Tame Plane Graphs

Gertrud Bauer and Tobias Nipkow

February 2, 2006

Contents

1 Basic Functions Old and New

theory *ListAux*
imports *Main EfficientNat*
begin

declare *Let-def* [*simp*]

declare *comp-def* [*code unfold*]

1.1 HOL

lemma *pairD*: (*a*,*b*) = *p* ⟹ *a* = *fst p* ∧ *b* = *snd p*
by *auto*

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

lemma [*code unfold*]: *max x y* == (*let u* = *x*; *v* = *y in if u* <= *v then v else u*)
by(*simp add*:*max-def*)
lemma [*code unfold*]: *min x y* == (*let u* = *x*; *v* = *y in if u* <= *v then u else v*)
by(*simp add*:*min-def*)

lemmas[*code*] = *lessThan-0 lessThan-Suc*

1.2 Lists

declare *mem-iff* [*simp*] *list-all-iff* [*simp*] *list-ex-iff* [*simp*]

1.2.1 *length*

syntax *-length* :: *′a list* ⇒ *nat* (|-|)
translations
|*xs*| == *length xs*

lemma *length3D*: |*xs*| = *3* ⟹ ∃ *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* ⟹ ∃ *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*]: (*filter P xs* = []) ⟹ *x* ∈ *set xs* ⟹ ¬ *P x*

```
  by (simp add: filter-empty-conv)

lemma filter-comm: [x ∈ xs. P x ∧ Q x] = [x ∈ xs. Q x ∧ P x]
  by (simp add: conj-aci)

lemma filter1:  [x∈xs . P x] = [] ⟹
    [x∈ xs. Q x ∧ P x] = []
  by (induct xs)  (simp-all split: split-if-asm)

lemma filter-prop: ⋀x. x ∈ set [u∈ys . P u] ⟹ P x
proof (induct ys)
  case Nil then show ?case by simp
next
  case (Cons y ys) then show ?case
    by (auto split: split-if-asm)
qed

lemma filter-Cons-prop: [u∈ys . P u] = x # xs ⟹ P x
proof −
  assume [u∈ys . P u] = x # xs
  then have x ∈ set [u∈ys . P u] by simp
  then show P x by (rule filter-prop)
qed

lemma filter-compl1:
 ([x ∈ xs. P x] = []) = ([x ∈ xs. ¬ P x] = xs) (is ?lhs = ?rhs)
proof
  show ?rhs ⟹ ?lhs
  proof (induct xs)
    case Nil then show ?case by simp
  next
    case (Cons x xs)
    have [u∈xs . ¬ P u] ≠ x # xs
    proof
      assume [u∈xs . ¬ P u] = x # xs
      then have |x # xs| = |[u∈xs . ¬ P u]| by simp
      also have ... ≤ |xs| by simp
      finally show False by simp
    qed
    with Cons show ?case by auto
  qed
next
  show ?lhs ⟹ ?rhs
    by (induct xs) (simp-all split: split-if-asm)
qed
lemma [simp]: Not ∘ (Not ∘ P) = P
  by (rule ext) simp

lemma filter-compl2: ⋀P. (filter (Not ∘ P) xs = []) = (filter P xs = xs)
```

```
  by (simp add: filter-compl1)

lemma filter-eqI:
  (⋀v. v ∈ set vs ⟹ P v = Q v) ⟹ [v ∈ vs . P v] = [v ∈ vs . Q v]
  by (induct vs) simp-all

lemma filter-simp: (⋀x. x ∈ set xs ⟹ P x) ⟹ [x ∈ xs. P x ∧ Q x] = [x ∈ xs.
Q x]
 by (induct xs) auto

lemma filter-True-eq1:
  (length [y ∈ xs. P y] = length xs) ⟹ (⋀y. y ∈ set xs ⟹ P y)
proof (induct xs)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  have l: length (filter P xs) ≤ length xs
    by (simp add: length-filter-le)
  have hyp: length (filter P (x # xs)) = length (x # xs) .
  then have P x  by (simp split: split-if-asm) (insert l, arith)
  moreover with hyp have length (filter P xs) = length xs
    by (simp split: split-if-asm)
  moreover have y ∈ set (x#xs) .
  ultimately show ?case by (auto dest: Cons(1))
qed

lemma length-filter-True-eq:
  (length [y ∈ xs. P y] = length xs) = (∀ y. y ∈ set xs ⟶ P y)
  by (intro iffI allI impI, erule filter-True-eq1) simp-all
```

1.2.3 *map*

```
syntax (xsymbols)
  @map :: [ 'b, pttrn, 'a list] => 'a list((1[-. - ∈ -]))
syntax
  @map :: [ 'b, pttrn, 'a list] => 'a list((1[-./ - : -]))

translations
  [f. x ∈ xs]== map (λx. f) xs
  [f. x : xs]== map (λx. f) xs
```

1.2.4 *map-filter*

```
syntax (xsymbols)
  @map-filter :: ['b, pttrn, 'a list, bool] => 'a list((1[-. - ∈ -, -]))
syntax
  @map-filter :: ['b, pttrn, 'a list, bool] => 'a list((1[-./ - : -, -]))

translations
  [f. x ∈ xs, P]== map-filter (λx. f) P xs
```

$[f.\ x : \mathit{xs},\ P] == \mathit{map\text{-}filter}\ (\lambda x.\ f)\ P\ \mathit{xs}$

lemma [*simp*]: $[f\,x.\ x \in \mathit{xs},\ P] = [f\,x.\ x \in [x \in \mathit{xs}.\ P\ x]]$
by (*induct xs*) *auto*

1.2.5 *concat*

syntax (*xsymbols*)
$@\mathit{concat}\quad :: \mathit{idt} => {'a}\ \mathit{list} => {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}\ \ (\bigsqcup_{\text{-}\in\text{ -}}\ \text{- } \mathit{10})$
translations $\bigsqcup_{x \in \mathit{xs}} f == \mathit{concat}\ [f.\ x \in \mathit{xs}]$

1.2.6 List product

constdefs $\mathit{listProd1} :: {'a} \Rightarrow {'b}\ \mathit{list} \Rightarrow ({'a} \times {'b})\ \mathit{list}$
$\mathit{listProd1}\ a\ \mathit{bs} \equiv [(a,b).\ b \in \mathit{bs}]$

constdefs $\mathit{listProd} :: {'a}\ \mathit{list} \Rightarrow {'b}\ \mathit{list} \Rightarrow ({'a} \times {'b})\ \mathit{list}$ (**infix** $\times$ *50*)
$\mathit{as} \times \mathit{bs} \equiv \bigsqcup_{a\ \in\ \mathit{as}} \mathit{listProd1}\ a\ \mathit{bs}$

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

1.2.7 Minimum and maximum

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

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

lemma *minimal-Cons1*:
$\forall y \in \mathit{set}\ \mathit{xs}.\ f\,x \leq f\,y \Longrightarrow \mathit{minimal}\,f\ (x\#\mathit{xs}) = x$
by (*induct xs*) *auto*

lemma *minimal-append2*:
$\forall x \in \mathit{set}\ \mathit{xs}.\ f\,x > f\,y \Longrightarrow \mathit{minimal}\,f\ (\mathit{xs}\ @\ y\ \#\ \mathit{ys}) = \mathit{minimal}\,f\ (y\ \#\ \mathit{ys})$
by (*induct xs*) *simp-all*

lemma *minimal-neq-lowerbound*:
$\mathit{xs} \neq [] \Longrightarrow \mathit{ALL}\ x{:}\ \mathit{set}\ \mathit{xs}.\ f\,x \geq n \Longrightarrow f(\mathit{minimal}\,f\,\mathit{xs}) \neq n$
$\Longrightarrow \mathit{ALL}\ x{:}\ \mathit{set}\ \mathit{xs}.\ f\,x \neq n$
apply(*induct xs*)
apply *simp*
apply (*auto split*:*split-if-asm*)
done

consts $minList :: nat\ list \Rightarrow nat$
primrec
$minList\ (x\#xs) = (if\ xs{=}[]\ then\ x\ else\ min\ x\ (minList\ xs))$

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

lemma $minList\text{-}conv\text{-}Min[simp]$:
$xs \neq [] \Longrightarrow minList\ xs = Min(set\ xs)$
by $(induct\ xs)\ auto$

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

1.2.8 replace

consts $replace :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow\ 'a\ list$
primrec
$replace\ x\ ys\ [] = []$
$replace\ x\ ys\ (z\#zs) =$
$\quad (if\ z = x\ then\ ys\ @\ zs\ else\ z\ \#\ (replace\ x\ ys\ zs))$

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

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

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

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

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

lemma $len\text{-}replace\text{-}ge[simp]$:

⟦ *length ys* ≥ *1*; *length xs* ≥ *length zs* ⟧ ⟹
length(*replace x ys xs*) ≥ *length zs*
apply(*drule len-replace-ge-same*[**where** *x* = *x* **and** *xs* = *xs*])
apply *arith*
done

lemma *replace-append*[*simp*]:
  replace x ys (*as* @ *bs*) =
    (*if x* ∈ *set as then replace x ys as* @ *bs else as* @ *replace x ys bs*)
by(*induct as*) *auto*

lemma *filter-replace*:
¬ *P y* ⟹ *filter P* (*replace x* [*y*] *xs*) = *remove1 x* (*filter P xs*)
by(*induct xs*) *simp-all*

lemma *distinct-set-replace*: *distinct xs* ⟹
set (*replace x ys xs*) =
(*if x* ∈ *set xs then* (*set xs* − {*x*}) ∪ *set ys else set xs*)
apply(*induct xs*)
 apply(*simp*)
apply *simp*
apply *blast*
done

lemma *replace1*:
f ∈ *set* (*replace f′ fs ls*) ⟹ *f* ∉ *set ls* ⟹ *f* ∈ *set fs*
proof (*induct ls*)
  case *Nil* **then show** *?case* **by** *simp*
next
  case (*Cons l ls*) **then show** *?case* **by** (*simp split*: *split-if-asm*)
qed

lemma *replace2*:
f′ ∉ *set ls* ⟹ *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*: *split-if-asm*)
qed

lemma *replace3*[*intro*]:
  f′ ∈ *set ls* ⟹ *f* ∈ *set fs* ⟹ *f* ∈ *set* (*replace f′ fs ls*)
by (*induct ls*) *auto*

lemma *replace4*:
  f ∈ *set ls* ⟹ *oldF* ≠ *f* ⟹ *f* ∈ *set* (*replace oldF fs ls*)
by (*induct ls*) *auto*

lemma *replace5*: *f* ∈ *set* (*replace oldF newfs fs*) ⟹ *f* ∈ *set fs* ∨ *f* ∈ *set newfs*
by (*induct fs*) (*auto split*: *split-if-asm*)

lemma *replace6*: *distinct oldfs* ⟹ *x* ∈ *set* (*replace oldF newfs oldfs*) =
((*x* ≠ *oldF* ∨ *oldF* ∈ *set newfs*) ∧ ((*oldF* ∈ *set oldfs* ∧ *x* ∈ *set newfs*) ∨ *x* ∈ *set oldfs*))
by (*induct oldfs*) *auto*

lemma *replace-delete-oldF*:
oldF ∉ *set fs* ⟹ *distinct ls* ⟹ *oldF* ∉ *set* (*replace oldF fs ls*)
by (*induct ls*) *auto*

lemma *distinct-replace*:
distinct fs ⟹ *distinct newFs* ⟹ *set fs* ∩ *set newFs* ⊆ {*oldF*} ⟹
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* ∉ *set newFs* **by** *simp blast*
with *Cons* **have** *f* ∉ *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* ∉ *set newfs* ⟹ *distinct xs* ⟹
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* ⟹ *distinct newfs* ⟹ *oldf* ∈ *set fs* ⟶ *set newfs* ∩ *set fs* ⊆ {*oldf*} ⟹
distinct (*replace oldf newfs fs*)
apply (*case-tac oldf* ∈ *set fs*) **apply** *simp*
apply (*induct fs*) **apply** *simp*
apply (*auto simp*: *replace2*) **apply** (*drule replace1*)
by *auto*

lemma *filter-replace2*:
⟦ ¬ *P x*; ∀ *y*∈ *set ys*. ¬ *P y* ⟧ ⟹

```
  filter P (replace x ys xs) = filter P xs
apply(cases x ∈ set xs)
 prefer 2 apply(simp add:replace2)
apply(induct xs)
 apply simp
apply clarsimp
done

lemma length-filter-replace1:
⟦ x ∈ set xs; ¬ P x ⟧ ⟹
  length(filter P (replace x ys xs)) =
  length(filter P xs) + length(filter P ys)
apply(induct xs)
 apply simp
apply fastsimp
done

lemma length-filter-replace2:
 ⟦ x ∈ set xs; P x ⟧ ⟹
  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
```

1.2.9 *distinct*

```
lemma dist-filter-single:
distinct ls ⟹ v ∈ set ls ⟹ [a∈ ls . a = v] = [v]
proof (induct ls)
  case Nil then show ?case by auto
next
  case (Cons l ls) then show ?case
  proof (cases l = v)
    case True with Cons
    have v ∉ set ls by auto
    then have [a∈ls . a = v] = [] by (induct ls) auto
    with Cons True show ?thesis by auto
  next
    case False with Cons show ?thesis by auto
  qed
qed

lemma dist-at1: ⋀ c vs. distinct vs ⟹ vs = a @ r # b ⟹ vs = c @ r # d ⟹
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 ∉ 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 ≠ [] ⟹ hd c = r by auto
  then have c ≠ [] ⟹ r ∈ 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 ⟹ vs = a @ r # b ⟹ vs = c @ r # d ⟹ a = c
∧ 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 ⟹ vs = a @ r # b ⟹ vs = c @ r # d ⟹ b = d
proof −
  assume dist: distinct vs and vs1: vs = a @ r # b and vs2: vs = c @ r # d
  then have a = c ∧ b = d by (rule-tac dist-at) auto
  then show ?thesis by auto
qed

lemma distinct-split1: distinct xs ⟹ xs = y @ [r] @ z  ⟹ r ∉ set y
  apply auto done

lemma distinct-split2: distinct xs ⟹ xs = y @ [r] @ z  ⟹ r ∉ set z apply auto
done

lemma distinct-hd-not-cons: distinct vs ⟹ ∃ as bs. vs = as @ x # hd vs # bs
⟹ False
proof −
  assume d: distinct vs and ex: ∃ 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
```

1.2.10 Misc

lemma *drop-last-in*: !!*n*. *n* < *length ls* ⟹ *last ls* ∈ *set* (*drop n 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*: *distinct ls* ⟹ *n* < *length ls* ⟹ *last ls* = *ls* ! *n* ⟹ *Suc*
n = *length ls*
proof (*rule ccontr*)
  assume *d*: *distinct ls* **and** *n*: *n* < *length ls* **and** *l*: *last ls* = *ls* ! *n* **and** *not*: *Suc*
n ≠ *length ls*
  then have *s*: *Suc n* < *length ls* **by** *auto*
  def *lls* ≡ *ls*!*n*
  with *n* **have** *take* (*Suc n*) *ls* = *take n ls* @ [*lls*] **apply** *simp* **by** (*rule take-Suc-conv-app-nth*)
  then have *take* (*Suc n*) *ls* @ *drop* (*Suc n*) *ls* = *take n ls* @ [*lls*] @ *drop* (*Suc n*)
ls **by** *auto*
  then have *ls*: *ls* = *take n ls* @ [*lls*] @ *drop* (*Suc n*) *ls* **by** *auto*
  with *d* **have** *dls*: *distinct* (*take n ls* @ [*lls*] @ *drop* (*Suc n*) *ls*) **by** *auto*
  from *lls-def l* **have** *lls* = (*last ls*) **by** *auto*
  with *s* **have** *lls* ∈ *set* (*drop* (*Suc n*) *ls*) **apply** *simp* **by** (*rule-tac drop-last-in*)
  with *dls* **show** *False* **by** *auto*
qed

1.2.11 *rotate*

lemma *plus-length1*[*simp*]: *rotate* (*k*+(*length ls*)) *ls* = *rotate k ls*
proof −
  have ⋀ *k ls*. *rotate k* (*rotate* (*length ls*) *ls*) = *rotate* (*k*+(*length ls*)) *ls*
    by (*rule rotate-rotate*)
  then have ⋀ *k ls*. *rotate k ls* = *rotate* (*k*+(*length ls*)) *ls* **by** *auto*
  then show *?thesis* **by** (*rule sym*)
qed

lemma *plus-length2*[*simp*]: *rotate* ((*length ls*)+*k*) *ls* = *rotate k ls*
proof −
  def *x* ≡ (*length ls*)+*k*
  then have *x* = *k*+(*length ls*) **by** *auto*
  with *x-def* **have** *rotate x ls* = *rotate* (*k*+(*length ls*)) *ls* **by** *simp*
  then have *rotate x ls* = *rotate k ls* **by** *simp*
  with *x-def* **show** *?thesis* **by** *simp*
qed

lemma *rotate-minus1*: *n* > *0* ⟹ *m* > *0* ⟹
 rotate n ls = *rotate m ms* ⟹ *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 ⟹ rotate n ls = ms ⟹
  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: ⋀ ms. n < length ls ⟹ rotate n ls = ms ⟹
  ls = rotate ((length ls) − n) 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]: rotate (n mod length ls) ls = rotate n ls
by(simp add:rotate-drop-take)

lemma rotate-inv2: rotate n ls = ms ⟹
 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-inv′: ls = rotate ((length ls) − (n mod length ls)) ms ⟹
 rotate n ls = ms
proof (cases ls  = [])
  assume r: ls = rotate ((length ls) − (n mod length ls)) ms
  case True with r show ?thesis by auto
next
  assume r: ls = rotate ((length ls) − (n mod length ls)) ms
  case False
  def len ≡ length ls
  with r have r′: ls = rotate (len − (n mod len)) ms by simp
  with len-def have lms: length ms = len by auto
  def y ≡ n mod len
  from len-def False have ll: len > 0 by auto
  with y-def have small-y: y < len by auto
  then have len-gz: len > 0 by auto
  from r′ len-def have mem0: rotate n ls = rotate n (rotate (len − (n mod len))
ms) by auto
   have rotate n (rotate (len − (n mod len)) ms) = rotate (n + (len − (n mod
len))) ms
     by (rule rotate-rotate)
  with mem0 have mem1: rotate n ls = rotate (n + (len − (n mod len))) ms by
auto
  have n mod len − n mod len = 0 by arith
  then have len + n mod len − n mod len = len by arith
  from y-def have ymod: n mod len = y by auto
  with len-gz obtain r where n: n = y + r∗len apply (drule-tac mod-eqD) by
auto
  then have n + len = (r∗len)+len  + y by arith
  then have n + len = ((r+1)∗len) + y by auto
  with len-gz small-y have n + (len − y) = ((r+1)∗len) by auto
  then have zero: (n + (len − y)) mod len = 0 by auto
  have rotate (n + (len − y)) ms = rotate ((n + (len − y)) mod length ms) ms
by (rule-tac rotate-conv-mod)
  with lms zero have rotate (n + (len − y)) ms = ms by auto
  with mem1 y-def show ?thesis 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 ⟹ ls!(Suc n) = (rotate1 ls)!n
  apply (auto simp: rotate1-def) 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 simp*: *rotate1-def*)

lemma *nth-rotate1*: *0 < length ls ⟹ 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*
  def *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*:*split-if-asm*)
  with *a m* **show** *?thesis* **apply** (*drule-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*: *split-if-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*: ⋀ *ls*. *0 < length ls ⟹ 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*)
  def *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

lemma *help1*: ∃ *n*. *filter f* (*rotate1 ls*) = *rotate n* (*filter f ls*)
proof (*cases ls*)
  case *Nil* **then show** *?thesis* **by** *auto*
next
  case (*Cons m ms*)
  then show *?thesis*

```
  proof (cases f m)
    case True with Cons
    have filter f (rotate1 (ls)) = rotate 1 (filter f (ls)) by auto
    then show ?thesis apply (rule-tac exI) .
  next
    case False with Cons
    have filter f (rotate1 (ls)) = rotate 0 (filter f (ls)) by auto
    then show ?thesis apply (rule-tac exI) .
  qed
qed

lemma help3: ¬ f l ⟹ n < length ls ⟹ filter f (rotate n ls) = filter f (rotate n
(rotate1 (l # ls)))
proof −
  assume fl: ¬ f l and n: n < length ls
  then have n2: n − length ls = 0 by auto
  from fl n show ?thesis apply simp
    apply (subst rotate-drop-take)+ by auto
qed

lemma help4: ¬ f l ⟹ n < length ls ⟹ filter f (rotate n ls) = filter f (rotate
(Suc n) (l # ls))
proof −
  assume fl: ¬ f l and n: n < length ls
   then have r1: filter f (rotate n ls) = filter f (rotate n (rotate1 (l # ls))) by
(rule help3)
  def ms ≡ l # ls
  have rotate n (rotate1 (ms)) = rotate (Suc n) (ms) by simp
  with ms-def have rotate n (rotate1 (l # ls)) = rotate (Suc n) (l # ls) by simp
  with r1 show ?thesis by auto
qed

lemma help2: ∃ n. rotate1 (filter f ls) = filter f (rotate n ls)
proof (induct ls)
  case Nil then show ?case by auto
next
  case (Cons m ms)
  then show ?case
  proof (cases f m)
    case True with Cons
    have filter f (m # ms) = m # filter f ms by auto
    from True Cons have
      rotate1 (filter f (m # ms)) = filter f (rotate 1 (m # ms)) by auto
    then show ?thesis apply (rule-tac exI) .
  next
    case False with Cons
    have f1: filter f (m # ms) = filter f ms by auto
    then have mem: rotate1 (filter f (m # ms)) = rotate1 (filter f ms) by auto
    from False have f2: ⋀ ks. filter f (ks @ [m]) = filter f ks by auto
```

from *Cons* **obtain** *n* **where** *n*: *rotate1* (*filter f ms*) = *filter f* (*rotate n ms*) **by** *auto*
with *mem* **have** *mem1*: *rotate1* (*filter f* (*m* # *ms*)) = *filter f* (*rotate n ms*) **by** *auto*
def *n′* ≡ *n mod length ms*
then have *filter f* (*rotate n ms*) = *filter f* (*rotate n′ ms*) **by** *auto*
with *mem1* **have** *mem2*: *rotate1* (*filter f* (*m* # *ms*)) = *filter f* (*rotate n′ ms*) **by** *auto*
from *n′-def* **have** *ms* ≠ [] ⟹ *n′* < *length ms* **by** *auto*
with *False* **have** *filter f* (*rotate n′ ms*) = *filter f* (*rotate* (*Suc n′*) (*m* # *ms*))
apply (*case-tac ms* = []) **apply** *force* **apply** (*rule-tac help4*) **by** *auto*
with *mem2* **have** *rotate1* (*filter f* (*m* # *ms*)) = *filter f* ((*rotate* (*Suc n′*) (*m* # *ms*))) **by** *auto*
then show *?thesis* **apply** (*rule-tac exI*) **.**
qed
qed

lemma *rotate-help5*: ∃ *n*. *filter f* (*rotate m ls*) = *rotate n* (*filter f ls*)
proof (*induct m*)
case *0* **then have** *filter f* (*rotate 0 ls*) = *rotate 0* (*filter f ls*) **by** *auto*
then show *?case* **by** (*rule exI*)
next
case (*Suc m*)
def *ks* ≡ *rotate m ls*
then have *mem0*: *filter f* (*rotate* (*Suc m*) *ls*) = *filter f* (*rotate1 ks*) **by** *auto*
from *ks-def Suc* **obtain** *n* **where** *n*: *filter f ks* = *rotate n* (*filter f ls*) **by** *auto*
have ∃ *n′*. *filter f* (*rotate1 ks*) = *rotate n′* (*filter f ks*) **by** (*rule help1*)
then obtain *n′* **where** *filter f* (*rotate1 ks*) = *rotate n′* (*filter f ks*) **by** *auto*
with *n* **have** *filter f* (*rotate1 ks*) = *rotate n′* (*rotate n* (*filter f ls*)) **by** *auto*
with *mem0* **have** *mem1*: *filter f* (*rotate* (*Suc m*) *ls*) = *rotate n′* (*rotate n* (*filter f ls*)) **by** *auto*
have *rotate n′* (*rotate n* (*filter f ls*)) = *rotate* (*n′*+*n*) (*filter f ls*) **by** (*rule rotate-rotate*)
with *mem1* **have** *filter f* (*rotate* (*Suc m*) *ls*) = *rotate* (*n′*+*n*) (*filter f ls*) **by** *auto*
then show *?case* **apply** (*rule-tac exI*) **.**
qed

lemma *rotate-help6*: ∃ *n*. *rotate m* (*filter f ls*) = *filter f* (*rotate n ls*)
proof (*induct m*)
case *0* **then have** *rotate 0* (*filter f ls*) = *filter f* (*rotate 0 ls*) **by** *auto*
then show *?case* **by** (*rule exI*)
next
case (*Suc m*)
have *mem0*: *rotate* (*Suc m*) (*filter f ls*) = *rotate1* (*rotate m* (*filter f ls*)) **by** *auto*
from *Suc* **obtain** *n* **where** *n*: *rotate m* (*filter f ls*) = *filter f* (*rotate n ls*) **by** *auto*
with *mem0* **have** *mem1*: *rotate* (*Suc m*) (*filter f ls*) = *rotate1* (*filter f* (*rotate n ls*)) **by** *auto*

def *ks ≡ rotate n ls*
from *help2* **obtain** *n′* **where** *rotate1 (filter f ks) = filter f (rotate n′ ks)* **by** *auto*
with *mem1 ks-def* **have** *mem2: rotate (Suc m) (filter f ls) = filter f (rotate n′ ks)* **by** *auto*
from *ks-def* **have** *rotate n′ ks = rotate (n′+n) ls* **apply** *simp* **by** *(rule rotate-rotate)*
with *mem2* **have** *rotate (Suc m) (filter f ls) = filter f(rotate (n′+n) ls)* **by** *simp*
then show *?case* **apply** *(rule-tac exI)* **.**
qed

1.3 *splitAt*

consts *splitAtRec* ::
′a ⇒ ′a list ⇒ ′a list ⇒ ′a list × ′a list
primrec
splitAtRec c bs [] = (bs,[])
splitAtRec c bs (a#as) = (if a = c then (bs, as)
else splitAtRec c (bs@[a]) as)

constdefs *splitAt* :: *′a ⇒ ′a list ⇒ ′a list × ′a list*
splitAt c as ≡ splitAtRec c [] as

1.3.1 *splitAtRec*

lemma *splitAtRec-conv*: *!!bs.*
splitAtRec x bs xs =
(bs @ takeWhile (%y. y≠x) xs, tl(dropWhile (%y. y≠x) xs))
by*(induct xs) auto*

lemma *splitAtRec-distinct-fst*: *⋀ s. distinct vs ⟹ distinct s ⟹ (set s) ∩ (set vs) = {} ⟹ distinct (fst (splitAtRec ram1 s vs))*
by *(induct vs) auto*

lemma *splitAtRec-distinct-snd*: *⋀ s. distinct vs ⟹ distinct s ⟹ (set s) ∩ (set vs) = {} ⟹ distinct (snd (splitAtRec ram1 s vs))*
by *(induct vs) auto*

lemma *splitAtRec-ram*:
⋀ us a b. ram ∈ set vs ⟹ (a, b) = splitAtRec ram us vs ⟹
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: split-if-asm)*
qed

lemma *splitAtRec-notRam*:
⋀ us. ram ∉ set vs ⟹ 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*: ⋀ *s*. *distinct vs* ⟹
distinct s ⟹ (*set s*) ∩ (*set vs*) = {} ⟹
set (*fst* (*splitAtRec ram s vs*)) ∩ *set* (*snd* (*splitAtRec ram s vs*)) = {}
by (*induct vs*) *auto*

1.3.2 *splitAt*

lemma *splitAt-conv*:
splitAt x xs = (*takeWhile* (%*y*. *y*≠*x*) *xs*, *tl*(*dropWhile* (%*y*. *y*≠*x*) *xs*))
by(*simp add*: *splitAt-def splitAtRec-conv*)

lemma *splitAt-no-ram*[*simp*]:
ram ∉ *set vs* ⟹ *splitAt ram vs* = (*vs*, [])
by (*auto simp*: *splitAt-def splitAtRec-notRam*)

lemma *splitAt-split*:
ram ∈ *set vs* ⟹ (*a*,*b*) = *splitAt ram vs* ⟹ *vs* = *a* @ *ram* # *b*
by (*auto simp*: *splitAt-def dest*: *splitAtRec-ram*)

lemma *splitAt-ram*:
ram ∈ *set vs* ⟹ *vs* = *fst* (*splitAt ram vs*) @ *ram* # *snd* (*splitAt ram vs*)
by (*rule-tac splitAt-split*) *auto*

lemma *fst-splitAt-last*:
⟦ *xs* ≠ []; *distinct xs* ⟧ ⟹ *fst* (*splitAt* (*last xs*) *xs*) = *butlast xs*
by(*simp add*:*splitAt-conv takeWhile-not-last*)

1.3.3 Sets

lemma *splitAtRec-union*:
⋀ *a b s*. (*a*,*b*) = *splitAtRec ram s vs* ⟹ (*set a* ∪ *set b*) − {*ram*} = (*set vs* ∪ *set s*) − {*ram*}
apply (*induct vs*) **by** (*auto split*: *split-if-asm*)

lemma *splitAt-union*:
(*a*,*b*) = *splitAt ram vs* ⟹ (*set a* ∪ *set b*) − {*ram*} = *set vs* − {*ram*}
by (*simp add*: *splitAt-def splitAtRec-union*)

lemma *splitAt-subset-ab*:
(*a*,*b*) = *splitAt ram vs* ⟹ *set a* ⊆ *set vs* ∧ *set b* ⊆ *set vs*
apply (*cases ram* ∈ *set vs*)
by (*auto dest*: *splitAt-split simp*: *splitAt-no-ram*)

lemma *splitAt-subset-fst*:
set (*fst* (*splitAt ram vs*)) ⊆ *set vs*
proof −

```
  def a: a ≡ fst (splitAt ram vs)
  def b: b ≡ snd (splitAt ram vs)
  with a have (a,b) = splitAt ram vs by auto
  with a show ?thesis by (auto dest: splitAt-subset-ab)
qed

lemma splitAt-subset-snd:
set (snd (splitAt ram vs)) ⊆ set vs
proof −
  def a: a ≡ fst (splitAt ram vs)
  def b: b ≡ snd (splitAt ram vs)
  with a have (a,b) = splitAt ram vs by auto
  with b show ?thesis by (auto dest: splitAt-subset-ab)
qed

lemma splitAt-in-fst[dest]: v ∈ set (fst (splitAt ram vs)) ⟹ v ∈ set vs
proof (cases ram ∈ set vs)
  assume v: v ∈ set (fst (splitAt ram vs))
  def a ≡ fst (splitAt ram vs)
  with v have vin: v ∈ set a by auto
  def 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 ∉ set vs ⟹ v ∉ set (fst (splitAt ram vs)) by (auto dest: splitAt-in-fst)

lemma splitAt-in-snd[dest]: v ∈ set (snd (splitAt ram vs)) ⟹ v ∈ set vs
proof (cases ram ∈ set vs)
  assume v: v ∈ set (snd (splitAt ram vs))
  def a ≡ fst (splitAt ram vs)
  def 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:
 distinct vs ⟹ (a,b) = splitAt ram vs ⟹ distinct a ∧ distinct b
apply (cases ram ∈ set vs) by (auto dest: splitAt-split simp: splitAt-no-ram)
```

lemma *splitAt-distinct-a*:
distinct vs ⟹ (*a*,*b*) = *splitAt ram vs* ⟹ *distinct a*
apply (*cases ram* ∈ *set vs*) **by** (*auto dest*: *splitAt-split simp*: *splitAt-no-ram*)

lemma *splitAt-distinct-b*:
distinct vs ⟹ (*a*,*b*) = *splitAt ram vs* ⟹ *distinct b*
apply (*cases ram* ∈ *set vs*) **by** (*auto dest*: *splitAt-split simp*: *splitAt-no-ram*)

lemma *splitAt-distinct-fst*[*intro*]:
distinct vs ⟹ *distinct* (*fst* (*splitAt ram vs*))
proof −
assume *d*: *distinct vs*
def *b*: *b* ≡ *snd* (*splitAt ram vs*)
def *a*: *a* ≡ *fst* (*splitAt ram vs*)
with *b* **have** (*a*,*b*) = *splitAt ram vs* **by** *auto*
with *a d* **show** *?thesis* **by** (*auto dest*: *splitAt-distinct-ab*)
qed

lemma *splitAt-distinct-snd*[*intro*]:
distinct vs ⟹ *distinct* (*snd* (*splitAt ram vs*))
proof −
assume *d*: *distinct vs*
def *b*: *b* ≡ *snd* (*splitAt ram vs*)
def *a*: *a* ≡ *fst* (*splitAt ram vs*)
with *b* **have** (*a*,*b*) = *splitAt ram vs* **by** *auto*
with *b d* **show** *?thesis* **by** (*auto dest*: *splitAt-distinct-ab*)
qed

lemma *splitAt-distinct-ab*:
distinct vs ⟹ (*a*,*b*) = *splitAt ram vs* ⟹ *set a* ∩ *set b* = {}
apply (*cases ram* ∈ *set vs*) **apply** (*drule-tac splitAt-split*)
by (*auto simp*: *splitAt-no-ram*)

lemma *splitAt-distinct-fst-snd*:
distinct vs ⟹ *set* (*fst* (*splitAt ram vs*)) ∩ *set* (*snd* (*splitAt ram vs*)) = {}
by (*rule-tac splitAt-distinct-ab*) *simp-all*

lemma *splitAt-distinct-ram-fst*[*intro*]:
distinct vs ⟹ *ram* ∉ *set* (*fst* (*splitAt ram vs*))
apply (*case-tac ram* ∈ *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 ⟹ *ram* ∉ *set* (*snd* (*splitAt ram vs*))
apply (*case-tac ram* ∈ *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 ∈ *set vs* ⟹ (*a*,*b*) = *splitAt ram vs* ⟹ *v* ∈ *set a* ∨ *v* ∈ *set b* ∨ *v* = *ram*
apply (*cases ram* ∈ *set vs*)
by (*auto dest*: *splitAt-split simp*: *splitAt-no-ram*)

lemma *splitAt-or*:
v ∈ *set vs* ⟹ *v* ∈ *set* (*fst* (*splitAt ram vs*)) ∨ *v* ∈ *set* (*snd* (*splitAt ram vs*)) ∨ *v* = *ram*
by (*rule splitAt-2*) *auto*

lemma *splitAt-distinct-fst*: *distinct vs* ⟹ *distinct* (*fst* (*splitAt ram1 vs*))
by (*simp add*: *splitAt-def splitAtRec-distinct-fst*)

lemma *splitAt-distinct-a*: *distinct vs* ⟹ (*a*,*b*) = *splitAt ram vs* ⟹ *distinct a*
by (*auto dest*: *splitAt-distinct-fst pairD*)

lemma *splitAt-distinct-snd*: *distinct vs* ⟹ *distinct* (*snd* (*splitAt ram1 vs*))
by (*simp add*: *splitAt-def splitAtRec-distinct-snd*)

lemma *splitAt-distinct-b*: *distinct vs* ⟹ (*a*,*b*) = *splitAt ram vs* ⟹ *distinct b*
by (*auto dest*: *splitAt-distinct-snd pairD*)

lemma *splitAt-distinct*: *distinct vs* ⟹ *set* (*fst* (*splitAt ram vs*)) ∩ *set* (*snd* (*splitAt ram vs*)) = {}
by (*simp add*: *splitAt-def splitAtRec-distinct*)

lemma *splitAt-subset*: (*a*,*b*) = *splitAt ram vs* ⟹ (*set a* ⊆ *set vs*) ∧ (*set b* ⊆ *set vs*)
apply (*cases ram* ∈ *set vs*) **by** (*auto dest*: *splitAt-split simp*: *splitAt-no-ram*)

lemma *splitAt-subset1*: (*a*,*b*) = *splitAt ram vs* ⟹ (*set a* ⊆ *set vs*)
by (*auto dest*: *splitAt-subset*)

lemma *splitAt-subset2*: (*a*,*b*) = *splitAt ram vs* ⟹ (*set b* ⊆ *set vs*)
by (*auto dest*: *splitAt-subset*)

1.3.5 *splitAt* composition

lemma *set-help*: *v* ∈ *set* (*as* @ *bs*) ⟹ *v* ∈ *set as* ∨ *v* ∈ *set bs* **by** *auto*

lemma *splitAt-elements*: *ram1* ∈ *set vs* ⟹ *ram2* ∈ *set vs* ⟹ *ram2* ∈ *set*(*fst* (*splitAt ram1 vs*)) ∨ *ram2* ∈ *set* [*ram1*] ∨ *ram2* ∈ *set*(*snd* (*splitAt ram1 vs*))
proof −
assume *r1*: *ram1* ∈ *set vs* **and** *r2*: *ram2* ∈ *set vs*
then have *ram2* ∈ *set*(*fst* (*splitAt ram1 vs*) @ [*ram1*]) ∨ *ram2* ∈ *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-ram2: ram2 ∉ set (snd (splitAt ram1 vs)) ⟹
  ram1 ∈ set vs ⟹ ram2 ∈ set vs ⟹ ram1 ≠ ram2 ⟹
  ram2 ∈ set (fst (splitAt ram1 vs)) by (auto dest: splitAt-elements)

lemma splitAt-ram3: ram2 ∉ set (fst (splitAt ram1 vs)) ⟹
  ram1 ∈ set vs ⟹ ram2 ∈ set vs ⟹ ram1 ≠ ram2 ⟹
  ram2 ∈ set (snd (splitAt ram1 vs)) by (auto dest: splitAt-elements)

lemma splitAt-dist-ram: distinct vs ⟹
 vs = a @ ram # b ⟹ (a,b) = splitAt ram vs
proof -
  assume dist: distinct vs and vs: vs = a @ ram # b
  from vs have r:ram ∈ 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 ⟹ ram ∈ set vs ⟹ EX! s. vs = (fst s) @
ram # (snd s)
proof
  assume d: distinct vs and r: ram ∈ set vs
  def 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: distinct vs ⟹ vs = a @ ram1 # b @ ram2 # c ⟹
 (a @ ram1 # b, c) = splitAt ram2 vs
by (auto intro: splitAt-dist-ram)

lemma splitAt-dist-ram20: distinct vs ⟹ vs = a @ ram1 # b @ ram2 # c ⟹
  c = snd (splitAt ram2 vs)
proof -
  assume dist: distinct vs and vs: vs = a @ ram1 # b @ ram2 # c
   then show c = snd (splitAt ram2 vs) apply (drule-tac splitAt-dist-ram2) by
```

(*auto dest*: *pairD*)
qed

lemma *splitAt-dist-ram21*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* $\Longrightarrow$ (*a*, *b*) = *splitAt ram1* (*fst* (*splitAt ram2 vs*))
proof −
assume *dist*: *distinct vs* **and** *vs*: *vs* = *a* @ *ram1* # *b* @ *ram2* # *c*
then have *fst* (*splitAt ram2 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*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* $\Longrightarrow$ (*c*, []) = *splitAt ram1* (*snd* (*splitAt ram2 vs*))
proof −
assume *dist*: *distinct vs* **and** *vs*: *vs* = *a* @ *ram1* # *b* @ *ram2* # *c*
then have *snd* (*splitAt ram2 vs*) = *c* **apply** (*drule-tac splitAt-dist-ram2*) **by** (*auto dest*: *pairD*)
with *dist vs* **have** *splitAt ram1* (*snd* (*splitAt ram2 vs*)) = (*c*, []) **by** (*auto intro*: *splitAt-no-ram*)
then show *?thesis* **by** *auto*
qed

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

lemma *splitAt-dist-ram10*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* $\Longrightarrow$ *a* = *fst* (*splitAt ram1 vs*)
proof −
assume *dist*: *distinct vs* **and** *vs*: *vs* = *a* @ *ram1* # *b* @ *ram2* # *c*
then show *a* = *fst* (*splitAt ram1 vs*) **apply** (*drule-tac splitAt-dist-ram1*) **by** (*auto dest*: *pairD*)
qed

lemma *splitAt-dist-ram11*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* $\Longrightarrow$ (*a*, []) = *splitAt ram2* (*fst* (*splitAt ram1 vs*))
proof −
assume *dist*: *distinct vs* **and** *vs*: *vs* = *a* @ *ram1* # *b* @ *ram2* # *c*
then have *fst* (*splitAt ram1 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* $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* $\Longrightarrow$ (*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 ⟹ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c*
⟹ (*a*, *b*) = *splitAt ram1* (*fst* (*splitAt ram2 vs*))
∧ (*c*, []) = *splitAt ram1* (*snd* (*splitAt ram2 vs*))
∧ (*a*, []) = *splitAt ram2* (*fst* (*splitAt ram1 vs*))
∧ (*b*, *c*) = *splitAt ram2* (*snd* (*splitAt ram1 vs*))
∧ *c* = *snd* (*splitAt ram2 vs*)
∧ *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 ⟹ *x* ∈ *set xs* ⟹
fst(*splitAt x* (*rev xs*)) = *rev*(*snd*(*splitAt x xs*))
by(*simp add*:*splitAt-conv takeWhile-neq-rev*)

lemma *snd-splitAt-rev*:
distinct xs ⟹ *x* ∈ *set xs* ⟹
snd(*splitAt x* (*rev xs*)) = *rev*(*fst*(*splitAt x xs*))
by(*simp add*:*splitAt-conv dropWhile-neq-rev*)

lemma *splitAt-take*[*simp*]: *distinct ls* ⟹ *i* < *length ls* ⟹ *fst* (*splitAt* (*ls*!*i*) *ls*) = *take i ls*
proof −
assume *d*: *distinct ls* **and** *si*: *i* < *length ls*
then have *ls1*: *ls* = *take i ls* @ *ls*!*i* # *drop* (*Suc i*) *ls* **by** (*rule-tac id-take-nth-drop*)
from *si* **have** *ls*!*i* ∈ *set ls* **by** *auto*
then have *ls2*: *ls* = *fst* (*splitAt* (*ls*!*i*) *ls*) @ *ls*!*i* # *snd* (*splitAt* (*ls*!*i*) *ls*) **by** (*auto dest*: *splitAt-ram*)
from *d ls2 ls1* **have** *fst* (*splitAt* (*ls*!*i*) *ls*) = *take i ls* ∧ *snd* (*splitAt* (*ls*!*i*) *ls*) = *drop* (*Suc i*) *ls* **by** (*rule dist-at*)
then show *?thesis* **by** *auto*

qed

lemma *splitAt-drop*[*simp*]: *distinct ls* ⟹ *i* < *length ls* ⟹ *snd* (*splitAt* (*ls*!*i*) *ls*) = *drop* (*Suc i*) *ls*
proof −
  assume *d*: *distinct ls* **and** *si*: *i* < *length ls*
  then have *ls1*: *ls* = *take i ls* @ *ls*!*i* # *drop* (*Suc i*) *ls* **by** (*rule-tac id-take-nth-drop*)
  from *si* **have** *ls*!*i* ∈ *set ls* **by** *auto*
  then have *ls2*: *ls* = *fst* (*splitAt* (*ls*!*i*) *ls*) @ *ls*!*i* # *snd* (*splitAt* (*ls*!*i*) *ls*) **by** (*auto dest*: *splitAt-ram*)
  from *d ls2 ls1* **have** *fst* (*splitAt* (*ls*!*i*) *ls*) = *take i ls* ∧ *snd* (*splitAt* (*ls*!*i*) *ls*) = *drop* (*Suc i*) *ls* **by** (*rule dist-at*)
   then show *?thesis* **by** *auto*
qed

lemma *fst-splitAt-upt*:
 j <= *i* ⟹ *i* < *k* ⟹ *fst*(*splitAt i* [*j*..<*k*]) = [*j*..<*i*]
using *splitAt-take*[**where** *ls* = [*j*..<*k*] **and** *i*=*i*−*j*]
apply (*simp del*:*splitAt-take*) **apply** *arith*
done

lemma *snd-splitAt-upt*:
 j <= *i* ⟹ *i* < *k* ⟹ *snd*(*splitAt i* [*j*..<*k*]) = [*i*+*1*..<*k*]
using *splitAt-drop*[**where** *ls* = [*j*..<*k*] **and** *i*=*i*−*j*]
apply (*simp del*:*splitAt-drop*) **apply** *arith*
done

lemma *local-help1*: ⋀ *a vs*. *vs* = *c* @ *r* # *d* ⟹ *vs* = *a* @ *r* # *b* ⟹ *r* ∉ *set a* ⟹ *r* ∉ *set b* ⟹ *a* = *c*
proof (*induct c*)
  case *Nil*
  then have *ra*: *r* ∉ *set a* **and** *vs1*: *vs* = *a* @ *r* # *b* **and** *vs2*: *vs* = [] @ *r* # *d*
    by *auto*
  from *vs1 vs2* **have** *a* @ *r* # *b* = *r* # *d* **by** *auto*
  then have *hd* (*a* @ *r* # *b*) = *r* **by** *auto*
  then have *a* ≠ [] ⟹ *hd a* = *r* **by** *auto*
  then have *a* ≠ [] ⟹ *r* ∈ *set a* **by** (*induct a*) *auto*
  with *ra* **have** *a*: *a* = [] **by** *auto*
  then show *?case* **by** *auto*
next
  case (*Cons x xs*) **then show** *?case* **by** (*induct a*) *auto*
qed

lemma *local-help*: *vs* = *a* @ *r* # *b* ⟹ *vs* = *c* @ *r* # *d* ⟹ *r* ∉ *set a* ⟹ *r* ∉ *set b* ⟹ *a* = *c* ∧ *b* = *d*
proof −
  assume *dist*: *r* ∉ *set a r* ∉ *set b* **and** *vs1*: *vs* = *a* @ *r* # *b* **and** *vs2*: *vs* = *c* @ *r* # *d*

then have *a* = *c* **apply** (*rule-tac local-help1*) **.**
with *dist vs1 vs2* **show** *?thesis* **by** *simp*
qed

lemma *local-help′*: *a* @ *r* # *b* = *c* @ *r* # *d* ⟹ *r* ∉ *set a* ⟹ *r* ∉ *set b* ⟹ *a* = *c* ∧ *b* = *d*
by (*rule local-help*) *auto*

lemma *splitAt-simp1*: *ram* ∉ *set a* ⟹ *ram* ∉ *set b* ⟹ *fst* (*splitAt ram* (*a* @ *ram* # *b*)) = *a*
proof −
assume *ramab*: *ram* ∉ *set a* *ram* ∉ *set b*
def *vs* ≡ *a* @ *ram* # *b*
then have *vs*: *vs* = *a* @ *ram* # *b* **by** *auto*
then have *ram* ∈ *set vs* **by** *auto*
then have *vs* = *fst* (*splitAt ram vs*) @ *ram* # *snd* (*splitAt 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 *splitAt-simp2*: *ram* ∉ *set b* ⟹ *fst* (*splitAt ram* (*ram* # *b*)) = []
apply (*subgoal-tac fst* (*splitAt ram* ([] @ *ram* # *b*)) = []) **apply** *force*
apply (*rule splitAt-simp1*) **by** *auto*

lemma *splitAt-simp3*: *ram* ∉ *set a* ⟹ *fst* (*splitAt ram* (*a* @ [*ram*])) = *a*
apply (*subgoal-tac fst* (*splitAt ram* (*a* @ *ram* # [])) = *a*) **apply** *force*
apply (*rule splitAt-simp1*) **by** *auto*

lemma *splitAt-simp4*: *ram* ∉ *set a* ⟹ *ram* ∉ *set b* ⟹ *snd* (*splitAt ram* (*a* @ *ram* # *b*)) = *b*
proof −
assume *ramab*: *ram* ∉ *set a* *ram* ∉ *set b*
def *vs* ≡ *a* @ *ram* # *b*
then have *vs*: *vs* = *a* @ *ram* # *b* **by** *auto*
then have *ram* ∈ *set vs* **by** *auto*
then have *vs* = *fst* (*splitAt ram vs*) @ *ram* # *snd* (*splitAt ram vs*) **by** (*auto dest*: *splitAt-ram*)
with *vs ramab* **show** *?thesis* **by** (*simp add*: *splitAt-simp1*)
qed

lemma *help′′′-in*: ⋀ *xs*. *ram* ∈ *set b* ⟹ *fst* (*splitAtRec ram xs b*) = *xs* @ *fst* (*splitAtRec 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*: ⋀ *xs*. *ram* ∉ *set b* ⟹ *fst* (*splitAtRec ram xs b*) = *xs* @ *fst* (*splitAtRec ram* [] *b*)
proof (*induct b*)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons b bs*)
then have *ram* ∉ *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* ≠ *a* ⟹ *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* ≠ *ram* ⟹ *fst* (*splitAt ram* (*a* # *b*)) = *a* # *fst* (*splitAt ram b*) **apply** (*rule splitAt-simpB*) **by** *auto*
lemma *splitAt-simpC*[*simp*]: *ram* ∉ *set a* ⟹ *fst* (*splitAt ram* (*a* @ *b*)) = *a* @ *fst* (*splitAt ram b*)
apply (*induct a*) **by** *auto*

lemma *help''''*: ⋀ *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* $\Longrightarrow$ *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* $\Longrightarrow$ *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* $\Longrightarrow$ *snd* (*splitAt ram* (*a* @ *b*)) = *snd* (*splitAt ram b*)
apply (*induct a*) **by** *auto*

lemma *splitAt-rotate-pair-conv*:
  !!*xs*. $[\![$ *distinct xs*; *x* $\in$ *set xs* $]\!]$
  $\Longrightarrow$ *snd* (*splitAt x* (*rotate n xs*)) @ *fst* (*splitAt x* (*rotate n xs*)) =
      snd (*splitAt x xs*) @ *fst* (*splitAt x xs*)
apply(*induct n*) **apply** *simp*
apply(*simp del*:*rotate-Suc2 add*:*rotate1-rotate-swap*)
apply(*clarsimp simp add*:*rotate1-def split*:*list.split*)
apply(*erule disjE*) **apply** *simp*
apply(*drule split-list*)
apply *clarsimp*
done

1.4 *between*

constdefs *between* :: *'a list* $\Rightarrow$ *'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a list*
between vs ram_1 ram_2 $\equiv$
  let (pre_1, $post_1$) = *splitAt* ram_1 *vs in*
  if ram_2 *mem* $post_1$
  then let (pre_2, $post_2$) = *splitAt* ram_2 $post_1$ *in* pre_2
  else let (pre_2, $post_2$) = *splitAt* ram_2 pre_1 *in* $post_1$ @ pre_2

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

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

```
lemma set-between-id:
 distinct xs ⟹ x ∈ set xs ⟹
  set(between xs x x) = set xs − {x}
apply(drule split-list)
apply (clarsimp simp:between-def split-def Un-commute)
done
```

```
lemma split-between:
 ⟦ distinct vs; r ∈ set vs; v ∈ set vs; u ∈ set(between vs r v) ⟧ ⟹
  between vs r v =
 (if r=u then [] else between vs r u @ [u]) @ between vs u v
apply(cases  r = v)
 apply(clarsimp)
 apply(frule split-list[of v])
 apply(clarsimp)
 apply(simp add:between-def split-def split:split-if-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:split-if-asm)
 apply simp
 apply(erule disjE)
  apply(frule split-list[of u])
  apply(clarsimp simp:between-def split-def split:split-if-asm)
 apply(frule split-list[of u])
 apply(clarsimp simp:between-def split-def split:split-if-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:split-if-asm)
apply simp
apply(frule split-list[of u])
apply(clarsimp simp:between-def split-def split:split-if-asm)
done
```

1.5 Tables

```
types ('a, 'b) table = ('a × 'b) list

constdefs isTable :: ('a ⇒ 'b) ⇒ 'a list ⇒ ('a, 'b) table ⇒ bool
  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


constdefs
removeKey  :: 'a ⇒ ('a × 'b) list ⇒ ('a × 'b) list
removeKey a ps ≡ [p ∈ ps. a ≠ fst p]

consts removeKeyList :: 'a list ⇒ ('a × 'b) list ⇒ ('a × 'b) list
primrec
  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) blast

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)
```

```
  have set (removeKey w (removeKeyList ws ps)) ⊆ set (removeKeyList ws ps)
    by (rule removeKey-subset)
  with Cons show ?case by (simp add: removeKey-def)
qed

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

lemma notin-removeKey: r ∉ fst ‘ set (removeKey r ps)
  by (induct ps) (auto simp add: removeKey-def)

lemma notin-removeKeyList1:
  ⋀a. a ∈ set rs ⟹ (a, b) ∉ set (removeKeyList rs ps)
proof (induct rs)
  case Nil then show ?case by simp
next
  case (Cons r rs) then show ?case
  proof cases
    assume a = r then show ?thesis
      by (auto simp add: split-beta Let-def notin-removeKey1 removeKey-subset)
  next
    assume a: a ≠ r
    with Cons have(a, b) ∉ set (removeKeyList rs ps) by simp
    moreover have set (removeKey r (removeKeyList rs ps))
        ⊆ set (removeKeyList rs ps)
       by (rule removeKey-subset)
    ultimately have (a, b) ∉ set (removeKey r (removeKeyList rs ps))
      by blast
    then show ?thesis by simp
  qed
qed

lemma notin-removeKeyList: ⋀r. r ∈ set rs ⟹ r ∉ fst ‘ set (removeKeyList rs
ps)
proof (induct rs)
  case Nil then show ?case by simp
next
  case (Cons r′ rs) then show ?case
    by (auto simp add: notin-removeKey1 split-paired-all removeKey-def)
qed

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*: *split-if-asm add*: *removeKey-def*)

end

2 Isomorphisms Between Plane Graphs

theory *PlaneGraphIso*
imports *Main*
begin

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

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

consts
pr-isomorphic :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** ≅ *60*)
isomorphic :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** ≃ *60*)

constdefs
Iso :: $('a * 'a)$ *set* ({≅})
$\{\cong\} \equiv \{(f_1, f_2).\ f_1 \cong f_2\}$

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:

types
'a Fgraph = *'a list set*
'a fgraph = *'a list list*

2.1 Equivalence of faces

Two faces are equivalent modulo rotation:

defs (**overloaded**) *congs-def*:
$F_1 \cong (F_2$::*'a list*) $\equiv \exists\, n.\ F_2 =$ *rotate n* F_1

lemma *congs-refl*[*iff*]: (*xs*::*'a list*) ≅ *xs*
apply(*simp add*:*congs-def*)
apply(*rule-tac x* = *0* **in** *exI*)

```
apply (simp)
done

lemma congs-sym: assumes A: (xs::'a list) ≅ ys shows ys ≅ xs
proof (simp add:congs-def)
  let ?l = length xs
  from A obtain n where ys: ys = rotate n xs by(fastsimp 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(fastsimp 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-def)
apply(rule conjI)
 apply simp
apply(rule conjI)
 apply(fastsimp intro:congs-sym)
apply(fastsimp intro:congs-trans)
done

lemma congs-distinct:
  F₁ ≅ F₂ ⟹ distinct F₂ = distinct F₁
by (auto simp: congs-def)

lemma congs-length:
  F₁ ≅ F₂ ⟹ length F₂ = length F₁
by (auto simp: congs-def)

lemma congs-pres-nodes: F₁ ≅ F₂ ⟹ set F₁ = set F₂
```

```
by(clarsimp simp:congs-def)

lemma congs-map:
 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 (fastsimp)
apply clarsimp
apply(fastsimp 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 add:diff-less)
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

```
constdefs
 is-Hom :: ('a ⇒ 'b) ⇒ 'a Fgraph ⇒ 'b Fgraph ⇒ bool
is-Hom φ Fs1 Fs2 ≡ (map φ ` Fs1)//{≅} = Fs2 //{≅}

 is-pr-Iso :: ('a ⇒ 'b) ⇒ 'a Fgraph ⇒ 'b Fgraph ⇒ bool
is-pr-Iso φ Fs1 Fs2 ≡ is-Hom φ Fs1 Fs2 ∧ inj-on φ (⋃F ∈ Fs1. set F)

 is-hom :: ('a ⇒ 'b) ⇒ 'a fgraph ⇒ 'b fgraph ⇒ bool
is-hom φ Fs1 Fs2 ≡ is-Hom φ (set Fs1) (set Fs2)

 is-pr-iso :: ('a ⇒ 'b) ⇒ 'a fgraph ⇒ 'b fgraph ⇒ bool
```

is-pr-iso φ Fs_1 Fs_2 $\equiv$ *is-pr-Iso* φ (*set* Fs_1) (*set* Fs_2)

Homomorphisms preserve the set of nodes.

lemma *UN-subset-iff*: $((\bigcup i{\in}I.\ f\ i) \subseteq B) = (\forall\, i{\in}I.\ f\ i \subseteq B)$
by *blast*

declare *Image-Collect-split*[*simp del*]

lemma *Hom-pres-face-nodes*:
is-Hom φ Fs_1 Fs_2 $\Longrightarrow (\bigcup F{\in}Fs_1.\ \{\varphi\ \text{‘}\ (set\ F)\}) = (\bigcup F{\in}Fs_2.\ \{set\ F\})$
apply(*clarsimp simp*:*is-Hom-def quotient-def*)
apply *auto*
apply(*subgoal-tac EX* F' : Fs_2. $\{\cong\}$ “ $\{map\ \varphi\ F\} = \{\cong\}$ “ $\{F'\}$)
prefer *2* **apply** *blast*
apply (*fastsimp simp*: *eq-equiv-class-iff*[*OF equiv-EqF*] *dest*!:*congs-pres-nodes*)
apply(*subgoal-tac EX* F' : Fs_1. $\{\cong\}$ “ $\{map\ \varphi\ F'\} = \{\cong\}$ “ $\{F\}$)
apply (*fastsimp simp*: *eq-equiv-class-iff*[*OF equiv-EqF*] *dest*!:*congs-pres-nodes*)
apply (*erule equalityE*)
apply(*fastsimp simp*:*UN-subset-iff*)
done

lemma *Hom-pres-nodes*:
is-Hom φ Fs_1 Fs_2 $\Longrightarrow \varphi\ \text{‘}\ (\bigcup F{\in}Fs_1.\ set\ F) = (\bigcup F{\in}Fs_2.\ set\ F)$
apply(*drule Hom-pres-face-nodes*)
apply(*rule equalityI*)
apply *blast*
apply(*clarsimp*)
apply(*subgoal-tac set F* : $(\bigcup F{\in}Fs_2.\ \{set\ F\})$)
prefer *2* **apply** *blast*
apply(*subgoal-tac set F* : $(\bigcup F{\in}Fs_1.\ \{\varphi\ \text{‘}\ set\ F\})$)
prefer *2* **apply** *blast*
apply(*subgoal-tac EX* $F'{:}Fs_1.\ \varphi\ \text{‘}\ set\ F' = set\ F$)
prefer *2* **apply** *blast*
apply *blast*
done

Therefore isomorphisms preserve cardinality of node set.

lemma *pr-Iso-same-no-nodes*:
$[\![$ *is-pr-Iso* φ Fs_1 Fs_2; *finite* Fs_1 $]\!]$
$\Longrightarrow card(\bigcup F{\in}Fs_1.\ set\ F) = card(\bigcup F{\in}Fs_2.\ set\ F)$
by(*clarsimp simp add*: *is-pr-Iso-def Hom-pres-nodes*[*symmetric*] *card-image*)

lemma *pr-iso-same-no-nodes*:
is-pr-iso φ Fs_1 Fs_2 $\Longrightarrow card(\bigcup F{\in}set\ Fs_1.\ set\ F) = card(\bigcup F{\in}set\ Fs_2.\ set\ F)$
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* Fs_1 **and** *dist2*: *distinct* Fs_2

```
  and inj1: inj-on (%xs.{xs}//{≅}) (set Fs1)
  and inj2: inj-on (%xs.{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-Hom-def inj-on-def)
  have inj1': inj-on (%xs. {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 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-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-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:
 ⟦ is-Hom φ Fs1 Fs2; ∀ F∈Fs1. distinct F; ∀ F∈Fs2. distinct F ⟧
  ⟹ ∀ F∈Fs1. distinct(map φ F)
apply(clarsimp simp add:is-Hom-def)
apply(subgoal-tac ∃ F' ∈ Fs2. (map φ F, F') : {≅})
 apply(fastsimp 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
```

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

```
lemma is-pr-Iso-rec:
 ⟦ inj-on (%xs. {xs}//{≅}) Fs1; inj-on (%xs. {xs}//{≅}) Fs2; F1 ∈ Fs1 ⟧ ⟹
 is-pr-Iso φ Fs1 Fs2 =
```

```
 (∃ F_2 ∈ Fs_2. length F_1 = length F_2 ∧ is-pr-Iso φ (Fs_1 − {F_1}) (Fs_2 − {F_2})
    ∧ (∃ n. map φ F_1 = rotate n F_2)
    ∧ inj-on φ (⋃ F∈Fs_1. set F))
apply(drule mk-disjoint-insert[of F_1])
apply clarify
apply(rename-tac Fs_1')
apply(rule iffI)

apply (clarsimp simp add:is-pr-Iso-def)
apply(clarsimp simp:is-Hom-def quotient-diff1)
apply(drule sym)
apply(clarsimp)
apply(subgoal-tac {≅} `` {map φ F_1} : Fs_2 // {≅})
 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 {≅} `` {F_2} = {≅} `` {map φ F_1})
  prefer 2
  apply(rule equiv-class-eq[OF equiv-EqF])
  apply(fastsimp intro: congs-sym)
 apply(subgoal-tac {F_2}//{≅} = {map φ F_1}//{≅})
  prefer 2 apply(simp add:singleton-quotient)
 apply(subgoal-tac ∀ F∈Fs_1'. ¬ (map φ F) ≅ (map φ F_1))
  apply(fastsimp simp:Iso-def quotient-def Image-Collect-split
                dest!: eq-equiv-class[OF - equiv-EqF])
 apply clarify
 apply(subgoal-tac inj-on φ (set F ∪ set F_1))
  prefer 2
  apply(erule subset-inj-on)
  apply(blast)
 apply(clarsimp simp add:congs-map)
 apply(subgoal-tac {≅} `` {F_1} = {≅} `` {F})
  apply(simp add:singleton-quotient)
 apply(rule equiv-class-eq[OF equiv-EqF])
 apply(blast intro:congs-sym)
apply(subgoal-tac F_2 ≅ (map φ F_1))
 apply (simp add:congs-def inj-on-Un)
apply(clarsimp intro!:congs-sym)

apply(clarsimp simp add: is-pr-Iso-def is-Hom-def quotient-diff1)
apply (simp add:singleton-quotient)
apply(subgoal-tac F_2 ≅ (map φ F_1))
```

prefer *2* **apply**(*fastsimp simp add*:*congs-def*)
apply(*subgoal-tac* $\{\cong\}``\{map\ \varphi\ F_1\} = \{\cong\}``\{F_2\}$)
prefer *2*
apply(*rule equiv-class-eq*[*OF equiv-EqF*])
apply(*fastsimp intro*:*congs-sym*)
apply(*subgoal-tac* $\{\cong\}``\{F_2\} \in Fs_2\ //\ \{\cong\}$)
prefer *2* **apply**(*erule quotientI*)
apply (*simp add*:*insert-absorb quotient-def*)
done

lemma *is-iso-Cons*:
$[\![$ *distinct* $(F_1\#Fs_1')$; *distinct* Fs_2;
inj-on $(\%xs.\{xs\}//\{\cong\})$ $(set(F_1\#Fs_1'))$; *inj-on* $(\%xs.\{xs\}//\{\cong\})$ $(set\ Fs_2)$ $]\!]$
$\Longrightarrow$
is-pr-iso φ $(F_1\#Fs_1')$ Fs_2 =
$(\exists\, F_2 \in set\ Fs_2.\ length\ F_1 = length\ F_2 \wedge$ *is-pr-iso* $\varphi\ Fs_1'$ $(remove1\ F_2\ Fs_2)$
$\wedge\ (\exists\, n.\ map\ \varphi\ F_1 = rotate\ n\ F_2)$
$\wedge$ *inj-on* φ $(set\ F_1 \cup (\bigcup F{\in}set\ Fs_1'.\ set\ F)))$
apply(*simp add*:*is-pr-iso-def*)
apply(*subst is-pr-Iso-rec*[**where** $?F_1.0 = F_1$])
apply(*simp-all*)
done

2.3 Isomorphism tests

lemma *map-upd-submap*:
$x \notin dom\ m \Longrightarrow (m(x \mapsto y) \subseteq_m m') = (m'\ x = Some\ y \wedge m \subseteq_m m')$
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* $]\!] \Longrightarrow$
$(map\text{-}of\ (zip\ xs\ ys) \subseteq_m Some \circ f) = (map\ f\ xs = ys)$
apply(*induct rule*: *list-induct2*)
apply(*simp*)
apply (*clarsimp simp*: *map-upd-submap simp del*:*o-apply fun-upd-apply*)
apply *simp*
done

consts
pr-iso-test0 :: $('a \sim\!\!=\!> 'b) \Rightarrow 'a\ fgraph \Rightarrow 'b\ fgraph \Rightarrow bool$

primrec
pr-iso-test0 m [] $Fs_2 = (Fs_2 = [])$

```
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: ⟦ f ⊆m Some o h; g ⊆m Some o h ⟧ ⟹ f ⊆m f++g
by (fastsimp simp add: map-le-def map-add-def dom-def split:option.splits)

lemma inj-on-map-addI1:
 ⟦ inj-on m A; m ⊆m m++m′; A ⊆ dom m ⟧ ⟹ 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 fastsimp
apply auto
 apply fastsimp
apply(subgoal-tac m x = Some a)
 prefer 2 apply (fastsimp)
apply(subgoal-tac m y = Some a)
 prefer 2 apply (fastsimp)
apply(subgoal-tac m x = m y)
 prefer 2 apply simp
apply (blast)
done

lemma map-image-eq: ⟦ A ⊆ dom m; m ⊆m m′ ⟧ ⟹ m ‘ A = m′ ‘ A
by(force simp:map-le-def dom-def split:option.splits)

lemma inj-on-map-add-Un:
 ⟦ inj-on m (dom m); inj-on m′ (dom m′); m ⊆m Some o f; m′ ⊆m Some o f;
    inj-on f (dom m′ ∪ dom m); A = dom m′; B = dom m ⟧
  ⟹ inj-on (m ++ m′) (A ∪ B)
apply(simp add:inj-on-Un)
apply(rule conjI)
 apply(fastsimp 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(fastsimp)
apply(insert map-image-eq[of dom m′ − dom m m++m′ Some o f])
apply(subgoal-tac dom m − dom m′ ⊆ dom(m++m′))
 prefer 2 apply(fastsimp)
apply(insert map-image-eq[of dom m − dom m′ m++m′ Some o f])
apply (clarsimp simp add:image-compose)
apply blast
done
```

```
lemma map-of-zip-eq-SomeD: length xs = length ys ⟹
  map-of (zip xs ys) x = Some y ⟹ y ∈ set ys
apply(induct rule:list-induct2)
 apply simp
apply (auto split:if-splits)
done

lemma inj-on-map-of-zip:
  ⟦ length xs = length ys; distinct ys ⟧
   ⟹ inj-on (map-of (zip xs ys)) (set xs)
apply(induct rule:list-induct2)
 apply simp
apply(clarsimp simp add:image-map-upd)
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: ⋀m Fs2.
 ⟦ ∀F∈set Fs1. distinct F; ∀F∈set Fs2. distinct F;
   distinct Fs1; inj-on (%xs.{xs}//{≅}) (set Fs1);
   distinct Fs2; inj-on (%xs.{xs}//{≅}) (set Fs2); inj-on m (dom m) ⟧ ⟹
       pr-iso-test0 m Fs1 Fs2 =
       (∃φ. is-pr-iso φ Fs1 Fs2 ∧ m ⊆m Some o φ ∧
            inj-on φ (dom m ∪ (⋃F∈set Fs1. 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-Hom-def)
  apply(rule-tac x = the o m in exI)
  apply (fastsimp simp: map-le-def)
 apply (clarsimp simp:is-pr-iso-def is-pr-Iso-def is-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 = φ in exI)
apply(rule conjI)
 apply(rule-tac x = F2 in bexI)
  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 = F2 in bexI)
 prefer 2 apply assumption
apply simp
apply(rule-tac x = n in exI)
apply(rule conjI)
apply clarsimp
apply(rule-tac x = φ in exI)
apply simp
apply(rule conjI)
 apply(fastsimp 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:
 ⟦ ∀F∈set Fs1. distinct F; ∀F∈set Fs2. distinct F;
   distinct Fs1; inj-on (%xs.{xs}//{≅}) (set Fs1);
   distinct Fs2; inj-on (%xs.{xs}//{≅}) (set Fs2) ⟧ ⟹
       pr-iso-test0 empty Fs1 Fs2 = (∃φ. is-pr-iso φ Fs1 Fs2)
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 [] to be able to restrict the search to $n <$ *length xs* (which would otherwise be vacuous).

consts
pr-iso-test1 :: (*'a* ~=> *'b*) ⇒ *'a fgraph* ⇒ *'b fgraph* ⇒ *bool*

primrec
pr-iso-test1 m [] Fs_2 = (Fs_2 = [])
pr-iso-test1 m (F_1#Fs_1) Fs_2 =
(∃ F_2 ∈ *set* Fs_2. *length* F_1 = *length* F_2 ∧
(∃ *n* < *length* F_2. *let* m' = *map-of*(*zip* F_1 (*rotate n* F_2)) *in*
if $m \subseteq_m m$ ++ m' ∧ *inj-on* (m++m') (*dom*(m++m'))
then pr-iso-test1 (m ++ m') Fs_1 (*remove1* F_2 Fs_2) *else False*))

lemma *test0-conv-test1*:
!!*m* Fs_2. [] ∉ *set* Fs_2 ⟹ *pr-iso-test1 m* Fs_1 Fs_2 = *pr-iso-test0 m* Fs_1 Fs_2
apply(*induct* Fs_1)
apply *simp*
apply *simp*
apply(*rule iffI*)
apply *blast*
apply (*clarsimp simp*:*Let-def*)
apply(*rule-tac x* = F_2 **in** *bexI*)
prefer *2* **apply** *assumption*
apply *simp*
apply(*subgoal-tac* F_2 ≠ [])
prefer *2* **apply** *blast*
apply(*rule-tac x* = *n mod length* F_2 **in** *exI*)
apply(*simp add*:*rotate-conv-mod*[*symmetric*])
done

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

corollary *pr-iso-test1-corr*:
⟦ ∀*F*∈*set* Fs_1. *distinct F*; ∀*F*∈*set* Fs_2. *distinct F*; [] ∉ *set* Fs_2;
distinct Fs_1; *inj-on* (%*xs*.{*xs*}//{≅}) (*set* Fs_1);
distinct Fs_2; *inj-on* (%*xs*.{*xs*}//{≅}) (*set* Fs_2) ⟧ ⟹
pr-iso-test1 empty Fs_1 Fs_2 = (∃φ. *is-pr-iso* φ Fs_1 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.

