\text{-succ } iM) i$ 
    using weaken unfolding is-map-succ-orbits-inner-inv-def
  by (auto simp: pc0.self-in-orbit-im-succ dest: segmentD-orbit split: if-splits)
  also have  $\dots \subseteq \text{arcs } (mk\text{-graph } iG)$ 
    by (rule orbit-ss) (auto simp: arcs-mkg pc0.im-succ-le-length ⟨succ-ok⟩ ⟨i
< -⟩)
  finally have  $\text{card } A' < \text{card } (\text{arcs } (mk\text{-graph } iG))$  finite A'
    using ⟨ $i' \notin A'$ ⟩ ⟨ $i' < -$ ⟩
    by - (intro psubset-card-mono, auto simp: arcs-mkg intro: finite-subset)
  then have  $\text{card } A' < \text{length } (ig\text{-edges } iG)$  by (simp add: arcs-mkg)
  show ?case
    using weaken Z ⟨ $\text{card } A' < \text{length } \rightarrow$ ⟩ by (auto simp: card-insert-if ⟨finite
A'⟩)
  }
  { case wf show ?case by simp
  }
  { case postcondition
  then interpret pc0: is-map-postcondition0 iG iM rev-ok succ-i succ-ok
    by unfold-locales auto
  from postcondition have ok succ-ok  $i < \text{length } (ig\text{-edges } iG)$  by simp-all

  from postcondition
  have  $i' \in A'$ 

```

```

      A' = (if i = i' ∧ i ∉ A' then {} else {i} ∪ segment (im-succ iM) i i')
      i' ∈ orbit (im-succ iM) i
      ig-tail iG i ∉ V
    by (simp-all add: is-map-succ-orbits-inner-inv-def)
  moreover
  then have i = i' by (simp split: if-splits add: not-in-segment2)
  ultimately
  have A' = {i} ∪ segment (im-succ iM) i i by simp
  also have segment (im-succ iM) i i = segment ?es i i
    by (auto simp: pc0.segment-es-eq-im ⟨succ-ok⟩ ⟨i < -⟩ arcs-mkg)
  also have ... = orbit ?es i - {i}
    using pc0.succ-ok-imp-permutes ⟨succ-ok⟩
    by (auto simp: permutation-permutes arcs-mkg intro!: segment-x-x-eq)
  also have ... = orbit (im-succ iM) i - {i}
    by (auto simp: pc0.orbit-es-eq-im ⟨succ-ok⟩ ⟨i < -⟩ arcs-mkg)
  finally
  have A': A' = orbit (im-succ iM) i
    using ⟨i < -⟩ ⟨succ-ok⟩ by (auto simp: pc0.self-in-orbit-im-succ)

  from postcondition
  have A = (⋃ k < i. orbit (im-succ iM) k)
  unfolding is-map-succ-orbits-inner-inv-def by (simp add: is-map-succ-orbits-inv-def)
  have A ∪ A' = (⋃ k < Suc i. orbit (im-succ iM) k)
    unfolding A' ⟨A = -⟩ by (auto 2 3 elim: less-SucE)

  from postcondition have V = {ig-tail iG ia | ia. ia < i}
    by (auto simp: ⟨ok⟩ is-map-succ-orbits-inv-def)
  then have V': insert (ig-tail iG i') V = {ig-tail iG ia | ia. ia < Suc i}
    by (auto simp add: ⟨i = i'⟩ elim: less-SucE)

  have *: ⋀ k. k < i ⟹ ig-tail iG k ≠ ig-tail iG i
    using ⟨V = -⟩ ⟨ig-tail iG i ∉ V⟩ by auto

  from postcondition have (∀ k < i. ∀ l < i. ig-tail iG k = ig-tail iG l ⟹ l ∈
  orbit (im-succ iM) k)
    by (simp add: is-map-succ-orbits-inv-def ⟨ok⟩)
  then have X: (∀ k < Suc i. ∀ l < Suc i. ig-tail iG k = ig-tail iG l ⟹ l ∈
  orbit (im-succ iM) k)
    by (auto simp add: all-less-Suc-eq pc0.self-in-orbit-im-succ ⟨succ-ok⟩ ⟨i
  < -⟩ dest: *)

  have is-map-succ-orbits-inv iG iM (i + 1) True (insert (ig-tail iG i') V)
  (A ∪ A')
  unfolding is-map-succ-orbits-inv-def by (simp add: ⟨A ∪ A' = -⟩ V' X)
  then show ?case
    using postcondition ⟨i < -⟩ by auto
  }
}
}
}

```

