Tame Plane Graphs

Gertrud Bauer and Tobias Nipkow

September 13, 2023

Abstract

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

Contents

1 Basic Functions Old and New **5**
- 1.1 HOL . . . 5
- 1.2 Lists . . . 5
- 1.3 *splitAt* . . . 10
- 1.4 *between* . . . 16
- 1.5 Tables . . . 17

2 Isomorphisms Between Plane Graphs **19**
- 2.1 Equivalence of faces . . . 20
- 2.2 Homomorphism and isomorphism . . . 21
- 2.3 Isomorphism tests . . . 22
- 2.4 Elementhood and containment modulo . . . 28

3 More Rotation **28**

4 Graph **29**
- 4.1 Notation . . . 29
- 4.2 Faces . . . 29
- 4.3 Graphs . . . 31
- 4.4 Operations on graphs . . . 32
- 4.5 Navigation in graphs . . . 33
- 4.6 Code generator setup . . . 33

5 Syntax for operations on immutable arrays **34**
5.1 Tabulation . . . 34
5.2 Access . . . 35

6 Enumerating Patches **35**

7 Subdividing a Face **36**

8 Transitive Closure of Successor List Function **38**

9 Plane Graph Enumeration **39**

10 Properties of Graph Utilities **41**
10.1 *nextElem* . . . 42
10.2 *nextVertex* . . . 43
10.3 $\mathcal{E}$ . . . 43
10.4 Triangles . . . 44
10.5 Quadrilaterals . . . 44
10.6 No loops . . . 45
10.7 *between* . . . 45

11 Properties of Patch Enumeration **46**

12 Properties of Face Division **50**
12.1 Finality . . . 50
12.2 *is-prefix* . . . 51
12.3 *is-sublist* . . . 52
12.4 *is-nextElem* . . . 54
12.5 *nextElem*, *sublist*, *is-nextElem* . . . 54
12.6 *before* . . . 55
12.7 *between* . . . 57
12.8 *split-face* . . . 60
12.9 *verticesFrom* . . . 61
12.10 *splitFace* . . . 63
12.11 *removeNones* . . . 72
12.12 *natToVertexList* . . . 72
12.13 *indexToVertexList* . . . 73
12.14 *pre-subdivFace*(′) . . . 75

13 Invariants of (Plane) Graphs **80**
13.1 Rotation of face into normal form . . . 80
13.2 Minimal (plane) graph properties . . . 80
13.3 *containsDuplicateEdge* . . . 84
13.4 *replacefacesAt* . . . 85
13.5 *normFace* . . . 87

13.6 Invariants of *splitFace* . . . 88
13.7 Invariants of *makeFaceFinal* . . . 92
13.8 Invariants of *subdivFace′* . . . 93
13.9 Invariants of *Seed* . . . 94
13.10 Increasing properties of *subdivFace′* . . . 95
13.11 Main invariant theorems . . . 96

14 Further Plane Graph Properties **97**
14.1 *final* . . . 97
14.2 *degree* . . . 97
14.3 Misc . . . 97
14.4 Increasing final faces . . . 98
14.5 Increasing vertices . . . 98
14.6 Increasing vertex degrees . . . 98
14.7 Increasing *except* . . . 99
14.8 Increasing edges . . . 99
14.9 Increasing final vertices . . . 99
14.10 Preservation of *facesAt* at final vertices . . . 99
14.11 Properties of *subdivFace′* . . . 100

15 Summation Over Lists **101**

16 Tameness **102**
16.1 Constants . . . 102
16.2 Separated sets of vertices . . . 103
16.3 Admissible weight assignments . . . 103
16.4 Tameness . . . 104

17 Enumeration of Tame Plane Graphs **105**

18 Tame Properties **107**

19 Neglectable Final Graphs **108**

20 Properties of Lower Bound Machinery **109**

21 Correctness of Lower Bound for Final Graphs **113**

22 Properties of Tame Graph Enumeration (1) **114**

23 Properties of Tame Graph Enumeration (2) **117**

24 Archive **123**

25 Comparing Enumeration and Archive **123**

26 Completeness of Archive Test **124**

27 Completeness Proofs under hypothetical computations **126**

Bibliography **128**

1 Basic Functions Old and New

theory *ListAux*
imports *Main*
begin

declare *Let-def*[*simp*]

1.1 HOL

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

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

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

lemma [*code*]:
enum 0 = {}
enum (*Suc n*) = *insert n* (*enum n*)
⟨*proof*⟩

1.2 Lists

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

1.2.1 *length*

notation *length* (|-|)

lemma *length3D*: $|xs| = 3 \Longrightarrow \exists\, x\ y\ z.\ xs = [x,\ y,\ z]$
⟨*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* = []) ⟹ $x \in set\ xs \Longrightarrow \neg\ P\ x$
⟨*proof*⟩

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

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

lemma *filter-compl1*:

([x←xs. P x] = []) = ([x←xs. ¬ P x] = xs) (**is** *?lhs = ?rhs*)
⟨*proof*⟩
lemma [*simp*]: *Not* ∘ (*Not* ∘ *P*) = *P*
⟨*proof*⟩

lemma *filter-eqI*:
(⋀*v*. *v* ∈ *set vs* ⟹ *P v* = *Q v*) ⟹ [*v*←*vs* . *P v*] = [*v*←*vs* . *Q v*]
⟨*proof*⟩

lemma *filter-simp*: (⋀*x*. *x* ∈ *set xs* ⟹ *P x*) ⟹ [*x*←*xs*. *P x* ∧ *Q x*] = [*x*←*xs*. *Q x*]
⟨*proof*⟩

lemma *filter-True-eq1*:
(*length* [*y*←*xs*. *P y*] = *length xs*) ⟹ (⋀*y*. *y* ∈ *set xs* ⟹ *P y*)
⟨*proof*⟩

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

1.2.3 *concat*

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

1.2.4 List product

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

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

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

1.2.5 Minimum and maximum

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

lemma *minimal-in-set*[*simp*]: *xs* ≠ [] ⟹ *minimal f xs* : *set xs*
⟨*proof*⟩

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

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

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

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

1.2.6 replace

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

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

lemma *length-mapAt*[*simp*]: ⋀*xs*. *length*(*mapAt vs f xs*) = *length xs*
⟨*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* ⟹ *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 *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 *distinct-replace*:
distinct fs ⟹ distinct newFs ⟹ set fs ∩ set newFs ⊆ {oldF} ⟹
distinct (replace oldF newFs fs)
⟨*proof*⟩

lemma *replace-replace*[*simp*]: *oldf ∉ set newfs ⟹ distinct xs ⟹*
replace oldf newfs (replace oldf newfs xs) = replace oldf newfs xs
⟨*proof*⟩

lemma *replace-distinct*: *distinct fs ⟹ distinct newfs ⟹ oldf ∈ set fs ⟶ set newfs ∩ set fs ⊆ {oldf} ⟹*
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*⟩

1.2.7 *distinct*

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*⟩

1.2.8 Misc

lemma *drop-last-in*: $\bigwedge$*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*⟩

1.2.9 *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* ⟹ *m* > *0* ⟹
rotate n ls = *rotate m ms* ⟹ *rotate* (*n* − *1*) *ls* = *rotate* (*m* − *1*) *ms*
⟨*proof*⟩

lemma *rotate-minus1'*: *n* > *0* ⟹ *rotate n ls* = *ms* ⟹
rotate (*n* − *1*) *ls* = *rotate* (*length ms* − *1*) *ms*
⟨*proof*⟩

lemma *rotate-inv1*: ⋀ *ms*. *n* < *length ls* ⟹ *rotate n ls* = *ms* ⟹
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* ⟹
ls = *rotate* ((*length ls*) − (*n mod length 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*⟩

1.3 *splitAt*

primrec *splitAtRec* :: *'a* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* × *'a list* **where**
splitAtRec c bs [] = (*bs*,[])

| *splitAtRec c bs (a#as) = (if a = c then (bs, as)*
else splitAtRec c (bs@[a]) as)

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

1.3.1 *splitAtRec*

lemma *splitAtRec-conv*: ⋀*bs.*
splitAtRec x bs xs =
(bs @ takeWhile (λy. y≠x) xs, tl(dropWhile (λy. y≠x) xs))
⟨*proof*⟩

lemma *splitAtRec-distinct-fst*: ⋀ *s. distinct vs ⟹ distinct s ⟹ (set s) ∩ (set vs) = {} ⟹ distinct (fst (splitAtRec ram1 s vs))*
⟨*proof*⟩

lemma *splitAtRec-distinct-snd*: ⋀ *s. distinct vs ⟹ distinct s ⟹ (set s) ∩ (set vs) = {} ⟹ distinct (snd (splitAtRec ram1 s vs))*
⟨*proof*⟩

lemma *splitAtRec-ram*:
⋀ *us a b. ram ∈ set vs ⟹ (a, b) = splitAtRec ram us vs ⟹*
us @ vs = a @ [ram] @ b
⟨*proof*⟩

lemma *splitAtRec-notRam*:
⋀ *us. ram ∉ set vs ⟹ splitAtRec ram us vs = (us @ vs, [])*
⟨*proof*⟩

lemma *splitAtRec-distinct*: ⋀ *s. distinct vs ⟹*
distinct s ⟹ (set s) ∩ (set vs) = {} ⟹
set (fst (splitAtRec ram s vs)) ∩ set (snd (splitAtRec ram s vs)) = {}
⟨*proof*⟩

1.3.2 *splitAt*

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

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

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

lemma *splitAt-ram*:

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

lemma *splitAt-distinct-fst-aux*[*intro*]:
distinct vs ⟹ *distinct* (*fst* (*splitAt ram vs*))
⟨*proof*⟩

lemma *splitAt-distinct-snd-aux*[*intro*]:
distinct vs ⟹ *distinct* (*snd* (*splitAt ram vs*))
⟨*proof*⟩

lemma *splitAt-distinct-ab*:
distinct vs ⟹ (*a*,*b*) = *splitAt ram vs* ⟹ *set a* ∩ *set b* = {}
⟨*proof*⟩

lemma *splitAt-distinct-fst-snd*:
distinct vs ⟹ *set* (*fst* (*splitAt ram vs*)) ∩ *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-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* $\Longrightarrow$ (*set a* $\subseteq$ *set vs*) $\wedge$ (*set b* $\subseteq$ *set vs*)
⟨*proof*⟩

1.3.5 *splitAt* composition

lemma *set-help*: *v* $\in$ *set* (*as* @ *bs*) $\Longrightarrow$ *v* $\in$ *set as* $\vee$ *v* $\in$ *set bs* ⟨*proof*⟩

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

lemma *splitAt-ram3*: *ram2* $\notin$ *set* (*fst* (*splitAt ram1 vs*)) $\Longrightarrow$
ram1 $\in$ *set vs* $\Longrightarrow$ *ram2* $\in$ *set vs* $\Longrightarrow$ *ram1* $\neq$ *ram2* $\Longrightarrow$
ram2 $\in$ *set* (*snd* (*splitAt ram1 vs*)) ⟨*proof*⟩

lemma *splitAt-dist-ram*: *distinct vs* $\Longrightarrow$
vs = *a* @ *ram* # *b* $\Longrightarrow$ (*a*,*b*) = *splitAt ram vs*

⟨*proof*⟩

lemma *distinct-unique1*: *distinct vs* ⟹ *ram* ∈ *set vs* ⟹ ∃!*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* ⟹ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c* ⟹ (*c*, []) = *splitAt ram1* (*snd* (*splitAt ram2 vs*))
⟨*proof*⟩

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

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

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

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

lemma *splitAt-dist-ram-all*:
distinct vs ⟹ *vs* = *a* @ *ram1* # *b* @ *ram2* # *c*
⟹ (*a*, *b*) = *splitAt ram1* (*fst* (*splitAt ram2 vs*))
∧ (*c*, []) = *splitAt ram1* (*snd* (*splitAt ram2 vs*))
∧ (*a*, []) = *splitAt ram2* (*fst* (*splitAt ram1 vs*))
∧ (*b*, *c*) = *splitAt ram2* (*snd* (*splitAt ram1 vs*))
∧ *c* = *snd* (*splitAt ram2 vs*)
∧ *a* = *fst* (*splitAt ram1 vs*)
⟨*proof*⟩

1.3.6 Mixed

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

lemma *snd-splitAt-rev*:
distinct xs $\Longrightarrow$ *x* $\in$ *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 $\leq$ *i* $\Longrightarrow$ *i* < *k* $\Longrightarrow$ *fst*(*splitAt i* [*j*..<*k*]) = [*j*..<*i*]
⟨*proof*⟩

lemma *snd-splitAt-upt*:
j $\leq$ *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 *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* $\Longrightarrow$ *fst* (*splitAt ram* (*a* # *b*)) = *a* # *fst* (*splitAt ram b*) ⟨*proof*⟩
lemma *splitAt-simpB'*[*simp*]: *a* ≠ *ram* $\Longrightarrow$ *fst* (*splitAt ram* (*a* # *b*)) = *a* # *fst* (*splitAt ram b*) ⟨*proof*⟩
lemma *splitAt-simpC*[*simp*]: *ram* $\notin$ *set a* $\Longrightarrow$ *fst* (*splitAt ram* (*a* @ *b*)) = *a* @ *fst* (*splitAt ram b*)
⟨*proof*⟩

lemma *help''''*: $\bigwedge$ *xs ys. snd* (*splitAtRec ram xs b*) = *snd* (*splitAtRec ram ys b*)
⟨*proof*⟩

lemma *splitAt-simpD*[*simp*]: $\bigwedge$ *a. ram* ≠ *a* $\Longrightarrow$ *snd* (*splitAt ram* (*a* # *b*)) = *snd* (*splitAt ram b*) ⟨*proof*⟩
lemma *splitAt-simpD'*[*simp*]: $\bigwedge$ *a. a* ≠ *ram* $\Longrightarrow$ *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* $\notin$ *set a* $\Longrightarrow$ *snd* (*splitAt ram* (*a* @ *b*)) = *snd* (*splitAt ram b*)
⟨*proof*⟩

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

1.4 *between*

definition *between* :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *'a list* **where**
between vs ram$_1$ *ram*$_2$ ≡
let (*pre*$_1$, *post*$_1$) = *splitAt ram*$_1$ *vs in*
if ram$_2$ ∈ *set post*$_1$
then let (*pre*$_2$, *post*$_2$) = *splitAt ram*$_2$ *post*$_1$ *in pre*$_2$
else let (*pre*$_2$, *post*$_2$) = *splitAt ram*$_2$ *pre*$_1$ *in post*$_1$ @ *pre*$_2$

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

lemma *notinset-notinbetween*:
$x \notin set\ xs \Longrightarrow x \notin set(between\ xs\ a\ b)$
⟨*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

type-synonym $('a, 'b)\ table = ('a \times 'b)\ list$

definition $isTable :: ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow ('a, 'b)\ table \Rightarrow bool$ **where**
$isTable\ f\ vs\ t \equiv \forall p.\ p \in set\ t \longrightarrow snd\ p = f\ (fst\ p) \wedge fst\ p \in set\ vs$

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*⟩

definition $removeKey :: 'a \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'b)\ list$ **where**
$removeKey\ a\ ps \equiv [p \leftarrow ps.\ a \neq fst\ p]$

primrec $removeKeyList :: 'a\ list \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'b)\ list$ **where**
$removeKeyList\ []\ ps = ps$
$|\ removeKeyList\ (w\#ws)\ ps = removeKey\ w\ (removeKeyList\ ws\ ps)$

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*) ≤ *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 *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
theory *Quasi-Order*
imports *Main*
begin

locale *quasi-order* =
fixes *qle* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ⪯ *60*)
assumes *qle-refl*[*iff*]: *x* ⪯ *x*
and *qle-trans*: *x* ⪯ *y* ⟹ *y* ⪯ *z* ⟹ *x* ⪯ *z*
begin

definition *in-qle* :: *'a* ⇒ *'a set* ⇒ *bool* (**infix** ∈⪯ *60*) **where**
x ∈⪯ *M* ≡ ∃ *y* ∈ *M*. *x* ⪯ *y*

definition *subseteq-qle* :: *'a set* ⇒ *'a set* ⇒ *bool* (**infix** ⊆⪯ *60*) **where**
M ⊆⪯ *N* ≡ ∀ *x* ∈ *M*. *x* ∈⪯ *N*

definition *seteq-qle* :: *'a set* ⇒ *'a set* ⇒ *bool* (**infix** =⪯ *60*) **where**
M =⪯ *N* ≡ *M* ⊆⪯ *N* ∧ *N* ⊆⪯ *M*

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

lemma *subseteq-qle-refl*[*simp*]: *M* ⊆⪯ *M*
⟨*proof*⟩

lemma *subseteq-qle-trans*: *A* ⊆⪯ *B* ⟹ *B* ⊆⪯ *C* ⟹ *A* ⊆⪯ *C*

⟨*proof*⟩

lemma *empty-subseteq-qle*[*simp*]: $\{\} \subseteq_{\preceq} A$
⟨*proof*⟩

lemma *subseteq-qleI2*: $(\bigwedge x.\ x \in M \Longrightarrow \exists\, y \in N.\ x \preceq y) \Longrightarrow M \subseteq_{\preceq} N$
⟨*proof*⟩

lemma *subseteq-qleD2*: $M \subseteq_{\preceq} N \Longrightarrow x \in M \Longrightarrow \exists\, y \in N.\ x \preceq y$
⟨*proof*⟩

lemma *seteq-qle-refl*[*iff*]: $A =_{\preceq} A$
⟨*proof*⟩

lemma *seteq-qle-trans*: $A =_{\preceq} B \Longrightarrow B =_{\preceq} C \Longrightarrow A =_{\preceq} C$
⟨*proof*⟩

end

end

2 Isomorphisms Between Plane Graphs

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

lemma *image-image-id-if*[*simp*]: $(\bigwedge x.\ f(f\ x) = x) \Longrightarrow f\ `\ f\ `\ M = M$
⟨*proof*⟩

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

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

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

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

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

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

2.1 Equivalence of faces

Two faces are equivalent modulo rotation:

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

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* $(UNIV{::}'a\ list\ set)$ {≅}
⟨*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*:
  $F_1 \cong F_2 \Longrightarrow map\ f\ F_1 \cong map\ f\ F_2$
⟨*proof*⟩

lemma *congs-map-eq-iff*:
 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*]:
  $(rev\ xs \cong rev\ ys) = (xs \cong ys)$
⟨*proof*⟩

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

2.2 Homomorphism and isomorphism

definition *is-pr-Hom* :: $('a \Rightarrow 'b) \Rightarrow 'a\ Fgraph \Rightarrow 'b\ Fgraph \Rightarrow bool$ **where**
$is\text{-}pr\text{-}Hom\ \varphi\ Fs_1\ Fs_2 \equiv (map\ \varphi\ `\ Fs_1)//\{\cong\} = Fs_2\ //\{\cong\}$

definition *is-pr-Iso* :: $('a \Rightarrow 'b) \Rightarrow 'a\ Fgraph \Rightarrow 'b\ Fgraph \Rightarrow bool$ **where**
$is\text{-}pr\text{-}Iso\ \varphi\ Fs_1\ Fs_2 \equiv is\text{-}pr\text{-}Hom\ \varphi\ Fs_1\ Fs_2 \wedge inj\text{-}on\ \varphi\ (\bigcup F \in Fs_1.\ set\ F)$

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

Homomorphisms preserve the set of nodes.

lemma *UN-subset-iff*: $((\bigcup i{\in}I.\ f\ i) \subseteq B) = (\forall i{\in}I.\ f\ i \subseteq B)$
⟨*proof*⟩

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

lemma *pr-Hom-pres-face-nodes*:
$is\text{-}pr\text{-}Hom\ \varphi\ Fs_1\ Fs_2 \Longrightarrow (\bigcup F{\in}Fs_1.\ \{\varphi\ `\ (set\ F)\}) = (\bigcup F{\in}Fs_2.\ \{set\ F\})$
⟨*proof*⟩

lemma *pr-Hom-pres-nodes*:
assumes $is\text{-}pr\text{-}Hom\ \varphi\ Fs_1\ Fs_2$
shows $\varphi\ `\ (\bigcup F{\in}Fs_1.\ set\ F) = (\bigcup F{\in}Fs_2.\ set\ F)$
⟨*proof*⟩

Therefore isomorphisms preserve cardinality of node set.

