Tame Plane Graphs

Gertrud Bauer and Tobias Nipkow

February 2, 2006

Contents

1 Basic Functions Old and New

theory *ListAux*
imports *Main EfficientNat*
begin

declare *Let-def*[*simp*]

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

1.1 HOL

lemma *pairD*: $(a,b) = p \Longrightarrow a = fst\ p \wedge b = snd\ p$
⟨*proof*⟩

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

lemma [*code unfold*]: $max\ x\ y == (let\ u = x;\ v = y\ in\ if\ u <= v\ then\ v\ else\ u)$
⟨*proof*⟩
lemma [*code unfold*]: $min\ x\ y == (let\ u = x;\ v = y\ in\ if\ u <= v\ then\ u\ else\ v)$
⟨*proof*⟩

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

1.2 Lists

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

1.2.1 *length*

syntax -*length* :: *'a list* ⇒ *nat* (|-|)
translations
$|xs| == length\ xs$

lemma *length3D*: $|xs| = 3 \Longrightarrow \exists\, x\ y\ z.\ xs = [x,\ y,\ z]$
⟨*proof*⟩

lemma *length4D*: $|xs| = 4 \Longrightarrow \exists\ a\ b\ c\ d.\ xs = [a,\ b,\ c,\ d]$
⟨*proof*⟩

1.2.2 *filter*

lemma *filter-emptyE*[*dest*]: $(filter\ P\ xs = []) \Longrightarrow\ x \in set\ xs \Longrightarrow\ \neg\ P\ x$
⟨*proof*⟩

lemma *filter-comm*: $[x \in xs.\ P\ x \wedge Q\ x] = [x \in xs.\ Q\ x \wedge P\ x]$
⟨*proof*⟩

lemma *filter1*: $[x{\in}xs\ .\ P\ x] = [] \Longrightarrow$
$[x{\in}\ xs.\ Q\ x \wedge P\ x] = []$
⟨*proof*⟩

lemma *filter-prop*: $\bigwedge x.\ x \in set\ [u{\in}ys\ .\ P\ u] \Longrightarrow P\ x$
⟨*proof*⟩

lemma *filter-Cons-prop*: $[u{\in}ys\ .\ P\ u] = x\ \#\ xs \Longrightarrow P\ x$
⟨*proof*⟩

lemma *filter-compl1*:
$([x \in xs.\ P\ x] = []) = ([x \in xs.\ \neg\ P\ x] = xs)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩
lemma [*simp*]: $Not \circ (Not \circ P) = P$
⟨*proof*⟩

lemma *filter-compl2*: $\bigwedge P.\ (filter\ (Not \circ P)\ xs = []) = (filter\ P\ xs = xs)$
⟨*proof*⟩

lemma *filter-eqI*:
$(\bigwedge v.\ v \in set\ vs \Longrightarrow P\ v = Q\ v) \Longrightarrow [v \in vs\ .\ P\ v] = [v \in vs\ .\ Q\ v]$
⟨*proof*⟩

lemma *filter-simp*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x) \Longrightarrow [x \in xs.\ P\ x \wedge Q\ x] = [x \in xs.\ Q\ x]$
⟨*proof*⟩

lemma *filter-True-eq1*:
$(length\ [y \in xs.\ P\ y] = length\ xs) \Longrightarrow (\bigwedge y.\ y \in set\ xs \Longrightarrow P\ y)$
⟨*proof*⟩

lemma *length-filter-True-eq*:
$(length\ [y \in xs.\ P\ y] = length\ xs) = (\forall\, y.\ y \in set\ xs \longrightarrow P\ y)$
⟨*proof*⟩

1.2.3 *map*

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

translations
$[f.\ x \in xs] == map\ (\lambda x.\ f)\ xs$
$[f.\ x : xs] == map\ (\lambda x.\ f)\ xs$

1.2.4 *map-filter*

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

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

translations
[*f*. *x* ∈ *xs*, *P*]== *map-filter* (λ*x*. *f*) *P xs*
[*f*. *x* : *xs*, *P*]== *map-filter* (λ*x*. *f*) *P xs*

lemma [*simp*]: [*f x*. *x* ∈ *xs*, *P*] = [*f x*. *x* ∈ [*x* ∈ *xs*. *P x*]]
⟨*proof*⟩

1.2.5 *concat*

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

1.2.6 List product

constdefs *listProd1* :: *'a* ⇒ *'b list* ⇒ (*'a* × *'b*) *list*
listProd1 a bs ≡ [(*a*,*b*). *b* ∈ *bs*]

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

lemma *set* (*xs* × *ys*) = (*set xs*) × (*set ys*)
⟨*proof*⟩

1.2.7 Minimum and maximum

consts *minimal*:: (*'a* ⇒ *nat*) ⇒ *'a list* ⇒ *'a*
primrec
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*
⟨*proof*⟩

lemma *minimal-Cons1*:
∀ *y* ∈ *set xs*. *f x* ≤ *f y* ⟹ *minimal f* (*x*#*xs*) = *x*
⟨*proof*⟩

lemma *minimal-append2*:
∀ *x* ∈ *set xs*. *f x* > *f y* ⟹ *minimal f* (*xs* @ *y* # *ys*) = *minimal f* (*y* # *ys*)
⟨*proof*⟩

lemma *minimal-neq-lowerbound*:
xs ≠ [] ⟹ *ALL x*: *set xs*. *f x* ≥ *n* ⟹ *f*(*minimal f xs*) ≠ *n*

$\Longrightarrow$ *ALL x: set xs. f x* $\neq$ *n*
⟨*proof*⟩

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

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

lemma *minList-conv-Min*[*simp*]:
xs ≠ [] $\Longrightarrow$ *minList xs* = *Min*(*set xs*)
⟨*proof*⟩

lemma *max-list-conv-Max*[*simp*]:
xs ≠ [] $\Longrightarrow$ *max-list xs* = *Max*(*set xs*)
⟨*proof*⟩

1.2.8 replace

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

consts *mapAt* :: *nat list* ⇒ (*'a* ⇒ *'a*) ⇒ (*'a list* ⇒ *'a list*)
primrec
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*
⟨*proof*⟩

lemma *length-replace1*[*simp*]: *length*(*replace x* [*y*] *xs*) = *length xs*
⟨*proof*⟩

lemma *replace-id*[*simp*]: *replace x* [*x*] *xs* = *xs*
⟨*proof*⟩

lemma *len-replace-ge-same*:
length ys ≥ *1* $\Longrightarrow$ *length*(*replace x ys xs*) ≥ *length xs*
⟨*proof*⟩

lemma *len-replace-ge*[*simp*]:
⟦ *length ys* ≥ *1*; *length xs* ≥ *length zs* ⟧ ⟹
length(*replace x ys xs*) ≥ *length zs*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *filter-replace*:
¬ *P y* ⟹ *filter P* (*replace x* [*y*] *xs*) = *remove1 x* (*filter P xs*)
⟨*proof*⟩

lemma *distinct-set-replace*: *distinct xs* ⟹
set (*replace x ys xs*) =
(*if x* ∈ *set xs then* (*set xs* − {*x*}) ∪ *set ys else set xs*)
⟨*proof*⟩

lemma *replace1*:
f ∈ *set* (*replace f' fs ls*) ⟹ *f* ∉ *set ls* ⟹ *f* ∈ *set fs*
⟨*proof*⟩

lemma *replace2*:
f' ∉ *set ls* ⟹ *replace f' fs ls* = *ls*
⟨*proof*⟩

lemma *replace3*[*intro*]:
f' ∈ *set ls* ⟹ *f* ∈ *set fs* ⟹ *f* ∈ *set* (*replace f' fs ls*)
⟨*proof*⟩

lemma *replace4*:
f ∈ *set ls* ⟹ *oldF* ≠ *f* ⟹ *f* ∈ *set* (*replace oldF fs ls*)
⟨*proof*⟩

lemma *replace5*: *f* ∈ *set* (*replace oldF newfs fs*) ⟹ *f* ∈ *set fs* ∨ *f* ∈ *set newfs*
⟨*proof*⟩

lemma *replace6*: *distinct oldfs* ⟹ *x* ∈ *set* (*replace oldF newfs oldfs*) =
((*x* ≠ *oldF* ∨ *oldF* ∈ *set newfs*) ∧ ((*oldF* ∈ *set oldfs* ∧ *x* ∈ *set newfs*) ∨ *x* ∈ *set oldfs*))
⟨*proof*⟩

lemma *replace-delete-oldF*:
oldF ∉ *set fs* ⟹ *distinct ls* ⟹ *oldF* ∉ *set* (*replace oldF fs ls*)
⟨*proof*⟩

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

lemma *replace-replace*[*simp*]: *oldf* $\notin$ *set newfs* $\Longrightarrow$ *distinct xs* $\Longrightarrow$
replace oldf newfs (*replace oldf newfs xs*) = *replace oldf newfs xs*
⟨*proof*⟩

lemma *replace-distinct*: *distinct fs* $\Longrightarrow$ *distinct newfs* $\Longrightarrow$ *oldf* $\in$ *set fs* $\longrightarrow$ *set newfs* $\cap$ *set fs* $\subseteq$ $\{oldf\}$ $\Longrightarrow$
distinct (*replace oldf newfs fs*)
⟨*proof*⟩

lemma *filter-replace2*:
$[\![\neg P\ x; \forall y \in set\ ys.\ \neg P\ y\ ]\!] \Longrightarrow$
filter P (*replace x ys xs*) = *filter P xs*
⟨*proof*⟩

lemma *length-filter-replace1*:
$[\![x \in set\ xs; \neg P\ x\ ]\!] \Longrightarrow$
length(*filter P* (*replace x ys xs*)) =
length(*filter P xs*) + *length*(*filter P ys*)
⟨*proof*⟩

lemma *length-filter-replace2*:
$[\![x \in set\ xs; P\ x\ ]\!] \Longrightarrow$
length(*filter P* (*replace x ys xs*)) =
length(*filter P xs*) + *length*(*filter P ys*) − *1*
⟨*proof*⟩

1.2.9 *distinct*

lemma *dist-filter-single*:
distinct ls $\Longrightarrow$ *v* $\in$ *set ls* $\Longrightarrow$ [*a*$\in$ *ls* . *a* = *v*] = [*v*]
⟨*proof*⟩

lemma *dist-at1*: $\bigwedge$ *c vs*. *distinct vs* $\Longrightarrow$ *vs* = *a* @ *r* # *b* $\Longrightarrow$ *vs* = *c* @ *r* # *d* $\Longrightarrow$ *a* = *c*
⟨*proof*⟩

lemma *dist-at*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *r* # *b* $\Longrightarrow$ *vs* = *c* @ *r* # *d* $\Longrightarrow$ *a* = *c* $\wedge$ *b* = *d*
⟨*proof*⟩

lemma *dist-at2*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *r* # *b* $\Longrightarrow$ *vs* = *c* @ *r* # *d* $\Longrightarrow$ *b* = *d*
⟨*proof*⟩

lemma *distinct-split1*: *distinct xs* $\Longrightarrow$ *xs* = *y* @ [*r*] @ *z* $\Longrightarrow$ *r* $\notin$ *set y*
⟨*proof*⟩

lemma *distinct-split2*: *distinct xs* $\Longrightarrow$ *xs* = *y* @ [*r*] @ *z* $\Longrightarrow$ *r* $\notin$ *set z* ⟨*proof*⟩

lemma *distinct-hd-not-cons*: *distinct vs* $\Longrightarrow$ $\exists$ *as bs*. *vs* = *as* @ *x* # *hd vs* # *bs*
$\Longrightarrow$ *False*
⟨*proof*⟩

1.2.10 Misc

lemma *drop-last-in*: !!*n*. *n* < *length ls* $\Longrightarrow$ *last ls* $\in$ *set* (*drop n ls*)
⟨*proof*⟩

lemma *nth-last-Suc-n*: *distinct ls* $\Longrightarrow$ *n* < *length ls* $\Longrightarrow$ *last ls* = *ls* ! *n* $\Longrightarrow$ *Suc*
n = *length ls*
⟨*proof*⟩

1.2.11 *rotate*

lemma *plus-length1*[*simp*]: *rotate* (*k*+(*length ls*)) *ls* = *rotate k ls*
⟨*proof*⟩

lemma *plus-length2*[*simp*]: *rotate* ((*length ls*)+*k*) *ls* = *rotate k ls*
⟨*proof*⟩

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*⟩

lemma *rotate-minus1′*: *n* > *0* $\Longrightarrow$ *rotate n ls* = *ms* $\Longrightarrow$
rotate (*n* − *1*) *ls* = *rotate* (*length ms* − *1*) *ms*
⟨*proof*⟩

lemma *rotate-inv1*: $\bigwedge$ *ms*. *n* < *length ls* $\Longrightarrow$ *rotate n ls* = *ms* $\Longrightarrow$
ls = *rotate* ((*length ls*) − *n*) *ms*
⟨*proof*⟩

lemma *rotate-conv-mod′*[*simp*]: *rotate* (*n mod length ls*) *ls* = *rotate n ls*
⟨*proof*⟩

lemma *rotate-inv2*: *rotate n ls* = *ms* $\Longrightarrow$
ls = *rotate* ((*length ls*) − (*n mod length ls*)) *ms*
⟨*proof*⟩

lemma *rotate-inv′*: *ls* = *rotate* ((*length ls*) − (*n mod length ls*)) *ms* $\Longrightarrow$
rotate n ls = *ms*
⟨*proof*⟩

lemma *rotate-id*[*simp*]: *rotate* ((*length ls*) − (*n mod length ls*)) (*rotate n ls*) = *ls*
⟨*proof*⟩

lemma *nth-rotate1-Suc*: *Suc n* < *length ls* ⟹ *ls*!(*Suc n*) = (*rotate1 ls*)!*n*
⟨*proof*⟩

lemma *nth-rotate1-0*: *ls*!*0* = (*rotate1 ls*)!(*length ls* − *1*) ⟨*proof*⟩

lemma *nth-rotate1*: *0* < *length ls* ⟹ *ls*!((*Suc n*) *mod* (*length ls*)) = (*rotate1 ls*)!(*n mod* (*length ls*))
⟨*proof*⟩

lemma *rotate-Suc2*[*simp*]: *rotate n* (*rotate1 xs*) = *rotate* (*Suc n*) *xs*
⟨*proof*⟩

lemma *nth-rotate*: ⋀ *ls*. *0* < *length ls* ⟹ *ls*!((*n*+*m*) *mod* (*length ls*)) = (*rotate m ls*)!(*n mod* (*length ls*))
⟨*proof*⟩

lemma *help1*: ∃ *n*. *filter f* (*rotate1 ls*) = *rotate n* (*filter f ls*)
⟨*proof*⟩

lemma *help3*: ¬ *f l* ⟹ *n* < *length ls* ⟹ *filter f* (*rotate n ls*) = *filter f* (*rotate n* (*rotate1* (*l* # *ls*)))
⟨*proof*⟩

lemma *help4*: ¬ *f l* ⟹ *n* < *length ls* ⟹ *filter f* (*rotate n ls*) = *filter f* (*rotate* (*Suc n*) (*l* # *ls*))
⟨*proof*⟩

lemma *help2*: ∃ *n*. *rotate1* (*filter f ls*) = *filter f* (*rotate n ls*)
⟨*proof*⟩

lemma *rotate-help5*: ∃ *n*. *filter f* (*rotate m ls*) = *rotate n* (*filter f ls*)
⟨*proof*⟩

lemma *rotate-help6*: ∃ *n*. *rotate m* (*filter f ls*) = *filter f* (*rotate n ls*)
⟨*proof*⟩

1.3 *splitAt*

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

constdefs *splitAt* :: *'a* ⇒ *'a list* ⇒ *'a list* × *'a list*

splitAt c as $\equiv$ *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*))
⟨*proof*⟩

lemma *splitAtRec-distinct-fst*: $\bigwedge$ *s*. *distinct vs* $\Longrightarrow$ *distinct s* $\Longrightarrow$ (*set s*) ∩ (*set vs*) = {} $\Longrightarrow$ *distinct* (*fst* (*splitAtRec ram1 s vs*))
⟨*proof*⟩

lemma *splitAtRec-distinct-snd*: $\bigwedge$ *s*. *distinct vs* $\Longrightarrow$ *distinct s* $\Longrightarrow$ (*set s*) ∩ (*set vs*) = {} $\Longrightarrow$ *distinct* (*snd* (*splitAtRec ram1 s vs*))
⟨*proof*⟩

lemma *splitAtRec-ram*:
$\bigwedge$ *us a b*. *ram* ∈ *set vs* $\Longrightarrow$ (*a*, *b*) = *splitAtRec ram us vs* $\Longrightarrow$
us @ *vs* = *a* @ [*ram*] @ *b*
⟨*proof*⟩

lemma *splitAtRec-notRam*:
$\bigwedge$ *us*. *ram* ∉ *set vs* $\Longrightarrow$ *splitAtRec ram us vs* = (*us* @ *vs*, [])
⟨*proof*⟩

lemma *splitAtRec-distinct*: $\bigwedge$ *s*. *distinct vs* $\Longrightarrow$
distinct s $\Longrightarrow$ (*set s*) ∩ (*set vs*) = {} $\Longrightarrow$
set (*fst* (*splitAtRec ram s vs*)) ∩ *set* (*snd* (*splitAtRec ram s vs*)) = {}
⟨*proof*⟩

1.3.2 *splitAt*

lemma *splitAt-conv*:
splitAt x xs = (*takeWhile* (%*y*. *y*≠*x*) *xs*, *tl*(*dropWhile* (%*y*. *y*≠*x*) *xs*))
⟨*proof*⟩

lemma *splitAt-no-ram*[*simp*]:
ram ∉ *set vs* $\Longrightarrow$ *splitAt ram vs* = (*vs*, [])
⟨*proof*⟩

lemma *splitAt-split*:
ram ∈ *set vs* $\Longrightarrow$ (*a*,*b*) = *splitAt ram vs* $\Longrightarrow$ *vs* = *a* @ *ram* # *b*
⟨*proof*⟩

lemma *splitAt-ram*:
ram ∈ *set vs* $\Longrightarrow$ *vs* = *fst* (*splitAt ram vs*) @ *ram* # *snd* (*splitAt ram vs*)
⟨*proof*⟩

lemma *fst-splitAt-last*:

⟦ *xs* ≠ []; *distinct xs* ⟧ ⟹ *fst* (*splitAt* (*last xs*) *xs*) = *butlast xs*
⟨*proof*⟩

1.3.3 Sets

lemma *splitAtRec-union*:
⋀ *a b s*. (*a*,*b*) = *splitAtRec ram s vs* ⟹ (*set a* ∪ *set b*) − {*ram*} = (*set vs* ∪ *set s*) − {*ram*}
⟨*proof*⟩

lemma *splitAt-union*:
(*a*,*b*) = *splitAt ram vs* ⟹ (*set a* ∪ *set b*) − {*ram*} = *set vs* − {*ram*}
⟨*proof*⟩

lemma *splitAt-subset-ab*:
(*a*,*b*) = *splitAt ram vs* ⟹ *set a* ⊆ *set vs* ∧ *set b* ⊆ *set vs*
⟨*proof*⟩

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

lemma *splitAt-subset-snd*:
set (*snd* (*splitAt ram vs*)) ⊆ *set vs*
⟨*proof*⟩

lemma *splitAt-in-fst*[*dest*]: *v* ∈ *set* (*fst* (*splitAt ram vs*)) ⟹ *v* ∈ *set vs*
⟨*proof*⟩

lemma *splitAt-not1*:
v ∉ *set vs* ⟹ *v* ∉ *set* (*fst* (*splitAt ram vs*)) ⟨*proof*⟩

lemma *splitAt-in-snd*[*dest*]: *v* ∈ *set* (*snd* (*splitAt ram vs*)) ⟹ *v* ∈ *set vs*
⟨*proof*⟩

1.3.4 Distinctness

lemma *splitAt-distinct-ab*:
distinct vs ⟹ (*a*,*b*) = *splitAt ram vs* ⟹ *distinct a* ∧ *distinct b*
⟨*proof*⟩

lemma *splitAt-distinct-a*:
distinct vs ⟹ (*a*,*b*) = *splitAt ram vs* ⟹ *distinct a*
⟨*proof*⟩

lemma *splitAt-distinct-b*:
distinct vs ⟹ (*a*,*b*) = *splitAt ram vs* ⟹ *distinct b*
⟨*proof*⟩

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

distinct vs $\Longrightarrow$ *distinct* (*fst* (*splitAt ram vs*))
⟨*proof*⟩

lemma *splitAt-distinct-snd*[*intro*]:
distinct vs $\Longrightarrow$ *distinct* (*snd* (*splitAt ram vs*))
⟨*proof*⟩

lemma *splitAt-distinct-ab*:
distinct vs $\Longrightarrow$ (*a*,*b*) = *splitAt ram vs* $\Longrightarrow$ *set a* $\cap$ *set b* = {}
⟨*proof*⟩

lemma *splitAt-distinct-fst-snd*:
distinct vs $\Longrightarrow$ *set* (*fst* (*splitAt ram vs*)) $\cap$ *set* (*snd* (*splitAt ram vs*)) = {}
⟨*proof*⟩

lemma *splitAt-distinct-ram-fst*[*intro*]:
distinct vs $\Longrightarrow$ *ram* $\notin$ *set* (*fst* (*splitAt ram vs*))
⟨*proof*⟩

lemma *splitAt-distinct-ram-snd*[*intro*]:
distinct vs $\Longrightarrow$ *ram* $\notin$ *set* (*snd* (*splitAt ram vs*))
⟨*proof*⟩

lemma *splitAt-1*[*simp*]:
splitAt ram [] = ([],[]) ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *splitAt-or*:
v $\in$ *set vs* $\Longrightarrow$ *v* $\in$ *set* (*fst* (*splitAt ram vs*)) $\vee$ *v* $\in$ *set* (*snd* (*splitAt ram vs*)) $\vee$ *v* = *ram*
⟨*proof*⟩

lemma *splitAt-distinct-fst*: *distinct vs* $\Longrightarrow$ *distinct* (*fst* (*splitAt ram1 vs*))
⟨*proof*⟩

lemma *splitAt-distinct-a*: *distinct vs* $\Longrightarrow$ (*a*,*b*) = *splitAt ram vs* $\Longrightarrow$ *distinct a*
⟨*proof*⟩

lemma *splitAt-distinct-snd*: *distinct vs* $\Longrightarrow$ *distinct* (*snd* (*splitAt ram1 vs*))
⟨*proof*⟩

lemma *splitAt-distinct-b*: *distinct vs* $\Longrightarrow$ (*a*,*b*) = *splitAt ram vs* $\Longrightarrow$ *distinct b*
⟨*proof*⟩

lemma *splitAt-distinct*: *distinct vs* $\Longrightarrow$ *set* (*fst* (*splitAt ram vs*)) $\cap$ *set* (*snd* (*splitAt*

ram vs)) = {}
⟨*proof*⟩

lemma *splitAt-subset*: (*a*,*b*) = *splitAt ram vs* ⟹ (*set a* ⊆ *set vs*) ∧ (*set b* ⊆ *set vs*)
⟨*proof*⟩

lemma *splitAt-subset1*: (*a*,*b*) = *splitAt ram vs* ⟹ (*set a* ⊆ *set vs*)
⟨*proof*⟩

lemma *splitAt-subset2*: (*a*,*b*) = *splitAt ram vs* ⟹ (*set b* ⊆ *set vs*)
⟨*proof*⟩

1.3.5 *splitAt* composition

lemma *set-help*: *v* ∈ *set* (*as* @ *bs*) ⟹ *v* ∈ *set as* ∨ *v* ∈ *set bs* ⟨*proof*⟩

lemma *splitAt-elements*: *ram1* ∈ *set vs* ⟹ *ram2* ∈ *set vs* ⟹ *ram2* ∈ *set*(*fst* (*splitAt ram1 vs*)) ∨ *ram2* ∈ *set* [*ram1*] ∨ *ram2* ∈ *set*(*snd* (*splitAt ram1 vs*))
⟨*proof*⟩

lemma *splitAt-ram2*: *ram2* ∉ *set* (*snd* (*splitAt ram1 vs*)) ⟹
ram1 ∈ *set vs* ⟹ *ram2* ∈ *set vs* ⟹ *ram1* ≠ *ram2* ⟹
ram2 ∈ *set* (*fst* (*splitAt ram1 vs*)) ⟨*proof*⟩

lemma *splitAt-ram3*: *ram2* ∉ *set* (*fst* (*splitAt ram1 vs*)) ⟹
ram1 ∈ *set vs* ⟹ *ram2* ∈ *set vs* ⟹ *ram1* ≠ *ram2* ⟹
ram2 ∈ *set* (*snd* (*splitAt ram1 vs*)) ⟨*proof*⟩

lemma *splitAt-dist-ram*: *distinct vs* ⟹
vs = *a* @ *ram* # *b* ⟹ (*a*,*b*) = *splitAt ram vs*
⟨*proof*⟩

lemma *distinct-unique1*: *distinct vs* ⟹ *ram* ∈ *set vs* ⟹ *EX*! *s*. *vs* = (*fst s*) @ *ram* # (*snd s*)
⟨*proof*⟩

lemma *splitAt-dist-ram2*: *distinct vs* ⟹ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* ⟹
(*a* @ *ram1* # *b*, *c*) = *splitAt ram2 vs*
⟨*proof*⟩

lemma *splitAt-dist-ram20*: *distinct vs* ⟹ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* ⟹
c = *snd* (*splitAt ram2 vs*)
⟨*proof*⟩

lemma *splitAt-dist-ram21*: *distinct vs* ⟹ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* ⟹
(*a*, *b*) = *splitAt ram1* (*fst* (*splitAt ram2 vs*))
⟨*proof*⟩

lemma *splitAt-dist-ram22*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* $\Longrightarrow$ (*c*, []) = *splitAt ram1* (*snd* (*splitAt ram2 vs*))
⟨*proof*⟩

lemma *splitAt-dist-ram1*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* $\Longrightarrow$ (*a*, *b* @ *ram2* # *c*) = *splitAt ram1 vs*
⟨*proof*⟩

lemma *splitAt-dist-ram10*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* $\Longrightarrow$ *a* = *fst* (*splitAt ram1 vs*)
⟨*proof*⟩

lemma *splitAt-dist-ram11*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* $\Longrightarrow$ (*a*, []) = *splitAt ram2* (*fst* (*splitAt ram1 vs*))
⟨*proof*⟩

lemma *splitAt-dist-ram12*: *distinct vs* $\Longrightarrow$ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* $\Longrightarrow$ (*b*, *c*) = *splitAt ram2* (*snd* (*splitAt ram1 vs*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *snd-splitAt-rev*:
distinct xs $\Longrightarrow$ *x* ∈ *set xs* $\Longrightarrow$
snd(*splitAt x* (*rev xs*)) = *rev*(*fst*(*splitAt x xs*))
⟨*proof*⟩

lemma *splitAt-take*[*simp*]: *distinct ls* $\Longrightarrow$ *i* < *length ls* $\Longrightarrow$ *fst* (*splitAt* (*ls*!*i*) *ls*) = *take i ls*
⟨*proof*⟩

lemma *splitAt-drop*[*simp*]: *distinct ls* $\Longrightarrow$ *i* < *length ls* $\Longrightarrow$ *snd* (*splitAt* (*ls*!*i*) *ls*) = *drop* (*Suc i*) *ls*
⟨*proof*⟩

lemma *fst-splitAt-upt*:
$j <= i \Longrightarrow i < k \Longrightarrow fst(splitAt\ i\ [j..<k]) = [j..<i]$
⟨*proof*⟩

lemma *snd-splitAt-upt*:
$j <= i \Longrightarrow i < k \Longrightarrow snd(splitAt\ i\ [j..<k]) = [i+1..<k]$
⟨*proof*⟩

lemma *local-help1*: $\bigwedge a\ vs.\ vs = c\ @\ r\ \#\ d \Longrightarrow vs = a\ @\ r\ \#\ b \Longrightarrow r \notin set\ a \Longrightarrow r \notin set\ b \Longrightarrow a = c$
⟨*proof*⟩

lemma *local-help*: $vs = a\ @\ r\ \#\ b \Longrightarrow vs = c\ @\ r\ \#\ d \Longrightarrow r \notin set\ a \Longrightarrow r \notin set\ b \Longrightarrow a = c \wedge b = d$
⟨*proof*⟩

lemma *local-help'*: $a\ @\ r\ \#\ b = c\ @\ r\ \#\ d \Longrightarrow r \notin set\ a \Longrightarrow r \notin set\ b \Longrightarrow a = c \wedge b = d$
⟨*proof*⟩

lemma *splitAt-simp1*: $ram \notin set\ a \Longrightarrow ram \notin set\ b \Longrightarrow fst\ (splitAt\ ram\ (a\ @\ ram\ \#\ b)) = a$
⟨*proof*⟩

lemma *splitAt-simp2*: $ram \notin set\ b \Longrightarrow fst\ (splitAt\ ram\ (ram\ \#\ b)) = []$
⟨*proof*⟩

lemma *splitAt-simp3*: $ram \notin set\ a \Longrightarrow fst\ (splitAt\ ram\ (a\ @\ [ram])) = a$
⟨*proof*⟩

lemma *splitAt-simp4*: $ram \notin set\ a \Longrightarrow ram \notin set\ b \Longrightarrow snd\ (splitAt\ ram\ (a\ @\ ram\ \#\ b)) = b$
⟨*proof*⟩

lemma *help'''-in*: $\bigwedge xs.\ ram \in set\ b \Longrightarrow fst\ (splitAtRec\ ram\ xs\ b) = xs\ @\ fst\ (splitAtRec\ ram\ []\ b)$
⟨*proof*⟩

lemma *help'''-notin*: $\bigwedge xs.\ ram \notin set\ b \Longrightarrow fst\ (splitAtRec\ ram\ xs\ b) = xs\ @\ fst\ (splitAtRec\ ram\ []\ b)$
⟨*proof*⟩

lemma *help'''*: $fst\ (splitAtRec\ ram\ xs\ b) = xs\ @\ fst\ (splitAtRec\ ram\ []\ b)$

⟨*proof*⟩

lemma *splitAt-simpA*[*simp*]: *fst* (*splitAt ram* (*ram* # *b*)) = [] ⟨*proof*⟩
lemma *splitAt-simpB*[*simp*]: *ram* ≠ *a* ⟹ *fst* (*splitAt ram* (*a* # *b*)) = *a* # *fst* (*splitAt ram b*) ⟨*proof*⟩
lemma *splitAt-simpB′*[*simp*]: *a* ≠ *ram* ⟹ *fst* (*splitAt ram* (*a* # *b*)) = *a* # *fst* (*splitAt ram b*) ⟨*proof*⟩
lemma *splitAt-simpC*[*simp*]: *ram* ∉ *set a* ⟹ *fst* (*splitAt ram* (*a* @ *b*)) = *a* @ *fst* (*splitAt ram b*)
⟨*proof*⟩

lemma *help′′′′*: ⋀ *xs ys*. *snd* (*splitAtRec ram xs b*) = *snd* (*splitAtRec ram ys b*)
⟨*proof*⟩

lemma *splitAt-simpD*[*simp*]: ⋀ *a*. *ram* ≠ *a* ⟹ *snd* (*splitAt ram* (*a* # *b*)) = *snd* (*splitAt ram b*) ⟨*proof*⟩
lemma *splitAt-simpD′*[*simp*]: ⋀ *a*. *a* ≠ *ram* ⟹ *snd* (*splitAt ram* (*a* # *b*)) = *snd* (*splitAt ram b*) ⟨*proof*⟩

lemma *splitAt-simpE*[*simp*]: *snd* (*splitAt ram* (*ram* # *b*)) = *b* ⟨*proof*⟩

lemma *splitAt-simpF*[*simp*]: *ram* ∉ *set a* ⟹ *snd* (*splitAt ram* (*a* @ *b*)) = *snd* (*splitAt ram b*)
⟨*proof*⟩

lemma *splitAt-rotate-pair-conv*:
!!*xs*. ⟦ *distinct xs*; *x* ∈ *set xs* ⟧
⟹ *snd* (*splitAt x* (*rotate n xs*)) @ *fst* (*splitAt x* (*rotate n xs*)) =
snd (*splitAt x xs*) @ *fst* (*splitAt x xs*)
⟨*proof*⟩

1.4 *between*

constdefs *between* :: *′a list* ⇒ *′a* ⇒ *′a* ⇒ *′a list*
between vs ram$_1$ *ram*$_2$ ≡
let (*pre*$_1$, *post*$_1$) = *splitAt ram*$_1$ *vs in*
if ram$_2$ *mem post*$_1$
then let (*pre*$_2$, *post*$_2$) = *splitAt ram*$_2$ *post*$_1$ *in pre*$_2$
else let (*pre*$_2$, *post*$_2$) = *splitAt ram*$_2$ *pre*$_1$ *in post*$_1$ @ *pre*$_2$

lemma *inbetween-inset*:
x ∈ *set*(*between xs a b*) ⟹ *x* ∈ *set xs*
⟨*proof*⟩

lemma *notinset-notinbetween*:
x ∉ *set xs* ⟹ *x* ∉ *set*(*between xs a b*)
⟨*proof*⟩

lemma *set-between-id*:
distinct xs $\Longrightarrow$ *x* $\in$ *set xs* $\Longrightarrow$
set(*between xs x x*) = *set xs* $-$ {*x*}
⟨*proof*⟩

lemma *split-between*:
⟦ *distinct vs*; *r* $\in$ *set vs*; *v* $\in$ *set vs*; *u* $\in$ *set*(*between vs r v*) ⟧ $\Longrightarrow$
between vs r v =
(*if r*=*u then* [] *else between vs r u* @ [*u*]) @ *between vs u v*
⟨*proof*⟩

1.5 Tables

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

constdefs *isTable* :: ('*a* ⇒ '*b*) ⇒ '*a list* ⇒ ('*a*, '*b*) *table* ⇒ *bool*
isTable f vs t ≡ ∀ *p*. *p* $\in$ *set t* ⟶ *snd p* = *f* (*fst p*) ∧ *fst p* $\in$ *set vs*

lemma *isTable-eq*: *isTable E vs* ((*a*,*b*)#*ps*) $\Longrightarrow$ *b* = *E a*
⟨*proof*⟩

lemma *isTable-subset*:
set qs $\subseteq$ *set ps* $\Longrightarrow$ *isTable E vs ps* $\Longrightarrow$ *isTable E vs qs*
⟨*proof*⟩

lemma *isTable-Cons*: *isTable E vs* ((*a*,*b*)#*ps*) $\Longrightarrow$ *isTable E vs ps*
⟨*proof*⟩

constdefs
removeKey :: '*a* ⇒ ('*a* × '*b*) *list* ⇒ ('*a* × '*b*) *list*
removeKey a ps ≡ [*p* $\in$ *ps*. *a* ≠ *fst p*]

