Tame Plane Graphs

Gertrud Bauer and Tobias Nipkow

February 22, 2013

Abstract

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

Contents

1 Basic Functions Old and New

theory *ListAux*
imports *Main*
begin

declare *Let-def*[*simp*]

1.1 HOL

lemma *pairD*: $(a,b) = p \Longrightarrow a = fst\ p \wedge b = snd\ p$
by *auto*

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

definition *enum* :: *nat* ⇒ *nat set* **where**
[*code del*]: *enum n* = {..<*n*}

declare *enum-def* [*symmetric*, *code-unfold*]

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

1.2 Lists

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

1.2.1 *length*

notation *length* (|-|)

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

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

1.2.2 *filter*

lemma *filter-emptyE*[*dest*]: (*filter P xs* = []) $\Longrightarrow$ $x \in set\ xs \Longrightarrow \neg\ P\ x$
by (*simp add*: *filter-empty-conv*)

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

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

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

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

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

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

by (*simp add*: *length-filter-le*)
have *hyp*: *length* (*filter P* (*x* # *xs*)) = *length* (*x* # *xs*) **by** *fact*
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*) **by** *fact*
ultimately show *?case* **by** (*auto dest*: *Cons*(*1*))
qed

lemma [*simp*]: [*f x*. *x* <− *xs*, *P x*] = [*f x*. *x* <− [*x* ← *xs*. *P x*]]
by (*induct xs*) *auto*

1.2.3 *concat*

syntax (*xsymbols*)
-concat :: *idt* => *'a list* => *'a list* ⇒ *'a list* (⨆-∈ - - *10*)
translations ⨆*x*∈*xs f* == *CONST concat* [*f*. *x* <− *xs*]

1.2.4 List product

definition *listProd1* :: *'a* ⇒ *'b list* ⇒ (*'a* × *'b*) *list* **where**
listProd1 a bs ≡ [(*a*,*b*). *b* <− *bs*]

definition *listProd* :: *'a list* ⇒ *'b list* ⇒ (*'a* × *'b*) *list* (**infix** × *50*) **where**
as × *bs* ≡ ⨆ *a* ∈ *as listProd1 a bs*

lemma *set* (*xs* × *ys*) = (*set xs*) × (*set ys*)
by (*auto simp*: *listProd-def listProd1-def*)

1.2.5 Minimum and maximum

primrec *minimal*:: (*'a* ⇒ *nat*) ⇒ *'a list* ⇒ *'a* **where**
minimal m (*x*#*xs*) =
(*if xs*=[] *then x else*
let mxs = *minimal m xs in*
if m x ≤ *m mxs then x else mxs*)

lemma *minimal-in-set*[*simp*]: *xs* ≠ [] ⟹ *minimal f xs* : *set xs*
by(*induct xs*) *auto*

primrec *min-list* :: *nat list* ⇒ *nat* **where**
min-list (*x*#*xs*) = (*if xs*=[] *then x else min x* (*min-list xs*))

primrec *max-list* :: *nat list* ⇒ *nat* **where**
max-list (*x*#*xs*) = (*if xs*=[] *then x else max x* (*max-list xs*))

lemma *min-list-conv-Min*[*simp*]:
xs ≠ [] ⟹ *min-list xs* = *Min* (*set xs*)

by (*induct xs*) *auto*

lemma *max-list-conv-Max*[*simp*]:
xs ≠ [] ⟹ *max-list xs* = *Max* (*set xs*)
by (*induct xs*) *auto*

1.2.6 replace

primrec *replace* :: *'a* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* **where**
replace x ys [] = []
| *replace x ys* (*z*#*zs*) =
(*if z* = *x then ys* @ *zs else z* # (*replace x ys zs*))

primrec *mapAt* :: *nat list* ⇒ (*'a* ⇒ *'a*) ⇒ (*'a list* ⇒ *'a list*) **where**
mapAt [] *f as* = *as*
| *mapAt* (*n*#*ns*) *f as* =
(*if n* < |*as*| *then mapAt ns f* (*as*[*n*:= *f* (*as*!*n*)])
else mapAt ns f as)

lemma *length-mapAt*[*simp*]: !!*xs*. *length*(*mapAt vs f xs*) = *length xs*
by(*induct vs*) *auto*

lemma *length-replace1*[*simp*]: *length*(*replace x* [*y*] *xs*) = *length xs*
by(*induct xs*) *simp-all*

lemma *replace-id*[*simp*]: *replace x* [*x*] *xs* = *xs*
by(*induct xs*) *simp-all*

lemma *len-replace-ge-same*:
length ys ≥ *1* ⟹ *length*(*replace x ys xs*) ≥ *length xs*
by (*induct xs*) *auto*

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

lemma *replace2*:
$f' \notin set\ ls \Longrightarrow replace\ f'\ fs\ ls\ = ls$
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*]:
$f' \in set\ ls \Longrightarrow f \in set\ fs \Longrightarrow f \in set\ (replace\ f'\ fs\ ls)$
by (*induct ls*) *auto*

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

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

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

lemma *distinct-replace*:
$distinct\ fs \Longrightarrow distinct\ newFs \Longrightarrow set\ fs \cap set\ newFs \subseteq \{oldF\} \Longrightarrow$
$distinct\ (replace\ oldF\ newFs\ fs)$
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 fastforce
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

1.2.7 *distinct*

lemma *dist-at1*: $\bigwedge$ *c vs*. *distinct vs* $\Longrightarrow$ *vs* = *a* @ *r* # *b* $\Longrightarrow$ *vs* = *c* @ *r* # *d* $\Longrightarrow$ *a* = *c*
proof (*induct a*)
case *Nil*
assume *dist*: *distinct vs* **and** *vs1*: *vs* = [] @ *r* # *b* **and** *vs2*: *vs* = *c* @ *r* # *d*
from *dist vs2* **have** *rc*: *r* $\notin$ *set c* **by** *auto*
from *vs1 vs2* **have** *c* @ *r* # *d* = *r* # *b* **by** *auto*
then have *hd* (*c* @ *r* # *d*) = *r* **by** *auto*
then have *c* $\neq$ [] $\Longrightarrow$ *hd c* = *r* **by** *auto*
then have *c* $\neq$ [] $\Longrightarrow$ *r* $\in$ *set c* **by** (*induct c*) *auto*
with *rc* **have** *c*: *c* = [] **by** *auto*
then show *?case* **by** *auto*
next
case (*Cons x xs*) **then show** *?case* **by** (*induct c*) *auto*
qed

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

lemma *dist-at2*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *r* # *b* $\Longrightarrow$ *vs* = *c* @ *r* # *d* $\Longrightarrow$ *b* = *d*
proof −
assume *dist*: *distinct vs* **and** *vs1*: *vs* = *a* @ *r* # *b* **and** *vs2*: *vs* = *c* @ *r* # *d*
then have *a* = *c* $\wedge$ *b* = *d* **by** (*rule-tac dist-at*) *auto*
then show *?thesis* **by** *auto*
qed

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

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

lemma *distinct-hd-not-cons*: *distinct vs* $\Longrightarrow$ $\exists$ *as bs*. *vs* = *as* @ *x* # *hd vs* # *bs* $\Longrightarrow$ *False*
proof −
assume *d*: *distinct vs* **and** *ex*: $\exists$ *as bs*. *vs* = *as* @ *x* # *hd vs* # *bs*
from *ex* **have** *vsne*: *vs* $\neq$ [] **by** *auto*

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

1.2.8 Misc

lemma *drop-last-in*: *!!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

1.2.9 *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*: $n > 0 \Longrightarrow m > 0 \Longrightarrow$
$rotate\ n\ ls = rotate\ m\ ms \Longrightarrow rotate\ (n - 1)\ ls = rotate\ (m - 1)\ ms$
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'*: $n > 0 \Longrightarrow rotate\ n\ ls = ms \Longrightarrow$
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-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 (*cases ls*) **apply** *auto*
by (*simp add*: *nth-append*)

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

lemma *nth-rotate1*: *0 < length ls ⟹ 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 $z \equiv n + m$
then have $n + Suc\ m = Suc\ (z)$ **by** *auto*
with *Suc* **have** *r1*: *ls* ! ((*Suc z*) *mod length ls*) = *rotate1 ls* ! (*z mod length ls*)
by (*rule-tac nth-rotate1*)
from *Suc* **have** $0 < length\ (rotate1\ ls)$ **by** *auto*
then have (*rotate1 ls*) ! ((*n* + *m*) *mod length* (*rotate1 ls*))
= *rotate m* (*rotate1 ls*) ! (*n mod length* (*rotate1 ls*)) **by** (*rule Suc*)
with *r1 z-def* **have** *ls* ! ((*n* + *Suc m*) *mod length ls*)
= *rotate m* (*rotate1 ls*) ! (*n mod length* (*rotate1 ls*)) **by** *auto*
then show *?case* **by** *auto*
qed

1.3 *splitAt*

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

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

1.3.1 *splitAtRec*

lemma *splitAtRec-conv*: !!*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*:

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

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

1.3.2 *splitAt*

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

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

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

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

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

1.3.3 Sets

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

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

lemma *splitAt-in-fst*[*dest*]: *v* $\in$ *set* (*fst* (*splitAt ram vs*)) $\Longrightarrow$ *v* $\in$ *set vs*
proof (*cases ram* $\in$ *set vs*)
assume *v*: *v* $\in$ *set* (*fst* (*splitAt ram vs*))
def *a* $\equiv$ *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-aux*:
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-fst-aux*[*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-aux*)
qed

lemma *splitAt-distinct-snd-aux*[*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-aux*)

qed

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

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

lemma *splitAt-distinct-ram-fst*[*intro*]:
distinct vs $\Longrightarrow$ *ram* $\notin$ *set* (*fst* (*splitAt ram vs*))
apply (*case-tac ram* $\in$ *set vs*) **apply** (*drule-tac splitAt-ram*)
apply (*rule distinct-split1*) **by** (*force dest*: *splitAt-in-fst*)+

lemma *splitAt-distinct-ram-snd*[*intro*]:
distinct vs $\Longrightarrow$ *ram* $\notin$ *set* (*snd* (*splitAt ram vs*))
apply (*case-tac ram* $\in$ *set vs*) **apply** (*drule-tac splitAt-ram*)
apply (*rule distinct-split2*) **by** (*force dest*: *splitAt-in-fst*)+

lemma *splitAt-1*[*simp*]:
splitAt ram [] = ([],[]) **by** (*simp add*: *splitAt-def*)

lemma *splitAt-2*:
v $\in$ *set vs* $\Longrightarrow$ $(a,b) = splitAt\ ram\ vs \Longrightarrow v \in set\ a \vee v \in set\ b \vee v = ram$
apply (*cases ram* $\in$ *set vs*)
by (*auto dest*: *splitAt-split simp*: *splitAt-no-ram*)

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

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

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

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

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

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

1.3.5 *splitAt* composition

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

lemma *splitAt-elements*: $ram1 \in set\ vs \Longrightarrow ram2 \in set\ vs \Longrightarrow ram2 \in set(\ fst\ (splitAt\ ram1\ vs)) \vee ram2 \in set\ [ram1] \vee\ ram2 \in set(\ snd\ (splitAt\ ram1\ vs))$
proof −
assume *r1*: $ram1 \in set\ vs$ **and** *r2*: $ram2 \in set\ vs$
then have $ram2 \in set(\ fst\ (splitAt\ ram1\ vs)\ @\ [ram1]) \vee\ ram2 \in set(\ snd\ (splitAt\ ram1\ vs))$
apply (*rule-tac set-help*)
apply (*drule-tac splitAt-ram*) **by** *auto*
then show *?thesis* **by** *auto*
qed

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

lemma *splitAt-dist-ram*: $distinct\ vs \Longrightarrow$
$vs = a\ @\ ram\ \#\ b \Longrightarrow (a,b) = splitAt\ ram\ vs$
proof −
assume *dist*: $distinct\ vs$ **and** *vs*: $vs = a\ @\ ram\ \#\ b$
from *vs* **have** *r*:$ram \in set\ vs$ **by** *auto*
with *dist vs* **have** *fst* $(splitAt\ ram\ vs) = a$ **apply** (*drule-tac splitAt-ram*) **by** (*rule-tac dist-at1*) *auto*
then have *first*:$a = fst\ (splitAt\ ram\ vs)$ **by** *auto*
from *r dist* **have** *second*: $b = snd\ (splitAt\ ram\ vs)$ **apply** (*drule-tac splitAt-ram*) **apply** (*rule dist-at2*) **apply** *simp*
by (*auto simp*: *vs*)
show *?thesis* **by** (*auto simp*: *first second*)
qed

lemma *distinct-unique1*: $distinct\ vs \Longrightarrow ram \in set\ vs \Longrightarrow EX!\ s.\ vs = (fst\ s)\ @\ ram\ \#\ (snd\ s)$
proof
assume *d*: $distinct\ vs$ **and** *r*: $ram \in set\ vs$
def $s\ \equiv splitAt\ ram\ vs$ **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*: $distinct\ vs \Longrightarrow vs = a\ @\ ram1\ \#\ b\ @\ ram2\ \#\ c \Longrightarrow$
$(a\ @\ ram1\ \#\ b,\ c) = splitAt\ ram2\ vs$
by (*auto intro*: *splitAt-dist-ram*)

lemma *splitAt-dist-ram20*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* $\Longrightarrow$
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 $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c*
$\Longrightarrow$ (*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 $\Longrightarrow$ *x* ∈ *set xs* $\Longrightarrow$
fst(*splitAt x* (*rev xs*)) = *rev*(*snd*(*splitAt x xs*))
by(*simp add*:*splitAt-conv takeWhile-neq-rev*)

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

lemma *splitAt-take*[*simp*]: *distinct ls* $\Longrightarrow$ *i* < *length ls* $\Longrightarrow$ *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*)
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*]
by *simp*

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*: $r \notin set\ a\ r \notin set\ b$ **and** *vs1*: $vs = a\ @\ r\ \#\ b$ **and** *vs2*: $vs = c\ @\ r\ \#\ d$
from *vs2 vs1 dist* **have** $a = c$ **by** (*rule local-help1*)
with *dist vs1 vs2* **show** *?thesis* **by** *simp*
qed

lemma *local-help'*: $a\ @\ r\ \#\ b = c\ @\ r\ \#\ d \Longrightarrow r \notin set\ a \Longrightarrow r \notin set\ b \Longrightarrow a = c \wedge b = d$
by (*rule local-help*) *auto*

lemma *splitAt-simp1*: $ram \notin set\ a \Longrightarrow ram \notin set\ b \Longrightarrow fst\ (splitAt\ ram\ (a\ @\ ram\ \#\ b)) = a$
proof −
assume *ramab*: $ram \notin set\ a\ \ ram \notin set\ b$
def $vs \equiv a\ @\ ram\ \#\ b$
then have *vs*: $vs = a\ @\ ram\ \#\ b$ **by** *auto*
then have $ram \in set\ vs$ **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 *help'''-in*: $\bigwedge xs.\ ram \in set\ b \Longrightarrow fst\ (splitAtRec\ ram\ xs\ b) = xs\ @\ fst\ (splitAtRec\ ram\ []\ b)$
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*: $\bigwedge xs.\ ram \notin\ set\ b \Longrightarrow fst\ (splitAtRec\ ram\ xs\ b) = xs\ @\ fst\ (splitAtRec\ ram\ []\ b)$
proof (*induct b*)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons b bs*)

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

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

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

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

lemma *splitAt-simpD*[*simp*]: $\bigwedge$ *a*. *ram* $\neq$ *a* $\Longrightarrow$ *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(*case-tac xs*) **apply** *clarsimp+*
apply(*erule disjE*) **apply** *simp*
apply(*drule split-list*)
apply *clarsimp*
done

1.4 *between*

definition *between* :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *'a list* **where**
between vs ram_1 ram_2 ≡
let (pre_1, $post_1$) = *splitAt* ram_1 *vs in*
if ram_2 ∈ *set* $post_1$
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 ∈ *set*(*between xs a b*) ⟹ *x* ∈ *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 ∉ *set xs* ⟹ *x* ∉ *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

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

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

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

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

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

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

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

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

theory *Quasi-Order*
imports *Main*
begin

locale *quasi-order* =
fixes *qle* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\preceq$ *60*)
assumes *qle-refl*[*iff*]: $x \preceq x$
and *qle-trans*: $x \preceq y \Longrightarrow y \preceq z \Longrightarrow x \preceq z$
begin

definition *in-qle* :: $'a \Rightarrow 'a\ set \Rightarrow bool$ (**infix** $\in_\preceq$ *60*) **where**
$x \in_\preceq M \equiv \exists y \in M.\ x \preceq y$

definition *subseteq-qle* :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$ (**infix** $\subseteq_\preceq$ *60*) **where**
$M \subseteq_\preceq N \equiv \forall x \in M.\ x \in_\preceq N$

definition *seteq-qle* :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$ (**infix** $=_\preceq$ *60*) **where**
$M =_\preceq N \ \equiv \ M \subseteq_\preceq N \wedge N \subseteq_\preceq M$

lemmas *defs* = *in-qle-def subseteq-qle-def seteq-qle-def*

lemma *subseteq-qle-refl*[*simp*]: $M \subseteq_\preceq M$
by(*auto simp add*: *subseteq-qle-def in-qle-def*)

lemma *subseteq-qle-trans*: $A \subseteq_\preceq B \Longrightarrow B \subseteq_\preceq C \Longrightarrow A \subseteq_\preceq C$
by (*simp add*: *subseteq-qle-def in-qle-def*) (*metis qle-trans*)

lemma *empty-subseteq-qle*[*simp*]: $\{\} \subseteq_\preceq A$
by (*simp add*: *subseteq-qle-def*)

lemma *subseteq-qleI2*: $(!!x.\ x \in M \Longrightarrow EX\ y : N.\ x \preceq y) \Longrightarrow M \subseteq_\preceq N$
by (*auto simp add*: *subseteq-qle-def in-qle-def*)

lemma *subseteq-qleD2*: $M \subseteq_\preceq N \Longrightarrow x \in M \Longrightarrow EX\ y : N.\ x \preceq y$
by (*auto simp add*: *subseteq-qle-def in-qle-def*)

lemma *seteq-qle-refl*[*iff*]: $A =_\preceq A$
by (*simp add*: *seteq-qle-def*)

lemma *seteq-qle-trans*: $A =_\preceq B \Longrightarrow B =_\preceq C \Longrightarrow A =_\preceq C$
by (*simp add*: *seteq-qle-def*) (*metis subseteq-qle-trans*)

end

end

2 Isomorphisms Between Plane Graphs

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

lemma *image-image-id-if*[*simp*]: $(!!x.\ f(f\ x) = x) \Longrightarrow f\ `\ f\ `\ M = M$
by (*auto simp*: *image-iff*)

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

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

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

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

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

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

type-synonym $'a$ *Fgraph* = $'a$ *list set*
type-synonym $'a$ *fgraph* = $'a$ *list list*

2.1 Equivalence of faces

Two faces are equivalent modulo rotation:

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

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

lemma *congs-sym*: **assumes** *A*: $(xs::'a\ list) \cong ys$ **shows** $ys \cong xs$
proof (*simp add*:*congs-def*)
let *?l* = *length xs*

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

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

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

lemma *congs-distinct*:
$F_1 \cong F_2 \Longrightarrow$ *distinct* F_2 = *distinct* F_1
by (*auto simp*: *congs-def*)

lemma *congs-length*:
$F_1 \cong F_2 \Longrightarrow$ *length* F_2 = *length* F_1
by (*auto simp*: *congs-def*)

lemma *congs-pres-nodes*: $F_1 \cong F_2 \Longrightarrow$ *set* F_1 = *set* F_2
by(*clarsimp simp*:*congs-def*)

lemma *congs-map*:
$F_1 \cong F_2 \Longrightarrow$ *map f* $F_1 \cong$ *map f* F_2
by (*auto simp*: *congs-def rotate-map*)

lemma *congs-map-eq-iff*:
inj-on f (*set xs* ∪ *set ys*) ⟹ (*map f xs* ≅ *map f ys*) = (*xs* ≅ *ys*)
apply(*simp add*:*congs-def*)
apply(*rule iffI*)
apply(*clarsimp simp*: *rotate-map*)
apply(*drule map-inj-on*)
apply(*simp add*:*Un-commute*)
apply (*fastforce*)
apply *clarsimp*
apply(*fastforce simp*: *rotate-map*)
done

lemma *list-cong-rev-iff*[*simp*]:
(*rev xs* ≅ *rev ys*) = (*xs* ≅ *ys*)
apply(*simp add*:*congs-def rotate-rev*)
apply(*rule iffI*)
apply *fast*
apply *clarify*
apply(*cases length xs* = *0*)
apply *simp*
apply(*case-tac n mod length xs* = *0*)
apply(*rule-tac x* = *n* **in** *exI*)
apply *simp*
apply(*subst rotate-conv-mod*)
apply(*rule-tac x* = *length xs* − *n mod length xs* **in** *exI*)
apply *simp*
done

lemma *singleton-list-cong-eq-iff*[*simp*]:
({*xs*::*'a list*} // {≅} = {*ys*} // {≅}) = (*xs* ≅ *ys*)
by(*simp add*: *eq-equiv-class-iff2*[*OF equiv-EqF*])

2.2 Homomorphism and isomorphism

definition *is-pr-Hom* :: (*'a* ⇒ *'b*) ⇒ *'a Fgraph* ⇒ *'b Fgraph* ⇒ *bool* **where**
is-pr-Hom φ Fs$_1$ *Fs*$_2$ ≡ (*map φ* ‘ *Fs*$_1$)//{≅} = *Fs*$_2$ //{≅}

definition *is-pr-Iso* :: (*'a* ⇒ *'b*) ⇒ *'a Fgraph* ⇒ *'b Fgraph* ⇒ *bool* **where**
is-pr-Iso φ Fs$_1$ *Fs*$_2$ ≡ *is-pr-Hom φ Fs*$_1$ *Fs*$_2$ ∧ *inj-on φ* (⋃ *F* ∈ *Fs*$_1$. *set F*)

definition *is-pr-iso* :: (*'a* ⇒ *'b*) ⇒ *'a fgraph* ⇒ *'b fgraph* ⇒ *bool* **where**
is-pr-iso φ Fs$_1$ *Fs*$_2$ ≡ *is-pr-Iso φ* (*set Fs*$_1$) (*set Fs*$_2$)

Homomorphisms preserve the set of nodes.

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

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

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

lemma *pr-Hom-pres-nodes*:
is-pr-Hom φ Fs_1 Fs_2 $\Longrightarrow$ $\varphi\ `\ (\bigcup F{\in}Fs_1.\ set\ F) = (\bigcup F{\in}Fs_2.\ set\ F)$
apply(*drule pr-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* : $(\bigcup F{\in}Fs_1.\ \{\varphi\ `\ set\ F\})$)
prefer *2* **apply** *blast*
apply(*subgoal-tac EX* F':Fs_1. $\varphi\ `\ set\ F' = set\ F$)
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 pr-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* $(\%xs.\{xs\}//\{\cong\})$ $(set\ Fs_1)$
and *inj2*: *inj-on* $(\%xs.\{xs\}//\{\cong\})$ $(set\ Fs_2)$ **and** *iso*: *is-pr-iso* φ Fs_1 Fs_2
shows *length* Fs_1 = *length* Fs_2
proof −
have *injphi*: $\forall F{\in}set\ Fs_1.\ \forall F'{\in}set\ Fs_1.$ *inj-on* φ $(set\ F \cup set\ F')$ **using** *iso*
by(*auto simp*:*is-pr-iso-def is-pr-Iso-def is-pr-Hom-def inj-on-def*)

have *inj1'*: *inj-on* (%*xs*. {*xs*} // {≅}) (*map* φ ‘ *set* Fs_1)
apply(*rule inj-on-imageI*)
apply(*simp add*:*inj-on-def quotient-def eq-equiv-class-iff*[*OF equiv-EqF*])
apply(*simp add*: *congs-map-eq-iff injphi*)
using *inj1*
apply(*simp add*:*inj-on-def quotient-def eq-equiv-class-iff*[*OF equiv-EqF*])
done
have *length* Fs_1 = *card*(*set* Fs_1) **by**(*simp add*:*distinct-card*[*OF dist1*])
also have ... = *card*(*map* φ ‘ *set* Fs_1) **using** *iso*
by(*auto simp*:*is-pr-iso-def is-pr-Iso-def is-pr-Hom-def inj-on-mapI card-image*)
also have ... = *card*((*map* φ ‘ *set* Fs_1) // {≅})
by(*simp add*: *card-quotient-disjoint*[*OF - inj1'*])
also have (*map* φ ‘ *set* Fs_1)//{≅} = *set* Fs_2 // {≅}
using *iso* **by**(*simp add*: *is-pr-iso-def is-pr-Iso-def is-pr-Hom-def*)
also have *card*(...) = *card*(*set* Fs_2)
by(*simp add*: *card-quotient-disjoint*[*OF - inj2*])
also have ... = *length* Fs_2 **by**(*simp add*:*distinct-card*[*OF dist2*])
finally show *?thesis* **.**
qed

lemma *is-Hom-distinct*:
⟦ *is-pr-Hom* φ Fs_1 Fs_2; ∀ *F*∈Fs_1. *distinct F*; ∀ *F*∈Fs_2. *distinct F* ⟧
⟹ ∀ *F*∈Fs_1. *distinct*(*map* φ *F*)
apply(*clarsimp simp add*:*is-pr-Hom-def*)
apply(*subgoal-tac* ∃ *F'* ∈ Fs_2. (*map* φ *F*, *F'*) : {≅})
apply(*fastforce simp add*: *congs-def*)
apply(*subgoal-tac* ∃ *F'* ∈ Fs_2. {*map* φ *F*}//{≅} = {*F'*}//{≅})
apply *clarify*
apply(*rule-tac x* = *F'* **in** *bexI*)
apply(*rule eq-equiv-class*[*OF - equiv-EqF*])
apply(*simp add*:*singleton-quotient*)
apply *blast*
apply *assumption*
apply(*simp add*:*quotient-def*)
apply(*rotate-tac 1*)
apply *blast*
done

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

lemma *is-pr-Hom-trans*: **assumes** *f*: *is-pr-Hom f A B* **and** *g*: *is-pr-Hom g B C*
shows *is-pr-Hom* (*g o f*) *A C*

proof−
from *f* **have** *f1*: *ALL a*:*A*. *EX b*:*B*. *map f a* ≅ *b*
apply(*simp add*: *is-pr-Hom-def quotient-def Iso-def*)
apply(*erule equalityE*)
apply *blast*
done
from *f* **have** *f2*: *ALL b*:*B*. *EX a*:*A*. *map f a* ≅ *b*
apply(*simp add*: *is-pr-Hom-def quotient-def Iso-def*)
apply(*erule equalityE*)
apply *blast*
done
from *g* **have** *g1*: *ALL b*:*B*. *EX c*:*C*. *map g b* ≅ *c*
apply(*simp add*: *is-pr-Hom-def quotient-def Iso-def*)
apply(*erule equalityE*)
apply *blast*
done
from *g* **have** *g2*: *ALL c*:*C*. *EX b*:*B*. *map g b* ≅ *c*
apply(*simp add*: *is-pr-Hom-def quotient-def Iso-def*)
apply(*erule equalityE*)
apply *blast*
done
show *?thesis*
apply(*auto simp add*: *is-pr-Hom-def quotient-def Iso-def Image-def*
map-comp-map[*symmetric*] *image-compose simp del*: *map-map map-comp-map*)
apply (*metis congs-map*[*of - - g*] *congs-trans f1 g1*)
by (*metis congs-map*[*of - - g*] *congs-sym congs-trans f2 g2*)
qed

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

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

lemma *is-pr-Iso-rec*:
⟦ *inj-on* (%*xs*. {*xs*}//{≅}) *Fs*$_1$; *inj-on* (%*xs*. {*xs*}//{≅}) *Fs*$_2$; *F*$_1$ ∈ *Fs*$_1$ ⟧ ⟹
is-pr-Iso φ Fs$_1$ *Fs*$_2$ =
(∃ *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-pr-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(*fastforce intro*: *congs-sym*)
 apply(*subgoal-tac* {F_2}//{≅} = {*map* φ F_1}//{≅})
  prefer *2* **apply**(*simp add*:*singleton-quotient*)
 apply(*subgoal-tac* $\forall F{\in}Fs_1{'}.\ \neg$ (*map* φ F) ≅ (*map* φ F_1))
 apply(*fastforce simp*:*Iso-def quotient-def Image-Collect-split simp del*: *Collect-congs-eq-iff*
          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-eq-iff*)
 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-pr-Hom-def quotient-diff1*)
apply (*simp add*:*singleton-quotient*)
apply(*subgoal-tac* F_2 ≅ (*map* φ F_1))
 prefer *2* **apply**(*fastforce simp add*:*congs-def*)
apply(*subgoal-tac* {≅}``{*map* $\varphi$ $F_1$} = {≅}``{F_2})
 prefer *2*
 apply(*rule equiv-class-eq*[*OF equiv-EqF*])
 apply(*fastforce intro*:*congs-sym*)
apply(*subgoal-tac* {≅}``{F_2} ∈ Fs_2 // {≅})
 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* $(\%xs.\{xs\}//\{\cong\})$ $(set\ Fs_2)$ ⟧
⟹
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* ⟧ ⟹
(*map-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

primrec *pr-iso-test0* :: $('a \sim\!\!=\!\!> 'b) \Rightarrow 'a\ fgraph \Rightarrow 'b\ fgraph \Rightarrow bool$ **where**
pr-iso-test0 m [] Fs_2 = (Fs_2 = [])
| *pr-iso-test0 m* $(F_1\#Fs_1)$ Fs_2 =
$(\exists F_2 \in set\ Fs_2.\ length\ F_1 = length\ F_2 \wedge$
$(\exists n.$ *let* m' = *map-of*(*zip* F_1 (*rotate n* F_2)) *in*
if $m \subseteq_m m +\!\!+ m' \wedge$ *inj-on* $(m +\!\!+ m')$ $(dom(m +\!\!+ m'))$
then pr-iso-test0 $(m +\!\!+ m')$ Fs_1 *(remove1* F_2 Fs_2) *else False*))

lemma *map-compatI*: ⟦ $f \subseteq_m$ *Some* o h; $g \subseteq_m$ *Some* o h ⟧ ⟹ $f \subseteq_m f +\!\!+ g$
by (*fastforce simp add*: *map-le-def map-add-def dom-def split*:*option.splits*)

lemma *inj-on-map-addI1*:
⟦ *inj-on m A*; $m \subseteq_m m +\!\!+ m'$; $A \subseteq dom\ m$ ⟧ ⟹ *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 *fastforce*
apply *auto*
apply *fastforce*
apply(*subgoal-tac m x = Some a*)
prefer *2* **apply** (*fastforce*)
apply(*subgoal-tac m y = Some a*)
prefer *2* **apply** (*fastforce*)
apply(*subgoal-tac m x = m y*)
prefer *2* **apply** *simp*
apply (*blast*)
done

lemma *map-image-eq*: ⟦ $A \subseteq dom\ m$; $m \subseteq_m m'$ ⟧ ⟹ $m\ `\ A = m'\ `\ A$
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'*); $m \subseteq_m Some\ o\ f$; $m' \subseteq_m Some\ o\ f$;
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(*fastforce intro*!: *inj-on-map-addI1 map-compatI*)
apply(*clarify*)
apply(*subgoal-tac m* ++ *m'* $\subseteq_m Some \circ f$)
prefer *2* **apply**(*fast intro*:*map-add-le-mapI map-compatI*)
apply(*subgoal-tac dom m'* − *dom m* ⊆ *dom*(*m*++*m'*))
prefer *2* **apply**(*fastforce*)
apply(*insert map-image-eq*[*of dom m'* − *dom m m*++*m' Some o f*])
apply(*subgoal-tac dom m* − *dom m'* ⊆ *dom*(*m*++*m'*))
prefer *2* **apply**(*fastforce*)
apply(*insert map-image-eq*[*of dom m* − *dom m' m*++*m' Some 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*
apply(*rule conjI*)
apply(*erule inj-on-fun-updI*)
apply(*simp add*:*image-def*)
apply *clarsimp*
apply(*drule* (*1*) *map-of-zip-eq-SomeD*[*OF - sym*])
apply *fast*
apply(*clarsimp simp add*:*image-def*)
apply(*drule* (*1*) *map-of-zip-eq-SomeD*[*OF - sym*])
apply *fast*
done

lemma *pr-iso-test0-correct*: $\bigwedge m\ Fs_2$.
$[\![\forall F{\in}set\ Fs_1.\ distinct\ F;\ \forall F{\in}set\ Fs_2.\ distinct\ F;$
$distinct\ Fs_1;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_1);$
$distinct\ Fs_2;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_2);\ inj\text{-}on\ m\ (dom\ m)\ ]\!] \Longrightarrow$
$pr\text{-}iso\text{-}test0\ m\ Fs_1\ Fs_2 =$
$(\exists \varphi.\ is\text{-}pr\text{-}iso\ \varphi\ Fs_1\ Fs_2 \wedge m \subseteq_m Some\ o\ \varphi \wedge$
$inj\text{-}on\ \varphi\ (dom\ m \cup (\bigcup F{\in}set\ Fs_1.\ set\ F)))$
apply(*induct* Fs_1)
apply(*simp add*:*inj-on-def dom-def*)
apply(*rule iffI*)
apply (*simp add*:*is-pr-iso-def is-pr-Iso-def is-pr-Hom-def*)
apply(*rule-tac* $x = the\ o\ m$ **in** *exI*)
apply (*fastforce simp*: *map-le-def*)
apply (*clarsimp simp*:*is-pr-iso-def is-pr-Iso-def is-pr-Hom-def*)
apply(*rename-tac* $F_1\ Fs_1{'}\ m\ Fs_2$)
apply(*clarsimp simp*:*Let-def Ball-def*)
apply(*simp add*: *is-iso-Cons*)
apply(*rule iffI*)

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

corollary *pr-iso-test0-corr*:
⟦ ∀ *F*∈*set* Fs_1. *distinct F*; ∀ *F*∈*set* Fs_2. *distinct F*;
distinct Fs_1; *inj-on* (%*xs*.{*xs*}//{≅}) (*set* Fs_1);
distinct Fs_2; *inj-on* (%*xs*.{*xs*}//{≅}) (*set* Fs_2) ⟧ ⟹
pr-iso-test0 empty Fs_1 Fs_2 = (∃ φ. *is-pr-iso* φ Fs_1 Fs_2)
apply(*subst pr-iso-test0-correct*)
apply *assumption*+
apply *simp*
apply (*simp add*:*is-pr-iso-def is-pr-Iso-def*)
done

Now we bound the number of rotations needed. We have to exclude the empty face [] to be able to restrict the search to $n <$ *length xs* (which would otherwise be vacuous).

primrec *pr-iso-test1* :: ('*a* ~=> '*b*) ⇒ '*a fgraph* ⇒ '*b fgraph* ⇒ *bool* **where**
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* ⊆$_m$ *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. [] $\notin$ *set* Fs_2 $\Longrightarrow$ *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* $F_2 \neq$ [])
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; [] $\notin$ *set* Fs_2;
distinct Fs_1; *inj-on* (%xs.{xs}//{≅}) (*set* Fs_1);
distinct Fs_2; *inj-on* (%xs.{xs}//{≅}) (*set* Fs_2) ⟧ $\Longrightarrow$
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.

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

type-synonym
('a,'b)*tester* = ('a * 'b)*list* ⇒ ('a * 'b)*list* ⇒ *bool*
type-synonym
('a,'b)*merger* = ('a * 'b)*list* ⇒ ('a * 'b)*list* ⇒ ('a * 'b)*list*

primrec *pr-iso-test2* :: ('a,'b)*tester* ⇒ ('a,'b)*merger* ⇒
('a * 'b)*list* ⇒ 'a *fgraph* ⇒ 'b *fgraph* ⇒ *bool* **where**
pr-iso-test2 tst mrg I [] Fs_2 = (Fs_2 = [])
| *pr-iso-test2 tst mrg I* (F_1#Fs_1) Fs_2 =
(∃ F_2 ∈ *set* Fs_2. *length* F_1 = *length* F_2 ∧
(∃ n < *length* F_2. *let* I' = *zip* F_1 (*rotate* n F_2) *in*
if *tst* I' I
then pr-iso-test2 tst mrg (*mrg* I' I) Fs_1 (*remove1* F_2 Fs_2) *else False*))

lemma *notin-range-map-of*:

y ∉ *snd* ‘ *set xys* ⟹ *Some y* ∉ *range*(*map-of xys*)
apply(*induct xys*)
apply (*simp add*:*image-def*)
apply(*clarsimp split*:*if-splits*)
done

lemma *inj-on-map-upd*:
⟦ *inj-on m* (*dom m*); *Some y* ∉ *range m* ⟧ ⟹ *inj-on* (*m*(*x*↦*y*)) (*dom m*)
apply(*simp add*:*inj-on-def dom-def image-def*)
apply (*blast intro*:*sym*)
done

lemma [*simp*]:
distinct(*map snd xys*) ⟹ *inj-on* (*map-of xys*) (*dom*(*map-of 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 *fastforce*
done

lemma *lem*: *Ball* (*set xs*) *P* ⟹ *Ball* (*set* (*remove1 x xs*)) *P* = *True*
by(*induct xs*) *simp-all*

lemma *pr-iso-test2-conv-1*:
!!*I Fs*$_2$.
⟦ ∀ *I I*′. *oneone I* ⟶ *oneone I*′ ⟶
tst I′ *I* = (*let m* = *map-of I*; *m*′ = *map-of I*′
in m ⊆$_m$ *m* ++ *m*′ ∧ *inj-on* (*m*++*m*′) (*dom*(*m*++*m*′)));
∀ *I I*′. *oneone I* ⟶ *oneone I*′ ⟶ *tst I*′ *I*
⟶ *map-of*(*mrg I*′ *I*) = *map-of I* ++ *map-of I*′;
∀ *I I*′. *oneone I* & *oneone I*′ ⟶ *tst I*′ *I* ⟶ *oneone* (*mrg I*′ *I*);
oneone I;
∀ *F* ∈ *set Fs*$_1$. *distinct F*; ∀ *F* ∈ *set Fs*$_2$. *distinct F* ⟧ ⟹
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 cong*: *conj-cong*)
done

A simple implementation

definition *compat* :: (′*a*,′*b*)*tester* **where**
compat I I′ ==
∀ (*x*,*y*) ∈ *set I*. ∀ (*x*′,*y*′) ∈ *set I*′. (*x* = *x*′) = (*y* = *y*′)

lemma *image-map-upd*:
x ∉ *dom m* ⟹ *m*(*x*↦*y*) ‘ *A* = *m* ‘ (*A*−{*x*}) ∪ (*if x* ∈ *A then* {*Some y*} *else*

{})
by(*auto simp*:*image-def dom-def*)

lemma *image-map-of-conv-Image*:
!!*A*. ⟦ *distinct*(*map fst xys*) ⟧
⟹ *map-of xys* ‘ *A* = *Some* ‘ (*set xys* ‘‘ *A*) ∪ (*if A* ⊆ *fst* ‘ *set xys then* {} *else* {*None*})
apply (*induct xys*)
apply (*simp add*:*image-def Image-def Collect-conv-if*)
apply (*simp add*:*image-map-upd dom-map-of-conv-image-fst*)
apply(*erule thin-rl*)
apply (*clarsimp simp*:*image-def Image-def*)
apply((*rule conjI*, *clarify*)+, *fastforce*)
apply *fastforce*
apply(*clarify*)
apply((*rule conjI*, *clarify*)+, *fastforce*)
apply *fastforce*
apply *fastforce*
apply *fastforce*
done

lemma [*simp*]: *m*++*m*′ ‘ (*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 *compat-correct*:
⟦ *oneone I*; *oneone I*′ ⟧ ⟹
compat I′ *I* = (*let m* = *map-of I*; *m*′ = *map-of I*′
in m ⊆*m m* ++ *m*′ ∧ *inj-on* (*m*++*m*′) (*dom*(*m*++*m*′)))
apply(*simp add*: *compat-def Let-def map-le-iff-map-add-commute*)
apply(*rule iffI*)
apply(*rule context-conjI*)
apply(*rule ext*)
apply (*fastforce simp add*:*map-add-def split*:*option.split*)
apply(*simp add*:*inj-on-Un*)
apply(*drule sym*)
apply *simp*
apply(*simp add*: *dom-map-of-conv-image-fst image-map-of-conv-Image*)
apply(*simp add*: *image-def Image-def*)

apply *fastforce*
apply *clarsimp*
apply(*rename-tac a b aa ba*)
apply(*rule iffI*)
apply (*clarsimp simp*: *fun-eq-iff*)
apply(*erule-tac x = aa* **in** *allE*)
apply (*simp add*:*map-add-def*)
apply (*clarsimp simp*:*dom-map-of-conv-image-fst*)
apply(*simp* (*no-asm-use*) *add*:*inj-on-def*)
apply(*drule-tac x = a* **in** *bspec*)
apply *force*
apply(*drule-tac x = aa* **in** *bspec*)
apply *force*
apply(*erule mp*)
apply *simp*
apply(*drule sym*)
apply *simp*
done

corollary *compat-corr*:
$\forall I\ I'.\ oneone\ I \longrightarrow oneone\ I' \longrightarrow$
compat $I'\ I$ = (*let* m = *map-of* I; m' = *map-of* I'
in $m \subseteq_m m\ ++\ m' \wedge$ *inj-on* $(m{+}{+}m')$ $(dom(m{+}{+}m'))$)
by(*simp add*: *compat-correct*)

definition *merge0* :: $('a,'b)$*merger* **where**
merge0 $I'\ I \equiv [xy \leftarrow I'.\ fst\ xy \notin fst\ `\ set\ I] @ I$

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 *merge0-correct*:
$\forall I\ I'.\ oneone\ I \longrightarrow oneone\ I' \longrightarrow compat\ I'\ I$
$\longrightarrow$ *map-of*(*merge0* $I'\ I$) = *map-of* I ++ *map-of* I'
apply(*simp add*:*compat-def merge0-def help1 fun-eq-iff map-add-def restrict-map-def split*:*option.split*)
apply *fastforce*
done

lemma *merge0-inv*:
$\forall I\ I'.\ oneone\ I \wedge oneone\ I' \longrightarrow compat\ I'\ I \longrightarrow oneone\ (merge0\ I'\ I)$

apply(*auto simp add*:*merge0-def distinct-map compat-def split-def*)
apply(*blast intro*:*subset-inj-on*)+
done

corollary *pr-iso-test2-corr*:
⟦ ∀ *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-test2 compat merge0 [] *Fs*$_1$ *Fs*$_2$ = (∃ φ. *is-pr-iso* φ *Fs*$_1$ *Fs*$_2$)
by(*simp add*: *pr-iso-test2-conv-1*[*OF compat-corr merge0-correct merge0-inv*]
pr-iso-test1-corr)

Implementing merge as a recursive function:

primrec *merge* :: ('*a*,'*b*)*merger* **where**
merge [] *I* = *I*
| *merge* (*xy*#*xys*) *I* = (*let* (*x*,*y*) = *xy in*
if ∀ (*x*′,*y*′) ∈ *set I*. *x* ≠ *x*′ *then xy* # *merge xys I else merge xys I*)

lemma *merge-conv-merge0*: *merge I*′ *I* = *merge0 I*′ *I*
apply(*induct I*′)
apply(*simp add*:*merge0-def*)
apply(*force simp add*:*Let-def list-all-iff merge0-def*)
done

primrec *pr-iso-test-rec* :: ('*a* ∗ '*b*)*list* ⇒ '*a fgraph* ⇒ '*b fgraph* ⇒ *bool* **where**
pr-iso-test-rec I [] *Fs*$_2$ = (*Fs*$_2$ = [])
| *pr-iso-test-rec I* (*F*$_1$#*Fs*$_1$) *Fs*$_2$ =
(∃ *F*$_2$ ∈ *set Fs*$_2$. *length F*$_1$ = *length F*$_2$ ∧
(∃ *n* < *length F*$_2$. *let I*′ = *zip F*$_1$ (*rotate n F*$_2$) *in*
compat I′ *I* ∧ *pr-iso-test-rec* (*merge I*′ *I*) *Fs*$_1$ (*remove1 F*$_2$ *Fs*$_2$)))

lemma *pr-iso-test-rec-conv-2*:
!!*I Fs*$_2$. *pr-iso-test-rec I Fs*$_1$ *Fs*$_2$ = *pr-iso-test2 compat merge0 I Fs*$_1$ *Fs*$_2$
apply(*induct Fs*$_1$)
apply *simp*
apply(*auto simp*: *merge-conv-merge0 list-ex-iff Bex-def Let-def*)
done

corollary *pr-iso-test-rec-corr*:
⟦ ∀ *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-test-rec [] *Fs*$_1$ *Fs*$_2$ = (∃ φ. *is-pr-iso* φ *Fs*$_1$ *Fs*$_2$)
by(*simp add*: *pr-iso-test-rec-conv-2 pr-iso-test2-corr*)

definition *pr-iso-test* :: '*a fgraph* ⇒ '*b fgraph* ⇒ *bool* **where**
pr-iso-test Fs$_1$ *Fs*$_2$ = *pr-iso-test-rec* [] *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*$_2$) ⟧ ⟹
pr-iso-test Fs$_1$ *Fs*$_2$ = (∃ φ. *is-pr-iso* φ *Fs*$_1$ *Fs*$_2$)
apply(*simp add*:*pr-iso-test-def pr-iso-test-rec-corr*)
done

2.3.2 'Improper' Isomorphisms

definition *is-Iso* :: ('*a* ⇒ '*b*) ⇒ '*a Fgraph* ⇒ '*b Fgraph* ⇒ *bool* **where**
is-Iso φ *Fs*$_1$ *Fs*$_2$ ≡ *is-pr-Iso* φ *Fs*$_1$ *Fs*$_2$ ∨ *is-pr-Iso* φ *Fs*$_1$ (*rev* ‘ *Fs*$_2$)

definition *is-iso* :: ('*a* ⇒ '*b*) ⇒ '*a fgraph* ⇒ '*b fgraph* ⇒ *bool* **where**
is-iso φ *Fs*$_1$ *Fs*$_2$ ≡ *is-Iso* φ (*set Fs*$_1$) (*set Fs*$_2$)

definition *iso-fgraph* :: '*a fgraph* ⇒ '*a fgraph* ⇒ *bool* (**infix** ≃ *60*) **where**
g$_1$ ≃ *g*$_2$ ≡ ∃ φ. *is-iso* φ *g*$_1$ *g*$_2$

lemma *iso-fgraph-trans*: **assumes** *f* ≃ (*g*::'*a fgraph*) **and** *g* ≃ *h* **shows** *f* ≃ *h*
proof−
{ **fix** φ φ′ **assume** *is-pr-Hom* φ (*set f*) (*set g*) *inj-on* φ (⋃ *F*∈*set f*. *set F*)
is-pr-Hom φ′ (*set g*) (*set h*) *inj-on* φ′ (⋃ *F*∈*set g*. *set F*)
hence *is-pr-Hom* (φ′ ∘ φ) (*set f*) (*set h*) ∧
inj-on (φ′ ∘ φ) (⋃ *F*∈*set f*. *set F*)
by(*simp add*: *is-pr-Hom-trans comp-inj-on pr-Hom-pres-nodes*)
} **moreover**
{ **fix** φ φ′ **assume** *is-pr-Hom* φ (*set f*) (*set g*) *inj-on* φ (⋃ *F*∈*set f*. *set F*)
is-pr-Hom φ′ (*set g*) (*rev* ‘ *set h*) *inj-on* φ′ (⋃ *F*∈*set g*. *set F*)
hence *is-pr-Hom* (φ′ ∘ φ) (*set f*) (*rev* ‘ *set h*) ∧
inj-on (φ′ ∘ φ) (⋃ *F*∈*set f*. *set F*)
by(*simp add*: *is-pr-Hom-trans comp-inj-on pr-Hom-pres-nodes*)
} **moreover**
{ **fix** φ φ′ **assume** *is-pr-Hom* φ (*set f*) (*rev* ‘ *set g*) *inj-on* φ (⋃ *F*∈*set f*. *set F*)
is-pr-Hom φ′ (*set g*) (*set h*) *inj-on* φ′ (⋃ *F*∈*set g*. *set F*)
with *this*(*3*)[*THEN is-pr-Hom-rev*]
have *is-pr-Hom* (φ′ ∘ φ) (*set f*) (*rev* ‘ *set h*) ∧
inj-on (φ′ ∘ φ) (⋃ *F*∈*set f*. *set F*)
by(*simp add*: *is-pr-Hom-trans comp-inj-on pr-Hom-pres-nodes*)
} **moreover**
{ **fix** φ φ′ **assume** *is-pr-Hom* φ (*set f*) (*rev* ‘ *set g*) *inj-on* φ (⋃ *F*∈*set f*. *set F*)
is-pr-Hom φ′ (*set g*) (*rev* ‘ *set h*) *inj-on* φ′ (⋃ *F*∈*set g*. *set F*)
with *this*(*3*)[*THEN is-pr-Hom-rev*]
have *is-pr-Hom* (φ′ ∘ φ) (*set f*) (*set h*) ∧
inj-on (φ′ ∘ φ) (⋃ *F*∈*set f*. *set F*)
by(*simp add*: *is-pr-Hom-trans comp-inj-on pr-Hom-pres-nodes*)
} **ultimately show** *?thesis* **using** *assms*
by(*simp add*: *iso-fgraph-def is-iso-def is-Iso-def is-pr-Iso-def*) *blast*

qed

definition *iso-test* :: *'a fgraph* ⇒ *'b fgraph* ⇒ *bool* **where**
iso-test g_1 g_2 ⟷ *pr-iso-test* g_1 g_2 ∨ *pr-iso-test* g_1 (*map rev* g_2)

theorem *iso-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) ⟧ ⟹
iso-test Fs_1 Fs_2 = (Fs_1 ≃ Fs_2)
apply(*simp add*:*iso-test-def pr-iso-test-correct iso-fgraph-def*)
apply(*subst pr-iso-test-correct*)
apply *simp*
apply *simp*
apply (*simp add*:*image-def*)
apply *simp*
apply *simp*
apply (*simp add*:*distinct-map*)
apply (*simp add*:*inj-on-image-iff*)
apply(*simp add*:*is-iso-def is-Iso-def is-pr-iso-def*)
apply *blast*
done

lemma *iso-fgraph-refl*[*iff*]: *g* ≃ *g*
apply(*simp add*: *iso-fgraph-def*)
apply(*rule-tac x* = *id* **in** *exI*)
apply(*simp add*: *is-iso-def is-Iso-def is-pr-Iso-def is-pr-Hom-def id-def*)
done

2.4 Elementhood and containment modulo

interpretation *qle-gr*: *quasi-order op* ≃
proof qed (*auto intro*:*iso-fgraph-trans*)

abbreviation *qle-gr-in* :: *'a fgraph* ⇒ *'a fgraph set* ⇒ *bool* (**infix** ∈≃ *60*)
where *x* ∈≃ *M* ≡ *qle-gr.in-qle x M*
abbreviation *qle-gr-sub* :: *'a fgraph set* ⇒ *'a fgraph set* ⇒ *bool* (**infix** ⊆≃ *60*)
where *x* ⊆≃ *M* ≡ *qle-gr.subseteq-qle x M*
abbreviation *qle-gr-eq* :: *'a fgraph set* ⇒ *'a fgraph set* ⇒ *bool* (**infix** =≃ *60*)
where *x* =≃ *M* ≡ *qle-gr.seteq-qle x M*

end

3 More Rotation

theory *Rotation*
imports *ListAux PlaneGraphIso*

begin

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

definition *rotate-min* :: *nat list* ⇒ *nat list* **where**
rotate-min vs ≡ *rotate-to vs* (*min-list 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*
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* (λ*vs*. {*vs*}//{≅}) *A*
proof −
{ fix *xs ys* **assume** *xs*: *xs* ∈ *A* **and** *ys* : *ys* ∈ *A*
hence *xs* ≠ [] ∧ *ys* ≠ [] **using** *assms*(*2*) **by** *blast*
hence (*rotate-min xs* = *rotate-min ys*) = (*xs* ≅ *ys*)
using *xs assms*(*1*)
by(*simp add*: *singleton-list-cong-eq-iff norm-eq-iff-face-cong*)
} thus *?thesis* **by**(*simp add*:*inj-on-def*)
qed

end

4 Graph

theory *Graph*
imports *Rotation*
begin

syntax (*xsymbols*)
-UNION1 :: *pttrns* => *'b set* => *'b set* (($3\bigcup(00\_)/$ -) [*0*, *10*] *10*)
-INTER1 :: *pttrns* => *'b set* => *'b set* (($3\bigcap(00\_)/$ -) [*0*, *10*] *10*)
-UNION :: *pttrn* => *'a set* => *'b set* => *'b set* (($3\bigcup(00\_{\in}\_)/$ -) [*0*, *0*, *10*] *10*)
-INTER :: *pttrn* => *'a set* => *'b set* => *'b set* (($3\bigcap(00\_{\in}\_)/$ -) [*0*, *0*, *10*] *10*)

4.1 Notation

type-synonym *vertex* = *nat*

consts

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

abbreviation *vertices-set* :: $'a \Rightarrow$ *vertex set* ($\mathcal{V}$) **where**
$\mathcal{V}\ f \equiv$ *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* :: $'a \Rightarrow$ *bool*
consts *type* :: $'a \Rightarrow$ *facetype*

overloading
final-face $\equiv$ *final* :: *face* $\Rightarrow$ *bool*
type-face $\equiv$ *type* :: *face* $\Rightarrow$ *facetype*
vertices-face $\equiv$ *vertices* :: *face* $\Rightarrow$ *vertex list*
begin

primrec *final-face* **where**
final (*Face vs f*) = (*case f of Final* $\Rightarrow$ *True* | *Nonfinal* $\Rightarrow$ *False*)

primrec *type-face* **where**
type (*Face vs f*) = *f*

primrec *vertices-face* **where**
vertices (*Face vs f*) = *vs*

end

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.

