

Chamber complexes, Coxeter systems, and buildings

Jeremy Sylvestre
University of Alberta, Augustana Campus
jeremy.sylvestre@ualberta.ca

September 13, 2023

Abstract

We provide a basic formal framework for the theory of chamber complexes and Coxeter systems, and for buildings as thick chamber complexes endowed with a system of apartments. Along the way, we develop some of the general theory of abstract simplicial complexes and of groups (relying on the *group_add* class for the basics), including free groups and group presentations, and their universal properties. The main results verified are that the deletion condition is both necessary and sufficient for a group with a set of generators of order two to be a Coxeter system, and that the apartments in a (thick) building are all uniformly Coxeter.

Contents

1 Preliminaries	5
1.1 Natural numbers	5
1.2 Logic	5
1.3 Sets	6
1.4 Functions and relations	7
1.4.1 Miscellaneous	7
1.4.2 Equality of functions restricted to a set	7
1.4.3 Injectivity, surjectivity, bijectivity, and inverses	9
1.4.4 Induced functions on sets of sets and lists of sets	10
1.4.5 Induced functions on quotients	11
1.4.6 Support of a function	11
1.5 Lists	12
1.5.1 Miscellaneous facts	12
1.5.2 Cases	12
1.5.3 Induction	13
1.5.4 Alternating lists	14

1.5.5	Binary relation chains	16
1.5.6	Set of subseqs	17
1.6	Orders and posets	18
1.6.1	Morphisms of posets	18
1.6.2	More <i>arg-min</i>	20
1.6.3	Bottom of a set	20
1.6.4	Minimal and pseudominimal elements in sets	22
1.6.5	Set of elements below another	23
1.6.6	Lower bounds	24
1.6.7	Simplex-like posets	25
1.6.8	The superset ordering	27
2	Algebra	28
2.1	Miscellaneous algebra facts	28
2.2	The type of permutations of a type	29
2.3	Natural action of <i>nat</i> on types of class <i>monoid-add</i>	30
2.3.1	Translation from class <i>power</i>	30
2.3.2	Additive order of an element	31
2.4	Partial sums of a list	32
2.5	Sums of alternating lists	33
2.6	Conjugation in <i>group-add</i>	34
2.6.1	Abbreviations and basic facts	34
2.6.2	The conjugation sequence	35
2.6.3	The action on signed <i>group-add</i> elements	36
2.7	Cosets	38
2.7.1	Basic facts	38
2.7.2	The supset order on cosets	38
2.7.3	The afforded partition	39
2.8	Groups	39
2.8.1	Locale definition and basic facts	39
2.8.2	Sets with a suitable binary operation	40
2.8.3	Cosets of a <i>Group</i>	42
2.8.4	The <i>Group</i> generated by a set	43
2.8.5	Homomorphisms and isomorphisms	45
2.8.6	Normal subgroups	47
2.8.7	Quotient groups	48
2.8.8	The induced homomorphism on a quotient group	49
2.9	Free groups	50
2.9.1	Words in letters of <i>signed</i> type	50
2.9.2	The collection of proper signed lists as a type	52
2.9.3	Lifts of functions on the letter type	55
2.9.4	Free groups on a set	57
2.9.5	Group presentations	59
2.10	Words over a generating set	63

3	Simplicial complexes	67
3.1	Geometric notions	67
3.1.1	Facets	67
3.1.2	Adjacency	68
3.1.3	Chains of adjacent sets	69
3.2	Locale and basic facts	69
3.3	Chains of maximal simplices	70
3.4	Isomorphisms of simplicial complexes	74
3.5	The complex associated to a poset	75
4	Chamber complexes	76
4.1	Locale definition and basic facts	76
4.2	The system of chambers and distance between chambers	79
4.3	Labelling a chamber complex	81
4.4	Morphisms of chamber complexes	82
4.4.1	Morphism locale and basic facts	82
4.4.2	Action on pregalleries and galleries	84
4.4.3	Properties of the image	84
4.4.4	Action on the chamber system	85
4.4.5	Isomorphisms	85
4.4.6	Endomorphisms	87
4.4.7	Automorphisms	89
4.4.8	Retractions	90
4.4.9	Foldings of chamber complexes	91
4.5	Thin chamber complexes	93
4.5.1	Locales and basic facts	93
4.5.2	The standard uniqueness argument for chamber morphisms of thin chamber complexes	95
4.6	Foldings of thin chamber complexes	96
4.6.1	Locale definition and basic facts	96
4.6.2	Pairs of opposed foldings	99
4.6.3	The automorphism induced by a pair of opposed foldings	101
4.6.4	Walls	105
4.7	Thin chamber complexes with many foldings	109
4.7.1	Locale definition and basic facts	109
4.7.2	The group of automorphisms	110
4.7.3	Action of the group of automorphisms on the chamber system	112
4.7.4	A labelling by the vertices of the fundamental chamber	114
4.7.5	More on the action of the group of automorphisms on chambers	116
4.7.6	A bijection between the fundamental chamber and the set of generating automorphisms	117
4.8	Thick chamber complexes	118

5	Coxeter systems and complexes	119
5.1	Coxeter-like systems	119
5.1.1	Locale definition and basic facts	119
5.1.2	Special cosets	120
5.1.3	Transfer from the free group over generators	121
5.1.4	Words in generators containing alternating subwords	123
5.1.5	Preliminary facts on the word problem	125
5.1.6	Preliminary facts related to the deletion condition	126
5.2	Coxeter-like systems with deletion	127
5.2.1	Locale definition	127
5.2.2	Consequences of the deletion condition	127
5.2.3	The exchange condition	128
5.2.4	More on words in generators containing alternating subwords	128
5.2.5	The word problem	129
5.2.6	Special subgroups and cosets	129
5.3	Coxeter systems	132
5.3.1	Locale definition and transfer from the associated free group	132
5.3.2	The deletion condition is necessary	132
5.3.3	The deletion condition is sufficient	133
5.3.4	The Coxeter system associated to a thin chamber com- plex with many foldings	134
5.4	Coxeter complexes	136
5.4.1	Locale and complex definitions	136
5.4.2	As a simplicial complex	136
5.4.3	As a chamber complex	138
5.4.4	The Coxeter complex associated to a thin chamber complex with many foldings	139
6	Buildings	141
6.1	Apartment systems	142
6.1.1	Locale and basic facts	142
6.1.2	Isomorphisms between apartments	144
6.1.3	Retractions onto apartments	145
6.1.4	Distances in apartments	146
6.1.5	Special situation: a triangle of apartments and chambers	147
6.2	Building locale and basic lemmas	153
6.3	Apartments are uniformly Coxeter	153

Note: A number of the proofs in this theory were modelled on or inspired by proofs in the books on buildings by Abramenko and Brown [1] and by

Garrett [2]. As well, some of the definitions, statements, and proofs appearing in the first two sections previously appeared in a submission to the *Archive of Formal Proofs* by the author of the current submission [4].

1 Preliminaries

In this section, we establish some basic facts about natural numbers, logic, sets, functions and relations, lists, and orderings and posets, that are either not available in the HOL library or are in a form not suitable for our purposes.

```
theory Prelim
imports Main HOL-Library.Set-Algebras
begin
```

```
declare image-cong-simp [cong del]
```

1.1 Natural numbers

```
lemma nat-cases-2Suc [case-names 0 1 SucSuc]:
  assumes    0:  $n = 0 \implies P$ 
  and       1:  $n = 1 \implies P$ 
  and       SucSuc:  $\bigwedge m. n = \text{Suc } (\text{Suc } m) \implies P$ 
  shows     P
  <proof>
```

```
lemma nat-even-induct [case-names - 0 SucSuc]:
  assumes   even: even n
  and       0: P 0
  and       SucSuc:  $\bigwedge m. \text{even } m \implies P\ m \implies P\ (\text{Suc } (\text{Suc } m))$ 
  shows     P n
  <proof>
```

```
lemma nat-induct-step2 [case-names 0 1 SucSuc]:
  assumes   0: P 0
  and       1: P 1
  and       SucSuc:  $\bigwedge m. P\ m \implies P\ (\text{Suc } (\text{Suc } m))$ 
  shows     P n
  <proof>
```

1.2 Logic

```
lemma ex1-unique:  $\exists! x. P\ x \implies P\ a \implies P\ b \implies a=b$ 
  <proof>
```

```
lemma not-the1:
```

assumes $\exists!x. P x y \neq (THE x. P x)$
shows $\neg P y$
 $\langle proof \rangle$

lemma *two-cases* [*case-names both one other neither*]:

assumes *both* : $P \implies Q \implies R$
and *one* : $P \implies \neg Q \implies R$
and *other* : $\neg P \implies Q \implies R$
and *neither*: $\neg P \implies \neg Q \implies R$
shows R
 $\langle proof \rangle$

1.3 Sets

lemma *beX1-equality*: $\llbracket \exists!x \in A. P x; x \in A; P x; y \in A; P y \rrbracket \implies x=y$
 $\langle proof \rangle$

lemma *prod-ballI*: $(\bigwedge a b. (a,b) \in A \implies P a b) \implies \forall (a,b) \in A. P a b$
 $\langle proof \rangle$

lemmas *seteqI* = *set-eqI*[*OF iffI*]

lemma *set-decomp-subset*:

$\llbracket U = A \cup B; A \subseteq X; B \subseteq Y; X \subseteq U; X \cap Y = \{\} \rrbracket \implies A = X$
 $\langle proof \rangle$

lemma *insert-subset-equality*: $\llbracket a \notin A; a \notin B; insert a A = insert a B \rrbracket \implies A=B$
 $\langle proof \rangle$

lemma *insert-compare-element*: $a \notin A \implies insert b A = insert a A \implies b=a$
 $\langle proof \rangle$

lemma *card1*:

assumes *card* $A = 1$
shows $\exists a. A = \{a\}$
 $\langle proof \rangle$

lemma *singleton-pow*: $a \in A \implies \{a\} \in Pow A$
 $\langle proof \rangle$

definition *separated-by* :: $'a set set \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$

where *separated-by* $w x y \equiv \exists A B. w = \{A,B\} \wedge x \in A \wedge y \in B$

lemma *separated-byI*: $x \in A \implies y \in B \implies separated-by \{A,B\} x y$
 $\langle proof \rangle$

lemma *separated-by-disjoint*: $\llbracket separated-by \{A,B\} x y; A \cap B = \{\}; x \in A \rrbracket \implies y \in B$
 $\langle proof \rangle$

lemma *separated-by-in-other*: *separated-by* $\{A,B\}$ $x y \implies x \notin A \implies x \in B \wedge y \in A$
<proof>

lemma *separated-by-not-empty*: *separated-by* $w x y \implies w \neq \{\}$
<proof>

lemma *not-self-separated-by-disjoint*: $A \cap B = \{\} \implies \neg$ *separated-by* $\{A,B\}$ $x x$
<proof>

1.4 Functions and relations

1.4.1 Miscellaneous

lemma *cong-let*: $(\text{let } x = y \text{ in } f x) = f y$ *<proof>*

lemma *sym-sym*: *sym* $(A \times A)$ *<proof>*

lemma *trans-sym*: *trans* $(A \times A)$ *<proof>*

lemma *map-prod-sym*: *sym* $A \implies$ *sym* $(\text{map-prod } f f ' A)$
<proof>

abbreviation *restrict1* :: $('a \Rightarrow 'a) \Rightarrow 'a \text{ set} \Rightarrow ('a \Rightarrow 'a)$
where *restrict1* $f A \equiv (\lambda a. \text{if } a \in A \text{ then } f a \text{ else } a)$

lemma *restrict1-image*: $B \subseteq A \implies$ *restrict1* $f A ' B = f ' B$
<proof>

1.4.2 Equality of functions restricted to a set

definition *fun-eq-on* $f g A \equiv (\forall a \in A. f a = g a)$

lemma *fun-eq-onI*: $(\bigwedge a. a \in A \implies f a = g a) \implies$ *fun-eq-on* $f g A$
<proof>

lemma *fun-eq-onD*: *fun-eq-on* $f g A \implies a \in A \implies f a = g a$
<proof>

lemma *fun-eq-on-UNIV*: $(\text{fun-eq-on } f g \text{ UNIV}) = (f = g)$
<proof>

lemma *fun-eq-on-subset*: *fun-eq-on* $f g A \implies B \subseteq A \implies$ *fun-eq-on* $f g B$
<proof>

lemma *fun-eq-on-sym*: *fun-eq-on* $f g A \implies$ *fun-eq-on* $g f A$
<proof>

lemma *fun-eq-on-trans*: *fun-eq-on* $f g A \implies$ *fun-eq-on* $g h A \implies$ *fun-eq-on* $f h A$
<proof>

lemma *fun-eq-on-cong*: $\text{fun-eq-on } f \ h \ A \implies \text{fun-eq-on } g \ h \ A \implies \text{fun-eq-on } f \ g \ A$
 ⟨proof⟩

lemma *fun-eq-on-im* : $\text{fun-eq-on } f \ g \ A \implies B \subseteq A \implies f'B = g'B$
 ⟨proof⟩

lemma *fun-eq-on-subset-and-diff-imp-eq-on*:
assumes $A \subseteq B \ \text{fun-eq-on } f \ g \ A \ \text{fun-eq-on } f \ g \ (B - A)$
shows $\text{fun-eq-on } f \ g \ B$
 ⟨proof⟩

lemma *fun-eq-on-set-and-comp-imp-eq*:
 $\text{fun-eq-on } f \ g \ A \implies \text{fun-eq-on } f \ g \ (-A) \implies f = g$
 ⟨proof⟩

lemma *fun-eq-on-bij-betw*: $\text{fun-eq-on } f \ g \ A \implies \text{bij-betw } f \ A \ B = \text{bij-betw } g \ A \ B$
 ⟨proof⟩

lemma *fun-eq-on-restrict1*: $\text{fun-eq-on } (\text{restrict1 } f \ A) \ f \ A$
 ⟨proof⟩

abbreviation *fixespointwise* $f \ A \equiv \text{fun-eq-on } f \ \text{id } A$

lemmas *fixespointwiseI* = fun-eq-onI [of - - id]
lemmas *fixespointwiseD* = fun-eq-onD [of - - id]
lemmas *fixespointwise-cong* = fun-eq-on-trans [of - - id]
lemmas *fixespointwise-subset* = fun-eq-on-subset [of - - id]
lemmas *fixespointwise2-imp-eq-on* = fun-eq-on-cong [of - - id]

lemmas *fixespointwise-subset-and-diff-imp-eq-on* =
 $\text{fun-eq-on-subset-and-diff-imp-eq-on}$ [of - - id]

lemma *id-fixespointwise*: $\text{fixespointwise } \text{id } A$
 ⟨proof⟩

lemma *fixespointwise-im*: $\text{fixespointwise } f \ A \implies B \subseteq A \implies f'B = B$
 ⟨proof⟩

lemma *fixespointwise-comp*:
 $\text{fixespointwise } f \ A \implies \text{fixespointwise } g \ A \implies \text{fixespointwise } (g \circ f) \ A$
 ⟨proof⟩

lemma *fixespointwise-insert*:
assumes $\text{fixespointwise } f \ A \ f' \ (\text{insert } a \ A) = \text{insert } a \ A$
shows $\text{fixespointwise } f \ (\text{insert } a \ A)$
 ⟨proof⟩

lemma *fixespointwise-restrict1*:
 $\text{fixespointwise } f \ A \implies \text{fixespointwise } (\text{restrict1 } f \ B) \ A$

<proof>

lemma *fold-fixespointwise*:

$\forall x \in \text{set } xs. \text{fixespointwise } (f \ x) \ A \implies \text{fixespointwise } (\text{fold } f \ xs) \ A$
<proof>

lemma *funpower-fixespointwise*:

assumes *fixespointwise* $f \ A$
shows *fixespointwise* $(f \ \sim^n) \ A$
<proof>

1.4.3 Injectivity, surjectivity, bijectivity, and inverses

lemma *inj-on-to-singleton*:

assumes *inj-on* $f \ A \ f' A = \{b\}$
shows $\exists a. A = \{a\}$
<proof>

lemmas *inj-inj-on = subset-inj-on*[*of - UNIV, OF - subset-UNIV*]

lemma *inj-on-eq-image'*: $\llbracket \text{inj-on } f \ A; X \subseteq A; Y \subseteq A; f' X \subseteq f' Y \rrbracket \implies X \subseteq Y$
<proof>

lemma *inj-on-eq-image*: $\llbracket \text{inj-on } f \ A; X \subseteq A; Y \subseteq A; f' X = f' Y \rrbracket \implies X = Y$
<proof>

lemmas *inj-eq-image = inj-on-eq-image*[*OF - subset-UNIV subset-UNIV*]

lemma *induced-pow-fun-inj-on*:

assumes *inj-on* $f \ A$
shows *inj-on* $((\cdot) \ f) \ (\text{Pow } A)$
<proof>

lemma *inj-on-minus-set*: *inj-on* $((-) \ A) \ (\text{Pow } A)$
<proof>

lemma *induced-pow-fun-surj*:

$((\cdot) \ f) \ ' (\text{Pow } A) = \text{Pow } (f' A)$
<proof>

lemma *bij-betw-f-the-inv-into-f*:

$\text{bij-betw } f \ A \ B \implies y \in B \implies f \ (\text{the-inv-into } A \ f \ y) = y$
— an equivalent lemma appears in the HOL library, but this version avoids the double *bij-betw* premises
<proof>

lemma *bij-betw-the-inv-into-onto*: $\text{bij-betw } f \ A \ B \implies \text{the-inv-into } A \ f \ ' B = A$
<proof>

lemma *bij-betw-imp-bij-betw-Pow*:
assumes *bij-betw f A B*
shows *bij-betw ((\cdot) f) (Pow A) (Pow B)*
 \langle *proof* \rangle

lemma *comps-fixpointwise-imp-bij-betw*:
assumes *f'X \subseteq Y g'Y \subseteq X fixespointwise (g \circ f) X fixespointwise (f \circ g) Y*
shows *bij-betw f X Y*
 \langle *proof* \rangle

lemma *set-permutation-bij-restrict1*:
assumes *bij-betw f A A*
shows *bij (restrict1 f A)*
 \langle *proof* \rangle

lemma *set-permutation-the-inv-restrict1*:
assumes *bij-betw f A A*
shows *the-inv (restrict1 f A) = restrict1 (the-inv-into A f) A*
 \langle *proof* \rangle

lemma *the-inv-into-the-inv-into*:
inj-on f A \implies a \in A \implies the-inv-into (f'A) (the-inv-into A f) a = f a
 \langle *proof* \rangle

lemma *the-inv-into-f-im-f-im*:
assumes *inj-on f A x \subseteq A*
shows *the-inv-into A f ' f ' x = x*
 \langle *proof* \rangle

lemma *f-im-the-inv-into-f-im*:
assumes *inj-on f A x \subseteq f'A*
shows *f ' the-inv-into A f ' x = x*
 \langle *proof* \rangle

lemma *the-inv-leftinv*: *bij f \implies the-inv f \circ f = id*
 \langle *proof* \rangle

1.4.4 Induced functions on sets of sets and lists of sets

Here we create convenience abbreviations for distributing a function over a set of sets and over a list of sets.

abbreviation *setsetmapim* :: (*'a \Rightarrow 'b*) \Rightarrow *'a set set \Rightarrow 'b set set* (**infix** \vdash 70)
where *f \vdash X* \equiv ((\cdot) f) ' X

abbreviation *setlistmapim* :: (*'a \Rightarrow 'b*) \Rightarrow *'a set list \Rightarrow 'b set list* (**infix** \models 70)
where *f \models Xs* \equiv map ((\cdot) f) Xs

lemma *setsetmapim-comp*: *(f \circ g) \vdash A = f \vdash (g \vdash A)*
 \langle *proof* \rangle

lemma *setlistmapim-comp*: $(f \circ g) \models xs = f \models (g \models xs)$
 ⟨proof⟩

lemma *setsetmapim-cong-subset*:
 assumes *fun-eq-on* $g f (\bigcup A) B \subseteq A$
 shows $g \vdash B \subseteq f \vdash B$
 ⟨proof⟩

lemma *setsetmapim-cong*:
 assumes *fun-eq-on* $g f (\bigcup A) B \subseteq A$
 shows $g \vdash B = f \vdash B$
 ⟨proof⟩

lemma *setsetmapim-restrict1*: $B \subseteq A \implies \text{restrict1 } f (\bigcup A) \vdash B = f \vdash B$
 ⟨proof⟩

lemma *setsetmapim-the-inv-into*:
 assumes *inj-on* $f (\bigcup A)$
 shows $(\text{the-inv-into } (\bigcup A) f) \vdash (f \vdash A) = A$
 ⟨proof⟩

1.4.5 Induced functions on quotients

Here we construct the induced function on a quotient for an inducing function that respects the relation that defines the quotient.

lemma *respects-imp-unique-image-rel*: $f \text{ respects } r \implies y \in f' r^{-1} \{a\} \implies y = f a$
 ⟨proof⟩

lemma *ex1-class-image*:
 assumes *refl-on* A r $f \text{ respects } r$ $X \in A // r$
 shows $\exists ! b. b \in f' X$
 ⟨proof⟩

definition *quotientfun* :: $('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow 'b$
 where $\text{quotientfun } f X = (\text{THE } b. b \in f' X)$

lemma *quotientfun-equality*:
 assumes *refl-on* A r $f \text{ respects } r$ $X \in A // r$ $b \in f' X$
 shows $\text{quotientfun } f X = b$
 ⟨proof⟩

lemma *quotientfun-classrep-equality*:
 $\llbracket \text{refl-on } A \text{ } r; f \text{ respects } r; a \in A \rrbracket \implies \text{quotientfun } f (r^{-1} \{a\}) = f a$
 ⟨proof⟩

1.4.6 Support of a function

definition *supp* :: $('a \Rightarrow 'b :: \text{zero}) \Rightarrow 'a \text{ set}$ where $\text{supp } f = \{x. f x \neq 0\}$

lemma *suppI-contr*: $x \notin \text{supp } f \implies f x = 0$
 ⟨*proof*⟩

lemma *suppD-contr*: $f x = 0 \implies x \notin \text{supp } f$
 ⟨*proof*⟩

abbreviation *restrict0* :: ('a ⇒ 'b :: zero) ⇒ 'a set ⇒ ('a ⇒ 'b)
where *restrict0* f A ≡ (λa. if a ∈ A then f a else 0)

lemma *supp-restrict0* : $\text{supp } (\text{restrict0 } f A) \subseteq A$
 ⟨*proof*⟩

1.5 Lists

1.5.1 Miscellaneous facts

lemma *snoc-conv-cons*: $\exists x xs. ys @ [y] = x \# xs$
 ⟨*proof*⟩

lemma *cons-conv-snoc*: $\exists ys y. x \# xs = ys @ [y]$
 ⟨*proof*⟩

lemma *distinct-count-list*:
distinct xs \implies *count-list* xs a = (if a ∈ set xs then 1 else 0)
 ⟨*proof*⟩

lemma *map-fst-map-const-snd*: $\text{map } \text{fst } (\text{map } (\lambda s. (s, b)) xs) = xs$
 ⟨*proof*⟩

lemma *inj-on-distinct-setlistmapim*:
assumes *inj-on* f A
shows $\forall X \in \text{set } Xs. X \subseteq A \implies \text{distinct } Xs \implies \text{distinct } (f|_Xs)$
 ⟨*proof*⟩

1.5.2 Cases

lemma *list-cases-Cons-snoc* [case-names Nil Single Cons-snoc]:
assumes *Nil*: $xs = [] \implies P$
and *Single*: $\bigwedge x. xs = [x] \implies P$
and *Cons-snoc*: $\bigwedge x ys y. xs = x \# ys @ [y] \implies P$
shows P
 ⟨*proof*⟩

lemma *two-lists-cases-Cons-Cons* [case-names Nil1 Nil2 ConsCons]:
assumes *Nil1*: $\bigwedge ys. as = [] \implies bs = ys \implies P$
and *Nil2*: $\bigwedge xs. as = xs \implies bs = [] \implies P$
and *ConsCons*: $\bigwedge x xs y ys. as = x \# xs \implies bs = y \# ys \implies P$
shows P
 ⟨*proof*⟩

lemma *two-lists-cases-snoc-Cons* [*case-names Nil1 Nil2 snoc-Cons*]:
assumes $Nil1: \bigwedge ys. as = [] \implies bs = ys \implies P$
and $Nil2: \bigwedge xs. as = xs \implies bs = [] \implies P$
and $snoc-Cons: \bigwedge xs\ x\ y\ ys. as = xs @ [x] \implies bs = y \# ys \implies P$
shows P
<proof>

lemma *two-lists-cases-snoc-Cons'* [*case-names both-Nil Nil1 Nil2 snoc-Cons*]:
assumes $both-Nil: as = [] \implies bs = [] \implies P$
and $Nil1: \bigwedge y\ ys. as = [] \implies bs = y \# ys \implies P$
and $Nil2: \bigwedge xs\ x. as = xs @ [x] \implies bs = [] \implies P$
and $snoc-Cons: \bigwedge xs\ x\ y\ ys. as = xs @ [x] \implies bs = y \# ys \implies P$
shows P
<proof>

lemma *two-prod-lists-cases-snoc-Cons*:
assumes $\bigwedge xs. as = xs \implies bs = [] \implies P \bigwedge ys. as = [] \implies bs = ys \implies P$
 $\bigwedge xs\ aa\ ba\ ab\ bb\ ys. as = xs @ [(aa, ba)] \wedge bs = (ab, bb) \# ys \implies P$
shows P
<proof>

lemma *three-lists-cases-snoc-mid-Cons*
[*case-names Nil1 Nil2 Nil3 snoc-single-Cons snoc-mid-Cons*]:
assumes $Nil1: \bigwedge ys\ zs. as = [] \implies bs = ys \implies cs = zs \implies P$
and $Nil2: \bigwedge xs\ zs. as = xs \implies bs = [] \implies cs = zs \implies P$
and $Nil3: \bigwedge xs\ ys. as = xs \implies bs = ys \implies cs = [] \implies P$
and *snoc-single-Cons*:
 $\bigwedge xs\ x\ y\ z\ zs. as = xs @ [x] \implies bs = [y] \implies cs = z \# zs \implies P$
and *snoc-mid-Cons*:
 $\bigwedge xs\ x\ w\ ys\ y\ z\ zs. as = xs @ [x] \implies bs = w \# ys @ [y] \implies$
 $cs = z \# zs \implies P$
shows P
<proof>

1.5.3 Induction

lemma *list-induct-CCons* [*case-names Nil Single CCons*]:
assumes $Nil : P []$
and $Single: \bigwedge x. P [x]$
and $CCons : \bigwedge x\ y\ xs. P (y \# xs) \implies P (x \# y \# xs)$
shows $P xs$
<proof>

lemma *list-induct-ssnoc* [*case-names Nil Single ssnoc*]:
assumes $Nil : P []$
and $Single: \bigwedge x. P [x]$
and $ssnoc : \bigwedge xs\ x\ y. P (xs @ [x]) \implies P (xs @ [x, y])$
shows $P xs$

<proof>

lemma *list-induct2-snoc* [*case-names Nil1 Nil2 snoc*]:
 assumes *Nil1*: $\bigwedge ys. P \ [] \ ys$
 and *Nil2*: $\bigwedge xs. P \ xs \ []$
 and *snoc*: $\bigwedge xs \ x \ ys \ y. P \ xs \ ys \implies P \ (xs@[x]) \ (ys@[y])$
 shows $P \ xs \ ys$

<proof>

lemma *list-induct2-snoc-Cons* [*case-names Nil1 Nil2 snoc-Cons*]:
 assumes *Nil1* : $\bigwedge ys. P \ [] \ ys$
 and *Nil2* : $\bigwedge xs. P \ xs \ []$
 and *snoc-Cons*: $\bigwedge xs \ x \ y \ ys. P \ xs \ ys \implies P \ (xs@[x]) \ (y\#ys)$
 shows $P \ xs \ ys$

<proof>

lemma *prod-list-induct3-snoc-Cons-snoc-Cons-pairwise*:
 assumes $\bigwedge ys \ zs. Q \ ([],ys,zs) \ \bigwedge xs \ zs. Q \ (xs,[],zs) \ \bigwedge xs \ ys. Q \ (xs,ys,[])$
 $\bigwedge xs \ x \ y \ z \ zs. Q \ (xs@[x],[y],z\#zs)$
 and *step*:
 $\bigwedge xs \ x \ y \ ys \ w \ z \ zs. Q \ (xs,ys,zs) \implies Q \ (xs,ys@[w],z\#zs) \implies$
 $Q \ (xs@[x],y\#ys,zs) \implies Q \ (xs@[x],y\#ys@[w],z\#zs)$
 shows $Q \ t$

<proof>

lemma *list-induct3-snoc-Cons-snoc-Cons-pairwise*
 [*case-names Nil1 Nil2 Nil3 snoc-single-Cons snoc-Cons-snoc-Cons*]:
 assumes *Nil1* : $\bigwedge ys \ zs. P \ [] \ ys \ zs$
 and *Nil2* : $\bigwedge xs \ zs. P \ xs \ [] \ zs$
 and *Nil3* : $\bigwedge xs \ ys. P \ xs \ ys \ []$
 and *snoc-single-Cons* : $\bigwedge xs \ x \ y \ z \ zs. P \ (xs@[x]) \ [y] \ (z\#zs)$
 and *snoc-Cons-snoc-Cons*:
 $\bigwedge xs \ x \ y \ ys \ w \ z \ zs. P \ xs \ ys \ zs \implies P \ xs \ (ys@[w]) \ (z\#zs) \implies$
 $P \ (xs@[x]) \ (y\#ys) \ zs \implies P \ (xs@[x]) \ (y\#ys@[w]) \ (z\#zs)$
 shows $P \ xs \ ys \ zs$

<proof>

1.5.4 Alternating lists

primrec *alternating-list* :: $nat \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \text{ list}$

where *zero*: *alternating-list* 0 *s t* = []
 | *Suc* : *alternating-list* (*Suc k*) *s t* =
 alternating-list k s t @ [if even k then s else t]

— could be defined using *Cons*, but we want the alternating list to always start with the same letter as it grows, and it's easier to do that via *append*

lemma *alternating-list2*: *alternating-list* 2 *s t* = [*s,t*]
<proof>

lemma *length-alternating-list*: $\text{length } (\text{alternating-list } n \ s \ t) = n$
(proof)

lemma *alternating-list-Suc-Cons*:
 $\text{alternating-list } (\text{Suc } k) \ s \ t = s \ \# \ \text{alternating-list } k \ t \ s$
(proof)

lemma *alternating-list-SucSuc-ConsCons*:
 $\text{alternating-list } (\text{Suc } (\text{Suc } k)) \ s \ t = s \ \# \ t \ \# \ \text{alternating-list } k \ s \ t$
(proof)

lemma *alternating-list-alternates*:
 $\text{alternating-list } n \ s \ t = as@[a,b,c]@bs \implies a=c$
(proof)

lemma *alternating-list-split*:
 $\text{alternating-list } (m+n) \ s \ t = \text{alternating-list } m \ s \ t \ @$
(if even m then $\text{alternating-list } n \ s \ t$ else $\text{alternating-list } n \ t \ s$)
(proof)

lemma *alternating-list-append*:
even $m \implies$
 $\text{alternating-list } m \ s \ t \ @ \ \text{alternating-list } n \ s \ t = \text{alternating-list } (m+n) \ s \ t$
odd $m \implies$
 $\text{alternating-list } m \ s \ t \ @ \ \text{alternating-list } n \ t \ s = \text{alternating-list } (m+n) \ s \ t$
(proof)

lemma *rev-alternating-list*:
 $\text{rev } (\text{alternating-list } n \ s \ t) =$
(if even n then $\text{alternating-list } n \ t \ s$ else $\text{alternating-list } n \ s \ t$)
(proof)

lemma *set-alternating-list*: $\text{set } (\text{alternating-list } n \ s \ t) \subseteq \{s,t\}$
(proof)

lemma *set-alternating-list1*:
assumes $n \geq 1$
shows $s \in \text{set } (\text{alternating-list } n \ s \ t)$
(proof)

lemma *set-alternating-list2*:
 $n \geq 2 \implies \text{set } (\text{alternating-list } n \ s \ t) = \{s,t\}$
(proof)

lemma *alternating-list-in-lists*: $a \in A \implies b \in A \implies \text{alternating-list } n \ a \ b \in \text{lists } A$
(proof)

1.5.5 Binary relation chains

Here we consider lists where each pair of adjacent elements satisfy a given relation.

```
fun binrelchain :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  bool
  where binrelchain P [] = True
        | binrelchain P [x] = True
        | binrelchain P (x # y # xs) = (P x y  $\wedge$  binrelchain P (y#xs))
```

lemma *binrelchain-Cons-reduce*: $\text{binrelchain } P \ (x\#xs) \Longrightarrow \text{binrelchain } P \ xs$
<proof>

lemma *binrelchain-append-reduce1*: $\text{binrelchain } P \ (xs@ys) \Longrightarrow \text{binrelchain } P \ xs$
<proof>

lemma *binrelchain-append-reduce2*:
 $\text{binrelchain } P \ (xs@ys) \Longrightarrow \text{binrelchain } P \ ys$
<proof>

lemma *binrelchain-Conssnoc-reduce*:
 $\text{binrelchain } P \ (x\#xs@[y]) \Longrightarrow \text{binrelchain } P \ xs$
<proof>

lemma *binrelchain-overlap-join*:
 $\text{binrelchain } P \ (xs@[x]) \Longrightarrow \text{binrelchain } P \ (x\#ys) \Longrightarrow \text{binrelchain } P \ (xs@x\#ys)$
<proof>

lemma *binrelchain-join*:
 $[\text{binrelchain } P \ (xs@[x]); \text{binrelchain } P \ (y\#ys); P \ x \ y] \Longrightarrow$
 $\text{binrelchain } P \ (xs @ x \# y \# ys)$
<proof>

lemma *binrelchain-snoc*:
 $\text{binrelchain } P \ (xs@[x]) \Longrightarrow P \ x \ y \Longrightarrow \text{binrelchain } P \ (xs@[x,y])$
<proof>

lemma *binrelchain-sym-rev*:
assumes $\bigwedge x \ y. P \ x \ y \Longrightarrow P \ y \ x$
shows $\text{binrelchain } P \ xs \Longrightarrow \text{binrelchain } P \ (\text{rev } xs)$
<proof>

lemma *binrelchain-remdup-adj*:
 $\text{binrelchain } P \ (xs@[x,x]@ys) \Longrightarrow \text{binrelchain } P \ (xs@x\#ys)$
<proof>

abbreviation *proper-binrelchain* $P \ xs \equiv \text{binrelchain } P \ xs \wedge \text{distinct } xs$

lemma *binrelchain-obtain-proper*:
 $x \neq y \Longrightarrow \text{binrelchain } P \ (x\#xs@[y]) \Longrightarrow$

$\exists zs. \text{set } zs \subseteq \text{set } xs \wedge \text{length } zs \leq \text{length } xs \wedge \text{proper-binrelchain } P (x\#zs@[y])$
 ⟨proof⟩

lemma *binrelchain-trans-Cons-snoc*:

assumes $\bigwedge x y z. P x y \implies P y z \implies P x z$
shows $\text{binrelchain } P (x\#xs@[y]) \implies P x y$
 ⟨proof⟩

lemma *binrelchain-cong*:

assumes $\bigwedge x y. P x y \implies Q x y$
shows $\text{binrelchain } P xs \implies \text{binrelchain } Q xs$
 ⟨proof⟩

lemma *binrelchain-funcong-Cons-snoc*:

assumes $\bigwedge x y. P x y \implies f y = f x \text{ binrelchain } P (x\#xs@[y])$
shows $f y = f x$
 ⟨proof⟩

lemma *binrelchain-funcong-extra-condition-Cons-snoc*:

assumes $\bigwedge x y. Q x \implies P x y \implies Q y \wedge x y. Q x \implies P x y \implies f y = f x$
shows $Q x \implies \text{binrelchain } P (x\#zs@[y]) \implies f y = f x$
 ⟨proof⟩

lemma *binrelchain-setfuncong-Cons-snoc*:

$\llbracket \forall x \in A. \forall y. P x y \longrightarrow y \in A; \forall x \in A. \forall y. P x y \longrightarrow f y = f x; x \in A; \text{binrelchain } P (x\#zs@[y]) \rrbracket \implies f y = f x$
 ⟨proof⟩

lemma *binrelchain-propcong-Cons-snoc*:

assumes $\bigwedge x y. Q x \implies P x y \implies Q y$
shows $Q x \implies \text{binrelchain } P (x\#xs@[y]) \implies Q y$
 ⟨proof⟩

1.5.6 Set of subseqs

lemma *subseqs-Cons*: $\text{subseqs } (x\#xs) = \text{map } (\text{Cons } x) (\text{subseqs } xs) @ (\text{subseqs } xs)$
 ⟨proof⟩

abbreviation $\text{ssubseqs } xs \equiv \text{set } (\text{subseqs } xs)$

lemma *nil-ssubseqs*: $[] \in \text{ssubseqs } xs$
 ⟨proof⟩

lemma *ssubseqs-Cons*: $\text{ssubseqs } (x\#xs) = (\text{Cons } x) ` (\text{ssubseqs } xs) \cup \text{ssubseqs } xs$
 ⟨proof⟩

lemma *ssubseqs-refl*: $xs \in \text{ssubseqs } xs$
 ⟨proof⟩

lemma *ssubseqs-subset*: $as \in \text{ssubseqs } bs \implies \text{ssubseqs } as \subseteq \text{ssubseqs } bs$
 ⟨proof⟩

lemma *ssubseqs-lists*:
 $as \in \text{lists } A \implies bs \in \text{ssubseqs } as \implies bs \in \text{lists } A$
 ⟨proof⟩

lemma *delete1-ssubseqs*:
 $as@bs \in \text{ssubseqs } (as@[a]@bs)$
 ⟨proof⟩

lemma *delete2-ssubseqs*:
 $as@bs@cs \in \text{ssubseqs } (as@[a]@bs@[b]@cs)$
 ⟨proof⟩

1.6 Orders and posets

We have chosen to work with the *ordering* locale instead of the *order* class to more easily facilitate simultaneously working with both an order and its dual.

