

Graph Theory

By Lars Noschinski

April 10, 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	8
3	Auxiliary List Lemmas	10
4	Permutations as Products of Disjoint Cycles	10
4.1	Cyclic Permutations	10
4.2	Arbitrary Permutations	12
5	List Orbits	14
5.1	Relation to <i>cyclic-on</i>	16
5.2	Permutations of a List	17
5.3	Enumerating Permutations from List Orbits	19
5.4	Lists of Permutations	19
6	Enumerating Maps	20
7	Compute Face Cycles	22
8	Kuratowski Graphs are not Combinatorially Planar	25
8.1	A concrete K5 graph	25
8.2	A concrete K33 graph	25
8.3	Generalization to arbitrary Kuratowski Graphs	25
8.3.1	Number of Face Cycles is a Graph Invariant	25

8.3.2	Combinatorial planarity is a Graph Invariant	26
8.3.3	Completeness is a Graph Invariant	26
8.3.4	Conclusion	27
9	<i>n</i>-step reachability	29
10	More	30
11	Modifying Permutations	31
12	Cyclic Permutations	32
13	Combinatorial Planarity and Subdivisions	33
14	Combinatorial Planarity and Subgraphs	38
14.1	Deleting an isolated vertex	40
14.2	Deleting an arc pair	42
14.3	Modifying <i>edge-rev</i>	53
14.4	Conclusion	53
15	Implementation of a Non-Planarity Checker	54
15.1	An abstract graph datatype	55
15.2	Code	55
16	Verification of a Non-Planarity Checker	61
16.1	Graph Basics and Implementation	61
16.2	Total Correctness	64
16.2.1	Procedure <i>is-subgraph</i>	64
16.2.2	Procedure <i>is-loop-free</i>	66
16.2.3	Procedure <i>select-nodes</i>	66
16.2.4	Procedure <i>find-endpoint</i>	67
16.2.5	Procedure <i>contract</i>	69
16.2.6	Procedure <i>is-K33</i>	71
16.2.7	Procedure <i>is-K5</i>	74
16.2.8	Soundness of the Checker	75
17	Auxilliary Lemmas for Autocorres	76
17.1	Option monad	76
18	AutoCorres setup for VCG labelling	77
18.1	Labeled VCG theorems for branching	77
18.2	Labelled VCG theorems for the option monad	78

19 Verification of a Planarity Checker	80
19.1 Implementation Types	80
19.2 Implementation	81
19.3 Verification	84
19.3.1 <i>is-map</i>	84
19.3.2 <i>isolated-nodes</i>	88
19.3.3 <i>face-cycles</i>	88

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 $(a - b) \bmod c = (a - b \bmod c) \bmod c$

<proof>

lemma *inj-on-f-imageI*:

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

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

<proof>

1 Combinatorial Maps

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

has-dom arev (arcs G)

<proof>

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** $M :: 'b\ \text{pre-map}$

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: $\bigwedge v. v \in \text{verts } G \implies \text{out-arcs } G\ v \neq \{\} \implies \text{cyclic-on } (edge-succ\ M)\ (\text{out-arcs } G\ v)$*

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*
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

context *digraph-map*
begin

lemma *digraph-map[intro]*: *digraph-map* *G* *M* $\langle \text{proof} \rangle$

lemma *permutation-edge-succ*: *permutation* (*edge-succ* *M*)
 $\langle \text{proof} \rangle$

lemma *edge-pred-succ[simp]*: *edge-pred* *M* (*edge-succ* *M* *a*) = *a*
 $\langle \text{proof} \rangle$

lemma *edge-succ-pred[simp]*: *edge-succ* *M* (*edge-pred* *M* *a*) = *a*
 $\langle \text{proof} \rangle$

lemma *edge-pred-permutes*: *edge-pred* *M* permutes arcs *G*
 $\langle \text{proof} \rangle$

lemma *permutation-edge-pred*: *permutation* (*edge-pred* *M*)
 $\langle \text{proof} \rangle$

lemma *edge-succ-eq-iff[simp]*: $\bigwedge x y. \text{edge-succ } M x = \text{edge-succ } M y \iff x = y$
 $\langle \text{proof} \rangle$

lemma *edge-rev-in-arcs[simp]*: *edge-rev* *M* *a* \in arcs *G* \iff *a* \in arcs *G*
 $\langle \text{proof} \rangle$

lemma *edge-succ-in-arcs[simp]*: *edge-succ* *M* *a* \in arcs *G* \iff *a* \in arcs *G*

$\langle \text{proof} \rangle$

lemma *edge-pred-in-arcs*[simp]: $\text{edge-pred } M \ a \in \text{arcs } G \longleftrightarrow a \in \text{arcs } G$
 $\langle \text{proof} \rangle$

lemma *tail-edge-succ*[simp]: $\text{tail } G \ (\text{edge-succ } M \ a) = \text{tail } G \ a$
 $\langle \text{proof} \rangle$

lemma *tail-edge-pred*[simp]: $\text{tail } G \ (\text{edge-pred } M \ a) = \text{tail } G \ a$
 $\langle \text{proof} \rangle$

lemma *bij-edge-succ*[intro]: $\text{bij} \ (\text{edge-succ } M)$
 $\langle \text{proof} \rangle$

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

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

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

lemma *face-cycle-pred-succ*[simp]:
shows $\text{face-cycle-pred} \ (\text{face-cycle-succ } a) = a$
 $\langle \text{proof} \rangle$

lemma *face-cycle-succ-pred*[simp]:
shows $\text{face-cycle-succ} \ (\text{face-cycle-pred } a) = a$
 $\langle \text{proof} \rangle$

lemma *tail-face-cycle-succ*: $a \in \text{arcs } G \implies \text{tail } G \ (\text{face-cycle-succ } a) = \text{head } G$
 a
 $\langle \text{proof} \rangle$

lemma *funpow-prop*:
assumes $\bigwedge x. P \ (f \ x) \longleftrightarrow P \ x$
shows $P \ ((f \ \sim^n) \ x) \longleftrightarrow P \ x$
 $\langle \text{proof} \rangle$

lemma *face-cycle-succ-no-arc*[simp]: $a \notin \text{arcs } G \implies \text{face-cycle-succ } a = a$
 $\langle \text{proof} \rangle$

lemma *funpow-face-cycle-succ-no-arc*[simp]:
assumes $a \notin \text{arcs } G$ **shows** $(\text{face-cycle-succ} \ \sim^n) \ a = a$
 $\langle \text{proof} \rangle$

lemma *funpow-face-cycle-pred-no-arc*[simp]:
assumes $a \notin \text{arcs } G$ **shows** $(\text{face-cycle-pred } \sim^n) a = a$
 ⟨proof⟩

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

lemma *face-cycle-pred-closed*[simp]:
 $\text{face-cycle-pred } a \in \text{arcs } G \longleftrightarrow a \in \text{arcs } G$
 ⟨proof⟩

lemma *face-cycle-succ-permutes*:
 $\text{face-cycle-succ permutes arcs } G$
 ⟨proof⟩

lemma *permutation-face-cycle-succ: permutation face-cycle-succ*
 ⟨proof⟩

lemma *bij-face-cycle-succ: bij face-cycle-succ*
 ⟨proof⟩

lemma *face-cycle-pred-permutes*:
 $\text{face-cycle-pred permutes arcs } G$
 ⟨proof⟩

definition (in *pre-digraph-map*) *face-cycle-set* :: 'b \Rightarrow 'b set **where**
 $\text{face-cycle-set } a = \text{orbit face-cycle-succ } a$

definition (in *pre-digraph-map*) *face-cycle-sets* :: 'b set set **where**
 $\text{face-cycle-sets} = \text{face-cycle-set } ` \text{arcs } G$

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

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

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

lemma *finite-face-cycle-set*[simp, intro]: $\text{finite } (\text{face-cycle-set } a)$
 ⟨proof⟩

lemma *finite-face-cycle-sets*[simp, intro]: $\text{finite face-cycle-sets}$
 ⟨proof⟩

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 $P a$
 $\langle \text{proof} \rangle$

lemma *face-cycle-succ-cyclic*:
cyclic-on face-cycle-succ (face-cycle-set a)
 $\langle \text{proof} \rangle$

lemma *face-cycle-eq*:
assumes $b \in \text{face-cycle-set } a$ **shows** $\text{face-cycle-set } b = \text{face-cycle-set } a$
 $\langle \text{proof} \rangle$

lemma *face-cycle-succ-in-arcsI*: $\bigwedge a. a \in \text{arcs } G \implies \text{face-cycle-succ } a \in \text{arcs } G$
 $\langle \text{proof} \rangle$

lemma *face-cycle-succ-inI*: $\bigwedge x y. x \in \text{face-cycle-set } y \implies \text{face-cycle-succ } x \in \text{face-cycle-set } y$
 $\langle \text{proof} \rangle$

lemma *face-cycle-succ-inD*: $\bigwedge x y. \text{face-cycle-succ } x \in \text{face-cycle-set } y \implies x \in \text{face-cycle-set } y$
 $\langle \text{proof} \rangle$

lemma *face-cycle-set-parts*:
 $\text{face-cycle-set } a = \text{face-cycle-set } b \vee \text{face-cycle-set } a \cap \text{face-cycle-set } b = \{\}$
 $\langle \text{proof} \rangle$

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*: $\text{reflp } \text{fc-equiv}$
 $\langle \text{proof} \rangle$

lemma *symp-fc-equiv*: $\text{symp } \text{fc-equiv}$
 $\langle \text{proof} \rangle$

lemma *transp-fc-equiv*: $\text{transp } \text{fc-equiv}$
 $\langle \text{proof} \rangle$

lemma *equivp-fc-equiv*
 $\langle \text{proof} \rangle$

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

lemma *in-face-cycle-setsD*:

assumes $x \in \text{face-cycle-sets}$ **shows** $x \subseteq \text{arcs } G$
 ⟨proof⟩

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

interpretation *wf-digraph* G ⟨proof⟩

lemma *isolated-verts-app-iso[simp]*:
pre-digraph.isolated-verts (*app-iso* $\text{hom } G$) = *iso-verts* hom ' *isolated-verts*
 ⟨proof⟩

lemma *card-isolated-verts-iso[simp]*:
card (*iso-verts* hom ' *pre-digraph.isolated-verts* G) = *card isolated-verts*
 ⟨proof⟩

end

context *digraph-map* **begin**

lemma *face-cycle-succ-neg*:
assumes $a \in \text{arcs } G$ *tail* $G \ a \neq \text{head } G \ a$ **shows** *face-cycle-succ* $a \neq a$
 ⟨proof⟩

end

2 Maps and Isomorphism

definition (in *pre-digraph*)

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

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

lemma *funcsetI-permutes*:
assumes $f\ permutes\ S$ **shows** $f \in S \rightarrow S$
 $\langle proof \rangle$

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

interpretation $wf\text{-}digraph\ G\ \langle proof \rangle$

lemma *wrap-iso-arcs-iso-arcs[simp]*:
assumes $x \in arcs\ G$
shows $wrap\text{-}iso\text{-}arcs\ hom\ f\ (iso\text{-}arcs\ hom\ x) = iso\text{-}arcs\ hom\ (f\ x)$
 $\langle proof \rangle$

lemma *inj-on-wrap-iso-arcs*:
assumes $dom: \bigwedge f. f \in F \implies has\text{-}dom\ f\ (arcs\ G)$
assumes $funcset: F \subseteq arcs\ G \rightarrow arcs\ G$
shows $inj\text{-}on\ (wrap\text{-}iso\text{-}arcs\ hom)\ F$
 $\langle proof \rangle$

lemma *inj-on-wrap-iso-arcs-f*:
assumes $A \subseteq arcs\ G\ f \in A \rightarrow A\ B = iso\text{-}arcs\ hom\ 'A$
assumes $inj\text{-}on\ f\ A$ **shows** $inj\text{-}on\ (wrap\text{-}iso\text{-}arcs\ hom\ f)\ B$
 $\langle proof \rangle$

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

lemma *wrap-iso-arcs-permutes*:
assumes $A \subseteq arcs\ G\ f\ permutes\ A$
shows $wrap\text{-}iso\text{-}arcs\ hom\ f\ permutes\ (iso\text{-}arcs\ hom\ 'A)$
 $\langle proof \rangle$

end

lemma (in *digraph-map*) *digraph-map-isoI*:

```

  assumes digraph-isomorphism hom shows digraph-map (app-iso hom G) (map-iso hom)
  <proof>

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

```

3 Auxiliary List Lemmas

```

lemma nth-rotate-conv-nth1-conv-nth:
  assumes  $m < \text{length } xs$ 
  shows  $\text{rotate1 } xs ! m = xs ! (\text{Suc } m \text{ mod length } xs)$ 
  <proof>

lemma nth-rotate-conv-nth:
  assumes  $m < \text{length } xs$ 
  shows  $\text{rotate } n \text{ } xs ! m = xs ! ((m + n) \text{ mod length } xs)$ 
  <proof>

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  $xs \ x = (\text{if } x \in \text{set } xs \text{ then } xs ! ((\text{index } xs \ x + 1) \text{ mod length } xs) \text{ else } x)$ 