```

{ case wf show ?case by auto
}
{ case postcondition
interpret pc: is-map-postcondition iG iM rev-ok succ-i succ-ok i ok V A
using postcondition by unfold-locales (auto simp: is-map-final-def)

show ?case (is ?L = ?R)
by (auto simp add: pc.ok-imp-digraph dest: pc.succ-ok-imp-pred elim:
pc.digraph-imp-ok)
}
}
qed

```

declare *ovallidNF-is-map*[*THEN ovalidNF-wp, THEN trivial-label, vcg-l*]

19.3.2 *isolated-nodes*

definition *inv-isolated-nodes s iG vcnt ecnt* \equiv
 $vcnt = length (ig-verts iG)$
 $\wedge ecnt = length (ig-edges iG)$
 $\wedge distinct (ig-verts iG)$
 $\wedge sym-digraph (mk-graph iG)$

definition *inv-isolated-nodes-outer iG i nz* \equiv
 $nz = card (pre-digraph.isolated-verts (mk-graph iG) \cap set (take i (ig-verts iG)))$

definition *inv-isolated-nodes-inner iG v j* \equiv
 $\forall k < j. v \neq ig-tail iG k \wedge v \neq ig-head iG k$

lemma (*in sym-digraph*) *in-arcs-empty-iff*:
 $in-arcs G v = \{\} \longleftrightarrow out-arcs G v = \{\}$
by (*auto simp: out-arcs-def in-arcs-def*)
(*metis graph-symmetric in-arcs-imp-in-arcs-ends reachableE*)+

lemma *take-nth-distinct*:
 $\llbracket distinct xs; n < length xs; xs ! n \in set (take n xs) \rrbracket \implies False$
by (*fastforce simp: distinct-conv-nth in-set-conv-nth*)

lemma *ovallidNF-isolated-nodes*:
 $ovallidNF (\lambda s. distinct (ig-verts iG) \wedge sym-digraph (mk-graph iG))$
(*isolated-nodes iG*)
($\lambda r s. r = (card (pre-digraph.isolated-verts (mk-graph iG)))$)
unfolding *isolated-nodes-def*
apply (*rewrite*
in *oreturn* ($length (ig-verts iG)$) |>> ($\lambda vcnt. \sqcap$)
in *oreturn* ($length (ig-edges iG)$) |>> ($\lambda ecnt. \sqcap$)
to owhile-inv - - -
($\lambda (i, nz) s. inv-isolated-nodes s iG vcnt ecnt$)