1.6.1 Morphisms of posets

locale *OrderingSetMap* =
 domain : ordering less-eq less
 + codomain: ordering less-eq' less'
 for less-eq :: 'a ⇒ 'a ⇒ bool (infix ≤ 50)
 and less :: 'a ⇒ 'a ⇒ bool (infix < 50)
 and less-eq' :: 'b ⇒ 'b ⇒ bool (infix ≤* 50)
 and less' :: 'b ⇒ 'b ⇒ bool (infix <* 50)
 + fixes P :: 'a set
 and f :: 'a ⇒ 'b
 assumes ordsetmap: $a \in P \implies b \in P \implies a \leq b \implies f a \leq* f b$
 begin

lemma *comp*:
 assumes *OrderingSetMap* less-eq' less' less-eq'' less'' Q g
 f'P ⊆ Q
 shows *OrderingSetMap* less-eq less less-eq'' less'' P (g∘f)
 ⟨proof⟩

lemma *subset*: $Q \subseteq P \implies \text{OrderingSetMap } (\leq) (<) (\leq*) (<*) Q f$
 ⟨proof⟩

end

locale *OrderingSetIso* = *OrderingSetMap* less-eq less less-eq' less' P f
 for less-eq :: 'a ⇒ 'a ⇒ bool (infix ≤ 50)
 and less :: 'a ⇒ 'a ⇒ bool (infix < 50)

```

and less-eq' :: 'b=>'b=>bool (infix ≤* 50)
and less'   :: 'b=>'b=>bool (infix <* 50)
and P      :: 'a set
and f      :: 'a=>'b
+ assumes inj          : inj-on f P
and   rev-OrderingSetMap:
      OrderingSetMap less-eq' less' less-eq less (f'P) (the-inv-into P f)

```

abbreviation subset-ordering-iso ≡ OrderingSetIso (⊆) (⊂) (⊆) (⊂)

```

lemma (in OrderingSetMap) isoI:
  assumes inj-on f P ∧ a b. a∈P ⇒ b∈P ⇒ f a ≤* f b ⇒ a ≤ b
  shows OrderingSetIso less-eq less less-eq' less' P f
  ⟨proof⟩

```

```

lemma OrderingSetIsoI-orders-greater2less:
  fixes f :: 'a::order ⇒ 'b::order
  assumes inj-on f P ∧ a b. a ∈ P ⇒ b ∈ P ⇒ (b≤a) = (f a ≤ f b)
  shows OrderingSetIso (greater-eq::'a=>'a=>bool) (greater::'a=>'a=>bool)
        (less-eq::'b=>'b=>bool) (less::'b=>'b=>bool) P f
  ⟨proof⟩

```

context OrderingSetIso
begin

lemmas ordsetmap = ordsetmap

```

lemma ordsetmap-strict: [ a∈P; b∈P; a<b ] ⇒ f a <* f b
  ⟨proof⟩

```

lemmas inv-ordsetmap = OrderingSetMap.ordsetmap[OF rev-OrderingSetMap]

```

lemma rev-ordsetmap: [ a∈P; b∈P; f a ≤* f b ] ⇒ a ≤ b
  ⟨proof⟩

```

```

lemma inv-iso: OrderingSetIso less-eq' less' less-eq less (f'P) (the-inv-into P f)
  ⟨proof⟩

```

lemmas inv-ordsetmap-strict = OrderingSetIso.ordsetmap-strict[OF inv-iso]

```

lemma rev-ordsetmap-strict: [ a∈P; b∈P; f a <* f b ] ⇒ a < b
  ⟨proof⟩

```

```

lemma iso-comp:
  assumes OrderingSetIso less-eq' less' less-eq'' less'' Q g f'P ⊆ Q
  shows OrderingSetIso less-eq less less-eq'' less'' P (g∘f)
  ⟨proof⟩

```

lemma *iso-subset*:

$Q \subseteq P \implies \text{OrderingSetIso } (\leq) (<) (\leq^*) (<^*) Q f$
<proof>

lemma *iso-dual*:

$\langle \text{OrderingSetIso } (\lambda a b. \text{less-eq } b a) (\lambda a b. \text{less } b a) (\lambda a b. \text{less-eq}' b a) (\lambda a b. \text{less}' b a) P f \rangle$
<proof>

end

lemma *induced-pow-fun-subset-ordering-iso*:

assumes *inj-on f A*
shows *subset-ordering-iso (Pow A) (($\hat{\cdot}$) f)*
<proof>

1.6.2 More *arg-min*

lemma *is-arg-minI*:

$\llbracket P x; \bigwedge y. P y \implies \neg m y < m x \rrbracket \implies \text{is-arg-min } m P x$
<proof>

lemma *is-arg-min-linorderI*:

$\llbracket P x; \bigwedge y. P y \implies m x \leq (m y :: \text{linorder}) \rrbracket \implies \text{is-arg-min } m P x$
<proof>

lemma *is-arg-min-eq*:

$\llbracket \text{is-arg-min } m P x; P z; m z = m x \rrbracket \implies \text{is-arg-min } m P z$
<proof>

lemma *is-arg-minD1*: *is-arg-min m P x $\implies P x$*

<proof>

lemma *is-arg-minD2*: *is-arg-min m P x $\implies P y \implies \neg m y < m x$*

<proof>

lemma *is-arg-min-size*: **fixes** *m :: 'a \Rightarrow 'b::linorder*

shows *is-arg-min m P x $\implies m x = m (\text{arg-min } m P)$*

<proof>

lemma *is-arg-min-size-subprop*:

fixes *m :: 'a \Rightarrow 'b::linorder*

assumes *is-arg-min m P x Q x $\bigwedge y. Q y \implies P y$*

shows *m (arg-min m Q) = m (arg-min m P)*

<proof>

1.6.3 Bottom of a set

context *ordering*

begin

definition *has-bottom* :: 'a set \Rightarrow bool

where *has-bottom* $P \equiv \exists z \in P. \forall x \in P. z \leq x$

lemma *has-bottomI*: $z \in P \Rightarrow (\bigwedge x. x \in P \Rightarrow z \leq x) \Rightarrow \text{has-bottom } P$
<proof>

lemma *has-uniq-bottom*: $\text{has-bottom } P \Rightarrow \exists! z \in P. \forall x \in P. z \leq x$
<proof>

definition *bottom* :: 'a set \Rightarrow 'a

where *bottom* $P \equiv (\text{THE } z. z \in P \wedge (\forall x \in P. z \leq x))$

lemma *bottomD*:

assumes *has-bottom* P

shows $\text{bottom } P \in P \wedge x \in P \Rightarrow \text{bottom } P \leq x$

<proof>

lemma *bottomI*: $z \in P \Rightarrow (\bigwedge y. y \in P \Rightarrow z \leq y) \Rightarrow z = \text{bottom } P$
<proof>

end

lemma *has-bottom-pow*: $\text{order.has-bottom } (\text{Pow } A)$
<proof>

lemma *bottom-pow*: $\text{order.bottom } (\text{Pow } A) = \{\}$
<proof>

context *OrderingSetMap*

begin

abbreviation *dombot* $\equiv \text{domain.bottom } P$

abbreviation *codbot* $\equiv \text{codomain.bottom } (f'P)$

lemma *im-has-bottom*: $\text{domain.has-bottom } P \Rightarrow \text{codomain.has-bottom } (f'P)$
<proof>

lemma *im-bottom*: $\text{domain.has-bottom } P \Rightarrow f \text{ dombot} = \text{codbot}$
<proof>

end

lemma (**in** *OrderingSetIso*) *pullback-has-bottom*:

assumes $\text{codomain.has-bottom } (f'P)$

shows $\text{domain.has-bottom } P$

<proof>

lemma (**in** *OrderingSetIso*) *pullback-bottom*:

[[*domain.has-bottom* P ; $x \in P$; $f\ x = \text{codomain.bottom } (f'P)$]] \implies
 $x = \text{domain.bottom } P$
 <proof>

1.6.4 Minimal and pseudominimal elements in sets

We will call an element of a poset pseudominimal if the only element below it is the bottom of the poset.

context *ordering*

begin

definition *minimal-in* :: 'a set \Rightarrow 'a \Rightarrow bool
where *minimal-in* $P\ x \equiv x \in P \wedge (\forall z \in P. \neg z < x)$

definition *pseudominimal-in* :: 'a set \Rightarrow 'a \Rightarrow bool
where *pseudominimal-in* $P\ x \equiv \text{minimal-in } (P - \{\text{bottom } P\})\ x$
 — only makes sense for *has-bottom* P

lemma *minimal-inD1*: *minimal-in* $P\ x \implies x \in P$
 <proof>

lemma *minimal-inD2*: *minimal-in* $P\ x \implies z \in P \implies \neg z < x$
 <proof>

lemma *pseudominimal-inD1*: *pseudominimal-in* $P\ x \implies x \in P$
 <proof>

lemma *pseudominimal-inD2*:
pseudominimal-in $P\ x \implies z \in P \implies z < x \implies z = \text{bottom } P$
 <proof>

lemma *pseudominimal-inI*:
assumes $x \in P\ x \neq \text{bottom } P \wedge z. z \in P \implies z < x \implies z = \text{bottom } P$
shows *pseudominimal-in* $P\ x$
 <proof>

lemma *pseudominimal-ne-bottom*: *pseudominimal-in* $P\ x \implies x \neq \text{bottom } P$
 <proof>

lemma *pseudominimal-comp*:
 [[*pseudominimal-in* $P\ x$; *pseudominimal-in* $P\ y$; $x \leq y$]] $\implies x = y$
 <proof>

end

lemma *pseudominimal-in-pow*:
assumes *order.pseudominimal-in* (*Pow* A) x
shows $\exists a \in A. x = \{a\}$
 <proof>

lemma *pseudominimal-in-pow-singleton*:
 $a \in A \implies \text{order.pseudominimal-in } (\text{Pow } A) \{a\}$
 ⟨proof⟩

lemma *no-pseudominimal-in-pow-is-empty*:
 $(\bigwedge x. \neg \text{order.pseudominimal-in } (\text{Pow } A) \{x\}) \implies A = \{\}$
 ⟨proof⟩

lemma (in *OrderingSetIso*) *pseudominimal-map*:
 $\text{domain.has-bottom } P \implies \text{domain.pseudominimal-in } P \ x \implies$
 $\text{codomain.pseudominimal-in } (f'P) \ (f \ x)$
 ⟨proof⟩

lemma (in *OrderingSetIso*) *pullback-pseudominimal-in*:
 $\llbracket \text{domain.has-bottom } P; \ x \in P; \ \text{codomain.pseudominimal-in } (f'P) \ (f \ x) \rrbracket \implies$
 $\text{domain.pseudominimal-in } P \ x$
 ⟨proof⟩

1.6.5 Set of elements below another

abbreviation (in *ordering*) *below-in* :: 'a set \Rightarrow 'a \Rightarrow 'a set (infix \leq 70)
 where $P.\leq x \equiv \{y \in P. y \leq x\}$

abbreviation (in *ord*) *below-in* :: 'a set \Rightarrow 'a \Rightarrow 'a set (infix \leq 70)
 where $P.\leq x \equiv \{y \in P. y \leq x\}$

context *ordering*
begin

lemma *below-in-refl*: $x \in P \implies x \in P.\leq x$
 ⟨proof⟩

lemma *below-in-singleton*: $x \in P \implies P.\leq x \subseteq \{y\} \implies y = x$
 ⟨proof⟩

lemma *bottom-in-below-in*: $\text{has-bottom } P \implies x \in P \implies \text{bottom } P \in P.\leq x$
 ⟨proof⟩

lemma *below-in-singleton-is-bottom*:
 $\llbracket \text{has-bottom } P; \ x \in P; \ P.\leq x = \{x\} \rrbracket \implies x = \text{bottom } P$
 ⟨proof⟩

lemma *bottom-below-in*:
 $\text{has-bottom } P \implies x \in P \implies \text{bottom } (P.\leq x) = \text{bottom } P$
 ⟨proof⟩

lemma *bottom-below-in-relative*:
 $\llbracket \text{has-bottom } (P.\leq y); \ x \in P; \ x \leq y \rrbracket \implies \text{bottom } (P.\leq x) = \text{bottom } (P.\leq y)$

<proof>

lemma *has-bottom-pseudominimal-in-below-inI:*

assumes *has-bottom P x ∈ P pseudominimal-in P y y ≤ x*

shows *pseudominimal-in (P. ≤ x) y*

<proof>

lemma *has-bottom-pseudominimal-in-below-in:*

assumes *has-bottom P x ∈ P pseudominimal-in (P. ≤ x) y*

shows *pseudominimal-in P y*

<proof>

lemma *pseudominimal-in-below-in:*

assumes *has-bottom (P. ≤ y) x ∈ P x ≤ y pseudominimal-in (P. ≤ x) w*

shows *pseudominimal-in (P. ≤ y) w*

<proof>

lemma *collect-pseudominimals-below-in-less-eq-top:*

assumes *OrderingSetIso less-eq less (⊆) (⊂) (P. ≤ x) f*

f'(P. ≤ x) = Pow A a ⊆ {y. pseudominimal-in (P. ≤ x) y}

defines *w ≡ the-inv-into (P. ≤ x) f (⋃(f'a))*

shows *w ≤ x*

<proof>

lemma *collect-pseudominimals-below-in-poset:*

assumes *OrderingSetIso less-eq less (⊆) (⊂) (P. ≤ x) f*

f'(P. ≤ x) = Pow A

a ⊆ {y. pseudominimal-in (P. ≤ x) y}

defines *w ≡ the-inv-into (P. ≤ x) f (⋃(f'a))*

shows *w ∈ P*

<proof>

lemma *collect-pseudominimals-below-in-eq:*

assumes *x ∈ P OrderingSetIso less-eq less (⊆) (⊂) (P. ≤ x) f*

f'(P. ≤ x) = Pow A a ⊆ {y. pseudominimal-in (P. ≤ x) y}

defines *w: w ≡ the-inv-into (P. ≤ x) f (⋃(f'a))*

shows *a = {y. pseudominimal-in (P. ≤ w) y}*

<proof>

end

1.6.6 Lower bounds

context *ordering*

begin

definition *lbound-of :: 'a ⇒ 'a ⇒ 'a ⇒ bool*

where *lbound-of x y b ≡ b ≤ x ∧ b ≤ y*

lemma *lbound-ofI*: $b \leq x \implies b \leq y \implies \text{lbound-of } x \ y \ b$
<proof>

lemma *lbound-ofD1*: $\text{lbound-of } x \ y \ b \implies b \leq x$
<proof>

lemma *lbound-ofD2*: $\text{lbound-of } x \ y \ b \implies b \leq y$
<proof>

definition *glbound-in-of* :: 'a set \Rightarrow 'a \Rightarrow 'a \Rightarrow bool
where *glbound-in-of* $P \ x \ y \ b \equiv$
 $b \in P \wedge \text{lbound-of } x \ y \ b \wedge (\forall a \in P. \text{lbound-of } x \ y \ a \longrightarrow a \leq b)$

lemma *glbound-in-ofI*:
 $\llbracket b \in P; \text{lbound-of } x \ y \ b; \bigwedge a. a \in P \implies \text{lbound-of } x \ y \ a \implies a \leq b \rrbracket \implies$
glbound-in-of $P \ x \ y \ b$
<proof>

lemma *glbound-in-ofD-in*: $\text{glbound-in-of } P \ x \ y \ b \implies b \in P$
<proof>

lemma *glbound-in-ofD-lbound*: $\text{glbound-in-of } P \ x \ y \ b \implies \text{lbound-of } x \ y \ b$
<proof>

lemma *glbound-in-ofD-glboud*:
 $\text{glbound-in-of } P \ x \ y \ b \implies a \in P \implies \text{lbound-of } x \ y \ a \implies a \leq b$
<proof>

lemma *glbound-in-of-less-eq1*: $\text{glbound-in-of } P \ x \ y \ b \implies b \leq x$
<proof>

lemma *glbound-in-of-less-eq2*: $\text{glbound-in-of } P \ x \ y \ b \implies b \leq y$
<proof>

lemma *pseudominimal-in-below-in-less-eq-glboud*:
assumes *pseudominimal-in* ($P. \leq x$) *w pseudominimal-in* ($P. \leq y$) *w*
glbound-in-of $P \ x \ y \ b$
shows $w \leq b$
<proof>

end

1.6.7 Simplex-like posets

Define a poset to be simplex-like if it is isomorphic to the power set of some set.

context *ordering*
begin

definition *simplex-like* :: 'a set \Rightarrow bool
where *simplex-like* $P \equiv$ finite $P \wedge$
 $(\exists f A::\text{nat set.}$
 $\text{OrderingSetIso less-eq less } (\subseteq) (\subset) P f \wedge f'P = \text{Pow } A$
 $)$

lemma *simplex-likeI*:
assumes finite P *OrderingSetIso less-eq less* $(\subseteq) (\subset) P f$
 $f'P = \text{Pow } (A::\text{nat set})$
shows *simplex-like* P
 $\langle \text{proof} \rangle$

lemma *simplex-likeD-finite*: *simplex-like* $P \Rightarrow$ finite P
 $\langle \text{proof} \rangle$

lemma *simplex-likeD-iso*:
simplex-like $P \Rightarrow$
 $\exists f A::\text{nat set. OrderingSetIso less-eq less } (\subseteq) (\subset) P f \wedge f'P = \text{Pow } A$
 $\langle \text{proof} \rangle$

lemma *simplex-like-has-bottom*: *simplex-like* $P \Rightarrow$ has-bottom P
 $\langle \text{proof} \rangle$

lemma *simplex-like-no-pseudominimal-imp-singleton*:
assumes *simplex-like* $P \wedge x. \neg$ pseudominimal-in $P x$
shows $\exists p. P = \{p\}$
 $\langle \text{proof} \rangle$

lemma *simplex-like-no-pseudominimal-in-below-in-imp-singleton*:
 $\llbracket x \in P; \text{simplex-like } (P.\leq x); \wedge z. \neg$ pseudominimal-in $(P.\leq x) z \rrbracket \Rightarrow$
 $P.\leq x = \{x\}$
 $\langle \text{proof} \rangle$

lemma *pseudo-simplex-like-has-bottom*:
OrderingSetIso less-eq less $(\subseteq) (\subset) P f \Rightarrow f'P = \text{Pow } A \Rightarrow$
has-bottom P
 $\langle \text{proof} \rangle$

lemma *pseudo-simplex-like-above-pseudominimal-is-top*:
assumes *OrderingSetIso less-eq less* $(\subseteq) (\subset) P f f'P = \text{Pow } A t \in P$
 $\wedge x. \text{pseudominimal-in } P x \Rightarrow x \leq t$
shows $f t = A$
 $\langle \text{proof} \rangle$

lemma *pseudo-simplex-like-below-in-above-pseudominimal-is-top*:
assumes $x \in P$ *OrderingSetIso less-eq less* $(\subseteq) (\subset) (P.\leq x) f$
 $f'(P.\leq x) = \text{Pow } A t \in P.\leq x$
 $\wedge y. \text{pseudominimal-in } (P.\leq x) y \Rightarrow y \leq t$
shows $t = x$

<proof>

lemma *simplex-like-below-in-above-pseudominimal-is-top:*

assumes $x \in P$ *simplex-like* ($P.\leq x$) $t \in P.\leq x$

$\wedge y.$ *pseudominimal-in* ($P.\leq x$) $y \implies y \leq t$

shows $t = x$

<proof>

end

lemma (**in** *OrderingSetIso*) *simplex-like-map:*

assumes *domain.simplex-like* P

shows *codomain.simplex-like* ($f \cdot P$)

<proof>

lemma (**in** *OrderingSetIso*) *pullback-simplex-like:*

assumes *finite* P *codomain.simplex-like* ($f \cdot P$)

shows *domain.simplex-like* P

<proof>

lemma *simplex-like-pow:*

assumes *finite* A

shows *order.simplex-like* ($Pow A$)

<proof>

1.6.8 The superset ordering

abbreviation *supset-has-bottom* \equiv *ordering.has-bottom* (\supseteq)

abbreviation *supset-bottom* \equiv *ordering.bottom* (\supseteq)

abbreviation *supset-lbound-of* \equiv *ordering.lbound-of* (\supseteq)

abbreviation *supset-glbound-in-of* \equiv *ordering.glbound-in-of* (\supseteq)

abbreviation *supset-simplex-like* \equiv *ordering.simplex-like* (\supseteq) (\supset)

abbreviation *supset-pseudominimal-in* \equiv
ordering.pseudominimal-in (\supseteq) (\supset)

abbreviation *supset-below-in* $:: 'a$ set $set \Rightarrow 'a$ set $\Rightarrow 'a$ set set (**infix** \supseteq 70)

where $P.\supseteq A \equiv$ *ordering.below-in* (\supseteq) $P A$

lemma *supset-poset:* *ordering* (\supseteq) (\supset) *<proof>*

lemmas *supset-bottomI* $=$ *ordering.bottomI* [*OF supset-poset*]

lemmas *supset-pseudominimal-inI* $=$ *ordering.pseudominimal-inI* [*OF supset-poset*]

lemmas *supset-pseudominimal-inD1* $=$ *ordering.pseudominimal-inD1* [*OF supset-poset*]

lemmas *supset-pseudominimal-inD2* $=$ *ordering.pseudominimal-inD2* [*OF supset-poset*]

lemmas *supset-lbound-ofI* $=$ *ordering.lbound-ofI* [*OF supset-poset*]

lemmas *supset-lbound-of-def* $=$ *ordering.lbound-of-def* [*OF supset-poset*]

lemmas *supset-glbound-in-ofI* $=$ *ordering.glbound-in-ofI* [*OF supset-poset*]

lemmas *supset-pseudominimal-ne-bottom* $=$
ordering.pseudominimal-ne-bottom [*OF supset-poset*]

lemmas *supset-has-bottom-pseudominimal-in-below-inI* =
ordering.has-bottom-pseudominimal-in-below-inI[*OF supset-poset*]

lemmas *supset-has-bottom-pseudominimal-in-below-in* =
ordering.has-bottom-pseudominimal-in-below-in[*OF supset-poset*]

lemma *OrderingSetIso-pow-complement*:
OrderingSetIso (\supseteq) (\supset) (\subseteq) (\subset) (*Pow A*) ($(-)$ *A*)
 ⟨*proof*⟩

lemma *simplex-like-pow-above-in*:
assumes *finite A X ⊆ A*
shows *supset-simplex-like ((Pow A).⊇X)*
 ⟨*proof*⟩

end

2 Algebra

In this section, we develop the necessary algebra for developing the theory of Coxeter systems, including groups, quotient groups, free groups, group presentations, and words in a group over a set of generators.

theory *Algebra*
imports *Prelim*

begin

2.1 Miscellaneous algebra facts

lemma *times2-conv-add*: $(j::nat) + j = 2*j$
 ⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *odd-n0*: $odd\ m \implies m \neq 0$
 ⟨*proof*⟩

lemma (**in** *semigroup-add*) *add-assoc4*: $a + b + c + d = a + (b + c + d)$
 ⟨*proof*⟩

lemmas (**in** *monoid-add*) *sum-list-map-cong* =
arg-cog[*OF map-cong*, *OF refl*, *of - - - sum-list*]

context *group-add*
begin

lemma *map-uminus-order2*:
 $\forall s \in set\ ss.\ s + s = 0 \implies map\ (uminus)\ ss = ss$
 ⟨*proof*⟩

lemma *uminus-sum-list*: $- sum-list\ as = sum-list\ (map\ uminus\ (rev\ as))$

<proof>

lemma *uminus-sum-list-order2*:

$\forall s \in \text{set } ss. s + s = 0 \implies - \text{sum-list } ss = \text{sum-list } (\text{rev } ss)$

<proof>

end

2.2 The type of permutations of a type

Here we construct a type consisting of all bijective functions on a type. This is the prototypical example of a group, where the group operation is composition, and every group can be embedded into such a type. It is for this purpose that we construct this type, so that we may confer upon suitable subsets of types that are not of class *group-add* the properties of that class, via a suitable injective correspondence to this permutation type.

typedef *'a permutation* = {*f*::'*a*⇒'*a*. *bij f*}
morphisms *permutation Abs-permutation*
<proof>

setup-lifting *type-definition-permutation*

abbreviation *permutation-apply* :: '*a permutation* ⇒ '*a* ⇒ '*a* (**infixr** → 90)
where *p* → *a* ≡ *permutation p a*

abbreviation *permutation-image* :: '*a permutation* ⇒ '*a set* ⇒ '*a set*
(**infixr** '→ 90)
where *p* '→ *A* ≡ *permutation p 'A*

lemma *permutation-eq-image*: *a* '→ *A* = *a* '→ *B* ⇒ *A=B*
<proof>

instantiation *permutation* :: (*type*) *zero*

begin

lift-definition *zero-permutation* :: '*a permutation* **is** *id*::'*a*⇒'*a* *<proof>*

instance *<proof>*

end

instantiation *permutation* :: (*type*) *plus*

begin

lift-definition *plus-permutation* :: '*a permutation* ⇒ '*a permutation* ⇒ '*a permutation*

is *comp*

<proof>

instance *<proof>*

end

lemma *plus-permutation-abs-eq*:

bij f ⇒ *bij g* ⇒

Abs-permutation $f + \text{Abs-permutation } g = \text{Abs-permutation } (f \circ g)$
 ⟨proof⟩

instance *permutation* :: (type) semigroup-add
 ⟨proof⟩

instance *permutation* :: (type) monoid-add
 ⟨proof⟩

instantiation *permutation* :: (type) uminus
begin
lift-definition *uminus-permutation* :: 'a permutation \Rightarrow 'a permutation
 is $\lambda f. \text{the-inv } f$
 ⟨proof⟩
instance ⟨proof⟩
end

instantiation *permutation* :: (type) minus
begin
lift-definition *minus-permutation* :: 'a permutation \Rightarrow 'a permutation \Rightarrow 'a permutation
 is $\lambda f g. f \circ (\text{the-inv } g)$
 ⟨proof⟩
instance ⟨proof⟩
end

lemma *minus-permutation-abs-eq*:
 $\text{bij } f \Rightarrow \text{bij } g \Rightarrow$
 $\text{Abs-permutation } f - \text{Abs-permutation } g = \text{Abs-permutation } (f \circ \text{the-inv } g)$
 ⟨proof⟩

instance *permutation* :: (type) group-add
 ⟨proof⟩

2.3 Natural action of *nat* on types of class *monoid-add*

2.3.1 Translation from class *power*.

Here we translate the *power* class to apply to types of class *monoid-add*.

context *monoid-add*
begin

sublocale *nataction*: *power 0 plus* ⟨proof⟩
sublocale *add-mult-translate*: *monoid-mult 0 plus*
 ⟨proof⟩

abbreviation *nataction* :: 'a \Rightarrow nat \Rightarrow 'a (**infix** $+\hat{80}$)
 where $a+\hat{n} \equiv \text{nataction.power } a \ n$

lemmas *nataction-2* = *add-mult-translate.power2-eq-square*
lemmas *nataction-Suc2* = *add-mult-translate.power-Suc2*

lemma *alternating-sum-list-conv-nataction*:
*sum-list (alternating-list (2*n) s t) = (s+t)+ \hat{n}*
<proof>

lemma *nataction-add-flip*: $(a+b)+\hat{\text{Suc } n} = a + (b+a)+\hat{n} + b$
<proof>

end

lemma (*in group-add*) *nataction-add-eq0-flip*:
assumes $(a+b)+\hat{n} = 0$
shows $(b+a)+\hat{n} = 0$
<proof>

2.3.2 Additive order of an element

context *monoid-add*
begin

definition *add-order* :: '*a* \Rightarrow *nat*
where *add-order a* \equiv *if* $(\exists n>0. a+\hat{n} = 0)$ *then*
(LEAST n. n>0 \wedge a+ $\hat{n} = 0$) else 0

lemma *add-order*: $a+\hat{(\text{add-order } a)} = 0$
<proof>

lemma *add-order-least*: $n>0 \implies a+\hat{n} = 0 \implies \text{add-order } a \leq n$
<proof>

lemma *add-order-equality*:
 $\llbracket n>0; a+\hat{n} = 0; (\bigwedge m. m>0 \implies a+\hat{m} = 0 \implies n \leq m) \rrbracket \implies$
 $\text{add-order } a = n$
<proof>

lemma *add-order0*: $\text{add-order } 0 = 1$
<proof>

lemma *add-order-gt0*: $(\text{add-order } a > 0) = (\exists n>0. a+\hat{n} = 0)$
<proof>

lemma *add-order-eq0*: $\text{add-order } a = 0 \implies n>0 \implies a+\hat{n} \neq 0$
<proof>

lemma *less-add-order-eq-0*:
assumes $a+\hat{k} = 0$ $k < \text{add-order } a$
shows $k = 0$

<proof>

lemma *less-add-order-eq-0-contr*: $k > 0 \implies k < \text{add-order } a \implies a + \hat{k} \neq 0$
<proof>

lemma *add-order-relator*: $\text{add-order } (a + \widehat{(\text{add-order } a)}) = 1$
<proof>

abbreviation *pair-relator-list* :: $'a \Rightarrow 'a \Rightarrow 'a \text{ list}$

where *pair-relator-list* $s\ t \equiv \text{alternating-list } (2 * \text{add-order } (s + t))\ s\ t$

abbreviation *pair-relator-halflist* :: $'a \Rightarrow 'a \Rightarrow 'a \text{ list}$

where *pair-relator-halflist* $s\ t \equiv \text{alternating-list } (\text{add-order } (s + t))\ s\ t$

abbreviation *pair-relator-halflist2* :: $'a \Rightarrow 'a \Rightarrow 'a \text{ list}$

where *pair-relator-halflist2* $s\ t \equiv$

*(if even (add-order (s + t)) then pair-relator-halflist s t else
pair-relator-halflist t s)*

lemma *sum-list-pair-relator-list*: $\text{sum-list } (\text{pair-relator-list } s\ t) = 0$
<proof>

end

context *group-add*

begin

lemma *add-order-add-eq1*: $\text{add-order } (s + t) = 1 \implies t = -s$
<proof>

lemma *add-order-add-sym*: $\text{add-order } (t + s) = \text{add-order } (s + t)$
<proof>

lemma *pair-relator-halflist-append*:

pair-relator-halflist $s\ t @ \text{pair-relator-halflist2 } s\ t = \text{pair-relator-list } s\ t$

<proof>

lemma *rev-pair-relator-list*: $\text{rev } (\text{pair-relator-list } s\ t) = \text{pair-relator-list } t\ s$
<proof>

lemma *pair-relator-halflist2-conv-rev-pair-relator-halflist*:

pair-relator-halflist2 $s\ t = \text{rev } (\text{pair-relator-halflist } t\ s)$

<proof>

end

2.4 Partial sums of a list

Here we construct a list that collects the results of adding the elements of a given list together one-by-one.

context *monoid-add*

begin

primrec *sums* :: 'a list \Rightarrow 'a list

where

$sums [] = [0]$
 $| sums (x\#xs) = 0 \# map ((+) x) (sums xs)$

lemma *length-sums*: $length (sums xs) = Suc (length xs)$
<proof>

lemma *sums-snoc*: $sums (xs@[x]) = sums xs @ [sum-list (xs@[x])]$
<proof>

lemma *sums-append2*:
 $sums (xs@ys) = butlast (sums xs) @ map ((+) (sum-list xs)) (sums ys)$
<proof>

lemma *sums-Cons-conv-append-tl*:
 $sums (x\#xs) = 0 \# x \# map ((+) x) (tl (sums xs))$
<proof>

lemma *pullback-sums-map-middle2*:
 $map F (sums xs) = ds@[d,e]@es \Longrightarrow$
 $\exists as a bs. xs = as@[a]@bs \wedge map F (sums as) = ds@[d] \wedge$
 $d = F (sum-list as) \wedge e = F (sum-list (as@[a]))$
<proof>

lemma *pullback-sums-map-middle3*:
 $map F (sums xs) = ds@[d,e,f]@fs \Longrightarrow$
 $\exists as a b bs. xs = as@[a,b]@bs \wedge d = F (sum-list as) \wedge$
 $e = F (sum-list (as@[a])) \wedge f = F (sum-list (as@[a,b]))$
<proof>

lemma *pullback-sums-map-double-middle2*:
assumes $map F (sums xs) = ds@[d,e]@es@[f,g]@gs$
shows $\exists as a bs b cs. xs = as@[a]@bs@[b]@cs \wedge d = F (sum-list as) \wedge$
 $e = F (sum-list (as@[a])) \wedge f = F (sum-list (as@[a]@bs)) \wedge$
 $g = F (sum-list (as@[a]@bs@[b]))$
<proof>

end

2.5 Sums of alternating lists

lemma (**in** *group-add*) *uminus-sum-list-alternating-order2*:
 $s+s=0 \Longrightarrow t+t=0 \Longrightarrow - sum-list (alternating-list n s t) =$
 $sum-list (if even n then alternating-list n t s else alternating-list n s t)$
<proof>

context *monoid-add*
begin

lemma *alternating-order2-cancel-1left*:

$s+s=0 \implies$
 $sum-list (s \# (alternating-list (Suc n) s t)) = sum-list (alternating-list n t s)$
<proof>

lemma *alternating-order2-cancel-2left*:

$s+s=0 \implies t+t=0 \implies$
 $sum-list (t \# s \# (alternating-list (Suc (Suc n)) s t)) =$
 $sum-list (alternating-list n s t)$
<proof>

lemma *alternating-order2-even-cancel-right*:

assumes $st : s+s=0 \ t+t=0$
and $even-n: even\ n$
shows $m \leq n \implies sum-list (alternating-list\ n\ s\ t\ @\ alternating-list\ m\ t\ s) =$
 $sum-list (alternating-list (n-m) s t)$
<proof>

end

2.6 Conjugation in *group-add*

2.6.1 Abbreviations and basic facts

context *group-add*
begin

abbreviation $lconjby :: 'a \Rightarrow 'a \Rightarrow 'a$
where $lconjby\ x\ y \equiv x+y-x$

abbreviation $rconjby :: 'a \Rightarrow 'a \Rightarrow 'a$
where $rconjby\ x\ y \equiv -x+y+x$

lemma *lconjby-add*: $lconjby (x+y) z = lconjby x (lconjby y z)$
<proof>

lemma *rconjby-add*: $rconjby (x+y) z = rconjby y (rconjby x z)$
<proof>

lemma *add-rconjby*: $rconjby x y + rconjby x z = rconjby x (y+z)$
<proof>

lemma *lconjby-uminus*: $lconjby x (-y) = - lconjby x y$
<proof>

lemma *rconjby-uminus*: $rconjby x (-y) = - rconjby x y$
<proof>

lemma *lconjby-rconjby*: $lconjby\ x\ (rconjby\ x\ y) = y$
 ⟨proof⟩

lemma *rconjby-lconjby*: $rconjby\ x\ (lconjby\ x\ y) = y$
 ⟨proof⟩

lemma *lconjby-inj*: $inj\ (lconjby\ x)$
 ⟨proof⟩

lemma *rconjby-inj*: $inj\ (rconjby\ x)$
 ⟨proof⟩

lemma *lconjby-surj*: $surj\ (lconjby\ x)$
 ⟨proof⟩

lemma *lconjby-bij*: $bij\ (lconjby\ x)$
 ⟨proof⟩

lemma *the-inv-lconjby*: $the\ inv\ (lconjby\ x) = (rconjby\ x)$
 ⟨proof⟩

lemma *lconjby-eq-conv-rconjby-eq*: $w = lconjby\ x\ y \implies y = rconjby\ x\ w$
 ⟨proof⟩

lemma *rconjby-order2*: $s+s = 0 \implies rconjby\ x\ s + rconjby\ x\ s = 0$
 ⟨proof⟩

lemma *rconjby-order2-eq-lconjby*:

assumes $s+s=0$

shows $rconjby\ s = lconjby\ s$

⟨proof⟩

lemma *lconjby-alternating-list-order2*:

assumes $s+s=0\ t+t=0$

shows $lconjby\ (sum\ list\ (alternating\ list\ k\ s\ t))\ (if\ even\ k\ then\ s\ else\ t) =$
 $sum\ list\ (alternating\ list\ (Suc\ (2*k))\ s\ t)$

⟨proof⟩

end

2.6.2 The conjugation sequence

Given a list in *group-add*, we create a new list by conjugating each term by all the previous terms. This sequence arises in Coxeter systems.

context *group-add*

begin

primrec *lconjseq* :: $'a\ list \Rightarrow 'a\ list$

where

$lconjseq [] = []$
 $| lconjseq (x\#xs) = x \# (map (lconjby x) (lconjseq xs))$

lemma *length-lconjseq*: $length (lconjseq xs) = length xs$
<proof>

lemma *lconjseq-snoc*: $lconjseq (xs@[x]) = lconjseq xs @ [lconjby (sum-list xs) x]$
<proof>

lemma *lconjseq-append*:
 $lconjseq (xs@ys) = lconjseq xs @ (map (lconjby (sum-list xs)) (lconjseq ys))$
<proof>

lemma *lconjseq-alternating-order2-repeats'*:
fixes $s t :: 'a$
defines $altst: altst \equiv \lambda n. alternating-list\ n\ s\ t$
and $altts: altts \equiv \lambda n. alternating-list\ n\ t\ s$
assumes $st : s+s=0\ t+t=0\ (s+t)+\hat{k} = 0$
shows $map (lconjby (sum-list (altst\ k)))$
 $(lconjseq (if\ even\ k\ then\ altst\ m\ else\ altts\ m)) = lconjseq (altst\ m)$
<proof>

lemma *lconjseq-alternating-order2-repeats*:
fixes $s t :: 'a$ **and** $k :: nat$
defines $altst: altst \equiv \lambda n. alternating-list\ n\ s\ t$
and $altts: altts \equiv \lambda n. alternating-list\ n\ t\ s$
assumes $st: s+s=0\ t+t=0\ (s+t)+\hat{k} = 0$
shows $lconjseq (altst (2*k)) = lconjseq (altst\ k) @ lconjseq (altst\ k)$
<proof>

lemma *even-count-lconjseq-alternating-order2*:
fixes $s t :: 'a$
assumes $s+s=0\ t+t=0\ (s+t)+\hat{k} = 0$
shows $even (count-list (lconjseq (alternating-list (2*k) s t)) x)$
<proof>

lemma *order2-hd-in-lconjseq-deletion*:
shows $s+s=0 \implies s \in set (lconjseq\ ss)$
 $\implies \exists as\ b\ bs. ss = as@[b]@bs \wedge sum-list (s\#\ss) = sum-list (as@bs)$
<proof>

end

2.6.3 The action on signed group-add elements

Here we construct an action of a group on itself by conjugation, where group elements are endowed with an auxiliary sign by pairing with a boolean element. In multiple applications of this action, the auxiliary sign helps keep

track of how many times the elements conjugating and being conjugated are the same. This action arises in exploring reduced expressions of group elements as words in a set of generators of order two (in particular, in a Coxeter group).

type-synonym $'a \text{ signed} = 'a \times \text{bool}$

definition $\text{signed-funaction} :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \text{ signed} \Rightarrow 'a \text{ signed}$
where $\text{signed-funaction } f \ s \ x \equiv \text{map-prod } (f \ s) \ (\lambda b. b \neq (\text{fst } x = s)) \ x$
— so the sign of x is flipped precisely when its first component is equal to s

context group-add
begin

abbreviation $\text{signed-lconjunction} \equiv \text{signed-funaction } \text{lconjby}$
abbreviation $\text{signed-rconjunction} \equiv \text{signed-funaction } \text{rconjby}$

lemmas $\text{signed-lconjunctionD} = \text{signed-funaction-def}[\text{of } \text{lconjby}]$
lemmas $\text{signed-rconjunctionD} = \text{signed-funaction-def}[\text{of } \text{rconjby}]$

abbreviation $\text{signed-lconjpermutation} :: 'a \Rightarrow 'a \text{ signed permutation}$
where $\text{signed-lconjpermutation } s \equiv \text{Abs-permutation } (\text{signed-lconjunction } s)$

abbreviation $\text{signed-list-lconjunction} :: 'a \text{ list} \Rightarrow 'a \text{ signed} \Rightarrow 'a \text{ signed}$
where $\text{signed-list-lconjunction } ss \equiv \text{foldr } \text{signed-lconjunction } ss$

lemma $\text{signed-lconjunction-fst: } \text{fst } (\text{signed-lconjunction } s \ x) = \text{lconjby } s \ (\text{fst } x)$
 $\langle \text{proof} \rangle$

lemma $\text{signed-lconjunction-rconjunction:}$
 $\text{signed-lconjunction } s \ (\text{signed-rconjunction } s \ x) = x$
 $\langle \text{proof} \rangle$

lemma $\text{signed-rconjunction-by-order2-eq-lconjunction:}$
 $s + s = 0 \implies \text{signed-rconjunction } s = \text{signed-lconjunction } s$
 $\langle \text{proof} \rangle$

lemma $\text{inj-signed-lconjunction: } \text{inj } (\text{signed-lconjunction } s)$
 $\langle \text{proof} \rangle$

lemma $\text{surj-signed-lconjunction: } \text{surj } (\text{signed-lconjunction } s)$
 $\langle \text{proof} \rangle$

lemma $\text{bij-signed-lconjunction: } \text{bij } (\text{signed-lconjunction } s)$
 $\langle \text{proof} \rangle$

lemma $\text{the-inv-signed-lconjunction:}$
 $\text{the-inv } (\text{signed-lconjunction } s) = \text{signed-rconjunction } s$
 $\langle \text{proof} \rangle$

lemma *the-inv-signed-lconjunction-by-order2*:
 $s+s=0 \implies \text{the-inv } (\text{signed-lconjunction } s) = \text{signed-lconjunction } s$
 ⟨proof⟩

lemma *signed-list-lconjunction-fst*:
 $\text{fst } (\text{signed-list-lconjunction } ss \ x) = \text{lconjby } (\text{sum-list } ss) (\text{fst } x)$
 ⟨proof⟩

lemma *signed-list-lconjunction-snd*:
shows $\forall s \in \text{set } ss. s+s=0 \implies \text{snd } (\text{signed-list-lconjunction } ss \ x)$
 $= (\text{if even } (\text{count-list } (\text{lconjseq } (\text{rev } ss))) (\text{fst } x)) \text{ then } \text{snd } x \text{ else } \neg \text{snd } x$
 ⟨proof⟩

end

2.7 Cosets

2.7.1 Basic facts

lemma *set-zero-plus'* [simp]: $(0::'a::\text{monoid-add}) +_o C = C$
 — lemma *Set-Algebras.set-zero-plus* is restricted to types of class *comm-monoid-add*;
 here is a version in *monoid-add*.
 ⟨proof⟩

lemma *lcaset-0*: $(w::'a::\text{monoid-add}) +_o 0 = \{w\}$
 ⟨proof⟩

lemma *lcaset-reft*: $(0::'a::\text{monoid-add}) \in A \implies a \in a +_o A$
 ⟨proof⟩

lemma *lcaset-eq-reps-subset*:
 $(a::'a::\text{group-add}) +_o A \subseteq a +_o B \implies A \subseteq B$
 ⟨proof⟩

lemma *lcaset-eq-reps*: $(a::'a::\text{group-add}) +_o A = a +_o B \implies A = B$
 ⟨proof⟩

lemma *lcaset-inj-on*: $\text{inj } ((+_o) (a::'a::\text{group-add}))$
 ⟨proof⟩

lemma *lcaset-conv-set*: $(a::'g::\text{group-add}) \in b +_o A \implies -b + a \in A$
 ⟨proof⟩

2.7.2 The supset order on cosets

lemma *supset-lbound-lcaset-shift*:
 $\text{supset-lbound-of } X \ Y \ B \implies$
 $\text{ordering.lbound-of } (\supseteq) (a +_o X) (a +_o Y) (a +_o B)$
 ⟨proof⟩

lemma *supset-glboun-d-in-of-lcoset-shift*:
fixes $P :: 'a::group-add\ set\ set$
assumes *supset-glboun-d-in-of* $P\ X\ Y\ B$
shows *supset-glboun-d-in-of* $((+o)\ a\ 'P)\ (a\ +o\ X)\ (a\ +o\ Y)\ (a\ +o\ B)$
 $\langle proof \rangle$

2.7.3 The afforded partition

definition *lcoset-rel* $:: 'a::\{uminus,plus\}\ set\ \Rightarrow\ ('a\times'a)\ set$
where *lcoset-rel* $A \equiv \{(x,y). -x + y \in A\}$

lemma *lcoset-relI*: $-x+y \in A \Longrightarrow (x,y) \in lcoset-rel\ A$
 $\langle proof \rangle$

2.8 Groups

We consider groups as closed sets in a type of class *group-add*.