```

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

```

```

lemma list-succ-Nil:
  list-succ [] = id
  <proof>

```

lemma *list-succ-singleton*:

list-succ [x] = *list-succ* []
(proof)

lemma *list-succ-short*:

assumes *length xs* < 2 **shows** *list-succ xs* = *id*
(proof)

lemma *list-succ-simps*:

index xs x + 1 = *length xs* \implies *list-succ xs x* = *xs* ! 0
index xs x + 1 < *length xs* \implies *list-succ xs x* = *xs* ! (*index xs x + 1*)
length xs \leq *index xs x* \implies *list-succ xs x* = *x*
(proof)

lemma *list-succ-not-in*:

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

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

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

lemma *inj-list-succ*: *distinct xs* \implies *inj (list-succ xs)*

(proof)

lemma *inv-list-succ-eq*: *distinct xs* \implies *inv (list-succ xs)* = *list-succ (rev xs)*

(proof)

lemma *bij-list-succ*: *distinct xs* \implies *bij (list-succ xs)*

(proof)

lemma *list-succ-permutes*:

assumes *distinct xs* **shows** *list-succ xs permutes set xs*
(proof)

lemma *permutation-list-succ*:

assumes *distinct xs* **shows** *permutation (list-succ xs)*
(proof)

lemma *list-succ-nth*:

assumes *distinct xs* $n < \text{length } xs$ **shows** *list-succ xs (xs* ! $n)$ = *xs* ! (*Suc n mod length xs*)
(proof)

lemma *list-succ-last[simp]*:

assumes *distinct xs* $xs \neq []$ **shows** *list-succ xs (last xs)* = *hd xs*
(proof)

lemma *list-succ-rotate1*[simp]:
assumes *distinct xs* **shows** $\text{list-succ } (\text{rotate1 } xs) = \text{list-succ } xs$
⟨*proof*⟩

lemma *list-succ-rotate*[simp]:
assumes *distinct xs* **shows** $\text{list-succ } (\text{rotate } n \ xs) = \text{list-succ } xs$
⟨*proof*⟩

lemma *list-succ-in-conv*:
 $\text{list-succ } xs \ x \in \text{set } xs \longleftrightarrow x \in \text{set } xs$
⟨*proof*⟩

lemma *list-succ-in-conv1*:
assumes $A \cap \text{set } xs = \{\}$
shows $\text{list-succ } xs \ x \in A \longleftrightarrow x \in A$
⟨*proof*⟩

lemma *list-succ-commute*:
assumes $\text{set } xs \cap \text{set } ys = \{\}$
shows $\text{list-succ } xs \ (\text{list-succ } ys \ x) = \text{list-succ } ys \ (\text{list-succ } xs \ x)$
⟨*proof*⟩

4.2 Arbitrary Permutations

fun *lists-succ* :: 'a list list \Rightarrow 'a \Rightarrow 'a **where**
 $\text{lists-succ } [] \ x = x$
| $\text{lists-succ } (xs \# \ xss) \ x = \text{list-succ } xs \ (\text{lists-succ } \ xss \ x)$

definition *distincts* :: 'a list list \Rightarrow bool **where**
 $\text{distincts } \ xss \equiv \text{distinct } \ xss \wedge (\forall xs \in \text{set } \ xss. \text{distinct } xs \wedge xs \neq []) \wedge (\forall xs \in \text{set } \ xss. \forall ys \in \text{set } \ xss. xs \neq ys \longrightarrow \text{set } xs \cap \text{set } ys = \{\})$

lemma *distincts-distinct*: $\text{distincts } \ xss \Longrightarrow \text{distinct } \ xss$
⟨*proof*⟩

lemma *distincts-Nil*[simp]: $\text{distincts } []$
⟨*proof*⟩

lemma *distincts-single*: $\text{distincts } [xs] \longleftrightarrow \text{distinct } xs \wedge xs \neq []$
⟨*proof*⟩

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

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

<proof>

lemma *distincts-rev*:

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

<proof>

lemma *length-distincts*:

assumes *distincts xss*

shows $length xss = card (set ` set xss)$

<proof>

lemma *distincts-remove1*: $distincts xss \implies distincts (remove1 xs xss)$

<proof>

lemma *distinct-Cons-remove1*:

$x \in set xs \implies distinct (x \# remove1 x xs) = distinct xs$

<proof>

lemma *set-Cons-remove1*:

$x \in set xs \implies set (x \# remove1 x xs) = set xs$

<proof>

lemma *distincts-Cons-remove1*:

$xs \in set xss \implies distincts (xs \# remove1 xs xss) = distincts xss$

<proof>

lemma *distincts-inj-on-set*:

assumes *distincts xss* **shows** *inj-on set (set xss)*

<proof>

lemma *distincts-distinct-set*:

assumes *distincts xss* **shows** *distinct (map set xss)*

<proof>

lemma *distincts-distinct-nth*:

assumes *distincts xss* $n < length xss$ **shows** *distinct (xss ! n)*

<proof>

lemma *lists-succ-not-in*:

assumes $x \notin (\bigcup xs \in set xss. set xs)$ **shows** *lists-succ xss x = x*

<proof>

lemma *lists-succ-in-conv*:

$lists-succ xss x \in (\bigcup xs \in set xss. set xs) \longleftrightarrow x \in (\bigcup xs \in set xss. set xs)$

<proof>

lemma *lists-succ-in-conv1*:

assumes $A \cap (\bigcup xs \in set xss. set xs) = \{\}$

shows $lists-succ xss x \in A \longleftrightarrow x \in A$

<proof>

lemma *lists-succ-Cons-pf*: $lists-succ (xs \# xss) = list-succ xs o lists-succ xss$
<proof>

lemma *lists-succ-Nil-pf*: $lists-succ [] = id$
<proof>

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

lemma *lists-succ-permutes*:
assumes *distincts xss*
shows $lists-succ xss permutes (\bigcup xs \in set xss. set xs)$
<proof>

lemma *bij-lists-succ*: $distincts xss \implies bij (lists-succ xss)$
<proof>

lemma *lists-succ-snoc*: $lists-succ (xss @ [xs]) = lists-succ xss o list-succ xs$
<proof>

lemma *inv-lists-succ-eq*:
assumes *distincts xss*
shows $inv (lists-succ xss) = lists-succ (rev (map rev xss))$
<proof>

lemma *lists-succ-remove1*:
assumes *distincts xss xs \in set xss*
shows $lists-succ (xs \# remove1 xs xss) = lists-succ xss$
<proof>

lemma *lists-succ-no-order*:
assumes *distincts xss distincts yss set xss = set yss*
shows $lists-succ xss = lists-succ yss$
<proof>

5 List Orbits

Computes the orbit of x under f

definition *orbit-list* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a list$ **where**
 $orbit-list f x \equiv iterate 0 (funpow-dist1 f x x) f x$

partial-function (*tailrec*)
 $orbit-list-impl :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a list \Rightarrow 'a \Rightarrow 'a list$
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 [] 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) f x = x # *iterate* 0 (*funpow-dist1* f (f x) y)
f (f x)
⟨*proof*⟩

lemma *orbit-list-impl-conv*:
 assumes y ∈ *orbit* f x
 shows *orbit-list-impl* f y acc x = rev acc @ *iterate* 0 (*funpow-dist1* f x y) f x
⟨*proof*⟩

lemma *orbit-list-conv-impl*:
 assumes x ∈ *orbit* f x
 shows *orbit-list* f x = *orbit-list-impl* f x [] x
⟨*proof*⟩

lemma *set-orbit-list*:
 assumes x ∈ *orbit* f x
 shows set (*orbit-list* f x) = *orbit* f x
⟨*proof*⟩

lemma *set-orbit-list'*:
 assumes permutation f **shows** set (*orbit-list* f x) = *orbit* f x
⟨*proof*⟩

lemma *distinct-orbit-list*:
 assumes x ∈ *orbit* f x
 shows *distinct* (*orbit-list* f x)
⟨*proof*⟩

lemma *distinct-orbit-list'*:

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

lemma *orbits-list-conv-impl*:

assumes *permutation f*
shows *orbits-list f xs = orbits-list-impl f xs*
<proof>

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

<proof>

lemma *sset-orbits-list*:

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

5.1 Relation to *cyclic-on*

lemma *list-succ-orbit-list*:

assumes *s \in orbit f s \wedge x. x \notin orbit f s \implies f x = x*
shows *list-succ (orbit-list f s) = f*
<proof>

lemma *list-succ-funpow-conv*:

assumes *A: distinct xs x \in set xs*
shows *(list-succ xs $\hat{~}$ n) x = xs ! ((index xs x + n) mod length xs)*
<proof>

lemma *orbit-list-succ*:

assumes *distinct xs x \in set xs*
shows *orbit (list-succ xs) x = set xs*
<proof>

lemma *cyclic-on-list-succ*:

assumes *distinct xs xs \neq []* **shows** *cyclic-on (list-succ xs) (set xs)*
<proof>

lemma *obtain-orbit-list-func*:

assumes *s \in orbit f s \wedge x. x \notin orbit f s \implies f x = x*
obtains *xs* **where** *f = list-succ xs set xs = orbit f s distinct xs hd xs = s*
<proof>

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

assumes *cyclic-on f S \wedge x. x \notin S \implies f x = x*
obtains *xs* **where** *f = list-succ xs set xs = S distinct xs*
<proof>

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

assumes *cyclic-on* f S f *permutes* S
obtains xs **where** $f = list\text{-succ } xs$ $set\ xs = S$ *distinct* xs
 $\langle proof \rangle$

lemma *list-succ-unique*:

assumes $s \in orbit\ f\ s \wedge x. x \notin orbit\ f\ s \implies f\ x = x$
shows $\exists! xs. f = list\text{-succ } xs \wedge distinct\ xs \wedge hd\ xs = s \wedge set\ xs = orbit\ f\ s$
 $\langle proof \rangle$

lemma *distincts-orbits-list*:

assumes *distinct as permutation* f
shows *distincts* (*orbits-list* f as)
 $\langle proof \rangle$

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

assumes *distincts* xss
shows $A \in sset\ xss \implies cyclic\text{-on } (lists\text{-succ } xss)\ A$
 $\langle proof \rangle$

lemma *cyclic-on-lists-succ*:

assumes *distincts* xss
shows $\bigwedge xs. xs \in set\ xss \implies cyclic\text{-on } (lists\text{-succ } xss)\ (set\ xs)$
 $\langle proof \rangle$

lemma *permutes-as-lists-succ*:

assumes *distincts* xss
assumes *ls-eq*: $\bigwedge xs. xs \in set\ xss \implies list\text{-succ } xs = perm\text{-restrict } f\ (set\ xs)$
assumes f *permutes* ($\bigcup (sset\ xss)$)
shows $f = lists\text{-succ } xss$
 $\langle proof \rangle$

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

assumes
permutes: f *permutes* S **and**
 $S: S = \bigcup (sset\ css)$ **and**
dists: *distincts* css **and**
cyclic: $\bigwedge cs. cs \in set\ css \implies cyclic\text{-on } f\ (set\ cs)$
obtains xss **where** $f = lists\text{-succ } xss$ *distincts* xss $map\ set\ xss = map\ set\ css$
 $map\ hd\ xss = map\ hd\ css$
 $\langle proof \rangle$

5.2 Permutations of a List

lemma *length-remove1-less*:

assumes $x \in set\ xs$ **shows** $length\ (remove1\ x\ xs) < length\ xs$
 $\langle proof \rangle$

context notes [*simp*] = *length-remove1-less* **begin**

fun *permutations* :: 'a list \Rightarrow '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 = []
  <proof>
```

```
lemma permutations-not-nil:
  assumes xs ≠ []
  shows permutations xs = concat (map (λx. map ((#) x) (permutations (remove1
x xs)))) xs
  <proof>
```

```
lemma set-permutations-step:
  assumes xs ≠ []
  shows set (permutations xs) = (⋃ x ∈ set xs. Cons x ` set (permutations (remove1
x xs)))
  <proof>
```

```
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)
  <proof>
```

```
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>
```

```
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>
```

```
lemma list-succ-set-cyc-permutations:
  assumes distinct xs xs ≠ []
```

shows $list\text{-}succ \text{ ' set } (cyc\text{-}permutations\ xs) = \{f. f\text{ permutes set } xs \wedge cyclic\text{-}on\ f\ (set\ xs)\}$ (**is** ?L = ?R)
 ⟨proof⟩

5.3 Enumerating Permutations from List Orbits

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

lemma $cyc\text{-}permutationss\text{-}Nil[simp]$: $cyc\text{-}permutationss\ [] = [[]]$
 ⟨proof⟩

lemma $in\text{-}set\text{-}cyc\text{-}permutationss$:

assumes $distincts\ xss$

shows $yss \in set\ (cyc\text{-}permutationss\ xss) \longleftrightarrow distincts\ yss \wedge map\ set\ xss = map\ set\ yss \wedge map\ hd\ xss = map\ hd\ yss$
 ⟨proof⟩

lemma $lists\text{-}succ\text{-}set\text{-}cyc\text{-}permutationss$:

assumes $distincts\ xss$

shows $lists\text{-}succ \text{ ' set } (cyc\text{-}permutationss\ xss) = \{f. f\text{ permutes } \bigcup (sset\ xss) \wedge (\forall c \in sset\ xss. cyclic\text{-}on\ f\ c)\}$ (**is** ?L = ?R)
 ⟨proof⟩

5.4 Lists of Permutations

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

lemma $permutationss\text{-}Nil[simp]$: $permutationss\ [] = [[]]$
 ⟨proof⟩

lemma $permutationss\text{-}Cons$:

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

lemma $in\text{-}set\text{-}permutationss$:

assumes $distincts\ xss$

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

lemma $set\text{-}permutationss$:

assumes $distincts\ xss$

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

lemma $permutationss\text{-}complete$:

assumes *distincts xss distincts yss xss* $\neq []$
and *set ' set xss = set ' set yss*
shows *set yss \in set ' set (permutationss xss)*
<proof>

lemma *permutations-complete:*
assumes *distinct xs distinct ys set xs = set ys*
shows *ys \in set (permutations xs)*
<proof>

end
theory *Digraph-Map-Impl*
imports
Graph-Genus
Executable-Permutations
Transitive-Closure.Transitive-Closure-Impl
begin

6 Enumerating Maps

definition *grouped-by-fst* :: $('a \times 'b)$ list \Rightarrow $('a \times 'b)$ list list **where**
grouped-by-fst xs = map ($\lambda u.$ filter ($\lambda x.$ fst x = u) xs) (remdups (map fst xs))

fun *grouped-out-arcs* :: $'a$ list \times $('a \times 'a)$ list \Rightarrow $('a \times 'a)$ list list **where**
grouped-out-arcs (vs,as) = grouped-by-fst as

definition *all-maps-list* :: $('a$ list \times $('a \times 'a)$ list) \Rightarrow $('a \times 'a)$ list list list **where**
all-maps-list G-list = (cyc-permutationss o grouped-out-arcs) G-list

definition *list-digraph-ext* ext G-list \equiv ($[]$ pverts = set (fst G-list), parcs = set (snd G-list), ... = ext $)$

abbreviation *list-digraph* \equiv *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)
<proof>

lemma *union-grouped-by-fst:*
 $(\bigcup xs \in set (grouped-by-fst ys). set xs) = set ys$
<proof>

lemma *union-grouped-out-arcs:*

$(\bigcup xs \in \text{set } (\text{grouped-out-arcs } G\text{-list}). \text{set } xs) = \text{set } (\text{snd } G\text{-list})$
 ⟨proof⟩

lemma *nil-not-in-grouped-out-arcs*: $[] \notin \text{set } (\text{grouped-out-arcs } G\text{-list})$
 ⟨proof⟩

lemma *set-grouped-out-arcs*:

assumes *pair-wf-digraph* (*list-digraph* $G\text{-list}$)

shows $\text{set } \{ \text{set } (\text{grouped-out-arcs } G\text{-list}) = \{ \text{out-arcs } (\text{list-digraph } G\text{-list}) \ v \mid v. v \in \text{pverts } (\text{list-digraph } G\text{-list}) \wedge \text{out-arcs } (\text{list-digraph } G\text{-list}) \ v \neq \{\} \} \}$

(**is** $?L = ?R$)

⟨proof⟩

lemma *distincts-grouped-by-fst*:

assumes *distinct* xs **shows** *distincts* (*grouped-by-fst* xs)

⟨proof⟩

lemma *distincts-grouped-arcs*:

assumes *distinct* (*snd* $G\text{-list}$) **shows** *distincts* (*grouped-out-arcs* $G\text{-list}$)

⟨proof⟩

lemma *distincts-in-all-maps-list*:

distinct (*snd* X) $\implies xss \in \text{set } (\text{all-maps-list } X) \implies \text{distincts } xss$

⟨proof⟩

definition *to-map* :: $'a \times 'a \text{ set} \Rightarrow ('a \times 'a \Rightarrow 'a \times 'a) \Rightarrow ('a \times 'a) \text{ pre-map}$
where

to-map $A \ f = (\text{edge-rev} = \text{swap-in } A, \text{edge-succ} = f)$

abbreviation *to-map'* $as \ xss \equiv \text{to-map } (\text{set } as) (\text{lists-succ } xss)$

definition *all-maps* :: $'a \text{ pair-pre-digraph} \Rightarrow ('a \times 'a) \text{ pre-map set}$ **where**

all-maps $G \equiv \text{to-map } (\text{arcs } G) \{ f. f \text{ permutes arcs } G \wedge (\forall v \in \text{verts } G. \text{out-arcs } G \ v \neq \{\} \longrightarrow \text{cyclic-on } f (\text{out-arcs } G \ v)) \}$

definition *maps-all-maps-list* :: $('a \text{ list} \times ('a \times 'a) \text{ list}) \Rightarrow ('a \times 'a) \text{ pre-map list}$

where

maps-all-maps-list $G\text{-list} = \text{map } (\text{to-map } (\text{set } (\text{snd } G\text{-list})) \circ \text{lists-succ}) (\text{all-maps-list } G\text{-list})$

lemma (**in** *pair-graph*) *all-maps-correct*:

shows *all-maps* $G = \{ M. \text{digraph-map } G \ M \}$

⟨proof⟩

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*⟩

7 Compute Face Cycles

definition *lists-fc-succ* :: ('a × 'a) list list ⇒ ('a × 'a) ⇒ ('a × 'a) **where**
lists-fc-succ *xss* = (let *sxss* = \bigcup (*sset* *xss*) in (λx . *lists-succ* *xss* (*swap-in* *sxss* *x*)))

locale *lists-digraph-map* =
fixes *G-list* :: 'b list × ('b × 'b) list
and *xss* :: ('b × 'b) list list
assumes *digraph-map*: *digraph-map* (*list-digraph* *G-list*) (*to-map'* (*snd* *G-list*)
xss)
assumes *no-loops*: $\bigwedge a$. *a* ∈ *parcs* (*list-digraph* *G-list*) ⇒ *fst* *a* ≠ *snd* *a*
assumes *distincts-xss*: *distincts* *xss*
assumes *parcs-xss*: *parcs* (*list-digraph* *G-list*) = \bigcup (*sset* *xss*)
begin

abbreviation (*input*) *G* ≡ *list-digraph* *G-list*

abbreviation (*input*) *M* ≡ *to-map'* (*snd* *G-list*) *xss*

lemma *edge-rev-simps*:

assumes (*u,v*) ∈ *parcs* *G* **shows** *edge-rev* *M* (*u,v*) = (*v,u*)
⟨*proof*⟩

end

sublocale *lists-digraph-map* ⊆ *digraph-map* *G* *M* ⟨*proof*⟩

sublocale *lists-digraph-map* ⊆ *pair-graph* *G*
⟨*proof*⟩

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

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

lemma *M-simps*:

edge-succ *M* = *lists-succ* *xss*
⟨*proof*⟩

lemma *lists-fc-succ-permutes*: *lists-fc-succ* *xss* *permutes* (\bigcup (*sset* *xss*))
⟨*proof*⟩

lemma *permutation-lists-fc-succ*[*intro*, *simp*]: *permutation* (*lists-fc-succ* *xss*)
⟨*proof*⟩

lemma *face-cycle-succ-conv*: *face-cycle-succ* = *lists-fc-succ* *xss*
⟨*proof*⟩

lemma *sset-lists-fcs*:

$sset (lists-fcs\ as) = \{face-cycle-set\ a \mid a. a \in set\ as\}$
<proof>

lemma *distincts-lists-fcs*: $distinct\ as \implies distincts\ (lists-fcs\ as)$

<proof>

lemma *face-cycle-set-ss*: $a \in parcs\ G \implies face-cycle-set\ a \subseteq parcs\ G$

<proof>

lemma *face-cycle-succ-neg*:

assumes $a \in parcs\ G$ **shows** $face-cycle-succ\ a \neq a$
<proof>

lemma *card-face-cycle-sets-conv*:

shows $card\ (pre-digraph-map.face-cycle-sets\ G\ M) = length\ (lists-fcs\ (remdups\ (snd\ G-list)))$
<proof>

end

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

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

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

It would suffice to check that $set\ (RTL.rtrancl-i\ A\ [u]) = set\ V$. 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**
 $sccs-verts-impl\ G \equiv set\ '(\lambda x. RTL.rtrancl-i\ (snd\ G)\ [x])\ 'set\ (fst\ G)$

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**

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

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>

lemma *isolated-verts-impl-correct*:

pre-digraph.isolated-verts (*list-digraph* *G*) = *set* (*isolated-verts-impl* *G*)

<proof>

lemma *pair-graph-impl-correct*[*code*]:

pair-graph (*list-digraph* *G*) = *pair-graph-impl* *G* (**is** ?*L* = ?*R*)

<proof>

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>

lemma *elems-all-maps-list*:

assumes $M \in set\ (all\text{-}maps\text{-}list\ G)$ *distinct* (*snd* *G*)

shows $\bigcup (sset\ M) = set\ (snd\ G)$

<proof>

lemma *comb-planar-impl-altdef*: *comb-planar-impl* *G* = $(\exists M \in set\ (all\text{-}maps\text{-}list\ G). genus\text{-}impl\ G\ M = 0)$

<proof>

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>

end

theory *Planar-Complete*

imports

Digraph-Map-Impl

begin

8 Kuratowski Graphs are not Combinatorially Planar

8.1 A concrete K5 graph

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

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

lemma $c\text{-}K5\text{-not-comb-planar}: \neg \text{comb-planar } c\text{-}K5$
 $\langle \text{proof} \rangle$

lemma $pverts\text{-}c\text{-}K5: pverts\ c\text{-}K5 = \{0..4\}$
 $\langle \text{proof} \rangle$

lemma $parcs\text{-}c\text{-}K5: parcs\ c\text{-}K5 = \{(u,v). u \in \{0..4\} \wedge v \in \{0..4\} \wedge u \neq v\}$
 $\langle \text{proof} \rangle$

lemmas $c\text{-}K5\text{-simps} = pverts\text{-}c\text{-}K5\ parcs\text{-}c\text{-}K5$

lemma $complete\text{-}c\text{-}K5: K_5\ c\text{-}K5$
 $\langle \text{proof} \rangle$

8.2 A concrete K33 graph

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

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

lemma $c\text{-}K33\text{-not-comb-planar}: \neg \text{comb-planar } c\text{-}K33$
 $\langle \text{proof} \rangle$

lemma $complete\text{-}c\text{-}K33: K_{3,3}\ c\text{-}K33$
 $\langle \text{proof} \rangle$

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 $\text{hom}: \text{digraph-isomorphism } \text{hom}$
assumes $f: f \in \text{arcs } G \rightarrow \text{arcs } G$ **and** $g: g \in \text{arcs } G \rightarrow \text{arcs } G$
shows $\text{wrap-iso-arcs } \text{hom } f (\text{wrap-iso-arcs } \text{hom } g\ x) = \text{wrap-iso-arcs } \text{hom } (f \circ g)\ x$
 $\langle \text{proof} \rangle$

lemma (in digraph-map) $\text{face-cycle-succ-iso}$:

assumes *hom: digraph-isomorphism hom x ∈ iso-arcs hom ‘ arcs G*
shows *pre-digraph-map.face-cycle-succ (map-iso hom) x = wrap-iso-arcs hom*
face-cycle-succ x
 ⟨*proof*⟩

lemma (in digraph-map) face-cycle-set-iso:
assumes *hom: digraph-isomorphism hom x ∈ iso-arcs hom ‘ 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*⟩

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
 ⟨*proof*⟩

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*⟩

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*
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma (in wf-digraph) comb-planar-iso:
assumes *digraph-isomorphism hom*
shows *comb-planar (app-iso hom G) ↔ comb-planar G*
 ⟨*proof*⟩