```

       $\wedge$  inv-isolated-nodes-outer iG i nz
       $\wedge$   $i \leq \text{vcnt}$ 
      (measure ( $\lambda(i, nz).$   $\text{vcnt} - i$ ))
      owhile-inv-def[symmetric] )
apply (rewrite
  in oreturn (fst iG ! i) |>> ( $\lambda v.$   $\sqcup$ )
  in owhile-inv - ( $\lambda(i, nz).$   $\sqcup$ )
  in oreturn (length (ig-verts iG)) |>> ( $\lambda \text{vcnt}.$   $\sqcup$ )
  in oreturn (length (ig-edges iG)) |>> ( $\lambda \text{ecnt}.$   $\sqcup$ )
  to owhile-inv - - -
  ( $\lambda(j, \text{ret})$  s. inv-isolated-nodes s iG vcnt ecnt
     $\wedge$  inv-isolated-nodes-inner iG v j
     $\wedge$  inv-isolated-nodes-outer iG i nz
     $\wedge$   $v = \text{ig-verts } iG ! i$ 
     $\wedge$   $\text{ret} = (j < \text{ecnt} \wedge \text{ig-tail } iG j \neq v \wedge \text{ig-head } iG j \neq v)$ 
     $\wedge$   $i < \text{vcnt}$ 
     $\wedge$   $j \leq \text{ecnt}$ 
    (measure ( $\lambda(j, \text{ret}).$   $\text{ecnt} - j$ ))
    owhile-inv-def[symmetric])
proof vcg-casify
  case (weaken s)
  then show ?case
    by (auto simp: inv-isolated-nodes-def inv-isolated-nodes-outer-def)
next
  case (while i nz)
  { case invariant
    { case (weaken s')
      then show ?case unfolding BRANCH-def by (auto simp: inv-isolated-nodes-inner-def)
      next
      case bind
      case bind
      case (while j cond)
      { case invariant
        { case weaken
          show ?case
          proof branch-casify
            case else case else case g
            with weaken have  $\text{length } (\text{ig-edges } iG) = j + 1$  by linarith
            with weaken show ?case
            by (auto simp: inv-isolated-nodes-inner-def elim: less-SucE)
          qed (insert weaken, auto simp: inv-isolated-nodes-inner-def elim: less-SucE)
          }
        }
      }
    }
  }
next
  case wf show ?case by auto
next
  case postcondition
  interpret G: sym-digraph mk-graph iG using postcondition by (simp add: inv-isolated-nodes-def)

```

```

have ?var using postcondition by auto

let ?v = ig-verts iG ! i

{ assume A: j = length (snd iG)
  have ?v ∈ pre-digraph.isolated-verts (mk-graph iG)
    using A postcondition by (auto simp: pre-digraph.isolated-verts-def
mkg-simps inv-isolated-nodes-inner-def arcs-mkg)

  have distinct (ig-verts iG) ?v = ig-verts iG ! i i < length (ig-verts iG)
    using postcondition by (auto simp: inv-isolated-nodes-def)
  then have ?v ∉ set (take i (ig-verts iG))
    by (metis take-nth-distinct)

  have Suc (card (pre-digraph.isolated-verts (mk-graph iG) ∩ set (take i (fst
iG))))
    = card (insert ?v (pre-digraph.isolated-verts (mk-graph iG) ∩ set (take
i (fst iG)))) (is - = card ?S)
    using ⟨?v ∉ -⟩ by simp
  also have ?S = pre-digraph.isolated-verts (mk-graph iG) ∩ set (take (Suc
i) (fst iG))
    using ⟨?v ∈ -⟩ ⟨i < -⟩ ⟨?v = -⟩ by (auto simp: take-Suc-conv-app-nth)
  finally
  have inv-isolated-nodes-outer iG (Suc i) (Suc nz)
    using postcondition by (auto simp: inv-isolated-nodes-outer-def)
}
moreover
{ assume A: j ≠ length (snd iG)

  then have *: j ∈ (out-arcs (mk-graph iG) ?v ∪ in-arcs (mk-graph iG) ?v)
    using postcondition by (auto simp: arcs-mkg mkg-simps ig-tail-def
ig-head-def)
  then have out-arcs (mk-graph iG) ?v ≠ {}
    by (auto simp del: in-in-arcs-conv in-out-arcs-conv)
    (auto simp: G.in-arcs-empty-iff[symmetric])
  then have ?v ∉ pre-digraph.isolated-verts (mk-graph iG)
    by (auto simp: pre-digraph.isolated-verts-def)
  then have inv-isolated-nodes-outer iG (Suc i) nz
    using postcondition by (auto simp: inv-isolated-nodes-outer-def take-Suc-conv-app-nth)
}
ultimately
have ?inv using postcondition by auto
from ⟨?var⟩ ⟨?inv⟩ show ?case by blast
}
}
next
case wf show ?case by auto
next
case postcondition

```