constdefs
oneone :: (*'a* ∗ *'b*)*list* ⇒ *bool*
oneone xys ≡ *distinct*(*map fst xys*) ∧ *distinct*(*map snd xys*)
declare *oneone-def*[*simp*]

types

$('a,'b)\mathit{tester} = ('a * 'b)\mathit{list} \Rightarrow ('a * 'b)\mathit{list} \Rightarrow \mathit{bool}$
$('a,'b)\mathit{merger} = ('a * 'b)\mathit{list} \Rightarrow ('a * 'b)\mathit{list} \Rightarrow ('a * 'b)\mathit{list}$

consts
$\mathit{pr\text{-}iso\text{-}test2} :: ('a,'b)\mathit{tester} \Rightarrow ('a,'b)\mathit{merger} \Rightarrow$
$\quad ('a * 'b)\mathit{list} \Rightarrow {'a}\ \mathit{fgraph} \Rightarrow {'b}\ \mathit{fgraph} \Rightarrow \mathit{bool}$

primrec
$\mathit{pr\text{-}iso\text{-}test2}\ \mathit{tst}\ \mathit{mrg}\ I\ []\ \mathit{Fs}_2 = (\mathit{Fs}_2 = [])$
$\mathit{pr\text{-}iso\text{-}test2}\ \mathit{tst}\ \mathit{mrg}\ I\ (F_1\#\mathit{Fs}_1)\ \mathit{Fs}_2 =$
$\quad (\exists F_2 \in \mathit{set}\ \mathit{Fs}_2.\ \mathit{length}\ F_1 = \mathit{length}\ F_2 \wedge$
$\quad\quad (\exists n < \mathit{length}\ F_2.\ \mathit{let}\ I' = \mathit{zip}\ F_1\ (\mathit{rotate}\ n\ F_2)\ \mathit{in}$
$\quad\quad\quad \mathit{if}\ \ \mathit{tst}\ I'\ I$
$\quad\quad\quad \mathit{then}\ \mathit{pr\text{-}iso\text{-}test2}\ \mathit{tst}\ \mathit{mrg}\ (\mathit{mrg}\ I'\ I)\ \mathit{Fs}_1\ (\mathit{remove1}\ F_2\ \mathit{Fs}_2)\ \mathit{else}\ \mathit{False}))$

lemma *notin-range-map-of*:
$y \notin \mathit{snd}\ `\ \mathit{set}\ \mathit{xys} \Longrightarrow \mathit{Some}\ y \notin \mathit{range}(\mathit{map\text{-}of}\ \mathit{xys})$
apply(*induct xys*)
apply (*simp add*:*image-def*)
apply(*clarsimp split*:*if-splits*)
done

lemma *inj-on-map-upd*:
$[\![\ \mathit{inj\text{-}on}\ m\ (\mathit{dom}\ m);\ \mathit{Some}\ y \notin \mathit{range}\ m\ ]\!] \Longrightarrow \mathit{inj\text{-}on}\ (m(x \mapsto y))\ (\mathit{dom}\ m)$
apply(*simp add*:*inj-on-def dom-def image-def*)
apply (*blast intro*:*sym*)
done

lemma [*simp*]:
$\mathit{distinct}(\mathit{map}\ \mathit{snd}\ \mathit{xys}) \Longrightarrow \mathit{inj\text{-}on}\ (\mathit{map\text{-}of}\ \mathit{xys})\ (\mathit{dom}(\mathit{map\text{-}of}\ \mathit{xys}))$
apply(*induct xys*)
apply *simp*
apply (*simp add*: *notin-range-map-of inj-on-map-upd*)
apply(*clarsimp simp add*:*image-def*)
apply(*drule map-of-is-SomeD*)
apply *fastsimp*
done

lemma *lem*: $\mathit{Ball}\ (\mathit{set}\ \mathit{xs})\ P \Longrightarrow \mathit{Ball}\ (\mathit{set}\ (\mathit{remove1}\ x\ \mathit{xs}))\ P = \mathit{True}$
by(*induct xs*) *simp-all*

lemma *pr-iso-test2-conv-1*:
$!!I\ \mathit{Fs}_2.$
$[\![\ \forall I\ I'.\ \mathit{oneone}\ I \longrightarrow \mathit{oneone}\ I' \longrightarrow$
$\quad \mathit{tst}\ I'\ I = (\mathit{let}\ m = \mathit{map\text{-}of}\ I;\ m' = \mathit{map\text{-}of}\ I'$
$\quad\quad \mathit{in}\ m \subseteq_m m ++ m' \wedge \mathit{inj\text{-}on}\ (m{+}{+}m')\ (\mathit{dom}(m{+}{+}m')));$
$\forall I\ I'.\ \mathit{oneone}\ I \longrightarrow \mathit{oneone}\ I' \longrightarrow \mathit{tst}\ I'\ I$
$\quad \longrightarrow \mathit{map\text{-}of}(\mathit{mrg}\ I'\ I) = \mathit{map\text{-}of}\ I ++ \mathit{map\text{-}of}\ I';$

$\forall I\ I'$. *oneone I* & *oneone* $I' \longrightarrow$ *tst* $I'\ I \longrightarrow$ *oneone* (*mrg* $I'\ I$);
oneone I;
$\forall F \in$ *set* Fs_1. *distinct F*; $\forall F \in$ *set* Fs_2. *distinct F* $\rrbracket \Longrightarrow$
pr-iso-test2 tst mrg I $Fs_1\ Fs_2$ = *pr-iso-test1* (*map-of I*) $Fs_1\ Fs_2$
apply(*induct* Fs_1)
apply *simp*
apply(*simp add*:*Let-def lem inj-on-map-of-zip del*:*mod-less distinct-map cong*:*conj-cong*)
done

A simple implementation

constdefs
test :: ('*a*,'*b*)*tester*
test I I' ==
$\forall$ *xy* $\in$ *set I*. $\forall$ *xy'* $\in$ *set I'*. (*fst xy* = *fst xy'*) = (*snd xy* = *snd xy'*)

lemma *image-map-upd*:
$x \notin$ *dom m* $\Longrightarrow$ $m(x \mapsto y)$ ‘ A = m ‘ $(A - \{x\}) \cup$ (*if* $x \in A$ *then* $\{$*Some y*$\}$ *else* $\{\}$)
by(*auto simp*:*image-def dom-def*)

lemma *image-map-of-conv-Image*:
!!A. $\llbracket$ *distinct*(*map fst xys*) $\rrbracket$
$\Longrightarrow$ *map-of xys* ‘ A = *Some* ‘ (*set xys* ‘‘ A) $\cup$ (*if* $A \subseteq$ *fst* ‘ *set xys then* $\{\}$ *else* $\{$*None*$\}$)
apply (*induct xys*)
apply (*simp add*:*image-def Image-def*)
apply *blast*
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*)+, *fastsimp*)
apply *fastsimp*
apply(*clarify*)
apply((*rule conjI*, *clarify*)+, *fastsimp*)
apply *fastsimp*
apply *fastsimp*
apply *fastsimp*
done

lemma [*simp*]: *m*++*m'* ‘ (*dom m'* − *A*) = *m'* ‘ (*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** *bexI*)
prefer *2* **apply** *blast*

```
apply simp
done

declare Diff-subset [iff]

lemma test-correct:
 ⟦ oneone I; oneone I' ⟧ ⟹
       test 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: test-def Let-def map-le-iff-map-add-commute)
apply(rule iffI)
 apply(rule context-conjI)
  apply(rule ext)
  apply (fastsimp 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 fastsimp
apply clarsimp
apply(rename-tac a b aa ba)
apply(rule iffI)
 apply (clarsimp simp:expand-fun-eq)
 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 test-corr:
∀ I I'. oneone I ⟶ oneone I' ⟶
        test 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: test-correct)

constdefs
  merge :: ('a,'b)merger
 merge I' I ≡ [xy : I'. fst xy ∉ fst ` set I] @ I
```

lemma *help1*:
distinct(*map fst xys*) $\Longrightarrow$ *map-of* (*filter P xys*) =
 map-of xys |‘ $\{x.\ \exists\, y.\ (x,y) \in$ *set xys* $\wedge\ P(x,y)\}$
apply(*induct xys*)
 apply *simp*
apply(*rule ext*)
apply (*simp add*:*restrict-map-def*)
apply *force*
done

lemma *merge-correct*:
  $\forall\, I\ I'.$ *oneone* $I \longrightarrow$ *oneone* $I' \longrightarrow$ *test* $I'\ I$
  $\longrightarrow$ *map-of*(*merge* $I'\ I$) = *map-of* I ++ *map-of* I'
apply(*simp add*:*test-def merge-def help1 expand-fun-eq map-add-def restrict-map-def*
split:*option.split*)
apply *fastsimp*
done

lemma *merge-inv*:
  $\forall\, I\ I'.$ *oneone* $I \wedge$ *oneone* $I' \longrightarrow$ *test* $I'\ I \longrightarrow$ *oneone* (*merge* $I'\ I$)
apply(*auto simp add*:*merge-def distinct-map test-def*)
apply(*blast intro*:*subset-inj-on*)+
done

corollary *pr-iso-test2-corr*:
 $[\![\ \forall F{\in}$*set* $Fs_1.$ *distinct* F; $\forall F{\in}$*set* $Fs_2.$ *distinct* F; [] $\notin$ *set* Fs_2;
   distinct Fs_1; *inj-on* (%*xs*.{*xs*}//{$\cong$}) (*set* Fs_1);
   distinct Fs_2; *inj-on* (%*xs*.{*xs*}//{$\cong$}) (*set* Fs_2) $]\!] \Longrightarrow$
     pr-iso-test2 test merge [] $Fs_1\ Fs_2$ = ($\exists\varphi.$ *is-pr-iso* $\varphi\ Fs_1\ Fs_2$)
by(*simp add*: *pr-iso-test2-conv-1*[*OF test-corr merge-correct merge-inv*]
       pr-iso-test1-corr)

The final stage: implementing test and merge as recursive functions.

constdefs
 test2 :: ($'a,'b$)*tester*
test2 $I\ I'$ == *list-all* (%(x,y). *list-all* (%(x',y'). $(x{=}x') = (y{=}y')$) I') I

lemma *test2-conv-test*: *test2* $I\ I'$ = *test* $I\ I'$
by (*simp add*:*test-def test2-def list-all-iff split-def*)

consts
  merge2 :: ($'a,'b$)*merger*
primrec
merge2 [] $I = I$
merge2 (*xy*#*xys*) I = (*let* (x,y) = *xy in*
  if list-all (%(x',y'). $x \neq x'$) I *then xy* # *merge2 xys* I
  else merge2 xys I)

lemma *merge2-conv-merge*: *merge2* $I'\ I$ = *merge* $I'\ I$

apply(*induct I'*)
 apply(*simp add*:*merge-def*)
apply(*force simp add*:*Let-def list-all-iff merge-def*)
done

consts
  pr-iso-test3 :: ($'a * 'b$)*list* ⇒ $'a$ *fgraph* ⇒ $'b$ *fgraph* ⇒ *bool*

primrec
pr-iso-test3 I [] Fs_2 = (Fs_2 = [])
pr-iso-test3 I (F_1#Fs_1) Fs_2 =
   list-ex (%F_2. *length* F_1 = *length* F_2 ∧
      list-ex (%*n*. *let I'* = *zip* F_1 (*rotate n* F_2) *in*
          if test2 I' I
          then pr-iso-test3 (*merge2 I' I*) Fs_1 (*remove1* F_2 Fs_2) *else False*)
        (*upt 0* (*length* F_2))) Fs_2

lemma *pr-iso-test3-conv-2*:
  !!*I* Fs_2. *pr-iso-test3 I* Fs_1 Fs_2 = *pr-iso-test2 test merge I* Fs_1 Fs_2
apply(*induct* Fs_1)
 apply *simp*
apply(*simp add*:*test2-conv-test merge2-conv-merge list-ex-iff*)
done

corollary *pr-iso-test3-corr*:
 ⟦ ∀*F*∈*set* Fs_1. *distinct F*; ∀*F*∈*set* Fs_2. *distinct F*; [] ∉ *set* Fs_2;
   distinct Fs_1; *inj-on* (%*xs*.{*xs*}//{≅}) (*set* Fs_1);
   distinct Fs_2; *inj-on* (%*xs*.{*xs*}//{≅}) (*set* Fs_2) ⟧ ⟹
       pr-iso-test3 [] Fs_1 Fs_2 = (∃ φ. *is-pr-iso* φ Fs_1 Fs_2)
by(*simp add*: *pr-iso-test3-conv-2 pr-iso-test2-corr*)

A final optimization.

constdefs
 pr-iso-test :: (*nat* ∗ $'a$ *fgraph*) ⇒ (*nat* ∗ $'b$ *fgraph*) ⇒ *bool*
pr-iso-test ≡ λ(n_1,Fs_1) (n_2,Fs_2). n_1 = n_2 ∧ *length* Fs_1 = *length* Fs_2
                                ∧ *pr-iso-test3* [] Fs_1 Fs_2

corollary *pr-iso-test-correct*:
 ⟦ ∀*F*∈*set* Fs_1. *distinct F*; ∀*F*∈*set* Fs_2. *distinct F*; [] ∉ *set* Fs_2;
   distinct Fs_1; *inj-on* (%*xs*.{*xs*}//{≅}) (*set* Fs_1);
   distinct Fs_2; *inj-on* (%*xs*.{*xs*}//{≅}) (*set* Fs_2);
   n_1 = *card*(⋃*F*∈*set* Fs_1. *set F*); n_2 = *card*(⋃*F*∈*set* Fs_2. *set F*) ⟧ ⟹
  pr-iso-test (n_1,Fs_1) (n_2,Fs_2) = (∃ φ. *is-pr-iso* φ Fs_1 Fs_2)
apply(*simp add*:*pr-iso-test-def pr-iso-test3-corr*)
apply(*blast intro*:*pr-iso-same-no-faces pr-iso-same-no-nodes*)
done

2.3.2 ‘Improper’ Isomorphisms

```
constdefs
 is-Iso :: ('a ⇒ 'b) ⇒ 'a Fgraph ⇒ 'b Fgraph ⇒ bool
is-Iso φ Fs₁ Fs₂ ≡ is-pr-Iso φ Fs₁ Fs₂ ∨ is-pr-Iso φ Fs₁ (rev ‘ Fs₂)

 is-iso :: ('a ⇒ 'b) ⇒ 'a fgraph ⇒ 'b fgraph ⇒ bool
is-iso φ Fs₁ Fs₂ ≡ is-Iso φ (set Fs₁) (set Fs₂)

defs (overloaded) iso-fgraph-def:
g₁ ≃ g₂  ≡  ∃φ. is-iso φ g₁ g₂
```

```
constdefs
 iso-test :: (nat ∗ 'a fgraph) ⇒ (nat ∗ 'b fgraph) ⇒ bool
iso-test ≡ %g₁ g₂. pr-iso-test g₁ g₂ ∨ pr-iso-test g₁ (fst g₂,map rev (snd g₂))
```

```
lemma inj-on-image-iff: ⟦ ALL x:A. ALL y:A. (g(f x) = g(f y)) = (g x = g y);
  inj-on f A ⟧ ⟹ inj-on g (f ‘ A) = inj-on g A
apply(unfold inj-on-def)
apply blast
done
```

```
theorem iso-correct:
 ⟦ ∀F∈set Fs₁. distinct F; ∀F∈set Fs₂. distinct F; [] ∉ set Fs₂;
   distinct Fs₁; inj-on (%xs.{xs}//{≅}) (set Fs₁);
   distinct Fs₂; inj-on (%xs.{xs}//{≅}) (set Fs₂);
   n₁ = card(⋃F∈set Fs₁. set F); n₂ = card(⋃F∈set Fs₂. set F) ⟧ ⟹
  iso-test (n₁,Fs₁) (n₂,Fs₂) = (Fs₁ ≃ Fs₂)
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
       apply (simp add:image-def)
      apply simp
     apply simp
    apply (simp add:distinct-map)
   apply (simp add:inj-on-image-iff)
  apply simp
 apply simp
apply(simp add:is-iso-def is-Iso-def is-pr-iso-def)
apply blast
done
```

2.4 Elementhood and containment modulo

```
constdefs
 pr-iso-in :: 'a ⇒ 'a set ⇒ bool  (infix ∈≅ 60)
```

$x \in_{\cong} M \equiv \exists\, y \in M.\ x \cong y$

pr-iso-subseteq :: *'a set* ⇒ *'a set* ⇒ *bool* (**infix** $\subseteq_{\cong}$ *60*)
$M \subseteq_{\cong} N \equiv \forall\, x \in M.\ x \in_{\cong} N$

iso-in :: *'a* ⇒ *'a set* ⇒ *bool* (**infix** $\in_{\simeq}$ *60*)
$x \in_{\simeq} M \equiv \exists\, y \in M.\ x \simeq y$

iso-subseteq :: *'a set* ⇒ *'a set* ⇒ *bool* (**infix** $\subseteq_{\simeq}$ *60*)
$M \subseteq_{\simeq} N \equiv \forall\, x \in M.\ x \in_{\simeq} N$

end

3 More Rotation

theory *Rotation*
imports *ListAux PlaneGraphIso*
begin

constdefs
rotate-to :: *'a list* ⇒ *'a* ⇒ *'a list*
rotate-to vs v ≡ *v* # *snd* (*splitAt v vs*) @ *fst* (*splitAt v vs*)

rotate-min :: *nat list* ⇒ *nat list*
rotate-min vs ≡ *rotate-to vs* (*minList vs*)

lemma *cong-rotate-to*:
x ∈ *set xs* ⟹ *xs* ≅ *rotate-to xs x*
proof −
assume *x*: *x* ∈ *set xs*
hence *ls1*: *xs* = *fst* (*splitAt x xs*) @ *x* # *snd* (*splitAt x xs*) **by** (*auto dest*: *splitAt-ram*)
def *i* ≡ *length*(*fst*(*splitAt x xs*))
hence *i* < *length*((*fst*(*splitAt x xs*)) @ [*x*] @ *snd*(*splitAt x xs*)) **by** *auto*
with *ls1* **have** *i-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-len* **have** *length* (*take i xs*) = *i* **by** *auto arith*
with *i-def* **have** *len-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-simp: *x* = *xs*!*i* **and**
take-i: *fst*(*splitAt x xs*) = *take i xs* **and**
drop-i': *snd*(*splitAt x xs*) = *drop* (*Suc i*) *xs* **by** *auto*
from *i-len* **have** *ls3*: *xs* = *take i xs* @ *drop i xs* **by** *auto*
with *take-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-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-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 prems(2) by blast
   hence (rotate-min xs = rotate-min ys) = (xs ≅ ys)
     using xs prems(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 (*xsymbols*)
  @UNION1 :: pttrns => 'b set => 'b set ((3⋃(00-)/ -) 10)
  @INTER1 :: pttrns => 'b set => 'b set ((3⋂(00-)/ -) 10)
  @UNION :: pttrn => 'a set => 'b set => 'b set ((3⋃(00-∈-)/ -) 10)
  @INTER :: pttrn => 'a set => 'b set => 'b set ((3⋂(00-∈-)/ -) 10)

4.1 Notation

types *vertex* = *nat*

consts
 vertices :: *'a* ⇒ *vertex list*
 edges :: *'a* ⇒ (*vertex* × *vertex*) *set* ($\mathcal{E}$)

syntax -*Vertices* :: *'a* ⇒ *vertex set* ($\mathcal{V}$)
translations $\mathcal{V}$ *f* == *set*(*vertices f*)

4.2 Faces

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

datatype *facetype* = *Final* | *Nonfinal*

datatype *face* = *Face* (*vertex list*) *facetype*

consts *final* :: *'a* ⇒ *bool*
primrec
  final (*Face vs f*) = (*case f of Final* ⇒ *True* | *Nonfinal* ⇒ *False*)

consts *type* :: *'a* ⇒ *facetype*
primrec *type* (*Face vs f*) = *f*

primrec
  vertices (*Face vs f*) = *vs*

defs (**overloaded**) *cong-face-def*:
 $f_1 \cong (f_2$::*face*$) \equiv$ *vertices* $f_1 \cong$ *vertices* f_2

The following operation makes a face final.

constdefs *setFinal* :: *face* ⇒ *face*
  setFinal f ≡ *Face* (*vertices f*) *Final*

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

consts *nextElem* :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *'a*
primrec
nextElem [] *b x* = *b*
nextElem (*a*#*as*) *b x* =
(*if x*=*a then* (*case as of* [] ⇒ *b* | (*a'*#*as'*) ⇒ *a'*) *else nextElem as b x*)

constdefs *nextVertex* :: *face* ⇒ *vertex* ⇒ *vertex*
f · *v* ≡ *let vs* = *vertices f in nextElem vs* (*hd vs*) *v*

nextVertices is n-fold application of *nextvertex*.

constdefs *nextVertices* :: *face* ⇒ *nat* ⇒ *vertex* ⇒ *vertex*
$f^n \cdot v \equiv ((f\ \cdot)\hat{\ }n)\ v$

lemma *nextV2*: $f^2 \cdot v = f \cdot (f \cdot v)$
by (*simp add*: *nextVertices-def nat-number*)
edges (*f*::*face*) ≡ {(*a*, *f* · *a*)|*a*. *a* ∈ 𝒱 *f*}

defs (*vs*::*vertex list*)^{op} ≡ *rev vs*
primrec (*Face vs f*)^{op} = *Face* (*rev vs*) *f*

constdefs *prevVertex* :: *face* ⇒ *vertex* ⇒ *vertex*
$f^{-1} \cdot v$ ≡ (*let vs* = *vertices f in nextElem* (*rev vs*) (*last vs*) *v*)

syntax *triangle* :: *face* ⇒ *bool*
translations *triangle f* == |*vertices f*| = *3*

4.3 Graphs

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

consts *faces* :: *graph* ⇒ *face list*
primrec *faces* (*Graph fs n f h*) = *fs*

syntax *-Faces* :: *graph* ⇒ *face set* (ℱ)
translations ℱ *g* == *set*(*faces g*)

consts *countVertices* :: *graph* ⇒ *nat*
primrec *countVertices* (*Graph fs n f h*) = *n*

primrec *vertices* (*Graph fs n f h*) = [*0* ..< *n*]

lemma *vertices-graph*: *vertices g* = [*0* ..< *countVertices g*]
by (*induct g*) *simp*

lemma *in-vertices-graph*[*THEN eq-reflection*, *code unfold*]:

v ∈ *set* (*vertices g*) = (*v* < *countVertices g*)
by (*simp add*:*vertices-graph*)

lemma *len-vertices-graph*[*THEN eq-reflection*, *code unfold*]:
|*vertices g*| = *countVertices g*
by (*simp add*:*vertices-graph*)

consts *faceListAt* :: *graph* ⇒ *face list list*
primrec
faceListAt (*Graph fs n f h*) = *f*

constdefs *facesAt* :: *graph* ⇒ *vertex* ⇒ *face list*
facesAt g v ≡ *if v* ∈ *set*(*vertices g*) *then faceListAt g* ! *v else* []

consts *heights* :: *graph* ⇒ *nat list*
primrec
heights (*Graph fs n f h*) = *h*

constdefs *height* :: *graph* ⇒ *vertex* ⇒ *nat*
height g v ≡ *heights g* ! *v*

lemma *graph-split*:
g = *Graph* (*faces g*)
(*countVertices g*)
(*faceListAt g*)
(*heights g*)
by (*induct g*) *simp*

constdefs *graph* :: *nat* ⇒ *graph*
graph n ≡
(*let vs* = [*0* ..< *n*];
fs = [*Face vs Final*, *Face* (*rev vs*) *Nonfinal*]
in (*Graph fs n* (*replicate n fs*) (*replicate n 0*)))

4.4 Operations on graphs

final graph, final / nonfinal faces

constdefs *finals* :: *graph* ⇒ *face list*
finals g ≡ [*f* ∈ *faces g*. *final f*]

constdefs *nonFinals* :: *graph* ⇒ *face list*
nonFinals g ≡ [*f* ∈ *faces g*. ¬ *final f*]

constdefs *countNonFinals* :: *graph* ⇒ *nat*
countNonFinals g ≡ |*nonFinals g*|

defs *finalGraph-def*: *final g* ≡ (*nonFinals g* = [])

lemma *finalGraph-faces*[*simp*]: *final g* ⟹ *finals g* = *faces g*
by (*simp add*: *finalGraph-def finals-def nonFinals-def filter-compl1*)

lemma *finalGraph-face*: *final g* ⟹ *f* ∈ *set* (*faces g*) ⟹ *final f*
by (*simp only*: *finalGraph-faces*[*symmetric*]) (*simp add*: *finals-def*)

constdefs *finalVertex* :: *graph* ⇒ *vertex* ⇒ *bool*
finalVertex g v ≡ ∀ *f* ∈ *set*(*facesAt g v*). *final f*

lemma *finalVertex-final-face*[*dest*]:
finalVertex g v ⟹ *f* ∈ *set* (*facesAt g v*) ⟹ *final f*
by (*auto simp add*: *finalVertex-def*)

counting faces

constdefs *degree* :: *graph* ⇒ *vertex* ⇒ *nat*
degree g v ≡ |*facesAt g v*|

constdefs *tri* :: *graph* ⇒ *vertex* ⇒ *nat*
tri g v ≡ |[*f*: *facesAt g v*. *final f* ∧ |*vertices f*| = *3*]|

constdefs *quad* :: *graph* ⇒ *vertex* ⇒ *nat*
quad g v ≡ |[*f*: *facesAt g v*. *final f* ∧ |*vertices f*| = *4*]|

constdefs *except* :: *graph* ⇒ *vertex* ⇒ *nat*
except g v ≡ |[*f*: *facesAt g v*. *final f* ∧ *5* ≤ |*vertices f*|]|

constdefs *vertextype* :: *graph* ⇒ *vertex* ⇒ *nat* × *nat* × *nat*
vertextype g v ≡ (*tri g v*, *quad g v*, *except g v*)

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

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

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

constdefs *exceptionalVertex* :: *graph* ⇒ *vertex* ⇒ *bool*
exceptionalVertex g v ≡ *except g v* ≠ *0*

constdefs *noExceptionals* :: *graph* ⇒ *vertex set* ⇒ *bool*
noExceptionals g V ≡ (∀ *v* ∈ *V*. ¬ *exceptionalVertex g v*)

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

defs *edges-graph-def*:
edges (*g*::*graph*) ≡ $\bigcup_{f \,\in\, \mathcal{F}\, g}$ *edges f*

constdefs *neighbors* :: *graph* ⇒ *vertex* ⇒ *vertex list*

```
 neighbors g v ≡ [f•v. f ∈ facesAt g v]
```

4.5 Navigation in graphs

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

```
constdefs nextFace :: graph × vertex ⇒ face ⇒ face
  (g,v) • f ≡ (let fs = (facesAt g v) in
   (case fs of [] ⇒ f
          | g#gs ⇒ nextElem fs (hd fs) f))

constdefs prevFace :: graph × vertex ⇒ face ⇒ face
  (g,v)^-1 • f ≡ (let fs = (facesAt g v) in
   (case fs of [] ⇒ f
          | g#gs ⇒ nextElem (rev fs) (last fs) f))


constdefs directedLength :: face ⇒ vertex ⇒ vertex ⇒ nat
  directedLength f a b ≡
  if a = b then 0 else |(between (vertices f) a b)| + 1

end
```

5 Vector

```
theory Vector
imports Main EfficientNat
begin

datatype 'a vector = Vector 'a list
```

5.1 Tabulation

```
constdefs
  tabulate' :: nat × (nat ⇒ 'a) ⇒ 'a vector
  tabulate' p ≡ Vector (map (snd p) [0 ..< fst p])

  tabulate :: nat ⇒ (nat ⇒ 'a) ⇒ 'a vector
  tabulate n f ≡ tabulate' (n, f)
  tabulate2 :: nat ⇒ nat ⇒ (nat ⇒ nat ⇒ 'a) ⇒ 'a vector vector
  tabulate2 m n f ≡ tabulate m (λi .tabulate n (f i))
  tabulate3 :: nat ⇒ nat ⇒ nat ⇒
  (nat ⇒ nat ⇒ nat ⇒ 'a) ⇒ 'a vector vector vector
  tabulate3 l m n f ≡ tabulate l (λi. tabulate m (λj .tabulate n (λk. f i j k)))


syntax
```

$-tabulate :: 'a \Rightarrow pttrn \Rightarrow nat \Rightarrow 'a\ vector\ \ ((\llbracket\text{-}.\ \text{-} < \text{-}\rrbracket))$
$-tabulate2 :: 'a \Rightarrow pttrn \Rightarrow nat \Rightarrow$
$pttrn \Rightarrow nat \Rightarrow 'a\ vector$
$((\llbracket\text{-}.\ \text{-} < \text{-},\ \text{-} < \text{-}\rrbracket))$
$-tabulate3 :: 'a \Rightarrow pttrn \Rightarrow nat \Rightarrow$
$pttrn \Rightarrow nat \Rightarrow$
$pttrn \Rightarrow nat \Rightarrow 'a\ vector$
$((\llbracket\text{-}.\ \text{-} < \text{-},\ \text{-} < \text{-},\ \text{-} < \text{-}\ \rrbracket))$

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

5.2 Access

constdefs
$sub1 :: 'a\ vector \times nat \Rightarrow 'a$
$sub1\ p \equiv let\ (a,\ n) = p\ in\ case\ a\ of\ (Vector\ as) \Rightarrow as\ !\ n$
$sub :: 'a\ vector \Rightarrow nat \Rightarrow 'a$
$sub\ a\ n \equiv sub1\ (a,\ n)$

syntax
$-sub :: 'a\ vector \Rightarrow nat \Rightarrow 'a\ \ ((\text{-}\llbracket\text{-}\rrbracket)\ [1000]\ 999)$
$-sub2 :: 'a\ vector\ vector \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ \ ((\text{-}\llbracket\text{-},\text{-}\rrbracket)\ [1000]\ 999)$
$-sub3 :: 'a\ vector\ vector\ vector \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ ((\text{-}\llbracket\text{-},\text{-},\text{-}\rrbracket)\ [1000]\ 999)$

translations
$(a\llbracket n\rrbracket) == sub\ a\ n$
$(as\llbracket m,\ n\rrbracket) == sub\ (sub\ as\ m)\ n$
$(as\llbracket l,\ m,\ n\rrbracket) == sub\ (sub\ (sub\ as\ l)\ m)\ n$

types-code
vector (- *vector*)

consts-code
tabulate' (*Vector.tabulate*)
sub1 (*Vector.sub*)

examples: $\llbracket 0::'a.\ i < 5\rrbracket$, $\llbracket i.\ i < 5,\ j < 3\rrbracket$

lemma *sub-tabulate*: $0 \le i ==> i < u ==>$
$(tabulate\ u\ f)\llbracket i\rrbracket = f\ i$
by (*simp add*: *tabulate-def tabulate'-def sub-def sub1-def Let-def*)

lemma *sub-tabulate3*: $0 \le i ==> 0 \le j ==> 0 \le k ==>$
$i < l ==> j < m ==> k < n ==>$
$(tabulate3\ l\ m\ n\ f)\llbracket i,\ j,\ k\rrbracket = f\ i\ j\ k$
by (*simp add*: *tabulate3-def tabulate-def tabulate'-def*
sub-def sub1-def Let-def Vector.inject

split: *Vector.split*)

end

6 Enumerating Patches

theory *Enumerator*
imports *Graph Vector*
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 $0, \ldots, outer - 1$, with 0 and $outer - 1$ the endpoints of e.

Generate all lists

$$[a0, \ldots, a_{inner_1}]$$

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

Construct the vertices $v_0, \ldots, v_{inner-1}$ 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 $[a0, \ldots, a_{inner_1}]$, we run over all osibilites fro the finishe face along the edge e inside F.

constdefs *enumBase* :: *nat* ⇒ *nat list list*
enumBase nmax ≡ [[*i*]. *i* ∈ [*0* .. *nmax*]]

constdefs *enumAppend* :: *nat* ⇒ *nat list list* ⇒ *nat list list*
enumAppend nmax iss ≡ $\bigsqcup_{is \in iss}$ [*is* @ [*n*]. *n* ∈ [*last is* .. *nmax*]]

constdefs *enumerator* :: *nat* ⇒ *nat* ⇒ *nat list list*
enumerator inner outer ≡
let nmax = *outer* − *2*; *k* = *inner* − *3 in*
[[*0*] @ *is* @ [*outer* − *1*]. *is* ∈ ((*enumAppend nmax*) ˆ*k*) (*enumBase nmax*)]

enumTab :: *nat list list vector vector*
enumTab ≡ ⟦ *enumerator inner outer*. *inner* < *9*, *outer* < *9* ⟧

enum :: *nat* ⇒ *nat* ⇒ *nat list list*

```
enum inner outer ≡ if inner < 9 & outer < 9 then enumTab⟦inner,outer⟧
                   else enumerator inner outer
```

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

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

constdefs indexToVertexList :: face ⇒ vertex ⇒ nat list ⇒ vertex option list
 indexToVertexList f v is ≡ hideDups [f^k•v. k ∈ is]


end
```

7 Subdividing a Face

```
theory FaceDivision
imports Graph
begin

constdefs split-face :: face ⇒ vertex ⇒ vertex ⇒ vertex list ⇒ face × face
 split-face f ram_1 ram_2 newVs ≡ let vs = vertices f;
    f_1 = [ram_1] @ between vs ram_1 ram_2 @ [ram_2];
    f_2 = [ram_2] @ between vs ram_2 ram_1 @ [ram_1] in
    (Face (rev newVs @ f_1) Nonfinal,
    Face (f_2 @ newVs) Nonfinal)


constdefs replacefacesAt ::
    nat list ⇒ face ⇒ face list ⇒ face list list ⇒ face list list
 replacefacesAt ns f fs F ≡ mapAt ns (replace f fs) F


constdefs makeFaceFinalFaceList :: face ⇒ face list ⇒ face list
  makeFaceFinalFaceList f fs ≡ replace f [setFinal f] fs

constdefs makeFaceFinal :: face ⇒ graph ⇒ graph
 makeFaceFinal f g ≡
    Graph (makeFaceFinalFaceList f (faces g))
```

$(\mathit{countVertices}\ g)$
$[\mathit{makeFaceFinalFaceList}\ f\ \mathit{fs}.\ \mathit{fs} \in \mathit{faceListAt}\ g]$
$(\mathit{heights}\ g)$

constdefs $\mathit{heightsNewVertices} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat\ list}$
$\mathit{heightsNewVertices}\ h_1\ h_2\ n \equiv [\mathit{min}\ (h_1 + i + 1)\ (h_2 + n - i).\ i \in [0\ ..{<}\ n]]$

constdefs *splitFace*
$:: \mathit{graph} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{face} \Rightarrow \mathit{vertex\ list} \Rightarrow \mathit{face} \times \mathit{face} \times \mathit{graph}$
$\mathit{splitFace}\ g\ \mathit{ram}_1\ \mathit{ram}_2\ \mathit{oldF}\ \mathit{newVs} \equiv$
$\mathit{let}\ \mathit{fs} = \mathit{faces}\ g;$
$n = \mathit{countVertices}\ g;$
$\mathit{Fs} = \mathit{faceListAt}\ g;$
$h = \mathit{heights}\ g;$
$\mathit{vs}_1 = \mathit{between}\ (\mathit{vertices}\ \mathit{oldF})\ \mathit{ram}_1\ \mathit{ram}_2;$
$\mathit{vs}_2 = \mathit{between}\ (\mathit{vertices}\ \mathit{oldF})\ \mathit{ram}_2\ \mathit{ram}_1;$
$(f_1, f_2) = \mathit{split\text{-}face}\ \mathit{oldF}\ \mathit{ram}_1\ \mathit{ram}_2\ \mathit{newVs};$
$\mathit{Fs} = \mathit{replacefacesAt}\ \mathit{vs}_1\ \mathit{oldF}\ [f_1]\ \mathit{Fs};$
$\mathit{Fs} = \mathit{replacefacesAt}\ \mathit{vs}_2\ \mathit{oldF}\ [f_2]\ \mathit{Fs};$
$\mathit{Fs} = \mathit{replacefacesAt}\ [\mathit{ram}_1]\ \mathit{oldF}\ [f_2, f_1]\ \mathit{Fs};$
$\mathit{Fs} = \mathit{replacefacesAt}\ [\mathit{ram}_2]\ \mathit{oldF}\ [f_1, f_2]\ \mathit{Fs};$
$\mathit{Fs} = \mathit{Fs}\ @\ \mathit{replicate}\ |\mathit{newVs}|\ [f_1, f_2]\ \mathit{in}$
$(f_1, f_2, \mathit{Graph}\ ((\mathit{replace}\ \mathit{oldF}\ [f_2]\ \mathit{fs})@\ [f_1])$
$(n + |\mathit{newVs}|\ )$
Fs
$(h\ @\ \mathit{heightsNewVertices}\ (h!\mathit{ram}_1)(h!\mathit{ram}_2)\ |\mathit{newVs}|\ ))$

consts $\mathit{subdivFace}' :: \mathit{graph} \Rightarrow \mathit{face} \Rightarrow \mathit{vertex} \Rightarrow \mathit{nat} \Rightarrow \mathit{vertex\ option\ list} \Rightarrow \mathit{graph}$
primrec $\mathit{subdivFace}'\ g\ f\ u\ n\ [] = \mathit{makeFaceFinal}\ f\ g$
$\mathit{subdivFace}'\ g\ f\ u\ n\ (\mathit{vo}\#\mathit{vos}) =$
$(\mathit{case}\ \mathit{vo}\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{subdivFace}'\ g\ f\ u\ (\mathit{Suc}\ n)\ \mathit{vos}$
$|\ (\mathit{Some}\ v) \Rightarrow$
$\mathit{if}\ f{\bullet}u = v \wedge n = 0$
$\mathit{then}\ \mathit{subdivFace}'\ g\ f\ v\ 0\ \mathit{vos}$
$\mathit{else}\ \mathit{let}\ \mathit{ws} = [\mathit{countVertices}\ g\ \ ..{<}\ \mathit{countVertices}\ g + n];$
$(f_1, f_2, g') = \mathit{splitFace}\ g\ u\ v\ f\ \mathit{ws}\ \mathit{in}$
$\mathit{subdivFace}'\ g'\ f_2\ v\ 0\ \mathit{vos})$

constdefs $\mathit{subdivFace} :: \mathit{graph} \Rightarrow \mathit{face} \Rightarrow \mathit{vertex\ option\ list} \Rightarrow \mathit{graph}$
$\mathit{subdivFace}\ g\ f\ \mathit{vos} \equiv \mathit{subdivFace}'\ g\ f\ (\mathit{the}(\mathit{hd}\ \mathit{vos}))\ 0\ (\mathit{tl}\ \mathit{vos})$

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 $'a \Rightarrow 'a\ list$. Instead of defining the closure again it would have been simpler to take $\{(x,\ y).\ y \in set\ (f\ x)\}^*$.

```
consts RTranCl :: ('a ⇒ 'a list) ⇒ ('a ∗ 'a) set
syntax in-set :: 'a ⇒ ('a ⇒ 'a list) ⇒ 'a ⇒ bool
      (- [-]→ - [55,0,55] 50)
translations g [succs]→ g' => g' ∈ set (succs g)
syntax in-RTranCl :: 'a ⇒ ('a ⇒ 'a list) ⇒ 'a ⇒ bool
      (- [-]→∗ - [55,0,55] 50)
translations g [succs]→∗ g' == (g,g') ∈ RTranCl succs

inductive RTranCl succs
intros
  refl: g [succs]→∗ g
  succs: g [succs]→ g' =⇒ g' [succs]→∗ g'' =⇒ g [succs]→∗ g''

lemmas RTranCl1 = RTranCl.succs[OF - RTranCl.refl]

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 P g g' ∈ set(succs g) (g', g'') ∈ RTranCl succs
      and IH: P g' =⇒ P g''
    have P g' by (rule s)
    then show P g'' by (rule IH)
  qed
qed

constdefs
 invariant :: ('a ⇒ bool) ⇒ ('a ⇒ 'a list) ⇒ bool
invariant P succs ≡ ∀ g g'. g' ∈ set(succs g) −→ P g −→ P g'
```

```
lemma invariantE:
  invariant P succs  ⟹ g [succs]→ g′ ⟹ P g ⟹ P g′
by(simp add:invariant-def)

lemma inv-subset:
 invariant P f ⟹ (!!g. P g ⟹ set(f′ g) ⊆ set(f g)) ⟹ invariant P f′
by(auto simp:invariant-def)

lemma inv-RTranCl-subset:
assumes inv: invariant P succs and f: (!!g. P g ⟹ (g,f g) ∈ RTranCl succs)
shows invariant P (%g. [f g])
proof(clarsimp simp:invariant-def)
  fix g assume P g
  from f[OF ‹P g›] show P(f g)
    by induct (auto intro!: ‹P g› elim!: invariantE[OF inv])
qed

lemma inv-comp-map:
 invariant P succs ⟹ (!!g. P g ⟹ P(f g)) ⟹ invariant P (map f o succs)
by(auto simp:invariant-def)

lemma RTranCl-inv:
  invariant P succs ⟹ (g,g′) ∈ RTranCl succs ⟹ P g ⟹ P g′
by (erule RTranCl-induct)(auto simp:invariant-def)

lemma RTranCl-subset: (!!g. set(f g) ⊆ set(h g)) ⟹
  (s,g) : RTranCl f ⟹ (s,g) : RTranCl h
apply(erule RTranCl.induct)
apply(rule RTranCl.intros)
apply(blast intro: RTranCl.intros)
done

lemma RTranCl-subset2:
assumes a: (s,g) : RTranCl f
shows (!!g. (s,g) ∈ RTranCl f ⟹ set(f g) ⊆ set(h g)) ⟹ (s,g) : 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

lemma RTranCl-rev-succs:
 (g, g′) ∈ RTranCl succs ⟹ g′′ ∈ set (succs g′) ⟹ (g, g′′) ∈ RTranCl succs
proof (induct rule: RTranCl.induct)
  fix g assume g′′ ∈ set (succs g)
  moreover have (g′′, g′′) ∈ RTranCl succs by (rule RTranCl.refl)
  ultimately show (g, g′′) ∈ RTranCl succs by (rule RTranCl.succs)
next
```

```
  fix g h′ h′′ assume h′ ∈ set (succs g)
  moreover assume (h′, h′′) ∈ RTranCl succs and g′′ ∈ set (succs h′′)
   and IH: g′′ ∈ set (succs h′′) ⟹ (h′, g′′) ∈ RTranCl succs
  have (h′, g′′) ∈ RTranCl succs by (rule IH)
  ultimately show (g, g′′) ∈ RTranCl succs by (rule RTranCl.succs)
qed

lemma RTranCl-compose:
assumes (g,g′) ∈ RTranCl succs
shows (g′,g′′) ∈ RTranCl succs ⟹ (g,g′′) ∈ RTranCl succs
using prems(1)
proof (induct rule: RTranCl-induct)
  case refl show ?case by assumption
next
  case succs thus ?case by(blast intro:RTranCl.succs)
qed


lemma RTranCl-map-comp-subset:
assumes inv: invariant P succs
and f: (!!g. P g ⟹ (g,f g) ∈ RTranCl succs)
and a: (s,g) ∈ RTranCl (map f o succs)
shows P s ⟹ (s,g) ∈ RTranCl succs
using a
proof(induct rule: RTranCl-induct)
  case refl thus ?case by(fast intro: RTranCl.refl)
next
  case (succs g g′)
  then obtain g′′ where g′′ ∈ set(succs g)  g′ = f g′′ by auto
  moreover then have P g′′
   using RTranCl-inv[OF inv, OF succs(2)] succs(3) inv by(simp add:invariant-def)
 ultimately show ?case by(blast intro:RTranCl-compose RTranCl.succs f succs(2−3))
qed


end
```

9 Plane Graph Enumeration

```
theory Plane
imports Enumerator FaceDivision RTranCl
begin


constdefs
 maxGon :: nat ⇒ nat
maxGon p ≡ p+3
```

declare *maxGon-def* [*simp*]

constdefs $\mathit{duplicateEdge} :: \mathit{graph} \Rightarrow \mathit{face} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{bool}$
$\mathit{duplicateEdge}\ g\ f\ a\ b \equiv$
$2 \le \mathit{directedLength}\ f\ a\ b \wedge 2 \le \mathit{directedLength}\ f\ b\ a \wedge b\ \mathit{mem}\ (\mathit{neighbors}\ g\ a)$

consts $\mathit{containsUnacceptableEdgeSnd} ::$
$(\mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}) \Rightarrow \mathit{nat} \Rightarrow \mathit{nat\ list} \Rightarrow \mathit{bool}$
primrec $\mathit{containsUnacceptableEdgeSnd}\ N\ v\ [] = \mathit{False}$
$\mathit{containsUnacceptableEdgeSnd}\ N\ v\ (w\#ws) =$
$(\mathit{case}\ ws\ \mathit{of}\ [] \Rightarrow \mathit{False}$
$\mid (w'\#ws') \Rightarrow \mathit{if}\ v < w \wedge w < w' \wedge N\ w\ w'\ \mathit{then}\ \mathit{True}$
$\mathit{else}\ \mathit{containsUnacceptableEdgeSnd}\ N\ w\ ws)$

consts $\mathit{containsUnacceptableEdge} :: (\mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}) \Rightarrow \mathit{nat\ list} \Rightarrow \mathit{bool}$
primrec $\mathit{containsUnacceptableEdge}\ N\ [] = \mathit{False}$
$\mathit{containsUnacceptableEdge}\ N\ (v\#vs) =$
$(\mathit{case}\ vs\ \mathit{of}\ [] \Rightarrow \mathit{False}$
$\mid (w\#ws) \Rightarrow \mathit{if}\ v < w \wedge N\ v\ w\ \mathit{then}\ \mathit{True}$
$\mathit{else}\ \mathit{containsUnacceptableEdgeSnd}\ N\ v\ vs)$

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

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

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

constdefs $\mathit{next\text{-}plane0} :: \mathit{nat} \Rightarrow \mathit{graph} \Rightarrow \mathit{graph\ list}\ (\mathit{next'\text{-}plane0}_{\_})$
$\mathit{next\text{-}plane0}_{p}\ g \equiv$
$\mathit{if\ final}\ g\ \mathit{then}\ []$
$\mathit{else} \bigsqcup_{f \in \mathit{nonFinals}\ g} \bigsqcup_{v \in \mathit{vertices}\ f} \bigsqcup_{i \in [3..\mathit{maxGon}\ p]} \mathit{generatePolygon}\ i\ v\ f\ g$

constdefs *Seed* :: *nat* ⇒ *graph* (*Seed*_)
$\mathit{Seed}_p \equiv \mathit{graph}(\mathit{maxGon}\ p)$

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

constdefs
PlaneGraphs0 :: *graph set*
$\mathit{PlaneGraphs0} \equiv \bigcup p.\ \{g.\ \mathit{Seed}_p\ [\mathit{next\text{-}plane0}_p]\rightarrow\!*\ g \wedge \mathit{final}\ g\}$

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.

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

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

constdefs *next-plane* :: *nat* ⇒ *graph* ⇒ *graph list* (*next′-plane*_)
$\mathit{next\text{-}plane}_p\ g \equiv$
  let fs = *nonFinals g in*
  if fs = [] *then* []
  else let f = *minimalFace fs*; *v* = *minimalVertex g f in*
    $\bigsqcup_{i\in[3..\mathit{maxGon}\ p]} \mathit{generatePolygon}\ i\ v\ f\ g$

constdefs
PlaneGraphsP :: *nat* ⇒ *graph set* (*PlaneGraphs*_)
$\mathit{PlaneGraphs}_p \equiv \{g.\ \mathit{Seed}_p\ [\mathit{next\text{-}plane}_p]\rightarrow\!*\ g \wedge \mathit{final}\ g\}$

PlaneGraphs :: *graph set*
$\mathit{PlaneGraphs} \equiv \bigcup p.\ \mathit{PlaneGraphs}_p$

end

10 Properties of Graph Utilities

theory *GraphProps*

```
imports Graph
begin

MLfast-arith-neq-limit := 3

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


lemma eq-setFinal-iff[iff]: (f = setFinal f) = 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]: (setFinal f = f) = 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 ∉ set xs ⟹ nextElem (xs @ ys) d y = nextElem ys d y
by(induct xs) auto

lemma nextElem-cases:
nextElem xs d x = y ⟹
 x ∉ set xs ∧ y = d ∨
 xs ≠ [] ∧ x = last xs ∧ y = d ∧ x ∉ set(butlast xs) ∨
 (∃ us vs. xs = us @ [x,y] @ vs ∧ x ∉ 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 ∉ set(butlast xs) ⟶ nextElem xs x y = x
by(induct xs) auto

lemma nextElem-in: nextElem xs x y : 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 ∉ set as ⟹ nextElem as c a = c
by(erule nextElem-append[where ys = [], simplified])

lemma nextElem-last[simp]: assumes dist: distinct xs
shows nextElem xs c (last xs) = c
proof cases
  assume xs = [] thus ?thesis by simp
next
  let ?xs = butlast xs @ [last xs]
  assume xs: xs ≠ []
  with dist have distinct ?xs by simp
  hence notin: last xs ∉ set(butlast xs) by simp
  from xs have nextElem xs c (last xs) = nextElem ?xs c (last xs) by simp
  also from notin have ... = c by simp
  finally show ?thesis .
qed


lemma prevElem-nextElem:
assumes dist: distinct xs and xxs: x : set xs
shows nextElem (rev xs) (last xs) (nextElem xs (hd xs) x) = x
proof −
  def x' ≡ nextElem xs (hd xs) x
  hence nE: nextElem xs (hd xs) x = x' by simp
  have xs ≠ [] ∧ x = last xs ∧ x' = hd xs ∨ (∃ us vs. xs = us @ [x, x'] @ vs)
    (is ?A ∨ ?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 ⟧ ⟹
  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:
 !!i. ⟦distinct xs; i < length xs ⟧
   ⟹ 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 ≠ [] ⟹ f · v ∈ 𝒱 f
proof –
  assume f: vertices f ≠ []
  def 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(fastsimp simp:neq-Nil-conv)
    done
qed
```

```
lemma nextVertex-in-face[simp]:
  v ∈ set (vertices f) ⟹ f · v ∈ 𝒱 f
 by (auto intro: nextVertex-in-face')
```

```
lemma nextVertex-prevVertex[simp]:
 ⟦ distinct(vertices f); v ∈ 𝒱 f ⟧
 ⟹ f · (f^-1 · v) = v
by(simp add:prevVertex-def nextVertex-def nextElem-prevElem)
```

lemma *prevVertex-nextVertex*[*simp*]:
⟦ *distinct*(*vertices f*); *v* ∈ 𝒱 *f* ⟧
⟹ f^{-1} · (*f* · *v*) = *v*
by(*simp add*:*prevVertex-def nextVertex-def prevElem-nextElem*)

lemma *prevVertex-in-face*[*simp*]:
v ∈ 𝒱 *f* ⟹ f^{-1} · *v* ∈ 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*:
⟦ *distinct*(*vertices f*); *i* < |*vertices f*| ⟧ ⟹
f · (*vertices f* ! *i*) = *vertices f* ! ((*i*+*1*) *mod* |*vertices f*|)
apply(*cases vertices f* = []) **apply** *simp*
apply(*simp add*:*nextVertex-def nextElem-nth hd-conv-nth*)
done

10.3 ℰ

lemma *finite-edges*: *finite*(ℰ(*f*::*face*))
apply(*simp add*:*edges-face-def*)
apply(*rule finite-surj*[*of* 𝒱 *f* - %*v*. (*v*, *f* · *v*)])
apply *simp*
apply *blast*
done

lemma *edges-face-eq*:
((*a*,*b*) ∈ ℰ (*f*::*face*)) = ((*f* · *a* = *b*) ∧ *a* ∈ 𝒱 *f*)
by (*auto simp add*: *edges-face-def*)

lemma *edges-setFinal*[*simp*]: ℰ(*setFinal f*) = ℰ *f*
by(*induct f*)(*simp add*:*setFinal-def edges-face-def nextVertex-def*)

lemma *in-edges-in-vertices*:
(*x*,*y*) ∈ ℰ(*f*::*face*) ⟹ *x* ∈ 𝒱 *f* ∧ *y* ∈ 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*:

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

lemma *nextVertex-in-edges*: $v \in \mathcal{V}\ f \Longrightarrow (v,\ f \cdot v) \in \mathit{edges}\ f$
by(*auto simp*:*edges-face-def*)

lemma *prevVertex-in-edges*:
$[\![\mathit{distinct}(\mathit{vertices}\ f);\ v \in \mathcal{V}\ f]\!] \Longrightarrow (f^{-1} \cdot v,\ v) \in \mathit{edges}\ f$
by(*simp add*:*edges-face-eq*)

10.4 Triangles

lemma *vertices-triangle*:
$|\mathit{vertices}\ f| = 3 \Longrightarrow a \in \mathcal{V}\ f \Longrightarrow$
$\mathit{distinct}\ (\mathit{vertices}\ f) \Longrightarrow$
$\mathcal{V}\ f = \{a,\ f \cdot a,\ f \cdot (f \cdot a)\}$
proof −
assume $|\mathit{vertices}\ f| = 3$
then obtain *a1 a2 a3* **where** $\mathit{vertices}\ f = [a1,\ a2,\ a3]$
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*:
$|\mathit{vertices}\ f| = 3 \Longrightarrow \mathit{distinct}(\mathit{vertices}\ f) \Longrightarrow v \in \mathcal{V}\ f$
$\Longrightarrow f \cdot (f \cdot (f \cdot v)) = v$
apply(*subgoal-tac ALL* (*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*:
$|\mathit{vertices}\ f| = 3 \Longrightarrow a \in \mathcal{V}\ f \Longrightarrow$
$\mathit{distinct}\ (\mathit{vertices}\ f) \Longrightarrow$
$f \cdot (f \cdot a) = f^{-1} \cdot a$
proof −
assume $|\mathit{vertices}\ f| = 3$
then obtain *a1 a2 a3* **where** $\mathit{vertices}\ f = [a1,\ a2,\ a3]$
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*:
$|vertices\ f| = 4 \Longrightarrow a \in \mathcal{V}\ f \Longrightarrow$
distinct (*vertices f*) $\Longrightarrow$
$\mathcal{V}\ f = \{a, f \cdot a, f \cdot (f \cdot a), f \cdot (f \cdot (f \cdot a))\}$
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*:
$[\![\ |vertices\ f| = 4;\ distinct(vertices\ f);\ v \in \mathcal{V}\ f\ ]\!] \Longrightarrow$
$f \cdot (f \cdot (f \cdot (f \cdot v))) = v$
apply(*subgoal-tac ALL* (*i*::*nat*) < *4*.
(((((((*i*+*1*) *mod 4*)+*1*) *mod 4*)+*1*) *mod 4*)+*1*) *mod 4* = *i*)
apply(*clarsimp simp*:*in-set-conv-nth nextVertex-nth*)
apply(*presburger*)
done

lemma *quad-nextVertex-prevVertex*:
$|vertices\ f| = 4 \Longrightarrow a \in \mathcal{V}\ f \Longrightarrow$ *distinct* (*vertices f*) $\Longrightarrow$
$f \cdot (f \cdot (f \cdot a)) = f^{-1} \cdot a$
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 *C0*[*dest*]: $f \in set\ (facesAt\ g\ v) \Longrightarrow v \in \mathcal{V}\ g$
by (*simp add*: *facesAt-def split*: *split-if-asm*)

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*:
$[\![\ distinct(vertices\ f);\ v \in \mathcal{V}\ f;\ u \in \mathcal{V}\ f;\ u \neq v\ ]\!] \Longrightarrow f \cdot v \neq v$
apply(*frule split-list*[*of v*])
apply(*clarsimp simp*: *nextVertex-def neq-Nil-conv hd-append*
  split:*list.splits split-if-asm*)
done

lemma *distinct-no-loop1*:
$[\![\ distinct(vertices\ f);\ v \in \mathcal{V}\ f;\ |vertices\ f| > 1\ ]\!] \Longrightarrow f \cdot v \neq v$
apply(*subgoal-tac* $\exists\, u \in \mathcal{V}\ f.\ u \neq v$)
 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*]:
 $v \notin set\ us \Longrightarrow between\ (u\ \#\ us\ @\ v\ \#\ vs)\ u\ v = us$
by(*simp add*:*between-def split-def*)

lemma *between-back*:
$[\![\ v \notin set\ us;\ u \notin set\ vs;\ v \neq u\ ]\!] \Longrightarrow between\ (v\ \#\ vs\ @\ u\ \#\ us)\ u\ v = us$
by(*simp add*:*between-def split-def*)

lemma *next-between*:
$[\![distinct(vertices\ f);\ v \in \mathcal{V}\ f;\ u \in \mathcal{V}\ f;\ f \cdot v \neq u\ ]\!]$
 $\Longrightarrow f \cdot v \in set(between\ (vertices\ f)\ v\ u)$
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(fastsimp 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(fastsimp)
done
```

```
lemma between-next-empty:
 distinct(vertices f) ⟹ between (vertices f) v (f • 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-next:
 ⟦ distinct(vertices f); u ∈ 𝒱 f; v ∈ 𝒱 f; f • v ≠ u ⟧ ⟹
  between (vertices f) v u = f • v # between (vertices f) (f • v) u
using split-between[OF - - - next-between]
```

```
apply (simp add: between-next-empty split:split-if-asm)
apply(blast dest:distinct-no-loop2 intro:sym)
done


lemma unroll-between-next2:
 ⟦ distinct(vertices f); u ∈ 𝒱 f; v ∈ 𝒱 f; u ≠ v ⟧ ⟹
  between (vertices f) u (f · v) = between (vertices f) u v @ [v]
using split-between[OF - - - next-between2]
by (simp add: between-next-empty split:split-if-asm)


lemma nextVertex-eq-lemma:
 ⟦ distinct(vertices f); x ∈ 𝒱 f; y ∈ 𝒱 f; x ≠ y;
    v ∈ set(x # between (vertices f) x y) ⟧ ⟹
  f · v = nextElem (x # between (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(fastsimp simp:append-eq-Cons-conv)
 apply(drule split-list)
 apply(clarsimp simp add:between-def split-def hd-append split:list.split)
 apply(fastsimp 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(fastsimp 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(fastsimp 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(fastsimp simp: hd-append neq-Nil-conv split:list.split)
done

lemma nextVertex-eq-if-between-eq:
 ⟦ between (vertices f) x y = between (vertices f′) x y;
```

```
    distinct(vertices f); distinct(vertices f'); x ≠ y;
    x ∈ 𝒱 f; y ∈ 𝒱 f; x ∈ 𝒱 f'; y ∈ 𝒱 f';
    v ∈ set(x # between (vertices f) x y) ⟧ ⟹
  f • v = f' • v
by(simp add:nextVertex-eq-lemma[where z = 0])

end
```

11 Properties of Patch Enumeration

```
theory EnumeratorProps
imports Enumerator GraphProps
begin

lemma length-hideDupsRec[simp]: ⋀x. length(hideDupsRec x xs) = length xs
by(induct xs) auto

lemma length-hideDups[simp]: length(hideDups xs) = length xs
by(cases xs) simp-all

lemma length-indexToVertexList[simp]:
 length(indexToVertexList x y xs) = length xs
by(simp add:indexToVertexList-def)



constdefs increasing:: ('a::linorder) list ⇒ bool
 increasing ls ≡ ∀ x y as bs. ls = as @ x # y # bs ⟶ x ≤ y

lemma increasing1: ⋀ as x. increasing ls ⟹ ls = as @ x # cs @ y # bs ⟹ x
≤ 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: increasing (as@bs) ⟹ x ∈ set as ⟹ y ∈ set bs ⟹ x ≤ 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*: ∀ *as bs*. (*ls* = *as* @ *bs* ⟶ (∀ *x* ∈ *set as*. ∀ *y* ∈ *set bs*. *x* ≤ *y*)) ⟹ *increasing* (*ls*)
apply (*simp add*: *increasing-def*) **apply** *safe*
proof −
fix *as bs x y*
assume *p*: ∀ *asa bsa*. *as* @ *x* # *y* # *bs* = *asa* @ *bsa* ⟶ (∀ *x*∈*set asa*. ∀ *y*∈*set bsa*. *x* ≤ *y*)
then have *p′*: ⋀ *asa bsa*. *as* @ *x* # *y* # *bs* = *asa* @ *bsa* ⟹ (∀ *x*∈*set asa*. ∀ *y*∈*set bsa*. *x* ≤ *y*) **by** *auto*
then have (∀ *x*∈*set* (*as* @ [*x*]). ∀ *y*∈*set* (*y* # *bs*). *x* ≤ *y*) **by** (*rule-tac p′*) *auto*
then show *x* ≤ *y* **by** (*auto simp*: *increasing-def*)
qed

lemma *increasing4*: *increasing* (*as*@*bs*) ⟹ *increasing as*
apply (*simp add*: *increasing-def*) **apply** *safe* **by** *auto*

lemma *increasing5*: *increasing* (*as*@*bs*) ⟹ *increasing bs*
proof −
assume *nd*: *increasing* (*as*@*bs*)
then have *r*: ⋀ *x y asa bsa*. (∃ *asa bsa*. *as* @ *bs* = *asa* @ *x* # *y* # *bsa*) ⟹ *x* ≤ *y* **by** (*auto simp*: *increasing-def*)
show *?thesis* **apply** (*simp add*: *increasing-def*) **apply** (*intro allI impI*) **apply** (*rule-tac r*)
apply *auto* **apply** (*intro exI*) **apply** (*subgoal-tac as* @ *asa* @ *x* # *y* # *bs* = (*as* @ *asa*) @ *x* # *y* # *bs*)
by *assumption auto*
qed

lemma *enumBase-length*: *ls* ∈ *set* (*enumBase nmax*) ⟹ *length ls* = *1*
by (*auto simp*: *enumBase-def*)

lemma *enumBase-lenght2*: ∀ *ls* ∈ *set* (*enumBase nmax*). *length ls* = *1*
by (*auto simp*: *enumBase-def*)

lemma *enumBase-bound*: ∀ *y* ∈ *set* (*enumBase nmax*). ∀ *z* ∈ *set y*. *z* ≤ *nmax*
by (*auto simp*: *enumBase-def*)

lemmas *enumBase-simps* = *enumBase-length enumBase-lenght2 enumBase-bound*

lemma *enumAppend-length1*: (∀ *ls* ∈ *set lss*. *length ls* = *k*) ⟹ *ls2* ∈ *set* (*enumAppend nmax lss*) ⟹ *length ls2* = *k* + *1*
by (*auto simp*: *enumAppend-def*)

lemma *enumAppend-length*: (∀ *ls* ∈ *set lss*. *length ls* = *k*) ⟹ (∀ *ls2* ∈ *set* (*enumAppend nmax lss*). *length ls2* = *Suc k*)
by (*auto simp*: *enumAppend-def*)

lemma *enumAppend-length-rec*: (∀ *ls* ∈ *set lss*. *length ls* = *k*) ⟹
  (∀ *ls2* ∈ *set* (((*enumAppend nmax*) ^*n*) *lss*). *length ls2* = *k* + *n*)
apply (*induct n*) **by** (*auto intro*!: *enumAppend-length*)

lemma *enumAppend-length-rec2*: (∀ *ls* ∈ *set lss*. *length ls* = *k*)
  ⟹ *ls2* ∈ *set* (((*enumAppend nmax*) ^*n*) *lss*) ⟹ *length ls2* = *k* + *n*
by (*auto dest*: *enumAppend-length-rec*)

lemma *enumAppend-length-rec3*: (∀ *ls* ∈ *set lss*. *length ls* = *1*)
  ⟹ *ls2* ∈ *set* (((*enumAppend nmax*) ^*n*) *lss*) ⟹ *length ls2* = *Suc n*
by (*auto dest*: *enumAppend-length-rec2*)

lemma *enumAppend-bound*: *ls* ∈ *set* ((*enumAppend nmax*) *lss*) ⟹
∀ *y* ∈ *set lss*. ∀ *z* ∈ *set y*. *z* ≤ *nmax* ⟹ *x* ∈ *set ls* ⟹ *x* ≤ *nmax*
by (*auto simp add*: *enumAppend-def split*: *split-if-asm*)

lemma *enumAppend-bound-rec*: *ls* ∈ *set* (((*enumAppend nmax*) ^*n*) *lss*) ⟹
  ∀ *y* ∈ *set lss*. ∀ *z* ∈ *set y*. *z* ≤ *nmax* ⟹ *x* ∈ *set ls* ⟹ *x* ≤ *nmax*
proof −
  assume *ls*: *ls* ∈ *set* ((*enumAppend nmax* ^ *n*) *lss*) **and** *lss*: ∀ *y*∈*set lss*. ∀ *z*∈*set y*. *z* ≤ *nmax* **and** *x*: *x* ∈ *set ls*
  have *ind*:⋀ *lss*. ∀ *y*∈*set lss*. ∀ *z*∈*set y*. *z* ≤ *nmax* ⟹ ∀ *y* ∈ *set* (((*enumAppend nmax*) ^*n*) *lss*). ∀ *z* ∈ *set y*. *z* ≤ *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** ∀ *y* ∈ *set* (((*enumAppend nmax*) ^*n*) *lss*). ∀ *z* ∈ *set y*. *z* ≤ *nmax* **apply** (*rule-tac ind*) **.**
  with *ls x* **show** *?thesis* **by** *auto*
qed

lemma *enumAppend-increase-rec*:
  ⋀ *m as bs*. *ls* ∈ *set* (((*enumAppend nmax*) ^ *m*) (*enumBase nmax*)) ⟹

```
  as @ bs = ls ⟹  ∀ x ∈ set as. ∀ y ∈ set bs. x ≤ 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: ⋀m as bs.
          ⟦xs ∈ set ((enumAppend nmax ^ m) (enumBase nmax)); as @ bs = xs⟧
          ⟹ ∀x∈set as. ∀xa∈set bs. x ≤ xa
   and xs:xs @ [x] ∈ set (enumAppend nmax ((enumAppend nmax ^ n) (enumBase
nmax)))
    and asbs: as @ bs = xs @ [x] and xa:xa ∈ set as and xb: xb ∈ set bs and m:
m = Suc n
  from ih have ih2: ⋀ as bs x y. ⟦xs ∈ set ((enumAppend nmax ^ n) (enumBase
nmax)); as @ bs = xs; x ∈ set as; y ∈ set bs⟧
          ⟹ x ≤ y by auto

  from xb have bs ≠ []  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 ≤ x
  proof (cases xs rule: rev-exhaust)
    case Nil with xa xs′ show ?thesis by auto
  next
    case (snoc ys y)
    have xa ≤ y
    proof (cases xa = y)
      case True then show ?thesis by auto
    next
      case False
      from xa xs′ have xa ∈ set xs by auto
      with False snoc have xa ∈ set ys by auto
      with xs snoc  show ?thesis
        apply (rule-tac ih2)
        by (auto simp: enumAppend-def)
    qed
    with xs snoc show xa ≤ x by (auto simp: enumAppend-def split:split-if-asm)
  qed
  then show xa ≤ xb apply (cases xb = b) apply (simp add: beq)
  proof (rule-tac ih2)
    from xs
    show xs ∈ set ((enumAppend nmax ^ n) (enumBase nmax))
    by (auto simp: enumAppend-def)
  next
    from xs′ show as @ bs′ = xs by auto
  next
    from xa show xa ∈ set as by auto
```

```
  next
    assume xb ≠ b
    with xb bs′ show xb ∈ set bs′ by auto
  qed
qed

lemma enumAppend-length1: ⋀ls. ls ∈ set (((enumAppend nmax) ^n) lss) ⟹
 (∀ l ∈ set lss. |l| = k) ⟹ |ls| = k + n
apply (induct n)
apply simp
by (auto simp add:enumAppend-def split: split-if-asm)

lemma enumAppend-length2: ⋀ls. ls ∈ set (((enumAppend nmax) ^n) lss) ⟹
 (⋀l. l ∈ set lss ⟹ |l| = k) ⟹ K = k + n ⟹ |ls| = K
 by (auto simp add: enumAppend-length1)



lemma enum-enumerator:
 enum i j = enumerator i j
by(simp add: enum-def enumTab-def tabulate2-def tabulate-def tabulate′-def sub-def
sub1-def)



lemma enumerator-hd: ls ∈ set (enumerator m n) ⟹ hd ls = 0
by (auto simp: enumerator-def split: split-if-asm)

lemma enumerator-last: ls ∈ set (enumerator m n) ⟹ last ls = (n − 1)
by (auto simp: enumerator-def split: split-if-asm)

lemma enumerator-length: ls ∈ set (enumerator m n) ⟹ 2 ≤ length ls
by (auto simp: enumerator-def split: split-if-asm)

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

lemma enumerator-not-empty[dest]: ls ∈ set (enumerator m n) ⟹ ls ≠ []
apply (subgoal-tac 2 ≤ length ls) apply force by (rule enumerator-length)

lemma enumerator-length2: ls ∈ set (enumerator m n) ⟹ 2 < m ⟹ length ls
= m
proof −
  assume ls:ls ∈ set (enumerator m n) and m: 2 < m
  def k ≡ m − 3
  with m have k: m = k + 3 by arith
  with ls have ls ∈ set (enumerator (k+3) n) by auto
  then have 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 ∈ set (enumerator m nmax) ⟹
 0 < nmax ⟹ x ∈ set ls ⟹ x < nmax
apply (auto simp: enumerator-def split: split-if-asm)
apply (subgoal-tac x ≤ nmax − 2) apply arith
apply (rule-tac enumAppend-bound-rec) by(auto simp:enumBase-simps)

lemma enumerator-bound2: ls ∈ set (enumerator m nmax) ⟹ 1 < nmax ⟹ x
∈ set (butlast ls) ⟹ x < nmax − Suc 0
apply (auto simp: enumerator-def split: split-if-asm)
apply (subgoal-tac x ≤  (nmax − 2)) apply arith
apply (rule-tac enumAppend-bound-rec) by(auto simp:enumBase-simps)

lemma enumerator-bound3: ls ∈ set (enumerator m nmax) ⟹ 1 < nmax ⟹
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 ≤ length (ys @ [y])) apply simp
apply (rule-tac enumerator-length) by auto


lemma enumerator-increase: ⋀ as bs. ls ∈ set (enumerator m nmax) ⟹  as @
bs = ls ⟹ ∀ x ∈ set as. ∀ y ∈ set bs. x ≤ y
apply (auto simp: enumerator-def del: Nat.diff-is-0-eq' split: split-if-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) apply arith
by (auto dest!: enumAppend-increase-rec)

lemma enumerator-increasing: ls ∈ set (enumerator m nmax) ⟹ increasing ls
apply (rule increasing3)
by (auto dest: enumerator-increase)

constdefs incrIndexList:: nat list ⇒ nat ⇒ nat ⇒ bool
 incrIndexList ls m nmax ≡
  1 < m ∧ 1 < nmax ∧
  hd ls = 0 ∧ last ls = (nmax − 1) ∧ length ls = m
 ∧ last (butlast ls) < last ls ∧ increasing ls

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

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

lemma *incrIndexList-help2*[*simp*]: *incrIndexList ls m nmax* ⟹ *hd ls* = *0*
by (*unfold incrIndexList-def*) *simp*

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

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

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

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

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

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

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

lemma *incrIndexList-help8*[*simp*]: *incrIndexList ls m nmax* ⟹ *butlast ls* ≠ []
proof (*rule ccontr*)
  assume *props*: *incrIndexList ls m nmax* **and** *butl*: ¬ *butlast ls* ≠ []
  then have *ls* ≠ [] **by** *auto*
  then have *ls'*: *ls* = (*butlast ls*) @ [*last ls*] **by** *auto*
  def *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*]: ¬ *incrIndexList* [*l*] *m nmax*
by (*auto dest*: *incrIndexList-help8*)

lemma *incrIndexList-help9*[*intro*]: (*incrIndexList ls m nmax*) ⟹
  x ∈ *set* (*butlast ls*) ⟹ *x* ≤ *nmax* − *2*

proof −
assume *props*: (*incrIndexList ls m nmax*) **and** *x*: *x* ∈ *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*) ≤ *nmax* − *2* **by** *arith*
from *props* **have** *ls* ≠ [] **by** *auto*
then have *ls1*: *ls* = *butlast ls* @ [*last ls*] **by** *auto*
def *ls′* ≡ (*butlast* (*butlast ls*))
def *last2* ≡ *last* (*butlast ls*)
def *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 enumerator-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* (λ*x*. *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* (λ*x*. *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)
  def ls' ≡ butlast ls
  def l ≡ last ls
  def ll ≡ last ls'
  def bl ≡ butlast ls'
  def ls'list ≡ (enumAppend nmax ^ n) (enumBase nmax)
  then have short: (enumAppend nmax ^ n) (enumBase nmax) = ls'list by simp
  from Suc have ls ≠ [] 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' ≠ [] by auto
  with ll-def bl-def have ls'1: ls' = bl @ [ll] by auto
  then have ll-in-ls': ll ∈ set ls' by simp
  from Suc ls1 have list-all (λx. x < Suc nmax) ls' by auto
  with ll-in-ls' have ll < Suc nmax by (induct ls') auto
  with ll-def have llsmall: last ls' ≤ nmax  by auto

  from ls1 have l-in-ls: l ∈ set ls by auto
  from Suc have list-all (λx. x < Suc nmax) ls by auto
  with l-in-ls have l < Suc nmax by (induct ls) auto
  then have lo: l ≤ nmax by auto

