# Chamber complexes, Coxeter systems, and buildings 

Jeremy Sylvestre<br>University of Alberta, Augustana Campus<br>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 ..... 6
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 ..... 8
1.4.3 Injectivity, surjectivity, bijectivity, and inverses ..... 10
1.4.4 Induced functions on sets of sets and lists of sets ..... 12
1.4.5 Induced functions on quotients ..... 13
1.4.6 Support of a function ..... 14
1.5 Lists ..... 15
1.5.1 Miscellaneous facts ..... 15
1.5.2 Cases ..... 15
1.5.3 Induction ..... 17
1.5.4 Alternating lists ..... 18
1.5.5 Binary relation chains ..... 20
1.5.6 Set of subseqs ..... 23
1.6 Orders and posets ..... 24
1.6.1 Morphisms of posets ..... 24
1.6.2 More arg-min ..... 27
1.6.3 Bottom of a set ..... 28
1.6.4 Minimal and pseudominimal elements in sets ..... 29
1.6.5 Set of elements below another ..... 31
1.6.6 Lower bounds ..... 34
1.6.7 Simplex-like posets ..... 35
1.6.8 The superset ordering ..... 38
2 Algebra ..... 39
2.1 Miscellaneous algebra facts ..... 39
2.2 The type of permutations of a type ..... 40
2.3 Natural action of nat on types of class monoid-add ..... 42
2.3.1 Translation from class power. ..... 42
2.3.2 Additive order of an element ..... 43
2.4 Partial sums of a list ..... 45
2.5 Sums of alternating lists ..... 47
2.6 Conjugation in group-add ..... 48
2.6.1 Abbreviations and basic facts ..... 48
2.6.2 The conjugation sequence ..... 49
2.6.3 The action on signed group-add elements ..... 52
2.7 Cosets ..... 54
2.7.1 Basic facts ..... 54
2.7.2 The supset order on cosets ..... 55
2.7.3 The afforded partition ..... 55
2.8 Groups ..... 56
2.8.1 Locale definition and basic facts ..... 56
2.8.2 Sets with a suitable binary operation ..... 57
2.8.3 Cosets of a Group ..... 60
2.8.4 The Group generated by a set ..... 62
2.8.5 Homomorphisms and isomorphisms ..... 65
2.8.6 Normal subgroups ..... 67
2.8.7 Quotient groups ..... 71
2.8.8 The induced homomorphism on a quotient group ..... 74
2.9 Free groups ..... 75
2.9.1 Words in letters of signed type ..... 75
2.9.2 The collection of proper signed lists as a type ..... 78
2.9.3 Lifts of functions on the letter type ..... 81
2.9.4 Free groups on a set ..... 86
2.9.5 Group presentations ..... 90
2.10 Words over a generating set ..... 96
3 Simplicial complexes ..... 101
3.1 Geometric notions ..... 101
3.1.1 Facets ..... 101
3.1.2 Adjacency ..... 102
3.1.3 Chains of adjacent sets ..... 104
3.2 Locale and basic facts ..... 104
3.3 Chains of maximal simplices ..... 106
3.4 Isomorphisms of simplicial complexes ..... 111
3.5 The complex associated to a poset ..... 113
4 Chamber complexes ..... 116
4.1 Locale definition and basic facts ..... 116
4.2 The system of chambers and distance between chambers ..... 121
4.3 Labelling a chamber complex ..... 123
4.4 Morphisms of chamber complexes ..... 123
4.4.1 Morphism locale and basic facts ..... 124
4.4.2 Action on pregalleries and galleries ..... 127
4.4.3 Properties of the image ..... 128
4.4.4 Action on the chamber system ..... 129
4.4.5 Isomorphisms ..... 130
4.4.6 Endomorphisms ..... 135
4.4.7 Automorphisms ..... 139
4.4.8 Retractions ..... 141
4.4.9 Foldings of chamber complexes ..... 142
4.5 Thin chamber complexes ..... 145
4.5.1 Locales and basic facts ..... 146
4.5.2 The standard uniqueness argument for chamber mor- phisms of thin chamber complexes ..... 149
4.6 Foldings of thin chamber complexes ..... 152
4.6.1 Locale definition and basic facts ..... 152
4.6.2 Pairs of opposed foldings ..... 160
4.6.3 The automorphism induced by a pair of opposed foldings1 ..... 164
4.6.4 Walls ..... 174
4.7 Thin chamber complexes with many foldings ..... 187
4.7.1 Locale definition and basic facts ..... 187
4.7.2 The group of automorphisms ..... 190
4.7.3 Action of the group of automorphisms on the chamber system ..... 194
4.7.4 A labelling by the vertices of the fundamental chamber ..... 199
4.7.5 More on the action of the group of automorphisms on chambers ..... 207
4.7.6 A bijection between the fundamental chamber and the set of generating automorphisms ..... 208
4.8 Thick chamber complexes ..... 210
5 Coxeter systems and complexes ..... 211
5.1 Coxeter-like systems ..... 211
5.1.1 Locale definition and basic facts ..... 212
5.1.2 Special cosets ..... 213
5.1.3 Transfer from the free group over generators ..... 214
5.1.4 Words in generators containing alternating subwords ..... 218
5.1.5 Preliminary facts on the word problem ..... 225
5.1.6 Preliminary facts related to the deletion condition ..... 226
5.2 Coxeter-like systems with deletion ..... 229
5.2.1 Locale definition ..... 229
5.2.2 Consequences of the deletion condition ..... 229
5.2.3 The exchange condition ..... 230
5.2.4 More on words in generators containing alternating subwords ..... 231
5.2.5 The word problem ..... 233
5.2.6 Special subgroups and cosets ..... 236
5.3 Coxeter systems ..... 243
5.3.1 Locale definition and transfer from the associated free group ..... 243
5.3.2 The deletion condition is necessary ..... 243
5.3.3 The deletion condition is sufficient ..... 246
5.3.4 The Coxeter system associated to a thin chamber com- plex with many foldings ..... 249
5.4 Coxeter complexes ..... 256
5.4.1 Locale and complex definitions ..... 256
5.4.2 As a simplicial complex ..... 256
5.4.3 As a chamber complex ..... 260
5.4.4 The Coxeter complex associated to a thin chamber complex with many foldings ..... 263
6 Buildings ..... 271
6.1 Apartment systems ..... 271
6.1.1 Locale and basic facts ..... 271
6.1.2 Isomorphisms between apartments ..... 273
6.1.3 Retractions onto apartments ..... 276
6.1.4 Distances in apartments ..... 279
6.1.5 Special situation: a triangle of apartments and chambers ..... 282
6.2 Building locale and basic lemmas ..... 300
6.3 Apartments are uniformly Coxeter ..... 300

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, statments, 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 01 SucSuc]:
    assumes \(\quad 0: n=0 \Longrightarrow P\)
    and \(\quad 1: n=1 \Longrightarrow P\)
    and \(\quad\) SucSuc: \(\bigwedge m . n=\operatorname{Suc}(\) Suc \(m) \Longrightarrow P\)
    shows \(P\)
proof (cases \(n\) )
    case (Suc m) with 1 SucSuc show ?thesis by (cases m) auto
qed (simp add: 0)
lemma nat-even-induct [case-names - 0 SucSuc]:
    assumes even: even \(n\)
    and \(\quad 0: P 0\)
    and \(\quad\) SucSuc: \(\bigwedge m\). even \(m \Longrightarrow P m \Longrightarrow P(\) Suc \((\) Suc \(m))\)
    shows \(P n\)
proof-
    from assms obtain \(k\) where \(n=2 * k\) using evenE by auto
    moreover from assms have \(P(2 * k)\) by (induct \(k\) ) auto
    ultimately show ?thesis by fast
qed
lemma nat-induct-step2 [case-names 01 SucSuc]:
    assumes 0: P 0
    and \(\quad 1: P 1\)
    and \(\quad S u c S u c: \bigwedge m . P m \Longrightarrow P(\) Suc \((\) Suc \(m))\)
    shows \(P n\)
proof (cases even \(n\) )
    case True
```

from this obtain $k$ where $n=2 * k$ using evenE by auto moreover have $P(2 * k)$ using 0 SucSuc by (induct $k$ ) auto ultimately show ?thesis by fast

## next

case False
from this obtain $k$ where $n=2 * k+1$ using oddE by blast
moreover have $P(2 * k+1)$ using 1 SucSuc by (induct $k$ ) auto
ultimately show ?thesis by fast
qed

### 1.2 Logic

lemma ex1-unique: $\exists$ ! $x . P x \Longrightarrow P a \Longrightarrow P b \Longrightarrow a=b$
by blast
lemma not-the1:
assumes $\exists!x . P x y \neq($ THE $x . P x)$
shows $\neg P y$
using assms(2) the1-equality[OF assms(1)]
by auto
lemma two-cases [case-names both one other neither]:
assumes both $: P \Longrightarrow Q \Longrightarrow R$
and one $: P \Longrightarrow \neg Q \Longrightarrow R$
and other : $\neg P \Longrightarrow Q \Longrightarrow R$
and $\quad$ neither $: \neg P \Longrightarrow \neg Q \Longrightarrow R$
shows $R$
using assms
by fast

### 1.3 Sets

lemma bex1-equality: $\llbracket \exists!x \in A . P x ; x \in A ; P x ; y \in A ; P y \rrbracket \Longrightarrow x=y$ by blast
lemma prod-ballI: $(\bigwedge a b .(a, b) \in A \Longrightarrow P a b) \Longrightarrow \forall(a, b) \in A . P a b$ by fast
lemmas seteq $I=$ set-eqI[OF iffI]
lemma set-decomp-subset:

$$
\llbracket U=A \cup B ; A \subseteq X ; B \subseteq Y ; X \subseteq U ; X \cap Y=\{ \} \rrbracket \Longrightarrow A=X
$$

by auto
lemma insert-subset-equality: $\llbracket a \notin A ; a \notin B ;$ insert a $A=$ insert a $B \rrbracket \Longrightarrow A=B$ by auto
lemma insert-compare-element: $a \notin A \Longrightarrow$ insert $b A=$ insert $a A \Longrightarrow b=a$ by auto

## lemma card1:

assumes card $A=1$
shows $\exists a$. $A=\{a\}$
proof-
from assms obtain $a$ where $a: a \in A$ by fastforce
with assms show ?thesis using card-ge-0-finite[of A] card-subset-eq[of $A$ \{a\}]
by auto
qed
lemma singleton-pow: $a \in A \Longrightarrow\{a\} \in$ Pow $A$
using Pow-mono Pow-top by fast
definition separated-by :: 'a set set $\Rightarrow^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow$ 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: $x \in A \Longrightarrow y \in B \Longrightarrow$ separated-by $\{A, B\} x y$ using separated-by-def by fastforce
lemma separated-by-disjoint: 【 separated-by $\{A, B\} x y ; A \cap B=\{ \} ; x \in A \rrbracket \Longrightarrow y \in B$ unfolding separated-by-def by fast
lemma separated-by-in-other: separated-by $\{A, B\} x y \Longrightarrow x \notin A \Longrightarrow x \in B \wedge y \in A$ unfolding separated-by-def by auto
lemma separated-by-not-empty: separated-by wxy $\Longrightarrow w \neq\{ \}$
unfolding separated-by-def by fast
lemma not-self-separated-by-disjoint: $A \cap B=\{ \} \Longrightarrow \neg$ separated-by $\{A, B\} x x$ unfolding separated-by-def by auto

### 1.4 Functions and relations

### 1.4.1 Miscellaneous

lemma cong-let: $($ let $x=y$ in $f x)=f y$ by simp
lemma sym-sym: sym $(A \times A)$ by (fast intro: symI)
lemma trans-sym: trans $(A \times A)$ by (fast intro: transI)
lemma map-prod-sym: sym $A \Longrightarrow$ sym (map-prod $f f$ ' $A$ )
using $\operatorname{sym} D[$ of $A]$ map-prod-def by (fast intro: symI)
abbreviation restrict1 $::\left({ }^{\prime} a \Rightarrow^{\prime} a\right) \Rightarrow{ }^{\prime} a$ set $\Rightarrow\left({ }^{\prime} a \Rightarrow^{\prime} a\right)$ where restrict1 $f A \equiv(\lambda a$. if $a \in A$ then $f$ a else $a)$
lemma restrict1-image: $B \subseteq A \Longrightarrow$ restrict1 $f A{ }^{\prime} B=f^{`} B$
by auto

### 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: ( $\bigwedge a . a \in A \Longrightarrow f a=g a) \Longrightarrow$ fun-eq-on $f g A$ using fun-eq-on-def by fast
lemma fun-eq-onD: fun-eq-on $f g A \Longrightarrow a \in A \Longrightarrow f a=g a$ using fun-eq-on-def by fast
lemma fun-eq-on-UNIV: (fun-eq-on $f g$ UNIV $)=(f=g)$ unfolding fun-eq-on-def by fast
lemma fun-eq-on-subset: fun-eq-on $f g A \Longrightarrow B \subseteq A \Longrightarrow$ fun-eq-on $f g B$ unfolding fun-eq-on-def by fast
lemma fun-eq-on-sym: fun-eq-on $f g A \Longrightarrow f u n-e q-o n g f A$ using fun-eq-onD by (fastforce intro: fun-eq-onI)
lemma fun-eq-on-trans: fun-eq-on $f g A$ fun-eq-on $g h A \Longrightarrow$ fun-eq-on $f h A$ using fun-eq-onD fun-eq-onD by (fastforce intro: fun-eq-onI)
lemma fun-eq-on-cong: fun-eq-on $f h A \Longrightarrow$ fun-eq-on $g h A$ fun-eq-on $f g A$ using fun-eq-on-trans fun-eq-on-sym by fastforce
lemma fun-eq-on-im: fun-eq-on $f g A \Longrightarrow B \subseteq A \Longrightarrow f^{\iota} B=g^{`} B$
using fun-eq-onD by force
lemma fun-eq-on-subset-and-diff-imp-eq-on:
assumes $A \subseteq B$ fun-eq-on $f g A$ fun-eq-on $f g(B-A)$
shows fun-eq-on $f g B$
proof (rule fun-eq-onI)
fix $x$ assume $x \in B$ with $\operatorname{assms}(1)$ show $f x=g x$ using fun-eq-onD[OF assms(2)] fun-eq-onD[OF assms(3)] by (cases $x \in A$ ) auto
qed
lemma fun-eq-on-set-and-comp-imp-eq: fun-eq-on $f g A \Longrightarrow$ fun-eq-on $f g(-A) \Longrightarrow f=g$ using fun-eq-on-subset-and-diff-imp-eq-on[of A UNIV] by (simp add: Compl-eq-Diff-UNIV fun-eq-on-UNIV)
lemma fun-eq-on-bij-betw: fun-eq-on $f g A \Longrightarrow$ bij-betw $f A B=$ bij-betw $g A B$ using bij-betw-cong unfolding fun-eq-on-def by fast
lemma fun-eq-on-restrict1: fun-eq-on (restrict1 $f A) f A$ by (auto intro: fun-eq-onI)
abbreviation fixespointwise $f A \equiv$ fun-eq-on $f$ id $A$

```
lemmas fixespointwiseI = fun-eq-onI [of --id]
lemmas fixespointwiseD = fun-eq-onD [\begin{array}{ll}{0f}&{-id}\end{array}]
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 =
    fun-eq-on-subset-and-diff-imp-eq-on[of-- id]
lemma id-fixespointwise: fixespointwise id A
    using fun-eq-on-def by fast
lemma fixespointwise-im: fixespointwise }fA\LongrightarrowB\subseteqA\Longrightarrow\mp@subsup{f}{}{`}B=
    by (auto simp add: fun-eq-on-im)
lemma fixespointwise-comp
    fixespointwise f A\Longrightarrow fixespointwise g A \Longrightarrow fixespointwise (g\circf) A
    unfolding fun-eq-on-def by simp
lemma fixespointwise-insert:
    assumes fixespointwise f A f'(insert a A) = insert a A
    shows fixespointwise f (insert a A)
    using assms(2) insert-compare-element[of a A f a]
            fixespointwiseD[OF assms(1)] fixespointwise-im[OF assms(1)]
    by (cases a\inA) (auto intro: fixespointwiseI)
lemma fixespointwise-restrict1:
    fixespointwise f A \Longrightarrow fixespointwise (restrict1 f B) A
    using fixespointwiseD[of f] by (auto intro: fixespointwiseI)
lemma fold-fixespointwise:
    \forallx\inset xs. fixespointwise ( }fx\mathrm{ ) A ב fixespointwise (fold f xs) A
proof (induct xs)
    case Nil show ?case using id-fixespointwise subst[of id] by fastforce
next
    case (Cons x xs)
    hence fixespointwise (fold f xs of x) A
        using fixespointwise-comp[of f x A fold f xs] by fastforce
    moreover have fold f xs \circf }x=\mathrm{ fold f (x#xs) by simp
    ultimately show ?case using subst[of - - \lambdaf. fixespointwise f A] by fast
qed
lemma funpower-fixespointwise:
    assumes fixespointwise f A
    shows fixespointwise (f~n)A
proof (induct n)
    case 0 show ?case using id-fixespointwise subst[of id] by fastforce
next
    case (Suc m)
```

with assms have fixespointwise $(f \circ(f \wedge m)) A$ using fixespointwise-comp by fast
moreover have $f \circ\left(f^{\wedge} m\right)=f^{\wedge \sim}($ Suc m) by simp
ultimately show ? case using subst $[$ of $-\lambda f$. fixespointwise $f A]$ by fast qed

### 1.4.3 Injectivity, surjectivity, bijectivity, and inverses

lemma inj-on-to-singleton:
assumes inj-on $f A f^{\prime} A=\{b\}$
shows $\exists a$. $A=\{a\}$
proof-
from assms(2) obtain $a$ where $a: a \in A f a=b$ by force
with assms have $A=\{a\}$ using inj-on $D[$ of $f A]$ by blast
thus ?thesis by fast
qed
lemmas $i n j-i n j-o n=$ subset-inj-on $[o f-U N I V, O F-s u b s e t-U N I V]$
lemma inj-on-eq-image': $\llbracket i n j$-on $f A ; X \subseteq A ; Y \subseteq A ; f^{‘} X \subseteq f^{\prime} Y \rrbracket \Longrightarrow X \subseteq Y$
unfolding inj-on-def by fast
lemma inj-on-eq-image: $\llbracket i n j$-on $f A ; X \subseteq A ; Y \subseteq A ; f^{\prime} X=f^{‘} Y \rrbracket \Longrightarrow X=Y$
using inj-on-eq-image' $[$ of $f A X Y]$ inj-on-eq-image' $[$ of $f A Y X]$ by simp
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 ((')f) (Pow A)
using inj-onD[OF assms] inj-onI[of Pow $\left.A\left({ }^{\circ}\right) f\right]$
by blast
lemma inj-on-minus-set: inj-on $((-) A)$ (Pow $A)$
by (fast intro: inj-onI)
lemma induced-pow-fun-surj:
$((\cdot) f) \cdot($ Pow $A)=\operatorname{Pow}\left(f^{\prime} A\right)$
proof (rule seteqI)
fix $X$ show $X \in\left(\left({ }^{\prime} f\right)\right.$ ' $($ Pow $A) \Longrightarrow X \in \operatorname{Pow}\left(f^{`} A\right)$ by fast
next
fix $Y$ assume $Y: Y \in \operatorname{Pow}\left(f^{\star} A\right)$
moreover hence $Y=f^{\prime}\{a \in A . f a \in Y\}$ by fast
ultimately show $Y \in\left(\left({ }^{\circ}\right) f\right)^{\prime}($ Pow $A)$ by auto
qed
lemma bij-betw-f-the-inv-into-f:
bij-betw $f A B \Longrightarrow y \in B \Longrightarrow f$ (the-inv-into $A f y)=y$

- an equivalent lemma appears in the HOL library, but this version avoids the

```
double bij-betw premises
    unfolding bij-betw-def by (blast intro: f-the-inv-into-f)
lemma bij-betw-the-inv-into-onto: bij-betw f A B\Longrightarrow the-inv-into A f`}B=
    unfolding bij-betw-def by force
lemma bij-betw-imp-bij-betw-Pow:
    assumes bij-betw f A B
    shows bij-betw ((`)f) (Pow A) (Pow B)
    unfolding bij-betw-def
proof (rule conjI, rule inj-onI)
    show \x y.\llbracketx\inPow A; y\inPow A; f`}x=\mp@subsup{f}{}{\prime}y\rrbracket\Longrightarrowx=
        using inj-onD[OF bij-betw-imp-inj-on, OF assms] by blast
    show (') f' Pow }A=\mathrm{ Pow }
    proof
        show (') f' Pow A\subseteqPow B using bij-betw-imp-surj-on[OF assms] by fast
        show (') f' Pow A}\supseteq Pow 
        proof
            fix }y\mathrm{ assume y:}y\in\operatorname{Pow}
            with assms have }y=f\mathrm{ ' the-inv-into A f'}
                using bij-betw-f-the-inv-into-f[THEN sym] by fastforce
            moreover from y assms have the-inv-into A f' y\subseteqA
            using bij-betw-the-inv-into-onto by fastforce
                ultimately show }y\in(`)f'Pow A by aut
        qed
    qed
qed
lemma comps-fixpointwise-imp-bij-betw:
    assumes f}\mp@subsup{f}{}{\prime}X\subseteqY g'Y\subseteqX fixespointwise (g\circf) X fixespointwise ( f\circg) Y
    shows bij-betw f X Y
    unfolding bij-betw-def
proof
    show inj-on f X
    proof (rule inj-onI)
        fix x y show \llbracket x\inX;y\inX;fx=fy\rrbracket\Longrightarrowx=y
        using fixespointwiseD[OF assms(3), of x] fixespointwiseD[OF assms(3), of y]
            by simp
    qed
    from assms(1,2) show f`X = Y using fixespointwiseD[OF assms(4)] by force
qed
lemma set-permutation-bij-restrict1:
    assumes bij-betwf A A
    shows bij (restrict1 f A)
proof (rule bijI)
    have bij-f: inj-on f A f}A=A using iffD1[OF bij-betw-def, OF assms] by aut
    show inj (restrict1 f A)
    proof (rule injI)
```

```
    fix xy show restrict1 f A x = restrict1 f A y \Longrightarrowx=y
        using inj-onD bij-f by (cases }x\inA\quady\inA rule: two-cases) aut
    qed
    show surj (restrict1 f A)
    proof (rule surjI)
    fix }
    define y where y = restrict1 (the-inv-into A f) A x
    thus restrict1 f A y =x
        using the-inv-into-into[of f] bij-f f-the-inv-into-f[of f] by (cases x\inA) auto
    qed
qed
lemma set-permutation-the-inv-restrict1:
    assumes bij-betw f A A
    shows the-inv (restrict1fA)}=\mathrm{ restrict1 (the-inv-into Af) A
proof (rule ext, rule the-inv-into-f-eq)
    from assms show inj (restrict1 f A)
        using bij-is-inj set-permutation-bij-restrict1 by fast
next
    fix a from assms show restrict1 f A (restrict1 (the-inv-into A f) A a)=a
        using bij-betw-def[of f] by (simp add: the-inv-into-into f-the-inv-into-f)
qed simp
lemma the-inv-into-the-inv-into:
    inj-on f A \Longrightarrowa\inA \Longrightarrow the-inv-into (f`A) (the-inv-into A f) a=fa
    using inj-on-the-inv-into by (force intro: the-inv-into-f-eq imageI)
lemma the-inv-into-f-im-f-im:
    assumes inj-on f A x\subseteqA
    shows the-inv-into }A\mp@subsup{f}{}{\prime}f\mp@subsup{f}{}{\prime}x=
    using assms(2) the-inv-into-f-f[OF assms(1)]
    by force
lemma f-im-the-inv-into-f-im:
    assumes inj-on f A x\subseteqf`A
    shows f'the-inv-into A f'}x=
    using assms(2) f-the-inv-into-f[OF assms(1)]
    by force
lemma the-inv-leftinv: bij f\Longrightarrow the-inv f\circf=id
    using bij-def[of f] the-inv-f-f by fastforce
```


### 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 \(::\left({ }^{\prime} a \Rightarrow{ }^{\prime} b\right) \Rightarrow{ }^{\prime} a\) set set \(\Rightarrow{ }^{\prime} b\) set set (infix \(\left.\vdash 70\right)\)
    where \(f \vdash X \equiv\left(\left({ }^{`}\right) f\right)^{\prime} X\)
```

```
abbreviation setlistmapim :: (' }a\mp@subsup{=}{}{\prime}b)=>\mp@subsup{)}{}{\prime}a\mathrm{ set list }=>\mp@subsup{'}{}{\prime}b\mathrm{ set list (infix }\models70
    where f}=Xs\equiv\operatorname{map}((`)f)X
lemma setsetmapim-comp: (f\circg)\vdashA=f\vdash(g\vdashA)
    by (auto simp add: image-comp)
lemma setlistmapim-comp: (f\circg)\modelsxs=f\models(g|=xs)
    by auto
lemma setsetmapim-cong-subset:
    assumes fun-eq-on g f (UA) B\subseteqA
    shows }\quadg\vdashB\subseteqf\vdash
proof
    fix y assume }y\ing\vdash
    from this obtain }x\mathrm{ where }x\inBy=\mp@subsup{g}{}{\prime}x\mathrm{ by fast
    with assms(2) show y f f\vdashB using fun-eq-on-im[OF assms(1), of x] by fast
qed
lemma setsetmapim-cong:
    assumes fun-eq-on g f (UA) B\subseteqA
    shows }\quadg\vdashB=f\vdash
    using setsetmapim-cong-subset[OF assms]
        setsetmapim-cong-subset[OF fun-eq-on-sym, OF assms]
    by fast
lemma setsetmapim-restrict1: B\subseteqA\Longrightarrow restrict1 f (UA)\vdashB=f\vdashB
    using setsetmapim-cong[of-f] fun-eq-on-restrict1[of \bigcupA f] by simp
lemma setsetmapim-the-inv-into:
    assumes inj-on f (UA)
    shows (the-inv-into (\bigcupA)f)\vdash(f\vdashA)=A
proof (rule seteqI)
    fix }x\mathrm{ assume }x\in(\mathrm{ the-inv-into (\A)f)}\vdash(f\vdashA
    from this obtain y where y:y\inf\vdashAx=the-inv-into (\bigcupA)f' y by auto
    from }y(1)\mathrm{ obtain }z\mathrm{ where z: zGA y= f`z by fast
    moreover from z(1) have the-inv-into (UA) f'f'}z=
        using the-inv-into-f-f[OF assms] by force
    ultimately show }x\inA\mathrm{ using y(2) the-inv-into-f-im-f-im[OF assms] by simp
next
    fix }x\mathrm{ assume }x:x\in
    moreover hence the-inv-into (UA) f'f'}x=
        using the-inv-into-f-im-f-im[OF assms, of x] by fast
    ultimately show }x\in(\mathrm{ the-inv-into ( }\bigcupA)f)\vdash(f\vdashA) by aut
qed
```


### 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 respects r\Longrightarrowy\inf`r"`{a}\Longrightarrowy=fa
    using congruentD[of rf] by auto
lemma ex1-class-image:
    assumes refl-on A r f respects r X\inA//r
    shows }\exists!b.b\in\mp@subsup{f}{}{\prime}
proof-
    from assms(3) obtain a where a: a\inA X=r"{a} by (auto intro: quotientE)
    thus ?thesis
        using refl-onD[OF assms(1)] ex1I[of - f a]
            respects-imp-unique-image-rel[OF assms(2), of - a]
        by force
qed
definition quotientfun :: (' }a>>'b)=>\mp@subsup{}{}{\prime}a\mathrm{ set }=>\mp@subsup{}{}{\prime}
    where quotientfun f X = (THE b.b\inf`X)
lemma quotientfun-equality:
    assumes refl-on A rf respects r X\inA//r b\inf`X
    shows quotientfun f X=b
    unfolding quotientfun-def
    using assms(4) ex1-class-image[OF assms(1-3)]
    by (auto intro: the1-equality)
lemma quotientfun-classrep-equality:
    \llbracketrefl-on A r; f respects r;a\inA\rrbracket\Longrightarrow quotientfun f (r``{a})=fa
    using refl-onD by (fastforce intro: quotientfun-equality quotientI)
```


### 1.4.6 Support of a function

