

Tame Plane Graphs

Gertrud Bauer and Tobias Nipkow

March 17, 2025

Abstract

These theories present the verified enumeration of *tame* plane graphs as defined by Thomas C. Hales in his revised proof of the Kepler Conjecture. Compared with his original proof, the notion of tameness has become simpler, there are many more tame graphs, but much of the earlier verification [1] carries over. For more details see <http://code.google.com/p/flyspeck/> and the forthcoming book “Dense Sphere Packings: A Blueprint for Formal Proofs” by Hales.

Contents

1	Basic Functions Old and New	5
1.1	HOL	5
1.2	Lists	5
1.3	<i>splitAt</i>	16
1.4	<i>between</i>	26
1.5	Tables	27
2	Isomorphisms Between Plane Graphs	30
2.1	Equivalence of faces	30
2.2	Homomorphism and isomorphism	32
2.3	Isomorphism tests	37
2.4	Elementhood and containment modulo	47
3	More Rotation	48
4	Graph	49
4.1	Notation	50
4.2	Faces	50
4.3	Graphs	51
4.4	Operations on graphs	52
4.5	Navigation in graphs	54
4.6	Code generator setup	54

5	Syntax for operations on immutable arrays	55
5.1	Tabulation	55
5.2	Access	56
6	Enumerating Patches	56
7	Subdividing a Face	57
8	Transitive Closure of Successor List Function	59
9	Plane Graph Enumeration	60
10	Properties of Graph Utilities	62
10.1	<i>nextElem</i>	63
10.2	<i>nextVertex</i>	65
10.3	\mathcal{E}	66
10.4	Triangles	67
10.5	Quadrilaterals	67
10.6	No loops	68
10.7	<i>between</i>	69
11	Properties of Patch Enumeration	71
12	Properties of Face Division	80
12.1	Finality	81
12.2	<i>is-prefix</i>	82
12.3	<i>is-sublist</i>	83
12.4	<i>is-nextElem</i>	89
12.5	<i>nextElem</i> , <i>sublist</i> , <i>is-nextElem</i>	92
12.6	<i>before</i>	94
12.7	<i>between</i>	98
12.8	<i>split-face</i>	112
12.9	<i>verticesFrom</i>	115
12.10	<i>splitFace</i>	121
12.11	<i>removeNones</i>	159
12.12	<i>natToVertexList</i>	159
12.13	<i>indexToVertexList</i>	160
12.14	<i>pre-subdivFace</i> ([!])	170
13	Invariants of (Plane) Graphs	194
13.1	Rotation of face into normal form	194
13.2	Minimal (plane) graph properties	194
13.3	<i>containsDuplicateEdge</i>	201
13.4	<i>replacefacesAt</i>	202
13.5	<i>normFace</i>	206

13.6	Invariants of <i>splitFace</i>	209
13.7	Invariants of <i>makeFaceFinal</i>	236
13.8	Invariants of <i>subdivFace'</i>	239
13.9	Invariants of <i>Seed</i>	248
13.10	Increasing properties of <i>subdivFace'</i>	251
13.11	Main invariant theorems	254
14	Further Plane Graph Properties	255
14.1	<i>final</i>	255
14.2	<i>degree</i>	256
14.3	Misc	257
14.4	Increasing final faces	258
14.5	Increasing vertices	259
14.6	Increasing vertex degrees	259
14.7	Increasing <i>except</i>	260
14.8	Increasing edges	260
14.9	Increasing final vertices	260
14.10	Preservation of <i>facesAt</i> at final vertices	261
14.11	Properties of <i>subdivFace'</i>	262
15	Summation Over Lists	269
16	Tameness	271
16.1	Constants	271
16.2	Separated sets of vertices	271
16.3	Admissible weight assignments	272
16.4	Tameness	272
17	Enumeration of Tame Plane Graphs	273
18	Tame Properties	276
19	Neglectable Final Graphs	278
20	Properties of Lower Bound Machinery	278
21	Correctness of Lower Bound for Final Graphs	293
22	Properties of Tame Graph Enumeration (1)	294
23	Properties of Tame Graph Enumeration (2)	305
24	Archive	314
25	Comparing Enumeration and Archive	315

26 Completeness of Archive Test	316
27 Completeness Proofs under hypothetical computations	318
Bibliography	321

1 Basic Functions Old and New

```
theory ListAux
imports Main
begin
```

```
declare Let-def[simp]
```

1.1 HOL

```
lemma pairD:  $(a,b) = p \implies a = \text{fst } p \wedge b = \text{snd } p$ 
by auto
```

```
lemmas conj-aci = conj-comms conj-assoc conj-absorb conj-left-absorb
```

```
definition enum :: nat  $\Rightarrow$  nat set where
[code-abbrev]: enum n = {.. $n$ }
```

```
lemma [code]:
enum 0 = {}
enum (Suc n) = insert n (enum n)
unfolding enum-def lessThan-0 lessThan-Suc by rule+
```

1.2 Lists

```
declare List.member-def[simp] list-all-iff[simp] list-ex-iff[simp]
```

1.2.1 length

```
notation length ( $\langle | \cdot \rangle$ )
```

```
lemma length3D:  $|xs| = 3 \implies \exists x y z. xs = [x, y, z]$ 
apply (cases xs) apply simp
apply (case-tac list) apply simp
apply (case-tac lista) by simp-all
```

```
lemma length4D:  $|xs| = 4 \implies \exists a b c d. xs = [a, b, c, d]$ 
apply (case-tac xs) apply simp
apply (case-tac list) apply simp
apply (case-tac lista) apply simp
apply (case-tac listb) by simp-all
```

1.2.2 filter

```
lemma filter-emptyE[dest]:  $(\text{filter } P \text{ } xs = []) \implies x \in \text{set } xs \implies \neg P x$ 
by (simp add: filter-empty-conv)
```

```
lemma filter-comm:  $[x \leftarrow xs. P x \wedge Q x] = [x \leftarrow xs. Q x \wedge P x]$ 
by (simp add: conj-aci)
```

lemma *filter-prop*: $x \in \text{set } [u \leftarrow ys . P u] \implies P x$
proof (*induct ys arbitrary: x*)
 case Nil then show ?case by simp
next
 case Cons then show ?case by (auto split: if-split-asm)
qed

lemma *filter-compl1*:
 $([x \leftarrow xs . P x] = []) = ([x \leftarrow xs . \neg P x] = xs)$ (**is** *?lhs = ?rhs*)
proof
 show ?rhs \implies ?lhs
 proof (*induct xs*)
 case Nil then show ?case by simp
 next
 case (Cons x xs)
 have $[u \leftarrow xs . \neg P u] \neq x \# xs$
 proof
 assume $[u \leftarrow xs . \neg P u] = x \# xs$
 then have $|x \# xs| = |[u \leftarrow xs . \neg P u]|$ **by simp**
 also have $\dots \leq |xs|$ **by simp**
 finally show False by simp
 qed
 with Cons show ?case by auto
 qed
next
 show ?lhs \implies ?rhs
 by (induct xs) (simp-all split: if-split-asm)
qed
lemma [*simp*]: $\text{Not} \circ (\text{Not} \circ P) = P$
 by (rule ext) simp

lemma *filter-eq1*:
 $(\bigwedge v. v \in \text{set } vs \implies P v = Q v) \implies [v \leftarrow vs . P v] = [v \leftarrow vs . Q v]$
 by (induct vs) simp-all

lemma *filter-simp*: $(\bigwedge x. x \in \text{set } xs \implies P x) \implies [x \leftarrow xs . P x \wedge Q x] = [x \leftarrow xs . Q x]$
 by (induct xs) auto

lemma *filter-True-eq1*:
 $(\text{length } [y \leftarrow xs . P y] = \text{length } xs) \implies (\bigwedge y. y \in \text{set } xs \implies P y)$
proof (*induct xs*)
 case Nil then show ?case by simp
next
 case (Cons x xs)
 have $l: \text{length } (\text{filter } P xs) \leq \text{length } xs$
 by (simp add: length-filter-le)
 have *hyp*: $\text{length } (\text{filter } P (x \# xs)) = \text{length } (x \# xs)$ **by fact**

then have $P x$ **by** (*simp split: if-split-asm*) (*insert l, arith*)
moreover with hyp have $\text{length} (\text{filter } P \text{ } xs) = \text{length } xs$
by (*simp split: if-split-asm*)
moreover have $y \in \text{set } (x\#xs)$ **by fact**
ultimately show $?case$ **by** (*auto dest: Cons(1)*)
qed

lemma [*simp*]: $[f x. x <- xs, P x] = [f x. x <- [x \leftarrow xs. P x]]$
by (*induct xs*) *auto*

1.2.3 concat

syntax

-concat :: $idt \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \ (\langle \lfloor _ \in _ \rightarrow 10 \rangle)$

syntax-consts

-concat == *concat*

translations

$\lfloor_{x \in xs} f == \text{CONST } \text{concat } [f. x <- xs]$

1.2.4 List product

definition *listProd1* :: $'a \Rightarrow 'b \text{ list} \Rightarrow ('a \times 'b) \text{ list}$ **where**
listProd1 $a \text{ } bs \equiv [(a,b). b <- bs]$

definition *listProd* :: $'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow ('a \times 'b) \text{ list}$ (*infix* $\langle \times \rangle$ 50) **where**
 $as \times bs \equiv \lfloor_a \in as \text{ } listProd1 \text{ } a \text{ } bs$

lemma $\text{set } (xs \times ys) = (\text{set } xs) \times (\text{set } ys)$
by (*auto simp: listProd-def listProd1-def*)

1.2.5 Minimum and maximum

primrec *minimal*:: $('a \Rightarrow \text{nat}) \Rightarrow 'a \text{ list} \Rightarrow 'a$ **where**
minimal $m \text{ } (x\#xs) =$
(if $xs = []$ *then* x *else*
let $mxs = \text{minimal } m \text{ } xs$ *in*
if $m \ x \leq m \ mxs$ *then* x *else* mxs)

lemma *minimal-in-set*[*simp*]: $xs \neq [] \Longrightarrow \text{minimal } f \text{ } xs : \text{set } xs$
by(*induct xs*) *auto*

primrec *min-list* :: $\text{nat list} \Rightarrow \text{nat}$ **where**
min-list $(x\#xs) = (\text{if } xs = [] \text{ then } x \text{ else } \text{min } x \text{ } (\text{min-list } xs))$

primrec *max-list* :: $\text{nat list} \Rightarrow \text{nat}$ **where**
max-list $(x\#xs) = (\text{if } xs = [] \text{ then } x \text{ else } \text{max } x \text{ } (\text{max-list } xs))$

lemma *min-list-conv-Min*[*simp*]:

$xs \neq [] \implies \text{min-list } xs = \text{Min } (\text{set } xs)$
by *(induct xs) auto*

lemma *max-list-conv-Max*[simp]:
 $xs \neq [] \implies \text{max-list } xs = \text{Max } (\text{set } xs)$
by *(induct xs) auto*

1.2.6 replace

primrec *replace* :: 'a \Rightarrow 'a list \Rightarrow 'a list \Rightarrow 'a list **where**
 $\text{replace } x \text{ ys } [] = []$
| $\text{replace } x \text{ ys } (z\#zs) =$
 $(\text{if } z = x \text{ then ys } @ \text{ zs else } z \# (\text{replace } x \text{ ys } zs))$

primrec *mapAt* :: nat list \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a list \Rightarrow 'a list) **where**
 $\text{mapAt } [] \text{ f as} = \text{as}$
| $\text{mapAt } (n\#ns) \text{ f as} =$
 $(\text{if } n < |\text{as}| \text{ then } \text{mapAt } ns \text{ f } (\text{as}[n:= \text{f } (\text{as}!n)])$
 $\text{else } \text{mapAt } ns \text{ f as})$

lemma *length-mapAt*[simp]: $\bigwedge xs. \text{length}(\text{mapAt } vs \text{ f } xs) = \text{length } xs$
by*(induct vs) auto*

lemma *length-replace1*[simp]: $\text{length}(\text{replace } x [y] xs) = \text{length } xs$
by*(induct xs) simp-all*

lemma *replace-id*[simp]: $\text{replace } x [x] xs = xs$
by*(induct xs) simp-all*

lemma *len-replace-ge-same*:
 $\text{length } ys \geq 1 \implies \text{length}(\text{replace } x \text{ ys } xs) \geq \text{length } xs$
by *(induct xs) auto*

lemma *len-replace-ge*[simp]:
 $\llbracket \text{length } ys \geq 1; \text{length } xs \geq \text{length } zs \rrbracket \implies$
 $\text{length}(\text{replace } x \text{ ys } xs) \geq \text{length } zs$
apply*(drule len-replace-ge-same[where x = x and xs = xs])*
apply *arith*
done

lemma *replace-append*[simp]:
 $\text{replace } x \text{ ys } (\text{as } @ \text{ bs}) =$
 $(\text{if } x \in \text{set as} \text{ then } \text{replace } x \text{ ys as } @ \text{ bs else as } @ \text{ replace } x \text{ ys bs})$
by*(induct as) auto*

lemma *distinct-set-replace*: $\text{distinct } xs \implies$
 $\text{set } (\text{replace } x \text{ ys } xs) =$

$(if\ x \in\ set\ xs\ then\ (set\ xs - \{x\}) \cup\ set\ ys\ else\ set\ xs)$
apply(*induct xs*)
apply(*simp*)
apply *simp*
apply *blast*
done

lemma *replace1*:
 $f \in set\ (replace\ f'\ fs\ ls) \implies f \notin set\ ls \implies f \in set\ fs$
proof (*induct ls*)
case *Nil* **then show** ?*case* **by** *simp*
next
case (*Cons l ls*) **then show** ?*case* **by** (*simp split: if-split-asm*)
qed

lemma *replace2*:
 $f' \notin set\ ls \implies replace\ f'\ fs\ ls = ls$
proof (*induct ls*)
case *Nil* **then show** ?*case* **by** *simp*
next
case (*Cons l ls*) **then show** ?*case* **by** (*auto split: if-split-asm*)
qed

lemma *replace3[intro]*:
 $f' \in set\ ls \implies f \in set\ fs \implies f \in set\ (replace\ f'\ fs\ ls)$
by (*induct ls*) *auto*

lemma *replace4*:
 $f \in set\ ls \implies oldF \neq f \implies f \in set\ (replace\ oldF\ fs\ ls)$
by (*induct ls*) *auto*

lemma *replace5*: $f \in set\ (replace\ oldF\ newfs\ fs) \implies f \in set\ fs \vee f \in set\ newfs$
by (*induct fs*) (*auto split: if-split-asm*)

lemma *replace6*: $distinct\ oldfs \implies x \in set\ (replace\ oldF\ newfs\ oldfs) =$
 $((x \neq oldF \vee oldF \in set\ newfs) \wedge ((oldF \in set\ oldfs \wedge x \in set\ newfs) \vee x \in set\ oldfs))$
by (*induct oldfs*) *auto*

lemma *distinct-replace*:
 $distinct\ fs \implies distinct\ newFs \implies set\ fs \cap set\ newFs \subseteq \{oldF\} \implies$
 $distinct\ (replace\ oldF\ newFs\ fs)$
proof (*induct fs*)
case *Nil* **then show** ?*case* **by** *simp*
next
case (*Cons f fs*)
then show ?*case*
proof (*cases f = oldF*)

```

    case True with Cons show ?thesis by simp blast
next
case False
moreover with Cons have f  $\notin$  set newFs by simp blast
with Cons have f  $\notin$  set (replace oldF newFs fs)
  by simp (blast dest: replace1)
moreover from Cons have distinct (replace oldF newFs fs)
  by (rule-tac Cons) auto
ultimately show ?thesis by simp
qed
qed

```

```

lemma replace-replace[simp]: oldf  $\notin$  set newfs  $\implies$  distinct xs  $\implies$ 
  replace oldf newfs (replace oldf newfs xs) = replace oldf newfs xs
apply (induct xs) apply auto apply (rule replace2) by simp

```

```

lemma replace-distinct: distinct fs  $\implies$  distinct newfs  $\implies$  oldf  $\in$  set fs  $\longrightarrow$  set
  newfs  $\cap$  set fs  $\subseteq$  {oldf}  $\implies$ 
  distinct (replace oldf newfs fs)
apply (case-tac oldf  $\in$  set fs) apply simp
apply (induct fs) apply simp
apply (auto simp: replace2) apply (drule replace1)
by auto

```

```

lemma filter-replace2:
   $\llbracket \neg P x; \forall y \in \text{set } ys. \neg P y \rrbracket \implies$ 
  filter P (replace x ys xs) = filter P xs
apply (cases x  $\in$  set xs)
prefer 2 apply (simp add: replace2)
apply (induct xs)
  apply simp
  apply clarsimp
done

```

```

lemma length-filter-replace1:
   $\llbracket x \in \text{set } xs; \neg P x \rrbracket \implies$ 
  length(filter P (replace x ys xs)) =
  length(filter P xs) + length(filter P ys)
apply (induct xs)
  apply simp
  apply fastforce
done

```

```

lemma length-filter-replace2:
   $\llbracket x \in \text{set } xs; P x \rrbracket \implies$ 
  length(filter P (replace x ys xs)) =
  length(filter P xs) + length(filter P ys) - 1
apply (induct xs)

```

```

apply simp
apply auto
apply (drule split-list)
apply clarsimp
done

```

1.2.7 *distinct*

lemma dist-at1: $\bigwedge c\ vs.\ distinct\ vs \implies vs = a @ r \# b \implies vs = c @ r \# d \implies a = c$

```

proof (induct a)
  case Nil
    assume dist: distinct vs and vs1: vs = [] @ r \# b and vs2: vs = c @ r \# d
    from dist vs2 have rc: r \notin set c by auto
    from vs1 vs2 have c @ r \# d = r \# b by auto
    then have hd (c @ r \# d) = r by auto
    then have c \neq [] \implies hd c = r by auto
    then have c \neq [] \implies r \in set c by (induct c) auto
    with rc have c: c = [] by auto
    then show ?case by auto
  next
    case (Cons x xs) then show ?case by (induct c) auto
qed

```

lemma dist-at: $distinct\ vs \implies vs = a @ r \# b \implies vs = c @ r \# d \implies a = c \wedge b = d$

```

proof -
  assume dist: distinct vs and vs1: vs = a @ r \# b and vs2: vs = c @ r \# d
  then have a = c by (rule-tac dist-at1) auto
  with dist vs1 vs2 show ?thesis by simp
qed

```

lemma dist-at2: $distinct\ vs \implies vs = a @ r \# b \implies vs = c @ r \# d \implies b = d$

```

proof -
  assume dist: distinct vs and vs1: vs = a @ r \# b and vs2: vs = c @ r \# d
  then have a = c \wedge b = d by (rule-tac dist-at) auto
  then show ?thesis by auto
qed

```

lemma distinct-split1: $distinct\ xs \implies xs = y @ [r] @ z \implies r \notin set\ y$
apply auto done

lemma distinct-split2: $distinct\ xs \implies xs = y @ [r] @ z \implies r \notin set\ z$ **apply auto done**

lemma distinct-hd-not-cons: $distinct\ vs \implies \exists\ as\ bs.\ vs = as @ x \# hd\ vs \# bs \implies False$

```

proof -
  assume d: distinct vs and ex: \exists as bs. vs = as @ x \# hd vs \# bs

```

```

from ex have vsne: vs ≠ [] by auto
with d ex show ?thesis apply (elim exE) apply (case-tac as)
  apply (subgoal-tac hd vs = x) apply simp apply (rule sym) apply simp
  apply (subgoal-tac x = hd (x # (hd vs # bs))) apply simp apply (thin-tac vs
= x # hd vs # bs)
  apply auto
  apply (subgoal-tac hd vs = a) apply simp
  apply (subgoal-tac a = hd (a # list @ x # hd vs # bs)) apply simp
  apply (thin-tac vs = a # list @ x # hd vs # bs) by auto
qed

```

1.2.8 Misc

lemma *drop-last-in*: $\bigwedge n. n < \text{length } ls \implies \text{last } ls \in \text{set } (\text{drop } n \text{ } ls)$

```

apply (frule-tac last-drop) apply(erule subst)
apply (case-tac drop n ls rule: rev-exhaust) by simp-all

```

lemma *nth-last-Suc-n*: $\text{distinct } ls \implies n < \text{length } ls \implies \text{last } ls = ls ! n \implies \text{Suc } n = \text{length } ls$

proof (*rule ccontr*)

assume *d*: *distinct ls* **and** *n*: $n < \text{length } ls$ **and** *l*: $\text{last } ls = ls ! n$ **and** *not*: $\text{Suc } n \neq \text{length } ls$

then **have** *s*: $\text{Suc } n < \text{length } ls$ **by** *auto*

define *lls* **where** $lls = ls ! n$

with *n* **have** $\text{take } (\text{Suc } n) \text{ } ls = \text{take } n \text{ } ls @ [lls]$ **apply** *simp* **by** (*rule take-Suc-conv-app-nth*)

then **have** $\text{take } (\text{Suc } n) \text{ } ls @ \text{drop } (\text{Suc } n) \text{ } ls = \text{take } n \text{ } ls @ [lls] @ \text{drop } (\text{Suc } n)$

ls **by** *auto*

then **have** *ls*: $ls = \text{take } n \text{ } ls @ [lls] @ \text{drop } (\text{Suc } n) \text{ } ls$ **by** *auto*

with *d* **have** *dls*: $\text{distinct } (\text{take } n \text{ } ls @ [lls] @ \text{drop } (\text{Suc } n) \text{ } ls)$ **by** *auto*

from *lls-def l* **have** $lls = (\text{last } ls)$ **by** *auto*

with *s* **have** $lls \in \text{set } (\text{drop } (\text{Suc } n) \text{ } ls)$ **apply** *simp* **by** (*rule-tac drop-last-in*)

with *dls* **show** *False* **by** *auto*

qed

1.2.9 rotate

lemma *plus-length1[simp]*: $\text{rotate } (k + (\text{length } ls)) \text{ } ls = \text{rotate } k \text{ } ls$

proof –

have $\bigwedge k \text{ } ls. \text{rotate } k \text{ } (\text{rotate } (\text{length } ls) \text{ } ls) = \text{rotate } (k + (\text{length } ls)) \text{ } ls$

by (*rule rotate-rotate*)

then **have** $\bigwedge k \text{ } ls. \text{rotate } k \text{ } ls = \text{rotate } (k + (\text{length } ls)) \text{ } ls$ **by** *auto*

then **show** *?thesis* **by** (*rule sym*)

qed

lemma *plus-length2[simp]*: $\text{rotate } ((\text{length } ls) + k) \text{ } ls = \text{rotate } k \text{ } ls$

proof –

define *x* **where** $x = (\text{length } ls) + k$

then **have** $x = k + (\text{length } ls)$ **by** *auto*

with *x-def* **have** $\text{rotate } x \text{ } ls = \text{rotate } (k + (\text{length } ls)) \text{ } ls$ **by** *simp*

then **have** $\text{rotate } x \text{ } ls = \text{rotate } k \text{ } ls$ **by** *simp*

with *x-def* **show** *?thesis* **by** *simp*
qed

lemma *rotate-minus1*: $n > 0 \implies m > 0 \implies$

$rotate\ n\ ls = rotate\ m\ ms \implies rotate\ (n - 1)\ ls = rotate\ (m - 1)\ ms$

proof (*cases* $ls = []$)

assume *r*: $rotate\ n\ ls = rotate\ m\ ms$

case *True* **with** *r*

have $rotate\ m\ ms = []$ **by** *auto*

then **have** $ms = []$ **by** *auto*

with *True* **show** *?thesis* **by** *auto*

next

assume *n*: $n > 0$ **and** *m*: $m > 0$ **and** *r*: $rotate\ n\ ls = rotate\ m\ ms$

case *False*

then **have** *lls*: $length\ ls > 0$ **by** *auto*

with *r* **have** *lms*: $length\ ms > 0$ **by** *auto*

have *mem1*: $rotate\ (n - 1)\ ls = rotate\ ((n - 1) + length\ ls)\ ls$ **by** *auto*

from *n lls* **have** $(n - 1) + length\ ls = (length\ ls - 1) + n$ **by** *arith*

then **have** $rotate\ ((n - 1) + length\ ls)\ ls = rotate\ ((length\ ls - 1) + n)\ ls$ **by** *auto*

with *mem1* **have** *mem2*: $rotate\ (n - 1)\ ls = rotate\ ((length\ ls - 1) + n)\ ls$ **by** *auto*

have $rotate\ ((length\ ls - 1) + n)\ ls = rotate\ (length\ ls - 1)\ (rotate\ n\ ls)$ **apply** (*rule sym*)

by (*rule rotate-rotate*)

with *mem2* **have** *mem3*: $rotate\ (n - 1)\ ls = rotate\ (length\ ls - 1)\ (rotate\ n\ ls)$ **by** *auto*

from *r* **have** $rotate\ (length\ ls - 1)\ (rotate\ n\ ls) = rotate\ (length\ ls - 1)\ (rotate\ m\ ms)$ **by** *auto*

with *mem3* **have** *mem4*: $rotate\ (n - 1)\ ls = rotate\ (length\ ls - 1)\ (rotate\ m\ ms)$ **by** *auto*

have $rotate\ (length\ ls - 1)\ (rotate\ m\ ms) = rotate\ (length\ ls - 1 + m)\ ms$ **by** (*rule rotate-rotate*)

with *mem4* **have** *mem5*: $rotate\ (n - 1)\ ls = rotate\ (length\ ls - 1 + m)\ ms$ **by** *auto*

from *r* **have** $length\ (rotate\ n\ ls) = length\ (rotate\ m\ ms)$ **by** *simp*

then **have** $length\ ls = length\ ms$ **by** *auto*

with *m lms* **have** $length\ ls - 1 + m = (m - 1) + length\ ms$ **by** *arith*

with *mem5* **have** *mem6*: $rotate\ (n - 1)\ ls = rotate\ ((m - 1) + length\ ms)\ ms$ **by** *auto*

have $rotate\ ((m - 1) + length\ ms)\ ms = rotate\ (m - 1)\ (rotate\ (length\ ms)\ ms)$ **by** *auto*

then **have** $rotate\ ((m - 1) + length\ ms)\ ms = rotate\ (m - 1)\ ms$ **by** *auto*

with *mem6* **show** *?thesis* **by** *auto*

qed

lemma *rotate-minus1'*: $n > 0 \implies rotate\ n\ ls = ms \implies$

$rotate\ (n - 1)\ ls = rotate\ (length\ ms - 1)\ ms$

proof (*cases* $ls = []$)

```

    assume r: rotate n ls = ms
    case True with r show ?thesis by auto
next
    assume n: n > 0 and r: rotate n ls = ms
    then have r': rotate n ls = rotate (length ms) ms by auto
    case False
    with n r' show ?thesis apply (rule-tac rotate-minus1) by auto
qed

lemma rotate-inv1:  $\bigwedge ms. n < \text{length } ls \implies \text{rotate } n \text{ } ls = ms \implies$ 
   $ls = \text{rotate } ((\text{length } ls) - n) \text{ } ms$ 
proof (induct n)
  case 0 then show ?case by auto
next
  case (Suc n) then show ?case
  proof (cases ls = [])
    case True with Suc
      show ?thesis by auto
  next
    case False
      then have ll: length ls > 0 by auto
      from Suc have nl: n < length ls by auto
      from Suc have r: rotate (Suc n) ls = ms by auto
      then have rotate (Suc n - 1) ls = rotate (length ms - 1) ms apply (rule-tac
rotate-minus1') by auto
      then have rotate n ls = rotate (length ms - 1) ms by auto
      then have mem: ls = rotate (length ls - n) (rotate (length ms - 1) ms)
        apply (rule-tac Suc) by (auto simp: nl)
      have rotate (length ls - n) (rotate (length ms - 1) ms) = rotate (length ls -
n + (length ms - 1)) ms
        by (rule rotate-rotate)
      with mem have mem2: ls = rotate (length ls - n + (length ms - 1)) ms by
auto
      from r have leq: length ms = length ls by auto
      with False nl have length ls - n + (length ms - 1) = length ms + (length ms
- (Suc n))
        by arith
      then have rotate (length ls - n + (length ms - 1)) ms = rotate (length ms +
(length ms - (Suc n))) ms
        by auto
      with mem2 have mem3: ls = rotate (length ms + (length ms - (Suc n))) ms
by auto
      have rotate (length ms + (length ms - (Suc n))) ms = rotate (length ms -
(Suc n)) ms by simp
      with mem3 leq show ?thesis by auto
  qed
qed

```

lemma rotate-conv-mod'[simp]: $\text{rotate } (n \bmod \text{length } ls) \text{ } ls = \text{rotate } n \text{ } ls$

by(*simp add:rotate-drop-take*)

lemma rotate-inv2: $rotate\ n\ ls = ms \implies$
 $ls = rotate\ ((length\ ls) - (n\ mod\ length\ ls))\ ms$
proof (*cases* $ls = []$)
 assume $r: rotate\ n\ ls = ms$
 case *True* **with** r **show** *?thesis* **by** *auto*
next
 assume $r: rotate\ n\ ls = ms$
 then **have** $r': rotate\ (n\ mod\ length\ ls)\ ls = ms$ **by** *auto*
 case *False*
 then **have** $length\ ls > 0$ **by** *auto*
 with r' **show** *?thesis* **apply** (*rule-tac rotate-inv1*) **by** *auto*
qed

lemma rotate-id[*simp*]: $rotate\ ((length\ ls) - (n\ mod\ length\ ls))\ (rotate\ n\ ls) = ls$
apply (*rule sym*) **apply** (*rule rotate-inv2*) **by** *simp*

lemma nth-rotate1-Suc: $Suc\ n < length\ ls \implies ls!(Suc\ n) = (rotate1\ ls)!n$
apply (*cases* ls) **apply** *auto*
by (*simp add: nth-append*)

lemma nth-rotate1-0: $ls!0 = (rotate1\ ls)!(length\ ls - 1)$ **apply** (*cases* ls) **by** *auto*

lemma nth-rotate1: $0 < length\ ls \implies ls!((Suc\ n)\ mod\ (length\ ls)) = (rotate1\ ls)!(n\ mod\ (length\ ls))$
proof (*cases* $0 < (Suc\ n)\ mod\ (length\ ls)$)
 assume $lls: 0 < length\ ls$
 case *True*
 define m **where** $m = (Suc\ n)\ mod\ (length\ ls) - 1$
 with *True* **have** $m: Suc\ m = (Suc\ n)\ mod\ (length\ ls)$ **by** *auto*
 with lls **have** $a: (Suc\ m) < length\ ls$ **by** *auto*
 from $lls\ m$ **have** $m = n\ mod\ (length\ ls)$ **by** (*simp add: mod-Suc split:if-split-asm*)
 with $a\ m$ **show** *?thesis* **apply** (*rule-tac nth-rotate1-Suc*) **by** *auto*
 next
 assume $lls: 0 < length\ ls$
 case *False*
 then **have** $a: (Suc\ n)\ mod\ (length\ ls) = 0$ **by** *auto*
 with lls **have** $Suc\ (n\ mod\ (length\ ls)) = (length\ ls)$ **by** (*auto simp: mod-Suc split:if-split-asm*)
 then **have** $(n\ mod\ (length\ ls)) = (length\ ls) - 1$ **by** *arith*
 with a **show** *?thesis* **by** (*auto simp: nth-rotate1-0*)
qed

lemma rotate-Suc2[*simp*]: $rotate\ n\ (rotate1\ xs) = rotate\ (Suc\ n)\ xs$
apply (*simp add:rotate-def*) **apply** (*induct* n) **by** *auto*

lemma nth-rotate: $\bigwedge\ ls.\ 0 < length\ ls \implies ls!((n+m)\ mod\ (length\ ls)) = (rotate\ m\ ls)!(n\ mod\ (length\ ls))$

```

proof (induct m)
  case 0 then show ?case by auto
next
  case (Suc m)
  define z where z = n + m
  then have n + Suc m = Suc (z) by auto
  with Suc have r1: ls ! ((Suc z) mod length ls) = rotate1 ls ! (z mod length ls)
    by (rule-tac nth-rotate1)
  from Suc have 0 < length (rotate1 ls) by auto
  then have (rotate1 ls) ! ((n + m) mod length (rotate1 ls))
    = rotate m (rotate1 ls) ! (n mod length (rotate1 ls)) by (rule Suc)
  with r1 z-def have ls ! ((n + Suc m) mod length ls)
    = rotate m (rotate1 ls) ! (n mod length (rotate1 ls)) by auto
  then show ?case by auto
qed

```

1.3 splitAt

```

primrec splitAtRec :: 'a ⇒ 'a list ⇒ 'a list ⇒ 'a list × 'a list where
  splitAtRec c bs [] = (bs, [])
| splitAtRec c bs (a#as) = (if a = c then (bs, as)
                           else splitAtRec c (bs@[a]) as)

```

```

definition splitAt :: 'a ⇒ 'a list ⇒ 'a list × 'a list where
  splitAt c as ≡ splitAtRec c [] as

```

1.3.1 splitAtRec

```

lemma splitAtRec-conv:  $\bigwedge$  bs.
  splitAtRec x bs xs =
  (bs @ takeWhile ( $\lambda y. y \neq x$ ) xs, tl(dropWhile ( $\lambda y. y \neq x$ ) xs))
by(induct xs) auto

```

```

lemma splitAtRec-distinct-fst:  $\bigwedge$  s. distinct vs  $\implies$  distinct s  $\implies$  (set s)  $\cap$  (set
vs) = {}  $\implies$  distinct (fst (splitAtRec ram1 s vs))
by (induct vs) auto

```

```

lemma splitAtRec-distinct-snd:  $\bigwedge$  s. distinct vs  $\implies$  distinct s  $\implies$  (set s)  $\cap$  (set
vs) = {}  $\implies$  distinct (snd (splitAtRec ram1 s vs))
by (induct vs) auto

```

```

lemma splitAtRec-ram:
 $\bigwedge$  us a b. ram  $\in$  set vs  $\implies$  (a, b) = splitAtRec ram us vs  $\implies$ 
us @ vs = a @ [ram] @ b

```

```

proof (induct vs)
case Nil then show ?case by simp
next
case (Cons v vs) then show ?case by (auto dest: Cons(1) split: if-split-asm)
qed

```


lemma *splitAtRec-notRam*:
 $\bigwedge us. ram \notin set\ vs \implies splitAtRec\ ram\ us\ vs = (us\ @\ vs, [])$
proof (*induct vs*)
case *Nil* **then show** *?case by simp*
next
case (*Cons v vs*) **then show** *?case by auto*
qed

lemma *splitAtRec-distinct*: $\bigwedge s. distinct\ vs \implies$
 $distinct\ s \implies (set\ s) \cap (set\ vs) = \{\}$
 $set\ (fst\ (splitAtRec\ ram\ s\ vs)) \cap set\ (snd\ (splitAtRec\ ram\ s\ vs)) = \{\}$
by (*induct vs*) *auto*

1.3.2 *splitAt*

lemma *splitAt-conv*:
 $splitAt\ x\ xs = (takeWhile\ (\lambda y. y \neq x)\ xs, tl(dropWhile\ (\lambda y. y \neq x)\ xs))$
by(*simp add: splitAt-def splitAtRec-conv*)

lemma *splitAt-no-ram*[*simp*]:
 $ram \notin set\ vs \implies splitAt\ ram\ vs = (vs, [])$
by (*auto simp: splitAt-def splitAtRec-notRam*)

lemma *splitAt-split*:
 $ram \in set\ vs \implies (a,b) = splitAt\ ram\ vs \implies vs = a\ @\ ram\ \# b$
by (*auto simp: splitAt-def dest: splitAtRec-ram*)

lemma *splitAt-ram*:
 $ram \in set\ vs \implies vs = fst\ (splitAt\ ram\ vs)\ @\ ram\ \# snd\ (splitAt\ ram\ vs)$
by (*rule-tac splitAt-split*) *auto*

lemma *fst-splitAt-last*:
 $\llbracket xs \neq []; distinct\ xs \rrbracket \implies fst\ (splitAt\ (last\ xs)\ xs) = butlast\ xs$
by(*simp add:splitAt-conv takeWhile-not-last*)

1.3.3 Sets

lemma *splitAtRec-union*:
 $\bigwedge a\ b\ s. (a,b) = splitAtRec\ ram\ s\ vs \implies (set\ a \cup set\ b) - \{ram\} = (set\ vs \cup set\ s) - \{ram\}$
apply (*induct vs*) **by** (*auto split: if-split-asm*)

lemma *splitAt-subset-ab*:
 $(a,b) = splitAt\ ram\ vs \implies set\ a \subseteq set\ vs \wedge set\ b \subseteq set\ vs$
apply (*cases ram \in set vs*)
by (*auto dest: splitAt-split simp: splitAt-no-ram*)

lemma *splitAt-in-fst*[*dest*]: $v \in set\ (fst\ (splitAt\ ram\ vs)) \implies v \in set\ vs$
proof (*cases ram \in set vs*)
assume *v: v \in set (fst (splitAt ram vs))*

```

define a where a = fst (splitAt ram vs)
with v have vin: v ∈ set a by auto
define b where b = snd (splitAt ram vs)
case True with a-def b-def have vs = a @ ram # b by (auto dest: splitAt-ram)
with vin show v ∈ set vs by auto
next
  assume v: v ∈ set (fst (splitAt ram vs))
  case False with v show ?thesis by (auto dest: splitAt-no-ram del: notI)
qed

```

lemma *splitAt-not1*:
 $v \notin \text{set } vs \implies v \notin \text{set } (\text{fst } (\text{splitAt } \text{ram } vs))$ **by** (*auto dest: splitAt-in-fst*)

lemma *splitAt-in-snd[dest]*: $v \in \text{set } (\text{snd } (\text{splitAt } \text{ram } vs)) \implies v \in \text{set } vs$
proof (*cases ram ∈ set vs*)
assume *v*: *v* ∈ *set* (*snd* (*splitAt* *ram* *vs*))
define *a* **where** *a* = *fst* (*splitAt* *ram* *vs*)
define *b* **where** *b* = *snd* (*splitAt* *ram* *vs*)
with *v* **have** *vin*: *v* ∈ *set* *b* **by** *auto*
case *True* **with** *a-def* *b-def* **have** *vs* = *a* @ *ram* # *b* **by** (*auto dest: splitAt-ram*)
with *vin* **show** *v* ∈ *set* *vs* **by** *auto*
next
assume *v*: *v* ∈ *set* (*snd* (*splitAt* *ram* *vs*))
case *False* **with** *v* **show** *?thesis* **by** (*auto dest: splitAt-no-ram del: notI*)
qed

1.3.4 Distinctness

lemma *splitAt-distinct-ab-aux*:
 $\text{distinct } vs \implies (a,b) = \text{splitAt } \text{ram } vs \implies \text{distinct } a \wedge \text{distinct } b$
by (*cases ram ∈ set vs*) (*auto dest: splitAt-split simp: splitAt-no-ram*)

lemma *splitAt-distinct-fst-aux[intro]*:
 $\text{distinct } vs \implies \text{distinct } (\text{fst } (\text{splitAt } \text{ram } vs))$
proof –
assume *d*: *distinct vs*
define *b* **where** *b* = *snd* (*splitAt* *ram* *vs*)
define *a* **where** *a* = *fst* (*splitAt* *ram* *vs*)
with *b-def* **have** (*a,b*) = *splitAt* *ram* *vs* **by** *auto*
with *a-def* *d* **show** *?thesis* **by** (*auto dest: splitAt-distinct-ab-aux*)
qed

lemma *splitAt-distinct-snd-aux[intro]*:
 $\text{distinct } vs \implies \text{distinct } (\text{snd } (\text{splitAt } \text{ram } vs))$
proof –
assume *d*: *distinct vs*
define *b* **where** *b* = *snd* (*splitAt* *ram* *vs*)
define *a* **where** *a* = *fst* (*splitAt* *ram* *vs*)
with *b-def* **have** (*a,b*) = *splitAt* *ram* *vs* **by** *auto*

with *b-def d show ?thesis* **by** (*auto dest: splitAt-distinct-ab-aux*)
qed

lemma *splitAt-distinct-ab*:

distinct vs $\implies (a,b) = \text{splitAt ram vs} \implies \text{set } a \cap \text{set } b = \{\}$
apply (*cases ram* \in *set vs*) **apply** (*drule-tac splitAt-split*)
by (*auto simp: splitAt-no-ram*)

lemma *splitAt-distinct-fst-snd*:

distinct vs $\implies \text{set } (\text{fst } (\text{splitAt ram vs})) \cap \text{set } (\text{snd } (\text{splitAt ram vs})) = \{\}$
by (*rule-tac splitAt-distinct-ab*) *simp-all*

lemma *splitAt-distinct-ram-fst*[*intro*]:

distinct vs $\implies \text{ram} \notin \text{set } (\text{fst } (\text{splitAt ram vs}))$
apply (*case-tac ram* \in *set vs*) **apply** (*drule-tac splitAt-ram*)
apply (*rule distinct-split1*) **by** (*force dest: splitAt-in-fst*)**+**

lemma *splitAt-distinct-ram-snd*[*intro*]:

distinct vs $\implies \text{ram} \notin \text{set } (\text{snd } (\text{splitAt ram vs}))$
apply (*case-tac ram* \in *set vs*) **apply** (*drule-tac splitAt-ram*)
apply (*rule distinct-split2*) **by** (*force dest: splitAt-in-fst*)**+**

lemma *splitAt-1*[*simp*]:

splitAt ram [] = ([], []) **by** (*simp add: splitAt-def*)

lemma *splitAt-2*:

$v \in \text{set } vs \implies (a,b) = \text{splitAt ram vs} \implies v \in \text{set } a \vee v \in \text{set } b \vee v = \text{ram}$
apply (*cases ram* \in *set vs*)
by (*auto dest: splitAt-split simp: splitAt-no-ram*)

lemma *splitAt-distinct-fst*: *distinct vs* $\implies \text{distinct } (\text{fst } (\text{splitAt ram1 vs}))$
by (*simp add: splitAt-def splitAtRec-distinct-fst*)

lemma *splitAt-distinct-a*: *distinct vs* $\implies (a,b) = \text{splitAt ram vs} \implies \text{distinct } a$
by (*auto dest: splitAt-distinct-fst pairD*)

lemma *splitAt-distinct-snd*: *distinct vs* $\implies \text{distinct } (\text{snd } (\text{splitAt ram1 vs}))$
by (*simp add: splitAt-def splitAtRec-distinct-snd*)

lemma *splitAt-distinct-b*: *distinct vs* $\implies (a,b) = \text{splitAt ram vs} \implies \text{distinct } b$
by (*auto dest: splitAt-distinct-snd pairD*)

lemma *splitAt-distinct*: *distinct vs* $\implies \text{set } (\text{fst } (\text{splitAt ram vs})) \cap \text{set } (\text{snd } (\text{splitAt ram vs})) = \{\}$
by (*simp add: splitAt-def splitAtRec-distinct*)

lemma *splitAt-subset*: $(a,b) = \text{splitAt ram vs} \implies (\text{set } a \subseteq \text{set } vs) \wedge (\text{set } b \subseteq \text{set } vs)$

apply (*cases* $ram \in set\ vs$) **by** (*auto dest: splitAt-split simp: splitAt-no-ram*)

1.3.5 *splitAt* composition

lemma *set-help*: $v \in set\ (as\ @\ bs) \implies v \in set\ as \vee v \in set\ bs$ **by** *auto*

lemma *splitAt-elements*: $ram1 \in set\ vs \implies ram2 \in set\ vs \implies ram2 \in set\ (fst\ (splitAt\ ram1\ vs)) \vee ram2 \in set\ [ram1] \vee ram2 \in set\ (snd\ (splitAt\ ram1\ vs))$

proof –

assume $r1: ram1 \in set\ vs$ **and** $r2: ram2 \in set\ vs$

then have $ram2 \in set\ (fst\ (splitAt\ ram1\ vs))\ @\ [ram1] \vee ram2 \in set\ (snd\ (splitAt\ ram1\ vs))$

apply (*rule-tac set-help*)

apply (*drule-tac splitAt-ram*) **by** *auto*

then show *?thesis* **by** *auto*

qed

lemma *splitAt-ram3*: $ram2 \notin set\ (fst\ (splitAt\ ram1\ vs)) \implies ram1 \in set\ vs \implies ram2 \in set\ vs \implies ram1 \neq ram2 \implies ram2 \in set\ (snd\ (splitAt\ ram1\ vs))$ **by** (*auto dest: splitAt-elements*)

lemma *splitAt-dist-ram*: *distinct vs* \implies

$vs = a\ @\ ram\ \# \ b \implies (a,b) = splitAt\ ram\ vs$

proof –

assume *dist: distinct vs* **and** *vs: vs = a @ ram # b*

from vs have $r: ram \in set\ vs$ **by** *auto*

with dist vs have $fst\ (splitAt\ ram\ vs) = a$ **apply** (*drule-tac splitAt-ram*) **by** (*rule-tac dist-at1*) *auto*

then have first: $a = fst\ (splitAt\ ram\ vs)$ **by** *auto*

from r dist have second: $b = snd\ (splitAt\ ram\ vs)$ **apply** (*drule-tac splitAt-ram*)

apply (*rule dist-at2*) **apply** *simp*

by (*auto simp: vs*)

show *?thesis* **by** (*auto simp: first second*)

qed

lemma *distinct-unique1*: *distinct vs* $\implies ram \in set\ vs \implies \exists!s. vs = (fst\ s)\ @\ ram\ \# \ (snd\ s)$

proof

assume $d: distinct\ vs$ **and** $r: ram \in set\ vs$

define s **where** $s = splitAt\ ram\ vs$

with r show $vs = (fst\ s)\ @\ ram\ \# \ (snd\ s)$

by (*auto intro: splitAt-ram*)

next

fix s

assume $d: distinct\ vs$ **and** $vs1: vs = fst\ s\ @\ ram\ \# \ snd\ s$

from d vs1 show $s = splitAt\ ram\ vs$ **apply** (*drule-tac splitAt-dist-ram*) **apply** *simp* **by** *simp*

qed

lemma *splitAt-dist-ram2*: $\text{distinct } vs \implies vs = a @ ram1 \# b @ ram2 \# c \implies (a @ ram1 \# b, c) = \text{splitAt } ram2 \text{ vs}$
by (*auto intro: splitAt-dist-ram*)

lemma *splitAt-dist-ram20*: $\text{distinct } vs \implies vs = a @ ram1 \# b @ ram2 \# c \implies c = \text{snd } (\text{splitAt } ram2 \text{ vs})$

proof –
assume *dist*: *distinct vs* **and** *vs*: $vs = a @ ram1 \# b @ ram2 \# c$
then show $c = \text{snd } (\text{splitAt } ram2 \text{ vs})$ **apply** (*drule-tac splitAt-dist-ram2*) **by** (*auto dest: pairD*)
qed

lemma *splitAt-dist-ram21*: $\text{distinct } vs \implies vs = a @ ram1 \# b @ ram2 \# c \implies (a, b) = \text{splitAt } ram1 \text{ (fst } (\text{splitAt } ram2 \text{ vs}))$

proof –
assume *dist*: *distinct vs* **and** *vs*: $vs = a @ ram1 \# b @ ram2 \# c$
then have $\text{fst } (\text{splitAt } ram2 \text{ vs}) = a @ ram1 \# b$ **apply** (*drule-tac splitAt-dist-ram2*)
by (*auto dest: pairD*)
with *dist vs* **show** *?thesis* **by** (*rule-tac splitAt-dist-ram*) *auto*
qed

lemma *splitAt-dist-ram22*: $\text{distinct } vs \implies vs = a @ ram1 \# b @ ram2 \# c \implies (c, []) = \text{splitAt } ram1 \text{ (snd } (\text{splitAt } ram2 \text{ vs}))$

proof –
assume *dist*: *distinct vs* **and** *vs*: $vs = a @ ram1 \# b @ ram2 \# c$
then have $\text{snd } (\text{splitAt } ram2 \text{ vs}) = c$ **apply** (*drule-tac splitAt-dist-ram2*) **by** (*auto dest: pairD*)
with *dist vs* **have** $\text{splitAt } ram1 \text{ (snd } (\text{splitAt } ram2 \text{ vs})) = (c, [])$ **by** (*auto intro: splitAt-no-ram*)
then show *?thesis* **by** *auto*
qed

lemma *splitAt-dist-ram1*: $\text{distinct } vs \implies vs = a @ ram1 \# b @ ram2 \# c \implies (a, b @ ram2 \# c) = \text{splitAt } ram1 \text{ vs}$
by (*auto intro: splitAt-dist-ram*)

lemma *splitAt-dist-ram10*: $\text{distinct } vs \implies vs = a @ ram1 \# b @ ram2 \# c \implies a = \text{fst } (\text{splitAt } ram1 \text{ vs})$

proof –
assume *dist*: *distinct vs* **and** *vs*: $vs = a @ ram1 \# b @ ram2 \# c$
then show $a = \text{fst } (\text{splitAt } ram1 \text{ vs})$ **apply** (*drule-tac splitAt-dist-ram1*) **by** (*auto dest: pairD*)
qed

lemma *splitAt-dist-ram11*: $\text{distinct } vs \implies vs = a @ ram1 \# b @ ram2 \# c \implies (a, []) = \text{splitAt } ram2 \text{ (fst } (\text{splitAt } ram1 \text{ vs}))$

proof –
assume *dist*: *distinct vs* **and** *vs*: $vs = a @ ram1 \# b @ ram2 \# c$
then have $\text{fst } (\text{splitAt } ram1 \text{ vs}) = a$ **apply** (*drule-tac splitAt-dist-ram1*) **by** (*auto*)

dest: pairD
with *dist vs* **have** *splitAt ram2 (fst (splitAt ram1 vs)) = (a, [])* **by** (*auto intro: splitAt-no-ram*)
then show *?thesis* **by** *auto*
qed

lemma *splitAt-dist-ram12*: *distinct vs \implies vs = a @ ram1 # b @ ram2 # c \implies (b, c) = splitAt ram2 (snd (splitAt ram1 vs))*

proof –

assume *dist: distinct vs* **and** *vs: vs = a @ ram1 # b @ ram2 # c*
then have *snd (splitAt ram1 vs) = b @ ram2 # c* **apply** (*drule-tac splitAt-dist-ram1*) **by** (*auto dest: pairD*)
with *dist vs* **show** *?thesis* **by** (*rule-tac splitAt-dist-ram*) *auto*
qed

lemma *splitAt-dist-ram-all*:

distinct vs \implies vs = a @ ram1 # b @ ram2 # c
 \implies *(a, b) = splitAt ram1 (fst (splitAt ram2 vs))*
 \wedge *(c, []) = splitAt ram1 (snd (splitAt ram2 vs))*
 \wedge *(a, []) = splitAt ram2 (fst (splitAt ram1 vs))*
 \wedge *(b, c) = splitAt ram2 (snd (splitAt ram1 vs))*
 \wedge *c = snd (splitAt ram2 vs)*
 \wedge *a = fst (splitAt ram1 vs)*
apply (*rule-tac conjI*) **apply** (*rule-tac splitAt-dist-ram21*) **apply** *simp* **apply** *simp*
apply (*rule-tac conjI*) **apply** (*rule-tac splitAt-dist-ram22*) **apply** *simp* **apply** *simp*
apply (*rule-tac conjI*) **apply** (*rule-tac splitAt-dist-ram11 splitAt-dist-ram22*) **apply** *simp* **apply** *simp*
apply (*rule-tac conjI*) **apply** (*rule-tac splitAt-dist-ram12*) **apply** *simp* **apply** *simp*
apply (*rule-tac conjI*) **apply** (*rule-tac splitAt-dist-ram20*) **apply** *simp* **apply** *simp*
by (*rule-tac splitAt-dist-ram10*) *auto*

1.3.6 Mixed

lemma *fst-splitAt-rev*:

distinct xs \implies x \in set xs \implies
 $\text{fst}(\text{splitAt } x (\text{rev } xs)) = \text{rev}(\text{snd}(\text{splitAt } x xs))$
by(*simp add:splitAt-conv takeWhile-neq-rev*)

lemma *snd-splitAt-rev*:

distinct xs \implies x \in set xs \implies
 $\text{snd}(\text{splitAt } x (\text{rev } xs)) = \text{rev}(\text{fst}(\text{splitAt } x xs))$
by(*simp add:splitAt-conv dropWhile-neq-rev*)

lemma *splitAt-take[simp]*: *distinct ls \implies i < length ls \implies fst (splitAt (ls!i) ls) = take i ls*

proof –

assume d : *distinct* ls **and** si : $i < \text{length } ls$
 then have $ls1$: $ls = \text{take } i \text{ } ls @ ls!i \# \text{drop } (Suc \ i) \text{ } ls$ **by** (*rule-tac id-take-nth-drop*)
 from si **have** $ls!i \in \text{set } ls$ **by** *auto*
 then have $ls2$: $ls = \text{fst } (\text{splitAt } (ls!i) \text{ } ls) @ ls!i \# \text{snd } (\text{splitAt } (ls!i) \text{ } ls)$ **by** (*auto dest: splitAt-ram*)
 from $d \text{ } ls2 \text{ } ls1$ **have** $\text{fst } (\text{splitAt } (ls!i) \text{ } ls) = \text{take } i \text{ } ls \wedge \text{snd } (\text{splitAt } (ls!i) \text{ } ls) = \text{drop } (Suc \ i) \text{ } ls$ **by** (*rule dist-at*)
 then show *?thesis* **by** *auto*
qed

lemma *splitAt-drop[simp]*: $\text{distinct } ls \implies i < \text{length } ls \implies \text{snd } (\text{splitAt } (ls!i) \text{ } ls) = \text{drop } (Suc \ i) \text{ } ls$

proof –

assume d : *distinct* ls **and** si : $i < \text{length } ls$
 then have $ls1$: $ls = \text{take } i \text{ } ls @ ls!i \# \text{drop } (Suc \ i) \text{ } ls$ **by** (*rule-tac id-take-nth-drop*)
 from si **have** $ls!i \in \text{set } ls$ **by** *auto*
 then have $ls2$: $ls = \text{fst } (\text{splitAt } (ls!i) \text{ } ls) @ ls!i \# \text{snd } (\text{splitAt } (ls!i) \text{ } ls)$ **by** (*auto dest: splitAt-ram*)
 from $d \text{ } ls2 \text{ } ls1$ **have** $\text{fst } (\text{splitAt } (ls!i) \text{ } ls) = \text{take } i \text{ } ls \wedge \text{snd } (\text{splitAt } (ls!i) \text{ } ls) = \text{drop } (Suc \ i) \text{ } ls$ **by** (*rule dist-at*)
 then show *?thesis* **by** *auto*
qed

lemma *fst-splitAt-upt*:

$j \leq i \implies i < k \implies \text{fst } (\text{splitAt } i \text{ } [j..<k]) = [j..<i]$
 using *splitAt-take[where* $ls = [j..<k]$ **and** $i=i-j]$
 apply (*simp del:splitAt-take*)
 done

lemma *snd-splitAt-upt*:

$j \leq i \implies i < k \implies \text{snd } (\text{splitAt } i \text{ } [j..<k]) = [i+1..<k]$
 using *splitAt-drop[where* $ls = [j..<k]$ **and** $i=i-j]$
 by *simp*

lemma *local-help1*: $\bigwedge a \text{ vs. } vs = c @ r \# d \implies vs = a @ r \# b \implies r \notin \text{set } a \implies r \notin \text{set } b \implies a = c$

proof (*induct* c)

case *Nil*

then have ra : $r \notin \text{set } a$ **and** $vs1$: $vs = a @ r \# b$ **and** $vs2$: $vs = [] @ r \# d$
 by *auto*

from $vs1 \text{ } vs2$ **have** $a @ r \# b = r \# d$ **by** *auto*

then have $\text{hd } (a @ r \# b) = r$ **by** *auto*

then have $a \neq [] \implies \text{hd } a = r$ **by** *auto*

then have $a \neq [] \implies r \in \text{set } a$ **by** (*induct* a) *auto*

with ra **have** $a = []$ **by** *auto*

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

next

case (*Cons* $x \text{ } xs$) **then show** *?case* **by** (*induct* a) *auto*

qed

lemma *local-help*: $vs = a @ r \# b \implies vs = c @ r \# d \implies r \notin \text{set } a \implies r \notin \text{set } b \implies a = c \wedge b = d$

proof –

assume *dist*: $r \notin \text{set } a \ r \notin \text{set } b$ **and** *vs1*: $vs = a @ r \# b$ **and** *vs2*: $vs = c @ r \# d$

from *vs2 vs1 dist* **have** $a = c$ **by** (*rule local-help1*)

with *dist vs1 vs2* **show** *?thesis* **by** *simp*

qed

lemma *local-help'*: $a @ r \# b = c @ r \# d \implies r \notin \text{set } a \implies r \notin \text{set } b \implies a = c \wedge b = d$

by (*rule local-help*) *auto*

lemma *splitAt-simp1*: $\text{ram} \notin \text{set } a \implies \text{ram} \notin \text{set } b \implies \text{fst} (\text{splitAt } \text{ram} (a @ \text{ram} \# b)) = a$

proof –

assume *ramab*: $\text{ram} \notin \text{set } a \ \text{ram} \notin \text{set } b$

define *vs* **where** $vs = a @ \text{ram} \# b$

then have *vs*: $vs = a @ \text{ram} \# b$ **by** *auto*

then have $\text{ram} \in \text{set } vs$ **by** *auto*

then have $vs = \text{fst} (\text{splitAt } \text{ram} vs) @ \text{ram} \# \text{snd} (\text{splitAt } \text{ram} vs)$ **by** (*auto dest: splitAt-ram*)

with *vs ramab* **show** *?thesis* **apply** *simp* **apply** (*rule-tac sym*) **apply** (*rule-tac local-help1*) **apply** *simp*

apply (*rule sym*) **apply** *assumption* **by** *auto*

qed

lemma *help'''-in*: $\bigwedge xs. \text{ram} \in \text{set } b \implies \text{fst} (\text{splitAtRec } \text{ram} \ xs \ b) = xs @ \text{fst} (\text{splitAtRec } \text{ram} \ [] \ b)$

proof (*induct b*)

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

next

case (*Cons b bs*) **show** *?case* **using** *Cons(2)*

apply (*case-tac b = ram*) **apply** *simp*

apply *simp*

apply (*subgoal-tac fst (splitAtRec ram (xs @ [b]) bs) = (xs@[b]) @ fst (splitAtRec ram [] bs)*) **apply** *simp*

apply (*subgoal-tac fst (splitAtRec ram [b] bs) = [b] @ fst (splitAtRec ram [] bs)*)

apply *simp*

apply (*rule Cons*) **apply** *force*

apply (*rule Cons*) **by** *force*

qed

lemma *help'''-notin*: $\bigwedge xs. \text{ram} \notin \text{set } b \implies \text{fst} (\text{splitAtRec } \text{ram} \ xs \ b) = xs @ \text{fst} (\text{splitAtRec } \text{ram} \ [] \ b)$

proof (*induct b*)
case Nil then show *?case by auto*
next
case (Cons b bs)
then have $ram \notin set\ bs$ **by auto**
then show *?case*
 apply (*case-tac b = ram*) **apply simp**
 apply simp
 apply (*subgoal-tac fst (splitAtRec ram (xs @ [b]) bs) = (xs@[b]) @ fst (splitAtRec ram [] bs)*) **apply simp**
 apply (*subgoal-tac fst (splitAtRec ram [b] bs) = [b] @ fst (splitAtRec ram [] bs)*)
apply simp
 apply (*rule Cons*) **apply simp**
 apply (*rule Cons*) **by simp**
qed

lemma help''': $fst\ (splitAtRec\ ram\ xs\ b) = xs\ @\ fst\ (splitAtRec\ ram\ []\ b)$
apply (*cases ram ∈ set b*)
apply (*rule-tac help'''-in*) **apply simp**
apply (*rule-tac help'''-notin*) **apply simp done**

lemma splitAt-simpA[simp]: $fst\ (splitAt\ ram\ (ram\ \# \ b)) = []$ **by** (*simp add: splitAt-def*)
lemma splitAt-simpB[simp]: $ram \neq a \implies fst\ (splitAt\ ram\ (a\ \# \ b)) = a\ \# \ fst\ (splitAt\ ram\ b)$ **apply** (*simp add: splitAt-def*)
 apply (*subgoal-tac fst (splitAtRec ram [a] b) = [a] @ fst (splitAtRec ram [] b)*)
apply simp by (*rule help'''*)
lemma splitAt-simpB'[simp]: $a \neq ram \implies fst\ (splitAt\ ram\ (a\ \# \ b)) = a\ \# \ fst\ (splitAt\ ram\ b)$ **apply** (*rule splitAt-simpB*) **by auto**
lemma splitAt-simpC[simp]: $ram \notin set\ a \implies fst\ (splitAt\ ram\ (a\ @ \ b)) = a\ @ \ fst\ (splitAt\ ram\ b)$
apply (*induct a*) **by auto**

lemma help'''': $\bigwedge\ xs\ ys.\ snd\ (splitAtRec\ ram\ xs\ b) = snd\ (splitAtRec\ ram\ ys\ b)$
apply (*induct b*) **by auto**

lemma splitAt-simpD[simp]: $\bigwedge\ a.\ ram \neq a \implies snd\ (splitAt\ ram\ (a\ \# \ b)) = snd\ (splitAt\ ram\ b)$ **apply** (*simp add: splitAt-def*)
by (*rule help''''*)
lemma splitAt-simpD'[simp]: $\bigwedge\ a.\ a \neq ram \implies snd\ (splitAt\ ram\ (a\ \# \ b)) = snd\ (splitAt\ ram\ b)$ **apply** (*rule splitAt-simpD*) **by auto**

lemma splitAt-simpE[simp]: $snd\ (splitAt\ ram\ (ram\ \# \ b)) = b$ **by** (*simp add: splitAt-def*)

lemma splitAt-simpF[simp]: $ram \notin set\ a \implies snd\ (splitAt\ ram\ (a\ @ \ b)) = snd\ (splitAt\ ram\ b)$
apply (*induct a*) **by auto**

lemma *splitAt-rotate-pair-conv*:
 $\bigwedge xs. \llbracket \text{distinct } xs; x \in \text{set } xs \rrbracket$
 $\implies \text{snd } (\text{splitAt } x \text{ (rotate } n \text{ } xs)) \text{ @ fst } (\text{splitAt } x \text{ (rotate } n \text{ } xs)) =$
 $\text{snd } (\text{splitAt } x \text{ } xs) \text{ @ fst } (\text{splitAt } x \text{ } xs)$
apply(*induct n*) **apply** *simp*
apply(*simp del:rotate-Suc2 add:rotate1-rotate-swap*)
apply(*case-tac xs*) **apply** *clarsimp+*
apply(*erule disjE*) **apply** *simp*
apply(*drule split-list*)
apply *clarsimp*
done

1.4 *between*

definition *between* :: 'a list \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a list **where**
between *vs ram₁ ram₂* \equiv
 $\text{let } (pre_1, post_1) = \text{splitAt } ram_1 \text{ } vs \text{ in}$
 $\text{if } ram_2 \in \text{set } post_1$
 $\text{then let } (pre_2, post_2) = \text{splitAt } ram_2 \text{ } post_1 \text{ in } pre_2$
 $\text{else let } (pre_2, post_2) = \text{splitAt } ram_2 \text{ } pre_1 \text{ in } post_1 \text{ @ } pre_2$

lemma *inbetween-inset*:
 $x \in \text{set}(\text{between } xs \text{ } a \text{ } b) \implies x \in \text{set } xs$
apply(*simp add:between-def split-def split-if-split-asm*)
apply(*blast dest:splitAt-in-snd*)
apply(*blast dest:splitAt-in-snd*)
done

lemma *notinset-notinbetween*:
 $x \notin \text{set } xs \implies x \notin \text{set}(\text{between } xs \text{ } a \text{ } b)$
by(*blast dest:inbetween-inset*)

lemma *set-between-id*:
 $\text{distinct } xs \implies x \in \text{set } xs \implies$
 $\text{set}(\text{between } xs \text{ } x \text{ } x) = \text{set } xs - \{x\}$
apply(*drule split-list*)
apply (*clarsimp simp:between-def split-def Un-commute*)
done

lemma *split-between*:
 $\llbracket \text{distinct } vs; r \in \text{set } vs; v \in \text{set } vs; u \in \text{set}(\text{between } vs \text{ } r \text{ } v) \rrbracket \implies$
 $\text{between } vs \text{ } r \text{ } v =$
 $(\text{if } r=u \text{ then } \llbracket \text{else between } vs \text{ } r \text{ } u \text{ @ } [u] \rrbracket \text{ @ between } vs \text{ } u \text{ } v$
apply(*cases r = v*)
apply(*clarsimp*)
apply(*frule split-list[of v]*)

```

apply(clarsimp)
apply(simp add:between-def split-def split:if-split-asm)
apply(erule disjE)
  apply(frule split-list[of u])
  apply(clarsimp)
  apply(frule split-list[of u])
  apply(clarsimp)
apply(clarsimp)
apply(frule split-list[of r])
apply(clarsimp)
apply(rename-tac as bs)
apply(erule disjE)
  apply(frule split-list[of v])
  apply(clarsimp)
  apply(rename-tac cs ds)
apply(subgoal-tac between (cs @ v # ds @ r # bs) r v = bs @ cs)
  prefer 2 apply(simp add:between-def split-def split:if-split-asm)
apply simp
apply(erule disjE)
  apply(frule split-list[of u])
  apply(clarsimp simp:between-def split-def split:if-split-asm)
apply(frule split-list[of u])
apply(clarsimp simp:between-def split-def split:if-split-asm)
apply(frule split-list[of v])
apply(clarsimp)
apply(rename-tac cs ds)
apply(subgoal-tac between (as @ r # cs @ v # ds) r v = cs)
  prefer 2 apply(simp add:between-def split-def split:if-split-asm)
apply simp
apply(frule split-list[of u])
apply(clarsimp simp:between-def split-def split:if-split-asm)
done

```

1.5 Tables

type-synonym (*'a*, *'b*) *table* = (*'a* × *'b*) *list*

definition *isTable* :: (*'a* ⇒ *'b*) ⇒ *'a list* ⇒ (*'a*, *'b*) *table* ⇒ *bool* **where**
isTable f vs t ≡ ∀ *p*. *p* ∈ *set t* → *snd p* = *f (fst p)* ∧ *fst p* ∈ *set vs*

lemma *isTable-eq*: *isTable E vs ((a,b)#ps)* ⇒ *b = E a*
by (*auto simp add: isTable-def*)

lemma *isTable-subset*:
set qs ⊆ *set ps* ⇒ *isTable E vs ps* ⇒ *isTable E vs qs*
by (*unfold isTable-def*) *auto*

lemma *isTable-Cons*: *isTable E vs ((a,b)#ps)* ⇒ *isTable E vs ps*
by (*unfold isTable-def*) *auto*

definition *removeKey* :: 'a ⇒ ('a × 'b) list ⇒ ('a × 'b) list **where**
removeKey a ps ≡ [p ← ps. a ≠ fst p]

primrec *removeKeyList* :: 'a list ⇒ ('a × 'b) list ⇒ ('a × 'b) list **where**
removeKeyList [] ps = ps
| *removeKeyList* (w#ws) ps = *removeKey* w (*removeKeyList* ws ps)

lemma *removeKey-subset[simp]*: set (*removeKey* a ps) ⊆ set ps
by (*simp* add: *removeKey-def*)

lemma *length-removeKey[simp]*: |*removeKey* w ps| ≤ |ps|
by (*simp* add: *removeKey-def*)

lemma *length-removeKeyList*:
length (*removeKeyList* ws ps) ≤ *length* ps (**is** ?P ws)
proof (*induct* ws)
 show ?P [] **by** *simp*
 fix w ws
 have *length* (*removeKey* w (*removeKeyList* ws ps))
 ≤ *length* (*removeKeyList* ws ps)
 by (*rule* *length-removeKey*)
 also assume ?P ws
 finally show ?P (w#ws) **by** *simp*
qed

lemma *removeKeyList-subset[simp]*: set (*removeKeyList* ws ps) ⊆ set ps
proof (*induct* ws)
 case Nil **then show** ?case **by** *simp*
next
 case (Cons w ws) **then show** ?case
 by (*metis* *dual-order.trans* *removeKeyList.simps(2)* *removeKey-subset*)
qed

lemma *notin-removeKey1*: (a, b) ∉ set (*removeKey* a ps)
by (*induct* ps) (*auto* *simp* add: *removeKey-def*)

lemma *removeKeyList-eq*:
removeKeyList as ps = [p ← ps. ∀ a ∈ set as. a ≠ fst p]
by (*induct* as) (*simp-all* add: *filter-comm* *removeKey-def*)

lemma *removeKey-empty[simp]*: *removeKey* a [] = []
by (*simp* add: *removeKey-def*)

lemma *removeKeyList-empty[simp]*: *removeKeyList* ps [] = []
by (*induct* ps) *simp-all*

lemma *removeKeyList-cons[simp]*:
removeKeyList ws (p#ps)
= (*if* fst p ∈ set ws *then* *removeKeyList* ws ps *else* p#(*removeKeyList* ws ps))

```

    by (induct ws) (simp-all split: if-split-asm add: removeKey-def)

end
theory Quasi-Order
imports Main
begin

locale quasi-order =
fixes gle :: 'a ⇒ 'a ⇒ bool (infix <≲> 60)
assumes gle-refl[iff]: x ≲ x
and gle-trans: x ≲ y ⇒ y ≲ z ⇒ x ≲ z
begin

definition in-gle :: 'a ⇒ 'a set ⇒ bool (infix <∈≲> 60) where
x ∈≲ M ≡ ∃ y ∈ M. x ≲ y

definition subseteq-gle :: 'a set ⇒ 'a set ⇒ bool (infix <⊆≲> 60) where
M ⊆≲ N ≡ ∀ x ∈ M. x ∈≲ N

definition seteq-gle :: 'a set ⇒ 'a set ⇒ bool (infix <=≲> 60) where
M =≲ N ≡ M ⊆≲ N ∧ N ⊆≲ M

lemmas defs = in-gle-def subseteq-gle-def seteq-gle-def

lemma subseteq-gle-refl[simp]: M ⊆≲ M
by(auto simp add: subseteq-gle-def in-gle-def)

lemma subseteq-gle-trans: A ⊆≲ B ⇒ B ⊆≲ C ⇒ A ⊆≲ C
by (simp add: subseteq-gle-def in-gle-def) (metis gle-trans)

lemma empty-subseteq-gle[simp]: {} ⊆≲ A
by (simp add: subseteq-gle-def)

lemma subseteq-gleI2: (∧ x. x ∈ M ⇒ ∃ y ∈ N. x ≲ y) ⇒ M ⊆≲ N
by (auto simp add: subseteq-gle-def in-gle-def)

lemma subseteq-gleD2: M ⊆≲ N ⇒ x ∈ M ⇒ ∃ y ∈ N. x ≲ y
by (auto simp add: subseteq-gle-def in-gle-def)

lemma seteq-gle-refl[iff]: A =≲ A
by (simp add: seteq-gle-def)

lemma seteq-gle-trans: A =≲ B ⇒ B =≲ C ⇒ A =≲ C
by (simp add: seteq-gle-def) (metis subseteq-gle-trans)

end

end

```

2 Isomorphisms Between Plane Graphs

```
theory PlaneGraphIso
imports Main Quasi-Order
begin
```

```
lemma image-image-id-if[simp]: ( $\bigwedge x. f(f x) = x$ )  $\implies f \circ f \circ M = M$ 
by (auto simp: image-iff)
```

```
declare not-None-eq [iff] not-Some-eq [iff]
```

The symbols \cong and \simeq are overloaded. They denote congruence and isomorphism on arbitrary types. On lists (representing faces of graphs), \cong means congruence modulo rotation; \simeq is currently unused. On graphs, \simeq means isomorphism and is a weaker version of \cong (proper isomorphism): \simeq also allows to reverse the orientation of all faces.

```
consts
pr-isomorphic :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool (infix <math>\cong</math> 60)
```

```
definition Iso :: ('a list * 'a list) set ( $\langle \cong \rangle$ ) where
 $\{ \cong \} \equiv \{ (F_1, F_2). F_1 \cong F_2 \}$ 
```

```
lemma [iff]:  $((x, y) \in \{ \cong \}) = x \cong y$ 
by (simp add: Iso-def)
```

A plane graph is a set or list (for executability) of faces (hence *Fgraph* and *fgraph*) and a face is a list of nodes:

```
type-synonym 'a Fgraph = 'a list set
type-synonym 'a fgraph = 'a list list
```

2.1 Equivalence of faces

Two faces are equivalent modulo rotation:

```
overloading congs  $\equiv$  pr-isomorphic :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool
begin
definition  $F_1 \cong (F_2 :: 'a list) \equiv \exists n. F_2 = \text{rotate } n \ F_1$ 
end
```

```
lemma congs-refl[iff]:  $(xs :: 'a list) \cong xs$ 
apply (simp add: congs-def)
apply (rule-tac  $x = 0$  in exI)
apply (simp)
done
```

```
lemma congs-sym: assumes  $A: (xs :: 'a list) \cong ys$  shows  $ys \cong xs$ 
```

```

proof (simp add:congs-def)
  let ?l = length xs
  from A obtain n where ys: ys = rotate n xs by(fastforce simp add:congs-def)
  have xs = rotate ?l xs by simp
  also have ... = rotate (?l - n mod ?l + n mod ?l) xs
  proof (cases)
    assume xs = [] thus ?thesis by simp
  next
    assume xs ≠ []
    hence n mod ?l < ?l by simp
    hence ?l = ?l - n mod ?l + n mod ?l by arith
    thus ?thesis by simp
  qed
  also have ... = rotate (?l - n mod ?l) (rotate (n mod ?l) xs)
    by(simp add:rotate-rotate)
  also have rotate (n mod ?l) xs = rotate n xs
    by(rule rotate-conv-mod[symmetric])
  finally show ∃ m. xs = rotate m ys by(fastforce simp add:ys)
qed

```

```

lemma congs-trans: (xs::'a list) ≅ ys ⇒ ys ≅ zs ⇒ xs ≅ zs
apply(clarsimp simp:congs-def rotate-def)
apply(rename-tac m n)
apply(rule-tac x = n+m in exI)
apply (simp add:funpow-add)
done

```

```

lemma equiv-EqF: equiv (UNIV::'a list set) {≅}
apply(unfold equiv-def sym-def trans-def refl-on-def)
apply(rule conjI)
  apply simp
apply(rule conjI)
  apply(fastforce intro:congs-sym)
apply(fastforce intro:congs-trans)
done

```

```

lemma congs-distinct:
   $F_1 \cong F_2 \implies \text{distinct } F_2 = \text{distinct } F_1$ 
by (auto simp: congs-def)

```

```

lemma congs-length:
   $F_1 \cong F_2 \implies \text{length } F_2 = \text{length } F_1$ 
by (auto simp: congs-def)

```

```

lemma congs-pres-nodes:  $F_1 \cong F_2 \implies \text{set } F_1 = \text{set } F_2$ 
by(clarsimp simp:congs-def)

```

```

lemma congs-map:
   $F_1 \cong F_2 \implies \text{map } f F_1 \cong \text{map } f F_2$ 

```

by (*auto simp: congs-def rotate-map*)

lemma *congs-map-eq-iff*:

inj-on f (set xs ∪ set ys) ⟹ (map f xs ≅ map f ys) = (xs ≅ ys)

apply(*simp add: congs-def*)

apply(*rule iffI*)

apply(*clarsimp simp: rotate-map*)

apply(*drule map-inj-on*)

apply(*simp add: Un-commute*)

apply (*fastforce*)

apply *clarsimp*

apply(*fastforce simp: rotate-map*)

done

lemma *list-cong-rev-iff*[*simp*]:

(rev xs ≅ rev ys) = (xs ≅ ys)

apply(*simp add: congs-def rotate-rev*)

apply(*rule iffI*)

apply *fast*

apply *clarify*

apply(*cases length xs = 0*)

apply *simp*

apply(*case-tac n mod length xs = 0*)

apply(*rule-tac x = n in exI*)

apply *simp*

apply(*subst rotate-conv-mod*)

apply(*rule-tac x = length xs - n mod length xs in exI*)

apply *simp*

done

lemma *singleton-list-cong-eq-iff*[*simp*]:

({xs::'a list} // {≅} = {ys} // {≅}) = (xs ≅ ys)

by(*simp add: eq-equiv-class-iff2[OF equiv-EqF]*)

2.2 Homomorphism and isomorphism

definition *is-pr-Hom* :: (*'a* ⇒ *'b*) ⇒ *'a* *Fgraph* ⇒ *'b* *Fgraph* ⇒ *bool* **where**

is-pr-Hom φ *F*₁ *F*₂ ≡ (*map* φ ‘ *F*₁) // {≅} = *F*₂ // {≅}

definition *is-pr-Iso* :: (*'a* ⇒ *'b*) ⇒ *'a* *Fgraph* ⇒ *'b* *Fgraph* ⇒ *bool* **where**

is-pr-Iso φ *F*₁ *F*₂ ≡ *is-pr-Hom* φ *F*₁ *F*₂ ∧ *inj-on* φ (⋃ *F* ∈ *F*₁. *set F*)

definition *is-pr-iso* :: (*'a* ⇒ *'b*) ⇒ *'a* *fgraph* ⇒ *'b* *fgraph* ⇒ *bool* **where**

is-pr-iso φ *F*₁ *F*₂ ≡ *is-pr-Iso* φ (*set F*₁) (*set F*₂)

Homomorphisms preserve the set of nodes.

lemma *UN-subset-iff*: ((⋃ *i* ∈ *I*. *f i*) ⊆ *B*) = (∀ *i* ∈ *I*. *f i* ⊆ *B*)

by *blast*

declare *Image-Collect-case-prod*[*simp del*]

lemma *pr-Hom-pres-face-nodes*:

is-pr-Hom φ Fs_1 $Fs_2 \implies (\bigcup F \in Fs_1. \{\varphi \text{ ' } (set F)\}) = (\bigcup F \in Fs_2. \{set F\})$

supply *image-cong-simp* [*cong del*]

apply(*clarsimp simp:is-pr-Hom-def quotient-def*)

apply *auto*

apply(*subgoal-tac* $\exists F' \in Fs_2. \{\cong\}$ “ $\{map \varphi F\} = \{\cong\}$ “ $\{F'\}$)

prefer 2 **apply** *blast*

apply (*fastforce simp: eq-equiv-class-iff*[*OF equiv-EqF*] *dest!:congs-pres-nodes*)

apply(*subgoal-tac* $\exists F' \in Fs_1. \{\cong\}$ “ $\{map \varphi F'\} = \{\cong\}$ “ $\{F\}$)

apply (*fastforce simp: eq-equiv-class-iff*[*OF equiv-EqF*] *dest!:congs-pres-nodes*)

apply (*erule equalityE*)

apply(*fastforce simp:UN-subset-iff*)

done

lemma *pr-Hom-pres-nodes*:

assumes *is-pr-Hom* φ Fs_1 Fs_2

shows $\varphi \text{ ' } (\bigcup F \in Fs_1. set F) = (\bigcup F \in Fs_2. set F)$

proof

from *assms* **have** *: $(\bigcup F \in Fs_1. \{\varphi \text{ ' } set F\}) = (\bigcup F \in Fs_2. \{set F\})$

by (*rule pr-Hom-pres-face-nodes*)

then show $\varphi \text{ ' } (\bigcup F \in Fs_1. set F) \subseteq (\bigcup F \in Fs_2. set F)$

by *blast*

show $(\bigcup F \in Fs_2. set F) \subseteq \varphi \text{ ' } (\bigcup F \in Fs_1. set F)$

proof

fix x

assume $x \in (\bigcup F \in Fs_2. set F)$

then obtain F **where** $F \in Fs_2$ **and** $x \in set F$..

then have $set F \in (\bigcup F \in Fs_2. \{set F\})$

by *blast*

then have $set F \in (\bigcup F \in Fs_1. \{\varphi \text{ ' } set F\})$

using * **by** *simp*

then obtain F' **where** $F' \in Fs_1$ **and** $set F \in \{\varphi \text{ ' } set F'\}$..

with $\langle x \in set F \rangle$ **show** $x \in \varphi \text{ ' } (\bigcup F \in Fs_1. set F)$

by *auto*

qed

qed

Therefore isomorphisms preserve cardinality of node set.

lemma *pr-Iso-same-no-nodes*:

$\llbracket is-pr-Iso \varphi Fs_1 Fs_2; finite Fs_1 \rrbracket$

$\implies card(\bigcup F \in Fs_1. set F) = card(\bigcup F \in Fs_2. set F)$

by(*clarsimp simp add: is-pr-Iso-def pr-Hom-pres-nodes*[*symmetric*] *card-image*)

lemma *pr-iso-same-no-nodes*:

is-pr-iso φ Fs_1 $Fs_2 \implies card(\bigcup F \in set Fs_1. set F) = card(\bigcup F \in set Fs_2. set F)$

by(*simp add: is-pr-iso-def pr-Iso-same-no-nodes*)

Isomorphisms preserve the number of faces.

lemma *pr-iso-same-no-faces*:

assumes *dist1: distinct Fs₁ and dist2: distinct Fs₂*
and *inj1: inj-on (λxs. {xs} // {≅}) (set Fs₁)*
and *inj2: inj-on (λxs. {xs} // {≅}) (set Fs₂) and iso: is-pr-iso φ Fs₁ Fs₂*
shows *length Fs₁ = length Fs₂*

proof –

have *injphi: ∀ F ∈ set Fs₁. ∀ F' ∈ set Fs₁. inj-on φ (set F ∪ set F')* **using** *iso*
by(*auto simp: is-pr-iso-def is-pr-Iso-def is-pr-Hom-def inj-on-def*)
have *inj1': inj-on (λxs. {xs} // {≅}) (map φ ' set Fs₁)*
apply(*rule inj-on-imageI*)
apply(*simp add: inj-on-def quotient-def eq-equiv-class-iff [OF equiv-EqF]*)
apply(*simp add: congs-map-eq-iff injphi*)
using *inj1*
apply(*simp add: inj-on-def quotient-def eq-equiv-class-iff [OF equiv-EqF]*)
done
have *length Fs₁ = card (set Fs₁)* **by**(*simp add: distinct-card [OF dist1]*)
also have *... = card (map φ ' set Fs₁)* **using** *iso*
by(*auto simp: is-pr-iso-def is-pr-Iso-def is-pr-Hom-def inj-on-mapI card-image*)
also have *... = card ((map φ ' set Fs₁) // {≅})*
by(*simp add: card-quotient-disjoint [OF - inj1']*)
also have *(map φ ' set Fs₁) // {≅} = set Fs₂ // {≅}*
using *iso* **by**(*simp add: is-pr-iso-def is-pr-Iso-def is-pr-Hom-def*)
also have *card (...) = card (set Fs₂)*
by(*simp add: card-quotient-disjoint [OF - inj2]*)
also have *... = length Fs₂* **by**(*simp add: distinct-card [OF dist2]*)
finally show *?thesis .*

qed

lemma *is-Hom-distinct*:

\llbracket *is-pr-Hom φ Fs₁ Fs₂; ∀ F ∈ Fs₁. distinct F; ∀ F' ∈ Fs₂. distinct F* \rrbracket
 \implies $\forall F \in Fs_1. \text{distinct}(\text{map } \varphi F)$

apply(*clarsimp simp add: is-pr-Hom-def*)
apply(*subgoal-tac ∃ F' ∈ Fs₂. (map φ F, F') : {≅}*)
apply(*fastforce simp add: congs-def*)
apply(*subgoal-tac ∃ F' ∈ Fs₂. {map φ F} // {≅} = {F'} // {≅}*)
apply *clarify*
apply(*rule-tac x = F' in bexI*)
apply(*rule eq-equiv-class [OF - equiv-EqF]*)
apply(*simp add: singleton-quotient*)
apply *blast*
apply *assumption*
apply(*simp add: quotient-def*)
apply(*rotate-tac 1*)
apply *blast*
done

```

lemma Collect-congs-eq-iff[simp]:
  Collect ((≅) x) = Collect ((≅) y) ↔ (x ≅ (y::'a list))
using eq-equiv-class-iff2[OF equiv-EqF]
apply(simp add: quotient-def Iso-def)
apply blast
done

lemma is-pr-Hom-trans: assumes f: is-pr-Hom f A B and g: is-pr-Hom g B C
shows is-pr-Hom (g ∘ f) A C
proof –
  from f have f1: ∀ a∈A. ∃ b∈B. map f a ≅ b
    apply(simp add: is-pr-Hom-def quotient-def Iso-def)
    apply(erule equalityE)
    apply blast
    done
  from f have f2: ∀ b∈B. ∃ a∈A. map f a ≅ b
    apply(simp add: is-pr-Hom-def quotient-def Iso-def)
    apply(erule equalityE)
    apply blast
    done
  from g have g1: ∀ b∈B. ∃ c∈C. map g b ≅ c
    apply(simp add: is-pr-Hom-def quotient-def Iso-def)
    apply(erule equalityE)
    apply blast
    done
  from g have g2: ∀ c∈C. ∃ b∈B. map g b ≅ c
    apply(simp add: is-pr-Hom-def quotient-def Iso-def)
    apply(erule equalityE)
    apply blast
    done
  show ?thesis
    apply(auto simp add: is-pr-Hom-def quotient-def Iso-def Image-def
      map-comp-map[symmetric] image-comp simp del: map-map map-comp-map)
    apply (metis congs-map[of - - g] congs-trans f1 g1)
    by (metis congs-map[of - - g] congs-sym congs-trans f2 g2)
qed

lemma is-pr-Hom-rev:
  is-pr-Hom φ A B ⇒ is-pr-Hom φ (rev ' A) (rev ' B)
apply(auto simp add: is-pr-Hom-def quotient-def Image-def Iso-def rev-map[symmetric])
apply(erule equalityE)
apply blast
apply(erule equalityE)
apply blast
done

```

A kind of recursion rule, a first step towards executability:

lemma *is-pr-Iso-rec*:

$\llbracket \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) F_{s_1}; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) F_{s_2}; F_1 \in F_{s_1} \rrbracket \implies$
 $\text{is-pr-Iso } \varphi F_{s_1} F_{s_2} =$
 $(\exists F_2 \in F_{s_2}. \text{length } F_1 = \text{length } F_2 \wedge \text{is-pr-Iso } \varphi (F_{s_1} - \{F_1\}) (F_{s_2} - \{F_2\}))$
 $\wedge (\exists n. \text{map } \varphi F_1 = \text{rotate } n F_2)$
 $\wedge \text{inj-on } \varphi (\bigcup F \in F_{s_1}. \text{set } F)$

apply(*drule mk-disjoint-insert*[of F_1])
apply *clarify*
apply(*rename-tac* F_{s_1})
apply(*rule iffI*)

apply (*clarsimp simp add:is-pr-Iso-def*)
apply(*clarsimp simp:is-pr-Hom-def quotient-diff1*)
apply(*drule-tac s=a // b for a b in sym*)
apply(*clarsimp*)
apply(*subgoal-tac* $\{\cong\}$ “ $\{\text{map } \varphi F_1\} : F_{s_2} // \{\cong\}$ ”)
prefer 2 **apply**(*simp add:quotient-def*)
apply(*erule quotientE*)
apply(*rename-tac* F_2)
apply(*drule eq-equiv-class*[*OF - equiv-EqF*])
apply *blast*
apply(*rule-tac* $x = F_2$ **in** *beXI*)
prefer 2 **apply** *assumption*
apply(*rule conjI*)
apply(*clarsimp simp: congs-def*)
apply(*rule conjI*)
apply(*subgoal-tac* $\{\cong\}$ “ $\{F_2\} = \{\cong\}$ ” “ $\{\text{map } \varphi F_1\}$ ”)
prefer 2
apply(*rule equiv-class-eq*[*OF equiv-EqF*])
apply(*fastforce intro: congs-sym*)
apply(*subgoal-tac* $\{F_2\} // \{\cong\} = \{\text{map } \varphi F_1\} // \{\cong\}$)
prefer 2 **apply**(*simp add:singleton-quotient*)
apply(*subgoal-tac* $\forall F \in F_{s_1}'. \neg (\text{map } \varphi F) \cong (\text{map } \varphi F_1)$)
apply(*fastforce simp:Iso-def quotient-def Image-Collect-case-prod simp del: Collect-congs-eq-iff*
dest!: eq-equiv-class[*OF - equiv-EqF*])
apply *clarify*
apply(*subgoal-tac inj-on* φ (*set* $F \cup \text{set } F_1$))
prefer 2
apply(*erule subset-inj-on*)
apply(*blast*)
apply(*clarsimp simp add:congs-map-eq-iff*)
apply(*subgoal-tac* $\{\cong\}$ “ $\{F_1\} = \{\cong\}$ ” “ $\{F\}$ ”)
apply(*simp add:singleton-quotient*)
apply(*rule equiv-class-eq*[*OF equiv-EqF*])
apply(*blast intro:congs-sym*)
apply(*subgoal-tac* $F_2 \cong (\text{map } \varphi F_1)$)
apply (*simp add:congs-def inj-on-Un*)
apply(*clarsimp intro!:congs-sym*)

```

apply(clarsimp simp add: is-pr-Iso-def is-pr-Hom-def quotient-diff1)
apply (simp add:singleton-quotient)
apply(subgoal-tac F2 ≅ (map φ F1))
  prefer 2 apply(fastforce simp add:congs-def)
apply(subgoal-tac {≅} “{map φ F1} = {≅} “{F2})
  prefer 2
  apply(rule equiv-class-eq[OF equiv-EqF])
  apply(fastforce intro:congs-sym)
apply(subgoal-tac {≅} “{F2} ∈ Fs2 // {≅})
  prefer 2 apply(erule quotientI)
apply (simp add:insert-absorb quotient-def)
done

```

lemma *is-iso-Cons*:

```

[[ distinct (F1#Fs1'); distinct Fs2;
  inj-on (λxs.{xs} // {≅}) (set(F1#Fs1'); inj-on (λxs.{xs} // {≅}) (set Fs2) ]]
⇒
is-pr-iso φ (F1#Fs1' Fs2 =
(∃ F2 ∈ set Fs2. length F1 = length F2 ∧ is-pr-iso φ Fs1' (remove1 F2 Fs2)
  ∧ (∃ n. map φ F1 = rotate n F2)
  ∧ inj-on φ (set F1 ∪ (∪ F ∈ set Fs1'. set F)))
apply(simp add:is-pr-iso-def)
apply(subst is-pr-Iso-rec[where ?F1.0 = F1])
apply(simp-all)
done

```

2.3 Isomorphism tests

lemma *map-upd-submap*:

```

x ∉ dom m ⇒ (m(x ↦ y) ⊆m m') = (m' x = Some y ∧ m ⊆m m')
apply(simp add:map-le-def dom-def)
apply(rule iffI)
  apply(rule conjI) apply (blast intro:sym)
  apply clarify
  apply(case-tac a=x)
  apply auto
done

```

lemma *map-of-zip-submap*: [[*length xs = length ys; distinct xs*]] ⇒

```

(map-of (zip xs ys) ⊆m Some ∘ f) = (map f xs = ys)
apply(induct rule: list-induct2)
  apply(simp)
apply (clarsimp simp: map-upd-submap simp del:o-apply fun-upd-apply)
apply simp
done

```

primrec *pr-iso-test0* :: (*'a* → *'b*) ⇒ *'a fgraph* ⇒ *'b fgraph* ⇒ *bool* **where**

```

pr-iso-test0 m [] Fs2 = (Fs2 = [])
| pr-iso-test0 m (F1#Fs1) Fs2 =
  (∃ F2 ∈ set Fs2. length F1 = length F2 ∧
   (∃ n. let m' = map-of(zip F1 (rotate n F2)) in
    if m ⊆m m ++ m' ∧ inj-on (m++m') (dom(m++m'))
    then pr-iso-test0 (m ++ m') Fs1 (remove1 F2 Fs2) else False))

```

lemma *map-compatI*: $\llbracket f \subseteq_m \text{Some} \circ h; g \subseteq_m \text{Some} \circ h \rrbracket \implies f \subseteq_m f++g$
by (*fastforce simp add: map-le-def map-add-def dom-def split:option.splits*)

lemma *inj-on-map-addI1*:

```

 $\llbracket \text{inj-on } m \ A; m \subseteq_m m++m'; A \subseteq \text{dom } m \rrbracket \implies \text{inj-on } (m++m') \ A$ 
apply (clarsimp simp add: inj-on-def map-add-def map-le-def dom-def
  split:option.splits)

```

```

apply(rule conjI)
apply fastforce
apply auto
apply fastforce
apply (rename-tac x a y)
apply(subgoal-tac m x = Some a)
prefer 2 apply (fastforce)
apply(subgoal-tac m y = Some a)
prefer 2 apply (fastforce)
apply(subgoal-tac m x = m y)
prefer 2 apply simp
apply (blast)
done

```

lemma *map-image-eq*: $\llbracket A \subseteq \text{dom } m; m \subseteq_m m' \rrbracket \implies m \text{ ' } A = m' \text{ ' } A$
by(*force simp:map-le-def dom-def split:option.splits*)

lemma *inj-on-map-add-Un*:

```

 $\llbracket \text{inj-on } m \ (\text{dom } m); \text{inj-on } m' \ (\text{dom } m'); m \subseteq_m \text{Some} \circ f; m' \subseteq_m \text{Some} \circ f;$ 
 $\text{inj-on } f \ (\text{dom } m' \cup \text{dom } m); A = \text{dom } m'; B = \text{dom } m \rrbracket$ 
 $\implies \text{inj-on } (m ++ m') \ (A \cup B)$ 

```

```

apply(simp add:inj-on-Un)
apply(rule conjI)
apply(fastforce intro!: inj-on-map-addI1 map-compatI)
apply(clarify)
apply(subgoal-tac m ++ m' ⊆m Some ∘ f)
prefer 2 apply(fast intro:map-add-le-mapI map-compatI)
apply(subgoal-tac dom m' - dom m ⊆ dom(m++m'))
prefer 2 apply(fastforce)
apply(insert map-image-eq[of dom m' - dom m m++m' Some ∘ f])
apply(subgoal-tac dom m - dom m' ⊆ dom(m++m'))
prefer 2 apply(fastforce)
apply(insert map-image-eq[of dom m - dom m' m++m' Some ∘ f])
apply (clarsimp simp add: image-comp [symmetric])
apply blast

```

done

lemma *map-of-zip-eq-SomeD*: $\text{length } xs = \text{length } ys \implies$
 $\text{map-of } (\text{zip } xs \ ys) \ x = \text{Some } y \implies y \in \text{set } ys$
apply (*induct rule:list-induct2*)
 apply *simp*
apply (*auto split:if-splits*)
done

lemma *inj-on-map-of-zip*:
 $\llbracket \text{length } xs = \text{length } ys; \text{distinct } ys \rrbracket$
 $\implies \text{inj-on } (\text{map-of } (\text{zip } xs \ ys)) \ (\text{set } xs)$
apply (*induct rule:list-induct2*)
 apply *simp*
apply *clarsimp*
apply (*rule conjI*)
 apply (*erule inj-on-fun-updI*)
 apply (*simp add:image-def*)
 apply *clarsimp*
 apply (*drule (1) map-of-zip-eq-SomeD[OF - sym]*)
 apply *fast*
apply (*clarsimp simp add:image-def*)
apply (*drule (1) map-of-zip-eq-SomeD[OF - sym]*)
apply *fast*
done

lemma *pr-iso-test0-correct*: $\bigwedge m \ Fs_2.$
 $\llbracket \forall F \in \text{set } Fs_1. \text{distinct } F; \forall F \in \text{set } Fs_2. \text{distinct } F;$
 $\text{distinct } Fs_1; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) \ (\text{set } Fs_1);$
 $\text{distinct } Fs_2; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) \ (\text{set } Fs_2); \text{inj-on } m \ (\text{dom } m) \rrbracket \implies$
 $\text{pr-iso-test0 } m \ Fs_1 \ Fs_2 =$
 $(\exists \varphi. \text{is-pr-iso } \varphi \ Fs_1 \ Fs_2 \wedge m \subseteq_m \text{Some } \circ \varphi \wedge$
 $\text{inj-on } \varphi \ (\text{dom } m \cup (\bigcup F \in \text{set } Fs_1. \text{set } F)))$
apply (*induct Fs₁*)
 apply (*simp add:inj-on-def dom-def*)
 apply (*rule iffI*)
 apply (*simp add:is-pr-iso-def is-pr-Iso-def is-pr-Hom-def*)
 apply (*rule-tac x = the \circ m in exI*)
 apply (*fastforce simp: map-le-def*)
 apply (*clarsimp simp:is-pr-iso-def is-pr-Iso-def is-pr-Hom-def*)
apply (*rename-tac F₁ Fs₁' m Fs₂*)
apply (*clarsimp simp:Let-def Ball-def*)
apply (*simp add: is-iso-Cons*)
apply (*rule iffI*)

apply *clarify*
apply (*clarsimp simp add:map-of-zip-submap inj-on-diff*)
apply (*rule-tac x = \varphi in exI*)
apply (*rule conjI*)

```

apply(rule-tac  $x = F_2$  in be $xI$ )
  prefer 2 apply assumption
apply(frule map-add-le-mapE)
apply(simp add:map-of-zip-submap is-pr-iso-def is-pr-Iso-def)
apply(rule conjI)
  apply blast
apply(erule subset-inj-on)
apply blast
apply(rule conjI)
  apply(blast intro: map-le-trans)
apply(erule subset-inj-on)
apply blast

apply(clarsimp simp: inj-on-diff)
apply(rule-tac  $x = F_2$  in be $xI$ )
  prefer 2 apply assumption
apply simp
apply(rule-tac  $x = n$  in ex $I$ )
apply(rule conjI)
apply clarsimp
apply(rule-tac  $x = \varphi$  in ex $I$ )
apply simp
apply(rule conjI)
  apply(fastforce intro!:map-add-le-mapI simp:map-of-zip-submap)
apply(simp add:Un-ac)
apply(rule context-conjI)
apply(simp add:map-of-zip-submap[symmetric])
apply(erule (1) map-compatI)
apply(simp add:map-of-zip-submap[symmetric])
apply(erule inj-on-map-add-Un)
  apply(simp add:inj-on-map-of-zip)
  apply assumption
  apply assumption
  apply simp
  apply(erule subset-inj-on)
  apply fast
  apply simp
apply(rule refl)
done

corollary pr-iso-test0-corr:
   $\llbracket \forall F \in \text{set } Fs_1. \text{distinct } F; \forall F \in \text{set } Fs_2. \text{distinct } F;$ 
   $\text{distinct } Fs_1; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) (\text{set } Fs_1);$ 
   $\text{distinct } Fs_2; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) (\text{set } Fs_2) \rrbracket \implies$ 
   $\text{pr-iso-test0 } \text{Map.empty } Fs_1 \text{ } Fs_2 = (\exists \varphi. \text{is-pr-iso } \varphi \text{ } Fs_1 \text{ } Fs_2)$ 
apply(subst pr-iso-test0-correct)
apply assumption+
apply simp
apply (simp add:is-pr-iso-def is-pr-Iso-def)

```


done

Now we bound the number of rotations needed. We have to exclude the empty face \square to be able to restrict the search to $n < \text{length } xs$ (which would otherwise be vacuous).

```
primrec pr-iso-test1 :: ('a  $\rightarrow$  'b)  $\Rightarrow$  'a fgraph  $\Rightarrow$  'b fgraph  $\Rightarrow$  bool where
  pr-iso-test1 m  $\square$  Fs2 = (Fs2 =  $\square$ )
| pr-iso-test1 m (F1 # Fs1) Fs2 =
  ( $\exists$  F2  $\in$  set Fs2. length F1 = length F2  $\wedge$ 
   ( $\exists$  n < length F2. let m' = map-of(zip F1 (rotate n F2)) in
    if m  $\subseteq_m$  m ++ m'  $\wedge$  inj-on (m ++ m') (dom(m ++ m'))
    then pr-iso-test1 (m ++ m') Fs1 (remove1 F2 Fs2) else False))
```

lemma *test0-conv-test1*:

```
 $\bigwedge$  m Fs2.  $\square \notin$  set Fs2  $\implies$  pr-iso-test1 m Fs1 Fs2 = pr-iso-test0 m Fs1 Fs2
apply(induct Fs1)
apply simp
apply simp
apply(rule iffI)
apply blast
apply (clarsimp simp:Let-def)
apply(rule-tac x = F2 in exI)
prefer 2 apply assumption
apply simp
apply(subgoal-tac F2  $\neq$   $\square$ )
prefer 2 apply blast
apply(rule-tac x = n mod length F2 in exI)
apply(simp add:rotate-conv-mod[symmetric])
done
```

Thus correctness carries over to *pr-iso-test1*:

corollary *pr-iso-test1-corr*:

```
 $\llbracket \forall F \in \text{set } Fs_1. \text{distinct } F; \forall F \in \text{set } Fs_2. \text{distinct } F; \square \notin \text{set } Fs_2;
\text{distinct } Fs_1; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) (\text{set } Fs_1);
\text{distinct } Fs_2; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) (\text{set } Fs_2) \rrbracket \implies
\text{pr-iso-test1 } \text{Map.empty } Fs_1 \text{ } Fs_2 = (\exists \varphi. \text{is-pr-iso } \varphi \text{ } Fs_1 \text{ } Fs_2)$ 
by(simp add: test0-conv-test1 pr-iso-test0-corr)
```

2.3.1 Implementing maps by lists

The representation are lists of pairs with no repetition in the first or second component.

definition *oneone* :: ('a * 'b)list \Rightarrow *bool* **where**

oneone *xys* \equiv *distinct*(*map* *fst* *xys*) \wedge *distinct*(*map* *snd* *xys*)

declare *oneone-def*[*simp*]

type-synonym

('a, 'b)tester = ('a * 'b)list \Rightarrow ('a * 'b)list \Rightarrow *bool*

type-synonym

$(\text{'a}, \text{'b})\text{merger} = (\text{'a} * \text{'b})\text{list} \Rightarrow (\text{'a} * \text{'b})\text{list} \Rightarrow (\text{'a} * \text{'b})\text{list}$

primrec $\text{pr-iso-test2} :: (\text{'a}, \text{'b})\text{tester} \Rightarrow (\text{'a}, \text{'b})\text{merger} \Rightarrow$

$(\text{'a} * \text{'b})\text{list} \Rightarrow \text{'a fgraph} \Rightarrow \text{'b fgraph} \Rightarrow \text{bool}$ **where**

$\text{pr-iso-test2} \text{ tst mrg } I \ [] \ F_{s_2} = (F_{s_2} = [])$

| $\text{pr-iso-test2} \text{ tst mrg } I \ (F_1 \# F_{s_1}) \ F_{s_2} =$

$(\exists F_2 \in \text{set } F_{s_2}. \text{length } F_1 = \text{length } F_2 \wedge$

$(\exists n < \text{length } F_2. \text{let } I' = \text{zip } F_1 \ (\text{rotate } n \ F_2) \ \text{in}$

$\text{if } \text{tst } I' \ I$

$\text{then } \text{pr-iso-test2} \ \text{tst mrg} \ (\text{mrg } I' \ I) \ F_{s_1} \ (\text{remove1 } F_2 \ F_{s_2}) \ \text{else } \text{False})$

lemma $\text{notin-range-map-of}$:

$y \notin \text{snd } \text{' set } xys \Longrightarrow \text{Some } y \notin \text{range}(\text{map-of } xys)$

apply($\text{induct } xys$)

apply ($\text{simp add: image-def}$)

apply($\text{clarsimp split: if-splits}$)

done

lemma inj-on-map-upd :

$\llbracket \text{inj-on } m \ (\text{dom } m); \text{Some } y \notin \text{range } m \rrbracket \Longrightarrow \text{inj-on } (m(x \mapsto y)) \ (\text{dom } m)$

apply($\text{simp add: inj-on-def dom-def image-def}$)

apply (blast intro: sym)

done

lemma [simp]:

$\text{distinct}(\text{map } \text{snd } xys) \Longrightarrow \text{inj-on } (\text{map-of } xys) \ (\text{dom}(\text{map-of } xys))$

apply($\text{induct } xys$)

apply simp

apply ($\text{simp add: notin-range-map-of inj-on-map-upd}$)

apply($\text{clarsimp simp add: image-def}$)

apply($\text{drule map-of-SomeD}$)

apply fastforce

done

lemma lem : $\text{Ball } (\text{set } xs) \ P \Longrightarrow \text{Ball } (\text{set } (\text{remove1 } x \ xs)) \ P = \text{True}$

by($\text{induct } xs$) simp-all

lemma $\text{pr-iso-test2-conv-1}$:

$\bigwedge I \ F_{s_2}.$

$\llbracket \forall I \ I'. \text{oneone } I \longrightarrow \text{oneone } I' \longrightarrow$

$\text{tst } I' \ I = (\text{let } m = \text{map-of } I; m' = \text{map-of } I'$

$\text{in } m \subseteq_m m \ ++ \ m' \wedge \text{inj-on } (m \ ++ \ m') \ (\text{dom}(m \ ++ \ m')) \rrbracket;$

$\forall I \ I'. \text{oneone } I \longrightarrow \text{oneone } I' \longrightarrow \text{tst } I' \ I$

$\longrightarrow \text{map-of}(\text{mrg } I' \ I) = \text{map-of } I \ ++ \ \text{map-of } I';$

$\forall I \ I'. \text{oneone } I \wedge \text{oneone } I' \longrightarrow \text{tst } I' \ I \longrightarrow \text{oneone } (\text{mrg } I' \ I);$

$\text{oneone } I;$

$\forall F \in \text{set } F_{s_1}. \text{distinct } F; \forall F \in \text{set } F_{s_2}. \text{distinct } F \rrbracket \Longrightarrow$

```

  pr-iso-test2 tst mrg I Fs1 Fs2 = pr-iso-test1 (map-of I) Fs1 Fs2
apply(induct Fs1)
apply simp
apply(simp add:Let-def lem inj-on-map-of-zip del: mod-less cong: conj-cong)
done

```

A simple implementation

```

definition compat :: ('a,'b)tester where
  compat I I' ==
   $\forall (x,y) \in \text{set } I. \forall (x',y') \in \text{set } I'. (x = x') = (y = y')$ 

```

```

lemma image-map-upd:
   $x \notin \text{dom } m \implies m(x \rightarrow y) \text{ ' } A = m \text{ ' } (A - \{x\}) \cup (\text{if } x \in A \text{ then } \{\text{Some } y\} \text{ else } \{\})$ 
by(auto simp:image-def dom-def)