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

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

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

definition *nextVertex* :: *face* $\Rightarrow$ *vertex* $\Rightarrow$ *vertex* **where**

$f \cdot \equiv$ *let vs = vertices f in nextElem vs (hd vs)*

nextVertices is n-fold application of *nextvertex*.

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

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

defs (*vs*::*vertex list*)^{op} ≡ *rev vs*
overloading
op-graph ≡ *Graph.op* :: *face* ⇒ *face*
begin

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

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

abbreviation
triangle :: *face* ⇒ *bool* **where**
triangle f == |*vertices f*| = *3*

4.3 Graphs

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

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

abbreviation
Faces :: *graph* ⇒ *face set* ($\mathcal{F}$) **where**
$\mathcal{F}$ *g* == *set*(*faces g*)

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

overloading
vertices-graph ≡ *vertices* :: *graph* ⇒ *vertex list*
begin

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

end

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

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

lemma *len-vertices-graph*:
|*vertices g*| = *countVertices g*
by (*simp add*:*vertices-graph*)

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

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

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

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

definition *graph* :: *nat* ⇒ *graph* **where**
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

definition *finals* :: *graph* ⇒ *face list* **where**
finals g ≡ [*f* ← *faces g*. *final f*]

definition *nonFinals* :: *graph* ⇒ *face list* **where**
nonFinals g ≡ [*f* ← *faces g*. ¬ *final f*]

definition *countNonFinals* :: *graph* ⇒ *nat* **where**
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*)

definition *finalVertex* :: *graph* ⇒ *vertex* ⇒ *bool* **where**

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

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

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

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

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

definition *vertextype* :: *graph* ⇒ *vertex* ⇒ *nat* × *nat* × *nat* **where**
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*)

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

definition *noExceptionals* :: *graph* ⇒ *vertex set* ⇒ *bool* **where**
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*) ≡ ⋃*f* ∈ ℱ *g* *edges f*

definition *neighbors* :: *graph* ⇒ *vertex* ⇒ *vertex list* **where**
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*

definition *nextFace* :: *graph* × *vertex* ⇒ *face* ⇒ *face* **where**

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

4.6 Code generator setup

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

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

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

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

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

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

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

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

end

5 Immutable Arrays with Code Generation

theory *IArray*
imports *Main*
begin

Immutable arrays are lists wrapped up in an additional constructor. There are no update operations. Hence code generation can safely implement this

type by efficient target language arrays. Currently only SML is provided. Should be extended to other target languages and more operations.

Note that arrays cannot be printed directly but only by turning them into lists first. Arrays could be converted back into lists for printing if they were wrapped up in an additional constructor.

datatype *'a iarray = IArray 'a list*

primrec *list-of* :: *'a iarray ⇒ 'a list* **where**
list-of (IArray xs) = xs
hide-const (**open**) *list-of*

definition *of-fun* :: *(nat ⇒ 'a) ⇒ nat ⇒ 'a iarray* **where**
[*simp*]: *of-fun f n = IArray (map f [0..<n])*
hide-const (**open**) *of-fun*

definition *sub* :: *'a iarray ⇒ nat ⇒ 'a* (**infixl** !! *100*) **where**
[*simp*]: *as !! n = IArray.list-of as ! n*
hide-const (**open**) *sub*

definition *length* :: *'a iarray ⇒ nat* **where**
[*simp*]: *length as = List.length (IArray.list-of as)*
hide-const (**open**) *length*

lemma *list-of-code* [*code*]:
IArray.list-of as = map (λn. as !! n) [0 ..< IArray.length as]
by (*cases as*) (*simp add*: *map-nth*)

5.1 Code Generation

code-reserved *SML Vector*

code-type *iarray*
(*SML - Vector.vector*)

code-const *IArray*
(*SML Vector.fromList*)

lemma [*code*]:
size (as :: 'a iarray) = 0
by (*cases as*) *simp*

lemma [*code*]:
iarray-size f as = Suc (list-size f (IArray.list-of as))
by (*cases as*) *simp*

lemma [*code*]:
iarray-rec f as = f (IArray.list-of as)
by (*cases as*) *simp*

lemma [*code*]:
iarray-case f as = *f* (*IArray.list-of as*)
by (*cases as*) *simp*

lemma [*code*]:
HOL.equal as bs ⟷ *HOL.equal* (*IArray.list-of as*) (*IArray.list-of bs*)
by (*cases as*, *cases bs*) (*simp add*: *equal*)

primrec *tabulate* :: *integer* × (*integer* ⇒ *'a*) ⇒ *'a iarray* **where**
tabulate (*n*, *f*) = *IArray* (*map* (*f* ∘ *integer-of-nat*) [*0*..<*nat-of-integer n*])
hide-const (**open**) *tabulate*

lemma [*code*]:
IArray.of-fun f n = *IArray.tabulate* (*integer-of-nat n*, *f* ∘ *nat-of-integer*)
by *simp*

code-const *IArray.tabulate*
(*SML Vector.tabulate*)

primrec *sub'* :: *'a iarray* × *integer* ⇒ *'a* **where**
sub' (*as*, *n*) = *IArray.list-of as* ! *nat-of-integer n*
hide-const (**open**) *sub'*

lemma [*code*]:
as !! *n* = *IArray.sub'* (*as*, *integer-of-nat n*)
by *simp*

code-const *IArray.sub'*
(*SML Vector.sub*)

definition *length'* :: *'a iarray* ⇒ *integer* **where**
[*simp*]: *length' as* = *integer-of-nat* (*List.length* (*IArray.list-of as*))
hide-const (**open**) *length'*

lemma [*code*]:
IArray.length as = *nat-of-integer* (*IArray.length' as*)
by *simp*

code-const *IArray.length'*
(*SML Vector.length*)

end

6 Syntax for operations on immutable arrays

theory *IArray-Syntax*

imports *Main* ~~/*src*/*HOL*/*Library*/*IArray*
begin

6.1 Tabulation

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

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

definition *tabulate3* :: *nat* ⇒ *nat* ⇒ *nat* ⇒
(*nat* ⇒ *nat* ⇒ *nat* ⇒ *'a*) ⇒ *'a iarray iarray iarray* **where**
tabulate3 l m n f ≡ *IArray.of-fun* (λ*i*. *IArray.of-fun* (λ*j*. *IArray.of-fun* (λ*k*. *f i j k*) *n*) *m*) *l*

syntax
-*tabulate* :: *'a* ⇒ *pttrn* ⇒ *nat* ⇒ *'a iarray* (([⟦-. - < -⟧]))
-*tabulate2* :: *'a* ⇒ *pttrn* ⇒ *nat* ⇒ *pttrn* ⇒ *nat* ⇒ *'a iarray*
(([⟦-. - < -, - < -⟧]))
-*tabulate3* :: *'a* ⇒ *pttrn* ⇒ *nat* ⇒ *pttrn* ⇒ *nat* ⇒ *pttrn* ⇒ *nat* ⇒ *'a iarray*
(([⟦-. - < -, - < -, - < - ⟧]))

translations
⟦*f*. *x* < *n*⟧ == *CONST tabulate n* (λ*x*. *f*)
⟦*f*. *x* < *m*, *y* < *n*⟧ == *CONST tabulate2 m n* (λ*x y*. *f*)
⟦*f*. *x* < *l*, *y* < *m*, *z* < *n*⟧ == *CONST tabulate3 l m n* (λ*x y z*. *f*)

6.2 Access

abbreviation *sub1-syntax* :: *'a iarray* ⇒ *nat* ⇒ *'a* ((-⟦-⟧) [*1000*] *999*)
where
a⟦*n*⟧ ≡ *IArray.sub a n*

abbreviation *sub2-syntax* :: *'a iarray iarray* ⇒ *nat* ⇒ *nat* ⇒ *'a* ((-⟦-,-⟧) [*1000*] *999*)
where
as⟦*m*, *n*⟧ ≡ *IArray.sub* (*IArray.sub as m*) *n*

abbreviation *sub3-syntax* :: *'a iarray iarray iarray* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *'a* ((-⟦-,-,-⟧) [*1000*] *999*)
where
as⟦*l*, *m*, *n*⟧ ≡ *IArray.sub* (*IArray.sub* (*IArray.sub as l*) *m*) *n*

examples: ⟦*0*::*'a*. *i* < *5*⟧, ⟦*i*. *i* < *5*, *j* < *3*⟧

end

7 Enumerating Patches

theory *Enumerator*
imports *Graph IArray-Syntax*
begin

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

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

Label the vertices of F consecutively $0, \ldots, outer - 1$, 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.

definition *enumBase* :: *nat* ⇒ *nat list list* **where**
enumBase nmax ≡ [[*i*]. *i* ← [*0* ..< *Suc nmax*]]

definition *enumAppend* :: *nat* ⇒ *nat list list* ⇒ *nat list list* **where**
enumAppend nmax iss ≡ ⨆ *is*∈*iss* [*is* @ [*n*]. *n* ← [*last is* ..< *Suc nmax*]]

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

definition *enumTab* :: *nat list list iarray iarray* **where**
enumTab ≡ ⟦ *enumerator inner outer*. *inner* < *9*, *outer* < *9* ⟧

definition *enum* :: *nat* ⇒ *nat* ⇒ *nat list list* **where**
enum inner outer ≡ *if inner* < *9* & *outer* < *9 then enumTab*⟦*inner*,*outer*⟧
  else enumerator inner outer

primrec *hideDupsRec* :: *'a* ⇒ *'a list* ⇒ *'a option list* **where**

```
  hideDupsRec a [] = []
| hideDupsRec a (b#bs) =
    (if a = b then None # hideDupsRec b bs
    else Some b # hideDupsRec b bs)

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

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

end
```

8 Subdividing a Face

```
theory FaceDivision
imports Graph
begin

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


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


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

definition makeFaceFinal :: face ⇒ graph ⇒ graph where
 makeFaceFinal f g ≡
    Graph (makeFaceFinalFaceList f (faces g))
        (countVertices g)
        [makeFaceFinalFaceList f fs. fs ← faceListAt g]
        (heights g)


definition heightsNewVertices :: nat ⇒ nat ⇒ nat ⇒ nat list where
 heightsNewVertices h1 h2 n ≡ [min (h1 + i + 1) (h2 + n − i). i ← [0 ..< n]]
```

definition *splitFace*
:: *graph* ⇒ *vertex* ⇒ *vertex* ⇒ *face* ⇒ *vertex list* ⇒ *face* × *face* × *graph* **where**
splitFace g ram$_1$ *ram*$_2$ *oldF newVs* ≡
let fs = *faces g*;
n = *countVertices g*;
Fs = *faceListAt g*;
h = *heights g*;
vs$_1$ = *between* (*vertices oldF*) *ram*$_1$ *ram*$_2$;
vs$_2$ = *between* (*vertices oldF*) *ram*$_2$ *ram*$_1$;
(*f*$_1$, *f*$_2$) = *split-face oldF ram*$_1$ *ram*$_2$ *newVs*;
Fs = *replacefacesAt vs*$_1$ *oldF* [*f*$_1$] *Fs*;
Fs = *replacefacesAt vs*$_2$ *oldF* [*f*$_2$] *Fs*;
Fs = *replacefacesAt* [*ram*$_1$] *oldF* [*f*$_2$, *f*$_1$] *Fs*;
Fs = *replacefacesAt* [*ram*$_2$] *oldF* [*f*$_1$, *f*$_2$] *Fs*;
Fs = *Fs* @ *replicate* |*newVs*| [*f*$_1$, *f*$_2$] *in*
(*f*$_1$, *f*$_2$, *Graph* ((*replace oldF* [*f*$_2$] *fs*)@ [*f*$_1$])
(*n* + |*newVs*|)
Fs
(*h* @ *heightsNewVertices* (*h*!*ram*$_1$)(*h*!*ram*$_2$) |*newVs*|))

primrec *subdivFace*′ :: *graph* ⇒ *face* ⇒ *vertex* ⇒ *nat* ⇒ *vertex option list* ⇒ *graph* **where**
subdivFace′ *g f u n* [] = *makeFaceFinal f g*
| *subdivFace*′ *g f u n* (*vo*#*vos*) =
(*case vo of None* ⇒ *subdivFace*′ *g f u* (*Suc n*) *vos*
| (*Some v*) ⇒
if f•*u* = *v* ∧ *n* = *0*
then subdivFace′ *g f v 0 vos*
else let ws = [*countVertices g* ..< *countVertices g* + *n*];
(*f*$_1$, *f*$_2$, *g*′) = *splitFace g u v f ws in*
subdivFace′ *g*′ *f*$_2$ *v 0 vos*)

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

end

9 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 $\{(x,\ y).\ y \in set\ (f\ x)\}^*$.

abbreviation (*input*)
  in-set :: *'a => ('a => 'b list) => 'b => bool* (- [-]→ - [*55*,*0*,*55*] *50*) **where**
  g [*succs*]→ *g'* == *g'* ∈ *set* (*succs g*)

inductive-set
  RTranCl :: (*'a* ⇒ *'a list*) ⇒ (*'a* ∗ *'a*) *set*
  and *in-RTranCl* :: *'a* ⇒ (*'a* ⇒ *'a list*) ⇒ *'a* ⇒ *bool*
    (- [-]→∗ - [*55*,*0*,*55*] *50*)
  for *succs* :: *'a* ⇒ *'a list*
where
  g [*succs*]→∗ *g'* ≡ (*g*,*g'*) ∈ *RTranCl succs*
| *refl*: *g* [*succs*]→∗ *g*
| *succs*: *g* [*succs*]→ *g'* ⟹ *g'* [*succs*]→∗ *g''* ⟹ *g* [*succs*]→∗ *g''*

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

lemma *RTranCl-induct*:
 (*h*, *h'*) ∈ *RTranCl succs* ⟹
  P h ⟹
  (⋀*g g'*. *g'* ∈ *set* (*succs g*) ⟹ *P g* ⟹ *P g'*) ⟹
  P h'
proof −
  assume *s*: ⋀*g g'*. *g'* ∈ *set* (*succs g*) ⟹ *P g* ⟹ *P g'*
  assume (*h*, *h'*) ∈ *RTranCl succs P h*
  then show *P h'*
  proof (*induct rule*: *RTranCl.induct*)
    fix *g* **assume** *P g* **then show** *P g* **.**
  next
    fix *g g' g''*
    assume *IH*: *P g'* ⟹ *P g''*
    assume *g'* ∈ *set*(*succs g*) *P g*
    then have *P g'* **by** (*rule s*)
    then show *P g''* **by** (*rule IH*)
  qed
qed

definition *invariant* :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a list*) ⇒ *bool* **where**
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 *RTranCl-inv*:
invariant P succs $\Longrightarrow$ $(g,g') \in$ *RTranCl succs* $\Longrightarrow$ *P g* $\Longrightarrow$ *P g'*
by (*erule RTranCl-induct*)(*auto simp*:*invariant-def*)

lemma *RTranCl-subset2*:
assumes *a*: (s,g) : *RTranCl f*
shows (!!*g*. $(s,g) \in$ *RTranCl f* $\Longrightarrow$ *set*(*f g*) $\subseteq$ *set*(*h g*)) $\Longrightarrow$ (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

end

10 Plane Graph Enumeration

theory *Plane*
imports *Enumerator FaceDivision RTranCl*
begin

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

declare *maxGon-def* [*simp*]

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

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

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

else containsUnacceptableEdgeSnd N v vs)

definition *containsDuplicateEdge* :: *graph* ⇒ *face* ⇒ *vertex* ⇒ *nat list* ⇒ *bool* **where**
containsDuplicateEdge g f v is ≡
containsUnacceptableEdge (λ*i j*. *duplicateEdge g f* (*f*^{i}•*v*) (*f*^{j}•*v*)) *is*

definition *containsDuplicateEdge*′ :: *graph* ⇒ *face* ⇒ *vertex* ⇒ *nat list* ⇒ *bool* **where**
containsDuplicateEdge′ *g f v is* ≡
2 ≤ |*is*| ∧
((∃ *k* < |*is*| − *2*. *let i0* = *is*!*k*; *i1* = *is*!(*k*+*1*); *i2* = *is*!(*k*+*2*) *in*
(*duplicateEdge g f* (*f*^{i1} •*v*) (*f*^{i2} •*v*)) ∧ (*i0* < *i1*) ∧ (*i1* < *i2*))
∨ (*let i0* = *is*!*0*; *i1* = *is*!*1 in*
(*duplicateEdge g f* (*f*^{i0} •*v*) (*f*^{i1} •*v*)) ∧ (*i0* < *i1*)))

definition *generatePolygon* :: *nat* ⇒ *vertex* ⇒ *face* ⇒ *graph* ⇒ *graph list* **where**
generatePolygon n v f g ≡
let enumeration = *enumerator n* |*vertices f*|;
enumeration = [*is* ← *enumeration*. ¬ *containsDuplicateEdge g f v is*];
vertexLists = [*indexToVertexList f v is*. *is* ← *enumeration*] *in*
[*subdivFace g f vs*. *vs* ← *vertexLists*]

definition *next-plane0* :: *nat* ⇒ *graph* ⇒ *graph list* (*next*′*-plane0*_) **where**
next-plane0$_p$ *g* ≡
if final g then []
else ⋃$_{f∈nonFinals\ g}$ ⋃$_{v∈vertices\ f}$ ⋃$_{i∈[3..<Suc(maxGon\ p)]}$ *generatePolygon i v f g*

definition *Seed* :: *nat* ⇒ *graph* (*Seed*_) **where**
Seed$_p$ ≡ *graph*(*maxGon p*)

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

definition *PlaneGraphs0* :: *graph set* **where**
PlaneGraphs0 ≡ ⋃*p*. {*g*. *Seed*$_p$ [*next-plane0*$_p$]→∗ *g* ∧ *final g*}

end

theory *Plane1*
imports *Plane*
begin

This is an optimized definition of plane graphs and the one we adopt as our

point of reference. In every step only one fixed nonfinal face (the smallest one) and one edge in that face are picked.

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

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

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

definition *PlaneGraphsP* :: *nat* ⇒ *graph set* (*PlaneGraphs*_) **where**
PlaneGraphs$_p$ ≡ {*g*. *Seed*$_p$ [*next-plane*$_p$]→∗ *g* ∧ *final g*}

definition *PlaneGraphs* :: *graph set* **where**
PlaneGraphs ≡ $\bigcup p$. *PlaneGraphs*$_p$

end

11 Properties of Graph Utilities

theory *GraphProps*
imports *Graph*
begin

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

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

lemma *eq-setFinal-iff*[*iff*]: (*f* = *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*

11.1 *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

11.2 *nextVertex*

lemma *nextVertex-in-face'*[*simp*]:
vertices f ≠ [] ⟹ *f* · *v* ∈ $\mathcal{V}$ *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(*fastforce simp*:*neq-Nil-conv*)
done
qed

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

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

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

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

lemma *nextVertex-nth*:
⟦ *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

11.3 $\mathcal{E}$

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

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

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

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

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

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

11.4 Triangles

lemma *vertices-triangle*:
$|vertices\ f| = 3 \Longrightarrow a \in \mathcal{V}\ f \Longrightarrow$
$distinct\ (vertices\ f) \Longrightarrow$
$\mathcal{V}\ f = \{a,\ f \cdot a,\ f \cdot (f \cdot a)\}$
proof −
assume $|vertices\ f| = 3$
then obtain *a1 a2 a3* **where** $vertices\ f = [a1,\ a2,\ a3]$
by (*auto dest*!: *length3D*)
moreover assume $a \in \mathcal{V}\ f$
moreover assume $distinct\ (vertices\ f)$
ultimately show *?thesis*
by (*simp*, *elim disjE*) (*auto simp add*: *nextVertex-def*)
qed

lemma *tri-next3-id*:
$|vertices\ f| = 3 \Longrightarrow distinct(vertices\ f) \Longrightarrow v \in \mathcal{V}\ f$
$\Longrightarrow f \cdot (f \cdot (f \cdot v)) = v$
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*:
$|vertices\ f| = 3 \Longrightarrow a \in \mathcal{V}\ f \Longrightarrow$
$distinct\ (vertices\ f) \Longrightarrow$
$f \cdot (f \cdot a) = f^{-1} \cdot a$
proof −
assume $|vertices\ f| = 3$
then obtain *a1 a2 a3* **where** $vertices\ f = [a1,\ a2,\ a3]$
by (*auto dest*!:*length3D*)
moreover assume $a \in \mathcal{V}\ f$
moreover assume $distinct\ (vertices\ f)$
ultimately show *?thesis*
by (*simp*, *elim disjE*) (*auto simp add*: *nextVertex-def prevVertex-def*)
qed

11.5 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 *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

11.6 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

11.7 *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*:
$[\![\ distinct(vertices\ f);\ v \in \mathcal{V}\ f;\ u \in \mathcal{V}\ f;\ u \neq v\ ]\!] \Longrightarrow$
$v \in set(between\ (vertices\ f)\ u\ (f \cdot v))$
apply(*frule split-list*[*of u*])
apply(*clarsimp*)
apply(*erule disjE*)
apply(*clarsimp simp*: *nextVertex-def hd-append split*:*list.split*)
apply(*rule conjI*)
apply(*clarsimp*)
apply(*frule split-list*[*of v*])
apply(*clarsimp simp*: *between-def split-def split*:*list.split*)
apply(*fastforce simp*: *append-eq-Cons-conv*)
apply(*frule split-list*[*of v*])
apply(*clarsimp simp*: *between-def split-def nextVertex-def split*:*list.splits*)
apply(*clarsimp simp*: *hd-append*)
apply(*erule disjE*)
apply(*clarsimp*)
apply(*frule split-list*)
apply(*fastforce*)
done

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

lemma *unroll-between-next2*:
⟦ *distinct*(*vertices f*); *u* $\in$ $\mathcal{V}$ *f*; *v* $\in$ $\mathcal{V}$ *f*; *u* $\neq$ *v* ⟧ $\Longrightarrow$
between (*vertices f*) *u* (*f* $\cdot$ *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* $\in$ $\mathcal{V}$ *f*; *y* $\in$ $\mathcal{V}$ *f*; *x* $\neq$ *y*;
v $\in$ *set*(*x* # *between* (*vertices f*) *x y*) ⟧ $\Longrightarrow$
f $\cdot$ *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(*fastforce simp*:*append-eq-Cons-conv*)
apply(*drule split-list*)
apply(*clarsimp simp add*:*between-def split-def hd-append split*:*list.split*)
apply(*fastforce simp*:*append-eq-Cons-conv*)
apply(*clarsimp*)
apply(*erule disjE*)
apply(*drule split-list*[*of y*])
apply(*clarsimp simp*:*between-def split-def*)
apply(*erule disjE*)
apply(*drule split-list*[*of v*])
apply(*fastforce simp*: *hd-append neq-Nil-conv split*:*list.split*)
apply(*drule split-list*[*of v*])
apply(*clarsimp*)
apply(*clarsimp simp*: *hd-append split*:*list.split*)
apply(*fastforce simp*:*append-eq-Cons-conv*)
apply(*drule split-list*[*of y*])
apply(*clarsimp simp*:*between-def split-def*)

apply(*drule split-list*[*of v*])
apply(*clarsimp*)
apply(*clarsimp simp*: *hd-append split*:*list.split*)
apply(*clarsimp simp*:*append-eq-Cons-conv*)
apply(*fastforce simp*: *hd-append neq-Nil-conv split*:*list.split*)
done

end

12 Properties of Patch Enumeration

theory *EnumeratorProps*
imports *Enumerator GraphProps*
begin

lemma *length-hideDupsRec*[*simp*]: $\bigwedge x.$ *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*)

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

lemma *increasing1*: $\bigwedge$ *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-bound*: ∀ *y* ∈ *set* (*enumBase nmax*). ∀ *z* ∈ *set y*. *z* <= *nmax*
by (*auto simp*: *enumBase-def*)

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

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

lemma *enumAppend-bound-rec*: $ls \in set\ (((enumAppend\ nmax)\ \hat{}\hat{}\ n)\ lss) \Longrightarrow$
$\forall\ y \in set\ lss.\ \forall\ z \in set\ y.\ z \leq nmax \Longrightarrow x \in set\ ls \Longrightarrow x \leq\ nmax$
proof −
assume *ls*: $ls \in set\ ((enumAppend\ nmax\ \hat{}\hat{}\ n)\ lss)$ **and** *lss*: $\forall y{\in}set\ lss.\ \forall z{\in}set\ y.\ z \leq nmax$ **and** *x*: $x \in set\ ls$
have *ind*:$\bigwedge lss.\ \forall y{\in}set\ lss.\ \forall z{\in}set\ y.\ z \leq nmax \Longrightarrow \forall\ y \in set\ (((enumAppend\ nmax)\ \hat{}\hat{}\ n)\ lss).\ \forall\ z \in set\ y.\ z \leq nmax$
proof (*induct n*)
case *0* **then show** *?case* **by** *auto*
next
case (*Suc n*) **show** *?case* **apply** (*intro ballI*) **apply** (*rule enumAppend-bound*)
by (*auto intro*!: *Suc*)
qed
with *lss* **have** $\forall\ y \in set\ (((enumAppend\ nmax)\ \hat{}\hat{}\ n)\ lss).\ \forall\ z \in set\ y.\ z \leq nmax$ **apply** (*rule-tac ind*) **.**
with *ls x* **show** *?thesis* **by** *auto*
qed

lemma *enumAppend-increase-rec*:
$\bigwedge m\ as\ bs.\ ls \in set\ (((enumAppend\ nmax)\ \hat{}\hat{}\ m)\ (enumBase\ nmax)) \Longrightarrow$
$as\ @\ bs = ls \Longrightarrow\ \forall\ x \in set\ as.\ \forall\ y \in set\ bs.\ x \leq y$
apply (*induct ls rule*: *rev-induct*) **apply** *force* **apply** *auto* **apply** (*case-tac m*)
apply *simp* **apply** (*drule-tac enumBase-length*)
apply (*case-tac as*) **apply** *simp-all*
proof −
fix *x xs m as bs xa xb n*
assume *ih*: $\bigwedge m\ as\ bs.$
$[\![xs \in set\ ((enumAppend\ nmax\ \hat{}\hat{}\ m)\ (enumBase\ nmax));\ as\ @\ bs = xs]\!]$
$\Longrightarrow \forall x{\in}set\ as.\ \forall xa{\in}set\ bs.\ x \leq xa$
and *xs*:$xs\ @\ [x] \in set\ (enumAppend\ nmax\ ((enumAppend\ nmax\ \hat{}\hat{}\ n)\ (enumBase\ nmax)))$
and *asbs*: $as\ @\ bs = xs\ @\ [x]$ **and** *xa*:$xa \in set\ as$ **and** *xb*: $xb \in set\ bs$ **and** *m*: $m = Suc\ n$
from *ih* **have** *ih2*: $\bigwedge as\ bs\ x\ y.\ [\![xs \in set\ ((enumAppend\ nmax\ \hat{}\hat{}\ n)\ (enumBase\ nmax));\ as\ @\ bs = xs;\ x \in set\ as;\ y \in set\ bs]\!]$
$\Longrightarrow x \leq y$ **by** *auto*

from *xb* **have** $bs \neq []$ **by** *auto*
then obtain *bs' b* **where** *bs'*: $bs = bs'\ @\ [b]$ **apply** (*cases rule*: *rev-exhaust*) **by** *auto*
with *asbs* **have** *beq*:$b = x$ **by** *auto*
from *bs' asbs* **have** *xs'*: $as\ @\ bs' = xs$ **by** *auto*

```
  with xs have xa ≤ 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)
```

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* $2 \leq$ *length* (*ys* @ [*y*])) **apply** *simp*
apply (*rule-tac enumerator-length*) **by** *auto*

lemma *enumerator-increase*: $\bigwedge$ *as bs. ls* $\in$ *set* (*enumerator m nmax*) $\Longrightarrow$ *as* @ *bs* = *ls* $\Longrightarrow \forall$ *x* $\in$ *set as.* $\forall$ *y* $\in$ *set bs. x* $\leq$ *y*
apply (*auto simp*: *enumerator-def del*: *Nat.diff-is-0-eq*′ *split*: *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*)
by (*auto dest*!: *enumAppend-increase-rec*)

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

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

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

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

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

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

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

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

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

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

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

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

lemma *incrIndexList-help8*[*simp*]: *incrIndexList ls m nmax* $\Longrightarrow$ *butlast ls* $\neq$ []
proof (*rule ccontr*)
assume *props*: *incrIndexList ls m nmax* **and** *butl*: $\neg$ *butlast ls* $\neq$ []
then have *ls* $\neq$ [] **by** *auto*
then have *ls'*: *ls* = (*butlast ls*) @ [*last ls*] **by** *auto*
def *l* $\equiv$ *last ls*
with *butl ls'* **have** *ls* = [*l*] **by** *auto*
then have *length ls* = *1* **by** *auto*
with *props* **have** *m* = *1* **by** *auto*
with *props* **show** *False* **by** (*auto dest*: *incrIndexList-1lem*)
qed

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

lemma *incrIndexList-help9*[*intro*]: (*incrIndexList ls m nmax*) $\Longrightarrow$
x $\in$ *set* (*butlast ls*) $\Longrightarrow$ *x* $\leq$ *nmax* $-$ *2*
proof $-$
assume *props*: (*incrIndexList ls m nmax*) **and** *x*: *x* $\in$ *set* (*butlast ls*)
then have *last* (*butlast ls*) $<$ *last ls* **by** *auto*
with *props* **have** *last* (*butlast ls*) $<$ *nmax* $-$ *1* **by** *auto*
then have *leq*: *last* (*butlast ls*) $\leq$ *nmax* $-$ *2* **by** *arith*
from *props* **have** *ls* $\neq$ [] **by** *auto*
then have *ls1*: *ls* = *butlast ls* @ [*last ls*] **by** *auto*
def *ls'* $\equiv$ (*butlast* (*butlast ls*))
def *last2* $\equiv$ *last* (*butlast ls*)
def *last1* $\equiv$ *last ls*
from *props* **have** *butlast ls* $\neq$ [] **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* $\in$ *set ls'* $\Longrightarrow$ *x* $\leq$ *last2* **by** (*auto intro*: *increasing2*)
then have *x* $\in$ *set* (*ls'* @ [*last2*]) $\Longrightarrow$ *x* $\leq$ *last2* **by** *auto*
with *bls x* **have** *x* $\leq$ *last2* **by** *auto*
with *leq last2-def* **show** *?thesis* **by** *auto*
qed

lemma *incrIndexList-help10*[*intro*]: (*incrIndexList ls m nmax*) $\Longrightarrow$
x $\in$ *set ls* $\Longrightarrow$ *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*: $2 < m \Longrightarrow 1 < nmax \Longrightarrow$
$ls \in set\ (enumerator\ m\ nmax) \Longrightarrow$
incrIndexList ls m nmax
proof −
assume *m*: $2 < m$ **and** *nmax*: $1 < nmax$ **and** *enum*: $ls \in set\ (enumerator\ m\ nmax)$
then have $(hd\ ls = 0 \wedge last\ ls = (nmax - 1) \wedge length\ ls = m \wedge last\ (butlast\ ls) < last\ ls \wedge\ increasing\ ls)$
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*: $\bigwedge ls.\ increasing\ ls \Longrightarrow ls \neq [] \Longrightarrow length\ ls = Suc\ ks \Longrightarrow list\text{-}all\ (\lambda x.\ x < Suc\ nmax)\ ls \Longrightarrow ls \in set\ ((enumAppend\ nmax\ @@\ ks)\ (enumBase\ nmax))$
proof (*induct ks*)
case *0*
assume *increasing ls* $ls \neq []$ *length ls* = *Suc 0 list-all* $(\lambda x.\ x < Suc\ nmax)$ *ls*
then have $\exists\ x.\ ls = [x]$
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 $ls' \equiv butlast\ ls$
def $l \equiv last\ ls$
def $ll \equiv last\ ls'$
def $bl \equiv butlast\ ls'$
def $ls'list \equiv (enumAppend\ nmax\ @@\ n)\ (enumBase\ nmax)$
then have *short*: $(enumAppend\ nmax\ @@\ n)\ (enumBase\ nmax) = ls'list$ **by** *simp*
from *Suc* **have** $ls \neq []$ **by** *auto*
then have $ls = butlast\ ls\ @\ [last\ ls]$ **by** *auto*
with *ls'-def l-def* **have** *ls1*: $ls = ls'\ @\ [l]$ **by** *auto*
with *Suc* **have** $length\ ls' = Suc\ n$ **by** *auto*
then have *ls'ne*: $ls' \neq []$ **by** *auto*
with *ll-def bl-def* **have** *ls'1*: $ls' = bl\ @\ [ll]$ **by** *auto*
then have *ll-in-ls'*: $ll \in set\ ls'$ **by** *simp*
from *Suc ls1* **have** *list-all* $(\lambda x.\ x < Suc\ nmax)\ ls'$ **by** *auto*
with *ll-in-ls'* **have** $ll < Suc\ nmax$ **by** (*induct ls'*) *auto*
with *ll-def* **have** *llsmall*: $last\ ls' \leq nmax$ **by** *auto*

from *ls1* **have** *l-in-ls*: $l \in set\ ls$ **by** *auto*
from *Suc* **have** *list-all* $(\lambda x.\ x < Suc\ nmax)\ ls$ **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* (λ*x*. *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

13 Properties of Face Division

theory *FaceDivisionProps*
imports *Plane EnumeratorProps*
begin

13.1 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 *in-set-repl-setFin*:
$f \in set\ fs \Longrightarrow final\ f \Longrightarrow f \in set\ (replace\ f'\ [setFinal\ f']\ fs)$
by (*induct fs*) *auto*

lemma *in-set-repl*: $f \in set\ fs \Longrightarrow f \neq f' \Longrightarrow f \in set\ (replace\ f'\ fs'\ fs)$
by (*induct fs*) *auto*

lemma *makeFaceFinals-preserve-finals*:
$f \in set\ (finals\ g) \Longrightarrow f \in set\ (finals\ (makeFaceFinal\ f'\ g))$
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*:
$f \in \mathcal{F}\ g \Longrightarrow \neg\ final\ f \Longrightarrow |finals\ (makeFaceFinal\ f\ g)| = |finals\ g| + 1$
by(*simp add*:*makeFaceFinal-def finals-def makeFaceFinalFaceList-def*
length-filter-replace1)

lemma *len-nonFinals-makeFaceFinal*:
$[\![\neg\ final\ f; f \in \mathcal{F}\ g]\!]$
$\Longrightarrow |nonFinals\ (makeFaceFinal\ f\ g)| = |nonFinals\ g| - 1$
by(*simp add*:*makeFaceFinal-def nonFinals-def makeFaceFinalFaceList-def*
length-filter-replace2)

lemma *set-finals-makeFaceFinal*[*simp*]: $distinct(faces\ g) \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow$
$set(finals\ (makeFaceFinal\ f\ g)) = insert\ (setFinal\ f)\ (set(finals\ g))$
by(*auto simp*:*finals-def makeFaceFinal-def makeFaceFinalFaceList-def*
distinct-set-replace)

lemma *splitFace-preserve-final*:
$f \in set\ (finals\ g) \Longrightarrow \neg\ final\ f' \Longrightarrow$
$f \in set\ (finals\ (snd\ (snd\ (splitFace\ g\ i\ j\ f'\ ns))))$
by (*induct g*) (*auto simp add*: *splitFace-def finals-def split-def*
intro: *in-set-repl*)

lemma *splitFace-nonFinal-face*:
$\neg\ final\ (fst\ (snd\ (splitFace\ g\ i\ j\ f'\ ns)))$
by (*simp add*: *splitFace-def split-def split-face-def*)

lemma *subdivFace'-preserve-finals*:

$\bigwedge n\ i\ f'\ g.\ f \in set\ (finals\ g) \Longrightarrow \neg\ final\ f' \Longrightarrow$
$f \in set\ (finals\ (subdivFace'\ g\ f'\ i\ n\ is))$
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*:
$f \in set\ (finals\ g) \Longrightarrow \neg\ final\ f' \Longrightarrow$
$f \in set\ (finals\ (subdivFace\ g\ f'\ is))$
by(*simp add*:*subdivFace-def subdivFace'-preserve-finals*)

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

13.2 *is-prefix*

definition *is-prefix* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
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*)

13.3 *is-sublist*

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

lemma *is-prefix-sublist*:

is-prefix ls vs $\Longrightarrow$ *is-sublist ls vs* **by** (*auto simp*: *is-prefix-def is-sublist-def*)

lemma *is-sublist-trans*: *is-sublist as bs* $\Longrightarrow$ *is-sublist bs cs* $\Longrightarrow$ *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* $\Longrightarrow$ *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


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* $\Longrightarrow$ *y* $\in$ *set vs* **by** (*auto simp*: *is-sublist-def*)

lemma *is-sublist-notlast*[*simp*]: *distinct vs* $\Longrightarrow$ *x* = *last vs* $\Longrightarrow$ $\neg$ *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*: *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** *vs2*: *vs* = *as* @ *bs* **using** *vs* **by** *auto*
with *as-def* **have** *xas*: *x* ∈ *set as* **by** *auto*
from *bsne vs2* **have** *last vs* = *last bs* **by** *auto*
with *xl* **have** *x* = *last bs* **by** *auto*
with *bsne* **have** *bs* = (*butlast bs*) @ [*x*] **by** *auto*
then have *x* ∈ *set bs* **by** (*induct bs*) *auto*
with *xas vs2 dvs* **show** *False* **by** *auto*
qed

lemma *is-sublist-nth1*: *is-sublist* [*x*,*y*] *ls* $\Longrightarrow$
∃ *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* $\Longrightarrow$
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-tl*: *is-sublist* (*a* # *as*) *vs* $\Longrightarrow$ *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-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* **using** *vs* **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-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-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-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*

13.4 *is-nextElem*

definition *is-nextElem* :: 'a *list* ⇒ 'a ⇒ 'a ⇒ *bool* **where**
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-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-sublistI*: *is-sublist* [*x*,*y*] *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 $\exists\ i\ j.\ i < length\ ls \wedge j < length\ ls \wedge ls!i = x \wedge ls!j = y \wedge (Suc\ i)\ mod\ (length\ ls) = j$
then obtain *i j* **where** *vors*: $i < length\ ls \wedge j < length\ ls \wedge ls!i = x \wedge ls!j = y \wedge (Suc\ i)\ mod\ (length\ ls) = j$ **by** *auto*
then show *?thesis*
proof (*cases Suc i = length ls*)
case *True* **with** *vors* **have** $j = 0$ **by** *auto*

with *True vors* **show** *?thesis* **apply** (*auto simp*: *is-nextElem-def*)
apply (*cases ls rule*: *rev-exhaust*) **apply** *auto* **apply** (*cases ls*) **by** *auto*
next
case *False* **with** *vors* **have** *is-sublist* [*x*,*y*] *ls*
apply (*rule-tac is-sublist-nth2*) **by** *auto*
then show *?thesis* **by** (*simp add*: *is-nextElem-def*)
qed
qed

lemma *is-nextElem-rotate1-aux*:
is-nextElem (*rotate m ls*) *x y* $\Longrightarrow$ *is-nextElem ls x y*
proof −
assume *is-nextElem*: *is-nextElem* (*rotate m ls*) *x y*
def $n \equiv m\ mod\ length\ ls$
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*:$i < length\ (rotate\ n\ ls) \wedge j < length\ (rotate\ n\ ls) \wedge$
$(rotate\ n\ ls)!i = x \wedge (rotate\ n\ ls)!j = y \wedge$
$(Suc\ i)\ mod\ (length\ (rotate\ n\ ls)) = j$ **by** (*drule-tac is-nextElem-nth1*) *auto*
then have *lls*: $0 < length\ ls$ **by** *auto*
def $k \equiv (i+n)\ mod\ (length\ ls)$
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 $l \equiv (j+n)\ mod\ (length\ ls)$
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'*: $\bigwedge\ a\ b\ c.\ (a\ mod\ c + b\ mod\ c)\ mod\ c = (a+b)\ mod\ (c::nat)$
apply (*rule sym*) **by** (*rule mod-add-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*]: $is\text{-}nextElem\ (rotate\ m\ ls)\ x\ y = is\text{-}nextElem\ ls\ x\ y$
apply (*auto dest*: *is-nextElem-rotate1-aux*) **apply** (*rule is-nextElem-rotate1-aux*)
apply (*subgoal-tac* $is\text{-}nextElem\ (rotate\ (length\ ls - m\ mod\ length\ ls)\ (rotate\ m\ ls))\ x\ y$)
apply *assumption* **by** *simp*

lemma *is-nextElem-congs-eq*: $ls \cong ms \Longrightarrow is\text{-}nextElem\ ls\ x\ y = is\text{-}nextElem\ ms\ x\ y$
by (*auto simp*: *congs-def*)

lemma *is-nextElem-rev*[*simp*]: $is\text{-}nextElem\ (rev\ zs)\ a\ b = is\text{-}nextElem\ zs\ b\ a$
apply (*simp add*: *is-nextElem-def is-sublist-rev*)
apply (*case-tac* $zs = []$) **apply** *simp* **apply** *simp*
apply (*case-tac* $a = hd\ zs$) **apply** (*case-tac zs*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac* $a = last\ (rev\ zs) \wedge b = last\ zs$) **apply** *simp*
apply (*case-tac zs rule*: *rev-exhaust*) **apply** *simp*
apply (*case-tac ys*) **apply** *simp* **apply** *simp* **by** *force*

lemma *is-nextElem-circ*:
$[\![\ distinct\ xs;\ is\text{-}nextElem\ xs\ a\ b;\ is\text{-}nextElem\ xs\ b\ a\ ]\!] \Longrightarrow |xs| \leq 2$
apply(*drule is-nextElem-nth1*)
apply(*drule is-nextElem-nth1*)
apply (*clarsimp*)
apply(*rename-tac i j*)
apply(*frule-tac i=j* **and** $j = Suc\ i\ mod\ |xs|$ **in** *nth-eq-iff-index-eq*)
apply *assumption+*
apply(*frule-tac j=i* **and** $i = Suc\ j\ mod\ |xs|$ **in** *nth-eq-iff-index-eq*)
apply *assumption+*
apply(*rule ccontr*)
apply(*simp add*: *distinct-conv-nth mod-Suc*)
done