2.8.1 Locale definition and basic facts

locale *Group* =
fixes $G :: 'g::group-add\ set$
assumes *nonempty* $: G \neq \{\}$
and *diff-closed*: $\bigwedge g\ h. g \in G \Longrightarrow h \in G \Longrightarrow g - h \in G$
begin

abbreviation *Subgroup* $:: 'g\ set\ \Rightarrow\ bool$
where *Subgroup* $H \equiv Group\ H \wedge H \subseteq G$

lemma *SubgroupD1*: *Subgroup* $H \Longrightarrow Group\ H\ \langle proof \rangle$

lemma *zero-closed* : $0 \in G$
 $\langle proof \rangle$

lemma *uminus-closed*: $g \in G \Longrightarrow -g \in G$
 $\langle proof \rangle$

lemma *add-closed*: $g \in G \Longrightarrow h \in G \Longrightarrow g+h \in G$
 $\langle proof \rangle$

lemma *uminus-add-closed*: $g \in G \Longrightarrow h \in G \Longrightarrow -g + h \in G$
 $\langle proof \rangle$

lemma *lconjby-closed*: $g \in G \Longrightarrow x \in G \Longrightarrow lconjby\ g\ x \in G$
 $\langle proof \rangle$

lemma *lconjby-set-closed*: $g \in G \Longrightarrow A \subseteq G \Longrightarrow lconjby\ g\ 'A \subseteq G$
 $\langle proof \rangle$

lemma *set-lconjby-subset-closed*:

$H \subseteq G \implies A \subseteq G \implies (\bigcup h \in H. \text{lconjby } h \text{ ' } A) \subseteq G$
 ⟨proof⟩

lemma *sum-list-map-closed*: $\text{set } (\text{map } f \text{ as}) \subseteq G \implies (\sum a \leftarrow \text{as}. f a) \in G$

⟨proof⟩

lemma *sum-list-closed*: $\text{set } \text{as} \subseteq G \implies \text{sum-list } \text{as} \in G$

⟨proof⟩

end

2.8.2 Sets with a suitable binary operation

We have chosen to only consider groups in types of class *group-add* so that we can take advantage of all the algebra lemmas already proven in *HOL.Groups*, as well as constructs like *sum-list*. The following locale builds a bridge between this restricted view of groups and the usual notion of a binary operation on a set satisfying the group axioms, by constructing an injective map into type *permutation* (which is of class *group-add* with respect to the composition operation) that respects the group operation. This bridge will be necessary to define quotient groups, in particular.

locale *BinOpSetGroup* =

fixes G :: 'a set

and $\text{binop} :: 'a \Rightarrow 'a \Rightarrow 'a$

and $e :: 'a$

assumes $\text{closed} : g \in G \implies h \in G \implies \text{binop } g \ h \in G$

and $\text{assoc} :$

$\llbracket g \in G; h \in G; k \in G \rrbracket \implies \text{binop } (\text{binop } g \ h) \ k = \text{binop } g \ (\text{binop } h \ k)$

and $\text{identity}: e \in G \ g \in G \implies \text{binop } g \ e = g \ g \in G \implies \text{binop } e \ g = g$

and $\text{inverses}: g \in G \implies \exists h \in G. \text{binop } g \ h = e \wedge \text{binop } h \ g = e$

begin

lemma *unique-identity1*: $g \in G \implies \forall x \in G. \text{binop } g \ x = x \implies g = e$

⟨proof⟩

lemma *unique-inverse*:

assumes $g \in G$

shows $\exists ! h. h \in G \wedge \text{binop } g \ h = e \wedge \text{binop } h \ g = e$

⟨proof⟩

abbreviation $G\text{-perm } g \equiv \text{restrict1 } (\text{binop } g) \ G$

definition $\text{Abs-}G\text{-perm} :: 'a \Rightarrow 'a \text{ permutation}$

where $\text{Abs-}G\text{-perm } g \equiv \text{Abs-permutation } (G\text{-perm } g)$

abbreviation $\mathbf{p} \equiv \text{Abs-}G\text{-perm}$ — the injection into type *permutation*

abbreviation $\mathbf{ip} \equiv \text{the-inv-into } G \ \mathbf{p}$ — the reverse correspondence

abbreviation $pG \equiv \mathfrak{p}'G$ — the resulting *Group* of type *permutation*

lemma *G-perm-comp*:

$g \in G \implies h \in G \implies G\text{-perm } g \circ G\text{-perm } h = G\text{-perm } (\text{binop } g \ h)$
<proof>

definition *the-inverse* :: $'a \Rightarrow 'a$

where *the-inverse* $g \equiv (\text{THE } h. h \in G \wedge \text{binop } g \ h = e \wedge \text{binop } h \ g = e)$

abbreviation $i \equiv \text{the-inverse}$

lemma *the-inverseD*:

assumes $g \in G$
shows $i \ g \in G \ \text{binop } g \ (i \ g) = e \ \text{binop } (i \ g) \ g = e$
<proof>

lemma *binop-G-comp-binop-iG*: $g \in G \implies x \in G \implies \text{binop } g \ (\text{binop } (i \ g) \ x) = x$
<proof>

lemma *bij-betw-binop-G*:

assumes $g \in G$
shows $\text{bij-betw } (\text{binop } g) \ G \ G$
<proof>

lemma *the-inv-into-G-binop-G*:

assumes $g \in G \ x \in G$
shows $\text{the-inv-into } G \ (\text{binop } g) \ x = \text{binop } (i \ g) \ x$
<proof>

lemma *restrict1-the-inv-into-G-binop-G*:

$g \in G \implies \text{restrict1 } (\text{the-inv-into } G \ (\text{binop } g)) \ G = G\text{-perm } (i \ g)$
<proof>

lemma *bij-G-perm*: $g \in G \implies \text{bij } (G\text{-perm } g)$

<proof>

lemma *G-perm-apply*: $g \in G \implies x \in G \implies \mathfrak{p} \ g \rightarrow x = \text{binop } g \ x$

<proof>

lemma *G-perm-apply-identity*: $g \in G \implies \mathfrak{p} \ g \rightarrow e = g$

<proof>

lemma *the-inv-G-perm*:

$g \in G \implies \text{the-inv } (G\text{-perm } g) = G\text{-perm } (i \ g)$
<proof>

lemma *Abs-G-perm-diff*:

$g \in G \implies h \in G \implies \mathfrak{p} \ g - \mathfrak{p} \ h = \mathfrak{p} \ (\text{binop } g \ (i \ h))$
<proof>

lemma *Group: Group pG*

<proof>

lemma *inj-on-p-G: inj-on p G*

<proof>

lemma *homs:*

$\bigwedge g h. g \in G \implies h \in G \implies \mathfrak{p} (\text{binop } g \ h) = \mathfrak{p} \ g + \mathfrak{p} \ h$

$\bigwedge x y. x \in pG \implies y \in pG \implies \text{binop} (\text{ip } x) (\text{ip } y) = \text{ip} (x+y)$

<proof>

lemmas *inv-correspondence-into =*

the-inv-into-into[OF inj-on-p-G, of - G, simplified]

lemma *inv-correspondence-conv-apply: $x \in pG \implies \text{ip } x = x \rightarrow e$*

<proof>

end

2.8.3 Cosets of a Group

context *Group*

begin

lemma *lcaset-refl: $a \in a +_o G$*

<proof>

lemma *lcaset-el-reduce:*

assumes $a \in G$

shows $a +_o G = G$

<proof>

lemma *lcaset-el-reduce0: $0 \in a +_o G \implies a +_o G = G$*

<proof>

lemma *lcaset-subgroup-imp-eq-reps:*

$\text{Group } H \implies w +_o H \subseteq w' +_o G \implies w' +_o G = w +_o G$

<proof>

lemma *lcaset-closed: $a \in G \implies A \subseteq G \implies a +_o A \subseteq G$*

<proof>

lemma *lcaset-rel-sym: $\text{sym} (\text{lcaset-rel } G)$*

<proof>

lemma *lcaset-rel-trans: $\text{trans} (\text{lcaset-rel } G)$*

<proof>

abbreviation $LCoset-rel :: 'g\ set \Rightarrow ('g \times 'g)\ set$
where $LCoset-rel\ H \equiv\ lcoset-rel\ H \cap (G \times G)$

lemma $refl-on-LCoset-rel: 0 \in H \Longrightarrow refl-on\ G\ (LCoset-rel\ H)$
 $\langle proof \rangle$

lemmas $subgroup-refl-on-LCoset-rel =$
 $refl-on-LCoset-rel[OF\ Group.zero-closed,\ OF\ SubgroupD1]$
lemmas $LCoset-rel-quotientI = quotientI[of - G\ LCoset-rel -]$
lemmas $LCoset-rel-quotientE = quotientE[of - G\ LCoset-rel -]$

lemma $lcoset-subgroup-rel-equiv:$
 $Subgroup\ H \Longrightarrow equiv\ G\ (LCoset-rel\ H)$
 $\langle proof \rangle$

lemma $trivial-LCoset: H \subseteq G \Longrightarrow H = LCoset-rel\ H\ \{\!\!-\}\ \{0\}$
 $\langle proof \rangle$

end

2.8.4 The Group generated by a set

inductive-set $genby :: 'a::group-add\ set \Rightarrow 'a\ set\ (\langle - \rangle)$
for $S :: 'a\ set$
where

$genby-0-closed : 0 \in \langle S \rangle$ — just in case S is empty
 $| genby-genset-closed: s \in S \Longrightarrow s \in \langle S \rangle$
 $| genby-diff-closed : w \in \langle S \rangle \Longrightarrow w' \in \langle S \rangle \Longrightarrow w - w' \in \langle S \rangle$

lemma $genby-Group: Group\ \langle S \rangle$
 $\langle proof \rangle$

lemmas $genby-uminus-closed = Group.uminus-closed\ [OF\ genby-Group]$
lemmas $genby-add-closed = Group.add-closed\ [OF\ genby-Group]$
lemmas $genby-uminus-add-closed = Group.uminus-add-closed\ [OF\ genby-Group]$
lemmas $genby-lcoset-refl = Group.lcoset-refl\ [OF\ genby-Group]$
lemmas $genby-lcoset-el-reduce = Group.lcoset-el-reduce\ [OF\ genby-Group]$
lemmas $genby-lcoset-el-reduce0 = Group.lcoset-el-reduce0\ [OF\ genby-Group]$
lemmas $genby-lcoset-closed = Group.lcoset-closed\ [OF\ genby-Group]$

lemmas $genby-lcoset-subgroup-imp-eq-reps =$
 $Group.lcoset-subgroup-imp-eq-reps[OF\ genby-Group,\ OF\ genby-Group]$

lemma $genby-genset-subset: S \subseteq \langle S \rangle$
 $\langle proof \rangle$

lemma $genby-uminus-genset-subset: uminus\ \{\!\!-\}\ S \subseteq \langle S \rangle$
 $\langle proof \rangle$

lemma *genby-in-sum-list-lists*:

fixes S

defines $S\text{-sum-lists}$: $S\text{-sum-lists} \equiv (\bigcup ss \in \text{lists } (S \cup \text{uminus } 'S). \{sum\text{-list } ss\})$

shows $w \in \langle S \rangle \implies w \in S\text{-sum-lists}$

<proof>

lemma *sum-list-lists-in-genby*: $ss \in \text{lists } (S \cup \text{uminus } 'S) \implies sum\text{-list } ss \in \langle S \rangle$

<proof>

lemma *sum-list-lists-in-genby-sym*:

$\text{uminus } 'S \subseteq S \implies ss \in \text{lists } S \implies sum\text{-list } ss \in \langle S \rangle$

<proof>

lemma *genby-eq-sum-lists*: $\langle S \rangle = (\bigcup ss \in \text{lists } (S \cup \text{uminus } 'S). \{sum\text{-list } ss\})$

<proof>

lemma *genby-mono*: $T \subseteq S \implies \langle T \rangle \subseteq \langle S \rangle$

<proof>

lemma (**in** *Group*) *genby-closed*:

assumes $S \subseteq G$

shows $\langle S \rangle \subseteq G$

<proof>

lemma (**in** *Group*) *genby-subgroup*: $S \subseteq G \implies \text{Subgroup } \langle S \rangle$

<proof>

lemma *genby-sym-eq-sum-lists*:

$\text{uminus } 'S \subseteq S \implies \langle S \rangle = (\bigcup ss \in \text{lists } S. \{sum\text{-list } ss\})$

<proof>

lemma *genby-empty'*: $w \in \langle \{\} \rangle \implies w = 0$

<proof>

lemma *genby-order2'*:

assumes $s+s=0$

shows $w \in \langle \{s\} \rangle \implies w = 0 \vee w = s$

<proof>

lemma *genby-order2*: $s+s=0 \implies \langle \{s\} \rangle = \{0, s\}$

<proof>

lemma *genby-empty*: $\langle \{\} \rangle = 0$

<proof>

lemma *genby-lcoset-order2*: $s+s=0 \implies w + o \langle \{s\} \rangle = \{w, w+s\}$

<proof>

lemma *genby-lcoset-empty*: $(w::'a::\text{group-add}) + o \langle \{\} \rangle = \{w\}$

<proof>

lemma (in *Group*) *genby-set-lconjby-set-lconjby-closed*:

fixes $A :: 'g \text{ set}$

defines $S \equiv (\bigcup_{g \in G}. \text{lconjby } g \text{ ` } A)$

assumes $g \in G$

shows $x \in \langle S \rangle \implies \text{lconjby } g \ x \in \langle S \rangle$

<proof>

lemma (in *Group*) *genby-set-lconjby-set-rconjby-closed*:

fixes $A :: 'g \text{ set}$

defines $S \equiv (\bigcup_{g \in G}. \text{lconjby } g \text{ ` } A)$

assumes $g \in G \ x \in \langle S \rangle$

shows $\text{rconjby } g \ x \in \langle S \rangle$

<proof>

2.8.5 Homomorphisms and isomorphisms

locale *GroupHom = Group* G

for $G :: 'g::\text{group-add set}$

+ **fixes** $T :: 'g \Rightarrow 'h::\text{group-add}$

assumes $\text{hom} : g \in G \implies g' \in G \implies T (g + g') = T g + T g'$

and $\text{supp} : \text{supp } T \subseteq G$

begin

lemma *im-zero*: $T \ 0 = 0$

<proof>

lemma *im-uminus*: $T (-g) = - T g$

<proof>

lemma *im-uminus-add*: $g \in G \implies g' \in G \implies T (-g + g') = - T g + T g'$

<proof>

lemma *im-diff*: $g \in G \implies g' \in G \implies T (g - g') = T g - T g'$

<proof>

lemma *im-lconjby*: $x \in G \implies g \in G \implies T (\text{lconjby } x \ g) = \text{lconjby } (T \ x) \ (T \ g)$

<proof>

lemma *im-sum-list-map*:

$\text{set } (\text{map } f \ as) \subseteq G \implies T (\sum a \leftarrow as. f \ a) = (\sum a \leftarrow as. T (f \ a))$

<proof>

lemma *comp*:

assumes $\text{GroupHom } H \ S \ T \ G \subseteq H$

shows $\text{GroupHom } G \ (S \circ T)$

<proof>

end

definition $ker :: ('a \Rightarrow 'b :: zero) \Rightarrow 'a \text{ set}$
where $ker\ f = \{a. f\ a = 0\}$

lemma $ker\text{-subset-ker-restrict0}$: $ker\ f \subseteq ker\ (restrict0\ f\ A)$
 $\langle proof \rangle$

context $GroupHom$
begin

abbreviation $Ker \equiv ker\ T \cap G$

lemma $uminus\text{-add-in-Ker-eq-eq-im}$:
 $g \in G \Rightarrow h \in G \Rightarrow (-g + h \in Ker) = (T\ g = T\ h)$
 $\langle proof \rangle$

end

locale $UGroupHom = GroupHom\ UNIV\ T$
for $T :: 'g :: group\text{-add} \Rightarrow 'h :: group\text{-add}$
begin

lemmas $im\text{-zero} = im\text{-zero}$
lemmas $im\text{-uminus} = im\text{-uminus}$

lemma hom : $T\ (g + g') = T\ g + T\ g'$
 $\langle proof \rangle$

lemma $im\text{-diff}$: $T\ (g - g') = T\ g - T\ g'$
 $\langle proof \rangle$

lemma $im\text{-lconjby}$: $T\ (lconjby\ x\ g) = lconjby\ (T\ x)\ (T\ g)$
 $\langle proof \rangle$

lemma $restrict0$:
assumes $Group\ G$
shows $GroupHom\ G\ (restrict0\ T\ G)$
 $\langle proof \rangle$

end

lemma $UGroupHomI$:
assumes $\bigwedge g\ g'. T\ (g + g') = T\ g + T\ g'$
shows $UGroupHom\ T$
 $\langle proof \rangle$

locale $GroupIso = GroupHom\ G\ T$

for $G :: 'g::group-add\ set$
and $T :: 'g \Rightarrow 'h::group-add$
+ **assumes** $inj-on: inj-on\ T\ G$

lemma (in $GroupHom$) $isoI$:
assumes $\bigwedge k. k \in G \Longrightarrow T\ k = 0 \Longrightarrow k=0$
shows $GroupIso\ G\ T$
 $\langle proof \rangle$

In a $BinOpSetGroup$, any map from the set into a type of class $group-add$ that respects the binary operation induces a $GroupHom$.

abbreviation (in $BinOpSetGroup$) $lift-hom\ T \equiv restrict0\ (T \circ ip)\ pG$

lemma (in $BinOpSetGroup$) $lift-hom$:
fixes $T :: 'a \Rightarrow 'b::group-add$
assumes $\forall g \in G. \forall h \in G. T\ (binop\ g\ h) = T\ g + T\ h$
shows $GroupHom\ pG\ (lift-hom\ T)$
 $\langle proof \rangle$

2.8.6 Normal subgroups

definition $rcoset-rel :: 'a::\{minus,plus\}\ set \Rightarrow ('a \times 'a)\ set$
where $rcoset-rel\ A \equiv \{(x,y). x-y \in A\}$

context $Group$
begin

lemma $rcoset-rel-conv-lcoset-rel$:
 $rcoset-rel\ G = map-prod\ uminus\ uminus\ `(\lcoset-rel\ G)$
 $\langle proof \rangle$

lemma $rcoset-rel-sym: sym\ (rcoset-rel\ G)$
 $\langle proof \rangle$

abbreviation $RCoset-rel :: 'g\ set \Rightarrow ('g \times 'g)\ set$
where $RCoset-rel\ H \equiv rcoset-rel\ H \cap (G \times G)$

definition $normal :: 'g\ set \Rightarrow bool$
where $normal\ H \equiv (\forall g \in G. LCoset-rel\ H\ \{\!-\!\} g = RCoset-rel\ H\ \{\!-\!\} g)$

lemma $normalI$:
assumes $Group\ H\ \forall g \in G. \forall h \in H. \exists h' \in H. g+h = h'+g$
 $\forall g \in G. \forall h \in H. \exists h' \in H. h+g = g+h'$
shows $normal\ H$
 $\langle proof \rangle$

lemma $normal-lconjby-closed$:
 $\llbracket Subgroup\ H; normal\ H; g \in G; h \in H \rrbracket \Longrightarrow lconjby\ g\ h \in H$
 $\langle proof \rangle$

lemma *normal-rconjby-closed*:

$\llbracket \text{Subgroup } H; \text{ normal } H; g \in G; h \in H \rrbracket \implies \text{rconjby } g \ h \in H$
<proof>

abbreviation *normal-closure* $A \equiv \langle \bigcup_{g \in G} \text{lconjby } g \ ' A \rangle$

lemma (in *Group*) *normal-closure*:

assumes $A \subseteq G$

shows *normal* (*normal-closure* A)

<proof>

end

2.8.7 Quotient groups

Here we use the bridge built by *BinOpSetGroup* to make the quotient of a *Group* by a normal subgroup into a *Group* itself.

context *Group*

begin

lemma *normal-quotient-add-well-defined*:

assumes *Subgroup* H *normal* H $g \in G$ $g' \in G$

shows $\text{LCoset-rel } H \ \{g\} + \text{LCoset-rel } H \ \{g'\} = \text{LCoset-rel } H \ \{g+g'\}$

<proof>

abbreviation *quotient-set* $H \equiv G // \text{LCoset-rel } H$

lemma *BinOpSetGroup-normal-quotient*:

assumes *Subgroup* H *normal* H

shows *BinOpSetGroup* (*quotient-set* H) (+) H

<proof>

abbreviation *abs-lcoset-perm* $H \equiv$

$\text{BinOpSetGroup.Abs-G-perm } (\text{quotient-set } H) (+)$

abbreviation *abs-lcoset-perm-lift* H $g \equiv \text{abs-lcoset-perm } H \ (\text{LCoset-rel } H \ \{g\})$

abbreviation *abs-lcoset-perm-lift-arg-permutation* g $H \equiv \text{abs-lcoset-perm-lift } H$ g

notation *abs-lcoset-perm-lift-arg-permutation* ($[-|-]$ [51,51] 50)

end

abbreviation *Group-abs-lcoset-perm-lift-arg-permutation* G' g $H \equiv$

$\text{Group.abs-lcoset-perm-lift-arg-permutation } G' g H$

notation *Group-abs-lcoset-perm-lift-arg-permutation* ($[-|-]$ [51,51,51] 50)

context *Group*

begin

lemmas *lcoset-perm-def* =
BinOpSetGroup.Abs-G-perm-def[*OF BinOpSetGroup-normal-quotient*]
lemmas *lcoset-perm-comp* =
BinOpSetGroup.G-perm-comp[*OF BinOpSetGroup-normal-quotient*]
lemmas *bij-lcoset-perm* =
BinOpSetGroup.bij-G-perm[*OF BinOpSetGroup-normal-quotient*]

lemma *trivial-lcoset-perm*:
assumes *Subgroup H normal H h∈H*
shows *restrict1 ((+) (LCoset-rel H “ {h})) (quotient-set H) = id*
<proof>

definition *quotient-group* :: *'g set ⇒ 'g set permutation set* **where**
quotient-group H ≡ BinOpSetGroup.pG (quotient-set H) (+)

abbreviation *natural-quotient-hom H ≡ restrict0 (λg. [g|H]) G*

theorem *quotient-group*:
Subgroup H ⇒ normal H ⇒ Group (quotient-group H)
<proof>

lemma *natural-quotient-hom*:
Subgroup H ⇒ normal H ⇒ GroupHom G (natural-quotient-hom H)
<proof>

lemma *natural-quotient-hom-image*:
natural-quotient-hom H ‘ G = quotient-group H
<proof>

lemma *quotient-group-UN*: *quotient-group H = (λg. [g|H]) ‘ G*
<proof>

lemma *quotient-identity-rule*: *[[Subgroup H; normal H; h∈H]] ⇒ [h|H] = 0*
<proof>

lemma *quotient-group-lift-to-quotient-set*:
[[Subgroup H; normal H; g∈G]] ⇒ ([g|H]) → H = LCoset-rel H “ {g}
<proof>

end

2.8.8 The induced homomorphism on a quotient group

A normal subgroup contained in the kernel of a homomorphism gives rise to a homomorphism on the quotient group by that subgroup. When the subgroup is the kernel itself (which is always normal), we obtain an isomorphism on the quotient.

context *GroupHom*

begin

lemma *respects-Ker-lcosets*: $H \subseteq \text{Ker} \implies T \text{ respects } (\text{LCoset-rel } H)$
<proof>

abbreviation *quotient-hom* $H \equiv$
 $\text{BinOpSetGroup.lift-hom } (\text{quotient-set } H) (+) (\text{quotientfun } T)$

lemmas *normal-subgroup-quotientfun-classrep-equality* =
quotientfun-classrep-equality [
 OF subgroup-refl-on-LCoset-rel, OF - respects-Ker-lcosets
]

lemma *quotient-hom-im*:
[[*Subgroup* H ; *normal* H ; $H \subseteq \text{Ker}$; $g \in G$]] $\implies \text{quotient-hom } H (\lceil g|H \rceil) = T g$
<proof>

lemma *quotient-hom*:
assumes *Subgroup* H *normal* H $H \subseteq \text{Ker}$
shows $\text{GroupHom } (\text{quotient-group } H) (\text{quotient-hom } H)$
<proof>

end

2.9 Free groups

2.9.1 Words in letters of *signed* type

Definitions and basic fact We pair elements of some type with type *bool*, where the *bool* part of the pair indicates inversion.

abbreviation *pairtrue* $\equiv \lambda s. (s, \text{True})$

abbreviation *pairfalse* $\equiv \lambda s. (s, \text{False})$

abbreviation *flip-signed* :: $'a \text{ signed} \Rightarrow 'a \text{ signed}$
where *flip-signed* $\equiv \text{apsnd } (\lambda b. \neg b)$

abbreviation *nflipped-signed* :: $'a \text{ signed} \Rightarrow 'a \text{ signed} \Rightarrow \text{bool}$
where *nflipped-signed* $x y \equiv y \neq \text{flip-signed } x$

lemma *flip-signed-order2*: $\text{flip-signed } (\text{flip-signed } x) = x$
<proof>

abbreviation *charpair* :: $'a::\text{uminus set} \Rightarrow 'a \Rightarrow 'a \text{ signed}$
where *charpair* $S s \equiv \text{if } s \in S \text{ then } (s, \text{True}) \text{ else } (-s, \text{False})$

lemma *map-charpair-uniform*:
 $ss \in \text{lists } S \implies \text{map } (\text{charpair } S) ss = \text{map } \text{pairtrue } ss$
<proof>

lemma *fst-set-map-charpair-un-uminus*:

fixes *ss* :: 'a::group-add list

shows $ss \in \text{lists } (S \cup \text{uminus } 'S) \implies \text{fst } ' \text{ set } (\text{map } (\text{charpair } S) \text{ ss}) \subseteq S$

<proof>

abbreviation *apply-sign* :: ('a ⇒ 'b::uminus) ⇒ 'a signed ⇒ 'b

where *apply-sign* f $x \equiv (\text{if } \text{snd } x \text{ then } f (\text{fst } x) \text{ else } - f (\text{fst } x))$

A word in such pairs will be considered proper if it does not contain consecutive letters that have opposite signs (and so are considered inverse), since such consecutive letters would be cancelled in a group.

abbreviation *proper-signed-list* :: 'a signed list ⇒ bool

where *proper-signed-list* $\equiv \text{binrelchain } \text{nflipped-signed}$

lemma *proper-map-flip-signed*:

proper-signed-list $xs \implies \text{proper-signed-list } (\text{map } \text{flip-signed } xs)$

<proof>

lemma *proper-rev-map-flip-signed*:

proper-signed-list $xs \implies \text{proper-signed-list } (\text{rev } (\text{map } \text{flip-signed } xs))$

<proof>

lemma *uniform-snd-imp-proper-signed-list*:

snd ' set $xs \subseteq \{b\} \implies \text{proper-signed-list } xs$

<proof>

lemma *proper-signed-list-map-uniform-snd*:

proper-signed-list $(\text{map } (\lambda s. (s,b)) \text{ as})$

<proof>

Algebra Addition is performed by appending words and recursively removing any newly created adjacent pairs of inverse letters. Since we will only ever be adding proper words, we only need to care about newly created adjacent inverse pairs in the middle.

function *prappend-signed-list* :: 'a signed list ⇒ 'a signed list ⇒ 'a signed list

where *prappend-signed-list* $xs [] = xs$

| *prappend-signed-list* $[] \text{ ys} = \text{ys}$

| *prappend-signed-list* $(xs@[x]) (y\#ys) = ($

if $y = \text{flip-signed } x$ then *prappend-signed-list* $xs \text{ ys}$ else $xs @ x \# y \# ys$

)

<proof>

termination *<proof>*

lemma *proper-prappend-signed-list*:

proper-signed-list $xs \implies \text{proper-signed-list } \text{ys}$

$\implies \text{proper-signed-list } (\text{prappend-signed-list } xs \text{ ys})$

<proof>

lemma *fully-prappend-signed-list*:

prappend-signed-list (rev (map *flip-signed* xs)) xs = []
<proof>

lemma *prappend-signed-list-single-Cons*:

prappend-signed-list [x] (y#ys) = (if y = *flip-signed* x then ys else x#y#ys)
<proof>

lemma *prappend-signed-list-map-uniform-snd*:

prappend-signed-list (map ($\lambda s. (s,b)$) xs) (map ($\lambda s. (s,b)$) ys) =
map ($\lambda s. (s,b)$) xs @ map ($\lambda s. (s,b)$) ys
<proof>

lemma *prappend-signed-list-assoc-conv-snoc2Cons*:

assumes *proper-signed-list* (xs@[y]) *proper-signed-list* (y#ys)
shows *prappend-signed-list* (xs@[y]) ys = *prappend-signed-list* xs (y#ys)
<proof>

lemma *prappend-signed-list-assoc*:

[*proper-signed-list* xs; *proper-signed-list* ys; *proper-signed-list* zs] \implies
prappend-signed-list (prappend-signed-list xs ys) zs =
prappend-signed-list xs (prappend-signed-list ys zs)
<proof>

lemma *fst-set-prappend-signed-list*:

fst ' set (prappend-signed-list xs ys) \subseteq *fst* ' (set xs \cup set ys)
<proof>

lemma *collapse-flipped-signed*:

prappend-signed-list [(s,b)] [(s,-b)] = []
<proof>

2.9.2 The collection of proper signed lists as a type

Here we create a type out of the collection of proper signed lists. This type will be of class *group-add*, with the empty list as zero, the modified append operation *prappend-signed-list* as addition, and inversion performed by flipping the signs of the elements in the list and then reversing the order.

Type definition, instantiations, and instances Here we define the type and instantiate it with respect to various type classes.

typedef 'a *freeword* = {as::'a *signed list. proper-signed-list* as}
morphisms *freeword* *Abs-freeword*
<proof>

These two functions act as the natural injections of letters and words in the letter type into the *freeword* type.

```

abbreviation Abs-freeletter :: 'a ⇒ 'a freeword
  where Abs-freeletter s ≡ Abs-freeword [pairtrue s]

abbreviation Abs-freelist :: 'a list ⇒ 'a freeword
  where Abs-freelist as ≡ Abs-freeword (map pairtrue as)

abbreviation Abs-freelistfst :: 'a signed list ⇒ 'a freeword
  where Abs-freelistfst xs ≡ Abs-freelist (map fst xs)

setup-lifting type-definition-freeword

instantiation freeword :: (type) zero
begin
lift-definition zero-freeword :: 'a freeword is []::'a signed list ⟨proof⟩
instance ⟨proof⟩
end

instantiation freeword :: (type) plus
begin
lift-definition plus-freeword :: 'a freeword ⇒ 'a freeword ⇒ 'a freeword
  is prappend-signed-list
  ⟨proof⟩
instance ⟨proof⟩
end

instantiation freeword :: (type) uminus
begin
lift-definition uminus-freeword :: 'a freeword ⇒ 'a freeword
  is λxs. rev (map flip-signed xs)
  ⟨proof⟩
instance ⟨proof⟩
end

instantiation freeword :: (type) minus
begin
lift-definition minus-freeword :: 'a freeword ⇒ 'a freeword ⇒ 'a freeword
  is λxs ys. prappend-signed-list xs (rev (map flip-signed ys))
  ⟨proof⟩
instance ⟨proof⟩
end

instance freeword :: (type) semigroup-add
  ⟨proof⟩

instance freeword :: (type) monoid-add
  ⟨proof⟩

instance freeword :: (type) group-add
  ⟨proof⟩

```

Basic algebra and transfer facts in the *freeword* type Here we record basic algebraic manipulations for the *freeword* type as well as various transfer facts for dealing with representations of elements of *freeword* type as lists of signed letters.

abbreviation *Abs-freeletter-add* :: 'a ⇒ 'a ⇒ 'a *freeword* (**infixl** [+] 65)
where $s \text{ [+} \text{] } t \equiv \text{Abs-freeletter } s + \text{Abs-freeletter } t$

lemma *Abs-freeword-Cons*:

assumes *proper-signed-list* (x#xs)

shows $\text{Abs-freeword } (x\#xs) = \text{Abs-freeword } [x] + \text{Abs-freeword } xs$

⟨*proof*⟩

lemma *Abs-freelist-Cons*: $\text{Abs-freelist } (x\#xs) = \text{Abs-freeletter } x + \text{Abs-freelist } xs$

⟨*proof*⟩

lemma *plus-freeword-abs-eq*:

proper-signed-list xs ⇒ *proper-signed-list* ys ⇒

$\text{Abs-freeword } xs + \text{Abs-freeword } ys = \text{Abs-freeword } (\text{prappend-signed-list } xs \text{ } ys)$

⟨*proof*⟩

lemma *Abs-freeletter-add*: $s \text{ [+} \text{] } t = \text{Abs-freelist } [s,t]$

⟨*proof*⟩

lemma *uminus-freeword-Abs-eq*:

proper-signed-list xs ⇒

$-\text{Abs-freeword } xs = \text{Abs-freeword } (\text{rev } (\text{map } \text{flip-signed } xs))$

⟨*proof*⟩

lemma *uminus-Abs-freeword-singleton*:

$-\text{Abs-freeword } [(s,b)] = \text{Abs-freeword } [(s,\neg b)]$

⟨*proof*⟩

lemma *Abs-freeword-append-uniform-snd*:

$\text{Abs-freeword } (\text{map } (\lambda s. (s,b)) (xs@ys)) =$

$\text{Abs-freeword } (\text{map } (\lambda s. (s,b)) xs) + \text{Abs-freeword } (\text{map } (\lambda s. (s,b)) ys)$

⟨*proof*⟩

lemmas *Abs-freelist-append* = *Abs-freeword-append-uniform-snd*[of True]

lemma *Abs-freelist-append-append*:

$\text{Abs-freelist } (xs@ys@zs) = \text{Abs-freelist } xs + \text{Abs-freelist } ys + \text{Abs-freelist } zs$

⟨*proof*⟩

lemma *Abs-freelist-inverse*: $\text{freeword } (\text{Abs-freelist } as) = \text{map } \text{pairtrue } as$

⟨*proof*⟩

lemma *Abs-freeword-singleton-conv-apply-sign-freeletter*:

$\text{Abs-freeword } [x] = \text{apply-sign } \text{Abs-freeletter } x$

⟨*proof*⟩

lemma *Abs-freeword-conv-freeletter-sum-list*:
proper-signed-list xs \implies
Abs-freeword xs = $(\sum x \leftarrow xs. \text{apply-sign } \text{Abs-freeletter } x)$
 <proof>

lemma *freeword-conv-freeletter-sum-list*:
 $x = (\sum s \leftarrow \text{freeword } x. \text{apply-sign } \text{Abs-freeletter } s)$
 <proof>

lemma *Abs-freeletter-prod-conv-Abs-freeword*:
 $\text{snd } x \implies \text{Abs-freeletter } (\text{fst } x) = \text{Abs-freeword } [x]$
 <proof>

2.9.3 Lifts of functions on the letter type

Here we lift functions on the letter type to type *freeword*. In particular, we are interested in the case where the function being lifted has codomain of class *group-add*.

The universal property The universal property for free groups says that every function from the letter type to some *group-add* type gives rise to a unique homomorphism.

lemma *extend-map-to-freeword-hom'*:
fixes $f :: 'a \Rightarrow 'b::\text{group-add}$
defines $h :: 'a \text{ signed} \Rightarrow 'b \equiv \lambda(s,b). \text{if } b \text{ then } f s \text{ else } - (f s)$
defines $g :: 'a \text{ signed list} \Rightarrow 'b \equiv \lambda xs. \text{sum-list } (\text{map } h xs)$
shows $g (\text{prappend-signed-list } xs ys) = g xs + g ys$
 <proof>

lemma *extend-map-to-freeword-hom1*:
fixes $f :: 'a \Rightarrow 'b::\text{group-add}$
defines $h :: 'a \text{ signed} \Rightarrow 'b \equiv \lambda(s,b). \text{if } b \text{ then } f s \text{ else } - (f s)$
defines $g :: 'a \text{ freeword} \Rightarrow 'b \equiv \lambda x. \text{sum-list } (\text{map } h (\text{freeword } x))$
shows $g (\text{Abs-freeletter } s) = f s$
 <proof>

lemma *extend-map-to-freeword-hom2*:
fixes $f :: 'a \Rightarrow 'b::\text{group-add}$
defines $h :: 'a \text{ signed} \Rightarrow 'b \equiv \lambda(s,b). \text{if } b \text{ then } f s \text{ else } - (f s)$
defines $g :: 'a \text{ freeword} \Rightarrow 'b \equiv \lambda x. \text{sum-list } (\text{map } h (\text{freeword } x))$
shows $U\text{GroupHom } g$
 <proof>

lemma *uniqueness-of-extended-map-to-freeword-hom'*:
fixes $f :: 'a \Rightarrow 'b::\text{group-add}$
defines $h :: 'a \text{ signed} \Rightarrow 'b \equiv \lambda(s,b). \text{if } b \text{ then } f s \text{ else } - (f s)$

defines $g: g::'a \text{ signed list} \Rightarrow 'b \equiv \lambda xs. \text{sum-list } (\text{map } h \text{ } xs)$
assumes $\text{singles}: \bigwedge s. k [(s, \text{True})] = f \text{ } s$
and $\text{adds} : \bigwedge xs \text{ } ys. \text{proper-signed-list } xs \Longrightarrow \text{proper-signed-list } ys$
 $\Longrightarrow k (\text{prappend-signed-list } xs \text{ } ys) = k \text{ } xs + k \text{ } ys$
shows $\text{proper-signed-list } xs \Longrightarrow k \text{ } xs = g \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *uniqueness-of-extended-map-to-freeword-hom*:
fixes $f :: 'a \Rightarrow 'b::\text{group-add}$
defines $h::'a \text{ signed} \Rightarrow 'b \equiv \lambda(s,b). \text{if } b \text{ then } f \text{ } s \text{ else } - (f \text{ } s)$
defines $g::'a \text{ freeword} \Rightarrow 'b \equiv \lambda x. \text{sum-list } (\text{map } h \text{ } (\text{freeword } x))$
assumes $k: k \circ \text{Abs-freeletter} = f \text{ } UGroupHom \text{ } k$
shows $k = g$
 $\langle \text{proof} \rangle$

theorem *universal-property*:
fixes $f :: 'a \Rightarrow 'b::\text{group-add}$
shows $\exists! g::'a \text{ freeword} \Rightarrow 'b. g \circ \text{Abs-freeletter} = f \wedge UGroupHom \text{ } g$
 $\langle \text{proof} \rangle$

Properties of homomorphisms afforded by the universal property

The lift of a function on the letter set is the unique additive function on *freeword* that agrees with the original function on letters.

definition *freeword-funlift* $:: ('a \Rightarrow 'b::\text{group-add}) \Rightarrow ('a \text{ freeword} \Rightarrow 'b::\text{group-add})$
where *freeword-funlift* $f \equiv (\text{THE } g. g \circ \text{Abs-freeletter} = f \wedge UGroupHom \text{ } g)$

lemma *additive-freeword-funlift*: $UGroupHom \text{ } (\text{freeword-funlift } f)$
 $\langle \text{proof} \rangle$

lemma *freeword-funlift-Abs-freeletter*: $\text{freeword-funlift } f \text{ } (\text{Abs-freeletter } s) = f \text{ } s$
 $\langle \text{proof} \rangle$

lemmas *freeword-funlift-add* $= UGroupHom.\text{hom} \quad [\text{OF additive-freeword-funlift}]$
lemmas *freeword-funlift-0* $= UGroupHom.\text{im-zero} \quad [\text{OF additive-freeword-funlift}]$
lemmas *freeword-funlift-uminus* $= UGroupHom.\text{im-uminus} \quad [\text{OF additive-freeword-funlift}]$
lemmas *freeword-funlift-diff* $= UGroupHom.\text{im-diff} \quad [\text{OF additive-freeword-funlift}]$
lemmas *freeword-funlift-lconjby* $= UGroupHom.\text{im-lconjby} \quad [\text{OF additive-freeword-funlift}]$

lemma *freeword-funlift-uminus-Abs-freeletter*:
 $\text{freeword-funlift } f \text{ } (\text{Abs-freeletter } [(s, \text{False})]) = - f \text{ } s$
 $\langle \text{proof} \rangle$

lemma *freeword-funlift-Abs-freeletter-singleton*:
 $\text{freeword-funlift } f \text{ } (\text{Abs-freeletter } [x]) = \text{apply-sign } f \text{ } x$
 $\langle \text{proof} \rangle$

lemma *freeword-funlift-Abs-freeletter-Cons*:
assumes $\text{proper-signed-list } (x \# xs)$

shows $\text{freeword-funlift } f \text{ (Abs-freeword } (x\#xs)) =$
 $\text{apply-sign } f \text{ } x + \text{freeword-funlift } f \text{ (Abs-freeword } xs)$
 ⟨proof⟩

lemma *freeword-funlift-Abs-freeword*:
 $\text{proper-signed-list } xs \implies \text{freeword-funlift } f \text{ (Abs-freeword } xs) =$
 $(\sum x \leftarrow xs. \text{apply-sign } f \text{ } x)$
 ⟨proof⟩

lemma *freeword-funlift-Abs-freelist*:
 $\text{freeword-funlift } f \text{ (Abs-freelist } xs) = (\sum x \leftarrow xs. f \text{ } x)$
 ⟨proof⟩

lemma *freeword-funlift-im'*:
 $\text{proper-signed-list } xs \implies \text{fst ' set } xs \subseteq S \implies$
 $\text{freeword-funlift } f \text{ (Abs-freeword } xs) \in \langle f'S \rangle$
 ⟨proof⟩

2.9.4 Free groups on a set

We now take the free group on a set to be the set in the *freeword* type with letters restricted to the given set.

Definition and basic facts Here we define the set of elements of the free group over a set of letters, and record basic facts about that set.

definition *FreeGroup* :: 'a set \Rightarrow 'a freeword set
where $\text{FreeGroup } S \equiv \{x. \text{fst ' set } (\text{freeword } x) \subseteq S\}$

lemma *FreeGroupI-transfer*:
 $\text{proper-signed-list } xs \implies \text{fst ' set } xs \subseteq S \implies \text{Abs-freeword } xs \in \text{FreeGroup } S$
 ⟨proof⟩

lemma *FreeGroupD*: $x \in \text{FreeGroup } S \implies \text{fst ' set } (\text{freeword } x) \subseteq S$
 ⟨proof⟩

lemma *FreeGroupD-transfer*:
 $\text{proper-signed-list } xs \implies \text{Abs-freeword } xs \in \text{FreeGroup } S \implies \text{fst ' set } xs \subseteq S$
 ⟨proof⟩

lemma *FreeGroupD-transfer'*:
 $\text{Abs-freelist } xs \in \text{FreeGroup } S \implies xs \in \text{lists } S$
 ⟨proof⟩

lemma *FreeGroup-0-closed*: $0 \in \text{FreeGroup } S$
 ⟨proof⟩

lemma *FreeGroup-diff-closed*:
assumes $x \in \text{FreeGroup } S \ y \in \text{FreeGroup } S$

shows $x-y \in \text{FreeGroup } S$
 ⟨proof⟩

lemma *FreeGroup-Group: Group (FreeGroup S)*
 ⟨proof⟩

lemmas *FreeGroup-add-closed = Group.add-closed [OF FreeGroup-Group]*
lemmas *FreeGroup-uminus-closed = Group.uminus-closed [OF FreeGroup-Group]*

lemmas *FreeGroup-genby-set-lconjby-set-rconjby-closed =*
Group.genby-set-lconjby-set-rconjby-closed[OF FreeGroup-Group]

lemma *Abs-freelist-in-FreeGroup: ss ∈ lists S ⇒ Abs-freelist ss ∈ FreeGroup S*
 ⟨proof⟩

lemma *Abs-freeletter-in-FreeGroup-iff: (Abs-freeletter s ∈ FreeGroup S) = (s ∈ S)*
 ⟨proof⟩

Lifts of functions from the letter set to some type of class *group-add*

We again obtain a universal property for functions from the (restricted) letter set to some type of class *group-add*.

abbreviation *res-freeword-funlift f S ≡*
restrict0 (freeword-funlift f) (FreeGroup S)

lemma *freeword-funlift-im: x ∈ FreeGroup S ⇒ freeword-funlift f x ∈ ⟨f ' S⟩*
 ⟨proof⟩

lemma *freeword-funlift-surj':*
ys ∈ lists (f'S ∪ uminus'f'S) ⇒ sum-list ys ∈ freeword-funlift f ' FreeGroup S
 ⟨proof⟩

lemma *freeword-funlift-surj:*
fixes $f :: 'a \Rightarrow 'b::\text{group-add}$
shows $\text{freeword-funlift } f \text{ ' FreeGroup } S = \langle f \text{ ' } S \rangle$
 ⟨proof⟩

lemma *hom-restrict0-freeword-funlift:*
GroupHom (FreeGroup S) (res-freeword-funlift f S)
 ⟨proof⟩

lemma *uniqueness-of-restricted-lift:*
assumes $\text{GroupHom (FreeGroup } S) T \forall s \in S. T (\text{Abs-freeletter } s) = f s$
shows $T = \text{res-freeword-funlift } f S$
 ⟨proof⟩

theorem *FreeGroup-universal-property:*
fixes $f :: 'a \Rightarrow 'b::\text{group-add}$
shows $\exists! T :: 'a \text{ freeword} \Rightarrow 'b. (\forall s \in S. T (\text{Abs-freeletter } s) = f s) \wedge$

GroupHom (FreeGroup S) T
 ⟨proof⟩

2.9.5 Group presentations

We now define a group presentation to be the quotient of a free group by the subgroup generated by all conjugates of a set of relators. We are most concerned with lifting functions on the letter set to the free group and with the associated induced homomorphisms on the quotient.