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

lemma *removeKey-subset*[*simp*]: *set* (*removeKey a ps*) $\subseteq$ *set ps*
⟨*proof*⟩

lemma *length-removeKey*[*simp*]: |*removeKey w ps*| $\leq$ |*ps*|
⟨*proof*⟩

lemma *length-removeKeyList*:
length (*removeKeyList ws ps*) $\leq$ *length ps* (**is** *?P ws*)
⟨*proof*⟩

lemma *removeKeyList-subset*[*simp*]: *set* (*removeKeyList ws ps*) ⊆ *set ps*
⟨*proof*⟩

lemma *notin-removeKey1*: (*a*, *b*) ∉ *set* (*removeKey a ps*)
⟨*proof*⟩

lemma *notin-removeKey*: *r* ∉ *fst* ‘ *set* (*removeKey r ps*)
⟨*proof*⟩

lemma *notin-removeKeyList1*:
⋀*a*. *a* ∈ *set rs* ⟹ (*a*, *b*) ∉ *set* (*removeKeyList rs ps*)
⟨*proof*⟩

lemma *notin-removeKeyList*: ⋀*r*. *r* ∈ *set rs* ⟹ *r* ∉ *fst* ‘ *set* (*removeKeyList rs ps*)
⟨*proof*⟩

lemma *removeKeyList-eq*:
removeKeyList as ps = [*p* ∈ *ps*. ∀ *a* ∈ *set as*. *a* ≠ *fst p*]
⟨*proof*⟩

lemma *removeKey-empty*[*simp*]: *removeKey a* [] = []
⟨*proof*⟩
lemma *removeKeyList-empty*[*simp*]: *removeKeyList ps* [] = []
⟨*proof*⟩
lemma *removeKeyList-cons*[*simp*]:
removeKeyList ws (*p*#*ps*)
= (*if fst p* ∈ *set ws then removeKeyList ws ps else p*#(*removeKeyList ws ps*))
⟨*proof*⟩

end

2 Isomorphisms Between Plane Graphs

theory *PlaneGraphIso*
imports *Main*
begin

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

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

consts

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

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

lemma [*iff*]: $((x,y) \in \{\cong\}) = x \cong y$
⟨*proof*⟩

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

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

2.1 Equivalence of faces

Two faces are equivalent modulo rotation:

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

lemma *congs-refl*[*iff*]: $(xs::'a\ list) \cong xs$
⟨*proof*⟩

lemma *congs-sym*: **assumes** *A*: $(xs::'a\ list) \cong ys$ **shows** $ys \cong xs$
⟨*proof*⟩

lemma *congs-trans*: $(xs::'a\ list) \cong ys \Longrightarrow ys \cong zs \Longrightarrow xs \cong zs$
⟨*proof*⟩

lemma *equiv-EqF*: *equiv* $(UNIV::'a\ list\ set)$ $\{\cong\}$
⟨*proof*⟩

lemma *congs-distinct*:
$F_1 \cong F_2 \Longrightarrow distinct\ F_2 = distinct\ F_1$
⟨*proof*⟩

lemma *congs-length*:
$F_1 \cong F_2 \Longrightarrow length\ F_2 = length\ F_1$
⟨*proof*⟩

lemma *congs-pres-nodes*: $F_1 \cong F_2 \Longrightarrow set\ F_1 = set\ F_2$
⟨*proof*⟩

lemma *congs-map*:
inj-on f $(set\ xs \cup set\ ys) \Longrightarrow (map\ f\ xs \cong map\ f\ ys) = (xs \cong ys)$
⟨*proof*⟩

lemma *list-cong-rev-iff*[*simp*]:
(*rev xs* ≅ *rev ys*) = (*xs* ≅ *ys*)
⟨*proof*⟩

lemma *singleton-list-cong-eq-iff*[*simp*]:
({*xs*::'*a list*} // {≅} = {*ys*} // {≅}) = (*xs* ≅ *ys*)
⟨*proof*⟩

2.2 Homomorphism and isomorphism

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

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

is-hom :: ('*a* ⇒ '*b*) ⇒ '*a fgraph* ⇒ '*b fgraph* ⇒ *bool*
is-hom φ *Fs*$_1$ *Fs*$_2$ ≡ *is-Hom* φ (*set Fs*$_1$) (*set Fs*$_2$)

is-pr-iso :: ('*a* ⇒ '*b*) ⇒ '*a fgraph* ⇒ '*b fgraph* ⇒ *bool*
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*)
⟨*proof*⟩

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

lemma *Hom-pres-face-nodes*:
is-Hom φ *Fs*$_1$ *Fs*$_2$ ⟹ (⋃ *F*∈*Fs*$_1$. {φ ‘ (*set F*)}) = (⋃ *F*∈*Fs*$_2$. {*set F*})
⟨*proof*⟩

lemma *Hom-pres-nodes*:
is-Hom φ *Fs*$_1$ *Fs*$_2$ ⟹ φ ‘ (⋃ *F*∈*Fs*$_1$. *set F*) = (⋃ *F*∈*Fs*$_2$. *set F*)
⟨*proof*⟩

Therefore isomorphisms preserve cardinality of node set.

lemma *pr-Iso-same-no-nodes*:
⟦ *is-pr-Iso* φ *Fs*$_1$ *Fs*$_2$; *finite Fs*$_1$ ⟧
⟹ *card*(⋃ *F*∈*Fs*$_1$. *set F*) = *card*(⋃ *F*∈*Fs*$_2$. *set F*)
⟨*proof*⟩

lemma *pr-iso-same-no-nodes*:
is-pr-iso φ *Fs*$_1$ *Fs*$_2$ ⟹ *card*(⋃ *F*∈*set Fs*$_1$. *set F*) = *card*(⋃ *F*∈*set Fs*$_2$. *set F*)
⟨*proof*⟩

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* $\varphi\ Fs_1\ Fs_2$
shows *length* Fs_1 = *length* Fs_2
⟨*proof*⟩

lemma *is-Hom-distinct*:
⟦ *is-Hom* $\varphi\ Fs_1\ Fs_2$; $\forall F{\in}Fs_1$. *distinct* F; $\forall F{\in}Fs_2$. *distinct* F ⟧
$\Longrightarrow \forall F{\in}Fs_1$. *distinct*$(map\ \varphi\ F)$
⟨*proof*⟩

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

lemma *is-pr-Iso-rec*:
⟦ *inj-on* $(\%xs.\ \{xs\}//\{\cong\})$ Fs_1; *inj-on* $(\%xs.\ \{xs\}//\{\cong\})$ Fs_2; $F_1 \in Fs_1$ ⟧ $\Longrightarrow$
is-pr-Iso $\varphi\ Fs_1\ Fs_2$ =
$(\exists F_2 \in Fs_2.$ *length* F_1 = *length* $F_2 \wedge$ *is-pr-Iso* $\varphi\ (Fs_1 - \{F_1\})\ (Fs_2 - \{F_2\})$
$\wedge\ (\exists n.\ map\ \varphi\ F_1 = rotate\ n\ F_2)$
$\wedge$ *inj-on* $\varphi\ (\bigcup F{\in}Fs_1.\ set\ F))$
⟨*proof*⟩

lemma *is-iso-Cons*:
⟦ *distinct* $(F_1\#Fs_1')$; *distinct* Fs_2;
inj-on $(\%xs.\{xs\}//\{\cong\})$ $(set(F_1\#Fs_1'))$; *inj-on* $(\%xs.\{xs\}//\{\cong\})$ $(set\ Fs_2)$ ⟧
$\Longrightarrow$
is-pr-iso $\varphi\ (F_1\#Fs_1')\ Fs_2$ =
$(\exists F_2 \in set\ Fs_2.$ *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* $\varphi\ (set\ F_1 \cup (\bigcup F{\in}set\ Fs_1'.\ set\ F)))$
⟨*proof*⟩

2.3 Isomorphism tests

lemma *map-upd-submap*:
$x \notin dom\ m \Longrightarrow (m(x \mapsto y) \subseteq_m m') = (m'\ x = Some\ y \wedge m \subseteq_m m')$
⟨*proof*⟩

lemma *map-of-zip-submap*: ⟦ *length xs* = *length ys*; *distinct xs* ⟧ $\Longrightarrow$
$(map\text{-}of\ (zip\ xs\ ys) \subseteq_m Some \circ f) = (map\ f\ xs = ys)$
⟨*proof*⟩

consts
pr-iso-test0 :: $('a\ \tilde{}{=>}\ 'b) \Rightarrow$ *'a fgraph* $\Rightarrow$ *'b fgraph* $\Rightarrow$ *bool*

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

pr-iso-test0 m $(F_1 \# Fs_1)$ Fs_2 =
$(\exists\, F_2 \in set\ Fs_2.\ length\ F_1 = length\ F_2 \wedge$
$(\exists\, n.\ let\ m' = map\text{-}of(zip\ F_1\ (rotate\ n\ F_2))\ in$
if $m \subseteq_m m +\!\!+ m' \wedge inj\text{-}on\ (m +\!\!+ m')\ (dom(m +\!\!+ m'))$
then pr-iso-test0 $(m +\!\!+ m')\ Fs_1\ (remove1\ F_2\ Fs_2)$ *else False*))

lemma *map-compatI*: $[\![\ f \subseteq_m Some \circ h;\ g \subseteq_m Some \circ h\ ]\!] \Longrightarrow f \subseteq_m f +\!\!+ g$
⟨*proof*⟩

lemma *inj-on-map-addI1*:
$[\![\ inj\text{-}on\ m\ A;\ m \subseteq_m m +\!\!+ m';\ A \subseteq dom\ m\ ]\!] \Longrightarrow inj\text{-}on\ (m +\!\!+ m')\ A$
⟨*proof*⟩

lemma *map-image-eq*: $[\![\ A \subseteq dom\ m;\ m \subseteq_m m'\ ]\!] \Longrightarrow m\ `\ A = m'\ `\ A$
⟨*proof*⟩

lemma *inj-on-map-add-Un*:
$[\![\ inj\text{-}on\ m\ (dom\ m);\ inj\text{-}on\ m'\ (dom\ m');\ m \subseteq_m Some \circ f;\ m' \subseteq_m Some \circ f;$
$inj\text{-}on\ f\ (dom\ m' \cup dom\ m);\ A = dom\ m';\ B = dom\ m\ ]\!]$
$\Longrightarrow inj\text{-}on\ (m +\!\!+ m')\ (A \cup B)$
⟨*proof*⟩

lemma *map-of-zip-eq-SomeD*: $length\ xs = length\ ys \Longrightarrow$
$map\text{-}of\ (zip\ xs\ ys)\ x = Some\ y \Longrightarrow y \in set\ ys$
⟨*proof*⟩

lemma *inj-on-map-of-zip*:
$[\![\ length\ xs = length\ ys;\ distinct\ ys\ ]\!]$
$\Longrightarrow inj\text{-}on\ (map\text{-}of\ (zip\ xs\ ys))\ (set\ xs)$
⟨*proof*⟩

lemma *pr-iso-test0-correct*: $\bigwedge m\ Fs_2.$
$[\![\ \forall F{\in}set\ Fs_1.\ distinct\ F;\ \forall F{\in}set\ Fs_2.\ distinct\ F;$
$distinct\ Fs_1;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_1);$
$distinct\ Fs_2;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_2);\ inj\text{-}on\ m\ (dom\ m)\ ]\!] \Longrightarrow$
$pr\text{-}iso\text{-}test0\ m\ Fs_1\ Fs_2 =$
$(\exists \varphi.\ is\text{-}pr\text{-}iso\ \varphi\ Fs_1\ Fs_2 \wedge m \subseteq_m Some \circ \varphi \wedge$
$inj\text{-}on\ \varphi\ (dom\ m \cup (\bigcup F{\in}set\ Fs_1.\ set\ F)))$
⟨*proof*⟩

corollary *pr-iso-test0-corr*:
$[\![\ \forall F{\in}set\ Fs_1.\ distinct\ F;\ \forall F{\in}set\ Fs_2.\ distinct\ F;$
$distinct\ Fs_1;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_1);$
$distinct\ Fs_2;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_2)\ ]\!] \Longrightarrow$
$pr\text{-}iso\text{-}test0\ empty\ Fs_1\ Fs_2 = (\exists \varphi.\ is\text{-}pr\text{-}iso\ \varphi\ Fs_1\ Fs_2)$
⟨*proof*⟩

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

otherwise be vacuous).

consts
pr-iso-test1 :: $('a \sim\!\!=\!\!> 'b) \Rightarrow 'a$ *fgraph* $\Rightarrow 'b$ *fgraph* $\Rightarrow$ *bool*

primrec
pr-iso-test1 m $[]$ $Fs_2 = (Fs_2 = [])$
pr-iso-test1 m $(F_1 \# Fs_1)$ $Fs_2 =$
$(\exists F_2 \in set\ Fs_2.\ length\ F_1 = length\ F_2 \wedge$
$(\exists n < length\ F_2.\ let\ m' = map\text{-}of(zip\ F_1\ (rotate\ n\ F_2))\ in$
if $m \subseteq_m m +\!\!+ m' \wedge inj\text{-}on\ (m +\!\!+ m')\ (dom(m +\!\!+ m'))$
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* $m\ Fs_1\ Fs_2$
⟨*proof*⟩

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

corollary *pr-iso-test1-corr*:
$[\![\forall F \in set\ Fs_1.\ distinct\ F;\ \forall F \in set\ Fs_2.\ distinct\ F;\ [] \notin set\ Fs_2;$
$distinct\ Fs_1;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_1);$
$distinct\ Fs_2;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_2)\ ]\!] \Longrightarrow$
pr-iso-test1 empty $Fs_1\ Fs_2 = (\exists \varphi.\ is\text{-}pr\text{-}iso\ \varphi\ Fs_1\ Fs_2)$
⟨*proof*⟩

2.3.1 Implementing maps by lists

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

constdefs
oneone :: $('a * 'b)list \Rightarrow$ *bool*
oneone xys $\equiv$ $distinct(map\ fst\ xys) \wedge distinct(map\ snd\ xys)$
declare *oneone-def*[*simp*]

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

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

primrec
pr-iso-test2 tst mrg I $[]$ $Fs_2 = (Fs_2 = [])$
pr-iso-test2 tst mrg I $(F_1 \# Fs_1)$ $Fs_2 =$
$(\exists F_2 \in set\ Fs_2.\ length\ F_1 = length\ F_2 \wedge$
$(\exists n < length\ F_2.\ let\ I' = zip\ F_1\ (rotate\ n\ F_2)\ in$
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*:
$y \notin snd\ `\ set\ xys \Longrightarrow Some\ y \notin range(map\text{-}of\ xys)$
⟨*proof*⟩

lemma *inj-on-map-upd*:
$[\![\ inj\text{-}on\ m\ (dom\ m);\ Some\ y \notin range\ m\ ]\!] \Longrightarrow inj\text{-}on\ (m(x\mapsto y))\ (dom\ m)$
⟨*proof*⟩

lemma [*simp*]:
$distinct(map\ snd\ xys) \Longrightarrow inj\text{-}on\ (map\text{-}of\ xys)\ (dom(map\text{-}of\ xys))$
⟨*proof*⟩

lemma *lem*: $Ball\ (set\ xs)\ P \Longrightarrow Ball\ (set\ (remove1\ x\ xs))\ P = True$
⟨*proof*⟩

lemma *pr-iso-test2-conv-1*:
$!!I\ Fs_2.$
$[\![\ \forall I\ I'.\ oneone\ I \longrightarrow oneone\ I' \longrightarrow$
$tst\ I'\ I = (let\ m = map\text{-}of\ I;\ m' = map\text{-}of\ I'$
$in\ m \subseteq_m m\ ++\ m' \wedge inj\text{-}on\ (m{+}{+}m')\ (dom(m{+}{+}m')));$
$\forall I\ I'.\ oneone\ I \longrightarrow oneone\ I' \longrightarrow tst\ I'\ I$
$\longrightarrow map\text{-}of(mrg\ I'\ I) = map\text{-}of\ I\ ++\ map\text{-}of\ I';$
$\forall I\ I'.\ oneone\ I\ \&\ oneone\ I' \longrightarrow tst\ I'\ I \longrightarrow oneone\ (mrg\ I'\ I);$
$oneone\ I;$
$\forall F \in set\ Fs_1.\ distinct\ F;\ \forall F \in set\ Fs_2.\ distinct\ F\ ]\!] \Longrightarrow$
$pr\text{-}iso\text{-}test2\ tst\ mrg\ I\ Fs_1\ Fs_2 = pr\text{-}iso\text{-}test1\ (map\text{-}of\ I)\ Fs_1\ Fs_2$
⟨*proof*⟩

A simple implementation

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

lemma *image-map-upd*:
$x \notin dom\ m \Longrightarrow m(x\mapsto y)\ `\ A = m\ `\ (A{-}\{x\}) \cup (if\ x \in A\ then\ \{Some\ y\}\ else\ \{\})$
⟨*proof*⟩

lemma *image-map-of-conv-Image*:
$!!A.\ [\![\ distinct(map\ fst\ xys)\ ]\!]$
$\Longrightarrow map\text{-}of\ xys\ `\ A = Some\ `\ (set\ xys\ ``\ A) \cup (if\ A \subseteq fst\ `\ set\ xys\ then\ \{\}\ else\ \{None\})$
⟨*proof*⟩

lemma [*simp*]: $m{+}{+}m' \mathbin{`} (dom\ m' - A) = m' \mathbin{`} (dom\ m' - A)$
⟨*proof*⟩

declare *Diff-subset* [*iff*]

lemma *test-correct*:
$[\![\ oneone\ I;\ oneone\ I'\ ]\!] \Longrightarrow$
$\quad test\ I'\ I = (let\ m = map\text{-}of\ I;\ m' = map\text{-}of\ I'$
$\qquad in\ m \subseteq_m m\ {+}{+}\ m' \wedge inj\text{-}on\ (m{+}{+}m')\ (dom(m{+}{+}m')))$
⟨*proof*⟩

corollary *test-corr*:
$\forall I\ I'.\ oneone\ I \longrightarrow oneone\ I' \longrightarrow$
$\quad test\ I'\ I = (let\ m = map\text{-}of\ I;\ m' = map\text{-}of\ I'$
$\qquad in\ m \subseteq_m m\ {+}{+}\ m' \wedge inj\text{-}on\ (m{+}{+}m')\ (dom(m{+}{+}m')))$
⟨*proof*⟩

constdefs
$merge :: ('a,'b)merger$
$merge\ I'\ I \equiv [xy : I'.\ fst\ xy \notin fst \mathbin{`} set\ I] \mathbin{@} I$

lemma *help1*:
$distinct(map\ fst\ xys) \Longrightarrow map\text{-}of\ (filter\ P\ xys) =$
$map\text{-}of\ xys \mathbin{|`} \{x.\ \exists y.\ (x,y) \in set\ xys \wedge P(x,y)\}$
⟨*proof*⟩

lemma *merge-correct*:
$\forall I\ I'.\ oneone\ I \longrightarrow oneone\ I' \longrightarrow test\ I'\ I$
$\longrightarrow map\text{-}of(merge\ I'\ I) = map\text{-}of\ I\ {+}{+}\ map\text{-}of\ I'$
⟨*proof*⟩

lemma *merge-inv*:
$\forall I\ I'.\ oneone\ I \wedge oneone\ I' \longrightarrow test\ I'\ I \longrightarrow oneone\ (merge\ I'\ I)$
⟨*proof*⟩

corollary *pr-iso-test2-corr*:
$[\![\ \forall F{\in}set\ Fs_1.\ distinct\ F;\ \forall F{\in}set\ Fs_2.\ distinct\ F;\ [] \notin set\ Fs_2;$
$\quad distinct\ Fs_1;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_1);$
$\quad distinct\ Fs_2;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_2)\ ]\!] \Longrightarrow$
$\qquad pr\text{-}iso\text{-}test2\ test\ merge\ []\ Fs_1\ Fs_2 = (\exists \varphi.\ is\text{-}pr\text{-}iso\ \varphi\ Fs_1\ Fs_2)$
⟨*proof*⟩

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

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

lemma *test2-conv-test*: $test2\ I\ I' = test\ I\ I'$

⟨*proof*⟩

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

lemma *merge2-conv-merge*: *merge2 I' I* = *merge I' I*
⟨*proof*⟩

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

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

lemma *pr-iso-test3-conv-2*:
  !!*I* Fs_2. *pr-iso-test3 I* Fs_1 Fs_2 = *pr-iso-test2 test merge I* Fs_1 Fs_2
⟨*proof*⟩

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

A final optimization.

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

corollary *pr-iso-test-correct*:
⟦ ∀ *F*∈*set* Fs_1. *distinct F*; ∀ *F*∈*set* Fs_2. *distinct F*; [] ∉ *set* Fs_2;
  distinct Fs_1; *inj-on* (%*xs*.{*xs*}//{≅}) (*set* Fs_1);
  distinct Fs_2; *inj-on* (%*xs*.{*xs*}//{≅}) (*set* Fs_2);
  n_1 = *card*(⋃ *F*∈*set* Fs_1. *set F*); n_2 = *card*(⋃ *F*∈*set* Fs_2. *set F*) ⟧ ⟹
  pr-iso-test (n_1,Fs_1) (n_2,Fs_2) = (∃ φ. *is-pr-iso* φ Fs_1 Fs_2)

⟨*proof*⟩

2.3.2 'Improper' Isomorphisms

constdefs
is-Iso :: $('a \Rightarrow 'b) \Rightarrow 'a\ Fgraph \Rightarrow 'b\ Fgraph \Rightarrow bool$
$is\text{-}Iso\ \varphi\ Fs_1\ Fs_2 \equiv is\text{-}pr\text{-}Iso\ \varphi\ Fs_1\ Fs_2 \vee is\text{-}pr\text{-}Iso\ \varphi\ Fs_1\ (rev\ `\ Fs_2)$

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

defs (**overloaded**) *iso-fgraph-def*:
$g_1 \simeq g_2 \ \equiv\ \exists\varphi.\ is\text{-}iso\ \varphi\ g_1\ g_2$

constdefs
iso-test :: $(nat * 'a\ fgraph) \Rightarrow (nat * 'b\ fgraph) \Rightarrow bool$
$iso\text{-}test \equiv \%g_1\ g_2.\ pr\text{-}iso\text{-}test\ g_1\ g_2 \vee pr\text{-}iso\text{-}test\ g_1\ (fst\ g_2, map\ rev\ (snd\ g_2))$

lemma *inj-on-image-iff*: $[\![\ ALL\ x{:}A.\ ALL\ y{:}A.\ (g(f\ x) = g(f\ y)) = (g\ x = g\ y);$
$inj\text{-}on\ f\ A\ ]\!] \Longrightarrow inj\text{-}on\ g\ (f\ `\ A) = inj\text{-}on\ g\ A$
⟨*proof*⟩

theorem *iso-correct*:
$[\![\ \forall F{\in}set\ Fs_1.\ distinct\ F;\ \forall F{\in}set\ Fs_2.\ distinct\ F;\ [] \notin set\ Fs_2;$
$distinct\ Fs_1;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_1);$
$distinct\ Fs_2;\ inj\text{-}on\ (\%xs.\{xs\}//\{\cong\})\ (set\ Fs_2);$
$n_1 = card(\bigcup F{\in}set\ Fs_1.\ set\ F);\ n_2 = card(\bigcup F{\in}set\ Fs_2.\ set\ F)\ ]\!] \Longrightarrow$
$iso\text{-}test\ (n_1{,}Fs_1)\ (n_2{,}Fs_2) = (Fs_1 \simeq Fs_2)$
⟨*proof*⟩

2.4 Elementhood and containment modulo

constdefs
pr-iso-in :: $'a \Rightarrow 'a\ set \Rightarrow bool$ (**infix** $\in_{\cong}$ *60*)
$x \in_{\cong} M \equiv \exists y \in M.\ x \cong y$

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

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

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

end

3 More Rotation

theory *Rotation*
imports *ListAux PlaneGraphIso*
begin

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

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

lemma *cong-rotate-to*:
x ∈ *set xs* ⟹ *xs* ≅ *rotate-to xs x*
⟨*proof*⟩

lemma *face-cong-if-norm-eq*:
⟦ *rotate-min xs* = *rotate-min ys*; *xs* ≠ []; *ys* ≠ [] ⟧ ⟹ *xs* ≅ *ys*
⟨*proof*⟩

lemma *norm-eq-if-face-cong*:
⟦ *xs* ≅ *ys*; *distinct xs*; *xs* ≠ [] ⟧ ⟹ *rotate-min xs* = *rotate-min ys*
⟨*proof*⟩

lemma *norm-eq-iff-face-cong*:
⟦ *distinct xs*; *xs* ≠ []; *ys* ≠ [] ⟧ ⟹
(*rotate-min xs* = *rotate-min ys*) = (*xs* ≅ *ys*)
⟨*proof*⟩

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*⟩

end

4 Graph

theory *Graph*
imports *Rotation*
begin

syntax (*xsymbols*)
@*UNION1* :: *pttrns* => *'b set* => *'b set* ((3⋃(00_)/ -) 10)
@*INTER1* :: *pttrns* => *'b set* => *'b set* ((3⋂(00_)/ -) 10)

$@UNION$ :: $pttrn => 'a\ set => 'b\ set => 'b\ set$ $((3\bigcup(00\_\in\_)/\ \_)\ 10)$
$@INTER$ :: $pttrn => 'a\ set => 'b\ set => 'b\ set$ $((3\bigcap(00\_\in\_)/\ \_)\ 10)$

4.1 Notation

types *vertex* = *nat*

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

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

4.2 Faces

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

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

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

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

consts *type* :: $'a \Rightarrow facetype$
primrec *type* (*Face vs f*) = *f*

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

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

The following operation makes a face final.

constdefs *setFinal* :: *face* $\Rightarrow$ *face*
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.

consts *nextElem* :: $'a\ list \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$
primrec
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*)

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

nextVertices is n-fold application of *nextvertex*.

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

lemma *nextV2*: $f^2 \cdot v = f \cdot (f \cdot v)$
⟨*proof*⟩ *edges* (*f*::*face*) ≡ {(*a*, *f* · *a*)|*a*. *a* ∈ $\mathcal{V}$ *f*}

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

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

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

4.3 Graphs

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

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

syntax -*Faces* :: *graph* ⇒ *face set* ($\mathcal{F}$)
translations $\mathcal{F}$ *g* == *set*(*faces g*)

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

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

lemma *vertices-graph*: *vertices g* = [*0* ..< *countVertices g*]
⟨*proof*⟩

lemma *in-vertices-graph*[*THEN eq-reflection*, *code unfold*]:
  v ∈ *set* (*vertices g*) = (*v* < *countVertices g*)
⟨*proof*⟩

lemma *len-vertices-graph*[*THEN eq-reflection*, *code unfold*]:
  |*vertices g*| = *countVertices g*
⟨*proof*⟩

consts *faceListAt* :: *graph* ⇒ *face list list*
primrec

faceListAt (*Graph fs n f h*) = *f*

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

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

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

lemma *graph-split*:
g = *Graph* (*faces g*)
(*countVertices g*)
(*faceListAt g*)
(*heights g*)
⟨*proof*⟩

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

4.4 Operations on graphs

final graph, final / nonfinal faces

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

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

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

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

lemma *finalGraph-faces*[*simp*]: *final g* ⟹ *finals g* = *faces g*
⟨*proof*⟩

lemma *finalGraph-face*: *final g* ⟹ *f* ∈ *set* (*faces g*) ⟹ *final f*
⟨*proof*⟩

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

lemma *finalVertex-final-face*[*dest*]:
finalVertex g v $\Longrightarrow$ *f* $\in$ *set* (*facesAt g v*) $\Longrightarrow$ *final f*
⟨*proof*⟩

counting faces

constdefs *degree* :: *graph* $\Rightarrow$ *vertex* $\Rightarrow$ *nat*
degree g v $\equiv$ |*facesAt g v*|

constdefs *tri* :: *graph* $\Rightarrow$ *vertex* $\Rightarrow$ *nat*
tri g v $\equiv$ |[*f*: *facesAt g v*. *final f* $\wedge$ |*vertices f*| = *3*]|

constdefs *quad* :: *graph* $\Rightarrow$ *vertex* $\Rightarrow$ *nat*
quad g v $\equiv$ |[*f*: *facesAt g v*. *final f* $\wedge$ |*vertices f*| = *4*]|

constdefs *except* :: *graph* $\Rightarrow$ *vertex* $\Rightarrow$ *nat*
except g v $\equiv$ |[*f*: *facesAt g v*. *final f* $\wedge$ *5* $\leq$ |*vertices f*|]|

constdefs *vertextype* :: *graph* $\Rightarrow$ *vertex* $\Rightarrow$ *nat* $\times$ *nat* $\times$ *nat*
vertextype g v $\equiv$ (*tri g v*, *quad g v*, *except g v*)

lemma[*simp*]: *0* $\leq$ *tri g v* ⟨*proof*⟩

lemma[*simp*]: *0* $\leq$ *quad g v* ⟨*proof*⟩

lemma[*simp*]: *0* $\leq$ *except g v* ⟨*proof*⟩

constdefs *exceptionalVertex* :: *graph* $\Rightarrow$ *vertex* $\Rightarrow$ *bool*
exceptionalVertex g v $\equiv$ *except g v* $\neq$ *0*

constdefs *noExceptionals* :: *graph* $\Rightarrow$ *vertex set* $\Rightarrow$ *bool*
noExceptionals g V $\equiv$ ($\forall$ *v* $\in$ *V*. $\neg$ *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*) $\equiv \bigcup_{f \in \mathcal{F}\ g}$ *edges f*

constdefs *neighbors* :: *graph* $\Rightarrow$ *vertex* $\Rightarrow$ *vertex list*
neighbors g v $\equiv$ [*f*$\cdot$*v*. *f* $\in$ *facesAt g v*]

4.5 Navigation in graphs

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

constdefs *nextFace* :: *graph* $\times$ *vertex* $\Rightarrow$ *face* $\Rightarrow$ *face*
(*g*,*v*) $\cdot$ *f* $\equiv$ (*let fs* = (*facesAt g v*) *in*
(*case fs of* [] $\Rightarrow$ *f*

```
        | g#gs ⇒ nextElem fs (hd fs) f))⟨proof⟩

constdefs prevFace :: graph × vertex ⇒ face ⇒ face
  (g,v)⁻¹ · f ≡ (let fs = (facesAt g v) in
   (case fs of [] ⇒ f
        | g#gs ⇒ nextElem (rev fs) (last fs) f))⟨proof⟩


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

end
```

5 Vector

```
theory Vector
imports Main EfficientNat
begin

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

5.1 Tabulation

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

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


syntax
 -tabulate :: 'a ⇒ pttrn ⇒ nat ⇒ 'a vector  ((⟦-. - < -⟧))
 -tabulate2 :: 'a ⇒ pttrn ⇒ nat ⇒
    pttrn ⇒ nat ⇒ 'a vector
  ((⟦-. - < -, - < -⟧))
 -tabulate3 :: 'a ⇒ pttrn ⇒ nat ⇒
     pttrn ⇒ nat ⇒
     pttrn ⇒ nat ⇒ 'a vector
  ((⟦-. - < -, - < -, - < - ⟧))
translations
  ⟦f. x < n⟧ == tabulate n (λx. f)
```

$[\![f.\ x < m,\ y < n]\!] == tabulate2\ m\ n\ (\lambda x\ y.\ f)$
$[\![f.\ x < l,\ y < m,\ z < n]\!] == tabulate3\ l\ m\ n\ (\lambda x\ y\ z.\ f)$

5.2 Access

constdefs
sub1 :: *'a vector* × *nat* ⇒ *'a*
sub1 p ≡ *let* (*a*, *n*) = *p in case a of* (*Vector as*) ⇒ *as* ! *n*
sub :: *'a vector* ⇒ *nat* ⇒ *'a*
sub a n ≡ *sub1* (*a*, *n*)

syntax
-sub :: *'a vector* ⇒ *nat* ⇒ *'a* ((-[![-])) [*1000*] *999*)
-sub2 :: *'a vector vector* ⇒ *nat* ⇒ *nat* ⇒ *'a* ((-[![-,-])) [*1000*] *999*)
-sub3 :: *'a vector vector vector* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *'a* ((-[![-,-,-])) [*1000*] *999*)

translations
(*a*[![*n*]!]) == *sub a n*
(*as*[![*m*, *n*]!]) == *sub* (*sub as m*) *n*
(*as*[![*l*, *m*, *n*]!]) == *sub* (*sub* (*sub as l*) *m*) *n*

types-code
vector (- *vector*)

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

examples: $[\![0::'a.\ i < 5]\!]$, $[\![i.\ i < 5,\ j < 3]\!]$

lemma *sub-tabulate*: *0* ≤ *i* ==> *i* < *u* ==>
(*tabulate u f*)[![*i*]!] = *f i*
⟨*proof*⟩

lemma *sub-tabulate3*: *0* ≤ *i* ==> *0* ≤ *j* ==> *0* ≤ *k* ==>
i < *l* ==> *j* < *m* ==> *k* < *n* ==>
(*tabulate3 l m n f*)[![*i*, *j*, *k*]!] = *f i j k*
⟨*proof*⟩

end

6 Enumerating Patches

theory *Enumerator*
imports *Graph Vector*
begin

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

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

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

Generate all lists

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

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

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

As we run over all *inner* and all lists $[a0, \ldots, a_{inner_1}]$, we run over all osibilites fro the finishe face along the edge e inside F.