13.5 *nextElem*, *sublist*, *is-nextElem*

lemma *is-sublist-eq*: *distinct vs* $\Longrightarrow$ $c \neq y \Longrightarrow$
(*nextElem vs c x* = *y*) = *is-sublist* [*x*,*y*] *vs*
proof −
assume *d*: *distinct vs* **and** *c*: $c \neq y$
have *r1*: *nextElem vs c x* = *y* $\Longrightarrow$ *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* $\Longrightarrow$ *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* $\Longrightarrow$ *x* $\in$ *set vs* $\Longrightarrow$ *nextElem vs* (*hd vs*) *x* = *y*
$\Longrightarrow$ *is-nextElem vs x y*
proof −
assume *d*: *distinct vs* **and** *x*: *x* $\in$ *set vs* **and** *fn*: *nextElem vs* (*hd vs*) *x* = *y*
from *x* **have** *r0*: *vs* $\neq$ [] **by** *auto*
from *d fn* **have** *r1*: *x* = *last vs* $\Longrightarrow$ *y* = *hd vs* **by** (*auto*)
from *d fn* **have** *r3*: *hd vs* $\neq$ *y* $\Longrightarrow$ ($\exists$ *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* $\neq$ *last vs* $\Longrightarrow$ *y* $\neq$ *hd vs*
proof −
assume *notx*: *x* $\neq$ *last vs*
from *vs notx* **have** *bs* $\neq$ [] **by** *auto*
with *vs* **have** *r2*: *vs* = *as* @ [*x*, *hd bs*] @ *tl bs* **by** *auto*
with *d* **have** *ineq*: *hd bs* $\neq$ *hd vs* **by** (*cases as*) *auto*
from *d fn r2* **have** *y* = *hd bs* **by** *auto*
with *ineq* **show** *?thesis* **by** *auto*
qed
from *r0 r1 r2 r3* **show** *?thesis* **apply** (*simp add*:*is-nextElem-def is-sublist-def*)
apply (*cases x* = *last vs*) **by** *auto*
qed

lemma *is-nextElem2*: *distinct vs* $\Longrightarrow$ *x* $\in$ *set vs* $\Longrightarrow$ *is-nextElem vs x y* $\Longrightarrow$ *nextElem vs* (*hd vs*) *x* = *y*
proof −
assume *d*: *distinct vs* **and** *x*: *x* $\in$ *set vs* **and** *is-nextElem*: *is-nextElem vs x y*
then show *?thesis* **apply** (*simp add*: *is-nextElem-def*) **apply** (*cases is-sublist* [*x*,*y*] *vs*)
apply (*cases y* = *hd vs*)

apply (*simp add*: *is-sublist-def*) **apply** (*force dest*: *distinct-hd-not-cons*)
apply (*subgoal-tac hd vs* $\neq$ *y*) **apply** (*simp add*: *is-sublist-eq*) **by** *auto*
qed

lemma *nextElem-is-nextElem*:
distinct xs $\Longrightarrow$ *x* $\in$ *set xs* $\Longrightarrow$
is-nextElem xs x y = (*nextElem xs* (*hd xs*) *x* = *y*)
by (*auto intro*!: *is-nextElem1 is-nextElem2*)

lemma *nextElem-congs-eq*: *xs* $\cong$ *ys* $\Longrightarrow$ *distinct xs* $\Longrightarrow$ *x* $\in$ *set xs* $\Longrightarrow$
nextElem xs (*hd xs*) *x* = *nextElem ys* (*hd ys*) *x*
proof −
assume *eq*: *xs* $\cong$ *ys* **and** *dist*: *distinct xs* **and** *x*: *x* $\in$ *set xs*
def *y* $\equiv$ *nextElem xs* (*hd xs*) *x*
then have *f1*:*nextElem xs* (*hd xs*) *x* = *y* **by** *auto*
with *dist x* **have** *is-nextElem xs x y* **by** (*auto intro*: *is-nextElem1*)
with *eq* **have** *is-nextElem ys x y* **by** (*simp add*:*is-nextElem-congs-eq*)
with *eq dist x* **have** *f2*:*nextElem ys* (*hd ys*) *x* = *y*
by (*auto simp*: *congs-distinct intro*: *is-nextElem2*)
from *f1 f2* **show** *?thesis* **by** *auto*
qed

lemma *is-sublist-is-nextElem*: *distinct vs* $\Longrightarrow$ *is-nextElem vs x y* $\Longrightarrow$ *is-sublist as vs* $\Longrightarrow$ *x* $\in$ *set as* $\Longrightarrow$ *x* $\neq$ *last as* $\Longrightarrow$ *is-sublist* [*x*,*y*] *as*
proof −
assume *d*: *distinct vs* **and** *is-nextElem*: *is-nextElem vs x y* **and** *subl*: *is-sublist as vs* **and** *xin*: *x* $\in$ *set as* **and** *xnl*: *x* $\neq$ *last as*
from *xin* **have** *asne*: *as* $\neq$ [] **by** *auto*
with *subl* **have** *vsne*: *vs* $\neq$ [] **by** (*auto simp*: *is-sublist-def*)
from *subl* **obtain** *rs ts* **where** *vs*: *vs* = *rs* @ *as* @ *ts* **apply** (*simp add*: *is-sublist-def*) **apply** (*elim exE*) **by** *auto*
with *d xnl asne* **have** *x* $\neq$ *last vs*
proof (*cases ts* = [])
case *True* **with** *d xnl asne vs* **show** *?thesis* **by** *force*
next
def *lastvs* $\equiv$ *last ts*
case *False*
with *vs lastvs-def* **have** *vs2*: *vs* = *rs* @ *as* @ *butlast ts* @ [*lastvs*] **by** *auto*
with *d* **have** *lastvs* $\notin$ *set as* **by** *auto*
with *xin* **have** *lastvs* $\neq$ *x* **by** *auto*
with *vs2* **show** *?thesis* **by** *auto*
qed
with *is-nextElem* **have** *subl-vs*: *is-sublist* [*x*,*y*] *vs* **by** (*auto simp*: *is-nextElem-def*)
from *d xin vs* **have** ¬ *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* $\notin$ *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

13.6 *before*

definition *before* :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *bool* **where**
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* ⟹ *distinct vs* ⟹ *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* ⟹ *distinct vs* ⟹ *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* ⟹ *distinct vs* ⟹ *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* ⟹ *distinct vs* ⟹ *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* ∈ *set vs* ⟹ *ram2* ∈ *set vs* ⟹ *ram1* ≠ *ram2* ⟹ *before vs ram1 ram2* ∨ *before vs ram2 ram1*
proof −
assume *r1*: *ram1* ∈ *set vs* **and** *r2*: *ram2* ∈ *set vs* **and** *r12*: *ram1* ≠ *ram2*
then show *?thesis*
proof (*cases ram2* ∈ *set* (*snd* (*splitAt ram1 vs*)))
def *a* ≡ *fst* (*splitAt ram1 vs*)
def *b* ≡ *fst* (*splitAt ram2* (*snd* (*splitAt ram1 vs*)))
def *c* ≡ *snd* (*splitAt ram2* (*snd* (*splitAt ram1 vs*)))
case *True* **with** *r1 a-def b-def c-def* **have** *vs* = *a* @ [*ram1*] @ *b* @ [*ram2*] @ *c*
by (*auto dest*!: *splitAt-ram*)
then show *?thesis* **apply** (*simp add*: *before-def*) **by** *auto*
next
def *ab* ≡ *fst* (*splitAt ram1 vs*)
case *False* **with** *r1 r2 r12 ab-def* **have** *r2'*: *ram2* ∈ *set ab* **by** (*auto intro*: *splitAt-ram3*)
def *a* ≡ *fst* (*splitAt ram2 ab*)
def *b* ≡ *snd* (*splitAt ram2 ab*)
def *c* ≡ *snd* (*splitAt ram1 vs*)
from *r1 ab-def c-def* **have** *vs* = *ab* @ [*ram1*] @ *c* **by** (*auto dest*!: *splitAt-ram*)
with *r2' a-def b-def* **have** *vs* = (*a* @ [*ram2*] @ *b*) @ [*ram1*] @ *c* **by** (*drule-tac splitAt-ram*) *simp*
then show *?thesis* **apply** (*simp add*: *before-def*) **apply** (*rule disjI2*) **by** *auto*
qed
qed

lemma *before-r1*:
before vs r1 r2 ⟹ *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-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 $t \equiv (a,bc)$
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*]:
$distinct\ vs \Longrightarrow before\ vs\ r1\ r2 \Longrightarrow r2 \notin set\ (fst\ (splitAt\ r1\ vs))$ **apply** (*frule before-dist-r2*) **by** (*auto dest*: *splitAt-distinct-fst-snd*)

lemma *before-dist-r1*:
$distinct\ vs \Longrightarrow before\ vs\ r1\ r2 \Longrightarrow r1 \in set\ (fst\ (splitAt\ r2\ vs))$
proof −
assume *d*: *distinct vs* **and** *b*: *before vs r1 r2*
from *d b* **have** *ex1*: $\exists!\ s.\ (vs = (fst\ s)\ @\ r2\ \#\ snd\ (s))$ **apply** (*drule-tac before-r2*) **apply** (*rule distinct-unique1*) **by** *auto*
from *d b ex1* **show** *?thesis* **apply** (*unfold before-def*)
proof (*elim exE ex1E*)
fix *a b c s*
assume *vs*: $vs = a\ @\ r1\ \#\ b\ @\ r2\ \#\ c$ **and** $\forall y.\ vs = fst\ y\ @\ r2\ \#\ snd\ y \longrightarrow y = s$
then have $\bigwedge y.\ vs = fst\ y\ @\ r2\ \#\ snd\ y \longrightarrow y = s$ **by** *auto*
then have *single*: $\bigwedge y.\ vs = fst\ y\ @\ r2\ \#\ snd\ y \Longrightarrow y = s$ **by** *auto*
def $ab \equiv a\ @\ r1\ \#\ b$
with *vs* **have** *vs2*: $vs = ab\ @\ r2\ \#\ c$ **by** *auto*
from *ab-def* **have** *r1*: $r1 \in set\ ab$ **by** *auto*
def $t \equiv (ab,c)$
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*]:
$distinct\ vs \Longrightarrow before\ vs\ r1\ r2 \Longrightarrow r1 \notin set\ (snd\ (splitAt\ r2\ vs))$ **apply** (*frule before-dist-r1*) **by** (*auto dest*: *splitAt-distinct-fst-snd*)

lemma *before-snd*:
$r2 \in set\ (snd\ (splitAt\ r1\ vs)) \Longrightarrow before\ vs\ r1\ r2$
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 *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

13.7 *between*

definition *pre-between* :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *bool* **where**
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-symI*:
pre-between vs ram1 ram2 $\Longrightarrow$ *pre-between vs ram2 ram1* **by** *auto*

lemma *pre-between-before*[*dest*]:
pre-between vs ram1 ram2 $\Longrightarrow$ *before vs ram1 ram2* $\vee$ *before vs ram2 ram1* **by** (*rule-tac before-or*) *auto*

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

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

lemma *pre-between vs ram1 ram2* $\Longrightarrow$ ($\neg$ *before vs ram1 ram2*) = *before vs ram2 ram1*
by (*simp add*: *before-dist-eq*)

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

lemma *between-simp1*[*simp*]:
before vs ram1 ram2 $\Longrightarrow$ *pre-between vs ram1 ram2* $\Longrightarrow$
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 $\Longrightarrow$ *pre-between vs ram1 ram2* $\Longrightarrow$
between vs ram2 ram1 = *snd* (*splitAt ram2 vs*) @ *fst* (*splitAt ram1 vs*)
proof −
assume *b*: *before vs ram1 ram2* **and** *p*: *pre-between vs ram1 ram2*
from *p b* **have** *b2*: $\neg$ *before vs ram2 ram1* **apply** (*simp add*: *pre-between-def*)
by (*auto dest*: *before-dist-not1*)
with *p* **have** *ram2* $\notin$ *set* (*fst* (*splitAt ram1 vs*)) **by** (*simp add*: *pre-between-def before-dist-eq-fst*)
then have *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 $\Longrightarrow$ *ram1* $\notin$ *set* (*between vs ram1 ram2*)
proof (*cases pre-between vs ram1 ram2*)
assume *d*: *distinct vs*
case *True* **then have** *p*: *pre-between vs ram1 ram2* **by** *auto*
then show *ram1* $\notin$ *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** *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 *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*)

13.7.1 *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

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 (*simp split*: *split-if-asm*)
apply (*case-tac b* = *aa*)
apply (*simp add*: *is-sublist-def*)
apply (*erule disjE*)
apply (*elim exE*)
apply (*case-tac as*)
apply *simp*
apply *simp*
apply *simp*
apply (*case-tac list rule*: *rev-exhaust*)
apply *simp*
apply *simp*
apply *simp*
apply (*subgoal-tac aa # list* = *vs*)
apply (*thin-tac vs* = *aa # list*)
apply *simp*

apply (*subgoal-tac distinct vs*)
apply (*simp add*: *is-sublist-def*)
apply (*elim exE*)
by *auto*

lemma *is-nextElem-between-empty'*: *between vs a b* = [] ⟹ *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* $\Longrightarrow$
between vs u (*nextElem vs* (*hd vs*) *v*) = *between vs u v* @ [*v*]
apply(*subgoal-tac pre-between vs v u*)
prefer *2* **apply** (*clarsimp simp add*:*pre-between-def*)
apply(*case-tac before vs u v*)
apply(*drule* (*1*) *between-eq2*)
apply(*clarsimp simp*:*pre-between-def hd-append split*:*list.split*)
apply(*simp add*:*between-def split-def*)
apply(*fastforce simp*:*neq-Nil-conv*)
apply (*simp only*:*before-xor*)
apply(*drule* (*1*) *between-eq2*)
apply(*clarsimp simp*:*pre-between-def hd-append split*:*list.split*)
apply (*simp add*: *append-eq-Cons-conv*)
apply(*fastforce simp*:*between-def split-def*)
done

lemma *nextVertices-in-face*[*simp*]: $v \in \mathcal{V}\ f \Longrightarrow f^n \cdot v \in \mathcal{V}\ f$
proof −
assume *v*: $v \in \mathcal{V}\ f$
then have *f*: *vertices f* $\neq$ [] **by** *auto*
show *?thesis* **apply** (*simp add*: *nextVertices-def*)
apply (*induct n*) **by** (*auto simp*: *f v*)
qed

13.7.2 *is-nextElem edges* equivalence

lemma *is-nextElem-edges1*: *distinct* (*vertices f*) $\Longrightarrow$ (*a*,*b*) $\in$ *edges* (*f*::*face*) $\Longrightarrow$ *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*) $\Longrightarrow$ *is-nextElem* (*vertices f*) *a b* $\Longrightarrow$
(*a*,*b*) $\in$ *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*)) $\Longrightarrow$
(*a*,*b*) $\in$ *edges f* = *is-nextElem* (*vertices f*) *a b*
by (*auto intro*: *is-nextElem-edges1 is-nextElem-edges2*)

13.7.3 *nextVertex*

lemma *nextElem-suc2*: *distinct* (*vertices f*) $\Longrightarrow$ *last* (*vertices f*) = *v* $\Longrightarrow$ *v* $\in$ *set* (*vertices f*) $\Longrightarrow$ *f* $\cdot$ *v* = *hd* (*vertices f*)
proof −
assume *dist*: *distinct* (*vertices f*) **and** *last*: *last* (*vertices f*) = *v* **and** *v*: *v* $\in$ *set* (*vertices f*)
def *ls* $\equiv$ *vertices f*
have *ind*: $\bigwedge$ *c*. *distinct ls* $\Longrightarrow$ *last ls* = *v* $\Longrightarrow$ *v* $\in$ *set ls* $\Longrightarrow$ *nextElem ls c v* = *c*
proof (*induct ls*)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons m ms*)
then show *?case*
proof (*cases m* = *v*)
case *True* **with** *Cons* **have** *ms* = [] **apply** (*cases ms rule*: *rev-exhaust*) **by** *auto*
then show *?thesis* **by** *auto*
next
case *False* **with** *Cons* **have** *a1*: *v* $\in$ *set ms* **by** *auto*
then have *ms*: *ms* $\neq$ [] **by** *auto*

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

13.8 *split-face*

definition *pre-split-face* :: *face* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat list* $\Rightarrow$ *bool* **where**
pre-split-face oldF ram1 ram2 newVertexList $\equiv$
distinct (*vertices oldF*) $\wedge$ *distinct* (*newVertexList*)

$\wedge\ \mathcal{V}\ oldF \cap set\ newVertexList = \{\}$
$\wedge\ ram1 \in \mathcal{V}\ oldF \wedge ram2 \in \mathcal{V}\ oldF \wedge ram1 \neq ram2$

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

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

lemma *pre-split-face-symI*:
pre-split-face oldF ram1 ram2 newVertexList $\Longrightarrow$ *pre-split-face oldF ram2 ram1 newVertexList* **by** *auto*

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

lemma *split-face-distinct1*:
(*f12*, *f21*) = *split-face oldF ram1 ram2 newVertexList* $\Longrightarrow$ *pre-split-face oldF ram1 ram2 newVertexList* $\Longrightarrow$
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* $\Longrightarrow$ *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* $\Longrightarrow$ *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

13.9 *verticesFrom*

definition *verticesFrom* :: *face* ⇒ *vertex* ⇒ *vertex list* **where**
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*]: $v \in \mathcal{V}\ f \Longrightarrow a \in \mathcal{V}\ f \Longrightarrow a \in set\ (verticesFrom\ f\ v)$
by (*auto dest*: *verticesFrom-congs congs-pres-nodes*)

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

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

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

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

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

```
    then show ?case
    proof (cases m = u)
      case True with Cons have u ∉ set 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 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*) $\Longrightarrow$ $v \in$ *set* (*vertices f*) $\Longrightarrow$
length (*verticesFrom f v*) = *length* (*vertices f*)
by (*auto intro*: *congs-length verticesFrom-congs*)

lemma *verticesFrom-between*: $v' \in \mathcal{V}\ f \Longrightarrow$ *pre-between* (*vertices f*) *u v* $\Longrightarrow$
between (*vertices f*) *u v* = *between* (*verticesFrom f v'*) *u v*
by (*auto intro*!: *between-congs verticesFrom-congs*)

lemma *verticesFrom-is-nextElem*: $v \in \mathcal{V}\ f \Longrightarrow$
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*: $v' \in \mathcal{V}\ f \Longrightarrow$ *distinct* (*vertices f*)
$\Longrightarrow$ *is-nextElem* (*verticesFrom f v'*) (*last* (*verticesFrom f v'*)) *v* $\Longrightarrow$ $v = v'$
apply (*subgoal-tac distinct* (*verticesFrom f v'*))
apply (*subgoal-tac last* (*verticesFrom f v'*) $\in$ *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* ∈ *set* (*verticesFrom f v*) **by** (*auto simp*: *verticesFrom-pres-nodes*)
ultimately have *w* ∈ *set* (*v* # *tl* (*verticesFrom f v*)) **by** *auto*
with *neq* **have** *w* ∈ *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 *last-vFrom*:
⟦ *distinct*(*vertices f*); $x \in \mathcal{V}\ f$ ⟧ ⟹ *last*(*verticesFrom f x*) = $f^{-1} \cdot x$
apply(*frule split-list*)
apply(*clarsimp simp*:*prevVertex-def verticesFrom-Def split*:*list.split*)
done

lemma *rotate-before-vFrom*:
⟦ *distinct*(*vertices f*); $r \in \mathcal{V}\ f$; $r \neq u$ ⟧ ⟹
before (*verticesFrom f r*) *u v* ⟹ *before* (*verticesFrom f v*) *r u*
using [[*linarith-neq-limit* = *1*]]
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*); $x \in \mathcal{V}\ f$; $x \neq y$ ⟧ ⟹
$y \in set$(*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(*fastforce simp*:*between-def split-def*)
done

lemma *before-between2*:
⟦ *before* (*verticesFrom f u*) *v w*; *distinct*(*vertices f*); $u \in \mathcal{V}\ f$ ⟧
⟹ $u = v \vee u \in set$ (*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

13.10 *splitFace*

definition *pre-splitFace* :: *graph* ⇒ *vertex* ⇒ *vertex* ⇒ *face* ⇒ *vertex list* ⇒ *bool*
where
pre-splitFace g ram1 ram2 oldF nvs ≡
oldF ∈ $\mathcal{F}$ *g* ∧ ¬ *final oldF* ∧ *distinct* (*vertices oldF*) ∧ *distinct nvs*
∧ $\mathcal{V}$ *g* ∩ *set nvs* = {}
∧ $\mathcal{V}$ *oldF* ∩ *set nvs* = {}
∧ *ram1* ∈ $\mathcal{V}$ *oldF* ∧ *ram2* ∈ $\mathcal{V}$ *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* ∈ $\mathcal{F}$ *g*
apply (*unfold pre-splitFace-def*) **by** *force*

declare *pre-splitFace-def* [*simp del*]

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

lemma *split-face-empty-ram2-ram1-in-f12*:
pre-split-face oldF ram1 ram2 [] ⟹
(*f12*, *f21*) = *split-face oldF ram1 ram2* [] ⟹ (*ram2*, *ram1*) ∈ *edges f12*
proof −
assume *split*: (*f12*, *f21*) = *split-face oldF ram1 ram2* []
pre-split-face oldF ram1 ram2 []
then have *ram2* ∈ $\mathcal{V}$ *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 [] ⟹
(*ram2*, *ram1*) ∈ *edges* (*fst* (*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** (*ram2*, *ram1*) ∈ *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 [] ⟹
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF* [] ⟹
(*ram2*, *ram1*) ∈ *edges f12*
proof −
assume *pre*: *pre-splitFace g ram1 ram2 oldF* []
then have *oldF*: *oldF* ∈ $\mathcal{F}$ *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′*)
apply (*rule pre-splitFace-pre-split-face*)

apply (*rule pre*)
done
qed

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

lemma *splitFace-add-vertices-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* ⟹
oldF ≠ *f12* ⟹ *oldF* ≠ *f21* ⟹ *distinct* (*faces g*) ⟹
oldF ∉ $\mathcal{F}$ *newGraph*
by (*simp add*: *splitFace-def split-def distinct-set-replace*)

lemma *splitFace-faces-1*:
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVertexList* ⟹
oldF ∈ $\mathcal{F}$ *g* ⟹
set (*faces newGraph*) ∪ {*oldF*} = {*f12*, *f21*} ∪ *set* (*faces g*)
(**is** *?oldF* ⟹ *?C* ⟹ *?A* = *?B*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x* ∈ *?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'*: $f' \in \mathcal{F}\ g' \Longrightarrow$ *fst* (*snd* (*splitFace g' v a f' nvl*))
∈ $\mathcal{F}$ (*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*: $f \in \mathcal{F}$ (*snd* (*snd* (*splitFace g v a f' nvl*))) ⟹
$f \in \mathcal{F}\ g$
∨ 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* ∈ $\mathcal{V}$ *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* $\neq$ *hd* (*snd* (*splitAt ram1* (*vertices f*)) @ *fst* (*splitAt ram1* (*vertices f*)))
apply (*simp add*: *is-nextElem-def verticesFrom-Def*)
apply (*cases snd* (*splitAt ram1* (*vertices f*)) @ *fst* (*splitAt ram1* (*vertices f*)))
apply *simp* **apply** *simp*
apply (*simp add*: *is-sublist-def*) **by** *auto*

from *pre-f* **have** *before*: *before* (*verticesFrom f ram1*) *ram1 ram2* **apply** (*unfold pre-splitFace-def*)
apply *safe* **apply** (*rule before-verticesFrom*) **by** *auto*
have *fst* (*splitAt ram2* (*snd* (*splitAt ram1* (*verticesFrom f ram1*)))) = [] $\Longrightarrow$ *False*
proof −
assume *fst* (*splitAt ram2* (*snd* (*splitAt ram1* (*verticesFrom f ram1*)))) = []
moreover
from *ram2 pre-f* **have** *ram2* $\in$ *set* (*verticesFrom f ram1*)**apply** (*unfold pre-splitFace-def*)
by *auto*
with *pre-f* **have** *ram2* $\in$ *set* (*snd* (*splitAt ram1* (*vertices f*)) @ *fst* (*splitAt ram1* (*vertices f*)))
apply (*simp add*: *verticesFrom-Def*)
apply (*unfold pre-splitFace-def*) **by** *auto*
moreover
note *dist'*
ultimately have *ram2* = *hd* (*snd* (*splitAt ram1* (*vertices f*)) @ *fst* (*splitAt ram1* (*vertices f*)))
apply (*rule-tac ccontr*)
apply (*cases* (*snd* (*splitAt ram1* (*vertices f*)) @ *fst* (*splitAt ram1* (*vertices f*))))
apply *simp*
apply *simp*
by (*simp add*: *verticesFrom-Def del*: *distinct-append*)
with *ram2-not* **show** *?thesis* **by** *auto*
qed
with *before pre-bet'* **have** *between* (*verticesFrom f ram1*) *ram1 ram2* $\neq$ [] **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

definition *Edges* :: *vertex list* ⇒ (*vertex* × *vertex*) *set* **where**
Edges vs ≡ {(*a*,*b*). *is-sublist* [*a*,*b*] *vs*}

lemma *Edges-Nil*[*simp*]: *Edges* [] = {}
by(*simp add*:*Edges-def is-sublist-def*)

lemma *Edges-rev*:
Edges (*rev* (*zs*::*vertex list*)) = {(*b*,*a*). (*a*,*b*) ∈ *Edges zs*}
by (*auto simp add*: *Edges-def is-sublist-rev*)

lemma *in-Edges-rev*[*simp*]:
((*a*,*b*) : *Edges* (*rev* (*zs*::*vertex list*))) = ((*b*,*a*) ∈ *Edges zs*)
by (*simp add*: *Edges-rev*)

lemma *notinset-notinEdge1*: *x* ∉ *set xs* ⟹ (*x*,*y*) ∉ *Edges xs*
by(*unfold Edges-def*)(*blast intro*:*is-sublist-in*)

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

lemma *in-Edges-in-set*: (*x*,*y*) : *Edges vs* ⟹ *x* ∈ *set vs* ∧ *y* ∈ *set vs*
by(*unfold Edges-def*)(*blast intro*:*is-sublist-in is-sublist-in1*)

lemma *edges-conv-Edges*:
distinct(*vertices*(*f*::*face*)) ⟹ $\mathcal{E}$ *f* =
Edges (*vertices f*) ∪
(*if vertices f* = [] *then* {} *else* {(*last*(*vertices f*), *hd*(*vertices f*))})
by(*induct f*) (*auto simp*: *Edges-def is-nextElem-def*)

lemma *Edges-Cons*: *Edges*(*x*#*xs*) =
(*if xs* = [] *then* {} *else Edges xs* ∪ {(*x*,*hd xs*)})
apply(*auto simp*:*Edges-def*)
apply(*rule ccontr*)
apply(*simp*)
apply(*erule thin-rl*)
apply(*erule contrapos-np*)
apply(*clarsimp simp*:*is-sublist-def Cons-eq-append-conv*)
apply(*rename-tac as bs*)
apply(*erule disjE*)
apply *simp*
apply(*clarsimp*)
apply(*rename-tac cs*)
apply(*rule-tac x* = *cs* **in** *exI*)
apply(*rule-tac x* = *bs* **in** *exI*)

apply(*rule HOL.refl*)
apply(*clarsimp simp*:*neq-Nil-conv*)
apply(*subst is-sublist-rec*)
apply *simp*
apply(*simp add*:*is-sublist-def*)
apply(*erule exE*)+
apply(*rename-tac as bs*)
apply *simp*
apply(*rule-tac* $x = x\#as$ **in** *exI*)
apply(*rule-tac* $x = bs$ **in** *exI*)
apply *simp*
done

lemma *Edges-append*: $Edges(xs\ @\ ys) =$
$(if\ xs = []\ then\ Edges\ ys\ else$
$if\ ys = []\ then\ Edges\ xs\ else$
$Edges\ xs \cup Edges\ ys \cup \{(last\ xs,\ hd\ ys)\})$
apply(*induct xs*)
apply *simp*
apply (*simp add*:*Edges-Cons*)
apply *blast*
done

lemma *Edges-rev-disj*: $distinct\ xs \Longrightarrow Edges(rev\ xs) \cap Edges(xs) = \{\}$
apply(*induct xs*)
apply *simp*
apply(*auto simp*:*Edges-Cons Edges-append last-rev*
notinset-notinEdge1 notinset-notinEdge2)
done

lemma *disj-sets-disj-Edges*:
$set\ xs \cap set\ ys = \{\} \Longrightarrow Edges\ xs \cap Edges\ ys = \{\}$
by(*unfold Edges-def*)(*blast intro*:*is-sublist-in is-sublist-in1*)

lemma *disj-sets-disj-Edges2*:
$set\ ys \cap set\ xs = \{\} \Longrightarrow Edges\ xs \cap Edges\ ys = \{\}$
by(*blast intro*!:*disj-sets-disj-Edges*)

lemma *finite-Edges*[*iff*]: *finite*(*Edges xs*)
by(*induct xs*)(*simp-all add*:*Edges-Cons*)

lemma *Edges-compl*:
$[\![\ distinct\ vs;\ x \in set\ vs;\ y \in set\ vs;\ x \neq y\ ]\!] \Longrightarrow$
$Edges(x\ \#\ between\ vs\ x\ y\ @\ [y]) \cap Edges(y\ \#\ between\ vs\ y\ x\ @\ [x]) = \{\}$
using [[*linarith-neq-limit* = *1*]]
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(*fastforce*)
apply(*frule split-list*[*of y*])
apply(*clarsimp simp add*:*between-def split-def*)
apply (*clarsimp simp add*:*Edges-Cons Edges-append notinset-notinEdge1*
notinset-notinEdge2 disj-sets-disj-Edges disj-sets-disj-Edges2
Int-Un-distrib Int-Un-distrib2)
apply *fastforce*
done

lemma *Edges-disj*:
⟦ *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 *fastforce*
apply(*frule split-list*[*of y*])

apply *clarsimp*
apply (*clarsimp simp add*:*Edges-Cons Edges-append notinset-notinEdge1*
notinset-notinEdge2 disj-sets-disj-Edges disj-sets-disj-Edges2
Int-Un-distrib Int-Un-distrib2)
apply *fastforce*
apply(*frule split-list*[*of z*])
apply(*clarsimp simp add*:*between-def split-def*)
apply(*frule split-list*[*of y*])
apply *clarsimp*
apply (*clarsimp simp add*:*Edges-Cons Edges-append notinset-notinEdge1*
notinset-notinEdge2 disj-sets-disj-Edges disj-sets-disj-Edges2
Int-Un-distrib Int-Un-distrib2)
apply *fastforce*
done

lemma *edges-conv-Un-Edges*:
⟦ *distinct*(*vertices*(*f*::*face*)); $x \in \mathcal{V}\ f$; $y \in \mathcal{V}\ f$; $x \neq y$ ⟧ ⟹
$\mathcal{E}\ f$ = *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(*fastforce*)
apply(*frule split-list*[*of y*])
apply(*clarsimp simp add*:*between-def split-def*)
apply (*clarsimp simp add*:*Edges-Cons Edges-append*
notinset-notinEdge1 notinset-notinEdge2
disj-sets-disj-Edges disj-sets-disj-Edges2
Int-Un-distrib Int-Un-distrib2)
apply(*fastforce*)
done

lemma *Edges-between-edges*:
⟦ (a,b) ∈ *Edges* (u # *between* (*vertices*(*f*::*face*)) $u\ v$ @ [v]);
pre-split-face f u v vs ⟧ ⟹ (a,b) ∈ $\mathcal{E}\ f$
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

lemma *edges-split-face1*: *pre-split-face f u v vs* $\Longrightarrow$
$\mathcal{E}$(*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* $\Longrightarrow$
$\mathcal{E}$(*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 [] $\Longrightarrow$
(*f12*, *f21*) = *split-face oldF ram1 ram2* [] $\Longrightarrow$ (*ram1*, *ram2*) ∈ *edges f21*
proof −
assume *split*: (*f12*, *f21*) = *split-face oldF ram1 ram2* []
pre-split-face oldF ram1 ram2 []
then have *ram1* ∈ $\mathcal{V}$ *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 [] $\Longrightarrow$
(*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 ′)
  apply (rule pre-splitFace-pre-split-face)
  apply (rule pre)
  done
qed

lemma splitFace-f21-new-vertices:
  (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs ⟹
  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-edges-f12:
assumes vors: pre-split-face f ram1 ram2 vs
              (f12, f21) = split-face f ram1 ram2 vs
              vs ≠ [] vs1 = between (vertices f) ram1 ram2 vs1 ≠ []
shows edges f12 =
      {(hd vs, ram1) , (ram1, hd vs1), (last vs1, ram2), (ram2, last vs)} ∪
      Edges(rev vs) ∪ Edges vs1 (is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs
  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*
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*:
assumes *vors*: *pre-split-face f ram1 ram2* []
(*f12*, *f21*) = *split-face f ram1 ram2* []

vs1 = *between* (*vertices f*) *ram1 ram2 vs1* ≠ []
shows *edges f12* = {(*ram2*, *ram1*) , (*ram1*, *hd vs1*), (*last vs1*, *ram2*)} ∪
Edges vs1 (**is** *?lhs* = *?rhs*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x*: *x* ∈ *?lhs*
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*
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*:
assumes *vors*: *pre-split-face f ram1 ram2 vs*
(*f12*, *f21*) = *split-face f ram1 ram2 vs*
vs ≠ [] *between* (*vertices f*) *ram1 ram2* = []
shows *edges f12* = {(*hd vs*, *ram1*) , (*ram1*, *ram2*), (*ram2*, *last vs*)} ∪
Edges(*rev vs*) (**is** *?lhs* = *?rhs*)

proof (*intro equalityI subsetI*)
fix x
assume x: $x \in ?lhs$
def $c \equiv fst\ x$
def $d \equiv snd\ x$
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** $x \in ?rhs$
apply (*simp add*: *split-face-def is-nextElem-def is-sublist-def dist-f12*)
apply (*case-tac* $c = ram2 \wedge d = last\ vs$) **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* $c \in set(rev\ vs)$)
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* $\exists\, as\ bs.$ *rev list* @ $[d, c] = as$ @ $d \# c \# bs$) **apply**
simp **apply** (*intro exI*) **apply** *simp*
apply (*subgoal-tac* $\exists\, asa\ bs.$ *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* $c \in set\ (rev\ vs$ @ $ram1 \#\ [ram2])$)
apply (*thin-tac rev vs* @ $ram1 \#\ [ram2] = as$ @ $c \# d \# bs$) **apply** *simp*
by *simp*
next
fix x
assume x: $x \in ?rhs$
def $c \equiv fst\ x$
def $d \equiv snd\ x$
with *c-def* **have** [*simp*]: $x = (c,d)$ **by** *simp*

from *vors* **have** *dist-f12*: *distinct* (*vertices f12*) **apply** (*rule-tac split-face-distinct1*)
by *auto*
from *x vors* **show** *x* ∈ *?lhs*
apply (*simp add*: *dist-f12 is-nextElem-def is-sublist-def*) **apply** (*simp add*: *split-face-def*)
apply (*case-tac c* = *hd vs* ∧ *d* = *ram1*)
apply (*case-tac vs*) **apply** *simp*
apply *force*
apply *simp*
apply (*case-tac c* = *ram1* ∧ *d* = *ram2*) **apply** *force*
apply *simp*
apply (*case-tac c* = *ram2* ∧ *d* = *last vs*)
apply (*case-tac vs rule*:*rev-exhaust*) **apply** *simp*
apply *simp*
apply (*simp add*: *Edges-def is-sublist-def*)
apply (*elim exE*) **apply** *simp* **apply** (*rule conjI*)
apply (*rule impI*) **apply** (*rule disjI1*) **apply** (*intro exI*)
apply (*subgoal-tac c* # *d* # *rev as* @ [*ram1*, *ram2*] = [] @ *c* # *d* # *rev as* @ [*ram1*,*ram2*]) **apply** *assumption* **apply** *simp*
apply (*rule impI*) **apply** (*rule disjI1*) **apply** (*intro exI*) **by** *simp*
qed

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

def $d \equiv snd\ x$
with *c-def* **have** [*simp*]: $x = (c,d)$ **by** *simp*
from *vors* **have** *dist-f21*: *distinct* (*vertices f21*) **apply** (*rule-tac split-face-distinct2*)
by *auto*
from x *vors* **show** $x \in$ *?rhs*
apply (*simp add*: *split-face-def is-nextElem-def is-sublist-def dist-f21*)
apply (*case-tac* $c = ram1 \wedge d = ram2$) **apply** *simp* **apply** *simp* **apply** (*elim exE*)
apply (*case-tac as*) **apply** *simp* **apply** (*case-tac list*) **by** *auto*
next
fix x
assume x: $x \in$ *?rhs*
def $c \equiv fst\ x$
def $d \equiv snd\ x$
with *c-def* **have** [*simp*]: $x = (c,d)$ **by** *simp*
from *vors* **have** *dist-f21*: *distinct* (*vertices f21*) **apply** (*rule-tac split-face-distinct2*)
by *auto*
from x *vors* **show** $x \in$ *?lhs*
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* $\Longrightarrow$
$(f12, f21) = split\text{-}face\ f\ ram1\ ram2\ vs \Longrightarrow vs \neq [] \Longrightarrow$
$\{(last\ vs, ram2), (ram1, hd\ vs)\} \cup Edges\ vs \subseteq edges\ f21$
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* $\Longrightarrow$
verticesFrom f ram1 $=$ *ram1* $\#$ *between* (*vertices f*) *ram1 ram2* @ *ram2* $\#$ *between* (*vertices f*) *ram2 ram1*
apply (*subgoal-tac pre-between* (*vertices f*) *ram1 ram2*)
apply (*subgoal-tac distinct* (*vertices f*))
apply (*case-tac before* (*vertices f*) *ram1 ram2*)
apply (*simp add*: *verticesFrom-Def*)
apply (*subgoal-tac ram2* $\in$ *set* (*snd* (*splitAt ram1* (*vertices f*)))) **apply** (*drule splitAt-ram*)
apply (*subgoal-tac snd* (*splitAt ram2* (*snd* (*splitAt ram1* (*vertices f*)))) $=$ *snd* (*splitAt ram2* (*vertices f*)))
apply *simp* **apply** (*thin-tac snd* (*splitAt ram1* (*vertices f*)) $=$
fst (*splitAt ram2* (*snd* (*splitAt ram1* (*vertices f*)))) @
ram2 $\#$ *snd* (*splitAt ram2* (*snd* (*splitAt ram1* (*vertices f*))))) **apply** *simp*
apply (*rule before-dist-r2*) **apply** *simp* **apply** *simp*
apply (*subgoal-tac before* (*vertices f*) *ram2 ram1*)
apply (*subgoal-tac pre-between* (*vertices f*) *ram2 ram1*)
apply (*simp add*: *verticesFrom-Def*)

apply (*subgoal-tac ram2 ∈ set* (*fst* (*splitAt ram1* (*vertices f*)))) **apply** (*drule splitAt-ram*)
apply (*subgoal-tac fst* (*splitAt ram2* (*fst* (*splitAt ram1* (*vertices f*)))) = *fst* (*splitAt ram2* (*vertices f*)))
apply *simp* **apply** (*thin-tac fst* (*splitAt ram1* (*vertices f*)) =
fst (*splitAt ram2* (*fst* (*splitAt ram1* (*vertices f*)))) @
ram2 # *snd* (*splitAt ram2* (*fst* (*splitAt ram1* (*vertices f*))))) **apply** *simp*
apply (*rule before-dist-r1*) **apply** *simp* **apply** *simp* **apply** (*simp add*: *pre-between-def*) **apply** *force*
apply (*simp add*: *pre-split-face-def*) **by** (*rule pre-split-face-p-between*)

lemma *split-face-edges-f-vs1-vs2*:
assumes *vors*: *pre-split-face f ram1 ram2 vs*
between (*vertices f*) *ram1 ram2* = []
between (*vertices f*) *ram2 ram1* = []
shows *edges f* = {(*ram2*, *ram1*) , (*ram1*, *ram2*)} (**is** *?lhs* = *?rhs*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x*: *x* ∈ *?lhs*
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*
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* ∈ $\mathcal{V}$ *f*) **apply** (*simp add*: *verticesFrom-is-nextElem verticesFrom-ram1*)
apply (*simp add*: *is-nextElem-def*) **apply** *blast*
by (*simp add*: *pre-split-face-def*)
qed

lemma *split-face-edges-f-vs1*:
assumes *vors*: *pre-split-face f ram1 ram2 vs*
between (*vertices f*) *ram1 ram2* = []
vs2 = *between* (*vertices f*) *ram2 ram1 vs2* ≠ []
shows *edges f* = {(*last vs2*, *ram1*) , (*ram1*, *ram2*), (*ram2*, *hd vs2*)} ∪

```
            Edges vs2 (is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x: x ∈ ?lhs
  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
  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** $x \in$ *?lhs* **apply** (*simp add*: *dist-f*)
apply (*subgoal-tac ram1* $\in$ *set* (*vertices f*)) **apply** (*simp add*: *verticesFrom-is-nextElem verticesFrom-ram1*)
apply (*simp add*: *is-nextElem-def*)
apply (*case-tac c* = *last* (*between* (*vertices f*) *ram2 ram1*) $\wedge$ *d* = *ram1*) **apply** *simp* **apply** *simp* **apply** (*rule disjI1*)
apply (*case-tac c* = *ram1* $\wedge$ *d* = *ram2*) **apply** (*simp add*: *is-sublist-def*) **apply** *force* **apply** *simp*
apply (*case-tac c* = *ram2* $\wedge$ *d* = *hd* (*between* (*vertices f*) *ram2 ram1*))
apply (*case-tac between* (*vertices f*) *ram2 ram1*) **apply** *simp* **apply** (*simp add*: *is-sublist-def*) **apply** (*intro exI*)
apply (*subgoal-tac ram1* # *ram2* # *a* # *list* =
[*ram1*] @ *ram2* # *a* # (*list*)) **apply** *assumption* **apply** *simp*
apply *simp*
apply (*subgoal-tac is-sublist* [*c*, *d*] ((*ram1* #
[*ram2*]) @ *between* (*vertices f*) *ram2 ram1* @ []))
apply *simp* **apply** (*rule is-sublist-add*) **apply** (*simp add*: *Edges-def*)
by (*simp add*: *pre-split-face-def*)
qed

lemma *split-face-edges-f-vs2*:
assumes *vors*: *pre-split-face f ram1 ram2 vs*
vs1 = *between* (*vertices f*) *ram1 ram2 vs1* $\neq$ []
between (*vertices f*) *ram2 ram1* = []
shows *edges f* = {(*ram2*, *ram1*) , (*ram1*, *hd vs1*), (*last vs1*, *ram2*)} $\cup$
Edges vs1 (**is** *?lhs* = *?rhs*)
proof (*intro equalityI subsetI*)
fix *x*
assume *x*: $x \in$ *?lhs*
def $c \equiv$ *fst x*
def $d \equiv$ *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** $x \in$ *?rhs* **apply** (*simp add*: *dist-f*)
apply (*subgoal-tac pre-between* (*vertices f*) *ram1 ram2*)
apply (*drule is-nextElem-or*) **apply** *assumption*
apply (*simp add*: *Edges-def*)
apply (*case-tac is-sublist* [*c*, *d*] (*ram1* # *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*
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* ∈ $\mathcal{V}$ *f*) **apply** (*simp add*: *verticesFrom-is-nextElem verticesFrom-ram1*)
apply (*simp add*: *is-nextElem-def*)
apply (*case-tac c* = *ram2* ∧ *d* = *ram1*) **apply** *simp* **apply** *simp* **apply** (*rule disjI1*)
apply (*case-tac c* = *ram1* ∧ *d* = *hd* (*between* (*vertices f*) *ram1 ram2*))
apply (*case-tac between* (*vertices f*) *ram1 ram2*) **apply** *simp* **apply** (*force simp*: *is-sublist-def*) **apply** *simp*
apply (*case-tac c* = *last* (*between* (*vertices f*) *ram1 ram2*) ∧ *d* = *ram2*)
apply (*case-tac between* (*vertices f*) *ram1 ram2 rule*: *rev-exhaust*) **apply** *simp* **apply** (*simp add*: *is-sublist-def*)
apply (*intro exI*)
apply (*subgoal-tac ram1* # *ys* @ [*y*, *ram2*] =
(*ram1* # *ys*) @ *y* # *ram2* # []) **apply** *assumption* **apply** *simp*
apply *simp*
apply (*simp add*: *Edges-def*)
apply (*subgoal-tac is-sublist* [*c*, *d*] ([*ram1*] @ *between* (*vertices f*) *ram1 ram2*

@ [*ram2*]))
  apply *simp* **apply** (*rule is-sublist-add*) **apply** *simp*
 by (*simp add*: *pre-split-face-def*)
qed

lemma *split-face-edges-f*:
assumes *vors*: *pre-split-face f ram1 ram2 vs*
  vs1 = *between* (*vertices f*) *ram1 ram2 vs1* ≠ []
  vs2 = *between* (*vertices f*) *ram2 ram1 vs2* ≠ []
shows *edges f* = {(*last vs2*, *ram1*) , (*ram1*, *hd vs1*), (*last vs1*, *ram2*), (*ram2*, *hd vs2*)} ∪
    Edges vs1 ∪ *Edges vs2* (**is** *?lhs* = *?rhs*)
proof (*intro equalityI subsetI*)
 fix *x*
 assume *x*: *x* ∈ *?lhs*
 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*
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 ∈ $\mathcal{F}$ *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′* = ($\bigcup_{a \in set\ (replace\ f\ [f21]\ (faces\ g))}$ *edges a*) ∪
edges (*f12*) **by**(*auto simp*: *splitFace-def split-def edges-graph-def*)
with *pre vs-neq* **show** *?thesis* **apply** (*simp add*: *UNION-eq*) **apply** (*rule equalityI*) **apply** *simp*
apply (*rule conjI*) **apply** (*rule subsetI*) **apply** *simp* **apply** (*erule bexE*) **apply** (*drule replace5*)
apply (*case-tac xa* ∈ $\mathcal{F}$ *g*) **apply** *simp*
apply (*subgoal-tac x* ∈ *edges g*) **apply** *simp*
apply (*simp add*: *edges-graph-def*) **apply** *force*
apply *simp*
apply (*subgoal-tac pre-split-face f ram1 ram2 vs*)
apply (*case-tac between* (*vertices f*) *ram2 ram1* = [])
apply (*frule split-face-edges-f21-bet*) **apply** (*rule split*) **apply** *simp* **apply** *simp*
apply (*case-tac between* (*vertices f*) *ram1 ram2* = [])
apply (*frule split-face-edges-f-vs1-vs2*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *force*
apply (*frule split-face-edges-f-vs2*) **apply** *simp* **apply** *simp* **apply** *simp*

apply *force*
apply (*frule split-face-edges-f21*) **apply** (*rule split*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac between* (*vertices f*) *ram1 ram2* = []) **apply** (*frule split-face-edges-f-vs1*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *force*
apply (*frule split-face-edges-f*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *force*
apply *simp*
apply (*subgoal-tac pre-split-face f ram1 ram2 vs*)
apply (*case-tac between* (*vertices f*) *ram1 ram2* = []) **apply** (*frule split-face-edges-f12-bet*) **apply** (*rule split*) **apply** *simp* **apply** *simp*
apply (*case-tac between* (*vertices f*) *ram2 ram1* = []) **apply** (*frule split-face-edges-f-vs1-vs2*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *force*
apply (*frule split-face-edges-f-vs1*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *force*
apply (*frule split-face-edges-f12*) **apply** (*rule split*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac between* (*vertices f*) *ram2 ram1* = []) **apply** (*frule split-face-edges-f-vs2*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *force*
apply (*frule split-face-edges-f*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *force*
apply *simp*
apply *simp*
apply (*subgoal-tac pre-split-face f ram1 ram2 vs*)
apply (*subgoal-tac* (*ram2*, *last vs*) ∈ *edges f12* ∧ (*hd vs*, *ram1*) ∈ *edges f12*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** *simp*
apply (*subgoal-tac* (*ram1*, *hd vs*) ∈ *edges f21* ∧ (*last vs*, *ram2*) ∈ *edges f21*)
apply (*rule conjI*) **apply** (*rule disjI1*) **apply** (*rule bexI*) **apply** (*elim conjE*) **apply** *simp*
apply (*rule replace3*) **apply**(*erule pre-splitFace-oldF*) **apply** *simp*
apply (*rule conjI*) **apply** (*rule disjI1*) **apply** (*rule bexI*) **apply** (*elim conjE*) **apply** *simp*
apply (*rule replace3*) **apply**(*erule pre-splitFace-oldF*)
apply *simp*
apply (*subgoal-tac edges f* ⊆ {*y*. ∃ *x*∈*set* (*replace f* [*f21*] (*faces g*)). *y* ∈ *edges x*} ∪ *edges f12*)
apply (*subgoal-tac edges g* ⊆ {*y*. ∃ *x*∈*set* (*replace f* [*f21*] (*faces g*)). *y* ∈ *edges x*} ∪ *edges f12*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** *simp*
apply (*subgoal-tac Edges*(*rev vs*) ⊆ *edges f12*) **apply** (*rule conjI*) **prefer** *2* **apply** *blast*
apply (*subgoal-tac Edges vs* ⊆ *edges f21*)
apply (*subgoal-tac Edges vs* ⊆ {*y*. ∃ *x*∈*set* (*replace f* [*f21*] (*faces g*)). *y* ∈

edges x}) **apply** *blast*
apply (*rule subset-trans*) **apply** *assumption* **apply** (*rule subsetI*) **apply** *simp* **apply** (*rule bexI*) **apply** *simp*
apply (*rule replace3*) **apply**(*erule pre-splitFace-oldF*) **apply** *simp*

apply (*frule split-face-edges-f21-subset*) **apply** (*rule split*) **apply** *simp* **apply** *simp*
apply (*frule split-face-edges-f12-subset*) **apply** (*rule split*) **apply** *simp* **apply** *simp*
apply (*simp add*: *edges-graph-def*) **apply** (*rule subsetI*) **apply** *simp* **apply** (*elim bexE*)
apply (*case-tac xa* = *f*) **apply** *simp* **apply** *blast*
apply (*rule disjI1*) **apply** (*rule bexI*) **apply** *simp* **apply** (*rule replace4*) **apply** *simp* **apply** *force*
apply (*rule subsetI*)
apply (*subgoal-tac* ∃ *u v*. *x* = (*u*,*v*)) **apply** (*elim exE conjE*)
apply (*frule split-face-edges-or* [*OF split*]) **apply** *simp*
apply (*case-tac* (*u*, *v*) ∈ *edges f12*) **apply** *simp* **apply** *simp*
apply (*rule bexI*) **apply** (*thin-tac* (*u*, *v*) ∈ *edges f*) **apply** *assumption*
apply (*rule replace3*) **apply**(*erule pre-splitFace-oldF*) **apply** *simp* **apply** *simp*
apply (*frule split-face-edges-f21-subset*) **apply** (*rule split*) **apply** *simp* **apply** *simp*
apply (*frule split-face-edges-f12-subset*) **apply** (*rule split*) **apply** *simp* **apply** *simp*
by *simp*
qed

lemma *pre-splitFace-edges-f-in-g*: *pre-splitFace g ram1 ram2 f vs* ⟹ *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-eq*) **apply** (*rule equalityI*)
apply *simp*
apply (*rule conjI*) **apply** (*rule subsetI*) **apply** *simp* **apply** (*erule bexE*) **apply** (*drule replace5*)
apply (*case-tac xa* ∈ $\mathcal{F}$ *g*) **apply** *simp*
apply (*subgoal-tac x* ∈ *edges g*) **apply** *simp*
apply (*simp add*: *edges-graph-def*) **apply** *force*
apply *simp*
apply (*subgoal-tac pre-split-face f ram1 ram2* [])
apply (*case-tac between* (*vertices f*) *ram2 ram1* = []) **apply** (*simp add*: *pre-FaceDiv-between2*)
apply (*frule split-face-edges-f21-vs*) **apply** (*rule split*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac x* = (*ram1*, *ram2*)) **apply** *simp* **apply** *simp* **apply** (*rule disjI2*)
apply (*rule pre-splitFace-edges-f-in-g2*) **apply** *simp*
apply (*subgoal-tac pre-split-face f ram1 ram2* [])
apply (*frule split-face-edges-f*) **apply** *simp* **apply** *simp* **apply** (*rule pre-FaceDiv-between1*) **apply** *simp* **apply** *simp*
apply *simp* **apply** *force* **apply** *simp* **apply** *simp*

apply (*rule subsetI*) **apply** *simp*
apply (*subgoal-tac pre-split-face f ram1 ram2* [])
apply (*case-tac between* (*vertices f*) *ram1 ram2* = []) **apply** (*simp add*: *pre-FaceDiv-between1*)
apply (*frule split-face-edges-f12-vs*) **apply** (*rule split*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac x* = (*ram2*, *ram1*)) **apply** *simp* **apply** *simp* **apply** (*rule disjI2*)
apply (*rule pre-splitFace-edges-f-in-g2*) **apply** *simp*
apply (*subgoal-tac pre-split-face f ram1 ram2* [])
apply (*frule split-face-edges-f*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** (*rule pre-FaceDiv-between2*) **apply** *simp*
apply *simp* **apply** *force* **apply** *simp* **apply** *simp*
apply *simp*
apply (*subgoal-tac pre-split-face f ram1 ram2* [])
apply (*subgoal-tac* (*ram1*, *ram2*) ∈ *edges f21*)
apply (*rule conjI*) **apply** (*rule disjI1*) **apply** (*rule bexI*) **apply** *simp* **apply** (*force*)
apply (*subgoal-tac* (*ram2*, *ram1*) ∈ *edges f12*)

apply (*rule conjI*) **apply** *force*
apply (*rule subsetI*) **apply** (*simp add*: *edges-graph-def*) **apply** (*elim bexE*)
apply (*case-tac xa* = *f*) **apply** *simp*
apply (*subgoal-tac* ∃ *u v*. *x* = (*u*,*v*)) **apply** (*elim exE conjE*)
apply (*subgoal-tac pre-split-face f ram1 ram2* []) **apply** (*frule split-face-edges-or* [*OF split*]) **apply** *simp*
apply (*case-tac* (*u*, *v*) ∈ *edges f12*) **apply** *simp* **apply** *simp* **apply** *force* **apply** *simp* **apply** *simp*
apply (*rule disjI1*) **apply** (*rule bexI*) **apply** *simp* **apply** (*rule replace4*) **apply** *simp* **apply** *force*
apply (*frule split-face-edges-f12-vs*) **apply** *simp* **apply** (*rule split*) **apply** *simp*
apply (*rule pre-FaceDiv-between1*) **apply** *simp* **apply** *simp*
apply (*frule split-face-edges-f21-vs*) **apply** *simp* **apply** (*rule split*) **apply** *simp*
apply (*rule pre-FaceDiv-between2*) **apply** *simp* **apply** *simp*
by *simp*
qed

lemma *splitFace-edges-incr*:
pre-splitFace g ram1 ram2 f vs ⟹
(*f*$_1$, *f*$_2$, *g*′) = *splitFace g ram1 ram2 f vs* ⟹
edges g ⊆ *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* ⟹
edges g ⊆ *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