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

  from Suc ls1 have vors: ls' ∈ set ((enumAppend nmax ^ n) (enumBase nmax))
    by (rule-tac Suc) (auto intro: increasing4)
  with short have ls' ∈ 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 ⟹ incrIndexList ls m nmax ⟹
  ls ∈ set (enumerator m nmax)
proof −
  assume m: 2 < m and props: incrIndexList ls m nmax
  then have props': (hd ls = 0 ∧ last ls = (nmax − 1)
   ∧ length ls = m ∧ last (butlast ls) < last ls ∧  increasing ls)
   by (unfold incrIndexList-def) auto
  show ?thesis
  proof −
    have props'': hd ls = 0 ∧ last ls = (nmax − 1) ∧ length ls = m ∧
      increasing ls
```

```
      by (auto simp: props')
    show ls ∈ set (enumerator m nmax)
    proof −
      from m props'' have l-ls: 2 < length ls  by auto
      then have ∃ 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*(*makeFaceFinal f g*) = *vertices g*
by(*induct g*)(*simp add*:*vertices-graph-def makeFaceFinal-def*)

lemma *edges-makeFaceFinal*: $\mathcal{E}$ (*makeFaceFinal f g*) = $\mathcal{E}$ *g*
proof −
  { fix *fs*
    have $(\bigcup_{f \in set\ (makeFaceFinalFaceList\ f\ fs)}\ edges\ f) = (\bigcup_{f \in\ set\ fs}\ edges\ f)$
    apply(*unfold makeFaceFinalFaceList-def*)
    apply(*induct f*)
    by(*induct fs*) *simp-all* **}**
  thus *?thesis* **by**(*simp add*:*edges-graph-def makeFaceFinal-def*)
qed

lemma *nonFins-mkFaceFin*:
  nonFinals(*makeFaceFinal f g*) = *remove1 f* (*nonFinals g*)
by(*simp add*:*makeFaceFinal-def nonFinals-def makeFaceFinalFaceList-def*
      filter-replace)

lemma *in-set-repl-setFin*:
  f ∈ *set fs* ⟹ *final f* ⟹ *f* ∈ *set* (*replace f′* [*setFinal f′*] *fs*)
by (*induct fs*) *auto*

lemma *in-set-repl*: *f* ∈ *set fs* ⟹ *f* ≠ *f′* ⟹ *f* ∈ *set* (*replace f′ fs′ fs*)
by (*induct fs*) *auto*

lemma *makeFaceFinals-preserve-finals*:
  f ∈ *set* (*finals g*) ⟹ *f* ∈ *set* (*finals* (*makeFaceFinal f′ g*))
by (*induct g*)
  (*simp add*:*makeFaceFinal-def finals-def makeFaceFinalFaceList-def*
      in-set-repl-setFin)

lemma *len-faces-makeFaceFinal*[*simp*]:
|*faces* (*makeFaceFinal f g*)| = |*faces g*|
by(*simp add*:*makeFaceFinal-def makeFaceFinalFaceList-def*)

lemma *len-finals-makeFaceFinal*:
f ∈ $\mathcal{F}$ *g* ⟹ ¬ *final f* ⟹ |*finals* (*makeFaceFinal f g*)| = |*finals g*| + *1*

by(*simp add*:*makeFaceFinal-def finals-def makeFaceFinalFaceList-def*
length-filter-replace1)

lemma *len-nonFinals-makeFaceFinal*:
⟦ ¬ *final f*; *f* ∈ 𝓕 *g*⟧
⟹ |*nonFinals* (*makeFaceFinal f g*)| = |*nonFinals g*| − *1*
by(*simp add*:*makeFaceFinal-def nonFinals-def makeFaceFinalFaceList-def*
length-filter-replace2)

lemma *set-finals-makeFaceFinal*[*simp*]: *distinct*(*faces g*) ⟹ *f* ∈ 𝓕 *g* ⟹
set(*finals* (*makeFaceFinal f g*)) = *insert* (*setFinal f*) (*set*(*finals g*))
by(*auto simp*:*finals-def makeFaceFinal-def makeFaceFinalFaceList-def*
distinct-set-replace)

lemma *splitFace-preserve-final*:
f ∈ *set* (*finals g*) ⟹ ¬ *final f′* ⟹
f ∈ *set* (*finals* (*snd* (*snd* (*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*:
¬ *final* (*fst* (*snd* (*splitFace g i j f′ ns*)))
by (*simp add*: *splitFace-def split-def split-face-def*)

lemma *subdivFace′-preserve-finals*:
⋀*n i f′ g*. *f* ∈ *set* (*finals g*) ⟹ ¬ *final f′* ⟹
f ∈ *set* (*finals* (*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 ∈ *set* (*finals g*) ⟹ ¬ *final f′* ⟹
f ∈ *set* (*finals* (*subdivFace g f′ is*))
by(*simp add*:*subdivFace-def subdivFace′-preserve-finals*)

declare *Nat.diff-is-0-eq′* [*simp del*]

12.2 *is-prefix*

constdefs *is-prefix* :: *′a list* ⇒ *′a list* ⇒ *bool*
is-prefix ls vs ≡ (∃ *bs*. *vs* = *ls* @ *bs*)

lemma *is-prefix-add*:
is-prefix ls vs ⟹ *is-prefix* (*as* @ *ls*) (*as* @ *vs*) **by** (*simp add*: *is-prefix-def*)

lemma *is-prefix-hd*[*simp*]:
is-prefix [*l*] *vs* = (*l* = *hd vs* ∧ *vs* ≠ [])
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* ∈ *set vs* ⟹ *is-prefix* (*fst* (*splitAt ram vs*) @ [*ram*]) *vs*
by (*auto dest*!: *splitAt-ram simp*: *is-prefix-def*)

12.3 *is-postfix*

constdefs *is-postfix* :: *′a list* ⇒ *′a list* ⇒ *bool*
is-postfix ls vs ≡ (∃ *as*. *vs* = *as* @ *ls*)

lemma *is-postfix-add*:
is-postfix ls vs ⟹ *is-postfix* (*ls* @ *bs*) (*vs* @ *bs*) **by** (*simp add*: *is-postfix-def*)

lemma *is-pre-post-eq*:
distinct vs ⟹ *ls* ≠ [] ⟹ *is-prefix ls vs* ⟹ *is-postfix ls vs* ⟹ *ls* = *vs*
proof −
assume *d*: *distinct vs* **and** *ls*: *ls* ≠ [] **and** *ispre*: *is-prefix ls vs* **and** *ispost*: *is-postfix ls vs*
from *ls* **have** *hd ls* # *tl ls* = *ls* **by** *auto*
from *ispre* **obtain** *bs* **where** *vs1*: *vs* = *ls* @ *bs* **apply** (*simp add*: *is-prefix-def*) **by** *auto*
from *ls vs1* **have** *vs2*: *vs* = *hd ls* # *tl ls* @ *bs* **by** *auto*
from *ispost* **obtain** *as* **where** *vs3*: *vs* = *as* @ *ls* **apply** (*simp add*: *is-postfix-def*) **by** *auto*
from *ls vs3* **have** *vs4*: *vs* = *as* @ *hd ls* # *tl ls* **by** *auto*
from *d vs2 vs4* **have** *as* = [] **apply** (*rule-tac dist-at1*) **apply** *assumption* **apply** *assumption* **by** *auto*
with *ls vs4* **show** *?thesis* **by** *auto*
qed

```
lemma splitAt-is-postfix: ram ∈ set vs ⟹ is-postfix (ram # snd (splitAt ram vs))
vs
by (auto dest!: splitAt-ram simp: is-postfix-def)
```

12.4 *is-sublist*

```
constdefs is-sublist :: 'a list ⇒ 'a list ⇒ bool
is-sublist ls vs ≡ (∃ as bs. vs = as @ ls @ bs)

lemma is-prefix-sublist:
  is-prefix ls vs ⟹ is-sublist ls vs by (auto simp: is-prefix-def is-sublist-def)

lemma is-postfix-sublist:
  is-postfix ls vs ⟹ is-sublist ls vs by (force simp: is-postfix-def is-sublist-def)

lemma is-sublist-trans: is-sublist as bs ⟹ is-sublist bs cs ⟹ 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 ⟹ 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)
  case goal1 show ?case
  proof
    case goal1 show ?case
    proof
      case goal1 note xs = goal1
      show ?case
      proof
        case goal1 show ?case by auto
      next
        case goal2 show ?case
        proof
          case goal1
          have ys = take |xs| ys @ drop |xs| ys by simp
          also have ... = [] @ xs @ drop |xs| ys by(simp add:xs[symmetric])
          finally show ?case by blast
        qed
      qed
    qed
```

```
  next
    case goal2 show ?case
    proof
      case goal1 note xs-neq = this
      show ?case
      proof
        case goal1 show ?case by auto
      next
        case goal2 show ?case
        proof
          case goal1 note not-less = this show ?case
          proof
            case goal1
            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 goal2
            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| ⟹ ¬ is-sublist xs ys
by(simp add:is-sublist-rec)

lemma is-sublist-simp[simp]: a ≠ v ⟹ is-sublist (a#as) (v#vs) = is-sublist
(a#as) vs
proof
  assume av: a ≠ v and subl: 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 ≠ [] 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-sublist (a # as) vs by (auto simp: is-sublist-def)
```

```
next
  assume av: a ≠ v and subl: is-sublist (a # as) vs
  then show is-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-sublist-id[simp]: is-sublist vs vs apply (auto simp: is-sublist-def) apply
(intro exI)
  apply (subgoal-tac vs = [] @ vs @ []) by (assumption) auto

lemma is-sublist-in: is-sublist (a#as) vs ⟹ a ∈ set vs by (auto simp: is-sublist-def)

lemma is-sublist-in1: is-sublist [x,y] vs ⟹ y ∈ set vs by (auto simp: is-sublist-def)

lemma is-sublist-notlast[simp]: distinct vs ⟹ x = last vs ⟹ ¬ is-sublist [x,y]
vs
proof
  assume dvs: distinct vs and xl: x = last vs and subl:is-sublist [x, y] vs
  then obtain rs ts where vs = rs @ x # y # ts by (auto simp: is-sublist-def)
  def as ≡  rs @ [x]
  def bs ≡  y # ts
  then have bsne: bs ≠ [] by auto
  from as-def bs-def have vs: vs = as @ bs by auto
  with as-def have xas: x ∈ set as by auto
  from bsne vs 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 ∈ set bs by (induct bs) auto
  with xas vs dvs show False by auto
qed

lemma is-sublist-nth1: is-sublist [x,y] ls ⟹
  ∃ i j. i < length ls ∧ j < length ls ∧ ls!i = x ∧ ls!j = y ∧ Suc i = j
proof −
  assume subl: is-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 ∧ (Suc (length as)) < length ls ∧ ls!(length
as) = x
      ∧ ls!(Suc (length as)) = y ∧ Suc (length as) = (Suc (length as))
    apply auto apply (induct as) by auto
  then show ?thesis by auto
qed

lemma is-sublist-nth2: ∃ i j. i < length ls ∧ j < length ls ∧ ls!i = x ∧ ls!j = y
∧ Suc i = j ⟹
 is-sublist [x,y] ls
proof −
```

assume ∃ *i j*. *i* < *length ls* ∧ *j* < *length ls* ∧ *ls*!*i* = *x* ∧ *ls*!*j* = *y* ∧ *Suc i* = *j*
then obtain *i j* **where** *vors*: *i* < *length ls* ∧ *j* < *length ls* ∧ *ls*!*i* = *x* ∧ *ls*!*j* = *y* ∧ *Suc i* = *j* **by** *auto*
then have *ls* = *take* (*Suc* (*Suc i*)) *ls* @ *drop* (*Suc* (*Suc i*)) *ls* **by** *auto*
with *vors* **have** *ls* = *take* (*Suc i*) *ls* @ [*ls*! (*Suc i*)] @ *drop* (*Suc* (*Suc i*)) *ls*
by (*auto simp*: *take-Suc-conv-app-nth*)
with *vors* **have** *ls* = *take i ls* @ [*ls*!*i*] @ [*ls*! (*Suc i*)] @ *drop* (*Suc* (*Suc i*)) *ls*
by (*auto simp*: *take-Suc-conv-app-nth*)
with *vors* **show** *?thesis* **by** (*auto simp*: *is-sublist-def*)
qed

lemma *is-sublist-nth-eq*: (∃ *i j*. *i* < *length ls* ∧ *j* < *length ls* ∧ *ls*!*i* = *x*
∧ *ls*!*j* = *y* ∧ *Suc i* = *j*) = *is-sublist* [*x*,*y*] *ls*
apply (*intro iffI*) **apply** (*rule is-sublist-nth2*) **apply** *assumption* **apply** (*rule is-sublist-nth1*) **.**

lemma *is-sublist-tl*: *is-sublist* (*a* # *as*) *vs* ⟹ *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* ⟹ *is-sublist* [*a*] *vs* **apply** (*simp add*: *is-sublist-def*) **by** *auto*

lemma *is-sublist-hd-eq*[*simp*]: (*is-sublist* [*a*] *vs*) = (*a* ∈ *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* ∈ *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*) ⟹ *distinct* (*v* # *vs*) ⟹ *is-prefix as vs*
proof −
assume *d*: *distinct* (*v* # *vs*) **and** *subl*: *is-sublist* (*v* # *as*) (*v* # *vs*)
from *subl* **obtain** *rs ts* **where** *v-vs*: *v* # *vs* = *rs* @ (*v* # *as*) @ *ts* **by** (*simp add*: *is-sublist-def*) *auto*
from *d* **have** *v*: *v* ∉ *set vs* **by** *auto*
then have ¬ *is-sublist* (*v* # *as*) *vs* **by** (*auto dest*: *is-sublist-hd*)
with *v-vs* **have** *rs* = [] **apply** (*cases rs*) **by** (*auto simp*: *is-sublist-def*)
with *v-vs* **show** *is-prefix as vs* **by** (*auto simp*: *is-prefix-def*)
qed

lemma *is-sublist-distinct*[*intro*]:
is-sublist as vs ⟹ *distinct vs* ⟹ *distinct as* **by** (*auto simp*: *is-sublist-def*)

lemma *is-sublist-x-last*: *distinct vs* ⟹ *x* = *last vs* ⟹ ¬ *is-sublist* [*x*,*y*] *vs*
proof

assume *d*: *distinct vs* **and** *xl*: *x* = *last vs* **and** *subl*: *is-sublist* [*x*, *y*] *vs*
then obtain *rs ts* **where** *vs*: *vs* = *rs* @ *x* # *y* # *ts* **by** (*auto simp*: *is-sublist-def*)
def *as* ≡ *rs* @ [*x*]
def *bs* ≡ *y* # *ts*
then have *bsne*: *bs* ≠ [] **by** *auto*
from *as-def bs-def* **have** *vs*: *vs* = *as* @ *bs* **by** *auto*
from *as-def* **have** *xas*: *x* ∈ *set as* **by** *auto*
from *vs bs-def* **have** *last vs* = *last bs* **by** *auto*
with *xl* **have** *x* = *last bs* **by** *auto*
with *bsne* **have** *x* ∈ *set bs* **by** (*induct bs*) *auto*
with *d xas vs* **show** *False* **by** *auto*
qed

lemma *is-sublist-y-hd*: *distinct vs* ⟹ *y* = *hd vs* ⟹ ¬ *is-sublist* [*x*,*y*] *vs*
proof
assume *d*: *distinct vs* **and** *yh*: *y* = *hd vs* **and** *subl*: *is-sublist* [*x*, *y*] *vs*
then obtain *rs ts* **where** *vs*: *vs* = *rs* @ *x* # *y* # *ts* **by** (*auto simp*: *is-sublist-def*)
def *as* ≡ *rs* @ [*x*]
then have *asne*: *as* ≠ [] **by** *auto*
def *bs* ≡ *y* # *ts*
then have *bsne*: *bs* ≠ [] **by** *auto*
from *as-def bs-def* **have** *vs2*: *vs* = *as* @ *bs* **by** *auto*
from *bs-def* **have** *xbs*: *y* ∈ *set bs* **by** *auto*
from *vs2 asne* **have** *hd vs* = *hd as* **by** *simp*
with *yh* **have** *y* = *hd as* **by** *auto*
with *asne* **have** *y* ∈ *set as* **by** (*induct as*) *auto*
with *d xbs vs2* **show** *False* **by** *auto*
qed

lemma *is-sublist-at1*: *distinct* (*as* @ *bs*) ⟹ *is-sublist* [*x*,*y*] (*as* @ *bs*) ⟹ *x* ≠ (*last as*) ⟹
is-sublist [*x*,*y*] *as* ∨ *is-sublist* [*x*,*y*] *bs*
proof (*cases x* ∈ *set as*)
assume *d*: *distinct* (*as* @ *bs*) **and** *subl*: *is-sublist* [*x*, *y*] (*as* @ *bs*) **and** *xnl*: *x* ≠ *last as*
def *vs* ≡ *as* @ *bs*
with *d* **have** *dvs*: *distinct vs* **by** *auto*
case *True*
with *xnl subl* **have** *ind*: *is-sublist* (*as*@*bs*) *vs* ⟹ *is-sublist* [*x*, *y*] *as*
proof (*induct as*)
case *Nil* **then show** *?case* **by** *force*
next
case (*Cons a as*)
assume *ih*: ⟦*is-sublist* (*as*@*bs*) *vs*; *x* ≠ *last as*; *is-sublist* [*x*,*y*] (*as* @ *bs*); *x* ∈ *set as*⟧ ⟹
is-sublist [*x*, *y*] *as* **and** *subl-aas-vs*: *is-sublist* ((*a* # *as*) @ *bs*) *vs*
and *xnl2*: *x* ≠ *last* (*a* # *as*) **and** *subl2*: *is-sublist* [*x*, *y*] ((*a* # *as*) @ *bs*)
and *x*: *x* ∈ *set* (*a* # *as*)
then have *rule1*: *x* ≠ *a* ⟹ *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*: *distinct* (*a* # *as* @ *bs*) **apply** (*rule-tac is-sublist-distinct*) **by** *auto*
from *xnl2* **have** *asne*: *x* = *a* ⟹ *as* ≠ [] **by** *auto*
with *subl2 daas* **have** *yhdas*: *x* = *a* ⟹ *y* = *hd as* **apply** *simp* **apply** (*drule-tac is-sublist-distinct-prefix*) **by** *auto*
with *asne* **have** *x* = *a* ⟹ *as* = *y* # *tl as* **by** *auto*
with *asne yhdas* **have** *x* = *a* ⟹ *is-prefix* [*x*,*y*] (*a* # *as*) **by** *auto*
then have *rule2*: *x* = *a* ⟹ *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*: *distinct* (*as* @ *bs*) **and** *subl*: *is-sublist* [*x*, *y*] (*as* @ *bs*) **and** *xnl*: *x* ≠ *last as*
def *ars* ≡ *as*
case *False*
with *ars-def* **have** *xars*: *x* ∉ *set ars* **by** *auto*
from *subl* **have** *ind*: *is-sublist as ars* ⟹ *is-sublist* [*x*, *y*] *bs*
proof (*induct as*)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons a as*)
assume *ih*: ⟦*is-sublist as ars*; *is-sublist* [*x*, *y*] (*as* @ *bs*)⟧ ⟹ *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* ≠ *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-at2*: *distinct* (*as* @ *bs*) ⟹ *is-sublist* [*x*,*y*] (*as* @ *bs*) ⟹ *x* ≠ (*last as*) ⟹
is-sublist [*x*,*y*] *as* ≠ *is-sublist* [*x*,*y*] *bs*
apply *auto*
apply (*subgoal-tac x* ∈ *set as*)
apply (*subgoal-tac x* ∈ *set bs*)
apply (*subgoal-tac x* ∈ (*set as* ∩ *set bs*)) **apply** *simp* **apply** *force* **apply** (*force simp*:*is-sublist-in*)
apply (*force simp*:*is-sublist-in*)
apply (*subgoal-tac distinct* (*as* @ *bs*))
apply (*frule is-sublist-at1*) **by** *auto*

```
lemma is-sublist-at3: distinct (as @ bs) ⟹ is-sublist [x,y] (as @ bs) ⟹
  (is-sublist [x,y] as ∨ is-sublist [x,y] bs) ∨ x = last as
  apply (rule disjCI) apply (rule is-sublist-at1) by auto

lemma is-sublist-at4: distinct (as @ bs) ⟹ is-sublist [x,y] (as @ bs) ⟹
  as ≠ [] ⟹ x = last as ⟹ y = hd bs
proof −
  assume d: distinct (as @ bs) and subl: is-sublist [x,y] (as @ bs)
    and asne: as ≠ [] and xl: x = last as
  def 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) ⟹ is-sublist [x,y] (as @ bs) ⟹
  is-sublist [x,y] as ∨ is-sublist [x,y] bs ∨ x = last as ∧ 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 (drule-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 = (rev bs) @ b # a # rev as) apply assumption
    apply (subgoal-tac rev (rev zs) = rev (as @ a # b # bs))
      apply (thin-tac rev zs = as @ a # b # bs) apply simp
      apply simp
    apply (elim exE) apply (intro exI) by force

lemma is-sublist-at5'[simp]:
 distinct as ⟹ distinct bs ⟹ set as ∩ set bs = {} ⟹ is-sublist [x,y] (as @ bs)
⟹
 is-sublist [x,y] as ∨ is-sublist [x,y] bs ∨ x = last as ∧ y = hd bs
apply (subgoal-tac distinct (as @ bs)) apply (drule is-sublist-at5) by auto

lemma splitAt-is-sublist1: is-sublist (fst (splitAt ram vs)) vs apply (cases ram ∈
set vs)
apply (auto dest!: splitAt-ram splitAt-no-ram simp: is-sublist-def) apply (intro
```

exI)
apply (*subgoal-tac vs* = [] @ *fst* (*splitAt ram vs*) @ *ram* # *snd* (*splitAt ram vs*))
apply *assumption* **by** *simp*

lemma *splitAt-is-sublist1R*[*simp*]: *ram* ∈ *set vs* ⟹ *is-sublist* (*fst* (*splitAt ram vs*) @ [*ram*]) *vs*
apply (*auto dest*!: *splitAt-ram simp*: *is-sublist-def*) **apply** (*intro exI*)
apply (*subgoal-tac vs* = [] @ *fst* (*splitAt ram vs*) @ *ram* # *snd* (*splitAt ram vs*))
apply *assumption* **by** *simp*

lemma *splitAt-is-sublist2*:*is-sublist* (*snd* (*splitAt ram vs*)) *vs* **apply** (*cases ram* ∈ *set vs*)
apply (*auto dest*!: *splitAt-ram splitAt-no-ram simp*: *is-sublist-def*) **apply** (*intro exI*)
apply (*subgoal-tac vs* = (*fst* (*splitAt ram vs*) @ [*ram*]) @ *snd* (*splitAt ram vs*) @ []) **apply** *assumption* **by** *auto*

lemma *splitAt-is-sublist2R*[*simp*]: *ram* ∈ *set vs* ⟹ *is-sublist* (*ram* # *snd* (*splitAt ram vs*)) *vs*
apply (*auto dest*!: *splitAt-ram splitAt-no-ram simp*: *is-sublist-def*) **apply** (*intro exI*)
apply (*subgoal-tac vs* = *fst* (*splitAt ram vs*) @ *ram* # *snd* (*splitAt ram vs*) @ [])
apply *assumption* **by** *auto*

12.5 *is-nextElem*

constdefs *is-nextElem* :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *bool*
is-nextElem xs x y ≡ *is-sublist* [*x*,*y*] *xs* ∨ *xs* ≠ [] ∧ *x* = *last xs* ∧ *y* = *hd xs*

lemma *is-nextElem-a*[*intro*]: *is-nextElem vs a b* ⟹ *a* ∈ *set vs*
by (*auto simp*: *is-nextElem-def is-sublist-def*)
lemma *is-nextElem-b*[*intro*]: *is-nextElem vs a b* ⟹ *b* ∈ *set vs*
by (*auto simp*: *is-nextElem-def is-sublist-def*)
lemma *is-nextElem-sublist*: *is-nextElem vs x y* ⟹ *x* ≠ *last vs* ⟹ *is-sublist* [*x*,*y*] *vs*
by (*auto simp*: *is-nextElem-def*)
lemma *is-nextElem-last-hd*[*intro*]: *distinct vs* ⟹ *is-nextElem vs x y* ⟹
x = *last vs* ⟹ *y* = *hd vs*
by (*auto simp*: *is-nextElem-def*)
lemma *is-nextElem-last-ne*[*intro*]: *distinct vs* ⟹ *is-nextElem vs x y* ⟹
x = *last vs* ⟹ *vs* ≠ []
by (*auto simp*: *is-nextElem-def*)
lemma *is-nextElem-sublist2*: *is-nextElem vs x y* ⟹ *y* ≠ *hd vs* ⟹ *is-sublist* [*x*,*y*] *vs*
by (*auto simp*: *is-nextElem-def*)
lemma *is-nextElem-sublistI*: *is-sublist* [*x*,*y*] *vs* ⟹ *is-nextElem vs x y*
by (*auto simp*: *is-nextElem-def*)
lemma *is-nextElem-hd-lastI*: *vs* ≠ [] ⟹ *y* = *hd vs* ⟹ *x* = *last vs* ⟹ *is-nextElem*

```
vs x y
  by (auto simp: is-nextElem-def)

lemma is-nextElem-nth1: is-nextElem ls x y ⟹ ∃ i j. i < length ls
  ∧ j < length ls ∧ ls!i = x ∧ ls!j = y ∧ (Suc i) mod (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 ≠ [] ∧ last ls = x ∧ 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 ∧ 0 < length ls ∧ ls!(length ls −
1) = x
    ∧ ls!0 = y ∧ (Suc (length ls − 1)) mod (length ls) = 0 by auto
  then show ?thesis apply (intro exI) .
qed

lemma is-nextElem-nth2:  ∃ i j. i < length ls ∧ j < length ls ∧ ls!i = x ∧ ls!j =
y
   ∧ (Suc i) mod (length ls) = j ⟹ is-nextElem ls x y
proof −
  assume ∃ i j. i < length ls ∧ j < length ls ∧ ls!i = x ∧ ls!j = y ∧ (Suc i) mod
(length ls) = j
  then obtain i j where vors: i < length ls ∧ j < length ls ∧ ls!i = x ∧ ls!j = y
    ∧ (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:
  is-nextElem (rotate m ls) x y ⟹ is-nextElem ls x y
proof −
  assume is-nextElem: is-nextElem (rotate m ls) x y
  def 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 auto
  then obtain i j where vors:i < length (rotate n ls) ∧ j < length (rotate n ls)
```

```
∧
    (rotate n ls)!i = x ∧ (rotate n ls)!j = y ∧
    (Suc i) mod (length (rotate n ls)) = j by (drule-tac is-nextElem-nth1) auto
  then have lls: 0 < length ls by auto
  def k ≡ (i+n) mod (length ls)
  with lls have sk: k < length ls by auto
  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 auto
  def l ≡ (j+n) mod (length ls)
  with lls have sl: l < length ls by auto
  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 auto
  have mod-add1-eq': ⋀ a b c. (a mod c + b mod c) mod c = (a+b) mod (c::nat)
    apply (rule sym) by (rule mod-add1-eq)
  from vors k-def l-def
  have (Suc i) mod length ls = j by auto
  then have (Suc i) mod length ls = j mod length ls by auto
  then have (Suc i) mod length ls + n mod (length ls) = j mod length ls + n 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 auto
  then have ((Suc i) + n) mod length ls = (j + n ) mod length ls by (simp add: mod-add1-eq')
  then have (1 + i + n) mod length ls = (j + n ) mod length ls by simp
  then have (1 mod length ls + (i + n) mod length ls) mod length ls = (j + n ) mod length ls
    by (simp add: mod-add1-eq')
  with lls have (1 + (i + n) mod length ls) mod length ls = (j + n ) mod length ls
     apply (cases 1 < length ls) apply force apply (subgoal-tac length ls = 1) apply simp by arith
  with vors k-def l-def have (Suc k) mod (length ls) = l by auto
  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) apply (rule is-nextElem-rotate1)
  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 ≅ ms ⟹ is-nextElem ls x y = is-nextElem ms x y
by (auto simp: congs-def)
```

lemma *is-nextElem-rotate1*: *distinct ls* $\Longrightarrow$ *is-nextElem ls x y* $\Longrightarrow$ *is-nextElem* (*rotate1 ls*) *x y*
proof (*cases ls*)
case *Nil*
assume *is-nextElem ls x y*
with *Nil* **show** *?thesis* **by** *auto*
next
assume *d*: *distinct ls* **and** *is-nextElem*: *is-nextElem ls x y*
case (*Cons m ms*)
then have *ls*: *ls* = *m* # *ms* **by** *auto*
have *ms* $\neq$ [] $\Longrightarrow$ *last ms* $\in$ *set ms* **by** (*induct ms*) *auto*
with *is-nextElem d ls* **have** *rule1*: *x* = *m* $\Longrightarrow$ *y* = *hd* (*ms* @ [*m*])
apply (*auto simp*: *is-nextElem-def is-sublist-def*) **apply** (*subgoal-tac ms* = *y* # *bs*) **apply** *force*
apply (*rule dist-at2*) **apply** (*rule d*) **apply** (*thin-tac m* # *ms* = *as* @ *m* # *y* # *bs*)
apply (*simp add*: *ls*) **apply** *force* **apply** (*simp add*: *Cons*)
by (*simp split*: *split-if-asm*)
from *is-nextElem d ls* **have** *rule2*: *x* $\neq$ *m* $\wedge$ *is-sublist* [*x*, *y*] *ms* $\Longrightarrow$ *is-nextElem* (*rotate1 ls*) *x y*
apply (*simp add*: *is-nextElem-def split*: *split-if-asm*)
apply *auto* **apply** (*subgoal-tac is-sublist* [*x*,*y*] ([] @ *ms* @ [*m*])) **apply** *force*
apply (*rule is-sublist-add*) **by** *simp*
from *is-nextElem d ls* **have** *rule3*: *x* $\neq$ *m* $\wedge$ $\neg$ *is-sublist* [*x*,*y*] *ms* $\Longrightarrow$ *is-nextElem* (*rotate1 ls*) *x y*
apply (*simp add*: *is-nextElem-def split*: *split-if-asm*)
apply *auto* **apply** (*simp add*: *is-sublist-def*) **apply** (*intro exI*)
apply (*subgoal-tac ms* @ [*m*] = *butlast ms* @ *last ms* # *m* # []) **apply** *assumption*
by *auto*
from *Cons rule1 rule2 rule3* **show** *?thesis* **apply** (*cases x* $\neq$ *m*)
apply *auto* **by** (*simp add*: *is-nextElem-def is-sublist-def split*: *split-if-asm*)
qed

lemma *is-nextElem-congsI*: *ls1* $\cong$ *ls2* $\Longrightarrow$ *distinct ls1* $\Longrightarrow$ *is-nextElem ls1 x y* $\Longrightarrow$ *is-nextElem ls2 x y*
proof –
assume *eq*: *ls1* $\cong$ *ls2* **and** *is-nextElem*: *is-nextElem ls1 x y* **and** *d*: *distinct ls1*
then obtain *n* **where** *rot*: *ls2* = *rotate n ls1* **by** (*unfold congs-def*) *auto*
from *is-nextElem d* **have** *is-nextElem* (*rotate n ls1*) *x y* **apply** (*induct n*)
by (*auto intro*: *is-nextElem-rotate1*)
with *rot* **show** *?thesis* **by** *auto*
qed

lemma *is-nextElem-congseq*: *ls1* $\cong$ *ls2* $\Longrightarrow$ *distinct ls1* $\Longrightarrow$ *is-nextElem ls1 x y* = *is-nextElem ls2 x y*
proof
assume *ls1* $\cong$ *ls2 distinct ls1 is-nextElem ls1 x y*
then show *is-nextElem ls2 x y* **by** (*auto intro*: *is-nextElem-congsI*)

```
next
  assume eqF: ls1 ≅ ls2 and d: distinct ls1 and is-nextElem: is-nextElem ls2 x y
  from eqF have eqF′: ls2 ≅ ls1 by (rule congs-sym)
  from eqF d have d′: distinct ls2 by (auto simp: congs-distinct)
 from eqF′ d′ is-nextElem show is-nextElem ls1 x y by (auto intro: is-nextElem-congsI)
qed

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) ∧ 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 ⟧ ⟹ |xs| ≤ 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


lemma cong-if-is-nextElem-eq:
 ⟦ distinct xs; distinct ys; set xs = set ys;
    !x y. is-nextElem xs x y = is-nextElem ys x y ⟧ ⟹
  xs ≅ ys
apply(subgoal-tac length xs = length ys)
 prefer 2 apply(simp add:distinct-card[symmetric])
apply(clarsimp simp:congs-def list-eq-iff-nth-eq)
apply(cases ys) apply simp
apply(rename-tac a as)
apply simp
apply(subgoal-tac a ∈ set xs)
 prefer 2 apply blast
apply(drule in-set-conv-nth[THEN iffD1])
apply clarsimp
apply(rename-tac k)
apply(rule-tac x = k in exI)
apply(rule)
```

```
apply(rename-tac i)
apply(induct-tac i)
 using nth-rotate[of xs 0, symmetric]apply simp
apply clarsimp
apply(subgoal-tac is-nextElem (rotate k xs) ((xs ! k # as) ! n) (as ! n))
 apply(drule is-nextElem-nth1)
 apply (clarsimp simp: nth-eq-iff-index-eq)
apply(subgoal-tac is-nextElem (xs) ((xs ! k # as) ! n) (as ! n))
 apply simp
apply(subgoal-tac is-nextElem (xs ! k # as) ((xs ! k # as) ! n) (as ! n))
 apply simp
apply (rule is-nextElem-nth2)
apply(rule-tac x=n in exI)
apply(rule-tac x=Suc n in exI)
apply simp
done
```

12.6 *nextElem*, *sublist*, *is-nextElem*

```
lemma is-sublist-eq: distinct vs ⟹ c ≠ y ⟹
 (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 ⟹ 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 ⟹ 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 ⟹ x ∈ set vs ⟹ nextElem vs (hd vs) x = y
⟹ 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 = last vs ⟹ y = hd vs by (auto)
  from d fn have r3: hd vs ≠ y ⟹ (∃ a b. vs = a @ [x,y] @ b) by (drule-tac
nextElem-cases) auto

   from x obtain n where xn:x = vs!n and nl: n < length vs by (auto simp:
in-set-conv-nth)
  def as ≡ take n vs
  def bs ≡ drop (Suc n) vs
 from as-def bs-def xn nl have vs:vs = as @ [x] @ bs by (auto intro: id-take-nth-drop)
  then have r2: x ≠ last vs ⟹ y ≠ hd vs
  proof −
```

```
    assume notx: x ≠ last vs
    from vs notx have bs ≠ [] by auto
    with vs have r2: vs = as @ [x, hd bs] @ tl bs by auto
    with d have ineq: hd bs ≠ 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 ⟹ x ∈ set vs ⟹ is-nextElem vs x y ⟹ nextElem vs (hd vs) x = y
proof –
  assume d: distinct vs and x: x ∈ 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 ≠ y)  apply (simp add: is-sublist-eq) by auto
qed

lemma nextElem-is-nextElem:
 distinct xs ⟹ x ∈ set xs ⟹
   is-nextElem xs x y = (nextElem xs (hd xs) x = y)
  by (auto intro!: is-nextElem1 is-nextElem2)

lemma nextElem-congs-eq: xs ≅ ys ⟹ distinct xs ⟹ x ∈ set xs  ⟹
   nextElem xs (hd xs) x = nextElem ys (hd ys) x
proof –
    assume eq: xs ≅ ys and dist: distinct xs and x: x ∈ set xs
    def 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 nextElem-iso-eq:
     as ≅ as' ⟹ distinct as ⟹ x ∈ set as ⟹
     nextElem as (hd as) x = nextElem as' (hd as') x
 by (simp add: nextElem-congs-eq congs-def)


lemma is-sublist-is-nextElem: distinct vs ⟹ is-nextElem vs x y ⟹ is-sublist as vs ⟹ x ∈ set as ⟹ x ≠ last as ⟹ 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* ∈ *set as* **and** *xnl*: *x* ≠ *last as*
from *xin* **have** *asne*: *as* ≠ [] **by** *auto*
with *subl* **have** *vsne*: *vs* ≠ [] **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* ≠ *last vs*
proof (*cases ts* = [])
case *True* **with** *d xnl asne vs* **show** *?thesis* **by** *force*
next
def *lastvs* ≡ *last ts*
case *False*
with *vs lastvs-def* **have** *vs2*: *vs* = *rs* @ *as* @ *butlast ts* @ [*lastvs*] **by** *auto*
with *d* **have** *lastvs* ∉ *set as* **by** *auto*
with *xin* **have** *lastvs* ≠ *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** ¬ *is-sublist* [*x*] *rs* **by** *auto*
then have *nrs*: ¬ *is-sublist* [*x*,*y*] *rs* **by** (*auto dest*: *is-sublist-hd*)
from *d xin vs* **have** ¬ *is-sublist* [*x*] *ts* **by** *auto*
then have *nts*: ¬ *is-sublist* [*x*,*y*] *ts* **by** (*auto dest*: *is-sublist-hd*)
from *d xin vs* **have** *xnrs*: *x* ∉ *set rs* **by** *auto*
then have *notrs*: ¬ *is-sublist* [*x*,*y*] *rs* **by** (*auto simp*:*is-sublist-in*)
from *xnrs* **have** *xnlrs*: *rs* ≠ [] ⟹ *x* ≠ *last rs* **by** (*induct rs*) *auto*
from *d xin vs* **have** *xnts*: *x* ∉ *set ts* **by** *auto*
then have *notts*: ¬ *is-sublist* [*x*,*y*] *ts* **by** (*auto simp*:*is-sublist-in*)
from *d vs subl-vs* **have** *is-sublist* [*x*,*y*] *rs* ∨ *is-sublist* [*x*,*y*] (*as*@*ts*) **apply** (*cases rs* = []) **apply** *simp* **apply** (*rule-tac is-sublist-at1*) **by** (*auto intro*!: *xnlrs*)
with *notrs* **have** *is-sublist* [*x*,*y*] (*as*@*ts*) **by** *auto*
with *d vs xnl* **have** *is-sublist* [*x*,*y*] *as* ∨ *is-sublist* [*x*,*y*] *ts* **apply** (*rule-tac is-sublist-at1*) **by** *auto*
with *notts* **show** *is-sublist* [*x*,*y*] *as* **by** *auto*
qed

12.7 *before*

constdefs *before* :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *bool*
before vs ram1 ram2 ≡ ∃ *a b c*. *vs* = *a* @ *ram1* # *b* @ *ram2* # *c*

lemma *before-dist-fst-fst*[*simp*]: *before vs ram1 ram2* ⟹ *distinct vs* ⟹ *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* ⟹ *distinct vs* ⟹ *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* $\Longrightarrow$ *distinct vs* $\Longrightarrow$ *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* $\Longrightarrow$ *distinct vs* $\Longrightarrow$ *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* $\Longrightarrow$ *distinct vs* $\Longrightarrow$ *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* $\Longrightarrow$ *distinct vs* $\Longrightarrow$ *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* $\Longrightarrow$ *ram2* $\in$ *set vs* $\Longrightarrow$ *ram1* $\neq$ *ram2* $\Longrightarrow$ *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*)))
      def *a* $\equiv$ *fst* (*splitAt ram1 vs*)
      def *b* $\equiv$ *fst* (*splitAt ram2* (*snd* (*splitAt ram1 vs*)))
      def *c* $\equiv$ *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
      def *ab* $\equiv$ *fst* (*splitAt ram1 vs*)
      case *False* **with** *r1 r2 r12 ab-def* **have** *r2'*: *ram2* $\in$ *set ab* **by** (*auto intro*: *splitAt-ram3*)
      def *a* $\equiv$ *fst* (*splitAt ram2 ab*)
      def *b* $\equiv$ *snd* (*splitAt ram2 ab*)
      def *c* $\equiv$ *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 ⟹ r1 ∈ set vs by (auto simp: before-def)

lemma before-r2:
  before vs r1 r2 ⟹ r2 ∈ set vs by (auto simp: before-def)

lemma before-dist-r1r2:
  distinct vs ⟹ before vs r1 r2 ⟹ r1 ≠ r2
  by (auto simp: before-def)

lemma before-dist-r2:
  distinct vs ⟹ before vs r1 r2 ⟹ r2 ∈ set (snd (splitAt r1 vs))
proof −
  assume d: distinct vs and b: before vs r1 r2
   from d b have ex1: ∃! 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 ∀ y. vs = fst y @ r1 # snd y
⟶ y = s
      then have ⋀ y. vs = fst y @ r1 # snd y ⟶ y = s by auto
      then have single: ⋀ y. vs = fst y @ r1 # snd y ⟹ y = s by auto
      def bc ≡ b @ r2 # c
      with vs have vs2: vs = a @ r1 # bc by auto
      from bc-def have r2: r2 ∈ set bc by auto
      def 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 ⟹ before vs r1 r2 ⟹ r2 ∉ set (fst (splitAt r1 vs)) apply (frule
before-dist-r2) by (auto dest: splitAt-distinct-fst-snd)

lemma before-dist-r1:
  distinct vs ⟹ before vs r1 r2 ⟹ r1 ∈ set (fst (splitAt r2 vs))
proof −
  assume d: distinct vs and b: before vs r1 r2
   from d b have ex1: ∃! 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 ∀ y. vs = fst y @ r2 # snd y ⟶ y = s
      then have ⋀ y. vs = fst y @ r2 # snd y ⟶ y = s by auto
      then have single: ⋀ y. vs = fst y @ r2 # snd y ⟹ y = s by auto
      def ab ≡ a @ r1 # b
      with vs have vs2: vs = ab @ r2 # c by auto
      from ab-def have r1: r1 ∈ set ab by auto
      def 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 ⟹ before vs r1 r2 ⟹ r1 ∉ set (snd (splitAt r2 vs)) apply (frule before-dist-r1) by (auto dest: splitAt-distinct-fst-snd)

lemma before-snd:
  r2 ∈ set (snd (splitAt r1 vs)) ⟹ before vs r1 r2
proof -
  assume r2: r2 ∈ set (snd (splitAt r1 vs))
  from r2 have r1: r1 ∈ set vs apply (rule-tac ccontr) apply (drule splitAt-no-ram) by simp
  def a ≡ fst (splitAt r1 vs)
  def bc ≡ snd (splitAt r1 vs)
  def b ≡ fst (splitAt r2 bc)
  def 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 ∈ 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 ∈ set vs ⟹ r1 ∈ set (fst (splitAt r2 vs)) ⟹ before vs r1 r2
proof -
  assume r2: r2 ∈ set vs and r1: r1 ∈ set (fst (splitAt r2 vs))
  def ab ≡ fst (splitAt r2 vs)
  def c ≡ snd (splitAt r2 vs)
  def a ≡ fst (splitAt r1 ab)
  def 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* ∈ *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 ⟹ *r2* ∈ *set vs* ⟹ *r1* ∈ *set* (*fst* (*splitAt r2 vs*)) = *before vs r1 r2*
by (*auto intro*: *before-fst before-dist-r1*)

lemma *before-dist-eq-snd*:
distinct vs ⟹ *r2* ∈ *set* (*snd* (*splitAt r1 vs*)) = *before vs r1 r2*
by (*auto intro*: *before-snd before-dist-r2*)

lemma *pair-snd*: (*a*,*b*) = *p* ⟹ *snd p* = *b* **by** *auto*

lemma *before-dist-eq-snd'*:
distinct vs ⟹ (*as*, *bs*) = (*splitAt* ram_1 *vs*) ⟹
ram_2 ∈ *set bs* = *before vs* ram_1 ram_2
by (*simp add*: *before-dist-eq-snd*[*symmetric*] *pair-snd*)

lemma *before-dist-not1*:
distinct vs ⟹ *before vs ram1 ram2* ⟹ ¬ *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* ∈ *set vs* **by** (*drule-tac before-r1*)
from *d b1* **have** *r2*: *ram2* ∈ *set* (*fst* (*splitAt ram1 vs*)) **by** (*rule before-dist-r1*)
from *d b2* **have** *r2'*:*ram2* ∈ *set* (*snd* (*splitAt ram1 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 ⟹ *ram1* ∈ *set vs* ⟹ *ram2* ∈ *set vs* ⟹ *ram1* ≠ *ram2* ⟹ ¬ (*before vs ram1 ram2*) ⟹ *before vs ram2 ram1*
proof −
assume *distinct vs ram1* ∈ *set vs ram2* ∈ *set vs ram1* ≠ *ram2* ¬ *before vs ram1 ram2*
then show *before vs ram2 ram1* **apply** (*frule-tac before-or*) **by** *auto*
qed

lemma *before-dist-eq*:
distinct vs ⟹ *ram1* ∈ *set vs* ⟹ *ram2* ∈ *set vs* ⟹ *ram1* ≠ *ram2* ⟹ (¬ (*before vs ram1 ram2*)) = *before vs ram2 ram1*
by (*auto intro*: *before-dist-not2 dest*: *before-dist-not1*)

lemma *before-vs*:
distinct vs ⟹ *before vs ram1 ram2* ⟹ *vs* = *fst* (*splitAt ram1 vs*) @ *ram1* # *fst* (*splitAt ram2* (*snd* (*splitAt ram1 vs*))) @ *ram2* # *snd* (*splitAt ram2 vs*)

proof −
assume *d*: *distinct vs* **and** *b*: *before vs ram1 ram2*
def *s* ≡ *snd* (*splitAt ram1 vs*)
from *b* **have** *ram1* ∈ *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* ∈ *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

12.8 *between*

constdefs *pre-between* :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *bool*
pre-between vs ram1 ram2 ≡
distinct vs ∧ *ram1* ∈ *set vs* ∧ *ram2* ∈ *set vs* ∧ *ram1* ≠ *ram2*

declare *pre-between-def* [*simp*]

lemma *pre-between-dist*[*intro*]:
pre-between vs ram1 ram2 ⟹ *distinct vs* **by** (*auto simp*: *pre-between-def*)

lemma *pre-between-r1*[*intro*]:
pre-between vs ram1 ram2 ⟹ *ram1* ∈ *set vs* **by** *auto*

lemma *pre-between-r2*[*intro*]:
pre-between vs ram1 ram2 ⟹ *ram2* ∈ *set vs* **by** *auto*

lemma *pre-between-r12*[*intro*]:
pre-between vs ram1 ram2 ⟹ *ram1* ≠ *ram2* **by** *auto*

lemma *pre-between-sym*:
pre-between vs ram1 ram2 = *pre-between vs ram2 ram1* **by** (*auto simp*: *pre-between-def*)

lemma *pre-between-symI*:
pre-between vs ram1 ram2 ⟹ *pre-between vs ram2 ram1* **by** *auto*

lemma *pre-between-before*[*dest*]:
pre-between vs ram1 ram2 ⟹ *before vs ram1 ram2* ∨ *before vs ram2 ram1* **by** (*rule-tac before-or*) *auto*

lemma *pre-between-rotate1*[*intro*]:
pre-between vs ram1 ram2 ⟹ *pre-between* (*rotate1 vs*) *ram1 ram2* **by** *auto*

lemma *pre-between-rotate*[*intro*]:
pre-between vs ram1 ram2 ⟹ *pre-between* (*rotate n vs*) *ram1 ram2* **by** *auto*

```
lemma pre-between vs ram1 ram2 ⟹ (¬ 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 ⟹ pre-between vs ram1 ram2 ⟹
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 ⟹ pre-between vs ram1 ram2 ⟹
  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: ¬ before vs ram2 ram1 apply (simp add: pre-between-def)
by (auto dest: before-dist-not1)
  with p have ram2 ∉ set (fst (splitAt ram1 vs)) by (simp add: pre-between-def
before-dist-eq-fst)
  then have fst (splitAt ram1 vs) = fst (splitAt ram2 (fst (splitAt ram1 vs))) by
(auto dest: splitAt-no-ram)
  then have fst (splitAt ram2 (fst (splitAt ram1 vs))) = fst (splitAt ram1 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]:
  distinct vs ⟹ ram1 ∉ 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 ram1 ∉ set (between vs ram1 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: ¬ pre-between vs ram1 ram2 by auto
  then show ?thesis
  proof (cases ram1 = ram2)
    case True with d have h1:ram2 ∉ set (snd (splitAt ram2 vs)) by (auto del:
```

```
notI)
    from True d have h2: ram2 ∉ 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 ≠ ram2 by auto
    then show ?thesis
    proof (cases ram1 ∉ 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: ram1 ∈ set vs by auto
      then show ?thesis
      proof (cases ram2 ∉ set vs)
        from d have h1: ram1 ∉ set (fst (splitAt ram1 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: ram2 ∈ set 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 ⟹ ram2 ∉ 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 ∉ set (between vs ram1 ram2)
  proof (cases before vs ram1 ram2)
    from d have ram2 ∉ set (fst (splitAt ram2 vs)) by (auto del: notI)
    then have h1: ram2 ∉ 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: ¬ pre-between vs ram1 ram2 by auto
  then show ?thesis
  proof (cases ram1 = ram2)
```

```
    case True with d have h1:ram2 ∉ set (snd (splitAt ram2 vs)) by (auto del:
notI)
    from True d have h2: ram2 ∉ 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 ≠ ram2 by auto
    then show ?thesis
    proof (cases ram2 ∉ 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 ∈ set vs by auto
      then show ?thesis
      proof (cases ram1 ∉ set vs)
        from d have h1: ram1 ∉ 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 ∈ set 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-distinct[intro]:
  distinct vs ⟹ distinct (between vs ram1 ram2)
proof -
  assume vs: distinct vs
  def a: a ≡ fst (splitAt ram1 vs)
  def b: b ≡ snd (splitAt ram1 vs)
  from a b have ab: (a,b) = splitAt ram1 vs by auto
  with vs have ab-disj:set a ∩ set b = {} by (drule-tac splitAt-distinct-ab)  auto
  def c: c ≡ fst (splitAt ram2 a)
  def d: d ≡ snd (splitAt ram2 a)
  from c d have c-d: (c,d) = splitAt ram2 a by auto
  with ab-disj have set c ∩ set b = {} by (drule-tac splitAt-subset-ab) auto
  with vs a b c show ?thesis
     by (auto simp: between-def split-def splitAt-no-ram dest: splitAt-ram intro:
splitAt-distinct-fst splitAt-distinct-snd)
qed

lemma between-distinct-r12:
  distinct vs ⟹ ram1 ≠ ram2 ⟹ distinct (ram1 # between vs ram1 ram2 @
[ram2]) by (auto del: notI)
```

lemma *between-vs*:
before vs ram1 ram2 ⟹ *pre-between vs ram1 ram2* ⟹
vs = *fst* (*splitAt ram1 vs*) @ *ram1* # (*between vs ram1 ram2*) @ *ram2* # *snd* (*splitAt ram2 vs*)
apply (*simp*) **apply** (*frule pre-between-dist*) **apply** (*drule before-vs*) **by** *auto*

lemma *between-in*:
before vs ram1 ram2 ⟹ *pre-between vs ram1 ram2* ⟹ *x* ∈ *set vs* ⟹ *x* = *ram1* ∨ *x* ∈ *set* (*between vs ram1 ram2*) ∨ *x* = *ram2* ∨ *x* ∈ *set* (*between vs ram2 ram1*)
proof −
assume *b*: *before vs ram1 ram2* **and** *p*: *pre-between vs ram1 ram2* **and** *xin*: *x* ∈ *set vs*
def *a* ≡ *fst* (*splitAt ram1 vs*)
def *b* ≡ *between vs ram1 ram2*
def *c* ≡ *snd* (*splitAt ram2 vs*)
from *p* **have** *distinct vs* **by** *auto*
from *p b a-def b-def c-def* **have** *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* **apply** (*drule-tac between-vs*) **by** *auto*
with *xin* **have** *x* ∈ *set* (*a* @ *ram1* # *b* @ *ram2* # *c*) **by** *auto*
then have *x* ∈ *set* (*a*) ∨ *x* ∈ *set* (*ram1* #*b*) ∨ *x* ∈ *set* (*ram2* # *c*) **by** *auto*
then have *x* ∈ *set* (*a*) ∨ *x* = *ram1* ∨ *x* ∈ *set b* ∨ *x* = *ram2* ∨ *x* ∈ *set c* **by** *auto*
then have *x* ∈ *set c* ∨ *x* ∈ *set* (*a*) ∨ *x* = *ram1* ∨ *x* ∈ *set b* ∨ *x* = *ram2* **by** *auto*
then have *x* ∈ *set* (*c* @ *a*) ∨ *x* = *ram1* ∨ *x* ∈ *set b* ∨ *x* = *ram2* **by** *auto*
with *b p a-def b-def c-def* **show** *?thesis* **by** *auto*
qed

lemma
before vs ram1 ram2 ⟹ *pre-between vs ram1 ram2* ⟹
hd vs ≠ *ram1* ⟹ (*a*,*b*) = *splitAt* (*hd vs*) (*between vs ram2 ram1*) ⟹
vs = [*hd vs*] @ *b* @ [*ram1*] @ (*between vs ram1 ram2*) @ [*ram2*] @ *a*
proof −
assume *b*: *before vs ram1 ram2* **and** *p*: *pre-between vs ram1 ram2* **and** *vs*: *hd vs* ≠ *ram1* **and** *ab*: (*a*,*b*) = *splitAt* (*hd vs*) (*between vs ram2 ram1*)
from *p* **have** *dist-b*: *distinct* (*between vs ram2 ram1*) **by** (*auto intro*: *between-distinct simp*: *pre-between-def*)
with *ab* **have** *distinct a* ∧ *distinct b* **by** (*auto intro*: *splitAt-distinct-a splitAt-distinct-b*)
def *r* ≡ *snd* (*splitAt ram1 vs*)
def *btw* ≡ *between vs ram2 ram1*
from *p r-def* **have** *vs2*: *vs* = *fst* (*splitAt ram1 vs*) @ [*ram1*] @ *r* **by** (*auto dest*: *splitAt-ram simp*: *pre-between-def*)
then have *fst* (*splitAt ram1 vs*) = [] ⟹ *hd vs* = *ram1* **by** *auto*
with *vs* **have** *neq*: *fst* (*splitAt ram1 vs*) ≠ [] **by** *auto*
with *vs2* **have** *vs-fst*: *hd vs* = *hd* (*fst* (*splitAt ram1 vs*)) **by** (*induct fst* (*splitAt ram1 vs*)) *auto*
with *neq* **have** *hd vs* ∈ *set* (*fst* (*splitAt ram1 vs*)) **by** (*induct fst* (*splitAt ram1 vs*)) *auto*
with *b p* **have** *hd vs* ∈ *set* (*between vs ram2 ram1*) **by** *auto*

```
 with btw-def have help1: btw = fst (splitAt (hd vs) btw) @ [hd vs] @ snd
(splitAt (hd vs) btw) by (auto dest: splitAt-ram)
 from p b btw-def have btw = snd (splitAt ram2 vs) @ fst (splitAt ram1 vs) by
auto
 with neq have btw = snd (splitAt ram2 vs) @ hd (fst (splitAt ram1 vs)) # tl
(fst (splitAt ram1 vs)) by auto
 with vs-fst have btw = snd (splitAt ram2 vs) @ [hd vs] @ tl (fst (splitAt ram1
vs)) by auto
 with help1 have eq: snd (splitAt ram2 vs) @ [hd vs] @ tl (fst (splitAt ram1 vs))
= fst (splitAt (hd vs) btw) @ [hd vs] @ snd (splitAt (hd vs) btw) by auto
 from dist-b btw-def help1 have distinct (fst (splitAt (hd vs) btw) @ [hd vs] @
snd (splitAt (hd vs) btw)) by auto
 with eq have eq2: snd (splitAt ram2 vs) = fst (splitAt (hd vs) btw) ∧ tl (fst
(splitAt ram1 vs)) = snd (splitAt (hd vs) btw) apply (rule-tac dist-at) by auto
 with btw-def ab have a: a = snd (splitAt ram2 vs) by (auto dest: pairD)
 from eq2 vs-fst have hd (fst (splitAt ram1 vs)) # tl (fst (splitAt ram1 vs)) =
hd vs # snd (splitAt (hd vs) btw) by auto
 with ab btw-def neq have hdb: hd vs # b = fst (splitAt ram1 vs) by (auto dest:
pairD)

 from b p have vs = fst (splitAt ram1 vs) @ [ram1] @ fst (splitAt ram2 (snd
(splitAt ram1 vs))) @ [ram2] @ snd (splitAt ram2 vs) apply simp
   apply (rule-tac before-vs) by (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) by auto
 with a b p show ?thesis by (simp)
qed

lemma between1: ram1 ∈ set vs ⟹ ram2 ∈ set vs ⟹
 ram2 ∈ set (snd (splitAt ram1 vs)) ⟹
 vs = fst (splitAt ram1 vs) @ [ram1] @ (between vs ram1 ram2)
     @ [ram2] @ snd (splitAt ram2 (snd(splitAt ram1 vs)))
 apply (simp add: between-def splitAt-ram split-def) by (auto dest!: splitAt-ram)

lemma between2: ram1 ∈ set vs ⟹ ram2 ∈ set vs ⟹ ram1 ≠ ram2 ⟹
 ram2 ∉ set (snd (splitAt ram1 vs)) ⟹
 vs @ vs = fst (splitAt ram1 vs) @ [ram1] @ (between vs ram1 ram2)
        @ [ram2] @ snd (splitAt ram2 (fst(splitAt ram1 vs))) @ [ram1]
        @ snd (splitAt ram1 vs)
proof -
 assume r1: ram1 ∈ set vs and r2: ram2 ∈ set vs and r12: ram1 ≠ ram2 and
rnot2: ram2 ∉ set (snd (splitAt ram1 vs))
 then have r2in: ram2 ∈ set (fst (splitAt ram1 vs))
  by (auto dest!: splitAt-ram2)
 then have xx: fst (splitAt ram2 (fst (splitAt ram1 vs))) @ ram2 # snd (splitAt
ram2 (fst (splitAt ram1 vs))) = fst (splitAt ram1 vs) by (auto dest!: splitAt-ram)
 from r1 have vs = fst (splitAt ram1 vs) @ ram1 # snd (splitAt ram1 vs) by
(auto dest!: splitAt-ram)
 with r2in have vs = (fst (splitAt ram2 (fst(splitAt ram1 vs))) @ ram2 # (snd
```

```
(splitAt ram2 (fst (splitAt ram1 vs))))) @ ram1 # snd (splitAt ram1 vs) by (simp
add: xx)
  with r1 rnot2 show ?thesis
  by (simp add: between-def splitAt-ram split-def)
qed


lemma between-congs: pre-between vs ram1 ram2 ⟹ vs ≅ vs' ⟹ between vs
ram1 ram2 = between vs' ram1 ram2
proof −
  have ⋀ us. pre-between us ram1 ram2 ⟹ before us ram1 ram2 ⟹ between us
ram1 ram2 = between (rotate1 us) ram1 ram2
  proof −
    fix us
    assume vors: pre-between us ram1 ram2  before us ram1 ram2
    then have pb2: pre-between (rotate1 us) ram1 ram2 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 ⋀ us. pre-between us ram1 ram2 ⟹ before us ram2 ram1 ⟹
between us ram1 ram2 = between (rotate1 us) ram1 ram2
  proof −
    fix us
    assume vors:  pre-between us ram1 ram2  before us ram2 ram1
    then have pb2: pre-between (rotate1 us) ram1 ram2 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: ⋀ us. pre-between us ram1 ram2 ⟹ between us ram1
ram2 = between (rotate1 us) ram1 ram2
    apply (subgoal-tac before us ram1 ram2 ∨ before us ram2 ram1) by auto

  assume vs ≅ vs' and pre-b: pre-between vs ram1 ram2
```

```
  then obtain n where vs': vs' = rotate n vs by (auto simp: congs-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 ⟹
  set (between vs ram1 ram2) ∩ 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)
```

12.8.1 *between is-nextElem*

```
lemma is-nextElem-or1: pre-between vs ram1 ram2 ⟹
```

```
  is-nextElem vs x y ⟹ before vs ram1 ram2 ⟹
  is-sublist [x,y] (ram1 # 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)
  def 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
    def 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
  def 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
    def cs1 ≡ ram2 # snd (splitAt ram2 vs)
    then have cs1ne: cs1 ≠ [] by auto
    def cs2 ≡ fst (splitAt ram1 vs)
    def 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 ⟹ 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 ≠ [] 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 ⟹ 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 ∈ set cs1 ⟹ is-sublist [x,y] bs2 by auto
    have x ∈ set cs2 ⟹ is-sublist [x,y] (cs2 @ [ram1])
    proof-
      assume xin2': x ∈ set cs2
      then have xin2: x ∈ set (cs2 @ [ram1]) by auto
      def 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 ≠ ram1 by (auto simp: pre-between-def)
      then have xnlcs2: x ≠ 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 xnlcs2 xin2 show ?thesis apply (rule-tac is-sublist-is-nextElem) by (auto simp: pre-between-def)
    qed
    with bs2 have x ∈ set cs2 ⟹ 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 ∈ set vs by auto
  with b p have x = ram1 ∨ x ∈ set (between vs ram1 ram2) ∨ x = ram2 ∨ x ∈ set (between vs ram2 ram1) by (rule-tac between-in) auto
```

then have *x* ∈ *set* (*ram1* # (*between vs ram1 ram2*)) ∨ *x* ∈ *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* ⟹ *is-nextElem vs x y* ⟹
is-sublist [*x*,*y*] (*ram1* # *between vs ram1 ram2* @ [*ram2*]) ∨ *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** *assumption+* **by** *auto*
qed

declare *distinct-append distinct.simps* [*simp del*]

lemma *pre-between vs ram1 ram2* ⟹ *is-nextElem vs x y* ⟹ *before vs ram1 ram2* ⟹
(*is-sublist* [*x*,*y*] (*ram1* # *between vs ram1 ram2* @ [*ram2*])
= (¬ *is-sublist* [*x*,*y*] (*ram2* # *between vs ram2 ram1* @ [*ram1*])))
apply (*rule iffI*)

apply (*subgoal-tac distinct* (*ram1* # *between vs ram1 ram2* @ [*ram2*]))
apply (*subgoal-tac distinct* (*ram2* # *between vs ram2 ram1* @ [*ram1*]))
defer
apply (*rule between-distinct-r12*) **apply** (*simp add*: *pre-between-def*) **apply** (*simp add*: *pre-between-def*) **apply** *force*
apply (*rule between-distinct-r12*) **apply** (*simp add*: *pre-between-def*) **apply** (*simp add*: *pre-between-def*)
apply (*drule is-nextElem-or*) **apply** *simp* **apply** *simp*

apply (*subgoal-tac x* ∈ *set vs* ∧ *y* ∈ *set vs*)
defer
apply *force*

apply (*elim conjE*) **apply** (*frule between-in*) **apply** *simp* **apply** *assumption*
apply (*subgoal-tac distinct* (*fst* (*splitAt ram1 vs*) @ *ram1* # *between vs ram1 ram2* @ *ram2* # *snd* (*splitAt ram2 vs*)))
defer
apply (*subgoal-tac fst* (*splitAt ram1 vs*) @ *ram1* # *between vs ram1 ram2* @ *ram2* # *snd* (*splitAt ram2 vs*) = *vs*)
apply (*simp add*: *pre-between-def between-vs*) **apply** (*rule sym*) **apply** (*rule*

between-vs) **apply** *simp* **apply** *simp*

apply (*case-tac x* = *ram1*) **apply** *simp*
apply (*case-tac x* = *ram2*) **apply** (*rule ccontr*) **apply** (*simp add*: *distinct-append distinct.simps*)
apply (*case-tac x* ∈ *set* (*between vs ram1 ram2*))
apply (*thin-tac distinct* (*ram1* # *between vs ram1 ram2* @ [*ram2*])) **apply** (*thin-tac distinct* (*ram2* # *between vs ram2 ram1* @ [*ram1*]))
apply (*rule ccontr*) **apply** (*case-tac x* ∈ *set* (*snd* (*splitAt ram2 vs*)))
apply (*subgoal-tac x* ∈ *set* (*between vs ram1 ram2*) ∩ *set* (*snd* (*splitAt ram2 vs*)))
apply (*simp add*: *distinct-append distinct.simps*) **apply** *simp*
apply (*subgoal-tac x* ∈ *set* (*fst* (*splitAt ram1 vs*))) **apply** (*simp add*: *distinct-append distinct.simps*)

apply (*subgoal-tac x* ∈ *set* (*fst* (*splitAt ram1 vs*)) ∩ (*set* (*fst* (*splitAt ram2* (*snd* (*splitAt ram1 vs*)))) ∪ *set* (*snd* (*splitAt ram2 vs*))))
apply (*simp*) **apply** (*rule IntI*) **apply** *simp* **apply** *simp* **apply** (*rule ccontr*) **apply** (*simp add*: *distinct-append distinct.simps*)
apply (*subgoal-tac x* ∈ *set* (*snd* (*splitAt ram2 vs*) @ *fst* (*splitAt ram1 vs*) @ [*ram1*])) **apply** *simp* **apply** (*rule is-sublist-in*) **apply** *assumption*
apply (*subgoal-tac x* ∈ *set* (*between vs ram2 ram1*))
apply (*subgoal-tac x* ∈ *set* ((*ram1* # *between vs ram1 ram2*) @ [*ram2*])) **apply** *simp*
apply (*rule is-sublist-in*) **apply** *simp* **by** *simp*

declare *distinct-append distinct.simps* [*simp*]

lemma *pre-between vs ram1 ram2* ⟹ *is-nextElem vs x y* ⟹
(*is-sublist* [*x*,*y*] (*ram1* # *between vs ram1 ram2* @ [*ram2*])
= (¬ *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 between-xor1*)
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 between-xor1*) **apply** *assumption*+ **by** *auto*
qed

lemma *pre-between vs ram1 ram2* ⟹
before vs ram1 ram2 ⟹
∃ *as bs*. *vs* = *as* @[*ram1*] @ *between vs ram1 ram2* @ [*ram2*] @ *bs*
apply *auto* **apply** (*intro exI*) **apply** (*simp add*: *pre-between-def*) **apply** *auto* **apply** (*frule-tac before-vs*) **by** *auto*

lemma *pre-between vs ram1 ram2* $\Longrightarrow$
before vs ram2 ram1 $\Longrightarrow$
$\exists$ *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* $\Longrightarrow$ *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 $\Longrightarrow$ *is-nextElem vs a b* $\Longrightarrow$ *between vs a b* = []
apply (*simp add*: *is-nextElem-def between-def split-def*)
apply (*cases vs*) **apply** *simp*+
apply (*case-tac list*)
apply (*simp add*: *is-sublist-def*)
apply (*erule disjE*)
apply (*elim exE*) **apply** (*case-tac as*) **apply** *simp*+
apply (*simp split*: *split-if-asm*)
apply (*case-tac a*=*aaa*)
apply *simp*
apply (*rule conjI*)
apply (*rule impI*) **apply** *simp*
apply *simp*
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 (*case-tac listb*)
apply *simp*
apply *simp*

```
 apply simp
 apply (case-tac lista rule: rev-exhaust)
  apply simp
 apply simp
apply simp
apply (subgoal-tac aa # aaa # lista = vs)
 apply (thin-tac vs = aa # aaa # lista)
 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 = [] ⟹ distinct vs ⟹ a ∈
set vs ⟹ b ∈ set vs  ⟹
  a ≠ b ⟹ is-nextElem vs a b
apply (simp add: is-nextElem-def between-def split-def split: split-if-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 ⟹
 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(fastsimp 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(fastsimp simp:between-def split-def)
done
```