lemma *pr-Iso-same-no-nodes*:
$[\![\ is\text{-}pr\text{-}Iso\ \varphi\ Fs_1\ Fs_2;\ finite\ Fs_1\ ]\!]$
$\Longrightarrow card(\bigcup F{\in}Fs_1.\ set\ F) = card(\bigcup F{\in}Fs_2.\ set\ F)$
⟨*proof*⟩

lemma *pr-iso-same-no-nodes*:
$is\text{-}pr\text{-}iso\ \varphi\ Fs_1\ Fs_2 \Longrightarrow card(\bigcup F{\in}set\ Fs_1.\ set\ F) = card(\bigcup F{\in}set\ Fs_2.\ set\ F)$
⟨*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*: $inj\text{-}on\ (\lambda xs.\{xs\}//\{\cong\})\ (set\ Fs_1)$
and *inj2*: $inj\text{-}on\ (\lambda xs.\{xs\}//\{\cong\})\ (set\ Fs_2)$ **and** *iso*: $is\text{-}pr\text{-}iso\ \varphi\ Fs_1\ Fs_2$
shows $length\ Fs_1 = length\ Fs_2$
⟨*proof*⟩

lemma *is-Hom-distinct*:
⟦ *is-pr-Hom* φ 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* φ F)
⟨*proof*⟩

lemma *Collect-congs-eq-iff*[*simp*]:
Collect ((≅) x) = *Collect* ((≅) y) ⟷ ($x \cong (y::'a\ list)$)
⟨*proof*⟩

lemma *is-pr-Hom-trans*: **assumes** f: *is-pr-Hom* f A B **and** g: *is-pr-Hom* g B C
shows *is-pr-Hom* ($g \circ f$) A C
⟨*proof*⟩

lemma *is-pr-Hom-rev*:
is-pr-Hom φ A $B \Longrightarrow$ *is-pr-Hom* φ (*rev* ‘ A) (*rev* ‘ B)
⟨*proof*⟩

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

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

lemma *is-iso-Cons*:
⟦ *distinct* ($F_1\#Fs_1'$); *distinct* Fs_2;
inj-on ($\lambda xs.\{xs\}//\{\cong\}$) (*set*($F_1\#Fs_1'$)); *inj-on* ($\lambda xs.\{xs\}//\{\cong\}$) (*set* Fs_2) ⟧
$\Longrightarrow$
is-pr-iso φ ($F_1\#Fs_1'$) Fs_2 =
($\exists F_2 \in$ *set* Fs_2. *length* F_1 = *length* $F_2 \wedge$ *is-pr-iso* φ Fs_1' (*remove1* F_2 Fs_2)
$\wedge$ ($\exists n$. *map* φ F_1 = *rotate* n F_2)
$\wedge$ *inj-on* φ (*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-of* (*zip* xs ys) $\subseteq_m$ *Some* $\circ$ f) = (*map* f xs = ys)
⟨*proof*⟩

primrec *pr-iso-test0* :: $('a \rightharpoonup 'b) \Rightarrow 'a$ *fgraph* $\Rightarrow 'b$ *fgraph* $\Rightarrow$ *bool* **where**
  pr-iso-test0 m [] *Fs*$_2$ = (*Fs*$_2$ = [])
| *pr-iso-test0 m* (*F*$_1$#*Fs*$_1$) *Fs*$_2$ =
   (∃ *F*$_2$ ∈ *set Fs*$_2$. *length F*$_1$ = *length F*$_2$ ∧
      (∃ *n*. *let m*′ = *map-of*(*zip F*$_1$ (*rotate n F*$_2$)) *in*
         if m ⊆$_m$ *m* ++ *m*′ ∧ *inj-on* (*m*++*m*′) (*dom*(*m*++*m*′))
         then pr-iso-test0 (*m* ++ *m*′) *Fs*$_1$ (*remove1 F*$_2$ *Fs*$_2$) *else False*))

lemma *map-compatI*: ⟦ *f* ⊆$_m$ *Some* ∘ *h*; *g* ⊆$_m$ *Some* ∘ *h* ⟧ ⟹ *f* ⊆$_m$ *f*++*g*
⟨*proof*⟩

lemma *inj-on-map-addI1*:
 ⟦ *inj-on m A*; *m* ⊆$_m$ *m*++*m*′; *A* ⊆ *dom m* ⟧ ⟹ *inj-on* (*m*++*m*′) *A*
⟨*proof*⟩

lemma *map-image-eq*: ⟦ *A* ⊆ *dom m*; *m* ⊆$_m$ *m*′ ⟧ ⟹ *m* ‘ *A* = *m*′ ‘ *A*
⟨*proof*⟩

lemma *inj-on-map-add-Un*:
 ⟦ *inj-on m* (*dom m*); *inj-on m*′ (*dom m*′); *m* ⊆$_m$ *Some* ∘ *f*; *m*′ ⊆$_m$ *Some* ∘ *f*;
    inj-on f (*dom m*′ ∪ *dom m*); *A* = *dom m*′; *B* = *dom m* ⟧
  ⟹ *inj-on* (*m* ++ *m*′) (*A* ∪ *B*)
⟨*proof*⟩

lemma *map-of-zip-eq-SomeD*: *length xs* = *length ys* ⟹
  map-of (*zip xs ys*) *x* = *Some y* ⟹ *y* ∈ *set ys*
⟨*proof*⟩

lemma *inj-on-map-of-zip*:
  ⟦ *length xs* = *length ys*; *distinct ys* ⟧
   ⟹ *inj-on* (*map-of* (*zip xs ys*)) (*set xs*)
⟨*proof*⟩

lemma *pr-iso-test0-correct*: ⋀*m Fs*$_2$.
 ⟦ ∀ *F*∈*set Fs*$_1$. *distinct F*; ∀ *F*∈*set Fs*$_2$. *distinct F*;
   distinct Fs$_1$; *inj-on* (λ*xs*.{*xs*}//{≅}) (*set Fs*$_1$);
   distinct Fs$_2$; *inj-on* (λ*xs*.{*xs*}//{≅}) (*set Fs*$_2$); *inj-on m* (*dom m*) ⟧ ⟹
      pr-iso-test0 m Fs$_1$ *Fs*$_2$ =
      (∃ φ. *is-pr-iso* φ *Fs*$_1$ *Fs*$_2$ ∧ *m* ⊆$_m$ *Some* ∘ φ ∧
          inj-on φ (*dom m* ∪ (⋃ *F*∈*set Fs*$_1$. *set F*)))
⟨*proof*⟩

corollary *pr-iso-test0-corr*:
 ⟦ ∀ *F*∈*set Fs*$_1$. *distinct F*; ∀ *F*∈*set Fs*$_2$. *distinct F*;
   distinct Fs$_1$; *inj-on* (λ*xs*.{*xs*}//{≅}) (*set Fs*$_1$);
   distinct Fs$_2$; *inj-on* (λ*xs*.{*xs*}//{≅}) (*set Fs*$_2$) ⟧ ⟹
      pr-iso-test0 Map.empty Fs$_1$ *Fs*$_2$ = (∃ φ. *is-pr-iso* φ *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).

primrec *pr-iso-test1* :: $('a \rightharpoonup 'b) \Rightarrow 'a\ fgraph \Rightarrow 'b\ fgraph \Rightarrow bool$ **where**
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*:
$\bigwedge m\ Fs_2.\ [] \notin set\ Fs_2 \Longrightarrow pr\text{-}iso\text{-}test1\ m\ Fs_1\ Fs_2 = pr\text{-}iso\text{-}test0\ m\ Fs_1\ Fs_2$
⟨*proof*⟩

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

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

2.3.1 Implementing maps by lists

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

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

type-synonym
$('a,'b)tester = ('a * 'b)list \Rightarrow ('a * 'b)list \Rightarrow bool$
type-synonym
$('a,'b)merger = ('a * 'b)list \Rightarrow ('a * 'b)list \Rightarrow ('a * 'b)list$

primrec *pr-iso-test2* :: $('a,'b)tester \Rightarrow ('a,'b)merger \Rightarrow$
$('a * 'b)list \Rightarrow 'a\ fgraph \Rightarrow 'b\ fgraph \Rightarrow bool$ **where**
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-on m* (*dom m*); *Some y* ∉ *range m* ⟧ ⟹ *inj-on* (*m*(*x*↦*y*)) (*dom m*)
⟨*proof*⟩

lemma [*simp*]:
distinct(*map snd xys*) ⟹ *inj-on* (*map-of xys*) (*dom*(*map-of xys*))
⟨*proof*⟩

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

lemma *pr-iso-test2-conv-1*:
⋀*I Fs*$_2$.
⟦ ∀ *I I'*. *oneone I* ⟶ *oneone I'* ⟶
tst I' I = (*let m* = *map-of I*; *m'* = *map-of I'*
in m ⊆$_m$ *m* ++ *m'* ∧ *inj-on* (*m*++*m'*) (*dom*(*m*++*m'*)));
∀ *I I'*. *oneone I* ⟶ *oneone I'* ⟶ *tst I' I*
⟶ *map-of*(*mrg I' I*) = *map-of I* ++ *map-of I'*;
∀ *I I'*. *oneone I* ∧ *oneone I'* ⟶ *tst I' I* ⟶ *oneone* (*mrg I' I*);
oneone I;
∀ *F* ∈ *set Fs*$_1$. *distinct F*; ∀ *F* ∈ *set Fs*$_2$. *distinct F* ⟧ ⟹
pr-iso-test2 tst mrg I Fs$_1$ *Fs*$_2$ = *pr-iso-test1* (*map-of I*) *Fs*$_1$ *Fs*$_2$
⟨*proof*⟩

A simple implementation

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

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

lemma *image-map-of-conv-Image*:
⋀*A*. ⟦ *distinct*(*map fst xys*) ⟧
⟹ *map-of xys* ‘ *A* = *Some* ‘ (*set xys* ‘‘ *A*) ∪ (*if A* ⊆ *fst* ‘ *set xys then* {} *else* {*None*})
⟨*proof*⟩

lemma [*simp*]: *m*++*m'* ‘ (*dom m'* − *A*) = *m'* ‘ (*dom m'* − *A*)
⟨*proof*⟩

declare *Diff-subset* [*iff*]

lemma *compat-correct*:

$[\![$ *oneone I*; *oneone I'* $]\!] \Longrightarrow$
compat I' I = (*let m* = *map-of I*; *m'* = *map-of I'*
in m $\subseteq_m$ *m* ++ *m'* $\wedge$ *inj-on* (*m*++*m'*) (*dom*(*m*++*m'*)))
⟨*proof*⟩

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

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

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

lemma *merge0-correct*:
$\forall$ *I I'*. *oneone I* $\longrightarrow$ *oneone I'* $\longrightarrow$ *compat I' I*
$\longrightarrow$ *map-of*(*merge0 I' I*) = *map-of I* ++ *map-of I'*
⟨*proof*⟩

lemma *merge0-inv*:
$\forall$ *I I'*. *oneone I* $\wedge$ *oneone I'* $\longrightarrow$ *compat I' I* $\longrightarrow$ *oneone* (*merge0 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*$_2$;
distinct Fs$_1$; *inj-on* (λ*xs*.{*xs*}//{$\cong$}) (*set Fs*$_1$);
distinct Fs$_2$; *inj-on* (λ*xs*.{*xs*}//{$\cong$}) (*set Fs*$_2$) $]\!] \Longrightarrow$
pr-iso-test2 compat merge0 [] *Fs*$_1$ *Fs*$_2$ = ($\exists$ φ. *is-pr-iso* φ *Fs*$_1$ *Fs*$_2$)
⟨*proof*⟩

Implementing merge as a recursive function:

primrec *merge* :: ('*a*,'*b*)*merger* **where**
merge [] *I* = *I*
| *merge* (*xy*#*xys*) *I* = (*let* (*x*,*y*) = *xy* *in*
if $\forall$ (*x'*,*y'*) $\in$ *set I*. *x* $\neq$ *x'* *then xy* # *merge xys I else merge xys I*)

lemma *merge-conv-merge0*: *merge I' I* = *merge0 I' I*
⟨*proof*⟩

primrec *pr-iso-test-rec* :: ('*a* * '*b*)*list* $\Rightarrow$ '*a fgraph* $\Rightarrow$ '*b fgraph* $\Rightarrow$ *bool* **where**
pr-iso-test-rec I [] *Fs*$_2$ = (*Fs*$_2$ = [])
| *pr-iso-test-rec I* (*F*$_1$#*Fs*$_1$) *Fs*$_2$ =

$(\exists\ F_2 \in set\ Fs_2.\ length\ F_1 = length\ F_2 \wedge$
$(\exists\, n < length\ F_2.\ let\ I' = zip\ F_1\ (rotate\ n\ F_2)\ in$
$compat\ I'\ I \wedge pr\text{-}iso\text{-}test\text{-}rec\ (merge\ I'\ I)\ Fs_1\ (remove1\ F_2\ Fs_2)))$

lemma *pr-iso-test-rec-conv-2*:
$\bigwedge I\ Fs_2.\ pr\text{-}iso\text{-}test\text{-}rec\ I\ Fs_1\ Fs_2 = pr\text{-}iso\text{-}test2\ compat\ merge0\ I\ Fs_1\ Fs_2$
⟨*proof*⟩

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

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

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

2.3.2 'Improper' Isomorphisms

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

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

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

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

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

theorem *iso-correct*:
$[\![\forall F{\in}set\ Fs_1.\ distinct\ F;\ \forall F{\in}set\ Fs_2.\ distinct\ F;\ [] \notin set\ Fs_2;$
$distinct\ Fs_1;\ inj\text{-}on\ (\lambda xs.\{xs\}//\{\cong\})\ (set\ Fs_1);$
$distinct\ Fs_2;\ inj\text{-}on\ (\lambda xs.\{xs\}//\{\cong\})\ (set\ Fs_2)\ ]\!] \Longrightarrow$

iso-test Fs_1 Fs_2 = ($Fs_1 \simeq Fs_2$)
⟨*proof*⟩

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

2.4 Elementhood and containment modulo

interpretation *qle-gr*: *quasi-order* ($\simeq$)
⟨*proof*⟩

abbreviation *qle-gr-in* :: *'a fgraph* ⇒ *'a fgraph set* ⇒ *bool* (**infix** $\in_\simeq$ *60*)
where $x \in_\simeq M \equiv$ *qle-gr.in-qle x M*
abbreviation *qle-gr-sub* :: *'a fgraph set* ⇒ *'a fgraph set* ⇒ *bool* (**infix** $\subseteq_\simeq$ *60*)
where $x \subseteq_\simeq M \equiv$ *qle-gr.subseteq-qle x M*
abbreviation *qle-gr-eq* :: *'a fgraph set* ⇒ *'a fgraph set* ⇒ *bool* (**infix** $=_\simeq$ *60*)
where $x =_\simeq M \equiv$ *qle-gr.seteq-qle x M*

end

3 More Rotation

theory *Rotation*
imports *ListAux PlaneGraphIso*
begin

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

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

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

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 $\forall vs \in A.\ distinct\ vs\ \ [] \notin A$
shows *inj-on rotate-min A* = *inj-on* ($\lambda vs.\ \{vs\}//\{\cong\}$) *A*
⟨*proof*⟩

end

4 Graph

theory *Graph*
imports *Rotation*
begin

syntax
-UNION1 :: *pttrns* ⇒ *'b set* ⇒ *'b set* (($3\bigcup$(‹*unbreakable*›_)/ -) [*0*, *10*] *10*)
-INTER1 :: *pttrns* ⇒ *'b set* ⇒ *'b set* (($3\bigcap$(‹*unbreakable*›_)/ -) [*0*, *10*] *10*)
-UNION :: *pttrn* ⇒ *'a set* ⇒ *'b set* ⇒ *'b set* (($3\bigcup$(‹*unbreakable*›_∈_)/ -) [*0*, *0*, *10*] *10*)
-INTER :: *pttrn* ⇒ *'a set* ⇒ *'b set* ⇒ *'b set* (($3\bigcap$(‹*unbreakable*›_∈_)/ -) [*0*, *0*, *10*] *10*)

4.1 Notation

type-synonym *vertex* = *nat*

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

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

4.2 Faces

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

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

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

consts *final* :: *'a* ⇒ *bool*
consts *type* :: *'a* ⇒ *facetype*

overloading
final-face ≡ *final* :: *face* ⇒ *bool*

type-face ≡ *type* :: *face* ⇒ *facetype*
vertices-face ≡ *vertices* :: *face* ⇒ *vertex list*
cong-face ≡ *pr-isomorphic* :: *face* ⇒ *face* ⇒ *bool*
begin

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

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

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

definition *cong-face* :: *face* ⇒ *face* ⇒ *bool*
where (f_1 :: *face*) ≅ f_2 ≡ *vertices* f_1 ≅ *vertices* f_2

end

The following operation makes a face final.

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

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

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

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

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

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

lemma *nextV2*: $f^2 \cdot v = f \cdot (f \cdot v)$
⟨*proof*⟩
overloading *op-vertices* ≡ *Graph.op* :: *vertex list* ⇒ *vertex list*
begin
definition (*vs*::*vertex list*)^{op} ≡ *rev vs*
end

overloading *op-graph* ≡ *Graph.op* :: *face* ⇒ *face*
begin
primrec *op-graph* **where** (*Face vs f*)^{op} = *Face* (*rev vs*) *f*
end
⟨*proof*⟩⟨*proof*⟩

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

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

4.3 Graphs

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

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

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

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

overloading
  vertices-graph ≡ *vertices* :: *graph* ⇒ *vertex list*
begin
  primrec *vertices-graph* **where** *vertices* (*Graph fs n f h*) = [0 ..< *n*]
end

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

lemma *in-vertices-graph*:
  v ∈ *set* (*vertices g*) = (*v* < *countVertices g*)
⟨*proof*⟩

lemma *len-vertices-graph*:
  |*vertices g*| = *countVertices g*
⟨*proof*⟩

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

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

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

definition *height* :: *graph* ⇒ *vertex* ⇒ *nat* **where**

height g v ≡ heights g ! v

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

4.4 Operations on graphs

final graph, final / nonfinal faces

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

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

definition *countNonFinals* :: *graph* ⇒ *nat* **where**
countNonFinals g ≡ |*nonFinals g*|

overloading *finalGraph* ≡ *final* :: *graph* ⇒ *bool*
begin
definition *finalGraph g* ≡ (*nonFinals g* = [])
end

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

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

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

lemma *finalVertex-final-face*[*dest*]:
finalVertex g v ⟹ *f* ∈ *set* (*facesAt g v*) ⟹ *final f*
⟨*proof*⟩

counting faces

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

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

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

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

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

lemma[*simp*]: *0* ≤ *tri g v* ⟨*proof*⟩

lemma[*simp*]: *0* ≤ *quad g v* ⟨*proof*⟩

lemma[*simp*]: *0* ≤ *except g v* ⟨*proof*⟩

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

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

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

overloading *edges-graph* ≡ *edges* :: *graph* ⇒ (*vertex* × *vertex*) *set*
begin
definition $\mathcal{E}$ (*g*::*graph*) ≡ $\bigcup_{f \in \mathcal{F}\ g}$ *edges f*
end

definition *neighbors* :: *graph* ⇒ *vertex* ⇒ *vertex list* **where**
neighbors g v ≡ [*f*•*v*. *f* ← *facesAt g v*]

4.5 Navigation in graphs

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

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

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

4.6 Code generator setup

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

lemma *final-face-code* [*code*]:
final-face (*Face vs Final*) ⟷ *True*
final-face (*Face vs Nonfinal*) ⟷ *False*

⟨*proof*⟩

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

lemma *final-graph-code* [*code*]: *final-graph g* = *List.null* (*nonFinals g*)
⟨*proof*⟩

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

lemma *vertices-face-code* [*code*]: *vertices-face* (*Face vs f*) = *vs*
⟨*proof*⟩

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

lemma *vertices-graph-code* [*code*]:
vertices-graph (*Graph fs n f h*) = [*0* ..< *n*]
⟨*proof*⟩

end

5 Syntax for operations on immutable arrays

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

5.1 Tabulation

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

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

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

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

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

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

5.2 Access

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

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

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

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

end

6 Enumerating Patches

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

Generates an Enumeration of lists. (See Kepler98, PartIII, section 8, p.11).
Used to construct all possible extensions of an unfinished outer face F with *outer* vertices by a new finished inner face with *inner* vertices, such a fixed edge e of the outer face is also contained in the inner face.
Label the vertices of F consecutively $0, \ldots, outer - 1$, with 0 and $outer - 1$ the endpoints of e.
Generate all lists

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

of length *inner*, such that $0 = a_0 \leq a_1 \ldots a_{inner-2} < a_{inner-1}$. Every list represents an inner face, with vertices $v_0, \ldots, v_{inner-1}$.
Construct the vertices $v_0, \ldots, v_{inner-1}$ inductively: If $i = 1$ or $a_i \neq a_{i-1}$, we set v_i to the vertex with index a_i of F. But if $a_i = a_{i-1}$, we add a new vertex

v_i to the planar map. The new face is to be drawn along the edge e over the face F.