A first group presentation locale and basic facts Here we define a locale that provides a way to construct a group by providing sets of generators and relator words.

locale *GroupByPresentation* =
fixes *S* :: 'a set — the set of generators
and *P* :: 'a signed list set — the set of relator words
assumes *P-S*: $ps \in P \implies \text{fst } ' \text{ set } ps \subseteq S$
and *proper-P*: $ps \in P \implies \text{proper-signed-list } ps$
begin

abbreviation *P'* \equiv *Abs-freeword ' P* — the set of relators
abbreviation *Q* \equiv *Group.normal-closure (FreeGroup S) P'*
 — the normal subgroup generated by relators inside the free group
abbreviation *G* \equiv *Group.quotient-group (FreeGroup S) Q*

lemmas *G-UN* = *Group.quotient-group-UN[OF FreeGroup-Group, of S Q]*

lemma *P'-FreeS*: $P' \subseteq \text{FreeGroup } S$
 ⟨proof⟩

lemma *relators*: $P' \subseteq Q$
 ⟨proof⟩

lemmas *lconjby-P'-FreeS* =
Group.set-lconjby-subset-closed[
OF FreeGroup-Group - P'-FreeS, OF basic-monos(1)
]

lemmas *Q-FreeS* =
Group.genby-closed[OF FreeGroup-Group lconjby-P'-FreeS]

lemmas *Q-subgroup-FreeS* =
Group.genby-subgroup[OF FreeGroup-Group lconjby-P'-FreeS]

lemmas *normal-Q* = *Group.normal-closure[OF FreeGroup-Group, OF P'-FreeS]*

lemmas *natural-hom* =

```

Group.natural-quotient-hom[
  OF FreeGroup-Group Q-subgroup-FreeS normal-Q
]

```

```

lemmas natural-hom-image =
  Group.natural-quotient-hom-image[OF FreeGroup-Group, of S Q]

```

end

Functions on the quotient induced from lifted functions A function on the generator set into a type of class *group-add* lifts to a unique homomorphism on the free group. If this lift is trivial on relators, then it factors to a homomorphism of the group described by the generators and relators.

```

locale GroupByPresentationInducedFun = GroupByPresentation S P
  for S :: 'a set
  and P :: 'a signed list set — the set of relator words
+ fixes f :: 'a ⇒ 'b::group-add
  assumes lift-f-trivial-P:
    ps∈P ⇒ freeword-funlift f (Abs-freeword ps) = 0
begin

```

```

abbreviation lift-f ≡ freeword-funlift f

```

```

definition induced-hom :: 'a freeword set permutation ⇒ 'b
  where induced-hom ≡ GroupHom.quotient-hom (FreeGroup S)
    (restrict0 lift-f (FreeGroup S)) Q
  — the restrict0 operation is really only necessary to make GroupByPresentationInducedFun.induced-hom a GroupHom

```

```

abbreviation F ≡ induced-hom

```

```

lemma lift-f-trivial-P': p∈P' ⇒ lift-f p = 0
  ⟨proof⟩

```

```

lemma lift-f-trivial-lconjby-P': p∈P' ⇒ lift-f (lconjby w p) = 0
  ⟨proof⟩

```

```

lemma lift-f-trivial-Q: q∈Q ⇒ lift-f q = 0
  ⟨proof⟩

```

```

lemma lift-f-ker-Q: Q ⊆ ker lift-f
  ⟨proof⟩

```

```

lemma lift-f-Ker-Q: Q ⊆ GroupHom.Ker (FreeGroup S) lift-f
  ⟨proof⟩

```

```

lemma restrict0-lift-f-Ker-Q:
  Q ⊆ GroupHom.Ker (FreeGroup S) (restrict0 lift-f (FreeGroup S))

```

<proof>

lemma *induced-hom-equality:*

$w \in \text{FreeGroup } S \implies F (\lceil \text{FreeGroup } S | w | Q \rceil) = \text{lift-}f \ w$

— algebraic properties of the induced homomorphism could be proved using its properties as a group homomorphism, but it's generally easier to prove them using the algebraic properties of the lift via this lemma

<proof>

lemma *hom-induced-hom:* $\text{GroupHom } G \ F$

<proof>

lemma *induced-hom-Abs-freeletter-equality:*

$s \in S \implies F (\lceil \text{FreeGroup } S | \text{Abs-freeletter } s | Q \rceil) = f \ s$

<proof>

lemma *uniqueness-of-induced-hom':*

defines $q \equiv \text{Group.natural-quotient-hom } (\text{FreeGroup } S) \ Q$

assumes $\text{GroupHom } G \ T \ \forall s \in S. \ T (\lceil \text{FreeGroup } S | \text{Abs-freeletter } s | Q \rceil) = f \ s$

shows $T \circ q = F \circ q$

<proof>

lemma *uniqueness-of-induced-hom:*

assumes $\text{GroupHom } G \ T \ \forall s \in S. \ T (\lceil \text{FreeGroup } S | \text{Abs-freeletter } s | Q \rceil) = f \ s$

shows $T = F$

<proof>

theorem *induced-hom-universal-property:*

$\exists ! F. \ \text{GroupHom } G \ F \ \wedge \ (\forall s \in S. \ F (\lceil \text{FreeGroup } S | \text{Abs-freeletter } s | Q \rceil) = f \ s)$

<proof>

lemma *induced-hom-Abs-freelist-conv-sum-list:*

$ss \in \text{lists } S \implies F (\lceil \text{FreeGroup } S | \text{Abs-freelist } ss | Q \rceil) = (\sum s \leftarrow ss. \ f \ s)$

<proof>

lemma *induced-hom-surj:* $F'G = \langle f'S \rangle$

<proof>

end

Groups affording a presentation The locale *GroupByPresentation* allows the construction of a *Group* out of any type from a set of generating letters and a set of relator words in (signed) letters. The following locale concerns the question of when the *Group* generated by a set in class *group-add* is isomorphic to a group presentation.

locale *GroupWithGeneratorsRelators* =

fixes $S :: 'g::\text{group-add set}$ — the set of generators

and $R :: 'g \ \text{list set}$ — the set of relator words

assumes *relators*: $rs \in R \implies rs \in \text{lists } (S \cup \text{uminus } 'S)$
 $rs \in R \implies \text{sum-list } rs = 0$
 $rs \in R \implies \text{proper-signed-list } (\text{map } (\text{charpair } S) rs)$

begin

abbreviation $P \equiv \text{map } (\text{charpair } S) 'R$
abbreviation $P' \equiv \text{GroupByPresentation}.P' P$
abbreviation $Q \equiv \text{GroupByPresentation}.Q S P$
abbreviation $G \equiv \text{GroupByPresentation}.G S P$
abbreviation *relator-freeword* $rs \equiv \text{Abs-freeword } (\text{map } (\text{charpair } S) rs)$
— this maps R onto P'

abbreviation *freeliftid* $\equiv \text{freeword-funlift id}$

abbreviation *induced-id* $:: 'g \text{ freeword set permutation} \Rightarrow 'g$
where *induced-id* $\equiv \text{GroupByPresentationInducedFun}.induced-hom S P id$

lemma *GroupByPresentation-S-P*: $\text{GroupByPresentation } S P$
<proof>

lemmas *G-UN* = $\text{GroupByPresentation}.G\text{-UN}[OF \text{GroupByPresentation-S-P}]$
lemmas *P'-FreeS* = $\text{GroupByPresentation}.P'\text{-FreeS}[OF \text{GroupByPresentation-S-P}]$

lemma *freeliftid-trivial-relator-freeword-R*:
 $rs \in R \implies \text{freeliftid } (\text{relator-freeword } rs) = 0$
<proof>

lemma *freeliftid-trivial-P*: $ps \in P \implies \text{freeliftid } (\text{Abs-freeword } ps) = 0$
<proof>

lemma *GroupByPresentationInducedFun-S-P-id*:
 $\text{GroupByPresentationInducedFun } S P id$
<proof>

lemma *induced-id-Abs-freelist-conv-sum-list*:
 $ss \in \text{lists } S \implies \text{induced-id } ([\text{FreeGroup } S | \text{Abs-freelist } ss | Q]) = \text{sum-list } ss$
<proof>

lemma *lconj-relator-freeword-R*:
 $\llbracket rs \in R; \text{proper-signed-list } xs; \text{fst } ' \text{ set } xs \subseteq S \rrbracket \implies$
 $\text{lconjby } (\text{Abs-freeword } xs) (\text{relator-freeword } rs) \in Q$
<proof>

lemma *rconj-relator-freeword*:
assumes $rs \in R \text{ proper-signed-list } xs \text{ fst } ' \text{ set } xs \subseteq S$
shows $\text{rconjby } (\text{Abs-freeword } xs) (\text{relator-freeword } rs) \in Q$
<proof>

lemma *lconjby-Abs-freelist-relator-freeword*:

$\llbracket rs \in R; xs \in \text{lists } S \rrbracket \implies \text{lconjby } (\text{Abs-freelist } xs) (\text{relator-freeword } rs) \in Q$
 <proof>

Here we record that the lift of the identity map to the free group on S induces a homomorphic surjection onto the group generated by S from the group presentation on S , subject to the same relations as the elements of S .

theorem *induced-id-hom-surj*: $\text{GroupHom } G \text{ induced-id induced-id } \langle G = \langle S \rangle$
 <proof>

end

locale *GroupPresentation* = *GroupWithGeneratorsRelators* $S R$

for $S :: 'g::\text{group-add set}$ — the set of generators

and $R :: 'g \text{ list set}$ — the set of relator words

+ **assumes** *induced-id-inj*: *inj-on induced-id* G

begin

abbreviation *inv-induced-id* \equiv *the-inv-into* G *induced-id*

lemma *inv-induced-id-sum-list-S*:

$ss \in \text{lists } S \implies \text{inv-induced-id } (\text{sum-list } ss) = (\llbracket \text{FreeGroup } S | \text{Abs-freelist } ss | Q \rrbracket)$

<proof>

end

2.10 Words over a generating set

Here we gather the necessary constructions and facts for studying a group generated by some set in terms of words in the generators.

context *monoid-add*

begin

abbreviation *word-for* $A a as \equiv as \in \text{lists } A \wedge \text{sum-list } as = a$

definition *reduced-word-for* $:: 'a \text{ set} \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$

where *reduced-word-for* $A a as \equiv \text{is-arg-min length } (\text{word-for } A a) as$

abbreviation *reduced-word* $A as \equiv \text{reduced-word-for } A (\text{sum-list } as) as$

abbreviation *reduced-words-for* $A a \equiv \text{Collect } (\text{reduced-word-for } A a)$

abbreviation *reduced-letter-set* $:: 'a \text{ set} \Rightarrow 'a \Rightarrow 'a \text{ set}$

where *reduced-letter-set* $A a \equiv \bigcup (\text{set } \langle (\text{reduced-words-for } A a) \rangle)$

— will be empty if a is not in the set generated by A

definition *word-length* $:: 'a \text{ set} \Rightarrow 'a \Rightarrow \text{nat}$

where *word-length* $A a \equiv \text{length } (\text{arg-min length } (\text{word-for } A a))$

lemma *reduced-word-forI*:

assumes $as \in \text{lists } A \text{ sum-list } as = a$
 $\bigwedge bs. bs \in \text{lists } A \implies \text{sum-list } bs = a \implies \text{length } as \leq \text{length } bs$
shows $\text{reduced-word-for } A \ a \ as$
 $\langle \text{proof} \rangle$

lemma *reduced-word-forI-compare*:
 $\llbracket \text{reduced-word-for } A \ a \ as; bs \in \text{lists } A; \text{sum-list } bs = a; \text{length } bs = \text{length } as \rrbracket$
 $\implies \text{reduced-word-for } A \ a \ bs$
 $\langle \text{proof} \rangle$

lemma *reduced-word-for-lists*: $\text{reduced-word-for } A \ a \ as \implies as \in \text{lists } A$
 $\langle \text{proof} \rangle$

lemma *reduced-word-for-sum-list*: $\text{reduced-word-for } A \ a \ as \implies \text{sum-list } as = a$
 $\langle \text{proof} \rangle$

lemma *reduced-word-for-minimal*:
 $\llbracket \text{reduced-word-for } A \ a \ as; bs \in \text{lists } A; \text{sum-list } bs = a \rrbracket \implies$
 $\text{length } as \leq \text{length } bs$
 $\langle \text{proof} \rangle$

lemma *reduced-word-for-length*:
 $\text{reduced-word-for } A \ a \ as \implies \text{length } as = \text{word-length } A \ a$
 $\langle \text{proof} \rangle$

lemma *reduced-word-for-eq-length*:
 $\text{reduced-word-for } A \ a \ as \implies \text{reduced-word-for } A \ a \ bs \implies \text{length } as = \text{length } bs$
 $\langle \text{proof} \rangle$

lemma *reduced-word-for-arg-min*:
 $as \in \text{lists } A \implies \text{sum-list } as = a \implies$
 $\text{reduced-word-for } A \ a \ (\text{arg-min length } (\text{word-for } A \ a))$
 $\langle \text{proof} \rangle$

lemma *nil-reduced-word-for-0*: $\text{reduced-word-for } A \ 0 \ []$
 $\langle \text{proof} \rangle$

lemma *reduced-word-for-0-imp-nil*: $\text{reduced-word-for } A \ 0 \ as \implies as = []$
 $\langle \text{proof} \rangle$

lemma *not-reduced-word-for*:
 $\llbracket bs \in \text{lists } A; \text{sum-list } bs = a; \text{length } bs < \text{length } as \rrbracket \implies$
 $\neg \text{reduced-word-for } A \ a \ as$
 $\langle \text{proof} \rangle$

lemma *reduced-word-for-imp-reduced-word*:
 $\text{reduced-word-for } A \ a \ as \implies \text{reduced-word } A \ as$
 $\langle \text{proof} \rangle$

lemma *sum-list-zero-nreduced*:

$as \neq [] \implies \text{sum-list } as = 0 \implies \neg \text{reduced-word } A \ as$
<proof>

lemma *order2-nreduced*: $a+a=0 \implies \neg \text{reduced-word } A \ [a,a]$

<proof>

lemma *reduced-word-append-reduce-contr1*:

assumes $\neg \text{reduced-word } A \ as$

shows $\neg \text{reduced-word } A \ (as@bs)$

<proof>

lemma *reduced-word-append-reduce-contr2*:

assumes $\neg \text{reduced-word } A \ bs$

shows $\neg \text{reduced-word } A \ (as@bs)$

<proof>

lemma *contains-nreduced-imp-nreduced*:

$\neg \text{reduced-word } A \ bs \implies \neg \text{reduced-word } A \ (as@bs@cs)$

<proof>

lemma *contains-order2-nreduced*: $a+a=0 \implies \neg \text{reduced-word } A \ (as@[a,a]@bs)$

<proof>

lemma *reduced-word-Cons-reduce-contr*:

$\neg \text{reduced-word } A \ as \implies \neg \text{reduced-word } A \ (a\#as)$

<proof>

lemma *reduced-word-Cons-reduce*: $\text{reduced-word } A \ (a\#as) \implies \text{reduced-word } A \ as$

<proof>

lemma *reduced-word-singleton*:

assumes $a \in A \ a \neq 0$

shows $\text{reduced-word } A \ [a]$

<proof>

lemma *el-reduced*:

assumes $0 \notin A \ as \in \text{lists } A \ \text{sum-list } as \in A \ \text{reduced-word } A \ as$

shows $\text{length } as = 1$

<proof>

lemma *reduced-letter-set-0*: $\text{reduced-letter-set } A \ 0 = \{\}$

<proof>

lemma *reduced-letter-set-subset*: $\text{reduced-letter-set } A \ a \subseteq A$

<proof>

lemma *reduced-word-forI-length*:

$\llbracket as \in \text{lists } A; \text{sum-list } as = a; \text{length } as = \text{word-length } A \ a \rrbracket \implies$

reduced-word-for A a as
<proof>

lemma *word-length-le:*

as ∈ lists A ⇒ sum-list as = a ⇒ word-length A a ≤ length as
<proof>

lemma *reduced-word-forI-length':*

[[as ∈ lists A; sum-list as = a; length as ≤ word-length A a]] ⇒
reduced-word-for A a as
<proof>

lemma *word-length-lt:*

as ∈ lists A ⇒ sum-list as = a ⇒ ¬ reduced-word-for A a as ⇒
word-length A a < length as
<proof>

end

lemma *in-genby-reduced-letter-set:*

assumes *as ∈ lists A sum-list as = a*
shows *a ∈ <reduced-letter-set A a>*
<proof>

lemma *reduced-word-for-genby-arg-min:*

fixes *A :: 'a::group-add set*
defines *B ≡ A ∪ uminus ' A*
assumes *a ∈ <A>*
shows *reduced-word-for B a (arg-min length (word-for B a))*
<proof>

lemma *reduced-word-for-genby-sym-arg-min:*

assumes *uminus ' A ⊆ A a ∈ <A>*
shows *reduced-word-for A a (arg-min length (word-for A a))*
<proof>

lemma *in-genby-imp-in-reduced-letter-set:*

fixes *A :: 'a::group-add set*
defines *B ≡ A ∪ uminus ' A*
assumes *a ∈ <A>*
shows *a ∈ <reduced-letter-set B a>*
<proof>

lemma *in-genby-sym-imp-in-reduced-letter-set:*

uminus ' A ⊆ A ⇒ a ∈ <A> ⇒ a ∈ <reduced-letter-set A a>
<proof>

end

3 Simplicial complexes

In this section we develop the basic theory of abstract simplicial complexes as a collection of finite sets, where the power set of each member set is contained in the collection. Note that in this development we allow the empty simplex, since allowing it or not seemed of no logical consequence, but of some small practical consequence.

theory *Simplicial*

imports *Prelim*

begin

3.1 Geometric notions

The geometric notions attached to a simplicial complex of main interest to us are those of facets (subsets of codimension one), adjacency (sharing a facet in common), and chains of adjacent simplices.

3.1.1 Facets

definition *facetrel* :: 'a set \Rightarrow 'a set \Rightarrow bool (**infix** \triangleleft 60)

where $y \triangleleft x \equiv \exists v. v \notin y \wedge x = \text{insert } v \ y$

lemma *facetrelI*: $v \notin y \Longrightarrow x = \text{insert } v \ y \Longrightarrow y \triangleleft x$

<proof>

lemma *facetrelI-card*: $y \subseteq x \Longrightarrow \text{card } (x - y) = 1 \Longrightarrow y \triangleleft x$

<proof>

lemma *facetrel-complement-vertex*: $y \triangleleft x \Longrightarrow x = \text{insert } v \ y \Longrightarrow v \notin y$

<proof>

lemma *facetrel-diff-vertex*: $v \in x \Longrightarrow x - \{v\} \triangleleft x$

<proof>

lemma *facetrel-conv-insert*: $y \triangleleft x \Longrightarrow v \in x - y \Longrightarrow x = \text{insert } v \ y$

<proof>

lemma *facetrel-psubset*: $y \triangleleft x \Longrightarrow y \subset x$

<proof>

lemma *facetrel-subset*: $y \triangleleft x \Longrightarrow y \subseteq x$

<proof>

lemma *facetrel-card*: $y \triangleleft x \Longrightarrow \text{card } (x - y) = 1$

<proof>

lemma *finite-facetrel-card*: $\text{finite } x \Longrightarrow y \triangleleft x \Longrightarrow \text{card } x = \text{Suc } (\text{card } y)$

<proof>

lemma *facetrelI-cardSuc*: $z \subseteq x \implies \text{card } x = \text{Suc } (\text{card } z) \implies z \triangleleft x$
<proof>

lemma *facet2-subset*: $\llbracket z \triangleleft x; z \triangleleft y; x \cap y - z \neq \{\} \rrbracket \implies x \subseteq y$
<proof>

lemma *inj-on-pullback-facet*:
 assumes *inj-on* f x $z \triangleleft f'x$
 obtains y **where** $y \triangleleft x$ $f'y = z$
<proof>

3.1.2 Adjacency

definition *adjacent* :: $'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$ (**infix** \sim 70)
 where $x \sim y \equiv \exists z. z \triangleleft x \wedge z \triangleleft y$

lemma *adjacentI*: $z \triangleleft x \implies z \triangleleft y \implies x \sim y$
<proof>

lemma *empty-not-adjacent*: $\neg \{\} \sim x$
<proof>

lemma *adjacent-sym*: $x \sim y \implies y \sim x$
<proof>

lemma *adjacent-refl*:
 assumes $x \neq \{\}$
 shows $x \sim x$
<proof>

lemma *common-facet*: $\llbracket z \triangleleft x; z \triangleleft y; x \neq y \rrbracket \implies z = x \cap y$
<proof>

lemma *adjacent-int-facet1*: $x \sim y \implies x \neq y \implies (x \cap y) \triangleleft x$
<proof>

lemma *adjacent-int-facet2*: $x \sim y \implies x \neq y \implies (x \cap y) \triangleleft y$
<proof>

lemma *adjacent-conv-insert*: $x \sim y \implies v \in x - y \implies x = \text{insert } v (x \cap y)$
<proof>

lemma *adjacent-int-decomp*:
 $x \sim y \implies x \neq y \implies \exists v. v \notin y \wedge x = \text{insert } v (x \cap y)$
<proof>

lemma *adj-antivertex*:

assumes $x \sim y$ $x \neq y$
shows $\exists! v. v \in x - y$
 ⟨proof⟩

lemma *adjacent-card*: $x \sim y \implies \text{card } x = \text{card } y$
 ⟨proof⟩

lemma *adjacent-to-adjacent-int-subset*:
assumes $C \sim D$ $f^{\cdot}C \sim f^{\cdot}D$ $f^{\cdot}C \neq f^{\cdot}D$
shows $f^{\cdot}C \cap f^{\cdot}D \subseteq f^{\cdot}(C \cap D)$
 ⟨proof⟩

lemma *adjacent-to-adjacent-int*:
 [$C \sim D$; $f^{\cdot}C \sim f^{\cdot}D$; $f^{\cdot}C \neq f^{\cdot}D$] $\implies f^{\cdot}(C \cap D) = f^{\cdot}C \cap f^{\cdot}D$
 ⟨proof⟩

3.1.3 Chains of adjacent sets

abbreviation *adjacentchain* \equiv *binrelchain adjacent*
abbreviation *padjacentchain* \equiv *proper-binrelchain adjacent*

lemmas *adjacentchain-Cons-reduce* = *binrelchain-Cons-reduce* [of adjacent]
lemmas *adjacentchain-obtain-proper* = *binrelchain-obtain-proper* [of - - adjacent]

lemma *adjacentchain-card*: *adjacentchain* ($x \# xs @ [y]$) $\implies \text{card } x = \text{card } y$
 ⟨proof⟩

3.2 Locale and basic facts

locale *SimplicialComplex* =
fixes $X :: 'a \text{ set set}$
assumes *finite-simplices*: $\forall x \in X. \text{finite } x$
and *faces* : $x \in X \implies y \subseteq x \implies y \in X$

context *SimplicialComplex*
begin

abbreviation *Subcomplex* $Y \equiv Y \subseteq X \wedge \text{SimplicialComplex } Y$

definition *maxsimp* $x \equiv x \in X \wedge (\forall z \in X. x \subseteq z \longrightarrow z = x)$

definition *adjacentset* :: $'a \text{ set} \Rightarrow 'a \text{ set set}$
where *adjacentset* $x = \{y \in X. x \sim y\}$

lemma *finite-simplex*: $x \in X \implies \text{finite } x$
 ⟨proof⟩

lemma *singleton-simplex*: $v \in \bigcup X \implies \{v\} \in X$
 ⟨proof⟩

lemma *maxsimpI*: $x \in X \implies (\bigwedge z. z \in X \implies x \subseteq z \implies z = x) \implies \text{maxsimp } x$
 ⟨proof⟩

lemma *maxsimpD-simplex*: $\text{maxsimp } x \implies x \in X$
 ⟨proof⟩

lemma *maxsimpD-maximal*: $\text{maxsimp } x \implies z \in X \implies x \subseteq z \implies z = x$
 ⟨proof⟩

lemmas *finite-maxsimp = finite-simplex*[OF *maxsimpD-simplex*]

lemma *maxsimp-nempty*: $X \neq \{\{\}\} \implies \text{maxsimp } x \implies x \neq \{\}$
 ⟨proof⟩

lemma *maxsimp-vertices*: $\text{maxsimp } x \implies x \subseteq \bigcup X$
 ⟨proof⟩

lemma *adjacentsetD-adj*: $y \in \text{adjacentset } x \implies x \sim y$
 ⟨proof⟩

lemma *max-in-subcomplex*:
 [*Subcomplex* Y ; $y \in Y$; $\text{maxsimp } y$] $\implies \text{SimplicialComplex.maxsimp } Y y$
 ⟨proof⟩

lemma *face-im*:
assumes $w \in X$ $y \subseteq f'w$
defines $u \equiv \{a \in w. f a \in y\}$
shows $y \in f \vdash X$
 ⟨proof⟩

lemma *im-faces*: $x \in f \vdash X \implies y \subseteq x \implies y \in f \vdash X$
 ⟨proof⟩

lemma *map-is-simplicial-morph*: *SimplicialComplex* $(f \vdash X)$
 ⟨proof⟩

lemma *vertex-set-int*:
assumes *SimplicialComplex* Y
shows $\bigcup (X \cap Y) = \bigcup X \cap \bigcup Y$
 ⟨proof⟩

end

3.3 Chains of maximal simplices

Chains of maximal simplices (with respect to adjacency) will allow us to walk through chamber complexes. But there is much we can say about them in simplicial complexes. We will call a chain of maximal simplices proper (using the prefix p as a naming convention to denote proper) if no

maximal simplex appears more than once in the chain. (Some sources elect to call improper chains prechains, and reserve the name chain to describe a proper chain. And usually a slightly weaker notion of proper is used, requiring only that no maximal simplex appear twice in succession. But it essentially makes no difference, and we found it easier to use *distinct* rather than *binrelchain* (\neq .)

context *SimplicialComplex*
begin

definition *maxsimpchain* $xs \equiv (\forall x \in \text{set } xs. \text{maxsimp } x) \wedge \text{adjacentchain } xs$

definition *pmaxsimpchain* $xs \equiv (\forall x \in \text{set } xs. \text{maxsimp } x) \wedge \text{padjacentchain } xs$

function *min-maxsimpchain* $:: 'a \text{ set list} \Rightarrow \text{bool}$

where

min-maxsimpchain [] = *True*

| *min-maxsimpchain* [x] = *maxsimp* x

| *min-maxsimpchain* (x#xs@[y]) =

($x \neq y \wedge \text{is-arg-min length } (\lambda zs. \text{maxsimpchain } (x\#zs@[y])) \text{ } xs$)

<proof>

termination *<proof>*

lemma *maxsimpchain-snocI*:

$\llbracket \text{maxsimpchain } (xs@[x]); \text{maxsimp } y; x \sim y \rrbracket \Longrightarrow \text{maxsimpchain } (xs@[x,y])$

<proof>

lemma *maxsimpchainD-maxsimp*:

$\text{maxsimpchain } xs \Longrightarrow x \in \text{set } xs \Longrightarrow \text{maxsimp } x$

<proof>

lemma *maxsimpchainD-adj*: $\text{maxsimpchain } xs \Longrightarrow \text{adjacentchain } xs$

<proof>

lemma *maxsimpchain-CConsI*:

$\llbracket \text{maxsimp } w; \text{maxsimpchain } (x\#xs); w \sim x \rrbracket \Longrightarrow \text{maxsimpchain } (w\#x\#xs)$

<proof>

lemma *maxsimpchain-Cons-reduce*:

$\text{maxsimpchain } (x\#xs) \Longrightarrow \text{maxsimpchain } xs$

<proof>

lemma *maxsimpchain-append-reduce1*:

$\text{maxsimpchain } (xs@ys) \Longrightarrow \text{maxsimpchain } xs$

<proof>

lemma *maxsimpchain-append-reduce2*:

$\text{maxsimpchain } (xs@ys) \Longrightarrow \text{maxsimpchain } ys$

<proof>

lemma *maxsimpchain-remdup-adj*:

$maxsimpchain (xs@[x,x]@ys) \implies maxsimpchain (xs@[x]@ys)$

<proof>

lemma *maxsimpchain-rev*: $maxsimpchain xs \implies maxsimpchain (rev xs)$

<proof>

lemma *maxsimpchain-overlap-join*:

$maxsimpchain (xs@[w]) \implies maxsimpchain (w\#ys) \implies$

$maxsimpchain (xs@w\#ys)$

<proof>

lemma *pmaxsimpchain*: $pmaxsimpchain xs \implies maxsimpchain xs$

<proof>

lemma *pmaxsimpchainI-maxsimpchain*:

$maxsimpchain xs \implies distinct xs \implies pmaxsimpchain xs$

<proof>

lemma *pmaxsimpchain-CConsI*:

$\llbracket maxsimp w; pmaxsimpchain (x\#xs); w \sim x; w \notin set (x\#xs) \rrbracket \implies$

$pmaxsimpchain (w\#x\#xs)$

<proof>

lemmas *pmaxsimpchainD-maxsimp =*

maxsimpchainD-maxsimp [OF pmaxsimpchain]

lemmas *pmaxsimpchainD-adj =*

maxsimpchainD-adj [OF pmaxsimpchain]

lemma *pmaxsimpchainD-distinct*: $pmaxsimpchain xs \implies distinct xs$

<proof>

lemma *pmaxsimpchain-Cons-reduce*:

$pmaxsimpchain (x\#xs) \implies pmaxsimpchain xs$

<proof>

lemma *pmaxsimpchain-append-reduce1*:

$pmaxsimpchain (xs@ys) \implies pmaxsimpchain xs$

<proof>

lemma *maxsimpchain-obtain-pmaxsimpchain*:

assumes $x \neq y$ $maxsimpchain (x\#xs@[y])$

shows $\exists ys. set ys \subseteq set xs \wedge length ys \leq length xs \wedge$

$pmaxsimpchain (x\#ys@[y])$

<proof>

lemma *min-maxsimpchainD-maxsimpchain*:

assumes *min-maxsimpchain xs*

shows *maxsimpchain xs*

<proof>

lemma *min-maxsimpchainD-min-betw:*

*min-maxsimpchain (x#xs@[y]) \implies maxsimpchain (x#ys@[y]) \implies
length ys \geq length xs*

<proof>

lemma *min-maxsimpchainI-betw:*

assumes *x#y maxsimpchain (x#xs@[y])*

\wedge ys. maxsimpchain (x#ys@[y]) \implies length xs \leq length ys

shows *min-maxsimpchain (x#xs@[y])*

<proof>

lemma *min-maxsimpchainI-betw-compare:*

assumes *x#y maxsimpchain (x#xs@[y])*

min-maxsimpchain (x#ys@[y]) length xs = length ys

shows *min-maxsimpchain (x#xs@[y])*

<proof>

lemma *min-maxsimpchain-pmaxsimpchain:*

assumes *min-maxsimpchain xs*

shows *pmaxsimpchain xs*

<proof>

lemma *min-maxsimpchain-rev:*

assumes *min-maxsimpchain xs*

shows *min-maxsimpchain (rev xs)*

<proof>

lemma *min-maxsimpchain-adj:*

[maxsimp x; maxsimp y; x~y; x#y] \implies min-maxsimpchain [x,y]

<proof>

lemma *min-maxsimpchain-betw-CCons-reduce:*

assumes *min-maxsimpchain (w#x#ys@[z])*

shows *min-maxsimpchain (x#ys@[z])*

<proof>

lemma *min-maxsimpchain-betw-uniform-length:*

assumes *min-maxsimpchain (x#xs@[y]) min-maxsimpchain (x#ys@[y])*

shows *length xs = length ys*

<proof>

lemma *not-min-maxsimpchainI-betw:*

[maxsimpchain (x#ys@[y]); length ys < length xs] \implies

\neg min-maxsimpchain (x#xs@[y])

<proof>

lemma *maxsimpchain-in-subcomplex:*

[[*Subcomplex* Y ; set $ys \subseteq Y$; *maxsimpchain* ys]] \implies
SimplicialComplex.maxsimpchain Y ys
 <proof>

end

3.4 Isomorphisms of simplicial complexes

Here we develop the concept of isomorphism of simplicial complexes. Note that we have not bothered to first develop the concept of morphism of simplicial complexes, since every function on the vertex set of a simplicial complex can be considered a morphism of complexes (see lemma *map-is-simplicial-morph* above).

locale *SimplicialComplexIsomorphism* = *SimplicialComplex* X
 for $X :: 'a$ set set
 + fixes $f :: 'a \Rightarrow 'b$
 assumes *inj*: *inj-on* f ($\bigcup X$)
begin

lemmas *morph* = *map-is-simplicial-morph*[*of* f]

lemma *iso-codim-map*:
 $x \in X \implies y \in X \implies \text{card } (f'x - f'y) = \text{card } (x - y)$
 <proof>

lemma *maxsimp-im-max*: *maxsimp* $x \implies w \in X \implies f'x \subseteq f'w \implies f'w = f'x$
 <proof>

lemma *maxsimp-map*:
maxsimp $x \implies \text{SimplicialComplex.maxsimp } (f \vdash X) (f'x)$
 <proof>

lemma *iso-adj-int-im*:
 assumes *maxsimp* x *maxsimp* y $x \sim y$ $x \neq y$
 shows $(f'x \cap f'y) \triangleleft f'x$
 <proof>

lemma *iso-adj-map*:
 assumes *maxsimp* x *maxsimp* y $x \sim y$ $x \neq y$
 shows $f'x \sim f'y$
 <proof>

lemma *pmaxsimpchain-map*:
pmaxsimpchain $xs \implies \text{SimplicialComplex.pmaxsimpchain } (f \vdash X) (f \models xs)$
 <proof>

end

3.5 The complex associated to a poset

A simplicial complex is naturally a poset under the subset relation. The following develops the reverse direction: constructing a simplicial complex from a suitable poset.

context *ordering*
begin

definition *PosetComplex* :: 'a set \Rightarrow 'a set set
where *PosetComplex* $P \equiv (\bigcup x \in P. \{ \{y. \text{pseudominimal-in } (P.\leq x) y\} \})$

lemma *poset-is-SimplicialComplex*:
assumes $\forall x \in P. \text{simplex-like } (P.\leq x)$
shows *SimplicialComplex* (*PosetComplex* P)
<proof>

definition *poset-simplex-map* :: 'a set \Rightarrow 'a \Rightarrow 'a set
where *poset-simplex-map* $P x = \{y. \text{pseudominimal-in } (P.\leq x) y\}$

lemma *poset-to-PosetComplex-OrderingSetMap*:
assumes $\bigwedge x. x \in P \Longrightarrow \text{simplex-like } (P.\leq x)$
shows *OrderingSetMap* (\leq) ($<$) (\subseteq) (\subset) P (*poset-simplex-map* P)
<proof>

end

When a poset affords a simplicial complex, there is a natural morphism of posets from the source poset into the poset of sets in the complex, as above. However, some further assumptions are necessary to ensure that this morphism is an isomorphism. These conditions are collected in the following locale.

locale *ComplexLikePoset* = *ordering less-eq less*
for *less-eq* :: 'a \Rightarrow 'a \Rightarrow bool (**infix** \leq 50)
and *less* :: 'a \Rightarrow 'a \Rightarrow bool (**infix** $<$ 50)
+ **fixes** $P :: 'a \text{ set}$
assumes *below-in-P-simplex-like*: $x \in P \Longrightarrow \text{simplex-like } (P.\leq x)$
and *P-has-bottom* : *has-bottom* P
and *P-has-glbs* : $x \in P \Longrightarrow y \in P \Longrightarrow \exists b. \text{glbound-in-of } P x y b$
begin

abbreviation *smap* $\equiv \text{poset-simplex-map } P$

lemma *smap-onto-PosetComplex*: *smap* ' $P = \text{PosetComplex } P$
<proof>

lemma *ordsetmap-smap*: $\llbracket a \in P; b \in P; a \leq b \rrbracket \Longrightarrow \text{smap } a \subseteq \text{smap } b$
<proof>

lemma *inj-on-smap: inj-on smap P*
 ⟨proof⟩

lemma *OrderingSetIso-smap:*
OrderingSetIso (\leq) ($<$) (\subseteq) (\subset) *P smap*
 ⟨proof⟩

lemmas *rev-ordsetmap-smap =*
OrderingSetIso.rev-ordsetmap[OF OrderingSetIso-smap]

end

end

4 Chamber complexes

Now we develop the basic theory of chamber complexes, including both thin and thick complexes. Some terminology: a maximal simplex is now called a chamber, and a chain (with respect to adjacency) of chambers is now called a gallery. A gallery in which no chamber appears more than once is called proper, and we use the prefix *p* as a naming convention to denote proper. Again, we remind the reader that some sources reserve the name gallery for (a slightly weaker notion of) what we are calling a proper gallery, using *pregallery* to denote an improper gallery.

theory *Chamber*
imports *Algebra Simplicial*

begin

4.1 Locale definition and basic facts

locale *ChamberComplex = SimplicialComplex X*
for *X :: 'a set set*
 + **assumes** *simplex-in-max* : $y \in X \implies \exists x. \text{maxsimp } x \wedge y \subseteq x$
and *maxsimp-connect*: $\llbracket x \neq y; \text{maxsimp } x; \text{maxsimp } y \rrbracket \implies$
 $\exists xs. \text{maxsimpchain } (x \# xs @ [y])$

context *ChamberComplex*
begin

abbreviation *chamber* $\equiv \text{maxsimp}$
abbreviation *gallery* $\equiv \text{maxsimpchain}$
abbreviation *pgallery* $\equiv \text{pmaxsimpchain}$
abbreviation *min-gallery* $\equiv \text{min-maxsimpchain}$
abbreviation *supchamber* $v \equiv (\text{SOME } C. \text{chamber } C \wedge v \in C)$

lemmas *faces* $= \text{faces}$

lemmas *singleton-simplex* = *singleton-simplex*
lemmas *chamberI* = *maxsimpI*
lemmas *chamberD-simplex* = *maxsimpD-simplex*
lemmas *chamberD-maximal* = *maxsimpD-maximal*
lemmas *finite-chamber* = *finite-maxsimp*
lemmas *chamber-nempty* = *maxsimp-nempty*
lemmas *chamber-vertices* = *maxsimp-vertices*
lemmas *gallery-def* = *maxsimpchain-def*
lemmas *gallery-snocI* = *maxsimpchain-snocI*
lemmas *galleryD-chamber* = *maxsimpchainD-maxsimp*
lemmas *galleryD-adj* = *maxsimpchainD-adj*
lemmas *gallery-CConsI* = *maxsimpchain-CConsI*
lemmas *gallery-Cons-reduce* = *maxsimpchain-Cons-reduce*
lemmas *gallery-append-reduce1* = *maxsimpchain-append-reduce1*
lemmas *gallery-append-reduce2* = *maxsimpchain-append-reduce2*
lemmas *gallery-remdup-adj* = *maxsimpchain-remdup-adj*
lemmas *gallery-obtain-pgallery* = *maxsimpchain-obtain-pmaxsimpchain*
lemmas *pgallery-def* = *pmaxsimpchain-def*
lemmas *pgalleryI-gallery* = *pmaxsimpchainI-maxsimpchain*
lemmas *pgalleryD-chamber* = *pmaxsimpchainD-maxsimp*
lemmas *pgalleryD-adj* = *pmaxsimpchainD-adj*
lemmas *pgalleryD-distinct* = *pmaxsimpchainD-distinct*
lemmas *pgallery-Cons-reduce* = *pmaxsimpchain-Cons-reduce*
lemmas *pgallery-append-reduce1* = *pmaxsimpchain-append-reduce1*
lemmas *pgallery* = *pmaxsimpchain*
lemmas *min-gallery-simps* = *min-maxsimpchain.simps*
lemmas *min-galleryI-betw* = *min-maxsimpchainI-betw*
lemmas *min-galleryI-betw-compare* = *min-maxsimpchainI-betw-compare*
lemmas *min-galleryD-min-betw* = *min-maxsimpchainD-min-betw*
lemmas *min-galleryD-gallery* = *min-maxsimpchainD-maxsimpchain*
lemmas *min-gallery-pgallery* = *min-maxsimpchain-pmaxsimpchain*
lemmas *min-gallery-rev* = *min-maxsimpchain-rev*
lemmas *min-gallery-adj* = *min-maxsimpchain-adj*
lemmas *not-min-galleryI-betw* = *not-min-maxsimpchainI-betw*

lemmas *min-gallery-betw-CCons-reduce* =
min-maxsimpchain-betw-CCons-reduce
lemmas *min-gallery-betw-uniform-length* =
min-maxsimpchain-betw-uniform-length
lemmas *vertex-set-int* = *vertex-set-int[OF ChamberComplex.axioms(1)]*

lemma *chamber-pconnect*:
 $\llbracket x \neq y; \text{chamber } x; \text{chamber } y \rrbracket \implies \exists xs. \text{pgallery } (x\#xs@[y])$
<proof>

lemma *supchamberD*:
assumes $v \in \bigcup X$
defines $C \equiv \text{supchamber } v$
shows *chamber* $C \ v \in C$

<proof>

definition

$\text{ChamberSubcomplex } Y \equiv Y \subseteq X \wedge \text{ChamberComplex } Y \wedge$
 $(\forall C. \text{ChamberComplex.chamber } Y C \longrightarrow \text{chamber } C)$

lemma *ChamberSubcomplexI:*

assumes $Y \subseteq X \text{ ChamberComplex } Y$

$\wedge y. \text{ChamberComplex.chamber } Y y \Longrightarrow \text{chamber } y$

shows $\text{ChamberSubcomplex } Y$

<proof>

lemma *ChamberSubcomplexD-sub:* $\text{ChamberSubcomplex } Y \Longrightarrow Y \subseteq X$

<proof>

lemma *ChamberSubcomplexD-complex:*

$\text{ChamberSubcomplex } Y \Longrightarrow \text{ChamberComplex } Y$

<proof>

lemma *chambersub-imp-sub:* $\text{ChamberSubcomplex } Y \Longrightarrow \text{Subcomplex } Y$

