Graph Theory

By Lars Noschinski

February 23, 2021

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

theory *Graph-Genus*
imports
Graph-Theory.Graph-Theory
HOL−Library.Permutations
begin

lemma *nat-diff-mod-right*:
fixes *a b c* :: *nat*
assumes $b < a$
shows $(a - b) \bmod c = (a - b \bmod c) \bmod c$
⟨*proof*⟩

lemma *inj-on-f-imageI*:
assumes *inj-on f S* $\bigwedge t.\ t \in T \Longrightarrow t \subseteq S$
shows *inj-on* ((`) *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* ⇒ *'b*
edge-succ :: *'b* ⇒ *'b*

definition *edge-pred* :: *'b pre-map* ⇒ *'b* ⇒ *'b* **where**
edge-pred M = *inv* (*edge-succ M*)

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

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

lemma (**in** *fin-digraph*) *digraph-mapI*:
assumes *bidi*: $\bigwedge a.\ a \notin arcs\ G \Longrightarrow edge\text{-}rev\ M\ a = a$
$\bigwedge a.\ a \in arcs\ G \Longrightarrow edge\text{-}rev\ M\ a \neq a$
$\bigwedge a.\ a \in arcs\ G \Longrightarrow edge\text{-}rev\ M\ (edge\text{-}rev\ M\ a) = a$
$\bigwedge a.\ a \in arcs\ G \Longrightarrow tail\ G\ (edge\text{-}rev\ M\ a) = head\ G\ a$

assumes *edge-succ-permutes*: *edge-succ M permutes arcs G*
assumes *edge-succ-cyclic*: $\bigwedge v.\ v \in verts\ G \Longrightarrow out\text{-}arcs\ G\ v \neq \{\} \Longrightarrow cyclic\text{-}on$ (*edge-succ M*) (*out-arcs G v*)
shows *digraph-map G M*
⟨*proof*⟩

lemma (**in** *fin-digraph*) *digraph-mapI-permutes*:
assumes *bidi*: *edge-rev M permutes arcs G*
$\bigwedge a.\ a \in arcs\ G \Longrightarrow edge\text{-}rev\ M\ a \neq a$
$\bigwedge a.\ a \in arcs\ G \Longrightarrow edge\text{-}rev\ M\ (edge\text{-}rev\ M\ a) = a$
$\bigwedge a.\ a \in arcs\ G \Longrightarrow tail\ G\ (edge\text{-}rev\ M\ a) = head\ G\ a$
assumes *edge-succ-permutes*: *edge-succ M permutes arcs G*
assumes *edge-succ-cyclic*: $\bigwedge v.\ v \in verts\ G \Longrightarrow out\text{-}arcs\ G\ v \neq \{\} \Longrightarrow cyclic\text{-}on$ (*edge-succ M*) (*out-arcs G v*)
shows *digraph-map G M*
⟨*proof*⟩

context *digraph-map*
begin

lemma *digraph-map*[*intro*]: *digraph-map G M* ⟨*proof*⟩

lemma *permutation-edge-succ*: *permutation* (*edge-succ M*)
⟨*proof*⟩

lemma *edge-pred-succ*[*simp*]: *edge-pred M* (*edge-succ M a*) = *a*
⟨*proof*⟩

lemma *edge-succ-pred*[*simp*]: *edge-succ M* (*edge-pred M a*) = *a*
⟨*proof*⟩

lemma *edge-pred-permutes*: *edge-pred M permutes arcs G*
⟨*proof*⟩

lemma *permutation-edge-pred*: *permutation* (*edge-pred M*)
⟨*proof*⟩

lemma *edge-succ-eq-iff*[*simp*]: $\bigwedge x\ y.$ *edge-succ M x* = *edge-succ M y* $\longleftrightarrow$ *x* = *y*
⟨*proof*⟩

lemma *edge-rev-in-arcs*[*simp*]: *edge-rev M a* ∈ *arcs G* $\longleftrightarrow$ *a* ∈ *arcs G*
⟨*proof*⟩

lemma *edge-succ-in-arcs*[*simp*]: *edge-succ M a* ∈ *arcs G* $\longleftrightarrow$ *a* ∈ *arcs G*
⟨*proof*⟩

lemma *edge-pred-in-arcs*[*simp*]: *edge-pred M a* ∈ *arcs G* $\longleftrightarrow$ *a* ∈ *arcs G*
⟨*proof*⟩

lemma *tail-edge-succ*[*simp*]: *tail G* (*edge-succ M a*) = *tail G a*
⟨*proof*⟩

lemma *tail-edge-pred*[*simp*]: *tail G* (*edge-pred M a*) = *tail G a*
⟨*proof*⟩

lemma *bij-edge-succ*[*intro*]: *bij* (*edge-succ M*)
⟨*proof*⟩

lemma *edge-pred-cyclic*:
assumes *v* ∈ *verts G out-arcs G v* ≠ {}
shows *cyclic-on* (*edge-pred M*) (*out-arcs G v*)
⟨*proof*⟩

definition (**in** *pre-digraph-map*) *face-cycle-succ* :: *'b* ⇒ *'b* **where**
face-cycle-succ ≡ *edge-succ M o edge-rev M*

definition (**in** *pre-digraph-map*) *face-cycle-pred* :: *'b* ⇒ *'b* **where**
face-cycle-pred ≡ *edge-rev M o edge-pred M*

lemma *face-cycle-pred-succ*[*simp*]:
shows *face-cycle-pred* (*face-cycle-succ a*) = *a*
⟨*proof*⟩

lemma *face-cycle-succ-pred*[*simp*]:
shows *face-cycle-succ* (*face-cycle-pred a*) = *a*
⟨*proof*⟩

lemma *tail-face-cycle-succ*: *a* ∈ *arcs G* ⟹ *tail G* (*face-cycle-succ a*) = *head G a*
⟨*proof*⟩

lemma *funpow-prop*:
assumes ⋀*x*. *P* (*f x*) ⟷ *P x*
shows *P* ((*f* ^^ *n*) *x*) ⟷ *P x*
⟨*proof*⟩

lemma *face-cycle-succ-no-arc*[*simp*]: *a* ∉ *arcs G* ⟹ *face-cycle-succ a* = *a*
⟨*proof*⟩

lemma *funpow-face-cycle-succ-no-arc*[*simp*]:
assumes *a* ∉ *arcs G* **shows** (*face-cycle-succ* ^^ *n*) *a* = *a*
⟨*proof*⟩

lemma *funpow-face-cycle-pred-no-arc*[*simp*]:
assumes *a* ∉ *arcs G* **shows** (*face-cycle-pred* ^^ *n*) *a* = *a*
⟨*proof*⟩

lemma *face-cycle-succ-closed*[*simp*]:
face-cycle-succ a ∈ *arcs G* ⟷ *a* ∈ *arcs G*
⟨*proof*⟩

lemma *face-cycle-pred-closed*[*simp*]:
face-cycle-pred a ∈ *arcs G* ⟷ *a* ∈ *arcs G*
⟨*proof*⟩

lemma *face-cycle-succ-permutes*:
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*:
face-cycle-pred permutes arcs G
⟨*proof*⟩

definition (**in** *pre-digraph-map*) *face-cycle-set* :: *'b* ⇒ *'b set* **where**
face-cycle-set a = *orbit face-cycle-succ a*

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

lemma *face-cycle-set-altdef*: *face-cycle-set a* = {(*face-cycle-succ* ^^ *n*) *a* | *n*. *True*}
⟨*proof*⟩

lemma *face-cycle-set-self*[*simp*, *intro*]: *a* ∈ *face-cycle-set a*
⟨*proof*⟩

lemma *empty-not-in-face-cycle-sets*: {} ∉ *face-cycle-sets*
⟨*proof*⟩

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

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

lemma *face-cycle-set-induct*[*case-names base step*, *induct set*: *face-cycle-set*]:
assumes *consume*: *a* ∈ *face-cycle-set x*
and *ih-base*: *P x*
and *ih-step*: ⋀*y*. *y* ∈ *face-cycle-set x* ⟹ *P y* ⟹ *P* (*face-cycle-succ y*)
shows *P a*

⟨*proof*⟩

lemma *face-cycle-succ-cyclic*:
cyclic-on face-cycle-succ (*face-cycle-set a*)
⟨*proof*⟩

lemma *face-cycle-eq*:
assumes $b \in$ *face-cycle-set a* **shows** *face-cycle-set b* = *face-cycle-set a*
⟨*proof*⟩

lemma *face-cycle-succ-in-arcsI*: $\bigwedge a.\ a \in arcs\ G \Longrightarrow$ *face-cycle-succ a* $\in arcs\ G$
⟨*proof*⟩

lemma *face-cycle-succ-inI*: $\bigwedge x\ y.\ x \in$ *face-cycle-set y* $\Longrightarrow$ *face-cycle-succ x* $\in$ *face-cycle-set y*
⟨*proof*⟩

lemma *face-cycle-succ-inD*: $\bigwedge x\ y.$ *face-cycle-succ x* $\in$ *face-cycle-set y* $\Longrightarrow x \in$ *face-cycle-set y*
⟨*proof*⟩

lemma *face-cycle-set-parts*:
face-cycle-set a = *face-cycle-set b* $\vee$ *face-cycle-set a* $\cap$ *face-cycle-set b* = $\{\}$
⟨*proof*⟩

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

lemma *reflp-fc-equiv*: *reflp fc-equiv*
⟨*proof*⟩

lemma *symp-fc-equiv*: *symp fc-equiv*
⟨*proof*⟩

lemma *transp-fc-equiv*: *transp fc-equiv*
⟨*proof*⟩

lemma *equivp fc-equiv*
⟨*proof*⟩

lemma *in-face-cycle-setD*:
assumes $y \in$ *face-cycle-set x* $x \in arcs\ G$ **shows** $y \in arcs\ G$
⟨*proof*⟩

lemma *in-face-cycle-setsD*:
assumes $x \in$ *face-cycle-sets* **shows** $x \subseteq arcs\ G$
⟨*proof*⟩

end

definition (**in** *pre-digraph*) *isolated-verts* :: *'a set* **where**
isolated-verts ≡ {*v* ∈ *verts G*. *out-arcs G v* = {}}

definition (**in** *pre-digraph-map*) *euler-char* :: *int* **where**
euler-char ≡ *int* (*card* (*verts G*)) − *int* (*card* (*arcs G*) *div 2*) + *int* (*card face-cycle-sets*)

definition (**in** *pre-digraph-map*) *euler-genus* :: *int* **where**
euler-genus ≡ (*int* (*2* ∗ *card sccs*) − *int* (*card isolated-verts*) − *euler-char*) *div 2*

definition *comb-planar* :: ('*a*,'*b*) *pre-digraph* ⇒ *bool* **where**
comb-planar G ≡ ∃ *M*. *digraph-map G M* ∧ *pre-digraph-map.euler-genus G M* = *0*

Number of isolated vertices is a graph invariant

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

interpretation *wf-digraph G* ⟨*proof*⟩

lemma *isolated-verts-app-iso*[*simp*]:
pre-digraph.isolated-verts (*app-iso 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-neq*:
assumes *a* ∈ *arcs G tail G a* ≠ *head G a* **shows** *face-cycle-succ a* ≠ *a*
⟨*proof*⟩

end

2 Maps and Isomorphism

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

definition (**in** *pre-digraph-map*) *map-iso* :: (*'a*,*'b*,*'a2*,*'b2*) *digraph-isomorphism* ⇒ *'b2 pre-map* **where**
map-iso f ≡
(‖ *edge-rev* = *wrap-iso-arcs f* (*edge-rev M*)
, *edge-succ* = *wrap-iso-arcs f* (*edge-succ M*)
‖)

lemma *funcsetI-permutes*:
assumes *f permutes S* **shows** *f* ∈ *S* → *S*
⟨*proof*⟩

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

interpretation *wf-digraph G* ⟨*proof*⟩

lemma *wrap-iso-arcs-iso-arcs*[*simp*]:
assumes *x* ∈ *arcs G*
shows *wrap-iso-arcs hom f* (*iso-arcs hom x*) = *iso-arcs hom* (*f x*)
⟨*proof*⟩

lemma *inj-on-wrap-iso-arcs*:
assumes *dom*: ⋀*f*. *f* ∈ *F* ⟹ *has-dom f* (*arcs G*)
assumes *funcset*: *F* ⊆ *arcs G* → *arcs G*
shows *inj-on* (*wrap-iso-arcs hom*) *F*
⟨*proof*⟩

lemma *inj-on-wrap-iso-arcs-f*:
assumes *A* ⊆ *arcs G f* ∈ *A* → *A B* = *iso-arcs hom* ‘ *A*
assumes *inj-on f A* **shows** *inj-on* (*wrap-iso-arcs hom f*) *B*
⟨*proof*⟩

lemma *wrap-iso-arcs-in-funcsetI*:
assumes *A* ⊆ *arcs G f* ∈ *A* → *A*
shows *wrap-iso-arcs hom f* ∈ *iso-arcs hom* ‘ *A* → *iso-arcs hom* ‘ *A*
⟨*proof*⟩

lemma *wrap-iso-arcs-permutes*:
assumes *A* ⊆ *arcs G f permutes A*
shows *wrap-iso-arcs hom f permutes* (*iso-arcs hom* ‘ *A*)
⟨*proof*⟩

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 $m < length\ xs$
shows $rotate1\ xs\ !\ m = xs\ !\ (Suc\ m\ mod\ length\ xs)$
⟨*proof*⟩

lemma *nth-rotate-conv-nth*:
assumes $m < length\ xs$
shows $rotate\ n\ xs\ !\ m = xs\ !\ ((m + n)\ mod\ length\ xs)$
⟨*proof*⟩

lemma *not-nil-if-in-set*:
assumes $x \in set\ xs$ **shows** $xs \neq []$
⟨*proof*⟩

lemma *length-fold-remove1-le*:
$length\ (fold\ remove1\ ys\ xs) \leq length\ xs$
⟨*proof*⟩

lemma *set-fold-remove1′*:
assumes $x \in set\ xs - set\ ys$ **shows** $x \in set\ (fold\ remove1\ ys\ xs)$
⟨*proof*⟩

lemma *set-fold-remove1*:
$set\ (fold\ remove1\ xs\ ys) \subseteq set\ ys$
⟨*proof*⟩

lemma *set-fold-remove1-distinct*:
assumes $distinct\ xs$ **shows** $set\ (fold\ remove1\ ys\ xs) = set\ xs - set\ ys$
⟨*proof*⟩

lemma *distinct-fold-remove1*:
assumes $distinct\ xs$
shows $distinct\ (fold\ remove1\ ys\ xs)$
⟨*proof*⟩

end

4 Permutations as Products of Disjoint Cycles

theory *Executable-Permutations*
imports
Graph-Theory.Funpow
List-Aux
HOL−Library.Permutations
HOL−Library.Rewrite
begin

4.1 Cyclic Permutations

definition *list-succ* :: *'a list* ⇒ *'a* ⇒ *'a* **where**
list-succ xs x = (*if x* ∈ *set xs then xs* ! ((*index xs x* + *1*) *mod length xs*) *else x*)