13.11 *removeNones*

definition *removeNones* :: *'a option list* ⇒ *'a list* **where**
removeNones vOptionList ≡ [*the x*. *x* ← *vOptionList*, *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-empty*[*simp*]: *removeNones* [] = [] **by** *auto*
declare *removeNones-def* [*simp del*]

13.12 *natToVertexList*

primrec *natToVertexListRec* ::
nat ⇒ *vertex* ⇒ *face* ⇒ *nat list* ⇒ *vertex option list*
where
natToVertexListRec old v f [] = [] |
natToVertexListRec old v f (*i*#*is*) =
(*if i* = *old then None*#*natToVertexListRec i v f is*
else Some (f^i • *v*)
natToVertexListRec i v f is)

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

13.13 *indexToVertexList*

lemma *nextVertex-inj*:
distinct (*vertices f*) ⟹ *v* ∈ $\mathcal{V}$ *f* ⟹
i < *length* (*vertices* (*f*::*face*)) ⟹ *a* < *length* (*vertices f*) ⟹
f^a•*v* = f^i•*v* ⟹ *i* = *a*
proof −
assume *d*: *distinct* (*vertices f*) **and** *v*: *v* ∈ $\mathcal{V}$ *f* **and** *i*: *i* < *length* (*vertices* (*f*::*face*))
and *a*: *a* < *length* (*vertices f*) **and** *eq*: f^a•*v* = f^i•*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 ⋀ *a i*. *a* < *length xs* ⟹ *i* < *length xs* ⟹ *take a xs* = *take i xs* ⟹ *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*) ⟹ *v* ∈ *V f* ⟹ (∀ *i* ∈ *set is*. *i* < *length* (*vertices f*)) ⟹
(⋀*a*. *a* < *length* (*vertices f*) ⟹ *hideDupsRec* ((*f* ∙ ^^ *a*) *v*) [(*f* ∙ ^^ *k*) *v*. *k* ← *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*) ⟹ *v* ∈ *V f* ⟹
(∀ *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*: $\bigwedge$ *old*. (*length* (*natToVertexListRec old v f ls*)) = *length ls*
apply (*induct ls*) **by** *auto*

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

lemma *natToVertexListRec-length*[*simp*]: $\bigwedge$ *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*)) $\Longrightarrow$
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*)) $\Longrightarrow$ *Suc n* < *length es* $\Longrightarrow$
(*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*: $\bigwedge$ *old n*. *Suc n* < *length es* $\Longrightarrow$
    (*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 $\bigwedge$ *old*. *natToVertexListRec old v f es* ! *Suc m* = (*if es* ! *m* = *es* ! *Suc m then None else Some* ($f^{es\ !\ Suc\ m}$ · *v*))
          by (*rule-tac cons1*) *auto*
        then show *?thesis* **apply** (*cases e* = *old*) **by** (*simp-all add*: *Suc*)
      qed
    qed
  qed

with *n* **have** *natToVertexListRec 0 v f es ! Suc n = (if es ! n = es ! Suc n then None else Some (f^es ! Suc n • v))* **by** (*rule-tac rec*) *auto*
with *incr* **show** *?thesis* **by** (*cases es*) *auto*
qed

lemma *natToVertexList-nth-0*: *incrIndexList es* (*length es*) (*length* (*vertices f*)) ⟹ *0 < length es* ⟹
(*natToVertexList v f es*)!*0* = *Some* (*f*^(*es*!*0*) • *v*)
apply (*cases es*)
apply (*simp-all add*: *nextVertices-def*)
by (*subgoal-tac a = 0*) *auto*

lemma *natToVertexList-hd*[*simp*]:
incrIndexList es (*length es*) (*length* (*vertices f*)) ⟹ *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* ∈ 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* ∈ 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'*) • *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*)) • *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*)) • *v* **by** (*auto intro*!: *verticesFrom-nth*)

from *vors lvf* **have** *verticesFrom f v ! ((length* (*vertices f*)) − (*Suc 0*)) = *last*

(*verticesFrom f v*)
  apply (*rule-tac nth-last*)
  by (*auto simp*: *verticesFrom-length*)
 with *rule1 rule2* **show** *?thesis* **by** *auto*
qed

lemma *indexToVertexList-last*[*simp*]:
 distinct (*vertices f*) ⟹ *v* ∈ $\mathcal{V}$ *f* ⟹ *incrIndexList es* (*length es*) (*length* (*vertices f*)) ⟹ *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*: ⋀ *n iset*. ∀ *i* ∈ *iset*. *i* < *n* ⟹ *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*: ⋀ *iset*. *sublist xs iset* = *sublist xs* {*i*. *i* < *length xs* ∧ *i* ∈ *iset*}
proof (*induct xs*)
 case *Nil* **then show** *?case* **by** *simp*
next
 case (*Cons x xs*) **then**
 have *sublist xs* {*j*. *Suc j* ∈ *iset*} = *sublist xs* {*i*. *i* < *length xs* ∧ *i* ∈ {*j*. *Suc j* ∈ *iset*}} **by** (*rule-tac Cons*)
 then show *?case* **by** (*simp add*: *sublist-Cons*)
qed

lemma *natToVertexList-sublist1*: *distinct* (*vertices f*) ⟹
 v ∈ $\mathcal{V}$ *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* ∈ $\mathcal{V}$ *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*) **by** *auto*
then have *small-n′*: *n′* < *length* (*butlast es*) **by** *auto*

from *Suc small-n* **have** *take-n′*: *take* (*Suc n′*) (*butlast es* @ [*last es*]) = *take* (*Suc n′*) (*butlast es*) **by** *auto*

from *small-n* **have** *es-exh*: *es* = *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* ≠ *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* ($f^{(es!(Suc\ n'))}$ · *v*)
apply (*simp add*: *natToVertexList-nth-Suc*)
by (*simp add*: *Suc*)
with *Suc* **have** *natToVertexList v f es* ! *n* = *Some* ($f^{(es!n)}$ · *v*) **by** *auto*
with *Some* **have** *v*′: *v*′ = $f^{(es!n)}$ · *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'* ≤ *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* **by** *auto*

have ⋀*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* **have** *x* ≤ *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* ∧ *i* ∈ *set* (*take n es*)} = (*set* (*take n es*)) **by** *auto*
then have *elim-insert*: {*i*. *i* < *es* ! *n* ∧ *i* ∈ *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*: $\bigwedge x.\ x \in (set\ (take\ n\ es)) \Longrightarrow x \leq\ (es\ !\ n')$
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* **show** $x \leq es!n'$ **by** (*auto simp*: *take-Suc-conv-app-nth*)
qed

def $m' \equiv m - 1$
def *Suc-es-n'* $\equiv Suc\ (es!n')$

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*: $\{j.\ j + length\ (take\ Suc\text{-}es\text{-}n'\ vs) \in set\ (take\ n\ es)\} = \{\}$
apply *auto* **apply** (*frule helper*) **by** *arith*

from *Suc-es-n'-def mgz esn m'-def* **have** *esn'*: $es!n = Suc\text{-}es\text{-}n' + m'$ **by** *auto*

with *smaller* **have** $(take\ (Suc\text{-}es\text{-}n' + m')\ vs) = take\ (Suc\text{-}es\text{-}n')\ vs\ @\ take\ m'\ (drop\ (Suc\text{-}es\text{-}n')\ vs)$
by (*auto intro*: *take-add*)
with *esn'* **have** *sublist* $(take\ (es\ !\ n)\ vs)\ (set\ (take\ n\ es))$
$= sublist\ (take\ (Suc\text{-}es\text{-}n')\ vs\ @\ take\ m'\ (drop\ (Suc\text{-}es\text{-}n')\ vs))\ (set\ (take\ n\ es))$ **by** *auto*
then have *sublist* $(take\ (es\ !\ n)\ vs)\ (set\ (take\ n\ es)) =$
$sublist\ (take\ (Suc\text{-}es\text{-}n')\ vs)\ (set\ (take\ n\ es))\ @$
$sublist\ (take\ m'\ (drop\ (Suc\text{-}es\text{-}n')\ vs))\ \{j.\ j + length\ (take\ (Suc\text{-}es\text{-}n')\ vs) : (set\ (take\ n\ es))\}$
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)))}
       by (auto simp: sublist-append)
      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** $(rule\ natToVertexList\text{-}sublist1)$
with *take-vs take-nTVL take-es vs-def* **show** *?thesis* **by** *simp*
qed

lemma *filter-Cons2*:
$x \notin set\ ys \Longrightarrow [y{\leftarrow}ys.\ \ y = x \vee P\ y] = [y{\leftarrow}ys.\ P\ y]$
by $(induct\ ys)\ (auto)$

lemma *natToVertexList-removeNones*:
distinct $(vertices\ f) \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow$
incrIndexList es $(length\ es)\ (length\ (vertices\ f)) \Longrightarrow$
$[x{\leftarrow}verticesFrom\ f\ v.\ x \in set\ (removeNones\ (natToVertexList\ v\ f\ es))]$
$= removeNones\ (natToVertexList\ v\ f\ es)$
proof −
assume *vors*: *distinct* $(vertices\ f)\ v \in \mathcal{V}\ f$
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** $(rule\ natToVertexList\text{-}sublist)$
from *dist* **have** $[x \leftarrow verticesFrom\ f\ v.$
$x \in set\ (sublist\ (verticesFrom\ f\ v)\ (set\ es))] = removeNones\ (natToVertexList$
$v\ f\ es)$
apply $(simp\ add{:}\ filter\text{-}in\text{-}sublist)$
by $(simp\ add{:}\ sub\text{-}eq)$
with *sub-eq* **show** *?thesis* **by** *simp*
qed

definition *is-duplicateEdge* :: *graph* ⇒ *face* ⇒ *vertex* ⇒ *vertex* ⇒ *bool* **where**
is-duplicateEdge g f a b ≡
$((a,\ b) \in edges\ g \wedge (a,\ b) \notin edges\ f \wedge (b,\ a) \notin edges\ f)$
$\vee\ ((b,\ a) \in edges\ g \wedge (b,\ a) \notin edges\ f \wedge (a,\ b) \notin edges\ f)$

definition *invalidVertexList* :: *graph* ⇒ *face* ⇒ *vertex option list* ⇒ *bool* **where**
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*

13.14 *pre-subdivFace(′)*

definition *pre-subdivFace-face* :: *face* ⇒ *vertex* ⇒ *vertex option list* ⇒ *bool* **where**
pre-subdivFace-face f v′ vOptionList ≡
$[v \leftarrow verticesFrom\ f\ v'.\ v \in set\ (removeNones\ vOptionList)]$
$= (removeNones\ vOptionList)$
$\wedge \neg\ final\ f \wedge distinct\ (vertices\ f)$

$\wedge$ *hd* (*vOptionList*) = *Some* v'
$\wedge$ $v' \in \mathcal{V}$ *f*
$\wedge$ *last* (*vOptionList*) = *Some* (*last* (*verticesFrom f v'*))
$\wedge$ *hd* (*tl* (*vOptionList*)) $\neq$ *last* (*vOptionList*)
$\wedge$ *2* $<$ | *vOptionList* |
$\wedge$ *vOptionList* $\neq$ []
$\wedge$ *tl* (*vOptionList*) $\neq$ []

definition *pre-subdivFace* :: *graph* $\Rightarrow$ *face* $\Rightarrow$ *vertex* $\Rightarrow$ *vertex option list* $\Rightarrow$ *bool* **where**
pre-subdivFace g f v' vOptionList $\equiv$
pre-subdivFace-face f v' vOptionList $\wedge$ $\neg$ *invalidVertexList g f vOptionList*

definition *pre-subdivFace'* :: *graph* $\Rightarrow$ *face* $\Rightarrow$ *vertex* $\Rightarrow$ *vertex* $\Rightarrow$ *nat* $\Rightarrow$ *vertex option list* $\Rightarrow$ *bool* **where**
pre-subdivFace' g f v' ram1 n vOptionList $\equiv$
$\neg$ *final f* $\wedge$ $v' \in \mathcal{V}$ *f* $\wedge$ *ram1* $\in \mathcal{V}$ *f*
$\wedge$ $v' \notin$ *set* (*removeNones vOptionList*)
$\wedge$ *distinct* (*vertices f*)
$\wedge$ (
[$v \leftarrow$ *verticesFrom f v'*. $v \in$ *set* (*removeNones vOptionList*)]
= (*removeNones vOptionList*)
$\wedge$ *before* (*verticesFrom f v'*) *ram1* (*hd* (*removeNones vOptionList*))
$\wedge$ *last* (*vOptionList*) = *Some* (*last* (*verticesFrom f v'*))
$\wedge$ *vOptionList* $\neq$ []
$\wedge$ ((v' = *ram1* $\wedge$ (*0* $<$ *n*)) $\vee$ ((v' = *ram1* $\wedge$ (*hd* (*vOptionList*) $\neq$ *Some* (*last* (*verticesFrom f v'*)))) $\vee$ ($v' \neq$ *ram1*)))
$\wedge$ $\neg$ *invalidVertexList g f vOptionList*
$\wedge$ (*n* = *0* $\wedge$ *hd* (*vOptionList*) $\neq$ *None* $\longrightarrow$ $\neg$ *is-duplicateEdge g f ram1* (*the* (*hd* (*vOptionList*))))
$\vee$ (*vOptionList* = [] $\wedge$ $v' \neq$ *ram1*)
)

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

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

lemma *pre-subdivFace-face-in-f'*[*intro*]: *pre-subdivFace-face f v ls* $\Longrightarrow$ *Some a* $\in$ *set ls* $\Longrightarrow$ *a* $\in \mathcal{V}$ *f*

apply (*cases a = v*) **apply** (*force simp*: *pre-subdivFace-face-def*)
apply (*rule verticesFrom-in'*) **apply** (*rule pre-subdivFace-face-in-f*)
by *auto*

lemma *filter-congs-shorten1*: *distinct* (*verticesFrom f v*) ⟹ [*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* (*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 *invalidVertexList-shorten*: *invalidVertexList g f vol* $\Longrightarrow$ *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 *pre-subdivFace-pre-subdivFace'*: *v* $\in$ $\mathcal{V}$ *f* $\Longrightarrow$ *pre-subdivFace g f v* (*vo* # *vol*) $\Longrightarrow$
pre-subdivFace' g f v v 0 (*vol*)
proof −
assume *vors*: *v* $\in$ $\mathcal{V}$ *f pre-subdivFace g f v* (*vo* # *vol*)
then have *vors'*: *v* $\in$ $\mathcal{V}$ *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*)) $\in$ *set vol* **apply** (*induct vol*) **apply** *auto* **apply** (*case-tac a*) **by** *auto*
then have *Some* (*hd* (*removeNones vol*)) $\in$ *set* (*vo* # *vol*) **by** *auto*
with *vors'* **have** *hd*: *hd* (*removeNones vol*) $\in$ $\mathcal{V}$ *f* **by** (*rule-tac pre-subdivFace-face-in-f'*)

from *vors'* **have** *Some v* = *vo* **by** (*auto simp*: *pre-subdivFace-face-def*)
with *vors'* **have** *v* $\notin$ *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* $\Longrightarrow$ *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* $\Longrightarrow$ *distinct* (*vertices f*) $\Longrightarrow$ *v* ∈ $\mathcal{V}$ *f* $\Longrightarrow$ *2* < |*es*| $\Longrightarrow$
incrIndexList es (*length es*) (*length* (*vertices f*)) $\Longrightarrow$
pre-subdivFace-face f v (*natToVertexList v f es*)
proof −
assume *vors*: ¬ *final f distinct* (*vertices f*) *v* ∈ $\mathcal{V}$ *f 2* < |*es*|
incrIndexList es (*length es*) (*length* (*vertices f*))
then have *lastOvl*: *last* (*natToVertexList v f es*) = *Some* (*last* (*verticesFrom f v*)) **by** *auto*

from *vors* **have** *nvl-l*: *2* < | *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*:
$\neg$ *final f* $\Longrightarrow$ *distinct* (*vertices f*) $\Longrightarrow$ *v* $\in$ $\mathcal{V}$ *f* $\Longrightarrow$ *2* < |*es*| $\Longrightarrow$
incrIndexList es (*length es*) (*length* (*vertices f*)) $\Longrightarrow$
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* $\Longrightarrow$ *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*) $\Longrightarrow$
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*) $\Longrightarrow$ *vertices f* = *a* @ *v* # *b* $\Longrightarrow$ *verticesFrom f v* = *v* # *b* @ *a*
by (*simp add*: *verticesFrom-Def*)

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

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

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

lemma *filter-distinct-at*:
distinct xs $\Longrightarrow$ *xs* = (*as* @ *u* # *bs*) $\Longrightarrow$ [*v*←*xs*. *v* = *u* ∨ *P v*] = *u* # *us* $\Longrightarrow$
[*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-at3*: *distinct xs* $\Longrightarrow$ *xs* = (*as* @ *u* # *bs*) $\Longrightarrow$
[*v*←*xs*. *v* = *u* ∨ *P v*] = *u* # *us* $\Longrightarrow$ ∀ *z* ∈ *set zs*. *z* ∈ *set as* ∨ ¬ (*P z*) $\Longrightarrow$
[*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* $\Longrightarrow$ *xs* = (*as* @ *u* # *bs*)
$\Longrightarrow$ [*v*←*xs*. *v* = *u* ∨ *v* ∈ *set us*] = *u* # *us*
$\Longrightarrow$ *set zs* ∩ *set us* ⊆ {*u*} ∪ *set as*
$\Longrightarrow$ [*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* $\Longrightarrow$ *xs* = (*as* @ *u* # *bs*)
$\Longrightarrow$ [*v*←*xs*. *v* = *u* ∨ *v* ∈ *set us*] = *u* # *us*
$\Longrightarrow$ *set zs* ∩ *set xs* ⊆ {*u*} ∪ *set as*
$\Longrightarrow$ [*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* $\Longrightarrow$ *xs* = (*as* @ *u* # *bs*)
$\Longrightarrow$ [*v*←*xs*. *v* = *u* ∨ *v* ∈ *set us*] = *u* # *us*
$\Longrightarrow$ *set zs* ∩ *set xs* ⊆ {*u*} ∪ *set as*
$\Longrightarrow$ [*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** $v \in set\ (verticesFrom\ f\ v')$ **by** (*auto simp*: *pre-subdivFace'-def before-def*)
then have $verticesFrom\ f\ v' =$
$fst\ (splitAt\ v\ (verticesFrom\ f\ v'))\ @\ v\ \#\ snd\ (splitAt\ v\ (verticesFrom\ f\ v'))$
by (*auto dest*: *splitAt-ram*)
then have *m*: $v'\ \#\ tl\ (verticesFrom\ f\ v')$
$=\ fst\ (splitAt\ v\ (verticesFrom\ f\ v'))\ @\ v\ \#\ snd\ (splitAt\ v\ (verticesFrom\ f\ v'))$
by (*simp add*: *verticesFrom-split*)

then have *vv'*: $v \neq v' \Longrightarrow fst\ (splitAt\ v\ (verticesFrom\ f\ v'))$
$= v'\ \#\ tl\ (fst\ (splitAt\ v\ (verticesFrom\ f\ v')))$
by (*cases fst* $(splitAt\ v\ (verticesFrom\ f\ v'))$) *auto*

ultimately have $v \neq v' \Longrightarrow vertices\ f21$
$= u\ \#\ snd\ (splitAt\ u\ (verticesFrom\ f\ v'))\ @\ v'\ \#\ tl\ (fst\ (splitAt\ v\ (verticesFrom\ f\ v')))\ @\ v\ \#\ ws$
by *auto*

moreover
with *f21* **have** *rule2*: $v' \in \mathcal{V}\ f21$ **by** *auto*
with *dist-f21* **have** *dist-f21-v'*: $distinct\ (verticesFrom\ f21\ v')$ **by** *auto*

ultimately have *m1*: $v \neq v' \Longrightarrow verticesFrom\ f21\ v'$
$= v'\ \#\ tl\ (fst\ (splitAt\ v\ (verticesFrom\ f\ v')))\ @\ v\ \#\ ws\ @\ u\ \#\ snd\ (splitAt\ u\ (verticesFrom\ f\ v'))$
apply *auto*
apply $(subgoal\text{-}tac\ snd\ (splitAt\ v'\ (vertices\ f21)) = tl\ (fst\ (splitAt\ v\ (verticesFrom\ f\ v')))\ @\ v\ \#\ ws)$
apply $(subgoal\text{-}tac\ fst\ (splitAt\ v'\ (vertices\ f21)) = u\ \#\ snd\ (splitAt\ u\ (verticesFrom\ f\ v')))$
apply $(subgoal\text{-}tac\ verticesFrom\ f21\ v' = v'\ \#\ snd\ (splitAt\ v'\ (vertices\ f21))\ @\ fst\ (splitAt\ v'\ (vertices\ f21)))$
apply *simp*
apply (*intro verticesFrom-v dist-f21*) **apply** *force*
apply $(subgoal\text{-}tac\ distinct\ (vertices\ f21))$ **apply** *simp*
apply (*rule-tac dist-f21*)
apply $(subgoal\text{-}tac\ distinct\ (vertices\ f21))$ **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*: $v = v' \Longrightarrow verticesFrom\ f21\ v' = v'\ \#\ ws\ @\ u\ \#\ snd\ (splitAt\ u\ (verticesFrom\ f\ v'))$
apply *auto* **apply** (*intro verticesFrom-v dist-f21*) **by** *simp*

from *pre-add* **have** *u*: $u \in set\ (verticesFrom\ f\ v')$ **by** (*fastforce simp*: *pre-subdivFace'-def before-def*)
then have *split-u*: $verticesFrom\ f\ v'$
$= fst\ (splitAt\ u\ (verticesFrom\ f\ v'))\ @\ u\ \#\ snd\ (splitAt\ u\ (verticesFrom\ f\ v'))$

by (*auto dest*!: *splitAt-ram*)

then have *rule1'*: [*v* ← *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′*: $u \neq v'$ **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** $v \in$ *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*: $\bigwedge$ *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*: $\bigwedge$ *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* $x \in set\ ws \cap set\ (snd\ (splitAt\ u\ (verticesFrom\ f\ v')))$)
apply *simp* **apply** (*elim conjE*)
apply (*thin-tac set ws* $\cap$ *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′* ∈ $\mathcal{V}$ *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* ∈ $\mathcal{F}$ *g*)
apply (*induct g*) **apply** (*simp add*: *edges-graph-def*) **apply** (*rule disjI1*)
apply (*rule bexI*) **apply** *simp* **apply** *simp*
apply (*drule replace1*) **apply** *simp* **by** (*simp add*: *f21-split-face*)

have *rule6*: *0* < |*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* $y \notin \mathcal{V}$ *f12*) **defer apply** (*rule removeNones-vol-not-f12*)
apply *simp*
apply (*simp add*: *is-duplicateEdge-def*)
apply (*subgoal-tac* $y \in$ *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* ∈ *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* ∈ *set ys* ∧ *v* ∈ *set ys* ∧ *u* ≠ *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*)
⟹ $f \in \mathcal{F}\ g$ ⟹ ($f \cdot v = u \longrightarrow n \neq 0$) ⟹ $\mathcal{V}\ f \subseteq \mathcal{V}\ g$
⟹ *pre-splitFace g v u f* [*countVertices g* ..< *countVertices g* + *n*]
proof −
assume *pre-add*: *pre-subdivFace′ g f v′ v n* ((*Some u*) # *vol*) **and** *f*: $f \in \mathcal{F}\ g$
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* ⟹ *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* ⟹ *distinct* (*verticesFrom f v′*) ⟹ *v* ≠ *v′* ⟹ ¬ *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

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′ ∈ $\mathcal{V}$ f*)
apply (*thin-tac u ∈ $\mathcal{V}$ f*)

apply (*simp add*: *verticesFrom-is-nextElem*)
apply (*rule ccontr*) **apply** *simp*
apply (*subgoal-tac v′ ∈ $\mathcal{V}$ f*)
apply (*drule verticesFrom-is-nextElem-hd*) **apply** *simp+*

apply (*elim conjE*) **apply** (*drule help2*)
apply *simp* **apply** *simp*
apply (*subgoal-tac is-nextElem* (*vertices f*) *u v = is-nextElem* (*verticesFrom f v′*) *u v*)
apply *simp*
apply (*rule verticesFrom-is-nextElem*) **by** *simp*
ultimately

show *?thesis*
apply (*simp add*: *pre-subdivFace′-def*)
apply (*unfold pre-splitFace-def*)
apply *simp*
apply (*cases 0 < n*) **apply** (*induct g*) **apply** (*simp add*: *ivl-disj-int*)
apply (*auto simp*: *invalidVertexList-def is-duplicateEdge-def*)
done
qed

lemma *pre-subdivFace′-Some1*:
pre-subdivFace′ g f v′ v n ((*Some u*) # *vol*)
$\Longrightarrow$ *f* $\in$ $\mathcal{F}$ *g* $\Longrightarrow$ (*f* $\cdot$ *v = u* $\longrightarrow$ *n* $\neq$ *0*) $\Longrightarrow$ $\mathcal{V}$ *f* $\subseteq$ $\mathcal{V}$ *g*
$\Longrightarrow$ *f21 = fst* (*snd* (*splitFace g v u f* [*countVertices g ..< countVertices g + n*]))
$\Longrightarrow$ *g′ = snd* (*snd* (*splitFace g v u f* [*countVertices g ..< countVertices g + n*]))
$\Longrightarrow$ *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

14 Invariants of (Plane) Graphs

theory *Invariants*
imports *FaceDivisionProps*
begin

14.1 Rotation of face into normal form

definition *minVertex* :: *face* ⇒ *vertex* **where**
minVertex f ≡ *min-list* (*vertices f*)

definition *normFace* :: *face* ⇒ *vertex list* **where**
normFace ≡ λ*f*. *verticesFrom f* (*minVertex f*)

definition *normFaces* :: *face list* ⇒ *vertex list list* **where**
normFaces fl ≡ *map normFace fl*

lemma *normFaces-distinct*: *distinct* (*normFaces fl*) ⟹ *distinct fl*
apply (*induct fl*) **by** (*auto simp*: *normFace-def normFaces-def*)

14.2 Minimal (plane) graph properties

definition *minGraphProps′* :: *graph* ⇒ *bool* **where**
minGraphProps′ g ≡ ∀ *f* ∈ $\mathcal{F}$ *g*. 2 < |*vertices f*| ∧ *distinct* (*vertices f*)

definition *edges-sym* :: *graph* ⇒ *bool* **where**
edges-sym g ≡ ∀ *a b*. (*a*,*b*) ∈ *edges g* ⟶ (*b*,*a*) ∈ *edges g*

definition *faceListAt-len* :: *graph* ⇒ *bool* **where**
faceListAt-len g ≡ (*length* (*faceListAt g*) = *countVertices g*)

definition *facesAt-eq* :: *graph* ⇒ *bool* **where**
facesAt-eq g ≡ ∀ *v* ∈ $\mathcal{V}$ *g*. *set*(*facesAt g v*) = {*f*. *f* ∈ $\mathcal{F}$ *g* ∧ *v* ∈ $\mathcal{V}$ *f*}

definition *facesAt-distinct* :: *graph* ⇒ *bool* **where**
facesAt-distinct g ≡ ∀ *v* ∈ $\mathcal{V}$ *g*. *distinct* (*normFaces* (*facesAt g v*))

definition *faces-distinct* :: *graph* ⇒ *bool* **where**
faces-distinct g ≡ *distinct* (*normFaces* (*faces g*))

definition *faces-subset* :: *graph* ⇒ *bool* **where**

faces-subset $g \equiv \forall f \in \mathcal{F}\ g.\ \mathcal{V}\ f \subseteq \mathcal{V}\ g$

definition *edges-disj* :: *graph* ⇒ *bool* **where**
edges-disj $g \equiv$
$\forall f \in \mathcal{F}\ g.\ \forall f' \in \mathcal{F}\ g.\ f \neq f' \longrightarrow \mathcal{E}\ f \cap \mathcal{E}\ f' = \{\}$

definition *face-face-op* :: *graph* ⇒ *bool* **where**
face-face-op $g \equiv |faces\ g| \neq 2 \longrightarrow$
$(\forall f{\in}\mathcal{F}\ g.\ \forall f'{\in}\mathcal{F}\ g.\ f \neq f' \longrightarrow \mathcal{E}\ f \neq (\mathcal{E}\ f')^{-1})$

definition *one-final-but* :: *graph* ⇒ (*vertex* × *vertex*)*set* ⇒ *bool* **where**
one-final-but $g\ E \equiv$
$\forall f \in \mathcal{F}\ g.\ \neg\ final\ f \longrightarrow$
$(\forall (a,b){\in}\mathcal{E}\ f\ -\ E.\ (b,a) : E \vee (\exists f'{\in}\mathcal{F}\ g.\ final\ f' \wedge (b,a) \in \mathcal{E}\ f'))$

definition *one-final* :: *graph* ⇒ *bool* **where**
one-final $g \equiv$ *one-final-but* $g\ \{\}$

definition *minGraphProps* :: *graph* ⇒ *bool* **where**
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

definition *inv* :: *graph* ⇒ *bool* **where**
inv $g \equiv$ *minGraphProps* $g \wedge$ *one-final* $g \wedge |faces\ g| \geq 2$

lemma *facesAt-distinctI*:
$(\bigwedge v.\ v \in \mathcal{V}\ g \Longrightarrow distinct\ (normFaces\ (facesAt\ g\ \ v))) \Longrightarrow facesAt\text{-}distinct\ g$
by (*simp add*: *facesAt-distinct-def*)

lemma *minGraphProps2*:
$minGraphProps\ g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow 2 < |vertices\ f|$
by (*unfold minGraphProps-def minGraphProps'-def*) *auto*

lemma *mgp-vertices3*:
$minGraphProps\ g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow |vertices\ f| \geq 3$
by(*auto dest*:*minGraphProps2*)

lemma *mgp-vertices-nonempty*:
$minGraphProps\ g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow vertices\ f \neq []$
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*; *v* : $\mathcal{V}$ *g*; *f* ∈ *set* (*facesAt g v*)⟧ $\Longrightarrow$ *f* ∈ $\mathcal{F}$ *g*
by(*auto simp*: *facesAt-def facesAt-eq-def minGraphProps-def minGraphProps'-def faceListAt-len-def split*:*split-if-asm*)

lemma *minGraphProps6*:
minGraphProps g $\Longrightarrow$ *v* : $\mathcal{V}$ *g* $\Longrightarrow$ *f* ∈ *set* (*facesAt g v*) $\Longrightarrow$ *v* ∈ $\mathcal{V}$ *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* ∈ $\mathcal{F}$ *g* $\Longrightarrow$ *v* ∈ $\mathcal{V}$ *f* $\Longrightarrow$ *v* ∈ $\mathcal{V}$ *g*
by (*unfold minGraphProps-def faces-subset-def*) *auto*

lemma *minGraphProps7*:
minGraphProps g $\Longrightarrow$ *f* ∈ $\mathcal{F}$ *g* $\Longrightarrow$ *v* ∈ $\mathcal{V}$ *f* $\Longrightarrow$ *f* ∈ *set* (*facesAt g v*)
apply(*frule* (*2*) *minGraphProps9*)
by (*unfold minGraphProps-def facesAt-eq-def*) *simp*

lemma *minGraphProps-facesAt-eq*: *minGraphProps g* $\Longrightarrow$
v ∈ $\mathcal{V}$ *g* $\Longrightarrow$ *set* (*facesAt g v*) = {*f* ∈ $\mathcal{F}$ *g*. *v* ∈ $\mathcal{V}$ *f*}
by (*simp add*: *minGraphProps-def facesAt-eq-def*)

lemma *mgp-dist-facesAt*[*simp*]:
minGraphProps g $\Longrightarrow$ *v* : $\mathcal{V}$ *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 $\Longrightarrow$ *v* : $\mathcal{V}$ *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* ∈ $\mathcal{V}$ *g* $\Longrightarrow$ *distinct* (*normFaces* (*faceListAt g* ! *v*))
apply (*frule* (*1*) *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 $\Longrightarrow$ (*a*, *b*) $\in$ *edges g* $\Longrightarrow$ (*b*, *a*) $\in$ *edges g*
apply (*unfold minGraphProps-def edges-sym-def*)
apply (*elim conjE allE impE*)
by *simp*+

lemma *minGraphProps11*:
minGraphProps g $\Longrightarrow$ *distinct* (*normFaces* (*faces g*))
by (*unfold minGraphProps-def faces-distinct-def*) *simp*

lemma *minGraphProps11'*:
minGraphProps g $\Longrightarrow$ *distinct* (*faces g*)
by(*simp add*: *minGraphProps11 normFaces-distinct*)

lemma *minGraphProps12*:
minGraphProps g $\Longrightarrow$ *f* $\in \mathcal{F}$ *g* $\Longrightarrow$ (*a*,*b*) $\in \mathcal{E}$ *f* $\Longrightarrow$ (*b*,*a*) $\notin \mathcal{E}$ *f*
apply (*subgoal-tac distinct* (*vertices f*)) **apply** (*simp add*: *is-nextElem-def*)
apply (*case-tac vertices f* = []) **apply** (*drule minGraphProps2*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac a* = *last* (*vertices f*) $\wedge$ *b* = *hd* (*vertices f*)) **apply** (*case-tac vertices f*) **apply** *simp*
apply (*case-tac list rule*: *rev-exhaust*) **apply** (*drule minGraphProps2*) **apply** *simp* **apply** *simp*
apply (*case-tac ys*) **apply** (*drule minGraphProps2*) **apply** *simp* **apply** *simp*
apply (*simp del*: *distinct-append distinct.simps*)
apply (*rule conjI*) **apply** (*rule ccontr*) **apply** (*simp del*: *distinct-append distinct.simps*) **apply** (*drule is-sublist-distinct-prefix*) **apply** *simp* **apply** (*simp add*: *is-prefix-def*) **apply** *simp*
apply (*rule conjI*) **apply** (*simp add*: *is-sublist-def*) **apply** (*elim exE*) **apply** (*intro allI*) **apply** (*rule ccontr*)
apply (*simp del*: *distinct-append distinct.simps*)
apply (*subgoal-tac asa* = *as* @ [*a*]) **apply** *simp* **apply** (*rule dist-at1*) **apply** *assumption* **apply** *force* **apply** (*rule sym*) **apply** *simp*
apply (*subgoal-tac is-sublist* [*a*, *b*] (*vertices f*))
apply (*rule impI*) **apply** (*rule ccontr*)
apply (*simp add*: *is-sublist-def del*: *distinct-append distinct.simps*)

apply (*subgoal-tac last* (*vertices f*) = *b* ∧ *hd* (*vertices f*) = *a*)
apply (*thin-tac a* = *hd* (*vertices f*)) **apply** (*thin-tac b* = *last* (*vertices f*)) **apply** (*elim conjE*)
apply (*elim exE*)
apply (*case-tac as*)
apply (*case-tac bs rule*: *rev-exhaust*) **apply** (*drule minGraphProps2*) **apply** *simp* **apply** *simp*
apply *simp+*
apply (*rule minGraphProps3*) **by** *simp+*

lemma *minGraphProps7'*: *minGraphProps g* ⟹
f ∈ ℱ *g* ⟹ *v* ∈ 𝒱 *f* ⟹ *f* ∈ *set* (*faceListAt g* ! *v*)
apply (*frule minGraphProps7*) **apply** *assumption+*
by (*simp add*: *facesAt-def split*: *split-if-asm*)

lemma *mgp-edges-disj*:
⟦ *minGraphProps g*; *f* ≠ *f'*; *f* ∈ ℱ *g*; *f'* ∈ ℱ *g* ⟧ ⟹
uv ∈ ℰ *f* ⟹ *uv* ∉ ℰ *f'*
by (*simp add*:*minGraphProps-def edges-disj-def*) *blast*

lemma *one-final-but-antimono*:
one-final-but g E ⟹ *E* ⊆ *E'* ⟹ *one-final-but g E'*
apply(*unfold one-final-but-def*)
apply *blast*
done

lemma *one-final-antimono*: *one-final g* ⟹ *one-final-but g E*
apply(*unfold one-final-def one-final-but-def*)
apply *blast*
done

lemma *inv-two-faces*: *inv g* ⟹ |*faces g*| ≥ *2*
by(*simp add*:*inv-def*)

lemma *inv-mgp*[*simp*]: *inv g* ⟹ *minGraphProps g*
by(*simp add*:*inv-def*)

lemma *makeFaceFinal-id*[*simp*]: *final f* ⟹ *makeFaceFinal f g* = *g*
apply(*cases g*)
apply (*simp add*:*makeFaceFinal-def makeFaceFinalFaceList-def*
setFinal-eq-iff [*THEN iffD2*])
done

lemma *inv-one-finalD'*:
⟦ *inv g*; *f* ∈ ℱ *g*; ¬ *final f*; (*a*,*b*) ∈ ℰ *f* ⟧ ⟹
∃*f'* ∈ ℱ *g*. *final f'* ∧ *f'* ≠ *f* ∧ (*b*,*a*) ∈ ℰ *f'*
apply(*unfold inv-def one-final-def one-final-but-def*)

apply *blast*
done

lemmas *minGraphProps* =
minGraphProps2 minGraphProps3 minGraphProps4
minGraphProps5 minGraphProps6 minGraphProps7 minGraphProps8
minGraphProps9

lemma *mgp-no-loop*[*simp*]:
minGraphProps g $\Longrightarrow$ *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*:
minGraphProps g $\Longrightarrow$ *v* : $\mathcal{V}$ *g* $\Longrightarrow$ *f* $\in$ *set* (*facesAt g v*) $\Longrightarrow$ *f* $\cdot$ *v* $\neq$ *v*
by(*blast dest*:*mgp-no-loop minGraphProps5 minGraphProps6*)

lemma *edge-pres-faceAt*:
⟦ *minGraphProps g*; *u* : $\mathcal{V}$ *g*; *f* $\in$ *set*(*facesAt g u*); (*u*,*v*) $\in$ $\mathcal{E}$ *f* ⟧ $\Longrightarrow$
f $\in$ *set*(*facesAt g v*)
apply(*auto simp*:*edges-face-eq*)
apply(*rule minGraphProps7*, *assumption*)
apply(*blast intro*:*minGraphProps*)
apply(*simp*)
done

lemma *in-facesAt-nextVertex*:
minGraphProps g $\Longrightarrow$ *v* : $\mathcal{V}$ *g* $\Longrightarrow$ *f* $\in$ *set*(*facesAt g v*) $\Longrightarrow$ *f* $\in$ *set*(*facesAt g* (*f* $\cdot$ *v*))
apply(*subgoal-tac* (*v*,*f* $\cdot$ *v*) $\in$ $\mathcal{E}$ *f*)
apply(*blast intro*:*edge-pres-faceAt*)
by(*blast intro*: *nextVertex-in-edges minGraphProps*)

lemma *mgp-edge-face-ex*:
assumes [*intro*]: *minGraphProps g v* : $\mathcal{V}$ *g*
and *fv*: *f* $\in$ *set*(*facesAt g v*) **and** *uv*: (*u*,*v*) $\in$ $\mathcal{E}$ *f*
shows $\exists f' \in$ *set*(*facesAt g v*). (*v*,*u*) $\in$ $\mathcal{E}$ *f'*
proof −
from *fv* **have** *f* $\in$ $\mathcal{F}$ *g* **by**(*blast intro*:*minGraphProps*)
with *uv* **have** (*u*,*v*) $\in$ $\mathcal{E}$ *g* **by**(*auto simp*:*edges-graph-def*)
hence (*v*,*u*) $\in$ $\mathcal{E}$ *g* **by**(*blast intro*:*minGraphProps10*)
then obtain *f'* **where** *f'*: *f'* $\in$ $\mathcal{F}$ *g* **and** *vu*: (*v*,*u*) $\in$ $\mathcal{E}$ *f'*
by(*auto simp*:*edges-graph-def*)
from *vu* **have** *v* $\in$ $\mathcal{V}$ *f'* **by**(*auto simp*:*edges-face-eq*)
with *f'* **have** *f'* $\in$ *set*(*facesAt g v*) **by**(*blast intro*:*minGraphProps*)
with *vu* **show** *?thesis* **by** *blast*

qed

lemma *nextVertex-in-graph*:
minGraphProps g $\Longrightarrow$ *v* : $\mathcal{V}$ *g* $\Longrightarrow$ *f* $\in$ *set*(*facesAt g v*) $\Longrightarrow$ *f* $\cdot$ *v* : $\mathcal{V}$ *g*
by(*blast intro*: *minGraphProps9 minGraphProps5 minGraphProps6 nextVertex-in-face*)

lemma *mgp-nextVertex-face-ex2*:
assumes *mgp*[*intro*]: *minGraphProps g v* : $\mathcal{V}$ *g* **and** *f*: *f* $\in$ *set*(*facesAt g v*)
shows $\exists f' \in$ *set*(*facesAt g* (*f* $\cdot$ *v*)). *f'* $\cdot$ (*f* $\cdot$ *v*) = *v*
proof −
from *f* **have** (*v*,*f* $\cdot$ *v*) $\in$ $\mathcal{E}$ *f*
by(*blast intro*: *nextVertex-in-edges minGraphProps*)
with *in-facesAt-nextVertex*[*OF mgp f*]
mgp-edge-face-ex[*OF mgp*(*1*) *nextVertex-in-graph*[*OF mgp f*]]
obtain *f'* :: *face* **where** *f'* $\in$ *set*(*facesAt g* (*f* $\cdot$ *v*))
and (*f* $\cdot$ *v*,*v*) $\in$ $\mathcal{E}$ *f'*
by(*blast*)
thus *?thesis* **by** (*auto simp*: *edges-face-eq*)
qed

lemma *inv-finals-nonempty*: *inv g* $\Longrightarrow$ *finals g* $\neq$ []
apply(*frule inv-two-faces*)
apply(*clarsimp simp*:*filter-empty-conv finals-def*)
apply(*subgoal-tac faces g* $\neq$ []*)*
prefer *2* **apply** *clarsimp*
apply(*simp add*:*neq-Nil-conv*)
apply *clarify*
apply(*rename-tac f fs*)
apply(*case-tac final f*)
apply *simp*
apply(*frule mgp-vertices-nonempty*[*OF inv-mgp*])
apply *fastforce*
apply(*clarsimp simp*:*neq-Nil-conv*)
apply(*rename-tac v vs*)
apply(*subgoal-tac v* $\in$ $\mathcal{V}$ *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