<proof>

lemma *chamber-in-subcomplex:*

$\llbracket \text{ChamberSubcomplex } Y; C \in Y; \text{chamber } C \rrbracket \Longrightarrow$

$\text{ChamberComplex.chamber } Y C$

<proof>

lemma *subcomplex-chamber:*

$\text{ChamberSubcomplex } Y \Longrightarrow \text{ChamberComplex.chamber } Y C \Longrightarrow \text{chamber } C$

<proof>

lemma *gallery-in-subcomplex:*

$\llbracket \text{ChamberSubcomplex } Y; \text{set } ys \subseteq Y; \text{gallery } ys \rrbracket \Longrightarrow$

$\text{ChamberComplex.gallery } Y ys$

<proof>

lemma *subcomplex-gallery:*

$\text{ChamberSubcomplex } Y \Longrightarrow \text{ChamberComplex.gallery } Y Cs \Longrightarrow \text{gallery } Cs$

<proof>

lemma *subcomplex-pgallery:*

$\text{ChamberSubcomplex } Y \Longrightarrow \text{ChamberComplex.pgallery } Y Cs \Longrightarrow \text{pgallery } Cs$

<proof>

lemma *min-gallery-in-subcomplex:*

assumes $\text{ChamberSubcomplex } Y \text{ min-gallery } Cs \text{ set } Cs \subseteq Y$

shows $\text{ChamberComplex.min-gallery } Y Cs$

<proof>

lemma *chamber-card*: $\text{chamber } C \implies \text{chamber } D \implies \text{card } C = \text{card } D$
 ⟨*proof*⟩

lemma *chamber-facet-is-chamber-facet*:
 [$\text{chamber } C; \text{chamber } D; z \triangleleft C; z \subseteq D$] $\implies z \triangleleft D$
 ⟨*proof*⟩

lemma *chamber-adj*:
assumes $\text{chamber } C \ D \in X \ C \sim D$
shows $\text{chamber } D$
 ⟨*proof*⟩

lemma *chambers-share-facet*:
assumes $\text{chamber } C \ \text{chamber } (\text{insert } v \ z) \ z \triangleleft C$
shows $z \triangleleft \text{insert } v \ z$
 ⟨*proof*⟩

lemma *adjacentset-chamber*: $\text{chamber } C \implies D \in \text{adjacentset } C \implies \text{chamber } D$
 ⟨*proof*⟩

lemma *chamber-shared-facet*: [$\text{chamber } C; z \triangleleft C; D \in X; z \triangleleft D$] $\implies \text{chamber } D$
 ⟨*proof*⟩

lemma *adjacentset-conv-facetchambersets*:
assumes $X \neq \{\{\}\}$ $\text{chamber } C$
shows $\text{adjacentset } C = (\bigcup v \in C. \{D \in X. C - \{v\} \triangleleft D\})$
 ⟨*proof*⟩

end

4.2 The system of chambers and distance between chambers

We now examine the system of all chambers in more detail, and explore the distance function on this system provided by lengths of minimal galleries.

context *ChamberComplex*
begin

definition *chamber-system* :: 'a set set
where $\text{chamber-system} \equiv \{C. \text{chamber } C\}$
abbreviation $C \equiv \text{chamber-system}$

definition *chamber-distance* :: 'a set \Rightarrow 'a set \Rightarrow nat
where $\text{chamber-distance } C \ D =$
 (if $C=D$ then 0 else
 Suc (length (ARG-MIN length Cs. gallery (C#Cs@[D]))))

definition *closest-supchamber* :: 'a set \Rightarrow 'a set \Rightarrow 'a set
where $\text{closest-supchamber } F \ D =$
 (ARG-MIN ($\lambda C. \text{chamber-distance } C \ D$)) C.

chamber $C \wedge F \subseteq C$)

definition *face-distance* $F D \equiv$ *chamber-distance* (*closest-supchamber* $F D$) D

lemma *chamber-system-simplices*: $C \subseteq X$
<proof>

lemma *gallery-chamber-system*: *gallery* $Cs \implies$ *set* $Cs \subseteq C$
<proof>

lemmas *pgallery-chamber-system* = *gallery-chamber-system*[*OF pgallery*]

lemma *chamber-distance-le*:
gallery ($C \# Cs @ [D]$) \implies *chamber-distance* $C D \leq$ *Suc* (*length* Cs)
<proof>

lemma *min-gallery-betw-chamber-distance*:
min-gallery ($C \# Cs @ [D]$) \implies *chamber-distance* $C D =$ *Suc* (*length* Cs)
<proof>

lemma *min-galleryI-chamber-distance-betw*:
gallery ($C \# Cs @ [D]$) \implies *Suc* (*length* Cs) = *chamber-distance* $C D \implies$
min-gallery ($C \# Cs @ [D]$)
<proof>

lemma *gallery-least-length*:
assumes *chamber* C *chamber* D $C \neq D$
defines $Cs \equiv$ *ARG-MIN length* Cs . *gallery* ($C \# Cs @ [D]$)
shows *gallery* ($C \# Cs @ [D]$)
<proof>

lemma *min-gallery-least-length*:
assumes *chamber* C *chamber* D $C \neq D$
defines $Cs \equiv$ *ARG-MIN length* Cs . *gallery* ($C \# Cs @ [D]$)
shows *min-gallery* ($C \# Cs @ [D]$)
<proof>

lemma *pgallery-least-length*:
assumes *chamber* C *chamber* D $C \neq D$
defines $Cs \equiv$ *ARG-MIN length* Cs . *gallery* ($C \# Cs @ [D]$)
shows *pgallery* ($C \# Cs @ [D]$)
<proof>

lemma *closest-supchamberD*:
assumes $F \in X$ *chamber* D
shows *chamber* (*closest-supchamber* $F D$) $F \subseteq$ *closest-supchamber* $F D$
<proof>

lemma *closest-supchamber-closest*:

chamber $C \implies F \subseteq C \implies$
chamber-distance (*closest-supchamber* $F D$) $D \leq$ *chamber-distance* $C D$
 ⟨*proof*⟩

lemma *face-distance-le*:
chamber $C \implies F \subseteq C \implies$ *face-distance* $F D \leq$ *chamber-distance* $C D$
 ⟨*proof*⟩

lemma *face-distance-eq-0*: *chamber* $C \implies F \subseteq C \implies$ *face-distance* $F C = 0$
 ⟨*proof*⟩

end

4.3 Labelling a chamber complex

A labelling of a chamber complex is a function on the vertex set so that each chamber is in bijective correspondence with the label set (chambers all have the same number of vertices).

context *ChamberComplex*
begin

definition *label-wrt* :: 'b set \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool
where *label-wrt* $B f \equiv (\forall C \in \mathcal{C}. \text{bij-betw } f C B)$

lemma *label-wrtD*: *label-wrt* $B f \implies C \in \mathcal{C} \implies \text{bij-betw } f C B$
 ⟨*proof*⟩

lemma *label-wrtD'*: *label-wrt* $B f \implies \text{chamber } C \implies \text{bij-betw } f C B$
 ⟨*proof*⟩

lemma *label-wrt-adjacent*:
assumes *label-wrt* $B f$ *chamber* C *chamber* D $C \sim D$ $v \in C - D$ $w \in D - C$
shows $f v = f w$
 ⟨*proof*⟩

lemma *label-wrt-adjacent-shared-facet*:
 [*label-wrt* $B f$; *chamber* (*insert* $v z$); *chamber* (*insert* $w z$); $v \notin z$; $w \notin z$] \implies
 $f v = f w$
 ⟨*proof*⟩

lemma *label-wrt-elt-image*: *label-wrt* $B f \implies v \in \bigcup X \implies f v \in B$
 ⟨*proof*⟩

end

4.4 Morphisms of chamber complexes

While any function on the vertex set of a simplicial complex can be considered a morphism of simplicial complexes onto its image, for chamber complexes we require the function send chambers onto chambers of the same cardinality in some chamber complex of the codomain.

4.4.1 Morphism locale and basic facts

locale *ChamberComplexMorphism* = *domain*: *ChamberComplex* *X* + *codomain*: *ChamberComplex* *Y*

for *X* :: 'a set set
and *Y* :: 'b set set
+ **fixes** *f* :: 'a ⇒ 'b
assumes *chamber-map*: *domain.chamber C* ⇒ *codomain.chamber (f·C)*
and *dim-map* : *domain.chamber C* ⇒ *card (f·C) = card C*

lemma (in *ChamberComplex*) *trivial-morphism*:
ChamberComplexMorphism X X id
⟨*proof*⟩

lemma (in *ChamberComplex*) *inclusion-morphism*:
assumes *ChamberSubcomplex Y*
shows *ChamberComplexMorphism Y X id*
⟨*proof*⟩

context *ChamberComplexMorphism*
begin

lemmas *domain-complex* = *domain.ChamberComplex-axioms*

lemmas *codomain-complex* = *codomain.ChamberComplex-axioms*

lemmas *simplicialcomplex-image* = *domain.map-is-simplicial-morph*[of *f*]

lemma *cong*: *fun-eq-on g f (⋃ X)* ⇒ *ChamberComplexMorphism X Y g*
⟨*proof*⟩

lemma *comp*:
assumes *ChamberComplexMorphism Y Z g*
shows *ChamberComplexMorphism X Z (g·f)*
⟨*proof*⟩

lemma *restrict-domain*:
assumes *domain.ChamberSubcomplex W*
shows *ChamberComplexMorphism W Y f*
⟨*proof*⟩

lemma *restrict-codomain*:
assumes *codomain.ChamberSubcomplex Z f·X ⊆ Z*

shows *ChamberComplexMorphism* $X Z f$
<proof>

lemma *inj-on-chamber*: $\text{domain.chamber } C \implies \text{inj-on } f C$
<proof>

lemma *bij-betw-chambers*: $\text{domain.chamber } C \implies \text{bij-betw } f C (f'C)$
<proof>

lemma *card-map*: $x \in X \implies \text{card } (f'x) = \text{card } x$
<proof>

lemma *codim-map*:
assumes $\text{domain.chamber } C y \subseteq C$
shows $\text{card } (f'C - f'y) = \text{card } (C - y)$
<proof>

lemma *simplex-map*: $x \in X \implies f'x \in Y$
<proof>

lemma *simplices-map*: $f \vdash X \subseteq Y$
<proof>

lemma *vertex-map*: $x \in \bigcup X \implies f x \in \bigcup Y$
<proof>

lemma *facet-map*: $\text{domain.chamber } C \implies z \triangleleft C \implies f'z \triangleleft f'C$
<proof>

lemma *adj-int-im*:
assumes $\text{domain.chamber } C \text{ domain.chamber } D C \sim D f'C \neq f'D$
shows $(f'C \cap f'D) \triangleleft f'C$
<proof>

lemma *adj-map'*:
assumes $\text{domain.chamber } C \text{ domain.chamber } D C \sim D f'C \neq f'D$
shows $f'C \sim f'D$
<proof>

lemma *adj-map*:
[[$\text{domain.chamber } C; \text{domain.chamber } D; C \sim D$] $\implies f'C \sim f'D$
<proof>

lemma *chamber-vertex-outside-facet-image*:
assumes $v \notin z \text{ domain.chamber } (\text{insert } v z)$
shows $f v \notin f'z$
<proof>

lemma *expand-codomain*:

assumes *ChamberComplex* Z *ChamberComplex.ChamberSubcomplex* Z Y
shows *ChamberComplexMorphism* X Z f
 ⟨*proof*⟩

end

4.4.2 Action on pregalleries and galleries

context *ChamberComplexMorphism*
begin

lemma *gallery-map*: *domain.gallery* $Cs \implies$ *codomain.gallery* $(f \models Cs)$
 ⟨*proof*⟩

lemma *gallery-betw-map*:
domain.gallery $(C \# Cs @ [D]) \implies$ *codomain.gallery* $(f' C \# f \models Cs @ [f' D])$
 ⟨*proof*⟩

end

4.4.3 Properties of the image

context *ChamberComplexMorphism*
begin

lemma *subcomplex-image*: *codomain.Subcomplex* $(f \vdash X)$
 ⟨*proof*⟩

lemmas *chamber-in-image* = *codomain.max-in-subcomplex*[*OF subcomplex-image*]

lemma *maxsimp-map-into-image*:
assumes *domain.chamber* x
shows *SimplicialComplex.maxsimp* $(f \vdash X)$ $(f' x)$
 ⟨*proof*⟩

lemma *maxsimp-preimage*:
assumes $C \in X$ *SimplicialComplex.maxsimp* $(f \vdash X)$ $(f' C)$
shows *domain.chamber* C
 ⟨*proof*⟩

lemma *chamber-preimage*:
 $C \in X \implies$ *codomain.chamber* $(f' C) \implies$ *domain.chamber* C
 ⟨*proof*⟩

lemma *chambercomplex-image*: *ChamberComplex* $(f \vdash X)$
 ⟨*proof*⟩

lemma *chambersubcomplex-image*: *codomain.ChamberSubcomplex* $(f \vdash X)$
 ⟨*proof*⟩

lemma *restrict-codomain-to-image*: $\text{ChamberComplexMorphism } X \ (f \vdash X) \ f$
 ⟨proof⟩

end

4.4.4 Action on the chamber system

context $\text{ChamberComplexMorphism}$

begin

lemma *chamber-system-into*: $f \vdash \text{domain.C} \subseteq \text{codomain.C}$
 ⟨proof⟩

lemma *chamber-system-image*: $f \vdash \text{domain.C} = \text{codomain.C} \cap (f \vdash X)$
 ⟨proof⟩

lemma *image-chamber-system*: $\text{ChamberComplex.C } (f \vdash X) = f \vdash \text{domain.C}$
 ⟨proof⟩

lemma *image-chamber-system-image*:
 $\text{ChamberComplex.C } (f \vdash X) = \text{codomain.C} \cap (f \vdash X)$
 ⟨proof⟩

lemma *face-distance-eq-chamber-distance-map*:
assumes $\text{domain.chamber } C \ \text{domain.chamber } D \ C \neq D \ z \subseteq C$
 $\text{codomain.face-distance } (f'z) \ (f'D) = \text{domain.face-distance } z \ D$
 $\text{domain.face-distance } z \ D = \text{domain.chamber-distance } C \ D$
shows $\text{codomain.face-distance } (f'z) \ (f'D) =$
 $\text{codomain.chamber-distance } (f'C) \ (f'D)$
 ⟨proof⟩

lemma *face-distance-eq-chamber-distance-min-gallery-betw-map*:
assumes $\text{domain.chamber } C \ \text{domain.chamber } D \ C \neq D \ z \subseteq C$
 $\text{codomain.face-distance } (f'z) \ (f'D) = \text{domain.face-distance } z \ D$
 $\text{domain.face-distance } z \ D = \text{domain.chamber-distance } C \ D$
 $\text{domain.min-gallery } (C \# Cs@[D])$
shows $\text{codomain.min-gallery } (f'=(C \# Cs@[D]))$
 ⟨proof⟩

end

4.4.5 Isomorphisms

locale $\text{ChamberComplexIsomorphism} = \text{ChamberComplexMorphism } X \ Y \ f$

for $X :: 'a \ \text{set} \ \text{set}$

and $Y :: 'b \ \text{set} \ \text{set}$

and $f :: 'a \Rightarrow 'b$

+ **assumes** *bij-betw-vertices*: $\text{bij-betw } f \ (\bigcup X) \ (\bigcup Y)$

and *surj-simplex-map* : $f \vdash X = Y$

lemma (in *ChamberComplexIsomorphism*) *inj*: *inj-on* f $(\bigcup X)$
⟨*proof*⟩

sublocale *ChamberComplexIsomorphism* < *SimplicialComplexIsomorphism*
⟨*proof*⟩

lemma (in *ChamberComplex*) *trivial-isomorphism*:
ChamberComplexIsomorphism X X *id*
⟨*proof*⟩

lemma (in *ChamberComplexMorphism*) *isoI-inverse*:
assumes *ChamberComplexMorphism* Y X g
fixespointwise $(g \circ f)$ $(\bigcup X)$ *fixespointwise* $(f \circ g)$ $(\bigcup Y)$
shows *ChamberComplexIsomorphism* X Y f
⟨*proof*⟩

context *ChamberComplexIsomorphism*
begin

lemmas *domain-complex* = *domain-complex*
lemmas *chamber-map* = *chamber-map*
lemmas *dim-map* = *dim-map*
lemmas *gallery-map* = *gallery-map*
lemmas *simplex-map* = *simplex-map*
lemmas *chamber-preimage* = *chamber-preimage*

lemma *chamber-morphism*: *ChamberComplexMorphism* X Y f ⟨*proof*⟩

lemma *pgallery-map*: *domain.pgallery* $Cs \implies$ *codomain.pgallery* $(f \models Cs)$
⟨*proof*⟩

lemma *iso-cong*:
assumes *fun-eq-on* g f $(\bigcup X)$
shows *ChamberComplexIsomorphism* X Y g
⟨*proof*⟩

lemma *iso-comp*:
assumes *ChamberComplexIsomorphism* Y Z g
shows *ChamberComplexIsomorphism* X Z $(g \circ f)$
⟨*proof*⟩

lemma *inj-on-chamber-system*: *inj-on* $((\cdot) f)$ *domain.C*
⟨*proof*⟩

lemma *inv*: *ChamberComplexIsomorphism* Y X (*the-inv-into* $(\bigcup X)$ f)
⟨*proof*⟩

lemma *chamber-distance-map*:
assumes *domain.chamber* C *domain.chamber* D

shows $\text{codomain.chamber-distance } (f^{\cdot}C) (f^{\cdot}D) =$
 $\text{domain.chamber-distance } C D$
 ⟨proof⟩

lemma *face-distance-map*:
assumes $\text{domain.chamber } C F \in X$
shows $\text{codomain.face-distance } (f^{\cdot}F) (f^{\cdot}C) = \text{domain.face-distance } F C$
 ⟨proof⟩

end

4.4.6 Endomorphisms

locale *ChamberComplexEndomorphism* = *ChamberComplexMorphism* $X X f$
for $X :: 'a \text{ set set}$
and $f :: 'a \Rightarrow 'a$
 + **assumes** *trivial-outside* : $v \notin \bigcup X \implies f v = v$
 — to facilitate uniqueness arguments

lemma (in *ChamberComplex*) *trivial-endomorphism*:
 $\text{ChamberComplexEndomorphism } X \text{ id}$
 ⟨proof⟩

context *ChamberComplexEndomorphism*
begin

abbreviation *ChamberSubcomplex* $\equiv \text{domain.ChamberSubcomplex}$

abbreviation *Subcomplex* $\equiv \text{domain.Subcomplex}$

abbreviation *chamber* $\equiv \text{domain.chamber}$

abbreviation *gallery* $\equiv \text{domain.gallery}$

abbreviation *C* $\equiv \text{domain.chamber-system}$

abbreviation *label-wrt* $\equiv \text{domain.label-wrt}$

lemmas *dim-map* = *dim-map*

lemmas *simplex-map* = *simplex-map*

lemmas *vertex-map* = *vertex-map*

lemmas *chamber-map* = *chamber-map*

lemmas *adj-map* = *adj-map*

lemmas *facet-map* = *facet-map*

lemmas *bij-betw-chambers* = *bij-betw-chambers*

lemmas *chamber-system-into* = *chamber-system-into*

lemmas *chamber-system-image* = *chamber-system-image*

lemmas *image-chamber-system* = *image-chamber-system*

lemmas *chambercomplex-image* = *chambercomplex-image*

lemmas *chambersubcomplex-image* = *chambersubcomplex-image*

lemmas *face-distance-eq-chamber-distance-map* =
face-distance-eq-chamber-distance-map

lemmas *face-distance-eq-chamber-distance-min-gallery-betw-map* =

face-distance-eq-chamber-distance-min-gallery-betw-map
lemmas *facelist-chdist-mingal-btwmap* =
face-distance-eq-chamber-distance-min-gallery-betw-map

lemmas *trivial-endorphism* = *domain.trivial-endorphism*
lemmas *finite-simplices* = *domain.finite-simplices*
lemmas *faces* = *domain.faces*
lemmas *maxsimp-connect* = *domain.maxsimp-connect*
lemmas *simplex-in-max* = *domain.simplex-in-max*
lemmas *chamberD-simplex* = *domain.chamberD-simplex*
lemmas *chamber-system-def* = *domain.chamber-system-def*
lemmas *chamber-system-simplices* = *domain.chamber-system-simplices*
lemmas *galleryD-chamber* = *domain.galleryD-chamber*
lemmas *galleryD-adj* = *domain.galleryD-adj*
lemmas *gallery-append-reduce1* = *domain.gallery-append-reduce1*
lemmas *gallery-Cons-reduce* = *domain.gallery-Cons-reduce*
lemmas *gallery-chamber-system* = *domain.gallery-chamber-system*
lemmas *label-wrtD* = *domain.label-wrtD*
lemmas *label-wrt-adjacent* = *domain.label-wrt-adjacent*

lemma *endo-comp*:
assumes *ChamberComplexEndomorphism X g*
shows *ChamberComplexEndomorphism X (g∘f)*
<proof>

lemma *restrict-endo*:
assumes *ChamberSubcomplex Y f† Y ⊆ Y*
shows *ChamberComplexEndomorphism Y (restrict1 f (⋃ Y))*
<proof>

lemma *funpower-endorphism*:
ChamberComplexEndomorphism X (f^{~n})
<proof>

end

lemma (**in** *ChamberComplex*) *fold-chamber-complex-endorph-list*:
 $\forall x \in \text{set } xs. \text{ChamberComplexEndomorphism } X (f \ x) \implies$
 $\text{ChamberComplexEndomorphism } X (\text{fold } f \ xs)$
<proof>

context *ChamberComplexEndomorphism*
begin

lemma *split-gallery*:
 $\llbracket C \in f \dagger C; D \in C - f \dagger C; \text{gallery } (C \# Cs @ [D]) \rrbracket \implies$
 $\exists As \ A \ B \ Bs. A \in f \dagger C \wedge B \in C - f \dagger C \wedge C \# Cs @ [D] = As @ A \# B \# Bs$
<proof>

lemma *respects-labels-adjacent*:
assumes *label-wrt B φ chamber C chamber D C~D* $\forall v \in C. \varphi (f v) = \varphi v$
shows $\forall v \in D. \varphi (f v) = \varphi v$
 ⟨*proof*⟩

lemma *respects-labels-gallery*:
assumes *label-wrt B φ* $\forall v \in C. \varphi (f v) = \varphi v$
shows *gallery (C#Cs@[D])* $\implies \forall v \in D. \varphi (f v) = \varphi v$
 ⟨*proof*⟩

lemma *respect-label-fix-chamber-imp-fun-eq-on*:
assumes *label : label-wrt B φ*
and *chamber: chamber C f' C = g' C*
and *respect: $\forall v \in C. \varphi (f v) = \varphi v \forall v \in C. \varphi (g v) = \varphi v$*
shows *fun-eq-on f g C*
 ⟨*proof*⟩

lemmas *respects-label-fixes-chamber-imp-fixespointwise =*
respect-label-fix-chamber-imp-fun-eq-on[of - - id, simplified]

end

4.4.7 Automorphisms

locale *ChamberComplexAutomorphism = ChamberComplexIsomorphism X X f*
for *X :: 'a set set*
and *f :: 'a \Rightarrow 'a*
 + **assumes** *trivial-outside : $v \notin \bigcup X \implies f v = v$*
 — to facilitate uniqueness arguments

sublocale *ChamberComplexAutomorphism < ChamberComplexEndomorphism*
 ⟨*proof*⟩

lemma (**in** *ChamberComplex*) *trivial-automorphism*:
ChamberComplexAutomorphism X id
 ⟨*proof*⟩

context *ChamberComplexAutomorphism*
begin

lemmas *facet-map = facet-map*
lemmas *chamber-map = chamber-map*
lemmas *chamber-morphism = chamber-morphism*
lemmas *bij-betw-vertices = bij-betw-vertices*
lemmas *surj-simplex-map = surj-simplex-map*

lemma *bij: bij f*
 ⟨*proof*⟩

lemma *comp*:
assumes *ChamberComplexAutomorphism X g*
shows *ChamberComplexAutomorphism X (g ∘ f)*
⟨*proof*⟩

lemma *equality*:
assumes *ChamberComplexAutomorphism X g fun-eq-on f g (⋃ X)*
shows $f = g$
⟨*proof*⟩

end

4.4.8 Retractions

A retraction of a chamber complex is an endomorphism that is the identity on its image.

locale *ChamberComplexRetraction = ChamberComplexEndomorphism X f*
for $X :: 'a \text{ set set}$
and $f :: 'a \Rightarrow 'a$
+ **assumes** *retraction: $v \in \bigcup X \implies f (f v) = f v$*
begin

lemmas *simplex-map = simplex-map*
lemmas *chamber-map = chamber-map*
lemmas *gallery-map = gallery-map*

lemma *vertex-retraction: $v \in f'(\bigcup X) \implies f v = v$*
⟨*proof*⟩

lemma *simplex-retraction1: $x \in f \vdash X \implies \text{fixespointwise } f x$*
⟨*proof*⟩

lemma *simplex-retraction2: $x \in f \vdash X \implies f'x = x$*
⟨*proof*⟩

lemma *chamber-retraction1: $C \in f \vdash C \implies \text{fixespointwise } f C$*
⟨*proof*⟩

lemma *chamber-retraction2: $C \in f \vdash C \implies f'C = C$*
⟨*proof*⟩

lemma *respects-labels*:
assumes *label-wrt B $\varphi v \in (\bigcup X)$*
shows $\varphi (f v) = \varphi v$
⟨*proof*⟩

end

4.4.9 Foldings of chamber complexes

A folding of a chamber complex is a retraction that literally folds the complex in half, in that each chamber in the image is the image of precisely two chambers: itself (since a folding is a retraction) and a unique chamber outside the image.

Locale definition Here we define the locale and collect some lemmas inherited from the *ChamberComplexRetraction* locale.

locale *ChamberComplexFolding* = *ChamberComplexRetraction* *X f*

for *X* :: 'a set set

and *f* :: 'a ⇒ 'a

+ **assumes** *folding*:

$chamber\ C \implies C \in f \vdash X \implies$

$\exists ! D. chamber\ D \wedge D \notin f \vdash X \wedge f'D = C$

begin

lemmas *folding-ex* = *ex1-implies-ex*[*OF folding*]

lemmas *chamber-system-into* = *chamber-system-into*

lemmas *gallery-map* = *gallery-map*

lemmas *chamber-retraction1* = *chamber-retraction1*

lemmas *chamber-retraction2* = *chamber-retraction2*

end

Decomposition into half chamber systems and half apartments

Here we describe how a folding splits the chamber system of the complex into its image and the complement of its image. The chamber subcomplex consisting of all simplices contained in a chamber of a given half of the chamber system is called a half-apartment.

context *ChamberComplexFolding*

begin

definition *opp-half-apartment* :: 'a set set

where *opp-half-apartment* ≡ {*x* ∈ *X*. ∃ *C* ∈ *C* - *f* ⊢ *C*. *x* ⊆ *C*}

abbreviation *Y* ≡ *opp-half-apartment*

lemma *opp-half-apartment-subset-complex*: *Y* ⊆ *X*

⟨*proof*⟩

lemma *simplicialcomplex-opp-half-apartment*: *SimplicialComplex Y*

⟨*proof*⟩

lemma *subcomplex-opp-half-apartment*: *Subcomplex Y*

⟨*proof*⟩

lemma *opp-half-apartmentI*: $\llbracket x \in X; C \in \mathcal{C} - f \vdash \mathcal{C}; x \subseteq C \rrbracket \implies x \in Y$
 ⟨proof⟩

lemma *opp-chambers-subset-opp-half-apartment*: $\mathcal{C} - f \vdash \mathcal{C} \subseteq Y$
 ⟨proof⟩

lemma *maxsimp-in-opp-half-apartment*:
 assumes *SimplicialComplex.maxsimp* $Y C$
 shows $C \in \mathcal{C} - f \vdash \mathcal{C}$
 ⟨proof⟩

lemma *chamber-in-opp-half-apartment*:
SimplicialComplex.maxsimp $Y C \implies \text{chamber } C$
 ⟨proof⟩

end

Mapping between half chamber systems for foldings Since each chamber in the image of the folding is the image of a unique chamber in the complement of the image, we obtain well-defined functions from one half chamber system to the other.

context *ChamberComplexFolding*
begin

abbreviation *opp-chamber* $C \equiv \text{THE } D. D \in \mathcal{C} - f \vdash \mathcal{C} \wedge f'D = C$
abbreviation *flop* $C \equiv \text{if } C \in f \vdash \mathcal{C} \text{ then opp-chamber } C \text{ else } f'C$

lemma *inj-on-opp-chambers'*:
 assumes *chamber* $C C \notin f \vdash X$ *chamber* $D D \notin f \vdash X f'C = f'D$
 shows $C = D$
 ⟨proof⟩

lemma *inj-on-opp-chambers''*:
 $\llbracket C \in \mathcal{C} - f \vdash \mathcal{C}; D \in \mathcal{C} - f \vdash \mathcal{C}; f'C = f'D \rrbracket \implies C = D$
 ⟨proof⟩

lemma *inj-on-opp-chambers*: *inj-on* $((\cdot) f) (\mathcal{C} - f \vdash \mathcal{C})$
 ⟨proof⟩

lemma *opp-chambers-surj*: $f \vdash (\mathcal{C} - (f \vdash \mathcal{C})) = f \vdash \mathcal{C}$
 ⟨proof⟩

lemma *opp-chambers-bij*: *bij-betw* $((\cdot) f) (\mathcal{C} - (f \vdash \mathcal{C})) (f \vdash \mathcal{C})$
 ⟨proof⟩

lemma *folding'*:
 assumes $C \in f \vdash \mathcal{C}$
 shows $\exists ! D \in \mathcal{C} - f \vdash \mathcal{C}. f'D = C$

<proof>

lemma *opp-chambers-distinct-map:*

set Cs ⊆ C-f⊢C ⇒ distinct Cs ⇒ distinct (f⊢Cs)

<proof>

lemma *opp-chamberD1: C ∈ f⊢C ⇒ opp-chamber C ∈ C-f⊢C*

<proof>

lemma *opp-chamberD2: C ∈ f⊢C ⇒ f'(opp-chamber C) = C*

<proof>

lemma *opp-chamber-reverse: C ∈ C-f⊢C ⇒ opp-chamber (f'C) = C*

<proof>

lemma *f-opp-chamber-list:*

set Cs ⊆ f⊢C ⇒ f⊢(map opp-chamber Cs) = Cs

<proof>

lemma *flop-chamber: chamber C ⇒ chamber (flop C)*

<proof>

end

4.5 Thin chamber complexes

A thin chamber complex is one in which every facet is a facet in exactly two chambers. Slightly more generally, we first consider the case of a chamber complex in which every facet is a facet of at most two chambers. One of the main results obtained at this point is the so-called standard uniqueness argument, which essentially states that two morphisms on a thin chamber complex that agree on a particular chamber must in fact agree on the entire complex. Following that, foldings of thin chamber complexes are investigated. In particular, we are interested in pairs of opposed foldings.

4.5.1 Locales and basic facts

locale *ThinishChamberComplex = ChamberComplex X*

for *X :: 'a set set*

+ **assumes** *thinish:*

[[chamber C; z ⊲ C; ∃ D ∈ X - {C}. z ⊲ D]] ⇒ ∃ ! D ∈ X - {C}. z ⊲ D

— being adjacent to a chamber, such a *D* would also be a chamber (see lemma *chamber-adj*)

begin

lemma *facet-unique-other-chamber:*

[[chamber B; z ⊲ B; chamber C; z ⊲ C; chamber D; z ⊲ D; C ≠ B; D ≠ B]]
⇒ C = D

<proof>

lemma *finite-adjacentset:*
 assumes *chamber C*
 shows *finite (adjacentset C)*
<proof>

lemma *label-wrt-eq-on-adjacent-vertex:*
 fixes *v v' :: 'a*
 and *z z' :: 'a set*
 defines *D : D ≡ insert v z*
 and *D' : D' ≡ insert v' z'*
 assumes *label : label-wrt B f f v = f v'*
 and *chambers: chamber C chamber D chamber D' z < C z' < C D ≠ C D' ≠ C*
 shows *D = D'*
<proof>

lemma *face-distance-eq-chamber-distance-compare-other-chamber:*
 assumes *chamber C chamber D z < C z < D C ≠ D*
 chamber-distance C E ≤ chamber-distance D E
 shows *face-distance z E = chamber-distance C E*
<proof>

end

lemma (**in** *ChamberComplexIsomorphism*) *thinish-image-shared-facet:*
 assumes *dom: domain.chamber C domain.chamber D z < C z < D C ≠ D*
 and *cod: ThinishChamberComplex Y codomain.chamber D' f'z < D'*
 D' ≠ f'C
 shows *f'D = D'*
<proof>

locale *ThinChamberComplex = ChamberComplex X*
 for *X :: 'a set set*
+ **assumes** *thin: chamber C ⇒ z < C ⇒ ∃! D ∈ X - {C}. z < D*

sublocale *ThinChamberComplex < ThinishChamberComplex*
<proof>

context *ThinChamberComplex*
begin

lemma *thinish: ThinishChamberComplex X <proof>*

lemmas *face-distance-eq-chamber-distance-compare-other-chamber =*
 face-distance-eq-chamber-distance-compare-other-chamber

abbreviation *the-adj-chamber C z ≡ THE D. D ∈ X - {C} ∧ z < D*

lemma *the-adj-chamber-simplex*:

chamber $C \implies z \triangleleft C \implies \text{the-adj-chamber } C z \in X$

<proof>

lemma *the-adj-chamber-facet*: *chamber* $C \implies z \triangleleft C \implies z \triangleleft \text{the-adj-chamber } C z$

<proof>

lemma *the-adj-chamber-is-adjacent*:

chamber $C \implies z \triangleleft C \implies C \sim \text{the-adj-chamber } C z$

<proof>

lemma *the-adj-chamber*:

chamber $C \implies z \triangleleft C \implies \text{chamber } (\text{the-adj-chamber } C z)$

<proof>

lemma *the-adj-chamber-neg*:

chamber $C \implies z \triangleleft C \implies \text{the-adj-chamber } C z \neq C$

<proof>

lemma *the-adj-chamber-adjacentset*:

chamber $C \implies z \triangleleft C \implies \text{the-adj-chamber } C z \in \text{adjacentset } C$

<proof>

end

lemmas (in *ChamberComplexIsomorphism*) *thin-image-shared-facet* =
thinish-image-shared-facet[*OF* - - - - *ThinChamberComplex.thinish*]

4.5.2 The standard uniqueness argument for chamber morphisms of thin chamber complexes

context *ThinishChamberComplex*

begin

lemma *standard-uniqueness-dbl*:

assumes *morph* : *ChamberComplexMorphism* $W X f$

ChamberComplexMorphism $W X g$

and *chambers*: *ChamberComplex.chamber* $W C$

ChamberComplex.chamber $W D$

$C \sim D f'D \neq f'C g'D \neq g'C \text{ chamber } (g'D)$

and *funeq* : *fun-eq-on* $f g C$

shows *fun-eq-on* $f g D$

<proof>

lemma *standard-uniqueness-pgallery-betw*:

assumes *morph* : *ChamberComplexMorphism* $W X f$

ChamberComplexMorphism $W X g$

and *chambers*: *fun-eq-on* $f g C \text{ ChamberComplex.gallery } W (C \# Cs@[D])$

pgallery ($f \models (C \# Cs@[D])$) *pgallery* ($g \models (C \# Cs@[D])$)

shows $\text{fun-eq-on } f \ g \ D$
 $\langle \text{proof} \rangle$

lemma *standard-uniqueness*:

assumes $\text{morph} : \text{ChamberComplexMorphism } W \ X \ f$
 $\text{ChamberComplexMorphism } W \ X \ g$

and $\text{chamber} : \text{ChamberComplex.chamber } W \ C \ \text{fun-eq-on } f \ g \ C$

and *map-gals*:

$\bigwedge Cs. \text{ChamberComplex.min-gallery } W \ (C \# \ Cs) \implies \text{pgallery } (f \models (C \# \ Cs))$

$\bigwedge Cs. \text{ChamberComplex.min-gallery } W \ (C \# \ Cs) \implies \text{pgallery } (g \models (C \# \ Cs))$

shows $\text{fun-eq-on } f \ g \ (\bigcup W)$

$\langle \text{proof} \rangle$

lemma *standard-uniqueness-isomorphs*:

assumes $\text{ChamberComplexIsomorphism } W \ X \ f$

$\text{ChamberComplexIsomorphism } W \ X \ g$

$\text{ChamberComplex.chamber } W \ C \ \text{fun-eq-on } f \ g \ C$

shows $\text{fun-eq-on } f \ g \ (\bigcup W)$

$\langle \text{proof} \rangle$

lemma *standard-uniqueness-automorphs*:

assumes $\text{ChamberComplexAutomorphism } X \ f$

$\text{ChamberComplexAutomorphism } X \ g$

$\text{chamber } C \ \text{fun-eq-on } f \ g \ C$

shows $f = g$

$\langle \text{proof} \rangle$

end

context *ThinChamberComplex*

begin

lemmas *standard-uniqueness* = *standard-uniqueness*

lemmas *standard-uniqueness-isomorphs* = *standard-uniqueness-isomorphs*

lemmas *standard-uniqueness-pgallery-betw* = *standard-uniqueness-pgallery-betw*

end

4.6 Foldings of thin chamber complexes

4.6.1 Locale definition and basic facts

locale *ThinChamberComplexFolding* =

ThinChamberComplex $X + \text{folding} : \text{ChamberComplexFolding } X \ f$

for $X :: 'a \ \text{set set}$

and $f :: 'a \Rightarrow 'a$

begin

abbreviation $\text{opp-chamber} \equiv \text{folding.opp-chamber}$

lemma *adjacent-half-chamber-system-image*:
assumes *chambers*: $C \in f \vdash \mathcal{C}$ $D \in \mathcal{C} - f \vdash \mathcal{C}$
and *adjacent*: $C \sim D$
shows $f'D = C$
 \langle *proof* \rangle

lemma *adjacent-half-chamber-system-image-reverse*:
 $\llbracket C \in f \vdash \mathcal{C}; D \in \mathcal{C} - f \vdash \mathcal{C}; C \sim D \rrbracket \implies \text{opp-chamber } C = D$
 \langle *proof* \rangle

lemma *chamber-image-closer*:
assumes $D \in \mathcal{C} - f \vdash \mathcal{C}$ $B \in f \vdash \mathcal{C}$ $B \neq f'D$ *gallery* $(B \# Ds @ [D])$
shows $\exists Cs. \text{gallery } (B \# Cs @ [f'D]) \wedge \text{length } Cs < \text{length } Ds$
 \langle *proof* \rangle

lemma *chamber-image-subset*:
assumes $D: D \in \mathcal{C} - f \vdash \mathcal{C}$
defines $C: C \equiv f'D$
defines *closerToC* $\equiv \{B \in \mathcal{C}. \text{chamber-distance } B \ C < \text{chamber-distance } B \ D\}$
shows $f \vdash \mathcal{C} \subseteq \text{closerToC}$
 \langle *proof* \rangle

lemma *gallery-double-cross-not-minimal-Cons1*:
 $\llbracket B \in f \vdash \mathcal{C}; C \in \mathcal{C} - f \vdash \mathcal{C}; D \in f \vdash \mathcal{C}; \text{gallery } (B \# C \# Cs @ [D]) \rrbracket \implies$
 $\neg \text{min-gallery } (B \# C \# Cs @ [D])$
 \langle *proof* \rangle

lemma *gallery-double-cross-not-minimal1*:
 $\llbracket B \in f \vdash \mathcal{C}; C \in \mathcal{C} - f \vdash \mathcal{C}; D \in f \vdash \mathcal{C}; \text{gallery } (B \# Bs @ C \# Cs @ [D]) \rrbracket \implies$
 $\neg \text{min-gallery } (B \# Bs @ C \# Cs @ [D])$
 \langle *proof* \rangle

end

locale *ThinChamberComplexFolding* =
ThinChamberComplex X + *folding*: *ChamberComplexFolding* X f
for $X :: 'a$ *set set*
and $f :: 'a \Rightarrow 'a$

sublocale *ThinChamberComplexFolding* < *ThinChamberComplexFolding* \langle *proof* \rangle

context *ThinChamberComplexFolding*
begin

abbreviation *flop* $\equiv \text{folding.flop}$

lemmas *adjacent-half-chamber-system-image* = *adjacent-half-chamber-system-image*
lemmas *gallery-double-cross-not-minimal1* = *gallery-double-cross-not-minimal1*

lemmas *gallery-double-cross-not-minimal-Cons1* =
gallery-double-cross-not-minimal-Cons1

lemma *adjacent-preimage*:
assumes *chambers*: $C \in \mathcal{C} - f \vdash \mathcal{C}$ $D \in \mathcal{C} - f \vdash \mathcal{C}$
and *adjacent*: $f' C \sim f' D$
shows $C \sim D$
 $\langle proof \rangle$

lemma *adjacent-opp-chamber*:
 $\llbracket C \in f \vdash \mathcal{C}; D \in f \vdash \mathcal{C}; C \sim D \rrbracket \implies \text{opp-chamber } C \sim \text{opp-chamber } D$
 $\langle proof \rangle$

lemma *adjacentchain-preimage*:
set $Cs \subseteq \mathcal{C} - f \vdash \mathcal{C} \implies \text{adjacentchain } (f \vdash Cs) \implies \text{adjacentchain } Cs$
 $\langle proof \rangle$

lemma *gallery-preimage*: *set* $Cs \subseteq \mathcal{C} - f \vdash \mathcal{C} \implies \text{gallery } (f \vdash Cs) \implies \text{gallery } Cs$
 $\langle proof \rangle$

lemma *chambercomplex-opp-half-apartment*: *ChamberComplex folding*. Y
 $\langle proof \rangle$

lemma *flop-adj*:
assumes *chamber* C *chamber* D $C \sim D$
shows $\text{flop } C \sim \text{flop } D$
 $\langle proof \rangle$

lemma *flop-gallery*: *gallery* $Cs \implies \text{gallery } (\text{map } \text{flop } Cs)$
 $\langle proof \rangle$

lemma *morphism-half-apartments*: *ChamberComplexMorphism folding*. Y $(f \vdash X)$ f
 $\langle proof \rangle$

lemma *chamber-image-complement-closer*:
 $\llbracket D \in \mathcal{C} - f \vdash \mathcal{C}; B \in \mathcal{C} - f \vdash \mathcal{C}; B \neq D; \text{gallery } (B \# Cs @ [f' D]) \rrbracket \implies$
 $\exists Ds. \text{gallery } (B \# Ds @ [D]) \wedge \text{length } Ds < \text{length } Cs$
 $\langle proof \rangle$

lemma *chamber-image-complement-subset*:
assumes $D: D \in \mathcal{C} - f \vdash \mathcal{C}$
defines $C: C \equiv f' D$
defines $\text{closerToD} \equiv \{B \in \mathcal{C}. \text{chamber-distance } B D < \text{chamber-distance } B C\}$
shows $\mathcal{C} - f \vdash \mathcal{C} \subseteq \text{closerToD}$
 $\langle proof \rangle$

lemma *chamber-image-and-complement*:
assumes $D: D \in \mathcal{C} - f \vdash \mathcal{C}$
defines $C: C \equiv f' D$

defines $closerToC \equiv \{B \in C. \text{chamber-distance } B \ C < \text{chamber-distance } B \ D\}$
and $closerToD \equiv \{B \in C. \text{chamber-distance } B \ D < \text{chamber-distance } B \ C\}$
shows $f \vdash C = closerToC \ C - f \vdash C = closerToD$
 $\langle proof \rangle$

end

4.6.2 Pairs of opposed foldings

A pair of foldings of a thin chamber complex are opposed or opposite if there is a corresponding pair of adjacent chambers, where each folding sends its corresponding chamber to the other chamber.