8.3.3 Completeness is a Graph Invariant

lemma (in loopfree-digraph) loopfree-digraphI-app-iso:
assumes *digraph-isomorphism hom*
shows *loopfree-digraph (app-iso hom G)*
 ⟨*proof*⟩

lemma (in nomulti-digraph) nomulti-digraphI-app-iso:
assumes *digraph-isomorphism hom*
shows *nomulti-digraph (app-iso hom G)*

<proof>

lemma (in *pre-digraph*) *symmetricI-app-iso*:
 assumes *digraph-isomorphism hom*
 assumes *symmetric G*
 shows *symmetric (app-iso hom G)*
<proof>

lemma (in *sym-digraph*) *sym-digraphI-app-iso*:
 assumes *digraph-isomorphism hom*
 shows *sym-digraph (app-iso hom G)*
<proof>

lemma (in *graph*) *graphI-app-iso*:
 assumes *digraph-isomorphism hom*
 shows *graph (app-iso hom G)*
<proof>

lemma (in *wf-digraph*) *graph-app-iso-eq*:
 assumes *digraph-isomorphism hom*
 shows *graph (app-iso hom G) \longleftrightarrow graph G*
<proof>

lemma (in *pre-digraph*) *arcs-ends-iso*:
 assumes *digraph-isomorphism hom*
 shows *arcs-ends (app-iso hom G) = $(\lambda(u,v). (iso-verts hom u, iso-verts hom v))$*
' arcs-ends G
<proof>

lemma *inj-onI-pair*:
 assumes *inj-on f S T $\subseteq S \times S$*
 shows *inj-on $(\lambda(u,v). (f u, f v)) T$*
<proof>

lemma (in *wf-digraph*) *complete-digraph-iso*:
 assumes *digraph-isomorphism hom*
 shows *$K_n (app-iso hom G) \longleftrightarrow K_n G$ (is ?L \longleftrightarrow ?R)*
<proof>

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 $(\mid$ *iso-verts* = *fv*, *iso-arcs* = *fa*,
 iso-head = *fv* *o* *head G* *o* *the-inv-into (arcs G) fa*,
 iso-tail = *fv* *o* *tail G* *o* *the-inv-into (arcs G) fa* \mid)

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

iso-verts (mk-iso fv fa) = fv
iso-arcs (mk-iso fv fa) = fa
 ⟨proof⟩

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)
 ⟨proof⟩

definition pairself f ≡ λx. case x of (u,v) ⇒ (f u, f v)

lemma inj-on-pairself:
assumes inj-on f S **and** T ⊆ S × S
shows inj-on (pairself f) T
 ⟨proof⟩

definition
 mk-iso-nomulti :: ('a,'b) pre-digraph ⇒ ('c,'d) pre-digraph ⇒ ('a ⇒ 'c) ⇒ ('a,
 'b, 'c, 'd) digraph-isomorphism

where

mk-iso-nomulti G H fv ≡ (
 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
 ⟨proof⟩

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)
 ⟨proof⟩

lemma complete-digraph-are-iso:
assumes K_n G K_n H **shows** digraph-iso G H
 ⟨proof⟩

lemma pairself-image-prod:

pairself $f \text{ ' } (A \times B) = f \text{ ' } A \times f \text{ ' } B$
 ⟨*proof*⟩

lemma *complete-bipartite-digraph-are-iso*:
 assumes $K_{m,n} G K_{m,n} H$ **shows** *digraph-iso* $G H$
 ⟨*proof*⟩

lemma *K5-not-comb-planar*:
 assumes $K_5 G$ **shows** $\neg \text{comb-planar } G$
 ⟨*proof*⟩

lemma *K33-not-comb-planar*:
 assumes $K_{3,3} G$ **shows** $\neg \text{comb-planar } G$
 ⟨*proof*⟩

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 $F :: 'a \text{ set}$ **and** $r :: 'a \text{ rel}$

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

⟨*proof*⟩

lemma *rtrancl-onE-ntrancl-onp*:

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

⟨*proof*⟩

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

⟨*proof*⟩

definition *nreachable* :: ('a,'b) *pre-digraph* \Rightarrow 'a \Rightarrow nat \Rightarrow 'a \Rightarrow bool (⟨ $\leftarrow \rightarrow^{-1} \rightarrow$ [100,100] 40) **where**

nreachable $G u n v \equiv \text{ntrancl-onp } (\text{verts } G) (\text{arcs-ends } G) n u v$

context *wf-digraph* **begin**

lemma *reachableE-nreachable*:

assumes $u \rightarrow^* v$ **obtains** n **where** $u \rightarrow^n v$
<proof>

lemma *converse-nreachable-cases*[*cases pred: nreachable*]:

assumes $u \rightarrow^n v$
obtains $(n \text{trancl-on-} 0) u = v \ n = 0 \ u \in \text{verts } G$
| $(n \text{trancl-on-Suc}) w \ m$ **where** $u \rightarrow w \ n = \text{Suc } m \ w \rightarrow^m v$
<proof>

lemma *converse-nreachable-induct*[*consumes 1, case-names base step, induct pred: reachable*]:

assumes *major*: $u \rightarrow^n_G v$
and cases: $v \in \text{verts } G \implies P \ 0 \ v$
 $\bigwedge n \ x \ y. \llbracket x \rightarrow_G y; y \rightarrow^n_G v; P \ n \ y \rrbracket \implies P \ (\text{Suc } n) \ x$
shows $P \ n \ u$
<proof>

lemma *converse-nreachable-induct-less*[*consumes 1, case-names base step, induct pred: reachable*]:

assumes *major*: $u \rightarrow^n_G v$
and cases: $v \in \text{verts } G \implies P \ 0 \ v$
 $\bigwedge n \ x \ y. \llbracket x \rightarrow_G y; y \rightarrow^n_G v; \bigwedge z \ m. m \leq n \implies (z \rightarrow^m_G v) \implies P \ m \ z \rrbracket \implies$
 $P \ (\text{Suc } n) \ x$
shows $P \ n \ u$
<proof>

end

end

theory *Permutations-2*

imports

HOL-Combinatorics.Permutations

Graph-Theory.Auxiliary

Executable-Permutations

begin

10 More

abbreviation *funswapid* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (**infix** $\langle \Rightarrow_F \rangle$ 90) **where**
 $x \Rightarrow_F y \equiv \text{transpose } x \ y$

lemma *in-funswapid-image-iff*: $x \in (a \Rightarrow_F b) \ 'S \longleftrightarrow (a \Rightarrow_F b) \ x \in S$
<proof>

lemma *bij-swap-compose*: $\text{bij } (x \Rightarrow_F y \circ f) \longleftrightarrow \text{bij } f$
<proof>

lemma *bij-eq-iff*:

assumes *bij f* **shows** $f x = f y \longleftrightarrow x = y$
<proof>

lemma *swap-swap-id[simp]*: $(x \Rightarrow_F y) ((x \Rightarrow_F y) z) = z$
<proof>

11 Modifying Permutations

definition *perm-swap* :: $'a \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$ **where**
 $perm\text{-}swap\ x\ y\ f \equiv x \Rightarrow_F y\ o\ f\ o\ x \Rightarrow_F y$

definition *perm-rem* :: $'a \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$ **where**
 $perm\text{-}rem\ x\ f \equiv if\ f\ x \neq x\ then\ x \Rightarrow_F f\ x\ o\ f\ else\ f$

An example:

$perm\text{-}rem\ 2\ (list\text{-}succ\ [1, 2, 3, 4])\ x = list\text{-}succ\ [1, 3, 4]\ x$

lemma *perm-swap-id[simp]*: $perm\text{-}swap\ a\ b\ id = id$
<proof>

lemma *perm-rem-permutes*:

assumes $f\ permutes\ S \cup \{x\}$
shows $perm\text{-}rem\ x\ f\ permutes\ S$
<proof>

lemma *perm-rem-same*:

assumes $bij\ f\ f\ y = y$ **shows** $perm\text{-}rem\ x\ f\ y = f\ y$
<proof>

lemma *perm-rem-simps*:

assumes *bij f*
shows
 $x = y \implies perm\text{-}rem\ x\ f\ y = x$
 $f\ y = x \implies perm\text{-}rem\ x\ f\ y = f\ x$
 $y \neq x \implies f\ y \neq x \implies perm\text{-}rem\ x\ f\ y = f\ y$
<proof>

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

lemma *perm-rem-conv*: $\bigwedge f\ x\ y.\ bij\ f \implies perm\text{-}rem\ x\ f\ y = ($
if $x = y$ *then* x
else if $f\ y = x$ *then* $f\ (f\ y)$
else $f\ y)$
<proof>

lemma *perm-rem-commutes*:

assumes *bij f shows perm-rem a (perm-rem b f) = perm-rem b (perm-rem a f)*
 ⟨proof⟩

lemma *perm-rem-id[simp]: perm-rem a id = id*
 ⟨proof⟩

lemma *perm-swap-comp: perm-swap a b (f ∘ g) x = perm-swap a b f (perm-swap a b g x)*
 ⟨proof⟩

lemma *bij-perm-swap-iff[simp]: bij (perm-swap a b f) ⟷ bij f*
 ⟨proof⟩

lemma *funpow-perm-swap: perm-swap a b f $\hat{\sim}$ n = perm-swap a b (f $\hat{\sim}$ n)*
 ⟨proof⟩

lemma *orbit-perm-swap: orbit (perm-swap a b f) x = (a \rightleftharpoons_F b) ‘ orbit f ((a \rightleftharpoons_F b) x)*
 ⟨proof⟩

lemma *has-dom-perm-swap: has-dom (perm-swap a b f) S = has-dom f ((a \rightleftharpoons_F b) ‘ S)*
 ⟨proof⟩

lemma *perm-restrict-dom-subset:*
assumes *has-dom f A shows perm-restrict f A = f*
 ⟨proof⟩

lemma *perm-swap-permutes2:*
assumes *f permutes ((x \rightleftharpoons_F y) ‘ S)*
shows *perm-swap x y f permutes S*
 ⟨proof⟩

12 Cyclic Permutations

lemma *cyclic-on-perm-swap:*
assumes *cyclic-on f S shows cyclic-on (perm-swap x y f) ((x \rightleftharpoons_F y) ‘ S)*
 ⟨proof⟩

lemma *orbit-perm-rem:*
assumes *bij f x ≠ y shows orbit (perm-rem y f) x = orbit f x - {y} (is ?L = ?R)*
 ⟨proof⟩

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})*
 ⟨proof⟩

```

lemma cyclic-on-perm-rem:
  assumes cyclic-on f S bij f S  $\neq \{x\}$  shows cyclic-on (perm-rem x f) (S - {x})
   $\langle$ proof $\rangle$ 

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  $\langle$ proof $\rangle$ 

sublocale subdiv1-contr  $\subseteq$  G: fin-digraph G
   $\langle$ proof $\rangle$ 

context subdiv1-contr begin

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

  lemma edge-rev-GM: edge-rev GM = rev-G
   $\langle$ proof $\rangle$ 

  lemma edge-succ-GM: edge-succ GM = perm-swap uw uv (perm-swap vw (rev-G
    uv) (fold perm-rem [wu, uv] (edge-succ HM)))
   $\langle$ proof $\rangle$ 

  lemma rev-H-eq-rev-G:
    assumes  $x \in \text{arcs } G - \{uv, vu\}$  shows rev-H x = rev-G x
   $\langle$ proof $\rangle$ 

  lemma edge-succ-permutes: edge-succ GM permutes arcs G
   $\langle$ proof $\rangle$ 

  lemma out-arcs-empty:

```

assumes $x \in \text{verts } G$
shows $\text{out-arcs } G \ x = \{\} \longleftrightarrow \text{out-arcs } H \ x = \{\}$
 $\langle \text{proof} \rangle$

lemma *cyclic-on-edge-succ*:
assumes $x \in \text{verts } G$ $\text{out-arcs } G \ x \neq \{\}$
shows $\text{cyclic-on } (\text{edge-succ } GM) (\text{out-arcs } G \ x)$
 $\langle \text{proof} \rangle$

lemma *digraph-map-GM*:
shows $\text{digraph-map } G \ GM$
 $\langle \text{proof} \rangle$

end

sublocale $\text{subdiv1-contr} \subseteq GM$: $\text{digraph-map } G \ GM$ $\langle \text{proof} \rangle$

context *subdiv1-contr* **begin**

lemma *reachableGD*:
assumes $x \rightarrow^*_G y$ **shows** $x \rightarrow^*_H y$
 $\langle \text{proof} \rangle$

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

lemma *proj-verts-H-in-G*: $x \in \text{verts } H \implies \text{proj-verts-H } x \in \text{verts } G$
 $\langle \text{proof} \rangle$

lemma *dominatesHD*:
assumes $x \rightarrow_H y$ **shows** $\text{proj-verts-H } x \rightarrow^*_G \text{proj-verts-H } y$
 $\langle \text{proof} \rangle$

lemma *reachableHD*:
assumes $\text{reach}: x \rightarrow^*_H y$ **shows** $\text{proj-verts-H } x \rightarrow^*_G \text{proj-verts-H } y$
 $\langle \text{proof} \rangle$

lemma *H-reach-conv*: $\bigwedge x \ y. x \rightarrow^*_H y \longleftrightarrow \text{proj-verts-H } x \rightarrow^*_G \text{proj-verts-H } y$
 $\langle \text{proof} \rangle$

lemma *sccs-eq*: $G.\text{sccs-verts} = ({}^{\circ}) \text{proj-verts-H } {}^{\circ} H.\text{sccs-verts}$ (**is** ?L = ?R)
 $\langle \text{proof} \rangle$

lemma *inj-on-proj-verts-H*: $\text{inj-on } ({}^{\circ}) \text{proj-verts-H}$ ($\text{pre-digraph.sccs-verts } H$)
 $\langle \text{proof} \rangle$

lemma *card-sccs-verts*: $\text{card } G.\text{sccs-verts} = \text{card } H.\text{sccs-verts}$
 $\langle \text{proof} \rangle$

lemma *card-sccs-eq*: $\text{card } G.\text{sccs} = \text{card } H.\text{sccs}$

<proof>

lemma *isolated-verts-eq*: $G.\text{isolated-verts} = H.\text{isolated-verts}$

<proof>

lemma *card-verts*: $\text{card } (\text{verts } H) = \text{card } (\text{verts } G) + 1$

<proof>

lemma *card-arcs*: $\text{card } (\text{arcs } H) = \text{card } (\text{arcs } G) + 2$

<proof>

lemma *edge-succ-wu*: $\text{edge-succ } HM \text{ } wu = wv$

<proof>

lemma *edge-succ-wv*: $\text{edge-succ } HM \text{ } wv = wu$

<proof>

lemmas $\text{edge-succ-}w = \text{edge-succ-}wu \text{ } \text{edge-succ-}wv$

lemma *H-face-cycle-succ*:

$H.\text{face-cycle-succ } uw = wv$

$H.\text{face-cycle-succ } wv = wu$

<proof>

lemma *H-edge-succ-tail-eqD*:

assumes $\text{edge-succ } HM \text{ } a = b$ **shows** $\text{tail } H \text{ } a = \text{tail } H \text{ } b$

<proof>

lemma *YYY*:

$(wu \Rightarrow_F wv) (\text{edge-succ } HM \text{ } wv) = (\text{edge-succ } HM \text{ } wv)$

$(wu \Rightarrow_F wv) (\text{edge-succ } HM \text{ } wu) = (\text{edge-succ } HM \text{ } wu)$

<proof>

Project arcs of H to corresponding arcs of G

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

$\text{proj-arcs-H } x \equiv$

$\text{if } x = uw \vee x = wv \text{ then } wv$

$\text{else if } x = wv \vee x = wu \text{ then } wu$

$\text{else } x$

Project arcs of G to corresponding arcs of H

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

$\text{proj-arcs-G } x \equiv$

$\text{if } x = uw \text{ then } wv$

$\text{else if } x = wv \text{ then } wu$

$\text{else } x$

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

$proj\text{-}arcs\text{-}H\ uw = uv$
 $proj\text{-}arcs\text{-}H\ vw = uv$
 $proj\text{-}arcs\text{-}H\ vw = vu$
 $proj\text{-}arcs\text{-}H\ wu = vu$
 $x \notin \{uw, vw, wu, uv\} \implies proj\text{-}arcs\text{-}H\ x = x$
 $a \in arcs\ G \implies proj\text{-}arcs\text{-}H\ a = a$
<proof>

lemma *proj-arcs-H-in-arcs-G*: $a \in arcs\ H \implies proj\text{-}arcs\text{-}H\ a \in arcs\ G$

<proof>

lemma *proj-arcs-eq-swap*:

assumes $a \notin \{uv, vu, wu, uv\}$
shows $proj\text{-}arcs\text{-}H\ a = (uw \Rightarrow_F uv \circ vw \Rightarrow_F vu)\ a$
<proof>

lemma *proj-arcs-G-simps*:

$proj\text{-}arcs\text{-}G\ uv = uw$
 $proj\text{-}arcs\text{-}G\ vu = vw$
 $a \notin \{uv, vu\} \implies proj\text{-}arcs\text{-}G\ a = a$
<proof>

lemma *proj-arcs-G-in-arcs-H*:

assumes $a \in arcs\ G$ **shows** $proj\text{-}arcs\text{-}G\ a \in arcs\ H$
<proof>

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

<proof>

lemma *fccs-proj-arcs-GH*:

assumes $a \in arcs\ H$ **shows** $H.\text{face-cycle-set}\ (proj\text{-}arcs\text{-}G\ (proj\text{-}arcs\text{-}H\ a)) = H.\text{face-cycle-set}\ a$
<proof>

lemma *H-face-cycle-succ-neq-uv*:

$a \notin \{uv, vu\} \implies H.\text{face-cycle-succ}\ a \notin \{uv, vu\}$
<proof>

lemma *face-cycle-succ-choose-inter*:

$\{H.\text{face-cycle-succ}\ uv, H.\text{face-cycle-succ}\ vw, H.\text{face-cycle-succ}\ wu, H.\text{face-cycle-succ}\ uv\} \cap \{uv, vu\} = \{\}$
<proof>

lemma *face-cycle-succ-choose-neq*:

$H.\text{face-cycle-succ}\ wu \notin \{wu, uv\}$
 $H.\text{face-cycle-succ}\ uv \notin \{wu, uv\}$
<proof>

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>

lemma

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

and

face-cycle-succ-vu: $GM.\text{face-cycle-succ } vu = \text{proj-arcs-}H (H.\text{face-cycle-succ } wu)$

<proof>

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

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

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

<proof>

lemmas $G.\text{face-cycle-succ} = \text{face-cycle-succ-}uw \text{ face-cycle-succ-}vu \text{ face-cycle-succ-not-}uw$

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

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

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

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

<proof>

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

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

assumes $x \in H.\text{face-cycle-set } a$

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

<proof>

lemma *G-fcs-eq*:

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

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

?L = ?R)

<proof>

lemma *H-fcs-eq*:

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

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

<proof>

lemma *face-cycle-sets*:

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

<proof>

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

<proof>

lemma *card-face-cycle-sets*: *card* *GM.face-cycle-sets* = *card* *H.face-cycle-sets*

<proof>

lemma *euler-char-eq*: $GM.euler-char = H.euler-char$
 ⟨proof⟩

lemma *euler-genus-eq*: $GM.euler-genus = H.euler-genus$
 ⟨proof⟩

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

⟨proof⟩

lemma *subdivision-genus*:

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

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

⟨proof⟩

lemma *subdivision-comb-planar*:

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

⟨proof⟩

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** $\neg x \rightarrow_G y$

⟨proof⟩

lemma (in *wf-digraph*) *reachable-refl-iff*: $u \rightarrow^* u \longleftrightarrow u \in \text{verts } G$

⟨proof⟩

context *digraph-map* **begin**

lemma *face-cycle-set-succ[simp]*: $\text{face-cycle-set } (face-cycle-succ a) = \text{face-cycle-set } a$

<proof>

lemma *face-cycle-succ-funpow-in[simp]:*

(face-cycle-succ $\overset{\sim}{\sim}$ n) a \in arcs G \longleftrightarrow a \in arcs G

<proof>

lemma *segment-face-cycle-x-x-eq:*

segment face-cycle-succ x x = face-cycle-set x - {x}

<proof>

lemma *fcs-x-eq-x: face-cycle-succ x = x \longleftrightarrow face-cycle-set x = {x} (is ?L \longleftrightarrow ?R)*

<proof>

end

lemma *(in bidirected-digraph) bidirected-digraph-del-arc:*

bidirected-digraph (pre-digraph.del-arc (pre-digraph.del-arc G (arev a)) a) (perm-restrict arev (arcs G - {a , arev a}))

<proof>

lemma *(in bidirected-digraph) bidirected-digraph-del-vert: bidirected-digraph (del-vert u) (perm-restrict arev (arcs (del-vert u)))*

<proof>

lemma *(in pre-digraph) ends-del-arc: arc-to-ends (del-arc u) = arc-to-ends G*

<proof>

lemma *(in pre-digraph) dominates-arcsD:*

assumes *v $\rightarrow_{del-arc u}$ w* **shows** *v \rightarrow_G w*

<proof>

lemma *(in wf-digraph) reachable-del-arcD:*

assumes *v $\rightarrow^*_{del-arc u}$ w* **shows** *v \rightarrow^*_G w*

<proof>

lemma *(in fin-digraph) finite-isolated-verts[intro!]: finite isolated-verts*

<proof>

lemma *(in wf-digraph) isolated-verts-in-sccs:*

assumes *u \in isolated-verts* **shows** *{u} \in sccs-verts*

<proof>

lemma *(in digraph-map) in-face-cycle-sets:*

a \in arcs G \implies face-cycle-set a \in face-cycle-sets

<proof>

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⟩

lemma (**in** *pre-digraph*) *casI-nth*:
assumes $p \neq []$ $u = \text{tail } G (\text{hd } p)$ $v = \text{head } G (\text{last } p) \wedge i. \text{Suc } i < \text{length } p \implies$
 $\text{head } G (p ! i) = \text{tail } G (p ! \text{Suc } i)$
shows $\text{cas } u \text{ } p \text{ } v$
 ⟨proof⟩

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** $\text{trail } (\text{tail } G a0) \text{ } p (\text{head } G an) \text{ } p \neq []$ $\text{hd } p = a0$ $\text{last } p = an$
 $\text{set } p \subseteq \text{face-cycle-set } a$
 ⟨proof⟩

lemma (**in** *digraph-map*) *obtain-trail-in-fcs'*:
assumes $a \in \text{arcs } G$ $u \in \text{tail } G \text{ ' face-cycle-set } a$ $v \in \text{tail } G \text{ ' face-cycle-set } a$
obtains p **where** $\text{trail } u \text{ } p \text{ } v \text{ set } p \subseteq \text{face-cycle-set } a$
 ⟨proof⟩

14.1 Deleting an isolated vertex

locale *del-vert-props* = *digraph-map* +
fixes u
assumes $u\text{-in}: u \in \text{verts } G$
assumes $u\text{-isolated}: \text{out-arcs } G \text{ } u = \{\}$

begin

lemma *u-isolated-in*: $\text{in-arcs } G \text{ } u = \{\}$
 ⟨proof⟩

lemma *arcs-dv*: $\text{arcs } (\text{del-vert } u) = \text{arcs } G$
 ⟨proof⟩

lemma *out-arcs-dv*: $\text{out-arcs } (\text{del-vert } u) = \text{out-arcs } G$
 ⟨proof⟩

lemma *digraph-map-del-vert*:
shows $\text{digraph-map } (\text{del-vert } u) \text{ } M$
 ⟨proof⟩

end

sublocale *del-vert-props* $\subseteq H$: *digraph-map del-vert u M* ⟨proof⟩

context *del-vert-props* **begin**

lemma *card-verts-dv*: $\text{card } (\text{verts } G) = \text{Suc } (\text{card } (\text{verts } (\text{del-vert } u)))$
<proof>

lemma *card-arcs-dv*: $\text{card } (\text{arcs } (\text{del-vert } u)) = \text{card } (\text{arcs } G)$
<proof>

lemma *isolated-verts-dv*: $H.\text{isolated-verts} = \text{isolated-verts} - \{u\}$
<proof>

lemma *u-in-isolated-verts*: $u \in \text{isolated-verts}$
<proof>

lemma *card-isolated-verts-dv*: $\text{card } \text{isolated-verts} = \text{Suc } (\text{card } H.\text{isolated-verts})$
<proof>

lemma *face-cycles-dv*: $H.\text{face-cycle-sets} = \text{face-cycle-sets}$
<proof>

lemma *euler-char-dv*: $\text{euler-char} = 1 + H.\text{euler-char}$
<proof>

lemma *adj-dv*: $v \rightarrow_{\text{del-vert } u} w \longleftrightarrow v \rightarrow_G w$
<proof>

lemma *reachable-del-vertD*:
assumes $v \rightarrow^*_{\text{del-vert } u} w$ **shows** $v \rightarrow^*_G w$
<proof>

lemma *reachable-del-vertI*:
assumes $v \rightarrow^*_G w$ $u \neq v \vee u \neq w$ **shows** $v \rightarrow^*_{\text{del-vert } u} w$
<proof>

lemma *G-reach-conv*: $v \rightarrow^*_G w \longleftrightarrow v \rightarrow^*_{\text{del-vert } u} w \vee (v = u \wedge w = u)$
<proof>

lemma *sccs-verts-dv*: $H.\text{sccs-verts} = \text{sccs-verts} - \{\{u\}\}$ (**is** ?L = ?R)
<proof>

lemma *card-sccs-verts-dv*: $\text{card } \text{sccs-verts} = \text{Suc } (\text{card } H.\text{sccs-verts})$
<proof>

lemma *card-sccs-dv*: $\text{card } \text{sccs} = \text{Suc } (\text{card } H.\text{sccs})$
<proof>

lemma *euler-genus-eq*: $H.\text{euler-genus} = \text{euler-genus}$
<proof>

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) *pre-digraph* **where**
 $H \equiv \text{pre-digraph.del-arc } (\text{pre-digraph.del-arc } G \ a') \ a$

definition *HM* :: 'b *pre-map* **where**
 $HM =$
 $($ *edge-rev* = *perm-restrict* (*edge-rev* *M*) (*arcs* *G* - {*a*, *a'*})
 $,$ *edge-succ* = *perm-rem* *a* (*perm-rem* *a'* (*edge-succ* *M*))
 $)$

lemma

verts-H: *verts* *H* = *verts* *G* **and**
arcs-H: *arcs* *H* = *arcs* *G* - {*a*, *a'*} **and**
tail-H: *tail* *H* = *tail* *G* **and**
head-H: *head* *H* = *head* *G* **and**
ends-H: *arc-to-ends* *H* = *arc-to-ends* *G* **and**
arcs-in: {*a*, *a'*} \subseteq *arcs* *G* **and**
ends-in: {*tail* *G* *a*, *head* *G* *a*} \subseteq *verts* *G*
 $\langle \text{proof} \rangle$

lemma *cyclic-on-edge-succ*:

assumes $x \in \text{verts } H$ *out-arcs* *H* $x \neq \{\}$
shows *cyclic-on* (*edge-succ* *HM*) (*out-arcs* *H* x)
 $\langle \text{proof} \rangle$

lemma *digraph-map*: *digraph-map* *H* *HM*

$\langle \text{proof} \rangle$

lemma *rev-H*: *bidel-arc.H* *G* *M* *a'* = *H* (**is** ?*t1*)

and *rev-HM*: *bidel-arc.HM* *G* *M* *a'* = *HM* (**is** ?*t2*)
 $\langle \text{proof} \rangle$

end

sublocale *bidel-arc* \subseteq *H*: *digraph-map* *H* *HM* $\langle \text{proof} \rangle$

context *bidel-arc* **begin**

lemma *a-neq-a'*: $a \neq a'$

$\langle \text{proof} \rangle$

lemma

arcs-G: $\text{arcs } G = \text{insert } a (\text{insert } a' (\text{arcs } H))$ **and**

arcs-not-in: $\{a, a'\} \cap \text{arcs } H = \{\}$

<proof>

lemma *card-arcs-da*: $\text{card } (\text{arcs } G) = 2 + \text{card } (\text{arcs } H)$

<proof>

lemma *cas-da*: $H.\text{cas} = G.\text{cas}$

<proof>

lemma *reachable-daD*:

assumes $v \rightarrow^*_H w$ **shows** $v \rightarrow^*_G w$

<proof>

lemma *not-G-isolated-a*: $\{\text{tail } G \ a, \text{head } G \ a\} \cap G.\text{isolated-verts} = \{\}$

<proof>

lemma *isolated-other-da*:

assumes $u \notin \{\text{tail } G \ a, \text{head } G \ a\}$ **shows** $u \in H.\text{isolated-verts} \longleftrightarrow u \in G.\text{isolated-verts}$

<proof>

lemma *isolated-da-pre*: $H.\text{isolated-verts} = G.\text{isolated-verts} \cup$

(if tail G a ∈ H.isolated-verts then {tail G a} else {}) \cup

(if head G a ∈ H.isolated-verts then {head G a} else {}) **(is ?L = ?R)**

<proof>

lemma *card-isolated-verts-da0*:

$\text{card } H.\text{isolated-verts} = \text{card } G.\text{isolated-verts} + \text{card } (\{\text{tail } G \ a, \text{head } G \ a\} \cap H.\text{isolated-verts})$

<proof>

lemma *segments-neq*:

assumes $\text{segment } G.\text{face-cycle-succ } a' \ a \neq \{\} \vee \text{segment } G.\text{face-cycle-succ } a \ a' \neq \{\}$

shows $\text{segment } G.\text{face-cycle-succ } a \ a' \neq \text{segment } G.\text{face-cycle-succ } a' \ a$

<proof>

lemma *H-fcs-eq-G-fcs*:

assumes $b \in \text{arcs } G \ \{b, G.\text{face-cycle-succ } b\} \cap \{a, a'\} = \{\}$

shows $H.\text{face-cycle-succ } b = G.\text{face-cycle-succ } b$

<proof>

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>

lemma *in-face-cycle-set-other*:
assumes $S \in G.\text{face-cycle-sets}$ $\{a, a'\} \cap S = \{\}$
shows $S \in H.\text{face-cycle-sets}$
 $\langle \text{proof} \rangle$

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 $H.\text{face-cycle-set } b \in G.\text{face-cycle-sets} - \{G.\text{face-cycle-set } a, G.\text{face-cycle-set } a'\}$
 $\langle \text{proof} \rangle$

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**
 $?L = ?R$)
 $\langle \text{proof} \rangle$

lemma *card-fcs-aa'-le*: $\text{card } \{G.\text{face-cycle-set } a, G.\text{face-cycle-set } a'\} \leq \text{card } G.\text{face-cycle-sets}$
 $\langle \text{proof} \rangle$

lemma *card-face-cycle-sets-da0*:
 $\text{card } H.\text{face-cycle-sets} = \text{card } G.\text{face-cycle-sets} - \text{card } \{G.\text{face-cycle-set } a, G.\text{face-cycle-set } a'\}$
 $+ \text{card } (H.\text{face-cycle-set } ' ((G.\text{face-cycle-set } a \cup G.\text{face-cycle-set } a') - \{a, a'\}))$
 $\langle \text{proof} \rangle$

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*: $a \in \text{orbit } G.\text{face-cycle-succ } a'$
 $\langle \text{proof} \rangle$

lemma *segment-a'-a-in*: $\text{segment } G.\text{face-cycle-succ } a' a \subseteq \text{arcs } H$ (**is** $?seg \subseteq -$)
 $\langle \text{proof} \rangle$

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** $?seg \in -$)
 $\langle \text{proof} \rangle$

lemma *segment-a-a'-neD*:
assumes $\text{segment } G.\text{face-cycle-succ } a a' \neq \{\}$
shows $\text{segment } G.\text{face-cycle-succ } a a' \in H.\text{face-cycle-sets}$
 $\langle \text{proof} \rangle$

lemma *H-fcs-full*:

assumes $SS \subseteq H.\text{face-cycle-sets}$ **shows** $H.\text{face-cycle-set } '(\bigcup SS) = SS$
<proof>

lemma *card-fcs-gt-0*: $0 < \text{card } G.\text{face-cycle-sets}$

<proof>

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>

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*: $a \notin G.\text{face-cycle-set } a' \ a' \notin G.\text{face-cycle-set } a$

<proof>

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>

lemma *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>

lemma *card-face-cycle-sets-da'*:

$\text{card } H.\text{face-cycle-sets} = \text{card } G.\text{face-cycle-sets} - \text{card } \{G.\text{face-cycle-set } a,$
 $G.\text{face-cycle-set } a'\} + (\text{if } S = \{\} \text{ then } 0 \text{ else } 1)$
<proof>

end

locale *bidel-arc-biconnected* = *bidel-arc* +

assumes *reach-a*: $\text{tail } G \ a \rightarrow^*_H \text{head } G \ a$

begin

lemma *reach-a'*: $\text{tail } G \ a' \rightarrow^*_H \text{head } G \ a'$

<proof>

lemma

tail-a': $\text{tail } G \ a' = \text{head } G \ a$ **and**

head-a': $\text{head } G \ a' = \text{tail } G \ a$

<proof>

lemma *reachable-daI*:

assumes $v \rightarrow^*_G w$ **shows** $v \rightarrow^*_H w$

<proof>

lemma *reachable-da*: $v \rightarrow^*_H w \longleftrightarrow v \rightarrow^*_G w$

<proof>

lemma *sccs-verts-da*: $H.\text{sccs-verts} = G.\text{sccs-verts}$

<proof>

lemma *card-sccs-da*: $\text{card } H.\text{sccs} = \text{card } G.\text{sccs}$

<proof>

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*: $G.\text{awalk } u \ p \ v \implies \{a, a'\} \cap \text{set } p = \{\} \implies H.\text{awalk } u \ p \ v$

<proof>

lemma *tail-neq-head*: $\text{tail } G \ a \neq \text{head } G \ a$

<proof>

lemma *scc-of-tail-neq-head*: $H.\text{scc-of } (\text{tail } G \ a) \neq H.\text{scc-of } (\text{head } G \ a)$

<proof>

lemma *scc-of-G-tail*:

assumes $u \in G.\text{scc-of } (\text{tail } G \ a)$

shows $H.\text{scc-of } u = H.\text{scc-of } (\text{tail } G \ a) \vee H.\text{scc-of } u = H.\text{scc-of } (\text{head } G \ a)$

<proof>

lemma *scc-of-other*:

assumes $u \notin G.\text{scc-of } (\text{tail } G \ a)$

shows $H.\text{scc-of } u = G.\text{scc-of } u$

<proof>

lemma *scc-of-tail-inter*:

$\text{tail } G \ a \in G.\text{scc-of } (\text{tail } G \ a) \cap H.\text{scc-of } (\text{tail } G \ a)$

<proof>

lemma *scc-of-head-inter*:

head G a $\in G.scc\text{-of } (tail\ G\ a) \cap H.scc\text{-of } (head\ G\ a)$
<proof>

lemma *G-scc-of-tail-not-in*: *G.scc-of (tail G a)* $\notin H.sccs\text{-verts}$

<proof>

lemma *H-scc-of-a-not-in*:

H.scc-of (tail G a) $\notin G.sccs\text{-verts}$ *H.scc-of (head G a)* $\notin G.sccs\text{-verts}$
<proof>

lemma *scc-verts-da*:

H.sccs-verts = (*G.sccs-verts* - {*G.scc-of (tail G a)*}) \cup {*H.scc-of (tail G a)*,
H.scc-of (head G a)} (**is** ?L = ?R)
<proof>

lemma *card-sccs-da*: *card H.sccs* = *Suc (card G.sccs)*

<proof>

end

sublocale *bidel-arc-not-biconnected* \subseteq *bidel-arc-same-face*

<proof>

locale *bidel-arc-tail-conn* = *bidel-arc* +

assumes *conn-tail*: *tail G a* $\notin H.isolated\text{-verts}$

locale *bidel-arc-head-conn* = *bidel-arc* +

assumes *conn-head*: *head G a* $\notin H.isolated\text{-verts}$

locale *bidel-arc-tail-isolated* = *bidel-arc* +

assumes *isolated-tail*: *tail G a* $\in H.isolated\text{-verts}$

locale *bidel-arc-head-isolated* = *bidel-arc* +

assumes *isolated-head*: *head G a* $\in H.isolated\text{-verts}$

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>

lemma *G-face-cycle-succ-a-no-loop*:

assumes *no-loop-a*: *head G a* \neq *tail G a* **shows** *G.face-cycle-succ a = a'*
<proof>

end

locale *bidel-arc-same-face-tail-conn* = *bidel-arc-same-face* + *bidel-arc-tail-conn*
begin

definition *a-neighbor* :: 'b **where**
a-neighbor \equiv *SOME* b. *G.face-cycle-succ* b = a

lemma *face-cycle-succ-a-neighbor*: *G.face-cycle-succ a-neighbor* = a
<proof>

lemma *a-neighbor-in*: *a-neighbor* \in *arcs* G
<proof>

lemma *a-neighbor-neq-a*: *a-neighbor* \neq a
<proof>

lemma *a-neighbor-neq-a'*: *a-neighbor* \neq a'
<proof>

lemma *edge-rev-a-neighbor-neq*: *edge-rev* M *a-neighbor* \neq a'
<proof>

lemma *edge-succ-a-neq*: *edge-succ* M a \neq a'
<proof>

lemma *H-face-cycle-succ-a-neighbor*: *H.face-cycle-succ a-neighbor* = *G.face-cycle-succ* a'
<proof>

lemma *H-fcs-a-neighbor*: *H.face-cycle-set a-neighbor* = *segment* *G.face-cycle-succ a'* a
(**is** ?L = ?R)
<proof>

end

locale *bidel-arc-isolated-loop* =
bidel-arc-biconnected + *bidel-arc-tail-isolated*
begin

lemma *loop-a[simp]*: *head* G a = *tail* G a
<proof>

end

sublocale *bidel-arc-isolated-loop* \subseteq *bidel-arc-head-isolated*
<proof>

context *bidel-arc-isolated-loop* **begin**

The edges a and a' form a loop on an otherwise isolated vertex

lemma *card-isolated-verts-da*: $\text{card } H.\text{isolated-verts} = \text{Suc } (\text{card } G.\text{isolated-verts})$
<proof>

lemma

G-edge-succ-a[simp]: $\text{edge-succ } M a = a'$ (**is** ?t1) **and**
G-edge-succ-a'[simp]: $\text{edge-succ } M a' = a$ (**is** ?t2)
<proof>

lemma

G-face-cycle-succ-a[simp]: $G.\text{face-cycle-succ } a = a$ **and**
G-face-cycle-succ-a'[simp]: $G.\text{face-cycle-succ } a' = a'$
<proof>

lemma

G-face-cycle-set-a[simp]: $G.\text{face-cycle-set } a = \{a\}$ **and**
G-face-cycle-set-a'[simp]: $G.\text{face-cycle-set } a' = \{a'\}$
<proof>

end

sublocale *bidel-arc-isolated-loop* \subseteq *bidel-arc-diff-face*
<proof>

context *bidel-arc-isolated-loop* **begin**

lemma *card-face-cycle-sets-da*: $\text{card } G.\text{face-cycle-sets} = \text{Suc } (\text{Suc } (\text{card } H.\text{face-cycle-sets}))$
<proof>

lemma *euler-genus-da*: $H.\text{euler-genus} = G.\text{euler-genus}$
<proof>

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*: $\text{head } G a \neq \text{tail } G a$
<proof>

lemma *card-isolated-verts-da*: $\text{card } H.\text{isolated-verts} = \text{Suc } (\text{Suc } (\text{card } G.\text{isolated-verts}))$