14.3 *containsDuplicateEdge*

definition
containsUnacceptableEdgeSnd' :: (*nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*) $\Rightarrow$ *nat list* $\Rightarrow$ *bool* **where**

```
containsUnacceptableEdgeSnd' N is ≡
 (∃ k < |is| − 2. let i0 = is!k; i1 = is!(k+1); i2 = is!(k+2) in
 N i1 i2 ∧ (i0 < i1) ∧ (i1 < i2))

lemma containsUnacceptableEdgeSnd-eq:
  containsUnacceptableEdgeSnd N v is = containsUnacceptableEdgeSnd' N (v#is)
proof (induct is arbitrary: v)
  case Nil then show ?case by (simp add: containsUnacceptableEdgeSnd'-def)
next
  case (Cons i is) then show ?case
    proof (rule-tac iffI)
      assume vors: containsUnacceptableEdgeSnd N v (i # is)
      then show containsUnacceptableEdgeSnd' N (v # i # is)
        apply (cases is) apply simp apply simp
        apply (simp split: 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*]

14.4 *replacefacesAt*

primrec *replacefacesAt2* ::
nat list ⇒ *face* ⇒ *face list* ⇒ *face list list* ⇒ *face list list* **where**
replacefacesAt2 [] *f fs F = F* |
replacefacesAt2 (*n*#*ns*) *f fs F* =
(*if n* < |*F*|
then replacefacesAt2 ns f fs (*F* [*n*:=*replace f fs* (*F*!*n*)])
else replacefacesAt2 ns f fs F)

lemma *replacefacesAt-eq*[*THEN eq-reflection*]:
replacefacesAt ns oldf newfs F = replacefacesAt2 ns oldf newfs F
by (*induct ns arbitrary*: *F*) (*auto simp add*: *replacefacesAt-def*)

lemma *replacefacesAt2-notin*:
i ∉ *set is* ⟹ (*replacefacesAt2 is olfF newFs Fss*)!*i* = *Fss*!*i*
proof (*induct is arbitrary*: *Fss*)
case *Nil* **then show** *?case* **by** (*simp*)
next
case (*Cons j js*) **then show** *?case*
by (*cases j* < |*Fss*|) (*auto*)
qed

lemma *replacefacesAt2-in*:
i ∈ *set is* ⟹ *distinct is* ⟹ *i* < |*Fss*| ⟹
(*replacefacesAt2 is olfF newFs Fss*)!*i* = *replace olfF newFs* (*Fss* !*i*)
proof (*induct is arbitrary*: *Fss*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons j js*)
then have *j* = *i* ∧ *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 (*auto simp add*: *replacefacesAt2-notin*)
next
assume *i* ∈ *set js i* ≠ *j* **with** *Cons* **show** *?thesis* **by** *simp*
qed
qed

lemma *distinct-replacefacesAt21*:
i < |*Fss*| ⟹ *i* ∈ *set is* ⟹ *distinct is* ⟹ *distinct* (*Fss*!*i*) ⟹ *distinct newFs*
⟹
set (*Fss* ! *i*) ∩ *set newFs* ⊆ {*olfF*} ⟹
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* ∧ *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*: *replacefacesAt2-notin distinct-replace*)
next
assume *i* ∈ *set js i* ≠ *j* **with** *Cons* **show** *?thesis*
by (*simp add*: *replacefacesAt2-in distinct-replace*)
qed
qed

lemma *distinct-replacefacesAt22*:
i < |*Fss*| ⟹ *i* ∉ *set is* ⟹ *distinct is* ⟹ *distinct* (*Fss*!*i*) ⟹ *distinct newFs*
⟹
set (*Fss* ! *i*) ∩ *set newFs* ⊆ {*olfF*} ⟹
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* ≠ *j* **by** *auto*
with *Cons* **show** *?case*
by (*simp add*: *replacefacesAt2-notin distinct-replace*)
qed

lemma *distinct-replacefacesAt2-2*:
i < |*Fss*| ⟹ *distinct is* ⟹ *distinct* (*Fss*!*i*) ⟹ *distinct newFs* ⟹
set (*Fss* ! *i*) ∩ *set newFs* ⊆ {*olfF*} ⟹
distinct ((*replacefacesAt2 is olfF newFs Fss*)! *i*)
by (*cases i* ∈ *set is*)
(*auto intro*: *distinct-replacefacesAt21 distinct-replacefacesAt22*)

lemma *replacefacesAt2-nth1*:
k ∉ *set ns* ⟹ (*replacefacesAt2 ns oldf newfs F*) ! *k* = *F* ! *k*
by (*induct ns arbitrary*: *F*) *auto*

lemma *replacefacesAt2-nth1′*: *k* ∈ *set ns* ⟹ *k* < |*F*| ⟹ *distinct ns* ⟹
(*replacefacesAt2 ns oldf newfs F*) ! *k* = (*replace oldf newfs* (*F*!*k*))

apply (*induct ns arbitrary*: *F*)
apply *auto*
apply (*simp add*: *replacefacesAt2-nth1*)+
by (*case-tac a* = *k*) *auto*

lemma *replacefacesAt2-nth2*: *k* < |*F*| ⟹
(*replacefacesAt2* [*k*] *oldf newfs F*) ! *k* = *replace oldf newfs* (*F*!*k*)
by (*auto*)

lemma *replacefacesAt2-length*[*simp*]:
|*replacefacesAt2 nvs f' f'' vs*| = |*vs*|
by (*induct nvs arbitrary*: *vs*) *simp-all*

lemma *replacefacesAt2-nth*: *k* ∈ *set ns* ⟹ *k* < |*F*| ⟹ *oldf* ∉ *set newfs* ⟹
distinct (*F*!*k*) ⟹ *distinct newfs* ⟹ *oldf* ∈ *set* (*F*!*k*) ⟶ *set newfs* ∩ *set* (*F*!*k*)
⊆ {*oldf*} ⟹
(*replacefacesAt2 ns oldf newfs F*) ! *k* = (*replace oldf newfs* (*F*!*k*))
proof (*induct ns arbitrary*: *F*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons n ns*) **then show** *?case*
apply (*simp only*: *replacefacesAt2.simps*)
apply *simp* **apply** (*case-tac n* = *k*)
apply (*simp*)
apply (*subgoal-tac replacefacesAt2 ns oldf newfs* (*F*[*k* := *replace oldf newfs*
(*F* ! *k*)]) ! *k* =
replace oldf newfs ((*F*[*k* := *replace oldf newfs* (*F* ! *k*)]) ! *k*))
apply *simp*
apply (*case-tac k* ∈ *set ns*) **apply** (*rule Cons*) **apply** *simp*+
apply (*rule replace-distinct*) **apply** *simp* **apply** *simp*
apply *simp*
apply *simp*
apply (*simp add*:*distinct-set-replace*)
apply (*simp add*: *replacefacesAt2-nth1*)
by *simp*
qed

lemma *replacefacesAt-notin*:
i ∉ *set is* ⟹ (*replacefacesAt is olfF newFs Fss*)!*i* = *Fss*!*i*
by (*simp add*: *replacefacesAt-eq replacefacesAt2-notin*)

lemma *replacefacesAt-in*:
i ∈ *set is* ⟹ *distinct is* ⟹ *i* < |*Fss*| ⟹
(*replacefacesAt is olfF newFs Fss*)!*i* = *replace olfF newFs* (*Fss* !*i*)
by (*simp add*: *replacefacesAt-eq replacefacesAt2-in*)

lemma *replacefacesAt-length*[*simp*]: |*replacefacesAt nvs f'* [*f''*] *vs*| = |*vs*|

by (*simp add*: *replacefacesAt-eq*)

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-nth*: $k \in$ *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*: *x* $\in$ *set* (*replacefacesAt2 ns oldf newfs F* ! *k*) $\Longrightarrow$ *x* $\in$ *set* (*F*!*k*) $\vee$ *x* $\in$ *set newfs*
proof (*induct ns arbitrary*: *F*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons n ns*)
then show *?case*
apply(*simp add*: *split*: *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 *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*)

14.5 *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(*cases f*)
apply(*cases f'*)
apply(*rename-tac vs ft vs' ft'*)
apply(*case-tac vs* = [])
apply(*simp add*:*vertices-face-def minVertex-def*)
apply(*subgoal-tac vs'* ≠ [])
prefer *2* **apply** *clarsimp*
apply(*simp add*:*vertices-face-def minVertex-def min-list-conv-Min*
insert-absorb del:*Min-insert*)
done

lemma *normFace-replace-in*:
normFace a ∈ *set* (*normFaces* (*replace oldF newFs fs*)) ⟹
normFace a ∈ *set* (*normFaces newFs*) ∨ *normFace a* ∈ *set* (*normFaces fs*)
apply (*induct fs*) **apply** *simp*
apply (*auto simp add*: *normFaces-def split*:*split-if-asm*)
done

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 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 < |*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 $j = i$ $i \notin set\ js$ **with** *Cons* **show** *?thesis*
by (*simp add*: *replacefacesAt-notin distinct-replace-norm*)
next
assume $i \in set\ js$ $i \neq j$ **with** *Cons* **show** *?thesis*
by (*simp add*: *replacefacesAt-in distinct-replace-norm*)
qed
qed

lemma *distinct-replacefacesAt2-norm*:
$i < |Fss| \Longrightarrow i \notin set\ is \Longrightarrow distinct\ is \Longrightarrow distinct\ (normFaces\ (Fss!i)) \Longrightarrow$
distinct (*normFaces newFs*) $\Longrightarrow$
set (*normFaces* (*Fss* ! *i*)) $\cap$ *set* (*normFaces newFs*) $\subseteq \{\} \Longrightarrow$
distinct (*normFaces* ((*replacefacesAt is 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*:
$i < |Fss| \Longrightarrow distinct\ is \Longrightarrow distinct\ (normFaces\ (Fss!i)) \Longrightarrow distinct\ (normFaces$
newFs) $\Longrightarrow$
set (*normFaces* (*Fss* ! *i*)) $\cap$ *set* (*normFaces newFs*) $\subseteq \{\} \Longrightarrow$
distinct (*normFaces* ((*replacefacesAt is olfF newFs Fss*)! *i*))
by (*cases* $i \in set\ is$)
(*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* ∈ *set* (*verticesFrom f* (*minVertex f*)))
apply (*subgoal-tac a* ∉ *set* (*verticesFrom f'* (*minVertex f'*))) **apply** *force*
apply (*subgoal-tac* (*vertices f'*) ≅ (*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*) ≅ (*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*)

14.6 Invariants of *splitFace*

lemma *splitFace-holds-minGraphProps'*:
pre-splitFace g' v a f' vs $\Longrightarrow$ *minGraphProps' g'* $\Longrightarrow$
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* $\in \mathcal{F}$ *g'*) **apply** *simp*
apply *safe*
apply (*simp add*: *split-face-def*) **apply** *safe* **apply** *simp* **apply** (*drule pre-FaceDiv-between1*)
apply *simp*
apply (*frule-tac replace1*)
apply *simp-all*
apply (*simp add*: *split-face-def*) **apply** *safe* **apply** *simp*
apply (*drule pre-FaceDiv-between2*) **apply** *simp*
apply (*drule splitFace-split*)
apply *safe*
apply *simp*
apply (*subgoal-tac pre-splitFace g' v a f' vs*)
apply (*drule splitFace-distinct2*)+ **apply** *simp*+
apply (*subgoal-tac pre-splitFace g' v a f' vs*)
apply (*drule splitFace-distinct1*)+
by *simp*+

lemma *splitFace-holds-faceListAt-len*:
pre-splitFace g' v a f' vs $\Longrightarrow$ *minGraphProps g'* $\Longrightarrow$
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*:
assumes *pre*: *pre-splitFace g ram1 ram2 oldF newVs*
and *props*: *minGraphProps g*
and *spl*: (*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs*
shows *f12* $\notin \mathcal{F}$ *g*
proof (*cases newVs*)
case *Nil* **with** *pre* **have** (*ram2*, *ram1*) $\notin$ *edges g*
by (*unfold pre-splitFace-def*) *auto*
moreover from *Nil pre*
have (*ram2*, *ram1*) $\in$ *edges f12*
apply (*rule-tac splitFace-empty-ram2-ram1-in-f12*)
apply (*auto simp*: *Nil*[*symmetric*])
apply (*rule spl*)
done
ultimately show *?thesis* **by** (*auto simp add*: *edges-graph-def*)

next
case (*Cons v vs*)
with *pre* **have** $v \notin \mathcal{V}\ g$
by (*auto simp*: *pre-splitFace-def*)
moreover from *Cons spl* **have** $v \in \mathcal{V}\ f12$
by (*simp add*: *splitFace-f12-new-vertices*)
moreover note *props*
ultimately show *?thesis* **by** (*auto dest*: *minGraphProps*)
qed

lemma *splitFace-new-f12-norm*:
assumes *pre*: *pre-splitFace g ram1 ram2 oldF newVs*
and *props*: *minGraphProps g*
and *spl*: (*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs*
shows *normFace f12* $\notin$ *set* (*normFaces* (*faces g*))
proof (*cases newVs*)
case *Nil* **with** *pre* **have** (*ram2*, *ram1*) $\notin$ *edges g*
by (*auto simp*: *pre-splitFace-def*)
moreover
from *pre spl* [*symmetric*] **have** *dist-f12*: *distinct* (*vertices f12*)
apply (*drule-tac splitFace-distinct2*) **by** *simp*
moreover
from *Nil pre*
have (*ram2*, *ram1*) $\in$ *edges f12*
apply (*rule-tac splitFace-empty-ram2-ram1-in-f12*)
apply (*auto simp*: *Nil*[*symmetric*])
apply (*rule spl*)
done
moreover
with *dist-f12* **have** *vertices f12* $\neq$ []
apply (*simp add*: *is-nextElem-def*) **apply** (*case-tac vertices f12*) **apply** (*simp add*: *is-sublist-def*)
by *simp*
ultimately show *?thesis*
apply (*auto simp add*: *edges-graph-def*) **apply** (*frule normFace-in-cong*)
apply (*rule props*)
apply *assumption*
apply (*elim bexE*)
apply (*subgoal-tac* (*ram2*, *ram1*) $\in$ *edges f'*) **apply** *simp*
apply (*subgoal-tac* (*vertices f12*) $\cong$ (*vertices f'*)) **apply** (*frule congs-distinct*)
apply (*simp add*: *cong-face-def is-nextElem-congs-eq*)+
done
next
case (*Cons v vs*)
with *pre* **have** $v \notin \mathcal{V}\ g$ **by** (*auto simp*: *pre-splitFace-def*)
moreover from *Cons spl* **have** $v \in \mathcal{V}\ f12$
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

lemma splitFace-new-f21:
assumes pre: pre-splitFace g ram1 ram2 oldF newVs
and props: minGraphProps g
and spl: (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs
shows f21 ∉ ℱ g
proof (cases newVs)
  case Nil with pre have (ram1, ram2) ∉ edges g
    by (auto simp: pre-splitFace-def)
  moreover from Nil pre
  have (ram1, ram2) ∈ edges f21
    apply (rule-tac splitFace-empty-ram1-ram2-in-f21)
     apply (auto simp: Nil[symmetric])
    apply (rule spl)
    done
  ultimately show ?thesis by (auto simp add: edges-graph-def)
next
  case (Cons v vs)
  with pre have v ∉ 𝒱 g by (auto simp: pre-splitFace-def)
  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

lemma splitFace-new-f21-norm:
assumes pre: pre-splitFace g ram1 ram2 oldF newVs
and props: minGraphProps g
and spl: (f12, f21, newGraph) = splitFace g ram1 ram2 oldF newVs
shows normFace f21 ∉ set (normFaces (faces g))
proof (cases newVs)
  case Nil with pre have (ram1, ram2) ∉ edges g
    by (auto simp: pre-splitFace-def)
  moreover
  from pre spl [symmetric] have dist-f21: distinct (vertices f21)
    apply (drule-tac splitFace-distinct1) by simp
  moreover
  from Nil pre
  have (ram1, ram2) ∈ edges f21
    apply (rule-tac splitFace-empty-ram1-ram2-in-f21)
     apply (auto simp: Nil[symmetric])
    apply (rule spl)
    done
```

```
  moreover
  with dist-f21 have vertices f21 ≠ []
    apply (simp add: is-nextElem-def) apply (case-tac vertices f21) apply (simp
add: is-sublist-def)
    by simp
  ultimately show ?thesis apply (auto simp add: edges-graph-def) apply (frule
normFace-in-cong)
      apply (rule props)
     apply assumption
    apply (elim bexE)
    apply (subgoal-tac (ram1, ram2) ∈ edges f') apply simp
    apply (subgoal-tac (vertices f21) ≅ (vertices f')) apply (frule congs-distinct)
     apply (simp add: cong-face-def is-nextElem-congs-eq)+
    done
next
  case (Cons v vs)
  with pre have v ∉ 𝒱 g by (auto simp: pre-splitFace-def)
  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

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
by (frule splitFace-new-f21) (auto)

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
by (frule splitFace-new-f12) (auto)


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*:
assumes *pre*: *pre-splitFace g v w f* [*countVertices g..<countVertices g* + *n*]
and *mgp*: *minGraphProps g*
shows *facesAt-distinct* (*snd* (*snd* (*splitFace g v w f* [*countVertices g..<countVertices g* + *n*])))
proof −
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*: $f21 \notin \mathcal{F}\ g$
apply (*rule-tac splitFace-new-f21*) **by** *simp-all*
from *pre mgp fdg* **have** *new-f12-norm*: *normFace f12* $\notin$ *set* (*normFaces* (*faces g*))
apply (*rule-tac splitFace-new-f12-norm*) **by** *simp-all*
from *pre mgp fdg* **have** *new-f21-norm*: *normFace f21* $\notin$ *set* (*normFaces* (*faces g*))
apply (*rule-tac splitFace-new-f21-norm*) **by** *simp-all*

have *facesAt-distinct g'*
proof (*rule facesAt-distinctI*)
fix *x* **assume** *x*: $x \in \mathcal{V}\ g'$
show *distinct* (*normFaces* (*facesAt g' x*))
proof −
from *mgp pre* **have** *a*: $v < |faceListAt\ g|\ w < |faceListAt\ g|$
apply (*unfold pre-splitFace-def*)
apply (*simp-all add*: *minGraphProps4*)
by (*auto intro*: *minGraphProps9'*)
then show *?thesis*
proof (*cases* $x = w$)
case *True*
moreover with *pre* **have** $v \neq w$
by (*unfold pre-splitFace-def*) *simp*
moreover note *a x pre mgp*
ultimately show *?thesis*
apply −
apply (*unfold pre-splitFace-def*)
apply (*unfold g'-def splitFace-def facesAt-def*)
apply (*simp add*: *split-def nth-append*)
apply (*rule distinct-replace-norm*)
apply (*rule distinct-replacefacesAt-norm*)
apply *simp*
apply (*rule between-distinct*)
apply *simp*
apply (*rule distinct-replacefacesAt-norm*)
apply *assumption*
apply (*rule between-distinct*)
apply *simp*
apply (*rule minGraphProps8a'*) **apply** *assumption*+ **apply** (*simp add*: *minGraphProps4*)
apply (*simp add*: *normFaces-def*)

apply (*subgoal-tac set* (*faceListAt g* ! *w*) = $\{f \in \mathcal{F}\ g.\ w \in \mathcal{V}\ f\}$) **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* $w \in \mathcal{V}\ g$) **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* $w \notin$ *set* (*between* (*vertices f*) *v w*))
apply (*simp add*: *replacefacesAt-notin*)
apply (*subgoal-tac set* (*faceListAt g* ! *w*) = $\{f \in \mathcal{F}\ g.\ w \in \mathcal{V}\ f\}$)
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* $w \in \mathcal{V}\ g$) **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* $w \notin$ *set* (*between* (*vertices f*) *w v*))
apply (*simp add*: *replacefacesAt-notin*)
apply (*subgoal-tac* $w \notin$ *set* (*between* (*vertices f*) *v w*))
apply (*simp add*: *replacefacesAt-notin*)
apply (*subgoal-tac set* (*faceListAt g* ! *w*) = $\{f \in \mathcal{F}\ g.\ w \in \mathcal{V}\ f\}$)
apply (*subgoal-tac set* (*normFaces* (*faces g*)) $\cap$ {*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* $w \in \mathcal{V}\ g$) **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*: $v \neq w$
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*) = {$f \in \mathcal{F}$ *g*. *v* $\in \mathcal{V}$ *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 (*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-notin*)
apply (*subgoal-tac set* (*faceListAt g* ! *v*) = {$f \in \mathcal{F}$ *g*. *v* $\in \mathcal{V}$ *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 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* ∉ *set* (*between* (*vertices f*) *w v*))
apply (*simp add*: *replacefacesAt-notin*)
apply (*subgoal-tac v* ∉ *set* (*between* (*vertices f*) *v w*))
apply (*simp add*: *replacefacesAt-notin*)
apply (*subgoal-tac set* (*faceListAt g* ! *v*) = {*f* ∈ $\mathcal{F}$ *g*. *v* ∈ $\mathcal{V}$ *f*})
apply (*subgoal-tac set* (*normFaces* (*faces g*)) ∩ {*normFace f21*,*normFace f12*} = {})
apply (*simp add*: *f12-def f21-def ws-def normFaces-def*) **apply** *blast*
apply (*simp add*: *new-f21-norm new-f12-norm*)
apply (*subgoal-tac set* (*normFaces* (*faces g*)) ∩ {*normFace f21*} = {})
apply (*simp add*: *new-f21-norm*)
apply (*frule minGraphProps-facesAt-eq*)
apply (*subgoal-tac v* ∈ $\mathcal{V}$ *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* ∈ $\mathcal{V}$ *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-notin*)
apply (*subgoal-tac set* (*faceListAt g* ! *x*) = {*f* ∈ *F g*. *x* ∈ *V f*})
apply (*subgoal-tac set* (*normFaces* (*faces g*)) ∩ {*normFace f21*}
= {})
apply (*simp add*: *f21-def ws-def normFaces-def*) **apply** *blast*
apply (*simp add*: *new-f21-norm*)

apply (*frule minGraphProps-facesAt-eq*) **apply** *assumption*
apply (*simp add*: *facesAt-def split*: *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 (*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 (*simp add*: *new-f12*)
apply (*frule minGraphProps-facesAt-eq*)
apply (*subgoal-tac x* ∈ 𝒱 *g*) **apply** *assumption*
apply (*simp add*: *minGraphProps4 vertices-graph*)
apply (*simp add*:*facesAt-def minGraphProps4 vertices-graph*)
apply (*simp add*: *f12-def* [*symmetric*] *f21-def* [*symmetric*] *ws-def* [*symmetric*])
apply (*subgoal-tac normFace f21* ∉ *set* (*normFaces* (*replace f* [*f12*] (*faceListAt g* ! *x*))))
apply (*simp add*: *normFaces-def*)
apply (*rule ccontr*) **apply** *simp*
apply (*frule normFace-replace-in*)
apply (*subgoal-tac normFace f12* ≠ *normFace f21*)
apply (*subgoal-tac normFace f21* ∉ *set* (*normFaces* (*faceListAt g* ! *x*)))
apply (*simp add*: *normFaces-def*)
apply (*rule ccontr*) **apply** *simp*
apply (*subgoal-tac normFace f21* ∉ *set* (*normFaces* (*facesAt g x*)))
apply (*simp add*: *facesAt-def*)
apply (*subgoal-tac normFace f21* ∉ *set* (*normFaces* (*faces g*))) **apply** (*frule minGraphProps-facesAt-eq*)
apply (*subgoal-tac x* ∈ 𝒱 *g*) **apply** *assumption* **apply** (*simp add*: *minGraphProps4 vertices-graph*)
apply (*simp add*: *normFaces-def*) **apply** (*rule ccontr*) **apply** *simp*
apply *blast*
apply (*rule new-f21-norm*)
apply (*rule splitFace-f12-f21-neq-norm*) **apply** (*rule pre*) **apply** *simp* **apply** (*rule fdg*)
apply (*simp add*: *minGraphProps4 vertices-graph*)

apply (*simp add*: *normFaces-def*)

```
                apply (subgoal-tac (x - |faceListAt g | ) < n) apply simp
                  apply (rule splitFace-f12-f21-neq-norm) apply (rule pre) apply
simp
                 apply (simp add: f12-def [symmetric] f21-def [symmetric]  ws-def
[symmetric]) apply (simp add: ws-def) apply (rule fdg)
               by (simp add: minGraphProps4)
             qed
           qed
         qed
       qed
  then show ?thesis by (simp add: g'-def)
qed


lemma splitFace-holds-facesAt-eq:
assumes pre-F: pre-splitFace g' v a f' [countVertices g'..<countVertices g' + n]
and mgp: minGraphProps g'
and g'': g'' = (snd (snd (splitFace g' v a f' [countVertices g'..<countVertices g'
+ n])))
shows facesAt-eq g''
proof -
  have [0..<countVertices g''] = [0..<countVertices g' + n]
    apply (simp add: g'') by (simp add: splitFace-def split-def)
  hence vg'': vertices g'' = [0..<countVertices g' + n] by (simp add:vertices-graph)

  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' ≠ f21 apply (rule-tac splitFace-f21-oldF-neq)
apply assumption by simp+
  from pre-F' mgp fdg have f'-f12: f' ≠ f12 apply (rule-tac splitFace-f12-oldF-neq)
apply assumption by simp+

  from f12-def vs1-def have vert-f12: vertices f12 = rev ws @ v # vs1 @ [a] by
(simp add: split-face-def)
  from f21-def vs2-def have vert-f21: vertices f21 = a # vs2 @ v # ws by (simp
add: split-face-def)
  from vs1-def vs2-def pre-F have vertFrom-f': verticesFrom f' v =
    v # vs1 @ a # vs2 apply simp
    apply (rule-tac verticesFrom-ram1) by (rule pre-splitFace-pre-split-face)
  from vs1-def vs2-def pre-F vertFrom-f' have vert-f': 𝒱 f' =  set vs1 ∪ 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-graph.simps split*:*split-if-asm*)
apply (*case-tac v* < |*faceListAt g'*| $\wedge$ *a* < | *faceListAt g'*|)
apply (*simp only*: *nth-append split*: *split-if-asm*)
apply (*case-tac va* < | *faceListAt g'* |)
apply (*subgoal-tac va* $\in \mathcal{V}$ *g'*)
apply (*subgoal-tac distinct vs1* $\wedge$ *distinct vs2* $\wedge$
v $\notin$ *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 (*case-tac a* = *va*)
apply (*simp add*:*replacefacesAt-nth2 replacefacesAt-notin*)
apply (*subgoal-tac distinct* (*faceListAt g'* ! *va*))
apply (*subgoal-tac distinct* (*faces g'*))
apply (*simp add*: *replace6*)
apply (*case-tac x* = *f12*) **apply** (*simp add*: *vert-f12*) **apply** *simp*
apply (*case-tac x* = *f'*) **apply** (*simp add*: *vert-f21*) **apply**(*simp*)
apply (*case-tac x* = *f21*) **apply** (*simp add*: *vert-f21*) **apply** (*simp*)
apply (*rule conjI*)
apply (*rule minGraphProps5*) **apply** *assumption* **apply** *assumption* **apply** (*fastforce simp*: *facesAt-def*)
apply (*rule minGraphProps6*) **apply** *assumption* **apply** *assumption* **apply**

(*simp add*: *facesAt-def*)
apply (*rule minGraphProps11'*) **apply** *simp*
apply (*subgoal-tac distinct* (*facesAt g' va*)) **apply** (*simp add*: *facesAt-def*)
apply (*rule normFaces-distinct*) **apply** (*rule minGraphProps8*) **apply** *simp*
apply *simp* **apply** *simp*
apply (*case-tac v = va*)
apply (*simp add*:*replacefacesAt-nth2 replacefacesAt-notin*)
apply (*subgoal-tac distinct* (*faceListAt g' ! va*))
apply (*subgoal-tac distinct* (*faces g'*))
apply (*simp add*: *replace6*)
apply (*case-tac x = f12*) **apply** (*simp add*: *vert-f12*) **apply** *simp*
apply (*case-tac x = f'*) **apply** (*simp add*: *vert-f21*) **apply** *simp*
apply (*case-tac x = f21*) **apply** (*simp add*: *vert-f21*) **apply** *simp*
apply (*rule conjI*)
apply (*rule minGraphProps5*) **apply** *assumption* **apply** *assumption* **apply**
(*fastforce simp*: *facesAt-def*)
apply (*rule minGraphProps6*) **apply** *assumption* **apply** *assumption* **ap-**
ply(*fastforce simp*: *facesAt-def*)
apply (*rule minGraphProps11'*) **apply** *simp*
apply (*subgoal-tac distinct* (*facesAt g' va*)) **apply** (*simp add*: *facesAt-def*)
apply (*rule normFaces-distinct*) **apply** (*rule minGraphProps8*) **apply** *simp*
apply *simp* **apply** *simp*
apply (*case-tac va ∈ set vs1*)
apply (*subgoal-tac va ∉ set vs2*)
apply (*simp add*:*replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in*)
apply (*subgoal-tac distinct* (*faceListAt g' ! va*))
apply (*subgoal-tac distinct* (*faces g'*))
apply (*simp add*: *replace6*)
apply (*case-tac x = f12*) **apply** (*simp add*: *vert-f12*) **apply** *simp*
apply (*case-tac x = f'*) **apply** (*simp add*: *vert-f21*) **apply** *simp*
apply (*rule conjI*)
apply (*rule disjI2*)
apply (*rule minGraphProps5*) **apply** *assumption* **apply** *assumption*
apply (*fastforce simp*: *facesAt-def*)
apply (*rule minGraphProps6*) **apply** *assumption* **apply** *assumption* **apply**
(*fastforce simp*: *facesAt-def*)
apply (*rule minGraphProps11'*) **apply** *simp*
apply (*subgoal-tac distinct* (*facesAt g' va*)) **apply** (*simp add*: *facesAt-def*)
apply (*rule normFaces-distinct*) **apply** (*rule minGraphProps8*) **apply**
assumption **apply** *assumption*
apply *blast*
apply (*case-tac va ∈ set vs2*)
apply (*simp add*:*replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in*)
apply (*subgoal-tac distinct* (*faceListAt g' ! va*))
apply (*subgoal-tac distinct* (*faces g'*))
apply (*simp add*: *replace6*)
apply (*case-tac x = f21*) **apply** (*simp add*: *vert-f21*) **apply** *simp*
apply (*case-tac x = f'*) **apply** (*simp add*: *vert-f21*) **apply** *simp*
apply (*rule conjI*)

apply (*rule disjI2*)
apply (*rule minGraphProps5*) **apply** *assumption* **apply** *assumption* **apply** (*fastforce simp*: *facesAt-def*)
apply (*rule minGraphProps6*) **apply** *assumption* **apply** *assumption* **apply** (*fastforce simp*: *facesAt-def*)
apply (*rule minGraphProps11'*) **apply** *simp*
apply (*subgoal-tac distinct* (*facesAt g' va*)) **apply** (*simp add*: *facesAt-def*)
apply (*rule normFaces-distinct*) **apply** (*rule minGraphProps8*) **apply** *assumption* **apply** *assumption*

apply (*simp add*:*replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in*)
apply (*subgoal-tac distinct* (*faceListAt g' ! va*))
apply (*subgoal-tac distinct* (*faces g'*))
apply (*simp add*: *replace6*)
apply (*case-tac x = f'*)
apply (*subgoal-tac va* $\in \mathcal{V}$ *f'*) **apply** *simp*
apply (*rule minGraphProps6*) **apply** *simp* **apply** (*simp add*: *fla*)
apply (*simp add*: *facesAt-def*)
apply *simp*
apply (*rule conjI*)
apply (*rule disjI2*) **apply** (*rule disjI2*)
apply (*rule minGraphProps5*) **apply** *assumption* **apply** *assumption* **apply** (*fastforce simp*: *facesAt-def*)
apply (*rule minGraphProps6*) **apply** *assumption* **apply** *assumption* **apply**(*fastforce simp*: *facesAt-def*)
apply (*rule minGraphProps11'*) **apply** *assumption*
apply (*subgoal-tac distinct* (*facesAt g' va*)) **apply** (*simp add*: *facesAt-def*)
apply (*rule normFaces-distinct*) **apply** (*rule minGraphProps8*) **apply** *assumption* **apply** *assumption* **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 (*rule minGraphProps11'*) **apply** *assumption*

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 (*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-notin*)
apply (*subgoal-tac distinct* (*faceListAt* g' ! *va*))
apply (*subgoal-tac distinct* (*faces* g'))
apply (*simp add*: *replace6*)
apply (*case-tac* $x = f12$) **apply** *simp* **apply** (*rule disjI1*) **apply** (*rule minGraphProps7'*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac* $x = f21$) **apply** *simp* **apply** (*rule disjI1*) **apply** (*rule minGraphProps7'*) **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp* **apply** (*rule minGraphProps7'*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*rule minGraphProps11'*) **apply** *simp*
apply (*rule normFaces-distinct*) **apply** (*rule minGraphProps8a*) **apply** *simp* **apply** *assumption*
apply *simp*
apply (*case-tac* $v = va$)
apply (*simp add*:*replacefacesAt-nth2 replacefacesAt-notin*)
apply (*subgoal-tac distinct* (*faceListAt* g' ! *va*))
apply (*subgoal-tac distinct* (*faces* g'))
apply (*simp add*: *replace6*)
apply (*case-tac* $x = f12$) **apply** *simp* **apply** (*rule disjI1*) **apply** (*rule minGraphProps7'*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac* $x = f21$) **apply** *simp* **apply** (*rule disjI1*) **apply** (*rule minGraphProps7'*) **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp* **apply** (*rule minGraphProps7'*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*rule minGraphProps11'*) **apply** *simp* **apply** (*rule normFaces-distinct*) **apply** (*rule minGraphProps8a*) **apply** *simp* **apply** *simp* **apply** (*case-tac* $va \in$ *set vs1*)
apply (*subgoal-tac* $va \notin$ *set vs2*)

apply (*simp add*:*replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in*)
apply (*subgoal-tac distinct* (*faceListAt g′* ! *va*))
apply (*subgoal-tac distinct* (*faces g′*))
apply (*simp add*: *replace6*)
apply (*case-tac x* = *f12*) **apply** *simp* **apply** (*rule disjI1*) **apply** (*rule minGraphProps7′*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac x* = *f′*)
apply (*subgoal-tac f′* ≠ *f21*) **apply** *simp* **apply** (*rule splitFace-f21-oldF-neq*)
apply (*rule pre-F′*)
apply *simp*
apply (*rule fdg*)
apply (*case-tac x* = *f21*) **apply** (*simp add*: *vert-f21 fla*) **apply** (*thin-tac* [*countVertices g′..*<*countVertices g′* + *n*] ≡ *ws*)
apply (*simp add*: *ws-def*)
apply *simp* **apply** (*rule minGraphProps7′*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*rule minGraphProps11′*) **apply** *simp*
apply (*rule normFaces-distinct*) **apply** (*rule minGraphProps8a*) **apply** *simp* **apply** *simp*
apply *blast*
apply (*case-tac va* ∈ *set vs2*)
apply (*simp add*:*replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in*)
apply (*subgoal-tac distinct* (*faceListAt g′* ! *va*))
apply (*subgoal-tac distinct* (*faces g′*))
apply (*simp add*: *replace6*)
apply (*case-tac x* = *f21*) **apply** *simp* **apply** (*rule disjI1*) **apply** (*rule minGraphProps7′*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*case-tac x* = *f′*)
apply (*subgoal-tac f′* ≠ *f12*) **apply** *simp* **apply** (*rule splitFace-f12-oldF-neq*)
apply (*rule pre-F′*) **apply** *simp* **apply** (*rule fdg*)
apply (*case-tac x* = *f12*) **apply** (*simp add*: *vert-f12 fla*) **apply** (*thin-tac* [*countVertices g′..*<*countVertices g′* + *n*] ≡ *ws*)
apply (*simp add*: *ws-def*)
apply *simp* **apply** (*rule minGraphProps7′*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*rule minGraphProps11′*) **apply** *simp* **apply** (*rule normFaces-distinct*) **apply** (*rule minGraphProps8a*) **apply** *simp* **apply** *simp*
apply (*simp add*:*replacefacesAt-nth2 replacefacesAt-notin replacefacesAt-in*)
apply (*subgoal-tac distinct* (*faces g′*))
apply (*simp add*: *replace6*)
apply (*rule minGraphProps7′*) **apply** *simp*
apply (*case-tac x* = *f21*) **apply** (*simp add*: *vert-f21*) **apply** (*thin-tac* [*countVertices g′..*<*countVertices g′* + *n*] ≡ *ws*)
apply (*simp add*: *ws-def vertices-graph*)
apply (*case-tac x* = *f12*) **apply** (*simp add*: *vert-f12*) **apply** (*thin-tac* [*countVertices g′..*<*countVertices g′* + *n*] ≡ *ws*)
apply (*simp add*: *ws-def vertices-graph*)
apply *simp*
apply *simp*

apply (*rule minGraphProps11'*) **apply** *simp*
apply (*subgoal-tac distinct* (*vertices f12*) ∧ *distinct* (*vertices f21*))
apply (*simp add*: *vert-f12 vert-f21*)
apply (*rule vs1-vs2-empty*)
apply (*subgoal-tac pre-split-face f' v a ws*)
apply (*simp add*: *f12-def f21-def split-face-distinct1' split-face-distinct2'*)
apply *simp*
apply (*simp add*: *vertices-graph fla*)
apply (*simp add*: *nth-append del*:*replacefacesAt-simps*)
apply (*subgoal-tac distinct* (*faces g'*))
apply (*simp add*: *replace6*)
apply (*thin-tac* [*countVertices g'..*<*countVertices g'* + *n*] ≡ *ws*)
apply (*subgoal-tac* (*va* − |*faceListAt g'*|) < |*ws*|) **apply** *simp*
apply (*rule ccontr*) **apply** *simp*
apply (*case-tac x* = *f'*) **apply** *simp* **apply** *simp*
apply (*subgoal-tac va* ∈ *V g'*) **apply** (*simp add*: *fla vertices-graph*)
apply (*rule minGraphProps9*) **apply** *simp* **apply** *force* **apply** *simp*
apply (*simp add*: *ws-def fla*)
apply (*rule minGraphProps11'*) **apply** *simp*
apply (*subgoal-tac v* ∈ *V g'* ∧ *a* ∈ *V g'*)
apply (*simp only*: *fla in-vertices-graph*)
apply (*subgoal-tac f'* ∈ *F g'*)
apply (*subgoal-tac v* ∈ *V f'* ∧ *a* ∈ *V f'*) **apply** (*simp only*: *minGraphProps9*)
apply *force*
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*:
assumes *pre-F*: *pre-splitFace g' v a f'* [*countVertices g'..*<*countVertices g'* + *n*]
and *mgp*: *minGraphProps g'*
shows *faces-subset* (*snd* (*snd* (*splitFace g' v a f'* [*countVertices g'..*<*countVertices g'* + *n*])))
proof −
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*:
assumes *pre-F*: *pre-splitFace g' v a f' ws*
and *mgp*: *minGraphProps g'*
shows *edges-sym* (*snd* (*snd* (*splitFace g' v a f' ws*)))
proof −
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:
assumes pre-F: pre-splitFace g' v a f' [countVertices g'..<countVertices g' + n]
and mgp: minGraphProps g'
shows faces-distinct (snd (snd (splitFace g' v a f' [countVertices g'..<countVertices
g' + n])))
proof −
  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' ≠ f21) apply simp
       apply (rule splitFace-f21-oldF-neq)
         apply (rule pre-F') apply simp
       apply (rule fdg)
      apply simp
      apply (case-tac x = f21) apply simp
       apply (subgoal-tac normFace f12 ≠ normFace f21) apply simp
```

apply (*rule splitFace-f12-f21-neq-norm*) **apply** *force* **apply** *simp*
apply (*simp add*: *fdg*) **apply** (*rule fdg*)
apply *simp*
apply (*subgoal-tac normFace f12* $\notin$ *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* $\in$ *set vs* $\Longrightarrow$ *y* $\in \mathcal{V}$ *f* $\Longrightarrow$ *x* $\neq$ *y*)
prefer *2* **apply** *blast*
apply(*subgoal-tac* !!*x y*. *x* $\in$ *set vs* $\Longrightarrow$ *y* $\in \mathcal{V}$ *f* $\Longrightarrow$ *y* $\neq$ *x*)
prefer *2* **apply** *blast*
apply(*subgoal-tac a* $\notin$ *set vs*)
prefer *2* **apply** *blast*
apply(*subgoal-tac b* $\notin$ *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* = [] $\Longrightarrow$ *between* (*vertices f*) *a b* $\neq$ [])
prefer *2* **apply** *clarsimp* **apply**(*frule* (*4*) *is-nextElem-between-empty'*)**apply** *blast*
apply(*subgoal-tac vs* = [] $\Longrightarrow$ *between* (*vertices f*) *b a* $\neq$ [])
prefer *2* **apply** *clarsimp*

apply(*frule* (*3*) *is-nextElem-between-empty'*)**apply** *simp* **apply** *blast*
apply(*subgoal-tac vs* $\neq$ [] $\Longrightarrow$ *hd vs* $\notin$ $\mathcal{V}$ *f*)
prefer *2* **apply**(*drule hd-in-set*) **apply** *blast*
apply(*subgoal-tac vs* $\neq$ [] $\Longrightarrow$ *last vs* $\notin$ $\mathcal{V}$ *f*)
prefer *2* **apply**(*drule last-in-set*) **apply** *blast*
apply(*subgoal-tac* !!*u v*. *between* (*vertices f*) *u v* $\neq$ [] $\Longrightarrow$ *hd*(*between* (*vertices f*) *u v*) $\in$ $\mathcal{V}$ *f*)
prefer *2* **apply**(*drule hd-in-set*)**apply**(*drule inbetween-inset*) **apply** *blast*
apply(*subgoal-tac* !!*u v*. *between* (*vertices f*) *u v* $\neq$ [] $\Longrightarrow$ *last* (*between* (*vertices f*) *u v*) $\in$ $\mathcal{V}$ *f*)
prefer *2* **apply**(*drule last-in-set*) **apply**(*drule inbetween-inset*) **apply** *blast*
apply(*simp add*:*split-face-def edges-conv-Edges Edges-append Edges-Cons*
last-rev notinset-notinEdge1 notinset-notinEdge2 notinset-notinbetween
between-not-r1 between-not-r2 help Edges-rev-disj disj-sets-disj-Edges
Int-Un-distrib Int-Un-distrib2)
apply *clarify*
apply(*rule conjI*)
apply *clarify*
apply(*rule disj-sets-disj-Edges*)
apply *simp*
apply(*blast dest*:*inbetween-inset*)
apply *clarify*
apply(*rule conjI*)
apply *clarify*
apply(*rule disj-sets-disj-Edges*)
apply *simp*
apply(*blast dest*:*inbetween-inset*)
apply *clarify*
apply(*rule conjI*)
apply(*rule disj-sets-disj-Edges*)
apply *simp*
apply(*blast dest*:*inbetween-inset*)
apply(*rule disj-sets-disj-Edges*)
apply(*blast dest*:*inbetween-inset*)
done

lemma *splitFace-edge-disj*:
assumes *mgp*: *minGraphProps g* **and** *pre*: *pre-splitFace g u v f vs*
and *FDG*: (f_1,f_2,g') = *splitFace g u v f vs*
shows *edges-disj g'*
proof −
from *mgp* **have** *disj*: *edges-disj g* **by**(*simp add*:*minGraphProps-def*)
have $\mathcal{V}$ *g* $\cap$ *set vs* = {} **using** *pre*
by (*simp add*: *pre-splitFace-def*)
hence *gvs*: $\forall f \in \mathcal{F}$ *g*. $\mathcal{V}$ *f* $\cap$ *set vs* = {}
by (*clarsimp simp*:*edges-graph-def edges-face-def*)
(*blast dest*: *minGraphProps9*[*OF mgp*])
have *f*: *f* $\in$ $\mathcal{F}$ *g* **by** (*rule pre-splitFace-oldF*[*OF pre*])

note *split-face* = *splitFace-split-face*[*OF f FDG*]
note *pre-split-face* = *pre-splitFace-pre-split-face*[*OF pre*]
have $\mathcal{E}\ f_1 \cap \mathcal{E}\ f_2 = \{\}$
apply(*rule split-face-edge-disj*[*OF pre-split-face split-face mgp-vertices3*[*OF mgp f*]])
using *pre*
apply(*simp add*:*pre-splitFace-def del*: *pre-splitFace-oldF*)
apply *clarify*
by(*simp*)
moreover
{ fix f' **assume** f': $f' \in \mathcal{F}\ g\ f' \neq f$
have $(\mathcal{E}\ f_1 \cup \mathcal{E}\ f_2) \cap \mathcal{E}\ f' = \{\}$
proof *cases*
assume *vs*: *vs* = []
have $(u,v) \notin \mathcal{E}\ g \wedge (v,u) \notin \mathcal{E}\ g$ **using** *pre vs*
by(*simp add*:*pre-splitFace-def*)
with *split-face-edges-f12-f21-vs*[*OF pre-split-face*[*simplified vs*] *split-face*[*simplified vs*]]
show *?thesis* **using** *f f' disj*
by(*simp add*:*is-duplicateEdge-def edges-graph-def edges-disj-def*)
next
assume *vs*: *vs* ≠ []
have *f12*: *vs* ≠ [] ⟹ $\mathcal{E}\ f_1 \cup \mathcal{E}\ f_2 \subseteq$
$\mathcal{E}\ f \cup$ *UNIV* × *set vs* ∪ *set vs* × *UNIV*
using *split-face-edges-f12-f21*[*OF pre-split-face split-face*]
by *simp* (*fastforce dest*:*in-Edges-in-set*)
have !!*x y*. $(y,x) \in \mathcal{E}\ f' \Longrightarrow x \notin$ *set vs* $\wedge\ y \notin$ *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** f_1 = *fst*(*splitFace g u v f vs*) **and** f_2 = *fst*(*snd*(*splitFace g u v f vs*))], *auto*)

lemma *vertices-conv-Union-edges2*:
distinct(*vertices f*) ⟹ $\mathcal{V}$(*f*::*face*) = $(\bigcup (a,b) \in \mathcal{E}\ f.\ \{b\})$
apply *auto*
apply(*fast intro*: *prevVertex-in-edges*)

done

lemma *splitFace-face-face-op*:
assumes *mgp*: *minGraphProps g* **and** *pre*: *pre-splitFace g u v f vs*
and *fdg*: $(f_1,f_2,g') = splitFace\ g\ u\ v\ f\ vs$
shows *face-face-op g'*
proof −
have *f12*: $(f_1, f_2) = split\text{-}face\ f\ u\ v\ vs$
and *Fg'*: $\mathcal{F}\ g' = \{f_1\} \cup set(replace\ f\ [f_2]\ (faces\ g))$
and *g'*: $g' = snd\ (snd\ (splitFace\ g\ u\ v\ f\ vs))$ **using** *fdg*
by(*auto simp add*:*splitFace-def split-def*)
have f_1: $f_1 = fst(split\text{-}face\ f\ u\ v\ vs)$ **and** f_2: $f_2 = snd(split\text{-}face\ f\ u\ v\ vs)$
using *f12*[*symmetric*] **by** *simp-all*
note *distF* = *minGraphProps11'*[*OF mgp*]
note *pre-split* = *pre-splitFace-pre-split-face*[*OF pre*]
note $distf_1$ = *split-face-distinct1*[*OF f12 pre-split*]
note $distf_2$ = *split-face-distinct2*[*OF f12 pre-split*]
from *pre* **have** *nf*: ¬ *final f* **and** *fg*: $f \in \mathcal{F}\ g$ **and** *nuv*: $u \neq v$
and *uinf*: $u \in \mathcal{V}\ f$**and** *vinf*: $v \in \mathcal{V}\ f$
and *distf*: *distinct*(*vertices f*) **and** *new*: $\mathcal{V}\ g \cap set\ vs = \{\}$
by(*unfold pre-splitFace-def*, *simp*)+
let *?fuv* = *between* (*vertices f*) *u v* **and** *?fvu* = *between* (*vertices f*) *v u*
have E_1: $\mathcal{E}\ f_1 = Edges\ (v\ \#\ rev\ vs\ @\ [u]) \cup Edges\ (u\ \#\ ?fuv\ @\ [v])$
using f_1 **by**(*simp add*:*edges-split-face1*[*OF pre-split*])
have E_2: $\mathcal{E}\ f_2 = Edges\ (u\ \#\ vs\ @\ [v]) \cup Edges\ (v\ \#\ ?fvu\ @\ [u])$
using f_2 **by**(*simp add*:*edges-split-face2*[*OF pre-split*])
have vf_1: $vertices\ f_1 = rev\ vs\ @\ u\ \#\ ?fuv\ @\ [v]$
using f_1 **by**(*simp add*:*split-face-def*)
have vf_2: $vertices\ f_2 = [v]\ @\ ?fvu\ @\ u\ \#\ vs$
using f_2 **by**(*simp add*:*split-face-def*)
have V_1: $\mathcal{V}\ f_1 = \{u,v\} \cup set(?fuv) \cup set(vs)$ **using** vf_1 **by** *auto*
have V_2: $\mathcal{V}\ f_2 = \{u,v\} \cup set(?fvu) \cup set(vs)$ **using** vf_2 **by** *auto*
have *2*: $(v,u) \in \mathcal{E}\ f_1 \wedge (u,v) \in \mathcal{E}\ f_2 \wedge vs = [] \vee$
$(\exists\, v \in \mathcal{V}\ f_1 \cap \mathcal{V}\ f_2.\ v \notin \mathcal{V}\ g)$
using E_1 E_2 V_1 V_2 *new* **by**(*cases vs*)(*simp-all add*:*Edges-Cons*)
have $\mathcal{V}\ f_1 \neq \mathcal{V}\ f_2$
proof *cases*
assume *A*: *?fvu* = []
have *?fuv* ≠ []
proof
assume *?fuv* = []
with *A* **have** $\mathcal{E}\ f = \{(v,u),(u,v)\}$
using *edges-conv-Un-Edges*[*OF distf uinf vinf nuv*]
by(*simp add*:*Edges-Cons*)
hence $\mathcal{V}\ f = \{u,v\}$ **by**(*simp add*:*vertices-conv-Union-edges*)
hence $card(\mathcal{V}\ f) \leq 2$ **by**(*simp add*:*card-insert-if*)
thus *False*
using *mgp-vertices3*[*OF mgp fg*] **by**(*simp add*:*distinct-card*[*OF distf*])
qed

moreover have *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*: $\mathcal{E}\ f_1 \neq (\mathcal{E}\ f_2)^{-1}$
proof
assume *A*: $\mathcal{E}\ f_1 = (\mathcal{E}\ f_2)^{-1}$
show *False*
proof −
have $\mathcal{V}\ f_1 = (\bigcup (a,b) \in \mathcal{E}\ f_1.\ \{a\})$
by(*rule vertices-conv-Union-edges*)
also have ... = $(\bigcup (b,a) \in \mathcal{E}\ f_2.\ \{a\})$ **by**(*auto simp*:*A*)
also have ... = $\mathcal{V}\ f_2$
by(*rule vertices-conv-Union-edges2*[*OF distf*$_2$, *symmetric*])
finally show *False* **using** ‹$\mathcal{V}\ f_1 \neq \mathcal{V}\ f_2$› **by** *blast*
qed
qed
{ fix *h* :: *face* **assume** *hg*: *h* ∈ $\mathcal{F}$ *g*
have $\mathcal{E}\ h \neq (\mathcal{E}\ f_1)^{-1} \wedge \mathcal{E}\ h \neq (\mathcal{E}\ f_2)^{-1}$ **using** *2*
proof
assume (*v*,*u*) ∈ $\mathcal{E}\ f_1 \wedge$ (*u*,*v*) ∈ $\mathcal{E}\ f_2 \wedge$ *vs* = []
moreover hence (*u*,*v*) ∉ $\mathcal{E}$ *g*
using *pre* **by**(*unfold pre-splitFace-def*)*simp*
moreover hence (*v*,*u*) ∉ $\mathcal{E}$ *g* **by**(*blast intro*:*minGraphProps10*[*OF mgp*])
ultimately show *?thesis* **using** *hg* **by**(*simp add*:*edges-graph-def*) *blast*
next
assume ∃ *x* ∈ $\mathcal{V}\ f_1 \cap \mathcal{V}\ f_2$. *x* ∉ $\mathcal{V}$ *g*
then obtain *x* **where** *x* ∈ $\mathcal{V}\ f_1$ **and** *x* ∈ $\mathcal{V}\ f_2$ **and** *x* ∉ $\mathcal{V}$ *g*
by *blast*
obtain *y* **where** (*x*,*y*) ∈ $\mathcal{E}\ f_1$ **using** ‹*x* ∈ $\mathcal{V}\ f_1$›
by(*auto simp*:*vertices-conv-Union-edges*)
moreover obtain *z* **where** (*x*,*z*) ∈ $\mathcal{E}\ f_2$ **using** ‹*x* ∈ $\mathcal{V}\ f_2$›
by(*auto simp*:*vertices-conv-Union-edges*)
moreover have ¬(*EX y*. (*y*,*x*) ∈ $\mathcal{E}$ *h*)
using ‹*x* ∉ $\mathcal{V}$ *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(fastforce simp: eval-nat-numeral 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(fastforce 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(fastforce 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:
  assumes precond: pre-splitFace g' v a f' [countVertices g'..<countVertices g' +
n]
  and min: minGraphProps g'
 shows minGraphProps (snd (snd (splitFace g' v a f' [countVertices g'..<countVertices
g' + n])))
proof −
  from min have minGraphProps' g' by (simp add: minGraphProps-def)
  then show ?thesis apply (simp add: minGraphProps-def) apply safe
      apply (rule splitFace-holds-minGraphProps') apply (rule precond) apply
assumption
    apply (rule splitFace-holds-facesAt-eq) apply (rule precond) apply (rule min)
apply simp
     apply (rule splitFace-holds-faceListAt-len) apply (rule precond) apply (rule
min)
    apply (rule splitFace-holds-facesAt-distinct) apply (rule precond) apply (rule
min)
     apply (rule splitFace-holds-faces-distinct) apply (rule precond) apply (rule
min)
   apply (rule splitFace-holds-faces-subset) apply (rule precond) apply (rule min)
```

apply (*rule splitFace-holds-edges-sym*) **apply** (*rule precond*) **apply** (*rule min*)
apply (*rule splitFace-edges-disj2*) **apply** (*rule min*) **apply** (*rule precond*)
apply (*rule splitFace-face-face-op2*) **apply** (*rule min*) **apply** (*rule precond*)
done
qed