```
definition supp \(::\left({ }^{\prime} a \Rightarrow\right.\) 'b::zero \() \Rightarrow^{\prime} a\) set where supp \(f=\{x . f x \neq 0\}\)
lemma suppI-contra: \(x \notin \operatorname{supp} f \Longrightarrow f x=0\)
using supp-def by fast
lemma suppD-contra: \(f x=0 \Longrightarrow x \notin\) supp \(f\)
using supp-def by fast
abbreviation restrict0 \(::\left({ }^{\prime} a \Rightarrow{ }^{\prime} b::\right.\) zero \() \Rightarrow^{\prime} a\) set \(\Rightarrow\left({ }^{\prime} a \Rightarrow{ }^{\prime} b\right)\)
where restrict0 f \(A \equiv(\lambda a\). if \(a \in A\) then \(f\) a else 0\()\)
lemma supp-restrict0 : supp (restrict0 f \(A\) ) \(\subseteq A\)
proof-
have \(\bigwedge a . a \notin A \Longrightarrow a \notin \operatorname{supp}(\) restrict0 \(f A)\)
using suppD-contra[of restrict0 f \(A\) ] by simp
thus ?thesis by fast
qed
```


### 1.5 Lists

### 1.5.1 Miscellaneous facts

lemma snoc-conv-cons: $\exists x x s . y s @[y]=x \# x s$
by (cases ys) auto
lemma cons-conv-snoc: $\exists$ ys $y . x \# x s=y s @[y]$
by (cases xs rule: rev-cases) auto
lemma distinct-count-list:
distinct xs $\Longrightarrow$ count-list xs $a=($ if $a \in$ set xs then 1 else 0$)$
by (induct xs) auto
lemma map-fst-map-const-snd: map fst (map $(\lambda s .(s, b)) x s)=x s$ by (induct $x s$ ) auto
lemma inj-on-distinct-setlistmapim:
assumes inj-on $f A$
shows $\forall X \in$ set $X s . X \subseteq A \Longrightarrow \operatorname{distinct} X s \Longrightarrow \operatorname{distinct}(f \mid=X s)$
proof (induct Xs)
case (Cons X Xs)
show ?case
proof (cases f' $X \in \operatorname{set}(f \models X s))$
case True
from this obtain $Y$ where $Y: Y \in$ set $X s f^{\prime} X=f^{\prime} Y$ by auto
with assms $Y(1) \operatorname{Cons}(2,3)$ show ?thesis
using inj-on-eq-image $[$ of $f A X Y$ by fastforce
next
case False with Cons show ?thesis by simp
qed
qed $\operatorname{simp}$

### 1.5.2 Cases

lemma list-cases-Cons-snoc [case-names Nil Single Cons-snoc]: assumes $\quad$ Nil: $x s=[] \Longrightarrow P$
and $\quad$ Single: $\bigwedge x . x s=[x] \Longrightarrow P$
and Cons-snoc: $\lfloor x y s y . x s=x \# y s @[y] \Longrightarrow P$
shows $P$
proof (cases xs, rule Nil)
case (Cons $x$ xs) with Single Cons-snoc show ?thesis
by (cases xs rule: rev-cases) auto
qed
lemma two-lists-cases-Cons-Cons [case-names Nil1 Nil2 ConsCons]:
assumes Nil1: \ys. as $=[] \Longrightarrow b s=y s \Longrightarrow P$
and $\quad$ Nil2: $\bigwedge x s . a s=x s \Longrightarrow b s=[] \Longrightarrow P$
and ConsCons: $\lfloor x$ xs y ys. as $=x \# x s \Longrightarrow b s=y \# y s \Longrightarrow P$
shows $P$
proof (cases as)
case Cons with $\operatorname{assms}(2,3)$ show ?thesis by (cases bs) auto qed (simp add: Nil1)
lemma two-lists-cases-snoc-Cons [case-names Nil1 Nil2 snoc-Cons]:
assumes $\quad$ Nil1: $\bigwedge y s . a s=[] \Longrightarrow b s=y s \Longrightarrow P$
and $\quad$ Nil2: $\bigwedge x s . a s=x s \Longrightarrow b s=[] \Longrightarrow P$
and $\quad$ snoc-Cons: $\bigwedge x s x y y s . a s=x s @[x] \Longrightarrow b s=y \# y s \Longrightarrow P$
shows $P$
proof (cases as rule: rev-cases)
case snoc with Nil2 snoc-Cons show ?thesis by (cases bs) auto
qed (simp add: Nil1)
lemma two-lists-cases-snoc-Cons' [case-names both-Nil Nil1 Nil2 snoc-Cons]:
assumes both-Nil: as $=[] \Longrightarrow b s=[] \Longrightarrow P$
and $\quad$ Nil1: $\bigwedge y$ ys. as $=[] \Longrightarrow b s=y \# y s \Longrightarrow P$
and $\quad$ Nil2: $\bigwedge x s$ x. as $=x s @[x] \Longrightarrow b s=[] \Longrightarrow P$
and $\quad$ snoc-Cons: $\bigwedge x s x$ y ys. as $=x s @[x] \Longrightarrow b s=y \# y s \Longrightarrow P$ shows $P$
proof (cases as bs rule: two-lists-cases-snoc-Cons)
case (Nil1 ys) with $\operatorname{assms}(1,2)$ show $P$ by (cases ys) auto
next
case (Nil2 xs) with assms ( 1,3 ) show $P$ by (cases xs rule: rev-cases) auto qed (rule snoc-Cons)
lemma two-prod-lists-cases-snoc-Cons:
assumes $\bigwedge x s . a s=x s \Longrightarrow b s=[] \Longrightarrow P \bigwedge y s . a s=[] \Longrightarrow b s=y s \Longrightarrow P$
$\wedge x s ~ a a b a b b b b y s . a s=x s @[(a a, b a)] \wedge b s=(a b, b b) \# y s \Longrightarrow P$
shows $P$
proof (rule two-lists-cases-snoc-Cons)
from assms
show \ys. as $=[] \Longrightarrow b s=y s \Longrightarrow P \bigwedge x s . a s=x s \Longrightarrow b s=[] \Longrightarrow P$
by auto
from $\operatorname{assms}(3)$ show $\bigwedge x s x y$ ys. as $=x s @[x] \Longrightarrow b s=y \# y s \Longrightarrow P$ by fast
qed
lemma three-lists-cases-snoc-mid-Cons
[case-names Nil1 Nil2 Nil3 snoc-single-Cons snoc-mid-Cons]:
assumes Nil1: ^ys zs. as $=[] \Longrightarrow b s=y s \Longrightarrow c s=z s \Longrightarrow P$
and $\quad$ Nil2: $\bigwedge x s$ zs. $a s=x s \Longrightarrow b s=[] \Longrightarrow c s=z s \Longrightarrow P$
and $\quad$ Nil3: $\bigwedge x s$ ys. $a s=x s \Longrightarrow b s=y s \Longrightarrow c s=[] \Longrightarrow P$
and snoc-single-Cons:
$\bigwedge x s x y z z s . a s=x s @[x] \Longrightarrow b s=[y] \Longrightarrow c s=z \# z s \Longrightarrow P$
and snoc-mid-Cons:
$\bigwedge x s x w y s y z z s . a s=x s @[x] \Longrightarrow b s=w \# y s @[y] \Longrightarrow$

$$
c s=z \# z s \Longrightarrow P
$$

shows $P$
proof (cases as cs rule: two-lists-cases-snoc-Cons)

```
    case Nil1 with assms(1) show P by simp
next
    case NilQ with assms(3) show P by simp
next
    case snoc-Cons
    with Nil2 snoc-single-Cons snoc-mid-Cons show P
        by (cases bs rule: list-cases-Cons-snoc) auto
qed
```


### 1.5.3 Induction

lemma list-induct-CCons [case-names Nil Single CCons]: assumes Nil : P []
and Single: $\bigwedge x . P[x]$
and CCons : $\bigwedge x y x s . P(y \# x s) \Longrightarrow P(x \# y \# x s)$
shows $P x s$
proof (induct $x s$ )
case (Cons $x$ xs) with Single CCons show ?case by (cases xs) auto
qed (rule Nil)
lemma list-induct-ssnoc [case-names Nil Single ssnoc]:
assumes Nil : P []
and Single: $\bigwedge x . P[x]$
and $\quad$ ssnoc : $\bigwedge x s x y . P(x s @[x]) \Longrightarrow P(x s @[x, y])$
shows $P x s$
proof (induct xs rule: rev-induct)
case (snoc $x$ xs) with Single ssnoc show ?case by (cases xs rule: rev-cases) auto qed (rule Nil)
lemma list-induct2-snoc [case-names Nil1 Nil2 snoc]:
assumes Nil1: \ys. P [] ys
and Nil2: $\bigwedge x s . P x s[]$
and $\quad$ snoc: $\bigwedge x s x$ ys y. P xs ys $\Longrightarrow P(x s @[x])(y s @[y])$ shows $P x s y s$
proof (induct xs arbitrary: ys rule: rev-induct, rule Nil1)
case (snoc bs) with assms(2,3) show ?case by (cases ys rule: rev-cases) auto
qed
lemma list-induct2-snoc-Cons [case-names Nil1 Nil2 snoc-Cons]:
assumes Nil1 : \ys. P [] ys
and Nil2 : $\bigwedge x s . P x s[]$
and snoc-Cons: $\bigwedge x s x y y s . P x s$ ys $\Longrightarrow P(x s @[x])(y \# y s)$
shows $P$ xs ys
proof (induct ys arbitrary: xs, rule Nil2)
case (Cons y ys) with Nil1 snoc-Cons show ?case
by (cases xs rule: rev-cases) auto
qed
lemma prod-list-induct3-snoc-Conssnoc-Cons-pairwise:

```
    assumes \ys zs.Q ([],ys,zs) \xszs.Q (xs,[],zs) \bigwedgexs ys.Q (xs,ys,[])
    \xs xyzzs.Q(xs@[x],[y],z#zs)
    and step:
    \s x y ys wzzs.Q (xs,ys,zs)\LongrightarrowQ(xs,ys@[w],z#zs)\Longrightarrow
        Q(xs@[x],y#ys,zs)\LongrightarrowQ(xs@[x],y#ys@[w],z#zs)
    shows Qt
proof (
    induct t
    taking: }\lambda(xs,ys,zs). length xs + length ys + length zs
    rule : measure-induct-rule
)
    case (less t)
    show ?case
    proof (cases t)
    case (fields xs ys zs) from assms less fields show ?thesis
        by (cases xs ys zs rule: three-lists-cases-snoc-mid-Cons) auto
    qed
qed
lemma list-induct3-snoc-Conssnoc-Cons-pairwise
    [case-names Nil1 Nil2 Nil3 snoc-single-Cons snoc-Conssnoc-Cons]:
    assumes Nil1: \ys zs. P [] ys zs
    and Nil2 : \xs zs.P xs [] zs
    and Nil3 : \xs ys. P xs ys []
    and snoc-single-Cons : \xs x y z zs. P (xs@[x]) [y] (z#zs)
    and snoc-Conssnoc-Cons:
    \ss x y ys wzzs. P xs ys zs \LongrightarrowP xs (ys@[w])(z#zs)\Longrightarrow
        P(xs@[x])(y#ys)zs\LongrightarrowP(xs@[x])(y#ys@[w])(z#zs)
    shows P xs ys zs
    using assms
        prod-list-induct3-snoc-Conssnoc-Cons-pairwise[of \lambda(xs,ys,zs). P xs ys zs]
    by auto
```


### 1.5.4 Alternating lists

primrec alternating-list :: nat $\Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow$ ' $a$ list
where zero: alternating-list 0 st $=[]$
| Suc : alternating-list (Suc k) st=
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 $s t=[s, t]$
using arg-cong $[O F$ Suc-1, THEN sym, of $\lambda n$. alternating-list $n s t]$ by simp
lemma length-alternating-list: length (alternating-list $n$ st) $=n$
by (induct $n$ ) auto
lemma alternating-list-Suc-Cons:

```
    alternating-list (Suc k) st=s # alternating-list kts
    by (induct k) auto
lemma alternating-list-SucSuc-ConsCons:
    alternating-list (Suc (Suc k)) st=s#t# alternating-list k s t
    using alternating-list-Suc-Cons[of Suc k s] alternating-list-Suc-Cons[of k t]
    by simp
lemma alternating-list-alternates:
    alternating-list n st=as@[a,b,c]@bs\Longrightarrowa=c
proof (induct n arbitrary:bs)
    case (Suc m) hence prevcase:
    \xs.alternating-list m s t=as@ [a,b,c]@ @s \Longrightarrowa=c
    alternating-list (Suc m) st=as@ [a,b,c]@ @s
    by auto
    show ?case
    proof (cases bs rule: rev-cases)
    case Nil show ?thesis
    proof (cases m)
        case 0 with prevcase(2) show ?thesis by simp
    next
        case (Suc k) with prevcase(2) Nil show ?thesis by (cases k) auto
    qed
    next
    case (snoc ds d) with prevcase show ?thesis by simp
    qed
qed simp
lemma alternating-list-split:
    alternating-list (m+n)st=alternating-list mst@
    (if even m then alternating-list nst else alternating-list n t s)
    using alternating-list-SucSuc-ConsCons[of - s]
    by (induct n rule: nat-induct-step2) auto
lemma alternating-list-append:
    even m \Longrightarrow
        alternating-list m st @ alternating-list n s t = alternating-list (m+n)st
    odd m \Longrightarrow
        alternating-list m st @ alternating-list n t s=alternating-list (m+n) st
    using alternating-list-split[THEN sym, of m] by auto
lemma rev-alternating-list:
    rev (alternating-list n s t) =
    (if even n then alternating-list n t s else alternating-list n st)
    using alternating-list-SucSuc-ConsCons[of - s]
    by (induct n rule: nat-induct-step2) auto
lemma set-alternating-list: set (alternating-list n s t)\subseteq{s,t}
    by (induct n) auto
```

```
lemma set-alternating-list1:
    assumes n\geq1
    shows s\in set (alternating-list n s t)
proof (cases n)
    case 0 with assms show ?thesis by simp
next
    case (Suc m) thus ?thesis using alternating-list-Suc-Cons[of m s] by simp
qed
lemma set-alternating-list2:
    n\geq2\Longrightarrowset (alternating-list nst)={s,t}
proof (induct n rule: nat-induct-step2)
    case (SucSuc m) thus ?case
    using set-alternating-list alternating-list-SucSuc-ConsCons[of m st] by fastforce
qed auto
lemma alternating-list-in-lists: }a\inA\Longrightarrowb\inA\Longrightarrow\mathrm{ alternating-list n a b lists A
    by (induct n) auto
```


### 1.5.5 Binary relation chains

Here we consider lists where each pair of adjacent elements satisfy a given relation.
fun binrelchain :: (' $a \Rightarrow^{\prime} a \Rightarrow$ bool $) \Rightarrow{ }^{\prime} a$ list $\Rightarrow$ bool where binrelchain $P[]=$ True
$\mid$ binrelchain $P[x]=$ True
| binrelchain $P(x \# y \# x s)=(P x y \wedge$ binrelchain $P(y \# x s))$
lemma binrelchain-Cons-reduce: binrelchain $P(x \# x s) \Longrightarrow$ binrelchain $P$ xs by (induct xs) auto
lemma binrelchain-append-reduce1: binrelchain $P(x s @ y s) \Longrightarrow$ binrelchain $P$ xs proof (induct xs rule: list-induct-CCons)
case (CCons $x$ y $x$ s) with binrelchain-Cons-reduce show ?case by fastforce qed auto
lemma binrelchain-append-reduce2:
binrelchain $P(x s @ y s) \Longrightarrow$ binrelchain $P$ ys
proof (induct xs)
case (Cons $x$ xs) with binrelchain-Cons-reduce show ?case by fastforce
qed $\operatorname{simp}$
lemma binrelchain-Conssnoc-reduce:
binrelchain P $(x \# x s @[y]) \Longrightarrow$ binrelchain P xs
using binrelchain-append-reduce1 binrelchain-Cons-reduce by fastforce
lemma binrelchain-overlap-join:
binrelchain $P(x s @[x]) \Longrightarrow$ binrelchain $P(x \# y s) \Longrightarrow$ binrelchain $P(x s @ x \# y s)$
by (induct xs rule: list-induct-CCons) auto

## lemma binrelchain-join:

$\llbracket$ binrelchain $P(x s @[x])$; binrelchain $P(y \# y s) ; P x y \rrbracket \Longrightarrow$ binrelchain P (xs @ $x \# y \# y s)$
using binrelchain-overlap-join by fastforce
lemma binrelchain-snoc:
binrelchain $P(x s @[x]) \Longrightarrow P x y \Longrightarrow$ binrelchain $P(x s @[x, y])$
using binrelchain-join by fastforce
lemma binrelchain-sym-rev:
assumes $\bigwedge x y$. Pxy $\begin{aligned} & \text { x } \\ & y\end{aligned}$
shows binrelchain $P x s \Longrightarrow$ binrelchain $P$ (rev xs)
proof (induct xs rule: list-induct-CCons)
case (CCons $x$ y $x$ ss) with assms show ?case by (auto intro: binrelchain-snoc)
qed auto
lemma binrelchain-remdup-adj:
binrelchain $P(x s @[x, x] @ y s) \Longrightarrow$ binrelchain P $(x s @ x \# y s)$
by (induct xs rule: list-induct-CCons) auto
abbreviation proper-binrelchain $P x s \equiv$ binrelchain $P x s \wedge$ distinct xs
lemma binrelchain-obtain-proper:

```
    \(x \neq y \Longrightarrow\) binrelchain \(P(x \# x s @[y]) \Longrightarrow\)
    \(\exists z s\). set \(z s \subseteq\) set \(x s \wedge\) length \(z s \leq\) length \(x s \wedge\) proper-binrelchain \(P(x \# z s @[y])\)
proof (induct xs arbitrary: \(x\) )
    case (Cons wws)
    show ?case
    proof (cases \(w=x w=y\) rule: two-cases)
        case one
        from one(1) Cons(3) have binrelchain \(P(x \# w s @[y])\)
            using binrelchain-Cons-reduce by simp
    with \(\operatorname{Cons}(1,2)\) obtain \(z s\)
        where set \(z s \subseteq\) set ws length \(z s \leq\) length ws proper-binrelchain \(P(x \# z s @[y])\)
        by auto
    thus ?thesis by auto
    next
        case other
        with Cons(3) have proper-binrelchain P \((x \#[] @[y])\)
        using binrelchain-append-reduce1 by simp
    moreover have length []\(\leq\) length \((w \# w s)\) set []\(\subseteq\) set \((w \# w s)\) by auto
    ultimately show ?thesis by blast
    next
    case neither
    from Cons(3) have binrelchain P(w\#ws@[y])
        using binrelchain-Cons-reduce by simp
    with neither(2) Cons(1) obtain zs
```

where $z s:$ set $z s \subseteq$ set ws length $z s \leq$ length ws proper-binrelchain $P(w \# z s @[y])$
by auto
show ?thesis
proof (cases $x \in$ set $z s$ )
case True
from this obtain as bs where asbs: zs $=a s @ x \# b s$
using in-set-conv-decomp $[$ of $x]$ by auto
with $z s(3)$ have proper-binrelchain $P(x \# b s @[y])$
using binrelchain-append-reduce2 $[$ of $P w \# a s]$ by auto
moreover from $z s(1)$ asbs have set $b s \subseteq$ set $(w \# w s)$ by auto
moreover from asbs $z s(2)$ have length $b s \leq$ length ( $w \# w s$ ) by simp
ultimately show ?thesis by auto
next
case False
with $z s(3)$ neither(1) Cons(2,3) have proper-binrelchain $P(x \#(w \# z s) @[y])$ by simp
moreover from $z s(1)$ have set $(w \# z s) \subseteq$ set $(w \# w s)$ by auto
moreover from $z s(2)$ have length $(w \# z s) \leq$ length $(w \# w s)$ by simp
ultimately show ?thesis by blast
qed
qed (fastforce simp add: Cons(2))
qed $\operatorname{simp}$
lemma binrelchain-trans-Cons-snoc:
assumes $\backslash x$ y z. $P x y \Longrightarrow P$ y $z \Longrightarrow P x z$
shows binrelchain $P(x \# x s @[y]) \Longrightarrow P x y$
proof (induct xs arbitrary: x)
case Cons with assms show ?case using binrelchain-Cons-reduce by auto qed $\operatorname{simp}$
lemma binrelchain-cong:
assumes $\bigwedge x y . P x y \Longrightarrow Q x y$
shows binrelchain $P$ xs $\Longrightarrow$ binrelchain $Q$ xs
using assms binrelchain-Cons-reduce
by (induct xs rule: list-induct-CCons) auto
lemma binrelchain-funcong-Cons-snoc:
assumes $\bigwedge x y . P x y \Longrightarrow f y=f x$ binrelchain $P(x \# x s @[y])$
shows $f y=f x$
using assms binrelchain-cong[of $P]$
binrelchain-trans-Cons-snoc[of $\lambda x y . f y=f x x$ xs $y]$
by auto
lemma binrelchain-funcong-extra-condition-Cons-snoc:
assumes $\bigwedge x y . Q x \Longrightarrow P x y \Longrightarrow Q y \bigwedge x y . Q x \Longrightarrow P x y \Longrightarrow f y=f x$ shows $\quad Q x \Longrightarrow$ binrelchain $P(x \# z s @[y]) \Longrightarrow f y=f x$
proof (induct zs arbitrary: $x$ )
case (Cons z zs) with assms show ?case
using binrelchain-Cons-reduce[of P x z\#zs@[y]] by fastforce qed (simp add: assms)
lemma binrelchain-setfuncong-Cons-snoc:
$\llbracket \forall x \in A . \forall y . P x y \longrightarrow y \in A ; \forall x \in A . \forall y . P x y \longrightarrow f y=f x ; x \in A ;$
binrelchain $P(x \# z s @[y]) \rrbracket \Longrightarrow f y=f x$
using binrelchain-funcong-extra-condition-Cons-snoc[of $\lambda x$. $x \in A P f x z s y]$ by fast
lemma binrelchain-propcong-Cons-snoc:
assumes $\bigwedge x y . Q x \Longrightarrow P x y \Longrightarrow Q y$
shows $\quad Q x \Longrightarrow$ binrelchain $P(x \# x s @[y]) \Longrightarrow Q y$
proof (induct xs arbitrary: x)
case Cons with assms show ?case using binrelchain-Cons-reduce by auto qed (simp add: assms)

### 1.5.6 Set of subseqs

lemma subseqs-Cons: subseqs $(x \# x s)=$ map (Cons $x)($ subseqs $x s) @($ subseqs $x s)$ using cong-let[of subseqs xs $\lambda$ xss. map (Cons x) xss @ xss] by simp
abbreviation ssubseqs $x s \equiv$ set (subseqs $x$ s)
lemma nil-ssubseqs: []$\in$ ssubseqs $x s$
proof (induct xs)
case (Cons $x$ xs) thus ?case using subseqs-Cons[of $x]$ by simp
qed $\operatorname{simp}$
lemma ssubseqs-Cons: ssubseqs $(x \# x s)=($ Cons $x)$ ' $($ ssubseqs $x s) \cup$ ssubseqs $x s$ using subseqs-Cons[of $x$ ] by simp
lemma ssubseqs-refl: xs $\in$ ssubseqs $x s$
proof (induct $x s$ )
case (Cons $x$ xs) thus ?case using ssubseqs-Cons by fast
qed (rule nil-ssubseqs)
lemma ssubseqs-subset: as $\in$ ssubseqs $b s \Longrightarrow$ ssubseqs as $\subseteq$ ssubseqs bs
proof (induct bs arbitrary: as)
case (Cons b bs) show ?case
proof (cases as $\in$ set (subseqs bs))
case True with Cons show ?thesis using ssubseqs-Cons by fastforce
next
case False with Cons show ?thesis
using nil-ssubseqs[of b\#bs] ssubseqs-Cons[of hd as] ssubseqs-Cons[of b] by (cases as) auto
qed
qed $\operatorname{simp}$
lemma ssubseqs-lists:

```
    as }\in\mathrm{ lists }A\Longrightarrowbs\in\mathrm{ ssubseqs as }\Longrightarrowbs\in lists A
proof (induct as arbitrary: bs)
    case (Cons a as) thus ?case using ssubseqs-Cons[of a] by fastforce
qed simp
lemma delete1-ssubseqs:
    as@bs\in ssubseqs (as@[a]@bs)
proof (induct as)
    case Nil show ?case using ssubseqs-refl ssubseqs-Cons[of a bs] by auto
next
    case (Cons x xs) thus ?case using ssubseqs-Cons[of x] by simp
qed
lemma delete2-ssubseqs:
    as@bs@cs \in ssubseqs (as@[a]@bs@[b]@cs)
    using delete1-ssubseqs[of as@[a]@bs] delete1-ssubseqs ssubseqs-subset
    by fastforce
```


### 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 :: ' \(a \Rightarrow^{\prime} a \Rightarrow\) bool (infix \(\leq 50\) )
    and less \(\quad::{ }^{\prime} a \Rightarrow^{\prime} a \Rightarrow b o o l(\) infix \(<50)\)
    and less-eq' :: ' \(b \Rightarrow\) ' \(b \Rightarrow\) bool (infix \(\leq * 50\) )
    and less' \(\quad:{ }^{\prime} b \Rightarrow ' b \Rightarrow\) bool (infix \(<* 50\) )
+ fixes \(P\) :: 'a set
    and \(f::{ }^{\prime} a \Rightarrow\) ' \(b\)
    assumes ordsetmap: \(a \in P \Longrightarrow b \in P \Longrightarrow a \leq b \Longrightarrow f a \leq * f b\)
begin
lemma comp:
    assumes OrderingSetMap less-eq' less' less-eq" less" \(Q g\)
        \(f^{\iota} P \subseteq Q\)
    shows OrderingSetMap less-eq less less-eq" less" \({ }^{\prime \prime}\) ( \(g \circ f\) )
proof -
    from assms(1) interpret \(I\) : OrderingSetMap less-eq' less' less-eq" less" \(Q\) g.
    show ?thesis
        by standard (use assms(2) in 〈auto intro: ordsetmap I.ordsetmap〉)
qed
lemma subset: \(Q \subseteq P \Longrightarrow\) OrderingSetMap \((\leq)(<)(\leq *)(<*) Q f\)
```

using ordsetmap by unfold-locales fast
end
locale OrderingSetIso $=$ OrderingSetMap less-eq less less-eq' less ${ }^{\prime} P f$
for less-eq :: ' $a \Rightarrow$ ' $a \Rightarrow b o o l$ (infix $\leq 50$ )
and less $\quad::{ }^{\prime} a{ }^{\prime} a \Rightarrow b o o l($ infix $<50)$
and less-eq' :: ' $b \Rightarrow^{\prime} b \Rightarrow b o o l(i n f i x \leq * 50)$
and less' $\quad:: ' b \Rightarrow ' b \Rightarrow$ bool (infix <* 50)
and $P$ :: 'a set
and $f::{ }^{\prime} a \Rightarrow^{\prime} b$

+ assumes inj $\quad: \operatorname{inj}$-on $f P$
and rev-OrderingSetMap:
OrderingSetMap less-eq' less' less-eq less $\left(f^{\prime} P\right)$ (the-inv-into $P f$ )
abbreviation subset-ordering-iso $\equiv$ OrderingSetIso $(\subseteq)(\subset)(\subseteq)(\subset)$
lemma (in OrderingSetMap) isoI:
assumes inj-on $f P \bigwedge a b$. $a \in P \Longrightarrow b \in P \Longrightarrow f a \leq * f b \Longrightarrow a \leq b$
shows OrderingSetIso less-eq less less-eq' less' P $f$
using assms the-inv-into-f-f[OF assms(1)]
by unfold-locales auto
lemma OrderingSetIsoI-orders-greater2less:
fixes $f::$ 'a::order $\Rightarrow$ 'b::order
assumes inj-on $f P \bigwedge a b . a \in P \Longrightarrow b \in P \Longrightarrow(b \leq a)=(f a \leq f b)$
shows OrderingSetIso (greater-eq::' $a \Rightarrow^{\prime} a \Rightarrow b o o l$ ) (greater::' $a \Rightarrow^{\prime} a \Rightarrow b o o l$ )
(less-eq::'b $\Rightarrow^{\prime} b \Rightarrow$ bool) (less::' $b \Rightarrow^{\prime} b \Rightarrow$ bool) $P f$
proof
from $\operatorname{assms}(2)$ show $\bigwedge a b . a \in P \Longrightarrow b \in P \Longrightarrow b \leq a \Longrightarrow f a \leq f b$ by auto
from assms(2)
show $\bigwedge a b . a \in f^{\prime} P \Longrightarrow b \in f^{\prime} P \Longrightarrow b \leq a \Longrightarrow$
the-inv-into $P f a \leq$ the-inv-into $P f b$
using the-inv-into-f-f[OF assms(1)]
by force
qed (rule assms(1))
context OrderingSetIso
begin
lemmas ordsetmap $=$ ordsetmap
lemma ordsetmap-strict: $\llbracket a \in P ; b \in P ; a<b \rrbracket \Longrightarrow f a<* f b$
using domain.strict-iff-order codomain.strict-iff-order ordsetmap inj inj-on-contraD
by fastforce
lemmas inv-ordsetmap $=$ OrderingSetMap.ordsetmap[OF rev-OrderingSetMap]

```
lemma rev-ordsetmap: \llbracketa\inP;b\inP;fa\leq*fb\rrbracket\Longrightarrowa\leqb
    using inv-ordsetmap the-inv-into-f-f[OF inj] by fastforce
lemma inv-iso: OrderingSetIso less-eq' less' less-eq less (f`P) (the-inv-into P f)
    using inv-ordsetmap inj-on-the-inv-into[OF inj] the-inv-into-onto[OF inj]
        ordsetmap the-inv-into-the-inv-into[OF inj]
    by unfold-locales auto
lemmas inv-ordsetmap-strict = OrderingSetIso.ordsetmap-strict[OF inv-iso]
lemma rev-ordsetmap-strict: \llbracketa\inP;b\inP;fa<* fb\rrbracket\Longrightarrowa<b
    using inv-ordsetmap-strict the-inv-into-f-f[OF inj] by fastforce
lemma iso-comp:
    assumes OrderingSetIso less-eq' less' less-eq" less'\prime Q g f`}P\subseteq
    shows OrderingSetIso less-eq less less-eq" less" P (g\circf)
proof (rule OrderingSetMap.isoI)
    from assms show OrderingSetMap (\leq) (<) less-eq" less" P ( g\circf)
        using OrderingSetIso.axioms(1) comp by fast
    from assms(2) show inj-on ( }g\circf\mathrm{ ) P
        using OrderingSetIso.inj[OF assms(1)]
            comp-inj-on[OF inj, OF subset-inj-on]
    by fast
next
    fix ab
    from assms(2) show \llbracketa\inP;b\inP; less-eq"\prime ((g\circf) a) ((g\circf)b)\rrbracket\Longrightarrowa\leqb
        using OrderingSetIso.rev-ordsetmap[OF assms(1)] rev-ordsetmap by force
qed
lemma iso-subset:
    Q\subseteqP\Longrightarrow OrderingSetIso (\leq) (<) (\leq*) (<*)Qf
    using subset[of Q] subset-inj-on[OF inj] rev-ordsetmap
    by (blast intro: OrderingSetMap.isoI)
lemma iso-dual:
    <OrderingSetIso (\lambdaa b. less-eq b a) (\lambdaa b.less b a)
    (\lambdaa b. less-eq' b a) ( \lambdaa b.less' b a) P f>
    apply (rule OrderingSetMap.isoI)
    apply unfold-locales
    using inj
                apply (auto simp add: domain.refl codomain.refl
                domain.irrefl codomain.irrefl
                domain.order-iff-strict codomain.order-iff-strict
                ordsetmap-strict rev-ordsetmap-strict inj-onD
                intro:domain.trans codomain.trans
                domain.strict-trans codomain.strict-trans
                domain.antisym codomain.antisym)
done
```

end
lemma induced-pow-fun-subset-ordering-iso:
assumes inj-on $f A$
shows subset-ordering-iso (Pow A) ((')f)
proof
show $\bigwedge a b . a \in \operatorname{Pow} A \Longrightarrow b \in \operatorname{Pow} A \Longrightarrow a \subseteq b \Longrightarrow f$ ' $a \subseteq f^{\prime} b$ by fast
from assms show 2:inj-on (( $)$ f) (Pow A) using induced-pow-fun-inj-on by fast
show $\bigwedge a b . a \in\left({ }^{\prime}\right) f$ 'Pow $A \Longrightarrow b \in(`) f$ 'Pow $A \Longrightarrow a \subseteq b$

```
                \Longrightarrow \text { the-inv-into (Pow A)((`)f)a@ the-inv-into (Pow A) ((`)f)b}
```

proof-
fix Y1 Y2
assume $Y: Y 1 \in\left(\left({ }^{`}\right) f\right)$ 'Pow $A Y 2 \in\left(\left({ }^{\prime}\right) f\right)$ 'Pow $A Y 1 \subseteq Y 2$
from $Y(1,2)$ obtain $X 1 X 2$ where $X 1 \subseteq A \quad X 2 \subseteq A Y 1=f^{\prime} X 1 \quad Y 2=f^{\prime} X 2$ by auto
with assms $Y(3)$
show the-inv-into (Pow $A)\left(\left({ }^{\circ}\right) f\right) Y 1 \subseteq$ the-inv-into $($ Pow $A)((`) f) Y 2$
using inj-onD[OF assms] the-inv-into-f-f[OF 2, of X1]
the-inv-into-f-f[OF 2, of X2]
by blast
qed
qed