We demonstrate the functions on the following simple lemmas
list-succ [*1*, *2*, *3*] *1* = *2 list-succ* [*1*, *2*, *3*] *2* = *3 list-succ* [*1*, *2*, *3*] *3* = *1*

lemma *list-succ-altdef*:
list-succ xs x = (*let n* = *index xs x in if n* + *1* = *length xs then xs* ! *0 else if n* + *1* < *length xs then xs* ! (*n* + *1*) *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* ⟹ *list-succ xs x* = *xs* ! *0*
index xs x + *1* < *length xs* ⟹ *list-succ xs x* = *xs* ! (*index xs x* + *1*)
length xs ≤ *index xs x* ⟹ *list-succ xs x* = *x*
⟨*proof*⟩

lemma *list-succ-not-in*:
assumes *x* ∉ *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* ⟹ *inj* (*list-succ xs*)

⟨*proof*⟩

lemma *inv-list-succ-eq*: *distinct xs* ⟹ *inv* (*list-succ xs*) = *list-succ* (*rev xs*)
⟨*proof*⟩

lemma *bij-list-succ*: *distinct xs* ⟹ *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* < *length xs* **shows** *list-succ xs* (*xs* ! *n*) = *xs* ! (*Suc n mod length xs*)
⟨*proof*⟩

lemma *list-succ-last*[*simp*]:
assumes *distinct xs xs* ≠ [] **shows** *list-succ xs* (*last xs*) = *hd xs*
⟨*proof*⟩

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

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

lemma *list-succ-in-conv*:
list-succ xs x ∈ *set xs* ⟷ *x* ∈ *set xs*
⟨*proof*⟩

lemma *list-succ-in-conv1*:
assumes *A* ∩ *set xs* = {}
shows *list-succ xs x* ∈ *A* ⟷ *x* ∈ *A*
⟨*proof*⟩

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

4.2 Arbitrary Permutations

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

definition *distincts* :: *'a list list* ⇒ *bool* **where**
  distincts xss ≡ *distinct xss* ∧ (∀ *xs* ∈ *set xss*. *distinct xs* ∧ *xs* ≠ []) ∧ (∀ *xs* ∈ *set xss*. ∀ *ys* ∈ *set xss*. *xs* ≠ *ys* ⟶ *set xs* ∩ *set ys* = {})

lemma *distincts-distinct*: *distincts xss* ⟹ *distinct xss*
  ⟨*proof*⟩

lemma *distincts-Nil*[*simp*]: *distincts* []
  ⟨*proof*⟩

lemma *distincts-single*: *distincts* [*xs*] ⟷ *distinct xs* ∧ *xs* ≠ []
  ⟨*proof*⟩

lemma *distincts-Cons*: *distincts* (*xs* # *xss*)
  ⟷ *xs* ≠ [] ∧ *distinct xs* ∧ *distincts xss* ∧ (*set xs* ∩ (⋃ *ys* ∈ *set xss*. *set ys*)) = {} (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *distincts-Cons'*: *distincts* (*xs* # *xss*)
  ⟷ *xs* ≠ [] ∧ *distinct xs* ∧ *distincts xss* ∧ (∀ *ys* ∈ *set xss*. *set xs* ∩ *set ys* = {}) (**is** *?L* ⟷ *?R*)
 ⟨*proof*⟩

lemma *distincts-rev*:
  distincts (*map rev xss*) ⟷ *distincts xss*
  ⟨*proof*⟩

lemma *length-distincts*:
  assumes *distincts xss*
  shows *length xss* = *card* (*set* ‘ *set xss*)
  ⟨*proof*⟩

lemma *distincts-remove1*: *distincts xss* ⟹ *distincts* (*remove1 xs xss*)
  ⟨*proof*⟩

lemma *distinct-Cons-remove1*:
  x ∈ *set xs* ⟹ *distinct* (*x* # *remove1 x xs*) = *distinct xs*
  ⟨*proof*⟩

lemma *set-Cons-remove1*:
  x ∈ *set xs* ⟹ *set* (*x* # *remove1 x xs*) = *set xs*
  ⟨*proof*⟩

lemma *distincts-Cons-remove1*:

$xs \in set\ xss \Longrightarrow distincts\ (xs\ \#\ remove1\ xs\ xss) = distincts\ xss$
⟨*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* $\Longrightarrow$ *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* ∈ *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* ⇒ '*a*) ⇒ '*a* ⇒ '*a list* **where**
orbit-list f x ≡ *iterate 0* (*funpow-dist1 f x x*) *f x*

partial-function (*tailrec*)
orbit-list-impl :: ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a list* ⇒ '*a* ⇒ '*a list*
where
orbit-list-impl f s acc x = (*let x'* = *f x in if x'* = *s then rev* (*x* # *acc*) *else orbit-list-impl f s* (*x* # *acc*) *x'*)

context notes [*simp*] = *length-fold-remove1-le* **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* (*fold remove1* (*orbit-list f x*) *xs*)

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* (*fold remove1 fc xs*))

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 $y \in \mathit{orbit}\ f\ x$
shows $\mathit{orbit\text{-}list\text{-}impl}\ f\ y\ \mathit{acc}\ x = \mathit{rev}\ \mathit{acc}\ @\ \mathit{iterate}\ 0\ (\mathit{funpow\text{-}dist1}\ f\ x\ y)\ f\ x$
⟨*proof*⟩

lemma *orbit-list-conv-impl*:
assumes $x \in \mathit{orbit}\ f\ x$
shows $\mathit{orbit\text{-}list}\ f\ x = \mathit{orbit\text{-}list\text{-}impl}\ f\ x\ []\ x$
⟨*proof*⟩

lemma *set-orbit-list*:
assumes $x \in \mathit{orbit}\ f\ x$
shows $\mathit{set}\ (\mathit{orbit\text{-}list}\ f\ x) = \mathit{orbit}\ f\ x$
⟨*proof*⟩

lemma *set-orbit-list′*:
assumes *permutation f* **shows** $\mathit{set}\ (\mathit{orbit\text{-}list}\ f\ x) = \mathit{orbit}\ f\ x$
⟨*proof*⟩

lemma *distinct-orbit-list*:
assumes $x \in \mathit{orbit}\ f\ x$
shows $\mathit{distinct}\ (\mathit{orbit\text{-}list}\ f\ x)$
⟨*proof*⟩

lemma *distinct-orbit-list′*:
assumes *permutation f* **shows** $\mathit{distinct}\ (\mathit{orbit\text{-}list}\ f\ x)$
⟨*proof*⟩

lemma *orbits-list-conv-impl*:
assumes *permutation f*
shows $\mathit{orbits\text{-}list}\ f\ \mathit{xs} = \mathit{orbits\text{-}list\text{-}impl}\ f\ \mathit{xs}$
⟨*proof*⟩

lemma *orbit-list-not-nil*[*simp*]: $\mathit{orbit\text{-}list}\ f\ x \neq []$
⟨*proof*⟩

lemma *sset-orbits-list*:
assumes *permutation f* **shows** $\mathit{sset}\ (\mathit{orbits\text{-}list}\ f\ \mathit{xs}) = (\mathit{orbit}\ f)\ `\ \mathit{set}\ \mathit{xs}$
⟨*proof*⟩

5.1 Relation to *cyclic-on*

lemma *list-succ-orbit-list*:
assumes $s \in \mathit{orbit}\ f\ s\ \bigwedge x.\ x \notin \mathit{orbit}\ f\ s \Longrightarrow f\ x = x$
shows $\mathit{list\text{-}succ}\ (\mathit{orbit\text{-}list}\ f\ s) = f$
⟨*proof*⟩

lemma *list-succ-funpow-conv*:
assumes *A*: *distinct xs x* ∈ *set xs*
shows (*list-succ xs* ^^ *n*) *x* = *xs* ! ((*index xs x* + *n*) *mod length xs*)
⟨*proof*⟩

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

lemma *cyclic-on-list-succ*:
assumes *distinct xs xs* ≠ [] **shows** *cyclic-on* (*list-succ xs*) (*set xs*)
⟨*proof*⟩

lemma *obtain-orbit-list-func*:
assumes *s* ∈ *orbit f s* ⋀*x*. *x* ∉ *orbit f s* ⟹ *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* ⋀*x*. *x* ∉ *S* ⟹ *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-succ xs set xs* = *S distinct xs*
⟨*proof*⟩

lemma *list-succ-unique*:
assumes *s* ∈ *orbit f s* ⋀*x*. *x* ∉ *orbit f s* ⟹ *f x* = *x*
shows ∃!*xs*. *f* = *list-succ xs* ∧ *distinct xs* ∧ *hd xs* = *s* ∧ *set xs* = *orbit f s*
⟨*proof*⟩

lemma *distincts-orbits-list*:
assumes *distinct as permutation f*
shows *distincts* (*orbits-list f as*)
⟨*proof*⟩

lemma *cyclic-on-lists-succ′*:
assumes *distincts xss*
shows *A* ∈ *sset xss* ⟹ *cyclic-on* (*lists-succ xss*) *A*
⟨*proof*⟩

lemma *cyclic-on-lists-succ*:
assumes *distincts xss*
shows ⋀*xs*. *xs* ∈ *set xss* ⟹ *cyclic-on* (*lists-succ xss*) (*set xs*)
⟨*proof*⟩

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

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

5.2 Permutations of a List

lemma *length-remove1-less*:
assumes $x \in set\ xs$ **shows** $length\ (remove1\ x\ xs) < length\ xs$
⟨*proof*⟩
context notes [*simp*] = *length-remove1-less* **begin**
fun *permutations* :: *'a list* ⇒ *'a list list* **where**
permutations-Nil: *permutations* [] = [[]]
| *permutations-Cons*:
permutations xs = [*y* # *ys*. *y* <− *xs*, *ys* <− *permutations* (*remove1 y xs*)]
end

declare *permutations-Cons*[*simp del*]

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

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

lemma *nil-in-permutations*[*simp*]: [] ∈ *set* (*permutations xs*) ⟷ *xs* = []
⟨*proof*⟩

lemma *permutations-not-nil*:
assumes $xs \neq []$
shows *permutations xs* = *concat* (*map* (λ*x*. *map* ((#) *x*) (*permutations* (*remove1 x xs*))) *xs*)
⟨*proof*⟩

lemma *set-permutations-step*:
assumes $xs \neq []$
shows *set* (*permutations xs*) = ($\bigcup x \in$ *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-succ* ‘ *set* (*cyc-permutations xs*) = {*f*. *f permutes set xs* ∧ *cyclic-on f* (*set xs*)} (**is** *?L* = *?R*)
⟨*proof*⟩

5.3 Enumerating Permutations from List Orbits

definition *cyc-permutationss* :: *'a list list* ⇒ *'a list list list* **where**
cyc-permutationss = *product-lists o map cyc-permutations*

lemma *cyc-permutationss-Nil*[*simp*]: *cyc-permutationss* [] = [[]]
⟨*proof*⟩

lemma *in-set-cyc-permutationss*:
assumes *distincts xss*
shows *yss* ∈ *set* (*cyc-permutationss xss*) ⟷ *distincts yss* ∧ *map set xss* = *map set yss* ∧ *map hd xss* = *map hd yss*
⟨*proof*⟩

lemma *lists-succ-set-cyc-permutationss*:
assumes *distincts xss*
shows *lists-succ* ‘ *set* (*cyc-permutationss xss*) = {*f*. *f permutes* $\bigcup$(*sset xss*) ∧ (∀ *c* ∈ *sset xss*. *cyclic-on f c*)} (**is** *?L* = *?R*)
⟨*proof*⟩

5.4 Lists of Permutations

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

lemma *permutationss-Nil*[*simp*]: *permutationss* [] = [[]]
⟨*proof*⟩

lemma *permutationss-Cons*:
permutationss (*xs* # *xss*) = *concat* (*map* (λ*ys*. *map* (*Cons ys*) (*permutationss xss*)) (*permutations xs*))
⟨*proof*⟩

lemma *in-set-permutationss*:
assumes *distincts xss*
shows *yss* ∈ *set* (*permutationss xss*) ⟷ *distincts yss* ∧ *map set xss* = *map set yss*
⟨*proof*⟩

lemma *set-permutationss*:
assumes *distincts xss*
shows *set* (*permutationss xss*) = {*yss*. *distincts yss* ∧ *map set xss* = *map set yss*}
⟨*proof*⟩

lemma *permutationss-complete*:
assumes *distincts xss distincts yss xss* ≠ []
and *set* ‘ *set xss* = *set* ‘ *set yss*
shows *set yss* ∈ *set* ‘ *set* (*permutationss xss*)
⟨*proof*⟩

lemma *permutations-complete*:
assumes *distinct xs distinct ys set xs* = *set ys*
shows *ys* ∈ *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* × *'b*) *list* ⇒ (*'a* × *'b*) *list list* **where**
grouped-by-fst xs = *map* (λ*u*. *filter* (λ*x*. *fst x* = *u*) *xs*) (*remdups* (*map fst xs*))

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

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

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

code-datatype *list-digraph-ext*

lemma *list-digraph-simps*:
pverts (*list-digraph G-list*) = *set* (*fst G-list*)
parcs (*list-digraph G-list*) = *set* (*snd G-list*)
⟨*proof*⟩

lemma *union-grouped-by-fst*:
(⋃ *xs* ∈ *set* (*grouped-by-fst ys*). *set xs*) = *set ys*
⟨*proof*⟩

lemma *union-grouped-out-arcs*:
(⋃ *xs* ∈ *set* (*grouped-out-arcs G-list*). *set xs*) = *set* (*snd G-list*)
⟨*proof*⟩

lemma *nil-not-in-grouped-out-arcs*: [] ∉ *set* (*grouped-out-arcs G-list*)
⟨*proof*⟩

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

lemma *distincts-grouped-by-fst*:
assumes *distinct xs* **shows** *distincts* (*grouped-by-fst xs*)
⟨*proof*⟩

lemma *distincts-grouped-arcs*:
assumes *distinct* (*snd G-list*) **shows** *distincts* (*grouped-out-arcs G-list*)
⟨*proof*⟩

lemma *distincts-in-all-maps-list*:
distinct (*snd X*) ⟹ *xss* ∈ *set* (*all-maps-list X*) ⟹ *distincts xss*
⟨*proof*⟩

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

abbreviation *to-map'* *as xss* ≡ *to-map* (*set as*) (*lists-succ xss*)

definition *all-maps* :: *'a pair-pre-digraph* $\Rightarrow$ $('a \times 'a)$ *pre-map set* **where**
all-maps G ≡ *to-map* (*arcs G*) ` {*f*. *f permutes arcs G* ∧ (∀ *v* ∈ *verts G*. *out-arcs G v* ≠ {} ⟶ *cyclic-on f* (*out-arcs G v*))}

definition *maps-all-maps-list* :: $('a\ list \times ('a \times 'a)\ list) \Rightarrow ('a \times 'a)$ *pre-map list*
where
maps-all-maps-list G-list = *map* (*to-map* (*set* (*snd G-list*)) *o lists-succ*) (*all-maps-list G-list*)

lemma (**in** *pair-graph*) *all-maps-correct*:
shows *all-maps G* = {*M*. *digraph-map G M*}
⟨*proof*⟩