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

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

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

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

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


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



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

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

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

end
```

7 Subdividing a Face

```
theory FaceDivision
imports Graph
```

begin

definition *split-face* :: *face* ⇒ *vertex* ⇒ *vertex* ⇒ *vertex list* ⇒ *face* × *face* **where**
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*)

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

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

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

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

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

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

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

end

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* ∈ *set* (*f x*)}*.

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

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

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

lemma *RTranCl-induct*:
 (*h*, *h'*) ∈ *RTranCl succs* ⟹
  P h ⟹

$(\bigwedge g\ g'.\ g' \in set\ (succs\ g) \Longrightarrow P\ g \Longrightarrow P\ g') \Longrightarrow$
$P\ h'$
⟨*proof*⟩

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

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

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

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

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

end

9 Plane Graph Enumeration

theory *Plane*
imports *Enumerator FaceDivision RTranCl*
begin

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

declare *maxGon-def* [*simp*]

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

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

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

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

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

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

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

definition *next-plane0* :: *nat* $\Rightarrow$ *graph* $\Rightarrow$ *graph list* $(next'\text{-}plane0_)$ **where**
next-plane0$_p$ *g* $\equiv$
$if\ final\ g\ then\ []$
$else \bigsqcup_{f \in nonFinals\ g} \bigsqcup_{v \in vertices\ f} \bigsqcup_{i \in [3..<Suc(maxGon\ p)]}\ generatePolygon\ i\ v\ f\ g$

definition *Seed* :: *nat* $\Rightarrow$ *graph* $(Seed_)$ **where**
$Seed_p \equiv graph(maxGon\ p)$

lemma *Seed-not-final*[*iff*]: $\neg$ *final* (*Seed p*)
⟨*proof*⟩

definition *PlaneGraphs0* :: *graph set* **where**
PlaneGraphs0 $\equiv \bigcup p.\ \{g.\ Seed_p\ [next\text{-}plane0_p]\rightarrow\!*\ g \wedge final\ g\}$

end

theory *Plane1*
imports *Plane*
begin

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

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

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

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

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

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

end

10 Properties of Graph Utilities

theory *GraphProps*
imports *Graph*
begin

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

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 ≠ [] ⟹ *f* · *v* ∈ $\mathcal{V}$ *f*
⟨*proof*⟩

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

lemma *nextVertex-prevVertex*[*simp*]:
⟦ *distinct*(*vertices f*); *v* ∈ $\mathcal{V}$ *f* ⟧
⟹ *f* · (*f*$^{-1}$ · *v*) = *v*
⟨*proof*⟩

lemma *prevVertex-nextVertex*[*simp*]:
⟦ *distinct*(*vertices f*); *v* ∈ $\mathcal{V}$ *f* ⟧
⟹ *f*$^{-1}$ · (*f* · *v*) = *v*
⟨*proof*⟩

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

lemma *nextVertex-nth*:
⟦ *distinct*(*vertices f*); *i* < |*vertices f*| ⟧ ⟹
f · (*vertices f* ! *i*) = *vertices f* ! ((*i*+*1*) *mod* |*vertices f*|)
⟨*proof*⟩

10.3 $\mathcal{E}$

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

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

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

lemma *vertices-conv-Union-edges*:
$\mathcal{V}$(*f*::*face*) = (⋃(*a*,*b*)∈$\mathcal{E}$ *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 *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-next2*:
$[\![\ distinct(vertices\ f);\ u \in \mathcal{V}\ f;\ v \in \mathcal{V}\ f;\ u \neq v\ ]\!] \Longrightarrow$

between (*vertices f*) *u* (*f* · *v*) = *between* (*vertices f*) *u v* @ [*v*]
⟨*proof*⟩

lemma *nextVertex-eq-lemma*:
⟦ *distinct*(*vertices f*); *x* ∈ $\mathcal{V}$ *f*; *y* ∈ $\mathcal{V}$ *f*; *x* ≠ *y*;
v ∈ *set*(*x* # *between* (*vertices f*) *x y*) ⟧ ⟹
f · *v* = *nextElem* (*x* # *between* (*vertices f*) *x y* @ [*y*]) *z v*
⟨*proof*⟩

end

11 Properties of Patch Enumeration

theory *EnumeratorProps*
imports *Enumerator GraphProps*
begin

lemma *length-hideDupsRec*[*simp*]: ⋀*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*⟩

definition *increasing* :: ('*a*::*linorder*) *list* ⇒ *bool* **where**
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*: $ls \in set\ (enumBase\ nmax) \Longrightarrow length\ ls = 1$
⟨*proof*⟩

lemma *enumBase-bound*: $\forall\, y \in set\ (enumBase\ nmax).\ \forall\, z \in set\ y.\ z \leq nmax$
⟨*proof*⟩

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

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

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

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

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

lemma *enumAppend-length2*: $\bigwedge ls.\ ls \in set\ ((enumAppend\ nmax\ ^\frown\ n)\ lss) \Longrightarrow$
$(\bigwedge l.\ l \in set\ lss \Longrightarrow |l| = k) \Longrightarrow K = k + n \Longrightarrow |ls| = K$
⟨*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*: $ls \in set\ (enumerator\ m\ n) \Longrightarrow 2 \leq length\ ls$
⟨*proof*⟩

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

lemma *enumerator-not-empty*[*dest*]: $ls \in set\ (enumerator\ m\ n) \Longrightarrow ls \neq\ []$
⟨*proof*⟩

lemma *enumerator-length2*: $ls \in set\ (enumerator\ m\ n) \Longrightarrow 2 < m \Longrightarrow length\ ls = m$
⟨*proof*⟩

lemma *enumerator-bound*: $ls \in set\ (enumerator\ m\ nmax) \Longrightarrow$
$0 < nmax \Longrightarrow x \in set\ ls \Longrightarrow x < nmax$
⟨*proof*⟩

lemma *enumerator-bound2*: $ls \in set\ (enumerator\ m\ nmax) \Longrightarrow 1 < nmax \Longrightarrow x \in set\ (butlast\ ls) \Longrightarrow x < nmax - Suc\ 0$
⟨*proof*⟩

lemma *enumerator-bound3*: $ls \in set\ (enumerator\ m\ nmax) \Longrightarrow 1 < nmax \Longrightarrow last\ (butlast\ ls) < nmax - Suc\ 0$
⟨*proof*⟩

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

lemma *enumerator-increasing*: $ls \in set\ (enumerator\ m\ nmax) \Longrightarrow increasing\ ls$
⟨*proof*⟩

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

lemma *incrIndexList-1lem*[*simp*]: *incrIndexList ls m nmax* ⟹ $Suc\ 0 < m$
⟨*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*: $2 < m \Longrightarrow$ *incrIndexList ls m nmax* $\Longrightarrow$
ls ∈ *set* (*enumerator m nmax*)
⟨*proof*⟩

lemma *enumerator-equiv*[*simp*]:
$2 < n \Longrightarrow 1 < m \Longrightarrow$ *is* ∈ *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 *in-set-repl-setFin*:
f ∈ *set fs* $\Longrightarrow$ *final f* $\Longrightarrow$ *f* ∈ *set* (*replace f'* [*setFinal f'*] *fs*)
⟨*proof*⟩

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

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

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

lemma *len-finals-makeFaceFinal*:
f ∈ $\mathcal{F}$ *g* $\Longrightarrow$ ¬ *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*

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

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

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

lemma *is-prefix-sublist*:
is-prefix ls vs ⟹ *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* ≠ *v* ⟹ *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* ⟹ *a* ∈ *set vs* ⟨*proof*⟩

lemma *is-sublist-in1*: *is-sublist* [*x*,*y*] *vs* ⟹ *y* ∈ *set vs* ⟨*proof*⟩

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

lemma *is-sublist-nth1*: *is-sublist* [*x*,*y*] *ls* ⟹
∃ *i j*. *i* < *length ls* ∧ *j* < *length ls* ∧ *ls*!*i* = *x* ∧ *ls*!*j* = *y* ∧ *Suc i* = *j*
⟨*proof*⟩

lemma *is-sublist-nth2*: ∃ *i j*. *i* < *length ls* ∧ *j* < *length ls* ∧ *ls*!*i* = *x* ∧ *ls*!*j* = *y* ∧

Suc i = j $\Longrightarrow$
is-sublist [*x*,*y*] *ls*
⟨*proof*⟩

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* ∈ *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-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* ≠ (*last as*) $\Longrightarrow$
is-sublist [*x*,*y*] *as* ∨ *is-sublist* [*x*,*y*] *bs*
⟨*proof*⟩

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

lemma *is-sublist-at5*: *distinct* (*as* @ *bs*) $\Longrightarrow$ *is-sublist* [*x*,*y*] (*as* @ *bs*) $\Longrightarrow$
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 $\Longrightarrow$ *distinct bs* $\Longrightarrow$ *set as* ∩ *set bs* = {} $\Longrightarrow$ *is-sublist* [*x*,*y*] (*as* @ *bs*) $\Longrightarrow$
is-sublist [*x*,*y*] *as* ∨ *is-sublist* [*x*,*y*] *bs* ∨ *x* = *last as* ∧ *y* = *hd bs*
⟨*proof*⟩

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

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

12.4 *is-nextElem*

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

lemma *is-nextElem-a*[*intro*]: *is-nextElem vs a b* ⟹ *a* ∈ *set vs*
⟨*proof*⟩
lemma *is-nextElem-b*[*intro*]: *is-nextElem vs a b* ⟹ *b* ∈ *set 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-sublistI*: *is-sublist* [*x*,*y*] *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-aux*:
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-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*⟩

12.5 *nextElem*, *sublist*, *is-nextElem*

lemma *is-sublist-eq*: *distinct vs* ⟹ *c* ≠ *y* ⟹

$(nextElem\ vs\ c\ x = y) = is\text{-}sublist\ [x,y]\ vs$
⟨*proof*⟩

lemma *is-nextElem1*: $distinct\ vs \Longrightarrow x \in set\ vs \Longrightarrow nextElem\ vs\ (hd\ vs)\ x = y \Longrightarrow is\text{-}nextElem\ vs\ x\ y$
⟨*proof*⟩

lemma *is-nextElem2*: $distinct\ vs \Longrightarrow x \in set\ vs \Longrightarrow is\text{-}nextElem\ vs\ x\ y \Longrightarrow nextElem\ vs\ (hd\ vs)\ x = y$
⟨*proof*⟩

lemma *nextElem-is-nextElem*:
$distinct\ xs \Longrightarrow x \in set\ xs \Longrightarrow$
$is\text{-}nextElem\ xs\ x\ y = (nextElem\ xs\ (hd\ xs)\ x = y)$
⟨*proof*⟩

lemma *nextElem-congs-eq*: $xs \cong ys \Longrightarrow distinct\ xs \Longrightarrow x \in set\ xs \Longrightarrow$
$nextElem\ xs\ (hd\ xs)\ x = nextElem\ ys\ (hd\ ys)\ x$
⟨*proof*⟩

lemma *is-sublist-is-nextElem*: $distinct\ vs \Longrightarrow is\text{-}nextElem\ vs\ x\ y \Longrightarrow is\text{-}sublist\ as\ vs \Longrightarrow x \in set\ as \Longrightarrow x \neq last\ as \Longrightarrow is\text{-}sublist\ [x,y]\ as$
⟨*proof*⟩

12.6 *before*

definition *before* :: $'a\ list \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
$before\ vs\ ram1\ ram2 \equiv \exists\ a\ b\ c.\ vs = a\ @\ ram1\ \#\ b\ @\ ram2\ \#\ c$

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

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

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

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

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

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

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

lemma *before-r1*:
before vs r1 r2 $\Longrightarrow$ *r1* ∈ *set vs* ⟨*proof*⟩

lemma *before-r2*:
before vs r1 r2 $\Longrightarrow$ *r2* ∈ *set vs* ⟨*proof*⟩

lemma *before-dist-r2*:
distinct vs $\Longrightarrow$ *before vs r1 r2* $\Longrightarrow$ *r2* ∈ *set* (*snd* (*splitAt r1 vs*))
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *before-dist-not1*:
distinct vs $\Longrightarrow$ *before vs ram1 ram2* $\Longrightarrow$ ¬ *before vs ram2 ram1*
⟨*proof*⟩

lemma *before-dist-not2*:
distinct vs $\Longrightarrow$ *ram1* $\in$ *set vs* $\Longrightarrow$ *ram2* $\in$ *set vs* $\Longrightarrow$ *ram1* $\neq$ *ram2* $\Longrightarrow$ $\neg$ (*before vs ram1 ram2*) $\Longrightarrow$ *before vs ram2 ram1*
⟨*proof*⟩

lemma *before-dist-eq*:
distinct vs $\Longrightarrow$ *ram1* $\in$ *set vs* $\Longrightarrow$ *ram2* $\in$ *set vs* $\Longrightarrow$ *ram1* $\neq$ *ram2* $\Longrightarrow$ ($\neg$ (*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*⟩

12.7 *between*

definition *pre-between* :: *'a list* $\Rightarrow$ *'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool* **where**
pre-between vs ram1 ram2 $\equiv$
distinct vs $\wedge$ *ram1* $\in$ *set vs* $\wedge$ *ram2* $\in$ *set vs* $\wedge$ *ram1* $\neq$ *ram2*

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

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

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

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

lemma *pre-between-r12*[*intro*]:
pre-between vs ram1 ram2 $\Longrightarrow$ *ram1* $\neq$ *ram2* ⟨*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* $\vee$ *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* $\Longrightarrow$ ($\neg$ *before vs ram1 ram2*) = *before vs ram2*

ram1
⟨*proof*⟩

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

lemma *between-simp1*[*simp*]:
before vs ram1 ram2 ⟹ *pre-between vs ram1 ram2* ⟹
between vs ram1 ram2 = *fst* (*splitAt ram2* (*snd* (*splitAt ram1 vs*)))
⟨*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 *between-congs*: *pre-between vs ram1 ram2* $\Longrightarrow$ *vs* $\cong$ *vs'* $\Longrightarrow$ *between vs ram1 ram2 = between vs' ram1 ram2*
⟨*proof*⟩

lemma *between-inter-empty*:
pre-between vs ram1 ram2 $\Longrightarrow$
set (*between vs ram1 ram2*) $\cap$ *set* (*between vs ram2 ram1*) = {}
⟨*proof*⟩

12.7.1 *between is-nextElem*

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

lemma *is-nextElem-or*: *pre-between vs ram1 ram2* $\Longrightarrow$ *is-nextElem vs x y* $\Longrightarrow$
is-sublist [*x*,*y*] (*ram1 # between vs ram1 ram2* @ [*ram2*]) $\vee$ *is-sublist* [*x*,*y*] (*ram2 # between vs ram2 ram1* @ [*ram1*])
⟨*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*⟩

12.7.2 *is-nextElem edges* equivalence

lemma *is-nextElem-edges1*: *distinct* (*vertices f*) ⟹ (*a*,*b*) ∈ *edges* (*f*::*face*) ⟹ *is-nextElem* (*vertices f*) *a b* ⟨*proof*⟩

lemma *is-nextElem-edges2*:
distinct (*vertices f*) ⟹ *is-nextElem* (*vertices f*) *a b* ⟹
(*a*,*b*) ∈ *edges* (*f*::*face*)
⟨*proof*⟩

lemma *is-nextElem-edges-eq*[*simp*]:
distinct (*vertices* (*f*::*face*)) ⟹
(*a*,*b*) ∈ *edges f* = *is-nextElem* (*vertices f*) *a b*
⟨*proof*⟩

12.7.3 *nextVertex*