14.7 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* (*1*) *minGraphProps6*) **apply** (*simp add*: *facesAt-def*)
apply *blast*
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* (*1*) *minGraphProps6*) **apply** (*simp add*: *facesAt-def*)
apply *simp*
apply (*rule conjI*) **apply** (*rule disjI2*)
apply (*erule* (*1*) *minGraphProps5*) **apply** (*fastforce simp*: *facesAt-def*)
apply (*erule* (*1*) *minGraphProps6*) **apply** (*fastforce simp*: *facesAt-def*)
apply (*rule minGraphProps11'*) **apply** *simp*
apply (*rule normFaces-distinct*) **apply** (*rule minGraphProps8a*) **apply** *simp*
apply *simp*
apply (*simp add*: *vertices-graph minGraphProps4*)

apply (*rule subsetI*) **apply** (*simp add*: *makeFaceFinal-def facesAt-def*)
apply (*subgoal-tac* $v <|$ *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 ∈ ℱ *g* ⟹ *minGraphProps g* ⟹ *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 ∈ ℱ *g* ⟹ *minGraphProps g* ⟹ *facesAt-distinct* (*makeFaceFinal f g*)
apply (*simp add*: *facesAt-distinct-def makeFaceFinal-def*)
apply (*clarsimp simp*: *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 ∈ ℱ *g* ⟹ *minGraphProps g* ⟹ *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* $x \in \mathcal{F}\ g$)
apply (*subgoal-tac* $(a,b) \in edges\ g$) **apply** (*frule minGraphProps10*) **apply** *assumption*
apply (*simp add*: *edges-graph-def*) **apply** (*elim bexE*)
apply (*case-tac* $xb = f$)
apply (*subgoal-tac* $(b,a) \in edges\ (setFinal\ f)$)
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* $(a,b) \in edges\ g$)
apply (*frule minGraphProps10*) **apply** *assumption* **apply** (*simp add*: *edges-graph-def*)
apply (*elim bexE*)
apply (*case-tac* $xb = f$)
apply (*subgoal-tac* $(b,\ a) \in edges\ (setFinal\ f)$)
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* $(a,b) \in edges\ f$)
apply (*simp add*: *edges-graph-def*) **apply** *force*
apply (*simp add*: *edges-setFinal*)
apply (*rule minGraphProps3*) **apply** *simp* **apply** *simp*
by (*rule minGraphProps11'*)

lemma *MakeFaceFinal-faces-distinct*:
$f \in \mathcal{F}\ g \Longrightarrow\ minGraphProps\ g \Longrightarrow faces\text{-}distinct\ (makeFaceFinal\ f\ g)$
apply (*simp add*: *faces-distinct-def makeFaceFinal-def normFaces-makeFaceFinalFaceList*)
by (*rule minGraphProps11*)

lemma *MakeFaceFinal-edges-disj*:
$f \in \mathcal{F}\ g \Longrightarrow\ minGraphProps\ g \Longrightarrow edges\text{-}disj\ (makeFaceFinal\ f\ g)$
apply(*frule minGraphProps11'*)
apply (*clarsimp simp*: *edges-disj-def makeFaceFinal-def edges-graph-def*
makeFaceFinalFaceList-def replace6)
apply(*case-tac* $f = f'$)
apply (*fastforce dest*:*mgp-edges-disj*)
apply (*fastforce dest*:*mgp-edges-disj*)
done

lemma *MakeFaceFinal-face-face-op*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps* $g \Longrightarrow$ *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*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps* $g \Longrightarrow$ *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

14.8 Invariants of *subdivFace'*

lemma *subdivFace'-holds-minGraphProps*: $\bigwedge f\ v'\ v\ n\ g$.
pre-subdivFace' $g\ f\ v'\ v\ n\ ovl \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow$
minGraphProps $g \Longrightarrow$ *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

abbreviation (*input*)
  Edges-if :: *face* ⇒ *vertex* ⇒ *vertex* ⇒ (*vertex* × *vertex*)*set* **where**
  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*: (*f*$_1$,*f*$_2$,*g′*) = *splitFace g u v f vs*
and *nrv*: *r* ≠ *v*
and *ruv*: *before* (*verticesFrom f r*) *u v* **and** *rf*: *r* ∈ $\mathcal{V}$ *f*
and *1*: *one-final-but g* (*Edges-if f r u*)
shows *one-final-but g′* (*Edges*(*r* # *between* (*vertices f*$_2$) *r v* @ [*v*]))
proof −
  have *f*$_1$: *f*$_1$= *fst*(*split-face f u v vs*) **and** *f*$_2$: *f*$_2$ = *snd*(*split-face f u v vs*)
   and *F*: $\mathcal{F}$ *g′* = {*f*$_1$} ∪ *set*(*replace f* [*f*$_2$] (*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* ∈ $\mathcal{F}$ *g* **and** *nuv*: *u* ≠ *v*
    and *uinf*: *u* ∈ $\mathcal{V}$ *f***and** *vinf*: *v* ∈ $\mathcal{V}$ *f*
    by(*unfold pre-splitFace-def*, *simp*)+
  from *mgp fg* **have** *distf*: *distinct*(*vertices f*) **by**(*rule minGraphProps3*)
  note *distFg* = *minGraphProps11′*[*OF mgp*]
  have *fvu*: *r*≠*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** *?f*$_2$*rv* = *between* (*vertices f*$_2$) *r v*
  have *E*$_1$: $\mathcal{E}$ *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*$_2$: $\mathcal{E}$ *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* ∈ $\mathcal{V}$ *f*$_2$ **using** *vf*$_2$ **by**(*simp*)
  have *rinf*$_2$: *r* ∈ $\mathcal{V}$ *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: $r{\neq}u \Longrightarrow$ *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*: $r \neq u$
have *vinf*$_2$: $v \in \mathcal{V}$ f_2 **by**(*simp add*: vf_2)
note *u-bet-rv* = *before-between*[*OF ruv distf rf nru*]
have *u-bet-rv*$_2$: $u \in$ *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*$_2$*rv* @ [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) ∈ $\mathcal{E}$ f_1
and *nab*: (a,b) ∉ *Edges* (r # *?f*$_2$*rv* @ [v]) **by** *fact+*
have (a,b) ∈ *Edges* (v # *rev vs* @ [u]) ∨
(a,b) ∈ *Edges* (u # *?fuv* @ [v]) (**is** *?A* ∨ *?B*)
using E_1 *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 # *?f*$_2$*rv* @ [v]) **using** E_2rv **by** *simp*
thus *?case* **by** *blast*
next
assume *abfuv*: *?B*
have *abf*: (a,b) ∈ $\mathcal{E}$ 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]) by fact+
  have f' = f₂ ∨ f' ∈ ℱ g ∧ f' ≠ f
   using f' by(simp add:replace6[OF distFg]) blast
  hence (b, a) ∈ Edges (r # between (vertices f₂) r v @ [v]) ∨
   (∃f'∈set (replace f [f₂] (faces g)). final f' ∧ (b, a) ∈ ℰ f')
   (is ?A ∨ ?B)
  proof
   assume [simp]: f' = f₂
   have (a,b) ∈ Edges (v # between (vertices f₂) v r @ [r])
    using abf' nab Edges-compl[OF distf₂ vinf₂ rinf₂ nrv[symmetric]]
    edges-conv-Un-Edges[OF distf₂ rinf₂ vinf₂ nrv] by auto
   moreover have eq: between(vertices f₂) v r = between (vertices f) v r
   proof (cases r=u)
    assume r=u thus ?thesis
```

```
    by(simp add:vf_2)(rule between-front[OF between-not-r2[OF distf]])
  next
    assume r≠u thus ?thesis
    by(simp add:vf_2 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 [f_2] (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 [f_2] (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 [f_2] (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(fastforce 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(fastforce simp add:Edges-Cons Edges-append)
        done
      have pre': pre-subdivFace' g f u w 0 ovs
     using pre Some n using [[simp-depth-limit = 5]] by (simp add: pre-subdivFace'-Some2)
      have one-final (subdivFace' g f w 0 ovs)
        by (simp add: IH[OF pre' mgp fg 2])
```

```
    } moreover
    { let ?vs = [countVertices g..<countVertices g + n]
      let ?fdg = splitFace g v w f ?vs
      let ?Ew = if u=w then {} else Edges(u # between(vertices (fst(snd ?fdg)))
u w @ [w])
      assume a: f · v = w ⟶ 0 < n
      have pre2: pre-subdivFace' g f u v n (Some w # ovs)
        using pre Some by simp
      have fsubg: 𝒱 f ⊆ 𝒱 g
        using mgp fg by(simp add: minGraphProps-def faces-subset-def)
      have pre-fdg: pre-splitFace g v w f ?vs
          apply (rule pre-subdivFace'-preFaceDiv[OF - fg - fsubg])
           using Some pre apply simp
          using a apply (simp)
          done
      have bet: before (verticesFrom f u) v w using pre Some
        by(unfold pre-subdivFace'-def) simp
      have 2: one-final-but(snd(snd ?fdg)) ?Ew
        using uw apply simp
        apply(rule FaceDivsionGraph-one-final-but[OF mgp pre-fdg - uw bet uf 1])
        apply(fastforce intro!:prod-eqI)
        done
      note mgp' = splitFace-holds-minGraphProps[OF pre-fdg mgp]
      have pre2': pre-subdivFace' (snd (snd ?fdg)) (fst (snd ?fdg)) u w 0 ovs
        by (rule pre-subdivFace'-Some1[OF pre2 fg - fsubg HOL.refl HOL.refl])
           (simp add:a)
      note f2inF = splitFace-add-f21'[OF fg]
      have one-final (subdivFace' (snd (snd ?fdg)) (fst (snd ?fdg)) w 0 ovs)
        by (simp add: IH[OF pre2' mgp' f2inF 2])
    } ultimately show ?thesis using Some by (simp add: split-def)
  qed
qed


lemma neighbors-edges:
 minGraphProps g ⟹ a : 𝒱 g ⟹ b ∈ set (neighbors g a) = ((a, b) ∈ edges g)
apply (rule iffI)
 apply (simp add: neighbors-def) apply clarify apply (frule (1) minGraphProps5)
  apply (simp add: vertices-graph)
 apply (simp add: edges-graph-def) apply (intro bexI)
  prefer 2 apply assumption
 apply(simp add:edges-face-eq)
 apply (erule (2) minGraphProps6)
apply (simp add: neighbors-def) apply (simp add: edges-graph-def) apply (elim
bexE)
apply (subgoal-tac x ∈ set (facesAt g a)) apply (simp add: edges-face-def)
apply (rule minGraphProps7) apply simp+ apply (simp add: edges-face-def)
done
```

lemma *no-self-edges*: *minGraphProps'* $g \Longrightarrow (a, a) \notin$ *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 $g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow a \in \mathcal{V}\ f \Longrightarrow b \in \mathcal{V}\ f \Longrightarrow$
duplicateEdge g f a b = *is-duplicateEdge g f a b*
apply (*subgoal-tac distinct* (*vertices f*))
prefer *2* **apply** (*simp add*: *minGraphProps3*)
apply(*subgoal-tac a* : $\mathcal{V}$ *g*)
prefer *2* **apply** (*simp add*: *minGraphProps9*)
apply (*simp add*: *duplicateEdge-def is-duplicateEdge-def neighbors-edges*)
apply (*rule iffI*)
apply (*simp add*: *minGraphProps10*)
apply (*cases a* = *b*) **apply** (*simp add*: *no-self-edges minGraphProps-def*)
apply (*rule ccontr*)
apply (*simp add*: *directedLength-def*)
apply (*case-tac is-nextElem* (*vertices f*) *a b*)
apply (*simp add*: *is-nextElem-between-empty*)
apply (*simp add*: *is-nextElem-between-empty*)
apply (*cases a* = *b*) **apply** (*simp add*: *no-self-edges minGraphProps-def*)
apply (*rule ccontr*)
apply (*simp add*: *directedLength-def*)
apply (*elim impE*)
apply (*cases between* (*vertices f*) *b a*)
apply (*simp add*: *is-nextElem-between-empty' del*:*is-nextElem-between-empty*)
apply *simp*
apply (*cases between* (*vertices f*) *a b*)
apply (*simp add*: *is-nextElem-between-empty' del*:*is-nextElem-between-empty*)
apply *simp*
apply (*simp add*: *minGraphProps10*)
done

lemma *incrIndexList-less-eq*:
incrIndexList ls m nmax $\Longrightarrow$ *Suc n* $<$ $|ls| \Longrightarrow$ *ls*!*n* $\leq$ *ls*!*Suc n*
apply (*subgoal-tac increasing ls*) **apply** (*thin-tac incrIndexList ls m nmax*) **apply** (*rule increasing1*) **apply** *simp*
apply (*subgoal-tac ls* = *take n ls* @ *ls*!*n* # [] @ *ls*!(*Suc n*) # *drop* (*Suc* (*Suc n*)) *ls*) **apply** *assumption*
apply *simp* **apply** (*subgoal-tac n* $<$ $|$ *ls*|) **apply** (*rotate-tac* -1) **apply** (*drule id-take-nth-drop*)
apply (*subgoal-tac drop* (*Suc n*) *ls* = *ls* ! *Suc n* # *drop* (*Suc* (*Suc n*)) *ls*) **apply**

simp **apply** (*drule drop-Suc-conv-tl*) **by** *auto*

lemma *incrIndexList-less*:
incrIndexList ls m nmax $\Longrightarrow$ *Suc n* $<$ $|ls|$ $\Longrightarrow$ *ls*!*n* $\neq$ *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..<Suc z*] *Final*) = *0*
apply (*induct z*) **apply** (*simp add*: *minVertex-def*)
by (*simp add*: *minVertex-def upt-conv-Cons del*: *upt-Suc*)

lemma *minVertex-zero2*: *minVertex* (*Face* (*rev* [*0..<Suc z*]) *Nonfinal*) = *0*
apply (*induct z*) **apply** (*simp add*: *minVertex-def*)
by (*simp add*: *minVertex-def min-def*)

14.9 Invariants of *Seed*

lemma *Seed-holds-facesAt-distinct*: *facesAt-distinct* (*Seed p*)
apply(*simp add*: *Seed-def graph-def*
facesAt-distinct-def normFaces-def facesAt-def normFace-def)
apply(*simp add*: *eval-nat-numeral minVertex-zero1 minVertex-zero2 verticesFrom-Def*
fst-splitAt-upt snd-splitAt-upt fst-splitAt-rev snd-splitAt-rev del:*upt-Suc*)
apply(*simp add*:*upt-conv-Cons del*:*upt-Suc*)
apply *simp*
done

lemma *Seed-holds-faces-subset*: *faces-subset* (*Seed p*)
by (*simp add*: *Seed-def graph-def faces-subset-def*)

lemma *Seed-holds-edges-sym*: *edges-sym* (*Seed p*)
by (*simp add*: *Seed-def graph-def edges-sym-def edges-graph-def*)

lemma *Seed-holds-edges-disj*: *edges-disj* (*Seed p*)
using *is-nextElem-circ*[*of* [*0..<*(*p*+*3*)]]
by (*fastforce simp add*: *Seed-def graph-def edges-disj-def edges-graph-def*)

lemma *Seed-holds-faces-distinct*: *faces-distinct* (*Seed p*)
apply(*simp add*: *Seed-def graph-def*
faces-distinct-def normFaces-def facesAt-def normFace-def)
apply(*simp add*: *eval-nat-numeral minVertex-zero1 minVertex-zero2 verticesFrom-Def*
fst-splitAt-upt snd-splitAt-upt fst-splitAt-rev snd-splitAt-rev del:*upt-Suc*)
apply(*simp add*:*upt-conv-Cons del*:*upt-Suc*)

apply *simp*
done

lemma *Seed-holds-faceListAt-len*: *faceListAt-len* (*Seed p*)
by (*simp add*: *Seed-def graph-def faceListAt-len-def*)

lemma *face-face-op-Seed*: *face-face-op*(*Seed p*)
by (*simp add*: *Seed-def graph-def face-face-op-def*)

lemma *one-final-Seed*: *one-final* $Seed_p$
by(*clarsimp simp*:*Seed-def one-final-def one-final-but-def graph-def*)

lemma *two-face-Seed*: $|faces\ Seed_p| \geq 2$
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*: $f \in$ *set* (*nonFinals g*)
and *v*: $v \in \mathcal{V}\ f$ **and** *e*: $e \in$ *set* (*enumerator i* $|vertices\ f|$)
and *containsNot*: $\neg$ *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*: $f \in \mathcal{F}\ g \neg final\ f$ **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* $f^{e\ !\ 0}$ $\cdot$ $v \in \mathcal{V}$ *f*)
apply (*subgoal-tac* $f^{e\ !\ Suc\ 0}$ $\cdot$ $v \in \mathcal{V}$ *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* ($f^{e\ !\ Suc\ nat}$ $\cdot$ *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* ($f^{e\ !\ Suc\ (Suc\ nat)}$ $\cdot$ *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* $f^{e\ !\ Suc\ nat}$ $\cdot$ $v \in \mathcal{V}$ *f*)
apply (*subgoal-tac* $f^{e\ !\ Suc\ (Suc\ nat)}$ $\cdot$ $v \in \mathcal{V}$ *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

14.10 Increasing properties of *subdivFace′*

lemma *subdivFace′-incr*:
assumes *Ptrans*: !!*x y z*. *Q x y* $\Longrightarrow$ *P y z* $\Longrightarrow$ *P x z*
and *mkFin*: !!*f g*. *f* $\in \mathcal{F}$ *g* $\Longrightarrow$ $\neg$ *final f* $\Longrightarrow$ *P g* (*makeFaceFinal f g*)
and *fdg-incr*: !! *g u v f vs*.
pre-splitFace g u v f vs $\Longrightarrow$
Q g (*snd*(*snd*(*splitFace g u v f vs*)))
shows
$\bigwedge$*f′ v n g*. *pre-subdivFace′ g f′ v′ v n ovl* $\Longrightarrow$
minGraphProps g $\Longrightarrow$ *f′* $\in \mathcal{F}$ *g* $\Longrightarrow$ *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(*fastforce simp add*:*neq-Nil-conv elim*:*pre-subdivFace-pre-subdivFace'*)
from *P*[*OF mgp this fg*] *g'* **show** *?thesis* **by** *simp*
qed

lemma *next-plane0-incr*:
assumes *Ptrans*: !!*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-plane0*$_p$]→ *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

14.10.1 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-plane0*$_p$]→ *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*

14.11 Main invariant theorems

lemma *inv-genPoly*:
assumes *inv*: *inv g* **and** *polygen*: $g' \in set(generatePolygon\ i\ v\ f\ g)$
and *f*: $f \in set\ (nonFinals\ g)$ **and** *i*: $2 < i$ **and** *v*: $v \in \mathcal{V}\ f$
shows *inv g′*
proof(*unfold inv-def*)
have *mgp*: *minGraphProps g* **and** *1*: *one-final g*
using *inv* **by**(*simp add*:*inv-def*)+
from *polygen*
obtain *is e* **where** *g′*: $g' = subdivFace\ g\ f\ is$
and *e*: $e \in set\ (enumerator\ i\ |vertices\ f|\ )$
and *containsNot*: $\neg\ containsDuplicateEdge\ g\ f\ v\ e$
and *is-eq*: $is = indexToVertexList\ f\ v\ e$
by (*auto simp*: *generatePolygon-def*)
have *f′*: $f \in \mathcal{F}\ g$ **using** *f* **by**(*simp add*:*nonFinals-def*)
note *pre-add* = *pre-subdivFace-indexToVertexList*[*OF mgp f v e containsNot i*]
with *g′ is-eq* **have** *g′*: $g' = subdivFace'\ g\ f\ v\ 0\ (tl\ is)$
by (*simp add*: *subdivFace-subdivFace′-eq*)
from *pre-add is-eq* **have** *i-nz*: $is \neq []$
by (*simp add*: *pre-subdivFace-def pre-subdivFace-face-def*)
with *pre-add v i-nz is-eq*
have *pre-addSnd*: *pre-subdivFace′ g f v v 0* (*tl is*)
by(*fastforce simp add*:*neq-Nil-conv elim*:*pre-subdivFace-pre-subdivFace′*)
note *2* = *one-final-antimono*[*OF 1*]
show *minGraphProps g′* ∧ *one-final g′* ∧ $|faces\ g'| \geq 2$
proof *auto*
show *minGraphProps g′* **using** *g′ pre-addSnd f*
apply (*simp add*:*nonFinals-def*)
apply (*rule subdivFace′-holds-minGraphProps*[*OF - - mgp*])
by (*simp-all add*: *succs*)
next
show *one-final g′* **using** *g′ 1*
by (*simp add*: *one-final-subdivFace′*[*OF pre-addSnd mgp f′ 2*])
next
show $|faces\ g'| \geq 2$ **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** $g' \in set\ (next\text{-}plane0_p\ g)$
then obtain *i v f* **where** $g' \in set(generatePolygon\ i\ v\ f\ g)$

and $f \in set\ (nonFinals\ g)$ **and** $2 < i$ **and** $v \in \mathcal{V}\ f$
by (*auto simp*: *next-plane0-def split*: *split-if-asm*)
thus *inv* g' **using** *inv* **by**(*blast intro*:*inv-genPoly*)
qed

end

15 Further Plane Graph Properties

theory *PlaneProps*
imports *Invariants*
begin

15.1 *final*

lemma *plane-final-facesAt*:
assumes *inv g final g v* : $\mathcal{V}\ g\ f \in set\ (facesAt\ g\ v)$ **shows** *final f*
proof −
from *assms*(*1*,*3*,*4*) **have** $f \in \mathcal{F}\ g$ **by**(*blast intro*: *minGraphProps inv-mgp*)
with *assms*(*2*) **show** *?thesis* **by** (*rule finalGraph-face*)
qed

lemma *finalVertexI*:
$[\![\ inv\ g;\ \ final\ g;\ \ v \in \mathcal{V}\ g\ ]\!] \Longrightarrow finalVertex\ g\ v$
by (*auto simp add*: *finalVertex-def nonFinals-def filter-empty-conv plane-final-facesAt*)

lemma *setFinal-notin-finals*:
$[\![\ f \in \mathcal{F}\ g; \neg\ final\ f;\ minGraphProps\ g\ ]\!] \Longrightarrow setFinal\ f \notin set\ (finals\ g)$
apply(*drule minGraphProps11*)
apply(*cases f*)
apply(*fastforce simp*:*finals-def setFinal-def normFaces-def normFace-def*
verticesFrom-def minVertex-def inj-on-def distinct-map
split:*facetype.splits*)
done

15.2 *degree*

lemma *planeN4*: $inv\ g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow 3 \leq |vertices\ f|$
apply(*subgoal-tac* $2 < |\ vertices\ f\ |$)
apply *arith*
apply(*drule inv-mgp*)
apply (*erule* (*1*) *minGraphProps2*)
done

lemma *degree-eq*:

assumes *pl*: *inv g* **and** *fin*: *final g* **and** *v*: $v : \mathcal{V}\ g$
shows *degree g v* = *tri g v* + *quad g v* + *except g v*
proof −
have *dist*: *distinct*(*facesAt g v*) **using** *pl v* **by** *simp*
have *3*: $\forall f \in set(facesAt\ g\ v).\ |vertices\ f| = 3 \vee |vertices\ f| = 4 \vee$
$|vertices\ f| \geq 5$
proof
fix *f* **assume** *f*: $f \in set\ (facesAt\ g\ v)$
hence $|vertices\ f| \geq 3$
using *minGraphProps5*[*OF inv-mgp*[*OF pl*] *v f*] *planeN4*[*OF pl*] **by** *blast*
thus $|vertices\ f| = 3 \vee |vertices\ f| = 4 \vee |vertices\ f| \geq 5$ **by** *arith*
qed
have *degree g v* = |*facesAt g v*| **by**(*simp add*:*degree-def*)
also have ... = *card*(*set*(*facesAt g v*)) **by** (*simp add*:*distinct-card*[*OF dist*])
also have *set*(*facesAt g v*) = $\{f \in set(facesAt\ g\ v).\ |vertices\ f| = 3\} \cup$
$\{f \in set(facesAt\ g\ v).\ |vertices\ f| = 4\} \cup$
$\{f \in set(facesAt\ g\ v).\ |vertices\ f| \geq 5\}$
(**is** - = *?T* ∪ *?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 *fastforce*
apply (*subst card-Un-disjoint*)
apply *simp*
apply *simp*
apply *fastforce*
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 fin v*] *cong*:*conj-cong*)
finally show *?thesis* **.**
qed

lemma *plane-fin-exceptionalVertex-def*:
assumes *pl*: *inv g* **and** *fin*: *final g* **and** *v*: $v : \mathcal{V}\ g$
shows *exceptionalVertex g v* =
(| [*f* ← *facesAt g v* . 5 ≤ |*vertices f*|] | ≠ 0)
proof −
have $\bigwedge f.\ f \in set\ (facesAt\ g\ v) \Longrightarrow final\ f$
by(*rule plane-final-facesAt*[*OF pl fin v*])
then show *?thesis* **by** (*simp add*: *filter-simp exceptionalVertex-def except-def*)
qed

lemma *not-exceptional*:
inv g ⟹ *final g* ⟹ $v : \mathcal{V}\ g$ ⟹ $f \in set\ (facesAt\ g\ v)$ ⟹

$\neg$ *exceptionalVertex g v* $\Longrightarrow$ $|$*vertices f*$|$ $\leq$ *4*
by (*auto simp add*: *plane-fin-exceptionalVertex-def except-def filter-empty-conv*)

15.3 Misc

lemma *in-next-plane0I*:
assumes *g'* $\in$ *set* (*generatePolygon n v f g*) *f* $\in$ *set* (*nonFinals g*)
v $\in$ $\mathcal{V}$ *f 3* $\leq$ *n n* $<$ *4*+*p*
shows *g'* $\in$ *set* (*next-plane0*$_p$ *g*)
proof $-$
have $\neg$ *final g* **using** *assms*(*2*)
by(*auto simp*: *nonFinals-def finalGraph-def filter-empty-conv*)
thus *?thesis* **using** *assms* **by**(*auto simp*:*next-plane0-def Bex-def*)
qed

lemma *next-plane0-nonfinals*: *g* [*next-plane0*$_p$]$\rightarrow$ *g'* $\Longrightarrow$ *nonFinals g* $\neq$ []
by(*auto simp*:*next-plane0-def finalGraph-def*)

lemma *next-plane0-ex*:
assumes *a*: *g* [*next-plane0*$_p$]$\rightarrow$ *g'*
shows $\exists f \in$ *set*(*nonFinals g*). $\exists v \in \mathcal{V}$ *f*. $\exists i \in$ *set*([*3*..<*Suc*(*maxGon p*)]).
g' $\in$ *set* (*generatePolygon i v f g*)
proof $-$
from *a* **have** $\neg$ *final g* **by** (*auto simp add*: *next-plane0-def*)
with *a* **show** *?thesis*
by (*auto simp add*: *next-plane0-def nonFinals-def*)
qed

lemma *step-outside2*:
inv g $\Longrightarrow$ *g* [*next-plane0*$_p$]$\rightarrow$ *g'* $\Longrightarrow$ $\neg$ *final g'* $\Longrightarrow$ $|$*faces g'*$|$ $\neq$ *2*
apply(*frule inv-two-faces*)
apply(*frule inv-finals-nonempty*)
apply(*drule inv-mgp*)
apply(*insert len-faces-sum*[*of g*] *len-faces-sum*[*of g'*])
apply(*subgoal-tac* $|$*nonFinals g*$|$ $\neq$ *0*)
prefer *2* **apply**(*drule next-plane0-nonfinals*) **apply** *simp*
apply(*subgoal-tac* $|$*nonFinals g'*$|$ $\neq$ *0*)
prefer *2* **apply**(*simp add*:*finalGraph-def*)
apply(*drule* (*1*) *next-plane0-incr-faces*)
apply(*case-tac* $|$*faces g*$|$ = *2*)
prefer *2* **apply** *arith*
apply(*subgoal-tac* $|$*finals g*$|$ $\neq$ *0*)
apply *arith*
apply *simp*
done

15.4 Increasing final faces

lemma *set-finals-splitFace*[*simp*]:
⟦ $f \in \mathcal{F}\ g$; ¬ *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' ⟹ $f \in$ *set*(*finals* g) ⟹ $f \in$ *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*:
$g' \in$ *set* (*next-plane0*$_p$ g) ⟹
set (*finals* g) ⊆ *set* (*finals* g')
by (*auto simp add*: *next-plane0-finals-incr*)

lemma *next-plane0-final-mono*:
⟦ $g' \in$ *set* (*next-plane0*$_p$ g); $f \in \mathcal{F}\ g$; *final* f ⟧ ⟹ $f \in \mathcal{F}\ g'$
apply(*drule next-plane0-finals-subset*)
apply(*simp add*:*finals-def*)
apply *blast*
done

15.5 Increasing vertices

lemma *next-plane0-vertices-subset*:
⟦ $g' \in$ *set* (*next-plane0*$_p$ g); *minGraphProps* g ⟧ ⟹ $\mathcal{V}\ g \subseteq \mathcal{V}\ g'$
apply(*rule next-plane0-incr*)
apply(*erule* (*1*) *subset-trans*)
apply(*simp add*: *vertices-makeFaceFinal*)
defer apply *assumption*+
apply (*auto simp*: *splitFace-def split-def vertices-graph*)
done

15.6 Increasing vertex degrees

lemma *next-plane0-incr-faceListAt*:
⟦ $g' \in$ *set* (*next-plane0*$_p$ g); *minGraphProps* g ⟧
⟹ |*faceListAt* g| ≤ |*faceListAt* g'| &

$(\forall\, v < |faceListAt\ g|.\ |faceListAt\ g\ !\ v| \leq |faceListAt\ g'\ !\ v|\ )$
(**is** - $\Longrightarrow$ - $\Longrightarrow$ *?Q g g'*)
apply(*rule next-plane0-incr*[**where** Q = *?Q*])
prefer *4* **apply** *assumption*
prefer *4* **apply** *assumption*
apply(*rule conjI*) **apply** *fastforce*
apply(*clarsimp*)
apply(*erule allE*, *erule impE*, *assumption*)
apply(*erule-tac* $x = v$ **in** *allE*, *erule impE*) **apply** *force*
apply *force*
apply(*simp add*: *makeFaceFinal-def makeFaceFinalFaceList-def*)
apply (*simp add*: *splitFace-def split-def nth-append nth-list-update*)
done

lemma *next-plane0-incr-degree*:
$[\![\ g' \in set\ (next\text{-}plane0_p\ g);\ minGraphProps\ g;\ v \in \mathcal{V}\ g\ ]\!]$
$\Longrightarrow degree\ g\ v \leq degree\ g'\ v$
apply(*frule* (*1*) *next-plane0-incr-faceListAt*)
apply(*frule* (*1*) *next-plane0-vertices-subset*)
apply(*simp add*:*degree-def facesAt-def*)
apply(*frule minGraphProps4*)
apply(*simp add*:*vertices-graph*)
done

15.7 Increasing *except*

lemma *next-plane0-incr-except*:
assumes $g' \in set\ (next\text{-}plane0_p\ g)$ *inv g* $v \in \mathcal{V}\ g$
shows *except g v* $\leq$ *except g' v*
proof (*unfold except-def*)
note *inv'* = *invariantE*[*OF inv-inv-next-plane0*, *OF assms*(*1*,*2*)]
note *mgp* = *inv-mgp*[*OF assms*(*2*)] **and** *mgp'* = *inv-mgp*[*OF inv'*]
note *dist* = *distinct-filter*[*OF mgp-dist-facesAt*[*OF mgp* ‹*v* : $\mathcal{V}$ *g*›]]
have $v \in \mathcal{V}\ g'$
using *assms*(*3*) *next-plane0-vertices-subset*[*OF assms*(*1*) *mgp*] **by** *blast*
note *dist'* = *distinct-filter*[*OF mgp-dist-facesAt*[*OF mgp'* ‹*v* : $\mathcal{V}$ *g'*›]]
have $|[f \leftarrow facesAt\ g\ v\ .\ final\ f \wedge 5 \leq |vertices\ f|\ ]| =$
card{*f*$\in$ *set*(*facesAt g v*) . *final f* $\wedge$ 5 $\leq$ |*vertices f*|}
(**is** *?L* = *card ?M*) **using** *distinct-card*[*OF dist*] **by** *simp*
also have *?M* = {*f*$\in \mathcal{F}$ *g*. $v \in \mathcal{V}\ f \wedge$ *final f* $\wedge$ 5 $\leq$ |*vertices f*|}
by(*simp add*: *minGraphProps-facesAt-eq*[*OF mgp assms*(*3*)])
also have ... = {*f* $\in$ *set*(*finals g*) . $v \in \mathcal{V}\ f \wedge$ 5 $\leq$ |*vertices f*|}
by(*auto simp*:*finals-def*)
also have *card* ... $\leq$ *card*{*f* $\in$ *set*(*finals g'*). $v \in \mathcal{V}\ f \wedge$ 5 $\leq$ |*vertices f*|}
(**is** - $\leq$ *card ?M*)
apply(*rule card-mono*)
apply *simp*
using *next-plane0-finals-subset*[*OF assms*(*1*)] **by** *blast*

also have *?M* = $\{f \in \mathcal{F}\ g' .\ v \in \mathcal{V}\ f \wedge$ *final* $f \wedge 5 \leq |$*vertices* $f|\}$
by(*auto simp*:*finals-def*)
also have ... = $\{f \in$ *set*(*facesAt* $g'\ v) .$ *final* $f \wedge 5 \leq |$*vertices* $f|\}$
by(*simp add*: *minGraphProps-facesAt-eq*[*OF mgp'* ‹$v \in \mathcal{V}\ g'$›])
also have *card* ... =
$|[f \leftarrow$ *facesAt* $g'\ v$. *final* $f \wedge 5 \leq |$*vertices* $f|\ ]|$ (**is** - = *?R*)
using *distinct-card*[*OF dist'*] **by** *simp*
finally show *?L* $\leq$ *?R* **.**
qed

15.8 Increasing edges

lemma *next-plane0-set-edges-subset*:
$[\![$ *minGraphProps* g; $g\ [$*next-plane0*$_p]\rightarrow g'\ ]\!] \Longrightarrow$ *edges* $g \subseteq$ *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

15.9 Increasing final vertices

declare *atLeastLessThan-iff*[*iff*]

lemma *next-plane0-incr-finV*:
$[\![g' \in$ *set* (*next-plane0*$_p$ g); *minGraphProps* $g\ ]\!]$
$\Longrightarrow \forall v \in \mathcal{V}\ g.\ v \in \mathcal{V}\ g' \wedge$
$((\forall f \in \mathcal{F}\ g.\ v \in \mathcal{V}\ f \longrightarrow$ *final* $f) \longrightarrow$
$(\forall f \in \mathcal{F}\ g'.\ v \in \mathcal{V}\ f \longrightarrow f \in \mathcal{F}\ g))$ (**is** - $\Longrightarrow$ - $\Longrightarrow$ *?Q* $g\ g'$)
apply(*rule next-plane0-incr*[**where** Q = *?Q* **and** g=g **and** g'=g'])
prefer *4* **apply** *assumption*
prefer *4* **apply** *assumption*
apply *fast*
apply(*clarsimp simp*:*makeFaceFinal-def vertices-graph makeFaceFinalFaceList-def*)
apply(*drule replace5*)
apply(*erule disjE*)**apply** *blast*
apply(*simp add*:*setFinal-def*)
apply(*unfold pre-splitFace-def*)
apply(*clarsimp simp*:*splitFace-def split-def vertices-graph*)
apply(*rule conjI*)
apply(*clarsimp simp*:*split-face-def vertices-graph atLeastLessThan-def*)
apply(*blast dest*:*inbetween-inset*)
apply(*clarsimp*)
apply(*erule disjE*[*OF replace5*]) **apply** *blast*
apply(*clarsimp simp*:*split-face-def vertices-graph*)
apply(*blast dest*:*inbetween-inset*)
done

lemma *next-plane0-finalVertex-mono*:
⟦$g' \in$ *set* (*next-plane0*$_p$ g); *inv* g; $u \in \mathcal{V}\ g$; *finalVertex* $g\ u$ ⟧
⟹ *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

15.10 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* $g\ v$ ⟧
⟹ *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 assms*(*1*,*2*)]
note *mgp* = *inv-mgp*[*OF assms*(*2*)] **and** *mgp'* = *inv-mgp*[*OF inv'*]
note *dist* = *distinct-filter*[*OF mgp-dist-facesAt*[*OF mgp* ‹$v : \mathcal{V}\ g$›]]
have $v \in \mathcal{V}\ g'$
using *assms*(*3*) *next-plane0-vertices-subset*[*OF assms*(*1*) *mgp*] **by** *blast*
note *dist'* = *distinct-filter*[*OF mgp-dist-facesAt*[*OF mgp'* ‹$v : \mathcal{V}\ g'$›]]
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*{$f \in$ *set*(*facesAt* $g\ v$) . $P\ f$}
by(*simp add*: *next-plane0-finalVertex-facesAt-eq*[*OF assms*])
also have ... = |*filter* P (*facesAt* $g\ v$)|
using *distinct-card*[*OF dist*] **by** *simp*
finally show *?thesis* **.**
qed

15.11 Properties of *subdivFace'*

lemma *new-edge-subdivFace'*:
⋀$f\ v\ n\ g$.
pre-subdivFace' $g\ f\ u\ v\ n$ *ovs* ⟹ *minGraphProps* g ⟹ $f \in \mathcal{F}\ g$ ⟹
subdivFace' $g\ f\ v\ n$ *ovs* = *makeFaceFinal* $f\ g$ ∨
(∀$f' \in \mathcal{F}$ (*subdivFace'* $g\ f\ v\ n$ *ovs*) − ($\mathcal{F}\ g$ − {f}).
∃$e \in \mathcal{E}\ f'$. $e \notin \mathcal{E}\ g$)
proof (*induct ovs*)

```
  case Nil thus ?case by simp
next
  case (Cons ov ovs)
  note IH = Cons(1) and pre = Cons(2) and mgp = Cons(3) and fg = Cons(4)
  have uf: u ∈ 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 using [[simp-depth-limit = 5]] by (simp add: pre-subdivFace'-Some2)
      note IH[OF pre' mgp fg]
    } moreover
    { let ?vs = [countVertices g..<countVertices g + n]
      let ?fdg = splitFace g v w f ?vs
      let  ?f₁ = fst ?fdg and ?f₂ = fst(snd ?fdg) and ?g' = snd(snd ?fdg)
      let ?g'' = subdivFace' ?g' ?f₂ w 0 ovs
      let ?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* ∈ $\mathcal{V}$ *f* ⟧ ⟹
between (*vertices f*) *u* (*last* (*verticesFrom f u*)) =
butlast(*tl*(*verticesFrom f u*))
apply(*drule split-list*)
apply (*fastforce dest*: *last-in-set*
simp: *between-def verticesFrom-Def split-def*
last-append butlast-append fst-splitAt-last)
done

lemma *final-subdivFace'*: ⋀*f u n g*. *minGraphProps g* ⟹
pre-subdivFace' g f r u n ovs ⟹ *f* ∈ $\mathcal{F}$ *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*) ∈ $\mathcal{E}$ *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*)
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* ∈ $\mathcal{V}$ *f* **and** *u* ∈ $\mathcal{V}$ *f* **and** *distf*: *distinct* (*vertices f*)
using *pre* **by**(*simp add*:*pre-subdivFace'-def*)+
show *?case*
proof (*cases ov*)
case *None*
have *pre'*: *pre-subdivFace' g f r u* (*Suc n*) *ovs*

```
    using None pre by (simp add: pre-subdivFace'-None)
  have ovs ≠ [] 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 using [[simp-depth-limit = 5]] by (simp 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' :: face 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 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: f∈set (finals g) and |vertices f| = maxGon p
    by auto
 moreover from succs(1) f have f∈set (finals g') by (rule next-plane0-finals-incr)
  ultimately show ?case by auto
qed


end
```

16 Summation Over Lists

```
theory ListSum
imports ListAux
begin

primrec ListSum :: 'b list ⇒ ('b ⇒ 'a::comm-monoid-add) ⇒ 'a::comm-monoid-add
where
  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)     (∑ -∈- - [0, 0, 10] 10)
translations ∑ x∈xs f == CONST ListSum xs (λx. f)
```

lemma [*simp*]: $(\sum v \in V\ 0) = (0::nat)$ **by** (*induct V*) *simp-all*

lemma *ListSum-compl1*:
$(\sum x \in [x \leftarrow xs.\ \neg\ P\ x]\ f\ x) + (\sum x \in [x \leftarrow xs.\ P\ x]\ f\ x) = (\sum x \in xs\ (f\ x::nat))$
by (*induct xs*) *simp-all*

lemma *ListSum-compl2*:
$(\sum x \in [x \leftarrow xs.\ P\ x]\ f\ x) + (\sum x \in [x \leftarrow xs.\ \neg\ P\ x]\ f\ x) = (\sum x \in xs\ (f\ x::nat))$
by (*induct xs*) *simp-all*

lemmas *ListSum-compl* = *ListSum-compl1 ListSum-compl2*

lemma *ListSum-conv-setsum*:
distinct xs $\Longrightarrow$ *ListSum xs f* = *setsum f* (*set xs*)
by(*induct xs*) *simp-all*

lemma *listsum-cong*:
$[\![\ xs = ys;\ \bigwedge y.\ y \in set\ ys \Longrightarrow f\ y = g\ y\ ]\!]$
$\Longrightarrow$ *ListSum xs f* = *ListSum ys g*
apply *simp*
apply(*erule thin-rl*)
by (*induct ys*) *simp-all*

lemma *strong-listsum-cong*[*cong*]:
$[\![\ xs = ys;\ \bigwedge y.\ y \in set\ ys =simp=> f\ y = g\ y\ ]\!]$
$\Longrightarrow$ *ListSum xs f* = *ListSum ys g*
by(*auto simp*:*simp-implies-def intro*!:*listsum-cong*)

lemma *ListSum-eq* [*trans*]:
$(\bigwedge v.\ v \in set\ V \Longrightarrow f\ v = g\ v) \Longrightarrow (\sum v \in V\ f\ v) = (\sum v \in V\ g\ v)$
by(*auto intro*!:*listsum-cong*)

lemma *ListSum-disj-union*:
distinct A $\Longrightarrow$ *distinct B* $\Longrightarrow$ *distinct C* $\Longrightarrow$
set C = *set A* $\cup$ *set B* $\Longrightarrow$
set A $\cap$ *set B* = $\{\}$ $\Longrightarrow$
$(\sum a \in C\ (f\ a)) = (\sum a \in A\ f\ a) + (\sum a \in B\ (f\ a::nat))$
by (*simp add*: *ListSum-conv-setsum setsum-Un-disjoint*)

lemma *listsum-const*[*simp*]:
$(\sum x \in xs\ k) = length\ xs * k$
by (*induct xs*) (*simp-all add*: *ring-distribs*)

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*
qed

lemma *ListSum1-bound*:
a ∈ set F ⟹ (d a::nat)≤ (∑ f ∈ F d f)
by (*induct F*) *auto*

end

17 Tameness

theory *Tame*
imports *Graph ListSum*
begin

17.1 Constants

definition *squanderTarget* :: *nat* **where**
squanderTarget ≡ *15410*

definition *excessTCount* :: *nat* **where**

a ≡ *6300*

definition *squanderVertex* :: *nat* ⇒ *nat* ⇒ *nat* **where**

b p q ≡ *if p = 0 ∧ q = 3 then 6180*
else if p = 0 ∧ q = 4 then 9700
else if p = 1 ∧ q = 2 then 6560
else if p = 1 ∧ q = 3 then 6180
else if p = 2 ∧ q = 1 then 7970
else if p = 2 ∧ q = 2 then 4120
else if p = 2 ∧ q = 3 then 12851
else if p = 3 ∧ q = 1 then 3110
else if p = 3 ∧ q = 2 then 8170

else if $p = 4 \wedge q = 0$ *then* 3470
else if $p = 4 \wedge q = 1$ *then* 3660
else if $p = 5 \wedge q = 0$ *then* 400
else if $p = 5 \wedge q = 1$ *then* 11360
else if $p = 6 \wedge q = 0$ *then* 6860
else if $p = 7 \wedge q = 0$ *then* 14500
else squanderTarget

definition *squanderFace* :: *nat* ⇒ *nat* **where**

d $n \equiv$ *if* $n = 3$ *then* 0
else if $n = 4$ *then* 2060
else if $n = 5$ *then* 4819
else if $n = 6$ *then* 7578
else squanderTarget

17.2 Separated sets of vertices

A set of vertices V is *separated*, iff the following conditions hold:

2. No two vertices in V are adjacent:

definition $separated_2$ :: *graph* ⇒ *vertex set* ⇒ *bool* **where**
$separated_2\ g\ V \equiv \forall v \in V.\ \forall f \in set\ (facesAt\ g\ v).\ f \cdot v \notin V$

3. No two vertices lie on a common quadrilateral:

definition $separated_3$ :: *graph* ⇒ *vertex set* ⇒ *bool* **where**
$separated_3\ g\ V \equiv$
$\forall v \in V.\ \forall f \in set\ (facesAt\ g\ v).\ |vertices\ f| \leq 4 \longrightarrow \mathcal{V}\ f \cap V = \{v\}$

A set of vertices is called *separated*, iff no two vertices are adjacent or lie on a common quadrilateral:

definition *separated* :: *graph* ⇒ *vertex set* ⇒ *bool* **where**
$separated\ g\ V \equiv separated_2\ g\ V \wedge separated_3\ g\ V$

17.3 Admissible weight assignments

A weight assignment w :: *face* ⇒ *nat* assigns a natural number to every face.