```

```

lemma image-map-of-conv-Image:
   $\bigwedge A. [\text{distinct}(\text{map } \text{fst } \text{xys})]$ 
   $\implies \text{map-of } \text{xys} \text{ ' } A = \text{Some} \text{ ' } (\text{set } \text{xys} \text{ " } A) \cup (\text{if } A \subseteq \text{fst} \text{ ' set } \text{xys} \text{ then } \{\} \text{ else } \{\text{None}\})$ 
supply image-cong-simp [cong del]
apply (induct xys)
apply (simp add:image-def Image-def Collect-conv-if)
apply (simp add:image-map-upd dom-map-of-conv-image-fst)
apply(erule thin-rl)
apply (clarsimp simp:image-def Image-def)
apply((rule conjI, clarify)+, fastforce)
apply fastforce
apply(clarify)
apply((rule conjI, clarify)+, fastforce)
apply fastforce
apply fastforce
apply fastforce
done

```

```

lemma [simp]:  $m ++ m' \text{ ' } (\text{dom } m' - A) = m' \text{ ' } (\text{dom } m' - A)$ 
apply(clarsimp simp add:map-add-def image-def dom-def inj-on-def split:option.splits)
apply auto
apply (blast intro:sym)
apply (blast intro:sym)
apply (rule-tac  $x = xa$  in bxI)
prefer 2 apply blast
apply simp
done

```

```

declare Diff-subset [iff]

```

```

lemma compat-correct:

```

```

[[ oneone I; oneone I' ]] ==>
  compat I' I = (let m = map-of I; m' = map-of I'
                in m ⊆m m ++ m' ∧ inj-on (m++m') (dom(m++m')))
apply(simp add: compat-def Let-def map-le-iff-map-add-commute)
apply(rule iffI)
apply(rule context-conjI)
apply(rule ext)
apply (fastforce simp add:map-add-def split:option.split)
apply(simp add:inj-on-Un)
apply(drule sym)
apply simp
apply(simp add: dom-map-of-conv-image-fst image-map-of-conv-Image)
apply(simp add: image-def Image-def)
apply fastforce
apply clarsimp
apply(rename-tac a b aa ba)
apply(rule iffI)
apply (clarsimp simp: fun-eq-iff)
apply(erule-tac x = aa in allE)
apply (simp add:map-add-def)
apply (clarsimp simp: dom-map-of-conv-image-fst)
apply(simp (no-asm-use) add:inj-on-def)
apply(drule-tac x = a in bspec)
apply force
apply(drule-tac x = aa in bspec)
apply force
apply(erule mp)
apply simp
apply(drule sym)
apply simp
done

```

corollary *compat-corr*:

```

∀ I I'. oneone I → oneone I' →
  compat I' I = (let m = map-of I; m' = map-of I'
                in m ⊆m m ++ m' ∧ inj-on (m++m') (dom(m++m')))
by(simp add: compat-correct)

```

definition *merge0* :: ('a,'b)merger **where**

```

merge0 I' I ≡ [xy ← I'. fst xy ∉ fst ' set I] @ I

```

lemma *help1*:

```

distinct(map fst xys) ==> map-of (filter P xys) =
  map-of xys |' {x. ∃ y. (x,y) ∈ set xys ∧ P(x,y)}
apply(induct xys)
apply simp
apply(rule ext)
apply (simp add:restrict-map-def)

```

apply *force*
done

lemma *merge0-correct*:

$\forall I I'. \text{oneone } I \longrightarrow \text{oneone } I' \longrightarrow \text{compat } I' I$
 $\longrightarrow \text{map-of}(\text{merge0 } I' I) = \text{map-of } I ++ \text{map-of } I'$

apply(*simp add:compat-def merge0-def help1 fun-eq-iff map-add-def restrict-map-def split:option.split*)

apply *fastforce*

done

lemma *merge0-inv*:

$\forall I I'. \text{oneone } I \wedge \text{oneone } I' \longrightarrow \text{compat } I' I \longrightarrow \text{oneone } (\text{merge0 } I' I)$

apply(*auto simp add:merge0-def distinct-map compat-def split-def*)

apply(*blast intro:subset-inj-on*)⁺

done

corollary *pr-iso-test2-corr*:

$\llbracket \forall F \in \text{set } Fs_1. \text{distinct } F; \forall F \in \text{set } Fs_2. \text{distinct } F; \llbracket \notin \text{set } Fs_2;$
 $\text{distinct } Fs_1; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) (\text{set } Fs_1);$
 $\text{distinct } Fs_2; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) (\text{set } Fs_2) \rrbracket \implies$
 $\text{pr-iso-test2 compat merge0 } \llbracket Fs_1 Fs_2 = (\exists \varphi. \text{is-pr-iso } \varphi Fs_1 Fs_2)$
by(*simp add: pr-iso-test2-conv-1[OF compat-corr merge0-correct merge0-inv] pr-iso-test1-corr*)

Implementing merge as a recursive function:

primrec *merge* :: ('a,'b)merger **where**

merge $\llbracket I = I$

| *merge* (*xy#xys*) *I* = (*let* (*x,y*) = *xy* *in*

if $\forall (x',y') \in \text{set } I. x \neq x'$ *then* *xy* # *merge* *xys* *I* *else* *merge* *xys* *I*)

lemma *merge-conv-merge0*: *merge* *I' I* = *merge0* *I' I*

apply(*induct I'*)

apply(*simp add:merge0-def*)

apply(*force simp add:Let-def list-all-iff merge0-def*)

done

primrec *pr-iso-test-rec* :: ('a * 'b)list \Rightarrow 'a fgraph \Rightarrow 'b fgraph \Rightarrow bool **where**

pr-iso-test-rec *I* $\llbracket Fs_2 = (Fs_2 = \llbracket)$

| *pr-iso-test-rec* *I* (*F*₁#*F*₁) *F*₂ =

($\exists F_2 \in \text{set } Fs_2. \text{length } F_1 = \text{length } F_2 \wedge$

($\exists n < \text{length } F_2. \text{let } I' = \text{zip } F_1 (\text{rotate } n F_2)$ *in*

compat *I' I* \wedge *pr-iso-test-rec* (*merge* *I' I*) *F*₁ (*remove1* *F*₂ *F*₂)))

lemma *pr-iso-test-rec-conv-2*:

$\bigwedge I Fs_2. \text{pr-iso-test-rec } I Fs_1 Fs_2 = \text{pr-iso-test2 compat merge0 } I Fs_1 Fs_2$

apply(*induct Fs*₁)

apply *simp*

apply(*auto simp: merge-conv-merge0 list-ex-iff Bex-def Let-def*)
done

corollary *pr-iso-test-rec-corr*:

$\llbracket \forall F \in \text{set } Fs_1. \text{distinct } F; \forall F \in \text{set } Fs_2. \text{distinct } F; \llbracket \notin \text{set } Fs_2;$
 $\text{distinct } Fs_1; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) (\text{set } Fs_1);$
 $\text{distinct } Fs_2; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) (\text{set } Fs_2) \rrbracket \implies$
 $\text{pr-iso-test-rec } \llbracket Fs_1 Fs_2 = (\exists \varphi. \text{is-pr-iso } \varphi Fs_1 Fs_2)$
by(*simp add: pr-iso-test-rec-conv-2 pr-iso-test2-corr*)

definition *pr-iso-test* :: '*a fgraph* \Rightarrow '*b fgraph* \Rightarrow *bool* **where**
pr-iso-test $Fs_1 Fs_2 = \text{pr-iso-test-rec } \llbracket Fs_1 Fs_2$

corollary *pr-iso-test-correct*:

$\llbracket \forall F \in \text{set } Fs_1. \text{distinct } F; \forall F \in \text{set } Fs_2. \text{distinct } F; \llbracket \notin \text{set } Fs_2;$
 $\text{distinct } Fs_1; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) (\text{set } Fs_1);$
 $\text{distinct } Fs_2; \text{inj-on } (\lambda xs. \{xs\} // \{\cong\}) (\text{set } Fs_2) \rrbracket \implies$
 $\text{pr-iso-test } Fs_1 Fs_2 = (\exists \varphi. \text{is-pr-iso } \varphi Fs_1 Fs_2)$
apply(*simp add: pr-iso-test-def pr-iso-test-rec-corr*)
done

2.3.2 ‘Improper’ Isomorphisms

definition *is-Is* :: ('*a* \Rightarrow '*b*) \Rightarrow '*a Fgraph* \Rightarrow '*b Fgraph* \Rightarrow *bool* **where**
is-Is $\varphi Fs_1 Fs_2 \equiv \text{is-pr-Is } \varphi Fs_1 Fs_2 \vee \text{is-pr-Is } \varphi Fs_1 (\text{rev } 'Fs_2)$

definition *is-iso* :: ('*a* \Rightarrow '*b*) \Rightarrow '*a fgraph* \Rightarrow '*b fgraph* \Rightarrow *bool* **where**
is-iso $\varphi Fs_1 Fs_2 \equiv \text{is-Is } \varphi (\text{set } Fs_1) (\text{set } Fs_2)$

definition *iso-fgraph* :: '*a fgraph* \Rightarrow '*a fgraph* \Rightarrow *bool* (**infix** $\langle \simeq \rangle$ 60) **where**
 $g_1 \simeq g_2 \equiv \exists \varphi. \text{is-iso } \varphi g_1 g_2$

lemma *iso-fgraph-trans*: **assumes** $f \simeq (g :: 'a fgraph)$ **and** $g \simeq h$ **shows** $f \simeq h$
proof –

{ **fix** $\varphi \varphi'$ **assume** *is-pr-Hom* $\varphi (\text{set } f) (\text{set } g)$ *inj-on* $\varphi (\bigcup F \in \text{set } f. \text{set } F)$
is-pr-Hom $\varphi' (\text{set } g) (\text{set } h)$ *inj-on* $\varphi' (\bigcup F \in \text{set } g. \text{set } F)$
hence *is-pr-Hom* $(\varphi' \circ \varphi) (\text{set } f) (\text{set } h) \wedge$
inj-on $(\varphi' \circ \varphi) (\bigcup F \in \text{set } f. \text{set } F)$
by(*simp add: is-pr-Hom-trans comp-inj-on pr-Hom-pres-nodes*)
} moreover
{ **fix** $\varphi \varphi'$ **assume** *is-pr-Hom* $\varphi (\text{set } f) (\text{set } g)$ *inj-on* $\varphi (\bigcup F \in \text{set } f. \text{set } F)$
is-pr-Hom $\varphi' (\text{set } g) (\text{rev } ' \text{set } h)$ *inj-on* $\varphi' (\bigcup F \in \text{set } g. \text{set } F)$
hence *is-pr-Hom* $(\varphi' \circ \varphi) (\text{set } f) (\text{rev } ' \text{set } h) \wedge$
inj-on $(\varphi' \circ \varphi) (\bigcup F \in \text{set } f. \text{set } F)$
by(*simp add: is-pr-Hom-trans comp-inj-on pr-Hom-pres-nodes*)
} moreover
{ **fix** $\varphi \varphi'$ **assume** *is-pr-Hom* $\varphi (\text{set } f) (\text{rev } ' \text{set } g)$ *inj-on* $\varphi (\bigcup F \in \text{set } f. \text{set } F)$
is-pr-Hom $\varphi' (\text{set } g) (\text{set } h)$ *inj-on* $\varphi' (\bigcup F \in \text{set } g. \text{set } F)$

```

with this( $\beta$ )[THEN is-pr-Hom-rev]
have is-pr-Hom ( $\varphi' \circ \varphi$ ) (set f) (rev ' set h)  $\wedge$ 
      inj-on ( $\varphi' \circ \varphi$ ) ( $\bigcup F \in \text{set } f. \text{ set } F$ )
      by(simp add: is-pr-Hom-trans comp-inj-on pr-Hom-pres-nodes)
} moreover
{ fix  $\varphi \varphi'$  assume is-pr-Hom  $\varphi$  (set f) (rev ' set g) inj-on  $\varphi$  ( $\bigcup F \in \text{set } f. \text{ set } F$ )
      is-pr-Hom  $\varphi'$  (set g) (rev ' set h) inj-on  $\varphi'$  ( $\bigcup F \in \text{set } g. \text{ set } F$ )
      with this( $\beta$ )[THEN is-pr-Hom-rev]
      have is-pr-Hom ( $\varphi' \circ \varphi$ ) (set f) (set h)  $\wedge$ 
            inj-on ( $\varphi' \circ \varphi$ ) ( $\bigcup F \in \text{set } f. \text{ set } F$ )
            by(simp add: is-pr-Hom-trans comp-inj-on pr-Hom-pres-nodes)
      } ultimately show ?thesis using assms
      by(simp add: iso-fgraph-def is-iso-def is-Iso-def is-pr-Iso-def) blast
qed

```

definition *iso-test* :: 'a fgraph \Rightarrow 'b fgraph \Rightarrow bool **where**
iso-test $g_1 g_2 \longleftrightarrow \text{pr-iso-test } g_1 g_2 \vee \text{pr-iso-test } g_1 (\text{map rev } g_2)$

theorem *iso-correct*:

```

[[  $\forall F \in \text{set } Fs_1. \text{distinct } F; \forall F \in \text{set } Fs_2. \text{distinct } F; [] \notin \text{set } Fs_2;$ 
   distinct  $Fs_1$ ; inj-on ( $\lambda xs. \{xs\} // \{\cong\}$ ) (set  $Fs_1$ );
   distinct  $Fs_2$ ; inj-on ( $\lambda xs. \{xs\} // \{\cong\}$ ) (set  $Fs_2$ ) ]]  $\implies$ 
iso-test  $Fs_1 Fs_2 = (Fs_1 \simeq Fs_2)$ 
apply(simp add: iso-test-def pr-iso-test-correct iso-fgraph-def)
apply(subst pr-iso-test-correct)
      apply simp
      apply simp
      apply (simp add: image-def)
      apply simp
      apply simp
      apply (simp add: distinct-map)
      apply (simp add: inj-on-image-iff)
apply(simp add: is-iso-def is-Iso-def is-pr-iso-def)
apply blast
done

```

lemma *iso-fgraph-refl*[*iff*]: $g \simeq g$

```

apply(simp add: iso-fgraph-def)
apply(rule-tac x = id in exI)
apply(simp add: is-iso-def is-Iso-def is-pr-Iso-def is-pr-Hom-def id-def)
done

```

2.4 Elementhood and containment modulo

interpretation *qle-gr*: *quasi-order* (\simeq)

proof **qed** (*auto intro: iso-fgraph-trans*)

abbreviation $gle\text{-}gr\text{-}in :: 'a\ fgraph \Rightarrow 'a\ fgraph\ set \Rightarrow bool$ (**infix** $\langle \in_{\sim} \rangle 60$)
where $x \in_{\sim} M \equiv gle\text{-}gr.in\text{-}gle\ x\ M$
abbreviation $gle\text{-}gr\text{-}sub :: 'a\ fgraph\ set \Rightarrow 'a\ fgraph\ set \Rightarrow bool$ (**infix** $\langle \subseteq_{\sim} \rangle 60$)
where $x \subseteq_{\sim} M \equiv gle\text{-}gr.subseteq\text{-}gle\ x\ M$
abbreviation $gle\text{-}gr\text{-}eq :: 'a\ fgraph\ set \Rightarrow 'a\ fgraph\ set \Rightarrow bool$ (**infix** $\langle =_{\sim} \rangle 60$)
where $x =_{\sim} M \equiv gle\text{-}gr.seteq\text{-}gle\ x\ M$

end

3 More Rotation

theory *Rotation*
imports *ListAux PlaneGraphIso*
begin

definition $rotate\text{-}to :: 'a\ list \Rightarrow 'a \Rightarrow 'a\ list$ **where**
 $rotate\text{-}to\ vs\ v \equiv v \# snd\ (splitAt\ v\ vs) \ @\ fst\ (splitAt\ v\ vs)$

definition $rotate\text{-}min :: nat\ list \Rightarrow nat\ list$ **where**
 $rotate\text{-}min\ vs \equiv rotate\text{-}to\ vs\ (min\text{-}list\ vs)$

lemma *cong-rotate-to:*

$x \in set\ xs \Longrightarrow xs \cong rotate\text{-}to\ xs\ x$

proof –

assume $x: x \in set\ xs$

hence $ls1: xs = fst\ (splitAt\ x\ xs) \ @\ x \# snd\ (splitAt\ x\ xs)$ **by** (*auto dest: splitAt-ram*)

define i **where** $i = length\ (fst\ (splitAt\ x\ xs))$

hence $i < length\ ((fst\ (splitAt\ x\ xs)) \ @\ [x] \ @\ snd\ (splitAt\ x\ xs))$ **by** *auto*

with $ls1$ **have** $i\text{-}len: i < length\ xs$ **by** *auto*

hence $ls2: xs = take\ i\ xs \ @\ xs!i \# drop\ (Suc\ i)\ xs$ **by** (*auto intro: id-take-nth-drop*)

from $i\text{-}len$ **have** $length\ (take\ i\ xs) = i$ **by** *auto*

with $i\text{-}def$ **have** $len\text{-}eq: length\ (take\ i\ xs) = length\ (fst\ (splitAt\ x\ xs))$ **by** *auto*

moreover

from $ls1\ ls2$ **have** $eq: take\ i\ xs \ @\ xs!i \# drop\ (Suc\ i)\ xs = fst\ (splitAt\ x\ xs) \ @\ x \# snd\ (splitAt\ x\ xs)$ **by** *simp*

ultimately **have**

$v\text{-}simp: x = xs!i$ **and**

$take\text{-}i: fst\ (splitAt\ x\ xs) = take\ i\ xs$ **and**

$drop\text{-}i': snd\ (splitAt\ x\ xs) = drop\ (Suc\ i)\ xs$ **by** *auto*

from $i\text{-}len$ **have** $ls3: xs = take\ i\ xs \ @\ drop\ i\ xs$ **by** *auto*

with $take\text{-}i$ **have** $eq: xs = fst\ (splitAt\ x\ xs) \ @\ drop\ i\ xs$ **by** *auto*

with $ls1$ **have** $fst\ (splitAt\ x\ xs) \ @\ drop\ i\ xs = fst\ (splitAt\ x\ xs) \ @\ x \# snd\ (splitAt\ x\ xs)$ **by** *auto*

then **have** $drop\text{-}i: drop\ i\ xs = x \# snd\ (splitAt\ x\ xs)$ **by** *auto*

have $rotate\ i\ xs = drop\ (i\ mod\ length\ xs)\ xs \ @\ take\ (i\ mod\ length\ xs)\ xs$ **by** (*rule rotate-drop-take*)

with $i\text{-}len$ **have** $rotate\ i\ xs = drop\ i\ xs \ @\ take\ i\ xs$ **by** *auto*


```

with take-i drop-i have rotate i xs = (x # snd(splitAt x xs)) @ fst(splitAt x xs)
by auto
thus ?thesis apply (auto simp: congs-def rotate-to-def) apply (rule exI) apply
(rule sym) .
qed

```

```

lemma face-cong-if-norm-eq:
  [| rotate-min xs = rotate-min ys; xs ≠ []; ys ≠ [] |] ⇒ xs ≅ ys
apply(simp add:rotate-min-def)
apply(subgoal-tac xs ≅ rotate-to xs (Min (set xs)))
apply(subgoal-tac ys ≅ rotate-to ys (Min (set ys)))
apply(simp) apply(blast intro:congs-sym congs-trans)
apply(simp add: cong-rotate-to)
apply(drule sym)
apply(simp add: cong-rotate-to)
done

```

```

lemma norm-eq-if-face-cong:
  [| xs ≅ ys; distinct xs; xs ≠ [] |] ⇒ rotate-min xs = rotate-min ys
by(auto simp:congs-def rotate-min-def rotate-to-def
  splitAt-rotate-pair-conv insert-absorb)

```

```

lemma norm-eq-iff-face-cong:
  [| distinct xs; xs ≠ []; ys ≠ [] |] ⇒
  (rotate-min xs = rotate-min ys) = (xs ≅ ys)
by(blast intro: face-cong-if-norm-eq norm-eq-if-face-cong)

```

```

lemma inj-on-rotate-min-iff:
assumes ∀ vs ∈ A. distinct vs [] ∉ A
shows inj-on rotate-min A = inj-on (λvs. {vs} // {≅}) A
proof -
  { fix xs ys assume xs: xs ∈ A and ys : ys ∈ A
    hence xs ≠ [] ∧ ys ≠ [] using assms(2) by blast
    hence (rotate-min xs = rotate-min ys) = (xs ≅ ys)
    using xs assms(1)
    by(simp add: singleton-list-cong-eq-iff norm-eq-iff-face-cong)
  } thus ?thesis by(simp add:inj-on-def)
qed

```

end

4 Graph

```

theory Graph
imports Rotation
begin

syntax

```

-UNION1 :: $p\text{trns} \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ $(\langle \exists \cup (\langle \text{unbreakable} \rangle -) / - \rangle [0, 10] 10)$
-INTER1 :: $p\text{trns} \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ $(\langle \exists \cap (\langle \text{unbreakable} \rangle -) / - \rangle [0, 10] 10)$
-UNION :: $p\text{trn} \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ $(\langle \exists \cup (\langle \text{unbreakable} \rangle - \in -) / - \rangle [0, 0, 10] 10)$
-INTER :: $p\text{trn} \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set}$ $(\langle \exists \cap (\langle \text{unbreakable} \rangle - \in -) / - \rangle [0, 0, 10] 10)$

4.1 Notation

type-synonym $\text{vertex} = \text{nat}$

consts

$\text{vertices} :: 'a \Rightarrow \text{vertex list}$
 $\text{edges} :: 'a \Rightarrow (\text{vertex} \times \text{vertex}) \text{ set } (\langle \mathcal{E} \rangle)$

abbreviation $\text{vertices-set} :: 'a \Rightarrow \text{vertex set } (\langle \mathcal{V} \rangle)$ **where**
 $\mathcal{V} f \equiv \text{set } (\text{vertices } f)$

4.2 Faces

We represent faces by (distinct) lists of vertices and a face type.

datatype $\text{facetype} = \text{Final} \mid \text{Nonfinal}$

datatype $\text{face} = \text{Face } (\text{vertex list}) \text{ facetype}$

consts $\text{final} :: 'a \Rightarrow \text{bool}$
consts $\text{type} :: 'a \Rightarrow \text{facetype}$

overloading

$\text{final-face} \equiv \text{final} :: \text{face} \Rightarrow \text{bool}$
 $\text{type-face} \equiv \text{type} :: \text{face} \Rightarrow \text{facetype}$
 $\text{vertices-face} \equiv \text{vertices} :: \text{face} \Rightarrow \text{vertex list}$
 $\text{cong-face} \equiv \text{pr-isomorphic} :: \text{face} \Rightarrow \text{face} \Rightarrow \text{bool}$

begin

primrec final-face **where**

$\text{final } (\text{Face } vs \ f) = (\text{case } f \text{ of } \text{Final} \Rightarrow \text{True} \mid \text{Nonfinal} \Rightarrow \text{False})$

primrec type-face **where**

$\text{type } (\text{Face } vs \ f) = f$

primrec vertices-face **where**

$\text{vertices } (\text{Face } vs \ f) = vs$

definition $\text{cong-face} :: \text{face} \Rightarrow \text{face} \Rightarrow \text{bool}$

where $(f_1 :: \text{face}) \cong f_2 \equiv \text{vertices } f_1 \cong \text{vertices } f_2$

end

The following operation makes a face final.

definition $setFinal :: face \Rightarrow face$ **where**
 $setFinal f \equiv Face (vertices f) Final$

The function $nextVertex$ (written $f \cdot v$) is based on $nextElem$, that returns the successor of an element in a list.

primrec $nextElem :: 'a list \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **where**
 $nextElem [] b x = b$
 $|nextElem (a\#as) b x =$
 $(if x=a then (case as of [] \Rightarrow b | (a'\#as') \Rightarrow a') else nextElem as b x)$

definition $nextVertex :: face \Rightarrow vertex \Rightarrow vertex$ **where**
 $f \cdot \equiv let vs = vertices f in nextElem vs (hd vs)$

$nextVertices$ is n -fold application of $nextvertex$.

definition $nextVertices :: face \Rightarrow nat \Rightarrow vertex \Rightarrow vertex$ **where**
 $f^n \cdot v \equiv (f \cdot \tilde{n} v)$

lemma $nextV2: f^2 \cdot v = f \cdot (f \cdot v)$
by (*simp add: nextVertices-def eval-nat-numeral*)

overloading $op\text{-}vertices \equiv Graph.op :: vertex list \Rightarrow vertex list$

begin

definition $(vs::vertex list)^{op} \equiv rev vs$

end

overloading $op\text{-}graph \equiv Graph.op :: face \Rightarrow face$

begin

primrec $op\text{-}graph$ **where** $(Face vs f)^{op} = Face (rev vs) f$

end

definition $prevVertex :: face \Rightarrow vertex \Rightarrow vertex$ **where**
 $f^{-1} \cdot v \equiv (let vs = vertices f in nextElem (rev vs) (last vs) v)$

abbreviation

$triangle :: face \Rightarrow bool$ **where**
 $triangle f == |vertices f| = 3$

4.3 Graphs

datatype $graph = Graph (face list) nat face list list nat list$

primrec $faces :: graph \Rightarrow face list$ **where**
 $faces (Graph fs n f h) = fs$

abbreviation

Faces :: *graph* \Rightarrow *face set* ($\langle \mathcal{F} \rangle$) **where**
 $\mathcal{F} \ g \equiv \text{set}(\text{faces } g)$

primrec *countVertices* :: *graph* \Rightarrow *nat* **where**
 $\text{countVertices } (\text{Graph } fs \ n \ f \ h) = n$

overloading

vertices-graph \equiv *vertices* :: *graph* \Rightarrow *vertex list*

begin

primrec *vertices-graph* **where** $\text{vertices } (\text{Graph } fs \ n \ f \ h) = [0 \ ..< \ n]$
end

lemma *vertices-graph*: $\text{vertices } g = [0 \ ..< \ \text{countVertices } g]$
by (*induct g*) *simp*

lemma *in-vertices-graph*:
 $v \in \text{set } (\text{vertices } g) = (v < \text{countVertices } g)$
by (*simp add:vertices-graph*)

lemma *len-vertices-graph*:
 $|\text{vertices } g| = \text{countVertices } g$
by (*simp add:vertices-graph*)

primrec *faceListAt* :: *graph* \Rightarrow *face list list* **where**
 $\text{faceListAt } (\text{Graph } fs \ n \ f \ h) = f$

definition *facesAt* :: *graph* \Rightarrow *vertex* \Rightarrow *face list* **where**
 ~~$\text{facesAt } g \ v \equiv \text{if } v \in \text{set } (\text{vertices } g) \ \text{then } \text{faceListAt } g \ ! \ v \ \text{else } []$~~

primrec *heights* :: *graph* \Rightarrow *nat list* **where**
 $\text{heights } (\text{Graph } fs \ n \ f \ h) = h$

definition *height* :: *graph* \Rightarrow *vertex* \Rightarrow *nat* **where**
 $\text{height } g \ v \equiv \text{heights } g \ ! \ v$

definition *graph* :: *nat* \Rightarrow *graph* **where**
 $\text{graph } n \equiv$
 $(\text{let } vs = [0 \ ..< \ n];$
 $fs = [\text{Face } vs \ \text{Final}, \ \text{Face } (\text{rev } vs) \ \text{Nonfinal}]$
 $\text{in } (\text{Graph } fs \ n \ (\text{replicate } n \ fs) \ (\text{replicate } n \ 0)))$

4.4 Operations on graphs

final graph, *final / nonfinal faces*

definition *finals* :: *graph* \Rightarrow *face list* **where**
 $\text{finals } g \equiv [f \ \leftarrow \ \text{faces } g. \ \text{final } f]$

definition *nonFinals* :: *graph* \Rightarrow *face list* **where**
 $\text{nonFinals } g \equiv [f \ \leftarrow \ \text{faces } g. \ \neg \ \text{final } f]$

definition *countNonFinals* :: *graph* \Rightarrow *nat* **where**

countNonFinals g \equiv $|nonFinals\ g|$

overloading *finalGraph* \equiv *final* :: *graph* \Rightarrow *bool*

begin

definition *finalGraph g* \equiv (*nonFinals g* = [])

end

lemma *finalGraph-faces[simp]*: *final g* \Longrightarrow *finals g* = *faces g*

by (*simp add: finalGraph-def finals-def nonFinals-def filter-compl1*)

lemma *finalGraph-face*: *final g* \Longrightarrow $f \in set\ (faces\ g) \Longrightarrow final\ f$

by (*simp only: finalGraph-faces[symmetric]*) (*simp add: finals-def*)

definition *finalVertex* :: *graph* \Rightarrow *vertex* \Rightarrow *bool* **where**

finalVertex g v \equiv $\forall f \in set(facesAt\ g\ v). final\ f$

lemma *finalVertex-final-face[dest]*:

finalVertex g v \Longrightarrow $f \in set\ (facesAt\ g\ v) \Longrightarrow final\ f$

by (*auto simp add: finalVertex-def*)

counting faces

definition *degree* :: *graph* \Rightarrow *vertex* \Rightarrow *nat* **where**

degree g v \equiv $|facesAt\ g\ v|$

definition *tri* :: *graph* \Rightarrow *vertex* \Rightarrow *nat* **where**

tri g v \equiv $||f \leftarrow facesAt\ g\ v. final\ f \wedge |vertices\ f| = 3||$

definition *quad* :: *graph* \Rightarrow *vertex* \Rightarrow *nat* **where**

quad g v \equiv $||f \leftarrow facesAt\ g\ v. final\ f \wedge |vertices\ f| = 4||$

definition *except* :: *graph* \Rightarrow *vertex* \Rightarrow *nat* **where**

except g v \equiv $||f \leftarrow facesAt\ g\ v. final\ f \wedge 5 \leq |vertices\ f| ||$

definition *vertextype* :: *graph* \Rightarrow *vertex* \Rightarrow *nat* \times *nat* \times *nat* **where**

vertextype g v \equiv (*tri g v*, *quad g v*, *except g v*)

lemma[*simp*]: $0 \leq tri\ g\ v$ **by** (*simp add: tri-def*)

lemma[*simp*]: $0 \leq quad\ g\ v$ **by** (*simp add: quad-def*)

lemma[*simp*]: $0 \leq except\ g\ v$ **by** (*simp add: except-def*)

definition *exceptionalVertex* :: *graph* \Rightarrow *vertex* \Rightarrow *bool* **where**

exceptionalVertex g v \equiv *except g v* \neq 0

definition *noExceptionals* :: *graph* \Rightarrow *vertex set* \Rightarrow *bool* **where**
noExceptionals *g V* \equiv $(\forall v \in V. \neg \text{exceptionalVertex } g v)$

An edge (a, b) is contained in face *f*, *b* is the successor of *a* in *f*.

overloading *edges-graph* \equiv *edges* :: *graph* \Rightarrow (*vertex* \times *vertex*) *set*
begin
definition \mathcal{E} (*g*::*graph*) \equiv $\bigcup_{f \in \mathcal{F} g} \text{edges } f$
end

definition *neighbors* :: *graph* \Rightarrow *vertex* \Rightarrow *vertex list* **where**
neighbors *g v* \equiv $[f \cdot v. f \leftarrow \text{facesAt } g v]$

4.5 Navigation in graphs

The function s' permutating the faces at a vertex, is implemented by the function *nextFace*

definition *nextFace* :: *graph* \times *vertex* \Rightarrow *face* \Rightarrow *face* **where**

definition *directedLength* :: *face* \Rightarrow *vertex* \Rightarrow *vertex* \Rightarrow *nat* **where**
directedLength *f a b* \equiv
if $a = b$ *then* 0 *else* $|\text{(between (vertices } f) a b)| + 1$

4.6 Code generator setup

definition *final-face* :: *face* \Rightarrow *bool* **where**
final-face-code-def: *final-face* = *final*
declare *final-face-code-def* [*symmetric*, *code-unfold*]

lemma *final-face-code* [*code*]:
final-face (*Face vs Final*) \longleftrightarrow *True*
final-face (*Face vs Nonfinal*) \longleftrightarrow *False*
by (*simp-all add*: *final-face-code-def*)

definition *final-graph* :: *graph* \Rightarrow *bool* **where**
final-graph-code-def: *final-graph* = *final*
declare *final-graph-code-def* [*symmetric*, *code-unfold*]

lemma *final-graph-code* [*code*]: *final-graph* *g* = *List.null* (*nonFinals* *g*)
unfolding *final-graph-code-def finalGraph-def null-def* ..

definition *vertices-face* :: *face* \Rightarrow *vertex list* **where**
vertices-face-code-def: *vertices-face* = *vertices*
declare *vertices-face-code-def* [*symmetric*, *code-unfold*]

lemma *vertices-face-code* [*code*]: *vertices-face* (*Face vs f*) = *vs*
unfolding *vertices-face-code-def* **by** *simp*

definition *vertices-graph* :: *graph* \Rightarrow *vertex list* **where**
vertices-graph-code-def: *vertices-graph* = *vertices*
declare *vertices-graph-code-def* [*symmetric*, *code-unfold*]

lemma *vertices-graph-code* [*code*]:
vertices-graph (*Graph fs n f h*) = $[0 \dots n]$
unfolding *vertices-graph-code-def* **by** *simp*

end

5 Syntax for operations on immutable arrays

theory *IArray-Syntax*
imports *Main HOL-Library.IArray*
begin

5.1 Tabulation

definition *tabulate* :: *nat* \Rightarrow (*nat* \Rightarrow 'a) \Rightarrow 'a *iarray*
where
tabulate *n f* = *IArray.of-fun f n*

definition *tabulate2* :: *nat* \Rightarrow *nat* \Rightarrow (*nat* \Rightarrow *nat* \Rightarrow 'a) \Rightarrow 'a *iarray iarray*
where
tabulate2 *m n f* = *IArray.of-fun* (λi . *IArray.of-fun* (*f i*) *n*) *m*

definition *tabulate3* :: *nat* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow
(*nat* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow 'a) \Rightarrow 'a *iarray iarray iarray* **where**
tabulate3 *l m n f* \equiv *IArray.of-fun* (λi . *IArray.of-fun* (λj . *IArray.of-fun* (λk . *f i j k*) *n*) *m*) *l*

syntax

-*tabulate* :: 'a \Rightarrow *pttrn* \Rightarrow *nat* \Rightarrow 'a *iarray* ($\langle\langle\llbracket - \cdot - \cdot - \rrbracket \rangle\rangle$)
-*tabulate2* :: 'a \Rightarrow *pttrn* \Rightarrow *nat* \Rightarrow *pttrn* \Rightarrow *nat* \Rightarrow 'a *iarray*
($\langle\langle\llbracket - \cdot - \cdot - , - \cdot - \rrbracket \rangle\rangle$)
-*tabulate3* :: 'a \Rightarrow *pttrn* \Rightarrow *nat* \Rightarrow *pttrn* \Rightarrow *nat* \Rightarrow *pttrn* \Rightarrow *nat* \Rightarrow 'a *iarray*
($\langle\langle\llbracket - \cdot - \cdot - , - \cdot - \cdot - \rrbracket \rangle\rangle$)

syntax-consts

-*tabulate* == *tabulate* **and**
-*tabulate2* == *tabulate2* **and**
-*tabulate3* == *tabulate3*

translations

$\llbracket f. x < n \rrbracket$ == *CONST tabulate* *n* (λx . *f*)
 $\llbracket f. x < m, y < n \rrbracket$ == *CONST tabulate2* *m n* ($\lambda x y$. *f*)
 $\llbracket f. x < l, y < m, z < n \rrbracket$ == *CONST tabulate3* *l m n* ($\lambda x y z$. *f*)

5.2 Access

abbreviation *sub1-syntax* :: 'a iarray \Rightarrow nat \Rightarrow 'a $\langle(-[-])\rangle$ [1000] 999)

where

$a[[n]] \equiv IArray.sub\ a\ n$

abbreviation *sub2-syntax* :: 'a iarray iarray \Rightarrow nat \Rightarrow nat \Rightarrow 'a $\langle(-[-,-])\rangle$ [1000] 999)

where

$as[[m, n]] \equiv IArray.sub\ (IArray.sub\ as\ m)\ n$

abbreviation *sub3-syntax* :: 'a iarray iarray iarray \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a $\langle(-[-,-,-])\rangle$ [1000] 999)

where

$as[[l, m, n]] \equiv IArray.sub\ (IArray.sub\ (IArray.sub\ as\ l)\ m)\ n$

examples: $[[0. i < 5]]$, $[[i. i < 5, j < 3]]$

end

6 Enumerating Patches

theory *Enumerator*

imports *Graph IArray-Syntax*

begin

Generates an Enumeration of lists. (See Kepler98, PartIII, section 8, p.11).

Used to construct all possible extensions of an unfinished outer face F with *outer* vertices by a new finished inner face with *inner* vertices, such a fixed edge e of the outer face is also contained in the inner face.

Label the vertices of F consecutively $0, \dots, outer - 1$, with 0 and $outer - 1$ the endpoints of e .

Generate all lists

$$[a_0, \dots, a_{inner-1}]$$

of length *inner*, such that $0 = a_0 \leq a_1 \dots a_{inner-2} < a_{inner-1}$. Every list represents an inner face, with vertices $v_0, \dots, v_{inner-1}$.

Construct the vertices $v_0, \dots, v_{inner-1}$ inductively: If $i = 1$ or $a_i \neq a_{i-1}$, we set v_i to the vertex with index a_i of F . But if $a_i = a_{i-1}$, we add a new vertex v_i to the planar map. The new face is to be drawn along the edge e over the face F .

As we run over all *inner* and all lists $[a_0, \dots, a_{inner-1}]$, we run over all possibilities from the finished face along the edge e inside F .

definition *enumBase* :: nat \Rightarrow nat list list **where**

$enumBase\ nmax \equiv [[i]. i \leftarrow [0 ..< Suc\ nmax]]$

definition *enumAppend* :: *nat* ⇒ *nat list list* ⇒ *nat list list* **where**
enumAppend nmax iss ≡ $\sqcup_{is \in iss} [is @ [n]. n \leftarrow [last\ is \ .. < \ Suc\ nmax]]$

definition *enumerator* :: *nat* ⇒ *nat* ⇒ *nat list list* **where**
enumerator inner outer ≡
 let *nmax* = *outer* - 2; *k* = *inner* - 3 in
 $[[0] @ is @ [outer - 1]. is \leftarrow (enumAppend\ nmax \ \sim k)\ (enumBase\ nmax)]$

definition *enumTab* :: *nat list list iarray iarray* **where**
enumTab ≡ $[[\ enumerator\ inner\ outer.\ inner < 9,\ outer < 9]]$

definition *enum* :: *nat* ⇒ *nat* ⇒ *nat list list* **where**
enum inner outer ≡ if *inner* < 9 ∧ *outer* < 9 then *enumTab*[[*inner*,*outer*]]
 else *enumerator inner outer*

primrec *hideDupsRec* :: '*a*' ⇒ '*a list*' ⇒ '*a option list*' **where**
hideDupsRec a [] = []
 | *hideDupsRec a (b#bs)* =
 (if *a* = *b* then *None* # *hideDupsRec b bs*
 else *Some b* # *hideDupsRec b bs*)

primrec *hideDups* :: '*a list*' ⇒ '*a option list*' **where**
hideDups [] = []
 | *hideDups (b#bs)* = *Some b* # *hideDupsRec b bs*

definition *indexToVertexList* :: *face* ⇒ *vertex* ⇒ *nat list* ⇒ *vertex option list*
where
indexToVertexList f v is ≡ *hideDups [f^k.v. k ← is]*

end

7 Subdividing a Face

theory *FaceDivision*
imports *Graph*
begin

definition *split-face* :: *face* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex list* ⇒ *face* × *face* **where**
split-face f ram₁ ram₂ newVs ≡ let *vs* = *vertices f*;
f₁ = [*ram₁*] @ *between vs ram₁ ram₂* @ [*ram₂*];
f₂ = [*ram₂*] @ *between vs ram₂ ram₁* @ [*ram₁*] in
 (*Face (rev newVs @ f₁) Nonfinal*,
Face (f₂ @ newVs) Nonfinal)

definition $\text{replacefacesAt} :: \text{nat list} \Rightarrow \text{face} \Rightarrow \text{face list} \Rightarrow \text{face list list} \Rightarrow \text{face list list}$ **where**
 $\text{replacefacesAt } ns \ f \ fs \ F \equiv \text{mapAt } ns \ (\text{replace } f \ fs) \ F$

definition $\text{makeFaceFinalFaceList} :: \text{face} \Rightarrow \text{face list} \Rightarrow \text{face list}$ **where**
 $\text{makeFaceFinalFaceList } f \ fs \equiv \text{replace } f \ [\text{setFinal } f] \ fs$

definition $\text{makeFaceFinal} :: \text{face} \Rightarrow \text{graph} \Rightarrow \text{graph}$ **where**
 $\text{makeFaceFinal } f \ g \equiv$
 $\text{Graph } (\text{makeFaceFinalFaceList } f \ (\text{faces } g))$
 $(\text{countVertices } g)$
 $[\text{makeFaceFinalFaceList } f \ fs. \ fs \leftarrow \text{faceListAt } g]$
 $(\text{heights } g)$

definition $\text{heightsNewVertices} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list}$ **where**
 $\text{heightsNewVertices } h_1 \ h_2 \ n \equiv [\text{min } (h_1 + i + 1) \ (h_2 + n - i). \ i \leftarrow [0 \ ..< \ n]]$

definition splitFace
 $:: \text{graph} \Rightarrow \text{vertex} \Rightarrow \text{vertex} \Rightarrow \text{face} \Rightarrow \text{vertex list} \Rightarrow \text{face} \times \text{face} \times \text{graph}$ **where**
 $\text{splitFace } g \ ram_1 \ ram_2 \ oldF \ newVs \equiv$
 $\text{let } fs = \text{faces } g;$
 $n = \text{countVertices } g;$
 $Fs = \text{faceListAt } g;$
 $h = \text{heights } g;$
 $vs_1 = \text{between } (\text{vertices } oldF) \ ram_1 \ ram_2;$
 $vs_2 = \text{between } (\text{vertices } oldF) \ ram_2 \ ram_1;$
 $(f_1, f_2) = \text{split-face } oldF \ ram_1 \ ram_2 \ newVs;$
 $Fs = \text{replacefacesAt } vs_1 \ oldF \ [f_1] \ Fs;$
 $Fs = \text{replacefacesAt } vs_2 \ oldF \ [f_2] \ Fs;$
 $Fs = \text{replacefacesAt } [ram_1] \ oldF \ [f_2, f_1] \ Fs;$
 $Fs = \text{replacefacesAt } [ram_2] \ oldF \ [f_1, f_2] \ Fs;$
 $Fs = Fs \ @ \ \text{replicate } |newVs| \ [f_1, f_2] \ \text{in}$
 $(f_1, f_2, \text{Graph } ((\text{replace } oldF \ [f_2] \ fs) @ [f_1])$
 $(n + |newVs|)$
 Fs
 $(h \ @ \ \text{heightsNewVertices } (h!ram_1)(h!ram_2) \ |newVs|))$

primrec $\text{subdivFace}' :: \text{graph} \Rightarrow \text{face} \Rightarrow \text{vertex} \Rightarrow \text{nat} \Rightarrow \text{vertex option list} \Rightarrow \text{graph}$
where
 $\text{subdivFace}' \ g \ f \ u \ n \ [] = \text{makeFaceFinal } f \ g$
 $| \text{subdivFace}' \ g \ f \ u \ n \ (vo\#\text{vos}) =$
 $(\text{case } vo \ \text{of } \text{None} \Rightarrow \text{subdivFace}' \ g \ f \ u \ (\text{Suc } n) \ vos$
 $| (\text{Some } v) \Rightarrow$
 $\text{if } f \cdot u = v \wedge n = 0$

```

then subdivFace' g f v 0 vos
else let ws = [countVertices g ..< countVertices g + n];
(f1, f2, g') = splitFace g u v f ws in
subdivFace' g' f2 v 0 vos)

```

definition *subdivFace* :: *graph* ⇒ *face* ⇒ *vertex option list* ⇒ *graph* **where**
subdivFace g f vos ≡ *subdivFace' g f (the(hd vos)) 0 (tl vos)*

end

8 Transitive Closure of Successor List Function

theory *RTranCl*

imports *Main*

begin

The reflexive transitive closure of a relation induced by a function of type '*a*' ⇒ '*a list*'. Instead of defining the closure again it would have been simpler to take $\{(x, y). y \in \text{set } (f x)\}^*$.

abbreviation (*input*)

```

in-set :: 'a ⇒ ('a ⇒ 'b list) ⇒ 'b ⇒ bool (⟦- [-]⟧ → - [55,0,55] 50) where
g [succs]→ g' == g' ∈ set (succs g)

```

inductive-set

```

RTranCl :: ('a ⇒ 'a list) ⇒ ('a * 'a) set
and in-RTranCl :: 'a ⇒ ('a ⇒ 'a list) ⇒ 'a ⇒ bool
(⟦- [-]⟧ → * - [55,0,55] 50)
for succs :: 'a ⇒ 'a list

```

where

```

g [succs]→* g' ≡ (g, g') ∈ RTranCl succs
| refl: g [succs]→* g
| succs: g [succs]→ g' ⇒ g' [succs]→* g'' ⇒ g [succs]→* g''

```

inductive-cases *RTranCl-elim*: (*h, h'*) : *RTranCl succs*

lemma *RTranCl-induct*:

```

(h, h') ∈ RTranCl succs ⇒
P h ⇒
(∧ g g'. g' ∈ set (succs g) ⇒ P g ⇒ P g') ⇒
P h'

```

proof –

```

assume s: ∧ g g'. g' ∈ set (succs g) ⇒ P g ⇒ P g'
assume (h, h') ∈ RTranCl succs P h
then show P h'
proof (induct rule: RTranCl.induct)
  fix g assume P g then show P g .
next

```

```

    fix g g' g''
    assume IH: P g'  $\implies$  P g''
    assume g'  $\in$  set(succs g) P g
    then have P g' by (rule s)
    then show P g'' by (rule IH)
  qed
qed

```

definition *invariant* :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a list) \Rightarrow bool **where**
invariant P succs $\equiv \forall g g'. g' \in \text{set}(\text{succs } g) \longrightarrow P g \longrightarrow P g'$

lemma *invariantE*:
invariant P succs $\implies g [\text{succs}] \rightarrow g' \implies P g \implies P g'$
by(*simp add:invariant-def*)

lemma *inv-subset*:
invariant P f $\implies (\bigwedge g. P g \implies \text{set}(f' g) \subseteq \text{set}(f g)) \implies \text{invariant } P f'$
by(*auto simp:invariant-def*)

lemma *RTranCl-inv*:
invariant P succs $\implies (g, g') \in \text{RTranCl succs} \implies P g \implies P g'$
by (*erule RTranCl-induct*)(*auto simp:invariant-def*)

lemma *RTranCl-subset2*:
assumes a: (s,g) : *RTranCl* f
shows ($\bigwedge g. (s,g) \in \text{RTranCl } f \implies \text{set}(f g) \subseteq \text{set}(h g)$) $\implies (s,g) : \text{RTranCl } h$
using a
proof (*induct rule: RTranCl.induct*)
 case refl **show** ?case **by**(*rule RTranCl.intros*)
next
 case succs **thus** ?case **by**(*blast intro: RTranCl.intros*)
qed

end

9 Plane Graph Enumeration

theory *Plane*
imports *Enumerator FaceDivision RTranCl*
begin

definition *maxGon* :: nat \Rightarrow nat **where**
maxGon p $\equiv p+3$

declare *maxGon-def* [*simp*]

definition $duplicateEdge :: graph \Rightarrow face \Rightarrow vertex \Rightarrow vertex \Rightarrow bool$ **where**
 $duplicateEdge\ g\ f\ a\ b \equiv$
 $2 \leq directedLength\ f\ a\ b \wedge 2 \leq directedLength\ f\ b\ a \wedge b \in set\ (neighbors\ g\ a)$

primrec $containsUnacceptableEdgeSnd ::$
 $(nat \Rightarrow nat \Rightarrow bool) \Rightarrow nat \Rightarrow nat\ list \Rightarrow bool$ **where**
 $containsUnacceptableEdgeSnd\ N\ v\ [] = False$ |
 $containsUnacceptableEdgeSnd\ N\ v\ (w\#\ ws) =$
 $(case\ ws\ of\ [] \Rightarrow False$
 $\quad | (w'\#\ ws') \Rightarrow if\ v < w \wedge w < w' \wedge N\ w\ w'\ then\ True$
 $\quad \quad \quad else\ containsUnacceptableEdgeSnd\ N\ w\ ws)$

primrec $containsUnacceptableEdge :: (nat \Rightarrow nat \Rightarrow bool) \Rightarrow nat\ list \Rightarrow bool$
where
 $containsUnacceptableEdge\ N\ [] = False$ |
 $containsUnacceptableEdge\ N\ (v\#\ vs) =$
 $(case\ vs\ of\ [] \Rightarrow False$
 $\quad | (w\#\ ws) \Rightarrow if\ v < w \wedge N\ v\ w\ then\ True$
 $\quad \quad \quad else\ containsUnacceptableEdgeSnd\ N\ v\ vs)$

definition $containsDuplicateEdge :: graph \Rightarrow face \Rightarrow vertex \Rightarrow nat\ list \Rightarrow bool$
where
 $containsDuplicateEdge\ g\ f\ v\ is \equiv$
 $containsUnacceptableEdge\ (\lambda i\ j.\ duplicateEdge\ g\ f\ (f^i \cdot v)\ (f^j \cdot v))\ is$

definition $containsDuplicateEdge' :: graph \Rightarrow face \Rightarrow vertex \Rightarrow nat\ list \Rightarrow bool$
where
 $containsDuplicateEdge'\ g\ f\ v\ is \equiv$
 $2 \leq |is| \wedge$
 $((\exists k < |is| - 2.\ let\ i0 = is!k; i1 = is!(k+1); i2 = is!(k+2)\ in$
 $\quad (duplicateEdge\ g\ f\ (f^{i1} \cdot v)\ (f^{i2} \cdot v)) \wedge (i0 < i1) \wedge (i1 < i2))$
 $\vee (let\ i0 = is!0; i1 = is!1\ in$
 $\quad (duplicateEdge\ g\ f\ (f^{i0} \cdot v)\ (f^{i1} \cdot v)) \wedge (i0 < i1))$

definition $generatePolygon :: nat \Rightarrow vertex \Rightarrow face \Rightarrow graph \Rightarrow graph\ list$ **where**
 $generatePolygon\ n\ v\ f\ g \equiv$
 $let\ enumeration = enumerator\ n\ |vertices\ f|;$
 $enumeration = [is \leftarrow enumeration.\ \neg\ containsDuplicateEdge\ g\ f\ v\ is];$
 $vertexLists = [indexToVertexList\ f\ v\ is.\ is \leftarrow enumeration]\ in$
 $[subdivFace\ g\ f\ vs.\ vs \leftarrow vertexLists]$

definition $next-plane0 :: nat \Rightarrow graph \Rightarrow graph\ list$ ($\langle next'-plane0 \cdot \rangle$) **where**
 $next-plane0_p\ g \equiv$
 $if\ final\ g\ then\ []$
 $else\ \bigsqcup_{f \in nonFinals\ g} \bigsqcup_{v \in vertices\ f} \bigsqcup_{i \in [3..<Suc(maxGon\ p)]}\ generatePolygon\ i$
 $v\ f\ g$

definition *Seed* :: nat ⇒ graph (⟨Seed-⟩) **where**
Seed_p ≡ graph(maxGon p)

lemma *Seed-not-final[iff]*: ¬ final (Seed p)
by(simp add:Seed-def graph-def finalGraph-def nonFinals-def)

definition *PlaneGraphs0* :: graph set **where**
PlaneGraphs0 ≡ ⋃ p. {g. Seed_p [next-plane0_p]→* g ∧ final g}

end

theory *Plane1*
imports *Plane*
begin

This is an optimized definition of plane graphs and the one we adopt as our point of reference. In every step only one fixed nonfinal face (the smallest one) and one edge in that face are picked.

definition *minimalFace* :: face list ⇒ face **where**
minimalFace ≡ minimal (length ∘ vertices)

definition *minimalVertex* :: graph ⇒ face ⇒ vertex **where**
minimalVertex g f ≡ minimal (height g) (vertices f)

definition *next-plane* :: nat ⇒ graph ⇒ graph list (⟨next'-plane-⟩) **where**
next-plane_p g ≡
 let fs = nonFinals g in
 if fs = [] then []
 else let f = minimalFace fs; v = minimalVertex g f in
 ⋃ i∈[3..*Suc*(maxGon p)] generatePolygon i v f g

definition *PlaneGraphsP* :: nat ⇒ graph set (⟨PlaneGraphs-⟩) **where**
PlaneGraphs_p ≡ {g. Seed_p [next-plane_p]→* g ∧ final g}

definition *PlaneGraphs* :: graph set **where**
PlaneGraphs ≡ ⋃ p. *PlaneGraphs_p*

end

10 Properties of Graph Utilities

theory *GraphProps*
imports *Graph*
begin

declare [[*linarith-neq-limit* = 3]]

lemma *final-setFinal*[iff]: *final(setFinal f)*
by (*simp add:setFinal-def*)

lemma *eq-setFinal-iff*[iff]: $(f = \text{setFinal } f) = \text{final } f$
proof (*induct f*)
case (*Face f t*)
then show *?case*
by (*cases t*) (*simp-all add: setFinal-def*)
qed

lemma *setFinal-eq-iff*[iff]: $(\text{setFinal } f = f) = \text{final } f$
by (*blast dest:sym intro:sym*)

lemma *distinct-vertices*[iff]: *distinct(vertices(g::graph))*
by(*induct g*) *simp*

10.1 *nextElem*

lemma *nextElem-append*[simp]:
 $y \notin \text{set } xs \implies \text{nextElem } (xs @ ys) \ d \ y = \text{nextElem } ys \ d \ y$
by(*induct xs*) *auto*

lemma *nextElem-cases*:
 $\text{nextElem } xs \ d \ x = y \implies$
 $x \notin \text{set } xs \wedge y = d \vee$
 $xs \neq [] \wedge x = \text{last } xs \wedge y = d \wedge x \notin \text{set}(\text{butlast } xs) \vee$
 $(\exists us \ vs. xs = us @ [x,y] @ vs \wedge x \notin \text{set } us)$
apply(*induct xs*)
apply *simp*
apply *simp*
apply(*split if-splits*)
apply(*simp split:list.splits*)
apply(*rule-tac x = [] in exI*)
apply *simp*
apply *simp*
apply(*erule disjE*)
apply *simp*
apply(*erule disjE*)
apply *clarsimp*
apply(*rule conjI*)
apply *clarsimp*
apply (*clarsimp*)
apply(*erule-tac x = a#us in allE*)
apply *simp*
done

lemma *nextElem-notin-butlast*[*rule-format,simp*]:
 $y \notin \text{set}(\text{butlast } xs) \longrightarrow \text{nextElem } xs \ x \ y = x$
by(*induct xs*) *auto*

lemma *nextElem-in*: $\text{nextElem } xs \ x \ y : \text{set}(x\#xs)$
apply (*induct xs*)
apply *simp*
apply *auto*
apply(*clarsimp split: list.splits*)
apply(*clarsimp split: list.splits*)
done

lemma *nextElem-notin*[*simp*]: $a \notin \text{set } as \implies \text{nextElem } as \ c \ a = c$
by(*erule nextElem-append*[**where** $ys = []$, *simplified*])

lemma *nextElem-last*[*simp*]: **assumes** *dist*: *distinct xs*
shows $\text{nextElem } xs \ c \ (\text{last } xs) = c$
proof *cases*
assume $xs = []$ **thus** *?thesis* **by** *simp*
next
let $?xs = \text{butlast } xs \ @ \ [\text{last } xs]$
assume $xs: xs \neq []$
with *dist* **have** *distinct ?xs* **by** *simp*
hence *notin*: $\text{last } xs \notin \text{set}(\text{butlast } xs)$ **by** *simp*
from *xs* **have** $\text{nextElem } xs \ c \ (\text{last } xs) = \text{nextElem } ?xs \ c \ (\text{last } xs)$ **by** *simp*
also from *notin* **have** $\dots = c$ **by** *simp*
finally show *?thesis* .
qed

lemma *prevElem-nextElem*:
assumes *dist*: *distinct xs* **and** *xxs*: $x : \text{set } xs$
shows $\text{nextElem } (\text{rev } xs) \ (\text{last } xs) \ (\text{nextElem } xs \ (\text{hd } xs) \ x) = x$
proof –
define x' **where** $x' = \text{nextElem } xs \ (\text{hd } xs) \ x$
hence *nE*: $\text{nextElem } xs \ (\text{hd } xs) \ x = x'$ **by** *simp*
have $xs \neq [] \wedge x = \text{last } xs \wedge x' = \text{hd } xs \vee (\exists us \ vs. xs = us \ @ \ [x, x'] \ @ \ vs)$
(is $?A \vee ?B$ **)**
using *nextElem-cases*[*OF nE*] *xxs* **by** *blast*
thus *?thesis*
proof
assume $?A$
thus *?thesis* **using** *dist* **by**(*clarsimp simp: neq-Nil-conv*)
next
assume $?B$
then obtain $us \ vs$ **where** [*simp*]: $xs = us \ @ \ [x, x'] \ @ \ vs$ **by** *blast*
thus *?thesis* **using** *dist* **by** *simp*
qed

qed

lemma *nextElem-prevElem*:
[[*distinct xs; x : set xs*]] \implies
 nextElem xs (hd xs) (nextElem (rev xs) (last xs) x) = x
apply(*cases xs = []*)
apply *simp*
using *prevElem-nextElem*[**where** *xs = rev xs and x=x*]
apply(*simp add:hd-rev last-rev*)
done

lemma *nextElem-nth*:
 $\wedge i.$ [[*distinct xs; i < length xs*]]
 \implies *nextElem xs z (xs!i) = (if length xs = i+1 then z else xs!(i+1))*
apply(*induct xs*) **apply** *simp*
apply(*case-tac i*)
apply(*simp split:list.split*)
apply *clarsimp*
done

10.2 *nextVertex*

lemma *nextVertex-in-face'*[*simp*]:
 vertices f \neq [] $\implies f \cdot v \in \mathcal{V} f$
proof –
 assume *f: vertices f \neq []*
 define *c* **where** *c = nextElem (vertices f) (hd (vertices f)) v*
 then have *nextElem (vertices f) (hd (vertices f)) v = c* **by auto**
 with f show ?thesis
 apply (*simp add: nextVertex-def*)
 apply (*drule-tac nextElem-cases*)
 apply(*fastforce simp:neq-Nil-conv*)
 done

qed

lemma *nextVertex-in-face*[*simp*]:
 v \in set (vertices f) $\implies f \cdot v \in \mathcal{V} f$
by (*auto intro: nextVertex-in-face'*)

lemma *nextVertex-prevVertex*[*simp*]:
 [[*distinct(vertices f); v \in $\mathcal{V} f$*]]
 $\implies f \cdot (f^{-1} \cdot v) = v$
by(*simp add:prevVertex-def nextVertex-def nextElem-prevElem*)

lemma *prevVertex-nextVertex*[*simp*]:
 [[*distinct(vertices f); v \in $\mathcal{V} f$*]]
 $\implies f^{-1} \cdot (f \cdot v) = v$

by(*simp add:prevVertex-def nextVertex-def prevElem-nextElem*)

lemma *prevVertex-in-face*[*simp*]:

$v \in \mathcal{V} f \implies f^{-1} \cdot v \in \mathcal{V} f$
apply(*cases vertices f = []*)
apply *simp*
using *nextElem-in*[*of rev (vertices f) (last (vertices f)) v*]
apply (*auto simp add: prevVertex-def*)
done

lemma *nextVertex-nth*:

$\llbracket \text{distinct}(\text{vertices } f); i < |\text{vertices } f| \rrbracket \implies$
 $f \cdot (\text{vertices } f ! i) = \text{vertices } f ! ((i+1) \bmod |\text{vertices } f|)$
apply(*cases vertices f = []*) **apply** *simp*
apply(*simp add:nextVertex-def nextElem-nth hd-conv-nth*)
done

10.3 \mathcal{E}

lemma *edges-face-eq*:

$((a,b) \in \mathcal{E} (f::\text{face})) = ((f \cdot a = b) \wedge a \in \mathcal{V} f)$
by (*auto simp add: edges-face-def*)

lemma *edges-setFinal*[*simp*]: $\mathcal{E}(\text{setFinal } f) = \mathcal{E} f$

by(*induct f*)(*simp add:setFinal-def edges-face-def nextVertex-def*)

lemma *in-edges-in-vertices*:

$(x,y) \in \mathcal{E}(f::\text{face}) \implies x \in \mathcal{V} f \wedge y \in \mathcal{V} f$
apply(*simp add:edges-face-eq nextVertex-def*)
apply(*cut-tac xs= vertices f and x= hd(vertices f) and y=x in nextElem-in*)
apply(*cases vertices f*)
apply(*auto*)
done

lemma *vertices-conv-Union-edges*:

$\mathcal{V}(f::\text{face}) = (\bigcup (a,b) \in \mathcal{E} f. \{a\})$
apply(*induct f*)
apply(*simp add:vertices-face-def edges-face-def*)
apply *blast*
done

lemma *nextVertex-in-edges*: $v \in \mathcal{V} f \implies (v, f \cdot v) \in \text{edges } f$

by(*auto simp:edges-face-def*)

lemma *prevVertex-in-edges*:

$\llbracket \text{distinct}(\text{vertices } f); v \in \mathcal{V} f \rrbracket \implies (f^{-1} \cdot v, v) \in \text{edges } f$

by(*simp add:edges-face-eq*)

10.4 Triangles

lemma *vertices-triangle*:

$|vertices\ f| = 3 \implies a \in \mathcal{V}\ f \implies$
 $distinct\ (vertices\ f) \implies$
 $\mathcal{V}\ f = \{a, f \cdot a, f \cdot (f \cdot a)\}$

proof –

assume $|vertices\ f| = 3$

then obtain $a1\ a2\ a3$ **where** $vertices\ f = [a1, a2, a3]$

by (*auto dest!: length3D*)

moreover assume $a \in \mathcal{V}\ f$

moreover assume $distinct\ (vertices\ f)$

ultimately show *?thesis*

by (*simp, elim disjE*) (*auto simp add: nextVertex-def*)

qed

lemma *tri-next3-id*:

$|vertices\ f| = 3 \implies distinct(vertices\ f) \implies v \in \mathcal{V}\ f$
 $\implies f \cdot (f \cdot (f \cdot v)) = v$

apply (*subgoal-tac* $\forall (i::nat) < 3. (((((i+1) mod 3)+1) mod 3)+1) mod 3 = i$)

apply (*clarsimp simp:in-set-conv-nth nextVertex-nth*)

apply (*presburger*)

done

lemma *triangle-nextVertex-prevVertex*:

$|vertices\ f| = 3 \implies a \in \mathcal{V}\ f \implies$
 $distinct\ (vertices\ f) \implies$
 $f \cdot (f \cdot a) = f^{-1} \cdot a$

proof –

assume $|vertices\ f| = 3$

then obtain $a1\ a2\ a3$ **where** $vertices\ f = [a1, a2, a3]$

by (*auto dest!:length3D*)

moreover assume $a \in \mathcal{V}\ f$

moreover assume $distinct\ (vertices\ f)$

ultimately show *?thesis*

by (*simp, elim disjE*) (*auto simp add: nextVertex-def prevVertex-def*)

qed

10.5 Quadrilaterals

lemma *vertices-quad*:

$|vertices\ f| = 4 \implies a \in \mathcal{V}\ f \implies$
 $distinct\ (vertices\ f) \implies$
 $\mathcal{V}\ f = \{a, f \cdot a, f \cdot (f \cdot a), f \cdot (f \cdot (f \cdot a))\}$

proof –

assume $|vertices\ f| = 4$

then obtain $a1\ a2\ a3\ a4$ **where** $vertices\ f = [a1, a2, a3, a4]$
 by (auto dest!: length₄D)
moreover assume $a \in \mathcal{V}\ f$
moreover assume $distinct\ (vertices\ f)$
ultimately show ?thesis
 by (simp, elim disjE) (auto simp add: nextVertex-def)
qed

lemma *quad-next₄-id*:
 $\llbracket |vertices\ f| = 4; distinct(vertices\ f); v \in \mathcal{V}\ f \rrbracket \implies$
 $f \cdot (f \cdot (f \cdot (f \cdot v))) = v$
apply(subgoal-tac $\forall (i::nat) < 4.$
 $(((((i+1) \bmod 4)+1) \bmod 4)+1) \bmod 4 = i)$)
apply(clarsimp simp:in-set-conv-nth nextVertex-nth)
apply(presburger)
done

lemma *quad-nextVertex-prevVertex*:
 $|vertices\ f| = 4 \implies a \in \mathcal{V}\ f \implies distinct\ (vertices\ f) \implies$
 $f \cdot (f \cdot (f \cdot a)) = f^{-1} \cdot a$
proof –
assume $|vertices\ f| = 4$
then obtain $a1\ a2\ a3\ a4$ **where** $vertices\ f = [a1, a2, a3, a4]$
 by (auto dest!: length₄D)
moreover assume $a \in \mathcal{V}\ f$
moreover assume $distinct\ (vertices\ f)$
ultimately show ?thesis
 by (auto) (auto simp add: nextVertex-def prevVertex-def)
qed

lemma *len-faces-sum*: $|faces\ g| = |finals\ g| + |nonFinals\ g|$
by(simp add:finals-def nonFinals-def sum-length-filter-compl)

lemma *graph-max-final-ex*:
 $\exists f \in set\ (finals\ (graph\ n)). |vertices\ f| = n$
proof (induct n)
case 0 **then show** ?case **by** (simp add: graph-def finals-def)
next
case (Suc n) **then show** ?case
 by (simp add: graph-def finals-def)
qed

10.6 No loops

lemma *distinct-no-loop2*:

```

[[ distinct(vertices f); v ∈ V f; u ∈ V f; u ≠ v ]] ⇒ f · v ≠ v
apply(frule split-list[of v])
apply(clarsimp simp: nextVertex-def neq-Nil-conv hd-append
  split:list.splits if-split-asm)
done

```

```

lemma distinct-no-loop1:
[[ distinct(vertices f); v ∈ V f; |vertices f| > 1 ]] ⇒ f · v ≠ v
apply(subgoal-tac ∃ u ∈ V f. u ≠ v)
apply(blast dest:distinct-no-loop2)
apply(cases vertices f) apply simp
apply(rename-tac a as)
apply (clarsimp simp: neq-Nil-conv)
done

```

10.7 between

```

lemma between-front[simp]:
v ∉ set us ⇒ between (u # us @ v # vs) u v = us
by(simp add:between-def split-def)

```

```

lemma between-back:
[[ v ∉ set us; u ∉ set vs; v ≠ u ]] ⇒ between (v # vs @ u # us) u v = us
by(simp add:between-def split-def)

```

```

lemma next-between:
[[ distinct(vertices f); v ∈ V f; u ∈ V f; f · v ≠ u ]]
⇒ f · v ∈ set(between (vertices f) v u)
apply(frule split-list[of u])
apply(clarsimp)
apply(erule disjE)
apply(clarsimp simp:set-between-id nextVertex-def hd-append split:list.split)
apply(erule disjE)
apply(frule split-list[of v])
apply(clarsimp simp: between-def split-def nextVertex-def split:list.split)
apply(clarsimp simp:append-eq-Cons-conv)
apply(frule split-list[of v])
apply(clarsimp simp: between-def split-def nextVertex-def split:list.split)
apply(clarsimp simp: hd-append)
done

```

```

lemma next-between2:
[[ distinct(vertices f); v ∈ V f; u ∈ V f; u ≠ v ]] ⇒
v ∈ set(between (vertices f) u (f · v))
apply(frule split-list[of u])
apply(clarsimp)
apply(erule disjE)

```

```

apply(clarsimp simp: nextVertex-def hd-append split:list.split)
apply(rule conjI)
  apply(clarsimp)
apply(frule split-list[of v])
apply(clarsimp simp: between-def split-def split:list.split)
apply(fastforce simp: append-eq-Cons-conv)
apply(frule split-list[of v])
apply(clarsimp simp: between-def split-def nextVertex-def split:list.splits)
apply(clarsimp simp: hd-append)
apply(erule disjE)
  apply(clarsimp)
apply(frule split-list)
apply(fastforce)
done

```

```

lemma between-next-empty:
   $\text{distinct}(\text{vertices } f) \implies \text{between } (\text{vertices } f) \ v \ (f \cdot v) = []$ 
apply(cases v ∈ V f)
  apply(frule split-list)
  apply(clarsimp simp: between-def split-def nextVertex-def
    neq-Nil-conv hd-append split:list.split)
  apply(clarsimp simp: between-def split-def nextVertex-def)
apply(cases vertices f)
  apply simp
apply simp
done

```

```

lemma unroll-between-next2:
   $\llbracket \text{distinct}(\text{vertices } f); u \in \mathcal{V} f; v \in \mathcal{V} f; u \neq v \rrbracket \implies$ 
   $\text{between } (\text{vertices } f) \ u \ (f \cdot v) = \text{between } (\text{vertices } f) \ u \ v \ @ \ [v]$ 
using split-between[OF - - - next-between2]
by (simp add: between-next-empty split:if-split-asm)

```

```

lemma nextVertex-eq-lemma:
   $\llbracket \text{distinct}(\text{vertices } f); x \in \mathcal{V} f; y \in \mathcal{V} f; x \neq y;$ 
   $v \in \text{set}(x \# \text{between } (\text{vertices } f) \ x \ y) \rrbracket \implies$ 
   $f \cdot v = \text{nextElem } (x \# \text{between } (\text{vertices } f) \ x \ y \ @ \ [y]) \ z \ v$ 
apply(drule split-list[of x])
apply(simp add: nextVertex-def)
apply(erule disjE)
apply(clarsimp)
apply(erule disjE)
apply(drule split-list)
apply(clarsimp simp add: between-def split-def hd-append split:list.split)
apply(fastforce simp: append-eq-Cons-conv)

```

```

apply(drule split-list)
apply(clarsimp simp add:between-def split-def hd-append split:list.split)
apply(fastforce simp:append-eq-Cons-conv)
apply(clarsimp)
apply(erule disjE)
apply(drule split-list[of y])
apply(clarsimp simp:between-def split-def)
apply(erule disjE)
  apply(drule split-list[of v])
  apply(fastforce simp: hd-append neq-Nil-conv split:list.split)
apply(drule split-list[of v])
apply(clarsimp)
apply(clarsimp simp: hd-append split:list.split)
apply(fastforce simp:append-eq-Cons-conv)
apply(drule split-list[of y])
apply(clarsimp simp:between-def split-def)
apply(drule split-list[of v])
apply(clarsimp)
apply(clarsimp simp: hd-append split:list.split)
apply(clarsimp simp:append-eq-Cons-conv)
apply(fastforce simp: hd-append neq-Nil-conv split:list.split)
done

end

```

11 Properties of Patch Enumeration

```

theory EnumeratorProps
imports Enumerator GraphProps
begin

```

```

lemma length-hideDupsRec[simp]:  $\bigwedge x. \text{length}(\text{hideDupsRec } x \text{ } xs) = \text{length } xs$ 
by(induct xs) auto

```

```

lemma length-hideDups[simp]:  $\text{length}(\text{hideDups } xs) = \text{length } xs$ 
by(cases xs) simp-all

```

```

lemma length-indexToVertexList[simp]:
   $\text{length}(\text{indexToVertexList } x \text{ } y \text{ } xs) = \text{length } xs$ 
by(simp add:indexToVertexList-def)

```

```

definition increasing :: ('a::linorder) list  $\Rightarrow$  bool where
  increasing ls  $\equiv \forall x \ y \ \text{as} \ \text{bs}. \text{ls} = \text{as} @ x \# y \# \text{bs} \longrightarrow x \leq y$ 

```

```

lemma increasing1:  $\bigwedge \text{as } x. \text{increasing } \text{ls} \Longrightarrow \text{ls} = \text{as} @ x \# \text{cs} @ y \# \text{bs} \Longrightarrow x \leq y$ 

```

```

proof (induct cs)
  case Nil then show ?case
    by (auto simp: increasing-def)
next
  case (Cons c cs) then show ?case
    apply (subgoal-tac c ≤ y)
    apply (force simp: increasing-def)
    apply (rule-tac Cons) by simp-all
qed

```

lemma *increasing2*: $\text{increasing } (as@bs) \implies x \in \text{set } as \implies y \in \text{set } bs \implies x \leq y$
proof –

```

  assume n:increasing (as@bs) and x:x ∈ set as and y: y ∈ set bs
  from x obtain as' as'' where as: as = as' @ x # as'' by (auto simp: in-set-conv-decomp)
  from y obtain bs' bs'' where bs: bs = bs' @ y # bs'' by (auto simp: in-set-conv-decomp)
  from n as bs show ?thesis
  apply (auto intro!: increasing1)
  apply (subgoal-tac as' @ x # as'' @ bs' @ y # bs'' = as' @ x # (as'' @ bs') @
y # bs'')
  by (assumption) auto
qed

```

lemma *increasing3*: $\forall as\ bs. (ls = as @ bs \longrightarrow (\forall x \in \text{set } as. \forall y \in \text{set } bs. x \leq y)) \implies \text{increasing } (ls)$

apply (simp add: increasing-def) **apply** safe

proof –

```

  fix as bs x y
  assume p:  $\forall asa\ bsa. as @ x \# y \# bs = asa @ bsa \longrightarrow (\forall x \in \text{set } asa. \forall y \in \text{set } bsa. x \leq y)$ 
  then have p':  $\bigwedge asa\ bsa. as @ x \# y \# bs = asa @ bsa \implies (\forall x \in \text{set } asa. \forall y \in \text{set } bsa. x \leq y)$  by auto
  then have  $(\forall x \in \text{set } (as @ [x]). \forall y \in \text{set } (y \# bs). x \leq y)$  by (rule-tac p') auto
  then show  $x \leq y$  by (auto simp: increasing-def)
qed

```

lemma *increasing4*: $\text{increasing } (as@bs) \implies \text{increasing } as$

apply (simp add: increasing-def) **apply** safe **by** auto

lemma *increasing5*: $\text{increasing } (as@bs) \implies \text{increasing } bs$

proof –

```

  assume nd: increasing (as@bs)
  then have r:  $\bigwedge x\ y\ asa\ bsa. (\exists asa\ bsa. as @ bs = asa @ x \# y \# bsa) \implies x \leq y$  by (auto simp: increasing-def)
  show ?thesis apply (clarsimp simp add: increasing-def)
  apply (rule-tac r)
  apply (rule-tac x=as @ - in exI)
  apply auto
  done
qed

```


lemma *enumBase-length*: $ls \in \text{set } (\text{enumBase } nmax) \implies \text{length } ls = 1$
by (*auto simp: enumBase-def*)

lemma *enumBase-bound*: $\forall y \in \text{set } (\text{enumBase } nmax). \forall z \in \text{set } y. z \leq nmax$
by (*auto simp: enumBase-def*)

lemmas *enumBase-simps* = *enumBase-length enumBase-bound*

lemma *enumAppend-bound*: $ls \in \text{set } ((\text{enumAppend } nmax) lss) \implies$
 $\forall y \in \text{set } lss. \forall z \in \text{set } y. z \leq nmax \implies x \in \text{set } ls \implies x \leq nmax$
by (*auto simp add: enumAppend-def split: if-split-asm*)

lemma *enumAppend-bound-rec*: $ls \in \text{set } (((\text{enumAppend } nmax) \sim n) lss) \implies$
 $\forall y \in \text{set } lss. \forall z \in \text{set } y. z \leq nmax \implies x \in \text{set } ls \implies x \leq nmax$

proof –

assume *ls*: $ls \in \text{set } ((\text{enumAppend } nmax \sim n) lss)$ **and** *lss*: $\forall y \in \text{set } lss. \forall z \in \text{set } y. z \leq nmax$ **and** *x*: $x \in \text{set } ls$

have *ind*: $\bigwedge lss. \forall y \in \text{set } lss. \forall z \in \text{set } y. z \leq nmax \implies \forall y \in \text{set } (((\text{enumAppend } nmax) \sim n) lss). \forall z \in \text{set } y. z \leq nmax$

proof (*induct n*)

case 0 **then show** ?*case* **by** *auto*

next

case (*Suc n*) **show** ?*case* **apply** (*intro ballI*) **apply** (*rule enumAppend-bound*)
by (*auto intro!: Suc*)

qed

with *lss* **have** $\forall y \in \text{set } (((\text{enumAppend } nmax) \sim n) lss). \forall z \in \text{set } y. z \leq nmax$ **apply** (*rule-tac ind*) .

with *ls x* **show** ?*thesis* **by** *auto*

qed

lemma *enumAppend-increase-rec*:

$\bigwedge m \text{ as } bs. ls \in \text{set } (((\text{enumAppend } nmax) \sim m) (\text{enumBase } nmax)) \implies$
 $as @ bs = ls \implies \forall x \in \text{set } as. \forall y \in \text{set } bs. x \leq y$

apply (*induct ls rule: rev-induct*) **apply** *force* **apply** *auto* **apply** (*case-tac m*)

apply *simp* **apply** (*drule-tac enumBase-length*)

apply (*case-tac as*) **apply** *simp-all*

proof –

fix *x xs m as bs xa xb n*

assume *ih*: $\bigwedge m \text{ as } bs.$

$\llbracket xs \in \text{set } ((\text{enumAppend } nmax \rightsquigarrow m) (\text{enumBase } nmax)); as @ bs = xs \rrbracket$
 $\implies \forall x \in \text{set } as. \forall xa \in \text{set } bs. x \leq xa$
and $xs:xs @ [x] \in \text{set } (\text{enumAppend } nmax ((\text{enumAppend } nmax \rightsquigarrow n) (\text{enumBase } nmax)))$
and $asbs: as @ bs = xs @ [x]$ **and** $xa:xa \in \text{set } as$ **and** $xb:xb \in \text{set } bs$ **and** $m:$
 $m = \text{Suc } n$
from ih **have** $ih2: \bigwedge as bs x y. \llbracket xs \in \text{set } ((\text{enumAppend } nmax \rightsquigarrow n) (\text{enumBase } nmax)); as @ bs = xs; x \in \text{set } as; y \in \text{set } bs \rrbracket$
 $\implies x \leq y$ **by** *auto*

from xb **have** $bs \neq []$ **by** *auto*
then obtain $bs' b$ **where** $bs': bs = bs' @ [b]$ **apply** (*cases rule: rev-exhaust*) **by**
auto
with $asbs$ **have** $beq:b = x$ **by** *auto*
from $bs' asbs$ **have** $xs': as @ bs' = xs$ **by** *auto*
with xs **have** $xa \leq x$
proof (*cases xs rule: rev-exhaust*)
case *Nil* **with** $xa xs'$ **show** $?thesis$ **by** *auto*
next
case (*snoc ys y*)
have $xa \leq y$
proof (*cases xa = y*)
case *True* **then show** $?thesis$ **by** *auto*
next
case *False*
from $xa xs'$ **have** $xa \in \text{set } xs$ **by** *auto*
with *False snoc* **have** $xa \in \text{set } ys$ **by** *auto*
with $xs snoc$ **show** $?thesis$
apply (*rule-tac ih2*)
by (*auto simp: enumAppend-def*)
qed
with $xs snoc$ **show** $xa \leq x$ **by** (*auto simp: enumAppend-def split:if-split-asm*)
qed
then show $xa \leq xb$ **apply** (*cases xb = b*) **apply** (*simp add: beq*)
proof (*rule-tac ih2*)
from xs
show $xs \in \text{set } ((\text{enumAppend } nmax \rightsquigarrow n) (\text{enumBase } nmax))$
by (*auto simp: enumAppend-def*)
next
from xs' **show** $as @ bs' = xs$ **by** *auto*
next
from xa **show** $xa \in \text{set } as$ **by** *auto*
next
assume $xb \neq b$
with $xb bs'$ **show** $xb \in \text{set } bs'$ **by** *auto*
qed
qed

lemma *enumAppend-length1*: $\bigwedge ls. ls \in \text{set } ((\text{enumAppend } nmax \rightsquigarrow n) lss) \implies$

$(\forall l \in \text{set } lss. |l| = k) \implies |ls| = k + n$
apply (*induct n*)
apply *simp*
by (*auto simp add: enumAppend-def split: if-split-asm*)

lemma *enumAppend-length2*: $\bigwedge ls. ls \in \text{set } ((\text{enumAppend } nmax \ \sim n) \ lss) \implies$
 $(\bigwedge l. l \in \text{set } lss \implies |l| = k) \implies K = k + n \implies |ls| = K$
by (*auto simp add: enumAppend-length1*)

lemma *enum-enumerator*:
 $\text{enum } i \ j = \text{enumerator } i \ j$
by (*simp add: enum-def enumTab-def tabulate2-def tabulate-def*)

lemma *enumerator-hd*: $ls \in \text{set } (\text{enumerator } m \ n) \implies \text{hd } ls = 0$
by (*auto simp: enumerator-def split: if-split-asm*)

lemma *enumerator-last*: $ls \in \text{set } (\text{enumerator } m \ n) \implies \text{last } ls = (n - 1)$
by (*auto simp: enumerator-def split: if-split-asm*)

lemma *enumerator-length*: $ls \in \text{set } (\text{enumerator } m \ n) \implies 2 \leq \text{length } ls$
by (*auto simp: enumerator-def split: if-split-asm*)

lemmas *set-enumerator-simps* = *enumerator-hd enumerator-last enumerator-length*

lemma *enumerator-not-empty[dest]*: $ls \in \text{set } (\text{enumerator } m \ n) \implies ls \neq []$
apply (*subgoal-tac 2 ≤ length ls*) **apply** *force* **by** (*rule enumerator-length*)

lemma *enumerator-length2*: $ls \in \text{set } (\text{enumerator } m \ n) \implies 2 < m \implies \text{length } ls = m$

proof –
assume $ls: ls \in \text{set } (\text{enumerator } m \ n)$ **and** $m: 2 < m$
define k **where** $k = m - 3$
with m **have** $k: m = k + 3$ **by** *arith*
with ls **have** $ls \in \text{set } (\text{enumerator } (k+3) \ n)$ **by** *auto*
then **have** $\text{length } ls = k + 3$
apply (*auto simp: enumerator-def enumBase-def*)
apply (*erule enumAppend-length2*) **by** *auto*
with k **show** *?thesis* **by** *simp*
qed

lemma *enumerator-bound*: $ls \in \text{set } (\text{enumerator } m \ nmax) \implies$
 $0 < nmax \implies x \in \text{set } ls \implies x < nmax$
apply (*auto simp: enumerator-def split: if-split-asm*)

apply (subgoal-tac $x \leq nmax - 2$) **apply** arith
apply (rule-tac enumAppend-bound-rec) **by**(auto simp:enumBase-simps)

lemma enumerator-bound2: $ls \in set (enumerator\ m\ nmax) \implies 1 < nmax \implies x \in set (butlast\ ls) \implies x < nmax - Suc\ 0$
apply (auto simp: enumerator-def split: if-split-asm)
apply (subgoal-tac $x \leq (nmax - 2)$) **apply** arith
apply (rule-tac enumAppend-bound-rec) **by**(auto simp:enumBase-simps)

lemma enumerator-bound3: $ls \in set (enumerator\ m\ nmax) \implies 1 < nmax \implies last (butlast\ ls) < nmax - Suc\ 0$
apply (case-tac ls rule: rev-exhaust) **apply** force
apply (rule-tac enumerator-bound2) **apply** assumption
apply auto
apply (case-tac ys rule: rev-exhaust) **apply** simp
apply (subgoal-tac $2 \leq length (ys @ [y])$) **apply** simp
apply (rule-tac enumerator-length) **by** auto

lemma enumerator-increase: $\bigwedge as\ bs.\ ls \in set (enumerator\ m\ nmax) \implies as @ bs = ls \implies \forall x \in set\ as.\ \forall y \in set\ bs.\ x \leq y$
apply (auto simp: enumerator-def del: Nat.diff-is-0-eq' split: if-split-asm intro: enumAppend-increase-rec)
apply (case-tac as) **apply** simp **apply** simp
apply (case-tac bs rule: rev-exhaust) **apply** simp **apply** simp **apply** auto
apply (drule-tac enumAppend-bound-rec) **apply** (auto simp:enumBase-simps)
by (auto dest!: enumAppend-increase-rec)

lemma enumerator-increasing: $ls \in set (enumerator\ m\ nmax) \implies increasing\ ls$
apply (rule increasing3)
by (auto dest: enumerator-increase)

definition incrIndexList :: $nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**
 $incrIndexList\ ls\ m\ nmax \equiv$
 $1 < m \wedge 1 < nmax \wedge$
 $hd\ ls = 0 \wedge last\ ls = (nmax - 1) \wedge length\ ls = m$
 $\wedge last (butlast\ ls) < last\ ls \wedge increasing\ ls$

lemma incrIndexList-1lem[simp]: $incrIndexList\ ls\ m\ nmax \implies Suc\ 0 < m$
by (unfold incrIndexList-def) simp

lemma incrIndexList-1len[simp]: $incrIndexList\ ls\ m\ nmax \implies Suc\ 0 < nmax$
by (unfold incrIndexList-def) simp

lemma incrIndexList-help2[simp]: $incrIndexList\ ls\ m\ nmax \implies hd\ ls = 0$
by (unfold incrIndexList-def) simp

lemma incrIndexList-help21[simp]: $incrIndexList (l \# ls) m nmax \implies l = 0$
by (auto dest: incrIndexList-help2)

lemma *incrIndexList-help3*[simp]: *incrIndexList ls m nmax* \implies *last ls* = (*nmax* - (*Suc 0*))
by (*unfold incrIndexList-def*) *simp*

lemma *incrIndexList-help4*[simp]: *incrIndexList ls m nmax* \implies *length ls* = *m*
by (*unfold incrIndexList-def*) *simp*

lemma *incrIndexList-help5*[intro]: *incrIndexList ls m nmax* \implies *last (butlast ls)* < *nmax - Suc 0*
by (*unfold incrIndexList-def*) *auto*

lemma *incrIndexList-help6*[simp]: *incrIndexList ls m nmax* \implies *increasing ls*
by (*unfold incrIndexList-def*) *simp*

lemma *incrIndexList-help7*[simp]: *incrIndexList ls m nmax* \implies *ls* \neq []
apply (*subgoal-tac length ls* \neq 0) **apply** *force*
apply *simp*
apply (*subgoal-tac 1 < m*) **apply** *arith* **apply** *force* **done**

lemma *incrIndexList-help71*[simp]: \neg *incrIndexList [] m nmax*
by (*auto dest: incrIndexList-help7*)

lemma *incrIndexList-help8*[simp]: *incrIndexList ls m nmax* \implies *butlast ls* \neq []
proof (*rule ccontr*)

assume *props: incrIndexList ls m nmax* **and** *butl: \neg butlast ls \neq []*
then have *ls* \neq [] **by** *auto*
then have *ls'*: *ls* = (*butlast ls*) @ [*last ls*] **by** *auto*
define *l* **where** *l* = *last ls*
with *butl ls'* **have** *ls* = [*l*] **by** *auto*
then have *length ls* = 1 **by** *auto*
with *props* **have** *m* = 1 **by** *auto*
with *props* **show** *False* **by** (*auto dest: incrIndexList-1lem*)
qed

lemma *incrIndexList-help81*[simp]: \neg *incrIndexList [l] m nmax*
by (*auto dest: incrIndexList-help8*)

lemma *incrIndexList-help9*[intro]: (*incrIndexList ls m nmax*) \implies *x* \in *set (butlast ls)* \implies *x* \leq *nmax - 2*

proof -
assume *props: (incrIndexList ls m nmax)* **and** *x: x* \in *set (butlast ls)*
then have *last (butlast ls)* < *last ls* **by** *auto*
with *props* **have** *last (butlast ls)* < *nmax - 1* **by** *auto*
then have *leq: last (butlast ls)* \leq *nmax - 2* **by** *arith*
from *props* **have** *ls* \neq [] **by** *auto*
then have *ls1: ls* = *butlast ls* @ [*last ls*] **by** *auto*
define *ls'* **where** *ls'* = *butlast (butlast ls)*
define *last2* **where** *last2* = *last (butlast ls)*

```

define last1 where last1 = last ls
from props have butlast ls ≠ [] by auto
with ls'-def last2-def have bls: butlast ls = ls' @ [last2] by auto
with last1-def ls1 props have ls3: ls = ls' @ [last2] @ [last1] by auto
from props have increasing ls by auto
with ls3 have increasing: increasing (ls' @ ([last2] @ [last1])) by auto
then have x ∈ set ls' ⇒ x ≤ last2 by (auto intro: increasing2)
then have x ∈ set (ls' @ [last2]) ⇒ x ≤ last2 by auto
with bls x have x ≤ last2 by auto
with leq last2-def show ?thesis by auto
qed

```

```

lemma incrIndexList-help10[intro]: (incrIndexList ls m nmax) ⇒
  x ∈ set ls ⇒ x < nmax apply (cases ls rule: rev-exhaust) apply auto
apply (frule incrIndexList-help3) apply (auto dest: incrIndexList-1len)
apply (frule incrIndexList-help9) apply auto apply (drule incrIndexList-1len)
by arith

```

```

lemma enumerator-correctness: 2 < m ⇒ 1 < nmax ⇒
  ls ∈ set (enumerator m nmax) ⇒
  incrIndexList ls m nmax

```

proof –

```

  assume m: 2 < m and nmax: 1 < nmax and enum: ls ∈ set (enumerator m
  nmax)
  then have (hd ls = 0 ∧ last ls = (nmax - 1) ∧ length ls = m ∧ last (butlast
  ls) < last ls ∧ increasing ls)
  by (auto intro: enumerator-increasing enumerator-hd enumerator-last enumer-
  ator-length2 enumerator-bound3 simp: set-enumerator-simps)
  with m nmax show ?thesis by (unfold incrIndexList-def) auto
qed

```

```

lemma enumerator-completeness-help: ∧ ls. increasing ls ⇒ ls ≠ [] ⇒ length ls
= Suc ks ⇒ list-all (λx. x < Suc nmax) ls ⇒ ls ∈ set ((enumAppend nmax ~
ks) (enumBase nmax))

```

proof (induct ks)

case 0

```

assume increasing ls ls ≠ [] length ls = Suc 0 list-all (λx. x < Suc nmax) ls

```

```

then have ∃ x. ls = [x]

```

```

  apply (case-tac ls::nat list) by auto

```

```

then obtain x where ls1: ls = [x] by auto

```

```

with 0 have x < Suc nmax by auto

```

```

with ls1 show ?case apply (simp add: enumBase-def) by auto

```

next

case (Suc n)

```

define ls' where ls' = butlast ls

```

```

define l where l = last ls

```

```

define ll where ll = last ls'

```

```

define bl where bl = butlast ls'

```

```

define ls'list where ls'list = (enumAppend nmax  $\sim n$ ) (enumBase nmax)
then have short: (enumAppend nmax  $\sim n$ ) (enumBase nmax) = ls'list by simp
from Suc have ls  $\neq []$  by auto
then have ls = butlast ls @ [last ls] by auto
with ls'-def l-def have ls1: ls = ls' @ [l] by auto
with Suc have length ls' = Suc n by auto
then have ls'ne: ls'  $\neq []$  by auto
with ll-def bl-def have ls'1: ls' = bl @ [ll] by auto
then have ll-in-ls': ll  $\in$  set ls' by simp
from Suc ls1 have list-all ( $\lambda x. x < \text{Suc } nmax$ ) ls' by auto
with ll-in-ls' have ll < Suc nmax by (induct ls') auto
with ll-def have llsmall: last ls'  $\leq nmax$  by auto

from ls1 have l-in-ls: l  $\in$  set ls by auto
from Suc have list-all ( $\lambda x. x < \text{Suc } nmax$ ) ls by auto
with l-in-ls have l < Suc nmax by (induct ls) auto
then have lo: l  $\leq nmax$  by auto

from Suc ls1 ls'1 have increasing ((bl @ [ll]) @ [l]) by auto
then have ll  $\leq l$  by (rule increasing2) auto
with ll-def have lu: last ls'  $\leq l$  by simp

from Suc ls1 have vors: ls'  $\in$  set ((enumAppend nmax  $\sim n$ ) (enumBase nmax))
by (rule-tac Suc) (auto intro: increasing4)
with short have ls'  $\in$  set ls'list by auto
with short llsmall ls1 lo lu show ?case apply simp apply (simp add: enumAppend-def)
apply (intro beXI) by auto
qed

lemma enumerator-completeness:  $2 < m \implies \text{incrIndexList } ls \ m \ nmax \implies$ 
  ls  $\in$  set (enumerator m nmax)
proof –
assume m:  $2 < m$  and props: incrIndexList ls m nmax
then have props': (hd ls = 0  $\wedge$  last ls = (nmax – 1)
   $\wedge$  length ls = m  $\wedge$  last (butlast ls) < last ls  $\wedge$  increasing ls)
by (unfold incrIndexList-def) auto
show ?thesis
proof –
have props'': hd ls = 0  $\wedge$  last ls = (nmax – 1)  $\wedge$  length ls = m  $\wedge$ 
  increasing ls
by (auto simp: props')
show ls  $\in$  set (enumerator m nmax)
proof –
from m props'' have l-ls:  $2 < \text{length } ls$  by auto
then have  $\exists x \ y \ ks. ls = x \# ks @ [y]$ 
apply (case-tac ls::(nat list)) apply auto
apply (case-tac list rule: rev-exhaust) by auto
then obtain x y ks where ls = x # ks @ [y] by auto

```

```

with props'' have ls': ls = 0 # ks @ [nmax - 1] by auto
with l-ls have l-ms: 0 < length ks by auto
then have ms-ne: ks ≠ [] by auto
from ls' have lks: length ks = length ls - 2 by auto
from props'' have nd: increasing ls by auto
from props'' have ∧ z. z ∈ set ks ⇒ 0 ≤ z by auto
from props'' ls' have increasing ((0 # ks) @ [nmax - 1]) by auto
then have z: ∧ z. z ∈ set ks ⇒ z ≤ (nmax - 1)
  by (drule-tac increasing2) auto
from props ls' have z': ∧ z. z ∈ set ks ⇒ z ≤ (nmax - 2) by auto

have ks ∈ set ((enumAppend (nmax - 2)
  ~ (length ks - Suc 0)) (enumBase (nmax - 2)))
proof (cases ks = [])
  case True with ms-ne show ?thesis by simp
next
  case False
  from props'' have increasing ls by auto
  with ls' have increasing (0 # ks) by (auto intro: increasing4)
  then have increasing ([0] @ ks) by auto
  then have ndks: increasing ks by (rule-tac increasing5)
  have listall: list-all (λx. x < Suc (nmax - 2)) ks
    apply (simp add: list-all-iff)
    by (auto dest: z')
  with False ndks show ?thesis
  apply (rule-tac enumerator-completeness-help) by auto
qed
with lks props' have
  ks ∈ set ((enumAppend (nmax - 2) ~ (m - 3)) (enumBase (nmax - 2)))
by auto
  with m ls' show ?thesis by (simp add: enumerator-def)
qed
qed
qed

```

```

lemma enumerator-equiv[simp]:
  2 < n ⇒ 1 < m ⇒ is ∈ set(enumerator n m) = incrIndexList is n m
by (auto intro: enumerator-correctness enumerator-completeness)

```

end

12 Properties of Face Division

```

theory FaceDivisionProps
imports Plane EnumeratorProps
begin

```


12.1 Finality

lemma *vertices-makeFaceFinal*: $vertices(\text{makeFaceFinal } f \ g) = vertices \ g$
by(*induct g*)(*simp add:vertices-graph-def makeFaceFinal-def*)

lemma *edges-makeFaceFinal*: $\mathcal{E} (\text{makeFaceFinal } f \ g) = \mathcal{E} \ g$

proof –

{ **fix** *fs*

have $(\bigcup_{f \in \text{set } fs} (\text{makeFaceFinalFaceList } f \ fs) \ \text{edges } f) = (\bigcup_{f \in \text{set } fs} \text{edges } f)$

apply(*unfold makeFaceFinalFaceList-def*)

apply(*induct f*)

by(*induct fs*) *simp-all* }

thus *?thesis* **by**(*simp add:edges-graph-def makeFaceFinal-def*)

qed

lemma *in-set-repl-setFin*:

$f \in \text{set } fs \implies \text{final } f \implies f \in \text{set} (\text{replace } f' \ [\text{setFinal } f'] \ fs)$

by (*induct fs*) *auto*

lemma *in-set-repl*: $f \in \text{set } fs \implies f \neq f' \implies f \in \text{set} (\text{replace } f' \ fs' \ fs)$

by (*induct fs*) *auto*

lemma *makeFaceFinals-preserve-finals*:

$f \in \text{set} (\text{finals } g) \implies f \in \text{set} (\text{finals} (\text{makeFaceFinal } f' \ g))$

by (*induct g*)

(*simp add:makeFaceFinal-def finals-def makeFaceFinalFaceList-def in-set-repl-setFin*)

lemma *len-faces-makeFaceFinal[simp]*:

$|\text{faces} (\text{makeFaceFinal } f \ g)| = |\text{faces } g|$

by(*simp add:makeFaceFinal-def makeFaceFinalFaceList-def*)

lemma *len-finals-makeFaceFinal*:

$f \in \mathcal{F} \ g \implies \neg \text{final } f \implies |\text{finals} (\text{makeFaceFinal } f \ g)| = |\text{finals } g| + 1$

by(*simp add:makeFaceFinal-def finals-def makeFaceFinalFaceList-def length-filter-replace1*)

lemma *len-nonFinals-makeFaceFinal*:

$\llbracket \neg \text{final } f; f \in \mathcal{F} \ g \rrbracket$

$\implies |\text{nonFinals} (\text{makeFaceFinal } f \ g)| = |\text{nonFinals } g| - 1$

by(*simp add:makeFaceFinal-def nonFinals-def makeFaceFinalFaceList-def length-filter-replace2*)

lemma *set-finals-makeFaceFinal[simp]*: $\text{distinct}(\text{faces } g) \implies f \in \mathcal{F} \ g \implies$

$\text{set}(\text{finals} (\text{makeFaceFinal } f \ g)) = \text{insert} (\text{setFinal } f) (\text{set}(\text{finals } g))$

by(*auto simp:finals-def makeFaceFinal-def makeFaceFinalFaceList-def distinct-set-replace*)

lemma *splitFace-preserve-final*:
 $f \in \text{set } (\text{finals } g) \implies \neg \text{final } f' \implies$
 $f \in \text{set } (\text{finals } (\text{snd } (\text{snd } (\text{splitFace } g \ i \ j \ f' \ ns))))$
by (*induct* *g*) (*auto simp add: splitFace-def finals-def split-def*
intro: in-set-repl)

lemma *splitFace-nonFinal-face*:
 $\neg \text{final } (\text{fst } (\text{snd } (\text{splitFace } g \ i \ j \ f' \ ns)))$
by (*simp add: splitFace-def split-def split-face-def*)

lemma *subdivFace'-preserve-finals*:
 $\bigwedge n \ i \ f' \ g. f \in \text{set } (\text{finals } g) \implies \neg \text{final } f' \implies$
 $f \in \text{set } (\text{finals } (\text{subdivFace}' \ g \ f' \ i \ n \ is))$
proof (*induct is*)
case *Nil* **then show** *?case* **by** (*simp add: makeFaceFinals-preserve-finals*)
next
case (*Cons j js*) **then show** *?case*
proof (*cases j*)
case *None* **with** *Cons* **show** *?thesis* **by** *simp*
next
case (*Some sj*)
with *Cons* **show** *?thesis*
by (*auto simp: splitFace-preserve-final splitFace-nonFinal-face split-def*)
qed
qed

lemma *subdivFace-pres-finals*:
 $f \in \text{set } (\text{finals } g) \implies \neg \text{final } f' \implies$
 $f \in \text{set } (\text{finals } (\text{subdivFace } g \ f' \ is))$
by (*simp add: subdivFace-def subdivFace'-preserve-finals*)

declare *Nat.diff-is-0-eq'* [*simp del*]

12.2 *is-prefix*

definition *is-prefix* :: '*a list* \Rightarrow '*a list* \Rightarrow *bool* **where**
is-prefix *ls vs* $\equiv (\exists \text{ bs. } vs = \text{ls} @ \text{bs})$

lemma *is-prefix-add*:
 $\text{is-prefix } ls \ vs \implies \text{is-prefix } (as @ ls) \ (as @ vs)$ **by** (*simp add: is-prefix-def*)

lemma *is-prefix-hd*[*simp*]:
 $\text{is-prefix } [l] \ vs = (l = \text{hd } vs \wedge vs \neq [])$
apply (*rule iffI*) **apply** (*auto simp: is-prefix-def*)
apply (*intro exI*) **apply** (*subgoal-tac vs = hd vs # tl vs*) **apply** *assumption* **by**

auto

lemma *is-prefix-f[simp]*:

is-prefix (a#as) (a#vs) = is-prefix as vs **by** (*auto simp: is-prefix-def*)

lemma *splitAt-is-prefix*: $ram \in set\ vs \implies is_prefix\ (fst\ (splitAt\ ram\ vs)\ @\ [ram])$
vs

by (*auto dest!: splitAt-ram simp: is-prefix-def*)

12.3 *is-sublist*

definition *is-sublist* :: *'a list* \Rightarrow *'a list* \Rightarrow *bool* **where**

is-sublist ls vs $\equiv (\exists\ as\ bs.\ vs = as\ @\ ls\ @\ bs)$

lemma *is-prefix-sublist*:

is-prefix ls vs $\implies is_sublist\ ls\ vs$ **by** (*auto simp: is-prefix-def is-sublist-def*)

lemma *is-sublist-trans*: *is-sublist as bs* $\implies is_sublist\ bs\ cs \implies is_sublist\ as\ cs$

apply (*simp add: is-sublist-def*) **apply** (*elim exE*)

apply (*subgoal-tac cs = (asaa @ asa) @ as @ (bsa @ bsaa)*)

apply (*intro exI*) **apply** *assumption* **by** *force*

lemma *is-sublist-add*: *is-sublist as bs* $\implies is_sublist\ as\ (xs\ @\ bs\ @\ ys)$

apply (*simp add: is-sublist-def*) **apply** (*elim exE*)

apply (*subgoal-tac xs @ bs @ ys = (xs @ asa) @ as @ (bsa @ ys)*)

apply (*intro exI*) **apply** *assumption* **by** *auto*

lemma *is-sublist-rec*:

is-sublist xs ys =

(*if length xs > length ys then False else*

if xs = take (length xs) ys then True else is-sublist xs (tl ys))

proof (*simp add:is-sublist-def, goal-cases*)

case 1 **show** *?case*

proof (*standard, goal-cases*)

case 1 **show** *?case*

proof (*standard, goal-cases*)

case xs: 1

show *?case*

proof (*standard, goal-cases*)

case 1 **show** *?case* **by** *auto*

next

case 2 **show** *?case*

proof (*standard, goal-cases*)

case 1

have $ys = take\ |xs|\ ys\ @\ drop\ |xs|\ ys$ **by** *simp*

also **have** $\dots = []\ @\ xs\ @\ drop\ |xs|\ ys$ **by** (*simp add:xs[symmetric]*)

finally **show** *?case* **by** *blast*

qed

```

    qed
  qed
next
case 2 show ?case
proof (standard, goal-cases)
  case xs-neq: 1
  show ?case
  proof (standard, goal-cases)
    case 1 show ?case by auto
  next
  case 2 show ?case
  proof (standard, goal-cases)
    case not-less: 1 show ?case
    proof (standard, goal-cases)
      case 1
      then obtain as bs where ys: ys = as @ xs @ bs by blast
      have as ≠ [] using xs-neq ys by auto
      then obtain a as' where as = a # as'
        by (simp add:neq-Nil-conv) blast
      hence tl ys = as' @ xs @ bs by (simp add:ys)
      thus ?case by blast
    next
    case 2
    then obtain as bs where ys: tl ys = as @ xs @ bs by blast
    have ys ≠ [] using xs-neq not-less by auto
    then obtain y ys' where ys = y # ys'
      by (simp add:neq-Nil-conv) blast
    hence ys = (y#as) @ xs @ bs using ys by simp
    thus ?case by blast
  qed
  qed
  qed
  qed
  qed
qed