locale *OpposedThinChamberComplexFoldings* =
ThinChamberComplex *X*

+ *folding-f*: *ChamberComplexFolding* *X* *f*

+ *folding-g*: *ChamberComplexFolding* *X* *g*

for *X* :: 'a set set

and *f* :: 'a \Rightarrow 'a

and *g* :: 'a \Rightarrow 'a

+ **fixes** *C0* :: 'a set

assumes *chambers*: chamber *C0* $C0 \sim g \ C0 \ C0 \neq g \ C0 \ f \ g \ C0 = C0$

begin

abbreviation *D0* $\equiv g \ C0$

lemmas *chamber-D0* = *folding-g.chamber-map*[*OF chambers*(*1*)]

lemma *ThinChamberComplexFolding-f*: *ThinChamberComplexFolding* *X* *f* $\langle proof \rangle$

lemma *ThinChamberComplexFolding-g*: *ThinChamberComplexFolding* *X* *g* $\langle proof \rangle$

lemmas *foldf* = *ThinChamberComplexFolding-f*

lemmas *foldg* = *ThinChamberComplexFolding-g*

lemma *fg-symmetric*: *OpposedThinChamberComplexFoldings* *X* *g* *f* *D0*
 $\langle proof \rangle$

lemma *basechambers-half-chamber-systems*: $C0 \in f \vdash C \ D0 \in g \vdash C$
 $\langle proof \rangle$

lemmas *basech-halfchsys* =
basechambers-half-chamber-systems

lemma *f-trivial-C0*: $v \in C0 \implies f \ v = v$
 $\langle proof \rangle$

lemmas *g-trivial-D0* =
OpposedThinChamberComplexFoldings.f-trivial-C0[*OF fg-symmetric*]

lemma *double-fold-D0*:

assumes $v \in D0 - C0$
shows $g(fv) = v$
 ⟨proof⟩

lemmas *double-fold-C0* =
OpposedThinChamberComplexFoldings.double-fold-D0[*OF fg-symmetric*]

lemma *flopped-half-chamber-systems-fg*: $C - f \vdash C = g \vdash C$
 ⟨proof⟩

lemmas *flopped-half-chamber-systems-gf* =
OpposedThinChamberComplexFoldings.flopped-half-chamber-systems-fg[
OF fg-symmetric
]

lemma *flopped-half-apartments-fg*: *folding-f.opp-half-apartment* = $g \vdash X$
 ⟨proof⟩

lemmas *flopped-half-apartments-gf* =
OpposedThinChamberComplexFoldings.flopped-half-apartments-fg[
OF fg-symmetric
]

lemma *vertex-set-split*: $\bigcup X = f'(\bigcup X) \cup g'(\bigcup X)$
 — f and g will both be the identity on the intersection
 ⟨proof⟩

lemma *half-chamber-system-disjoint-union*:
 $C = f \vdash C \cup g \vdash C$ $(f \vdash C) \cap (g \vdash C) = \{\}$
 ⟨proof⟩

lemmas *halfchsys-decomp* =
half-chamber-system-disjoint-union

lemma *chamber-in-other-half-fg*: *chamber* $C \implies C \notin f \vdash C \implies C \in g \vdash C$
 ⟨proof⟩

lemma *adjacent-half-chamber-system-image-fg*:
 $C \in f \vdash C \implies D \in g \vdash C \implies C \sim D \implies f'D = C$
 ⟨proof⟩

lemmas *adjacent-half-chamber-system-image-gf* =
OpposedThinChamberComplexFoldings.adjacent-half-chamber-system-image-fg[
OF fg-symmetric
]

lemmas *adjhalfchsys-image-gf* =
adjacent-half-chamber-system-image-gf

lemma *switch-basechamber*:

assumes $C \in f \vdash \mathcal{C}$ $C \sim g' C$

shows *OpposedThinChamberComplexFoldings* $X f g C$

<proof>

lemma *unique-half-chamber-system-f*:

assumes *OpposedThinChamberComplexFoldings* $X f' g' C0 g'' C0 = D0$

shows $f \vdash \mathcal{C} = f' \vdash \mathcal{C}$

<proof>

lemma *unique-half-chamber-system-g*:

OpposedThinChamberComplexFoldings $X f' g' C0 \implies g'' C0 = D0 \implies$

$g \vdash \mathcal{C} = g' \vdash \mathcal{C}$

<proof>

lemma *split-gallery-fg*:

$\llbracket C \in f \vdash \mathcal{C}; D \in g \vdash \mathcal{C}; \text{gallery } (C \# Cs @ [D]) \rrbracket \implies$

$\exists As A B Bs. A \in f \vdash \mathcal{C} \wedge B \in g \vdash \mathcal{C} \wedge C \# Cs @ [D] = As @ A \# B \# Bs$

<proof>

lemmas *split-gallery-gf* =

OpposedThinChamberComplexFoldings.split-gallery-fg [*OF fg-symmetric*]

end

4.6.3 The automorphism induced by a pair of opposed foldings

Recall that a folding of a chamber complex is a special kind of chamber complex retraction, and so is the identity on its image. Hence a pair of opposed foldings will be the identity on the intersection of their images and so we can stitch them together to create an automorphism of the chamber complex, by allowing each folding to act on the complement of its image. This automorphism will be of order two, and will be the unique automorphism of the chamber complex that fixes pointwise the facet shared by the pair of adjacent chambers associated to the opposed foldings.

context *OpposedThinChamberComplexFoldings*

begin

definition *induced-automorphism* $:: 'a \Rightarrow 'a$

where *induced-automorphism* $v \equiv$

if $v \in f'(\bigcup X)$ *then* $g v$ *else if* $v \in g'(\bigcup X)$ *then* $f v$ *else* v

— f and g will both be the identity on the intersection of their images

abbreviation $s \equiv \text{induced-automorphism}$

lemma *induced-automorphism-fg-symmetric*:

$s = \text{OpposedThinChamberComplexFoldings.s } X g f$

<proof>

lemma *induced-automorphism-on-simplices-fg*: $x \in f \vdash X \implies v \in x \implies s \ v = g \ v$
(proof)

lemma *induced-automorphism-eq-foldings-on-chambers-fg*:
 $C \in f \vdash \mathcal{C} \implies \text{fun-eq-on } s \ g \ C$
(proof)

lemmas *indaut-eq-foldch-fg* =
induced-automorphism-eq-foldings-on-chambers-fg

lemma *induced-automorphism-eq-foldings-on-chambers-gf*:
 $C \in g \vdash \mathcal{C} \implies \text{fun-eq-on } s \ f \ C$
(proof)

lemma *induced-automorphism-on-chamber-vertices-f*:
chamber $C \implies v \in C \implies s \ v = (\text{if } C \in f \vdash \mathcal{C} \text{ then } g \ v \text{ else } f \ v)$
(proof)

lemma *induced-automorphism-simplex-image*:
 $C \in f \vdash \mathcal{C} \implies x \subseteq C \implies s \ 'x = g \ 'x \ C \in g \vdash \mathcal{C} \implies x \subseteq C \implies s \ 'x = f \ 'x$
(proof)

lemma *induced-automorphism-chamber-list-image-fg*:
set $Cs \subseteq f \vdash \mathcal{C} \implies s \models Cs = g \models Cs$
(proof)

lemma *induced-automorphism-chamber-image-fg*:
chamber $C \implies s \ 'C = (\text{if } C \in f \vdash \mathcal{C} \text{ then } g \ 'C \text{ else } f \ 'C)$
(proof)

lemma *induced-automorphism-C0*: $s \ 'C0 = D0$
(proof)

lemma *induced-automorphism-fixespointwise-C0-int-D0*:
fixespointwise $s \ (C0 \cap D0)$
(proof)

lemmas *indaut-fixes-fundfacet* =
induced-automorphism-fixespointwise-C0-int-D0

lemma *induced-automorphism-adjacent-half-chamber-system-image-fg*:
 $\llbracket C \in f \vdash \mathcal{C}; D \in g \vdash \mathcal{C}; C \sim D \rrbracket \implies s \ 'D = C$
(proof)

lemmas *indaut-adj-halfchsys-im-fg* =
induced-automorphism-adjacent-half-chamber-system-image-fg

lemma *induced-automorphism-chamber-map*: *chamber* $C \implies \text{chamber } (s \ 'C)$
(proof)

lemmas *indaut-chmap = induced-automorphism-chamber-map*

lemma *induced-automorphism-ntrivial: s ≠ id*
⟨proof⟩

lemma *induced-automorphism-bij-between-half-chamber-systems-f:*
bij-betw ((·) s) (C-f⊢C) (f⊢C)
⟨proof⟩

lemmas *indaut-bij-btw-halfchsys-f =*
induced-automorphism-bij-between-half-chamber-systems-f

lemma *induced-automorphism-bij-between-half-chamber-systems-g:*
bij-betw ((·) s) (C-g⊢C) (g⊢C)
⟨proof⟩

lemma *induced-automorphism-halfmorphism-fopp-to-fimage:*
ChamberComplexMorphism folding-f.opp-half-apartment (f⊢X) s
⟨proof⟩

lemmas *indaut-halfmorph-fopp-fim =*
induced-automorphism-halfmorphism-fopp-to-fimage

lemma *induced-automorphism-half-chamber-system-gallery-map-f:*
set Cs ⊆ f⊢C ⇒ gallery Cs ⇒ gallery (s|=Cs)
⟨proof⟩

lemma *induced-automorphism-half-chamber-system-pgallery-map-f:*
set Cs ⊆ f⊢C ⇒ pgallery Cs ⇒ pgallery (s|=Cs)
⟨proof⟩

lemmas *indaut-halfchsys-pgal-map-f =*
induced-automorphism-half-chamber-system-pgallery-map-f

lemma *induced-automorphism-half-chamber-system-pgallery-map-g:*
set Cs ⊆ g⊢C ⇒ pgallery Cs ⇒ pgallery (s|=Cs)
⟨proof⟩

lemma *induced-automorphism-halfmorphism-fimage-to-fopp:*
ChamberComplexMorphism (f⊢X) folding-f.opp-half-apartment s
⟨proof⟩

lemma *induced-automorphism-selfcomp-halfmorphism-f:*
ChamberComplexMorphism (f⊢X) (f⊢X) (sos)
⟨proof⟩

lemma *induced-automorphism-selfcomp-halftrivial-f: fixespointwise (sos) (⋃ (f⊢X))*
⟨proof⟩

lemmas *indaut-selfcomp-halftriv-f = induced-automorphism-selfcomp-halftrivial-f*

lemma *induced-automorphism-selfcomp-halftrivial-g: fixespointwise (sos) (U (g+ X))*
<proof>

lemma *induced-automorphism-trivial-outside:*
assumes $v \notin \bigcup X$
shows $s v = v$
<proof>

lemma *induced-automorphism-morphism: ChamberComplexEndomorphism X s*
<proof>

lemmas *indaut-morph = induced-automorphism-morphism*

lemma *induced-automorphism-morphism-order2: sos = id*
<proof>

lemmas *indaut-order2 = induced-automorphism-morphism-order2*

lemmas *induced-automorphism-bij =*
o-bij[OF
induced-automorphism-morphism-order2
induced-automorphism-morphism-order2
]

lemma *induced-automorphism-surj-on-vertexset: s'(\bigcup X) = \bigcup X*
<proof>

lemma *induced-automorphism-bij-betw-vertexset: bij-betw s (\bigcup X) (\bigcup X)*
<proof>

lemma *induced-automorphism-surj-on-simplices: s\ X = X*
<proof>

lemma *induced-automorphism-automorphism:*
ChamberComplexAutomorphism X s
<proof>

lemmas *indaut-aut = induced-automorphism-automorphism*

lemma *induced-automorphism-unique-automorphism':*
assumes *ChamberComplexAutomorphism X s s \neq id fixespointwise s (C0 \cap D0)*
shows *fun-eq-on s s C0*
<proof>

lemma *induced-automorphism-unique-automorphism:*

[[ChamberComplexAutomorphism X s; s≠id; fixespointwise s (C0∩D0)]]
 ⇒ s = s
 ⟨proof⟩

lemmas *indaut-uniq-aut =*
induced-automorphism-unique-automorphism

lemma *induced-automorphism-unique:*
OpposedThinChamberComplexFoldings X f' g' C0 ⇒ g'C0 = g'C0 ⇒
OpposedThinChamberComplexFoldings.induced-automorphism X f' g' = s
 ⟨proof⟩

lemma *induced-automorphism-sym:*
OpposedThinChamberComplexFoldings.induced-automorphism X g f = s
 ⟨proof⟩

lemma *induced-automorphism-respects-labels:*
assumes *label-wrt B φ v∈(∪ X)*
shows *φ (s v) = φ v*
 ⟨proof⟩

lemmas *indaut-resplabels =*
induced-automorphism-respects-labels

end

4.6.4 Walls

A pair of opposed foldings of a thin chamber complex defines a decomposition of the chamber system into the two disjoint chamber system images. Call such a decomposition a wall, as we image that disjointness erects a wall between the two half chamber systems. By considering the collection of all possible opposed folding pairs, and their associated walls, we can obtain information about minimality of galleries by considering the walls they cross.

context *ThinChamberComplex*
begin

definition *foldpairs :: (('a⇒'a) × ('a⇒'a)) set*
where *foldpairs ≡ {(f,g). ∃ C. OpposedThinChamberComplexFoldings X f g C}*

abbreviation *walls ≡ ∪ (f,g)∈foldpairs. {{f⊢C,g⊢C}}*

abbreviation *the-wall-betw C D ≡*
THE-default {} (λH. H∈walls ∧ separated-by H C D)

definition *walls-betw :: 'a set ⇒ 'a set ⇒ 'a set set set set*
where *walls-betw C D ≡ {H∈walls. separated-by H C D}*

fun *wall-crossings* :: 'a set list \Rightarrow 'a set set set list
where *wall-crossings* [] = []
| *wall-crossings* [C] = []
| *wall-crossings* (B#C#Cs) = *the-wall-betw* B C # *wall-crossings* (C#Cs)

lemma *foldpairs-sym*: (f,g) \in *foldpairs* \implies (g,f) \in *foldpairs*
<proof>

lemma *not-self-separated-by-wall*: H \in *walls* \implies \neg *separated-by* H C C
<proof>

lemma *the-wall-betw-nempty*:
assumes *the-wall-betw* C D \neq {}
shows *the-wall-betw* C D \in *walls* *separated-by* (*the-wall-betw* C D) C D
<proof>

lemma *the-wall-betw-self-empty*: *the-wall-betw* C C = {}
<proof>

lemma *length-wall-crossings*: *length* (*wall-crossings* Cs) = *length* Cs - 1
<proof>

lemma *wall-crossings-snoc*:
wall-crossings (Cs@[D,E]) = *wall-crossings* (Cs@[D]) @ [*the-wall-betw* D E]
<proof>

lemma *wall-crossings-are-walls*:
H \in *set* (*wall-crossings* Cs) \implies H \neq { } \implies H \in *walls*
<proof>

lemma *in-set-wall-crossings-decomp*:
H \in *set* (*wall-crossings* Cs) \implies
 \exists As A B Bs. Cs = As@[A,B]@Bs \wedge H = *the-wall-betw* A B
<proof>

end

context *OpposedThinChamberComplexFoldings*
begin

lemma *foldpair*: (f,g) \in *foldpairs*
<proof>

lemma *separated-by-this-wall-fg*:
separated-by {f \vdash C, g \vdash C} C D \implies C \in f \vdash C \implies D \in g \vdash C
<proof>

lemmas *separated-by-this-wall-gf* =
OpposedThinChamberComplexFoldings.separated-by-this-wall-fg[

OF *fg-symmetric*
]

lemma *induced-automorphism-this-wall-vertex*:
assumes $C \in f \vdash \mathcal{C}$ $D \in g \vdash \mathcal{C}$ $v \in C \cap D$
shows $s \ v = v$
<proof>

lemmas *indaut-wallvertex* =
induced-automorphism-this-wall-vertex

lemma *unique-wall*:
assumes opp' : *OpposedThinChamberComplexFoldings* $X \ f' \ g' \ C'$
and *chambers*: $A \in f \vdash \mathcal{C}$ $A \in f' \vdash \mathcal{C}$ $B \in g \vdash \mathcal{C}$ $B \in g' \vdash \mathcal{C}$ $A \sim B$
shows $\{f \vdash \mathcal{C}, g \vdash \mathcal{C}\} = \{f' \vdash \mathcal{C}, g' \vdash \mathcal{C}\}$
<proof>

end

context *ThinChamberComplex*
begin

lemma *separated-by-wall-ex-foldpair*:
assumes $H \in walls$ *separated-by* $H \ C \ D$
shows $\exists (f, g) \in foldpairs. H = \{f \vdash \mathcal{C}, g \vdash \mathcal{C}\} \wedge C \in f \vdash \mathcal{C} \wedge D \in g \vdash \mathcal{C}$
<proof>

lemma *not-separated-by-wall-ex-foldpair*:
assumes *chambers*: *chamber* C *chamber* D
and *wall* : $H \in walls \neg separated-by \ H \ C \ D$
shows $\exists (f, g) \in foldpairs. H = \{f \vdash \mathcal{C}, g \vdash \mathcal{C}\} \wedge C \in f \vdash \mathcal{C} \wedge D \in f \vdash \mathcal{C}$
<proof>

lemma *adj-wall-imp-ex1-wall*:
assumes adj : $C \sim D$
and *wall*: $H \in walls$ *separated-by* $H \ C \ D$
shows $\exists ! H \in walls. separated-by \ H \ C \ D$
<proof>

end

context *OpposedThinChamberComplexFoldings*
begin

lemma *this-wall-betwI*:
assumes $C \in f \vdash \mathcal{C}$ $D \in g \vdash \mathcal{C}$ $C \sim D$
shows *the-wall-betw* $C \ D = \{f \vdash \mathcal{C}, g \vdash \mathcal{C}\}$
<proof>

lemma *this-wall-betw-basechambers:*
the-wall-betw C0 D0 = {f⊢C,g⊢C}
 ⟨proof⟩

lemma *this-wall-in-crossingsI-fg:*
defines $H: H \equiv \{f\vdash C, g\vdash C\}$
assumes $D: D \in g\vdash C$
shows $C \in f\vdash C \implies \text{gallery } (C \# Cs@[D]) \implies H \in \text{set } (\text{wall-crossings } (C \# Cs@[D]))$
 ⟨proof⟩

end

lemma (in *ThinChamberComplex*) *walls-betw-subset-wall-crossings:*
assumes $\text{gallery } (C \# Cs@[D])$
shows $\text{walls-betw } C D \subseteq \text{set } (\text{wall-crossings } (C \# Cs@[D]))$
 ⟨proof⟩

context *OpposedThinChamberComplexFoldings*
begin

lemma *same-side-this-wall-wall-crossings-not-distinct-f:*
 $\text{gallery } (C \# Cs@[D]) \implies C \in f\vdash C \implies D \in f\vdash C \implies$
 $\{f\vdash C, g\vdash C\} \in \text{set } (\text{wall-crossings } (C \# Cs@[D])) \implies$
 $\neg \text{distinct } (\text{wall-crossings } (C \# Cs@[D]))$
 ⟨proof⟩

lemmas *sside-wcrossings-ndistinct-f =*
same-side-this-wall-wall-crossings-not-distinct-f

lemma *separated-by-this-wall-chain3-fg:*
assumes $B \in f\vdash C$ *chamber C chamber D*
 $\text{separated-by } \{f\vdash C, g\vdash C\} B C \text{ separated-by } \{f\vdash C, g\vdash C\} C D$
shows $C \in g\vdash C D \in f\vdash C$
 ⟨proof⟩

lemmas *sepwall-chain3-fg =*
separated-by-this-wall-chain3-fg

end

context *ThinChamberComplex*
begin

lemma *wall-crossings-min-gallery-betwI:*
assumes $\text{gallery } (C \# Cs@[D])$
 $\text{distinct } (\text{wall-crossings } (C \# Cs@[D]))$
 $\forall H \in \text{set } (\text{wall-crossings } (C \# Cs@[D])). \text{ separated-by } H C D$
shows $\text{min-gallery } (C \# Cs@[D])$
 ⟨proof⟩

lemma *ex-nonseparating-wall-imp-wall-crossings-not-distinct*:

assumes $gal : gallery (C \# Cs @ [D])$
and $wall : H \in set (wall-crossings (C \# Cs @ [D])) \ H \neq \{\}$
 $\neg separated-by \ H \ C \ D$
shows $\neg distinct (wall-crossings (C \# Cs @ [D]))$
<proof>

lemma *not-min-gallery-double-crosses-wall*:

assumes $gallery \ Cs \ \neg \ min-gallery \ Cs \ \{\}$ $\notin set (wall-crossings \ Cs)$
shows $\neg distinct (wall-crossings \ Cs)$
<proof>

lemma *not-distinct-crossings-split-gallery*:

$\llbracket gallery \ Cs; \ \{\} \notin set (wall-crossings \ Cs); \ \neg \ distinct (wall-crossings \ Cs) \rrbracket \implies$
 $\exists f \ g \ As \ A \ B \ Bs \ E \ F \ Fs.$
 $(f, g) \in foldpairs \wedge A \in f \vdash C \wedge B \in g \vdash C \wedge E \in g \vdash C \wedge F \in f \vdash C \wedge$
 $(Cs = As @ [A, B, F] @ Fs \vee Cs = As @ [A, B] @ Bs @ [E, F] @ Fs)$
<proof>

lemma *not-min-gallery-double-split*:

$\llbracket gallery \ Cs; \ \neg \ min-gallery \ Cs; \ \{\} \notin set (wall-crossings \ Cs) \rrbracket \implies$
 $\exists f \ g \ As \ A \ B \ Bs \ E \ F \ Fs.$
 $(f, g) \in foldpairs \wedge A \in f \vdash C \wedge B \in g \vdash C \wedge E \in g \vdash C \wedge F \in f \vdash C \wedge$
 $(Cs = As @ [A, B, F] @ Fs \vee Cs = As @ [A, B] @ Bs @ [E, F] @ Fs)$
<proof>

end

4.7 Thin chamber complexes with many foldings

Here we begin to examine thin chamber complexes in which every pair of adjacent chambers affords a pair of opposed foldings of the complex. This condition will ultimately be shown to be sufficient to ensure that a thin chamber complex is isomorphic to some Coxeter complex.

4.7.1 Locale definition and basic facts

locale *ThinChamberComplexManyFoldings* = *ThinChamberComplex* X

for $X :: 'a \ set \ set$

+ **fixes** $C0 :: 'a \ set$

assumes *fundchamber*: *chamber* $C0$

and *ex-walls* :

$\llbracket chamber \ C; \ chamber \ D; \ C \sim D; \ C \neq D \rrbracket \implies$

$\exists f \ g. \ OpposedThinChamberComplexFoldings \ X \ f \ g \ C \wedge D = g' C$

lemma (in *ThinChamberComplex*) *ThinChamberComplexManyFoldingsI*:

assumes *chamber* $C0$

and $\bigwedge C \ D. \llbracket chamber \ C; \ chamber \ D; \ C \sim D; \ C \neq D \rrbracket \implies$

$\exists f.g. \text{OpposedThinChamberComplexFoldings } X f g C \wedge D=g'C$
shows $\text{ThinChamberComplexManyFoldings } X C0$
 $\langle \text{proof} \rangle$

lemma (in $\text{ThinChamberComplexManyFoldings}$) *wall-crossings-subset-walls-betw*:
assumes $\text{min-gallery } (C\#Cs@[D])$
shows $\text{set } (\text{wall-crossings } (C\#Cs@[D])) \subseteq \text{walls-betw } C D$
 $\langle \text{proof} \rangle$

4.7.2 The group of automorphisms

Recall that a pair of opposed foldings of a thin chamber complex can be stitched together to form an automorphism of the complex. Choosing an arbitrary chamber in the complex to act as a sort of centre of the complex (referred to as the fundamental chamber), we consider the group (under composition) generated by the automorphisms afforded by the chambers adjacent to the fundamental chamber via the pairs of opposed foldings that we have assumed to exist.

context $\text{ThinChamberComplexManyFoldings}$
begin

definition $\text{fundfoldpairs} :: ((a \Rightarrow' a) \times (a \Rightarrow' a)) \text{ set}$
where $\text{fundfoldpairs} \equiv \{(f,g). \text{OpposedThinChamberComplexFoldings } X f g C0\}$

abbreviation $\text{fundadjset} \equiv \text{adjacentset } C0 - \{C0\}$

abbreviation $\text{induced-automorph} :: (a \Rightarrow' a) \Rightarrow (a \Rightarrow' a) \Rightarrow (a \Rightarrow' a)$
where $\text{induced-automorph } f g \equiv$
 $\text{OpposedThinChamberComplexFoldings.induced-automorphism } X f g$

abbreviation $\text{Abs-induced-automorph} :: (a \Rightarrow' a) \Rightarrow (a \Rightarrow' a) \Rightarrow 'a \text{ permutation}$
where $\text{Abs-induced-automorph } f g \equiv \text{Abs-permutation } (\text{induced-automorph } f g)$

abbreviation $S \equiv \bigcup (f,g) \in \text{fundfoldpairs}. \{\text{Abs-induced-automorph } f g\}$

abbreviation $W \equiv \langle S \rangle$

lemma $\text{fundfoldpairs-induced-autormorph-bij}$:
 $(f,g) \in \text{fundfoldpairs} \implies \text{bij } (\text{induced-automorph } f g)$
 $\langle \text{proof} \rangle$

lemmas $\text{permutation-conv-induced-automorph} =$
 $\text{Abs-permutation-inverse}[OF \text{CollectI}, OF \text{fundfoldpairs-induced-autormorph-bij}]$

lemma $\text{fundfoldpairs-induced-autormorph-order2}$:
 $(f,g) \in \text{fundfoldpairs} \implies \text{induced-automorph } f g \circ \text{induced-automorph } f g = \text{id}$
 $\langle \text{proof} \rangle$

lemma $\text{fundfoldpairs-induced-autormorph-ntrivial}$:

$(f,g) \in \text{fundfoldpairs} \implies \text{induced-automorph } f g \neq \text{id}$
 ⟨proof⟩

lemma *fundfoldpairs-fundchamber-image:*

$(f,g) \in \text{fundfoldpairs} \implies \text{Abs-induced-automorph } f g \text{ '}\rightarrow C0 = g \text{ '}C0$
 ⟨proof⟩

lemma *fundfoldpair-fundchamber-in-half-chamber-system-f:*

$(f,g) \in \text{fundfoldpairs} \implies C0 \in f \vdash C$
 ⟨proof⟩

lemma *fundfoldpair-unique-half-chamber-system-f:*

assumes $(f,g) \in \text{fundfoldpairs} \ (f',g') \in \text{fundfoldpairs}$
 $\text{Abs-induced-automorph } f' g' = \text{Abs-induced-automorph } f g$
shows $f \vdash C = f' \vdash C$
 ⟨proof⟩

lemma *fundfoldpair-unique-half-chamber-systems-chamber-ng-f:*

assumes $(f,g) \in \text{fundfoldpairs} \ (f',g') \in \text{fundfoldpairs}$
 $\text{Abs-induced-automorph } f' g' = \text{Abs-induced-automorph } f g$
chamber $C \ C \notin g \vdash C$
shows $C \in f \vdash C$
 ⟨proof⟩

lemma *the-wall-betw-adj-fundchamber:*

$(f,g) \in \text{fundfoldpairs} \implies$
 $\text{the-wall-betw } C0 \ (\text{Abs-induced-automorph } f g \text{ '}\rightarrow C0) = \{f \vdash C, g \vdash C\}$
 ⟨proof⟩

lemma *zero-notin-S: 0 ∉ S*

⟨proof⟩

lemma *S-order2-add: s ∈ S ⟹ s + s = 0*

⟨proof⟩

lemma *S-add-order2:*

assumes $s \in S$
shows $\text{add-order } s = 2$
 ⟨proof⟩

lemmas $S\text{-uminus} = \text{minus-unique}[OF \ S\text{-order2-add}]$

lemma *S-sym: uminus ' S ⊆ S*

⟨proof⟩

lemmas $\text{sum-list-S-in-W} = \text{sum-list-lists-in-genby-sym}[OF \ S\text{-sym}]$

lemmas $\text{W-conv-sum-lists} = \text{genby-sym-eq-sum-lists}[OF \ S\text{-sym}]$

lemma *S-endomorphism:*

$s \in S \implies \text{ChamberComplexEndomorphism } X \text{ (permutation } s)$
(proof)

lemma *S-list-endomorphism:*

$ss \in \text{lists } S \implies \text{ChamberComplexEndomorphism } X \text{ (permutation (sum-list ss))}$
(proof)

lemma *W-endomorphism:*

$w \in W \implies \text{ChamberComplexEndomorphism } X \text{ (permutation } w)$
(proof)

lemma *S-automorphism:*

$s \in S \implies \text{ChamberComplexAutomorphism } X \text{ (permutation } s)$
(proof)

lemma *S-list-automorphism:*

$ss \in \text{lists } S \implies \text{ChamberComplexAutomorphism } X \text{ (permutation (sum-list ss))}$
(proof)

lemma *W-automorphism:*

$w \in W \implies \text{ChamberComplexAutomorphism } X \text{ (permutation } w)$
(proof)

lemma *S-respects-labels:* $\llbracket \text{label-wrt } B \ \varphi; s \in S; v \in (\bigcup X) \rrbracket \implies \varphi (s \rightarrow v) = \varphi v$
(proof)

lemma *S-list-respects-labels:*

$\llbracket \text{label-wrt } B \ \varphi; ss \in \text{lists } S; v \in (\bigcup X) \rrbracket \implies \varphi (\text{sum-list } ss \rightarrow v) = \varphi v$
(proof)

lemma *W-respects-labels:*

$\llbracket \text{label-wrt } B \ \varphi; w \in W; v \in (\bigcup X) \rrbracket \implies \varphi (w \rightarrow v) = \varphi v$
(proof)

end

4.7.3 Action of the group of automorphisms on the chamber system

Now we examine the action of the group W on the chamber system. In particular, we show that the action is transitive.

context *ThinChamberComplexManyFoldings*
begin

lemma *fundchamber-S-chamber:* $s \in S \implies \text{chamber } (s' \rightarrow C0)$
(proof)

lemma *fundchamber-W-image-chamber:*

$w \in W \implies \text{chamber } (w' \rightarrow C0)$

<proof>

lemma *fundchamber-S-adjacent*: $s \in S \implies C0 \sim (s' \rightarrow C0)$
<proof>

lemma *fundchamber-WS-image-adjacent*:
 $w \in W \implies s \in S \implies (w' \rightarrow C0) \sim ((w+s)' \rightarrow C0)$
<proof>

lemma *fundchamber-S-image-neq-fundchamber*: $s \in S \implies s' \rightarrow C0 \neq C0$
<proof>

lemma *fundchamber-next-WS-image-neq*:
assumes $s \in S$
shows $(w+s)' \rightarrow C0 \neq w' \rightarrow C0$
<proof>

lemma *fundchamber-S-fundadjset*: $s \in S \implies s' \rightarrow C0 \in \text{fundadjset}$
<proof>

lemma *fundadjset-eq-S-image*: $D \in \text{fundadjset} \implies \exists s \in S. D = s' \rightarrow C0$
<proof>

lemma *S-fixespointwise-fundchamber-image-int*:
assumes $s \in S$
shows *fixespointwise* $((\rightarrow) s) (C0 \cap s' \rightarrow C0)$
<proof>

lemma *S-fixes-fundchamber-image-int*:
 $s \in S \implies s' \rightarrow (C0 \cap s' \rightarrow C0) = C0 \cap s' \rightarrow C0$
<proof>

lemma *fundfacets*:
assumes $s \in S$
shows $C0 \cap s' \rightarrow C0 \triangleleft C0$ $C0 \cap s' \rightarrow C0 \triangleleft s' \rightarrow C0$
<proof>

lemma *fundadjset-ex1-eq-S-image*:
assumes $D \in \text{fundadjset}$
shows $\exists ! s \in S. D = s' \rightarrow C0$
<proof>

lemma *fundchamber-S-image-inj-on*: *inj-on* $(\lambda s. s' \rightarrow C0) S$
<proof>

lemma *S-list-image-gallery*:
 $ss \in \text{lists } S \implies \text{gallery } (\text{map } (\lambda w. w' \rightarrow C0)) (\text{sums } ss)$
<proof>

lemma *pgallery-last-eq-W-image*:
 $pgallery (C0\#Cs@[C]) \implies \exists w \in W. C = w' \rightarrow C0$
 <proof>

lemma *chamber-eq-W-image*:
assumes *chamber C*
shows $\exists w \in W. C = w' \rightarrow C0$
 <proof>

lemma *S-list-image-crosses-walls*:
 $ss \in lists\ S \implies \{\} \notin set\ (wall\ crossings\ (map\ (\lambda w. w' \rightarrow C0)\ (sums\ ss)))$
 <proof>

end

4.7.4 A labelling by the vertices of the fundamental chamber

Here we show that by repeatedly applying the composition of all the elements in the collection S of fundamental automorphisms, we can retract the entire chamber complex onto the fundamental chamber. This retraction provides a means of labelling the chamber complex, using the vertices of the fundamental chamber as labels.

context *ThinChamberComplexManyFoldings*
begin

definition *Spair* :: 'a permutation \Rightarrow ('a \Rightarrow 'a) \times ('a \Rightarrow 'a)
where *Spair s* \equiv
 $SOME\ fg. fg \in fundfoldpairs \wedge s = case\ prod\ Abs\ induced\ automorph\ fg$

lemma *Spair-fundfoldpair*: $s \in S \implies Spair\ s \in fundfoldpairs$
 <proof>

lemma *Spair-induced-automorph*:
 $s \in S \implies s = case\ prod\ Abs\ induced\ automorph\ (Spair\ s)$
 <proof>

lemma *S-list-pgallery-decomp1*:
assumes *ss: set ss = S and gal: Cs \neq [] pgallery (C0#Cs)*
shows $\exists s \in set\ ss. \exists C \in set\ Cs. \forall (f,g) \in fundfoldpairs.$
 $s = Abs\ induced\ automorph\ f\ g \longrightarrow C \in g\vdash C$
 <proof>

lemma *S-list-pgallery-decomp2*:
assumes *set ss = S Cs \neq [] pgallery (C0#Cs)*
shows
 $\exists rs\ s\ ts. ss = rs@s\#ts \wedge$
 $(\exists C \in set\ Cs. \forall (f,g) \in fundfoldpairs.$
 $s = Abs\ induced\ automorph\ f\ g \longrightarrow C \in g\vdash C) \wedge$

$(\forall r \in \text{set } rs. \forall C \in \text{set } Cs. \forall (f,g) \in \text{fundfoldpairs}.$
 $r = \text{Abs-induced-automorph } f g \longrightarrow C \in \text{f}\vdash\text{C}$
 <proof>

lemma *S-list-pgallery-decomp3*:

assumes *set ss = S Cs#[] pgallery (C0#Cs)*

shows

$\exists rs s ts As B Bs. ss = rs@s\#ts \wedge Cs = As@B\#Bs \wedge$
 $(\forall (f,g) \in \text{fundfoldpairs}. s = \text{Abs-induced-automorph } f g \longrightarrow B \in \text{g}\vdash\text{C}) \wedge$
 $(\forall A \in \text{set } As. \forall (f,g) \in \text{fundfoldpairs}.$
 $s = \text{Abs-induced-automorph } f g \longrightarrow A \in \text{f}\vdash\text{C}) \wedge$
 $(\forall r \in \text{set } rs. \forall C \in \text{set } Cs. \forall (f,g) \in \text{fundfoldpairs}.$
 $r = \text{Abs-induced-automorph } f g \longrightarrow C \in \text{f}\vdash\text{C})$

<proof>

lemma *fundfold-trivial-fC*:

$r \in S \implies \forall (f,g) \in \text{fundfoldpairs}. r = \text{Abs-induced-automorph } f g \longrightarrow C \in \text{f}\vdash\text{C} \implies$
 $\text{fst } (\text{Spair } r) \text{ ' } C = C$

<proof>

lemma *fundfold-comp-trivial-fC*:

$\text{set } rs \subseteq S \implies$
 $\forall r \in \text{set } rs. \forall (f,g) \in \text{fundfoldpairs}.$
 $r = \text{Abs-induced-automorph } f g \longrightarrow C \in \text{f}\vdash\text{C} \implies$
 $\text{fold fst } (\text{map Spair } rs) \text{ ' } C = C$

<proof>

lemma *fundfold-trivial-fC-list*:

$r \in S \implies$
 $\forall C \in \text{set } Cs. \forall (f,g) \in \text{fundfoldpairs}.$
 $r = \text{Abs-induced-automorph } f g \longrightarrow C \in \text{f}\vdash\text{C} \implies$
 $\text{fst } (\text{Spair } r) \models Cs = Cs$

<proof>

lemma *fundfold-comp-trivial-fC-list*:

$\text{set } rs \subseteq S \implies$
 $\forall r \in \text{set } rs. \forall C \in \text{set } Cs. \forall (f,g) \in \text{fundfoldpairs}.$
 $r = \text{Abs-induced-automorph } f g \longrightarrow C \in \text{f}\vdash\text{C} \implies$
 $\text{fold fst } (\text{map Spair } rs) \models Cs = Cs$

<proof>

lemma *fundfold-gallery-map*:

$s \in S \implies \text{gallery } Cs \implies \text{gallery } (\text{fst } (\text{Spair } s) \models Cs)$
 <proof>

lemma *fundfold-comp-gallery-map*:

assumes *pregal: gallery Cs*
shows $\text{set } ss \subseteq S \implies \text{gallery } (\text{fold fst } (\text{map Spair } ss) \models Cs)$

<proof>

lemma *fundfold-comp-pgallery-ex-funpow*:
assumes *ss*: set *ss* = *S*
shows $pgallery (C0 \# Cs@[C]) \implies$
 $\exists n. (fold\ fst\ (map\ Spair\ ss)\ \sim\ n) \cdot C = C0$
 $\langle proof \rangle$

lemma *fundfold-comp-chamber-ex-funpow*:
assumes *ss*: set *ss* = *S* **and** *C*: chamber *C*
shows $\exists n. (fold\ fst\ (map\ Spair\ ss)\ \sim\ n) \cdot C = C0$
 $\langle proof \rangle$

lemma *fundfold-comp-fixespointwise-C0*:
assumes set *ss* $\subseteq S$
shows *fixespointwise* (fold fst (map Spair ss)) *C0*
 $\langle proof \rangle$

lemma *fundfold-comp-endomorphism*:
assumes set *ss* $\subseteq S$
shows *ChamberComplexEndomorphism* *X* (fold fst (map Spair ss))
 $\langle proof \rangle$

lemma *finite-S*: finite *S*
 $\langle proof \rangle$

lemma *ex-label-retraction*: $\exists \varphi. label\ wrt\ C0\ \varphi \wedge fixespointwise\ \varphi\ C0$
 $\langle proof \rangle$

lemma *ex-label-map*: $\exists \varphi. label\ wrt\ C0\ \varphi$
 $\langle proof \rangle$

end

4.7.5 More on the action of the group of automorphisms on chambers

Recall that we have already verified that W acts transitively on the chamber system. We now use the labelling of the chamber complex examined in the previous section to show that this action is simply transitive.

context *ThinChamberComplexManyFoldings*
begin

lemma *fundchamber-W-image-ker*:
assumes $w \in W\ w' \rightarrow C0 = C0$
shows $w = 0$
 $\langle proof \rangle$

lemma *fundchamber-W-image-inj-on*:

inj-on ($\lambda w. w' \rightarrow C0$) W
 ⟨*proof*⟩

end

4.7.6 A bijection between the fundamental chamber and the set of generating automorphisms

Removing a single vertex from the fundamental chamber determines a facet, a facet in the fundamental chamber determines an adjacent chamber (since our complex is thin), and a chamber adjacent to the fundamental chamber determines an automorphism (via some pair of opposed foldings) in our generating set S . Here we show that this correspondence is bijective.

context *ThinChamberComplexManyFoldings*
begin

definition *fundantivertex* :: 'a permutation \Rightarrow 'a
where *fundantivertex* $s \equiv$ (*THE* $v. v \in C0 - s' \rightarrow C0$)

abbreviation *fundantipermutation* \equiv *the-inv-into* S *fundantivertex*

lemma *fundantivertex*: $s \in S \implies$ *fundantivertex* $s \in C0 - s' \rightarrow C0$
 ⟨*proof*⟩

lemma *fundantivertex-fundchamber-decomp*:
 $s \in S \implies C0 = \text{insert } (\text{fundantivertex } s) (C0 \cap s' \rightarrow C0)$
 ⟨*proof*⟩

lemma *fundantivertex-unstable*:
 $s \in S \implies s \rightarrow$ *fundantivertex* $s \neq$ *fundantivertex* s
 ⟨*proof*⟩

lemma *fundantivertex-inj-on*: *inj-on* *fundantivertex* S
 ⟨*proof*⟩

lemma *fundantivertex-surj-on*: *fundantivertex* ' $S = C0$
 ⟨*proof*⟩

lemma *fundantivertex-bij-betw*: *bij-betw* *fundantivertex* S $C0$
 ⟨*proof*⟩

lemma *card-S-fundchamber*: *card* $S =$ *card* $C0$
 ⟨*proof*⟩

lemma *card-S-chamber*:
chamber $C \implies$ *card* $C =$ *card* S
 ⟨*proof*⟩

lemma *fundantipermutation1*:
 $v \in C0 \implies \text{fundantipermutation } v \in S$
 ⟨proof⟩

end

4.8 Thick chamber complexes

A thick chamber complex is one in which every facet is a facet of at least three chambers.

locale *ThickChamberComplex* = *ChamberComplex* *X*
for *X* :: 'a set set
 + **assumes** *thick*:
 $\text{chamber } C \implies z \triangleleft C \implies$
 $\exists D E. D \in X - \{C\} \wedge z \triangleleft D \wedge E \in X - \{C, D\} \wedge z \triangleleft E$

begin

definition *some-third-chamber* :: 'a set \implies 'a set \implies 'a set \implies 'a set
where *some-third-chamber* *C D z* \equiv *SOME* *E. E* $\in X - \{C, D\} \wedge z \triangleleft E$

lemma *facet-ex-third-chamber*: *chamber* *C* $\implies z \triangleleft C \implies \exists E \in X - \{C, D\}. z \triangleleft E$
 ⟨proof⟩

lemma *some-third-chamberD-facet*:
 $\text{chamber } C \implies z \triangleleft C \implies z \triangleleft \text{some-third-chamber } C D z$
 ⟨proof⟩

lemma *some-third-chamberD-simplex*:
 $\text{chamber } C \implies z \triangleleft C \implies \text{some-third-chamber } C D z \in X$
 ⟨proof⟩

lemma *some-third-chamberD-adj*:
 $\text{chamber } C \implies z \triangleleft C \implies C \sim \text{some-third-chamber } C D z$
 ⟨proof⟩

lemma *chamber-some-third-chamber*:
 $\text{chamber } C \implies z \triangleleft C \implies \text{chamber } (\text{some-third-chamber } C D z)$
 ⟨proof⟩

lemma *some-third-chamberD-ne*:
assumes *chamber* *C* $z \triangleleft C$
shows *some-third-chamber* *C D z* $\neq C$ *some-third-chamber* *C D z* $\neq D$
 ⟨proof⟩

end

end

5 Coxeter systems and complexes

A Coxeter system is a group that affords a presentation, where each generator is of order two, and each relator is an alternating word of even length in two generators.

```
theory Coxeter
imports Chamber
```

```
begin
```

5.1 Coxeter-like systems

First we work in a group generated by elements of order two.