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* :: $('a \times 'a)\ list\ list \Rightarrow ('a \times 'a) \Rightarrow ('a \times 'a)$ **where**
lists-fc-succ xss = (*let sxss* = ⋃(*sset xss*) *in* (λ*x*. *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*: ⋀*a*. *a* ∈ *parcs* (*list-digraph G-list*) ⟹ *fst a* ≠ *snd a*
assumes *distincts-xss*: *distincts xss*
assumes *parcs-xss*: *parcs* (*list-digraph G-list*) = ⋃(*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* (⋃(*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* | *a*. *a* ∈ *set as*}
⟨*proof*⟩

lemma *distincts-lists-fcs*: *distinct as* ⟹*distincts* (*lists-fcs as*)
⟨*proof*⟩

lemma *face-cycle-set-ss*: *a* ∈ *parcs G* ⟹ *face-cycle-set a* ⊆ *parcs G*
⟨*proof*⟩

lemma *face-cycle-succ-neq*:
assumes *a* ∈ *parcs G* **shows** *face-cycle-succ a* ≠ *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* ≡ λ*as xs*. [*b*. (*a*,*b*) <− *as*, *a* ∈ *set xs*]

interpretation *RTLI*: *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* (*RTLI.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* × (*'a* × *'a*) *list* ⇒ *'a set set* **where**
sccs-verts-impl G ≡ *set* ‘ (λ*x*. *RTLI.rtrancl-i* (*snd G*) [*x*]) ‘ *set* (*fst G*)

definition *isolated-verts-impl* :: *'a list* × (*'a* × *'a*) *list* ⇒ *'a list* **where**
isolated-verts-impl G = [*v* ← (*fst G*). ¬(∃ *e* ∈ *set* (*snd G*). *fst e* = *v*)]

definition *pair-graph-impl* :: *'a list* × (*'a* × *'a*) *list* ⇒ *bool* **where**
pair-graph-impl G ≡ *case G of* (*V*,*A*) ⇒ (∀ (*u*,*v*) ∈ *set A*. *u* ≠ *v* ∧ *u* ∈ *set V* ∧ *v* ∈ *set V* ∧ (*v*,*u*) ∈ *set A*)

definition *genus-impl* :: *'a list* × (*'a* × *'a*) *list* ⇒ (*'a* × *'a*) *list list* ⇒ *int* **where**
genus-impl G M ≡ *case G of* (*V*,*A*) ⇒
(*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* × (*'a* × *'a*) *list* ⇒ *bool* **where**
comb-planar-impl G ≡ *case G of* (*V*,*A*) ⇒
let i = *int* (2∗*card* (*sccs-verts-impl G*)) − *int* (*length* (*isolated-verts-impl G*))
− *int* (*length V*) + *int* (*length A*) *div* 2
in (∃ *M*∈*set* (*all-maps-list G*). (*i* − *int* (*length* (*orbits-list-impl* (*lists-fc-succ M*) *A*))) *div* 2 = 0)

lemma *sccs-verts-impl-correct*:
assumes *pair-pseudo-graph* (*list-digraph G*)
shows *pre-digraph.sccs-verts* (*list-digraph G*) = *sccs-verts-impl G*
⟨*proof*⟩

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-maps-list G*) *distinct* (*snd G*)
shows $\bigcup$(*sset M*) = *set* (*snd G*)
⟨*proof*⟩

lemma *comb-planar-impl-altdef*: *comb-planar-impl G* = (∃ *M*∈*set* (*all-maps-list G*). *genus-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-K5-list* ≡ ([*0..4*], [(*x*,*y*). *x* <− [*0..4*], *y* <− [*0..4*], *x* ≠ *y*])

abbreviation *c-K5* :: *int pair-pre-digraph* **where**
c-K5 ≡ *list-digraph c-K5-list*

lemma *c-K5-not-comb-planar*: ¬*comb-planar c-K5*
⟨*proof*⟩

lemma *pverts-c-K5*: *pverts c-K5* = {*0..4*}
⟨*proof*⟩

lemma *parcs-c-K5*: *parcs c-K5* = {(*u*,*v*). *u* ∈ {*0..4*} ∧ *v* ∈ {*0..4*} ∧ *u* ≠ *v*}
⟨*proof*⟩

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

lemma *complete-c-K5*: K_5 *c-K5*
⟨*proof*⟩

8.2 A concrete K33 graph

definition *c-K33-list* ≡ ([*0..5*], [(*x*,*y*). *x* <− [*0..5*], *y* <− [*0..5*], *even x* ⟷ *odd y*])

abbreviation *c-K33* :: *int pair-pre-digraph* **where**
c-K33 ≡ *list-digraph c-K33-list*

lemma *c-K33-not-comb-planar*: ¬*comb-planar c-K33*
⟨*proof*⟩

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

8.3 Generalization to arbitrary Kuratowski Graphs

8.3.1 Number of Face Cycles is a Graph Invariant

lemma (**in** *digraph-map*) *wrap-wrap-iso*:
assumes *hom*: *digraph-isomorphism hom*
assumes *f*: *f* ∈ *arcs G* → *arcs G* **and** *g*: *g* ∈ *arcs G* → *arcs G*
shows *wrap-iso-arcs hom f* (*wrap-iso-arcs hom g x*) = *wrap-iso-arcs hom* (*f o g*) *x*
⟨*proof*⟩

lemma (**in** *digraph-map*) *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*) = (λ*x*. *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*) ⟷ *graph G*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *arcs-ends-iso*:
assumes *digraph-isomorphism hom*
shows *arcs-ends* (*app-iso hom G*) = (λ(*u*,*v*). (*iso-verts hom u*, *iso-verts hom v*)) ‘ *arcs-ends G*
⟨*proof*⟩

lemma *inj-onI-pair*:
assumes *inj-on f S T* ⊆ *S* × *S*
shows *inj-on* (λ(*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*) ⟷ *K*$_n$ *G* (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

8.3.4 Conclusion

definition (**in** *pre-digraph*)
mk-iso :: (′*a* ⇒ ′*c*) ⇒ (′*b* ⇒ ′*d*) ⇒ (′*a*, ′*b*, ′*c*, ′*d*) *digraph-isomorphism*
where
mk-iso fv fa ≡ (‖ *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* ‖)

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* ≡ λ*x*. *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 ‘ (*A* × *B*) = *f* ‘ *A* × *f* ‘ *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** ¬*comb-planar G*
⟨*proof*⟩

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

end

9 n-step reachability

theory *Reachablen*
imports
Graph-Theory.Graph-Theory
begin

inductive
ntrancl-onp :: *'a set* ⇒ *'a rel* ⇒ *nat* ⇒ *'a* ⇒ *'a* ⇒ *bool*
for *F* :: *'a set* **and** *r* :: *'a rel*
where
ntrancl-on-0: *a* = *b* ⟹ *a* ∈ *F* ⟹ *ntrancl-onp F r 0 a b*
| *ntrancl-on-Suc*: (*a*,*b*) ∈ *r* ⟹ *ntrancl-onp F r n b c* ⟹ *a* ∈ *F* ⟹ *ntrancl-onp F r* (*Suc n*) *a c*

lemma *ntrancl-onpD-rtrancl-on*:
assumes *ntrancl-onp F r n a b* **shows** (*a*,*b*) ∈ *rtrancl-on F r*
⟨*proof*⟩

lemma *rtrancl-onE-ntrancl-onp*:
assumes (*a*,*b*) ∈ *rtrancl-on F r* **obtains** *n* **where** *ntrancl-onp F r n a b*
⟨*proof*⟩

lemma *rtrancl-on-conv-ntrancl-onp*: (*a*,*b*) ∈ *rtrancl-on F r* ⟷ (∃ *n*. *ntrancl-onp F r n a b*)
⟨*proof*⟩

definition *nreachable* :: (*'a*,*'b*) *pre-digraph* ⇒ *'a* ⇒ *nat* ⇒ *'a* ⇒ *bool* (- →⁻¹ - [*100*,*100*] *40*) **where**
nreachable G u n v ≡ *ntrancl-onp* (*verts G*) (*arcs-ends G*) *n u v*

context *wf-digraph* **begin**

lemma *reachableE-nreachable*:
assumes *u* →* *v* **obtains** *n* **where** *u* →^*n* *v*
⟨*proof*⟩

lemma *converse-nreachable-cases*[*cases pred*: *nreachable*]:
assumes *u* →^*n* *v*
obtains (*ntrancl-on-0*) *u* = *v n* = *0 u* ∈ *verts G*
| (*ntrancl-on-Suc*) *w m* **where** *u* → *w n* = *Suc m w* →^*m* *v*
⟨*proof*⟩

lemma *converse-nreachable-induct*[*consumes 1*, *case-names base step*, *induct pred*: *reachable*]:
assumes *major*: *u* →^*n*_*G* *v*
and *cases*: *v* ∈ *verts G* ⟹ *P 0 v*
⋀*n x y*. ⟦*x* →_*G* *y*; *y* →^*n*_*G* *v*; *P n y*⟧ ⟹ *P* (*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* →^*n*_*G* *v*

and *cases*: $v \in verts\ G \Longrightarrow P\ 0\ v$
$\bigwedge n\ x\ y.\ [\![x \rightarrow_G y;\ y \rightarrow^n{}_G v;\ \bigwedge z\ m.\ m \leq n \Longrightarrow (z \rightarrow^m{}_G v) \Longrightarrow P\ m\ z]\!] \Longrightarrow P\ (Suc\ n)\ x$
shows $P\ n\ u$
⟨*proof*⟩

end

end
theory *Permutations-2*
imports
HOL−Library.Permutations
Executable-Permutations
Graph-Theory.Funpow
begin

10 Modifying Permutations

abbreviation *funswapid* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (**infix** $\rightleftharpoons_F$ *90*) **where**
$x \rightleftharpoons_F y \equiv Fun.swap\ x\ y\ id$

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

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

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
$x = y \Longrightarrow perm\text{-}rem\ x\ f\ y = x$
$f\ y = x \Longrightarrow perm\text{-}rem\ x\ f\ y = f\ x$
$y \neq x \Longrightarrow f\ y \neq x \Longrightarrow perm\text{-}rem\ x\ f\ y = f\ y$
⟨*proof*⟩

lemma *bij-swap-compose*: *bij* $(x \rightleftharpoons_F y \circ f) \longleftrightarrow$ *bij f*
⟨*proof*⟩

lemma *bij-perm-rem*[*simp*]: *bij* (*perm-rem x f*) $\longleftrightarrow$ *bij f*
⟨*proof*⟩

lemma *perm-rem-conv*: $\bigwedge f\ x\ y.$ *bij f* $\Longrightarrow$ *perm-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 *bij-eq-iff*:
assumes *bij f* **shows** *f x* = *f y* $\longleftrightarrow$ *x* = *y*
⟨*proof*⟩

lemma *swap-swap-id*[*simp*]: $(x \rightleftharpoons_F y)\ ((x \rightleftharpoons_F y)\ z) = z$
⟨*proof*⟩

lemma *in-funswapid-image-iff*: $\bigwedge a\ b\ x\ S.\ x \in (a \rightleftharpoons_F b)\ `\ S \longleftrightarrow (a \rightleftharpoons_F b)\ x \in S$
⟨*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*) $\longleftrightarrow$ *bij f*
⟨*proof*⟩

lemma *funpow-perm-swap*: *perm-swap a b f* $\widehat{\ }$ *n* = *perm-swap a b* (*f* $\widehat{\ }$ *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 *has-domD*: *has-dom f S* ⟹ *x* ∉ *S* ⟹ *f x = x*
⟨*proof*⟩

lemma *has-domI*: (⋀*x*. *x* ∉ *S* ⟹ *f x = x*) ⟹ *has-dom f S*
⟨*proof*⟩

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

11 Cyclic Permutations

lemma *cyclic-on-perm-swap*:
assumes *cyclic-on f S* **shows** *cyclic-on* (*perm-swap x y f*) ((*x* ⇌$_F$ *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* ≠ {*x*} **shows** *cyclic-on* (*perm-rem x f*) (*S* − {*x*})
⟨*proof*⟩

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

12 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* ⊆ *H*: *digraph-map H HM*
rewrites *edge-rev HM* = *rev-H* ⟨*proof*⟩

sublocale *subdiv1-contr* ⊆ *G*: *fin-digraph G*
⟨*proof*⟩

context *subdiv1-contr* **begin**

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

lemma *edge-rev-GM*: *edge-rev GM* = *rev-G*
⟨*proof*⟩

lemma *edge-succ-GM*: *edge-succ GM* = *perm-swap uw uv* (*perm-swap vw* (*rev-G uv*) (*fold perm-rem* [*wu*, *wv*] (*edge-succ HM*)))
⟨*proof*⟩

lemma *rev-H-eq-rev-G*:
assumes *x* ∈ *arcs G* − {*uv*,*vu*} **shows** *rev-H x* = *rev-G x*
⟨*proof*⟩

lemma *edge-succ-permutes*: *edge-succ GM permutes arcs G*
⟨*proof*⟩

lemma *out-arcs-empty*:
assumes *x* ∈ *verts G*
shows *out-arcs G x* = {} ⟷ *out-arcs H x* = {}
⟨*proof*⟩

lemma *cyclic-on-edge-succ*:
assumes *x* ∈ *verts G out-arcs G x* ≠ {}
shows *cyclic-on* (*edge-succ GM*) (*out-arcs G x*)
⟨*proof*⟩

lemma *digraph-map-GM*:
shows *digraph-map G GM*
⟨*proof*⟩

end

sublocale *subdiv1-contr* $\subseteq$ *GM*: *digraph-map G GM* ⟨*proof*⟩

context *subdiv1-contr* **begin**

lemma *reachableGD*:
assumes $x \rightarrow^*_G y$ **shows** $x \rightarrow^*_H y$
⟨*proof*⟩

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

lemma *proj-verts-H-in-G*: *x* ∈ *verts H* ⟹ *proj-verts-H x* ∈ *verts G*
⟨*proof*⟩

lemma *dominatesHD*:
assumes $x \rightarrow_H y$ **shows** *proj-verts-H x* $\rightarrow^*_G$ *proj-verts-H y*
⟨*proof*⟩

lemma *reachableHD*:
assumes *reach*:$x \rightarrow^*_H y$ **shows** *proj-verts-H x* $\rightarrow^*_G$ *proj-verts-H y*
⟨*proof*⟩

lemma *H-reach-conv*: $\bigwedge x\ y.\ x \rightarrow^*_H y \longleftrightarrow$ *proj-verts-H x* $\rightarrow^*_G$ *proj-verts-H y*
⟨*proof*⟩

lemma *sccs-eq*: *G.sccs-verts* = (‘) *proj-verts-H* ‘ *H.sccs-verts* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *inj-on-proj-verts-H*: *inj-on* ((‘) *proj-verts-H*) (*pre-digraph.sccs-verts H*)
⟨*proof*⟩

lemma *card-sccs-verts*: *card G.sccs-verts* = *card H.sccs-verts*
⟨*proof*⟩

lemma *card-sccs-eq*: *card G.sccs* = *card H.sccs*
⟨*proof*⟩

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

lemma *card-verts*: *card* (*verts H*) = *card* (*verts G*) + *1*
⟨*proof*⟩

lemma *card-arcs*: *card* (*arcs H*) = *card* (*arcs G*) + *2*
⟨*proof*⟩

lemma *edge-succ-wu*: *edge-succ HM wu* = *wv*
⟨*proof*⟩

lemma *edge-succ-wv*: *edge-succ HM wv = wu*
⟨*proof*⟩

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

lemma *H-face-cycle-succ*:
H.face-cycle-succ uw = wv
H.face-cycle-succ vw = wu
⟨*proof*⟩

lemma *H-edge-succ-tail-eqD*:
assumes *edge-succ HM a = b* **shows** *tail H a = tail H b*
⟨*proof*⟩

lemma *YYY*:
(wu ⇌$_F$ wv) (edge-succ HM vw) = (edge-succ HM vw)
(wu ⇌$_F$ wv) (edge-succ HM uw) = (edge-succ HM uw)
⟨*proof*⟩