We formalize the admissibility requirements as follows:

definition $admissible_1$ :: (*face* ⇒ *nat*) ⇒ *graph* ⇒ *bool* **where**
$admissible_1\ w\ g \equiv \forall f \in \mathcal{F}\ g.$ d $|vertices\ f| \leq w\ f$

definition $admissible_2$ :: (*face* ⇒ *nat*) ⇒ *graph* ⇒ *bool* **where**
$admissible_2\ w\ g \equiv$
$\forall v \in \mathcal{V}\ g.\ except\ g\ v = 0 \longrightarrow$ b $(tri\ g\ v)\ (quad\ g\ v) \leq (\sum_{f \in facesAt\ g\ v} w\ f)$

definition $admissible_3$ :: (*face* ⇒ *nat*) ⇒ *graph* ⇒ *bool* **where**

admissible$_3$ *w g* ≡
∀ *v* ∈ $\mathcal{V}$ *g*. *vertextype g v* = (5,0,1) ⟶ ($\sum_{f \in filter\ triangle\ (facesAt\ g\ v)}$ *w*(*f*))
>= a

Finally we define admissibility of weights functions.

definition *admissible* :: (*face* ⇒ *nat*) ⇒ *graph* ⇒ *bool* **where**
admissible w g ≡ *admissible*$_1$ *w g* ∧ *admissible*$_2$ *w g* ∧ *admissible*$_3$ *w g*

17.4 Tameness

definition *tame9a* :: *graph* ⇒ *bool* **where**
tame9a g ≡ ∀ *f* ∈ $\mathcal{F}$ *g*. 3 ≤ |*vertices f*| ∧ |*vertices f*| ≤ 6

definition *tame10* :: *graph* ⇒ *bool* **where**
tame10 g = (*let n* = *countVertices g in* 13 <= *n* & *n* <= 15)

definition *tame10ub* :: *graph* ⇒ *bool* **where**
tame10ub g = (*countVertices g* <= 15)

definition *tame11a* :: *graph* ⇒ *bool* **where**
tame11a g = (∀ *v* ∈ $\mathcal{V}$ *g*. 3 <= *degree g v*)

definition *tame11b* :: *graph* ⇒ *bool* **where**
tame11b g = (∀ *v* ∈ $\mathcal{V}$ *g*. *degree g v* ≤ (*if except g v* = 0 *then* 7 *else* 6))

definition *tame12o* :: *graph* ⇒ *bool* **where**
tame12o g =
(∀ *v* ∈ $\mathcal{V}$ *g*. *except g v* ≠ 0 ∧ *degree g v* = 6 ⟶ *vertextype g v* = (5,0,1))

7. There exists an admissible weight assignment of total weight less than the target:

definition *tame13a* :: *graph* ⇒ *bool* **where**
tame13a g = (∃ *w*. *admissible w g* ∧ ($\sum_{f\ \in\ faces\ g}$ *w f*) < *squanderTarget*)

Finally we define the notion of tameness.

definition *tame* :: *graph* ⇒ *bool* **where**
tame g ≡ *tame9a g* ∧ *tame10 g* ∧ *tame11a g* ∧ *tame11b g* ∧ *tame12o g* ∧ *tame13a g*

theory *Plane1Props*
imports *Plane1 PlaneProps Tame*
begin

lemma *next-plane-subset*:
∀ *f* ∈ $\mathcal{F}$ *g*. *vertices f* ≠ [] ⟹
set (*next-plane*$_p$ *g*) ⊆ *set* (*next-plane0*$_p$ *g*)
apply(*clarsimp simp*:*next-plane0-def next-plane-def minimalFace-def finalGraph-def*)
apply(*rule-tac x* = *minimal* (*size* ∘ *vertices*) (*nonFinals g*) **in** *bexI*)

apply(*rule-tac* x = *minimalVertex* g (*minimal* (*size* ∘ *vertices*) (*nonFinals* g)) **in** *bexI*)
apply *blast*
apply(*subgoal-tac* ∀f∈*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 $g \Longrightarrow g\ [next\text{-}plane_p]\rightarrow g' \Longrightarrow g\ [next\text{-}plane0_p]\rightarrow g'$
using *next-plane-subset* **by**(*blast dest*: *mgp-vertices-nonempty*)

lemma *inv-inv-next-plane*: *invariant inv* $next\text{-}plane_p$
apply(*rule inv-subset*[*OF inv-inv-next-plane0*])
apply(*blast dest*: *mgp-next-plane0-if-next-plane*[*OF inv-mgp*])
done

end

18 Enumeration of Tame Plane Graphs

theory *Generator*
imports *Plane1 Tame*
begin

definition *faceSquanderLowerBound* :: *graph* ⇒ *nat* **where**
faceSquanderLowerBound $g \equiv \sum_{f\ \in\ finals\ g}$ d $|vertices\ f|$

definition *d3-const* :: *nat* **where**
d3-const == d *3*

definition *d4-const* :: *nat* **where**
d4-const == d *4*

definition *excessAtType* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
excessAtType t q e ≡
if e = 0 then if 7 < t + q then squanderTarget
else b *t q − t ∗ d3-const − q ∗ d4-const*
else if t + q + e ≠ 6 then 0
else if t=5 then a *else squanderTarget*

declare *d3-const-def*[*simp*] *d4-const-def*[*simp*]

definition *ExcessAt* :: *graph* ⇒ *vertex* ⇒ *nat* **where**
ExcessAt g v ≡ *if* ¬ *finalVertex g v then 0*
else excessAtType (*tri g v*) (*quad g v*) (*except g v*)

definition *ExcessTable* :: *graph* ⇒ *vertex list* ⇒ (*vertex* × *nat*) *list* **where**
ExcessTable g vs ≡
[(*v*, *ExcessAt g v*). *v* ← [*v* ← *vs*. *0* < *ExcessAt g v*]]

Implementation:

lemma [*code*]:
ExcessTable g =
List.map-filter (λ*v*. *let e* = *ExcessAt g v in if 0* < *e then Some* (*v*, *e*) *else None*)
by (*rule ext*) (*simp add*: *ExcessTable-def map-filter-def*)

definition *deleteAround* :: *graph* ⇒ *vertex* ⇒ (*vertex* × *nat*) *list* ⇒ (*vertex* × *nat*) *list* **where**
deleteAround g v ps ≡
let fs = *facesAt g v*;
ws = $\bigsqcup_{f \in fs}$ *if* |*vertices f*| = *4 then* [*f*•*v*, f^2•*v*] *else* [*f*•*v*] *in*
removeKeyList ws ps

19 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* ∈ $\mathcal{V}$ *g* ⟧ ⟹ *tri g v* + *quad g v* ≤ *7*
using *tri-quad-le-degree*[*of g v*]

by(*auto simp*:*tame-def tame11b-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**(*fastforce intro*:*RTranCl.succs*)
  next
    assume ¬ *ok h'* **note** *fin-tame* = *ok-untame*[*OF* ‹*P h'*› ‹¬ *ok h'*›]
    have *h''* = *h'* **using** *fin-tame*
      by(*rule-tac RTranCl.cases*[*OF succs*(*2*)])(*auto simp*:*fin*)
    hence *False* **using** *fin-tame succs* **by** *fast*
    thus *?case* **..**
  qed
qed

definition *untame* :: (*graph* ⇒ *bool*) ⇒ *bool* **where**
untame P ≡ ∀ *g*. *final g* ∧ *P g* ⟶ ¬ *tame g*

lemma *filterout-untame-succs*:
assumes *invP*: *invariant P f* **and** *invPU*: *invariant* (%*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 $h' \notin set(f'\ h)$
with $succs(4)$ $succs(1)$ **have** $U\ h'$ **by** (*rule new-untame*)
hence *False* **using** Ph' *RTranCl-inv*[*OF invPU*] *untame succs*
by (*unfold untame-def*) *fast*
thus *?case* **..**
qed
qed

end

20 Neglectable Final Graphs

theory *TameEnum*
imports *Generator*
begin

definition *is-tame* :: *graph* ⇒ *bool* **where**
is-tame g ≡ *tame10 g* ∧ *tame11a g* ∧ *tame12o g* ∧ *is-tame13a g*

definition *next-tame* :: *nat* ⇒ *graph* ⇒ *graph list* (*next'-tame*$_-$) **where**
$next\text{-}tame_p \equiv filter\ (\lambda g.\ \neg\ final\ g \vee is\text{-}tame\ g)\ o\ next\text{-}tame0_p$

definition *TameEnumP* :: *nat* ⇒ *graph set* (*TameEnum*$_-$) **where**
$TameEnum_p \equiv \{g.\ Seed_p\ [next\text{-}tame_p]{\rightarrow}*\ g \wedge final\ g\}$

definition *TameEnum* :: *graph set* **where**
$TameEnum \equiv \bigcup p{\leq}3.\ TameEnum_p$

end

21 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* ∈ {*v*. ∃*f* ∈ *set* (*facesAt g a*).
(*length* (*vertices f*) = 4) ∧ *v* ∈ {*f* · *a*, *f* · (*f* · *a*)}
∨ (*length* (*vertices f*) ≠ 4) ∧ (*v* = *f* · *a*)}
then deleteAround g a ps

```
    else p#deleteAround g a ps)
by (fastforce simp: nextV2 deleteAround-def)

lemma deleteAround-subset: set (deleteAround g a ps) ⊆ 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


definition deleteAround′ :: graph ⇒ vertex ⇒ (vertex × nat) list ⇒
   (vertex × nat) list where
  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 $\Longrightarrow$ *a* : $\mathcal{V}$ *g* $\Longrightarrow$ *f* $\in$ *set* (*facesAt g a*) $\Longrightarrow$
($f^{-1} \cdot a$, *b*) $\notin$ *set* (*deleteAround g a ps*)
proof −
assume *a*: *minGraphProps g a* : $\mathcal{V}$ *g f* $\in$ *set* (*facesAt g a*)
have ($f^{-1} \cdot a$, *a*) $\in$ $\mathcal{E}$ *f* **using** *a*
by(*blast intro*:*prevVertex-in-edges minGraphProps*)
then obtain *f* ′ :: *face* **where** *f* ′: *f* ′ $\in$ *set*(*facesAt g a*)
and *e*: (*a*, $f^{-1} \cdot a$) $\in$ $\mathcal{E}$ *f* ′
using *a* **by**(*blast dest*:*mgp-edge-face-ex*)
have (*f* ′ · *a*, *b*) $\notin$ *set* (*deleteAround g a ps*) **using** *f* ′
by (*auto simp add*: *deleteAround-eq deleteAround*′*-def removeKeyList-eq*)
moreover have *f* ′ · *a* = $f^{-1} \cdot a$
using *e* **by** (*simp add*:*edges-face-eq*)
ultimately show *?thesis* **by** *simp*
qed

lemma *deleteAround-separated*:
assumes *mgp*: *minGraphProps g* **and** *fin*: *final g* **and** *ag*: *a* : $\mathcal{V}$ *g* **and** *4*: |*vertices f*| ≤ *4*
and *f*: *f* $\in$ *set*(*facesAt g a*)
shows $\mathcal{V}$ *f* ∩ *set* [*fst p*. *p* ← *deleteAround g a ps*] ⊆ {*a*} (**is** *?A*)
proof −
note *MGP* = *mgp ag f*
have *af*: *a* $\in$ $\mathcal{V}$ *f* **using** *MGP* **by**(*blast intro*:*minGraphProps*)
have *2* < |*vertices f*| **using** *MGP* **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* $\in$ $\mathcal{V}$ *f*
b $\in$ *set* (*map fst* (*deleteAround g a ps*)) **by** *auto*
from *MGP* **have** *d*: *distinct* (*vertices f*)
by(*blast intro*:*minGraphProps*)
with *af 3* **have** $\mathcal{V}$ *f* = {*a*, *f* · *a*, *f* · (*f* · *a*)}
by (*rule-tac vertices-triangle*)
also from *d af 3* **have**
f · (*f* · *a*) = $f^{-1} \cdot a$
by (*simp add*: *triangle-nextVertex-prevVertex*)
finally have
b $\in$ {*f* · *a*, $f^{-1} \cdot a$}
using *b1* **by** *simp*
with *MGP* **have** *b* $\notin$ *set* (*map fst* (*deleteAround g a ps*))

```
      using deleteAround-nextVertex deleteAround-prevVertex by auto
      then show False by contradiction (rule b1)
    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 have d: distinct (vertices f) by(blast intro:minGraphProps)
      with af 4 have 𝒱 f = {a, f · a, f · (f · a), f · (f · (f · a))}
        by (rule-tac vertices-quad)
      also from d af 4 have f · (f · (f · a)) = f⁻¹ · a
        by (simp add: quad-nextVertex-prevVertex)
      finally have b ∈ {f · a, f · (f · a), f⁻¹ · a}
      using b1 by simp
      with MGP 4 have b ∉ set (map fst (deleteAround g a ps))
      using deleteAround-nextVertex deleteAround-prevVertex
          deleteAround-nextVertex-nextVertex by auto
      then show False by contradiction (rule b1)
    qed
  qed
qed


lemma [iff]: separated g {}
by (simp add: separated-def separated₂-def separated₃-def)

lemma separated-insert:
assumes mgp: minGraphProps g and a: a ∈ 𝒱 g
  and Vg: V <= 𝒱 g
  and ps: separated g V
  and s2: (⋀f. f ∈ set (facesAt g a) ⟹ f · a ∉ V)
  and s3: (⋀f. f ∈ set (facesAt g a) ⟹
      |vertices f| ≤ 4 ⟹ 𝒱 f ∩ V ⊆ {a})
  shows separated g (insert a V)
proof (simp add: separated-def separated₂-def separated₃-def,
 intro conjI ballI impI)
  fix f assume f: f ∈ set (facesAt g a)
  then show f · a ≠ a by (rule mgp-facesAt-no-loop[OF mgp a])
  from f show f · a ∉ V by (rule s2)
next
  fix f v assume v: f ∈ set (facesAt g v) and vV: v ∈ V
  have v : 𝒱 g using vV Vg by blast
  show f · v ≠ a
  proof
    assume f: f · v = a
    then obtain f' where f': f' ∈ set(facesAt g a) and v: f' · a = v
```

```
    using mgp-nextVertex-face-ex2[OF mgp ⟨v : 𝒱 g⟩ v] by blast
   have f' · a ∈ V using v vV by simp
   with f' s2 show False by blast
  qed
  from ps v vV show f · v ∉ V
   by (simp add: separated-def separated₂-def)
next
  fix f assume f:  f ∈ set (facesAt g a) |vertices f| ≤ 4
  then have 𝒱 f ∩ V ⊆ {a} by (rule s3)
  moreover from mgp a f have a ∈ 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: separated-def separated₃-def)
  have v : 𝒱 g using a Vg by blast
  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 ⟨v : 𝒱 g⟩ 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


function ExcessNotAtRecList :: (vertex, nat) table ⇒ graph ⇒ vertex list where
  ExcessNotAtRecList [] = (%g. [])
  | ExcessNotAtRecList ((x, y) # ps) = (%g.
     let l1 = ExcessNotAtRecList ps g;
     l2 = ExcessNotAtRecList (deleteAround g x ps) g in
     if ExcessNotAtRec ps g
      ≤ y + ExcessNotAtRec (deleteAround g x ps) g
     then x # l2 else l1)
by pat-completeness auto
termination by (relation measure size)
  (auto simp add: less-Suc-eq-le length-deleteAround)
```

lemma *isTable-deleteAround*:
isTable E vs ((*a*,*b*)#*ps*) ⟹ *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*: *ExcessNotAtRecList.induct*)
case *1* **show** *?case* **by** *simp*
next
case (*2 a b ps*)
from *2* **have** *prem*: *?T* ((*a*,*b*)#*ps*) **by** *blast*
then have *E*: *b* = *E a* **by** (*simp add*: *isTable-eq*)
from *2* **have** *hyp1*: *?T* (*deleteAround g a ps*) ⟹
?P (*deleteAround g a ps*) **by** *blast*
from *2* **have** *hyp2*: *?T ps* ⟹ *?P ps* **by** *blast*
have *H1*: *?P* (*deleteAround g a ps*)
by (*rule hyp1*, *rule isTable-deleteAround*) (*rule prem*)
have *H2*: *?P ps* **by** (*rule hyp2*, *rule isTable-Cons*, *rule prem*)
show *?P* ((*a*,*b*)#*ps*)
proof *cases*
assume
ExcessNotAtRec ps g
≤ *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*)
case *1* **show** *?case* **by** *simp*
next
case (*2 a b ps*)
presume *H1*: *?P* (*deleteAround g a ps*)
presume *H2*: *?P ps*
show *?P* ((*a*, *b*) # *ps*)
proof *cases*
assume *a*: *ExcessNotAtRec ps g*
≤ *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 separated-ExcessNotAtRecList:
 minGraphProps g ⟹ final g ⟹ isTable E (vertices g) ps ⟹
  separated g (set (ExcessNotAtRecList ps g))
proof −
  assume fin: final g and mgp: minGraphProps g
  show
   isTable E (vertices g) ps ⟹ separated g (set (ExcessNotAtRecList ps g))
   (is ?T ps ⟹ ?P ps)
  proof (induct rule: ExcessNotAtRec.induct)
   case 1 show ?case by simp
  next
   case (2 a b ps)
   from 2 have prem: ?T ((a,b)#ps) by blast
   then have E: b = E a by (simp add: isTable-eq)
   have a :𝒱 g using prem by(auto simp: isTable-def)
   from 2 have hyp1: ?T (deleteAround g a ps) ⟹
    ?P (deleteAround g a ps) by blast
   from 2 have hyp2:  ?T ps ⟹ ?P ps by blast
   have H1: ?P (deleteAround g a ps)
    by (rule hyp1, rule isTable-deleteAround) (rule prem)
   have H2: ?P ps by (rule hyp2, rule isTable-Cons) (rule prem)

   show ?P ((a,b)#ps)
   proof cases
    assume c: ExcessNotAtRec ps g
     ≤ b + ExcessNotAtRec (deleteAround g a ps) g
    have separated g
     (insert a (set (ExcessNotAtRecList (deleteAround g a ps) g)))
    proof (rule separated-insert[OF mgp])
     from prem show a ∈ set (vertices g) by (auto simp add: isTable-def)

     show set (ExcessNotAtRecList (deleteAround g a ps) g) ⊆ 𝒱 g
     proof−
      have set (ExcessNotAtRecList (deleteAround g a ps) g) <=
           set (map fst (deleteAround g a ps))
       by(rule ExcessNotAtRecList-subset[simplified concat-map-singleton])
      also have ... <= set (map fst ps)
```

```
      using deleteAround-subset by fastforce
    finally show ?thesis using prem by(auto simp: isTable-def)
  qed
  from H1
  show pS: separated g
      (set (ExcessNotAtRecList (deleteAround g a ps) g))
    by simp

  fix f assume f: 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
  from this f have set (vertices f)
    ∩ set [fst p. p ← deleteAround g a ps] ⊆ {a}
    by (rule deleteAround-separated[OF mgp fin ‹a : V g›])
  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*) $\Longrightarrow$ *distinct* (*ExcessNotAtRecList ps g*)
    (**is** *?T ps* $\Longrightarrow$ *?P ps*)
proof (*induct rule*: *ExcessNotAtRec.induct*)
  case *1* **show** *?case* **by** *simp*
next
  case (*2 a b ps*)
  from *2* **have** *prem*: *?T* ((*a*,*b*)#*ps*) **by** *blast*
  then have *a*: *a* $\notin$ *set* (*map fst ps*) **by** *simp*
  from *2* **have** *hyp1*: *?T* (*deleteAround g a ps*) $\Longrightarrow$
    ?P (*deleteAround g a ps*) **by** *blast*
  from *2* **have** *hyp2*: *?T ps* $\Longrightarrow$ *?P ps* **by** *blast*
  from *2* **have** *?T ps* **by** *simp*
  then have *H1*: *?P* (*deleteAround g a ps*)
   by (*rule-tac hyp1*) (*rule distinct-deleteAround* [*simplified*])
  from *prem* **have** *H2*: *?P ps*
    by (*rule-tac hyp2*) *simp*

  have *a* $\notin$ *set* (*ExcessNotAtRecList* (*deleteAround g a ps*) *g*)
  proof
    assume *a* $\in$ *set* (*ExcessNotAtRecList* (*deleteAround g a ps*) *g*)
    also have *set* (*ExcessNotAtRecList* (*deleteAround g a ps*) *g*)
      $\subseteq$ *set* [*fst p*. *p* $\leftarrow$ *deleteAround g a ps*]
     by (*rule ExcessNotAtRecList-subset*)
    also have *set* (*deleteAround g a ps*) $\subseteq$ *set ps*
      by (*rule deleteAround-subset*)
    then have *set* [*fst p*. *p* $\leftarrow$ *deleteAround g a ps*]
      $\subseteq$ *set* [*fst p*. *p* $\leftarrow$ *ps*] **by** *auto*
    finally have *a* $\in$ *set* (*map fst ps*) **by** *simp*
    with *a* **show** *False* **by** *contradiction*
  qed
  with *H1 H2* **show** *?P* ((*a*,*b*)#*ps*)
    by (*simp add*: *ExcessNotAtRecList-subset*)
qed

primrec *ExcessTable-cont* ::
  (*vertex* $\Rightarrow$ *nat*) $\Rightarrow$ *vertex list* $\Rightarrow$ (*vertex* $\times$ *nat*) *list*
where
  ExcessTable-cont ExcessAtPG [] = [] |
  ExcessTable-cont ExcessAtPG (*v*#*vs*) =
   (*let vi* = *ExcessAtPG v in*
     if 0 < *vi*
     then (*v*, *vi*)#*ExcessTable-cont ExcessAtPG vs*
     else ExcessTable-cont ExcessAtPG vs)

definition *ExcessTable'* :: *graph* $\Rightarrow$ *vertex list* $\Rightarrow$ (*vertex* $\times$ *nat*) *list* **where**

ExcessTable′ g ≡ 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)

```
   ∧ separated 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 separated g (set ?V)
    by(rule separated-ExcessNotAtRecList[OF mgp fin t])

  have set (ExcessNotAtRecList ?ps g) ⊆ set (map fst ?ps)
    by (rule ExcessNotAtRecList-subset[simplified concat-map-singleton])
  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[simplified concat-map-singleton])
qed

lemma excess-eq:
  assumes 7: t + q ≤ 7
  shows excessAtType t q 0 + t * d 3 + q * d 4 = b t q
proof −
  note simps = excessAtType-def squanderVertex-def squanderFace-def
    nat-minus-add-max squanderTarget-def
  from 7 have q=0 ∨ q=1 ∨ q=2 ∨ q=3 ∨ q=4 ∨ q=5 ∨ q=6 ∨ q=7 by arith
  then show ?thesis
  proof (elim disjE)
    assume q: q = 0
    with 7 show ?thesis by (simp add: simps)
  next
    assume q: q = 1
    with 7 show ?thesis by (simp add: simps)
  next
    assume q: q = 2
    with 7 show ?thesis by (simp add: simps)
  next
    assume q: q = 3
    with 7 show ?thesis by (simp add: simps)
  next
    assume q: q = 4
    with 7 show ?thesis by (simp add: simps)
  next
    assume q: q = 5
```

```
    with 7 show ?thesis by (simp add: simps)
  next
    assume q: q = 6
    with 7 show ?thesis by (simp add: simps)
  next
    assume q: q = 7
    with 7 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

separating

definition separating :: 'a set ⇒ ('a ⇒ 'b set) ⇒ bool where
  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 *fin*: *finite V a* $\notin$ *V* $\bigwedge f$. *finite* (*F f*) **by** *fact+*

moreover from *s* **have** $\bigwedge v$. *a* $\notin$ *V* $\Longrightarrow$ *v* $\in$ *V* $\Longrightarrow$ *F a* $\cap$ *F v* = {}
by (*simp add*: *separating-insert2*)
with *fin* **have** (*F a*) $\cap$ ($\bigcup v \in V$. *F v*) = {} **by** *auto*

ultimately show *?case* **by** (*simp add*: *setsum-Un-disjoint*)
qed

lemma *separated-separating*:
assumes *Vg*: *set V* <= $\mathcal{V}$ *g*
and *pS*: *separated g* (*set V*)
and *noex*: *ALL f*:*P*. |*vertices f*| <= *4*
shows *separating* (*set V*) (λv. *set* (*facesAt g v*) *Int P*)
proof −
from *pS* **have** *i*: $\forall v \in$*set V*. $\forall f \in$*set* (*facesAt g v*).
|*vertices f*| $\leq$ *4* $\longrightarrow$ *set* (*vertices f*) $\cap$ *set V* = {*v*}
by (*simp add*: *separated-def separated*$_3$*-def*)
show *separating* (*set V*) (λv. *set* (*facesAt g v*) *Int P*)
proof (*simp add*: *separating-def*, *intro ballI impI*)
fix *v1 v2* **assume** *v*: *v1* $\in$ *set V v2* $\in$ *set V v1* $\neq$ *v2*
hence *v1* : $\mathcal{V}$ *g* **using** *Vg* **by** *blast*
show (*set* (*facesAt g v1*) $\cap$ *P*) $\cap$ (*set* (*facesAt g v2*) $\cap$ *P*) = {} (**is** *?P*)
proof (*rule ccontr*)
assume ¬ *?P*
then obtain *f* **where** *f1*: *f* $\in$ *set* (*facesAt g v1*)
and *f2*: *f* $\in$ *set* (*facesAt g v2*) **and** *f* : *P* **by** *auto*
with *noex* **have** *l*: |*vertices f*| $\leq$ *4* **by** *blast*
from *v f1 l i* **have** *set* (*vertices f*) $\cap$ *set V* = {*v1*} **by** *simp*
also from *v f2 l i*
have *set* (*vertices f*) $\cap$ *set V* = {*v2*} **by** *simp*
finally have *v1* = *v2* **by** *auto*
then show *False* **by** *contradiction* (*rule v*)
qed
qed
qed

lemma *ListSum-V-F-eq-ListSum-F*:
assumes *pl*: *inv g*
and *pS*: *separated g* (*set V*) **and** *dist*: *distinct V*
and *V-subset*: *set V* $\subseteq$ *set* (*vertices g*)
and *noex*: *ALL f* : *Collect P*. |*vertices f*| <= *4*
shows ($\sum v \in V \sum f \in$ *filter P* (*facesAt g v*) (*w*::*face* $\Rightarrow$ *nat*) *f*)
= ($\sum f \in$ [*f*←*faces g* . $\exists v \in$ *set V*. *f* $\in$ *set* (*facesAt g v*) *Int Collect P*] *w f*)
proof −
have *s*: *separating* (*set V*) (λv. *set* (*facesAt g v*) *Int Collect P*)
by (*rule separated-separating*[*OF V-subset pS noex*])
moreover note *dist*
moreover from *pl V-subset*

have $\bigwedge v.\ v \in set\ V \Longrightarrow distinct\ (facesAt\ g\ v)$
by(*blast intro*:*mgp-dist-facesAt*[*OF inv-mgp*])
hence *v*: $\bigwedge v.\ v \in set\ V \Longrightarrow distinct\ (filter\ P\ (facesAt\ g\ v))$
by *simp*
moreover
have *distinct* [*f*←*faces g* . $\exists\, v \in set\ V.\ f \in set\ (facesAt\ g\ v)$ *Int Collect P*]
by (*intro distinct-filter minGraphProps11'*[*OF inv-mgp*[*OF pl*]])
moreover from *pl* **have** $\{x.\ x \in \mathcal{F}\ g \wedge (\exists\, v \in set\ V.\ x \in set\ (facesAt\ g\ v) \wedge P\ x)\} =$
$(\bigcup v{\in}set\ V.\ set\ (facesAt\ g\ v)\ Int\ Collect\ P)$ **using** *V-subset*
by (*blast intro*:*minGraphProps inv-mgp*)
moreover from *v* **have** $(\sum v{\in}set\ V.\ ListSum\ (filter\ P\ (facesAt\ g\ v))\ w) = (\sum v{\in}set\ V.\ setsum\ w\ (set(facesAt\ g\ v)\ Int\ Collect\ P))$
by (*auto simp add*: *ListSum-conv-setsum Int-def*)
ultimately show *?thesis*
by (*simp add*: *ListSum-conv-setsum setsum-disj-Union*)
qed

lemma *separated-disj-Union2*:
assumes *pl*: *inv g* **and** *fin*: *final g* **and** *ne*: *noExceptionals g* (*set V*)
and *pS*: *separated g* (*set V*) **and** *dist*: *distinct V*
and *V-subset*: *set V* ⊆ *set* (*vertices g*)
shows $(\sum v \in V \sum f \in facesAt\ g\ v\ (w{::}face \Rightarrow nat)\ f)$
$= (\sum f \in [f{\leftarrow}faces\ g\ .\ \exists\, v \in set\ V.\ f \in set\ (facesAt\ g\ v)]\ w\ f)$
proof −
let *?P* = $\lambda f.\ |vertices\ f| <= 4$
have *ALL v* : *set V*. *ALL f* : *set* (*facesAt g v*). $|vertices\ f| <= 4$
using *V-subset ne*
by (*auto simp*: *noExceptionals-def*
intro: *minGraphProps5*[*OF inv-mgp*[*OF pl*]] *not-exceptional*[*OF pl fin*])
thus *?thesis*
using *ListSum-V-F-eq-ListSum-F*[**where** *P* = *?P*, *OF pl pS dist V-subset*]
by (*simp add*: *Int-def cong*: *conj-cong*)
qed

lemma *squanderFace-distr2*: *inv g* ⟹ *final g* ⟹ *noExceptionals g* (*set V*) ⟹
separated g (*set V*) ⟹ *distinct V* ⟹ *set V* ⊆ *set* (*vertices g*) ⟹
$(\sum f \in [f{\leftarrow}faces\ g.\ \exists\, v \in set\ V.\ f \in set\ (facesAt\ g\ v)]$
$\mathrm{d}\ |vertices\ f|\ )$
$= (\sum v \in V\ ((tri\ g\ v) * \mathrm{d}\ 3 + (quad\ g\ v) * \mathrm{d}\ 4))$
proof −
assume *pl*: *inv g*
assume *fin*: *final g*
assume *ne*: *noExceptionals g* (*set V*)
assume *separated g* (*set V*) *distinct V* **and** *V-subset*: *set V* ⊆ *set* (*vertices g*)
with *pl ne fin* **have**
$(\sum f \in [f{\leftarrow}faces\ g.\ \exists\, v{\in}set\ V.\ f{\in}set\ (facesAt\ g\ v)]\ \mathrm{d}\ |vertices\ f|\ )$
$= (\sum v \in V \sum f \in facesAt\ g\ v\ \mathrm{d}\ |vertices\ f|\ )$

```
    by (simp add: separated-disj-Union2)
  also have ⋀v. v ∈ set V ⟹
    (∑f ∈ facesAt g v. d |vertices f| )
  = (tri g v) * d 3 + (quad g v) * d 4
  proof −
    fix v assume v1: v ∈ set V
    with V-subset have v: v ∈ 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] v])
      with ne f v1 pl fin v 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 *separated-subset*:
$V1 \subseteq V2 \Longrightarrow$ *separated g V2* $\Longrightarrow$ *separated g V1*
proof (*simp add*: *separated-def separated*$_3$*-def separated*$_2$*-def*,
elim conjE, *intro allI impI ballI conjI*)
fix *v f*
assume *a*: $v \in V1$ $V1 \subseteq V2$ $f \in$ *set* (*facesAt g v*)
$|$*vertices f*$| \leq 4$
$\forall v{\in}V2.\ \forall f{\in}$*set* (*facesAt g v*). $|$*vertices f*$| \leq 4 \longrightarrow$
set (*vertices f*) $\cap$ *V2* $= \{v\}$
then show *set* (*vertices f*) $\cap$ *V1* $= \{v\}$ **by** *auto*
next
fix *v f*
assume *a*: $v \in V1$ $V1 \subseteq V2$ $f \in$ *set* (*facesAt g v*)
$\forall v{\in}V2.\ \forall f{\in}$*set* (*facesAt g v*). $f \cdot v \notin V2$
then have $v \in V2$ **by** *auto*
with *a* **have** $f \cdot v \notin V2$ **by** *auto*
with *a* **show** $f \cdot v \notin V1$ **by** *auto*
qed

end

22 Correctness of Lower Bound for Final Graphs

theory *LowerBound*
imports *PlaneProps ScoreProps*
begin

theorem *total-weight-lowerbound*:
inv g $\Longrightarrow$ *final g* $\Longrightarrow$ *tame g* $\Longrightarrow$ *admissible w g* $\Longrightarrow$
$(\sum_{f \in \textit{faces } g} w\ f) <$ *squanderTarget* $\Longrightarrow$
squanderLowerBound g $\leq (\sum_{f \in \textit{faces } g} w\ f)$
proof −
assume *final*: *final g* **and** *tame*: *tame g* **and** *pl*: *inv g*
assume *admissible*: *admissible w g*
assume *w*: $(\sum_{f \in \textit{faces } g} w\ f) <$ *squanderTarget*

have *squanderLowerBound g*
$=$ *ExcessNotAt g None* $+$ *faceSquanderLowerBound g*
by (*simp add*: *squanderLowerBound-def*)

We expand the definition of *faceSquanderLowerBound*.

also have *faceSquanderLowerBound g* $= (\sum_{f \in \textit{faces } g} \mathrm{d}\ |\textit{vertices } f|\ )$

We expand the definition of *ExcessNotAt*.

also from *ExcessNotAt-eq*[*OF pl*[*THEN inv-mgp*] *final*] **obtain** *V*
where *eq*: *ExcessNotAt g None* = $(\sum v \in V$ *ExcessAt g v*)
and *pS*: *separated g* (*set V*)
and *V-subset*: *set V* $\subseteq$ *set*(*vertices g*)
and *V-distinct*: *distinct V*

23 Properties of Tame Graph Enumeration (1)

theory *GeneratorProps*
imports *Plane1Props Generator TameProps LowerBound*
begin

lemma *genPolyTame-spec*:
generatePolygonTame n v f g = [$g' \leftarrow$ *generatePolygon n v f g* . ¬ *notame g'*]
by(*simp add*:*generatePolygonTame-def generatePolygon-def enum-enumerator*)

lemma *genPolyTame-subset-genPoly*:
$g' \in$ *set*(*generatePolygonTame i v f g*) $\Longrightarrow$
$g' \in$ *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*) $\subseteq$ *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*:
⟦$g' \in$ *set* (*generatePolygon n v f g*); *minGraphProps g*; $f \in$ *set* (*nonFinals g*);
$v \in \mathcal{V}$ *f*; *n* $\geq$ *3* ⟧ $\Longrightarrow$
$\exists f \in$ *set*(*finals g'*) − *set*(*finals g*). |*vertices f*| = *n*
apply(*auto simp add*:*generatePolygon-def image-def*)
apply(*rename-tac is*)
apply(*frule enumerator-length2*)
apply *arith*
apply(*frule* (*4*) *pre-subdivFace-indexToVertexList*)
apply(*arith*)
apply(*subgoal-tac indexToVertexList f v is* ≠ [])
prefer *2* **apply**(*subst length-0-conv*[*symmetric*]) **apply** *simp*
apply(*simp add*: *subdivFace-subdivFace'-eq*)
apply(*clarsimp simp*:*neq-Nil-conv*)
apply(*rename-tac ovs*)
apply(*subgoal-tac* |*indexToVertexList f v is*| = |*ovs*| + *1*)
prefer *2* **apply**(*simp*)
apply(*drule* (*1*) *pre-subdivFace-pre-subdivFace'*)
apply(*drule* (*1*) *final-subdivFace'*)
apply(*simp add*:*nonFinals-def*)

apply(*simp add*:*pre-subdivFace'-def*)
apply (*simp* (*no-asm-use*))
apply(*simp*)
apply *blast*
done

lemma *genPoly-incr-facesquander-lb*:
assumes $g' \in$ *set* (*generatePolygon n v f g*) *inv g*
$f \in$ *set*(*nonFinals g*) $v \in \mathcal{V}$ *f 3* $\leq n$
shows *faceSquanderLowerBound* $g' \geq$ *faceSquanderLowerBound g* + d *n*
proof −
from *genPoly-new-face*[*OF assms*(*1*) *inv-mgp*[*OF assms*(*2*)] *assms*(*3*−*5*)] **obtain** *f*
where *f*: $f \in$ *set* (*finals g'*) − *set*(*finals g*)
and *size*: |*vertices f*| = *n* **by** *auto*
have *g'*: $g' \in$ *set*(*next-plane0* (*n* − *3*) *g*) **using** *assms*(*5*)
by(*rule-tac in-next-plane0I*[*OF assms*(*1*,*3*−*5*)]) *simp*
note *dist* = *minGraphProps11'*[*OF inv-mgp*[*OF assms*(*2*)]]
note *inv'* = *invariantE*[*OF inv-inv-next-plane0*, *OF g' assms*(*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*| =
($\sum f \in set(finals\ g)$. d |*vertices f*|) + d |*vertices f*|
using *dist* **by**(*simp add*:*finals-def ListSum-conv-setsum*)
also have ... = ($\sum f \in set(finals\ g) \cup \{f\}$. d |*vertices f*|)
using *f* **by** *simp*
also have ... ≤ ($\sum f \in set(finals\ g')$. d |*vertices f*|)
using *f subset* **by**(*fastforce 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

definition *close* :: *graph* ⇒ *vertex* ⇒ *vertex* ⇒ *bool* **where**
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** *ug*: $u : \mathcal{V}\ g$ **and** *cl*: *close g u v*
shows *close g v u*
proof −
  obtain *f* **where** *f*: $f \in set(facesAt\ g\ u)$ **and**
    if: *if* $|vertices\ f| = 4$ *then* $v = f \cdot u \vee v = f \cdot (f \cdot u)$ *else* $v = f \cdot u$
    using *cl* **by** (*unfold close-def*) *blast*
  note *uf* = *minGraphProps6*[*OF mgp ug f*]
  note *distf* = *minGraphProps3*[*OF mgp minGraphProps5*[*OF mgp ug f*]]
  show *?thesis*
  proof *cases*
    assume *4*: $|vertices\ f| = 4$
    hence $v = f \cdot u \vee v = f \cdot (f \cdot u)$ **using** *if* **by** *simp*
    thus *?thesis*
    proof
      assume $v = f \cdot u$
      then obtain f' **where** $f' \in set(facesAt\ g\ v)\ f' \cdot v = u$
        using *mgp-nextVertex-face-ex2*[*OF mgp ug f*] **by** *blast*
      thus *?thesis* **by**(*auto simp*:*close-def*)
    next
      assume *v*: $v = f \cdot (f \cdot u)$
      hence $f \cdot (f \cdot v) = u$ **using** *quad-next4-id*[*OF 4 distf uf*] **by** *simp*
      moreover have $f \in set(facesAt\ g\ v)$ **using** *v uf*
        by(*simp add*: *minGraphProps7*[*OF mgp minGraphProps5*[*OF mgp ug f*]])
      ultimately show *?thesis* **using** *4* **by**(*auto simp*:*close-def*)
    qed
  next
    assume $|vertices\ f| \neq 4$
    hence $v = f \cdot u$ **using** *if* **by** *simp*
    then obtain f' **where** $f' \in set(facesAt\ g\ v)\ f' \cdot v = u$
      using *mgp-nextVertex-face-ex2*[*OF mgp ug f*] **by** *blast*
    thus *?thesis* **by**(*auto simp*:*close-def*)
  qed
qed

lemma *sep-conv*:
assumes *mgp*: *minGraphProps g* **and** $V <= \mathcal{V}\ g$
shows *separated g V* = $(\forall u{\in}V.\forall v{\in}V.\ u \neq v \longrightarrow \neg\ close\ g\ u\ v)$ (**is** *?P* = *?Q*)
proof
  assume *sep*: *?P*
  show *?Q*
  proof(*clarify*)
    fix *u v* **assume** *uv*: $u \in V\ v \in V\ u \neq v$ **and** *cl*: *close g u v*
    from *cl* **obtain** *f* **where** *f*: $f \in set(facesAt\ g\ u)$ **and**
      if: *if* $|vertices\ f| = 4$ *then* $(v = f \cdot u) \vee (v = f \cdot (f \cdot u))$

```
                    else (v = f · u)
    by (unfold close-def) blast
  have u : 𝒱 g using ‹u : V› ‹V <= 𝒱 g› by blast
  note uf = minGraphProps6[OF mgp ‹u : 𝒱 g› 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 sep f uv
        by(simp add:separated-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 sep f uv ‹u ∈ 𝒱 f›
        apply(unfold separated-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 sep f uv
      by(simp add:separated-def separated₂-def separated₃-def)
  qed
 qed
next
 assume not-cl: ?Q
 show ?P
 proof(simp add:separated-def, rule conjI)
  show separated₂ g V
  proof (clarsimp simp:separated₂-def)
    fix v f assume a: v ∈ V f ∈ set (facesAt g v) f · v ∈ V
    have v : 𝒱 g using a(1) ‹V <= 𝒱 g› by blast
    show False using a not-cl mgp-facesAt-no-loop[OF mgp ‹v : 𝒱 g› a(2)]
      by(fastforce 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
    have vg: v : 𝒱 g using ‹v : V› ‹V <= 𝒱 g› by blast
    note distf = minGraphProps3[OF mgp minGraphProps5[OF mgp vg f]]
    note vf = minGraphProps6[OF mgp vg 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 vg 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 vg f]] by
arith
    hence 𝒱 f = {v, f · v, f · (f · v), f · (f · (f · v))}
      using vertices-quad[OF - vf distf] by simp
    moreover
    { assume u = f · v
      hence u = v
        using not-cl f ‹u ∈ V› ‹v ∈ V› 4
        by(force simp:close-def split: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 vg 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 sep-ne: ∃ P ⊆ M. separated g (fst ‘ P)
by(unfold separated-def separated2-def separated3-def) blast

lemma ExcessNotAtRec-conv-Max:
assumes mgp: minGraphProps g
shows set(map fst ps) <= 𝒱 g ⟹ distinct(map fst ps) ⟹
  ExcessNotAtRec ps g =
  Max{ ∑p∈P. snd p |P. P ⊆ set ps ∧ separated g (fst ‘ P)}
  (is - ⟹ - ⟹ - = Max(?M ps) is - ⟹ - ⟹ - = Max{- |P. ?S ps P})
proof(induct ps rule: length-induct)
  case (1 ps0)
  note IH = 1(1) and subset = 1(2) and dist = 1(3)
  show ?case
  proof (cases ps0)
    case Nil thus ?thesis by(simp 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 fst p : 𝒱 g and fst ‘ set ps <= 𝒱 g
      using subset Cons by auto
    have sub1: !!P Q. P <= {x : set ps. Q x} ⟹ fst ‘ P <= 𝒱 g
      using subset Cons by auto
    have sub2: !!P Q. P <= insert p {x : set ps. Q x} ⟹ fst ‘ P <= 𝒱 g
```

using *subset Cons* **by** *auto*
have *sub3*: !!*P*. *P* <= *insert p* (*set ps*) ⟹ *fst* ' *P* <= $\mathcal{V}$ *g*
using *subset Cons* **by** *auto*
have !!*a*. *set* (*map fst* (*deleteAround g a ps*)) ⊆ $\mathcal{V}$ *g*
using *deleteAround-subset*[*of g - ps*] *subset Cons*
by *auto*
hence *ExcessNotAtRec ps0 g* = *max* (*Max*(*?M ps*)) (*snd p* + *Max*(*?M ?ps*))
using *Cons IH subset le dist dist'* **by** (*cases p*) *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 sep-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(*fastforce intro*!: *setsum-insert*[*THEN trans*,*symmetric*] *elim*: *finite-subset*)
apply(*rule-tac x*=*P* **in** *exI*)
apply(*fastforce intro*!: *setsum-insert*[*THEN trans*] *elim*: *finite-subset*)
done
also have ... = {*setsum snd P* |*P*.
P ⊆ *insert p* (*set ?ps*) ∧ *p* ∈ *P* ∧ *separated g* (*fst* ' *P*)}
apply(*auto simp add*:*sep-conv*[*OF mgp*] *sub1 sub2 delAround-def cong*: *conj-cong*)
apply(*rule-tac x* = *insert p P* **in** *exI*)
apply *simp*
apply(*rule conjI*) **apply** *blast*
using ‹*image fst* (*set ps*) <= $\mathcal{V}$ *g*› ‹*fst p* : $\mathcal{V}$ *g*›
apply (*blast intro*:*close-sym*[*OF mgp*])
apply(*rule-tac x* = *P*−{*p*} **in** *exI*)
apply (*simp add*:*insert-absorb*)
apply *blast*
done
also have ... = {*setsum snd P* |*P*.
P ⊆ *insert p* (*set ps*) ∧ *p* ∈ *P* ∧ *separated g* (*fst* ' *P*)}
using *Cons dist*
apply(*auto simp add*:*sep-conv*[*OF mgp*] *sub2 sub3 delAround-def cong*: *conj-cong*)
apply(*rule-tac x* = *P* **in** *exI*)
apply *simp*
apply *auto*
done
also have *max* (*Max*(*?M ps*)) (*Max* ...) = *Max*(*?M ps* ∪ {*setsum snd P* |*P*.
P ⊆ *insert p* (*set ps*) ∧ *p* ∈ *P* ∧ *separated g* (*fst* ' *P*)})
(**is** - = *Max ?U*)
proof −
have {*setsum snd P* |*P*.
P ⊆ *insert p* (*set ps*) ∧ *p* ∈ *P* ∧ *separated g* (*fst* ' *P*)} ≠ {}

apply *simp*
apply(*rule-tac x*={*p*} **in** *exI*)
using ‹*fst p* : $\mathcal{V}$ *g*› **by**(*simp add*:*sep-conv*[*OF mgp*])
thus *?thesis* **by**(*simp add*: *Max-Un fin-aux sep-ne*)
qed
also have *?U* = *?M ps0* **using** *Cons* **by** *simp blast*
finally show *?thesis* **.**
qed
qed

lemma *dist-ExcessTab*: *distinct* (*map fst* (*ExcessTable g* (*vertices g*)))
by(*simp add*:*ExcessTable-def vertices-graph o-def*)

lemma *mono-ExcessTab*: ⟦*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 *fst-set-ExcessTable-subset*:
fst ‘ *set* (*ExcessTable g* (*vertices g*)) ⊆ $\mathcal{V}$ *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*]
fst-set-ExcessTable-subset)
apply(*rule Max-mono*)
prefer *2* **apply** (*simp add*: *sep-ne*)
prefer *2* **apply** (*simp add*: *fin-aux*)
apply *auto*
apply(*rule-tac x*=*P* **in** *exI*)
apply *auto*
apply(*subgoal-tac fst* ‘ *P* <= $\mathcal{V}$ *g'*)
prefer *2* **apply** (*blast dest*: *ExcessTab-vertex*)
apply(*subgoal-tac fst* ‘ *P* <= $\mathcal{V}$ *g*)
prefer *2* **apply** (*blast dest*: *ExcessTab-vertex*)
apply(*simp add*:*sep-conv*)
apply (*blast intro*:*close-antimono ExcessTab-final ExcessTab-vertex*)
done

lemma *next-plane0-incr-squander-lb*:
⟦*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*; *notame7 g*⟧
⟹ *notame7 g'*
apply(*simp add*:*notame-def notame7-def tame11b-def is-tame13a-def tame10ub-def*
del:*disj-not1*)
apply(*frule inv-mgp*)
apply(*frule* (*1*) *next-plane0-vertices-subset*)
apply(*erule disjE*)
apply(*simp add*:*vertices-graph*)
apply(*rule disjI2*)
apply(*erule disjE*)
apply *clarify*
apply(*frule* (*2*) *next-plane0-incr-degree*)
apply(*frule* (*2*) *next-plane0-incr-except*)
apply (*force split*:*split-if-asm*)
apply(*frule* (*1*) *next-plane0-incr-squander-lb*)
apply(*arith*)
done

lemma *inv-inv-notame*:

invariant(λ*g*. *inv g* ∧ *notame7 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 (λ*g*. *inv g* ∧ *notame7 g*)
proof(*clarsimp simp add*: *notame-def notame7-def untame-def tame11b-def is-tame13a-def tame10ub-def*
linorder-not-le linorder-not-less)
fix *g* **assume** *final g inv g tame g*
and *cases*: *15* < *countVertices g* ∨
(∃ *v*∈*V g*. (*except g v* = *0* ⟶ *7* < *degree g v*) ∧
(*0* < *except g v* ⟶ *6* < *degree g v*))
∨ *squanderTarget* ≤ *squanderLowerBound g*
(**is** *?A* ∨ *?B* ∨ *?C* **is** - ∨ (∃ *v*∈*V g*. *?B′ v*) ∨ -)
from *cases* **show** *False*
proof(*elim disjE*)
assume *?B*
then obtain *v* **where** *v*: *v* ∈*V g ?B′ v* **by** *auto*
show *False*
proof *cases*
assume *except g v* = *0*
thus *False* **using** ‹*tame g*› *v* **by**(*auto simp*: *tame-def tame11b-def*)
next
assume *except g v* ≠ *0*
thus *False* **using** ‹*tame g*› *v*
by(*auto simp*: *except-def filter-empty-conv tame-def tame11b-def*
minGraphProps-facesAt-eq[*OF inv-mgp*[*OF* ‹*inv g*›]] *split*:*split-if-asm*)
qed
next
assume *?A*
thus *False* **using** ‹*tame g*› **by**(*simp add*:*tame-def tame10-def*)
next
assume *?C*
thus *False* **using** *total-weight-lowerbound*[*OF* ‹*inv g*› ‹*final g*› ‹*tame g*›]
‹*tame g*› **by**(*force simp add*:*tame-def tame13a-def*)
qed
qed

lemma *polysizes-tame*:
⟦ *g′* ∈ *set* (*generatePolygon n v f g*); *inv g*; *f* ∈ *set*(*nonFinals g*);
v ∈ *V f*; *3* ≤ *n*; *n* < *4*+*p*; *n* ∉ *set*(*polysizes p g*) ⟧
⟹ *notame7 g′*
apply(*frule* (*4*) *in-next-plane0I*)
apply(*frule* (*4*) *genPoly-incr-facesquander-lb*)

apply(*frule* (*1*) *next-plane0-incr-ExcessNotAt*)
apply(*simp add*: *notame-def notame7-def is-tame13a-def faceSquanderLowerBound-def*
polysizes-def squanderLowerBound-def)
done

lemma *genPolyTame-notame*:
⟦ $g' \in$ *set* (*generatePolygon n v f g*); $g' \notin$ *set* (*generatePolygonTame n v f g*);
inv g; $3 \leq n$ ⟧
⟹ *notame7 g'*
by(*fastforce simp*:*generatePolygon-def generatePolygonTame-def enum-enumerator*
notame-def notame7-def)

declare *upt-Suc*[*simp del*]
lemma *excess-notame*:
⟦ *inv g*; $g' \in$ *set* (*next-plane*$_p$ *g*); $g' \notin$ *set* (*next-tame0 p g*) ⟧
⟹ *notame7 g'*
apply(*frule* (*1*) *mgp-next-plane0-if-next-plane*[*OF inv-mgp*])
apply(*auto simp add*:*next-tame0-def next-plane-def split*:*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*)) ∈ $\mathcal{V}$(*minimalFace*
(*nonFinals g*)))
apply(*blast intro*:*polysizes-tame*)
apply(*simp add*:*minimalVertex-def*)
apply(*rule minimal-in-set*)
apply(*erule mgp-vertices-nonempty*[*OF inv-mgp*])
apply(*simp add*:*nonFinals-def*)
done
declare *upt-Suc*[*simp*]

lemma *next-tame0-comp*: ⟦ *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 *inv-inv-next-tame0*: *invariant inv* (*next-tame0 p*)
by(*rule inv-subset*[*OF inv-inv-next-plane next-tame0-subset-plane*])

lemma *inv-inv-next-tame*: *invariant inv next-tame*$_p$
apply(*simp add*:*next-tame-def*)
apply(*rule inv-subset*[*OF inv-inv-next-tame0*])
apply *auto*
done

lemma *mgp-TameEnum*: $g \in$ *TameEnum*$_p \Longrightarrow$ *minGraphProps g*
by (*unfold TameEnumP-def*)
(*blast intro*: *RTranCl-inv*[*OF inv-inv-next-tame*] *inv-Seed inv-mgp*)

end

24 Properties of Tame Graph Enumeration (2)

theory *TameEnumProps*
imports *GeneratorProps*
begin

Completeness of filter for final graphs.

lemma *untame-negFin*:
assumes *pl*: *inv g* **and** *fin*: *final g* **and** *tame*: *tame g*
shows *is-tame g*
proof (*unfold is-tame-def*, *intro conjI*)
show *tame10 g* **using** *tame* **by**(*auto simp*:*tame-def*)
next
show *tame11a g* **using** *tame* **by**(*auto simp*:*tame-def*)
next
show *tame12o g* **using** *tame* **by**(*auto simp*:*tame-def*)
next
next
from *tame* **obtain** *w* **where** *adm*: *admissible w g*
and *sqn*: $(\sum_{f \in \mathit{faces}\ g} w\, f) <$ *squanderTarget* **by**(*auto simp*:*tame-def tame13a-def*)
moreover have *squanderLowerBound g* $\leq$ $(\sum_{f \in \mathit{faces}\ g} w\, f)$
using *pl fin tame adm sqn* **by** (*rule total-weight-lowerbound*)
ultimately show *is-tame13a g* **by**(*auto simp*:*is-tame13a-def*)
qed

lemma *next-tame-comp*:
$[\![$ *tame g*; *final g*; *Seed*$_p$ [*next-tame0 p*]$\rightarrow *$ *g* $]\!]$
$\Longrightarrow$ *Seed*$_p$ [*next-tame*$_p$]$\rightarrow *$ *g*
apply (*unfold next-tame-def*)
apply(*rule filter-tame-succs*[*OF inv-inv-next-tame0*])
apply(*simp add*:*next-tame0-def finalGraph-def*)
apply(*rule context-conjI*)