### 1.6.2 More arg-min

lemma is-arg-minI:
$\llbracket P x ; \bigwedge y . P y \Longrightarrow \neg m y<m x \rrbracket \Longrightarrow i s$-arg-min $m P x$ by (simp add: is-arg-min-def)
lemma is-arg-min-linorderI:
$\llbracket P x ; \bigwedge y . P y \Longrightarrow m x \leq(m y::-:: l i n o r d e r) \rrbracket \Longrightarrow$ is-arg-min $m P x$
by (simp add: is-arg-min-linorder)
lemma is-arg-min-eq:
$\llbracket i s$-arg-min $m P x ; P z ; m z=m x \rrbracket \Longrightarrow i s$-arg-min $m P z$
by (metis is-arg-min-def)
lemma is-arg-minD1: is-arg-min $m P x \Longrightarrow P x$
unfolding is-arg-min-def by fast
lemma is-arg-minD2: is-arg-min $m P x \Longrightarrow P y \Longrightarrow \neg m y<m x$
unfolding is-arg-min-def by fast
lemma is-arg-min-size: fixes $m::{ }^{\prime} a \Rightarrow{ }^{\prime} b::$ linorder
shows is-arg-min mPx mx=m (arg-min mP)
by (metis arg-min-equality is-arg-min-linorder)

```
lemma is-arg-min-size-subprop:
    fixes }m::\mp@subsup{'}{}{\prime}a=\mp@subsup{|}{}{\prime}b::linorder
    assumes is-arg-min m PxQ x \y.Q y \LongrightarrowPy
    shows m(arg-min m Q)=m(arg-min mP)
proof -
    have }\neg\mathrm{ is-arg-min m Q x C ᄀis-arg-min m P x
    proof
        assume x: ᄀis-arg-min m Q x
        from assms(2,3) show False
            using contrapos-nn[OF x, OF is-arg-minI] is-arg-minD2[OF assms(1)] by
auto
    qed
    with assms(1) show ?thesis
        using is-arg-min-size[of m] is-arg-min-size[of m] by fastforce
qed
```


### 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: $z \in P \Longrightarrow(\bigwedge x . x \in P \Longrightarrow z \leq x) \Longrightarrow$ has-bottom $P$ using has-bottom-def by auto
lemma has-uniq-bottom: has-bottom $P \Longrightarrow \exists!z \in P . \forall x \in P . z \leq x$
using has-bottom-def antisym by force
definition bottom :: 'a set $\Rightarrow{ }^{\prime} a$
where bottom $P \equiv($ THE $z . z \in P \wedge(\forall x \in P . z \leq x))$
lemma bottomD:
assumes has-bottom $P$
shows bottom $P \in P x \in P \Longrightarrow$ bottom $P \leq x$
using assms has-uniq-bottom theI' $[$ of $\lambda z . z \in P \wedge(\forall x \in P . z \leq x)]$
unfolding bottom-def
by auto
lemma bottomI: $z \in P \Longrightarrow(\bigwedge y . y \in P \Longrightarrow z \leq y) \Longrightarrow z=$ bottom $P$
using has-bottomI has-uniq-bottom
the1-equality[THEN sym, of $\lambda z . z \in P \wedge(\forall x \in P . z \leq x)]$
unfolding bottom-def
by $\operatorname{simp}$
end
lemma has-bottom-pow: order.has-bottom (Pow A)

```
    by (fast intro: order.has-bottomI)
lemma bottom-pow: order.bottom (Pow A) = {}
proof (rule order.bottomI[THEN sym]) qed auto
context OrderingSetMap
begin
abbreviation dombot \equivdomain.bottom P
abbreviation codbot \equiv codomain.bottom ( }\mp@subsup{f}{}{`}P
lemma im-has-bottom: domain.has-bottom P \Longrightarrow codomain.has-bottom (f`P)
    using domain.bottomD ordsetmap by (fast intro: codomain.has-bottomI)
lemma im-bottom: domain.has-bottom P\Longrightarrowf dombot = codbot
    using domain.bottomD ordsetmap by (auto intro: codomain.bottomI)
end
lemma (in OrderingSetIso) pullback-has-bottom:
    assumes codomain.has-bottom (f`P)
    shows domain.has-bottom P
proof (rule domain.has-bottomI)
    from assms show the-inv-into P f codbot }\in
        using codomain.bottomD(1) the-inv-into-into[OF inj] by fast
    from assms show \xx. x\inP\Longrightarrow the-inv-into P f codbot \leqx
        using codomain.bottomD inv-ordsetmap[of codbot] the-inv-into-f-f[OF inj]
        by fastforce
qed
lemma (in OrderingSetIso) pullback-bottom:
    \llbracketdomain.has-bottom P; x\inP; f x = codomain.bottom (f`P)\rrbracket\Longrightarrow
        x = domain.bottom P
    using im-has-bottom codomain.bottomD(2) rev-ordsetmap
    by (auto intro: domain.bottomI)
```


### 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 }=>\mp@subsup{}{}{\prime}'a=>\mathrm{ bool
    where minimal-in P x \equivx\inP^(\forallz\inP.\negz<x)
definition pseudominimal-in :: 'a set }=>\mathrm{ ' 'a }=>\mathrm{ bool
    where pseudominimal-in P x \equivminimal-in (P-{bottom P}) x
- only makes sense for has-bottom P
```

```
lemma minimal-inD1: minimal-in P }x\Longrightarrowx\in
    using minimal-in-def by fast
lemma minimal-inD2: minimal-in P x \Longrightarrow z\inP\Longrightarrow\negz<x
    using minimal-in-def by fast
lemma pseudominimal-inD1: pseudominimal-in P x \Longrightarrowx\inP
    using pseudominimal-in-def minimal-inD1 by fast
lemma pseudominimal-inD2:
    pseudominimal-in P x \Longrightarrowz\inP\Longrightarrowz<x\Longrightarrowz=bottom P
    using pseudominimal-in-def minimal-inD2 by fast
lemma pseudominimal-inI:
    assumes }\quadx\inPx\not=bottom P \z. z\inP\Longrightarrowz<x\Longrightarrowz= bottom P
    shows pseudominimal-in P x
    using assms
    unfolding pseudominimal-in-def minimal-in-def
    by fast
lemma pseudominimal-ne-bottom: pseudominimal-in P x \Longrightarrow x = bottom P
    using pseudominimal-in-def minimal-inD1 by fast
lemma pseudominimal-comp:
    \llbracket pseudominimal-in P x; pseudominimal-in P y; x\leqy\rrbracket\Longrightarrowx=y
    using pseudominimal-inD1 pseudominimal-inD2 pseudominimal-ne-bottom
        strict-iff-order[of x y]
    by force
end
lemma pseudominimal-in-pow:
    assumes order.pseudominimal-in (Pow A) x
    shows \existsa\inA. x={a}
proof-
    from assms obtain a where {a}\subseteqx
        using order.pseudominimal-ne-bottom bottom-pow[of A] by fast
    with assms show ?thesis
        using order.pseudominimal-inD1 order.pseudominimal-inD2[of - x {a}]
            bottom-pow
    by fast
qed
lemma pseudominimal-in-pow-singleton:
    a\inA\Longrightarrow order.pseudominimal-in (Pow A) {a}
    using singleton-pow bottom-pow by (fast intro: order.pseudominimal-inI)
lemma no-pseudominimal-in-pow-is-empty:
```

```
(\bigwedgex. ᄀ order.pseudominimal-in (Pow A) {x})\LongrightarrowA={}
using pseudominimal-in-pow-singleton by (fast intro: equals0I)
lemma (in OrderingSetIso) pseudominimal-map:
    domain.has-bottom P \Longrightarrow domain.pseudominimal-in P x \Longrightarrow
    codomain.pseudominimal-in (f`P) (f x)
using domain.pseudominimal-inD1 pullback-bottom
            domain.pseudominimal-ne-bottom rev-ordsetmap-strict
            domain.pseudominimal-inD2 im-bottom
    by (blast intro:codomain.pseudominimal-inI)
lemma (in OrderingSetIso) pullback-pseudominimal-in:
    |domain.has-bottom P; x\inP; codomain.pseudominimal-in (f`P) (fx)\rrbracket\Longrightarrow
        domain.pseudominimal-in P x
    using im-bottom codomain.pseudominimal-ne-bottom ordsetmap-strict
        codomain.pseudominimal-inD2 pullback-bottom
    by (blast intro: domain.pseudominimal-inI)
```


### 1.6.5 Set of elements below another

abbreviation (in ordering) below-in :: 'a set $\Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a$ set (infix.$\leq 70$ ) where $P . \leq x \equiv\{y \in P . y \leq x\}$
abbreviation (in ord) below-in :: 'a set $\Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a$ set (infix.$\left.\leq 70\right)$ where $P . \leq x \equiv\{y \in P . y \leq x\}$
context ordering
begin
lemma below-in-refl: $x \in P \Longrightarrow x \in P . \leq x$ using refl by fast
lemma below-in-singleton: $x \in P \Longrightarrow P . \leq x \subseteq\{y\} \Longrightarrow y=x$ using below-in-refl by fast
lemma bottom-in-below-in: has-bottom $P \Longrightarrow x \in P \Longrightarrow$ bottom $P \in P . \leq x$ using bottomD by fast
lemma below-in-singleton-is-bottom:
$\llbracket$ has-bottom $P ; x \in P ; P . \leq x=\{x\} \rrbracket \Longrightarrow x=$ bottom $P$
using bottom-in-below-in by fast
lemma bottom-below-in:
has-bottom $P \Longrightarrow x \in P \Longrightarrow$ bottom $(P . \leq x)=$ bottom $P$
using bottom-in-below-in by (fast intro: bottomI[THEN sym])
lemma bottom-below-in-relative:
$\llbracket$ has-bottom $(P . \leq y) ; x \in P ; x \leq y \rrbracket \Longrightarrow \operatorname{bottom}(P . \leq x)=\operatorname{bottom}(P . \leq y)$
using bottomD trans by (blast intro: bottomI[THEN sym])

```
lemma has-bottom-pseudominimal-in-below-inI:
    assumes has-bottom P x\inP pseudominimal-in P y y\leqx
    shows pseudominimal-in (P.\leqx) y
    using assms(3,4) pseudominimal-inD1[OF assms(3)]
        pseudominimal-inD2[OF assms(3)]
        bottom-below-in[OF assms(1,2)] pseudominimal-ne-bottom
    by (force intro: pseudominimal-inI)
lemma has-bottom-pseudominimal-in-below-in:
    assumes has-bottom P x\inP pseudominimal-in (P.\leqx) y
    shows pseudominimal-in P y
    using pseudominimal-inD1[OF assms(3)]
        pseudominimal-inD2[OF assms(3)]
        pseudominimal-ne-bottom[OF assms(3)]
        bottom-below-in[OF assms(1,2)]
        strict-implies-order[of - y] trans[of - y x]
    by (force intro: pseudominimal-inI)
lemma pseudominimal-in-below-in:
    assumes has-bottom (P.\leqy) x\inP x\leqy pseudominimal-in (P.\leqx) w
    shows pseudominimal-in (P.\leqy) w
    using assms(3) trans[of wx y] trans[of - w x] strict-iff-order
        pseudominimal-inD1[OF assms(4)]
        pseudominimal-inD2[OF assms(4)]
        pseudominimal-ne-bottom[OF assms(4)]
        bottom-below-in-relative[OF assms(1-3)]
    by (force intro: pseudominimal-inI)
lemma collect-pseudominimals-below-in-less-eq-top:
    assumes OrderingSetIso less-eq less (\subseteq)(\subset) (P.\leqx) f
        f}(P.\leqx)=Pow A a\subseteq{y. pseudominimal-in (P.\leqx) y
    defines w\equiv the-inv-into (P.\leqx) f(U(\mp@subsup{f}{}{\prime}a))
    shows w\leqx
proof-
    from assms(2,3) have (U(\mp@subsup{f}{}{\prime}a))\in\mp@subsup{f}{}{`}(P.\leqx)
    using pseudominimal-inD1 by fastforce
    with assms(4) show ?thesis
    using OrderingSetIso.inj[OF assms(1)] the-inv-into-into[of f P. }\leqx]\mathrm{ by force
qed
lemma collect-pseudominimals-below-in-poset:
    assumes OrderingSetIso less-eq less (\subseteq)(\subset) (P.\leqx)f
        f}(P.\leqx)= Pow 
    a\subseteq{y.pseudominimal-in (P.\leqx) y}
defines w 三 the-inv-into (P.\leqx) f(\bigcup}(\mp@subsup{f}{}{\prime}a)
shows }\quadw\in
using assms(2-4) OrderingSetIso.inj[OF assms(1)] pseudominimal-inD1
    the-inv-into-into[of f P. \leqx \bigcup(f`a)]
```

```
    by
        force
lemma collect-pseudominimals-below-in-eq:
    assumes }x\inP\mathrm{ OrderingSetIso less-eq less (`)(C) (P. \x) f
        f}(P.\leqx)=Pow A a \subseteq{y.pseudominimal-in (P.\leqx) y
    defines w: w\equiv the-inv-into (P.\leqx) f(U(f`a))
    shows }a={y.pseudominimal-in (P.\leqw) y
proof
    from assms(3) have has-bot-ltx: has-bottom (P.\leqx)
        using has-bottom-pow OrderingSetIso.pullback-has-bottom[OF assms(2)]
        by auto
    from assms(3,4) have Un-fa: (U(\mp@subsup{f}{}{\prime}a)) \in\mp@subsup{f}{}{\prime}(P.\leqx)
    using pseudominimal-inD1 by fastforce
    from assms have w-le-x:w\leqx and w-P:w\inP
    using collect-pseudominimals-below-in-less-eq-top
        collect-pseudominimals-below-in-poset
    by auto
    show }a\subseteq{y.pseudominimal-in (P.\leqw) y
    proof
    fix y assume y:y\ina
    show }y\in{y.pseudominimal-in (P.\leqw) y
    proof (rule CollectI, rule pseudominimal-inI, rule CollectI, rule conjI)
        from y assms(4) have y-le-x: y \inP.\leqx using pseudominimal-inD1 by fast
        thus }y\inP\mathrm{ by simp
        from y w show y\leqw
            using y-le-x Un-fa OrderingSetIso.inv-ordsetmap[OF assms(2)]
                the-inv-into-f-f[OF OrderingSetIso.inj, OF assms(2), of y]
            by fastforce
        from assms(1) y assms(4) show y}\not=\operatorname{bottom (P.\leqw)
            using w-P w-le-x has-bot-ltx bottom-below-in-relative
                pseudominimal-ne-bottom
            by fast
    next
        fix z assume z: z\inP.\leqwz<y
        with y assms(4) have }\overline{z}=\operatorname{bottom}(P.\leqx
            using w-le-x trans pseudominimal-inD2[ of P.\leqx y z] by fast
        moreover from assms(1) have bottom (P.\leqw)= bottom (P.\leqx)
            using has-bot-ltx w-P w-le-x bottom-below-in-relative by fast
            ultimately show z=bottom (P.\leqw) by simp
        qed
    qed
    show }a\supseteq{y.pseudominimal-in (P.\leqw) y
    proof
    fix v}\mathrm{ assume v}\in{y.pseudominimal-in (P.\leqw) y
    hence pseudominimal-in (P.\leqw) v by fast
    moreover hence v-pm-ltx: pseudominimal-in (P.\leqx) v
        using has-bot-ltx w-P w-le-x pseudominimal-in-below-in by fast
    ultimately
        have fv\leq(\bigcup(\mp@subsup{f}{}{\prime}a))
```

using $w$ pseudominimal-inD1[of - v] pseudominimal-inD1[of - v] w-le-x w-P OrderingSetIso.ordsetmap $[O F \operatorname{assms}$ (2), of $v w] U n$-fa OrderingSetIso.inj[OF assms(2)] f-the-inv-into-f
by force
with $\operatorname{assms}(3)$ obtain $y$ where $y \in a f v \subseteq f y$
using v-pm-ltx has-bot-ltx pseudominimal-in-pow OrderingSetIso.pseudominimal-map[OF assms(2)]
by force
with $\operatorname{assms}(2,4)$ show $v \in a$
using $v$-pm-ltx pseudominimal-inD1 pseudominimal-comp[of - vy] OrderingSetIso.rev-ordsetmap[OF assms(2), of $v y]$
by fast
qed
qed
end

### 1.6.6 Lower bounds

context ordering
begin
definition lbound-of :: ' $a \Rightarrow^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow$ bool where lbound-of $x$ y $b \equiv b \leq x \wedge b \leq y$
lemma lbound-ofI: $b \leq x \Longrightarrow b \leq y \Longrightarrow$ lbound-of $x$ y $b$ using lbound-of-def by fast
lemma lbound-ofD1: lbound-of $x$ y $b \Longrightarrow b \leq x$ using lbound-of-def by fast
lemma lbound-ofD2: lbound-of $x$ y $b \Longrightarrow b \leq y$ using lbound-of-def by fast
definition glbound-in-of :: ' $a$ set $\Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow$ bool where glbound-in-of $P x y b \equiv$ $b \in P \wedge$ lbound-of $x$ y $b \wedge(\forall a \in P$. lbound-of $x$ y $a \longrightarrow a \leq b)$
lemma glbound-in-ofI:
$\llbracket b \in P$; lbound-of $x$ y $b ; \bigwedge a . a \in P \Longrightarrow$ lbound-of $x$ y $a \Longrightarrow a \leq b \rrbracket \Longrightarrow$ glbound-in-of P x y b
using glbound-in-of-def by auto
lemma glbound-in-ofD-in: glbound-in-of $P x y b \Longrightarrow b \in P$
using glbound-in-of-def by fast
lemma glbound-in-ofD-lbound: glbound-in-of $P x y b \Longrightarrow l b o u n d-o f x$ y $b$ using glbound-in-of-def by fast

```
lemma glbound-in-ofD-glbound:
    glbound-in-of P x y b\Longrightarrowa\inP\Longrightarrowlbound-of x y a \Longrightarrowa\leqb
    using glbound-in-of-def by fast
lemma glbound-in-of-less-eq1: glbound-in-of P x y b \Longrightarrowb\leqx
    using glbound-in-ofD-lbound lbound-ofD1 by fast
lemma glbound-in-of-less-eq2: glbound-in-of P x y b \Longrightarrowb\leqy
    using glbound-in-ofD-lbound lbound-ofD2 by fast
lemma pseudominimal-in-below-in-less-eq-glbound:
    assumes pseudominimal-in (P.\leqx) w pseudominimal-in (P.\leqy) w
        glbound-in-of P x y b
    shows w\leqb
    using assms lbound-ofI glbound-in-ofD-glbound
    pseudominimal-inD1[of P.\leqx] pseudominimal-inD1[of P.\leqy]
    by fast
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 }=>\mathrm{ bool
    where simplex-like P}\equiv\mathrm{ finite P ^
        ( }\existsfA::nat set
            OrderingSetIso less-eq less (\subseteq)(\subset)Pf\wedge (`)P=Pow A
            )
lemma simplex-likeI:
    assumes finite P OrderingSetIso less-eq less (\subseteq)(\subset)Pf
        f}\mp@subsup{|}{}{\prime}=P\mathrm{ Pow (A::nat set)
    shows simplex-like P
    using assms simplex-like-def by auto
lemma simplex-likeD-finite: simplex-like P \Longrightarrow finite P
    using simplex-like-def by simp
lemma simplex-likeD-iso:
    simplex-like P\Longrightarrow
        \existsf A::nat set. OrderingSetIso less-eq less (\subseteq)(\subset) Pf ^ff}P=Pow 
    using simplex-like-def by simp
lemma simplex-like-has-bottom: simplex-like P \Longrightarrow has-bottom P
```

using simplex-likeD-iso has-bottom-pow OrderingSetIso.pullback-has-bottom by fastforce
lemma simplex-like-no-pseudominimal-imp-singleton: assumes simplex-like $P \bigwedge x$. $\neg$ pseudominimal-in $P x$ shows $\exists p$. $P=\{p\}$
proof -
obtain $f$ and $A::$ nat set
where fA: OrderingSetIso less-eq less $(\subseteq)(\subset) P f f^{\prime} P=$ Pow $A$
using simplex-likeD-iso[OF assms(1)]
by auto
define $e$ where $e: e \equiv\{ \}::$ nat set
with $f A(2)$ have $e \in f^{\prime} P$ using Pow-bottom by simp
from this obtain $p$ where $p \in P f p=e$ by fast
have $\wedge x$. $\neg$ order.pseudominimal-in (Pow $A)\{x\}$
proof
fix $x$ :: nat assume order.pseudominimal-in (Pow $A$ ) $\{x\}$
moreover with $f A(2)$ have $\{x\} \in f^{\prime} P$
using order.pseudominimal-inD1 by fastforce
ultimately show False
using assms fA simplex-like-has-bottom
OrderingSetIso.pullback-pseudominimal-in
by fastforce
qed
with e $f A(2)$ show ?thesis
using no-pseudominimal-in-pow-is-empty
inj-on-to-singleton[OF OrderingSetIso.inj, OF fA(1)]
by force
qed
lemma simplex-like-no-pseudominimal-in-below-in-imp-singleton:
$\llbracket x \in P ;$ simplex-like $(P . \leq x) ; \bigwedge z . \neg$ pseudominimal-in $(P . \leq x) z \rrbracket \Longrightarrow$ $P . \leq x=\{x\}$
using simplex-like-no-pseudominimal-imp-singleton below-in-singleton $[$ of $x P]$ by fast
lemma pseudo-simplex-like-has-bottom:
OrderingSetIso less-eq less $(\subseteq)(\subset) P f \Longrightarrow f^{\prime} P=$ Pow $A \Longrightarrow$ has-bottom $P$
using has-bottom-pow OrderingSetIso.pullback-has-bottom by fastforce
lemma pseudo-simplex-like-above-pseudominimal-is-top:
assumes OrderingSetIso less-eq less $(\subseteq)(\subset) P f f^{‘} P=$ Pow A $t \in P$ $\bigwedge x$. pseudominimal-in $P x \Longrightarrow x \leq t$
shows $f t=A$
proof
from $\operatorname{assms}(2,3)$ show $f t \subseteq A$ by fast
show $A \subseteq f t$
proof

```
    fix a assume }a\in
    moreover with assms(2) have {a}\in f`}P\mathrm{ by simp
    ultimately show a}\inf
    using assms pseudominimal-in-pow-singleton[of a A]
        pseudo-simplex-like-has-bottom[of P f]
            OrderingSetIso.pullback-pseudominimal-in[OF assms(1)]
            OrderingSetIso.ordsetmap[OF assms(1),of - t]
        by force
    qed
qed
lemma pseudo-simplex-like-below-in-above-pseudominimal-is-top:
    assumes x\inP OrderingSetIso less-eq less (\subseteq)(\subset) (P.\leqx) f
        f}(P.\leqx)= Pow A t E P. \leqx x
    \y.pseudominimal-in (P.\leqx) y \Longrightarrowy\leqt
    shows t=x
    using assms(1,3-5)
            pseudo-simplex-like-above-pseudominimal-is-top[OF assms(2)]
            below-in-refl[of x P] OrderingSetIso.ordsetmap[OF assms(2), of t x]
            inj-onD[OF OrderingSetIso.inj[OF assms(2)], of t x]
    by
        auto
lemma simplex-like-below-in-above-pseudominimal-is-top:
    assumes }x\inP\mathrm{ simplex-like (P. \x) t f P. \x
    \y.pseudominimal-in (P.\leqx) y\Longrightarrowy\leqt
    shows t=x
    using assms simplex-likeD-iso
    pseudo-simplex-like-below-in-above-pseudominimal-is-top[of x P - t]
    by blast
end
lemma (in OrderingSetIso) simplex-like-map:
    assumes domain.simplex-like P
    shows codomain.simplex-like (f`P)
proof-
    obtain g::'a m nat set and A::nat set
    where gA: OrderingSetIso (\leq) (<) (\subseteq) (\subset)Pggg`P=Pow A
    using domain.simplex-likeD-iso[OF assms]
    by auto
    from gA(1) inj
    have OrderingSetIso (\leq*)(<*)(\subseteq)(\subset) (f`P)
                (g\circ(the-inv-into Pf))
    using OrderingSetIso.iso-comp[OF inv-iso] the-inv-into-onto
    by fast
moreover from gA(2) inj have (g\circ(the-inv-into P f))'(f`P) = Pow A
    using the-inv-into-onto by (auto simp add: image-comp[THEN sym])
    moreover from assms have finite (f`P)
    using domain.simplex-likeD-finite by fast
```

ultimately show ？thesis by（auto intro：codomain．simplex－likeI） qed
lemma（in OrderingSetIso）pullback－simplex－like：
assumes finite $P$ codomain．simplex－like（ $f^{〔} P$ ）
shows domain．simplex－like $P$
proof－
obtain $g:: ' b \Rightarrow$ nat set and $A::$ nat set
where $g$ A：OrderingSetIso $(\leq *)(<*)(\subseteq)(\subset)\left(f^{\prime} P\right) g$ $g^{\prime}\left(f^{\iota} P\right)=$ Pow $A$
using codomain．simplex－likeD－iso［OF assms（2）］
by auto
from assms（1）$g A(2)$ show ？thesis
using iso－comp［OF gA（1）］
by（auto intro：domain．simplex－likeI simp add：image－comp）
qed
lemma simplex－like－pow：
assumes finite $A$
shows order．simplex－like（Pow A）
proof－
from assms obtain $f::^{\prime} a \Rightarrow$ nat where $\operatorname{inj}$－on $f A$
using finite－imp－inj－to－nat－seg［of $A]$ by auto
hence subset－ordering－iso（Pow A）（（＇）f）
using induced－pow－fun－subset－ordering－iso by fast
with assms show ？thesis using induced－pow－fun－surj
by（blast intro：order．simplex－likeI）
qed

## 1．6．8 The superset ordering

abbreviation supset－has－bottom $\equiv$ ordering．has－bottom $\quad(\supseteq)$
abbreviation supset－bottom $\quad \equiv$ ordering．bottom
abbreviation supset－lbound－of $\equiv$ ordering．lbound－of $\quad(\supseteq$ ）
abbreviation supset－glbound－in－of $\equiv$ ordering．glbound－in－of $\quad(\supseteq)$
abbreviation supset－simplex－like $\equiv$ ordering．simplex－like（〕）（ゝ）
abbreviation supset－pseudominimal－in $\equiv$

$$
\text { ordering.pseudominimal-in }(\supseteq)(\supset)
$$

abbreviation supset－below－in ：：＇a set set $\Rightarrow{ }^{\prime}$＇a set $\Rightarrow{ }^{\prime}$＇a set set（infix ．〇 70）
where $P . \supseteq A \equiv$ ordering．below－in（ొ）PA
lemma supset－poset：ordering（〇）（ゝ）．．
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 $\quad[$ 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)
    using inj-on-minus-set by (fast intro: OrderingSetIsoI-orders-greater2less)
lemma simplex-like-pow-above-in:
    assumes finite A X\subseteqA
    shows supset-simplex-like ((Pow A).\supseteqX)
proof (
    rule OrderingSetIso.pullback-simplex-like, rule OrderingSetIso.iso-subset,
    rule OrderingSetIso-pow-complement
)
    from assms(1) show finite ((Pow A).\supseteqX) by simp
    from assms(1) have finite (Pow (A-X)) by fast
    moreover from assms(2) have ((-)A)'((Pow A).\supseteqX)=Pow (A-X)
        by auto
    ultimately
        show ordering.simplex-like (\subseteq)(\subset)(((-)A)'((Pow A).\supseteqX))
        using simplex-like-pow
        by fastforce
qed fast
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: $(j:: n a t)+j=2 * j$
by (induct $j$ ) auto
lemma (in comm-semiring-1) odd-n0: odd $m \Longrightarrow m \neq 0$
using dvd-0-right by fast
lemma (in semigroup-add) add-assoc4: $a+b+c+d=a+(b+c+d)$
using add.assoc by simp
lemmas (in monoid-add) sum-list-map-cong $=$ arg-cong[OF map-cong, OF refl, of - - sum-list]
context group-add
begin
lemma map-uminus-order2: $\forall s \in$ set ss. $s+s=0 \Longrightarrow$ map (uminus) $s s=s s$
by (induct ss) (auto simp add: minus-unique)
lemma uminus-sum-list: - sum-list as $=$ sum-list (mapuminus (rev as))
by (induct as) (auto simp add: minus-add)
lemma uminus-sum-list-order2:
$\forall s \in$ set ss. $s+s=0 \Longrightarrow-$ sum-list ss $=$ sum-list (rev ss)
using uminus-sum-list map-uminus-order2 by simp
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
    by fast
setup-lifting type-definition-permutation
abbreviation permutation-apply :: 'a permutation }=>\mp@subsup{}{}{\prime}a=\mp@subsup{|}{}{\prime}a(\mathbf{infixr}->90
    where p->a\equiv permutation pa
abbreviation permutation-image :: 'a permutation => 'a set }=>\mp@subsup{}{}{\prime}'a se
    (infixr '}->90\mathrm{ )
    where p'->A\equiv permutation p'A
lemma permutation-eq-image: a ' }->A=\mp@subsup{a}{}{\prime}->B\LongrightarrowA=
    using permutation[of a] inj-eq-image[OF bij-is-inj] by auto
instantiation permutation :: (type) zero
```

```
begin
lift-definition zero-permutation :: 'a permutation is id::' }a\not=\mp@subsup{}{}{\prime}a\mathrm{ by simp
instance ..
end
instantiation permutation :: (type) plus
begin
lift-definition plus-permutation :: 'a permutation }=>\mp@subsup{|}{}{\prime}a\mathrm{ permutation }=>\mp@subsup{}{}{\prime}\mathrm{ 'a permu-
tation
    is comp
    using bij-comp
    by fast
instance ..
end
lemma plus-permutation-abs-eq:
    bijf\Longrightarrowbij g\Longrightarrow
    Abs-permutation f + Abs-permutation g=Abs-permutation (f\circg)
    by (simp add: plus-permutation.abs-eq eq-onp-same-args)
instance permutation :: (type) semigroup-add
proof
    fix ab c :: 'a permutation show }a+b+c=a+(b+c
        using comp-assoc[of permutation a permutation b permutation c]
        by transfer simp
qed
instance permutation :: (type) monoid-add
proof
    fix a :: 'a permutation
    show }0+a=a\mathrm{ by transfer simp
    show }a+0=a\mathrm{ by transfer simp
qed
instantiation permutation :: (type) uminus
begin
lift-definition uminus-permutation :: 'a permutation => 'a permutation
    is }\quad\lambdaf.the-inv 
    using bij-betw-the-inv-into
    by fast
instance ..
end
instantiation permutation :: (type) minus
begin
lift-definition minus-permutation :: 'a permutation }=>\mp@subsup{|}{}{\prime}a\mathrm{ permutation }=>\mp@subsup{|}{}{\prime}\mathrm{ 'a per-
mutation
    is }\quad\lambdafg.f\circ(the-inv g
    using bij-betw-the-inv-into bij-comp
```

```
    by fast
instance ..
end
lemma minus-permutation-abs-eq:
    bij}f\Longrightarrow\mathrm{ bij }g
    Abs-permutation f - Abs-permutation g=Abs-permutation (f\circ the-inv g)
    by (simp add: minus-permutation.abs-eq eq-onp-same-args)
instance permutation :: (type) group-add
proof
    fix a b :: 'a permutation
    show - a + a = 0 using the-inv-leftinv[of permutation a] by transfer simp
    show }a+-b=a-b by transfer sim
qed
```