Project arcs of H to corresponding arcs of G

definition *proj-arcs-H* :: *'b ⇒ 'b* **where**
proj-arcs-H x ≡
if x = uw ∨ x = wv then uv
else if x = vw ∨ x = wu then vu
else x

Project arcs of G to corresponding arcs of H

definition *proj-arcs-G* :: *'b ⇒ 'b* **where**
proj-arcs-G x ≡
if x = uv then uw
else if x = vu then vw
else x

lemma *proj-arcs-H-simps*[*simp*]:
proj-arcs-H uw = uv
proj-arcs-H wv = uv
proj-arcs-H vw = vu
proj-arcs-H wu = vu
x ∉ {uw,vw,wu,wv} ⟹ proj-arcs-H x = x
a ∈ arcs G ⟹ proj-arcs-H a = a
⟨*proof*⟩

lemma *proj-arcs-H-in-arcs-G*: *a ∈ arcs H ⟹ proj-arcs-H a ∈ arcs G*
⟨*proof*⟩

lemma *proj-arcs-eq-swap*:
assumes *a ∉ {uv,vu,wu,wv}*
shows *proj-arcs-H a = (uw ⇌$_F$ uv o vw ⇌$_F$ vu) a*
⟨*proof*⟩

lemma *proj-arcs-G-simps*:
proj-arcs-G uv = uw
proj-arcs-G vu = vw
a ∉ {uv,vu} ⟹ proj-arcs-G a = a
⟨*proof*⟩

lemma *proj-arcs-G-in-arcs-H*:
assumes *a ∈ arcs G* **shows** *proj-arcs-G a ∈ arcs H*
⟨*proof*⟩

lemma *proj-arcs-HG*: *a ∈ arcs G ⟹ proj-arcs-H (proj-arcs-G a) = a*
⟨*proof*⟩

lemma *fcs-proj-arcs-GH*:
assumes *a ∈ arcs H* **shows** *H.face-cycle-set (proj-arcs-G (proj-arcs-H a)) = H.face-cycle-set a*
⟨*proof*⟩

lemma *H-face-cycle-succ-neq-uv*:
a ∉ {uv,vu} ⟹ H.face-cycle-succ a ∉ {uv,vu}
⟨*proof*⟩

lemma *face-cycle-succ-choose-inter*:
{H.face-cycle-succ uw, H.face-cycle-succ vw, H.face-cycle-succ wu, H.face-cycle-succ wv} ∩ {uv,vu} = {}
⟨*proof*⟩

lemma *face-cycle-succ-choose-neq*:
H.face-cycle-succ wu ∉ {wu,wv}
H.face-cycle-succ wv ∉ {wu,wv}
⟨*proof*⟩

lemma *H-face-cycle-succ-G-not-in*:
assumes *a ∈ arcs G* **shows** *H.face-cycle-succ a ∉ {wu,wv}*
⟨*proof*⟩

lemma
face-cycle-succ-uv: *GM.face-cycle-succ uv = proj-arcs-H (H.face-cycle-succ wv)*
and
face-cycle-succ-vu: *GM.face-cycle-succ vu = proj-arcs-H (H.face-cycle-succ wu)*
⟨*proof*⟩

lemma *face-cycle-succ-not-uv*:
assumes *a ∈ arcs G a ∉ {uv,vu}*
shows *GM.face-cycle-succ a = proj-arcs-H (H.face-cycle-succ a)*
⟨*proof*⟩

lemmas *G-face-cycle-succ = face-cycle-succ-uv face-cycle-succ-vu face-cycle-succ-not-uv*

lemma *in-G-fcs-in-H-fcs*:
assumes $a \in arcs\ G$
assumes $x \in GM.face\text{-}cycle\text{-}set\ a$
shows $x \in proj\text{-}arcs\text{-}H\ `\ H.face\text{-}cycle\text{-}set\ (proj\text{-}arcs\text{-}G\ a)$
⟨*proof*⟩

lemma *in-H-fcs-in-G-fcs*:
assumes $a \in arcs\ H$
assumes $x \in H.face\text{-}cycle\text{-}set\ a$
shows $x \in proj\text{-}arcs\text{-}H\ -`\ GM.face\text{-}cycle\text{-}set\ (proj\text{-}arcs\text{-}H\ a)$
⟨*proof*⟩

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

lemma *H-fcs-eq*:
assumes $a \in arcs\ H$
shows $proj\text{-}arcs\text{-}H\ `\ H.face\text{-}cycle\text{-}set\ a = GM.face\text{-}cycle\text{-}set\ (proj\text{-}arcs\text{-}H\ a)$
⟨*proof*⟩

lemma *face-cycle-sets*:
shows $GM.face\text{-}cycle\text{-}sets = (`)\ proj\text{-}arcs\text{-}H\ `\ H.face\text{-}cycle\text{-}sets$ (**is** $?L = ?R$)
⟨*proof*⟩

lemma *inj-on-proj-arcs-H*: $inj\text{-}on\ ((`)\ proj\text{-}arcs\text{-}H)\ H.face\text{-}cycle\text{-}sets$
⟨*proof*⟩

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

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

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

end

lemma *subdivision-genus-same-rev*:
assumes $subdivision\ (G, rev\text{-}G)\ (H, edge\text{-}rev\ HM)\ digraph\text{-}map\ H\ HM\ pre\text{-}digraph\text{-}map.euler\text{-}genus\ H\ HM = m$
shows $\exists GM.\ digraph\text{-}map\ G\ GM \wedge pre\text{-}digraph\text{-}map.euler\text{-}genus\ G\ GM = m \wedge edge\text{-}rev\ GM = rev\text{-}G$
⟨*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 ∃ *GM*. *digraph-map G GM* ∧ *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

13 Combinatorial Planarity and Subgraphs

lemma *out-arcs-emptyD-dominates*:
assumes *out-arcs G x* = {} **shows** ¬*x* →*G y*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *reachable-refl-iff*: *u* →* *u* ⟷ *u* ∈ *verts G*
⟨*proof*⟩

context *digraph-map* **begin**

lemma *face-cycle-set-succ*[*simp*]: *face-cycle-set* (*face-cycle-succ a*) = *face-cycle-set a*
⟨*proof*⟩

lemma *face-cycle-succ-funpow-in*[*simp*]:
(*face-cycle-succ* ^^ *n*) *a* ∈ *arcs G* ⟷ *a* ∈ *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* ⟷ *face-cycle-set x* = {*x*} (**is** *?L* ⟷ *?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 $v \rightarrow_{\textit{del-arc } u} w$ **shows** $v \rightarrow_G w$
⟨*proof*⟩

lemma (**in** *wf-digraph*) *reachable-del-arcD*:
assumes $v \rightarrow^*_{\textit{del-arc } u} w$ **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* ∈ *isolated-verts* **shows** {*u*} ∈ *sccs-verts*
⟨*proof*⟩

lemma (**in** *digraph-map*) *in-face-cycle-sets*:
a ∈ *arcs G* ⟹ *face-cycle-set a* ∈ *face-cycle-sets*
⟨*proof*⟩

lemma (**in** *digraph-map*) *heads-face-cycle-set*:
assumes *a* ∈ *arcs G*
shows *head G* ‘ *face-cycle-set a* = *tail G* ‘ *face-cycle-set a* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma (**in** *pre-digraph*) *casI-nth*:
assumes *p* ≠ [] *u* = *tail G* (*hd p*) *v* = *head G* (*last p*) ⋀*i*. *Suc i* < *length p* ⟹ *head G* (*p* ! *i*) = *tail G* (*p* ! *Suc i*)
shows *cas u p v*
⟨*proof*⟩

lemma (**in** *digraph-map*) *obtain-trail-in-fcs*:
assumes *a* ∈ *arcs G a0* ∈ *face-cycle-set a an* ∈ *face-cycle-set a*
obtains *p* **where** *trail* (*tail G a0*) *p* (*head G an*) *p* ≠ [] *hd p* = *a0* *last p* = *an* *set p* ⊆ *face-cycle-set a*
⟨*proof*⟩

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

13.1 Deleting an isolated vertex

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

begin

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

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

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

lemma *digraph-map-del-vert*:
shows *digraph-map* (*del-vert u*) *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*: $card\ (verts\ G) = Suc\ (card\ (verts\ (del\text{-}vert\ u)))$
⟨*proof*⟩

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

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

lemma *u-in-isolated-verts*: $u \in isolated\text{-}verts$
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *euler-char-dv*: *euler-char* = *1* + *H.euler-char*
⟨*proof*⟩

lemma *adj-dv*: $v \rightarrow_{\textit{del-vert } u} w \longleftrightarrow v \rightarrow_G w$
⟨*proof*⟩

lemma *reachable-del-vertD*:
assumes $v \rightarrow^*_{\textit{del-vert } u} w$ **shows** $v \rightarrow^*_G w$
⟨*proof*⟩

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

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

lemma *sccs-verts-dv*: *H.sccs-verts* = *sccs-verts* − {{*u*}} (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *card-sccs-verts-dv*: *card sccs-verts* = *Suc* (*card H.sccs-verts*)
⟨*proof*⟩

lemma *card-sccs-dv*: *card sccs* = *Suc* (*card H.sccs*)
⟨*proof*⟩

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

end

13.2 Deleting an arc pair

locale *bidel-arc* = *G*: *digraph-map* +
fixes *a*
assumes *a-in*: *a* ∈ *arcs G*

begin

abbreviation $a' \equiv \textit{edge-rev } M\ a$

definition *H* :: (′*a*,′*b*) *pre-digraph* **where**
H ≡ *pre-digraph.del-arc* (*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′*} ⊆ *arcs G* **and**
ends-in: {*tail G a*, *head G a*} ⊆ *verts G*
⟨*proof*⟩

lemma *cyclic-on-edge-succ*:
assumes *x* ∈ *verts H out-arcs H x* ≠ {}
shows *cyclic-on* (*edge-succ HM*) (*out-arcs H x*)
⟨*proof*⟩

lemma *digraph-map*: *digraph-map H HM*
⟨*proof*⟩

lemma *rev-H*: *bidel-arc.H G M a′* = *H* (**is** *?t1*)
and *rev-HM*: *bidel-arc.HM G M a′* = *HM* (**is** *?t2*)
⟨*proof*⟩

end

sublocale *bidel-arc* ⊆ *H*: *digraph-map H HM* ⟨*proof*⟩

context *bidel-arc* **begin**

lemma *a-neq-a′*: *a* ≠ *a′*
⟨*proof*⟩

lemma
arcs-G: *arcs G* = *insert a* (*insert a′* (*arcs H*)) **and**
arcs-not-in: {*a*,*a′*} ∩ *arcs H* = {}
⟨*proof*⟩

lemma *card-arcs-da*: *card* (*arcs G*) = *2* + *card* (*arcs H*)
⟨*proof*⟩

lemma *cas-da*: *H.cas* = *G.cas*
⟨*proof*⟩

lemma *reachable-daD*:
assumes *v* →**H w* **shows** *v* →**G w*
⟨*proof*⟩

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

lemma *isolated-other-da*:
assumes $u \notin \{tail\ G\ a,\ head\ G\ a\}$ **shows** $u \in H.isolated\text{-}verts \longleftrightarrow u \in G.isolated\text{-}verts$
⟨*proof*⟩

lemma *isolated-da-pre*: $H.isolated\text{-}verts = G.isolated\text{-}verts \cup$
(*if* $tail\ G\ a \in H.isolated\text{-}verts$ *then* $\{tail\ G\ a\}$ *else* $\{\}$) $\cup$
(*if* $head\ G\ a \in H.isolated\text{-}verts$ *then* $\{head\ G\ a\}$ *else* $\{\}$) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *card-isolated-verts-da0*:
$card\ H.isolated\text{-}verts = card\ G.isolated\text{-}verts + card\ (\{tail\ G\ a,\ head\ G\ a\} \cap H.isolated\text{-}verts)$
⟨*proof*⟩

lemma *segments-neq*:
assumes $segment\ G.face\text{-}cycle\text{-}succ\ a'\ a \neq \{\} \vee segment\ G.face\text{-}cycle\text{-}succ\ a\ a' \neq \{\}$
shows $segment\ G.face\text{-}cycle\text{-}succ\ a\ a' \neq segment\ G.face\text{-}cycle\text{-}succ\ a'\ a$
⟨*proof*⟩

lemma *H-fcs-eq-G-fcs*:
assumes $b \in arcs\ G\ \{b, G.face\text{-}cycle\text{-}succ\ b\} \cap \{a, a'\} = \{\}$
shows $H.face\text{-}cycle\text{-}succ\ b = G.face\text{-}cycle\text{-}succ\ b$
⟨*proof*⟩

lemma *face-cycle-set-other-da*:
assumes $\{a,\ a'\} \cap G.face\text{-}cycle\text{-}set\ b = \{\}\ b \in arcs\ G$
shows $H.face\text{-}cycle\text{-}set\ b = G.face\text{-}cycle\text{-}set\ b$
⟨*proof*⟩

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

lemma *H-fcs-in-G-fcs*:
assumes $b \in arcs\ H - (G.face\text{-}cycle\text{-}set\ a \cup G.face\text{-}cycle\text{-}set\ a')$
shows $H.face\text{-}cycle\text{-}set\ b \in G.face\text{-}cycle\text{-}sets - \{G.face\text{-}cycle\text{-}set\ a,\ G.face\text{-}cycle\text{-}set\ a'\}$
⟨*proof*⟩

lemma *face-cycle-sets-da0*:
$H.face\text{-}cycle\text{-}sets = G.face\text{-}cycle\text{-}sets - \{G.face\text{-}cycle\text{-}set\ a,\ G.face\text{-}cycle\text{-}set\ a'\}$
$\cup\ H.face\text{-}cycle\text{-}set\ `\ ((G.face\text{-}cycle\text{-}set\ a \cup G.face\text{-}cycle\text{-}set\ a') - \{a, a'\})$ (**is** *?L* = *?R*)