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

constdefs *enumAppend* :: *nat* ⇒ *nat list list* ⇒ *nat list list*
enumAppend nmax iss ≡ ⋃ *is*∈*iss* [*is* @ [*n*]. *n* ∈ [*last is* .. *nmax*]]

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

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

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

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

consts *hideDups* :: *'a list* ⇒ *'a option list*
primrec *hideDups* [] = []

hideDups (*b*#*bs*) = *Some b* # *hideDupsRec b bs*

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

end

7 Subdividing a Face

theory *FaceDivision*
imports *Graph*
begin

constdefs *split-face* :: *face* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex list* ⇒ *face* × *face*
split-face f ram$_1$ *ram*$_2$ *newVs* ≡ *let vs* = *vertices f*;
f$_1$ = [*ram*$_1$] @ *between vs ram*$_1$ *ram*$_2$ @ [*ram*$_2$];
f$_2$ = [*ram*$_2$] @ *between vs ram*$_2$ *ram*$_1$ @ [*ram*$_1$] *in*
(*Face* (*rev newVs* @ *f*$_1$) *Nonfinal*,
Face (*f*$_2$ @ *newVs*) *Nonfinal*)

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

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

constdefs *makeFaceFinal* :: *face* ⇒ *graph* ⇒ *graph*
makeFaceFinal f g ≡
Graph (*makeFaceFinalFaceList f* (*faces g*))
(*countVertices g*)
[*makeFaceFinalFaceList f fs*. *fs* ∈ *faceListAt g*]
(*heights g*)

constdefs *heightsNewVertices* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat list*
heightsNewVertices h$_1$ *h*$_2$ *n* ≡ [*min* (*h*$_1$ + *i* + *1*) (*h*$_2$ + *n* − *i*). *i* ∈ [*0* ..< *n*]]

constdefs *splitFace*
:: *graph* ⇒ *vertex* ⇒ *vertex* ⇒ *face* ⇒ *vertex list* ⇒ *face* × *face* × *graph*
splitFace g ram$_1$ *ram*$_2$ *oldF newVs* ≡
let fs = *faces g*;
n = *countVertices g*;
Fs = *faceListAt g*;

```
  h = heights g;
  vs1 = between (vertices oldF) ram1 ram2;
  vs2 = between (vertices oldF) ram2 ram1;
  (f1, f2) = split-face oldF ram1 ram2 newVs;
  Fs = replacefacesAt vs1 oldF [f1] Fs;
  Fs = replacefacesAt vs2 oldF [f2] Fs;
  Fs = replacefacesAt [ram1] oldF [f2, f1] Fs;
  Fs = replacefacesAt [ram2] oldF [f1, f2] Fs;
  Fs = Fs @ replicate |newVs| [f1, f2] in
  (f1, f2, Graph ((replace oldF [f2] fs)@ [f1])
                 (n + |newVs| )
                 Fs
                 (h @ heightsNewVertices (h!ram1)(h!ram2) |newVs| ))
```

```
consts subdivFace' :: graph ⇒ face ⇒ vertex ⇒ nat ⇒ vertex option list ⇒ graph
primrec 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];
         (f1, f2, g') = splitFace g u v f ws in
         subdivFace' g' f2 v 0 vos)

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

end
```

8 Transitive Closure of Successor List Function

```
theory RTranCl
imports Main
begin
```

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

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

(- [-]→∗ - [55,0,55] 50)
translations $g\ [succs]\rightarrow* g' == (g,g') \in RTranCl\ succs$

inductive *RTranCl succs*
intros
refl: $g\ [succs]\rightarrow* g$
succs: $g\ [succs]\rightarrow g' \Longrightarrow g'\ [succs]\rightarrow* g'' \Longrightarrow g\ [succs]\rightarrow* g''$

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

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

lemma *RTranCl-induct*:
$(h,\ h') \in RTranCl\ succs \Longrightarrow$
$P\ h \Longrightarrow$
$(\bigwedge g\ g'.\ g' \in set\ (succs\ g) \Longrightarrow P\ g \Longrightarrow P\ g') \Longrightarrow$
$P\ h'$
⟨*proof*⟩

constdefs
invariant :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a\ list) \Rightarrow bool$
$invariant\ P\ succs \equiv \forall g\ g'.\ g' \in set(succs\ g) \longrightarrow P\ g \longrightarrow P\ g'$

lemma *invariantE*:
$invariant\ P\ succs \Longrightarrow g\ [succs]\rightarrow g' \Longrightarrow P\ g \Longrightarrow P\ g'$
⟨*proof*⟩

lemma *inv-subset*:
$invariant\ P\ f \Longrightarrow (!!g.\ P\ g \Longrightarrow set(f'\ g) \subseteq set(f\ g)) \Longrightarrow invariant\ P\ f'$
⟨*proof*⟩

lemma *inv-RTranCl-subset*:
assumes *inv*: $invariant\ P\ succs$ **and** f: $(!!g.\ P\ g \Longrightarrow (g,f\ g) \in RTranCl\ succs)$
shows $invariant\ P\ (\%g.\ [f\ g])$
⟨*proof*⟩

lemma *inv-comp-map*:
$invariant\ P\ succs \Longrightarrow (!!g.\ P\ g \Longrightarrow P(f\ g)) \Longrightarrow invariant\ P\ (map\ f\ o\ succs)$
⟨*proof*⟩

lemma *RTranCl-inv*:
$invariant\ P\ succs \Longrightarrow (g,g') \in RTranCl\ succs \Longrightarrow P\ g \Longrightarrow P\ g'$
⟨*proof*⟩

lemma *RTranCl-subset*: $(!!g.\ set(f\ g) \subseteq set(h\ g)) \Longrightarrow$
$(s,g) : RTranCl\ f \Longrightarrow (s,g) : RTranCl\ h$
⟨*proof*⟩

lemma *RTranCl-subset2*:

assumes a: $(s,g) : RTranCl\ f$
shows $(!!g.\ (s,g) \in RTranCl\ f \Longrightarrow set(f\ g) \subseteq set(h\ g)) \Longrightarrow (s,g) : RTranCl\ h$
⟨*proof*⟩

lemma *RTranCl-rev-succs*:
$(g,\ g') \in RTranCl\ succs \Longrightarrow g'' \in set\ (succs\ g') \Longrightarrow (g,\ g'') \in RTranCl\ succs$
⟨*proof*⟩

lemma *RTranCl-compose*:
assumes $(g,g') \in RTranCl\ succs$
shows $(g',g'') \in RTranCl\ succs \Longrightarrow (g,g'') \in RTranCl\ succs$
⟨*proof*⟩

lemma *RTranCl-map-comp-subset*:
assumes *inv*: *invariant P succs*
and f: $(!!g.\ P\ g \Longrightarrow (g,f\ g) \in RTranCl\ succs)$
and a: $(s,g) \in RTranCl\ (map\ f\ o\ succs)$
shows $P\ s \Longrightarrow (s,g) \in RTranCl\ succs$
⟨*proof*⟩

end

9 Plane Graph Enumeration

theory *Plane*
imports *Enumerator FaceDivision RTranCl*
begin

constdefs
maxGon :: $nat \Rightarrow nat$
$maxGon\ p \equiv p+3$

declare *maxGon-def* [*simp*]

constdefs *duplicateEdge* :: *graph* ⇒ *face* ⇒ *vertex* ⇒ *vertex* ⇒ *bool*
duplicateEdge g f a b ≡
$2 \leq directedLength\ f\ a\ b \wedge 2 \leq directedLength\ f\ b\ a \wedge b\ mem\ (neighbors\ g\ a)$

consts *containsUnacceptableEdgeSnd* ::
$(nat \Rightarrow nat \Rightarrow bool) \Rightarrow nat \Rightarrow nat\ list \Rightarrow bool$
primrec $containsUnacceptableEdgeSnd\ N\ v\ [] = False$
$containsUnacceptableEdgeSnd\ N\ v\ (w\#ws) =$
$(case\ ws\ of\ [] \Rightarrow False$

$$\mid (w'\#ws') \Rightarrow \mathit{if}\ v < w \wedge w < w' \wedge N\ w\ w'\ \mathit{then}\ \mathit{True}$$
$$\mathit{else}\ \mathit{containsUnacceptableEdgeSnd}\ N\ w\ ws)$$

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

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

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

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

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

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

lemma $\mathit{Seed\text{-}not\text{-}final}[\mathit{iff}]$: $\neg\ \mathit{final}\ (\mathit{Seed}\ p)$
$\langle \mathit{proof} \rangle$

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

end

theory *Plane1*
imports *Plane*
begin

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

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

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

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

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

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

end

10 Properties of Graph Utilities

theory *GraphProps*
imports *Graph*
begin

⟨*ML*⟩

lemma *final-setFinal*[*iff*]: *final*(*setFinal f*)
⟨*proof*⟩

lemma *eq-setFinal-iff*[*iff*]: (*f* = *setFinal f*) = *final f*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *distinct-vertices*[*iff*]: *distinct*(*vertices*(*g*::*graph*))
⟨*proof*⟩

10.1 *nextElem*

lemma *nextElem-append*[*simp*]:
y ∉ *set xs* ⟹ *nextElem* (*xs* @ *ys*) *d y* = *nextElem ys d y*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *nextElem-notin-butlast*[*rule-format*,*simp*]:
y ∉ *set*(*butlast xs*) ⟶ *nextElem xs x y* = *x*
⟨*proof*⟩

lemma *nextElem-in*: *nextElem xs x y* : *set*(*x*#*xs*)
⟨*proof*⟩

lemma *nextElem-notin*[*simp*]: *a* ∉ *set as* ⟹ *nextElem as c a* = *c*
⟨*proof*⟩

lemma *nextElem-last*[*simp*]: **assumes** *dist*: *distinct xs*
shows *nextElem xs c* (*last xs*) = *c*
⟨*proof*⟩

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*⟩

lemma *nextElem-prevElem*:
⟦ *distinct xs*; *x* : *set xs* ⟧ ⟹
nextElem xs (*hd xs*) (*nextElem* (*rev xs*) (*last xs*) *x*) = *x*
⟨*proof*⟩

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*))

⟨*proof*⟩

10.2 *nextVertex*

lemma *nextVertex-in-face'*[*simp*]:
$vertices\ f \neq [] \Longrightarrow f \cdot v \in \mathcal{V}\ f$
⟨*proof*⟩

lemma *nextVertex-in-face*[*simp*]:
$v \in set\ (vertices\ f) \Longrightarrow f \cdot v \in \mathcal{V}\ f$
⟨*proof*⟩

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

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

lemma *prevVertex-in-face*[*simp*]:
$v \in \mathcal{V}\ f \Longrightarrow f^{-1} \cdot v \in \mathcal{V}\ f$
⟨*proof*⟩

lemma *nextVertex-nth*:
$[\![\ distinct(vertices\ f);\ i < |vertices\ f|\ ]\!] \Longrightarrow$
$f \cdot (vertices\ f\ !\ i) = vertices\ f\ !\ ((i+1)\ mod\ |vertices\ f|\ )$
⟨*proof*⟩

10.3 $\mathcal{E}$

lemma *finite-edges*: $finite(\mathcal{E}(f::face))$
⟨*proof*⟩

lemma *edges-face-eq*:
$((a,b) \in \mathcal{E}\ (f::face)) = ((f \cdot a = b) \wedge a \in \mathcal{V}\ f)$
⟨*proof*⟩

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

lemma *in-edges-in-vertices*:
$(x,y) \in \mathcal{E}(f::face) \Longrightarrow x \in \mathcal{V}\ f \wedge y \in \mathcal{V}\ f$
⟨*proof*⟩

lemma *vertices-conv-Union-edges*:
$\mathcal{V}(f{::}face) = (\bigcup (a,b){\in}\mathcal{E}\ f.\ \{a\})$
⟨*proof*⟩

lemma *nextVertex-in-edges*: $v \in \mathcal{V}\ f \Longrightarrow (v, f \cdot v) \in edges\ f$
⟨*proof*⟩

lemma *prevVertex-in-edges*:
$[\![distinct(vertices\ f);\ v \in \mathcal{V}\ f]\!] \Longrightarrow (f^{-1} \cdot v,\ v) \in edges\ f$
⟨*proof*⟩

10.4 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*⟩

lemma *tri-next3-id*:
$|vertices\ f| = 3 \Longrightarrow distinct(vertices\ f) \Longrightarrow v \in \mathcal{V}\ f$
$\Longrightarrow f \cdot (f \cdot (f \cdot v)) = v$
⟨*proof*⟩

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*⟩

10.5 Quadrilaterals

lemma *vertices-quad*:
$|vertices\ f| = 4 \Longrightarrow a \in \mathcal{V}\ f \Longrightarrow$
$distinct\ (vertices\ f) \Longrightarrow$
$\mathcal{V}\ f = \{a, f \cdot a, f \cdot (f \cdot a), f \cdot (f \cdot (f \cdot a))\}$
⟨*proof*⟩

lemma *quad-next4-id*:
$[\![\ |vertices\ f| = 4;\ distinct(vertices\ f);\ v \in \mathcal{V}\ f\ ]\!] \Longrightarrow$
$f \cdot (f \cdot (f \cdot (f \cdot v))) = v$
⟨*proof*⟩

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*⟩

lemma *C0*[*dest*]: $f \in set\ (facesAt\ g\ v) \Longrightarrow v \in \mathcal{V}\ g$
⟨*proof*⟩

lemma *len-faces-sum*: $|faces\ g| = |finals\ g| + |nonFinals\ g|$
⟨*proof*⟩

lemma *graph-max-final-ex*:
$\exists f{\in}set\ (finals\ (graph\ n)).\ |vertices\ f| = n$
⟨*proof*⟩

10.6 No loops

lemma *distinct-no-loop2*:
$[\![\ distinct(vertices\ f);\ v \in \mathcal{V}\ f;\ u \in \mathcal{V}\ f;\ u \neq v\ ]\!] \Longrightarrow f \cdot v \neq v$
⟨*proof*⟩

lemma *distinct-no-loop1*:
$[\![\ distinct(vertices\ f);\ v \in \mathcal{V}\ f;\ |vertices\ f| > 1\ ]\!] \Longrightarrow f \cdot v \neq v$
⟨*proof*⟩

10.7 *between*

lemma *between-front*[*simp*]:
$v \notin set\ us \Longrightarrow between\ (u\ \#\ us\ @\ v\ \#\ vs)\ u\ v = us$
⟨*proof*⟩

lemma *between-back*:
$[\![\ v \notin set\ us;\ u \notin set\ vs;\ v \neq u\ ]\!] \Longrightarrow between\ (v\ \#\ vs\ @\ u\ \#\ us)\ u\ v = us$
⟨*proof*⟩

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

lemma *next-between2*:
$[\![\ distinct(vertices\ f);\ v \in \mathcal{V}\ f;\ u \in \mathcal{V}\ f;\ u \neq v\ ]\!] \Longrightarrow$
$v \in set(between\ (vertices\ f)\ u\ (f \cdot v))$
⟨*proof*⟩

lemma *between-next-empty*:

$distinct(vertices\ f) \Longrightarrow between\ (vertices\ f)\ v\ (f \cdot v) = []$
⟨*proof*⟩

lemma *unroll-between-next*:
$[\![\ distinct(vertices\ f);\ u \in \mathcal{V}\ f;\ v \in \mathcal{V}\ f;\ f \cdot v \neq u\ ]\!] \Longrightarrow$
$between\ (vertices\ f)\ v\ u = f \cdot v\ \#\ between\ (vertices\ f)\ (f \cdot v)\ u$
⟨*proof*⟩

lemma *unroll-between-next2*:
$[\![\ distinct(vertices\ f);\ u \in \mathcal{V}\ f;\ v \in \mathcal{V}\ f;\ u \neq v\ ]\!] \Longrightarrow$
$between\ (vertices\ f)\ u\ (f \cdot v) = between\ (vertices\ f)\ u\ v\ @\ [v]$
⟨*proof*⟩

lemma *nextVertex-eq-lemma*:
$[\![\ distinct(vertices\ f);\ x \in \mathcal{V}\ f;\ y \in \mathcal{V}\ f;\ x \neq y;$
$v \in set(x\ \#\ between\ (vertices\ f)\ x\ y)\ ]\!] \Longrightarrow$
$f \cdot v = nextElem\ (x\ \#\ between\ (vertices\ f)\ x\ y\ @\ [y])\ z\ v$
⟨*proof*⟩

lemma *nextVertex-eq-if-between-eq*:
$[\![\ between\ (vertices\ f)\ x\ y = between\ (vertices\ f')\ x\ y;$
$distinct(vertices\ f);\ distinct(vertices\ f');\ x \neq y;$
$x \in \mathcal{V}\ f;\ y \in \mathcal{V}\ f;\ x \in \mathcal{V}\ f';\ y \in \mathcal{V}\ f';$
$v \in set(x\ \#\ between\ (vertices\ f)\ x\ y)\ ]\!] \Longrightarrow$
$f \cdot v = f' \cdot v$
⟨*proof*⟩

end

11 Properties of Patch Enumeration

theory *EnumeratorProps*
imports *Enumerator GraphProps*
begin

lemma *length-hideDupsRec*[*simp*]: $\bigwedge x.\ length(hideDupsRec\ x\ xs) = length\ xs$
⟨*proof*⟩

lemma *length-hideDups*[*simp*]: $length(hideDups\ xs) = length\ xs$
⟨*proof*⟩

lemma *length-indexToVertexList*[*simp*]:
$length(indexToVertexList\ x\ y\ xs) = length\ xs$
⟨*proof*⟩

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

lemma *increasing1*: ⋀ *as x*. *increasing ls* ⟹ *ls* = *as* @ *x* # *cs* @ *y* # *bs* ⟹ *x* ≤ *y*
⟨*proof*⟩

lemma *increasing2*: *increasing* (*as*@*bs*) ⟹ *x* ∈ *set as* ⟹ *y* ∈ *set bs* ⟹ *x* ≤ *y*
⟨*proof*⟩

lemma *increasing3*: ∀ *as bs*. (*ls* = *as* @ *bs* ⟶ (∀ *x* ∈ *set as*. ∀ *y* ∈ *set bs*. *x* ≤ *y*)) ⟹ *increasing* (*ls*)
⟨*proof*⟩

lemma *increasing4*: *increasing* (*as*@*bs*) ⟹ *increasing as*
⟨*proof*⟩

lemma *increasing5*: *increasing* (*as*@*bs*) ⟹ *increasing bs*
⟨*proof*⟩

lemma *enumBase-length*: *ls* ∈ *set* (*enumBase nmax*) ⟹ *length ls* = *1*
⟨*proof*⟩

lemma *enumBase-lenght2*: ∀ *ls* ∈ *set* (*enumBase nmax*). *length ls* = *1*
⟨*proof*⟩

lemma *enumBase-bound*: ∀ *y* ∈ *set* (*enumBase nmax*). ∀ *z* ∈ *set y*. *z* ≤ *nmax*
⟨*proof*⟩

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

lemma *enumAppend-length1*: (∀ *ls* ∈ *set lss*. *length ls* = *k*) ⟹ *ls2* ∈ *set* (*enumAppend nmax lss*) ⟹ *length ls2* = *k* + *1*
⟨*proof*⟩

lemma *enumAppend-length*: (∀ *ls* ∈ *set lss*. *length ls* = *k*) ⟹ (∀ *ls2* ∈ *set* (*enumAppend nmax lss*). *length ls2* = *Suc k*)
⟨*proof*⟩

lemma *enumAppend-length-rec*: $(\forall\ ls \in set\ lss.\ length\ ls = k) \Longrightarrow$
$(\forall\ ls2 \in set\ (((enumAppend\ nmax)\ \hat{}n)\ lss).\ \ length\ ls2 = k + n)$
⟨*proof*⟩

lemma *enumAppend-length-rec2*: $(\forall\ ls \in set\ lss.\ length\ ls = k)$
$\Longrightarrow ls2 \in set\ (((enumAppend\ nmax)\ \hat{}n)\ lss) \Longrightarrow\ \ length\ ls2 = k + n$
⟨*proof*⟩

lemma *enumAppend-length-rec3*: $(\forall\ ls \in set\ lss.\ length\ ls = 1)$
$\Longrightarrow ls2 \in set\ (((enumAppend\ nmax)\ \hat{}n)\ lss) \Longrightarrow\ \ length\ ls2 = Suc\ n$
⟨*proof*⟩

lemma *enumAppend-bound*: $ls \in set\ ((enumAppend\ nmax)\ lss) \Longrightarrow$
$\forall\ y \in set\ lss.\ \forall\ z \in set\ y.\ z \leq nmax \Longrightarrow x \in set\ ls \Longrightarrow x \leq\ \ nmax$
⟨*proof*⟩

lemma *enumAppend-bound-rec*: $ls \in set\ (((enumAppend\ nmax)\ \hat{}n)\ lss) \Longrightarrow$
$\forall\ y \in set\ lss.\ \forall\ z \in set\ y.\ z \leq nmax \Longrightarrow x \in set\ ls \Longrightarrow x \leq\ \ nmax$
⟨*proof*⟩

lemma *enumAppend-increase-rec*:
$\bigwedge m\ as\ bs.\ ls \in set\ (((enumAppend\ nmax)\ \hat{}\ m)\ (enumBase\ nmax)) \Longrightarrow$
$as\ @\ bs = ls \Longrightarrow\ \forall\ x \in set\ as.\ \forall\ y \in set\ bs.\ x \leq y$
⟨*proof*⟩

lemma *enumAppend-length1*: $\bigwedge ls.\ ls \in set\ (((enumAppend\ nmax)\ \hat{}n)\ lss) \Longrightarrow$
$(\forall\, l \in set\ lss.\ |l| = k) \Longrightarrow |ls| = k + n$
⟨*proof*⟩

lemma *enumAppend-length2*: $\bigwedge ls.\ ls \in set\ (((enumAppend\ nmax)\ \hat{}n)\ lss) \Longrightarrow$
$(\bigwedge l.\ l \in set\ lss \Longrightarrow |l| = k) \Longrightarrow K = k + n \Longrightarrow |ls| = K$
⟨*proof*⟩

lemma *enum-enumerator*:
$enum\ i\ j = enumerator\ i\ j$
⟨*proof*⟩

lemma *enumerator-hd*: $ls \in set\ (enumerator\ m\ n) \Longrightarrow hd\ ls = 0$
⟨*proof*⟩

lemma *enumerator-last*: $ls \in set\ (enumerator\ m\ n) \Longrightarrow last\ ls = (n - 1)$

⟨*proof*⟩

lemma *enumerator-length*: *ls* ∈ *set* (*enumerator m n*) ⟹ *2* ≤ *length ls*
⟨*proof*⟩

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

lemma *enumerator-not-empty*[*dest*]: *ls* ∈ *set* (*enumerator m n*) ⟹ *ls* ≠ []
⟨*proof*⟩

lemma *enumerator-length2*: *ls* ∈ *set* (*enumerator m n*) ⟹ *2* < *m* ⟹ *length ls* = *m*
⟨*proof*⟩

lemma *enumerator-bound*: *ls* ∈ *set* (*enumerator m nmax*) ⟹
0 < *nmax* ⟹ *x* ∈ *set ls* ⟹ *x* < *nmax*
⟨*proof*⟩

lemma *enumerator-bound2*: *ls* ∈ *set* (*enumerator m nmax*) ⟹ *1* < *nmax* ⟹ *x* ∈ *set* (*butlast ls*) ⟹ *x* < *nmax* − *Suc 0*
⟨*proof*⟩

lemma *enumerator-bound3*: *ls* ∈ *set* (*enumerator m nmax*) ⟹ *1* < *nmax* ⟹ *last* (*butlast ls*) < *nmax* − *Suc 0*
⟨*proof*⟩

lemma *enumerator-increase*: ⋀ *as bs*. *ls* ∈ *set* (*enumerator m nmax*) ⟹ *as* @ *bs* = *ls* ⟹ ∀ *x* ∈ *set as*. ∀ *y* ∈ *set bs*. *x* ≤ *y*
⟨*proof*⟩

lemma *enumerator-increasing*: *ls* ∈ *set* (*enumerator m nmax*) ⟹ *increasing ls*
⟨*proof*⟩

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

lemma *incrIndexList-1lem*[*simp*]: *incrIndexList ls m nmax* ⟹ *Suc 0* < *m*
⟨*proof*⟩

lemma *incrIndexList-1len*[*simp*]: *incrIndexList ls m nmax* ⟹ *Suc 0* < *nmax*
⟨*proof*⟩

lemma *incrIndexList-help2*[*simp*]: *incrIndexList ls m nmax* ⟹ *hd ls* = *0*
⟨*proof*⟩

lemma *incrIndexList-help21*[*simp*]: *incrIndexList* (*l* # *ls*) *m nmax* ⟹ *l* = *0*
⟨*proof*⟩

lemma *incrIndexList-help3*[*simp*]: *incrIndexList ls m nmax* ⟹ *last ls* = (*nmax* − (*Suc 0*))
⟨*proof*⟩

lemma *incrIndexList-help4*[*simp*]: *incrIndexList ls m nmax* ⟹ *length ls* = *m*
⟨*proof*⟩

lemma *incrIndexList-help5*[*intro*]: *incrIndexList ls m nmax* ⟹ *last* (*butlast ls*) < *nmax* − *Suc 0*
⟨*proof*⟩

lemma *incrIndexList-help6*[*simp*]: *incrIndexList ls m nmax* ⟹ *increasing ls*
⟨*proof*⟩

lemma *incrIndexList-help7*[*simp*]: *incrIndexList ls m nmax* ⟹ *ls* ≠ []
⟨*proof*⟩

lemma *incrIndexList-help71*[*simp*]: ¬ *incrIndexList* [] *m nmax*
⟨*proof*⟩

lemma *incrIndexList-help8*[*simp*]: *incrIndexList ls m nmax* ⟹ *butlast ls* ≠ []
⟨*proof*⟩

lemma *incrIndexList-help81*[*simp*]: ¬ *incrIndexList* [*l*] *m nmax*
⟨*proof*⟩

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

lemma *incrIndexList-help10*[*intro*]: (*incrIndexList ls m nmax*) ⟹
x ∈ *set ls* ⟹ *x* < *nmax* ⟨*proof*⟩

lemma *enumerator-correctness*: *2* < *m* ⟹ *1* < *nmax* ⟹
ls ∈ *set* (*enumerator m nmax*) ⟹
incrIndexList ls m nmax
⟨*proof*⟩

lemma *enumerator-completeness-help*: ⋀ *ls*. *increasing ls* ⟹ *ls* ≠ [] ⟹ *length ls* = *Suc ks* ⟹ *list-all* (λ*x*. *x* < *Suc nmax*) *ls* ⟹ *ls* ∈ *set* ((*enumAppend nmax* ^ *ks*) (*enumBase nmax*))
⟨*proof*⟩

lemma *enumerator-completeness*: *2* < *m* ⟹ *incrIndexList ls m nmax* ⟹
ls ∈ *set* (*enumerator m nmax*)

⟨*proof*⟩

lemma *enumerator-equiv*[*simp*]:
$2 < n \Longrightarrow 1 < m \Longrightarrow is \in set(enumerator\ n\ m) = incrIndexList\ is\ n\ m$
⟨*proof*⟩

end

12 Properties of Face Division

theory *FaceDivisionProps*
imports *Plane EnumeratorProps*
begin

12.1 Finality

lemma *vertices-makeFaceFinal*: $vertices(makeFaceFinal\ f\ g) = vertices\ g$
⟨*proof*⟩

lemma *edges-makeFaceFinal*: $\mathcal{E}\ (makeFaceFinal\ f\ g) = \mathcal{E}\ g$
⟨*proof*⟩

lemma *nonFins-mkFaceFin*:
$nonFinals(makeFaceFinal\ f\ g) = remove1\ f\ (nonFinals\ g)$
⟨*proof*⟩

lemma *in-set-repl-setFin*:
$f \in set\ fs \Longrightarrow final\ f \Longrightarrow f \in set\ (replace\ f'\ [setFinal\ f']\ fs)$
⟨*proof*⟩

lemma *in-set-repl*: $f \in set\ fs \Longrightarrow f \neq f' \Longrightarrow f \in set\ (replace\ f'\ fs'\ fs)$
⟨*proof*⟩

lemma *makeFaceFinals-preserve-finals*:
$f \in set\ (finals\ g) \Longrightarrow f \in set\ (finals\ (makeFaceFinal\ f'\ g))$
⟨*proof*⟩

lemma *len-faces-makeFaceFinal*[*simp*]:
$|faces\ (makeFaceFinal\ f\ g)| = |faces\ g|$
⟨*proof*⟩

lemma *len-finals-makeFaceFinal*:
$f \in \mathcal{F}\ g \Longrightarrow \neg\ final\ f \Longrightarrow |finals\ (makeFaceFinal\ f\ g)| = |finals\ g| + 1$
⟨*proof*⟩

lemma *len-nonFinals-makeFaceFinal*:
$[\![\neg \mathit{final}\ f; f \in \mathcal{F}\ g]\!]$
$\Longrightarrow |\mathit{nonFinals}\ (\mathit{makeFaceFinal}\ f\ g)| = |\mathit{nonFinals}\ g| - 1$
⟨*proof*⟩

lemma *set-finals-makeFaceFinal*[*simp*]: $\mathit{distinct}(\mathit{faces}\ g) \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow$
$\mathit{set}(\mathit{finals}\ (\mathit{makeFaceFinal}\ f\ g)) = \mathit{insert}\ (\mathit{setFinal}\ f)\ (\mathit{set}(\mathit{finals}\ g))$
⟨*proof*⟩

lemma *splitFace-preserve-final*:
$f \in \mathit{set}\ (\mathit{finals}\ g) \Longrightarrow \neg\ \mathit{final}\ f' \Longrightarrow$
$f \in \mathit{set}\ (\mathit{finals}\ (\mathit{snd}\ (\mathit{snd}\ (\mathit{splitFace}\ g\ i\ j\ f'\ ns))))$
⟨*proof*⟩

lemma *splitFace-nonFinal-face*:
$\neg\ \mathit{final}\ (\mathit{fst}\ (\mathit{snd}\ (\mathit{splitFace}\ g\ i\ j\ f'\ ns)))$
⟨*proof*⟩

lemma *subdivFace′-preserve-finals*:
$\bigwedge n\ i\ f'\ g.\ f \in \mathit{set}\ (\mathit{finals}\ g) \Longrightarrow\ \neg\ \mathit{final}\ f' \Longrightarrow$
$f \in \mathit{set}\ (\mathit{finals}\ (\mathit{subdivFace}'\ g\ f'\ i\ n\ is))$
⟨*proof*⟩

lemma *subdivFace-pres-finals*:
$f \in \mathit{set}\ (\mathit{finals}\ g) \Longrightarrow \neg\ \mathit{final}\ f' \Longrightarrow$
$f \in \mathit{set}\ (\mathit{finals}\ (\mathit{subdivFace}\ g\ f'\ is))$
⟨*proof*⟩

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

12.2 *is-prefix*

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

lemma *is-prefix-add*:
is-prefix ls vs ⟹ *is-prefix* (*as* @ *ls*) (*as* @ *vs*) ⟨*proof*⟩

lemma *is-prefix-hd*[*simp*]:
is-prefix [*l*] *vs* = (*l* = *hd vs* ∧ *vs* ≠ [])
⟨*proof*⟩

lemma *is-prefix-f*[*simp*]:
is-prefix (*a*#*as*) (*a*#*vs*) = *is-prefix as vs* ⟨*proof*⟩

lemma *splitAt-is-prefix*: *ram* ∈ *set vs* ⟹ *is-prefix* (*fst* (*splitAt ram vs*) @ [*ram*]) *vs*
⟨*proof*⟩

12.3 *is-postfix*

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

lemma *is-postfix-add*:
is-postfix ls vs ⟹ *is-postfix* (*ls* @ *bs*) (*vs* @ *bs*) ⟨*proof*⟩

lemma *is-pre-post-eq*:
distinct vs ⟹ *ls* ≠ [] ⟹ *is-prefix ls vs* ⟹ *is-postfix ls vs* ⟹ *ls* = *vs*
⟨*proof*⟩

lemma *splitAt-is-postfix*: *ram* ∈ *set vs* ⟹ *is-postfix* (*ram* # *snd* (*splitAt ram vs*)) *vs*
⟨*proof*⟩

12.4 *is-sublist*

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

lemma *is-prefix-sublist*:
is-prefix ls vs ⟹ *is-sublist ls vs* ⟨*proof*⟩

lemma *is-postfix-sublist*:
is-postfix ls vs ⟹ *is-sublist ls vs* ⟨*proof*⟩

lemma *is-sublist-trans*: *is-sublist as bs* ⟹ *is-sublist bs cs* ⟹ *is-sublist as cs*
⟨*proof*⟩

lemma *is-sublist-add*: *is-sublist as bs* ⟹ *is-sublist as* (*xs* @ *bs* @ *ys*)
⟨*proof*⟩

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*⟩

lemma *not-sublist-len*[*simp*]:
|*ys*| < |*xs*| ⟹ ¬ *is-sublist xs ys*
⟨*proof*⟩

lemma *is-sublist-simp*[*simp*]: $a \neq v \Longrightarrow$ *is-sublist* $(a\#as)$ $(v\#vs)$ = *is-sublist* $(a\#as)$ *vs*
⟨*proof*⟩

lemma *is-sublist-id*[*simp*]: *is-sublist vs vs* ⟨*proof*⟩

lemma *is-sublist-in*: *is-sublist* $(a\#as)$ *vs* $\Longrightarrow$ $a \in$ *set vs* ⟨*proof*⟩

lemma *is-sublist-in1*: *is-sublist* [*x*,*y*] *vs* $\Longrightarrow$ $y \in$ *set vs* ⟨*proof*⟩

lemma *is-sublist-notlast*[*simp*]: *distinct vs* $\Longrightarrow$ $x =$ *last vs* $\Longrightarrow$ ¬ *is-sublist* [*x*,*y*] *vs*
⟨*proof*⟩

lemma *is-sublist-nth1*: *is-sublist* [*x*,*y*] *ls* $\Longrightarrow$
$\exists\ i\ j.\ i < length\ ls \wedge j < length\ ls \wedge ls!i = x \wedge ls!j = y \wedge Suc\ i = j$
⟨*proof*⟩

lemma *is-sublist-nth2*: $\exists\ i\ j.\ i < length\ ls \wedge j < length\ ls \wedge ls!i = x \wedge ls!j = y \wedge Suc\ i = j \Longrightarrow$
is-sublist [*x*,*y*] *ls*
⟨*proof*⟩

lemma *is-sublist-nth-eq*: $(\exists\ i\ j.\ i < length\ ls \wedge j < length\ ls \wedge ls!i = x \wedge ls!j = y \wedge Suc\ i = j)$ = *is-sublist* [*x*,*y*] *ls*
⟨*proof*⟩

lemma *is-sublist-tl*: *is-sublist* $(a\ \#\ as)$ *vs* $\Longrightarrow$ *is-sublist as vs* ⟨*proof*⟩

lemma *is-sublist-hd*: *is-sublist* $(a\ \#\ as)$ *vs* $\Longrightarrow$ *is-sublist* [*a*] *vs* ⟨*proof*⟩

lemma *is-sublist-hd-eq*[*simp*]: (*is-sublist* [*a*] *vs*) = ($a \in$ *set vs*) ⟨*proof*⟩

lemma *is-sublist-distinct-prefix*:
is-sublist $(v\#as)$ $(v\ \#\ vs)$ $\Longrightarrow$ *distinct* $(v\ \#\ vs)$ $\Longrightarrow$ *is-prefix as vs*
⟨*proof*⟩

lemma *is-sublist-distinct*[*intro*]:
is-sublist as vs $\Longrightarrow$ *distinct vs* $\Longrightarrow$ *distinct as* ⟨*proof*⟩

lemma *is-sublist-x-last*: *distinct vs* $\Longrightarrow$ $x =$ *last vs* $\Longrightarrow$ ¬ *is-sublist* [*x*,*y*] *vs*
⟨*proof*⟩

lemma *is-sublist-y-hd*: *distinct vs* $\Longrightarrow$ $y =$ *hd vs* $\Longrightarrow$ ¬ *is-sublist* [*x*,*y*] *vs*
⟨*proof*⟩

lemma *is-sublist-at1*: *distinct* $(as\ @\ bs)$ $\Longrightarrow$ *is-sublist* [*x*,*y*] $(as\ @\ bs)$ $\Longrightarrow$ $x \neq (last\ as)$ $\Longrightarrow$
is-sublist [*x*,*y*] *as* ∨ *is-sublist* [*x*,*y*] *bs*