```
lemma nextVertices-in-face[simp]: v ∈ 𝒱 f ⟹ f^n · v ∈ 𝒱 f
proof -
  assume v: v ∈ 𝒱 f
  then have f: vertices f ≠ [] by auto
  show ?thesis apply (simp add: nextVertices-def)
    apply (induct n) by (auto simp: f v)
```

qed

12.8.2 *is-nextElem edges* equivalence

lemma *is-nextElem-edges1*: *distinct* (*vertices f*) ⟹ (*a*,*b*) ∈ *edges* (*f*::*face*) ⟹ *is-nextElem* (*vertices f*) *a b* **apply** (*simp add*: *edges-face-def nextVertex-def*) **apply** (*rule is-nextElem1*) **by** *auto*

lemma *is-nextElem-edges2*:
distinct (*vertices f*) ⟹ *is-nextElem* (*vertices f*) *a b* ⟹
(*a*,*b*) ∈ *edges* (*f*::*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*]:
distinct (*vertices* (*f*::*face*)) ⟹
(*a*,*b*) ∈ *edges f* = *is-nextElem* (*vertices f*) *a b*
by (*auto intro*: *is-nextElem-edges1 is-nextElem-edges2*)

12.8.3 *nextVertex*

lemma *nextElem-suc2*: *distinct* (*vertices f*) ⟹ *last* (*vertices f*) = *v* ⟹ *v* ∈ *set* (*vertices f*) ⟹ *f* · *v* = *hd* (*vertices f*)
proof −
assume *dist*: *distinct* (*vertices f*) **and** *last*: *last* (*vertices f*) = *v* **and** *v*: *v* ∈ *set* (*vertices f*)
def *ls* ≡ *vertices f*
have *ind*: ⋀ *c*. *distinct ls* ⟹ *last ls* = *v* ⟹ *v* ∈ *set ls* ⟹ *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* ∈ *set ms* **by** *auto*
then have *ms*: *ms* ≠ [] **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

lemma *nextVertex-congs-eq*: $f_1 \cong f_2 \Longrightarrow$ *distinct* (*vertices* f_1) $\Longrightarrow v \in \mathcal{V}\, f_1 \Longrightarrow$ $f_1 \cdot v = f_2 \cdot v$
by (*simp add*: *cong-face-def nextVertex-def*)
(*rule nextElem-congs-eq*)

12.9 *split-face*

constdefs *pre-split-face* :: *face* ⇒ *nat* ⇒ *nat* ⇒ *nat list* ⇒ *bool*
pre-split-face oldF ram1 ram2 newVertexList ≡
distinct (*vertices oldF*) ∧ *distinct* (*newVertexList*)
∧ $\mathcal{V}$ *oldF* ∩ *set newVertexList* = {}
∧ *ram1* ∈ $\mathcal{V}$ *oldF* ∧ *ram2* ∈ $\mathcal{V}$ *oldF* ∧ *ram1* ≠ *ram2*

declare *pre-split-face-def* [*simp*]

lemma *pre-split-face-p-between*[*intro*]:
pre-split-face oldF ram1 ram2 newVertexList ⟹ *pre-between* (*vertices oldF*) *ram1 ram2* **by** (*simp add*: *pre-between-def*)

lemma *pre-split-face-symI*:
pre-split-face oldF ram1 ram2 newVertexList ⟹ *pre-split-face oldF ram2 ram1 newVertexList* **by** *auto*

lemma *pre-split-face-rev*[*intro*]:
pre-split-face oldF ram1 ram2 newVertexList ⟹ *pre-split-face oldF ram1 ram2* (*rev newVertexList*) **by** *auto*

lemma *split-face-distinct1*:
(*f12*, *f21*) = *split-face oldF ram1 ram2 newVertexList* ⟹ *pre-split-face oldF ram1 ram2 newVertexList* ⟹
distinct (*vertices f12*)
proof −
assume *split*: (*f12*, *f21*) = *split-face oldF ram1 ram2 newVertexList* **and** *p*: *pre-split-face oldF ram1 ram2 newVertexList*
def *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* ⟹ *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 ⟹ 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 ⟹
    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 ⟹
   pre-split-face oldF ram1 ram2 newVertexList ⟹ distinct (vertices f21)
proof −
  assume split: (f12, f21) = split-face oldF ram1 ram2 newVertexList
    and p: pre-split-face oldF ram1 ram2 newVertexList
  def 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 ⟹ distinct (vertices f21)
```

```
  proof -
    assume b: before old-vs ram1 ram2
    with split p
     have f21 = Face (ram2 # between (vertices oldF) 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]) ∩ set newVertexList = {}
      by auto
    with h1 d1 p1 show ?thesis by auto
  qed
  have rule2: before old-vs ram2 ram1 ⟹ 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 = Face (ram2 # between (vertices oldF) 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]) ∩ 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 ⟹ 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*) = *split-face oldF ram1 ram2 newVertexList* ⟹ *pre-split-face oldF ram1 ram2 newVertexList* ⟹ (*a*, *b*) ∈ *edges oldF* ⟹ (*a*,*b*) ∈ *edges f12* ∨ (*a*,*b*) ∈ *edges f21*
proof −
 assume *nf*: (*f12*, *f21*) = *split-face oldF ram1 ram2 newVertexList* **and** *p*: *pre-split-face oldF ram1 ram2 newVertexList* **and** *old*:(*a*, *b*) ∈ *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*]) ∨ *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*
 ∨ *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*])
 ∨ *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

lemma *is-nextElem-hd-last*: *distinct vs* ⟹ *is-nextElem vs x y* ⟹ *y* = *hd vs* ⟹ *x* = *last vs*
apply (*cases vs*) **apply** (*simp add*: *is-nextElem-def is-sublist-def*) **apply** *simp*
apply *auto* **apply** (*auto simp*: *is-nextElem-def is-sublist-def*)
apply (*case-tac as*) **apply** *simp*+ **apply** (*case-tac as*) **by** *simp*+

lemma *split-face-xor*: (*f12*, *f21*) = *split-face oldF ram1 ram2 newVertexList* ⟹
 pre-split-face oldF ram1 ram2 newVertexList ⟹
 (*ram1*,*ram2*) ∉ *edges oldF* ⟹ (*ram2*,*ram1*) ∉ *edges oldF* ⟹
 (*a*, *b*) ∈ *edges oldF* ⟹ (*a*,*b*) ∈ *edges f12* = ((*a*,*b*) ∉ *edges f21*)
proof

assume *split*: (*f12*, *f21*) = *split-face oldF ram1 ram2 newVertexList* **and**
pre-fdg: *pre-split-face oldF ram1 ram2 newVertexList* **and**
oldF: (*ram1*, *ram2*) ∉ *edges oldF* (*ram2*, *ram1*) ∉ *edges oldF* (*a*, *b*) ∈ *edges oldF* **and**
f12-edge: (*a*, *b*) ∈ *edges f12*
from *pre-fdg split oldF*
have *oldF′*: *is-nextElem* (*vertices oldF*) *a b*
by (*auto simp*: *pre-split-face-def*)

from *pre-fdg split oldF*
have ¬ *is-nextElem* (*vertices oldF*) *ram1 ram2* ∧ ¬ *is-nextElem* (*vertices oldF*) *ram2 ram1*
by (*auto simp*: *pre-split-face-def split-face-distinct1 split-face-distinct2*)
then have *oldF-ram*: ¬ *is-nextElem* (*vertices oldF*) *ram1 ram2* ¬ *is-nextElem* (*vertices oldF*) *ram2 ram1* **by** *auto*

from *pre-fdg split f12-edge* **have** *f12-edge′*: *is-nextElem* (*vertices f12*) *a b* **by** (*simp add*: *split-face-distinct1*)

from *pre-fdg oldF′* **have** *a′*: *a* ∉ *set newVertexList* **by** (*auto simp*: *pre-split-face-def*)
from *pre-fdg oldF′* **have** *b′*: *b* ∉ *set newVertexList* **by** (*auto simp*: *pre-split-face-def*)

from *a′* **have** *rule1*: ¬ *is-sublist* [*a*, *b*] (*newVertexList*) **by** (*auto simp*:*is-sublist-in*)
from *a′* **have** *rule2*: ¬ *is-sublist* [*a*, *b*] (*rev newVertexList*) **apply** (*rule-tac ccontr*) **apply** *simp* **apply** (*drule is-sublist-in*) **by** *simp*

from *pre-fdg split* **have** *f21*: *f21* = *Face* (*ram2* # *between* (*vertices oldF*) *ram2 ram1* @ *ram1* # *newVertexList*) *Nonfinal*
by (*simp add*: *split-face-def*)
from *pre-fdg split* **have** *f12*: *f12* = *Face* (*rev newVertexList* @ *ram1* # *between* (*vertices oldF*) *ram1 ram2* @ [*ram2*]) *Nonfinal*
by (*simp add*: *split-face-def*)

from *f12-edge′ f12* **have** *a* ≠ *ram2* ⟹ *is-sublist* [*a*, *b*] (*vertices f12*)
by (*simp add*: *is-nextElem-def*)
with *pre-fdg split f12 rule2 a′* **have** *a* ≠ *ram2* ⟹ *is-sublist* [*a*,*b*] (*ram1* # *between* (*vertices oldF*) *ram1 ram2* @ [*ram2*])
apply (*subgoal-tac distinct* (*rev newVertexList* @ *ram1* # *between* (*vertices oldF*) *ram1 ram2* @ [*ram2*]))
prefer *2* **apply** (*drule* (*1*) *split-face-distinct1*) **apply** *simp*
apply (*drule is-sublist-at5*) **apply** *simp*
apply (*case-tac newVertexList*) **by** *auto*
with *pre-fdg oldF′* **have** *a* ≠ *ram2* ⟹ ¬ *is-sublist* [*a*,*b*] (*ram2* # *between* (*vertices oldF*) *ram2 ram1* @ [*ram1*])
apply (*subgoal-tac pre-between* (*vertices oldF*) *ram1 ram2*) **apply** (*drule between-xor*) **by** *auto*
with *pre-fdg split f21 rule1 f12-edge′ b′* **have** *rule-a*: *a* ≠ *ram2* ⟹ ¬ *is-sublist*

[*a*, *b*] (*vertices f21*) **apply** (*rule-tac ccontr*) **apply** *simp*

apply (*subgoal-tac distinct* ((*between* (*vertices oldF*) *ram2 ram1* @ [*ram1*]) @ *newVertexList*))

apply (*drule is-sublist-at5*) **apply** *simp* **apply** *simp*

apply (*case-tac newVertexList*) **apply** *simp* **apply** (*elim conjE*) **apply** *simp*

apply (*drule split-face-distinct2*) **by** *auto*

have *help*: $\bigwedge$ *A B C*. $C \cap (A \cup B) = \{\} \Longrightarrow B \cap C = \{\}$ **by** *auto*

from *pre-fdg* **have** *before* (*vertices oldF*) *ram1 ram2* $\vee$ *before* (*vertices oldF*) *ram2 ram1*

apply (*rule-tac before-or*) **by** (*auto simp*: *pre-split-face-def*)

with *pre-fdg oldF-ram* **have** *rule3*: $\neg$ *is-sublist* [*ram2*, *ram1*] (*ram2* # *between* (*vertices oldF*) *ram2 ram1* @ [*ram1*])

apply (*elim disjE*) **apply** (*frule between-vs*) **apply** *force*

apply (*subgoal-tac pre-between* (*vertices oldF*) *ram1 ram2*) **apply** (*rule ccontr*)

apply (*subgoal-tac between* (*vertices oldF*) *ram2 ram1* = [])

apply (*simp add*: *is-nextElem-def*) **apply** (*elim conjE*) **apply** (*case-tac* (*vertices oldF*) *rule*: *rev-exhaust*) **apply** *simp* **apply** *simp*

apply (*case-tac ys*) **apply** *simp* **apply** *simp* **apply** *simp*

apply (*case-tac snd* (*splitAt ram2* (*vertices oldF*))) **apply** *simp*

apply (*case-tac fst* (*splitAt ram1* (*vertices oldF*))) **apply** *simp* **apply** *simp* **apply** (*simp add*: *pre-split-face-def*)

apply (*subgoal-tac distinct* (*ram2* # *a* # *list* @ [*ram1*])) **apply** (*rotate-tac* $-$*1*) **apply** (*frule is-sublist-distinct-prefix*)

apply *simp* **apply** *simp* **apply** (*case-tac snd* (*splitAt ram1* (*fst* (*splitAt ram2* (*vertices oldF*))))) **apply** *force* **apply** *force*

apply *simp*

apply (*subgoal-tac distinct* (*ram2* # *a* # *list* @ *fst* (*splitAt ram1* (*vertices oldF*)) @ [*ram1*])) **apply** (*rotate-tac* $-$*1*) **apply** (*frule is-sublist-distinct-prefix*)

apply *simp* **apply** *simp*

apply (*subgoal-tac distinct* (*fst* (*splitAt ram1* (*vertices oldF*)) @ *ram1* # *fst* (*splitAt ram2* (*snd* (*splitAt ram1* (*vertices oldF*)))) @ *ram2* # *a* # *list*))

apply (*thin-tac vertices oldF* =
fst (*splitAt ram1* (*vertices oldF*)) @ *ram1* # *fst* (*splitAt ram2* (*snd* (*splitAt ram1* (*vertices oldF*)))) @ *ram2* # *a* # *list*)

apply *simp* **apply** (*elim conjE*) **apply** (*rule help*) **apply** *simp*

apply (*simp add*: *pre-split-face-def*)

apply *force*

apply (*frule between-vs*) **apply** (*simp add*: *pre-between-def pre-split-face-def*)

apply (*subgoal-tac pre-between* (*vertices oldF*) *ram2 ram1*) **apply** (*rule ccontr*) **apply** *simp*

apply (*subgoal-tac fst* (*splitAt ram1* (*snd* (*splitAt ram2* (*vertices oldF*)))) = [])

apply (*simp add*: *is-nextElem-def*) **apply** (*elim conjE*) **apply** (*simp add*: *is-sublist-def*)

apply (*subgoal-tac distinct* (*ram2* # *fst* (*splitAt ram1* (*snd* (*splitAt ram2* (*vertices oldF*)))) @ [*ram1*]))

apply (*rotate-tac* $-$*1*) **apply** (*frule is-sublist-distinct-prefix*) **apply** *simp*

apply (*case-tac fst* (*splitAt ram1* (*snd* (*splitAt ram2* (*vertices oldF*))))) **apply** *simp* **apply** *simp*

apply (*subgoal-tac distinct* (*fst* (*splitAt ram2* (*vertices oldF*)) @
ram2 # *fst* (*splitAt ram1* (*snd* (*splitAt ram2* (*vertices oldF*)))) @ *ram1* # *snd* (*splitAt ram1* (*vertices oldF*))))
apply (*thin-tac vertices oldF* =
fst (*splitAt ram2* (*vertices oldF*)) @
ram2 # *fst* (*splitAt ram1* (*snd* (*splitAt ram2* (*vertices oldF*)))) @ *ram1* # *snd* (*splitAt ram1* (*vertices oldF*)))
apply *force* **apply** (*simp add*: *pre-split-face-def*) **apply** (*rule pre-between-symI*) **by** *force*

from *pre-fdg* **have** *before* (*vertices oldF*) *ram2 ram1* ∨ *before* (*vertices oldF*) *ram1 ram2*
apply (*rule-tac before-or*) **by** (*auto simp*: *pre-split-face-def*)
with *pre-fdg oldF-ram* **have** *rule4*: ¬ *is-sublist* [*ram1*, *ram2*] (*ram1* # *between* (*vertices oldF*) *ram1 ram2* @ [*ram2*])
apply (*elim disjE*) **apply** (*frule between-vs*) **apply** (*simp add*: *pre-split-face-def pre-between-def*)
apply (*subgoal-tac pre-between* (*vertices oldF*) *ram2 ram1*) **apply** (*rule ccontr*)
apply (*subgoal-tac between* (*vertices oldF*) *ram1 ram2* = [])
apply (*simp add*: *is-nextElem-def*) **apply** (*elim conjE*) **apply** (*case-tac* (*vertices oldF*) *rule*: *rev-exhaust*) **apply** *simp* **apply** *simp*
apply (*case-tac ys*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac snd* (*splitAt ram1* (*vertices oldF*))) **apply** *simp*
apply (*case-tac fst* (*splitAt ram2* (*vertices oldF*))) **apply** *simp* **apply** *simp* **apply** (*simp add*: *pre-split-face-def*)
apply (*subgoal-tac distinct* (*ram1* # *a* # *list* @ [*ram2*])) **apply** (*rotate-tac* −*1*) **apply** (*frule is-sublist-distinct-prefix*)
apply *simp* **apply** *simp* **apply** (*case-tac snd* (*splitAt ram2* (*fst* (*splitAt ram1* (*vertices oldF*))))) **apply** *force* **apply** *bestsimp*
apply *simp*
apply (*subgoal-tac distinct* (*ram1* # *a* # *list* @ *fst* (*splitAt ram2* (*vertices oldF*)) @ [*ram2*])) **apply** (*rotate-tac* −*1*) **apply** (*frule is-sublist-distinct-prefix*)
apply *simp* **apply** *simp*
apply (*subgoal-tac distinct* (*fst* (*splitAt ram2* (*vertices oldF*)) @ *ram2* # *fst* (*splitAt ram1* (*snd* (*splitAt ram2* (*vertices oldF*)))) @ *ram1* # *a* # *list*))
apply (*thin-tac vertices oldF* =
fst (*splitAt ram2* (*vertices oldF*)) @ *ram2* # *fst* (*splitAt ram1* (*snd* (*splitAt ram2* (*vertices oldF*)))) @ *ram1* # *a* # *list*)
apply *simp* **apply** (*elim conjE*) **apply** (*rule help*) **apply** *simp*
apply (*simp add*: *pre-split-face-def*) **apply** (*simp add*: *pre-split-face-def pre-between-def*)

apply (*frule between-vs*) **apply** *force*
apply (*subgoal-tac pre-between* (*vertices oldF*) *ram1 ram2*) **apply** (*rule ccontr*) **apply** *simp*
apply (*subgoal-tac fst* (*splitAt ram2* (*snd* (*splitAt ram1* (*vertices oldF*)))) = [])

```
     apply (simp  add: is-nextElem-def) apply (elim conjE) apply (simp add:
is-sublist-def)
     apply (subgoal-tac  distinct (ram1 # fst (splitAt ram2 (snd (splitAt ram1
(vertices oldF)))) @ [ram2]))
    apply (rotate-tac −1) apply (frule is-sublist-distinct-prefix) apply simp
    apply (case-tac fst (splitAt ram2 (snd (splitAt ram1 (vertices oldF))))) apply
simp apply simp

    apply (subgoal-tac distinct (fst (splitAt ram1 (vertices oldF)) @
     ram1 # fst (splitAt ram2 (snd (splitAt ram1 (vertices oldF)))) @ ram2 # snd
(splitAt ram2 (vertices oldF))))
    apply (thin-tac vertices oldF =
     fst (splitAt ram1 (vertices oldF)) @
     ram1 # fst (splitAt ram2 (snd (splitAt ram1 (vertices oldF)))) @ ram2 # snd
(splitAt ram2 (vertices oldF)))
    apply force apply (simp add: pre-split-face-def) by force


  from pre-fdg f12-edge′ split have ¬ is-nextElem (vertices f21) a b
    apply (rule-tac ccontr) apply simp
    apply (simp add: is-nextElem-def)
    apply (case-tac a = ram2)
     apply (simp  add: f12 f21)
     apply (case-tac newVertexList rule:rev-exhaust)
      apply (subgoal-tac ram2 ≠ ram1)
       apply simp
       apply (case-tac b = ram1) apply (simp add: rule3) apply simp
       apply (subgoal-tac distinct (between (vertices oldF) ram1 ram2 @ [ram2]))
        apply (rotate-tac −1)
         apply (drule is-sublist-x-last) apply simp apply force
        apply (drule split-face-distinct1) apply simp
        apply simp
       apply (force simp: pre-split-face-def)
      apply simp
      apply (case-tac ram2 = y) apply (simp add: pre-split-face-def) apply force
      apply (case-tac b = y)
       apply (subgoal-tac b ∉ set (newVertexList)) apply simp
       apply (rule b′)
      apply simp
      apply (subgoal-tac distinct (rev ys @ ram1 # between (vertices oldF) ram1
ram2 @ [ram2]))
       apply (drule  is-sublist-x-last) apply simp apply force
      apply (drule split-face-distinct1) apply simp
      apply simp
     apply (frule rule-a)  apply (simp add: f12 f21)
     apply (case-tac newVertexList rule:rev-exhaust)
      apply simp
      apply (case-tac ram1 = ram2)  apply (simp only: pre-split-face-def) apply
(elim conjE)  apply (simp only:)  apply simp
```

apply (*simp add*: *rule4*)
apply *simp*
apply (*subgoal-tac distinct*(*y* # *rev ys* @ *ram1* # *between* (*vertices oldF*) *ram1 ram2* @ [*ram2*]))
prefer *2* **apply** (*drule split-face-distinct1*) **apply** *simp* **apply** *simp*
apply (*frule is-sublist-distinct-prefix*) **apply** *simp*
apply (*case-tac ys rule*:*rev-exhaust*) **by** *simp-all*

with *pre-fdg split* **show** (*a*,*b*) ∉ *edges f21* **by** (*simp add*: *split-face-distinct2*)
next
assume (*f12*, *f21*) = *split-face oldF ram1 ram2 newVertexList pre-split-face oldF ram1 ram2 newVertexList*
(*ram1*, *ram2*) ∉ *edges oldF* (*ram2*, *ram1*) ∉ *edges oldF* (*a*, *b*) ∈ *edges oldF* (*a*, *b*) ∉ *edges f21*
then show (*a*,*b*) ∈ *edges f12* **apply** (*drule-tac split-face-edges-or*) **by** *auto*
qed

12.10 *verticesFrom*

constdefs *verticesFrom* :: *face* ⇒ *vertex* ⇒ *vertex list*
verticesFrom f ≡ *rotate-to* (*vertices f*)

lemmas *verticesFrom-Def* = *verticesFrom-def rotate-to-def*

lemma *len-vFrom*[*simp*]:
v ∈ 𝒱 *f* ⟹ |*verticesFrom f v*| = |*vertices f*|
apply(*drule split-list-first*)
apply(*clarsimp simp*: *verticesFrom-Def*)
done

lemma *verticesFrom-empty*[*simp*]:
v ∈ 𝒱 *f* ⟹ (*verticesFrom f v* = []) = (*vertices f* = [])
by(*clarsimp simp*: *verticesFrom-Def*)

lemma *verticesFrom-congs*:
v ∈ 𝒱 *f* ⟹ (*vertices f*) ≅ (*verticesFrom f v*)
by(*simp add*:*verticesFrom-def cong-rotate-to*)

lemma *verticesFrom-eq-if-vertices-cong*:
⟦*distinct*(*vertices f*); *distinct*(*vertices f′*);
vertices f ≅ *vertices f′*; *x* ∈ 𝒱 *f* ⟧ ⟹
verticesFrom f x = *verticesFrom f′ x*
by(*clarsimp simp*:*congs-def verticesFrom-Def splitAt-rotate-pair-conv*)

lemma *verticesFrom-in*[*intro*]: *v* ∈ 𝒱 *f* ⟹ *a* ∈ 𝒱 *f* ⟹ *a* ∈ *set* (*verticesFrom f v*)
by (*auto dest*: *verticesFrom-congs congs-pres-nodes*)

lemma *verticesFrom-in'*: $a \in \mathit{set}\ (\mathit{verticesFrom}\ f\ v) \Longrightarrow a \neq v \Longrightarrow a \in \mathcal{V}\ f$
apply (*cases* $v \in \mathcal{V}\ f$) **apply** (*auto dest*: *verticesFrom-congs congs-pres-nodes*)
by (*simp add*: *verticesFrom-Def*)

lemma *set-verticesFrom*:
$v \in \mathcal{V}\ f \Longrightarrow \mathit{set}\ (\mathit{verticesFrom}\ f\ v) = \mathcal{V}\ f$
apply(*auto*)
apply (*auto simp add*: *verticesFrom-Def*)
done

lemma *verticesFrom-hd*: $\mathit{hd}\ (\mathit{verticesFrom}\ f\ v) = v$ **by** (*simp add*: *verticesFrom-Def*)

lemma *verticesFrom-distinct*[*simp*]: $\mathit{distinct}\ (\mathit{vertices}\ f) \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow \mathit{distinct}$
$(\mathit{verticesFrom}\ f\ v)$ **apply** (*frule-tac verticesFrom-congs*) **by** (*auto simp*: *congs-distinct*)

lemma *verticesFrom-nextElem-eq*:
$\mathit{distinct}\ (\mathit{vertices}\ f) \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow u \in \mathcal{V}\ f \Longrightarrow$
$\mathit{nextElem}\ (\mathit{verticesFrom}\ f\ v)\ (\mathit{hd}\ (\mathit{verticesFrom}\ f\ v))\ u$
$= \mathit{nextElem}\ (\mathit{vertices}\ f)\ (\mathit{hd}\ (\mathit{vertices}\ f))\ u$ **apply** (*subgoal-tac* $(\mathit{verticesFrom}\ f\ v)$
$\cong (\mathit{vertices}\ f))$
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*: $\mathit{distinct}\ (\mathit{vertices}\ f) \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow i < \mathit{length}$
$(\mathit{vertices}\ f) \Longrightarrow \mathit{last}\ (\mathit{verticesFrom}\ f\ v) \neq u \Longrightarrow (\mathit{verticesFrom}\ f\ v)!i = u \Longrightarrow f \cdot$
$u = (\mathit{verticesFrom}\ f\ v)!(\mathit{Suc}\ i)$
proof −
assume *dist*: $\mathit{distinct}\ (\mathit{vertices}\ f)$ **and** *ith*: $(\mathit{verticesFrom}\ f\ v)!i = u$ **and** *small-i*:
$i < \mathit{length}\ (\mathit{vertices}\ f)$ **and** *notlast*: $\mathit{last}\ (\mathit{verticesFrom}\ f\ v) \neq u$ **and** *v*: $v \in \mathcal{V}\ f$
hence *eq*: $(\mathit{vertices}\ f) \cong (\mathit{verticesFrom}\ f\ v)$ **by** (*auto simp*: *verticesFrom-congs*)
from *ith eq small-i* **have** $u \in \mathit{set}\ (\mathit{verticesFrom}\ f\ v)$ **by** (*auto simp*: *congs-length*)
with *eq* **have** *u*: $u \in \mathcal{V}\ f$ **by** (*auto simp*: *congs-pres-nodes*)
def $\mathit{ls} \equiv \mathit{verticesFrom}\ f\ v$
with *dist ith* **have**
$\mathit{ls}!i = u$ **by** *auto*
have *ind*: $\bigwedge c\ i.\ i < \mathit{length}\ \mathit{ls} \Longrightarrow \mathit{distinct}\ \mathit{ls} \Longrightarrow \mathit{last}\ \mathit{ls} \neq u \Longrightarrow \mathit{ls}!i = u \Longrightarrow$
$u \in \mathit{set}\ \mathit{ls} \Longrightarrow$
$\mathit{nextElem}\ \mathit{ls}\ c\ u = \mathit{ls}\ !\ \mathit{Suc}\ i$
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** $u \notin \mathit{set}\ \mathit{ms}$ **by** *auto*
with *Cons True* **have** *i*: $i = 0$ **apply** (*induct i*) **by** *auto*
with *Cons* **show** *?thesis* **apply** (*simp split*: *split-if-asm*) **apply** (*cases ms*)
by *simp-all*

```
    next
      case False with Cons have a1: u ∈ set ms by auto
      then have ms: ms ≠ [] by auto
      from False Cons have i: 0 < i by auto
      def 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 nextElem-vFrom-suc2: distinct (vertices f) ⟹ last (verticesFrom f v) =
u ⟹ v ∈ 𝒱 f ⟹ u ∈ 𝒱 f ⟹ f · u = hd (verticesFrom f v)
proof −
  assume dist: distinct (vertices f) and last: last (verticesFrom f v) = u
    and v: v ∈ 𝒱 f and u: u ∈ 𝒱 f
  def ls ≡ verticesFrom f v
  have ind: ⋀ c. distinct ls ⟹ last ls = u ⟹ u ∈ set ls ⟹ nextElem ls c u =
c
  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 ms = []  apply (cases ms rule: rev-exhaust) by
auto
      then show ?thesis by auto
    next
      case False with Cons have a1: u ∈ set ms by auto
      then have ms: ms ≠ [] by auto

       with False Cons ms have nextElem ms c u = c apply (rule-tac Cons) by
simp-all
      with False ms show ?thesis by simp
    qed
  qed
  from dist ls-def last v u have nextElem ls (hd ls) u = hd ls apply (rule-tac ind)
by auto
 with dist u v ls-def show ?thesis by (simp add: nextVertex-def verticesFrom-nextElem-eq)
qed

lemma verticesFrom-nth: distinct (vertices f) ⟹ d < length (vertices f) ⟹
```

```
  v ∈ 𝒱 f ⟹ (verticesFrom f v)!d = f^d · 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 ∈ 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: distinct (vertices f) ⟹ v ∈ set (vertices f) ⟹
  length (verticesFrom f v) = length (vertices f)
by (auto intro: congs-length verticesFrom-congs)

lemma verticesFrom-between: v' ∈ 𝒱 f ⟹ pre-between (vertices f) u v ⟹
  between (vertices f) u v = between (verticesFrom f v') u v
by (auto intro!: between-congs verticesFrom-congs)


lemma verticesFrom-is-nextElem: v ∈ 𝒱 f ⟹
   is-nextElem (vertices f) a b = is-nextElem (verticesFrom f v) a b
    apply (rule is-nextElem-congs-eq) by (rule verticesFrom-congs)

lemma verticesFrom-is-nextElem-last: v' ∈ 𝒱 f ⟹ distinct (vertices f)
  ⟹ is-nextElem (verticesFrom f v') (last (verticesFrom f v')) v  ⟹ 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*: $v' \in \mathcal{V}\ f \Longrightarrow$ *distinct* (*vertices f*)
$\Longrightarrow$ *is-nextElem* (*verticesFrom f v'*) *u v'* $\Longrightarrow u =$ *last* (*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*: $v \in \mathcal{V}\ f \Longrightarrow \mathcal{V}\ f =$ *set*(*verticesFrom f v*)
proof –
assume $v \in \mathcal{V}\ f$
then have *fst* (*splitAt v* (*vertices f*)) @ [*v*] @ *snd* (*splitAt v* (*vertices f*)) = *vertices f*
apply (*drule-tac splitAt-ram*) **by** *simp*
moreover have *set* (*fst* (*splitAt v* (*vertices f*)) @ [*v*] @ *snd* (*splitAt v* (*vertices f*))) = *set* (*verticesFrom f v*)
by (*auto simp*: *verticesFrom-Def*)
ultimately show *?thesis* **by** *simp*
qed

lemma *verticesFrom-pres-nodes*: $v \in \mathcal{V}\ f \Longrightarrow w \in \mathcal{V}\ f \Longrightarrow w \in$ *set* (*verticesFrom f v*)
by (*auto dest*: *verticesFrom-pres-nodes1*)

lemma *before-verticesFrom*: *distinct* (*vertices f*) $\Longrightarrow v \in \mathcal{V}\ f \Longrightarrow w \in \mathcal{V}\ f \Longrightarrow$
$v \neq w \Longrightarrow$ *before* (*verticesFrom f v*) *v w*
proof –
assume *v*: $v \in \mathcal{V}\ f$ **and** *w*: $w \in \mathcal{V}\ f$ **and** *neq*: $v \neq w$
have *hd* (*verticesFrom f v*) = *v* **by** (*rule verticesFrom-hd*)
with *v* **have** *vf*:*verticesFrom f v* = *v* # *tl* (*verticesFrom f v*)
apply (*frule-tac verticesFrom-pres-nodes1*)
apply (*cases verticesFrom f v*) **by** *auto*
moreover with *v w* **have** $w \in$ *set* (*verticesFrom f v*) **by** (*auto simp*: *verticesFrom-pres-nodes*)
ultimately have $w \in$ *set* (*v* # *tl* (*verticesFrom f v*)) **by** *auto*
with *neq* **have** $w \in$ *set* (*tl* (*verticesFrom f v*)) **by** *auto*
with *vf* **have** *verticesFrom f v* =
[] @ *v* # *fst* (*splitAt w* (*tl* (*verticesFrom f v*))) @ *w* # *snd* (*splitAt w* (*tl* (*verticesFrom f v*)))
by (*auto dest*: *splitAt-ram*)
then show *?thesis* **apply** (*unfold before-def*) **by** (*intro exI*)
qed

lemma *next-not-before*:
$[\![$ *distinct*(*vertices f*); $x \in \mathcal{V}\ f$; $y \in \mathcal{V}\ f$; $x \neq f \cdot y$ $]\!] \Longrightarrow$
$\neg$ *before* (*verticesFrom f x*) ($f \cdot y$) *y*
apply(*drule split-list*)
apply(*clarsimp*)
apply(*clarsimp simp*:*before-def verticesFrom-Def*)
apply(*erule disjE*)

```
 apply(clarsimp simp:Cons-eq-append-conv append-eq-append-conv2)
 apply(erule disjE)
  apply(clarsimp simp:append-eq-Cons-conv)
  apply(erule disjE)
   apply clarsimp
  apply(clarsimp simp:append-eq-append-conv2)
  apply(fastsimp simp:append-eq-Cons-conv)
 apply clarsimp
apply(erule disjE)
 apply(drule split-list)
 apply(clarsimp simp:nextVertex-def Cons-eq-append-conv split:list.splits)
 apply(clarsimp simp:append-eq-append-conv2)
 apply(erule disjE)
  apply(fastsimp simp:append-eq-Cons-conv)
 apply clarsimp
 apply(erule disjE)
  apply(clarsimp simp:append-eq-Cons-conv)
 apply(fastsimp simp:Cons-eq-append-conv)
apply(drule split-list)
apply(clarsimp simp:nextVertex-def split:list.splits)
 apply(clarsimp simp:hd-append split: split-if-asm)
 apply(clarsimp simp:Cons-eq-append-conv)
 apply(clarsimp simp:append-eq-append-conv2)
 apply(erule disjE)
  apply(fastsimp simp:append-eq-Cons-conv)
 apply(fastsimp simp:Cons-eq-append-conv neq-Nil-conv)
apply(clarsimp simp:Cons-eq-append-conv)
apply(clarsimp simp:append-eq-append-conv2)
apply(erule disjE)
 apply(clarsimp simp:append-eq-Cons-conv)
apply(clarsimp simp:Cons-eq-append-conv)
apply(erule disjE)
 apply(clarsimp simp:append-eq-append-conv2)
 apply(erule disjE)
  apply(fastsimp simp:append-eq-Cons-conv)
 apply clarsimp
apply(clarsimp simp:append-eq-append-conv2)
apply(fastsimp simp:append-eq-Cons-conv)
done

lemma last-vFrom:
 ⟦ distinct(vertices f); x ∈ 𝒱 f ⟧ ⟹ last(verticesFrom f x) = f^-1 · x
apply(frule split-list)
apply(clarsimp simp:prevVertex-def verticesFrom-Def split:list.split)
done


MLfast-arith-neq-limit := 1
lemma rotate-before-vFrom:
```

```
  ⟦ distinct(vertices f); r ∈ 𝒱 f; r≠u ⟧ ⟹
   before (verticesFrom f r) u v ⟹ before (verticesFrom f v) r u
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:
 ⟦ before(verticesFrom f x) y z; distinct(vertices f); x ∈ 𝒱 f; x ≠ y ⟧ ⟹
  y ∈ set(between (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(fastsimp simp:between-def split-def)
done
MLfast-arith-neq-limit := 3
```

```
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:split-if-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
```

12.11 *splitFace*

```
constdefs pre-splitFace :: graph ⇒ vertex ⇒ vertex ⇒ face ⇒ vertex list ⇒ bool
  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 ⟹ 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 ⟹ oldF ∈ ℱ g
  apply (unfold pre-splitFace-def) by force

declare pre-splitFace-def [simp del]

lemma p-splitFace-ne:
  pre-splitFace g ram1 ram2 oldF nvs ⟹ (ram1, ram2) ∈ edges oldF ⟹ nvs ≠
[]
  apply (unfold pre-splitFace-def) by force


lemma splitFace-preserve-face-not-empty:
  f ∉ ℱ g  ⟹  vertices f' = vs ⟹ ram2 ∈ set vs ⟹
   f ∈ ℱ (snd (snd (splitFace g ram1 ram2 f' ns))) ⟹ vertices f ≠ []
proof (induct g)
  assume r2: ram2 ∈ set vs
  then have vs: vs ≠ [] by auto
  def x ≡  hd vs
  def xs ≡ tl vs
  with x-def xs-def vs have vs-eq: vs = x # xs by simp
  note vs-eq-sym = vs-eq [symmetric]

  case (Graph fs n Fs hs) then  show ?thesis apply (case-tac f = Face (rev ns
@ ram1 # between vs ram1 ram2 @ [ram2]) Nonfinal)
    apply (simp add: Graph r2 splitFace-def split-face-def split-def)
    apply (simp add: splitFace-def split-face-def split-def)
    apply (drule-tac replace1) by (auto simp: vs-eq-sym)
qed

lemma splitFace-preserve-faces:
   f ∈ ℱ g ⟹ f ≠ oldF ⟹
   f ∈ ℱ (snd (snd (splitFace g ram1 ram2 oldF nvs)))
apply (induct g)
by (auto intro: replace4 simp: splitFace-def split-def split-face-def)

lemma splitFace-split-face:
```

oldF $\in \mathcal{F}$ *g* $\Longrightarrow$
$(f_1, f_2,$ *newGraph*$)$ = *splitFace g* ram_1 ram_2 *oldF newVs* $\Longrightarrow$
(f_1, f_2) = *split-face oldF* ram_1 ram_2 *newVs*
by (*simp add*: *splitFace-def split-def*)

lemma *splitFace-add-f21*:
(*f12*, *f21*, *g'*) = *splitFace g ram1 ram2 oldF newVertexList* $\Longrightarrow$ *oldF* $\in \mathcal{F}$ *g*
$\Longrightarrow$ *f21* $\in \mathcal{F}$ *g'*
by (*auto simp add*: *splitFace-def split-def*)

lemma *split-face-empty-ram2-ram1-in-f12*:
pre-split-face oldF ram1 ram2 [] $\Longrightarrow$
(*f12*, *f21*) = *split-face oldF ram1 ram2* [] $\Longrightarrow$ (*ram2*, *ram1*) $\in$ *edges f12*
proof −
assume *split*: (*f12*, *f21*) = *split-face oldF ram1 ram2* []
pre-split-face oldF ram1 ram2 []
then have *ram2* $\in \mathcal{V}$ *f12* **by** (*simp add*: *split-face-def*)
moreover with *split* **have** *f12* · *ram2* = *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** *ram1* = *f12* · *ram2*
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 [] $\Longrightarrow$
(*ram2*, *ram1*) $\in$ *edges* (*fst* (*split-face oldF ram1 ram2* []))
proof −
assume *split*: *pre-split-face oldF ram1 ram2* []
def *f12* $\equiv$ *fst* (*split-face oldF ram1 ram2* [])
def *f21* $\equiv$ *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** (*ram2*, *ram1*) $\in$ *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 [] $\Longrightarrow$
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF* [] $\Longrightarrow$
(*ram2*, *ram1*) $\in$ *edges f12*
proof −
assume *pre*: *pre-splitFace g ram1 ram2 oldF* []
then have *oldF*: *oldF* $\in \mathcal{F}$ *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'*)
by (*rule pre-splitFace-pre-split-face*)
qed

lemma *splitFace-f12-new-vertices*:
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs* $\Longrightarrow$
v $\in$ *set newVs* $\Longrightarrow$ *v* $\in$ $\mathcal{V}$ *f12*
apply (*unfold splitFace-def*)
apply (*simp add*: *split-def*)
apply (*unfold split-face-def Let-def*)
by *simp*

lemma *splitFace-add-vertices*:
newVertexList = [*countVertices g* ..< *countVertices g* + *n*]
$\Longrightarrow$ (*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF* (*rev newVertexList*)
$\Longrightarrow$ *vertices newGraph* = *vertices g* @ *newVertexList*
apply (*auto simp*: *splitFace-def split-def*) **apply** (*cases g*) **apply** *simp* **apply** *auto*
apply (*induct n*) **by** *auto*

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* $\Longrightarrow$
oldF $\neq$ *f12* $\Longrightarrow$ *oldF* $\neq$ *f21* $\Longrightarrow$ *distinct* (*faces g*) $\Longrightarrow$
oldF $\notin$ $\mathcal{F}$ *newGraph*
by (*simp add*: *splitFace-def split-def distinct-set-replace*)

lemma *splitFace-faces-1*:
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVertexList* $\Longrightarrow$
oldF $\in$ $\mathcal{F}$ *g* $\Longrightarrow$
set (*faces newGraph*) $\cup$ {*oldF*} = {*f12*, *f21*} $\cup$ *set* (*faces g*)
(**is** *?oldF* $\Longrightarrow$ *?C* $\Longrightarrow$ *?A* = *?B*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x* $\in$ *?A* **and** *?C* **and** *?oldF*

```
 then show x ∈ ?B apply (simp add: splitFace-def split-def) by (auto dest:
replace1)
next
  fix x
  assume x ∈ ?B and ?C and ?oldF
  then show x ∈ ?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
⟹
  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
⟹
  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 ∈ ℱ g
  ∨ f = fst (splitFace g v a f' nvl)
  ∨ f = (fst (snd (splitFace g v a f' nvl)))
 apply (simp add: splitFace-def split-def)
 apply safe by (frule replace5) simp

lemma pre-FaceDiv-between1: pre-splitFace g' ram1 ram2 f [] ⟹
  ¬ between (vertices f) ram1 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 ∈ 𝒱 f apply (unfold pre-splitFace-def) by simp
 from pre-f have ¬ is-nextElem (vertices f) ram1 ram2 apply (unfold pre-splitFace-def)
by auto
  with pre-f have ¬ 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 ∈ set (verticesFrom f ram1) apply (unfold pre-splitFace-def)
by auto
  moreover
  from pre-f have ram2 ≠ ram1 apply (unfold pre-splitFace-def) by auto
  ultimately have ram2-not: ram2 ≠ 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)))) = [] ⟹
False
  proof −
    assume fst (splitAt ram2 (snd (splitAt ram1 (verticesFrom f ram1)))) = []
    moreover
   from ram2 pre-f have ram2 ∈ set (verticesFrom f ram1)apply (unfold pre-splitFace-def)
      by auto
     with pre-f have ram2 ∈ 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* $\neq$ [] **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* [] $\Longrightarrow$
¬ *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

constdefs
Edges :: *vertex list* ⇒ (*vertex* × *vertex*) *set*
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* $\Longrightarrow$ (*x*,*y*) ∉ *Edges xs*
by(*unfold Edges-def*)(*blast intro*:*is-sublist-in*)

lemma *notinset-notinEdge2*: *y* ∉ *set xs* $\Longrightarrow$ (*x*,*y*) ∉ *Edges xs*
by(*unfold Edges-def*)(*blast intro*:*is-sublist-in1*)

lemma *in-Edges-in-set*: (*x*,*y*) : *Edges vs* $\Longrightarrow$ *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*)) $\Longrightarrow$ $\mathcal{E}$ *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-conv-Edges-if-cong:
 ⟦ vertices (f::face) ≅ vs; distinct vs; vs ≠ [] ⟧ ⟹
 ℰ f = Edges vs ∪ {(last vs,hd vs)}
apply(frule congs-distinct)
apply(clarsimp simp: expand-set-eq is-nextElem-congs-eq)
using is-nextElem-edges-eq[of Face vs Final,symmetric]
apply(simp del:is-nextElem-edges-eq)
apply(simp add:edges-conv-Edges)
done

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 add:is-sublist-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 ∩ set ys = {} ⟹ Edges xs ∩ Edges ys = {}
by(unfold Edges-def)(blast intro:is-sublist-in is-sublist-in1)
```

```
lemma disj-sets-disj-Edges2:
 set ys ∩ set xs = {} ⟹ Edges xs ∩ 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 length-conv-card-Edges:
 distinct xs ⟹ |xs| = (if |xs| = 0 then 0 else card (Edges xs) + 1)
apply(induct xs)
 apply simp
apply(clarsimp simp:Edges-Cons notinset-notinEdge1)
done
```

```
lemma card-edges:
 distinct(vertices(f::face)) ⟹ card(ℰ f) = |vertices f|
apply(frule length-conv-card-Edges)
apply(simp add:edges-conv-Edges)
apply(clarsimp simp:Edges-Cons notinset-notinEdge2 card-insert-if neq-Nil-conv
  split:split-if-asm)
done
```

```
MLfast-arith-neq-limit := 1
lemma Edges-compl:
 ⟦ distinct vs; x ∈ set vs; y ∈ set vs; x ≠ y ⟧ ⟹
 Edges(x # between vs x y @ [y]) ∩ Edges(y # between vs y x @ [x]) = {}
apply(subgoal-tac
 !!xs (ys::vertex list). xs ≠ [] ⟹ set xs ∩ set ys = {} ⟹ hd xs ∉ 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(fastsimp)
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 fastsimp
done

lemma Edges-disj:
 ⟦ distinct vs; x ∈ set vs; z ∈ set vs; x ≠ y; y ≠ z;
    y ∈ set(between vs x z) ⟧ ⟹
 Edges(x # between vs x y @ [y]) ∩ Edges(y # between vs y z @ [z]) = {}
apply(subgoal-tac
 !!xs (ys::vertex list). xs ≠ [] ⟹ set xs ∩ set ys = {} ⟹ hd xs ∉ 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 fastsimp
 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 fastsimp
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 fastsimp
done

lemma edges-conv-Un-Edges:
 ⟦ distinct(vertices(f::face)); x ∈ 𝒱 f; y ∈ 𝒱 f; x ≠ y ⟧ ⟹
  ℰ f = Edges(x # between (vertices f) x y @ [y]) ∪
          Edges(y # between (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(fastsimp)
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(fastsimp)
done
MLfast-arith-neq-limit := 3


lemma Edges-between-edges:
  ⟦ (a,b) ∈ Edges (u # between (vertices(f::face)) u v @ [v]);
    pre-split-face f u v vs ⟧ ⟹ (a,b) ∈ ℰ f
apply(simp add:pre-split-face-def)
apply(induct f)
apply(simp add:edges-conv-Edges Edges-Cons)
apply clarify
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 m = |as|+1 in is-nextElem-rotate-eq[THEN iffD1,standard])
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


MLfast-arith-neq-limit := 10
lemma triangle-if-3circular:
 ⟦ distinct[a,b,c]; distinct(vertices(f::face));
   (a,b) ∈ ℰ f; (b,c) ∈ ℰ f; (c,a) ∈ ℰ f ⟧
  ⟹ ℰ f = {(a,b),(b,c),(c,a)}
apply(rule card-seteq[OF finite-edges, symmetric])
 apply simp
```

```
apply(simp add:card-edges)
apply(rule ccontr)
apply(drule is-nextElem-nth1)+
apply clarsimp
apply(simp add:nth-eq-iff-index-eq)
apply(case-tac Suc i = |vertices f|)
 apply(simp)
apply(case-tac Suc(Suc i) = |vertices f|)
 apply(simp)
apply(case-tac Suc(Suc(Suc i)) = |vertices f|)
 apply simp
apply simp
done
MLfast-arith-neq-limit := 3
```

```
lemma edges-split-face1: pre-split-face f u v vs ⟹
 ℰ(fst(split-face f u v vs)) =
 Edges(v # rev vs @ [u]) ∪ Edges(u # between (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 ⟹
 ℰ(snd(split-face f u v vs)) =
 Edges(u # vs @ [v]) ∪ Edges(v # between (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 [] ⟹
  (f12, f21) = split-face oldF ram1 ram2 [] ⟹ (ram1, ram2) ∈ edges f21
proof −
  assume split: (f12, f21) = split-face oldF ram1 ram2 []
  pre-split-face oldF ram1 ram2 []
  then have ram1 ∈ 𝒱 f21 by (simp add: split-face-def)
  moreover with split have f21 · ram1 = hd (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 ram2 = f21 · 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 [] ⟹
  (ram1, ram2) ∈ edges (snd (split-face oldF ram1 ram2 []))
proof −
  assume split: pre-split-face oldF ram1 ram2 []
  def f12 ≡ fst (split-face oldF ram1 ram2 [])
  def 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 [] ⟹
  (f12, f21, newGraph) = splitFace g ram1 ram2 oldF [] ⟹
  (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')
  by (rule pre-splitFace-pre-split-face)
qed

lemma splitFace-f21-new-vertices:
  (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs ⟹
  v ∈ set  newVs ⟹ v ∈ 𝓥 f21
  apply (unfold splitFace-def)
  apply (simp add: split-def)
  apply (unfold split-face-def)
  by simp

lemma split-face-f12-f21-neq:
  pre-split-face oldF ram1 ram2 vs ⟹
  (f12, f21) = split-face oldF ram1 ram2 vs ⟹ f12 ≠ f21
proof −
  assume split: (f12, f21) = split-face oldF ram1 ram2 vs
  pre-split-face oldF ram1 ram2 vs
  then have distinct (vertices f12) by (rule split-face-distinct1)
```

with *split* **show** *?thesis* **apply** (*simp add*: *split-face-def*)
apply (*case-tac vs rule*:*rev-exhaust*) **apply** (*simp add*: *pre-split-face-def*) **by** *simp*
qed

lemma *split-face-edges-f12*: *pre-split-face f ram1 ram2 vs* ⟹
(*f12*, *f21*) = *split-face f ram1 ram2 vs* ⟹ *vs* ≠ [] ⟹ *vs1* = *between* (*vertices f*) *ram1 ram2* ⟹ *vs1* ≠ [] ⟹
edges f12 = {(*hd vs*, *ram1*) , (*ram1*, *hd vs1*), (*last vs1*, *ram2*), (*ram2*, *last vs*)} ∪
Edges(*rev vs*) ∪ *Edges vs1* (**concl is** *?lhs* = *?rhs*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x*: *x* ∈ *?lhs* **and** *vors*: *pre-split-face f ram1 ram2 vs*
(*f12*, *f21*) = *split-face f ram1 ram2 vs vs* ≠ [] *vs1* = *between* (*vertices f*) *ram1 ram2* *vs1* ≠ []
def *c* ≡ *fst x*
def *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 (*subgoal-tac ys* = *as*) **apply**(*drule last-rev*) **apply** (*simp*)
apply (*rule dist-at1*) **apply** (*rule dist-f12*) **apply** (*rule sym*) **apply** *simp*
apply *simp*
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* **and** *vors*: *pre-split-face f ram1 ram2 vs*
(*f12*, *f21*) = *split-face f ram1 ram2 vs vs* ≠ [] *vs1* = *between* (*vertices f*) *ram1 ram2* *vs1* ≠ []
def *c* ≡ *fst x*
def *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 = 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: pre-split-face f ram1 ram2 [] ⟹
  (f12, f21) = split-face f ram1 ram2 [] ⟹ vs1 = between (vertices f) ram1 ram2
⟹ vs1 ≠ [] ⟹
  edges f12 = {(ram2, ram1) , (ram1, hd vs1), (last vs1, ram2)} ∪
  Edges vs1 (concl is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs and vors: pre-split-face f ram1 ram2 []
    (f12, f21) = split-face f ram1 ram2 []  vs1 = between (vertices f) ram1 ram2
vs1 ≠ []
  def c ≡ fst x
  def 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* **and** *vors*: *pre-split-face f ram1 ram2* []
(*f12*, *f21*) = *split-face f ram1 ram2* [] *vs1* = *between* (*vertices f*) *ram1 ram2* *vs1* ≠ []
def *c* ≡ *fst x*
def *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* = *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*: *pre-split-face f ram1 ram2 vs* ⟹
(*f12*, *f21*) = *split-face f ram1 ram2 vs* ⟹ *vs* ≠ [] ⟹ *between* (*vertices f*) *ram1 ram2* = [] ⟹
edges f12 = {(*hd vs*, *ram1*) , (*ram1*, *ram2*), (*ram2*, *last vs*)} ∪
Edges(*rev vs*) (**concl is** *?lhs* = *?rhs*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x*: *x* ∈ *?lhs* **and** *vors*: *pre-split-face f ram1 ram2 vs*
(*f12*, *f21*) = *split-face f ram1 ram2 vs vs* ≠ [] *between* (*vertices f*) *ram1 ram2* = []
def *c* ≡ *fst x*
def *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* **and** *vors*: *pre-split-face f ram1 ram2 vs*
(*f12*, *f21*) = *split-face f ram1 ram2 vs vs* ≠ [] *between* (*vertices f*) *ram1 ram2* = []
def *c* ≡ *fst x*
def *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*: *pre-split-face f ram1 ram2 []* ⟹
(*f12*, *f21*) = *split-face f ram1 ram2 []* ⟹ *between* (*vertices f*) *ram1 ram2* = [] ⟹
edges f12 = {(*ram2*, *ram1*) , (*ram1*, *ram2*)} (**concl is** *?lhs* = *?rhs*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x*: *x* ∈ *?lhs* **and** *vors*: *pre-split-face f ram1 ram2* []
(*f12*, *f21*) = *split-face f ram1 ram2* [] *between* (*vertices f*) *ram1 ram2* = []
def *c* ≡ *fst x*
def *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** *force*
apply *simp* **apply** (*elim exE*)
apply (*case-tac c* = *ram1*) **apply** *simp*
apply (*case-tac as*) **apply** *simp*
apply (*case-tac list*) **apply** *simp* **apply** *simp*
apply (*case-tac as*) **apply** *simp* **apply** (*case-tac list*) **apply** *simp* **by** *simp*
next
fix *x*
assume *x*: *x* ∈ *?rhs* **and** *vors*: *pre-split-face f ram1 ram2* []
(*f12*, *f21*) = *split-face f ram1 ram2* [] *between* (*vertices f*) *ram1 ram2* = []
def *c* ≡ *fst x*
def *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*)
by *auto*
qed

lemma *split-face-edges-f12-subset*: *pre-split-face f ram1 ram2 vs* ⟹
(*f12*, *f21*) = *split-face f ram1 ram2 vs* ⟹ *vs* ≠ [] ⟹

{(*hd vs*, *ram1*), (*ram2*, *last vs*)} ∪ *Edges*(*rev vs*) ⊆ *edges f12*
apply (*case-tac between* (*vertices f*) *ram1 ram2*)
apply (*frule split-face-edges-f12-bet*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *force*
apply (*frule split-face-edges-f12*) **apply** *simp+* **by** *force*

lemma *split-face-edges-f21*: *pre-split-face f ram1 ram2 vs* ⟹
(*f12*, *f21*) = *split-face f ram1 ram2 vs* ⟹ *vs* ≠ [] ⟹ *vs2* = *between* (*vertices f*) *ram2 ram1* ⟹ *vs2* ≠ [] ⟹
edges f21 = {(*last vs2*, *ram1*) , (*ram1*, *hd vs*), (*last vs*, *ram2*), (*ram2*, *hd vs2*)} ∪
Edges vs ∪ *Edges vs2* (**concl is** *?lhs* = *?rhs*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x*: *x* ∈ *?lhs* **and** *vors*: *pre-split-face f ram1 ram2 vs*
(*f12*, *f21*) = *split-face f ram1 ram2 vs vs* ≠ [] *vs2* = *between* (*vertices f*) *ram2 ram1* *vs2* ≠ []
def *c* ≡ *fst x*
def *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* = *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* **and** *vors*: *pre-split-face f ram1 ram2 vs*
(*f12*, *f21*) = *split-face f ram1 ram2 vs vs* ≠ [] *vs2* = *between* (*vertices f*) *ram2*

ram1 vs2 ≠ []
def *c* ≡ *fst x*
def *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*)
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* ∧ *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*) ∧ *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*) ∈ *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*: *pre-split-face f ram1 ram2* [] ⟹
(*f12*, *f21*) = *split-face f ram1 ram2* [] ⟹ *vs2* = *between* (*vertices f*) *ram2 ram1* ⟹ *vs2* ≠ [] ⟹
edges f21 = {(*last vs2*, *ram1*) , (*ram1*, *ram2*), (*ram2*, *hd vs2*)} ∪
Edges vs2 (**concl is** *?lhs* = *?rhs*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x*: *x* ∈ *?lhs* **and** *vors*: *pre-split-face f ram1 ram2* []
(*f12*, *f21*) = *split-face f ram1 ram2* [] *vs2* = *between* (*vertices f*) *ram2 ram1* *vs2* ≠ []
def *c* ≡ *fst x*
def *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 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* ∈ *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 ∈ 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 ∈ ?rhs and vors: pre-split-face f ram1 ram2 []
    (f12, f21) = split-face f ram1 ram2 [] vs2 = between (vertices f) ram2 ram1
vs2 ≠ []
  def c ≡ fst x
  def 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)
    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] =
       [] @ ram2 # hd (between (vertices f) ram2 ram1) # tl (between (vertices f)
ram2 ram1) @ [ram1]) apply assumption apply simp
   apply (case-tac c = ram1 ∧ d = ram2) apply (rule disjI2) apply simp apply
simp
   apply (case-tac c = last (between (vertices f) ram2 ram1) ∧ 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*: *pre-split-face f ram1 ram2 vs* ⟹
(*f12*, *f21*) = *split-face f ram1 ram2 vs* ⟹ *vs* ≠ [] ⟹ *between* (*vertices f*) *ram2 ram1* = [] ⟹
edges f21 = {(*ram1*, *hd vs*), (*last vs*, *ram2*), (*ram2*, *ram1*)} ∪
Edges vs (**concl is** *?lhs* = *?rhs*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x*: *x* ∈ *?lhs* **and** *vors*: *pre-split-face f ram1 ram2 vs*
(*f12*, *f21*) = *split-face f ram1 ram2 vs vs* ≠ [] *between* (*vertices f*) *ram2 ram1* = []
def *c* ≡ *fst x*
def *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* = *last vs* ∧ *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* ∈ *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* # *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* # *ram1* # *ys* = *as*)
apply (*subgoal-tac* (*ram2* # *ram1* # *ys*) @ *c* # *a* # *list* = *as* @ *c* # *d* # *bs*)
apply (*thin-tac ram2* # *ram1* # *ys* @ *c* # *a* # *list* = *as* @ *c* # *d* # *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* = *ram2*) **apply** *simp*
apply (*subgoal-tac* [] = *as*) **apply** *simp*
apply (*rule dist-at1*) **apply** (*rule dist-f21*) **apply** (*rule sym*) **apply** *simp* **apply** *simp*

```
    apply (subgoal-tac c ∈ set (ram2 # ram1 # vs))
     apply (thin-tac ram2 # ram1 # vs = as @ c # d # bs) apply simp
    by simp
next
  fix x
  assume x: x ∈ ?rhs and vors: pre-split-face f ram1 ram2 vs
    (f12, f21) = split-face f ram1 ram2 vs vs ≠ []  between (vertices f) ram2 ram1 = []
  def c ≡ fst x
  def 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)
   apply (case-tac c = ram2 ∧ d = ram1) apply simp apply (rule disjI1) apply force
    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 # ram1 # a # list =
        [ram2] @ ram1 # a # list) apply assumption  apply simp
    apply (case-tac c = last vs ∧ d = 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: pre-split-face f ram1 ram2 [] ⟹
  (f12, f21) = split-face f ram1 ram2 [] ⟹ between (vertices f) ram2 ram1 = [] ⟹
  edges f21 = {(ram1, ram2), (ram2, ram1)} (concl is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs and vors: pre-split-face f ram1 ram2 []
    (f12, f21) = split-face f ram1 ram2 [] between (vertices f) ram2 ram1 = []
  def c ≡ fst x
  def 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* **and** *vors*: *pre-split-face f ram1 ram2* []
(*f12*, *f21*) = *split-face f ram1 ram2* [] *between* (*vertices f*) *ram2 ram1* = []
def *c* ≡ *fst x*
def *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* ⟹
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* ∈ *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* ∈ *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: pre-split-face f ram1 ram2 vs ⟹
  between (vertices f) ram1 ram2 = [] ⟹
  between (vertices f) ram2 ram1 = [] ⟹
  edges f = {(ram2, ram1) , (ram1, ram2)} (concl is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs and vors: pre-split-face f ram1 ram2 vs
     between (vertices f) ram1 ram2 = []
     between (vertices f) ram2 ram1 = []
  def c ≡ fst x
  def 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 ∈ ?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 and vors: pre-split-face f ram1 ram2 vs
    between (vertices f) ram1 ram2 = []
    between (vertices f) ram2 ram1 = []
  def c ≡ fst x
  def 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: pre-split-face f ram1 ram2 vs ⟹
  between (vertices f) ram1 ram2 = [] ⟹
  vs2 = between (vertices f) ram2 ram1 ⟹ vs2 ≠ [] ⟹
```

```
  edges f = {(last vs2, ram1) , (ram1, ram2), (ram2, hd vs2)} ∪
  Edges vs2 (concl is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs and vors: pre-split-face f ram1 ram2 vs
     between (vertices f) ram1 ram2 = []
     vs2 = between (vertices f) ram2 ram1  vs2 ≠ []
  def c ≡ fst x
  def 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 ¬ 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 ∈ ?rhs and vors: pre-split-face f ram1 ram2 vs
    between (vertices f) ram1 ram2 = []
    vs2 = between (vertices f) ram2 ram1  vs2 ≠ []
  def c ≡ fst x
  def 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 ∈ ?lhs apply (simp add: dist-f)
   apply (subgoal-tac ram1 ∈ 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) ∧ d = ram1) apply
simp apply simp apply (rule disjI1)
     apply (case-tac c = ram1 ∧ d = ram2) apply (simp add: is-sublist-def)
apply force apply simp
    apply (case-tac c = ram2 ∧ 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: pre-split-face f ram1 ram2 vs ⟹
  vs1 = between (vertices f) ram1 ram2 ⟹ vs1 ≠ [] ⟹
  between (vertices f) ram2 ram1 = [] ⟹
  edges f = {(ram2, ram1) , (ram1, hd vs1), (last vs1, ram2)} ∪
  Edges vs1 (concl is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs and vors: pre-split-face f ram1 ram2 vs
    vs1 = between (vertices f) ram1 ram2  vs1 ≠ []
    between (vertices f) ram2 ram1 = []
  def c ≡ fst x
  def 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 and vors: pre-split-face f ram1 ram2 vs
    vs1 = between (vertices f) ram1 ram2  vs1 ≠ []
    between (vertices f) ram2 ram1 = []
  def c ≡ fst x
  def 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 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 (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: pre-split-face f ram1 ram2 vs ⟹
  vs1 = between (vertices f) ram1 ram2 ⟹ vs1 ≠ [] ⟹
  vs2 = between (vertices f) ram2 ram1 ⟹ vs2 ≠ [] ⟹
  edges f = {(last vs2, ram1) , (ram1, hd vs1), (last vs1, ram2), (ram2, hd vs2)}
∪
  Edges vs1 ∪ Edges vs2 (concl is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs and vors: pre-split-face f ram1 ram2 vs
     vs1 = between (vertices f) ram1 ram2  vs1 ≠ []
     vs2 = between (vertices f) ram2 ram1  vs2 ≠ []
  def c ≡ fst x
  def 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 and vors: pre-split-face f ram1 ram2 vs
    vs1 = between (vertices f) ram1 ram2  vs1 ≠ []
    vs2 = between (vertices f) ram2 ram1  vs2 ≠ []
  def c ≡ fst x
  def 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 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 (case-tac c = last (between (vertices f) ram2 ram1) ∧ 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 # between (vertices f) ram2 ram1
=
```

(*ram1* # *ys*) @ *y* # *ram2* # (*between* (*vertices f*) *ram2 ram1*)) **apply** *assumption* **apply** *simp* **apply** *simp*
apply (*case-tac c* = *ram2* ∧ *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* # *between* (*vertices f*) *ram1 ram2* @ *ram2* # *a* # *list* =
(*ram1* # *between* (*vertices f*) *ram1 ram2*) @ *ram2* # *a* # (*list*)) **apply** *assumption* **apply** *simp* **apply** *simp*
apply (*case-tac* (*c*, *d*) ∈ *Edges* (*between* (*vertices f*) *ram1 ram2*)) **apply** (*simp add*: *Edges-def*)
apply (*subgoal-tac is-sublist* [*c*, *d*] ([*ram1*] @ *between* (*vertices f*) *ram1 ram2* @
(*ram2* # *between* (*vertices f*) *ram2 ram1*)))
apply *simp* **apply** (*rule is-sublist-add*) **apply** *simp*
apply *simp*
apply (*subgoal-tac is-sublist* [*c*, *d*] ((*ram1* # *between* (*vertices f*) *ram1 ram2* @
[*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-f12-f21*:
pre-split-face f ram1 ram2 vs ⟹ (*f12*, *f21*) = *split-face f ram1 ram2 vs* ⟹
vs ≠ []
⟹ *edges f12* ∪ *edges f21* = *edges f* ∪
{(*hd vs*, *ram1*), (*ram1*, *hd vs*), (*last vs*, *ram2*), (*ram2*, *last vs*)} ∪
Edges vs ∪
Edges (*rev vs*)
apply (*case-tac between* (*vertices f*) *ram1 ram2* = [])
apply (*case-tac between* (*vertices f*) *ram2 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 [] ⟹ (*f12*, *f21*) = *split-face f ram1 ram2* []
⟹ *edges f12* ∪ *edges f21* = *edges f* ∪
{(*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 ∈ ℱ g ⟹
  pre-split-face f ram1 ram2 vs ⟹ (f12, f21) = split-face f ram1 ram2 vs
  ⟹ ((a,b) ∈ edges f12 ∨ (a,b) ∈  edges f21) =
     ((a,b) ∈ edges f  ∨
  (((b,a) ∈ edges f12 ∨ (b,a) ∈  edges f21) ∧
  ((a,b) ∈ edges f12 ∨ (a,b) ∈ edges f21)))
  apply (subgoal-tac ((a,b) ∈ edges f12 ∪ edges f21) =
     ((a,b) ∈ edges f ∨ ((b,a) ∈ edges f12 ∪ edges f21) ∧ (a,b) ∈ edges f12 ∪ 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 ⟹
  (f12, f21, g′) = splitFace g ram1 ram2 f vs ⟹ vs ≠ [] ⟹
  edges g′ = edges g ∪ edges f ∪ Edges vs ∪ Edges(rev vs) ∪
  {(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-def) apply (rule equalityI) apply simp
   apply (rule conjI) apply (rule subsetI) apply simp apply (erule bexE) apply
(drule replace5)
    apply (case-tac xa ∈ 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* ⟹ *edges f* ⊆ *edges g*
apply (*simp add*: *edges-graph-def*) **by** (*force*)

lemma *pre-splitFace-edges-f-in-g2*: *pre-splitFace g ram1 ram2 f vs* ⟹ *x* ∈ *edges f* ⟹ *x* ∈ *edges g*
apply (*simp add*: *edges-graph-def*) **by** (*force*)

lemma *splitFace-edges-g'*: *pre-splitFace g ram1 ram2 f vs* ⟹
(*f12*, *f21*, *g'*) = *splitFace g ram1 ram2 f vs* ⟹ *vs* ≠ [] ⟹

edges g′ = edges g ∪ Edges vs ∪ Edges(rev vs) ∪
{(ram2, last vs), (hd vs, ram1), (ram1, hd vs), (last vs, ram2)}
apply (*subgoal-tac edges g ∪ 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* [] ⟹
(*f12*, *f21*, *g′*) = *splitFace g ram1 ram2 f* [] ⟹
edges g′ = edges g ∪ {(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′* = ($\bigcup_{a \in set\ (replace\ f\ [f21]\ (faces\ g))}$ *edges a*) ∪
edges (*f12*) **by** (*auto simp*: *splitFace-def split-def edges-graph-def*)
with *pre* **show** *?thesis* **apply** (*simp add*: *UNION-def*) **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* [])
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*) $\in$ *edges f21*)
apply (*rule conjI*) **apply** (*rule disjI1*) **apply** (*rule bexI*) **apply** *simp* **apply** (*force*)
apply (*subgoal-tac* (*ram2*, *ram1*) $\in$ *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* $\exists$ *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*) $\in$ *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 $\Longrightarrow$
(f_1, f_2, g') = *splitFace g ram1 ram2 f vs* $\Longrightarrow$
edges g $\subseteq$ *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 v_1 v_2 *f vs* $\Longrightarrow$
edges g $\subseteq$ *edges*(*snd*(*snd*(*splitFace g* v_1 v_2 *f vs*)))
apply(*erule splitFace-edges-incr*
[**where** f_1 = *fst*(*splitFace g* v_1 v_2 *f vs*)
and f_2 = *fst*(*snd*(*splitFace g* v_1 v_2 *f vs*))])
apply(*auto*)
done

12.12 *removeNones*

constdefs *removeNones* :: *'a option list* ⇒ *'a list*
removeNones vOptionList ≡ [*the x*. *x* ∈ *vOptionList*, (*λx*. *x* ≠ *None*)]

declare *removeNones-def* [*simp*]
lemma *removeNones-inI*[*intro*]: *Some a* ∈ *set ls* ⟹ *a* ∈ *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-inI*[*intro*]: *Some a* ∈ *set ls* ⟹ *a* ∈ *set* (*removeNones ls*)
by (*induct ls*) *auto*
lemma *removeNones-empty*[*simp*]: *removeNones* [] = [] **by** *auto*
declare *removeNones-def* [*simp del*]

12.13 *natToVertexList*

consts *natToVertexListRec* ::
  nat ⇒ *vertex* ⇒ *face* ⇒ *nat list* ⇒ *vertex option list*
primrec
  natToVertexListRec old v f [] = []
  natToVertexListRec old v f (*i*#*is*) =
    (*if i* = *old then None*#*natToVertexListRec i v f is*
     else Some ($f^i \cdot v$)
        # *natToVertexListRec i v f is*)

consts *natToVertexList* ::
  vertex ⇒ *face* ⇒ *nat list* ⇒ *vertex option list*
primrec
  natToVertexList v f [] = []
  natToVertexList v f (*i*#*is*) =
    (*if i* = *0 then* (*Some v*)#(*natToVertexListRec i v f is*) *else* [])

12.14 *indexToVertexList*

lemma *nextVertex-inj*:
 distinct (*vertices f*) ⟹ *v* ∈ $\mathcal{V}$ *f* ⟹
 i < *length* (*vertices* (*f*::*face*)) ⟹ *a* < *length* (*vertices f*) ⟹

$f^{a}\!\cdot v = f^{i}\!\cdot v \Longrightarrow i = a$
proof –
assume *d*: *distinct* (*vertices f*) **and** *v*: $v \in \mathcal{V}\ f$ **and** *i*: $i <$ *length* (*vertices* (*f*::*face*))
and *a*: $a <$ *length* (*vertices f*) **and** *eq*: $f^{a}\!\cdot v = f^{i}\!\cdot v$
then have *eq*: (*verticesFrom f v*)!*a* = (*verticesFrom f v*)!*i* **by** (*simp add*: *verticesFrom-nth*)
def *xs* ≡ *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** (*verticesFrom f v*) = *take a xs* @ *xs* ! *a* # *drop* (*Suc a*) *xs*
by (*auto intro*: *id-take-nth-drop simp*: *verticesFrom-length*)
with *eq* **have** (*verticesFrom f v*) = *take a xs* @ *xs* ! *i* # *drop* (*Suc a*) *xs* **by** *simp*
moreover
from *xs-def i v d* **have** (*verticesFrom f v*) = *take i xs* @ *xs* ! *i* # *drop* (*Suc i*) *xs*
by (*auto intro*: *id-take-nth-drop simp*: *verticesFrom-length*)
ultimately have *take a xs* = *take i xs* **by** (*rule dist-at1*)
moreover
from *v d* **have** *vertFrom*[*simp*]: *length* (*vertices f*) = *length* (*verticesFrom f v*)
by (*auto simp*: *verticesFrom-length*)
from *xs-def a i* **have** $a <$ *length xs* $i <$ *length xs* **by** *auto*
moreover
have $\bigwedge a\ i.\ a <$ *length xs* $\Longrightarrow i <$ *length xs* $\Longrightarrow$ *take a xs* = *take i xs* $\Longrightarrow a = i$
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons x xs*) **then show** *?case*
apply (*cases a*) **apply** *auto*
apply (*cases i*) **apply** *auto*
apply (*cases i*) **by** *auto*
qed
ultimately show *?thesis* **by** *simp*
qed

lemma *a*: *distinct* (*vertices f*) $\Longrightarrow v \in \mathcal{V}\ f \Longrightarrow (\forall i \in$ *set is*. $i <$ *length* (*vertices f*)) $\Longrightarrow$
($\bigwedge a.\ a <$ *length* (*vertices f*) $\Longrightarrow$ *hideDupsRec* ((*f* · ^ *a*) *v*) [(*f* · ^ *k*) *v*. $k \in$ *is*] = *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*) $\Longrightarrow v \in \mathcal{V}\ f$ $\Longrightarrow$