```

lemma *not-sublist-len*[simp]:
 $|ys| < |xs| \implies \neg \text{is-sublist } xs \ ys$
by(*simp add:is-sublist-rec*)

lemma *is-sublist-simp*[simp]: $a \neq v \implies \text{is-sublist } (a\#as) (v\#vs) = \text{is-sublist } (a\#as) \ vs$

proof
 assume *av*: $a \neq v$ and *subl*: $\text{is-sublist } (a \# as) (v \# vs)$
 then obtain *rs ts* where *vvs*: $v\#vs = rs @ (a \# as) @ ts$ **by** (*auto simp: is-sublist-def*)
 with *av* have $rs \neq []$ **by** *auto*
 with *vvs* have $tl (v\#vs) = tl rs @ a \# as @ ts$ **by** *auto*

then have $vs = tl\ rs\ @\ a\ \#\ as\ @\ ts$ **by** *auto*
then show $is\text{-}sublist\ (a\ \#\ as)\ vs$ **by** (*auto simp: is-sublist-def*)
next
assume $av: a \neq v$ **and** $subl: is\text{-}sublist\ (a\ \#\ as)\ vs$
then show $is\text{-}sublist\ (a\ \#\ as)\ (v\ \#\ vs)$ **apply** (*auto simp: is-sublist-def*) **apply**
(*intro exI*)
apply (*subgoal-tac* $v\ \#\ asa\ @\ a\ \#\ as\ @\ bs = (v\ \#\ asa)\ @\ a\ \#\ as\ @\ bs$) **apply**
assumption **by** *auto*
qed

lemma $is\text{-}sublist\text{-}id[simp]: is\text{-}sublist\ vs\ vs$ **apply** (*auto simp: is-sublist-def*) **apply**
(*intro exI*)
apply (*subgoal-tac* $vs = []\ @\ vs\ @\ []$) **by** (*assumption*) *auto*

lemma $is\text{-}sublist\text{-}in: is\text{-}sublist\ (a\ \#\ as)\ vs \implies a \in set\ vs$ **by** (*auto simp: is-sublist-def*)

lemma $is\text{-}sublist\text{-}in1: is\text{-}sublist\ [x,y]\ vs \implies y \in set\ vs$ **by** (*auto simp: is-sublist-def*)

lemma $is\text{-}sublist\text{-}notlast[simp]: distinct\ vs \implies x = last\ vs \implies \neg is\text{-}sublist\ [x,y]\ vs$
proof
assume $dvs: distinct\ vs$ **and** $xl: x = last\ vs$ **and** $subl: is\text{-}sublist\ [x,y]\ vs$
then obtain $rs\ ts$ **where** $vs = rs\ @\ x\ \#\ y\ \#\ ts$ **by** (*auto simp: is-sublist-def*)
define as **where** $as = rs\ @\ [x]$
define bs **where** $bs = y\ \#\ ts$
then have $bsne: bs \neq []$ **by** *auto*
from $as\text{-}def\ bs\text{-}def$ **have** $vs2: vs = as\ @\ bs$ **using** vs **by** *auto*
with $as\text{-}def$ **have** $xas: x \in set\ as$ **by** *auto*
from $bsne\ vs2$ **have** $last\ vs = last\ bs$ **by** *auto*
with xl **have** $x = last\ bs$ **by** *auto*
with $bsne$ **have** $bs = (butlast\ bs)\ @\ [x]$ **by** *auto*
then have $x \in set\ bs$ **by** (*induct* bs) *auto*
with $xas\ vs2\ dvs$ **show** *False* **by** *auto*
qed

lemma $is\text{-}sublist\text{-}nth1: is\text{-}sublist\ [x,y]\ ls \implies$
 $\exists\ i\ j. i < length\ ls \wedge j < length\ ls \wedge ls!i = x \wedge ls!j = y \wedge Suc\ i = j$
proof –
assume $subl: is\text{-}sublist\ [x,y]\ ls$
then obtain $as\ bs$ **where** $ls = as\ @\ x\ \#\ y\ \#\ bs$ **by** (*auto simp: is-sublist-def*)
then have $(length\ as) < length\ ls \wedge (Suc\ (length\ as)) < length\ ls \wedge ls!(length\ as) = x$
 $\wedge ls!(Suc\ (length\ as)) = y \wedge Suc\ (length\ as) = (Suc\ (length\ as))$
apply *auto* **apply** *hypsubst-thin* **apply** (*induct* as) **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma $is\text{-}sublist\text{-}nth2: \exists\ i\ j. i < length\ ls \wedge j < length\ ls \wedge ls!i = x \wedge ls!j = y \wedge$
 $Suc\ i = j \implies$
 $is\text{-}sublist\ [x,y]\ ls$

proof –

assume $\exists i j. i < \text{length } ls \wedge j < \text{length } ls \wedge ls!i = x \wedge ls!j = y \wedge \text{Suc } i = j$
then obtain $i j$ **where** $vors: i < \text{length } ls \wedge j < \text{length } ls \wedge ls!i = x \wedge ls!j = y$
 $\wedge \text{Suc } i = j$ **by** *auto*

then have $ls = \text{take } (\text{Suc } (\text{Suc } i)) \text{ } ls @ \text{drop } (\text{Suc } (\text{Suc } i)) \text{ } ls$ **by** *auto*
with $vors$ **have** $ls = \text{take } (\text{Suc } i) \text{ } ls @ [ls! (\text{Suc } i)] @ \text{drop } (\text{Suc } (\text{Suc } i)) \text{ } ls$
by (*auto simp: take-Suc-conv-app-nth*)
with $vors$ **have** $ls = \text{take } i \text{ } ls @ [ls!i] @ [ls! (\text{Suc } i)] @ \text{drop } (\text{Suc } (\text{Suc } i)) \text{ } ls$
by (*auto simp: take-Suc-conv-app-nth*)
with $vors$ **show** *?thesis* **by** (*auto simp: is-sublist-def*)

qed

lemma *is-sublist-tl: is-sublist (a # as) vs \implies is-sublist as vs* **apply** (*simp add: is-sublist-def*)

apply (*elim exE*) **apply** (*intro exI*)
apply (*subgoal-tac vs = (asa @ [a]) @ as @ bs*) **apply** *assumption* **by** *auto*

lemma *is-sublist-hd: is-sublist (a # as) vs \implies is-sublist [a] vs* **apply** (*simp add: is-sublist-def*) **by** *auto*

lemma *is-sublist-hd-eq[simp]: (is-sublist [a] vs) = (a \in set vs)* **apply** (*rule-tac iffI*)
apply (*simp add: is-sublist-def*) **apply** *force*
apply (*simp add: is-sublist-def*) **apply** (*induct vs*) **apply** *force* **apply** (*case-tac a = aa*) **apply** *force*
apply (*subgoal-tac a \in set vs*) **apply** *simp* **apply** (*elim exE*) **apply** (*intro exI*)
apply (*subgoal-tac aa # vs = (aa # as) @ a # bs*) **apply** (*assumption*) **by** *auto*

lemma *is-sublist-distinct-prefix:*

is-sublist (v # as) (v # vs) \implies distinct (v # vs) \implies is-prefix as vs

proof –

assume $d: \text{distinct } (v \# vs)$ **and** $\text{subl: is-sublist } (v \# as) (v \# vs)$
from subl **obtain** $rs \ ts$ **where** $v\text{-vs: } v \# vs = rs @ (v \# as) @ ts$ **by** (*simp add: is-sublist-def*) *auto*

from d **have** $v: v \notin \text{set } vs$ **by** *auto*
then have $\neg \text{is-sublist } (v \# as) \text{ } vs$ **by** (*auto dest: is-sublist-hd*)
with $v\text{-vs}$ **have** $rs = []$ **apply** (*cases rs*) **by** (*auto simp: is-sublist-def*)
with $v\text{-vs}$ **show** *is-prefix as vs* **by** (*auto simp: is-prefix-def*)

qed

lemma *is-sublist-distinct[intro]:*

is-sublist as vs \implies distinct vs \implies distinct as **by** (*auto simp: is-sublist-def*)

lemma *is-sublist-y-hd: distinct vs \implies y = hd vs \implies \neg is-sublist [x,y] vs*

proof

assume $d: \text{distinct } vs$ **and** $yh: y = \text{hd } vs$ **and** $\text{subl: is-sublist } [x, y] \text{ } vs$
then obtain $rs \ ts$ **where** $vs = rs @ x \# y \# ts$ **by** (*auto simp: is-sublist-def*)
define as **where** $as = rs @ [x]$
then have $asne: as \neq []$ **by** *auto*
define bs **where** $bs = y \# ts$

```

then have bsne:  $bs \neq []$  by auto
from as-def bs-def have vs2:  $vs = as @ bs$  using vs by auto
from bs-def have xbs:  $y \in \text{set } bs$  by auto
from vs2 asne have hd vs = hd as by simp
with yh have  $y = \text{hd } as$  by auto
with asne have  $y \in \text{set } as$  by (induct as) auto
with d xbs vs2 show False by auto
qed

lemma is-sublist-at1:  $\text{distinct } (as @ bs) \implies \text{is-sublist } [x,y] (as @ bs) \implies x \neq (\text{last } as)$   $\implies$ 
 $\text{is-sublist } [x,y] as \vee \text{is-sublist } [x,y] bs$ 
proof (cases  $x \in \text{set } as$ )
  assume d:  $\text{distinct } (as @ bs)$  and subl:  $\text{is-sublist } [x, y] (as @ bs)$  and xnl:  $x \neq \text{last } as$ 
  define vs where  $vs = as @ bs$ 
  with d have dvs:  $\text{distinct } vs$  by auto
  case True
  with xnl subl have ind:  $\text{is-sublist } (as@bs) vs \implies \text{is-sublist } [x, y] as$ 
  proof (induct as)
    case Nil
    then show ?case by force
  next
  case (Cons a as)
    assume ih:  $[[\text{is-sublist } (as@bs) vs; x \neq \text{last } as; \text{is-sublist } [x,y] (as @ bs); x \in \text{set } as]] \implies$ 
 $\text{is-sublist } [x, y] as$  and subl-aas-vs:  $\text{is-sublist } ((a \# as) @ bs) vs$ 
and xnl2:  $x \neq \text{last } (a \# as)$  and subl2:  $\text{is-sublist } [x, y] ((a \# as) @ bs)$ 
and x:  $x \in \text{set } (a \# as)$ 
    then have rule1:  $x \neq a \implies \text{is-sublist } [x,y] as$  apply (cases  $as = []$ ) apply
simp
apply (rule-tac ih) by (auto dest: is-sublist-tl)

    from dvs subl-aas-vs have daas:  $\text{distinct } (a \# as @ bs)$  apply (rule-tac
is-sublist-distinct) by auto
    from xnl2 have asne:  $x = a \implies as \neq []$  by auto
    with subl2 daas have yhdas:  $x = a \implies y = \text{hd } as$  apply simp apply (drule-tac
is-sublist-distinct-prefix) by auto
    with asne have  $x = a \implies as = y \# \text{tl } as$  by auto
    with asne yhdas have  $x = a \implies \text{is-prefix } [x,y] (a \# as)$  by auto
    then have rule2:  $x = a \implies \text{is-sublist } [x,y] (a \# as)$  by (simp add: is-prefix-sublist)

    from rule1 rule2 show ?case by (cases  $x = a$ ) auto
  qed
from vs-def d have is-sublist  $[x, y] as$  by (rule-tac ind) auto
then show ?thesis by auto
next
  assume d:  $\text{distinct } (as @ bs)$  and subl:  $\text{is-sublist } [x, y] (as @ bs)$  and xnl:  $x \neq \text{last } as$ 

```

```

define ars where ars = as
case False
with ars-def have xars:  $x \notin \text{set } ars$  by auto
from subl have ind:  $is\_sublist\ as\ ars \implies is\_sublist\ [x,\ y]\ bs$ 
proof (induct as)
  case Nil
  then show ?case by auto
next
  case (Cons a as)
  assume ih:  $\llbracket is\_sublist\ as\ ars;\ is\_sublist\ [x,\ y]\ (as\ @\ bs) \rrbracket \implies is\_sublist\ [x,\ y]\ bs$ 
  and subl-aasbsvs:  $is\_sublist\ (a\ \#\ as)\ ars$  and subl2:  $is\_sublist\ [x,\ y]\ ((a\ \#\ as)\ @\ bs)$ 
  from subl-aasbsvs ars-def False have  $x \neq a$  by (auto simp:is-sublist-in)
  with subl-aasbsvs subl2 show ?thesis apply (rule-tac ih) by (auto dest:is-sublist-tl)
  qed
  from ars-def have  $is\_sublist\ [x,\ y]\ bs$  by (rule-tac ind) auto
  then show ?thesis by auto
qed

```

```

lemma is-sublist-at4:  $distinct\ (as\ @\ bs) \implies is\_sublist\ [x,y]\ (as\ @\ bs) \implies as \neq [] \implies x = last\ as \implies y = hd\ bs$ 
proof -
  assume d:  $distinct\ (as\ @\ bs)$  and subl:  $is\_sublist\ [x,y]\ (as\ @\ bs)$ 
  and asne:  $as \neq []$  and xl:  $x = last\ as$ 
  define vs where  $vs = as\ @\ bs$ 
  with subl have  $is\_sublist\ [x,y]\ vs$  by auto
  then obtain rs ts where  $vs2: vs = rs\ @\ x\ \#\ y\ \#\ ts$  by (auto simp: is-sublist-def)
  from vs-def d have dvs:  $distinct\ vs$  by auto
  from asne xl have as:  $as = butlast\ as\ @\ [x]$  by auto
  with vs-def have vs3:  $vs = butlast\ as\ @\ x\ \#\ bs$  by auto
  from dvs vs2 vs3 have  $rs = butlast\ as$  apply (rule-tac dist-at1) by auto
  then have  $rs\ @\ [x] = butlast\ as\ @\ [x]$  by auto
  with as have  $rs\ @\ [x] = as$  by auto
  then have  $as = rs\ @\ [x]$  by auto
  with vs2 vs-def have  $bs = y\ \#\ ts$  by auto
  then show ?thesis by auto
qed

```

```

lemma is-sublist-at5:  $distinct\ (as\ @\ bs) \implies is\_sublist\ [x,y]\ (as\ @\ bs) \implies is\_sublist\ [x,y]\ as \vee is\_sublist\ [x,y]\ bs \vee x = last\ as \wedge y = hd\ bs$ 
apply (case-tac as = []) apply simp apply (cases x = last as)
apply (subgoal-tac y = hd bs) apply simp
apply (rule is-sublist-at4) apply assumption+
apply (rule-tac is-sublist-at1) by auto

```

```

lemma is-sublist-rev:  $is\_sublist\ [a,b]\ (rev\ zs) = is\_sublist\ [b,a]\ zs$ 
apply (simp add: is-sublist-def)
apply (intro iffI) apply (elim exE) apply (intro exI)

```


apply (*subgoal-tac* $zs = (\text{rev } bs) @ b \# a \# \text{rev } as$) **apply** *assumption*
apply (*subgoal-tac* $\text{rev } (\text{rev } zs) = \text{rev } (as @ a \# b \# bs)$)
apply (*thin-tac* $\text{rev } zs = as @ a \# b \# bs$) **apply** *simp*
apply *simp*
apply (*elim* exE) **apply** (*intro* exI) **by** *force*

lemma *is-sublist-at5'*[*simp*]:
 $\text{distinct } as \implies \text{distinct } bs \implies \text{set } as \cap \text{set } bs = \{\} \implies \text{is-sublist } [x,y] (as @ bs)$
 \implies
 $\text{is-sublist } [x,y] as \vee \text{is-sublist } [x,y] bs \vee x = \text{last } as \wedge y = \text{hd } bs$
apply (*subgoal-tac* $\text{distinct } (as @ bs)$) **apply** (*drule* *is-sublist-at5*) **by** *auto*

lemma *splitAt-is-sublist1R*[*simp*]: $\text{ram} \in \text{set } vs \implies \text{is-sublist } (\text{fst } (\text{splitAt } \text{ram } vs) @ [\text{ram}]) vs$
apply (*auto* *dest!*: *splitAt-ram simp: is-sublist-def*) **apply** (*intro* exI)
apply (*subgoal-tac* $vs = [] @ \text{fst } (\text{splitAt } \text{ram } vs) @ \text{ram} \# \text{snd } (\text{splitAt } \text{ram } vs)$)
apply *assumption* **by** *simp*

lemma *splitAt-is-sublist2R*[*simp*]: $\text{ram} \in \text{set } vs \implies \text{is-sublist } (\text{ram} \# \text{snd } (\text{splitAt } \text{ram } vs)) vs$
apply (*auto* *dest!*: *splitAt-ram splitAt-no-ram simp: is-sublist-def*) **apply** (*intro* exI)
apply (*subgoal-tac* $vs = \text{fst } (\text{splitAt } \text{ram } vs) @ \text{ram} \# \text{snd } (\text{splitAt } \text{ram } vs) @ []$)
apply *assumption* **by** *auto*

12.4 *is-nextElem*

definition *is-nextElem* :: 'a list \Rightarrow 'a \Rightarrow 'a \Rightarrow bool **where**
 $\text{is-nextElem } xs \ x \ y \equiv \text{is-sublist } [x,y] xs \vee xs \neq [] \wedge x = \text{last } xs \wedge y = \text{hd } xs$

lemma *is-nextElem-a*[*intro*]: $\text{is-nextElem } vs \ a \ b \implies a \in \text{set } vs$
by (*auto* *simp: is-nextElem-def is-sublist-def*)

lemma *is-nextElem-b*[*intro*]: $\text{is-nextElem } vs \ a \ b \implies b \in \text{set } vs$
by (*auto* *simp: is-nextElem-def is-sublist-def*)

lemma *is-nextElem-last-hd*[*intro*]: $\text{distinct } vs \implies \text{is-nextElem } vs \ x \ y \implies x = \text{last } vs \implies y = \text{hd } vs$
by (*auto* *simp: is-nextElem-def*)

lemma *is-nextElem-last-ne*[*intro*]: $\text{distinct } vs \implies \text{is-nextElem } vs \ x \ y \implies x = \text{last } vs \implies vs \neq []$
by (*auto* *simp: is-nextElem-def*)

lemma *is-nextElem-sublistI*: $\text{is-sublist } [x,y] vs \implies \text{is-nextElem } vs \ x \ y$
by (*auto* *simp: is-nextElem-def*)

lemma *is-nextElem-nth1*: $\text{is-nextElem } ls \ x \ y \implies \exists \ i \ j. \ i < \text{length } ls \wedge j < \text{length } ls \wedge ls!i = x \wedge ls!j = y \wedge (\text{Suc } i) \bmod (\text{length } ls) = j$

proof (*cases* *is-sublist* $[x,y] ls$)

assume *is-nextElem: is-nextElem* $ls \ x \ y$

case *True* **then show** *?thesis* **apply** (*drule-tac* *is-sublist-nth1*) **by** *auto*
next

assume *is-nextElem*: *is-nextElem* *ls* *x* *y*
case *False* **with** *is-nextElem* **have** *hl*: $ls \neq [] \wedge last\ ls = x \wedge hd\ ls = y$
by (*auto simp: is-nextElem-def*)
then have *j*: $ls!0 = y$ **by** (*cases ls*) *auto*
from *hl* **have** *i*: $ls!(length\ ls - 1) = x$ **by** (*cases ls rule: rev-exhaust*) *auto*
from *i j hl* **have** $(length\ ls - 1) < length\ ls \wedge 0 < length\ ls \wedge ls!(length\ ls - 1)$
 $= x$
 $\wedge ls!0 = y \wedge (Suc\ (length\ ls - 1))\ mod\ (length\ ls) = 0$ **by** *auto*
then show *?thesis* **apply** (*intro exI*) .
qed

lemma *is-nextElem-nth2*: $\exists\ i\ j. i < length\ ls \wedge j < length\ ls \wedge ls!i = x \wedge ls!j = y$
 $\wedge (Suc\ i)\ mod\ (length\ ls) = j \implies is-nextElem\ ls\ x\ y$
proof –
assume $\exists\ i\ j. i < length\ ls \wedge j < length\ ls \wedge ls!i = x \wedge ls!j = y \wedge (Suc\ i)\ mod$
 $(length\ ls) = j$
then obtain *i j* **where** *vors*: $i < length\ ls \wedge j < length\ ls \wedge ls!i = x \wedge ls!j = y$
 $\wedge (Suc\ i)\ mod\ (length\ ls) = j$ **by** *auto*
then show *?thesis*
proof (*cases Suc i = length ls*)
case *True* **with** *vors* **have** $j = 0$ **by** *auto*

with *True vors* **show** *?thesis* **apply** (*auto simp: is-nextElem-def*)
apply (*cases ls rule: rev-exhaust*) **apply** *auto* **apply** (*cases ls*) **by** *auto*
next
case *False* **with** *vors* **have** *is-sublist* $[x,y]\ ls$
apply (*rule-tac is-sublist-nth2*) **by** *auto*
then show *?thesis* **by** (*simp add: is-nextElem-def*)
qed
qed

lemma *is-nextElem-rotate1-aux*:
 $is-nextElem\ (rotate\ m\ ls)\ x\ y \implies is-nextElem\ ls\ x\ y$
proof –
assume *is-nextElem*: *is-nextElem* $(rotate\ m\ ls)\ x\ y$
define *n* **where** $n = m\ mod\ length\ ls$
then have *rot-eq*: $rotate\ m\ ls = rotate\ n\ ls$
by (*auto intro: rotate-conv-mod*)
with *is-nextElem* **have** *is-nextElem* $(rotate\ n\ ls)\ x\ y$
by *simp*
then obtain *i j* **where** *vors*: $i < length\ (rotate\ n\ ls) \wedge j < length\ (rotate\ n\ ls) \wedge$
 $(rotate\ n\ ls)!i = x \wedge (rotate\ n\ ls)!j = y \wedge$
 $(Suc\ i)\ mod\ (length\ (rotate\ n\ ls)) = j$
by (*drule-tac is-nextElem-nth1*) *auto*
then have *lls*: $0 < length\ ls$
by *auto*
define *k* **where** $k = (i+n)\ mod\ (length\ ls)$
with *lls* **have** *sk*: $k < length\ ls$

```

  by simp
from k-def lls vors have ls!k = (rotate n ls)!(i mod (length ls))
  by (simp add: nth-rotate)
with vors have lsk: ls!k = x
  by simp
define l where l = (j+n) mod (length ls)
with lls have sl: l < length ls
  by simp
from l-def lls vors have ls!l = (rotate n ls)!(j mod (length ls))
  by (simp add: nth-rotate)
with vors have lsl: ls!l = y
  by simp
from vors k-def l-def
have (Suc i) mod length ls = j
  by simp
then have (Suc i) mod length ls = j mod length ls
  by auto
then have ((Suc i) mod length ls + n mod (length ls)) mod length ls
  = (j mod length ls + n mod (length ls)) mod length ls
  by simp
then have ((Suc i) + n) mod length ls = (j + n) mod length ls
  by (simp add: mod-simps)
with vors k-def l-def have (Suc k) mod (length ls) = l
  by (simp add: mod-simps)
with sk lsk sl lsl
show ?thesis
  by (auto intro: is-nextElem-nth2)
qed

```

lemma *is-nextElem-rotate-eq[simp]*: $is_nextElem\ (rotate\ m\ ls)\ x\ y = is_nextElem\ ls\ x\ y$
 apply (auto dest: is-nextElem-rotate1-aux) apply (rule is-nextElem-rotate1-aux)
 apply (subgoal-tac is-nextElem (rotate (length ls - m mod length ls) (rotate m ls)) x y)
 apply assumption by simp

lemma *is-nextElem-congs-eq*: $ls \cong ms \implies is_nextElem\ ls\ x\ y = is_nextElem\ ms\ x\ y$
 by (auto simp: congs-def)

lemma *is-nextElem-rev[simp]*: $is_nextElem\ (rev\ zs)\ a\ b = is_nextElem\ zs\ b\ a$
 apply (simp add: is-nextElem-def is-sublist-rev)
 apply (case-tac zs = []) apply simp apply simp
 apply (case-tac a = hd zs) apply (case-tac zs) apply simp apply simp apply simp
 apply (case-tac a = last (rev zs) \wedge b = last zs) apply simp
 apply (case-tac zs rule: rev-exhaust) apply simp
 apply (case-tac ys) apply simp apply simp by force

```

lemma is-nextElem-circ:
  [| distinct xs; is-nextElem xs a b; is-nextElem xs b a |]  $\implies$   $|xs| \leq 2$ 
apply(drule is-nextElem-nth1)
apply(drule is-nextElem-nth1)
apply (clarsimp)
apply(rename-tac i j)
apply(frule-tac i=j and j = Suc i mod |xs| in nth-eq-iff-index-eq)
  apply assumption+
apply(frule-tac j=i and i = Suc j mod |xs| in nth-eq-iff-index-eq)
  apply assumption+
apply(rule ccontr)
apply(simp add: distinct-conv-nth mod-Suc)
done

```

12.5 *nextElem*, *sublist*, *is-nextElem*

```

lemma is-sublist-eq: distinct vs  $\implies$   $c \neq y \implies$ 
  (nextElem vs c x = y) = is-sublist [x,y] vs
proof -
  assume d: distinct vs and c: c ≠ y
  have r1: nextElem vs c x = y  $\implies$  is-sublist [x,y] vs
  proof -
    assume fn: nextElem vs c x = y
    with c show ?thesis by(drule-tac nextElem-cases)(auto simp: is-sublist-def)
  qed
  with d have r2: is-sublist [x,y] vs  $\implies$  nextElem vs c x = y
  apply (simp add: is-sublist-def) apply (elim exE) by auto
  show ?thesis apply (intro iffI r1) by (auto intro: r2)
qed

```

```

lemma is-nextElem1: distinct vs  $\implies$   $x \in \text{set } vs \implies$  nextElem vs (hd vs) x = y
 $\implies$  is-nextElem vs x y
proof -
  assume d: distinct vs and x: x ∈ set vs and fn: nextElem vs (hd vs) x = y
  from x have r0: vs ≠ [] by auto
  from d fn have r1:  $x = \text{last } vs \implies y = \text{hd } vs$  by (auto)
  from d fn have r3:  $\text{hd } vs \neq y \implies (\exists a b. \text{vs} = a @ [x,y] @ b)$  by (drule-tac
nextElem-cases) auto

```

```

  from x obtain n where xn:  $x = \text{vs}!n$  and nl:  $n < \text{length } vs$  by (auto simp:
in-set-conv-nth)
  define as where as = take n vs
  define bs where bs = drop (Suc n) vs
  from as-def bs-def xn nl have vs:  $\text{vs} = \text{as} @ [x] @ \text{bs}$  by (auto intro: id-take-nth-drop)
  then have r2:  $x \neq \text{last } vs \implies y \neq \text{hd } vs$ 
  proof -
    assume notx:  $x \neq \text{last } vs$ 
    from vs notx have bs  $\neq []$  by auto

```

with vs **have** $r2: vs = as @ [x, hd bs] @ tl bs$ **by** *auto*
with d **have** $ineq: hd bs \neq hd vs$ **by** (*cases as*) *auto*
from d **fn** $r2$ **have** $y = hd bs$ **by** *auto*
with $ineq$ **show** $?thesis$ **by** *auto*
qed
from $r0 r1 r2 r3$ **show** $?thesis$ **apply** (*simp add:is-nextElem-def is-sublist-def*)
apply (*cases x = last vs*) **by** *auto*
qed

lemma *is-nextElem2*: $distinct\ vs \implies x \in set\ vs \implies is-nextElem\ vs\ x\ y \implies nextElem\ vs\ (hd\ vs)\ x = y$
proof –
assume $d: distinct\ vs$ **and** $x: x \in set\ vs$ **and** $is-nextElem: is-nextElem\ vs\ x\ y$
then **show** $?thesis$ **apply** (*simp add: is-nextElem-def*) **apply** (*cases is-sublist [x,y] vs*)
apply (*cases y = hd vs*)
apply (*simp add: is-sublist-def*) **apply** (*force dest: distinct-hd-not-cons*)
apply (*subgoal-tac hd vs \neq y*) **apply** (*simp add: is-sublist-eq*) **by** *auto*
qed

lemma *nextElem-is-nextElem*:
 $distinct\ xs \implies x \in set\ xs \implies is-nextElem\ xs\ x\ y = (nextElem\ xs\ (hd\ xs)\ x = y)$
by (*auto intro!: is-nextElem1 is-nextElem2*)

lemma *nextElem-congs-eq*: $xs \cong ys \implies distinct\ xs \implies x \in set\ xs \implies nextElem\ xs\ (hd\ xs)\ x = nextElem\ ys\ (hd\ ys)\ x$
proof –
assume $eq: xs \cong ys$ **and** $dist: distinct\ xs$ **and** $x: x \in set\ xs$
define y **where** $y = nextElem\ xs\ (hd\ xs)\ x$
then **have** $f1: nextElem\ xs\ (hd\ xs)\ x = y$ **by** *auto*
with $dist\ x$ **have** $is-nextElem\ xs\ x\ y$ **by** (*auto intro: is-nextElem1*)
with eq **have** $is-nextElem\ ys\ x\ y$ **by** (*simp add:is-nextElem-congs-eq*)
with $eq\ dist\ x$ **have** $f2: nextElem\ ys\ (hd\ ys)\ x = y$
by (*auto simp: congs-distinct intro: is-nextElem2*)
from $f1\ f2$ **show** $?thesis$ **by** *auto*
qed

lemma *is-sublist-is-nextElem*: $distinct\ vs \implies is-nextElem\ vs\ x\ y \implies is-sublist\ as\ vs \implies x \in set\ as \implies x \neq last\ as \implies is-sublist\ [x,y]\ as$
proof –
assume $d: distinct\ vs$ **and** $is-nextElem: is-nextElem\ vs\ x\ y$ **and** $subl: is-sublist\ as\ vs$ **and** $xin: x \in set\ as$ **and** $xnl: x \neq last\ as$
from xin **have** $asne: as \neq []$ **by** *auto*
with $subl$ **have** $vsne: vs \neq []$ **by** (*auto simp: is-sublist-def*)
from $subl$ **obtain** $rs\ ts$ **where** $vs: vs = rs @ as @ ts$ **apply** (*simp add: is-sublist-def*) **apply** (*elim exE*) **by** *auto*
with $d\ xnl\ asne$ **have** $x \neq last\ vs$

```

proof (cases ts = [])
  case True
    with d xnl asne vs show ?thesis by force
  next
    define lastvs where lastvs = last ts
    case False
      with vs lastvs-def have vs2: vs = rs @ as @ butlast ts @ [lastvs] by auto
      with d have lastvs  $\notin$  set as by auto
      with xin have lastvs  $\neq$  x by auto
      with vs2 show ?thesis by auto
    qed
  with is-nextElem have subl-vs: is-sublist [x,y] vs by (auto simp: is-nextElem-def)
  from d xin vs have  $\neg$  is-sublist [x] rs by auto
  then have nrs:  $\neg$  is-sublist [x,y] rs by (auto dest: is-sublist-hd)
  from d xin vs have  $\neg$  is-sublist [x] ts by auto
  then have nts:  $\neg$  is-sublist [x,y] ts by (auto dest: is-sublist-hd)
  from d xin vs have xnrs: x  $\notin$  set rs by auto
  then have notrs:  $\neg$  is-sublist [x,y] rs by (auto simp: is-sublist-in)
  from xnrs have xnrs: rs  $\neq$  []  $\implies$  x  $\neq$  last rs by (induct rs) auto
  from d xin vs have xnts: x  $\notin$  set ts by auto
  then have notts:  $\neg$  is-sublist [x,y] ts by (auto simp: is-sublist-in)
  from d vs subl-vs have is-sublist [x,y] rs  $\vee$  is-sublist [x,y] (as@ts) apply (cases
rs = []) apply simp apply (rule-tac is-sublist-at1) by (auto intro!: xnrs)
  with notrs have is-sublist [x,y] (as@ts) by auto
  with d vs xnl have is-sublist [x,y] as  $\vee$  is-sublist [x,y] ts apply (rule-tac is-sublist-at1)
by auto
  with notts show is-sublist [x,y] as by auto
qed

```

12.6 before

definition before :: 'a list \Rightarrow 'a \Rightarrow 'a \Rightarrow bool **where**
before vs ram1 ram2 \equiv \exists a b c. vs = a @ ram1 # b @ ram2 # c

lemma before-dist-fst-fst[simp]: before vs ram1 ram2 \implies distinct vs \implies fst (splitAt ram2 (fst (splitAt ram1 vs))) = fst (splitAt ram1 (fst (splitAt ram2 vs)))
apply (simp add: before-def) **apply** (elim exE)
apply (drule splitAt-dist-ram-all) **by** (auto dest!: pairD)

lemma before-dist-fst-snd[simp]: before vs ram1 ram2 \implies distinct vs \implies fst (splitAt ram2 (snd (splitAt ram1 vs))) = snd (splitAt ram1 (fst (splitAt ram2 vs)))
apply (simp add: before-def) **apply** (elim exE)
apply (drule-tac splitAt-dist-ram-all) **by** (auto dest!: pairD)

lemma before-dist-snd-fst[simp]: before vs ram1 ram2 \implies distinct vs \implies snd (splitAt ram2 (fst (splitAt ram1 vs))) = snd (splitAt ram1 (snd (splitAt ram2 vs)))
apply (simp add: before-def) **apply** (elim exE)
apply (drule-tac splitAt-dist-ram-all) **by** (auto dest!: pairD)

lemma *before-dist-snd-snd[simp]*: $before\ vs\ ram1\ ram2 \implies distinct\ vs \implies snd\ (splitAt\ ram2\ (snd\ (splitAt\ ram1\ vs))) = fst\ (splitAt\ ram1\ (snd\ (splitAt\ ram2\ vs)))$
apply (*simp add: before-def*) **apply** (*elim exE*)
apply (*drule-tac splitAt-dist-ram-all*) **by** (*auto dest!: pairD*)

lemma *before-dist-snd[simp]*: $before\ vs\ ram1\ ram2 \implies distinct\ vs \implies fst\ (splitAt\ ram1\ (snd\ (splitAt\ ram2\ vs))) = snd\ (splitAt\ ram2\ vs)$
apply (*simp add: before-def*) **apply** (*elim exE*)
apply (*drule-tac splitAt-dist-ram-all*) **by** (*auto dest!: pairD*)

lemma *before-dist-fst[simp]*: $before\ vs\ ram1\ ram2 \implies distinct\ vs \implies fst\ (splitAt\ ram1\ (fst\ (splitAt\ ram2\ vs))) = fst\ (splitAt\ ram1\ vs)$
apply (*simp add: before-def*) **apply** (*elim exE*)
apply (*drule-tac splitAt-dist-ram-all*) **by** (*auto dest!: pairD*)

lemma *before-or*: $ram1 \in set\ vs \implies ram2 \in set\ vs \implies ram1 \neq ram2 \implies before\ vs\ ram1\ ram2 \vee before\ vs\ ram2\ ram1$

proof –

assume $r1: ram1 \in set\ vs$ **and** $r2: ram2 \in set\ vs$ **and** $r12: ram1 \neq ram2$

then show *?thesis*

proof (*cases ram2 \in set (snd (splitAt ram1 vs))*)

define a **where** $a = fst\ (splitAt\ ram1\ vs)$

define b **where** $b = fst\ (splitAt\ ram2\ (snd\ (splitAt\ ram1\ vs)))$

define c **where** $c = snd\ (splitAt\ ram2\ (snd\ (splitAt\ ram1\ vs)))$

case *True* **with** $r1$ *a-def b-def c-def* **have** $vs = a @ [ram1] @ b @ [ram2] @ c$
by (*auto dest!: splitAt-ram*)

then show *?thesis* **apply** (*simp add: before-def*) **by** *auto*

next

define ab **where** $ab = fst\ (splitAt\ ram1\ vs)$

case *False*

with $r1\ r2\ r12$ *ab-def* **have** $r2': ram2 \in set\ ab$ **by** (*auto intro: splitAt-ram3*)

define a **where** $a = fst\ (splitAt\ ram2\ ab)$

define b **where** $b = snd\ (splitAt\ ram2\ ab)$

define c **where** $c = snd\ (splitAt\ ram1\ vs)$

from $r1$ *ab-def c-def* **have** $vs = ab @ [ram1] @ c$ **by** (*auto dest!: splitAt-ram*)

with $r2'$ *a-def b-def* **have** $vs = (a @ [ram2] @ b) @ [ram1] @ c$ **by** (*drule-tac splitAt-ram simp*)

then show *?thesis* **apply** (*simp add: before-def*) **apply** (*rule disjI2*) **by** *auto*

qed

qed

lemma *before-r1*:

$before\ vs\ r1\ r2 \implies r1 \in set\ vs$ **by** (*auto simp: before-def*)

lemma *before-r2*:

$before\ vs\ r1\ r2 \implies r2 \in set\ vs$ **by** (*auto simp: before-def*)

lemma *before-dist-r2*:

$distinct\ vs \implies before\ vs\ r1\ r2 \implies r2 \in set\ (snd\ (splitAt\ r1\ vs))$

proof –
assume d : *distinct vs* **and** b : *before vs r1 r2*
from d b **have** $ex1$: $\exists! s. (vs = (fst\ s) @ r1 \# snd\ (s))$ **apply** (*drule-tac before-r1*)
apply (*rule distinct-unique1*) **by** *auto*
from d b $ex1$ **show** *?thesis* **apply** (*unfold before-def*)
proof (*elim exE ex1E*)
fix a b c s
assume vs : $vs = a @ r1 \# b @ r2 \# c$ **and** $\forall y. vs = fst\ y @ r1 \# snd\ y$
 $\longrightarrow y = s$
then **have** $\bigwedge y. vs = fst\ y @ r1 \# snd\ y \longrightarrow y = s$ **by** (*clarify, hypsubst-thin, auto*)
then **have** *single*: $\bigwedge y. vs = fst\ y @ r1 \# snd\ y \implies y = s$ **by** *auto*
define bc **where** $bc = b @ r2 \# c$
with vs **have** $vs2$: $vs = a @ r1 \# bc$ **by** *auto*
from *bc-def* **have** $r2$: $r2 \in set\ bc$ **by** *auto*
define t **where** $t = (a, bc)$
with $vs2$ **have** $vs3$: $vs = fst\ (t) @ r1 \# snd\ (t)$ **by** *auto*
with *single* **have** ts : $t = s$ **by** (*rule-tac single*) *auto*
from b **have** *splitAt r1 vs = s* **apply** (*drule-tac before-r1*) **apply** (*drule-tac splitAt-ram*) **by** (*rule single*) *auto*
with ts **have** $t = splitAt\ r1\ vs$ **by** *simp*
with *t-def* **have** $bc = snd(splitAt\ r1\ vs)$ **by** *simp*
with $r2$ **show** *?thesis* **by** *simp*
qed
qed

lemma *before-dist-not-r2[intro]*:
 $distinct\ vs \implies before\ vs\ r1\ r2 \implies r2 \notin set\ (fst\ (splitAt\ r1\ vs))$ **apply** (*frule before-dist-r2*) **by** (*auto dest: splitAt-distinct-fst-snd*)

lemma *before-dist-r1*:
 $distinct\ vs \implies before\ vs\ r1\ r2 \implies r1 \in set\ (fst\ (splitAt\ r2\ vs))$

proof –
assume d : *distinct vs* **and** b : *before vs r1 r2*
from d b **have** $ex1$: $\exists! s. (vs = (fst\ s) @ r2 \# snd\ (s))$ **apply** (*drule-tac before-r2*)
apply (*rule distinct-unique1*) **by** *auto*
from d b $ex1$ **show** *?thesis* **apply** (*unfold before-def*)
proof (*elim exE ex1E*)
fix a b c s
assume vs : $vs = a @ r1 \# b @ r2 \# c$ **and** $\forall y. vs = fst\ y @ r2 \# snd\ y$
 $\longrightarrow y = s$
then **have** $\bigwedge y. vs = fst\ y @ r2 \# snd\ y \longrightarrow y = s$ **by** (*clarify, hypsubst-thin, auto*)
then **have** *single*: $\bigwedge y. vs = fst\ y @ r2 \# snd\ y \implies y = s$ **by** *auto*
define ab **where** $ab = a @ r1 \# b$
with vs **have** $vs2$: $vs = ab @ r2 \# c$ **by** *auto*
from *ab-def* **have** $r1$: $r1 \in set\ ab$ **by** *auto*
define t **where** $t = (ab, c)$
with $vs2$ **have** $vs3$: $vs = fst\ (t) @ r2 \# snd\ (t)$ **by** *auto*

with *single* **have** $ts: t = s$ **by** (*rule-tac single*) *auto*
from *b* **have** $splitAt\ r2\ vs = s$ **apply** (*drule-tac before-r2*) **apply** (*drule-tac splitAt-ram*) **by** (*rule single*) *auto*
with *ts* **have** $t = splitAt\ r2\ vs$ **by** *simp*
with *t-def* **have** $ab = fst(splitAt\ r2\ vs)$ **by** *simp*
with *r1* **show** *?thesis* **by** *simp*
qed
qed

lemma *before-dist-not-r1[intro]*:

$distinct\ vs \implies before\ vs\ r1\ r2 \implies r1 \notin set\ (snd\ (splitAt\ r2\ vs))$ **apply** (*frule before-dist-r1*) **by** (*auto dest: splitAt-distinct-fst-snd*)

lemma *before-snd*:

$r2 \in set\ (snd\ (splitAt\ r1\ vs)) \implies before\ vs\ r1\ r2$

proof –

assume *r2*: $r2 \in set\ (snd\ (splitAt\ r1\ vs))$

from *r2* **have** *r1*: $r1 \in set\ vs$ **apply** (*rule-tac ccontr*) **apply** (*drule splitAt-no-ram*) **by** *simp*

define *a* **where** $a = fst\ (splitAt\ r1\ vs)$

define *bc* **where** $bc = snd\ (splitAt\ r1\ vs)$

define *b* **where** $b = fst\ (splitAt\ r2\ bc)$

define *c* **where** $c = snd\ (splitAt\ r2\ bc)$

from *r1* *a-def* *bc-def* **have** *vs*: $vs = a\ @\ [r1]\ @\ bc$ **by** (*auto dest: splitAt-ram*)

from *r2* *bc-def* **have** *r2*: $r2 \in set\ bc$ **by** *simp*

with *b-def* *c-def* **have** $bc = b\ @\ [r2]\ @\ c$ **by** (*auto dest: splitAt-ram*)

with *vs* **show** *?thesis* **by** (*simp add: before-def*) *auto*

qed

lemma *before-fst*:

$r2 \in set\ vs \implies r1 \in set\ (fst\ (splitAt\ r2\ vs)) \implies before\ vs\ r1\ r2$

proof –

assume *r2*: $r2 \in set\ vs$ **and** *r1*: $r1 \in set\ (fst\ (splitAt\ r2\ vs))$

define *ab* **where** $ab = fst\ (splitAt\ r2\ vs)$

define *c* **where** $c = snd\ (splitAt\ r2\ vs)$

define *a* **where** $a = fst\ (splitAt\ r1\ ab)$

define *b* **where** $b = snd\ (splitAt\ r1\ ab)$

from *r2* *ab-def* *c-def* **have** *vs*: $vs = ab\ @\ [r2]\ @\ c$ **by** (*auto dest: splitAt-ram*)

from *r1* *ab-def* **have** *r1*: $r1 \in set\ ab$ **by** *simp*

with *a-def* *b-def* **have** $ab = a\ @\ [r1]\ @\ b$ **by** (*auto dest: splitAt-ram*)

with *vs* **show** *?thesis* **by** (*simp add: before-def*) *auto*

qed

lemma *before-dist-eq-fst*:

$distinct\ vs \implies r2 \in set\ vs \implies r1 \in set\ (fst\ (splitAt\ r2\ vs)) = before\ vs\ r1\ r2$

by (*auto intro: before-fst before-dist-r1*)

lemma *before-dist-eq-snd*:

distinct vs $\implies r2 \in \text{set } (\text{snd } (\text{splitAt } r1 \text{ vs})) = \text{before } vs \ r1 \ r2$
by (*auto intro: before-snd before-dist-r2*)

lemma *before-dist-not1*:

distinct vs $\implies \text{before } vs \ ram1 \ ram2 \implies \neg \text{before } vs \ ram2 \ ram1$

proof

assume *d: distinct vs* **and** *b1: before vs ram2 ram1* **and** *b2: before vs ram1 ram2*

from *b2* **have** *r1: ram1* $\in \text{set } vs$ **by** (*drule-tac before-r1*)

from *d b1* **have** *r2: ram2* $\in \text{set } (\text{fst } (\text{splitAt } ram1 \text{ vs}))$ **by** (*rule before-dist-r1*)

from *d b2* **have** *r2':ram2* $\in \text{set } (\text{snd } (\text{splitAt } ram1 \text{ vs}))$ **by** (*rule before-dist-r2*)

from *d r1 r2 r2'* **show** *False* **by** (*drule-tac splitAt-distinct-fst-snd*) *auto*

qed

lemma *before-dist-not2*:

distinct vs $\implies ram1 \in \text{set } vs \implies ram2 \in \text{set } vs \implies ram1 \neq ram2 \implies \neg (\text{before } vs \ ram1 \ ram2) \implies \text{before } vs \ ram2 \ ram1$

proof –

assume *distinct vs ram1* $\in \text{set } vs$ *ram2* $\in \text{set } vs$ *ram1* $\neq ram2$ $\neg \text{before } vs \ ram1 \ ram2$

then show *before vs ram2 ram1* **apply** (*frule-tac before-or*) **by** *auto*

qed

lemma *before-dist-eq*:

distinct vs $\implies ram1 \in \text{set } vs \implies ram2 \in \text{set } vs \implies ram1 \neq ram2 \implies (\neg (\text{before } vs \ ram1 \ ram2)) = \text{before } vs \ ram2 \ ram1$

by (*auto intro: before-dist-not2 dest: before-dist-not1*)

lemma *before-vs*:

distinct vs $\implies \text{before } vs \ ram1 \ ram2 \implies vs = \text{fst } (\text{splitAt } ram1 \text{ vs}) @ ram1 \# \text{fst } (\text{splitAt } ram2 \text{ (snd } (\text{splitAt } ram1 \text{ vs}))) @ ram2 \# \text{snd } (\text{splitAt } ram2 \text{ vs})$

proof –

assume *d: distinct vs* **and** *b: before vs ram1 ram2*

define *s* **where** *s = snd (splitAt ram1 vs)*

from *b* **have** *ram1* $\in \text{set } vs$ **by** (*auto simp: before-def*)

with *s-def* **have** *vs: vs = fst (splitAt ram1 vs) @ [ram1] @ s* **by** (*auto dest: splitAt-ram*)

from *d b s-def* **have** *ram2* $\in \text{set } s$ **by** (*auto intro: before-dist-r2*)

then have *snd: s = fst (splitAt ram2 s) @ [ram2] @ snd (splitAt ram2 s)*

by (*auto dest: splitAt-ram*)

with *vs* **have** *vs = fst (splitAt ram1 vs) @ [ram1] @ fst (splitAt ram2 s) @ [ram2]*

@ *snd (splitAt ram2 s)* **by** *auto*

with *d b s-def* **show** *?thesis* **by** *auto*

qed

12.7 between

definition *pre-between* :: '*a* list \Rightarrow '*a* \Rightarrow '*a* \Rightarrow bool **where**

pre-between vs ram1 ram2 \equiv

$distinct\ vs \wedge ram1 \in set\ vs \wedge ram2 \in set\ vs \wedge ram1 \neq ram2$

declare *pre-between-def* [*simp*]

lemma *pre-between-dist*[*intro*]:
 $pre-between\ vs\ ram1\ ram2 \implies distinct\ vs$ **by** (*auto simp: pre-between-def*)

lemma *pre-between-r1*[*intro*]:
 $pre-between\ vs\ ram1\ ram2 \implies ram1 \in set\ vs$ **by auto**

lemma *pre-between-r2*[*intro*]:
 $pre-between\ vs\ ram1\ ram2 \implies ram2 \in set\ vs$ **by auto**

lemma *pre-between-r12*[*intro*]:
 $pre-between\ vs\ ram1\ ram2 \implies ram1 \neq ram2$ **by auto**

lemma *pre-between-symI*:
 $pre-between\ vs\ ram1\ ram2 \implies pre-between\ vs\ ram2\ ram1$ **by auto**

lemma *pre-between-before*[*dest*]:
 $pre-between\ vs\ ram1\ ram2 \implies before\ vs\ ram1\ ram2 \vee before\ vs\ ram2\ ram1$ **by**
(*rule-tac before-or*) *auto*

lemma *pre-between-rotate1*[*intro*]:
 $pre-between\ vs\ ram1\ ram2 \implies pre-between\ (rotate1\ vs)\ ram1\ ram2$ **by auto**

lemma *pre-between-rotate*[*intro*]:
 $pre-between\ vs\ ram1\ ram2 \implies pre-between\ (rotate\ n\ vs)\ ram1\ ram2$ **by auto**

lemma *pre-between vs ram1 ram2 \implies (\neg before vs ram1 ram2) = before vs ram2 ram1*
by (*simp add: before-dist-eq*)

declare *pre-between-def* [*simp del*]

lemma *between-simp1*[*simp*]:
 $before\ vs\ ram1\ ram2 \implies pre-between\ vs\ ram1\ ram2 \implies$
 $between\ vs\ ram1\ ram2 = fst\ (splitAt\ ram2\ (snd\ (splitAt\ ram1\ vs)))$
by (*simp add: pre-between-def between-def split-def before-dist-eq-snd*)

lemma *between-simp2*[*simp*]:
 $before\ vs\ ram1\ ram2 \implies pre-between\ vs\ ram1\ ram2 \implies$
 $between\ vs\ ram2\ ram1 = snd\ (splitAt\ ram2\ vs) @ fst\ (splitAt\ ram1\ vs)$

proof –

assume *b*: *before vs ram1 ram2* **and** *p*: *pre-between vs ram1 ram2*

from *p b* **have** *b2*: \neg *before vs ram2 ram1* **apply** (*simp add: pre-between-def*)

by (*auto dest: before-dist-not1*)

with *p* **have** $ram2 \notin set\ (fst\ (splitAt\ ram1\ vs))$ **by** (*simp add: pre-between-def*)

before-dist-eq-fst
then have $\text{fst } (\text{splitAt } \text{ram1 } \text{vs}) = \text{fst } (\text{splitAt } \text{ram2 } (\text{fst } (\text{splitAt } \text{ram1 } \text{vs})))$ **by**
(auto dest: splitAt-no-ram)
then have $\text{fst } (\text{splitAt } \text{ram2 } (\text{fst } (\text{splitAt } \text{ram1 } \text{vs}))) = \text{fst } (\text{splitAt } \text{ram1 } \text{vs})$ **by**
auto
with $b2$ b p **show** *?thesis* **apply** (*simp add: pre-between-def between-def split-def*)
by (*auto dest: before-dist-not-r1*)
qed

lemma *between-not-r1* [*intro*]:
 $\text{distinct } \text{vs} \implies \text{ram1} \notin \text{set } (\text{between } \text{vs } \text{ram1 } \text{ram2})$
proof (*cases pre-between vs ram1 ram2*)
assume d : *distinct vs*
case *True* **then have** p : *pre-between vs ram1 ram2* **by** *auto*
then show $\text{ram1} \notin \text{set } (\text{between } \text{vs } \text{ram1 } \text{ram2})$
proof (*cases before vs ram1 ram2*)
case *True* **with** d p **show** *?thesis* **by** (*auto del: notI*)
next
from p **have** $p2$: *pre-between vs ram2 ram1* **by** (*auto intro: pre-between-symI*)
case *False* **with** p **have** *before vs ram2 ram1* **by** *auto*
with d $p2$ **show** *?thesis* **by** (*auto del: notI*)
qed
next
assume d : *distinct vs*
case *False* **then have** p : \neg *pre-between vs ram1 ram2* **by** *auto*
then show *?thesis*
proof (*cases ram1 = ram2*)
case *True* **with** d **have** $h1$: $\text{ram2} \notin \text{set } (\text{snd } (\text{splitAt } \text{ram2 } \text{vs}))$ **by** (*auto del: notI*)
from *True* d **have** $h2$: $\text{ram2} \notin \text{set } (\text{fst } (\text{splitAt } \text{ram2 } (\text{fst } (\text{splitAt } \text{ram2 } \text{vs}))))$
by (*auto del: notI*)
with *True* d $h1$ **show** *?thesis* **by** (*auto simp: between-def split-def*)
next
case *False* **then have** neq : $\text{ram1} \neq \text{ram2}$ **by** *auto*
then show *?thesis*
proof (*cases ram1 \notin set vs*)
case *True* **with** d **show** *?thesis* **by** (*auto dest: splitAt-no-ram splitAt-in-fst simp: between-def split-def*)
next
case *False* **then have** $r1in$: $\text{ram1} \in \text{set } \text{vs}$ **by** *auto*
then show *?thesis*
proof (*cases ram2 \notin set vs*)
from d **have** $h1$: $\text{ram1} \notin \text{set } (\text{fst } (\text{splitAt } \text{ram1 } \text{vs}))$ **by** (*auto del: notI*)
case *True* **with** d $h1$ **show** *?thesis*
by (*auto dest: splitAt-not1 splitAt-in-fst splitAt-ram splitAt-no-ram simp: between-def split-def del: notI*)
next
case *False* **then have** $r2in$: $\text{ram2} \in \text{set } \text{vs}$ **by** *auto*
with d neq $r1in$ **have** *pre-between vs ram1 ram2*

```

    by (auto simp: pre-between-def)
  with p show ?thesis by auto
qed
qed
qed
qed

lemma between-not-r2[intro]:
  distinct vs  $\implies$  ram2  $\notin$  set (between vs ram1 ram2)
proof (cases pre-between vs ram1 ram2)
  assume d: distinct vs
  case True then have p: pre-between vs ram1 ram2 by auto
  then show ram2  $\notin$  set (between vs ram1 ram2)
  proof (cases before vs ram1 ram2)
    from d have ram2  $\notin$  set (fst (splitAt ram2 vs)) by (auto del: notI)
    then have h1: ram2  $\notin$  set (snd (splitAt ram1 (fst (splitAt ram2 vs))))
      by (auto dest: splitAt-in-fst)
    case True with d p h1 show ?thesis by (auto del: notI)
  next
    from p have p2: pre-between vs ram2 ram1 by (auto intro: pre-between-symI)
    case False with p have before vs ram2 ram1 by auto
    with d p2 show ?thesis by (auto del: notI)
  qed
next
  assume d: distinct vs
  case False then have p:  $\neg$  pre-between vs ram1 ram2 by auto
  then show ?thesis
  proof (cases ram1 = ram2)
    case True with d have h1: ram2  $\notin$  set (snd (splitAt ram2 vs)) by (auto del:
notI)
    from True d have h2: ram2  $\notin$  set (fst (splitAt ram2 (fst (splitAt ram2 vs))))
  by (auto del: notI)
  with True d h1 show ?thesis by (auto simp: between-def split-def)
  next
    case False then have neq: ram1  $\neq$  ram2 by auto
    then show ?thesis
    proof (cases ram2  $\notin$  set vs)
      case True with d show ?thesis
      by (auto dest: splitAt-no-ram splitAt-in-fst
splitAt-in-fst simp: between-def split-def)
    next
      case False then have r1in: ram2  $\in$  set vs by auto
      then show ?thesis
      proof (cases ram1  $\notin$  set vs)
        from d have h1: ram1  $\notin$  set (fst (splitAt ram1 vs)) by (auto del: notI)
        case True with d h1 show ?thesis by (auto dest: splitAt-ram splitAt-no-ram
simp: between-def split-def del: notI)
      next
        case False then have r2in: ram1  $\in$  set vs by auto

```

with $d \text{ neq } r1$ **in have** $\text{pre-between } vs \text{ ram1 ram2}$ **by** ($\text{auto simp: pre-between-def}$)
with p **show** $?thesis$ **by auto**
qed
qed
qed
qed

lemma $\text{between-distinct[intro]}$:

$\text{distinct } vs \implies \text{distinct } (\text{between } vs \text{ ram1 ram2})$

proof –

assume vs : $\text{distinct } vs$

define a **where** $a = \text{fst } (\text{splitAt } \text{ram1 } vs)$

define b **where** $b = \text{snd } (\text{splitAt } \text{ram1 } vs)$

from $a\text{-def } b\text{-def}$ **have** ab : $(a,b) = \text{splitAt } \text{ram1 } vs$ **by auto**

with vs **have** $ab\text{-disj}$: $\text{set } a \cap \text{set } b = \{\}$ **by** ($\text{drule-tac splitAt-distinct-ab}$) auto

define c **where** $c = \text{fst } (\text{splitAt } \text{ram2 } a)$

define d **where** $d = \text{snd } (\text{splitAt } \text{ram2 } a)$

from $c\text{-def } d\text{-def}$ **have** $c\text{-d}$: $(c,d) = \text{splitAt } \text{ram2 } a$ **by auto**

with $ab\text{-disj}$ **have** $\text{set } c \cap \text{set } b = \{\}$ **by** ($\text{drule-tac splitAt-subset-ab}$) auto

with vs $a\text{-def } b\text{-def } c\text{-def}$ **show** $?thesis$

by ($\text{auto simp: between-def split-def splitAt-no-ram dest: splitAt-ram intro: splitAt-distinct-fst splitAt-distinct-snd}$)

qed

lemma $\text{between-distinct-r12}$:

$\text{distinct } vs \implies \text{ram1} \neq \text{ram2} \implies \text{distinct } (\text{ram1} \# \text{between } vs \text{ ram1 ram2} \text{ @ } [\text{ram2}])$ **by** (auto del: notI)

lemma between-vs :

$\text{before } vs \text{ ram1 ram2} \implies \text{pre-between } vs \text{ ram1 ram2} \implies$

$vs = \text{fst } (\text{splitAt } \text{ram1 } vs) \text{ @ } \text{ram1} \# (\text{between } vs \text{ ram1 ram2}) \text{ @ } \text{ram2} \# \text{snd } (\text{splitAt } \text{ram2 } vs)$

apply (simp) **apply** ($\text{frule pre-between-dist}$) **apply** (drule before-vs) **by auto**

lemma between-in :

$\text{before } vs \text{ ram1 ram2} \implies \text{pre-between } vs \text{ ram1 ram2} \implies x \in \text{set } vs \implies x = \text{ram1} \vee x \in \text{set } (\text{between } vs \text{ ram1 ram2}) \vee x = \text{ram2} \vee x \in \text{set } (\text{between } vs \text{ ram2 ram1})$

proof –

assume b : $\text{before } vs \text{ ram1 ram2}$ **and** p : $\text{pre-between } vs \text{ ram1 ram2}$ **and** xin : $x \in \text{set } vs$

define a **where** $a = \text{fst } (\text{splitAt } \text{ram1 } vs)$

define b **where** $b = \text{between } vs \text{ ram1 ram2}$

define c **where** $c = \text{snd } (\text{splitAt } \text{ram2 } vs)$

from p **have** $\text{distinct } vs$ **by auto**

from p $a\text{-def } b\text{-def } c\text{-def}$ **have** $vs = a \text{ @ } \text{ram1} \# b \text{ @ } \text{ram2} \# c$ **apply** ($\text{drule-tac between-vs}$) **by auto**

with xin **have** $x \in \text{set } (a \text{ @ } \text{ram1} \# b \text{ @ } \text{ram2} \# c)$ **by auto**

then **have** $x \in \text{set } (a) \vee x \in \text{set } (\text{ram1} \# b) \vee x \in \text{set } (\text{ram2} \# c)$ **by auto**

then **have** $x \in \text{set } (a) \vee x = \text{ram1} \vee x \in \text{set } b \vee x = \text{ram2} \vee x \in \text{set } c$ **by auto**

then have $x \in \text{set } c \vee x \in \text{set } (a) \vee x = \text{ram1} \vee x \in \text{set } b \vee x = \text{ram2}$ **by auto**
then have $x \in \text{set } (c @ a) \vee x = \text{ram1} \vee x \in \text{set } b \vee x = \text{ram2}$ **by auto**
with b p a -def b -def c -def **show** $?thesis$ **by auto**
qed

lemma

$\text{before } vs \text{ ram1 ram2} \implies \text{pre-between } vs \text{ ram1 ram2} \implies$
 $\text{hd } vs \neq \text{ram1} \implies (a,b) = \text{splitAt } (\text{hd } vs) (\text{between } vs \text{ ram2 ram1}) \implies$
 $vs = [\text{hd } vs] @ b @ [\text{ram1}] @ (\text{between } vs \text{ ram1 ram2}) @ [\text{ram2}] @ a$

proof –

assume b : $\text{before } vs \text{ ram1 ram2}$ **and** p : $\text{pre-between } vs \text{ ram1 ram2}$ **and** vs : $\text{hd } vs \neq \text{ram1}$ **and** ab : $(a,b) = \text{splitAt } (\text{hd } vs) (\text{between } vs \text{ ram2 ram1})$
from p **have** dist-b : $\text{distinct } (\text{between } vs \text{ ram2 ram1})$ **by** ($\text{auto intro: between-distinct simp: pre-between-def}$)

with ab **have** $\text{distinct } a \wedge \text{distinct } b$ **by** ($\text{auto intro: splitAt-distinct-a splitAt-distinct-b}$)

define r **where** $r = \text{snd } (\text{splitAt } \text{ram1 } vs)$

define btw **where** $btw = \text{between } vs \text{ ram2 ram1}$

from p r -def **have** $vs2$: $vs = \text{fst } (\text{splitAt } \text{ram1 } vs) @ [\text{ram1}] @ r$ **by** ($\text{auto dest: splitAt-ram simp: pre-between-def}$)

then have $\text{fst } (\text{splitAt } \text{ram1 } vs) = [] \implies \text{hd } vs = \text{ram1}$ **by auto**

with vs **have** neq : $\text{fst } (\text{splitAt } \text{ram1 } vs) \neq []$ **by auto**

with $vs2$ **have** vs -fst: $\text{hd } vs = \text{hd } (\text{fst } (\text{splitAt } \text{ram1 } vs))$ **by** ($\text{induct } (\text{fst } (\text{splitAt } \text{ram1 } vs)))$ **auto**

with neq **have** $\text{hd } vs \in \text{set } (\text{fst } (\text{splitAt } \text{ram1 } vs))$ **by auto**

with b p **have** $\text{hd } vs \in \text{set } (\text{between } vs \text{ ram2 ram1})$ **by auto**

with btw -def **have** help1 : $btw = \text{fst } (\text{splitAt } (\text{hd } vs) btw) @ [\text{hd } vs] @ \text{snd } (\text{splitAt } (\text{hd } vs) btw)$ **by** ($\text{auto dest: splitAt-ram}$)

from p b btw -def **have** $btw = \text{snd } (\text{splitAt } \text{ram2 } vs) @ \text{fst } (\text{splitAt } \text{ram1 } vs)$ **by auto**

with neq **have** $btw = \text{snd } (\text{splitAt } \text{ram2 } vs) @ \text{hd } (\text{fst } (\text{splitAt } \text{ram1 } vs)) \# \text{tl } (\text{fst } (\text{splitAt } \text{ram1 } vs))$ **by auto**

with vs -fst **have** $btw = \text{snd } (\text{splitAt } \text{ram2 } vs) @ [\text{hd } vs] @ \text{tl } (\text{fst } (\text{splitAt } \text{ram1 } vs))$ **by auto**

with help1 **have** eq : $\text{snd } (\text{splitAt } \text{ram2 } vs) @ [\text{hd } vs] @ \text{tl } (\text{fst } (\text{splitAt } \text{ram1 } vs)) = \text{fst } (\text{splitAt } (\text{hd } vs) btw) @ [\text{hd } vs] @ \text{snd } (\text{splitAt } (\text{hd } vs) btw)$ **by auto**

from dist-b btw -def help1 **have** $\text{distinct } (\text{fst } (\text{splitAt } (\text{hd } vs) btw) @ [\text{hd } vs] @ \text{snd } (\text{splitAt } (\text{hd } vs) btw))$ **by auto**

with eq **have** eq2 : $\text{snd } (\text{splitAt } \text{ram2 } vs) = \text{fst } (\text{splitAt } (\text{hd } vs) btw) \wedge \text{tl } (\text{fst } (\text{splitAt } \text{ram1 } vs)) = \text{snd } (\text{splitAt } (\text{hd } vs) btw)$ **apply** (rule-tac dist-at) **by auto**

with btw -def ab **have** a : $a = \text{snd } (\text{splitAt } \text{ram2 } vs)$ **by** (auto dest: pairD)

from eq2 vs -fst **have** $\text{hd } (\text{fst } (\text{splitAt } \text{ram1 } vs)) \# \text{tl } (\text{fst } (\text{splitAt } \text{ram1 } vs)) = \text{hd } vs \# \text{snd } (\text{splitAt } (\text{hd } vs) btw)$ **by auto**

with ab btw -def neq **have** hdb : $\text{hd } vs \# b = \text{fst } (\text{splitAt } \text{ram1 } vs)$ **by** (auto dest: pairD)

from b p **have** $vs = \text{fst } (\text{splitAt } \text{ram1 } vs) @ [\text{ram1}] @ \text{fst } (\text{splitAt } \text{ram2 } (\text{snd } (\text{splitAt } \text{ram1 } vs))) @ [\text{ram2}] @ \text{snd } (\text{splitAt } \text{ram2 } vs)$ **apply** simp

apply ($\text{rule-tac before-vs}$) **by** ($\text{auto simp: pre-between-def}$)

with *hdb* **have** $vs = (hd\ vs\ \#\ b)\ @\ [ram1]\ @\ fst\ (splitAt\ ram2\ (snd\ (splitAt\ ram1\ vs)))\ @\ [ram2]\ @\ snd\ (splitAt\ ram2\ vs)\ \mathbf{by}\ auto$
with *a b p* **show** *?thesis* **by** (*simp*)
qed

lemma *between-congs*: $pre\text{-}between\ vs\ ram1\ ram2 \implies vs \cong vs' \implies between\ vs\ ram1\ ram2 = between\ vs'\ ram1\ ram2$

proof –
have $\bigwedge\ us.\ pre\text{-}between\ us\ ram1\ ram2 \implies before\ us\ ram1\ ram2 \implies between\ us\ ram1\ ram2 = between\ (rotate1\ us)\ ram1\ ram2$
proof –
fix *us*
assume *vors*: $pre\text{-}between\ us\ ram1\ ram2\ before\ us\ ram1\ ram2$
then **have** *pb2*: $pre\text{-}between\ (rotate1\ us)\ ram1\ ram2\ \mathbf{by}\ auto$
with *vors* **show** $between\ us\ ram1\ ram2 = between\ (rotate1\ us)\ ram1\ ram2$
proof (*cases us*)
case *Nil* **then** **show** *?thesis* **by** *auto*
next
case (*Cons u' us'*)
with *vors pb2* **show** *?thesis* **apply** (*auto simp: before-def*)
apply (*case-tac a*) **apply** *auto*
by (*simp-all add: between-def split-def pre-between-def*)
qed
qed

moreover **have** $\bigwedge\ us.\ pre\text{-}between\ us\ ram1\ ram2 \implies before\ us\ ram2\ ram1 \implies between\ us\ ram1\ ram2 = between\ (rotate1\ us)\ ram1\ ram2$

proof –
fix *us*
assume *vors*: $pre\text{-}between\ us\ ram1\ ram2\ before\ us\ ram2\ ram1$
then **have** *pb2*: $pre\text{-}between\ (rotate1\ us)\ ram1\ ram2\ \mathbf{by}\ auto$
with *vors* **show** $between\ us\ ram1\ ram2 = between\ (rotate1\ us)\ ram1\ ram2$
proof (*cases us*)
case *Nil* **then** **show** *?thesis* **by** *auto*
next
case (*Cons u' us'*)
with *vors pb2* **show** *?thesis* **apply** (*auto simp: before-def*)
apply (*case-tac a*) **apply** *auto*
by (*simp-all add: between-def split-def pre-between-def*)
qed
qed

ultimately **have** *help*: $\bigwedge\ us.\ pre\text{-}between\ us\ ram1\ ram2 \implies between\ us\ ram1\ ram2 = between\ (rotate1\ us)\ ram1\ ram2$

apply (*subgoal-tac before us ram1 ram2 \vee before us ram2 ram1*) **by** *auto*

assume $vs \cong vs'$ **and** *pre-b*: $pre\text{-}between\ vs\ ram1\ ram2$
then **obtain** *n* **where** $vs': vs' = rotate\ n\ vs\ \mathbf{by}\ (auto\ simp:\ congs\text{-}def)$
have $between\ vs\ ram1\ ram2 = between\ (rotate\ n\ vs)\ ram1\ ram2$


```

proof (induct n)
  case 0 then show ?case by auto
next
  case (Suc m) then show ?case apply simp
  apply (subgoal-tac between (rotate1 (rotate m vs)) ram1 ram2 = between
(rotate m vs) ram1 ram2)
  by (auto intro: help [symmetric] pre-b)
qed
with vs' show ?thesis by auto
qed

```

lemma *between-inter-empty*:

```

pre-between vs ram1 ram2  $\implies$ 
  set (between vs ram1 ram2)  $\cap$  set (between vs ram2 ram1) = {}
apply (case-tac before vs ram1 ram2)
apply (simp add: pre-between-def)
apply (elim conjE)
apply (frule (1) before-vs)
apply (subgoal-tac distinct (fst (splitAt ram1 vs) @
ram1 # fst (splitAt ram2 (snd (splitAt ram1 vs))) @ ram2 # snd (splitAt
ram2 vs)))
apply (thin-tac vs = fst (splitAt ram1 vs) @
ram1 # fst (splitAt ram2 (snd (splitAt ram1 vs))) @ ram2 # snd (splitAt
ram2 vs))
apply (frule (1) before-dist-fst-snd)
apply (simp)
apply blast
apply (simp only:)
apply (simp add: before-xor)
apply (subgoal-tac pre-between vs ram2 ram1)
apply (simp add: pre-between-def)
apply (elim conjE)
apply (frule (1) before-vs)
apply (subgoal-tac distinct (fst (splitAt ram2 vs) @
ram2 # fst (splitAt ram1 (snd (splitAt ram2 vs))) @ ram1 # snd (splitAt
ram1 vs)))
apply (thin-tac vs = fst (splitAt ram2 vs) @
ram2 # fst (splitAt ram1 (snd (splitAt ram2 vs))) @ ram1 # snd (splitAt
ram1 vs))
apply (simp)
apply blast
apply (simp only:)
by (rule pre-between-symI)

```

12.7.1 *between is-nextElem*

lemma *is-nextElem-or1*: $pre\text{-}between\ vs\ ram1\ ram2 \implies$
 $is\text{-}nextElem\ vs\ x\ y \implies before\ vs\ ram1\ ram2 \implies$
 $is\text{-}sublist\ [x,y]\ (ram1\ \# \textit{between}\ vs\ ram1\ ram2\ @\ [ram2])$

```

∨ is-sublist [x,y] (ram2 # between vs ram2 ram1 @ [ram1])
proof –
  assume p: pre-between vs ram1 ram2 and is-nextElem: is-nextElem vs x y and
  b: before vs ram1 ram2
  from p have r1: ram1 ∈ set vs by (auto simp: pre-between-def)
  define bs where bs = [ram1] @ (between vs ram1 ram2) @ [ram2]
  have rule1: x ∈ set (ram1 # (between vs ram1 ram2)) ⇒ is-sublist [x,y] bs
  proof –
    assume xin:x ∈ set (ram1 # (between vs ram1 ram2))
    with bs-def have xin2: x ∈ set bs by auto
    define s where s = snd (splitAt ram1 vs)
    from r1 s-def have sub1:is-sublist (ram1 # s) vs by (auto intro: splitAt-is-sublist2R)
    from b p s-def have ram2 ∈ set s by (auto intro!: before-dist-r2 simp:
pre-between-def)
    then have is-prefix (fst (splitAt ram2 s) @ [ram2]) s by (auto intro!: splitAt-is-prefix)
    then have is-prefix ([ram1] @ ((fst (splitAt ram2 s)) @ [ram2])) ([ram1] @ s)
by (rule-tac is-prefix-add) auto
    with sub1 have is-sublist (ram1 # (fst (splitAt ram2 s)) @ [ram2]) vs apply
(rule-tac is-sublist-trans) apply (rule is-prefix-sublist)
    by simp-all
    with p b s-def bs-def have subl: is-sublist bs vs by (auto)
    with p have db: distinct bs by (auto simp: pre-between-def)
    with xin bs-def have xnlb:x ≠ last bs by auto
    with p is-nextElem subl xin2 show is-sublist [x,y] bs apply (rule-tac is-sublist-is-nextElem)
by (auto simp: pre-between-def)
  qed
  define bs2 where bs2 = [ram2] @ (between vs ram2 ram1) @ [ram1]
  have rule2: x ∈ set (ram2 # (between vs ram2 ram1)) ⇒ is-sublist [x,y] bs2
  proof –
    assume xin:x ∈ set (ram2 # (between vs ram2 ram1))
    with bs2-def have xin2: x ∈ set bs2 by auto
    define cs1 where cs1 = ram2 # snd (splitAt ram2 vs)
    then have cs1ne: cs1 ≠ [] by auto
    define cs2 where cs2 = fst (splitAt ram1 vs)
    define cs2ram1 where cs2ram1 = cs2 @ [ram1]
    from cs1-def cs2-def bs2-def p b have bs2: bs2 = cs1 @ cs2 @ [ram1] by (auto
simp: pre-between-def)
    have x ∈ set cs1 ⇒ x ≠ last cs1 ⇒ is-sublist [x,y] cs1
    proof–
      assume xin2: x ∈ set cs1 and xnlcs1: x ≠ last cs1
      from cs1-def p have is-sublist cs1 vs by (simp add: pre-between-def)
      with p is-nextElem xnlcs1 xin2 show ?thesis apply (rule-tac is-sublist-is-nextElem)
by (auto simp: pre-between-def)
    qed
    with bs2 have incs1nl: x ∈ set cs1 ⇒ x ≠ last cs1 ⇒ is-sublist [x,y] bs2
    apply (auto simp: is-sublist-def) apply (intro exI)
    apply (subgoal-tac as @ x # y # bs @ cs2 @ [ram1] = as @ x # y # (bs @
cs2 @ [ram1]))

```

```

    apply assumption by auto
  have x = last cs1  $\implies$  y = hd (cs2 @ [ram1])
  proof -
    assume xl: x = last cs1
    from p have distinct vs by auto
    with p have vs = fst (splitAt ram2 vs) @ ram2 # snd (splitAt ram2 vs) by
(auto intro: splitAt-ram)
    with cs1-def have last vs = last (fst (splitAt ram2 vs) @ cs1) by auto
    with cs1ne have last vs = last cs1 by auto
    with xl have x = last vs by auto
    with p is-nextElem have yhd: y = hd vs by auto
    from p have vs = fst (splitAt ram1 vs) @ ram1 # snd (splitAt ram1 vs) by
(auto intro: splitAt-ram)
    with yhd cs2ram1-def cs2-def have yhd2: y = hd (cs2ram1 @ snd (splitAt
ram1 vs)) by auto
    from cs2ram1-def have cs2ram1  $\neq$  [] by auto
    then have hd (cs2ram1 @ snd(splitAt ram1 vs)) = hd (cs2ram1) by auto
    with yhd2 cs2ram1-def show ?thesis by auto
  qed
  with bs2 cs1ne have x = last cs1  $\implies$  is-sublist [x,y] bs2
  apply (auto simp: is-sublist-def) apply (intro exI)
  apply (subgoal-tac cs1 @ cs2 @ [ram1] = butlast cs1 @ last cs1 # hd (cs2
@ [ram1]) # tl (cs2 @ [ram1]))
  apply assumption by auto
  with incs1nl have incs1: x  $\in$  set cs1  $\implies$  is-sublist [x,y] bs2 by auto
  have x  $\in$  set cs2  $\implies$  is-sublist [x,y] (cs2 @ [ram1])
  proof -
    assume xin2': x  $\in$  set cs2
    then have xin2: x  $\in$  set (cs2 @ [ram1]) by auto
    define fr2 where fr2 = snd (splitAt ram1 vs)
    from p have vs = fst (splitAt ram1 vs) @ ram1 # snd (splitAt ram1 vs) by
(auto intro: splitAt-ram)
    with fr2-def cs2-def have vs = cs2 @ [ram1] @ fr2 by simp
    with p xin2' have x  $\neq$  ram1 by (auto simp: pre-between-def)
    then have xncls2: x  $\neq$  last (cs2 @ [ram1]) by auto
    from cs2-def p have is-sublist (cs2 @ [ram1]) vs by (simp add: pre-between-def)
    with p is-nextElem xncls2 xin2 show ?thesis apply (rule-tac is-sublist-is-nextElem)
by (auto simp: pre-between-def)
  qed
  with bs2 have x  $\in$  set cs2  $\implies$  is-sublist [x,y] bs2
  apply (auto simp: is-sublist-def) apply (intro exI)
  apply (subgoal-tac cs1 @ as @ x # y # bs = (cs1 @ as) @ x # y # bs)
  apply assumption by auto
  with p b cs1-def cs2-def incs1 xin show ?thesis by auto
  qed
  from is-nextElem have x  $\in$  set vs by auto
  with b p have x = ram1  $\vee$  x  $\in$  set (between vs ram1 ram2)  $\vee$  x = ram2  $\vee$  x  $\in$ 
set (between vs ram2 ram1) by (rule-tac between-in) auto
  then have x  $\in$  set (ram1 # (between vs ram1 ram2))  $\vee$  x  $\in$  set (ram2 # (between

```

```

vs ram2 ram1)) by auto
  with rule1 rule2 bs-def bs2-def show ?thesis by auto
qed

```

```

lemma is-nextElem-or: pre-between vs ram1 ram2  $\implies$  is-nextElem vs x y  $\implies$ 
  is-sublist [x,y] (ram1 # between vs ram1 ram2 @ [ram2])  $\vee$  is-sublist [x,y] (ram2
# between vs ram2 ram1 @ [ram1])
proof (cases before vs ram1 ram2)
  case True
  assume pre-between vs ram1 ram2 is-nextElem vs x y
  with True show ?thesis by (rule-tac is-nextElem-or1)
next
  assume p: pre-between vs ram1 ram2 and is-nextElem: is-nextElem vs x y
  from p have p': pre-between vs ram2 ram1 by (auto intro: pre-between-symI)
  case False with p have before vs ram2 ram1 by auto
  with p' is-nextElem show ?thesis apply (drule-tac is-nextElem-or1) apply as-
sumption+ by auto
qed

```

```

lemma pre-between vs ram1 ram2  $\implies$ 
  before vs ram2 ram1  $\implies$ 
   $\exists$  as bs cs. between vs ram1 ram2 = cs @ as  $\wedge$  vs = as @[ram2] @ bs @ [ram1]
@ cs
  apply (subgoal-tac pre-between vs ram2 ram1)
  apply auto apply (intro exI conjI) apply simp apply (simp add: pre-between-def)
apply auto
  apply (frule-tac before-vs) apply auto by (auto simp: pre-between-def)

```

```

lemma is-sublist-same-len[simp]:
  length xs = length ys  $\implies$  is-sublist xs ys = (xs = ys)
apply (cases xs)
  apply simp
apply (rename-tac a as)
apply (cases ys)
  apply simp
apply (rename-tac b bs)
apply (case-tac a = b)
  apply (subst is-sublist-rec)
  apply simp
apply simp
done

```

```

lemma is-nextElem-between-empty[simp]:
  distinct vs  $\implies$  is-nextElem vs a b  $\implies$  between vs a b = []
apply (simp add: is-nextElem-def between-def split-def)
apply (cases vs) apply simp+

```

```

apply (simp split: if-split-asm)
apply (case-tac b = aa)
apply (simp add: is-sublist-def)
apply (erule disjE)
apply (elim exE)
apply (case-tac as)
apply simp
apply simp
apply simp
apply (case-tac list rule: rev-exhaust)
apply simp
apply simp
apply simp
apply (subgoal-tac aa # list = vs)
apply (thin-tac vs = aa # list)
apply simp
apply (subgoal-tac distinct vs)
apply (simp add: is-sublist-def)
apply (elim exE)
by auto

```

lemma *is-nextElem-between-empty'*: $between\ vs\ a\ b = [] \implies distinct\ vs \implies a \in set\ vs \implies b \in set\ vs \implies$

```

 $a \neq b \implies is-nextElem\ vs\ a\ b$ 
apply (simp add: is-nextElem-def between-def split-def split: if-split-asm)
apply (case-tac vs) apply simp
apply simp
apply (rule conjI)
apply (rule impI)
apply simp
apply (case-tac aa = b)
apply simp
apply (rule impI)
apply (case-tac list rule: rev-exhaust)
apply simp
apply simp
apply (case-tac a = y) apply simp
apply simp
apply (elim conjE)
apply (drule split-list-first)
apply (elim exE)
apply simp
apply (rule impI)
apply (subgoal-tac b # aa)
apply simp
apply (case-tac a = aa)
apply simp
apply (case-tac list) apply simp

```

```

apply simp
apply (case-tac aaa = b) apply (simp add: is-sublist-def) apply force
apply simp
apply simp
apply (drule split-list-first)
apply (elim exE)
apply simp
apply (case-tac zs) apply simp
apply simp
apply (case-tac aaa = b)
apply simp
apply (simp add: is-sublist-def) apply force
apply simp
apply force
apply (case-tac vs) apply simp
apply simp
apply (rule conjI)
apply (rule impI) apply simp
apply (rule impI)
apply (case-tac b = aa)
apply simp
apply (case-tac list rule: rev-exhaust) apply simp
apply simp
apply (case-tac a = y) apply simp
apply simp
apply (drule split-list-first)
apply (elim exE)
apply simp
apply simp apply (case-tac a = aa) by auto

```

```

lemma between-nextElem: pre-between vs u v  $\implies$ 
  between vs u (nextElem vs (hd vs) v) = between vs u v @ [v]
apply(subgoal-tac pre-between vs v u)
prefer 2 apply (clarsimp simp add:pre-between-def)
apply(case-tac before vs u v)
apply(drule (1) between-eq2)
  apply(clarsimp simp:pre-between-def hd-append split:list.split)
  apply(simp add:between-def split-def)
  apply(fastforce simp:neq-Nil-conv)
apply (simp only:before-xor)
apply(drule (1) between-eq2)
apply(clarsimp simp:pre-between-def hd-append split:list.split)
apply (simp add: append-eq-Cons-conv)
apply(fastforce simp:between-def split-def)
done

```

lemma *nextVertices-in-face*[simp]: $v \in \mathcal{V} f \implies f^n \cdot v \in \mathcal{V} f$
proof –
assume $v: v \in \mathcal{V} f$
then have $f: \text{vertices } f \neq []$ **by** *auto*
show *?thesis* **apply** (*simp add: nextVertices-def*)
apply (*induct n*) **by** (*auto simp: f v*)
qed

12.7.2 *is-nextElem* edges equivalence

lemma *is-nextElem-edges1*: $\text{distinct } (\text{vertices } f) \implies (a,b) \in \text{edges } (f::\text{face}) \implies \text{is-nextElem } (\text{vertices } f) a b$ **apply** (*simp add: edges-face-def nextVertex-def*) **apply** (*rule is-nextElem1*) **by** *auto*

lemma *is-nextElem-edges2*:
 $\text{distinct } (\text{vertices } f) \implies \text{is-nextElem } (\text{vertices } f) a b \implies (a,b) \in \text{edges } (f::\text{face})$
apply (*auto simp add: edges-face-def nextVertex-def*)
apply (*rule sym*)
apply (*rule is-nextElem2*) **by** (*auto intro: is-nextElem-a*)

lemma *is-nextElem-edges-eq*[simp]:
 $\text{distinct } (\text{vertices } (f::\text{face})) \implies (a,b) \in \text{edges } f = \text{is-nextElem } (\text{vertices } f) a b$
by (*auto intro: is-nextElem-edges1 is-nextElem-edges2*)

12.7.3 *nextVertex*

lemma *nextElem-suc2*: $\text{distinct } (\text{vertices } f) \implies \text{last } (\text{vertices } f) = v \implies v \in \text{set } (\text{vertices } f) \implies f \cdot v = \text{hd } (\text{vertices } f)$

proof –
assume *dist*: $\text{distinct } (\text{vertices } f)$ **and** *last*: $\text{last } (\text{vertices } f) = v$ **and** $v: v \in \text{set } (\text{vertices } f)$
define *ls* **where** $ls = \text{vertices } f$
have *ind*: $\bigwedge c. \text{distinct } ls \implies \text{last } ls = v \implies v \in \text{set } ls \implies \text{nextElem } ls c v = c$
proof (*induct ls*)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons m ms*)
then show *?case*
proof (*cases m = v*)
case *True* **with** *Cons* **have** $ms = []$ **apply** (*cases ms rule: rev-exhaust*) **by** *auto*
then show *?thesis* **by** *auto*
next
case *False* **with** *Cons* **have** $a1: v \in \text{set } ms$ **by** *auto*
then have $ms: ms \neq []$ **by** *auto*

```

    with False Cons ms have nextElem ms c v = c apply (rule-tac Cons) by
simp-all
    with False ms show ?thesis by simp
    qed
    qed
    from dist ls-def last v have nextElem ls (hd ls) v = hd ls apply (rule-tac ind)
by auto
    with ls-def show ?thesis by (simp add: nextVertex-def)
    qed

```

12.8 split-face

```

definition pre-split-face :: face  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat list  $\Rightarrow$  bool where
pre-split-face oldF ram1 ram2 newVertexList  $\equiv$ 
  distinct (vertices oldF)  $\wedge$  distinct (newVertexList)
 $\wedge$   $\mathcal{V}$  oldF  $\cap$  set newVertexList = {}
 $\wedge$  ram1  $\in$   $\mathcal{V}$  oldF  $\wedge$  ram2  $\in$   $\mathcal{V}$  oldF  $\wedge$  ram1  $\neq$  ram2

```

```

declare pre-split-face-def [simp]

```

```

lemma pre-split-face-p-between[intro]:
pre-split-face oldF ram1 ram2 newVertexList  $\implies$  pre-between (vertices oldF) ram1
ram2 by (simp add: pre-between-def)

```

```

lemma pre-split-face-symI:
pre-split-face oldF ram1 ram2 newVertexList  $\implies$  pre-split-face oldF ram2 ram1
newVertexList by auto

```

```

lemma pre-split-face-rev[intro]:
pre-split-face oldF ram1 ram2 newVertexList  $\implies$  pre-split-face oldF ram1 ram2
(rev newVertexList) by auto

```

```

lemma split-face-distinct1:
(f12, f21) = split-face oldF ram1 ram2 newVertexList  $\implies$  pre-split-face oldF
ram1 ram2 newVertexList  $\implies$ 
  distinct (vertices f12)

```

```

proof –
  assume split: (f12, f21) = split-face oldF ram1 ram2 newVertexList and p:
pre-split-face oldF ram1 ram2 newVertexList
  define old-vs where old-vs = vertices oldF
  with p have d-old-vs: distinct old-vs by auto
  from p have p2: pre-between (vertices oldF) ram1 ram2 by auto
  have rule1: before old-vs ram1 ram2  $\implies$  distinct (vertices f12)
  proof –
    assume b: before old-vs ram1 ram2
    with split p have f12 = Face (rev newVertexList @ ram1 # between (vertices

```



```

oldF) ram1 ram2 @ [ram2]) Nonfinal by (simp add: split-face-def)
  then have h1:vertices f12 = rev newVertexList @ ram1 # between (vertices
oldF) ram1 ram2 @ [ram2] by auto
  from p have d1: distinct(ram1 # between (vertices oldF) ram1 ram2 @ [ram2])
by (auto del: notI)
  from b p p2 old-vs-def have d2: set (ram1 # between (vertices oldF) ram1
ram2 @ [ram2]) ∩ set newVertexList = {}
  by (auto dest!: splitAt-in-fst splitAt-in-snd)
  with h1 d1 p show ?thesis by auto
qed
have rule2: before old-vs ram2 ram1  $\implies$  distinct (vertices f12)
proof –
  assume b: before old-vs ram2 ram1
  from p have p3: pre-split-face oldF ram1 ram2 newVertexList
  by (auto intro: pre-split-face-symI)
  then have p4: pre-between (vertices oldF) ram2 ram1 by auto
  with split p have
    f12 = Face (rev newVertexList @ ram1 # between (vertices oldF) ram1 ram2
@ [ram2]) Nonfinal
  by (simp add: split-face-def)
  then have h1:vertices f12 = rev newVertexList @ ram1 # between (vertices
oldF) ram1 ram2 @ [ram2]
  by auto
  from p3 have d1: distinct(ram1 # between (vertices oldF) ram1 ram2 @
[ram2])
  by (auto del: notI)
  from b p3 p4 old-vs-def
  have d2: set (ram1 # between (vertices oldF) ram1 ram2 @ [ram2]) ∩ set
newVertexList = {}
  by auto
  with h1 d1 p show ?thesis by auto
qed
from p2 rule1 rule2 old-vs-def show ?thesis by auto
qed

```

```

lemma split-face-distinct1 '[intro]:
  pre-split-face oldF ram1 ram2 newVertexList  $\implies$ 
  distinct (vertices (fst(split-face oldF ram1 ram2 newVertexList)))
apply (rule-tac split-face-distinct1)
by (auto simp del: pre-split-face-def simp: split-face-def)

```

```

lemma split-face-distinct2:
  (f12, f21) = split-face oldF ram1 ram2 newVertexList  $\implies$ 
  pre-split-face oldF ram1 ram2 newVertexList  $\implies$  distinct (vertices f21)
proof –
  assume split: (f12, f21) = split-face oldF ram1 ram2 newVertexList
  and p: pre-split-face oldF ram1 ram2 newVertexList
  define old-vs where old-vs = vertices oldF
  with p have d-old-vs: distinct old-vs by auto

```

```

with  $p$  have  $p1$ : pre-split-face oldF ram1 ram2 newVertexList by auto
from  $p$  have  $p2$ : pre-between (vertices oldF) ram1 ram2 by auto
have  $rule1$ : before old-vs ram1 ram2  $\implies$  distinct (vertices f21)
proof –
  assume  $b$ : before old-vs ram1 ram2
  with split p
    have  $f21 = \text{Face } (ram2 \# \text{between } (vertices \text{oldF}) \text{ ram2 ram1 } @ [ram1] @$ 
newVertexList) Nonfinal
    by (simp add: split-face-def)
    then have  $h1$ :vertices f21 = ram2 # between (vertices oldF) ram2 ram1 @
[ram1] @ newVertexList
    by auto
    from  $p$  have  $d1$ : distinct(ram1 # between (vertices oldF) ram2 ram1 @ [ram2])
    by (auto del: notI)
    from  $b$   $p1$   $p2$  old-vs-def
    have  $d2$ : set (ram2 # between (vertices oldF) ram2 ram1 @ [ram1])  $\cap$  set
newVertexList = {}
    by auto
    with  $h1$   $d1$   $p1$  show ?thesis by auto
  qed
have  $rule2$ : before old-vs ram2 ram1  $\implies$  distinct (vertices f21)
proof –
  assume  $b$ : before old-vs ram2 ram1
  from  $p$  have  $p3$ : pre-split-face oldF ram1 ram2 newVertexList
  by (auto intro: pre-split-face-symI)
  then have  $p4$ : pre-between (vertices oldF) ram2 ram1 by auto
  with split p
    have  $f21 = \text{Face } (ram2 \# \text{between } (vertices \text{oldF}) \text{ ram2 ram1 } @ [ram1] @$ 
newVertexList) Nonfinal
    by (simp add: split-face-def)
    then have  $h1$ :vertices f21 = ram2 # between (vertices oldF) ram2 ram1 @
[ram1] @ newVertexList
    by auto
    from  $p3$  have  $d1$ : distinct(ram2 # between (vertices oldF) ram2 ram1 @
[ram1])
    by (auto del: notI)
    from  $b$   $p3$   $p4$  old-vs-def
    have  $d2$ : set (ram2 # between (vertices oldF) ram2 ram1 @ [ram1])  $\cap$  set
newVertexList = {}
    by auto
    with  $h1$   $d1$   $p1$  show ?thesis by auto
  qed
from  $p2$   $rule1$   $rule2$  old-vs-def show ?thesis by auto
qed

```

```

lemma split-face-distinct2'[intro]:
  pre-split-face oldF ram1 ram2 newVertexList  $\implies$  distinct (vertices (snd(split-face
oldF ram1 ram2 newVertexList)))
apply (rule-tac split-face-distinct2) by (auto simp del: pre-split-face-def simp:

```

split-face-def)

declare *pre-split-face-def* [*simp del*]

lemma *split-face-edges-or*: $(f12, f21) = \text{split-face oldF ram1 ram2 newVertexList} \implies \text{pre-split-face oldF ram1 ram2 newVertexList} \implies (a, b) \in \text{edges oldF} \implies (a, b) \in \text{edges f12} \vee (a, b) \in \text{edges f21}$

proof –

assume *nf*: $(f12, f21) = \text{split-face oldF ram1 ram2 newVertexList}$ **and** *p*: *pre-split-face oldF ram1 ram2 newVertexList* **and** *old*: $(a, b) \in \text{edges oldF}$

from *p* **have** *d1*: *distinct (vertices oldF)* **by** *auto*

from *nf p* **have** *d2*: *distinct (vertices f12)* **by** (*auto dest: pairD*)

from *nf p* **have** *d3*: *distinct (vertices f21)* **by** (*auto dest: pairD*)

from *p* **have** *p'*: *pre-between (vertices oldF) ram1 ram2* **by** *auto*

from *old d1* **have** *is-nextElem*: *is-nextElem (vertices oldF) a b* **by** *simp*

with *p* **have** *is-sublist [a,b] (ram1 # (between (vertices oldF) ram1 ram2) @ [ram2])* \vee *is-sublist [a,b] (ram2 # (between (vertices oldF) ram2 ram1) @ [ram1])*

apply (*rule-tac is-nextElem-or*) **by** *auto*

then **have** *is-nextElem (rev newVertexList @ ram1 # between (vertices oldF) ram1 ram2 @ [ram2]) a b*

\vee *is-nextElem (ram2 # between (vertices oldF) ram2 ram1 @ ram1 # newVertexList) a b*

proof (*cases is-sublist [a,b] (ram1 # (between (vertices oldF) ram1 ram2) @ [ram2])*)

case *True* **then** **show** *?thesis* **by** (*auto dest: is-sublist-add intro!: is-nextElem-sublistI*)

next

case *False*

assume *is-sublist [a,b] (ram1 # (between (vertices oldF) ram1 ram2) @ [ram2])*

\vee *is-sublist [a,b] (ram2 # (between (vertices oldF) ram2 ram1) @ [ram1])*

with *False* **have** *is-sublist [a,b] (ram2 # (between (vertices oldF) ram2 ram1) @ [ram1])* **by** *auto*

then **show** *?thesis* **apply** (*rule-tac disjI2*) **apply** (*rule-tac is-nextElem-sublistI*)

apply (*subgoal-tac is-sublist [a, b] ([] @ (ram2 # between (vertices oldF) ram2 ram1 @ [ram1]) @ newVertexList)*) **apply** *force* **by** (*frule is-sublist-add*)

qed

with *nf d1 d2 d3* **show** *?thesis* **by** (*simp add: split-face-def*)

qed

12.9 *verticesFrom*

definition *verticesFrom* :: *face* \Rightarrow *vertex* \Rightarrow *vertex list* **where**
verticesFrom f \equiv *rotate-to (vertices f)*

lemmas *verticesFrom-Def* = *verticesFrom-def rotate-to-def*

lemma *len-vFrom[simp]*:

$v \in \mathcal{V} f \implies |\text{verticesFrom } f \ v| = |\text{vertices } f|$

apply(*drule split-list-first*)

apply(*clarsimp simp: verticesFrom-Def*)
done

lemma *verticesFrom-empty[simp]*:
 $v \in \mathcal{V} f \implies (\text{verticesFrom } f \ v = []) = (\text{vertices } f = [])$
by(*clarsimp simp: verticesFrom-Def*)

lemma *verticesFrom-congs*:
 $v \in \mathcal{V} f \implies (\text{vertices } f) \cong (\text{verticesFrom } f \ v)$
by(*simp add: verticesFrom-def cong-rotate-to*)

lemma *verticesFrom-eq-if-vertices-cong*:
 $\llbracket \text{distinct}(\text{vertices } f); \text{distinct}(\text{vertices } f'); x \in \mathcal{V} f \rrbracket \implies$
 $\text{verticesFrom } f \ x = \text{verticesFrom } f' \ x$
by(*clarsimp simp: congs-def verticesFrom-Def splitAt-rotate-pair-conv*)

lemma *verticesFrom-in[intro]*: $v \in \mathcal{V} f \implies a \in \mathcal{V} f \implies a \in \text{set } (\text{verticesFrom } f \ v)$
by (*auto dest: verticesFrom-congs congs-pres-nodes*)

lemma *verticesFrom-in'*: $a \in \text{set } (\text{verticesFrom } f \ v) \implies a \neq v \implies a \in \mathcal{V} f$
apply (*cases v \in \mathcal{V} f*) **apply** (*auto dest: verticesFrom-congs congs-pres-nodes*)
by (*simp add: verticesFrom-Def*)

lemma *set-verticesFrom*:
 $v \in \mathcal{V} f \implies \text{set } (\text{verticesFrom } f \ v) = \mathcal{V} f$
apply(*auto*)
apply (*auto simp add: verticesFrom-Def*)
done

lemma *verticesFrom-hd*: $\text{hd } (\text{verticesFrom } f \ v) = v$ **by** (*simp add: verticesFrom-Def*)

lemma *verticesFrom-distinct[simp]*: $\text{distinct } (\text{vertices } f) \implies v \in \mathcal{V} f \implies \text{distinct } (\text{verticesFrom } f \ v)$ **apply** (*frule-tac verticesFrom-congs*) **by** (*auto simp: congs-distinct*)

lemma *verticesFrom-nextElem-eq*:
 $\text{distinct } (\text{vertices } f) \implies v \in \mathcal{V} f \implies u \in \mathcal{V} f \implies$
 $\text{nextElem } (\text{verticesFrom } f \ v) (\text{hd } (\text{verticesFrom } f \ v)) \ u$
 $= \text{nextElem } (\text{vertices } f) (\text{hd } (\text{vertices } f)) \ u$ **apply** (*subgoal-tac (verticesFrom } f \ v)*)
 $\cong (\text{vertices } f)$
apply (*rule nextElem-congs-eq*) **apply** (*auto simp: verticesFrom-congs congs-pres-nodes*)
apply (*rule congs-sym*)
by (*simp add: verticesFrom-congs*)

lemma *nextElem-vFrom-suc1*: $\text{distinct } (\text{vertices } f) \implies v \in \mathcal{V} f \implies i < \text{length } (\text{vertices } f) \implies \text{last } (\text{verticesFrom } f \ v) \neq u \implies (\text{verticesFrom } f \ v)!.i = u \implies f \cdot u = (\text{verticesFrom } f \ v)!(\text{Suc } i)$

proof –
assume $dist$: $distinct (vertices f)$ **and** ith : $(verticesFrom f v)!i = u$ **and** $small-i$:
 $i < length (vertices f)$ **and** $notlast$: $last (verticesFrom f v) \neq u$ **and** v : $v \in \mathcal{V} f$
hence eq : $(vertices f) \cong (verticesFrom f v)$ **by** $(auto simp: verticesFrom-congs)$
from $ith eq small-i$ **have** $u \in set (verticesFrom f v)$ **by** $(auto simp: congs-length)$
with eq **have** u : $u \in \mathcal{V} f$ **by** $(auto simp: congs-pres-nodes)$
define ls **where** $ls = verticesFrom f v$
with $dist ith$ **have** $ls!i = u$ **by** $auto$
have ind : $\bigwedge c i. i < length ls \implies distinct ls \implies last ls \neq u \implies ls!i = u \implies u$
 $\in set ls \implies$
 $nextElem ls c u = ls ! Suc i$
proof $(induct ls)$
case Nil **then show** $?case$ **by** $auto$
next
case $(Cons m ms)$
then show $?case$
proof $(cases m = u)$
case $True$ **with** $Cons$ **have** $u \notin set ms$ **by** $auto$
with $Cons True$ **have** i : $i = 0$ **apply** $(induct i)$ **by** $auto$
with $Cons$ **show** $?thesis$ **apply** $(simp split: if-split-asm)$ **apply** $(cases ms)$
by $simp-all$
next
case $False$ **with** $Cons$ **have** $a1$: $u \in set ms$ **by** $auto$
then have ms : $ms \neq []$ **by** $auto$
from $False Cons$ **have** i : $0 < i$ **by** $auto$
define i' **where** $i' = i - 1$
with i **have** i' : $i = Suc i'$ **by** $auto$
with $False Cons i' ms$ **have** $nextElem ms c u = ms ! Suc i'$ **apply** $(rule-tac$
 $Cons)$ **by** $simp-all$
with $False Cons i' ms$ **show** $?thesis$ **by** $simp$
qed
qed
from $eq dist ith ls-def small-i notlast v$
have $nextElem ls (hd ls) u = ls ! Suc i$
apply $(rule-tac ind)$ **by** $(auto simp: congs-length)$
with $dist u v ls-def$ **show** $?thesis$ **by** $(simp add: nextVertex-def verticesFrom-nextElem-eq)$
qed

lemma $verticesFrom-nth$: $distinct (vertices f) \implies d < length (vertices f) \implies$
 $v \in \mathcal{V} f \implies (verticesFrom f v)!d = f^d \cdot v$
proof $(induct d)$
case 0 **then show** $?case$ **by** $(simp add: verticesFrom-Def nextVertices-def)$
next
case $(Suc n)$
then have $dist2$: $distinct (verticesFrom f v)$
apply $(frule-tac verticesFrom-congs)$ **by** $(auto simp: congs-distinct)$
from Suc **have** $in2$: $v \in set (verticesFrom f v)$
apply $(frule-tac verticesFrom-congs)$ **by** $(auto dest: congs-pres-nodes)$
then have $vFrom$: $(verticesFrom f v) = butlast (verticesFrom f v) @ [last (verticesFrom$

```

f v)]
  apply (cases (verticesFrom f v) rule: rev-exhaust) by auto
  from Suc show ?case
  proof (cases last (verticesFrom f v) = f^n · v)
    case True with Suc have verticesFrom f v ! n = f^n · v by (rule-tac Suc) auto
    with True have last (verticesFrom f v) = verticesFrom f v ! n by auto
    with Suc dist2 in2 have Suc n = length (verticesFrom f v)
      apply (rule-tac nth-last-Suc-n) by auto
    with Suc show ?thesis by auto
  next
  case False with Suc show ?thesis apply (simp add: nextVertices-def) apply
(rule sym)
  apply (rule-tac nextElem-vFrom-suc1) by simp-all
  qed
qed

```

lemma *verticesFrom-length*: $\text{distinct } (\text{vertices } f) \implies v \in \text{set } (\text{vertices } f) \implies \text{length } (\text{verticesFrom } f v) = \text{length } (\text{vertices } f)$
by (*auto intro: congs-length verticesFrom-congs*)

lemma *verticesFrom-between*: $v' \in \mathcal{V} f \implies \text{pre-between } (\text{vertices } f) u v \implies \text{between } (\text{vertices } f) u v = \text{between } (\text{verticesFrom } f v') u v$
by (*auto intro!: between-congs verticesFrom-congs*)

lemma *verticesFrom-is-nextElem*: $v \in \mathcal{V} f \implies \text{is-nextElem } (\text{vertices } f) a b = \text{is-nextElem } (\text{verticesFrom } f v) a b$
apply (*rule is-nextElem-congs-eq*) **by** (*rule verticesFrom-congs*)

lemma *verticesFrom-is-nextElem-last*: $v' \in \mathcal{V} f \implies \text{distinct } (\text{vertices } f) \implies \text{is-nextElem } (\text{verticesFrom } f v') (\text{last } (\text{verticesFrom } f v')) v \implies v = v'$
apply (*subgoal-tac distinct (verticesFrom f v')*)
apply (*subgoal-tac last (verticesFrom f v') ∈ set (verticesFrom f v')*)
apply (*simp add: nextElem-is-nextElem*)
apply (*simp add: verticesFrom-hd*)
apply (*cases (verticesFrom f v') rule: rev-exhaust*) **apply** (*simp add: verticesFrom-Def*)
by *auto*

lemma *verticesFrom-is-nextElem-hd*: $v' \in \mathcal{V} f \implies \text{distinct } (\text{vertices } f) \implies \text{is-nextElem } (\text{verticesFrom } f v') u v' \implies u = \text{last } (\text{verticesFrom } f v')$
apply (*subgoal-tac distinct (verticesFrom f v')*)
apply (*thin-tac distinct (vertices f)*) **apply** (*auto simp: is-nextElem-def*)
apply (*drule is-sublist-y-hd*) **apply** (*simp add: verticesFrom-hd*)
by *auto*

lemma *verticesFrom-pres-nodes1*: $v \in \mathcal{V} f \implies \mathcal{V} f = \text{set}(\text{verticesFrom } f v)$
proof –

assume $v \in \mathcal{V} f$
then have $\text{fst} (\text{splitAt } v (\text{vertices } f)) @ [v] @ \text{snd} (\text{splitAt } v (\text{vertices } f)) = \text{vertices } f$
apply (*drule-tac splitAt-ram*) **by** *simp*
moreover have $\text{set} (\text{fst} (\text{splitAt } v (\text{vertices } f)) @ [v] @ \text{snd} (\text{splitAt } v (\text{vertices } f))) = \text{set} (\text{verticesFrom } f v)$
by (*auto simp: verticesFrom-Def*)
ultimately show *?thesis* **by** *simp*
qed

lemma *verticesFrom-pres-nodes*: $v \in \mathcal{V} f \implies w \in \mathcal{V} f \implies w \in \text{set} (\text{verticesFrom } f v)$
by (*auto dest: verticesFrom-pres-nodes1*)

lemma *before-verticesFrom*: $\text{distinct} (\text{vertices } f) \implies v \in \mathcal{V} f \implies w \in \mathcal{V} f \implies v \neq w \implies \text{before} (\text{verticesFrom } f v) v w$

proof –

assume $v: v \in \mathcal{V} f$ **and** $w: w \in \mathcal{V} f$ **and** $\text{neg}: v \neq w$
have $\text{hd} (\text{verticesFrom } f v) = v$ **by** (*rule verticesFrom-hd*)
with v **have** $\text{vf}:\text{verticesFrom } f v = v \# \text{tl} (\text{verticesFrom } f v)$
apply (*frule-tac verticesFrom-pres-nodes1*)
apply (*cases verticesFrom f v*) **by** *auto*
moreover with $v w$ **have** $w \in \text{set} (\text{verticesFrom } f v)$ **by** (*auto simp: verticesFrom-pres-nodes*)
ultimately have $w \in \text{set} (v \# \text{tl} (\text{verticesFrom } f v))$ **by** *auto*
with neg **have** $w \in \text{set} (\text{tl} (\text{verticesFrom } f v))$ **by** *auto*
with vf **have** $\text{verticesFrom } f v =$
 $\quad \llbracket @ v \# \text{fst} (\text{splitAt } w (\text{tl} (\text{verticesFrom } f v))) @ w \# \text{snd} (\text{splitAt } w (\text{tl} (\text{verticesFrom } f v))) \rrbracket$
by (*auto dest: splitAt-ram*)
then show *?thesis* **apply** (*unfold before-def*) **by** (*intro exI*)
qed

lemma *last-vFrom*:

$\llbracket \text{distinct} (\text{vertices } f); x \in \mathcal{V} f \rrbracket \implies \text{last} (\text{verticesFrom } f x) = f^{-1} \cdot x$
apply (*frule split-list*)
apply (*clarsimp simp: prevVertex-def verticesFrom-Def split:list.split*)
done

lemma *rotate-before-vFrom*:

$\llbracket \text{distinct} (\text{vertices } f); r \in \mathcal{V} f; r \neq u \rrbracket \implies$
 $\text{before} (\text{verticesFrom } f r) u v \implies \text{before} (\text{verticesFrom } f v) r u$
using $\llbracket \text{linarith-neq-limit} = 1 \rrbracket$
apply (*frule split-list*)
apply (*clarsimp simp: verticesFrom-Def*)
apply (*rename-tac as bs*)
apply (*clarsimp simp: before-def*)

```

apply(rename-tac xs ys zs)
apply(subst (asm) Cons-eq-append-conv)
apply clarsimp
apply(rename-tac bs^)
apply(subst (asm) append-eq-append-conv2)
apply clarsimp
apply(rename-tac as^)
apply(erule disjE)
  defer
    apply clarsimp
    apply(rule-tac x = v#zs in exI)
    apply(rule-tac x = bs@as' in exI)
    apply simp
    apply clarsimp
    apply(subst (asm) append-eq-Cons-conv)
    apply(erule disjE)
    apply clarsimp
    apply(rule-tac x = v#zs in exI)
    apply simp apply blast
    apply clarsimp
    apply(rename-tac ys^)
    apply(subst (asm) append-eq-append-conv2)
    apply clarsimp
    apply(rename-tac vs^)
    apply(erule disjE)
    apply clarsimp
    apply(subst (asm) append-eq-Cons-conv)
    apply(erule disjE)
    apply clarsimp
    apply(rule-tac x = v#zs in exI)
    apply simp apply blast
    apply clarsimp
    apply(rule-tac x = v#ys'@as in exI)
    apply simp apply blast
    apply clarsimp
    apply(rule-tac x = v#zs in exI)
    apply simp apply blast
  done