⟨*proof*⟩

lemma *is-sublist-at2*: *distinct* (*as* @ *bs*) ⟹ *is-sublist* [*x*,*y*] (*as* @ *bs*) ⟹ *x* ≠ (*last as*) ⟹
is-sublist [*x*,*y*] *as* ≠ *is-sublist* [*x*,*y*] *bs*
⟨*proof*⟩

lemma *is-sublist-at3*: *distinct* (*as* @ *bs*) ⟹ *is-sublist* [*x*,*y*] (*as* @ *bs*) ⟹
(*is-sublist* [*x*,*y*] *as* ∨ *is-sublist* [*x*,*y*] *bs*) ∨ *x* = *last as*
⟨*proof*⟩

lemma *is-sublist-at4*: *distinct* (*as* @ *bs*) ⟹ *is-sublist* [*x*,*y*] (*as* @ *bs*) ⟹
as ≠ [] ⟹ *x* = *last as* ⟹ *y* = *hd bs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is-sublist-rev*: *is-sublist* [*a*,*b*] (*rev zs*) = *is-sublist* [*b*,*a*] *zs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *splitAt-is-sublist1*: *is-sublist* (*fst* (*splitAt ram vs*)) *vs* ⟨*proof*⟩

lemma *splitAt-is-sublist1R*[*simp*]: *ram* ∈ *set vs* ⟹ *is-sublist* (*fst* (*splitAt ram vs*) @ [*ram*]) *vs*
⟨*proof*⟩

lemma *splitAt-is-sublist2*:*is-sublist* (*snd* (*splitAt ram vs*)) *vs* ⟨*proof*⟩

lemma *splitAt-is-sublist2R*[*simp*]: *ram* ∈ *set vs* ⟹ *is-sublist* (*ram* # *snd* (*splitAt ram vs*)) *vs*
⟨*proof*⟩

12.5 *is-nextElem*

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

lemma *is-nextElem-a*[*intro*]: *is-nextElem vs a b* ⟹ *a* ∈ *set vs*
⟨*proof*⟩
lemma *is-nextElem-b*[*intro*]: *is-nextElem vs a b* ⟹ *b* ∈ *set vs*

⟨*proof*⟩
lemma *is-nextElem-sublist*: *is-nextElem vs x y* ⟹ *x* ≠ *last vs* ⟹ *is-sublist* [*x*,*y*] *vs*
⟨*proof*⟩
lemma *is-nextElem-last-hd*[*intro*]: *distinct vs* ⟹ *is-nextElem vs x y* ⟹
x = *last vs* ⟹ *y* = *hd vs*
⟨*proof*⟩
lemma *is-nextElem-last-ne*[*intro*]: *distinct vs* ⟹ *is-nextElem vs x y* ⟹
x = *last vs* ⟹ *vs* ≠ []
⟨*proof*⟩
lemma *is-nextElem-sublist2*: *is-nextElem vs x y* ⟹ *y* ≠ *hd vs* ⟹ *is-sublist* [*x*,*y*] *vs*
⟨*proof*⟩
lemma *is-nextElem-sublistI*: *is-sublist* [*x*,*y*] *vs* ⟹ *is-nextElem vs x y*
⟨*proof*⟩
lemma *is-nextElem-hd-lastI*: *vs* ≠ [] ⟹ *y* = *hd vs* ⟹ *x* = *last vs* ⟹ *is-nextElem vs x y*
⟨*proof*⟩

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*⟩

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*⟩

lemma *is-nextElem-rotate1*:
is-nextElem (*rotate m ls*) *x y* ⟹ *is-nextElem ls x y*
⟨*proof*⟩

lemma *is-nextElem-rotate-eq*[*simp*]: *is-nextElem* (*rotate m ls*) *x y* = *is-nextElem ls x y*
⟨*proof*⟩

lemma *is-nextElem-congs-eq*: *ls* ≅ *ms* ⟹ *is-nextElem ls x y* = *is-nextElem ms x y*
⟨*proof*⟩

lemma *is-nextElem-rotate1*: *distinct ls* ⟹ *is-nextElem ls x y* ⟹ *is-nextElem* (*rotate1 ls*) *x y*
⟨*proof*⟩

lemma *is-nextElem-congsI*: *ls1* ≅ *ls2* ⟹ *distinct ls1* ⟹ *is-nextElem ls1 x y* ⟹ *is-nextElem ls2 x y*
⟨*proof*⟩

lemma *is-nextElem-congseq*: *ls1* ≅ *ls2* ⟹ *distinct ls1* ⟹ *is-nextElem ls1 x y* =

is-nextElem ls2 x y
⟨*proof*⟩

lemma *is-nextElem-rev*[*simp*]: *is-nextElem* (*rev zs*) *a b* = *is-nextElem zs b a*
⟨*proof*⟩

lemma *is-nextElem-circ*:
⟦ *distinct xs*; *is-nextElem xs a b*; *is-nextElem xs b a* ⟧ ⟹ |*xs*| ≤ *2*
⟨*proof*⟩

lemma *cong-if-is-nextElem-eq*:
⟦ *distinct xs*; *distinct ys*; *set xs* = *set ys*;
!*x y*. *is-nextElem xs x y* = *is-nextElem ys x y* ⟧ ⟹
xs ≅ *ys*
⟨*proof*⟩

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

lemma *is-sublist-eq*: *distinct vs* ⟹ *c* ≠ *y* ⟹
(*nextElem vs c x* = *y*) = *is-sublist* [*x*,*y*] *vs*
⟨*proof*⟩

lemma *is-nextElem1*: *distinct vs* ⟹ *x* ∈ *set vs* ⟹ *nextElem vs* (*hd vs*) *x* = *y*
⟹ *is-nextElem vs x y*
⟨*proof*⟩

lemma *is-nextElem2*: *distinct vs* ⟹ *x* ∈ *set vs* ⟹ *is-nextElem vs x y* ⟹ *nextElem vs* (*hd vs*) *x* = *y*
⟨*proof*⟩

lemma *nextElem-is-nextElem*:
distinct xs ⟹ *x* ∈ *set xs* ⟹
is-nextElem xs x y = (*nextElem xs* (*hd xs*) *x* = *y*)
⟨*proof*⟩

lemma *nextElem-congs-eq*: *xs* ≅ *ys* ⟹ *distinct xs* ⟹ *x* ∈ *set xs* ⟹
nextElem xs (*hd xs*) *x* = *nextElem ys* (*hd ys*) *x*
⟨*proof*⟩

lemma *nextElem-iso-eq*:
as ≅ *as'* ⟹ *distinct as* ⟹ *x* ∈ *set as* ⟹
nextElem as (*hd as*) *x* = *nextElem as'* (*hd as'*) *x*
⟨*proof*⟩

lemma *is-sublist-is-nextElem*: *distinct vs* ⟹ *is-nextElem vs x y* ⟹ *is-sublist as vs* ⟹ *x* ∈ *set as* ⟹ *x* ≠ *last as* ⟹ *is-sublist* [*x*,*y*] *as*

⟨*proof*⟩

12.7 *before*

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

lemma *before-dist-fst-fst*[*simp*]: *before vs ram1 ram2* ⟹ *distinct vs* ⟹ *fst* (*splitAt ram2* (*fst* (*splitAt ram1 vs*))) = *fst* (*splitAt ram1* (*fst* (*splitAt ram2 vs*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *before-dist-snd*[*simp*]: *before vs ram1 ram2* ⟹ *distinct vs* ⟹ *fst* (*splitAt ram1* (*snd* (*splitAt ram2 vs*))) = *snd* (*splitAt ram2 vs*)
⟨*proof*⟩

lemma *before-dist-fst*[*simp*]: *before vs ram1 ram2* ⟹ *distinct vs* ⟹ *fst* (*splitAt ram1* (*fst* (*splitAt ram2 vs*))) = *fst* (*splitAt ram1 vs*)
⟨*proof*⟩

lemma *before-or*: *ram1* ∈ *set vs* ⟹ *ram2* ∈ *set vs* ⟹ *ram1* ≠ *ram2* ⟹ *before vs ram1 ram2* ∨ *before vs ram2 ram1*
⟨*proof*⟩

lemma *before-r1*:
before vs r1 r2 ⟹ *r1* ∈ *set vs* ⟨*proof*⟩

lemma *before-r2*:
before vs r1 r2 ⟹ *r2* ∈ *set vs* ⟨*proof*⟩

lemma *before-dist-r1r2*:
distinct vs ⟹ *before vs r1 r2* ⟹ *r1* ≠ *r2*
⟨*proof*⟩

lemma *before-dist-r2*:
distinct vs ⟹ *before vs r1 r2* ⟹ *r2* ∈ *set* (*snd* (*splitAt r1 vs*))

⟨*proof*⟩

lemma *before-dist-not-r2*[*intro*]:
distinct vs ⟹ *before vs r1 r2* ⟹ *r2* ∉ *set* (*fst* (*splitAt r1 vs*)) ⟨*proof*⟩

lemma *before-dist-r1*:
distinct vs ⟹ *before vs r1 r2* ⟹ *r1* ∈ *set* (*fst* (*splitAt r2 vs*))
⟨*proof*⟩

lemma *before-dist-not-r1*[*intro*]:
distinct vs ⟹ *before vs r1 r2* ⟹ *r1* ∉ *set* (*snd* (*splitAt r2 vs*)) ⟨*proof*⟩

lemma *before-snd*:
r2 ∈ *set* (*snd* (*splitAt r1 vs*)) ⟹ *before vs r1 r2*
⟨*proof*⟩

lemma *before-fst*:
r2 ∈ *set vs* ⟹ *r1* ∈ *set* (*fst* (*splitAt r2 vs*)) ⟹ *before vs r1 r2*
⟨*proof*⟩

lemma *before-dist-eq-fst*:
distinct vs ⟹ *r2* ∈ *set vs* ⟹ *r1* ∈ *set* (*fst* (*splitAt r2 vs*)) = *before vs r1 r2*
⟨*proof*⟩

lemma *before-dist-eq-snd*:
distinct vs ⟹ *r2* ∈ *set* (*snd* (*splitAt r1 vs*)) = *before vs r1 r2*
⟨*proof*⟩

lemma *pair-snd*: (*a*,*b*) = *p* ⟹ *snd p* = *b* ⟨*proof*⟩

lemma *before-dist-eq-snd′*:
distinct vs ⟹ (*as*, *bs*) = (*splitAt* ram_1 *vs*) ⟹
ram_2 ∈ *set bs* = *before vs* ram_1 ram_2
⟨*proof*⟩

lemma *before-dist-not1*:
distinct vs ⟹ *before vs ram1 ram2* ⟹ ¬ *before vs ram2 ram1*
⟨*proof*⟩

lemma *before-dist-not2*:
distinct vs ⟹ *ram1* ∈ *set vs* ⟹ *ram2* ∈ *set vs* ⟹ *ram1* ≠ *ram2* ⟹ ¬ (*before vs ram1 ram2*) ⟹ *before vs ram2 ram1*
⟨*proof*⟩

lemma *before-dist-eq*:
distinct vs ⟹ *ram1* ∈ *set vs* ⟹ *ram2* ∈ *set vs* ⟹ *ram1* ≠ *ram2* ⟹ (¬ (*before vs ram1 ram2*)) = *before vs ram2 ram1*
⟨*proof*⟩

lemma *before-vs*:
distinct vs $\Longrightarrow$ *before vs ram1 ram2* $\Longrightarrow$ *vs* = *fst* (*splitAt ram1 vs*) @ *ram1* # *fst* (*splitAt ram2* (*snd* (*splitAt ram1 vs*))) @ *ram2* # *snd* (*splitAt ram2 vs*)
⟨*proof*⟩

12.8 *between*

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

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

lemma *pre-between-dist*[*intro*]:
pre-between vs ram1 ram2 $\Longrightarrow$ *distinct vs* ⟨*proof*⟩

lemma *pre-between-r1*[*intro*]:
pre-between vs ram1 ram2 $\Longrightarrow$ *ram1* ∈ *set vs* ⟨*proof*⟩

lemma *pre-between-r2*[*intro*]:
pre-between vs ram1 ram2 $\Longrightarrow$ *ram2* ∈ *set vs* ⟨*proof*⟩

lemma *pre-between-r12*[*intro*]:
pre-between vs ram1 ram2 $\Longrightarrow$ *ram1* ≠ *ram2* ⟨*proof*⟩

lemma *pre-between-sym*:
pre-between vs ram1 ram2 = *pre-between vs ram2 ram1* ⟨*proof*⟩

lemma *pre-between-symI*:
pre-between vs ram1 ram2 $\Longrightarrow$ *pre-between vs ram2 ram1* ⟨*proof*⟩

lemma *pre-between-before*[*dest*]:
pre-between vs ram1 ram2 $\Longrightarrow$ *before vs ram1 ram2* ∨ *before vs ram2 ram1* ⟨*proof*⟩

lemma *pre-between-rotate1*[*intro*]:
pre-between vs ram1 ram2 $\Longrightarrow$ *pre-between* (*rotate1 vs*) *ram1 ram2* ⟨*proof*⟩

lemma *pre-between-rotate*[*intro*]:
pre-between vs ram1 ram2 $\Longrightarrow$ *pre-between* (*rotate n vs*) *ram1 ram2* ⟨*proof*⟩

lemma *pre-between vs ram1 ram2* $\Longrightarrow$ (¬ *before vs ram1 ram2*) = *before vs ram2 ram1*
⟨*proof*⟩

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)))
⟨*proof*⟩

lemma *between-simp2*[*simp*]:
before vs ram1 ram2 ⟹ pre-between vs ram1 ram2 ⟹
between vs ram2 ram1 = snd (splitAt ram2 vs) @ fst (splitAt ram1 vs)
⟨*proof*⟩

lemma *between-not-r1*[*intro*]:
distinct vs ⟹ ram1 ∉ set (between vs ram1 ram2)
⟨*proof*⟩

lemma *between-not-r2*[*intro*]:
distinct vs ⟹ ram2 ∉ set (between vs ram1 ram2)
⟨*proof*⟩

lemma *between-distinct*[*intro*]:
distinct vs ⟹ distinct (between vs ram1 ram2)
⟨*proof*⟩

lemma *between-distinct-r12*:
distinct vs ⟹ ram1 ≠ ram2 ⟹ distinct (ram1 # between vs ram1 ram2 @ [ram2]) ⟨*proof*⟩

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)
⟨*proof*⟩

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*⟩

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*⟩

lemma *between1*: *ram1 ∈ set vs ⟹ ram2 ∈ set vs ⟹*
ram2 ∈ set (snd (splitAt ram1 vs)) ⟹
vs = fst (splitAt ram1 vs) @ [ram1] @ (between vs ram1 ram2)
@ [ram2] @ snd (splitAt ram2 (snd(splitAt ram1 vs)))
⟨*proof*⟩

lemma *between2*: *ram1 ∈ set vs ⟹ ram2 ∈ set vs ⟹ ram1 ≠ ram2 ⟹*

ram2 ∉ *set* (*snd* (*splitAt ram1 vs*)) ⟹
vs @ *vs* = *fst* (*splitAt ram1 vs*) @ [*ram1*] @ (*between vs ram1 ram2*)
@ [*ram2*] @ *snd* (*splitAt ram2* (*fst*(*splitAt ram1 vs*))) @ [*ram1*]
@ *snd* (*splitAt ram1 vs*)
⟨*proof*⟩

lemma *between-congs*: *pre-between vs ram1 ram2* ⟹ *vs* ≅ *vs'* ⟹ *between vs ram1 ram2* = *between vs' ram1 ram2*
⟨*proof*⟩

lemma *between-inter-empty*:
pre-between vs ram1 ram2 ⟹
set (*between vs ram1 ram2*) ∩ *set* (*between vs ram2 ram1*) = {}
⟨*proof*⟩

12.8.1 *between is-nextElem*

lemma *is-nextElem-or1*: *pre-between vs ram1 ram2* ⟹
is-nextElem vs x y ⟹ *before vs ram1 ram2* ⟹
is-sublist [*x*,*y*] (*ram1* # *between vs ram1 ram2* @ [*ram2*])
∨ *is-sublist* [*x*,*y*] (*ram2* # *between vs ram2 ram1* @ [*ram1*])
⟨*proof*⟩

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*⟩

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

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

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

lemma *pre-between vs ram1 ram2* ⟹ *is-nextElem vs x y* ⟹
(*is-sublist* [*x*,*y*] (*ram1* # *between vs ram1 ram2* @ [*ram2*])
= (¬ *is-sublist* [*x*,*y*] (*ram2* # *between vs ram2 ram1* @ [*ram1*])))
⟨*proof*⟩

lemma *pre-between vs ram1 ram2* ⟹
before vs ram1 ram2 ⟹
∃ *as bs*. *vs* = *as* @[*ram1*] @ *between vs ram1 ram2* @ [*ram2*] @ *bs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is-sublist-same-len*[*simp*]:
length xs = *length ys* $\Longrightarrow$ *is-sublist xs ys* = (*xs* = *ys*)
⟨*proof*⟩

lemma *is-nextElem-between-empty*[*simp*]:
distinct vs $\Longrightarrow$ *is-nextElem vs a b* $\Longrightarrow$ *between vs a b* = []
⟨*proof*⟩

lemma *is-nextElem-between-empty'*: *between vs a b* = [] $\Longrightarrow$ *distinct vs* $\Longrightarrow$ *a* $\in$ *set vs* $\Longrightarrow$ *b* $\in$ *set vs* $\Longrightarrow$
a $\neq$ *b* $\Longrightarrow$ *is-nextElem vs a b*
⟨*proof*⟩

lemma *between-nextElem*: *pre-between vs u v* $\Longrightarrow$
between vs u (*nextElem vs* (*hd vs*) *v*) = *between vs u v* @ [*v*]
⟨*proof*⟩

lemma *nextVertices-in-face*[*simp*]: *v* $\in$ $\mathcal{V}$ *f* $\Longrightarrow$ $f^n \cdot v \in \mathcal{V}$ *f*
⟨*proof*⟩

12.8.2 *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* ⟨*proof*⟩

lemma *is-nextElem-edges2*:
distinct (*vertices f*) $\Longrightarrow$ *is-nextElem* (*vertices f*) *a b* $\Longrightarrow$
(*a*,*b*) $\in$ *edges* (*f*::*face*)
⟨*proof*⟩

lemma *is-nextElem-edges-eq*[*simp*]:
distinct (*vertices* (*f*::*face*)) $\Longrightarrow$
(*a*,*b*) $\in$ *edges f* = *is-nextElem* (*vertices f*) *a b*

⟨*proof*⟩

12.8.3 *nextVertex*

lemma *nextElem-suc2*: *distinct* (*vertices f*) $\Longrightarrow$ *last* (*vertices f*) = *v* $\Longrightarrow$ *v* $\in$ *set* (*vertices f*) $\Longrightarrow$ *f* · *v* = *hd* (*vertices f*)
⟨*proof*⟩

lemma *nextVertex-congs-eq*: $f_1 \cong f_2 \Longrightarrow$ *distinct* (*vertices* f_1) $\Longrightarrow$ *v* $\in \mathcal{V}\ f_1 \Longrightarrow$ f_1 · *v* = f_2 · *v*
⟨*proof*⟩

12.9 *split-face*

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

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

lemma *pre-split-face-p-between*[*intro*]:
pre-split-face oldF ram1 ram2 newVertexList $\Longrightarrow$ *pre-between* (*vertices oldF*) *ram1 ram2* ⟨*proof*⟩

lemma *pre-split-face-symI*:
pre-split-face oldF ram1 ram2 newVertexList $\Longrightarrow$ *pre-split-face oldF ram2 ram1 newVertexList* ⟨*proof*⟩

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

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*⟩

lemma *split-face-distinct1'*[*intro*]:
pre-split-face oldF ram1 ram2 newVertexList $\Longrightarrow$
distinct (*vertices* (*fst*(*split-face oldF ram1 ram2 newVertexList*)))
⟨*proof*⟩

lemma *split-face-distinct2*:
(*f12*, *f21*) = *split-face oldF ram1 ram2 newVertexList* $\Longrightarrow$
pre-split-face oldF ram1 ram2 newVertexList $\Longrightarrow$ *distinct* (*vertices f21*)

⟨*proof*⟩

lemma *split-face-distinct2'*[*intro*]:
pre-split-face oldF ram1 ram2 newVertexList ⟹ *distinct* (*vertices* (*snd*(*split-face oldF ram1 ram2 newVertexList*)))
⟨*proof*⟩

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*⟩

lemma *is-nextElem-hd-last*: *distinct vs* ⟹ *is-nextElem vs x y* ⟹ *y* = *hd vs* ⟹ *x* = *last vs*
⟨*proof*⟩

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

12.10 *verticesFrom*

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

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

lemma *len-vFrom*[*simp*]:
v ∈ 𝒱 *f* ⟹ |*verticesFrom f v*| = |*vertices f*|
⟨*proof*⟩

lemma *verticesFrom-empty*[*simp*]:
v ∈ 𝒱 *f* ⟹ (*verticesFrom f v* = []) = (*vertices f* = [])
⟨*proof*⟩

lemma *verticesFrom-congs*:
v ∈ 𝒱 *f* ⟹ (*vertices f*) ≅ (*verticesFrom f v*)
⟨*proof*⟩

lemma *verticesFrom-eq-if-vertices-cong*:
⟦*distinct*(*vertices f*); *distinct*(*vertices f'*);
vertices f ≅ *vertices f'*; *x* ∈ 𝒱 *f* ⟧ ⟹

$verticesFrom\ f\ x = verticesFrom\ f'\ x$
⟨*proof*⟩

lemma *verticesFrom-in*[*intro*]: $v \in \mathcal{V}\ f \Longrightarrow a \in \mathcal{V}\ f \Longrightarrow a \in set\ (verticesFrom\ f\ v)$
⟨*proof*⟩

lemma *verticesFrom-in'*: $a \in set\ (verticesFrom\ f\ v) \Longrightarrow a \neq v \Longrightarrow a \in \mathcal{V}\ f$
⟨*proof*⟩

lemma *set-verticesFrom*:
$v \in \mathcal{V}\ f \Longrightarrow set\ (verticesFrom\ f\ v) = \mathcal{V}\ f$
⟨*proof*⟩

lemma *verticesFrom-hd*: $hd\ (verticesFrom\ f\ v) = v$ ⟨*proof*⟩

lemma *verticesFrom-distinct*[*simp*]: $distinct\ (vertices\ f) \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow distinct\ (verticesFrom\ f\ v)$ ⟨*proof*⟩

lemma *verticesFrom-nextElem-eq*:
$distinct\ (vertices\ f) \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow u \in \mathcal{V}\ f \Longrightarrow$
$nextElem\ (verticesFrom\ f\ v)\ (hd\ (verticesFrom\ f\ v))\ u$
$= nextElem\ (vertices\ f)\ (hd\ (vertices\ f))\ u$ ⟨*proof*⟩

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*⟩

lemma *nextElem-vFrom-suc2*: $distinct\ (vertices\ f) \Longrightarrow last\ (verticesFrom\ f\ v) = u \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow u \in \mathcal{V}\ f \Longrightarrow f \cdot u = hd\ (verticesFrom\ f\ v)$
⟨*proof*⟩

lemma *verticesFrom-nth*: $distinct\ (vertices\ f) \Longrightarrow d < length\ (vertices\ f) \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow (verticesFrom\ f\ v)!d = f^d \cdot v$
⟨*proof*⟩

lemma *verticesFrom-length*: $distinct\ (vertices\ f) \Longrightarrow v \in set\ (vertices\ f) \Longrightarrow length\ (verticesFrom\ f\ v) = length\ (vertices\ f)$
⟨*proof*⟩

lemma *verticesFrom-between*: $v' \in \mathcal{V}\ f \Longrightarrow pre\text{-}between\ (vertices\ f)\ u\ v \Longrightarrow between\ (vertices\ f)\ u\ v = between\ (verticesFrom\ f\ v')\ u\ v$
⟨*proof*⟩

lemma *verticesFrom-is-nextElem*: $v \in \mathcal{V}\ f \Longrightarrow$

is-nextElem (*vertices f*) *a b* = *is-nextElem* (*verticesFrom f v*) *a b*
⟨*proof*⟩

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* $\Longrightarrow v = v'$
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *verticesFrom-pres-nodes1*: $v \in \mathcal{V}\ f \Longrightarrow \mathcal{V}\ f =$ *set*(*verticesFrom f v*)
⟨*proof*⟩

lemma *verticesFrom-pres-nodes*: $v \in \mathcal{V}\ f \Longrightarrow w \in \mathcal{V}\ f \Longrightarrow w \in$ *set* (*verticesFrom f v*)
⟨*proof*⟩

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*⟩

lemma *next-not-before*:
⟦ *distinct*(*vertices f*); $x \in \mathcal{V}\ f$; $y \in \mathcal{V}\ f$; $x \neq f \cdot y$ ⟧ $\Longrightarrow$
¬ *before* (*verticesFrom f x*) ($f \cdot y$) *y*
⟨*proof*⟩

lemma *last-vFrom*:
⟦ *distinct*(*vertices f*); $x \in \mathcal{V}\ f$ ⟧ $\Longrightarrow$ *last*(*verticesFrom f x*) = $f^{-1} \cdot x$
⟨*proof*⟩

⟨*ML*⟩
lemma *rotate-before-vFrom*:
⟦ *distinct*(*vertices f*); $r \in \mathcal{V}\ f$; $r \neq u$ ⟧ $\Longrightarrow$
before (*verticesFrom f r*) *u v* $\Longrightarrow$ *before* (*verticesFrom f v*) *r u*
⟨*proof*⟩

lemma *before-between*:
⟦ *before*(*verticesFrom f x*) *y z*; *distinct*(*vertices f*); $x \in \mathcal{V}\ f$; $x \neq y$ ⟧ $\Longrightarrow$
$y \in$ *set*(*between* (*vertices f*) *x z*)
⟨*proof*⟩
⟨*ML*⟩

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

12.11 *splitFace*

constdefs *pre-splitFace* :: *graph* ⇒ *vertex* ⇒ *vertex* ⇒ *face* ⇒ *vertex list* ⇒ *bool*
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*
⟨*proof*⟩

lemma *pre-splitFace-oldF*[*simp*]:
pre-splitFace g ram1 ram2 oldF nvs ⟹ *oldF* ∈ $\mathcal{F}$ *g*
⟨*proof*⟩

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

lemma *p-splitFace-ne*:
pre-splitFace g ram1 ram2 oldF nvs ⟹ (*ram1*, *ram2*) ∈ *edges oldF* ⟹ *nvs* ≠
[]
⟨*proof*⟩

lemma *splitFace-preserve-face-not-empty*:
f ∉ $\mathcal{F}$ *g* ⟹ *vertices f′* = *vs* ⟹ *ram2* ∈ *set vs* ⟹
f ∈ $\mathcal{F}$ (*snd* (*snd* (*splitFace g ram1 ram2 f′ ns*))) ⟹ *vertices f* ≠ []
⟨*proof*⟩

lemma *splitFace-preserve-faces*:
f ∈ $\mathcal{F}$ *g* ⟹ *f* ≠ *oldF* ⟹
f ∈ $\mathcal{F}$ (*snd* (*snd* (*splitFace g ram1 ram2 oldF nvs*)))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *splitFace-add-f21*:
(*f12*, *f21*, *g′*) = *splitFace g ram1 ram2 oldF newVertexList* ⟹ *oldF* ∈ $\mathcal{F}$ *g*
⟹ *f21* ∈ $\mathcal{F}$ *g′*

⟨*proof*⟩

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*⟩

lemma *split-face-empty-ram2-ram1-in-f12'*:
pre-split-face oldF ram1 ram2 [] ⟹
(*ram2*, *ram1*) ∈ *edges* (*fst* (*split-face oldF ram1 ram2* []))
⟨*proof*⟩

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*⟩

lemma *splitFace-f12-new-vertices*:
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs* ⟹
v ∈ *set newVs* ⟹ *v* ∈ 𝒱 *f12*
⟨*proof*⟩

lemma *splitFace-add-vertices*:
newVertexList = [*countVertices g* ..< *countVertices g* + *n*]
⟹ (*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF* (*rev newVertexList*)
⟹ *vertices newGraph* = *vertices g* @ *newVertexList*
⟨*proof*⟩

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*]
⟨*proof*⟩

lemma *splitFace-delete-oldF*:
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVertexList* ⟹
oldF ≠ *f12* ⟹ *oldF* ≠ *f21* ⟹ *distinct* (*faces g*) ⟹
oldF ∉ ℱ *newGraph*
⟨*proof*⟩

lemma *splitFace-faces-1*:
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVertexList* ⟹
oldF ∈ ℱ *g* ⟹
set (*faces newGraph*) ∪ {*oldF*} = {*f12*, *f21*} ∪ *set* (*faces g*)
(**is** *?oldF* ⟹ *?C* ⟹ *?A* = *?B*)
⟨*proof*⟩

lemma *splitFace-distinct1*[*intro*]:*pre-splitFace g ram1 ram2 oldF newVertexList*
$\Longrightarrow$
distinct (*vertices* (*fst* (*snd* (*splitFace g ram1 ram2 oldF newVertexList*))))
⟨*proof*⟩

lemma *splitFace-distinct2*[*intro*]:*pre-splitFace g ram1 ram2 oldF newVertexList*
$\Longrightarrow$
distinct (*vertices* (*fst* (*splitFace g ram1 ram2 oldF newVertexList*)))
⟨*proof*⟩

lemma *splitFace-add-f21'*: $f' \in \mathcal{F}\ g' \Longrightarrow$ *fst* (*snd* (*splitFace g' v a f' nvl*))
$\in \mathcal{F}$ (*snd* (*snd* (*splitFace g' v a f' nvl*)))
⟨*proof*⟩

lemma *split-face-help*[*simp*]: *Suc 0* $<$ |*vertices* (*fst* (*split-face f' v a nvl*))|
⟨*proof*⟩

lemma *split-face-help'*[*simp*]: *Suc 0* $<$ |*vertices* (*snd* (*split-face f' v a nvl*))|
⟨*proof*⟩

lemma *splitFace-split*: $f \in \mathcal{F}$ (*snd* (*snd* (*splitFace g v a f' nvl*))) $\Longrightarrow$
$f \in \mathcal{F}\ g$
$\vee$ f = *fst* (*splitFace g v a f' nvl*)
$\vee$ f = (*fst* (*snd* (*splitFace g v a f' nvl*)))
⟨*proof*⟩

lemma *pre-FaceDiv-between1*: *pre-splitFace g' ram1 ram2 f* [] $\Longrightarrow$
$\neg$ *between* (*vertices f*) *ram1 ram2* = []
⟨*proof*⟩

lemma *pre-FaceDiv-between2*: *pre-splitFace g' ram1 ram2 f* [] $\Longrightarrow$
$\neg$ *between* (*vertices f*) *ram2 ram1* = []
⟨*proof*⟩

constdefs
Edges :: *vertex list* $\Rightarrow$ (*vertex* $\times$ *vertex*) *set*
Edges vs $\equiv$ {(*a*,*b*). *is-sublist* [*a*,*b*] *vs*}

lemma *Edges-Nil*[*simp*]: *Edges* [] = {}
⟨*proof*⟩

lemma *Edges-rev*:
Edges (*rev* (*zs*::*vertex list*)) = {(*b*,*a*). (*a*,*b*) $\in$ *Edges zs*}

⟨*proof*⟩

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

lemma *notinset-notinEdge1*: *x* ∉ *set xs* ⟹ (*x*,*y*) ∉ *Edges xs*
⟨*proof*⟩

lemma *notinset-notinEdge2*: *y* ∉ *set xs* ⟹ (*x*,*y*) ∉ *Edges xs*
⟨*proof*⟩

lemma *in-Edges-in-set*: (*x*,*y*) : *Edges vs* ⟹ *x* ∈ *set vs* ∧ *y* ∈ *set vs*
⟨*proof*⟩

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*))})
⟨*proof*⟩

lemma *edges-conv-Edges-if-cong*:
⟦ *vertices* (*f*::*face*) ≅ *vs*; *distinct vs*; *vs* ≠ [] ⟧ ⟹
$\mathcal{E}$ *f* = *Edges vs* ∪ {(*last vs*,*hd vs*)}
⟨*proof*⟩

lemma *Edges-Cons*: *Edges*(*x*#*xs*) =
(*if xs* = [] *then* {} *else Edges xs* ∪ {(*x*,*hd xs*)})
⟨*proof*⟩

lemma *Edges-append*: *Edges*(*xs* @ *ys*) =
(*if xs* = [] *then Edges ys else*
if ys = [] *then Edges xs else*
Edges xs ∪ *Edges ys* ∪ {(*last xs*, *hd ys*)})
⟨*proof*⟩

lemma *Edges-rev-disj*: *distinct xs* ⟹ *Edges*(*rev xs*) ∩ *Edges*(*xs*) = {}
⟨*proof*⟩

lemma *disj-sets-disj-Edges*:
set xs ∩ *set ys* = {} ⟹ *Edges xs* ∩ *Edges ys* = {}
⟨*proof*⟩

lemma *disj-sets-disj-Edges2*:
set ys ∩ *set xs* = {} ⟹ *Edges xs* ∩ *Edges ys* = {}
⟨*proof*⟩

lemma *finite-Edges*[*iff*]: *finite*(*Edges xs*)
⟨*proof*⟩

lemma *length-conv-card-Edges*:
distinct xs ⟹ |*xs*| = (*if* |*xs*| = *0 then 0 else card* (*Edges xs*) + *1*)
⟨*proof*⟩

lemma *card-edges*:
distinct(*vertices*(*f*::*face*)) ⟹ *card*(*ℰ f*) = |*vertices f*|
⟨*proof*⟩

⟨*ML*⟩
lemma *Edges-compl*:
⟦ *distinct vs*; *x* ∈ *set vs*; *y* ∈ *set vs*; *x* ≠ *y* ⟧ ⟹
Edges(*x* # *between vs x y* @ [*y*]) ∩ *Edges*(*y* # *between vs y x* @ [*x*]) = {}
⟨*proof*⟩

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*]) = {}
⟨*proof*⟩

lemma *edges-conv-Un-Edges*:
⟦ *distinct*(*vertices*(*f*::*face*)); *x* ∈ *𝒱 f*; *y* ∈ *𝒱 f*; *x* ≠ *y* ⟧ ⟹
ℰ f = *Edges*(*x* # *between* (*vertices f*) *x y* @ [*y*]) ∪
Edges(*y* # *between* (*vertices f*) *y x* @ [*x*])
⟨*proof*⟩
⟨*ML*⟩

lemma *Edges-between-edges*:
⟦ (*a*,*b*) ∈ *Edges* (*u* # *between* (*vertices*(*f*::*face*)) *u v* @ [*v*]);
pre-split-face f u v vs ⟧ ⟹ (*a*,*b*) ∈ *ℰ f*
⟨*proof*⟩