### 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 .
sublocale add-mult-translate: monoid-mult 0 plus
by unfold-locales (auto simp add: add.assoc)
abbreviation nataction :: ' $a \Rightarrow n a t \Rightarrow{ }^{\prime} a$ (infix $+{ }^{\wedge} 80$ )
where $a+{ }^{\widehat{ } n} \equiv$ 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 $(2 * n)$ st) $=(s+t)+{ }^{\wedge} n$
by (induct $n$ ) (auto simp add: nataction-Suc2[THEN sym])
lemma nataction-add-flip: $(a+b)+\widehat{ }($ Suc $n)=a+(b+a)+\widehat{ } n+b$
using nataction-Suc2 add.assoc by (induct $n$ arbitrary: a b) auto
end
lemma (in group-add) nataction-add-eq0-flip:
assumes $(a+b)+{ }^{\wedge} n=0$
shows $(b+a)+{ }^{\widehat{ }} n=0$
proof (cases $n$ )
case (Suc k) with assms show ?thesis
using nataction-add-flip add.assoc $[$ of $-a a+b(a+b)+\uparrow k]$ by simp
qed $\operatorname{simp}$

### 2.3.2 Additive order of an element

context monoid-add

## begin

definition add-order :: 'a $\Rightarrow$ nat
where add-order $a \equiv$ if $\left(\exists n>0 . a+{ }^{\wedge} n=0\right)$ then
(LEAST $n, n>0 \wedge a+{ }^{\widehat{ } n}=0$ ) else 0
lemma add-order: $a+\uparrow($ add-order $a)=0$
using LeastI-ex[of $\lambda n$. $\left.n>0 \wedge a+{ }^{`} n=0\right]$ add-order-def by simp
lemma add-order-least: $n>0 \Longrightarrow a+{ }^{\wedge} n=0 \Longrightarrow$ add-order $a \leq n$
using Least-le[of $\left.\lambda n . n>0 \wedge a+{ }^{\wedge} n=0\right]$ add-order-def by simp
lemma add-order-equality:
$\llbracket n>0 ; a+{ }^{`} n=0 ;\left(\bigwedge m . m>0 \Longrightarrow a+{ }^{`} m=0 \Longrightarrow n \leq m\right) \rrbracket \Longrightarrow$ add-order $a=n$
using Least-equality[of $\lambda n . n>0 \wedge a+{ }^{\wedge} n=0$ ] add-order-def by auto
lemma add-order0: add-order $0=1$
using add-order-equality by simp
lemma add-order-gt0: (add-order $a>0)=\left(\exists n>0 . a+{ }^{`} n=0\right)$
using LeastI-ex[of $\lambda n$. $n>0 \wedge a+{ }^{\wedge} n=0$ ] add-order-def by simp
lemma add-order-eq0: add-order $a=0 \Longrightarrow n>0 \Longrightarrow a+{ }^{\wedge} n \neq 0$
using add-order-gt0 by force
lemma less-add-order-eq-0:
assumes $a+\wedge_{k}=0 k<$ add-order $a$
shows $k=0$
proof (cases $k=0$ )
case False
moreover with $\operatorname{assms}(1)$ have $\exists n>0 . a+{ }^{\wedge} n=0$ by fast
ultimately show ?thesis
using assms add-order-def not-less-Least[of $k \lambda n . n>0 \wedge a+\widehat{n}=0]$
by auto
qed $\operatorname{simp}$
lemma less-add-order-eq-0-contra: $k>0 \Longrightarrow k<a d d$-order $a \Longrightarrow a+{ }^{\wedge} k \neq 0$ using less-add-order-eq-0 by fast
lemma add-order-relator: add-order $(a+\uparrow(a d d$-order $a))=1$
using add-order by (auto intro: add-order-equality)
abbreviation pair-relator-list :: ' $a \Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a$ list
where pair-relator-list $s t \equiv$ alternating-list (2*add-order $(s+t)) s t$
abbreviation pair-relator-halfist $::{ }^{\prime} a{ }^{\prime} a \Rightarrow$ 'a list
where pair-relator-halfist $s t \equiv$ alternating-list (add-order $(s+t))$ st
abbreviation pair-relator-halfist2 :: ' $a \Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a$ list
where pair-relator-halfist2 s $t \equiv$
(if even (add-order $(s+t)$ ) then pair-relator-halfist $s t$ else pair-relator-halfist $t s$ )
lemma sum-list-pair-relator-list: sum-list (pair-relator-list st) $=0$
by (auto simp add: add-order alternating-sum-list-conv-nataction)
end
context group-add
begin
lemma add-order-add-eq1: add-order $(s+t)=1 \Longrightarrow t=-s$
using add-order [of $s+t$ ] by (simp add: minus-unique)
lemma add-order-add-sym: add-order $(t+s)=a d d$-order $(s+t)$
proof (cases add-order $(t+s)=0$ add-order $(s+t)=0$ rule: two-cases $)$
case one thus?thesis
using add-order nataction-add-eq0-flip[of st] add-order-eq0 by auto
next
case other thus ?thesis
using add-order nataction-add-eq0-flip $[$ of $t s]$ add-order-eq0 by auto
next
case neither thus ?thesis
using add-order[of $s+t$ ] add-order [of $t+s$ ]
nataction-add-eq0-flip[of st] nataction-add-eq0-flip[of $t s]$
add-order-least[of add-order $(s+t)]$ add-order-least[of add-order $(t+s)$ ]
by fastforce
qed $\operatorname{simp}$
lemma pair-relator-halfist-append:
pair-relator-halfist st @ pair-relator-halfist2 st=pair-relator-list st
using alternating-list-split[of add-order ( $s+t$ ) add-order $(s+t) \mathrm{s} t$ ]
by (auto simp add: times2-conv-add add-order-add-sym)
lemma rev-pair-relator-list: rev (pair-relator-list st) $=$ pair-relator-list $t s$ by (simp add:rev-alternating-list add-order-add-sym)
lemma pair-relator-halfist2-conv-rev-pair-relator-halfist:
pair-relator-halfist2 st=rev (pair-relator-halfist ts)
by (auto simp add: add-order-add-sym rev-alternating-list)
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 # ' 'a list
    where
        sums [] = [0]
    | sums (x#xs) = 0 # map ((+) x) (sums xs)
lemma length-sums:length (sums xs)=Suc (length xs)
    by (induct xs) auto
lemma sums-snoc: sums (xs@[x]) = sums xs @ [sum-list (xs@[x])]
    by (induct xs) (auto simp add: add.assoc)
lemma sums-append2:
    sums (xs@ys) = butlast (sums xs)@ map ((+) (sum-list xs)) (sums ys)
proof (induct ys rule: rev-induct)
    case Nil show ?case by (cases xs rule: rev-cases) (auto simp add: sums-snoc)
next
    case (snoc y ys) thus ?case using sums-snoc[of xs@ys] by (simp add: sums-snoc)
qed
lemma sums-Cons-conv-append-tl:
    sums (x#xs) = 0 # x # map ((+) x) (tl (sums xs))
    by (cases xs) auto
lemma pullback-sums-map-middle2:
    map F (sums xs) = ds@[d,e]@es \Longrightarrow
    \existsas a bs. xs =as@[a]@bs ^ map F (sums as) =ds@[d]^
        d=F(sum-list as)}\wedgee=F(sum-list (as@[a])
proof (induct xs es rule: list-induct2-snoc)
    case (Nil2 xs)
    show ?case
    proof (cases xs rule: rev-cases)
    case Nil with Nil2 show ?thesis by simp
    next
    case (snoc ys y) have ys: xs = ys@[y] by fact
    with Nil2(1) have y: map F (sums ys) = ds@[d] e=F(sum-list (ys@[y]))
        by (auto simp add: sums-snoc)
    show ?thesis
    proof (cases ys rule: rev-cases)
        case Nil
        with ys y have
            xs=[]@[y]@[] map F (sums [])=ds@[d]
            d=F(sum-list []) e=F(sum-list ([]@[y]))
```

```
            by auto
            thus ?thesis by fast
        next
            case (snoc zs z)
            with y(1) have z: map F (sums zs)=ds d=F (sum-list (zs@[z]))
                by (auto simp add: sums-snoc)
                    from z(1) ys y snoc have
            xs =(zs@[z])@[y]@[] map F (sums (zs@[z]))=ds@[d]
            e=F(sum-list ((zs@[z])@[y]))
            by auto
            with z(2) show ?thesis by fast
        qed
    qed
next
    case snoc thus ?case by (fastforce simp add: sums-snoc)
qed simp
lemma pullback-sums-map-middle3:
    map F (sums xs)=ds@[d,e,f]@fs \Longrightarrow
        \existsas a b bs.xs=as@[a,b]@bs ^d=F (sum-list as) }
            e=F(sum-list (as@[a]))^f=F(sum-list (as@[a,b]))
proof (induct xs fs rule: list-induct2-snoc)
    case (Nil2 xs)
    show ?case
    proof (cases xs rule: rev-cases)
        case Nil with Nil2 show ?thesis by simp
    next
        case (snoc ys y)
        with Nil2 have y:map F (sums ys)=ds@[d,e]f=F(sum-list (ys@[y]))
            by (auto simp add: sums-snoc)
            from y(1) obtain as a bs where asabs:
            ys=as@[a]@bs map F (sums as)=ds@[d]
            d=F(sum-list as) e=F(sum-list (as@[a]))
            using pullback-sums-map-middle2[of F ys ds]
            by fastforce
        have bs=[]
        proof-
            from y(1) asabs(1,2) have Suc (length bs)=Suc 0
                    by (auto simp add: sums-append2 map-butlast length-sums[THEN sym])
            thus ?thesis by fast
        qed
        with snoc asabs(1) y(2) have xs =as@[a,y]@[]f=F(sum-list (as@[a,y]))
            by auto
        with asabs(3,4) show ?thesis by fast
    qed
next
    case snoc thus ?case by (fastforce simp add: sums-snoc)
qed simp
```

```
lemma pullback-sums-map-double-middle2:
    assumes map F (sums xs)=ds@[d,e]@es@[f,g]@gs
    shows \existsas a bs b cs.xs=as@[a]@bs@[b]@cs \wedged=F(sum-list as)}
        e=F(sum-list (as@[a]))}\wedgef=F(sum-list (as@[a]@bs))
        g=F(sum-list(as@[a]@bs@[b]))
proof -
    from assms obtain As b cs where Asbcs:
        xs = As@[b]@cs map F (sums As)=ds@[d,e]@es@[f]
        f=F(sum-list As) g=F (sum-list (As@[b]))
        using pullback-sums-map-middle2[of F xs ds@[d,e]@es]
        by fastforce
    from Asbcs show ?thesis
        using pullback-sums-map-middle2[of F As ds d e es@[f]] by fastforce
qed
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 st) $=$
sum-list (if even $n$ then alternating-list $n t s$ else alternating-list $n s t$ )
using uminus-sum-list-order2 set-alternating-list[of $n$ ] rev-alternating-list[of $n s$ s]
by fastforce
context monoid-add
begin
lemma alternating-order2-cancel-1left:
$s+s=0 \Longrightarrow$
sum-list (s \# (alternating-list (Suc n) st)) = sum-list (alternating-list $n t s)$
using add.assoc[of s s] alternating-list-Suc-Cons[of $n$ s $]$ by simp
lemma alternating-order2-cancel-2left:
$s+s=0 \Longrightarrow t+t=0 \Longrightarrow$
sum-list $(t \# s \#($ alternating-list $(S u c(S u c ~ n)) ~ s t))=$ sum-list (alternating-list $n$ st)
using alternating-order2-cancel-1left[of s Suc n] alternating-order2-cancel-1left[of $t n]$
by $\operatorname{simp}$
lemma alternating-order2-even-cancel-right:
assumes st : $s+s=0 t+t=0$
and even-n: even $n$
shows $m \leq n \Longrightarrow$ sum-list (alternating-list $n s t$ @ alternating-list $m t s$ ) $=$ sum-list (alternating-list $(n-m)$ st)
proof (induct $n$ arbitrary: $m$ rule: nat-even-induct, rule even- $n$ )
case (SucSuc $k$ ) with st show ?case
using alternating-order2-cancel-2left[of $t s$ ]

```
    by (cases m rule: nat-cases-2Suc) auto
qed simp
end
```


### 2.6 Conjugation in group-add

### 2.6.1 Abbreviations and basic facts

```
context group-add
begin
abbreviation lconjby :: ' \(a \Rightarrow^{\prime} a \Rightarrow^{\prime} a\) where lconjby \(x y \equiv x+y-x\)
abbreviation rconjby :: ' \(a \Rightarrow^{\prime} a \Rightarrow^{\prime} a\) where rconjby \(x y \equiv-x+y+x\)
lemma lconjby-add: lconjby \((x+y) z=\) lconjby \(x\) (lconjby y \(z\) )
by (auto simp add: algebra-simps)
lemma rconjby-add: rconjby \((x+y) z=\) rconjby \(y\) (rconjby \(x z)\)
by (simp add: minus-add add.assoc[THEN sym])
lemma add-rconjby: rconjby \(x y+\) rconjby \(x z=r\) conjby \(x(y+z)\)
by (simp add: add.assoc)
lemma lconjby-uminus: lconjby \(x(-y)=-\) lconjby \(x\) y
using minus-unique[of lconjby \(x\) y, THEN sym] by (simp add: algebra-simps)
lemma rconjby-uminus: rconjby \(x(-y)=-\) rconjby \(x y\)
using minus-unique[of rconjby \(x y\) ] add-assoc4[of rconjby \(x y-x-y\) d] by simp
lemma lconjby-rconjby: lconjby \(x(\) rconjby \(x y)=y\)
by (simp add: algebra-simps)
lemma rconjby-lconjby: rconjby \(x(\) lconjby \(x y)=y\)
by (simp add: algebra-simps)
lemma lconjby-inj: inj (lconjby x)
using rconjby-lconjby by (fast intro: inj-on-inverseI)
lemma rconjby-inj: inj (rconjby \(x\) )
using lconjby-rconjby by (fast intro: inj-on-inverseI)
lemma lconjby-surj: surj (lconjby x)
using lconjby-rconjby surjI[of lconjby \(x\) ] by fast
lemma lconjby-bij: bij (lconjby x)
unfolding bij-def using lconjby-inj lconjby-surj by fast
```

```
lemma the-inv-lconjby: the-inv (lconjby x) = (rconjby x)
    using bij-betw-f-the-inv-into-f[OF lconjby-bij, of - x] lconjby-rconjby
    by (force intro: inj-onD[OF lconjby-inj, of x])
lemma lconjby-eq-conv-rconjby-eq: w = lconjby x y y = rconjby x w
    using the-inv-lconjby the-inv-into-f-f[OF lconjby-inj] by force
lemma rconjby-orderD: s+s=0\Longrightarrow rconjby x s+rconjby x s=0
    by (simp add: add-rconjby)
lemma rconjby-order2-eq-lconjby:
    assumes }s+s=
    shows rconjby s=lconjby s
proof-
    have rconjby s = lconjby ( }-s\mathrm{ ) by simp
    with assms show ?thesis using minus-unique by simp
qed
lemma lconjby-alternating-list-order2:
    assumes }s+s=0 t+t=
    shows lconjby (sum-list (alternating-list k s t)) (if even k then s else t)=
                sum-list (alternating-list (Suc (2*k)) st)
proof (induct k rule: nat-induct-step2)
    case (SucSuc m)
    have lconjby (sum-list (alternating-list (Suc (Suc m)) s t))
                (if even (Suc (Suc m)) then s else t) =s+t+
                lconjby (sum-list (alternating-list m st)) (if even m then s else t) - t - s
    using alternating-list-SucSuc-ConsCons[of m st]
    by (simp add: algebra-simps)
    also from assms SucSuc
    have ... = sum-list (alternating-list (Suc (2*Suc (Suc m))) s t)
    using alternating-list-SucSuc-ConsCons[of Suc (2*m) st]
                sum-list.append[of alternating-list (Suc (2*Suc m)) st [t]]
    by (simp add: algebra-simps)
    finally show ?case by fast
qed (auto simp add: assms(1) algebra-simps)
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 = 'a list
    where
```

```
    lconjseq [] = []
    | lconjseq (x#xs)=x# (map (lconjby x ) (lconjseq xs))
lemma length-lconjseq: length (lconjseq xs) = length xs
    by (induct xs) auto
lemma lconjseq-snoc:lconjseq (xs@[x])=lconjseq xs @ [lconjby (sum-list xs) x]
    by (induct xs) (auto simp add: lconjby-add)
lemma lconjseq-append:
    lconjseq (xs@ys) = lconjseq xs @ (map (lconjby (sum-list xs)) (lconjseq ys))
proof (induct ys rule: rev-induct)
    case (snoc y ys) thus ?case
        using lconjseq-snoc[of xs@ys] lconjseq-snoc[of ys] by (simp add: lconjby-add)
qed simp
lemma lconjseq-alternating-order2-repeats':
    fixes st:: '}
    defines altst: altst }\equiv\lambdan\mathrm{ . alternating-list n s t
    and altts: altts \equiv \n.alternating-list n t s
    assumes st : s+s=0 t+t=0(s+t)+`k=0
    shows map (lconjby (sum-list (altst k)))
        (lconjseq (if even k then altst m else altts m))=lconjseq (altst m)
proof (induct m)
    case (Suc j)
    with altst altts
        have map (lconjby (sum-list (altst k)))
                (lconjseq (if even k then altst (Suc j) else altts (Suc j))) =
                lconjseq(altst j) @
                [lconjby (sum-list (altst k@ (if even k then altst j else altts j)))
                (if even k then (if even j then s else t) else (if even j then t else s))]
        by (auto simp add: lconjseq-snoc lconjby-add)
    also from altst altts st(1,2)
        have ...= lconjseq (altst j) @ [sum-list (altst (Suc (2*(k+j))))]
        using lconjby-alternating-list-order2[of s t k+j]
        by (cases even k)
            (auto simp add: alternating-list-append[of k])
    finally show ?case using altst st
        by (auto simp add:
                alternating-list-append(1)[THEN sym]
                alternating-sum-list-conv-nataction
                lconjby-alternating-list-order2 lconjseq-snoc
            )
qed (simp add: altst altts)
lemma lconjseq-alternating-order2-repeats:
    fixes st::' 'a and k:: nat
    defines altst: altst \equiv\lambdan. alternating-list n s t
    and altts:altts \equiv\lambdan.alternating-list n t s
```

```
    assumes st: s+s=0 t+t=0 (s+t)+`k=0
    shows lconjseq (altst (2*k)) = lconjseq (altst k)@ lconjseq (altst k)
proof-
    from altst altts
    have lconjseq (altst (2*k)) = lconjseq (altst k) @
                    map (lconjby (sum-list (altst k)))
                    (lconjseq (if even k then altst k else altts k))
    using alternating-list-append[THEN sym, of k kst]
    by (auto simp add: times2-conv-add lconjseq-append)
    with altst altts st show ?thesis
    using lconjseq-alternating-order2-repeats'[of stkk] by auto
qed
lemma even-count-lconjseq-alternating-order2:
    fixes st:: '}
    assumes }s+s=0t+t=0(s+t)+`k=
    shows even (count-list (lconjseq (alternating-list (2*k) st)) x)
proof-
    define xs where xs: xs \equivlconjseq (alternating-list (2*k) st)
    with assms obtain as where xs =as@as
        using lconjseq-alternating-order2-repeats by fast
    hence count-list xs x = 2*(count-list as x)
    by (simp add: times2-conv-add)
    with xs show ?thesis by simp
qed
lemma order2-hd-in-lconjseq-deletion:
    shows s+s=0\Longrightarrows\in set (lconjseq ss)
                            \Longrightarrow\existsasbbs.ss=as@[b]@bs ^ sum-list (s#ss)=sum-list (as@bs)
proof (induct ss arbitrary: s rule: rev-induct)
    case (snoc t ts) show ?case
    proof (cases s \in set (lconjseq ts))
    case True
    with snoc(1,2) obtain as b bs
            where asbbs:ts=as@[b]@bs sum-list (s#ts)=sum-list (as@bs)
            by fastforce
    from asbbs(2) have sum-list (s#ts@[t]) = sum-list (as@(bs@[t]))
            using sum-list.append[of s#ts [t]] sum-list.append[of as@bs [t]] by simp
            with asbbs(1) show ?thesis by fastforce
next
    case False
    with snoc(3) have s: s=lconjby (sum-list ts) t by (simp add:lconjseq-snoc)
    with snoc(2) have t+t=0
        using lconjby-eq-conv-rconjby-eq[of s sum-list ts t]
                rconjby-order2[of s sum-list ts]
        by simp
    moreover from s have sum-list (s#ts@[t])=sum-list ts+t+t
        using add.assoc[of sum-list ts +t - sum-list ts sum-list ts]
        by (simp add: algebra-simps)
```

```
    ultimately have sum-list (s#ts@[t]) = sum-list (ts@[])
    by (simp add: algebra-simps)
    thus ?thesis by fast
qed
qed simp
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 signed = 'a\timesbool
definition signed-funaction :: (' }a\mp@subsup{=}{}{\prime}a\mp@subsup{|}{}{\prime}a)=>'a=>'a signed = 'a signed
    where signed-funaction f s }x\equiv\mathrm{ map-prod (fs) ( }\lambdab.b\not=(fstx=s))
```

    - so the sign of \(x\) is flipped precisely when its first component is equal to \(s\)
    context group-add
begin

```
abbreviation signed-lconjaction \equiv signed-funaction lconjby
abbreviation signed-rconjaction \equiv signed-funaction rconjby
lemmas signed-lconjactionD = signed-funaction-def[of lconjby]
lemmas signed-rconjactionD = signed-funaction-def[of rconjby]
abbreviation signed-lconjpermutation :: ' }a>>'\mp@code{'a signed permutation
    where signed-lconjpermutation s}\equivAbs-permutation (signed-lconjaction s
abbreviation signed-list-lconjaction :: 'a list }=>\mathrm{ ' 'a signed }=>\mathrm{ ' 'a signed
    where signed-list-lconjaction ss \equiv foldr signed-lconjaction ss
lemma signed-lconjaction-fst: fst (signed-lconjaction s x) = lconjby s (fst x)
    using signed-lconjactionD by simp
lemma signed-lconjaction-rconjaction:
    signed-lconjaction s(signed-rconjaction s }x\mathrm{ ) =x
proof-
    obtain }a::'a and b::bool where x = (a,b) by fastforce
    thus ?thesis
        using signed-lconjactionD signed-rconjactionD injD[OF rconjby-inj, of s a]
            lconjby-rconjby[of s a]
```

```
    by auto
qed
lemma signed-rconjaction-by-order2-eq-lconjaction:
    s+s=0\Longrightarrow signed-rconjaction s= signed-lconjaction s
    using signed-funaction-def[of lconjby s] signed-funaction-def[of rconjby s]
        rconjby-order2-eq-lconjby[of s]
    by auto
lemma inj-signed-lconjaction: inj (signed-lconjaction s)
proof (rule injI)
    fix }xy\mathrm{ assume 1: signed-lconjaction s x = signed-lconjaction s y
    moreover obtain a1 a2 :: 'a and b1 b2 :: bool
    where xy: }x=(a1,b1)y=(a2,b2
    by fastforce
    ultimately show }x=
    using injD[OF lconjby-inj, of s a1 a2] signed-lconjactionD
    by (cases a1=s a2=s rule: two-cases) auto
qed
lemma surj-signed-lconjaction: surj (signed-lconjaction s)
    using signed-lconjaction-rconjaction[THEN sym] by fast
lemma bij-signed-lconjaction: bij (signed-lconjaction s)
    using inj-signed-lconjaction surj-signed-lconjaction by (fast intro: bijI)
lemma the-inv-signed-lconjaction:
    the-inv (signed-lconjaction s)= signed-rconjaction s
proof
    fix }
    show the-inv (signed-lconjaction s) x = signed-rconjaction s x
    proof (rule the-inv-into-f-eq, rule inj-signed-lconjaction)
        show signed-lconjaction s(signed-rconjaction s x) =x
            using signed-lconjaction-rconjaction by fast
    qed (simp add: surj-signed-lconjaction)
qed
lemma the-inv-signed-lconjaction-by-order2:
    s+s=0\Longrightarrow the-inv (signed-lconjaction s) = signed-lconjaction s
    using the-inv-signed-lconjaction signed-rconjaction-by-order2-eq-lconjaction
    by simp
lemma signed-list-lconjaction-fst:
    fst (signed-list-lconjaction ss x) = lconjby (sum-list ss) (fst x)
    using signed-lconjaction-fst lconjby-add by (induct ss) auto
lemma signed-list-lconjaction-snd:
    shows }\foralls\in\mathrm{ set ss. s+s=0 ב snd (signed-list-lconjaction ss x)
        = (if even (count-list (lconjseq (rev ss)) (fst x)) then snd x else \negsnd x)
```

```
proof (induct ss)
    case (Cons s ss) hence prevcase:
        snd (signed-list-lconjaction ss }x\mathrm{ ) =
            (if even (count-list (lconjseq (rev ss)) (fst x)) then snd x else }\neg\mathrm{ snd x)
        by simp
    have 1: snd (signed-list-lconjaction (s# ss) x)=
                snd (signed-lconjaction s (signed-list-lconjaction ss x))
    by simp
    show ?case
    proof (cases fst (signed-list-lconjaction ss x)=s)
        case True
        with 1 prevcase
            have snd (signed-list-lconjaction (s# ss) x)=
                                    (if even (count-list (lconjseq (rev ss)) (fst x)) then }\neg\mathrm{ snd }x\mathrm{ else snd }x\mathrm{ )
            by (simp add: signed-lconjactionD)
    with True Cons(2) rconjby-lconjby show ?thesis
            by (auto simp add: signed-list-lconjaction-fst lconjseq-snoc
                simp flip: uminus-sum-list-order2
                    )
    next
        case False
        hence rconjby (sum-list ss) (lconjby (sum-list ss) (fst x)) =
                rconjby (sum-list ss) s
            by (simp add: signed-list-lconjaction-fst)
        with Cons(2)
            have count-list (lconjseq (rev (s#ss))) (fst x)=
                                    count-list (lconjseq (rev ss)) (fst x)
            by (simp add:
                rconjby-lconjby uminus-sum-list-order2[THEN sym]
                        lconjseq-snoc
                )
        moreover from False 1 prevcase
            have snd (signed-list-lconjaction (s # ss) x)=
                (if even (count-list (lconjseq (rev ss)) (fst x)) then snd x else ᄀ snd x)
            by (simp add: signed-lconjactionD)
            ultimately show ?thesis by simp
    qed
qed simp
end
```


### 2.7 Cosets

### 2.7.1 Basic facts

lemma set-zero-plus ${ }^{\prime}[$ simp $]$ : ( $0::{ }^{\prime} a::$ 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.
by (auto simp add: elt-set-plus-def)
lemma lcoset- $0:\left(w::^{\prime} a:: m o n o i d-a d d\right)+o \quad 0=\{w\}$ using elt-set-plus-def $[o f w]$ by simp
lemma lcoset-refl: ( $\left.0::^{\prime} a:: m o n o i d-a d d\right) \in A \Longrightarrow a \in a+o A$ using elt-set-plus-def by force
lemma lcoset-eq-reps-subset:
( $a:$ :' $a::$ group-add $)+o ~ A \subseteq a+o B \Longrightarrow A \subseteq B$
using elt-set-plus-def[of a] by auto
lemma lcoset-eq-reps: ( $a::^{\prime} a::$ group-add $)+o A=a+o B \Longrightarrow A=B$
using lcoset-eq-reps-subset[of a $A \quad B]$ lcoset-eq-reps-subset $[o f ~ a ~ B ~ A] ~ b y ~ a u t o ~$
lemma lcoset-inj-on: inj $\left((+o)\left(a::^{\prime} a:: g r o u p-a d d\right)\right)$
using lcoset-eq-reps inj-onI[of UNIV (+o) a] by auto
lemma lcoset-conv-set: ( $a::{ }^{\prime} g::$ group-add $) \in b+o A \Longrightarrow-b+a \in A$ by (auto simp add: elt-set-plus-def)


### 2.7.2 The supset order on cosets

```
lemma supset-lbound-lcoset-shift:
    supset-lbound-of X Y B \(\Longrightarrow\)
    ordering.lbound-of (〇) \((a+o X)(a+o Y)(a+o B)\)
    using ordering.lbound-of-def[OF supset-poset, of X Y B]
    by (fast intro: ordering.lbound-ofI supset-poset)
lemma supset-glbound-in-of-lcoset-shift:
    fixes \(\quad P::\) ' \(a::\) group-add set set
    assumes supset-glbound-in-of P X Y B
    shows supset-glbound-in-of \(\left((+o) a^{\prime} P\right)(a+o X)(a+o Y)(a+o B)\)
    using ordering.glbound-in-ofD-in[OF supset-poset, OF assms]
        ordering.glbound-in-ofD-lbound[OF supset-poset, OF assms]
    supset-lbound-lcoset-shift \(\left[\begin{array}{llll}\text { of } & Y & B & a\end{array}\right]\)
    supset-lbound-lcoset-shift[of \(a+o X a+o Y--a]\)
    ordering.glbound-in-ofD-glbound[OF supset-poset, OF assms]
    ordering.glbound-in-ofI [
    OF supset-poset, of \(a+o B(+o) a{ }^{\prime} P a+o X a+o Y\)
    ]
    by (fastforce simp add: set-plus-rearrange2)
```