⟨*proof*⟩

lemma *card-fcs-aa′-le*: *card* {*G.face-cycle-set a*, *G.face-cycle-set a′*} ≤ *card G.face-cycle-sets*
⟨*proof*⟩

lemma *card-face-cycle-sets-da0*:
card H.face-cycle-sets = *card G.face-cycle-sets* − *card* {*G.face-cycle-set a*, *G.face-cycle-set a′*}
+ *card* (*H.face-cycle-set* ‘ ((*G.face-cycle-set a* ∪ *G.face-cycle-set a′*) − {*a*,*a′*}))
⟨*proof*⟩

end

locale *bidel-arc-same-face* = *bidel-arc* +
assumes *same-face*: *G.face-cycle-set a′* = *G.face-cycle-set a*
begin
lemma *a-in-o*: *a* ∈ *orbit G.face-cycle-succ a′*
⟨*proof*⟩

lemma *segment-a′-a-in*: *segment G.face-cycle-succ a′ a* ⊆ *arcs H* (**is** *?seg* ⊆ -)
⟨*proof*⟩

lemma *segment-a′-a-neD*:
assumes *segment G.face-cycle-succ a′ a* ≠ {}
shows *segment G.face-cycle-succ a′ a* ∈ *H.face-cycle-sets* (**is** *?seg* ∈ -)
⟨*proof*⟩

lemma *segment-a-a′-neD*:
assumes *segment G.face-cycle-succ a a′* ≠ {}
shows *segment G.face-cycle-succ a a′* ∈ *H.face-cycle-sets*
⟨*proof*⟩

lemma *H-fcs-full*:
assumes *SS* ⊆ *H.face-cycle-sets* **shows** *H.face-cycle-set* ‘ (⋃ *SS*) = *SS*
⟨*proof*⟩

lemma *card-fcs-gt-0*: *0* < *card G.face-cycle-sets*
⟨*proof*⟩

lemma *card-face-cycle-sets-da′*:
card H.face-cycle-sets = *card G.face-cycle-sets* − *1*
+ *card* ({*segment G.face-cycle-succ a a′*, *segment G.face-cycle-succ a′ a*, {}} − {{}})
⟨*proof*⟩

end

locale *bidel-arc-diff-face* = *bidel-arc* +
  assumes *diff-face*: *G.face-cycle-set a'* ≠ *G.face-cycle-set a*
begin

  definition *S* :: *'b set* **where**
    S ≡ *segment G.face-cycle-succ a a* ∪ *segment G.face-cycle-succ a' a'*

  lemma *diff-face-not-in*: *a* ∉ *G.face-cycle-set a' a'* ∉ *G.face-cycle-set a*
    ⟨*proof*⟩

  lemma *H-fcs-eq-for-a*:
    assumes *b* ∈ *arcs H* ∩ *G.face-cycle-set a*
    shows *H.face-cycle-set b* = *S* (**is** *?L* = *?R*)
  ⟨*proof*⟩

  lemma *HJ-fcs-eq-for-a'*:
    assumes *b* ∈ *arcs H* ∩ *G.face-cycle-set a'*
    shows *H.face-cycle-set b* = *S*
  ⟨*proof*⟩

  lemma *card-face-cycle-sets-da'*:
      card H.face-cycle-sets = *card G.face-cycle-sets* − *card* {*G.face-cycle-set a*, *G.face-cycle-set a'*} + (*if S* = {} *then 0 else 1*)
  ⟨*proof*⟩

end

locale *bidel-arc-biconnected* = *bidel-arc* +
  assumes *reach-a*: *tail G a* →** *H head G a*
begin

  lemma *reach-a'*: *tail G a'* →** *H head G a'*
    ⟨*proof*⟩

  lemma
    tail-a': *tail G a'* = *head G a* **and**
    head-a': *head G a'* = *tail G a*
    ⟨*proof*⟩

  lemma *reachable-daI*:
    assumes *v* →** *G w* **shows** *v* →** *H w*
  ⟨*proof*⟩

  lemma *reachable-da*: *v* →** *H w* ⟷ *v* →** *G w*
    ⟨*proof*⟩

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

lemma *card-sccs-da*: *card H.sccs = card G.sccs*
⟨*proof*⟩

end

locale *bidel-arc-not-biconnected = bidel-arc +*
assumes *not-reach-a*: $\neg tail\ G\ a \rightarrow^*_H head\ G\ a$
begin

lemma *H-awalkI*: *G.awalk u p v* $\Longrightarrow$ $\{a,a'\} \cap set\ p = \{\}$ $\Longrightarrow$ *H.awalk u p v*
⟨*proof*⟩

lemma *tail-neq-head*: *tail G a* $\neq$ *head G a*
⟨*proof*⟩

lemma *scc-of-tail-neq-head*: *H.scc-of (tail G a)* $\neq$ *H.scc-of (head G a)*
⟨*proof*⟩

lemma *scc-of-G-tail*:
assumes *u* $\in$ *G.scc-of (tail G a)*
shows *H.scc-of u = H.scc-of (tail G a)* $\vee$ *H.scc-of u = H.scc-of (head G a)*
⟨*proof*⟩

lemma *scc-of-other*:
assumes *u* $\notin$ *G.scc-of (tail G a)*
shows *H.scc-of u = G.scc-of u*
⟨*proof*⟩

lemma *scc-of-tail-inter*:
tail G a $\in$ *G.scc-of (tail G a)* $\cap$ *H.scc-of (tail G a)*
⟨*proof*⟩

lemma *scc-of-head-inter*:
head G a $\in$ *G.scc-of (tail G a)* $\cap$ *H.scc-of (head G a)*
⟨*proof*⟩

lemma *G-scc-of-tail-not-in*: *G.scc-of (tail G a)* $\notin$ *H.sccs-verts*
⟨*proof*⟩

lemma *H-scc-of-a-not-in*:
H.scc-of (tail G a) $\notin$ *G.sccs-verts H.scc-of (head G a)* $\notin$ *G.sccs-verts*
⟨*proof*⟩

lemma *scc-verts-da*:
H.sccs-verts = (G.sccs-verts − {G.scc-of (tail G a)}) ∪ {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* ⊆ *bidel-arc-same-face*
⟨*proof*⟩

locale *bidel-arc-tail-conn = bidel-arc* +
assumes *conn-tail*: *tail G a* ∉ *H.isolated-verts*

locale *bidel-arc-head-conn = bidel-arc* +
assumes *conn-head*: *head G a* ∉ *H.isolated-verts*

locale *bidel-arc-tail-isolated = bidel-arc* +
assumes *isolated-tail*: *tail G a* ∈ *H.isolated-verts*

locale *bidel-arc-head-isolated = bidel-arc* +
assumes *isolated-head*: *head G a* ∈ *H.isolated-verts*
begin

lemma *G-edge-succ-a'-no-loop*:
assumes *no-loop-a*: *head G a* ≠ *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* ≠ *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-neigh* :: *'b* **where**
a-neigh ≡ *SOME b. G.face-cycle-succ b = a*

lemma *face-cycle-succ-a-neigh*: *G.face-cycle-succ a-neigh = a*
⟨*proof*⟩

lemma *a-neigh-in*: *a-neigh* ∈ *arcs G*
⟨*proof*⟩

lemma *a-neigh-neq-a*: $a\text{-}neigh \neq a$
⟨*proof*⟩

lemma *a-neigh-neq-a'*: $a\text{-}neigh \neq a'$
⟨*proof*⟩

lemma *edge-rev-a-neigh-neq*: $edge\text{-}rev\ M\ a\text{-}neigh \neq a'$
⟨*proof*⟩

lemma *edge-succ-a-neq*: $edge\text{-}succ\ M\ a \neq a'$
⟨*proof*⟩

lemma *H-face-cycle-succ-a-neigh*: $H.face\text{-}cycle\text{-}succ\ a\text{-}neigh = G.face\text{-}cycle\text{-}succ\ a'$
⟨*proof*⟩

lemma *H-fcs-a-neigh*: $H.face\text{-}cycle\text{-}set\ a\text{-}neigh = segment\ G.face\text{-}cycle\text{-}succ\ a'\ a$ (**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* ⊆ *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*: $card\ H.isolated\text{-}verts = Suc\ (card\ G.isolated\text{-}verts)$
⟨*proof*⟩

lemma
G-edge-succ-a[*simp*]: $edge\text{-}succ\ M\ a = a'$ (**is** *?t1*) **and**
G-edge-succ-a'[*simp*]: $edge\text{-}succ\ M\ a' = a$ (**is** *?t2*)
⟨*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*} **and**
G-face-cycle-set-a'[*simp*]: *G.face-cycle-set a'* = {*a'*}
⟨*proof*⟩

end

sublocale *bidel-arc-isolated-loop* ⊆ *bidel-arc-diff-face*
⟨*proof*⟩

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

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

lemma *euler-genus-da*: *H.euler-genus* = *G.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*: *head G a* ≠ *tail G a*
⟨*proof*⟩

lemma *card-isolated-verts-da*: *card H.isolated-verts* = *Suc* (*Suc* (*card G.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* ⊆ *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* ≠ {}
⟨*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*: $H.euler\text{-}genus = G.euler\text{-}genus$
⟨*proof*⟩

end

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

lemma *isolated-verts-da*: $H.isolated\text{-}verts = G.isolated\text{-}verts$
⟨*proof*⟩

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

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

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

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

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

end

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

sublocale *bidel-arc-biconnected-non-triv* ⊆ *bidel-arc-head-conn*
⟨*proof*⟩

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

lemma *isolated-verts-da*: $H.isolated\text{-}verts = G.isolated\text{-}verts$
⟨*proof*⟩

end

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

sublocale *bidel-arc-biconnected-same* ⊆ *bidel-arc-same-face-tail-conn*
⟨*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*) ≥ (*card G.face-cycle-sets*)
⟨*proof*⟩

lemma *euler-genus-da*: *H.euler-genus* ≤ *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* ≠ {*a*} ∨ *G.face-cycle-set a′* ≠ {*a′*}
⟨*proof*⟩

lemma *S-ne*: *S* ≠ {}
⟨*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* ≤ *G.euler-genus*
⟨*proof*⟩
end

13.3 Modifying *edge-rev*

definition (**in** *pre-digraph-map*) *rev-swap* :: *′b* ⇒ *′b* ⇒ *′b pre-map* **where**
rev-swap a b = (‖ *edge-rev* = *perm-swap a b* (*edge-rev M*), *edge-succ* = *perm-swap a b* (*edge-succ M*) ‖)

context *digraph-map* **begin**

lemma *digraph-map-rev-swap*:
assumes *arc-to-ends G a* = *arc-to-ends G b* {*a*,*b*} ⊆ *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*} ⊆ *arcs G*
shows *pre-digraph-map.euler-genus G* (*rev-swap a b*) = *euler-genus*
⟨*proof*⟩

end

13.4 Conclusion

lemma *bidirected-subgraph-obtain*:
assumes *sg*: *subgraph H G arcs H* ≠ *arcs G*
assumes *fin*: *finite* (*arcs G*)
assumes *bidir*: ∃ *rev*. *bidirected-digraph G rev* ∃ *rev*. *bidirected-digraph H rev*
obtains *a a'* **where** {*a*,*a'*} ⊆ *arcs G* − *arcs H a'* ≠ *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*: ∃ *rev*. *bidirected-digraph H rev*
obtains *HM* **where** *digraph-map H HM pre-digraph-map.euler-genus H HM* ≤ *pre-digraph-map.euler-genus G GM*
⟨*proof*⟩

lemma (**in** *digraph-map*) *nonneg-euler-genus*: *0* ≤ *euler-genus*
⟨*proof*⟩

lemma *subgraph-comb-planar*:
assumes *subgraph G H comb-planar H* ∃ *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* (λ-. *V*) (λ-. *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* (λ-. *V*) (λ-. *c*)
⟨*proof*⟩

end

14 Implementation of a Non-Planarity Checker

theory *Check-Non-Planarity-Impl*
imports
Simpl.Vcg
Simpl-Anno
Graph-Theory.Graph-Theory
begin

14.1 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*)}

14.2 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
```

15 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
```

15.1 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*: *verts3 G* ⊆ *V V* ⊆ *pverts G u* ∈ *V*
assumes *p*: *awalk u p v set* (*inner-verts p*) ∩ *V* = {} *progressing p*
shows *distinct* (*inner-verts p*)
⟨*proof*⟩

lemma (**in** *wf-digraph*) *inner-verts-conv'*:
assumes *awalk u p v 2* ≤ *length p* **shows** *inner-verts p* = *awalk-verts* (*head G* (*hd p*)) (*butlast* (*tl p*))
⟨*proof*⟩

lemma *verts3-in-verts*:
assumes *x* ∈ *verts3 G* **shows** *x* ∈ *verts G*
⟨*proof*⟩

lemma (**in** *pair-graph*) *deg2-awalk-is-iapath*:
assumes *V*: *verts3 G* ⊆ *V V* ⊆ *pverts G*
assumes *p*: *awalk u p v set* (*inner-verts p*) ∩ *V* = {} *progressing p*
assumes *in-V*: *u* ∈ *V v* ∈ *V*
assumes *u* ≠ *v*
shows *gen-iapath V u p v*
⟨*proof*⟩

lemma (**in** *pair-graph*) *inner-verts-min-degree*:
assumes *walk-p*: *awalk u p v* **and** *progress*: *progressing p*
and *w-p*: *w* ∈ *set* (*inner-verts p*)
shows *2* ≤*in-degree G w*
⟨*proof*⟩

lemma (**in** *pair-pseudo-graph*) *gen-iapath-same2E*:
assumes *verts3 G* ⊆ *V V* ⊆ *pverts G*
and *gen-iapath V u p v gen-iapath V w q x*
and *e* ∈ *set p e* ∈ *set q*
obtains *p* = *q*
⟨*proof*⟩

definition *mk-graph'* :: *IGraph* ⇒ *ig-vertex pair-pre-digraph* **where**
mk-graph' IG ≡ (| *pverts* = *set* (*ig-verts IG*), *parcs* = *set* (*ig-arcs IG*)|)

definition *mk-graph* :: *IGraph* ⇒ *ig-vertex pair-pre-digraph* **where**
mk-graph IG ≡ *mk-symmetric* (*mk-graph' IG*)

lemma *verts-mkg'*: *pverts* (*mk-graph' G*) = *set* (*ig-verts G*)
⟨*proof*⟩

lemma *arcs-mkg'*: *parcs* (*mk-graph' G*) = *set* (*ig-arcs G*)
⟨*proof*⟩

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

lemma *verts-mkg*: *pverts* (*mk-graph G*) = *set* (*ig-verts G*)
⟨*proof*⟩

lemma *parcs-mk-symmetric-symcl*: *parcs* (*mk-symmetric G*) = (*arcs-ends G*)^s
⟨*proof*⟩

lemma *arcs-mkg*: *parcs* (*mk-graph G*) = (*set* (*ig-arcs G*))^s
⟨*proof*⟩

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

definition *iadj* :: *IGraph* ⇒ *ig-vertex* ⇒ *ig-vertex* ⇒ *bool* **where**
iadj G u v ≡ (*u*,*v*) ∈ *set* (*ig-arcs G*) ∨ (*v*,*u*) ∈ *set* (*ig-arcs G*)

definition *loop-free G* ≡ (∀ *e* ∈ *parcs G*. *fst e* ≠ *snd e*)

lemma *ig-opposite-simps*:
ig-opposite G (*u*,*v*) *u* = *v 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 $\longleftrightarrow$ *pair-fin-digraph* (*mk-graph′ G*)
⟨*proof*⟩

lemma *IGraph-inv-conv′*:
IGraph-inv G $\longleftrightarrow$ *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*) $\longleftrightarrow$ (∀ *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) $\longleftrightarrow$ *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* ∈ *set* (*ig-verts G*) *v* ∈ *set* (*ig-verts G*)
⟨*proof*⟩

lemma *card-ig-neighbors-indegree*:
assumes *IGraph-inv G*
shows *card* (*ig-neighbors G u*) = *in-degree* (*mk-graph G*) *u*