lemma *nextElem-suc2*: *distinct* (*vertices f*) ⟹ *last* (*vertices f*) = *v* ⟹ *v* ∈ *set* (*vertices f*) ⟹ *f* · *v* = *hd* (*vertices f*)
⟨*proof*⟩

12.8 *split-face*

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

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

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

lemma *pre-split-face-symI*:
pre-split-face oldF ram1 ram2 newVertexList ⟹ *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 $\Longrightarrow$ *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* $\Longrightarrow$ *pre-split-face oldF ram1 ram2 newVertexList* $\Longrightarrow$ (*a*, *b*) ∈ *edges oldF* $\Longrightarrow$ (*a*,*b*) ∈ *edges f12* ∨ (*a*,*b*) ∈ *edges f21*
⟨*proof*⟩

12.9 *verticesFrom*

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

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

lemma *len-vFrom*[*simp*]:
v ∈ $\mathcal{V}$ *f* $\Longrightarrow$ |*verticesFrom f v*| = |*vertices f*|
⟨*proof*⟩

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

lemma *verticesFrom-congs*:

$v \in \mathcal{V}\ f \Longrightarrow (vertices\ f) \cong (verticesFrom\ f\ v)$
⟨*proof*⟩

lemma *verticesFrom-eq-if-vertices-cong*:
$[\![distinct(vertices\ f);\ distinct(vertices\ f');$
$vertices\ f \cong vertices\ f';\ x \in \mathcal{V}\ f\ ]\!] \Longrightarrow$
$verticesFrom\ f\ x = verticesFrom\ f'\ x$
⟨*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 *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*: $v \in \mathcal{V}\ f \Longrightarrow$
$is\text{-}nextElem\ (vertices\ f)\ a\ b = is\text{-}nextElem\ (verticesFrom\ f\ v)\ a\ b$
⟨*proof*⟩

lemma *verticesFrom-is-nextElem-last*: $v' \in \mathcal{V}\ f \Longrightarrow distinct\ (vertices\ f)$
$\Longrightarrow is\text{-}nextElem\ (verticesFrom\ f\ v')\ (last\ (verticesFrom\ f\ v'))\ v \Longrightarrow v = v'$
⟨*proof*⟩

lemma *verticesFrom-is-nextElem-hd*: $v' \in \mathcal{V}\ f \Longrightarrow distinct\ (vertices\ f)$
$\Longrightarrow is\text{-}nextElem\ (verticesFrom\ f\ v')\ u\ v' \Longrightarrow u = last\ (verticesFrom\ f\ v')$
⟨*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 *last-vFrom*:
$[\![\ distinct(vertices\ f);\ x \in \mathcal{V}\ f\ ]\!] \Longrightarrow last(verticesFrom\ f\ x) = f^{-1} \cdot x$
⟨*proof*⟩

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*⟩

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*⟩

12.10 *splitFace*

definition *pre-splitFace* :: *graph* ⇒ *vertex* ⇒ *vertex* ⇒ *face* ⇒ *vertex list* ⇒ *bool*
where

pre-splitFace g ram1 ram2 oldF nvs ≡
oldF ∈ $\mathcal{F}$ *g* ∧ ¬ *final oldF* ∧ *distinct* (*vertices oldF*) ∧ *distinct nvs*
∧ $\mathcal{V}$ *g* ∩ *set nvs* = {}
∧ $\mathcal{V}$ *oldF* ∩ *set nvs* = {}
∧ *ram1* ∈ $\mathcal{V}$ *oldF* ∧ *ram2* ∈ $\mathcal{V}$ *oldF*
∧ *ram1* ≠ *ram2*
∧ (((*ram1*,*ram2*) ∉ *edges oldF* ∧ (*ram2*,*ram1*) ∉ *edges oldF*
∧ (*ram1*, *ram2*) ∉ *edges g* ∧ (*ram2*, *ram1*) ∉ *edges g*) ∨ *nvs* ≠ [])

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

lemma *pre-splitFace-pre-split-face*[*simp*]:
pre-splitFace g ram1 ram2 oldF nvs ⟹ *pre-split-face oldF ram1 ram2 nvs*
⟨*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 *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 *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* ∈ $\mathcal{V}$ *f12*
⟨*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* ⟹
distinct (*vertices* (*fst* (*snd* (*splitFace g ram1 ram2 oldF newVertexList*))))
⟨*proof*⟩

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

lemma *splitFace-add-f21'*: *f'* ∈ ℱ *g'* ⟹ *fst* (*snd* (*splitFace g' v a f' nvl*))
∈ 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* ∈ ℱ (*snd* (*snd* (*splitFace g v a f' nvl*))) ⟹
f ∈ ℱ *g*
∨ *f* = *fst* (*splitFace g v a f' nvl*)
∨ *f* = (*fst* (*snd* (*splitFace g v a f' nvl*)))
⟨*proof*⟩

lemma *pre-FaceDiv-between1*: *pre-splitFace g′ ram1 ram2 f* [] ⟹
¬ *between* (*vertices f*) *ram1 ram2* = []
⟨*proof*⟩

lemma *pre-FaceDiv-between2*: *pre-splitFace g′ ram1 ram2 f* [] ⟹
¬ *between* (*vertices f*) *ram2 ram1* = []
⟨*proof*⟩

definition *Edges* :: *vertex list* ⇒ (*vertex* × *vertex*) *set* **where**
Edges vs ≡ {(*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*) ∈ *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-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*

$Edges\ xs \cup Edges\ ys \cup \{(last\ xs,\ hd\ ys)\})$
⟨*proof*⟩

lemma *Edges-rev-disj*: $distinct\ xs \Longrightarrow Edges(rev\ xs) \cap Edges(xs) = \{\}$
⟨*proof*⟩

lemma *disj-sets-disj-Edges*:
$set\ xs \cap set\ ys = \{\} \Longrightarrow Edges\ xs \cap Edges\ ys = \{\}$
⟨*proof*⟩

lemma *disj-sets-disj-Edges2*:
$set\ ys \cap set\ xs = \{\} \Longrightarrow Edges\ xs \cap Edges\ ys = \{\}$
⟨*proof*⟩

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

lemma *Edges-compl*:
$[\![\ distinct\ vs;\ x \in set\ vs;\ y \in set\ vs;\ x \neq y\ ]\!] \Longrightarrow$
$Edges(x\ \#\ between\ vs\ x\ y\ @\ [y]) \cap Edges(y\ \#\ between\ vs\ y\ x\ @\ [x]) = \{\}$
⟨*proof*⟩

lemma *Edges-disj*:
$[\![\ distinct\ vs;\ x \in set\ vs;\ z \in set\ vs;\ x \neq y;\ y \neq z;$
$y \in set(between\ vs\ x\ z)\ ]\!] \Longrightarrow$
$Edges(x\ \#\ between\ vs\ x\ y\ @\ [y]) \cap Edges(y\ \#\ between\ vs\ y\ z\ @\ [z]) = \{\}$
⟨*proof*⟩

lemma *edges-conv-Un-Edges*:
$[\![\ distinct(vertices(f{::}face));\ x \in \mathcal{V}\ f;\ y \in \mathcal{V}\ f;\ x \neq y\ ]\!] \Longrightarrow$
$\mathcal{E}\ f = Edges(x\ \#\ between\ (vertices\ f)\ x\ y\ @\ [y]) \cup$
$Edges(y\ \#\ between\ (vertices\ f)\ y\ x\ @\ [x])$
⟨*proof*⟩

lemma *Edges-between-edges*:
$[\![\ (a,b) \in Edges\ (u\ \#\ between\ (vertices(f{::}face))\ u\ v\ @\ [v]);$
$pre\text{-}split\text{-}face\ f\ u\ v\ vs\ ]\!] \Longrightarrow (a,b) \in \mathcal{E}\ f$
⟨*proof*⟩

lemma *edges-split-face1*: *pre-split-face f u v vs* $\Longrightarrow$
$\mathcal{E}(fst(split\text{-}face\ f\ u\ v\ vs)) =$
$Edges(v\ \#\ rev\ vs\ @\ [u]) \cup Edges(u\ \#\ between\ (vertices\ f)\ u\ v\ @\ [v])$
⟨*proof*⟩

lemma *edges-split-face2*: *pre-split-face f u v vs* $\Longrightarrow$
$\mathcal{E}(snd(split\text{-}face\ f\ u\ v\ vs)) =$
$Edges(u\ \#\ vs\ @\ [v]) \cup Edges(v\ \#\ between\ (vertices\ f)\ v\ u\ @\ [u])$
⟨*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)* $\in$ *edges f21*
⟨*proof*⟩

lemma *split-face-empty-ram1-ram2-in-f21'*:
pre-split-face oldF ram1 ram2 [] $\Longrightarrow$
(ram1, ram2) $\in$ *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) $\in$ *edges f21*
⟨*proof*⟩

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

lemma *split-face-edges-f12*:
assumes *vors*: *pre-split-face f ram1 ram2 vs*
(f12, f21) = split-face f ram1 ram2 vs
vs $\neq$ [] *vs1 = between (vertices f) ram1 ram2 vs1* $\neq$ []
shows *edges f12* =
$\{(hd\ vs,\ ram1)\ ,\ (ram1,\ hd\ vs1),\ (last\ vs1,\ ram2),\ (ram2,\ last\ vs)\} \cup$
$Edges(rev\ vs) \cup Edges\ vs1$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *split-face-edges-f12-vs*:
assumes *vors*: *pre-split-face f ram1 ram2* []
(f12, f21) = split-face f ram1 ram2 []
vs1 = between (vertices f) ram1 ram2 vs1 $\neq$ []
shows *edges f12* $= \{(ram2,\ ram1)\ ,\ (ram1,\ hd\ vs1),\ (last\ vs1,\ ram2)\} \cup$
Edges vs1 (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *split-face-edges-f12-bet*:

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

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

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

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

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

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

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*:
assumes *vors*: *pre-split-face f ram1 ram2 vs*
between (*vertices f*) *ram1 ram2* = []
between (*vertices f*) *ram2 ram1* = []
shows *edges f* = {(*ram2*, *ram1*) , (*ram1*, *ram2*)} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

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

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

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

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*⟩

12.11 *removeNones*

definition *removeNones* :: *'a option list* $\Rightarrow$ *'a list* **where**
removeNones vOptionList $\equiv$ [*the x*. *x* $\leftarrow$ *vOptionList*, *x* $\neq$ *None*]

declare *removeNones-def* [*simp*]
lemma *removeNones-inI*[*intro*]: *Some a* $\in$ *set ls* $\Longrightarrow$ *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-empty*[*simp*]: *removeNones* [] = [] ⟨*proof*⟩
declare *removeNones-def* [*simp del*]

12.12 *natToVertexList*

primrec *natToVertexListRec* ::
  nat $\Rightarrow$ *vertex* $\Rightarrow$ *face* $\Rightarrow$ *nat list* $\Rightarrow$ *vertex option list*
where
  natToVertexListRec old v f [] = [] |

natToVertexListRec old v f (i#is) =
(if i = old then None#natToVertexListRec i v f is
else Some (f^i • v)
 # natToVertexListRec i v f is)

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

12.13 *indexToVertexList*

lemma *nextVertex-inj*:
distinct (vertices f) ⟹ v ∈ 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 ∈ 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 ∈ 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 *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 *nths-take*: ⋀ *n iset*. ∀ *i* ∈ *iset*. *i* < *n* ⟹ *nths* (*take n xs*) *iset* = *nths xs iset*
⟨*proof*⟩

lemma *nths-reduceIndices*: ⋀ *iset*. *nths xs iset* = *nths xs* {*i*. *i* < *length xs* ∧ *i* ∈ *iset*}
⟨*proof*⟩

lemma *natToVertexList-nths1*: *distinct* (*vertices f*) ⟹
v ∈ 𝒱 *f* ⟹ *vs* = *verticesFrom f v* ⟹

incrIndexList es (length es) (length vs) $\Longrightarrow$ *n* $\leq$ *length es* $\Longrightarrow$
nths (take (Suc (es!(n − 1))) vs) (set (take n es))
= removeNones (take n (natToVertexList v f es))
⟨*proof*⟩

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

lemma *filter-Cons2*:
x $\notin$ *set ys* $\Longrightarrow$ [*y*←*ys*. *y = x* $\vee$ *P y*] = [*y*←*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*←*verticesFrom f v*. *x* $\in$ *set (removeNones (natToVertexList v f es))*]
= removeNones (natToVertexList v f es)
⟨*proof*⟩

definition *is-duplicateEdge* :: *graph* ⇒ *face* ⇒ *vertex* ⇒ *vertex* ⇒ *bool* **where**
is-duplicateEdge g f a b ≡
((*a*, *b*) ∈ *edges g* ∧ (*a*, *b*) ∉ *edges f* ∧ (*b*, *a*) ∉ *edges f*)
∨ ((*b*, *a*) ∈ *edges g* ∧ (*b*, *a*) ∉ *edges f* ∧ (*a*, *b*) ∉ *edges f*)

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

12.14 *pre-subdivFace(′)*

definition *pre-subdivFace-face* :: *face* ⇒ *vertex* ⇒ *vertex option list* ⇒ *bool* **where**
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′* ∈ $\mathcal{V}$ *f*
∧ *last* (*vOptionList*) = *Some* (*last* (*verticesFrom f v′*))
∧ *hd* (*tl* (*vOptionList*)) ≠ *last* (*vOptionList*)
∧ *2* < | *vOptionList* |
∧ *vOptionList* ≠ []

$\wedge$ *tl* (*vOptionList*) $\neq$ []

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

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

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

lemma *pre-subdivFace-in-f*[*intro*]: *pre-subdivFace g f v ls* $\Longrightarrow$ *Some a* $\in$ *set ls* $\Longrightarrow$ *a* $\in$ *set* (*verticesFrom f v*)
⟨*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 *filter-congs-shorten1*: *distinct* (*verticesFrom f v*) $\Longrightarrow$ [*v*$\leftarrow$*verticesFrom f v* . *v* = *a* $\vee$ *v* $\in$ *set vs*] = (*a* # *vs*)
$\Longrightarrow$ [*v*$\leftarrow$*verticesFrom f v* . *v* $\in$ *set vs*] = *vs*
⟨*proof*⟩

lemma *ovl-shorten*: *distinct* (*verticesFrom f v*) $\Longrightarrow$

$[v \leftarrow verticesFrom\ f\ v\ .\ v \in set\ (removeNones\ (va\ \#\ vol))] = (removeNones\ (va\ \#\ vol))$
$\Longrightarrow [v \leftarrow verticesFrom\ f\ v\ .\ v \in set\ (removeNones\ (vol))] = (removeNones\ (vol))$
⟨*proof*⟩

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

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

lemma *pre-subdivFace-pre-subdivFace'*: $v \in \mathcal{V}\ f \Longrightarrow pre\text{-}subdivFace\ g\ f\ v\ (vo\ \#\ vol) \Longrightarrow$
$pre\text{-}subdivFace'\ g\ f\ v\ v\ 0\ (vol)$
⟨*proof*⟩

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

lemma *natToVertexList-pre-subdivFace-face*:
$\neg\ final\ f \Longrightarrow distinct\ (vertices\ f) \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow 2 < |es| \Longrightarrow$
$incrIndexList\ es\ (length\ es)\ (length\ (vertices\ f)) \Longrightarrow$
$pre\text{-}subdivFace\text{-}face\ f\ v\ (natToVertexList\ v\ f\ es)$
⟨*proof*⟩

lemma *indexToVertexList-pre-subdivFace-face*:
$\neg\ final\ f \Longrightarrow distinct\ (vertices\ f) \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow 2 < |es| \Longrightarrow$
$incrIndexList\ es\ (length\ es)\ (length\ (vertices\ f)) \Longrightarrow$
$pre\text{-}subdivFace\text{-}face\ f\ v\ (indexToVertexList\ f\ v\ es)$
⟨*proof*⟩

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

lemma *pre-subdivFace'-None*:
$pre\text{-}subdivFace'\ g\ f\ v'\ v\ n\ (None\ \#\ vol) \Longrightarrow$
$pre\text{-}subdivFace'\ g\ f\ v'\ v\ (Suc\ n)\ vol$
⟨*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*:
$distinct\ xs \Longrightarrow xs = (as\ @\ u\ \#\ bs) \Longrightarrow [v{\leftarrow}xs.\ v = u \vee P\ v] = u\ \#\ us \Longrightarrow [v{\leftarrow}bs.\ P\ v] = us \wedge [v{\leftarrow}as.\ P\ v] = []$
⟨*proof*⟩

lemma *filter-distinct-at3*: $distinct\ xs \Longrightarrow xs = (as\ @\ u\ \#\ bs) \Longrightarrow [v{\leftarrow}xs.\ v = u \vee P\ v] = u\ \#\ us \Longrightarrow \forall\ z \in set\ zs.\ z \in set\ as \vee \neg\ (\ P\ z) \Longrightarrow [v{\leftarrow}zs@bs.\ P\ v] = us$
⟨*proof*⟩

lemma *filter-distinct-at4*: $distinct\ xs \Longrightarrow xs = (as\ @\ u\ \#\ bs)$
$\Longrightarrow [v{\leftarrow}xs.\ v = u \vee v \in set\ us] = u\ \#\ us$
$\Longrightarrow set\ zs \cap set\ us \subseteq \{u\} \cup set\ as$
$\Longrightarrow [v \leftarrow zs@bs.\ v \in set\ us] = us$
⟨*proof*⟩

lemma *filter-distinct-at5*: $distinct\ xs \Longrightarrow xs = (as\ @\ u\ \#\ bs)$
$\Longrightarrow [v{\leftarrow}xs.\ v = u \vee v \in set\ us] = u\ \#\ us$
$\Longrightarrow set\ zs \cap set\ xs \subseteq \{u\} \cup set\ as$
$\Longrightarrow [v \leftarrow zs@bs.\ v \in set\ us] = us$
⟨*proof*⟩

lemma *filter-distinct-at6*: $distinct\ xs \Longrightarrow xs = (as\ @\ u\ \#\ bs)$
$\Longrightarrow [v{\leftarrow}xs.\ v = u \vee v \in set\ us] = u\ \#\ us$

$\Longrightarrow$ *set zs* $\cap$ *set xs* $\subseteq$ $\{u\}$ $\cup$ *set as*
$\Longrightarrow$ $[v \leftarrow zs@bs.\ v \in set\ us] = us \wedge [v \leftarrow bs.\ v \in set\ us] = us$
⟨*proof*⟩

lemma *filter-distinct-at-special*:
distinct xs $\Longrightarrow$ $xs = (as\ @\ u\ \#\ bs)$
$\Longrightarrow$ $[v \leftarrow xs.\ v = u \vee v \in set\ us] = u\ \#\ us$
$\Longrightarrow$ *set zs* $\cap$ *set xs* $\subseteq$ $\{u\}$ $\cup$ *set as*
$\Longrightarrow$ *us* = *hd-us* # *tl-us*
$\Longrightarrow$ $[v \leftarrow zs@bs.\ v \in set\ us] = us \wedge hd\text{-}us \in set\ bs$
⟨*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* $\Longrightarrow$ *distinct xs* $\Longrightarrow$ *before ys u v* $\Longrightarrow$ *before xs u v*
⟨*proof*⟩

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

lemma *pre-subdivFace'-preFaceDiv*: *pre-subdivFace' g f v' v n* ((*Some u*) # *vol*)