<proof>

lemma *G-edge-succ-a'[simp]: edge-succ M a' = a'*
<proof>

lemma *G-edge-succ-a[simp]: edge-succ M a = a*
<proof>

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
<proof>

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>

lemma *card-face-cycle-sets-da: card G.face-cycle-sets = Suc (card H.face-cycle-sets)*
<proof>

lemma *euler-genus-da: H.euler-genus = G.euler-genus*
<proof>

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*
<proof>

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)*
<proof>

lemma *segment-a'-a-ne: segment G.face-cycle-succ a' a \neq {}*
<proof>

lemma *segment-a-a'-e: segment G.face-cycle-succ a a' = {}*
<proof>

lemma *card-face-cycle-sets-da: card H.face-cycle-sets = card G.face-cycle-sets*
<proof>

lemma *euler-genus-da*: $H.euler-genus = G.euler-genus$
⟨*proof*⟩

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* $G M a'$
⟨*proof*⟩

interpretation *rev*: *bidel-arc-only-tail-not-isol* $G M a'$
⟨*proof*⟩

lemma *euler-genus-da*: $H.euler-genus = G.euler-genus$
⟨*proof*⟩

end

locale *bidel-arc-two-not-isol* = *bidel-arc-tail-not-isol* +
bidel-arc-head-conn

begin

lemma *isolated-verts-da*: $H.isolated-verts = G.isolated-verts$
⟨*proof*⟩

lemma *segment-a'-a-ne'*: *segment* $G.face-cycle-succ a' a \neq \{\}$
⟨*proof*⟩

interpretation *rev*: *bidel-arc-tail-not-isol* $G M a'$
⟨*proof*⟩

lemma *segment-a-a'-ne'*: *segment* $G.face-cycle-succ a a' \neq \{\}$
⟨*proof*⟩

lemma *card-face-cycle-sets-da*: $card H.face-cycle-sets = Suc (card G.face-cycle-sets)$
⟨*proof*⟩

lemma *euler-genus-da*: $H.euler-genus = G.euler-genus$
⟨*proof*⟩

end

locale *bidel-arc-biconnected-non-triv* = *bidel-arc-biconnected* +
bidel-arc-tail-conn

sublocale *bidel-arc-biconnected-non-triv* \subseteq *bidel-arc-head-conn*
<proof>

context *bidel-arc-biconnected-non-triv* **begin**

lemma *isolated-verts-da*: $H.isolated-verts = G.isolated-verts$
<proof>

end

locale *bidel-arc-biconnected-same* = *bidel-arc-biconnected-non-triv* +
bidel-arc-same-face

sublocale *bidel-arc-biconnected-same* \subseteq *bidel-arc-same-face-tail-conn*
<proof>

context *bidel-arc-biconnected-same* **begin**

interpretation *rev*: *bidel-arc-same-face-tail-conn* $G M a'$
<proof>

lemma *card-face-cycle-sets-da*: $Suc (card H.face-cycle-sets) \geq (card G.face-cycle-sets)$
<proof>

lemma *euler-genus-da*: $H.euler-genus \leq G.euler-genus$
<proof>

end

locale *bidel-arc-biconnected-diff* = *bidel-arc-biconnected-non-triv* +
bidel-arc-diff-face

begin

lemma *fcs-not-triv*: $G.face-cycle-set a \neq \{a\} \vee G.face-cycle-set a' \neq \{a'\}$
<proof>

lemma *S-ne*: $S \neq \{\}$
<proof>

lemma *card-face-cycle-sets-da*: $card G.face-cycle-sets = Suc (card H.face-cycle-sets)$
<proof>

lemma *euler-genus-da*: $H.euler-genus = G.euler-genus$
<proof>

end

context *bidel-arc* **begin**

lemma *euler-genus-da*: $H.euler-genus \leq G.euler-genus$
<proof>
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>

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>

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** $\{a, a'\} \subseteq$ *arcs G* - *arcs H* $a' \neq a$
tail G a' = head G a *head G a' = tail G a*
<proof>