5.1.1 Locale definition and basic facts

```
locale PreCoxeterSystem =
  fixes S :: 'w::group-add set
  assumes genset-order2:  $s \in S \implies \text{add-order } s = 2$ 
begin
```

```
abbreviation W  $\equiv \langle S \rangle$ 
```

```
abbreviation S-length  $\equiv \text{word-length } S$ 
```

```
abbreviation S-reduced-for  $\equiv \text{reduced-word-for } S$ 
```

```
abbreviation S-reduced  $\equiv \text{reduced-word } S$ 
```

```
abbreviation relfun  $\equiv \lambda s t. \text{add-order } (s+t)$ 
```

```
lemma no-zero-genset:  $0 \notin S$ 
<proof>
```

```
lemma genset-order2-add:  $s \in S \implies s + s = 0$ 
<proof>
```

```
lemmas genset-uminus = minus-unique[OF genset-order2-add]
```

```
lemma relfun-S:  $s \in S \implies \text{relfun } s s = 1$ 
<proof>
```

```
lemma relfun-eq1:  $\llbracket s \in S; \text{relfun } s t = 1 \rrbracket \implies t = s$ 
<proof>
```

```
lemma S-relator-list:  $s \in S \implies \text{pair-relator-list } s s = [s,s]$ 
<proof>
```

```
lemma S-sym:  $T \subseteq S \implies \text{uminus } ' T \subseteq T$ 
<proof>
```

```
lemmas special-subgroup-eq-sum-list =
```

```

    genby-sym-eq-sum-lists[OF S-sym]
lemmas genby-S-reduced-word-for-arg-min =
    reduced-word-for-genby-sym-arg-min[OF S-sym]
lemmas in-genby-S-reduced-letter-set =
    in-genby-sym-imp-in-reduced-letter-set[OF S-sym]

end

```

5.1.2 Special cosets

From a Coxeter system we will eventually construct an associated chamber complex. To do so, we will consider the collection of special cosets: left cosets of subgroups generated by subsets of the generating set S . This collection forms a poset under the supset relation that, under a certain extra assumption, can be used to form a simplicial complex whose poset of simplices is isomorphic to this poset of special cosets. In the literature, groups generated by subsets of S are often referred to as parabolic subgroups of W , and their cosets as parabolic cosets, but following Garrett [2] we have opted for the names special subgroups and special cosets.

```

context PreCoxeterSystem
begin

```

```

definition special-cosets :: 'w set set
  where special-cosets  $\equiv$  ( $\bigcup T \in Pow S. (\bigcup w \in W. \{ w + o \langle T \rangle \})$ )
abbreviation  $\mathcal{P} \equiv$  special-cosets

```

```

lemma special-cosetsI:  $T \in Pow S \implies w \in W \implies w + o \langle T \rangle \in \mathcal{P}$ 
   $\langle$ proof $\rangle$ 

```

```

lemma special-coset-singleton:  $w \in W \implies \{w\} \in \mathcal{P}$ 
   $\langle$ proof $\rangle$ 

```

```

lemma special-coset-nempty:  $X \in \mathcal{P} \implies X \neq \{\}$ 
   $\langle$ proof $\rangle$ 

```

```

lemma special-subgroup-special-coset:  $T \in Pow S \implies \langle T \rangle \in \mathcal{P}$ 
   $\langle$ proof $\rangle$ 

```

```

lemma special-cosets-lcoset-closed:  $w \in W \implies X \in \mathcal{P} \implies w + o X \in \mathcal{P}$ 
   $\langle$ proof $\rangle$ 

```

```

lemma special-cosets-lcoset-shift:  $w \in W \implies ((+o) w) ' \mathcal{P} = \mathcal{P}$ 
   $\langle$ proof $\rangle$ 

```

```

lemma special-cosets-has-bottom: supset-has-bottom  $\mathcal{P}$ 
   $\langle$ proof $\rangle$ 

```

```

lemma special-cosets-bottom: supset-bottom  $\mathcal{P} = W$ 

```


<proof>

end

5.1.3 Transfer from the free group over generators

We form a set of relators and show that it and S form a *GroupWithGeneratorsRelators*. The associated quotient group G maps surjectively onto W . In the *CoxeterSystem* locale below, this correspondence will be assumed to be injective as well.

context *PreCoxeterSystem*
begin

abbreviation $R :: 'w \text{ list set}$ **where** $R \equiv (\bigcup s \in S. \bigcup t \in S. \{\text{pair-relator-list } s \ t\})$

abbreviation $P \equiv \text{map } (\text{charpair } S) \text{ ' } R$

abbreviation $P' \equiv \text{GroupWithGeneratorsRelators.P' } S \ R$

abbreviation $Q \equiv \text{GroupWithGeneratorsRelators.Q } S \ R$

abbreviation $G \equiv \text{GroupWithGeneratorsRelators.G } S \ R$

abbreviation *relator-freeword* \equiv

GroupWithGeneratorsRelators.relator-freeword S

abbreviation *pair-relator-freeword* $:: 'w \Rightarrow 'w \Rightarrow 'w \text{ freeword}$

where *pair-relator-freeword* $s \ t \equiv \text{Abs-freelist } (\text{pair-relator-list } s \ t)$

abbreviation *freeliftid* $\equiv \text{freeword-funlift id}$

abbreviation *induced-id* $:: 'w \text{ freeword set permutation} \Rightarrow 'w$

where *induced-id* $\equiv \text{GroupWithGeneratorsRelators.induced-id } S \ R$

lemma *S-relator-freeword*: $s \in S \Longrightarrow \text{pair-relator-freeword } s \ s = s[+]s$

<proof>

lemma *map-charpair-map-pairtrue-R*:

$s \in S \Longrightarrow t \in S \Longrightarrow$

$\text{map } (\text{charpair } S) (\text{pair-relator-list } s \ t) = \text{map } \text{pairtrue } (\text{pair-relator-list } s \ t)$

<proof>

lemma *relator-freeword*:

$s \in S \Longrightarrow t \in S \Longrightarrow$

$\text{pair-relator-freeword } s \ t = \text{relator-freeword } (\text{pair-relator-list } s \ t)$

<proof>

lemma *relator-freewords*: $\text{Abs-freelist ' } R = P'$

<proof>

lemma *GroupWithGeneratorsRelators-S-R*: *GroupWithGeneratorsRelators* $S \ R$

<proof>

lemmas *GroupByPresentation-S-P* =

GroupWithGeneratorsRelators.GroupByPresentation-S-P
OF GroupWithGeneratorsRelators-S-R
]

lemmas *Q-FreeS = GroupByPresentation.Q-FreeS*[*OF GroupByPresentation-S-P*]

lemma *relator-freeword-Q*: $s \in S \implies t \in S \implies \text{pair-relator-freeword } s \ t \in Q$
 ⟨*proof*⟩

lemmas *P'-FreeS =*
GroupWithGeneratorsRelators.P'-FreeS
OF GroupWithGeneratorsRelators-S-R
]

lemmas *GroupByPresentationInducedFun-S-P-id =*
GroupWithGeneratorsRelators.GroupByPresentationInducedFun-S-P-id
OF GroupWithGeneratorsRelators-S-R
]

lemma *rconj-relator-freeword*:
 [$s \in S$; $t \in S$; *proper-signed-list* xs ; *fst* ' *set* $xs \subseteq S$] \implies
rconjby (*Abs-freeword* xs) (*pair-relator-freeword* $s \ t$) $\in Q$
 ⟨*proof*⟩

lemma *lconjby-Abs-freelist-relator-freeword*:
 [$s \in S$; $t \in S$; $xs \in \text{lists } S$] \implies
lconjby (*Abs-freelist* xs) (*pair-relator-freeword* $s \ t$) $\in Q$
 ⟨*proof*⟩

lemma *Abs-freelist-rev-append-alternating-list-in-Q*:
assumes $s \in S \ t \in S$
shows *Abs-freelist* (*rev* (*alternating-list* $n \ s \ t$) @ *alternating-list* $n \ s \ t$) $\in Q$
 ⟨*proof*⟩

lemma *Abs-freeword-freelist-uminus-add-in-Q*:
proper-signed-list $xs \implies \text{fst}$ ' *set* $xs \subseteq S \implies$
 – *Abs-freelistfst* $xs + \text{Abs-freeword } xs \in Q$
 ⟨*proof*⟩

lemma *Q-freelist-freeword'*:
 [*proper-signed-list* xs ; *fst* ' *set* $xs \subseteq S$; *Abs-freelistfst* $xs \in Q$] \implies
Abs-freeword $xs \in Q$
 ⟨*proof*⟩

lemma *Q-freelist-freeword*:
 $c \in \text{FreeGroup } S \implies \text{Abs-freelist} (\text{map } \text{fst} (\text{freeword } c)) \in Q \implies c \in Q$
 ⟨*proof*⟩

Here we show that the lift of the identity map to the free group on S is

really just summation.

lemma *freeliftid-Abs-freeword-conv-sum-list*:
proper-signed-list xs \implies *fst* ' *set xs* $\subseteq S \implies$
freeliftid (Abs-freeword xs) = sum-list (map fst xs)
<proof>

end

5.1.4 Words in generators containing alternating subwords

Besides cancelling subwords equal to relators, the primary algebraic manipulation in seeking to reduce a word in generators in a Coxeter system is to reverse the order of alternating subwords of half the length of the associated relator, in order to create adjacent repeated letters that can be cancelled. Here we detail the mechanics of such manipulations.

context *PreCoxeterSystem*
begin

lemma *sum-list-pair-relator-halflist-flip*:
 $s \in S \implies t \in S \implies$
sum-list (pair-relator-halflist s t) = sum-list (pair-relator-halflist t s)
<proof>

definition *flip-altsublist-adjacent* :: 'w list \Rightarrow 'w list \Rightarrow bool
where *flip-altsublist-adjacent ss ts*
 $\equiv \exists s t \text{ as bs. } ss = \text{as} @ (\text{pair-relator-halflist } s \ t) @ \text{bs} \wedge$
 $ts = \text{as} @ (\text{pair-relator-halflist } t \ s) @ \text{bs}$

abbreviation *flip-altsublist-chain* \equiv *binrelchain flip-altsublist-adjacent*

lemma *flip-altsublist-adjacentI*:
 $ss = \text{as} @ (\text{pair-relator-halflist } s \ t) @ \text{bs} \implies$
 $ts = \text{as} @ (\text{pair-relator-halflist } t \ s) @ \text{bs} \implies$
flip-altsublist-adjacent ss ts
<proof>

lemma *flip-altsublist-adjacent-Cons-grow*:
assumes *flip-altsublist-adjacent ss ts*
shows *flip-altsublist-adjacent (a#ss) (a#ts)*
<proof>

lemma *flip-altsublist-chain-map-Cons-grow*:
flip-altsublist-chain tss \implies *flip-altsublist-chain (map ((#) t) tss)*
<proof>

lemma *flip-altsublist-adjacent-refl*:
 $ss \neq [] \implies ss \in \text{lists } S \implies \text{flip-altsublist-adjacent } ss \ ss$
<proof>

lemma *flip-altsublist-adjacent-sym:*

flip-altsublist-adjacent ss ts \implies flip-altsublist-adjacent ts ss

<proof>

lemma *rev-flip-altsublist-chain:*

flip-altsublist-chain xss \implies flip-altsublist-chain (rev xss)

<proof>

lemma *flip-altsublist-adjacent-set:*

assumes *ss \in lists S flip-altsublist-adjacent ss ts*

shows *set ts = set ss*

<proof>

lemma *flip-altsublist-adjacent-set-ball:*

\forall ss \in lists S. \forall ts. flip-altsublist-adjacent ss ts \longrightarrow set ts = set ss

<proof>

lemma *flip-altsublist-adjacent-lists:*

ss \in lists S \implies flip-altsublist-adjacent ss ts \implies ts \in lists S

<proof>

lemma *flip-altsublist-adjacent-lists-ball:*

\forall ss \in lists S. \forall ts. flip-altsublist-adjacent ss ts \longrightarrow ts \in lists S

<proof>

lemma *flip-altsublist-chain-lists:*

ss \in lists S \implies flip-altsublist-chain (ss#xss@[ts]) \implies ts \in lists S

<proof>

lemmas *flip-altsublist-chain-funcong-Cons-snoc =*

binrelchain-setfuncong-Cons-snoc[OF flip-altsublist-adjacent-lists-ball]

lemmas *flip-altsublist-chain-set =*

flip-altsublist-chain-funcong-Cons-snoc[

OF flip-altsublist-adjacent-set-ball

]

lemma *flip-altsublist-adjacent-length:*

flip-altsublist-adjacent ss ts \implies length ts = length ss

<proof>

lemmas *flip-altsublist-chain-length =*

binrelchain-funcong-Cons-snoc[

of flip-altsublist-adjacent length, OF flip-altsublist-adjacent-length, simplified

]

lemma *flip-altsublist-adjacent-sum-list:*

assumes *ss \in lists S flip-altsublist-adjacent ss ts*

shows *sum-list ts = sum-list ss*

<proof>

lemma *flip-altsublist-adjacent-sum-list-ball:*

$\forall ss \in \text{lists } S. \forall ts. \text{flip-altsublist-adjacent } ss \ ts \longrightarrow \text{sum-list } ts = \text{sum-list } ss$

<proof>

lemma *S-reduced-forI-flip-altsublist-adjacent:*

$S\text{-reduced-for } w \ ss \Longrightarrow \text{flip-altsublist-adjacent } ss \ ts \Longrightarrow S\text{-reduced-for } w \ ts$

<proof>

lemma *flip-altsublist-adjacent-in-Q':*

fixes $as \ bs \ s \ t$

defines $xs: xs \equiv as \ @ \ \text{pair-relator-halflist } s \ t \ @ \ bs$

and $ys: ys \equiv as \ @ \ \text{pair-relator-halflist } t \ s \ @ \ bs$

assumes $Axs: \text{Abs-freelist } xs \in Q$

shows $\text{Abs-freelist } ys \in Q$

<proof>

lemma *flip-altsublist-adjacent-in-Q:*

$\text{Abs-freelist } ss \in Q \Longrightarrow \text{flip-altsublist-adjacent } ss \ ts \Longrightarrow \text{Abs-freelist } ts \in Q$

<proof>

lemma *flip-altsublist-chain-G-in-Q:*

$\llbracket \text{Abs-freelist } ss \in Q; \text{flip-altsublist-chain } (ss \# xss \ @ \ [ts]) \rrbracket \Longrightarrow \text{Abs-freelist } ts \in Q$

<proof>

lemma *alternating-S-no-flip:*

assumes $s \in S \ t \in S \ n > 0 \ n < \text{relfun } s \ t \vee \text{relfun } s \ t = 0$

shows $\text{sum-list } (\text{alternating-list } n \ s \ t) \neq \text{sum-list } (\text{alternating-list } n \ t \ s)$

<proof>

lemma *exchange-alternating-not-in-alternating:*

assumes $n \geq 2 \ n < \text{relfun } s \ t \vee \text{relfun } s \ t = 0$

$S\text{-reduced-for } w \ (\text{alternating-list } n \ s \ t \ @ \ cs)$

$\text{alternating-list } n \ s \ t \ @ \ cs = xs \ @ \ [x] \ @ \ ys \ S\text{-reduced-for } w \ (t \ # \ xs \ @ \ ys)$

shows $\text{length } xs \geq n$

<proof>

end

5.1.5 Preliminary facts on the word problem

The word problem seeks criteria for determining whether two words over the generator set represent the same element in W . Here we establish one direction of the word problem, as well as a preliminary step toward the other direction.

context *PreCoxeterSystem*

begin

lemmas *flip-altsublist-chain-sum-list* =

flip-altsublist-chain-funcong-Cons-snoc[*OF flip-altsublist-adjacent-sum-list-ball*]

— This lemma represents one direction in the word problem: if a word in generators can be transformed into another by a sequence of manipulations, each of which consists of replacing a half-relator subword by its reversal, then the two words sum to the same element of W .

lemma *reduced-word-problem-eq-hd-step*:

assumes *step*: $\bigwedge y\ ss\ ts.$ \llbracket

$S\text{-length } y < S\text{-length } w; y \neq 0; S\text{-reduced-for } y\ ss; S\text{-reduced-for } y\ ts$

$\rrbracket \implies \exists xss. \text{flip-altsublist-chain } (ss \# xss @ [ts])$

and *set-up*: $S\text{-reduced-for } w\ (a\#ss)\ S\text{-reduced-for } w\ (a\#ts)$

shows $\exists xss. \text{flip-altsublist-chain } ((a\#ss) \# xss @ [a\#ts])$

<proof>

end

5.1.6 Preliminary facts related to the deletion condition

The deletion condition states that in a Coxeter system, every non-reduced word in the generating set can be shortened to an equivalent word by deleting some particular pair of letters. This condition is both necessary and sufficient for a group generated by elements of order two to be a Coxeter system. Here we establish some facts related to the deletion condition that are true in any group generated by elements of order two.

context *PreCoxeterSystem*

begin

abbreviation $\mathcal{H} \equiv (\bigcup w \in W. \text{lconjby } w \text{ ' } S)$ — the set of reflections

abbreviation *lift-signed-lconjperm* $\equiv \text{freeword-funlift signed-lconjpermutation}$

lemma *lconjseq-reflections*: $ss \in \text{lists } S \implies \text{set } (\text{lconjseq } ss) \subseteq \mathcal{H}$

<proof>

lemma *deletion'*:

$ss \in \text{lists } S \implies \neg \text{distinct } (\text{lconjseq } ss) \implies$

$\exists a\ b\ as\ bs\ cs. ss = as @ [a] @ bs @ [b] @ cs \wedge$

$\text{sum-list } ss = \text{sum-list } (as @ bs @ cs)$

<proof>

lemma *S-reduced-imp-distinct-lconjseq'*:

assumes $ss \in \text{lists } S \neg \text{distinct } (\text{lconjseq } ss)$

shows $\neg S\text{-reduced } ss$

<proof>

lemma *S-reduced-imp-distinct-lconjseq*: $S\text{-reduced } ss \implies \text{distinct } (\text{lconjseq } ss)$

<proof>

lemma *permutation-lift-signed-lconjperm-eq-signed-list-lconjunction'*:
proper-signed-list $xs \implies \text{fst } \text{' set } xs \subseteq S \implies$
permutation (lift-signed-lconjperm (Abs-freeword xs)) =
signed-list-lconjunction (map fst xs)
<proof>

lemma *permutation-lift-signed-lconjperm-eq-signed-list-lconjunction*:
 $x \in \text{FreeGroup } S \implies$
permutation (lift-signed-lconjperm x) =
signed-list-lconjunction (map fst (freeword x))
<proof>

lemma *even-count-lconjseq-rev-relator*:
 $s \in S \implies t \in S \implies \text{even } (\text{count-list } (\text{lconjseq } (\text{rev } (\text{pair-relator-list } s \ t)))) \ x$
<proof>

lemma *GroupByPresentationInducedFun-S-R-signed-lconjunction*:
GroupByPresentationInducedFun $S \ P$ signed-lconjpermutation
<proof>

end

5.2 Coxeter-like systems with deletion

Here we add the so-called deletion condition as an assumption, and explore its consequences.

5.2.1 Locale definition

locale *PreCoxeterSystemWithDeletion* = *PreCoxeterSystem* S
for $S :: 'w::\text{group-add set}$
+ **assumes** *deletion*:
 $ss \in \text{lists } S \implies \neg \text{reduced-word } S \ ss \implies$
 $\exists a \ b \ as \ bs \ cs. \ ss = as \ @ \ [a] \ @ \ bs \ @ \ [b] \ @ \ cs \wedge$
 $\text{sum-list } ss = \text{sum-list } (as@bs@cs)$

5.2.2 Consequences of the deletion condition

context *PreCoxeterSystemWithDeletion*
begin

lemma *deletion-reduce*:
 $ss \in \text{lists } S \implies \exists ts. \ ts \in \text{ssubseqs } ss \cap \text{reduced-words-for } S \ (\text{sum-list } ss)$
<proof>

lemma *deletion-reduce'*:
 $ss \in \text{lists } S \implies \exists ts \in \text{reduced-words-for } S \ (\text{sum-list } ss). \ \text{set } ts \subseteq \text{set } ss$

<proof>

end

5.2.3 The exchange condition

The exchange condition states that, given a reduced word in the generators, if prepending a letter to the word does not remain reduced, then the new word can be shortened to a word equivalent to the original one by deleting some letter other than the prepended one. Thus, one able to exchange some letter for the addition of a desired letter at the beginning of a word, without changing the element represented.

context *PreCoxeterSystemWithDeletion*
begin

lemma *exchange:*

assumes $s \in S$ *S-reduced-for* w $ss \neg S\text{-reduced}$ $(s\#ss)$

shows $\exists t$ as bs . $ss = as\@t\#bs \wedge$ *reduced-word-for* S w $(s\#as\@bs)$

<proof>

lemma *reduced-head-imp-exchange:*

assumes *reduced-word-for* S w $(s\#as)$ *reduced-word-for* S cs

shows $\exists a$ ds es . $cs = ds\@[a]\@es \wedge$ *reduced-word-for* S w $(s\#ds\@es)$

<proof>

end

5.2.4 More on words in generators containing alternating subwords

Here we explore more of the mechanics of manipulating words over S that contain alternating subwords, in preparation of the word problem.

context *PreCoxeterSystemWithDeletion*
begin

lemma *two-reduced-heads-imp-reduced-alt-step:*

assumes $s \neq t$ *reduced-word-for* S w $(t\#bs)$ $n < \text{relfun } s \ t \vee \text{relfun } s \ t = 0$
reduced-word-for S w $(\text{alternating-list } n \ s \ t \ @ \ cs)$

shows $\exists ds$. *reduced-word-for* S w $(\text{alternating-list } (\text{Suc } n) \ t \ s \ @ \ ds)$

<proof>

lemma *two-reduced-heads-imp-reduced-alt':*

assumes $s \neq t$ *reduced-word-for* S w $(s\#as)$ *reduced-word-for* S w $(t\#bs)$

shows $n \leq \text{relfun } s \ t \vee \text{relfun } s \ t = 0 \implies (\exists cs$
reduced-word-for S w $(\text{alternating-list } n \ s \ t \ @ \ cs) \vee$
reduced-word-for S w $(\text{alternating-list } n \ t \ s \ @ \ cs)$
)

<proof>

lemma *two-reduced-heads-imp-reduced-alt:*

assumes $s \neq t$ *reduced-word-for* S w ($s \# as$) *reduced-word-for* S w ($t \# bs$)

shows $\exists cs.$ *reduced-word-for* S w (*pair-relator-halflist* s t @ cs)

<proof>

lemma *two-reduced-heads-imp-nzero-relfun:*

assumes $s \neq t$ *reduced-word-for* S w ($s \# as$) *reduced-word-for* S w ($t \# bs$)

shows *relfun* s $t \neq 0$

<proof>

end

5.2.5 The word problem

Here we establish the other direction of the word problem for reduced words.

context *PreCoxeterSystemWithDeletion*

begin

lemma *reduced-word-problem-ConsCons-step:*

assumes $\bigwedge y$ ss $ts.$ $\llbracket S\text{-length } y < S\text{-length } w; y \neq 0; \text{reduced-word-for } S$ y $ss;$
reduced-word-for S y ts $\rrbracket \implies \exists xss.$ *flip-altsublist-chain* (ss # xss @ $[ts]$)

reduced-word-for S w ($a \# as$) *reduced-word-for* S w ($b \# bs$) $a \neq b$

shows $\exists xss.$ *flip-altsublist-chain* ($(a \# as) \# xss @ [b \# bs]$)

<proof>

lemma *reduced-word-problem:*

$\llbracket w \neq 0; \text{reduced-word-for } S$ w $ss; \text{reduced-word-for } S$ w ts $\rrbracket \implies$

$\exists xss.$ *flip-altsublist-chain* ($ss \# xss @ [ts]$)

<proof>

lemma *reduced-word-letter-set:*

assumes *S-reduced-for* w ss

shows *reduced-letter-set* S $w = \text{set } ss$

<proof>

end

5.2.6 Special subgroups and cosets

Recall that special subgroups are those generated by subsets of the generating set S . Here we show that the presence of the deletion condition guarantees that the collection of special subgroups and their left cosets forms a poset under reverse inclusion that satisfies the necessary properties to ensure that the poset of simplices in the associated simplicial complex is isomorphic to this poset of special cosets.

context *PreCoxeterSystemWithDeletion*

begin

lemma *special-subgroup-int-S:*

assumes $T \in Pow\ S$
shows $\langle T \rangle \cap S = T$

<proof>

lemma *special-subgroup-inj: inj-on genby (Pow S)*

<proof>

lemma *special-subgroup-genby-subset-ordering-iso:*

subset-ordering-iso (Pow S) genby

<proof>

lemmas *special-subgroup-genby-rev-mono*

$= OrderingSetIso.rev-ordsetmap[OF\ special-subgroup-genby-subset-ordering-iso]$

lemma *special-subgroup-word-length:*

assumes $T \in Pow\ S\ w \in \langle T \rangle$
shows $word-length\ T\ w = S-length\ w$

<proof>

lemma *S-subset-reduced-imp-S-reduced:*

$T \in Pow\ S \implies reduced-word\ T\ ts \implies S-reduced\ ts$

<proof>

lemma *smallest-genby: T ∈ Pow S ⟹ w ∈ ⟨T⟩ ⟹ reduced-letter-set S w ⊆ T*

<proof>

lemma *special-cosets-below-in:*

assumes $w \in W\ T \in Pow\ S$
shows $\mathcal{P}.\supseteq(w + o \langle T \rangle) = (\bigcup R \in (Pow\ S).\supseteq T. \{w + o \langle R \rangle\})$

<proof>

lemmas *special-coset-inj*

$= comp-inj-on[OF\ special-subgroup-inj,\ OF\ inj-inj-on,\ OF\ lcoset-inj-on]$

lemma *special-coset-eq-imp-eq-gensets:*

$\llbracket T1 \in Pow\ S;\ T2 \in Pow\ S;\ w1 + o \langle T1 \rangle = w2 + o \langle T2 \rangle \rrbracket \implies T1 = T2$

<proof>

lemma *special-subgroup-special-coset-subset-ordering-iso:*

subset-ordering-iso (genby ' Pow S) ((+o) w)

<proof>

lemma *special-coset-subset-ordering-iso:*

subset-ordering-iso (Pow S) ((+o) w ◦ genby)

<proof>

lemmas *special-coset-subset-rev-mono* =
OrderingSetIso.rev-ordsetmap[*OF special-coset-subset-ordering-iso*]

lemma *special-coset-below-in-subset-ordering-iso*:
subset-ordering-iso ((*Pow S*). \supseteq *T*) ((*+o*) *w* \circ *genby*)
 ⟨*proof*⟩

lemma *special-coset-below-in-supset-ordering-iso*:
OrderingSetIso (\supseteq) (\supset) (\supseteq) (\supset) ((*Pow S*). \supseteq *T*) ((*+o*) *w* \circ *genby*)
 ⟨*proof*⟩

lemma *special-coset-pseudominimals*:
assumes *supset-pseudominimal-in* \mathcal{P} *X*
shows $\exists w s. w \in W \wedge s \in S \wedge X = w + o \langle S - \{s\} \rangle$
 ⟨*proof*⟩

lemma *special-coset-pseudominimal-in-below-in*:
assumes $w \in W$ $T \in Pow\ S$ *supset-pseudominimal-in* ($\mathcal{P}.\supseteq(w + o \langle T \rangle)$) *X*
shows $\exists s \in S - T. X = w + o \langle S - \{s\} \rangle$
 ⟨*proof*⟩

lemma *exclude-one-is-pseudominimal*:
assumes $w \in W$ $t \in S$
shows *supset-pseudominimal-in* \mathcal{P} ($w + o \langle S - \{t\} \rangle$)
 ⟨*proof*⟩

lemma *exclude-one-is-pseudominimal-in-below-in*:
 [$w \in W$; $T \in Pow\ S$; $s \in S - T$] \implies
supset-pseudominimal-in ($\mathcal{P}.\supseteq(w + o \langle T \rangle)$) ($w + o \langle S - \{s\} \rangle$)
 ⟨*proof*⟩

lemma *glb-special-subset-coset*:
assumes wTT' : $w \in W$ $T \in Pow\ S$ $T' \in Pow\ S$
defines U : $U \equiv T \cup T' \cup$ *reduced-letter-set* S *w*
shows *supset-glboun-d-in-of* \mathcal{P} $\langle T \rangle$ ($w + o \langle T' \rangle$) $\langle U \rangle$
 ⟨*proof*⟩

lemma *glb-special-subset-coset-ex*:
assumes $w \in W$ $T \in Pow\ S$ $T' \in Pow\ S$
shows $\exists B. \textit{supset-glboun-d-in-of}$ \mathcal{P} $\langle T \rangle$ ($w + o \langle T' \rangle$) B
 ⟨*proof*⟩

lemma *special-cosets-have-glbs*:
assumes $X \in \mathcal{P}$ $Y \in \mathcal{P}$
shows $\exists B. \textit{supset-glboun-d-in-of}$ \mathcal{P} X Y B
 ⟨*proof*⟩

end

5.3 Coxeter systems

5.3.1 Locale definition and transfer from the associated free group

Now we consider groups generated by elements of order two with an additional assumption to ensure that the natural correspondence between the group W and the group presentation on the generating set S and its relations is bijective. Below, such groups will be shown to satisfy the deletion condition.

locale *CoxeterSystem* = *PreCoxeterSystem* S
for S :: ' w ::group-add set
+ **assumes** *induced-id-inj*: *inj-on induced-id* G

lemma (in *PreCoxeterSystem*) *CoxeterSystemI*:
assumes $\bigwedge g. g \in G \implies \text{induced-id } g = 0 \implies g = 0$
shows *CoxeterSystem* S
⟨*proof*⟩

context *CoxeterSystem*
begin

abbreviation *inv-induced-id* \equiv *GroupPresentation.inv-induced-id* $S R$

lemma *GroupPresentation-S-R*: *GroupPresentation* $S R$
⟨*proof*⟩

lemmas *inv-induced-id-sum-list* =
GroupPresentation.inv-induced-id-sum-list-S[*OF GroupPresentation-S-R*]

end

5.3.2 The deletion condition is necessary

Call an element of W a reflection if it is a conjugate of a generating element (and so is also of order two). Here we use the action of words over S on such reflections to show that Coxeter systems satisfy the deletion condition.

context *CoxeterSystem*
begin

abbreviation *induced-signed-lconjperm* \equiv
GroupByPresentationInducedFun.induced-hom $S P$ *signed-lconjpermutation*

definition *flipped-reflections* :: ' $w \Rightarrow 'w$ set
where *flipped-reflections* $w \equiv$
 $\{t \in \mathcal{H}. \text{induced-signed-lconjperm } (\text{inv-induced-id } (-w)) \rightarrow$
 $(t, \text{True}) = (\text{rconjby } w t, \text{False})\}$

lemma *induced-signed-lconjperm-inv-induced-id-sum-list*:

$ss \in \text{lists } S \implies \text{induced-signed-lconjperm } (\text{inv-induced-id } (\text{sum-list } ss)) =$
 $\text{sum-list } (\text{map signed-lconjpermutation } ss)$
 $\langle \text{proof} \rangle$

lemma *induced-signed-eq-lconjpermutation:*

$ss \in \text{lists } S \implies$
 $\text{permutation } (\text{induced-signed-lconjperm } (\text{inv-induced-id } (\text{sum-list } ss))) =$
 $\text{signed-list-lconjunction } ss$
 $\langle \text{proof} \rangle$

lemma *flipped-reflections-odd-lconjseq:*

assumes $ss \in \text{lists } S$
shows $\text{flipped-reflections } (\text{sum-list } ss) = \{t \in \mathcal{H}. \text{ odd } (\text{count-list } (\text{lconjseq } ss) t)\}$
 $\langle \text{proof} \rangle$

lemma *flipped-reflections-in-lconjseq:*

$ss \in \text{lists } S \implies \text{flipped-reflections } (\text{sum-list } ss) \subseteq \text{set } (\text{lconjseq } ss)$
 $\langle \text{proof} \rangle$

lemma *flipped-reflections-distinct-lconjseq-eq-lconjseq:*

assumes $ss \in \text{lists } S$ *distinct* $(\text{lconjseq } ss)$
shows $\text{flipped-reflections } (\text{sum-list } ss) = \text{set } (\text{lconjseq } ss)$
 $\langle \text{proof} \rangle$

lemma *flipped-reflections-reduced-eq-lconjseq:*

$S\text{-reduced } ss \implies \text{flipped-reflections } (\text{sum-list } ss) = \text{set } (\text{lconjseq } ss)$
 $\langle \text{proof} \rangle$

lemma *card-flipped-reflections:*

assumes $w \in W$
shows $\text{card } (\text{flipped-reflections } w) = S\text{-length } w$
 $\langle \text{proof} \rangle$

end

sublocale $\text{CoxeterSystem} < \text{PreCoxeterSystemWithDeletion}$

$\langle \text{proof} \rangle$

5.3.3 The deletion condition is sufficient

Now we come full circle and show that a pair consisting of a group and a generating set of order-two elements that satisfies the deletion condition affords a presentation that makes it a Coxeter system.

context $\text{PreCoxeterSystemWithDeletion}$

begin

lemma *reducible-by-flipping:*

$ss \in \text{lists } S \implies \neg S\text{-reduced } ss \implies$
 $\exists xss \text{ as } t \text{ bs. flip-altsublist-chain } (ss \# xss @ [as@[t,t]@bs])$

<proof>

lemma *freeliftid-kernel'*:

$ss \in \text{lists } S \implies \text{sum-list } ss = 0 \implies \text{Abs-freelist } ss \in Q$

<proof>

lemma *freeliftid-kernel*:

assumes $c \in \text{FreeGroup } S$ *freeliftid* $c = 0$

shows $c \in Q$

<proof>

lemma *induced-id-kernel*:

$c \in \text{FreeGroup } S \implies \text{induced-id } ([\text{FreeGroup } S | c | Q]) = 0 \implies c \in Q$

<proof>

theorem *CoxeterSystem*: *CoxeterSystem* S

<proof>

end

5.3.4 The Coxeter system associated to a thin chamber complex with many foldings

We now show that the fundamental automorphisms in a thin chamber complex with many foldings satisfy the deletion condition, and hence form a Coxeter system.

context *ThinChamberComplexManyFoldings*

begin

lemma *not-reduced-word-not-min-gallery*:

assumes $ss \in \text{lists } S \neg \text{reduced-word } S$ ss

shows $\neg \text{min-gallery } (\text{map } (\lambda w. w' \rightarrow C0) (\text{sums } ss))$

<proof>

lemma *S-list-not-min-gallery-double-split*:

assumes $ss \in \text{lists } S$ $ss \neq []$ $\neg \text{min-gallery } (\text{map } (\lambda w. w' \rightarrow C0) (\text{sums } ss))$

shows

$\exists f g$ as s bs t cs .