(∀ *i* ∈ *set is*. *i* < *length* (*vertices f*)) ⟹ *is* ≠ [] ⟹
hd is = *0* ⟹ *indexToVertexList f v is* = *natToVertexList v f is*
apply (*cases is*) **by** (*auto simp*: *a* [**where** *a* = *0*, *simplified*] *indexToVertexList-def nextVertices-def*)

lemma *nvlr-length*: ⋀ *old*. (*length* (*natToVertexListRec old v f ls*)) = *length ls*
apply (*induct ls*) **by** *auto*

lemma *nvl-length*[*simp*]: *hd e* = *0* ⟹ *length* (*natToVertexList v f e*) = *length e*
apply (*cases e*)
by (*auto intro*: *nvlr-length*)

lemma *nvl-nvlRec*: *1* < *i* ⟹ *incrIndexList e i* (*length* (*vertices f*)) ⟹ *natToVertexList v f e* = (*Some v*) # *natToVertexListRec 0 v f* (*tl e*)
apply (*case-tac e*) **apply** *simp* **apply** (*subgoal-tac a* = *0*) **by** *auto*

lemma *natToVertexListRec-length*[*simp*]: ⋀ *e f*. *length* (*natToVertexListRec e v f es*) = *length es*
by (*induct es*) *auto*

lemma *natToVertexList-length*[*simp*]: *incrIndexList es* (*length es*) (*length* (*vertices f*)) ⟹
length (*natToVertexList v f es*) = *length es* **apply** (*case-tac es*) **by** *simp-all*

lemma *natToVertexList-nth-Suc*: *incrIndexList es* (*length es*) (*length* (*vertices f*)) ⟹ *Suc n* < *length es* ⟹
(*natToVertexList v f es*)!(*Suc n*) = (*if* (*es*!*n* = *es*!(*Suc n*)) *then None else* (*Some* $f^{(es!Suc\ n)}$ · *v*))
proof −
  assume *incr*: *incrIndexList es* (*length es*) (*length* (*vertices f*)) **and** *n*: *Suc n* < *length es*
  have *rec*: ⋀ *old n*. *Suc n* < *length es* ⟹
    (*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 ⋀ *old*. *natToVertexListRec old v f es* ! *Suc m* = (*if es* ! *m* = *es* ! *Suc m then None else Some* $f^{es\ !\ Suc\ m} \cdot v$)
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* $f^{es\ !\ Suc\ n} \cdot v$) **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*)) ⟹ *0* < *length es* ⟹
(*natToVertexList v f es*)!*0* = (*Some* $f^{(es!0)} \cdot v$) **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*)) ⟹ *hd* (*natToVertexList v f es*) = *Some v*
apply (*cases es*) **by** (*simp-all add*: *nextVertices-def*)

lemma *nth-last*[*intro*]: *Suc i* = *length xs* ⟹ *xs*!*i* = *last xs*
by (*cases xs rule*: *rev-exhaust*) *auto*

declare *incrIndexList-help4* [*simp del*]

lemma *natToVertexList-last*[*simp*]:
distinct (*vertices f*) ⟹ *v* ∈ $\mathcal{V}$ *f* ⟹ *incrIndexList es* (*length es*) (*length* (*vertices f*)) ⟹ *last* (*natToVertexList v f es*) = *Some* (*last* (*verticesFrom f v*))
proof −
assume *vors*: *distinct* (*vertices f*) *v* ∈ $\mathcal{V}$ *f* **and** *incr*: *incrIndexList es* (*length es*) (*length* (*vertices f*))
def *n′* ≡ *length es* − *2*
from *incr* **have** *1* < *length es* **by** *auto*
with *n′-def* **have** *n′l*: *Suc* (*Suc n′*) = *length es* **by** *arith*
from *incr n′l* **have** *last-ntvl*: (*natToVertexList v f es*)!(*Suc n′*) = *last* (*natToVertexList v f es*) **by** *auto*
from *n′l* **have** *last-es*: *es*!(*Suc n′*) = *last es* **by** *auto*

from *n'l* **have** *es*!*n'* = *last* (*butlast es*) **apply** (*cases es rule*: *rev-exhaust*) **by** (*auto simp*: *nth-append*)
with *last-es incr* **have** *less*: *es*!*n'* < *es*!(*Suc n'*) **by** *auto*
from *n'l* **have** *Suc n'* < *length es* **by** *arith*
with *incr less* **have** (*natToVertexList v f es*)!(*Suc n'*) = (*Some* $f^{(es!Suc\ n')} \cdot v$) **by** (*auto dest*: *natToVertexList-nth-Suc*)
with *incr last-ntvl last-es* **have** *rule1*: *last* (*natToVertexList v f es*) = *Some* $f^{((length\ (vertices\ f)) - (Suc\ 0))} \cdot v$ **by** *auto*

from *incr* **have** *lvf*: *1* < *length* (*vertices f*) **by** *auto*
with *vors* **have** *rule2*: *verticesFrom f v* ! ((*length* (*vertices f*)) − (*Suc 0*)) = $f^{((length\ (vertices\ f)) - (Suc\ 0))} \cdot v$ **apply** (*auto intro*!: *verticesFrom-nth*) **by** *arith*

from *vors lvf* **have** *verticesFrom f v* ! ((*length* (*vertices f*)) − (*Suc 0*)) = *last* (*verticesFrom f v*)
apply (*rule-tac nth-last*)
apply (*auto simp*: *verticesFrom-length*)
by *arith*
with *rule1 rule2* **show** *?thesis* **by** *auto*
qed

lemma *indexToVertexList-last*[*simp*]:
distinct (*vertices f*) $\Longrightarrow$ *v* $\in \mathcal{V}$ *f* $\Longrightarrow$ *incrIndexList es* (*length es*) (*length* (*vertices f*)) $\Longrightarrow$ *last* (*indexToVertexList f v es*) = *Some* (*last* (*verticesFrom f v*))
apply (*subgoal-tac indexToVertexList f v es* = *natToVertexList v f es*) **apply** *simp*
apply (*rule indexToVertexList-natToVertexList-eq*) **by** *auto*

lemma *sublist-take*: $\bigwedge$ *n iset*. $\forall$ *i* $\in$ *iset*. *i* < *n* $\Longrightarrow$ *sublist* (*take n xs*) *iset* = *sublist xs iset*
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*) **then show** *?case* **apply** (*simp add*: *sublist-Cons*) **apply** (*cases n*) **apply** *simp* **apply** (*simp add*: *sublist-Cons*) **apply** (*rule Cons*) **by** *auto*
qed

lemma *sublist-reduceIndices*: $\bigwedge$ *iset*. *sublist xs iset* = *sublist xs* {*i*. *i* < *length xs* $\wedge$ *i* $\in$ *iset*}
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*) **then**
have *sublist xs* {*j*. *Suc j* $\in$ *iset*} = *sublist xs* {*i*. *i* < *length xs* $\wedge$ *i* $\in$ {*j*. *Suc j* $\in$ *iset*}} **by** (*rule-tac Cons*)
then show *?case* **by** (*simp add*: *sublist-Cons*)
qed

lemma *natToVertexList-sublist1*: *distinct* (*vertices f*) $\Longrightarrow$

v ∈ 𝒱 *f* ⟹ *vs* = *verticesFrom f v* ⟹
incrIndexList es (*length es*) (*length vs*) ⟹ *n* ≤ *length es* ⟹
sublist (*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 *sublist* (*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*: *natToVertexList-nth-Suc split*: *split-if-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*) **apply** *auto* **by** *arith*
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* = *butlast es* @ [*last es*] **by** (*cases es rule*: *rev-exhaust*) *auto*

from *n′l* **have** *take* (*Suc n′*) (*butlast es* @ [*last es*]) = *butlast es* **by** *auto*
with *es-exh* **have** *take* (*Suc n′*) *es* = *butlast es* **by** *auto*
with *small-n Suc* **have** *take n′ es* @ [*es*!*n′*] = (*butlast es*) **by** (*simp add*: *take-Suc-conv-app-nth*)
with *small-n′* **have** *esn′-simps*: *take n′ es* = *butlast* (*butlast es*) ∧ *es*!*n′* =

```
last (butlast es)
          by (cases butlast es rule: rev-exhaust) auto

        from suc1 have last (butlast es) < 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 < length es by auto
         from suc1 le have es = take (Suc n) es @ es!(Suc n) # drop (Suc (Suc
n)) es by (auto intro: id-take-nth-drop)
       with suc1 have increasing (take (Suc n) es @ es!(Suc n) # drop (Suc (Suc
n)) es) by auto
        then have ∀ i ∈ (set (take (Suc n) es)). i ≤ es ! (Suc n) by (auto intro:
increasing2)
         with suc1 have ∀ i ∈ (set (take n es)). i ≤ es ! (Suc n) by (simp add:
take-Suc-conv-app-nth)
       then have seq: sublist (take (Suc (es ! Suc n)) (verticesFrom f v)) (set (take
n es))
          = sublist (verticesFrom f v) (set (take n es))
          apply (rule-tac sublist-take) by auto
        from suc1 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 ∀ i ∈ (set (take n es)). i ≤ es ! n by (auto intro: increasing2)
         with suc1 esn have ∀ i ∈ (set (take n es)). i ≤ es ! n' by (simp add:
take-Suc-conv-app-nth)
         with Suc have seq2: sublist (take (Suc (es ! n')) (verticesFrom f v)) (set
(take n es))
         = sublist (verticesFrom f v) (set (take n es))
          apply (rule-tac sublist-take) by auto
        from Suc suc1 have (insert (es ! n') (set (take n es))) = set (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 verticesFrom f v ≠ [] 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: sublist (verticesFrom f v) {0} = [v] by (auto simp: sublist-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
sublist-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*: *split-if-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*: $sublist\ ([((verticesFrom\ f\ v)!(es!n))])$
$\{j.\ j + (es!n) : (insert\ (es\ !\ n)\ (set\ (take\ n\ es)))\} = [v']$ **by** *auto*

from *suc1 smaller lvs* **have** *len*: $length\ (take\ (es\ !\ n)\ (verticesFrom\ f\ v)) = es!n$ **apply** *auto* **by** *arith*

have $\bigwedge x.\ x \in (set\ (take\ n\ es)) \Longrightarrow x < (es\ !\ n)$
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 $\bigwedge x.\ x \in set\ (take\ n'\ es) \Longrightarrow x \leq es!n'$ **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 $\{i.\ i < es\ !\ n \wedge i \in set\ (take\ n\ es)\} = (set\ (take\ n\ es))$ **by** *auto*
then have *elim-insert*: $\{i.\ i < es\ !\ n \wedge i \in insert\ (es\ !\ n)\ (set\ (take\ n\ es))\} = (set\ (take\ n\ es))$ **by** *auto*

have $sublist\ (take\ (es\ !\ n)\ (verticesFrom\ f\ v))\ (insert\ (es\ !\ n)\ (set\ (take\ n\ es))) =$

```
          sublist (take (es ! n) (verticesFrom f v)) {i. i < length (take (es ! n) (verticesFrom f v))
         ∧ i ∈ (insert (es ! n) (set (take n es)))} by (rule sublist-reduceIndices)
      with len have sublist (take (es ! n) (verticesFrom f v)) (insert (es ! n) (set (take n es))) =
         sublist (take (es ! n) (verticesFrom f v)) {i. i < (es ! n) ∧ i ∈ (insert (es ! n) (set (take n es)))}
        by simp
      with elim-insert have sub2: sublist (take (es ! n) (verticesFrom f v)) (insert (es ! n) (set (take n es))) =
         sublist (take (es ! n) (verticesFrom f v)) (set (take n es)) by simp

      def 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: ⋀x. x ∈ (set (take n es)) ⟹ x ≤  (es ! n′)
      proof −
        fix x
        assume x: x ∈ 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 ⋀ x. x ∈ set (take n′ es) ⟹ x ≤ es!n′ by (auto intro!: increasing2)
        with x Suc suc1 show x ≤ es!n′ by (auto simp: take-Suc-conv-app-nth)
      qed

      def m′ ≡ m − 1
      def Suc-es-n′ ≡ Suc (es!n′)

      from smaller smaller-n have Suc (es!n′) < length vs by auto
      then have min (length vs) (Suc (es ! n′)) = Suc (es!n′) by arith
      with Suc-es-n′-def have empty: {j. j + length (take Suc-es-n′ vs) ∈ 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 sublist (take (es ! n) vs) (set (take n es))
          = sublist (take (Suc-es-n′) vs @ take m′ (drop (Suc-es-n′) vs)) (set (take n es)) by auto
      then have sublist (take (es ! n) vs) (set (take n es)) =
        sublist (take (Suc-es-n′) vs) (set (take n es)) @
```

```
        sublist (take m' (drop (Suc-es-n') vs)) {j. j + length (take (Suc-es-n') vs)
: (set (take n es))}
        by (simp add: sublist-append)
      with empty Suc-es-n'-def have sublist (take (es ! n) vs) (set (take n es)) =
        sublist (take (Suc (es!n')) vs) (set (take n es)) by simp
      with suc1 sub2 have sub3: sublist (take (es ! n) (verticesFrom f v)) (insert
(es ! n) (set (take n es))) =
        sublist (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
        sublist (take (Suc (es ! n)) (verticesFrom f v)) (insert (es ! n) (set (take n
es))) =
         sublist (take (es ! n) (verticesFrom f v)) (insert (es ! n) (set (take n es)))
          @ sublist ([((verticesFrom f v)!(es!n))]) {j. j + (es!n) : (insert (es ! n)
(set (take n es)))}
        apply (auto simp: sublist-append) by arith
      with sub1 sub3 have sublist (take (Suc (es ! n)) (verticesFrom f v)) (insert
(es ! n) (set (take n es)))
       = sublist (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-sublist: distinct (vertices f) ⟹ v ∈ 𝒱 f ⟹
  incrIndexList es (length es) (length (vertices f)) ⟹
  sublist (verticesFrom f v) (set es) = removeNones (natToVertexList v f es)
proof -
  assume vors1: distinct (vertices f) v ∈ 𝒱 f
     incrIndexList es (length es) (length (vertices f))
  def 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 ∈ 𝒱 f
    vs = verticesFrom f v  incrIndexList es (length es) (length vs) by auto

  with lvs have vsne: vs ≠ [] by auto
  def n ≡ length es
  then have es!(n − 1) = last es
  proof (cases n)
    case 0 with n-def vors show ?thesis by (cases es) auto
  next
    case (Suc n')
    with n-def have small-n': n' < length es by arith
    from Suc n-def 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 *es*!*n′* = *last es* **by** (*cases es rule*: *rev-exhaust*) *auto*
with *Suc* **show** *?thesis* **by** *auto*
qed
with *vors* **have** *es*!(*n* − *1*) = (*length vs*) − *1* **by** *auto*
with *vsne* **have** *Suc* (*es*! (*n* − *1*)) = (*length vs*) **by** *auto*
then have *take-vs*: *take* (*Suc* (*es*!(*n* − *1*))) *vs* = *vs* **by** *auto*

from *n-def vors* **have** *n* = *length* (*natToVertexList v f es*) **by** *auto*
then have *take-nTVL*: *take n* (*natToVertexList v f es*) = *natToVertexList v f es* **by** *auto*

from *n-def* **have** *take-es*: *take n es* = *es* **by** *auto*

from *n-def* **have** *n* ≤ *length es* **by** *auto*
with *vors* **have** *sublist* (*take* (*Suc* (*es*!(*n* − *1*))) *vs*) (*set* (*take n es*))
= *removeNones* (*take n* (*natToVertexList v f es*)) **by** (*rule natToVertexList-sublist1*)
with *take-vs take-nTVL take-es vs-def* **show** *?thesis* **by** *simp*
qed

lemma *filter-Cons2*:
x ∉ *set ys* ⟹ [*y* ∈ *ys*. *y* = *x* ∨ *P y*] = [*y* ∈ *ys*. *P y*]
by (*induct ys*) (*auto*)

lemma *natToVertexList-removeNones*:
distinct (*vertices f*) ⟹ *v* ∈ 𝒱 *f* ⟹
incrIndexList es (*length es*) (*length* (*vertices f*)) ⟹
[*x* ∈ *verticesFrom f v*. *x* ∈ *set* (*removeNones* (*natToVertexList v f es*))]
= *removeNones* (*natToVertexList v f es*)
proof −
assume *vors*: *distinct* (*vertices f*) *v* ∈ 𝒱 *f*
incrIndexList es (*length es*) (*length* (*vertices f*))
then have *dist*: *distinct* (*verticesFrom f v*) **by** *auto*
from *vors* **have** *sub-eq*: *sublist* (*verticesFrom f v*) (*set es*)
= *removeNones* (*natToVertexList v f es*) **by** (*rule natToVertexList-sublist*)
from *dist* **have** [*x* ∈ *verticesFrom f v*.
x ∈ *set* (*sublist* (*verticesFrom f v*) (*set es*))] = *removeNones* (*natToVertexList v f es*)
apply (*simp add*: *filter-in-sublist*)
by (*simp add*: *sub-eq*)
with *sub-eq* **show** *?thesis* **by** *simp*
qed

lemma *indexToVertexList-removeNones*:
distinct (*vertices f*) ⟹ *v* ∈ 𝒱 *f* ⟹
incrIndexList es (*length es*) (*length* (*vertices f*)) ⟹
[*x* ∈ *verticesFrom f v*. *x* ∈ *set* (*removeNones* (*indexToVertexList f v es*))]
= *removeNones* (*indexToVertexList f v es*)
apply (*subgoal-tac indexToVertexList f v es* = *natToVertexList v f es*)

apply *simp* **apply** (*rule natToVertexList-removeNones*) **apply** *assumption+*
apply (*rule indexToVertexList-natToVertexList-eq*) **apply** (*simp-all*) **by** *auto*

constdefs *is-duplicateEdge* :: *graph* ⇒ *face* ⇒ *vertex* ⇒ *vertex* ⇒ *bool*
is-duplicateEdge g f a b ≡
  $((a,\ b) \in edges\ g \land (a,\ b) \notin edges\ f \land (b,\ a) \notin edges\ f)$
$\lor\ ((b,\ a) \in edges\ g \land (b,\ a) \notin edges\ f \land (a,\ b) \notin edges\ f)$

constdefs *invalidVertexList* :: *graph* ⇒ *face* ⇒ *vertex option list* ⇒ *bool*
invalidVertexList g f vs ≡
    $\exists\, i < |vs| - 1.$
        case vs!*i of None* ⇒ *False*
            | *Some a* ⇒ *case vs*!(*i*+*1*) *of None* ⇒ *False*
            | *Some b* ⇒ *is-duplicateEdge g f a b*

12.15 *pre-subdivFace*(′)

constdefs *pre-subdivFace-face* :: *face* ⇒ *vertex* ⇒ *vertex option list* ⇒ *bool*
pre-subdivFace-face f v′ vOptionList ≡
    [*v* ∈ *verticesFrom f v′*. *v* ∈ *set* (*removeNones vOptionList*)]
      = (*removeNones vOptionList*)
  ∧ ¬ *final f* ∧ *distinct* (*vertices f*)
  ∧ *hd* (*vOptionList*) = *Some v′*
  ∧ $v' \in \mathcal{V}\ f$
  ∧ *last* (*vOptionList*) = *Some* (*last* (*verticesFrom f v′*))
  ∧ *hd* (*tl* (*vOptionList*)) ≠ *last* (*vOptionList*)
  ∧ *2* < | *vOptionList* |
  ∧ *vOptionList* ≠ []
  ∧ *tl* (*vOptionList*) ≠ []

constdefs *pre-subdivFace* :: *graph* ⇒ *face* ⇒ *vertex* ⇒ *vertex option list* ⇒ *bool*
pre-subdivFace g f v′ vOptionList ≡
  pre-subdivFace-face f v′ vOptionList ∧ ¬ *invalidVertexList g f vOptionList*

constdefs *pre-subdivFace′* :: *graph* ⇒ *face* ⇒ *vertex* ⇒ *vertex* ⇒ *nat* ⇒ *vertex option list* ⇒ *bool*
pre-subdivFace′ g f v′ ram1 n vOptionList ≡
  ¬ *final f* ∧ $v' \in \mathcal{V}\ f \land ram1 \in \mathcal{V}\ f$
  ∧ *v′* ∉ *set* (*removeNones vOptionList*)
  ∧ *distinct* (*vertices f*)
  ∧ (
    [*v* ∈ *verticesFrom f v′*. *v* ∈ *set* (*removeNones vOptionList*)]
       = (*removeNones vOptionList*)
  ∧ *before* (*verticesFrom f v′*) *ram1* (*hd* (*removeNones vOptionList*))
  ∧ *last* (*vOptionList*) = *Some* (*last* (*verticesFrom f v′*))

∧ *vOptionList* ≠ []
∧ ((*v′* = *ram1* ∧ (*0* < *n*)) ∨ ((*v′* = *ram1* ∧ (*hd* (*vOptionList*) ≠ *Some* (*last* (*verticesFrom f v′*)))) ∨ (*v′* ≠ *ram1*)))
∧ ¬ *invalidVertexList g f vOptionList*
∧ (*n* = *0* ∧ *hd* (*vOptionList*) ≠ *None* ⟶ ¬ *is-duplicateEdge g f ram1* (*the* (*hd* (*vOptionList*))))
∨ (*vOptionList* = [] ∧ *v′* ≠ *ram1*)
)

lemma *pre-subdivFace-face-in-f* [*intro*]: *pre-subdivFace-face f v ls* ⟹ *Some a* ∈ *set ls* ⟹ *a* ∈ *set* (*verticesFrom f v*)
apply (*subgoal-tac a* ∈ *set* (*removeNones ls*)) **apply** (*auto simp*: *pre-subdivFace-face-def*)
apply (*subgoal-tac a* ∈ *set* [*v*∈*verticesFrom f v* . *v* ∈ *set* (*removeNones ls*)])
apply (*thin-tac* [*v*∈*verticesFrom f v* . *v* ∈ *set* (*removeNones ls*)] = *removeNones ls*) **by** *auto*

lemma *pre-subdivFace-in-f* [*intro*]: *pre-subdivFace g f v ls* ⟹ *Some a* ∈ *set ls* ⟹ *a* ∈ *set* (*verticesFrom f v*)
by (*auto simp*: *pre-subdivFace-def*)

lemma *pre-subdivFace-face-in-f′* [*intro*]: *pre-subdivFace-face f v ls* ⟹ *Some a* ∈ *set ls* ⟹ *a* ∈ 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 *hilf1*: *a* ∈ *set b* ⟹ (*v* = *a* ∨ *v* ∈ *set b*) = (*v* ∈ *set b*) **by** *auto*

lemma *filter-congs-shorten1′*: *distinct* (*vertices f*) ⟹ [*v*∈*vertices f* . *v* = *a* ∨ *v* ∈ *set vs*] ≅ (*a* # *vs*)
⟹ [*v*∈*vertices f* . *v* ∈ *set vs*] ≅ *vs*
proof −
assume *dist*: *distinct* (*vertices f*) **and** [*v*∈*vertices f* . *v* = *a* ∨ *v* ∈ *set vs*] ≅ (*a* # *vs*)
then obtain *n* **where** *removeNones-vol*: *a* # *vs* = *rotate n* [*v*∈*vertices f* . *v* = *a* ∨ *v* ∈ *set vs*]
by (*auto simp*: *congs-def*)
have ∃ *m*. *rotate n* [*v*∈*vertices f* . *v* = *a* ∨ *v* ∈ *set vs*] = [*v*∈(*rotate m* (*vertices f*)) . *v* = *a* ∨ *v* ∈ *set vs*]
by (*auto intro*: *rotate-help6*)
then obtain *m* **where** *rotate n* [*v*∈*vertices f* . *v* = *a* ∨ *v* ∈ *set vs*]
= [*v*∈(*rotate m* (*vertices f*)) . *v* = *a* ∨ *v* ∈ *set vs*] **by** *auto*
with *removeNones-vol* **have** *a-removeNones-vol*: [*v*∈(*rotate m* (*vertices f*)) . *v* = *a* ∨ *v* ∈ *set vs*] = *a* # *vs* **by** *simp*

have *rule1*: ⋀ *vs a ys*. *distinct vs* ⟹ [*v*∈*vs* . *v* = *a* ∨ *v* ∈ *set ys*] = *a* # *ys* ⟹ [*v* ∈ *vs*. *v* ∈ *set ys*] = *ys*
proof −
fix *vs a ys*
assume *dist*: *distinct vs* **and** *ays*: [*v*∈*vs* . *v* = *a* ∨ *v* ∈ *set ys*] = *a* # *ys*
then have *distinct* ([*v*∈*vs* . *v* = *a* ∨ *v* ∈ *set ys*]) **by** (*rule-tac distinct-filter*)
with *ays* **have** *distys*: *distinct* (*a* # *ys*) **by** *simp*
from *dist distys ays* **show** [*v* ∈ *vs*. *v* ∈ *set ys*] = *ys*
apply (*induct vs*) **by** (*auto split*: *split-if-asm simp*: *filter-Cons2*)
qed

from *dist* **have** *distinct* (*rotate m* (*vertices f*)) **by** *auto*
with *a-removeNones-vol* **have** [*v* ∈ (*rotate m* (*vertices f*)). *v* ∈ *set vs*] = *vs* **by** (*rule-tac rule1*)
also have ∃ *l*. [*v* ∈ (*rotate m* (*vertices f*)). *v* ∈ *set vs*] = *rotate l* [*v* ∈ *vertices f*. *v* ∈ *set vs*]
by (*rule rotate-help5*)
then obtain *l* **where** [*v* ∈ (*rotate m* (*vertices f*)). *v* ∈ *set vs*] = *rotate l* [*v* ∈ *vertices f*. *v* ∈ *set vs*] **by** *auto*
ultimately have *vs* = *rotate l* [*v*∈*vertices f* . *v* ∈ *set vs*] **by** *simp*
thus *?thesis* **by** (*auto simp*: *congs-def*)
qed

lemma *filter-congs-shorten1*: *distinct* (*verticesFrom f v*) ⟹ [*v*∈*verticesFrom f v* . *v* = *a* ∨ *v* ∈ *set vs*] = (*a* # *vs*)
⟹ [*v*∈*verticesFrom f v* . *v* ∈ *set vs*] = *vs*
proof −
assume *dist*: *distinct* (*verticesFrom f v*) **and** *eq*: [*v*∈*verticesFrom f v* . *v* = *a* ∨ *v* ∈ *set vs*] = (*a* # *vs*)
have *rule1*: ⋀ *vs a ys*. *distinct vs* ⟹ [*v*∈*vs* . *v* = *a* ∨ *v* ∈ *set ys*] = *a* # *ys* ⟹ [*v* ∈ *vs*. *v* ∈ *set ys*] = *ys*
proof −
fix *vs a ys*
assume *dist*: *distinct vs* **and** *ays*: [*v*∈*vs* . *v* = *a* ∨ *v* ∈ *set ys*] = *a* # *ys*
then have *distinct* ([*v*∈*vs* . *v* = *a* ∨ *v* ∈ *set ys*]) **by** (*rule-tac distinct-filter*)
with *ays* **have** *distys*: *distinct* (*a* # *ys*) **by** *simp*
from *dist distys ays* **show** [*v* ∈ *vs*. *v* ∈ *set ys*] = *ys*
apply (*induct vs*) **by** (*auto split*: *split-if-asm simp*: *filter-Cons2*)
qed

from *dist eq* **show** *?thesis* **by** (*rule-tac rule1*)
qed

lemma *ovl-shorten'*: *distinct* (*vertices f*) ⟹ [*v*∈*vertices f* . *v* ∈ *set* (*removeNones* (*va* # *vol*))] ≅ (*removeNones* (*va* # *vol*))
⟹ [*v*∈*vertices f* . *v* ∈ *set* (*removeNones* (*vol*))] ≅ (*removeNones* (*vol*))
proof −
assume *dist*: *distinct* (*vertices f*) **and** *vors*: [*v*∈*vertices f* . *v* ∈ *set* (*removeNones*

(*va* # *vol*))] ≅ (*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 *ovl-shorten*: *distinct* (*verticesFrom f v*) ⟹
[*v*∈*verticesFrom f v* . *v* ∈ *set* (*removeNones* (*va* # *vol*))] = (*removeNones* (*va* # *vol*))
⟹ [*v*∈*verticesFrom f v* . *v* ∈ *set* (*removeNones* (*vol*))] = (*removeNones* (*vol*))
proof −
assume *dist*: *distinct* (*verticesFrom f v*)
and *vors*: [*v*∈*verticesFrom f v* . *v* ∈ *set* (*removeNones* (*va* # *vol*))] = (*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*: *pre-subdivFace-face f v vol* ⟹ *distinct* (*removeNones vol*)
apply (*auto dest*!: *verticesFrom-distinct simp*: *pre-subdivFace-face-def*)
apply (*subgoal-tac distinct* ([*v*∈*verticesFrom f v* . *v* ∈ *set* (*removeNones vol*)]))
apply *simp*
apply (*thin-tac* [*v*∈*verticesFrom f v* . *v* ∈ *set* (*removeNones vol*)] = *removeNones vol*) **by** *auto*

lemma *pre-subdivFace-face-not-empty*: *pre-subdivFace-face f v* (*vo* # *vol*) ⟹ *vol* ≠ []
by (*simp split*: *split-if-asm add*: *pre-subdivFace-face-def*)

lemma *invalidVertexList-shorten*: *invalidVertexList g f vol* ⟹ *invalidVertexList g f* (*v* # *vol*)
apply (*simp add*: *invalidVertexList-def*) **apply** *auto* **apply** (*rule exI*) **apply** *safe*
apply (*subgoal-tac* (*Suc i*) < | *vol* |) **apply** *assumption* **apply** *arith*
apply *auto* **apply** (*case-tac vol*!*i*) **by** *auto*

lemma *edges-in-faces-in-graph*: *x* ∈ *edges f* ⟹ *f* ∈ $\mathcal{F}$ *g* ⟹ *x* ∈ *edges g*
apply (*simp add*: *edges-graph-def*) **by** *auto*

lemma *pre-subdivFace-pre-subdivFace′*: *v* ∈ $\mathcal{V}$ *f* ⟹ *pre-subdivFace g f v* (*vo* # *vol*) ⟹
pre-subdivFace′ g f v v 0 (*vol*)

proof −
assume *vors*: *v* ∈ 𝒱 *f* *pre-subdivFace g f v* (*vo* # *vol*)
then have *vors'*: *v* ∈ 𝒱 *f pre-subdivFace-face f v* (*vo* # *vol*) ¬ *invalidVertexList g f* (*vo* # *vol*)
by (*auto simp*: *pre-subdivFace-def*)
then have *r*: *removeNones vol* ≠ [] **apply** (*cases vol rule*: *rev-exhaust*) **by** (*auto simp*: *pre-subdivFace-face-def*)
then have *Some* (*hd* (*removeNones vol*)) ∈ *set vol* **apply** (*induct vol*) **apply** *auto* **apply** (*case-tac a*) **by** *auto*
then have *Some* (*hd* (*removeNones vol*)) ∈ *set* (*vo* # *vol*) **by** *auto*
with *vors'* **have** *hd*: *hd* (*removeNones vol*) ∈ 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* ∉ *set* (*tl* (*removeNones* (*vo* # *vol*))) **apply** (*drule-tac pre-subdivFace-face-distinct*) **by** *auto*
with *vors' r* **have** *ne*: *v* ≠ *hd* (*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*: ¬ *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* ⟹ *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*∈*verticesFrom f v′* . *v* ∈ *set* (*removeNones* (*a* # *list*))]) **apply** *force*
apply (*thin-tac* [*v*∈*verticesFrom f v′* . *v* ∈ *set* (*removeNones* (*a* # *list*))] = *removeNones* (*a* # *list*))
by *auto*

lemma *natToVertexList-pre-subdivFace-face*:
¬ *final f* ⟹ *distinct* (*vertices f*) ⟹ *v* ∈ 𝒱 *f* ⟹ *2* < |*es*| ⟹
incrIndexList es (*length es*) (*length* (*vertices f*)) ⟹
pre-subdivFace-face f v (*natToVertexList v f es*)
proof −
assume *vors*: ¬ *final f distinct* (*vertices f*) *v* ∈ 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* < | *natToVertexList v f es* |
by *auto*

from *vors* **have** *distinct* [*x*∈*verticesFrom f v* . *x* ∈ *set* (*removeNones* (*natToVertexList v f es*))] **by** *auto*
with *vors* **have** *distinct* (*removeNones* (*natToVertexList v f es*)) **by** (*simp add*: *natToVertexList-removeNones*)
with *nvl-l lastOvl* **have** *hd-last*: *hd* (*tl* (*natToVertexList v f es*)) ≠ *last* (*natToVertexList v f 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*: *split-if-asm*)
qed

lemma *indexToVertexList-pre-subdivFace-face*:
¬ *final f* ⟹ *distinct* (*vertices f*) ⟹ *v* ∈ 𝒱 *f* ⟹ *2* < |*es*| ⟹
incrIndexList es (*length es*) (*length* (*vertices f*)) ⟹
pre-subdivFace-face f v (*indexToVertexList f v 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*: *pre-subdivFace g f v vol* ⟹ *subdivFace g f vol*

= *subdivFace′ g f v 0* (*tl vol*)
by (*simp-all add*: *subdivFace-def pre-subdivFace-def pre-subdivFace-face-def*)

lemma *pre-subdivFace′-None*:
pre-subdivFace′ g f v′ v n (*None* # *vol*) ⟹
pre-subdivFace′ g f v′ v (*Suc n*) *vol*
by(*auto simp*: *pre-subdivFace′-def dest*:*invalidVertexList-shorten*
split:*split-if-asm*)

declare *verticesFrom-between* [*simp del*]

lemma *verticesFrom-split*: *v* # *tl* (*verticesFrom f v*) = *verticesFrom f v* **by** (*auto simp*: *verticesFrom-Def*)

lemma *verticesFrom-v*: *distinct* (*vertices f*) ⟹ *vertices f* = *a* @ *v* # *b* ⟹ *verticesFrom f v* = *v* # *b* @ *a*
by (*simp add*: *verticesFrom-Def*)

lemma *splitAt-fst*[*simp*]: *distinct xs* ⟹ *xs* = *a* @ *v* # *b* ⟹ *fst* (*splitAt v xs*) = *a*
by *auto*

lemma *splitAt-snd*[*simp*]: *distinct xs* ⟹ *xs* = *a* @ *v* # *b* ⟹ *snd* (*splitAt v xs*) = *b*
by *auto*

lemma *verticesFrom-splitAt-v-fst*[*simp*]:
distinct (*verticesFrom f v*) ⟹ *fst* (*splitAt v* (*verticesFrom f v*)) = []
by (*simp add*: *verticesFrom-Def*)
lemma *verticesFrom-splitAt-v-snd*[*simp*]:
distinct (*verticesFrom f v*) ⟹ *snd* (*splitAt v* (*verticesFrom f v*)) = *tl* (*verticesFrom f v*)
by (*simp add*: *verticesFrom-Def*)

lemma *filter-distinct-at*:
distinct xs ⟹ *xs* = (*as* @ *u* # *bs*) ⟹ [*v* ∈ *xs*. *v* = *u* ∨ *P v*] = *u* # *us* ⟹
[*v* ∈ *bs*. *P v*] = *us* ∧ [*v* ∈ *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-at2*: *distinct xs* ⟹ *xs* = (*as* @ *u* # *bs*) ⟹
[*v* ∈ *xs*. *v* = *u* ∨ *P v*] = *u* # *us* ⟹ *filter P zs* = [] ⟹ [*v* ∈ *zs*@*bs*. *P v*] = *us*
apply (*drule filter-distinct-at*) **apply** *assumption*+ **by** *simp*

lemma *filter-distinct-at3*: *distinct xs* ⟹ *xs* = (*as* @ *u* # *bs*) ⟹
[*v* ∈ *xs*. *v* = *u* ∨ *P v*] = *u* # *us* ⟹ ∀ *z* ∈ *set zs*. *z* ∈ *set as* ∨ ¬ (*P z*) ⟹
[*v* ∈ *zs*@*bs*. *P v*] = *us*
apply (*drule filter-distinct-at*) **apply** *assumption*+ **apply** *simp*
by (*induct zs*) *auto*

lemma *filter-distinct-at4*: *distinct xs* ⟹ *xs* = (*as* @ *u* # *bs*)
⟹ [*v* ∈ *xs*. *v* = *u* ∨ *v* ∈ *set us*] = *u* # *us*
⟹ *set zs* ∩ *set us* ⊆ {*u*} ∪ *set as*
⟹ [*v* ∈ *zs*@*bs*. *v* ∈ *set us*] = *us*
proof −
assume *vors*: *distinct xs xs* = (*as* @ *u* # *bs*)
[*v* ∈ *xs*. *v* = *u* ∨ *v* ∈ *set us*] = *u* # *us*
set zs ∩ *set us* ⊆ {*u*} ∪ *set as*
then have *distinct* ([*v* ∈ *xs*. *v* = *u* ∨ *v* ∈ *set us*]) **apply** (*rule-tac distinct-filter*)
by *simp*
with *vors* **have** *dist*: *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* ⟹ *xs* = (*as* @ *u* # *bs*)
⟹ [*v* ∈ *xs*. *v* = *u* ∨ *v* ∈ *set us*] = *u* # *us*
⟹ *set zs* ∩ *set xs* ⊆ {*u*} ∪ *set as*
⟹ [*v* ∈ *zs*@*bs*. *v* ∈ *set us*] = *us*
proof −
assume *vors*: *distinct xs xs* = (*as* @ *u* # *bs*)
[*v* ∈ *xs*. *v* = *u* ∨ *v* ∈ *set us*] = *u* # *us*
set zs ∩ *set xs* ⊆ {*u*} ∪ *set as*
have *set* ([*v* ∈ *xs*. *v* = *u* ∨ *v* ∈ *set us*]) ⊆ *set xs* **by** *auto*
with *vors* **have** *set* (*u* # *us*) ⊆ *set xs* **by** *simp*
then have *set us* ⊆ *set xs* **by** *simp*
with *vors* **have** *set zs* ∩ *set us* ⊆ *set zs* ∩ *insert u* (*set as* ∪ *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* ⟹ *xs* = (*as* @ *u* # *bs*)
⟹ [*v* ∈ *xs*. *v* = *u* ∨ *v* ∈ *set us*] = *u* # *us*
⟹ *set zs* ∩ *set xs* ⊆ {*u*} ∪ *set as*
⟹ [*v* ∈ *zs*@*bs*. *v* ∈ *set us*] = *us* ∧ [*v* ∈ *bs*. *v* ∈ *set us*] = *us*
proof −
assume *vors*: *distinct xs xs* = (*as* @ *u* # *bs*)
[*v* ∈ *xs*. *v* = *u* ∨ *v* ∈ *set us*] = *u* # *us*
set zs ∩ *set xs* ⊆ {*u*} ∪ *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 ⟹ xs = (as @ u # bs)
  ⟹ [v ∈ xs. v = u ∨ v ∈ set us] = u # us
  ⟹ set zs ∩ set xs ⊆ {u} ∪ set as
  ⟹ us = hd-us # tl-us
  ⟹ [v ∈ zs@bs. v ∈ set us] = us ∧ hd-us ∈ set bs
proof −
  assume vors: distinct xs xs = (as @ u # bs)
    [v ∈ xs. v = u ∨ v ∈ set us] = u # us
    set zs ∩ set xs ⊆ {u} ∪ set as
    us = hd-us # tl-us
  then have [v ∈ zs@bs. v ∈ set us] = us ∧ [v ∈ bs. v ∈ 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) ⊆ set bs) apply simp
   apply (subgoal-tac set [v∈bs . v = hd-us ∨ v ∈ set tl-us] ⊆ 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*) ≅ (*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* ∈ *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* ≠ *v′* ⟹ *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* ≠ *v′* ⟹ *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′* ∈ 𝒱 *f21* **by** *auto*
with *dist-f21* **have** *dist-f21-v′*: *distinct* (*verticesFrom f21 v′*) **by** *auto*

ultimately have *m1*: *v* ≠ *v′* ⟹ *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′* ⟹ *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* ∈ *set* (*verticesFrom f v′*) **by** (*fastsimp 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*∈*snd* (*splitAt u* (*verticesFrom f v′*)) . *v* ∈ *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* [] ∩ *set* (*verticesFrom f v′*) ⊆ {*u*} ∪ *set* (*fst* (*splitAt u* (*verticesFrom f v′*))) **by** *auto*
from *split-u dist-vf-v′ pre-add*
have [*v*∈ [] @ *snd* (*splitAt u* (*verticesFrom f v′*)) . *v* ∈ *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*: ⋀ *x*. *x* ∈ *set* (*removeNones vol*) ⟹ *x* ∈ *set* (*snd* (*splitAt u* (*verticesFrom f v′*)))
apply (*subgoal-tac x* ∈ *set* [*v*∈*snd* (*splitAt u* (*verticesFrom f v′*)) . *v* ∈ *set* (*removeNones vol*)] ⟹
x ∈ *set* (*snd* (*splitAt u* (*verticesFrom f v′*))))
apply *force* **apply** (*thin-tac* [*v*∈*snd* (*splitAt u* (*verticesFrom f v′*)) . *v* ∈ *set* (*removeNones vol*)] = *removeNones vol*)
by *simp*

from *split-u dist-vf-v′* **have** *notinFst-u*: ⋀ *x*. *x* ∈ *set* (*removeNones vol*) ⟹
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** ⋀ *a b*. *is-nextElem* (*vertices f*) *a b* ∧ *a* ∈ *set* (*removeNones vol*) ∧ *b* ∈ *set* (*removeNones vol*) ⟹
is-nextElem (*vertices f21*) *a b*
proof −
fix *a b*
assume *vors*: *is-nextElem* (*vertices f*) *a b* ∧ *a* ∈ *set* (*removeNones vol*) ∧ *b* ∈ *set* (*removeNones vol*)
def *vor-u* ≡ *fst* (*splitAt u* (*verticesFrom f v′*))
def *nach-u* ≡ *snd* (*splitAt u* (*verticesFrom f v′*))
from *vors v′* **have** *is-nextElem* (*verticesFrom f v′*) *a b* **by** (*simp add*: *verticesFrom-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:split-if-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* ∧ *a* ∈ *set* (*removeNones vol*) ∧ *b* ∈ *set* (*removeNones vol*) ⟹
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′*:
⋀ *a b*. (*a*,*b*) ∈ *edges f* ∧ *a* ∈ *set* (*removeNones vol*) ∧ *b* ∈ *set* (*removeNones vol*) ⟹ (*a*, *b*) ∈ *edges f21*
by (*simp add*: *is-nextElem-edges-eq pre-subdivFace′-def*)

have *rule1*: [*v*∈*verticesFrom f21 v′* . *v* ∈ *set* (*removeNones vol*)]
= *removeNones vol* ∧ *hd* (*removeNones vol*) ∈ *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* ≠ *v′* ⟹ *fst* (*splitAt u* (*verticesFrom f v′*))
= *v′* # *tl* (*fst* (*splitAt u* (*verticesFrom f v′*))) **by** (*cases fst* (*splitAt u* (*verticesFrom f v′*))) *auto*
moreover
have *v′* ∈ *set* (*v′* # *tl* (*fst* (*splitAt u* (*verticesFrom f v′*)))) **by** *simp*
ultimately have *u* ≠ *v′* ⟹ *v′* ∈ *set* (*fst* (*splitAt u* (*verticesFrom f v′*))) **by** *simp*
moreover
from *pre-fdg* **have** *set* (*v′* # *ws* @ [*u*]) ∩ *set* (*verticesFrom f v′*) ⊆ {*v′*, *u*}
apply (*simp add*: *set-eq*)
by (*unfold pre-splitFace-def*) *auto*
ultimately have *help*: *set* (*v′* # *ws* @ [*u*]) ∩ *set* (*verticesFrom f v′*)
⊆ {*u*} ∪ *set* (*fst* (*splitAt u* (*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*∈ (*v′* # *ws* @ [*u*]) @ *snd* (*splitAt u* (*verticesFrom f v′*)) . *v* ∈ *set* (*removeNones vol*)]
= *removeNones vol* ∧ *hd* (*removeNones vol*) ∈ *set* (*snd* (*splitAt u* (*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* ≠ *v′* **by** *auto*
def *fst-u* ≡ *fst* (*splitAt u* (*verticesFrom f v′*))

```
    def fst-v ≡  fst (splitAt v (verticesFrom f v′))

    from pre-add u dist-vf-v′ have v ∈ set (fst (splitAt u (verticesFrom f v′)))
      apply (rule-tac before-dist-r1) by (auto  simp: pre-subdivFace′-def)
    with fst-u-def have fst-u = fst (splitAt v (fst (splitAt u (verticesFrom f v′))))
        @ v # snd (splitAt v (fst (splitAt u (verticesFrom f v′))))
      by (auto dest: splitAt-ram)
    with pre-add fst-v-def pre-bet′ have fst-u′:fst-u
      = fst-v @ v # snd (splitAt v (fst (splitAt u (verticesFrom f v′)))) by (simp
add: pre-subdivFace′-def)

     from pre-fdg have set (v # ws @ [u]) ∩ set (verticesFrom f v′) ⊆  {v, u}
apply (simp add: set-eq)
      by (unfold pre-splitFace-def) auto

   with fst-u′ have set (v # ws @ [u]) ∩ set (verticesFrom f v′) ⊆ {u} ∪ set fst-u
by auto
    moreover
    from fst-u′ have set fst-v ⊆ set fst-u by auto
    ultimately
     have (set (v # ws @ [u]) ∪ set fst-v) ∩ set (verticesFrom f v′) ⊆ {u} ∪ set
fst-u by auto
    with fst-u-def fst-v-def
   have set (fst (splitAt v (verticesFrom f v′)) @ v # ws @ [u]) ∩ set (verticesFrom
f v′)
      ⊆ {u} ∪ set (fst (splitAt u (verticesFrom f v′))) by auto
    moreover
    with False vv′ have  v′ # tl (fst (splitAt v (verticesFrom f v′)))
      = fst (splitAt v (verticesFrom f v′)) by auto
    ultimately have set ((v′ # tl (fst (splitAt v (verticesFrom f v′)))) @ v # ws
@ [u]) ∩ set (verticesFrom f v′)
      ⊆ {u} ∪ set (fst (splitAt u (verticesFrom f v′)))
     by (simp only:)
    then have help: set (v′ # tl (fst (splitAt v (verticesFrom f v′))) @ v # ws @
[u]) ∩ set (verticesFrom f v′)
      ⊆ {u} ∪ set (fst (splitAt u (verticesFrom f v′))) by auto


    from split-u dist-vf-v′ pre-add pre-fdg removeNones-split have
      [v∈ (v′ # tl (fst (splitAt v (verticesFrom f v′))) @ v # ws @ [u])
         @ snd (splitAt u (verticesFrom f v′)) . v ∈ set (removeNones vol)]
     = removeNones vol ∧ hd (removeNones vol) ∈ set (snd (splitAt u (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** (*hd* (*removeNones vol*)) ∈ *set* (*snd* (*splitAt u* (*verticesFrom f v'*))) **by** *auto*

with *m1 m2 dist-f21-v'* **have** *rule3*: *before* (*verticesFrom f21 v'*) *u* (*hd* (*removeNones vol*))

proof −

assume *hd-ram*: (*hd* (*removeNones vol*)) ∈ *set* (*snd* (*splitAt u* (*verticesFrom f v'*)))

from *m1 m2 dist-f21-v'* **have** *distinct* (*snd* (*splitAt u* (*verticesFrom f v'*))) **apply** (*cases v* = *v'*)

by *auto*

moreover

def *z1* ≡ *fst* (*splitAt* (*hd* (*removeNones vol*)) (*snd* (*splitAt u* (*verticesFrom f v'*))))

def *z2* ≡ *snd* (*splitAt* (*hd* (*removeNones vol*)) (*snd* (*splitAt u* (*verticesFrom f v'*))))

note *z1-def z2-def hd-ram*

ultimately have *snd* (*splitAt u* (*verticesFrom f v'*)) = *z1* @ (*hd* (*removeNones 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*)) ∈ *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*: ⋀ *a b f v*. *v* ∈ *set* (*vertices f*) ⟹ *is-nextElem* (*vertices f*) *a b* = *is-nextElem* (*verticesFrom f v*) *a b*
apply (*rule is-nextElem-congs-eq*) **by** (*rule verticesFrom-congs*)

def *f12* ≡ *fst* (*split-face 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* (*split-face 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*: ⋀ *x*. *x* ∈ *set* (*removeNones vol*) ⟹ *x* ∉ *set* (*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* ∈ *set ws* ∩ *set* (*snd* (*splitAt u* (*verticesFrom f v′*))))
apply *simp* **apply** (*elim conjE*)
apply (*thin-tac set ws* ∩ *set* (*snd* (*splitAt u* (*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* ∈ *set ws* ∩ *set* (*snd* (*splitAt u* (*verticesFrom f v′*))))
apply *simp* **apply** (*elim conjE*)
apply (*thin-tac set ws* ∩ *set* (*snd* (*splitAt u* (*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* $\in$ *set* (*removeNones vol*) $\Longrightarrow$ (*x*,*y*) $\notin$ *edges f12*
apply (*drule-tac removeNones-vol-not-f12*) **by** *auto*
from *dist-f12* **have** *removeNones-vol-edges-not-f12'*: $\bigwedge$ *x y*. *y* $\in$ *set* (*removeNones vol*) $\Longrightarrow$ (*x*,*y*) $\notin$ *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*) $\in$ *edges g'* $\wedge$ (*a*,*b*) $\notin$ *edges g*
$\wedge$ *a* $\in$ *set* (*removeNones vol*) $\wedge$ *b* $\in$ *set* (*removeNones vol*) $\Longrightarrow$ (*a*, *b*) $\in$ *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* $\in$ *set* (*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*) $\in$ *edges g'* $\wedge$ (*a*,*b*) $\notin$ *edges g*
$\wedge$ *a* $\in$ *set* (*removeNones vol*) $\wedge$ *b* $\in$ *set* (*removeNones vol*) $\Longrightarrow$ (*a*, *b*) $\in$ *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* $\neq$ [] $\Longrightarrow$ *edges g'* = *edges g* $\cup$ *Edges ws* $\cup$ *Edges*(*rev ws*) $\cup$
{(*u*, *last ws*), (*hd ws*, *v*), (*v*, *hd ws*), (*last ws*, *u*)}
apply (*rule-tac splitFace-edges-g'*) **apply** *simp*
apply (*subgoal-tac* (*f12*, *f21*, *g'*) = *splitFace g v u f ws*) **apply** *assumption* **by** *auto*

from *pre-fdg fdg f12-fdg g'* **have** *edges-g'2*: *ws* = [] $\Longrightarrow$ *edges g'* = *edges g* $\cup$
{(*v*, *u*), (*u*, *v*)}
apply (*rule-tac splitFace-edges-g'-vs*) **apply** *simp*
apply (*subgoal-tac* (*f12*, *f21*, *g'*) = *splitFace g v u f* []) **apply** *assumption* **by** *auto*

from *f12-split-face f21-split-face* **have** *split*: (*f12*,*f21*) = *split-face f v u ws* **by**

```
simp

  from pre-add have ¬ invalidVertexList g f vol
    by (auto simp: pre-subdivFace′-def dest: invalidVertexList-shorten)
  then have rule5: ¬ invalidVertexList g′ f21 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 ¬ is-duplicateEdge g f a aa)
     apply (thin-tac ∀ i<|vol| − Suc 0. ¬ (case vol ! i of None ⇒ False
        | Some a ⇒ option-case False (is-duplicateEdge g f a) (vol ! (i+1))))
     apply (simp add: is-duplicateEdge-def)
     apply (subgoal-tac a ∈ set (removeNones vol) ∧ aa ∈ set (removeNones vol))
      apply (rule conjI)
       apply (rule impI)
       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 (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*: ⋀ *a b*. (*a*,*b*) ∈ *edges f* ⟹ (*a*,*b*) ∈ *edges f12* ∨ (*a*,*b*) ∈ *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*:
⋀ *a b*. (*a*,*b*) ∈ *edges g′* ⟹
(*a*,*b*) ∈ *edges g* ∨ (*a*,*b*) ∈ *edges f12* ∨ (*a*,*b*) ∈ *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* ∈ 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* < |*vol*| ⟹ ¬ *invalidVertexList g f* (*Some u* # *vol*) ⟹
(∃ *y*. *hd vol* = *Some y*) ⟶ ¬ *is-duplicateEdge g′ f21 u* (*the* (*hd 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* ∉ 𝒱 *f12*) **defer** **apply** (*rule removeNones-vol-not-f12*) **apply** *simp*
apply (*simp add*: *is-duplicateEdge-def*)

```
  apply (subgoal-tac y ∈ set (removeNones vol))
   apply (rule conjI)
    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)
   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** ¬ *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*: ⋀ *ys*. *filter P xs* = *ys* ⟹ *distinct xs* ⟹ *before ys u v* ⟹ *before xs u v*
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* ¬ *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* $u \in$ *set ys* ∧ $v \in$ *set ys* ∧ $u \neq v$) **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 ∈ set (filter P xs)) apply force
  apply (simp add: before-def) apply (elim exE) apply simp
  apply (subgoal-tac u ∈ set (filter P 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) ⟹ 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 ∧ u ≠ last (verticesFrom f v') ∨ v' ≠ v)
apply auto
    apply(rule verticesFrom-in'[where v = v'])
     apply(clarsimp simp:before-def)
    apply simp
   apply (rule ovl-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*)
$\Longrightarrow f \in \mathcal{F}\ g \Longrightarrow (f \cdot v = u \longrightarrow n \neq 0) \Longrightarrow \mathcal{V}\ f \subseteq \mathcal{V}\ g$
$\Longrightarrow$ *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*: $f \in \mathcal{F}\ g$
and *nextVert*: $(f \cdot v = u \longrightarrow n \neq 0)$ **and** *subset*: $\mathcal{V}\ f \subseteq \mathcal{V}\ g$
have *distinct* [*countVertices g* ..< *countVertices g* + *n*] **by** (*induct n*) *auto*
moreover
have $\mathcal{V}\ g \cap$ *set* [*countVertices g* ..< *countVertices g* + *n*] = {}
apply (*cases g*) **by** *auto*
with *subset* **have** $\mathcal{V}\ f \cap$ *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*: $v \in \mathcal{V}\ f \wedge u \in \mathcal{V}\ f \wedge v \neq u$
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* $\Longrightarrow 0 < n$ **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* $\Longrightarrow$ *distinct* (*verticesFrom f v′*) $\Longrightarrow v \neq v′ \Longrightarrow \neg$ *is-nextElem* (*verticesFrom f v′*) *u v*
apply (*simp add*: *before-def is-nextElem-def verticesFrom-hd is-sublist-def*) **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
have $\bigwedge m.\ \{k.\ k < m\} \cap \{k.\ m \leq k \wedge k < (m + n)\} = \{\}$ **by** *auto*
moreover

from *pre-add f help2 help1 help* **have** [*countVertices g*..<*countVertices g* + *n*] = [] $\Longrightarrow (v, u) \notin$ *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: split-if-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' ∈ 𝒱 f)
      apply (thin-tac u ∈ 𝒱 f)

      apply (simp add: verticesFrom-is-nextElem)
      apply (rule ccontr) apply simp
      apply (subgoal-tac v' ∈ 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+

    apply (rule conjI) apply (induct g) apply simp
    apply (rule ccontr) apply (elim conjE)
    by (simp add: invalidVertexList-def is-duplicateEdge-def)
qed


lemma pre-subdivFace'-Some1:
  pre-subdivFace' g f v' v n ((Some u) # vol)
  ⟹ f ∈ ℱ g ⟹ (f • v = u ⟶ n ≠ 0) ⟹ 𝒱 f ⊆ 𝒱 g
  ⟹ f21 = fst (snd (splitFace g v u f [countVertices g ..< countVertices g + n]))
  ⟹ g' =  snd (snd (splitFace g v u f [countVertices g ..< countVertices g + n]))
  ⟹ pre-subdivFace' g' f21 v' u 0 vol
  apply (subgoal-tac pre-splitFace g v u f [countVertices g ..< countVertices g + n])
   apply (rule pre-subdivFace'-Some1') apply assumption+
  apply simp
  apply (rule pre-subdivFace'-preFaceDiv)
  by auto
```

end

13 Invariants of (Plane) Graphs

theory *Invariants*
imports *FaceDivisionProps*
begin

13.1 Rotation of face into normal form

constdefs *minVertex* :: *face* ⇒ *vertex*
minVertex f ≡ *minList* (*vertices f*)

constdefs *normFace* :: *face* ⇒ *vertex list*
normFace ≡ λ*f*. *verticesFrom f* (*minVertex f*)

constdefs *normFaces* :: *face list* ⇒ *vertex list list*
normFaces fl ≡ *map normFace fl*

lemma *normFaces-distinct*: *distinct* (*normFaces fl*) ⟹ *distinct fl*
apply (*induct fl*) **by** (*auto simp*: *normFace-def normFaces-def*)

13.2 Minimal (plane) graph properties

constdefs *minGraphProps′* :: *graph* ⇒ *bool*
minGraphProps′ g ≡ ∀ *f* ∈ $\mathcal{F}$ *g*. *2* < |*vertices f*| ∧ *distinct* (*vertices f*)

constdefs *edges-sym*:: *graph* ⇒ *bool*
edges-sym g ≡ ∀ *a b*. (*a*,*b*) ∈ *edges g* ⟶ (*b*,*a*) ∈ *edges g*

constdefs *faceListAt-len*:: *graph* ⇒ *bool*
faceListAt-len g ≡ (*length* (*faceListAt g*) = *countVertices g*)

constdefs *facesAt-eq*:: *graph* ⇒ *bool*
facesAt-eq g ≡ ∀ *v* ∈ $\mathcal{V}$ *g*. *set*(*facesAt g v*) = {*f*. *f* ∈ $\mathcal{F}$ *g* ∧ *v* ∈ $\mathcal{V}$ *f*}

constdefs *facesAt-distinct*:: *graph* ⇒ *bool*
facesAt-distinct g ≡ ∀ *v* ∈ $\mathcal{V}$ *g*. *distinct* (*normFaces* (*facesAt g v*))

constdefs *faces-distinct*:: *graph* ⇒ *bool*
faces-distinct g ≡ *distinct* (*normFaces* (*faces g*))

constdefs *faces-subset*:: *graph* ⇒ *bool*
faces-subset g ≡ ∀ *f* ∈ $\mathcal{F}$ *g*. $\mathcal{V}$ *f* ⊆ $\mathcal{V}$ *g*

constdefs *edges-disj* :: *graph* ⇒ *bool*

edges-disj g $\equiv$
$\forall f \in \mathcal{F}\ g.\ \forall f' \in \mathcal{F}\ g.\ f \neq f' \longrightarrow \mathcal{E}\ f \cap \mathcal{E}\ f' = \{\}$

constdefs
face-face-op:: *graph* $\Rightarrow$ *bool*
face-face-op g $\equiv$ $|$*faces g*$|$ $\neq$ *2* $\longrightarrow$
$(\forall f{\in}\mathcal{F}\ g.\ \forall f'{\in}\mathcal{F}\ g.\ f \neq f' \longrightarrow \mathcal{E}\ f \neq (\mathcal{E}\ f')^{-1})$

constdefs
one-final-but :: *graph* $\Rightarrow$ (*vertex* $\times$ *vertex*)*set* $\Rightarrow$ *bool*
one-final-but g E $\equiv$
$\forall f \in \mathcal{F}\ g.\ \neg$ *final f* $\longrightarrow$
$(\forall (a,b){\in}\mathcal{E}\ f - E.\ (b,a) : E \vee (\exists f'{\in}\mathcal{F}\ g.\ \mathit{final}\ f' \wedge (b,a) \in \mathcal{E}\ f'))$

one-final :: *graph* $\Rightarrow$ *bool*
one-final g $\equiv$ *one-final-but g* $\{\}$

constdefs
minGraphProps :: *graph* $\Rightarrow$ *bool*
minGraphProps g $\equiv$ *minGraphProps′ g* $\wedge$ *facesAt-eq g* $\wedge$ *faceListAt-len g* $\wedge$ *facesAt-distinct g* $\wedge$ *faces-distinct g* $\wedge$ *faces-subset g* $\wedge$ *edges-sym g* $\wedge$ *edges-disj g* $\wedge$ *face-face-op g*

inv :: *graph* $\Rightarrow$ *bool*
inv g $\equiv$ *minGraphProps g* $\wedge$ *one-final g* $\wedge$ $|$*faces g*$|$ $\geq$ *2*

lemma *facesAt-distinctI*:
$(\bigwedge v.\ v \in \mathcal{V}\ g \Longrightarrow$ *distinct* (*normFaces* (*facesAt g v*))) $\Longrightarrow$ *facesAt-distinct g*
by (*simp add*: *facesAt-distinct-def*)

lemma *minGraphProps2*: *minGraphProps g* $\Longrightarrow$
$f \in \mathcal{F}\ g \Longrightarrow 2 < |\mathit{vertices}\ f|$
by (*unfold minGraphProps-def minGraphProps′-def*) *auto*

lemma *mgp-vertices3*:
minGraphProps g $\Longrightarrow f \in \mathcal{F}\ g \Longrightarrow |\mathit{vertices}\ f| \geq 3$
by(*auto dest*:*minGraphProps2*)

lemma *mgp-vertices-nonempty*:
minGraphProps g $\Longrightarrow f \in \mathcal{F}\ g \Longrightarrow \mathit{vertices}\ f \neq []$
by(*auto dest*:*minGraphProps2*)

lemma *minGraphProps3*: *minGraphProps g* $\Longrightarrow$
$f \in \mathcal{F}\ g \Longrightarrow$ *distinct* (*vertices f*)
by (*unfold minGraphProps-def minGraphProps′-def*) *auto*

lemma *minGraphProps4*: *minGraphProps g* $\Longrightarrow$
(*length* (*faceListAt g*) = *countVertices g*)
by (*unfold minGraphProps-def faceListAt-len-def*) *simp*

lemma *minGraphProps5*:
$[\![$*minGraphProps g*; $f \in$ *set* (*facesAt g v*)$]\!]$ $\Longrightarrow f \in \mathcal{F}\ g$
by(*auto simp*: *facesAt-def facesAt-eq-def minGraphProps-def minGraphProps′-def faceListAt-len-def split*:*split-if-asm*)

lemma *minGraphProps6*:
minGraphProps g $\Longrightarrow f \in$ *set* (*facesAt g v*) $\Longrightarrow v \in \mathcal{V}\ f$
by(*auto simp*: *facesAt-def facesAt-eq-def minGraphProps-def minGraphProps′-def faceListAt-len-def split*:*split-if-asm*)

lemma *minGraphProps9*: *minGraphProps g* $\Longrightarrow$
$f \in \mathcal{F}\ g \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow v \in \mathcal{V}\ g$
by (*unfold minGraphProps-def faces-subset-def*) *auto*

lemma *minGraphProps7*: *minGraphProps g* $\Longrightarrow$
$f \in \mathcal{F}\ g \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow f \in$ *set* (*facesAt g v*)
apply(*frule* (*2*) *minGraphProps9*)
by (*unfold minGraphProps-def facesAt-eq-def*) *simp*

lemma *minGraphProps-facesAt-eq*: *minGraphProps g* $\Longrightarrow$
$v \in \mathcal{V}\ g \Longrightarrow$ *set* (*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*]: *minGraphProps g* $\Longrightarrow$ *distinct* (*facesAt g v*)
by(*auto simp*: *facesAt-def minGraphProps-def minGraphProps′-def facesAt-distinct-def dest*:*normFaces-distinct*)

lemma *minGraphProps8*: *minGraphProps g* $\Longrightarrow$ *distinct* (*normFaces* (*facesAt g v*))
by(*auto simp*: *facesAt-def minGraphProps-def minGraphProps′-def facesAt-distinct-def normFaces-def*)

lemma *minGraphProps8a*: *minGraphProps g* $\Longrightarrow$
$v \in \mathcal{V}\ g \Longrightarrow$ *distinct* (*normFaces* (*faceListAt g* ! *v*))
apply (*frule minGraphProps8*[**where** *v*=*v*]) **by** (*simp add*: *facesAt-def*)

lemma *minGraphProps8a′*: *minGraphProps g* $\Longrightarrow$
$v <$ *countVertices g* $\Longrightarrow$ *distinct* (*normFaces* (*faceListAt g* ! *v*))
by (*simp add*: *minGraphProps8a vertices-graph*)

lemma *minGraphProps9′*: *minGraphProps g* $\Longrightarrow$
$f \in \mathcal{F}\ g \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow v <$ *countVertices g*

by (*simp add*: *minGraphProps9 in-vertices-graph*[*symmetric*])

lemma *minGraphProps10*:
minGraphProps g ⟹ (*a*, *b*) ∈ *edges g* ⟹ (*b*, *a*) ∈ *edges g*
apply (*unfold minGraphProps-def edges-sym-def*)
apply (*elim conjE allE impE*)
by *simp*+

lemma *minGraphProps11*: *minGraphProps g* ⟹
distinct (*normFaces* (*faces g*))
by (*unfold minGraphProps-def faces-distinct-def*) *simp*

lemma *minGraphProps11′*: *minGraphProps g* ⟹
distinct (*faces g*)
by(*simp add*: *minGraphProps11 normFaces-distinct*)

lemma *face-eq-if-normFace-eq*:
⟦ *minGraphProps g*; *f* ∈ $\mathcal{F}$ *g*; *f′* ∈ $\mathcal{F}$ *g*; *normFace f* = *normFace f′* ⟧
⟹ *f* = *f′*
apply(*drule minGraphProps11*)
apply(*simp add*:*normFaces-def distinct-map*)
apply(*blast dest*:*inj-onD*)
done

lemma *minGraphProps12*:
minGraphProps g ⟹ *f* ∈ $\mathcal{F}$ *g* ⟹ (*a*,*b*) ∈ $\mathcal{E}$ *f* ⟹ (*b*,*a*) ∉ $\mathcal{E}$ *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*) ∧ *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* $\wedge$ *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* $\Longrightarrow$
f $\in \mathcal{F}$ *g* $\Longrightarrow$ *v* $\in \mathcal{V}$ *f* $\Longrightarrow$ *f* $\in$ *set* (*faceListAt g* ! *v*)
apply (*frule minGraphProps7*) **apply** *assumption+*
by (*simp add*: *facesAt-def split*: *split-if-asm*)

lemma *mgp-edges-disj*:
$\llbracket$ *minGraphProps g*; *f* $\neq$ *f'*; *f* $\in \mathcal{F}$ *g*; *f'* $\in \mathcal{F}$ *g* $\rrbracket \Longrightarrow$
uv $\in \mathcal{E}$ *f* $\Longrightarrow$ *uv* $\notin \mathcal{E}$ *f'*
by (*simp add*:*minGraphProps-def edges-disj-def*) *blast*

lemma *one-final-but-antimono*:
one-final-but g E $\Longrightarrow$ *E* $\subseteq$ *E'* $\Longrightarrow$ *one-final-but g E'*
apply(*unfold one-final-but-def*)
apply *blast*
done

lemma *one-final-antimono*: *one-final g* $\Longrightarrow$ *one-final-but g E*
apply(*unfold one-final-def one-final-but-def*)
apply *blast*
done

lemma *inv-two-faces*: *inv g* $\Longrightarrow$ |*faces g*| $\geq$ *2*
by(*simp add*:*inv-def*)

lemma *inv-mgp*[*simp*]: *inv g* $\Longrightarrow$ *minGraphProps g*
by(*simp add*:*inv-def*)

lemma *makeFaceFinal-id*[*simp*]: *final f* $\Longrightarrow$ *makeFaceFinal f g* $=$ *g*
apply(*induct g*)
apply (*simp add*:*makeFaceFinal-def makeFaceFinalFaceList-def*
setFinal-eq-iff [*THEN iffD2*])
done

lemma *inv-one-finalD′*:
$[\![\ \mathit{inv}\ g;\ f \in \mathcal{F}\ g;\ \neg\ \mathit{final}\ f;\ (a,b) \in \mathcal{E}\ f\ ]\!] \Longrightarrow$
$\exists f' \in \mathcal{F}\ g.\ \mathit{final}\ f' \wedge f' \neq f \wedge (b,a) \in \mathcal{E}\ f'$
apply(*unfold inv-def one-final-def one-final-but-def*)
apply *blast*
done

lemmas *minGraphProps* =
minGraphProps2 minGraphProps3 minGraphProps4
minGraphProps5 minGraphProps6 minGraphProps7 minGraphProps8
minGraphProps9

lemmas *minGraphProps-simps* = *minGraphProps4*

lemma *mgp-no-loop*[*simp*]:
$\mathit{minGraphProps}\ g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow f \cdot v \neq v$
apply(*frule* (*1*) *mgp-vertices3*)
apply(*frule* (*1*) *minGraphProps3*)
apply(*simp add*: *distinct-no-loop1*)
done

lemma *mgp-facesAt-no-loop*:
$\mathit{minGraphProps}\ g \Longrightarrow f \in \mathit{set}\ (\mathit{facesAt}\ g\ v) \Longrightarrow f \cdot v \neq v$
by(*blast dest*:*mgp-no-loop minGraphProps5 minGraphProps6*)

lemma *edge-pres-faceAt*:
$[\![\ \mathit{minGraphProps}\ g;\ f \in \mathit{set}(\mathit{facesAt}\ g\ u);\ (u,v) \in \mathcal{E}\ f\ ]\!] \Longrightarrow$
$f \in \mathit{set}(\mathit{facesAt}\ g\ v)$
apply(*auto simp*:*edges-face-eq*)
apply(*rule minGraphProps7*, *assumption*)
apply(*blast intro*:*minGraphProps*)
apply(*simp*)
done

lemma *in-facesAt-nextVertex*:
$\mathit{minGraphProps}\ g \Longrightarrow f \in \mathit{set}(\mathit{facesAt}\ g\ v) \Longrightarrow f \in \mathit{set}(\mathit{facesAt}\ g\ (f \cdot v))$
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*

```
and fv: f ∈ set(facesAt g v) and uv: (u,v) ∈ ℰ f
shows ∃f' ∈ set(facesAt g v). (v,u) ∈ ℰ f'
proof −
  from fv have f ∈ ℱ g by(blast intro:minGraphProps)
  with uv have (u,v) ∈ ℰ g by(auto simp:edges-graph-def)
  hence (v,u) ∈ ℰ g by(blast intro:minGraphProps10)
  then obtain f' where f': f' ∈ ℱ g and vu: (v,u) ∈ ℰ f'
    by(auto simp:edges-graph-def)
  from vu have v ∈ 𝒱 f' by(auto simp:edges-face-eq)
  with f' have f' ∈ set(facesAt g v) by(blast intro:minGraphProps)
  with vu show ?thesis by blast
qed


lemma mgp-nextVertex-face-ex2:
assumes mgp[intro]: minGraphProps g and f: f ∈ set(facesAt g v)
shows ∃f' ∈ set(facesAt g (f · v)). f' · (f · v) = v
proof −
  from f have (v,f · v) ∈ ℰ f
    by(blast intro: nextVertex-in-edges minGraphProps)
  with in-facesAt-nextVertex[OF mgp f]
  obtain f' where f' ∈ set(facesAt g (f · v))
    and (f · v,v) ∈ ℰ f'
    by(blast dest: mgp-edge-face-ex[OF mgp])
  thus ?thesis by (auto simp: edges-face-eq)
qed


lemma inv-finals-nonempty: inv g ⟹ finals g ≠ []
apply(frule inv-two-faces)
apply(clarsimp simp:filter-empty-conv finals-def)
apply(subgoal-tac faces g ≠ [])
 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 fastsimp
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*