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>

lemma (in *digraph-map*) *nonneg-euler-genus*: $0 \leq$ *euler-genus*
<proof>

lemma *subgraph-comb-planar*:

```

    assumes subgraph G H comb-planar H  $\exists$  rev. bidirected-digraph G rev shows
comb-planar G
  <proof>

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>

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
  <proof>

lemma annotate-named-loop-inv-fix:
  whileAnno b (named-loop name) V c = whileAnnoFix b I ( $\lambda$ -. V) ( $\lambda$ -. c)
  <proof>

lemma annotate-named-loop-var:
  whileAnno b (named-loop name) V' c = whileAnno b I V c
  <proof>

lemma annotate-named-loop-var-fix:
  whileAnno b (named-loop name) V' c = whileAnnoFix b I ( $\lambda$ -. V) ( $\lambda$ -. c)
  <proof>

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*}
⟨*proof*⟩

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* (λ*e*. *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  $u = u'\ v = v'$   
  <proof>
```

```
lemma awalk-nonempty-ends:  
  assumes  $p \neq []$  awalk  $u\ p\ v$  awalk  $u'\ p\ v'$   
  shows  $u = u'\ v = v'$   
  <proof>
```

```
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  $p$ )  
  <proof>
```

```
lemma (in wf-digraph) inner-verts-conv':  
  assumes awalk  $u\ p\ v\ 2 \leq \text{length}\ p$  shows inner-verts  $p = \text{awalk-verts}$  (head  $G$   
  (hd  $p$ )) (butlast (tl  $p$ ))  
  <proof>
```

```
lemma verts3-in-verts:  
  assumes  $x \in \text{verts3}\ G$  shows  $x \in \text{verts}\ G$   
  <proof>
```

```
lemma (in pair-graph) deg2-awalk-is-iapath:  
  assumes  $V: \text{verts3}\ G \subseteq V\ V \subseteq \text{pverts}\ G$   
  assumes  $p: \text{awalk}\ u\ p\ v\ \text{set}\ (\text{inner-verts}\ p) \cap V = \{\}$  progressing  $p$   
  assumes in-V:  $u \in V\ v \in V$   
  assumes  $u \neq v$   
  shows gen-iapath  $V\ u\ p\ v$   
  <proof>
```

```
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 $2 \leq \text{in-degree } G \ w$
 $\langle \text{proof} \rangle$

lemma (*in pair-pseudo-graph*) *gen-iapath-same2E*:
assumes *verts3* $G \subseteq V \ V \subseteq \text{pverts } G$
and *gen-iapath* $V \ u \ p \ v \ \text{gen-iapath } V \ w \ q \ x$
and $e \in \text{set } p \ e \in \text{set } q$
obtains $p = q$
 $\langle \text{proof} \rangle$

definition *mk-graph'* :: $I\text{Graph} \Rightarrow \text{ig-vertex pair-pre-digraph}$ **where**
 $\text{mk-graph}' \ IG \equiv (\text{pverts} = \text{set } (\text{ig-verts } IG), \text{parcs} = \text{set } (\text{ig-arcs } IG))$

definition *mk-graph* :: $I\text{Graph} \Rightarrow \text{ig-vertex pair-pre-digraph}$ **where**
 $\text{mk-graph } IG \equiv \text{mk-symmetric } (\text{mk-graph}' \ IG)$

lemma *verts-mkg'*: $\text{pverts } (\text{mk-graph}' \ G) = \text{set } (\text{ig-verts } G)$
 $\langle \text{proof} \rangle$

lemma *arcs-mkg'*: $\text{parcs } (\text{mk-graph}' \ G) = \text{set } (\text{ig-arcs } G)$
 $\langle \text{proof} \rangle$

lemmas *mkg'-simps* = *verts-mkg' arcs-mkg'*

lemma *verts-mkg*: $\text{pverts } (\text{mk-graph } G) = \text{set } (\text{ig-verts } G)$
 $\langle \text{proof} \rangle$

lemma *parcs-mk-symmetric-symcl*: $\text{parcs } (\text{mk-symmetric } G) = (\text{arcs-ends } G)^s$
 $\langle \text{proof} \rangle$

lemma *arcs-mkg*: $\text{parcs } (\text{mk-graph } G) = (\text{set } (\text{ig-arcs } G))^s$
 $\langle \text{proof} \rangle$

lemmas *mkg-simps* = *verts-mkg arcs-mkg*

definition *iadj* :: $I\text{Graph} \Rightarrow \text{ig-vertex} \Rightarrow \text{ig-vertex} \Rightarrow \text{bool}$ **where**
 $\text{iadj } G \ u \ v \equiv (u,v) \in \text{set } (\text{ig-arcs } G) \vee (v,u) \in \text{set } (\text{ig-arcs } G)$

definition *loop-free* $G \equiv (\forall e \in \text{parcs } G. \text{fst } e \neq \text{snd } e)$

lemma *ig-opposite-simps*:
 $\text{ig-opposite } G \ (u,v) \ u = v \ \text{ig-opposite } G \ (v,u) \ u = v$

<proof>

lemma *distinct-ig-verts:*

distinct (ig-verts G)

<proof>

lemma *set-ig-arcs-verts:*

assumes *IGraph-inv G (u,v) ∈ set (ig-arcs G)* **shows** *u ∈ set (ig-verts G) v ∈ set (ig-verts G)*

<proof>

lemma *IGraph-inv-conv:*

IGraph-inv G ↔ pair-fin-digraph (mk-graph' G)

<proof>

lemma *IGraph-inv-conv':*

IGraph-inv G ↔ pair-pseudo-graph (mk-graph G)

<proof>

lemma *iadj-io-edge:*

assumes *u ∈ set (ig-verts G) e ∈ set (ig-in-out-arcs G u)*

shows *iadj G u (ig-opposite G e u)*

<proof>

lemma *All-set-ig-verts: (∀ v ∈ set (ig-verts G). P v) ↔ (∀ i < ig-verts-cnt G. P (ig-verts G ! i))*

<proof>

lemma *IGraph-imp-ppd-mkg':*

assumes *IGraph-inv G* **shows** *pair-fin-digraph (mk-graph' G)*

<proof>

lemma *finite-symcl-iff: finite (R^s) ↔ finite R*

<proof>

lemma *(in pair-fin-digraph) pair-pseudo-graphI-mk-symmetric:*

pair-pseudo-graph (mk-symmetric G)

<proof>

lemma *IGraph-imp-ppg-mkg:*

assumes *IGraph-inv G* **shows** *pair-pseudo-graph (mk-graph G)*

<proof>

lemma *IGraph-lf-imp-pg-mkg:*

assumes *IGraph-inv G loop-free (mk-graph G)* **shows** *pair-graph (mk-graph G)*

<proof>

lemma *set-ig-arcs-imp-verts:*

assumes *(u,v) ∈ set (ig-arcs G) IGraph-inv G* **shows** *u ∈ set (ig-verts G) v ∈*

set (ig-verts G)
 ⟨proof⟩

lemma *iadj-imp-verts*:

assumes *iadj G u v IGraph-inv G* **shows** $u \in \text{set } (ig\text{-verts } G) \ v \in \text{set } (ig\text{-verts } G)$
 ⟨proof⟩

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⟩

lemma *iadjD*:

assumes *iadj G u v*
shows $\exists e \in \text{set } (ig\text{-in-out-arcs } G \ u). (e = (u,v) \vee e = (v,u))$
 ⟨proof⟩

lemma

ig-verts-empty[simp]: $ig\text{-verts } ig\text{-empty} = []$ **and**
ig-verts-add-e[simp]: $ig\text{-verts } (ig\text{-add-e } G \ u \ v) = ig\text{-verts } G$ **and**
ig-verts-add-v[simp]: $ig\text{-verts } (ig\text{-add-v } G \ v) = ig\text{-verts } G @ [v]$ (if $v \in \text{set } (ig\text{-verts } G)$) then [] else [v]
 ⟨proof⟩

lemma

ig-arcs-empty[simp]: $ig\text{-arcs } ig\text{-empty} = []$ **and**
ig-arcs-add-e[simp]: $ig\text{-arcs } (ig\text{-add-e } G \ u \ v) = ig\text{-arcs } G @ [(u,v)]$ **and**
ig-arcs-add-v[simp]: $ig\text{-arcs } (ig\text{-add-v } G \ v) = ig\text{-arcs } G$
 ⟨proof⟩

16.2 Total Correctness

16.2.1 Procedure *is-subgraph*

definition *is-subgraph-verts-inv* :: $IGraph \Rightarrow IGraph \Rightarrow nat \Rightarrow bool$ **where**
is-subgraph-verts-inv G H i $\equiv \text{set } (take \ i \ (ig\text{-verts } G)) \subseteq \text{set } (ig\text{-verts } H)$

definition *is-subgraph-arcs-inv* :: $IGraph \Rightarrow IGraph \Rightarrow nat \Rightarrow bool$ **where**
is-subgraph-arcs-inv G H i $\equiv \forall j < i. \text{let } (u,v) = ig\text{-arcs } G ! j \text{ in}$
 $((u,v) \in \text{set } (ig\text{-arcs } H) \vee (v,u) \in \text{set } (ig\text{-arcs } H))$
 $\wedge u \in \text{set } (ig\text{-verts } G) \wedge v \in \text{set } (ig\text{-verts } G)$

lemma *is-subgraph-verts-0*: $is\text{-subgraph-verts-inv } G \ H \ 0$
 ⟨proof⟩

lemma *is-subgraph-verts-step*:

assumes *is-subgraph-verts-inv G H i ig-verts G ! i* $i \in \text{set } (ig\text{-verts } H)$
assumes $i < \text{length } (ig\text{-verts } G)$
shows $is\text{-subgraph-verts-inv } G \ H \ (Suc \ i)$

<proof>

lemma *is-subgraph-verts-last:*

is-subgraph-verts-inv G H (length (ig-verts G)) \longleftrightarrow pverts (mk-graph G) \subseteq pverts (mk-graph H)

<proof>

lemma *is-subgraph-arcs-0: is-subgraph-arcs-inv G H 0*

<proof>

lemma *is-subgraph-arcs-step:*

assumes *is-subgraph-arcs-inv G H i*

e \in set (ig-arcs H) \vee (snd e, fst e) \in set (ig-arcs H)

fst e \in set (ig-verts G) snd e \in set (ig-verts G)

assumes *e = ig-arcs G ! i*

assumes *i < length (ig-arcs G)*

shows *is-subgraph-arcs-inv G H (Suc i)*

<proof>

lemma *wellformed-pseudo-graph-mkg:*

shows *pair-wf-digraph (mk-graph G) = pair-pseudo-graph(mk-graph G) (is ?L = ?R)*

<proof>

lemma *is-subgraph-arcs-last:*

is-subgraph-arcs-inv G H (length (ig-arcs G)) \longleftrightarrow parcs (mk-graph G) \subseteq parcs (mk-graph H) \wedge pair-pseudo-graph (mk-graph G)

<proof>

lemma *is-subgraph-verts-arcs-last:*

assumes *is-subgraph-verts-inv G H (ig-verts-cnt G)*

assumes *is-subgraph-arcs-inv G H (ig-arcs-cnt G)*

assumes *IGraph-inv H*

shows *subgraph (mk-graph G) (mk-graph H) (is ?T1)*

pair-pseudo-graph (mk-graph G) (is ?T2)

<proof>

lemma *is-subgraph-false:*

assumes *subgraph (mk-graph G) (mk-graph H)*

obtains *$\forall i < \text{length (ig-verts G)}. \text{ig-verts G ! } i \in \text{set (ig-verts H)}$*

$\forall i < \text{length (ig-arcs G)}. \text{let } (u,v) = \text{ig-arcs G ! } i \text{ in}$

$((u,v) \in \text{set (ig-arcs H)} \vee (v,u) \in \text{set (ig-arcs H)})$

$\wedge u \in \text{set (ig-verts G)} \wedge v \in \text{set (ig-verts G)}$

<proof>

lemma *(in is-subgraph-impl) is-subgraph-spec:*

$\forall \sigma. \Gamma \vdash_t \{ \sigma. \text{IGraph-inv } 'H \} 'R := \text{PROC is-subgraph}('G, 'H) \{ 'G = \sigma G \wedge 'H = \sigma H \wedge 'R = (\text{subgraph (mk-graph } 'G) (\text{mk-graph } 'H) \wedge \text{IGraph-inv } 'G) \}$

<proof>

16.2.2 Procedure *is-loop-free*

definition *is-loopfree-inv* $G\ k \equiv \forall j < k. \text{fst } (ig\text{-arcs } G\ !\ j) \neq \text{snd } (ig\text{-arcs } G\ !\ j)$

lemma *is-loopfree-0*:

is-loopfree-inv $G\ 0$
 ⟨*proof*⟩

lemma *is-loopfree-step1*:

assumes *is-loopfree-inv* $G\ n$
assumes $\text{fst } (ig\text{-arcs } G\ !\ n) \neq \text{snd } (ig\text{-arcs } G\ !\ n)$
assumes $n < ig\text{-arcs-cnt } G$
shows *is-loopfree-inv* $G\ (\text{Suc } n)$
 ⟨*proof*⟩

lemma *is-loopfree-step2*:

assumes *loop-free* (*mk-graph* G)
assumes $n < ig\text{-arcs-cnt } G$
shows $\text{fst } (ig\text{-arcs } G\ !\ n) \neq \text{snd } (ig\text{-arcs } G\ !\ n)$
 ⟨*proof*⟩

lemma *is-loopfree-last*:

assumes *is-loopfree-inv* $G\ (ig\text{-arcs-cnt } G)$
shows *loop-free* (*mk-graph* G)
 ⟨*proof*⟩

lemma (in *is-loopfree-impl*) *is-loopfree-spec*:

$\forall \sigma. \Gamma \vdash_t \{\sigma. IGraph\text{-inv } 'G\} 'R ::= PROC\ is\text{-loopfree}('G) \{\} 'G = \sigma\ G \wedge 'R$
 $= \text{loop-free } (mk\text{-graph } 'G) \{\}$
 ⟨*proof*⟩

16.2.3 Procedure *select-nodes*

definition *select-nodes-inv* $:: IGraph \Rightarrow IGraph \Rightarrow nat \Rightarrow bool$ **where**

select-nodes-inv $G\ H\ i \equiv \text{set } (ig\text{-verts } H) = \{v \in \text{set } (\text{take } i\ (ig\text{-verts } G)). \text{card } (ig\text{-neighbors } G\ v) \geq 3\} \wedge IGraph\text{-inv } H$

lemma *select-nodes-inv-step*:

fixes $G\ H\ i$
defines $v \equiv ig\text{-verts } G\ !\ i$
assumes $G\text{-inv}: IGraph\text{-inv } G$
assumes $sni\text{-inv}: \text{select-nodes-inv } G\ H\ i$
assumes $less: i < ig\text{-verts-cnt } G$
assumes $H': H' = (\text{if } 3 \leq \text{card } (ig\text{-neighbors } G\ v) \text{ then } ig\text{-add-v } H\ v \text{ else } H)$
shows *select-nodes-inv* $G\ H'\ (\text{Suc } i)$
 ⟨*proof*⟩

definition *select-nodes-prop* $:: IGraph \Rightarrow IGraph \Rightarrow bool$ **where**

select-nodes-prop $G\ H \equiv pverts\ (mk\text{-graph } H) = \text{verts3 } (mk\text{-graph } G)$

lemma (*in select-nodes-impl*) *select-nodes-spec*:
 $\forall \sigma. \Gamma \vdash_t \{\sigma. IGraph\text{-inv } 'G\} 'R := PROC \text{select-nodes}('G)$
 $\{\text{select-nodes-prop } \sigma G 'R \wedge IGraph\text{-inv } 'R \wedge \text{set}(\text{ig-arcs } 'R) = \{\}\}$
 $\langle \text{proof} \rangle$

16.2.4 Procedure *find-endpoint*

definition *find-endpoint-path-inv where*

find-endpoint-path-inv $G H \text{ len } u v w x \equiv$
 $\exists p. \text{pre-digraph.awalk}(\text{mk-graph } G) u p x \wedge \text{length } p = \text{len} \wedge$
 $\text{hd } p = (u, v) \wedge \text{last } p = (w, x) \wedge$
 $\text{set}(\text{pre-digraph.inner-verts}(\text{mk-graph } G) p) \cap \text{set}(\text{ig-verts } H) = \{\} \wedge$
progressing p

definition *find-endpoint-arcs-inv where*

find-endpoint-arcs-inv $G \text{ found } k v0 v1 v0' v1' \equiv$
 $(\text{found} \longrightarrow (\exists i < k. v1' = \text{ig-opposite } G (\text{ig-in-out-arcs } G v1 ! i) v1 \wedge v0' =$
 $v1 \wedge v0 \neq v1')) \wedge$
 $(\neg \text{found} \longrightarrow (\forall i < k. v0 = \text{ig-opposite } G (\text{ig-in-out-arcs } G v1 ! i) v1) \wedge v0 =$
 $v0' \wedge v1 = v1')$

lemma *find-endpoint-path-first*:

assumes *iadj* $G u v u \neq v IGraph\text{-inv } G$
shows *find-endpoint-path-inv* $G H (\text{Suc } 0) u v u v$
 $\langle \text{proof} \rangle$

lemma *find-endpoint-arcs-0*:

find-endpoint-arcs-inv $G \text{ False } 0 v0 v1 v0 v1$
 $\langle \text{proof} \rangle$

lemma *find-endpoint-path-lastE*:

assumes *find-endpoint-path-inv* $G H \text{ len } u v w x$
assumes *ig*: $IGraph\text{-inv } G$ **and** *lf*: *loop-free* $(\text{mk-graph } G)$
assumes *snp*: *select-nodes-prop* $G H$
assumes $0 < \text{len}$
assumes *u*: $u \in \text{set}(\text{ig-verts } H)$
obtains p **where** *pre-digraph.awalk* $(\text{mk-graph } G) u ((u, v) \# p) x$
and *progressing* $((u, v) \# p)$
and *set* $(\text{pre-digraph.inner-verts}(\text{mk-graph } G) ((u, v) \# p)) \cap \text{set}(\text{ig-verts } H)$
 $= \{\}$
and $\text{len} \leq \text{ig-verts-cnt } G$
 $\langle \text{proof} \rangle$

lemma *find-endpoint-path-last1*:

assumes *find-endpoint-path-inv* $G H \text{ len } u v w x$
assumes *ig*: $IGraph\text{-inv } G$ **and** *lf*: *loop-free* $(\text{mk-graph } G)$
assumes *snp*: *select-nodes-prop* $G H$
assumes $0 < \text{len}$
assumes *mem*: $u \in \text{set}(\text{ig-verts } H) x \in \text{set}(\text{ig-verts } H) u \neq x$

shows $\exists p. \text{pre-digraph.iapath } (mk\text{-graph } G) \ u \ ((u,v) \# p) \ x$
 <proof>

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* $(mk\text{-graph } G)$
assumes *snp*: *select-nodes-prop* $G \ H$
assumes $0 < len$
assumes *mem*: $u \in \text{set } (ig\text{-verts } H)$
assumes *iapath*: *pre-digraph.iapath* $(mk\text{-graph } G) \ u \ ((u,v) \# p) \ x$
shows *False*
 <proof>

lemma *find-endpoint-arcs-last*:

assumes *arcs*: *find-endpoint-arcs-inv* $G \ False \ (\text{length } (ig\text{-in-out-arcs } G \ v1)) \ v0 \ v1 \ v0a \ v1a$
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* $(mk\text{-graph } G)$
assumes *snp*: *select-nodes-prop* $G \ H$
assumes *mem*: $v\text{-tail} \in \text{set } (ig\text{-verts } H)$
assumes $0 < len$
shows $\neg \text{pre-digraph.iapath } (mk\text{-graph } G) \ v\text{-tail} \ ((v\text{-tail}, v\text{-next}) \# p) \ x$
 <proof>

lemma *find-endpoint-arcs-step1E*:

assumes *find-endpoint-arcs-inv* $G \ False \ k \ v0 \ v1 \ v0' \ v1'$
assumes *ig-opposite* $G \ (ig\text{-in-out-arcs } G \ v1 \ ! \ k) \ v1' \neq v0'$
obtains $v0 = v0' \ v1 = v1' \ \text{find-endpoint-arcs-inv } G \ True \ (Suc \ k) \ v0 \ v1 \ v1$
(ig-opposite $G \ (ig\text{-in-out-arcs } G \ v1 \ ! \ k) \ v1)$
 <proof>

lemma *find-endpoint-arcs-step2E*:

assumes *find-endpoint-arcs-inv* $G \ False \ k \ v0 \ v1 \ v0' \ v1'$
assumes *ig-opposite* $G \ (ig\text{-in-out-arcs } G \ v1 \ ! \ k) \ v1' = v0'$
obtains $v0 = v0' \ v1 = v1' \ \text{find-endpoint-arcs-inv } G \ False \ (Suc \ k) \ v0 \ v1 \ v0 \ v1$
 <proof>

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* $G \ True \ k \ w \ x \ w' \ x'$
 $k \leq \text{length } (ig\text{-in-out-arcs } G \ x)$
assumes *ig*: *IGraph-inv* G
assumes *not-end*: $x \notin \text{set } (ig\text{-verts } H)$
shows *find-endpoint-path-inv* $G \ H \ (Suc \ len) \ u \ v \ w' \ x'$
 <proof>

lemma *no-loop-path*:

assumes $u = v$ **and** *ig*: *IGraph-inv* G
shows $\neg (\exists p \ w. \ \text{pre-digraph.iapath } (mk\text{-graph } G) \ u \ ((u, v) \# p) \ w)$

$\langle \text{proof} \rangle$

lemma (in *find-endpoint-impl*) *find-endpoint-spec*:

$\forall \sigma. \Gamma \vdash_t \{ \sigma. \text{select-nodes-prop } 'G \ 'H \wedge \text{loop-free } (\text{mk-graph } 'G) \wedge 'v\text{-tail} \in \text{set } (\text{ig-verts } 'H) \wedge \text{iadj } 'G \ 'v\text{-tail } 'v\text{-next} \wedge \text{IGraph-inv } 'G \}$
 $'R := \text{PROC find-endpoint}('G, 'H, 'v\text{-tail}, 'v\text{-next})$
 $\{ \text{case } 'R \text{ of None} \Rightarrow \neg(\exists p \ w. \text{pre-digraph.iapath } (\text{mk-graph } ^\sigma G) \ ^\sigma v\text{-tail } ((^\sigma v\text{-tail},$
 $^\sigma v\text{-next}) \# p) \ w)$
 $| \text{Some } w \Rightarrow (\exists p. \text{pre-digraph.iapath } (\text{mk-graph } ^\sigma G) \ ^\sigma v\text{-tail } ((^\sigma v\text{-tail}, ^\sigma v\text{-next})$
 $\# p) \ w) \}$
 $\langle \text{proof} \rangle$

16.2.5 Procedure contract

definition *contract-iter-nodes-inv* **where**

$\text{contract-iter-nodes-inv } G \ H \ k \equiv$
 $\text{set } (\text{ig-arcs } H) = (\bigcup i < k. \{(u,v). u = (\text{ig-verts } H \ ! \ i) \wedge (\exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) \ u \ p \ v)\})$

definition *contract-iter-adj-inv* $:: \text{IGraph} \Rightarrow \text{IGraph} \Rightarrow \text{IGraph} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**

$\text{contract-iter-adj-inv } G \ H0 \ H \ u \ l \equiv (\text{set } (\text{ig-arcs } H) - (\{u\} \times \text{UNIV}) = \text{set } (\text{ig-arcs } H0)) \wedge$
 $\text{ig-verts } H = \text{ig-verts } H0 \wedge$
 $(\forall v. (u,v) \in \text{set } (\text{ig-arcs } H) \longleftrightarrow$
 $((\exists j < l. \exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) \ u \ ((u, \text{ig-opposite } G \ (\text{ig-in-out-arcs } G \ u \ ! \ j) \ u) \# p) \ v)))$

lemma *contract-iter-adj-invE*:

assumes $\text{contract-iter-adj-inv } G \ H0 \ H \ u \ l$

obtains $\text{set } (\text{ig-arcs } H) - (\{u\} \times \text{UNIV}) = \text{set } (\text{ig-arcs } H0) \ \text{ig-verts } H = \text{ig-verts } H0$

$\wedge v. (u,v) \in \text{set } (\text{ig-arcs } H) \longleftrightarrow ((\exists j < l. \exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) \ u \ ((u, \text{ig-opposite } G \ (\text{ig-in-out-arcs } G \ u \ ! \ j) \ u) \# p) \ v))$
 $\langle \text{proof} \rangle$

lemma *contract-iter-adj-inv-def'*:

$\text{contract-iter-adj-inv } G \ H0 \ H \ u \ l \longleftrightarrow ($
 $\text{set } (\text{ig-arcs } H) - (\{u\} \times \text{UNIV}) = \text{set } (\text{ig-arcs } H0)) \wedge \text{ig-verts } H = \text{ig-verts } H0 \wedge$
 $(\forall v. ((\exists j < l. \exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) \ u \ ((u, \text{ig-opposite } G \ (\text{ig-in-out-arcs } G \ u \ ! \ j) \ u) \# p) \ v) \longrightarrow (u,v) \in \text{set } (\text{ig-arcs } H)) \wedge$
 $((u,v) \in \text{set } (\text{ig-arcs } H) \longrightarrow ((\exists j < l. \exists p. \text{pre-digraph.iapath } (\text{mk-graph } G) \ u \ ((u, \text{ig-opposite } G \ (\text{ig-in-out-arcs } G \ u \ ! \ j) \ u) \# p) \ v))))$
 $\langle \text{proof} \rangle$

lemma *select-nodes-prop-add-e[simp]*:

$\text{select-nodes-prop } G \ (\text{ig-add-e } H \ u \ v) = \text{select-nodes-prop } G \ H$

$\langle \text{proof} \rangle$

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) w
shows *contract-iter-adj-inv* G H0 (*ig-add-e* H u w) u (*Suc* l)
⟨*proof*⟩

lemma *contract-iter-adj-inv-step2*:
assumes *ciai*: *contract-iter-adj-inv* G H0 H u l
assumes *iapath*: $\bigwedge p w. \neg \text{pre-digraph.iapath (mk-graph G) u ((u, \text{ig-opposite G (ig-in-out-arcs G u ! l) u) \# p) w}$
shows *contract-iter-adj-inv* G H0 H u (*Suc* l)
⟨*proof*⟩

definition *contract-iter-adj-prop where*
contract-iter-adj-prop G H0 H u \equiv *ig-verts* H = *ig-verts* H0
 \wedge *set* (*ig-arcs* H) = *set* (*ig-arcs* H0) \cup ($\{u\} \times \{v. \exists p. \text{pre-digraph.iapath (mk-graph G) u p v}\}$)

lemma *contract-iter-adj-propI*:
assumes *nodes*: *contract-iter-nodes-inv* G H i
assumes *ciai*: *contract-iter-adj-inv* G H H' u (*length* (*ig-in-out-arcs* G u))
assumes *u*: u = *ig-verts* H ! i
shows *contract-iter-adj-prop* G H H' u
⟨*proof*⟩

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*⟩

lemma *contract-iter-nodes-0*:
assumes *set* (*ig-arcs* H) = {} **shows** *contract-iter-nodes-inv* G H 0
⟨*proof*⟩

lemma *contract-iter-adj-0*:
assumes *nodes*: *contract-iter-nodes-inv* G H i
assumes *i*: i < *ig-verts-cnt* H
shows *contract-iter-adj-inv* G H H (*ig-verts* H ! i) 0
⟨*proof*⟩

lemma *snp-vertexes*:

assumes *select-nodes-prop* $G H u \in \text{set } (ig\text{-verts } H)$ **shows** $u \in \text{set } (ig\text{-verts } G)$
 $\langle \text{proof} \rangle$

lemma *igraph-ig-add-eI*:
assumes *IGraph-inv* G
assumes $u \in \text{set } (ig\text{-verts } G) v \in \text{set } (ig\text{-verts } G)$
shows *IGraph-inv* (*ig-add-e* $G u v$)
 $\langle \text{proof} \rangle$

lemma *snp-iapath-ends-in*:
assumes *select-nodes-prop* $G H$
assumes *pre-digraph.iapath* (*mk-graph* G) $u p v$
shows $u \in \text{set } (ig\text{-verts } H) v \in \text{set } (ig\text{-verts } H)$
 $\langle \text{proof} \rangle$

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'* $H = \text{contr-graph } (mk\text{-graph } G)$ (**is** ?*t1*)
and *symmetric* (*mk-graph'* H) (**is** ?*t2*)
 $\langle \text{proof} \rangle$

lemma (**in** *contract-impl*) *contract-spec*:
 $\forall \sigma. \Gamma \vdash_t \{ \sigma. \text{select-nodes-prop } 'G 'H \wedge \text{IGraph-inv } 'G \wedge \text{loop-free } (mk\text{-graph } 'G) \wedge \text{IGraph-inv } 'H \wedge \text{set } (ig\text{-arcs } 'H) = \{ \} \}$
 $'R ::= \text{PROC } \text{contract} ('G, 'H)$
 $\{ 'G = 'G \wedge mk\text{-graph}' 'R = \text{contr-graph } (mk\text{-graph } 'G) \wedge \text{symmetric } (mk\text{-graph}' 'R) \wedge \text{IGraph-inv } 'R \}$
 $\langle \text{proof} \rangle$

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 $G u k \text{blue} \equiv \forall v \in \text{set } (ig\text{-verts } G). \text{blue } v \longleftrightarrow$
 $(\exists i < k. v = \text{ig-opposite } G (ig\text{-in-out-arcs } G u ! i) u)$

definition *is-K33-component-size-inv* :: *IGraph* \Rightarrow *nat* \Rightarrow (*ig-vertex* \Rightarrow *bool*) \Rightarrow *nat* \Rightarrow *bool* **where**
is-K33-component-size-inv $G k \text{blue } cnt \equiv cnt = \text{card } \{ i. i < k \wedge \text{blue } (ig\text{-verts } G ! i) \}$

definition *is-K33-outer-inv* :: *IGraph* \Rightarrow *nat* \Rightarrow (*ig-vertex* \Rightarrow *bool*) \Rightarrow *bool* **where**
is-K33-outer-inv $G k \text{blue} \equiv \forall i < k. \forall v \in \text{set } (ig\text{-verts } G).$
 $\text{blue } (ig\text{-verts } G ! i) = \text{blue } v \longleftrightarrow (ig\text{-verts } G ! i, v) \notin \text{set } (ig\text{-arcs } G)$

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 (λ -. *False*)
 ⟨*proof*⟩

lemma *is-K33-component-size-0*: *is-K33-component-size-inv* G 0 *blue* 0
 ⟨*proof*⟩

lemma *is-K33-outer-0*: *is-K33-outer-inv* G 0 *blue*
 ⟨*proof*⟩

lemma *is-K33-inner-0*: *is-K33-inner-inv* G k 0 *blue*
 ⟨*proof*⟩

lemma *is-K33-colorize-last*:
assumes $u \in \text{set } (\text{ig-verts } G)$
shows *is-K33-colorize-inv* G u (*length* (*ig-in-out-arcs* G u)) *blue*
 = ($\forall v \in \text{set } (\text{ig-verts } G)$. *blue* $v \longleftrightarrow$ *iadj* G u v) (**is** ? L = ? R)
 ⟨*proof*⟩

lemma *is-K33-component-size-last*:
assumes $k = \text{ig-verts-cnt } G$
shows *is-K33-component-size-inv* G k *blue* $\text{cnt} \longleftrightarrow \text{card } \{u \in \text{set } (\text{ig-verts } G).$
blue $u\} = \text{cnt}$
 ⟨*proof*⟩

lemma *is-K33-outer-last*:
is-K33-outer-inv G (*ig-verts-cnt* G) *blue* $\longleftrightarrow (\forall u \in \text{set } (\text{ig-verts } G).$ $\forall v \in \text{set}$
 (*ig-verts* G).
blue $u = \text{blue } v \longleftrightarrow (u, v) \notin \text{set } (\text{ig-arcs } G))$
 ⟨*proof*⟩

lemma *is-K33-inner-last*:
is-K33-inner-inv G k (*ig-verts-cnt* G) *blue* $\longleftrightarrow (\forall v \in \text{set } (\text{ig-verts } G).$
blue (*ig-verts* G ! k) = *blue* $v \longleftrightarrow (\text{ig-verts } G$! k , $v) \notin \text{set } (\text{ig-arcs } G))$
 ⟨*proof*⟩

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*))
 ⟨*proof*⟩

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*⟩

lemma *is-K33-component-size-step2*:
assumes *comp:is-K33-component-size-inv* G k *blue* *blue-cnt*
assumes *blue*: \neg *blue* (*ig-verts* G ! k)
shows *is-K33-component-size-inv* G (*Suc* k) *blue* *blue-cnt*
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *K33-mkg'I*:
fixes G *col* *cnt*
defines $u \equiv$ *ig-verts* G ! 0
assumes *ig*: *IGraph-inv* G
assumes *iv-cnt*: *ig-verts-cnt* $G = 6$ **and** *c1-cnt*: *cnt = 3*
assumes *colorize*: *is-K33-colorize-inv* G u (*length* (*ig-in-out-arcs* G u)) *blue*
assumes *comp*: *is-K33-component-size-inv* G (*ig-verts-cnt* G) *blue* *cnt*
assumes *outer*: *is-K33-outer-inv* G (*ig-verts-cnt* G) *blue*
shows $K_{3,3}$ (*mk-graph'* G)
 ⟨*proof*⟩

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 : $u \in$ *set* (*ig-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*⟩

lemma *K33-card*:
assumes $K_{3,3}$ (*mk-graph'* G) **shows** *ig-verts-cnt* $G = 6$
 ⟨*proof*⟩

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*

$(ig\text{-in-out-arcs } G (ig\text{-verts } G ! 0))$ *blue*

abbreviation (*input*) *is-K33-component-size-inv-last* :: $I\text{Graph} \Rightarrow (ig\text{-vertex} \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
is-K33-component-size-inv-last G *blue* $\equiv is\text{-K33-component-size-inv } G (ig\text{-verts-cnt } G)$ *blue* \exists

lemma *is-K33-outerD*:

assumes *is-K33-outer-inv* $G (ig\text{-verts-cnt } G)$ *blue*
assumes $i < ig\text{-verts-cnt } G$ $j < ig\text{-verts-cnt } G$
shows $(blue (ig\text{-verts } G ! i) = blue (ig\text{-verts } G ! j)) \longleftrightarrow (ig\text{-verts } G ! i, ig\text{-verts } G ! j) \notin set (ig\text{-arcs } G)$
 $\langle proof \rangle$

lemma (**in** *is-K33-impl*) *is-K33-spec*:

$\forall \sigma. \Gamma \vdash_t \{ \sigma. I\text{Graph-inv } 'G \wedge symmetric (mk\text{-graph}' 'G) \}$
 $'R ::= PROC is\text{-K33}('G)$
 $\{ 'G = {}^\sigma G \wedge 'R = K_{3,3}(mk\text{-graph}' 'G) \}$
 $\langle proof \rangle$

16.2.7 Procedure *is-K5*

definition

is-K5-outer-inv G $k \equiv \forall i < k. \forall v \in set (ig\text{-verts } G). ig\text{-verts } G ! i \neq v$
 $\longleftrightarrow (ig\text{-verts } G ! i, v) \in set (ig\text{-arcs } G)$

definition

is-K5-inner-inv G k $l \equiv \forall j < l. ig\text{-verts } G ! k \neq ig\text{-verts } G ! j$
 $\longleftrightarrow (ig\text{-verts } G ! k, ig\text{-verts } G ! j) \in set (ig\text{-arcs } G)$

lemma *K5-card*:

assumes $K_5 (mk\text{-graph}' G)$ **shows** $ig\text{-verts-cnt } G = 5$
 $\langle proof \rangle$

lemma *is-K5-inner-0*: *is-K5-inner-inv* G k 0

$\langle proof \rangle$

lemma *is-K5-inner-last*:

assumes $l = ig\text{-verts-cnt } G$
shows *is-K5-inner-inv* G k $l \longleftrightarrow (\forall v \in set (ig\text{-verts } G). ig\text{-verts } G ! k \neq v$
 $\longleftrightarrow (ig\text{-verts } G ! k, v) \in set (ig\text{-arcs } G))$
 $\langle proof \rangle$

lemma *is-K5-outer-step*:

assumes *is-K5-outer-inv* G k
assumes *is-K5-inner-inv* G k $(ig\text{-verts-cnt } G)$
shows *is-K5-outer-inv* G $(Suc k)$
 $\langle proof \rangle$

lemma *is-K5-outer-last*:

assumes *is-K5-outer-inv* G (*ig-verts-cnt* G)

assumes *IGraph-inv* G *ig-verts-cnt* $G = 5$ *symmetric* (*mk-graph'* G)

shows K_5 (*mk-graph'* G)

<proof>

lemma *is-K5-inner-step*:

assumes *is-K5-inner-inv* G k l

assumes $k < \text{ig-verts-cnt } G$

assumes $k \neq l \longleftrightarrow (\text{ig-verts } G \ ! \ k, \text{ig-verts } G \ ! \ l) \in \text{set } (\text{ig-arcs } G)$

shows *is-K5-inner-inv* G k (*Suc* l)

<proof>

lemma *iK5E*:

assumes K_5 (*mk-graph'* G)

obtains *ig-verts-cnt* $G = 5$ $\llbracket i < \text{ig-verts-cnt } G; j < \text{ig-verts-cnt } G \rrbracket \implies i \neq j$

$\longleftrightarrow (\text{ig-verts } G \ ! \ i, \text{ig-verts } G \ ! \ j) \in \text{set } (\text{ig-arcs } G)$

<proof>

lemma (**in** *is-K5-impl*) *is-K5-spec*:

$\forall \sigma. \Gamma \vdash_t \{ \sigma. \text{IGraph-inv } 'G \wedge \text{symmetric } (\text{mk-graph}' \ 'G) \}$

$'R := \text{PROC } \text{is-K5} ('G)$

$\{ 'G = \sigma G \wedge 'R = K_5(\text{mk-graph}' \ 'G) \}$

<proof>

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 $\neg \text{kuratowski-planar } G$

<proof>

definition *witness* $:: 'a$ *pair-pre-digraph* $\Rightarrow 'a$ *pair-pre-digraph* $\Rightarrow \text{bool}$ **where**

witness G $K \equiv \text{loop-free } K \wedge \text{pair-pseudo-graph } K \wedge \text{subgraph } K$ G

$\wedge (K_{3,3} (\text{contr-graph } K) \vee K_5 (\text{contr-graph } K))$

lemma *witness* (*mk-graph* G) (*mk-graph* K) $\longleftrightarrow \text{pair-pre-digraph.certify } (\text{mk-graph } G)$ (*mk-graph* K) $\wedge \text{loop-free } (\text{mk-graph } K)$

<proof>

lemma *pwd-imp-ppg-mkg*:

assumes *pair-wf-digraph* (*mk-graph* G)

shows *pair-pseudo-graph* (*mk-graph* G)

<proof>

theorem (in *check-kuratowski-impl*) *check-kuratowski-spec*:
 $\forall \sigma. \Gamma \vdash_t \{ \sigma. \text{pair-wf-digraph } (\text{mk-graph } 'G) \}$
 $'R := \text{PROC check-kuratowski}('G, 'K)$
 $\{ 'G = \sigma G \wedge 'K = \sigma K \wedge 'R \longleftrightarrow \text{witness } (\text{mk-graph } 'G) (\text{mk-graph } 'K) \}$
 $\langle \text{proof} \rangle$

lemma *check-kuratowski-correct*:
assumes *pair-pseudo-graph* G
assumes *witness* $G K$
shows $\neg \text{kuratowski-planar } G$
 $\langle \text{proof} \rangle$

lemma *check-kuratowski-correct-comb*:
assumes *pair-pseudo-graph* G
assumes *witness* $G K$
shows $\neg \text{comb-planar } G$
 $\langle \text{proof} \rangle$

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* $G K$
 $\langle \text{proof} \rangle$

end
theory *AutoCorres-Misc* **imports**
 $\dots / \text{lv} / \text{lib} / \text{OptionMonadWP}$
begin

17 Auxilliary Lemmas for Autocorres

17.1 Option monad

definition *owhile-inv* :: $('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow ('s, 'a) \text{lookup}) \Rightarrow 'a \Rightarrow ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow 'a \text{ rel} \Rightarrow ('s, 'a) \text{lookup}$ **where**
 $\text{owhile-inv } c \ b \ a \ I \ R \equiv \text{owhile } c \ b \ a$

lemma *owhile-unfold*: $\text{owhile } C \ B \ r \ s = \text{ocondition } (C \ r) (B \ r \ |>> \text{owhile } C \ B)$
 $(\text{oreturn } r) \ s$
 $\langle \text{proof} \rangle$

lemma *ovalidNF-owhile*:
assumes $\bigwedge s. P \ r \ s \Longrightarrow I \ r \ s$
and $\bigwedge r \ s. \text{ovalidNF } (\lambda s'. I \ r \ s' \wedge C \ r \ s' \wedge s' = s) (B \ r) (\lambda r' s'. I \ r' \ s' \wedge (r', r) \in R)$
and *wf* R
and $\bigwedge r \ s. I \ r \ s \Longrightarrow \neg C \ r \ s \Longrightarrow Q \ r \ s$
shows $\text{ovalidNF } (P \ r) (\text{OptionMonad.owhile } C \ B \ r) \ Q$

<proof>

lemma *ovalidNF-owhile-inv*[*wf*]:

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*

<proof>

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>

named-theorems *vcg-l*

named-theorems *vcg-l-comb*

named-theorems *vcg-elim*

named-theorems *vcg-simp*

<ML>

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* $\equiv P$

named-theorems *branch-l*

named-theorems *branch-l-comb*

context begin

interpretation *Labeling-Syntax* *<proof>*

lemma *DC-if*[*branch-l*]:

fixes *ct* **defines** *ct'* $\equiv \lambda pos \text{ name. } (name, pos, []) \# ct$

assumes $a \implies C\langle \text{Suc } inp, ct' \text{ inp } \text{"then"}, outp': b \rangle$
assumes $\neg a \implies C\langle \text{Suc } outp', ct' \text{ outp}' \text{"else"}, outp: c \rangle$
shows $C\langle inp, ct, outp: \text{BRANCH (if } a \text{ then } b \text{ else } c) \rangle$
 $\langle \text{proof} \rangle$

lemma *DC-final*:
assumes $V\langle \text{"g"}, inp, [] \rangle, ct: a$
shows $C\langle inp, ct, \text{Suc } inp: a \rangle$
 $\langle \text{proof} \rangle$

end

$\langle \text{ML} \rangle$

method *branch-casify* = ((*rule Initial-Label*, *branch-l*; (*rule DC-final*)?), *casify*)

18.2 Labelled VCG theorems for the option monad

definition

$lpred\text{-conj} :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool})$ (**infixr** $\langle \text{land} \rangle$ 35)

where

$lpred\text{-conj } P \ Q \equiv \lambda x. P \ x \wedge Q \ x$

context begin

interpretation *Labeling-Syntax* $\langle \text{proof} \rangle$

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* ($\lambda \cdot R$))
shows *CTXT IC CT OC* (*ovalidNF P* ($f \ |>> K\text{-bind } g$) *Q*)
 $\langle \text{proof} \rangle$

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*)
 $\langle \text{proof} \rangle$

lemma *L-ovalidNF-obind* [*vcg-l*]:
assumes $\bigwedge r. \text{CTXT } (\text{Suc } OC1) (\text{"bind"}, \text{Suc } OC1, [\text{VAR } r]) \# \text{CT} \ OC$
 $(\text{ovalidNF } (R \ r) (g \ r) \ Q)$
and *CTXT IC CT OC1* (*ovalidNF P f R*)
shows *CTXT IC CT OC* (*ovalidNF P* ($f \ |>> (\lambda r. g \ r)$) *Q*)
 $\langle \text{proof} \rangle$

lemma *ovalidNF-K-bind* [*vcg-l*]:
assumes *CTXT IC CT OC* (*ovalidNF P f Q*)
shows *CTXT IC CT OC* (*ovalidNF P* (*K-bind* $f \ x$) *Q*)
 $\langle \text{proof} \rangle$

lemma *L-ovalidNF-prod-case*[vcg-l]:
assumes $\bigwedge x y. \text{SPLIT } v (x,y) \implies \text{CTXT IC CT OC } (\text{ovalidNF } (P x y) (B x y) Q)$
shows $\text{CTXT IC CT OC } (\text{ovalidNF } (\text{case } v \text{ of } (x, y) \Rightarrow P x y) (\text{case } v \text{ of } (x, y) \Rightarrow B x y) Q)$
 $\langle \text{proof} \rangle$

lemma *L-ovalidNF-oreturn-NF*[vcg-l]:
shows $\text{CTXT IC CT IC } (\text{ovalidNF } (P x) (\text{oreturn } x) P)$
 $\langle \text{proof} \rangle$

lemma *L-ovalidNF-owhile-inv*[vcg-l]:
fixes CT IC
defines $\text{CT}' \equiv \lambda r. (\text{"while"}, \text{IC}, [\text{VAR } r]) \# \text{CT}$
assumes $\bigwedge r s. \text{CTXT IC } (\text{"invariant"}, \text{IC}, [\text{VAR } s]) \# \text{CT}' r) \text{OC}$
 $(\text{ovalidNF}$
 $(\text{BIND "loop-inv" IC } (I r) \text{land}$
 $\text{BIND "loop-cond" IC } (C r) \text{land}$
 $\text{BIND "loop-var" IC } (\lambda s'. s' = s))$
 $(B r)$
 $(\lambda r'. \text{BIND "inv" IC } (I r') \text{land BIND "var" IC } (\lambda -. (r', r) \in R)))$
and $\bigwedge r. \text{VC } (\text{"wf"}, \text{OC}, []) (\text{CT}' r) (\text{wf } R)$
and $\bigwedge r s. I r s \implies \neg C r s \implies$
 $\text{VC } (\text{"postcondition"}, \text{Suc OC}, [\text{VAR } s]) (\text{CT}' r) (Q r s)$
shows $\text{CTXT IC CT } (\text{Suc OC}) (\text{ovalidNF } (I r) (\text{owhile-inv } C B r I R) Q)$
 $\langle \text{proof} \rangle$

lemma *L-ovalidNF-wp-comb2*[vcg-l-comb]:
assumes $\text{CTXT IC CT OC } (\text{ovalidNF } P f Q)$
and $\bigwedge s. P' s \implies \text{VC } (\text{"weaken"}, \text{IC}, [\text{VAR } s]) \text{CT } (P s)$
shows $\text{CTXT IC CT OC } (\text{ovalidNF } P' f Q)$
 $\langle \text{proof} \rangle$

lemma *L-condition-NF-wp*[vcg-l]:
fixes CT IC
defines $\text{CT}' \equiv (\text{"if"}, \text{IC}, []) \# \text{CT}$
assumes $\text{CTXT IC } (\text{"then"}, \text{IC}, []) \# \text{CT}' \text{OC1 } (\text{ovalidNF } L l Q)$
and $\text{CTXT } (\text{Suc OC1}) (\text{"else"}, \text{Suc OC1}, []) \# \text{CT}' \text{OC } (\text{ovalidNF } R r Q)$
shows $\text{CTXT IC CT OC } (\text{ovalidNF } (\lambda s. \text{BRANCH } (\text{if } C s \text{ then } L s \text{ else } R s))$
 $(\text{condition } C l r) Q)$
 $\langle \text{proof} \rangle$

lemma *L-ogets-NF-wp*[vcg-l]: $\text{CTXT IC CT IC } (\text{ovalidNF } (\lambda s. P (f s) s) (\text{ogets } f) P)$
 $\langle \text{proof} \rangle$

lemma *elim-land*[vcg-elim]:
assumes $(P \text{ land } Q) s$ **obtains** $P s Q s$
 $\langle \text{proof} \rangle$