$(f, g) \in \text{foldpairs} \wedge$

$\text{sum-list } as \rightarrow C0 \in f \vdash \mathcal{C} \wedge$

$\text{sum-list } (as @ [s]) \rightarrow C0 \in g \vdash \mathcal{C} \wedge$

$\text{sum-list } (as @ [s] @ bs) \rightarrow C0 \in g \vdash \mathcal{C} \wedge$

$\text{sum-list } (as @ [s] @ bs @ [t]) \rightarrow C0 \in f \vdash \mathcal{C} \wedge$

$ss = as @ [s] @ bs @ [t] @ cs$

<proof>

lemma *fold-end-sum-chain-fg*:

fixes $f g :: 'a \Rightarrow 'a$

defines $s : s \equiv \text{induced-automorph } f g$
assumes $fg : (f,g) \in \text{foldpairs}$
and $as : as \in \text{lists } S$
and $s : s \in S$
and $sep: \text{sum-list } as \xrightarrow{} C0 \in f\vdash\mathcal{C} \text{sum-list } (as@[s]) \xrightarrow{} C0 \in g\vdash\mathcal{C}$
shows $bs \in \text{lists } S \implies$
 $s \text{ 'sum-list } (as@[s]@bs) \xrightarrow{} C0 = \text{sum-list } (as@bs) \xrightarrow{} C0$
 $\langle \text{proof} \rangle$

lemma *fold-end-sum-chain-gf*:
fixes $fg :: 'a \Rightarrow 'a$
defines $s \equiv \text{induced-automorph } f g$
assumes $fg : (f,g) \in \text{foldpairs}$
and $as \in \text{lists } S \ s \in S \ bs \in \text{lists } S$
 $\text{sum-list } as \xrightarrow{} C0 \in g\vdash\mathcal{C}$
 $\text{sum-list } (as@[s]) \xrightarrow{} C0 \in f\vdash\mathcal{C}$
shows $s \text{ 'sum-list } (as@[s]@bs) \xrightarrow{} C0 = \text{sum-list } (as@bs) \xrightarrow{} C0$
 $\langle \text{proof} \rangle$

lemma *fold-middle-sum-chain*:
assumes $fg : (f,g) \in \text{foldpairs}$
and $S : as \in \text{lists } S \ s \in S \ bs \in \text{lists } S \ t \in S \ cs \in \text{lists } S$
and $sep: \text{sum-list } as \xrightarrow{} C0 \in f\vdash\mathcal{C}$
 $\text{sum-list } (as@[s]) \xrightarrow{} C0 \in g\vdash\mathcal{C}$
 $\text{sum-list } (as@[s]@bs) \xrightarrow{} C0 \in g\vdash\mathcal{C} \text{sum-list } (as@[s]@bs@[t]) \xrightarrow{} C0$
 $\in f\vdash\mathcal{C}$
shows $\text{sum-list } (as@[s]@bs@[t]@cs) \xrightarrow{} C0 = \text{sum-list } (as@bs@cs) \xrightarrow{} C0$
 $\langle \text{proof} \rangle$

lemma *S-list-not-min-gallery-deletion*:
fixes $ss :: 'a \text{ permutation list}$
defines $w : w \equiv \text{sum-list } ss$
assumes $ss: ss \in \text{lists } S \ ss \neq [] \ \neg \text{min-gallery } (\text{map } (\lambda w. w \xrightarrow{} C0) (\text{sums } ss))$
shows $\exists a \ b \ as \ bs \ cs. ss = as@[a]@bs@[b]@cs \wedge w = \text{sum-list } (as@bs@cs)$
 $\langle \text{proof} \rangle$

lemma *deletion*:
 $ss \in \text{lists } S \implies \neg \text{reduced-word } S \ ss \implies$
 $\exists a \ b \ as \ bs \ cs. ss = as@[a]@bs@[b]@cs \wedge \text{sum-list } ss = \text{sum-list } (as@bs@cs)$
 $\langle \text{proof} \rangle$

lemma *PreCoxeterSystemWithDeletion*: *PreCoxeterSystemWithDeletion* S
 $\langle \text{proof} \rangle$

lemma *CoxeterSystem*: *CoxeterSystem* S
 $\langle \text{proof} \rangle$

end

5.4 Coxeter complexes

5.4.1 Locale and complex definitions

Now we add in the assumption that the generating set is finite, and construct the associated Coxeter complex from the poset of special cosets.

```
locale CoxeterComplex = CoxeterSystem S
  for S :: 'w::group-add set
+ assumes finite-genset: finite S
begin

definition TheComplex :: 'w set set set
  where TheComplex  $\equiv$  ordering.PosetComplex ( $\supseteq$ ) ( $\supset$ )  $\mathcal{P}$ 
abbreviation  $\Sigma \equiv$  TheComplex

end
```

5.4.2 As a simplicial complex

Here we record the fact that the Coxeter complex associated to a Coxeter system is a simplicial complex, and note that the poset of special cosets is complex-like. This last fact allows us to reason about the complex by reasoning about the poset, via the poset isomorphism *ComplexLikePoset.smap*.

```
context CoxeterComplex
begin

lemma simplex-like-special-cosets:
  assumes  $X \in \mathcal{P}$ 
  shows supset-simplex-like ( $\mathcal{P}.\supseteq X$ )
   $\langle$ proof $\rangle$ 

lemma SimplicialComplex- $\Sigma$ : SimplicialComplex  $\Sigma$ 
   $\langle$ proof $\rangle$ 

lemma ComplexLikePoset-special-cosets: ComplexLikePoset ( $\supseteq$ ) ( $\supset$ )  $\mathcal{P}$ 
   $\langle$ proof $\rangle$ 

abbreviation smap  $\equiv$  ordering.poset-simplex-map ( $\supseteq$ ) ( $\supset$ )  $\mathcal{P}$ 

lemmas smap-def = ordering.poset-simplex-map-def[OF supset-poset, of  $\mathcal{P}$ ]

lemma ordsetmap-smap:  $\llbracket X \in \mathcal{P}; Y \in \mathcal{P}; X \supseteq Y \rrbracket \implies \text{smap } X \subseteq \text{smap } Y$ 
   $\langle$ proof $\rangle$ 

lemma rev-ordsetmap-smap:  $\llbracket X \in \mathcal{P}; Y \in \mathcal{P}; \text{smap } X \subseteq \text{smap } Y \rrbracket \implies X \supseteq Y$ 
   $\langle$ proof $\rangle$ 

lemma smap-onto-PosetComplex: smap '  $\mathcal{P} = \Sigma$ 
```


<proof>

lemmas *simplices-conv-special-cosets = smap-onto-PosetComplex*[*THEN sym*]

lemma *smap-into-PosetComplex*: $X \in \mathcal{P} \implies \text{smap } X \in \Sigma$

<proof>

lemma *smap-pseudominimal*:

$w \in W \implies s \in S \implies \text{smap } (w + o \langle S - \{s\} \rangle) = \{w + o \langle S - \{s\} \rangle\}$

<proof>

lemma *exclude-one-notin-smap-singleton*:

$s \in S \implies w + o \langle S - \{s\} \rangle \notin \text{smap } (w + o \langle \{s\} \rangle)$

<proof>

lemma *maxsimp-vertices*: $w \in W \implies s \in S \implies w + o \langle S - \{s\} \rangle \in \text{smap } \{w\}$

<proof>

lemma *maxsimp-singleton*:

assumes $w \in W$

shows $\text{SimplicialComplex.maxsimp } \Sigma (\text{smap } \{w\})$

<proof>

lemma *maxsimp-is-singleton*:

assumes $\text{SimplicialComplex.maxsimp } \Sigma x$

shows $\exists w \in W. \text{smap } \{w\} = x$

<proof>

lemma *maxsimp-vertex-conv-special-coset*:

$w \in W \implies X \in \text{smap } \{w\} \implies \exists s \in S. X = w + o \langle S - \{s\} \rangle$

<proof>

lemma *vertices*: $w \in W \implies s \in S \implies w + o \langle S - \{s\} \rangle \in \bigcup \Sigma$

<proof>

lemma *smap0-conv-special-subgroups*:

$\text{smap } 0 = (\lambda s. \langle S - \{s\} \rangle) \text{ ' } S$

<proof>

lemma *S-bij-betw-chamber0*: *bij-betw* $(\lambda s. \langle S - \{s\} \rangle) S (\text{smap } 0)$

<proof>

lemma *smap-singleton-conv-W-image*:

$w \in W \implies \text{smap } \{w\} = ((+o) w) \text{ ' } (\text{smap } 0)$

<proof>

lemma *W-lcoset-bij-betw-singletons*:

assumes $w \in W$

shows *bij-betw* $((+o) w) (\text{smap } 0) (\text{smap } \{w\})$

<proof>

lemma *facets*:

assumes $w \in W \ s \in S$

shows $\text{smap } (w + o \ \langle \{s\} \rangle) \triangleleft \text{smap } \{w\}$

<proof>

lemma *facets'*: $w \in W \implies s \in S \implies \text{smap } \{w, w+s\} \triangleleft \text{smap } \{w\}$

<proof>

lemma *adjacent*: $w \in W \implies s \in S \implies \text{smap } \{w+s\} \sim \text{smap } \{w\}$

<proof>

lemma *singleton-adjacent-0*: $s \in S \implies \text{smap } \{s\} \sim \text{smap } 0$

<proof>

end

5.4.3 As a chamber complex

Now we verify that a Coxeter complex is a chamber complex.

context *CoxeterComplex*

begin

abbreviation *chamber* $\equiv \text{SimplicialComplex.maxsimp } \Sigma$

abbreviation *gallery* $\equiv \text{SimplicialComplex.maxsimpchain } \Sigma$

lemmas *chamber-singleton* = *maxsimp-singleton*

lemmas *chamber-vertex-conv-special-coset* = *maxsimp-vertex-conv-special-coset*

lemmas *chamber-vertices* = *maxsimp-vertices*

lemmas *chamber-is-singleton* = *maxsimp-is-singleton*

lemmas *faces* = *SimplicialComplex.faces* [*OF SimplicialComplex- Σ*]

lemmas *gallery-def* = *SimplicialComplex.maxsimpchain-def* [*OF SimplicialComplex- Σ*]

lemmas *gallery-rev* = *SimplicialComplex.maxsimpchain-rev* [*OF SimplicialComplex- Σ*]

lemmas *chamberD-simplex* =
SimplicialComplex.maxsimpD-simplex [*OF SimplicialComplex- Σ*]

lemmas *gallery-CConsI* =
SimplicialComplex.maxsimpchain-CConsI [*OF SimplicialComplex- Σ*]

lemmas *gallery-overlap-join* =
SimplicialComplex.maxsimpchain-overlap-join [*OF SimplicialComplex- Σ*]

lemma *word-gallery-to-0*:

$ss \neq [] \implies ss \in \text{lists } S \implies \exists xs. \text{gallery } (\text{smap } \{\text{sum-list } ss\} \# xs @ [\text{smap } 0])$
<proof>

lemma *gallery-to-0*:
assumes $w \in W$ $w \neq 0$
shows $\exists xs. \text{gallery } (smap \{w\} \# xs @ [smap 0])$
 $\langle \text{proof} \rangle$

lemma *ChamberComplex- Σ : ChamberComplex Σ*
 $\langle \text{proof} \rangle$

lemma *card-chamber: chamber $x \implies \text{card } x = \text{card } S$*
 $\langle \text{proof} \rangle$

lemma *vertex-conv-special-coset*:
 $X \in \bigcup \Sigma \implies \exists w s. w \in W \wedge s \in S \wedge X = w + o \langle S - \{s\} \rangle$
 $\langle \text{proof} \rangle$

end

5.4.4 The Coxeter complex associated to a thin chamber complex with many foldings

Having previously verified that the fundamental automorphisms in a thin chamber complex with many foldings form a Coxeter system, we now record the existence of a chamber complex isomorphism onto the associated Coxeter complex.

context *ThinChamberComplexManyFoldings*
begin

lemma *CoxeterComplex: CoxeterComplex S*
 $\langle \text{proof} \rangle$

abbreviation $\Sigma \equiv \text{CoxeterComplex.TheComplex } S$

lemma *S-list-not-min-gallery-not-reduced*:
assumes $ss \neq [] \wedge \neg \text{min-gallery } (\text{map } (\lambda w. w' \rightarrow C0) (\text{sums } ss))$
shows $\neg \text{reduced-word } S \text{ } ss$
 $\langle \text{proof} \rangle$

lemma *reduced-S-list-min-gallery*:
 $ss \neq [] \implies \text{reduced-word } S \text{ } ss \implies \text{min-gallery } (\text{map } (\lambda w. w' \rightarrow C0) (\text{sums } ss))$
 $\langle \text{proof} \rangle$

lemma *fundchamber-vertex-stabilizer1*:
fixes t
defines $v: v \equiv \text{fundantivertex } t$
assumes $tw: t \in S \ w \in W \ w \rightarrow v = v$
shows $w \in \langle S - \{t\} \rangle$
 $\langle \text{proof} \rangle$

lemma *fundchamber-vertex-stabilizer2*:

assumes $s: s \in S$

defines $v: v \equiv \text{fundantivertex } s$

shows $w \in \langle S - \{s\} \rangle \implies w \rightarrow v = v$

<proof>

lemma *label-wrt-special-coset1*:

assumes *label-wrt* $C0$ φ *fixespointwise* φ $C0$ $w0 \in W$ $s \in S$

defines $v \equiv \text{fundantivertex } s$

shows $\{w \in W. w \rightarrow \varphi (w0 \rightarrow v) = w0 \rightarrow v\} = w0 +_o \langle S - \{s\} \rangle$

<proof>

lemma *label-wrt-special-coset1'*:

assumes *label-wrt* $C0$ φ *fixespointwise* φ $C0$ $w0 \in W$ $v \in C0$

defines $s \equiv \text{fundantipermutation } v$

shows $\{w \in W. w \rightarrow \varphi (w0 \rightarrow v) = w0 \rightarrow v\} = w0 +_o \langle S - \{s\} \rangle$

<proof>

lemma *label-wrt-special-coset2'*:

assumes *label-wrt* $C0$ φ *fixespointwise* φ $C0$ $w0 \in W$ $v \in w0' \rightarrow C0$

defines $s \equiv \text{fundantipermutation } (\varphi v)$

shows $\{w \in W. w \rightarrow \varphi v = v\} = w0 +_o \langle S - \{s\} \rangle$

<proof>

lemma *label-stab-map-W-fundchamber-image*:

assumes *label-wrt* $C0$ φ *fixespointwise* φ $C0$ $w0 \in W$

defines $\psi \equiv \lambda v. \{w \in W. w \rightarrow (\varphi v) = v\}$

shows $\psi'(w0' \rightarrow C0) = \text{CoxeterComplex.smap } S \{w0\}$

<proof>

lemma *label-stab-map-chamber-map*:

assumes $\varphi: \text{label-wrt } C0$ φ *fixespointwise* φ $C0$

and $C: \text{chamber } C$

defines $\psi: \psi \equiv \lambda v. \{w \in W. w \rightarrow (\varphi v) = v\}$

shows $\text{CoxeterComplex.chamber } S (\psi' C)$

<proof>

lemma *label-stab-map-inj-on-vertices*:

assumes $\varphi: \text{label-wrt } C0$ φ *fixespointwise* φ $C0$

defines $\psi: \psi \equiv \lambda v. \{w \in W. w \rightarrow (\varphi v) = v\}$

shows *inj-on* $\psi (\bigcup X)$

<proof>

lemma *label-stab-map-surj-on-vertices*:

assumes *label-wrt* $C0$ φ *fixespointwise* φ $C0$

defines $\psi \equiv \lambda v. \{w \in W. w \rightarrow (\varphi v) = v\}$

shows $\psi'(\bigcup X) = \bigcup \Sigma$

<proof>

lemma *label-stab-map-bij-betw-vertices:*
assumes *label-wrt C0 φ fixespointwise φ C0*
defines $\psi \equiv \lambda v. \{w \in W. w \rightarrow (\varphi v) = v\}$
shows *bij-betw ψ (∪ X) (∪ Σ)*
<proof>

lemma *label-stab-map-bij-betw-W-chambers:*
assumes *label-wrt C0 φ fixespointwise φ C0 w0 ∈ W*
defines $\psi \equiv \lambda v. \{w \in W. w \rightarrow (\varphi v) = v\}$
shows *bij-betw ψ (w0' → C0) (CoxeterComplex.smap S {w0})*
<proof>

lemma *label-stab-map-surj-on-simplices:*
assumes *φ: label-wrt C0 φ fixespointwise φ C0*
defines $\psi: \psi \equiv \lambda v. \{w \in W. w \rightarrow (\varphi v) = v\}$
shows $\psi \vdash X = \Sigma$
<proof>

lemma *label-stab-map-iso-to-coxeter-complex:*
assumes *label-wrt C0 φ fixespointwise φ C0*
defines $\psi \equiv \lambda v. \{w \in W. w \rightarrow (\varphi v) = v\}$
shows *ChamberComplexIsomorphism X Σ ψ*
<proof>

lemma *ex-iso-to-coxeter-complex':*
 $\exists \psi. \text{ChamberComplexIsomorphism } X \text{ (CoxeterComplex.TheComplex } S) \psi$
<proof>

lemma *ex-iso-to-coxeter-complex:*
 $\exists S::'a \text{ permutation set. CoxeterComplex } S \wedge$
 $(\exists \psi. \text{ChamberComplexIsomorphism } X \text{ (CoxeterComplex.TheComplex } S) \psi)$
<proof>

end

end

6 Buildings

In this section we collect the axioms for a (thick) building in a locale, and prove that apartments in a building are uniformly Coxeter.

theory *Building*
imports *Coxeter*

begin

6.1 Apartment systems

First we describe and explore the basic structure of apartment systems. An apartment system is a collection of isomorphic thin chamber subcomplexes with certain intersection properties.

6.1.1 Locale and basic facts

```

locale ChamberComplexWithApartmentSystem = ChamberComplex X
  for X :: 'a set set
+ fixes A :: 'a set set set
  assumes subcomplexes      : A ∈ A ⇒ ChamberSubcomplex A
  and thincomplexes        : A ∈ A ⇒ ThinChamberComplex A
  and no-trivial-apartments: {} ∉ A
  and containtwo          :
    chamber C ⇒ chamber D ⇒ ∃ A ∈ A. C ∈ A ∧ D ∈ A
  and intersecttwo         :
    [ A ∈ A; A' ∈ A; x ∈ A ∩ A'; C ∈ A ∩ A'; chamber C ] ⇒
    ∃ f. ChamberComplexIsomorphism A A' f ∧ fixespointwise f x ∧
    fixespointwise f C
begin

```

```

lemmas complexes          = ChamberSubcomplexD-complex [OF subcomplexes]
lemmas apartment-simplices = ChamberSubcomplexD-sub   [OF subcomplexes]
lemmas chamber-in-apartment = chamber-in-subcomplex  [OF subcomplexes]
lemmas apartment-chamber   = subcomplex-chamber      [OF subcomplexes]
lemmas gallery-in-apartment = gallery-in-subcomplex  [OF subcomplexes]
lemmas apartment-gallery   = subcomplex-gallery      [OF subcomplexes]
lemmas min-gallery-in-apartment = min-gallery-in-subcomplex [OF subcomplexes]

```

```

lemmas apartment-simplex-in-max =
  ChamberComplex.simplex-in-max [OF complexes]

```

```

lemmas apartment-faces =
  ChamberComplex.faces [OF complexes]

```

```

lemmas apartment-chamber-system-def =
  ChamberComplex.chamber-system-def [OF complexes]

```

```

lemmas apartment-chamberD-simplex =
  ChamberComplex.chamberD-simplex [OF complexes]

```

```

lemmas apartment-chamber-distance-def =
  ChamberComplex.chamber-distance-def [OF complexes]

```

```

lemmas apartment-galleryD-chamber =
  ChamberComplex.galleryD-chamber [OF complexes]

```

```

lemmas apartment-gallery-least-length =

```

ChamberComplex.gallery-least-length [OF complexes]

lemmas *apartment-min-galleryD-gallery* =
ChamberComplex.min-galleryD-gallery [OF complexes]

lemmas *apartment-min-gallery-pgallery* =
ChamberComplex.min-gallery-pgallery [OF complexes]

lemmas *apartment-trivial-morphism* =
ChamberComplex.trivial-morphism [OF complexes]

lemmas *apartment-chamber-system-simplices* =
ChamberComplex.chamber-system-simplices [OF complexes]

lemmas *apartment-min-gallery-least-length* =
ChamberComplex.min-gallery-least-length [OF complexes]

lemmas *apartment-vertex-set-int* =
ChamberComplex.vertex-set-int[OF complexes complexes]

lemmas *apartment-standard-uniqueness-pgallery-betw* =
ThinChamberComplex.standard-uniqueness-pgallery-betw[OF thincomplexes]

lemmas *apartment-standard-uniqueness* =
ThinChamberComplex.standard-uniqueness[OF thincomplexes]

lemmas *apartment-standard-uniqueness-isomorphs* =
ThinChamberComplex.standard-uniqueness-isomorphs[OF thincomplexes]

abbreviation *supapartment C D* \equiv (SOME A. $A \in \mathcal{A} \wedge C \in A \wedge D \in A$)

lemma *supapartmentD*:
assumes *CD*: chamber C chamber D
defines *A* : $A \equiv \text{supapartment } C D$
shows $A \in \mathcal{A} \ C \in A \ D \in A$
⟨proof⟩

lemma *iso-fixespointwise-chamber-in-int-apartments*:
assumes *apartments*: $A \in \mathcal{A} \ A' \in \mathcal{A}$
and *chamber* : chamber C $C \in A \cap A'$
and *iso* : ChamberComplexIsomorphism A A' f fixespointwise f C
shows fixespointwise f $(\bigcup (A \cap A'))$
⟨proof⟩

lemma *strong-intersecttwo*:
 $\llbracket A \in \mathcal{A}; A' \in \mathcal{A}; \text{chamber } C; C \in A \cap A' \rrbracket \implies$
 $\exists f. \text{ChamberComplexIsomorphism } A \ A' \ f \wedge \text{fixespointwise } f \ (\bigcup (A \cap A'))$
⟨proof⟩

end

6.1.2 Isomorphisms between apartments

By standard uniqueness, the isomorphism between overlapping apartments guaranteed by the axiom *intersecttwo* is unique.

context *ChamberComplexWithApartmentSystem*
begin

lemma *ex1-apartment-iso*:

assumes $A \in \mathcal{A}$ $A' \in \mathcal{A}$ *chamber* C $C \in A \cap A'$

shows $\exists! f. \text{ChamberComplexIsomorphism } A \ A' \ f \wedge$
 $\text{fixespointwise } f \ (\bigcup (A \cap A')) \wedge \text{fixespointwise } f \ (-\bigcup A)$

— The third clause in the conjunction is to facilitate uniqueness.

<proof>

definition *the-apartment-iso* :: $'a \ \text{set} \ \text{set} \Rightarrow 'a \ \text{set} \ \text{set} \Rightarrow ('a \Rightarrow 'a)$

where *the-apartment-iso* $A \ A' \equiv$

$(\text{THE } f. \text{ChamberComplexIsomorphism } A \ A' \ f \wedge$
 $\text{fixespointwise } f \ (\bigcup (A \cap A')) \wedge \text{fixespointwise } f \ (-\bigcup A)$)

lemma *the-apartment-isoD*:

assumes $A \in \mathcal{A}$ $A' \in \mathcal{A}$ *chamber* C $C \in A \cap A'$

defines $f \equiv \text{the-apartment-iso } A \ A'$

shows $\text{ChamberComplexIsomorphism } A \ A' \ f \ \text{fixespointwise } f \ (\bigcup (A \cap A'))$
 $\text{fixespointwise } f \ (-\bigcup A)$

<proof>

lemmas *the-apartment-iso-apartment-chamber-map* =

$\text{ChamberComplexIsomorphism.chamber-map} \ [OF \ \text{the-apartment-isoD}(1)]$

lemmas *the-apartment-iso-apartment-simplex-map* =

$\text{ChamberComplexIsomorphism.simplex-map} \ [OF \ \text{the-apartment-isoD}(1)]$

lemma *the-apartment-iso-chamber-map*:

$[[A \in \mathcal{A}; B \in \mathcal{A}; \text{chamber } C; C \in A \cap B; \text{chamber } D; D \in A] \Longrightarrow$
 $\text{chamber } (\text{the-apartment-iso } A \ B \ 'D)$

<proof>

lemma *the-apartment-iso-comp*:

assumes *apartments*: $A \in \mathcal{A}$ $A' \in \mathcal{A}$ $A'' \in \mathcal{A}$

and *chamber* : $\text{chamber } C$ $C \in A \cap A' \cap A''$

defines $f \equiv \text{the-apartment-iso } A \ A'$

and $g \equiv \text{the-apartment-iso } A' \ A''$

and $h \equiv \text{the-apartment-iso } A \ A''$

defines $gf \equiv \text{restrict1 } (g \circ f) \ (\bigcup A)$

shows $h = gf$

<proof>

lemma *the-apartment-iso-int-im:*
assumes $A \in \mathcal{A}$ $A' \in \mathcal{A}$ *chamber* C $C \in A \cap A'$ $x \in A \cap A'$
defines $f \equiv \text{the-apartment-iso } A \ A'$
shows $f'x = x$
 $\langle \text{proof} \rangle$

end

6.1.3 Retractions onto apartments

Since the isomorphism between overlapping apartments is the identity on their intersection, starting with a fixed chamber in a fixed apartment, we can construct a retraction onto that apartment as follows. Given a vertex in the complex, that vertex is contained a chamber, and that chamber lies in a common apartment with the fixed chamber. We then apply to the vertex the apartment isomorphism from that common apartment to the fixed apartment. It turns out that the image of the vertex does not depend on the containing chamber and apartment chosen, and so since the isomorphisms between apartments used are unique, such a retraction onto an apartment is canonical.

context *ChamberComplexWithApartmentSystem*
begin

definition *canonical-retraction* $:: 'a \text{ set } \Rightarrow 'a \text{ set} \Rightarrow ('a \Rightarrow 'a)$
where *canonical-retraction* $A \ C =$
 $\text{restrict1 } (\lambda v. \text{the-apartment-iso } (\text{supapartment } (\text{supchamber } v) \ C) \ A \ v)$
 $(\bigcup X)$

lemma *canonical-retraction-retraction:*
assumes $A \in \mathcal{A}$ *chamber* C $C \in A$ $v \in \bigcup A$
shows *canonical-retraction* $A \ C \ v = v$
 $\langle \text{proof} \rangle$

lemma *canonical-retraction-simplex-retraction1:*
 $\llbracket A \in \mathcal{A}; \text{chamber } C; C \in A; a \in A \rrbracket \Longrightarrow$
 $\text{fixespointwise } (\text{canonical-retraction } A \ C) \ a$
 $\langle \text{proof} \rangle$

lemma *canonical-retraction-simplex-retraction2:*
 $\llbracket A \in \mathcal{A}; \text{chamber } C; C \in A; a \in A \rrbracket \Longrightarrow \text{canonical-retraction } A \ C \ 'a = a$
 $\langle \text{proof} \rangle$

lemma *canonical-retraction-uniform:*
assumes *apartments:* $A \in \mathcal{A}$ $B \in \mathcal{A}$
and *chambers* $: \text{chamber } C \ C \in A \cap B$
shows *fun-eq-on* $(\text{canonical-retraction } A \ C) \ (\text{the-apartment-iso } B \ A) \ (\bigcup B)$
 $\langle \text{proof} \rangle$

lemma *canonical-retraction-uniform-im:*
 $\llbracket A \in \mathcal{A}; B \in \mathcal{A}; \text{chamber } C; C \in A \cap B; x \in B \rrbracket \implies$
canonical-retraction $A \ C \ ' \ x = \text{the-apartment-iso } B \ A \ ' \ x$
 $\langle \text{proof} \rangle$

lemma *canonical-retraction-simplex-im:*
assumes $A \in \mathcal{A}$ *chamber* $C \ C \in A$
shows *canonical-retraction* $A \ C \vdash X = A$
 $\langle \text{proof} \rangle$

lemma *canonical-retraction-vertex-im:*
 $\llbracket A \in \mathcal{A}; \text{chamber } C; C \in A \rrbracket \implies \text{canonical-retraction } A \ C \ ' \ \bigcup X = \bigcup A$
 $\langle \text{proof} \rangle$

lemma *canonical-retraction:*
assumes $A \in \mathcal{A}$ *chamber* $C \ C \in A$
shows *ChamberComplexRetraction* X (*canonical-retraction* $A \ C$)
 $\langle \text{proof} \rangle$

lemma *canonical-retraction-comp-endomorphism:*
 $\llbracket A \in \mathcal{A}; B \in \mathcal{A}; \text{chamber } C; \text{chamber } D; C \in A; D \in B \rrbracket \implies$
ChamberComplexEndomorphism X
(*canonical-retraction* $A \ C \circ \text{canonical-retraction } B \ D$)
 $\langle \text{proof} \rangle$

lemma *canonical-retraction-comp-simplex-im-subset:*
 $\llbracket A \in \mathcal{A}; B \in \mathcal{A}; \text{chamber } C; \text{chamber } D; C \in A; D \in B \rrbracket \implies$
(*canonical-retraction* $A \ C \circ \text{canonical-retraction } B \ D$) $\vdash X \subseteq A$
 $\langle \text{proof} \rangle$

lemma *canonical-retraction-comp-apartment-endomorphism:*
 $\llbracket A \in \mathcal{A}; B \in \mathcal{A}; \text{chamber } C; \text{chamber } D; C \in A; D \in B \rrbracket \implies$
ChamberComplexEndomorphism A
(*restrict1* (*canonical-retraction* $A \ C \circ \text{canonical-retraction } B \ D$) ($\bigcup A$))
 $\langle \text{proof} \rangle$

end

6.1.4 Distances in apartments

Here we examine distances between chambers and between a facet and a chamber, especially with respect to canonical retractions onto an apartment. Note that a distance measured within an apartment is equal to the distance measured between the same objects in the wider chamber complex. In other words, the shortest distance between chambers can always be achieved within an apartment.

context *ChamberComplexWithApartmentSystem*

begin

lemma *apartment-chamber-distance:*

assumes $A \in \mathcal{A}$ chamber C chamber D $C \in A$ $D \in A$

shows $\text{ChamberComplex.chamber-distance } A \ C \ D = \text{chamber-distance } C \ D$
<proof>

lemma *apartment-min-gallery:*

assumes $A \in \mathcal{A}$ $\text{ChamberComplex.min-gallery } A \ Cs$

shows $\text{min-gallery } Cs$
<proof>

lemma *apartment-face-distance:*

assumes $A \in \mathcal{A}$ chamber C $C \in A$ $F \in A$

shows $\text{ChamberComplex.face-distance } A \ F \ C = \text{face-distance } F \ C$
<proof>

lemma *apartment-face-distance-eq-chamber-distance-compare-other-chamber:*

assumes $A \in \mathcal{A}$ chamber C chamber D chamber E $C \in A$ $D \in A$ $E \in A$

$z \triangleleft C$ $z \triangleleft D$ $C \neq D$ $\text{chamber-distance } C \ E \leq \text{chamber-distance } D \ E$

shows $\text{face-distance } z \ E = \text{chamber-distance } C \ E$
<proof>

lemma *canonical-retraction-face-distance-map:*

assumes $A \in \mathcal{A}$ chamber C chamber D $C \in A$ $F \subseteq C$

shows $\text{face-distance } F \ (\text{canonical-retraction } A \ C \ ' \ D) = \text{face-distance } F \ D$
<proof>

end

6.1.5 Special situation: a triangle of apartments and chambers

To facilitate proving that apartments in buildings have sufficient foldings to be Coxeter, we explore the situation of three chambers sharing a common facet, along with three apartments, each of which contains two of the chambers. A folding of one of the apartments is constructed by composing two apartment retractions, and by symmetry we automatically obtain an opposed folding.

locale $\text{ChamberComplexApartmentSystemTriangle} =$

$\text{ChamberComplexWithApartmentSystem } X \ \mathcal{A}$

for $X :: 'a \ \text{set} \ \text{set}$

and $\mathcal{A} :: 'a \ \text{set} \ \text{set} \ \text{set}$

+ **fixes** $A \ B \ B' :: 'a \ \text{set} \ \text{set}$

and $C \ D \ E \ z :: 'a \ \text{set}$

assumes $\text{apartments} \quad : A \in \mathcal{A} \ B \in \mathcal{A} \ B' \in \mathcal{A}$

and $\text{chambers} \quad : \text{chamber } C \ \text{chamber } D \ \text{chamber } E$

and $\text{facet} \quad : z \triangleleft C \ z \triangleleft D \ z \triangleleft E$

and $\text{in-apartments: } C \in A \cap B \ D \in A \cap B' \ E \in B \cap B'$

and $chambers-ne : D \neq C \ E \neq D \ C \neq E$
begin

abbreviation $fold-A \equiv canonical-retraction \ A \ D \circ canonical-retraction \ B \ C$

abbreviation $res-fold-A \equiv restrict1 \ fold-A \ (\bigcup A)$

abbreviation $opp-fold-A \equiv canonical-retraction \ A \ C \circ canonical-retraction \ B' \ D$

abbreviation $res-opp-fold-A \equiv restrict1 \ opp-fold-A \ (\bigcup A)$

lemma $rotate : ChamberComplexApartmentSystemTriangle \ X \ A \ B' \ A \ B \ D \ E \ C \ z$
 $\langle proof \rangle$

lemma $reflect : ChamberComplexApartmentSystemTriangle \ X \ A \ A \ B' \ B \ D \ C \ E \ z$
 $\langle proof \rangle$

lemma $facet-in-chambers : z \subseteq C \ z \subseteq D \ z \subseteq E$
 $\langle proof \rangle$

lemma $A-chambers :$
 $ChamberComplex.chamber \ A \ C \ ChamberComplex.chamber \ A \ D$
 $\langle proof \rangle$

lemma $res-fold-A-A-chamber-image :$
 $ChamberComplex.chamber \ A \ F \implies res-fold-A \ 'F = fold-A \ 'F$
 $\langle proof \rangle$

lemma $the-apartment-iso-middle-im : the-apartment-iso \ A \ B \ 'D = E$
 $\langle proof \rangle$

lemma $inside-canonical-retraction-chamber-images :$
 $canonical-retraction \ B \ C \ 'C = C$
 $canonical-retraction \ B \ C \ 'D = E$
 $canonical-retraction \ B \ C \ 'E = E$
 $\langle proof \rangle$

lemmas $in-canretract-chimages =$
 $inside-canonical-retraction-chamber-images$

lemma $outside-canonical-retraction-chamber-images :$
 $canonical-retraction \ A \ D \ 'C = C$
 $canonical-retraction \ A \ D \ 'D = D$
 $canonical-retraction \ A \ D \ 'E = C$
 $\langle proof \rangle$

lemma $fold-A-chamber-images :$
 $fold-A \ 'C = C \ fold-A \ 'D = C \ fold-A \ 'E = C$
 $\langle proof \rangle$

lemmas $opp-fold-A-chamber-images =$
 $ChamberComplexApartmentSystemTriangle.fold-A-chamber-images[OF \ reflect]$

lemma *res-fold-A-chamber-images*: $\text{res-fold-A } C = C \text{ res-fold-A } D = C$
 ⟨proof⟩

lemmas *res-opp-fold-A-chamber-images* =
ChamberComplexApartmentSystemTriangle.res-fold-A-chamber-images[*OF reflect*]

lemma *fold-A-fixespointwise1*: $\text{fixespointwise fold-A } C$
 ⟨proof⟩

lemmas *opp-fold-A-fixespointwise2* =
ChamberComplexApartmentSystemTriangle.fold-A-fixespointwise1[*OF reflect*]

lemma *fold-A-facet-im*: $\text{fold-A } z = z$
 ⟨proof⟩

lemma *fold-A-endo-X*: $\text{ChamberComplexEndomorphism } X \text{ fold-A}$
 ⟨proof⟩

lemma *res-fold-A-endo-A*: $\text{ChamberComplexEndomorphism } A \text{ res-fold-A}$
 ⟨proof⟩

lemmas *opp-res-fold-A-endo-A* =
ChamberComplexApartmentSystemTriangle.res-fold-A-endo-A[*OF reflect*]

lemma *fold-A-morph-A-A*: $\text{ChamberComplexMorphism } A \ A \ \text{fold-A}$
 ⟨proof⟩

lemmas *opp-fold-A-morph-A-A* =
ChamberComplexApartmentSystemTriangle.fold-A-morph-A-A[*OF reflect*]

lemma *res-fold-A-A-im-fold-A-A-im*: $\text{res-fold-A } \vdash A = \text{fold-A } \vdash A$
 ⟨proof⟩

lemmas *res-opp-fold-A-A-im-opp-fold-A-A-im* =
ChamberComplexApartmentSystemTriangle.res-fold-A-A-im-fold-A-A-im[
OF reflect
]

lemma *res-fold-A-C-A-im-fold-A-C-A-im*:
 $\text{res-fold-A } \vdash (\text{ChamberComplex.chamber-system } A) =$
 $\text{fold-A } \vdash (\text{ChamberComplex.chamber-system } A)$
 ⟨proof⟩

lemmas *res-opp-fold-A-C-A-im-opp-fold-A-C-A-im* =
ChamberComplexApartmentSystemTriangle.res-fold-A-C-A-im-fold-A-C-A-im[
OF reflect
]

lemma *chambercomplex-fold-A-im*: *ChamberComplex* (fold-A \vdash A)
 ⟨proof⟩

lemmas *chambercomplex-opp-fold-A-im* =
ChamberComplexApartmentSystemTriangle.chambercomplex-fold-A-im[
 OF reflect
]

lemma *chambersubcomplex-fold-A-im*:
ChamberComplex.ChamberSubcomplex A (fold-A \vdash A)
 ⟨proof⟩

lemmas *chambersubcomplex-opp-fold-A-im* =
ChamberComplexApartmentSystemTriangle.chambersubcomplex-fold-A-im[
 OF reflect
]

lemma *fold-A-facet-distance-map*:
chamber F \implies *face-distance* z (fold-A 'F) = *face-distance* z F
 ⟨proof⟩

lemma *fold-A-min-gallery-betw-map*:
assumes *chamber* F *chamber* G $z \subseteq F$
face-distance z G = *chamber-distance* F G *min-gallery* (F #Fs@[G])
shows *min-gallery* (fold-A |(F #Fs@[G]))
 ⟨proof⟩

lemma *fold-A-chamber-system-image-fixespointwise'*:
defines C-A : C-A \equiv *ChamberComplex.C* A
defines fC-A: fC-A \equiv {F \in C-A. *face-distance* z F = *chamber-distance* C F}
assumes F : F \in fC-A
shows *fixespointwise* fold-A F
 ⟨proof⟩

lemma *fold-A-chamber-system-image*:
defines C-A : C-A \equiv *ChamberComplex.C* A
defines fC-A: fC-A \equiv {F \in C-A. *face-distance* z F = *chamber-distance* C F}
shows fold-A \vdash C-A = fC-A
 ⟨proof⟩

lemmas *opp-fold-A-chamber-system-image* =
ChamberComplexApartmentSystemTriangle.fold-A-chamber-system-image[
 OF reflect
]

lemma *fold-A-chamber-system-image-fixespointwise*:
 F \in *ChamberComplex.C* A \implies *fixespointwise* fold-A (fold-A 'F)
 ⟨proof⟩

lemmas *fold-A-chsys-imfix* = *fold-A-chamber-system-image-fixespointwise*

lemmas *opp-fold-A-chamber-system-image-fixespointwise* =
 ChamberComplexApartmentSystemTriangle.fold-A-chsys-imfix[
 OF reflect
]

lemma *chamber-in-fold-A-im*:
 chamber F $\implies F \in \text{fold-A} \vdash A \implies F \in \text{fold-A} \vdash \text{ChamberComplex.C } A$
 ⟨*proof*⟩

lemmas *chamber-in-opp-fold-A-im* =
 ChamberComplexApartmentSystemTriangle.chamber-in-fold-A-im[*OF reflect*]

lemma *simplex-in-fold-A-im-image*:
 assumes $x \in \text{fold-A} \vdash A$
 shows $\text{fold-A } 'x = x$
 ⟨*proof*⟩

lemma *chamber1-notin-rfold-im*: $C \notin \text{opp-fold-A} \vdash A$
 ⟨*proof*⟩

lemma *fold-A-min-gallery-from1-map*:
 [[*chamber F*; $F \in \text{fold-A} \vdash A$; *min-gallery* ($C \# Fs @ [F]$)]] \implies
 min-gallery ($C \# \text{fold-A} \models Fs @ [F]$)
 ⟨*proof*⟩

lemma *fold-A-min-gallery-from2-map*:
 [[*chamber F*; $F \in \text{opp-fold-A} \vdash A$; *min-gallery* ($D \# Fs @ [F]$)]] \implies
 min-gallery ($C \# \text{fold-A} \models (Fs @ [F])$)
 ⟨*proof*⟩

lemma *fold-A-min-gallery-to2-map*:
 assumes *chamber F* $F \in \text{opp-fold-A} \vdash A$ *min-gallery* ($F \# Fs @ [D]$)
 shows *min-gallery* ($\text{fold-A} \models (F \# Fs) @ [C]$)
 ⟨*proof*⟩

lemmas *opp-fold-A-min-gallery-from1-map* =
 ChamberComplexApartmentSystemTriangle.fold-A-min-gallery-from2-map[
 OF reflect
]

lemmas *opp-fold-A-min-gallery-to1-map* =
 ChamberComplexApartmentSystemTriangle.fold-A-min-gallery-to2-map[
 OF reflect
]

lemma *closer-to-chamber1-not-in-rfold-im-chamber-system*:
 assumes *chamber-distance C F* \leq *chamber-distance D F*

shows $F \notin \text{ChamberComplex.C } (\text{opp-fold-}A \vdash A)$
<proof>

lemmas *clsrch1-nin-rfold-im-chsys* =
closer-to-chamber1-not-in-rfold-im-chamber-system

lemmas *closer-to-chamber2-not-in-fold-im-chamber-system* =
ChamberComplexApartmentSystemTriangle.clsrch1-nin-rfold-im-chsys[
OF reflect
]

lemma *fold-A-opp-fold-A-chamber-systems*:
ChamberComplex.C A =
 $(\text{ChamberComplex.C } (\text{fold-}A \vdash A)) \cup (\text{ChamberComplex.C } (\text{opp-fold-}A \vdash A))$
 $(\text{ChamberComplex.C } (\text{fold-}A \vdash A)) \cap (\text{ChamberComplex.C } (\text{opp-fold-}A \vdash A)) =$
 $\{\}$
<proof>

lemma *fold-A-im-min-gallery'*:
assumes *ChamberComplex.min-gallery* ($\text{fold-}A \vdash A$) ($C \# Cs$)
shows *ChamberComplex.min-gallery* A ($C \# Cs$)
<proof>

lemma *fold-A-im-min-gallery*:
ChamberComplex.min-gallery ($\text{fold-}A \vdash A$) ($C \# Cs$) \implies *min-gallery* ($C \# Cs$)
<proof>

lemma *fold-A-comp-fixespointwise*:
fixespointwise ($\text{fold-}A \circ \text{opp-fold-}A$) ($\bigcup (\text{fold-}A \vdash A)$)
<proof>

lemmas *opp-fold-A-comp-fixespointwise* =
ChamberComplexApartmentSystemTriangle.fold-A-comp-fixespointwise[*OF reflect*]

lemma *fold-A-fold*:
ChamberComplexIsomorphism ($\text{opp-fold-}A \vdash A$) ($\text{fold-}A \vdash A$) *fold-A*
<proof>

lemma *res-fold-A: ChamberComplexFolding A res-fold-A*
<proof>

lemmas *opp-res-fold-A* =
ChamberComplexApartmentSystemTriangle.res-fold-A[*OF reflect*]

end

6.2 Building locale and basic lemmas

Finally, we define a (thick) building to be a thick chamber complex with a system of apartments.

```

locale Building = ChamberComplexWithApartmentSystem X A
  for X :: 'a set set
  and A :: 'a set set set
+ assumes thick: ThickChamberComplex X
begin

```

```

abbreviation some-third-chamber ≡
  ThickChamberComplex.some-third-chamber X

```

```

lemmas some-third-chamberD-facet =
  ThickChamberComplex.some-third-chamberD-facet [OF thick]

```

```

lemmas some-third-chamberD-ne =
  ThickChamberComplex.some-third-chamberD-ne [OF thick]

```

```

lemmas chamber-some-third-chamber =
  ThickChamberComplex.chamber-some-third-chamber [OF thick]

```

end

6.3 Apartments are uniformly Coxeter

Using the assumption of thickness, we may use the special situation *ChamberComplexApartmentSystemTriangle* to verify that apartments have enough pairs of opposed foldings to ensure that they are isomorphic to a Coxeter complex. Since the apartments are all isomorphic, they are uniformly isomorphic to a single Coxeter complex.

```

context Building
begin

```

```

lemma apartments-have-many-foldings1:
  assumes A ∈ A chamber C chamber D C ~ D C ≠ D C ∈ A D ∈ A
  defines E ≡ some-third-chamber C D (C ∩ D)
  defines B ≡ supapartment C E
  and B' ≡ supapartment D E
  defines f ≡ restrict1 (canonical-retraction A D ∘ canonical-retraction B C)
    (∪ A)
  and g ≡ restrict1 (canonical-retraction A C ∘ canonical-retraction B' D)
    (∪ A)
  shows f'D = C ChamberComplexFolding A f
    g'C = D ChamberComplexFolding A g
  ⟨proof⟩

```

```

lemma apartments-have-many-foldings2:

```

assumes $A \in \mathcal{A}$ chamber C chamber D $C \sim D$ $C \neq D$ $C \in A$ $D \in A$
defines $E \equiv \text{some-third-chamber } C D (C \cap D)$
defines $B \equiv \text{supartment } C E$
and $B' \equiv \text{supartment } D E$
defines $f \equiv \text{restrict1 (canonical-retraction } A D \circ \text{canonical-retraction } B C)$
 $(\bigcup A)$
and $g \equiv \text{restrict1 (canonical-retraction } A C \circ \text{canonical-retraction } B' D)$
 $(\bigcup A)$
shows $\text{OpposedThinChamberComplexFoldings } A f g C$
 $\langle \text{proof} \rangle$

lemma *apartments-have-many-foldings3*:
assumes $A \in \mathcal{A}$ chamber C chamber D $C \sim D$ $C \neq D$ $C \in A$ $D \in A$
shows $\exists f g. \text{OpposedThinChamberComplexFoldings } A f g C \wedge D = g' C$
 $\langle \text{proof} \rangle$

lemma *apartments-have-many-foldings*:
assumes $A \in \mathcal{A}$ $C \in A$ chamber C
shows $\text{ThinChamberComplexManyFoldings } A C$
 $\langle \text{proof} \rangle$

theorem *apartments-are-coxeter*:
 $A \in \mathcal{A} \implies \exists S::'a \text{ permutation set. (}$
 $\text{CoxeterComplex } S \wedge$
 $(\exists \psi. \text{ChamberComplexIsomorphism } A (\text{CoxeterComplex.TheComplex } S) \psi)$
 $)$
 $\langle \text{proof} \rangle$

corollary *apartments-are-uniformly-coxeter*:
assumes $X \neq \{\}$
shows $\exists S::'a \text{ permutation set. CoxeterComplex } S \wedge$
 $(\forall A \in \mathcal{A}. \exists \psi.$
 $\text{ChamberComplexIsomorphism } A (\text{CoxeterComplex.TheComplex } S) \psi$
 $)$
 $\langle \text{proof} \rangle$

end
end

Bibliography

- [1] P. Abramenko and K. S. Brown. *Buildings: Theory and applications*, volume 248 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 2010.
- [2] P. Garrett. *Buildings and classical groups*. Chapman & Hall, London, 1997.
- [3] D. L. Johnson. *Presentations of groups*. Cambridge University Press, Cambridge, U.K, 2 edition, 1997.
- [4] J. Sylvestre. Representations of finite groups. *Archive of Formal Proofs*, Aug. 2015. https://www.isa-afp.org/entries/Rep_Fin_Groups.shtml, Formal proof development.