⟨*ML*⟩
lemma *triangle-if-3circular*:
⟦ *distinct*[*a*,*b*,*c*]; *distinct*(*vertices*(*f*::*face*));
(*a*,*b*) ∈ *ℰ f*; (*b*,*c*) ∈ *ℰ f*; (*c*,*a*) ∈ *ℰ f* ⟧
⟹ *ℰ f* = {(*a*,*b*),(*b*,*c*),(*c*,*a*)}
⟨*proof*⟩
⟨*ML*⟩

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*])
⟨*proof*⟩

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*])
⟨*proof*⟩

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*⟩

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*⟩

lemma *splitFace-empty-ram1-ram2-in-f21*:
pre-splitFace g ram1 ram2 oldF [] $\Longrightarrow$
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF* [] $\Longrightarrow$
(*ram1*, *ram2*) ∈ *edges f21*
⟨*proof*⟩

lemma *splitFace-f21-new-vertices*:
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs* $\Longrightarrow$
v ∈ *set newVs* $\Longrightarrow$ *v* ∈ $\mathcal{V}$ *f21*
⟨*proof*⟩

lemma *split-face-f12-f21-neq*:
pre-split-face oldF ram1 ram2 vs $\Longrightarrow$
(*f12*, *f21*) = *split-face oldF ram1 ram2 vs* $\Longrightarrow$ *f12* ≠ *f21*
⟨*proof*⟩

lemma *split-face-edges-f12*: *pre-split-face f ram1 ram2 vs* $\Longrightarrow$
(*f12*, *f21*) = *split-face f ram1 ram2 vs* $\Longrightarrow$ *vs* ≠ [] $\Longrightarrow$ *vs1* = *between* (*vertices f*) *ram1 ram2* $\Longrightarrow$ *vs1* ≠ [] $\Longrightarrow$
edges f12 = {(*hd vs*, *ram1*) , (*ram1*, *hd vs1*), (*last vs1*, *ram2*), (*ram2*, *last vs*)}
∪
Edges(*rev vs*) ∪ *Edges vs1* (**concl is** *?lhs* = *?rhs*)

⟨*proof*⟩

lemma *split-face-edges-f12-vs*: *pre-split-face f ram1 ram2* [] ⟹
(*f12*, *f21*) = *split-face f ram1 ram2* [] ⟹ *vs1* = *between* (*vertices f*) *ram1 ram2*
⟹ *vs1* ≠ [] ⟹
edges f12 = {(*ram2*, *ram1*) , (*ram1*, *hd vs1*), (*last vs1*, *ram2*)} ∪
Edges vs1 (**concl is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *split-face-edges-f12-bet*: *pre-split-face f ram1 ram2 vs* ⟹
(*f12*, *f21*) = *split-face f ram1 ram2 vs* ⟹ *vs* ≠ [] ⟹ *between* (*vertices f*) *ram1*
ram2 = [] ⟹
edges f12 = {(*hd vs*, *ram1*) , (*ram1*, *ram2*), (*ram2*, *last vs*)} ∪
Edges(*rev vs*) (**concl is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *split-face-edges-f12-bet-vs*: *pre-split-face f ram1 ram2* [] ⟹
(*f12*, *f21*) = *split-face f ram1 ram2* [] ⟹ *between* (*vertices f*) *ram1 ram2* = []
⟹
edges f12 = {(*ram2*, *ram1*) , (*ram1*, *ram2*)} (**concl is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *split-face-edges-f21*: *pre-split-face f ram1 ram2 vs* ⟹
(*f12*, *f21*) = *split-face f ram1 ram2 vs* ⟹ *vs* ≠ [] ⟹ *vs2* = *between* (*vertices*
f) *ram2 ram1* ⟹ *vs2* ≠ [] ⟹
edges f21 = {(*last vs2*, *ram1*) , (*ram1*, *hd vs*), (*last vs*, *ram2*), (*ram2*, *hd vs2*)}
∪
Edges vs ∪ *Edges vs2* (**concl is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *split-face-edges-f21-vs*: *pre-split-face f ram1 ram2* [] ⟹
(*f12*, *f21*) = *split-face f ram1 ram2* [] ⟹ *vs2* = *between* (*vertices f*) *ram2 ram1*
⟹ *vs2* ≠ [] ⟹
edges f21 = {(*last vs2*, *ram1*) , (*ram1*, *ram2*), (*ram2*, *hd vs2*)} ∪
Edges vs2 (**concl is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

ram1 = [] ⟹
edges f21 = {(*ram1*, *hd vs*), (*last vs*, *ram2*), (*ram2*, *ram1*)} ∪
Edges vs (**concl is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *split-face-edges-f21-bet-vs*: *pre-split-face f ram1 ram2* [] ⟹
(*f12*, *f21*) = *split-face f ram1 ram2* [] ⟹ *between* (*vertices f*) *ram2 ram1* = []
⟹
edges f21 = {(*ram1*, *ram2*), (*ram2*, *ram1*)} (**concl is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *split-face-edges-f21-subset*: *pre-split-face f ram1 ram2 vs* ⟹
(*f12*, *f21*) = *split-face f ram1 ram2 vs* ⟹ *vs* ≠ [] ⟹
{(*last vs*, *ram2*), (*ram1*, *hd vs*)} ∪ *Edges vs* ⊆ *edges f21*
⟨*proof*⟩

lemma *verticesFrom-ram1*: *pre-split-face f ram1 ram2 vs* ⟹
verticesFrom f ram1 = *ram1* # *between* (*vertices f*) *ram1 ram2* @ *ram2* #
between (*vertices f*) *ram2 ram1*
⟨*proof*⟩

lemma *split-face-edges-f-vs1-vs2*: *pre-split-face f ram1 ram2 vs* ⟹
between (*vertices f*) *ram1 ram2* = [] ⟹
between (*vertices f*) *ram2 ram1* = [] ⟹
edges f = {(*ram2*, *ram1*) , (*ram1*, *ram2*)} (**concl is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *split-face-edges-f-vs1*: *pre-split-face f ram1 ram2 vs* ⟹
between (*vertices f*) *ram1 ram2* = [] ⟹
vs2 = *between* (*vertices f*) *ram2 ram1* ⟹ *vs2* ≠ [] ⟹
edges f = {(*last vs2*, *ram1*) , (*ram1*, *ram2*), (*ram2*, *hd vs2*)} ∪
Edges vs2 (**concl is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *split-face-edges-f-vs2*: *pre-split-face f ram1 ram2 vs* ⟹
vs1 = *between* (*vertices f*) *ram1 ram2* ⟹ *vs1* ≠ [] ⟹
between (*vertices f*) *ram2 ram1* = [] ⟹
edges f = {(*ram2*, *ram1*) , (*ram1*, *hd vs1*), (*last vs1*, *ram2*)} ∪
Edges vs1 (**concl is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

Edges vs1 ∪ *Edges vs2* (**concl is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)}
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*⟩

lemma *pre-splitFace-edges-f-in-g*: *pre-splitFace g ram1 ram2 f vs* ⟹ *edges f* ⊆ *edges g*
⟨*proof*⟩

lemma *pre-splitFace-edges-f-in-g2*: *pre-splitFace g ram1 ram2 f vs* ⟹ *x* ∈ *edges f* ⟹ *x* ∈ *edges g*
⟨*proof*⟩

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*)}
⟨*proof*⟩

lemma *splitFace-edges-g'-vs*: *pre-splitFace g ram1 ram2 f* [] $\Longrightarrow$
(f12, f21, g') = splitFace g ram1 ram2 f [] $\Longrightarrow$
edges g' = edges g $\cup$ {*(ram1, ram2), (ram2, ram1)*}
⟨*proof*⟩

lemma *splitFace-edges-incr*:
pre-splitFace g ram1 ram2 f vs $\Longrightarrow$
(f_1, f_2, g') = splitFace g ram1 ram2 f vs $\Longrightarrow$
edges g $\subseteq$ *edges g'*
⟨*proof*⟩

lemma *snd-snd-splitFace-edges-incr*:
pre-splitFace g v_1 v_2 f vs $\Longrightarrow$
edges g $\subseteq$ *edges(snd(snd(splitFace g v_1 v_2 f vs)))*
⟨*proof*⟩

12.12 *removeNones*

constdefs *removeNones* :: *'a option list* $\Rightarrow$ *'a list*
removeNones vOptionList $\equiv$ [*the x. x* $\in$ *vOptionList*, (λ*x. x* $\neq$ *None*)]

declare *removeNones-def* [*simp*]
lemma *removeNones-inI*[*intro*]: *Some a* $\in$ *set ls* $\Longrightarrow$ *a* $\in$ *set* (*removeNones ls*) ⟨*proof*⟩
lemma *removeNones-hd*[*simp*]: *removeNones* (*Some a # ls*) = *a # removeNones ls* ⟨*proof*⟩
lemma *removeNones-last*[*simp*]: *removeNones* (*ls* @ [*Some a*]) = *removeNones ls* @ [*a*] ⟨*proof*⟩
lemma *removeNones-in*[*simp*]: *removeNones* (*as* @ *Some a # bs*) = *removeNones as* @ *a # removeNones bs* ⟨*proof*⟩
lemma *removeNones-none-hd*[*simp*]: *removeNones* (*None # ls*) = *removeNones ls* ⟨*proof*⟩
lemma *removeNones-none-last*[*simp*]: *removeNones* (*ls* @ [*None*]) = *removeNones ls* ⟨*proof*⟩
lemma *removeNones-none-in*[*simp*]: *removeNones* (*as* @ *None # bs*) = *removeNones* (*as* @ *bs*) ⟨*proof*⟩
lemma *removeNones-inI*[*intro*]: *Some a* $\in$ *set ls* $\Longrightarrow$ *a* $\in$ *set* (*removeNones ls*) ⟨*proof*⟩
lemma *removeNones-empty*[*simp*]: *removeNones* [] = [] ⟨*proof*⟩
declare *removeNones-def* [*simp del*]

12.13 *natToVertexList*

consts *natToVertexListRec* ::
nat $\Rightarrow$ *vertex* $\Rightarrow$ *face* $\Rightarrow$ *nat list* $\Rightarrow$ *vertex option list*

primrec
natToVertexListRec old v f [] = []
natToVertexListRec old v f (*i*#*is*) =
(*if i* = *old then None*#*natToVertexListRec i v f is*
else Some (f^i · *v*)
natToVertexListRec i v f is)

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

12.14 *indexToVertexList*

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

lemma *a*: *distinct* (*vertices f*) ⟹ *v* ∈ $\mathcal{V}$ *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*⟩

lemma *indexToVertexList-natToVertexList-eq*: *distinct* (*vertices f*) ⟹ *v* ∈ $\mathcal{V}$ *f* ⟹
(∀ *i* ∈ *set is*. *i* < *length* (*vertices f*)) ⟹ *is* ≠ [] ⟹
hd is = *0* ⟹ *indexToVertexList f v is* = *natToVertexList v f is*
⟨*proof*⟩

lemma *nvlr-length*: ⋀ *old*. (*length* (*natToVertexListRec old v f ls*)) = *length ls*
⟨*proof*⟩

lemma *nvl-length*[*simp*]: *hd e* = *0* ⟹ *length* (*natToVertexList v f e*) = *length e*
⟨*proof*⟩

lemma *nvl-nvlRec*: *1* < *i* ⟹ *incrIndexList e i* (*length* (*vertices f*)) ⟹ *natToVer-*

texList v f e = (Some v) # natToVertexListRec 0 v f (tl e)
⟨*proof*⟩

lemma *natToVertexListRec-length*[*simp*]: ⋀ *e f*. *length* (*natToVertexListRec e v f es*) = *length es*
⟨*proof*⟩

lemma *natToVertexList-length*[*simp*]: *incrIndexList es* (*length es*) (*length* (*vertices f*)) ⟹
length (*natToVertexList v f es*) = *length es* ⟨*proof*⟩

lemma *natToVertexList-nth-Suc*: *incrIndexList es* (*length es*) (*length* (*vertices f*)) ⟹ *Suc n* < *length es* ⟹
(*natToVertexList v f es*)!(*Suc n*) = (*if* (*es*!*n* = *es*!(*Suc n*)) *then None else* (*Some* $f^{(es!Suc\ n)}$ ⋅ *v*))
⟨*proof*⟩

lemma *natToVertexList-nth-0*: *incrIndexList es* (*length es*) (*length* (*vertices f*)) ⟹ *0* < *length es* ⟹
(*natToVertexList v f es*)!*0* = (*Some* $f^{(es!0)}$ ⋅ *v*) ⟨*proof*⟩

lemma *natToVertexList-hd*[*simp*]:
incrIndexList es (*length es*) (*length* (*vertices f*)) ⟹ *hd* (*natToVertexList v f es*) = *Some v*
⟨*proof*⟩

lemma *nth-last*[*intro*]: *Suc i* = *length xs* ⟹ *xs*!*i* = *last xs*
⟨*proof*⟩

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*⟩

lemma *indexToVertexList-last*[*simp*]:
distinct (*vertices f*) ⟹ *v* ∈ 𝒱 *f* ⟹ *incrIndexList es* (*length es*) (*length* (*vertices f*)) ⟹ *last* (*indexToVertexList f v es*) = *Some* (*last* (*verticesFrom f v*))
⟨*proof*⟩

lemma *sublist-take*: ⋀ *n iset*. ∀ *i* ∈ *iset*. *i* < *n* ⟹ *sublist* (*take n xs*) *iset* = *sublist xs iset*
⟨*proof*⟩

lemma *sublist-reduceIndices*: ⋀ *iset*. *sublist xs iset* = *sublist xs* {*i*. *i* < *length xs*

$\wedge\ i \in iset\}$
⟨*proof*⟩

lemma *natToVertexList-sublist1*: *distinct* (*vertices f*) $\Longrightarrow$
v $\in$ $\mathcal{V}$ *f* $\Longrightarrow$ *vs* = *verticesFrom f v* $\Longrightarrow$
incrIndexList es (*length es*) (*length vs*) $\Longrightarrow$ *n* $\leq$ *length es* $\Longrightarrow$
sublist (*take* (*Suc* (*es*!(*n* − *1*))) *vs*) (*set* (*take n es*))
= *removeNones* (*take n* (*natToVertexList v f es*))
⟨*proof*⟩

lemma *natToVertexList-sublist*: *distinct* (*vertices f*) $\Longrightarrow$ *v* $\in$ $\mathcal{V}$ *f* $\Longrightarrow$
incrIndexList es (*length es*) (*length* (*vertices f*)) $\Longrightarrow$
sublist (*verticesFrom f v*) (*set es*) = *removeNones* (*natToVertexList v f es*)
⟨*proof*⟩

lemma *filter-Cons2*:
x $\notin$ *set ys* $\Longrightarrow$ [*y* $\in$ *ys*. *y* = *x* $\vee$ *P y*] = [*y* $\in$ *ys*. *P y*]
⟨*proof*⟩

lemma *natToVertexList-removeNones*:
distinct (*vertices f*) $\Longrightarrow$ *v* $\in$ $\mathcal{V}$ *f* $\Longrightarrow$
incrIndexList es (*length es*) (*length* (*vertices f*)) $\Longrightarrow$
[*x* $\in$ *verticesFrom f v*. *x* $\in$ *set* (*removeNones* (*natToVertexList v f es*))]
= *removeNones* (*natToVertexList v f es*)
⟨*proof*⟩

lemma *indexToVertexList-removeNones*:
distinct (*vertices f*) $\Longrightarrow$ *v* $\in$ $\mathcal{V}$ *f* $\Longrightarrow$
incrIndexList es (*length es*) (*length* (*vertices f*)) $\Longrightarrow$
[*x* $\in$ *verticesFrom f v*. *x* $\in$ *set* (*removeNones* (*indexToVertexList f v es*))]
= *removeNones* (*indexToVertexList f v es*)
⟨*proof*⟩

constdefs *is-duplicateEdge* :: *graph* ⇒ *face* ⇒ *vertex* ⇒ *vertex* ⇒ *bool*
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*)

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

12.15 *pre-subdivFace(′)*

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

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

constdefs *pre-subdivFace′* :: *graph* ⇒ *face* ⇒ *vertex* ⇒ *vertex* ⇒ *nat* ⇒ *vertex option list* ⇒ *bool*
pre-subdivFace′ g f v′ ram1 n vOptionList ≡
 ¬ *final f* ∧ *v′* ∈ 𝒱 *f* ∧ *ram1* ∈ 𝒱 *f*
 ∧ *v′* ∉ *set* (*removeNones vOptionList*)
 ∧ *distinct* (*vertices f*)
 ∧ (
  [*v* ∈ *verticesFrom f v′*. *v* ∈ *set* (*removeNones vOptionList*)]
    = (*removeNones vOptionList*)
 ∧ *before* (*verticesFrom f v′*) *ram1* (*hd* (*removeNones vOptionList*))
 ∧ *last* (*vOptionList*) = *Some* (*last* (*verticesFrom f v′*))
 ∧ *vOptionList* ≠ []
 ∧ ((*v′* = *ram1* ∧ (*0* < *n*)) ∨ ((*v′* = *ram1* ∧ (*hd* (*vOptionList*) ≠ *Some* (*last* (*verticesFrom f v′*)))) ∨ (*v′* ≠ *ram1*)))
 ∧ ¬ *invalidVertexList g f vOptionList*
 ∧ (*n* = *0* ∧ *hd* (*vOptionList*) ≠ *None* ⟶ ¬ *is-duplicateEdge g f ram1* (*the* (*hd* (*vOptionList*))))
 ∨ (*vOptionList* = [] ∧ *v′* ≠ *ram1*)
)

lemma *pre-subdivFace-face-in-f*[*intro*]: *pre-subdivFace-face f v ls* ⟹ *Some a* ∈ *set ls* ⟹ *a* ∈ *set* (*verticesFrom f v*)
 ⟨*proof*⟩

lemma *pre-subdivFace-in-f*[*intro*]: *pre-subdivFace g f v ls* ⟹ *Some a* ∈ *set ls* ⟹ *a* ∈ *set* (*verticesFrom f v*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *hilf1*: *a* $\in$ *set b* $\Longrightarrow$ (*v* = *a* $\vee$ *v* $\in$ *set b*) = (*v* $\in$ *set b*) ⟨*proof*⟩

lemma *filter-congs-shorten1'*: *distinct* (*vertices f*) $\Longrightarrow$ [*v*∈*vertices f* . *v* = *a* $\vee$ *v* $\in$ *set vs*] $\cong$ (*a* # *vs*)
$\Longrightarrow$ [*v*∈*vertices f* . *v* $\in$ *set vs*] $\cong$ *vs*
⟨*proof*⟩

lemma *filter-congs-shorten1*: *distinct* (*verticesFrom f v*) $\Longrightarrow$ [*v*∈*verticesFrom f v* . *v* = *a* $\vee$ *v* $\in$ *set vs*] = (*a* # *vs*)
$\Longrightarrow$ [*v*∈*verticesFrom f v* . *v* $\in$ *set vs*] = *vs*
⟨*proof*⟩

lemma *ovl-shorten'*: *distinct* (*vertices f*) $\Longrightarrow$ [*v*∈*vertices f* . *v* $\in$ *set* (*removeNones* (*va* # *vol*))] $\cong$ (*removeNones* (*va* # *vol*))
$\Longrightarrow$ [*v*∈*vertices f* . *v* $\in$ *set* (*removeNones* (*vol*))] $\cong$ (*removeNones* (*vol*))
⟨*proof*⟩

lemma *ovl-shorten*: *distinct* (*verticesFrom f v*) $\Longrightarrow$
[*v*∈*verticesFrom f v* . *v* $\in$ *set* (*removeNones* (*va* # *vol*))] = (*removeNones* (*va* # *vol*))
$\Longrightarrow$ [*v*∈*verticesFrom f v* . *v* $\in$ *set* (*removeNones* (*vol*))] = (*removeNones* (*vol*))
⟨*proof*⟩

lemma *pre-subdivFace-face-distinct*: *pre-subdivFace-face f v vol* $\Longrightarrow$ *distinct* (*removeNones vol*)
⟨*proof*⟩

lemma *pre-subdivFace-face-not-empty*: *pre-subdivFace-face f v* (*vo* # *vol*) $\Longrightarrow$ *vol* $\neq$ []
⟨*proof*⟩

lemma *invalidVertexList-shorten*: *invalidVertexList g f vol* $\Longrightarrow$ *invalidVertexList g f* (*v* # *vol*)
⟨*proof*⟩

lemma *edges-in-faces-in-graph*: *x* $\in$ *edges f* $\Longrightarrow$ *f* $\in$ $\mathcal{F}$ *g* $\Longrightarrow$ *x* $\in$ *edges g*
⟨*proof*⟩

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*⟩

lemma *pre-subdivFace'-distinct*: *pre-subdivFace' g f v' v n vol* $\Longrightarrow$ *distinct* (*removeNones vol*)
⟨*proof*⟩

lemma *natToVertexList-pre-subdivFace-face*:
¬ *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 (*natToVertexList v f es*)
⟨*proof*⟩

lemma *indexToVertexList-pre-subdivFace-face*:
¬ *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*)
⟨*proof*⟩

lemma *subdivFace-subdivFace'-eq*: *pre-subdivFace g f v vol* $\Longrightarrow$ *subdivFace g f vol* = *subdivFace' g f v 0* (*tl vol*)
⟨*proof*⟩

lemma *pre-subdivFace'-None*:
pre-subdivFace' g f v' v n (*None* # *vol*) $\Longrightarrow$
pre-subdivFace' g f v' v (*Suc n*) *vol*
⟨*proof*⟩

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

lemma *verticesFrom-split*: *v* # *tl* (*verticesFrom f v*) = *verticesFrom f v* ⟨*proof*⟩

lemma *verticesFrom-v*: *distinct* (*vertices f*) $\Longrightarrow$ *vertices f* = *a* @ *v* # *b* $\Longrightarrow$ *verticesFrom f v* = *v* # *b* @ *a*
⟨*proof*⟩

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

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

lemma *verticesFrom-splitAt-v-fst*[*simp*]:
distinct (*verticesFrom f v*) $\Longrightarrow$ *fst* (*splitAt v* (*verticesFrom f v*)) = []
⟨*proof*⟩
lemma *verticesFrom-splitAt-v-snd*[*simp*]:
distinct (*verticesFrom f v*) $\Longrightarrow$ *snd* (*splitAt v* (*verticesFrom f v*)) = *tl* (*verticesFrom f v*)
⟨*proof*⟩

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*] = []
⟨*proof*⟩

lemma *filter-distinct-at2*: *distinct xs* $\Longrightarrow$ *xs* = (*as* @ *u* # *bs*) $\Longrightarrow$
[*v* ∈ *xs*. *v* = *u* ∨ *P v*] = *u* # *us* $\Longrightarrow$ *filter P zs* = [] $\Longrightarrow$ [*v* ∈ *zs*@*bs*. *P v*] = *us*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *filter-distinct-at-special*:
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*
$\Longrightarrow$ *us* = *hd-us* # *tl-us*
$\Longrightarrow$ [*v* ∈ *zs*@*bs*. *v* ∈ *set us*] = *us* ∧ *hd-us* ∈ *set bs*
⟨*proof*⟩

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*⟩

lemma *before-filter*: $\bigwedge$ *ys. filter P xs = ys ⟹ distinct xs ⟹ before ys u v ⟹ before xs u v*
⟨*proof*⟩

lemma *pre-subdivFace'-Some2*: *pre-subdivFace' g f v' v 0 ((Some u) # vol) ⟹ pre-subdivFace' g f v' u 0 vol*
⟨*proof*⟩

lemma *pre-subdivFace'-preFaceDiv*: *pre-subdivFace' g f v' v n ((Some u) # vol)*
⟹ *f ∈ ℱ g ⟹ (f ∙ v = u ⟶ n ≠ 0) ⟹ 𝒱 f ⊆ 𝒱 g*
⟹ *pre-splitFace g v u f [countVertices g ..< countVertices g + n]*
⟨*proof*⟩

lemma *pre-subdivFace'-Some1*:
pre-subdivFace' g f v' v n ((Some u) # vol)
⟹ *f ∈ ℱ g ⟹ (f ∙ v = u ⟶ n ≠ 0) ⟹ 𝒱 f ⊆ 𝒱 g*
⟹ *f21 = fst (snd (splitFace g v u f [countVertices g ..< countVertices g + n]))*
⟹ *g' = snd (snd (splitFace g v u f [countVertices g ..< countVertices g + n]))*
⟹ *pre-subdivFace' g' f21 v' u 0 vol*
⟨*proof*⟩

end

13 Invariants of (Plane) Graphs

theory *Invariants*
imports *FaceDivisionProps*
begin

13.1 Rotation of face into normal form

constdefs *minVertex* :: *face* ⇒ *vertex*
minVertex f ≡ *minList* (*vertices f*)

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

constdefs *normFaces* :: *face list* ⇒ *vertex list list*
normFaces fl ≡ *map normFace fl*

lemma *normFaces-distinct*: *distinct* (*normFaces fl*) ⟹ *distinct fl*
⟨*proof*⟩

13.2 Minimal (plane) graph properties

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

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

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

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

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

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

constdefs *faces-subset*:: *graph* ⇒ *bool*
faces-subset g ≡ ∀ *f* ∈ $\mathcal{F}$ *g*. $\mathcal{V}$ *f* ⊆ $\mathcal{V}$ *g*

constdefs *edges-disj* :: *graph* ⇒ *bool*
edges-disj g ≡
∀ *f* ∈ $\mathcal{F}$ *g*. ∀ *f'* ∈ $\mathcal{F}$ *g*. *f* ≠ *f'* ⟶ $\mathcal{E}$ *f* ∩ $\mathcal{E}$ *f'* = {}

constdefs
face-face-op:: *graph* ⇒ *bool*
face-face-op g ≡ |*faces g*| ≠ 2 ⟶
(∀ *f*∈$\mathcal{F}$ *g*. ∀ *f'*∈$\mathcal{F}$ *g*. *f* ≠ *f'* ⟶ $\mathcal{E}$ *f* ≠ ($\mathcal{E}$ *f'*)$^{-1}$)

constdefs
one-final-but :: *graph* ⇒ (*vertex* × *vertex*)*set* ⇒ *bool*
one-final-but g E ≡

$\forall f \in \mathcal{F}\ g.\ \neg\ final\ f \longrightarrow$
$(\forall (a,b) \in \mathcal{E}\ f\ -\ E.\ (b,a) : E \vee (\exists f' \in \mathcal{F}\ g.\ final\ f' \wedge (b,a) \in \mathcal{E}\ f'))$

one-final :: *graph* $\Rightarrow$ *bool*
one-final g $\equiv$ *one-final-but g* $\{\}$

constdefs
minGraphProps :: *graph* $\Rightarrow$ *bool*
minGraphProps g $\equiv$ *minGraphProps' g* $\wedge$ *facesAt-eq g* $\wedge$ *faceListAt-len g* $\wedge$ *facesAt-distinct g* $\wedge$ *faces-distinct g* $\wedge$ *faces-subset g* $\wedge$ *edges-sym g* $\wedge$ *edges-disj g* $\wedge$ *face-face-op g*

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

lemma *facesAt-distinctI*:
$(\bigwedge v.\ v \in \mathcal{V}\ g \Longrightarrow distinct\ (normFaces\ (facesAt\ g\ \ v))) \Longrightarrow$ *facesAt-distinct g*
⟨*proof*⟩

lemma *minGraphProps2*: *minGraphProps g* $\Longrightarrow$
$f \in \mathcal{F}\ g \Longrightarrow 2 < |vertices\ f|$
⟨*proof*⟩

lemma *mgp-vertices3*:
minGraphProps g $\Longrightarrow f \in \mathcal{F}\ g \Longrightarrow |vertices\ f| \geq 3$
⟨*proof*⟩

lemma *mgp-vertices-nonempty*:
minGraphProps g $\Longrightarrow f \in \mathcal{F}\ g \Longrightarrow vertices\ f \neq []$
⟨*proof*⟩

lemma *minGraphProps3*: *minGraphProps g* $\Longrightarrow$
$f \in \mathcal{F}\ g \Longrightarrow$ *distinct* (*vertices f*)
⟨*proof*⟩

lemma *minGraphProps4*: *minGraphProps g* $\Longrightarrow$
(*length* (*faceListAt g*) = *countVertices g*)
⟨*proof*⟩

lemma *minGraphProps5*:
$[\![minGraphProps\ g; f \in set\ (facesAt\ g\ v)]\!] \Longrightarrow f \in \mathcal{F}\ g$
⟨*proof*⟩

lemma *minGraphProps6*:
minGraphProps g $\Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow v \in \mathcal{V}\ f$

⟨*proof*⟩

lemma *minGraphProps9*: *minGraphProps g* ⟹
f ∈ ℱ *g* ⟹ *v* ∈ 𝒱 *f* ⟹ *v* ∈ 𝒱 *g*
⟨*proof*⟩

lemma *minGraphProps7*: *minGraphProps g* ⟹
f ∈ ℱ *g* ⟹ *v* ∈ 𝒱 *f* ⟹ *f* ∈ *set* (*facesAt g v*)
⟨*proof*⟩

lemma *minGraphProps-facesAt-eq*: *minGraphProps g* ⟹
v ∈ 𝒱 *g* ⟹ *set* (*facesAt g v*) = {*f* ∈ ℱ *g*. *v* ∈ 𝒱 *f*}
⟨*proof*⟩

lemma *mgp-dist-facesAt*[*simp*]: *minGraphProps g* ⟹ *distinct* (*facesAt g v*)
⟨*proof*⟩

lemma *minGraphProps8*: *minGraphProps g* ⟹ *distinct* (*normFaces* (*facesAt g v*))
⟨*proof*⟩

lemma *minGraphProps8a*: *minGraphProps g* ⟹
v ∈ 𝒱 *g* ⟹ *distinct* (*normFaces* (*faceListAt g* ! *v*))
⟨*proof*⟩

lemma *minGraphProps8a′*: *minGraphProps g* ⟹
v < *countVertices g* ⟹ *distinct* (*normFaces* (*faceListAt g* ! *v*))
⟨*proof*⟩

lemma *minGraphProps9′*: *minGraphProps g* ⟹
f ∈ ℱ *g* ⟹ *v* ∈ 𝒱 *f* ⟹ *v* < *countVertices g*
⟨*proof*⟩

lemma *minGraphProps10*:
minGraphProps g ⟹ (*a*, *b*) ∈ *edges g* ⟹ (*b*, *a*) ∈ *edges g*
⟨*proof*⟩

lemma *minGraphProps11*: *minGraphProps g* ⟹
distinct (*normFaces* (*faces g*))
⟨*proof*⟩

lemma *minGraphProps11′*: *minGraphProps g* ⟹
distinct (*faces g*)
⟨*proof*⟩

lemma *face-eq-if-normFace-eq*:

⟦ *minGraphProps g*; $f \in \mathcal{F}\ g$; $f' \in \mathcal{F}\ g$; *normFace f* = *normFace f'* ⟧
⟹ $f = f'$
⟨*proof*⟩

lemma *minGraphProps12*:
minGraphProps g ⟹ $f \in \mathcal{F}\ g$ ⟹ $(a,b) \in \mathcal{E}\ f$ ⟹ $(b,a) \notin \mathcal{E}\ f$
⟨*proof*⟩

lemma *minGraphProps7'*: *minGraphProps g* ⟹
$f \in \mathcal{F}\ g$ ⟹ $v \in \mathcal{V}\ f$ ⟹ $f \in$ *set* (*faceListAt g* ! *v*)
⟨*proof*⟩

lemma *mgp-edges-disj*:
⟦ *minGraphProps g*; $f \neq f'$; $f \in \mathcal{F}\ g$; $f' \in \mathcal{F}\ g$ ⟧ ⟹
$uv \in \mathcal{E}\ f$ ⟹ $uv \notin \mathcal{E}\ f'$
⟨*proof*⟩

lemma *one-final-but-antimono*:
one-final-but g E ⟹ $E \subseteq E'$ ⟹ *one-final-but g E'*
⟨*proof*⟩

lemma *one-final-antimono*: *one-final g* ⟹ *one-final-but g E*
⟨*proof*⟩

lemma *inv-two-faces*: *inv g* ⟹ |*faces g*| ≥ 2
⟨*proof*⟩

lemma *inv-mgp*[*simp*]: *inv g* ⟹ *minGraphProps g*
⟨*proof*⟩

lemma *makeFaceFinal-id*[*simp*]: *final f* ⟹ *makeFaceFinal f g* = *g*
⟨*proof*⟩

lemma *inv-one-finalD'*:
⟦ *inv g*; $f \in \mathcal{F}\ g$; ¬ *final f*; $(a,b) \in \mathcal{E}\ f$ ⟧ ⟹
$\exists f' \in \mathcal{F}\ g$. *final f'* ∧ $f' \neq f$ ∧ $(b,a) \in \mathcal{E}\ f'$
⟨*proof*⟩

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

minGraphProps9

lemmas *minGraphProps-simps* = *minGraphProps4*

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*
⟨*proof*⟩

lemma *mgp-facesAt-no-loop*:
minGraphProps g $\Longrightarrow$ *f* $\in$ *set* (*facesAt g v*) $\Longrightarrow$ *f* $\cdot$ *v* $\neq$ *v*
⟨*proof*⟩

lemma *edge-pres-faceAt*:
⟦ *minGraphProps g*; *f* $\in$ *set*(*facesAt g u*); (*u*,*v*) $\in \mathcal{E}$ *f* ⟧ $\Longrightarrow$
f $\in$ *set*(*facesAt g v*)
⟨*proof*⟩

lemma *in-facesAt-nextVertex*:
minGraphProps g $\Longrightarrow$ *f* $\in$ *set*(*facesAt g v*) $\Longrightarrow$ *f* $\in$ *set*(*facesAt g* (*f* $\cdot$ *v*))
⟨*proof*⟩

lemma *mgp-edge-face-ex*:
assumes [*intro*]: *minGraphProps 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*⟩

lemma *mgp-nextVertex-face-ex2*:
assumes *mgp*[*intro*]: *minGraphProps 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*⟩

lemma *inv-finals-nonempty*: *inv g* $\Longrightarrow$ *finals g* $\neq$ []
⟨*proof*⟩

13.3 *containsDuplicateEdge*

constdefs *containsUnacceptableEdgeSnd'* :: (*nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*) $\Rightarrow$ *nat list* $\Rightarrow$ *bool*
containsUnacceptableEdgeSnd' N is $\equiv$
($\exists k <$ |*is*| $-$ *2*. *let i0* = *is*![*k*]; *i1* = *is*![*k*+*1*]; *i2* = *is*![*k*+*2*]) *in*
N i1 i2 $\wedge$ (*i0* $<$ *i1*) $\wedge$ (*i1* $<$ *i2*))

lemma *containsUnacceptableEdgeSnd-eq*: $\bigwedge$ *v*. *containsUnacceptableEdgeSnd N v*

is = containsUnacceptableEdgeSnd' N (v#is)
⟨*proof*⟩

lemma *containsDuplicateEdge-eq1*: *containsDuplicateEdge g f v is = containsDuplicateEdge' g f v is*
⟨*proof*⟩

lemma *containsDuplicateEdge-eq*: *containsDuplicateEdge = containsDuplicateEdge'*
⟨*proof*⟩