```

have pre-digraph.isolated-verts (mk-graph iG)  $\cap$  set (fst iG) = pre-digraph.isolated-verts
(mk-graph iG)
  by (auto simp: pre-digraph.isolated-verts-def mkg-simps)
  with postcondition show ?case
  by (auto simp: inv-isolated-nodes-def inv-isolated-nodes-outer-def)
}
qed

```

```

declare ovalidNF-isolated-nodes[THEN ovalidNF-wp, THEN trivial-label, vcg-l]

```

19.3.3 face-cycles

```

definition inv-face-cycles s iG iM ecnt  $\equiv$ 
  ecnt = length (ig-edges iG)
   $\wedge$  digraph-map (mk-graph iG) iM

```

```

definition fcs-upto :: nat pre-map  $\Rightarrow$  nat  $\Rightarrow$  nat set set where
  fcs-upto iM i  $\equiv$  {pre-digraph-map.face-cycle-set iM k | k. k < i}

```

```

definition inv-face-cycles-outer s iG iM i c edge-info  $\equiv$ 
  let fcs = fcs-upto iM i in
  c = card fcs
   $\wedge$  ( $\forall k < \text{length } (ig\text{-edges } iG). k \in \text{edge-info} \longleftrightarrow k \in \bigcup fcs$ )

```

```

definition inv-face-cycles-inner s iG iM i j c edge-info  $\equiv$ 
  j  $\in$  pre-digraph-map.face-cycle-set iM i
   $\wedge$  c = card (fcs-upto iM i)
   $\wedge$  i  $\notin \bigcup (fcs\text{-upto } iM i)$ 
   $\wedge$  ( $\forall k < \text{length } (ig\text{-edges } iG). k \in \text{edge-info} \longleftrightarrow$ 
    (k  $\in \bigcup (fcs\text{-upto } iM i)$ 
       $\vee (\exists l < \text{funpow-dist1 } (pre\text{-digraph-map.face-cycle-succ } iM) i j. (pre\text{-digraph-map.face-cycle-succ}$ 
iM  $\rightsquigarrow$  l) i = k)))

```

```

lemma finite-fcs-upto: finite (fcs-upto iM i)
  by (auto simp: fcs-upto-def)

```

```

lemma card-orbit-eq-funpow-dist1:

```

```

  assumes x  $\in$  orbit f x shows card (orbit f x) = funpow-dist1 f x x

```

```

proof -

```

```

  have card (orbit f x) = card (( $\lambda n. (f \rightsquigarrow n) x$ ) ' {0..<funpow-dist1 f x x})

```