```

lemma *before-between*:

```


$$\llbracket \text{before}(\text{verticesFrom } f \ x) \ y \ z; \text{distinct}(\text{vertices } f); x \in \mathcal{V} \ f; x \neq y \rrbracket \implies$$


$$y \in \text{set}(\text{between } (\text{vertices } f) \ x \ z)$$

apply(induct f)
apply(clarsimp simp:verticesFrom-Def before-def)
apply(frule split-list)
apply(clarsimp simp:Cons-eq-append-conv)
apply(subst (asm) append-eq-append-conv2)
apply clarsimp
apply(erule disjE)

```



```

apply(clarsimp simp:append-eq-Cons-conv)
prefer 2 apply(clarsimp simp add:between-def split-def)
apply(erule disjE)
apply (clarsimp simp:between-def split-def)
apply clarsimp
apply(subst (asm) append-eq-append-conv2)
apply clarsimp
apply(erule disjE)
prefer 2 apply(clarsimp simp add:between-def split-def)
apply(clarsimp simp:append-eq-Cons-conv)
apply(fastforce simp:between-def split-def)
done

```

lemma *before-between2*:

```

[[ before (verticesFrom f u) v w; distinct(vertices f); u ∈ V f ]]
  ⇒ u = v ∨ u ∈ set (between (vertices f) w v)
apply(subgoal-tac pre-between (vertices f) v w)
apply(subst verticesFrom-between)
apply assumption
apply(erule pre-between-symI)
apply(frule pre-between-r1)
apply(drule (1) verticesFrom-distinct)
using verticesFrom-hd[of f u]
apply(clarsimp simp add:before-def between-def split-def hd-append
  split:if-split-asm)
apply(frule (1) verticesFrom-distinct)
apply(clarsimp simp:pre-between-def before-def simp del:verticesFrom-distinct)
apply(rule conjI)
apply(cases v = u)
apply simp
apply(rule verticesFrom-in'[of v f u])apply simp apply assumption
apply(cases w = u)
apply simp
apply(rule verticesFrom-in'[of w f u])apply simp apply assumption
done

```

12.10 *splitFace*

definition *pre-splitFace* :: *graph* ⇒ *vertex* ⇒ *vertex* ⇒ *face* ⇒ *vertex list* ⇒ *bool*
where

```

pre-splitFace g ram1 ram2 oldF nvs ≡
  oldF ∈ F g ∧ ¬ final oldF ∧ distinct (vertices oldF) ∧ distinct nvs
  ∧ V g ∩ set nvs = {}
  ∧ V oldF ∩ set nvs = {}
  ∧ ram1 ∈ V oldF ∧ ram2 ∈ V oldF
  ∧ ram1 ≠ ram2
  ∧ (((ram1,ram2) ∉ edges oldF ∧ (ram2,ram1) ∉ edges oldF
    ∧ (ram1, ram2) ∉ edges g ∧ (ram2, ram1) ∉ edges g) ∨ nvs ≠ []))

```

declare *pre-splitFace-def* [*simp*]

lemma *pre-splitFace-pre-split-face*[*simp*]:

pre-splitFace g ram1 ram2 oldF nvs \implies *pre-split-face oldF ram1 ram2 nvs*
by (*simp add: pre-splitFace-def pre-split-face-def*)

lemma *pre-splitFace-oldF*[*simp*]:

pre-splitFace g ram1 ram2 oldF nvs \implies *oldF* \in \mathcal{F} *g*
apply (*unfold pre-splitFace-def*) **by force**

declare *pre-splitFace-def* [*simp del*]

lemma *splitFace-split-face*:

oldF \in \mathcal{F} *g* \implies
(*f*₁, *f*₂, *newGraph*) = *splitFace g ram₁ ram₂ oldF newVs* \implies
(*f*₁, *f*₂) = *split-face oldF ram₁ ram₂ newVs*
by (*simp add: splitFace-def split-def*)

lemma *split-face-empty-ram2-ram1-in-f12*:

pre-split-face oldF ram1 ram2 $\square \implies$
(*f*₁₂, *f*₂₁) = *split-face oldF ram1 ram2* $\square \implies$ (*ram*₂, *ram*₁) \in *edges f12*

proof –

assume *split*: (*f*₁₂, *f*₂₁) = *split-face oldF ram1 ram2* \square
pre-split-face oldF ram1 ram2 \square
then have *ram*₂ \in \mathcal{V} *f12* **by** (*simp add: split-face-def*)
moreover with *split* **have** *f12* \cdot *ram*₂ = *hd (vertices f12)*
apply (*rule-tac nextElem-suc2*)
apply (*simp add: pre-split-face-def split-face-distinct1*)
by (*simp add: split-face-def*)
with *split* **have** *ram*₁ = *f12* \cdot *ram*₂
by (*simp add: split-face-def*)
ultimately show *?thesis* **by** (*simp add: edges-face-def*)

qed

lemma *split-face-empty-ram2-ram1-in-f12'*:

pre-split-face oldF ram1 ram2 $\square \implies$
(*ram*₂, *ram*₁) \in *edges (fst (split-face oldF ram1 ram2* $\square))$

proof –

assume *split*: *pre-split-face oldF ram1 ram2* \square
define *f12* **where** *f12* = *fst (split-face oldF ram1 ram2* $\square)$
define *f21* **where** *f21* = *snd (split-face oldF ram1 ram2* $\square)$
then have (*f*₁₂, *f*₂₁) = *split-face oldF ram1 ram2* \square **by** (*simp add: f12-def f21-def*)
with *split* **have** (*ram*₂, *ram*₁) \in *edges f12*
by (*rule split-face-empty-ram2-ram1-in-f12*)
with *f12-def* **show** *?thesis* **by simp**

qed

lemma *splitFace-empty-ram2-ram1-in-f12*:

pre-splitFace g ram1 ram2 oldF [] \implies
(f12, f21, newGraph) = splitFace g ram1 ram2 oldF [] \implies
(ram2, ram1) ∈ edges f12

proof –

assume *pre*: *pre-splitFace g ram1 ram2 oldF* []
then have *oldF*: *oldF ∈ F g* **by** (*unfold pre-splitFace-def*) *simp*
assume *sp*: *(f12, f21, newGraph) = splitFace g ram1 ram2 oldF* []
with *oldF* **have** *(f12, f21) = split-face oldF ram1 ram2* []
by (*rule splitFace-split-face*)

with *pre sp* **show** *?thesis*
apply (*unfold splitFace-def pre-splitFace-def*)
apply (*simp add: split-def*)
apply (*rule split-face-empty-ram2-ram1-in-f12'*)
apply (*rule pre-splitFace-pre-split-face*)
apply (*rule pre*)
done

qed

lemma *splitFace-f12-new-vertices*:

(f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs \implies
v ∈ set newVs $\implies v ∈ V f12$
apply (*unfold splitFace-def*)
apply (*simp add: split-def*)
apply (*unfold split-face-def Let-def*)
by *simp*

lemma *splitFace-add-vertices-direct*[*simp*]:

vertices (snd (snd (splitFace g ram1 ram2 oldF [countVertices g ..< countVertices g + n])))
= vertices g @ [countVertices g ..< countVertices g + n]
apply (*auto simp: splitFace-def split-def*) **apply** (*cases g*)
apply *auto* **apply** (*induct n*) **by** *auto*

lemma *splitFace-delete-oldF*:

(f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVertexList \implies
oldF ≠ f12 \implies *oldF ≠ f21* \implies *distinct (faces g)* \implies
oldF ∉ F newGraph
by (*simp add: splitFace-def split-def distinct-set-replace*)

lemma *splitFace-faces-1*:

(f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVertexList \implies
oldF ∈ F g \implies
set (faces newGraph) ∪ {oldF} = {f12, f21} ∪ set (faces g)
(is *?oldF* \implies *?C* \implies *?A = ?B*)
proof (*intro equalityI subsetI*)

```

fix x
  assume  $x \in ?A$  and  $?C$  and  $?oldF$ 
  then show  $x \in ?B$  apply (simp add: splitFace-def split-def) by (auto dest: replace1)
next
  fix x
  assume  $x \in ?B$  and  $?C$  and  $?oldF$ 
  then show  $x \in ?A$  apply (simp add: splitFace-def split-def)
    apply (cases x = oldF) apply simp-all
    apply (cases x = f12) apply simp-all
    apply (cases x = f21) by (auto intro: replace3 replace4)
qed

```

```

lemma splitFace-distinct1[intro]:pre-splitFace g ram1 ram2 oldF newVertexList
 $\implies$ 
  distinct (vertices (fst (snd (splitFace g ram1 ram2 oldF newVertexList))))
  apply (unfold splitFace-def split-def)
  by (auto simp add: split-def)

```

```

lemma splitFace-distinct2[intro]:pre-splitFace g ram1 ram2 oldF newVertexList
 $\implies$ 
  distinct (vertices (fst (splitFace g ram1 ram2 oldF newVertexList)))
  apply (unfold splitFace-def split-def)
  by (auto simp add: split-def)

```

```

lemma splitFace-add-f21': f' ∈ F g' ⟹ fst (snd (splitFace g' v a f' nvl))
  ∈ F (snd (snd (splitFace g' v a f' nvl)))
apply (simp add: splitFace-def split-def) apply (rule disjI2)
apply (rule replace3) by simp-all

```

```

lemma split-face-help[simp]: Suc 0 < |vertices (fst (split-face f' v a nvl))|
  by (simp add: split-face-def)

```

```

lemma split-face-help'[simp]: Suc 0 < |vertices (snd (split-face f' v a nvl))|
  by (simp add: split-face-def)

```

```

lemma splitFace-split: f ∈ F (snd (snd (splitFace g v a f' nvl))) ⟹
   $f \in \mathcal{F} g$ 
   $\vee f = \text{fst } (\text{splitFace } g \ v \ a \ f' \ \text{nvl})$ 
   $\vee f = (\text{fst } (\text{snd } (\text{splitFace } g \ v \ a \ f' \ \text{nvl})))$ 
apply (simp add: splitFace-def split-def)
apply safe by (frule replace5) simp

```

```

lemma pre-FaceDiv-between1: pre-splitFace g' ram1 ram2 f [] ⟹
   $\neg \text{between } (\text{vertices } f) \ \text{ram1} \ \text{ram2} = []$ 

```

```

proof –
  assume pre-f: pre-splitFace g' ram1 ram2 f []

```

```

then have pre-bet: pre-between (vertices f) ram1 ram2 apply (unfold pre-splitFace-def)
  by (simp add: pre-between-def)
then have pre-bet': pre-between (verticesFrom f ram1) ram1 ram2
  by (simp add: pre-between-def set-verticesFrom)
from pre-f have dist': distinct (verticesFrom f ram1) apply (unfold pre-splitFace-def)
by simp
from pre-f have ram2: ram2  $\in \mathcal{V} f$  apply (unfold pre-splitFace-def) by simp
from pre-f have  $\neg$  is-nextElem (vertices f) ram1 ram2 apply (unfold pre-splitFace-def)
by auto
with pre-f have  $\neg$  is-nextElem (verticesFrom f ram1) ram1 ram2 apply (unfold
pre-splitFace-def)
  by (simp add: verticesFrom-is-nextElem [symmetric])
moreover
from pre-f have ram2  $\in$  set (verticesFrom f ram1) apply (unfold pre-splitFace-def)
by auto
moreover
from pre-f have ram2  $\neq$  ram1 apply (unfold pre-splitFace-def) by auto
ultimately have ram2-not: ram2  $\neq$  hd (snd (splitAt ram1 (vertices f))) @ fst
(splitAt ram1 (vertices f)))
  apply (simp add: is-nextElem-def verticesFrom-Def)
  apply (cases snd (splitAt ram1 (vertices f))) @ fst (splitAt ram1 (vertices f)))
  apply simp apply simp
  apply (simp add: is-sublist-def) by auto

from pre-f have before: before (verticesFrom f ram1) ram1 ram2 apply (unfold
pre-splitFace-def)
  apply safe apply (rule before-verticesFrom) by auto
have fst (splitAt ram2 (snd (splitAt ram1 (verticesFrom f ram1)))) = []  $\implies$ 
False
proof –
  assume fst (splitAt ram2 (snd (splitAt ram1 (verticesFrom f ram1)))) = []
  moreover
from ram2 pre-f have ram2  $\in$  set (verticesFrom f ram1) apply (unfold pre-splitFace-def)
  by auto
with pre-f have ram2  $\in$  set (snd (splitAt ram1 (vertices f))) @ fst (splitAt
ram1 (vertices f)))
  apply (simp add: verticesFrom-Def)
  apply (unfold pre-splitFace-def) by auto
moreover
note dist'
ultimately have ram2 = hd (snd (splitAt ram1 (vertices f))) @ fst (splitAt
ram1 (vertices f)))
  apply (rule-tac ccontr)
  apply (cases (snd (splitAt ram1 (vertices f))) @ fst (splitAt ram1 (vertices
f))))
  apply simp
  apply simp
  by (simp add: verticesFrom-Def del: distinct-append)

```

with *ram2-not* **show** *?thesis* **by** *auto*
qed
with *before pre-bet'* **have** *between (verticesFrom f ram1) ram1 ram2 ≠ []* **by**
auto
with *pre-f pre-bet* **show** *?thesis* **apply** (*unfold pre-splitFace-def*) **apply** *safe*
by (*simp add: verticesFrom-between*)
qed

lemma *pre-FaceDiv-between2: pre-splitFace g' ram1 ram2 f [] ⇒*
¬ between (vertices f) ram2 ram1 = []

proof –

assume *pre-f: pre-splitFace g' ram1 ram2 f []*
then have *pre-splitFace g' ram2 ram1 f []* **apply** (*unfold pre-splitFace-def*) **by**
auto
then show *?thesis* **by** (*rule pre-FaceDiv-between1*)
qed

definition *Edges :: vertex list ⇒ (vertex × vertex) set* **where**
Edges vs ≡ {(a,b). is-sublist [a,b] vs}

lemma *Edges-Nil[simp]: Edges [] = {}*
by(*simp add: Edges-def is-sublist-def*)

lemma *Edges-rev:*

Edges (rev (zs::vertex list)) = {(b,a). (a,b) ∈ Edges zs}
by (*auto simp add: Edges-def is-sublist-rev*)

lemma *in-Edges-rev[simp]:*

((a,b) : Edges (rev (zs::vertex list))) = ((b,a) ∈ Edges zs)
by (*simp add: Edges-rev*)

lemma *notinset-notinEdge1: x ∉ set xs ⇒ (x,y) ∉ Edges xs*
by(*unfold Edges-def*)(*blast intro:is-sublist-in*)

lemma *notinset-notinEdge2: y ∉ set xs ⇒ (x,y) ∉ Edges xs*
by(*unfold Edges-def*)(*blast intro:is-sublist-in1*)

lemma *in-Edges-in-set: (x,y) : Edges vs ⇒ x ∈ set vs ∧ y ∈ set vs*
by(*unfold Edges-def*)(*blast intro:is-sublist-in is-sublist-in1*)

lemma *edges-conv-Edges:*

distinct(vertices(f::face)) ⇒ $\mathcal{E} f =$
Edges (vertices f) ∪
(if vertices f = [] then {} else {(last(vertices f), hd(vertices f))})
by(*induct f*) (*auto simp: Edges-def is-nextElem-def*)

```

lemma Edges-Cons: Edges(x#xs) =
  (if xs = [] then {} else Edges xs ∪ {(x,hd xs)})
apply(auto simp:Edges-def)
  apply(rule ccontr)
  apply(simp)
  apply(erule thin-rl)
  apply(erule contrapos-np)
  apply(clarsimp simp:is-sublist-def Cons-eq-append-conv)
  apply(rename-tac as bs)
  apply(erule disjE)
  apply simp
  apply(clarsimp)
  apply(rename-tac cs)
  apply(rule-tac x = cs in exI)
  apply(rule-tac x = bs in exI)
  apply(rule HOL.refl)
  apply(clarsimp simp:neq-Nil-conv)
  apply(subst is-sublist-rec)
  apply simp
  apply(simp add:is-sublist-def)
  apply(erule exE)+
  apply(rename-tac as bs)
  apply simp
  apply(rule-tac x = x#as in exI)
  apply(rule-tac x = bs in exI)
  apply simp
done

```

```

lemma Edges-append: Edges(xs @ ys) =
  (if xs = [] then Edges ys else
   if ys = [] then Edges xs else
   Edges xs ∪ Edges ys ∪ {(last xs, hd ys)})
apply(induct xs)
  apply simp
  apply (simp add:Edges-Cons)
apply blast
done

```

```

lemma Edges-rev-disj: distinct xs ⟹ Edges(rev xs) ∩ Edges(xs) = {}
apply(induct xs)
  apply simp
apply(auto simp:Edges-Cons Edges-append last-rev
  notinset-notinEdge1 notinset-notinEdge2)
done

```

```

lemma disj-sets-disj-Edges:

```

$set\ xs \cap set\ ys = \{\} \implies Edges\ xs \cap Edges\ ys = \{\}$
by(*unfold Edges-def*)(*blast intro:is-sublist-in is-sublist-in1*)

lemma *disj-sets-disj-Edges2*:
 $set\ ys \cap set\ xs = \{\} \implies Edges\ xs \cap Edges\ ys = \{\}$
by(*blast intro!:disj-sets-disj-Edges*)

lemma *finite-Edges[iff]*: *finite*(*Edges xs*)
by(*induct xs*)(*simp-all add:Edges-Cons*)

lemma *Edges-compl*:
 $\llbracket distinct\ vs; x \in set\ vs; y \in set\ vs; x \neq y \rrbracket \implies$
 $Edges(x \# between\ vs\ x\ y\ @\ [y]) \cap Edges(y \# between\ vs\ y\ x\ @\ [x]) = \{\}$
using $\llbracket linarith-neq-limit = 1 \rrbracket$
apply(*subgoal-tac*)
 $\wedge xs\ (ys::vertex\ list). xs \neq [] \implies set\ xs \cap set\ ys = \{\} \implies hd\ xs \notin set\ ys$
prefer 2 **apply**(*drule hd-in-set*) **apply**(*blast*)
apply(*frule split-list[of x]*)
apply *clarsimp*
apply(*erule disjE*)
apply(*frule split-list[of y]*)
apply(*clarsimp simp add:between-def split-def*)
apply (*clarsimp simp add:Edges-Cons Edges-append*
notinset-notinEdge1 notinset-notinEdge2
disj-sets-disj-Edges disj-sets-disj-Edges2
Int-Un-distrib Int-Un-distrib2)
apply(*fastforce*)
apply(*frule split-list[of y]*)
apply(*clarsimp simp add:between-def split-def*)
apply (*clarsimp simp add:Edges-Cons Edges-append notinset-notinEdge1*
notinset-notinEdge2 disj-sets-disj-Edges disj-sets-disj-Edges2
Int-Un-distrib Int-Un-distrib2)
apply *fastforce*
done

lemma *Edges-disj*:
 $\llbracket distinct\ vs; x \in set\ vs; z \in set\ vs; x \neq y; y \neq z;$
 $y \in set(between\ vs\ x\ z) \rrbracket \implies$
 $Edges(x \# between\ vs\ x\ y\ @\ [y]) \cap Edges(y \# between\ vs\ y\ z\ @\ [z]) = \{\}$
apply(*subgoal-tac*)
 $\wedge xs\ (ys::vertex\ list). xs \neq [] \implies set\ xs \cap set\ ys = \{\} \implies hd\ xs \notin set\ ys$
prefer 2 **apply**(*drule hd-in-set*) **apply**(*blast*)
apply(*frule split-list[of x]*)
apply *clarsimp*
apply(*erule disjE*)
apply *simp*
apply(*drule inbetween-inset*)


```

apply(rule Edges-compl)
  apply simp
  apply simp
  apply simp
  apply simp
apply(erule disjE)
apply(frule split-list[of z])
apply(clarsimp simp add:between-def split-def)
apply(erule disjE)
apply(frule split-list[of y])
apply clarsimp
apply (clarsimp simp add:Edges-Cons Edges-append
  notinset-notinEdge1 notinset-notinEdge2
  disj-sets-disj-Edges disj-sets-disj-Edges2
  Int-Un-distrib Int-Un-distrib2)
apply fastforce
apply(frule split-list[of y])
apply clarsimp
apply (clarsimp simp add:Edges-Cons Edges-append notinset-notinEdge1
  notinset-notinEdge2 disj-sets-disj-Edges disj-sets-disj-Edges2
  Int-Un-distrib Int-Un-distrib2)
apply fastforce
apply(frule split-list[of z])
apply(clarsimp simp add:between-def split-def)
apply(frule split-list[of y])
apply clarsimp
apply (clarsimp simp add:Edges-Cons Edges-append notinset-notinEdge1
  notinset-notinEdge2 disj-sets-disj-Edges disj-sets-disj-Edges2
  Int-Un-distrib Int-Un-distrib2)
apply fastforce
done

```

lemma *edges-conv-Un-Edges*:

$$\llbracket \text{distinct}(\text{vertices}(f::\text{face})); x \in \mathcal{V} f; y \in \mathcal{V} f; x \neq y \rrbracket \implies$$

$$\mathcal{E} f = \text{Edges}(x \# \text{between}(\text{vertices } f) x y @ [y]) \cup$$

$$\text{Edges}(y \# \text{between}(\text{vertices } f) y x @ [x])$$

```

apply(simp add:edges-conv-Edges)
apply(rule conjI)
  apply clarsimp
apply clarsimp
apply(frule split-list[of x])
apply clarsimp
apply(erule disjE)
  apply(frule split-list[of y])
  apply(clarsimp simp add:between-def split-def)
apply (clarsimp simp add:Edges-Cons Edges-append
  notinset-notinEdge1 notinset-notinEdge2
  disj-sets-disj-Edges disj-sets-disj-Edges2
  Int-Un-distrib Int-Un-distrib2)

```

```

apply(fastforce)
apply(frule split-list[of y])
apply(clarsimp simp add:between-def split-def)
apply (clarsimp simp add:Edges-Cons Edges-append
  notinset-notinEdge1 notinset-notinEdge2
  disj-sets-disj-Edges disj-sets-disj-Edges2
  Int-Un-distrib Int-Un-distrib2)
apply(fastforce)
done

```

```

lemma Edges-between-edges:
   $\llbracket (a,b) \in \text{Edges } (u \# \text{between } (\text{vertices}(f::\text{face})) \ u \ v \ @ \ [v]);$ 
   $\text{pre-split-face } f \ u \ v \ vs \rrbracket \implies (a,b) \in \mathcal{E} \ f$ 
apply(simp add:pre-split-face-def)
apply(induct f)
apply(simp add:edges-conv-Edges Edges-Cons)
apply clarify
apply(rename-tac list)
apply(case-tac between list u v = [])
apply simp
apply(drule (4) is-nextElem-between-empty')
apply(simp add:Edges-def)
apply(subgoal-tac pre-between list u v)
prefer 2 apply (simp add:pre-between-def)
apply(subgoal-tac pre-between list v u)
prefer 2 apply (simp add:pre-between-def)
apply(case-tac before list u v)
apply(drule (1) between-eq2)
apply clarsimp
apply(erule disjE)
apply (clarsimp simp:neq-Nil-conv)
apply(rule is-nextElem-sublistI)
apply(simp (no-asm) add:is-sublist-def)
apply blast
apply(rule is-nextElem-sublistI)
apply(clarsimp simp add:Edges-def is-sublist-def)
apply(rename-tac as bs cs xs ys)
apply(rule-tac x = as @ u # xs in exI)
apply(rule-tac x = ys @ cs in exI)
apply simp
apply (simp only:before-xor)
apply(drule (1) between-eq2)
apply clarsimp
apply(rename-tac as bs cs)
apply(erule disjE)
apply (clarsimp simp:neq-Nil-conv)
apply(case-tac cs)
apply clarsimp

```

```

apply(simp add:is-nextElem-def)
apply simp
apply(rule is-nextElem-sublistI)
apply(simp (no-asm) add:is-sublist-def)
apply(rule-tac x = as @ v # bs in exI)
apply simp
apply(rule-tac m1 = |as|+1 in is-nextElem-rotate-eq[THEN iffD1])
apply simp
apply(simp add:rotate-drop-take)
apply(rule is-nextElem-sublistI)
apply(clarsimp simp add:Edges-def is-sublist-def)
apply(rename-tac xs ys)
apply(rule-tac x = bs @ u # xs in exI)
apply simp
done

```

```

lemma edges-split-face1: pre-split-face f u v vs  $\implies$ 
   $\mathcal{E}(\text{fst}(\text{split-face } f \ u \ v \ vs)) =$ 
   $\text{Edges}(v \# \text{rev } vs \ @ \ [u]) \cup \text{Edges}(u \# \text{between } (\text{vertices } f) \ u \ v \ @ \ [v])$ 
apply(simp add: edges-conv-Edges split-face-distinct1')
apply(auto simp add:split-face-def Edges-Cons Edges-append)
done

```

```

lemma edges-split-face2: pre-split-face f u v vs  $\implies$ 
   $\mathcal{E}(\text{snd}(\text{split-face } f \ u \ v \ vs)) =$ 
   $\text{Edges}(u \# \text{vs} \ @ \ [v]) \cup \text{Edges}(v \# \text{between } (\text{vertices } f) \ v \ u \ @ \ [u])$ 
apply(simp add: edges-conv-Edges split-face-distinct2')
apply(auto simp add:split-face-def Edges-Cons Edges-append)
done

```

```

lemma split-face-empty-ram1-ram2-in-f21:
  pre-split-face oldF ram1 ram2 []  $\implies$ 
   $(f12, f21) = \text{split-face } \text{oldF } \text{ram1 } \text{ram2 } [] \implies (\text{ram1}, \text{ram2}) \in \text{edges } f21$ 
proof –
  assume split:  $(f12, f21) = \text{split-face } \text{oldF } \text{ram1 } \text{ram2 } []$ 
  pre-split-face oldF ram1 ram2 []
  then have ram1  $\in \mathcal{V} \ f21$  by (simp add: split-face-def)
  moreover with split have  $f21 \cdot \text{ram1} = \text{hd } (\text{vertices } f21)$ 
  apply (rule-tac nextElem-suc2)
  apply (simp add: pre-split-face-def split-face-distinct2)
  by (simp add: split-face-def)
  with split have  $\text{ram2} = f21 \cdot \text{ram1}$ 
  by (simp add: split-face-def)

```

ultimately show *?thesis* **by** (*simp add: edges-face-def*)
qed

lemma *split-face-empty-ram1-ram2-in-f21'*:

pre-split-face oldF ram1 ram2 [] \implies
(ram1, ram2) ∈ edges (snd (split-face oldF ram1 ram2 []))

proof –

assume *split: pre-split-face oldF ram1 ram2 []*

define *f12* **where** *f12 = fst (split-face oldF ram1 ram2 [])*

define *f21* **where** *f21 = snd (split-face oldF ram1 ram2 [])*

then have (*f12, f21*) = *split-face oldF ram1 ram2 []* **by** (*simp add: f12-def f21-def*)

with *split* **have** (*ram1, ram2*) ∈ *edges f21*

by (*rule split-face-empty-ram1-ram2-in-f21*)

with *f21-def* **show** *?thesis* **by** *simp*

qed

lemma *splitFace-empty-ram1-ram2-in-f21*:

pre-splitFace g ram1 ram2 oldF [] \implies
(f12, f21, newGraph) = splitFace g ram1 ram2 oldF [] \implies
(ram1, ram2) ∈ edges f21

proof –

assume *pre: pre-splitFace g ram1 ram2 oldF []*

then have *oldF: oldF ∈ F g* **by** (*unfold pre-splitFace-def*) *simp*

assume *sp: (f12, f21, newGraph) = splitFace g ram1 ram2 oldF []*

with *oldF* **have** (*f12, f21*) = *split-face oldF ram1 ram2 []*

by (*rule splitFace-split-face*)

with *pre sp* **show** *?thesis*

apply (*unfold splitFace-def pre-splitFace-def*)

apply (*simp add: split-def*)

apply (*rule split-face-empty-ram1-ram2-in-f21'*)

apply (*rule pre-splitFace-pre-split-face*)

apply (*rule pre*)

done

qed

lemma *splitFace-f21-new-vertices*:

(f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs \implies
v ∈ set newVs \implies *v ∈ V f21*

apply (*unfold splitFace-def*)

apply (*simp add: split-def*)

apply (*unfold split-face-def*)

by *simp*

lemma *split-face-edges-f12*:

assumes *vors: pre-split-face f ram1 ram2 vs*

(f12, f21) = split-face f ram1 ram2 vs

vs ≠ [] vs1 = between (vertices f) ram1 ram2 vs1 ≠ []

```

shows edges f12 =
  {(hd vs, ram1), (ram1, hd vs1), (last vs1, ram2), (ram2, last vs)} ∪
  Edges(rev vs) ∪ Edges vs1 (is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs
  define c where c = fst x
  define d where d = snd x
  with c-def have [simp]: x = (c,d) by simp
  from vors have dist-f12: distinct (vertices f12) apply (rule-tac split-face-distinct1)
by auto
  from x vors show x ∈ ?rhs
    apply (simp add: split-face-def is-nextElem-def is-sublist-def dist-f12)
    apply (case-tac c = ram2 ∧ d = last vs) apply simp apply simp apply (elim
exE)
    apply (case-tac c = ram1) apply simp
    apply (subgoal-tac between (vertices f) ram1 ram2 @ [ram2] = d # bs)
    apply (case-tac between (vertices f) ram1 ram2) apply simp apply simp
    apply (rule dist-at2) apply (rule dist-f12) apply (rule sym) apply simp
apply simp

    apply (case-tac c ∈ set(rev vs))
    apply (subgoal-tac distinct(rev vs)) apply (simp only: in-set-conv-decomp)
apply (elim exE) apply simp
    apply (case-tac zs) apply simp
    apply (metis dist-at2 dist-f12 last-rev last-snoc list.inject vertices-face.simps)
    apply (simp add: rev-swap)
    apply (subgoal-tac ys = as)
    apply (clarsimp simp add: Edges-def is-sublist-def)
    apply (rule conjI)
    apply (subgoal-tac ∃ as bs. rev list @ [d, c] = as @ d # c # bs) apply simp
apply (intro exI) apply simp
    apply (subgoal-tac ∃ asa bs. rev list @ d # c # rev as = asa @ d # c # bs)
apply simp apply (intro exI) apply simp
    apply (rule dist-at1) apply (rule dist-f12) apply (rule sym) apply simp
apply simp
    apply (simp add: pre-split-face-def)

    apply (case-tac c ∈ set (between (vertices f) ram1 ram2))
    apply (subgoal-tac distinct (between (vertices f) ram1 ram2)) apply (simp
add: in-set-conv-decomp) apply (elim exE) apply simp
    apply (case-tac zs) apply simp apply (subgoal-tac rev vs @ ram1 # ys =
as) apply force
    apply (rule dist-at1) apply (rule dist-f12) apply (rule sym) apply simp
apply simp
    apply simp
    apply (subgoal-tac rev vs @ ram1 # ys = as) apply (simp add: Edges-def
is-sublist-def)
    apply (subgoal-tac (rev vs @ ram1 # ys) @ c # a # list @ [ram2] = as @

```

```

c # d # bs) apply simp
  apply (rule conjI) apply (rule impI) apply (rule disjI2)+ apply (rule
exI) apply force
  apply (rule impI) apply (rule disjI2)+ apply force
  apply force
  apply (rule dist-at1) apply (rule dist-f12) apply (rule sym) apply simp
apply simp
  apply (thin-tac rev vs @ ram1 # between (vertices f) ram1 ram2 @ [ram2] =
as @ c # d # bs)
  apply (subgoal-tac distinct (vertices f12)) apply simp
  apply (rule dist-f12)

  apply (subgoal-tac c = ram2) apply simp
  apply (subgoal-tac rev vs @ ram1 # between (vertices f) ram1 ram2 = as)
apply force
  apply (rule dist-at1) apply (rule dist-f12) apply (rule sym) apply simp
apply simp

  apply (subgoal-tac c ∈ set (rev vs @ ram1 # between (vertices f) ram1 ram2
@ [ram2]))
  apply (thin-tac rev vs @ ram1 # between (vertices f) ram1 ram2 @ [ram2] =
as @ c # d # bs) apply simp
  by simp
next
  fix x
  assume x: x ∈ ?rhs
  define c where c = fst x
  define d where d = snd x
  with c-def have [simp]: x = (c,d) by simp
from vors have dist-f12: distinct (vertices f12) apply (rule-tac split-face-distinct1)
by auto
from x vors show x ∈ ?lhs
  supply [[simproc del: defined-all]]
  apply (simp add: dist-f12 is-nextElem-def is-sublist-def) apply (simp add:
split-face-def)
  apply (case-tac c = ram2 ∧ d = last vs) apply simp
  apply (rule disjCI) apply simp
  apply (case-tac c = hd vs ∧ d = ram1)
  apply (case-tac vs) apply simp
  apply force
  apply simp
  apply (case-tac c = ram1 ∧ d = hd (between (vertices f) ram1 ram2))
  apply (case-tac between (vertices f) ram1 ram2) apply simp apply force
apply simp
  apply (case-tac c = last (between (vertices f) ram1 ram2) ∧ d = ram2)
  apply (case-tac between (vertices f) ram1 ram2 rule: rev-exhaust) apply simp
apply simp
  apply (intro exI) apply (subgoal-tac rev vs @ ram1 # ys @ [y, ram2] = (rev

```

```

vs @ ram1 # ys) @ y # ram2 # []
  apply assumption
  apply simp
  apply simp
  apply (case-tac (d,c) ∈ Edges vs) apply (simp add: Edges-def is-sublist-def)
  apply (elim exE) apply simp apply (intro exI) apply simp
  apply (simp add: Edges-def is-sublist-def)
  apply (elim exE) apply simp apply (intro exI)
  apply (subgoal-tac rev vs @ ram1 # as @ c # d # bs @ [ram2] = (rev vs @
ram1 # as) @ c # d # bs @ [ram2])
  apply assumption
  by simp
qed

```

```

lemma split-face-edges-f12-vs:
assumes vors: pre-split-face f ram1 ram2 []
      (f12, f21) = split-face f ram1 ram2 []
      vs1 = between (vertices f) ram1 ram2 vs1 ≠ []
shows edges f12 = {(ram2, ram1) , (ram1, hd vs1), (last vs1, ram2)} ∪
      Edges vs1 (is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs
  define c where c = fst x
  define d where d = snd x
  with c-def have [simp]: x = (c,d) by simp
  from vors have dist-f12: distinct (vertices f12) apply (rule-tac split-face-distinct1)
  by auto
  from x vors show x ∈ ?rhs
    apply (simp add: split-face-def is-nextElem-def is-sublist-def dist-f12)
    apply (case-tac c = ram2 ∧ d = ram1) apply simp
    apply simp apply (elim exE)
    apply (case-tac c = ram1) apply simp
    apply (subgoal-tac as = []) apply simp
    apply (case-tac between (vertices f) ram1 ram2) apply simp
    apply simp
    apply (rule dist-at1) apply (rule dist-f12) apply force apply (rule sym)
  apply simp

  apply (case-tac c ∈ set (between (vertices f) ram1 ram2))
  apply (subgoal-tac distinct (between (vertices f) ram1 ram2)) apply (simp
add: in-set-conv-decomp) apply (elim exE) apply simp
  apply (case-tac zs) apply simp apply (subgoal-tac ram1 # ys = as) apply
force
  apply (rule dist-at1) apply (rule dist-f12) apply (rule sym) apply simp
apply simp
  apply simp
  apply (subgoal-tac ram1 # ys = as) apply (simp add: Edges-def is-sublist-def)
  apply (subgoal-tac (ram1 # ys) @ c # a # list @ [ram2] = as @ c # d #

```

```

bs) apply simp
      apply (rule conjI) apply (rule impI) apply (rule disjI2)+ apply (rule
exI) apply force
      apply (rule impI) apply (rule disjI2)+ apply force
      apply force
      apply (rule dist-at1) apply (rule dist-f12) apply (rule sym) apply simp
apply simp
      apply (thin-tac ram1 # between (vertices f) ram1 ram2 @ [ram2] = as @ c
# d # bs)
      apply (subgoal-tac distinct (vertices f12)) apply simp
      apply (rule dist-f12)

      apply (subgoal-tac c = ram2) apply simp
      apply (subgoal-tac ram1 # between (vertices f) ram1 ram2 = as) apply force
      apply (rule dist-at1) apply (rule dist-f12) apply (rule sym) apply simp
apply simp

      apply (subgoal-tac c ∈ set (ram1 # between (vertices f) ram1 ram2 @ [ram2]))
      apply (thin-tac ram1 # between (vertices f) ram1 ram2 @ [ram2] = as @ c
# d # bs) apply simp
      by simp
next
fix x
assume x: x ∈ ?rhs
define c where c = fst x
define d where d = snd x
with c-def have [simp]: x = (c,d) by simp
from vors have dist-f12: distinct (vertices f12) apply (rule-tac split-face-distinct1)
by auto
from x vors show x ∈ ?lhs
      supply [[simproc del: defined-all]]
      apply (simp add: dist-f12 is-nextElem-def is-sublist-def) apply (simp add:
split-face-def)
      apply (case-tac c = ram2 ∧ d = ram1) apply simp
      apply (rule disjCI) apply simp
      apply (case-tac c = ram1 ∧ d = hd (between (vertices f) ram1 ram2))
      apply (case-tac between (vertices f) ram1 ram2) apply simp
      apply force
      apply simp
      apply (case-tac c = last (between (vertices f) ram1 ram2) ∧ d = ram2)
      apply (case-tac between (vertices f) ram1 ram2 rule: rev-exhaust) apply simp
      apply simp
      apply (intro exI) apply (subgoal-tac ram1 # ys @ [y, ram2] = (ram1 # ys)
@ y # ram2 # [])
      apply assumption
      apply simp
      apply simp
      apply (case-tac (c, d) ∈ Edges vs) apply (simp add: Edges-def is-sublist-def)

```



```

    apply (elim exE) apply simp apply (intro exI)
    apply (subgoal-tac ram1 # as @ c # d # bs @ [ram2] = (ram1 # as) @ c #
d # (bs @ [ram2])) apply assumption
    apply simp
    apply (simp add: Edges-def is-sublist-def)
    apply (elim exE) apply simp apply (intro exI)
    apply (subgoal-tac ram1 # as @ c # d # bs @ [ram2] = (ram1 # as) @ c #
d # bs @ [ram2])
    apply assumption
    by simp
qed

```

lemma *split-face-edges-f12-bet*:

```

assumes vors: pre-split-face f ram1 ram2 vs
          (f12, f21) = split-face f ram1 ram2 vs
          vs ≠ [] between (vertices f) ram1 ram2 = []
shows edges f12 = {(hd vs, ram1), (ram1, ram2), (ram2, last vs)} ∪
                Edges(rev vs) (is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs
  define c where c = fst x
  define d where d = snd x
  with c-def have [simp]: x = (c,d) by simp
  from vors have dist-f12: distinct (vertices f12) apply (rule-tac split-face-distinct1)
  by auto
  from x vors show x ∈ ?rhs
    apply (simp add: split-face-def is-nextElem-def is-sublist-def dist-f12)
    apply (case-tac c = ram2 ∧ d = last vs) apply simp
    apply simp apply (elim exE)
    apply (case-tac c = ram1) apply simp
    apply (subgoal-tac rev vs = as) apply simp
    apply (rule dist-at1) apply (rule dist-f12) apply (rule sym) apply simp
  apply simp

```

```

    apply (case-tac c ∈ set(rev vs))
    apply (subgoal-tac distinct(rev vs)) apply (simp only: in-set-conv-decomp)
  apply (elim exE) apply simp
    apply (case-tac zs) apply simp apply (subgoal-tac ys = as) apply (simp
add:rev-swap)
    apply (rule dist-at1) apply (rule dist-f12) apply (rule sym) apply simp
  apply simp apply simp
    apply (subgoal-tac ys = as) apply (simp add: Edges-def is-sublist-def rev-swap)
    apply (rule conjI)
    apply (subgoal-tac ∃ as bs. rev list @ [d, c] = as @ d # c # bs) apply simp
  apply (intro exI) apply simp
    apply (subgoal-tac ∃ asa bs. rev list @ d # c # rev as = asa @ d # c # bs)
  apply simp apply (intro exI) apply simp
    apply (rule dist-at1) apply (rule dist-f12) apply (rule sym) apply simp

```

```

apply simp
  apply (simp add: pre-split-face-def)

  apply (subgoal-tac c = ram2) apply simp
  apply (subgoal-tac rev vs @ [ram1] = as) apply force
  apply (rule dist-at1) apply (rule dist-f12) apply (rule sym) apply simp
apply simp

  apply (subgoal-tac c ∈ set (rev vs @ ram1 # [ram2]))
  apply (thin-tac rev vs @ ram1 # [ram2] = as @ c # d # bs) apply simp
  by simp
next
  fix x
  assume x: x ∈ ?rhs
  define c where c = fst x
  define d where d = snd x
  with c-def have [simp]: x = (c,d) by simp
  from vors have dist-f12: distinct (vertices f12) apply (rule-tac split-face-distinct1)
by auto
  from x vors show x ∈ ?lhs
  apply (simp add: dist-f12 is-nextElem-def is-sublist-def) apply (simp add:
split-face-def)
  apply (case-tac c = hd vs ∧ d = ram1)
  apply (case-tac vs) apply simp
  apply force
  apply simp
  apply (case-tac c = ram1 ∧ d = ram2) apply force
  apply simp
  apply (case-tac c = ram2 ∧ d = last vs)
  apply (case-tac vs rule:rev-exhaust) apply simp
  apply simp
  apply (simp add: Edges-def is-sublist-def)
  apply (elim exE) apply simp apply (rule conjI)
  apply (rule impI) apply (rule disjI1) apply (intro exI)
  apply (subgoal-tac c # d # rev as @ [ram1, ram2] = [] @ c # d # rev as @
[ram1,ram2]) apply assumption apply simp
  apply (rule impI) apply (rule disjI1) apply (intro exI) by simp
qed

lemma split-face-edges-f12-bet-vs:
assumes vors: pre-split-face f ram1 ram2 []
  (f12, f21) = split-face f ram1 ram2 []
  between (vertices f) ram1 ram2 = []
shows edges f12 = {(ram2, ram1) , (ram1, ram2)} (is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs
  define c where c = fst x

```

```

define  $d$  where  $d = \text{snd } x$ 
with  $c\text{-def}$  have  $[simp]: x = (c,d)$  by  $simp$ 
from  $vors$  have  $dist\text{-}f12: \text{distinct } (vertices\ f12)$  apply  $(rule\text{-}tac\ split\text{-}face\text{-}distinct1)$ 
by  $auto$ 
from  $x\ vors$  show  $x \in ?rhs$ 
  apply  $(simp\ add: split\text{-}face\text{-}def\ is\text{-}nextElem\text{-}def\ is\text{-}sublist\text{-}def\ dist\text{-}f12)$ 
  apply  $(case\text{-}tac\ c = ram2 \wedge d = ram1)$  apply  $force$ 
  apply  $simp$  apply  $(elim\ exE)$ 
  apply  $(case\text{-}tac\ c = ram1)$  apply  $simp$ 
  apply  $(case\text{-}tac\ as)$  apply  $simp$ 
  apply  $(case\text{-}tac\ list)$  apply  $simp$  apply  $simp$ 
  apply  $(case\text{-}tac\ as)$  apply  $simp$  apply  $(case\text{-}tac\ list)$  apply  $simp$  by  $simp$ 
next
fix  $x$ 
assume  $x: x \in ?rhs$ 
define  $c$  where  $c = \text{fst } x$ 
define  $d$  where  $d = \text{snd } x$ 
with  $c\text{-def}$  have  $[simp]: x = (c,d)$  by  $simp$ 
from  $vors$  have  $dist\text{-}f12: \text{distinct } (vertices\ f12)$  apply  $(rule\text{-}tac\ split\text{-}face\text{-}distinct1)$ 
by  $auto$ 
from  $x\ vors$  show  $x \in ?lhs$ 
  apply  $(simp\ add: dist\text{-}f12\ is\text{-}nextElem\text{-}def\ is\text{-}sublist\text{-}def)$  apply  $(simp\ add: split\text{-}face\text{-}def)$ 
  by  $auto$ 
qed

```

```

lemma  $split\text{-}face\text{-}edges\text{-}f12\text{-}subset: pre\text{-}split\text{-}face\ f\ ram1\ ram2\ vs \implies$ 
 $(f12, f21) = split\text{-}face\ f\ ram1\ ram2\ vs \implies vs \neq [] \implies$ 
 $\{(hd\ vs, ram1), (ram2, last\ vs)\} \cup Edges(rev\ vs) \subseteq edges\ f12$ 
apply  $(case\text{-}tac\ between\ (vertices\ f)\ ram1\ ram2)$ 
  apply  $(frule\ split\text{-}face\text{-}edges\text{-}f12\text{-}bet)$  apply  $simp$  apply  $simp$  apply  $simp$ 
apply  $force$ 
  apply  $(frule\ split\text{-}face\text{-}edges\text{-}f12)$  apply  $simp+$  by  $force$ 

```

```

lemma  $split\text{-}face\text{-}edges\text{-}f21:$ 
assumes  $vors: pre\text{-}split\text{-}face\ f\ ram1\ ram2\ vs$ 
 $(f12, f21) = split\text{-}face\ f\ ram1\ ram2\ vs$ 
 $vs \neq []\ vs2 = between\ (vertices\ f)\ ram2\ ram1\ vs2 \neq []$ 
shows  $edges\ f21 = \{(last\ vs2, ram1), (ram1, hd\ vs), (last\ vs, ram2), (ram2, hd\ vs2)\} \cup$ 
 $Edges\ vs \cup Edges\ vs2$  (is ?lhs = ?rhs)
proof  $(intro\ equalityI\ subsetI)$ 
fix  $x$ 
assume  $x: x \in ?lhs$ 
define  $c$  where  $c = \text{fst } x$ 
define  $d$  where  $d = \text{snd } x$ 

```

```

with c-def have [simp]: x = (c,d) by simp
from vors have dist-f21: distinct (vertices f21) apply (rule-tac split-face-distinct2)
by auto
from x vors show x ∈ ?rhs
  apply (simp add: split-face-def is-nextElem-def is-sublist-def dist-f21)
  apply (case-tac c = last vs ∧ d = ram2) apply simp
  apply simp apply (elim exE)
  apply (case-tac c = ram1) apply simp
  apply (subgoal-tac ram2 # between (vertices f) ram2 ram1 = as)
  apply clarsimp
  apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp
apply simp

  apply (case-tac c ∈ set vs)
  apply (subgoal-tac distinct vs)
  apply (simp add: in-set-conv-decomp) apply (elim exE) apply simp
  apply (case-tac zs) apply simp
  apply (subgoal-tac ram2 # between (vertices f) ram2 ram1 @ ram1 # ys =
as) apply force
  apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp
apply simp
  apply simp
  apply (subgoal-tac ram2 # between (vertices f) ram2 ram1 @ ram1 # ys =
as)
  apply (subgoal-tac (ram2 # between (vertices f) ram2 ram1 @ ram1 # ys)
@c # a # list = as @ c # d # bs)
  apply (thin-tac ram2 # between (vertices f) ram2 ram1 @ ram1 # ys @ c
# a # list = as @ c # d # bs)
  apply (simp add: Edges-def is-sublist-def)
  apply (rule conjI)
  apply (subgoal-tac ∃ as bs. ys @ [c, d] = as @ c # d # bs) apply simp
apply force
  apply force
  apply force
  apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp
apply simp
  apply (simp add: pre-split-face-def)

  apply (case-tac c ∈ set (between (vertices f) ram2 ram1))
  apply (subgoal-tac distinct (between (vertices f) ram2 ram1)) apply (simp
add: in-set-conv-decomp) apply (elim exE) apply simp
  apply (case-tac zs) apply simp apply (subgoal-tac ram2 # ys = as) apply
force
  apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp
apply simp
  apply simp
  apply (subgoal-tac ram2 # ys = as) apply (simp add: Edges-def is-sublist-def)
  apply (subgoal-tac (ram2 # ys) @ c # a # list @ ram1 # vs = as @ c #
d # bs)

```

```

    apply (thin-tac ram2 # ys @ c # a # list @ ram1 # vs = as @ c # d #
bs)
    apply (rule conjI) apply (rule impI) apply (rule disjI2)+ apply force
    apply (rule impI) apply (rule disjI2)+ apply force
    apply force
    apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp
apply simp
    apply (subgoal-tac distinct (vertices f21))
    apply (thin-tac ram2 # between (vertices f) ram2 ram1 @ ram1 # vs = as
@ c # d # bs) apply simp
    apply (rule dist-f21)

    apply (subgoal-tac c = ram2) apply simp
    apply (subgoal-tac [] = as) apply simp apply (case-tac between (vertices f)
ram2 ram1) apply simp apply simp
    apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp
apply simp

    apply (subgoal-tac c ∈ set (ram2 # between (vertices f) ram2 ram1 @ ram1
# vs))
    apply (thin-tac ram2 # between (vertices f) ram2 ram1 @ ram1 # vs = as
@ c # d # bs) apply simp
    by simp
next
fix x
assume x: x ∈ ?rhs
define c where c = fst x
define d where d = snd x
with c-def have [simp]: x = (c,d) by simp
from vors have dist-f21: distinct (vertices f21) apply (rule-tac split-face-distinct2)
by auto
from x vors show x ∈ ?lhs
supply [[simproc del: defined-all]]
    apply (simp add: dist-f21 is-nextElem-def is-sublist-def) apply (simp add:
split-face-def)
    apply (case-tac c = ram2 ∧ d = hd (between (vertices f) ram2 ram1)) apply
simp apply (rule disjI1)
    apply (intro exI) apply (subgoal-tac ram2 # between (vertices f) ram2 ram1
@ ram1 # vs =
    [] @ ram2 # hd (between (vertices f) ram2 ram1) # tl (between (vertices f)
ram2 ram1) @ ram1 # vs) apply assumption apply simp
    apply (case-tac c = ram1 ∧ d = hd vs) apply (rule disjI1)
    apply (case-tac vs) apply simp
    apply simp apply (intro exI)
    apply (subgoal-tac ram2 # between (vertices f) ram2 ram1 @ ram1 # a #
list =
    (ram2 # between (vertices f) ram2 ram1) @ ram1 # a # list) apply
assumption apply simp

```

```

apply (case-tac c = last vs  $\wedge$  d = ram2)
apply (case-tac vs rule:rev-exhaust) apply simp
apply simp
apply simp
apply (case-tac c = last (between (vertices f) ram2 ram1)  $\wedge$  d = ram1) apply
(rule disjI1)
apply (case-tac between (vertices f) ram2 ram1 rule: rev-exhaust) apply simp
apply (intro exI) apply simp
apply (subgoal-tac ram2 # ys @ y # ram1 # vs = (ram2 # ys) @ y # ram1
# vs)
apply assumption
apply simp
apply simp
apply (case-tac (c, d)  $\in$  Edges vs) apply (simp add: Edges-def is-sublist-def)
apply (elim exE) apply simp apply (rule conjI) apply (rule impI) apply
(rule disjI1) apply (intro exI)
apply (subgoal-tac ram2 # between (vertices f) ram2 ram1 @ ram1 # as @
[c, d]
= (ram2 # between (vertices f) ram2 ram1 @ ram1 # as) @ c # d # [])
apply assumption apply simp
apply (rule impI) apply (rule disjI1) apply (intro exI)
apply (subgoal-tac ram2 # between (vertices f) ram2 ram1 @ ram1 # as @
c # d # bs
= (ram2 # between (vertices f) ram2 ram1 @ ram1 # as) @ c # d # bs)
apply assumption apply simp
apply (simp add: Edges-def is-sublist-def)
apply (elim exE) apply simp apply (rule disjI1) apply (intro exI)
apply (subgoal-tac ram2 # as @ c # d # bs @ ram1 # vs = (ram2 # as) @
c # d # (bs @ ram1 # vs))
apply assumption by simp
qed

```

```

lemma split-face-edges-f21-vs:
assumes vors: pre-split-face f ram1 ram2 []
(f12, f21) = split-face f ram1 ram2 []
vs2 = between (vertices f) ram2 ram1 vs2  $\neq$  []
shows edges f21 = {(last vs2, ram1), (ram1, ram2), (ram2, hd vs2)}  $\cup$ 
Edges vs2 (is ?lhs = ?rhs)
proof (intro equalityI subsetI)
fix x
assume x: x  $\in$  ?lhs
define c where c = fst x
define d where d = snd x
with c-def have [simp]: x = (c,d) by simp
from vors have dist-f21: distinct (vertices f21) apply (rule-tac split-face-distinct2)
by auto
from x vors show x  $\in$  ?rhs
apply (simp add: split-face-def is-nextElem-def is-sublist-def dist-f21)

```

```

apply (case-tac  $c = ram1 \wedge d = ram2$ ) apply simp apply simp apply (elim
exE)

apply (case-tac  $c = ram1$ ) apply simp
apply (subgoal-tac  $ram2 \# between (vertices f) ram2 ram1 = as$ )
apply (subgoal-tac ( $ram2 \# between (vertices f) ram2 ram1$ ) @ [ram1] = as
@ ram1 # d # bs)
apply (thin-tac  $ram2 \# between (vertices f) ram2 ram1 @ [ram1] = as @$ 
ram1 # d # bs)
apply simp apply force
apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp apply
simp

apply (case-tac  $c \in set (between (vertices f) ram2 ram1)$ )
apply (subgoal-tac distinct ( $between (vertices f) ram2 ram1$ )) apply (simp add:
in-set-conv-decomp) apply (elim exE) apply simp
apply (case-tac  $zs$ ) apply simp apply (subgoal-tac  $ram2 \# ys = as$ ) apply
force
apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp apply
simp apply simp
apply (subgoal-tac  $ram2 \# ys = as$ ) apply (simp add: Edges-def is-sublist-def)
apply (subgoal-tac ( $ram2 \# ys$ ) @ c # a # list @ [ram1] = as @ c # d # bs)
apply (thin-tac  $ram2 \# ys @ c # a # list @ [ram1] = as @ c # d # bs$ )
apply (rule conjI) apply (rule impI) apply (rule disjI2)+ apply force
apply (rule impI) apply (rule disjI2)+ apply force apply force
apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp apply
simp
apply (subgoal-tac distinct ( $vertices f21$ ))
apply (thin-tac  $ram2 \# between (vertices f) ram2 ram1 @ [ram1] = as @ c \#$ 
d # bs) apply simp
apply (rule dist-f21)

apply (subgoal-tac  $c = ram2$ ) apply simp
apply (subgoal-tac [] = as) apply simp apply (case-tac  $between (vertices f)$ 
ram2 ram1) apply simp apply simp
apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp
apply simp

apply (subgoal-tac  $c \in set (ram2 \# between (vertices f) ram2 ram1 @ [ram1])$ )
apply (thin-tac  $ram2 \# between (vertices f) ram2 ram1 @ [ram1] = as @ c \#$ 
d # bs) apply simp
by simp
next
fix x
assume x:  $x \in ?rhs$ 
define c where  $c = fst x$ 
define d where  $d = snd x$ 
with c-def have [simp]:  $x = (c,d)$  by simp

```

```

from vors have dist-f21: distinct (vertices f21) apply (rule-tac split-face-distinct2)
by auto
from x vors show  $x \in ?lhs$ 
  supply [simproc del: defined-all]
  apply (simp add: dist-f21 is-nextElem-def is-sublist-def) apply (simp add: split-face-def)
  apply (case-tac c = ram2  $\wedge$  d = hd (between (vertices f) ram2 ram1)) apply
simp apply (rule disjI1)
  apply (intro exI) apply (subgoal-tac ram2 # between (vertices f) ram2 ram1
  @ [ram1] =
    [] @ ram2 # hd (between (vertices f) ram2 ram1) # tl (between (vertices f)
ram2 ram1) @ [ram1]) apply assumption apply simp
  apply (case-tac c = ram1  $\wedge$  d = ram2) apply (rule disjI2) apply simp apply
simp
  apply (case-tac c = last (between (vertices f) ram2 ram1)  $\wedge$  d = ram1) apply
(rule disjI1)
  apply (case-tac between (vertices f) ram2 ram1 rule: rev-exhaust) apply simp
  apply (intro exI) apply simp
  apply (subgoal-tac ram2 # ys @ y # [ram1] = (ram2 # ys) @ y # [ram1])
  apply assumption apply simp apply simp
apply (simp add: Edges-def is-sublist-def)
  apply (elim exE) apply simp apply (rule disjI1) apply (intro exI)
  apply (subgoal-tac ram2 # as @ c # d # bs @ [ram1] = (ram2 # as) @ c
# d # (bs @ [ram1]))
  apply assumption by simp
qed

```

lemma *split-face-edges-f21-bet*:

```

assumes vors: pre-split-face f ram1 ram2 vs
  (f12, f21) = split-face f ram1 ram2 vs
   $vs \neq []$  between (vertices f) ram2 ram1 = []
shows edges f21 = {(ram1, hd vs), (last vs, ram2), (ram2, ram1)}  $\cup$ 
  Edges vs (is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x:  $x \in ?lhs$ 
  define c where  $c = fst\ x$ 
  define d where  $d = snd\ x$ 
  with c-def have [simp]:  $x = (c, d)$  by simp
from vors have dist-f21: distinct (vertices f21) apply (rule-tac split-face-distinct2)
by auto
from x vors show  $x \in ?rhs$ 
  apply (simp add: split-face-def is-nextElem-def is-sublist-def dist-f21)
  apply (case-tac c = last vs  $\wedge$  d = ram2) apply simp
  apply simp apply (elim exE)
  apply (case-tac c = ram1) apply simp
  apply (subgoal-tac [ram2] = as) apply clarsimp
  apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp

```



```

apply simp

  apply (case-tac  $c \in \text{set } vs$ )
    apply (subgoal-tac distinct vs) apply (simp add: in-set-conv-decomp) apply
(elim exE) apply simp
    apply (case-tac zs) apply simp
    apply (subgoal-tac  $\text{ram2} \# \text{ram1} \# \text{ys} = \text{as}$ ) apply force
    apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp
apply simp
  apply simp
  apply (subgoal-tac  $\text{ram2} \# \text{ram1} \# \text{ys} = \text{as}$ )
  apply (subgoal-tac ( $\text{ram2} \# \text{ram1} \# \text{ys}$ ) @  $c \# a \# \text{list} = \text{as} @ c \# d \#$ 
bs)
    apply (thin-tac  $\text{ram2} \# \text{ram1} \# \text{ys} @ c \# a \# \text{list} = \text{as} @ c \# d \# \text{bs}$ )
    apply (simp add: Edges-def is-sublist-def) apply force
    apply force
    apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp
apply simp
  apply (simp add: pre-split-face-def)

  apply (subgoal-tac  $c = \text{ram2}$ ) apply simp
  apply (subgoal-tac  $[] = \text{as}$ ) apply simp
  apply (rule dist-at1) apply (rule dist-f21) apply (rule sym) apply simp
apply simp

  apply (subgoal-tac  $c \in \text{set} (\text{ram2} \# \text{ram1} \# \text{vs})$ )
  apply (thin-tac  $\text{ram2} \# \text{ram1} \# \text{vs} = \text{as} @ c \# d \# \text{bs}$ ) apply simp
  by simp
next
  fix  $x$ 
  assume  $x: x \in ?\text{rhs}$ 
  define  $c$  where  $c = \text{fst } x$ 
  define  $d$  where  $d = \text{snd } x$ 
  with  $c\text{-def}$  have [simp]:  $x = (c, d)$  by simp
  from  $\text{vors}$  have dist-f21: distinct (vertices f21) apply (rule-tac split-face-distinct2)
by auto
  from  $x \text{ vors}$  show  $x \in ?\text{lhs}$ 
  supply [[simproc del: defined-all]]
  apply (simp add: dist-f21 is-nextElem-def is-sublist-def) apply (simp add:
split-face-def)
  apply (case-tac  $c = \text{ram2} \wedge d = \text{ram1}$ ) apply simp apply (rule disjI1) apply
force
  apply (case-tac  $c = \text{ram1} \wedge d = \text{hd } \text{vs}$ ) apply (rule disjI1)
  apply (case-tac vs) apply simp
  apply simp apply (intro exI)
  apply (subgoal-tac  $\text{ram2} \# \text{ram1} \# a \# \text{list} =$ 
 $[\text{ram2}] @ \text{ram1} \# a \# \text{list}$ ) apply assumption apply simp
  apply (case-tac  $c = \text{last } \text{vs} \wedge d = \text{ram2}$ )
  apply (case-tac vs rule: rev-exhaust) apply simp

```

```

apply simp
apply (simp add: Edges-def is-sublist-def)
apply (elim exE) apply simp apply (rule conjI) apply (rule impI) apply
(rule disjI1) apply (intro exI)
apply (subgoal-tac ram2 # ram1 # as @ [c, d]
  = (ram2 # ram1 # as) @ c # d # []) apply assumption apply simp
apply (rule impI) apply (rule disjI1) apply (intro exI)
apply (subgoal-tac ram2 # ram1 # as @ c # d # bs
  = (ram2 # ram1 # as) @ c # d # bs) apply assumption by simp
qed

```

```

lemma split-face-edges-f21-bet-vs:
assumes vors: pre-split-face f ram1 ram2 []
  (f12, f21) = split-face f ram1 ram2 []
  between (vertices f) ram2 ram1 = []
shows edges f21 = {(ram1, ram2), (ram2, ram1)} (is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs
  define c where c = fst x
  define d where d = snd x
  with c-def have [simp]: x = (c,d) by simp
  from vors have dist-f21: distinct (vertices f21) apply (rule-tac split-face-distinct2)
by auto
  from x vors show x ∈ ?rhs
  apply (simp add: split-face-def is-nextElem-def is-sublist-def dist-f21)
  apply (case-tac c = ram1 ∧ d = ram2) apply simp apply simp apply (elim
exE)
  apply (case-tac as) apply simp apply (case-tac list) by auto
next
  fix x
  assume x: x ∈ ?rhs
  define c where c = fst x
  define d where d = snd x
  with c-def have [simp]: x = (c,d) by simp
  from vors have dist-f21: distinct (vertices f21) apply (rule-tac split-face-distinct2)
by auto
  from x vors show x ∈ ?lhs
  apply (simp add: dist-f21 is-nextElem-def is-sublist-def) apply (simp add:
split-face-def)
  by auto
qed

```

```

lemma split-face-edges-f21-subset: pre-split-face f ram1 ram2 vs ⇒
  (f12, f21) = split-face f ram1 ram2 vs ⇒ vs ≠ [] ⇒
  {(last vs, ram2), (ram1, hd vs)} ∪ Edges vs ⊆ edges f21
apply (case-tac between (vertices f) ram2 ram1)
  apply (frule split-face-edges-f21-bet) apply simp apply simp apply simp

```

apply force
apply (*frule split-face-edges-f21*) **apply simp+ by force**

lemma verticesFrom-ram1: pre-split-face f ram1 ram2 vs \implies
verticesFrom f ram1 = ram1 # between (vertices f) ram1 ram2 @ ram2 #
between (vertices f) ram2 ram1
apply (*subgoal-tac pre-between (vertices f) ram1 ram2*)
apply (*subgoal-tac distinct (vertices f)*)
apply (*case-tac before (vertices f) ram1 ram2*)
apply (*simp add: verticesFrom-Def*)
apply (*subgoal-tac ram2 \in set (snd (splitAt ram1 (vertices f)))*) **apply** (*drule splitAt-ram*)
apply (*subgoal-tac snd (splitAt ram2 (snd (splitAt ram1 (vertices f)))) = snd (splitAt ram2 (vertices f))*)
apply simp **apply** (*thin-tac snd (splitAt ram1 (vertices f)) =*
fst (splitAt ram2 (snd (splitAt ram1 (vertices f)))) @
ram2 # snd (splitAt ram2 (snd (splitAt ram1 (vertices f))))) **apply simp**
apply (*rule before-dist-r2*) **apply simp** **apply simp**
apply (*subgoal-tac before (vertices f) ram2 ram1*)
apply (*subgoal-tac pre-between (vertices f) ram2 ram1*)
apply (*simp add: verticesFrom-Def*)
apply (*subgoal-tac ram2 \in set (fst (splitAt ram1 (vertices f)))*) **apply** (*drule splitAt-ram*)
apply (*subgoal-tac fst (splitAt ram2 (fst (splitAt ram1 (vertices f)))) = fst (splitAt ram2 (vertices f))*)
apply simp **apply** (*thin-tac fst (splitAt ram1 (vertices f)) =*
fst (splitAt ram2 (fst (splitAt ram1 (vertices f)))) @
ram2 # snd (splitAt ram2 (fst (splitAt ram1 (vertices f))))) **apply simp**
apply (*rule before-dist-r1*) **apply simp** **apply simp** **apply** (*simp add: pre-between-def*)
apply force
apply (*simp add: pre-split-face-def*) **by** (*rule pre-split-face-p-between*)

lemma split-face-edges-f-vs1-vs2:
assumes *vors: pre-split-face f ram1 ram2 vs*
between (vertices f) ram1 ram2 = []
between (vertices f) ram2 ram1 = []
shows *edges f = {(ram2, ram1), (ram1, ram2)}* (**is** *?lhs = ?rhs*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x: x \in ?lhs*
define *c* **where** *c = fst x*
define *d* **where** *d = snd x*
with *c-def* **have** [*simp*]: *x = (c,d)* **by** *simp*
from *vors* **have** *dist-f: distinct (vertices f)* **by** (*simp add: pre-split-face-def*)
from *x vors* **show** *x \in ?rhs* **apply** (*simp add: dist-f*)
apply (*subgoal-tac pre-between (vertices f) ram1 ram2*)
apply (*drule is-nextElem-or*) **apply** *assumption*
apply (*simp add: Edges-def*)
apply (*case-tac is-sublist [c, d] [ram1, ram2]*) **apply** (*simp*)

```

    apply (simp) apply blast
  by (rule pre-split-face-p-between)
next
fix x
assume x: x ∈ ?rhs
define c where c = fst x
define d where d = snd x
with c-def have [simp]: x = (c,d) by simp
from vors have dist-f: distinct (vertices f) by (simp add: pre-split-face-def)
from x vors show x ∈ ?lhs apply (simp add: dist-f)
  apply (subgoal-tac ram1 ∈ V f) apply (simp add: verticesFrom-is-nextElem
verticesFrom-ram1)
  apply (simp add: is-nextElem-def) apply blast
  by (simp add: pre-split-face-def)
qed

```

lemma split-face-edges-f-vs1:

```

assumes vors: pre-split-face f ram1 ram2 vs
          between (vertices f) ram1 ram2 = []
          vs2 = between (vertices f) ram2 ram1 vs2 ≠ []
shows edges f = {(last vs2, ram1), (ram1, ram2), (ram2, hd vs2)} ∪
              Edges vs2 (is ?lhs = ?rhs)

```

proof (intro equalityI subsetI)

```

  fix x
  assume x: x ∈ ?lhs
  define c where c = fst x
  define d where d = snd x
  with c-def have [simp]: x = (c,d) by simp
  from vors have dist-f: distinct (vertices f) by (simp add: pre-split-face-def)
  from vors have dist-vs2: distinct (ram2 # vs2 @ [ram1]) apply (simp only:)
    apply (rule between-distinct-r12) apply (rule dist-f) apply (rule not-sym) by
(simp add: pre-split-face-def)
  from x vors show x ∈ ?rhs apply (simp add: dist-f)
    apply (subgoal-tac pre-between (vertices f) ram1 ram2)
    apply (drule is-nextElem-or) apply assumption
    apply (simp add: Edges-def)
    apply (case-tac is-sublist [c, d] [ram1, ram2])
      apply simp
      apply simp
    apply (erule disjE) apply blast
    apply (case-tac c = ram2)
      apply (case-tac between (vertices f) ram2 ram1) apply simp
      apply simp
    apply (drule is-sublist-distinct-prefix)
      apply (subgoal-tac distinct (ram2 # vs2 @ [ram1]))
        apply simp
        apply (rule dist-vs2)
      apply simp
    apply (case-tac c = ram1)

```

```

apply (subgoal-tac  $\neg$  is-sublist [c, d] (ram2 # vs2 @ [ram1]))
apply simp
apply (rule is-sublist-notlast)
apply (rule-tac dist-vs2)
apply simp
apply simp
apply (simp add: is-sublist-def)
apply (elim exE)
apply (case-tac between (vertices f) ram2 ram1 rule: rev-exhaust) apply simp
apply simp
apply (case-tac bs rule: rev-exhaust) apply simp
apply simp
apply (rule disjI2)
apply (intro exI)
apply simp
apply (rule pre-split-face-p-between) by simp
next
fix x
assume x: x  $\in$  ?rhs
define c where c = fst x
define d where d = snd x
with c-def have [simp]: x = (c,d) by simp
from vors have dist-f: distinct (vertices f) by (simp add: pre-split-face-def)
from vors have dist-vs2: distinct (ram2 # vs2 @ [ram1]) apply (simp only:)
apply (rule between-distinct-r12) apply (rule dist-f) apply (rule not-sym) by
(simp add: pre-split-face-def)
from x vors show x  $\in$  ?lhs
supply [[simproc del: defined-all]]
apply (simp add: dist-f)
apply (subgoal-tac ram1  $\in$  set (vertices f)) apply (simp add: verticesFrom-is-nextElem
verticesFrom-ram1)
apply (simp add: is-nextElem-def)
apply (case-tac c = last (between (vertices f) ram2 ram1)  $\wedge$  d = ram1) apply
simp apply simp apply (rule disjI1)
apply (case-tac c = ram1  $\wedge$  d = ram2) apply (simp add: is-sublist-def) apply
force apply simp
apply (case-tac c = ram2  $\wedge$  d = hd (between (vertices f) ram2 ram1))
apply (case-tac between (vertices f) ram2 ram1) apply simp apply (simp
add: is-sublist-def) apply (intro exI)
apply (subgoal-tac ram1 # ram2 # a # list =
[ram1] @ ram2 # a # (list)) apply assumption apply simp
apply simp
apply (subgoal-tac is-sublist [c, d] ((ram1 #
[ram2]) @ between (vertices f) ram2 ram1 @ []))
apply simp apply (rule is-sublist-add) apply (simp add: Edges-def)
by (simp add: pre-split-face-def)
qed

```

```

lemma split-face-edges-f-vs2:
assumes vors: pre-split-face f ram1 ram2 vs
           vs1 = between (vertices f) ram1 ram2 vs1 ≠ []
           between (vertices f) ram2 ram1 = []
shows edges f = {(ram2, ram1) , (ram1, hd vs1), (last vs1, ram2)} ∪
        Edges vs1 (is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs
  define c where c = fst x
  define d where d = snd x
  with c-def have [simp]: x = (c,d) by simp
  from vors have dist-f: distinct (vertices f) by (simp add: pre-split-face-def)
  from vors have dist-vs1: distinct (ram1 # vs1 @ [ram2]) apply (simp only:)
  apply (rule between-distinct-r12) apply (rule dist-f) by (simp add: pre-split-face-def)
  from x vors show x ∈ ?rhs apply (simp add: dist-f)
  apply (subgoal-tac pre-between (vertices f) ram1 ram2)
  apply (drule is-nextElem-or) apply assumption
  apply (simp add: Edges-def)
  apply (case-tac is-sublist [c, d] (ram1 # between (vertices f) ram1 ram2 @
[ram2]))
  apply simp
  apply (case-tac c = ram1)
  apply (case-tac between (vertices f) ram1 ram2) apply simp
  apply simp
  apply (drule is-sublist-distinct-prefix)
  apply (subgoal-tac distinct (ram1 # vs1 @ [ram2])) apply simp
  apply (rule dist-vs1)
  apply simp
  apply (case-tac c = ram2)
  apply (subgoal-tac ¬ is-sublist [c, d] (ram1 # vs1 @ [ram2])) apply simp
  apply (rule is-sublist-notlast) apply (rule-tac dist-vs1)
  apply simp
  apply simp
  apply (simp add: is-sublist-def)
  apply (elim exE)
  apply (case-tac between (vertices f) ram1 ram2 rule: rev-exhaust) apply simp
  apply simp
  apply (case-tac bs rule: rev-exhaust) apply simp
  apply simp
  apply (rule disjI2)
  apply (intro exI)
  apply simp
  apply simp
  apply (rule pre-split-face-p-between) by simp
next
fix x
assume x: x ∈ ?rhs
define c where c = fst x

```

```

define d where d = snd x
with c-def have [simp]: x = (c,d) by simp
from vors have dist-f: distinct (vertices f) by (simp add: pre-split-face-def)
from vors have dist-vs1: distinct (ram1 # vs1 @ [ram2]) apply (simp only:)
apply (rule between-distinct-r12) apply (rule dist-f) by (simp add: pre-split-face-def)
from x vors show x ∈ ?lhs
supply [[simproc del: defined-all]]
apply (simp add: dist-f)
apply (subgoal-tac ram1 ∈  $\mathcal{V}$  f) apply (simp add: verticesFrom-is-nextElem
verticesFrom-ram1)
apply (simp add: is-nextElem-def)
apply (case-tac c = ram2 ∧ d = ram1) apply simp apply simp apply (rule
disjI1)
apply (case-tac c = ram1 ∧ d = hd (between (vertices f) ram1 ram2))
apply (case-tac between (vertices f) ram1 ram2) apply simp apply (force
simp: is-sublist-def) apply simp
apply (case-tac c = last (between (vertices f) ram1 ram2) ∧ d = ram2)
apply (case-tac between (vertices f) ram1 ram2 rule: rev-exhaust) apply simp
apply (simp add: is-sublist-def)
apply (intro exI)
apply (subgoal-tac ram1 # ys @ [y, ram2] =
(ram1 # ys) @ y # ram2 # []) apply assumption apply simp
apply simp
apply (simp add: Edges-def)
apply (subgoal-tac is-sublist [c, d] ([ram1] @ between (vertices f) ram1 ram2
@ [ram2]))
apply simp apply (rule is-sublist-add) apply simp
by (simp add: pre-split-face-def)
qed

```

lemma *split-face-edges-f*:

assumes *vors*: *pre-split-face* *f* *ram1* *ram2* *vs*

$vs1 = \text{between} (\text{vertices } f) \text{ ram1 ram2 } vs1 \neq []$

$vs2 = \text{between} (\text{vertices } f) \text{ ram2 ram1 } vs2 \neq []$

shows $\text{edges } f = \{(\text{last } vs2, \text{ram1}), (\text{ram1}, \text{hd } vs1), (\text{last } vs1, \text{ram2}), (\text{ram2}, \text{hd } vs2)\} \cup$

$\text{Edges } vs1 \cup \text{Edges } vs2$ (**is** ?*lhs* = ?*rhs*)

proof (*intro* *equalityI* *subsetI*)

fix *x*

assume *x*: *x* ∈ ?*lhs*

define *c* **where** *c* = *fst* *x*

define *d* **where** *d* = *snd* *x*

with *c-def* **have** [*simp*]: *x* = (*c*,*d*) **by** *simp*

from *vors* **have** *dist-f*: *distinct* (*vertices* *f*) **by** (*simp* *add*: *pre-split-face-def*)

from *vors* **have** *dist-vs1*: *distinct* (*ram1* # *vs1* @ [*ram2*]) **apply** (*simp* *only*:)

apply (*rule* *between-distinct-r12*) **apply** (*rule* *dist-f*) **by** (*simp* *add*: *pre-split-face-def*)

from *vors* **have** *dist-vs2*: *distinct* (*ram2* # *vs2* @ [*ram1*]) **apply** (*simp* *only*:)

apply (*rule* *between-distinct-r12*) **apply** (*rule* *dist-f*) **apply** (*rule* *not-sym*) **by**

```

(simp add: pre-split-face-def)
  from x vors show x ∈ ?rhs apply (simp add: dist-f)
    apply (subgoal-tac pre-between (vertices f) ram1 ram2)
    apply (drule is-nextElem-or) apply assumption apply (simp add: Edges-def)
    apply (case-tac is-sublist [c, d] (ram1 # between (vertices f) ram1 ram2 @
[ram2])) apply simp
      apply (case-tac c = ram1)
        apply (case-tac between (vertices f) ram1 ram2) apply simp apply simp
        apply (drule is-sublist-distinct-prefix) apply (subgoal-tac distinct (ram1 #
vs1 @ [ram2]))
          apply simp apply (rule dist-vs1) apply simp
          apply (case-tac c = ram2)
            apply (subgoal-tac ¬ is-sublist [c, d] (ram1 # vs1 @ [ram2])) apply simp
            apply (rule is-sublist-notlast) apply (rule-tac dist-vs1) apply simp
            apply simp apply (simp add: is-sublist-def) apply (elim exE)
              apply (case-tac between (vertices f) ram1 ram2 rule: rev-exhaust) apply
simp apply simp
                apply (case-tac bs rule: rev-exhaust) apply simp apply simp
                apply (rule disjI2) apply (rule disjI2) apply (rule disjI1) apply (intro
exI) apply simp
                apply simp
                apply (case-tac c = ram2)
                  apply (case-tac between (vertices f) ram2 ram1) apply simp apply simp
                  apply (drule is-sublist-distinct-prefix) apply (subgoal-tac distinct (ram2 #
vs2 @ [ram1]))
                    apply simp apply (rule dist-vs2) apply simp
                    apply (case-tac c = ram1)
                      apply (subgoal-tac ¬ is-sublist [c, d] (ram2 # vs2 @ [ram1])) apply simp
                      apply (rule is-sublist-notlast) apply (rule-tac dist-vs2) apply simp
                      apply simp apply (simp add: is-sublist-def) apply (elim exE)
                        apply (case-tac between (vertices f) ram2 ram1 rule: rev-exhaust) apply
simp apply simp
                          apply (case-tac bs rule: rev-exhaust) apply simp apply simp
                          apply (rule disjI2) apply (rule disjI2) apply (rule disjI2) apply (intro
exI) apply simp
                          apply (rule pre-split-face-p-between) by simp
                    next
                      fix x
                      assume x: x ∈ ?rhs
                      define c where c = fst x
                      define d where d = snd x
                      with c-def have [simp]: x = (c,d) by simp
                      from vors have dist-f: distinct (vertices f) by (simp add: pre-split-face-def)
                      from vors have dist-vs1: distinct (ram1 # vs1 @ [ram2]) apply (simp only:)
                        apply (rule between-distinct-r12) apply (rule dist-f) by (simp add: pre-split-face-def)
                      from vors have dist-vs2: distinct (ram2 # vs2 @ [ram1]) apply (simp only:)
                        apply (rule between-distinct-r12) apply (rule dist-f) apply (rule not-sym) by
(simp add: pre-split-face-def)
                      from x vors show x ∈ ?lhs

```


supply $[[\text{simproc del: defined-all}]]$
apply (simp add: dist-f)
apply (subgoal-tac $\text{ram1} \in \mathcal{V} f$) **apply** (simp add: verticesFrom-is-nextElem
verticesFrom-ram1)
apply (simp add: is-nextElem-def)
apply (case-tac $c = \text{last} (\text{between} (\text{vertices } f) \text{ram2 } \text{ram1}) \wedge d = \text{ram1}$) **apply**
simp **apply** simp **apply** (rule disjI1)
apply (case-tac $c = \text{ram1} \wedge d = \text{hd} (\text{between} (\text{vertices } f) \text{ram1 } \text{ram2})$)
apply (case-tac $\text{between} (\text{vertices } f) \text{ram1 } \text{ram2}$) **apply** simp **apply** (force
simp: is-sublist-def) **apply** simp
apply (case-tac $c = \text{last} (\text{between} (\text{vertices } f) \text{ram1 } \text{ram2}) \wedge d = \text{ram2}$)
apply (case-tac $\text{between} (\text{vertices } f) \text{ram1 } \text{ram2}$ rule: rev-exhaust) **apply** simp
apply (simp add: is-sublist-def)
apply (intro exI)
apply (subgoal-tac $\text{ram1} \# \text{ys} @ y \# \text{ram2} \# \text{between} (\text{vertices } f) \text{ram2 } \text{ram1}$
=
 $(\text{ram1} \# \text{ys}) @ y \# \text{ram2} \# (\text{between} (\text{vertices } f) \text{ram2 } \text{ram1})$) **apply**
assumption **apply** simp **apply** simp
apply (case-tac $c = \text{ram2} \wedge d = \text{hd} (\text{between} (\text{vertices } f) \text{ram2 } \text{ram1})$)
apply (case-tac $\text{between} (\text{vertices } f) \text{ram2 } \text{ram1}$) **apply** simp **apply** (simp
add: is-sublist-def) **apply** (intro exI)
apply (subgoal-tac $\text{ram1} \# \text{between} (\text{vertices } f) \text{ram1 } \text{ram2} @ \text{ram2} \# a \#$
list =
 $(\text{ram1} \# \text{between} (\text{vertices } f) \text{ram1 } \text{ram2}) @ \text{ram2} \# a \# (\text{list})$) **apply**
assumption **apply** simp **apply** simp
apply (case-tac $(c, d) \in \text{Edges} (\text{between} (\text{vertices } f) \text{ram1 } \text{ram2})$) **apply** (simp
add: Edges-def)
apply (subgoal-tac is-sublist [c, d] ($[\text{ram1}] @ \text{between} (\text{vertices } f) \text{ram1 } \text{ram2}$
@
 $(\text{ram2} \# \text{between} (\text{vertices } f) \text{ram2 } \text{ram1}))$)
apply simp **apply** (rule is-sublist-add) **apply** simp
apply simp
apply (subgoal-tac is-sublist [c, d] ($(\text{ram1} \# \text{between} (\text{vertices } f) \text{ram1 } \text{ram2}$
@
 $[\text{ram2}]) @ \text{between} (\text{vertices } f) \text{ram2 } \text{ram1} @ []$))
apply simp **apply** (rule is-sublist-add) **apply** (simp add: Edges-def)
by (simp add: pre-split-face-def)
qed

lemma split-face-edges-f12-f21:

$\text{pre-split-face } f \text{ram1 } \text{ram2 } \text{vs} \implies (f12, f21) = \text{split-face } f \text{ram1 } \text{ram2 } \text{vs} \implies$
 $\text{vs} \neq []$
 $\implies \text{edges } f12 \cup \text{edges } f21 = \text{edges } f \cup$
 $\{(hd \text{ vs}, \text{ram1}), (\text{ram1}, hd \text{ vs}), (\text{last } \text{vs}, \text{ram2}), (\text{ram2}, \text{last } \text{vs})\} \cup$
 $\text{Edges } \text{vs} \cup$
 $\text{Edges } (\text{rev } \text{vs})$
apply (case-tac $\text{between} (\text{vertices } f) \text{ram1 } \text{ram2} = []$)
apply (case-tac $\text{between} (\text{vertices } f) \text{ram2 } \text{ram1} = []$)

apply (*simp add: split-face-edges-f12-bet split-face-edges-f21-bet split-face-edges-f-vs1-vs2*)
apply force
apply (*simp add: split-face-edges-f12-bet split-face-edges-f21 split-face-edges-f-vs1*)
apply force
apply (*case-tac between (vertices f) ram2 ram1 = []*)
apply (*simp add: split-face-edges-f21-bet split-face-edges-f12 split-face-edges-f-vs2*)
apply force
apply (*simp add: split-face-edges-f21 split-face-edges-f12 split-face-edges-f*) **by**
force

lemma *split-face-edges-f12-f21-vs:*

pre-split-face f ram1 ram2 [] \implies (f12, f21) = split-face f ram1 ram2 []
 \implies *edges f12 \cup edges f21 = edges f \cup*
{(ram2, ram1), (ram1, ram2)}
apply (*case-tac between (vertices f) ram1 ram2 = []*)
apply (*case-tac between (vertices f) ram2 ram1 = []*)
apply (*simp add: split-face-edges-f12-bet-vs split-face-edges-f21-bet-vs split-face-edges-f-vs1-vs2*)
apply force
apply (*simp add: split-face-edges-f12-bet-vs split-face-edges-f21-vs split-face-edges-f-vs1*)
apply force
apply (*case-tac between (vertices f) ram2 ram1 = []*)
apply (*simp add: split-face-edges-f21-bet-vs split-face-edges-f12-vs split-face-edges-f-vs2*)
apply force
apply (*simp add: split-face-edges-f21-vs split-face-edges-f12-vs split-face-edges-f*)
by force

lemma *split-face-edges-f12-f21-sym:*

f $\in \mathcal{F}$ g \implies
pre-split-face f ram1 ram2 vs \implies (f12, f21) = split-face f ram1 ram2 vs
 \implies *((a,b) \in edges f12 \vee (a,b) \in edges f21) =*
((a,b) \in edges f \vee
((b,a) \in edges f12 \vee (b,a) \in edges f21) \wedge
((a,b) \in edges f12 \vee (a,b) \in edges f21)))
apply (*subgoal-tac ((a,b) \in edges f12 \cup edges f21) =*
((a,b) \in edges f \vee ((b,a) \in edges f12 \cup edges f21) \wedge (a,b) \in edges f12 \cup edges
f21)) **apply force**
apply (*case-tac vs = []*)
apply (*subgoal-tac pre-split-face f ram1 ram2 []*)
apply (*drule split-face-edges-f12-f21-vs*) **apply simp apply simp apply force**
apply simp
apply (*drule split-face-edges-f12-f21*) **apply simp apply simp**
apply simp by force

lemma *splitFace-edges-g'-help: pre-splitFace g ram1 ram2 f vs \implies*

(f12, f21, g') = splitFace g ram1 ram2 f vs \implies vs \neq [] \implies
edges g' = edges g \cup edges f \cup Edges vs \cup Edges(rev vs) \cup
{(ram2, last vs), (hd vs, ram1), (ram1, hd vs), (last vs, ram2)}

proof –

```
assume pre: pre-splitFace g ram1 ram2 f vs
and fdg: (f12, f21, g') = splitFace g ram1 ram2 f vs
and vs-neq: vs ≠ []

from pre fdg have split: (f12, f21) = split-face f ram1 ram2 vs
apply (unfold pre-splitFace-def) apply (elim conjE)
by (simp add: splitFace-split-face)

from fdg pre have edges g' = (⋃a∈set (replace f [f21] (faces g)) edges a) ∪
edges (f12) by (auto simp: splitFace-def split-def edges-graph-def)
with pre vs-neq show ?thesis apply (simp add: UNION-eq) apply (rule equalityI)
apply simp
apply (rule conjI) apply (rule subsetI) apply simp apply (erule bexE) apply
(drule replace5)
apply (case-tac xa ∈  $\mathcal{F}$  g) apply simp
apply (subgoal-tac x ∈ edges g) apply simp
apply (simp add: edges-graph-def) apply force
apply simp
apply (subgoal-tac pre-split-face f ram1 ram2 vs)
apply (case-tac between (vertices f) ram2 ram1 = [])
apply (frule split-face-edges-f21-bet) apply (rule split) apply simp apply
simp
apply (case-tac between (vertices f) ram1 ram2 = [])
apply (frule split-face-edges-f-vs1-vs2) apply simp apply simp apply simp
apply force
apply (frule split-face-edges-f-vs2) apply simp apply simp apply simp
apply force
apply (frule split-face-edges-f21) apply (rule split) apply simp apply simp
apply simp
apply (case-tac between (vertices f) ram1 ram2 = [])
apply (frule split-face-edges-f-vs1) apply simp apply simp apply simp
apply simp apply force
apply (frule split-face-edges-f) apply simp apply simp apply simp apply
simp apply force
apply simp
apply (subgoal-tac pre-split-face f ram1 ram2 vs)
apply (case-tac between (vertices f) ram1 ram2 = [])
apply (frule split-face-edges-f12-bet) apply (rule split) apply simp apply
simp
apply (case-tac between (vertices f) ram2 ram1 = [])
apply (frule split-face-edges-f-vs1-vs2) apply simp apply simp apply simp
apply force
apply (frule split-face-edges-f-vs1) apply simp apply simp apply simp
apply force
apply (frule split-face-edges-f12) apply (rule split) apply simp apply simp
apply simp
apply (case-tac between (vertices f) ram2 ram1 = [])
apply (frule split-face-edges-f-vs2) apply simp apply simp apply simp
```

apply simp apply force
apply (frule split-face-edges-f) apply simp apply simp apply simp apply
simp apply force
apply simp
apply simp
apply (subgoal-tac pre-split-face f ram1 ram2 vs)
apply (subgoal-tac (ram2, last vs) ∈ edges f12 ∧ (hd vs, ram1) ∈ edges f12)
apply (rule conjI) apply simp
apply (rule conjI) apply simp
apply (subgoal-tac (ram1, hd vs) ∈ edges f21 ∧ (last vs, ram2) ∈ edges f21)
apply (rule conjI) apply (rule disjI1) apply (rule bexI) apply (elim conjE)
apply simp
apply (rule replace3) apply(erule pre-splitFace-oldF) apply simp
apply (rule conjI) apply (rule disjI1) apply (rule bexI) apply (elim conjE)
apply simp
apply (rule replace3) apply(erule pre-splitFace-oldF)
apply simp
apply (subgoal-tac edges f ⊆ {y. ∃ x∈set (replace f [f21] (faces g)). y ∈ edges
x} ∪ edges f12)
apply (subgoal-tac edges g ⊆ {y. ∃ x∈set (replace f [f21] (faces g)). y ∈ edges
x} ∪ edges f12)
apply (rule conjI) apply simp
apply (rule conjI) apply simp
apply (subgoal-tac Edges(rev vs) ⊆ edges f12) apply (rule conjI) prefer 2
apply blast
apply (subgoal-tac Edges vs ⊆ edges f21)
apply (subgoal-tac Edges vs ⊆ {y. ∃ x∈set (replace f [f21] (faces g)). y ∈
edges x}) apply blast
apply (rule subset-trans) apply assumption apply (rule subsetI) apply
simp apply (rule bexI) apply simp
apply (rule replace3) apply(erule pre-splitFace-oldF) apply simp

apply (frule split-face-edges-f21-subset) apply (rule split) apply simp apply
simp
apply (frule split-face-edges-f12-subset) apply (rule split) apply simp apply
simp
apply (simp add: edges-graph-def) apply (rule subsetI) apply simp apply
(elim bexE)
apply (case-tac xa = f) apply simp apply blast
apply (rule disjI1) apply (rule bexI) apply simp apply (rule replace4)
apply simp apply force
apply (rule subsetI)
apply (subgoal-tac ∃ u v. x = (u,v)) apply (elim exE conjE)
apply (frule split-face-edges-or [OF split]) apply simp
apply (case-tac (u, v) ∈ edges f12) apply simp apply simp
apply (rule bexI) apply (thin-tac (u, v) ∈ edges f) apply assumption
apply (rule replace3) apply(erule pre-splitFace-oldF) apply simp apply
simp
apply (frule split-face-edges-f21-subset) apply (rule split) apply simp apply

simp
apply (*frule split-face-edges-f12-subset*) **apply** (*rule split*) **apply simp apply**
simp
by simp
qed

lemma *pre-splitFace-edges-f-in-g*: *pre-splitFace g ram1 ram2 f vs* \implies *edges f* \subseteq *edges g*
apply (*simp add: edges-graph-def*) **by** (*force*)

lemma *pre-splitFace-edges-f-in-g2*: *pre-splitFace g ram1 ram2 f vs* \implies $x \in$ *edges f* \implies $x \in$ *edges g*
apply (*simp add: edges-graph-def*) **by** (*force*)

lemma *splitFace-edges-g'*: *pre-splitFace g ram1 ram2 f vs* \implies
(f12, f21, g') = *splitFace g ram1 ram2 f vs* \implies *vs* \neq [] \implies
edges g' = *edges g* \cup *Edges vs* \cup *Edges (rev vs)* \cup
{(*ram2*, *last vs*), (*hd vs*, *ram1*), (*ram1*, *hd vs*), (*last vs*, *ram2*)}
apply (*subgoal-tac edges g* \cup *edges f* = *edges g*)
apply (*frule splitFace-edges-g'-help*) **apply simp apply simp apply simp**
apply (*frule pre-splitFace-edges-f-in-g*) **by** *blast*

lemma *splitFace-edges-g'-vs*: *pre-splitFace g ram1 ram2 f []* \implies
(f12, f21, g') = *splitFace g ram1 ram2 f []* \implies
edges g' = *edges g* \cup {(*ram1*, *ram2*), (*ram2*, *ram1*)}

proof –

assume *pre*: *pre-splitFace g ram1 ram2 f []*
and *fdg*: (*f12, f21, g'*) = *splitFace g ram1 ram2 f []*

from *pre fdg* **have** *split*: (*f12, f21*) = *split-face f ram1 ram2 []*
apply (*unfold pre-splitFace-def*) **apply** (*elim conjE*)
by (*simp add: splitFace-split-face*)

from *fdg pre* **have** *edges g'* = ($\bigcup_{a \in \text{set } (replace f [f21] (faces g))}$ *edges a*) \cup
edges (f12) **by** (*auto simp: splitFace-def split-def edges-graph-def*)

with *pre* **show** *?thesis* **apply** (*simp add: UNION-eq*) **apply** (*rule equalityI*)
apply simp

apply (*rule conjI*) **apply** (*rule subsetI*) **apply simp apply** (*erule bexE*) **apply**
(*drule replace5*)

apply (*case-tac xa* \in \mathcal{F} *g*) **apply simp**
apply (*subgoal-tac x* \in *edges g*) **apply simp**
apply (*simp add: edges-graph-def*) **apply force**
apply simp
apply (*subgoal-tac pre-split-face f ram1 ram2 []*)

apply (*case-tac between (vertices f) ram2 ram1 = []*) **apply** (*simp add:*
pre-FaceDiv-between2)

apply (*frule split-face-edges-f21-vs*) **apply** (*rule split*) **apply simp apply**
simp apply simp

apply (*case-tac* $x = (ram1, ram2)$) **apply simp apply simp apply** (*rule disjI2*)
apply (*rule pre-splitFace-edges-f-in-g2*) **apply simp**
apply (*subgoal-tac pre-split-face f ram1 ram2 []*)
apply (*frule split-face-edges-f*) **apply simp apply simp apply** (*rule pre-FaceDiv-between1*)
apply simp apply simp
apply simp apply force apply simp apply simp

apply (*rule subsetI*) **apply simp**
apply (*subgoal-tac pre-split-face f ram1 ram2 []*)
apply (*case-tac between (vertices f) ram1 ram2 = []*) **apply** (*simp add: pre-FaceDiv-between1*)
apply (*frule split-face-edges-f12-vs*) **apply** (*rule split*) **apply simp apply simp apply simp**
apply (*case-tac x = (ram2, ram1)*) **apply simp apply simp apply** (*rule disjI2*)
apply (*rule pre-splitFace-edges-f-in-g2*) **apply simp**
apply (*subgoal-tac pre-split-face f ram1 ram2 []*)
apply (*frule split-face-edges-f*) **apply simp apply simp apply simp apply** (*rule pre-FaceDiv-between2*) **apply simp**
apply simp apply force apply simp apply simp
apply simp
apply (*subgoal-tac pre-split-face f ram1 ram2 []*)
apply (*subgoal-tac (ram1, ram2) ∈ edges f21*)
apply (*rule conjI*) **apply** (*rule disjI1*) **apply** (*rule bexI*) **apply simp apply** (*force*)
apply (*subgoal-tac (ram2, ram1) ∈ edges f12*)
apply (*rule conjI*) **apply force**
apply (*rule subsetI*) **apply** (*simp add: edges-graph-def*) **apply** (*elim bexE*)
apply (*case-tac xa = f*) **apply simp**
apply (*subgoal-tac ∃ u v. x = (u,v)*) **apply** (*elim exE conjE*)
apply (*subgoal-tac pre-split-face f ram1 ram2 []*)
apply (*frule split-face-edges-or [OF split]*) **apply simp**
apply (*case-tac (u, v) ∈ edges f12*) **apply simp apply simp apply force apply simp apply simp**
apply (*rule disjI1*) **apply** (*rule bexI*) **apply simp apply** (*rule replace4*) **apply simp apply force**
apply (*frule split-face-edges-f12-vs*) **apply simp apply** (*rule split*) **apply simp**
apply (*rule pre-FaceDiv-between1*) **apply simp apply simp**
apply (*frule split-face-edges-f21-vs*) **apply simp apply** (*rule split*) **apply simp**
apply (*rule pre-FaceDiv-between2*) **apply simp apply simp**
by simp
qed

lemma *splitFace-edges-incr*:
pre-splitFace g ram1 ram2 f vs \implies
 $(f_1, f_2, g') = \text{splitFace } g \text{ ram1 ram2 } f \text{ vs} \implies$
 $\text{edges } g \subseteq \text{edges } g'$

```

apply(cases vs)
  apply(simp add:splitFace-edges-g'-vs)
  apply blast
apply(simp add:splitFace-edges-g')
apply blast
done

```

```

lemma snd-snd-splitFace-edges-incr:
  pre-splitFace g v1 v2 f vs  $\implies$ 
  edges g  $\subseteq$  edges(snd(snd(splitFace g v1 v2 f vs)))
apply(erule splitFace-edges-incr
  [where f1 = fst(splitFace g v1 v2 f vs)
  and f2 = fst(snd(splitFace g v1 v2 f vs))])
apply(auto)
done

```

12.11 removeNones

definition removeNones :: 'a option list \Rightarrow 'a list **where**
 removeNones vOptionList \equiv [the x. x \leftarrow vOptionList, x \neq None]

```

declare removeNones-def [simp]
lemma removeNones-inI[intro]: Some a  $\in$  set ls  $\implies$  a  $\in$  set (removeNones ls) by
  (induct ls) auto
lemma removeNones-hd[simp]: removeNones (Some a # ls) = a # removeNones
  ls by auto
lemma removeNones-last[simp]: removeNones (ls @ [Some a]) = removeNones ls
  @ [a] by auto
lemma removeNones-in[simp]: removeNones (as @ Some a # bs) = removeNones
  as @ a # removeNones bs by auto
lemma removeNones-none-hd[simp]: removeNones (None # ls) = removeNones
  ls by auto
lemma removeNones-none-last[simp]: removeNones (ls @ [None]) = removeNones
  ls by auto
lemma removeNones-none-in[simp]: removeNones (as @ None # bs) = removeNones
  (as @ bs) by auto
lemma removeNones-empty[simp]: removeNones [] = [] by auto
declare removeNones-def [simp del]

```

12.12 natToVertexList

```

primrec natToVertexListRec ::
  nat  $\Rightarrow$  vertex  $\Rightarrow$  face  $\Rightarrow$  nat list  $\Rightarrow$  vertex option list
where
  natToVertexListRec old v f [] = [] |
  natToVertexListRec old v f (i#is) =
    (if i = old then None#natToVertexListRec i v f is
     else Some (fi · v))

```

natToVertexListRec i v f is)

primrec *natToVertexList* ::

vertex \Rightarrow *face* \Rightarrow *nat list* \Rightarrow *vertex option list*

where

natToVertexList *v f* [] = [] |

natToVertexList *v f* (*i#is*) =

(if *i* = 0 then (Some *v*)#(*natToVertexListRec i v f is*) else [])

12.13 *indexToVertexList*

lemma *nextVertex-inj*:

distinct (vertices f) \Longrightarrow $v \in \mathcal{V} f \Longrightarrow$

$i < \text{length} (\text{vertices } (f::\text{face})) \Longrightarrow a < \text{length} (\text{vertices } f) \Longrightarrow$

$f^a \cdot v = f^i \cdot v \Longrightarrow i = a$

proof –

assume *d*: *distinct (vertices f)* **and** *v*: $v \in \mathcal{V} f$ **and** *i*: $i < \text{length} (\text{vertices } (f::\text{face}))$

and *a*: $a < \text{length} (\text{vertices } f)$ **and** *eq*: $f^a \cdot v = f^i \cdot v$

then have *eq*: $(\text{verticesFrom } f v) ! a = (\text{verticesFrom } f v) ! i$ **by** (*simp add: verticesFrom-nth*)

define *xs* **where** $xs = \text{verticesFrom } f v$

with *eq* **have** *eq*: $xs ! a = xs ! i$ **by** *auto*

from *d v* **have** *z*: *distinct (verticesFrom f v)* **by** *auto*

moreover

from *xs-def a v d* **have** $(\text{verticesFrom } f v) = \text{take } a \text{ } xs @ xs ! a \# \text{drop } (\text{Suc } a) \text{ } xs$

by (*auto intro: id-take-nth-drop simp: verticesFrom-length*)

with *eq* **have** $(\text{verticesFrom } f v) = \text{take } a \text{ } xs @ xs ! i \# \text{drop } (\text{Suc } a) \text{ } xs$ **by** *simp*

moreover

from *xs-def i v d* **have** $(\text{verticesFrom } f v) = \text{take } i \text{ } xs @ xs ! i \# \text{drop } (\text{Suc } i) \text{ } xs$

by (*auto intro: id-take-nth-drop simp: verticesFrom-length*)

ultimately have $\text{take } a \text{ } xs = \text{take } i \text{ } xs$ **by** (*rule dist-at1*)

moreover

from *v d* **have** *vertFrom[simp]*: $\text{length} (\text{vertices } f) = \text{length} (\text{verticesFrom } f v)$

by (*auto simp: verticesFrom-length*)

from *xs-def a i* **have** $a < \text{length } xs \ i < \text{length } xs$ **by** *auto*

moreover

have $\bigwedge a \ i. a < \text{length } xs \Longrightarrow i < \text{length } xs \Longrightarrow \text{take } a \text{ } xs = \text{take } i \text{ } xs \Longrightarrow a = i$

proof (*induct xs*)

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

next

case (*Cons x xs*) **then show** *?case*

by (*auto simp: less-Suc-eq-0-disj*)

qed

ultimately show *?thesis* **by** *simp*

qed

lemma *a*: *distinct (vertices f)* $\Longrightarrow v \in \mathcal{V} f \Longrightarrow (\forall i \in \text{set } is. i < \text{length} (\text{vertices$

$f)) \implies$
 $(\bigwedge a. a < \text{length } (\text{vertices } f) \implies \text{hideDupsRec } ((f \cdot \sim a) v) [(f \cdot \sim k) v. k \leftarrow is] = \text{natToVertexListRec } a v f is)$
proof *(induct is)*
case Nil then show ?case by simp
next
case (Cons i is) then show ?case
by (auto simp: nextVertices-def intro: nextVertex-inj)
qed

lemma indexToVertexList-natToVertexList-eq: distinct (vertices f) $\implies v \in \mathcal{V} f \implies$
 $(\forall i \in \text{set } is. i < \text{length } (\text{vertices } f)) \implies is \neq [] \implies$
 $hd is = 0 \implies \text{indexToVertexList } f v is = \text{natToVertexList } v f is$
apply (cases is) by (auto simp: a [where a = 0, simplified] indexToVertexList-def nextVertices-def)

lemma nvlr-length: $\bigwedge old. (\text{length } (\text{natToVertexListRec } old v f ls)) = \text{length } ls$
apply (induct ls) by auto

lemma nvl-length[simp]: $hd e = 0 \implies \text{length } (\text{natToVertexList } v f e) = \text{length } e$
apply (cases e)
by (auto intro: nvlr-length)

lemma natToVertexListRec-length[simp]: $\bigwedge e f. \text{length } (\text{natToVertexListRec } e v f es) = \text{length } es$
by (induct es) auto

lemma natToVertexList-length[simp]: $\text{incrIndexList } es (\text{length } es) (\text{length } (\text{vertices } f)) \implies$
 $\text{length } (\text{natToVertexList } v f es) = \text{length } es$ **apply (case-tac es) by simp-all**

lemma natToVertexList-nth-Suc: $\text{incrIndexList } es (\text{length } es) (\text{length } (\text{vertices } f)) \implies \text{Suc } n < \text{length } es \implies$
 $(\text{natToVertexList } v f es)!(\text{Suc } n) = (\text{if } (es!n = es!(\text{Suc } n)) \text{ then None else Some } (f(es!\text{Suc } n) \cdot v))$
proof –
assume incr: $\text{incrIndexList } es (\text{length } es) (\text{length } (\text{vertices } f))$ and $n: \text{Suc } n < \text{length } es$
have rec: $\bigwedge old n. \text{Suc } n < \text{length } es \implies$

```

    (natToVertexListRec old v f es)!(Suc n) = (if (es!n = es!(Suc n)) then None
else Some (f(es!Suc n) . v))
  proof (induct es)
    case Nil then show ?case by auto
  next
    case (Cons e es)
    note cons1 = this
    then show ?case
    proof (cases es)
      case Nil with cons1 show ?thesis by simp
    next
      case (Cons e' es')
      with cons1 show ?thesis
      proof (cases n)
        case 0 with Cons cons1 show ?thesis by simp
      next
        case (Suc m) with Cons cons1
          have  $\wedge$  old. natToVertexListRec old v f es ! Suc m = (if es ! m = es ! Suc
m then None else Some (f(es ! Suc m) . v))
          by (rule-tac cons1) auto
          then show ?thesis apply (cases e = old) by (simp-all add: Suc)
        qed
      qed
    qed
  with n have natToVertexListRec 0 v f es ! Suc n = (if es ! n = es ! Suc n then
None else Some (f(es ! Suc n) . v)) by (rule-tac rec) auto
  with incr show ?thesis by (cases es) auto
qed

```

```

lemma natToVertexList-nth-0: incrIndexList es (length es) (length (vertices f))
 $\implies 0 < \text{length } es \implies$ 
(natToVertexList v f es)!0 = Some (f(es!0) . v)
  apply (cases es)
  apply (simp-all add: nextVertices-def)
  by (subgoal-tac a = 0) auto

```

```

lemma natToVertexList-hd[simp]:
  incrIndexList es (length es) (length (vertices f))  $\implies$  hd (natToVertexList v f es)
= Some v
  apply (cases es) by (simp-all add: nextVertices-def)

```

```

lemma nth-last[intro]: Suc i = length xs  $\implies$  xs!i = last xs
by (cases xs rule: rev-exhaust) auto

```

```

declare incrIndexList-help4 [simp del]

```

```

lemma natToVertexList-last[simp]:
  distinct (vertices f)  $\implies v \in \mathcal{V} f \implies$  incrIndexList es (length es) (length (vertices