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

13.4 *replacefacesAt*

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

lemma *replacefacesAt-eq*[*THEN eq-reflection*]:
⋀*F. replacefacesAt ns oldf newfs F = replacefacesAt2 ns oldf newfs F*
⟨*proof*⟩

lemma *replacefacesAt2-notin*:
⋀*Fss. i ∉ set is* ⟹ (*replacefacesAt2 is olfF newFs Fss*)!*i = Fss*!*i*
⟨*proof*⟩

lemma *replacefacesAt2-in*:
⋀*Fss i. i ∈ set is* ⟹ *distinct is* ⟹ *i < |Fss|* ⟹
(*replacefacesAt2 is olfF newFs Fss*)!*i = replace olfF newFs* (*Fss* !*i*)
⟨*proof*⟩

lemma *distinct-replacefacesAt21*:
⋀*i. 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*⟩

lemma *distinct-replacefacesAt22*:

$\bigwedge i.\ i < |Fss| \Longrightarrow i \notin set\ is \Longrightarrow distinct\ is \Longrightarrow distinct\ (Fss!i) \Longrightarrow distinct\ newFs \Longrightarrow$
$set\ (Fss\ !\ i) \cap set\ newFs \subseteq \{olfF\} \Longrightarrow$
$distinct\ ((replacefacesAt2\ is\ olfF\ newFs\ Fss)!\ i)$
⟨*proof*⟩

lemma *distinct-replacefacesAt2-2*:
$\bigwedge i.\ i < |Fss| \Longrightarrow distinct\ is \Longrightarrow distinct\ (Fss!i) \Longrightarrow distinct\ newFs \Longrightarrow$
$set\ (Fss\ !\ i) \cap set\ newFs \subseteq \{olfF\} \Longrightarrow$
$distinct\ ((replacefacesAt2\ is\ olfF\ newFs\ Fss)!\ i)$
⟨*proof*⟩

lemma *replacefacesAt2-length*: $\bigwedge\ vs.\ |replacefacesAt2\ nvs\ f'\ [f'']\ vs| = |vs|$
⟨*proof*⟩

lemma *replacefacesAt2-nth1*: $!!F.\ k \notin set\ ns \Longrightarrow$
$(replacefacesAt2\ ns\ oldf\ newfs\ F)\ !\ k\ =\ F\ !\ k$
⟨*proof*⟩

lemma *replacefacesAt2-nth1'*: $!!F.\ k \in set\ ns \Longrightarrow k < |F| \Longrightarrow distinct\ ns \Longrightarrow$
$(replacefacesAt2\ ns\ oldf\ newfs\ F)\ !\ k\ =\ (replace\ oldf\ newfs\ (F!k))$
⟨*proof*⟩

lemma *replacefacesAt2-nth2*: $k < |F| \Longrightarrow$
$(replacefacesAt2\ [k]\ oldf\ newfs\ F)\ !\ k = replace\ oldf\ newfs\ (F!k)$
⟨*proof*⟩

lemma *replacefacesAt2-length*[*simp*]: $\bigwedge\ vs.\ |replacefacesAt2\ nvs\ f'\ f''\ vs| = |vs|$
⟨*proof*⟩

lemma *replacefacesAt2-replacefacesAt2-nth1*:
$!!F.\ k < |\ F\ | \Longrightarrow k \notin set\ ns \Longrightarrow distinct\ ms \Longrightarrow$
$replacefacesAt2\ ms\ oldf'\ newfs'\ (replacefacesAt2\ ns\ oldf\ newfs\ F)\ !\ k\ =$
$replacefacesAt2\ ms\ oldf'\ newfs'\ F\ !\ k$
⟨*proof*⟩

lemma *replacefacesAt2-nth*: $!!F.\ k \in\ set\ ns \Longrightarrow k < |F| \Longrightarrow oldf \notin set\ newfs \Longrightarrow$
$distinct\ (F!k) \Longrightarrow distinct\ newfs \Longrightarrow oldf \in set\ (F!k) \longrightarrow set\ newfs \cap set\ (F!k) \subseteq \{oldf\} \Longrightarrow$
$(replacefacesAt2\ ns\ oldf\ newfs\ F)\ !\ k\ =\ (replace\ \ oldf\ newfs\ (F!k))$
⟨*proof*⟩

lemma *replacefacesAt-notin*:
$\bigwedge Fss.\ i \notin set\ is \Longrightarrow (replacefacesAt\ is\ olfF\ newFs\ Fss)!i = Fss!i$
⟨*proof*⟩

lemma *replacefacesAt-in*:
$\bigwedge Fss\ i.\ i \in set\ is \Longrightarrow distinct\ is \Longrightarrow i < |Fss| \Longrightarrow$
$(replacefacesAt\ is\ olfF\ newFs\ Fss)!i = replace\ olfF\ newFs\ (Fss\ !i)$
⟨*proof*⟩

lemma *distinct-replacefacesAt1*:
$\bigwedge i.\ i < |Fss| \Longrightarrow i \in set\ is \Longrightarrow distinct\ is \Longrightarrow distinct\ (Fss!i) \Longrightarrow distinct\ newFs \Longrightarrow$
$set\ (Fss\ !\ i) \cap set\ newFs \subseteq \{olfF\} \Longrightarrow$
$distinct\ ((replacefacesAt\ is\ olfF\ newFs\ Fss)!\ i)$
⟨*proof*⟩

lemma *distinct-replacefacesAt2*:
$\bigwedge i.\ i < |Fss| \Longrightarrow i \notin set\ is \Longrightarrow distinct\ is \Longrightarrow distinct\ (Fss!i) \Longrightarrow distinct\ newFs \Longrightarrow$
$set\ (Fss\ !\ i) \cap set\ newFs \subseteq \{olfF\} \Longrightarrow$
$distinct\ ((replacefacesAt\ is\ olfF\ newFs\ Fss)!\ i)$
⟨*proof*⟩

lemma *distinct-replacefacesAt*:
$\bigwedge i.\ i < |Fss| \Longrightarrow distinct\ is \Longrightarrow distinct\ (Fss!i) \Longrightarrow distinct\ newFs \Longrightarrow$
$set\ (Fss\ !\ i) \cap set\ newFs \subseteq \{olfF\} \Longrightarrow$
$distinct\ ((replacefacesAt\ is\ olfF\ newFs\ Fss)!\ i)$
⟨*proof*⟩

lemma *replacefacesAt-length*[*simp*]: $|replacefacesAt\ nvs\ f'\ [f'']\ vs| = |vs|$
⟨*proof*⟩

lemma *replacefacesAt-nth1*: $k \notin set\ ns \Longrightarrow$
$(replacefacesAt\ ns\ oldf\ newfs\ F)\ !\ k\ =\ F\ !\ k$
⟨*proof*⟩

lemma *replacefacesAt-nth1'*: $k \in set\ ns \Longrightarrow k < |F| \Longrightarrow distinct\ ns \Longrightarrow$
$(replacefacesAt\ ns\ oldf\ newfs\ F)\ !\ k\ =\ (replace\ oldf\ newfs\ (F!k))$
⟨*proof*⟩

lemma *replacefacesAt-nth2*: $k < |F| \Longrightarrow$
$(replacefacesAt\ [k]\ oldf\ newfs\ F)\ !\ k = replace\ oldf\ newfs\ (F!k)$
⟨*proof*⟩

lemma *replacefacesAt-replacefacesAt-nth1*:
$k < |\ F\ | \Longrightarrow k \notin set\ ns \Longrightarrow distinct\ ms \Longrightarrow$
$replacefacesAt\ ms\ oldf'\ newfs'\ (replacefacesAt\ ns\ oldf\ newfs\ F)\ !\ k\ =$
$replacefacesAt\ ms\ oldf'\ newfs'\ F\ !\ k$
⟨*proof*⟩

lemma *replacefacesAt-nth*: $!!F.\ k \in\ set\ ns \Longrightarrow k < |F| \Longrightarrow oldf \notin set\ newfs \Longrightarrow$
$distinct\ (F!k) \Longrightarrow distinct\ newfs \Longrightarrow oldf \in set\ (F!k) \longrightarrow set\ newfs \cap set\ (F!k)$
$\subseteq \{oldf\} \Longrightarrow$

$(replacefacesAt\ ns\ oldf\ newfs\ F)\ !\ k\ =\ (replace\ \ oldf\ newfs\ (F!k))$
⟨*proof*⟩

lemma *replacefacesAt2-5*: $\bigwedge F.\ x \in set\ (replacefacesAt2\ ns\ oldf\ newfs\ F\ !\ k) \Longrightarrow x \in set\ (F!k) \vee x \in set\ newfs$
⟨*proof*⟩

lemma *replacefacesAt5*: $\bigwedge F.\ x \in set\ (replacefacesAt\ ns\ oldf\ newfs\ F\ !\ k) \Longrightarrow x \in set\ (F!k) \vee x \in set\ newfs$
⟨*proof*⟩

lemma *replacefacesAt-delete-oldF*: $oldF \notin set\ newfs \Longrightarrow distinct\ (F!k) \Longrightarrow k \in set\ ns \Longrightarrow distinct\ newfs \Longrightarrow$
$oldF \in set\ (F\ !\ k) \longrightarrow set\ newfs \cap set\ (F\ !\ k) \subseteq \{oldF\} \Longrightarrow k < |F| \Longrightarrow$
$oldF \notin set\ (replacefacesAt\ ns\ oldF\ newfs\ F\ !\ k)$
⟨*proof*⟩

lemma *replacefacesAt-Nil*[*simp*]: $replacefacesAt\ []\ f\ fs\ F = F$
⟨*proof*⟩

lemma *replacefacesAt-Cons*[*simp*]:
$replacefacesAt\ (n\ \#\ ns)\ f\ fs\ F =$
$(if\ n < |F|\ then\ replacefacesAt\ ns\ f\ fs\ (F[n := replace\ f\ fs\ (F!n)])$
$\ else\ replacefacesAt\ ns\ f\ fs\ F)$
⟨*proof*⟩

lemmas *replacefacesAt-simps* = *replacefacesAt-Nil replacefacesAt-Cons*

lemma *len-nth-repAt*[*simp*]:
$!!xs.\ i < |xs| \Longrightarrow |replacefacesAt\ is\ x\ [y]\ xs\ !\ i| = |xs!i|$
⟨*proof*⟩

13.5 *normFace*

lemma *minVertex-in*: $vertices\ f \neq [] \Longrightarrow minVertex\ f \in \mathcal{V}\ f$
⟨*proof*⟩

lemma *minVertex-eq-if-vertices-eq*:
$\mathcal{V}\ f = \mathcal{V}\ f' \Longrightarrow minVertex\ f = minVertex\ f'$
⟨*proof*⟩

lemma *normFace-eq-if-edges-eq*:
$[\![\ distinct(vertices\ f);\ distinct(vertices\ f');\ \mathcal{E}\ f = \mathcal{E}\ f'\ ]\!]$
$\Longrightarrow normFace\ f = normFace\ f'$

⟨*proof*⟩

lemma *normFace-replace-in*: *normFace a* ∈ *set* (*normFaces* (*replace oldF newFs fs*)) ⟹
normFace a ∈ *set* (*normFaces newFs*) ∨ *normFace a* ∈ *set* (*normFaces fs*)
⟨*proof*⟩

lemma *distinct-replace-norm*:
distinct (*normFaces fs*) ⟹ *distinct* (*normFaces newFs*) ⟹
set (*normFaces fs*) ∩ *set* (*normFaces newFs*) ⊆ {} ⟹ *distinct* (*normFaces* (*replace oldF newFs fs*))
⟨*proof*⟩

lemma *distinct-replacefacesAt1-norm*:
⋀*i*. *i* < |*Fss*| ⟹ *i* ∈ *set is* ⟹ *distinct is* ⟹ *distinct* (*normFaces* (*Fss*!*i*)) ⟹ *distinct* (*normFaces newFs*) ⟹
set (*normFaces* (*Fss* ! *i*)) ∩ *set* (*normFaces newFs*) ⊆ {} ⟹
distinct (*normFaces* ((*replacefacesAt is oldF newFs Fss*)! *i*))
⟨*proof*⟩

lemma *distinct-replacefacesAt2-norm*:
⋀*i*. *i* < |*Fss*| ⟹ *i* ∉ *set is* ⟹ *distinct is* ⟹ *distinct* (*normFaces* (*Fss*!*i*)) ⟹ *distinct* (*normFaces newFs*) ⟹
set (*normFaces* (*Fss* ! *i*)) ∩ *set* (*normFaces newFs*) ⊆ {} ⟹
distinct (*normFaces* ((*replacefacesAt is oldF newFs Fss*)! *i*))
⟨*proof*⟩

lemma *distinct-replacefacesAt-norm*:
⋀*i*. *i* < |*Fss*| ⟹ *distinct is* ⟹ *distinct* (*normFaces* (*Fss*!*i*)) ⟹ *distinct* (*normFaces newFs*) ⟹
set (*normFaces* (*Fss* ! *i*)) ∩ *set* (*normFaces newFs*) ⊆ {} ⟹
distinct (*normFaces* ((*replacefacesAt is olfF newFs Fss*)! *i*))
⟨*proof*⟩

lemma *normFace-in-cong*: *vertices f* ≠ [] ⟹ *minGraphProps g* ⟹ *normFace f* ∈ *set* (*normFaces* (*faces g*)) ⟹
∃ *f'* ∈ *set* (*faces g*). *f* ≅ *f'*
⟨*proof*⟩

lemma *normFace-neq*: *a* ∈ 𝒱 *f* ⟹ *a* ∉ 𝒱 *f'* ⟹ *vertices f'* ≠ [] ⟹ *normFace f* ≠ *normFace f'*
⟨*proof*⟩

lemma *split-face-f12-f21-neq-norm*:
pre-split-face oldF ram1 ram2 vs ⟹

$2 < |vertices\ oldF| \Longrightarrow 2 < |vertices\ f12| \Longrightarrow 2 < |vertices\ f21| \Longrightarrow$
$(f12, f21) = split\text{-}face\ oldF\ ram1\ ram2\ vs \Longrightarrow normFace\ f12 \neq normFace\ f21$
⟨*proof*⟩

lemma *normFace-in*: $f \in set\ fs \Longrightarrow normFace\ f \in set\ (normFaces\ fs)$
⟨*proof*⟩

13.6 Invariants of *splitFace*

lemma *splitFace-holds-minGraphProps'*:
$pre\text{-}splitFace\ g'\ v\ a\ f'\ vs \Longrightarrow minGraphProps'\ g' \Longrightarrow$
$minGraphProps'\ (snd\ (snd\ (splitFace\ g'\ v\ a\ f'\ vs)))$
⟨*proof*⟩

lemma *splitFace-holds-faceListAt-len*:
$pre\text{-}splitFace\ g'\ v\ a\ f'\ vs \Longrightarrow$
$minGraphProps\ g' \Longrightarrow$
$faceListAt\text{-}len\ (snd\ (snd\ (splitFace\ g'\ v\ a\ f'\ vs)))$
⟨*proof*⟩

lemma *splitFace-new-f12*:
$pre\text{-}splitFace\ g\ ram1\ ram2\ oldF\ newVs \Longrightarrow$
$minGraphProps\ g \Longrightarrow$
$(f12, f21, newGraph) = splitFace\ g\ ram1\ ram2\ oldF\ newVs \Longrightarrow$
$f12 \notin \mathcal{F}\ g$
⟨*proof*⟩

lemma *splitFace-new-f12-norm*:
$pre\text{-}splitFace\ g\ ram1\ ram2\ oldF\ newVs \Longrightarrow$
$minGraphProps\ g \Longrightarrow$
$(f12, f21, newGraph) = splitFace\ g\ ram1\ ram2\ oldF\ newVs \Longrightarrow$
$normFace\ f12 \notin set\ (normFaces\ (faces\ g))$
⟨*proof*⟩

lemma *splitFace-new-f21*:
$pre\text{-}splitFace\ g\ ram1\ ram2\ oldF\ newVs \Longrightarrow$
$minGraphProps\ g \Longrightarrow$
$(f12, f21, newGraph) = splitFace\ g\ ram1\ ram2\ oldF\ newVs \Longrightarrow$
$f21 \notin \mathcal{F}\ g$
⟨*proof*⟩

lemma *splitFace-new-f21-norm*:
$pre\text{-}splitFace\ g\ ram1\ ram2\ oldF\ newVs \Longrightarrow$
$minGraphProps\ g \Longrightarrow$
$(f12, f21, newGraph) = splitFace\ g\ ram1\ ram2\ oldF\ newVs \Longrightarrow$
$normFace\ f21 \notin set\ (normFaces\ (faces\ g))$

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *splitFace-f21-oldF-neq-norm*:
pre-splitFace g ram1 ram2 oldF newVs ⟹
minGraphProps g ⟹
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs* ⟹
normFace oldF ≠ *normFace f21*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *splitFace-f12-oldF-neq-norm*:
pre-splitFace g ram1 ram2 oldF newVs ⟹
minGraphProps g ⟹
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs* ⟹
normFace oldF ≠ *normFace f12*
⟨*proof*⟩

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*
⟨*proof*⟩

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*})
⟨*proof*⟩

declare *minGraphProps8 minGraphProps8a minGraphProps8a'* [*intro*]

lemma *splitFace-holds-facesAt-distinct*:
pre-splitFace g v w f [*countVertices g*..<*countVertices g* + *n*] ⟹

minGraphProps g $\Longrightarrow$
facesAt-distinct (*snd* (*snd* (*splitFace g v w f* [*countVertices g*..<*countVertices g* + *n*])))
⟨*proof*⟩

lemma *splitFace-holds-facesAt-eq*:
pre-splitFace g′ v a f′ [*countVertices g′*..<*countVertices g′* + *n*] $\Longrightarrow$
minGraphProps g′ $\Longrightarrow$
g″ = (*snd* (*snd* (*splitFace g′ v a f′* [*countVertices g′*..<*countVertices g′* + *n*]))) $\Longrightarrow$
facesAt-eq g″
⟨*proof*⟩

lemma *splitFace-holds-faces-subset*:
pre-splitFace g′ v a f′ [*countVertices g′*..<*countVertices g′* + *n*] $\Longrightarrow$
minGraphProps g′ $\Longrightarrow$
faces-subset (*snd* (*snd* (*splitFace g′ v a f′* [*countVertices g′*..<*countVertices g′* + *n*])))
⟨*proof*⟩

lemma *splitFace-holds-edges-sym*:
pre-splitFace g′ v a f′ ws $\Longrightarrow$
minGraphProps g′ $\Longrightarrow$
edges-sym (*snd* (*snd* (*splitFace g′ v a f′ ws*)))
⟨*proof*⟩

lemma *splitFace-holds-faces-distinct*:
pre-splitFace g′ v a f′ [*countVertices g′*..<*countVertices g′* + *n*] $\Longrightarrow$
minGraphProps g′ $\Longrightarrow$
faces-distinct (*snd* (*snd* (*splitFace g′ v a f′* [*countVertices g′*..<*countVertices g′* + *n*])))
⟨*proof*⟩

lemma *help*:
shows *xs* ≠ [] $\Longrightarrow$ *x* ∉ *set xs* $\Longrightarrow$ *x* ≠ *hd xs* **and**
xs ≠ [] $\Longrightarrow$ *x* ∉ *set xs* $\Longrightarrow$ *x* ≠ *last xs* **and**
xs ≠ [] $\Longrightarrow$ *x* ∉ *set xs* $\Longrightarrow$ *hd xs* ≠ *x* **and**
xs ≠ [] $\Longrightarrow$ *x* ∉ *set xs* $\Longrightarrow$ *last xs* ≠ *x*
⟨*proof*⟩

lemma *split-face-edge-disj*:
⟦ *pre-split-face f a b vs*; (*f*$_1$, *f*$_2$) = *split-face f a b vs*; |*vertices f*| ≥ *3*;
vs = [] ⟶ (*a*,*b*) ∉ *edges f* ∧ (*b*,*a*) ∉ *edges f* ⟧
$\Longrightarrow$ $\mathcal{E}$ *f*$_1$ ∩ $\mathcal{E}$ *f*$_2$ = {}
⟨*proof*⟩

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*⟩

lemma *splitFace-edges-disj2*:
minGraphProps g $\Longrightarrow$ *pre-splitFace g u v f vs*
$\Longrightarrow$ *edges-disj*(*snd*(*snd*(*splitFace g u v f vs*)))
⟨*proof*⟩

lemma *vertices-conv-Union-edges2*:
distinct(*vertices f*) $\Longrightarrow \mathcal{V}(f::face) = (\bigcup (a,b)\in\mathcal{E}\ f.\ \{b\})$
⟨*proof*⟩

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*⟩

lemma *splitFace-face-face-op2*:
minGraphProps g $\Longrightarrow$ *pre-splitFace g u v f vs*
$\Longrightarrow$ *face-face-op*(*snd*(*snd*(*splitFace g u v f vs*)))
⟨*proof*⟩

lemma *splitFace-holds-minGraphProps*:
pre-splitFace g' v a f' [*countVertices g'*..<*countVertices g'* + *n*] $\Longrightarrow$
minGraphProps g' $\Longrightarrow$
minGraphProps (*snd* (*snd* (*splitFace g' v a f'* [*countVertices g'*..<*countVertices g'* + *n*])))
⟨*proof*⟩

13.7 Invariants of *makeFaceFinal*

lemma *MakeFaceFinal-minGraphProps'*:
f $\in \mathcal{F}$ *g* $\Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *minGraphProps'* (*makeFaceFinal f g*)
⟨*proof*⟩

lemma *MakeFaceFinal-facesAt-eq*:
f $\in \mathcal{F}$ *g* $\Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *facesAt-eq* (*makeFaceFinal f g*)
⟨*proof*⟩

lemma *MakeFaceFinal-faceListAt-len*:
f $\in \mathcal{F}$ *g* $\Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *faceListAt-len* (*makeFaceFinal f g*)
⟨*proof*⟩

lemma *normFaces-makeFaceFinalFaceList*: (*normFaces* (*makeFaceFinalFaceList f fs*)) = (*normFaces fs*)
⟨*proof*⟩

lemma *MakeFaceFinal-facesAt-distinct*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *facesAt-distinct* (*makeFaceFinal f g*)
⟨*proof*⟩

lemma *MakeFaceFinal-faces-subset*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *faces-subset* (*makeFaceFinal f g*)
⟨*proof*⟩

lemma *MakeFaceFinal-edges-sym*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *edges-sym* (*makeFaceFinal f g*)
⟨*proof*⟩

lemma *MakeFaceFinal-faces-distinct*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *faces-distinct* (*makeFaceFinal f g*)
⟨*proof*⟩

lemma *MakeFaceFinal-edges-disj*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *edges-disj* (*makeFaceFinal f g*)
⟨*proof*⟩

lemma *MakeFaceFinal-face-face-op*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *face-face-op* (*makeFaceFinal f g*)
⟨*proof*⟩

lemma *MakeFaceFinal-minGraphProps*:
$f \in \mathcal{F}\ g \Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *minGraphProps* (*makeFaceFinal f g*)
⟨*proof*⟩

13.8 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*⟩

syntax *Edges-if* :: *face* ⇒ *vertex* ⇒ *vertex* ⇒ (*vertex* × *vertex*)*set*
translations
Edges-if f u v =>
if u=*v then* {} *else Edges*(*u* # *between* (*vertices f*) *u v* @ [*v*])

lemma *FaceDivsionGraph-one-final-but*:
assumes *mgp*: *minGraphProps g* **and** *pre*: *pre-splitFace g u v f vs*
and *fdg*: $(f_1, f_2, g') = splitFace\ g\ u\ v\ f\ vs$
and *nrv*: $r \neq v$
and *ruv*: *before* (*verticesFrom f r*) *u v* **and** *rf*: $r \in \mathcal{V}\ f$
and *1*: *one-final-but g* (*Edges-if f r u*)
shows *one-final-but g'* (*Edges*(*r* # *between* (*vertices* f_2) *r v* @ [*v*]))
⟨*proof*⟩

lemma *one-final-but-makeFaceFinal*:
⟦ *minGraphProps g*; *one-final-but g E*; $E \subseteq \mathcal{E}\ f$; $f \in \mathcal{F}\ g$; ¬ *final f* ⟧ ⟹
one-final (*makeFaceFinal f g*)
⟨*proof*⟩

lemma *one-final-subdivFace′*:
⋀*f v n g*.
pre-subdivFace′ g f u v n ovs ⟹ *minGraphProps g* ⟹ $f \in \mathcal{F}\ g$ ⟹
one-final-but g (*Edges-if f u v*) ⟹
one-final(*subdivFace′ g f v n ovs*)
⟨*proof*⟩

lemma *neighbors-edges*:
minGraphProps g ⟹ *b* ∈ *set* (*neighbors g a*) = ((*a*, *b*) ∈ *edges g*)
⟨*proof*⟩

lemma *no-self-edges*: *minGraphProps′ g* ⟹ (*a*, *a*) ∉ *edges g* ⟨*proof*⟩

Requires only *distinct* (*vertices f*) and that *g* has no self-loops.

lemma *duplicateEdge-is-duplicateEdge-eq*:
minGraphProps g ⟹ $f \in \mathcal{F}\ g$ ⟹ $a \in \mathcal{V}\ f$ ⟹ $b \in \mathcal{V}\ f$ ⟹
duplicateEdge g f a b = *is-duplicateEdge g f a b*
⟨*proof*⟩

lemma *incrIndexList-less-eq*:
incrIndexList ls m nmax ⟹ *Suc n* < |*ls*| ⟹ *ls*!*n* ≤ *ls*!*Suc n*
⟨*proof*⟩

lemma *incrIndexList-less*:
incrIndexList ls m nmax ⟹ *Suc n* < |*ls*| ⟹ *ls*!*n* ≠ *ls*!*Suc n*⟹ *ls*!*n* < *ls*!*Suc n*
⟨*proof*⟩

lemma *Seed-holds-minGraphProps′*: *minGraphProps′* (*Seed p*)
⟨*proof*⟩

lemma *Seed-holds-facesAt-eq*: *facesAt-eq* (*Seed p*)
⟨*proof*⟩

lemma *minVertex-zero1*: *minVertex* (*Face* [*0..z*] *Final*) = *0*
⟨*proof*⟩

lemma *minVertex-zero2*: *minVertex* (*Face* (*rev* [*0..z*]) *Nonfinal*) = *0*
⟨*proof*⟩

13.9 Invariants of *Seed*

lemma *Seed-holds-facesAt-distinct*: *facesAt-distinct* (*Seed p*)
⟨*proof*⟩

lemma *Seed-holds-faces-subset*: *faces-subset* (*Seed p*)
⟨*proof*⟩

lemma *Seed-holds-edges-sym*: *edges-sym* (*Seed p*)
⟨*proof*⟩

lemma *Seed-holds-edges-disj*: *edges-disj* (*Seed p*)
⟨*proof*⟩

lemma *Seed-holds-faces-distinct*: *faces-distinct* (*Seed p*)
⟨*proof*⟩

lemma *Seed-holds-faceListAt-len*: *faceListAt-len* (*Seed p*)
⟨*proof*⟩

lemma *face-face-op-Seed*: *face-face-op*(*Seed p*)
⟨*proof*⟩

lemma *one-final-Seed*: *one-final* $Seed_p$
⟨*proof*⟩

lemma *two-face-Seed*: $|faces\ Seed_p| \geq 2$
⟨*proof*⟩

lemma *inv-Seed*: *inv* (*Seed p*)
⟨*proof*⟩

lemma *pre-subdivFace-indexToVertexList*:
assumes *mgp*: *minGraphProps g* **and** *f*: *f* ∈ *set* (*nonFinals g*)
and *v*: *v* ∈ $\mathcal{V}$ *f* **and** *e*: *e* ∈ *set* (*enumerator i* |*vertices f*|)
and *containsNot*: ¬ *containsDuplicateEdge g f v e* **and** *i*: *2* < *i*

shows *pre-subdivFace g f v (indexToVertexList f v e)*
⟨*proof*⟩

13.10 Increasing properties of *subdivFace′*

lemma *subdivFace′-incr*:
assumes *Ptrans*: !!*x y z*. *Q x y* ⟹ *P y z* ⟹ *P x z*
and *mkFin*: !!*f g*. *f* ∈ ℱ *g* ⟹ ¬ *final f* ⟹ *P g* (*makeFaceFinal f g*)
and *fdg-incr*: !! *g u v f vs*.
pre-splitFace g u v f vs ⟹
Q g (*snd*(*snd*(*splitFace g u v f vs*)))
shows
⋀*f′ v n g*. *pre-subdivFace′ g f′ v′ v n ovl* ⟹
minGraphProps g ⟹ *f′* ∈ ℱ *g* ⟹ *P g* (*subdivFace′ g f′ v n ovl*)
⟨*proof*⟩

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*⟩

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′*
⟨*proof*⟩

13.10.1 Increasing number of faces

lemma *splitFace-incr-faces*:
pre-splitFace g u v f vs ⟹
finals(*snd*(*snd*(*splitFace g u v f vs*))) = *finals g* ∧
|*nonFinals*(*snd*(*snd*(*splitFace g u v f vs*)))| = *Suc* |*nonFinals g*|
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *next-plane0-incr-faces*:
minGraphProps g $\Longrightarrow$ *g* [*next-plane0*$_p$]$\rightarrow$ *g'* $\Longrightarrow$
|*finals g'*| = |*finals g*|+*1* $\wedge$ |*nonFinals g'*| $\geq$ |*nonFinals g*| $-$ *1*
⟨*proof*⟩

lemma *two-faces-subdivFace'*:
pre-subdivFace' g f u v n ovs $\Longrightarrow$ *minGraphProps g* $\Longrightarrow$ *f* $\in \mathcal{F}$ *g* $\Longrightarrow$
|*faces g*| $\geq$ *2* $\Longrightarrow$ |*faces*(*subdivFace' g f v n ovs*)| $\geq$ *2*
⟨*proof*⟩

13.11 Main invariant theorems

lemma *inv-genPoly*:
assumes *inv*: *inv g* **and** *polygen*: *g'* $\in$ *set*(*generatePolygon i v f g*)
and *f*: *f* $\in$ *set* (*nonFinals g*) **and** *i*: *2* $<$ *i* **and** *v*: *v* $\in \mathcal{V}$ *f*
shows *inv g'*
⟨*proof*⟩

lemma *inv-inv-next-plane0*: *invariant inv next-plane0*$_p$
⟨*proof*⟩

end

14 Further Plane Graph Poroperties

theory *PlaneProps*
imports *Invariants*
begin

14.1 *final*

lemma *plane-final-facesAt*:
⟦ *inv g*; *final g*; *f* $\in$ *set* (*facesAt g v*) ⟧ $\Longrightarrow$ *final f*
⟨*proof*⟩

lemma *finalVertexI*:
⟦ *inv g*; *final g*; *v* $\in \mathcal{V}$ *g* ⟧ $\Longrightarrow$ *finalVertex g v*
⟨*proof*⟩

lemma *setFinal-notin-finals*:
⟦ *f* $\in \mathcal{F}$ *g*; $\neg$ *final f*; *minGraphProps g* ⟧ $\Longrightarrow$ *setFinal f* $\notin$ *set* (*finals g*)
⟨*proof*⟩

14.2 *degree*

lemma *planeN4*: $inv\ g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow 3 \leq |vertices\ f|$
⟨*proof*⟩

lemma *degree-eq*:
assumes *pl*: *inv g* **and** *fin*: *final g*
shows $degree\ g\ v = tri\ g\ v + quad\ g\ v + except\ g\ v$
⟨*proof*⟩

lemma *plane-fin-exceptionalVertex-def*:
assumes *pl*: *inv g* **and** *fin*: *final g*
shows $exceptionalVertex\ g\ v =$
$(\ |\ [f \in facesAt\ g\ v\ .\ 5 \leq |vertices\ f|\ ]\ | \neq 0)$
⟨*proof*⟩

lemma *not-exceptional*:
$inv\ g \Longrightarrow final\ g \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow$
$\neg\ exceptionalVertex\ g\ v \Longrightarrow |vertices\ f| \leq 4$
⟨*proof*⟩

14.3 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 $g' \in set\ (next\text{-}plane0_p\ g)$
⟨*proof*⟩

lemma *next-plane0-nonfinals*: $g\ [next\text{-}plane0_p]\rightarrow g' \Longrightarrow nonFinals\ g \neq []$
⟨*proof*⟩

lemma *next-plane0-ex*:
assumes *a*: $g\ [next\text{-}plane0_p]\rightarrow g'$
shows $\exists f \in set(nonFinals\ g).\ \exists v \in \mathcal{V}\ f.\ \exists i \in set([3..maxGon\ p]).$
$g' \in set\ (generatePolygon\ i\ v\ f\ g)$
⟨*proof*⟩

lemma *step-outside2*:
$inv\ g \Longrightarrow g\ [next\text{-}plane0_p]\rightarrow g' \Longrightarrow \neg\ final\ g' \Longrightarrow |faces\ g'| \neq 2$
⟨*proof*⟩

14.4 Increasing final faces

lemma *set-finals-splitFace*[*simp*]:
$[\![\ f \in \mathcal{F}\ g; \neg\ final\ f\ ]\!] \Longrightarrow$
$set(finals(snd(snd(splitFace\ g\ u\ v\ f\ vs)))) = set(finals\ g)$

⟨*proof*⟩

lemma *next-plane0-finals-incr*:
$g\ [\textit{next-plane0}_p]\rightarrow g' \Longrightarrow f \in \textit{set}(\textit{finals}\ g) \Longrightarrow f \in \textit{set}(\textit{finals}\ g')$
⟨*proof*⟩

lemma *next-plane0-finals-subset*:
$g' \in \textit{set}\ (\textit{next-plane0}_p\ g) \Longrightarrow$
$\textit{set}\ (\textit{finals}\ g) \subseteq \textit{set}\ (\textit{finals}\ g')$
⟨*proof*⟩

lemma *next-plane0-final-mono*:
$[\![\ g' \in \textit{set}\ (\textit{next-plane0}_p\ g); f \in \mathcal{F}\ g; \textit{final}\ f\ ]\!] \Longrightarrow f \in \mathcal{F}\ g'$
⟨*proof*⟩

14.5 Increasing vertices

lemma *next-plane0-vertices-subset*:
$[\![\ g' \in \textit{set}\ (\textit{next-plane0}_p\ g); \textit{minGraphProps}\ g\ ]\!] \Longrightarrow \mathcal{V}\ g \subseteq \mathcal{V}\ g'$
⟨*proof*⟩