```
    apply(simp)
   apply(blast dest:untame-negFin)
  apply assumption
 apply(rule inv-Seed)
apply assumption+
done


end

theory Worklist
imports ~~/src/HOL/Library/While-Combinator RTranCl Quasi-Order
begin

definition
  worklist-aux :: ('s ⇒ 'a ⇒ 'a list) ⇒ ('a ⇒ 's ⇒ 's)
    ⇒ 'a list * 's ⇒ ('a list * 's)option
where
worklist-aux succs f =
 while-option
   (%(ws,s). ws ≠ [])
   (%(ws,s). case ws of x#ws' ⇒ (succs s x @ ws', f x s))

definition worklist :: ('s ⇒ 'a ⇒ 'a list) ⇒ ('a ⇒ 's ⇒ 's)
    ⇒ 'a list ⇒ 's ⇒ 's option where
worklist succs f ws s =
  (case worklist-aux succs f (ws,s) of
     None ⇒ None | Some(ws,s) ⇒ Some s)

lemma worklist-aux-Nil: worklist-aux succs f ([],s) = Some([],s)
by(simp add: worklist-aux-def while-option-unfold)

lemma worklist-aux-Cons:
 worklist-aux succs f (x#ws',s) = worklist-aux succs f (succs s x @ ws', f x s)
by(simp add: worklist-aux-def while-option-unfold)

lemma worklist-aux-unfold[code]:
 worklist-aux succs f (ws,s) =
 (case ws of [] ⇒ Some([],s)
  | x#ws' ⇒ worklist-aux succs f (succs s x @ ws', f x s))
by(simp add: worklist-aux-Nil worklist-aux-Cons split: list.split)

definition
  worklist-tree-aux :: ('a ⇒ 'a list) ⇒ ('a ⇒ 's ⇒ 's)
    ⇒ 'a list * 's ⇒ ('a list * 's)option
where
worklist-tree-aux succs = worklist-aux (%s. succs)

lemma worklist-tree-aux-unfold[code]:
```

worklist-tree-aux succs f (*ws*,*s*) =
(*case ws of* [] ⇒ *Some*([],*s*) |
x#*ws'* ⇒ *worklist-tree-aux succs f* (*succs x* @ *ws'*, *f x s*))
by(*simp add*: *worklist-tree-aux-def worklist-aux-Nil worklist-aux-Cons split*: *list.split*)

abbreviation *Rel* :: ('*a* ⇒ '*a list*) ⇒ ('*a* * '*a*)*set* **where**
Rel f == {(*x*,*y*). *y* : *set*(*f x*)}

lemma *Image-Rel-set*:
(*Rel succs*)^* `` *set*(*succs x*) = (*Rel succs*)^+ `` {*x*}
by(*auto simp add*: *trancl-unfold-left*)

lemma *RTranCl-conv*:
g [*succs*]→* *h* ⟷ (*g*,*h*) : ((*Rel succs*)^*) (**is** *?L* = *?R*)
proof−
have *?L* ⟹ *?R*
apply(*erule RTranCl-induct*)
apply *blast*
apply (*auto elim*: *rtrancl-into-rtrancl*)
done
moreover
have *?R* ⟹ *?L*
apply(*erule converse-rtrancl-induct*)
apply(*rule RTranCl.refl*)
apply (*auto elim*: *RTranCl.succs*)
done
ultimately show *?thesis* **by** *blast*
qed

lemma *worklist-end-empty*:
worklist-aux succs f (*ws*,*s*) = *Some*(*ws'*,*s'*) ⟹ *ws'* = []
unfolding *worklist-aux-def*
by (*drule while-option-stop*) *simp*

theorem *worklist-tree-aux-Some-foldl*:
assumes *worklist-tree-aux succs f* (*ws*,*s*) = *Some*(*ws'*,*s'*)
shows *EX rs*. *set rs* = ((*Rel succs*)^*) `` (*set ws*) &
s' = *foldl* (%*s x*. *f x s*) *s rs*
proof −
let *?b* = %(*ws*,*s*). *ws* ≠ []
let *?c* = %(*ws*,*s*). *case ws of x*#*ws'* ⇒ (*succs x* @ *ws'*, *f x s*)
let *?Q* = % *ws' s' done*.
s' = *foldl* (%*x s*. *f s x*) *s done* &
((*Rel succs*)^*) `` (*set ws*) =
set done Un ((*Rel succs*)^*) `` *set ws'*
let *?P* = %(*ws*,*s*). *EX done*. *?Q ws s done*
have *0*: *while-option ?b ?c* (*ws*,*s*) = *Some*(*ws'*,*s'*)
using *assms* **by**(*simp add*: *worklist-tree-aux-def worklist-aux-def*)

```
  from while-option-stop[OF 0] have ws' = [] by simp
  have init: ?P (ws,s)
    apply auto
    apply(rule-tac x = [] in exI)
    apply simp
    done
  { fix ws s
    assume ?P (ws,s)
    then obtain done where ?Q ws s done by blast
    assume ?b(ws,s)
    then obtain x ws' where ws = x # ws' by(auto simp: neq-Nil-conv)
    then have ?Q (succs x @ ws') (f x s) (done @ [x])
      using ‹?Q ws s done›
      apply simp
      apply(erule thin-rl)+
      apply (auto simp add: Image-Un Image-Rel-set)
      apply (blast elim: rtranclE intro: rtrancl-into-trancl1)
      done
    hence ?P(?c(ws,s)) using ‹ws=x#ws'›
      by(simp only: split-conv list.cases) blast
  }
  then have ?P(ws',s')
    using while-option-rule[where P=?P, OF - 0 init]
    by auto
  then show ?thesis using ‹ws'=[]› by auto
qed

definition worklist-tree succs f ws s =
  (case worklist-tree-aux succs f (ws,s) of
     None ⇒ None | Some(ws,s) ⇒ Some s)

theorem worklist-tree-Some-foldl:
  worklist-tree succs f ws s = Some s' ⟹
   EX rs. set rs = ((Rel succs)^*) `` (set ws) &
             s' = foldl (%s x. f x s) s rs
by(simp add: worklist-tree-def worklist-tree-aux-Some-foldl split:option.splits prod.splits)

lemma invariant-succs:
assumes invariant I succs
and ALL x:S. I x
shows ALL x: (Rel succs)^* `` S. I x
proof−
  { fix x y have (x,y) : (Rel succs)^* ⟹ I x ⟹ I y
    proof(induct rule:rtrancl-induct)
      case base thus ?case .
    next
      case step with assms(1) show ?case by(auto simp:invariant-def)
    qed
  } with assms(2) show ?thesis by blast
```

qed

lemma *worklist-tree-aux-rule*:
assumes *worklist-tree-aux succs f* (*ws*,*s*) = *Some*(*ws'*,*s'*)
and *invariant I succs*
and *ALL x* : *set ws*. *I x*
and !!*s*. *P* [] *s s*
and !!*r x ws s*. *I x* ⟹ ∀ *x* ∈ *set ws*. *I x* ⟹ *P ws* (*f x s*) *r* ⟹ *P* (*x*#*ws*) *s r*
shows ∃ *rs*. *set rs* = ((*Rel succs*) ^∗) `` (*set ws*) ∧ *P rs s s'*
proof−
let *?R* = (*Rel succs*)∗ `` *set ws*
from *worklist-tree-aux-Some-foldl*[*OF assms*(*1*)] **obtain** *rs* **where**
rs: *set rs* = *?R s'* = *foldl* (%*s x*. *f x s*) *s rs* **by** *blast*
{ fix *xs* **have** (*ALL x* : *set xs*. *I x*) ⟹ *P xs s* (*foldl* (%*s x*. *f x s*) *s xs*)
proof(*induct xs arbitrary*: *s*)
case *Nil* **show** *?case* **using** *assms*(*4*) **by** *simp*
next
case *Cons* **thus** *?case* **using** *assms*(*5*) **by** *simp*
qed
}
with *invariant-succs*[*OF assms*(*2*,*3*)] **show** *?thesis* **by** (*metis rs*)
qed

lemma *worklist-tree-aux-rule2*:
assumes *worklist-tree-aux succs f* (*ws*,*s*) = *Some*(*ws'*,*s'*)
and *invariant I succs*
and *ALL x* : *set ws*. *I x*
and *S s* **and** !!*x s*. *I x* ⟹ *S s* ⟹ *S*(*f x s*)
and !!*s*. *P* [] *s s*
and !!*r x ws s*. *I x* ⟹ ∀ *x* ∈ *set ws*. *I x* ⟹ *S s*
⟹ *P ws* (*f x s*) *r* ⟹ *P* (*x*#*ws*) *s r*
shows ∃ *rs*. *set rs* = ((*Rel succs*) ^∗) `` (*set ws*) ∧ *P rs s s'*
proof−
let *?R* = (*Rel succs*)∗ `` *set ws*
from *worklist-tree-aux-Some-foldl*[*OF assms*(*1*)] **obtain** *rs* **where**
rs: *set rs* = *?R s'* = *foldl* (%*s x*. *f x s*) *s rs* **by** *blast*
{ fix *xs* **have** (*ALL x* : *set xs*. *I x*) ⟹ *S s* ⟹ *P xs s* (*foldl* (%*s x*. *f x s*) *s xs*)
proof(*induct xs arbitrary*: *s*)
case *Nil* **show** *?case* **using** *assms*(*6*) **by** *simp*
next
case *Cons* **thus** *?case* **using** *assms*(*5*,*7*) **by** *simp*
qed
}
with *invariant-succs*[*OF assms*(*2*,*3*)] *assms*(*4*) **show** *?thesis* **by** (*metis rs*)
qed

lemma *worklist-tree-rule*:
assumes *worklist-tree succs f ws s* = *Some*(*s'*)
and *invariant I succs*

and *ALL x : set ws. I x*
and *!!s. P [] s s*
and *!!r x ws s. I x ⟹ ∀x ∈ set ws. I x ⟹ P ws (f x s) r ⟹ P (x#ws) s r*
shows *EX rs. set rs = ((Rel succs)^*) `` (set ws) ∧ P rs s s'*
proof−
obtain *ws'* **where** *worklist-tree-aux succs f (ws,s) = Some(ws',s')* **using** *assms(1)*
by(*simp add*: *worklist-tree-def split*: *option.split-asm prod.split-asm*)
from *worklist-tree-aux-rule*[**where** *P=P,OF this assms(2−5)*] **show** *?thesis* **by** *blast*
qed

lemma *worklist-tree-rule2*:
assumes *worklist-tree succs f ws s = Some(s')*
and *invariant I succs*
and *ALL x : set ws. I x*
and *S s* **and** *!!x s. I x ⟹ S s ⟹ S(f x s)*
and *!!s. P [] s s*
and *!!r x ws s. I x ⟹ ∀x ∈ set ws. I x ⟹ S s*
⟹ P ws (f x s) r ⟹ P (x#ws) s r
shows *EX rs. set rs = ((Rel succs)^*) `` (set ws) ∧ P rs s s'*
proof−
obtain *ws'* **where** *worklist-tree-aux succs f (ws,s) = Some(ws',s')* **using** *assms(1)*
by(*simp add*: *worklist-tree-def split*: *option.split-asm prod.split-asm*)
from *worklist-tree-aux-rule2*[**where** *P=P* **and** *S=S,OF this assms(2−7)*]
show *?thesis* **by** *blast*
qed

lemma *worklist-tree-aux-state-inv*:
assumes *worklist-tree-aux succs f (ws,s) = Some(ws',s')*
and *I s*
and *!!x s. I s ⟹ I(f x s)*
shows *I s'*
proof−
from *worklist-tree-aux-rule*[**where** *P=%ws s s'. I s ⟶ I s'* **and** *I=%x. True*,
OF assms(1)] *assms(2−3)*
show *?thesis* **by**(*auto simp*: *invariant-def*)
qed

lemma *worklist-tree-state-inv*:
worklist-tree succs f ws s = Some(s')
⟹ I s ⟹ (!!x s. I s ⟹ I(f x s)) ⟹ I s'
unfolding *worklist-tree-def*
by(*auto intro*: *worklist-tree-aux-state-inv split*: *option.splits*)

locale *set-modulo = quasi-order +*
fixes *empty :: 's*
and *insert-mod :: 'a => 's => 's*
and *set-of :: 's => 'a set*

and $I :: 'a \Rightarrow bool$
and $S :: 's \Rightarrow bool$
assumes *set-of-empty*: $set\text{-}of\ empty = \{\}$
and *set-of-insert-mod*: $I\ x \Longrightarrow S\ s \wedge (\forall x \in set\text{-}of\ s.\ I\ x)$
$\Longrightarrow$
$set\text{-}of\ (insert\text{-}mod\ x\ s) = insert\ x\ (set\text{-}of\ s) \vee$
$(EX\ y : set\text{-}of\ s.\ x \preceq y) \wedge set\text{-}of\ (insert\text{-}mod\ x\ s) = set\text{-}of\ s$
and *S-empty*: $S\ empty$
and *S-insert-mod*: $S\ s \Longrightarrow S\ (insert\text{-}mod\ x\ s)$
begin

definition *insert-mod2* :: $('b \Rightarrow bool) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'b \Rightarrow 's \Rightarrow 's$ **where**
$insert\text{-}mod2\ P\ f\ x\ s = (if\ P\ x\ then\ insert\text{-}mod\ (f\ x)\ s\ else\ s)$

definition $SI\ s = (S\ s \wedge (\forall x \in set\text{-}of\ s.\ I\ x))$

lemma *SI-empty*: $SI\ empty$
by(*simp add*: *SI-def S-empty set-of-empty*)

lemma *SI-insert-mod*:
$I\ x \Longrightarrow SI\ s \Longrightarrow SI\ (insert\text{-}mod\ x\ s)$
apply(*simp add*: *SI-def S-insert-mod*)
by (*metis insertE set-of-insert-mod*)

lemma *SI-insert-mod2*: $(!!x.\ inv0\ x \Longrightarrow I\ (f\ x)) \Longrightarrow$
$inv0\ x \Longrightarrow SI\ s \Longrightarrow SI\ (insert\text{-}mod2\ P\ f\ x\ s)$
by (*metis insert-mod2-def SI-insert-mod*)

definition *worklist-tree-coll-aux* ::
$('b \Rightarrow 'b\ list) \Rightarrow ('b \Rightarrow bool) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'b\ list \Rightarrow 's \Rightarrow 's\ option$
where
$worklist\text{-}tree\text{-}coll\text{-}aux\ succs\ P\ f = worklist\text{-}tree\ succs\ (insert\text{-}mod2\ P\ f)$

definition *worklist-tree-coll* ::
$('b \Rightarrow 'b\ list) \Rightarrow ('b \Rightarrow bool) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'b\ list \Rightarrow 's\ option$
where
$worklist\text{-}tree\text{-}coll\ succs\ P\ f\ ws = worklist\text{-}tree\text{-}coll\text{-}aux\ succs\ P\ f\ ws\ empty$

lemma *worklist-tree-coll-aux-equiv*:
assumes $worklist\text{-}tree\text{-}coll\text{-}aux\ succs\ P\ f\ ws\ s = Some\ s'$
and *invariant inv0 succs*
and $\forall x \in set\ ws.\ inv0\ x$
and $!!x.\ inv0\ x \Longrightarrow I(f\ x)$
and $SI\ s$
shows $set\text{-}of\ s' =_{\preceq}$
$f\ `\ \{x : (Rel\ succs)\hat{}*\ ``\ (set\ ws).\ P\ x\} \cup set\text{-}of\ s$
apply(*insert assms*(*1*))
unfolding *worklist-tree-coll-aux-def*
apply(*drule worklist-tree-rule2*[**where** $I = inv0$ **and** $S = SI$ **and**

$P = \%ws\ s\ s'.\ SI\ s \longrightarrow set\text{-}of\ s' =_{\preceq} f\ `\ \{x : set\ ws.\ P\ x\}\ Un\ set\text{-}of\ s,$
$OF\ -\ assms(2,3,5)]$)
apply(*simp add*: *SI-insert-mod2 assms*(*4*))
apply(*clarsimp*)
apply(*clarsimp simp add*: *insert-mod2-def split*: *split-if-asm*)
apply(*frule assms*(*4*))
apply(*frule SI-def*[*THEN iffD1*])
apply(*frule* (*1*) *set-of-insert-mod*)
apply (*simp add*: *SI-insert-mod*)
apply(*erule disjE*)
prefer *2*
apply(*rule seteq-qle-trans*)
apply *assumption*
apply (*simp add*: *defs*)
apply *blast*
apply(*rule seteq-qle-trans*)
apply *assumption*
apply (*simp add*: *defs*)
apply *blast*
apply(*rule seteq-qle-trans*)
apply *assumption*
apply (*simp add*: *defs*)
apply *blast*
using *assms*(*5*)
apply *auto*
done

lemma *worklist-tree-coll-equiv*:
worklist-tree-coll succs P f ws = *Some s'* $\Longrightarrow$ *invariant inv0 succs*
$\Longrightarrow \forall x \in set\ ws.\ inv0\ x \Longrightarrow (!!x.\ inv0\ x \Longrightarrow I(f\ x))$
$\Longrightarrow set\text{-}of\ s' =_{\preceq} f\ `\ \{x : (Rel\ succs)\ \hat{}*\ ``\ (set\ ws).\ P\ x\}$
unfolding *worklist-tree-coll-def*
apply(*drule* (*2*) *worklist-tree-coll-aux-equiv*)
apply(*auto simp*: *set-of-empty SI-empty*)
done

lemma *worklist-tree-coll-aux-subseteq*:
worklist-tree-coll-aux succs P f ws t_0 = *Some t* $\Longrightarrow$
invariant inv0 succs $\Longrightarrow$ *ALL g* : *set ws. inv0 g* $\Longrightarrow$
$(!!x.\ inv0\ x \Longrightarrow I(f\ x)) \Longrightarrow SI\ t_0 \Longrightarrow$
$set\text{-}of\ t \subseteq set\text{-}of\ t_0 \cup f\ `\ \{h : (Rel\ succs)\ \hat{}*\ ``\ set\ ws.\ P\ h\}$
unfolding *worklist-tree-coll-aux-def*
apply(*drule worklist-tree-rule2*[**where** I = *inv0* **and** S = *SI* **and** P =
$\%ws\ t\ t'.\ set\text{-}of\ t' \subseteq set\text{-}of\ t \cup f\ `\ \{g \in set\ ws.\ P\ g\}$])
apply *assumption*
apply *assumption*
apply *assumption*
apply(*simp add*: *SI-insert-mod2*)
apply *clarsimp*

apply (*clarsimp simp*: *insert-mod2-def split*: *split-if-asm*)
using *set-of-insert-mod*
apply(*simp add*: *SI-def image-def*)
apply(*blast*)
apply *blast*
apply *blast*
done

lemma *worklist-tree-coll-subseteq*:
worklist-tree-coll succs P f ws = *Some t* ⟹
invariant inv0 succs ⟹ *ALL g* : *set ws*. *inv0 g* ⟹
(!!*x*. *inv0 x* ⟹ *I*(*f x*)) ⟹
set-of t ⊆ *f* ‘ {*h* : (*Rel succs*) ^* ‘‘ *set ws*. *P h*}
unfolding *worklist-tree-coll-def*
apply(*drule* (*1*) *worklist-tree-coll-aux-subseteq*)
apply(*auto simp*: *set-of-empty SI-empty*)
done

lemma *worklist-tree-coll-inv*:
worklist-tree-coll succs P f ws = *Some s* ⟹ *S s*
unfolding *worklist-tree-coll-def worklist-tree-coll-aux-def*
apply(*drule worklist-tree-state-inv*[**where** *I* = *S*])
apply (*auto simp*: *S-empty insert-mod2-def S-insert-mod*)
done

end

end

theory *Maps*
imports *Worklist Quasi-Order*
begin

locale *maps* =
fixes *empty* :: *'m*
and *up* :: *'m* ⇒ *'a* ⇒ *'b list* ⇒ *'m*
and *map-of* :: *'m* ⇒ *'a* ⇒ *'b list*
and *M* :: *'m* ⇒ *bool*
assumes *map-empty*: *map-of empty* = (%*a*. [])
and *map-up*: *map-of* (*up m a b*) = (*map-of m*)(*a* := *b*)
and *M-empty*: *M empty*
and *M-up*: *M m* ⟹ *M* (*up m a b*)
begin

definition *set-of m* = (*UN x*. *set*(*map-of m x*))

end

locale *set-mod-maps* = *maps empty up map-of M* + *quasi-order qle*

for *empty* :: *'m*
and *up* :: *'m* ⇒ *'a* ⇒ *'b list* ⇒ *'m*
and *map-of* :: *'m* ⇒ *'a* ⇒ *'b list*
and *M* :: *'m* ⇒ *bool*
and *qle* :: *'b* ⇒ *'b* ⇒ *bool* (**infix** ⪯ *60*)
+
fixes *subsumed* :: *'b* ⇒ *'b* ⇒ *bool*
and *I* :: *'b* ⇒ *bool*
and *key* :: *'b* ⇒ *'a*
assumes *equiv-iff-qle*: *I x* ⟹ *I y* ⟹ *subsumed x y* = (*x* ⪯ *y*)
and *key*=*key*
begin

definition *insert-mod x m* =
(*let k* = *key x*; *ys* = *map-of m k*
in if (*EX y* : *set ys*. *subsumed x y*) *then m else up m k* (*x*#*ys*))

end

sublocale
set-mod-maps <
set-by-maps: *set-modulo qle empty insert-mod set-of I M*
proof
case *goal1* **show** *?case* **by**(*simp add*:*set-of-def map-empty*)
next
case *goal2* **thus** *?case*
by (*auto simp*: *Let-def insert-mod-def set-of-def map-up equiv-iff-qle*
split:*split-if-asm*)
next
case *goal3* **show** *?case* **by**(*simp add*: *M-empty*)
next
case *goal4* **thus** *?case*
by(*simp add*: *insert-mod-def Let-def M-up*)
qed

end

25 Tries (List Version)

theory *Tries*
imports *Maps*
begin

25.1 Association lists

primrec *rem-alist* :: *'key* ⇒ (*'key* ∗ *'val*)*list* ⇒ (*'key* ∗ *'val*)*list* **where**
rem-alist k [] = [] |
rem-alist k (*p*#*ps*) = (*if fst p* = *k then ps else p* # *rem-alist k ps*)

lemma *rem-alist-id*[*simp*]: $k \notin fst\ `\ set\ al \Longrightarrow rem\text{-}alist\ k\ al = al$
by(*induct al*, *auto*)

lemma *set-rem-alist*:
$distinct(map\ fst\ al) \Longrightarrow set\ (rem\text{-}alist\ a\ al) =$
$(set\ al) - \{(a,the(map\text{-}of\ al\ a))\}$
apply(*induct al*)
apply *simp*
apply *simp*
apply *fastforce*
done

lemma *fst-set-rem-alist*[*simp*]:
$distinct(map\ fst\ al) \Longrightarrow fst\ `\ set\ (rem\text{-}alist\ a\ al) = fst\ `\ (set\ al) - \{a\}$
apply(*induct al*)
apply *simp*
apply *simp*
apply *blast*
done

lemma *distinct-map-fst-rem-alist*[*simp*]:
$distinct\ (map\ fst\ al) \Longrightarrow distinct\ (map\ fst\ (rem\text{-}alist\ a\ al))$
by(*induct al*) *simp-all*

lemma *map-of-rem-distinct-alist*: $distinct(map\ fst\ al) \Longrightarrow$
$map\text{-}of(rem\text{-}alist\ k\ al) = (map\text{-}of\ al)(k := None)$
apply(*induct al*)
apply *simp*
apply *clarify*
apply(*rule ext*)
apply *auto*
apply(*erule ssubst*)
apply *simp*
done

lemma *map-of-rem-alist*[*simp*]:
$k' \neq k \Longrightarrow map\text{-}of\ (rem\text{-}alist\ k\ al)\ k' = map\text{-}of\ al\ k'$
by(*induct al*, *auto*)

25.2 Tries

datatype $('a,'v)tries = Tries\ \ 'v\ list\ \ ('a * ('a,'v)tries)list$

primrec $values :: ('a,'v)tries \Rightarrow 'v\ list$ **where**
$values(Tries\ vs\ al) = vs$

primrec $alist :: ('a,'v)tries \Rightarrow ('a * ('a,'v)tries)list$ **where**
$alist(Tries\ vs\ al) = al$

```
fun inv :: ('a,'v)tries ⇒ bool where
inv(Tries - al) = (distinct(map fst al) & (∀(a,t) ∈ set al. inv t))

primrec lookup :: ('a,'v)tries ⇒ 'a list ⇒ 'v list where
lookup t [] = values t |
lookup t (a#as) = (case map-of (alist t) a of
                     None ⇒ []
                   | Some at ⇒ lookup at as)

primrec update :: ('a,'v)tries ⇒ 'a list ⇒ 'v list ⇒ ('a,'v)tries where
update t []     vs = Tries vs (alist t) |
update t (a#as) vs =
  (let tt = (case map-of (alist t) a of
               None ⇒ Tries [] [] | Some at ⇒ at)
   in Tries (values t) ((a,update tt as vs) # rem-alist a (alist t)))

primrec insert :: ('a,'v)tries ⇒ 'a list ⇒ 'v      ⇒ ('a,'v)tries where
insert t []     v = Tries (v # values t) (alist t) |
insert t (a#as) vs =
  (let tt = (case map-of (alist t) a of
               None ⇒ Tries [] [] | Some at ⇒ at)
   in Tries (values t) ((a,insert tt as vs) # rem-alist a (alist t)))


lemma lookup-empty[simp]: lookup (Tries [] []) as = []
by(case-tac as, simp-all)

theorem lookup-update:
 lookup (update t as vs) bs =
 (if as=bs then vs else lookup t bs)
apply(induct as arbitrary: t v bs)
 apply clarsimp
 apply(case-tac bs)
  apply simp
 apply simp
apply clarsimp
apply(case-tac bs)
 apply (auto simp add:Let-def split:option.split)
done

theorem insert-conv:
 insert t as v = update t as (v#lookup t as)
apply(induct as arbitrary: t)
apply (auto simp:Let-def split:option.split)
done

lemma inv-insert: inv t ⟹ inv(insert t as v)
apply(induct as arbitrary: t)
```

apply(*case-tac t*)
apply *simp*
apply(*case-tac t*)
apply *simp*
apply (*auto simp*:*set-rem-alist split*: *option.split*)
done

lemma *inv-update*: *inv t* $\Longrightarrow$ *inv*(*update t as v*)
apply(*induct as arbitrary*: *t*)
apply(*case-tac t*)
apply *simp*
apply(*case-tac t*)
apply *simp*
apply (*auto simp*:*set-rem-alist split*: *option.split*)
done

definition *trie-of-list* :: ('*b* $\Rightarrow$ '*a list*) $\Rightarrow$ '*b list* $\Rightarrow$ ('*a*,'*b*)*tries* **where**
trie-of-list key = *foldl* (%*t v*. *insert t* (*key v*) *v*) (*Tries* [] [])

lemma *inv-foldl-insert*:
inv t $\Longrightarrow$ *inv* (*foldl* (%*t v*. *insert t* (*key v*) *v*) *t xs*)
apply(*induct xs arbitrary*: *t*)
apply(*auto simp*: *inv-insert*)
done

lemma *inv-of-list*: *inv* (*trie-of-list k xs*)
unfolding *trie-of-list-def*
apply(*rule inv-foldl-insert*)
apply *simp*
done

lemma *in-set-lookup-of-list*:
v $\in$ *set*(*lookup* (*trie-of-list key vs*) (*key v*)) = (*v* $\in$ *set vs*)
proof −
have !!*t*.
v $\in$ *set*(*lookup* (*foldl* (%*t v*. *insert t* (*key v*) *v*) *t vs*) (*key v*)) =
(*v* $\in$ *set vs* $\cup$ *set*(*lookup t* (*key v*)))
apply(*induct vs*)
apply (*simp add*:*trie-of-list-def*)
apply (*simp add*:*trie-of-list-def*)
apply(*simp add*:*insert-conv lookup-update*)
apply *blast*
done
thus *?thesis* **by**(*simp add*:*trie-of-list-def*)
qed

lemma *in-set-lookup-of-listD*:
assumes *v* $\in$ *set*(*lookup* (*trie-of-list f vs*) *xs*) **shows** *v* $\in$ *set vs*
proof −

have !!t.
$v \in set(lookup\ (foldl\ (\%t\ v.\ insert\ t\ (f\ v)\ v)\ t\ vs)\ xs) \Longrightarrow$
$v \in set\ vs \cup set(lookup\ t\ xs)$
apply$(induct\ vs)$
apply $(simp\ add{:}trie\text{-}of\text{-}list\text{-}def)$
apply $(simp\ add{:}trie\text{-}of\text{-}list\text{-}def)$
apply$(force\ simp{:}insert\text{-}conv\ lookup\text{-}update\ split{:}split\text{-}if\text{-}asm)$
done
thus *?thesis* **using** *assms* **by**$(force\ simp\ add{:}trie\text{-}of\text{-}list\text{-}def)$
qed

definition $set\text{-}of :: ('a,'b)tries \Rightarrow 'b\ set$ **where**
$set\text{-}of\ t = Union\ \{gs.\ \exists\, a.\ gs = set(lookup\ t\ a)\}$

lemma $set\text{-}of\text{-}empty[simp]$: $set\text{-}of\ (Tries\ []\ [])\ =\ \{\}$
by$(simp\ add{:}\ set\text{-}of\text{-}def)$

lemma $set\text{-}of\text{-}insert[simp]$:
$set\text{-}of\ (insert\ t\ a\ x)\ =\ Set.insert\ x\ (set\text{-}of\ t)$
by$(auto\ simp{:}\ set\text{-}of\text{-}def\ insert\text{-}conv\ lookup\text{-}update)$

lemma $set\text{-}of\text{-}foldl\text{-}insert$:
$set\text{-}of\ (foldl\ (\%t\ v.\ insert\ t\ (key\ v)\ v)\ t\ xs)\ =$
$set\ xs\ Un\ set\text{-}of\ t$
by$(induct\ xs\ arbitrary{:}\ t)\ auto$

lemma $set\text{-}of\text{-}of\text{-}list[simp]$:
$set\text{-}of(trie\text{-}of\text{-}list\ key\ xs)\ =\ set\ xs$
by$(simp\ add{:}\ trie\text{-}of\text{-}list\text{-}def\ set\text{-}of\text{-}foldl\text{-}insert)$

lemma $in\text{-}set\text{-}lookup\text{-}set\text{-}ofD$:
$x{\in}set\ (lookup\ t\ a) \Longrightarrow x \in set\text{-}of\ t$
by$(auto\ simp{:}\ set\text{-}of\text{-}def)$

fun $all :: ('v \Rightarrow bool) \Rightarrow ('a,'v)tries \Rightarrow bool$ **where**
$all\ P\ (Tries\ vs\ al)\ =$
$((\forall\, v \in set\ vs.\ P\ v) \wedge (\forall\, (a,t) \in set\ al.\ all\ P\ t))$

interpretation map:
$maps\ Tries\ []\ []\ update\ lookup\ inv$
proof
case *goal1* **show** *?case* **by**$(rule\ ext)\ simp$
next
case *goal2* **show** *?case* **by**$(rule\ ext)\ (simp\ add{:}lookup\text{-}update)$
next
case *goal3* **show** *?case* **by**$(simp)$
next
case *goal4* **thus** *?case* **by**$(simp\ add{:}inv\text{-}update)$
qed

lemma *set-of-conv*: *set-of* = *maps.set-of lookup*
by(*rule ext*) (*auto simp add*: *set-of-def map.set-of-def*)

hide-const (**open**) *inv lookup update insert set-of all*

end

26 Archive

theory *Arch*
imports *Main* ~~/*src/HOL/Library/Code-Target-Numeral*
begin

setup ⟪ *fn thy* =>
  let
    val T = @{*typ integer list list list*}
    val dir = *Thy-Load.master-directory thy*
  in
    thy |>
    Code-Runtime.polyml-as-definition
      [(@{*binding Tri'*}, *T*), (@{*binding Quad'*}, *T*), (@{*binding Pent'*}, *T*),
       (@{*binding Hex'*}, *T*)]
      (*map* (*Path.append dir o Path.explode*)
        [*Archives/Tri.ML*, *Archives/Quad.ML*,
          Archives/Pent.ML, *Archives/Hex.ML*])
  end
⟫

The definition of these constants is only ever needed at the ML level when running the eval proof method.

definition *Tri* :: *nat list list list*
where
  Tri = (*map* ∘ *map* ∘ *map*) *nat-of-integer Tri'*

definition *Quad* :: *nat list list list*
where
  Quad = (*map* ∘ *map* ∘ *map*) *nat-of-integer Quad'*

definition *Pent* :: *nat list list list*
where
  Pent = (*map* ∘ *map* ∘ *map*) *nat-of-integer Pent'*

definition *Hex* :: *nat list list list*
where
  Hex = (*map* ∘ *map* ∘ *map*) *nat-of-integer Hex'*

end

27 Comparing Enumeration and Archive

theory *ArchCompAux*
imports *TameEnum Tries Arch Worklist*
begin

function *qsort* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a list* ⇒ *'a list* **where**
qsort le [] = [] |
qsort le (*x*#*xs*) = *qsort le* [*y*←*xs* . ~ *le x y*] @ [*x*] @
qsort le [*y*←*xs* . *le x y*]
by *pat-completeness auto*
termination by (*relation measure* (*size o snd*))
(*auto simp add*: *length-filter-le* [*THEN le-less-trans*])

definition *nof-vertices* :: *'a fgraph* ⇒ *nat* **where**
nof-vertices = *length o remdups o concat*

definition *fgraph* :: *graph* ⇒ *nat fgraph* **where**
fgraph g = *map vertices* (*faces g*)

definition *hash* :: *nat fgraph* ⇒ *nat list* **where**
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* ∈ *set f*]))
[*0*..<*n*]))

definition *samet* :: (*nat*,*nat fgraph*) *tries option* ⇒ *nat fgraph list* ⇒ *bool*
where
samet fgto ags = (*case fgto of None* ⇒ *False* | *Some tfgs* ⇒
let tags = *trie-of-list hash ags in*
(*Tries.all* (%*fg*. *list-ex* (*iso-test fg*) (*Tries.lookup tags* (*hash fg*))) *tfgs* &
Tries.all (%*ag*. *list-ex* (*iso-test ag*) (*Tries.lookup tfgs* (*hash ag*))) *tags*))

definition *pre-iso-test* :: *vertex fgraph* ⇒ *bool* **where**
pre-iso-test Fs ⟷
[] ∉ *set Fs* ∧ (∀ *F*∈*set Fs*. *distinct F*) ∧ *distinct* (*map rotate-min Fs*)

end

28 Completeness of Archive Test

theory *ArchCompProps*

imports *TameEnumProps ArchCompAux*
begin
lemma *mgp-pre-iso-test*: *minGraphProps g* $\Longrightarrow$ *pre-iso-test*(*fgraph g*)
apply(*simp add*:*pre-iso-test-def fgraph-def image-def*)
apply(*rule conjI*) **apply**(*blast dest*: *mgp-vertices-nonempty*[*symmetric*])
apply(*rule conjI*) **apply**(*blast intro*:*minGraphProps*)
apply(*drule minGraphProps11*)
apply(*simp add*:*normFaces-def normFace-def verticesFrom-def minVertex-def*
rotate-min-def o-def)
done

corollary *iso-test-correct*:
$[\![$ *pre-iso-test* Fs_1; *pre-iso-test* Fs_2 $]\!] \Longrightarrow$
iso-test Fs_1 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 *trie-all-eq-set-of-trie*:
Tries.inv t $\Longrightarrow$ *Tries.all P t* = ($\forall$ *v* $\in$ *Tries.set-of t*. *P v*)
apply(*induct t rule*:*Tries.inv.induct*)
apply (*auto simp*: *Tries.set-of-def*)
apply(*case-tac a*)
apply *simp*
apply *auto*
apply *blast*
apply(*erule allE*)
apply(*erule impE*)
apply(*rule-tac x* = [] **in** *exI*)
apply(*rule HOL.refl*)
apply *simp*
apply(*erule meta-allE*)+
apply(*erule meta-impE*)
apply *assumption*
apply(*erule meta-impE*)
apply *fast*
apply(*erule meta-impE*)
apply *fast*
apply *clarsimp*
apply(*erule allE*)
apply(*erule impE*)
apply(*rule-tac x* = *a*#*aa* **in** *exI*)
apply(*rule HOL.refl*)
apply *auto*
done

lemma *samet-imp-iso-seteq*:
assumes *pre1*: $\bigwedge$*gs g*. *gsopt* = *Some gs* $\Longrightarrow$ *g* $\in$ *Tries.set-of gs* $\Longrightarrow$ *pre-iso-test g*
and *pre2*: $\bigwedge$*g*. *g* $\in$ *set arch* $\Longrightarrow$ *pre-iso-test g*

and *inv*: !!*gs*. *gsopt* = *Some gs* ⟹ *Tries.inv gs*
and *same*: *samet gsopt arch*
shows ∃ *gs*. *gsopt* = *Some gs* ∧ *Tries.set-of gs* =≃ *set arch*
proof −
obtain *gs* **where** [*simp*]: *gsopt* = *Some gs* **and** *test1*: ⋀*g*. *g* ∈ *Tries.set-of gs* ⟹
∃ *h* ∈ *set arch*. *iso-test g h* **and** *test2*: ⋀*g*. *g* ∈ *set arch* ⟹
∃ *h* ∈ *Tries.set-of gs*. *iso-test g h*
using *same inv*
by(*force simp*: *samet-def trie-all-eq-set-of-trie inv-of-list*
split:*option.splits*
dest: *in-set-lookup-of-listD in-set-lookup-set-ofD*)
have *Tries.set-of gs* ⊆≃ *set arch*
proof (*auto simp*:*qle-gr.defs*)
fix *g* **assume** *g*: *g* ∈ *Tries.set-of gs*
obtain *h* **where** *h*: *h* ∈ *set arch* **and** *test*: *iso-test g h*
using *test1*[*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
moreover
have *set arch* ⊆≃ *Tries.set-of gs*
proof (*auto simp*:*qle-gr.defs*)
fix *g* **assume** *g*: *g* ∈ *set arch*
obtain *h* **where** *h*: *h* ∈ *Tries.set-of gs* **and** *test*: *iso-test g h*
using *test2*[*OF g*] **by** *blast*
thus ∃ *h* ∈ *Tries.set-of gs*. *g* ≃ *h*
using *h pre1*[*OF - h*] *pre2*[*OF g*] **by** (*auto simp*:*iso-test-correct*)
qed
ultimately show *?thesis* **by** (*auto simp*: *qle-gr.seteq-qle-def*)
qed

lemma *samet-imp-iso-subseteq*:
assumes *pre1*: ⋀*gs g*. *gsopt* = *Some gs* ⟹ *g* ∈ *Tries.set-of gs* ⟹ *pre-iso-test g*
and *pre2*: ⋀*g*. *g* ∈ *set arch* ⟹ *pre-iso-test g*
and *inv*: !!*gs*. *gsopt* = *Some gs* ⟹ *Tries.inv gs*
and *same*: *samet gsopt arch*
shows ∃ *gs*. *gsopt* = *Some gs* ∧ *Tries.set-of gs* ⊆≃ *set arch*
using *qle-gr.seteq-qle-def assms samet-imp-iso-seteq* **by** *metis*

definition [*code del*]:
insert-mod-trie = *set-mod-maps.insert-mod Tries.update Tries.lookup iso-test hash*
definition [*code del*]:
worklist-tree-coll-trie = *set-modulo.worklist-tree-coll* (*Tries* [] []) *insert-mod-trie*
definition [*code del*]:
worklist-tree-coll-aux-trie = *set-modulo.worklist-tree-coll-aux insert-mod-trie*
definition [*code del*]:
insert-mod2-trie = *set-modulo.insert-mod2 insert-mod-trie*

interpretation *set-mod-trie*:
set-mod-maps Tries [] [] Tries.update Tries.lookup Tries.inv op $\simeq$ *iso-test pre-iso-test hash*
where *set-modulo.worklist-tree-coll* (*Tries [] []*) *insert-mod-trie* = *worklist-tree-coll-trie*
and *set-modulo.worklist-tree-coll-aux insert-mod-trie* = *worklist-tree-coll-aux-trie*
and *set-mod-maps.insert-mod Tries.update Tries.lookup iso-test hash* = *insert-mod-trie*
and *set-modulo.insert-mod2 insert-mod-trie* = *insert-mod2-trie*
proof *unfold-locales*
qed (*auto simp*:*iso-test-correct worklist-tree-coll-trie-def worklist-tree-coll-aux-trie-def insert-mod-trie-def insert-mod2-trie-def*)

definition *enum-filter-finals* ::
(*graph* $\Rightarrow$ *graph list*) $\Rightarrow$ *graph list*
$\Rightarrow$ (*nat*,*nat fgraph*) *tries option* **where**
enum-filter-finals succs = *set-mod-trie.worklist-tree-coll succs final fgraph*

definition *tameEnumFilter* :: *nat* $\Rightarrow$ (*nat*,*nat fgraph*)*tries option* **where**
tameEnumFilter p = *enum-filter-finals* (*next-tame p*) [*Seed p*]

lemma *TameEnum-tameEnumFilter*:
tameEnumFilter p = *Some t* $\Longrightarrow$ *Tries.set-of t* $=_\simeq$ *fgraph* ‘ *TameEnum*$_p$
apply(*auto simp*: *tameEnumFilter-def TameEnumP-def enum-filter-finals-def*)
apply(*drule set-mod-trie.worklist-tree-coll-equiv*[*OF - inv-inv-next-tame*])
apply (*auto simp*: *Tries.set-of-conv inv-Seed mgp-pre-iso-test RTranCl-conv*)
done

lemma *tameEnumFilter-subseteq-TameEnum*:
tameEnumFilter p = *Some t* $\Longrightarrow$ *Tries.set-of t* <= *fgraph* ‘ *TameEnum*$_p$
by(*auto simp add*:*tameEnumFilter-def TameEnumP-def enum-filter-finals-def*
Tries.set-of-conv inv-Seed mgp-pre-iso-test RTranCl-conv
dest!: *set-mod-trie.worklist-tree-coll-subseteq*[*OF - inv-inv-next-tame*])

lemma *inv-tries-tameEnumFilter*:
tameEnumFilter p = *Some t* $\Longrightarrow$ *Tries.inv t*
unfolding *tameEnumFilter-def enum-filter-finals-def*
by(*erule set-mod-trie.worklist-tree-coll-inv*)

theorem *combine-evals-filter*:
$\forall$ *g* $\in$ *set arch*. *pre-iso-test g* $\Longrightarrow$ *samet* (*tameEnumFilter p*) *arch*
$\Longrightarrow$ *fgraph* ‘ *TameEnum*$_p$ $\subseteq_\simeq$ *set arch*
apply(*subgoal-tac* $\exists$ *t*. *tameEnumFilter p* = *Some t* $\wedge$ *Tries.set-of t* $\subseteq_\simeq$ *set arch*)
apply(*metis TameEnum-tameEnumFilter qle-gr.seteq-qle-def qle-gr.subseteq-qle-trans*)
apply(*fastforce intro*!: *samet-imp-iso-subseteq*
dest: *inv-tries-tameEnumFilter tameEnumFilter-subseteq-TameEnum mgp-TameEnum mgp-pre-iso-test*)
done

end

29 Comparing Enumeration and Archive

theory *ArchComp*
imports *ArchCompProps* ~~/*src*/*HOL*/*Library*/*Code-Target-Numeral*
begin

method-setup *cond-eval* = ⟪
  Scan.succeed (*fn ctxt* =>
    SIMPLE-METHOD'
     (*if getenv ISABELLE-FULL-TEST* = *true then eval-tac ctxt*
      else SELECT-GOAL (*Skip-Proof.cheat-tac* (*Proof-Context.theory-of ctxt*))))
⟫ *solve goal by evaluation if ISABELLE-FULL-TEST=true*)

29.1 Proofs by evaluation using generated code

lemma *pre-iso-test3*: ∀ *g* ∈ *set Tri*. *pre-iso-test g*
by *eval*

lemma *pre-iso-test4*: ∀ *g* ∈ *set Quad*. *pre-iso-test g*
by *eval*

lemma *pre-iso-test5*: ∀ *g* ∈ *set Pent*. *pre-iso-test g*
by *eval*

lemma *pre-iso-test6*: ∀ *g* ∈ *set Hex*. *pre-iso-test g*
by *eval*

lemma *same3*: *samet* (*tameEnumFilter 0*) *Tri*
by *eval*

lemma *same4*: *samet* (*tameEnumFilter 1*) *Quad*
by *cond-eval*

lemma *same5*: *samet* (*tameEnumFilter 2*) *Pent*
by *cond-eval*

lemma *same6*: *samet* (*tameEnumFilter 3*) *Hex*
by *cond-eval*

end

30 Combining All Completeness Proofs

theory *Completeness*
imports *ArchCompProps ArchComp*
begin

definition *Archive* :: *vertex fgraph set* **where**
Archive ≡ *set*(*Tri* @ *Quad* @ *Pent* @ *Hex*)

theorem *TameEnum-Archive*: *fgraph* ' *TameEnum* ⊆∼ *Archive*
using *combine-evals-filter*[*OF pre-iso-test3 same3*]
combine-evals-filter[*OF pre-iso-test4 same4*]
combine-evals-filter[*OF pre-iso-test5 same5*]
combine-evals-filter[*OF pre-iso-test6 same6*]
by(*fastforce simp*:*TameEnum-def Archive-def image-def qle-gr.defs*
eval-nat-numeral le-Suc-eq)

lemma *TameEnum-comp*:
assumes $Seed_p$ [*next-plane*$_p$]→∗ *g* **and** *final g* **and** *tame g*
shows $Seed_p$ [*next-tame*$_p$]→∗ *g*
proof −
from *assms* **have** $Seed_p$ [*next-tame0 p*]→∗ *g* **by**(*rule next-tame0-comp*)
with *assms* **show** $Seed_p$ [*next-tame*$_p$]→∗ *g* **by**(*blast intro*: *next-tame-comp*)
qed

lemma *tame5*:
assumes *g*: $Seed_p$ [*next-plane0*$_p$]→∗ *g* **and** *final g* **and** *tame g*
shows *p* ≤ *3*
proof −
from ‹*tame g*› **have** *bound*: ∀ *f* ∈ $\mathcal{F}$ *g*. |*vertices f*| ≤ *6*
by (*simp add*: *tame-def tame9a-def*)
show *p* ≤ *3*
proof (*rule ccontr*)
assume *6*: ¬ *p* ≤ *3*
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* ∈ $\mathcal{F}$ *g* & *6* < |*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 ≤ 3 by(blast intro:tame5)
  with g1 ⟨tame g⟩ ⟨final g⟩ show ?thesis using TameEnum-Archive
    by(simp add: qle-gr.defs TameEnum-def TameEnumP-def)
      (blast intro: TameEnum-comp)
qed

end
```

References

[1] Tobias Nipkow, Gertrud Bauer, and Paula Schultz. Flyspeck I: Tame graphs. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *LNCS*, pages 21–35. Springer, 2006.