```

$f)) \implies \text{last } (\text{natToVertexList } v \text{ } f \text{ } es) = \text{Some } (\text{last } (\text{verticesFrom } f \text{ } v))$
proof –
assume vors : $\text{distinct } (\text{vertices } f) \text{ } v \in \mathcal{V} \text{ } f$ **and** incr : $\text{incrIndexList } es \text{ } (\text{length } es)$
 $(\text{length } (\text{vertices } f))$
define n' **where** $n' = \text{length } es - 2$
from incr **have** $1 < \text{length } es$ **by** auto
with n' - def **have** $n'l$: $\text{Suc } (\text{Suc } n') = \text{length } es$ **by** arith
from $\text{incr } n'l$ **have** last-ntvl : $(\text{natToVertexList } v \text{ } f \text{ } es)!(\text{Suc } n') = \text{last } (\text{natToVertexList } v \text{ } f \text{ } es)$ **by** auto
from $n'l$ **have** last-es : $es!(\text{Suc } n') = \text{last } es$ **by** auto
from $n'l$ **have** $es!n'$ = $\text{last } (\text{butlast } es)$ **apply** $(\text{cases } es \text{ rule: rev-exhaust})$ **by**
 $(\text{auto simp: nth-append})$
with $\text{last-es } \text{incr}$ **have** less : $es!n' < es!(\text{Suc } n')$ **by** auto
from $n'l$ **have** $\text{Suc } n' < \text{length } es$ **by** arith
with $\text{incr } \text{less}$ **have** $(\text{natToVertexList } v \text{ } f \text{ } es)!(\text{Suc } n') = (\text{Some } (f^{(es! \text{Suc } n')} \cdot v))$
by $(\text{auto dest: natToVertexList-nth-Suc})$
with $\text{incr } \text{last-ntvl } \text{last-es}$ **have** rule1 : $\text{last } (\text{natToVertexList } v \text{ } f \text{ } es) = \text{Some } (f^{((\text{length } (\text{vertices } f)) - (\text{Suc } 0))} \cdot v)$ **by** auto

from incr **have** lf : $1 < \text{length } (\text{vertices } f)$ **by** auto
with vors **have** rule2 : $\text{verticesFrom } f \text{ } v ! ((\text{length } (\text{vertices } f)) - (\text{Suc } 0)) = f^{((\text{length } (\text{vertices } f)) - (\text{Suc } 0))} \cdot v$ **by** $(\text{auto intro!: verticesFrom-nth})$
from $\text{vors } \text{lf}$ **have** $\text{verticesFrom } f \text{ } v ! ((\text{length } (\text{vertices } f)) - (\text{Suc } 0)) = \text{last } (\text{verticesFrom } f \text{ } v)$
apply $(\text{rule-tac nth-last})$
by $(\text{auto simp: verticesFrom-length})$
with $\text{rule1 } \text{rule2}$ **show** $?thesis$ **by** auto
qed

lemma $\text{indexToVertexList-last[simp]}$:
 $\text{distinct } (\text{vertices } f) \implies v \in \mathcal{V} \text{ } f \implies \text{incrIndexList } es \text{ } (\text{length } es) \text{ } (\text{length } (\text{vertices } f)) \implies \text{last } (\text{indexToVertexList } f \text{ } v \text{ } es) = \text{Some } (\text{last } (\text{verticesFrom } f \text{ } v))$
apply $(\text{subgoal-tac indexToVertexList } f \text{ } v \text{ } es = \text{natToVertexList } v \text{ } f \text{ } es)$ **apply** simp
apply $(\text{rule indexToVertexList-natToVertexList-eq})$ **by** auto

lemma nths-take : $\bigwedge n \text{ iset. } \forall i \in \text{iset. } i < n \implies \text{nths } (\text{take } n \text{ } xs) \text{ } \text{iset} = \text{nths } xs \text{ } \text{iset}$
proof $(\text{induct } xs)$
case Nil **then show** $?case$ **by** simp
next
case $(\text{Cons } x \text{ } xs)$ **then show** $?case$ **apply** $(\text{simp add: nths-Cons})$ **apply** $(\text{cases } n)$ **apply** simp **apply** $(\text{simp add: nths-Cons})$ **apply** (rule Cons) **by** auto
qed

lemma $\text{nths-reduceIndices}$: $\bigwedge \text{iset. } \text{nths } xs \text{ } \text{iset} = \text{nths } xs \text{ } \{i. i < \text{length } xs \wedge i \in \text{iset}\}$
proof $(\text{induct } xs)$

```

case Nil then show ?case by simp
next
  case (Cons x xs) then
    have nths xs {j. Suc j ∈ iset} = nths xs {i. i < length xs ∧ i ∈ {j. Suc j ∈ iset}}
by (rule-tac Cons)
    then show ?case by (simp add: nths-Cons)
qed

lemma natToVertexList-nths1: distinct (vertices f) ⇒
  v ∈ V f ⇒ vs = verticesFrom f v ⇒
  incrIndexList es (length es) (length vs) ⇒ n ≤ length es ⇒
  nths (take (Suc (es!(n - 1))) vs) (set (take n es))
  = removeNones (take n (natToVertexList v f es))
proof (induct n)
  case 0 then show ?case by simp
next
  case (Suc n)
    then have nths (take (Suc (es ! (n - Suc 0))) (verticesFrom f v)) (set (take n
  es)) = removeNones (take n (natToVertexList v f es))
      distinct (vertices f) v ∈ V f vs = verticesFrom f v incrIndexList es (length es)
  (length (verticesFrom f v)) Suc n ≤ length es by auto
    note suc1 = this
    then have lvs: length vs = length (vertices f) by (auto intro: verticesFrom-length)
    with suc1 have vsne: vs ≠ [] by auto
    with suc1 show ?case
    proof (cases natToVertexList v f es ! n)
      case None then show ?thesis
      proof (cases n)
        case 0 with None suc1 lvs show ?thesis by (simp add: take-Suc-conv-app-nth
  natToVertexList-nth-0)
      next
        case (Suc n')
          with None suc1 lvs have esn: es!n = es!n' by (simp add: natToVer-
  texList-nth-Suc split: if-split-asm)
          from Suc have n': n - Suc 0 = n' by auto
          show ?thesis
          proof (cases Suc n = length es)
            case True then
              have small-n: n < length es by auto
              from True have take (Suc n) es = es by auto
              with small-n have take n es @ [es!n] = es by (simp add: take-Suc-conv-app-nth)
              then have esn-simps: take n es = butlast es ∧ es!n = last es by (cases es
  rule: rev-exhaust) auto

              from True Suc have n'l: Suc n' = length (butlast es) by auto
              then have small-n': n' < length (butlast es) by auto

              from Suc small-n have take-n': take (Suc n') (butlast es @ [last es]) = take
  (Suc n') (butlast es) by auto

```

```

from small-n have es-exh:  $es = \text{butlast } es @ [\text{last } es]$  by (cases es rule:
rev-exhaust) auto

from n'l have  $\text{take } (Suc \ n') \ (\text{butlast } es @ [\text{last } es]) = \text{butlast } es$  by auto
with es-exh have  $\text{take } (Suc \ n') \ es = \text{butlast } es$  by auto
with small-n Suc have  $\text{take } n' \ es @ [es!n'] = (\text{butlast } es)$  by (simp add:
take-Suc-conv-app-nth)
with small-n' have esn'-simps:  $\text{take } n' \ es = \text{butlast } (\text{butlast } es) \wedge es!n' =$ 
 $\text{last } (\text{butlast } es)$ 
by (cases butlast es rule: rev-exhaust) auto

from suc1 have  $\text{last } (\text{butlast } es) < \text{last } es$  by auto
with esn esn-simps esn'-simps have False by auto
then show ?thesis by auto
next
case False with suc1 have le:  $Suc \ n < \text{length } es$  by auto
from suc1 le have  $es = \text{take } (Suc \ n) \ es @ es!(Suc \ n) \ \# \ \text{drop } (Suc \ (Suc \ n))$ 
 $es$  by (auto intro: id-take-nth-drop)
with suc1 have increasing ( $\text{take } (Suc \ n) \ es @ es!(Suc \ n) \ \# \ \text{drop } (Suc \ (Suc \ n)) \ es$ )
by auto
then have  $\forall i \in (\text{set } (\text{take } (Suc \ n) \ es)). i \leq es! \ (Suc \ n)$  by (auto intro:
increasing2)
with suc1 have  $\forall i \in (\text{set } (\text{take } n \ es)). i \leq es! \ (Suc \ n)$  by (simp add:
take-Suc-conv-app-nth)
then have seq:  $nths \ (\text{take } (Suc \ (es! \ Suc \ n)) \ (\text{verticesFrom } f \ v)) \ (\text{set } (\text{take} \ n \ es))$ 
 $= nths \ (\text{verticesFrom } f \ v) \ (\text{set } (\text{take } n \ es))$ 
apply (rule-tac nths-take) by auto
from suc1 have  $es = \text{take } n \ es @ es!n \ \# \ \text{drop } (Suc \ n) \ es$  by (auto intro:
id-take-nth-drop)
with suc1 have increasing ( $\text{take } n \ es @ es!n \ \# \ \text{drop } (Suc \ n) \ es$ ) by auto
then have  $\forall i \in (\text{set } (\text{take } n \ es)). i \leq es! \ n$  by (auto intro: increasing2)
with suc1 esn have  $\forall i \in (\text{set } (\text{take } n \ es)). i \leq es! \ n'$  by (simp add:
take-Suc-conv-app-nth)
with Suc have seq2:  $nths \ (\text{take } (Suc \ (es! \ n')) \ (\text{verticesFrom } f \ v)) \ (\text{set } (\text{take} \ n \ es))$ 
 $= nths \ (\text{verticesFrom } f \ v) \ (\text{set } (\text{take } n \ es))$ 
apply (rule-tac nths-take) by auto
from Suc suc1 have ( $\text{insert } (es! \ n') \ (\text{set } (\text{take } n \ es))) = \text{set } (\text{take } n \ es)$ )
apply auto by (simp add: take-Suc-conv-app-nth)
with esn None suc1 seq seq2 n' show ?thesis by (simp add: take-Suc-conv-app-nth)
qed
qed
next
case (Some v') then show ?thesis
proof (cases n)
case 0
from suc1 lvs have  $\text{verticesFrom } f \ v \neq []$  by auto

```

then have $verticesFrom\ f\ v = hd\ (verticesFrom\ f\ v) \# tl\ (verticesFrom\ f\ v)$
by *auto*
then have $verticesFrom\ f\ v = v \# tl\ (verticesFrom\ f\ v)$ **by** (*simp add: verticesFrom-hd*)
then obtain z **where** $verticesFrom\ f\ v = v \# z$ **by** *auto*
then have $sub: nths\ (verticesFrom\ f\ v)\ \{0\} = [v]$ **by** (*auto simp: nths-Cons*)
from $0\ suc1$ **have** $es!0 = 0$ **by** (*cases es auto*)
with $0\ Some\ suc1\ lvs\ sub\ vsne$ **show** *?thesis*
by (*simp add: take-Suc-conv-app-nth natToVertexList-nth-0 nextVertices-def take-Suc nths-Cons verticesFrom-hd del:verticesFrom-empty*)
next
case $(Suc\ n')$
with $Some\ suc1\ lvs$ **have** $esn: es!n \neq es!n'$ **by** (*simp add: natToVertexList-nth-Suc split: if-split-asm*)
from $suc1\ Suc$ **have** $Suc\ n' < length\ es$ **by** *auto*
with $suc1\ lvs\ esn$ **have** $natToVertexList\ v\ f\ es!\ (Suc\ n') = Some\ (f(es!(Suc\ n')) \cdot v)$
apply (*simp add: natToVertexList-nth-Suc*)
by (*simp add: Suc*)
with Suc **have** $natToVertexList\ v\ f\ es!\ n = Some\ (f(es!n) \cdot v)$ **by** *auto*
with $Some$ **have** $v': v' = f(es!n) \cdot v$ **by** *simp*
from Suc **have** $n': n - Suc\ 0 = n'$ **by** *auto*
from $suc1\ Suc$ **have** $es = take\ (Suc\ n')\ es\ @\ es!n \# drop\ (Suc\ n)\ es$ **by** (*auto intro: id-take-nth-drop*)
with $suc1$ **have** *increasing* $(take\ (Suc\ n')\ es\ @\ es!n \# drop\ (Suc\ n)\ es)$ **by** *auto*
with $suc1\ Suc$ **have** $es!n' \leq es!n$ **apply** (*auto intro!: increasing2*)
by (*auto simp: take-Suc-conv-app-nth*)
with esn **have** *smaller-n*: $es!n' < es!n$ **by** *auto*
from $suc1\ lvs$ **have** *smaller*: $(es!n) < length\ vs$ **by** *auto*
from $suc1\ smaller\ lvs$ **have** $(verticesFrom\ f\ v)!(es!n) = f(es!n) \cdot v$ **by** (*auto intro: verticesFrom-nth*)
with v' **have** $(verticesFrom\ f\ v)!(es!n) = v'$ **by** *auto*
then have $sub1: nths\ [((verticesFrom\ f\ v)!(es!n))]\ \{j.\ j + (es!n) : (insert\ (es!\ n)\ (set\ (take\ n\ es)))\} = [v']$ **by** *auto*
from $suc1\ smaller\ lvs$ **have** $len: length\ (take\ (es!\ n)\ (verticesFrom\ f\ v)) = es!n$ **by** *auto*
have $\bigwedge x. x \in (set\ (take\ n\ es)) \implies x < (es!\ n)$
proof –
fix x
assume $x: x \in set\ (take\ n\ es)$
from $suc1\ Suc$ **have** $es = take\ n'\ es\ @\ es!n' \# drop\ (Suc\ n')\ es$ **by** (*auto intro: id-take-nth-drop*)
with $suc1$ **have** *increasing* $(take\ n'\ es\ @\ es!n' \# drop\ (Suc\ n')\ es)$ **by** *auto*
then have $\bigwedge x. x \in set\ (take\ n'\ es) \implies x \leq es!n'$ **by** (*auto intro!: increasing2*)

with x *Suc suc1* **have** $x \leq es!n'$ **by** (*auto simp: take-Suc-conv-app-nth*)
with *smaller-n* **show** $x < es!n$ **by** *auto*
qed
then **have** $\{i. i < es!n \wedge i \in \text{set } (take\ n\ es)\} = \text{set } (take\ n\ es)$ **by** *auto*
then **have** *elim-insert*: $\{i. i < es!n \wedge i \in \text{insert } (es!n) (\text{set } (take\ n\ es))\}$
 $= \text{set } (take\ n\ es)$ **by** *auto*

have *nths* ($take\ (es!n)\ (\text{verticesFrom}\ f\ v)$) ($\text{insert } (es!n)\ (\text{set } (take\ n\ es))$)
 $=$
nths ($take\ (es!n)\ (\text{verticesFrom}\ f\ v)$) $\{i. i < \text{length } (take\ (es!n)\ (\text{verticesFrom}\ f\ v))$
 $\wedge i \in (\text{insert } (es!n)\ (\text{set } (take\ n\ es)))\}$ **by** (*rule nths-reduceIndices*)
with *len* **have** *nths* ($take\ (es!n)\ (\text{verticesFrom}\ f\ v)$) ($\text{insert } (es!n)\ (\text{set } (take\ n\ es))$)
 $=$
nths ($take\ (es!n)\ (\text{verticesFrom}\ f\ v)$) $\{i. i < (es!n) \wedge i \in (\text{insert } (es!n)\ (\text{set } (take\ n\ es)))\}$
by *simp*
with *elim-insert* **have** *sub2*: *nths* ($take\ (es!n)\ (\text{verticesFrom}\ f\ v)$) ($\text{insert } (es!n)\ (\text{set } (take\ n\ es))$)
 $=$
nths ($take\ (es!n)\ (\text{verticesFrom}\ f\ v)$) ($\text{set } (take\ n\ es)$) **by** *simp*

define m **where** $m = es!n - es!n'$
with *smaller-n* **have** *mgz*: $0 < m$ **by** *auto*
with *m-def* **have** *esn*: $es!n = (es!n') + m$ **by** *auto*

have *helper*: $\bigwedge x. x \in \text{set } (take\ n\ es) \implies x \leq (es!n')$
proof –
fix x
assume $x \in \text{set } (take\ n\ es)$
from *suc1 Suc* **have** $es = take\ n'\ es @ es!n' \# drop\ (Suc\ n')\ es$ **by** (*auto intro: id-take-nth-drop*)
with *suc1* **have** *increasing* ($take\ n'\ es @ es!n' \# drop\ (Suc\ n')\ es$) **by** *auto*
then **have** $\bigwedge x. x \in \text{set } (take\ n'\ es) \implies x \leq es!n'$ **by** (*auto intro!: increasing2*)
with x *Suc suc1* **show** $x \leq es!n'$ **by** (*auto simp: take-Suc-conv-app-nth*)
qed

define m' **where** $m' = m - 1$
define *Suc-es-n'* **where** $Suc-es-n' = Suc\ (es!n')$

from *smaller smaller-n* **have** $Suc\ (es!n') < \text{length}\ vs$ **by** *auto*
then **have** $\text{min } (\text{length}\ vs)\ (Suc\ (es!n')) = Suc\ (es!n')$ **by** *arith*
with *Suc-es-n'-def* **have** *empty*: $\{j. j + \text{length } (take\ Suc-es-n'\ vs) \in \text{set } (take\ n\ es)\} = \{\}$
apply *auto* **apply** (*frule helper*) **by** *arith*

from *Suc-es-n'-def mgz esn m'-def* **have** *esn'*: $es!n = Suc-es-n' + m'$ **by** *auto*

with smaller have $(take (Suc-es-n' + m') vs) = take (Suc-es-n') vs @ take m' (drop (Suc-es-n') vs)$
by *(auto intro: take-add)*
with esn' have $nths (take (es ! n) vs) (set (take n es)) = nths (take (Suc-es-n') vs @ take m' (drop (Suc-es-n') vs)) (set (take n es))$ **by** *auto*
then have $nths (take (es ! n) vs) (set (take n es)) = nths (take (Suc-es-n') vs) (set (take n es)) @ nths (take m' (drop (Suc-es-n') vs)) \{j. j + length (take (Suc-es-n') vs) : (set (take n es))\}$
by *(simp add: nths-append)*
with empty Suc-es-n'-def have $nths (take (es ! n) vs) (set (take n es)) = nths (take (Suc (es!n')) vs) (set (take n es))$ **by** *simp*
with suc1 sub2 have $sub3: nths (take (es ! n) (verticesFrom f v)) (insert (es ! n) (set (take n es))) = nths (take (Suc (es!n')) (verticesFrom f v)) (set (take n es))$ **by** *simp*

from smaller suc1 have $take (Suc (es ! n)) (verticesFrom f v) = take (es ! n) (verticesFrom f v) @ [((verticesFrom f v)!(es!n))]$
by *(auto simp: take-Suc-conv-app-nth)*
with suc1 smaller have $nths (take (Suc (es ! n)) (verticesFrom f v)) (insert (es ! n) (set (take n es))) = nths (take (es ! n) (verticesFrom f v)) (insert (es ! n) (set (take n es))) @ nths [((verticesFrom f v)!(es!n))] \{j. j + (es!n) : (insert (es ! n) (set (take n es)))\}$
by *(auto simp: nths-append)*
with sub1 sub3 have $nths (take (Suc (es ! n)) (verticesFrom f v)) (insert (es ! n) (set (take n es))) = nths (take (Suc (es ! n')) (verticesFrom f v)) (set (take n es)) @ [v']$ **by** *auto*
with Some suc1 lvs n' show *?thesis* **by** *(simp add: take-Suc-conv-app-nth)*
qed
qed
qed

lemma *natToVertexList-nths: distinct (vertices f) $\implies v \in \mathcal{V} f \implies incrIndexList es (length es) (length (vertices f)) \implies nths (verticesFrom f v) (set es) = removeNones (natToVertexList v f es)$*

proof –

assume *vors1: distinct (vertices f) v $\in \mathcal{V} f$*

incrIndexList es (length es) (length (vertices f))

define *vs where vs = verticesFrom f v*

with vors1 have *lvs: length vs = length (vertices f)* **by** *(auto intro: verticesFrom-length)*

with vors1 vs-def have *vors: distinct (vertices f) v $\in \mathcal{V} f$*

vs = verticesFrom f v incrIndexList es (length es) (length vs) **by** *auto*

with lvs have *vsne: vs $\neq []$* **by** *auto*


```

define  $n$  where  $n = \text{length } es$ 
then have  $es!(n - 1) = \text{last } es$ 
proof (cases  $n$ )
  case 0 with  $n\text{-def } vors$  show ?thesis by (cases  $es$ ) auto
next
  case (Suc  $n'$ )
  with  $n\text{-def}$  have  $\text{small-}n'$ :  $n' < \text{length } es$  by arith
  from Suc  $n\text{-def}$  have take (Suc  $n'$ )  $es = es$  by auto
  with  $\text{small-}n'$  have take  $n'$   $es @ [es!n'] = es$  by (simp add: take-Suc-conv-app-nth)
  then have  $es!n' = \text{last } es$  by (cases  $es$  rule: rev-exhaust) auto
  with Suc show ?thesis by auto
qed
with vors have  $es!(n - 1) = (\text{length } vs) - 1$  by auto
with vsne have Suc ( $es!(n - 1)$ ) = ( $\text{length } vs$ ) by auto
then have take-vs: take (Suc ( $es!(n - 1)$ ))  $vs = vs$  by auto

from  $n\text{-def } vors$  have  $n = \text{length } (\text{natToVertexList } v f es)$  by auto
then have take-nTVL: take  $n$  ( $\text{natToVertexList } v f es$ ) =  $\text{natToVertexList } v f es$ 
by auto

from  $n\text{-def}$  have take-es: take  $n es = es$  by auto

from  $n\text{-def}$  have  $n \leq \text{length } es$  by auto
with vors have nth (take (Suc ( $es!(n - 1)$ ))  $vs$ ) (set (take  $n es$ ))
  = removeNones (take  $n$  ( $\text{natToVertexList } v f es$ )) by (rule natToVertexList-nths1)
with take-vs take-nTVL take-es vs-def show ?thesis by simp
qed

lemma filter-Cons2:
 $x \notin \text{set } ys \implies [y \leftarrow ys. y = x \vee P y] = [y \leftarrow ys. P y]$ 
by (induct  $ys$ ) (auto)

lemma natToVertexList-removeNones:
 $\text{distinct } (\text{vertices } f) \implies v \in \mathcal{V} f \implies$ 
 $\text{incrIndexList } es (\text{length } es) (\text{length } (\text{vertices } f)) \implies$ 
 $[x \leftarrow \text{verticesFrom } f v. x \in \text{set } (\text{removeNones } (\text{natToVertexList } v f es))]$ 
= removeNones ( $\text{natToVertexList } v f es$ )
proof -
assume vors:  $\text{distinct } (\text{vertices } f) v \in \mathcal{V} f$ 
   $\text{incrIndexList } es (\text{length } es) (\text{length } (\text{vertices } f))$ 
then have dist:  $\text{distinct } (\text{verticesFrom } f v)$  by auto
from vors have sub-eq:  $\text{nths } (\text{verticesFrom } f v) (\text{set } es)$ 
  = removeNones ( $\text{natToVertexList } v f es$ ) by (rule natToVertexList-nths)
from dist have  $[x \leftarrow \text{verticesFrom } f v.$ 
   $x \in \text{set } (\text{nths } (\text{verticesFrom } f v) (\text{set } es))] = \text{removeNones } (\text{natToVertexList } v f$ 
 $es)$ 
apply (simp add: filter-in-nths)
by (simp add: sub-eq)

```

with *sub-eq* **show** *?thesis* **by** *simp*
qed

definition *is-duplicateEdge* :: *graph* \Rightarrow *face* \Rightarrow *vertex* \Rightarrow *vertex* \Rightarrow *bool* **where**
is-duplicateEdge *g f a b* \equiv
 $((a, b) \in \text{edges } g \wedge (a, b) \notin \text{edges } f \wedge (b, a) \notin \text{edges } f)$
 $\vee ((b, a) \in \text{edges } g \wedge (b, a) \notin \text{edges } f \wedge (a, b) \notin \text{edges } f)$

definition *invalidVertexList* :: *graph* \Rightarrow *face* \Rightarrow *vertex option list* \Rightarrow *bool* **where**
invalidVertexList *g f vs* \equiv
 $\exists i < |vs| - 1.$
*case vs!**i* *of* *None* \Rightarrow *False*
| *Some a* \Rightarrow *case vs!**(i+1)* *of* *None* \Rightarrow *False*
| *Some b* \Rightarrow *is-duplicateEdge* *g f a b*

12.14 *pre-subdivFace*(')

definition *pre-subdivFace-face* :: *face* \Rightarrow *vertex* \Rightarrow *vertex option list* \Rightarrow *bool* **where**
pre-subdivFace-face *f v' vOptionList* \equiv
 $[v \leftarrow \text{verticesFrom } f \ v'. \ v \in \text{set } (\text{removeNones } vOptionList)]$
 $= (\text{removeNones } vOptionList)$
 $\wedge \neg \text{final } f \wedge \text{distinct } (\text{vertices } f)$
 $\wedge \text{hd } (vOptionList) = \text{Some } v'$
 $\wedge v' \in \mathcal{V} f$
 $\wedge \text{last } (vOptionList) = \text{Some } (\text{last } (\text{verticesFrom } f \ v'))$
 $\wedge \text{hd } (\text{tl } (vOptionList)) \neq \text{last } (vOptionList)$
 $\wedge 2 < |vOptionList|$
 $\wedge vOptionList \neq []$
 $\wedge \text{tl } (vOptionList) \neq []$

definition *pre-subdivFace* :: *graph* \Rightarrow *face* \Rightarrow *vertex* \Rightarrow *vertex option list* \Rightarrow *bool*
where
pre-subdivFace *g f v' vOptionList* \equiv
pre-subdivFace-face *f v' vOptionList* $\wedge \neg \text{invalidVertexList } g \ f \ vOptionList$

definition *pre-subdivFace'* :: *graph* \Rightarrow *face* \Rightarrow *vertex* \Rightarrow *vertex* \Rightarrow *nat* \Rightarrow *vertex option list* \Rightarrow *bool* **where**
pre-subdivFace' *g f v' ram1 n vOptionList* \equiv
 $\neg \text{final } f \wedge v' \in \mathcal{V} f \wedge \text{ram1} \in \mathcal{V} f$
 $\wedge v' \notin \text{set } (\text{removeNones } vOptionList)$
 $\wedge \text{distinct } (\text{vertices } f)$
 $\wedge ($
 $[v \leftarrow \text{verticesFrom } f \ v'. \ v \in \text{set } (\text{removeNones } vOptionList)]$
 $= (\text{removeNones } vOptionList)$
 $\wedge \text{before } (\text{verticesFrom } f \ v') \ \text{ram1} \ (\text{hd } (\text{removeNones } vOptionList))$
 $\wedge \text{last } (vOptionList) = \text{Some } (\text{last } (\text{verticesFrom } f \ v'))$
 $)$

$\wedge vOptionList \neq []$
 $\wedge ((v' = ram1 \wedge (0 < n)) \vee ((v' = ram1 \wedge (hd (vOptionList) \neq Some (last (verticesFrom f v')))) \vee (v' \neq ram1)))$
 $\wedge \neg invalidVertexList g f vOptionList$
 $\wedge (n = 0 \wedge hd (vOptionList) \neq None \longrightarrow \neg is-duplicateEdge g f ram1 (the (hd (vOptionList))))$
 $\vee (vOptionList = [] \wedge v' \neq ram1)$
 $)$

lemma *pre-subdivFace-face-in-f[intro]*: $pre-subdivFace-face f v ls \implies Some a \in set ls \implies a \in set (verticesFrom f v)$
apply (*subgoal-tac* $a \in set (removeNones ls)$) **apply** (*auto simp: pre-subdivFace-face-def*)
apply (*subgoal-tac* $a \in set [v \leftarrow verticesFrom f v . v \in set (removeNones ls)]$)
apply (*thin-tac* $[v \leftarrow verticesFrom f v . v \in set (removeNones ls)] = removeNones ls$) **by** *auto*

lemma *pre-subdivFace-in-f[intro]*: $pre-subdivFace g f v ls \implies Some a \in set ls \implies a \in set (verticesFrom f v)$
by (*auto simp: pre-subdivFace-def*)

lemma *pre-subdivFace-face-in-f'[intro]*: $pre-subdivFace-face f v ls \implies Some a \in set ls \implies a \in \mathcal{V} f$
apply (*cases* $a = v$) **apply** (*force simp: pre-subdivFace-face-def*)
apply (*rule verticesFrom-in'*) **apply** (*rule pre-subdivFace-face-in-f*)
by *auto*

lemma *filter-congs-shorten1*: $distinct (verticesFrom f v) \implies [v \leftarrow verticesFrom f v . v = a \vee v \in set vs] = (a \# vs)$
 $\implies [v \leftarrow verticesFrom f v . v \in set vs] = vs$

proof –

assume *dist*: $distinct (verticesFrom f v)$ **and** *eq*: $[v \leftarrow verticesFrom f v . v = a \vee v \in set vs] = (a \# vs)$

have *rule1*: $\bigwedge vs a ys. distinct vs \implies [v \leftarrow vs . v = a \vee v \in set ys] = a \# ys \implies [v \leftarrow vs. v \in set ys] = ys$

proof –

fix $vs a ys$

assume *dist*: $distinct vs$ **and** *ays*: $[v \leftarrow vs . v = a \vee v \in set ys] = a \# ys$

then have $distinct ([v \leftarrow vs . v = a \vee v \in set ys])$ **by** (*rule-tac distinct-filter*)

with *ays* **have** *distys*: $distinct (a \# ys)$ **by** *simp*

from *dist distys ays* **show** $[v \leftarrow vs. v \in set ys] = ys$

apply (*induct vs*) **by** (*auto split: if-split-asm simp: filter-Cons2*)

qed

from *dist eq* **show** *?thesis* **by** (*rule-tac rule1*)

qed

lemma *ovl-shorten*: $\text{distinct } (\text{verticesFrom } f \ v) \implies$
 $[\text{v} \leftarrow \text{verticesFrom } f \ v . v \in \text{set } (\text{removeNones } (va \ \# \ vol))] = (\text{removeNones } (va \ \# \ vol))$
 $\implies [\text{v} \leftarrow \text{verticesFrom } f \ v . v \in \text{set } (\text{removeNones } (vol))] = (\text{removeNones } (vol))$
proof –
assume *dist*: $\text{distinct } (\text{verticesFrom } f \ v)$
and *vors*: $[\text{v} \leftarrow \text{verticesFrom } f \ v . v \in \text{set } (\text{removeNones } (va \ \# \ vol))] = (\text{removeNones } (va \ \# \ vol))$
then show *?thesis*
proof (*cases va*)
case *None* **with** *vors Cons* **show** *?thesis* **by** *auto*
next
case (*Some a*) **with** *vors dist* **show** *?thesis* **by** (*auto intro!*: *filter-congs-shorten1*)
qed
qed

lemma *pre-subdivFace-face-distinct*: $\text{pre-subdivFace-face } f \ v \ vol \implies \text{distinct } (\text{removeNones } vol)$
apply (*auto dest!*: *verticesFrom-distinct simp*: *pre-subdivFace-face-def*)
apply (*subgoal-tac distinct* ($[\text{v} \leftarrow \text{verticesFrom } f \ v . v \in \text{set } (\text{removeNones } vol)]$))
apply *simp*
apply (*thin-tac* $[\text{v} \leftarrow \text{verticesFrom } f \ v . v \in \text{set } (\text{removeNones } vol)] = \text{removeNones } vol$) **by** *auto*

lemma *invalidVertexList-shorten*: $\text{invalidVertexList } g \ f \ vol \implies \text{invalidVertexList } g \ f \ (v \ \# \ vol)$
apply (*simp add*: *invalidVertexList-def*) **apply** *auto* **apply** (*rule exI*) **apply** *safe*
apply (*subgoal-tac* ($\text{Suc } i < | \ vol |$)) **apply** *assumption* **apply** *arith*
apply *auto* **apply** (*case-tac vol!*: *i*) **by** *auto*

lemma *pre-subdivFace-pre-subdivFace'*: $v \in \mathcal{V} \ f \implies \text{pre-subdivFace } g \ f \ v \ (vo \ \# \ vol) \implies$
 $\text{pre-subdivFace}' \ g \ f \ v \ v \ 0 \ (vol)$
proof –
assume *vors*: $v \in \mathcal{V} \ f \ \text{pre-subdivFace } g \ f \ v \ (vo \ \# \ vol)$
then have *vors'*: $v \in \mathcal{V} \ f \ \text{pre-subdivFace-face } f \ v \ (vo \ \# \ vol) \neg \text{invalidVertexList } g \ f \ (vo \ \# \ vol)$
by (*auto simp*: *pre-subdivFace-def*)
then have *r*: $\text{removeNones } vol \neq []$ **apply** (*cases vol rule*: *rev-exhaust*) **by** (*auto simp*: *pre-subdivFace-face-def*)
then have *Some* ($\text{hd } (\text{removeNones } vol) \in \text{set } vol$) **apply** (*induct vol*) **apply** *auto* **apply** (*case-tac a*) **by** *auto*
then have *Some* ($\text{hd } (\text{removeNones } vol) \in \text{set } (vo \ \# \ vol)$) **by** *auto*
with *vors'* **have** *hd*: $\text{hd } (\text{removeNones } vol) \in \mathcal{V} \ f$ **by** (*rule-tac pre-subdivFace-face-in-f'*)

from *vors'* **have** *Some* $v = vo$ **by** (*auto simp*: *pre-subdivFace-face-def*)
with *vors'* **have** $v \notin \text{set } (\text{tl } (\text{removeNones } (vo \ \# \ vol)))$ **apply** (*drule-tac pre-subdivFace-face-distinct*) **by** *auto*
with *vors'* *r* **have** *ne*: $v \neq \text{hd } (\text{removeNones } vol)$ **by** (*cases removeNones vol*)

(*auto simp: pre-subdivFace-face-def*)

from *vors'* **have** *dist: distinct (removeNones (vo # vol))* **apply** (*rule-tac pre-subdivFace-face-distinct*) .

from *vors'* **have** *invalid: \neg invalidVertexList g f vol* **by** (*auto simp: invalidVertexList-shorten*)

from *ne hd vors' invalid dist* **show** *?thesis* **apply** (*unfold pre-subdivFace'-def*)
apply (*simp add: pre-subdivFace'-def pre-subdivFace-face-def*)
apply *safe*
apply (*rule ovl-shorten*)
apply (*simp add: pre-subdivFace-face-def*) **apply** *assumption*
apply (*rule before-verticesFrom*)
apply *simp+*
apply (*simp add: invalidVertexList-def*)
apply (*erule allE*)
apply (*erule impE*)
apply (*subgoal-tac 0 < |vol|*)
apply (*thin-tac Suc 0 < |vol|*)
apply *assumption*
apply *simp*
apply (*simp*)
apply (*case-tac vol*) **apply** *simp* **by** (*simp add: is-duplicateEdge-def*)

qed

lemma *pre-subdivFace'-distinct: pre-subdivFace' g f v' v n vol \implies distinct (removeNones vol)*

apply (*unfold pre-subdivFace'-def*)
apply (*cases vol*) **apply** *simp+*
apply (*elim conjE*)
apply (*drule-tac verticesFrom-distinct*) **apply** *assumption*
apply (*subgoal-tac distinct [v \leftarrow verticesFrom f v' . v \in set (removeNones (a # list))]*) **apply** *force*
apply (*thin-tac [v \leftarrow verticesFrom f v' . v \in set (removeNones (a # list))] = removeNones (a # list)*)
by *auto*

lemma *natToVertexList-pre-subdivFace-face:*

\neg *final f \implies distinct (vertices f) \implies v \in \mathcal{V} f \implies 2 < |es| \implies incrIndexList es (length es) (length (vertices f)) \implies pre-subdivFace-face f v (natToVertexList v f es)*

proof –

assume *vors: \neg final f distinct (vertices f) v \in \mathcal{V} f 2 < |es|*
incrIndexList es (length es) (length (vertices f))

then have *lastOvl: last (natToVertexList v f es) = Some (last (verticesFrom f*

v) **by** *auto*

from *vors* **have** *nvl-l*: $2 < | \text{natToVertexList } v \text{ } f \text{ } es |$
by *auto*

from *vors* **have** *distinct* [$x \leftarrow \text{verticesFrom } f \text{ } v . x \in \text{set } (\text{removeNones } (\text{natToVertexList } v \text{ } f \text{ } es))$] **by** *auto*

with *vors* **have** *distinct* ($\text{removeNones } (\text{natToVertexList } v \text{ } f \text{ } es)$) **by** (*simp add*:
natToVertexList-removeNones)

with *nvl-l lastOvl* **have** *hd-last*: $\text{hd } (\text{tl } (\text{natToVertexList } v \text{ } f \text{ } es)) \neq \text{last } (\text{natToVertexList } v \text{ } f \text{ } es)$ **apply** *auto*

apply (*cases natToVertexList v f es*) **apply** *simp*

apply (*case-tac list rule: rev-exhaust*) **apply** *simp*

apply (*case-tac ys*) **apply** *simp*

apply (*case-tac a*) **apply** *simp by simp*

from *vors lastOvl hd-last nvl-l* **show** *?thesis*

apply (*auto intro: natToVertexList-removeNones simp: pre-subdivFace-face-def*)

apply (*cases es*) **apply** *auto*

apply (*cases es*) **apply** *auto*

apply (*subgoal-tac 0 < length list*) **apply** (*case-tac list*) **by** (*auto split: if-split-asm*)

qed

lemma *indexToVertexList-pre-subdivFace-face*:

$\neg \text{final } f \implies \text{distinct } (\text{vertices } f) \implies v \in \mathcal{V} f \implies 2 < |es| \implies$

$\text{incrIndexList } es \text{ } (\text{length } es) \text{ } (\text{length } (\text{vertices } f)) \implies$

$\text{pre-subdivFace-face } f \text{ } v \text{ } (\text{indexToVertexList } f \text{ } v \text{ } es)$

apply (*subgoal-tac indexToVertexList f v es = natToVertexList v f es*) **apply** *simp*

apply (*rule natToVertexList-pre-subdivFace-face*) **apply** *assumption+*

apply (*rule indexToVertexList-natToVertexList-eq*) **by** *auto*

lemma *subdivFace-subdivFace'-eq*: $\text{pre-subdivFace } g \text{ } f \text{ } v \text{ } vol \implies \text{subdivFace } g \text{ } f \text{ } vol$
 $= \text{subdivFace}' g \text{ } f \text{ } v \text{ } 0 \text{ } (\text{tl } vol)$

by (*simp-all add: subdivFace-def pre-subdivFace-def pre-subdivFace-face-def*)

lemma *pre-subdivFace'-None*:

$\text{pre-subdivFace}' g \text{ } f \text{ } v' \text{ } v \text{ } n \text{ } (\text{None} \# \text{vol}) \implies$

$\text{pre-subdivFace}' g \text{ } f \text{ } v' \text{ } v \text{ } (\text{Suc } n) \text{ } vol$

by(*auto simp: pre-subdivFace'-def dest:invalidVertexList-shorten*
split:if-split-asm)

declare *verticesFrom-between* [*simp del*]

lemma *verticesFrom-split*: $v \# \text{tl } (\text{verticesFrom } f \text{ } v) = \text{verticesFrom } f \text{ } v$ **by** (*auto*
simp: verticesFrom-Def)

lemma *verticesFrom-v*: $\text{distinct } (\text{vertices } f) \implies \text{vertices } f = a @ v \# b \implies \text{verticesFrom } f v = v \# b @ a$
by (*simp add: verticesFrom-Def*)

lemma *splitAt-fst[simp]*: $\text{distinct } xs \implies xs = a @ v \# b \implies \text{fst } (\text{splitAt } v xs) = a$
by *auto*

lemma *splitAt-snd[simp]*: $\text{distinct } xs \implies xs = a @ v \# b \implies \text{snd } (\text{splitAt } v xs) = b$
by *auto*

lemma *verticesFrom-splitAt-v-fst[simp]*:
 $\text{distinct } (\text{verticesFrom } f v) \implies \text{fst } (\text{splitAt } v (\text{verticesFrom } f v)) = []$
by (*simp add: verticesFrom-Def*)

lemma *verticesFrom-splitAt-v-snd[simp]*:
 $\text{distinct } (\text{verticesFrom } f v) \implies \text{snd } (\text{splitAt } v (\text{verticesFrom } f v)) = \text{tl } (\text{verticesFrom } f v)$
by (*simp add: verticesFrom-Def*)

lemma *filter-distinct-at*:
 $\text{distinct } xs \implies xs = (as @ u \# bs) \implies [v \leftarrow xs. v = u \vee P v] = u \# us \implies [v \leftarrow bs. P v] = us \wedge [v \leftarrow as. P v] = []$
apply (*subgoal-tac filter P as @ u \# filter P bs = [] @ u \# us*)
apply (*drule local-help'*) **by** (*auto simp: filter-Cons2*)

lemma *filter-distinct-at3*: $\text{distinct } xs \implies xs = (as @ u \# bs) \implies [v \leftarrow xs. v = u \vee P v] = u \# us \implies \forall z \in \text{set } as. z \in \text{set } as \vee \neg (P z) \implies [v \leftarrow zs @ bs. P v] = us$
apply (*drule filter-distinct-at*) **apply** *assumption+* **apply** *simp*
by (*induct zs*) *auto*

lemma *filter-distinct-at4*: $\text{distinct } xs \implies xs = (as @ u \# bs) \implies [v \leftarrow xs. v = u \vee v \in \text{set } us] = u \# us \implies \text{set } zs \cap \text{set } us \subseteq \{u\} \cup \text{set } as \implies [v \leftarrow zs @ bs. v \in \text{set } us] = us$

proof –

assume *vors*: $\text{distinct } xs \implies xs = (as @ u \# bs)$

$[v \leftarrow xs. v = u \vee v \in \text{set } us] = u \# us$

$\text{set } zs \cap \text{set } us \subseteq \{u\} \cup \text{set } as$

then have $\text{distinct } ([v \leftarrow xs. v = u \vee v \in \text{set } us])$ **apply** (*rule-tac distinct-filter*)

by *simp*

with *vors* **have** *dist*: $\text{distinct } (u \# us)$ **by** *auto*

with *vors* **show** *?thesis*

apply (*rule-tac filter-distinct-at3*) **by** *assumption+* *auto*

qed

lemma *filter-distinct-at5*: $distinct\ xs \implies xs = (as\ @\ u\ \#\ bs)$

$\implies [v \leftarrow xs. v = u \vee v \in set\ us] = u\ \#\ us$

$\implies set\ zs \cap set\ xs \subseteq \{u\} \cup set\ as$

$\implies [v \leftarrow zs@bs. v \in set\ us] = us$

proof –

assume *vors*: $distinct\ xs\ xs = (as\ @\ u\ \#\ bs)$

$[v \leftarrow xs. v = u \vee v \in set\ us] = u\ \#\ us$

$set\ zs \cap set\ xs \subseteq \{u\} \cup set\ as$

have $set\ ([v \leftarrow xs. v = u \vee v \in set\ us]) \subseteq set\ xs$ **by** *auto*

with *vors* **have** $set\ (u\ \#\ us) \subseteq set\ xs$ **by** *simp*

then **have** $set\ us \subseteq set\ xs$ **by** *simp*

with *vors* **have** $set\ zs \cap set\ us \subseteq set\ zs \cap insert\ u\ (set\ as \cup set\ bs)$ **by** *auto*

with *vors* **show** *?thesis* **apply** (*rule-tac filter-distinct-at4*) **apply** *assumption+*

by *auto*

qed

lemma *filter-distinct-at6*: $distinct\ xs \implies xs = (as\ @\ u\ \#\ bs)$

$\implies [v \leftarrow xs. v = u \vee v \in set\ us] = u\ \#\ us$

$\implies set\ zs \cap set\ xs \subseteq \{u\} \cup set\ as$

$\implies [v \leftarrow zs@bs. v \in set\ us] = us \wedge [v \leftarrow bs. v \in set\ us] = us$

proof –

assume *vors*: $distinct\ xs\ xs = (as\ @\ u\ \#\ bs)$

$[v \leftarrow xs. v = u \vee v \in set\ us] = u\ \#\ us$

$set\ zs \cap set\ xs \subseteq \{u\} \cup set\ as$

then **show** *?thesis* **apply** (*rule-tac conjI*) **apply** (*rule-tac filter-distinct-at5*)

apply *assumption+*

apply (*drule filter-distinct-at*) **apply** *assumption+* **by** *auto*

qed

lemma *filter-distinct-at-special*:

$distinct\ xs \implies xs = (as\ @\ u\ \#\ bs)$

$\implies [v \leftarrow xs. v = u \vee v \in set\ us] = u\ \#\ us$

$\implies set\ zs \cap set\ xs \subseteq \{u\} \cup set\ as$

$\implies us = hd-us\ \#\ tl-us$

$\implies [v \leftarrow zs@bs. v \in set\ us] = us \wedge hd-us \in set\ bs$

proof –

assume *vors*: $distinct\ xs\ xs = (as\ @\ u\ \#\ bs)$

$[v \leftarrow xs. v = u \vee v \in set\ us] = u\ \#\ us$

$set\ zs \cap set\ xs \subseteq \{u\} \cup set\ as$

$us = hd-us\ \#\ tl-us$

then **have** $[v \leftarrow zs@bs. v \in set\ us] = us \wedge [v \leftarrow bs. v \in set\ us] = us$

by (*rule-tac filter-distinct-at6*)

with *vors* **show** *?thesis* **apply** (*rule-tac conjI*) **apply** *safe* **apply** *simp*

apply (*subgoal-tac set (hd-us\ \#\ tl-us) \subseteq set\ bs*) **apply** *simp*

apply (*subgoal-tac set [v \leftarrow bs. v = hd-us \vee v \in set\ tl-us] \subseteq set\ bs*) **apply** *simp*

by (*rule-tac filter-is-subset*)

qed


```

lemma pre-subdivFace'-Some1':
assumes pre-add: pre-subdivFace' g f v' v n ((Some u) # vol)
  and pre-fdg: pre-splitFace g v u f ws
  and fdg: f21 = fst (snd (splitFace g v u f ws))
  and g': g' = snd (snd (splitFace g v u f ws))
shows pre-subdivFace' g' f21 v' u 0 vol
proof (cases vol = [])
  case True then show ?thesis using pre-add fdg pre-fdg
    apply (unfold pre-subdivFace'-def pre-splitFace-def)
    apply (simp add: splitFace-def split-face-def split-def del:distinct.simps)
    apply (rule conjI)
    apply (clarsimp)
    apply (rule before-between)
      apply (erule (5) rotate-before-vFrom)
    apply (erule not-sym)
    apply (clarsimp simp:between-distinct between-not-r1 between-not-r2)
    apply (blast dest:inbetween-inset)
    done
  next
    case False
    with pre-add
      have removeNones vol ≠ [] apply (cases vol rule: rev-exhaust) by (auto simp:
pre-subdivFace'-def)
      then have removeNones-split: removeNones vol = hd (removeNones vol) # tl
(removeNones vol) by auto

    from pre-add have dist: distinct (removeNones ((Some u) # vol)) by (rule-tac
pre-subdivFace'-distinct)

    from pre-add have v': v' ∈ V f by (auto simp: pre-subdivFace'-def)
    hence (vertices f) ≅ (verticesFrom f v') by (rule verticesFrom-congs)
    hence set-eq: set (verticesFrom f v') = V f
      apply (rule-tac sym) by (rule congs-pres-nodes)

    from pre-fdg fdg have dist-f21: distinct (vertices f21) by auto

    from pre-add have pre-bet': pre-between (verticesFrom f v') u v
      apply (simp add: pre-between-def pre-subdivFace'-def)
      apply (elim conjE) apply (thin-tac n = 0 → ¬ is-duplicateEdge g f v u)
      apply (thin-tac v' = v ∧ 0 < n ∨ v' = v ∧ u ≠ last (verticesFrom f v') ∨ v'
≠ v)
      apply (auto simp add: before-def)
      apply (subgoal-tac distinct (verticesFrom f v')) apply simp
      apply (rule-tac verticesFrom-distinct) by auto

```

with *pre-add* **have** *pre-bet*: *pre-between* (*vertices f*) *u v*
apply (*subgoal-tac* (*vertices f*) \cong (*verticesFrom f v'*))
apply (*simp add*: *pre-between-def pre-subdivFace'-def*)
by (*auto dest*: *congs-pres-nodes intro*: *verticesFrom-congs simp*: *pre-subdivFace'-def*)

from *pre-bet pre-add* **have** *bet-eq[simp]*: *between* (*vertices f*) *u v* = *between*
(*verticesFrom f v'*) *u v*
by (*auto intro*: *verticesFrom-between simp*: *pre-subdivFace'-def*)

from *fdg* **have** *f21-split-face*: *f21* = *snd* (*split-face f v u ws*)
by (*simp add*: *splitFace-def split-def*)
then **have** *f21*: *f21* = *Face* (*u # between* (*vertices f*) *u v @ v # ws*) *Nonfinal*
by (*simp add*: *split-face-def*)
with *pre-add pre-bet'*
have *vert-f21*: *vertices f21*
= *u # snd* (*splitAt u* (*verticesFrom f v'*)) @ *fst* (*splitAt v* (*verticesFrom f v'*)) @
v # ws
apply (*drule-tac pre-between-symI*)
by (*auto simp*: *pre-subdivFace'-def between-simp2 intro*: *pre-between-symI*)

moreover
from *pre-add* **have** *v* \in *set* (*verticesFrom f v'*) **by** (*auto simp*: *pre-subdivFace'-def*
before-def)
then **have** *verticesFrom f v'* =
fst (*splitAt v* (*verticesFrom f v'*)) @ *v # snd* (*splitAt v* (*verticesFrom f v'*))
by (*auto dest*: *splitAt-ram*)
then **have** *m*: *v' # tl* (*verticesFrom f v'*)
= *fst* (*splitAt v* (*verticesFrom f v'*)) @ *v # snd* (*splitAt v* (*verticesFrom f v'*))
by (*simp add*: *verticesFrom-split*)

then **have** *vv'*: *v* \neq *v'* \implies *fst* (*splitAt v* (*verticesFrom f v'*))
= *v' # tl* (*fst* (*splitAt v* (*verticesFrom f v'*)))
by (*cases fst* (*splitAt v* (*verticesFrom f v'*))) *auto*

ultimately **have** *v* \neq *v'* \implies *vertices f21*
= *u # snd* (*splitAt u* (*verticesFrom f v'*)) @ *v' # tl* (*fst* (*splitAt v* (*verticesFrom*
f v'))) @ *v # ws*
by *auto*

moreover
with *f21* **have** *rule2*: *v'* \in \mathcal{V} *f21* **by** *auto*
with *dist-f21* **have** *dist-f21-v'*: *distinct* (*verticesFrom f21 v'*) **by** *auto*

ultimately **have** *m1*: *v* \neq *v'* \implies *verticesFrom f21 v'*
= *v' # tl* (*fst* (*splitAt v* (*verticesFrom f v'*))) @ *v # ws @ u # snd* (*splitAt u*
(*verticesFrom f v'*))
apply *auto*
apply (*subgoal-tac snd* (*splitAt v'* (*vertices f21*))) = *tl* (*fst* (*splitAt v* (*verticesFrom*

$f v')$) @ $v \# ws$)
apply (*subgoal-tac* *fst* (*splitAt* v' (*vertices* $f21$)) = $u \# snd$ (*splitAt* u (*verticesFrom* $f v'$)))
apply (*subgoal-tac* *verticesFrom* $f21 v' = v' \# snd$ (*splitAt* v' (*vertices* $f21$)))
@ *fst* (*splitAt* v' (*vertices* $f21$)))
apply *simp*
apply (*intro* *verticesFrom-v* *dist-f21*) **apply** *force*
apply (*subgoal-tac* *distinct* (*vertices* $f21$)) **apply** *simp*
apply (*rule-tac* *dist-f21*)
apply (*subgoal-tac* *distinct* (*vertices* $f21$)) **apply** *simp*
by (*rule-tac* *dist-f21*)

from *pre-add* **have** *dist-vf-v'*: *distinct* (*verticesFrom* $f v'$) **by** (*simp* *add*: *pre-subdivFace'-def*)
with *vert-f21* **have** *m2*: $v = v' \implies verticesFrom f21 v' = v' \# ws$ @ $u \# snd$ (*splitAt* u (*verticesFrom* $f v'$))
apply *auto* **apply** (*intro* *verticesFrom-v* *dist-f21*) **by** *simp*

from *pre-add* **have** $u: u \in set$ (*verticesFrom* $f v'$) **by** (*fastforce* *simp*: *pre-subdivFace'-def* *before-def*)
then **have** *split-u*: *verticesFrom* $f v'$
= *fst* (*splitAt* u (*verticesFrom* $f v'$)) @ $u \# snd$ (*splitAt* u (*verticesFrom* $f v'$))
by (*auto* *dest!*: *splitAt-ram*)

then **have** *rule1'*: $[v \leftarrow snd$ (*splitAt* u (*verticesFrom* $f v'$)) . $v \in set$ (*removeNones* *vol*)] = *removeNones* *vol*
proof –
from *split-u* **have** $v' \# tl$ (*verticesFrom* $f v'$)
= *fst* (*splitAt* u (*verticesFrom* $f v'$)) @ $u \# snd$ (*splitAt* u (*verticesFrom* $f v'$))
by (*simp* *add*: *verticesFrom-split*)
have *help*: $set \ [] \cap set$ (*verticesFrom* $f v'$) $\subseteq \{u\} \cup set$ (*fst* (*splitAt* u (*verticesFrom* $f v'$))) **by** *auto*
from *split-u* *dist-vf-v'* *pre-add*
have $[v \leftarrow \ []$ @ snd (*splitAt* u (*verticesFrom* $f v'$)) . $v \in set$ (*removeNones* *vol*)]
= *removeNones* *vol*
apply (*rule-tac* *filter-distinct-at5*) **apply** *assumption+*
apply (*simp* *add*: *pre-subdivFace'-def*) **by** (*rule* *help*)
then **show** *?thesis* **by** *auto*
qed
then **have** *inSnd-u*: $\bigwedge x. x \in set$ (*removeNones* *vol*) $\implies x \in set$ (*snd* (*splitAt* u (*verticesFrom* $f v'$)))
apply (*subgoal-tac* $x \in set$ $[v \leftarrow snd$ (*splitAt* u (*verticesFrom* $f v'$)) . $v \in set$ (*removeNones* *vol*)] \implies
 $x \in set$ (*snd* (*splitAt* u (*verticesFrom* $f v'$))))
apply *force* **apply** (*thin-tac* $[v \leftarrow snd$ (*splitAt* u (*verticesFrom* $f v'$)) . $v \in set$ (*removeNones* *vol*)] = *removeNones* *vol*)
by *simp*

from *split-u* *dist-vf-v'* **have** *notinFst-u*: $\bigwedge x. x \in set$ (*removeNones* *vol*) \implies

```

    x ∉ set ((fst (splitAt u (verticesFrom f v'))) @ [u]) apply (drule-tac inSnd-u)
    apply (subgoal-tac distinct (fst (splitAt u (verticesFrom f v'))) @ u # snd
(splitAt u (verticesFrom f v'))))
    apply (thin-tac verticesFrom f v'
= fst (splitAt u (verticesFrom f v'))) @ u # snd (splitAt u (verticesFrom f
v')))
    apply simp apply safe
    apply (subgoal-tac x ∈ set (fst (splitAt u (verticesFrom f v'))) ∩ set (snd (splitAt
u (verticesFrom f v'))))
    apply simp
    apply (thin-tac set (fst (splitAt u (verticesFrom f v'))) ∩ set (snd (splitAt u
(verticesFrom f v'))) = {}))
    apply simp
    by (simp only:)

from rule2 v' have  $\bigwedge a b. is\_nextElem (vertices f) a b \wedge a \in set (removeNones
vol) \wedge b \in set (removeNones vol) \implies$ 
is-nextElem (vertices f21) a b
proof -
  fix a b
  assume vors: is-nextElem (vertices f) a b  $\wedge a \in set (removeNones vol) \wedge b \in$ 
set (removeNones vol)
  define vor-u where vor-u = fst (splitAt u (verticesFrom f v'))
  define nach-u where nach-u = snd (splitAt u (verticesFrom f v'))
  from vors v' have is-nextElem (verticesFrom f v') a b by (simp add: vertices-
From-is-nextElem)
  moreover
  from vors inSnd-u nach-u-def have a ∈ set (nach-u) by auto
  moreover
  from vors inSnd-u nach-u-def have b ∈ set (nach-u) by auto
  moreover
  from split-u vor-u-def nach-u-def have verticesFrom f v' = vor-u @ u # nach-u
by auto
  moreover
  note dist-vf-v'
  ultimately have is-sublist [a,b] (nach-u) apply (simp add: is-nextElem-def
split:if-split-asm)
  apply (subgoal-tac b ≠ hd (vor-u @ u # nach-u))
  apply simp
  apply (subgoal-tac distinct (vor-u @ (u # nach-u)))
  apply (drule is-sublist-at5)
  apply simp
  apply simp
  apply (erule disjE)
  apply (drule is-sublist-in1)+
  apply (subgoal-tac b ∈ set vor-u ∩ set nach-u) apply simp
  apply (thin-tac set vor-u ∩ set nach-u = {})
  apply simp
  apply (erule disjE)

```

```

apply (subgoal-tac distinct ([u] @ nach-u))
apply (drule is-sublist-at5)
  apply simp
  apply simp
apply (erule disjE)
  apply simp
  apply simp
apply simp
apply (subgoal-tac distinct (vor-u @ (u # nach-u)))
apply (drule is-sublist-at5) apply simp
apply (erule disjE)
apply (drule is-sublist-in1)+
apply simp
apply (erule disjE)
apply (drule is-sublist-in1)+ apply simp
apply simp
apply simp
apply simp
apply (cases vor-u) by auto

with nach-u-def have is-sublist [a,b] (snd (splitAt u (verticesFrom f v'))) by
auto
then have is-sublist [a,b] (verticesFrom f21 v')
  apply (cases v = v') apply (simp-all add: m1 m2)
apply (subgoal-tac is-sublist [a, b] ((v' # ws @ [u]) @ snd (splitAt u (verticesFrom
f v'))) @ [])
  apply simp apply (rule is-sublist-add) apply simp
  apply (subgoal-tac is-sublist [a, b]
    ((v' # tl (fst (splitAt v (verticesFrom f v'))) @ v # ws @ [u]) @ (snd (splitAt
u (verticesFrom f v'))) @ []))
  apply simp apply (rule is-sublist-add) by simp
with rule2 show is-nextElem (vertices f) a b  $\wedge$  a  $\in$  set (removeNones vol)  $\wedge$ 
b  $\in$  set (removeNones vol)  $\implies$ 
  is-nextElem (vertices f21) a b apply (simp add: verticesFrom-is-nextElem)
by (auto simp: is-nextElem-def)
qed
with pre-add dist-f21 have rule5':
 $\wedge$  a b. (a,b)  $\in$  edges f  $\wedge$  a  $\in$  set (removeNones vol)  $\wedge$  b  $\in$  set (removeNones
vol)  $\implies$  (a, b)  $\in$  edges f21
by (simp add: is-nextElem-edges-eq pre-subdivFace'-def)

have rule1: [v $\leftarrow$ verticesFrom f21 v' . v  $\in$  set (removeNones vol)]
= removeNones vol  $\wedge$  hd (removeNones vol)  $\in$  set (snd (splitAt u (verticesFrom
f v')))
proof (cases v = v')
case True
from split-u have v' # tl (verticesFrom f v')
= fst (splitAt u (verticesFrom f v'))) @ u # snd (splitAt u (verticesFrom f v'))

```

by (simp add: verticesFrom-split)
 then have $u \neq v' \implies \text{fst} (\text{splitAt } u (\text{verticesFrom } f v'))$
 $= v' \# \text{tl} (\text{fst} (\text{splitAt } u (\text{verticesFrom } f v')))$ by (cases fst (splitAt u
 (verticesFrom f v'))) auto
 moreover
 have $v' \in \text{set} (v' \# \text{tl} (\text{fst} (\text{splitAt } u (\text{verticesFrom } f v'))))$ by simp
 ultimately have $u \neq v' \implies v' \in \text{set} (\text{fst} (\text{splitAt } u (\text{verticesFrom } f v')))$ by
 simp
 moreover
 from pre-fdg have $\text{set} (v' \# \text{ws} @ [u]) \cap \text{set} (\text{verticesFrom } f v') \subseteq \{v', u\}$
 apply (simp add: set-eq)
 by (unfold pre-splitFace-def) auto
 ultimately have help: $\text{set} (v' \# \text{ws} @ [u]) \cap \text{set} (\text{verticesFrom } f v')$
 $\subseteq \{u\} \cup \text{set} (\text{fst} (\text{splitAt } u (\text{verticesFrom } f v')))$ apply (rule-tac subset-trans)
 apply assumption apply (cases $u = v'$) by simp-all
 from split-u dist-vf-v' pre-add pre-fdg removeNones-split have
 $[v \leftarrow (v' \# \text{ws} @ [u]) @ \text{snd} (\text{splitAt } u (\text{verticesFrom } f v')) . v \in \text{set}$
 (removeNones vol)]
 $= \text{removeNones vol} \wedge \text{hd} (\text{removeNones vol}) \in \text{set} (\text{snd} (\text{splitAt } u (\text{verticesFrom}$
 $f v')))$
 apply (rule-tac filter-distinct-at-special) apply assumption+
 apply (simp add: pre-subdivFace'-def) apply (rule help) .
 with True m2 show ?thesis by auto
 next
 case False

 with m1 dist-f21-v' have ne-uv': $u \neq v'$ by auto
 define fst-u where $\text{fst-u} = \text{fst} (\text{splitAt } u (\text{verticesFrom } f v'))$
 define fst-v where $\text{fst-v} = \text{fst} (\text{splitAt } v (\text{verticesFrom } f v'))$

 from pre-add u dist-vf-v' have $v \in \text{set} (\text{fst} (\text{splitAt } u (\text{verticesFrom } f v')))$
 apply (rule-tac before-dist-r1) by (auto simp: pre-subdivFace'-def)
 with fst-u-def have $\text{fst-u} = \text{fst} (\text{splitAt } v (\text{fst} (\text{splitAt } u (\text{verticesFrom } f v'))))$
 $@ v \# \text{snd} (\text{splitAt } v (\text{fst} (\text{splitAt } u (\text{verticesFrom } f v'))))$
 by (auto dest: splitAt-ram)
 with pre-add fst-v-def pre-bet' have $\text{fst-u}': \text{fst-u}$
 $= \text{fst-v} @ v \# \text{snd} (\text{splitAt } v (\text{fst} (\text{splitAt } u (\text{verticesFrom } f v'))))$ by (simp
 add: pre-subdivFace'-def)

 from pre-fdg have $\text{set} (v \# \text{ws} @ [u]) \cap \text{set} (\text{verticesFrom } f v') \subseteq \{v, u\}$ apply
 (simp add: set-eq)
 by (unfold pre-splitFace-def) auto

 with fst-u' have $\text{set} (v \# \text{ws} @ [u]) \cap \text{set} (\text{verticesFrom } f v') \subseteq \{u\} \cup \text{set } \text{fst-u}$
 by auto
 moreover
 from fst-u' have $\text{set } \text{fst-v} \subseteq \text{set } \text{fst-u}$ by auto
 ultimately
 have $(\text{set} (v \# \text{ws} @ [u]) \cup \text{set } \text{fst-v}) \cap \text{set} (\text{verticesFrom } f v') \subseteq \{u\} \cup \text{set}$

fst-u **by** *auto*
with *fst-u-def* *fst-v-def*
have $\text{set} (\text{fst} (\text{splitAt } v (\text{verticesFrom } f \ v')) \ @ \ v \ # \ \text{ws} \ @ \ [u]) \cap \text{set} (\text{verticesFrom } f \ v')$
 $\subseteq \{u\} \cup \text{set} (\text{fst} (\text{splitAt } u (\text{verticesFrom } f \ v')))$ **by** *auto*
moreover
with *False* *vv'* **have** $v' \ # \ \text{tl} (\text{fst} (\text{splitAt } v (\text{verticesFrom } f \ v')))$
 $= \text{fst} (\text{splitAt } v (\text{verticesFrom } f \ v'))$ **by** *auto*
ultimately **have** $\text{set} ((v' \ # \ \text{tl} (\text{fst} (\text{splitAt } v (\text{verticesFrom } f \ v')))) \ @ \ v \ # \ \text{ws} \ @ \ [u]) \cap \text{set} (\text{verticesFrom } f \ v')$
 $\subseteq \{u\} \cup \text{set} (\text{fst} (\text{splitAt } u (\text{verticesFrom } f \ v')))$
by (*simp only*:)
then **have** *help*: $\text{set} (v' \ # \ \text{tl} (\text{fst} (\text{splitAt } v (\text{verticesFrom } f \ v')))) \ @ \ v \ # \ \text{ws} \ @ \ [u] \cap \text{set} (\text{verticesFrom } f \ v')$
 $\subseteq \{u\} \cup \text{set} (\text{fst} (\text{splitAt } u (\text{verticesFrom } f \ v')))$ **by** *auto*

from *split-u* *dist-vf-v'* *pre-add* *pre-fdg* *removeNones-split* **have**
 $[v \leftarrow (v' \ # \ \text{tl} (\text{fst} (\text{splitAt } v (\text{verticesFrom } f \ v')))) \ @ \ v \ # \ \text{ws} \ @ \ [u]]$
 $\ @ \ \text{snd} (\text{splitAt } u (\text{verticesFrom } f \ v')) \ . \ v \in \text{set} (\text{removeNones } \text{vol})$
 $= \text{removeNones } \text{vol} \wedge \text{hd} (\text{removeNones } \text{vol}) \in \text{set} (\text{snd} (\text{splitAt } u (\text{verticesFrom } f \ v')))$
apply (*rule-tac* *filter-distinct-at-special*) **apply** *assumption+*
apply (*simp add: pre-subdivFace'-def*) **apply** (*rule help*) .
with *False* *m1* **show** *?thesis* **by** *auto*
qed

from *rule1* **have** $(\text{hd} (\text{removeNones } \text{vol})) \in \text{set} (\text{snd} (\text{splitAt } u (\text{verticesFrom } f \ v')))$ **by** *auto*
with *m1* *m2* *dist-f21-v'* **have** *rule3*: *before* $(\text{verticesFrom } f \ 21 \ v') \ u \ (\text{hd} (\text{removeNones } \text{vol}))$
proof –
assume *hd-ram*: $(\text{hd} (\text{removeNones } \text{vol})) \in \text{set} (\text{snd} (\text{splitAt } u (\text{verticesFrom } f \ v')))$
from *m1* *m2* *dist-f21-v'* **have** *distinct* $(\text{snd} (\text{splitAt } u (\text{verticesFrom } f \ v')))$
apply (*cases* $v = v'$)
by *auto*
moreover
define *z1* **where** $z1 = \text{fst} (\text{splitAt} (\text{hd} (\text{removeNones } \text{vol})) (\text{snd} (\text{splitAt } u (\text{verticesFrom } f \ v'))))$
define *z2* **where** $z2 = \text{snd} (\text{splitAt} (\text{hd} (\text{removeNones } \text{vol})) (\text{snd} (\text{splitAt } u (\text{verticesFrom } f \ v'))))$
note *z1-def* *z2-def* *hd-ram*
ultimately **have** $\text{snd} (\text{splitAt } u (\text{verticesFrom } f \ v')) = z1 \ @ \ (\text{hd} (\text{removeNones } \text{vol})) \ # \ z2$
by (*auto intro: splitAt-ram*)
with *m1* *m2* **show** *?thesis* **apply** (*cases* $v = v'$) **apply** (*auto simp: before-def*)
apply (*intro exI*)

apply (*subgoal-tac* $v' \# ws @ u \# z1 @ hd (removeNones vol) \# z2 = (v' \# ws) @ u \# z1 @ hd (removeNones vol) \# z2$)
apply *assumption* **apply** *simp*
apply (*intro* *exI*)
apply (*subgoal-tac* $v' \# tl (fst (splitAt v (verticesFrom f v'))) @ v \# ws @ u \# z1 @ hd (removeNones vol) \# z2 = (v' \# tl (fst (splitAt v (verticesFrom f v'))) @ v \# ws) @ u \# z1 @ hd (removeNones vol) \# z2$)
apply *assumption* **by** *simp*
qed

from *rule1* **have** $ne:(hd (removeNones vol)) \in set (snd (splitAt u (verticesFrom f v')))$ **by** *auto*
with *m1 m2* **have** $last (verticesFrom f21 v') = last (snd (splitAt u (verticesFrom f v')))$
apply (*cases* $snd (splitAt u (verticesFrom f v'))$ *rule: rev-exhaust*) **apply** *simp-all*
apply (*cases* $v = v'$) **by** *simp-all*
moreover
from *ne* **have** $last (fst (splitAt u (verticesFrom f v'))) @ u \# snd (splitAt u (verticesFrom f v')) = last (snd (splitAt u (verticesFrom f v')))$ **by** *auto*
moreover
note *split-u*
ultimately **have** *rule4*: $last (verticesFrom f v') = last (verticesFrom f21 v')$ **by** *simp*

have $l: \bigwedge a b f v. v \in set (vertices f) \implies is_nextElem (vertices f) a b = is_nextElem (verticesFrom f v) a b$
apply (*rule is-nextElem-congs-eq*) **by** (*rule verticesFrom-congs*)

define *f12* **where** $f12 = fst (splitFace f v u ws)$
then **have** *f12-fdg*: $f12 = fst (splitFace g v u f ws)$
by (*simp add: splitFace-def split-def*)

from *pre-bet pre-add* **have** *bet-eq2[simp]*: $between (vertices f) v u = between (verticesFrom f v') v u$
apply (*drule-tac pre-between-symI*)
by (*auto intro: verticesFrom-between simp: pre-subdivFace'-def*)

from *f12-fdg* **have** *f12-split-face*: $f12 = fst (splitFace f v u ws)$
by (*simp add: splitFace-def split-def*)
then **have** *f12*: $f12 = Face (rev ws @ v \# between (verticesFrom f v') v u @ [u])$
Nonfinal
by (*simp add: split-face-def*)
then **have** $vertices f12 = rev ws @ v \# between (verticesFrom f v') v u @ [u]$ **by** *simp*
with *pre-add pre-bet'* **have** *vert-f12*: $vertices f12$


```

    = rev ws @ v # snd (splitAt v (fst (splitAt u (verticesFrom f v')))) @ [u]
apply (subgoal-tac between (verticesFrom f v') v u = fst (splitAt u (snd (splitAt
v (verticesFrom f v')))))
    apply (simp add: pre-subdivFace'-def)
    apply (rule between-simp1)
    apply (simp add: pre-subdivFace'-def)
    apply (rule pre-between-symI) .
with dist-f21-v' have removeNones-vol-not-f12:  $\bigwedge x. x \in \text{set } (\text{removeNones vol})$ 
 $\implies x \notin \text{set } (\text{vertices } f12)$ 
    apply (frule-tac notinFst-u) apply (drule inSnd-u) apply simp
    apply (case-tac v = v') apply (simp add: m1 m2)
    apply (rule conjI) apply force
    apply (rule conjI) apply (rule ccontr) apply simp
    apply (subgoal-tac  $x \in \text{set } ws \cap \text{set } (\text{snd } (\text{splitAt } u (\text{verticesFrom } f v')))$ )
    apply simp apply (elim conjE)
    apply (thin-tac  $\text{set } ws \cap \text{set } (\text{snd } (\text{splitAt } u (\text{verticesFrom } f v'))) = \{\}$ )
    apply simp
    apply force

    apply (simp add: m1 m2)
    apply (rule conjI) apply force
    apply (rule conjI) apply (rule ccontr) apply simp
    apply (subgoal-tac  $x \in \text{set } ws \cap \text{set } (\text{snd } (\text{splitAt } u (\text{verticesFrom } f v')))$ )
    apply simp apply (elim conjE)
    apply (thin-tac  $\text{set } ws \cap \text{set } (\text{snd } (\text{splitAt } u (\text{verticesFrom } f v'))) = \{\}$ ) apply
simp
    by force

from pre-fdg f12-split-face have dist-f12: distinct (vertices f12) by (auto intro:
split-face-distinct1')

    then have removeNones-vol-edges-not-f12:  $\bigwedge x y. x \in \text{set } (\text{removeNones vol})$ 
 $\implies (x,y) \notin \text{edges } f12$ 
    apply (drule-tac removeNones-vol-not-f12) by auto
    from dist-f12 have removeNones-vol-edges-not-f12':  $\bigwedge x y. y \in \text{set } (\text{removeNones}$ 
vol)  $\implies (x,y) \notin \text{edges } f12$ 
    apply (drule-tac removeNones-vol-not-f12) by auto

from f12-fdg pre-fdg g' fdg have face-set-eq:  $\mathcal{F} g' \cup \{f\} = \{f12, f21\} \cup \mathcal{F} g$ 
apply (rule-tac splitFace-faces-1)
by (simp-all)

have rule5'':  $\bigwedge a b. (a,b) \in \text{edges } g' \wedge (a,b) \notin \text{edges } g$ 
 $\wedge a \in \text{set } (\text{removeNones vol}) \wedge b \in \text{set } (\text{removeNones vol}) \implies (a, b) \in \text{edges}$ 
f21
    apply (simp add: edges-graph-def) apply safe
    apply (case-tac x = f) apply simp apply (rule rule5') apply safe
    apply (subgoal-tac  $x \in \mathcal{F} g' \cup \{f\}$ ) apply (thin-tac  $x \neq f$ )
    apply (thin-tac  $x \in \text{set } (\text{faces } g')$ ) apply (simp only: add: face-set-eq)

```

apply safe apply (drule removeNones-vol-edges-not-f12) **by auto**
have rule5''': $\bigwedge a b. (a,b) \in \text{edges } g' \wedge (a,b) \notin \text{edges } g$
 $\wedge a \in \text{set } (\text{removeNones } \text{vol}) \wedge b \in \text{set } (\text{removeNones } \text{vol}) \implies (a, b) \in \text{edges}$
f21

apply (simp add: edges-graph-def) **apply safe**
apply (case-tac $x = f$) **apply simp apply** (rule rule5') **apply safe**
apply (subgoal-tac $x \in \mathcal{F} g' \cup \{f\}$) **apply** (thin-tac $x \neq f$)
apply (thin-tac $x \in \mathcal{F} g'$) **apply** (simp only: add: face-set-eq)
apply safe apply (drule removeNones-vol-edges-not-f12) **by auto**

from pre-fdg fdg f12-fdg g' have edges-g'1: $ws \neq [] \implies \text{edges } g' = \text{edges } g \cup$
 $\text{Edges } ws \cup \text{Edges}(\text{rev } ws) \cup$
 $\{(u, \text{last } ws), (\text{hd } ws, v), (v, \text{hd } ws), (\text{last } ws, u)\}$
apply (rule-tac splitFace-edges-g') **apply simp**
apply (subgoal-tac $(f12, f21, g') = \text{splitFace } g v u f ws$) **apply assumption by**
auto

from pre-fdg fdg f12-fdg g' have edges-g'2: $ws = [] \implies \text{edges } g' = \text{edges } g \cup$
 $\{(v, u), (u, v)\}$
apply (rule-tac splitFace-edges-g'-vs) **apply simp**
apply (subgoal-tac $(f12, f21, g') = \text{splitFace } g v u f []$) **apply assumption by**
auto

from f12-split-face f21-split-face have split: $(f12, f21) = \text{split-face } f v u ws$ **by**
simp

from pre-add have $\neg \text{invalidVertexList } g f \text{vol}$
by (auto simp: pre-subdivFace'-def dest: invalidVertexList-shorten)
then have rule5: $\neg \text{invalidVertexList } g' f21 \text{vol}$

apply (simp add: invalidVertexList-def)
apply (intro allI impI)
apply (case-tac vol!i) **apply simp+**
apply (case-tac vol!Suc i) **apply simp+**
apply (subgoal-tac $\neg \text{is-duplicateEdge } g f a aa$)
apply (thin-tac $\forall i < |\text{vol}| - \text{Suc } 0. \neg (\text{case } \text{vol} ! i \text{ of } \text{None} \Rightarrow \text{False}$
 $| \text{Some } a \Rightarrow \text{case-option } \text{False } (\text{is-duplicateEdge } g f a) (\text{vol} ! (i+1))))$)
apply (simp add: is-duplicateEdge-def)
apply (subgoal-tac $a \in \text{set } (\text{removeNones } \text{vol}) \wedge aa \in \text{set } (\text{removeNones } \text{vol})$)
apply (rule conjI)
apply (rule impI)
apply (case-tac $(a, aa) \in \text{edges } f$)
apply simp
apply (subgoal-tac pre-split-face $f v u ws$)
apply (frule split-face-edges-or [OF split]) **apply simp**

```

    apply (simp add: removeNones-vol-edges-not-f12)
    apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
  apply (case-tac (aa, a) ∈ edges f)
  apply simp
  apply (subgoal-tac pre-split-face f v u ws)
    apply (frule split-face-edges-or [OF split]) apply simp
    apply (simp add: removeNones-vol-edges-not-f12)
  apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
  apply simp
  apply (case-tac ws = []) apply (frule edges-g'2) apply simp
  apply (subgoal-tac pre-split-face f v u [])
    apply (subgoal-tac (f12, f21) = split-face f v u ws)
    apply (case-tac between (vertices f) u v = [])
      apply (frule split-face-edges-f21-bet-vs) apply simp apply simp
      apply simp
    apply (frule split-face-edges-f21-vs) apply simp apply simp apply simp
    apply (case-tac a = v ∧ aa = u) apply simp apply simp
  apply (rule split)
  apply (subgoal-tac pre-split-face f v u ws) apply simp
  apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
  apply (frule edges-g'1) apply simp
  apply (subgoal-tac pre-split-face f v u ws)
    apply (subgoal-tac (f12, f21) = split-face f v u ws)
    apply (case-tac between (vertices f) u v = [])
      apply (frule split-face-edges-f21-bet) apply simp apply simp apply simp
      apply simp
    apply (case-tac a = u ∧ aa = last ws) apply simp apply simp
    apply (case-tac a = hd ws ∧ aa = v) apply simp apply simp
    apply (case-tac a = v ∧ aa = hd ws) apply simp apply simp
    apply (case-tac a = last ws ∧ aa = u) apply simp apply simp
    apply (case-tac (a, aa) ∈ Edges ws) apply simp
  apply simp
  apply (frule split-face-edges-f21) apply simp apply simp apply simp
  apply simp
  apply (force)
  apply (rule split)
  apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
  apply (rule impI)
  apply (case-tac (aa,a) ∈ edges f) apply simp
  apply (subgoal-tac pre-split-face f v u ws)
    apply (frule split-face-edges-or [OF split]) apply simp
    apply (simp add: removeNones-vol-edges-not-f12)
  apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
  apply (case-tac (a,aa) ∈ edges f) apply simp
  apply (subgoal-tac pre-split-face f v u ws)
    apply (frule split-face-edges-or [OF split]) apply simp
    apply (simp add: removeNones-vol-edges-not-f12)
  apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
  apply simp

```

```

apply (case-tac ws = []) apply (frule edges-g'2) apply simp
apply (subgoal-tac pre-split-face f v u [])
apply (subgoal-tac (f12, f21) = split-face f v u ws)
apply (case-tac between (vertices f) u v = [])
apply (frule split-face-edges-f21-bet-vs) apply simp apply simp
apply simp
apply (frule split-face-edges-f21-vs) apply simp apply simp apply simp
apply force
apply (rule split)
apply (subgoal-tac pre-split-face f v u ws) apply simp
apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
apply (frule edges-g'1) apply simp
apply (subgoal-tac pre-split-face f v u ws)
apply (subgoal-tac (f12, f21) = split-face f v u ws)
apply (case-tac between (vertices f) u v = [])
apply (frule split-face-edges-f21-bet) apply simp apply simp apply simp
apply (force)
apply (frule split-face-edges-f21) apply simp apply simp apply simp apply
simp
apply (force)
apply (rule split)
apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
apply (rule conjI)
apply (subgoal-tac Some a ∈ set vol) apply (induct vol) apply simp apply
force
apply (subgoal-tac vol ! i ∈ set vol) apply simp
apply (rule nth-mem) apply arith
apply (subgoal-tac Some aa ∈ set vol) apply (induct vol) apply simp apply
force
apply (subgoal-tac vol ! (Suc i) ∈ set vol) apply simp apply (rule nth-mem)
apply arith
by auto

from pre-fdg dist-f21 v' have dists: distinct (vertices f) distinct (vertices f12)
distinct (vertices f21) v' ∈ V f
apply auto defer
apply (drule splitFace-distinct2) apply (simp add: f12-fdg)
apply (unfold pre-splitFace-def) by simp
with pre-fdg have edges-or:  $\bigwedge a b. (a,b) \in \text{edges } f \implies (a,b) \in \text{edges } f12 \vee (a,b) \in \text{edges } f21$ 
apply (rule-tac split-face-edges-or) apply (simp add: f12-split-face f21-split-face)
by simp+

from pre-fdg have dist-f: distinct (vertices f) apply (unfold pre-splitFace-def)
by simp

from g' have edges-g': edges g'

```

= (UN h:set(replace f [snd (split-face f v u ws)] (faces g)). edges h)
 ∪ edges (fst (split-face f v u ws))
 by (auto simp add: splitFace-def split-def edges-graph-def)

from pre-fdg edges-g' **have** edges-g'-or:
 $\bigwedge a b. (a,b) \in \text{edges } g' \implies$
 $(a,b) \in \text{edges } g \vee (a,b) \in \text{edges } f12 \vee (a,b) \in \text{edges } f21$
apply simp **apply** (case-tac (a, b) ∈ edges (fst (split-face f v u ws)))
apply (simp add:f12-split-face) **apply** simp
apply (elim bexE) **apply** (simp add: f12-split-face) **apply** (case-tac x ∈ \mathcal{F} g)
apply (induct g) **apply** (simp add: edges-graph-def) **apply** (rule disjI1)
apply (rule beXI) **apply** simp **apply** simp
apply (drule replace1) **apply** simp **by** (simp add: f21-split-face)

have rule6: $0 < |\text{vol}| \implies \neg \text{invalidVertexList } g f (\text{Some } u \# \text{vol}) \implies$
 $(\exists y. \text{hd } \text{vol} = \text{Some } y) \longrightarrow \neg \text{is-duplicateEdge } g' f21 u (\text{the } (\text{hd } \text{vol}))$

apply (rule impI)
apply (erule exE) **apply** simp **apply** (case-tac vol) **apply** simp+
apply (simp add: invalidVertexList-def) **apply** (erule allE) **apply** (erule impE)

apply force
apply (simp)
apply (subgoal-tac $y \notin \mathcal{V} f12$) **defer** **apply** (rule removeNones-vol-not-f12)

apply simp
apply (simp add: is-duplicateEdge-def)
apply (subgoal-tac $y \in \text{set } (\text{removeNones } \text{vol})$)
apply (rule conjI)
apply (rule impI)
apply (case-tac $(u, y) \in \text{edges } f$) **apply** simp
apply (subgoal-tac pre-split-face f v u ws)
apply (frule split-face-edges-or [OF split]) **apply** simp
apply (simp add: removeNones-vol-edges-not-f12')
apply (rule pre-splitFace-pre-split-face) **apply** simp **apply** (rule pre-fdg)
apply (case-tac $(y, u) \in \text{edges } f$) **apply** simp
apply (subgoal-tac pre-split-face f v u ws)
apply (frule split-face-edges-or [OF split]) **apply** simp
apply (simp add: removeNones-vol-edges-not-f12)
apply (rule pre-splitFace-pre-split-face) **apply** simp **apply** (rule pre-fdg)

apply simp
apply (case-tac $\text{ws} = []$) **apply** (frule edges-g'2) **apply** simp
apply (subgoal-tac pre-split-face f v u [])
apply (subgoal-tac $(f12, f21) = \text{split-face } f v u \text{ws}$)
apply (case-tac between (vertices f) u v = [])
apply (frule split-face-edges-f21-bet-vs) **apply** simp **apply** simp **apply** simp

simp
apply (frule split-face-edges-f21-vs) **apply** simp **apply** simp **apply** simp
apply force

```

    apply (rule split)
    apply (subgoal-tac pre-split-face f v u ws) apply simp
    apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
    apply (frule edges-g'1) apply simp
    apply (subgoal-tac pre-split-face f v u ws)
    apply (subgoal-tac (f12, f21) = split-face f v u ws)
    apply (case-tac between (vertices f) u v = [])
    apply (frule split-face-edges-f21-bet) apply simp apply simp apply simp
    apply (force)
    apply (frule split-face-edges-f21) apply simp apply simp apply simp apply
simp
    apply (force)
    apply (rule split)
    apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
    apply (rule impI)
    apply (case-tac (u, y) ∈ edges f) apply simp
    apply (subgoal-tac pre-split-face f v u ws)
    apply (frule split-face-edges-or [OF split]) apply simp apply (simp add:
removeNones-vol-edges-not-f12')
    apply (rule pre-splitFace-pre-split-face) apply simp apply (rule pre-fdg)
    apply (case-tac (y, u) ∈ edges f) apply simp
    apply (subgoal-tac pre-split-face f v u ws)
    apply (frule split-face-edges-or [OF split]) apply simp apply (simp add:
removeNones-vol-edges-not-f12)
    apply (rule pre-splitFace-pre-split-face) apply simp apply (rule pre-fdg)
    apply simp
    apply (case-tac ws = []) apply (frule edges-g'2) apply simp
    apply (subgoal-tac pre-split-face f v u [])
    apply (subgoal-tac (f12, f21) = split-face f v u ws)
    apply (case-tac between (vertices f) u v = [])
    apply (frule split-face-edges-f21-bet-vs) apply simp apply simp
    apply simp
    apply (frule split-face-edges-f21-vs) apply simp apply simp apply simp
    apply force
    apply (rule split)
    apply (subgoal-tac pre-split-face f v u ws) apply simp
    apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
    apply (frule edges-g'1) apply simp
    apply (subgoal-tac pre-split-face f v u ws)
    apply (subgoal-tac (f12, f21) = split-face f v u ws)
    apply (case-tac between (vertices f) u v = [])
    apply (frule split-face-edges-f21-bet) apply simp apply simp apply simp
    apply (force)
    apply (frule split-face-edges-f21) apply simp apply simp apply simp apply
simp
    apply (force)
    apply (rule split)
    apply (rule pre-splitFace-pre-split-face) apply (rule pre-fdg)
by simp

```

```

have u21:  $u \in \mathcal{V}$  f21 by(simp add:f21)
from fdg have  $\neg$  final f21
  by(simp add:splitFace-def split-face-def split-def)
with pre-add rule1 rule2 rule3 rule4 rule5 rule6 dist-f21 False dist u21
show ?thesis by (simp-all add: pre-subdivFace'-def l)
qed

```

```

lemma before-filter:  $\bigwedge$  ys. filter P xs = ys  $\implies$  distinct xs  $\implies$  before ys u v  $\implies$ 
before xs u v
  supply subst-all [simp del]
  apply (subgoal-tac P u)
  apply (subgoal-tac P v)
  apply (subgoal-tac pre-between xs u v)
  apply (rule ccontr) apply (simp add: before-xor)
  apply (subgoal-tac before ys v u)
  apply (subgoal-tac  $\neg$  before ys v u)
  apply simp
  apply (rule before-dist-not1) apply force apply simp
  apply (simp add: before-def) apply (elim exE) apply simp
  apply (subgoal-tac a @ u # b @ v # c = filter P aa @ v # filter P ba @ u #
filter P ca)
  apply (intro exI) apply assumption
  apply simp
  apply (subgoal-tac  $u \in \text{set } ys \wedge v \in \text{set } ys \wedge u \neq v$ ) apply (simp add: pre-between-def)
apply force
  apply (subgoal-tac distinct ys)
  apply (simp add: before-def) apply (elim exE) apply simp
  apply force
  apply (subgoal-tac  $v \in \text{set } (\text{filter } P \text{ xs})$ ) apply force
  apply (simp add: before-def) apply (elim exE) apply simp
  apply (subgoal-tac  $u \in \text{set } (\text{filter } P \text{ xs})$ ) apply force
  apply (simp add: before-def) apply (elim exE) by simp

```

```

lemma pre-subdivFace'-Some2: pre-subdivFace' g f v' v 0 ((Some u) # vol)  $\implies$ 
pre-subdivFace' g f v' u 0 vol
apply (cases vol = [])
apply (simp add: pre-subdivFace'-def)
  apply (cases u = v') apply simp
apply(rule verticesFrom-in')
  apply(rule last-in-set)
  apply(simp add:verticesFrom-Def)
apply clarsimp
apply (simp add: pre-subdivFace'-def)
apply (elim conjE)
apply (thin-tac  $v' = v \wedge u \neq \text{last } (\text{verticesFrom } f \text{ v}') \vee v' \neq v$ )
apply auto
  apply(rule verticesFrom-in'[where v = v'])

```

```

    apply (clarsimp simp:before-def)
  apply simp
  apply (rule owl-shorten) apply simp
  apply (subgoal-tac [v ← verticesFrom f v' . v ∈ set (removeNones ((Some u) #
vol))]) = removeNones ((Some u) # vol))
    apply assumption
    apply simp
  apply (rule before-filter)
    apply assumption
    apply simp
  apply (simp add: before-def)
  apply (intro exI)
  apply (subgoal-tac u # removeNones vol = [] @ u # [] @ hd (removeNones vol)
# tl (removeNones vol)) apply assumption
    apply simp
    apply (subgoal-tac removeNones vol ≠ []) apply simp
    apply (cases vol rule: rev-exhaust) apply simp-all
  apply (simp add: invalidVertexList-shorten)
  apply (simp add: is-duplicateEdge-def)
  apply (case-tac vol) apply simp
  apply simp
  apply (simp add: invalidVertexList-def)
  apply (elim allE)
  apply (rotate-tac -1)
  apply (erule impE)
  apply (subgoal-tac 0 < Suc |list|)
    apply assumption
    apply simp
  apply simp
  by (simp add: is-duplicateEdge-def)

lemma pre-subdivFace'-preFaceDiv: pre-subdivFace' g f v' v n ((Some u) # vol)
  ⇒ f ∈  $\mathcal{F}$  g ⇒ (f · v = u → n ≠ 0) ⇒  $\mathcal{V}$  f ⊆  $\mathcal{V}$  g
  ⇒ pre-splitFace g v u f [countVertices g ..< countVertices g + n]
proof -
  assume pre-add: pre-subdivFace' g f v' v n ((Some u) # vol) and f: f ∈  $\mathcal{F}$  g
  and nextVert: (f · v = u → n ≠ 0) and subset:  $\mathcal{V}$  f ⊆  $\mathcal{V}$  g
  have distinct [countVertices g ..< countVertices g + n] by (induct n) auto
  moreover
  have  $\mathcal{V}$  g ∩ set [countVertices g ..< countVertices g + n] = {}
    apply (cases g) by auto
  with subset have  $\mathcal{V}$  f ∩ set [countVertices g ..< countVertices g + n] = {} by
auto
  moreover
  from pre-add have  $\mathcal{V}$  f = set (verticesFrom f v') apply (intro congs-pres-nodes
verticesFrom-congs)
    by (simp add: pre-subdivFace'-def)
  with pre-add have help: v ∈  $\mathcal{V}$  f ∧ u ∈  $\mathcal{V}$  f ∧ v ≠ u
    apply (simp add: pre-subdivFace'-def before-def)

```



```

    apply (elim conjE exE)
    apply (subgoal-tac distinct (verticesFrom f v')) apply force
    apply (rule verticesFrom-distinct) by simp-all
  moreover
  from help pre-add nextVert have help1: is-nextElem (vertices f) v u  $\implies$   $0 < n$ 
  apply auto
    apply (simp add: nextVertex-def)
    by (simp add: nextElem-is-nextElem pre-subdivFace'-def)
  moreover
  have help2: before (verticesFrom f v') v u  $\implies$  distinct (verticesFrom f v')  $\implies$   $v \neq v' \implies \neg$  is-nextElem (verticesFrom f v') u v
    apply (simp add: before-def is-nextElem-def verticesFrom-hd is-sublist-def) apply
  apply safe
    apply (frule dist-at)
    apply simp
    apply (thin-tac verticesFrom f v' = a @ v # b @ u # c)
    apply (subgoal-tac verticesFrom f v' = (as @ [u]) @ v # bs) apply assumption
    apply simp apply (subgoal-tac distinct (a @ v # b @ u # c)) apply force by
  simp
  note pre-add f
  moreover

  from pre-add f help2 help1 help have [countVertices g..<countVertices g + n] =
  []  $\implies$   $(v, u) \notin$  edges f  $\wedge$   $(u, v) \notin$  edges f
    apply (cases  $0 < n$ ) apply (induct g) apply simp+
    apply (simp add: pre-subdivFace'-def)
    apply (rule conjI) apply force
    apply (simp split: if-split-asm)
    apply (rule ccontr) apply simp
    apply (subgoal-tac  $v = v'$ ) apply simp apply (elim conjE) apply (simp
  only:)
    apply (rule verticesFrom-is-nextElem-last) apply force apply force
    apply (simp add: verticesFrom-is-nextElem [symmetric])
    apply (cases  $v = v'$ ) apply simp
    apply (subgoal-tac  $v' \in \mathcal{V} f$ )
    apply (thin-tac  $u \in \mathcal{V} f$ )

    apply (simp add: verticesFrom-is-nextElem)
    apply (rule ccontr) apply simp
    apply (subgoal-tac  $v' \in \mathcal{V} f$ )
    apply (drule verticesFrom-is-nextElem-hd) apply simp+

  apply (elim conjE) apply (drule help2)
    apply simp apply simp
    apply (subgoal-tac is-nextElem (vertices f) u v = is-nextElem (verticesFrom f
  v') u v)
    apply simp
    apply (rule verticesFrom-is-nextElem) by simp
  ultimately

```

```

show ?thesis
  apply (simp add: pre-subdivFace'-def)
  apply (unfold pre-splitFace-def)
  apply simp
  apply (cases 0 < n) apply (induct g) apply (simp add: ivl-disj-int)
  apply (auto simp: invalidVertexList-def is-duplicateEdge-def)
done
qed

```

```

lemma pre-subdivFace'-Some1:
  pre-subdivFace' g f v' v n ((Some u) # vol)
   $\implies f \in \mathcal{F} g \implies (f \cdot v = u \longrightarrow n \neq 0) \implies \mathcal{V} f \subseteq \mathcal{V} g$ 
   $\implies f21 = fst (snd (splitFace g v u f [countVertices g ..< countVertices g + n]))$ 
   $\implies g' = snd (snd (splitFace g v u f [countVertices g ..< countVertices g + n]))$ 
   $\implies pre-subdivFace' g' f21 v' u 0 vol$ 
  by (meson pre-subdivFace'-Some1' pre-subdivFace'-preFaceDiv)
end

```

13 Invariants of (Plane) Graphs

```

theory Invariants
imports FaceDivisionProps
begin

```

13.1 Rotation of face into normal form

```

definition minVertex :: face  $\Rightarrow$  vertex where
minVertex f  $\equiv$  min-list (vertices f)

```

```

definition normFace :: face  $\Rightarrow$  vertex list where
normFace  $\equiv$   $\lambda f$ . verticesFrom f (minVertex f)

```

```

definition normFaces :: face list  $\Rightarrow$  vertex list list where
normFaces fl  $\equiv$  map normFace fl

```

```

lemma normFaces-distinct: distinct (normFaces fl)  $\implies$  distinct fl
apply (induct fl) by (auto simp: normFace-def normFaces-def)

```

13.2 Minimal (plane) graph properties

```

definition minGraphProps' :: graph  $\Rightarrow$  bool where
minGraphProps' g  $\equiv$   $\forall f \in \mathcal{F} g. 2 < |vertices f| \wedge$  distinct (vertices f)

```

```

definition edges-sym :: graph  $\Rightarrow$  bool where

```

$edges\text{-}sym\ g \equiv \forall a\ b. (a,b) \in edges\ g \longrightarrow (b,a) \in edges\ g$

definition $faceListAt\text{-}len :: graph \Rightarrow bool$ **where**
 $faceListAt\text{-}len\ g \equiv (length\ (faceListAt\ g) = countVertices\ g)$

definition $facesAt\text{-}eq :: graph \Rightarrow bool$ **where**
 $facesAt\text{-}eq\ g \equiv \forall v \in \mathcal{V}\ g. set(facesAt\ g\ v) = \{f. f \in \mathcal{F}\ g \wedge v \in \mathcal{V}\ f\}$

definition $facesAt\text{-}distinct :: graph \Rightarrow bool$ **where**
 $facesAt\text{-}distinct\ g \equiv \forall v \in \mathcal{V}\ g. distinct\ (normFaces\ (facesAt\ g\ v))$

definition $faces\text{-}distinct :: graph \Rightarrow bool$ **where**
 $faces\text{-}distinct\ g \equiv distinct\ (normFaces\ (faces\ g))$

definition $faces\text{-}subset :: graph \Rightarrow bool$ **where**
 $faces\text{-}subset\ g \equiv \forall f \in \mathcal{F}\ g. \mathcal{V}\ f \subseteq \mathcal{V}\ g$

definition $edges\text{-}disj :: graph \Rightarrow bool$ **where**
 $edges\text{-}disj\ g \equiv$
 $\forall f \in \mathcal{F}\ g. \forall f' \in \mathcal{F}\ g. f \neq f' \longrightarrow \mathcal{E}\ f \cap \mathcal{E}\ f' = \{\}$

definition $face\text{-}face\text{-}op :: graph \Rightarrow bool$ **where**
 $face\text{-}face\text{-}op\ g \equiv |faces\ g| \neq 2 \longrightarrow$
 $(\forall f \in \mathcal{F}\ g. \forall f' \in \mathcal{F}\ g. f \neq f' \longrightarrow \mathcal{E}\ f \neq (\mathcal{E}\ f')^{-1})$

definition $one\text{-}final\text{-}but :: graph \Rightarrow (vertex \times vertex)set \Rightarrow bool$ **where**
 $one\text{-}final\text{-}but\ g\ E \equiv$
 $\forall f \in \mathcal{F}\ g. \neg\ final\ f \longrightarrow$
 $(\forall (a,b) \in \mathcal{E}\ f - E. (b,a) : E \vee (\exists f' \in \mathcal{F}\ g. final\ f' \wedge (b,a) \in \mathcal{E}\ f'))$

definition $one\text{-}final :: graph \Rightarrow bool$ **where**
 $one\text{-}final\ g \equiv one\text{-}final\text{-}but\ g\ \{\}$

definition $minGraphProps :: graph \Rightarrow bool$ **where**
 $minGraphProps\ g \equiv minGraphProps'\ g \wedge facesAt\text{-}eq\ g \wedge faceListAt\text{-}len\ g \wedge facesAt\text{-}distinct\ g \wedge faces\text{-}distinct\ g \wedge faces\text{-}subset\ g \wedge edges\text{-}sym\ g \wedge edges\text{-}disj\ g \wedge face\text{-}face\text{-}op\ g$

definition $inv :: graph \Rightarrow bool$ **where**
 $inv\ g \equiv minGraphProps\ g \wedge one\text{-}final\ g \wedge |faces\ g| \geq 2$

lemma $facesAt\text{-}distinctI$:
 $(\bigwedge v. v \in \mathcal{V}\ g \implies distinct\ (normFaces\ (facesAt\ g\ v))) \implies facesAt\text{-}distinct\ g$
by (*simp add: facesAt-distinct-def*)

lemma $minGraphProps2$:

$\text{minGraphProps } g \implies f \in \mathcal{F} g \implies 2 < |\text{vertices } f|$
by (*unfold minGraphProps-def minGraphProps'-def*) *auto*

lemma *mgp-vertices3*:

$\text{minGraphProps } g \implies f \in \mathcal{F} g \implies |\text{vertices } f| \geq 3$
by(*auto dest:minGraphProps2*)

lemma *mgp-vertices-nonempty*:

$\text{minGraphProps } g \implies f \in \mathcal{F} g \implies \text{vertices } f \neq []$
by(*auto dest:minGraphProps2*)

lemma *minGraphProps3*:

$\text{minGraphProps } g \implies f \in \mathcal{F} g \implies \text{distinct } (\text{vertices } f)$
by (*unfold minGraphProps-def minGraphProps'-def*) *auto*

lemma *minGraphProps4*:

$\text{minGraphProps } g \implies (\text{length } (\text{faceListAt } g) = \text{countVertices } g)$
by (*unfold minGraphProps-def faceListAt-len-def*) *simp*

lemma *minGraphProps5*:

$[\text{minGraphProps } g; v : \mathcal{V} g; f \in \text{set } (\text{facesAt } g v)] \implies f \in \mathcal{F} g$
by(*auto simp: facesAt-def facesAt-eq-def minGraphProps-def minGraphProps'-def faceListAt-len-def split:if-split-asm*)

lemma *minGraphProps6*:

$\text{minGraphProps } g \implies v : \mathcal{V} g \implies f \in \text{set } (\text{facesAt } g v) \implies v \in \mathcal{V} f$
by(*auto simp: facesAt-def facesAt-eq-def minGraphProps-def minGraphProps'-def faceListAt-len-def split:if-split-asm*)

lemma *minGraphProps9*:

$\text{minGraphProps } g \implies f \in \mathcal{F} g \implies v \in \mathcal{V} f \implies v \in \mathcal{V} g$
by (*unfold minGraphProps-def faces-subset-def*) *auto*

lemma *minGraphProps7*:

$\text{minGraphProps } g \implies f \in \mathcal{F} g \implies v \in \mathcal{V} f \implies f \in \text{set } (\text{facesAt } g v)$
apply(*frule (2) minGraphProps9*)
by (*unfold minGraphProps-def facesAt-eq-def*) *simp*

lemma *minGraphProps-facesAt-eq*: $\text{minGraphProps } g \implies$

$v \in \mathcal{V} g \implies \text{set } (\text{facesAt } g v) = \{f \in \mathcal{F} g. v \in \mathcal{V} f\}$
by (*simp add: minGraphProps-def facesAt-eq-def*)

lemma *mgp-dist-facesAt[simp]*:

$\text{minGraphProps } g \implies v : \mathcal{V} g \implies \text{distinct } (\text{facesAt } g v)$
by(*auto simp: facesAt-def minGraphProps-def minGraphProps'-def facesAt-distinct-def*)

dest: normFaces-distinct)

lemma *minGraphProps8*:

minGraphProps g \implies *v* : $\mathcal{V} g \implies$ *distinct* (*normFaces* (*facesAt g v*))

by (*auto simp: facesAt-def minGraphProps-def minGraphProps'-def facesAt-distinct-def normFaces-def*)

lemma *minGraphProps8a*:

minGraphProps g \implies *v* $\in \mathcal{V} g \implies$ *distinct* (*normFaces* (*faceListAt g ! v*))

apply (*frule* (1) *minGraphProps8*[**where** *v=v*]) **by** (*simp add: facesAt-def*)

lemma *minGraphProps8a'*: *minGraphProps g* \implies

v < *countVertices g* \implies *distinct* (*normFaces* (*faceListAt g ! v*))

by (*simp add: minGraphProps8a vertices-graph*)

lemma *minGraphProps9'*:

minGraphProps g \implies *f* $\in \mathcal{F} g \implies$ *v* $\in \mathcal{V} f \implies$ *v* < *countVertices g*

by (*simp add: minGraphProps9 in-vertices-graph[symmetric]*)

lemma *minGraphProps10*:

minGraphProps g \implies (*a, b*) \in *edges g* \implies (*b, a*) \in *edges g*

apply (*unfold minGraphProps-def edges-sym-def*)

apply (*elim conjE allE impE*)

by *simp+*

lemma *minGraphProps11*:

minGraphProps g \implies *distinct* (*normFaces* (*faces g*))

by (*unfold minGraphProps-def faces-distinct-def*) *simp*

lemma *minGraphProps11'*:

minGraphProps g \implies *distinct* (*faces g*)

by (*simp add: minGraphProps11 normFaces-distinct*)

lemma *minGraphProps12*:

minGraphProps g \implies *f* $\in \mathcal{F} g \implies$ (*a, b*) $\in \mathcal{E} f \implies$ (*b, a*) $\notin \mathcal{E} f$

apply (*subgoal-tac distinct* (*vertices f*)) **apply** (*simp add: is-nextElem-def*)

apply (*case-tac vertices f = []*)

apply (*drule minGraphProps2*)

apply *simp*

apply *simp*

apply *simp*

apply (*case-tac a = last* (*vertices f*) \wedge *b = hd* (*vertices f*))

apply (*case-tac vertices f*) **apply** *simp*

apply (*case-tac list rule: rev-exhaust*)

apply (*drule minGraphProps2*) **apply** *simp*

apply *simp*

apply (*case-tac ys*)

apply (*drule minGraphProps2*) **apply** *simp* **apply** *simp*

```

apply (simp del: distinct-append distinct.simps)
apply (rule conjI)
apply (rule ccontr) apply (simp del: distinct-append distinct.simps)
apply (drule is-sublist-distinct-prefix) apply simp
apply (simp add: is-prefix-def)
apply simp
apply (rule conjI)
apply (simp add: is-sublist-def) apply (elim exE) apply (intro allI) apply (rule
ccontr)
apply (simp del: distinct-append distinct.simps)
apply (subgoal-tac asa = as @ [a]) apply simp
apply (rule dist-at1) apply assumption apply force apply (rule sym) apply
simp
apply (subgoal-tac is-sublist [a, b] (vertices f))
apply (rule impI) apply (rule ccontr)
apply (simp add: is-sublist-def del: distinct-append distinct.simps)
apply (subgoal-tac last (vertices f) = b ∧ hd (vertices f) = a)
apply (thin-tac a = hd (vertices f)) apply (thin-tac b = last (vertices f)) apply
(elim conjE)
apply (elim exE)
apply (case-tac as)
apply (case-tac bs rule: rev-exhaust) apply (drule minGraphProps2) apply
simp apply simp
apply simp+
apply (rule minGraphProps3) by simp+

```

```

lemma minGraphProps7': minGraphProps g ⇒
  f ∈  $\mathcal{F}$  g ⇒ v ∈  $\mathcal{V}$  f ⇒ f ∈ set (faceListAt g ! v)
apply (frule minGraphProps7) apply assumption+
by (simp add: facesAt-def split: if-split-asm)

```

```

lemma mgp-edges-disj:
  [ minGraphProps g; f ≠ f'; f ∈  $\mathcal{F}$  g; f' ∈  $\mathcal{F}$  g ] ⇒
  uv ∈  $\mathcal{E}$  f ⇒ uv ∉  $\mathcal{E}$  f'
by (simp add: minGraphProps-def edges-disj-def) blast

```

```

lemma one-final-but-antimono:
  one-final-but g E ⇒ E ⊆ E' ⇒ one-final-but g E'
apply (unfold one-final-but-def)
apply blast
done

```

```

lemma one-final-antimono: one-final g ⇒ one-final-but g E
apply (unfold one-final-def one-final-but-def)
apply blast
done