### 2.7.3 The afforded partition

definition lcoset-rel :: ' $a::\{$ uminus,plus $\}$ set $\Rightarrow\left({ }^{\prime} a \times^{\prime} a\right)$ 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 $A$ using lcoset-rel-def by fast

### 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}\not={
    and diff-closed: \gh. g\inG\Longrightarrowh\inG\Longrightarrowg-h\inG
begin
abbreviation Subgroup :: 'g set => bool
    where Subgroup H}\equiv\mathrm{ Group }H\wedgeH\subseteq
lemma SubgroupD1: Subgroup H C Group H by fast
lemma zero-closed:0\inG
proof-
    from nonempty obtain g}\mathrm{ where g}GG\mathrm{ by fast
    hence g-g\inG using diff-closed by fast
    thus?thesis by simp
qed
lemma uminus-closed: }g\inG\Longrightarrow-g\in
    using zero-closed diff-closed[of 0 g] by simp
lemma add-closed: g\inG\Longrightarrowh\inG\Longrightarrowg+h\inG
    using uminus-closed[of h] diff-closed[of g -h] by simp
lemma uminus-add-closed: }g\inG\Longrightarrowh\inG\Longrightarrow-g+h\in
    using uminus-closed add-closed by fast
lemma lconjby-closed: g\inG\Longrightarrowx\inG\Longrightarrow lconjby g }x\in
    using add-closed diff-closed by fast
lemma lconjby-set-closed: g\inG\LongrightarrowA\subseteqG\Longrightarrowlconjby g ' }A\subseteq
    using lconjby-closed by fast
lemma set-lconjby-subset-closed:
    H\subseteqG\LongrightarrowA\subseteqG\Longrightarrow(\bigcuph\inH.lconjby h'A)\subseteqG
    using lconjby-set-closed[of - A] by fast
lemma sum-list-map-closed: set (map f as)\subseteqG\Longrightarrow(\suma\leftarrowas.f a) \inG
    using zero-closed add-closed by (induct as) auto
lemma sum-list-closed: set as \subseteqG\Longrightarrow sum-list as }\in
    using sum-list-map-closed by force
```

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 \(G \quad::\) ' \(a\) set
    and binop :: ' \(a \Rightarrow^{\prime} a \Rightarrow^{\prime} a\)
    and \(e \quad::{ }^{\prime} a\)
    assumes closed \(: g \in G \Longrightarrow h \in G \Longrightarrow\) binop \(g h \in G\)
    and assoc :
    \(\llbracket g \in G ; h \in G ; k \in G \rrbracket \Longrightarrow\) binop (binop \(g h) k=\) binop \(g(\) binop \(h k)\)
    and identity: \(e \in G g \in G \Longrightarrow\) binop \(g\) e \(=g g \in G \Longrightarrow\) binop e \(g=g\)
    and inverses: \(g \in G \Longrightarrow \exists h \in G\). binop \(g h=e \wedge\) binop \(h g=e\)
begin
lemma unique-identity1: \(g \in G \Longrightarrow \forall x \in G\). binop \(g x=x \Longrightarrow g=e\)
    using identity \((1,2)\) by auto
lemma unique-inverse:
    assumes \(g \in G\)
    shows \(\exists!h . h \in G \wedge\) binop \(g h=e \wedge\) binop \(h g=e\)
proof (rule ex-ex1I)
    from assms show \(\exists h . h \in G \wedge\) binop \(g h=e \wedge\) binop \(h g=e\)
        using inverses by fast
next
    fix \(h k\)
    assume \(h \in G \wedge\) binop \(g h=e \wedge\) binop \(h g=e k \in G \wedge\)
                binop \(g k=e \wedge\) binop \(k g=e\)
    hence \(h\) : \(h \in G\) binop \(g h=e\) binop \(h g=e\)
        and \(k: k \in G\) binop \(g k=e\) binop \(k g=e\)
        by auto
    from assms \(h(1,3) k(1,2)\) show \(h=k\) using identity \((2,3)\) assoc by force
qed
abbreviation \(G\)-perm \(g \equiv\) restrict1 (binop \(g\) ) \(G\)
definition Abs-G-perm :: ' \(a \Rightarrow\) 'a permutation
    where Abs-G-perm \(g \equiv\) Abs-permutation ( \(G\)-perm \(g\) )
abbreviation \(\mathfrak{p} \equiv A b s\) - \(G\)-perm - the injection into type permutation
```

abbreviation $\mathfrak{i p} \equiv$ the-inv-into $G \mathfrak{p}$ - the reverse correspondence abbreviation $p G \equiv \mathfrak{p}^{`} G$ - the resulting Group of type permutation
lemma $G$-perm-comp:
$g \in G \Longrightarrow h \in G \Longrightarrow G$-perm $g \circ G$-perm $h=G$-perm (binop $g h$ )
using closed by (auto simp add: assoc)
definition the-inverse :: ' $a \Rightarrow{ }^{\prime} a$
where the-inverse $g \equiv(T H E h . h \in G \wedge$ binop $g h=e \wedge$ binop $h g=e)$
abbreviation $\mathfrak{i} \equiv$ the-inverse
lemma the-inverseD:
assumes $g \in G$
shows $\quad \mathfrak{i} g \in G$ binop $g(\mathfrak{i} g)=e$ binop $(\mathfrak{i} g) g=e$
using assms the I' [OF unique-inverse]
unfolding the-inverse-def
by auto
lemma binop- $G$-comp-binop- $\mathfrak{i} G: g \in G \Longrightarrow x \in G \Longrightarrow$ binop $g$ (binop $(\mathfrak{i} g) x)=x$ using the-inverse $D$ (1) assoc $[$ of $g \mathfrak{i} g x]$ by (simp add: identity(3) the-inverseD(2))
lemma bij-betw-binop- $G$ :
assumes $g \in G$
shows bij-betw (binop g) G G
unfolding bij-betw-def
proof
show inj-on (binop g) $G$
proof (rule inj-onI)
fix $h k$ assume $h k$ : $h \in G k \in G$ binop $g h=$ binop $g k$
with assms have binop (binop (ig) g) $h=\operatorname{binop}(\operatorname{binop}(\mathfrak{i} g) g$ ) $k$ using the-inverse $D$ (1) by (simp add: assoc)
with assms $h k(1,2)$ show $h=k$ using the-inverse $D$ (3) identity by simp
qed
show binop $g{ }^{\prime} G=G$
proof
from assms show binop $g$ ' $G \subseteq G$ using closed by fast
from assms show binop $g$ ' $G \supseteq G$
using binop-G-comp-binop-i $G[T H E N$ sym $]$ the-inverseD(1) closed by fast qed
qed
lemma the-inv-into-G-binop-G:
assumes $g \in G x \in G$
shows the-inv-into $G$ (binop $g$ ) $x=$ binop $(\mathfrak{i} g) x$
proof (rule the-inv-into-f-eq)
from $\operatorname{assms}(1)$ show inj-on (binop $g$ ) $G$
using bij-betw-imp-inj-on[OF bij-betw-binop-G] by fast
from assms show binop $g$ (binop $(\mathfrak{i} g) x)=x$
using binop-G-comp-binop-i $G$ by fast
from assms show binop (i $g$ ) $x \in G$ using closed the-inverse $D(1)$ by fast qed
lemma restrict1-the-inv-into-G-binop- $G$ :
$g \in G \Longrightarrow$ restrict1 (the-inv-into $G$ (binop g)) $G=G$-perm (ig)
using the-inv-into-G-binop- $G$ by auto
lemma bij-G-perm: $g \in G \Longrightarrow b i j(G$-perm $g)$
using set-permutation-bij-restrict1 bij-betw-binop- $G$ by fast
lemma $G$-perm-apply: $g \in G \Longrightarrow x \in G \Longrightarrow \mathfrak{p} g \rightarrow x=$ binop $g x$ using Abs-G-perm-def Abs-permutation-inverse bij-G-perm by fastforce
lemma $G$-perm-apply-identity: $g \in G \Longrightarrow \mathfrak{p} g \rightarrow e=g$ using $G$-perm-apply identity $(1,2)$ by simp
lemma the-inv- $G$-perm:
$g \in G \Longrightarrow$ the-inv $(G$-perm $g)=G$-perm $(\mathfrak{i} g)$
using set-permutation-the-inv-restrict1 bij-betw-binop- $G$
restrict1-the-inv-into-G-binop-G
by fastforce
lemma Abs-G-perm-diff:
$g \in G \Longrightarrow h \in G \Longrightarrow \mathfrak{p} g-\mathfrak{p} h=\mathfrak{p}$ (binop $g(i) h)$ )
using Abs-G-perm-def minus-permutation-abs-eq[OF bij-G-perm bij-G-perm] the-inv-G-perm G-perm-comp the-inverseD (1)
by simp
lemma Group: Group $p G$
using identity (1) Abs-G-perm-diff the-inverseD(1) closed by unfold-locales auto
lemma inj-on-p-G: inj-on $\mathfrak{p ~} G$
proof (rule inj-onI)
fix $x y$ assume $x y$ : $x \in G y \in G \mathfrak{p} x=\mathfrak{p} y$
hence Abs-permutation $(G$-perm (binop $x(\mathfrak{i} y))$ ) Abs-permutation id using Abs-G-perm-diff Abs-G-perm-def
by (fastforce simp add: zero-permutation.abs-eq)
moreover from $x y(1,2)$ have 1 : binop $x(\mathfrak{i} y) \in G$
using bij-id closed the-inverseD (1) by fast
ultimately have 2: $G$-perm (binop $x(\mathfrak{i} y)$ ) $=$ id
using Abs-permutation-inject[of G-perm (binop x (i $y$ ))] bij-G-perm bij-id
by simp
have $\forall z \in G$. binop (binop $x(\mathfrak{i} y)) z=z$
proof
fix $z$ assume $z \in G$
thus binop (binop $x(\mathfrak{i} y)$ ) $z=z$ using fun-cong[OF 2, of $z]$ by simp
qed
with $x y(1,2)$ have binop $x$ (binop $(\mathfrak{i} y) y)=y$
using unique-identity1[OF 1] the-inverseD(1) by (simp add: assoc) with $x y(1,2)$ show $x=y$ using the-inverseD(3) identity(2) by simp qed
lemma homs:
$\bigwedge g h . g \in G \Longrightarrow h \in G \Longrightarrow \mathfrak{p}($ binop $g h)=\mathfrak{p} g+\mathfrak{p} h$
$\bigwedge x y . x \in p G \Longrightarrow y \in p G \Longrightarrow$ binop (ip $x)(\mathfrak{i p} y)=\mathfrak{i p}(x+y)$

## proof-

show 1: $\bigwedge g h . g \in G \Longrightarrow h \in G \Longrightarrow \mathfrak{p}($ binop $g h)=\mathfrak{p} g+\mathfrak{p} h$
using $A b s$-G-perm-def G-perm-comp
plus-permutation-abs-eq[OF bij-G-perm bij-G-perm]
by simp
show $\bigwedge x y . x \in p G \Longrightarrow y \in p G \Longrightarrow$ binop $(\mathfrak{i p} x)(\mathfrak{i p} y)=\mathfrak{i p}(x+y)$
proof-
fix $x y$ assume $x \in p G y \in p G$
moreover hence $\mathfrak{i p}(\mathfrak{p}($ binop $(\mathfrak{i p} x)(\mathfrak{i p} y)))=\mathfrak{i p}(x+y)$
using 1 the-inv-into-into $[O F$ inj-on-p- $G] f$-the-inv-into-f $[O F$ inj-on-p- $G]$
by $\operatorname{simp}$
ultimately show binop (ip $x$ ) $(\mathfrak{i p} y)=\mathfrak{i p}(x+y)$
using the-inv-into-into[OF inj-on-p-G] closed the-inv-into-f-f[OF inj-on-p-G]
by $\operatorname{simp}$
qed
qed
lemmas inv-correspondence-into $=$
the-inv-into-into[OF inj-on-p-G, of - G, simplified]
lemma inv-correspondence-conv-apply: $x \in p G \Longrightarrow \mathfrak{i p} x=x \rightarrow e$
using $G$-perm-apply-identity inj-on-p- $G$ by (auto intro: the-inv-into-f-eq)
end

### 2.8.3 Cosets of a Group

## context Group

begin
lemma lcoset-refl: $a \in a+o G$ using lcoset-refl zero-closed by fast
lemma lcoset-el-reduce:
assumes $a \in G$
shows $a+o G=G$
proof (rule seteqI)
fix $x$ assume $x \in a+o G$
from this obtain $g$ where $g \in G x=a+g$ using elt-set-plus-def $[o f a]$ by auto
with assms show $x \in G$ by (simp add: add-closed)
next
fix $x$ assume $x \in G$
with assms have $-a+x \in G$ by (simp add: uminus-add-closed) thus $x \in a+o G$ using elt-set-plus-def by force qed
lemma lcoset-el-reduce0: $0 \in a+o G \Longrightarrow a+o G=G$
using elt-set-plus-def[of a $G$ ] minus-unique uminus-closed $[$ of $-a]$ lcoset-el-reduce
by fastforce
lemma lcoset-subgroup-imp-eq-reps:
Group $H \Longrightarrow w+o H \subseteq w^{\prime}+o G \Longrightarrow w^{\prime}+o G=w+o G$
using Group.lcoset-refl[of H w] lcoset-conv-set[of w] lcoset-el-reduce set-plus-rearrange2 $\left[\right.$ of $\left.w^{\prime}-w^{\prime}+w G\right]$
by force
lemma lcoset-closed: $a \in G \Longrightarrow A \subseteq G \Longrightarrow a+o A \subseteq G$
using elt-set-plus-def[of a] add-closed by auto
lemma lcoset-rel-sym: sym (lcoset-rel $G$ )
proof (rule symI)
fix $a b$ show $(a, b) \in$ lcoset-rel $G \Longrightarrow(b, a) \in$ lcoset-rel $G$
using uminus-closed minus-add[of -a b] lcoset-rel-def[of $G]$ by fastforce
qed
lemma lcoset-rel-trans: trans (lcoset-rel G)
proof (rule transI)
fix $x y z$ assume $x y:(x, y) \in l$ coset-rel $G$ and $y z:(y, z) \in$ lcoset-rel $G$
from this obtain $g g^{\prime}$ where $g \in G-x+y=g g^{\prime} \in G-y+z=g^{\prime}$ using lcoset-rel-def[of $G$ ] by fast
thus $(x, z) \in$ lcoset-rel $G$
using add.assoc[of $g-y z]$ add-closed lcoset-rel-def[of $G]$ by auto
qed
abbreviation LCoset-rel :: 'g set $\Rightarrow\left({ }^{\prime} g \times{ }^{\prime} g\right)$ 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 $H$ )
using lcoset-rel-def by (fastforce intro: refl-onI)
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 $\quad=$ quotientE[of - G LCoset-rel -]
lemma lcoset-subgroup-rel-equiv:
Subgroup $H \Longrightarrow$ equiv $G$ (LCoset-rel $H$ )
using Group.lcoset-rel-sym sym-sym sym-Int Group.lcoset-rel-trans trans-sym trans-Int subgroup-refl-on-LCoset-rel
by (blast intro: equivI)

```
lemma trivial-LCoset: H\subseteqG\LongrightarrowH=LCoset-rel H" {0}
    using zero-closed unfolding lcoset-rel-def by auto
end
```


### 2.8.4 The Group generated by a set

```
inductive-set genby :: 'a::group-add set => ' }a\mathrm{ set (<->)
```

    for \(S\) :: 'a set
    where
        genby- 0 -closed \(: 0 \in\langle S\rangle\) - just in case \(S\) is empty
        | genby-genset-closed: \(s \in S \Longrightarrow s \in\langle S\rangle\)
        \(\mid\) genby-diff-closed \(: w \in\langle S\rangle \Longrightarrow w^{\prime} \in\langle S\rangle \Longrightarrow w-w^{\prime} \in\langle S\rangle\)
    lemma genby-Group: Group $\langle S\rangle$
using genby-0-closed genby-diff-closed by unfold-locales fast
lemmas genby-uminus-closed $=$ Group.uminus-closed $\quad$ [OF genby-Group]
lemmas genby-add-closed $\quad=$ Group.add-closed $\quad[$ OF genby-Group]
lemmas genby-uminus-add-closed $=$ Group.uminus-add-closed [OF genby-Group]
lemmas genby-lcoset-refl $=$ Group.lcoset-refl $\quad$ [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 $\quad[$ 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: $S \subseteq\langle S\rangle$
using genby-genset-closed by fast
lemma genby-uminus-genset-subset: uminus ' $S \subseteq\langle S\rangle$
using genby-genset-subset genby-uminus-closed by auto
lemma genby-in-sum-list-lists:
fixes $S$
defines $S$-sum-lists: $S$-sum-lists $\equiv(\bigcup$ ss $\in$ lists $(S \cup$ uminus' $S)$. $\{$ sum-list ss $\})$
shows $\quad w \in\langle S\rangle \Longrightarrow w \in S$-sum-lists
proof (erule genby.induct)
have $0=$ sum-list [] by simp
with $S$-sum-lists show $0 \in S$-sum-lists by blast
next
fix $s$ assume $s \in S$
hence $[s] \in$ lists $(S \cup$ uminus ' $S$ ) by simp
moreover have $s=$ sum-list [ $s$ ] by simp
ultimately show $s \in S$-sum-lists using $S$-sum-lists by blast
next
fix $w w^{\prime}$ assume $w w^{\prime}: w \in S$-sum-lists $w^{\prime} \in S$-sum-lists

```
    with S-sum-lists obtain ss ts
        where ss: ss }\in\mathrm{ lists ( }S\cup\mathrm{ uminus' 'S) w = sum-list ss
```



```
        by fastforce
    from ss(2) ts(2) have w-w' = sum-list (ss @ map uminus (rev ts))
        by (simp add: diff-conv-add-uminus uminus-sum-list)
    moreover from ss(1) ts(1)
        have ss @ map uminus (rev ts) \in lists (S\cupuminus' }S\mathrm{ )
        by fastforce
    ultimately show w- w'\inS-sum-lists using S-sum-lists by fast
qed
lemma sum-list-lists-in-genby: ss }\in\mathrm{ lists (S U uminus'}S)\Longrightarrow\mathrm{ sum-list ss }\in\langleS
proof (induct ss)
    case Nil show ?case using genby-0-closed by simp
next
    case (Cons s ss) thus ?case
        using genby-genset-subset[of S] genby-uminus-genset-subset
            genby-add-closed[of s S sum-list ss]
        by auto
qed
lemma sum-list-lists-in-genby-sym:
        uminus ' }S\subseteqS\Longrightarrow\mathrm{ ss }\in\mathrm{ lists }S\Longrightarrow\mathrm{ sum-list ss }\in\langleS
    using sum-list-lists-in-genby by fast
lemma genby-eq-sum-lists: }\langleS\rangle=(\bigcup\mathrm{ ssєlists (S U uminus` S). {sum-list ss})
    using genby-in-sum-list-lists sum-list-lists-in-genby by fast
lemma genby-mono: T\subseteqS\Longrightarrow\langleT\rangle\subseteq\langleS\rangle
    using genby-eq-sum-lists[of T] genby-eq-sum-lists[of S] by force
lemma (in Group) genby-closed:
    assumes S\subseteqG
    shows }\langleS\rangle\subseteq
proof
    fix }x\mathrm{ show }x\in\langleS\rangle\Longrightarrowx\in
    proof (erule genby.induct, rule zero-closed)
        from assms show \}\s.s\inS\Longrightarrows\inG by fas
        show }\bigwedgew\mp@subsup{w}{}{\prime}.w\inG\Longrightarrow\mp@subsup{w}{}{\prime}\inG\Longrightarroww-\mp@subsup{w}{}{\prime}\inG\mathrm{ using diff-closed by fast
    qed
qed
lemma (in Group) genby-subgroup: S\subseteqG\Longrightarrow Subgroup }\langleS
    using genby-closed genby-Group by simp
lemma genby-sym-eq-sum-lists:
    uminus' }S\subseteqS\Longrightarrow\langleS\rangle=(\bigcupss\inlists S. {sum-list ss}
    using lists-mono genby-eq-sum-lists[of S] by force
```

```
lemma genby-empty': \(w \in\langle\}\rangle \Longrightarrow w=0\)
proof (erule genby.induct) qed auto
lemma genby-order2':
    assumes \(s+s=0\)
    shows \(\quad w \in\langle\{s\}\rangle \Longrightarrow w=0 \vee w=s\)
proof (erule genby.induct)
    fix \(w w^{\prime}\) assume \(w=0 \vee w=s w^{\prime}=0 \vee w^{\prime}=s\)
    with assms show \(w-w^{\prime}=0 \vee w-w^{\prime}=s\)
        by (cases \(w^{\prime}=0\) ) (auto simp add: minus-unique)
qed auto
lemma genby-order2: \(s+s=0 \Longrightarrow\langle\{s\}\rangle=\{0, s\}\)
    using genby-order2'[of s] genby-0-closed genby-genset-closed by auto
lemma genby-empty: \(\langle\}\rangle=0\)
    using genby-empty' genby-0-closed by auto
lemma genby-lcoset-order2: \(s+s=0 \Longrightarrow w+o\langle\{s\}\rangle=\{w, w+s\}\)
    using elt-set-plus-def \([\) of \(w]\) by (auto simp add: genby-order2)
lemma genby-lcoset-empty: (w::'a::group-add) \(+o\langle\{ \}\rangle=\{w\}\)
proof-
    have \(\langle\}:: ' a\) set \(\rangle=(0:: ' a\) set \()\) using genby-empty by fast
    thus? ?thesis using lcoset-0 by simp
qed
lemma (in Group) genby-set-lconjby-set-lconjby-closed:
    fixes \(A::\) ' \(g\) set
    defines \(S \equiv(\bigcup g \in G\). lconjby g ' \(A\) )
    assumes \(g \in G\)
    shows \(\quad x \in\langle S\rangle \Longrightarrow\) lconjby g \(x \in\langle S\rangle\)
proof (erule genby.induct)
    show lconjby g \(0 \in\langle S\rangle\) using genby-0-closed by simp
    from assms show \(\wedge s . s \in S \Longrightarrow\) lconjby \(g s \in\langle S\rangle\)
        using add-closed genby-genset-closed \([\) of - \(S\) ] by (force simp add: lconjby-add)
next
    fix \(w w^{\prime}\)
    assume \(w w^{\prime}\) : lconjby g \(w \in\langle S\rangle\) lconjby \(g w^{\prime} \in\langle S\rangle\)
    have lconjby \(g\left(w-w^{\prime}\right)=\) lconjby \(g w+l\) lconjby \(g\left(-w^{\prime}\right)\)
    by (simp add: algebra-simps)
    with \(w w^{\prime}\) show lconjby \(g\left(w-w^{\prime}\right) \in\langle S\rangle\)
    using lconjby-uminus[of g] diff-conv-add-uminus[of - lconjby g w]
                genby-diff-closed
    by fastforce
qed
lemma (in Group) genby-set-lconjby-set-rconjby-closed:
```

fixes $A::$ ' $g$ set
defines $S \equiv\left(\bigcup g \in G\right.$. lconjby $\left.g{ }^{\prime} A\right)$
assumes $g \in G x \in\langle S\rangle$
shows rconjby $g x \in\langle S\rangle$
using assms uminus-closed genby-set-lconjby-set-lconjby-closed
by fastforce

### 2.8.5 Homomorphisms and isomorphisms

locale GroupHom $=$ Group $G$
for $G::$ ' $g$ ::group-add set

+ fixes $T::{ }^{\prime} g \Rightarrow{ }^{\prime} h::$ group-add
assumes hom: $g \in G \Longrightarrow g^{\prime} \in G \Longrightarrow T\left(g+g^{\prime}\right)=T g+T g^{\prime}$
and supp: supp $T \subseteq G$
begin
lemma im-zero: $T 0=0$
using zero-closed hom[of 0 0 $]$ add-diff-cancel[ of T 0 T 0] by simp
lemma im-uminus: $T(-g)=-T g$
using im-zero hom $[$ of $g-g]$ uminus-closed $[$ of $g]$ minus-unique[of $T g$ ] uminus-closed $[$ of $-g]$ supp suppI-contra $[$ of $g T]$ suppI-contra[of $-g T]$
by fastforce
lemma im-uminus-add: $g \in G \Longrightarrow g^{\prime} \in G \Longrightarrow T\left(-g+g^{\prime}\right)=-T g+T g^{\prime}$ by (simp add: uminus-closed hom im-uminus)
lemma im-diff: $g \in G \Longrightarrow g^{\prime} \in G \Longrightarrow T\left(g-g^{\prime}\right)=T g-T g^{\prime}$ using hom uminus-closed hom[of $g-g$ ] im-uminus by simp
lemma im-lconjby: $x \in G \Longrightarrow g \in G \Longrightarrow T$ (lconjby $x g)=$ lconjby $(T x)(T g)$ using add-closed by (simp add: im-diff hom)
lemma im-sum-list-map:
set $(\operatorname{map} f a s) \subseteq G \Longrightarrow T\left(\sum a \leftarrow a s . f a\right)=\left(\sum a \leftarrow a s . T(f a)\right)$
using hom im-zero sum-list-closed by (induct as) auto
lemma comp:
assumes GroupHom H S T‘G $\subseteq H$
shows GroupHom $G(S \circ T)$
proof
fix $g g^{\prime}$ assume $g \in G g^{\prime} \in G$
with hom assms(2) show $(S \circ T)\left(g+g^{\prime}\right)=(S \circ T) g+(S \circ T) g^{\prime}$
using GroupHom.hom[OF assms(1)] by fastforce
next
from supp have $\bigwedge g . g \notin G \Longrightarrow(S \circ T) g=0$
using suppI-contra GroupHom.im-zero[OF assms(1)] by fastforce
thus supp $(S \circ T) \subseteq G$ using suppD-contra by fast


## qed

end

```
definition ker \(::\left({ }^{\prime} a \Rightarrow^{\prime} b::\right.\) zero \() \Rightarrow^{\prime} a\) set
    where \(\operatorname{ker} f=\{a . f a=0\}\)
lemma ker-subset-ker-restrict0: \(\operatorname{ker} f \subseteq \operatorname{ker}(\) restrict0 f \(A\) )
    unfolding ker-def by auto
context GroupHom
begin
abbreviation \(\operatorname{Ker} \equiv \operatorname{ker} T \cap G\)
lemma uminus-add-in-Ker-eq-eq-im:
    \(g \in G \Longrightarrow h \in G \Longrightarrow(-g+h \in \operatorname{Ker})=(T g=T h)\)
    using neg-equal-iff-equal
    by (simp add: uminus-add-closed ker-def im-uminus-add eq-neg-iff-add-eq-0)
end
locale UGroupHom \(=\) GroupHom UNIV T
    for \(T::\) 'g::group-add \(\Rightarrow\) 'h::group-add
begin
lemmas im-zero \(=\) im-zero
lemmas im-uminus \(=\) im-uminus
lemma hom: \(T\left(g+g^{\prime}\right)=T g+T g^{\prime}\)
    using hom by simp
lemma im-diff: \(T\left(g-g^{\prime}\right)=T g-T g^{\prime}\)
    using im-diff by simp
lemma im-lconjby: \(T\) (lconjby \(x g)=l c o n j b y ~(T x)(T g)\)
    using im-lconjby by simp
lemma restrict0:
    assumes Group \(G\)
    shows GroupHom \(G\) (restrict0 \(T G\) )
proof (intro-locales, rule assms, unfold-locales)
    from hom
        show \(\bigwedge g g^{\prime} . g \in G \Longrightarrow g^{\prime} \in G \Longrightarrow\)
            restrict0 \(T G\left(g+g^{\prime}\right)=\) restrict0 \(T G g+\) restrict0 \(T G g^{\prime}\)
        using Group.add-closed[OF assms]
        by auto
    show supp \((\) restrict0 \(T G) \subseteq G\) using supp-restrict0 \([\) of \(G T]\) by fast
```


## qed

end
lemma UGroupHomI:
assumes $\bigwedge g g^{\prime} . T\left(g+g^{\prime}\right)=T g+T g^{\prime}$
shows UGroupHom $T$
using assms
by unfold-locales auto
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
proof (unfold-locales, rule inj-onI)
fix $x y$ from assms show $\llbracket x \in G ; y \in G ; T x=T y \rrbracket \Longrightarrow x=y$ using im-diff diff-closed by force
qed
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 $T \equiv \operatorname{restrict0~(~} T \circ \mathfrak{i p ) ~ p G}$

```
lemma (in BinOpSetGroup) lift-hom:
    fixes \(T::{ }^{\prime} a \Rightarrow\) '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)
proof (intro-locales, rule Group, unfold-locales)
    from assms
        show \(\bigwedge x y . x \in p G \Longrightarrow y \in p G \Longrightarrow\)
            lift-hom \(T(x+y)=\) lift-hom \(T x+\) lift-hom \(T y\)
        using Group.add-closed[OF Group] inv-correspondence-into
        by (simp add: homs(2)[THEN sym])
qed (rule supp-restrict0)
```