⟨*proof*⟩

lemma *iadjD*:
assumes *iadj G u v*
shows ∃ *e* ∈ *set* (*ig-in-out-arcs G u*). (*e* = (*u*,*v*) ∨ *e* = (*v*,*u*))
⟨*proof*⟩

lemma
ig-verts-empty[*simp*]: *ig-verts ig-empty* = [] **and**
ig-verts-add-e[*simp*]: *ig-verts* (*ig-add-e G u v*) = *ig-verts G* **and**
ig-verts-add-v[*simp*]: *ig-verts* (*ig-add-v G v*) = *ig-verts G* @ (*if v* ∈ *set* (*ig-verts G*) *then* [] *else* [*v*])
⟨*proof*⟩

lemma
ig-arcs-empty[*simp*]: *ig-arcs ig-empty* = [] **and**
ig-arcs-add-e[*simp*]: *ig-arcs* (*ig-add-e G u v*) = *ig-arcs G* @ [(*u*,*v*)] **and**
ig-arcs-add-v[*simp*]: *ig-arcs* (*ig-add-v G v*) = *ig-arcs G*
⟨*proof*⟩

15.2 Total Correctness

15.2.1 Procedure *is-subgraph*

definition *is-subgraph-verts-inv* :: *IGraph* ⇒ *IGraph* ⇒ *nat* ⇒ *bool* **where**
is-subgraph-verts-inv G H i ≡ *set* (*take i* (*ig-verts G*)) ⊆ *set* (*ig-verts H*)

definition *is-subgraph-arcs-inv* :: *IGraph* ⇒ *IGraph* ⇒ *nat* ⇒ *bool* **where**
is-subgraph-arcs-inv G H i ≡ ∀ *j* < *i*. *let* (*u*,*v*) = *ig-arcs G* ! *j in*
((*u*,*v*) ∈ *set* (*ig-arcs H*) ∨ (*v*,*u*) ∈ *set* (*ig-arcs H*))
∧ *u* ∈ *set* (*ig-verts G*) ∧ *v* ∈ *set* (*ig-verts G*)

lemma *is-subgraph-verts-0*: *is-subgraph-verts-inv G H 0*
⟨*proof*⟩

lemma *is-subgraph-verts-step*:
assumes *is-subgraph-verts-inv G H i ig-verts G* ! *i* ∈ *set* (*ig-verts H*)
assumes *i* < *length* (*ig-verts G*)
shows *is-subgraph-verts-inv G H* (*Suc i*)
⟨*proof*⟩

lemma *is-subgraph-verts-last*:
is-subgraph-verts-inv G H (*length* (*ig-verts G*)) ⟷ *pverts* (*mk-graph G*) ⊆ *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 ∈ set (ig-arcs H) ∨ (snd e, fst e) ∈ set (ig-arcs H)
fst e ∈ set (ig-verts G) snd e ∈ 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)) ⟷ parcs (mk-graph G) ⊆ parcs (mk-graph H) ∧ 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 *∀ i < length (ig-verts G). ig-verts G ! i ∈ set (ig-verts H)*
∀ i < length (ig-arcs G). let (u,v) = ig-arcs G ! i in
((u,v)∈ set (ig-arcs H) ∨ (v,u) ∈ set (ig-arcs H))
∧ u ∈ set (ig-verts G) ∧ v ∈ set (ig-verts G)
⟨*proof*⟩

lemma (**in** *is-subgraph-impl*) *is-subgraph-spec*:
∀σ. Γ ⊢$_t$ {|σ. IGraph-inv´ H |}´ R :== PROC is-subgraph(G,´ H) {|´ G = ᵒ G ∧´ H = ᵒ H ∧´ R = (subgraph (mk-graph´ G) (mk-graph´ H) ∧ IGraph-inv´ G)|}
⟨*proof*⟩

15.2.2 Procedure *is-loop-free*

definition *is-loopfree-inv G k ≡ ∀ j<k. fst (ig-arcs G ! j) ≠ snd (ig-arcs G ! j)*

lemma *is-loopfree-0*:
is-loopfree-inv G 0
⟨*proof*⟩

lemma *is-loopfree-step1*:
assumes *is-loopfree-inv G n*

assumes *fst* (*ig-arcs G* ! *n*) ≠ *snd* (*ig-arcs G* ! *n*)
assumes *n* < *ig-arcs-cnt G*
shows *is-loopfree-inv G* (*Suc n*)
⟨*proof*⟩

lemma *is-loopfree-step2*:
assumes *loop-free* (*mk-graph G*)
assumes *n* < *ig-arcs-cnt G*
shows *fst* (*ig-arcs G* ! *n*) ≠ *snd* (*ig-arcs G* ! *n*)
⟨*proof*⟩

lemma *is-loopfree-last*:
assumes *is-loopfree-inv G* (*ig-arcs-cnt G*)
shows *loop-free* (*mk-graph G*)
⟨*proof*⟩

lemma (**in** *is-loopfree-impl*) *is-loopfree-spec*:
∀ σ. Γ ⊢$_t$ ⦃σ. *IGraph-inv*´ *G* ⦄´ *R* :== *PROC is-loopfree*(*G*) ⦃´ *G* = ᵟ *G* ∧´ *R* = *loop-free* (*mk-graph*´ *G*) ⦄
⟨*proof*⟩

15.2.3 Procedure *select-nodes*

definition *select-nodes-inv* :: *IGraph* ⇒ *IGraph* ⇒ *nat* ⇒ *bool* **where**
select-nodes-inv G H i ≡ *set* (*ig-verts H*) = {*v* ∈ *set* (*take i* (*ig-verts G*)). *card* (*ig-neighbors G v*) ≥ *3*} ∧ *IGraph-inv H*

lemma *select-nodes-inv-step*:
fixes *G H i*
defines *v* ≡ *ig-verts G* ! *i*
assumes *G-inv*: *IGraph-inv G*
assumes *sni-inv*: *select-nodes-inv G H i*
assumes *less*: *i* < *ig-verts-cnt G*
assumes *H'*: *H'* = (*if 3* ≤ *card* (*ig-neighbors G v*) *then ig-add-v H v else H*)
shows *select-nodes-inv G H'* (*Suc i*)
⟨*proof*⟩

definition *select-nodes-prop* :: *IGraph* ⇒ *IGraph* ⇒ *bool* **where**
select-nodes-prop G H ≡ *pverts* (*mk-graph H*) = *verts3* (*mk-graph G*)

lemma (**in** *select-nodes-impl*) *select-nodes-spec*:
∀ σ. Γ ⊢$_t$ ⦃σ. *IGraph-inv*´ *G*⦄´ *R* :== *PROC select-nodes*(*G*)
⦃ *select-nodes-prop* ᵟ *G*´ *R* ∧ *IGraph-inv*´ *R* ∧ *set* (*ig-arcs*´ *R*) = {}⦄
⟨*proof*⟩

15.2.4 Procedure *find-endpoint*

definition *find-endpoint-path-inv* **where**
find-endpoint-path-inv G H len u v w x ≡
∃ *p*. *pre-digraph.awalk* (*mk-graph G*) *u p x* ∧ *length p* = *len* ∧

hd p = (u,v) ∧ last p = (w, x) ∧
set (pre-digraph.inner-verts (mk-graph G) p) ∩ set (ig-verts H) = {} ∧
progressing p

definition *find-endpoint-arcs-inv* **where**
find-endpoint-arcs-inv G found k v0 v1 v0′ v1′ ≡
(found ⟶ (∃ i < k. v1′ = ig-opposite G (ig-in-out-arcs G v1 ! i) v1 ∧ v0′ = v1 ∧ v0 ≠ v1′)) ∧
(¬found ⟶ (∀ i < k. v0 = ig-opposite G (ig-in-out-arcs G v1 ! i) v1) ∧ v0 = v0′ ∧ v1 = v1′)

lemma *find-endpoint-path-first*:
assumes *iadj G u v u ≠ v IGraph-inv G*
shows *find-endpoint-path-inv G H (Suc 0) u v u v*
⟨*proof*⟩

lemma *find-endpoint-arcs-0*:
find-endpoint-arcs-inv G False 0 v0 v1 v0 v1
⟨*proof*⟩

lemma *find-endpoint-path-lastE*:
assumes *find-endpoint-path-inv G H len u v w x*
assumes *ig*: *IGraph-inv G* **and** *lf*: *loop-free (mk-graph G)*
assumes *snp*: *select-nodes-prop G H*
assumes *0 < len*
assumes *u*: *u ∈ set (ig-verts H)*
obtains *p* **where** *pre-digraph.awalk (mk-graph G) u ((u,v) # p) x*
and *progressing ((u,v) # p)*
and *set (pre-digraph.inner-verts (mk-graph G) ((u,v) # p)) ∩ set (ig-verts H) = {}*
and *len ≤ ig-verts-cnt G*
⟨*proof*⟩

lemma *find-endpoint-path-last1*:
assumes *find-endpoint-path-inv G H len u v w x*
assumes *ig*: *IGraph-inv G* **and** *lf*: *loop-free (mk-graph G)*
assumes *snp*: *select-nodes-prop G H*
assumes *0 < len*
assumes *mem*: *u ∈ set (ig-verts H) x ∈ set (ig-verts H) u ≠ x*
shows *∃ p. pre-digraph.iapath (mk-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-graph G)*
assumes *snp*: *select-nodes-prop G H*
assumes *0 < len*
assumes *mem*: *u ∈ set (ig-verts H)*
assumes *iapath*: *pre-digraph.iapath (mk-graph G) u ((u,v) # p) x*

shows *False*
⟨*proof*⟩

lemma *find-endpoint-arcs-last*:
assumes *arcs*: *find-endpoint-arcs-inv G False* (*length* (*ig-in-out-arcs G v1*)) *v0 v1 v0a v1a*
assumes *path*: *find-endpoint-path-inv G H len v-tail v-next v0 v1*
assumes *ig*: *IGraph-inv G* **and** *lf*: *loop-free* (*mk-graph G*)
assumes *snp*: *select-nodes-prop G H*
assumes *mem*: *v-tail* ∈ *set* (*ig-verts H*)
assumes *0* < *len*
shows ¬ *pre-digraph.iapath* (*mk-graph G*) *v-tail* ((*v-tail*, *v-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-in-out-arcs G v1* ! *k*) *v1′* ≠ *v0′*
obtains *v0* = *v0′ v1* = *v1′ find-endpoint-arcs-inv G True* (*Suc k*) *v0 v1 v1* (*ig-opposite G* (*ig-in-out-arcs G v1* ! *k*) *v1*)
⟨*proof*⟩

lemma *find-endpoint-arcs-step2E*:
assumes *find-endpoint-arcs-inv G False k v0 v1 v0′ v1′*
assumes *ig-opposite G* (*ig-in-out-arcs G v1* ! *k*) *v1′* = *v0′*
obtains *v0* = *v0′ v1* = *v1′ find-endpoint-arcs-inv G False* (*Suc k*) *v0 v1 v0 v1*
⟨*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 ≤ *length* (*ig-in-out-arcs G x*)
assumes *ig*: *IGraph-inv G*
assumes *not-end*: *x* ∉ *set* (*ig-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 ¬ (∃ *p w*. *pre-digraph.iapath* (*mk-graph G*) *u* ((*u*, *v*) # *p*) *w*)
⟨*proof*⟩

lemma (**in** *find-endpoint-impl*) *find-endpoint-spec*:
∀ σ. Γ ⊢$_t$ ⦃σ. *select-nodes-prop´ G´ H* ∧ *loop-free* (*mk-graph´ G*) ∧´ *v-tail* ∈ *set* (*ig-verts´ H*) ∧ *iadj´ G´ v-tail´ v-next* ∧ *IGraph-inv´ G*⦄
´*R* :== *PROC find-endpoint*(*G*,´*H*,´*v-tail*,´*v-next*)
⦃*case´ R of None* ⇒ ¬(∃ *p w*. *pre-digraph.iapath* (*mk-graph* σ *G*) σ *v-tail* ((σ *v-tail*, σ *v-next*) # *p*) *w*)
| *Some w* ⇒ (∃ *p*. *pre-digraph.iapath* (*mk-graph* σ *G*) σ *v-tail* ((σ *v-tail*, σ *v-next*) # *p*) *w*) ⦄

⟨*proof*⟩

15.2.5 Procedure *contract*

definition *contract-iter-nodes-inv* **where**
contract-iter-nodes-inv G H k ≡
set (*ig-arcs H*) = (⋃ *i*<*k*. {(*u*,*v*). *u* = (*ig-verts H* ! *i*) ∧ (∃ *p*. *pre-digraph.iapath* (*mk-graph G*) *u p v*)})

definition *contract-iter-adj-inv* :: *IGraph* ⇒ *IGraph* ⇒ *IGraph* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
contract-iter-adj-inv G H0 H u l ≡ (*set* (*ig-arcs H*) − ({*u*} × *UNIV*) = *set* (*ig-arcs H0*)) ∧
ig-verts H = *ig-verts H0* ∧
(∀ *v*. (*u*,*v*) ∈ *set* (*ig-arcs H*) ⟷
(∃ *j*<*l*. ∃ *p*. *pre-digraph.iapath* (*mk-graph G*) *u* ((*u*, *ig-opposite G* (*ig-in-out-arcs G u* ! *j*) *u*) # *p*) *v*)))

lemma *contract-iter-adj-invE*:
assumes *contract-iter-adj-inv G H0 H u l*
obtains *set* (*ig-arcs H*) − ({*u*} × *UNIV*) = *set* (*ig-arcs H0*) *ig-verts H* = *ig-verts H0*
⋀ *v*. (*u*,*v*) ∈ *set* (*ig-arcs H*) ⟷ (∃ *j*<*l*. ∃ *p*. *pre-digraph.iapath* (*mk-graph G*) *u* ((*u*, *ig-opposite G* (*ig-in-out-arcs G u* ! *j*) *u*) # *p*) *v*))
⟨*proof*⟩

lemma *contract-iter-adj-inv-def'*:
contract-iter-adj-inv G H0 H u l ⟷ (
set (*ig-arcs H*) − ({*u*} × *UNIV*) = *set* (*ig-arcs H0*)) ∧ *ig-verts H* = *ig-verts H0* ∧
(∀ *v*. ((∃ *j*<*l*. ∃ *p*. *pre-digraph.iapath* (*mk-graph G*) *u* ((*u*, *ig-opposite G* (*ig-in-out-arcs G u* ! *j*) *u*) # *p*) *v*) ⟶ (*u*,*v*) ∈ *set* (*ig-arcs H*)) ∧
((*u*,*v*) ∈ *set* (*ig-arcs H*) ⟶ (∃ *j*<*l*. ∃ *p*. *pre-digraph.iapath* (*mk-graph G*) *u* ((*u*, *ig-opposite G* (*ig-in-out-arcs G u* ! *j*) *u*) # *p*) *v*))))
⟨*proof*⟩

lemma *select-nodes-prop-add-e*[*simp*]:
select-nodes-prop G (*ig-add-e H u v*) = *select-nodes-prop G H*
⟨*proof*⟩