```

lemma *inv-two-faces*: $inv\ g \implies |faces\ g| \geq 2$
by(*simp add:inv-def*)

lemma *inv-mgp[simp]*: $inv\ g \implies minGraphProps\ g$
by(*simp add:inv-def*)

lemma *makeFaceFinal-id[simp]*: $final\ f \implies makeFaceFinal\ f\ g = g$
apply(*cases\ g*)
apply (*simp add:makeFaceFinal-def makeFaceFinalFaceList-def*
setFinal-eq-iff[THEN\ iffD2])
done

lemma *inv-one-finalD'*:
 $\llbracket inv\ g; f \in \mathcal{F}\ g; \neg\ final\ f; (a,b) \in \mathcal{E}\ f \rrbracket \implies$
 $\exists f' \in \mathcal{F}\ g. final\ f' \wedge f' \neq f \wedge (b,a) \in \mathcal{E}\ f'$
apply(*unfold\ inv-def\ one-final-def\ one-final-but-def*)
apply *blast*
done

lemmas *minGraphProps =*
minGraphProps2\ minGraphProps3\ minGraphProps4
minGraphProps5\ minGraphProps6\ minGraphProps7\ minGraphProps8
minGraphProps9

lemma *mgp-no-loop[simp]*:
 $minGraphProps\ g \implies f \in \mathcal{F}\ g \implies v \in \mathcal{V}\ f \implies f \cdot v \neq v$
apply(*frule\ (1)\ mgp-vertices3*)
apply(*frule\ (1)\ minGraphProps3*)
apply(*simp\ add:\ distinct-no-loop1*)
done

lemma *mgp-facesAt-no-loop*:
 $minGraphProps\ g \implies v : \mathcal{V}\ g \implies f \in set\ (facesAt\ g\ v) \implies f \cdot v \neq v$
by(*blast\ dest:mgp-no-loop\ minGraphProps5\ minGraphProps6*)

lemma *edge-pres-faceAt*:
 $\llbracket minGraphProps\ g; u : \mathcal{V}\ g; f \in set\ (facesAt\ g\ u); (u,v) \in \mathcal{E}\ f \rrbracket \implies$
 $f \in set\ (facesAt\ g\ v)$
apply(*auto\ simp:edges-face-eq*)
apply(*rule\ minGraphProps7,\ assumption*)
apply(*blast\ intro:minGraphProps*)
apply(*simp*)
done

lemma *in-facesAt-nextVertex*:
 $minGraphProps\ g \implies v : \mathcal{V}\ g \implies f \in set\ (facesAt\ g\ v) \implies f \in set\ (facesAt\ g\ (f \cdot v))$
apply(*subgoal-tac\ (v,f \cdot v) \in \mathcal{E}\ f*)
apply(*blast\ intro:edge-pres-faceAt*)

by(blast intro: nextVertex-in-edges minGraphProps)

lemma *mgp-edge-face-ex*:

assumes [intro]: minGraphProps g $v : \mathcal{V} g$

and $fv: f \in \text{set}(\text{facesAt } g \ v)$ **and** $uv: (u,v) \in \mathcal{E} f$

shows $\exists f' \in \text{set}(\text{facesAt } g \ v). (v,u) \in \mathcal{E} f'$

proof –

from fv **have** $f \in \mathcal{F} g$ **by**(blast intro:minGraphProps)

with uv **have** $(u,v) \in \mathcal{E} g$ **by**(auto simp:edges-graph-def)

hence $(v,u) \in \mathcal{E} g$ **by**(blast intro:minGraphProps10)

then obtain f' **where** $f': f' \in \mathcal{F} g$ **and** $vu: (v,u) \in \mathcal{E} f'$

by(auto simp:edges-graph-def)

from vu **have** $v \in \mathcal{V} f'$ **by**(auto simp:edges-face-eq)

with f' **have** $f' \in \text{set}(\text{facesAt } g \ v)$ **by**(blast intro:minGraphProps)

with vu **show** ?thesis **by** blast

qed

lemma *nextVertex-in-graph*:

$\text{minGraphProps } g \implies v : \mathcal{V} g \implies f \in \text{set}(\text{facesAt } g \ v) \implies f \cdot v : \mathcal{V} g$

by(blast intro: minGraphProps9 minGraphProps5 minGraphProps6 nextVertex-in-face)

lemma *mgp-nextVertex-face-ex2*:

assumes $\text{mgp}[intro]: \text{minGraphProps } g \ v : \mathcal{V} g$ **and** $f: f \in \text{set}(\text{facesAt } g \ v)$

shows $\exists f' \in \text{set}(\text{facesAt } g \ (f \cdot v)). f' \cdot (f \cdot v) = v$

proof –

from f **have** $(v,f \cdot v) \in \mathcal{E} f$

by(blast intro: nextVertex-in-edges minGraphProps)

with *in-facesAt-nextVertex*[OF $\text{mgp } f$]

mgp-edge-face-ex[OF $\text{mgp}(1)$ *nextVertex-in-graph*[OF $\text{mgp } f$]]

obtain $f' :: \text{face}$ **where** $f' \in \text{set}(\text{facesAt } g \ (f \cdot v))$

and $(f \cdot v,v) \in \mathcal{E} f'$

by(blast)

thus ?thesis **by** (auto simp: edges-face-eq)

qed

lemma *inv-finals-nonempty*: $\text{inv } g \implies \text{finals } g \neq []$

apply(frule *inv-two-faces*)

apply(clarsimp simp:filter-empty-conv *finals-def*)

apply(subgoal-tac $\text{faces } g \neq []$)

prefer 2 **apply** clarsimp

apply(simp add:neq-Nil-conv)

apply clarify

apply(rename-tac $f \ fs$)

apply(case-tac *final* f)

apply simp

apply(frule *mgp-vertices-nonempty*[OF *inv-mgp*])

apply fastforce


```

apply(clarsimp simp:neq-Nil-conv)
apply(rename-tac v vs)
apply(subgoal-tac v ∈ V f)
  prefer 2 apply simp
apply(drule nextVertex-in-edges)
apply(drule inv-one-finalD')
  prefer 2 apply assumption
  apply simp
apply assumption
apply(auto)
done

```

13.3 containsDuplicateEdge

definition

```

containsUnacceptableEdgeSnd' :: (nat ⇒ nat ⇒ bool) ⇒ nat list ⇒ bool where
containsUnacceptableEdgeSnd' N is ≡
  (∃ k < |is| - 2. let i0 = is!k; i1 = is!(k+1); i2 = is!(k+2) in
  N i1 i2 ∧ (i0 < i1) ∧ (i1 < i2))

```

lemma *containsUnacceptableEdgeSnd-eq:*

```

containsUnacceptableEdgeSnd N v is = containsUnacceptableEdgeSnd' N (v#is)

```

proof (*induct is arbitrary: v*)

```

case Nil then show ?case by (simp add: containsUnacceptableEdgeSnd'-def)

```

next

```

case (Cons i is) then show ?case

```

```

  proof (rule-tac iffI)

```

```

    assume vors: containsUnacceptableEdgeSnd N v (i # is)

```

```

    then show containsUnacceptableEdgeSnd' N (v # i # is)

```

```

      apply (cases is) apply simp apply simp

```

```

      apply (simp split: if-split-asm del: containsUnacceptableEdgeSnd.simps)

```

```

      apply (simp add: containsUnacceptableEdgeSnd'-def) apply force

```

```

      apply (subgoal-tac a # list = is) apply (thin-tac is = a # list) apply
(simp add: Cons)

```

```

        apply (simp add: containsUnacceptableEdgeSnd'-def) apply (elim exE)

```

```

        apply (rule exI) apply (subgoal-tac Suc k < |is|) apply (rule conjI) apply

```

assumption by auto

next

```

  assume vors: containsUnacceptableEdgeSnd' N (v # i # is)

```

```

  then show containsUnacceptableEdgeSnd N v (i # is)

```

```

  apply simp apply (cases is) apply (simp add: containsUnacceptableEdgeSnd'-def)

```

```

  apply (simp del: containsUnacceptableEdgeSnd.simps)

```

```

  apply (subgoal-tac a # list = is) apply (thin-tac is = a # list)

```

```

  apply (simp add: Cons)

```

```

  apply (subgoal-tac is = a # list) apply (thin-tac a # list = is)

```

```

  apply (simp add: containsUnacceptableEdgeSnd'-def)

```

```

  apply (elim exE) apply (case-tac k) apply simp apply simp apply (intro
impI exI)

```

```

    apply (rule conjI) apply (elim conjE) apply assumption by auto

```

qed
qed

lemma *containsDuplicateEdge-eq1*:
containsDuplicateEdge g f v is = containsDuplicateEdge' g f v is
apply (*simp add: containsDuplicateEdge-def*)
apply (*cases is*) **apply** (*simp add: containsDuplicateEdge'-def*)
apply *simp*
apply (*case-tac list*) **apply** (*simp add: containsDuplicateEdge'-def*)
apply (*simp add: containsUnacceptableEdgeSnd-eq del: containsUnacceptableEdgeSnd.simps*)
apply (*rule conjI*) **apply** (*simp add: containsDuplicateEdge'-def*)
apply (*rule impI*)
apply (*case-tac a < aa*)
by (*simp-all add: containsDuplicateEdge'-def containsUnacceptableEdgeSnd'-def*)

lemma *containsDuplicateEdge-eq*:
containsDuplicateEdge = containsDuplicateEdge'
apply (*rule ext*)+
by (*simp add: containsDuplicateEdge-eq1*)

declare *Nat.diff-is-0-eq'* [*simp del*]

13.4 *replacefacesAt*

primrec *replacefacesAt2* ::
nat list ⇒ face ⇒ face list ⇒ face list list where
replacefacesAt2 [] f fs F = F |
replacefacesAt2 (n#ns) f fs F =
(if n < |F|
then replacefacesAt2 ns f fs (F [n:=replace f fs (F!n)])
else replacefacesAt2 ns f fs F)

lemma *replacefacesAt-eq[THEN eq-reflection]*:
replacefacesAt ns oldf newfs F = replacefacesAt2 ns oldf newfs F
by (*induct ns arbitrary: F*) (*auto simp add: replacefacesAt-def*)

lemma *replacefacesAt2-notin*:
i ∉ set is ⇒ (replacefacesAt2 is olfF newFs Fss)!i = Fss!i
proof (*induct is arbitrary: Fss*)
case *Nil* **then show** *?case* **by** (*simp*)
next
case (*Cons j js*) **then show** *?case*
by (*cases j < |Fss|*) (*auto*)
qed

lemma *replacefacesAt2-in*:
 $i \in \text{set } is \implies \text{distinct } is \implies i < |Fss| \implies$
 $(\text{replacefacesAt2 } is \text{ olfF newFs Fss})!i = \text{replace olfF newFs (Fss !}i)$
proof (*induct is arbitrary: Fss*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons j js*)
then have $j = i \wedge i \notin \text{set } js \vee i \neq j \wedge i \in \text{set } js$ **by** *auto*
then show *?case*
proof (*elim disjE conjE*)
assume $j = i \ i \notin \text{set } js$ **with** *Cons* **show** *?thesis*
by (*auto simp add: replacefacesAt2-notin*)
next
assume $i \in \text{set } js \ i \neq j$ **with** *Cons* **show** *?thesis* **by** *simp*
qed
qed

lemma *distinct-replacefacesAt21*:
 $i < |Fss| \implies i \in \text{set } is \implies \text{distinct } is \implies \text{distinct (Fss!}i) \implies \text{distinct newFs}$
 \implies
 $\text{set (Fss ! }i) \cap \text{set newFs} \subseteq \{\text{olfF}\} \implies$
 $\text{distinct } ((\text{replacefacesAt2 } is \text{ olfF newFs Fss})! i)$
proof (*induct is*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons j js*)
then have $j = i \wedge i \notin \text{set } js \vee i \neq j \wedge i \in \text{set } js$ **by** *auto*
then show *?case*
proof (*elim disjE conjE*)
assume $j = i \ i \notin \text{set } js$ **with** *Cons* **show** *?thesis*
by (*simp add: replacefacesAt2-notin distinct-replace*)
next
assume $i \in \text{set } js \ i \neq j$ **with** *Cons* **show** *?thesis*
by (*simp add: replacefacesAt2-in distinct-replace*)
qed
qed

lemma *distinct-replacefacesAt22*:
 $i < |Fss| \implies i \notin \text{set } is \implies \text{distinct } is \implies \text{distinct (Fss!}i) \implies \text{distinct newFs}$
 \implies
 $\text{set (Fss ! }i) \cap \text{set newFs} \subseteq \{\text{olfF}\} \implies$
 $\text{distinct } ((\text{replacefacesAt2 } is \text{ olfF newFs Fss})! i)$
proof (*induct is*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons j js*)
then have $i \neq j$ **by** *auto*
with *Cons* **show** *?case*

by (simp add: replacefacesAt2-notin distinct-replace)
qed

lemma *distinct-replacefacesAt2-2*:

$i < |Fss| \implies \text{distinct } is \implies \text{distinct } (Fss!i) \implies \text{distinct } newFs \implies$
 $\text{set } (Fss ! i) \cap \text{set } newFs \subseteq \{oldF\} \implies$
 $\text{distinct } ((\text{replacefacesAt2 } is \text{ oldF } newFs \text{ } Fss)! i)$

by (cases $i \in \text{set } is$)
(auto intro: *distinct-replacefacesAt21 distinct-replacefacesAt22*)

lemma *replacefacesAt2-nth1*:

$k \notin \text{set } ns \implies (\text{replacefacesAt2 } ns \text{ oldf } newfs \text{ } F) ! k = F ! k$

by (induct ns arbitrary: F) auto

lemma *replacefacesAt2-nth1'*: $k \in \text{set } ns \implies k < |F| \implies \text{distinct } ns \implies$

$(\text{replacefacesAt2 } ns \text{ oldf } newfs \text{ } F) ! k = (\text{replace oldf } newfs \text{ } (F!k))$

apply (induct ns arbitrary: F)

apply auto

apply (simp add: *replacefacesAt2-nth1*)

by (case-tac $a = k$) auto

lemma *replacefacesAt2-nth2*: $k < |F| \implies$

$(\text{replacefacesAt2 } [k] \text{ oldf } newfs \text{ } F) ! k = \text{replace oldf } newfs \text{ } (F!k)$

by (auto)

lemma *replacefacesAt2-length[simp]*:

$|\text{replacefacesAt2 } nvs \text{ } f' \text{ } f'' \text{ } vs| = |vs|$

by (induct nvs arbitrary: vs) simp-all

lemma *replacefacesAt2-nth*: $k \in \text{set } ns \implies k < |F| \implies \text{oldf} \notin \text{set } newfs \implies$

$\text{distinct } (F!k) \implies \text{distinct } newfs \implies \text{oldf} \in \text{set } (F!k) \longrightarrow \text{set } newfs \cap \text{set } (F!k)$
 $\subseteq \{\text{oldf}\} \implies$

$(\text{replacefacesAt2 } ns \text{ oldf } newfs \text{ } F) ! k = (\text{replace oldf } newfs \text{ } (F!k))$

proof (induct ns arbitrary: F)

case Nil then show ?case by simp

next

case (Cons $n \ ns$) then show ?case

apply (simp only: *replacefacesAt2.simps*)

apply simp apply (case-tac $n = k$)

apply (simp)

apply (subgoal-tac *replacefacesAt2 ns oldf newfs (F[k := replace oldf newfs (F ! k)]) ! k =*

replace oldf newfs ((F[k := replace oldf newfs (F ! k)]) ! k))

apply simp

apply (case-tac $k \in \text{set } ns$) apply (rule Cons) apply simp+

apply (rule *replace-distinct*) apply simp apply simp

apply simp

apply simp

apply (*simp add: distinct-set-replace*)
apply (*simp add: replacefacesAt2-nth1*)
by *simp*
qed

lemma *replacefacesAt-notin*:
 $i \notin \text{set } is \implies (\text{replacefacesAt } is \text{ oldF newFs Fss})!i = Fss!i$
by (*simp add: replacefacesAt-eq replacefacesAt2-notin*)

lemma *replacefacesAt-in*:
 $i \in \text{set } is \implies \text{distinct } is \implies i < |Fss| \implies$
 $(\text{replacefacesAt } is \text{ oldF newFs Fss})!i = \text{replace oldF newFs } (Fss !i)$
by (*simp add: replacefacesAt-eq replacefacesAt2-in*)

lemma *replacefacesAt-length[simp]*: $|\text{replacefacesAt } nvs \ f' \ [f'] \ vs| = |vs|$
by (*simp add: replacefacesAt-eq*)

lemma *replacefacesAt-nth2*: $k < |F| \implies$
 $(\text{replacefacesAt } [k] \ \text{oldf newfs } F) ! k = \text{replace oldf newfs } (F!k)$
by (*simp add: replacefacesAt-eq replacefacesAt2-nth2*)

lemma *replacefacesAt-nth*: $k \in \text{set } ns \implies k < |F| \implies \text{oldf} \notin \text{set newfs} \implies$
 $\text{distinct } (F!k) \implies \text{distinct newfs} \implies \text{oldf} \in \text{set } (F!k) \longrightarrow \text{set newfs} \cap \text{set } (F!k)$
 $\subseteq \{\text{oldf}\} \implies$
 $(\text{replacefacesAt } ns \ \text{oldf newfs } F) ! k = (\text{replace oldf newfs } (F!k))$
by (*simp add: replacefacesAt-eq replacefacesAt2-nth*)

lemma *replacefacesAt2-5*: $x \in \text{set } (\text{replacefacesAt2 } ns \ \text{oldf newfs } F ! k) \implies x \in$
 $\text{set } (F!k) \vee x \in \text{set newfs}$

proof (*induct ns arbitrary: F*)

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

next

case (*Cons n ns*)

then show *?case*

apply(*simp add: split: if-split-asm*) **apply** (*frule Cons*)

apply (*thin-tac* $\bigwedge F. x \in \text{set } (\text{replacefacesAt2 } ns \ \text{oldf newfs } F ! k) \implies x \in \text{set}$
 $(F ! k) \vee x \in \text{set newfs}$)

apply (*case-tac* $x \in \text{set newfs}$) **apply** *simp* **apply** *simp*

apply (*case-tac* $k = n$) **apply** *simp* **apply** (*frule replace5*) **apply** *simp* **by**

simp

qed

lemma *replacefacesAt-Nil[simp]*: $\text{replacefacesAt } [] \ f \ fs \ F = F$
by (*simp add: replacefacesAt-eq*)

lemma *replacefacesAt-Cons[simp]*:
 $\text{replacefacesAt } (n \# ns) \ f \ fs \ F =$

(if $n < |F|$ then $\text{replacefacesAt } ns \ f \ fs \ (F[n := \text{replace } f \ fs \ (F!n)])$
 else $\text{replacefacesAt } ns \ f \ fs \ F$)
by (*simp add: replacefacesAt-eq*)

lemmas *replacefacesAt-simps = replacefacesAt-Nil replacefacesAt-Cons*

lemma *len-nth-repAt[simp]:*
 $\bigwedge xs. i < |xs| \implies |\text{replacefacesAt } is \ x \ [y] \ xs \ ! \ i| = |xs!i|$
by (*induct is*) (*simp-all add: add:nth-list-update*)

13.5 *normFace*

lemma *minVertex-in: vertices f \neq [] \implies minVertex f \in $\mathcal{V} \ f$*
by (*simp add: minVertex-def*)

lemma *minVertex-eq-if-vertices-eq:*
 $\mathcal{V} \ f = \mathcal{V} \ f' \implies \text{minVertex } f = \text{minVertex } f'$
apply (*cases f*)
apply (*cases f'*)
apply (*rename-tac vs ft vs' ft'*)
apply (*case-tac vs = []*)
apply (*simp add: vertices-face-def minVertex-def*)
apply (*subgoal-tac vs' \neq []*)
prefer 2 apply clarsimp
apply (*simp add: vertices-face-def minVertex-def min-list-conv-Min*
insert-absorb del: Min-insert)
done

lemma *normFace-replace-in:*
 $\text{normFace } a \in \text{set } (\text{normFaces } (\text{replace } \text{oldF } \text{newFs } fs)) \implies$
 $\text{normFace } a \in \text{set } (\text{normFaces } \text{newFs}) \vee \text{normFace } a \in \text{set } (\text{normFaces } fs)$
apply (*induct fs*) **apply** *simp*
apply (*auto simp add: normFaces-def split:if-split-asm*)
done

lemma *distinct-replace-norm:*
 $\text{distinct } (\text{normFaces } fs) \implies \text{distinct } (\text{normFaces } \text{newFs}) \implies$
 $\text{set } (\text{normFaces } fs) \cap \text{set } (\text{normFaces } \text{newFs}) \subseteq \{\}$ $\implies \text{distinct } (\text{normFaces}$
 $(\text{replace } \text{oldF } \text{newFs } fs))$
apply (*induct fs*) **apply** *simp*
apply *simp*
apply (*case-tac a = oldF*) **apply** (*simp add: normFaces-def*) **apply** *blast*
apply (*simp add: normFaces-def*) **apply** (*rule ccontr*)
apply (*subgoal-tac normFace a \in set(normFaces (replace oldF newFs fs))*)

apply (*frule normFace-replace-in*)
by (*simp add: normFaces-def*)⁺

lemma *distinct-replacefacesAt1-norm*:

$i < |Fss| \implies i \in \text{set } is \implies \text{distinct } is \implies \text{distinct } (\text{normFaces } (Fss!i)) \implies$
 $\text{distinct } (\text{normFaces } \text{newFs}) \implies$

$\text{set } (\text{normFaces } (Fss ! i)) \cap \text{set } (\text{normFaces } \text{newFs}) \subseteq \{\} \implies$
 $\text{distinct } (\text{normFaces } ((\text{replacefacesAt } is \text{ oldF } \text{newFs } Fss)! i))$

proof (*induct is*)

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

next

case (*Cons j js*)

then have $j = i \wedge i \notin \text{set } js \vee i \neq j \wedge i \in \text{set } js$ **by** *auto*

then show *?case*

proof (*elim disjE conjE*)

assume $j = i \wedge i \notin \text{set } js$ **with** *Cons* **show** *?thesis*

by (*simp add: replacefacesAt-notin distinct-replace-norm*)

next

assume $i \in \text{set } js \wedge i \neq j$ **with** *Cons* **show** *?thesis*

by (*simp add: replacefacesAt-in distinct-replace-norm*)

qed

qed

lemma *distinct-replacefacesAt2-norm*:

$i < |Fss| \implies i \notin \text{set } is \implies \text{distinct } is \implies \text{distinct } (\text{normFaces } (Fss!i)) \implies$
 $\text{distinct } (\text{normFaces } \text{newFs}) \implies$

$\text{set } (\text{normFaces } (Fss ! i)) \cap \text{set } (\text{normFaces } \text{newFs}) \subseteq \{\} \implies$
 $\text{distinct } (\text{normFaces } ((\text{replacefacesAt } is \text{ oldF } \text{newFs } Fss)! i))$

proof (*induct is*)

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

next

case (*Cons j js*)

then have $i \neq j$ **by** *auto*

with *Cons* **show** *?case*

by (*simp add: replacefacesAt-notin distinct-replace-norm*)

qed

lemma *distinct-replacefacesAt-norm*:

$i < |Fss| \implies \text{distinct } is \implies \text{distinct } (\text{normFaces } (Fss!i)) \implies \text{distinct } (\text{normFaces } \text{newFs}) \implies$

$\text{set } (\text{normFaces } (Fss ! i)) \cap \text{set } (\text{normFaces } \text{newFs}) \subseteq \{\} \implies$
 $\text{distinct } (\text{normFaces } ((\text{replacefacesAt } is \text{ oldF } \text{newFs } Fss)! i))$

by (*cases i \in set is*)

(*auto intro: distinct-replacefacesAt1-norm distinct-replacefacesAt2-norm*)

lemma *normFace-in-cong*:

$\text{vertices } f \neq [] \implies \text{minGraphProps } g \implies \text{normFace } f \in \text{set } (\text{normFaces } (\text{faces } g))$

$\implies \exists f' \in \text{set}(\text{faces } g). f \cong f'$
apply (*simp add: normFace-def normFaces-def*)
apply (*erule imageE*)
apply (*rename-tac f'*)
apply (*rule bezI*)
defer **apply** *assumption*
apply (*simp add: cong-face-def*)
apply (*rule congs-trans*) **apply** (*rule verticesFrom-congs*)
apply (*rule minVertex-in*) **apply** *simp*
apply (*rule congs-sym*) **apply** (*simp add: normFace-def*)
apply (*rule verticesFrom-congs*) **apply** (*rule minVertex-in*)
apply (*subgoal-tac 2 < |vertices f'|*) **apply** *force*
by (*simp add: minGraphProps2*)

lemma *normFace-neq*:

$a \in \mathcal{V} f \implies a \notin \mathcal{V} f' \implies \text{vertices } f' \neq \square \implies \text{normFace } f \neq \text{normFace } f'$
apply (*simp add: normFace-def*)
apply (*subgoal-tac a \in \text{set}(\text{verticesFrom } f (\text{minVertex } f))*)
apply (*subgoal-tac a \notin \text{set}(\text{verticesFrom } f' (\text{minVertex } f'))*) **apply** *force*
apply (*subgoal-tac (\text{vertices } f') \cong (\text{verticesFrom } f' (\text{minVertex } f'))*) **apply** (*simp add: congs-pres-nodes*)
apply (*rule verticesFrom-congs*) **apply** (*rule minVertex-in*) **apply** *simp*
apply (*subgoal-tac (\text{vertices } f) \cong (\text{verticesFrom } f (\text{minVertex } f))*) **apply** (*simp add: congs-pres-nodes*)
apply (*rule verticesFrom-congs*) **apply** (*rule minVertex-in*) **by** *auto*

lemma *split-face-f12-f21-neq-norm*:

$\text{pre-split-face } \text{oldF } \text{ram1 } \text{ram2 } \text{vs} \implies$
 $2 < |\text{vertices } \text{oldF}| \implies 2 < |\text{vertices } f12| \implies 2 < |\text{vertices } f21| \implies$
 $(f12, f21) = \text{split-face } \text{oldF } \text{ram1 } \text{ram2 } \text{vs} \implies \text{normFace } f12 \neq \text{normFace } f21$

proof –

assume *split*: $(f12, f21) = \text{split-face } \text{oldF } \text{ram1 } \text{ram2 } \text{vs}$
 $\text{pre-split-face } \text{oldF } \text{ram1 } \text{ram2 } \text{vs}$
and *minlen*: $2 < |\text{vertices } \text{oldF}| \quad 2 < |\text{vertices } f12| \quad 2 < |\text{vertices } f21|$
from *split* **have** *dist-f12*: $\text{distinct}(\text{vertices } f12)$ **by** (*rule split-face-distinct1*)
from *split* **have** *dist-f21*: $\text{distinct}(\text{vertices } f21)$ **by** (*rule split-face-distinct2*)

from *split* *dist-f12* *dist-f21* *minlen* **show** *?thesis*
apply (*simp add: split-face-def*)
apply (*case-tac between (\text{vertices } \text{oldF}) \text{ram2 } \text{ram1}*)
apply (*case-tac between (\text{vertices } \text{oldF}) \text{ram1 } \text{ram2}*)
apply *simp* **apply** (*subgoal-tac |\text{vertices } \text{oldF}| = 2*)
apply *simp* **apply** (*rule verticesFrom-ram1*)
apply (*subgoal-tac distinct (\text{vertices } \text{oldF})*) **apply** (*erule verticesFrom-length*)
apply (*subgoal-tac \text{ram1} \in \mathcal{V} \text{oldF}*) **apply** *assumption* **apply** (*simp add: pre-split-face-def*) **apply** *simp*
apply (*simp add: pre-split-face-def*)
apply (*rule normFace-neq*)
apply (*subgoal-tac a \in \mathcal{V} (\text{Face } (\text{rev } \text{vs } @ \text{ram1 } \# \text{between } (\text{vertices } \text{oldF}))*)


```

ram1 ram2 @ [ram2]) Nonfinal))
  apply assumption apply simp apply force apply simp
  apply (rule not-sym)
  apply (rule normFace-neq)
  apply (subgoal-tac a ∈ V (Face (ram2 # between (vertices oldF) ram2 ram1
@ ram1 # vs) Nonfinal))
  apply assumption apply simp
  apply (frule verticesFrom-ram1)
  apply (subgoal-tac distinct (verticesFrom oldF ram1)) apply clarsimp
  apply (rule verticesFrom-distinct)
  by (simp add: pre-split-face-def)+
qed

```

lemma *normFace-in*: $f \in \text{set } fs \implies \text{normFace } f \in \text{set } (\text{normFaces } fs)$
by (*simp add: normFaces-def*)

13.6 Invariants of *splitFace*

lemma *splitFace-holds-minGraphProps'*:
 $\text{pre-splitFace } g' v a f' vs \implies \text{minGraphProps}' g' \implies$
 $\text{minGraphProps}' (\text{snd } (\text{snd } (\text{splitFace } g' v a f' vs)))$
apply (*simp add: minGraphProps'-def*)
apply *safe*
apply (*simp add: splitFace-def split-def*)
apply (*case-tac* $f \in \mathcal{F} g'$) **apply** *simp*
apply *safe*
apply (*simp add: split-face-def*) **apply** *safe* **apply** *simp* **apply** (*drule pre-FaceDiv-between1*)
apply *simp*
apply (*frule-tac* *replace1*)
apply *simp-all*
apply (*simp add: split-face-def*) **apply** *safe* **apply** *simp*
apply (*drule pre-FaceDiv-between2*) **apply** *simp*
apply (*drule splitFace-split*)
apply *safe*
apply *simp*
apply (*subgoal-tac* *pre-splitFace* $g' v a f' vs$)
apply (*drule splitFace-distinct2*) **apply** *simp* **+**
apply (*subgoal-tac* *pre-splitFace* $g' v a f' vs$)
apply (*drule splitFace-distinct1*) **+**
by *simp* **+**

lemma *splitFace-holds-faceListAt-len*:
 $\text{pre-splitFace } g' v a f' vs \implies \text{minGraphProps}' g' \implies$
 $\text{faceListAt-len } (\text{snd } (\text{snd } (\text{splitFace } g' v a f' vs)))$
by (*simp add: minGraphProps-def faceListAt-len-def splitFace-def split-def*)

```

lemma splitFace-new-f12:
assumes pre: pre-splitFace g ram1 ram2 oldF newVs
  and props: minGraphProps g
  and spl: (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs
shows f12  $\notin \mathcal{F}$  g
proof (cases newVs)
  case Nil with pre have (ram2, ram1)  $\notin$  edges g
    by (unfold pre-splitFace-def) auto
  moreover from Nil pre
  have (ram2, ram1)  $\in$  edges f12
    apply (rule-tac splitFace-empty-ram2-ram1-in-f12)
    apply (auto simp: Nil[symmetric])
    apply (rule spl)
  done
  ultimately show ?thesis by (auto simp add: edges-graph-def)
next
  case (Cons v vs)
  with pre have v  $\notin \mathcal{V}$  g
    by (auto simp: pre-splitFace-def)
  moreover from Cons spl have v  $\in \mathcal{V}$  f12
    by (simp add: splitFace-f12-new-vertices)
  moreover note props
  ultimately show ?thesis by (auto dest: minGraphProps)
qed

lemma splitFace-new-f12-norm:
assumes pre: pre-splitFace g ram1 ram2 oldF newVs
and props: minGraphProps g
and spl: (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs
shows normFace f12  $\notin$  set (normFaces (faces g))
proof (cases newVs)
  case Nil with pre have (ram2, ram1)  $\notin$  edges g
    by (auto simp: pre-splitFace-def)
  moreover
  from pre spl [symmetric] have dist-f12: distinct (vertices f12)
    apply (drule-tac splitFace-distinct2) by simp
  moreover
  from Nil pre
  have (ram2, ram1)  $\in$  edges f12
    apply (rule-tac splitFace-empty-ram2-ram1-in-f12)
    apply (auto simp: Nil[symmetric])
    apply (rule spl)
  done
  moreover
  with dist-f12 have vertices f12  $\neq []$ 
    apply (simp add: is-nextElem-def) apply (case-tac vertices f12) apply (simp
add: is-sublist-def)
    by simp
  ultimately show ?thesis

```

```

apply (auto simp add: edges-graph-def) apply (frule normFace-in-cong)
  apply (rule props)
  apply assumption
  apply (elim bexE)
  apply (subgoal-tac (ram2, ram1) ∈ edges f') apply simp
  apply (subgoal-tac (vertices f12) ≅ (vertices f')) apply (frule congs-distinct)
  apply (simp add: cong-face-def is-nextElem-congs-eq)+
  done
next
case (Cons v vs)
with pre have  $v \notin \mathcal{V} g$  by (auto simp: pre-splitFace-def)
moreover from Cons spl have  $v \in \mathcal{V} f12$ 
  by (simp add: splitFace-f12-new-vertices)
moreover note props
ultimately show ?thesis
  apply auto
  apply (subgoal-tac (vertices f12) ≠ [])
  apply (frule normFace-in-cong) apply assumption+ apply (erule bexE)
  apply (subgoal-tac  $v \in \mathcal{V} f'$ ) apply (simp add: minGraphProps9)
  apply (simp add: congs-pres-nodes cong-face-def) by auto
qed

```

```

lemma splitFace-new-f21:
assumes pre: pre-splitFace g ram1 ram2 oldF newVs
and props: minGraphProps g
and spl: (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs
shows f21 ∉  $\mathcal{F} g$ 
proof (cases newVs)
  case Nil with pre have (ram1, ram2) ∉ edges g
    by (auto simp: pre-splitFace-def)
  moreover from Nil pre
  have (ram1, ram2) ∈ edges f21
    apply (rule-tac splitFace-empty-ram1-ram2-in-f21)
    apply (auto simp: Nil[symmetric])
    apply (rule spl)
  done
  ultimately show ?thesis by (auto simp add: edges-graph-def)
next
case (Cons v vs)
with pre have  $v \notin \mathcal{V} g$  by (auto simp: pre-splitFace-def)
moreover from Cons spl have  $v \in \mathcal{V} f21$ 
  by (simp add: splitFace-f21-new-vertices)
moreover note props
ultimately show ?thesis by (auto dest: minGraphProps)
qed

```

```

lemma splitFace-new-f21-norm:
assumes pre: pre-splitFace g ram1 ram2 oldF newVs
and props: minGraphProps g

```

```

and spl: (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs
shows normFace f21  $\notin$  set (normFaces (faces g))
proof (cases newVs)
  case Nil with pre have (ram1, ram2)  $\notin$  edges g
    by (auto simp: pre-splitFace-def)
  moreover
  from pre spl [symmetric] have dist-f21: distinct (vertices f21)
    apply (drule-tac splitFace-distinct1) by simp
  moreover
  from Nil pre
  have (ram1, ram2)  $\in$  edges f21
    apply (rule-tac splitFace-empty-ram1-ram2-in-f21)
    apply (auto simp: Nil[symmetric])
    apply (rule spl)
  done
  moreover
  with dist-f21 have vertices f21  $\neq$  []
    apply (simp add: is-nextElem-def) apply (case-tac vertices f21) apply (simp
  add: is-sublist-def)
    by simp
  ultimately show ?thesis apply (auto simp add: edges-graph-def) apply (frule
  normFace-in-cong)
    apply (rule props)
    apply assumption
    apply (elim bexE)
    apply (subgoal-tac (ram1, ram2)  $\in$  edges f') apply simp
    apply (subgoal-tac (vertices f21)  $\cong$  (vertices f')) apply (frule congs-distinct)
    apply (simp add: cong-face-def is-nextElem-congs-eq)+
  done
next
  case (Cons v vs)
  with pre have v  $\notin$   $\mathcal{V}$  g by (auto simp: pre-splitFace-def)
  moreover from Cons spl have v  $\in$   $\mathcal{V}$  f21
    by (simp add: splitFace-f21-new-vertices)
  moreover note props
  ultimately show ?thesis apply auto
    apply (subgoal-tac (vertices f21)  $\neq$  [])
    apply (frule normFace-in-cong) apply assumption+ apply (erule bexE)
    apply (subgoal-tac v  $\in$   $\mathcal{V}$  f') apply (simp add: minGraphProps9)
    apply (simp add: congs-pres-nodes cong-face-def) by auto
qed

lemma splitFace-f21-oldF-neq:
  pre-splitFace g ram1 ram2 oldF newVs  $\implies$ 
  minGraphProps g  $\implies$ 
  (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs  $\implies$ 
  oldF  $\neq$  f21
by (frule splitFace-new-f21) (auto)

```

lemma *splitFace-f12-oldF-neq*:
pre-splitFace g ram1 ram2 oldF newVs \implies
minGraphProps g \implies
(f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs \implies
oldF \neq *f12*
by (*frule splitFace-new-f12*) (*auto*)

lemma *splitFace-f12-f21-neq-norm*:
pre-splitFace g ram1 ram2 oldF vs \implies *minGraphProps g* \implies
(f12, f21, newGraph) = splitFace g ram1 ram2 oldF vs \implies
normFace f12 \neq *normFace f21*
apply (*subgoal-tac minGraphProps' newGraph*)
apply (*subgoal-tac f12* \in \mathcal{F} *newGraph* \wedge *f21* \in \mathcal{F} *newGraph*)
apply (*subgoal-tac pre-split-face oldF ram1 ram2 vs*)
apply (*frule split-face-f12-f21-neq-norm*) **apply** (*rule minGraphProps2*) **apply**
simp **apply**(*erule pre-splitFace-oldF*)
apply (*subgoal-tac 2* $<$ $|$ *vertices f12* $|$) **apply** *assumption* **apply** (*force simp*:
minGraphProps'-def)
apply (*subgoal-tac 2* $<$ $|$ *vertices f21* $|$) **apply** *assumption* **apply** (*force simp*:
minGraphProps'-def)
apply (*simp add: splitFace-def split-def*)
apply *simp*
apply *force*
apply (*simp add: splitFace-def split-def*)
apply (*rule disjI2*)
apply (*erule replace3[OF pre-splitFace-oldF]*)
apply *simp*
apply (*frule splitFace-holds-minGraphProps'*) **apply** (*simp add: minGraphProps-def*
minGraphProps'-def)
by (*simp add: splitFace-def split-def*)

lemma *set-faces-splitFace*:
 \llbracket *minGraphProps g*; *f* \in \mathcal{F} *g*; *pre-splitFace g v1 v2 f vs*;
(f1, f2, g') = splitFace g v1 v2 f vs \rrbracket
 \implies \mathcal{F} *g'* = $\{f1, f2\} \cup (\mathcal{F}$ *g* - $\{f\})$
apply(*frule minGraphProps11'*)
apply(*blast dest:splitFace-new-f21 splitFace-new-f12*
splitFace-faces-1 splitFace-delete-oldF)
done

declare *minGraphProps8 minGraphProps8a minGraphProps8a'* [*intro*]

lemma *splitFace-holds-facesAt-distinct*:
assumes *pre: pre-splitFace g v w f* [*countVertices g..<countVertices g + n*]
and *mgp: minGraphProps g*
shows *facesAt-distinct (snd (snd (splitFace g v w f* [*countVertices g..<countVertices*

```

g + n]))))
proof –
  define ws where ws = [countVertices g..<countVertices g + n]
  define f21 where f21 = snd (split-face f v w ws)
  with pre ws-def have dist-f21: distinct (vertices f21) by (auto intro: split-face-distinct2)
  define f12 where f12 = fst (split-face f v w ws)
  with pre ws-def have dist-f12: distinct (vertices f12) by (auto intro: split-face-distinct1)
  define vs1 where vs1 = between (vertices f) v w
  define vs2 where vs2 = between (vertices f) w v
  define g' where g' = snd (snd (splitFace g v w f [countVertices g..<countVertices
g + n]))
  from f12-def f21-def ws-def g'-def
  have fdg: (f12, f21, g') = splitFace g v w f [countVertices g..<countVertices g +
n]
    by (simp add: splitFace-def split-def)
  from pre mgp fdg have new-f12: f12  $\notin$   $\mathcal{F}$  g
    apply (rule-tac splitFace-new-f12) by simp-all
  from pre mgp fdg have new-f21: f21  $\notin$   $\mathcal{F}$  g
    apply (rule-tac splitFace-new-f21) by simp-all
  from pre mgp fdg have new-f12-norm: normFace f12  $\notin$  set (normFaces (faces
g))
    apply (rule-tac splitFace-new-f12-norm) by simp-all
  from pre mgp fdg have new-f21-norm: normFace f21  $\notin$  set (normFaces (faces
g))
    apply (rule-tac splitFace-new-f21-norm) by simp-all

have facesAt-distinct g'
proof (rule facesAt-distinctI)
  fix x assume x: x  $\in$   $\mathcal{V}$  g'
  show distinct (normFaces (facesAt g' x))
    proof –
      from mgp pre have a: v < |faceListAt g| w < |faceListAt g|
        apply (unfold pre-splitFace-def)
        apply (simp-all add: minGraphProps4)
        by (auto intro: minGraphProps9')
      then show ?thesis
        proof (cases x = w)
          case True
            moreover with pre have v  $\neq$  w
              by (unfold pre-splitFace-def) simp
            moreover note a x pre mgp
            ultimately show ?thesis
              apply –
                apply (unfold pre-splitFace-def)
                apply (unfold g'-def splitFace-def facesAt-def)
                apply (simp add: split-def nth-append)
                apply (rule distinct-replace-norm)
                apply (rule distinct-replacefacesAt-norm)

```

```

    apply simp
    apply (rule between-distinct)
    apply simp
    apply (rule distinct-replacefacesAt-norm)
      apply assumption
      apply (rule between-distinct)
      apply simp
      apply (rule minGraphProps8a') apply assumption+ apply (simp
add: minGraphProps4)
      apply (simp add: normFaces-def)

simp
apply (subgoal-tac set (faceListAt g ! w) = {f ∈ ℱ g. w ∈ ℳ f}) apply

apply (subgoal-tac set (normFaces (faces g)) ∩ {normFace f12} = {})
apply (simp add: f12-def ws-def normFaces-def) apply blast
apply (simp add: new-f12-norm)

apply (frule minGraphProps-facesAt-eq)
apply (subgoal-tac w ∈ ℳ g) apply assumption
apply (rule minGraphProps9) apply assumption apply blast apply

simp
apply (simp add: facesAt-def split: if-split-asm)

apply (simp add: normFaces-def)

apply (subgoal-tac w ∉ set (between (vertices f) v w))
apply (simp add: replacefacesAt-notin)
apply (subgoal-tac set (faceListAt g ! w) = {f ∈ ℱ g. w ∈ ℳ f})
apply (subgoal-tac set (normFaces (faces g)) ∩ {normFace f21} = {})
apply (simp add: f21-def ws-def normFaces-def) apply blast
apply (simp add: new-f21-norm)

apply (frule minGraphProps-facesAt-eq)
apply (subgoal-tac w ∈ ℳ g) apply assumption
apply (rule minGraphProps9) apply assumption apply blast apply

simp
apply (simp add: facesAt-def minGraphProps4 vertices-graph)
apply (rule between-not-r2) apply simp

apply (simp add: normFaces-def) apply (rule splitFace-f12-f21-neq-norm)
apply (rule pre) apply simp
apply (subgoal-tac (f12, f21, g') = splitFace g v w f [countVertices
g.<countVertices g + n])
apply (simp add: f12-def f21-def g'-def ws-def)
apply (rule fdg)

apply (subgoal-tac w ∉ set (between (vertices f) w v))
apply (simp add: replacefacesAt-notin)
apply (subgoal-tac w ∉ set (between (vertices f) v w))

```

```

    apply (simp add: replacefacesAt-notin)
    apply (subgoal-tac set (faceListAt g ! w) = {f ∈ ℱ g. w ∈ ℳ f})
    apply (subgoal-tac set (normFaces (faces g)) ∩ {normFace f12, normFace
f21} = {})
      apply (simp add: f12-def f21-def ws-def normFaces-def) apply blast
      apply (simp add: new-f21-norm new-f12-norm)
      apply (frule minGraphProps-facesAt-eq)
      apply (subgoal-tac w ∈ ℳ g) apply assumption
      apply (rule minGraphProps9) apply assumption apply blast apply
simp
      apply (simp add: facesAt-def minGraphProps4 vertices-graph)
      apply (rule between-not-r2) apply simp
      apply (rule between-not-r1) by simp
next
from pre have vw-neq: v ≠ w
  by (unfold pre-splitFace-def) simp
case False then show ?thesis
  proof (cases x = v)
  case True
    with a x pre mgp vw-neq
    show ?thesis
      apply -
      apply (unfold pre-splitFace-def)
      apply (unfold g'-def splitFace-def facesAt-def)
      apply (simp add: split-def nth-append)
      apply (rule distinct-replace-norm)
      apply (rule distinct-replacefacesAt-norm)
      apply simp
      apply (rule between-distinct)
      apply simp
      apply (rule distinct-replacefacesAt-norm)
      apply assumption
      apply (rule between-distinct)
      apply simp
      apply (rule minGraphProps8a) apply assumption+ apply (simp
add: minGraphProps4 vertices-graph)

      apply (simp add: normFaces-def)

      apply (subgoal-tac set (faceListAt g ! v) = {f ∈ ℱ g. v ∈ ℳ f})
      apply (subgoal-tac set (normFaces (faces g)) ∩ {normFace f12}
= {})
        apply (simp add: f12-def ws-def normFaces-def) apply blast
        apply (simp add: new-f12-norm)

      apply (frule minGraphProps-facesAt-eq)
      apply (subgoal-tac v ∈ ℳ g) apply assumption
      apply (rule minGraphProps9) apply assumption apply blast
apply simp

```



```

    apply (simp add: facesAt-def split: if-split-asm)

    apply (simp add: normFaces-def)

    apply (subgoal-tac v ∉ set (between (vertices f) v w))
    apply (simp add: replacefacesAt-notin)
    apply (subgoal-tac set (faceListAt g ! v) = {f ∈ ℱ g. v ∈ ℳ f})
    apply (subgoal-tac set (normFaces (faces g)) ∩ {normFace f21})
= {}

    apply (simp add: f21-def ws-def normFaces-def) apply blast
    apply (simp add: new-f21-norm)

    apply (frule minGraphProps-facesAt-eq)
    apply (subgoal-tac v ∈ ℳ g) apply assumption
    apply (rule minGraphProps9) apply assumption apply blast
apply simp
    apply (simp add: facesAt-def split: if-split-asm)
    apply (rule between-not-r1) apply simp
    apply (simp add: normFaces-def) apply (rule not-sym)
    apply (rule splitFace-f12-f21-neq-norm) apply (rule pre) apply simp
    apply (subgoal-tac (f12, f21, g') = splitFace g v w f [countVertices
g..<countVertices g + n])
    apply (simp add: f12-def f21-def ws-def g'-def) apply (rule fdg)

    apply (subgoal-tac v ∉ set (between (vertices f) w v))
    apply (simp add: replacefacesAt-notin)
    apply (subgoal-tac v ∉ set (between (vertices f) v w))
    apply (simp add: replacefacesAt-notin)
    apply (subgoal-tac set (faceListAt g ! v) = {f ∈ ℱ g. v ∈ ℳ f})
    apply (subgoal-tac set (normFaces (faces g)) ∩ {normFace
f21,normFace f12} = {})
    apply (simp add: f12-def f21-def ws-def normFaces-def) apply
blast

    apply (simp add: new-f21-norm new-f12-norm)
    apply (subgoal-tac set (normFaces (faces g)) ∩ {normFace f21} =
{})

    apply (simp add: new-f21-norm)
    apply (frule minGraphProps-facesAt-eq)
    apply (subgoal-tac v ∈ ℳ g) apply assumption
    apply (rule minGraphProps9) apply assumption apply blast
apply simp
    apply (simp add: facesAt-def minGraphProps4 vertices-graph)
    apply (simp add: new-f21-norm)
    apply (rule between-not-r1) apply simp
    apply (rule between-not-r2) by simp
next
assume xw-neq: x ≠ w
case False
with a x pre mgp vw-neq xw-neq

```

```

show ?thesis
  apply -
  apply (unfold pre-splitFace-def g'-def splitFace-def facesAt-def)
  apply (simp add: split-def nth-append)
  apply (case-tac x < |faceListAt g|)
  apply simp
  apply (subgoal-tac x ∈ V g)
  apply (rule distinct-replacefacesAt-norm)
    apply simp
    apply (rule between-distinct)
    apply simp
  apply (rule distinct-replacefacesAt-norm) apply assumption
    apply (rule between-distinct)
    apply simp
  apply (rule minGraphProps8a) apply assumption apply (simp
add: minGraphProps4)

    apply (simp add: normFaces-def)

  apply (subgoal-tac set (faceListAt g ! x) = {f ∈ F g. x ∈ V f})
  apply (subgoal-tac set (normFaces (faces g)) ∩ {normFace f12})
= {}

    apply (simp add: f12-def ws-def normFaces-def) apply blast
  apply (simp add: new-f12-norm)

  apply (frule minGraphProps-facesAt-eq) apply assumption
  apply (simp add: facesAt-def split: if-split-asm)
  apply (simp add: normFaces-def)

  apply (case-tac x ∉ set (between (vertices f) v w))
  apply (simp add: replacefacesAt-notin)
  apply (subgoal-tac set (faceListAt g ! x) = {f ∈ F g. x ∈ V f})
  apply (subgoal-tac set (normFaces (faces g)) ∩ {normFace f21})
= {}

    apply (simp add: f21-def ws-def normFaces-def) apply blast
  apply (simp add: new-f21-norm)

  apply (frule minGraphProps-facesAt-eq) apply assumption
  apply (simp add: facesAt-def split: if-split-asm)
  apply (simp add: normFaces-def)
  apply (drule replacefacesAt-nth) apply assumption
    apply (subgoal-tac f ∉ set [fst (split-face f v w [countVertices
g..<countVertices g + n])])
    apply assumption apply simp
    apply (rule splitFace-f12-oldF-neq)
    apply (subgoal-tac pre-splitFace g v w f [countVertices
g..<countVertices g + n])
    apply assumption apply (simp add: pre) apply assumption+
  apply (simp add: splitFace-def split-def)

```

```

      apply (rule normFaces-distinct)
      apply (rule minGraphProps8a) apply assumption apply (simp
add: minGraphProps4 vertices-graph)
      apply (simp add: normFaces-def)
      apply (rule impI) apply simp
      apply (subgoal-tac set (faceListAt g ! x) = {f ∈ ℱ g. x ∈ V f})
      apply (subgoal-tac ℱ g ∩ {f12} = {})
      apply (simp add: f12-def ws-def)
      apply (simp add: new-f12)
      apply (frule minGraphProps-facesAt-eq)
      apply (subgoal-tac x ∈ V g) apply assumption
      apply (simp add: minGraphProps4 vertices-graph)
      apply (simp add: facesAt-def minGraphProps4 vertices-graph)
      apply (frule replacefacesAt-nth) apply assumption

      apply (subgoal-tac f ∉ set [fst (split-face f v w [countVertices
g..<countVertices g + n])])
      apply assumption apply simp apply (rule splitFace-f12-oldF-neq)
      apply (subgoal-tac pre-splitFace g v w f [countVertices
g..<countVertices g + n]) apply assumption
      apply (simp add: pre) apply assumption apply (simp add:
splitFace-def split-def)
      apply (rule normFaces-distinct)
      apply (rule minGraphProps8a') apply assumption apply (simp
add: minGraphProps4)
      apply simp
      apply (rule impI) apply simp
      apply (subgoal-tac set (faceListAt g ! x) = {f ∈ ℱ g. x ∈ V f})
      apply (subgoal-tac ℱ g ∩ {f12} = {})
      apply (simp add: f12-def ws-def)
      apply (simp add: new-f12)
      apply (frule minGraphProps-facesAt-eq)
      apply (subgoal-tac x ∈ V g) apply assumption
      apply (simp add: minGraphProps4 vertices-graph)
      apply (simp add: facesAt-def minGraphProps4 vertices-graph)
      apply (simp add: f12-def [symmetric] f21-def [symmetric] ws-def
[symmetric])
      apply (subgoal-tac normFace f21 ∉ set (normFaces (replace f [f12]
(faceListAt g ! x))))
      apply (simp add: normFaces-def)
      apply (rule ccontr) apply simp
      apply (frule normFace-replace-in)
      apply (subgoal-tac normFace f12 ≠ normFace f21)
      apply (subgoal-tac normFace f21 ∉ set (normFaces (faceListAt g
! x)))

      apply (simp add: normFaces-def)
      apply (rule ccontr) apply simp
      apply (subgoal-tac normFace f21 ∉ set (normFaces (facesAt g x)))
      apply (simp add: facesAt-def)

```

```

      apply (subgoal-tac normFace f21 ∉ set (normFaces (faces g)))
    apply (frule minGraphProps-facesAt-eq)
      apply (subgoal-tac x ∈ V g) apply assumption apply (simp
add: minGraphProps4 vertices-graph)
      apply (simp add: normFaces-def) apply (rule ccontr) apply
simp
      apply blast
      apply (rule new-f21-norm)
      apply (rule splitFace-f12-f21-neq-norm) apply (rule pre) apply
simp apply (rule fdg)
      apply (simp add: minGraphProps4 vertices-graph)

    apply (simp add: normFaces-def)
    apply (subgoal-tac (x - |faceListAt g|) < n) apply simp
      apply (rule splitFace-f12-f21-neq-norm) apply (rule pre) apply
simp
      apply (simp add: f12-def [symmetric] f21-def [symmetric] ws-def
[symmetric]) apply (simp add: ws-def) apply (rule fdg)
      by (simp add: minGraphProps4)
    qed
  qed
  qed
  qed
  then show ?thesis by (simp add: g'-def)
qed

```

lemma *splitFace-holds-facesAt-eq*:

```

assumes pre-F: pre-splitFace g' v a f' [countVertices g'..<countVertices g' + n]
and mgp: minGraphProps g'
and g'': g'' = (snd (snd (splitFace g' v a f' [countVertices g'..<countVertices g' +
n])))
shows facesAt-eq g''
proof -
  have [0..<countVertices g''] = [0..<countVertices g' + n]
    apply (simp add: g'') by (simp add: splitFace-def split-def)
  hence vg'': vertices g'' = [0..<countVertices g' + n] by (simp add: vertices-graph)

```

```

define ws where ws = [countVertices g'..<countVertices g' + n]
define f21 where f21 = snd (split-face f' v a ws)
define f12 where f12 = fst (split-face f' v a ws)
define vs1 where vs1 = between (vertices f') v a
define vs2 where vs2 = between (vertices f') a v
from ws-def [symmetric] f21-def [symmetric] f12-def [symmetric] g'' have fdg:
(f12, f21, g'') = splitFace g' v a f' ws
  by (simp add: splitFace-def split-def)
from pre-F have pre-F': pre-splitFace g' v a f' ws apply (unfold pre-splitFace-def
ws-def) by force

```

```

from pre-F' mgp fdg have f'-f21: f' ≠ f21 apply (rule-tac splitFace-f21-oldF-neq)
apply assumption by simp+
from pre-F' mgp fdg have f'-f12: f' ≠ f12 apply (rule-tac splitFace-f12-oldF-neq)
apply assumption by simp+

from f12-def vs1-def have vert-f12: vertices f12 = rev ws @ v # vs1 @ [a] by
(simp add: split-face-def)
from f21-def vs2-def have vert-f21: vertices f21 = a # vs2 @ v # ws by (simp
add: split-face-def)
from vs1-def vs2-def pre-F have vertFrom-f': verticesFrom f' v =
v # vs1 @ a # vs2 apply simp
apply (rule-tac verticesFrom-ram1) by (rule pre-splitFace-pre-split-face)
from vs1-def vs2-def pre-F vertFrom-f' have vert-f':  $\mathcal{V} f' = set vs1 \cup set vs2$ 
 $\cup \{a, v\}$ 
apply (subgoal-tac (vertices f')  $\cong$  (verticesFrom f' v)) apply (drule congs-pres-nodes)
apply (simp add: congs-pres-nodes) apply blast
apply (rule verticesFrom-congs) by (simp only: pre-splitFace-def)
from pre-F have dist-vertFrom-f': distinct (verticesFrom f' v) apply (rule-tac
verticesFrom-distinct)
by (simp only: pre-splitFace-def)+
then have vs1-vs2-empty: set vs1  $\cap$  set vs2 =  $\{\}$  by (simp add: vertFrom-f')

from ws-def f21-def f12-def have faces g'' = (replace f' [f21] (faces g')) @ [f12]
apply (simp add: g'') by (simp add: splitFace-def split-def)

from mgp have dist-all:  $\bigwedge x. x \in \mathcal{V} g' \implies distinct (faceListAt g' ! x)$ 
apply (rule-tac normFaces-distinct)
by (simp add: minGraphProps-def facesAt-distinct-def facesAt-def)

from mgp have fla: |faceListAt g'| = countVertices g'
by (simp add: minGraphProps-def faceListAt-len-def)

from ws-def [symmetric] f21-def [symmetric] f12-def [symmetric]
vs1-def [symmetric] vs2-def [symmetric] pre-F mgp vert-f'
show ?thesis
apply (simp add: g'')
apply (unfold splitFace-def facesAt-eq-def facesAt-def)
apply (rule ballI)
apply (simp only: split-def Let-def)
apply (simp only: snd-conv)
apply (rule equalityI)
apply (rule subsetI)
apply (simp only: faceListAt.simps vertices-graph.simps split-if-split-asm)
apply (case-tac v < |faceListAt g'|  $\wedge$  a < |faceListAt g'|)
apply (simp only: nth-append split: if-split-asm)
apply (case-tac va < |faceListAt g'|)
apply (subgoal-tac va  $\in \mathcal{V} g'$ )
apply (subgoal-tac distinct vs1  $\wedge$  distinct vs2  $\wedge$ 

```

$v \notin \text{set } vs1 \wedge v \notin \text{set } vs2 \wedge a \notin \text{set } vs1 \wedge a \notin \text{set } vs2 \wedge a \neq v \wedge v \neq a \wedge$
 $\text{set } vs1 \cap \text{set } vs2 = \{\}$)
apply (case-tac a = va)
apply (simp add:replacefacesAt-nth2 replacefacesAt-notin)
apply (subgoal-tac distinct (faceListAt g' ! va))
apply (subgoal-tac distinct (faces g'))
apply (simp add: replace6)
apply (case-tac x = f12) **apply** (simp add: vert-f12) **apply** simp
apply (case-tac x = f') **apply** (simp add: vert-f21) **apply** (simp)
apply (case-tac x = f21) **apply** (simp add: vert-f21) **apply** (simp)
apply (rule conjI)
apply (rule minGraphProps5) **apply** assumption **apply** assumption **apply**
(fastforce simp: facesAt-def)
apply (rule minGraphProps6) **apply** assumption **apply** assumption **apply**
(simp add: facesAt-def)
apply (rule minGraphProps11') **apply** simp
apply (subgoal-tac distinct (facesAt g' va)) **apply** (simp add: facesAt-def)
apply (rule normFaces-distinct) **apply** (rule minGraphProps8) **apply** simp
apply simp **apply** simp
apply (case-tac v = va)
apply (simp add:replacefacesAt-nth2 replacefacesAt-notin)
apply (subgoal-tac distinct (faceListAt g' ! va))
apply (subgoal-tac distinct (faces g'))
apply (simp add: replace6)
apply (case-tac x = f12) **apply** (simp add: vert-f12) **apply** simp
apply (case-tac x = f') **apply** (simp add: vert-f21) **apply** simp
apply (case-tac x = f21) **apply** (simp add: vert-f21) **apply** simp
apply (rule conjI)
apply (rule minGraphProps5) **apply** assumption **apply** assumption **apply**
(fastforce simp: facesAt-def)
apply (rule minGraphProps6) **apply** assumption **apply** assumption **ap-**
ply(fastforce simp: facesAt-def)
apply (rule minGraphProps11') **apply** simp
apply (subgoal-tac distinct (facesAt g' va)) **apply** (simp add: facesAt-def)
apply (rule normFaces-distinct) **apply** (rule minGraphProps8) **apply** simp
apply simp **apply** simp
apply (case-tac va ∈ set vs1)
apply (subgoal-tac va ∉ set vs2)
apply (simp add:replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in)
apply (subgoal-tac distinct (faceListAt g' ! va))
apply (subgoal-tac distinct (faces g'))
apply (simp add: replace6)
apply (case-tac x = f12) **apply** (simp add: vert-f12) **apply** simp
apply (case-tac x = f') **apply** (simp add: vert-f21) **apply** simp
apply (rule conjI)
apply (rule disjI2)
apply (rule minGraphProps5) **apply** assumption **apply** assumption
apply (fastforce simp: facesAt-def)
apply (rule minGraphProps6) **apply** assumption **apply** assumption **apply**

(fastforce simp: facesAt-def)
apply (rule *minGraphProps11'*) **apply** *simp*
apply (subgoal-tac *distinct (facesAt g' va)*) **apply** (simp add: *facesAt-def*)
apply (rule *normFaces-distinct*) **apply** (rule *minGraphProps8*) **apply**
assumption **apply** *assumption*
apply *blast*
apply (case-tac *va ∈ set vs2*)
apply (simp add: *replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in*)
apply (subgoal-tac *distinct (faceListAt g' ! va)*)
apply (subgoal-tac *distinct (faces g')*)
apply (simp add: *replace6*)
apply (case-tac *x = f21*) **apply** (simp add: *vert-f21*) **apply** *simp*
apply (case-tac *x = f'*) **apply** (simp add: *vert-f21*) **apply** *simp*
apply (rule *conjI*)
apply (rule *disjI2*)
apply (rule *minGraphProps5*) **apply** *assumption* **apply** *assumption* **apply**
(fastforce simp: facesAt-def)
apply (rule *minGraphProps6*) **apply** *assumption* **apply** *assumption* **apply**
(fastforce simp: facesAt-def)
apply (rule *minGraphProps11'*) **apply** *simp*
apply (subgoal-tac *distinct (facesAt g' va)*) **apply** (simp add: *facesAt-def*)
apply (rule *normFaces-distinct*) **apply** (rule *minGraphProps8*) **apply** *as-*
sumption **apply** *assumption*

apply (simp add: *replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in*)
apply (subgoal-tac *distinct (faceListAt g' ! va)*)
apply (subgoal-tac *distinct (faces g')*)
apply (simp add: *replace6*)
apply (case-tac *x = f'*)
apply (subgoal-tac *va ∈ V f'*) **apply** *simp*
apply (rule *minGraphProps6*) **apply** *simp* **apply** (simp add: *fla*)
apply (simp add: *facesAt-def*)
apply *simp*
apply (rule *conjI*)
apply (rule *disjI2*) **apply** (rule *disjI2*)
apply (rule *minGraphProps5*) **apply** *assumption* **apply** *assumption* **apply**
(fastforce simp: facesAt-def)
apply (rule *minGraphProps6*) **apply** *assumption* **apply** *assumption* **ap-**
ply *(fastforce simp: facesAt-def)*
apply (rule *minGraphProps11'*) **apply** *assumption*
apply (subgoal-tac *distinct (facesAt g' va)*) **apply** (simp add: *facesAt-def*)
apply (rule *normFaces-distinct*) **apply** (rule *minGraphProps8*) **apply** *as-*
sumption **apply** *assumption* **apply** *simp*
apply (subgoal-tac *distinct (vertices f12) ∧ distinct (vertices f21)*)
apply (simp add: *vert-f12 vert-f21*)
apply (rule *vs1-vs2-empty*)
apply (subgoal-tac *pre-split-face f' v a ws*)
apply (simp add: *f12-def f21-def split-face-distinct1' split-face-distinct2'*)
apply *simp*

apply (*simp add: vertices-graph fla*)

apply *simp*
apply (*subgoal-tac distinct (faces g')*)
apply (*simp add: replace6*)
apply (*thin-tac [countVertices g'..<countVertices g' + n] = ws*)
apply (*subgoal-tac (va - |faceListAt g'|) < |ws|*) **apply** *simp* **apply** (*rule conjI*) **apply** *blast*
apply (*subgoal-tac va ∈ set ws*)
apply (*case-tac x = f12*) **apply** (*simp add: vert-f12*) **apply** (*simp add: vert-f21*)
apply (*simp add: ws-def fla*)
apply (*simp add: ws-def fla*)
apply (*rule minGraphProps11'*) **apply** *assumption*
apply (*subgoal-tac v ∈ V g' ∧ a ∈ V g'*)
apply (*simp only: fla in-vertices-graph*)
apply (*subgoal-tac f' ∈ F g'*)
apply (*subgoal-tac v ∈ V f' ∧ a ∈ V f'*) **apply** (*simp only: minGraphProps9*)
apply *force*
apply (*subgoal-tac pre-split-face f' v a ws*) **apply** (*simp only: pre-split-face-def*)
apply *force*
apply (*rule pre-splitFace-pre-split-face*) **apply** *assumption*
apply (*simp only: pre-splitFace-def*)

apply (*rule subsetI*)
apply (*case-tac v < |faceListAt g'| ∧ a < |faceListAt g'|*)
apply (*case-tac va < |faceListAt g'|*)
apply (*subgoal-tac va ∈ V g'*)
apply (*subgoal-tac distinct vs1 ∧ distinct vs2 ∧ v ∉ set vs1 ∧ v ∉ set vs2 ∧ a ∉ set vs1 ∧ a ∉ set vs2 ∧ a ≠ v ∧ v ≠ a ∧ set vs1 ∩ set vs2 = {}*)
apply (*simp del: replacefacesAt-simps add: nth-append*)
apply (*case-tac a = va*)
apply (*simp add: replacefacesAt-nth2 replacefacesAt-notin*)
apply (*subgoal-tac distinct (faceListAt g' ! va)*)
apply (*subgoal-tac distinct (faces g')*)
apply (*simp add: replace6*)
apply (*case-tac x = f12*) **apply** *simp* **apply** (*rule disjI1*) **apply** (*rule minGraphProps7'*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac x = f21*) **apply** *simp* **apply** (*rule disjI1*) **apply** (*rule minGraphProps7'*) **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp* **apply** (*rule minGraphProps7'*) **apply** *simp* **apply** *simp* **apply** *simp*
simp
apply (*rule minGraphProps11'*) **apply** *simp*
apply (*rule normFaces-distinct*) **apply** (*rule minGraphProps8a*) **apply** *simp*
apply *assumption*
apply *simp*
apply (*case-tac v = va*)


```

apply (simp add:replacefacesAt-nth2 replacefacesAt-notin)
apply (subgoal-tac distinct (faceListAt g' ! va))
apply (subgoal-tac distinct (faces g'))
apply (simp add: replace6)
apply (case-tac x = f12) apply simp apply (rule disjI1) apply (rule
minGraphProps7') apply simp apply simp apply simp
apply (case-tac x = f21) apply simp apply (rule disjI1) apply (rule
minGraphProps7') apply simp apply simp apply simp
apply simp apply (rule minGraphProps7') apply simp apply simp apply
simp
apply (rule minGraphProps11') apply simp apply (rule normFaces-distinct)
apply (rule minGraphProps8a) apply simp apply simp apply (case-tac va ∈
set vs1)
apply (subgoal-tac va ∉ set vs2)
apply (simp add:replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in)
apply (subgoal-tac distinct (faceListAt g' ! va))
apply (subgoal-tac distinct (faces g'))
apply (simp add: replace6)
apply (case-tac x = f12) apply simp apply (rule disjI1) apply (rule
minGraphProps7') apply simp apply simp apply simp
apply (case-tac x = f')
apply (subgoal-tac f' ≠ f21) apply simp apply (rule splitFace-f21-oldF-neq)
apply (rule pre-F')
apply simp
apply (rule fdg)
apply (case-tac x = f21) apply (simp add: vert-f21 fla) apply (thin-tac
[countVertices g'..<countVertices g' + n] = ws)
apply (simp add: ws-def)
apply simp apply (rule minGraphProps7') apply simp apply simp apply
simp
apply (rule minGraphProps11') apply simp
apply (rule normFaces-distinct) apply (rule minGraphProps8a) apply simp
apply simp
apply blast
apply (case-tac va ∈ set vs2)
apply (simp add:replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in)
apply (subgoal-tac distinct (faceListAt g' ! va))
apply (subgoal-tac distinct (faces g'))
apply (simp add: replace6)
apply (case-tac x = f21) apply simp apply (rule disjI1) apply (rule
minGraphProps7') apply simp apply simp apply simp
apply (case-tac x = f')
apply (subgoal-tac f' ≠ f12) apply simp apply (rule splitFace-f12-oldF-neq)
apply (rule pre-F') apply simp apply (rule fdg)
apply (case-tac x = f12) apply (simp add: vert-f12 fla) apply (thin-tac
[countVertices g'..<countVertices g' + n] = ws)
apply (simp add: ws-def)
apply simp apply (rule minGraphProps7') apply simp apply simp apply
simp

```

```

    apply (rule minGraphProps11') apply simp apply (rule normFaces-distinct)
  apply (rule minGraphProps8a) apply simp apply simp
    apply (simp add: replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in)
    apply (subgoal-tac distinct (faces g'))
    apply (simp add: replace6)
    apply (rule minGraphProps7') apply simp
      apply (case-tac x = f21) apply (simp add: vert-f21) apply (thin-tac
[countVertices g'..<countVertices g' + n] = ws)
        apply (simp add: ws-def vertices-graph)
        apply (case-tac x = f12) apply (simp add: vert-f12) apply (thin-tac
[countVertices g'..<countVertices g' + n] = ws)
          apply (simp add: ws-def vertices-graph)
          apply simp
          apply simp
        apply (rule minGraphProps11') apply simp
      apply (subgoal-tac distinct (vertices f12) ∧ distinct (vertices f21))
      apply (simp add: vert-f12 vert-f21)
      apply (rule vs1-vs2-empty)
      apply (subgoal-tac pre-split-face f' v a ws)
      apply (simp add: f12-def f21-def split-face-distinct1' split-face-distinct2')
      apply (simp add: pre-splitFace-pre-split-face[OF pre-F'])
      apply (simp add: vertices-graph fla)
      apply (simp add: nth-append del:replacefacesAt-simps)
      apply (subgoal-tac distinct (faces g'))
      apply (simp add: replace6)
      apply (thin-tac [countVertices g'..<countVertices g' + n] = ws)
      apply (subgoal-tac (va - |faceListAt g'|) < |ws|) apply simp
      apply (rule ccontr) apply simp
      apply (case-tac x = f') apply simp apply simp
      apply (subgoal-tac va ∈ V g') apply (simp add: fla vertices-graph)
      apply (rule minGraphProps9) apply simp apply force
      apply (simp add: fla) apply (metis minGraphProps9')
      apply (simp add: ws-def fla)
      apply (rule minGraphProps11') apply simp
      apply (subgoal-tac v ∈ V g' ∧ a ∈ V g')
      apply (simp only: fla in-vertices-graph)
      apply (subgoal-tac f' ∈ F g')
      apply (subgoal-tac v ∈ V f' ∧ a ∈ V f') apply (simp only: minGraphProps9)
      apply force
    by force
  qed

```

lemma *splitFace-holds-faces-subset*:

assumes *pre-F*: *pre-splitFace g' v a f'* [*countVertices g'..<countVertices g' + n*]

and *mgp*: *minGraphProps g'*

shows *faces-subset* (*snd* (*snd* (*splitFace g' v a f'* [*countVertices g'..<countVertices g' + n*])))

proof –

define *g''* where *g''* = (*snd* (*snd* (*splitFace g' v a f'* [*countVertices g'..<countVertices*

```

g' + n]))))
define ws where ws = [countVertices g'..<countVertices g' + n]
define f21 where f21 = snd (split-face f' v a ws)
define f12 where f12 = fst (split-face f' v a ws)
define vs1 where vs1 = between (vertices f') v a
define vs2 where vs2 = between (vertices f') a v
from ws-def [symmetric] f21-def [symmetric] f12-def [symmetric] g''-def
  have fdg: (f12, f21, g'') = splitFace g' v a f' ws
  by (simp add: splitFace-def split-def)
from pre-F have pre-F': pre-splitFace g' v a f' ws apply (unfold pre-splitFace-def
ws-def) by force

from f12-def vs1-def have vert-f12: vertices f12 = rev ws @ v # vs1 @ [a] by
(simp add: split-face-def)
from f21-def vs2-def have vert-f21: vertices f21 = a # vs2 @ v # ws by (simp
add: split-face-def)
from vs1-def vs2-def pre-F have vertFrom-f': verticesFrom f' v =
  v # vs1 @ a # vs2 apply simp
apply (rule-tac verticesFrom-ram1) by (rule pre-splitFace-pre-split-face)
from vs1-def vs2-def pre-F vertFrom-f' have vert-f':  $\mathcal{V}$  f' = set vs1  $\cup$  set vs2
 $\cup$  {a,v}
apply (subgoal-tac (vertices f')  $\cong$  (verticesFrom f' v)) apply (drule congs-pres-nodes)
apply (simp add: congs-pres-nodes) apply blast
apply (rule verticesFrom-congs) by (simp only: pre-splitFace-def)

from ws-def f21-def f12-def have faces:faces g'' = (replace f' [f21] (faces g'))
@ [f12]
apply (simp add: g''-def) by (simp add: splitFace-def split-def)

from ws-def have vertices:vertices g'' = vertices g' @ ws by (simp add: g''-def)

from ws-def [symmetric] f21-def [symmetric] f12-def [symmetric]
  vs1-def [symmetric] vs2-def [symmetric] pre-F mgp g''-def [symmetric] show
?thesis
apply (simp add: faces-subset-def) apply (rule ballI) apply (simp add: faces
vertices)
apply (subgoal-tac  $\mathcal{V}$  f'  $\subseteq$   $\mathcal{V}$  g')
apply (case-tac f = f12) apply (simp add: vert-f12 vert-f') apply force
apply simp apply (drule replace5)
apply (case-tac f = f21) apply (simp add: vert-f21 vert-f') apply force
apply simp apply (rule subsetI) apply (frule minGraphProps9) apply as-
sumption+ apply simp
apply (rule subsetI) apply (rule minGraphProps9) by auto
qed

```

lemma *splitFace-holds-edges-sym*:
assumes *pre-F*: *pre-splitFace* *g'* *v* *a* *f'* *ws*
and *mgp*: *minGraphProps* *g'*

shows *edges-sym* (*snd* (*snd* (*splitFace* *g'* *v* *a* *f' ws*)))
proof –
define *g''* **where** *g''* = (*snd* (*snd* (*splitFace* *g'* *v* *a* *f' ws*)))
define *f21* **where** *f21* = *snd* (*split-face* *f'* *v* *a* *ws*)
define *f12* **where** *f12* = *fst* (*split-face* *f'* *v* *a* *ws*)
define *vs1* **where** *vs1* = *between* (*vertices* *f'*) *v* *a*
define *vs2* **where** *vs2* = *between* (*vertices* *f'*) *a* *v*
from *f21-def* [*symmetric*] *f12-def* [*symmetric*] *g''-def*
have *fdg*: (*f12*, *f21*, *g''*) = *splitFace* *g'* *v* *a* *f' ws*
by (*simp* *add*: *splitFace-def* *split-def*)
from *pre-F* **have** *pre-F'*: *pre-splitFace* *g'* *v* *a* *f' ws* **apply** (*unfold* *pre-splitFace-def*)
by *force*

from *f21-def* *f12-def* **have** *faces:faces* *g''* = (*replace* *f'* [*f21*] (*faces* *g'*)) @ [*f12*]
apply (*simp* *add*: *g''-def*) **by** (*simp* *add*: *splitFace-def* *split-def*)

from *f12-def* *f21-def* **have** *split*: (*f12*, *f21*) = *split-face* *f'* *v* *a* *ws* **by** *simp*

from *pre-F* *mgp* *g''-def* [*symmetric*] *split* **show** ?*thesis*
apply (*simp* *add*: *edges-sym-def* *edges-graph-def* *f21-def* [*symmetric*] *f12-def* [*symmetric*])
vs1-def [*symmetric*] *vs2-def* [*symmetric*]
apply (*intro* *allI* *impI*) **apply** (*elim* *bexE*) **apply** (*simp* *add*: *faces*)
apply (*case-tac* *x* = *f12* \vee *x* = *f21*)
apply (*subgoal-tac* (*aa,b*) \in *edges* *f'* \vee ((*b,aa*) \in (*edges* *f12* \cup *edges* *f21*) \wedge (*aa,b*) \in (*edges* *f12* \cup *edges* *f21*))) **apply** *simp*
apply (*case-tac* (*aa*, *b*) \in *edges* *f'*)
apply (*subgoal-tac* (*b,aa*) \in *edges* *g'*)
apply (*simp* *add*: *edges-graph-def*) **apply** (*elim* *bexE*) **apply** (*rule* *disjI2*)
apply (*rule* *beXI*)
apply *simp*
apply (*subgoal-tac* *xa* \neq *f'*) **apply** (*rule* *replace4*) **apply** *simp* **apply** *force*
apply (*drule* *minGraphProps12*) **apply** *simp* **apply** *simp*
apply (*rule* *ccontr*) **apply** *simp*
apply (*rule* *minGraphProps10*) **apply** *simp* **apply** (*simp* *add*: *edges-graph-def*)
apply (*rule* *beXI*) **apply** (*thin-tac* (*aa*, *b*) \in *edges* *x*) **apply** *simp*
apply *simp*
apply *simp*
apply (*case-tac* (*b*, *aa*) \in *edges* *f12*) **apply** *simp* **apply** *simp*
apply (*case-tac* (*b*, *aa*) \in *edges* *f21*) **apply** (*rule* *beXI*)
apply *simp*
apply (*rule* *replace3*) **apply** *simp*
apply *simp*
apply *simp*
apply (*subgoal-tac*
((*aa,b*) \in *edges* *f'* \vee ((*b,aa*) \in (*edges* *f12* \cup *edges* *f21*) \wedge (*aa,b*) \in (*edges* *f12* \cup *edges* *f21*))) = ((*aa,b*) \in *edges* *f12* \vee (*aa,b*) \in *edges* *f21*)) **apply** *force*
apply (*rule* *sym*) **apply** *simp*
apply (*rule* *split-face-edges-f12-f21-sym*) **apply** (*erule* *pre-splitFace-oldF*)

```

  apply (subgoal-tac pre-split-face f' v a ws) apply assumption apply simp
  apply (rule split)
  apply simp
  apply (subgoal-tac distinct (faces g')) apply (simp add: replace6)
  apply (case-tac x = f') apply simp apply simp
  apply (subgoal-tac (b,aa) ∈ edges g')
  apply (simp add: edges-graph-def) apply (elim bexE)
  apply (case-tac xa = f')
  apply simp apply (frule split-face-edges-or) apply simp apply simp
  apply (case-tac (b, aa) ∈ edges f12) apply simp apply simp
  apply (rule bexE) apply (thin-tac (b, aa) ∈ edges f')
  apply simp
  apply (rule replace3) apply simp apply simp
  apply (rule disjI2) apply (rule bexE) apply simp
  apply (rule replace4) apply simp
  apply force
  apply (rule minGraphProps10) apply simp
  apply (simp add: edges-graph-def)
  apply (rule bexE) apply simp apply simp
  apply (rule minGraphProps11') by simp
qed

```

```

lemma splitFace-holds-faces-distinct:
  assumes pre-F: pre-splitFace g' v a f' [countVertices g'..<countVertices g' + n]
  and mgp: minGraphProps g'
  shows faces-distinct (snd (snd (splitFace g' v a f' [countVertices g'..<countVertices
  g' + n])))
  proof -
    define g'' where g'' = snd (snd (splitFace g' v a f' [countVertices g'..<countVertices
  g' + n]))
    define ws where ws ≡ [countVertices g'..<countVertices g' + n]
    define f21 where f21 = snd (split-face f' v a ws)
    define f12 where f12 = fst (split-face f' v a ws)
    define vs1 where vs1 = between (vertices f') v a
    define vs2 where vs2 = between (vertices f') a v
    from ws-def [symmetric] f21-def [symmetric] f12-def [symmetric] g''-def
    have fdg: (f12, f21, g'') = splitFace g' v a f' ws
      by (simp add: splitFace-def split-def)
    from pre-F have pre-F': pre-splitFace g' v a f' ws apply (unfold pre-splitFace-def
  ws-def) by force

    from ws-def f21-def f12-def have faces:faces g'' = (replace f' [f21] (faces g'))
  @ [f12]
    apply (simp add: g''-def) by (simp add: splitFace-def split-def)
    from f12-def f21-def have split: (f12, f21) = split-face f' v a ws by simp

    from ws-def [symmetric] pre-F mgp g''-def [symmetric] split show ?thesis
    apply (simp add: faces-distinct-def faces)

```

```

apply (subgoal-tac distinct (normFaces (replace f' [f21] (faces g'))))
apply (simp add: normFaces-def)
apply safe
apply (subgoal-tac distinct (faces g')) apply (simp add: replace6)
apply (case-tac x = f') apply simp
apply (subgoal-tac f' ≠ f21) apply simp
apply (rule splitFace-f21-oldF-neq)
apply (rule pre-F') apply simp
apply (rule fdg)
apply simp
apply (case-tac x = f21) apply simp
apply (subgoal-tac normFace f12 ≠ normFace f21) apply simp
apply (rule splitFace-f12-f21-neq-norm) apply force apply simp
apply (simp add: fdg) apply (rule fdg)
apply simp
apply (subgoal-tac normFace f12 ∉ set (normFaces (faces g')))
apply (simp add: normFaces-def)
apply (rule splitFace-new-f12-norm) apply (rule pre-F') apply simp
apply (rule fdg)
apply (rule minGraphProps11') apply simp
apply (rule distinct-replace-norm) apply (rule minGraphProps11) apply simp
apply (simp add: normFaces-def)
apply (subgoal-tac normFace f21 ∉ set (normFaces (faces g')))
apply (simp add: normFaces-def)
apply (rule splitFace-new-f21-norm) apply (rule pre-F') apply simp
by (rule fdg)
qed

```

lemma *help*:

shows $xs \neq [] \implies x \notin \text{set } xs \implies x \neq \text{hd } xs$ **and**
 $xs \neq [] \implies x \notin \text{set } xs \implies x \neq \text{last } xs$ **and**
 $xs \neq [] \implies x \notin \text{set } xs \implies \text{hd } xs \neq x$ **and**
 $xs \neq [] \implies x \notin \text{set } xs \implies \text{last } xs \neq x$

by(*auto*)

lemma *split-face-edge-disj*:

$\llbracket \text{pre-split-face } f \ a \ b \ vs; (f_1, f_2) = \text{split-face } f \ a \ b \ vs; |\text{vertices } f| \geq 3;$
 $vs = [] \longrightarrow (a,b) \notin \text{edges } f \wedge (b,a) \notin \text{edges } f \rrbracket$
 $\implies \mathcal{E} \ f_1 \cap \mathcal{E} \ f_2 = \{\}$

apply(*frule pre-split-face-p-between[THEN between-inter-empty]*)
apply(*unfold pre-split-face-def*)
apply *clarify*
apply(*subgoal-tac* $\bigwedge x \ y. x \in \text{set } vs \implies y \in \mathcal{V} \ f \implies x \neq y$)
prefer 2 **apply** *blast*
apply(*subgoal-tac* $\bigwedge x \ y. x \in \text{set } vs \implies y \in \mathcal{V} \ f \implies y \neq x$)
prefer 2 **apply** *blast*
apply(*subgoal-tac* $a \notin \text{set } vs$)
prefer 2 **apply** *blast*

```

apply(subgoal-tac  $b \notin \text{set } vs$ )
  prefer 2 apply blast
apply(subgoal-tac  $\text{distinct}(vs @ a \# \text{between}(\text{vertices } f) a b @ [b])$ )
  prefer 2 apply(simp add:between-not-r1 between-not-r2 between-distinct)
  apply(blast dest:inbetween-inset)
apply(subgoal-tac  $\text{distinct}(b \# \text{between}(\text{vertices } f) b a @ a \# \text{rev } vs)$ )
  prefer 2 apply(simp add:between-not-r1 between-not-r2 between-distinct)
  apply(blast dest:inbetween-inset)
apply(subgoal-tac  $vs = [] \implies \text{between}(\text{vertices } f) a b \neq []$ )
  prefer 2 apply clarsimp apply(frule (4) is-nextElem-between-empty')apply blast
apply(subgoal-tac  $vs = [] \implies \text{between}(\text{vertices } f) b a \neq []$ )
  prefer 2 apply clarsimp
apply(frule (3) is-nextElem-between-empty')apply simp apply blast
apply(subgoal-tac  $vs \neq [] \implies \text{hd } vs \notin \mathcal{V} f$ )
  prefer 2 apply(drule hd-in-set) apply blast
apply(subgoal-tac  $vs \neq [] \implies \text{last } vs \notin \mathcal{V} f$ )
  prefer 2 apply(drule last-in-set) apply blast
apply(subgoal-tac  $\bigwedge u v. \text{between}(\text{vertices } f) u v \neq [] \implies \text{hd}(\text{between}(\text{vertices } f) u v) \in \mathcal{V} f$ )
  prefer 2 apply(drule hd-in-set)apply(drule inbetween-inset) apply blast
apply(subgoal-tac  $\bigwedge u v. \text{between}(\text{vertices } f) u v \neq [] \implies \text{last}(\text{between}(\text{vertices } f) u v) \in \mathcal{V} f$ )
  prefer 2 apply(drule last-in-set) apply(drule inbetween-inset) apply blast
apply(simp add:split-face-def edges-conv-Edges Edges-append Edges-Cons
  last-rev notinset-notinEdge1 notinset-notinEdge2 notinset-notinbetween
  between-not-r1 between-not-r2 help Edges-rev-disj disj-sets-disj-Edges
  Int-Un-distrib Int-Un-distrib2)
apply clarify
apply(rule conjI)
  apply clarify
  apply(rule disj-sets-disj-Edges)
  apply simp
  apply(blast dest:inbetween-inset)
apply clarify
apply(rule conjI)
  apply clarify
  apply(rule disj-sets-disj-Edges)
  apply simp
  apply(blast dest:inbetween-inset)
apply clarify
apply(rule conjI)
  apply(rule disj-sets-disj-Edges)
  apply simp
  apply(blast dest:inbetween-inset)
apply(rule disj-sets-disj-Edges)
apply(blast dest:inbetween-inset)
done

```

lemma *splitFace-edge-disj*:
assumes *mgp*: *minGraphProps* *g* **and** *pre*: *pre-splitFace* *g* *u* *v* *f* *vs*
and *FDG*: $(f_1, f_2, g') = \text{splitFace } g \ u \ v \ f \ vs$
shows *edges-disj* *g'*
proof –
from *mgp* **have** *disj*: *edges-disj* *g* **by** (*simp* *add*: *minGraphProps-def*)
have $\mathcal{V} \ g \cap \text{set } vs = \{\}$ **using** *pre*
by (*simp* *add*: *pre-splitFace-def*)
hence *gvs*: $\forall f \in \mathcal{F} \ g. \ \mathcal{V} \ f \cap \text{set } vs = \{\}$
by (*clarsimp* *simp*: *edges-graph-def* *edges-face-def*)
(blast *dest*: *minGraphProps9*[*OF* *mgp*])
have *f*: $f \in \mathcal{F} \ g$ **by** (*rule* *pre-splitFace-oldF*[*OF* *pre*])
note *split-face* = *splitFace-split-face*[*OF* *f* *FDG*]
note *pre-split-face* = *pre-splitFace-pre-split-face*[*OF* *pre*]
have $\mathcal{E} \ f_1 \cap \mathcal{E} \ f_2 = \{\}$
apply(*rule* *split-face-edge-disj*[*OF* *pre-split-face* *split-face* *mgp-vertices3*[*OF* *mgp* *f*]])
using *pre*
apply(*simp* *add*: *pre-splitFace-def* *del*: *pre-splitFace-oldF*)
apply *clarify*
by(*simp*)
moreover
{ **fix** *f'* **assume** *f'*: $f' \in \mathcal{F} \ g \ f' \neq f$
have $(\mathcal{E} \ f_1 \cup \mathcal{E} \ f_2) \cap \mathcal{E} \ f' = \{\}$
proof *cases*
assume *vs*: $vs = []$
have $(u, v) \notin \mathcal{E} \ g \wedge (v, u) \notin \mathcal{E} \ g$ **using** *pre* *vs*
by (*simp* *add*: *pre-splitFace-def*)
with *split-face-edges-f12-f21-vs*[*OF* *pre-split-face*[*simplified* *vs*] *split-face*[*simplified* *vs*]]
show *?thesis* **using** *f* *f'* *disj*
by (*simp* *add*: *is-duplicateEdge-def* *edges-graph-def* *edges-disj-def*)
next
assume *vs*: $vs \neq []$
have *f12*: $vs \neq [] \implies \mathcal{E} \ f_1 \cup \mathcal{E} \ f_2 \subseteq$
 $\mathcal{E} \ f \cup UNIV \times \text{set } vs \cup \text{set } vs \times UNIV$
using *split-face-edges-f12-f21*[*OF* *pre-split-face* *split-face*]
by *simp* (*fastforce* *dest*: *in-Edges-in-set*)
have $\bigwedge x \ y. (y, x) \in \mathcal{E} \ f' \implies x \notin \text{set } vs \wedge y \notin \text{set } vs$
using *f'* *gvs* **by** (*blast* *dest*: *in-edges-in-vertices*)
then **show** *?thesis* **using** *f* *f'* *f12* *disj* *vs*
by (*simp* *add*: *edges-graph-def* *edges-disj-def*) *blast*
qed }
ultimately **show** *?thesis* **using** *disj*
by (*simp* *add*: *edges-disj-def* *set-faces-splitFace*[*OF* *mgp* *f* *pre* *FDG*])
blast
qed

lemma *splitFace-edges-disj2*:


```

minGraphProps g  $\implies$  pre-splitFace g u v f vs
 $\implies$  edges-disj(snd(snd(splitFace g u v f vs)))
apply(subgoal-tac pre-splitFace g u v f vs)
prefer 2 apply(simp)
by(drule (1) splitFace-edge-disj[where f1 = fst(splitFace g u v f vs) and f2 =
fst(snd(splitFace g u v f vs))], auto)

```

```

lemma vertices-conv-Union-edges2:
distinct(vertices f)  $\implies$   $\mathcal{V}(f::\text{face}) = (\bigcup_{(a,b)\in\mathcal{E}} f. \{b\})$ 
apply auto
apply(fast intro: prevVertex-in-edges)
done

```

```

lemma splitFace-face-face-op:
assumes mgp: minGraphProps g and pre: pre-splitFace g u v f vs
and fdg: (f1,f2,g') = splitFace g u v f vs
shows face-face-op g'
proof -
have f12: (f1, f2) = split-face f u v vs
and Fg':  $\mathcal{F} g' = \{f_1\} \cup \text{set}(\text{replace } f [f_2] (\text{faces } g))$ 
and g': g' = snd (snd (splitFace g u v f vs)) using fdg
by(auto simp add:splitFace-def split-def)
have f1: f1 = fst(split-face f u v vs) and f2: f2 = snd(split-face f u v vs)
using f12[symmetric] by simp-all
note distF = minGraphProps11 '[OF mgp]
note pre-split = pre-splitFace-pre-split-face[OF pre]
note distf1 = split-face-distinct1[OF f12 pre-split]
note distf2 = split-face-distinct2[OF f12 pre-split]
from pre have nf:  $\neg$  final f and fg: f  $\in \mathcal{F} g$  and nuv: u  $\neq$  v
and uinf: u  $\in \mathcal{V} f$  and vinf: v  $\in \mathcal{V} f$ 
and distf: distinct(vertices f) and new:  $\mathcal{V} g \cap \text{set } vs = \{\}$ 
by(unfold pre-splitFace-def, simp)+
let ?fuv = between (vertices f) u v and ?fvu = between (vertices f) v u
have E1:  $\mathcal{E} f_1 = \text{Edges } (v \# \text{rev } vs @ [u]) \cup \text{Edges } (u \# ?fuv @ [v])$ 
using f1 by(simp add:edges-split-face1[OF pre-split])
have E2:  $\mathcal{E} f_2 = \text{Edges } (u \# vs @ [v]) \cup \text{Edges } (v \# ?fvu @ [u])$ 
using f2 by(simp add:edges-split-face2[OF pre-split])
have vf1: vertices f1 = rev vs @ u # ?fuv @ [v]
using f1 by(simp add:split-face-def)
have vf2: vertices f2 = [v] @ ?fvu @ u # vs
using f2 by(simp add:split-face-def)
have V1:  $\mathcal{V} f_1 = \{u,v\} \cup \text{set}(?fuv) \cup \text{set}(vs)$  using vf1 by auto
have V2:  $\mathcal{V} f_2 = \{u,v\} \cup \text{set}(?fvu) \cup \text{set}(vs)$  using vf2 by auto
have 2: (v,u)  $\in \mathcal{E} f_1 \wedge (u,v) \in \mathcal{E} f_2 \wedge vs = [] \vee$ 
 $(\exists v \in \mathcal{V} f_1 \cap \mathcal{V} f_2. v \notin \mathcal{V} g)$ 
using E1 E2 V1 V2 new by(cases vs)(simp-all add:Edges-Cons)
have  $\mathcal{V} f_1 \neq \mathcal{V} f_2$ 
proof cases

```

```

assume A: ?fvu = []
have ?fuw ≠ []
proof
  assume ?fuw = []
  with A have  $\mathcal{E} f = \{(v,u),(u,v)\}$ 
    using edges-conv-Un-Edges[OF distf uinf vinf nuw]
    by(simp add:Edges-Cons)
  hence  $\mathcal{V} f = \{u,v\}$  by(simp add:vertices-conv-Union-edges)
  hence  $\text{card}(\mathcal{V} f) \leq 2$  by(simp add:card-insert-if)
  thus False
    using mgp-vertices3[OF mgp fg] by(simp add:distinct-card[OF distf])
qed
moreover have  $\text{set } ?fuw \cap \text{set } vs = \{\}$ 
  using new minGraphProps9[OF mgp fg inbetween-inset] by blast
moreover have  $\{u,v\} \cap \text{set } ?fuw = \{\}$ 
  using between-not-r1[OF distf] between-not-r2[OF distf] by blast
ultimately show ?thesis using V1 V2 A by (auto simp:neq-Nil-conv)
next
assume ?fvu ≠ []
moreover have  $\{u,v\} \cap \text{set } ?fvu = \{\}$ 
  using between-not-r1[OF distf] between-not-r2[OF distf] by blast
moreover have  $\text{set } ?fuw \cap \text{set } ?fvu = \{\}$ 
  by(simp add:pre-between-def between-inter-empty distf uinf vinf nuw)
moreover have  $\text{set } ?fvu \cap \text{set } vs = \{\}$ 
  using new minGraphProps9[OF mgp fg inbetween-inset] by blast
ultimately show ?thesis using V1 V2 by (auto simp:neq-Nil-conv)
qed
have C12:  $\mathcal{E} f_1 \neq (\mathcal{E} f_2)^{-1}$ 
proof
  assume A:  $\mathcal{E} f_1 = (\mathcal{E} f_2)^{-1}$ 
  show False
  proof -
    have  $\mathcal{V} f_1 = (\bigcup (a,b) \in \mathcal{E} f_1. \{a\})$ 
      by(rule vertices-conv-Union-edges)
    also have  $\dots = (\bigcup (b,a) \in \mathcal{E} f_2. \{a\})$  by(auto simp:A)
    also have  $\dots = \mathcal{V} f_2$ 
      by(rule vertices-conv-Union-edges2[OF distf2, symmetric])
    finally show False using  $\langle \mathcal{V} f_1 \neq \mathcal{V} f_2 \rangle$  by blast
  qed
qed
qed
{ fix h :: face assume hg:  $h \in \mathcal{F} g$ 
  have  $\mathcal{E} h \neq (\mathcal{E} f_1)^{-1} \wedge \mathcal{E} h \neq (\mathcal{E} f_2)^{-1}$  using 2
  proof
    assume  $(v,u) \in \mathcal{E} f_1 \wedge (u,v) \in \mathcal{E} f_2 \wedge vs = []$ 
    moreover hence  $(u,v) \notin \mathcal{E} g$ 
      using pre by(unfold pre-splitFace-def) simp
    moreover hence  $(v,u) \notin \mathcal{E} g$  by(blast intro:minGraphProps10[OF mgp])
    ultimately show ?thesis using hg by(simp add:edges-graph-def) blast
  next

```

```

assume  $\exists x \in \mathcal{V} f_1 \cap \mathcal{V} f_2. x \notin \mathcal{V} g$ 
then obtain  $x$  where  $x \in \mathcal{V} f_1$  and  $x \in \mathcal{V} f_2$  and  $x \notin \mathcal{V} g$ 
  by blast
obtain  $y$  where  $(x,y) \in \mathcal{E} f_1$  using  $\langle x \in \mathcal{V} f_1 \rangle$ 
  by(auto simp:vertices-conv-Union-edges)
moreover obtain  $z$  where  $(x,z) \in \mathcal{E} f_2$  using  $\langle x \in \mathcal{V} f_2 \rangle$ 
  by(auto simp:vertices-conv-Union-edges)
moreover have  $\neg(\exists y. (y,x) \in \mathcal{E} h)$ 
  using  $\langle x \notin \mathcal{V} g \rangle$  minGraphProps9[OF mgp hg]
  by(blast dest:in-edges-in-vertices)
ultimately show ?thesis by blast
qed
}
note  $Cg12 = this$ 
show ?thesis
proof cases
  assume  $2: |\text{faces } g| = 2$ 
  with  $fg$  obtain  $f'$  where  $Fg: \mathcal{F} g = \{f,f'\}$ 
  by(fastforce simp: eval-nat-numeral length-Suc-conv)
  moreover hence  $f \neq f'$  using  $2$  distinct-card[OF distF] by auto
  ultimately have  $Fg': \mathcal{F} g' = \{f_1,f_2,f'\}$ 
  using set-faces-splitFace[OF mgp fg pre fdg] by blast
  show ?thesis using  $Fg'$   $C12$   $Cg12$   $Fg$ 
  by(fastforce simp:face-face-op-def)
next
  assume  $|\text{faces } g| \neq 2$ 
  hence  $E: \bigwedge f f'. f \in \mathcal{F} g \implies f' \in \mathcal{F} g \implies f \neq f' \implies \mathcal{E} f \neq (\mathcal{E} f')^{-1}$ 
  using mgp by(simp add:minGraphProps-def face-face-op-def)
  thus ?thesis using set-faces-splitFace[OF mgp fg pre fdg]  $C12$   $Cg12$ 
  by(fastforce simp:face-face-op-def)
qed
qed

lemma splitFace-face-face-op2:
  minGraphProps g  $\implies$  pre-splitFace g u v f vs
   $\implies$  face-face-op(snd(snd(splitFace g u v f vs)))
apply(subgoal-tac pre-splitFace g u v f vs)
prefer  $2$  apply(simp)
by(drule (1) splitFace-face-face-op[where f1 = fst(splitFace g u v f vs) and f2 =
fst(snd(splitFace g u v f vs))], auto)

lemma splitFace-holds-minGraphProps:
  assumes precond: pre-splitFace g' v a f' [countVertices g'..<countVertices g' +
   $n]$ 
  and min: minGraphProps g'
  shows minGraphProps (snd (snd (splitFace g' v a f' [countVertices g'..<countVertices
   $g' + n])))$ 
proof –
  from min have minGraphProps' g' by (simp add: minGraphProps-def)

```

then show *?thesis* **apply** (*simp add: minGraphProps-def*) **apply** *safe*
apply (*rule splitFace-holds-minGraphProps'*) **apply** (*rule precondition*) **apply** *as-*
assumption
apply (*rule splitFace-holds-facesAt-eq*) **apply** (*rule precondition*) **apply** (*rule min*)
apply *simp*
apply (*rule splitFace-holds-faceListAt-len*) **apply** (*rule precondition*) **apply** (*rule*
min)
apply (*rule splitFace-holds-facesAt-distinct*) **apply** (*rule precondition*) **apply** (*rule*
min)
apply (*rule splitFace-holds-faces-distinct*) **apply** (*rule precondition*) **apply** (*rule*
min)
apply (*rule splitFace-holds-faces-subset*) **apply** (*rule precondition*) **apply** (*rule min*)
apply (*rule splitFace-holds-edges-sym*) **apply** (*rule precondition*) **apply** (*rule min*)
apply (*rule splitFace-edges-disj2*) **apply** (*rule min*) **apply** (*rule precondition*)
apply (*rule splitFace-face-face-op2*) **apply** (*rule min*) **apply** (*rule precondition*)
done
qed

13.7 Invariants of *makeFaceFinal*

lemma *MakeFaceFinal-minGraphProps'*:

$f \in \mathcal{F} \ g \implies \text{minGraphProps} \ g \implies \text{minGraphProps}' \ (\text{makeFaceFinal} \ f \ g)$
apply (*simp add: minGraphProps-def minGraphProps'-def makeFaceFinal-def*)
apply (*subgoal-tac* $2 < |\text{vertices} \ f| \wedge \text{distinct} \ (\text{vertices} \ f)$)
apply (*rule ballI*) **apply** (*elim conjE ballE*) **apply** (*rule conjI*) **apply** *simp* **apply**
simp
apply (*simp add: makeFaceFinalFaceList-def*) **apply** (*drule replace5*) **apply**
(*simp add: setFinal-def*)
by *force*

lemma *MakeFaceFinal-facesAt-eq*:

$f \in \mathcal{F} \ g \implies \text{minGraphProps} \ g \implies \text{facesAt-eq} \ (\text{makeFaceFinal} \ f \ g)$
apply (*simp add: facesAt-eq-def*) **apply** (*rule ballI*)
apply (*subgoal-tac* $v \in \mathcal{V} \ g$)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp add: makeFaceFinal-def facesAt-def*)
apply (*subgoal-tac* $v < |\text{faceListAt} \ g|$)
apply (*simp add: makeFaceFinalFaceList-def*)
apply (*subgoal-tac* $\text{distinct} \ ((\text{faceListAt} \ g \ ! \ v))$)
apply (*subgoal-tac* $\text{distinct} \ (\text{faces} \ g)$)
apply (*simp add: replace6*)
apply (*case-tac* $x = f$)
apply *simp* **apply** (*erule* (1) *minGraphProps6*) **apply** (*simp add: facesAt-def*)
apply *blast*
apply *simp*
apply (*case-tac* $f \in \text{set} \ (\text{faceListAt} \ g \ ! \ v) \wedge x = \text{setFinal} \ f$) **apply** *simp*
apply (*subgoal-tac* $v \in \mathcal{V} \ f$) **apply** (*simp add: setFinal-def*)
apply (*erule* (1) *minGraphProps6*) **apply** (*simp add: facesAt-def*)

apply simp
apply (rule conjI) apply (rule disjI2)
apply (erule (1) minGraphProps5) apply (fastforce simp: facesAt-def)
apply (erule (1) minGraphProps6) apply (fastforce simp: facesAt-def)
apply (rule minGraphProps11') apply simp
apply (rule normFaces-distinct) apply (rule minGraphProps8a) apply simp
apply simp
apply (simp add: vertices-graph minGraphProps4)

apply (rule subsetI) apply (simp add: makeFaceFinal-def facesAt-def)
apply (subgoal-tac $v < | \text{faceListAt } g |$) apply simp
apply (subgoal-tac distinct (faceListAt $g ! v$))
apply (subgoal-tac distinct (faces g))
apply (simp add: makeFaceFinalFaceList-def replace6)
apply (case-tac $x = \text{setFinal } f$) apply simp
apply (rule disjI1) apply (rule minGraphProps7') apply simp apply simp
apply (simp add: setFinal-def) apply simp
apply (rule minGraphProps7') apply simp apply simp apply simp
apply (rule minGraphProps11') apply simp
apply (rule normFaces-distinct) apply (rule minGraphProps8a) apply simp
apply simp
apply (simp add: vertices-graph minGraphProps4)

apply (simp add: makeFaceFinal-def) by (simp add: in-vertices-graph minGraph-Props4)

lemma MakeFaceFinal-faceListAt-len:
 $f \in \mathcal{F} \ g \implies \text{minGraphProps } g \implies \text{faceListAt-len (makeFaceFinal } f \ g)$
apply (simp add: faceListAt-len-def makeFaceFinal-def) apply (rule minGraph-Props4) by simp

lemma normFaces-makeFaceFinalFaceList: $(\text{normFaces (makeFaceFinalFaceList } f \ fs)) = (\text{normFaces } fs)$
apply (simp add: normFaces-def) apply (simp add: makeFaceFinalFaceList-def)
apply (induct fs) apply simp apply simp apply (rule impI)
by (simp add: setFinal-def normFace-def verticesFrom-def minVertex-def)

lemma MakeFaceFinal-facesAt-distinct:
 $f \in \mathcal{F} \ g \implies \text{minGraphProps } g \implies \text{facesAt-distinct (makeFaceFinal } f \ g)$
apply (simp add: facesAt-distinct-def makeFaceFinal-def)
apply (clarsimp simp: facesAt-def)
apply (subgoal-tac $v < | \text{faceListAt } g |$) apply (simp add: normFaces-makeFaceFinalFaceList)
apply (rule minGraphProps8a') apply simp apply simp by (simp add: min-GraphProps4)

lemma MakeFaceFinal-faces-subset:
 $f \in \mathcal{F} \ g \implies \text{minGraphProps } g \implies \text{faces-subset (makeFaceFinal } f \ g)$
apply (simp add: faces-subset-def) apply (intro ballI subsetI)

apply (*simp add: makeFaceFinal-def makeFaceFinalFaceList-def*)
apply (*drule replace5*)
apply (*case-tac fa ∈ F g*) **apply** *simp* **apply** (*rule minGraphProps9'*)
apply *simp* **apply** (*thin-tac f ∈ F g*) **apply** *simp+*
apply (*rule minGraphProps9'*) **apply** *simp* **apply** *simp* **by** (*simp add: setFinal-def*)

lemma *MakeFaceFinal-edges-sym:*

$f \in \mathcal{F} \ g \implies \text{minGraphProps } g \implies \text{edges-sym } (\text{makeFaceFinal } f \ g)$
apply (*simp add: edges-sym-def*) **apply** (*intro allI impI*)
apply (*simp add: makeFaceFinal-def edges-graph-def*)
apply (*elim bexE*) **apply** (*simp add: makeFaceFinalFaceList-def*)
apply (*subgoal-tac distinct (faces g)*)
apply (*case-tac x ∈ F g*)
apply (*subgoal-tac (a,b) ∈ edges g*) **apply** (*frule minGraphProps10*) **apply**
assumption
apply (*simp add: edges-graph-def*) **apply** (*elim bexE*)
apply (*case-tac xb = f*)
apply (*subgoal-tac (b,a) ∈ edges (setFinal f)*)
apply (*rule beXI*) **apply** (*rotate-tac -1*) **apply** *assumption*
apply (*rule replace3*) **apply** *simp* **apply** *simp*
apply (*subgoal-tac distinct (vertices f)*)
apply (*simp add: edges-setFinal*)
apply (*rule minGraphProps3*) **apply** *simp* **apply** *simp*
apply (*rule beXI*) **apply** *assumption* **apply** (*rule replace4*) **apply** *simp* **apply**
force
apply (*simp add: edges-graph-def*) **apply** *force*
apply (*frule replace5*) **apply** *simp*
apply (*subgoal-tac (a,b) ∈ edges g*)
apply (*frule minGraphProps10*) **apply** *assumption* **apply** (*simp add: edges-graph-def*)
apply (*elim bexE*)
apply (*case-tac xb = f*)
apply (*subgoal-tac (b, a) ∈ edges (setFinal f)*)
apply (*rule beXI*) **apply** (*rotate-tac -1*) **apply** *assumption*
apply (*rule replace3*) **apply** *simp* **apply** *simp*
apply (*subgoal-tac distinct (vertices f)*)
apply (*simp add: edges-setFinal*)
apply (*rule minGraphProps3*) **apply** *simp* **apply** *simp*
apply (*rule beXI*) **apply** *simp* **apply** (*rule replace4*) **apply** *simp* **apply** *force*
apply (*subgoal-tac distinct (vertices f)*)
apply (*subgoal-tac (a,b) ∈ edges f*)
apply (*simp add: edges-graph-def*) **apply** *force*
apply (*simp add: edges-setFinal*)
apply (*rule minGraphProps3*) **apply** *simp* **apply** *simp*
by (*rule minGraphProps11'*)

lemma *MakeFaceFinal-faces-distinct:*

$f \in \mathcal{F} \ g \implies \text{minGraphProps } g \implies \text{faces-distinct } (\text{makeFaceFinal } f \ g)$
apply (*simp add: faces-distinct-def makeFaceFinal-def normFaces-makeFaceFinalFaceList*)

by (rule *minGraphProps11*)

lemma *MakeFaceFinal-edges-disj*:

$f \in \mathcal{F} g \implies \text{minGraphProps } g \implies \text{edges-disj } (\text{makeFaceFinal } f g)$
apply(frule *minGraphProps11* ')
apply (clarsimp simp: *edges-disj-def* *makeFaceFinal-def* *edges-graph-def*
 makeFaceFinalFaceList-def *replace6*)
apply(case-tac $f = f'$)
 apply (*fastforce* *dest:mgp-edges-disj*)
apply (*fastforce* *dest:mgp-edges-disj*)
done

lemma *MakeFaceFinal-face-face-op*:

$f \in \mathcal{F} g \implies \text{minGraphProps } g \implies \text{face-face-op } (\text{makeFaceFinal } f g)$
apply(subgoal-tac *face-face-op* g)
 prefer 2 **apply**(simp add: *minGraphProps-def*)
apply(drule *minGraphProps11* ')
apply(auto simp: *face-face-op-def* *makeFaceFinal-def* *makeFaceFinalFaceList-def*
 distinct-set-replace)
done

lemma *MakeFaceFinal-minGraphProps*:

$f \in \mathcal{F} g \implies \text{minGraphProps } g \implies \text{minGraphProps } (\text{makeFaceFinal } f g)$
apply (simp (no-asm) add: *minGraphProps-def*)
apply (simp add: *MakeFaceFinal-minGraphProps'* *MakeFaceFinal-facesAt-eq*
 MakeFaceFinal-faceListAt-len *MakeFaceFinal-facesAt-distinct*
 MakeFaceFinal-faces-subset *MakeFaceFinal-edges-sym*
 MakeFaceFinal-edges-disj *MakeFaceFinal-faces-distinct*
 MakeFaceFinal-face-face-op)
done

13.8 Invariants of *subdivFace'*

lemma *subdivFace'-holds-minGraphProps*: $\bigwedge f v' v n g.$

$\text{pre-subdivFace}' g f v' v n \text{ ovl} \implies f \in \mathcal{F} g \implies$
 $\text{minGraphProps } g \implies \text{minGraphProps } (\text{subdivFace}' g f v n \text{ ovl})$

proof (induct *ovl*)

case *Nil* **then show** ?case **by** (simp add: *MakeFaceFinal-minGraphProps*)
next

case (*Cons* *ov* *ovl*) **then show** ?case

apply *auto*

apply (*cases* *ov*)

apply (simp-all *split: if-split-asm*)

apply (rule *Cons*)

apply (rule *pre-subdivFace'-None*)

apply *simp-all*

apply (*intro* *conjI*)

```

apply clarsimp
apply (rule Cons)
  apply (rule pre-subdivFace'-Some2)
  apply simp-all
apply (clarsimp simp: split-def)
apply (rule Cons)
  apply (rule pre-subdivFace'-Some1)
  apply simp-all
  apply (simp add: minGraphProps-def faces-subset-def)
apply (rule splitFace-add-f21')
apply simp-all
apply (rule splitFace-holds-minGraphProps)
apply simp-all
apply (rule pre-subdivFace'-preFaceDiv)
  apply simp-all
by (simp add: minGraphProps-def faces-subset-def)
qed

```

abbreviation (*input*)

```

Edges-if :: face  $\Rightarrow$  vertex  $\Rightarrow$  vertex  $\Rightarrow$  (vertex  $\times$  vertex)set where
Edges-if f u v ==
  if u=v then {} else Edges(u # between (vertices f) u v @ [v])

```

lemma *FaceDivisionGraph-one-final-but*:

```

assumes mgp: minGraphProps g and pre: pre-splitFace g u v f vs
and fdg: (f1,f2,g') = splitFace g u v f vs
and nrv: r  $\neq$  v
and ruv: before (verticesFrom f r) u v and rf: r  $\in$   $\mathcal{V}$  f
and 1: one-final-but g (Edges-if f r u)
shows one-final-but g' (Edges(r # between (vertices f2) r v @ [v]))
proof –
  have f1: f1 = fst(split-face f u v vs) and f2: f2 = snd(split-face f u v vs)
  and F:  $\mathcal{F}$  g' = {f1}  $\cup$  set(replace f [f2] (faces g))
  and g': g' = snd (snd (splitFace g u v f vs)) using fdg
  by(auto simp add:splitFace-def split-def)
  note pre-split = pre-splitFace-pre-split-face[OF pre]
  from pre have nf:  $\neg$  final f and fg: f  $\in$   $\mathcal{F}$  g and nuv: u  $\neq$  v
  and uinf: u  $\in$   $\mathcal{V}$  f and vinf: v  $\in$   $\mathcal{V}$  f
  by(unfold pre-splitFace-def, simp)+
  from mgp fg have distf: distinct(vertices f) by(rule minGraphProps3)
  note distFg = minGraphProps11'[OF mgp]
  have fvu: r  $\neq$  u  $\implies$  between (vertices f) v u =
    between (vertices f) v r @ r # between (vertices f) r u
  using before-between2[OF ruv distf rf] nrv
  split-between[OF distf vinf uinf, of r] by (auto)
  let ?fuv = between (vertices f) u v and ?fvu = between (vertices f) v u

```



```

let ?fru = between (vertices f) r u and ?f2rv = between (vertices f2) r v
have E1:  $\mathcal{E} f_1 = \text{Edges}(v \# \text{rev } vs @ [u]) \cup \text{Edges}(u \# ?fw @ [v])$ 
  using f1 by(simp add:edges-split-face1[OF pre-split])
have E2:  $\mathcal{E} f_2 = \text{Edges}(u \# vs @ [v]) \cup \text{Edges}(v \# ?fvu @ [u])$ 
  using f2 by(simp add:edges-split-face2[OF pre-split])
have vf2: vertices f2 = [v] @ ?fvu @ u # vs
  using f2 by(simp add:split-face-def)
have vinf2:  $v \in \mathcal{V} f_2$  using vf2 by(simp)
have rinf2:  $r \in \mathcal{V} f_2$ 
proof cases
  assume r=u thus ?thesis by(simp add:vf2)
next
  assume r≠u thus ?thesis by(simp add: vf2 fvu)
qed
have distf2: distinct(vertices f2)
  by(simp add:f2)(rule split-face-distinct2'[OF pre-split])
have f2uv: between (vertices f2) u v = vs
  using vf2 distf2 by(simp add:between-def split-def)
have f2ru:  $r \neq u \implies \text{between}(vertices f_2) r u = \text{between}(vertices f) r u$ 
  using vf2 fvu distf2 by(simp add:between-def split-def)
hence f2rv: between (vertices f2) r v =
  (if r=u then [] else ?fru @ [u]) @ vs
proof cases
  assume r=u thus ?thesis by(simp add: f2uv)
next
  assume nru:  $r \neq u$ 
  have vinf2:  $v \in \mathcal{V} f_2$  by(simp add: vf2)
  note u-bet-rv = before-between[OF ruw distf rf nru]
  have u-bet-rv2:  $u \in \text{set}(\text{between}(vertices f_2) r v)$ 
    using distf2 nru
  apply(simp add:vf2 fvu)
  apply(subst between-def[of - r v])
  apply(simp add:split-def)
  done
  show ?thesis
  by(simp add:split-between[OF distf2 rinf2 vinf2 u-bet-rv2] f2ru f2uv)
qed
have E2rv:  $\text{Edges}(r \# ?f2rv @ [v]) =$ 
   $\text{Edges-if } f r u \cup \text{Edges}(u \# vs @ [v])$  (is ?L = ?R)
proof -
  have ?L =  $\text{Edges}((\text{if } r=u \text{ then } [] \text{ else } r \# ?fru) @ (u \# vs @ [v]))$ 
    by (simp add: f2rv)
  also have ... = ?R by(auto simp:Edges-Cons Edges-append)
  finally show ?thesis .
qed
show ?thesis
proof (auto del: disjCI simp:one-final-but-def F, goal-cases)
  case prems: (1 a b)
  have ab:  $(a,b) \in \mathcal{E} f_1$ 

```

```

    and nab: (a,b) ∉ Edges (r # ?f2rv @ [v]) by fact+
  have (a,b) ∈ Edges (v # rev vs @ [u]) ∨
    (a,b) ∈ Edges (u # ?fuv @ [v]) (is ?A ∨ ?B)
  using E1 ab by blast
  thus ?case
  proof
    assume ?A
    hence (b,a) ∈ Edges (rev(v # rev vs @ [u])) by (simp del:rev.simps)
    hence (b,a) ∈ Edges (r # ?f2rv @ [v]) using E2rv by simp
    thus ?case by blast
  next
  assume abfu: ?B
  have abf: (a,b) ∈ E f
    by(rule Edges-between-edges[OF abfu pre-split])
  have (∃ f' ∈ set(replace f [f2] (faces g)). final f' ∧ (b,a) ∈ E f')
  proof cases
    assume r = u
    then obtain f' where f' ∈ F g ∧ final f' ∧ (b, a) ∈ E f'
      using abf 1 nf fg by(simp add:one-final-but-def)fast
    moreover then have f' ∈ set (replace f [f2] (faces g))
      by(clarsimp simp: replace6[OF distFg] nf)
    ultimately show ?thesis by blast
  next
  assume nru: r ≠ u
  moreover hence (a,b) ∉ Edges (r # ?fru @ [u])
    using abfu Edges-disj[OF distf rf uinf nru nuv
      before-between[OF ruv distf rf nru]] by fast
  moreover have (b,a) ∉ Edges (r # ?fru @ [u])
  proof
    assume (b,a) ∈ Edges (r # ?fru @ [u])
    moreover have pre-split-face f r u []
      by(simp add:pre-split-face-def distf rf uinf nru)
    ultimately show False
      using minGraphProps12[OF mgp fg abf]
      by(blast dest:Edges-between-edges)
  qed
  ultimately obtain f' where f' ∈ F g ∧ final f' ∧ (b, a) ∈ E f'
    using abf 1 nf fg by(simp add:one-final-but-def)fast
  moreover hence f' ∈ set (replace f [f2] (faces g))
    by(clarsimp simp: replace6[OF distFg] nf)
  ultimately show ?thesis by blast
  qed
  thus ?case ..
  qed
  next
  case (2 f' a b)
  have f': f' ∈ set (replace f [f2] (faces g))
  and nf': ¬ final f' and abf': (a,b) ∈ E f'
  and nab: (a,b) ∉ Edges (r # between (vertices f2) r v @ [v]) by fact+

```

```

have f' = f2 ∨ f' ∈ ℱ g ∧ f' ≠ f
  using f' by(simp add:replace6[OF distFg]) blast
hence (b, a) ∈ Edges (r # between (vertices f2) r v @ [v]) ∨
  (∃ f' ∈ set (replace f [f2] (faces g)). final f' ∧ (b, a) ∈ ℰ f')
(is ?A ∨ ?B)
proof
  assume [simp]: f' = f2
  have (a,b) ∈ Edges (v # between (vertices f2) v r @ [r])
    using abf' nab Edges-compl[OF distf2 vinf2 rinf2 nrv[symmetric]]
    edges-conv-Un-Edges[OF distf2 rinf2 vinf2 nrv] by auto
  moreover have eq: between(vertices f2) v r = between (vertices f) v r
  proof (cases r=u)
    assume r=u thus ?thesis
      by(simp add:vf2)(rule between-front[OF between-not-r2[OF distf]])
    next
      assume r≠u thus ?thesis
        by(simp add:vf2 fvu)(rule between-front[OF between-not-r2[OF distf]])
  qed
  ultimately
  have abfvr: (a,b) ∈ Edges (v # between (vertices f) v r @ [r])
    by simp
  have abf: (a,b) ∈ ℰ f
    apply(rule Edges-between-edges[where vs = [], OF abfvr])
    using distf rf vinf nrv by(simp add:pre-split-face-def)
  have (∃ f' ∈ set(replace f [f2] (faces g)). final f' ∧ (b,a) ∈ ℰ f')
  proof cases
    assume r = u
    then obtain f' where f' ∈ ℱ g ∧ final f' ∧ (b, a) ∈ ℰ f'
      using abf 1 nf fg by(simp add:one-final-but-def)fast
    moreover then have f' ∈ set (replace f [f2] (faces g))
      by(clarsimp simp: replace6[OF distFg] nf)
    ultimately show ?thesis by blast
  next
    assume nru: r ≠ u
    note uvr = rotate-before-vFrom[OF distf rf nru ruv]
    note bet = before-between[OF uvr distf vinf nrv[symmetric]]
    have (a,b) ∉ Edges (r # ?fru @ [u])
      using abfvr Edges-disj[OF distf vinf uinf nrv[symmetric] nru bet]
      by fast
    moreover have (b,a) ∉ Edges (r # ?fru @ [u])
  proof
    assume (b,a) ∈ Edges (r # ?fru @ [u])
    moreover have pre-split-face f r u []
      by(simp add:pre-split-face-def distf rf uinf nru)
    ultimately show False
      using minGraphProps12[OF mgp fg abf]
      by(blast dest:Edges-between-edges)
  qed
  ultimately obtain f' where f' ∈ ℱ g ∧ final f' ∧ (b, a) ∈ ℰ f'

```

```

    using abf 1 nf fg nru by(simp add:one-final-but-def)fast
  moreover hence  $f' \in \text{set } (\text{replace } f [f_2] (\text{faces } g))$ 
    by(clarsimp simp: replace6[OF distFg] nf)
  ultimately show ?thesis by blast
qed
thus ?thesis ..
next
assume  $f': f' \in \mathcal{F} g \wedge f' \neq f$ 
moreover
hence  $\mathcal{E} f' \cap \mathcal{E} f = \{\}$ 
  using fg by(blast dest: mgp-edges-disj[OF mgp])
moreover
have  $\text{Edges-if } f r u \subseteq \mathcal{E} f$ 
  using distf rf uinf
  apply(clarsimp simp del:is-nextElem-edges-eq)
  apply(erule Edges-between-edges[where vs = []])
  by(simp add:pre-split-face-def)
ultimately
have  $(b,a) : \text{Edges-if } f r u \vee$ 
   $(\exists f'' \in \mathcal{F} g. \text{final } f'' \wedge (b,a) \in \mathcal{E} f'')$  (is ?A  $\vee$  ?B)
  using 1 f' nf' abf'
  by(simp add:one-final-but-def split:if-split-asm) blast+
thus ?thesis (is ?A'  $\vee$  ?B')
proof
  assume ?A
  moreover
  have  $\text{Edges-if } f r u \subseteq \text{Edges } (r \# \text{between } (\text{vertices } f_2) r v @ [v])$ 
    using f2rv by (auto simp:Edges-Cons Edges-append)
  ultimately have ?A' by blast
  thus ?thesis ..
next
  assume ?B
  then obtain  $f''$  where  $f'' \in \mathcal{F} g \wedge \text{final } f'' \wedge (b, a) \in \mathcal{E} f''$ 
    by blast
  moreover hence  $f'' \neq f$  using nf by blast
  ultimately have ?B' by (blast intro:in-set-repl)
  thus ?thesis ..
qed
qed
thus ?case by blast
qed
qed

```

lemma *one-final-but-makeFaceFinal*:

```

[[ minGraphProps g; one-final-but g E; E  $\subseteq$   $\mathcal{E} f$ ; f  $\in$   $\mathcal{F} g$ ;  $\neg$  final f ]]  $\implies$ 
  one-final (makeFaceFinal f g)
apply(frule minGraphProps11')
apply(clarsimp simp add:one-final-but-def one-final-def makeFaceFinal-def)

```

```

      makeFaceFinalFaceList-def replace6)
apply(rename-tac f' a b)
apply(erule disjE)
  apply(simp)
apply(subgoal-tac (a,b)  $\notin$  E)
  prefer 2 apply (simp add:minGraphProps-def edges-disj-def) apply blast
apply(drule-tac x = f' in bspec)
  apply assumption
apply simp
apply(drule-tac x = (a,b) in bspec)
  apply simp
apply(fastforce simp add: replace6)
done

```

lemma one-final-subdivFace':

```

 $\bigwedge f v n g.$ 
  pre-subdivFace' g f u v n ovs  $\implies$  minGraphProps g  $\implies$  f  $\in$   $\mathcal{F}$  g  $\implies$ 
  one-final-but g (Edges-if f u v)  $\implies$ 
  one-final(subdivFace' g f v n ovs)
proof (induct ovs)
  case Nil
  hence pre-split-face f u v []
    by(simp add:pre-split-face-def pre-subdivFace'-def)
  hence Edges(u # between (vertices f) u v @ [v])  $\subseteq$   $\mathcal{E}$  f
    by(auto simp add:Edges-between-edges)
  moreover have  $\neg$  final f using Nil by(simp add:pre-subdivFace'-def)
  ultimately show ?case using Nil by (simp add: one-final-but-makeFaceFinal)
next
  case (Cons ov ovs)
  note IH = Cons(1) and pre = Cons(2) and mgp = Cons(3) and fg = Cons(4)
  note 1 = Cons(5)
  have nf:  $\neg$  final f and uf: u  $\in$   $\mathcal{V}$  f and vf: v  $\in$   $\mathcal{V}$  f
    using pre by(simp add:pre-subdivFace'-def)+
  show ?case
  proof (cases ov)
  case None
  have pre': pre-subdivFace' g f u v (Suc n) ovs
    using None pre by (simp add: pre-subdivFace'-None)
  show ?thesis using None
    by (simp add: IH[OF pre' mgp fg 1])
  next
  case (Some w)
  have uw: u  $\neq$  w using pre Some by(clarsimp simp: pre-subdivFace'-def)
  { assume w: f  $\cdot$  v = w and n: n = 0
    from w minGraphProps3[OF mgp fg]
    have vw: nextElem (vertices f) (hd(vertices f)) v = w
      by(simp add:nextVertex-def)
    have 2: one-final-but g (if u=w then {} else Edges (u # between (vertices f)

```

```

u w @ [w])
  apply (rule one-final-but-antimono[OF 1])
  using uw apply clarsimp
  apply(subgoal-tac pre-between (vertices f) u v)
  prefer 2
  using pre vf apply(simp add:pre-subdivFace'-def pre-between-def)
  apply(simp add:between-nextElem vw[symmetric])
  apply(fastforce simp add:Edges-Cons Edges-append)
  done
  have pre': pre-subdivFace' g f u w 0 ovs
  using pre Some n using [[simp-depth-limit = 5]] by (simp add: pre-subdivFace'-Some2)
  have one-final (subdivFace' g f w 0 ovs)
  by (simp add: IH[OF pre' mgp fg 2])
} moreover
{ let ?vs = [countVertices g..<countVertices g + n]
  let ?fdg = splitFace g v w f ?vs
  let ?Ew = if u=w then {} else Edges(u # between(vertices (fst(snd ?fdg))) u
w @ [w])
  assume a: f · v = w → 0 < n
  have pre2: pre-subdivFace' g f u v n (Some w # ovs)
  using pre Some by simp
  have fsubg: V f ⊆ V g
  using mgp fg by(simp add: minGraphProps-def faces-subset-def)
  have pre-fdg: pre-splitFace g v w f ?vs
  apply (rule pre-subdivFace'-preFaceDiv[OF fg fsubg])
  using Some pre apply simp
  using a apply (simp)
  done
  have bet: before (verticesFrom f u) v w using pre Some
  by(unfold pre-subdivFace'-def) simp
  have 2: one-final-but(snd(snd ?fdg)) ?Ew
  using uw apply simp
  apply(rule FaceDivisionGraph-one-final-but[OF mgp pre-fdg - uw bet uf 1])
  apply(fastforce intro!:prod-eqI)
  done
  note mgp' = splitFace-holds-minGraphProps[OF pre-fdg mgp]
  have pre2': pre-subdivFace' (snd (snd ?fdg)) (fst (snd ?fdg)) u w 0 ovs
  by (rule pre-subdivFace'-Some1[OF pre2 fg fsubg HOL.refl HOL.refl])
  (simp add:a)
  note f2inF = splitFace-add-f21'[OF fg]
  have one-final (subdivFace' (snd (snd ?fdg)) (fst (snd ?fdg)) w 0 ovs)
  by (simp add: IH[OF pre2' mgp' f2inF 2])
} ultimately show ?thesis using Some by (simp add: split-def)
qed
qed

```

lemma neighbors-edges:

$\text{minGraphProps } g \implies a : \mathcal{V} g \implies b \in \text{set } (\text{neighbors } g a) = ((a, b) \in \text{edges } g)$

```

apply (rule iffI)
apply (simp add: neighbors-def) apply clarify apply (frule (1) minGraphProps5)
apply (simp add: vertices-graph)
apply (simp add: edges-graph-def) apply (intro bexE)
prefer 2 apply assumption
apply(simp add:edges-face-eq)
apply (erule (2) minGraphProps6)
apply (simp add: neighbors-def) apply (simp add: edges-graph-def) apply (elim
bexE)
apply (subgoal-tac  $x \in \text{set } (\text{facesAt } g \ a)$ ) apply (simp add: edges-face-def)
apply (rule minGraphProps7) apply simp+ apply (simp add: edges-face-def)
done

```

```

lemma no-self-edges: minGraphProps'  $g \implies (a, a) \notin \text{edges } g$  apply (simp add:
minGraphProps'-def)
apply (induct  $g$ ) apply simp apply (simp add: edges-graph-def) apply auto ap-
ply (drule bspec) apply assumption
apply auto apply (simp add: is-nextElem-def) apply safe apply (simp add:
is-sublist-def) apply force
apply (case-tac vertices  $x$ ) apply simp apply (case-tac list rule: rev-exhaust) ap-
ply simp by simp

```

Requires only *distinct* (vertices f) and that g has no self-loops.

```

lemma duplicateEdge-is-duplicateEdge-eq:
minGraphProps  $g \implies f \in \mathcal{F} \ g \implies a \in \mathcal{V} \ f \implies b \in \mathcal{V} \ f \implies$ 
duplicateEdge  $g \ f \ a \ b = \text{is-duplicateEdge } g \ f \ a \ b$ 
apply (subgoal-tac distinct (vertices  $f$ ))
prefer 2 apply (simp add: minGraphProps3)
apply(subgoal-tac  $a : \mathcal{V} \ g$ )
prefer 2 apply (simp add: minGraphProps9)
apply (simp add: duplicateEdge-def is-duplicateEdge-def neighbors-edges)
apply (rule iffI)
apply (simp add: minGraphProps10)
apply (cases  $a = b$ ) apply (simp add: no-self-edges minGraphProps-def)
apply (rule ccontr)
apply (simp add: directedLength-def)
apply (case-tac is-nextElem (vertices  $f$ )  $a \ b$ )
apply (simp add: is-nextElem-between-empty)
apply (simp add: is-nextElem-between-empty)
apply (cases  $a = b$ ) apply (simp add: no-self-edges minGraphProps-def)
apply (rule ccontr)
apply (simp add: directedLength-def)
apply (elim impE)
apply (cases between (vertices  $f$ )  $b \ a$ )
apply (simp add: is-nextElem-between-empty' del:is-nextElem-between-empty)
apply simp
apply (cases between (vertices  $f$ )  $a \ b$ )
apply (simp add: is-nextElem-between-empty' del:is-nextElem-between-empty)

```

apply *simp*
apply (*simp add: minGraphProps10*)
done

lemma *incrIndexList-less-eq*:

incrIndexList ls m nmax \implies Suc n < |ls| \implies ls!n \leq ls!Suc n
apply (*subgoal-tac increasing ls*) **apply** (*thin-tac incrIndexList ls m nmax*) **apply**
(*rule increasing1*) **apply** *simp*
apply (*subgoal-tac ls = take n ls @ ls!n # [] @ ls!(Suc n) # drop (Suc (Suc n))*
ls) **apply** *assumption*
apply *simp* **apply** (*subgoal-tac n < |ls|*) **apply** (*rotate-tac -1*) **apply** (*drule*
id-take-nth-drop)
apply (*subgoal-tac drop (Suc n) ls = ls ! Suc n # drop (Suc (Suc n)) ls*) **apply**
simp **apply** (*drule Cons-nth-drop-Suc*)
by *auto*

lemma *incrIndexList-less*:

incrIndexList ls m nmax \implies Suc n < |ls| \implies ls!n \neq ls!Suc n \implies ls!n < ls!Suc n
apply (*drule incrIndexList-less-eq*) **by** *auto*

lemma *Seed-holds-minGraphProps'*: *minGraphProps' (Seed p)*

by (*simp add: graph-def Seed-def minGraphProps'-def*)

lemma *Seed-holds-facesAt-eq*: *facesAt-eq (Seed p)*

by (*force simp add: graph-def Seed-def facesAt-eq-def facesAt-def*)

lemma *minVertex-zero1*: *minVertex (Face [0..*Suc z*] Final) = 0*

apply (*induct z*) **apply** (*simp add: minVertex-def*)

by (*simp add: minVertex-def upt-conv-Cons del: upt-Suc*)

lemma *minVertex-zero2*: *minVertex (Face (rev [0..*Suc z*]) Nonfinal) = 0*

apply (*induct z*) **apply** (*simp add: minVertex-def*)

by (*simp add: minVertex-def min-def*)

13.9 Invariants of *Seed*

lemma *Seed-holds-facesAt-distinct*: *facesAt-distinct (Seed p)*

apply (*simp add: Seed-def graph-def*

facesAt-distinct-def normFaces-def facesAt-def normFace-def)

apply (*simp add: eval-nat-numeral minVertex-zero1 minVertex-zero2 verticesFrom-Def*
fst-splitAt-upt snd-splitAt-upt fst-splitAt-rev snd-splitAt-rev del:upt-Suc)

apply (*simp add: upt-conv-Cons del:upt-Suc*)

apply *simp*

done

lemma *Seed-holds-faces-subset*: *faces-subset (Seed p)*

by (*simp add: Seed-def graph-def faces-subset-def*)

lemma *Seed-holds-edges-sym*: *edges-sym (Seed p)*

by (*simp add: Seed-def graph-def edges-sym-def edges-graph-def*)

lemma *Seed-holds-edges-disj: edges-disj (Seed p)*
using *is-nextElem-circ[of [0..<(p+3)]]*
by (*fastforce simp add: Seed-def graph-def edges-disj-def edges-graph-def*)

lemma *Seed-holds-faces-distinct: faces-distinct (Seed p)*
apply(*simp add: Seed-def graph-def*
 faces-distinct-def normFaces-def facesAt-def normFace-def)
apply(*simp add: eval-nat-numeral minVertex-zero1 minVertex-zero2 verticesFrom-Def*
 fst-splitAt-upt snd-splitAt-upt fst-splitAt-rev snd-splitAt-rev del:upt-Suc)
apply(*simp add:upt-conv-Cons del:upt-Suc*)
apply *simp*
done

lemma *Seed-holds-faceListAt-len: faceListAt-len (Seed p)*
by (*simp add: Seed-def graph-def faceListAt-len-def*)

lemma *face-face-op-Seed: face-face-op(Seed p)*
by (*simp add: Seed-def graph-def face-face-op-def*)

lemma *one-final-Seed: one-final Seed_p*
by(*clarsimp simp:Seed-def one-final-def one-final-but-def graph-def*)

lemma *two-face-Seed: |faces Seed_p| ≥ 2*
by(*simp add:Seed-def graph-def*)

lemma *inv-Seed: inv (Seed p)*
by (*simp add: inv-def minGraphProps-def Seed-holds-minGraphProps'*
 Seed-holds-facesAt-eq Seed-holds-facesAt-distinct Seed-holds-faces-subset
 Seed-holds-edges-sym Seed-holds-edges-disj face-face-op-Seed
 Seed-holds-faces-distinct Seed-holds-faceListAt-len
 one-final-Seed two-face-Seed)

lemma *pre-subdivFace-indexToVertexList:*
assumes *mgp: minGraphProps g* **and** *f: f ∈ set (nonFinals g)*
 and *v: v ∈ V f* **and** *e: e ∈ set (enumerator i |vertices f|)*
 and *containsNot: ¬ containsDuplicateEdge g f v e* **and** *i: 2 < i*
shows *pre-subdivFace g f v (indexToVertexList f v e)*
proof –
 from *e i* **have** *le: |e| = i* **by** (*auto intro: enumerator-length2*)
 from *f* **have** *fg: f ∈ F g* **and** *¬ final f* **by** (*auto simp: nonFinals-def*)
 with *mgp* **have** *le-vf: 2 < |vertices f|*
 by (*simp add: minGraphProps-def minGraphProps'-def*)
 from *fg mgp* **have** *dist-f: distinct (vertices f)*
 by (*simp add: minGraphProps-def minGraphProps'-def*)

```

with le v i e le-vf fg have pre-subdivFace-face f v (indexToVertexList f v e)
  by (rule-tac indexToVertexList-pre-subdivFace-face) simp-all
moreover
from dist-f v e le-vf have indexToVertexList f v e = natToVertexList v f e
  apply (rule-tac indexToVertexList-natToVertexList-eq)
  apply simp
  apply simp
  prefer 2 apply (simp add: enumerator-not-empty)
  by (auto simp:set-enumerator-simps intro:enumerator-bound)
moreover
from e le-vf le i have incrIndexList e i |vertices f| by simp
moreover note mgp containsNot i dist-f v le
ultimately show ?thesis
  apply (simp add: pre-subdivFace-def)
  apply (simp add: invalidVertexList-def)
  apply (simp add: containsDuplicateEdge-eq containsDuplicateEdge'-def)
  apply (rule allI) apply(rename-tac j) apply (rule impI)
  apply (case-tac natToVertexList v f e ! j) apply simp
  apply simp
  apply (case-tac natToVertexList v f e ! Suc j) apply simp
  apply simp
  apply (case-tac j) apply (simp add: natToVertexList-nth-0 natToVertexList-nth-Suc
split: if-split-asm)
  apply (drule-tac spec) apply (rotate-tac -1) apply (erule impE)
  apply (subgoal-tac e ! 0 < e ! Suc 0) apply assumption
  apply (cases e) apply simp
  apply simp
  apply (drule incrIndexList-help21)
  apply simp
  apply (subgoal-tac fe ! 0 . v ∈ V f)
  apply (subgoal-tac fe ! Suc 0 . v ∈ V f)
  apply (simp add: duplicateEdge-is-duplicateEdge-eq [symmetric] fg)
  apply (rule ccontr)
  apply simp
  apply (cases e) apply simp
  apply simp
  apply (drule incrIndexList-help21) apply clarify apply (drule not-sym)
apply (rotate-tac -2) apply simp
  apply (rule nextVertices-in-face) apply simp
  apply (rule nextVertices-in-face) apply simp
  apply simp
  apply (subgoal-tac natToVertexList v f e ! Suc nat =
    (if e ! nat = e ! Suc nat then None else Some (fe ! Suc nat . v)))
  apply (simp split: if-split-asm)
  apply (subgoal-tac natToVertexList v f e ! Suc (Suc nat) =
    (if e ! (Suc nat) = e ! Suc (Suc nat) then None else Some (fe ! Suc (Suc nat)
. v)))
  apply (simp split: if-split-asm)
  apply (drule spec) apply (rotate-tac -1) apply (erule impE)

```

```

apply (subgoal-tac e ! nat < e ! Suc nat) apply assumption
apply (rule incrIndexList-less) apply assumption apply arith
apply simp
apply simp
apply (subgoal-tac fe ! Suc nat . v ∈ V f)
apply (subgoal-tac fe ! Suc (Suc nat) . v ∈ V f)
apply (simp add: duplicateEdge-is-duplicateEdge-eq [symmetric] fg)
apply (rule ccontr) apply simp
apply (rotate-tac -4) apply (erule impE) apply arith
apply (subgoal-tac e ! Suc nat < e ! Suc (Suc nat)) apply force
apply (rule incrIndexList-less) apply assumption apply arith
apply simp
apply (rule nextVertices-in-face) apply simp
apply (rule nextVertices-in-face) apply simp
apply (rule natToVertexList-nth-Suc) apply simp apply arith
apply (rule natToVertexList-nth-Suc) apply simp by arith
qed

```

13.10 Increasing properties of *subdivFace'*

```

lemma subdivFace'-incr:
assumes Ptrans:  $\bigwedge x y z. Q x y \implies P y z \implies P x z$ 
and mkFin:  $\bigwedge f g. f \in \mathcal{F} g \implies \neg \text{final } f \implies P g (\text{makeFaceFinal } f g)$ 
and fdg-incr:  $\bigwedge g u v f vs. \text{pre-splitFace } g u v f vs \implies Q g (\text{snd}(\text{snd}(\text{splitFace } g u v f vs)))$ 
shows
 $\bigwedge f' v n g. \text{pre-subdivFace}' g f' v' v n \text{ovl} \implies \text{minGraphProps } g \implies f' \in \mathcal{F} g \implies P g (\text{subdivFace}' g f' v n \text{ovl})$ 
proof (induct ovl)
case Nil thus ?case by (simp add: pre-subdivFace'-def mkFin)
next
case (Cons ov ovl) then show ?case
apply simp
apply (cases ov)
apply (simp)
apply (rule Cons)
apply (rule pre-subdivFace'-None)
apply assumption+
apply simp
apply (intro conjI)
apply rule
apply simp
apply (rule Cons)
apply (rule pre-subdivFace'-Some2)
apply assumption+
apply (rule)
apply (simp add: split-def)
apply (rule Ptrans)

```

```

prefer 2
apply (rule Cons)
  apply (erule (1) pre-subdivFace'-Some1[OF - - - HOL.refl HOL.refl])
  apply simp
  apply (simp add: minGraphProps-def faces-subset-def)
  apply (rule splitFace-holds-minGraphProps)
  apply (erule (1) pre-subdivFace'-preFaceDiv)
  apply simp
  apply (simp add: minGraphProps-def faces-subset-def)
  apply assumption
  apply (erule splitFace-add-f21')
  apply (rule fdg-incr)
  apply (erule (1) pre-subdivFace'-preFaceDiv)
  apply simp
  apply (simp add: minGraphProps-def faces-subset-def)
done
qed

lemma next-plane0-via-subdivFace':
  assumes mgp: minGraphProps g and gg': g [next-plane0p] → g'
  and P:  $\bigwedge f v' v n g ovs. \text{minGraphProps } g \implies \text{pre-subdivFace}' g f v' v n ovs \implies$ 
     $f \in \mathcal{F} g \implies P g (\text{subdivFace}' g f v n ovs)$ 
  shows P g g'
  proof -
    from gg'
    obtain f v i is e where g': g' = subdivFace g f is
      and f: f ∈ set (nonFinals g) and v: v ∈ V f
      and e: e ∈ set (enumerator i |vertices f| ) and i: 2 < i
      and containsNot:  $\neg \text{containsDuplicateEdge } g f v e$ 
      and is-eq: is = indexToVertexList f v e
    by (auto simp: next-plane0-def generatePolygon-def image-def split:if-split-asm)
    from f have fg: f ∈ F g by (simp add: nonFinals-def)
    note pre-add = pre-subdivFace-indexToVertexList[OF mgp f v e containsNot i]
    with g' is-eq have g': g' = subdivFace' g f v 0 (tl is)
      by (simp add: subdivFace-subdivFace'-eq)
    from pre-add is-eq have is ≠ []
      by (simp add: pre-subdivFace-def pre-subdivFace-face-def)
    with pre-add v is-eq
    have pre-subdivFace' g f v v 0 (tl is)
      by (fastforce simp add: neq-Nil-conv elim: pre-subdivFace-pre-subdivFace')
    from P[OF mgp this fg] g' show ?thesis by simp
  qed

lemma next-plane0-incr:
  assumes Ptrans:  $\bigwedge x y z. Q x y \implies P y z \implies P x z$ 
  and mkFin:  $\bigwedge f g. f \in \mathcal{F} g \implies \neg \text{final } f \implies P g (\text{makeFaceFinal } f g)$ 
  and fdg-incr:  $\bigwedge g u v f vs. \text{pre-splitFace } g u v f vs \implies$ 
     $Q g (\text{snd}(\text{snd}(\text{splitFace } g u v f vs)))$ 

```

```

and mgp: minGraphProps g and gg': g [next-plane0p] → g'
shows P g g'
apply(rule next-plane0-via-subdivFace'[OF mgp gg'])
apply(rule subdivFace'-incr)
  apply(erule (1) Ptrans)
  apply(erule (1) mkFin)
  apply(erule fdg-incr)
apply assumption+
done

```

13.10.1 Increasing number of faces

```

lemma splitFace-incr-faces:
  pre-splitFace g u v f vs ⇒
    finals(snd(snd(splitFace g u v f vs))) = finals g ∧
      |nonFinals(snd(snd(splitFace g u v f vs)))| = Suc |nonFinals g|
apply(unfold pre-splitFace-def)
apply(simp add: splitFace-def split-def finals-def nonFinals-def
  split-face-def filter-replace2 length-filter-replace2)
done

```

```

lemma subdivFace'-incr-faces:
  pre-subdivFace' g f u v n ovs ⇒
    minGraphProps g ⇒ f ∈  $\mathcal{F}$  g ⇒
      |finals (subdivFace' g f v n ovs)| = Suc |finals g| ∧
      |nonFinals(subdivFace' g f v n ovs)| ≥ |nonFinals g| − Suc 0
apply(rule subdivFace'-incr)
prefer 4 apply assumption
prefer 4 apply assumption
prefer 4 apply assumption
prefer 2
apply(simp add: pre-subdivFace'-def len-finals-makeFaceFinal
  len-nonFinals-makeFaceFinal)
prefer 2
apply(erule splitFace-incr-faces)
apply (rule conjI)
  apply simp
apply arith
done

```

```

lemma next-plane0-incr-faces:
  minGraphProps g ⇒ g [next-plane0p] → g' ⇒
    |finals g'| = |finals g| + 1 ∧ |nonFinals g'| ≥ |nonFinals g| − 1
apply simp
apply(rule next-plane0-incr)
prefer 4 apply assumption
prefer 4 apply assumption

```

```

prefer 2
apply(simp add: pre-subdivFace'-def len-finals-makeFaceFinal
      len-nonFinals-makeFaceFinal)
prefer 2
apply(erule splitFace-incr-faces)
apply (rule conjI)
  apply simp
apply arith
done

```

```

lemma two-faces-subdivFace':
  pre-subdivFace' g f u v n ovs  $\implies$  minGraphProps g  $\implies$  f  $\in$   $\mathcal{F}$  g  $\implies$ 
  |faces g|  $\geq$  2  $\implies$  |faces(subdivFace' g f v n ovs)|  $\geq$  2
apply(drule (2) subdivFace'-incr-faces)
using len-faces-sum[of g] len-faces-sum[of subdivFace' g f v n ovs] by arith

```

13.11 Main invariant theorems

```

lemma inv-genPoly:
assumes inv: inv g and polygen: g'  $\in$  set(generatePolygon i v f g)
and f: f  $\in$  set (nonFinals g) and i: 2 < i and v: v  $\in$   $\mathcal{V}$  f
shows inv g'
proof(unfold inv-def)
  have mgp: minGraphProps g and 1: one-final g
    using inv by(simp add:inv-def)+
  from polygen
obtain is e where g': g' = subdivFace g f is
  and e: e  $\in$  set (enumerator i |vertices f| )
  and containsNot:  $\neg$  containsDuplicateEdge g f v e
  and is-eq: is = indexToVertexList f v e
  by (auto simp: generatePolygon-def)
have f': f  $\in$   $\mathcal{F}$  g using f by(simp add:nonFinals-def)
note pre-add = pre-subdivFace-indexToVertexList[OF mgp f v e containsNot i]
with g' is-eq have g': g' = subdivFace' g f v 0 (tl is)
  by (simp add: subdivFace-subdivFace'-eq)
from pre-add is-eq have i-nz: is  $\neq$  []
  by (simp add: pre-subdivFace-def pre-subdivFace-face-def)
with pre-add v i-nz is-eq
have pre-addSnd: pre-subdivFace' g f v v 0 (tl is)
  by(fastforce simp add:neq-Nil-conv elim:pre-subdivFace-pre-subdivFace')
note 2 = one-final-antimono[OF 1]
show minGraphProps g'  $\wedge$  one-final g'  $\wedge$  |faces g'|  $\geq$  2
proof auto
  show minGraphProps g' using g' pre-addSnd f
    apply (simp add:nonFinals-def)
    apply (rule subdivFace'-holds-minGraphProps[OF - - mgp])
    by (simp-all add: succs)
next
  show one-final g' using g' 1

```

```

    by (simp add: one-final-subdivFace'[OF pre-addSnd mgp f' 2])
  next
    show |faces g'| ≥ 2 using g'
    by (simp add: two-faces-subdivFace'[OF pre-addSnd mgp f' inv-two-faces[OF
inv]])
  qed
qed

```

```

lemma inv-inv-next-plane0: invariant inv next-plane0p
proof(clarsimp simp:invariant-def)
  fix g g'
  assume inv: inv g and g' ∈ set (next-plane0p g)
  then obtain i v f where g' ∈ set(generatePolygon i v f g)
    and f ∈ set (nonFinals g) and 2 < i and v ∈ V f
    by (auto simp: next-plane0-def split: if-split-asm)
  thus inv g' using inv by(blast intro:inv-genPoly)
qed

end

```

14 Further Plane Graph Properties

```

theory PlaneProps
imports Invariants
begin

```

14.1 final

```

lemma plane-final-facesAt:
assumes inv g final g v : V g f ∈ set (facesAt g v) shows final f
proof –
  from assms(1,3,4) have f ∈ F g by(blast intro: minGraphProps inv-mgp)
  with assms(2) show ?thesis by (rule finalGraph-face)
qed

```

```

lemma finalVertexI:
  [| inv g; final g; v ∈ V g |] ⇒ finalVertex g v
by (auto simp add: finalVertex-def nonFinals-def filter-empty-conv plane-final-facesAt)

```

```

lemma setFinal-notin-finals:
  [| f ∈ F g; ¬ final f; minGraphProps g |] ⇒ setFinal f ∉ set (finals g)
apply(drule minGraphProps11)
apply(cases f)
apply(fastforce simp:finals-def setFinal-def normFaces-def normFace-def
verticesFrom-def minVertex-def inj-on-def distinct-map
split:facetype.splits)

```

done

14.2 degree

lemma *planeN4*: $inv\ g \implies f \in \mathcal{F}\ g \implies 3 \leq |vertices\ f|$
apply (*subgoal-tac* 2 < | *vertices f* |)
 apply *arith*
 apply (*drule inv-mgp*)
 apply (*erule* (1) *minGraphProps2*)
done

lemma *degree-eq*:

assumes *pl*: *inv g* **and** *fin*: *final g* **and** *v*: $v : \mathcal{V}\ g$

shows $degree\ g\ v = tri\ g\ v + quad\ g\ v + except\ g\ v$

proof –

have *dist*: *distinct(facesAt g v)* **using** *pl v* **by** *simp*

have $3: \forall f \in set(facesAt\ g\ v). |vertices\ f| = 3 \vee |vertices\ f| = 4 \vee |vertices\ f| \geq 5$

proof

fix *f* **assume** *f*: $f \in set(facesAt\ g\ v)$

hence $|vertices\ f| \geq 3$

using *minGraphProps5*[*OF inv-mgp*[*OF pl*] *v f*] *planeN4*[*OF pl*] **by** *blast*

thus $|vertices\ f| = 3 \vee |vertices\ f| = 4 \vee |vertices\ f| \geq 5$ **by** *arith*

qed

have $degree\ g\ v = |facesAt\ g\ v|$ **by** (*simp add:degree-def*)

also have $\dots = card(set(facesAt\ g\ v))$ **by** (*simp add:distinct-card*[*OF dist*])

also have $set(facesAt\ g\ v) = \{f \in set(facesAt\ g\ v). |vertices\ f| = 3\} \cup$
 $\{f \in set(facesAt\ g\ v). |vertices\ f| = 4\} \cup$
 $\{f \in set(facesAt\ g\ v). |vertices\ f| \geq 5\}$

(*is - = ?T* \cup *?Q* \cup *?E*)

using 3 **by** *blast*

also have $card(?T \cup ?Q \cup ?E) = card\ ?T + card\ ?Q + card\ ?E$

apply (*subst card-Un-disjoint*)

apply *simp*

apply *simp*

apply *fastforce*

apply (*subst card-Un-disjoint*)

apply *simp*

apply *simp*

apply *fastforce*

apply *simp*

done

also have $\dots = tri\ g\ v + quad\ g\ v + except\ g\ v$ **using** *fin*

by (*simp add:tri-def quad-def except-def*

distinct-card[*symmetric*] *distinct-filter*[*OF dist*]

plane-final-facesAt[*OF pl fin v*] *cong:conj-cong*)

finally show *?thesis* .

qed

lemma *plane-fin-exceptionalVertex-def*:
assumes *pl*: *inv g* **and** *fin*: *final g* **and** *v*: $v : \mathcal{V} g$
shows *exceptionalVertex g v* =
 $(\mid [f \leftarrow \text{facesAt } g \ v . 5 \leq \mid \text{vertices } f \mid] \mid \neq 0)$
proof –
have $\bigwedge f. f \in \text{set } (\text{facesAt } g \ v) \implies \text{final } f$
by(*rule plane-final-facesAt[OF pl fin v]*)
then show *?thesis* **by** (*simp add: filter-simp exceptionalVertex-def except-def*)
qed

lemma *not-exceptional*:
 $\text{inv } g \implies \text{final } g \implies v : \mathcal{V} g \implies f \in \text{set } (\text{facesAt } g \ v) \implies$
 $\neg \text{exceptionalVertex } g \ v \implies \mid \text{vertices } f \mid \leq 4$
by (*auto simp add: plane-fin-exceptionalVertex-def except-def filter-empty-conv*)

14.3 Misc

lemma *in-next-plane0I*:
assumes $g' \in \text{set } (\text{generatePolygon } n \ v \ f \ g) \ f \in \text{set } (\text{nonFinals } g)$
 $v \in \mathcal{V} f \ 3 \leq n \ n < 4 + p$
shows $g' \in \text{set } (\text{next-plane0}_p \ g)$
proof –
from *assms* **have**
 $\exists n \in \{3..<4 + p\}. g' \in \text{set } (\text{generatePolygon } n \ v \ f \ g)$
by *auto*
with *assms* **have**
 $\exists v \in \mathcal{V} f. \exists n \in \{3..<4 + p\}. g' \in \text{set } (\text{generatePolygon } n \ v \ f \ g)$
by *auto*
with *assms* **have**
 $\exists f \in \text{set } (\text{nonFinals } g). \exists v \in \mathcal{V} f. \exists n \in \{3..<4 + p\}. g' \in \text{set } (\text{generatePolygon } n$
 $v \ f \ g)$
by *auto*
moreover **have** $\neg \text{final } g$ **using** *assms(2)*
by (*auto simp: nonFinals-def finalGraph-def filter-empty-conv*)
ultimately show *?thesis*
by (*simp add: next-plane0-def*)
qed

lemma *next-plane0-nonfinals*: $g \ [\text{next-plane0}_p] \rightarrow g' \implies \text{nonFinals } g \neq []$
by(*auto simp: next-plane0-def finalGraph-def*)

lemma *next-plane0-ex*:
assumes $a: g \ [\text{next-plane0}_p] \rightarrow g'$
shows $\exists f \in \text{set } (\text{nonFinals } g). \exists v \in \mathcal{V} f. \exists i \in \text{set } ([3..<\text{Suc } (\text{max } \text{Gon } p)]) .$
 $g' \in \text{set } (\text{generatePolygon } i \ v \ f \ g)$
proof –

```

from a have  $\neg$  final g by (auto simp add: next-plane0-def)
with a show ?thesis
by (auto simp add: next-plane0-def nonFinals-def)
qed

```

```

lemma step-outside2:
  inv g  $\implies$  g [next-plane0p] $\rightarrow$  g'  $\implies$   $\neg$  final g'  $\implies$  |faces g'|  $\neq$  2
apply(frule inv-two-faces)
apply(frule inv-finals-nonempty)
apply(drule inv-mgp)
apply(insert len-faces-sum[of g] len-faces-sum[of g'])
apply(subgoal-tac |nonFinals g|  $\neq$  0)
prefer 2 apply(drule next-plane0-nonfinals) apply simp
apply(subgoal-tac |nonFinals g'|  $\neq$  0)
prefer 2 apply(simp add:finalGraph-def)
apply(drule (1) next-plane0-incr-faces)
apply(case-tac |faces g| = 2)
prefer 2 apply arith
apply(subgoal-tac |finals g|  $\neq$  0)
apply arith
apply simp
done

```

14.4 Increasing final faces

```

lemma set-finals-splitFace[simp]:
   $\llbracket f \in \mathcal{F} g; \neg$  final f  $\rrbracket \implies$ 
  set(finals(snd(snd(splitFace g u v f vs)))) = set(finals g)
apply(auto simp add:splitFace-def split-def finals-def
  split-face-def)
apply(drule replace5)
apply(clarsimp)
apply(erule replace4)
applyclarsimp
done

```

```

lemma next-plane0-finals-incr:
  g [next-plane0p] $\rightarrow$  g'  $\implies$  f  $\in$  set(finals g)  $\implies$  f  $\in$  set(finals g')
apply(auto simp:next-plane0-def generatePolygon-def split:if-split-asm)
apply(erule subdivFace-pres-finals)
apply (simp add:nonFinals-def)
done

```

```

lemma next-plane0-finals-subset:
  g'  $\in$  set (next-plane0p g)  $\implies$ 
  set (finals g)  $\subseteq$  set (finals g')
by (auto simp add: next-plane0-finals-incr)

```

lemma *next-plane0-final-mono*:
 $\llbracket g' \in \text{set } (\text{next-plane0}_p g); f \in \mathcal{F} g; \text{final } f \rrbracket \implies f \in \mathcal{F} g'$
apply(*drule next-plane0-finals-subset*)
apply(*simp add:finals-def*)
apply *blast*
done

14.5 Increasing vertices

lemma *next-plane0-vertices-subset*:
 $\llbracket g' \in \text{set } (\text{next-plane0}_p g); \text{minGraphProps } g \rrbracket \implies \mathcal{V} g \subseteq \mathcal{V} g'$
apply(*rule next-plane0-incr*)
apply(*erule (1) subset-trans*)
apply(*simp add: vertices-makeFaceFinal*)
defer apply *assumption+*
apply (*auto simp: splitFace-def split-def vertices-graph*)
done

14.6 Increasing vertex degrees

lemma *next-plane0-incr-faceListAt*:
 $\llbracket g' \in \text{set } (\text{next-plane0}_p g); \text{minGraphProps } g \rrbracket$
 $\implies |\text{faceListAt } g| \leq |\text{faceListAt } g'| \wedge$
 $(\forall v < |\text{faceListAt } g|. |\text{faceListAt } g ! v| \leq |\text{faceListAt } g' ! v|)$
 $(\text{is } - \implies - \implies ?Q g g')$
apply(*rule next-plane0-incr[where Q = ?Q]*)
prefer 4 apply *assumption*
prefer 4 apply *assumption*
apply(*rule conjI*) **apply** *fastforce*
apply(*clarsimp*)
apply(*erule allE, erule impE, assumption*)
apply(*erule-tac x = v in allE, erule impE*) **apply** *force*
apply *force*
apply(*simp add: makeFaceFinal-def makeFaceFinalFaceList-def*)
apply (*simp add: splitFace-def split-def nth-append nth-list-update*)
done

lemma *next-plane0-incr-degree*:
 $\llbracket g' \in \text{set } (\text{next-plane0}_p g); \text{minGraphProps } g; v \in \mathcal{V} g \rrbracket$
 $\implies \text{degree } g v \leq \text{degree } g' v$
apply(*frule (1) next-plane0-incr-faceListAt*)
apply(*frule (1) next-plane0-vertices-subset*)
apply(*simp add:degree-def facesAt-def*)
apply(*frule minGraphProps4*)
apply(*simp add:vertices-graph*)
done

14.7 Increasing *except*

lemma *next-plane0-incr-except*:

assumes $g' \in \text{set } (\text{next-plane0}_p \ g) \text{ inv } g \ v \in \mathcal{V} \ g$

shows $\text{except } g \ v \leq \text{except } g' \ v$

proof (*unfold except-def*)

note $\text{inv}' = \text{invariantE}[OF \ \text{inv-inv-next-plane0}, OF \ \text{assms}(1,2)]$

note $\text{mgp} = \text{inv-mgp}[OF \ \text{assms}(2)]$ **and** $\text{mgp}' = \text{inv-mgp}[OF \ \text{inv}']$

note $\text{dist} = \text{distinct-filter}[OF \ \text{mgp-dist-facesAt}[OF \ \text{mgp} \ \langle v : \mathcal{V} \ g \rangle]]$

have $v \in \mathcal{V} \ g'$

using $\text{assms}(3) \ \text{next-plane0-vertices-subset}[OF \ \text{assms}(1) \ \text{mgp}]$ **by** *blast*

note $\text{dist}' = \text{distinct-filter}[OF \ \text{mgp}'\text{-dist-facesAt}[OF \ \text{mgp}' \ \langle v : \mathcal{V} \ g' \rangle]]$

have $||f \leftarrow \text{facesAt } g \ v \ . \ \text{final } f \wedge 5 \leq |\text{vertices } f| || =$

$\text{card}\{f \in \text{set}(\text{facesAt } g \ v) \ . \ \text{final } f \wedge 5 \leq |\text{vertices } f|\}$

(**is** $?L = \text{card } ?M$) **using** $\text{distinct-card}[OF \ \text{dist}]$ **by** *simp*

also have $?M = \{f \in \mathcal{F} \ g \ . \ v \in \mathcal{V} \ f \wedge \text{final } f \wedge 5 \leq |\text{vertices } f|\}$

by(*simp add: minGraphProps-facesAt-eq[OF mgp assms(3)]*)

also have $\dots = \{f \in \text{set}(\text{finals } g) \ . \ v \in \mathcal{V} \ f \wedge 5 \leq |\text{vertices } f|\}$

by(*auto simp:finals-def*)

also have $\text{card } \dots \leq \text{card}\{f \in \text{set}(\text{finals } g') \ . \ v \in \mathcal{V} \ f \wedge 5 \leq |\text{vertices } f|\}$

(**is** $- \leq \text{card } ?M$)

apply(*rule card-mono*)

apply *simp*

using $\text{next-plane0-final-subset}[OF \ \text{assms}(1)]$ **by** *blast*

also have $?M = \{f \in \mathcal{F} \ g' \ . \ v \in \mathcal{V} \ f \wedge \text{final } f \wedge 5 \leq |\text{vertices } f|\}$

by(*auto simp:finals-def*)

also have $\dots = \{f \in \text{set}(\text{facesAt } g' \ v) \ . \ \text{final } f \wedge 5 \leq |\text{vertices } f|\}$

by(*simp add: minGraphProps-facesAt-eq[OF mgp' \langle v \in \mathcal{V} \ g' \rangle]*)

also have $\text{card } \dots =$

$||f \leftarrow \text{facesAt } g' \ v \ . \ \text{final } f \wedge 5 \leq |\text{vertices } f| ||$ (**is** $- = ?R$)

using $\text{distinct-card}[OF \ \text{dist}']$ **by** *simp*

finally show $?L \leq ?R$.

qed

14.8 Increasing edges

lemma *next-plane0-set-edges-subset*:

$[[\text{minGraphProps } g; \ g \ [\text{next-plane0}_p] \rightarrow g']] \implies \text{edges } g \subseteq \text{edges } g'$

apply(*rule next-plane0-incr*)

apply(*erule (1) subset-trans*)

apply(*simp add: edges-makeFaceFinal*)

apply(*erule snd-snd-splitFace-edges-incr*)

apply *assumption+*

done

14.9 Increasing final vertices

declare *atLeastLessThan-iff[iff]*

lemma *next-plane0-incr-finV*:

```

[[g' ∈ set (next-plane0p g); minGraphProps g ]
  ⇒ ∀ v ∈ V g. v ∈ V g' ∧
    ((∀ f ∈ F g. v ∈ V f → final f) →
     (∀ f ∈ F g'. v ∈ V f → f ∈ F g)) (is - ⇒ - ⇒ ?Q g g')
apply(rule next-plane0-incr[where Q = ?Q and g=g and g'=g'])
prefer 4 apply assumption
prefer 4 apply assumption
  apply fast
apply(clarsimp simp:makeFaceFinal-def vertices-graph makeFaceFinalFaceList-def)
apply(drule replace5)
apply(erule disjE)apply blast
apply(simp add:setFinal-def)
apply(unfold pre-splitFace-def)
apply(clarsimp simp:splitFace-def split-def vertices-graph)
apply(rule conjI)
  apply(clarsimp simp:split-face-def vertices-graph atLeastLessThan-def)
  apply(blast dest:inbetween-inset)
apply(clarsimp)
apply(erule disjE[OF replace5]) apply blast
apply(clarsimp simp:split-face-def vertices-graph)
apply(blast dest:inbetween-inset)
done

```

lemma next-plane0-finalVertex-mono:

```

[[g' ∈ set (next-plane0p g); inv g; u ∈ V g; finalVertex g u ]
  ⇒ finalVertex g' u
apply(frule (1) invariantE[OF inv-inv-next-plane0])
apply(subgoal-tac u ∈ V g')
  prefer 2 apply(blast dest:next-plane0-vertices-subset inv-mgp)
apply(clarsimp simp:finalVertex-def minGraphProps-facesAt-eq[OF inv-mgp])
apply(blast dest:next-plane0-incr-finV inv-mgp)
done

```

14.10 Preservation of facesAt at final vertices

lemma next-plane0-finalVertex-facesAt-eq:

```

[[g' ∈ set (next-plane0p g); inv g; v ∈ V g; finalVertex g v ]
  ⇒ set(facesAt g' v) = set(facesAt g v)
apply(frule (1) invariantE[OF inv-inv-next-plane0])
apply(subgoal-tac v ∈ V g')
  prefer 2 apply(blast dest:next-plane0-vertices-subset inv-mgp)
apply(clarsimp simp:finalVertex-def minGraphProps-facesAt-eq[OF inv-mgp])
by(blast dest:next-plane0-incr-finV next-plane0-final-mono inv-mgp)

```

lemma next-plane0-len-filter-eq:

```

assumes g' ∈ set (next-plane0p g) inv g v ∈ V g finalVertex g v
shows |filter P (facesAt g' v)| = |filter P (facesAt g v)|

```

proof –
note $inv' = invariantE[OF\ inv\text{-}inv\text{-}next\text{-}plane0, OF\ assms(1,2)]$
note $mgp = inv\text{-}mgp[OF\ assms(2)]$ **and** $mgp' = inv\text{-}mgp[OF\ inv']$
note $dist = distinct\text{-}filter[OF\ mgp\text{-}dist\text{-}facesAt[OF\ mgp\ \langle v : \mathcal{V}\ g \rangle]]$
have $v \in \mathcal{V}\ g'$
using $assms(3)\ next\text{-}plane0\text{-}vertices\text{-}subset[OF\ assms(1)\ mgp]$ **by** *blast*
note $dist' = distinct\text{-}filter[OF\ mgp\text{-}dist\text{-}facesAt[OF\ mgp'\ \langle v : \mathcal{V}\ g' \rangle]]$
have $|filter\ P\ (facesAt\ g'\ v)| = card\{f \in set(facesAt\ g'\ v) . P\ f\}$
using $distinct\text{-}card[OF\ dist']$ **by** *simp*
also have $\dots = card\{f \in set(facesAt\ g\ v) . P\ f\}$
by (*simp add: next-plane0-finalVertex-facesAt-eq[OF assms]*)
also have $\dots = |filter\ P\ (facesAt\ g\ v)|$
using $distinct\text{-}card[OF\ dist]$ **by** *simp*
finally show *?thesis* .
qed

14.11 Properties of *subdivFace'*

lemma *new-edge-subdivFace'*:

$\bigwedge f\ v\ n\ g.$
 $pre\text{-}subdivFace'\ g\ f\ u\ v\ n\ ovs \implies minGraphProps\ g \implies f \in \mathcal{F}\ g \implies$
 $subdivFace'\ g\ f\ v\ n\ ovs = makeFaceFinal\ f\ g \vee$
 $(\forall f' \in \mathcal{F}\ (subdivFace'\ g\ f\ v\ n\ ovs) - (\mathcal{F}\ g - \{f\})).$
 $\exists e \in \mathcal{E}\ f'. e \notin \mathcal{E}\ g)$

proof (*induct ovs*)

case *Nil* **thus** *?case by simp*

next

case (*Cons ov ovs*)

note $IH = Cons(1)$ **and** $pre = Cons(2)$ **and** $mgp = Cons(3)$ **and** $fg = Cons(4)$

have $uf: u \in \mathcal{V}\ f$ **and** $vf: v \in \mathcal{V}\ f$ **and** $distf: distinct\ (vertices\ f)$

using pre **by** (*simp add: pre-subdivFace'-def*)**+**

note $distFg = minGraphProps11'[OF\ mgp]$

show *?case*

proof (*cases ov*)

case *None*

have $pre': pre\text{-}subdivFace'\ g\ f\ u\ v\ (Suc\ n)\ ovs$

using *None pre by (simp add: pre-subdivFace'-None)*

show *?thesis using None*

by (*simp add: IH[OF pre' mgp fg]*)

next

case (*Some w*)

note $pre = pre[simplified\ Some]$

have $uw: before\ (verticesFrom\ f\ u)\ v\ w$

using pre **by** (*simp add: pre-subdivFace'-def*)

have $uw: u \neq w$ **using** pre **by** (*clarsimp simp: pre-subdivFace'-def*)

{ assume $w: f \cdot v = w$ **and** $n: n = 0$

have $pre': pre\text{-}subdivFace'\ g\ f\ u\ w\ 0\ ovs$

using $pre\ Some\ n$ **using** [*simp-depth-limit = 5*] **by** (*simp add: pre-subdivFace'-Some2*)

note $IH[OF\ pre'\ mgp\ fg]$

```

} moreover
{ let ?vs = [countVertices g..<countVertices g + n]
  let ?fdg = splitFace g v w f ?vs
  let ?f1 = fst ?fdg and ?f2 = fst(snd ?fdg) and ?g' = snd(snd ?fdg)
  let ?g'' = subdivFace' ?g' ?f2 w 0 ovs
  let ?fw = between(vertices f) v w and ?fw' = between(vertices f) w v
  assume a: f · v = w → 0 < n
  have fsubg: V f ⊆ V g
    using mgp fg by(simp add: minGraphProps-def faces-subset-def)
  have pre-fdg: pre-splitFace g v w f ?vs
    apply (rule pre-subdivFace'-preFaceDiv[OF pre fg - fsubg])
    using a by (simp)
  hence v ≠ w and w ∈ V f by(unfold pre-splitFace-def) simp+
  have f1: ?f1 = fst(split-face f v w ?vs)
    and f2: ?f2 = snd(split-face f v w ?vs)
    by(auto simp add:splitFace-def split-def)
  note pre-split = pre-splitFace-pre-split-face[OF pre-fdg]
  have E1: E ?f1 = Edges (w # rev ?vs @ [v]) ∪ Edges (v # ?fw' @ [w])
    using f1 by(simp add:edges-split-face1[OF pre-split])
  have E2: E ?f2 = Edges (v # ?vs @ [w]) ∪ Edges (w # ?fw' @ [v])
    by(simp add:splitFace-def split-def
      edges-split-face2[OF pre-split])
  note mgp' = splitFace-holds-minGraphProps[OF pre-fdg mgp]
  note distFg' = minGraphProps11'[OF mgp']
  have pre': pre-subdivFace' ?g' ?f2 u w 0 ovs
    by (rule pre-subdivFace'-Some1[OF pre fg - fsubg HOL.refl HOL.refl])
    (simp add:a)
  note f2inF = splitFace-add-f21'[OF fg]
  have 1: ∃ e ∈ E ?f1. e ∉ E g
  proof cases
    assume rev ?vs = []
    hence (w,v) ∈ E ?f1 ∧ (w,v) ∉ E g using pre-fdg E1
      by(unfold pre-splitFace-def) (auto simp:Edges-Cons)
    thus ?thesis by blast
  next
    assume rev ?vs ≠ []
    then obtain x xs where rvs: rev ?vs = x#xs
      by(auto simp only:neq-Nil-conv)
    hence (w,x) ∈ E ?f1 using E1 by (auto simp:Edges-Cons)
    moreover have (w,x) ∉ E g
    proof -
      have x ∈ set(rev ?vs) using rvs by simp
      hence x ≥ countVertices g by simp
      hence x ∉ V g by(induct g) (simp add:vertices-graph-def)
      thus ?thesis
        by (auto simp:edges-graph-def)
        (blast dest: in-edges-in-vertices minGraphProps9[OF mgp])
    qed
  ultimately show ?thesis by blast

```

qed
have 2: $\exists e \in \mathcal{E} \ ?f_2. e \notin \mathcal{E} \ g$
proof cases
 assume $?vs = []$
 hence $(v,w) \in \mathcal{E} \ ?f_2 \wedge (v,w) \notin \mathcal{E} \ g$ **using** *pre-fdg E₂*
 by(*unfold pre-splitFace-def*) (*auto simp:Edges-Cons*)
 thus *?thesis by blast*
next
 assume $?vs \neq []$
 then obtain $x \ xs$ **where** $vs: ?vs = x\#xs$
 by(*auto simp only:neq-Nil-conv*)
 hence $(v,x) \in \mathcal{E} \ ?f_2$ **using** *E₂* **by** (*auto simp:Edges-Cons*)
 moreover have $(v,x) \notin \mathcal{E} \ g$
 proof –
 have $x \in \text{set } ?vs$ **using** *vs by simp*
 hence $x \geq \text{countVertices } g$ **by** *simp*
 hence $x \notin \mathcal{V} \ g$ **by**(*induct g*) (*simp add:vertices-graph-def*)
 thus *?thesis*
 by (*auto simp:edges-graph-def*)
 (*blast dest: in-edges-in-vertices minGraphProps9[OF mgp]*)
 qed
 ultimately show *?thesis by blast*
qed
have *fdg: (?f₁,?f₂,?g') = splitFace g v w f ?vs by auto*
hence $Fg': \mathcal{F} \ ?g' = \{?f_1,?f_2\} \cup (\mathcal{F} \ g - \{f\})$
 using *set-faces-splitFace[OF mgp fg pre-fdg]* **by** *blast*
have $\forall f' \in \mathcal{F} \ ?g'' - (\mathcal{F} \ g - \{f\}). \exists e \in \mathcal{E} \ f'. e \notin \mathcal{E} \ g$
proof (*clarify*)
 fix f' **assume** $f'g'': f' \in \mathcal{F} \ ?g''$ **and** $f'ng: f' \notin \mathcal{F} \ g - \{f\}$
 from *IH[OF pre' mgp' f2inF]*
 show $\exists e \in \mathcal{E} \ f'. e \notin \mathcal{E} \ g$
 proof
 assume $?g'' = \text{makeFaceFinal } ?f_2 \ ?g'$
 hence $f' = \text{setFinal } ?f_2 \vee f' = ?f_1$ (**is** $?A \vee ?B$)
 using $f'g'' \ Fg' \ f'ng$
 by(*auto simp:makeFaceFinal-def makeFaceFinalFaceList-def*
 distinct-set-replace[OF distFg'])
 thus *?thesis*
 proof
 assume $?A$ **thus** *?thesis using 2 by(simp)*
 next
 assume $?B$ **thus** *?thesis using 1 by blast*
 qed
next
 assume $A: \forall f' \in \mathcal{F} \ ?g'' - (\mathcal{F} \ ?g' - \{?f_2\}).$
 $\exists e \in \mathcal{E} \ f'. e \notin \mathcal{E} \ ?g'$
 show *?thesis*
 proof cases
 assume $f' \in \{?f_1,?f_2\}$


```

      thus ?thesis using 1 2 by blast
    next
      assume  $f' \notin \{?f_1, ?f_2\}$ 
      hence  $\exists e \in \mathcal{E} f'. e \notin \mathcal{E} ?g'$ 
      using  $A f' g'' f' n g Fg'$  by simp
      with splitFace-edges-incr[OF pre-fdg fdg]
      show ?thesis by blast
    qed
  qed
}
ultimately show ?thesis using Some by(auto simp: split-def)
qed

```

lemma *dist-edges-subdivFace'*:

```

pre-subdivFace' g f u v n ovs  $\implies$  minGraphProps g  $\implies$   $f \in \mathcal{F} g \implies$ 
subdivFace' g f v n ovs = makeFaceFinal f g  $\vee$ 
 $(\forall f' \in \mathcal{F} (\text{subdivFace}' g f v n ovs) - (\mathcal{F} g - \{f\}). \mathcal{E} f' \neq \mathcal{E} f)$ 
apply(drule (2) new-edge-subdivFace')
apply(erule disjE)
  apply blast
  apply(rule disjI2)
  apply(clarify)
  apply(drule bspec)
  apply fast
  apply(simp add: edges-graph-def)
by(blast)

```

lemma *between-last*: $\llbracket \text{distinct}(\text{vertices } f); u \in \mathcal{V} f \rrbracket \implies$

```

between (vertices f) u (last (verticesFrom f u)) =
butlast(tl(verticesFrom f u))
apply(drule split-list)
apply (fastforce dest: last-in-set
  simp: between-def verticesFrom-Def split-def
  last-append butlast-append fst-splitAt-last)
done

```

lemma *final-subdivFace'*: $\bigwedge f u n g. \text{minGraphProps } g \implies$

```

pre-subdivFace' g f r u n ovs  $\implies$   $f \in \mathcal{F} g \implies$ 
 $(\text{ovs} = [] \implies n=0 \wedge u = \text{last}(\text{verticesFrom } f r)) \implies$ 
 $\exists f' \in \text{set}(\text{finals}(\text{subdivFace}' g f u n ovs)) - \text{set}(\text{finals } g).$ 
 $(f^{-1} \cdot r, r) \in \mathcal{E} f' \wedge |\text{vertices } f'| =$ 
 $n + |\text{ovs}| + (\text{if } r=u \text{ then } 1 \text{ else } |\text{between } (\text{vertices } f) r u| + 2)$ 
proof (induct ovs)

```

```

case Nil show ?case (is  $\exists f' \in ?F. ?P f'$ )
proof
  show ?P (setFinal f) (is ?A  $\wedge$  ?B)
  proof
    show ?A using Nil
    by(simp add:pre-subdivFace'-def prevVertex-in-edges
      del:is-nextElem-edges-eq)
    show ?B
    using Nil mgp-vertices3[OF Nil(1,3)]
    by(simp add: setFinal-def between-last pre-subdivFace'-def)
  qed
next
show setFinal f  $\in$  ?F using Nil
  by(simp add:pre-subdivFace'-def setFinal-notin-finals minGraphProps11')
qed
next
case (Cons ov ovs)
note IH = Cons(1) and mgp = Cons(2) and pre = Cons(3) and fg = Cons(4)
  and mt = Cons(5)
have  $r \in \mathcal{V} f$  and  $u \in \mathcal{V} f$  and distf: distinct (vertices f)
  using pre by(simp add:pre-subdivFace'-def)+
show ?case
proof (cases ov)
  case None
  have pre': pre-subdivFace' g f r u (Suc n) ovs
    using None pre by (simp add: pre-subdivFace'-None)
  have ovs  $\neq []$  using pre None by (auto simp: pre-subdivFace'-def)
  thus ?thesis using None IH[OF mgp pre' fg] by simp
next
  case (Some v)
  note pre = pre[simplified Some]
  have ruv: before (verticesFrom f r) u v and  $r \neq v$ 
    using pre by(simp add:pre-subdivFace'-def)+
  show ?thesis
  proof (cases  $f \cdot u = v \wedge n = 0$ )
    case True
    have pre': pre-subdivFace' g f r v 0 ovs
      using pre True using [[simp-depth-limit = 5]] by (simp add: pre-subdivFace'-Some2)
    have mt: ovs = []  $\longrightarrow$   $0 = 0 \wedge v = \text{last}$  (verticesFrom f r)
      using pre by(clarsimp simp:pre-subdivFace'-def)
    show ?thesis using Some True IH[OF mgp pre' fg mt]  $\langle r \neq v \rangle$ 
      by(auto simp: between-next-empty[OF distf]
        unroll-between-next2[OF distf  $\langle r \in \mathcal{V} f \rangle \langle u \in \mathcal{V} f \rangle$ ])
    case False
    let ?vs = [countVertices g.. $\text{countVertices } g + n$ ]
    let ?fdg = splitFace g u v f ?vs
    let ?g' = snd(snd ?fdg) and ?f2 = fst(snd ?fdg)
    let ?fvu = between (vertices f) v u

```

```

have False':  $f \cdot u = v \longrightarrow n \neq 0$  using False by auto
have VfVg:  $\mathcal{V} f \subseteq \mathcal{V} g$  using mgp fg
  by (simp add: minGraphProps-def faces-subset-def)
note pre-fdg = pre-subdivFace'-preFaceDiv[OF pre fg False' VfVg]
hence  $u \neq v$  and  $v \in \mathcal{V} f$  and disj:  $\mathcal{V} f \cap \text{set } ?vs = \{\}$ 
  by (unfold pre-splitFace-def) simp+
hence vvs:  $v \notin \text{set } ?vs$  by auto
have vf2: vertices ?f2 = [v] @ ?f2v @ u # ?vs
  by (simp add: split-face-def splitFace-def split-def)
hence betwvf2: between (vertices ?f2) u v = ?vs
  using splitFace-distinct1[OF pre-fdg]
  by (simp add: between-back)
have betrvf2:  $r \neq u \implies$  between (vertices ?f2) r v =
  between (vertices f) r u @ [u] @ ?vs
proof -
  assume r≠u
  have r:  $r \in \text{set } (\text{between } (\text{vertices } f) v u)$ 
    using ⟨r≠u⟩ ⟨r≠v⟩ ⟨u≠v⟩ ⟨v ∈ V f⟩ ⟨r ∈ V f⟩ distf ruw
    by (blast intro: rotate-before-vFrom before-between)
  have between (vertices f) v u =
    between (vertices f) v r @ [r] @ between (vertices f) r u
    using split-between[OF distf ⟨v ∈ V f⟩ ⟨u ∈ V f⟩ r] ⟨r≠v⟩
    by simp
  moreover hence  $v \notin \text{set } (\text{between } (\text{vertices } f) r u)$ 
    using between-not-r1[OF distf, of v u] by simp
  ultimately show ?thesis using vf2 ⟨r≠v⟩ ⟨u≠v⟩ vvs
    by (simp add: between-back between-not-r2[OF distf])
qed
note mgp' = splitFace-holds-minGraphProps[OF pre-fdg mgp]
note f2g = splitFace-add-f21'[OF fg]
note pre' = pre-subdivFace'-Some1[OF pre fg False' VfVg HOL.refl HOL.refl]
from pre-fdg have  $v \in \mathcal{V} f$  and disj:  $\mathcal{V} f \cap \text{set } ?vs = \{\}$ 
  by (unfold pre-splitFace-def, simp)+
have fr:  $?f_2^{-1} \cdot r = f^{-1} \cdot r$ 
proof -
  note pre-split = pre-splitFace-pre-split-face[OF pre-fdg]
  have rinff2:  $r \in \mathcal{V} ?f_2$ 
  proof cases
    assume  $r = u$  thus ?thesis by (simp add: vf2)
  next
    assume  $r \neq u$ 
    hence  $r \in \text{set } ?f_2v$  using distf ⟨v : V f⟩ ⟨r≠v⟩ ⟨r : V f⟩ ruw
      by (blast intro: before-between rotate-before-vFrom)
    thus ?thesis by (simp add: vf2)
  qed
have E2:  $\mathcal{E} ?f_2 = \text{Edges } (u \# ?vs @ [v]) \cup$ 
  Edges (v # ?f2v @ [u])
  by (simp add: splitFace-def split-def
    edges-split-face2[OF pre-split])

```

moreover have $(?f_2^{-1} \cdot r, r) \in \mathcal{E} \ ?f_2$
by(blast intro: prevVertex-in-edges rinf₂
splitFace-distinct1[OF pre-fdg])
moreover have $(?f_2^{-1} \cdot r, r) \notin \text{Edges } (u \# ?vs @ [v])$
proof –
have $r \notin \text{set } ?vs$ **using** $\langle r : \mathcal{V} f \rangle$ **disj by blast**
thus $?thesis$ **using** $\langle r \neq v \rangle$
by(simp add:Edges-Cons Edges-append notinset-notinEdge2) **arith**
qed
ultimately have $(?f_2^{-1} \cdot r, r) \in \text{Edges } (v \# ?fvu @ [u])$ **by blast**
hence $(?f_2^{-1} \cdot r, r) \in \mathcal{E} \ f$ **using** pre-split-face-sym1[OF pre-split]
by(blast intro: Edges-between-edges)
hence eq: $f \cdot (?f_2^{-1} \cdot r) = r$ **and inf:** $?f_2^{-1} \cdot r \in \mathcal{V} \ f$
by(simp add:edges-face-eq)+
have $?f_2^{-1} \cdot r = f^{-1} \cdot (f \cdot (?f_2^{-1} \cdot r))$
using prevVertex-nextVertex[OF distf inf] **by simp**
also have $\dots = f^{-1} \cdot r$ **using eq by simp**
finally show $?thesis$.
qed
hence $mt: ovs = [] \longrightarrow 0 = 0 \wedge v = \text{last } (\text{verticesFrom } ?f_2 \ r)$
using pre' pre **by**(auto simp:pre-subdivFace'-def splitFace-def
split-def last-vFrom)
from IH[OF mgp' pre' f2g mt] $\langle r \neq v \rangle$ **obtain** $f' :: \text{face}$ **where**
 $f: f' \in \text{set } (\text{finals } (\text{subdivFace}' \ ?g' \ ?f_2 \ v \ 0 \ ovs)) - \text{set } (\text{finals } ?g')$
and $ff: (?f_2^{-1} \cdot r, r) \in \mathcal{E} \ f'$
 $|\text{vertices } f'| = |\text{ovs}| + |\text{between } (\text{vertices } ?f_2) \ r \ v| + 2$
by simp blast
show $?thesis$ **(is** $\exists f' \in ?F. ?P \ f')$
proof
show $f' \in ?F$ **using** f pre **Some fg**
by(simp add:False split-def pre-subdivFace'-def)
show $?P \ f'$ **using** $ff \ fr$ **by**(clarsimp simp:betw_{f2} betrv_{f2})
qed
qed
qed
qed

lemma Seed-max-final-ex:

$\exists f \in \text{set } (\text{finals } (\text{Seed } p)). |\text{vertices } f| = \text{maxGon } p$
by (simp add: Seed-def graph-max-final-ex)

lemma max-face-ex: **assumes** $a: \text{Seed}_p \ [\text{next-plane0 } p] \rightarrow^* g$

shows $\exists f \in \text{set } (\text{finals } g). |\text{vertices } f| = \text{maxGon } p$
using a

proof (induct rule: RTranCl-induct)

case refl then show $?case$ **using** Seed-max-final-ex **by** simp
next

case (succs $g \ g'$)

then obtain f **where** $f: f \in \text{set } (\text{finals } g)$ **and** $|\text{vertices } f| = \text{maxGon } p$
by *auto*
moreover from $\text{succs}(1) f$ **have** $f \in \text{set } (\text{finals } g)$ **by** (*rule next-plane0-final-incr*)
ultimately show *?case* **by** *auto*
qed

end

15 Summation Over Lists

theory *ListSum*
imports *ListAux*
begin

primrec *ListSum* :: 'b list \Rightarrow ('b \Rightarrow 'a::comm-monoid-add) \Rightarrow 'a::comm-monoid-add
where

$\text{ListSum } [] f = 0$
 $|\text{ListSum } (l\#ls) f = f l + \text{ListSum } ls f$

syntax *-ListSum* :: idt \Rightarrow 'b list \Rightarrow ('a::comm-monoid-add) \Rightarrow
('a::comm-monoid-add) ($\langle \sum _ \in _ _ \rangle [0, 0, 10] 10$)

syntax-consts *-ListSum* == *ListSum*

translations $\sum_{x \in xs} f == \text{CONST } \text{ListSum } xs (\lambda x. f)$

lemma [*simp*]: $(\sum_{v \in V} 0) = (0::\text{nat})$ **by** (*induct V simp-all*)

lemma *ListSum-compl1*:

$(\sum_{x \in [x \leftarrow xs. \neg P x]} f x) + (\sum_{x \in [x \leftarrow xs. P x]} f x) = (\sum_{x \in xs} (f x::\text{nat}))$
by (*induct xs simp-all*)

lemma *ListSum-compl2*:

$(\sum_{x \in [x \leftarrow xs. P x]} f x) + (\sum_{x \in [x \leftarrow xs. \neg P x]} f x) = (\sum_{x \in xs} (f x::\text{nat}))$
by (*induct xs simp-all*)

lemmas *ListSum-compl* = *ListSum-compl1 ListSum-compl2*

lemma *ListSum-conv-sum*:

$\text{distinct } xs \Longrightarrow \text{ListSum } xs f = \text{sum } f (\text{set } xs)$
by (*induct xs simp-all*)

lemma *listsum-cong*:

$[[xs = ys; \bigwedge y. y \in \text{set } ys \Longrightarrow f y = g y]]$
 $\Longrightarrow \text{ListSum } xs f = \text{ListSum } ys g$

apply *simp*

apply(*erule thin-rl*)
by (*induct ys simp-all*)

lemma *strong-listsum-cong*[*cong*]:
 $\llbracket xs = ys; \bigwedge y. y \in \text{set } ys = \text{simp} \Rightarrow f y = g y \rrbracket$
 $\Rightarrow \text{ListSum } xs f = \text{ListSum } ys g$
by(*auto simp:simp-implies-def intro!:listsum-cong*)

lemma *ListSum-eq* [*trans*]:
 $(\bigwedge v. v \in \text{set } V \Rightarrow f v = g v) \Rightarrow (\sum_{v \in V} f v) = (\sum_{v \in V} g v)$
by(*auto intro!:listsum-cong*)

lemma *ListSum-disj-union*:
 $\text{distinct } A \Rightarrow \text{distinct } B \Rightarrow \text{distinct } C \Rightarrow$
 $\text{set } C = \text{set } A \cup \text{set } B \Rightarrow$
 $\text{set } A \cap \text{set } B = \{\}$ \Rightarrow
 $(\sum_{a \in C} (f a)) = (\sum_{a \in A} f a) + (\sum_{a \in B} (f a :: \text{nat}))$
by (*simp add: ListSum-conv-sum sum.union-disjoint*)

lemma *listsum-const*[*simp*]:
 $(\sum_{x \in xs} k) = \text{length } xs * k$
by (*induct xs (simp-all add: ring-distrib)*)

lemma *ListSum-add*:
 $(\sum_{x \in V} f x) + (\sum_{x \in V} g x) = (\sum_{x \in V} (f x + (g x :: \text{nat})))$
by (*induct V auto*)

lemma *ListSum-le*:
 $(\bigwedge v. v \in \text{set } V \Rightarrow f v \leq g v) \Rightarrow (\sum_{v \in V} f v) \leq (\sum_{v \in V} (g v :: \text{nat}))$
proof (*induct V*)
 case *Nil* **then show** *?case* **by** *simp*
next
 case (*Cons v V*) **then have** $(\sum_{v \in V} f v) \leq (\sum_{v \in V} g v)$ **by** *simp*
 moreover from *Cons* **have** $f v \leq g v$ **by** *simp*
 ultimately show *?case* **by** *simp*
qed

lemma *ListSum1-bound*:
 $a \in \text{set } F \Rightarrow (d a :: \text{nat}) \leq (\sum_{f \in F} d f)$
by (*induct F auto*)

end

16 Tameness

```
theory Tame
imports Graph ListSum
begin
```

16.1 Constants

```
definition squanderTarget :: nat where
  squanderTarget  $\equiv$  15410
```

```
definition excessTCount :: nat where
```

```
  a  $\equiv$  6295
```

```
definition squanderVertex :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat where
```

```
  b p q  $\equiv$  if p = 0  $\wedge$  q = 3 then 6177
    else if p = 0  $\wedge$  q = 4 then 9696
    else if p = 1  $\wedge$  q = 2 then 6557
    else if p = 1  $\wedge$  q = 3 then 6176
    else if p = 2  $\wedge$  q = 1 then 7967
    else if p = 2  $\wedge$  q = 2 then 4116
    else if p = 2  $\wedge$  q = 3 then 12846
    else if p = 3  $\wedge$  q = 1 then 3106
    else if p = 3  $\wedge$  q = 2 then 8165
    else if p = 4  $\wedge$  q = 0 then 3466
    else if p = 4  $\wedge$  q = 1 then 3655
    else if p = 5  $\wedge$  q = 0 then 395
    else if p = 5  $\wedge$  q = 1 then 11354
    else if p = 6  $\wedge$  q = 0 then 6854
    else if p = 7  $\wedge$  q = 0 then 14493
    else squanderTarget
```

```
definition squanderFace :: nat  $\Rightarrow$  nat where
```

```
  d n  $\equiv$  if n = 3 then 0
    else if n = 4 then 2058
    else if n = 5 then 4819
    else if n = 6 then 7120
    else squanderTarget
```

16.2 Separated sets of vertices

A set of vertices V is *separated*, iff the following conditions hold:

2. No two vertices in V are adjacent:

```
definition separated2 :: graph  $\Rightarrow$  vertex set  $\Rightarrow$  bool where
  separated2 g V  $\equiv$   $\forall v \in V. \forall f \in \text{set } (\text{facesAt } g v). f \cdot v \notin V$ 
```

3. No two vertices lie on a common quadrilateral:

definition $separated_3 :: graph \Rightarrow vertex\ set \Rightarrow bool$ **where**

$separated_3\ g\ V \equiv$

$\forall v \in V. \forall f \in set\ (facesAt\ g\ v). |vertices\ f| \leq 4 \longrightarrow \forall f \cap V = \{v\}$

A set of vertices is called *separated*, iff no two vertices are adjacent or lie on a common quadrilateral:

definition $separated :: graph \Rightarrow vertex\ set \Rightarrow bool$ **where**

$separated\ g\ V \equiv separated_2\ g\ V \wedge separated_3\ g\ V$

16.3 Admissible weight assignments

A weight assignment $w :: face \Rightarrow nat$ assigns a natural number to every face.

We formalize the admissibility requirements as follows:

definition $admissible_1 :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$ **where**

$admissible_1\ w\ g \equiv \forall f \in \mathcal{F}\ g. d\ |vertices\ f| \leq w\ f$

definition $admissible_2 :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$ **where**

$admissible_2\ w\ g \equiv$

$\forall v \in \mathcal{V}\ g. except\ g\ v = 0 \longrightarrow b\ (tri\ g\ v)\ (quad\ g\ v) \leq (\sum_{f \in facesAt\ g\ v} w\ f)$

definition $admissible_3 :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$ **where**

$admissible_3\ w\ g \equiv$

$\forall v \in \mathcal{V}\ g. vertextype\ g\ v = (5,0,1) \longrightarrow (\sum_{f \in filter\ triangle\ (facesAt\ g\ v)} w(f)) \geq$
a

Finally we define admissibility of weights functions.

definition $admissible :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$ **where**

$admissible\ w\ g \equiv admissible_1\ w\ g \wedge admissible_2\ w\ g \wedge admissible_3\ w\ g$

16.4 Tameness

definition $tame9a :: graph \Rightarrow bool$ **where**

$tame9a\ g \equiv \forall f \in \mathcal{F}\ g. 3 \leq |vertices\ f| \wedge |vertices\ f| \leq 6$

definition $tame10 :: graph \Rightarrow bool$ **where**

$tame10\ g = (let\ n = countVertices\ g\ in\ 13 \leq n \wedge n \leq 15)$

definition $tame10ub :: graph \Rightarrow bool$ **where**

$tame10ub\ g = (countVertices\ g \leq 15)$

definition $tame11a :: graph \Rightarrow bool$ **where**

$tame11a\ g = (\forall v \in \mathcal{V}\ g. 3 \leq degree\ g\ v)$

definition $tame11b :: graph \Rightarrow bool$ **where**

$tame11b\ g = (\forall v \in \mathcal{V}\ g. degree\ g\ v \leq (if\ except\ g\ v = 0\ then\ 7\ else\ 6))$

definition *tame12o* :: *graph* \Rightarrow *bool* **where**

tame12o *g* =

$(\forall v \in \mathcal{V} \ g. \text{except } g \ v \neq 0 \wedge \text{degree } g \ v = 6 \longrightarrow \text{vertextype } g \ v = (5,0,1))$

7. There exists an admissible weight assignment of total weight less than the target:

definition *tame13a* :: *graph* \Rightarrow *bool* **where**

tame13a *g* = $(\exists w. \text{admissible } w \ g \wedge (\sum_{f \in \text{faces } g} w \ f) < \text{squanderTarget})$

Finally we define the notion of tameness.

definition *tame* :: *graph* \Rightarrow *bool* **where**

tame *g* \equiv *tame9a* *g* \wedge *tame10* *g* \wedge *tame11a* *g* \wedge *tame11b* *g* \wedge *tame12o* *g* \wedge *tame13a* *g*

theory *Plane1Props*

imports *Plane1* *PlaneProps* *Tame*

begin

lemma *next-plane-subset*:

$\forall f \in \mathcal{F} \ g. \text{vertices } f \neq [] \implies$

$\text{set } (\text{next-plane}_p \ g) \subseteq \text{set } (\text{next-plane0}_p \ g)$

apply(*clarsimp simp:next-plane0-def next-plane-def minimalFace-def finalGraph-def*)

apply(*rule-tac x = minimal (size \circ vertices) (nonFinals g) in bexI*)

apply(*rule-tac x = minimalVertex g (minimal (size \circ vertices) (nonFinals g)) in bexI*)

apply *blast*

apply(*subgoal-tac $\forall f \in \text{set } (\text{nonFinals } g). \text{vertices } f \neq []$*)

apply(*simp add:minimalVertex-def*)

apply(*simp add:nonFinals-def*)

apply *simp*

done

lemma *mgp-next-plane0-if-next-plane*:

$\text{minGraphProps } g \implies g [\text{next-plane}_p] \rightarrow g' \implies g [\text{next-plane0}_p] \rightarrow g'$

using *next-plane-subset* **by**(*blast dest: mgp-vertices-nonempty*)

lemma *inv-inv-next-plane: invariant inv next-plane_p*

apply(*rule inv-subset[OF inv-inv-next-plane0]*)

apply(*blast dest: mgp-next-plane0-if-next-plane[OF inv-mgp]*)

done

end

17 Enumeration of Tame Plane Graphs

theory *Generator*

```

imports Plane1 Tame
begin

```

```

definition faceSquanderLowerBound :: graph ⇒ nat where
faceSquanderLowerBound g ≡ ∑ f ∈ finals g d |vertices f|

```

```

definition d3-const :: nat where
d3-const == d 3

```

```

definition d4-const :: nat where
d4-const == d 4

```

```

definition excessAtType :: nat ⇒ nat ⇒ nat ⇒ nat where
excessAtType t q e ≡
  if e = 0 then if 7 < t + q then squanderTarget
    else b t q - t * d3-const - q * d4-const
  else if t + q + e ≠ 6 then 0
    else if t=5 then a else squanderTarget

```

```

declare d3-const-def[simp] d4-const-def[simp]

```

```

definition ExcessAt :: graph ⇒ vertex ⇒ nat where
ExcessAt g v ≡ if ¬ finalVertex g v then 0
  else excessAtType (tri g v) (quad g v) (except g v)

```

```

definition ExcessTable :: graph ⇒ vertex list ⇒ (vertex × nat) list where
ExcessTable g vs ≡
  [(v, ExcessAt g v). v ← [v ← vs. 0 < ExcessAt g v]]

```

Implementation:

```

lemma [code]:
  ExcessTable g =
    List.map-filter (λv. let e = ExcessAt g v in if 0 < e then Some (v, e) else None)
by (rule ext) (simp add: ExcessTable-def map-filter-def)

```

```

definition deleteAround :: graph ⇒ vertex ⇒ (vertex × nat) list ⇒ (vertex ×
nat) list where
deleteAround g v ps ≡
  let fs = facesAt g v;
  ws = ⋃ f ∈ fs if |vertices f| = 4 then [f.v, f2.v] else [f.v] in
  removeKeyList ws ps

```

Implementation:

```

lemma [code]: deleteAround g v ps =

```

(let vs = (λf . let n = $f \cdot v$
in if |vertices f| = 4 then [n, f·n] else [n])
in removeKeyList (concat(map vs (facesAt g v))) ps)
by (simp only: concat-map-singleton Let-def deleteAround-def nextV2)

lemma length-deleteAround: length (deleteAround g v ps) \leq length ps
by (auto simp only: deleteAround-def length-removeKeyList Let-def)

function ExcessNotAtRec :: (nat, nat) table \Rightarrow graph \Rightarrow nat **where**
ExcessNotAtRec [] = (λg . 0)
| ExcessNotAtRec ((x, y)#ps) = (λg . max (ExcessNotAtRec ps g)
(y + ExcessNotAtRec (deleteAround g x ps) g))
by pat-completeness auto
termination by (relation measure size)
(auto simp add: length-deleteAround less-Suc-eq-le)

definition ExcessNotAt :: graph \Rightarrow vertex option \Rightarrow nat **where**
ExcessNotAt g v-opt \equiv
let ps = ExcessTable g (vertices g) in
case v-opt of None \Rightarrow ExcessNotAtRec ps g
| Some v \Rightarrow ExcessNotAtRec (deleteAround g v ps) g

definition squanderLowerBound :: graph \Rightarrow nat **where**
squanderLowerBound g \equiv faceSquanderLowerBound g + ExcessNotAt g None

definition is-tame13a :: graph \Rightarrow bool **where**
is-tame13a g \equiv squanderLowerBound g < squanderTarget

definition notame :: graph \Rightarrow bool **where**
notame g \equiv \neg (tame10ub g \wedge tame11b g)

definition notame7 :: graph \Rightarrow bool **where**
notame7 g \equiv \neg (tame10ub g \wedge tame11b g \wedge is-tame13a g)

definition generatePolygonTame :: nat \Rightarrow vertex \Rightarrow face \Rightarrow graph \Rightarrow graph list
where
generatePolygonTame n v f g \equiv
let
enumeration = enum n |vertices f|;
enumeration = [is \leftarrow enumeration. \neg containsDuplicateEdge g f v is];
vertexLists = [indexToVertexList f v is. is \leftarrow enumeration]
in
[g' \leftarrow [subdivFace g f vs. vs \leftarrow vertexLists] . \neg notame g']

definition polysizes :: nat \Rightarrow graph \Rightarrow nat list **where**
polysizes p g \equiv
let lb = squanderLowerBound g in

$[n \leftarrow [3 \dots < \text{Suc}(\text{maxGon } p)]. \text{lb} + \text{d } n < \text{squanderTarget}]$

definition $\text{next-tame0} :: \text{nat} \Rightarrow \text{graph} \Rightarrow \text{graph list } (\langle \text{next'-tame0-} \rangle)$ **where**
 $\text{next-tame0}_p g \equiv$
 let $fs = \text{nonFinals } g$ in
 if $fs = []$ then []
 else let $f = \text{minimalFace } fs$; $v = \text{minimalVertex } g f$
 in $\bigsqcup i \in \text{polysizes } p g \text{ generatePolygonTame } i v f g$

Extensionally, next-tame0 is just $\text{filter } P \circ \text{next-plane}_p$ for some suitable P . But efficiency suffers considerably if we first create many graphs and then filter out the ones not in polysizes .

end

18 Tame Properties

theory TameProps
imports Tame RTranCl
begin

lemma $\text{length-disj-filter-le}: \forall x \in \text{set } xs. \neg(P x \wedge Q x) \Longrightarrow$
 $\text{length}(\text{filter } P xs) + \text{length}(\text{filter } Q xs) \leq \text{length } xs$
by($\text{induct } xs$) auto

lemma $\text{tri-quad-le-degree}: \text{tri } g v + \text{quad } g v \leq \text{degree } g v$

proof –

let $?fins = [f \leftarrow \text{facesAt } g v . \text{final } f]$
 have $\text{tri } g v + \text{quad } g v =$
 $|[f \leftarrow ?fins . \text{triangle } f]| + |[f \leftarrow ?fins . |\text{vertices } f| = 4]|$
by($\text{simp add:tri-def quad-def}$)
 also have $\dots \leq |[f \leftarrow \text{facesAt } g v . \text{final } f]|$
by($\text{rule length-disj-filter-le simp}$)
 also have $\dots \leq |\text{facesAt } g v|$ **by**($\text{rule length-filter-le}$)
finally show $?thesis$ **by**($\text{simp add:degree-def}$)
qed

lemma $\text{faceCountMax-bound}$:

$\llbracket \text{tame } g; v \in \mathcal{V} g \rrbracket \Longrightarrow \text{tri } g v + \text{quad } g v \leq 7$
using $\text{tri-quad-le-degree}$ [$\text{of } g v$]
by($\text{auto simp:tame-def tame11b-def split:if-split-asm}$)

lemma filter-tame-succs :

assumes $\text{invP}: \text{invariant } P \text{ succs}$ **and** $\text{fin}: \bigwedge g. \text{final } g \Longrightarrow \text{succs } g = []$
and $\text{ok-untame}: \bigwedge g. P g \Longrightarrow \neg \text{ok } g \Longrightarrow \text{final } g \wedge \neg \text{tame } g$
and $\text{gg}': g [\text{succs}] \rightarrow^* g'$
shows $P g \Longrightarrow \text{final } g' \Longrightarrow \text{tame } g' \Longrightarrow g [\text{filter ok} \circ \text{succs}] \rightarrow^* g'$

```

using gg'
proof (induct rule:RTranCl.induct)
  case refl show ?case by(rule RTranCl.refl)
next
  case (succs h h' h'')
  hence P h' using invP by(unfold invariant-def) blast
  show ?case
  proof cases
    assume ok h'
    thus ?thesis using succs ⟨P h'⟩ by(fastforce intro:RTranCl.succs)
  next
    assume ¬ ok h' note fin-tame = ok-untame[OF ⟨P h'⟩ ⟨¬ ok h'⟩]
    have h'' = h' using fin-tame
      by(rule-tac RTranCl.cases[OF succs(2)])(auto simp:fin)
    hence False using fin-tame succs by fast
    thus ?case ..
  qed
qed

```

definition *untame* :: (graph ⇒ bool) ⇒ bool **where**
untame P ≡ ∀ g. final g ∧ P g → ¬ tame g

lemma *filterout-untame-succs*:

```

assumes invP: invariant P f and invPU: invariant (λg. P g ∧ U g) f
and untame: untame(λg. P g ∧ U g)
and new-untame: ∧g g'. [ P g; g' ∈ set(f g); g' ∉ set(f' g) ] ⇒ U g'
and gg': g [f]→* g'
shows P g ⇒ final g' ⇒ tame g' ⇒ g [f']→* g'
using gg'
proof (induct rule:RTranCl.induct)
  case refl show ?case by(rule RTranCl.refl)
next
  case (succs h h' h'')
  hence Ph': P h' using invP by(unfold invariant-def) blast
  show ?case
  proof cases
    assume h' ∈ set(f' h)
    thus ?thesis using succs Ph' by(blast intro:RTranCl.succs)
  next
    assume h' ∉ set(f' h)
    with succs(4) succs(1) have U h' by (rule new-untame)
    hence False using Ph' RTranCl-inv[OF invPU] untame succs
      by (unfold untame-def) fast
    thus ?case ..
  qed
qed

```

end

19 Neglectable Final Graphs

theory *TameEnum*
imports *Generator*
begin

definition *is-tame* :: *graph* \Rightarrow *bool* **where**
is-tame *g* \equiv *tame10* *g* \wedge *tame11a* *g* \wedge *tame12o* *g* \wedge *is-tame13a* *g*

definition *next-tame* :: *nat* \Rightarrow *graph* \Rightarrow *graph list* (\langle *next'-tame* \rangle) **where**
*next-tame*_{*p*} \equiv *filter* ($\lambda g. \neg$ *final* *g* \vee *is-tame* *g*) \circ *next-tame0*_{*p*}

definition *TameEnumP* :: *nat* \Rightarrow *graph set* (\langle *TameEnum* \rangle) **where**
*TameEnum*_{*p*} \equiv {*g*. *Seed*_{*p*} [*next-tame*_{*p*}] \rightarrow^* *g* \wedge *final* *g*}

definition *TameEnum* :: *graph set* **where**
TameEnum \equiv $\bigcup_{p \leq 3} .$ *TameEnum*_{*p*}

end

20 Properties of Lower Bound Machinery

theory *ScoreProps*
imports *ListSum* *TameEnum* *PlaneProps* *TameProps*
begin

lemma *deleteAround-empty[simp]*: *deleteAround* *g* *a* [] = []
by (*simp* *add*: *deleteAround-def*)

lemma *deleteAroundCons*:
deleteAround *g* *a* (*p*#*ps*) =
 (*if* *fst* *p* \in {*v*. $\exists f \in$ *set* (*facesAt* *g* *a*)}.
 (*length* (*vertices* *f*) = 4) \wedge *v* \in {*f* \cdot *a*, *f* \cdot (*f* \cdot *a*)}
 \vee (*length* (*vertices* *f*) \neq 4) \wedge (*v* = *f* \cdot *a*)}
 then *deleteAround* *g* *a* *ps*
 else *p*#*deleteAround* *g* *a* *ps*)
by (*fastforce* *simp*: *nextV2* *deleteAround-def*)

lemma *deleteAround-subset*: *set* (*deleteAround* *g* *a* *ps*) \subseteq *set* *ps*
by (*simp* *add*: *deleteAround-def*)

lemma *distinct-deleteAround*: *distinct* (*map* *fst* *ps*) \implies
 distinct (*map* *fst* (*deleteAround* *g* (*fst* (*a*, *b*)) *ps*))
proof (*induct* *ps*)

```

case Nil then show ?case by simp
next
case (Cons p ps)
then have fst p  $\notin$  fst ' set ps by simp
moreover have set (deleteAround g a ps)  $\subseteq$  set ps
  by (rule deleteAround-subset)
ultimately have fst p  $\notin$  fst ' set (deleteAround g a ps) by auto

moreover from Cons have distinct (map fst ps) by simp
then have distinct (map fst (deleteAround g (fst (a, b)) ps))
  by (rule Cons)
ultimately show ?case by (simp add: deleteAroundCons)
qed

```

```

definition deleteAround' :: graph  $\Rightarrow$  vertex  $\Rightarrow$  (vertex  $\times$  nat) list  $\Rightarrow$ 
  (vertex  $\times$  nat) list where
  deleteAround' g v ps  $\equiv$ 
    let fs = facesAt g v;
        vs = ( $\lambda$ f. let n1 = f  $\cdot$  v;
                  n2 = f  $\cdot$  n1 in
              if length (vertices f) = 4 then [n1, n2] else [n1]);
        ws = concat (map vs fs) in
    removeKeyList ws ps

```

```

lemma deleteAround-eq: deleteAround g v ps = deleteAround' g v ps
apply (auto simp add: deleteAround-def deleteAround'-def split: if-split-asm)
apply (unfold nextV2[THEN eq-reflection], simp)
done

```

```

lemma deleteAround-nextVertex:
  f  $\in$  set (facesAt g a)  $\implies$ 
  (f  $\cdot$  a, b)  $\notin$  set (deleteAround g a ps)
by (auto simp add: deleteAround-eq deleteAround'-def removeKeyList-eq)

```

```

lemma deleteAround-nextVertex-nextVertex:
  f  $\in$  set (facesAt g a)  $\implies$  |vertices f| = 4  $\implies$ 
  (f  $\cdot$  (f  $\cdot$  a), b)  $\notin$  set (deleteAround g a ps)
by (auto simp add: deleteAround-eq deleteAround'-def removeKeyList-eq)

```

```

lemma deleteAround-prevVertex:
  minGraphProps g  $\implies$  a :  $\mathcal{V}$  g  $\implies$  f  $\in$  set (facesAt g a)  $\implies$ 
  (f-1  $\cdot$  a, b)  $\notin$  set (deleteAround g a ps)
proof -
  assume a: minGraphProps g a :  $\mathcal{V}$  g f  $\in$  set (facesAt g a)
  have (f-1  $\cdot$  a, a)  $\in$   $\mathcal{E}$  f using a
  by(blast intro:prevVertex-in-edges minGraphProps)