```

    using assms by (simp only: orbit-conv-funpow-dist1[symmetric])

```

```

  also have  $\dots$  = card {0..<funpow-dist1 f x x}

```

```

    using assms by (intro card-image inj-on-funpow-dist1)

```

```

  finally show ?thesis by simp

```

```

qed

```

```

lemma funpow-dist1-le:

```

```

  assumes y  $\in$  orbit f x x  $\in$  orbit f x

```

shows $\text{funpow-dist1 } f x y \leq \text{funpow-dist1 } f x x$
using *assms* **by** (*intro funpow-dist1-le-self funpow-dist1-prop*) *simp-all*

lemma *funpow-dist1-le-card*:
assumes $y \in \text{orbit } f x x \in \text{orbit } f x$
shows $\text{funpow-dist1 } f x y \leq \text{card } (\text{orbit } f x)$
using *funpow-dist1-le[OF assms]* **using** *assms*
by (*simp add: card-orbit-eq-funpow-dist1*)

lemma (*in digraph-map*) *funpow-dist1-le-card-fcs*:
assumes $b \in \text{face-cycle-set } a$
shows $\text{funpow-dist1 } \text{face-cycle-succ } a b \leq \text{card } (\text{face-cycle-set } a)$
by (*metis assms face-cycle-set-def face-cycle-set-self funpow-dist1-le-card*)

lemma *funpow-dist1-f-eq*:
assumes $b \in \text{orbit } f a a \in \text{orbit } f a a \neq b$
shows $\text{funpow-dist1 } f a (f b) = \text{Suc } (\text{funpow-dist1 } f a b)$
proof –
have *f-inj*: *inj-on* $(\lambda n. (f \text{ `` } n) a) \{0..<\text{funpow-dist1 } f a a\}$
by (*rule inj-on-funpow-dist1*) (*rule assms*)
have $\text{funpow-dist1 } f a b \leq \text{funpow-dist1 } f a a$
using *assms* **by** (*intro funpow-dist1-le*)
moreover
have $\text{funpow-dist1 } f a b \neq \text{funpow-dist1 } f a a$
by (*metis assms funpow-dist1-prop*)
ultimately
have *f-less*: $\text{funpow-dist1 } f a b < \text{funpow-dist1 } f a a$ **by** *simp*

have *f-Suc-eq*: $(f \text{ `` } \text{Suc } (\text{funpow-dist1 } f a b)) a = f b$
using *assms* **by** (*metis funpow.simps(2) o-apply funpow-dist1-prop*)
show *?thesis*
proof (*cases f b = a*)
case *True*
then show *?thesis*
by (*metis Suc-lessI f-Suc-eq f-less assms(2) funpow.simps(1) funpow-neq-less-funpow-dist1 id-apply old.nat.distinct(1) zero-less-Suc*)
next
case *False*
then have $*$: $\text{Suc } (\text{funpow-dist1 } f a b) < \text{funpow-dist1 } f a a$
using *f-Suc-eq* **by** (*metis assms(2) f-less funpow-dist1-prop le-less-Suc-eq less-Suc-eq-le not-less-eq*)
from *f-inj* **have** $**$: $\bigwedge n. n < \text{funpow-dist1 } f a a \implies n \neq \text{Suc } (\text{funpow-dist1 } f a b) \implies (f \text{ `` } n) a \neq f b$
using *f-Suc-eq* **by** (*auto dest!: inj-onD*) (*metis * assms(2) f-Suc-eq funpow-neq-less-funpow-dist1*)
show *?thesis*
proof (*rule ccontr*)
assume *A*: $\neg ?thesis$
have $(f \text{ `` } (\text{funpow-dist1 } f a (f b))) a = f b$

```

    using assms by (intro funpow-dist1-prop) (simp add: orbit.intros)
  with A ** have funpow-dist1 f a a ≤ (funpow-dist1 f a (f b))
    by (metis less-Suc-eq-le not-less-eq)
  then have Suc (funpow-dist1 f a b) < (funpow-dist1 f a (f b)) using * by
linarith
  then have (f ~ Suc (funpow-dist1 f a b)) a ≠ f b
    by (intro funpow-dist1-least) simp-all
  then show False using f-Suc-eq by simp
qed
qed
qed

lemma (in -) funpow-dist1-less-f:
  assumes b ∈ orbit f a a ∈ orbit f a a ≠ b
  shows funpow-dist1 f a b < funpow-dist1 f a (f b)
  using assms by (simp add: funpow-dist1-f-eq)

lemma ovalidNF-face-cycles:
  ovalidNF (λs. digraph-map (mk-graph iG) iM)
  (face-cycles iG iM)
  (λr s. r = card (pre-digraph-map.face-cycle-sets (mk-graph iG) iM))

unfolding face-cycles-def
apply (rewrite
  in oreturn (length (ig-edges iG)) |>> (λecnt. □)
  to owhile-inv - - -
    (λ(edge-info, c, i) s. inv-face-cycles s iG iM ecnt
      ∧ inv-face-cycles-outer s iG iM i c edge-info
      ∧ i ≤ ecnt)
    (measure (λ(edge-info, c, i). ecnt - i))
    owhile-inv-def[symmetric]
  )
apply (rewrite
  in owhile-inv - (λ(-, c, i). □)
  in oreturn (length (ig-edges iG)) |>> (λecnt. □)
  to owhile-inv - - -
    (λ(edge-info, j) s. inv-face-cycles s iG iM ecnt
      ∧ inv-face-cycles-inner s iG iM i j c edge-info
      ∧ i < ecnt)
    (measure (λ(edge-info, j). card (pre-digraph-map.face-cycle-set iM i) -
funpow-dist1 (pre-digraph-map.face-cycle-succ iM) i j))
    owhile-inv-def[symmetric]
  )
proof vcg-casify
  { case (weaken s)
  then show ?case by (auto simp add: inv-face-cycles-def inv-face-cycles-outer-def
fcs-upto-def)
  }
  { case (while edge-info c i)

```