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*: $\bigwedge p\ w.\ \neg$*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 H u* (*Suc l*)
⟨*proof*⟩

definition *contract-iter-adj-prop* **where**
contract-iter-adj-prop G H0 H u ≡ *ig-verts H* = *ig-verts H0*
∧ *set* (*ig-arcs H*) = *set* (*ig-arcs H0*) ∪ ({*u*} × {*v*. ∃ *p*. *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* ∈ *set* (*ig-verts H*) **shows** *u* ∈ *set* (*ig-verts G*)
⟨*proof*⟩

lemma *igraph-ig-add-eI*:
assumes *IGraph-inv G*
assumes *u* ∈ *set* (*ig-verts G*) *v* ∈ *set* (*ig-verts G*)
shows *IGraph-inv* (*ig-add-e G u v*)
⟨*proof*⟩

lemma *snp-iapath-ends-in*:

assumes *select-nodes-prop G H*
assumes *pre-digraph.iapath* (*mk-graph G*) *u p v*
shows *u* ∈ *set* (*ig-verts H*) *v* ∈ *set* (*ig-verts H*)
⟨*proof*⟩

lemma *contract-iter-nodes-last*:
assumes *nodes*: *contract-iter-nodes-inv G H* (*ig-verts-cnt H*)
assumes *snp*: *select-nodes-prop G H*
assumes *igraph*: *IGraph-inv G*
shows *mk-graph′ H* = *contr-graph* (*mk-graph G*) (**is** *?t1*)
and *symmetric* (*mk-graph′ H*) (**is** *?t2*)
⟨*proof*⟩

lemma (**in** *contract-impl*) *contract-spec*:
∀ σ. Γ ⊢$_t$ ⦃σ. *select-nodes-prop′ G′ H* ∧ *IGraph-inv′ G* ∧ *loop-free* (*mk-graph′ G*) ∧ *IGraph-inv′ H* ∧ *set* (*ig-arcs′ H*) = {}⦄
´*R* :== *PROC contract*(*G*,´*H*)
⦃*G* = $^{\sigma}$ *G* ∧ *mk-graph″ R* = *contr-graph* (*mk-graph′ G*) ∧ *symmetric* (*mk-graph′* ´*R*) ∧ *IGraph-inv′ R*⦄
⟨*proof*⟩

15.2.6 Procedure *is-K33*

definition *is-K33-colorize-inv* :: *IGraph* ⇒ *ig-vertex* ⇒ *nat* ⇒ (*ig-vertex* ⇒ *bool*) ⇒ *bool* **where**
is-K33-colorize-inv G u k blue ≡ ∀ *v* ∈ *set* (*ig-verts G*). *blue v* ⟷
(∃ *i* < *k*. *v* = *ig-opposite G* (*ig-in-out-arcs G u* ! *i*) *u*)

definition *is-K33-component-size-inv* :: *IGraph* ⇒ *nat* ⇒ (*ig-vertex* ⇒ *bool*) ⇒ *nat* ⇒ *bool* **where**
is-K33-component-size-inv G k blue cnt ≡ *cnt* = *card* {*i*. *i* < *k* ∧ *blue* (*ig-verts G* ! *i*)}

definition *is-K33-outer-inv* :: *IGraph* ⇒ *nat* ⇒ (*ig-vertex* ⇒ *bool*) ⇒ *bool* **where**
is-K33-outer-inv G k blue ≡ ∀ *i* < *k*. ∀ *v* ∈ *set* (*ig-verts G*).
blue (*ig-verts G* ! *i*) = *blue v* ⟷ (*ig-verts G* ! *i*, *v*) ∉ *set* (*ig-arcs G*)

definition *is-K33-inner-inv* :: *IGraph* ⇒ *nat* ⇒ *nat* ⇒ (*ig-vertex* ⇒ *bool*) ⇒ *bool* **where**
is-K33-inner-inv G k l blue ≡ ∀ *j* < *l*.
blue (*ig-verts G* ! *k*) = *blue* (*ig-verts G* ! *j*) ⟷ (*ig-verts G* ! *k*, *ig-verts G* ! *j*) ∉ *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 $u \in$ *set* (*ig-verts G*)
shows *is-K33-colorize-inv G u* (*length* (*ig-in-out-arcs G u*)) *blue*
$=$ ($\forall\, v \in$ *set* (*ig-verts G*). *blue v* $\longleftrightarrow$ *iadj G u v*) (**is** *?L* $=$ *?R*)
⟨*proof*⟩

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

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

lemma *is-K33-inner-last*:
is-K33-inner-inv G k (*ig-verts-cnt G*) *blue* $\longleftrightarrow$ ($\forall\, v \in$ *set* (*ig-verts G*).
blue (*ig-verts G* ! *k*) $=$ *blue v* $\longleftrightarrow$ (*ig-verts G* ! *k*, *v*) $\notin$ *set* (*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*: ¬*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*)) ⟷ (*ig-verts G* ! *i*, *ig-verts G* ! *j*) ∉ *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* ≡ *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* ∈ *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* ⇒ (*ig-vertex* ⇒ *bool*) ⇒ *bool* **where**
is-K33-colorize-inv-last G blue ≡ *is-K33-colorize-inv G* (*ig-verts G* ! *0*) (*length* (*ig-in-out-arcs G* (*ig-verts G* ! *0*))) *blue*

abbreviation (*input*) *is-K33-component-size-inv-last* :: *IGraph* ⇒ (*ig-vertex* ⇒ *bool*) ⇒ *bool* **where**
is-K33-component-size-inv-last G blue ≡ *is-K33-component-size-inv G* (*ig-verts-cnt G*) *blue 3*

lemma *is-K33-outerD*:
assumes *is-K33-outer-inv G* (*ig-verts-cnt G*) *blue*
assumes *i* < *ig-verts-cnt G j* < *ig-verts-cnt G*

shows (*blue* (*ig-verts G ! i*) = *blue* (*ig-verts G ! j*)) ⟷ (*ig-verts G ! i*, *ig-verts G ! j*) ∉ *set* (*ig-arcs G*)
⟨*proof*⟩

lemma (**in** *is-K33-impl*) *is-K33-spec*:
∀ σ. Γ ⊢$_t$ ⦃σ. *IGraph-inv*´ *G* ∧ *symmetric* (*mk-graph*″ *G*)⦄
´*R* :== *PROC is-K33*(*G*)
⦃´*G* = $^{\sigma}$ *G* ∧´ *R* = $K_{3,3}$(*mk-graph*″ *G*)⦄
⟨*proof*⟩

15.2.7 Procedure *is-K5*

definition
is-K5-outer-inv G k ≡ ∀ *i*<*k*. ∀ *v* ∈ *set* (*ig-verts G*). *ig-verts G ! i* ≠ *v*
⟷ (*ig-verts G ! i*, *v*) ∈ *set* (*ig-arcs G*)

definition
is-K5-inner-inv G k l ≡ ∀ *j* < *l*. *ig-verts G ! k* ≠ *ig-verts G ! j*
⟷ (*ig-verts G ! k*, *ig-verts G ! j*) ∈ *set* (*ig-arcs G*)

lemma *K5-card*:
assumes K_5 (*mk-graph*′ *G*) **shows** *ig-verts-cnt G* = 5
⟨*proof*⟩

lemma *is-K5-inner-0*: *is-K5-inner-inv G k 0*
⟨*proof*⟩

lemma *is-K5-inner-last*:
assumes *l* = *ig-verts-cnt G*
shows *is-K5-inner-inv G k l* ⟷ (∀ *v* ∈ *set* (*ig-verts G*). *ig-verts G ! k* ≠ *v*
⟷ (*ig-verts G ! k*, *v*) ∈ *set* (*ig-arcs G*))
⟨*proof*⟩

lemma *is-K5-outer-step*:
assumes *is-K5-outer-inv G k*
assumes *is-K5-inner-inv G k* (*ig-verts-cnt G*)
shows *is-K5-outer-inv G* (*Suc k*)
⟨*proof*⟩

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* < *ig-verts-cnt G*

assumes $k \neq l \longleftrightarrow (\mathit{ig\text{-}verts}\ G\ !\ k,\ \mathit{ig\text{-}verts}\ G\ !\ l) \in \mathit{set}\ (\mathit{ig\text{-}arcs}\ G)$
shows *is-K5-inner-inv G k* (*Suc l*)
⟨*proof*⟩

lemma *iK5E*:
assumes $K_5\ (\mathit{mk\text{-}graph}'\ G)$
obtains $\mathit{ig\text{-}verts\text{-}cnt}\ G = 5\ [\![i < \mathit{ig\text{-}verts\text{-}cnt}\ G;\ j < \mathit{ig\text{-}verts\text{-}cnt}\ G]\!] \Longrightarrow i \neq j \longleftrightarrow (\mathit{ig\text{-}verts}\ G\ !\ i,\ \mathit{ig\text{-}verts}\ G\ !\ j) \in \mathit{set}\ (\mathit{ig\text{-}arcs}\ G)$
⟨*proof*⟩

lemma (**in** *is-K5-impl*) *is-K5-spec*:
$\forall \sigma.\ \Gamma \vdash_t \{\!|\sigma.\ \mathit{IGraph\text{-}inv}'\ G \wedge \mathit{symmetric}\ (\mathit{mk\text{-}graph}''\ G)|\!\}$
$'R :== \mathit{PROC\ is\text{-}K5}(G)$
$\{\!|{}'G = {}^{\sigma}G \wedge {}'R = K_5(\mathit{mk\text{-}graph}''\ G)\ |\!\}$
⟨*proof*⟩

15.2.8 Soundness of the Checker

lemma *planar-theorem*:
assumes *pair-pseudo-graph G pair-pseudo-graph K*
and *subgraph K G*
and $K_{3,3}\ (\mathit{contr\text{-}graph}\ K) \vee K_5\ (\mathit{contr\text{-}graph}\ K)$
shows $\neg \mathit{kuratowski\text{-}planar}\ G$
⟨*proof*⟩

definition *witness* :: *'a pair-pre-digraph* ⇒ *'a pair-pre-digraph* ⇒ *bool* **where**
witness G K ≡ *loop-free K* ∧ *pair-pseudo-graph K* ∧ *subgraph K G*
∧ ($K_{3,3}$ (*contr-graph K*) ∨ K_5 (*contr-graph K*))

lemma *witness* (*mk-graph G*) (*mk-graph K*) ⟷ *pair-pre-digraph.certify* (*mk-graph G*) (*mk-graph K*) ∧ *loop-free* (*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*:
$\forall \sigma.\ \Gamma \vdash_t \{\!|\sigma.\ \mathit{pair\text{-}wf\text{-}digraph}\ (\mathit{mk\text{-}graph}'\ G)|\!\}$
$'R :== \mathit{PROC\ check\text{-}kuratowski}(G,'K)$
$\{\!|{}'G = {}^{\sigma}G \wedge {}'K = {}^{\sigma}K \wedge {}'R \longleftrightarrow \mathit{witness}\ (\mathit{mk\text{-}graph}'\ G)\ (\mathit{mk\text{-}graph}'\ K)|\!\}$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *check-kuratowski-correct-comb*:
assumes *pair-pseudo-graph G*
assumes *witness G K*
shows ¬*comb-planar G*
⟨*proof*⟩

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* ∨ K_5 *H*
shows *witness G K*
⟨*proof*⟩

end
theory *AutoCorres-Misc* **imports**
../l4v/lib/OptionMonadWP
begin

16 Auxilliary Lemmas for Autocorres

16.1 Option monad

definition *owhile-inv* :: ('*a* ⇒ '*s* ⇒ *bool*) ⇒ ('*a* ⇒ ('*s*,'*a*) *lookup*) ⇒ '*a* ⇒ ('*a* ⇒ '*s* ⇒ *bool*) ⇒ '*a rel* ⇒ ('*s*,'*a*) *lookup* **where**
owhile-inv c b a I R ≡ *owhile c b a*

lemma *owhile-unfold*: *owhile C B r s* = *ocondition* (*C r*) (*B r* |>> *owhile C B*) (*oreturn r*) *s*
⟨*proof*⟩

lemma *ovalidNF-owhile*:
assumes ⋀*s*. *P r s* ⟹ *I r s*
and ⋀*r s*. *ovalidNF* (λ*s'*. *I r s'* ∧ *C r s'* ∧ *s'* = *s*) (*B r*) (λ*r' s'*. *I r' s'* ∧ (*r'*, *r*) ∈ *R*)
and *wf R*
and ⋀*r s*. *I r s* ⟹ ¬ *C r s* ⟹ *Q r s*
shows *ovalidNF* (*P r*) (*OptionMonad.owhile C B r*) *Q*
⟨*proof*⟩

lemma *ovalidNF-owhile-inv*[*wp*]:
assumes ⋀*r s*. *ovalidNF* (λ*s'*. *I r s'* ∧ *C r s'* ∧ *s'* = *s*) (*B r*) (λ*r' s'*. *I r' s'* ∧ (*r'*, *r*) ∈ *R*)
and *wf R*
and ⋀*r s*. *I r s* ⟹ ¬ *C r s* ⟹ *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

17 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*)

17.1 Labeled VCG theorems for branching

definition *BRANCH P* ≡ *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′* ≡ λ*pos name*. (*name*, *pos*,[]) # *ct*
assumes *a* ⟹ *C*⟨*Suc inp*,*ct′ inp ″then″*, *outp′*: *b*⟩
assumes ¬*a* ⟹ *C*⟨*Suc outp′*,*ct′ outp′ ″else″*, *outp*: *c*⟩
shows *C*⟨*inp*,*ct*,*outp*: *BRANCH* (*if a then b else c*)⟩
⟨*proof*⟩

lemma *DC-final*:
assumes *V*⟨(*″g″*,*inp*,[]), *ct*: *a*⟩
shows *C*⟨*inp*,*ct*,*Suc inp*: *a*⟩
⟨*proof*⟩

end

⟨*ML*⟩

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

17.2 Labelled VCG theorems for the option monad

definition
lpred-conj :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) (**infixr** *land 35*)
where
lpred-conj P Q ≡ λ*x*. *P x* ∧ *Q x*

context begin
interpretation *Labeling-Syntax* ⟨*proof*⟩

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* (λ-. *R*))
shows *CTXT IC CT OC* (*ovalidNF P* (*f* |>> *K-bind g*) *Q*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *L-ovalidNF-obind*[*vcg-l*]:
assumes ⋀*r*. *CTXT* (*Suc OC1*) ((*''bind''*, *Suc OC1*, [*VAR r*]) # *CT*) *OC*
(*ovalidNF* (*R r*) (*g r*) *Q*)
and *CTXT IC CT OC1* (*ovalidNF P f R*)
shows *CTXT IC CT OC* (*ovalidNF P* (*f* |>> (λ*r*. *g r*)) *Q*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *L-ovalidNF-prod-case*[*vcg-l*]:
assumes ⋀*x y*. *SPLIT v* (*x*,*y*) ⟹ *CTXT IC CT OC* (*ovalidNF* (*P x y*) (*B x y*) *Q*)
shows *CTXT IC CT OC* (*ovalidNF* (*case v of* (*x*, *y*) ⇒ *P x y*) (*case v of* (*x*, *y*) ⇒ *B x y*) *Q*)
⟨*proof*⟩

lemma *L-ovalidNF-oreturn-NF*[*vcg-l*]:
shows *CTXT IC CT IC* (*ovalidNF* (*P x*) (*oreturn x*) *P*)
⟨*proof*⟩

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

lemma *L-ovalidNF-wp-comb2*[*vcg-l-comb*]:
assumes $CTXT\ IC\ CT\ OC\ (ovalidNF\ P\ f\ Q)$
and $\bigwedge s.\ P'\ s \Longrightarrow VC\ (''weaken'', IC, [VAR\ s])\ CT\ (P\ s)$
shows $CTXT\ IC\ CT\ OC\ (ovalidNF\ P'\ f\ Q)$
⟨*proof*⟩