```

then obtain $f' :: \text{face}$ **where** $f': f' \in \text{set}(\text{facesAt } g \ a)$
and $e: (a, f^{-1} \cdot a) \in \mathcal{E} \ f'$
using a **by** $(\text{blast } \text{dest}:\text{mgp-edge-face-ex})$
have $(f' \cdot a, b) \notin \text{set}(\text{deleteAround } g \ a \ ps)$ **using** f'
by $(\text{auto } \text{simp } \text{add}:\text{deleteAround-eq } \text{deleteAround'-def } \text{removeKeyList-eq})$
moreover have $f' \cdot a = f^{-1} \cdot a$
using e **by** $(\text{simp } \text{add}:\text{edges-face-eq})$
ultimately show $?thesis$ **by** simp
qed

lemma $\text{deleteAround-separated}$:

assumes $\text{mgp}: \text{minGraphProps } g$ **and** $\text{fin}: \text{final } g$ **and** $\text{ag}: a : \mathcal{V} \ g$ **and** $4: |\text{vertices } f| \leq 4$

and $f: f \in \text{set}(\text{facesAt } g \ a)$

shows $\mathcal{V} \ f \cap \text{set}[\text{fst } p. p \leftarrow \text{deleteAround } g \ a \ ps] \subseteq \{a\}$ **(is ?A)**

proof –

note $MGP = \text{mgp } ag \ f$

have $af: a \in \mathcal{V} \ f$ **using** MGP **by** $(\text{blast } \text{intro}:\text{minGraphProps})$

have $2 < |\text{vertices } f|$ **using** MGP **by** $(\text{blast } \text{intro}:\text{minGraphProps})$

with 4 **have** $|\text{vertices } f| = 3 \vee |\text{vertices } f| = 4$ **by** arith

then show $?A$

proof

assume $3: |\text{vertices } f| = 3$

show $?A$

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

assume $\neg ?A$

then obtain b **where** $b1: b \neq a \ b \in \mathcal{V} \ f$

$b \in \text{set}(\text{map } \text{fst}(\text{deleteAround } g \ a \ ps))$ **by** auto

from MGP **have** $d: \text{distinct}(\text{vertices } f)$

by $(\text{blast } \text{intro}:\text{minGraphProps})$

with $af \ 3$ **have** $\mathcal{V} \ f = \{a, f \cdot a, f \cdot (f \cdot a)\}$

by $(\text{rule-tac } \text{vertices-triangle})$

also from $d \ af \ 3$ **have**

$f \cdot (f \cdot a) = f^{-1} \cdot a$

by $(\text{simp } \text{add}:\text{triangle-nextVertex-prevVertex})$

finally have

$b \in \{f \cdot a, f^{-1} \cdot a\}$

using $b1$ **by** simp

with MGP **have** $b \notin \text{set}(\text{map } \text{fst}(\text{deleteAround } g \ a \ ps))$

using $\text{deleteAround-nextVertex } \text{deleteAround-prevVertex}$ **by** auto

then show False **by** $\text{contradiction}(\text{rule } b1)$

qed

next

assume $4: |\text{vertices } f| = 4$

show $?A$

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

assume $\neg ?A$

then obtain b **where** $b1: b \neq a \ b \in \mathcal{V} \ f$


```

    b ∈ set (map fst (deleteAround g a ps)) by auto
from MGP have d: distinct (vertices f) by (blast intro: minGraphProps)
with af 4 have V f = {a, f · a, f · (f · a), f · (f · (f · a))}
    by (rule-tac vertices-quad)
also from d af 4 have f · (f · (f · a)) = f-1 · a
    by (simp add: quad-nextVertex-prevVertex)
finally have b ∈ {f · a, f · (f · a), f-1 · a}
using b1 by simp
with MGP 4 have b ∉ set (map fst (deleteAround g a ps))
using deleteAround-nextVertex deleteAround-prevVertex
    deleteAround-nextVertex-nextVertex by auto
then show False by contradiction (rule b1)
qed
qed
qed

```

```

lemma [iff]: separated g {}
by (simp add: separated-def separated2-def separated3-def)

```

lemma separated-insert:

```

assumes mgp: minGraphProps g and a: a ∈ V g
    and Vg: V ⊆ V g
    and ps: separated g V
    and s2: (∧f. f ∈ set (facesAt g a) ⇒ f · a ∉ V)
    and s3: (∧f. f ∈ set (facesAt g a) ⇒
        |vertices f| ≤ 4 ⇒ V f ∩ V ⊆ {a})
shows separated g (insert a V)
proof (simp add: separated-def separated2-def separated3-def,
    intro conjI ballI impI)
    fix f assume f: f ∈ set (facesAt g a)
    then show f · a ≠ a by (rule mgp-facesAt-no-loop[OF mgp a])
    from f show f · a ∉ V by (rule s2)
next
    fix f v assume v: f ∈ set (facesAt g v) and vV: v ∈ V
    have v : V g using vV Vg by blast
    show f · v ≠ a
    proof
        assume f: f · v = a
        then obtain f' where f': f' ∈ set(facesAt g a) and v: f' · a = v
        using mgp-nextVertex-face-ex2[OF mgp ⟨v : V g⟩ v] by blast
        have f' · a ∈ V using v vV by simp
        with f' s2 show False by blast
    qed
    from ps v vV show f · v ∉ V
    by (simp add: separated-def separated2-def)
next
    fix f assume f: f ∈ set (facesAt g a) |vertices f| ≤ 4
    then have V f ∩ V ⊆ {a} by (rule s3)

```

```

moreover from mgp a f have  $a \in \mathcal{V} f$  by (blast intro:minGraphProps)
ultimately show  $\mathcal{V} f \cap \text{insert } a V = \{a\}$  by auto
next
fix v f
assume  $a: v \in V f \in \text{set } (\text{facesAt } g v)$ 
   $|\text{vertices } f| \leq 4$ 
with ps have  $v: \mathcal{V} f \cap V = \{v\}$ 
  by (simp add: separated-def separated3-def)
have  $v: \mathcal{V} g$  using a Vg by blast
show  $\mathcal{V} f \cap \text{insert } a V = \{v\}$ 
proof cases
  assume  $a = v$ 
  with v mgp a show ?thesis by (blast intro:minGraphProps)
next
assume  $n: a \neq v$ 
have  $a \notin \mathcal{V} f$ 
proof
  assume  $a2: a \in \mathcal{V} f$ 
  with mgp a (v : V g) have  $f \in \mathcal{F} g$  by (blast intro:minGraphProps)
  with mgp a2 have  $f \in \text{set } (\text{facesAt } g a)$  by (blast intro:minGraphProps)
  with a have  $\mathcal{V} f \cap V \subseteq \{a\}$  by (simp add: s3)
  with v have  $a = v$  by auto
  with n show False by auto
qed
with a v show  $\mathcal{V} f \cap \text{insert } a V = \{v\}$  by blast
qed
qed

```

```

function ExcessNotAtRecList :: (vertex, nat) table  $\Rightarrow$  graph  $\Rightarrow$  vertex list where
  ExcessNotAtRecList [] = ( $\lambda g. []$ )
  | ExcessNotAtRecList ((x, y) # ps) = ( $\lambda g.$ 
    let l1 = ExcessNotAtRecList ps g;
    l2 = ExcessNotAtRecList (deleteAround g x ps) g in
    if ExcessNotAtRec ps g
       $\leq y + \text{ExcessNotAtRec } (\text{deleteAround } g x ps) g$ 
    then  $x \# l2$  else l1)
by pat-completeness auto
termination by (relation measure size)
  (auto simp add: less-Suc-eq-le length-deleteAround)

```

```

lemma isTable-deleteAround:
  isTable E vs ((a,b)#ps)  $\implies$  isTable E vs (deleteAround g a ps)
by (rule isTable-subset, rule deleteAround-subset,
  rule isTable-Cons)

```

```

lemma ListSum-ExcessNotAtRecList:
  isTable E vs ps  $\implies$  ExcessNotAtRec ps g
  = ( $\sum p \in \text{ExcessNotAtRecList } ps g E p$ ) (is ?T ps  $\implies$  ?P ps)

```

```

proof (induct ps rule: ExcessNotAtRecList.induct)
  case 1 show ?case by simp
next
  case (2 a b ps)
  from 2 have prem: ?T ((a,b)#ps) by blast
  then have E: b = E a by (simp add: isTable-eq)
  from 2 have hyp1: ?T (deleteAround g a ps)  $\implies$ 
    ?P (deleteAround g a ps) by blast
  from 2 have hyp2: ?T ps  $\implies$  ?P ps by blast
  have H1: ?P (deleteAround g a ps)
    by (rule hyp1, rule isTable-deleteAround) (rule prem)
  have H2: ?P ps by (rule hyp2, rule isTable-Cons, rule prem)
  show ?P ((a,b)#ps)
proof cases
  assume
    ExcessNotAtRec ps g
     $\leq$  b + ExcessNotAtRec (deleteAround g a ps) g
  with H1 E show ?thesis
    by (simp add: max-def split: if-split-asm)
next
  assume  $\neg$  ExcessNotAtRec ps g
     $\leq$  b + ExcessNotAtRec (deleteAround g a ps) g
  with H2 E show ?thesis
    by (simp add: max-def split: if-split-asm)
qed
qed

lemma ExcessNotAtRecList-subset:
  set (ExcessNotAtRecList ps g)  $\subseteq$  set [fst p. p  $\leftarrow$  ps] (is ?P ps)
proof (induct ps rule: ExcessNotAtRecList.induct)
  case 1 show ?case by simp
next
  case (2 a b ps)
  presume H1: ?P (deleteAround g a ps)
  presume H2: ?P ps
  show ?P ((a, b) # ps)
proof cases
  assume a: ExcessNotAtRec ps g
     $\leq$  b + ExcessNotAtRec (deleteAround g a ps) g
  have set (deleteAround g a ps)  $\subseteq$  set ps
    by (simp add: deleteAround-subset)
  then have
    fst ' set (deleteAround g a ps)  $\subseteq$  insert a (fst ' set ps)
    by blast
  with a H1 show ?thesis by (simp)
next
  assume  $\neg$  ExcessNotAtRec ps g
     $\leq$  b + ExcessNotAtRec (deleteAround g a ps) g
  with H2 show ?thesis by (auto)

```

qed
qed simp

lemma separated-ExcessNotAtRecList:

$\text{minGraphProps } g \implies \text{final } g \implies \text{isTable } E \text{ (vertices } g) \text{ } ps \implies$
 $\text{separated } g \text{ (set (ExcessNotAtRecList } ps \text{ } g))$

proof –

assume $\text{fin: final } g$ **and** $\text{mgp: minGraphProps } g$

show

$\text{isTable } E \text{ (vertices } g) \text{ } ps \implies \text{separated } g \text{ (set (ExcessNotAtRecList } ps \text{ } g))$
 $(\text{is } ?T \text{ } ps \implies ?P \text{ } ps)$

proof (*induct rule: ExcessNotAtRec.induct*)

case 1 show $?case$ **by** *simp*

next

case $(2 \ a \ b \ ps)$

from 2 have $\text{prem: } ?T \ ((a,b)\#ps)$ **by** *blast*

then have $E: b = E \ a$ **by** (*simp add: isTable-eq*)

have $a : \mathcal{V} \ g$ **using** prem **by** (*auto simp: isTable-def*)

from 2 have $\text{hyp1: } ?T \ (\text{deleteAround } g \ a \ ps) \implies$

$?P \ (\text{deleteAround } g \ a \ ps)$ **by** *blast*

from 2 have $\text{hyp2: } ?T \ ps \implies ?P \ ps$ **by** *blast*

have $H1: ?P \ (\text{deleteAround } g \ a \ ps)$

by (*rule hyp1, rule isTable-deleteAround*) (*rule prem*)

have $H2: ?P \ ps$ **by** (*rule hyp2, rule isTable-Cons*) (*rule prem*)

show $?P \ ((a,b)\#ps)$

proof *cases*

assume $c: \text{ExcessNotAtRec } ps \ g$

$\leq b + \text{ExcessNotAtRec } (\text{deleteAround } g \ a \ ps) \ g$

have *separated* g

$(\text{insert } a \ (\text{set } (\text{ExcessNotAtRecList } (\text{deleteAround } g \ a \ ps) \ g)))$

proof (*rule separated-insert[OF mgp]*)

from prem **show** $a \in \text{set } (\text{vertices } g)$ **by** (*auto simp add: isTable-def*)

show $\text{set } (\text{ExcessNotAtRecList } (\text{deleteAround } g \ a \ ps) \ g) \subseteq \mathcal{V} \ g$

proof–

have $\text{set } (\text{ExcessNotAtRecList } (\text{deleteAround } g \ a \ ps) \ g) \subseteq$
 $\text{set } (\text{map } \text{fst } (\text{deleteAround } g \ a \ ps))$

by (*rule ExcessNotAtRecList-subset[simplified concat-map-singleton]*)

also have $\dots \subseteq \text{set } (\text{map } \text{fst } ps)$

using *deleteAround-subset* **by** *fastforce*

finally show $?thesis$ **using** prem **by** (*auto simp: isTable-def*)

qed

from $H1$

show $pS: \text{separated } g$

$(\text{set } (\text{ExcessNotAtRecList } (\text{deleteAround } g \ a \ ps) \ g))$

by *simp*

fix f **assume** $f: f \in \text{set } (\text{facesAt } g \ a)$

then have
 $f \cdot a \notin \text{set } [\text{fst } p. p \leftarrow \text{deleteAround } g \ a \ ps]$
by (*auto simp add: facesAt-def deleteAround-eq deleteAround'-def*
removeKeyList-eq split: if-split-asm)
moreover
have $\text{set } (\text{ExcessNotAtRecList } (\text{deleteAround } g \ a \ ps) \ g)$
 $\subseteq \text{set } [\text{fst } p. p \leftarrow \text{deleteAround } g \ a \ ps]$
by (*rule ExcessNotAtRecList-subset*)
ultimately
show $f \cdot a$
 $\notin \text{set } (\text{ExcessNotAtRecList } (\text{deleteAround } g \ a \ ps) \ g)$
by *auto*
assume $|\text{vertices } f| \leq 4$
from *this* **have** $\text{set } (\text{vertices } f)$
 $\cap \text{set } [\text{fst } p. p \leftarrow \text{deleteAround } g \ a \ ps] \subseteq \{a\}$
by (*rule deleteAround-separated[OF mgp fin ‹a : V g›]*)
moreover
have $\text{set } (\text{ExcessNotAtRecList } (\text{deleteAround } g \ a \ ps) \ g)$
 $\subseteq \text{set } [\text{fst } p. p \leftarrow \text{deleteAround } g \ a \ ps]$
by (*rule ExcessNotAtRecList-subset*)
ultimately
show $\text{set } (\text{vertices } f)$
 $\cap \text{set } (\text{ExcessNotAtRecList } (\text{deleteAround } g \ a \ ps) \ g) \subseteq \{a\}$
by *blast*
qed
with *H1 E c* **show** *?thesis* **by** (*simp*)
next
assume $\neg \text{ExcessNotAtRec } ps \ g$
 $\leq b + \text{ExcessNotAtRec } (\text{deleteAround } g \ a \ ps) \ g$
with *H2 E* **show** *?thesis* **by** *simp*
qed
qed
qed

lemma *isTable-ExcessTable*:
 $\text{isTable } (\lambda v. \text{ExcessAt } g \ v) \ vs \ (\text{ExcessTable } g \ vs)$
by (*auto simp add: isTable-def ExcessTable-def ExcessAt-def*)

lemma *ExcessTable-subset*:
 $\text{set } (\text{map } \text{fst } (\text{ExcessTable } g \ vs)) \subseteq \text{set } vs$
by (*induct vs*) (*auto simp add: ExcessTable-def*)

lemma *distinct-ExcessNotAtRecList*:
 $\text{distinct } (\text{map } \text{fst } ps) \implies \text{distinct } (\text{ExcessNotAtRecList } ps \ g)$
(is ?T ps \implies ?P ps)
proof (*induct rule: ExcessNotAtRec.induct*)
case 1 **show** *?case* **by** *simp*
next
case (2 *a b ps*)

from 2 **have** $prem: ?T ((a,b)\#ps)$ **by** *blast*
then have $a \notin set (map\ fst\ ps)$ **by** *simp*
from 2 **have** $hyp1: ?T (deleteAround\ g\ a\ ps) \implies$
 $?P (deleteAround\ g\ a\ ps)$ **by** *blast*
from 2 **have** $hyp2: ?T\ ps \implies ?P\ ps$ **by** *blast*
from 2 **have** $?T\ ps$ **by** *simp*
then have $H1: ?P (deleteAround\ g\ a\ ps)$
by (*rule-tac hyp1*) (*rule distinct-deleteAround [simplified]*)
from $prem$ **have** $H2: ?P\ ps$
by (*rule-tac hyp2*) *simp*

have $a \notin set (ExcessNotAtRecList (deleteAround\ g\ a\ ps)\ g)$
proof
assume $a \in set (ExcessNotAtRecList (deleteAround\ g\ a\ ps)\ g)$
also have $set (ExcessNotAtRecList (deleteAround\ g\ a\ ps)\ g)$
 $\subseteq set [fst\ p.\ p \leftarrow deleteAround\ g\ a\ ps]$
by (*rule ExcessNotAtRecList-subset*)
also have $set (deleteAround\ g\ a\ ps) \subseteq set\ ps$
by (*rule deleteAround-subset*)
then have $set [fst\ p.\ p \leftarrow deleteAround\ g\ a\ ps]$
 $\subseteq set [fst\ p.\ p \leftarrow ps]$ **by** *auto*
finally have $a \in set (map\ fst\ ps)$ **by** *simp*
with a **show** *False* **by** *contradiction*
qed
with $H1\ H2$ **show** $?P ((a,b)\#ps)$
by (*simp add: ExcessNotAtRecList-subset*)
qed

primrec *ExcessTable-cont* ::
 $(vertex \Rightarrow nat) \Rightarrow vertex\ list \Rightarrow (vertex \times nat)\ list$
where
 $ExcessTable-cont\ ExcessAtPG\ [] = [] \mid$
 $ExcessTable-cont\ ExcessAtPG\ (v\#\ vs) =$
 $(let\ vi = ExcessAtPG\ v\ in$
 $if\ 0 < vi$
 $then\ (v,\ vi)\#ExcessTable-cont\ ExcessAtPG\ vs$
 $else\ ExcessTable-cont\ ExcessAtPG\ vs)$

definition *ExcessTable'* :: $graph \Rightarrow vertex\ list \Rightarrow (vertex \times nat)\ list$ **where**
 $ExcessTable'\ g \equiv ExcessTable-cont (ExcessAt\ g)$

lemma *distinct-ExcessTable-cont*:
 $distinct\ vs \implies$
 $distinct (map\ fst (ExcessTable-cont (ExcessAt\ g)\ vs))$
proof (*induct vs*)
case *Nil* **then show** $?case$ **by** (*simp add: ExcessTable-def*)

```

next
  case (Cons v vs)
  from Cons have  $v \notin \text{set } vs$  by simp
  from Cons have distinct vs by simp
  with Cons have IH:
    distinct (map fst (ExcessTable-cont (ExcessAt g) vs))
    by simp
  moreover have
     $v \notin \text{fst } \text{'set } (ExcessTable-cont (ExcessAt g) vs)$ 
  proof
    assume  $v \in \text{fst } \text{'set } (ExcessTable-cont (ExcessAt g) vs)$ 
    also have  $\text{fst } \text{'set } (ExcessTable-cont (ExcessAt g) vs) \subseteq \text{set } vs$ 
      by (induct vs) auto
    finally have  $v \in \text{set } vs$  .
    with  $v$  show False by contradiction
  qed
  ultimately show ?case by (simp add: ExcessTable-def)
qed

```

```

lemma ExcessTable-cont-eq:
  ExcessTable-cont E vs =
    [( $v, E v$ ).  $v \leftarrow [v \leftarrow vs . 0 < E v]$ ]
by (induct vs) (simp-all)

```

```

lemma ExcessTable-eq: ExcessTable = ExcessTable'
proof (rule ext, rule ext)
  fix  $p g vs$  show ExcessTable g vs = ExcessTable' g vs
  by (simp add: ExcessTable-def ExcessTable'-def ExcessTable-cont-eq)
qed

```

```

lemma distinct-ExcessTable:
  distinct vs  $\implies$  distinct [fst p. p  $\leftarrow$  ExcessTable g vs]
by (simp-all add: ExcessTable-eq ExcessTable'-def distinct-ExcessTable-cont)

```

```

lemma ExcessNotAt-eq:
  minGraphProps g  $\implies$  final g  $\implies$ 
   $\exists V. \text{ExcessNotAt } g \text{ None}$ 
  =  $(\sum_{v \in V} \text{ExcessAt } g v)$ 
   $\wedge \text{separated } g (\text{set } V) \wedge \text{set } V \subseteq \text{set } (\text{vertices } g)$ 
   $\wedge \text{distinct } V$ 
proof (intro exI conjI)
  assume  $mgp: \text{minGraphProps } g$  and  $fin: \text{final } g$ 
  let  $?ps = \text{ExcessTable } g (\text{vertices } g)$ 
  let  $?V = \text{ExcessNotAtRecList } ?ps g$ 
  let  $?vs = \text{vertices } g$ 
  let  $?E = \lambda v. \text{ExcessAt } g v$ 
  have  $t: \text{isTable } ?E ?vs ?ps$  by (rule isTable-ExcessTable)

```

```

with this show ExcessNotAt g None = ( $\sum v \in ?V ?E v$ )
  by (simp add: ListSum-ExcessNotAtRecList ExcessNotAt-def)

show separated g (set ?V)
  by(rule separated-ExcessNotAtRecList[OF mgp fin t])

have set (ExcessNotAtRecList ?ps g)  $\subseteq$  set (map fst ?ps)
  by (rule ExcessNotAtRecList-subset[simplified concat-map-singleton])
also have  $\dots \subseteq$  set (vertices g) by (rule ExcessTable-subset)
finally show set ?V  $\subseteq$  set (vertices g) .

show distinct ?V
  by (simp add: distinct-ExcessNotAtRecList distinct-ExcessTable[simplified concat-map-singleton])
qed

lemma excess-eq:
  assumes  $\gamma: t + q \leq \gamma$ 
  shows excessAtType t q  $0 + t * d$   $\beta + q * d$   $\delta = b$  t q
proof -
  note simps = excessAtType-def squanderVertex-def squanderFace-def
    nat-minus-add-max squanderTarget-def
  from  $\gamma$  have  $q=0 \vee q=1 \vee q=2 \vee q=3 \vee q=4 \vee q=5 \vee q=6 \vee q=7$  by arith
  then show ?thesis
  proof (elim disjE)
    assume  $q: q = 0$ 
    with  $\gamma$  show ?thesis by (simp add: simps)
  next
    assume  $q: q = 1$ 
    with  $\gamma$  show ?thesis by (simp add: simps)
  next
    assume  $q: q = 2$ 
    with  $\gamma$  show ?thesis by (simp add: simps)
  next
    assume  $q: q = 3$ 
    with  $\gamma$  show ?thesis by (simp add: simps)
  next
    assume  $q: q = 4$ 
    with  $\gamma$  show ?thesis by (simp add: simps)
  next
    assume  $q: q = 5$ 
    with  $\gamma$  show ?thesis by (simp add: simps)
  next
    assume  $q: q = 6$ 
    with  $\gamma$  show ?thesis by (simp add: simps)
  next
    assume  $q: q = 7$ 
    with  $\gamma$  show ?thesis by (simp add: simps)
qed

```


qed

lemma *excess-eq1*:

$$\begin{aligned} & \llbracket \text{inv } g; \text{final } g; \text{tame } g; \text{except } g \ v = 0; v \in \text{set}(\text{vertices } g) \rrbracket \implies \\ & \text{ExcessAt } g \ v + (\text{tri } g \ v) * d \ 3 + (\text{quad } g \ v) * d \ 4 \\ & = b \ (\text{tri } g \ v) \ (\text{quad } g \ v) \end{aligned}$$

apply(*subgoal-tac finalVertex g v*)

apply(*simp add: ExcessAt-def excess-eq faceCountMax-bound*)

apply(*auto simp:finalVertex-def plane-final-facesAt*)

done

separating

definition *separating* :: 'a set \Rightarrow ('a \Rightarrow 'b set) \Rightarrow bool **where**

$$\text{separating } V \ F \equiv$$

$$(\forall v1 \in V. \forall v2 \in V. v1 \neq v2 \longrightarrow F \ v1 \cap F \ v2 = \{\})$$

lemma *separating-insert1*:

$$\text{separating } (\text{insert } a \ V) \ F \implies \text{separating } V \ F$$

by (*simp add: separating-def*)

lemma *separating-insert2*:

$$\text{separating } (\text{insert } a \ V) \ F \implies a \notin V \implies v \in V \implies$$

$$F \ a \cap F \ v = \{\}$$

by (*auto simp add: separating-def*)

lemma *sum-disj-Union*:

$$\text{finite } V \implies$$

$$(\bigwedge f. \text{finite } (F \ f)) \implies$$

$$\text{separating } V \ F \implies$$

$$(\sum v \in V. \sum f \in (F \ v). (w \ f :: \text{nat})) = (\sum f \in (\bigcup v \in V. F \ v). w \ f)$$

proof (*induct rule: finite-induct*)

case empty then show ?case **by** *simp*

next

case (*insert a V*)

then have *s*: *separating (insert a V) F* **by** *simp*

then have *separating V F* **by** (*rule-tac separating-insert1*)

with *insert*

have *IH*: $(\sum v \in V. \sum f \in (F \ v). w \ f) = (\sum f \in (\bigcup v \in V. F \ v). w \ f)$

by *simp*

moreover have *fin*: *finite V a \notin V \bigwedge f. finite (F f)* **by** *fact+*

moreover from *s* **have** $\bigwedge v. a \notin V \implies v \in V \implies F \ a \cap F \ v = \{\}$

by (*simp add: separating-insert2*)

with *fin* **have** $(F \ a) \cap (\bigcup v \in V. F \ v) = \{\}$ **by** *auto*

ultimately show ?case **by** (*simp add: sum.union-disjoint*)

qed

lemma *separated-separating*:
assumes $Vg: \text{set } V \subseteq \mathcal{V} \ g$
and $pS: \text{separated } g \ (\text{set } V)$
and $noex: \forall f \in P. |\text{vertices } f| \leq 4$
shows *separating* ($\text{set } V$) ($\lambda v. \text{set } (\text{facesAt } g \ v) \cap P$)
proof –
from pS **have** $i: \forall v \in \text{set } V. \forall f \in \text{set } (\text{facesAt } g \ v).$
 $|\text{vertices } f| \leq 4 \longrightarrow \text{set } (\text{vertices } f) \cap \text{set } V = \{v\}$
by (*simp add: separated-def separated₃-def*)
show *separating* ($\text{set } V$) ($\lambda v. \text{set } (\text{facesAt } g \ v) \cap P$)
proof (*simp add: separating-def, intro ballI impI*)
fix $v1 \ v2$ **assume** $v: v1 \in \text{set } V \ v2 \in \text{set } V \ v1 \neq v2$
hence $v1 : \mathcal{V} \ g$ **using** Vg **by** *blast*
show $(\text{set } (\text{facesAt } g \ v1) \cap P) \cap (\text{set } (\text{facesAt } g \ v2) \cap P) = \{\}$ (**is** $?P$)
proof (*rule ccontr*)
assume $\neg ?P$
then obtain f **where** $f1: f \in \text{set } (\text{facesAt } g \ v1)$
and $f2: f \in \text{set } (\text{facesAt } g \ v2)$ **and** $f : P$ **by** *auto*
with $noex$ **have** $l: |\text{vertices } f| \leq 4$ **by** *blast*
from $v \ f1 \ l \ i$ **have** $\text{set } (\text{vertices } f) \cap \text{set } V = \{v1\}$ **by** *simp*
also from $v \ f2 \ l \ i$
have $\text{set } (\text{vertices } f) \cap \text{set } V = \{v2\}$ **by** *simp*
finally have $v1 = v2$ **by** *auto*
then show *False* **by** *contradiction* (*rule v*)
qed
qed
qed

lemma *ListSum-V-F-eq-ListSum-F*:
assumes $pl: \text{inv } g$
and $pS: \text{separated } g \ (\text{set } V)$ **and** $dist: \text{distinct } V$
and $V\text{-subset}: \text{set } V \subseteq \text{set } (\text{vertices } g)$
and $noex: \forall f \in \text{Collect } P. |\text{vertices } f| \leq 4$
shows $(\sum v \in V \sum f \in \text{filter } P \ (\text{facesAt } g \ v) \ (w::\text{face} \Rightarrow \text{nat}) \ f)$
 $= (\sum f \in [f \leftarrow \text{faces } g . \exists v \in \text{set } V. f \in \text{set } (\text{facesAt } g \ v) \cap \text{Collect } P] \ w \ f)$
proof –
have $s: \text{separating} \ (\text{set } V) \ (\lambda v. \text{set } (\text{facesAt } g \ v) \cap \text{Collect } P)$
by (*rule separated-separating[OF V-subset pS noex]*)
moreover note $dist$
moreover from $pl \ V\text{-subset}$
have $\bigwedge v. v \in \text{set } V \Longrightarrow \text{distinct} \ (\text{facesAt } g \ v)$
by (*blast intro:mgp-dist-facesAt[OF inv-mgp]*)
hence $v: \bigwedge v. v \in \text{set } V \Longrightarrow \text{distinct} \ (\text{filter } P \ (\text{facesAt } g \ v))$
by *simp*
moreover
have $\text{distinct} \ [f \leftarrow \text{faces } g . \exists v \in \text{set } V. f \in \text{set } (\text{facesAt } g \ v) \cap \text{Collect } P]$
by (*intro distinct-filter minGraphProps11'[OF inv-mgp[OF pl]]*)
moreover from pl **have** $\{x. x \in \mathcal{F} \ g \wedge (\exists v \in \text{set } V. x \in \text{set } (\text{facesAt } g \ v)) \wedge P$

$x\} =$
 $(\bigcup_{v \in \text{set } V}. \text{set } (\text{facesAt } g \ v) \cap \text{Collect } P)$ **using** $V\text{-subset}$
by $(\text{blast intro: minGraphProps inv-mgp})$
moreover from v **have** $(\sum_{v \in \text{set } V}. \text{ListSum } (\text{filter } P \ (\text{facesAt } g \ v)) \ w) =$
 $(\sum_{v \in \text{set } V}. \text{sum } w \ (\text{set}(\text{facesAt } g \ v) \cap \text{Collect } P))$
by $(\text{auto simp add: ListSum-conv-sum Int-def})$
ultimately show $?thesis$
by $(\text{simp add: ListSum-conv-sum sum-disj-Union})$
qed

lemma separated-disj-Union2:

assumes $pl: \text{inv } g$ **and** $fin: \text{final } g$ **and** $ne: \text{noExceptionals } g \ (\text{set } V)$

and $pS: \text{separated } g \ (\text{set } V)$ **and** $dist: \text{distinct } V$

and $V\text{-subset: } \text{set } V \subseteq \text{set } (\text{vertices } g)$

shows $(\sum_{v \in V} \sum_{f \in \text{facesAt } g \ v} (w::\text{face} \Rightarrow \text{nat}) \ f)$

$$= (\sum_{f \in [f \leftarrow \text{faces } g. \exists v \in \text{set } V. f \in \text{set } (\text{facesAt } g \ v)]} w \ f)$$

proof –

let $?P = \lambda f. |\text{vertices } f| \leq 4$

have $\forall v \in \text{set } V. \forall f \in \text{set } (\text{facesAt } g \ v). |\text{vertices } f| \leq 4$

using $V\text{-subset } ne$

by $(\text{auto simp: noExceptionals-def})$

$\text{intro: minGraphProps5}[OF \text{inv-mgp}[OF \text{pl}]] \text{not-exceptional}[OF \text{pl } fin]$

thus $?thesis$

using $\text{ListSum-V-F-eq-ListSum-F}[\text{where } P = ?P, OF \text{pl } pS \ \text{dist } V\text{-subset}]$

by $(\text{simp add: Int-def cong: conj-cong})$

qed

lemma squanderFace-distr2: $\text{inv } g \Longrightarrow \text{final } g \Longrightarrow \text{noExceptionals } g \ (\text{set } V) \Longrightarrow$

$\text{separated } g \ (\text{set } V) \Longrightarrow \text{distinct } V \Longrightarrow \text{set } V \subseteq \text{set } (\text{vertices } g) \Longrightarrow$

$$\left(\sum_{f \in [f \leftarrow \text{faces } g. \exists v \in \text{set } V. f \in \text{set } (\text{facesAt } g \ v)]} d \ |\text{vertices } f| \right)$$

$$= \left(\sum_{v \in V} ((\text{tri } g \ v) * d \ 3 + (\text{quad } g \ v) * d \ 4) \right)$$

proof –

assume $pl: \text{inv } g$

assume $fin: \text{final } g$

assume $ne: \text{noExceptionals } g \ (\text{set } V)$

assume $\text{separated } g \ (\text{set } V) \ \text{distinct } V$ **and** $V\text{-subset: } \text{set } V \subseteq \text{set } (\text{vertices } g)$

with $pl \ ne \ fin$ **have**

$$\left(\sum_{f \in [f \leftarrow \text{faces } g. \exists v \in \text{set } V. f \in \text{set } (\text{facesAt } g \ v)]} d \ |\text{vertices } f| \right)$$

$$= \left(\sum_{v \in V} \sum_{f \in \text{facesAt } g \ v} d \ |\text{vertices } f| \right)$$

by $(\text{simp add: separated-disj-Union2})$

also have $\bigwedge v. v \in \text{set } V \Longrightarrow$

$$\left(\sum_{f \in \text{facesAt } g \ v} d \ |\text{vertices } f| \right)$$

$$= (\text{tri } g \ v) * d \ 3 + (\text{quad } g \ v) * d \ 4$$

proof –

fix v **assume** $v1: v \in \text{set } V$

with $V\text{-subset}$ **have** $v: v \in \text{set } (\text{vertices } g)$ **by** auto

with ne **have** d :
 $\bigwedge f. f \in \text{set } (\text{facesAt } g \ v) \implies$
 $|\text{vertices } f| = 3 \vee |\text{vertices } f| = 4$
proof –
fix f **assume** $f: f \in \text{set } (\text{facesAt } g \ v)$
then have $ff: f \in \text{set } (\text{faces } g)$ **by** $(\text{rule } \text{minGraphProps5}[\text{OF } \text{inv-mgp}[\text{OF } \text{pl}]$
 $v])$
with $ne \ f \ v1 \ \text{pl} \ \text{fin} \ v$ **have** $|\text{vertices } f| \leq 4$
by $(\text{auto } \text{simp} \ \text{add: } \text{noExceptionals-def } \text{not-exceptional})$
moreover from $\text{pl} \ \text{ff}$ **have** $3 \leq |\text{vertices } f|$ **by** $(\text{rule } \text{planeN4})$
ultimately show $?thesis \ f$ **by** arith
qed

from $d \ \text{pl} \ v$ **have**
 $(\sum f \in \text{facesAt } g \ v \ \text{d } |\text{vertices } f|)$
 $= (\sum f \in [f \leftarrow \text{facesAt } g \ v. |\text{vertices } f| = 3] \ \text{d } |\text{vertices } f|)$
 $+ (\sum f \in [f \leftarrow \text{facesAt } g \ v. |\text{vertices } f| = 4] \ \text{d } |\text{vertices } f|)$
apply $(\text{rule-tac } \text{ListSum-disj-union})$
apply $(\text{rule } \text{distinct-filter})$ **apply** simp
apply $(\text{rule } \text{distinct-filter})$ **apply** simp
apply simp
apply force
apply force
done

also have $\dots = \text{tri } g \ v * \text{d } 3 + \text{quad } g \ v * \text{d } 4$
proof –
from $\text{pl} \ \text{fin} \ v$ **have** $\bigwedge A. [f \leftarrow \text{facesAt } g \ v. \ \text{final } f \wedge A \ f]$
 $= [f \leftarrow \text{facesAt } g \ v. \ A \ f]$
by $(\text{rule-tac } \text{filter-eqI}) \ (\text{auto } \text{simp:plane-final-facesAt})$
with fin **show** $?thesis$ **by** $(\text{auto } \text{simp} \ \text{add: } \text{tri-def } \text{quad-def})$
qed

finally show $(\sum f \in \text{facesAt } g \ v \ \text{d } |\text{vertices } f|) = \text{tri } g \ v * \text{d } 3 + \text{quad } g \ v * \text{d } 4$.
qed

then have $(\sum v \in V \ \sum f \in \text{facesAt } g \ v \ \text{d } |\text{vertices } f|) =$
 $(\sum v \in V \ (\text{tri } g \ v * \text{d } 3 + \text{quad } g \ v * \text{d } 4))$
by $(\text{rule } \text{ListSum-eq})$
finally show $?thesis$.
qed

lemma *separated-subset*:

$V1 \subseteq V2 \implies \text{separated } g \ V2 \implies \text{separated } g \ V1$

proof $(\text{simp} \ \text{add: } \text{separated-def } \text{separated}_3\text{-def } \text{separated}_2\text{-def},$
 $\text{elim } \text{conjE}, \ \text{intro } \text{allI } \text{impI } \text{ballI } \text{conjI})$

fix $v \ f$

assume $a: v \in V1 \ V1 \subseteq V2 \ f \in \text{set } (\text{facesAt } g \ v)$

$|\text{vertices } f| \leq 4$

```

     $\forall v \in V2. \forall f \in \text{set } (\text{facesAt } g \ v). |\text{vertices } f| \leq 4 \implies$ 
     $\text{set } (\text{vertices } f) \cap V2 = \{v\}$ 
then show  $\text{set } (\text{vertices } f) \cap V1 = \{v\}$  by auto
next
  fix  $v \ f$ 
assume  $a: v \in V1 \ V1 \subseteq V2 \ f \in \text{set } (\text{facesAt } g \ v)$ 
     $\forall v \in V2. \forall f \in \text{set } (\text{facesAt } g \ v). f \cdot v \notin V2$ 
then have  $v \in V2$  by auto
with  $a$  have  $f \cdot v \notin V2$  by auto
with  $a$  show  $f \cdot v \notin V1$  by auto
qed

end

```

21 Correctness of Lower Bound for Final Graphs

```

theory LowerBound
imports PlaneProps ScoreProps
begin

```

```

theorem total-weight-lowerbound:
   $\text{inv } g \implies \text{final } g \implies \text{tame } g \implies \text{admissible } w \ g \implies$ 
   $(\sum_{f \in \text{faces } g} w \ f) < \text{squanderTarget} \implies$ 
   $\text{squanderLowerBound } g \leq (\sum_{f \in \text{faces } g} w \ f)$ 

```

proof –

```

assume final:  $\text{final } g$  and tame:  $\text{tame } g$  and pl:  $\text{inv } g$ 
assume admissible:  $\text{admissible } w \ g$ 
assume  $w: (\sum_{f \in \text{faces } g} w \ f) < \text{squanderTarget}$ 

```

```

have squanderLowerBound  $g$ 
  =  $\text{ExcessNotAt } g \ \text{None} + \text{faceSquanderLowerBound } g$ 
by (simp add: squanderLowerBound-def)

```

We expand the definition of *faceSquanderLowerBound*.

```

also have faceSquanderLowerBound  $g = (\sum_{f \in \text{faces } g} d \ |\text{vertices } f|)$ 

```

We expand the definition of *ExcessNotAt*.

```

also from ExcessNotAt-eq[OF pl[THEN inv-mgp] final] obtain  $V$ 
where eq:  $\text{ExcessNotAt } g \ \text{None} = (\sum_{v \in V} \text{ExcessAt } g \ v)$ 
and pS: separated  $g$  (set  $V$ )
and V-subset:  $\text{set } V \subseteq \text{set}(\text{vertices } g)$ 
and V-distinct: distinct  $V$ 

```

22 Properties of Tame Graph Enumeration (1)

```

theory GeneratorProps
imports Plane1Props Generator TameProps LowerBound
begin

lemma genPolyTame-spec:
  generatePolygonTame n v f g = [g' ← generatePolygon n v f g . ¬ notame g']
by(simp add:generatePolygonTame-def generatePolygon-def enum-enumerator)

lemma genPolyTame-subset-genPoly:
  g' ∈ set(generatePolygonTame i v f g) ⇒
  g' ∈ set(generatePolygon i v f g)
by(auto simp add:generatePolygon-def generatePolygonTame-def enum-enumerator)

lemma next-tame0-subset-plane:
  set(next-tame0 p g) ⊆ set(next-plane p g)
by(auto simp add:next-tame0-def next-plane-def polysizes-def
  elim!:genPolyTame-subset-genPoly simp del:upt-Suc)

lemma genPoly-new-face:
  [[g' ∈ set (generatePolygon n v f g); minGraphProps g; f ∈ set (nonFinals g);
  v ∈ V f; n ≥ 3]] ⇒
  ∃f ∈ set(finals g') - set(finals g). |vertices f| = n
apply(auto simp add:generatePolygon-def image-def)
apply(rename-tac is)
apply(frule enumerator-length2)
apply arith
apply(frule (4) pre-subdivFace-indexToVertexList)
apply(arith)
apply(subgoal-tac indexToVertexList f v is ≠ [])
prefer 2 apply(subst length-0-conv[symmetric]) apply simp
apply(simp add: subdivFace-subdivFace'-eq)
apply(clarsimp simp:neq-Nil-conv)
apply(rename-tac ovs)
apply(subgoal-tac |indexToVertexList f v is| = |ovs| + 1)
prefer 2 apply(simp)
apply(drule (1) pre-subdivFace-pre-subdivFace')
apply(drule (1) final-subdivFace')
apply(simp add:nonFinals-def)
apply(simp add:pre-subdivFace'-def)
apply (simp (no-asm-use))
apply(simp)
apply blast
done

```

lemma *genPoly-incr-facesquander-lb*:
assumes $g' \in \text{set } (\text{generatePolygon } n \ v \ f \ g) \ \text{inv } g$
 $f \in \text{set}(\text{nonFinals } g) \ v \in \mathcal{V} \ f \ 3 \leq n$
shows $\text{faceSquanderLowerBound } g' \geq \text{faceSquanderLowerBound } g + d \ n$
proof –
from $\text{genPoly-new-face}[OF \ \text{assms}(1) \ \text{inv-mgp}[OF \ \text{assms}(2)] \ \text{assms}(3-5)]$ **ob-**
tain f
where $f: f \in \text{set } (\text{finals } g') - \text{set}(\text{finals } g)$
and $\text{size}: |\text{vertices } f| = n$ **by** *auto*
have $g': g' \in \text{set}(\text{next-plane0 } (n - 3) \ g)$ **using** $\text{assms}(5)$
by $(\text{rule-tac } \text{in-next-plane0I}[OF \ \text{assms}(1,3-5)])$ *simp*
note $\text{dist} = \text{minGraphProps11}'[OF \ \text{inv-mgp}[OF \ \text{assms}(2)]]$
note $\text{inv}' = \text{invariantE}[OF \ \text{inv-inv-next-plane0}, OF \ g' \ \text{assms}(2)]$
note $\text{dist}' = \text{minGraphProps11}'[OF \ \text{inv-mgp}[OF \ \text{inv}']]$
note $\text{subset} = \text{next-plane0-final-subset}[OF \ g']$
have $\text{faceSquanderLowerBound } g' \geq$
 $\text{faceSquanderLowerBound } g + d \ |\text{vertices } f|$
proof $(\text{unfold } \text{faceSquanderLowerBound-def})$
have $(\sum_{f \in \text{finals } g} d \ |\text{vertices } f|) + d \ |\text{vertices } f| =$
 $(\sum_{f \in \text{set}(\text{finals } g)} d \ |\text{vertices } f|) + d \ |\text{vertices } f|$
using dist **by** $(\text{simp } \text{add:finals-def } \text{ListSum-conv-sum})$
also have $\dots = (\sum_{f \in \text{set}(\text{finals } g) \cup \{f\}} d \ |\text{vertices } f|)$
using f **by** *simp*
also have $\dots \leq (\sum_{f \in \text{set}(\text{finals } g')} d \ |\text{vertices } f|)$
using $f \ \text{subset}$ **by** $(\text{fastforce } \text{intro!}: \text{sum-mono2})$
also have $\dots = (\sum_{f \in \text{finals } g'} d \ |\text{vertices } f|)$
using dist' **by** $(\text{simp } \text{add:finals-def } \text{ListSum-conv-sum})$
finally show $(\sum_{f \in \text{finals } g} d \ |\text{vertices } f|) + d \ |\text{vertices } f|$
 $\leq (\sum_{f \in \text{finals } g'} d \ |\text{vertices } f|)$.
qed
with size **show** *?thesis* **by** *blast*
qed

definition $\text{close} :: \text{graph} \Rightarrow \text{vertex} \Rightarrow \text{vertex} \Rightarrow \text{bool}$ **where**
 $\text{close } g \ u \ v \equiv$
 $\exists f \in \text{set}(\text{facesAt } g \ u)$. *if* $|\text{vertices } f| = 4$ *then* $v = f \cdot u \vee v = f \cdot (f \cdot u)$
else $v = f \cdot u$

lemma *delAround-def*: $\text{deleteAround } g \ u \ ps = [p \leftarrow ps. \neg \text{close } g \ u \ (fst \ p)]$
by $(\text{induct } ps)$ $(\text{auto } \text{simp: } \text{deleteAroundCons } \text{close-def})$

lemma *close-sym*: **assumes** $\text{mgp}: \text{minGraphProps } g$ **and** $ug: u : \mathcal{V} \ g$ **and** $cl: \text{close}$
 $g \ u \ v$
shows $\text{close } g \ v \ u$
proof –

obtain f **where** $f: f \in \text{set}(\text{facesAt } g \ u)$ **and**
if: if $|\text{vertices } f| = 4$ *then* $v = f \cdot u \vee v = f \cdot (f \cdot u)$ *else* $v = f \cdot u$
using cl **by** (*unfold close-def*) *blast*
note $uf = \text{minGraphProps6}[OF \ \text{mgp} \ ug \ f]$
note $\text{distf} = \text{minGraphProps3}[OF \ \text{mgp} \ \text{minGraphProps5}[OF \ \text{mgp} \ ug \ f]]$
show *?thesis*
proof *cases*
assume $4: |\text{vertices } f| = 4$
hence $v = f \cdot u \vee v = f \cdot (f \cdot u)$ **using** *if* **by** *simp*
thus *?thesis*
proof
assume $v = f \cdot u$
then obtain f' **where** $f' \in \text{set}(\text{facesAt } g \ v)$ $f' \cdot v = u$
using *mgp-nextVertex-face-ex2*[*OF mgp ug f*] **by** *blast*
thus *?thesis* **by**(*auto simp:close-def*)
next
assume $v: v = f \cdot (f \cdot u)$
hence $f \cdot (f \cdot v) = u$ **using** *quad-next4-id*[*OF 4 distf uf*] **by** *simp*
moreover have $f \in \text{set}(\text{facesAt } g \ v)$ **using** $v \ uf$
by(*simp add: minGraphProps7*[*OF mgp minGraphProps5*[*OF mgp ug f*]])
ultimately show *?thesis* **using** 4 **by**(*auto simp:close-def*)
qed
next
assume $|\text{vertices } f| \neq 4$
hence $v = f \cdot u$ **using** *if* **by** *simp*
then obtain f' **where** $f' \in \text{set}(\text{facesAt } g \ v)$ $f' \cdot v = u$
using *mgp-nextVertex-face-ex2*[*OF mgp ug f*] **by** *blast*
thus *?thesis* **by**(*auto simp:close-def*)
qed
qed

lemma *sep-conv*:

assumes $\text{mgp}: \text{minGraphProps } g$ **and** $V \subseteq \mathcal{V} \ g$

shows *separated* $g \ V = (\forall u \in V. \forall v \in V. u \neq v \longrightarrow \neg \text{close } g \ u \ v)$ (**is** $?P = ?Q$)

proof

assume *sep: ?P*

show *?Q*

proof(*clarify*)

fix $u \ v$ **assume** $uv: u \in V \ v \in V \ u \neq v$ **and** $cl: \text{close } g \ u \ v$

from cl **obtain** f **where** $f: f \in \text{set}(\text{facesAt } g \ u)$ **and**

if: if $|\text{vertices } f| = 4$ *then* $(v = f \cdot u) \vee (v = f \cdot (f \cdot u))$
else $(v = f \cdot u)$

by (*unfold close-def*) *blast*

have $u : \mathcal{V} \ g$ **using** $\langle u : V \rangle \langle V \subseteq \mathcal{V} \ g \rangle$ **by** *blast*

note $uf = \text{minGraphProps6}[OF \ \text{mgp} \ \langle u : \mathcal{V} \ g \rangle \ f]$

show *False*

proof *cases*

assume $4: |\text{vertices } f| = 4$


```

hence  $v = f \cdot u \vee v = f \cdot (f \cdot u)$  using if by simp
thus False
proof
  assume  $v = f \cdot u$ 
  thus False using sep f uv
  by(simp add:separated-def separated2-def separated3-def)
next
  assume  $v = f \cdot (f \cdot u)$ 
  moreover hence  $v \in \mathcal{V} f$  using  $\langle u \in \mathcal{V} f \rangle$  by simp
  moreover have  $|\text{vertices } f| \leq 4$  using 4 by arith
  ultimately show False using sep f uv  $\langle u \in \mathcal{V} f \rangle$ 
  apply(unfold separated-def separated2-def separated3-def)

  apply(subgoal-tac  $f \cdot (f \cdot u) \in \mathcal{V} f \cap V$ )
  prefer 2 apply blast
  by simp
qed
next
  assume 4:  $|\text{vertices } f| \neq 4$ 
  hence  $v = f \cdot u$  using if by simp
  thus False using sep f uv
  by(simp add:separated-def separated2-def separated3-def)
qed
qed
next
  assume not-cl: ?Q
  show ?P
  proof(simp add:separated-def, rule conjI)
  show separated2 g V
  proof (clarsimp simp:separated2-def)
  fix  $v f$  assume a:  $v \in V f \in \text{set } (\text{facesAt } g v) f \cdot v \in V$ 
  have  $v : \mathcal{V} g$  using a(1)  $\langle V \subseteq \mathcal{V} g \rangle$  by blast
  show False using a not-cl mgp-facesAt-no-loop[OF mgp  $\langle v : \mathcal{V} g \rangle$  a(2)]
  by(fastforce simp: close-def split:if-split-asm)
qed
  show separated3 g V
  proof (clarsimp simp:separated3-def)
  fix  $v f$ 
  assume  $v \in V$  and f:  $f \in \text{set } (\text{facesAt } g v)$  and len:  $|\text{vertices } f| \leq 4$ 
  have vg:  $v : \mathcal{V} g$  using  $\langle v : V \rangle \langle V \subseteq \mathcal{V} g \rangle$  by blast
  note distf = minGraphProps3[OF mgp minGraphProps5[OF mgp vg f]]
  note vf = minGraphProps6[OF mgp vg f]
  { fix u assume  $u \in \mathcal{V} f$  and  $u \in V$ 
  have  $u = v$ 
  proof cases
  assume 3:  $|\text{vertices } f| = 3$ 
  hence  $\mathcal{V} f = \{v, f \cdot v, f \cdot (f \cdot v)\}$ 
  using vertices-triangle[OF - vf distf] by simp
  moreover

```

```

{ assume u = f · v
  hence u = v
  using not-cl f ⟨u ∈ V⟩ ⟨v ∈ V⟩ 3
  by(force simp:close-def split:if-split-asm)
}
moreover
{ assume u = f · (f · v)
  hence fu: f · u = v
  by(simp add: tri-next3-id[OF 3 distf ⟨v ∈ V f⟩])
  hence (u,v) ∈ E f using nextVertex-in-edges[OF ⟨u ∈ V f⟩]
  by(simp add:fu)
  then obtain f' where f' ∈ set(facesAt g v) (v,u) ∈ E f'
  using mgp-edge-face-ex[OF mgp vg f] by blast
  hence u = v using not-cl ⟨u ∈ V⟩ ⟨v ∈ V⟩ 3
  by(force simp:close-def edges-face-eq split:if-split-asm)
}
ultimately show u=v using ⟨u ∈ V f⟩ by blast
next
assume 3: |vertices f| ≠ 3
hence 4: |vertices f| = 4
  using len mgp-vertices3[OF mgp minGraphProps5[OF mgp vg f]] by
arith
hence V f = {v, f · v, f · (f · v), f · (f · (f · v))}
  using vertices-quad[OF - vf distf] by simp
moreover
{ assume u = f · v
  hence u = v
  using not-cl f ⟨u ∈ V⟩ ⟨v ∈ V⟩ 4
  by(force simp:close-def split:if-split-asm)
}
moreover
{ assume u = f · (f · v)
  hence u = v
  using not-cl f ⟨u ∈ V⟩ ⟨v ∈ V⟩ 4
  by(force simp:close-def split:if-split-asm)
}
moreover
{ assume u = f · (f · (f · v))
  hence fu: f · u = v
  by(simp add: quad-next4-id[OF 4 distf ⟨v ∈ V f⟩])
  hence (u,v) ∈ E f using nextVertex-in-edges[OF ⟨u ∈ V f⟩]
  by(simp add:fu)
  then obtain f' where f' ∈ set(facesAt g v) (v,u) ∈ E f'
  using mgp-edge-face-ex[OF mgp vg f] by blast
  hence u = v using not-cl ⟨u ∈ V⟩ ⟨v ∈ V⟩ 4
  by(force simp:close-def edges-face-eq split:if-split-asm)
}
ultimately show u=v using ⟨u ∈ V f⟩ by blast
qed

```

```

    }
    thus  $\mathcal{V} f \cap V = \{v\}$  using  $\langle v \in V \rangle$  vf by blast
  qed
qed
qed

```

lemma *sep-ne*: $\exists P \subseteq M. \text{separated } g \text{ (fst ' } P)$
by(*unfold separated-def separated₂-def separated₃-def*) *blast*

lemma *ExcessNotAtRec-conv-Max*:

assumes *mgp*: *minGraphProps* *g*

shows $\text{set}(\text{map } \text{fst } ps) \subseteq \mathcal{V} g \implies \text{distinct}(\text{map } \text{fst } ps) \implies$

ExcessNotAtRec *ps* *g* =

$\text{Max}\{\sum p \in P. \text{snd } p \mid P. P \subseteq \text{set } ps \wedge \text{separated } g \text{ (fst ' } P)\}$

(**is** $- \implies - \implies - = \text{Max}(\text{?M } ps)$ **is** $- \implies - \implies - = \text{Max}\{- \mid P. \text{?S } ps P\}$)

proof(*induct ps rule: length-induct*)

case (*1 ps0*)

note *IH* = *1(1)* **and** *subset* = *1(2)* **and** *dist* = *1(3)*

show *?case*

proof (*cases ps0*)

case *Nil* **thus** *?thesis* **by** *simp*

next

case (*Cons p ps*)

let *?ps* = *deleteAround g (fst p) ps*

have *le*: $|\text{?ps}| \leq |ps|$ **by**(*simp add:delAround-def*)

have *dist'*: $\text{distinct}(\text{map } \text{fst } \text{?ps})$ **using** *dist Cons*

apply (*clarsimp simp:delAround-def*)

apply(*drule distinct-filter[where P = Not o close g (fst p)]*)

apply(*simp add: filter-map o-def*)

done

have *fst p* : $\mathcal{V} g$ **and** *fst ' set ps* $\subseteq \mathcal{V} g$

using *subset Cons* **by** *auto*

have *sub1*: $\bigwedge P Q. P \subseteq \{x : \text{set } ps. Q x\} \implies \text{fst ' } P \subseteq \mathcal{V} g$

using *subset Cons* **by** *auto*

have *sub2*: $\bigwedge P Q. P \subseteq \text{insert } p \{x : \text{set } ps. Q x\} \implies \text{fst ' } P \subseteq \mathcal{V} g$

using *subset Cons* **by** *auto*

have *sub3*: $\bigwedge P. P \subseteq \text{insert } p (\text{set } ps) \implies \text{fst ' } P \subseteq \mathcal{V} g$

using *subset Cons* **by** *auto*

have $\bigwedge a. \text{set} (\text{map } \text{fst} (\text{deleteAround } g a ps)) \subseteq \mathcal{V} g$

using *deleteAround-subset[of g - ps] subset Cons*

by *auto*

hence *ExcessNotAtRec ps0 g* = $\text{max} (\text{Max}(\text{?M } ps)) (\text{Max}(\text{?M } \text{?ps}) + \text{snd } p)$

using *Cons IH subset le dist dist'* **by** (*cases p*) *simp*

also have $\text{Max} (\text{?M } \text{?ps}) + \text{snd } p =$

$\text{Max} \{(\sum p \in P. \text{snd } p) + \text{snd } p \mid P. \text{?S } \text{?ps } P\}$

by (*auto simp add:setcompr-eq-image Max-add-commute[symmetric] sep-ne*

intro!: *arg-cong [where f=Max]*)

also have $\{(\sum p \in P. \text{snd } p) + \text{snd } p \mid P. \text{?S } \text{?ps } P\} =$

$\{\text{sum } \text{snd} (\text{insert } p P) \mid P. \text{?S } \text{?ps } P\}$

```

using dist Cons
apply (auto simp:delAround-def)
apply(rule-tac x=P in exI)
apply(fastforce intro!: sum.insert[THEN trans,symmetric] elim: finite-subset)
apply(rule-tac x=P in exI)
apply(fastforce intro!: sum.insert[THEN trans] elim: finite-subset)
done
also have ... = {sum snd P |P.
  P ⊆ insert p (set ?ps) ∧ p ∈ P ∧ separated g (fst ‘ P)}
apply(auto simp add:sep-conv[OF mgp] sub1 sub2 delAround-def cong: conj-cong)
apply(rule-tac x = insert p P in exI)
apply simp
apply(rule conjI) apply blast
using ⟨image fst (set ps) ⊆ V g⟩ ⟨fst p : V g⟩
apply (blast intro:close-sym[OF mgp])
apply(rule-tac x = P-⟨p⟩ in exI)
apply (simp add:insert-absorb)
apply blast
done
also have ... = {sum snd P |P.
  P ⊆ insert p (set ps) ∧ p ∈ P ∧ separated g (fst ‘ P)}
using Cons dist
apply(auto simp add:sep-conv[OF mgp] sub2 sub3 delAround-def cong: conj-cong)
apply(rule-tac x = P in exI)
apply simp
apply auto
done
also have max (Max(?M ps)) (Max ...) = Max(?M ps ∪ {sum snd P |P.
  P ⊆ insert p (set ps) ∧ p ∈ P ∧ separated g (fst ‘ P)})
(is - = Max ?U)
proof -
  have {sum snd P |P.
    P ⊆ insert p (set ps) ∧ p ∈ P ∧ separated g (fst ‘ P)} ≠ {}
  apply simp
  apply(rule-tac x={p} in exI)
  using ⟨fst p : V g⟩ by(simp add:sep-conv[OF mgp])
  thus ?thesis by(simp add: Max-Un sep-ne)
qed
also have ?U = ?M ps0 using Cons by simp blast
finally show ?thesis .
qed
qed

```

lemma *dist-ExcessTab*: *distinct (map fst (ExcessTable g (vertices g)))*
by(simp add:ExcessTable-def vertices-graph o-def)

lemma *mono-ExcessTab*: $\llbracket g' \in \text{set}(\text{next-plane0}_p g); \text{inv } g \rrbracket \implies$
 $\text{set}(\text{ExcessTable } g (\text{vertices } g)) \subseteq \text{set}(\text{ExcessTable } g' (\text{vertices } g'))$
apply(*clarsimp simp:ExcessTable-def image-def*)
apply(*rule conjI*)
apply(*blast dest:next-plane0-vertices-subset inv-mgp*)
apply(*clarsimp simp:ExcessAt-def split:if-split-asm*)
apply(*frule (3) next-plane0-finalVertex-mono*)
apply(*simp add: next-plane0-len-filter-eq tri-def quad-def except-def*)
done

lemma *close-antimono*:
 $\llbracket g' \in \text{set}(\text{next-plane0}_p g); \text{inv } g; u \in \mathcal{V} g; \text{finalVertex } g u \rrbracket \implies$
 $\text{close } g' u v \implies \text{close } g u v$
by(*simp add:close-def next-plane0-finalVertex-facesAt-eq*)

lemma *ExcessTab-final*:
 $p \in \text{set}(\text{ExcessTable } g (\text{vertices } g)) \implies \text{finalVertex } g (\text{fst } p)$
by(*clarsimp simp:ExcessTable-def image-def ExcessAt-def split:if-split-asm*)

lemma *ExcessTab-vertex*:
 $p \in \text{set}(\text{ExcessTable } g (\text{vertices } g)) \implies \text{fst } p \in \mathcal{V} g$
by(*clarsimp simp:ExcessTable-def image-def ExcessAt-def split:if-split-asm*)

lemma *fst-set-ExcessTable-subset*:
 $\text{fst } \text{'set}(\text{ExcessTable } g (\text{vertices } g)) \subseteq \mathcal{V} g$
by(*clarsimp simp:ExcessTable-def image-def ExcessAt-def split:if-split-asm*)

lemma *next-plane0-incr-ExcessNotAt*:
 $\llbracket g' \in \text{set}(\text{next-plane0}_p g); \text{inv } g \rrbracket \implies$
 $\text{ExcessNotAt } g \text{ None} \leq \text{ExcessNotAt } g' \text{ None}$
apply(*frule (1) invariantE[OF inv-inv-next-plane0]*)
apply(*frule (1) mono-ExcessTab*)
apply(*simp add: ExcessNotAt-def ExcessNotAtRec-conv-Max[OF - - dist-ExcessTab]*
 $\text{fst-set-ExcessTable-subset}$)
apply(*rule Max-mono*)
prefer 2 apply(*simp add: sep-ne*)
prefer 2 apply(*simp*)
apply *auto*
apply(*rule-tac x=P in exI*)
apply *auto*
apply(*subgoal-tac fst ' P $\subseteq \mathcal{V} g'$*)
prefer 2 apply(*blast dest: ExcessTab-vertex*)
apply(*subgoal-tac fst ' P $\subseteq \mathcal{V} g$*)
prefer 2 apply(*blast dest: ExcessTab-vertex*)
apply(*simp add:sep-conv*)
apply(*blast intro:close-antimono ExcessTab-final ExcessTab-vertex*)
done

lemma *next-plane0-incr-squander-lb*:
 $\llbracket g' \in \text{set } (\text{next-plane0}_p g); \text{inv } g \rrbracket \implies$
 $\text{squanderLowerBound } g \leq \text{squanderLowerBound } g'$
apply(*simp add:squanderLowerBound-def*)
apply(*frule (1) next-plane0-incr-ExcessNotAt*)
apply(*clarsimp simp add:next-plane0-def split:if-split-asm*)
apply(*drule (4) genPoly-incr-facesquander-lb*)
apply *arith*
done

lemma *inv-notame*:
 $\llbracket g' \in \text{set } (\text{next-plane0}_p g); \text{inv } g; \text{notame7 } g \rrbracket$
 $\implies \text{notame7 } g'$
apply(*simp add:notame-def notame7-def tame11b-def is-tame13a-def tame10ub-def*
del:disj-not1)
apply(*frule inv-mgp*)
apply(*frule (1) next-plane0-vertices-subset*)
apply(*erule disjE*)
apply(*simp add:vertices-graph*)
apply(*rule disjI2*)
apply(*erule disjE*)
apply *clarify*
apply(*frule (2) next-plane0-incr-degree*)
apply(*frule (2) next-plane0-incr-except*)
apply (*force split:if-split-asm*)
apply(*frule (1) next-plane0-incr-squander-lb*)
apply(*arith*)
done

lemma *inv-inv-notame*:
 $\text{invariant}(\lambda g. \text{inv } g \wedge \text{notame7 } g) \text{ next-plane}_p$
apply(*simp add:invariant-def*)
apply(*blast intro: inv-notame mgp-next-plane0-if-next-plane[OF inv-mgp]*
invariantE[OF inv-inv-next-plane])
done

lemma *untame-notame*:
 $\text{untame } (\lambda g. \text{inv } g \wedge \text{notame7 } g)$
proof(*clarsimp simp add: notame-def notame7-def untame-def tame11b-def is-tame13a-def*
tame10ub-def
 $\text{linorder-not-le linorder-not-less}$)
fix *g* **assume** *final g inv g tame g*
and *cases: 15 < countVertices g \vee*
 $(\exists v \in \mathcal{V} g. (\text{except } g v = 0 \longrightarrow 7 < \text{degree } g v) \wedge$
 $(0 < \text{except } g v \longrightarrow 6 < \text{degree } g v))$

```

      ∨ squanderTarget ≤ squanderLowerBound g
      (is ?A ∨ ?B ∨ ?C is - ∨ (∃ v ∈ V g. ?B' v) ∨ -)
from cases show False
proof (elim disjE)
  assume ?B
  then obtain v where v: v ∈ V g ?B' v by auto
  show False
  proof cases
    assume except g v = 0
    thus False using ⟨tame g⟩ v by (auto simp: tame-def tame11b-def)
  next
    assume except g v ≠ 0
    thus False using ⟨tame g⟩ v
      by (auto simp: except-def filter-empty-conv tame-def tame11b-def
        minGraphProps-facesAt-eq[OF inv-mgp[OF ⟨inv g⟩]] split:if-split-asm)
  qed
next
  assume ?A
  thus False using ⟨tame g⟩ by (simp add: tame-def tame10-def)
next
  assume ?C
  thus False using total-weight-lowerbound[OF ⟨inv g⟩ ⟨final g⟩ ⟨tame g⟩
    ⟨tame g⟩ by (force simp add: tame-def tame13a-def)
  qed
qed

```

lemma *polysizes-tame*:

```

[[ g' ∈ set (generatePolygon n v f g); inv g; f ∈ set (nonFinals g);
  v ∈ V f; 3 ≤ n; n < 4 + p; n ∉ set (polysizes p g) ]]
⇒ notame7 g'
apply (frule (4) in-next-plane0I)
apply (frule (4) genPoly-incr-facesquander-lb)
apply (frule (1) next-plane0-incr-ExcessNotAt)
apply (simp add: notame-def notame7-def is-tame13a-def faceSquanderLowerBound-def
  polysizes-def squanderLowerBound-def)
done

```

lemma *genPolyTame-notame*:

```

[[ g' ∈ set (generatePolygon n v f g); g' ∉ set (generatePolygonTame n v f g);
  inv g; 3 ≤ n ]]
⇒ notame7 g'
by (fastforce simp: generatePolygon-def generatePolygonTame-def enum-enumerator
  notame-def notame7-def)

```

declare *upt-Suc*[*simp del*]

lemma *excess-notame*:

```

[[ inv g; g' ∈ set (next-planep g); g' ∉ set (next-tame0 p g) ]]
⇒ notame7 g'

```

```

apply(frule (1) mgp-next-plane0-if-next-plane[OF inv-mgp])
apply(auto simp add:next-tame0-def next-plane-def split:if-split-asm)
apply(rename-tac n)
apply(case-tac n ∈ set(polysizes p g))
  apply(drule bspec) apply assumption
  apply(simp add:genPolyTame-notame)
apply(subgoal-tac minimalFace (nonFinals g) ∈ set(nonFinals g))
  prefer 2 apply(simp add:minimalFace-def)
apply(subgoal-tac minimalVertex g (minimalFace (nonFinals g)) ∈  $\mathcal{V}$ (minimalFace
(nonFinals g)))
  apply(blast intro:polysizes-tame)
apply(simp add:minimalVertex-def)
apply(rule minimal-in-set)
apply(erule mgp-vertices-nonempty[OF inv-mgp])
apply(simp add:nonFinals-def)
done
declare upt-Suc[simp]

```

```

lemma next-tame0-comp:  $\llbracket$  Seedp [next-plane p] →* g; final g; tame g  $\rrbracket$ 
 $\implies$  Seedp [next-tame0 p] →* g
apply(rule filterout-untame-succs[OF inv-inv-next-plane inv-inv-notame
untame-notame])
  apply(blast intro:excess-notame)
  apply assumption
  apply(rule inv-Seed)
  apply assumption
apply assumption
done

```

```

lemma inv-inv-next-tame0: invariant inv (next-tame0 p)
by(rule inv-subset[OF inv-inv-next-plane next-tame0-subset-plane])

```

```

lemma inv-inv-next-tame: invariant inv next-tamep
apply(simp add:next-tame-def)
apply(rule inv-subset[OF inv-inv-next-tame0])
apply auto
done

```

```

lemma mgp-TameEnum:  $g \in \text{TameEnum}_p \implies \text{minGraphProps } g$ 
by (unfold TameEnumP-def)
  (blast intro: RTranCl-inv[OF inv-inv-next-tame] inv-Seed inv-mgp)

```

end

23 Properties of Tame Graph Enumeration (2)

```

theory TameEnumProps
imports GeneratorProps
begin

```

Completeness of filter for final graphs.

```

lemma untame-negFin:
assumes pl: inv g and fin: final g and tame: tame g
shows is-tame g
proof (unfold is-tame-def, intro conjI)
  show tame10 g using tame by(auto simp:tame-def)
next
  show tame11a g using tame by(auto simp:tame-def)
next
  show tame12o g using tame by(auto simp:tame-def)
next
next
  from tame obtain w where adm: admissible w g
  and sqn:  $(\sum f \in \text{faces } g \ w f) < \text{squanderTarget}$  by(auto simp:tame-def tame13a-def)
  moreover have squanderLowerBound g  $\leq (\sum f \in \text{faces } g \ w f)$ 
  using pl fin tame adm sqn by (rule total-weight-lowerbound)
  ultimately show is-tame13a g by(auto simp:is-tame13a-def)
qed

```

```

lemma next-tame-comp:
   $\llbracket \text{tame } g; \text{final } g; \text{Seed}_p [\text{next-tame0 } p] \rightarrow^* g \rrbracket$ 
   $\implies \text{Seed}_p [\text{next-tame}_p] \rightarrow^* g$ 
apply (unfold next-tame-def)
apply(rule filter-tame-succs[OF inv-inv-next-tame0])
  apply(simp add:next-tame0-def finalGraph-def)
  apply(rule context-conjI)
  apply(simp)
  apply(blast dest:untame-negFin)
  apply assumption
  apply(rule inv-Seed)
apply assumption+
done

```

```

end
theory Worklist
imports HOL-Library.While-Combinator RTranCl Quasi-Order
begin

```

```

definition
  worklist-aux ::  $('s \Rightarrow 'a \Rightarrow 'a \text{ list}) \Rightarrow ('a \Rightarrow 's \Rightarrow 's)$ 
   $\Rightarrow 'a \text{ list} * 's \Rightarrow ('a \text{ list} * 's) \text{option}$ 

```

where

worklist-aux succs f =
while-option
 $(\lambda(ws,s). ws \neq [])$
 $(\lambda(ws,s). \text{case } ws \text{ of } x\#ws' \Rightarrow (\text{succs } s \ x \ @ \ ws', f \ x \ s))$

definition *worklist* :: ('s \Rightarrow 'a \Rightarrow 'a list) \Rightarrow ('a \Rightarrow 's \Rightarrow 's)
 \Rightarrow 'a list \Rightarrow 's \Rightarrow 's option **where**
worklist succs f ws s =
 $(\text{case } \text{worklist-aux succs } f \ (ws,s) \ \text{of}$
 $\text{None} \Rightarrow \text{None} \mid \text{Some}(ws,s) \Rightarrow \text{Some } s)$

lemma *worklist-aux-Nil*: *worklist-aux succs f* ([],s) = Some([],s)
by(*simp add: worklist-aux-def while-option-unfold*)

lemma *worklist-aux-Cons*:
worklist-aux succs f (x#ws',s) = *worklist-aux succs f* (succs s x @ ws', f x s)
by(*simp add: worklist-aux-def while-option-unfold*)

lemma *worklist-aux-unfold*[code]:
worklist-aux succs f (ws,s) =
 $(\text{case } ws \ \text{of } [] \Rightarrow \text{Some}([],s) \mid$
 $x\#ws' \Rightarrow \text{worklist-aux succs } f \ (\text{succs } s \ x \ @ \ ws', f \ x \ s))$
by(*simp add: worklist-aux-Nil worklist-aux-Cons split: list.split*)

definition
worklist-tree-aux :: ('a \Rightarrow 'a list) \Rightarrow ('a \Rightarrow 's \Rightarrow 's)
 \Rightarrow 'a list * 's \Rightarrow ('a list * 's)option

where
worklist-tree-aux succs = *worklist-aux* ($\lambda s. \text{succs}$)

lemma *worklist-tree-aux-unfold*[code]:
worklist-tree-aux succs f (ws,s) =
 $(\text{case } ws \ \text{of } [] \Rightarrow \text{Some}([],s) \mid$
 $x\#ws' \Rightarrow \text{worklist-tree-aux succs } f \ (\text{succs } x \ @ \ ws', f \ x \ s))$
by(*simp add: worklist-tree-aux-def worklist-aux-Nil worklist-aux-Cons split: list.split*)

abbreviation *Rel* :: ('a \Rightarrow 'a list) \Rightarrow ('a * 'a)set **where**
Rel f == {(x,y). y : set(f x)}

lemma *Image-Rel-set*:
 $(\text{Rel succs})^* \ \text{“ } \text{set}(\text{succs } x) = (\text{Rel succs})^+ \ \text{“ } \{x\}$
by(*auto simp add: trancl-unfold-left*)

lemma *RTranCl-conv*:
 $g \ [\text{succs}] \rightarrow^* \ h \ \longleftrightarrow \ (g,h) : ((\text{Rel succs})^*) \ (\text{is } ?L = ?R)$
proof –
have ?L \Longrightarrow ?R

```

apply(erule RTranCl-induct)
apply blast
apply (auto elim: rtrancl-into-rtrancl)
done
moreover
have ?R  $\implies$  ?L
  apply(erule converse-rtrancl-induct)
  apply(rule RTranCl.refl)
  apply (auto elim: RTranCl.succs)
  done
ultimately show ?thesis by blast
qed

```

lemma *worklist-end-empty*:

```

worklist-aux succs f (ws,s) = Some(ws',s')  $\implies$  ws' = []
unfolding worklist-aux-def
by (drule while-option-stop) simp

```

theorem *worklist-tree-aux-Some-foldl*:

```

assumes worklist-tree-aux succs f (ws,s) = Some(ws',s')
shows  $\exists$  rs. set rs = ((Rel succs)*) “ (set ws)  $\wedge$ 
s' = foldl ( $\lambda$  s x. f x s) s rs

```

proof –

```

let ?b =  $\lambda$ (ws,s). ws  $\neq$  []
let ?c =  $\lambda$ (ws,s). case ws of x#ws'  $\Rightarrow$  (succs x @ ws', f x s)
let ?Q =  $\lambda$ ws' s' done.
s' = foldl ( $\lambda$  x s. f s x) s done  $\wedge$ 
((Rel succs)*) “ (set ws) =
set done  $\cup$  ((Rel succs)*) “ set ws'
let ?P =  $\lambda$ (ws,s).  $\exists$  done. ?Q ws s done
have 0: while-option ?b ?c (ws,s) = Some(ws',s')
using assms by(simp add: worklist-tree-aux-def worklist-aux-def)
from while-option-stop[OF 0] have ws' = [] by simp
have init: ?P (ws,s)
apply auto
apply(rule-tac x = [] in exI)
apply simp
done
{ fix ws s
assume ?P (ws,s)
then obtain done where ?Q ws s done by blast
assume ?b(ws,s)
then obtain x ws' where ws = x # ws' by(auto simp: neq-Nil-conv)
then have ?Q (succs x @ ws') (f x s) (done @ [x])
using <?Q ws s done>
apply simp
apply(erule thin-rl)+
apply (auto simp add: Image-Un Image-Rel-set)
apply (blast elim: rtranclE intro: rtrancl-into-trancl1)

```

```

done
hence ?P(?c(ws,s)) using ‹ws=x#ws'›
  by(simp only: split-conv list.cases) blast
}
then have ?P(ws',s')
  using while-option-rule[where P=?P, OF - 0 init]
  by auto
then show ?thesis using ‹ws'=[]› by auto
qed

```

definition *worklist-tree succs* f ws $s =$
(case worklist-tree-aux succs f (ws,s) *of*
None \Rightarrow *None* | *Some*(ws,s) \Rightarrow *Some* s)

theorem *worklist-tree-Some-foldl*:
worklist-tree succs f ws $s =$ *Some* $s' \implies$
 $\exists rs. \text{set } rs = ((\text{Rel succs})^*) \text{ `` } (\text{set } ws) \wedge$
 $s' = \text{foldl } (\lambda s x. f x s) s rs$
by (*simp add: worklist-tree-def worklist-tree-aux-Some-foldl split:option.splits prod.splits*)

lemma *invariant-succs*:
assumes *invariant* I *succs*
and $\forall x \in S. I x$
shows $\forall x \in (\text{Rel succs})^* \text{ `` } S. I x$
proof –
 { **fix** $x y$ **have** $(x,y) : (\text{Rel succs})^* \implies I x \implies I y$
proof (*induct rule:rtrancl-induct*)
 case *base* **thus** ?*case* .
 next
 case *step* **with** *assms*(1) **show** ?*case* **by** (*auto simp:invariant-def*)
 qed
 } **with** *assms*(2) **show** ?*thesis* **by** *blast*
qed

lemma *worklist-tree-aux-rule*:
assumes *worklist-tree-aux succs* f $(ws,s) =$ *Some*(ws',s')
and *invariant* I *succs*
and $\forall x \in \text{set } ws. I x$
and $\bigwedge s. P [] s s$
and $\bigwedge r x ws s. I x \implies \forall x \in \text{set } ws. I x \implies P ws (f x s) r \implies P (x\#ws) s r$
shows $\exists rs. \text{set } rs = ((\text{Rel succs})^*) \text{ `` } (\text{set } ws) \wedge P rs s s'$
proof –
 let ? $R = (\text{Rel succs})^* \text{ `` } \text{set } ws$
from *worklist-tree-aux-Some-foldl*[*OF* *assms*(1)] **obtain** rs **where**
 $rs: \text{set } rs = ?R s' = \text{foldl } (\lambda s x. f x s) s rs$ **by** *blast*
 { **fix** xs **have** $(\forall x \in \text{set } xs. I x) \implies P xs s (\text{foldl } (\lambda s x. f x s) s xs)$
proof (*induct xs arbitrary: s*)
 case *Nil* **show** ?*case* **using** *assms*(4) **by** *simp*
 next

```

      case Cons thus ?case using assms(5) by simp
    qed
  }
  with invariant-succs[OF assms(2,3)] show ?thesis by (metis rs)
qed

```

lemma *worklist-tree-aux-rule2*:

assumes *worklist-tree-aux succs* $f (ws, s) = \text{Some}(ws', s')$

and *invariant* $I \text{ succs}$

and $\forall x \in \text{set } ws. I x$

and $S s$ **and** $\bigwedge x s. I x \implies S s \implies S(f x s)$

and $\bigwedge s. P \square s s$

and $\bigwedge r x ws s. I x \implies \forall x \in \text{set } ws. I x \implies S s$

$\implies P ws (f x s) r \implies P (x \# ws) s r$

shows $\exists rs. \text{set } rs = ((\text{Rel succs})^*) \text{ `` } (\text{set } ws) \wedge P rs s s'$

proof –

let $?R = (\text{Rel succs})^* \text{ `` } \text{set } ws$

from *worklist-tree-aux-Some-foldl*[OF assms(1)] **obtain** rs **where**

$rs: \text{set } rs = ?R s' = \text{foldl } (\lambda s x. f x s) s rs$ **by** *blast*

{ **fix** xs **have** $(\forall x \in \text{set } xs. I x) \implies S s \implies P xs s$ (*foldl* $(\lambda s x. f x s) s xs$)

proof(*induct xs arbitrary: s*)

case *Nil* **show** ?case **using** assms(6) **by** *simp*

next

case *Cons* **thus** ?case **using** assms(5,7) **by** *simp*

qed

}

with *invariant-succs*[OF assms(2,3)] *assms*(4) **show** ?thesis **by** (*metis* rs)

qed

lemma *worklist-tree-rule*:

assumes *worklist-tree succs* $f ws s = \text{Some}(s')$

and *invariant* $I \text{ succs}$

and $\forall x \in \text{set } ws. I x$

and $\bigwedge s. P \square s s$

and $\bigwedge r x ws s. I x \implies \forall x \in \text{set } ws. I x \implies P ws (f x s) r \implies P (x \# ws) s r$

shows $\exists rs. \text{set } rs = ((\text{Rel succs})^*) \text{ `` } (\text{set } ws) \wedge P rs s s'$

proof –

obtain ws' **where** *worklist-tree-aux succs* $f (ws, s) = \text{Some}(ws', s')$ **using** *assms*(1)

by(*simp add: worklist-tree-def split: option.split-asm prod.split-asm*)

from *worklist-tree-aux-rule*[**where** $P=P, OF$ *this* *assms*(2–5)] **show** ?thesis **by** *blast*

qed

lemma *worklist-tree-rule2*:

assumes *worklist-tree succs* $f ws s = \text{Some}(s')$

and *invariant* $I \text{ succs}$

and $\forall x \in \text{set } ws. I x$

and $S s$ **and** $\bigwedge x s. I x \implies S s \implies S(f x s)$

and $\bigwedge s. P \square s s$

and $\bigwedge r x ws s. I x \implies \forall x \in \text{set } ws. I x \implies S s$
 $\implies P ws (f x s) r \implies P (x \# ws) s r$
shows $\exists rs. \text{set } rs = ((\text{Rel } \text{succs})^*) \text{ `` } (\text{set } ws) \wedge P rs s s'$
proof –
obtain ws' **where** $\text{worklist-tree-aux succs } f (ws, s) = \text{Some}(ws', s')$ **using** $\text{assms}(1)$
by $(\text{simp add: worklist-tree-def split: option.split-asm prod.split-asm})$
from $\text{worklist-tree-aux-rule2}[\text{where } P=P \text{ and } S=S, \text{OF this assms}(2-7)]$
show $?thesis$ **by** blast
qed

lemma $\text{worklist-tree-aux-state-inv}$:
assumes $\text{worklist-tree-aux succs } f (ws, s) = \text{Some}(ws', s')$
and $I s$
and $\bigwedge x s. I s \implies I(f x s)$
shows $I s'$
proof –
from $\text{worklist-tree-aux-rule}[\text{where } P=\lambda ws s s'. I s \longrightarrow I s' \text{ and } I=\lambda x. \text{True},$
 $\text{OF assms}(1)] \text{ assms}(2-3)$
show $?thesis$ **by** $(\text{auto simp: invariant-def})$
qed

lemma $\text{worklist-tree-state-inv}$:
 $\text{worklist-tree succs } f ws s = \text{Some}(s')$
 $\implies I s \implies (\bigwedge x s. I s \implies I(f x s)) \implies I s'$
unfolding worklist-tree-def
by $(\text{auto intro: worklist-tree-aux-state-inv split: option.splits})$

locale $\text{set-modulo} = \text{quasi-order} +$
fixes $\text{empty} :: 's$
and $\text{insert-mod} :: 'a \Rightarrow 's \Rightarrow 's$
and $\text{set-of} :: 's \Rightarrow 'a \text{ set}$
and $I :: 'a \Rightarrow \text{bool}$
and $S :: 's \Rightarrow \text{bool}$
assumes $\text{set-of-empty: set-of empty} = \{\}$
and $\text{set-of-insert-mod: } I x \implies S s \wedge (\forall x \in \text{set-of } s. I x)$
 \implies
 $\text{set-of}(\text{insert-mod } x s) = \text{insert } x (\text{set-of } s) \vee$
 $(\exists y \in \text{set-of } s. x \preceq y) \wedge \text{set-of}(\text{insert-mod } x s) = \text{set-of } s$
and $S\text{-empty: } S \text{ empty}$
and $S\text{-insert-mod: } S s \implies S (\text{insert-mod } x s)$
begin

definition $\text{insert-mod2} :: ('b \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'b \Rightarrow 's \Rightarrow 's$ **where**
 $\text{insert-mod2 } P f x s = (\text{if } P x \text{ then } \text{insert-mod } (f x) s \text{ else } s)$

definition $SI s = (S s \wedge (\forall x \in \text{set-of } s. I x))$

lemma $SI\text{-empty: } SI \text{ empty}$

by(*simp add: SI-def S-empty set-of-empty*)

lemma *SI-insert-mod:*

$I\ x \implies SI\ s \implies SI\ (insert\text{-}mod\ x\ s)$

apply(*simp add: SI-def S-insert-mod*)

by (*metis insertE set-of-insert-mod*)

lemma *SI-insert-mod2:* $(\bigwedge x. inv0\ x \implies I\ (f\ x)) \implies$

$inv0\ x \implies SI\ s \implies SI\ (insert\text{-}mod2\ P\ f\ x\ s)$

by (*metis insert-mod2-def SI-insert-mod*)

definition *worklist-tree-coll-aux* ::

$('b \Rightarrow 'b\ list) \Rightarrow ('b \Rightarrow bool) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'b\ list \Rightarrow 's \Rightarrow 's\ option$

where

$worklist\text{-}tree\text{-}coll\text{-}aux\ succs\ P\ f = worklist\text{-}tree\ succs\ (insert\text{-}mod2\ P\ f)$

definition *worklist-tree-coll* ::

$('b \Rightarrow 'b\ list) \Rightarrow ('b \Rightarrow bool) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'b\ list \Rightarrow 's\ option$

where

$worklist\text{-}tree\text{-}coll\ succs\ P\ f\ ws = worklist\text{-}tree\text{-}coll\text{-}aux\ succs\ P\ f\ ws\ empty$

lemma *worklist-tree-coll-aux-equiv:*

assumes $worklist\text{-}tree\text{-}coll\text{-}aux\ succs\ P\ f\ ws\ s = Some\ s'$

and *invariant* $inv0\ succs$

and $\forall x \in set\ ws. inv0\ x$

and $\bigwedge x. inv0\ x \implies I(f\ x)$

and $SI\ s$

shows $set\text{-}of\ s' =_{\leq}$

$f\ ' \{x : (Rel\ succs)^* \text{ `` } (set\ ws). P\ x\} \cup set\text{-}of\ s$

apply(*insert assms(1)*)

unfolding *worklist-tree-coll-aux-def*

apply(*drule worklist-tree-rule2*[**where** $I = inv0$ and $S = SI$ and

$P = \lambda ws\ s\ s'. SI\ s \longrightarrow set\text{-}of\ s' =_{\leq} f\ ' \{x : set\ ws. P\ x\} \cup set\text{-}of\ s,$
OF - assms(2,3,5)])

apply(*simp add: SI-insert-mod2 assms(4)*)

apply(*clarsimp*)

apply(*clarsimp simp add: insert-mod2-def split: if-split-asm*)

apply(*frule assms(4)*)

apply(*frule SI-def[THEN iffD1]*)

apply(*frule (1) set-of-insert-mod*)

apply (*simp add: SI-insert-mod*)

apply(*erule disjE*)

prefer 2

apply(*rule seteq-qle-trans*)

apply *assumption*

apply (*simp add: defs*)

apply *blast*

apply(*rule seteq-qle-trans*)

apply *assumption*

```

apply (simp add: defs)
apply blast
apply(rule seteq-qle-trans)
apply assumption
apply (simp add: defs)
apply blast
using assms(5)
apply auto
done

```

lemma *worklist-tree-coll-equiv*:

```

worklist-tree-coll succs P f ws = Some s'  $\implies$  invariant inv0 succs
 $\implies \forall x \in \text{set } ws. \text{inv0 } x \implies (\bigwedge x. \text{inv0 } x \implies I(f x))$ 
 $\implies \text{set-of } s' =_{\geq} f \text{ ' } \{x : (\text{Rel succs})^* \text{ " (set } ws). P x\}$ 

```

```

unfolding worklist-tree-coll-def
apply(drule (2) worklist-tree-coll-aux-equiv)
apply(auto simp: set-of-empty SI-empty)
done

```

lemma *worklist-tree-coll-aux-subseteq*:

```

worklist-tree-coll-aux succs P f ws t0 = Some t  $\implies$ 
invariant inv0 succs  $\implies \forall g \in \text{set } ws. \text{inv0 } g \implies$ 
 $(\bigwedge x. \text{inv0 } x \implies I(f x)) \implies SI t_0 \implies$ 
set-of t  $\subseteq$  set-of t0  $\cup f \text{ ' } \{h : (\text{Rel succs})^* \text{ " set } ws. P h\}$ 

```

```

unfolding worklist-tree-coll-aux-def
apply(drule worklist-tree-rule2[where I = inv0 and S = SI and P =
 $\lambda ws t t'. \text{set-of } t' \subseteq \text{set-of } t \cup f \text{ ' } \{g \in \text{set } ws. P g\}$ ])
apply assumption
apply assumption
apply assumption
apply(simp add: SI-insert-mod2)
apply clarsimp
apply (clarsimp simp: insert-mod2-def split: if-split-asm)
using set-of-insert-mod
apply(simp add: SI-def image-def)
apply(blast)
apply blast
apply blast
done

```

lemma *worklist-tree-coll-subseteq*:

```

worklist-tree-coll succs P f ws = Some t  $\implies$ 
invariant inv0 succs  $\implies \forall g \in \text{set } ws. \text{inv0 } g \implies$ 
 $(\bigwedge x. \text{inv0 } x \implies I(f x)) \implies$ 
set-of t  $\subseteq f \text{ ' } \{h : (\text{Rel succs})^* \text{ " set } ws. P h\}$ 

```

```

unfolding worklist-tree-coll-def
apply(drule (1) worklist-tree-coll-aux-subseteq)
apply(auto simp: set-of-empty SI-empty)
done

```



```

lemma worklist-tree-coll-inv:
  worklist-tree-coll succs P f ws = Some s  $\implies$  S s
unfolding worklist-tree-coll-def worklist-tree-coll-aux-def
apply (drule worklist-tree-state-inv[where  $I = S$ ])
apply (auto simp: S-empty insert-mod2-def S-insert-mod)
done

end

end
theory Maps
imports Worklist Quasi-Order
begin

locale maps =
fixes empty :: 'm
and up :: 'a  $\Rightarrow$  'b list  $\Rightarrow$  'm  $\Rightarrow$  'm
and map-of :: 'm  $\Rightarrow$  'a  $\Rightarrow$  'b list
and M :: 'm  $\Rightarrow$  bool
assumes map-empty: map-of empty = ( $\lambda a.$  [])
and map-up: map-of (up a b m) = (map-of m)(a := b)
and M-empty: M empty
and M-up: M m  $\implies$  M (up a b m)
begin

definition set-of m = (UN x. set(map-of m x))

end

locale set-mod-maps = maps empty up map-of M + quasi-order gle
for empty :: 'm
and up :: 'a  $\Rightarrow$  'b list  $\Rightarrow$  'm  $\Rightarrow$  'm
and map-of :: 'm  $\Rightarrow$  'a  $\Rightarrow$  'b list
and M :: 'm  $\Rightarrow$  bool
and gle :: 'b  $\Rightarrow$  'b  $\Rightarrow$  bool (infix  $\langle \preceq \rangle$  60)
+
fixes subsumed :: 'b  $\Rightarrow$  'b  $\Rightarrow$  bool
and I :: 'b  $\Rightarrow$  bool
and key :: 'b  $\Rightarrow$  'a
assumes equiv-iff-gle: I x  $\implies$  I y  $\implies$  subsumed x y = (x  $\preceq$  y)
and key=key
begin

definition insert-mod x m =
  (let k = key x; ys = map-of m k
   in if ( $\exists y \in \text{set } ys. \text{subsumed } x y$ ) then m else up k (x\#ys) m)

end

```

```

sublocale
  set-mod-maps <
  set-by-maps?: set-modulo qle empty insert-mod set-of I M
proof (standard, goal-cases)
  case 1 show ?case by(simp add:set-of-def map-empty)
next
  case 2 thus ?case
    by (auto simp: Let-def insert-mod-def set-of-def map-up equiv-iff-qle
      split:if-split-asm)
next
  case 3 show ?case by(simp add: M-empty)
next
  case 4 thus ?case
    by(simp add: insert-mod-def Let-def M-up)
qed

end

```

24 Archive

```

theory Arch
imports Main HOL-Library.Code-Target-Numeral
begin

setup <fn thy =>
  let
    val T = @{typ integer list list list}
    val dir = Resources.master-directory thy
  in
    thy |>
    Code-Runtime.polyml-as-definition
    [(@{binding Tri'}, T), (@{binding Quad'}, T), (@{binding Pent'}, T),
      (@{binding Hex'}, T)]
    (map (Path.append dir)
      [path <Archives/Tri.ML>, path <Archives/Quad.ML>,
        path <Archives/Pent.ML>, path <Archives/Hex.ML>])
  end
  >

```

The definition of these constants is only ever needed at the ML level when running the eval proof method.

```

definition Tri :: nat list list list
where
  Tri = (map o map o map) nat-of-integer Tri'

```

```

definition Quad :: nat list list list
where

```

$Quad = (map \circ map \circ map) \text{ nat-of-integer Quad}'$

definition $Pent :: nat \text{ list list list}$

where

$Pent = (map \circ map \circ map) \text{ nat-of-integer Pent}'$

definition $Hex :: nat \text{ list list list}$

where

$Hex = (map \circ map \circ map) \text{ nat-of-integer Hex}'$

end

25 Comparing Enumeration and Archive

theory *ArchCompAux*

imports *TameEnum Trie.Tries Maps Arch Worklist*

begin

function $qsort :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**

$qsort \text{ le } [] = [] \mid$

$qsort \text{ le } (x\#xs) = qsort \text{ le } [y \leftarrow xs . \neg \text{le } x \ y] @ [x] @$
 $qsort \text{ le } [y \leftarrow xs . \text{le } x \ y]$

by *pat-completeness auto*

termination by (*relation measure (size o snd)*)

(*auto simp add: length-filter-le [THEN le-less-trans]*)

definition $\text{nof-vertices} :: 'a \text{ fgraph} \Rightarrow nat$ **where**

$\text{nof-vertices} = \text{length} \circ \text{remdups} \circ \text{concat}$

definition $\text{fgraph} :: graph \Rightarrow nat \text{ fgraph}$ **where**

$\text{fgraph } g = \text{map vertices (faces } g)$

definition $\text{hash} :: nat \text{ fgraph} \Rightarrow nat \text{ list}$ **where**

$\text{hash } fs = (\text{let } n = \text{nof-vertices } fs \text{ in}$

$[n, \text{size } fs] @$

$qsort (\lambda x \ y. y < x) (\text{map } (\lambda i. \text{foldl } (+) \ 0 \ (\text{map size } [f \leftarrow fs. i \in \text{set } f]))$
 $[0..<n]))$

definition $\text{samet} :: (nat, nat \text{ fgraph}) \text{ tries option} \Rightarrow nat \text{ fgraph list} \Rightarrow bool$

where

$\text{samet } fgto \ ags = (\text{case } fgto \ \text{of } None \Rightarrow False \mid \text{Some } tfgs \Rightarrow$

$\text{let } tags = \text{tries-of-list hash } ags \text{ in}$

$(\text{all-tries } (\lambda fg. \text{list-ex } (\text{iso-test } fg) (\text{lookup-tries tags (hash } fg)))) \text{ tfgs} \wedge$

$\text{all-tries } (\lambda ag. \text{list-ex } (\text{iso-test } ag) (\text{lookup-tries tfgs (hash } ag)))) \text{ tags})$

definition $\text{pre-iso-test} :: vertex \text{ fgraph} \Rightarrow bool$ **where**

$\text{pre-iso-test } Fs \longleftrightarrow$

$[] \notin \text{set } Fs \wedge (\forall F \in \text{set } Fs. \text{distinct } F) \wedge \text{distinct } (\text{map rotate-min } Fs)$

```

interpretation map:
  maps Trie None [] update-trie lookup-tries invar-trie
proof (standard, goal-cases)
  case 1 show ?case by(rule ext) simp
next
  case 2 show ?case by(rule ext) (simp add: lookup-update)
next
  case 3 show ?case by(simp)
next
  case 4 thus ?case by (simp add: invar-trie-update)
qed

```

```

lemma set-of-conv: set-tries = maps.set-of lookup-tries
by(rule ext) (auto simp add: set-tries-def map.set-of-def)

```

```

end

```

26 Completeness of Archive Test

```

theory ArchCompProps
imports TameEnumProps ArchCompAux
begin
lemma mgp-pre-iso-test: minGraphProps g  $\implies$  pre-iso-test(fgraph g)
apply(simp add:pre-iso-test-def fgraph-def image-def)
apply(rule conjI) apply(blast dest: mgp-vertices-nonempty[symmetric])
apply(rule conjI) apply(blast intro:minGraphProps)
apply(drule minGraphProps11)
apply(simp add:normFaces-def normFace-def verticesFrom-def minVertex-def
  rotate-min-def o-def)
done

```

```

corollary iso-test-correct:
  [[ pre-iso-test Fs1; pre-iso-test Fs2 ]]  $\implies$ 
  iso-test Fs1 Fs2 = (Fs1  $\simeq$  Fs2)
by(simp add:pre-iso-test-def iso-correct inj-on-rotate-min-iff[symmetric]
  distinct-map nof-vertices-def length-remdups-concat)

```

```

lemma trie-all-eq-set-of-trie:
  invar-trie t  $\implies$  all-trie (list-all P) t = ( $\forall v \in$  set-tries t. P v)
by(simp add: all-trie-eq-ran set-tries-eq-ran)

```

```

lemma samet-imp-iso-seteq:
assumes pre1:  $\bigwedge g. gsopt = \text{Some } gs \implies g \in \text{set-tries } gs \implies \text{pre-iso-test } g$ 
and pre2:  $\bigwedge g. g \in \text{set arch} \implies \text{pre-iso-test } g$ 
and inv:  $\bigwedge gs. gsopt = \text{Some } gs \implies \text{invar-trie } gs$ 
and same: samet gsopt arch

```

shows $\exists gs. gsopt = Some\ gs \wedge set_tries\ gs \simeq set\ arch$
proof –
obtain gs **where** $[simp]: gsopt = Some\ gs$ **and** $test1: \bigwedge g. g \in set_tries\ gs \implies \exists h \in set\ arch. iso_test\ g\ h$ **and** $test2: \bigwedge g. g \in set\ arch \implies \exists h \in set_tries\ gs. iso_test\ g\ h$
using $same\ inv$
by($force\ simp: samet_def\ trie_all_eq_set_of_trie\ invar_of_list\ all_tries_def$
 $split: option.splits$
 $dest: in_set_lookup_of_listD\ in_set_lookup_set_triesD$)
have $set_tries\ gs \subseteq_{\simeq} set\ arch$
proof ($auto\ simp: qle_gr.defs$)
fix g **assume** $g: g \in set_tries\ gs$
obtain h **where** $h: h \in set\ arch$ **and** $test: iso_test\ g\ h$
using $test1[OF\ g]$ **by** $blast$
thus $\exists h \in set\ arch. g \simeq h$
using $h\ pre1[OF\ -\ g]\ pre2[OF\ h]$ **by** ($auto\ simp: iso_test_correct$)
qed
moreover
have $set\ arch \subseteq_{\simeq} set_tries\ gs$
proof ($auto\ simp: qle_gr.defs$)
fix g **assume** $g: g \in set\ arch$
obtain h **where** $h: h \in set_tries\ gs$ **and** $test: iso_test\ g\ h$
using $test2[OF\ g]$ **by** $blast$
thus $\exists h \in set_tries\ gs. g \simeq h$
using $h\ pre1[OF\ -\ h]\ pre2[OF\ g]$ **by** ($auto\ simp: iso_test_correct$)
qed
ultimately show $?thesis$ **by** ($auto\ simp: qle_gr.seteq_qle_def$)
qed

lemma $samet_imp_iso_subseq$:
assumes $pre1: \bigwedge gs\ g. gsopt = Some\ gs \implies g \in set_tries\ gs \implies pre_iso_test\ g$
and $pre2: \bigwedge g. g \in set\ arch \implies pre_iso_test\ g$
and $inv: \bigwedge gs. gsopt = Some\ gs \implies invar_trie\ gs$
and $same: samet\ gsopt\ arch$
shows $\exists gs. gsopt = Some\ gs \wedge set_tries\ gs \subseteq_{\simeq} set\ arch$
using $qle_gr.seteq_qle_def\ assms\ samet_imp_iso_seteq$ **by** $metis$

global-interpretation set_mod_trie :
 $set_mod_maps\ Trie\ None\ []\ update_trie\ lookup_tries\ invar_trie\ (\simeq)\ iso_test\ pre_iso_test\ hash$
defines $insert_mod_trie = set_mod_maps.insert_mod\ update_trie\ lookup_tries\ iso_test\ hash$
and $worklist_tree_coll_trie = set_modulo.worklist_tree_coll\ (Trie\ None\ [])\ insert_mod_trie$
and $worklist_tree_coll_aux_trie = set_modulo.worklist_tree_coll_aux\ insert_mod_trie$
and $insert_mod2_trie = set_modulo.insert_mod2\ insert_mod_trie$
by $standard\ (simp_all\ add: iso_test_correct)$

definition $enum_filter_finals :: (graph \Rightarrow graph\ list) \Rightarrow graph\ list$

\Rightarrow (*nat, nat fgraph*) *tries option where*
enum-filter-finals succs = set-mod-trie.worklist-tree-coll succs final fgraph

definition *tameEnumFilter* :: *nat* \Rightarrow (*nat, nat fgraph*)*tries option where*
tameEnumFilter p = enum-filter-finals (next-tame p) [Seed p]

lemma *TameEnum-tameEnumFilter*:

tameEnumFilter p = Some t \Rightarrow *set-tries t* \simeq *fgraph ' TameEnum_p*
apply(*auto simp: tameEnumFilter-def TameEnumP-def enum-filter-finals-def*)
apply(*drule set-mod-trie.worklist-tree-coll-equiv[OF - inv-inv-next-tame]*)
apply (*auto simp: set-of-conv inv-Seed mgp-pre-iso-test RTranCl-conv*)
done

lemma *tameEnumFilter-subseteq-TameEnum*:

tameEnumFilter p = Some t \Rightarrow *set-tries t* \subseteq *fgraph ' TameEnum_p*
by(*auto simp add: tameEnumFilter-def TameEnumP-def enum-filter-finals-def*
set-of-conv inv-Seed mgp-pre-iso-test RTranCl-conv
dest!: set-mod-trie.worklist-tree-coll-subseteq[OF - inv-inv-next-tame])

lemma *inv-tries-tameEnumFilter*:

tameEnumFilter p = Some t \Rightarrow *invar-trie t*
unfolding *tameEnumFilter-def enum-filter-finals-def*
by(*erule set-mod-trie.worklist-tree-coll-inv*)

theorem *combine-evals-filter*:

$\forall g \in \text{set arch. pre-iso-test } g \Rightarrow \text{samet (tameEnumFilter } p) \text{ arch}$
 $\Rightarrow \text{fgraph ' TameEnum}_p \subseteq_{\simeq} \text{set arch}$
apply(*subgoal-tac* $\exists t. \text{tameEnumFilter } p = \text{Some } t \wedge \text{set-tries } t \subseteq_{\simeq} \text{set arch}$)
apply(*metis TameEnum-tameEnumFilter qle-gr.seteq-qle-def qle-gr.subseteq-qle-trans*)
apply(*fastforce intro!: samet-imp-iso-subseteq*
dest: inv-tries-tameEnumFilter tameEnumFilter-subseteq-TameEnum mgp-TameEnum
mgp-pre-iso-test)
done

end

27 Completeness Proofs under hypothetical computations

theory *Relative-Completeness*

imports *ArchCompProps*

begin

definition *Archive* :: *vertex fgraph set where*

Archive $\equiv \text{set}(Tri @ Quad @ Pent @ Hex)$

```

locale archive-by-computation =
  assumes pre-iso-test3:  $\forall g \in \text{set } \text{Tri. } \text{pre-iso-test } g$ 
  assumes pre-iso-test4:  $\forall g \in \text{set } \text{Quad. } \text{pre-iso-test } g$ 
  assumes pre-iso-test5:  $\forall g \in \text{set } \text{Pent. } \text{pre-iso-test } g$ 
  assumes pre-iso-test6:  $\forall g \in \text{set } \text{Hex. } \text{pre-iso-test } g$ 
  assumes same3: samet (tameEnumFilter 0) Tri
  assumes same4: samet (tameEnumFilter 1) Quad
  assumes same5: samet (tameEnumFilter 2) Pent
  assumes same6: samet (tameEnumFilter 3) Hex
begin

theorem TameEnum-Archive: fgraph ' TameEnum  $\subseteq_{\approx}$  Archive
using combine-vals-filter[OF pre-iso-test3 same3]
  combine-vals-filter[OF pre-iso-test4 same4]
  combine-vals-filter[OF pre-iso-test5 same5]
  combine-vals-filter[OF pre-iso-test6 same6]
by(fastforce simp:TameEnum-def Archive-def image-def qle-gr.defs
  eval-nat-numeral le-Suc-eq)

lemma TameEnum-comp:
assumes Seedp [next-planep] $\rightarrow^*$  g and final g and tame g
shows Seedp [next-tamep] $\rightarrow^*$  g
proof –
  from assms have Seedp [next-tame0 p] $\rightarrow^*$  g by(rule next-tame0-comp)
  with assms show Seedp [next-tamep] $\rightarrow^*$  g by(blast intro: next-tame-comp)
qed

lemma tame5:
assumes g: Seedp [next-plane0 p] $\rightarrow^*$  g and final g and tame g
shows  $p \leq 3$ 
proof –
  from  $\langle \textit{tame } g \rangle$  have bound:  $\forall f \in \mathcal{F} \textit{ g. } |\textit{vertices } f| \leq 6$ 
  by (simp add: tame-def tame9a-def)
  show  $p \leq 3$ 
  proof (rule ccontr)
    assume 6:  $\neg p \leq 3$ 
    obtain f where  $f \in \textit{set } (\textit{finals } g) \wedge |\textit{vertices } f| = p+3$ 
    using max-face-ex[OF g] by auto
    also from  $\langle \textit{final } g \rangle$  have  $\textit{set } (\textit{finals } g) = \textit{set } (\textit{faces } g)$  by simp
    finally have  $f \in \mathcal{F} \textit{ g} \wedge 6 < |\textit{vertices } f|$  using 6 by arith
    with bound show False by auto
  qed
qed

theorem completeness:
assumes g  $\in$  PlaneGraphs and tame g shows fgraph g  $\in_{\approx}$  Archive
proof –
  from  $\langle g \in \textit{PlaneGraphs} \rangle$  obtain p where g1: Seedp [next-planep] $\rightarrow^*$  g

```

```

and final g
by(auto simp:PlaneGraphs-def PlaneGraphsP-def)
have Seedp [next-plane0p]→* g
by(rule RTranCl-subset2[OF g1])
  (blast intro:inv-mgp inv-Seed mgp-next-plane0-if-next-plane
  dest:RTranCl-inv[OF inv-inv-next-plane])
with ⟨tame g⟩ ⟨final g⟩ have  $p \leq 3$  by(blast intro:tame5)
with g1 ⟨tame g⟩ ⟨final g⟩ show ?thesis using TameEnum-Archive
by(simp add: qle-gr.defs TameEnum-def TameEnumP-def)
  (blast intro: TameEnum-comp)
qed

end

end

```


References

- [1] Tobias Nipkow, Gertrud Bauer, and Paula Schultz. Flyspeck I: Tame graphs. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *LNCS*, pages 21–35. Springer, 2006.