### 2.8.6 Normal subgroups

definition rcoset-rel :: ' $a::\{$ minus,plus $\}$ set $\Rightarrow\left({ }^{\prime} a \times{ }^{\prime} a\right)$ 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 $G)$

```
proof (rule set-eqI)
    fix x :: ' }g\times'
    obtain a b where ab: x=(a,b) by fastforce
    hence (x\in rcoset-rel G) = (a-b\inG) using rcoset-rel-def by auto
    also have ... = ( (-b,-a) \in lcoset-rel G)
        using uminus-closed lcoset-rel-def by fastforce
    finally
        show (x rcoset-rel G)=(x\in map-prod uminus uminus'(lcoset-rel G))
        using ab symD[OF lcoset-rel-sym] map-prod-def
        by force
qed
lemma rcoset-rel-sym: sym (rcoset-rel G)
    using rcoset-rel-conv-lcoset-rel map-prod-sym lcoset-rel-sym by simp
abbreviation RCoset-rel :: 'g set => ('g\times'g) set
    where RCoset-rel H}\equiv\mathrm{ rcoset-rel H}\cap(G\timesG
definition normal :: 'g set }=>\mathrm{ bool
    where normal H}\equiv(\forallg\inG. LCoset-rel H" {g}=RCoset-rel H" {g}
lemma normalI:
    assumes Group H}\forallg\inG.\forallh\inH.\exists\mp@subsup{h}{}{\prime}\inH.g+h=\mp@subsup{h}{}{\prime}+
            \forallg\inG.}\forallh\inH.\exists\mp@subsup{h}{}{\prime}\inH.h+g=g+\mp@subsup{h}{}{\prime
    shows normal H
    unfolding normal-def
proof
    fix g}\mathrm{ assume g: gGG
    show LCoset-rel H" {g}=RCoset-rel H" {g}
    proof (rule seteqI)
        fix }x\mathrm{ assume }x\inL\mathrm{ Coset-rel H" {g}
        with g have x: x\inG-g+x\inH unfolding lcoset-rel-def by auto
        from gx(2) assms(2) obtain h}\mathrm{ where h: hfHg-x= -h
        by (fastforce simp add: algebra-simps)
        with assms(1) gx(1) show }x\inR\mathrm{ Coset-rel H" {g}
            using Group.uminus-closed unfolding rcoset-rel-def by simp
    next
        fix }x\mathrm{ assume }x\inR\mathrm{ Coset-rel H" {g}
        with g}\mathrm{ have x:x<G g-x fH unfolding rcoset-rel-def by auto
        with assms(3) obtain h where h: }h\inH-g+x=-
            by (fastforce simp add: algebra-simps minus-add)
        with assms(1) g x (1) show x L LCoset-rel H" {g}
            using Group.uminus-closed unfolding lcoset-rel-def by simp
    qed
qed
lemma normal-lconjby-closed:
    \llbracket Subgroup H; normal H;g\inG;h\inH\rrbracket\Longrightarrow lconjby g h \in H
    using lcoset-relI[of g g+h H] add-closed[of g h] normal-def[of H]
```

```
        symD[OF Group.rcoset-rel-sym, of H g g+h] rcoset-rel-def[of H]
by
    auto
lemma normal-rconjby-closed:
    \llbracket Subgroup H; normal H;g\inG; h\inH\rrbracket\Longrightarrow rconjby g h \in H
    using normal-lconjby-closed[of H-g h] uminus-closed[of g] by auto
abbreviation normal-closure }A\equiv\langle\bigcupg\inG.lconjby g'A
lemma (in Group) normal-closure:
    assumes }A\subseteq
    shows normal (normal-closure A)
proof (rule normalI, rule genby-Group)
    show }\forallx\inG.\forallh\in\langle\bigcupg\inG.lconjby g'` A\rangle
        \exists}\mp@subsup{h}{}{\prime}\in\langle\bigcupg\inG.lconjby g'A\rangle. x + h= h' + x
    proof
    fix }x\mathrm{ assume }x:x\in
    show }\forallh\in\langle\bigcupg\inG. lconjby g' A\rangle
        \exists}\mp@subsup{h}{}{\prime}\in\langle\bigcupg\inG.lconjby g' A\rangle.x+h= h' + x
    proof (rule ballI, erule genby.induct)
        show }\existsh\in\langle\bigcupg\inG.lconjby g'A\rangle.x+0=h+
        using genby-0-closed by force
    next
        fix s assume s\in(\bigcupg\inG.lconjby g'}A
        from this obtain ga where ga: g\inG a\inA s=lconjby g a by fast
        from ga(3) have x + s=lconjby x (lconjby ga) + x
            by (simp add: algebra-simps)
        hence }x+s=lconjby (x+g)a+x by (simp add:lconjby-add)
        with x ga(1,2) show \exists
            using add-closed by (blast intro: genby-genset-closed)
    next
        fix w w'
        assume w: w\in\langle\bigcupg\inG.lconjby g'A\rangle
                    \existsh\in\langle\bigcupg\inG.lconjby g' }A\rangle.x+w=h+
        and w': w'\in\langle\bigcupg\inG.lconjby g'}A
                    \exists\mp@subsup{h}{}{\prime}\in\\bigcupg\inG.lconjby g' }A\rangle.x+\mp@subsup{w}{}{\prime}=\mp@subsup{h}{}{\prime}+
        from w(2) w'(2) obtain h h'
            where h:h\in\langle\bigcupg\inG.lconjby g'A\rangle x + w = h+x
            and }\mp@subsup{h}{}{\prime}:\mp@subsup{h}{}{\prime}\in\langle\bigcupg\inG.lconjby g' 'A\ranglex+ w' = h'+
        by fast
        have }x+(w-\mp@subsup{w}{}{\prime})=x+w-(-x+(x+\mp@subsup{w}{}{\prime})
            by (simp add: algebra-simps)
        also from h(2) h'(2) have ... = h+x + (-( (h' + x) +x)
            by (simp add: algebra-simps)
        also have ... = h+x+(-x+-\mp@subsup{h}{}{\prime})+x
            by (simp add: minus-add add.assoc)
        finally have }x+(w-\mp@subsup{w}{}{\prime})=h-\mp@subsup{h}{}{\prime}+
            using add.assoc[of h+x -x - h] by simp
        with h(1) h'(1)
```

```
            show \existsh\in\\bigcupg\inG.lconjby g'A\rangle.x + (w-w')=h+x
            using genby-diff-closed
            by fast
        qed
    qed
    show }\forallx\inG.\forallh\in\langle\bigcupg\inG.lconjby g'A\rangle
        \exists}\mp@subsup{h}{}{\prime}\in\langle\bigcupg\inG.lconjby g'A\rangle.h+x=x+h'
    proof
        fix }x\mathrm{ assume }x:x\in
        show }\forallh\in\langle\bigcupg\inG.lconjby g' A\rangle
            \exists}\mp@subsup{h}{}{\prime}\in\langle\bigcupg\inG.lconjby g'A\rangle.h+x=x+ h'
    proof (rule ballI, erule genby.induct)
        show }\existsh\in\langle\bigcupg\inG. lconjby g'A\rangle.0 + x = x +
            using genby-0-closed by force
        next
            fix }s\mathrm{ assume }s\in(\bigcupg\inG.lconjby g'A
            from this obtain g a where ga: g\inG a\inA s=lconjby g a by fast
            from ga(3) have }s+x=x+(((-x+g)+a)+-g)+
                by (simp add: algebra-simps)
            also have ... = x + (-x+g+a+-g+x) by (simp add: add.assoc)
            finally have }s+x=x+lconjby (-x+g)
                by (simp add: algebra-simps lconjby-add)
            with x ga(1,2) show \existsh\in\\bigcup g\inG. lconjby g'A\rangle.s+x=x+h
            using uminus-add-closed by (blast intro: genby-genset-closed)
        next
            fix w w'
            assume w: w\in\langle\bigcupg\inG.lconjby g'A\rangle
                    \existsh\in\\bigcupg\inG.lconjby g'A\rangle.w + x = x + h
            and w': w'\in\langle\bigcupg\inG. lconjby g'A\rangle
                    \exists}\mp@subsup{h}{}{\prime}\in\langle\bigcupg\inG.lconjby g' A\rangle. w' + x = x + h'
            from w(2) w'(2) obtain h h'
                where h:h\in\langle\bigcupg\inG.lconjby g'A\ranglew+x=x+h
                and }\mp@subsup{h}{}{\prime}:\mp@subsup{h}{}{\prime}\in\langle\bigcupg\inG.lconjby g'A\rangle w' + x = x + h'
            by fast
    have }w-\mp@subsup{w}{}{\prime}+x=w+x+(-x+-\mp@subsup{w}{}{\prime})+x\mathrm{ by (simp add: algebra-simps)
            also from h(2) h'(2) have ... = x +h+(-h'+-x) +x
            using minus-add[of w' x] minus-add[of x h'] by simp
            finally have }w-\mp@subsup{w}{}{\prime}+x=x+(h-\mp@subsup{h}{}{\prime})\mathbf{by (simp add: algebra-simps)
            with h(1) h'(1) show \existsh\in\langle\bigcupg\inG. lconjby g'A\rangle.w - w' + x = x +h
            using genby-diff-closed by fast
        qed
    qed
qed
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\inG g'\inG
    shows LCoset-rel H" {g}+LCoset-rel H" {g'} = LCoset-rel H" {g+g'}
proof (rule seteqI)
    fix }x\mathrm{ assume }x\inL\mathrm{ Loset-rel H" {g}+LCoset-rel H" {g'}
    from this obtain yz
        where y\inLCoset-rel H" {g}z\inLCoset-rel H" {g'}x=y+z
        unfolding set-plus-def
        by fast
    with assms show }x\inL\mathrm{ Loset-rel H" {g+g'}
        using lcoset-rel-def[of H] normal-lconjby-closed[of H g
                Group.add-closed
                normal-rconjby-closed[of H g' -g+y + (z-g')]
                add.assoc[of - - ' -g]
                add-closed lcoset-relI[of g+\mp@subsup{g}{}{\prime}}y+z
    by (fastforce simp add: add.assoc minus-add)
next
    fix x assume x LCoset-rel H" {g+g'}
    moreover define h where h\equiv-(g+g')+x
    moreover hence x=g+( (g'+h)
        using add.assoc[of - - ' - g x] by (simp add: add.assoc minus-add)
    ultimately show }x\inL\mathrm{ Loset-rel H"{g} + LCoset-rel H" {g'}
        using assms(1,3,4) lcoset-rel-def[of H] add-closed
                refl-onD[OF subgroup-refl-on-LCoset-rel, of H]
        by force
qed
```

abbreviation quotient-set $H \equiv G / /$ LCoset-rel $H$
lemma BinOpSetGroup-normal-quotient:
assumes Subgroup H normal H
shows BinOpSetGroup (quotient-set H) (+)H
proof
from $\operatorname{assms}(1)$ have $H 0: H=$ LCoset-rel $H "\{0\}$
using trivial-LCoset by auto
from $\operatorname{assms}(1)$ show $H \in$ quotient-set $H$
using HO zero-closed LCoset-rel-quotientI[of 0 H$]$ by simp
fix $x$ assume $x \in$ quotient-set $H$
from this obtain $g x$ where $g x: g x \in G x=$ LCoset-rel $H$ " $\{g x\}$
by (fast elim: LCoset-rel-quotientE)

```
with \(\operatorname{assms}(1,2)\) show \(x+H=x H+x=x\)
    using normal-quotient-add-well-defined \([\) of \(H\) gx 0 \(]\)
            normal-quotient-add-well-defined[of H 0 gx ]
            H0 zero-closed
    by auto
from \(g x(1)\) have LCoset-rel \(H\) " \(\{-g x\} \in\) quotient-set \(H\)
    using uminus-closed by (fast intro: LCoset-rel-quotientI)
moreover from \(\operatorname{assms}(1,2) g x\)
    have \(x+\) LCoset-rel \(H\) " \(\{-g x\}=H\) LCoset-rel \(H\) " \(\{-g x\}+x=H\)
    using H0 uminus-closed normal-quotient-add-well-defined
    by auto
ultimately show \(\exists x^{\prime} \in q u o t i e n t-s e t ~ H . x+x^{\prime}=H \wedge x^{\prime}+x=H\) by fast
fix \(y\) assume \(y \in\) quotient-set \(H\)
from this obtain gy where gy: gy G \(y=\) LCoset-rel \(H\) " \(\{g y\}\)
    by (fast elim: LCoset-rel-quotientE)
with assms gx show \(x+y \in\) quotient-set \(H\)
    using add-closed normal-quotient-add-well-defined
    by (auto intro: LCoset-rel-quotientI)
qed (rule add.assoc)
abbreviation abs-lcoset-perm \(H \equiv\)
    BinOpSetGroup.Abs-G-perm (quotient-set H) (+)
abbreviation abs-lcoset-perm-lift H g abs-lcoset-perm H (LCoset-rel H" \(\{g\}\) )
abbreviation abs-lcoset-perm-lift-arg-permutation \(g H \equiv\) abs-lcoset-perm-lift Hg
notation abs-lcoset-perm-lift-arg-permutation (Г-|-ך [51,51] 50)
end
abbreviation Group-abs-lcoset-perm-lift-arg-permutation \(G^{\prime} g H \equiv\)
    Group.abs-lcoset-perm-lift-arg-permutation \(G^{\prime} g H\)
notation Group-abs-lcoset-perm-lift-arg-permutation (โ-|-|-† [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 \(H h \in H\)
    shows restrict1 \(((+)(\) LCoset-rel \(H "\{h\}))(\) quotient-set \(H)=i d\)
```

```
proof (rule ext, simp, rule impI)
```

    fix \(x\) assume \(x: x \in\) quotient-set \(H\)
    then obtain \(k\) where \(k: k \in G x=\) LCoset-rel \(H\) " \(\{k\}\)
        by (blast elim: LCoset-rel-quotientE)
    with \(x\) have LCoset-rel \(H\) " \(\{h\}+x=\) LCoset-rel \(H\) " \(\{h+k\}\)
        using assms normal-quotient-add-well-defined by auto
    with assms \(k\) show LCoset-rel \(H\) " \(\{h\}+x=x\)
        using add-closed[of \(h k\) ] lcoset-relI[of \(k h+k H\) ]
                normal-rconjby-closed[of H \(k\) h]
                eq-equiv-class-iff[OF lcoset-subgroup-rel-equiv, of \(H\) ]
    by (auto simp add: add.assoc)
    qed
definition quotient-group $::$ ' $g$ set $\Rightarrow$ ' $g$ set permutation set where quotient-group $H \equiv$ BinOpSetGroup.pG (quotient-set $H)(+)$
abbreviation natural-quotient-hom $H \equiv \operatorname{restrict0}(\lambda g .\lceil g \mid H\rceil) G$
theorem quotient-group:
Subgroup $H \Longrightarrow$ normal $H \Longrightarrow$ Group (quotient-group $H$ )
unfolding quotient-group-def
using BinOpSetGroup.Group[OF BinOpSetGroup-normal-quotient]
by auto
lemma natural-quotient-hom:
Subgroup $H \Longrightarrow$ normal $H \Longrightarrow$ GroupHom $G$ (natural-quotient-hom $H$ )
using add-closed bij-lcoset-perm lcoset-perm-def supp-restrict0
normal-quotient-add-well-defined[THEN sym]
LCoset-rel-quotientI[of - H]
by unfold-locales
(force simp add: lcoset-perm-comp plus-permutation-abs-eq)
lemma natural-quotient-hom-image:
natural-quotient-hom $H$ ' $G=$ quotient-group $H$
unfolding quotient-group-def
by (force elim: LCoset-rel-quotientE intro: LCoset-rel-quotientI)
lemma quotient-group-UN: quotient-group $H=(\lambda g .\lceil g \mid H\rceil)$ ' $G$
using natural-quotient-hom-image by auto
lemma quotient-identity-rule: $\llbracket$ Subgroup $H$; normal $H ; h \in H \rrbracket \Longrightarrow\lceil h \mid H\rceil=0$ using lcoset-perm-def
by (simp add: trivial-lcoset-perm zero-permutation.abs-eq)
lemma quotient-group-lift-to-quotient-set:
$\llbracket$ Subgroup $H$; normal $H ; g \in G \rrbracket \Longrightarrow(\lceil g \mid H\rceil) \rightarrow H=$ LCoset-rel $H$ " $\{g\}$
using LCoset-rel-quotientI
BinOpSetGroup. G-perm-apply-identity[
OF BinOpSetGroup-normal-quotient

```
    ]
    by simp
```

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 Ker \LongrightarrowT respects (LCoset-rel H)
    using uminus-add-in-Ker-eq-eq-im
    unfolding lcoset-rel-def
    by (blast intro: congruentI)
abbreviation quotient-hom H}
    BinOpSetGroup.lift-hom (quotient-set H) (+)(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\subseteqKer;g\inG\rrbracket\Longrightarrow quotient-hom H (\lceilg|H\rceil)=Tg
    using quotient-group-def quotient-group-UN quotient-group-lift-to-quotient-set
            BinOpSetGroup.inv-correspondence-conv-apply[
                        OF BinOpSetGroup-normal-quotient
        ]
        normal-subgroup-quotientfun-classrep-equality
    by auto
lemma quotient-hom:
    assumes Subgroup H normal H H\subseteq Ker
    shows GroupHom (quotient-group H) (quotient-hom H)
    unfolding quotient-group-def
proof (
    rule BinOpSetGroup.lift-hom, rule BinOpSetGroup-normal-quotient, rule assms(1),
    rule assms(2)
)
    from assms
        show }\forallx\inquotient-set H.\forally\inquotient-set H
            quotientfun T (x+y) = quotientfun T x + quotientfun T y
    using normal-quotient-add-well-defined normal-subgroup-quotientfun-classrep-equality
```

```
        add-closed hom
    by (fastforce elim: LCoset-rel-quotientE)
qed
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 $\equiv \lambda s$. $(s$, True $)$
abbreviation pairfalse $\equiv \lambda s .(s, F a l s e)$
abbreviation flip-signed $::$ 'a signed $\Rightarrow$ 'a signed
where flip-signed $\equiv$ apsnd $(\lambda b . \neg b)$
abbreviation nflipped-signed $::$ 'a signed $\Rightarrow$ ' $a$ signed $\Rightarrow$ bool
where nflipped-signed $x y \equiv y \neq$ flip-signed $x$
lemma flip-signed-order2: flip-signed $($ flip-signed $x)=x$
using apsnd-conv[of $\lambda b$. $\neg b$ fst $x$ snd $x]$ by simp
abbreviation charpair :: ' $a::$ uminus set $\Rightarrow{ }^{\prime} a \Rightarrow$ 'a signed where charpair $S s \equiv$ if $s \in S$ then ( $s$, True) else $(-s$, False $)$
lemma map-charpair-uniform:
ss $\in$ lists $S \Longrightarrow$ map (charpair $S$ ) ss = map pairtrue ss
by (induct ss) auto
lemma fst-set-map-charpair-un-uminus:
fixes ss :: 'a::group-add list
shows ss $\in$ lists $(S \cup$ uminus' $S) \Longrightarrow f s t$ 'set (map (charpair $S)$ ss) $\subseteq S$
by (induct ss) auto
abbreviation apply-sign $::\left({ }^{\prime} a \Rightarrow\right.$ ' $b::$ uminus $) \Rightarrow{ }^{\prime} a$ signed $\Rightarrow{ }^{\prime} b$ where apply-sign $f x \equiv$ (if snd $x$ then $f(f s t x)$ else $-f(f s t x))$

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 $\Rightarrow$ bool where proper-signed-list $\equiv$ binrelchain nflipped-signed
lemma proper-map-flip-signed:
proper-signed-list $x s \Longrightarrow$ proper-signed-list (map fip-signed $x s$ )
by (induct xs rule: list-induct-CCons) auto
lemma proper-rev-map-flip-signed:
proper-signed-list $x s \Longrightarrow$ proper-signed-list (rev (map flip-signed $x s$ ))
using proper-map-flip-signed binrelchain-sym-rev[of nflipped-signed] by fastforce
lemma uniform-snd-imp-proper-signed-list:
snd'set $x s \subseteq\{b\} \Longrightarrow$ proper-signed-list xs
proof (induct xs rule: list-induct-CCons)
case CCons thus ?case by force
qed auto
lemma proper-signed-list-map-uniform-snd: proper-signed-list (map $(\lambda s .(s, b))$ as)
using uniform-snd-imp-proper-signed-list $[$ of - b] by force

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 $\Rightarrow$ 'a signed list $\Rightarrow$ 'a signed list where prappend-signed-list xs []$=x s$
| prappend-signed-list [] ys = ys
| prappend-signed-list $(x s @[x])(y \# y s)=($
if $y=$ flip-signed $x$ then prappend-signed-list $x s$ ys else $x s @ x \# y \# y s$ )
by (auto) (rule two-prod-lists-cases-snoc-Cons)
termination by (relation measure $(\lambda(x s, y s)$. length $x s+$ length $y s)$ ) auto
lemma proper-prappend-signed-list:
proper-signed-list xs $\Longrightarrow$ proper-signed-list ys $\Longrightarrow$ proper-signed-list (prappend-signed-list xs ys)
proof (induct xs ys rule: list-induct2-snoc-Cons)
case (snoc-Cons xs x y ys)
show ?case
proof (cases $y=$ flip-signed $x$ )
case True with snoc-Cons show ?thesis
using binrelchain-append-reduce1 [of nflipped-signed] binrelchain-Cons-reduce[of nflipped-signed $y$ ]
by auto
next
case False with snoc-Cons(2,3) show ?thesis
using binrelchain-join[of nflipped-signed] by simp
qed
qed auto
lemma fully-prappend-signed-list:
prappend-signed-list (rev (map flip-signed $x s$ ) ) xs = []
by (induct $x s$ ) auto

```
lemma prappend-signed-list-single-Cons:
    prappend-signed-list [x] (y#ys)=(if y= flip-signed x then ys else }x#y#ys
    using prappend-signed-list.simps(3)[of [] x] by simp
lemma prappend-signed-list-map-uniform-snd:
    prappend-signed-list (map ( }\lambdas.(s,b))xs)(\operatorname{map}(\lambdas.(s,b))ys)
        map (\lambdas. (s,b)) xs @ map (\lambdas. (s,b)) ys
    by (cases xs ys rule: two-lists-cases-snoc-Cons) auto
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 (cases xs ys rule: two-lists-cases-snoc-Cons')
    case Nil1 with assms(2) show ?thesis
        by (simp add: prappend-signed-list-single-Cons)
next
    case Nil2 with assms(1) show ?thesis
        using binrelchain-append-reduce2 by force
next
    case (snoc-Cons as a b bs)
    with assms show ?thesis
        using prappend-signed-list.simps(3)[of as@[a]]
            binrelchain-append-reduce2[of nflipped-signed as [a,y]]
        by simp
qed simp
lemma prappend-signed-list-assoc:
    \llbracket proper-signed-list xs; proper-signed-list ys; proper-signed-list zs \rrbracket\Longrightarrow
        prappend-signed-list (prappend-signed-list xs ys) zs=
        prappend-signed-list xs (prappend-signed-list ys zs)
proof (induct xs ys zs rule: list-induct3-snoc-Conssnoc-Cons-pairwise)
    case (snoc-single-Cons xs x y z zs)
    thus ?case
        using prappend-signed-list.simps(3)[of [] y]
            prappend-signed-list.simps(3)[of xs@[x]]
        by (cases y = flip-signed x z = flip-signed y rule: two-cases)
                (auto simp add:
                        flip-signed-order2 prappend-signed-list-assoc-conv-snoc2Cons
                )
next
    case (snoc-Conssnoc-Cons xs x y ys wzzs)
    thus ?case
        using binrelchain-Cons-reduce[of nflipped-signed y ys@[w]]
                binrelchain-Cons-reduce[of nflipped-signed z zs]
                binrelchain-append-reduce1[of nflipped-signed xs]
                binrelchain-append-reduce1[of nflipped-signed y#ys]
                binrelchain-Conssnoc-reduce[of nflipped-signed y ys]
                prappend-signed-list.simps(3)[of y#ys]
```

```
        prappend-signed-list.simps(3)[of xs@x#y#ys]
    by (cases y = flip-signed x z = flip-signed w rule: two-cases) auto
qed auto
lemma fst-set-prappend-signed-list:
    fst ' set (prappend-signed-list xs ys) \subseteqfst '(set xs \cup set ys)
    by (induct xs ys rule: list-induct2-snoc-Cons) auto
lemma collapse-flipped-signed:
    prappend-signed-list [(s,b)] [(s,\negb)] = []
    using prappend-signed-list.simps(3)[of [] (s,b)] by simp
```


### 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
    using binrelchain.simps(1) by fast
```

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

```
abbreviation Abs-freeletter :: ' \(a \Rightarrow\) ' \(a\) freeword
    where Abs-freeletter \(s \equiv\) Abs-freeword [pairtrue \(s\) ]
abbreviation Abs-freelist :: 'a list \(\Rightarrow\) 'a freeword
    where Abs-freelist as \(\equiv\) Abs-freeword (map pairtrue as)
abbreviation Abs-freelistfst :: 'a signed list \(\Rightarrow\) 'a freeword
    where Abs-freelistfst \(x s \equiv\) 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 by simp
instance ..
end
instantiation freeword :: (type) plus
begin
lift-definition plus-freeword :: 'a freeword \(\Rightarrow\) 'a freeword \(\Rightarrow\) 'a freeword
```

```
    is prappend-signed-list
    using proper-prappend-signed-list
    by fast
instance ..
end
instantiation freeword :: (type) uminus
begin
lift-definition uminus-freeword :: 'a freeword => 'a freeword
    is \lambdaxs. rev (map flip-signed xs)
    by (rule proper-rev-map-flip-signed)
instance ..
end
instantiation freeword :: (type) minus
begin
lift-definition minus-freeword :: 'a freeword }=>\mathrm{ 'a freeword }=>\mathrm{ 'a freeword
    is \lambdaxs ys. prappend-signed-list xs (rev (map flip-signed ys))
    using proper-rev-map-flip-signed proper-prappend-signed-list by fast
instance ..
end
instance freeword :: (type) semigroup-add
proof
    fix a b c:: 'a freeword show }a+b+c=a+(b+c
        using prappend-signed-list-assoc[of freeword a freeword b freeword c]
            by transfer simp
qed
instance freeword :: (type) monoid-add
proof
    fix ab c :: 'a freeword
    show 0 + a =a by transfer simp
    show }a+0=a\mathrm{ by transfer simp
qed
instance freeword :: (type) group-add
proof
    fix a b :: 'a freeword
    show - a+a=0
        using fully-prappend-signed-list[of freeword a] by transfer simp
    show }a+-b=a-b by transfer sim
qed
```

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>\mp@subsup{}{}{\prime}a=>'a freeword (infixl [+] 65)
    where s}[+]t\equivAbs-freeletter s+Abs-freeletter t
lemma Abs-freeword-Cons:
    assumes proper-signed-list (x#xs)
    shows Abs-freeword (x#xs) = Abs-freeword [x] + Abs-freeword xs
proof (cases xs)
    case Nil thus ?thesis
        using add-0-right[of Abs-freeword [x]] by (simp add:zero-freeword.abs-eq)
next
    case (Cons y ys)
    with assms
        have freeword (Abs-freeword (x#xs)) =
                freeword (Abs-freeword [x] + Abs-freeword xs)
    by (simp add:
            plus-freeword.rep-eq Abs-freeword-inverse
            prappend-signed-list-single-Cons
            )
    thus ?thesis using freeword-inject by fast
qed
lemma Abs-freelist-Cons:Abs-freelist (x#xs) = Abs-freeletter x + Abs-freelist xs
    using proper-signed-list-map-uniform-snd[of True x#xs] Abs-freeword-Cons
    by simp
lemma plus-freeword-abs-eq:
    proper-signed-list xs \Longrightarrow proper-signed-list ys \Longrightarrow
    Abs-freeword xs + Abs-freeword ys = Abs-freeword (prappend-signed-list xs ys)
    using plus-freeword.abs-eq unfolding eq-onp-def by simp
lemma Abs-freeletter-add:s[+] t=Abs-freelist [s,t]
    using Abs-freelist-Cons[of s [t]] by simp
lemma uminus-freeword-Abs-eq:
    proper-signed-list xs \Longrightarrow
    - Abs-freeword xs = Abs-freeword (rev (map flip-signed xs))
    using uminus-freeword.abs-eq unfolding eq-onp-def by simp
lemma uminus-Abs-freeword-singleton:
    - Abs-freeword [(s,b)]=Abs-freeword [(s,\neg b)]
    using uminus-freeword-Abs-eq[of [(s,b)]] by simp
lemma Abs-freeword-append-uniform-snd:
    Abs-freeword (map (\lambdas. (s,b)) (xs@ys)) =
    Abs-freeword (map (\lambdas. (s,b)) xs) + Abs-freeword (map ( }\lambdas.(s,b)) ys
using proper-signed-list-map-uniform-snd[of b xs]
        proper-signed-list-map-uniform-snd[of b ys]
        plus-freeword-abs-eq prappend-signed-list-map-uniform-snd[of b xs ys]
by force
```

```
lemmas Abs-freelist-append = Abs-freeword-append-uniform-snd[of True]
lemma Abs-freelist-append-append:
    Abs-freelist (xs@ys@zs)=Abs-freelist xs + Abs-freelist ys + Abs-freelist zs
    using Abs-freelist-append[of xs@ys] Abs-freelist-append by simp
lemma Abs-freelist-inverse: freeword (Abs-freelist as) = map pairtrue as
    using proper-signed-list-map-uniform-snd Abs-freeword-inverse by fast
lemma Abs-freeword-singleton-conv-apply-sign-freeletter:
    Abs-freeword [x] = apply-sign Abs-freeletter x
    by (cases x) (auto simp add:uminus-Abs-freeword-singleton)
lemma Abs-freeword-conv-freeletter-sum-list:
    proper-signed-list xs \Longrightarrow
        Abs-freeword xs =(\sumx\leftarrowxs. apply-sign Abs-freeletter x)
proof (induct xs)
    case (Cons x xs) thus ?case
        using Abs-freeword-Cons[of x] binrelchain-Cons-reduce[of - x]
        by (simp add: Abs-freeword-singleton-conv-apply-sign-freeletter)
qed (simp add: zero-freeword.abs-eq)
lemma freeword-conv-freeletter-sum-list:
    x=(\sums\leftarrow\mathrm{ freeword x. apply-sign Abs-freeletter s)}
    using Abs-freeword-conv-freeletter-sum-list[of freeword x] freeword
    by (auto simp add: freeword-inverse)
lemma Abs-freeletter-prod-conv-Abs-freeword:
    snd x Abs-freeletter (fst x) =Abs-freeword [x]
    using prod-eqI[of x pairtrue (fst x)] by simp
```


### 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 = 'b::group-add
    defines h:h::'a signed => ' }b\equiv\lambda(s,b)\mathrm{ . if b then f s else - (f s)
    defines g: g::'a signed list }=>\mp@subsup{}{}{\prime}'b\equiv\lambdaxs\mathrm{ . sum-list (map h xs)
    shows g(prappend-signed-list xs ys) =gxs + g ys
proof (induct xs ys rule: list-induct2-snoc-Cons)
```