$\Longrightarrow$ $f \in \mathcal{F}\ g \Longrightarrow (f \cdot v = u \longrightarrow n \neq 0) \Longrightarrow \mathcal{V}\ f \subseteq \mathcal{V}\ g$
$\Longrightarrow$ *pre-splitFace g v u f* [*countVertices g* ..< *countVertices g* + *n*]
⟨*proof*⟩

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

end

13 Invariants of (Plane) Graphs

theory *Invariants*
imports *FaceDivisionProps*
begin

13.1 Rotation of face into normal form

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

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

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

lemma *normFaces-distinct*: *distinct* (*normFaces fl*) ⟹ *distinct fl*
⟨*proof*⟩

13.2 Minimal (plane) graph properties

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

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

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

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

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

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

definition *faces-subset* :: *graph* ⇒ *bool* **where**
faces-subset g ≡ ∀ *f* ∈ $\mathcal{F}$ *g*. $\mathcal{V}$ *f* ⊆ $\mathcal{V}$ *g*

definition *edges-disj* :: *graph* ⇒ *bool* **where**
edges-disj g ≡

$\forall f \in \mathcal{F}\ g.\ \forall f' \in \mathcal{F}\ g.\ f \neq f' \longrightarrow \mathcal{E}\ f \cap \mathcal{E}\ f' = \{\}$

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

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

definition *one-final* :: *graph* ⇒ *bool* **where**
one-final g ≡ *one-final-but g* {}

definition *minGraphProps* :: *graph* ⇒ *bool* **where**
minGraphProps g ≡ *minGraphProps' g* ∧ *facesAt-eq g* ∧ *faceListAt-len g* ∧ *facesAt-distinct g* ∧ *faces-distinct g* ∧ *faces-subset g* ∧ *edges-sym g* ∧ *edges-disj g* ∧ *face-face-op g*

definition *inv* :: *graph* ⇒ *bool* **where**
inv g ≡ *minGraphProps g* ∧ *one-final g* ∧ |*faces g*| ≥ *2*

lemma *facesAt-distinctI*:
($\bigwedge v.\ v \in \mathcal{V}\ g \Longrightarrow$ *distinct* (*normFaces* (*facesAt g v*))) ⟹ *facesAt-distinct g*
⟨*proof*⟩

lemma *minGraphProps2*:
minGraphProps g ⟹ $f \in \mathcal{F}\ g$ ⟹ *2* < |*vertices f*|
⟨*proof*⟩

lemma *mgp-vertices3*:
minGraphProps g ⟹ $f \in \mathcal{F}\ g$ ⟹ |*vertices f*| ≥ *3*
⟨*proof*⟩

lemma *mgp-vertices-nonempty*:
minGraphProps g ⟹ $f \in \mathcal{F}\ g$ ⟹ *vertices f* ≠ []
⟨*proof*⟩

lemma *minGraphProps3*:
minGraphProps g ⟹ $f \in \mathcal{F}\ g$ ⟹ *distinct* (*vertices f*)
⟨*proof*⟩

lemma *minGraphProps4*:
minGraphProps g ⟹ (*length* (*faceListAt g*) = *countVertices g*)
⟨*proof*⟩

lemma *minGraphProps5*:
$[\![minGraphProps\ g;\ v : \mathcal{V}\ g;\ f \in set\ (facesAt\ g\ v)]\!] \Longrightarrow f \in \mathcal{F}\ g$
⟨*proof*⟩

lemma *minGraphProps6*:
$minGraphProps\ g \Longrightarrow v : \mathcal{V}\ g \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow v \in \mathcal{V}\ f$
⟨*proof*⟩

lemma *minGraphProps9*:
$minGraphProps\ g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow v \in \mathcal{V}\ g$
⟨*proof*⟩

lemma *minGraphProps7*:
$minGraphProps\ g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow\ f \in set\ (facesAt\ g\ v)$
⟨*proof*⟩

lemma *minGraphProps-facesAt-eq*: $minGraphProps\ g \Longrightarrow$
$v \in \mathcal{V}\ g \Longrightarrow set\ (facesAt\ g\ v) = \{f \in \mathcal{F}\ g.\ v \in \mathcal{V}\ f\}$
⟨*proof*⟩

lemma *mgp-dist-facesAt*[*simp*]:
$minGraphProps\ g \Longrightarrow v : \mathcal{V}\ g \Longrightarrow distinct\ (facesAt\ g\ v)$
⟨*proof*⟩

lemma *minGraphProps8*:
$minGraphProps\ g \Longrightarrow v : \mathcal{V}\ g \Longrightarrow distinct\ (normFaces\ (facesAt\ g\ v))$
⟨*proof*⟩

lemma *minGraphProps8a*:
$minGraphProps\ g \Longrightarrow v \in \mathcal{V}\ g \Longrightarrow distinct\ (normFaces\ (faceListAt\ g\ !\ v))$
⟨*proof*⟩

lemma *minGraphProps8a′*: $minGraphProps\ g \Longrightarrow$
$v < countVertices\ g \Longrightarrow distinct\ (normFaces\ (faceListAt\ g\ !\ v))$
⟨*proof*⟩

lemma *minGraphProps9′*:
$minGraphProps\ g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow v < countVertices\ g$
⟨*proof*⟩

lemma *minGraphProps10*:
$minGraphProps\ g \Longrightarrow (a,\ b) \in edges\ g \Longrightarrow (b,\ a) \in edges\ g$
⟨*proof*⟩

lemma *minGraphProps11*:
minGraphProps g $\Longrightarrow$ *distinct* (*normFaces* (*faces g*))
⟨*proof*⟩

lemma *minGraphProps11′*:
minGraphProps g $\Longrightarrow$ *distinct* (*faces g*)
⟨*proof*⟩

lemma *minGraphProps12*:
minGraphProps g $\Longrightarrow$ $f \in \mathcal{F}\ g \Longrightarrow (a,b) \in \mathcal{E}\ f \Longrightarrow (b,a) \notin \mathcal{E}\ f$
⟨*proof*⟩

lemma *minGraphProps7′*: *minGraphProps g* $\Longrightarrow$
$f \in \mathcal{F}\ g \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow$ *f* ∈ *set* (*faceListAt g* ! *v*)
⟨*proof*⟩

lemma *mgp-edges-disj*:
⟦ *minGraphProps g*; $f \neq f'$; $f \in \mathcal{F}\ g$; $f' \in \mathcal{F}\ g$ ⟧ $\Longrightarrow$
$uv \in \mathcal{E}\ f \Longrightarrow uv \notin \mathcal{E}\ f'$
⟨*proof*⟩

lemma *one-final-but-antimono*:
one-final-but g E $\Longrightarrow$ $E \subseteq E' \Longrightarrow$ *one-final-but g E′*
⟨*proof*⟩

lemma *one-final-antimono*: *one-final g* $\Longrightarrow$ *one-final-but g E*
⟨*proof*⟩

lemma *inv-two-faces*: *inv g* $\Longrightarrow$ $|faces\ g| \geq 2$
⟨*proof*⟩

lemma *inv-mgp*[*simp*]: *inv g* $\Longrightarrow$ *minGraphProps g*
⟨*proof*⟩

lemma *makeFaceFinal-id*[*simp*]: *final f* $\Longrightarrow$ *makeFaceFinal f g* = *g*
⟨*proof*⟩

lemma *inv-one-finalD′*:
⟦ *inv g*; $f \in \mathcal{F}\ g$; ¬ *final f*; $(a,b) \in \mathcal{E}\ f$ ⟧ $\Longrightarrow$
$\exists f' \in \mathcal{F}\ g.$ *final* $f' \wedge f' \neq f \wedge (b,a) \in \mathcal{E}\ f'$
⟨*proof*⟩

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

lemma *mgp-no-loop*[*simp*]:
minGraphProps g $\Longrightarrow$ $f \in \mathcal{F}\ g \Longrightarrow v \in \mathcal{V}\ f \Longrightarrow f \cdot v \neq v$
⟨*proof*⟩

lemma *mgp-facesAt-no-loop*:
minGraphProps g $\Longrightarrow$ $v : \mathcal{V}\ g \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow f \cdot v \neq v$
⟨*proof*⟩

lemma *edge-pres-faceAt*:
⟦ *minGraphProps g*; $u : \mathcal{V}\ g$; $f \in set(facesAt\ g\ u)$; $(u,v) \in \mathcal{E}\ f$ ⟧ $\Longrightarrow$
$f \in set(facesAt\ g\ v)$
⟨*proof*⟩

lemma *in-facesAt-nextVertex*:
minGraphProps g $\Longrightarrow$ $v : \mathcal{V}\ g \Longrightarrow f \in set(facesAt\ g\ v) \Longrightarrow f \in set(facesAt\ g\ (f \cdot v))$
⟨*proof*⟩

lemma *mgp-edge-face-ex*:
assumes [*intro*]: *minGraphProps g* $v : \mathcal{V}\ g$
and *fv*: $f \in set(facesAt\ g\ v)$ **and** *uv*: $(u,v) \in \mathcal{E}\ f$
shows $\exists f' \in set(facesAt\ g\ v).\ (v,u) \in \mathcal{E}\ f'$
⟨*proof*⟩

lemma *nextVertex-in-graph*:
minGraphProps g $\Longrightarrow$ $v : \mathcal{V}\ g \Longrightarrow f \in set(facesAt\ g\ v) \Longrightarrow f \cdot v : \mathcal{V}\ g$
⟨*proof*⟩

lemma *mgp-nextVertex-face-ex2*:
assumes *mgp*[*intro*]: *minGraphProps g* $v : \mathcal{V}\ g$ **and** *f*: $f \in set(facesAt\ g\ v)$
shows $\exists f' \in set(facesAt\ g\ (f \cdot v)).\ f' \cdot (f \cdot v) = v$
⟨*proof*⟩

lemma *inv-finals-nonempty*: *inv g* $\Longrightarrow$ *finals g* $\neq$ []
⟨*proof*⟩

13.3 *containsDuplicateEdge*

definition
containsUnacceptableEdgeSnd' :: (*nat* ⇒ *nat* ⇒ *bool*) ⇒ *nat list* ⇒ *bool* **where**
containsUnacceptableEdgeSnd' N is ≡
(∃ *k* < |*is*| − *2*. *let i0* = *is*!*k*; *i1* = *is*!(*k*+*1*); *i2* = *is*!(*k*+*2*) *in*
N i1 i2 ∧ (*i0* < *i1*) ∧ (*i1* < *i2*))

lemma *containsUnacceptableEdgeSnd-eq*:
containsUnacceptableEdgeSnd N v is = *containsUnacceptableEdgeSnd' N* (*v*#*is*)
⟨*proof*⟩

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*

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

lemma *replacefacesAt-eq*[*THEN eq-reflection*]:
replacefacesAt ns oldf newfs F = replacefacesAt2 ns oldf newfs F
⟨*proof*⟩

lemma *replacefacesAt2-notin*:
i ∉ *set is* ⟹ (*replacefacesAt2 is olfF newFs Fss*)!*i* = *Fss*!*i*
⟨*proof*⟩

lemma *replacefacesAt2-in*:
i ∈ *set is* ⟹ *distinct is* ⟹ *i < |Fss|* ⟹
(*replacefacesAt2 is olfF newFs Fss*)!*i* = *replace olfF newFs* (*Fss* !*i*)
⟨*proof*⟩

lemma *distinct-replacefacesAt21*:
i < |Fss| ⟹ *i* ∈ *set is* ⟹ *distinct is* ⟹ *distinct* (*Fss*!*i*) ⟹ *distinct newFs*
⟹
set (*Fss* ! *i*) ∩ *set newFs* ⊆ {*olfF*} ⟹
distinct ((*replacefacesAt2 is olfF newFs Fss*)! *i*)
⟨*proof*⟩

lemma *distinct-replacefacesAt22*:
i < |Fss| ⟹ *i* ∉ *set is* ⟹ *distinct is* ⟹ *distinct* (*Fss*!*i*) ⟹ *distinct newFs*
⟹

set (*Fss* ! *i*) ∩ *set newFs* ⊆ {*olfF*} ⟹
distinct ((*replacefacesAt2 is olfF newFs Fss*)! *i*)
⟨*proof*⟩

lemma *distinct-replacefacesAt2-2*:
i < |*Fss*| ⟹ *distinct is* ⟹ *distinct* (*Fss*!*i*) ⟹ *distinct newFs* ⟹
set (*Fss* ! *i*) ∩ *set newFs* ⊆ {*olfF*} ⟹
distinct ((*replacefacesAt2 is olfF newFs Fss*)! *i*)
⟨*proof*⟩

lemma *replacefacesAt2-nth1*:
k ∉ *set ns* ⟹ (*replacefacesAt2 ns oldf newfs F*) ! *k* = *F* ! *k*
⟨*proof*⟩

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

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

lemma *replacefacesAt2-length*[*simp*]:
|*replacefacesAt2 nvs f′ f″ vs*| = |*vs*|
⟨*proof*⟩

lemma *replacefacesAt2-nth*: *k* ∈ *set ns* ⟹ *k* < |*F*| ⟹ *oldf* ∉ *set newfs* ⟹
distinct (*F*!*k*) ⟹ *distinct newfs* ⟹ *oldf* ∈ *set* (*F*!*k*) ⟶ *set newfs* ∩ *set* (*F*!*k*)
⊆ {*oldf*} ⟹
(*replacefacesAt2 ns oldf newfs F*) ! *k* = (*replace oldf newfs* (*F*!*k*))
⟨*proof*⟩

lemma *replacefacesAt-notin*:
i ∉ *set is* ⟹ (*replacefacesAt is olfF newFs Fss*)!*i* = *Fss*!*i*
⟨*proof*⟩

lemma *replacefacesAt-in*:
i ∈ *set is* ⟹ *distinct is* ⟹ *i* < |*Fss*| ⟹
(*replacefacesAt is olfF newFs Fss*)!*i* = *replace olfF newFs* (*Fss* !*i*)
⟨*proof*⟩

lemma *replacefacesAt-length*[*simp*]: |*replacefacesAt nvs f′* [*f″*] *vs*| = |*vs*|
⟨*proof*⟩

lemma *replacefacesAt-nth2*: *k* < |*F*| ⟹
(*replacefacesAt* [*k*] *oldf newfs F*) ! *k* = *replace oldf newfs* (*F*!*k*)
⟨*proof*⟩

lemma *replacefacesAt-nth*: $k \in$ *set ns* $\Longrightarrow k <$ *|F|* $\Longrightarrow$ *oldf* $\notin$ *set newfs* $\Longrightarrow$
distinct (*F*!*k*) $\Longrightarrow$ *distinct newfs* $\Longrightarrow$ *oldf* $\in$ *set* (*F*!*k*) $\longrightarrow$ *set newfs* $\cap$ *set* (*F*!*k*)
$\subseteq \{$*oldf*$\} \Longrightarrow$
(*replacefacesAt ns oldf newfs F*) ! *k* = (*replace oldf newfs* (*F*!*k*))
⟨*proof*⟩

lemma *replacefacesAt2-5*: *x* $\in$ *set* (*replacefacesAt2 ns oldf newfs F* ! *k*) $\Longrightarrow$ *x* $\in$
set (*F*!*k*) $\vee$ *x* $\in$ *set newfs*
⟨*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*]:
$\bigwedge$*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-replace-in*:
normFace a $\in$ *set* (*normFaces* (*replace oldF newFs fs*)) $\Longrightarrow$
normFace a $\in$ *set* (*normFaces newFs*) $\vee$ *normFace a* $\in$ *set* (*normFaces fs*)
⟨*proof*⟩

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

(*replace oldF newFs fs*))
⟨*proof*⟩

lemma *distinct-replacefacesAt1-norm*:
i < |*Fss*| ⟹ *i* ∈ *set is* ⟹ *distinct is* ⟹ *distinct* (*normFaces* (*Fss*!*i*)) ⟹ *distinct* (*normFaces newFs*) ⟹
set (*normFaces* (*Fss* ! *i*)) ∩ *set* (*normFaces newFs*) ⊆ {} ⟹
distinct (*normFaces* ((*replacefacesAt is oldF newFs Fss*)! *i*))
⟨*proof*⟩

lemma *distinct-replacefacesAt2-norm*:
i < |*Fss*| ⟹ *i* ∉ *set is* ⟹ *distinct is* ⟹ *distinct* (*normFaces* (*Fss*!*i*)) ⟹ *distinct* (*normFaces newFs*) ⟹
set (*normFaces* (*Fss* ! *i*)) ∩ *set* (*normFaces newFs*) ⊆ {} ⟹
distinct (*normFaces* ((*replacefacesAt is oldF newFs Fss*)! *i*))
⟨*proof*⟩

lemma *distinct-replacefacesAt-norm*:
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*| ⟹ 2 < |*vertices f12*| ⟹ 2 < |*vertices f21*| ⟹
(*f12*, *f21*) = *split-face oldF ram1 ram2 vs* ⟹ *normFace f12* ≠ *normFace f21*
⟨*proof*⟩

lemma *normFace-in*: *f* ∈ *set fs* ⟹ *normFace f* ∈ *set* (*normFaces fs*)
⟨*proof*⟩

13.6 Invariants of *splitFace*

lemma *splitFace-holds-minGraphProps'*:
pre-splitFace g' v a f' vs ⟹ *minGraphProps' g'* ⟹

minGraphProps' (*snd* (*snd* (*splitFace g' v a f' vs*)))
⟨*proof*⟩

lemma *splitFace-holds-faceListAt-len*:
pre-splitFace g' v a f' vs ⟹ *minGraphProps g'* ⟹
faceListAt-len (*snd* (*snd* (*splitFace g' v a f' vs*)))
⟨*proof*⟩

lemma *splitFace-new-f12*:
assumes *pre*: *pre-splitFace g ram1 ram2 oldF newVs*
and *props*: *minGraphProps g*
and *spl*: (*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs*
shows *f12* ∉ $\mathcal{F}$ *g*
⟨*proof*⟩

lemma *splitFace-new-f12-norm*:
assumes *pre*: *pre-splitFace g ram1 ram2 oldF newVs*
and *props*: *minGraphProps g*
and *spl*: (*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs*
shows *normFace f12* ∉ *set* (*normFaces* (*faces g*))
⟨*proof*⟩

lemma *splitFace-new-f21*:
assumes *pre*: *pre-splitFace g ram1 ram2 oldF newVs*
and *props*: *minGraphProps g*
and *spl*: (*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs*
shows *f21* ∉ $\mathcal{F}$ *g*
⟨*proof*⟩

lemma *splitFace-new-f21-norm*:
assumes *pre*: *pre-splitFace g ram1 ram2 oldF newVs*
and *props*: *minGraphProps g*
and *spl*: (*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs*
shows *normFace f21* ∉ *set* (*normFaces* (*faces g*))
⟨*proof*⟩

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-f12-oldF-neq*:
pre-splitFace g ram1 ram2 oldF newVs ⟹
minGraphProps g ⟹
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF newVs* ⟹

oldF $\neq$ *f12*
⟨*proof*⟩

lemma *splitFace-f12-f21-neq-norm*:
pre-splitFace g ram1 ram2 oldF vs $\Longrightarrow$ *minGraphProps g* $\Longrightarrow$
(*f12*, *f21*, *newGraph*) = *splitFace g ram1 ram2 oldF vs* $\Longrightarrow$
normFace f12 $\neq$ *normFace f21*
⟨*proof*⟩

lemma *set-faces-splitFace*:
⟦ *minGraphProps g*; *f* $\in$ $\mathcal{F}$ *g*; *pre-splitFace g v1 v2 f vs*;
(*f1*, *f2*, *g'*) = *splitFace g v1 v2 f vs* ⟧
$\Longrightarrow$ $\mathcal{F}$ *g'* = {*f1*,*f2*} $\cup$ ($\mathcal{F}$ *g* − {*f*})
⟨*proof*⟩

declare *minGraphProps8 minGraphProps8a minGraphProps8a'* [*intro*]

lemma *splitFace-holds-facesAt-distinct*:
assumes *pre*: *pre-splitFace g v w f* [*countVertices g..<countVertices g* + *n*]
and *mgp*: *minGraphProps g*
shows *facesAt-distinct* (*snd* (*snd* (*splitFace g v w f* [*countVertices g..<countVertices g* + *n*])))
⟨*proof*⟩

lemma *splitFace-holds-facesAt-eq*:
assumes *pre-F*: *pre-splitFace g' v a f'* [*countVertices g'..<countVertices g'* + *n*]
and *mgp*: *minGraphProps g'*
and *g''*: *g''* = (*snd* (*snd* (*splitFace g' v a f'* [*countVertices g'..<countVertices g'* + *n*])))
shows *facesAt-eq g''*
⟨*proof*⟩

lemma *splitFace-holds-faces-subset*:
assumes *pre-F*: *pre-splitFace g' v a f'* [*countVertices g'..<countVertices g'* + *n*]
and *mgp*: *minGraphProps g'*
shows *faces-subset* (*snd* (*snd* (*splitFace g' v a f'* [*countVertices g'..<countVertices g'* + *n*])))
⟨*proof*⟩

lemma *splitFace-holds-edges-sym*:
assumes *pre-F*: *pre-splitFace g' v a f' ws*
and *mgp*: *minGraphProps g'*
shows *edges-sym* (*snd* (*snd* (*splitFace g' v a f' ws*)))
⟨*proof*⟩

lemma *splitFace-holds-faces-distinct*:
assumes *pre-F*: *pre-splitFace g′ v a f′* [*countVertices g′..<countVertices g′ + n*]