```

lemma simp-bind[vcg-simp]:  $BIND\ ct\ n\ P\ s \longleftrightarrow BIND\ ct\ n\ (P\ s)$ 
  <proof>

lemma simp-land[vcg-simp]:  $(P\ land\ Q)\ s \longleftrightarrow P\ s \wedge Q\ s$ 
  <proof>
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 × IVert
type-synonym IGraph = IVert list × IEdge list

abbreviation (input) ig-edges :: IGraph ⇒ IEdge list where
  ig-edges G ≡ snd G

abbreviation (input) ig-verts :: IGraph ⇒ IVert list where
  ig-verts G ≡ fst G

definition ig-tail :: IGraph ⇒ nat ⇒ IVert where
  ig-tail IG a = fst (ig-edges IG ! a)

definition ig-head :: IGraph ⇒ nat ⇒ IVert where
  ig-head IG a = snd (ig-edges IG ! a)

type-synonym IMap = (nat ⇒ nat) × (nat ⇒ nat) × (nat ⇒ nat)

definition im-rev :: IMap ⇒ (nat ⇒ nat) where
  im-rev iM = fst iM

definition im-succ :: IMap ⇒ (nat ⇒ nat) where
  im-succ iM = fst (snd iM)

definition im-pred :: IMap ⇒ (nat ⇒ nat) where
  im-pred iM = snd (snd iM)

```

definition *mk-graph* :: *IGraph* ⇒ (*IVert*, *nat*) *pre-digraph* **where**

```

mk-graph IG ≡ ⟨
  verts = set (ig-verts IG),
  arcs = {0..< length (ig-edges IG)},
  tail = ig-tail IG,
  head = ig-head IG
⟩

```

lemma *mkg-simps*:

```

verts (mk-graph IG) = set (ig-verts IG)
tail (mk-graph IG) = ig-tail IG
head (mk-graph IG) = ig-head IG
⟨proof⟩

```

lemma *arcs-mkg*: *arcs (mk-graph IG)* = {*0..< length (ig-edges IG)*}

⟨*proof*⟩

lemma *arc-to-ends-mkg*: *arc-to-ends (mk-graph IG) a* = *ig-edges IG ! a*

⟨*proof*⟩

definition *mk-map* :: (*-*, *nat*) *pre-digraph* ⇒ *IMap* ⇒ *nat pre-map* **where**

```

mk-map G iM ≡ ⟨
  edge-rev = perm-restrict (im-rev iM) (arcs G),
  edge-succ = perm-restrict (im-succ iM) (arcs G)
⟩

```

lemma *mkm-simps*:

```

edge-rev (mk-map G iM) = perm-restrict (im-rev iM) (arcs G)
edge-succ (mk-map G iM) = perm-restrict (im-succ iM) (arcs G)
⟨proof⟩

```

lemma *es-eq-im*: *a ∈ arcs (mk-graph iG)* ⇒ *edge-succ (mk-map (mk-graph iG) iM) a* = *im-succ iM a*

⟨*proof*⟩

19.2 Implementation

definition *is-map* *iG iM* ≡

```

DO ecnt ← oreturn (length (snd iG));
  vcnt ← oreturn (length (fst iG));
  (i, revOk) ← owhile
    (λ(i, ok) s. i < ecnt ∧ ok)
    (λ(i, ok).
      DO
        j ← oreturn (im-rev iM i);
        revIn ← oreturn (j < length (ig-edges iG));
        revNeq ← oreturn (j ≠ i);
        revRevs ← oreturn (ig-edges iG ! j = prod.swap (ig-edges iG ! i));
        invol ← oreturn (im-rev iM j = i);

```

```

    oreturn (i + 1, revIn ∧ revNeq ∧ revRevs ∧ invol)
  OD)
(0, True);
(i, succPerm) ← owhile
(λ(i, ok) s. i < ecnt ∧ ok)
(λ(i, ok).
  DO
    j ← oreturn (im-succ iM i);
    succIn ← oreturn (j < length (ig-edges iG));
    succEnd ← oreturn (ig-tail iG i = ig-tail iG j);
    isPerm ← oreturn (im-pred iM j = i);
    oreturn (i + 1, succIn ∧ succEnd ∧ isPerm)
  OD)
(0, True);
(i, succOrbits, V, A) ← owhile
(λ(i, ok, V, A) s. i < ecnt ∧ succPerm ∧ ok)
(λ(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* ⇒ *nat pre-map* ⇒ - ⇒ *nat option* **where**

face-cycles *iG iM* ≡

```

      DO ecnt ← oreturn (length (snd iG));
      (edge-info, c, i) ←
      owhile
      (λ(edge-info, c, i) s. i < ecnt)
      (λ(edge-info, c, i).
      DO (edge-info, c) ←
      ocondition (λs. i ∉ edge-info)
      (DO j ← oreturn i;
      edge-info ← oreturn (insert j edge-info);
      ret' ← oreturn (pre-digraph-map.face-cycle-succ iM j);
      (edge-info, j) ←
      owhile
      (λ(edge-info, j) s. i ≠ j)
      (λ(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 $n \leftarrow \text{oreturn } (\text{length } (\text{ig-edges } iG));$
 $m \leftarrow \text{oreturn } (\text{length } (\text{ig-verts } iG));$
 $nz \leftarrow \text{isolated-nodes } iG;$
 $fc \leftarrow \text{face-cycles } iG\ iM;$
 $\text{oreturn } ((\text{int } n \text{ div } 2 + 2 * \text{int } c - \text{int } m - \text{int } nz - \text{int } fc) \text{ div } 2)$
OD

definition *certify* $iG\ iM\ c \equiv$
DO
 $\text{map} \leftarrow \text{is-map } iG\ iM;$
 $\text{ocondition } (\lambda-. \text{map})$
(DO
 $\text{gen} \leftarrow \text{euler-genus } iG\ (\text{mk-map } (\text{mk-graph } iG)\ iM)\ c;$
 $\text{oreturn } (\text{gen} = 0)$
OD)
 $(\text{oreturn } \text{False})$
OD

19.3 Verification

context begin
interpretation *Labeling-Syntax* $\langle \text{proof} \rangle$
lemma *trivial-label*: $P \implies \text{CTXT } IC\ CT\ OC\ P$
 $\langle \text{proof} \rangle$
end

lemma *ovalidNF-wp*:
assumes *ovalidNF* $P\ c\ (\lambda r\ s. r = x)$
shows *ovalidNF* $(\lambda s. Q\ x\ s \wedge P\ s)\ c\ Q$
 $\langle \text{proof} \rangle$

19.3.1 *is-map*

definition *is-map-rev-ok-inv* $iG\ iM\ k\ ok \equiv ok \longleftrightarrow (\forall i < k.$
 $\text{im-rev } iM\ i < \text{length } (\text{ig-edges } iG)$
 $\wedge \text{ig-edges } iG\ !\ \text{im-rev } iM\ i = \text{prod.swap } (\text{ig-edges } iG\ !\ i)$
 $\wedge \text{im-rev } iM\ i \neq i$
 $\wedge \text{im-rev } iM\ (\text{im-rev } iM\ i) = i)$

definition *is-map-succ-perm-inv* $iG\ iM\ k\ ok \equiv ok \longleftrightarrow (\forall i < k.$
 $\text{im-succ } iM\ i < \text{length } (\text{ig-edges } iG)$
 $\wedge \text{ig-tail } iG\ (\text{im-succ } iM\ i) = \text{ig-tail } iG\ i$
 $\wedge \text{im-pred } iM\ (\text{im-succ } iM\ i) = i)$

definition *is-map-succ-orbits-inv* $iG\ iM\ k\ ok\ V\ A \equiv$
 $A = (\bigcup i < (if\ ok\ then\ k\ else\ k - 1). \text{orbit } (\text{im-succ } iM)\ i) \wedge$

$$V = \{ig\text{-tail } iG \ i \mid i. \ i < (if \ ok \ then \ k \ else \ k - 1)\} \wedge$$

$$ok = (\forall i < k. \forall j < k. \ ig\text{-tail } iG \ i = ig\text{-tail } iG \ j \longrightarrow j \in orbit \ (im\text{-succ } iM) \ i)$$

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* $f \ A \ A$
 $\langle proof \rangle$

lemma *permutesI-finite-dom*:
assumes *finite* A
assumes $f \in A \rightarrow A$
assumes $\wedge a. \ a \notin A \implies f \ a = a$
assumes $\wedge a. \ a \in A \implies g \ (f \ a) = a$
shows f *permutes* A
 $\langle proof \rangle$

lemma *orbit-ss*:
assumes $f \in A \rightarrow A \ a \in A$
shows $orbit \ f \ a \subseteq A$
 $\langle proof \rangle$

lemma *segment-eq-orbit*:
assumes $y \notin orbit \ f \ x$ **shows** $segment \ f \ x \ y = orbit \ f \ x$
 $\langle proof \rangle$

lemma *funpow-in-funcset*:
assumes $x \in A \ f \in A \rightarrow A$ **shows** $(f \ \overset{\sim}{\sim} \ n) \ x \in A$
 $\langle proof \rangle$

lemma *funpow-eq-funcset*:
assumes $x \in A \ f \in A \rightarrow A \ \wedge y. \ y \in A \implies f \ y = g \ y$
shows $(f \ \overset{\sim}{\sim} \ n) \ x = (g \ \overset{\sim}{\sim} \ n) \ x$
 $\langle proof \rangle$

lemma *funpow-dist1-eq-funcset*:
assumes $y \in orbit \ f \ x \ x \in A \ f \in A \rightarrow A \ \wedge y. \ y \in A \implies f \ y = g \ y$
shows $funpow\text{-dist1} \ f \ x \ y = funpow\text{-dist1} \ g \ x \ y$
 $\langle proof \rangle$

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*⟩

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*⟩

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 $\implies i < \text{length} (\text{ig-edges } iG) \implies \text{ig-tail } iG (\text{im-succ } iM i) = \text{ig-tail } iG$
i
 ⟨*proof*⟩

lemma *succ-ok-imp-pred*:

succ-ok $\implies i < \text{length} (\text{ig-edges } iG) \implies \text{im-pred } iM (\text{im-succ } iM i) = i$
 ⟨*proof*⟩

lemma *succ-ok-imp-permutes*:

assumes *succ-ok*
shows *edge-succ* (*mk-map* (*mk-graph* *iG*) *iM*) *permutes arcs* (*mk-graph* *iG*)
 ⟨*proof*⟩

lemma *es-A2A*: *succ-ok* $\implies \text{edge-succ} (\text{mk-map} (\text{mk-graph } iG) iM) \in \text{arcs}$
 (*mk-graph* *iG*) $\rightarrow \text{arcs} (\text{mk-graph } iG)$
 ⟨*proof*⟩

lemma *im-succ-le-length*: *succ-ok* $\implies i < \text{length} (\text{ig-edges } iG) \implies \text{im-succ } iM i$
 $< \text{length} (\text{ig-edges } iG)$
 ⟨*proof*⟩

lemma *orbit-es-eq-im*:

succ-ok $\implies a \in \text{arcs} (\text{mk-graph } iG) \implies \text{orbit} (\text{edge-succ} (\text{mk-map} (\text{mk-graph}$
iG) *iM*)) $a = \text{orbit} (\text{im-succ } iM) a$
 ⟨*proof*⟩

lemma *segment-es-eq-im*:

succ-ok $\implies a \in \text{arcs} (\text{mk-graph } iG) \implies \text{segment} (\text{edge-succ} (\text{mk-map} (\text{mk-graph}$
iG) *iM*)) $a b = \text{segment} (\text{im-succ } iM) a b$
 ⟨*proof*⟩

lemma *in-orbit-im-succE*:

assumes $j \in \text{orbit } (im\text{-succ } iM) \ i \ \text{succ-ok } i < \text{length } (ig\text{-edges } iG)$

obtains $ig\text{-tail } iG \ j = ig\text{-tail } iG \ i \ j < \text{length } (ig\text{-edges } iG)$

<proof>

lemma *self-in-orbit-im-succ*:

assumes $\text{succ-ok } i < \text{length } (ig\text{-edges } iG)$ **shows** $i \in \text{orbit } (im\text{-succ } iM) \ i$

<proof>

end

locale *is-map-postcondition* = *is-map-postcondition0* +

fixes $so\text{-i } so\text{-ok } V \ A$

assumes $rev: rev\text{-ok} \longleftrightarrow \text{bidirected-digraph } (mk\text{-graph } iG) \ (\text{edge-rev } (mk\text{-map } (mk\text{-graph } iG) \ iM))$

assumes $\text{succ-orbits}: is\text{-map-succ-orbits-inv } iG \ iM \ so\text{-i } so\text{-ok } V \ A \ \text{succ-ok} \longrightarrow is\text{-map-fnal } iG \ so\text{-i } so\text{-ok}$

begin

lemma *ok-imp-digraph*:

assumes $rev\text{-ok } succ\text{-ok } so\text{-ok}$

shows $\text{digraph-map } (mk\text{-graph } iG) \ (mk\text{-map } (mk\text{-graph } iG) \ iM)$

<proof>

lemma *digraph-imp-ok*:

assumes $dm: \text{digraph-map } (mk\text{-graph } iG) \ (mk\text{-map } (mk\text{-graph } iG) \ iM)$

assumes $\text{pred}: \bigwedge i. i < \text{length } (ig\text{-edges } iG) \implies im\text{-pred } iM \ (im\text{-succ } iM \ i) = i$

obtains $rev\text{-ok } succ\text{-ok } so\text{-ok}$

<proof>

end

lemma *all-less-Suc-eq*: $(\forall x < \text{Suc } n. P \ x) \longleftrightarrow (\forall x < n. P \ x) \wedge P \ n$

<proof>

lemma *in-orbit-imp-in-segment*:

assumes $y \in \text{orbit } f \ x \ x \neq y \ \text{bij } f$ **shows** $y \in \text{segment } f \ x \ (f \ y)$

<proof>

lemma *ovalidNF-is-map*:

$\text{ovalidNF } (\lambda s. \text{distinct } (ig\text{-verts } iG) \wedge \text{wf-digraph } (mk\text{-graph } iG))$

$(is\text{-map } iG \ iM)$

$(\lambda r \ s. r \longleftrightarrow \text{digraph-map } (mk\text{-graph } iG) \ (mk\text{-map } (mk\text{-graph } iG) \ iM) \wedge (\forall i < \text{length } (ig\text{-edges } iG). im\text{-pred } iM \ (im\text{-succ } iM \ i) = i))$

<proof>

declare *ovalidNF-is-map*[*THEN ovalidNF-wp, THEN trivial-label, vcg-l*]

19.3.2 *isolated-nodes*

definition *inv-isolated-nodes* *s iG vcnt ecnt* \equiv

$vcnt = \text{length } (ig\text{-verts } iG)$
 $\wedge ecnt = \text{length } (ig\text{-edges } iG)$
 $\wedge \text{distinct } (ig\text{-verts } iG)$
 $\wedge \text{sym-digraph } (mk\text{-graph } iG)$

definition *inv-isolated-nodes-outer* *iG i nz* \equiv

$nz = \text{card } (\text{pre-digraph.isolated-verts } (mk\text{-graph } iG) \cap \text{set } (\text{take } i (ig\text{-verts } iG)))$

definition *inv-isolated-nodes-inner* *iG v j* \equiv

$\forall k < j. v \neq ig\text{-tail } iG k \wedge v \neq ig\text{-head } iG k$

lemma (*in sym-digraph*) *in-arcs-empty-iff*:

$in\text{-arcs } G v = \{\} \longleftrightarrow out\text{-arcs } G v = \{\}$
<proof>

lemma *take-nth-distinct*:

$\llbracket \text{distinct } xs; n < \text{length } xs; xs ! n \in \text{set } (\text{take } n xs) \rrbracket \implies \text{False}$
<proof>

lemma *ovalidNF-isolated-nodes*:

$ovalidNF (\lambda s. \text{distinct } (ig\text{-verts } iG) \wedge \text{sym-digraph } (mk\text{-graph } iG))$
 $(\text{isolated-nodes } iG)$
 $(\lambda r s. r = (\text{card } (\text{pre-digraph.isolated-verts } (mk\text{-graph } iG))))$
<proof>

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

$ecnt = \text{length } (ig\text{-edges } iG)$
 $\wedge \text{digraph-map } (mk\text{-graph } iG) iM$

definition *fcs-upto* $:: \text{nat pre-map} \Rightarrow \text{nat} \Rightarrow \text{nat set set}$ **where**

$fcs\text{-upto } iM i \equiv \{\text{pre-digraph-map.face-cycle-set } iM k \mid k. k < i\}$

definition *inv-face-cycles-outer* *s iG iM i c edge-info* \equiv

$\text{let } fcs = fcs\text{-upto } iM i \text{ in}$
 $c = \text{card } fcs$
 $\wedge (\forall k < \text{length } (ig\text{-edges } iG). k \in \text{edge-info} \longleftrightarrow k \in \bigcup fcs)$

definition *inv-face-cycles-inner* *s iG iM i j c edge-info* \equiv

$j \in \text{pre-digraph-map.face-cycle-set } iM i$

$\wedge c = \text{card } (\text{fcs-upto } iM \ i)$
 $\wedge i \notin \bigcup (\text{fcs-upto } iM \ i)$
 $\wedge (\forall k < \text{length } (\text{ig-edges } iG). k \in \text{edge-info} \longleftrightarrow$
 $\quad (k \in \bigcup (\text{fcs-upto } iM \ i)$
 $\quad \vee (\exists l < \text{funpow-dist1 } (\text{pre-digraph-map.face-cycle-succ } iM) \ i \ j. (\text{pre-digraph-map.face-cycle-succ}$
 $\quad iM \ \sim l) \ i = k)))$

lemma *finite-fcs-upto: finite (fcs-upto iM i)*
 $\langle \text{proof} \rangle$

lemma *card-orbit-eq-funpow-dist1:*
assumes $x \in \text{orbit } f \ x$ **shows** $\text{card } (\text{orbit } f \ x) = \text{funpow-dist1 } f \ x \ x$
 $\langle \text{proof} \rangle$

lemma *funpow-dist1-le:*
assumes $y \in \text{orbit } f \ x \ x \in \text{orbit } f \ x$
shows $\text{funpow-dist1 } f \ x \ y \leq \text{funpow-dist1 } f \ x \ x$
 $\langle \text{proof} \rangle$

lemma *funpow-dist1-le-card:*
assumes $y \in \text{orbit } f \ x \ x \in \text{orbit } f \ x$
shows $\text{funpow-dist1 } f \ x \ y \leq \text{card } (\text{orbit } f \ x)$
 $\langle \text{proof} \rangle$

lemma *(in digraph-map) funpow-dist1-le-card-fcs:*
assumes $b \in \text{face-cycle-set } a$
shows $\text{funpow-dist1 } \text{face-cycle-succ } a \ b \leq \text{card } (\text{face-cycle-set } a)$
 $\langle \text{proof} \rangle$

lemma *funpow-dist1-f-eq:*
assumes $b \in \text{orbit } f \ a \ a \in \text{orbit } f \ a \ a \neq b$
shows $\text{funpow-dist1 } f \ a \ (f \ b) = \text{Suc } (\text{funpow-dist1 } f \ a \ b)$
 $\langle \text{proof} \rangle$

lemma *(in -) funpow-dist1-less-f:*
assumes $b \in \text{orbit } f \ a \ a \in \text{orbit } f \ a \ a \neq b$
shows $\text{funpow-dist1 } f \ a \ b < \text{funpow-dist1 } f \ a \ (f \ b)$
 $\langle \text{proof} \rangle$

lemma *ovaidNF-face-cycles:*
 $\text{ovaidNF } (\lambda s. \text{digraph-map } (\text{mk-graph } iG) \ iM)$
 $\quad (\text{face-cycles } iG \ iM)$
 $(\lambda r \ s. r = \text{card } (\text{pre-digraph-map.face-cycle-sets } (\text{mk-graph } iG) \ iM))$

$\langle \text{proof} \rangle$

declare *ovaidNF-face-cycles[THEN ovalidNF-wp, THEN trivial-label, vcg-l]*

lemma *ovaidNF-euler-genus:*
 $\text{ovaidNF } (\lambda s. \text{distinct } (\text{ig-verts } iG) \wedge \text{digraph-map } (\text{mk-graph } iG) \ iM \wedge c = \text{card}$

```

(pre-digraph.sccs (mk-graph iG))
  (euler-genus iG iM c)
  ( $\lambda r s. r = \text{pre-digraph-map.euler-genus (mk-graph iG) iM}$ )

<proof>

declare ovalidNF-euler-genus[THEN ovalidNF-wp, THEN trivial-label, vcg-l]

lemma ovalidNF-certify:
  ovalidNF ( $\lambda s. \text{distinct (ig-verts iG)} \wedge \text{fin-digraph (mk-graph iG)} \wedge c = \text{card}$ 
(pre-digraph.sccs (mk-graph iG))
  (certify iG iM c)
  ( $\lambda r s. r \longleftrightarrow \text{pre-digraph-map.euler-genus (mk-graph iG) (mk-map (mk-graph iG)$ 
iM) = 0
   $\wedge \text{digraph-map (mk-graph iG) (mk-map (mk-graph iG) iM)}$ 
   $\wedge (\forall i < \text{length (ig-edges iG)}. \text{im-pred iM (im-succ iM i) = i})$  )

<proof>

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.