14.6 Increasing vertex degrees

lemma *next-plane0-incr-faceListAt*:
$[\![\ g' \in \textit{set}\ (\textit{next-plane0}_p\ g); \textit{minGraphProps}\ g\ ]\!]$
$\Longrightarrow |\textit{faceListAt}\ g| \leq |\textit{faceListAt}\ g'|\ \&$
$(\forall v < |\textit{faceListAt}\ g|.\ |\textit{faceListAt}\ g\ !\ v| \leq |\textit{faceListAt}\ g'\ !\ v|\ )$
(**concl is** *?Q g g'*)
⟨*proof*⟩

lemma *next-plane0-incr-degree*:
$[\![\ g' \in \textit{set}\ (\textit{next-plane0}_p\ g); \textit{minGraphProps}\ g; v \in \mathcal{V}\ g\ ]\!]$
$\Longrightarrow \textit{degree}\ g\ v \leq \textit{degree}\ g'\ v$
⟨*proof*⟩

14.7 Increasing *except*

lemma *next-plane0-incr-except*:
assumes $g' \in \textit{set}\ (\textit{next-plane0}_p\ g)\ \textit{inv}\ g\ v \in \mathcal{V}\ g$
shows $\textit{except}\ g\ v \leq \textit{except}\ g'\ v$
⟨*proof*⟩

14.8 Increasing edges

lemma *next-plane0-set-edges-subset*:
$[\![\ \textit{minGraphProps}\ g;\ \ g\ [\textit{next-plane0}_p]\rightarrow g'\ ]\!] \Longrightarrow \textit{edges}\ g \subseteq \textit{edges}\ g'$
⟨*proof*⟩

14.9 Increasing final vertices

lemma *next-plane0-incr-finV*:
$[\![g' \in set\ (next\text{-}plane0_p\ g);\ minGraphProps\ g\ ]\!]$
$\Longrightarrow \forall v \in \mathcal{V}\ g.\ v \in \mathcal{V}\ g' \wedge$
$((\forall f \in \mathcal{F}\ g.\ v \in \mathcal{V}\ f \longrightarrow final\ f) \longrightarrow$
$(\forall f \in \mathcal{F}\ g'.\ v \in \mathcal{V}\ f \longrightarrow f \in \mathcal{F}\ g))$ (**concl is** *?Q g g'*)
⟨*proof*⟩

lemma *next-plane0-finalVertex-mono*:
$[\![g' \in set\ (next\text{-}plane0_p\ g);\ inv\ g;\ u \in \mathcal{V}\ g;\ finalVertex\ g\ u\ ]\!]$
$\Longrightarrow finalVertex\ g'\ u$
⟨*proof*⟩

14.10 Preservation of *facesAt* at final vertices

lemma *next-plane0-finalVertex-facesAt-eq*:
$[\![g' \in set\ (next\text{-}plane0_p\ g);\ inv\ g;\ v \in \mathcal{V}\ g;\ finalVertex\ g\ v\ ]\!]$
$\Longrightarrow set(facesAt\ g'\ v) = set(facesAt\ g\ v)$
⟨*proof*⟩

lemma *next-plane0-len-filter-eq*:
assumes $g' \in set\ (next\text{-}plane0_p\ g)$ $inv\ g$ $v \in \mathcal{V}\ g$ $finalVertex\ g\ v$
shows $|filter\ P\ (facesAt\ g'\ v)| = |filter\ P\ (facesAt\ g\ v)|$
⟨*proof*⟩

14.11 Properties of *subdivFace′*

lemma *new-edge-subdivFace′*:
$\bigwedge f\ v\ n\ g.$
$pre\text{-}subdivFace'\ g\ f\ u\ v\ n\ ovs \Longrightarrow minGraphProps\ g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow$
$subdivFace'\ g\ f\ v\ n\ ovs = makeFaceFinal\ f\ g \vee$
$(\forall f' \in \mathcal{F}\ (subdivFace'\ g\ f\ v\ n\ ovs) - (\mathcal{F}\ g - \{f\}).$
$\exists e \in \mathcal{E}\ f'.\ e \notin \mathcal{E}\ g)$
⟨*proof*⟩

lemma *dist-edges-subdivFace′*:
$pre\text{-}subdivFace'\ g\ f\ u\ v\ n\ ovs \Longrightarrow minGraphProps\ g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow$
$subdivFace'\ g\ f\ v\ n\ ovs = makeFaceFinal\ f\ g \vee$
$(\forall f' \in \mathcal{F}\ (subdivFace'\ g\ f\ v\ n\ ovs) - (\mathcal{F}\ g - \{f\}).\ \mathcal{E}\ f' \neq \mathcal{E}\ f)$
⟨*proof*⟩

lemma *between-last*: $[\![\ distinct(vertices\ f);\ u \in \mathcal{V}\ f\ ]\!] \Longrightarrow$
$between\ (vertices\ f)\ u\ (last\ (verticesFrom\ f\ u)) =$
$butlast(tl(verticesFrom\ f\ u))$

⟨*proof*⟩

lemma *final-subdivFace'*: $\bigwedge f\ u\ n\ g.\ minGraphProps\ g \Longrightarrow$
$pre\text{-}subdivFace'\ g\ f\ r\ u\ n\ ovs \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow$
$(ovs = [] \longrightarrow n{=}0 \wedge u = last(verticesFrom\ f\ r)) \Longrightarrow$
$\exists f' \in set(finals(subdivFace'\ g\ f\ u\ n\ ovs)) - set(finals\ g).$
$(f^{-1} \cdot r,r) \in \mathcal{E}\ f' \wedge\ |vertices\ f'| =$
$n + |ovs| + (if\ r{=}u\ then\ 1\ else\ |between\ (vertices\ f)\ r\ u| + 2)$
⟨*proof*⟩

lemma *subdivFace'-final-base*: $\bigwedge f\ u\ n\ g.\ minGraphProps\ g \Longrightarrow$
$pre\text{-}subdivFace'\ g\ f\ r\ u\ n\ ovs \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow$
$\exists f' \in \mathcal{F}\ (subdivFace'\ g\ f\ u\ n\ ovs).\ final\ f' \wedge (f^{-1} \cdot r,r) \in \mathcal{E}\ f'$
⟨*proof*⟩

lemma *Seed-max-final-ex*:
$\exists f{\in}set\ (finals\ (Seed\ p)).\ |vertices\ f| = maxGon\ p$
⟨*proof*⟩

lemma *max-face-ex*: **assumes** a: $Seed_p\ [next\text{-}plane0_p]{\rightarrow}*\ g$
shows $\exists f \in set\ (finals\ g).\ |vertices\ f| = maxGon\ p$
⟨*proof*⟩

end

15 Summation Over Lists

theory *ListSum*
imports *ListAux*
begin

consts *ListSum* :: *'b list* ⇒ (*'b* ⇒ *'a*::*comm-monoid-add*) ⇒ *'a*::*comm-monoid-add*

primrec
ListSum [] *f* = *0*
ListSum (*l*#*ls*) *f* = *f l* + *ListSum ls f*

syntax -*ListSum* :: *idt* ⇒ *'b list* ⇒ (*'a*::*comm-monoid-add*) ⇒
(*'a*::*comm-monoid-add*) (∑ -∈- -)
translations $\sum x{\in}xs\ f$ == *ListSum xs* (λ*x*. *f*)

consts *natListSum* :: *'b list* ⇒ (*'b* ⇒ *nat*) ⇒ *nat*
primrec
natListSum [] *f* = *0*
natListSum (*l*#*ls*) *f* = *f l* + *natListSum ls f*

consts *intListSum* :: *'b list* ⇒ (*'b* ⇒ *int*) ⇒ *int*
primrec
intListSum [] *f* = *0*
intListSum (*l*#*ls*) *f* = *f l* + *intListSum ls f*

lemma [*THEN eq-reflection*, *code unfold*]: ((*ListSum ls f*)::*nat*) = *natListSum ls f*
⟨*proof*⟩

lemma [*THEN eq-reflection*, *code unfold*]: ((*ListSum ls f*)::*int*) = *intListSum ls f*
⟨*proof*⟩

lemma [*simp*]: ∑ *v* ∈ *V* *0* = (*0*::*nat*) ⟨*proof*⟩

lemma *ListSum-compl1*:
(∑ *x* ∈ [*x*∈*xs*. ¬ *P x*] *f x*) + ∑ *x* ∈ [*x*∈*xs*. *P x*] *f x* = ∑ *x* ∈ *xs* (*f x*::*nat*)
⟨*proof*⟩

lemma *ListSum-compl2*:
(∑ *x* ∈ [*x*∈*xs*. *P x*] *f x*) + ∑ *x* ∈ [*x*∈*xs*. ¬ *P x*] *f x* = ∑ *x* ∈ *xs* (*f x*::*nat*)
⟨*proof*⟩

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

lemma *ListSum-conv-setsum*:
distinct xs ⟹ *ListSum xs f* = *setsum f* (*set xs*)
⟨*proof*⟩

lemma *listsum-cong*:
⟦ *xs* = *ys*; ⋀*y*. *y* ∈ *set ys* ==> *f y* = *g y* ⟧
⟹ *ListSum xs f* = *ListSum ys g*
⟨*proof*⟩

lemma *strong-listsum-cong*[*cong*]:
⟦ *xs* = *ys*; ⋀*y*. *y* ∈ *set ys* =*simp*=> *f y* = *g y* ⟧
⟹ *ListSum xs f* = *ListSum ys g*

⟨*proof*⟩

lemma *ListSum-eq* [*trans*]:
(⋀*v*. *v* ∈ *set V* ⟹ *f v* = *g v*) ⟹ ∑ *v* ∈ *V f v* = ∑ *v* ∈ *V g v*
⟨*proof*⟩

lemma *ListSum-set-eq*:
⋀*C*. *distinct B* ⟹ *distinct C* ⟹ *set B* = *set C* ⟹
∑ *a* ∈ *B f a* = ∑ *a* ∈ *C* (*f a*::*nat*)
⟨*proof*⟩

lemma *ListSum-disj-union*:
distinct A ⟹ *distinct B* ⟹ *distinct C* ⟹
set C = *set A* ∪ *set B* ⟹
set A ∩ *set B* = {} ⟹
∑ *a* ∈ *C* (*f a*) = (∑ *a* ∈ *A f a*) + (∑ *a* ∈ *B* (*f a*::*nat*))
⟨*proof*⟩

constdefs *separating* :: *'a set* ⇒ (*'a* ⇒ *'b set*) ⇒ *bool*
separating V F ≡
(∀ *v1* ∈ *V*. ∀ *v2* ∈ *V*. *v1* ≠ *v2* ⟶ *F v1* ∩ *F v2* = {})

lemma *separating-insert1*:
separating (*insert a V*) *F* ⟹ *separating V F*
⟨*proof*⟩

lemma *separating-insert2*:
separating (*insert a V*) *F* ⟹ *a* ∉ *V* ⟹ *v* ∈ *V* ⟹
F a ∩ *F v* = {}
⟨*proof*⟩

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*⟩

lemma *listsum-const*[*simp*]:
∑ *x* ∈ *xs k* = *length xs* ∗ *k*
⟨*proof*⟩

lemma *ListSum-add*:
(∑ *x* ∈ *V f x*) + ∑ *x* ∈ *V g x* = ∑ *x* ∈ *V* (*f x* + (*g x*::*nat*))

⟨*proof*⟩

lemma *ListSum-le*:
(⋀*v*. *v* ∈ *set V* ⟹ *f v* ≤ *g v*) ⟹ ∑ *v* ∈ *V f v* ≤ ∑ *v* ∈ *V* (*g v*::*nat*)
⟨*proof*⟩

lemma *ListSum1-bound*:
a ∈ *set F* ⟹ (*d a*::*nat*) ≤ ∑ *f* ∈ *F d f*
⟨*proof*⟩

lemma *ListSum2-bound*:
a ∈ *set F* ⟹ *b* ∈ *set F* ⟹ *a* ≠ *b* ⟹ *d a* + *d b* ≤ ∑ *f* ∈ *F* (*d f*::*nat*)
⟨*proof*⟩

end

16 Tameness

theory *Tame*
imports *Graph ListSum*
begin

16.1 Constants

constdefs *squanderTarget* :: *nat*
squanderTarget ≡ *14800*

constdefs *excessTCount* :: *nat* ⇒ *nat*
a *t* ≡ *if t* < *3 then squanderTarget*
else if t = *3 then 1400*
else if t = *4 then 1500*
else 0

constdefs *squanderVertex* :: *nat* ⇒ *nat* ⇒ *nat*
b *p q* ≡ *if p* = *0* ∧ *q* = *3 then 7135*
else if p = *0* ∧ *q* = *4 then 10649*
else if p = *1* ∧ *q* = *2 then 6950*
else if p = *1* ∧ *q* = *3 then 7135*
else if p = *2* ∧ *q* = *1 then 8500*
else if p = *2* ∧ *q* = *2 then 4756*
else if p = *2* ∧ *q* = *3 then 12981*
else if p = *3* ∧ *q* = *1 then 3642*
else if p = *3* ∧ *q* = *2 then 8334*
else if p = *4* ∧ *q* = *0 then 4139*
else if p = *4* ∧ *q* = *1 then 3781*
else if p = *5* ∧ *q* = *0 then 550*
else if p = *5* ∧ *q* = *1 then 11220*
else if p = *6* ∧ *q* = *0 then 6339*

else squanderTarget

constdefs *scoreFace* :: *nat* ⇒ *int*
c *n* ≡ *if n* = *3 then 1000*
else if n = *4 then 0*
else if n = *5 then* −*1030*
else if n = *6 then* −*2060*
else if n = *7 then* −*3030*
else if n = *8 then* −*3030*
else −*3030*

constdefs *squanderFace* :: *nat* ⇒ *nat*
d *n* ≡ *if n* = *3 then 0*
else if n = *4 then 2378*
else if n = *5 then 4896*
else if n = *6 then 7414*
else if n = *7 then 9932*
else if n = *8 then 10916*
else squanderTarget

16.2 Separated sets of vertices

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

1. For each vertex in V there is an exceptional face containing it:

constdefs *separated*$_1$ :: *graph* ⇒ *vertex set* ⇒ *bool*
separated$_1$ *g V* ≡ ∀ *v* ∈ *V*. *except g v* ≠ *0*

2. No two vertices in V are adjacent:

constdefs *separated*$_2$ :: *graph* ⇒ *vertex set* ⇒ *bool*
separated$_2$ *g V* ≡ ∀ *v* ∈ *V*. ∀ *f* ∈ *set* (*facesAt g v*). *f*•*v* ∉ *V*

3. No two vertices lie on a common quadrilateral:

constdefs *separated*$_3$ :: *graph* ⇒ *vertex set* ⇒ *bool*
separated$_3$ *g V* ≡
∀ *v* ∈ *V*. ∀ *f* ∈ *set* (*facesAt g v*). |*vertices f*|≤*4* ⟶ $\mathcal{V}$ *f* ∩ *V* = {*v*}

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

constdefs *preSeparated* :: *graph* ⇒ *vertex set* ⇒ *bool*
preSeparated g V ≡ *separated*$_2$ *g V* ∧ *separated*$_3$ *g V*

4. Every vertex in V has degree 5:

constdefs *separated*$_4$ :: *graph* ⇒ *vertex set* ⇒ *bool*
separated$_4$ *g V* ≡ ∀ *v* ∈ *V*. *degree g v* = *5*

constdefs *separated* :: *graph* ⇒ *vertex set* ⇒ *bool*
separated g V ≡
separated$_1$ *g V* ∧ *separated*$_2$ *g V* ∧ *separated*$_3$ *g V* ∧ *separated*$_4$ *g V*

16.3 Admissible weight assignments

A weight assignment $w :: face \Rightarrow nat$ assigns a natural number to every face.

We formalize the admissibility requirements as follows:

1. $\mathrm{d}(|f|) \leq w(f)$:

constdefs $admissible_1 :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$
$admissible_1\ w\ g \equiv \forall f \in \mathcal{F}\ g.\ \mathrm{d}\ |vertices\ f| \leq w\ f$

2. If v has type (p, q), then $\mathrm{b}(p, q) \leq \sum_{v \in f} w(f)$:

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

4. Let V be any separated set of vertices.
Then $\sum_{v \in V} \mathrm{a}(tri(v)) \leq \sum_{V \cap f \neq \emptyset} (w(f) - d(|f|))$:

constdefs $admissible_3 :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$
$admissible_3\ w\ g \equiv$
$\forall V.\ separated\ g\ (set\ V) \wedge set\ V \subseteq \mathcal{V}\ g \longrightarrow$
$(\sum_{v \in V} \mathrm{a}\ (tri\ g\ v))$
$+ (\sum_{f \in [f \in faces\ g.\ \exists v \in set\ V.\ f \in set\ (facesAt\ g\ v)]} \mathrm{d}\ |vertices\ f|\ )$
$\leq \sum_{f \in [f \in faces\ g.\ \exists v \in set\ V.\ f \in set\ (facesAt\ g\ v)]} w\ f$

Finally we define admissibility of weights functions.

constdefs $admissible :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$
$admissible\ w\ g \equiv admissible_1\ w\ g \wedge admissible_2\ w\ g \wedge admissible_3\ w\ g$

16.4 Tameness

1. The length of each face is (at least 3 and) at most 8:

constdefs $tame_1 :: graph \Rightarrow bool$
$tame_1\ g \equiv \forall f \in \mathcal{F}\ g.\ 3 \leq |vertices\ f| \wedge |vertices\ f| \leq 8$

2. Every 3-cycle is a face or the opposite of a face:

A face given by a vertex list *vs* is contained in a graph g, if it is isomorphic to one of the faces in g. The notation $f \in_{\cong} F$ means $\exists f' \in F.\ f \cong f'$, where $\cong$ is the equivalence relation on faces (see Chapter **??**).

The notions *is-path* and *is-cycle* are defined locally (rather than in the theory of graphs) because no lemmas need to be proved about them because the generation of tame graphs uses these same executable functions as the definition of tameness. Hence there is nothing to prove.

consts $is\text{-}path :: graph \Rightarrow vertex\ list \Rightarrow bool$

primrec
is-path g [] = *True*
is-path g (*u*#*vs*) = (*case vs of* [] ⇒ *True*
| *v*#*ws* ⇒ *v* ∈ *set*(*neighbors g u*) ∧ *is-path g vs*)

constdefs
is-cycle :: *graph* ⇒ *vertex list* ⇒ *bool*
is-cycle g vs ≡ *hd vs* ∈ *set*(*neighbors g* (*last vs*)) ∧ *is-path g vs*

constdefs *tame*$_2$ :: *graph* ⇒ *bool*
tame$_2$ *g* ≡
∀ *a b c*. *is-cycle g* [*a*,*b*,*c*] ∧ *distinct* [*a*,*b*,*c*] ⟶
(∃ *f* ∈ $\mathcal{F}$ *g*. *vertices f* ≅ [*a*,*b*,*c*] ∨ *vertices f* ≅ [*c*,*b*,*a*])

3. Every 4-cycle surrounds one of the following configurations:

constdefs *tameConf*$_1$:: *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex list list*
tameConf$_1$ *a b c d* ≡ [[*a*,*b*,*c*,*d*]]

constdefs *tameConf*$_2$:: *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex list list*
tameConf$_2$ *a b c d* ≡ [[*a*,*b*,*c*], [*a*,*c*,*d*]]

constdefs *tameConf*$_3$ :: *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex list list*
tameConf$_3$ *a b c d e* ≡ [[*a*,*b*,*e*], [*b*,*c*,*e*], [*a*,*e*,*c*,*d*]]

constdefs *tameConf*$_4$ :: *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex list list*
tameConf$_4$ *a b c d e* ≡ [[*a*,*b*,*e*], [*b*,*c*,*e*], [*c*,*d*,*e*], [*d*,*a*,*e*]]

Given a fixed 4-cycle and using the convention of drawing faces clockwise, a tame configuration can occur in the 'interior' or on the outside of the 4-cycle. For configuration *tameConf*$_2$ there are two possible rotations of the triangles, for configuration *tameConf*$_3$ there are 4. The notation $F_1 \subseteq_{\cong} F_2$ means $\forall f \in F_1.\ f \in_{\cong} F_2$.

Note that our definition only ensures the existence of certain faces in the graph, not the fact that no other faces of the graph may lie in the interior or on the outside. Hence it is slightly weaker than the definition in Hales' paper.

constdefs *tame-quad* :: *graph* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *bool*
tame-quad g a b c d ≡
set(*tameConf*$_1$ *a b c d*) ⊆$_{\cong}$ *vertices* ‘ $\mathcal{F}$ *g*
∨ *set*(*tameConf*$_2$ *a b c d*) ⊆$_{\cong}$ *vertices* ‘ $\mathcal{F}$ *g*
∨ *set*(*tameConf*$_2$ *b c d a*) ⊆$_{\cong}$ *vertices* ‘ $\mathcal{F}$ *g*
∨ (∃ *e* ∈ $\mathcal{V}$ *g* − {*a*,*b*,*c*,*d*}.
set(*tameConf*$_3$ *a b c d e*) ⊆$_{\cong}$ *vertices* ‘ $\mathcal{F}$ *g*
∨ *set*(*tameConf*$_3$ *b c d a e*) ⊆$_{\cong}$ *vertices* ‘ $\mathcal{F}$ *g*
∨ *set*(*tameConf*$_3$ *c d a b e*) ⊆$_{\cong}$ *vertices* ‘ $\mathcal{F}$ *g*

$\vee$ *set*(*tameConf*$_3$ *d a b c e*) $\subseteq_{\cong}$ *vertices* ‘ $\mathcal{F}$ *g*
$\vee$ *set*(*tameConf*$_4$ *a b c d e*) $\subseteq_{\cong}$ *vertices* ‘ $\mathcal{F}$ *g*)

constdefs *tame*$_3$ :: *graph* $\Rightarrow$ *bool*
tame$_3$ *g* $\equiv \forall$ *a b c d*. *is-cycle g* [*a*,*b*,*c*,*d*] $\wedge$ *distinct*[*a*,*b*,*c*,*d*] $\longrightarrow$
tame-quad g a b c d $\vee$ *tame-quad g d c b a*

4. The degree of every vertex is at least 2 and at most 6:

constdefs *tame*$_{45}$ :: *graph* $\Rightarrow$ *bool*
tame$_{45}$ *g* $\equiv \forall$ *v* $\in \mathcal{V}$ *g*. *degree g v* $\leq$ (*if except g v* = *0 then 6 else 5*)

6. The following inequality holds:

constdefs *tame*$_6$ :: *graph* $\Rightarrow$ *bool*
tame$_6$ *g* $\equiv$ *8000* $\leq \sum_{f \in faces\ g}$ c |*vertices f*|

This property implies that there are at least 8 triangles in a tame graph.

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

constdefs *tame*$_7$ :: *graph* $\Rightarrow$ *bool*
tame$_7$ *g* $\equiv \exists$ *w*. *admissible w g* $\wedge \sum_{f \in faces\ g}$ *w f* $<$ *squanderTarget*

8. We formalize the additional restriction (compared with the original definition) that tame graphs do not contain two adjacent vertices of type $(4, 0)$:

constdefs *type40* :: *graph* $\Rightarrow$ *vertex* $\Rightarrow$ *bool*
type40 g v $\equiv$ *tri g v* = *4* $\wedge$ *quad g v* = *0* $\wedge$ *except g v* = *0*

constdefs *tame*$_8$ :: *graph* $\Rightarrow$ *bool*
tame$_8$ *g* $\equiv \neg(\exists$ *v*$\in \mathcal{V}$ *g*. *type40 g v* $\wedge$ ($\exists$ *w*$\in$ *set*(*neighbors g v*). *type40 g w*))

Finally we define the notion of tameness.

constdefs *tame* :: *graph* $\Rightarrow$ *bool*
tame g $\equiv$
tame$_1$ *g* $\wedge$ *tame*$_2$ *g* $\wedge$ *tame*$_3$ *g* $\wedge$ *tame*$_{45}$ *g* (*$\wedge$ *tame*$_5$ *g*) $\wedge$ *tame*$_6$ *g* $\wedge$ *tame*$_7$ *g*
$\wedge$ *tame*$_8$ *g*

theory *Plane1Props*
imports *Plane1 PlaneProps Tame*
begin

lemma *next-plane-subset*:
$\forall f \in \mathcal{F}$ *g*. *vertices f* $\neq$ [] $\Longrightarrow$
set (*next-plane*$_p$ *g*) $\subseteq$ *set* (*next-plane0*$_p$ *g*)
⟨*proof*⟩

lemma *mgp-next-plane0-if-next-plane*:
minGraphProps g $\Longrightarrow$ *g* $[$*next-plane*$_p]\rightarrow$ *g'* $\Longrightarrow$ *g* $[$*next-plane0*$_p]\rightarrow$ *g'*
⟨*proof*⟩

lemma *inv-inv-next-plane*: *invariant inv next-plane*$_p$
⟨*proof*⟩

end

17 Enumeration of Tame Plane Graphs

theory *Generator*
imports *Vector Plane1 Tame*
begin

constdefs
faceSquanderLowerBound :: *graph* ⇒ *nat*
faceSquanderLowerBound g ≡ $\sum_{f \in \mathit{finals}\ g}$ d |*vertices f*|

constdefs
d3-const :: *nat*
d3-const == d *3*
d4-const :: *nat*
d4-const == d *4*

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

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

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

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

Implementation:

lemma [*code*]:
ExcessTable g =
filtermap (λ*v*. *let e* = *ExcessAt g v in if* 0 < *e then Some* (*v*, *e*) *else None*)
⟨*proof*⟩

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

18 Tame Properties

theory *TameProps*
imports *Tame RTranCl*
begin

lemma *length-disj-filter-le*: ∀ *x* ∈ *set xs*. ¬(*P x* ∧ *Q x*) ⟹
length(*filter P xs*) + *length*(*filter Q xs*) ≤ *length xs*
⟨*proof*⟩

lemma *tri-quad-le-degree*: *tri g v* + *quad g v* ≤ *degree g v*
⟨*proof*⟩

lemma *faceCountMax-bound*:
⟦ *tame g*; *v* ∈ $\mathcal{V}$ *g* ⟧ ⟹ *tri g v* + *quad g v* ≤ 6
⟨*proof*⟩

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'*
⟨*proof*⟩

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

lemma *filterout-untame-succs*:

assumes *invP*: *invariant P f* **and** *invPU*: *invariant* (%*g*. *P g* ∧ *U g*) *f*
and *untame*: *untame*(%*g*. *P g* ∧ *U g*)
and *new-untame*: !!*g g'*. ⟦ *P g*; *g'* ∈ *set*(*f g*); *g'* ∉ *set*(*f' g*) ⟧ ⟹ *U g'*
and *gg'*: *g* [*f*]→∗ *g'*
shows *P g* ⟹ *final g'* ⟹ *tame g'* ⟹ *g* [*f'*]→∗ *g'*
⟨*proof*⟩

lemma *comp-map-tame-pres*:
assumes *invP*: *invariant P succs*
and *invPU*: *invariant* (%*g*. *P g* ∧ *U g*) *succs* **and** *untame*: *untame U*
and *f-fin*: !!*g*. *final g* ⟹ *f g* = *g*
and *invPUf*: *invariant* (%*g*. *P g* ∧ *U g*) (%*g*. [*f g*])
and *succs-f*: !!*g*$_0$ *g g'*. *P g*$_0$ ⟹ *g* ∈ *set*(*succs g*$_0$) ⟹ *g'* ∈ *set*(*succs g*) ⟹
U g' ∨ *f g* = *g* ∨ *f g'* = *f g*
and *gg'*: *g* [*succs*]→∗ *g'*
shows *P g* ⟹ *final g'* ⟹ *tame g'* ⟹ *g* [*map f o succs*]→∗ *g'*
⟨*proof*⟩

end

19 All About Finalizing Triangles

theory *Plane3Props*
imports *Plane1Props Generator TameProps*
begin

19.1 Correctness

lemma *decomp-nonFinal3*:
assumes *mgp*: *minGraphProps g*
and *ffs*: *f*#*fs* = [*f* ∈ *faces g*. ¬ *final f* ∧ *triangle f*]
shows *f* = *minimalFace*(*nonFinals g*) &
fs = [*f* ∈ *faces*(*makeFaceFinal* (*minimalFace*(*nonFinals g*)) *g*).
¬ *final f* ∧ *triangle f*] (**is** *?A* & *?B*)
⟨*proof*⟩

lemma *noDupEdge3*:
⟦ *minGraphProps g*; *f* ∈ ℱ *g*; *triangle f*; *v* ∈ 𝒱 *f* ⟧
⟹ ¬ *containsDuplicateEdge g f v* [*0..<3*]
⟨*proof*⟩

lemma *indexToVs3*:
⟦ *triangle f*; *distinct*(*vertices f*); *v* ∈ 𝒱 *f* ⟧
⟹ *indexToVertexList f v* [*0..<3*] = [*Some v*, *Some*(*f* · *v*), *Some*(*f* · (*f* · *v*))]
⟨*proof*⟩

lemma *upt3-in-enumerator*: $[0..<3] \in set\ (enumerator\ 3\ 3)$
⟨*proof*⟩

lemma *mkFaceFin3-in-succs1*:
assumes *mgp*: *minGraphProps g*
and *ffs*: $f\#fs = [f \in faces\ g.\ \neg\ final\ f\ \wedge\ triangle\ f]$
shows *Graph* (*makeFaceFinalFaceList f* (*faces g*)) (*countVertices g*)
(*map* (*makeFaceFinalFaceList f*) (*faceListAt g*)) (*heights g*)
$\in set\ (next\text{-}plane_p\ g)$ (**is** *?g'* ∈ -)
⟨*proof*⟩

lemma *mkFaceFin3-in-rtrancl*:
$minGraphProps\ g \Longrightarrow f\#fs = [f \in faces\ g\ .\ \neg\ final\ f\ \wedge\ triangle\ f] \Longrightarrow$
$g\ [next\text{-}plane_p]{\rightarrow}*\ makeFaceFinal\ f\ g$
⟨*proof*⟩

lemma *mk3Fin-lem*:
$\bigwedge g.\ minGraphProps\ g \Longrightarrow fs = [f \in faces\ g.\ \neg\ final\ f\ \wedge\ triangle\ f] \Longrightarrow$
$g\ [next\text{-}plane_p]{\rightarrow}*\ foldl\ (\%g\ f.\ makeFaceFinal\ f\ g)\ g\ fs$
⟨*proof*⟩

lemma *mk3Fin-in-RTranCl*:
$inv\ g \Longrightarrow g\ [next\text{-}plane_p]{\rightarrow}*\ makeTrianglesFinal\ g$
⟨*proof*⟩

19.2 Completeness

constdefs
in2finals g a b ≡
$\exists f \in set(finals\ g).\ \exists f' \in set(finals\ g).\ (a,b) \in \mathcal{E}\ f\ \wedge\ (b,a) \in \mathcal{E}\ f'$

$untame_2$ :: *graph* ⇒ *bool*
$untame_2\ g \equiv \exists\ a\ b\ c.$ *is-cycle g* $[a,b,c]\ \wedge$ *distinct*$[a,b,c]\ \wedge$
$(\forall f \in \mathcal{F}\ g.\ \mathcal{E}\ f \neq \{(c,a), (a,b), (b,c)\}\ \wedge$
$\mathcal{E}\ f \neq \{(a,c), (c,b), (b,a)\})\ \wedge$
in2finals g a b

lemma *untame2*: $untame(untame_2)$
⟨*proof*⟩

lemma *mk3Fin-id*: $final\ g \Longrightarrow makeTrianglesFinal\ g = g$
⟨*proof*⟩

lemma *inv-untame2*:
$invariant\ (\lambda g.\ inv\ g\ \wedge\ untame_2\ g)\ next\text{-}plane_p$

⟨*proof*⟩

lemma *mk3Fin-id2*:
assumes *mgp*: *minGraphProps g* **and** *nf*: *nonFinals g* ≠ []
and *n3*: ¬ *triangle*(*minimalFace*(*nonFinals g*))
shows *makeTrianglesFinal g* = *g*
⟨*proof*⟩

lemma *mk3Fin-mkFaceFin*:
assumes *mgp*: *minGraphProps g* **and** *nf*: *nonFinals g* ≠ []
and *3*: *triangle*(*minimalFace*(*nonFinals g*))
shows *makeTrianglesFinal* (*makeFaceFinal* (*minimalFace* (*nonFinals g*)) *g*) =
makeTrianglesFinal g
⟨*proof*⟩

lemma *next-plane-mk3Fin-alternatives*:
assumes *inv*: *inv g* **and** *2*: |*faces g*| ≠ *2* **and** *1*: *g* [*next-plane*$_p$]→ *g*′
shows *untame*$_2$ *g*′ ∨ *makeTrianglesFinal g* = *g* ∨
makeTrianglesFinal g′ = *makeTrianglesFinal g*
⟨*proof*⟩

theorem *make3Fin-complete*:
⟦ *invariant inv* (*succ p*);
⋀*g*. *inv g* ⟹ *set* (*succ p g*) ⊆ *set* (*next-plane p g*);
tame g; *final g*; *Seed*$_p$ [*succ p*]→∗ *g* ⟧ ⟹
Seed$_p$ [*map makeTrianglesFinal o succ p*]→∗ *g*
⟨*proof*⟩

end

20 Checking Final Quadrilaterals

theory *Plane4*
imports *While-Combinator Tame*
begin

constdefs
norm-subset :: *vertex list list* ⇒ *vertex list list* ⇒ *bool*
norm-subset xs ys ≡ *let zs* = *map rotate-min ys*
in ∀ *x* ∈ *set xs*. *rotate-min x* ∈ *set zs*

consts *remrevdups* :: *'a list list* ⇒ *'a list list*

primrec
remrevdups [] = []
remrevdups (*xs*#*xss*) = (*if xs mem xss* ∨ *rev xs mem xss then remrevdups xss*
else xs # *remrevdups xss*)

constdefs
find-cycles1 :: *nat* ⇒ *graph* ⇒ *vertex* ⇒ *vertex list list*
find-cycles1 n g v ≡
snd(*snd*(
while (%(*vs*,*wss*,*res*). *vs* ≠ [])
(%(*vs*,*wss*,*res*).
let ws = *hd wss in*
if ws = [] *then* (*tl vs*, *tl wss*, *res*)
else if length vs = *n*
then (*tl vs*, *tl wss*, *if last vs* ∈ *set ws then vs*#*res else res*)
else let vs′ = (*if length vs* + 1 = *n then butlast vs else vs*);
xs = *filter* (%*x*. *x* ∉ *set vs*′) (*neighbors g* (*hd ws*)) *in*
(*hd ws* # *vs*, *xs* # *tl ws* # *tl wss*, *res*))
([*v*], [*neighbors g v*], [])
))

find-cycles :: *nat* ⇒ *graph* ⇒ *vertex list list*
find-cycles n g ≡
remrevdups(*map rotate-min* (*foldr* (%*v vss*. *find-cycles1 n g v* @ *vss*) (*vertices g*)
[]))

constdefs

ok42 :: *vertex list* ⇒ *vertex list list* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex* ⇒ *bool*
ok42 vs fs a b c d ==
norm-subset (*tameConf*$_1$ *a b c d*) *fs* ∨
norm-subset (*tameConf*$_2$ *a b c d*) *fs* ∨
norm-subset (*tameConf*$_2$ *b c d a*) *fs* ∨
(*EX e*:*set vs*. *e* ∉ *set*[*a*,*b*,*c*,*d*] ∧
(*norm-subset* (*tameConf*$_3$ *a b c d e*) *fs* ∨
norm-subset (*tameConf*$_3$ *b c d a e*) *fs* ∨
norm-subset (*tameConf*$_3$ *c d a b e*) *fs* ∨
norm-subset (*tameConf*$_3$ *d a b c e*) *fs* ∨
norm-subset (*tameConf*$_4$ *a b c d e*) *fs*)
)

ok4 :: *graph* ⇒ *vertex list* ⇒ *bool*
ok4 g vs ≡
let fs = *map vertices* (*faces g*); *gvs* = *vertices g*;
a = *hd vs*; *b* = *hd*(*tl vs*); *c* = *hd*(*tl*(*tl vs*)); *d* = *hd*(*tl*(*tl*(*tl vs*)))
in ok42 gvs fs a b c d ∨ *ok42 gvs fs d c b a*