and *mgp*: *minGraphProps g′*
shows *faces-distinct* (*snd* (*snd* (*splitFace g′ v a f′* [*countVertices g′..<countVertices g′ + n*])))
⟨*proof*⟩

lemma *help*:
shows $xs \neq [] \Longrightarrow x \notin set\ xs \Longrightarrow x \neq hd\ xs$ **and**
$xs \neq [] \Longrightarrow x \notin set\ xs \Longrightarrow x \neq last\ xs$ **and**
$xs \neq [] \Longrightarrow x \notin set\ xs \Longrightarrow hd\ xs \neq x$ **and**
$xs \neq [] \Longrightarrow x \notin set\ xs \Longrightarrow last\ xs \neq x$
⟨*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*$| \geq 3$;
$vs = [] \longrightarrow (a,b) \notin edges\ f \wedge (b,a) \notin edges\ f\ ]\!]$
$\Longrightarrow \mathcal{E}\ f_1 \cap \mathcal{E}\ f_2 = \{\}$
⟨*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*:
assumes *precond*: *pre-splitFace* g' v a f' [*countVertices* g'..<*countVertices* g' + n]
and *min*: *minGraphProps* g'
shows *minGraphProps* (*snd* (*snd* (*splitFace* g' v a f' [*countVertices* g'..<*countVertices* g' + n])))
⟨*proof*⟩

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*⟩

abbreviation (*input*)
Edges-if :: *face* ⇒ *vertex* ⇒ *vertex* ⇒ (*vertex* × *vertex*)*set* **where**
Edges-if $f\ u\ v$ ==
if u=v *then* {} *else Edges*(u # *between* (*vertices* f) $u\ v$ @ [v])

lemma *FaceDivsionGraph-one-final-but*:
assumes *mgp*: *minGraphProps* g **and** *pre*: *pre-splitFace* $g\ u\ v\ f\ vs$
and *fdg*: $(f_1,f_2,g') =$ *splitFace* $g\ u\ v\ f\ vs$
and *nrv*: $r \neq v$
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′*:
$\bigwedge f\ v\ n\ g$.
pre-subdivFace′ $g\ f\ u\ v\ n$ *ovs* $\Longrightarrow$ *minGraphProps* $g \Longrightarrow f \in \mathcal{F}\ g \Longrightarrow$
one-final-but g (*Edges-if* $f\ u\ v$) ⟹
one-final(*subdivFace′* $g\ f\ v\ n$ *ovs*)
⟨*proof*⟩

lemma *neighbors-edges*:

minGraphProps g $\Longrightarrow$ *a* : $\mathcal{V}$ *g* $\Longrightarrow$ *b* $\in$ *set* (*neighbors g a*) = ((*a*, *b*) $\in$ *edges g*)
⟨*proof*⟩

lemma *no-self-edges*: *minGraphProps'* *g* $\Longrightarrow$ (*a*, *a*) $\notin$ *edges g* ⟨*proof*⟩

Requires only *distinct* (*vertices f*) and that *g* has no self-loops.

lemma *duplicateEdge-is-duplicateEdge-eq*:
minGraphProps g $\Longrightarrow$ *f* $\in$ $\mathcal{F}$ *g* $\Longrightarrow$ *a* $\in$ $\mathcal{V}$ *f* $\Longrightarrow$ *b* $\in$ $\mathcal{V}$ *f* $\Longrightarrow$
duplicateEdge g f a b = *is-duplicateEdge g f a b*
⟨*proof*⟩

lemma *incrIndexList-less-eq*:
incrIndexList ls m nmax $\Longrightarrow$ *Suc n* < |*ls*| $\Longrightarrow$ *ls*!*n* $\leq$ *ls*!*Suc n*
⟨*proof*⟩

lemma *incrIndexList-less*:
incrIndexList ls m nmax $\Longrightarrow$ *Suc n* < |*ls*| $\Longrightarrow$ *ls*!*n* $\neq$ *ls*!*Suc n*$\Longrightarrow$ *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*..<*Suc z*] *Final*) = *0*
⟨*proof*⟩

lemma *minVertex-zero2*: *minVertex* (*Face* (*rev* [*0*..<*Suc 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*: $f \in set\ (nonFinals\ g)$
and *v*: $v \in \mathcal{V}\ f$ **and** *e*: $e \in set\ (enumerator\ i\ |vertices\ f|\ )$
and *containsNot*: $\neg$ *containsDuplicateEdge g f v e* **and** *i*: $2 < i$
shows *pre-subdivFace g f v* (*indexToVertexList f v e*)
⟨*proof*⟩

13.10 Increasing properties of *subdivFace′*

lemma *subdivFace′-incr*:
assumes *Ptrans*: $\bigwedge x\ y\ z.\ \ Q\ x\ y \Longrightarrow P\ y\ z \Longrightarrow P\ x\ z$
and *mkFin*: $\bigwedge f\ g.\ f \in \mathcal{F}\ g \Longrightarrow \neg\ final\ f \Longrightarrow P\ g\ (makeFaceFinal\ f\ g)$
and *fdg-incr*: $\bigwedge g\ u\ v\ f\ vs.$
pre-splitFace g u v f vs $\Longrightarrow$
$Q\ g\ (snd(snd(splitFace\ g\ u\ v\ f\ vs)))$
shows
$\bigwedge f'\ v\ n\ g.$ *pre-subdivFace′ g f′ v′ v n ovl* $\Longrightarrow$
minGraphProps g $\Longrightarrow$ $f' \in \mathcal{F}\ g \Longrightarrow P\ g\ (subdivFace'\ g\ f'\ v\ n\ ovl)$
⟨*proof*⟩

lemma *next-plane0-via-subdivFace′*:
assumes *mgp*: *minGraphProps g* **and** *gg′*: $g\ [next\text{-}plane0_p]\rightarrow g'$
and *P*: $\bigwedge f\ v'\ v\ n\ g\ ovs.$ *minGraphProps g* $\Longrightarrow$ *pre-subdivFace′ g f v′ v n ovs* $\Longrightarrow$
$f \in \mathcal{F}\ g \Longrightarrow P\ g\ (subdivFace'\ g\ f\ v\ n\ ovs)$
shows $P\ g\ g'$
⟨*proof*⟩

lemma *next-plane0-incr*:
assumes *Ptrans*: $\bigwedge x\ y\ z.\ Q\ x\ y \Longrightarrow P\ y\ z \Longrightarrow P\ x\ z$
and *mkFin*: $\bigwedge f\ g.\ f \in \mathcal{F}\ g \Longrightarrow \neg\ final\ f \Longrightarrow P\ g\ (makeFaceFinal\ f\ g)$
and *fdg-incr*: $\bigwedge g\ u\ v\ f\ vs.$

pre-splitFace g u v f vs $\Longrightarrow$
Q g (*snd*(*snd*(*splitFace g u v f vs*)))
and *mgp*: *minGraphProps g* **and** *gg'*: *g* [*next-plane0*$_p$]$\rightarrow$ *g'*
shows *P g g'*
⟨*proof*⟩

13.10.1 Increasing number of faces

lemma *splitFace-incr-faces*:
pre-splitFace g u v f vs $\Longrightarrow$
finals(*snd*(*snd*(*splitFace g u v f vs*))) = *finals g* $\wedge$
|*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 $\Longrightarrow$
minGraphProps g $\Longrightarrow$ *f* $\in \mathcal{F}$ *g* $\Longrightarrow$
|*finals* (*subdivFace' g f v n ovs*)| = *Suc* |*finals g*| $\wedge$
|*nonFinals*(*subdivFace' g f v n ovs*)| $\geq$ |*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 Properties

theory *PlaneProps*
imports *Invariants*
begin

14.1 *final*

lemma *plane-final-facesAt*:
assumes *inv g final g v* : $\mathcal{V}$ *g f* $\in$ *set* (*facesAt g v*) **shows** *final f*
⟨*proof*⟩

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* **and** *v*: *v* : $\mathcal{V}$ *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* **and** *v*: *v* : $\mathcal{V}$ *g*
shows *exceptionalVertex g v* =
(| [*f* $\leftarrow$ *facesAt g v* . *5* $\leq$ |*vertices f*|] | $\neq$ *0*)
⟨*proof*⟩

lemma *not-exceptional*:
inv g $\Longrightarrow$ *final g* $\Longrightarrow$ *v* : $\mathcal{V}$ *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-plane0*$_p$ *g*)
⟨*proof*⟩

lemma *next-plane0-nonfinals*: $g\ [\textit{next-plane0}_p]\rightarrow g' \Longrightarrow \textit{nonFinals}\ g \neq []$
⟨*proof*⟩

lemma *next-plane0-ex*:
assumes *a*: $g\ [\textit{next-plane0}_p]\rightarrow g'$
shows $\exists f \in \textit{set}(\textit{nonFinals}\ g).\ \exists v \in \mathcal{V}\ f.\ \exists i \in \textit{set}([3..<\textit{Suc}(\textit{maxGon}\ p)]).$
$g' \in \textit{set}\ (\textit{generatePolygon}\ i\ v\ f\ g)$
⟨*proof*⟩

lemma *step-outside2*:
$\textit{inv}\ g \Longrightarrow g\ [\textit{next-plane0}_p]\rightarrow g' \Longrightarrow \neg\ \textit{final}\ g' \Longrightarrow |\textit{faces}\ g'| \neq 2$
⟨*proof*⟩

14.4 Increasing final faces

lemma *set-finals-splitFace*[*simp*]:
$[\![\ f \in \mathcal{F}\ g;\ \neg\ \textit{final}\ f\ ]\!] \Longrightarrow$
$\textit{set}(\textit{finals}(\textit{snd}(\textit{snd}(\textit{splitFace}\ g\ u\ v\ f\ vs)))) = \textit{set}(\textit{finals}\ g)$
⟨*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*:
$[\![\ g' \in \textit{set}\ (\textit{next-plane0}_p\ g);\ \textit{minGraphProps}\ g\ ]\!]$
$\Longrightarrow |\textit{faceListAt}\ g| \leq |\textit{faceListAt}\ g'| \wedge$
$(\forall v < |\textit{faceListAt}\ g|.\ |\textit{faceListAt}\ g\ !\ v| \leq |\textit{faceListAt}\ g'\ !\ v|\ )$

(**is** - $\Longrightarrow$ - $\Longrightarrow$ *?Q g g'*)
⟨*proof*⟩

lemma *next-plane0-incr-degree*:
⟦ $g' \in$ *set* (*next-plane0*$_p$ *g*); *minGraphProps g*; $v \in \mathcal{V}$ *g* ⟧
$\Longrightarrow$ *degree g v* $\leq$ *degree g' v*
⟨*proof*⟩

14.7 Increasing *except*

lemma *next-plane0-incr-except*:
assumes $g' \in$ *set* (*next-plane0*$_p$ *g*) *inv g v* $\in \mathcal{V}$ *g*
shows *except g v* $\leq$ *except g' v*
⟨*proof*⟩

14.8 Increasing edges

lemma *next-plane0-set-edges-subset*:
⟦ *minGraphProps g*; *g* [*next-plane0*$_p$]$\rightarrow$ *g'* ⟧ $\Longrightarrow$ *edges g* $\subseteq$ *edges g'*
⟨*proof*⟩

14.9 Increasing final vertices

declare *atLeastLessThan-iff*[*iff*]

lemma *next-plane0-incr-finV*:
⟦$g' \in$ *set* (*next-plane0*$_p$ *g*); *minGraphProps g* ⟧
$\Longrightarrow \forall v \in \mathcal{V}\ g.\ v \in \mathcal{V}\ g' \wedge$
$((\forall f \in \mathcal{F}\ g.\ v \in \mathcal{V}\ f \longrightarrow$ *final f*$) \longrightarrow$
$(\forall f \in \mathcal{F}\ g'.\ v \in \mathcal{V}\ f \longrightarrow f \in \mathcal{F}\ g))$ (**is** - $\Longrightarrow$ - $\Longrightarrow$ *?Q g g'*)
⟨*proof*⟩

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

14.10 Preservation of *facesAt* at final vertices

lemma *next-plane0-finalVertex-facesAt-eq*:
⟦$g' \in$ *set* (*next-plane0*$_p$ *g*); *inv g*; $v \in \mathcal{V}$ *g*; *finalVertex g v* ⟧
$\Longrightarrow$ *set*(*facesAt g' v*) = *set*(*facesAt g v*)
⟨*proof*⟩

lemma *next-plane0-len-filter-eq*:
assumes $g' \in$ *set* (*next-plane0*$_p$ *g*) *inv g v* $\in \mathcal{V}$ *g finalVertex g v*
shows |*filter P* (*facesAt g' v*)| = |*filter P* (*facesAt g v*)|

⟨*proof*⟩

14.11 Properties of *subdivFace′*

lemma *new-edge-subdivFace′*:
⋀*f v n g*.
pre-subdivFace′ g f u v n ovs ⟹ *minGraphProps g* ⟹ *f* ∈ *ℱ g* ⟹
subdivFace′ g f v n ovs = *makeFaceFinal f g* ∨
(∀*f′* ∈ *ℱ* (*subdivFace′ g f v n ovs*) − (*ℱ g* − {*f*}).
∃ *e* ∈ *ℰ f′*. *e* ∉ *ℰ g*)
⟨*proof*⟩

lemma *dist-edges-subdivFace′*:
pre-subdivFace′ g f u v n ovs ⟹ *minGraphProps g* ⟹ *f* ∈ *ℱ g* ⟹
subdivFace′ g f v n ovs = *makeFaceFinal f g* ∨
(∀*f′* ∈ *ℱ* (*subdivFace′ g f v n ovs*) − (*ℱ g* − {*f*}). *ℰ f′* ≠ *ℰ f*)
⟨*proof*⟩

lemma *between-last*: ⟦ *distinct*(*vertices f*); *u* ∈ *𝒱 f* ⟧ ⟹
between (*vertices f*) *u* (*last* (*verticesFrom f u*)) =
butlast(*tl*(*verticesFrom f u*))
⟨*proof*⟩

lemma *final-subdivFace′*: ⋀*f u n g*. *minGraphProps g* ⟹
pre-subdivFace′ g f r u n ovs ⟹ *f* ∈ *ℱ g* ⟹
(*ovs* = [] ⟶ *n*=*0* ∧ *u* = *last*(*verticesFrom f r*)) ⟹
∃*f′* ∈ *set*(*finals*(*subdivFace′ g f u n ovs*)) − *set*(*finals g*).
(*f*$^{-1}$ ⋅ *r*,*r*) ∈ *ℰ f′* ∧ |*vertices f′*| =
n + |*ovs*| + (*if r*=*u then 1 else* |*between* (*vertices f*) *r u*| + *2*)
⟨*proof*⟩

lemma *Seed-max-final-ex*:
∃*f*∈*set* (*finals* (*Seed p*)). |*vertices f*| = *maxGon p*
⟨*proof*⟩

lemma *max-face-ex*: **assumes** *a*: *Seed*$_p$ [*next-plane0*$_p$]⟶∗ *g*
shows ∃*f* ∈ *set* (*finals g*). |*vertices f*| = *maxGon p*
⟨*proof*⟩

end

15 Summation Over Lists

theory *ListSum*
imports *ListAux*
begin

primrec *ListSum* :: *'b list* ⇒ (*'b* ⇒ *'a*::*comm-monoid-add*) ⇒ *'a*::*comm-monoid-add*
where
  ListSum [] *f* = *0*
| *ListSum* (*l*#*ls*) *f* = *f l* + *ListSum ls f*

syntax -*ListSum* :: *idt* ⇒ *'b list* ⇒ (*'a*::*comm-monoid-add*) ⇒
  (*'a*::*comm-monoid-add*) (∑ -∈- - [*0*, *0*, *10*] *10*)
translations ∑ *x*∈*xs* *f* == *CONST ListSum xs* (λ*x*. *f*)

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-sum*:
 distinct xs ⟹ *ListSum xs f* = *sum 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-disj-union*:
distinct A $\Longrightarrow$ *distinct B* $\Longrightarrow$ *distinct C* $\Longrightarrow$
set C = *set A* $\cup$ *set B* $\Longrightarrow$
set A $\cap$ *set B* = {} $\Longrightarrow$
$(\sum a \in C\ (f\ a)) = (\sum a \in A\ f\ a) + (\sum a \in B\ (f\ a::nat))$
⟨*proof*⟩

lemma *listsum-const*[*simp*]:
$(\sum x \in xs\ k) = length\ xs * k$
⟨*proof*⟩

lemma *ListSum-add*:
$(\sum x \in V\ f\ x) + (\sum x \in V\ g\ x) = (\sum x \in V\ (f\ x + (g\ x::nat)))$
⟨*proof*⟩

lemma *ListSum-le*:
$(\bigwedge v.\ v \in set\ V \Longrightarrow f\ v \leq g\ v) \Longrightarrow (\sum v \in V\ f\ v) \leq (\sum v \in V\ (g\ v::nat))$
⟨*proof*⟩

lemma *ListSum1-bound*:
$a \in set\ F \Longrightarrow (d\ a::nat) \leq (\sum f \in F\ d\ f)$
⟨*proof*⟩

end

16 Tameness

theory *Tame*
imports *Graph ListSum*
begin

16.1 Constants

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

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

a ≡ *6295*

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

b *p q* ≡ *if p = 0 ∧ q = 3 then 6177*
else if p = 0 ∧ q = 4 then 9696
else if p = 1 ∧ q = 2 then 6557
else if p = 1 ∧ q = 3 then 6176

else if $p = 2 \wedge q = 1$ *then* 7967
else if $p = 2 \wedge q = 2$ *then* 4116
else if $p = 2 \wedge q = 3$ *then* 12846
else if $p = 3 \wedge q = 1$ *then* 3106
else if $p = 3 \wedge q = 2$ *then* 8165
else if $p = 4 \wedge q = 0$ *then* 3466
else if $p = 4 \wedge q = 1$ *then* 3655
else if $p = 5 \wedge q = 0$ *then* 395
else if $p = 5 \wedge q = 1$ *then* 11354
else if $p = 6 \wedge q = 0$ *then* 6854
else if $p = 7 \wedge q = 0$ *then* 14493
else squanderTarget

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

d $n \equiv$ *if* $n = 3$ *then* 0
else if $n = 4$ *then* 2058
else if $n = 5$ *then* 4819
else if $n = 6$ *then* 7120
else squanderTarget

16.2 Separated sets of vertices

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

2. No two vertices in V are adjacent:

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

3. No two vertices lie on a common quadrilateral:

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

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

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

16.3 Admissible weight assignments

A weight assignment w :: *face* ⇒ *nat* assigns a natural number to every face. We formalize the admissibility requirements as follows:

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

definition $admissible_2$ :: (*face* ⇒ *nat*) ⇒ *graph* ⇒ *bool* **where**

admissible$_2$ *w g* ≡
∀ *v* ∈ $\mathcal{V}$ *g. except g v = 0* ⟶ b (*tri g v*) (*quad g v*) ≤ ($\sum_{f \in facesAt\ g\ v}$ *w f*)

definition *admissible*$_3$ :: (*face* ⇒ *nat*) ⇒ *graph* ⇒ *bool* **where**
admissible$_3$ *w g* ≡
∀ *v* ∈ $\mathcal{V}$ *g. vertextype g v* = (5,0,1) ⟶ ($\sum_{f \in filter\ triangle\ (facesAt\ g\ v)}$ *w*(*f*)) ≥ a

Finally we define admissibility of weights functions.

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

16.4 Tameness

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

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

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

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

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

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

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

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

Finally we define the notion of tameness.

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

theory *Plane1Props*
imports *Plane1 PlaneProps Tame*
begin

lemma *next-plane-subset*:

$\forall f \in \mathcal{F}\ g.\ vertices\ f \neq [] \Longrightarrow$
$set\ (next\text{-}plane_p\ g) \subseteq set\ (next\text{-}plane0_p\ g)$
⟨*proof*⟩

lemma *mgp-next-plane0-if-next-plane*:
$minGraphProps\ g \Longrightarrow g\ [next\text{-}plane_p]\rightarrow g' \Longrightarrow g\ [next\text{-}plane0_p]\rightarrow g'$
⟨*proof*⟩

lemma *inv-inv-next-plane*: *invariant inv* $next\text{-}plane_p$
⟨*proof*⟩

end

17 Enumeration of Tame Plane Graphs

theory *Generator*
imports *Plane1 Tame*
begin

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

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

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

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

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