```
constdefs containsUnacceptableEdgeSnd' :: (nat ⇒ nat ⇒ bool) ⇒ nat list ⇒ bool
 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: ⋀ v. containsUnacceptableEdgeSnd N v is = containsUnacceptableEdgeSnd' N (v#is)
proof (induct is)
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: split-if-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*

consts *replacefacesAt2* :: *nat list* ⇒ *face* ⇒ *face list* ⇒ *face list list* ⇒ *face list list*
primrec *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*]:
$\bigwedge F$. *replacefacesAt ns oldf newfs F* = *replacefacesAt2 ns oldf newfs F*
apply (*induct ns*)
by (*auto simp add*: *replacefacesAt-def*)

lemma *replacefacesAt2-notin*:
$\bigwedge Fss$. $i \notin$ *set is* $\Longrightarrow$ (*replacefacesAt2 is olfF newFs Fss*)!*i* = *Fss*!*i*
proof (*induct is*)
case *Nil* **then show** *?case* **by** (*simp*)
next
case (*Cons j js*)
then show *?case*
apply (*case-tac* $j < |Fss|$) **by** (*auto*)
qed

lemma *replacefacesAt2-in*:
$\bigwedge Fss\ i$. $i \in$ *set is* $\Longrightarrow$ *distinct is* $\Longrightarrow$ $i < |Fss|$ $\Longrightarrow$
(*replacefacesAt2 is olfF newFs Fss*)!*i* = *replace olfF newFs* (*Fss* !*i*)
proof (*induct is*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons j js*)
then have $j = i \wedge i \notin$ *set js* $\vee$ $i \neq j \wedge i \in$ *set js* **by** *auto*

then show *?case*
proof (*elim disjE conjE*)
assume $j = i$ $i \notin$ *set js* **with** *Cons* **show** *?thesis*
by (*auto simp add*: *replacefacesAt2-notin*)
next
assume $i \in$ *set js* $i \neq j$ **with** *Cons* **show** *?thesis* **by** *simp*
qed
qed

lemma *distinct-replacefacesAt21*:
$\bigwedge i.\ i < |\mathit{Fss}| \Longrightarrow i \in \mathit{set\ is} \Longrightarrow \mathit{distinct\ is} \Longrightarrow \mathit{distinct}\ (\mathit{Fss}!i) \Longrightarrow \mathit{distinct}$ $\mathit{newFs} \Longrightarrow$
$\mathit{set}\ (\mathit{Fss}\ !\ i) \cap \mathit{set\ newFs} \subseteq \{\mathit{olfF}\} \Longrightarrow$
distinct ((*replacefacesAt2 is olfF newFs Fss*)! *i*)
proof (*induct is*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons j js*)
then have $j = i \wedge i \notin$ *set js* $\vee$ $i \neq j \wedge i \in$ *set js* **by** *auto*
then show *?case*
proof (*elim disjE conjE*)
assume $j = i$ $i \notin$ *set js* **with** *Cons* **show** *?thesis*
by (*simp add*: *replacefacesAt2-notin distinct-replace*)
next
assume $i \in$ *set js* $i \neq j$ **with** *Cons* **show** *?thesis*
by (*simp add*: *replacefacesAt2-in distinct-replace*)
qed
qed

lemma *distinct-replacefacesAt22*:
$\bigwedge i.\ i < |\mathit{Fss}| \Longrightarrow i \notin \mathit{set\ is} \Longrightarrow \mathit{distinct\ is} \Longrightarrow \mathit{distinct}\ (\mathit{Fss}!i) \Longrightarrow \mathit{distinct}$ $\mathit{newFs} \Longrightarrow$
$\mathit{set}\ (\mathit{Fss}\ !\ i) \cap \mathit{set\ newFs} \subseteq \{\mathit{olfF}\} \Longrightarrow$
distinct ((*replacefacesAt2 is 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*:
$\bigwedge i.\ i < |\mathit{Fss}| \Longrightarrow \mathit{distinct\ is} \Longrightarrow \mathit{distinct}\ (\mathit{Fss}!i) \Longrightarrow \mathit{distinct\ newFs} \Longrightarrow$
$\mathit{set}\ (\mathit{Fss}\ !\ i) \cap \mathit{set\ newFs} \subseteq \{\mathit{olfF}\} \Longrightarrow$
distinct ((*replacefacesAt2 is olfF newFs Fss*)! *i*)
apply (*cases* $i \in$ *set is*)

by (*auto intro*: *distinct-replacefacesAt21 distinct-replacefacesAt22*)

lemma *replacefacesAt2-length*: $\bigwedge$ *vs*. $|$*replacefacesAt2 nvs f′* $[$*f″*$]$ *vs*$|$ = $|$*vs*$|$
by (*induct nvs*) *simp-all*

lemma *replacefacesAt2-nth1*: !!*F*. $k \notin$ *set ns* $\Longrightarrow$
(*replacefacesAt2 ns oldf newfs F*) ! *k* = *F* ! *k*
apply (*induct ns*) **by** *auto*

lemma *replacefacesAt2-nth1′*: !!*F*. $k \in$ *set ns* $\Longrightarrow k < |F| \Longrightarrow$ *distinct ns* $\Longrightarrow$
(*replacefacesAt2 ns oldf newfs F*) ! *k* = (*replace oldf newfs* (*F*!*k*))
apply (*induct ns*) **apply** *auto* **apply** (*simp add*: *replacefacesAt2-nth1*)+
apply (*case-tac* $a = k$) **by** *auto*

lemma *replacefacesAt2-nth2*: $k < |F| \Longrightarrow$
(*replacefacesAt2* $[k]$ *oldf newfs F*) ! *k* = *replace oldf newfs* (*F*!*k*)
by (*auto*)

lemma *replacefacesAt2-length*[*simp*]: $\bigwedge$ *vs*. $|$*replacefacesAt2 nvs f′ f″ vs*$|$ = $|$*vs*$|$
by (*induct nvs*) *simp-all*

lemma *replacefacesAt2-replacefacesAt2-nth1*:
!!*F*. $k < |\ F\ | \Longrightarrow k \notin$ *set ns* $\Longrightarrow$ *distinct ms* $\Longrightarrow$
replacefacesAt2 ms oldf′ newfs′ (*replacefacesAt2 ns oldf newfs F*) ! *k* =
replacefacesAt2 ms oldf′ newfs′ F ! *k*
proof (*induct ms*)
case *Nil* **then show** *?case* **by** (*simp add*: *replacefacesAt2-nth1*)
next
case (*Cons m ms*) **then show** *?case*
apply (*case-tac* $m = k$) **apply** (*simp*) **apply** *safe*
apply (*simp add*: *replacefacesAt2-nth1*)
apply (*simp*) **apply** (*rule impI*) **apply** *safe*
apply (*case-tac* $k \in$ *set ms*) **apply** (*simp add*: *replacefacesAt2-nth1′*)
by (*simp add*: *replacefacesAt2-nth1*)
qed

lemma *replacefacesAt2-nth*: !!*F*. $k \in$ *set ns* $\Longrightarrow k < |F| \Longrightarrow$ *oldf* $\notin$ *set newfs*
$\Longrightarrow$
distinct (*F*!*k*) $\Longrightarrow$ *distinct newfs* $\Longrightarrow$ *oldf* $\in$ *set* (*F*!*k*) $\longrightarrow$ *set newfs* $\cap$ *set* (*F*!*k*)
$\subseteq$ {*oldf*} $\Longrightarrow$
(*replacefacesAt2 ns oldf newfs F*) ! *k* = (*replace oldf newfs* (*F*!*k*))
proof (*induct ns*)
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* $\in$ *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*:
$\bigwedge$*Fss*. *i* $\notin$ *set is* $\Longrightarrow$ (*replacefacesAt is olfF newFs Fss*)!*i* = *Fss*!*i*
by (*simp add*: *replacefacesAt-eq replacefacesAt2-notin*)

lemma *replacefacesAt-in*:
$\bigwedge$*Fss i*. *i* $\in$ *set is* $\Longrightarrow$ *distinct is* $\Longrightarrow$ *i* < |*Fss*| $\Longrightarrow$
(*replacefacesAt is olfF newFs Fss*)!*i* = *replace olfF newFs* (*Fss* !*i*)
by (*simp add*: *replacefacesAt-eq replacefacesAt2-in*)

lemma *distinct-replacefacesAt1*:
$\bigwedge$*i*. *i* < |*Fss*| $\Longrightarrow$ *i* $\in$ *set is* $\Longrightarrow$ *distinct is* $\Longrightarrow$ *distinct* (*Fss*!*i*) $\Longrightarrow$ *distinct newFs* $\Longrightarrow$
set (*Fss* ! *i*) $\cap$ *set newFs* $\subseteq$ {*olfF*} $\Longrightarrow$
distinct ((*replacefacesAt is olfF newFs Fss*)! *i*)
by (*simp add*: *replacefacesAt-eq distinct-replacefacesAt21*)

lemma *distinct-replacefacesAt2*:
$\bigwedge$*i*. *i* < |*Fss*| $\Longrightarrow$ *i* $\notin$ *set is* $\Longrightarrow$ *distinct is* $\Longrightarrow$ *distinct* (*Fss*!*i*) $\Longrightarrow$ *distinct newFs* $\Longrightarrow$
set (*Fss* ! *i*) $\cap$ *set newFs* $\subseteq$ {*olfF*} $\Longrightarrow$
distinct ((*replacefacesAt is olfF newFs Fss*)! *i*)
by (*simp add*: *replacefacesAt-eq distinct-replacefacesAt22*)

lemma *distinct-replacefacesAt*:
$\bigwedge$*i*. *i* < |*Fss*| $\Longrightarrow$ *distinct is* $\Longrightarrow$ *distinct* (*Fss*!*i*) $\Longrightarrow$ *distinct newFs* $\Longrightarrow$
set (*Fss* ! *i*) $\cap$ *set newFs* $\subseteq$ {*olfF*} $\Longrightarrow$
distinct ((*replacefacesAt is olfF newFs Fss*)! *i*)
by (*simp add*: *replacefacesAt-eq distinct-replacefacesAt2-2*)

lemma *replacefacesAt-length*[*simp*]: |*replacefacesAt nvs f′* [*f″*] *vs*| = |*vs*|
by (*simp add*: *replacefacesAt-eq*)

lemma *replacefacesAt-nth1*: *k* $\notin$ *set ns* $\Longrightarrow$
(*replacefacesAt ns oldf newfs F*) ! *k* = *F* ! *k*
by (*simp add*: *replacefacesAt-eq replacefacesAt2-nth1*)

lemma *replacefacesAt-nth1′*: *k* $\in$ *set ns* $\Longrightarrow$ *k* < |*F*| $\Longrightarrow$ *distinct ns* $\Longrightarrow$
(*replacefacesAt ns oldf newfs F*) ! *k* = (*replace oldf newfs* (*F*!*k*))
by (*simp add*: *replacefacesAt-eq replacefacesAt2-nth1′*)

lemma *replacefacesAt-nth2*: $k < |F| \Longrightarrow$
 (*replacefacesAt* [*k*] *oldf newfs F*) ! k = *replace oldf newfs* (*F*!*k*)
by (*simp add*: *replacefacesAt-eq replacefacesAt2-nth2*)

lemma *replacefacesAt-replacefacesAt-nth1*:
 $k < |\ F\ | \Longrightarrow k \notin$ *set ns* $\Longrightarrow$ *distinct ms* $\Longrightarrow$
 replacefacesAt ms oldf′ newfs′ (*replacefacesAt ns oldf newfs F*) ! k =
 replacefacesAt ms oldf′ newfs′ F ! k
by (*simp add*: *replacefacesAt-eq replacefacesAt2-replacefacesAt2-nth1*)

lemma *replacefacesAt-nth*: !!*F*. $k \in$ *set ns* $\Longrightarrow k < |F| \Longrightarrow$ *oldf* $\notin$ *set newfs* $\Longrightarrow$
 distinct (*F*!*k*) $\Longrightarrow$ *distinct newfs* $\Longrightarrow$ *oldf* $\in$ *set* (*F*!*k*) $\longrightarrow$ *set newfs* $\cap$ *set* (*F*!*k*) $\subseteq$ {*oldf*} $\Longrightarrow$
 (*replacefacesAt ns oldf newfs F*) ! k = (*replace oldf newfs* (*F*!*k*))
by (*simp add*: *replacefacesAt-eq replacefacesAt2-nth*)

lemma *replacefacesAt2-5*: $\bigwedge$ *F*. $x \in$ *set* (*replacefacesAt2 ns oldf newfs F* ! k) $\Longrightarrow$ $x \in$ *set* (*F*!*k*) $\vee$ $x \in$ *set newfs*
proof (*induct ns*)
 case *Nil* **then show** *?case* **by** *simp*
next
 case (*Cons n ns*)
 then show *?case* **apply**(*simp add*: *split*: *split-if-asm*) **apply** (*frule Cons*)
  apply (*thin-tac* $\bigwedge$*F*. $x \in$ *set* (*replacefacesAt2 ns oldf newfs F* ! k) $\Longrightarrow$ $x \in$ *set* (*F* ! k) $\vee$ $x \in$ *set newfs*)
  apply (*case-tac* $x \in$ *set newfs*) **apply** *simp* **apply** *simp*
  apply (*case-tac* $k = n$) **apply** *simp* **apply** (*frule replace5*) **apply** *simp* **by** *simp*
qed

lemma *replacefacesAt5*: $\bigwedge$ *F*. $x \in$ *set* (*replacefacesAt ns oldf newfs F* ! k) $\Longrightarrow$ $x \in$ *set* (*F*!*k*) $\vee$ $x \in$ *set newfs*
 by (*simp add*: *replacefacesAt-eq replacefacesAt2-5*)

lemma *replacefacesAt-delete-oldF*: *oldF* $\notin$ *set newfs* $\Longrightarrow$ *distinct* (*F*!*k*) $\Longrightarrow$ $k \in$ *set ns* $\Longrightarrow$ *distinct newfs* $\Longrightarrow$
 oldF $\in$ *set* (*F* ! k) $\longrightarrow$ *set newfs* $\cap$ *set* (*F* ! k) $\subseteq$ {*oldF*} $\Longrightarrow k < |F| \Longrightarrow$
 oldF $\notin$ *set* (*replacefacesAt ns oldF newfs F* ! k)
 apply (*frule replacefacesAt-nth*) **apply** (*simp-all*)
 by (*simp-all add*:*distinct-set-replace*)

lemma *replacefacesAt-Nil*[*simp*]: *replacefacesAt* [] *f fs F* = *F*
 by (*simp add*: *replacefacesAt-eq*)

lemma *replacefacesAt-Cons*[*simp*]:
 replacefacesAt (*n* # *ns*) *f fs F* =
 (*if* $n < |F|$ *then replacefacesAt ns f fs* (*F*[*n* := *replace f fs* (*F*!*n*)])

else replacefacesAt ns f fs F)
by (*simp add*: *replacefacesAt-eq*)

lemmas *replacefacesAt-simps* = *replacefacesAt-Nil replacefacesAt-Cons*

lemma *len-nth-repAt*[*simp*]:
$!!xs.\ i < |xs| \Longrightarrow |replacefacesAt\ is\ x\ [y]\ xs\ !\ i| = |xs!i|$
by (*induct is*) (*simp-all add*: *add*:*nth-list-update*)

13.5 *normFace*

lemma *minVertex-in*: $vertices\ f \neq [] \Longrightarrow minVertex\ f \in \mathcal{V}\ f$
by (*simp add*: *minVertex-def*)

lemma *minVertex-eq-if-vertices-eq*:
$\mathcal{V}\ f = \mathcal{V}\ f' \Longrightarrow minVertex\ f = minVertex\ f'$
apply(*induct f*)
apply(*induct 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 minList-conv-Min*
    insert-absorb del:*Min-insert*)
done

lemma *normFace-eq-if-edges-eq*:
 $[\![\ distinct(vertices\ f);\ distinct(vertices\ f');\ \mathcal{E}\ f = \mathcal{E}\ f'\ ]\!]$
 $\Longrightarrow normFace\ f = normFace\ f'$
apply(*subgoal-tac* $vertices\ f \cong vertices\ f'$)
 prefer *2*
 apply(*rule cong-if-is-nextElem-eq*)
   apply *assumption*+
  apply(*simp add*:*vertices-conv-Union-edges*)
 apply(*simp add*:*expand-set-eq*)
apply(*subgoal-tac* $\mathcal{V}\ f = \mathcal{V}\ f'$)
 prefer *2* **apply**(*simp add*:*vertices-conv-Union-edges*)
apply(*frule minVertex-eq-if-vertices-eq*)
apply(*simp add*:*edges-conv-Edges normFace-def*)
apply(*cases vertices f* = [])
 apply (*simp add*:*minVertex-def verticesFrom-def split*:*split-if-asm*)
apply(*cases vertices f'* = [])
 apply (*simp add*:*minVertex-def verticesFrom-def split*:*split-if-asm*)
apply(*simp*)

apply(*subgoal-tac minVertex f′ ∈ 𝒱 f′*)
 prefer *2* **apply**(*erule minVertex-in*)
apply(*subgoal-tac minVertex f ∈ 𝒱 f*)
 prefer *2* **apply**(*erule minVertex-in*)
apply(*erule* (*2*) *verticesFrom-eq-if-vertices-cong*)
 apply *simp*
done

lemma *normFace-replace-in*: *normFace a* ∈ *set* (*normFaces* (*replace oldF newFs fs*)) ⟹
  normFace a ∈ *set* (*normFaces newFs*) ∨ *normFace a* ∈ *set* (*normFaces fs*)
 apply (*induct fs*) **apply** *simp* **apply** (*simp add*: *normFaces-def split*:*split-if-asm*)
by *auto*

lemma *distinct-replace-norm*:
distinct (*normFaces fs*) ⟹ *distinct* (*normFaces newFs*) ⟹
  set (*normFaces fs*) ∩ *set* (*normFaces newFs*) ⊆ {} ⟹ *distinct* (*normFaces* (*replace oldF newFs fs*))
 apply (*induct fs*) **apply** *simp* **apply** *simp*
 apply (*case-tac a* = *oldF*) **apply** (*simp add*: *normFaces-def*) **apply** *blast*
 apply *simp* **apply** (*simp add*: *normFaces-def*) **apply** (*rule ccontr*)
 apply (*subgoal-tac normFace a* ∈ *set*(*normFaces* (*replace oldF newFs fs*)))
 apply (*frule normFace-replace-in*) **by** (*simp add*: *normFaces-def*)+

lemma *distinct-replacefacesAt1-norm*:
 ⋀*i*. *i* < |*Fss*| ⟹ *i* ∈ *set is* ⟹ *distinct is* ⟹ *distinct* (*normFaces* (*Fss*!*i*)) ⟹ *distinct* (*normFaces newFs*) ⟹
  set (*normFaces* (*Fss* ! *i*)) ∩ *set* (*normFaces newFs*) ⊆ {} ⟹
  distinct (*normFaces* ((*replacefacesAt is oldF newFs Fss*)! *i*))
proof (*induct is*)
 case *Nil* **then show** *?case* **by** *simp*
next
 case (*Cons j js*)
 then have *j* = *i* ∧ *i* ∉ *set js* ∨ *i* ≠ *j* ∧ *i* ∈ *set js* **by** *auto*
 then show *?case*
 proof (*elim disjE conjE*)
  assume *j* = *i i* ∉ *set js* **with** *Cons* **show** *?thesis*
   by (*simp add*: *replacefacesAt-notin distinct-replace-norm*)
 next
  assume *i* ∈ *set js i* ≠ *j* **with** *Cons* **show** *?thesis*
   by (*simp add*: *replacefacesAt-in distinct-replace-norm*)
 qed
qed

lemma *distinct-replacefacesAt2-norm*:
 ⋀*i*. *i* < |*Fss*| ⟹ *i* ∉ *set is* ⟹ *distinct is* ⟹ *distinct* (*normFaces* (*Fss*!*i*)) ⟹ *distinct* (*normFaces newFs*) ⟹

set (*normFaces* (*Fss* ! *i*)) $\cap$ *set* (*normFaces newFs*) $\subseteq$ {} $\Longrightarrow$
distinct (*normFaces* ((*replacefacesAt is oldF 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*:
$\bigwedge i.\ i < |Fss| \Longrightarrow$ *distinct is* $\Longrightarrow$ *distinct* (*normFaces* (*Fss*!*i*)) $\Longrightarrow$ *distinct* (*normFaces newFs*) $\Longrightarrow$
set (*normFaces* (*Fss* ! *i*)) $\cap$ *set* (*normFaces newFs*) $\subseteq$ {} $\Longrightarrow$
distinct (*normFaces* ((*replacefacesAt is olfF newFs Fss*)! *i*))
apply (*cases* $i \in$ *set is*)
by (*auto intro*: *distinct-replacefacesAt1-norm distinct-replacefacesAt2-norm*)

lemma *normFace-in-cong*: *vertices f* $\neq$ [] $\Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *normFace f* $\in$ *set* (*normFaces* (*faces g*)) $\Longrightarrow$
$\exists\, f' \in$ *set* (*faces g*). $f \cong f'$
apply (*simp add*: *normFace-def normFaces-def*)
apply (*erule imageE*)
apply(*rename-tac f'*)
apply (*rule bexI*)
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*: $a \in \mathcal{V}\, f \Longrightarrow a \notin \mathcal{V}\, f' \Longrightarrow$ *vertices f'* $\neq$ [] $\Longrightarrow$ *normFace f* $\neq$ *normFace f'*
apply (*simp add*: *normFace-def*)
apply (*subgoal-tac* $a \in$ *set* (*verticesFrom f* (*minVertex f*)))
apply (*subgoal-tac* $a \notin$ *set* (*verticesFrom f'* (*minVertex f'*))) **apply** *force*
apply (*subgoal-tac* (*vertices f'*) $\cong$ (*verticesFrom f'* (*minVertex f'*))) **apply** (*simp add*: *congs-pres-nodes*)
apply (*rule verticesFrom-congs*) **apply** (*rule minVertex-in*) **apply** *simp*
apply (*subgoal-tac* (*vertices f*) $\cong$ (*verticesFrom f* (*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*:

```
  pre-split-face oldF ram1 ram2 vs ⟹
  2 < |vertices oldF| ⟹ 2 < |vertices f12| ⟹ 2 < |vertices f21| ⟹
  (f12, f21) = split-face oldF ram1 ram2 vs ⟹ normFace f12 ≠ normFace f21
proof –
  assume split: (f12, f21) = split-face oldF ram1 ram2 vs
  pre-split-face oldF ram1 ram2 vs
    and minlen: 2 < |vertices oldF|  2 < | vertices f12|  2 < | vertices f21|
  from split have dist-f12: distinct (vertices f12) by (rule split-face-distinct1)
  from split have dist-f21: distinct (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 (vertices oldF) ram2 ram1)
 apply (case-tac between (vertices oldF) ram1 ram2)
 apply simp apply (subgoal-tac |vertices oldF| = 2)

 apply simp apply (frule verticesFrom-ram1)
 apply (subgoal-tac distinct (vertices oldF)) apply (drule verticesFrom-length)
 apply (subgoal-tac ram1 ∈ 𝒱 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 ∈ 𝒱 (Face (rev vs @ ram1 # between (vertices 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 ∈ set fs ⟹ normFace f ∈ set (normFaces fs)
  by (simp add: normFaces-def)
```

13.6 Invariants of *splitFace*

```
lemma splitFace-holds-minGraphProps':
  pre-splitFace g' v a f' vs ⟹ minGraphProps' g' ⟹
  minGraphProps' (snd (snd (splitFace g' v a f' vs)))
apply (simp add: minGraphProps'-def)
apply safe
 apply (simp add: splitFace-def split-def)
```

```
 apply (case-tac f ∈ 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:
 pre-splitFace g' v a f' vs ⟹
 minGraphProps g' ⟹
 faceListAt-len (snd (snd (splitFace g' v a f' vs)))
by (simp add: minGraphProps-def faceListAt-len-def splitFace-def split-def)


lemma splitFace-new-f12:
 pre-splitFace g ram1 ram2 oldF newVs ⟹
  minGraphProps g ⟹
 (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs ⟹
  f12 ∉ ℱ g
proof −
  assume props: minGraphProps g
  assume pre: pre-splitFace g ram1 ram2 oldF newVs
  assume spl: (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs
  show ?thesis
  proof (cases newVs)
    case Nil with pre have (ram2, ram1) ∉ edges g
      by (unfold pre-splitFace-def) auto
    moreover from Nil pre
    have (ram2, ram1) ∈ edges f12
     apply (rule-tac splitFace-empty-ram2-ram1-in-f12)
     by (auto simp: Nil[symmetric])
    ultimately show ?thesis by (auto simp add: edges-graph-def)
  next
    case (Cons v vs)
    with pre have v ∉ 𝒱 g
      apply (unfold pre-splitFace-def) by auto
    moreover from Cons spl have v ∈ 𝒱 f12
      by (simp add: splitFace-f12-new-vertices)
```

```
    moreover note props
    ultimately show ?thesis by (auto dest: minGraphProps)
  qed
qed

lemma splitFace-new-f12-norm:
 pre-splitFace g ram1 ram2 oldF newVs ⟹
  minGraphProps g ⟹
 (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs ⟹
  normFace f12 ∉ set (normFaces (faces g))
proof −
  assume props: minGraphProps g
  assume pre: pre-splitFace g ram1 ram2 oldF newVs
  assume spl: (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs
  show ?thesis
  proof (cases newVs)
    case Nil with pre have (ram2, ram1) ∉ edges g
      by (unfold pre-splitFace-def) auto
    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) ∈ edges f12
      apply (rule-tac splitFace-empty-ram2-ram1-in-f12)
      by (auto simp: Nil[symmetric])
    moreover
    with dist-f12 have vertices f12 ≠ []
     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 assumption+ apply (elim bexE)
      apply (subgoal-tac (ram2, ram1) ∈ edges f') apply simp
     apply (subgoal-tac (vertices f12) ≅ (vertices f')) apply (frule congs-distinct)
      by (simp add: cong-face-def is-nextElem-congs-eq)+
  next
    case (Cons v vs)
    with pre have v ∉ 𝒱 g
      apply (unfold pre-splitFace-def) by auto
    moreover from Cons spl have v ∈ 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 ∈ 𝒱 f') apply (simp add: minGraphProps9)
      apply (simp add: congs-pres-nodes cong-face-def) by auto
```

```
  qed
qed

lemma splitFace-new-f21:
 pre-splitFace g ram1 ram2 oldF newVs ⟹
  minGraphProps g ⟹
 (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs ⟹
  f21 ∉ ℱ g
proof −
  assume props: minGraphProps g
  assume pre: pre-splitFace g ram1 ram2 oldF newVs
  assume spl: (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs
  show ?thesis
  proof (cases newVs)
    case Nil with pre have (ram1, ram2) ∉ edges g
      by (unfold pre-splitFace-def) auto
    moreover from Nil pre
    have (ram1, ram2) ∈ edges f21
     apply (rule-tac splitFace-empty-ram1-ram2-in-f21)
     by (auto simp: Nil[symmetric])
    ultimately show ?thesis by (auto simp add: edges-graph-def)
  next
    case (Cons v vs)
    with pre have v ∉ 𝒱 g
      apply (unfold pre-splitFace-def) by auto
    moreover from Cons spl have v ∈ 𝒱 f21
      by (simp add: splitFace-f21-new-vertices)
    moreover note props
    ultimately show ?thesis by (auto dest: minGraphProps)
  qed
qed

lemma splitFace-new-f21-norm:
 pre-splitFace g ram1 ram2 oldF newVs ⟹
  minGraphProps g ⟹
 (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs ⟹
  normFace f21 ∉ set (normFaces (faces g))
proof −
  assume props: minGraphProps g
  assume pre: pre-splitFace g ram1 ram2 oldF newVs
  assume spl: (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs
  show ?thesis
  proof (cases newVs)
    case Nil with pre have (ram1, ram2) ∉ edges g
      by (unfold pre-splitFace-def) auto
    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) ∈ edges f21
      apply (rule-tac splitFace-empty-ram1-ram2-in-f21)
      by (auto simp: Nil[symmetric])
    moreover
    with dist-f21 have vertices f21 ≠ []
     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 assumption+ apply (elim bexE)
      apply (subgoal-tac (ram1, ram2) ∈ edges f′) apply simp
     apply (subgoal-tac (vertices f21) ≅ (vertices f′)) apply (frule congs-distinct)
      by (simp add: cong-face-def is-nextElem-congs-eq)+
  next
    case (Cons v vs)
    with pre have v ∉ 𝒱 g
      apply (unfold pre-splitFace-def) by auto
    moreover from Cons spl have v ∈ 𝒱 f21
      by (simp add: splitFace-f21-new-vertices)
    moreover note props
    ultimately show ?thesis apply auto
      apply (subgoal-tac (vertices f21) ≠ [])
      apply (frule normFace-in-cong) apply assumption+ apply (erule bexE)
      apply (subgoal-tac v ∈ 𝒱 f′) apply (simp add: minGraphProps9)
      apply (simp add: congs-pres-nodes cong-face-def) by auto
  qed
qed

lemma splitFace-f21-oldF-neq:
 pre-splitFace g ram1 ram2 oldF newVs ⟹
  minGraphProps g ⟹
 (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs ⟹
  oldF ≠ f21
  apply (frule splitFace-new-f21)
  by (auto)

lemma splitFace-f21-oldF-neq-norm:
 pre-splitFace g ram1 ram2 oldF newVs ⟹
  minGraphProps g ⟹
 (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs ⟹
  normFace oldF ≠ normFace f21
  apply (frule splitFace-new-f21-norm) apply simp  apply simp
  apply (subgoal-tac normFace oldF ∈ set (normFaces (faces g))) apply force
  apply (rule normFace-in) by (simp)

lemma splitFace-f12-oldF-neq:
 pre-splitFace g ram1 ram2 oldF newVs ⟹
```

```
  minGraphProps g ⟹
 (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs ⟹
  oldF ≠ f12
  apply (frule splitFace-new-f12)
  by(auto)

lemma splitFace-f12-oldF-neq-norm:
 pre-splitFace g ram1 ram2 oldF newVs ⟹
  minGraphProps g ⟹
 (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs ⟹
  normFace oldF ≠ normFace f12
  apply (frule splitFace-new-f12-norm) apply simp apply simp
  apply (subgoal-tac normFace oldF ∈ set (normFaces (faces g))) apply force
  apply (rule normFace-in) by (simp)


lemma splitFace-f12-f21-neq-norm:
 pre-splitFace g ram1 ram2 oldF vs ⟹ minGraphProps g ⟹
 (f12, f21, newGraph) = splitFace g ram1 ram2 oldF vs ⟹
  normFace f12 ≠ normFace f21
apply (subgoal-tac minGraphProps' newGraph)
 apply (subgoal-tac f12 ∈ ℱ newGraph ∧ f21 ∈ 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:
⟦ minGraphProps g; f ∈ ℱ g; pre-splitFace g v1 v2 f vs;
  (f1, f2, g') = splitFace g v1 v2 f vs ⟧
  ⟹ ℱ g' = {f1,f2} ∪ (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:
 pre-splitFace g v w f [countVertices g..<countVertices g + n] ⟹
  minGraphProps g ⟹
  facesAt-distinct (snd (snd (splitFace g v w f [countVertices g..<countVertices g
+ n])))
proof −
  assume pre: pre-splitFace g v w f [countVertices g..<countVertices g + n]
    and mgp: minGraphProps g
  def ws ≡ [countVertices g..<countVertices g + n]
  def f21 ≡ snd (split-face f v w ws)
  with pre ws-def have dist-f21: distinct (vertices f21) by (auto intro: split-face-distinct2)
  def f12 ≡ fst (split-face f v w ws)
  with pre ws-def have dist-f12: distinct (vertices f12) by (auto intro: split-face-distinct1)
  def vs1 ≡ between (vertices f) v w
  def vs2 ≡ between (vertices f) w v
  def 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 ∉ ℱ g
    apply (rule-tac splitFace-new-f12) by simp-all
  from pre mgp fdg have new-f21: f21 ∉ ℱ g
    apply (rule-tac splitFace-new-f21) by simp-all
  from pre mgp fdg have new-f12-norm: normFace f12 ∉ set (normFaces (faces
g))
    apply (rule-tac splitFace-new-f12-norm) by simp-all
  from pre mgp fdg have new-f21-norm: normFace f21 ∉ 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 ∈ 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 ≠ 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)

            apply (subgoal-tac set (faceListAt g ! w) = {f ∈ 𝓕 g. w ∈ 𝒱 f}) apply
simp
             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 ∈ 𝒱 g) apply assumption
              apply (rule minGraphProps9) apply assumption apply blast apply
simp
             apply (simp add: facesAt-def split: split-if-asm)

            apply (simp add: normFaces-def)

           apply (subgoal-tac w ∉ set (between (vertices f) v w))
            apply (simp add: replacefacesAt-nth1)
            apply (subgoal-tac set (faceListAt g ! w) = {f ∈ 𝓕 g. w ∈ 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 (subgoal-tac w ∈ 𝒱 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-nth1)
           apply (subgoal-tac w ∉ set (between (vertices f) v w))
            apply (simp add: replacefacesAt-nth1)
            apply (subgoal-tac set (faceListAt g ! w) = {f ∈ ℱ g. w ∈ 𝒱 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 ∈ 𝒱 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 ∈ 𝒱 f})
```

apply (*subgoal-tac set* (*normFaces* (*faces g*)) $\cap$ {*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* $v \in \mathcal{V}$ *g*) **apply** *assumption*

apply (*rule minGraphProps9*) **apply** *assumption* **apply** *blast* **apply** *simp*

apply (*simp add*: *facesAt-def split*: *split-if-asm*)

apply (*simp add*: *normFaces-def*)

apply (*subgoal-tac* $v \notin$ *set* (*between* (*vertices f*) *v w*))

apply (*simp add*: *replacefacesAt-nth1*)

apply (*subgoal-tac set* (*faceListAt g* ! *v*) = {$f \in \mathcal{F}$ *g*. $v \in \mathcal{V}$ *f*})

apply (*subgoal-tac set* (*normFaces* (*faces g*)) $\cap$ {*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* $v \in \mathcal{V}$ *g*) **apply** *assumption*

apply (*rule minGraphProps9*) **apply** *assumption* **apply** *blast* **apply** *simp*

apply (*simp add*: *facesAt-def split*: *split-if-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* $v \notin$ *set* (*between* (*vertices f*) *w v*))

apply (*simp add*: *replacefacesAt-nth1*)

apply (*subgoal-tac* $v \notin$ *set* (*between* (*vertices f*) *v w*))

apply (*simp add*: *replacefacesAt-nth1*)

apply (*subgoal-tac set* (*faceListAt g* ! *v*) = {$f \in \mathcal{F}$ *g*. $v \in \mathcal{V}$ *f*})

apply (*subgoal-tac set* (*normFaces* (*faces g*)) $\cap$ {*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*)) $\cap$ {*normFace f21*} = {})

apply (*simp add*: *new-f21-norm*)

apply (*frule minGraphProps-facesAt-eq*)

apply (*subgoal-tac* $v \in \mathcal{V}$ *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: split-if-asm)
                    apply (simp add: normFaces-def)

                   apply (case-tac x ∉ set (between (vertices f) v w))
                    apply (simp add: replacefacesAt-nth1)
                    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: split-if-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 force
     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 blast
    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 force
     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 blast
    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 x* ∈ 𝒱 *g*) **apply** (*simp add*: *normFaces-def*)
apply (*simp add*: *minGraphProps4 vertices-graph*)
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*)
apply (*simp add*: *minGraphProps4*) **by** *arith*
qed
qed
qed
qed
then show *?thesis* **by** (*simp add*: *g′-def*)
qed

lemma *splitFace-holds-facesAt-eq*:
pre-splitFace g′ v a f′ [*countVertices g′..<countVertices g′* + *n*] ⟹
minGraphProps g′ ⟹
g″ = (*snd* (*snd* (*splitFace g′ v a f′* [*countVertices g′..<countVertices g′* + *n*]))) ⟹
facesAt-eq g″
proof −
assume *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*])))
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*)

def *ws* ≡ [*countVertices g′*..<*countVertices g′* + *n*]
def *f21* ≡ *snd* (*split-face f′ v a ws*)
def *f12* ≡ *fst* (*split-face f′ v a ws*)
def *vs1* ≡ *between* (*vertices f′*) *v a*
def *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′* $\neq$ *f21* **apply** (*rule-tac splitFace-f21-oldF-neq*) **apply** *assumption* **by** *simp*+
from *pre-F′ mgp fdg* **have** *f′-f12*: *f′* $\neq$ *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′* $\Longrightarrow$ *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.simps split:split-if-asm)
  apply (case-tac v < |faceListAt g′| ∧ a < | faceListAt g′|)
   apply (simp only: nth-append split: split-if-asm)
    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 (case-tac a = va)
        apply (simp add:replacefacesAt-nth2 replacefacesAt-nth1)
        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 simp apply (fastsimp simp:
facesAt-def)
         apply (rule minGraphProps6) apply simp 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 (case-tac v = va)
        apply (simp add:replacefacesAt-nth2 replacefacesAt-nth1)
        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 simp apply (fastsimp simp:
facesAt-def)
               apply (rule minGraphProps6) apply simp apply (fastsimp 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 (case-tac va ∈ set vs1)
        apply (subgoal-tac va ∉ set vs2)
      apply (simp add:replacefacesAt-nth2 replacefacesAt-nth1 replacefacesAt-nth1')
         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 simp apply (fastsimp simp:
facesAt-def)
               apply (rule minGraphProps6) apply simp apply (fastsimp 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 blast
       apply (case-tac va ∈ set vs2)
      apply (simp add:replacefacesAt-nth2 replacefacesAt-nth1 replacefacesAt-nth1')
        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 simp apply (fastsimp simp:
facesAt-def)
              apply (rule minGraphProps6) apply simp apply (fastsimp 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 add:replacefacesAt-nth2 replacefacesAt-nth1 replacefacesAt-nth1')
       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 simp apply (fastsimp simp:
facesAt-def)
```

apply (*rule minGraphProps6*) **apply** *simp* **apply**(*fastsimp 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 (*subgoal-tac distinct* (*vertices f12*) $\wedge$ *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* $g' + n$] $\equiv$ *ws*)
apply (*subgoal-tac* (*va* $-$ |*faceListAt* g'|) $<$ | *ws* |) **apply** *simp* **apply** (*rule conjI*) **apply** *blast*
apply (*subgoal-tac va* $\in$ *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** *arith*
apply (*rule minGraphProps11'*) **apply** *simp*
apply (*subgoal-tac* $v \in \mathcal{V}\ g' \wedge a \in \mathcal{V}\ g'$)
apply (*simp only*: *fla in-vertices-graph*)
apply (*subgoal-tac* $f' \in \mathcal{F}\ g'$)
apply (*subgoal-tac* $v \in \mathcal{V}\ f' \wedge a \in \mathcal{V}\ f'$) **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 *simp*

apply (*rule subsetI*)
apply (*case-tac* $v <$ |*faceListAt* g'| $\wedge$ $a <$ | *faceListAt* g'|)
apply (*case-tac va* $<$ | *faceListAt* g' |)
apply (*subgoal-tac va* $\in \mathcal{V}\ g'$)
apply (*subgoal-tac distinct vs1* $\wedge$ *distinct vs2* $\wedge$
$v \notin$ *set vs1* $\wedge$ $v \notin$ *set vs2* $\wedge$ $a \notin$ *set vs1* $\wedge$ $a \notin$ *set vs2* $\wedge$ $a \neq v \wedge v \neq a \wedge$ *set vs1* $\cap$ *set vs2* = {})
apply (*simp del*: *replacefacesAt-simps add*: *nth-append*)
apply (*case-tac* a = *va*)
apply (*simp add*:*replacefacesAt-nth2 replacefacesAt-nth1*)
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 v = va)
      apply (simp add:replacefacesAt-nth2 replacefacesAt-nth1)
      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-nth1 replacefacesAt-nth1')
       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-nth1 replacefacesAt-nth1')
      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* $f' \neq f12$) **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* $[countVertices\ g'..<countVertices\ g' + n] \equiv ws$)
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-nth1 replacefacesAt-nth1'*)
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* $[countVertices\ g'..<countVertices\ g' + n] \equiv ws$)
apply (*simp add*: *ws-def vertices-graph*)
apply (*case-tac* $x = f12$) **apply** (*simp add*: *vert-f12*) **apply** (*thin-tac* $[countVertices\ g'..<countVertices\ g' + n] \equiv ws$)
apply (*simp add*: *ws-def vertices-graph*)
apply *simp* **apply** *simp* **apply** (*rule minGraphProps11'*) **apply** *simp*
apply (*subgoal-tac distinct* (*vertices f12*) $\wedge$ *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 add*: *nth-append del*:*replacefacesAt-simps*)
apply (*subgoal-tac distinct* (*faces g'*))
apply (*simp add*: *replace6*)
apply (*thin-tac* $[countVertices\ g'..<countVertices\ g' + n] \equiv ws$)
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* $va \in \mathcal{V}\ g'$) **apply** (*simp add*: *fla vertices-graph*)
apply (*rule minGraphProps9*) **apply** *simp* **apply** *force* **apply** *simp*
apply (*simp add*: *ws-def fla*) **apply** *arith*
apply (*rule minGraphProps11'*) **apply** *simp*
apply (*subgoal-tac* $v \in \mathcal{V}\ g' \wedge a \in \mathcal{V}\ g'$)
apply (*simp only*: *fla in-vertices-graph*)
apply (*subgoal-tac* $f' \in \mathcal{F}\ g'$)
apply (*subgoal-tac* $v \in \mathcal{V}\ f' \wedge a \in \mathcal{V}\ f'$) **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*
by (*simp only*: *pre-splitFace-def*)
qed

lemma *splitFace-holds-faces-subset*:
pre-splitFace g' v a f' [*countVertices g'*..<*countVertices g'* + *n*] ⟹
minGraphProps g' ⟹
faces-subset (*snd* (*snd* (*splitFace g' v a f'* [*countVertices g'*..<*countVertices g'* + *n*])))
proof −
assume *pre-F*: *pre-splitFace g' v a f'* [*countVertices g'*..<*countVertices g'* + *n*]
and *mgp*: *minGraphProps g'*
def *g''* ≡ (*snd* (*snd* (*splitFace g' v a f'* [*countVertices g'*..<*countVertices g'* + *n*])))
def *ws* ≡ [*countVertices g'*..<*countVertices g'* + *n*]
def *f21* ≡ *snd* (*split-face f' v a ws*)
def *f12* ≡ *fst* (*split-face f' v a ws*)
def *vs1* ≡ *between* (*vertices f'*) *v a*
def *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* ∪ *set vs2* ∪ {*a*,*v*}
apply (*subgoal-tac* (*vertices f'*) ≅ (*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'* ⊆ $\mathcal{V}$ *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** *assumption+* **apply** *simp*
apply (*rule subsetI*) **apply** (*rule minGraphProps9*) **by** *auto*
qed

lemma *splitFace-holds-edges-sym*:
pre-splitFace g' v a f' ws ⟹
minGraphProps g' ⟹
edges-sym (*snd* (*snd* (*splitFace g' v a f' ws*)))
proof −
assume *pre-F*: *pre-splitFace g' v a f' ws*
and *mgp*: *minGraphProps g'*
def *g''* ≡ (*snd* (*snd* (*splitFace g' v a f' ws*)))
def *f21* ≡ *snd* (*split-face f' v a ws*)
def *f12* ≡ *fst* (*split-face f' v a ws*)
def *vs1* ≡ *between* (*vertices f'*) *v a*
def *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* ∨ *x* = *f21*)
apply (*subgoal-tac* (*aa*,*b*) ∈ *edges f'* ∨ ((*b*,*aa*) ∈ (*edges f12* ∪ *edges f21*) ∧ (*aa*,*b*) ∈ (*edges f12* ∪ *edges f21*))) **apply** *simp*
apply (*case-tac* (*aa*, *b*) ∈ *edges f'*)
apply (*subgoal-tac* (*b*,*aa*) ∈ *edges g'*)
apply (*simp add*: *edges-graph-def*) **apply** (*elim bexE*) **apply** (*rule disjI2*) **apply** (*rule bexI*)
apply *simp* **apply** (*subgoal-tac xa* ≠ *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*) ∈ *edges x*) **apply** *simp* **apply** *simp*
apply *simp*
apply (*case-tac* (*b*, *aa*) ∈ *edges f12*) **apply** *simp* **apply** *simp*

apply (*case-tac* (*b*, *aa*) ∈ *edges f21*) **apply** (*rule bexI*)
apply *simp* **apply** (*rule replace3*) **apply** *simp* **apply** *simp* **apply** *simp*

apply (*subgoal-tac*
((*aa*,*b*) ∈ *edges f'* ∨ ((*b*,*aa*) ∈ (*edges f12* ∪ *edges f21*) ∧ (*aa*,*b*) ∈ (*edges f12* ∪ *edges f21*))) = ((*aa*,*b*) ∈ *edges f12* ∨ (*aa*,*b*) ∈ *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 bexI*) **apply** (*thin-tac* (*b*, *aa*) ∈ *edges f'*)
apply *simp* **apply** (*rule replace3*) **apply** *simp* **apply** *simp*
apply (*rule disjI2*) **apply** (*rule bexI*) **apply** *simp*
apply (*rule replace4*) **apply** *simp* **apply** *force*
apply (*rule minGraphProps10*) **apply** *simp* **apply** (*simp add*: *edges-graph-def*)
apply (*rule bexI*) **apply** *simp* **apply** *simp*
apply (*rule minGraphProps11'*) **by** *simp*
qed

lemma *splitFace-holds-faces-distinct*:
pre-splitFace g' v a f' [*countVertices g'*..<*countVertices g'* + *n*] ⟹
minGraphProps g' ⟹
faces-distinct (*snd* (*snd* (*splitFace g' v a f'* [*countVertices g'*..<*countVertices g'* + *n*])))
proof −
assume *pre-F*: *pre-splitFace g' v a f'* [*countVertices g'*..<*countVertices g'* + *n*]
and *mgp*: *minGraphProps g'*
def *g''* ≡ (*snd* (*snd* (*splitFace g' v a f'* [*countVertices g'*..<*countVertices g'* + *n*])))
def *ws* ≡ [*countVertices g'*..<*countVertices g'* + *n*]
def *f21* ≡ *snd* (*split-face f' v a ws*)
def *f12* ≡ *fst* (*split-face f' v a ws*)
def *vs1* ≡ *between* (*vertices f'*) *v a*
def *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* $f' \neq$ *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* $\neq$ *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* $\notin$ *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* $\notin$ *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 $xs \neq [] \Longrightarrow x \notin set\ xs \Longrightarrow x \neq hd\ xs$ **and**
$xs \neq [] \Longrightarrow x \notin set\ xs \Longrightarrow x \neq last\ xs$ **and**
$xs \neq [] \Longrightarrow x \notin set\ xs \Longrightarrow hd\ xs \neq x$ **and**
$xs \neq [] \Longrightarrow x \notin set\ xs \Longrightarrow last\ xs \neq x$
by(*auto*)

lemma *split-face-edge-disj*:
$[\![$ *pre-split-face f a b vs*; (f_1, f_2) = *split-face f a b vs*; $|vertices\ f| \geq 3$;
$vs = [] \longrightarrow (a,b) \notin edges\ f \wedge (b,a) \notin edges\ f$ $]\!]$
$\Longrightarrow \mathcal{E}\ f_1 \cap \mathcal{E}\ f_2 = \{\}$
apply(*frule pre-split-face-p-between*[*THEN between-inter-empty*])
apply(*unfold pre-split-face-def*)
apply *clarify*

```
apply(subgoal-tac !!x y. x ∈ set vs ⟹ y ∈ 𝒱 f ⟹ x ≠ y)
 prefer 2 apply blast
apply(subgoal-tac !!x y. x ∈ set vs ⟹ y ∈ 𝒱 f ⟹ y ≠ x)
 prefer 2 apply blast
apply(subgoal-tac a ∉ set vs)
 prefer 2 apply blast
apply(subgoal-tac b ∉ set vs)
 prefer 2 apply blast
apply(subgoal-tac distinct(vs @ a # between (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 distinct(b # between (vertices f) b a @ a # rev vs))
 prefer 2 apply(simp add:between-not-r1 between-not-r2 between-distinct)
 apply(blast dest:inbetween-inset)
apply(subgoal-tac vs = [] ⟹ between (vertices f) a b ≠ [])
 prefer 2 apply clarsimp apply(frule (4) is-nextElem-between-empty')apply blast
apply(subgoal-tac vs = [] ⟹ between (vertices f) b a ≠ [])
 prefer 2 apply clarsimp
apply(frule (3) is-nextElem-between-empty')apply simp apply blast
apply(subgoal-tac vs ≠ [] ⟹ hd vs ∉ 𝒱 f)
 prefer 2 apply(drule hd-in-set) apply blast
apply(subgoal-tac vs ≠ [] ⟹ last vs ∉ 𝒱 f)
 prefer 2 apply(drule last-in-set) apply blast
apply(subgoal-tac !!u v. between (vertices f) u v ≠ [] ⟹ hd(between (vertices f)
u v) ∈ 𝒱 f)
 prefer 2 apply(drule hd-in-set)apply(drule inbetween-inset) apply blast
apply(subgoal-tac !!u v. between (vertices f) u v ≠ [] ⟹ last (between (vertices
f) u v) ∈ 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') = $ *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* ∩ *set vs* = {} **using** *pre*
    by (*simp add*: *pre-splitFace-def*)
  hence *gvs*: ∀ *f* ∈ $\mathcal{F}$ *g*. $\mathcal{V}$ *f* ∩ *set vs* = {}
    by (*clarsimp simp*:*edges-graph-def edges-face-def*)
      (*blast dest*: *minGraphProps9*[*OF mgp*])
  have *f*: *f* ∈ $\mathcal{F}$ *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'* ∈ $\mathcal{F}$ *g f'* ≠ *f*
    have $(\mathcal{E}\ f_1 \cup \mathcal{E}\ f_2) \cap \mathcal{E}\ f' = \{\}$
    proof *cases*
      assume *vs*: *vs* = []
      have (*u*,*v*) ∉ $\mathcal{E}$ *g* ∧ (*v*,*u*) ∉ $\mathcal{E}$ *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* ≠ []
      have *f12*: *vs* ≠ [] ⟹ $\mathcal{E}\ f_1 \cup \mathcal{E}\ f_2 \subseteq$
          $\mathcal{E}$ *f* ∪ *UNIV* × *set vs* ∪ *set vs* × *UNIV*
        using *split-face-edges-f12-f21*[*OF pre-split-face split-face*]
        by *simp* (*fastsimp dest*:*in-Edges-in-set*)
      have !!*x y*. (*y*,*x*) ∈ $\mathcal{E}$ *f'* ⟹ *x* ∉ *set vs* ∧ *y* ∉ *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 ⟹ pre-splitFace g u v f vs
 ⟹ 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) ⟹ 𝒱(f::face) = (⋃(a,b)∈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': ℱ g' = {f₁} ∪ 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)
  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: ¬ final f and fg: f ∈ ℱ g and nuv: u ≠ v
    and uinf: u ∈ 𝒱 f and vinf: v ∈ 𝒱 f
    and distf: distinct(vertices f) and new: 𝒱 g ∩ set vs = {}
    by(unfold pre-splitFace-def, simp)+
  let ?fuv = between (vertices f) u v and ?fvu = between (vertices f) v u
  have E₁: ℰ f₁ = Edges (v # rev vs @ [u]) ∪ Edges (u # ?fuv @ [v])
    using f₁ by(simp add:edges-split-face1[OF pre-split])
  have E₂: ℰ f₂ = Edges (u # vs @ [v]) ∪ 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: 𝒱 f₁ = {u,v} ∪ set(?fuv) ∪ set(vs) using vf₁ by auto
```

```
have V_2: 𝒱 f_2 = {u,v} ∪ set(?fvu) ∪ set(vs) using vf_2 by auto
have 2: (v,u) ∈ ℰ f_1 ∧ (u,v) ∈ ℰ f_2 ∧ vs = [] ∨
        (∃ v ∈ 𝒱 f_1 ∩ 𝒱 f_2. v ∉ 𝒱 g)
  using E_1 E_2 V_1 V_2 new by(cases vs)(simp-all add:Edges-Cons)
have 𝒱 f_1 ≠ 𝒱 f_2
proof cases
  assume A: ?fvu = []
  have ?fuv ≠ []
  proof
    assume ?fuv = []
    with A have ℰ f = {(v,u),(u,v)}
      using edges-conv-Un-Edges[OF distf uinf vinf nuv]
      by(simp add:Edges-Cons)
    hence 𝒱 f = {u,v} by(simp add:vertices-conv-Union-edges)
    hence card(𝒱 f) ≤ 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 set ?fuv ∩ set vs = {}
    using new minGraphProps9[OF mgp fg inbetween-inset] by blast
  moreover have {u,v} ∩ set ?fuv = {}
    using between-not-r1[OF distf] between-not-r2[OF distf] by blast
  ultimately show ?thesis using V_1 V_2 A by (auto simp:neq-Nil-conv)
next
  assume ?fvu ≠ []
  moreover have {u,v} ∩ set ?fvu = {}
    using between-not-r1[OF distf] between-not-r2[OF distf] by blast
  moreover have set ?fuv ∩ set ?fvu = {}
    by(simp add:pre-between-def between-inter-empty distf uinf vinf nuv)
  moreover have set ?fvu ∩ set vs = {}
    using new minGraphProps9[OF mgp fg inbetween-inset] by blast
  ultimately show ?thesis using V_1 V_2 by (auto simp:neq-Nil-conv)
qed
have C12: ℰ f_1 ≠ (ℰ f_2)^-1
proof
  assume A: ℰ f_1 = (ℰ f_2)^-1
  show False
  proof -
    have 𝒱 f_1 = (⋃(a,b)∈ℰ f_1. {a})
      by(rule vertices-conv-Union-edges)
    also have ... = (⋃(b,a)∈ℰ f_2. {a}) by(auto simp:A)
    also have ... = 𝒱 f_2
      by(rule vertices-conv-Union-edges2[OF distf_2, symmetric])
    finally show False using ‹𝒱 f_1 ≠ 𝒱 f_2› by blast
  qed
qed
{ fix h :: face assume hg: h ∈ ℱ g
  have ℰ h ≠ (ℰ f_1)^-1 ∧ ℰ h ≠ (ℰ f_2)^-1 using 2
  proof
```

```
      assume (v,u) ∈ ℰ f1 ∧ (u,v) ∈ ℰ f2 ∧ vs = []
      moreover hence (u,v) ∉ ℰ g
        using pre by(unfold pre-splitFace-def)simp
      moreover hence (v,u) ∉ ℰ g by(blast intro:minGraphProps10[OF mgp])
      ultimately show ?thesis using hg by(simp add:edges-graph-def) blast
    next
      assume ∃ x ∈ 𝒱 f1 ∩ 𝒱 f2. x ∉ 𝒱 g
      then obtain x where x ∈ 𝒱 f1 and x ∈ 𝒱 f2 and x ∉ 𝒱 g
        by blast
      obtain y where (x,y) ∈ ℰ f1 using ‹x ∈ 𝒱 f1›
        by(auto simp:vertices-conv-Union-edges)
      moreover obtain z where (x,z) ∈ ℰ f2 using ‹x ∈ 𝒱 f2›
        by(auto simp:vertices-conv-Union-edges)
      moreover have ¬(EX y. (y,x) ∈ ℰ h)
        using ‹x ∉ 𝒱 g› 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: |faces g| = 2
    with fg obtain f' where Fg: ℱ g = {f,f'}
      by(fastsimp simp: nat-number length-Suc-conv)
    moreover hence f ≠ f' using 2 distinct-card[OF distF] by auto
    ultimately have Fg': ℱ g' = {f1,f2,f'}
      using set-faces-splitFace[OF mgp fg pre fdg] by blast
    show ?thesis using Fg' C12 Cg12 Fg
      by(fastsimp simp:face-face-op-def)
  next
    assume |faces g| ≠ 2
    hence E: !!f f'. f∈ℱ g ⟹ f'∈ℱ g ⟹ f ≠ f' ⟹ ℰ f ≠ (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(fastsimp simp:face-face-op-def)
  qed
qed

lemma splitFace-face-face-op2:
 minGraphProps g ⟹ pre-splitFace g u v f vs
 ⟹ 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:
 pre-splitFace g' v a f' [countVertices g'..<countVertices g' + n] ⟹
```

minGraphProps g' $\Longrightarrow$
minGraphProps (*snd* (*snd* (*splitFace g' v a f'* [*countVertices g'*..<*countVertices g'* + *n*])))
proof −
assume *precond*: *pre-splitFace g' v a f'* [*countVertices g'*..<*countVertices g'* + *n*]
minGraphProps g'
then have *minGraphProps' g'* **by** (*simp add*: *minGraphProps-def*)
then show *?thesis* **apply** (*simp add*: *minGraphProps-def*) **apply** *safe*
apply (*rule splitFace-holds-minGraphProps'*) **apply** *assumption+*
apply (*rule splitFace-holds-facesAt-eq*) **apply** *assumption+* **apply** *simp*
apply (*rule splitFace-holds-faceListAt-len*) **apply**(*simp add*:*precond*) **apply** *assumption+*
apply (*rule splitFace-holds-facesAt-distinct*) **apply** *assumption+*
apply (*rule splitFace-holds-faces-distinct*) **apply** *assumption+*
apply (*rule splitFace-holds-faces-subset*) **apply** *assumption+*
apply (*rule splitFace-holds-edges-sym*) **apply**(*simp add*:*precond*) **apply** *assumption*
apply (*rule splitFace-edges-disj2*) **apply** *assumption+*
apply (*rule splitFace-face-face-op2*) **apply** *assumption+*
done
qed

13.7 Invariants of *makeFaceFinal*

lemma *MakeFaceFinal-minGraphProps'*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *minGraphProps'* (*makeFaceFinal f g*)
apply (*simp add*: *minGraphProps-def minGraphProps'-def makeFaceFinal-def*)
apply (*subgoal-tac* $2 < |$*vertices f*$| \wedge$ *distinct* (*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*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *facesAt-eq* (*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* $v < |$ *faceListAt g* $|$)
apply (*simp add*: *makeFaceFinalFaceList-def*)
apply (*subgoal-tac distinct* ((*faceListAt g* ! *v*)))
apply (*subgoal-tac distinct* (*faces g*))
apply (*simp add*: *replace6*)
apply (*case-tac* $x = f$) **apply** *simp* **apply** (*erule minGraphProps6*)
apply (*simp add*: *facesAt-def*) **apply** *force* **apply** *simp*
apply (*case-tac* $f \in$ *set* (*faceListAt g* ! *v*) $\wedge\ x =$ *setFinal f*) **apply** *simp*
apply (*subgoal-tac* $v \in \mathcal{V}\ f$) **apply** (*simp add*: *setFinal-def*)
apply (*erule minGraphProps6*) **apply** (*simp add*: *facesAt-def*) **apply** *simp*

apply (*rule conjI*) **apply** (*rule disjI2*)
apply (*erule minGraphProps5*) **apply** (*fastsimp simp*: *facesAt-def*)
apply (*erule minGraphProps6*) **apply** (*fastsimp 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* $v < |$ *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 = *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 minGraphProps4*)

lemma *MakeFaceFinal-faceListAt-len*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *faceListAt-len* (*makeFaceFinal f g*)
apply (*simp add*: *faceListAt-len-def makeFaceFinal-def*) **apply** (*rule minGraphProps4*) **by** *simp*

lemma *normFaces-makeFaceFinalFaceList*: (*normFaces* (*makeFaceFinalFaceList f fs*)) = (*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*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *facesAt-distinct* (*makeFaceFinal f g*)
apply (*simp add*: *facesAt-distinct-def makeFaceFinal-def*)
apply (*rule allI*) **apply** (*rule impI*)
apply (*simp add*: *facesAt-def*)
apply (*subgoal-tac* $v < |$ (*faceListAt g*) $|$) **apply** (*simp add*: *normFaces-makeFaceFinalFaceList*)
apply (*rule minGraphProps8a'*) **apply** *simp* **apply** *simp* **by** (*simp add*: *minGraphProps4*)

lemma *MakeFaceFinal-faces-subset*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *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 ∈ ℱ g ⟹ minGraphProps g ⟹ edges-sym (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 ∈ ℱ g ⟹ minGraphProps g ⟹ faces-distinct (makeFaceFinal f g)
 apply (simp add: faces-distinct-def makeFaceFinal-def normFaces-makeFaceFinalFaceList)
  by (rule minGraphProps11)
```

```
lemma MakeFaceFinal-edges-disj:
f ∈ ℱ g ⟹  minGraphProps g ⟹ edges-disj (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 (fastsimp dest:mgp-edges-disj)
apply (fastsimp dest:mgp-edges-disj)
done
```

```
lemma MakeFaceFinal-face-face-op:
f ∈ ℱ g ⟹ minGraphProps g ⟹ face-face-op (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 ∈ ℱ g ⟹ minGraphProps g ⟹ minGraphProps (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: ⋀ f v' v n g.
  pre-subdivFace' g f v' v n ovl ⟹ f ∈ ℱ g ⟹
  minGraphProps g ⟹ minGraphProps (subdivFace' g f v n 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: split-if-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
```

```
syntax Edges-if :: face ⇒ vertex ⇒ vertex ⇒ (vertex × vertex)set
translations
Edges-if f u v =>
if u=v then {} else Edges(u # between (vertices f) u v @ [v])

lemma FaceDivsionGraph-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 ≠ v
and ruv: before (verticesFrom f r) u v and rf: r ∈ 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: ℱ g' = {f₁} ∪ 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: ¬ final f and fg: f ∈ ℱ g and nuv: u ≠ v
    and uinf: u ∈ 𝒱 f and vinf: v ∈ 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≠u ⟹ 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 E_1: ℰ f_1 = Edges (v # rev vs @ [u]) ∪ Edges (u # ?fuv @ [v])
  using f_1 by(simp add:edges-split-face1[OF pre-split])
have E_2: ℰ f_2 = Edges (u # vs @ [v]) ∪ Edges (v # ?fvu @ [u])
  using f_2 by(simp add:edges-split-face2[OF pre-split])
have vf_2: vertices f_2 = [v] @ ?fvu @ u # vs
  using f_2 by(simp add:split-face-def)
have vinf_2: v ∈ 𝒱 f_2 using vf_2 by(simp)
have rinf_2: r ∈ 𝒱 f_2
proof cases
  assume r=u thus ?thesis by(simp add:vf_2)
next
  assume r≠u thus ?thesis by(simp add: vf_2 fvu)
qed
have distf_2: distinct(vertices f_2)
  by(simp add:f_2)(rule split-face-distinct2'[OF pre-split])
have f_2uv: between (vertices f_2) u v = vs
  using vf_2 distf_2 by(simp add:between-def split-def)
have f_2ru: r≠u ⟹ between (vertices f_2) r u = between (vertices f) r u
  using vf_2 fvu distf distf_2 by(simp add:between-def split-def)
hence f_2rv: between (vertices f_2) r v =
          (if r=u then [] else ?fru @ [u]) @ vs
proof cases
  assume r=u thus ?thesis by(simp add: f_2uv)
next
  assume nru: r ≠ u
  have vinf_2: v ∈ 𝒱 f_2 by(simp add: vf_2)
  note u-bet-rv = before-between[OF ruv distf rf nru]
  have u-bet-rv_2: u ∈ set (between (vertices f_2) r v)
    using distf_2 nru
    apply(simp add:vf_2 fvu)
    apply(subst between-def[of - r v])
    apply(simp add:split-def)
    done
  show ?thesis
    by(simp add:split-between[OF distf_2 rinf_2 vinf_2 u-bet-rv_2] f_2ru f_2uv)
qed
have E_2rv: Edges(r # ?f_2rv @ [v]) =
       Edges-if f r u ∪ Edges(u # vs @ [v]) (is ?L = ?R)
proof -
  have ?L = Edges((if r=u then [] else r # ?fru) @ (u # vs @ [v]))
    by (simp add: f_2rv)
  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)
  case (goal1 a b)
  have ab: (a,b) ∈ ℰ f_1
    and nab: (a,b) ∉ Edges (r # ?f_2rv @ [v]) .
```

```
  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 abfuv: ?B
    have abf: (a,b) ∈ ℰ f
      by(rule Edges-between-edges[OF abfuv 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 abfuv Edges-disj[OF distf rf vinf 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 (goal2 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]) .
  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′ ∈ set (replace f [f₂] (faces g))
          by(clarsimp simp: replace6[OF distFg] nf)
        ultimately show ?thesis by blast
      qed
      thus ?thesis ..
    next
      assume f′: f′ ∈ ℱ g ∧ f′ ≠ f
      moreover
      hence ℰ f′ ∩ ℰ f = {}
        using fg by(blast dest: mgp-edges-disj[OF mgp])
      moreover
      have Edges-if f r u ⊆ 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) : Edges-if f r u ∨
            (∃f″∈ℱ g. final f″ ∧ (b,a) ∈ ℰ f″) (is ?A ∨ ?B)
        using 1 f′ nf′ abf′
        by(simp add:one-final-but-def split:split-if-asm) blast+
      thus ?thesis (is ?A′ ∨ ?B′)
      proof
        assume ?A
        moreover
        have Edges-if f r u ⊆ Edges (r # between (vertices f₂) 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″∈ℱ g ∧ final f″ ∧ (b, a) ∈ ℰ f″
          by blast
        moreover hence f″ ≠ 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 ⊆ ℰ f; f ∈ ℱ g; ¬ final f ⟧ ⟹
  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) ∉ 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(fastsimp simp add: replace6)
done


lemma one-final-subdivFace′:
  ⋀f v n g.
  pre-subdivFace′ g f u v n ovs ⟹ minGraphProps g ⟹ f ∈ ℱ g ⟹
  one-final-but g (Edges-if f u v) ⟹
  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]) ⊆ ℰ f
    by(auto simp add:Edges-between-edges)
  moreover have ¬ 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: ¬ final f and uf: u ∈ 𝒱 f and vf: v ∈ 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 ≠ w using pre Some by(clarsimp simp: pre-subdivFace′-def)
    { assume w: f · 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(fastsimp simp add:Edges-Cons Edges-append)
          done
      have pre': pre-subdivFace' g f u w 0 ovs
        using pre Some n by (simp (depth-limit:5) 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 FaceDivsionGraph-one-final-but[OF mgp pre-fdg - uw bet uf 1])
        apply(fastsimp 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:
 minGraphProps g ⟹ b ∈ set (neighbors g a) = ((a, b) ∈ edges g)
apply(cases a ∈ 𝒱 g)
```

```
 apply (rule iffI)
  apply (simp add: neighbors-def) apply clarify apply (frule minGraphProps5)
   apply (simp add: vertices-graph)
  apply (simp add: edges-graph-def) apply (intro bexI)
   prefer 2 apply assumption
  apply(simp add:edges-face-eq)
  apply (erule (1) minGraphProps6)
 apply (simp add: neighbors-def) apply (simp add: edges-graph-def) apply (elim
bexE)
 apply (subgoal-tac x ∈ set (facesAt g a)) apply (simp add: edges-face-def)
 apply (rule minGraphProps7) apply simp+ apply (simp add: edges-face-def)
apply(auto simp: neighbors-def image-def edges-graph-def)
apply(blast dest:in-edges-in-vertices  minGraphProps6 minGraphProps9)
done
```

```
lemma no-self-edges: minGraphProps′ g ⟹ (a, a) ∉ edges g apply (simp add:
minGraphProps′-def)
apply (induct g) apply simp apply (simp add: edges-graph-def) apply auto apply (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)
apply simp by simp
```

Requires only *distinct* (*vertices f*) and that *g* has no self-loops.

```
lemma duplicateEdge-is-duplicateEdge-eq:
minGraphProps g ⟹ f ∈ ℱ g ⟹ a ∈ 𝒱 f ⟹ b ∈ 𝒱 f ⟹
 duplicateEdge g f a b = is-duplicateEdge g f a b
apply (subgoal-tac distinct (vertices f))
 prefer 2 apply (simp add: minGraphProps3)
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 $\Longrightarrow$ *Suc n* < |*ls*| $\Longrightarrow$ *ls*!*n* ≤ *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 drop-Suc-conv-tl*) **by** *auto*

lemma *incrIndexList-less*:
incrIndexList ls m nmax $\Longrightarrow$ *Suc n* < |*ls*| $\Longrightarrow$ *ls*!*n* ≠ *ls*!*Suc n* $\Longrightarrow$ *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*..*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*..*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*: *nat-number 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*]]
apply (*fastsimp simp add*: *Seed-def graph-def edges-disj-def edges-graph-def*)
done

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*: *nat-number 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*: $|\mathit{faces}\ \mathit{Seed}_p| \geq 2$
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* ¬ *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: split-if-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 f^(e ! 0) · v ∈ 𝒱 f)
      apply (subgoal-tac f^(e ! 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 f^(e ! Suc nat) · v))
     apply (simp split: split-if-asm)
     apply (subgoal-tac natToVertexList v f e ! Suc (Suc nat) =
       (if e ! (Suc nat) = e ! Suc (Suc nat) then None else Some f^(e ! Suc (Suc nat))
· v))
      apply (simp split: split-if-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 f^(e ! Suc nat) • v ∈ 𝒱 f)
     apply (subgoal-tac f^(e ! 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: !!x y z.  Q x y ⟹ P y z ⟹ P x z
and mkFin: !!f g. f ∈ ℱ g ⟹ ¬ final f ⟹ P g (makeFaceFinal f g)
and fdg-incr: !! g u v f vs.
   pre-splitFace g u v f vs ⟹
   Q g (snd(snd(splitFace g u v f vs)))
shows
⋀f′ v n g. pre-subdivFace′ g f′ v′ v n ovl ⟹
 minGraphProps g ⟹  f′ ∈ ℱ g ⟹ P g (subdivFace′ g f′ v n 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-plane0_p]→ g'
and P: ⋀f v' v n g ovs. minGraphProps g ⟹ pre-subdivFace' g f v' v n ovs ⟹
  f ∈ ℱ g ⟹ P g (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: ¬ containsDuplicateEdge g f v e
    and is-eq: is = indexToVertexList f v e
   by (auto simp: next-plane0-def generatePolygon-def image-def split:split-if-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(fastsimp 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: !!x y z. Q x y ⟹ P y z ⟹ P x z
and mkFin: !!f g. f ∈ ℱ g ⟹ ¬ final f ⟹ P g (makeFaceFinal f g)
and fdg-incr: !! g u v f vs.
   pre-splitFace g u v f vs ⟹
```

```
  Q g (snd(snd(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 ∈ 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 ⟹ minGraphProps g ⟹ f ∈ ℱ g ⟹
  |faces g| ≥ 2 ⟹ |faces(subdivFace′ g f v n ovs)| ≥ 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′ ∈ set(generatePolygon i v f g)
and f: f ∈ set (nonFinals g) and i: 2 < i and v: v ∈ 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 ∈ set (enumerator i |vertices f| )
    and containsNot: ¬ containsDuplicateEdge g f v e
    and is-eq: is = indexToVertexList f v e
    by (auto simp: generatePolygon-def)
  have f′: f ∈ 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 ≠ []
    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(fastsimp simp add:neq-Nil-conv elim:pre-subdivFace-pre-subdivFace′)
  note 2 = one-final-antimono[OF 1]
  show minGraphProps g′ ∧ one-final g′ ∧ |faces g′| ≥ 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-plane0_p
proof(clarsimp simp:invariant-def)
  fix g g′
  assume  inv: inv g and g′ ∈ set (next-plane0_p 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: split-if-asm)
  thus inv g′ using inv by(blast intro:inv-genPoly)
qed

end
```

14 Further Plane Graph Poroperties

```
theory PlaneProps
imports Invariants
begin
```

14.1 *final*

```
lemma plane-final-facesAt:
  ⟦ inv g;  final g;  f ∈ set (facesAt g v) ⟧ ⟹ final f
proof −
  assume g: inv g and fin:final g and f ∈ set (facesAt g v)
  with g have f ∈ ℱ g by(blast intro: minGraphProps inv-mgp)
  with fin 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(fastsimp 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 ⟹ f ∈ ℱ g ⟹ 3 ≤ |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
shows degree g v = tri g v + quad g v + except g v
proof −
  have dist: distinct(facesAt g v) using pl by simp
  have 3: ∀ f ∈ set(facesAt g v). |vertices f| = 3 ∨ |vertices f| = 4 ∨
                                   |vertices f| ≥ 5
  proof
    fix f assume f: f ∈ set (facesAt g v)
    hence |vertices f| ≥ 3
      using minGraphProps5[OF inv-mgp[OF pl] f] planeN4[OF pl] by blast
    thus |vertices f| = 3 ∨ |vertices f| = 4 ∨ |vertices f| ≥ 5 by arith
  qed
  have degree g v = |facesAt g v| by(simp add:degree-def)
  also have ... = card(set(facesAt g v)) by (simp add:distinct-card[OF dist])
  also have set(facesAt g v) = {f ∈ set(facesAt g v). |vertices f| = 3} ∪
                               {f ∈ set(facesAt g v). |vertices f| = 4} ∪
                               {f ∈ set(facesAt g v). |vertices f| ≥ 5}
    (is - = ?T ∪ ?Q ∪ ?E)
    using 3 by blast
  also have card(?T ∪ ?Q ∪ ?E) = card ?T + card ?Q + card ?E
    apply (subst card-Un-disjoint)
    apply simp
    apply simp
    apply fastsimp
    apply (subst card-Un-disjoint)
    apply simp
    apply simp
    apply fastsimp
    apply simp
    done
  also have ... = 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*] *cong*:*conj-cong*)
finally show *?thesis* **.**
qed

lemma *plane-fin-exceptionalVertex-def*:
assumes *pl*: *inv g* **and** *fin*: *final g*
shows *exceptionalVertex g v* =
(| [*f* ∈ *facesAt g v* . *5* ≤ |*vertices f*|] | ≠ *0*)
proof −
have ⋀*f*. *f* ∈ *set* (*facesAt g v*) ⟹ *final f*
by(*rule plane-final-facesAt*[*OF pl fin*])
then show *?thesis* **by** (*simp add*: *filter-simp exceptionalVertex-def except-def*)
qed

lemma *not-exceptional*:
inv g ⟹ *final g* ⟹ *f* ∈ *set* (*facesAt g v*) ⟹
¬ *exceptionalVertex g v* ⟹ |*vertices f*| ≤ *4*
by (*frule C0*)
(*auto simp add*: *plane-fin-exceptionalVertex-def except-def filter-empty-conv*)

14.3 Misc

lemma *in-next-plane0I*:
assumes *g*′ ∈ *set* (*generatePolygon n v f g*) *f* ∈ *set* (*nonFinals g*)
v ∈ 𝒱 *f* *3* ≤ *n n* < *4*+*p*
shows *g*′ ∈ *set* (*next-plane0*$_p$ *g*)
proof −
have ¬ *final g* **using** *prems*(*2*)
by(*auto simp*: *nonFinals-def finalGraph-def filter-empty-conv*)
thus *?thesis* **using** *prems* **by**(*auto simp*:*next-plane0-def*)
qed

lemma *next-plane0-nonfinals*: *g* [*next-plane0*$_p$]→ *g*′ ⟹ *nonFinals g* ≠ []
by(*auto simp*:*next-plane0-def finalGraph-def*)

lemma *next-plane0-ex*:
assumes *a*: *g* [*next-plane0*$_p$]→ *g*′
shows ∃*f*∈ *set*(*nonFinals g*). ∃*v* ∈ 𝒱 *f*. ∃*i* ∈ *set*([*3*..*maxGon p*]).
g′ ∈ *set* (*generatePolygon i v f g*)
proof −
from *a* **have** ¬ *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 ⟹ *g* [*next-plane0*$_p$]→ *g*′ ⟹ ¬ *final g*′ ⟹ |*faces g*′| ≠ *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*| ≠ *0*)
 prefer *2* **apply**(*drule next-plane0-nonfinals*) **apply** *simp*
apply(*subgoal-tac* |*nonFinals g*′| ≠ *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*| ≠ *0*)
 apply *arith*
apply *simp*
done

14.4 Increasing final faces

lemma *set-finals-splitFace*[*simp*]:
⟦ *f* ∈ $\mathcal{F}$ *g*; ¬ *final f* ⟧ ⟹
 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*)
apply *clarsimp*
done

lemma *next-plane0-finals-incr*:
 g [*next-plane0*$_p$]→ *g*′ ⟹ *f* ∈ *set*(*finals g*) ⟹ *f* ∈ *set*(*finals g*′)
apply(*auto simp*:*next-plane0-def generatePolygon-def split*:*split-if-asm*)
apply(*erule subdivFace-pres-finals*)
apply (*simp add*:*nonFinals-def*)
done

lemma *next-plane0-finals-subset*:
 g′ ∈ *set* (*next-plane0*$_p$ *g*) ⟹
 set (*finals g*) ⊆ *set* (*finals g*′)
 by (*auto simp add*: *next-plane0-finals-incr*)

lemma *next-plane0-final-mono*:
 ⟦ *g*′ ∈ *set* (*next-plane0*$_p$ *g*); *f* ∈ $\mathcal{F}$ *g*; *final f* ⟧ ⟹ *f* ∈ $\mathcal{F}$ *g*′
apply(*drule next-plane0-finals-subset*)
apply(*simp add*:*finals-def*)
apply *blast*
done

14.5 Increasing vertices