```

{ case (postcondition s)
  moreover have fcs-upto iM (length (ig-edges iG))
    = pre-digraph-map.face-cycle-sets (mk-graph iG) iM
  by (auto simp: pre-digraph-map.face-cycle-sets-def arcs-mkg fcs-upto-def)
  ultimately show ?case by (auto simp: inv-face-cycles-outer-def Let-def)
}
{ case (invariant s)
  { case (weaken s')
    interpret G: digraph-map mk-graph iG iM
    using weaken by (auto simp: inv-face-cycles-def)
    show ?case
    proof branch-casify
      case else case g
      then have G.face-cycle-set i ∈ {G.face-cycle-set k | k. k < i}
        using weaken by (auto simp: inv-face-cycles-outer-def fcs-upto-def dest:
G.face-cycle-eq)
      then have {G.face-cycle-set k | k. k < Suc i} = {G.face-cycle-set k | k. k
< i}
        by (auto elim: less-SucE)
      then have inv-face-cycles-outer s' iG iM (i + 1) c edge-info
        using weaken unfolding inv-face-cycles-outer-def by (auto simp:
fcs-upto-def)
      then have ?inv using weaken by auto
      then show ?case using weaken by auto
    next
      case then case g
      have fd1-triv:  $\bigwedge f x. \text{funpow-dist1 } f x (f x) = 1$ 
        by (simp add: funpow-dist-0)
      have fcs-in: G.face-cycle-succ i ∈ G.face-cycle-set i
        by (simp add: G.face-cycle-succ-inI)

      have i-not-in-fcs:  $i \notin \bigcup (fcs-upto iM i)$ 
        using g weaken
        by (auto simp: inv-face-cycles-outer-def fcs-upto-def)

      from weaken show ?case
        unfolding inv-face-cycles-inner-def inv-face-cycles-outer-def
        using i-not-in-fcs by (auto simp: fd1-triv fcs-in fcs-upto-def)
      qed
    }
  }
{ case if case then
  { case (while edge-info j)
    { case (postcondition s')

      interpret G: digraph-map mk-graph iG iM
      using postcondition by (auto simp: inv-face-cycles-def)

      have ?var using postcondition by auto

```

```

have fu-Suc: fcs-upto iM (Suc j) = fcs-upto iM j ∪ {G.face-cycle-set j}
  by (auto simp: fcs-upto-def elim: less-SucE)
moreover
have G.face-cycle-set j ∉ fcs-upto iM j c = card (fcs-upto iM j)
  using postcondition by (auto simp: inv-face-cycles-inner-def)
ultimately
have Suc c = card (fcs-upto iM (Suc j)) by (simp add: finite-fcs-upto)

have *: ∀ k < length (snd iG). k ∈ edge-info ⟷ (∃ x ∈ fcs-upto iM (Suc
j). k ∈ x)
proof -
  have *: j ∈ orbit G.face-cycle-succ j
    by (simp add: G.face-cycle-set-def[symmetric])
  have ∧ k. (∃ l < funpow-dist1 G.face-cycle-succ j j. (G.face-cycle-succ ~
l) j = k) ⟷ (k ∈ G.face-cycle-set j)
    by (auto simp: G.face-cycle-set-def orbit-conv-funpow-dist1[OF *])
  moreover
  from postcondition have inv-face-cycles-inner s' iG iM j j c edge-info
    by auto
  ultimately
  show ?thesis unfolding inv-face-cycles-inner-def fu-Suc by auto
qed