$is\text{-}tame_3 :: graph \Rightarrow bool$
$is\text{-}tame_3\ g \equiv \forall vs \in set(find\text{-}cycles\ 4\ g).$
$\quad is\text{-}cycle\ g\ vs \wedge distinct\ vs \wedge |vs| = 4 \longrightarrow ok4\ g\ vs$

end

21 Neglectable Final Graphs

theory *TameEnum*
imports *Generator Plane4*
begin

constdefs
$is\text{-}tame :: graph \Rightarrow bool$
$is\text{-}tame\ g \equiv tame_{45}\ g \wedge tame_6\ g \wedge tame_8\ g \wedge is\text{-}tame_7\ g \wedge is\text{-}tame_3\ g$

constdefs
$next\text{-}tame :: nat \Rightarrow graph \Rightarrow graph\ list \quad (next'\text{-}tame\_)$
$next\text{-}tame_p \equiv filter\ (\lambda g.\ \neg\ final\ g \vee is\text{-}tame\ g)\ o\ next\text{-}tame1_p$

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

$TameEnum :: graph\ set$
$TameEnum \equiv \bigcup p{\leq}5.\ TameEnum_p$

end

22 Properties of Lower Bound Machinery

theory *ScoreProps*
imports *ListSum TameEnum PlaneProps TameProps*
begin

lemma *deleteAround-empty*[*simp*]: $deleteAround\ g\ a\ [] = []$
$\langle proof \rangle$

lemma *deleteAroundCons*:
$deleteAround\ g\ a\ (p\#ps) =$
$\quad (if\ fst\ p \in \{v.\ \exists f \in set\ (facesAt\ g\ a).$
$\qquad\qquad length\ (vertices\ f) = 4$
$\qquad\qquad \wedge\ v \in \{f \cdot a, f \cdot (f \cdot a)\}$
$\qquad\qquad \vee\ length\ (vertices\ f) \neq 4 \wedge v = f \cdot a\}$
$\quad\ then\ deleteAround\ g\ a\ ps$

else p#deleteAround g a ps)
⟨*proof*⟩

lemma *deleteAround-subset*: *set* (*deleteAround g a ps*) ⊆ *set ps*
⟨*proof*⟩

lemma *distinct-deleteAround*: *distinct* (*map fst ps*) ⟹
distinct (*map fst* (*deleteAround g* (*fst* (*a*, *b*)) *ps*))
⟨*proof*⟩

constdefs
deleteAround′ :: *graph* ⇒ *vertex* ⇒ (*vertex* × *nat*) *list* ⇒
(*vertex* × *nat*) *list*
deleteAround′ g v ps ≡
let fs = *facesAt g v*;
vs = (λ*f*. *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*
⟨*proof*⟩

lemma *deleteAround-nextVertex*:
f ∈ *set* (*facesAt g a*) ⟹
(*f* · *a*, *b*) ∉ *set* (*deleteAround g a ps*)
⟨*proof*⟩

lemma *deleteAround-nextVertex-nextVertex*:
f ∈ *set* (*facesAt g a*) ⟹ |*vertices f*| = 4 ⟹
(*f* · (*f* · *a*), *b*) ∉ *set* (*deleteAround g a ps*)
⟨*proof*⟩

lemma *deleteAround-prevVertex*:
minGraphProps g ⟹ *f* ∈ *set* (*facesAt g a*) ⟹
(f^{-1} · *a*, *b*) ∉ *set* (*deleteAround g a ps*)
⟨*proof*⟩

lemma *deleteAround-separated*:
minGraphProps g ⟹ *final g* ⟹ |*vertices f*| ≤ 4 ⟹ *f* ∈ *set*(*facesAt g a*) ⟹
$\mathcal{V}$ *f* ∩ *set* [*fst p*. *p* ∈ *deleteAround g a ps*] ⊆ {*a*}
(**concl is** *?A*)
⟨*proof*⟩

lemma [*iff*]: *preSeparated g* {}
⟨*proof*⟩

lemma *preSeparated-insert*:
assumes *mgp*: *minGraphProps g* **and** *a*: $a \in \mathcal{V}\ g$
and *ps*: *preSeparated g V*
and *s2*: $(\bigwedge f.\ f \in set\ (facesAt\ g\ a) \Longrightarrow f \cdot a \notin V)$
and *s3*: $(\bigwedge f.\ f \in set\ (facesAt\ g\ a) \Longrightarrow$
$|vertices\ f| \leq 4 \Longrightarrow \mathcal{V}\ f \cap V \subseteq \{a\})$
shows *preSeparated g* (*insert a V*)
⟨*proof*⟩

consts *ExcessNotAtRecList* :: (*vertex*, *nat*) *table* ⇒ *graph* ⇒ *vertex list*
recdef *ExcessNotAtRecList measure* (λ*ps*. *size ps*)
ExcessNotAtRecList [] = (%*g*. [])
ExcessNotAtRecList (*p*#*ps*) = (%*g*.
let l1 = *ExcessNotAtRecList ps g*;
l2 = *ExcessNotAtRecList* (*deleteAround g* (*fst p*) *ps*) *g in*
if ExcessNotAtRec ps g
≤ *snd p* + *ExcessNotAtRec* (*deleteAround g* (*fst p*) *ps*) *g*
then fst p#*l2 else l1*)
(**hints** *recdef-simp*: *less-Suc-eq-le length-deleteAround*)

lemma *isTable-deleteAround*:
isTable E vs ((*a*,*b*)#*ps*) ⟹ *isTable E vs* (*deleteAround g a ps*)
⟨*proof*⟩

lemma *ListSum-ExcessNotAtRecList*:
isTable E vs ps ⟹ *ExcessNotAtRec ps g*
= $\sum p \in$ *ExcessNotAtRecList ps g E p* (**is** *?T ps* ⟹ *?P ps*)
⟨*proof*⟩

lemma *ExcessNotAtRecList-subset*:
set (*ExcessNotAtRecList ps g*) ⊆ *set* [*fst p*. *p* ∈ *ps*] (**is** *?P ps*)
⟨*proof*⟩

lemma *preSeparated-ExcessNotAtRecList*:
minGraphProps g ⟹ *final g* ⟹ *isTable E* (*vertices g*) *ps* ⟹
preSeparated g (*set* (*ExcessNotAtRecList ps g*))
⟨*proof*⟩

lemma *isTable-ExcessTable*:

isTable (λ*v*. *ExcessAt g v*) *vs* (*ExcessTable g vs*)
⟨*proof*⟩

lemma *ExcessTable-subset*:
set (*map fst* (*ExcessTable g vs*)) ⊆ *set vs*
⟨*proof*⟩

lemma *distinct-ExcessNotAtRecList*:
distinct (*map fst ps*) ⟹ *distinct* (*ExcessNotAtRecList ps g*)
(**is** *?T ps* ⟹ *?P ps*)
⟨*proof*⟩

consts *ExcessTable-cont* ::
(*vertex* ⇒ *nat*) ⇒ *vertex list* ⇒ (*vertex* × *nat*) *list*
primrec
ExcessTable-cont ExcessAtPG [] = []
ExcessTable-cont ExcessAtPG (*v*#*vs*) =
(*let vi* = *ExcessAtPG v in*
if 0 < *vi*
then (*v*, *vi*)#*ExcessTable-cont ExcessAtPG vs*
else ExcessTable-cont ExcessAtPG vs)

constdefs
ExcessTable′ :: *graph* ⇒ *vertex list* ⇒ (*vertex* × *nat*) *list*
ExcessTable′ g ≡ *ExcessTable-cont* (*ExcessAt g*)

lemma *distinct-ExcessTable-cont*:
distinct vs ⟹
distinct (*map fst* (*ExcessTable-cont* (*ExcessAt g*) *vs*))
⟨*proof*⟩

lemma *ExcessTable-cont-eq*:
ExcessTable-cont E vs =
[(*v*, *E v*). *v* ∈ [*v*∈*vs* . 0 < *E v*]]
⟨*proof*⟩

lemma *ExcessTable-eq*: *ExcessTable* = *ExcessTable′*
⟨*proof*⟩

lemma *distinct-ExcessTable*:
distinct vs ⟹ *distinct* [*fst p*. *p* ∈ *ExcessTable g vs*]
⟨*proof*⟩

lemma *ExcessNotAt-eq*:

minGraphProps g $\Longrightarrow$ *final g* $\Longrightarrow$
$\exists$ *V*. *ExcessNotAt g None*
$= \sum v \in V$ *ExcessAt g v*
$\wedge$ *preSeparated g* (*set V*) $\wedge$ *set V* $\subseteq$ *set* (*vertices g*)
$\wedge$ *distinct V*
⟨*proof*⟩

lemma *excess-eq*:
assumes *6*: $t + q \leq 6$
shows *excessAtType t q 0* $+ t *$ d *3* $+ q *$ d *4* $=$ b *t q*
⟨*proof*⟩

lemma *excess-eq1*:
⟦ *inv g*; *final g*; *tame g*; *except g v* $= 0$; $v \in$ *set*(*vertices g*) ⟧ $\Longrightarrow$
ExcessAt g v + (*tri g v*) $*$ d *3* + (*quad g v*) $*$ d *4*
$=$ b (*tri g v*) (*quad g v*)
⟨*proof*⟩

preSeparating

lemma *preSeparated-separating*:
assumes *pl*: *inv g* **and** *fin*: *final g* **and** *ne*: *noExceptionals g* (*set V*)
and *pS*: *preSeparated g* (*set V*)
shows *separating* (*set V*) (λv. *set* (*facesAt g v*))
⟨*proof*⟩

lemma *preSeparated-disj-Union2*:
assumes *pl*: *inv g* **and** *fin*: *final g* **and** *ne*: *noExceptionals g* (*set V*)
and *pS*: *preSeparated g* (*set V*) **and** *dist*: *distinct V*
and *V-subset*: *set V* $\subseteq$ *set* (*vertices g*)
shows ($\sum v \in V \sum f \in$ *facesAt g v* (*w*::*face* $\Rightarrow$ *nat*) *f*)
$= \sum f \in$ [*f*$\in$*faces g* . $\exists v \in$ *set V*. *f* $\in$ *set* (*facesAt g v*)] *w f*
⟨*proof*⟩

lemma *squanderFace-distr2*: *inv g* $\Longrightarrow$ *final g* $\Longrightarrow$ *noExceptionals g* (*set V*) $\Longrightarrow$
preSeparated g (*set V*) $\Longrightarrow$ *distinct V* $\Longrightarrow$ *set V* $\subseteq$ *set* (*vertices g*) $\Longrightarrow$
$\sum f \in$ [*f*$\in$*faces g*. $\exists v \in$ *set V*. *f* $\in$ *set* (*facesAt g v*)]
d |*vertices f*|
$= \sum v \in V$ ((*tri g v*) $*$ d *3*
+ (*quad g v*) $*$ d *4*)
⟨*proof*⟩

lemma *preSeparated-subset*:
V1 $\subseteq$ *V2* $\Longrightarrow$ *preSeparated g V2* $\Longrightarrow$ *preSeparated g V1*
⟨*proof*⟩

end

23 Correctness of Lower Bound for Final Graphs

theory *LowerBound*
imports *PlaneProps ScoreProps*
begin
⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

theorem *total-weight-lowerbound*:
$inv\ g \Longrightarrow final\ g \Longrightarrow tame\ g \Longrightarrow admissible\ w\ g \Longrightarrow$
$\sum_{f\ \in\ faces\ g}\ w\ f < squanderTarget \Longrightarrow$
$squanderLowerBound\ g \leq \sum_{f\ \in\ faces\ g}\ w\ f$
⟨*proof*⟩

24 Properties of Tame Graph Enumeration (1)

theory *GeneratorProps*
imports *Plane3Props LowerBound*
begin

lemma *genPolyTame-spec*:
$generatePolygonTame\ n\ v\ f\ g = [g' \in generatePolygon\ n\ v\ f\ g\ .\ \neg\ notame\ g']$
⟨*proof*⟩

lemma *genPolyTame-subset-genPoly*:
$g' \in set(generatePolygonTame\ i\ v\ f\ g) \Longrightarrow$
$g' \in set(generatePolygon\ i\ v\ f\ g)$
⟨*proof*⟩

lemma *next-tame0-subset-plane*:
$set(next\text{-}tame0\ p\ g) \subseteq set(next\text{-}plane\ p\ g)$
⟨*proof*⟩

lemma *genPoly-new-face*:
$[\![g' \in set\ (generatePolygon\ n\ v\ f\ g);\ minGraphProps\ g;\ f \in set\ (nonFinals\ g);$
$v \in \mathcal{V}\ f;\ n \geq 3\ ]\!] \Longrightarrow$
$\exists f \in set(finals\ g') - set(finals\ g).\ |vertices\ f| = n$
⟨*proof*⟩

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 $faceSquanderLowerBound\ g' \geq faceSquanderLowerBound\ g + \mathrm{d}\ n$
⟨*proof*⟩

lemma *ExcessTable-empty*:
$\forall x \in \mathcal{V}\ g.\ \neg\ finalVertex\ g\ x \Longrightarrow ExcessTable\ g\ (vertices\ g) = []$
⟨*proof*⟩

constdefs
$close :: graph \Rightarrow vertex \Rightarrow vertex \Rightarrow bool$
$close\ g\ u\ v \equiv$
$\exists f \in set(facesAt\ g\ u).\ if\ |vertices\ f| = 4\ then\ v = f \cdot u \vee v = f \cdot (f \cdot u)$
$else\ v = f \cdot u$

lemma *delAround-def*: $deleteAround\ g\ u\ ps = [p \in ps.\ \neg\ close\ g\ u\ (fst\ p)]$
⟨*proof*⟩

lemma *close-sym*: **assumes** *mgp*: *minGraphProps g* **and** *cl*: *close g u v*
shows *close g v u*
⟨*proof*⟩

lemma *preSep-conv*:
assumes *mgp*: *minGraphProps g*
shows $preSeparated\ g\ V = (\forall u{\in}V.\forall v{\in}V.\ u \neq v \longrightarrow \neg\ close\ g\ u\ v)$ (**is** *?P* = *?Q*)
⟨*proof*⟩

lemma *fin-aux*: $finite\ B \Longrightarrow finite\{f\ A|A.\ A \subseteq B \wedge P\ A\}$
⟨*proof*⟩

lemma *preSep-ne*: $\exists P \subseteq M.\ preSeparated\ g\ (fst\ `\ P)$
⟨*proof*⟩

lemma *ExcessNotAtRec-conv-Max*:
assumes *mgp*: *minGraphProps g*
shows $distinct(map\ fst\ ps) \Longrightarrow ExcessNotAtRec\ ps\ g =$
$Max\{\ \sum p{\in}P.\ snd\ p\ |P.\ P \subseteq set\ ps \wedge preSeparated\ g\ (fst\ `\ P)\}$
(**concl is** - = *Max*(*?M ps*) **is** - = *Max*{- |*P*. *?S ps P*})
⟨*proof*⟩

lemma *dist-ExcessTab*: $distinct\ (map\ fst\ (ExcessTable\ g\ (vertices\ g)))$

⟨*proof*⟩

lemma *mono-ExcessTab*: ⟦$g' \in$ *set* (*next-plane0*$_p$ g); *inv* g ⟧ ⟹
set(*ExcessTable* g (*vertices* g)) ⊆ *set*(*ExcessTable* g' (*vertices* g'))
⟨*proof*⟩

lemma *close-antimono*:
⟦$g' \in$ *set* (*next-plane0*$_p$ g); *inv* g; $u \in \mathcal{V}$ g; *finalVertex* g u ⟧ ⟹
close g' u v ⟹ *close* g u v
⟨*proof*⟩

lemma *ExcessTab-final*:
$p \in$ *set*(*ExcessTable* g (*vertices* g)) ⟹ *finalVertex* g (*fst* p)
⟨*proof*⟩

lemma *ExcessTab-vertex*:
$p \in$ *set*(*ExcessTable* g (*vertices* g)) ⟹ *fst* $p \in \mathcal{V}$ g
⟨*proof*⟩

lemma *next-plane0-incr-ExcessNotAt*:
⟦$g' \in$ *set* (*next-plane0*$_p$ g); *inv* g ⟧ ⟹
ExcessNotAt g *None* ≤ *ExcessNotAt* g' *None*
⟨*proof*⟩

lemma *next-plane0-incr-squander-lb*:
⟦$g' \in$ *set* (*next-plane0*$_p$ g); *inv* g ⟧ ⟹
squanderLowerBound g ≤ *squanderLowerBound* g'
⟨*proof*⟩

lemma *inv-notame*:
⟦$g' \in$ *set* (*next-plane0*$_p$ g); *inv* g; *notame* g⟧
⟹ *notame* g'
⟨*proof*⟩

lemma *inv-inv-notame*:
invariant(λg. *inv* g ∧ *notame* g) *next-plane*$_p$
⟨*proof*⟩

lemma *untame-notame*:
untame (λg. *inv* g ∧ *notame* g)
⟨*proof*⟩

lemma *polysizes-tame*:
⟦ $g' \in set\ (generatePolygon\ n\ v\ f\ g)$; $inv\ g$; $f \in set(nonFinals\ g)$;
$v \in \mathcal{V}\ f$; $3 \leq n$; $n < 4+p$; $n \notin set(polysizes\ p\ g)$ ⟧
$\Longrightarrow notame\ g'$
⟨*proof*⟩

lemma *genPolyTame-notame*:
⟦ $g' \in set\ (generatePolygon\ n\ v\ f\ g)$; $g' \notin set\ (generatePolygonTame\ n\ v\ f\ g)$;
$inv\ g$; $3 \leq n$ ⟧
$\Longrightarrow notame\ g'$
⟨*proof*⟩

declare *upt-Suc*[*simp del*]
lemma *excess-notame*:
⟦ $inv\ g$; $g' \in set\ (next\text{-}plane_p\ g)$; $g' \notin set\ (next\text{-}tame0\ p\ g)$ ⟧
$\Longrightarrow notame\ g'$
⟨*proof*⟩
declare *upt-Suc*[*simp*]

lemma *next-tame0-comp*: ⟦ $Seed_p\ [next\text{-}plane\ p]{\rightarrow}*\ g$; $final\ g$; $tame\ g$ ⟧
$\Longrightarrow Seed_p\ [next\text{-}tame0\ p]{\rightarrow}*\ g$
⟨*proof*⟩

lemma *next-tame1-comp*:
⟦ $tame\ g$; $final\ g$; $Seed_p\ [next\text{-}tame0\ p]{\rightarrow}*\ g$ ⟧
$\Longrightarrow Seed_p\ [next\text{-}tame1_p]{\rightarrow}*\ g$
⟨*proof*⟩

lemma *inv-inv-next-tame0*: $invariant\ inv\ (next\text{-}tame0\ p)$
⟨*proof*⟩

lemma *inv-inv-next-tame1*: $invariant\ inv\ next\text{-}tame1_p$
⟨*proof*⟩

lemma *inv-inv-next-tame*: $invariant\ inv\ next\text{-}tame_p$
⟨*proof*⟩

lemma *mgp-TameEnum*: $g \in TameEnum_p \Longrightarrow minGraphProps\ g$
⟨*proof*⟩

end

25 Properties of Tame Graph Enumeration (2)

theory *TameEnumProps*
imports *GeneratorProps*
begin

Completeness of filter for final graphs.

lemma *help*: $(EX\ x.\ (EX\ y{:}A.\ x = f\ y)\ \&\ P\ x) = (EX\ y{:}A.\ P(f\ y))$
⟨*proof*⟩

lemma $tame_3$-*is-tame*$_3$: $tame_3\ g \Longrightarrow$ *is-tame*$_3\ g$
⟨*proof*⟩

lemma *untame-negFin*:
assumes *pl*: *inv g* **and** *fin*: *final g* **and** *tame*: *tame g*
shows *is-tame g*
⟨*proof*⟩

lemma *next-tame-comp*:
$[\![\ tame\ g;\ final\ g;\ Seed_p\ [next\text{-}tame1\ p]{\rightarrow}*\ g\ ]\!]$
$\Longrightarrow Seed_p\ [next\text{-}tame_p]{\rightarrow}*\ g$
⟨*proof*⟩

end

26 Trie (List Version)

theory *TrieList*
imports *Main*
begin

26.1 Association lists

consts
assoc :: $('key * 'val)list \Rightarrow 'key \Rightarrow 'val\ option$
rem-alist :: $'key \Rightarrow ('key * 'val)list \Rightarrow ('key * 'val)list$
upd-alist :: $'key \Rightarrow 'val \Rightarrow ('key * 'val)list \Rightarrow ('key * 'val)list$

primrec
assoc [] *x* = *None*
assoc (*p*#*ps*) *x* = (*let* (*a*,*b*) = *p* *in* *if* *a*=*x* *then* *Some* *b* *else* *assoc* *ps* *x*)
primrec
rem-alist k [] = []
rem-alist k (*p*#*ps*) = (*if* *fst p* = *k* *then* *ps* *else* *p* # *rem-alist k ps*)

primrec
upd-alist k v [] = [(*k*,*v*)]
upd-alist k v (*p*#*ps*) = (*if fst p* = *k then* (*k*,*v*)#*ps else p* # *upd-alist k v ps*)

lemma *assoc-conv*: *assoc al x* = *map-of al x*
⟨*proof*⟩

lemma *map-of-upd-alist*: *map-of*(*upd-alist k v al*) = (*map-of al*)(*k* ↦ *v*)
⟨*proof*⟩

lemma *rem-alist-id*[*simp*]: *k* ∉ *fst* ' *set al* ⟹ *rem-alist k al* = *al*
⟨*proof*⟩

lemma *map-of-rem-distinct-alist*: *distinct*(*map fst al*) ⟹
map-of(*rem-alist k al*) = (*map-of al*)(*k* := *None*)
⟨*proof*⟩

lemma *map-of-rem-alist*[*simp*]:
k' ≠ *k* ⟹ *map-of* (*rem-alist k al*) *k'* = *map-of al k'*
⟨*proof*⟩

26.2 Trie

datatype ('*a*,'*v*)*trie* = *Trie* '*v list* ('*a* ∗ ('*a*,'*v*)*trie*)*list*

consts *values* :: ('*a*,'*v*)*trie* ⇒ '*v list*
alist :: ('*a*,'*v*)*trie* ⇒ ('*a* ∗ ('*a*,'*v*)*trie*)*list*
primrec *values*(*Trie vs al*) = *vs*
primrec *alist*(*Trie vs al*) = *al*

consts
lookup-trie :: ('*a*,'*v*)*trie* ⇒ '*a list* ⇒ '*v list*
update-trie :: ('*a*,'*v*)*trie* ⇒ '*a list* ⇒ '*v list* ⇒ ('*a*,'*v*)*trie*
insert-trie :: ('*a*,'*v*)*trie* ⇒ '*a list* ⇒ '*v* ⇒ ('*a*,'*v*)*trie*

primrec
lookup-trie t [] = *values t*
lookup-trie t (*a*#*as*) = (*case assoc* (*alist t*) *a of*
None ⇒ []
| *Some at* ⇒ *lookup-trie at as*)
primrec
update-trie t [] *vs* = *Trie vs* (*alist t*)
update-trie t (*a*#*as*) *vs* =
(*let tt* = (*case assoc* (*alist t*) *a of*
None ⇒ *Trie* [] [] | *Some at* ⇒ *at*)
in Trie (*values t*) ((*a*,*update-trie tt as vs*) # *rem-alist a* (*alist t*)))

primrec
insert-trie t [] *v* = *Trie* (*v* # *values t*) (*alist t*)

insert-trie t (a#as) vs =
(let tt = (case assoc (alist t) a of
None ⇒ Trie [] [] | Some at ⇒ at)
in Trie (values t) ((a,insert-trie tt as vs) # rem-alist a (alist t)))

lemma *lookup-empty*[*simp*]: *lookup-trie (Trie [] []) as = []*
⟨*proof*⟩

theorem *lookup-update-trie*: !*t v bs.*
lookup-trie (update-trie t as vs) bs = (if as=bs then vs else lookup-trie t bs)
⟨*proof*⟩

theorem *insert-trie-conv*:
!!*t. insert-trie t as v = update-trie t as (v#lookup-trie t as)*
⟨*proof*⟩

constdefs
trie-of-list :: (*'b ⇒ 'a list*) ⇒ *'b list ⇒* (*'a*,*'b*)*trie*
trie-of-list key == foldl (%t v. insert-trie t (key v) v) (Trie [] [])

lemma *in-set-lookup-trie-of-list*:
v ∈ set(*lookup-trie (trie-of-list key vs) (key v)*) = (*v ∈ set vs*)
⟨*proof*⟩

lemma *in-set-lookup-trie-of-listD*:
assumes *v ∈ set*(*lookup-trie (trie-of-list f vs) xs*) **shows** *v ∈ set vs*
⟨*proof*⟩

end

27 Archive

theory *Arch*
imports *Main*
uses (*arch.ML*)
(*Archives/Tri.ML*)
(*Archives/Quad.ML*)
(*Archives/Pent.ML*)
(*Archives/Hex.ML*)
(*Archives/Hept.ML*)
(*Archives/Oct.ML*)
begin

The Archive is contained in 6 ML files.

⟨*ML*⟩
First the ML values are loaded. Then they are turned into Isabelle defini-

tions of the constants *Tri*, *Quad*, *Pent*, *Hex*, *Hept*, *Oct*, all of type *nat list list list*.

end

28 Comparing Enumeration and Achive

theory *ArchComp*
imports *TameEnum TrieList Arch*
begin

consts *qsort* :: ('*a* ⇒ '*a* => *bool*) * '*a list* ⇒ '*a list*
recdef *qsort measure* (*size o snd*)
  qsort(*le*, []) = []
  qsort(*le*, *x*#*xs*) = *qsort*(*le*, [*y*:*xs* . ~ *le x y*]) @ [*x*] @
    qsort(*le*, [*y*:*xs* . *le x y*])
(**hints** *recdef-simp*: *length-filter-le*[*THEN le-less-trans*])

constdefs
 nof-vertices :: '*a fgraph* ⇒ *nat*
nof-vertices == *length o remdups o concat*

 fgraph :: *graph* ⇒ *nat fgraph*
fgraph g == *map vertices* (*faces g*)

 hash :: *nat fgraph* ⇒ *nat list*
hash fs == *let n* = *nof-vertices fs in*
  [*n*, *size fs*] @
  qsort (%*x y*. *y* < *x*, *map* (%*i*. *foldl* (*op* +) *0* (*map size* [*f*:*fs*. *i mem f*]))
    [*0*..<*n*])

consts
 enum-finals :: (*graph* ⇒ *graph list*) ⇒ *nat* ⇒ *graph list* ⇒ *nat fgraph list*
    ⇒ *nat fgraph list option*
primrec
enum-finals succs 0 todo fins = *None*
enum-finals succs (*Suc n*) *todo fins* =
 (*if todo* = [] *then Some fins*
  else let g = *hd todo*; *todo'* = *tl todo in*
   if final g then enum-finals succs n todo' (*fgraph g* # *fins*)
   else enum-finals succs n (*succs g* @ *todo'*) *fins*)

constdefs
 tameEnum :: *nat* ⇒ *nat* ⇒ *nat fgraph list option*
tameEnum p n == *enum-finals* (*next-tame p*) *n* [*Seed p*] []

 diff2 :: *nat fgraph list* ⇒ *nat fgraph list* ⇒

$(nat * nat\ fgraph)\ list * (nat * nat\ fgraph)list$
diff2 fgs ags ==
let xfgs = map (%fs. (nof-vertices fs, fs)) fgs;
xags = map (%fs. (nof-vertices fs, fs)) ags in
(filter (%xfg. ~list-ex (iso-test xfg) xags) xfgs,
filter (%xag. ~list-ex (iso-test xag) xfgs) xags)

same :: nat fgraph list option ⇒ nat fgraph list ⇒ bool
same fgso ags == case fgso of None ⇒ False | Some fgs ⇒
let niso-test = (%fs1 fs2. iso-test (nof-vertices fs1,fs1)
(nof-vertices fs2,fs2));
tfgs = trie-of-list hash fgs;
tags = trie-of-list hash ags in
(list-all (%fg. list-ex (niso-test fg) (lookup-trie tags (hash fg))) fgs &
list-all (%ag. list-ex (niso-test ag) (lookup-trie tfgs (hash ag))) ags)

constdefs
pre-iso-test :: vertex fgraph ⇒ bool
pre-iso-test Fs ≡
$[] \notin$ *set Fs* $\wedge$ $(\forall$ *F*$\in$*set Fs. distinct F*$)$ $\wedge$ *distinct (map rotate-min Fs)*

lemma *pre-iso-test3*: $\forall$ *g* $\in$ *set Tri. pre-iso-test g*
⟨*proof*⟩

lemma *pre-iso-test4*: $\forall$ *g* $\in$ *set Quad. pre-iso-test g*
⟨*proof*⟩

lemma *pre-iso-test5*: $\forall$ *g* $\in$ *set Pent. pre-iso-test g*
⟨*proof*⟩

lemma *pre-iso-test6*: $\forall$ *g* $\in$ *set Hex. pre-iso-test g*
⟨*proof*⟩

lemma *pre-iso-test7*: $\forall$ *g* $\in$ *set Hept. pre-iso-test g*
⟨*proof*⟩

lemma *pre-iso-test8*: $\forall$ *g* $\in$ *set Oct. pre-iso-test g*
⟨*proof*⟩

⟨*ML*⟩

lemma *same3*: *same (tameEnum 0 800000) Tri*
⟨*proof*⟩

lemma *same4*: *same (tameEnum 1 8000000) Quad*
⟨*proof*⟩

lemma *same5*: *same (tameEnum 2 20000000) Pent*

⟨*proof*⟩

lemma *same6*: *same* (*tameEnum 3 4000000*) *Hex*
⟨*proof*⟩

lemma *same7*: *same* (*tameEnum 4 1000000*) *Hept*
⟨*proof*⟩

lemma *same8*: *same* (*tameEnum 5 2000000*) *Oct*
⟨*proof*⟩

⟨*ML*⟩

end

29 Completeness of Archive Test

theory *ArchCompProps*
imports *TameEnumProps ArchComp*
begin

corollary *iso-test-correct*:
⟦ *pre-iso-test* Fs_1; *pre-iso-test* Fs_2 ⟧ ⟹
iso-test (*nof-vertices* Fs_1,Fs_1) (*nof-vertices* Fs_2,Fs_2) = ($Fs_1 \simeq Fs_2$)
⟨*proof*⟩

lemma *same-imp-iso-subset*:
assumes *pre1*: $\bigwedge gs\ g.\ gsopt = Some\ gs \Longrightarrow g \in set\ gs \Longrightarrow pre\text{-}iso\text{-}test\ g$
and *pre2*: $\bigwedge g.\ g \in set\ arch \Longrightarrow pre\text{-}iso\text{-}test\ g$
and *same*: *same gsopt arch*
shows $\exists gs.\ gsopt = Some\ gs \wedge set\ gs \subseteq_{\simeq} set\ arch$
⟨*proof*⟩

lemma *enum-finals-tree*:
$\forall g.\ final\ g \longrightarrow next\ g = [] \Longrightarrow enum\text{-}finals\ next\ n\ todo\ Fs_0 = Some\ Fs \Longrightarrow$
$set\ Fs = set\ Fs_0 \cup fgraph\ `\ \{h.\ \exists g \in set\ todo.\ g\ [next]\!\rightarrow\!* h \wedge final\ h\}$
⟨*proof*⟩

lemma *next-tame-of-final*: $\forall g.\ final\ g \longrightarrow next\text{-}tame_p\ g = []$
⟨*proof*⟩

lemma *tameEnum-TameEnum*: *tameEnum p n* = *Some Fs* ⟹ *set Fs* = *fgraph* ‘ $TameEnum_p$
⟨*proof*⟩

lemma *mgp-pre-iso-test*: *minGraphProps g* $\Longrightarrow$ *pre-iso-test*(*fgraph g*)
⟨*proof*⟩

theorem *combine-evals*:
$\forall g \in$ *set arch*. *pre-iso-test g* $\Longrightarrow$ *same* (*tameEnum p n*) *arch*
$\Longrightarrow$ *fgraph* ‘ $TameEnum_p \subseteq_\simeq$ *set arch*
⟨*proof*⟩

end

30 Combining All Completeness Proofs

theory *Completeness*
imports *ArchCompProps*
begin

constdefs
Archive :: *vertex fgraph set*
Archive ≡
set(*Tri* @ *Quad* @ *Pent* @ *Hex* @ *Hept* @ *Oct*)

theorem *TameEnum-Archive*: *fgraph* ‘ *TameEnum* $\subseteq_\simeq$ *Archive*
⟨*proof*⟩

lemma *TameEnum-comp*:
assumes *pg*: $Seed_p$ $[next\text{-}plane_p]\rightarrow*$ *g* **and** *final g* **and** *tame g*
shows $Seed_p$ $[next\text{-}tame_p]\rightarrow*$ *g*
⟨*proof*⟩

lemma *Seed-max-final-ex*:
$\exists f{\in}set$ (*finals* (*Seed p*)). |*vertices f*| = *maxGon p*
⟨*proof*⟩

lemma *max-face-ex*: **assumes** *a*: $Seed_p$ $[next\text{-}plane0_p]\rightarrow*$ *g*
shows $\exists f \in$ *set* (*finals g*). |*vertices f*| = *maxGon p*
⟨*proof*⟩

lemma *tame5*:
assumes *g*: $Seed_p$ $[next\text{-}plane0_p]\rightarrow*$ *g* **and** *final g* **and** *tame g*
shows $p \leq 5$
⟨*proof*⟩

theorem *completeness*:
assumes $g \in$ *PlaneGraphs* **and** *tame g* **shows** *fgraph g* $\in_\simeq$ *Archive*

⟨*proof*⟩

end