```
    case (snoc-Cons xs x y ys)
    show ?case
    proof (cases y = flip-signed }x\mathrm{ )
    case True
    with h have hy=-hx
        using split-beta'[of \lambdas b. if b then f s else - (f s)] by simp
    with g have g(xs@ @x])+g(y#ys)=gxs+gys
        by (simp add: algebra-simps)
    with True snoc-Cons show ?thesis by simp
    next
    case False with g show ?thesis
        using sum-list.append[of map h(xs@[x]) map h(y#ys)] by simp
    qed
qed (auto simp add: h g)
lemma extend-map-to-freeword-hom1:
    fixes f:: ' }a>>'b::group-add
    defines h::'a signed }\mp@subsup{=>}{}{\prime}b\equiv\lambda(s,b)\mathrm{ . if b then f s else - (f s)
    defines g::'a freeword }\mp@subsup{=>}{}{\prime}b\equiv\lambdax\mathrm{ . sum-list (map h (freeword x))
    shows g(Abs-freeletter s)=fs
    using assms
    by (simp add: Abs-freeword-inverse)
lemma extend-map-to-freeword-hom2:
    fixes f:: 'a = 'b::group-add
    defines h::'a signed }=>\mp@subsup{}{}{\prime}b\equiv\lambda(s,b)\mathrm{ . if b then f s else - (f s)
    defines g::'a freeword }=>\mp@subsup{}{}{\prime}b\equiv\lambdax\mathrm{ . sum-list (map h (freeword x))
    shows UGroupHom g
    using assms
    by
        auto intro:UGroupHomI
        simp add: plus-freeword.rep-eq extend-map-to-freeword-hom'
        )
lemma uniqueness-of-extended-map-to-freeword-hom':
    fixes f::' 'a > 'b::group-add
    defines h:h::'a signed => 'b \equiv\lambda(s,b). if b then f s else - (f s)
    defines g:g::'a signed list }=>\mp@subsup{}{}{\prime}b\equiv\lambdaxs\mathrm{ . sum-list (map h xs)
    assumes singles: \s. k[(s,True)]=fs
    and adds : \bigwedgexs ys. proper-signed-list xs \Longrightarrow proper-signed-list ys
        \Longrightarrow k ( \text { prappend-signed-list xs ys) =kxs + k ys}
    shows proper-signed-list xs \Longrightarrowkxs=gxs
proof-
    have knil: k [] = 0 using adds[of [] []] add.assoc[of k [] k [] - k []] by simp
    have ksingle: }\x.k[x]=g[x
    proof-
        fix }x::\mathrm{ 'a signed
        obtain s b where x: x = (s,b) by fastforce
        show }k[x]=g[x
```

```
    proof (cases b)
        case False
        from adds x singles
            have k(prappend-signed-list [x][(s,True)]) =k[x]+fs
            by simp
    moreover have prappend-signed-list [(s,False)] [(s,True)] = []
            using collapse-flipped-signed[of s False] by simp
        ultimately have - fs=k[x]+fs+-fs using x False knil by simp
        with x False gh show }k[x]=g[x] by (simp add: algebra-simps
    qed (simp add: x g h singles)
    qed
    show proper-signed-list xs \Longrightarrowkxs=gxs
    proof (induct xs rule: list-induct-CCons)
        case (CCons x y xs)
        with gh show??case
        using adds[of [x] y#xs]
        by (simp add:
                prappend-signed-list-single-Cons
                    ksingle extend-map-to-freeword-hom'
            )
    qed (auto simp add: g h knil ksingle)
qed
lemma uniqueness-of-extended-map-to-freeword-hom:
    fixes f:: 'a = 'b::group-add
    defines h::'a signed }=>\mp@subsup{}{}{\prime}b\equiv\lambda(s,b)\mathrm{ . if b then fs else - (f s)
    defines g::'a freeword => 'b\equiv\lambdax. sum-list (map h (freeword x))
    assumes k: k\circAbs-freeletter =f UGroupHom k
    shows k=g
proof
    fix x::'a freeword
    define }\mp@subsup{k}{}{\prime}\mathrm{ where }\mp@subsup{k}{}{\prime}:\mp@subsup{k}{}{\prime}\equivk\circ\mathrm{ Abs-freeword
    have }\mp@subsup{k}{}{\prime}(\mathrm{ freeword }x\mathrm{ ) = gx unfolding h-def g-def
    proof (rule uniqueness-of-extended-map-to-freeword-hom')
        from }\mp@subsup{k}{}{\prime}k(1)\mathrm{ show \s. 兆[pairtrue s]=fs by auto
        show \xs ys. proper-signed-list xs \Longrightarrow proper-signed-list ys
                            \Longrightarrow k ^ { \prime } ( \text { prappend-signed-list xs ys) = k'xs + k' ys}
        proof-
            fix xs ys :: 'a signed list
            assume xsys: proper-signed-list xs proper-signed-list ys
            with }\mp@subsup{k}{}{\prime
                    show }\mp@subsup{k}{}{\prime}(\mathrm{ prappend-signed-list xs ys) = k'xs + k' ys
                    using UGroupHom.hom[OF k(2), of Abs-freeword xs Abs-freeword ys]
                    by (simp add: plus-freeword-abs-eq)
        qed
        show proper-signed-list (freeword x) using freeword by fast
    qed
    with }\mp@subsup{k}{}{\prime}\mathrm{ show kx = gx using freeword-inverse[of x] by simp
qed
```

```
theorem universal-property:
    fixes f :: ' }a>>''\mp@code{':group-add
    shows }\exists\mathrm{ ! g::'a freeword }\mp@subsup{=>}{}{\prime}b.g\circ\mathrm{ Abs-freeletter }=f\wedge\mathrm{ UGroupHom g
proof
    define h where h: h\equiv\lambda(s,b). if b then f s else - (f s)
    define g}\mathrm{ where g: g}\equiv\lambdax\mathrm{ . sum-list (map h (freeword x))
    from gh show g\circAbs-freeletter =f^UGroupHom g
        using extend-map-to-freeword-hom1[of f] extend-map-to-freeword-hom2
        by auto
    from gh show \k.k\circAbs-freeletter =f^UGroupHom k\Longrightarrowk=g
        using uniqueness-of-extended-map-to-freeword-hom by auto
qed
```

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 :: (' }=>>'b::group-add) => ('a freeword ='b::group-add)
    where freeword-funlift f\equiv(THE g.g\circAbs-freeletter = f ^UGroupHom g)
lemma additive-freeword-funlift: UGroupHom (freeword-funlift f)
    using theI'[OF universal-property, of f] unfolding freeword-funlift-def by simp
lemma freeword-funlift-Abs-freeletter: freeword-funlift f(Abs-freeletter s) =f s
    using theI'[OF universal-property, of f]
                comp-apply[of freeword-funlift f Abs-freeletter]
    unfolding freeword-funlift-def
    by fastforce
lemmas freeword-funlift-add = UGroupHom.hom [OF additive-freeword-funlift]
lemmas freeword-funlift-0 = UGroupHom.im-zero [OF additive-freeword-funlift]
lemmas freeword-funlift-uminus = UGroupHom.im-uminus [OF additive-freeword-funlift]
lemmas freeword-funlift-diff = UGroupHom.im-diff [OF additive-freeword-funlift]
lemmas freeword-funlift-lconjby = UGroupHom.im-lconjby [OF additive-freeword-funlift]
lemma freeword-funlift-uminus-Abs-freeletter:
    freeword-funlift f(Abs-freeword [(s,False)]) =-fs
    using freeword-funlift-uminus[of f Abs-freeword [(s,False)]]
        uminus-freeword-Abs-eq[of [(s,False)]]
        freeword-funlift-Abs-freeletter[of f]
    by simp
lemma freeword-funlift-Abs-freeword-singleton:
    freeword-funlift f(Abs-freeword [x]) = apply-sign f x
proof-
    obtain s b where x: x= (s,b) by fastforce
    thus ?thesis
        using freeword-funlift-Abs-freeletter freeword-funlift-uminus-Abs-freeletter
```

```
    by (cases b) auto
qed
lemma freeword-funlift-Abs-freeword-Cons:
    assumes proper-signed-list (x#xs)
    shows freeword-funlift f(Abs-freeword (x#xs)) =
        apply-sign fx + freeword-funlift f(Abs-freeword xs)
proof -
    from assms
        have freeword-funlift f (Abs-freeword (x#xs))=
            freeword-funlift f (Abs-freeword [x]) +
            freeword-funlift f (Abs-freeword xs)
        using Abs-freeword-Cons[of x xs] freeword-funlift-add by simp
    thus ?thesis
    using freeword-funlift-Abs-freeword-singleton[of f x] by simp
qed
lemma freeword-funlift-Abs-freeword:
    proper-signed-list xs \Longrightarrow freeword-funlift f(Abs-freeword xs)=
        (\sumx\leftarrowxs.apply-sign f x )
proof (induct xs)
    case (Cons x xs) thus ?case
        using freeword-funlift-Abs-freeword-Cons[of - - f]
                binrelchain-Cons-reduce[of - x xs]
    by simp
qed (simp add:zero-freeword.abs-eq[THEN sym] freeword-funlift-0)
lemma freeword-funlift-Abs-freelist:
    freeword-funlift f(Abs-freelist xs)}=(\sumx\leftarrowxs.fx
proof (induct xs)
    case (Cons x xs) thus ?case
            using Abs-freelist-Cons[of x xs]
            by (simp add: freeword-funlift-add freeword-funlift-Abs-freeletter)
qed (simp add:zero-freeword.abs-eq[THEN sym] freeword-funlift-0)
lemma freeword-funlift-im':
    proper-signed-list xs \Longrightarrowfst'set xs \subseteqS\Longrightarrow
            freeword-funlift f(Abs-freeword xs) \in\langlef`S\rangle
proof (induct xs)
    case Nil
    have Abs-freeword ([]::'a signed list) = (0::'a freeword)
            using zero-freeword.abs-eq[THEN sym] by simp
    thus freeword-funlift f (Abs-freeword ([]::'a signed list)) \in\langlef'S\rangle
            using freeword-funlift-0[of f] genby-0-closed by simp
next
    case (Cons x xs)
    define }y\mathrm{ where y:}y\equiv\operatorname{apply-sign f x
    define z where z:z\equiv freeword-funlift f (Abs-freeword xs)
    from Cons(3) have fst' set xs \subseteqS by simp
```

```
    with z Cons(1,2) have z\in\langlef'S\rangle using binrelchain-Cons-reduce by fast
    with y Cons(3) have }y+z\in\langle\mp@subsup{f}{}{\prime}S
    using genby-genset-closed[of - f`S]
        genby-uminus-closed genby-add-closed[of y]
    by fastforce
with Cons(2) y z show ?case
    using freeword-funlift-Abs-freeword-Cons
        subst[
            OF sym,
            of freeword-funlift f (Abs-freeword (x#xs)) y+z
                \lambdab. b\in\langlef`}\\
        ]
    by fast
qed
```


### 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 = 'a freeword set
    where FreeGroup S \equiv{x. fst'set (freeword x)\subseteqS}
lemma FreeGroupI-transfer:
    proper-signed-list xs \Longrightarrowfst' set xs \subseteqS\LongrightarrowAbs-freeword xs }\in\mathrm{ FreeGroup S
    using Abs-freeword-inverse unfolding FreeGroup-def by fastforce
lemma FreeGroupD: x F FreeGroup S\Longrightarrowfst'set (freeword x)\subseteqS
    using FreeGroup-def by fast
lemma FreeGroupD-transfer:
    proper-signed-list xs \LongrightarrowAbs-freeword xs }\in\mathrm{ FreeGroup S # fst'set xs }\subseteq
    using Abs-freeword-inverse unfolding FreeGroup-def by fastforce
lemma FreeGroupD-transfer':
```



```
    using proper-signed-list-map-uniform-snd FreeGroupD-transfer by fastforce
lemma FreeGroup-0-closed: 0 E FreeGroup S
proof-
    have (0::'a freeword) = Abs-freeword [] using zero-freeword.abs-eq by fast
    moreover have Abs-freeword [] \in FreeGroup S
        using FreeGroupI-transfer[of []] by simp
    ultimately show ?thesis by simp
qed
```

```
lemma FreeGroup-diff-closed:
    assumes }x\in\mathrm{ FreeGroup S y f FreeGroup S
    shows }x-y\in\mathrm{ FreeGroup S
proof-
    define xs where xs: xs \equiv freeword x
    define ys where ys: ys \equiv freeword y
    have freeword (x-y)=
        prappend-signed-list (freeword x) (rev (map flip-signed (freeword y)))
        by transfer simp
    hence fst' set (freeword (x-y))\subseteqfst ' (set (freeword x)\cup set (freeword y))
        using fst-set-prappend-signed-list by force
    with assms show ?thesis unfolding FreeGroup-def by fast
qed
lemma FreeGroup-Group:Group (FreeGroup S)
    using FreeGroup-0-closed FreeGroup-diff-closed by unfold-locales fast
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 }\in\mathrm{ lists S \Abs-freelist ss }\in\mathrm{ FreeGroup S
    using proper-signed-list-map-uniform-snd by (fastforce intro: FreeGroupI-transfer)
lemma Abs-freeletter-in-FreeGroup-iff:(Abs-freeletter s \in FreeGroup S)=(s\inS)
    using Abs-freeword-inverse[of [pairtrue s]] unfolding FreeGroup-def by simp
```

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 $f S \equiv$
restrict0 (freeword-funlift f) (FreeGroup S)
lemma freeword-funlift-im: $x \in$ FreeGroup $S \Longrightarrow$ freeword-funlift $f x \in\langle f$ ' $S\rangle$
using freeword $[$ of $x]$ freeword-funlift-im' $[$ of freeword $x]$
freeword-inverse $[$ of $x]$
unfolding FreeGroup-def
by auto
lemma freeword-funlift-surj':
ys $\in$ lists $\left(f^{\prime} S \cup\right.$ uminus' $f$ ' $\left.S\right) \Longrightarrow$ sum-list ys $\in$ freeword-funlift $f$ ' FreeGroup $S$ proof (induct ys)
case Nil thus ?case using FreeGroup-0-closed freeword-funlift-0 by fastforce next
case (Cons y ys)
from this obtain $x$

```
    where x: x F FreeGroup S sum-list ys = freeword-funlift f x
    by auto
    show sum-list (y#ys) \in freeword-funlift f 'FreeGroup S
    proof (cases y f f'S)
    case True
    from this obtain s where s: s\inS y=fs by fast
    from }s(1)x(1) have Abs-freeletter s+x\in FreeGroup 
        using FreeGroupI-transfer[of-S] FreeGroup-add-closed[of - S] by force
    moreover from s(2) x(2)
        have freeword-funlift f(Abs-freeletter s + x)=sum-list (y#ys)
        using freeword-funlift-add[of f] freeword-funlift-Abs-freeletter
        by simp
    ultimately show ?thesis by force
    next
    case False
    with Cons(2) obtain s where s: s\inS y=-fs by auto
    from s(1)x(1) have Abs-freeword [(s,False)] + x FreeGroup S
        using FreeGroupI-transfer[of - S] FreeGroup-add-closed[of - S] by force
    moreover from s(2) x(2)
        have freeword-funlift f (Abs-freeword [(s,False)] + x)= sum-list (y#ys)
        using freeword-funlift-add[of f] freeword-funlift-uminus-Abs-freeletter
        by simp
    ultimately show ?thesis by force
    qed
qed
lemma freeword-funlift-surj:
    fixes f :: ' }a>>'\mp@code{'::group-add
    shows freeword-funlift f' FreeGroup S = < f'S\rangle
proof (rule seteqI)
    show \a. a f freeword-funlift f 'FreeGroup S\Longrightarrowa\in\langlef`S\rangle
        using freeword-funlift-im by auto
next
    fix w assume w\in\langlef`}S
    from this obtain ys where ys: ys \inlists (f`S\cupuminus'f`}S)w=sum-list y
        using genby-eq-sum-lists[of f'S] by auto
    thus w\in freeword-funlift f 'FreeGroup S using freeword-funlift-surj' by simp
qed
lemma hom-restrict0-freeword-funlift:
    GroupHom (FreeGroup S) (res-freeword-funlift f S)
    using UGroupHom.restrict0 additive-freeword-funlift FreeGroup-Group
    by auto
lemma uniqueness-of-restricted-lift:
    assumes GroupHom (FreeGroup S)T\foralls\inS.T (Abs-freeletter s)=fs
    shows T}=\mathrm{ res-freeword-funlift f S
proof
    fix }
```

define $F$ where $F \equiv$ res-freeword-funlift f $S$
define $u$-Abs where $u$-Abs $\equiv \lambda a::^{\prime} a$ signed. apply-sign Abs-freeletter a
show $T x=F x$
proof (cases $x \in$ FreeGroup $S$ )
case True
have 1: set (map u-Abs (freeword $x)$ ) $\subseteq$ FreeGroup $S$
using $u$-Abs-def FreeGroupD[OF True]
Abs-freeletter-in-FreeGroup-iff $[$ of $-S$ ]
FreeGroup-uminus-closed
by auto
moreover from $u$-Abs-def have $x=\left(\sum a \leftarrow\right.$ freeword $x$. $u$-Abs $\left.a\right)$
using freeword-conv-freeletter-sum-list by fast
ultimately
have $T x=\left(\sum a \leftarrow\right.$ freeword $x . T(u$-Abs $\left.a)\right)$
$F x=\left(\sum a \leftarrow\right.$ freeword $\left.x . F(u-A b s a)\right)$
using $F$-def
GroupHom.im-sum-list-map[OF assms(1), of $u$-Abs freeword $x]$ GroupHom.im-sum-list-map [
OF hom-restrict0-freeword-funlift, of $u$-Abs freeword $x S f$ ]
by auto
moreover have $\forall a \in \operatorname{set}($ freeword $x) . T(u$-Abs $a)=F(u$-Abs $a)$
proof
fix $a$ assume $a \in \operatorname{set}$ (freeword $x$ )
moreover define $b$ where $b \equiv A b s$-freeletter ( $f s t a$ )
ultimately show $T(u$-Abs $a)=F(u$-Abs $a)$
using F-def u-Abs-def True assms(2) FreeGroupD $[$ of $x$ S $]$
GroupHom.im-uminus[OF assms(1)]
Abs-freeletter-in-FreeGroup-iff[of fst a S]
GroupHom.im-uminus[OF hom-restrict0-freeword-funlift, of b S f]
freeword-funlift-Abs-freeletter[of f]
by auto
qed
ultimately show ?thesis
using $F$-def
sum-list-map-cong[of freeword $x \lambda s . T(u-A b s s) \lambda s . F(u-A b s s)]$
by $\operatorname{simp}$
next
case False
with assms(1) F-def show ?thesis
using hom-restrict0-freeword-funlift GroupHom.supp suppI-contra[of x T]
suppI-contra[of $x$ F]
by fastforce
qed
qed
theorem FreeGroup-universal-property:
fixes $f::$ ' $a \Rightarrow$ ' $b::$ group-add

```
    shows \(\exists!T::^{\prime} a\) freeword \(\Rightarrow{ }^{\prime} b .(\forall s \in S . T(A b s\)-freeletter \(s)=f s) \wedge\)
    GroupHom (FreeGroup S) T
proof (rule ex1I, rule conjI)
    show \(\forall s \in S\). res-freeword-funlift \(f S(\) Abs-freeletter \(s)=f s\)
        using Abs-freeletter-in-FreeGroup-iff[of - S] freeword-funlift-Abs-freeletter
        by auto
    show \(\wedge T .(\forall s \in S . T(\) Abs-freeletter \(s)=f s) \wedge\)
                    GroupHom (FreeGroup \(S\) ) \(T \Longrightarrow\)
        \(T=\) restrict0 (freeword-funlift f) (FreeGroup \(S\) )
    using uniqueness-of-restricted-lift by auto
qed (rule hom-restrict0-freeword-funlift)
```


### 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 }\quadP:: ' a signed list set - the set of relator word
    assumes }P-S:ps\inP\Longrightarrowfst'set ps\subseteq
    and proper-P: ps\inP\Longrightarrow proper-signed-list ps
begin
abbreviation }\mp@subsup{P}{}{\prime}\equivAbs-freeword ' P- the set of relator
abbreviation Q \equivGroup.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 FreeGroup S
    using P-S proper-P by (blast intro: FreeGroupI-transfer)
lemma relators: }\mp@subsup{P}{}{\prime}\subseteq
    using FreeGroup-0-closed genby-genset-subset by fastforce
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 se
    and }\quadP::' 'a signed list set - the set of relator word
+ fixes f ::' 'a = 'b::group-add
    assumes lift-f-trivial-P:
        ps\inP\Longrightarrow freeword-funlift f(Abs-freeword ps)=0
begin
abbreviation lift-f \equiv freeword-funlift f
definition induced-hom :: 'a freeword set permutation }=>\mathrm{ 'b
    where induced-hom \equivGroupHom.quotient-hom (FreeGroup S)
            (restrict0 lift-f (FreeGroup S)) Q
        - the restrict0 operation is really only necessary to make GroupByPresenta-
tionInducedFun.induced-hom a GroupHom
abbreviation F\equivinduced-hom
lemma lift-f-trivial-P': p\inP' \Longrightarrow lift-f p=0
    using lift-f-trivial-P by fast
lemma lift-f-trivial-lconjby-P': p\in\mp@subsup{P}{}{\prime}\Longrightarrow lift-f (lconjby w p)=0
    using freeword-funlift-lconjby[of f] lift-f-trivial-P' by simp
lemma lift-f-trivial-Q: q\inQ \Longrightarrow lift-f q=0
proof (erule genby.induct, rule freeword-funlift-0)
```

```
    show \s. s }\in(\bigcupww FreeGroup S.lconjby w' P')\Longrightarrow lift-f s=
    using lift-f-trivial-lconjby-P' by fast
next
    fix w w' :: 'a freeword assume ww': lift-f w=0 lift-f w'=0
    have lift-f (w-\mp@subsup{w}{}{\prime})=\mathrm{ lift-f w- lift-f w'}
        using freeword-funlift-diff[of f w] by simp
    with }w\mp@subsup{w}{}{\prime}\mathrm{ show lift-f (w-w') = 0 by simp
qed
lemma lift-f-ker-Q:Q\subseteq ker lift-f
    using lift-f-trivial-Q unfolding ker-def by auto
lemma lift-f-Ker-Q:Q\subseteq GroupHom.Ker (FreeGroup S) lift-f
    using lift-f-ker-Q Q-FreeS by fast
lemma restrict0-lift-f-Ker-Q:
    Q\subseteqGroupHom.Ker (FreeGroup S)(restrict0 lift-f (FreeGroup S))
    using lift-f-Ker-Q ker-subset-ker-restrict0 by fast
lemma induced-hom-equality:
```



```
- 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
    unfolding induced-hom-def
    using GroupHom.quotient-hom-im hom-restrict0-freeword-funlift
        Q-subgroup-FreeS normal-Q restrict0-lift-f-Ker-Q
    by fastforce
lemma hom-induced-hom: GroupHom G F
    unfolding induced-hom-def
    using GroupHom.quotient-hom hom-restrict0-freeword-funlift
        Q-subgroup-FreeS normal-Q restrict0-lift-f-Ker-Q
    by fast
lemma induced-hom-Abs-freeletter-equality:
    s\inS\LongrightarrowF (\lceilFreeGroup S|Abs-freeletter s|Q\rceil)=fs
    using Abs-freeletter-in-FreeGroup-iff[of s S]
    by (simp add: induced-hom-equality freeword-funlift-Abs-freeletter)
lemma uniqueness-of-induced-hom':
    defines q \equivGroup.natural-quotient-hom(FreeGroup S) Q
    assumes GroupHom G T\foralls\inS.T (\lceilFreeGroup S|Abs-freeletter s|Q\rceil)=fs
    shows }T\circq=F\circ
proof-
    from assms have Toq= res-freeword-funlift fS
        using natural-hom natural-hom-image Abs-freeletter-in-FreeGroup-iff[of - S]
    by (force intro: uniqueness-of-restricted-lift GroupHom.comp)
    moreover from q-def have F\circq=res-freeword-funlift f S
```

```
        using induced-hom-equality GroupHom.im-zero[OF hom-induced-hom]
        by auto
    ultimately show ?thesis by simp
qed
lemma uniqueness-of-induced-hom:
    assumes GroupHom G T\foralls\inS.T (\lceilFreeGroup S|Abs-freeletter s }|Q\rceil)=f
    shows }T=
proof
    fix }
    show T }x=F
    proof (cases x\inG)
        case True
        define q where q\equivGroup.natural-quotient-hom(FreeGroup S) Q
        from True obtain w where w}\in\mathrm{ FreeGroup S x = (`FreeGroup S| w|Q`)
            using G-UN by fast
        with q-def have T }T=(T\circq)wFx=(F\circq)w by aut
        with assms q-def show ?thesis using uniqueness-of-induced-hom' by simp
    next
        case False
        with assms(1) show ?thesis
            using hom-induced-hom GroupHom.supp suppI-contra[of x T]
                suppI-contra[of x F]
            by fastforce
    qed
qed
theorem induced-hom-universal-property:
    \exists!F.GroupHom GF^(\foralls\inS.F (\lceilFreeGroup S|Abs-freeletter s |Q\rceil) =fs)
    using hom-induced-hom induced-hom-Abs-freeletter-equality
        uniqueness-of-induced-hom
    by blast
lemma induced-hom-Abs-freelist-conv-sum-list:
    ss\inlists S\LongrightarrowF(\lceilFreeGroup S|Abs-freelist ss |Q\rceil)=(\sums\leftarrowss.fs)
    by (simp add:
        Abs-freelist-in-FreeGroup induced-hom-equality freeword-funlift-Abs-freelist
        )
lemma induced-hom-surj: F}\mp@subsup{F}{}{`}G=\langle\mp@subsup{f}{}{`}S
proof (rule seteqI)
    show }\x.x\in\mp@subsup{F}{}{\prime}G\Longrightarrowx\in\langle\mp@subsup{f}{}{\prime}S
        using G-UN induced-hom-equality freeword-funlift-surj[of f S] by auto
next
    fix }x\mathrm{ assume }x\in\langle\mp@subsup{f}{}{\prime}S
    hence x \inlift-f`FreeGroup S using freeword-funlift-surj[of f S] by fast
    thus }x\in\mp@subsup{F}{}{\prime}G\mathrm{ using induced-hom-equality G-UN by force
qed
```

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 Group WithGeneratorsRelators \(=\)
    fixes \(S::\) ' \(g::\) group-add set - the set of generators
    and \(R::\) ' \(g\) list set - the set of relator words
    assumes relators: \(r s \in R \Longrightarrow r s \in\) lists \((S \cup u m i n u s\) ' \(S)\)
                                    \(r s \in R \Longrightarrow\) sum-list \(r s=0\)
                                    \(r s \in R \Longrightarrow\) proper-signed-list (map (charpair \(S\) ) rs)
begin
abbreviation \(P \equiv\) map (charpair \(S\) )' \(R\)
abbreviation \(P^{\prime} \equiv\) GroupByPresentation. \(P^{\prime} P\)
abbreviation \(Q \equiv\) GroupByPresentation. \(Q\) S P
abbreviation \(G \equiv\) GroupByPresentation. \(G\) S P
abbreviation relator-freeword \(r s \equiv A b s\)-freeword (map (charpair \(S\) ) rs)
- this maps R onto P '
abbreviation freeliftid \(\equiv\) freeword-funlift id
abbreviation induced-id :: 'g freeword set permutation \(\Rightarrow{ }^{\prime} g\)
    where induced-id \(\equiv\) GroupByPresentationInducedFun.induced-hom S P id
lemma GroupByPresentation-S-P: GroupByPresentation S P
proof
    show \(\bigwedge p s . p s \in P \Longrightarrow f s t\) ' set \(p s \subseteq S\)
        using fst-set-map-charpair-un-uminus relators(1) by fast
    show \(\bigwedge p s . p s \in P \Longrightarrow\) proper-signed-list ps using relators(3) by fast
qed
lemmas \(G\)-UN \(=\) GroupByPresentation. \(G\)-UN[OF GroupByPresentation-S-P]
lemmas \(P^{\prime}-\) Free \(S=\) GroupByPresentation. \(P^{\prime}\)-FreeS \([\) OF GroupByPresentation-S-P]
lemma freeliftid-trivial-relator-freeword- \(R\) :
    \(r s \in R \Longrightarrow\) freeliftid (relator-freeword \(r s\) ) \(=0\)
    using relators(2,3) freeword-funlift-Abs-freeword[of map (charpair S) rs id]
        sum-list-map-cong[of rs (apply-sign id) \(\circ(\) charpair \(S\) ) id]
    by \(\operatorname{simp}\)
lemma freeliftid-trivial- \(P: p s \in P \Longrightarrow\) freeliftid (Abs-freeword \(p s)=0\)
    using freeliftid-trivial-relator-freeword- \(R\) by fast
lemma GroupByPresentationInducedFun-S-P-id:
    GroupByPresentationInducedFun S P id
```

```
    by (
        intro-locales, rule GroupByPresentation-S-P,
        unfold-locales, rule freeliftid-trivial-P
    )
lemma induced-id-Abs-freelist-conv-sum-list:
    ss\inlists S\Longrightarrow induced-id (\lceilFreeGroup S|Abs-freelist ss|Q\rceil)= sum-list ss
    by (simp add:
        GroupByPresentationInducedFun.induced-hom-Abs-freelist-conv-sum-list[
        OF GroupByPresentationInducedFun-S-P-id
    ]
    )
lemma lconj-relator-freeword-R:
    \llbracketrs\inR; proper-signed-list xs; fst' set xs\subseteqS\rrbracket\Longrightarrow
        lconjby (Abs-freeword xs) (relator-freeword rs) \inQ
    by (blast intro: genby-genset-closed FreeGroupI-transfer)
lemma rconj-relator-freeword:
    assumes rs\inR proper-signed-list xs fst ' set xs \subseteqS
    shows rconjby (Abs-freeword xs) (relator-freeword rs) }\in
proof (rule genby-genset-closed, rule UN-I)
    show - Abs-freeword xs \in FreeGroup S
        using FreeGroupI-transfer[OF assms(2,3)] FreeGroup-uminus-closed by fast
    from assms(1)
        show rconjby (Abs-freeword xs) (relator-freeword rs) \in
                lconjby (- Abs-freeword xs)'Abs-freeword ' P
    by simp
qed
lemma lconjby-Abs-freelist-relator-freeword:
    |rs\inR; xs\inlists S \\Longrightarrowlconjby (Abs-freelist xs)(relator-freeword rs) \inQ
    using proper-signed-list-map-uniform-snd by (force intro: lconj-relator-freeword-R)
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: GroupHom G induced-id induced-id ' \(G=\langle S\rangle\)
using GroupByPresentationInducedFun.hom-induced-hom[
OF GroupByPresentationInducedFun-S-P-id
]
GroupByPresentationInducedFun.induced-hom-surj[
OF GroupByPresentationInducedFun-S-P-id
]
by auto
end
locale GroupPresentation \(=\) Group WithGeneratorsRelators \(S R\)
```