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

lemma *L-ogets-NF-wp*[*vcg-l*]: $CTXT\ IC\ CT\ IC\ (ovalidNF\ (\lambda s.\ P\ (f\ s)\ s)\ (ogets$
$f)\ P)$
⟨*proof*⟩

lemma *elim-land*[*vcg-elim*]:
assumes $(P\ land\ Q)\ s$ **obtains** $P\ s\ Q\ s$
⟨*proof*⟩

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

18 Verififcation of a Planarity Checker

theory *Check-Planarity-Verification*
imports
../Planarity/Graph-Genus
Setup-AutoCorres
HOL−Library.Rewrite
begin

18.1 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*⟩

18.2 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 ≡
  DO n ← oreturn (length (ig-edges iG));
     m ← oreturn (length (ig-verts iG));
     nz ← isolated-nodes iG;
     fc ← face-cycles iG iM;
     oreturn ((int n div 2 + 2 * int c − int m − int nz − int fc) div 2)
  OD

definition certify iG iM c ≡
 DO
   map ← is-map iG iM;
```

ocondition (λ-. *map*)
(*DO*
gen ← *euler-genus iG* (*mk-map* (*mk-graph iG*) *iM*) *c*;
oreturn (*gen* = *0*)
OD)
(*oreturn False*)
OD

18.3 Verification

context begin
interpretation *Labeling-Syntax* ⟨*proof*⟩
lemma *trivial-label*: *P* ⟹ *CTXT IC CT OC P*
⟨*proof*⟩
end

lemma *ovalidNF-wp*:
assumes *ovalidNF P c* (λ*r s*. *r* = *x*)
shows *ovalidNF* (λ*s*. *Q x s* ∧ *P s*) *c Q*
⟨*proof*⟩

18.3.1 *is-map*

definition *is-map-rev-ok-inv iG iM k ok* ≡ *ok* ⟷ (∀ *i* < *k*.
im-rev iM i < *length* (*ig-edges iG*)
∧ *ig-edges iG* ! *im-rev iM i* = *prod.swap* (*ig-edges iG* ! *i*)
∧ *im-rev iM i* ≠ *i*
∧ *im-rev iM* (*im-rev iM i*) = *i*)

definition *is-map-succ-perm-inv iG iM k ok* ≡ *ok* ⟷ (∀ *i* < *k*.
im-succ iM i < *length* (*ig-edges iG*)
∧ *ig-tail iG* (*im-succ iM i*) = *ig-tail iG i*
∧ *im-pred iM* (*im-succ iM i*) = *i*)

definition *is-map-succ-orbits-inv iG iM k ok V A* ≡
A = (⋃ *i*<(*if ok then k else k* − *1*). *orbit* (*im-succ iM*) *i*) ∧
V = {*ig-tail iG i* | *i*. *i*<(*if ok then k else k* − *1*)} ∧
ok = (∀ *i* < *k*. ∀ *j* < *k*. *ig-tail iG i* = *ig-tail iG j* ⟶ *j* ∈ *orbit* (*im-succ iM*) *i*)

definition *is-map-succ-orbits-inner-inv iG iM i j A′* ≡
A′ = (*if i* = *j* ∧ *i* ∉ *A′ then* {} *else* {*i*} ∪ *segment* (*im-succ iM*) *i j*)
∧ *j* ∈ *orbit* (*im-succ iM*) *i*

definition *is-map-final iG k ok* ≡ (*ok* ⟶ *k* = *length* (*ig-edges iG*)) ∧ *k* ≤ *length* (*ig-edges iG*)

lemma *bij-betwI-finite-dom*:
assumes *finite A f* ∈ *A* → *A* ⋀*a*. *a* ∈ *A* ⟹ *g* (*f a*) = *a*
shows *bij-betw f A A*
⟨*proof*⟩

lemma *permutesI-finite-dom*:
assumes *finite A*
assumes *f* ∈ *A* → *A*
assumes ⋀*a*. *a* ∉ *A* ⟹ *f a* = *a*
assumes ⋀*a*. *a* ∈ *A* ⟹ *g* (*f a*) = *a*
shows *f permutes A*
⟨*proof*⟩

lemma *orbit-ss*:
assumes *f* ∈ *A* → *A a* ∈ *A*
shows *orbit f a* ⊆ *A*
⟨*proof*⟩

lemma *segment-eq-orbit*:
assumes *y* ∉ *orbit f x* **shows** *segment f x y* = *orbit f x*
⟨*proof*⟩

lemma *funpow-in-funcset*:
assumes *x* ∈ *A f* ∈ *A* → *A* **shows** (*f* ^^ *n*) *x* ∈ *A*
⟨*proof*⟩

lemma *funpow-eq-funcset*:
assumes *x* ∈ *A f* ∈ *A* → *A* ⋀*y*. *y* ∈ *A* ⟹ *f y* = *g y*
shows (*f* ^^ *n*) *x* = (*g* ^^ *n*) *x*
⟨*proof*⟩

lemma *funpow-dist1-eq-funcset*:
assumes *y* ∈ *orbit f x x* ∈ *A f* ∈ *A* → *A* ⋀*y*. *y* ∈ *A* ⟹ *f y* = *g y*
shows *funpow-dist1 f x y* = *funpow-dist1 g x y*
⟨*proof*⟩

lemma *segment-cong0*:
assumes *x* ∈ *A f* ∈ *A* → *A* ⋀*y*. *y* ∈ *A* ⟹ *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* ⟷ *bidirected-digraph* (*mk-graph iG*) (*edge-rev* (*mk-map* (*mk-graph iG*) *iM*)) (**is** *?L* ⟷ *?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 $\Longrightarrow$ *i* $<$ *length* (*ig-edges iG*) $\Longrightarrow$ *ig-tail iG* (*im-succ iM i*) = *ig-tail iG i*
⟨*proof*⟩

lemma *succ-ok-imp-pred*:
succ-ok $\Longrightarrow$ *i* $<$ *length* (*ig-edges iG*) $\Longrightarrow$ *im-pred iM* (*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* $\Longrightarrow$ *edge-succ* (*mk-map* (*mk-graph iG*) *iM*) $\in$ *arcs* (*mk-graph iG*) $\rightarrow$ *arcs* (*mk-graph iG*)
⟨*proof*⟩

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

lemma *orbit-es-eq-im*:
succ-ok $\Longrightarrow$ *a* $\in$ *arcs* (*mk-graph iG*) $\Longrightarrow$ *orbit* (*edge-succ* (*mk-map* (*mk-graph iG*) *iM*)) *a* = *orbit* (*im-succ iM*) *a*
⟨*proof*⟩

lemma *segment-es-eq-im*:
succ-ok $\Longrightarrow$ *a* $\in$ *arcs* (*mk-graph iG*) $\Longrightarrow$ *segment* (*edge-succ* (*mk-map* (*mk-graph iG*) *iM*)) *a b* = *segment* (*im-succ iM*) *a b*
⟨*proof*⟩

lemma *in-orbit-im-succE*:
assumes *j* $\in$ *orbit* (*im-succ iM*) *i succ-ok i* $<$ *length* (*ig-edges iG*)
obtains *ig-tail iG j* = *ig-tail iG i j* $<$ *length* (*ig-edges iG*)
⟨*proof*⟩

lemma *self-in-orbit-im-succ*:
assumes *succ-ok i* $<$ *length* (*ig-edges iG*) **shows** *i* $\in$ *orbit* (*im-succ iM*) *i*
⟨*proof*⟩

end

locale *is-map-postcondition = is-map-postcondition0 +*
fixes *so-i so-ok V A*
assumes *rev*: *rev-ok* $\longleftrightarrow$ *bidirected-digraph* (*mk-graph iG*) (*edge-rev* (*mk-map* (*mk-graph iG*) *iM*))
assumes *succ-orbits*: *is-map-succ-orbits-inv iG iM so-i so-ok V A succ-ok* $\longrightarrow$ *is-map-final iG so-i so-ok*
begin

lemma *ok-imp-digraph*:
assumes *rev-ok succ-ok so-ok*
shows *digraph-map* (*mk-graph iG*) (*mk-map* (*mk-graph iG*) *iM*)
⟨*proof*⟩

lemma *digraph-imp-ok*:
assumes *dm*: *digraph-map* (*mk-graph iG*) (*mk-map* (*mk-graph iG*) *iM*)
assumes *pred*: $\bigwedge i$. $i <$ *length* (*ig-edges iG*) $\Longrightarrow$ *im-pred iM* (*im-succ iM i*) $= i$
obtains *rev-ok succ-ok so-ok*
⟨*proof*⟩

end

lemma *all-less-Suc-eq*: $(\forall\, x < Suc\ n.\ P\ x) \longleftrightarrow (\forall\, x < n.\ P\ x) \wedge P\ n$
⟨*proof*⟩

lemma *in-orbit-imp-in-segment*:
assumes $y \in$ *orbit f x* $x \neq y$ *bij f* **shows** $y \in$ *segment f x* (*f y*)
⟨*proof*⟩

lemma *ovalidNF-is-map*:
ovalidNF (λs. *distinct* (*ig-verts iG*) $\wedge$ *wf-digraph* (*mk-graph iG*))
(*is-map iG iM*)
($\lambda r\ s$. $r \longleftrightarrow$ *digraph-map* (*mk-graph iG*) (*mk-map* (*mk-graph iG*) *iM*) $\wedge$ ($\forall\, i <$ *length* (*ig-edges iG*). *im-pred iM* (*im-succ iM i*) $= i$))

⟨*proof*⟩

declare *ovalidNF-is-map*[*THEN ovalidNF-wp*, *THEN trivial-label*, *vcg-l*]

18.3.2 *isolated-nodes*

definition *inv-isolated-nodes s iG vcnt ecnt* $\equiv$
vcnt $=$ *length* (*ig-verts iG*)
$\wedge$ *ecnt* $=$ *length* (*ig-edges iG*)
$\wedge$ *distinct* (*ig-verts iG*)
$\wedge$ *sym-digraph* (*mk-graph iG*)

definition *inv-isolated-nodes-outer iG i nz* $\equiv$

nz = *card* (*pre-digraph.isolated-verts* (*mk-graph iG*) ∩ *set* (*take i* (*ig-verts iG*)))

definition *inv-isolated-nodes-inner iG v j* ≡
∀ *k* < *j*. *v* ≠ *ig-tail iG k* ∧ *v* ≠ *ig-head iG k*

lemma (**in** *sym-digraph*) *in-arcs-empty-iff*:
in-arcs G v = {} ⟷ *out-arcs G v* = {}
⟨*proof*⟩

lemma *take-nth-distinct*:
⟦*distinct xs*; *n* < *length xs*; *xs* ! *n* ∈ *set* (*take n xs*)⟧ ⟹ *False*
⟨*proof*⟩

lemma *ovalidNF-isolated-nodes*:
ovalidNF (λ*s*. *distinct* (*ig-verts iG*) ∧ *sym-digraph* (*mk-graph iG*))
(*isolated-nodes iG*)
(λ*r s*. *r* = (*card* (*pre-digraph.isolated-verts* (*mk-graph iG*))))
⟨*proof*⟩

declare *ovalidNF-isolated-nodes*[*THEN ovalidNF-wp*, *THEN trivial-label*, *vcg-l*]

18.3.3 *face-cycles*

definition *inv-face-cycles s iG iM ecnt* ≡
ecnt = *length* (*ig-edges iG*)
∧ *digraph-map* (*mk-graph iG*) *iM*

definition *fcs-upto* :: *nat pre-map* ⇒ *nat* ⇒ *nat set set* **where**
fcs-upto iM i ≡ {*pre-digraph-map.face-cycle-set iM k* | *k*. *k* < *i*}

definition *inv-face-cycles-outer s iG iM i c edge-info* ≡
let fcs = *fcs-upto iM i in*
c = *card fcs*
∧ (∀ *k* < *length* (*ig-edges iG*). *k* ∈ *edge-info* ⟷ *k* ∈ ⋃*fcs*)

definition *inv-face-cycles-inner s iG iM i j c edge-info* ≡
j ∈ *pre-digraph-map.face-cycle-set iM i*
∧ *c* = *card* (*fcs-upto iM i*)
∧ *i* ∉ ⋃(*fcs-upto iM i*)
∧ (∀ *k* < *length* (*ig-edges iG*). *k* ∈ *edge-info* ⟷
(*k* ∈ ⋃(*fcs-upto iM i*)
∨ (∃ *l* < *funpow-dist1* (*pre-digraph-map.face-cycle-succ iM*) *i j*. (*pre-digraph-map.face-cycle-succ iM* ^^ *l*) *i* = *k*)))

lemma *finite-fcs-upto*: *finite* (*fcs-upto iM i*)
⟨*proof*⟩

lemma *card-orbit-eq-funpow-dist1*:

assumes $x \in$ *orbit f x* **shows** *card* (*orbit f x*) = *funpow-dist1 f x x*
⟨*proof*⟩

lemma *funpow-dist1-le*:
assumes $y \in$ *orbit f x x* $\in$ *orbit f x*
shows *funpow-dist1 f x y* $\leq$ *funpow-dist1 f x x*
⟨*proof*⟩

lemma *funpow-dist1-le-card*:
assumes $y \in$ *orbit f x x* $\in$ *orbit f x*
shows *funpow-dist1 f x y* $\leq$ *card* (*orbit f x*)
⟨*proof*⟩

lemma (**in** *digraph-map*) *funpow-dist1-le-card-fcs*:
assumes $b \in$ *face-cycle-set a*
shows *funpow-dist1 face-cycle-succ a b* $\leq$ *card* (*face-cycle-set a*)
⟨*proof*⟩

lemma *funpow-dist1-f-eq*:
assumes $b \in$ *orbit f a a* $\in$ *orbit f a a* $\neq b$
shows *funpow-dist1 f a* (*f b*) = *Suc* (*funpow-dist1 f a b*)
⟨*proof*⟩

lemma (**in** −) *funpow-dist1-less-f*:
assumes $b \in$ *orbit f a a* $\in$ *orbit f a a* $\neq b$
shows *funpow-dist1 f a b* < *funpow-dist1 f a* (*f b*)
⟨*proof*⟩

lemma *ovalidNF-face-cycles*:
ovalidNF (λ*s. digraph-map* (*mk-graph iG*) *iM*)
(*face-cycles iG iM*)
(λ*r s. r* = *card* (*pre-digraph-map.face-cycle-sets* (*mk-graph iG*) *iM*))

⟨*proof*⟩
declare *ovalidNF-face-cycles*[*THEN ovalidNF-wp*, *THEN trivial-label*, *vcg-l*]

lemma *ovalidNF-euler-genus*:
ovalidNF (λ*s. distinct* (*ig-verts iG*) ∧ *digraph-map* (*mk-graph iG*) *iM* ∧ *c* = *card* (*pre-digraph.sccs* (*mk-graph iG*)))
(*euler-genus iG iM c*)
(λ*r s. r* = *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 (λ*s. distinct* (*ig-verts iG*) ∧ *fin-digraph* (*mk-graph iG*) ∧ *c* = *card* (*pre-digraph.sccs* (*mk-graph iG*)))

(*certify iG iM c*)
(λ*r s*. *r* ⟷ *pre-digraph-map.euler-genus* (*mk-graph iG*) (*mk-map* (*mk-graph iG*) *iM*) = *0*
∧ *digraph-map* (*mk-graph iG*) (*mk-map* (*mk-graph iG*) *iM*)
∧ (∀ *i* < *length* (*ig-edges iG*). *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.