```
lemma next-plane0-vertices-subset:
⟦ g′ ∈ set (next-plane0_p g); minGraphProps g ⟧ ⟹ 𝒱 g ⊆ 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:
⟦ g′ ∈ set (next-plane0_p g); minGraphProps g ⟧
 ⟹ |faceListAt g| ≤ |faceListAt g′| &
     (∀ v < |faceListAt g|. |faceListAt g ! v| ≤ |faceListAt g′ ! v| )
 (concl is ?Q g g′)
apply(rule next-plane0-incr[where Q = ?Q])
prefer 4 apply assumption
prefer 4 apply assumption
  apply(rule conjI) apply fastsimp
  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:
⟦ g′ ∈ set (next-plane0_p g); minGraphProps g; v ∈ 𝒱 g ⟧
 ⟹ degree g v ≤ 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(rule conjI)
 prefer 2 apply blast
apply(frule minGraphProps4)
apply(simp add:vertices-graph)
done
```

14.7 Increasing *except*

```
lemma next-plane0-incr-except:
assumes g′ ∈ set (next-plane0_p g) inv g v ∈ 𝒱 g
shows except g v ≤ except g′ v
proof (unfold except-def)
  note inv′ = invariantE[OF inv-inv-next-plane0, OF prems(1,2)]
```

```
  note mgp = inv-mgp[OF prems(2)] and mgp' = inv-mgp[OF inv']
  note dist = distinct-filter[OF mgp-dist-facesAt[OF mgp]]
  note dist' = distinct-filter[OF mgp-dist-facesAt[OF mgp']]
  have v ∈ 𝒱 g'
    using prems(3) next-plane0-vertices-subset[OF prems(1) mgp] by blast
  have |[f∈facesAt g v . final f ∧ 5 ≤ |vertices f| ]| =
        card{f∈ set(facesAt g v) . final f ∧ 5 ≤ |vertices f|}
    (is ?L = card ?M) using distinct-card[OF dist] by simp
  also have ?M = {f∈ ℱ g. v ∈ 𝒱 f ∧ final f ∧ 5 ≤ |vertices f|}
    by(simp add: minGraphProps-facesAt-eq[OF mgp prems(3)])
  also have ... = {f ∈ set(finals g) . v ∈ 𝒱 f ∧ 5 ≤ |vertices f|}
    by(auto simp:finals-def)
  also have card ... ≤ card{f ∈ set(finals g'). v ∈ 𝒱 f ∧ 5 ≤ |vertices f|}
    (is - ≤ card ?M)
    apply(rule card-mono)
    apply simp
    using next-plane0-finals-subset[OF prems(1)] by blast
  also have ?M = {f∈ ℱ g' . v ∈ 𝒱 f ∧ final f ∧ 5 ≤ |vertices f|}
    by(auto simp:finals-def)
  also have ... = {f ∈ set(facesAt g' v) . final f ∧ 5 ≤ |vertices f|}
    by(simp add: minGraphProps-facesAt-eq[OF mgp' ‹v ∈ 𝒱 g'›])
  also have card ... =
    |[f∈facesAt g' v . final f ∧ 5 ≤ |vertices f| ]| (is - = ?R)
    using distinct-card[OF dist'] by simp
  finally show ?L ≤ ?R .
qed
```

14.8 Increasing edges

```
lemma next-plane0-set-edges-subset:
  ⟦ minGraphProps g; g [next-plane0_p]→ g' ⟧ ⟹ edges g ⊆ 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

```
lemma next-plane0-incr-finV:
 ⟦g' ∈ set (next-plane0_p 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)) (concl 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 blast
 apply(clarsimp simp:makeFaceFinal-def vertices-graph makeFaceFinalFaceList-def)
```

apply(*drule replace5*)
apply(*erule disjE*)**apply** *blast* **apply** *simp*
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*)
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*:
⟦$g' \in$ *set* (*next-plane0*$_p$ g); *inv* g; $u \in \mathcal{V}$ g; *finalVertex* g u ⟧
$\Longrightarrow$ *finalVertex* g' u
apply(*frule* (*1*) *invariantE*[*OF inv-inv-next-plane0*])
apply(*subgoal-tac* $u \in \mathcal{V}$ 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*:
⟦$g' \in$ *set* (*next-plane0*$_p$ g); *inv* g; $v \in \mathcal{V}$ g; *finalVertex* g v ⟧
$\Longrightarrow$ *set*(*facesAt* g' v) = *set*(*facesAt* g v)
apply(*frule* (*1*) *invariantE*[*OF inv-inv-next-plane0*])
apply(*subgoal-tac* $v \in \mathcal{V}$ 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 $g' \in$ *set* (*next-plane0*$_p$ g) *inv* g $v \in \mathcal{V}$ g *finalVertex* g v
shows |*filter* P (*facesAt* g' v)| = |*filter* P (*facesAt* g v)|
proof −
note *inv'* = *invariantE*[*OF inv-inv-next-plane0*, *OF prems*(*1*,*2*)]
note *mgp* = *inv-mgp*[*OF prems*(*2*)] **and** *mgp'* = *inv-mgp*[*OF inv'*]
note *dist* = *distinct-filter*[*OF mgp-dist-facesAt*[*OF mgp*]]
note *dist'* = *distinct-filter*[*OF mgp-dist-facesAt*[*OF mgp'*]]
have $v \in \mathcal{V}$ g'
using *prems*(*3*) *next-plane0-vertices-subset*[*OF prems*(*1*) *mgp*] **by** *blast*
have |*filter* P (*facesAt* g' v)| = *card*{$f \in$ *set*(*facesAt* g' v) . P f}
using *distinct-card*[*OF dist'*] **by** *simp*

```
  also have ... = card{f ∈ set(facesAt g v) . P f}
    by(simp add: next-plane0-finalVertex-facesAt-eq[OF prems])
  also have ... = |filter P (facesAt g v)|
    using distinct-card[OF dist] by simp
  finally show ?thesis .
qed
```

14.11 Properties of *subdivFace′*

```
lemma new-edge-subdivFace′:
  ⋀f v n g.
  pre-subdivFace′ g f u v n ovs ⟹ minGraphProps g ⟹ f ∈ ℱ g ⟹
  subdivFace′ g f v n ovs = makeFaceFinal f g ∨
  (∀ f′ ∈ ℱ (subdivFace′ g f v n ovs) − (ℱ g − {f}).
   ∃ e ∈ ℰ f′. e ∉ 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 ∈ 𝒱 f and vf: v ∈ 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-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 uvw: before (verticesFrom f u) v w
      using pre by(simp add:pre-subdivFace′-def)
    have uw: u ≠ w using pre by(clarsimp simp: pre-subdivFace′-def)
    { assume w: f · v = w and n: n = 0
      have pre′: pre-subdivFace′ g f u w 0 ovs
        using pre Some n by (simp (depth-limit:5) 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  ?f_1 = fst ?fdg and ?f_2 = fst(snd ?fdg) and ?g′ = snd(snd ?fdg)
      let ?g″ = subdivFace′ ?g′ ?f_2 w 0 ovs
      let ?fvw = between(vertices f) v w and ?fwv = 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 # ?fvw @ [w])
  using f₁ by(simp add:edges-split-face1[OF pre-split])
have E₂: ℰ ?f₂ = Edges (v # ?vs @ [w]) ∪ Edges (w # ?fwv @ [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: ∃ e ∈ ℰ ?f₂. e ∉ ℰ g
proof cases
  assume ?vs = []
  hence (v,w) ∈ ℰ ?f₂ ∧ (v,w) ∉ ℰ g using pre-fdg E₂
    by(unfold pre-splitFace-def) (auto simp:Edges-Cons)
  thus ?thesis by blast
next
  assume ?vs ≠ []
```

```
  then obtain x xs where vs: ?vs = x#xs
    by(auto simp only:neq-Nil-conv)
  hence (v,x) ∈ ℰ ?f2 using E2 by (auto simp:Edges-Cons)
  moreover have (v,x) ∉ ℰ g
  proof −
    have x ∈ set ?vs using vs 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 fdg: (?f1,?f2,?g') = splitFace g v w f ?vs by auto
hence Fg': ℱ ?g' = {?f1,?f2} ∪ (ℱ g − {f})
  using set-faces-splitFace[OF mgp fg pre-fdg] by blast
have ∀f' ∈ ℱ ?g'' − (ℱ g − {f}). ∃e ∈ ℰ f'. e ∉ ℰ g
proof (clarify)
  fix f' assume f'g'': f' ∈ ℱ ?g'' and f'ng: f' ∉ ℱ g − {f}
  from IH[OF pre' mgp' f2inF]
  show ∃e ∈ ℰ f'. e ∉ ℰ g
  proof
    assume ?g'' = makeFaceFinal ?f2 ?g'
    hence f' = setFinal ?f2 ∨ f' = ?f1 (is ?A ∨ ?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: ∀f' ∈ ℱ ?g'' − (ℱ ?g' − {?f2}).
               ∃e ∈ ℰ f'. e ∉ ℰ ?g'
    show ?thesis
    proof cases
      assume f' ∈ {?f1,?f2}
      thus ?thesis using 1 2 by blast
    next
      assume f' ∉ {?f1,?f2}
      hence ∃e∈ℰ f'. e ∉ ℰ ?g'
        using A f'g'' f'ng Fg' by simp
      with splitFace-edges-incr[OF pre-fdg fdg]
      show ?thesis by blast
    qed
  qed
```

```
      qed
    }
    ultimately show ?thesis using Some by(auto simp: split-def)
  qed
qed


lemma dist-edges-subdivFace':
  pre-subdivFace' g f u v n ovs ⟹ minGraphProps g ⟹ f ∈ ℱ g ⟹
  subdivFace' g f v n ovs = makeFaceFinal f g ∨
  (∀ f' ∈ ℱ (subdivFace' g f v n ovs) − (ℱ g − {f}). ℰ f' ≠ 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: ⟦ distinct(vertices f); u ∈ 𝒱 f ⟧ ⟹
   between (vertices f) u (last (verticesFrom f u)) =
   butlast(tl(verticesFrom f u))
apply(drule split-list)
apply (fastsimp dest: last-in-set
  simp: between-def verticesFrom-Def split-def
       last-append butlast-append fst-splitAt-last)
done


lemma final-subdivFace': ⋀f u n g. minGraphProps g ⟹
  pre-subdivFace' g f r u n ovs ⟹ f ∈ ℱ g ⟹
  (ovs = [] ⟶ n=0 ∧ u = last(verticesFrom f r)) ⟹
  ∃ f' ∈ set(finals(subdivFace' g f u n ovs)) − set(finals g).
   (f⁻¹ · r,r) ∈ ℰ f' ∧  |vertices f'| =
     n + |ovs| + (if r=u then 1 else |between (vertices f) r u| + 2)
proof (induct ovs)
  case Nil show ?case (is ∃ f' ∈ ?F. ?P f')
  proof
    show ?P (setFinal f) (is ?A ∧ ?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) arith
    qed
  next
    show setFinal f ∈ ?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 ∈ 𝒱 f and u ∈ 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 ≠ [] 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 ≠ v
        using pre by(simp add:pre-subdivFace'-def)+
    show ?thesis
    proof (cases f · u = v ∧ n = 0)
      case True
      have pre': pre-subdivFace' g f r v 0 ovs
        using pre True by (simp (depth-limit:5) add: pre-subdivFace'-Some2)
      have mt: ovs = [] ⟶ 0 = 0 ∧ v = last (verticesFrom f r)
        using pre by(clarsimp simp:pre-subdivFace'-def)
      show ?thesis using Some True IH[OF mgp pre' fg mt] ‹r ≠ v›
        by(auto simp: between-next-empty[OF distf]
          unroll-between-next2[OF distf ‹r ∈ 𝒱 f› ‹u ∈ 𝒱 f›])
    next
      case False
      let ?vs = [countVertices g..<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 · u = v ⟶ n ≠ 0 using False by auto
      have VfVg: 𝒱 f ⊆ 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 ≠ v and v ∈ 𝒱 f and disj: 𝒱 f ∩ set ?vs = {}
        by(unfold pre-splitFace-def)simp+
      hence vvs: v ∉ set ?vs by auto
      have vf2: vertices ?f2 = [v] @ ?fvu @ u # ?vs
```

```
  by(simp add:split-face-def splitFace-def split-def)
hence betuvf2: between (vertices ?f2) u v = ?vs
  using splitFace-distinct1[OF pre-fdg]
  by(simp add: between-back)
have betrvf2: r ≠ u ⟹ between (vertices ?f2) r v =
  between (vertices f) r u @ [u] @ ?vs
proof -
  assume r≠u
  have r: r ∈ set (between (vertices f) v u)
    using ‹r≠u› ‹r≠v› ‹u≠v› ‹v ∈ 𝒱 f› ‹r ∈ 𝒱 f› distf ruv
    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 ∉ set (between (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 ∈ 𝒱 f and disj: 𝒱 f ∩ set ?vs = {}
  by(unfold pre-splitFace-def, simp)+
have fr: ?f2^-1 · r = f^-1 · r
proof -
  note pre-split = pre-splitFace-pre-split-face[OF pre-fdg]
  have rinf2: r ∈ 𝒱 ?f2
  proof cases
    assume r = u thus ?thesis by(simp add:vf2)
  next
    assume r ≠ u
    hence r ∈ set ?fvu using distf ‹v : 𝒱 f› ‹r≠v› ‹r : 𝒱 f› ruv
      by(blast intro: before-between rotate-before-vFrom)
    thus ?thesis by(simp add:vf2)
  qed
  have E2: ℰ ?f2 = Edges (u # ?vs @ [v]) ∪
                   Edges (v # ?fvu @ [u])
    by(simp add:splitFace-def split-def
      edges-split-face2[OF pre-split])
  moreover have (?f2^-1 · r, r) ∈ ℰ ?f2
    by(blast intro: prevVertex-in-edges rinf2
      splitFace-distinct1[OF pre-fdg])
  moreover have (?f2^-1 · r, r) ∉ Edges (u # ?vs @ [v])
  proof -
    have r ∉ set ?vs using ‹r : 𝒱 f› disj by blast
    thus ?thesis using ‹r ≠ v›
      by(simp add:Edges-Cons Edges-append notinset-notinEdge2) arith
```

```
        qed
        ultimately have (?f₂⁻¹ · r, r) ∈ Edges (v # ?fvu @ [u]) by blast
        hence (?f₂⁻¹ · r, r) ∈ ℰ f using pre-split-face-symI[OF pre-split]
          by(blast intro: Edges-between-edges)
        hence eq: f · (?f₂⁻¹ · r) = r and inf: ?f₂⁻¹ · r ∈ 𝒱 f
          by(simp add:edges-face-eq)+
        have ?f₂⁻¹ · r = f⁻¹ · (f · (?f₂⁻¹ · r))
          using prevVertex-nextVertex[OF distf inf] by simp
        also have ... = f⁻¹ · r using eq by simp
        finally show ?thesis .
      qed
      hence mt: ovs = [] ⟶ 0 = 0 ∧ v = last (verticesFrom ?f₂ r)
        using pre' pre by(auto simp:pre-subdivFace'-def splitFace-def
          split-def last-vFrom)
      from IH[OF mgp' pre' f2g mt] ‹r ≠ v› obtain f' where
        f: f' ∈ set(finals(subdivFace' ?g' ?f₂ v 0 ovs)) − set(finals ?g')
        and ff: (?f₂⁻¹ · r, r) ∈ ℰ f'
        |vertices f'| = |ovs| + |between (vertices ?f₂) r v| + 2
        by simp blast
      show ?thesis (is ∃f' ∈ ?F. ?P f')
      proof
        show f' ∈ ?F using f pre Some fg
          by(simp add:False split-def pre-subdivFace'-def)
        show ?P f' using ff fr by(clarsimp simp:betuvf₂ betrvf₂)
      qed
    qed
  qed
qed

lemma subdivFace'-final-base: ⋀f u n g. minGraphProps g ⟹
  pre-subdivFace' g f r u n ovs ⟹ f ∈ ℱ g ⟹
  ∃f' ∈ ℱ (subdivFace' g f u n ovs). final f' ∧ (f⁻¹ · r,r) ∈ ℰ f'
proof (induct ovs)
  case Nil show ?case (is ∃f' ∈ ?F. ?P f')
  proof
    show ?P (setFinal f) using Nil
      by(simp add:pre-subdivFace'-def prevVertex-in-edges
        del:is-nextElem-edges-eq)
  next
    show setFinal f ∈ ?F using Nil
      by (simp add: makeFaceFinal-def makeFaceFinalFaceList-def replace3)
  qed
next
  case (Cons ov ovs)
  show ?case
  proof (cases ov)
    case None thus ?thesis using Cons by(fastsimp simp:pre-subdivFace'-None)
  next
    case (Some v)
```

```
show ?thesis
proof (cases f • u = v ∧ n = 0)
  case True with Cons Some show ?thesis
    by(fastsimp intro:pre-subdivFace'-Some2)
next
  case False
   note IH = Cons(1) and mgp = Cons(2) and pre = Cons(3) and fg = Cons(4)
  note pre = pre[simplified Some]
  have ruv: before (verticesFrom f r) u v and v ≠ r
    and distf: distinct (vertices f) and r ∈ 𝒱 f
    using pre by(simp add:pre-subdivFace'-def)+
  let ?vs = [countVertices g..<countVertices g + n]
  let ?fdg = splitFace g u v f ?vs
  let ?g' = snd(snd ?fdg) and ?f₂ = fst(snd ?fdg)
  have False': f • u = v ⟶ n ≠ 0 using False by auto
  have VfVg: 𝒱 f ⊆ 𝒱 g using mgp fg
      by (simp add: minGraphProps-def faces-subset-def)
  note pre-fdg = pre-subdivFace'-preFaceDiv[OF pre fg False' VfVg]
  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 ∈ 𝒱 f and disj: 𝒱 f ∩ set ?vs = {}
by(unfold pre-splitFace-def, simp)+
  from IH[OF mgp' pre' f2g] obtain f' where
    f' ∈ ℱ (subdivFace' ?g' ?f₂ v 0 ovs) and
    final f' and (?f₂⁻¹ • r, r) ∈ ℰ f'
    by blast
  moreover have ?f₂⁻¹ • r = f⁻¹ • r
  proof -
    let ?fvu = between (vertices f) v u
    note pre-split = pre-splitFace-pre-split-face[OF pre-fdg]
    have vf₂: vertices ?f₂ = [v] @ ?fvu @ u # ?vs
      by(simp add:split-face-def splitFace-def split-def)
    have rinf₂: r ∈ 𝒱 ?f₂
    proof cases
      assume r = u thus ?thesis by(simp add:vf₂)
    next
      assume r ≠ u
      hence r ∈ set ?fvu using distf ‹v : 𝒱 f› ‹v≠r› ‹r : 𝒱 f› ruv
        by(blast intro: before-between rotate-before-vFrom)
      thus ?thesis by(simp add:vf₂)
    qed
    have E₂: ℰ ?f₂ = Edges (u # ?vs @ [v]) ∪
                      Edges (v # ?fvu @ [u])
      by(simp add:splitFace-def split-def
        edges-split-face2[OF pre-split])
    moreover have (?f₂⁻¹ • r, r) ∈ ℰ ?f₂
      by(blast intro: prevVertex-in-edges rinf₂
```

```
      splitFace-distinct1[OF pre-fdg])
    moreover have (?f2^-1 · r, r) ∉ Edges (u # ?vs @ [v])
    proof -
     have r ∉ set ?vs using ‹r : 𝒱 f› disj by blast
     thus ?thesis using ‹v ≠ r›
      by(simp add:Edges-Cons Edges-append notinset-notinEdge2) arith
    qed
    ultimately have (?f2^-1 · r, r) ∈ Edges (v # ?fvu @ [u]) by blast
    hence (?f2^-1 · r, r) ∈ ℰ f using pre-split-face-symI[OF pre-split]
     by(blast intro: Edges-between-edges)
    hence eq: f · (?f2^-1 · r) = r and inf: ?f2^-1 · r ∈ 𝒱 f
     by(simp add:edges-face-eq)+
    have ?f2^-1 · r = f^-1 · (f · (?f2^-1 · r))
     using prevVertex-nextVertex[OF distf inf] by simp
    also have ... = f^-1 · r using eq by simp
    finally show ?thesis .
   qed
   ultimately show ?thesis using Cons Some
    by(simp add:False split-def) blast
  qed
 qed
qed


lemma Seed-max-final-ex:
 ∃f∈set (finals (Seed p)). |vertices f| = maxGon p
 by (simp add: Seed-def graph-max-final-ex)

lemma max-face-ex: assumes a: Seed_p [next-plane0_p]→* g
shows ∃f ∈ set (finals g). |vertices f| = 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∈set (finals g) and |vertices f| = maxGon p
  by auto
 moreover have f∈set (finals g') by (rule next-plane0-finals-incr)
 ultimately show ?case by auto
qed


end
```

15 Summation Over Lists

theory *ListSum*

imports *ListAux*
begin

consts *ListSum* :: *'b list* ⇒ (*'b* ⇒ *'a*::*comm-monoid-add*) ⇒ *'a*::*comm-monoid-add*

primrec
ListSum [] *f* = *0*
ListSum (*l*#*ls*) *f* = *f l* + *ListSum ls f*

syntax *-ListSum* :: *idt* ⇒ *'b list* ⇒ (*'a*::*comm-monoid-add*) ⇒
(*'a*::*comm-monoid-add*) $(\sum_{-\in-}\ -)$
translations $\sum_{x\in xs} f$ == *ListSum xs* (λ*x*. *f*)

consts *natListSum* :: *'b list* ⇒ (*'b* ⇒ *nat*) ⇒ *nat*
primrec
natListSum [] *f* = *0*
natListSum (*l*#*ls*) *f* = *f l* + *natListSum ls f*

consts *intListSum* :: *'b list* ⇒ (*'b* ⇒ *int*) ⇒ *int*
primrec
intListSum [] *f* = *0*
intListSum (*l*#*ls*) *f* = *f l* + *intListSum ls f*

lemma [*THEN eq-reflection*, *code unfold*]: ((*ListSum ls f*)::*nat*) = *natListSum ls f*
by (*induct ls*) *simp-all*

lemma [*THEN eq-reflection*, *code unfold*]: ((*ListSum ls f*)::*int*) = *intListSum ls f*
by (*induct ls*) *simp-all*

lemma [*simp*]: $\sum_{v\ \in\ V} 0 = (0::nat)$ **by** (*induct V*) *simp-all*

lemma *ListSum-compl1*:
$(\sum_{x\ \in\ [x\in xs.\ \neg\ P\ x]} f\ x) + \sum_{x\ \in\ [x\in xs.\ P\ x]} f\ x = \sum_{x\ \in\ xs} (f\ x::nat)$
by (*induct xs*) *simp-all*

lemma *ListSum-compl2*:
$(\sum_{x\ \in\ [x\in xs.\ P\ x]} f\ x) + \sum_{x\ \in\ [x\in xs.\ \neg\ P\ x]} f\ x = \sum_{x\ \in\ xs} (f\ x::nat)$
by (*induct xs*) *simp-all*

lemmas *ListSum-compl* = *ListSum-compl1 ListSum-compl2*

lemma *ListSum-conv-setsum*:
distinct xs ⟹ *ListSum xs f* = *setsum f* (*set xs*)
by(*induct xs*) *simp-all*

lemma *listsum-cong*:
⟦ *xs* = *ys*; ⋀*y*. *y* ∈ *set ys* ==> *f y* = *g y* ⟧
⟹ *ListSum xs f* = *ListSum ys g*
apply *simp*
apply(*erule thin-rl*)
by (*induct ys*) *simp-all*

lemma *strong-listsum-cong*[*cong*]:
⟦ *xs* = *ys*; ⋀*y*. *y* ∈ *set ys* =*simp*=> *f y* = *g y* ⟧
⟹ *ListSum xs f* = *ListSum ys g*
by(*auto simp*:*simp-implies-def intro*!:*listsum-cong*)

lemma *ListSum-eq* [*trans*]:
(⋀*v*. *v* ∈ *set V* ⟹ *f v* = *g v*) ⟹ $\sum_{v \in V} f\ v = \sum_{v \in V} g\ v$
by(*auto intro*!:*listsum-cong*)

lemma *ListSum-set-eq*:
⋀*C*. *distinct B* ⟹ *distinct C* ⟹ *set B* = *set C* ⟹
$\sum_{a \in B} f\ a = \sum_{a \in C} (f\ a{::}nat)$
by (*simp add*: *ListSum-conv-setsum*)

lemma *ListSum-disj-union*:
distinct A ⟹ *distinct B* ⟹ *distinct C* ⟹
set C = *set A* ∪ *set B* ⟹
set A ∩ *set B* = {} ⟹
$\sum_{a \in C} (f\ a) = (\sum_{a \in A} f\ a) + (\sum_{a \in B} (f\ a{::}nat))$
by (*simp add*: *ListSum-conv-setsum setsum-Un-disjoint*)

constdefs *separating* :: *'a set* ⇒ (*'a* ⇒ *'b set*) ⇒ *bool*
separating V F ≡
(∀ *v1* ∈ *V*. ∀ *v2* ∈ *V*. *v1* ≠ *v2* ⟶ *F v1* ∩ *F v2* = {})

lemma *separating-insert1*:
separating (*insert a V*) *F* ⟹ *separating V F*
by (*simp add*: *separating-def*)

lemma *separating-insert2*:
separating (*insert a V*) *F* ⟹ *a* ∉ *V* ⟹ *v* ∈ *V* ⟹

```
  F a ∩ F v = {}
  by (auto simp add: separating-def)

lemma setsum-disj-Union:
 finite V ⟹
  (⋀f. finite (F f)) ⟹
  separating V F ⟹
  (∑v∈V. ∑f∈(F v). (w f::nat)) = (∑f∈(⋃v∈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: (∑v∈V. ∑f∈(F v). w f) = (∑f∈(⋃v∈V. F v). w f)
    by simp

  moreover have prems: finite V a ∉ V ⋀f. finite (F f) .

  moreover from s have ⋀v. a ∉ V ⟹ v ∈ V ⟹ F a ∩ F v = {}
   by (simp add: separating-insert2)
  with prems have (F a) ∩ (⋃v∈V. F v) = {} by auto

  ultimately show ?case by (simp add: setsum-Un-disjoint)
qed


lemma listsum-const[simp]:
  ∑x ∈ xs k = length xs * k
by (induct xs) (simp-all add: ring-distrib)

lemma ListSum-add:
  (∑x ∈ V f x) + ∑x ∈ V g x = ∑x ∈ V (f x + (g x::nat))
  by (induct V) auto

lemma ListSum-le:
  (⋀v. v ∈ set V ⟹ f v ≤ g v) ⟹ ∑v ∈ V f v ≤ ∑v ∈ V (g v::nat)
proof (induct V)
  case Nil then show ?case by simp
next
  case (Cons v V) then have ∑v ∈ V f v ≤ ∑v ∈ V g v by simp
  moreover from Cons have f v ≤ g v by simp
  ultimately show ?case by simp arith
qed

lemma ListSum1-bound:
 a ∈ set F ⟹ (d a::nat) ≤ ∑f ∈ F d f
by (induct F) auto
```

```
lemma ListSum2-bound:
  a ∈ set F ⟹ b ∈ set F ⟹ a ≠ b ⟹ d a + d b ≤ ∑f ∈ F (d f::nat)
proof (induct F)
  case Nil then show ?case by simp
next
  case (Cons f F)
  then have a = f ∨ a ∈ set F  b = f ∨ b ∈ set F by simp-all
  then show ?case
  proof (elim disjE)
    assume a = f  b = f with Cons have False by simp
    then show ?thesis by simp
  next
    assume a = f  b ∈ set F
    with Cons show ?thesis by (simp add: ListSum1-bound)
  next
    assume a ∈ set F b = f
    with Cons show ?thesis by (simp add: ListSum1-bound)
  next
    assume a ∈ set F b ∈ set F
    with Cons have d a + d b ≤ ∑f ∈ F d f by simp
    then show ?thesis by simp arith
  qed
qed

end
```

16 Tameness

```
theory Tame
imports Graph ListSum
begin
```

16.1 Constants

```
constdefs squanderTarget :: nat
 squanderTarget ≡ 14800

constdefs excessTCount :: nat ⇒ nat
 a t ≡ if t < 3 then squanderTarget
     else if t = 3 then 1400
     else if t = 4 then 1500
     else 0

constdefs squanderVertex :: nat ⇒ nat ⇒ nat
 b p q ≡ if p = 0 ∧ q = 3 then 7135
     else if p = 0 ∧ q = 4 then 10649
     else if p = 1 ∧ q = 2 then  6950
```

else if $p = 1 \wedge q = 3$ *then* 7135
else if $p = 2 \wedge q = 1$ *then* 8500
else if $p = 2 \wedge q = 2$ *then* 4756
else if $p = 2 \wedge q = 3$ *then* 12981
else if $p = 3 \wedge q = 1$ *then* 3642
else if $p = 3 \wedge q = 2$ *then* 8334
else if $p = 4 \wedge q = 0$ *then* 4139
else if $p = 4 \wedge q = 1$ *then* 3781
else if $p = 5 \wedge q = 0$ *then* 550
else if $p = 5 \wedge q = 1$ *then* 11220
else if $p = 6 \wedge q = 0$ *then* 6339
else squanderTarget

constdefs *scoreFace* :: *nat* ⇒ *int*
c $n \equiv$ *if* $n = 3$ *then* 1000
else if $n = 4$ *then* 0
else if $n = 5$ *then* -1030
else if $n = 6$ *then* -2060
else if $n = 7$ *then* -3030
else if $n = 8$ *then* -3030
else -3030

constdefs *squanderFace* :: *nat* ⇒ *nat*
d $n \equiv$ *if* $n = 3$ *then* 0
else if $n = 4$ *then* 2378
else if $n = 5$ *then* 4896
else if $n = 6$ *then* 7414
else if $n = 7$ *then* 9932
else if $n = 8$ *then* 10916
else squanderTarget

16.2 Separated sets of vertices

A set of vertices V is *separated*, iff the following conditions hold:

1. For each vertex in V there is an exceptional face containing it:

constdefs *separated*$_1$ :: *graph* ⇒ *vertex set* ⇒ *bool*
separated$_1$ $g\ V \equiv \forall v \in V.\ \mathit{except}\ g\ v \neq 0$

2. No two vertices in V are adjacent:

constdefs *separated*$_2$ :: *graph* ⇒ *vertex set* ⇒ *bool*
separated$_2$ $g\ V \equiv \forall v \in V.\ \forall f \in \mathit{set}\ (\mathit{facesAt}\ g\ v).\ f \bullet v \notin V$

3. No two vertices lie on a common quadrilateral:

constdefs *separated*$_3$ :: *graph* ⇒ *vertex set* ⇒ *bool*
separated$_3$ $g\ V \equiv$
$\forall v \in V.\ \forall f \in \mathit{set}\ (\mathit{facesAt}\ g\ v).\ |\mathit{vertices}\ f| \leq 4 \longrightarrow \mathcal{V}\ f \cap V = \{v\}$

A set of vertices is called *preseparated*, iff no two vertices are adjacent or lie on a common quadrilateral:

constdefs $\mathit{preSeparated} :: \mathit{graph} \Rightarrow \mathit{vertex\ set} \Rightarrow \mathit{bool}$
$\mathit{preSeparated}\ g\ V \equiv \mathit{separated}_2\ g\ V \wedge \mathit{separated}_3\ g\ V$

4. Every vertex in V has degree 5:

constdefs $\mathit{separated}_4 :: \mathit{graph} \Rightarrow \mathit{vertex\ set} \Rightarrow \mathit{bool}$
$\mathit{separated}_4\ g\ V \equiv \forall v \in V.\ \mathit{degree}\ g\ v = 5$

constdefs $\mathit{separated} :: \mathit{graph} \Rightarrow \mathit{vertex\ set} \Rightarrow \mathit{bool}$
$\mathit{separated}\ g\ V \equiv$
$\quad \mathit{separated}_1\ g\ V \wedge \mathit{separated}_2\ g\ V \wedge \mathit{separated}_3\ g\ V \wedge \mathit{separated}_4\ g\ V$

16.3 Admissible weight assignments

A weight assignment $w :: \mathit{face} \Rightarrow \mathit{nat}$ assigns a natural number to every face.

We formalize the admissibility requirements as follows:

1. $\mathrm{d}(|f|) \leq w(f)$:

constdefs $\mathit{admissible}_1 :: (\mathit{face} \Rightarrow \mathit{nat}) \Rightarrow \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{admissible}_1\ w\ g \equiv \forall f \in \mathcal{F}\ g.\ \mathrm{d}\ |\mathit{vertices}\ f| \leq w\ f$

2. If v has type (p, q), then $\mathrm{b}(p, q) \leq \sum\limits_{v \in f} w(f)$:

constdefs $\mathit{admissible}_2 :: (\mathit{face} \Rightarrow \mathit{nat}) \Rightarrow \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{admissible}_2\ w\ g \equiv$
$\forall v \in \mathcal{V}\ g.\ \mathit{except}\ g\ v = 0 \longrightarrow \mathrm{b}\ (\mathit{tri}\ g\ v)\ (\mathit{quad}\ g\ v) \leq \sum_{f \in \mathit{facesAt}\ g\ v} w\ f$

4. Let V be any separated set of vertices.
Then $\sum\limits_{v \in V} \mathrm{a}(\mathit{tri}(v)) \leq \sum\limits_{V \cap f \neq \emptyset} (w(f) - d(|f|))$:

constdefs $\mathit{admissible}_3 :: (\mathit{face} \Rightarrow \mathit{nat}) \Rightarrow \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{admissible}_3\ w\ g \equiv$
$\quad \forall V.\ \mathit{separated}\ g\ (\mathit{set}\ V) \wedge \mathit{set}\ V \subseteq \mathcal{V}\ g \longrightarrow$
$\quad\quad (\sum_{v \in V} \mathrm{a}\ (\mathit{tri}\ g\ v))$
$\quad + (\sum_{f \in [f \in \mathit{faces}\ g.\ \exists v \in \mathit{set}\ V.\ f \in \mathit{set}\ (\mathit{facesAt}\ g\ v)]} \mathrm{d}\ |\mathit{vertices}\ f|\ )$
$\quad \leq \sum_{f \in [f \in \mathit{faces}\ g.\ \exists v \in \mathit{set}\ V.\ f \in \mathit{set}\ (\mathit{facesAt}\ g\ v)]} w\ f$

Finally we define admissibility of weights functions.

constdefs $\mathit{admissible} :: (\mathit{face} \Rightarrow \mathit{nat}) \Rightarrow \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{admissible}\ w\ g \equiv \mathit{admissible}_1\ w\ g \wedge \mathit{admissible}_2\ w\ g \wedge \mathit{admissible}_3\ w\ g$

16.4 Tameness

1. The length of each face is (at least 3 and) at most 8:

constdefs $\mathit{tame}_1 :: \mathit{graph} \Rightarrow \mathit{bool}$

$\mathit{tame}_1\ g \equiv \forall f \in \mathcal{F}\ g.\ 3 \leq |\mathit{vertices}\ f| \wedge |\mathit{vertices}\ f| \leq 8$

2. Every 3-cycle is a face or the opposite of a face:

A face given by a vertex list *vs* is contained in a graph g, if it is isomorphic to one of the faces in g. The notation $f \in_{\cong} F$ means $\exists f' \in F.\ f \cong f'$, where $\cong$ is the equivalence relation on faces (see Chapter **??**).

The notions *is-path* and *is-cycle* are defined locally (rather than in the theory of graphs) because no lemmas need to be proved about them because the generation of tame graphs uses these same executable functions as the definition of tameness. Hence there is nothing to prove.

consts $\mathit{is\text{-}path} :: \mathit{graph} \Rightarrow \mathit{vertex\ list} \Rightarrow \mathit{bool}$
primrec
$\mathit{is\text{-}path}\ g\ [] = \mathit{True}$
$\mathit{is\text{-}path}\ g\ (u\#vs) = (\mathit{case}\ vs\ \mathit{of}\ [] \Rightarrow \mathit{True}$
$\qquad\qquad\qquad\qquad | \ v\#ws \Rightarrow v \in \mathit{set}(\mathit{neighbors}\ g\ u) \wedge \mathit{is\text{-}path}\ g\ vs)$

constdefs
$\mathit{is\text{-}cycle} :: \mathit{graph} \Rightarrow \mathit{vertex\ list} \Rightarrow \mathit{bool}$
$\mathit{is\text{-}cycle}\ g\ vs \equiv \mathit{hd}\ vs \in \mathit{set}(\mathit{neighbors}\ g\ (\mathit{last}\ vs)) \wedge \mathit{is\text{-}path}\ g\ vs$

constdefs $\mathit{tame}_2 :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame}_2\ g \equiv$
$\quad \forall\, a\ b\ c.\ \mathit{is\text{-}cycle}\ g\ [a,b,c] \wedge \mathit{distinct}\ [a,b,c] \longrightarrow$
$\quad (\exists f \in \mathcal{F}\ g.\ \mathit{vertices}\ f \cong [a,b,c] \vee \mathit{vertices}\ f \cong [c,b,a])$

3. Every 4-cycle surrounds one of the following configurations:

constdefs $\mathit{tameConf}_1 :: \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex\ list\ list}$
$\mathit{tameConf}_1\ a\ b\ c\ d \equiv [[a,b,c,d]]$

constdefs $\mathit{tameConf}_2 :: \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex\ list\ list}$
$\mathit{tameConf}_2\ a\ b\ c\ d \equiv [[a,b,c],\ [a,c,d]]$

constdefs $\mathit{tameConf}_3 :: \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex\ list\ list}$
$\mathit{tameConf}_3\ a\ b\ c\ d\ e \equiv [[a,b,e],\ [b,c,e],\ [a,e,c,d]]$

constdefs $\mathit{tameConf}_4 :: \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex\ list\ list}$
$\mathit{tameConf}_4\ a\ b\ c\ d\ e \equiv [[a,b,e],\ [b,c,e],\ [c,d,e],\ [d,a,e]]$

Given a fixed 4-cycle and using the convention of drawing faces clockwise, a tame configuration can occur in the 'interior' or on the outside of the 4-cycle. For configuration $\mathit{tameConf}_2$ there are two possible rotations of the triangles, for configuration $\mathit{tameConf}_3$ there are 4. The notation $F_1 \subseteq_{\cong} F_2$ means $\forall f \in F_1.\ f \in_{\cong} F_2$.

Note that our definition only ensures the existence of certain faces in the graph, not the fact that no other faces of the graph may lie in the interior or on the outside. Hence it is slightly weaker than the definition in Hales' paper.

constdefs $\mathit{tame\text{-}quad} :: \mathit{graph} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{bool}$
$\mathit{tame\text{-}quad}\ g\ a\ b\ c\ d \equiv$
$\quad \mathit{set}(\mathit{tameConf}_1\ a\ b\ c\ d) \subseteq_{\cong} \mathit{vertices}\ \text{‘}\ \mathcal{F}\ g$
$\quad \lor\ \mathit{set}(\mathit{tameConf}_2\ a\ b\ c\ d) \subseteq_{\cong} \mathit{vertices}\ \text{‘}\ \mathcal{F}\ g$
$\quad \lor\ \mathit{set}(\mathit{tameConf}_2\ b\ c\ d\ a) \subseteq_{\cong} \mathit{vertices}\ \text{‘}\ \mathcal{F}\ g$
$\quad \lor\ (\exists\, e \in \mathcal{V}\ g - \{a,b,c,d\}.$
$\qquad \mathit{set}(\mathit{tameConf}_3\ a\ b\ c\ d\ e) \subseteq_{\cong} \mathit{vertices}\ \text{‘}\ \mathcal{F}\ g$
$\qquad \lor\ \mathit{set}(\mathit{tameConf}_3\ b\ c\ d\ a\ e) \subseteq_{\cong} \mathit{vertices}\ \text{‘}\ \mathcal{F}\ g$
$\qquad \lor\ \mathit{set}(\mathit{tameConf}_3\ c\ d\ a\ b\ e) \subseteq_{\cong} \mathit{vertices}\ \text{‘}\ \mathcal{F}\ g$
$\qquad \lor\ \mathit{set}(\mathit{tameConf}_3\ d\ a\ b\ c\ e) \subseteq_{\cong} \mathit{vertices}\ \text{‘}\ \mathcal{F}\ g$
$\qquad \lor\ \mathit{set}(\mathit{tameConf}_4\ a\ b\ c\ d\ e) \subseteq_{\cong} \mathit{vertices}\ \text{‘}\ \mathcal{F}\ g)$

constdefs $\mathit{tame}_3 :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame}_3\ g \equiv \forall\, a\ b\ c\ d.\ \mathit{is\text{-}cycle}\ g\ [a,b,c,d] \land \mathit{distinct}[a,b,c,d] \longrightarrow$
$\quad \mathit{tame\text{-}quad}\ g\ a\ b\ c\ d \lor \mathit{tame\text{-}quad}\ g\ d\ c\ b\ a$

4. The degree of every vertex is at least 2 and at most 6:

constdefs $\mathit{tame}_{45} :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame}_{45}\ g \equiv \forall\, v \in \mathcal{V}\ g.\ \mathit{degree}\ g\ v \le (\mathit{if}\ \mathit{except}\ g\ v = 0\ \mathit{then}\ 6\ \mathit{else}\ 5)$

6. The following inequality holds:

constdefs $\mathit{tame}_6 :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame}_6\ g \equiv 8000 \le \sum_{f \in \mathit{faces}\ g} \mathrm{c}\ |\mathit{vertices}\ f|$

This property implies that there are at least 8 triangles in a tame graph.

7. There exists an admissible weight assignment of total weight less than the target:

constdefs $\mathit{tame}_7 :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame}_7\ g \equiv \exists\, w.\ \mathit{admissible}\ w\ g \land \sum_{f \in \mathit{faces}\ g} w\ f < \mathit{squanderTarget}$

8. We formalize the additional restriction (compared with the original definition) that tame graphs do not contain two adjacent vertices of type $(4,0)$:

constdefs $\mathit{type40} :: \mathit{graph} \Rightarrow \mathit{vertex} \Rightarrow \mathit{bool}$
$\mathit{type40}\ g\ v \equiv \mathit{tri}\ g\ v = 4 \land \mathit{quad}\ g\ v = 0 \land \mathit{except}\ g\ v = 0$

constdefs $\mathit{tame}_8 :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame}_8\ g \equiv \neg(\exists\, v \in \mathcal{V}\ g.\ \mathit{type40}\ g\ v \land (\exists\, w \in \mathit{set}(\mathit{neighbors}\ g\ v).\ \mathit{type40}\ g\ w))$

Finally we define the notion of tameness.

constdefs $\mathit{tame} :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame}\ g \equiv$
$\quad \mathit{tame}_1\ g \land \mathit{tame}_2\ g \land \mathit{tame}_3\ g \land \mathit{tame}_{45}\ g\ (*\land\ \mathit{tame}_5\ g*) \land \mathit{tame}_6\ g \land \mathit{tame}_7\ g$

$\wedge$ *tame*$_8$ *g*

theory *Plane1Props*
imports *Plane1 PlaneProps Tame*
begin

lemma *next-plane-subset*:
$\forall f \in \mathcal{F}$ *g*. *vertices f* $\neq [] \Longrightarrow$
set (*next-plane*$_p$ *g*) $\subseteq$ *set* (*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* $\forall f \in$*set* (*nonFinals g*). *vertices f* $\neq$ [])
apply(*simp add*:*minimalVertex-def*)
apply(*simp add*:*nonFinals-def*)
apply *simp*
done

lemma *mgp-next-plane0-if-next-plane*:
minGraphProps g $\Longrightarrow$ *g* [*next-plane*$_p$]$\rightarrow$ *g'* $\Longrightarrow$ *g* [*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 *Vector Plane1 Tame*
begin

constdefs
faceSquanderLowerBound :: *graph* $\Rightarrow$ *nat*
faceSquanderLowerBound g $\equiv \sum_{f \,\in\, \mathit{finals}\ g}$ d |*vertices f*|

constdefs
d3-const :: *nat*
d3-const == d *3*
d4-const :: *nat*
d4-const == d *4*

excessAtType :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
excessAtType t q e ≡
  if e = *0 then if 6* < *t* + *q then squanderTarget*
    else b *t q* − *t* ∗ *d3-const* − *q* ∗ *d4-const*
  else if t + *q* + *e* ≠ *5 then 0 else* a *t*

declare *d3-const-def* [*simp*] *d4-const-def* [*simp*]

constdefs *ExcessAt* :: *graph* ⇒ *vertex* ⇒ *nat*
ExcessAt g v ≡ *if* ¬ *finalVertex g v then 0*
  else excessAtType (*tri g v*) (*quad g v*) (*except g v*)

constdefs *ExcessTable* :: *graph* ⇒ *vertex list* ⇒ (*vertex* × *nat*) *list*
ExcessTable g vs ≡
  [(*v*, *ExcessAt g v*). *v* ∈ [*v* ∈ *vs*. *0* < *ExcessAt g v*]]

Implementation:

lemma [*code*]:
ExcessTable g =
filtermap (λ*v*. *let e* = *ExcessAt g v in if 0* < *e then Some* (*v*, *e*) *else None*)
by (*rule ext*) (*simp add*: *ExcessTable-def filtermap-conv*)

constdefs *deleteAround* ::
  graph ⇒ *vertex* ⇒ (*vertex* × *nat*) *list* ⇒ (*vertex* × *nat*) *list*
deleteAround g v ps ≡
  let fs = *facesAt g v*;
  ws = $\bigsqcup_{f \in fs}$ *if* |*vertices f*| = *4 then* [*f*•*v*, f^{2}•*v*] *else* [*f*•*v*] *in*
  removeKeyList ws ps

18 Tame Properties

theory *TameProps*
imports *Tame RTranCl*
begin

lemma *length-disj-filter-le*: ∀ *x* ∈ *set xs*. ¬(*P x* ∧ *Q x*) ⟹
length(*filter P xs*) + *length*(*filter Q xs*) ≤ *length xs*

```
by(induct xs) auto

lemma tri-quad-le-degree: tri g v + quad g v ≤ degree g v
proof –
  let ?fins = [f∈ facesAt g v . final f]
  have tri g v + quad g v =
        |[f∈ ?fins . triangle f]| + |[f∈ ?fins. |vertices f| = 4]|
    by(simp add:tri-def quad-def)
  also have ... ≤ |[f ∈ facesAt g v. final f]|
    by(rule length-disj-filter-le) simp
  also have ... ≤ |facesAt g v| by(rule length-filter-le)
  finally show ?thesis by(simp add:degree-def)
qed

lemma faceCountMax-bound:
 ⟦ tame g; v ∈ 𝒱 g ⟧ ⟹ tri g v + quad g v ≤ 6
using tri-quad-le-degree[of g v]
by(auto simp:tame-def tame₄₅-def split:split-if-asm)


lemma filter-tame-succs:
assumes invP: invariant P succs and fin: !!g. final g ⟹ succs g = []
and ok-untame: !!g. P g ⟹ ¬ ok g ⟹ final g ∧ ¬ tame g
and gg': g [succs]→* g'
shows P g ⟹ final g' ⟹ tame g' ⟹ g [filter ok o succs]→* 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(fastsimp 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.elims[OF succs(2)])(auto simp:fin)
    hence False using fin-tame succs by fast
    thus ?case ..
  qed
qed


constdefs
 untame :: (graph ⇒ bool) ⇒ bool
untame P ≡ ∀ g. final g ∧ P g ⟶ ¬ tame g
```

```
lemma filterout-untame-succs:
assumes invP: invariant P f and invPU: invariant (%g. P g ∧  U g) f
and untame: untame(%g. 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


lemma comp-map-tame-pres:
assumes invP: invariant P succs
and invPU: invariant (%g. P g ∧ U g) succs and untame: untame U
and f-fin: !!g. final g ⟹ f g = g
and invPUf: invariant (%g. P g ∧ U g) (%g. [f g])
and succs-f: !!g0 g g′. P g0 ⟹ g ∈ set(succs g0) ⟹ g′ ∈ set(succs g) ⟹
            U g′ ∨ f g = g ∨ f g′ = f g
and gg′: g [succs]→∗ g′
shows P g ⟹ final g′ ⟹ tame g′ ⟹ g [map f o succs]→∗ 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
  hence IH: h′ [map f ∘ succs]→∗ h′′ using succs by blast
  thus ?case
  proof (rule RTranCl-elim)
    assume h′′ = h′
    moreover hence f h′ = h′ using f-fin[OF succs(5)] by simp
    ultimately show ?case using succs(1)
      by(force intro!: RTranCl.succs[OF - RTranCl.refl])
```

```
  next
    fix h_2
    assume 1: h' [map f ∘ succs]→ h_2 and
           2: h_2 [map f ∘ succs]→* h''
    from 1 obtain h_1 where h_1: h_1 ∈ set(succs h') and [simp]: h_2 = f h_1
      by force
    { assume Uh_1: U h_1
      have invPU2: invariant (%g. P g ∧ U g) (map f o succs)
        using invPU invPUf by(auto simp:invariant-def)
      have Ph_1: P h_1 using succs(1) succs(4) h_1 invP
        by(unfold invariant-def) blast
      have PUh_2: P h_2 ∧ U h_2 using invariantE[OF invPUf] Ph_1 Uh_1 by auto
      from RTranCl-inv[OF invPU2 2, OF PUh_2]
      have False using succs(5−6) untame by(auto simp:untame-def)
      hence ?thesis ..
    } moreover
    { assume f h' = h'
      hence h' ∈ set((map f o succs) h) using succs(1) by force
      hence ?thesis using IH by(rule RTranCl.succs)
    } moreover
    { assume ff: f h_1 = f h'
      hence h_2 ∈ set((map f o succs) h) using succs(1) by auto
      hence ?thesis using 2 by(rule RTranCl.succs)
    }
    ultimately show ?thesis using succs-f[OF succs(4) succs(1) h_1] by blast
  qed
qed

end
```

19 All About Finalizing Triangles

```
theory Plane3Props
imports Plane1Props Generator TameProps
begin
```

19.1 Correctness

```
lemma decomp-nonFinal3:
assumes mgp: minGraphProps g
and ffs: f#fs = [f ∈ faces g. ¬ final f ∧ triangle f]
shows f = minimalFace(nonFinals g) &
      fs = [f ∈ faces(makeFaceFinal (minimalFace(nonFinals g)) g).
            ¬ final f ∧ triangle f] (is ?A & ?B)
proof −
  have 3: ∀f ∈ ℱ g. 3 ≤ |vertices f|
    using minGraphProps2[OF mgp] by fastsimp
```

```
  have ffs': f#fs = [f ∈ nonFinals g. triangle f] using ffs
    by (simp add:nonFinals-def)
  from Cons-eq-filterD[OF ffs'] obtain us vs where
    [simp]: nonFinals g = us @ f # vs triangle f
      fs = [f∈vs . triangle f] and us: ∀ u∈set us. ¬ triangle u
    by blast
  have ∀ u ∈ set(nonFinals g). u ∈ ℱ g by(simp add:nonFinals-def)
  hence usg: ∀ u∈set us. u ∈ ℱ g
    and vsg: ∀ u∈set vs. u ∈ ℱ g by simp+
  let ?m = size o vertices
  have us3: ∀ u∈set us. |vertices u| > 3
  proof
    fix f' assume f' ∈ set us
    thus |vertices f'| > 3
      using bspec[OF us, of f'] bspec[OF 3, of f'] bspec[OF usg, of f']
      by simp
  qed
  have A: ?A
  proof −
    have minimal ?m (us @ f # vs) = minimal ?m (f # vs)
      using us3 by(simp add: minimal-append2)
    also have ... = f using 3 vsg
      by (rule-tac minimal-Cons1) simp
    finally show ?A by(simp add: minimalFace-def)
  qed
  moreover have ?B (is ?l = ?r)
  proof −
    have ?r = [h ∈ faces (makeFaceFinal f g) . ¬ final h ∧ triangle h]
      using A by simp
    also have ... = [h ∈ nonFinals (makeFaceFinal f g). triangle h]
      by(simp add:nonFinals-def)
    also have ... = [h ∈ remove1 f (nonFinals g). triangle h]
      by(simp add:nonFins-mkFaceFin)
    also have ... = [h ∈ vs. triangle h]
    proof −
      have f ∉ set us using us by auto
      thus ?thesis using us by(simp add:remove1-append)
    qed
    finally show ?B by simp
  qed
  ultimately show ?thesis ..
qed


lemma noDupEdge3:
 ⟦ minGraphProps g; f ∈ ℱ g; triangle f; v ∈ 𝒱 f ⟧
 ⟹ ¬ containsDuplicateEdge g f v [0..<3]
apply(frule (1) minGraphProps3)
apply(drule length3D)
```

```
apply auto
apply(simp add: containsDuplicateEdge-eq containsDuplicateEdge'-def
          nat-number nextVertices-def nextVertex-def
      duplicateEdge-is-duplicateEdge-eq is-duplicateEdge-def is-nextElem-def)
apply(simp add:is-sublist-rec[where 'a = nat])
apply(simp add: containsDuplicateEdge-eq containsDuplicateEdge'-def
          nat-number nextVertices-def nextVertex-def
      duplicateEdge-is-duplicateEdge-eq is-duplicateEdge-def is-nextElem-def)
apply(simp add: containsDuplicateEdge-eq containsDuplicateEdge'-def
          nat-number nextVertices-def nextVertex-def
      duplicateEdge-is-duplicateEdge-eq is-duplicateEdge-def is-nextElem-def)
apply(simp add:is-sublist-rec[where 'a = nat])
done

lemma indexToVs3:
 ⟦ triangle f; distinct(vertices f); v ∈ 𝒱 f ⟧
 ⟹ indexToVertexList f v [0..<3] = [Some v, Some(f · v), Some(f · (f · v))]
apply(auto dest!: length3D)
apply(auto simp:indexToVertexList-def nat-number
          nextVertices-def nextVertex-def)
done

lemma upt3-in-enumerator: [0..<3] ∈ set (enumerator 3 3)
apply(simp only:enumerator-def incrIndexList-def enumBase-def)
apply(simp add:nat-number)
done

lemma mkFaceFin3-in-succs1:
assumes mgp: minGraphProps g
and ffs: f#fs = [f ∈ faces g. ¬ final f ∧ triangle f]
shows Graph (makeFaceFinalFaceList f (faces g)) (countVertices g)
      (map (makeFaceFinalFaceList f) (faceListAt g)) (heights g)
   ∈ set (next-plane_p g) (is ?g' ∈ -)
proof −
  have ¬ final g using Cons-eq-filterD[OF ffs]
    by(auto simp: finalGraph-def nonFinals-def)
  moreover have ?g' ∈ set (generatePolygon 3 (minimalVertex g f) f g)
  proof −
    have [simp]: |vertices f| = 3 and fg: f ∈ ℱ g
      using Cons-eq-filterD[OF ffs] by auto
    have vertices f ≠ [] by (auto simp:length-0-conv[symmetric])
    hence min: minimalVertex g f ∈ 𝒱 f  by(simp add:minimalVertex-def)
    note upt3-in-enumerator noDupEdge3[OF mgp fg - min]
    moreover have
      ?g' = subdivFace g f (indexToVertexList f (minimalVertex g f) [0..<3])
      using min minGraphProps3[OF mgp fg]
        by(simp add:indexToVs3 makeFaceFinal-def subdivFace-def)
    ultimately show ?thesis
      by (fastsimp simp:generatePolygon-def image-def del:enumerator-equiv)
```

qed
moreover have *f* = *minimalFace* (*nonFinals g*)
using *decomp-nonFinal3*[*OF mgp ffs*] **by** *simp*
ultimately show *?thesis* **by**(*auto simp add*:*next-plane-def finalGraph-def*)
qed

lemma *mkFaceFin3-in-rtrancl*:
minGraphProps g ⟹ *f*#*fs* = [*f* ∈ *faces g* . ¬ *final f* ∧ *triangle f*] ⟹
g [$\textit{next-plane}_p$]→∗ *makeFaceFinal f g*
apply(*simp add*:*makeFaceFinal-def*)
apply(*rule RTranCl.succs*[*OF - RTranCl.refl*])
apply(*erule* (*1*) *mkFaceFin3-in-succs1*)
done

lemma *mk3Fin-lem*:
⋀*g*. *minGraphProps g* ⟹ *fs* = [*f* ∈ *faces g*. ¬ *final f* ∧ *triangle f*] ⟹
g [$\textit{next-plane}_p$]→∗ *foldl* (%*g f*. *makeFaceFinal f g*) *g fs*
proof(*induct fs*)
case *Nil* **show** *?case* **by** (*simp add*:*RTranCl.refl*)
next
case (*Cons f fs*)
let *?g* = *makeFaceFinal f g*
note *mkFaceFin3-in-rtrancl*[*OF Cons*(*2*−*3*)] **moreover**
have *?g* [$\textit{next-plane}_p$]→∗ *foldl* (%*g f*. *makeFaceFinal f g*) *?g fs*
using *Cons-eq-filterD*[*OF Cons*(*3*)] *Cons*(*2*)
by(*rule-tac Cons*(*1*)[*OF MakeFaceFinal-minGraphProps*])
(*auto simp*: *makeFaceFinal-def makeFaceFinalFaceList-def pre-subdivFace′-def*)
ultimately show *?case* **by**(*rule-tac RTranCl-compose*) *auto*
qed

lemma *mk3Fin-in-RTranCl*:
inv g ⟹ *g* [$\textit{next-plane}_p$]→∗ *makeTrianglesFinal g*
by(*simp add*:*makeTrianglesFinal-def mk3Fin-lem*)

19.2 Completeness

constdefs
in2finals g a b ≡
∃*f* ∈ *set*(*finals g*). ∃*f′* ∈ *set*(*finals g*). (*a*,*b*) ∈ $\mathcal{E}$ *f* ∧ (*b*,*a*) ∈ $\mathcal{E}$ *f′*

$untame_2$ :: *graph* ⇒ *bool*
$untame_2$ *g* ≡ ∃ *a b c*. *is-cycle g* [*a*,*b*,*c*] ∧ *distinct*[*a*,*b*,*c*] ∧
(∀*f* ∈ $\mathcal{F}$ *g*. $\mathcal{E}$ *f* ≠ {(*c*,*a*), (*a*,*b*), (*b*,*c*)} ∧
$\mathcal{E}$ *f* ≠ {(*a*,*c*), (*c*,*b*), (*b*,*a*)}) ∧
in2finals g a b

lemma *untame2*: *untame*($untame_2$)

```
apply(auto simp:untame-def untame_2-def tame-def tame_2-def Edges-Cons)
apply(erule allE,erule allE,erule allE,erule impE,assumption)
apply(bestsimp dest!:edges-conv-Edges-if-cong simp: Edges-Cons)
done

lemma mk3Fin-id: final g ⟹ makeTrianglesFinal g = g
apply(subgoal-tac [f∈faces g . ¬ final f ∧ triangle f] = [])
 apply(simp add: makeTrianglesFinal-def)
apply(simp add: finalGraph-def nonFinals-def filter-empty-conv)
done


lemma inv-untame2:
 invariant (λg. inv g ∧ untame_2 g) next-plane_p
proof (clarsimp simp:invariant-def invariantE[OF inv-inv-next-plane])
  fix g g' assume g': g [next-plane_p]→ g'
    and inv: inv g and un: untame_2 g
  note mgp' = inv-mgp[OF invariantE[OF inv-inv-next-plane, OF g' inv]]
  show untame_2 g' using un
  proof(clarsimp simp:untame_2-def)
    fix a b c
    assume 3: is-cycle g [a,b,c] and abc: a ≠ b a ≠ c b ≠ c
      and untame: ∀f∈ ℱ g.
            ℰ f ≠ {(c, a), (a, b), (b, c)} ∧
            ℰ f ≠ {(a, c), (c, b), (b, a)} (is ?P g a b c)
      and in2: in2finals g a b
    from in2 obtain f_1 f_2 :: face
      where f_1: f_1 ∈ ℱ g ∧ final f_1 ∧ (a,b) ∈ ℰ f_1
      and   f_2: f_2 ∈ ℱ g ∧ final f_2 ∧ (b,a) ∈ ℰ f_2
      by(auto simp:finals-def in2finals-def)+
    note mgp = inv-mgp[OF inv]
    note g'1 = mgp-next-plane0-if-next-plane[OF mgp g']
    have f_1': f_1 ∈ ℱ g' and f_2': f_2 ∈ ℱ g'
      using next-plane0-finals-incr[OF g'1] f_1 f_2 by(auto simp:finals-def)
    have is-cycle g' [a,b,c]
      using 3 next-plane0-set-edges-subset[OF mgp g'1]
      by(auto simp:is-cycle-def edges-graph-def
        neighbors-edges[OF mgp] neighbors-edges[OF mgp'])
    moreover have ?P g' a b c
    proof (rule ccontr, auto)
      fix f assume fg: f ∈ ℱ g' and Ef: ℰ f = {(c,a), (a,b), (b,c)}
      moreover have f ≠ f_1 using untame f_1 Ef by blast
      ultimately show False
        using f_1 f_1' by(blast dest: mgp-edges-disj[OF mgp'])
    next
      fix f assume fg: f ∈ ℱ g' and Ef: ℰ f = {(a,c), (c,b), (b,a)}
      moreover have f ≠ f_2 using untame f_2 Ef by blast
      ultimately show False
        using f_2 f_2' by(blast dest: mgp-edges-disj[OF mgp'])
```

```
    qed
    moreover have in2finals g′ a b using in2 next-plane0-finals-subset[OF g′1]
      by(auto simp:in2finals-def)
    ultimately
    show ∃ a b c. is-cycle g′ [a,b,c] ∧ a≠b ∧ a≠c ∧ b≠c ∧ ?P g′ a b c ∧ in2finals
g′ a b
      using abc by blast
  qed
qed
```

```
lemma mk3Fin-id2:
assumes mgp: minGraphProps g and nf: nonFinals g ≠ []
and n3: ¬ triangle(minimalFace(nonFinals g))
shows makeTrianglesFinal g = g
proof −
  have ∀ f∈set (nonFinals g). 3 ≤ |vertices f|
    using mgp-vertices3[OF mgp] by(simp add:nonFinals-def)
  with n3 have ∀ f ∈ set(nonFinals g). ¬ triangle f
    by(rule-tac minimal-neq-lowerbound[OF nf])
      (auto simp:minimalFace-def o-def)
  hence ∀ f ∈ set(faces g). final f ∨ ¬ triangle f
    by(auto simp:nonFinals-def)
  thus ?thesis  by(simp add:makeTrianglesFinal-def)
qed
```

```
lemma mk3Fin-mkFaceFin:
assumes mgp: minGraphProps g and nf: nonFinals g ≠ []
and 3: triangle(minimalFace(nonFinals g))
shows makeTrianglesFinal (makeFaceFinal (minimalFace (nonFinals g)) g) =
       makeTrianglesFinal g
proof −
  let ?f = minimalFace (nonFinals g)
  from nf 3 have minimalFace(nonFinals g) ∈ set(nonFinals g)
    by (simp add:minimalFace-def)
  with 3 have [f ∈ faces g. ¬ final f ∧ triangle f] ≠ []
    by (auto simp: filter-empty-conv nonFinals-def)
  then obtain f fs where ffs: f#fs = [f ∈ faces g. ¬ final f ∧ triangle f]
    by(fastsimp simp add:neq-Nil-conv)
  with decomp-nonFinal3[OF mgp ffs]
  have [f ∈ faces g. ¬ final f ∧ triangle f] =
       ?f # [f ∈ faces(makeFaceFinal ?f g). ¬ final f ∧ triangle f] by(simp)
  thus ?thesis by(simp add:makeTrianglesFinal-def)
qed
```

```
lemma next-plane-mk3Fin-alternatives:
```

```
assumes inv: inv g and 2: |faces g| ≠ 2 and 1: g [next-plane_p]→ g'
shows untame_2 g' ∨ makeTrianglesFinal g = g ∨
      makeTrianglesFinal g' = makeTrianglesFinal g
proof −
  note mgp = inv-mgp[OF inv]
  note mgp' = inv-mgp[OF invariantE[OF inv-inv-next-plane, OF 1 inv]]
  have nf: nonFinals g ≠ [] using 1 by(auto simp:next-plane-def)
  note gg' = mgp-next-plane0-if-next-plane[OF mgp 1]
  show ?thesis
  proof (cases triangle(minimalFace(nonFinals g)))
    case True
    let ?f = minimalFace(nonFinals g)
    have f: ?f ∈ set(nonFinals g) using nf by (simp add:minimalFace-def)
    hence fg: ?f ∈ ℱ g by(simp add:nonFinals-def)
    from f have fnf: ¬ final ?f by(simp add:nonFinals-def)
    note distf = minGraphProps3[OF mgp fg]
    def a ≡ minimalVertex g ?f
    from mgp-vertices-nonempty[OF mgp fg]
    have v: a ∈ set(vertices ?f) by (simp add:a-def minimalVertex-def)
    from 1 obtain i e
      where g'0: g' = subdivFace g ?f (indexToVertexList ?f a e)
      and e: e ∈ set (enumerator i |vertices ?f| ) and i: 2 < i
      and containsNot: ¬ containsDuplicateEdge g ?f a e
      by (auto simp: a-def next-plane-def generatePolygon-def image-def
              split:split-if-asm)
    note pre-add = pre-subdivFace-indexToVertexList[OF mgp f v e containsNot i]
    with g'0 have g': g' = subdivFace' g ?f a 0 (tl(indexToVertexList ?f a e))
      by (simp  add: subdivFace-subdivFace'-eq)
    from pre-add have indexToVertexList ?f a e ≠ []
      by (simp add: pre-subdivFace-def pre-subdivFace-face-def)
    with pre-add v
    have pre: pre-subdivFace' g ?f a a 0 (tl(indexToVertexList ?f a e))
      by(fastsimp simp add:neq-Nil-conv elim:pre-subdivFace-pre-subdivFace')
    have g' = makeFaceFinal ?f g ∨
         (∀f' ∈ ℱ g' − (ℱ g − {?f}). ℰ f' ≠ ℰ ?f) (is ?A ∨ ?B)
      by(simp only:g')(rule dist-edges-subdivFace'[OF pre mgp fg])
    thus ?thesis
    proof
      assume ?A
      hence makeTrianglesFinal g' = makeTrianglesFinal g
        by(simp add:mk3Fin-mkFaceFin[OF mgp nf True])
      thus ?thesis by blast
    next
      assume B: ?B
      obtain b c where fabs: vertices ?f = [a,b,c] ∨ vertices ?f = [c,a,b] ∨
                       vertices ?f = [b,c,a]
        using v True by(fastsimp simp: nat-number length-Suc-conv)
      hence Ef: ℰ ?f = {(a,b), (b,c), (c,a)}
        by(fastsimp simp: edges-conv-Edges[OF distf] Edges-Cons)
```

```
hence {(a,b), (b,c), (c,a)} ⊆ ℰ g'
  using fg next-plane0-set-edges-subset[OF mgp gg']
  by(unfold edges-graph-def) blast
hence 3: is-cycle g' [c,a,b]
  by(auto simp: is-cycle-def neighbors-edges[OF mgp'])
have abc: distinct[a,b,c] using distf fabs by auto
obtain f' where f': f' ∈ ℱ g ∧ final f' ∧ (a,c) ∈ ℰ f'
  using Ef inv-one-finalD'[OF inv fg fnf]
  by blast
hence f'g': f' ∈ ℱ g'
  using f' next-plane0-finals-subset[OF gg'] by (simp add:finals-def)blast
have ?f^-1 · a = c using fabs abc by(auto simp:prevVertex-def)
then obtain f'' where f'': f'' ∈ ℱ g' ∧ final f'' ∧ (c,a)∈ ℰ f''
  using g' subdivFace'-final-base[OF mgp pre fg] by simp blast
{ fix ff assume ffg': ff ∈ ℱ g'
  assume ℰ ff = {(b,c), (c,a), (a,b)} ∨
          ℰ ff = {(c,b), (b,a), (a,c)} (is ?ab ∨ ?ba)
  hence False
  proof
    assume Eff: ?ab
    have ff ∉ ℱ g − {?f}
    proof
      assume ff ∈ ℱ g − {?f}
      hence ffg: ff ∈ ℱ g and neq: ff ≠ ?f by auto
      note distff = minGraphProps3[OF mgp ffg]
      have ff = ?f using Ef Eff
        by(blast intro: face-eq-if-normFace-eq[OF mgp ffg fg]
                  normFace-eq-if-edges-eq[OF distff distf])
      with neq show False by fast
    qed
    with B ffg' have ℰ ff ≠ ℰ ?f by blast
    with Eff Ef show False by blast
  next
    assume Eff: ?ba
    obtain h1 :: face where h1g: h1 ∈ ℱ g and fh1: final h1
      and ach1: (a,c) : ℰ h1
      using inv-one-finalD'[OF inv fg fnf] Ef by blast
    obtain h2 :: face where h2g: h2 ∈ ℱ g and fh2: final h2
      and cbh2: (c,b) : ℰ h2
      using inv-one-finalD'[OF inv fg fnf] Ef by blast
    obtain h3 :: face where h3g: h3 ∈ ℱ g and fh3: final h3
      and bah3: (b,a) : ℰ h3
      using inv-one-finalD'[OF inv fg fnf] Ef by blast
    have h1g': h1 ∈ ℱ g'
      using h1g fh1 subdivFace-pres-finals[OF - fnf] g'0 fg
      by(simp add:finals-def)
    have h2g': h2 ∈ ℱ g'
      using h2g fh2 subdivFace-pres-finals[OF - fnf] g'0 fg
      by(simp add:finals-def)
```

```
          have h3g': h3 ∈ ℱ g'
            using h3g fh3 subdivFace-pres-finals[OF - fnf] g'0 fg
            by(simp add:finals-def)
          note mgp' = inv-mgp[OF inv-inv-next-plane0[THEN invariantE, OF gg'
inv]]
          note disth1 = minGraphProps3[OF mgp h1g]
          have ¬(h1 = h2 ∧ h1 = h3 ∧ h2 = h3)
          proof
            assume [simp]: h1 = h2 ∧ h1 = h3 ∧ h2 = h3
            have face-face-op g using mgp by(simp add:minGraphProps-def)
            moreover have h1 ≠ ?f using fnf fh1 by auto
            ultimately have ℰ h1 ≠ (ℰ ?f)^-1 using 2 h1g fg
              by(simp add:face-face-op-def)
            moreover have ℰ h1 = {(c,b),(b,a),(a,c)}
              using abc ach1 cbh2 bah3
              by(rule-tac triangle-if-3circular[OF - disth1])auto
            ultimately show False using Ef by fastsimp
          qed
          hence ¬(h1 = ff ∧ h2 = ff ∧ h3 = ff) by blast
          thus False using Eff
            mgp-edges-disj[OF mgp' - h1g' ffg' ach1]
            mgp-edges-disj[OF mgp' - h2g' ffg' cbh2]
            mgp-edges-disj[OF mgp' - h3g' ffg' bah3]
            by blast
        qed }
      with 3 abc f' f'g' f'' have untame2 g'
        by(simp add: untame2-def in2finals-def finals-def) blast
      thus ?thesis by blast
    qed
  next
    case False
    thus ?thesis by(simp add:mk3Fin-id2[OF mgp nf])
  qed
qed


theorem make3Fin-complete:
 ⟦ invariant inv (succ p);
   ⋀g. inv g ⟹ set (succ p g) ⊆ set (next-plane p g);
   tame g; final g; Seedp [succ p]→* g ⟧ ⟹
 Seedp [map makeTrianglesFinal o succ p]→* g
apply(rule comp-map-tame-pres[OF - - untame2])
        apply assumption
       apply(rule inv-subset[OF inv-untame2[where p=p]])
       apply blast
      apply(erule mk3Fin-id)
     apply(rule inv-RTranCl-subset[OF inv-untame2[where p=p]])
     apply(rule mk3Fin-in-RTranCl)
     apply blast
```

```
    defer
    apply assumption
   apply(rule inv-Seed)
  apply assumption
 apply assumption
apply(rule next-plane-mk3Fin-alternatives)
  apply(rule invariantE[OF inv-inv-next-plane])
   apply blast
  apply assumption
 apply(rule step-outside2)
   apply assumption
  apply(rule mgp-next-plane0-if-next-plane[OF inv-mgp])
   apply assumption
  apply blast
 apply(simp add:finalGraph-def)
 apply(rule next-plane0-nonfinals)
 apply(rule mgp-next-plane0-if-next-plane[OF inv-mgp])
  apply(rule invariantE[OF inv-inv-next-plane])
   apply blast
  apply assumption
 apply (blast dest:invariantE)
apply (blast dest:invariantE)
done

end
```