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

definition *ExcessTable* :: *graph* ⇒ *vertex list* ⇒ (*vertex* × *nat*) *list* **where**
ExcessTable g vs ≡

$[(v,\ ExcessAt\ g\ v).\ v \leftarrow [v \leftarrow vs.\ 0 < ExcessAt\ g\ v\ ]]$

Implementation:

lemma [*code*]:
ExcessTable g =
List.map-filter (λv. *let e* = *ExcessAt g v in if 0* < *e then Some* (*v*, *e*) *else None*)
⟨*proof*⟩

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

Implementation:

lemma [*code*]: *deleteAround g v ps* =
(*let vs* = (λf. *let n* = $f{\cdot}v$
in if |*vertices f*| = *4 then* [*n*, $f{\cdot}n$] *else* [*n*])
in removeKeyList (*concat*(*map vs* (*facesAt g v*))) *ps*)
⟨*proof*⟩

lemma *length-deleteAround*: *length* (*deleteAround g v ps*) ≤ *length ps*
⟨*proof*⟩

function *ExcessNotAtRec* :: (*nat*, *nat*) *table* ⇒ *graph* ⇒ *nat* **where**
ExcessNotAtRec [] = (λg. *0*)
| *ExcessNotAtRec* ((*x*, *y*)#*ps*) = (λg. *max* (*ExcessNotAtRec ps g*)
(*y* + *ExcessNotAtRec* (*deleteAround g x ps*) *g*))
⟨*proof*⟩
termination ⟨*proof*⟩

definition *ExcessNotAt* :: *graph* ⇒ *vertex option* ⇒ *nat* **where**
ExcessNotAt g v-opt ≡
let ps = *ExcessTable g* (*vertices g*) *in*
case v-opt of None ⇒ *ExcessNotAtRec ps g*
| *Some v* ⇒ *ExcessNotAtRec* (*deleteAround g v ps*) *g*

definition *squanderLowerBound* :: *graph* ⇒ *nat* **where**
squanderLowerBound g ≡ *faceSquanderLowerBound g* + *ExcessNotAt g None*

definition *is-tame13a* :: *graph* ⇒ *bool* **where**
is-tame13a g ≡ *squanderLowerBound g* < *squanderTarget*

definition *notame* :: *graph* ⇒ *bool* **where**

notame g ≡ ¬ (*tame10ub g* ∧ *tame11b g*)

definition *notame7* :: *graph* ⇒ *bool* **where**
notame7 g ≡ ¬ (*tame10ub g* ∧ *tame11b g* ∧ *is-tame13a g*)

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

definition *polysizes* :: *nat* ⇒ *graph* ⇒ *nat list* **where**
polysizes p g ≡
  let lb = *squanderLowerBound g in*
  [*n* ← [*3* ..< *Suc*(*maxGon p*)]. *lb* + d *n* < *squanderTarget*]

definition *next-tame0* :: *nat* ⇒ *graph* ⇒ *graph list* (*next*′*-tame0*_) **where**
next-tame0$_p$ *g* ≡
  let fs = *nonFinals g in*
  if fs = [] *then* []
  else let f = *minimalFace fs*; *v* = *minimalVertex g f*
    in ⨆ *i* ∈ *polysizes p g generatePolygonTame i v f g*

Extensionally, *next-tame0* is just *filter P* ∘ *next-plane*$_p$ for some suitable *P*. But efficiency suffers considerably if we first create many graphs and then filter out the ones not in *polysizes*.

end

18 Tame Properties

theory *TameProps*
imports *Tame RTranCl*
begin

lemma *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* ∈ 𝒱 *g* ⟧ ⟹ *tri g v* + *quad g v* ≤ *7*

⟨*proof*⟩

lemma *filter-tame-succs*:
assumes *invP*: *invariant P succs* **and** *fin*: $\bigwedge g.$ *final* $g \Longrightarrow$ *succs* $g = []$
and *ok-untame*: $\bigwedge g.$ $P\ g \Longrightarrow \neg$ *ok* $g \Longrightarrow$ *final* $g \wedge \neg$ *tame* g
and *gg'*: g [*succs*]$\rightarrow *$ g'
shows $P\ g \Longrightarrow$ *final* $g' \Longrightarrow$ *tame* $g' \Longrightarrow g$ [*filter ok* $\circ$ *succs*]$\rightarrow *$ g'
⟨*proof*⟩

definition *untame* :: (*graph* $\Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**
untame $P \equiv \forall g.$ *final* $g \wedge P\ g \longrightarrow \neg$ *tame* g

lemma *filterout-untame-succs*:
assumes *invP*: *invariant P f* **and** *invPU*: *invariant* $(\lambda g.\ P\ g \wedge\ U\ g)\ f$
and *untame*: *untame*$(\lambda g.\ P\ g \wedge U\ g)$
and *new-untame*: $\bigwedge g\ g'.\ [\![\ P\ g;\ g' \in set(f\ g);\ g' \notin set(f'\ g)\ ]\!] \Longrightarrow U\ g'$
and *gg'*: g [f]$\rightarrow *$ g'
shows $P\ g \Longrightarrow$ *final* $g' \Longrightarrow$ *tame* $g' \Longrightarrow g$ [f']$\rightarrow *$ g'
⟨*proof*⟩

end

19 Neglectable Final Graphs

theory *TameEnum*
imports *Generator*
begin

definition *is-tame* :: *graph* $\Rightarrow$ *bool* **where**
is-tame $g\ \equiv\ tame10\ g \wedge tame11a\ g \wedge tame12o\ g \wedge is\text{-}tame13a\ g$

definition *next-tame* :: *nat* $\Rightarrow$ *graph* $\Rightarrow$ *graph list* (*next'-tame*$_-$) **where**
next-tame$_p \equiv$ *filter* $(\lambda g.\ \neg$ *final* $g \vee$ *is-tame* $g) \circ$ *next-tame0*$_p$

definition *TameEnumP* :: *nat* $\Rightarrow$ *graph set* (*TameEnum*$_-$) **where**
TameEnum$_p \equiv \{g.\ Seed_p$ [*next-tame*$_p$]$\rightarrow *$ $g \wedge$ *final* $g\}$

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

end

20 Properties of Lower Bound Machinery

theory *ScoreProps*
imports *ListSum TameEnum PlaneProps TameProps*
begin

lemma *deleteAround-empty*[*simp*]: *deleteAround g a* [] = []
⟨*proof*⟩

lemma *deleteAroundCons*:
  deleteAround g a (*p*#*ps*) =
    (*if fst p* ∈ {*v*. ∃*f* ∈ *set* (*facesAt g a*).
            (*length* (*vertices f*) = 4) ∧ *v* ∈ {*f* · *a*, *f* · (*f* · *a*)}
          ∨ (*length* (*vertices f*) ≠ 4) ∧ (*v* = *f* · *a*)}
     then deleteAround g a ps
     else p#*deleteAround g a ps*)
⟨*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*⟩

definition *deleteAround′* :: *graph* ⇒ *vertex* ⇒ (*vertex* × *nat*) *list* ⇒
    (*vertex* × *nat*) *list* **where**
  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 $\Longrightarrow$ *a* : $\mathcal{V}$ *g* $\Longrightarrow$ *f* $\in$ *set* (*facesAt g a*) $\Longrightarrow$
(f^{-1} · *a*, *b*) $\notin$ *set* (*deleteAround g a ps*)
⟨*proof*⟩

lemma *deleteAround-separated*:
assumes *mgp*: *minGraphProps g* **and** *fin*: *final g* **and** *ag*: *a* : $\mathcal{V}$ *g* **and** *4*: |*vertices f*| $\leq$ *4*
and *f*: *f* $\in$ *set*(*facesAt g a*)
shows $\mathcal{V}$ *f* $\cap$ *set* [*fst p*. *p* $\leftarrow$ *deleteAround g a ps*] $\subseteq$ {*a*} (**is** *?A*)
⟨*proof*⟩

lemma [*iff*]: *separated g* {}
⟨*proof*⟩

lemma *separated-insert*:
assumes *mgp*: *minGraphProps g* **and** *a*: *a* $\in$ $\mathcal{V}$ *g*
and *Vg*: *V* $\subseteq$ $\mathcal{V}$ *g*
and *ps*: *separated g V*
and *s2*: ($\bigwedge$*f*. *f* $\in$ *set* (*facesAt g a*) $\Longrightarrow$ *f* · *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 *separated g* (*insert a V*)
⟨*proof*⟩

function *ExcessNotAtRecList* :: (*vertex*, *nat*) *table* ⇒ *graph* ⇒ *vertex list* **where**
ExcessNotAtRecList [] = (λ*g*. [])
| *ExcessNotAtRecList* ((*x*, *y*) # *ps*) = (λ*g*.
let l1 = *ExcessNotAtRecList ps g*;
l2 = *ExcessNotAtRecList* (*deleteAround g x ps*) *g in*
if ExcessNotAtRec ps g
≤ *y* + *ExcessNotAtRec* (*deleteAround g x ps*) *g*
then x # *l2 else l1*)
⟨*proof*⟩
termination ⟨*proof*⟩

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

lemma *ListSum-ExcessNotAtRecList*:
isTable E vs ps $\Longrightarrow$ *ExcessNotAtRec ps g*
= ($\sum$ *p* $\in$ *ExcessNotAtRecList ps g E p*) (**is** *?T ps* $\Longrightarrow$ *?P ps*)
⟨*proof*⟩

lemma *ExcessNotAtRecList-subset*:
set (*ExcessNotAtRecList ps g*) ⊆ *set* [*fst p*. *p* ← *ps*] (**is** *?P ps*)
⟨*proof*⟩

lemma *separated-ExcessNotAtRecList*:
minGraphProps g ⟹ *final g* ⟹ *isTable E* (*vertices g*) *ps* ⟹
separated 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*⟩

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

definition *ExcessTable′* :: *graph* ⇒ *vertex list* ⇒ (*vertex* × *nat*) *list* **where**
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 ⟹ *final g* ⟹
∃ *V*. *ExcessNotAt g None*
= (∑ *v* ∈ *V* *ExcessAt g v*)
∧ *separated g* (*set V*) ∧ *set V* ⊆ *set* (*vertices g*)
∧ *distinct V*
⟨*proof*⟩

lemma *excess-eq*:
assumes *7*: *t* + *q* ≤ *7*
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* ∈ *set*(*vertices g*) ⟧ ⟹
ExcessAt g v + (*tri g v*) ∗ d *3* + (*quad g v*) ∗ d *4*
= b (*tri g v*) (*quad g v*)
⟨*proof*⟩

separating

definition *separating* :: *'a set* ⇒ (*'a* ⇒ *'b set*) ⇒ *bool* **where**
separating V F ≡
(∀ *v1* ∈ *V*. ∀ *v2* ∈ *V*. *v1* ≠ *v2* ⟶ *F v1* ∩ *F v2* = {})

lemma *separating-insert1*:
separating (*insert a V*) *F* ⟹ *separating V F*
⟨*proof*⟩

lemma *separating-insert2*:
separating (*insert a V*) *F* ⟹ *a* ∉ *V* ⟹ *v* ∈ *V* ⟹
F a ∩ *F v* = {}
⟨*proof*⟩

lemma *sum-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 *separated-separating*:
assumes *Vg*: *set V* $\subseteq \mathcal{V}$ *g*
and *pS*: *separated g* (*set V*)
and *noex*: $\forall f{\in}P.$ |*vertices f*| $\leq$ *4*
shows *separating* (*set V*) ($\lambda v.$ *set* (*facesAt g v*) $\cap$ *P*)
⟨*proof*⟩

lemma *ListSum-V-F-eq-ListSum-F*:
assumes *pl*: *inv g*
and *pS*: *separated g* (*set V*) **and** *dist*: *distinct V*
and *V-subset*: *set V* $\subseteq$ *set* (*vertices g*)
and *noex*: $\forall f \in$ *Collect P*. |*vertices f*| $\leq$ *4*
shows ($\sum v \in V \sum f \in$ *filter P* (*facesAt g v*) (*w*::*face* $\Rightarrow$ *nat*) *f*)
= ($\sum f \in$ [*f*←*faces g* . $\exists v \in$ *set V*. *f* $\in$ *set* (*facesAt g v*) $\cap$ *Collect P*] *w f*)
⟨*proof*⟩

lemma *separated-disj-Union2*:
assumes *pl*: *inv g* **and** *fin*: *final g* **and** *ne*: *noExceptionals g* (*set V*)
and *pS*: *separated g* (*set V*) **and** *dist*: *distinct V*
and *V-subset*: *set V* $\subseteq$ *set* (*vertices g*)
shows ($\sum v \in V \sum f \in$ *facesAt g v* (*w*::*face* $\Rightarrow$ *nat*) *f*)
= ($\sum f \in$ [*f*←*faces g* . $\exists v \in$ *set V*. *f* $\in$ *set* (*facesAt g v*)] *w f*)
⟨*proof*⟩

lemma *squanderFace-distr2*: *inv g* $\Longrightarrow$ *final g* $\Longrightarrow$ *noExceptionals g* (*set V*) $\Longrightarrow$
separated g (*set V*) $\Longrightarrow$ *distinct V* $\Longrightarrow$ *set V* $\subseteq$ *set* (*vertices g*) $\Longrightarrow$
($\sum f \in$ [*f*←*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 *separated-subset*:
V1 $\subseteq$ *V2* $\Longrightarrow$ *separated g V2* $\Longrightarrow$ *separated g V1*
⟨*proof*⟩

end

21 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*⟩

22 Properties of Tame Graph Enumeration (1)

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

lemma *genPolyTame-spec*:
$generatePolygonTame\ n\ v\ f\ g = [g' \leftarrow generatePolygon\ n\ v\ f\ g\ .\ \neg\ notame\ g']$
⟨*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*⟩

definition *close* :: *graph* ⇒ *vertex* ⇒ *vertex* ⇒ *bool* **where**
$close\ g\ u\ v \equiv$
$\exists f \in set(facesAt\ g\ u).\ if\ |vertices\ f| = 4\ then\ v = f \cdot u \vee v = f \cdot (f \cdot u)$
$else\ v = f \cdot u$

lemma *delAround-def*: *deleteAround g u ps* = [*p* ← *ps*. ¬ *close g u* (*fst p*)]
⟨*proof*⟩

lemma *close-sym*: **assumes** *mgp*: *minGraphProps g* **and** *ug*: *u* : $\mathcal{V}$ *g* **and** *cl*: *close g u v*
shows *close g v u*
⟨*proof*⟩

lemma *sep-conv*:
assumes *mgp*: *minGraphProps g* **and** *V* ⊆ $\mathcal{V}$ *g*
shows *separated g V* = (∀ *u*∈*V*.∀ *v*∈*V*. *u* ≠ *v* ⟶ ¬ *close g u v*) (**is** *?P* = *?Q*)
⟨*proof*⟩

lemma *sep-ne*: ∃ *P* ⊆ *M*. *separated g* (*fst* ‘ *P*)
⟨*proof*⟩

lemma *ExcessNotAtRec-conv-Max*:
assumes *mgp*: *minGraphProps g*
shows *set*(*map fst ps*) ⊆ $\mathcal{V}$ *g* ⟹ *distinct*(*map fst ps*) ⟹
ExcessNotAtRec ps g =
Max{ ∑ *p*∈*P*. *snd p* |*P*. *P* ⊆ *set ps* ∧ *separated g* (*fst* ‘ *P*)}
(**is** - ⟹ - ⟹ - = *Max*(*?M ps*) **is** - ⟹ - ⟹ - = *Max*{- |*P*. *?S ps P*})
⟨*proof*⟩

lemma *dist-ExcessTab*: *distinct* (*map fst* (*ExcessTable g* (*vertices g*)))
⟨*proof*⟩

lemma *mono-ExcessTab*: ⟦*g*′ ∈ *set* (*next-plane0*$_p$ *g*); *inv g* ⟧ ⟹
set(*ExcessTable g* (*vertices g*)) ⊆ *set*(*ExcessTable g*′ (*vertices g*′))
⟨*proof*⟩

lemma *close-antimono*:
⟦*g*′ ∈ *set* (*next-plane0*$_p$ *g*); *inv g*; *u* ∈ $\mathcal{V}$ *g*; *finalVertex g u* ⟧ ⟹
close g′ *u v* ⟹ *close g u v*
⟨*proof*⟩

lemma *ExcessTab-final*:
p ∈ *set*(*ExcessTable g* (*vertices g*)) ⟹ *finalVertex g* (*fst p*)
⟨*proof*⟩

lemma *ExcessTab-vertex*:

$p \in set(ExcessTable\ g\ (vertices\ g)) \Longrightarrow fst\ p \in \mathcal{V}\ g$
⟨*proof*⟩

lemma *fst-set-ExcessTable-subset*:
$fst\ `\ set\ (ExcessTable\ g\ (vertices\ g)) \subseteq \mathcal{V}\ g$
⟨*proof*⟩

lemma *next-plane0-incr-ExcessNotAt*:
$[\![g' \in set\ (next\text{-}plane0_p\ g);\ inv\ g\ ]\!] \Longrightarrow$
$ExcessNotAt\ g\ None \leq ExcessNotAt\ g'\ None$
⟨*proof*⟩

lemma *next-plane0-incr-squander-lb*:
$[\![g' \in set\ (next\text{-}plane0_p\ g);\ inv\ g\ ]\!] \Longrightarrow$
$squanderLowerBound\ g \leq squanderLowerBound\ g'$
⟨*proof*⟩

lemma *inv-notame*:
$[\![g' \in set\ (next\text{-}plane0_p\ g);\ inv\ g;\ notame7\ g]\!]$
$\Longrightarrow notame7\ g'$
⟨*proof*⟩

lemma *inv-inv-notame*:
$invariant(\lambda g.\ inv\ g \wedge notame7\ g)\ next\text{-}plane_p$
⟨*proof*⟩

lemma *untame-notame*:
$untame\ (\lambda g.\ inv\ g \wedge notame7\ g)$
⟨*proof*⟩

lemma *polysizes-tame*:
$[\![\ g' \in set\ (generatePolygon\ n\ v\ f\ g);\ inv\ g;\ f \in set(nonFinals\ g);$
$v \in \mathcal{V}\ f;\ 3 \leq n;\ n < 4{+}p;\ n \notin set(polysizes\ p\ g)\ ]\!]$
$\Longrightarrow notame7\ g'$
⟨*proof*⟩

lemma *genPolyTame-notame*:
$[\![\ g' \in set\ (generatePolygon\ n\ v\ f\ g);\ g' \notin set\ (generatePolygonTame\ n\ v\ f\ g);$
$inv\ g;\ 3 \leq n\ ]\!]$
$\Longrightarrow notame7\ g'$
⟨*proof*⟩

declare *upt-Suc*[*simp del*]
lemma *excess-notame*:

⟦ *inv g*; *g′* ∈ *set* (*next-plane*$_p$ *g*); *g′* ∉ *set* (*next-tame0 p g*) ⟧
⟹ *notame7 g′*
⟨*proof*⟩
declare *upt-Suc*[*simp*]

lemma *next-tame0-comp*: ⟦ *Seed*$_p$ [*next-plane p*]→∗ *g*; *final g*; *tame g* ⟧
⟹ *Seed*$_p$ [*next-tame0 p*]→∗ *g*
⟨*proof*⟩

lemma *inv-inv-next-tame0*: *invariant inv* (*next-tame0 p*)
⟨*proof*⟩

lemma *inv-inv-next-tame*: *invariant inv next-tame*$_p$
⟨*proof*⟩

lemma *mgp-TameEnum*: *g* ∈ *TameEnum*$_p$ ⟹ *minGraphProps g*
⟨*proof*⟩

end

23 Properties of Tame Graph Enumeration (2)

theory *TameEnumProps*
imports *GeneratorProps*
begin

Completeness of filter for final graphs.

lemma *untame-negFin*:
assumes *pl*: *inv g* **and** *fin*: *final g* **and** *tame*: *tame g*
shows *is-tame g*
⟨*proof*⟩

lemma *next-tame-comp*:
⟦ *tame g*; *final g*; *Seed*$_p$ [*next-tame0 p*]→∗ *g* ⟧
⟹ *Seed*$_p$ [*next-tame*$_p$]→∗ *g*
⟨*proof*⟩

end
theory *Worklist*
imports *HOL−Library.While-Combinator RTranCl Quasi-Order*
begin

definition
worklist-aux :: (*′s* ⇒ *′a* ⇒ *′a list*) ⇒ (*′a* ⇒ *′s* ⇒ *′s*)

$\Rightarrow$ *'a list* $*$ *'s* $\Rightarrow$ (*'a list* $*$ *'s*)*option*
where
worklist-aux succs f =
while-option
(λ(*ws*,*s*). *ws* $\neq$ []
(λ(*ws*,*s*). *case ws of x*#*ws'* $\Rightarrow$ (*succs s x* @ *ws'*, *f x s*))

definition *worklist* :: (*'s* $\Rightarrow$ *'a* $\Rightarrow$ *'a list*) $\Rightarrow$ (*'a* $\Rightarrow$ *'s* $\Rightarrow$ *'s*)
$\Rightarrow$ *'a list* $\Rightarrow$ *'s* $\Rightarrow$ *'s option* **where**
worklist succs f ws s =
(*case worklist-aux succs f* (*ws*,*s*) *of*
None $\Rightarrow$ *None* | *Some*(*ws*,*s*) $\Rightarrow$ *Some s*)

lemma *worklist-aux-Nil*: *worklist-aux succs f* ([],*s*) = *Some*([],*s*)
⟨*proof*⟩

lemma *worklist-aux-Cons*:
worklist-aux succs f (*x*#*ws'*,*s*) = *worklist-aux succs f* (*succs s x* @ *ws'*, *f x s*)
⟨*proof*⟩

lemma *worklist-aux-unfold*[*code*]:
worklist-aux succs f (*ws*,*s*) =
(*case ws of* [] $\Rightarrow$ *Some*([],*s*)
| *x*#*ws'* $\Rightarrow$ *worklist-aux succs f* (*succs s x* @ *ws'*, *f x s*))
⟨*proof*⟩

definition
worklist-tree-aux :: (*'a* $\Rightarrow$ *'a list*) $\Rightarrow$ (*'a* $\Rightarrow$ *'s* $\Rightarrow$ *'s*)