have ?inv using postcondition *
  by (auto simp: inv-face-cycles-outer-def ⟨Suc c = -⟩)
with ⟨?var⟩ show ?case by blast
}
{ case (invariant s')
  { case (weaken s'')
    interpret G: digraph-map mk-graph iG iM
      using weaken by (auto simp: inv-face-cycles-def)
    have j ∈ G.face-cycle-set i
      using weaken by (auto simp: inv-face-cycles-inner-def)
    then have j ∈ arcs (mk-graph iG)
      by (metis G.face-cycle-set-def G.funpow-face-cycle-succ-no-arc
G.in-face-cycle-setD
funpow-dist1-prop weaken.loop-cond)

    have A: j ∈ pre-digraph-map.face-cycle-set iM i
      using weaken by (auto simp: inv-face-cycles-inner-def)
    then have A': (G.face-cycle-succ ~ funpow-dist1 G.face-cycle-succ i
j) i = j
      by (intro funpow-dist1-prop) (simp add: G.face-cycle-set-def[symmetric])

    { fix k
      have *: ∧ i n f x . i < n ⟹ ∃ j < n. (f ~ j) x = (f ~ i) x by auto

      have (∃ l < funpow-dist1 G.face-cycle-succ i (G.face-cycle-succ j).
(G.face-cycle-succ ~ l) i = k)

```



```

qed auto
declare ovalidNF-face-cycles[THEN ovalidNF-wp, THEN trivial-label, vcg-l]

lemma ovalidNF-euler-genus:
  ovalidNF ( $\lambda s. \text{distinct } (ig\text{-verts } iG) \wedge \text{digraph-map } (mk\text{-graph } iG) iM \wedge c = \text{card}$ 
  (pre-digraph.sccs (mk-graph iG)))
  (euler-genus iG iM c)
  ( $\lambda r s. r = \text{pre-digraph-map.euler-genus } (mk\text{-graph } iG) iM$ )

  unfolding euler-genus-def
proof vcg-casify
  case weaken
  have distinct (ig-verts iG) using weaken by simp
  interpret G: digraph-map mk-graph iG iM using weaken by simp
  have len-card:
    length (ig-verts iG) = card (verts (mk-graph iG))
    length (ig-edges iG) = card (arcs (mk-graph iG))
    using  $\langle \text{distinct} \rightarrow \rangle$  by (auto simp: mkg-simps arcs-mkg distinct-card)
  show ?case
    using weaken by (auto simp: G.euler-genus-def G.euler-char-def len-card)
qed

declare ovalidNF-euler-genus[THEN ovalidNF-wp, THEN trivial-label, vcg-l]

lemma ovalidNF-certify:
  ovalidNF ( $\lambda s. \text{distinct } (ig\text{-verts } iG) \wedge \text{fin-digraph } (mk\text{-graph } iG) \wedge c = \text{card}$ 
  (pre-digraph.sccs (mk-graph iG)))
  (certify iG iM c)
  ( $\lambda r s. r \longleftrightarrow \text{pre-digraph-map.euler-genus } (mk\text{-graph } iG) (mk\text{-map } (mk\text{-graph } iG)$ 
  iM) = 0
   $\wedge \text{digraph-map } (mk\text{-graph } iG) (mk\text{-map } (mk\text{-graph } iG) iM)$ 
   $\wedge (\forall i < \text{length } (ig\text{-edges } iG). \text{im-pred } iM (\text{im-succ } iM i) = i)$  )

  unfolding certify-def
proof vcg-casify
  case weaken
  then interpret fin-digraph mk-graph iG by auto
  from weaken show ?case by (auto simp: BRANCH-def intro: wf-digraph)
qed

end
theory Planarity-Certificates
imports
  Planarity/Kuratowski-Combinatorial
  Verification/Check-Non-Planarity-Verification
  Verification/Check-Planarity-Verification
begin

end

```

References

- [1] L. Noschinski. *Formalizing Graph Theory and Planarity Certificates*. PhD thesis, Technische Universität München, München, Nov. 2015.