20 Checking Final Quadrilaterals

```
theory Plane4
imports While-Combinator Tame
begin

constdefs
 norm-subset :: vertex list list ⇒ vertex list list ⇒ bool
norm-subset xs ys ≡ let zs = map rotate-min ys
                    in ∀ x ∈ set xs. rotate-min x ∈ set zs

consts remrevdups :: 'a list list ⇒ 'a list list
primrec
remrevdups [] = []
remrevdups (xs#xss) = (if xs mem xss ∨ rev xs mem xss then remrevdups xss
                       else xs # remrevdups xss)

constdefs
 find-cycles1 :: nat ⇒ graph ⇒ vertex ⇒ vertex list list
find-cycles1 n g v ≡
 snd(snd(
```

```
 while (%(vs,wss,res). vs ≠ [])
 (%(vs,wss,res).
   let ws = hd wss in
   if ws = [] then (tl vs, tl wss, res)
   else if length vs = n
        then (tl vs, tl wss, if last vs ∈ set ws then vs#res else res)
        else let vs' = (if length vs + 1 = n then butlast vs else vs);
                 xs = filter (%x. x ∉ set vs') (neighbors g (hd ws)) in
             (hd ws # vs, xs # tl ws # tl wss, res))
 ([v], [neighbors g v], [])
 ))

 find-cycles :: nat ⇒ graph ⇒ vertex list list
find-cycles n g ≡
 remrevdups(map rotate-min (foldr (%v vss. find-cycles1 n g v @ vss) (vertices g)
[]))
```

constdefs

```
 ok42 :: vertex list ⇒ vertex list list ⇒ vertex ⇒ vertex ⇒ vertex ⇒ vertex ⇒
bool
ok42 vs fs a b c d ==
  norm-subset (tameConf_1 a b c d) fs ∨
  norm-subset (tameConf_2 a b c d) fs ∨
  norm-subset (tameConf_2 b c d a) fs ∨
  (EX e:set vs. e ∉ set[a,b,c,d] ∧
      (norm-subset (tameConf_3 a b c d e) fs ∨
       norm-subset (tameConf_3 b c d a e) fs ∨
       norm-subset (tameConf_3 c d a b e) fs ∨
       norm-subset (tameConf_3 d a b c e) fs ∨
       norm-subset (tameConf_4 a b c d e) fs)
  )

 ok4 :: graph ⇒ vertex list ⇒ bool
ok4 g vs  ≡
 let fs = map vertices (faces g); gvs = vertices g;
     a = hd vs; b = hd(tl vs); c = hd(tl(tl vs)); d = hd(tl(tl(tl vs)))
 in ok42 gvs fs a b c d ∨ ok42 gvs fs d c b a

 is-tame_3 :: graph ⇒ bool
is-tame_3 g ≡ ∀ vs ∈ set(find-cycles 4 g).
   is-cycle g vs ∧ distinct vs ∧ |vs| = 4 −→ ok4 g vs
```

end

21 Neglectable Final Graphs

theory *TameEnum*
imports *Generator Plane4*
begin

constdefs
$\textit{is-tame} :: \textit{graph} \Rightarrow \textit{bool}$
$\textit{is-tame}\ g\ \equiv\ \textit{tame}_{45}\ g \wedge \textit{tame}_{6}\ g \wedge \textit{tame}_{8}\ g \wedge \textit{is-tame}_{7}\ g \wedge \textit{is-tame}_{3}\ g$

constdefs
$\textit{next-tame} :: \textit{nat} \Rightarrow \textit{graph} \Rightarrow \textit{graph list}\ \ (\textit{next}'\textit{-tame}_{-})$
$\textit{next-tame}_{p} \equiv \textit{filter}\ (\lambda g.\ \neg\ \textit{final}\ g \vee \textit{is-tame}\ g)\ o\ \textit{next-tame1}_{p}$

constdefs
$\textit{TameEnumP} :: \textit{nat} \Rightarrow \textit{graph set}\ (\textit{TameEnum}_{-})$
$\textit{TameEnum}_{p} \equiv \{g.\ \textit{Seed}_{p}\ [\textit{next-tame}_{p}]{\rightarrow}{*}\ g \wedge \textit{final}\ g\}$

$\textit{TameEnum} :: \textit{graph set}$
$\textit{TameEnum} \equiv \bigcup p{\le}5.\ \textit{TameEnum}_{p}$

end

22 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*)
apply (*auto simp add*: *deleteAround-def*)
    apply *blast*
   apply (*fastsimp simp*:*nextV2*)
  apply(*drule-tac x* = [$f \cdot a$, $f^{2} \cdot a$] **in** *bspec*)
   apply (*fastsimp simp*:*image-def*)
  apply (*simp*)
 apply(*drule-tac x* = [$f \cdot a$, $f^{2} \cdot a$] **in** *bspec*)
  apply (*fastsimp simp*:*image-def*)

```
 apply (simp add:nextV2)
apply(drule-tac x = [f · a] in bspec)
 apply (fastsimp simp:image-def)
apply (simp)
done

lemma deleteAround-subset: set (deleteAround g a ps) ⊆ set ps
by (simp add: deleteAround-def)

lemma distinct-deleteAround: distinct (map fst ps) =⇒
    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 ∉ fst ‘ set ps by simp
  moreover have set (deleteAround g a ps) ⊆ set ps
    by (rule deleteAround-subset)
  ultimately have fst p ∉ 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


constdefs
  deleteAround′ :: graph ⇒ vertex ⇒ (vertex × nat) list ⇒
    (vertex × nat) list
  deleteAround′ g v ps ≡
      let fs = facesAt g v;
      vs = (λf. let n1 = f · v;
                n2 = f · 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: split-if-asm)
apply (unfold nextV2[THEN eq-reflection], simp)
done

lemma deleteAround-nextVertex:
  f ∈ set (facesAt g a) =⇒
  (f · a, b) ∉ set (deleteAround g a ps)
  by (auto simp add: deleteAround-eq deleteAround′-def removeKeyList-eq)
```

```
lemma deleteAround-nextVertex-nextVertex:
  f ∈ set (facesAt g a) ⟹ |vertices f| = 4 ⟹
  (f • (f • a), b) ∉ set (deleteAround g a ps)
  by (auto simp add: deleteAround-eq deleteAround'-def removeKeyList-eq)


lemma deleteAround-prevVertex:
  minGraphProps g ⟹ f ∈ set (facesAt g a) ⟹
  (f⁻¹ • a, b) ∉ set (deleteAround g a ps)
proof −
  assume a: minGraphProps g f ∈ set (facesAt g a)
  have (f⁻¹ • a, a) ∈ ℰ f using a
    by(blast intro:prevVertex-in-edges minGraphProps)
  then obtain f' where f': f' ∈ set(facesAt g a)
    and e: (a, f⁻¹ • a) ∈ ℰ f'
    using a by(blast dest:mgp-edge-face-ex)
  have (f' • a, b) ∉ set (deleteAround g a ps) using f'
    by (auto simp add: deleteAround-eq deleteAround'-def removeKeyList-eq)
  moreover have f' • a = f⁻¹ • a
    using e by (simp add:edges-face-eq)
  ultimately show ?thesis by simp
qed


lemma deleteAround-separated:
  minGraphProps g ⟹ final g ⟹ |vertices f| ≤ 4 ⟹ f ∈ set(facesAt g a) ⟹
  𝒱 f ∩ set [fst p. p ∈ deleteAround g a ps] ⊆ {a}
  (concl is ?A)
proof −
  assume fin: final g and mgp: minGraphProps g and 4: |vertices f| ≤ 4
  assume f: f ∈ set (facesAt g a)
  then have a: a ∈ 𝒱 f using mgp f by(blast intro:minGraphProps)
  have 2 < |vertices f| using mgp f by(blast intro:minGraphProps)
  with 4 have |vertices f| = 3 ∨ |vertices f| = 4 by arith
  then show ?A
  proof
    assume 3: |vertices f| = 3
    show ?A
    proof (rule ccontr)
      assume ¬ ?A
      then obtain b where b1: b ≠ a b ∈ 𝒱 f
        b ∈ set (map fst (deleteAround g a ps)) by auto
      from mgp f have d: distinct (vertices f)
        by(blast intro:minGraphProps)
      with a 3 have 𝒱 f = {a, f • a, f • (f • a)}
          by (rule-tac vertices-triangle)
      also from d a 3 have
        f • (f • a) = f⁻¹ • a
        by (simp add: triangle-nextVertex-prevVertex)
```

```
      finally have
        b ∈ {f · a, f^-1 · a}
        using b1 by simp
      with mgp f have b ∉ set (map fst (deleteAround g a ps))
      using deleteAround-nextVertex deleteAround-prevVertex by auto
      then show False by contradiction
    qed
  next
    assume 4: |vertices f| = 4
    show ?A
    proof (rule ccontr)
      assume ¬ ?A
      then obtain b where b1: b ≠ a b ∈ 𝒱 f
        b ∈ set (map fst (deleteAround g a ps)) by auto
      from mgp f have d: distinct (vertices f) by(blast intro:minGraphProps)
      with a 4 have 𝒱 f = {a, f · a, f · (f · a), f · (f · (f · a))}
          by (rule-tac vertices-quad)
      also from d a 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 f 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
    qed
  qed
qed



lemma [iff]: preSeparated g {}
  by (simp add: preSeparated-def separated_2-def separated_3-def)

lemma preSeparated-insert:
assumes mgp: minGraphProps g and a: a ∈ 𝒱 g
  and ps: preSeparated 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 preSeparated g (insert a V)
proof (simp add: preSeparated-def separated_2-def separated_3-def,
 intro conjI ballI impI)
  fix f assume f ∈ set (facesAt g a)
  then show f · a ≠ a by (rule mgp-facesAt-no-loop[OF mgp])
  show f · a ∉ V by (rule s2)
next
  fix f v assume v: f ∈ set (facesAt g v) and vV: v ∈ V
  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] by blast
    have f′ • a ∈ V using v by simp
    with f′ s2 show False by blast
  qed
  from ps v vV show f • v ∉ V
    by (simp add: preSeparated-def separated₂-def)
next
  fix f assume f:  f ∈ set (facesAt g a) |vertices f| ≤ 4
  have 𝒱 f ∩ V ⊆ {a} by (rule-tac s3)
  moreover from mgp f have a ∈ 𝒱 f by(blast intro:minGraphProps)
  ultimately show 𝒱 f ∩ insert a V = {a} by auto
next
  fix v f
  assume a: v ∈ V f ∈ set (facesAt g v)
    |vertices f| ≤ 4
  with ps have v: 𝒱 f ∩ V = {v}
    by (simp add: preSeparated-def separated₃-def)
  show  𝒱 f ∩ insert a V = {v}
  proof cases
    assume a = v
    with v mgp a show ?thesis by(blast intro:minGraphProps)
  next
    assume n: a ≠ v
    have  a ∉ 𝒱 f
    proof
      assume a2: a ∈ 𝒱 f
      with mgp a have f ∈ ℱ g by(blast intro:minGraphProps)
      with mgp a2 have f ∈ set (facesAt g a) by(blast intro:minGraphProps)
      with a have 𝒱 f ∩ V ⊆ {a} by (simp add: s3)
      with v have a = v by auto
      with n show False by auto
    qed
    with a  v show 𝒱 f ∩ insert a V = {v} by blast
  qed
qed
```

```
consts ExcessNotAtRecList :: (vertex, nat) table ⇒ graph ⇒ vertex list
recdef ExcessNotAtRecList measure (λps. size ps)
ExcessNotAtRecList [] = (%g. [])
ExcessNotAtRecList (p#ps) = (%g.
  let l1 = ExcessNotAtRecList ps g;
  l2 = ExcessNotAtRecList (deleteAround g (fst p) ps) g in
  if ExcessNotAtRec ps g
   ≤ snd p + ExcessNotAtRec (deleteAround g (fst p) ps) g
```

```
  then fst p#l2 else l1)
(hints recdef-simp: less-Suc-eq-le length-deleteAround)


lemma isTable-deleteAround:
  isTable E vs ((a,b)#ps) ⟹ isTable E vs (deleteAround g a ps)
  by (rule isTable-subset, rule deleteAround-subset,
    rule isTable-Cons)

lemma ListSum-ExcessNotAtRecList:
 isTable E vs ps ⟹ ExcessNotAtRec ps g
  = ∑ p ∈ ExcessNotAtRecList ps g E p (is ?T ps ⟹ ?P ps)
proof (induct ps rule: ExcessNotAtRec.induct)
  show ?P [] by simp
next
  fix a b ps
  assume prem: ?T ((a,b)#ps)
  then have E: b = E a by (simp add: isTable-eq)
  assume hyp1: ?T (deleteAround g (fst (a, b)) ps) ⟹
   ?P (deleteAround g (fst (a, b)) ps)
  assume hyp2:  ?T ps ⟹ ?P ps
  have H1: ?P (deleteAround g (fst (a, b)) ps)
    by (rule hyp1, rule isTable-deleteAround) simp
  have H2: ?P ps by (rule hyp2, rule isTable-Cons)

  show ?P ((a,b)#ps)
  proof cases
    assume
    ExcessNotAtRec ps g
    ≤ b + ExcessNotAtRec (deleteAround g a ps) g
    with H1 E show ?thesis
      by (simp add: max-def split: split-if-asm)
  next
    assume ¬ ExcessNotAtRec ps g
       ≤ b + ExcessNotAtRec (deleteAround g a ps) g
    with H2 E show ?thesis
      by (simp add: max-def split: split-if-asm)
  qed
qed

lemma ExcessNotAtRecList-subset:
  set (ExcessNotAtRecList ps g) ⊆ set [fst p. p ∈ ps] (is ?P ps)
proof (induct ps rule: ExcessNotAtRecList.induct)
  show ?P [] by simp
next
  fix 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
      ≤ b + ExcessNotAtRec (deleteAround g a ps) g
    have set (deleteAround g a ps) ⊆ set ps
      by (simp add: deleteAround-subset)
    then have
    fst ‘ set (deleteAround g a ps) ⊆ insert a (fst ‘ set ps)
      by blast
    with a H1 show ?thesis by (simp)
  next
    assume ¬ ExcessNotAtRec ps g
      ≤ b + ExcessNotAtRec (deleteAround g a ps) g
    with H2 show ?thesis by (auto)
  qed
qed simp


lemma preSeparated-ExcessNotAtRecList:
 minGraphProps g ⟹ final g ⟹ isTable E (vertices g) ps ⟹
  preSeparated g (set (ExcessNotAtRecList ps g))
proof −
  assume fin: final g and mgp: minGraphProps g
  show
   isTable E (vertices g) ps ⟹ preSeparated g (set (ExcessNotAtRecList ps g))
   (is ?T ps ⟹ ?P ps)
  proof (induct rule: ExcessNotAtRec.induct)
    show ?P [] by simp
  next
    fix a b ps
    assume prem: ?T ((a,b)#ps)
    then have E: b = E a by (simp add: isTable-eq)
    assume hyp1: ?T (deleteAround g (fst (a, b)) ps) ⟹
      ?P (deleteAround g (fst (a, b)) ps)
    assume hyp2:  ?T ps ⟹ ?P ps
    have H1: ?P (deleteAround g (fst (a, b)) ps)
      by (rule hyp1, rule isTable-deleteAround) simp
    have H2: ?P ps by (rule hyp2, rule isTable-Cons)

    show ?P ((a,b)#ps)
    proof cases
      assume c: ExcessNotAtRec ps g
        ≤ b + ExcessNotAtRec (deleteAround g a ps) g
      have preSeparated g
       (insert a (set (ExcessNotAtRecList (deleteAround g a ps) g)))
      proof (rule preSeparated-insert)
        from prem show a ∈ set (vertices g) by (auto simp add: isTable-def)
        from H1
        show pS: preSeparated g
            (set (ExcessNotAtRecList (deleteAround g a ps) g))
```

```
          by simp

        fix f assume f ∈ set (facesAt g a)
        then have
        f • a ∉ set [fst p. p ∈ deleteAround g a ps]
          by (auto simp add: facesAt-def deleteAround-eq deleteAround′-def
            removeKeyList-eq split: split-if-asm)
        moreover
        have set (ExcessNotAtRecList (deleteAround g a ps) g)
          ⊆ set [fst p. p ∈ deleteAround g a ps]
          by (rule ExcessNotAtRecList-subset)
        ultimately
        show f • a
          ∉ set (ExcessNotAtRecList (deleteAround g a ps) g)
          by auto
        assume |vertices f| ≤ 4
        have set (vertices f)
          ∩ set [fst p. p ∈ deleteAround g a ps] ⊆ {a}
          by (rule-tac deleteAround-separated[OF mgp fin])
        moreover
        have set (ExcessNotAtRecList (deleteAround g a ps) g)
          ⊆ set [fst p. p ∈ deleteAround g a ps]
           by (rule ExcessNotAtRecList-subset)
        ultimately
        show set (vertices f)
           ∩ set (ExcessNotAtRecList (deleteAround g a ps) g) ⊆ {a}
          by blast
      qed
      with H1 E c show ?thesis by (simp)
    next
      assume ¬ ExcessNotAtRec ps g
        ≤ b + ExcessNotAtRec (deleteAround g a ps) g
      with H2 E show ?thesis by simp
    qed
  qed
qed


lemma isTable-ExcessTable:
  isTable (λv. ExcessAt g v) vs (ExcessTable g vs)
  by  (auto simp add: isTable-def ExcessTable-def ExcessAt-def)

lemma ExcessTable-subset:
  set (map fst (ExcessTable g vs)) ⊆ set vs
  by (induct vs) (auto simp add: ExcessTable-def)

lemma distinct-ExcessNotAtRecList:
  distinct (map fst ps) ⟹ distinct (ExcessNotAtRecList ps g)
    (is ?T ps ⟹ ?P ps)
```

```
proof (induct rule: ExcessNotAtRec.induct)
  show ?P [] by simp
next
  fix a b ps
  assume prem: ?T ((a,b)#ps)
  then have a: a ∉ set (map fst ps) by simp
  assume hyp1: ?T (deleteAround g (fst (a, b)) ps) ⟹
    ?P (deleteAround g (fst (a, b)) ps)
  assume hyp2:  ?T ps ⟹ ?P ps
  from prems have ?T ps by simp
  then have H1: ?P (deleteAround g (fst (a, b)) ps)
   by (rule-tac hyp1) (rule distinct-deleteAround)
  from prem have H2: ?P ps
    by (rule-tac hyp2) simp

  have a ∉ set (ExcessNotAtRecList (deleteAround g a ps) g)
  proof
    assume a ∈ set (ExcessNotAtRecList (deleteAround g a ps) g)
    also have set (ExcessNotAtRecList (deleteAround g a ps) g)
      ⊆ set [fst p. p ∈ deleteAround g a ps]
     by (rule ExcessNotAtRecList-subset)
    also have set (deleteAround g a ps) ⊆ set ps
      by (rule deleteAround-subset)
    then have set [fst p. p ∈ deleteAround g a ps]
      ⊆ set [fst p. p ∈ ps] by auto
    finally have a ∈ set (map fst ps) .
    with a show False by contradiction
  qed
  with H1 H2 show ?P ((a,b)#ps)
    by ( simp add: ExcessNotAtRecList-subset)
qed


consts ExcessTable-cont ::
  (vertex ⇒ nat) ⇒ vertex list ⇒ (vertex × nat) list
primrec
  ExcessTable-cont ExcessAtPG [] = []
  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)

constdefs
  ExcessTable′ :: graph ⇒ vertex list ⇒ (vertex × nat) list
  ExcessTable′ g ≡ ExcessTable-cont (ExcessAt g)
```

```
lemma distinct-ExcessTable-cont:
  distinct vs ⟹
  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: v ∉ 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 ∉ fst ` set (ExcessTable-cont (ExcessAt g) vs)
  proof
    assume v ∈ fst ` set (ExcessTable-cont (ExcessAt g) vs)
    also have fst ` set (ExcessTable-cont (ExcessAt g) vs) ⊆ set vs
      by (induct vs) auto
    finally have  v ∈ 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 ∈ [v∈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 ⟹ distinct [fst p. p ∈ ExcessTable g vs]
  by (simp-all add: ExcessTable-eq ExcessTable′-def distinct-ExcessTable-cont)
```

```
lemma ExcessNotAt-eq:
  minGraphProps g ⟹ final g ⟹
  ∃ V. ExcessNotAt g None
      = ∑v ∈ V ExcessAt g v
   ∧ preSeparated g (set V) ∧ set V ⊆ set (vertices g)
   ∧ distinct V
proof (intro exI conjI)
  assume mgp: minGraphProps g and fin: final g
```

```
  let ?ps = ExcessTable g (vertices g)
  let ?V = ExcessNotAtRecList ?ps g
  let ?vs = vertices g
  let ?E = λv. ExcessAt g v
  have t: isTable ?E ?vs ?ps by (rule isTable-ExcessTable)
  with this show ExcessNotAt g None = ∑ v ∈ ?V ?E v
    by (simp add: ListSum-ExcessNotAtRecList ExcessNotAt-def)

  show preSeparated g (set ?V)
    by(rule preSeparated-ExcessNotAtRecList[OF mgp fin t])

  have set (ExcessNotAtRecList ?ps g) ⊆ set (map fst ?ps)
    by (rule ExcessNotAtRecList-subset)
  also have ... ⊆ set (vertices g) by (rule ExcessTable-subset)
  finally show set ?V ⊆ set (vertices g) .

  show distinct ?V
   by (simp add: distinct-ExcessNotAtRecList distinct-ExcessTable)
qed



lemma excess-eq:
  assumes 6: t + q ≤ 6
  shows excessAtType t q 0 + t ∗ d 3 + q ∗ d 4 = b t q
proof −
 note simps = excessAtType-def squanderVertex-def squanderFace-def squanderTarget-def
  from 6 have q=0 ∨ q=1 ∨ q=2 ∨ q=3 ∨ q=4 ∨ q=5 ∨ q=6 by arith
  then show ?thesis
  proof (elim disjE)
    assume q: q = 0
    with prems show ?thesis by (simp add:simps)
  next
    assume q: q = 1
    with prems show ?thesis by (simp add:simps)
  next
    assume q: q = 2
    with prems show ?thesis by (simp add: simps)
  next
    assume q: q = 3
    with prems show ?thesis  by (simp add: simps)
  next
    assume q: q = 4
    with prems show ?thesis  by (simp add: simps)
  next
    assume q: q = 5
    with prems show ?thesis  by (simp add: simps)
  next
    assume q: q = 6
```

```
    with prems show ?thesis by (simp add: simps)
  qed
qed

lemma excess-eq1:
  ⟦ inv g; final g; tame g; except g v = 0; v ∈ set(vertices g) ⟧ ⟹
   ExcessAt g v + (tri g v) ∗ d 3 + (quad g v) ∗ d 4
   = b (tri g v) (quad g v)
apply(subgoal-tac finalVertex g v)
apply(simp add: ExcessAt-def excess-eq faceCountMax-bound)
apply(auto simp:finalVertex-def plane-final-facesAt)
done
```

preSeparating

```
lemma preSeparated-separating:
assumes pl: inv g and fin: final g and ne: noExceptionals g (set V)
and pS: preSeparated g (set V)
shows separating (set V) (λv. set (facesAt g v))
proof −
  from pS have i: ∀ v∈set V. ∀ f∈set (facesAt g v).
    |vertices f| ≤ 4 ⟶ set (vertices f) ∩ set V = {v}
    by (simp add: preSeparated-def separated3-def)

  show separating (set V) (λv. set (facesAt g v))
  proof (simp add: separating-def, intro ballI impI)
    fix v1 v2 assume v: v1 ∈ set V v2 ∈ set V v1 ≠ v2
    show set (facesAt g v1) ∩ set (facesAt g v2) = {} (is ?P)
    proof (rule ccontr)
      assume ¬ ?P
      then obtain f where f1: f ∈ set (facesAt g v1)
        and f2: f ∈ set (facesAt g v2) by auto
      from v ne have ¬ exceptionalVertex g v1
        ¬ exceptionalVertex g v2
        by (simp-all add: noExceptionals-def)
      with f1 pl fin have l: |vertices f| ≤ 4
        by (simp add: not-exceptional)
      from v f1 l i have set (vertices f) ∩ set V = {v1} by simp
      also from v f2 l i
      have set (vertices f) ∩ set V = {v2} by simp
      finally have v1 = v2 by auto
      then show False by contradiction
    qed
  qed
qed

lemma preSeparated-disj-Union2:
assumes pl: inv g and fin: final g and ne: noExceptionals g (set V)
and pS: preSeparated g (set V) and dist: distinct V
and V-subset: set V ⊆ set (vertices g)
```

shows $(\sum_{v \in V} \sum_{f \in facesAt\ g\ v} (w{::}face \Rightarrow nat)\ f)$
$= \sum_{f \in [f{\in}faces\ g\ .\ \exists\, v \in set\ V.\ f \in set\ (facesAt\ g\ v)]} w\ f$
proof −
have *s*: $separating\ (set\ V)\ (\lambda v.\ set\ (facesAt\ g\ v))$
by (*rule preSeparated-separating*[*OF pl fin ne pS*])
moreover note *dist*
moreover from *pl V-subset*
have *v*: $\bigwedge v.\ v \in set\ V \Longrightarrow distinct\ (facesAt\ g\ v)$
by(*blast intro*:*mgp-dist-facesAt*[*OF inv-mgp*])
moreover
have $distinct\ [f{\in}faces\ g\ .\ \exists\, v \in set\ V.\ f \in set\ (facesAt\ g\ v)]$
by (*intro distinct-filter minGraphProps11'*[*OF inv-mgp*[*OF pl*]])
moreover from *pl* **have** $\{x.\ x \in set\ (faces\ g) \wedge (\exists\, v \in set\ V.\ x \in set\ (facesAt\ g\ v))\} =$
$(\bigcup v{\in}set\ V.\ set\ (facesAt\ g\ v))$
by (*blast intro*:*minGraphProps inv-mgp*)
moreover from *v* **have** $(\sum v{\in}set\ V.\ ListSum\ (facesAt\ g\ v)\ w) = (\sum v{\in}set\ V.\ setsum\ w\ (set\ (facesAt\ g\ v)))$
by (*auto simp add*: *ListSum-conv-setsum*)

ultimately show *?thesis*
apply (*simp add*: *ListSum-conv-setsum*)
apply (*rule setsum-disj-Union*) **by** *simp*+
qed

lemma *squanderFace-distr2*: $inv\ g \Longrightarrow final\ g \Longrightarrow noExceptionals\ g\ (set\ V) \Longrightarrow$
$preSeparated\ g\ (set\ V) \Longrightarrow distinct\ V \Longrightarrow set\ V \subseteq set\ (vertices\ g) \Longrightarrow$
$\sum_{f \in [f{\in}faces\ g.\ \exists\, v \in set\ V.\ f \in set\ (facesAt\ g\ v)]}$
$\mathrm{d}\ |vertices\ f|$
$= \sum_{v \in V} ((tri\ g\ v) * \mathrm{d}\ 3$
$+ (quad\ g\ v) * \mathrm{d}\ 4)$
proof −
assume *pl*: *inv g*
assume *fin*: *final g*
assume *ne*: *noExceptionals g* (*set V*)
assume *preSeparated g* (*set V*) *distinct V* **and** *V-subset*: $set\ V \subseteq set\ (vertices\ g)$
with *pl ne fin* **have**
$\sum_{f \in [f{\in}faces\ g.\ \exists\, v{\in}set\ V.\ f{\in}set\ (facesAt\ g\ v)]} \mathrm{d}\ |vertices\ f|$
$= \sum_{v \in V} \sum_{f \in facesAt\ g\ v} \mathrm{d}\ |vertices\ f|$
by (*simp add*: *preSeparated-disj-Union2*)
also have $\bigwedge v.\ v \in set\ V \Longrightarrow$
$\sum_{f \in facesAt\ g\ v} \mathrm{d}\ |vertices\ f|$
$= (tri\ g\ v) * \mathrm{d}\ 3 + (quad\ g\ v) * \mathrm{d}\ 4$
proof −
fix *v* **assume** *v1*: $v \in set\ V$
with *V-subset* **have** *v*: $v \in set\ (vertices\ g)$ **by** *auto*

with *ne* **have** *d*:

```
    ⋀f. f ∈ set (facesAt g v) ⟹
    |vertices f| = 3 ∨ |vertices f| = 4
  proof −
    fix f assume f: f ∈ set (facesAt g v)
     then have ff: f ∈ set (faces g) by (rule minGraphProps5[OF inv-mgp[OF
pl]])
    with ne f v1 pl fin have |vertices f| ≤ 4
      by (auto simp add: noExceptionals-def not-exceptional)
    moreover from pl ff have 3 ≤ |vertices f| by(rule planeN4)
    ultimately show ?thesis f by arith
  qed

  from d pl v have
    ∑f ∈ facesAt g v d |vertices f|
  = (∑f∈[f ∈ facesAt g v. |vertices f| = 3] d |vertices f| )
  + (∑f∈[f ∈ facesAt g v. |vertices f| = 4] d |vertices f| )
    apply (rule-tac ListSum-disj-union)
    apply (rule distinct-filter) apply simp
    apply (rule distinct-filter) apply simp
    apply simp
    apply force
    apply force
    done
  also have ... = tri g v ∗ d 3 + quad g v ∗ d 4
  proof −
    from pl fin v have ⋀A.[f ∈ facesAt g v. final f ∧ A f]
      = [f ∈ facesAt g v. A f]
      by (rule-tac filter-eqI) (auto simp:plane-final-facesAt)
    with fin show ?thesis by (auto simp add: tri-def quad-def)
  qed
  finally show ∑f ∈ facesAt g v d |vertices f|
  = tri g v ∗ d 3 + quad g v ∗ d 4 .
 qed
 then have ∑v ∈ V ∑f ∈ facesAt g v d |vertices f| =
       ∑v ∈ V (tri g v ∗ d 3 + quad g v ∗ d 4)
   by (rule ListSum-eq)
 finally show ?thesis .
qed


lemma preSeparated-subset:
  V1 ⊆ V2 ⟹ preSeparated g V2 ⟹ preSeparated g V1
proof (simp add: preSeparated-def separated₃-def separated₂-def,
  elim conjE, intro allI impI ballI conjI)
 fix v f
 assume a: v ∈ V1 V1 ⊆ V2 f ∈ set (facesAt g v)
   |vertices f| ≤ 4
   ∀ v∈V2. ∀ f∈set (facesAt g v). |vertices f| ≤ 4 ⟶
```

```
    set (vertices f) ∩ V2 = {v}
  then show set (vertices f) ∩ V1 = {v} by auto
next
  fix v f
  assume a: v ∈ V1 V1 ⊆ V2 f ∈ set (facesAt g v)
    ∀ v∈V2. ∀ f∈set (facesAt g v). f • v ∉ V2
  then have v ∈ V2 by auto
  with a have f • v ∉ V2 by auto
  with a show f • v ∉ V1 by auto
qed

end
```

23 Correctness of Lower Bound for Final Graphs

```
theory LowerBound
imports PlaneProps ScoreProps
begin


theorem total-weight-lowerbound:
 inv g ⟹ final g ⟹ tame g ⟹ admissible w g ⟹
 ∑f ∈ faces g w f < squanderTarget ⟹
 squanderLowerBound g ≤ ∑f ∈ faces g w f
proof −
  assume final: final g and tame: tame g and pl: inv g
  assume admissible: admissible w g
  assume w: ∑f ∈ faces g w f < squanderTarget

  have squanderLowerBound g
     = ExcessNotAt g None + faceSquanderLowerBound g
    by (simp add: squanderLowerBound-def)
```

We expand the definition of *faceSquanderLowerBound*.

```
  also have faceSquanderLowerBound g = ∑f ∈ faces g d |vertices f|
```

We expand the definition of *ExcessNotAt*.

```
  also from ExcessNotAt-eq[OF pl[THEN inv-mgp] final] obtain V
    where eq: ExcessNotAt g None = ∑v ∈ V ExcessAt g v
    and pS:  preSeparated g (set V)
    and V-subset: set V ⊆ set(vertices g)
    and V-distinct: distinct V
```

24 Properties of Tame Graph Enumeration (1)

```
theory GeneratorProps
imports Plane3Props 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′ ∈ set (generatePolygon n v f g) inv g*
f ∈ set(nonFinals g) v ∈ $\mathcal{V}$ f 3 ≤ n
shows *faceSquanderLowerBound g′ ≥ faceSquanderLowerBound g* + d *n*
proof −
from *genPoly-new-face*[*OF prems*(*1*) *inv-mgp*[*OF prems*(*2*)] *prems*(*3*−*5*)] **obtain** *f*
where *f*: *f ∈ set (finals g′) − set(finals g)*
and *size*: |*vertices f*| = *n* **by** *auto*
have *g′*: *g′ ∈ set*(*next-plane0* (*n* − *3*) *g*) **using** *prems*(*5*)
by(*rule-tac in-next-plane0I*[*OF prems*(*1*,*3*−*5*)]) *simp*
note *dist = minGraphProps11′*[*OF inv-mgp*[*OF prems*(*2*)]]
note *inv′ = invariantE*[*OF inv-inv-next-plane0*, *OF g′ prems*(*2*)]
note *dist′ = minGraphProps11′*[*OF inv-mgp*[*OF inv′*]]
note *subset = next-plane0-finals-subset*[*OF g′*]
have *faceSquanderLowerBound g′ ≥*
faceSquanderLowerBound g + d |*vertices f*|
proof(*unfold faceSquanderLowerBound-def*)
have ($\sum_{f \in finals\ g}$ d |*vertices f*|) + d |*vertices f*| =
($\sum$*f∈set*(*finals g*). d |*vertices f*|) + d |*vertices f*|
using *dist* **by**(*simp add*:*finals-def ListSum-conv-setsum*)
also have . . . = ($\sum$*f∈set*(*finals g*) ∪ {*f*}. d |*vertices f*|)
using *f* **by** *simp*
also have . . . ≤ ($\sum$*f∈set*(*finals g′*). d |*vertices f*|)
using *f subset* **by**(*fastsimp intro*!: *setsum-mono3*)
also have . . . = ($\sum_{f \in finals\ g'}$ d |*vertices f*|)
using *dist′* **by**(*simp add*:*finals-def ListSum-conv-setsum*)
finally show ($\sum_{f \in finals\ g}$ d |*vertices f*|) + d |*vertices f*|
≤ $\sum_{f \in finals\ g'}$ d |*vertices f*| **.**
qed
with *size* **show** *?thesis* **by** *blast*
qed

lemma *ExcessTable-empty*:
∀ *x ∈ $\mathcal{V}$ g*. ¬ *finalVertex g x* ⟹ *ExcessTable g* (*vertices g*) = []
by(*simp add*:*ExcessTable-def filter-empty-conv ExcessAt-def*)

constdefs
close :: *graph* ⇒ *vertex* ⇒ *vertex* ⇒ *bool*
close g u v ≡
∃*f ∈ set*(*facesAt g u*). *if* |*vertices f*| = *4 then v = f* • *u* ∨ *v = f* • (*f* • *u*)
else v = f • *u*

lemma *delAround-def*: *deleteAround g u ps* = [*p ∈ ps*. ¬ *close g u* (*fst p*)]
by (*induct ps*) (*auto simp*: *deleteAroundCons close-def*)

```
lemma close-sym: assumes mgp: minGraphProps g and cl: close g u v
shows close g v u
proof –
  obtain f where f: f ∈ set(facesAt g u) and
    if: if |vertices f| = 4 then v = f • u ∨ v = f • (f • u) else v = f • u
    using cl by (unfold close-def) blast
  note uf = minGraphProps6[OF mgp f]
  note distf = minGraphProps3[OF mgp minGraphProps5[OF mgp f]]
  show ?thesis
  proof cases
    assume 4: |vertices f| = 4
    hence v = f • u ∨ v = f • (f • u) using if by simp
    thus ?thesis
    proof
      assume v = f • u
      then obtain f' where f' ∈ set(facesAt g v) f' • v = u
        using mgp-nextVertex-face-ex2[OF mgp f] by blast
      thus ?thesis by(auto simp:close-def)
    next
      assume v: v = f • (f • u)
      hence f • (f • v) = u using quad-next4-id[OF 4 distf uf] by simp
      moreover have f ∈ set(facesAt g v) using v uf
        by(simp add: minGraphProps7[OF mgp minGraphProps5[OF mgp f]])
      ultimately show ?thesis using 4 by(auto simp:close-def)
    qed
  next
    assume |vertices f| ≠ 4
    hence v = f • u using if by simp
    then obtain f' where f' ∈ set(facesAt g v) f' • v = u
      using mgp-nextVertex-face-ex2[OF mgp f] by blast
    thus ?thesis by(auto simp:close-def)
  qed
qed


lemma preSep-conv:
assumes mgp: minGraphProps g
shows preSeparated g V = (∀ u∈V.∀ v∈V. u ≠ v ⟶ ¬ close g u v) (is ?P =
?Q)
proof
  assume preSep: ?P
  show ?Q
  proof(clarify)
    fix u v assume uv: u ∈ V v ∈ V u ≠ v and cl: close g u v
    from cl obtain f where f: f ∈ set(facesAt g u) and
      if: if |vertices f| = 4 then (v = f • u) ∨ (v = f • (f • u))
                                else (v = f • u)
```

```
      by (unfold close-def) blast
    note uf = minGraphProps6[OF mgp f]
    show False
    proof cases
      assume 4: |vertices f| = 4
      hence v = f • u ∨ v = f • (f • u) using if by simp
      thus False
      proof
        assume v = f • u
        thus False using preSep f uv
          by(simp add:preSeparated-def separated₂-def separated₃-def)
      next
        assume v = f • (f • u)
        moreover hence v ∈ 𝒱 f using ‹u ∈ 𝒱 f› by simp
        moreover have |vertices f| ≤ 4 using 4 by arith
        ultimately show False using preSep f uv ‹u ∈ 𝒱 f›
          apply(unfold preSeparated-def separated₂-def separated₃-def)

          apply(subgoal-tac f • (f • u) ∈ 𝒱 f ∩ V)
          prefer 2 apply blast
          by simp
      qed
    next
      assume 4: |vertices f| ≠ 4
      hence v = f • u using if by simp
      thus False using preSep f uv
        by(simp add:preSeparated-def separated₂-def separated₃-def)
    qed
  qed
next
  assume not-cl: ?Q
  show ?P
  proof(simp add:preSeparated-def, rule conjI)
    show separated₂ g V
    proof (clarsimp simp:separated₂-def)
      fix v f assume v ∈ V and f: f ∈ set (facesAt g v) and f • v ∈ V
      thus False using not-cl mgp-facesAt-no-loop[OF mgp f]
        by(fastsimp simp: close-def split:split-if-asm)
    qed
    show separated₃ g V
    proof (clarsimp simp:separated₃-def)
      fix v f
      assume v ∈ V and f: f ∈ set (facesAt g v) and len: |vertices f| ≤ 4
      note distf = minGraphProps3[OF mgp minGraphProps5[OF mgp f]]
      note vf = minGraphProps6[OF mgp f]
      { fix u assume u ∈ 𝒱 f and u ∈ V
        have u = v
        proof cases
          assume 3: |vertices f| = 3
```

```
  hence 𝒱 f = {v, f • v, f • (f • 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:split-if-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 f] by blast
    hence u = v using not-cl ‹u ∈ V› ‹v ∈ V› 3
      by(force simp:close-def edges-face-eq split:split-if-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 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:split-if-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:split-if-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 f] by blast
    hence u = v using not-cl ‹u ∈ V› ‹v ∈ V› 4
      by(force simp:close-def edges-face-eq split:split-if-asm)
  }
```

```
          ultimately show u=v using ‹u ∈ 𝒱 f› by blast
        qed
      }
      thus 𝒱 f ∩ V = {v} using ‹v ∈ V› vf by blast
    qed
  qed
qed


lemma fin-aux: finite B ⟹ finite{f A|A. A ⊆ B ∧ P A}
apply(rule finite-subset[where B = f ‘ Pow B])
 apply blast
apply simp
done

lemma preSep-ne: ∃P ⊆ M. preSeparated g (fst ‘ P)
by(unfold preSeparated-def separated₂-def separated₃-def) blast

lemma ExcessNotAtRec-conv-Max:
assumes mgp: minGraphProps g
shows distinct(map fst ps) ⟹ ExcessNotAtRec ps g =
  Max{ ∑p∈P. snd p |P. P ⊆ set ps ∧ preSeparated g (fst ‘ P)}
  (concl is - = Max(?M ps) is - = Max{- |P. ?S ps P})
proof(induct ps rule: length-induct)
  case (1 ps0)
  note IH = 1(1) and dist = 1(2)
  show ?case
  proof (cases ps0)
    case Nil thus ?thesis by(simp add:Max-def)
  next
    case (Cons p ps)
    let ?ps = deleteAround g (fst p) ps
    have le: |?ps| ≤ |ps| by(simp add:delAround-def)
    have dist': distinct(map fst ?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 ExcessNotAtRec ps0 g = max (Max(?M ps)) (snd p + Max(?M ?ps))
      using Cons IH le dist dist' by(simp)
    also have snd p + Max(?M ?ps) =
      (Max{snd p + (∑p∈P. snd p) |P. ?S ?ps P})
     by(auto simp add:add-Max-commute fin-aux preSep-ne intro!:arg-cong[where
f=Max])
    also have {snd p + (∑p∈P. snd p) |P. ?S ?ps P} =
      {setsum snd (insert p P) |P. ?S ?ps P}
      using dist Cons
      apply (auto simp:delAround-def)
      apply(rule-tac x=P in exI)
```

```
    apply(fastsimp intro!: setsum-insert[THEN trans,symmetric] elim: finite-subset)
     apply(rule-tac x=P in exI)
     apply(fastsimp intro!: setsum-insert[THEN trans] elim: finite-subset)
     done
    also have ... = {setsum snd P |P.
           P ⊆ insert p (set ?ps) ∧ p ∈ P ∧ preSeparated g (fst ‘ P)}
     apply(auto simp add:preSep-conv[OF mgp] delAround-def)
     apply(rule-tac x = insert p P in exI)
     apply simp
     apply(rule conjI) apply blast
     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 ... = {setsum snd P |P.
           P ⊆ insert p (set ps) ∧ p ∈ P ∧ preSeparated g (fst ‘ P)}
     using Cons dist
     apply(auto simp add:preSep-conv[OF mgp] delAround-def)
     apply(rule-tac x = P in exI)
     apply simp
     apply auto
     done
    also have max (Max(?M ps)) (Max ...) = Max(?M ps ∪ {setsum snd P |P.
           P ⊆ insert p (set ps) ∧ p ∈ P ∧ preSeparated g (fst ‘ P)})
     (is - = Max ?U)
    proof −
     have {setsum snd P |P.
           P ⊆ insert p (set ps) ∧ p ∈ P ∧ preSeparated g (fst ‘ P)} ≠ {}
      apply simp
      apply(rule-tac x={p} in exI)
      by(simp add:preSep-conv[OF mgp])
     thus ?thesis by(simp add: Max-Un fin-aux preSep-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 map-compose[symmetric] o-def)



lemma mono-ExcessTab: ⟦g′ ∈ set (next-plane0_p g); inv g ⟧ =⇒
  set(ExcessTable g (vertices g)) ⊆ set(ExcessTable g′ (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*:*split-if-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*:
⟦*g*′ ∈ *set* (*next-plane0*$_p$ *g*); *inv g*; *u* ∈ $\mathcal{V}$ *g*; *finalVertex g u* ⟧ ⟹
close g′ *u v* ⟹ *close g u v*
by(*simp add*:*close-def next-plane0-finalVertex-facesAt-eq*)

lemma *ExcessTab-final*:
p ∈ *set*(*ExcessTable g* (*vertices g*)) ⟹ *finalVertex g* (*fst p*)
by(*clarsimp simp*:*ExcessTable-def image-def ExcessAt-def split*:*split-if-asm*)

lemma *ExcessTab-vertex*:
p ∈ *set*(*ExcessTable g* (*vertices g*)) ⟹ *fst p* ∈ $\mathcal{V}$ *g*
by(*clarsimp simp*:*ExcessTable-def image-def ExcessAt-def split*:*split-if-asm*)

lemma *next-plane0-incr-ExcessNotAt*:
⟦*g*′ ∈ *set* (*next-plane0*$_p$ *g*); *inv g* ⟧ ⟹
ExcessNotAt g None ≤ *ExcessNotAt g*′ *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*])
apply(*rule Max-mono*)
prefer *2* **apply** (*simp add*: *preSep-ne*)
prefer *2* **apply** (*simp add*: *fin-aux*)
apply *auto*
apply(*rule-tac x*=*P* **in** *exI*)
apply *auto*
apply(*simp add*:*preSep-conv*)
apply (*blast intro*:*close-antimono ExcessTab-final ExcessTab-vertex*)
done

lemma *next-plane0-incr-squander-lb*:
⟦*g*′ ∈ *set* (*next-plane0*$_p$ *g*); *inv g* ⟧ ⟹
squanderLowerBound g ≤ *squanderLowerBound g*′
apply(*simp add*:*squanderLowerBound-def*)
apply(*frule* (*1*) *next-plane0-incr-ExcessNotAt*)
apply(*clarsimp simp add*:*next-plane0-def split*:*split-if-asm*)
apply(*drule* (*4*) *genPoly-incr-facesquander-lb*)
apply *arith*
done

lemma *inv-notame*:

```
⟦g' ∈ set (next-plane0_p g); inv g; notame g⟧
 ⟹ notame g'
apply(simp add:notame-def tame_45-def is-tame_7-def del:disj-not1)
apply(erule disjE)
 apply(rule disjI1)
 apply(clarify)
 apply(frule inv-mgp)
 apply(frule (2) next-plane0-incr-degree)
 apply(frule (2) next-plane0-incr-except)
 apply(frule (1) next-plane0-vertices-subset)
 apply(case-tac except g' v = 0)
  apply(rule bexI) prefer 2 apply fast
  apply simp
 apply(rule bexI) prefer 2 apply fast
 apply (simp split:split-if-asm)
apply(frule (1) next-plane0-incr-squander-lb)
apply(arith)
done


lemma inv-inv-notame:
 invariant(λg. inv g ∧ notame g) 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:
 untame (λg. inv g ∧ notame g)
proof(clarsimp simp add: notame-def untame-def tame_45-def is-tame_7-def
                         linorder-not-le linorder-not-less)
  fix g assume final g inv g tame g
    and cases: (∃ v∈𝒱 g. (except g v = 0 ⟶ 6 < degree g v) ∧
                          (0 < except g v ⟶ 5 < degree g v))
               ∨ squanderTarget ≤ squanderLowerBound g
               (is ?A ∨ ?B is (∃ v∈𝒱 g. ?C v) ∨ -)
  from cases show False
  proof
    assume ?A
    then obtain v where v: v ∈𝒱 g ?C v by auto
    show False
    proof cases
      assume except g v = 0
      thus False using ‹tame g› v by(auto simp: tame-def tame_45-def)
    next
      assume except g v ≠ 0
      thus False using ‹tame g› v
        by(auto simp: except-def filter-empty-conv tame-def tame_45-def
```

```
        minGraphProps-facesAt-eq[OF inv-mgp[OF ‹inv g›]] split:split-if-asm)
   qed
  next
   assume ?B
   thus False using total-weight-lowerbound[OF ‹inv g› ‹final g› ‹tame g›]
     ‹tame g›  by(force simp add:tame-def tame_7-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) ⟧
 ⟹ notame 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 is-tame_7-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 ⟧
  ⟹ notame g'
by(fastsimp simp:generatePolygon-def generatePolygonTame-def enum-enumerator
                 notame-def)


declare upt-Suc[simp del]
lemma excess-notame:
 ⟦ inv g; g' ∈ set (next-plane_p g); g' ∉ set (next-tame0 p g) ⟧
       ⟹ notame g'
apply(frule (1) mgp-next-plane0-if-next-plane[OF inv-mgp])
apply(auto simp add:next-tame0-def next-plane-def split:split-if-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)) ∈ 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: ⟦ Seed_p [next-plane p]→* g; final g; tame g ⟧
 ⟹ Seed_p [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 next-tame1-comp:
 ⟦ tame g; final g; Seed_p [next-tame0 p]→* g ⟧
 ⟹ Seed_p [next-tame1_p]→* g
apply (unfold next-tame1-def)
apply(rule make3Fin-complete)
    apply(blast intro: inv-subset[OF inv-inv-next-plane next-tame0-subset-plane])
   apply(rule next-tame0-subset-plane)
  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-tame1: invariant inv next-tame1_p
apply(simp add:next-tame1-def)
apply(rule inv-comp-map[OF inv-inv-next-tame0])
by(blast intro:RTranCl-inv[OF inv-inv-next-plane] mk3Fin-in-RTranCl)

lemma inv-inv-next-tame: invariant inv next-tame_p
apply(simp add:next-tame-def)
apply(rule inv-subset[OF inv-inv-next-tame1])
apply auto
done

lemma mgp-TameEnum: g ∈ TameEnum_p ⟹ minGraphProps g
by (unfold TameEnumP-def)
   (blast intro: RTranCl-inv[OF inv-inv-next-tame] inv-Seed inv-mgp)


end
```

25 Properties of Tame Graph Enumeration (2)

theory *TameEnumProps*
imports *GeneratorProps*
begin

Completeness of filter for final graphs.

lemma *help*: (*EX x*. (*EX y*:*A*. *x* = *f y*) & *P x*) = (*EX y*:*A*. *P*(*f y*))
by *blast*

lemma *tame*$_3$*-is-tame*$_3$: *tame*$_3$ *g* $\Longrightarrow$ *is-tame*$_3$ *g*
apply(*clarsimp simp*: *tame*$_3$*-def is-tame*$_3$*-def nat-number length-Suc-conv*)
apply((*erule allE*)+, *erule impE*, *blast*)
apply(*simp add*: *ok4-def ok42-def tame-quad-def norm-subset-def*
  pr-iso-subseteq-def pr-iso-in-def image-def
  ex-disj-distrib bex-disj-distrib help conj-disj-distribL)
apply(*erule disjE*, *rule disjI1*)
apply(*fastsimp intro*!:*norm-eq-if-face-cong simp*:*tameConf*$_1$*-def*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_2$*-def intro*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_2$*-def dest*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_3$*-def intro*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_3$*-def intro*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_3$*-def intro*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_3$*-def intro*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_4$*-def intro*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp intro*!:*norm-eq-if-face-cong simp*:*tameConf*$_1$*-def*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_2$*-def intro*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_2$*-def dest*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_3$*-def intro*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_3$*-def intro*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_3$*-def intro*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*, *erule disjE*, *rule disjI1*)
apply(*fastsimp simp*: *tameConf*$_3$*-def intro*!:*norm-eq-if-face-cong*)
apply(*rule disjI2*)
apply(*fastsimp simp*: *tameConf*$_4$*-def intro*!:*norm-eq-if-face-cong*)
done

```
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 tame45 g using tame by(auto simp:tame-def)
next
  show tame6 g using tame by(auto simp:tame-def)
next
  show tame8 g using tame by(auto simp:tame-def)
next
  show is-tame3 g using tame by(simp add:tame-def tame3-is-tame3)
next
  from tame obtain w where admissible w g
    and ∑f ∈ faces g w f < squanderTarget by(auto simp:tame-def tame7-def)
  moreover have squanderLowerBound g ≤ ∑f ∈ faces g w f
    by (rule total-weight-lowerbound)
  ultimately show is-tame7 g by(auto simp:is-tame7-def)
qed


lemma next-tame-comp:
 ⟦ tame g; final g; Seedp [next-tame1 p]→* g ⟧
 ⟹ Seedp [next-tamep]→* g
apply (unfold next-tame-def)
apply(rule filter-tame-succs[OF inv-inv-next-tame1])
     apply(simp add:next-tame1-def 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
```

26 Trie (List Version)

```
theory TrieList
imports Main
begin
```

26.1 Association lists

```
consts
 assoc :: ('key ∗ 'val)list ⇒ 'key ⇒ 'val option
 rem-alist :: 'key ⇒ ('key ∗ 'val)list ⇒  ('key ∗ 'val)list
 upd-alist :: 'key ⇒ 'val ⇒ ('key ∗ 'val)list ⇒  ('key ∗ 'val)list

primrec
assoc [] x = None
assoc (p#ps) x = (let (a,b) = p in if a=x then Some b else assoc ps x)
primrec
rem-alist k [] = []
rem-alist k (p#ps) = (if fst p = k then ps else p # rem-alist k ps)
primrec
upd-alist k v [] = [(k,v)]
upd-alist k v (p#ps) = (if fst p = k then (k,v)#ps else p # upd-alist k v ps)

lemma assoc-conv: assoc al x = map-of al x
by(induct al, auto)

lemma map-of-upd-alist: map-of(upd-alist k v al) = (map-of al)(k ↦ v)
apply(induct al)
 apply simp
apply(rule ext)
apply auto
done

lemma rem-alist-id[simp]: k ∉ fst ‘ set al ⟹ rem-alist k al = al
by(induct al, auto)

lemma map-of-rem-distinct-alist: distinct(map fst al) ⟹
  map-of(rem-alist k al) = (map-of al)(k := None)
apply(induct al)
 apply simp
apply clarify
apply(rule ext)
apply auto
apply(erule ssubst)
apply simp
done

lemma map-of-rem-alist[simp]:
 k' ≠ k ⟹ map-of (rem-alist k al) k' = map-of al k'
by(induct al, auto)
```

26.2 Trie

```
datatype ('a,'v)trie = Trie  'v list  ('a ∗ ('a,'v)trie)list

consts values :: ('a,'v)trie ⇒ 'v list
```

```
   alist :: ('a,'v)trie ⇒ ('a * ('a,'v)trie)list
primrec values(Trie vs al) = vs
primrec alist(Trie vs al) = al

consts
 lookup-trie :: ('a,'v)trie ⇒ 'a list ⇒ 'v list
 update-trie :: ('a,'v)trie ⇒ 'a list ⇒ 'v list ⇒ ('a,'v)trie
 insert-trie :: ('a,'v)trie ⇒ 'a list ⇒ 'v      ⇒ ('a,'v)trie

primrec
lookup-trie t [] = values t
lookup-trie t (a#as) = (case assoc (alist t) a of
                         None ⇒ []
                       | Some at ⇒ lookup-trie at as)
primrec
update-trie t []     vs = Trie vs (alist t)
update-trie t (a#as) vs =
   (let tt = (case assoc (alist t) a of
                None ⇒ Trie [] [] | Some at ⇒ at)
    in Trie (values t) ((a,update-trie tt as vs) # rem-alist a (alist t)))

primrec
insert-trie t []     v = Trie (v # values t) (alist t)
insert-trie t (a#as) vs =
   (let tt = (case assoc (alist t) a of
                None ⇒ Trie [] [] | Some at ⇒ at)
    in Trie (values t) ((a,insert-trie tt as vs) # rem-alist a (alist t)))


lemma lookup-empty[simp]: lookup-trie (Trie [] []) as = []
by(case-tac as, simp-all)

theorem lookup-update-trie: !t v bs.
 lookup-trie (update-trie t as vs) bs = (if as=bs then vs else lookup-trie t bs)
apply(induct as)
 apply clarsimp
 apply(case-tac bs)
  apply simp
 apply simp
apply clarsimp
apply(case-tac bs)
 apply (auto simp add:Let-def assoc-conv split:option.split)
done

theorem insert-trie-conv:
!!t. insert-trie t as v = update-trie t as (v#lookup-trie t as)
apply(induct as)
apply (auto simp:Let-def assoc-conv split:option.split)
done
```

```
constdefs
 trie-of-list :: ('b ⇒ 'a list) ⇒ 'b list ⇒ ('a,'b)trie
trie-of-list key == foldl (%t v. insert-trie t (key v) v) (Trie [] [])

lemma in-set-lookup-trie-of-list:
 v ∈ set(lookup-trie (trie-of-list key vs) (key v)) = (v ∈ set vs)
proof -
  have !!t.
 v ∈ set(lookup-trie (foldl (%t v. insert-trie t (key v) v) t vs) (key v)) =
 (v ∈ set vs ∪ set(lookup-trie t (key v)))
    apply(induct vs)
     apply (simp add:trie-of-list-def)
    apply (simp add:trie-of-list-def)
    apply(simp add:insert-trie-conv lookup-update-trie)
    apply blast
    done
  thus ?thesis by(simp add:trie-of-list-def)
qed

lemma in-set-lookup-trie-of-listD:
assumes v ∈ set(lookup-trie (trie-of-list f vs) xs) shows v ∈ set vs
proof -
  have !!t.
  v ∈ set(lookup-trie (foldl (%t v. insert-trie t (f v) v) t vs) xs) ⟹
  v ∈ set vs ∪ set(lookup-trie t xs)
    apply(induct vs)
     apply (simp add:trie-of-list-def)
    apply (simp add:trie-of-list-def)
    apply(force simp:insert-trie-conv lookup-update-trie split:split-if-asm)
    done
  thus ?thesis using prems by(force simp add:trie-of-list-def)
qed

end
```

27 Archive

```
theory Arch
imports Main
uses (arch.ML)
 (Archives/Tri.ML)
 (Archives/Quad.ML)
 (Archives/Pent.ML)
 (Archives/Hex.ML)
 (Archives/Hept.ML)
 (Archives/Oct.ML)
begin
```

The Archive is contained in 6 ML files.

use *Archives/Tri.ML*
use *Archives/Quad.ML*
use *Archives/Pent.ML*
use *Archives/Hex.ML*
use *Archives/Hept.ML*
use *Archives/Oct.ML*

use *arch.ML*
setup ⟪ *add-archconstdefs* ⟫

First the ML values are loaded. Then they are turned into Isabelle definitions of the constants *Tri*, *Quad*, *Pent*, *Hex*, *Hept*, *Oct*, all of type *nat list list list*.

end

28 Comparing Enumeration and Achive

theory *ArchComp*
imports *TameEnum TrieList Arch*
begin

consts *qsort* :: (*'a* ⇒ *'a* => *bool*) ∗ *'a list* ⇒ *'a list*
recdef *qsort measure* (*size o snd*)
  qsort(*le*, []) = []
  qsort(*le*, *x*#*xs*) = *qsort*(*le*, [*y*:*xs* . ~ *le x y*]) @ [*x*] @
      qsort(*le*, [*y*:*xs* . *le x y*])
(**hints** *recdef-simp*: *length-filter-le*[*THEN le-less-trans*])

constdefs
nof-vertices :: *'a fgraph* ⇒ *nat*
nof-vertices == *length o remdups o concat*

fgraph :: *graph* ⇒ *nat fgraph*
fgraph g == *map vertices* (*faces g*)

hash :: *nat fgraph* ⇒ *nat list*
hash fs == *let n* = *nof-vertices fs in*
  [*n*, *size fs*] @
  qsort (%*x y*. *y* < *x*, *map* (%*i*. *foldl* (*op* +) *0* (*map size* [*f*:*fs*. *i mem f*]))
      [*0*..<*n*])

consts
enum-finals :: (*graph* ⇒ *graph list*) ⇒ *nat* ⇒ *graph list* ⇒ *nat fgraph list*
    ⇒ *nat fgraph list option*
primrec

```
enum-finals succs 0 todo fins = None
enum-finals succs (Suc n) todo fins =
 (if todo = [] then Some fins
  else let g = hd todo; todo′ = tl todo in
    if final g then enum-finals succs n todo′ (fgraph g # fins)
    else enum-finals succs n (succs g @ todo′) fins)
```

constdefs

```
 tameEnum :: nat ⇒ nat ⇒ nat fgraph list option
tameEnum p n == enum-finals (next-tame p) n [Seed p] []

 diff2 :: nat fgraph list ⇒ nat fgraph list ⇒
          (nat * nat fgraph) list * (nat * nat fgraph)list
diff2 fgs ags ==
 let xfgs = map (%fs. (nof-vertices fs, fs)) fgs;
     xags = map (%fs. (nof-vertices fs, fs)) ags in
 (filter (%xfg. ~list-ex (iso-test xfg) xags) xfgs,
  filter (%xag. ~list-ex (iso-test xag) xfgs) xags)

 same :: nat fgraph list option ⇒ nat fgraph list ⇒ bool
same fgso ags == case fgso of None ⇒ False | Some fgs ⇒
 let niso-test = (%fs1 fs2. iso-test (nof-vertices fs1,fs1)
                                     (nof-vertices fs2,fs2));
     tfgs = trie-of-list hash fgs;
     tags = trie-of-list hash ags in
 (list-all (%fg. list-ex (niso-test fg) (lookup-trie tags (hash fg))) fgs &
  list-all (%ag. list-ex (niso-test ag) (lookup-trie tfgs (hash ag))) ags)
```

constdefs

```
 pre-iso-test :: vertex fgraph ⇒ bool
pre-iso-test Fs ≡
 [] ∉ set Fs ∧ (∀F∈set Fs. distinct F) ∧ distinct (map rotate-min Fs)
```

lemma *pre-iso-test3*: $\forall g \in$ *set Tri. pre-iso-test g*
by *evaluation*

lemma *pre-iso-test4*: $\forall g \in$ *set Quad. pre-iso-test g*
by *evaluation*

lemma *pre-iso-test5*: $\forall g \in$ *set Pent. pre-iso-test g*
by *evaluation*

lemma *pre-iso-test6*: $\forall g \in$ *set Hex. pre-iso-test g*
by *evaluation*

lemma *pre-iso-test7*: $\forall g \in$ *set Hept. pre-iso-test g*
by *evaluation*

lemma *pre-iso-test8*: $\forall g \in$ *set Oct. pre-iso-test g*

by *evaluation*

ML*set Toplevel.timing*

lemma *same3*: *same* (*tameEnum 0 800000*) *Tri*
by *evaluation*

lemma *same4*: *same* (*tameEnum 1 8000000*) *Quad*
by *evaluation*

lemma *same5*: *same* (*tameEnum 2 20000000*) *Pent*
by *evaluation*

lemma *same6*: *same* (*tameEnum 3 4000000*) *Hex*
by *evaluation*

lemma *same7*: *same* (*tameEnum 4 1000000*) *Hept*
by *evaluation*

lemma *same8*: *same* (*tameEnum 5 2000000*) *Oct*
by *evaluation*

ML*reset Toplevel.timing*

end

29 Completeness of Archive Test

theory *ArchCompProps*
imports *TameEnumProps ArchComp*
begin

corollary *iso-test-correct*:
$[\![$ *pre-iso-test* Fs_1; *pre-iso-test* Fs_2 $]\!] \Longrightarrow$
iso-test (*nof-vertices* Fs_1,Fs_1) (*nof-vertices* Fs_2,Fs_2) = ($Fs_1 \simeq Fs_2$)
by(*simp add*:*pre-iso-test-def iso-correct inj-on-rotate-min-iff*[*symmetric*]
distinct-map nof-vertices-def length-remdups-concat)

lemma *same-imp-iso-subset*:
assumes *pre1*: $\bigwedge$*gs g*. *gsopt* = *Some gs* $\Longrightarrow$ *g* $\in$ *set gs* $\Longrightarrow$ *pre-iso-test g*
and *pre2*: $\bigwedge$*g*. *g* $\in$ *set arch* $\Longrightarrow$ *pre-iso-test g*
and *same*: *same gsopt arch*
shows $\exists$ *gs*. *gsopt* = *Some gs* $\wedge$ *set gs* $\subseteq_{\simeq}$ *set arch*
proof −

```
  obtain gs where [simp]: gsopt = Some gs and test: ⋀g. g ∈ set gs ⟹
    ∃ h ∈ set arch. iso-test (nof-vertices g,g) (nof-vertices h,h)
    using same by(force simp:same-def split:option.splits
                        dest: in-set-lookup-trie-of-listD)
  have set gs ⊆≃ set arch
  proof (auto simp:iso-subseteq-def iso-in-def)
    fix g assume g: g∈set gs
    obtain h where h: h ∈ set arch and
      test: iso-test (nof-vertices g,g) (nof-vertices h,h)
      using test[OF g] by blast
    thus ∃ h∈set arch. g ≃ h
      using h pre1[OF - g] pre2[OF h] by (auto simp:iso-test-correct)
  qed
  thus ?thesis by auto
qed

lemma enum-finals-tree:
∀ g. final g ⟶ next g = [] ⟹ enum-finals next n todo Fs0 = Some Fs ⟹
  set Fs = set Fs0 ∪ fgraph ‘ {h. ∃ g ∈ set todo. g [next]→* h ∧ final h}
apply(induct n fixing: todo Fs0) apply simp
apply (clarsimp simp:image-def neq-Nil-conv split:split-if-asm)
 apply(rule equalityI)
  apply (blast intro:RTranCl.refl)
 apply(rule)
 apply simp
 apply(erule disjE) apply blast
 apply (erule exE conjE)+
 apply(erule disjE) apply (fastsimp elim:RTranCl-elim)
 apply(blast)
apply(rule equalityI)
 apply (blast intro:succs)
apply(rule)
apply simp
apply(erule disjE) apply blast
apply (erule exE conjE)+
apply(erule disjE) apply (fastsimp elim:RTranCl-elim)
apply(blast)
done

lemma next-tame-of-final: ∀ g. final g ⟶ next-tamep g = []
by(auto simp: next-tame-def next-tame1-def next-tame0-def
              nonFinals-def filter-empty-conv finalGraph-def)

lemma tameEnum-TameEnum: tameEnum p n = Some Fs ⟹ set Fs = fgraph ‘
TameEnump
by(simp add: tameEnum-def TameEnumP-def enum-finals-tree[OF next-tame-of-final])

lemma mgp-pre-iso-test: minGraphProps g ⟹ 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 map-compose[*symmetric*] *o-def*)
done

theorem *combine-evals*:
$\forall\, g \in$ *set arch*. *pre-iso-test g* $\Longrightarrow$ *same* (*tameEnum p n*) *arch*
 $\Longrightarrow$ *fgraph* ‘ *TameEnum*$_p$ $\subseteq_{\simeq}$ *set arch*
apply(*subgoal-tac* $\exists$ *gs*. *tameEnum p n* = *Some gs* $\wedge$ *set gs* $\subseteq_{\simeq}$ *set arch*)
 apply(*fastsimp simp*: *image-def dest*: *tameEnum-TameEnum*)
apply(*fastsimp intro*!: *same-imp-iso-subset simp*: *image-def*
 dest: *tameEnum-TameEnum mgp-TameEnum mgp-pre-iso-test*)
done

end

30 Combining All Completeness Proofs

theory *Completeness*
imports *ArchCompProps*
begin

constdefs
 Archive :: *vertex fgraph set*
Archive $\equiv$
 set(*Tri* @ *Quad* @ *Pent* @ *Hex* @ *Hept* @ *Oct*)

theorem *TameEnum-Archive*: *fgraph* ‘ *TameEnum* $\subseteq_{\simeq}$ *Archive*
using *combine-evals*[*OF pre-iso-test3 same3*]
  combine-evals[*OF pre-iso-test4 same4*]
  combine-evals[*OF pre-iso-test5 same5*]
  combine-evals[*OF pre-iso-test6 same6*]
  combine-evals[*OF pre-iso-test7 same7*]
  combine-evals[*OF pre-iso-test8 same8*]
apply(*clarsimp simp*:*TameEnum-def Archive-def image-def iso-subseteq-def*
        iso-in-def nat-number le-Suc-eq)
apply *fast*
done

lemma *TameEnum-comp*:
assumes *pg*: *Seed*$_p$ [*next-plane*$_p$]$\rightarrow*$ *g* **and** *final g* **and** *tame g*
shows *Seed*$_p$ [*next-tame*$_p$]$\rightarrow*$ *g*
proof −
 from *prems* **have** *Seed*$_p$ [*next-tame0 p*]$\rightarrow*$ *g* **by**(*rule next-tame0-comp*)

```
  with prems have Seed_p [next-tame1 p]→* g by(blast intro: next-tame1-comp)
  with prems show Seed_p [next-tame_p]→* g by(blast intro: next-tame-comp)
qed

lemma Seed-max-final-ex:
 ∃f∈set (finals (Seed p)). |vertices f| = maxGon p
  by (simp add: Seed-def graph-max-final-ex)

lemma max-face-ex: assumes a: Seed_p [next-plane0_p]→* g
shows ∃f ∈ set (finals g). |vertices f| = maxGon p
proof −
  show ?thesis
  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∈set (finals g) and |vertices f| = maxGon p
      by auto
    moreover have f∈set (finals g') by (rule next-plane0-finals-incr)
    ultimately show ?case by auto
  qed
qed


lemma tame5:
assumes g: Seed_p [next-plane0_p]→* g and final g and tame g
shows p ≤ 5
proof −
  from ‹tame g› have bound: ∀f ∈ ℱ g. |vertices f| ≤ 8
   by (simp add: tame-def tame_1-def)
  show p ≤ 5
  proof (rule ccontr)
    assume 6: ¬ p ≤ 5
    obtain f where f ∈ set (finals g) & |vertices f| = p+3
      using max-face-ex[OF g] by auto
    also from ‹final g› have set (finals g) = set (faces g) by simp
    finally have f ∈ ℱ g & 8 < |vertices f| using 6 by arith
    with bound show False by auto
  qed
qed


theorem completeness:
assumes g ∈ PlaneGraphs and tame g shows fgraph g ∈≃ Archive
proof −
  from ‹g ∈ PlaneGraphs› obtain p where g1: Seed_p [next-plane_p]→* g
    and final g
    by(auto simp:PlaneGraphs-def PlaneGraphsP-def)
```

```
  have Seed_p [next-plane0_p]→* 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 ≤ 5 by(blast intro:tame5)
  with g1 ‹tame g› ‹final g› show ?thesis using TameEnum-Archive
    by(simp add: iso-subseteq-def TameEnum-def TameEnumP-def)
      (blast intro: TameEnum-comp)
qed


end
```