$\Rightarrow$ *'a list* $*$ *'s* $\Rightarrow$ (*'a list* $*$ *'s*)*option*
where
worklist-tree-aux succs = *worklist-aux* (λ*s*. *succs*)

lemma *worklist-tree-aux-unfold*[*code*]:
worklist-tree-aux succs f (*ws*,*s*) =
(*case ws of* [] $\Rightarrow$ *Some*([],*s*) |
x#*ws'* $\Rightarrow$ *worklist-tree-aux succs f* (*succs x* @ *ws'*, *f x s*))
⟨*proof*⟩

abbreviation *Rel* :: (*'a* $\Rightarrow$ *'a list*) $\Rightarrow$ (*'a* $*$ *'a*)*set* **where**
Rel f == {(*x*,*y*). *y* : *set*(*f x*)}

lemma *Image-Rel-set*:
(*Rel succs*)* `` *set*(*succs x*) = (*Rel succs*)$^+$ `` {*x*}
⟨*proof*⟩

lemma *RTranCl-conv*:
g [*succs*]→* *h* ⟷ (*g*,*h*) : ((*Rel succs*)*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *worklist-end-empty*:
worklist-aux succs f (ws,s) = Some(ws',s') ⟹ ws' = []
⟨*proof*⟩

theorem *worklist-tree-aux-Some-foldl*:
assumes *worklist-tree-aux succs f (ws,s) = Some(ws',s')*
shows *∃ rs. set rs = ((Rel succs)*) `` (set ws) ∧*
s' = foldl (λs x. f x s) s rs
⟨*proof*⟩

definition *worklist-tree succs f ws s =*
(case worklist-tree-aux succs f (ws,s) of
None ⇒ None | Some(ws,s) ⇒ Some s)

theorem *worklist-tree-Some-foldl*:
worklist-tree succs f ws s = Some s' ⟹
∃ rs. set rs = ((Rel succs)) `` (set ws) ∧*
s' = foldl (λs x. f x s) s rs
⟨*proof*⟩

lemma *invariant-succs*:
assumes *invariant I succs*
and *∀ x∈S. I x*
shows *∀ x ∈ (Rel succs)* `` S. I x*
⟨*proof*⟩

lemma *worklist-tree-aux-rule*:
assumes *worklist-tree-aux succs f (ws,s) = Some(ws',s')*
and *invariant I succs*
and *∀ x ∈ set ws. I x*
and *⋀s. P [] s s*
and *⋀r x ws s. I x ⟹ ∀ x ∈ set ws. I x ⟹ P ws (f x s) r ⟹ P (x#ws) s r*
shows *∃ rs. set rs = ((Rel succs)*) `` (set ws) ∧ P rs s s'*
⟨*proof*⟩

lemma *worklist-tree-aux-rule2*:
assumes *worklist-tree-aux succs f (ws,s) = Some(ws',s')*
and *invariant I succs*
and *∀ x ∈ set ws. I x*
and *S s* **and** *⋀x s. I x ⟹ S s ⟹ S(f x s)*
and *⋀s. P [] s s*
and *⋀r x ws s. I x ⟹ ∀ x ∈ set ws. I x ⟹ S s*
⟹ P ws (f x s) r ⟹ P (x#ws) s r
shows *∃ rs. set rs = ((Rel succs)*) `` (set ws) ∧ P rs s s'*
⟨*proof*⟩

lemma *worklist-tree-rule*:
assumes *worklist-tree succs f ws s = Some(s')*

and *invariant I succs*
and $\forall\, x \in set\ ws.\ I\ x$
and $\bigwedge s.\ P\ []\ s\ s$
and $\bigwedge r\ x\ ws\ s.\ I\ x \Longrightarrow \forall\, x \in set\ ws.\ I\ x \Longrightarrow P\ ws\ (f\ x\ s)\ r \Longrightarrow P\ (x\#ws)\ s\ r$
shows $\exists\, rs.\ set\ rs = ((Rel\ succs)^*)\ ``\ (set\ ws) \wedge P\ rs\ s\ s'$
⟨*proof*⟩

lemma *worklist-tree-rule2*:
assumes *worklist-tree succs f ws s* = *Some*(s')
and *invariant I succs*
and $\forall\, x \in set\ ws.\ I\ x$
and $S\ s$ **and** $\bigwedge x\ s.\ I\ x \Longrightarrow S\ s \Longrightarrow S(f\ x\ s)$
and $\bigwedge s.\ P\ []\ s\ s$
and $\bigwedge r\ x\ ws\ s.\ I\ x \Longrightarrow \forall\, x \in set\ ws.\ I\ x \Longrightarrow S\ s$
$\Longrightarrow P\ ws\ (f\ x\ s)\ r \Longrightarrow P\ (x\#ws)\ s\ r$
shows $\exists\, rs.\ set\ rs = ((Rel\ succs)^*)\ ``\ (set\ ws) \wedge P\ rs\ s\ s'$
⟨*proof*⟩

lemma *worklist-tree-aux-state-inv*:
assumes *worklist-tree-aux succs f* (*ws*,*s*) = *Some*(*ws*′,*s*′)
and $I\ s$
and $\bigwedge x\ s.\ I\ s \Longrightarrow I(f\ x\ s)$
shows $I\ s'$
⟨*proof*⟩

lemma *worklist-tree-state-inv*:
worklist-tree succs f ws s = *Some*(s')
$\Longrightarrow I\ s \Longrightarrow (\bigwedge x\ s.\ I\ s \Longrightarrow I(f\ x\ s)) \Longrightarrow I\ s'$
⟨*proof*⟩

locale *set-modulo* = *quasi-order* +
fixes *empty* :: $'s$
and *insert-mod* :: $'a \Rightarrow 's \Rightarrow 's$
and *set-of* :: $'s \Rightarrow 'a\ set$
and I :: $'a \Rightarrow bool$
and S :: $'s \Rightarrow bool$
assumes *set-of-empty*: *set-of empty* = {}
and *set-of-insert-mod*: $I\ x \Longrightarrow S\ s \wedge (\forall\, x \in set\text{-}of\ s.\ I\ x)$
$\Longrightarrow$
set-of(*insert-mod x s*) = *insert x* (*set-of s*) ∨
$(\exists\, y \in set\text{-}of\ s.\ x \preceq y) \wedge set\text{-}of\ (insert\text{-}mod\ x\ s) = set\text{-}of\ s$
and *S-empty*: *S empty*
and *S-insert-mod*: $S\ s \Longrightarrow S\ (insert\text{-}mod\ x\ s)$
begin

definition *insert-mod2* :: $('b \Rightarrow bool) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'b \Rightarrow 's \Rightarrow 's$ **where**
insert-mod2 P f x s = (*if P x then insert-mod* (*f x*) *s else s*)

definition $SI\ s = (S\ s \wedge (\forall x \in \textit{set-of}\ s.\ I\ x))$

lemma *SI-empty*: $SI\ \textit{empty}$
⟨*proof*⟩

lemma *SI-insert-mod*:
$I\ x \Longrightarrow SI\ s \Longrightarrow SI\ (\textit{insert-mod}\ x\ s)$
⟨*proof*⟩

lemma *SI-insert-mod2*: $(\bigwedge x.\ \textit{inv0}\ x \Longrightarrow I\ (f\ x)) \Longrightarrow$
$\textit{inv0}\ x \Longrightarrow SI\ s \Longrightarrow SI\ (\textit{insert-mod2}\ P\ f\ x\ s)$
⟨*proof*⟩

definition *worklist-tree-coll-aux* ::
$('b \Rightarrow 'b\ \textit{list}) \Rightarrow ('b \Rightarrow \textit{bool}) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'b\ \textit{list} \Rightarrow 's \Rightarrow 's\ \textit{option}$
where
$\textit{worklist-tree-coll-aux}\ \textit{succs}\ P\ f = \textit{worklist-tree}\ \textit{succs}\ (\textit{insert-mod2}\ P\ f)$

definition *worklist-tree-coll* ::
$('b \Rightarrow 'b\ \textit{list}) \Rightarrow ('b \Rightarrow \textit{bool}) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'b\ \textit{list} \Rightarrow 's\ \textit{option}$
where
$\textit{worklist-tree-coll}\ \textit{succs}\ P\ f\ \textit{ws} = \textit{worklist-tree-coll-aux}\ \textit{succs}\ P\ f\ \textit{ws}\ \textit{empty}$

lemma *worklist-tree-coll-aux-equiv*:
assumes $\textit{worklist-tree-coll-aux}\ \textit{succs}\ P\ f\ \textit{ws}\ s = \textit{Some}\ s'$
and $\textit{invariant}\ \textit{inv0}\ \textit{succs}$
and $\forall x \in \textit{set ws}.\ \textit{inv0}\ x$
and $\bigwedge x.\ \textit{inv0}\ x \Longrightarrow I(f\ x)$
and $SI\ s$
shows $\textit{set-of}\ s' =_{\preceq}$
$f\ `\ \{x : (\textit{Rel succs})^*\ ``\ (\textit{set ws}).\ P\ x\} \cup \textit{set-of}\ s$
⟨*proof*⟩

lemma *worklist-tree-coll-equiv*:
$\textit{worklist-tree-coll}\ \textit{succs}\ P\ f\ \textit{ws} = \textit{Some}\ s' \Longrightarrow \textit{invariant}\ \textit{inv0}\ \textit{succs}$
$\Longrightarrow \forall x \in \textit{set ws}.\ \textit{inv0}\ x \Longrightarrow (\bigwedge x.\ \textit{inv0}\ x \Longrightarrow I(f\ x))$
$\Longrightarrow \textit{set-of}\ s' =_{\preceq} f\ `\ \{x : (\textit{Rel succs})^*\ ``\ (\textit{set ws}).\ P\ x\}$
⟨*proof*⟩

lemma *worklist-tree-coll-aux-subseteq*:
$\textit{worklist-tree-coll-aux}\ \textit{succs}\ P\ f\ \textit{ws}\ t_0 = \textit{Some}\ t \Longrightarrow$
$\textit{invariant}\ \textit{inv0}\ \textit{succs} \Longrightarrow\ \forall g \in \textit{set ws}.\ \textit{inv0}\ g \Longrightarrow$
$(\bigwedge x.\ \textit{inv0}\ x \Longrightarrow I(f\ x)) \Longrightarrow SI\ t_0 \Longrightarrow$
$\textit{set-of}\ t \subseteq \textit{set-of}\ t_0 \cup f\ `\ \{h : (\textit{Rel succs})^*\ ``\ \textit{set ws}.\ P\ h\}$
⟨*proof*⟩

lemma *worklist-tree-coll-subseteq*:
$\textit{worklist-tree-coll}\ \textit{succs}\ P\ f\ \textit{ws} = \textit{Some}\ t \Longrightarrow$
$\textit{invariant}\ \textit{inv0}\ \textit{succs} \Longrightarrow \forall g \in \textit{set ws}.\ \textit{inv0}\ g \Longrightarrow$

$(\bigwedge x.\ inv0\ x \Longrightarrow I(f\ x)) \Longrightarrow$
$set\text{-}of\ t \subseteq f\ `\ \{h : (Rel\ succs)^*\ ``\ set\ ws.\ P\ h\}$
⟨*proof*⟩

lemma *worklist-tree-coll-inv*:
$worklist\text{-}tree\text{-}coll\ succs\ P\ f\ ws = Some\ s \Longrightarrow S\ s$
⟨*proof*⟩

end

end
theory *Maps*
imports *Worklist Quasi-Order*
begin

locale *maps* =
fixes *empty* :: $'m$
and *up* :: $'a \Rightarrow 'b\ list \Rightarrow 'm \Rightarrow 'm$
and *map-of* :: $'m \Rightarrow 'a \Rightarrow 'b\ list$
and *M* :: $'m \Rightarrow bool$
assumes *map-empty*: $map\text{-}of\ empty = (\lambda a.\ [])$
and *map-up*: $map\text{-}of\ (up\ a\ b\ m) = (map\text{-}of\ m)(a := b)$
and *M-empty*: $M\ empty$
and *M-up*: $M\ m \Longrightarrow M\ (up\ a\ b\ m)$
begin

definition $set\text{-}of\ m = (UN\ x.\ set(map\text{-}of\ m\ x))$

end

locale *set-mod-maps* = *maps empty up map-of M* + *quasi-order qle*
for *empty* :: $'m$
and *up* :: $'a \Rightarrow 'b\ list \Rightarrow 'm \Rightarrow 'm$
and *map-of* :: $'m \Rightarrow 'a \Rightarrow 'b\ list$
and *M* :: $'m \Rightarrow bool$
and *qle* :: $'b \Rightarrow 'b \Rightarrow bool$ (**infix** $\preceq$ *60*)
+
fixes *subsumed* :: $'b \Rightarrow 'b \Rightarrow bool$
and *I* :: $'b \Rightarrow bool$
and *key* :: $'b \Rightarrow 'a$
assumes *equiv-iff-qle*: $I\ x \Longrightarrow I\ y \Longrightarrow subsumed\ x\ y = (x \preceq y)$
and *key=key*
begin

definition $insert\text{-}mod\ x\ m =$
$(let\ k = key\ x;\ ys = map\text{-}of\ m\ k$
$in\ if\ (\exists\ y \in set\ ys.\ subsumed\ x\ y)\ then\ m\ else\ up\ k\ (x\#ys)\ m)$

end

sublocale
set-mod-maps <
set-by-maps?: *set-modulo qle empty insert-mod set-of I M*
⟨*proof*⟩

end

24 Archive

theory *Arch*
imports *Main HOL−Library.Code-Target-Numeral*
begin

⟨*ML*⟩

The definition of these constants is only ever needed at the ML level when running the eval proof method.

definition *Tri* :: *nat list list list*
where
Tri = (*map* ∘ *map* ∘ *map*) *nat-of-integer Tri′*

definition *Quad* :: *nat list list list*
where
Quad = (*map* ∘ *map* ∘ *map*) *nat-of-integer Quad′*

definition *Pent* :: *nat list list list*
where
Pent = (*map* ∘ *map* ∘ *map*) *nat-of-integer Pent′*

definition *Hex* :: *nat list list list*
where
Hex = (*map* ∘ *map* ∘ *map*) *nat-of-integer Hex′*

end

25 Comparing Enumeration and Archive

theory *ArchCompAux*
imports *TameEnum Trie.Tries Maps Arch Worklist*
begin

function *qsort* :: (′*a* ⇒ ′*a* ⇒ *bool*) ⇒ ′*a list* ⇒ ′*a list* **where**
qsort le [] = [] |
qsort le (*x*#*xs*) = *qsort le* [*y*←*xs* . ¬ *le x y*] @ [*x*] @
qsort le [*y*←*xs* . *le x y*]

⟨*proof*⟩
termination ⟨*proof*⟩

definition *nof-vertices* :: *'a fgraph* ⇒ *nat* **where**
nof-vertices = *length* ∘ *remdups* ∘ *concat*

definition *fgraph* :: *graph* ⇒ *nat fgraph* **where**
fgraph g = *map vertices* (*faces g*)

definition *hash* :: *nat fgraph* ⇒ *nat list* **where**
hash fs = (*let n* = *nof-vertices fs in*
[*n*, *size fs*] @
qsort (λ*x y*. *y* < *x*) (*map* (λ*i*. *foldl* (+) *0* (*map size* [*f*←*fs*. *i* ∈ *set f*]))
[*0*..<*n*]))

definition *samet* :: (*nat*,*nat fgraph*) *tries option* ⇒ *nat fgraph list* ⇒ *bool*
where
samet fgto ags = (*case fgto of None* ⇒ *False* | *Some tfgs* ⇒
let tags = *tries-of-list hash ags in*
(*all-tries* (λ*fg*. *list-ex* (*iso-test fg*) (*lookup-tries tags* (*hash fg*))) *tfgs* ∧
all-tries (λ*ag*. *list-ex* (*iso-test ag*) (*lookup-tries tfgs* (*hash ag*))) *tags*))

definition *pre-iso-test* :: *vertex fgraph* ⇒ *bool* **where**
pre-iso-test Fs ⟷
[] ∉ *set Fs* ∧ (∀ *F*∈*set Fs*. *distinct F*) ∧ *distinct* (*map rotate-min Fs*)

interpretation *map*:
maps Trie None [] *update-trie lookup-tries invar-trie*
⟨*proof*⟩

lemma *set-of-conv*: *set-tries* = *maps.set-of lookup-tries*
⟨*proof*⟩

end

26 Completeness of Archive Test

theory *ArchCompProps*
imports *TameEnumProps ArchCompAux*
begin
lemma *mgp-pre-iso-test*: *minGraphProps g* ⟹ *pre-iso-test*(*fgraph g*)
⟨*proof*⟩

corollary *iso-test-correct*:
⟦ *pre-iso-test* Fs_1; *pre-iso-test* Fs_2 ⟧ ⟹
iso-test Fs_1 Fs_2 = (Fs_1 ≃ Fs_2)
⟨*proof*⟩

lemma *trie-all-eq-set-of-trie*:
invar-trie t $\Longrightarrow$ *all-trie* (*list-all P*) *t* = ($\forall$ *v* $\in$ *set-tries t*. *P v*)
⟨*proof*⟩

lemma *samet-imp-iso-seteq*:
assumes *pre1*: $\bigwedge$*gs g*. *gsopt* = *Some gs* $\Longrightarrow$ *g* $\in$ *set-tries gs* $\Longrightarrow$ *pre-iso-test g*
and *pre2*: $\bigwedge$*g*. *g* $\in$ *set arch* $\Longrightarrow$ *pre-iso-test g*
and *inv*: $\bigwedge$*gs*. *gsopt* = *Some gs* $\Longrightarrow$ *invar-trie gs*
and *same*: *samet gsopt arch*
shows $\exists$ *gs*. *gsopt* = *Some gs* $\wedge$ *set-tries gs* $=_{\simeq}$ *set arch*
⟨*proof*⟩

lemma *samet-imp-iso-subseteq*:
assumes *pre1*: $\bigwedge$*gs g*. *gsopt* = *Some gs* $\Longrightarrow$ *g* $\in$ *set-tries gs* $\Longrightarrow$ *pre-iso-test g*
and *pre2*: $\bigwedge$*g*. *g* $\in$ *set arch* $\Longrightarrow$ *pre-iso-test g*
and *inv*: $\bigwedge$*gs*. *gsopt* = *Some gs* $\Longrightarrow$ *invar-trie gs*
and *same*: *samet gsopt arch*
shows $\exists$ *gs*. *gsopt* = *Some gs* $\wedge$ *set-tries gs* $\subseteq_{\simeq}$ *set arch*
⟨*proof*⟩

global-interpretation *set-mod-trie*:
set-mod-maps Trie None [] *update-trie lookup-tries invar-trie* ($\simeq$) *iso-test pre-iso-test hash*
defines *insert-mod-trie* = *set-mod-maps.insert-mod update-trie lookup-tries iso-test hash*
and *worklist-tree-coll-trie* = *set-modulo.worklist-tree-coll* (*Trie None* []) *insert-mod-trie*
and *worklist-tree-coll-aux-trie* = *set-modulo.worklist-tree-coll-aux insert-mod-trie*
and *insert-mod2-trie* = *set-modulo.insert-mod2 insert-mod-trie*
⟨*proof*⟩

definition *enum-filter-finals* ::
(*graph* ⇒ *graph list*) ⇒ *graph list*
⇒ (*nat*,*nat fgraph*) *tries option* **where**
enum-filter-finals succs = *set-mod-trie.worklist-tree-coll succs final fgraph*

definition *tameEnumFilter* :: *nat* ⇒ (*nat*,*nat fgraph*)*tries option* **where**
tameEnumFilter p = *enum-filter-finals* (*next-tame p*) [*Seed p*]

lemma *TameEnum-tameEnumFilter*:
tameEnumFilter p = *Some t* $\Longrightarrow$ *set-tries t* $=_{\simeq}$ *fgraph* ‘ $TameEnum_p$
⟨*proof*⟩

lemma *tameEnumFilter-subseteq-TameEnum*:
tameEnumFilter p = *Some t* $\Longrightarrow$ *set-tries t* $\subseteq$ *fgraph* ‘ $TameEnum_p$
⟨*proof*⟩

lemma *inv-tries-tameEnumFilter*:

tameEnumFilter p = Some t ⟹ invar-trie t
⟨*proof*⟩

theorem *combine-evals-filter*:
∀ *g* ∈ *set arch*. *pre-iso-test g* ⟹ *samet* (*tameEnumFilter p*) *arch*
⟹ *fgraph* ‘ *TameEnum*$_p$ ⊆$_\sim$ *set arch*
⟨*proof*⟩

end

27 Completeness Proofs under hypothetical computations

theory *Relative-Completeness*
imports *ArchCompProps*
begin

definition *Archive* :: *vertex fgraph set* **where**
Archive ≡ *set*(*Tri* @ *Quad* @ *Pent* @ *Hex*)

locale *archive-by-computation* =
assumes *pre-iso-test3*: ∀ *g* ∈ *set Tri*. *pre-iso-test g*
assumes *pre-iso-test4*: ∀ *g* ∈ *set Quad*. *pre-iso-test g*
assumes *pre-iso-test5*: ∀ *g* ∈ *set Pent*. *pre-iso-test g*
assumes *pre-iso-test6*: ∀ *g* ∈ *set Hex*. *pre-iso-test g*
assumes *same3*: *samet* (*tameEnumFilter 0*) *Tri*
assumes *same4*: *samet* (*tameEnumFilter 1*) *Quad*
assumes *same5*: *samet* (*tameEnumFilter 2*) *Pent*
assumes *same6*: *samet* (*tameEnumFilter 3*) *Hex*
begin

theorem *TameEnum-Archive*: *fgraph* ‘ *TameEnum* ⊆$_\sim$ *Archive*
⟨*proof*⟩

lemma *TameEnum-comp*:
assumes *Seed*$_p$ [*next-plane*$_p$]→∗ *g* **and** *final g* **and** *tame g*
shows *Seed*$_p$ [*next-tame*$_p$]→∗ *g*
⟨*proof*⟩

lemma *tame5*:
assumes *g*: *Seed*$_p$ [*next-plane0*$_p$]→∗ *g* **and** *final g* **and** *tame g*
shows *p* ≤ *3*
⟨*proof*⟩

theorem *completeness*:
assumes *g* ∈ *PlaneGraphs* **and** *tame g* **shows** *fgraph g* ∈$_\sim$ *Archive*

⟨*proof*⟩

end

end

References

[1] Tobias Nipkow, Gertrud Bauer, and Paula Schultz. Flyspeck I: Tame graphs. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *LNCS*, pages 21–35. Springer, 2006.