```
    for S :: ' g::group-add set - the set of generators
    and R :: 'g 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\mathrm{ lists S C inv-induced-id (sum-list ss) = ( }\\mathrm{ FreeGroup S|Abs-freelist ss |Q`)
    using G-UN induced-id-inj induced-id-Abs-freelist-conv-sum-list
        Abs-freelist-in-FreeGroup
    by (blast intro: the-inv-into-f-eq)
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 \inlists A ^ sum-list as=a
definition reduced-word-for :: 'a set }=>\mp@subsup{}{}{\prime}a=>\mp@subsup{}{}{\prime}a\mathrm{ list }=>\mathrm{ bool
    where reduced-word-for A a as \equivis-arg-min length (word-for A a) as
abbreviation reduced-word A as \equiv reduced-word-for A (sum-list as) as
abbreviation reduced-words-for A a \equivCollect (reduced-word-for A a)
abbreviation reduced-letter-set :: 'a set }=>\mp@subsup{}{}{\prime}'a=>\mathrm{ 'a set
    where reduced-letter-set A a \equiv\bigcup( set '(reduced-words-for A a))
    - will be empty if }a\mathrm{ is not in the set generated by }
definition word-length :: 'a set }=>\mp@subsup{}{}{\prime}a=>\mathrm{ nat
    where word-length A a \equiv length (arg-min length (word-for A a))
lemma reduced-word-forI:
    assumes as t lists A sum-list as =a
        \bs.bs flists A \Longrightarrow sum-list bs =a \Longrightarrow length as \leq length bs
    shows reduced-word-for A a as
    using assms
    unfolding reduced-word-for-def
    by (force intro: is-arg-minI)
lemma reduced-word-forI-compare:
    \llbracket reduced-word-for A a as; bs \in lists A; sum-list bs =a; length bs = length as\rrbracket
        \Longrightarrow ~ r e d u c e d - w o r d - f o r ~ A ~ a ~ b s ~
    using reduced-word-for-def is-arg-min-eq[of length] by fast
```

```
lemma reduced-word-for-lists: reduced-word-for A a as \Longrightarrow as flists A
    using reduced-word-for-def is-arg-minD1 by fast
lemma reduced-word-for-sum-list: reduced-word-for A a as \Longrightarrowsum-list as =a
    using reduced-word-for-def is-arg-minD1 by fast
lemma reduced-word-for-minimal:
    \llbracket reduced-word-for A a as; bs \inlists A; sum-list bs =a\rrbracket\Longrightarrow
        length as \leq length bs
    using reduced-word-for-def is-arg-minD2[of length]
    by fastforce
lemma reduced-word-for-length:
    reduced-word-for A a as \Longrightarrow length as = word-length A a
    unfolding word-length-def reduced-word-for-def is-arg-min-def
    by (fastforce intro: arg-min-equality[THEN sym])
lemma reduced-word-for-eq-length:
    reduced-word-for A a as \Longrightarrow reduced-word-for A a bs \Longrightarrowlength as = length bs
    using reduced-word-for-length by simp
lemma reduced-word-for-arg-min:
    as}\in\mathrm{ lists }A\Longrightarrow\mathrm{ sum-list as =a }
        reduced-word-for A a (arg-min length (word-for A a))
    using is-arg-min-arg-min-nat[of word-for A a]
    unfolding reduced-word-for-def
    by fast
lemma nil-reduced-word-for-0: reduced-word-for A 0 []
    by (auto intro: reduced-word-forI)
lemma reduced-word-for-0-imp-nil: reduced-word-for A 0 as \Longrightarrowas=[]
    using nil-reduced-word-for-0[of A] reduced-word-for-minimal[of A 0 as]
    unfolding reduced-word-for-def is-arg-min-def
    by (metis (mono-tags, opaque-lifting) length-0-conv length-greater-0-conv)
lemma not-reduced-word-for:
    \llbracketbs\inlists A; sum-list bs =a; length bs < length as \rrbracket\Longrightarrow
        \neg reduced-word-for A a as
    using reduced-word-for-minimal by fastforce
lemma reduced-word-for-imp-reduced-word:
    reduced-word-for A a as \Longrightarrow reduced-word A as
unfolding reduced-word-for-def is-arg-min-def
by (fast intro: reduced-word-forI)
lemma sum-list-zero-nreduced:
    as }\not=[]\Longrightarrow\mathrm{ sum-list as = 0 C ᄀ reduced-word A as
```

```
    using not-reduced-word-for[of []] by simp
lemma order2-nreduced: a+a=0\Longrightarrow\neg reduced-word A [a,a]
    using sum-list-zero-nreduced by simp
lemma reduced-word-append-reduce-contra1:
    assumes \neg reduced-word A as
    shows \neg reduced-word A (as@bs)
proof (cases as \in lists A bs \in lists A rule: two-cases)
    case both
    define cs where cs:cs \equivARG-MIN length cs.cs \in lists A ^ sum-list cs =
sum-list as
    with both(1) have reduced-word-for A (sum-list as) cs
        using reduced-word-for-def is-arg-min-arg-min-nat[of word-for A (sum-list as)]
        by auto
    with assms both show ?thesis
        using reduced-word-for-lists reduced-word-for-sum-list
                reduced-word-for-minimal[of A sum-list as cs as]
                reduced-word-forI-compare[of A sum-list as cs as]
                not-reduced-word-for[of cs@bs A sum-list (as@bs)]
            by fastforce
next
    case one thus ?thesis using reduced-word-for-lists by fastforce
next
    case other thus ?thesis using reduced-word-for-lists by fastforce
next
    case neither thus ?thesis using reduced-word-for-lists by fastforce
qed
lemma reduced-word-append-reduce-contra2:
    assumes \neg reduced-word A bs
    shows \neg reduced-word A(as@bs)
proof (cases as \in lists A bs \in lists A rule: two-cases)
    case both
    define cs where cs:cs\equivARG-MIN length cs.cs \in lists A ^ sum-list cs =
sum-list bs
    with both(2) have reduced-word-for A (sum-list bs) cs
        using reduced-word-for-def is-arg-min-arg-min-nat[of word-for A (sum-list bs) ]
        by auto
    with assms both show ?thesis
        using reduced-word-for-lists reduced-word-for-sum-list
            reduced-word-for-minimal[of A sum-list bs cs bs]
                        reduced-word-forI-compare[of A sum-list bs cs bs]
                not-reduced-word-for[of as@cs A sum-list (as@bs)]
        by fastforce
next
    case one thus ?thesis using reduced-word-for-lists by fastforce
next
    case other thus ?thesis using reduced-word-for-lists by fastforce
```

```
next
    case neither thus ?thesis using reduced-word-for-lists by fastforce
qed
lemma contains-nreduced-imp-nreduced:
    \neg reduced-word A bs \Longrightarrow ᄀ reduced-word A (as@bs@cs)
    using reduced-word-append-reduce-contra1 reduced-word-append-reduce-contra2
    by fast
lemma contains-order2-nreduced: a+a=0\Longrightarrow ᄀ reduced-word A(as@[a,a]@bs)
    using order2-nreduced contains-nreduced-imp-nreduced by fast
lemma reduced-word-Cons-reduce-contra:
    \neg \text { reduced-word A as } \Longrightarrow \neg \text { reduced-word A (a\#as)}
    using reduced-word-append-reduce-contra2[of A as [a]] by simp
lemma reduced-word-Cons-reduce: reduced-word A (a#as) \Longrightarrow reduced-word A as
    using reduced-word-Cons-reduce-contra by fast
lemma reduced-word-singleton:
    assumes }a\inA\quada\not=
    shows reduced-word A [a]
proof (rule reduced-word-forI)
    from assms(1) show [a]\in lists A by simp
next
    fix bs assume bs:bs\inlists A sum-list bs = sum-list [a]
    with assms(2) show length [a] \leq length bs by (cases bs) auto
qed simp
lemma el-reduced:
    assumes 0\not\inA as\inlists A sum-list as }\inA\mathrm{ reduced-word A as
    shows length as = 1
proof
    define }n\mathrm{ where n: n}\equiv\mathrm{ length as
    from assms(3) obtain a where [a]\inlists A sum-list as = sum-list [a] by auto
    with n assms(1,3,4) have n\leq1 n>0
        using reduced-word-for-minimal[of A - as [a]] by auto
    hence }n=1\mathrm{ by simp
    with n show ?thesis by fast
qed
lemma reduced-letter-set-0: reduced-letter-set A 0={}
    using reduced-word-for-0-imp-nil by simp
lemma reduced-letter-set-subset:reduced-letter-set A a\subseteqA
    using reduced-word-for-lists by fast
lemma reduced-word-forI-length:
    \llbracketas \in lists A; sum-list as = a; length as = word-length A a \rrbracket\Longrightarrow
```

reduced-word-for A a as
using reduced-word-for-arg-min reduced-word-for-length
reduced-word-forI-compare[of A a-as]
by fastforce
lemma word-length-le:
as $\in$ lists $A \Longrightarrow$ sum-list as $=a \Longrightarrow$ word-length $A \quad a \leq$ length as
using reduced-word-for-arg-min reduced-word-for-length
reduced-word-for-minimal[of $A]$
by fastforce
lemma reduced-word-forI-length':
$\llbracket$ as $\in$ lists $A$; sum-list as $=a ;$ length as $\leq$ word-length $A a \rrbracket \Longrightarrow$ reduced-word-for $A$ a as
using word-length-le[of as A] reduced-word-forI-length[of as A] by fastforce
lemma word-length-lt:
as $\in$ lists $A \Longrightarrow$ sum-list as $=a \Longrightarrow \neg$ reduced-word-for $A$ a as $\Longrightarrow$ word-length $A \quad a<$ length as
using reduced-word-forI-length' by fastforce
end
lemma in-genby-reduced-letter-set: assumes as lists $A$ sum-list as $=a$ shows $\quad a \in\langle$ reduced-letter-set $A a\rangle$
proof-
define $x s$ where $x s: x s \equiv$ arg-min length (word-for $A$ a)
with assms have $x s \in$ lists (reduced-letter-set $A$ a) sum-list xs $=a$
using reduced-word-for-arg-min [of as A] reduced-word-for-sum-list by auto
thus ?thesis using genby-eq-sum-lists by force
qed
lemma reduced-word-for-genby-arg-min:
fixes $A:: ' a:: g r o u p-a d d$ set
defines $B \equiv A \cup$ uminus ' $A$
assumes $a \in\langle A\rangle$
shows reduced-word-for $B$ a (arg-min length (word-for Ba))
using assms genby-eq-sum-lists[of $A$ ] reduced-word-for-arg-min $[o f-B a]$
by auto
lemma reduced-word-for-genby-sym-arg-min:
assumes uminus ' $A \subseteq A a \in\langle A\rangle$
shows reduced-word-for A a (arg-min length (word-for A a))
proof-
from assms(1) have $A=A \cup$ uminus ' $A$ by auto
with assms(2) show ?thesis
using reduced-word-for-genby-arg-min $[$ of $a A]$ by simp
qed

```
lemma in-genby-imp-in-reduced-letter-set:
    fixes A :: 'a::group-add set
    defines }B\equivA\cupuminus ' 
    assumes }a\in\langleA
    shows }\quada\in\langlereduced-letter-set B a
    using assms genby-eq-sum-lists[of A] in-genby-reduced-letter-set[of - B]
    by auto
lemma in-genby-sym-imp-in-reduced-letter-set:
    uminus ' }A\subseteqA\Longrightarrowa\in\langleA\rangle\Longrightarrowa\in\langlereduced-letter-set A a
    using in-genby-imp-in-reduced-letter-set by (fastforce simp add: Un-absorb2)
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 $y \triangleleft x \equiv \exists v . v \notin y \wedge x=$ insert $v y$
lemma facetrelI: $v \notin y \Longrightarrow x=$ insert $v y \Longrightarrow y \triangleleft x$ using facetrel-def by fast
lemma facetrelI-card: $y \subseteq x \Longrightarrow \operatorname{card}(x-y)=1 \Longrightarrow y \triangleleft x$ using card1 $[$ of $x-y]$ by (blast intro: facetrelI)
lemma facetrel-complement-vertex: $y \triangleleft x \Longrightarrow x=$ insert $v y \Longrightarrow v \notin y$ using facetrel-def $[$ of $y x]$ by fastforce

```
lemma facetrel-diff-vertex:v\inx\Longrightarrowx-{v} \triangleleftx
    by (auto intro: facetrelI)
lemma facetrel-conv-insert: }y\triangleleftx\Longrightarrowv\inx-y\Longrightarrowx=\mathrm{ insert v y
    unfolding facetrel-def by fast
lemma facetrel-psubset: }y\triangleleftx\Longrightarrowy\subset
    unfolding facetrel-def by fast
lemma facetrel-subset: }y\triangleleftx\Longrightarrowy\subseteq
    using facetrel-psubset by fast
lemma facetrel-card: }y\triangleleftx\Longrightarrow\mathrm{ card ( }x-y)=
    using insert-Diff-if[of - y y] unfolding facetrel-def by fastforce
lemma finite-facetrel-card: finite }x\Longrightarrowy\triangleleftx\Longrightarrow\mathrm{ card x = Suc (card y)
    using facetrel-def[of y x] card-insert-disjoint[of x] by auto
lemma facetrelI-cardSuc: z\subseteqx \Longrightarrow card x=Suc (card z)\Longrightarrowz\triangleleftx
    using card-ge-0-finite finite-subset[of z] card-Diff-subset[of z x]
    by (force intro: facetrelI-card)
lemma facet2-subset:\llbracketz\triangleleftx;z\trianglelefty;x\capy-z\not={}\rrbracket\Longrightarrowx\subseteqy
    unfolding facetrel-def by force
lemma inj-on-pullback-facet:
    assumes inj-on fx z\triangleleftf'x
    obtains }y\mathrm{ where }y\triangleleftx\mp@subsup{f}{}{\prime}y=
proof
    from assms(2) obtain v where v:v\not\inz f
        using facetrel-def[of z] by auto
    define }u\mathrm{ and }y\mathrm{ where }u\equiv\mathrm{ the-inv-into xf v and y: y 三{v<x.fv f z}
    moreover with assms(2) v have x = insert u y
        using the-inv-into-f-eq[OF assms(1)] the-inv-into-into[OF assms(1)]
        by fastforce
    ultimately show }y\triangleleft
        using vf-the-inv-into-f[OF assms(1)] by (force intro: facetrelI)
    from y assms(2) show f'y =z using facetrel-subset by fast
qed
```


### 3.1.2 Adjacency

```
definition adjacent :: 'a set \(\Rightarrow\) 'a set \(\Rightarrow\) bool (infix ~ 70)
    where }x~y\equiv\existsz.z\triangleleftx\wedgez\triangleleft
lemma adjacentI: z\triangleleftx\Longrightarrowz\trianglelefty\Longrightarrowx~y
    using adjacent-def by fast
lemma empty-not-adjacent: }\neg{}~
```

unfolding facetrel-def adjacent-def by fast
lemma adjacent-sym: $x \sim y \Longrightarrow y \sim x$
unfolding adjacent-def by fast
lemma adjacent-refl:
assumes $x \neq\{ \}$
shows $x \sim x$
proof-
from assms obtain $v$ where $v: v \in x$ by fast
thus $x \sim x$ using facetrelI [of $v x-\{v\}]$ unfolding adjacent-def by fast
qed
lemma common-facet: $\llbracket z \triangleleft x ; z \triangleleft y ; x \neq y \rrbracket \Longrightarrow z=x \cap y$
using facetrel-subset facet2-subset by fast
lemma adjacent-int-facet1: $x \sim y \Longrightarrow x \neq y \Longrightarrow(x \cap y) \triangleleft x$ using common-facet unfolding adjacent-def by fast
lemma adjacent-int-facet2: $x \sim y \Longrightarrow x \neq y \Longrightarrow(x \cap y) \triangleleft y$ using adjacent-sym adjacent-int-facet1 by (fastforce simp add: Int-commute)
lemma adjacent-conv-insert: $x \sim y \Longrightarrow v \in x-y \Longrightarrow x=$ insert $v(x \cap y)$ using adjacent-int-facet1 facetrel-conv-insert by fast
lemma adjacent-int-decomp:
$x \sim y \Longrightarrow x \neq y \Longrightarrow \exists v . v \notin y \wedge x=$ insert $v(x \cap y)$
using adjacent-int-facet1 unfolding facetrel-def by fast
lemma adj-antivertex:
assumes $x \sim y x \neq y$
shows $\exists!v . v \in x-y$
proof (rule ex-ex1I)
from assms obtain $w$ where $w: w \notin y x=$ insert $w(x \cap y)$ using adjacent-int-decomp by fast
thus $\exists v . v \in x-y$ by auto
from $w$ have $\bigwedge v . v \in x-y \Longrightarrow v=w$ by fast
thus $\bigwedge v v^{\prime} . v \in x-y \Longrightarrow v^{\prime} \in x-y \Longrightarrow v=v^{\prime}$ by auto
qed
lemma adjacent-card: $x \sim y \Longrightarrow$ card $x=$ card $y$
unfolding adjacent-def facetrel-def by (cases finite $x$ x=y rule: two-cases) auto
lemma adjacent-to-adjacent-int-subset:
assumes $C \sim D f^{\bullet} C \sim f^{\bullet} D f^{\prime} C \neq f^{\bullet} D$
shows $\quad f^{\bullet} C \cap f^{\bullet} D \subseteq f^{\prime}(C \cap D)$

## proof

from $\operatorname{assms}(1,3)$ obtain $v$ where $v: v \notin D C=$ insert $v(C \cap D)$
using adjacent-int-decomp by fast

```
    from \(\operatorname{assms}(2,3)\) obtain \(w\) where \(w: w \notin f^{\prime} D f^{`} C=\) insert \(w\left(f^{‘} C \cap f^{`} D\right)\)
    using adjacent-int-decomp \(\left[\right.\) of \(\left.f^{‘} C f^{\prime} D\right]\) by fast
    from \(w\) have \(w^{\prime}: w \in f^{‘} C-f^{\prime} D\) by fast
    with \(v\) assms \((1,2)\) have \(f v\)-w: \(f v=w\) using adjacent-conv-insert by fast
    fix \(b\) assume \(b \in f^{`} C \cap f^{\prime} D\)
    from this obtain a1 a2
        where \(a 1: a 1 \in C b=f a 1\)
        and \(\quad a 2: a 2 \in D b=f a 2\)
        by fast
    from \(v\) a1 \(a 2(2)\) have \(a 1 \notin D \Longrightarrow f a 2=w\) using \(f v-w\) by auto
    with \(a 2(1) w^{\prime}\) have \(a 1 \in D\) by fast
    with \(a 1\) show \(b \in f^{\prime}(C \cap D)\) by fast
qed
lemma adjacent-to-adjacent-int:
```



```
    using adjacent-to-adjacent-int-subset by fast
```


### 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 \((x \# x s @[y]) \Longrightarrow\) card \(x=\) card \(y\) using adjacent-card by (induct xs arbitrary: x) auto
```


### 3.2 Locale and basic facts

```
locale SimplicialComplex =
```

locale SimplicialComplex =
fixes }X :: 'a set se
fixes }X :: 'a set se
assumes finite-simplices: }\forallx\inX\mathrm{ . finite }
assumes finite-simplices: }\forallx\inX\mathrm{ . finite }
and faces }:x\inX\Longrightarrowy\subseteqx\Longrightarrowy\in
and faces }:x\inX\Longrightarrowy\subseteqx\Longrightarrowy\in
context SimplicialComplex
context SimplicialComplex
begin
begin
abbreviation Subcomplex Y}<br>\X\wedge SimplicialComplex Y
abbreviation Subcomplex Y}<br>\X\wedge SimplicialComplex Y
definition maxsimp }x\equivx\inX\wedge(\forallz\inX.x\subseteqz\longrightarrowz=x
definition maxsimp }x\equivx\inX\wedge(\forallz\inX.x\subseteqz\longrightarrowz=x
definition adjacentset :: 'a set }=>\mathrm{ ' 'a set set
definition adjacentset :: 'a set }=>\mathrm{ ' 'a set set
where adjacentset }x={y\inX.x~y
where adjacentset }x={y\inX.x~y
lemma finite-simplex: }x\inX\Longrightarrow\mathrm{ finite }
lemma finite-simplex: }x\inX\Longrightarrow\mathrm{ finite }
using finite-simplices by simp
using finite-simplices by simp
lemma singleton-simplex: v\in\bigcupX\Longrightarrow{v}\inX

```
lemma singleton-simplex: v\in\bigcupX\Longrightarrow{v}\inX
```

using faces by auto
lemma maxsimpI: $x \in X \Longrightarrow(\bigwedge z . z \in X \Longrightarrow x \subseteq z \Longrightarrow z=x) \Longrightarrow$ maxsimp $x$ using maxsimp-def by auto
lemma maxsimpD-simplex: maxsimp $x \Longrightarrow x \in X$ using maxsimp-def by fast
lemma maxsimpD-maximal: maxsimp $x \Longrightarrow z \in X \Longrightarrow x \subseteq z \Longrightarrow z=x$ using maxsimp-def by auto
lemmas finite-maxsimp $=$ finite-simplex $[$ OF maxsimpD-simplex $]$
lemma maxsimp-nempty: $X \neq\{\{ \}\} \Longrightarrow \operatorname{maxsimp} x \Longrightarrow x \neq\{ \}$ unfolding maxsimp-def by fast
lemma maxsimp-vertices: maxsimp $x \Longrightarrow x \subseteq \bigcup X$ using maxsimpD-simplex by fast
lemma adjacentsetD-adj: $y \in$ adjacentset $x \Longrightarrow x \sim y$ using adjacentset-def by fast
lemma max-in-subcomplex:
$\llbracket$ Subcomplex $Y ; y \in Y$; maxsimp $y \rrbracket \Longrightarrow$ SimplicialComplex.maxsimp $Y$ y using maxsimpD-maximal by (fast intro: SimplicialComplex.maxsimpI)
lemma face-im:
assumes $w \in X y \subseteq f^{\prime} w$
defines $u \equiv\{a \in w . f a \in y\}$
shows $y \in f \vdash X$
using assms faces[of $w u$ ] image-eqI[of y (') fuX]
by fast
lemma im-faces: $x \in f \vdash X \Longrightarrow y \subseteq x \Longrightarrow y \in f \vdash X$ using faces face-im[of $-y]$ by (cases $y=\{ \}$ ) auto
lemma map-is-simplicial-morph: SimplicialComplex $(f \vdash X)$
proof
show $\forall x \in f \vdash X$. finite $x$ using finite-simplices by fast show $\backslash x y . x \in f \vdash X \Longrightarrow y \subseteq x \Longrightarrow y \in f \vdash X$ using im-faces by fast
qed
lemma vertex-set-int:
assumes SimplicialComplex Y
shows $\bigcup(X \cap Y)=\bigcup X \cap \bigcup Y$
proof
have $\bigwedge v . v \in \bigcup X \cap \bigcup Y \Longrightarrow v \in \bigcup(X \cap Y)$
using faces SimplicialComplex.faces[OF assms] by auto
thus $\bigcup(X \cap Y) \supseteq \bigcup X \cap \bigcup Y$ by fast

```
qed auto
```

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 $x s \equiv(\forall x \in$ set $x s$. maxsimp $x) \wedge$ adjacentchain $x s$ definition pmaxsimpchain $x s \equiv(\forall x \in$ set $x s . \operatorname{maxsimp} x) \wedge$ padjacentchain $x s$
function min-maxsimpchain :: 'a set list $\Rightarrow$ bool
where
min-maxsimpchain []$=$ True
$\mid$ min-maxsimpchain $[x]=$ maxsimp $x$
$\mid$ min-maxsimpchain $(x \# x s @[y])=$
$(x \neq y \wedge i s$-arg-min length $(\lambda z s$. maxsimpchain $(x \# z s @[y])) x s)$
by (auto, rule list-cases-Cons-snoc)
termination by (relation measure length) auto
lemma maxsimpchain-snocI:
$\llbracket$ maxsimpchain $(x s @[x]) ;$ maxsimp $y ; x \sim y \rrbracket \Longrightarrow$ maxsimpchain $(x s @[x, y])$
using maxsimpchain-def binrelchain-snoc maxsimpchain-def by auto
lemma maxsimpchainD-maxsimp:
maxsimpchain $x s \Longrightarrow x \in$ set $x s \Longrightarrow$ maxsimp $x$
using maxsimpchain-def by fast
lemma maxsimpchainD-adj: maxsimpchain $x s \Longrightarrow$ adjacentchain xs using maxsimpchain-def by fast
lemma maxsimpchain-CConsI:
$\llbracket$ maxsimp $w$; maxsimpchain $(x \# x s) ; w \sim x \rrbracket \Longrightarrow$ maxsimpchain $(w \# x \# x s)$
using maxsimpchain-def by auto
lemma maxsimpchain-Cons-reduce:
maxsimpchain $(x \# x s) \Longrightarrow$ maxsimpchain $x s$
using adjacentchain-Cons-reduce maxsimpchain-def by fastforce
lemma maxsimpchain-append-reduce1:
maxsimpchain (xs@ys) $\Longrightarrow$ maxsimpchain xs
using binrelchain-append-reduce1 maxsimpchain-def by auto
lemma maxsimpchain-append-reduce2:
maxsimpchain (xs@ys) $\Longrightarrow$ maxsimpchain ys
using binrelchain-append-reduce2 maxsimpchain-def by auto
lemma maxsimpchain-remdup-adj:
maxsimpchain (xs@ $x, x] @ y s) \Longrightarrow$ maxsimpchain $(x s @[x] @ y s)$
using maxsimpchain-def binrelchain-remdup-adj by auto
lemma maxsimpchain-rev: maxsimpchain $x s \Longrightarrow$ maxsimpchain (rev $x s$ )
using maxsimpchainD-maxsimp adjacent-sym
binrelchain-sym-rev[of adjacent]
unfolding maxsimpchain-def
by fastforce
lemma maxsimpchain-overlap-join:
maxsimpchain $(x s @[w]) \Longrightarrow$ maxsimpchain $(w \# y s) \Longrightarrow$ maxsimpchain (xs@w\#ys)
using binrelchain-overlap-join maxsimpchain-def by auto
lemma pmaxsimpchain: pmaxsimpchain $x s \Longrightarrow$ maxsimpchain xs using maxsimpchain-def pmaxsimpchain-def by fast
lemma pmaxsimpchainI-maxsimpchain: maxsimpchain $x s \Longrightarrow$ distinct $x s \Longrightarrow$ pmaxsimpchain $x s$ using maxsimpchain-def pmaxsimpchain-def by fast
lemma pmaxsimpchain-CConsI:
$\llbracket$ maxsimp $w ;$ pmaxsimpchain $(x \# x s) ; w \sim x ; w \notin \operatorname{set}(x \# x s) \rrbracket \Longrightarrow$ pmaxsimpchain ( $w \# x \# x s$ )
using pmaxsimpchain-def by auto
lemmas pmaxsimpchainD-maxsimp $=$ maxsimpchainD-maxsimp[OF pmaxsimpchain]
lemmas pmaxsimpchainD-adj $=$ maxsimpchainD-adj [OF pmaxsimpchain]
lemma pmaxsimpchainD-distinct: pmaxsimpchain $x s \Longrightarrow$ distinct $x s$ using pmaxsimpchain-def by fast
lemma pmaxsimpchain-Cons-reduce: pmaxsimpchain $(x \# x s) \Longrightarrow$ pmaxsimpchain $x s$ using maxsimpchain-Cons-reduce pmaxsimpchain pmaxsimpchainD-distinct by (fastforce intro: pmaxsimpchainI-maxsimpchain)

```
lemma pmaxsimpchain-append-reduce1:
    pmaxsimpchain (xs@ys) \Longrightarrow pmaxsimpchain xs
    using maxsimpchain-append-reduce1 pmaxsimpchain pmaxsimpchainD-distinct
    by (fastforce intro: pmaxsimpchainI-maxsimpchain)
lemma maxsimpchain-obtain-pmaxsimpchain:
    assumes }x\not=y\mathrm{ maxsimpchain ( }x#xs@[y]
    shows \existsys. set ys \subseteq set xs ^ length ys \leqlength xs ^
            pmaxsimpchain(x#ys@[y])
proof -
    obtain ys
        where ys: set ys \subseteq set xs length ys \leq length xs padjacentchain (x#ys@[y])
        using maxsimpchainD-adj[OF assms(2)]
            adjacentchain-obtain-proper[OF assms(1)]
        by auto
    from ys(1) assms(2) have }\foralla\in\operatorname{set}(x#ys@[y]). maxsimp a
        using maxsimpchainD-maxsimp by auto
    with ys show ?thesis unfolding pmaxsimpchain-def by auto
qed
lemma min-maxsimpchainD-maxsimpchain:
    assumes min-maxsimpchain xs
    shows maxsimpchain xs
proof (cases xs rule: list-cases-Cons-snoc)
    case Nil thus ?thesis using maxsimpchain-def by simp
next
    case Single with assms show ?thesis using maxsimpchain-def by simp
next
    case Cons-snoc with assms show ?thesis using is-arg-minD1 by fastforce
qed
lemma min-maxsimpchainD-min-betw:
    min-maxsimpchain (x#xs@[y])\Longrightarrow maxsimpchain (x#ys@[y])\Longrightarrow
        length ys \geq length xs
    using is-arg-minD2 by fastforce
lemma min-maxsimpchainI-betw:
    assumes }x\not=y\mathrm{ maxsimpchain ( }x#xs@[y]
    \ \ y s . ~ m a x s i m p c h a i n ~ ( x \# y s @ [ y ] ) \Longrightarrow ~ l e n g t h ~ x s ~ \leq l e n g t h ~ y s ~
    shows min-maxsimpchain (x#xs@[y])
    using assms by (simp add: is-arg-min-linorderI)
lemma min-maxsimpchainI-betw-compare:
    assumes }x\not=y\mathrm{ maxsimpchain (x#xs@[y])
        min-maxsimpchain (x#ys@[y]) length xs = length ys
    shows min-maxsimpchain (x#xs@[y])
    using assms min-maxsimpchainD-min-betw min-maxsimpchainI-betw
    by auto
```

```
lemma min-maxsimpchain-pmaxsimpchain:
    assumes min-maxsimpchain xs
    shows pmaxsimpchain xs
proof (
    rule pmaxsimpchainI-maxsimpchain, rule min-maxsimpchainD-maxsimpchain,
    rule assms, cases xs rule: list-cases-Cons-snoc
)
    case (Cons-snoc x ys y)
    have ᄀ distinct (x#ys@[y])\Longrightarrow False
    proof (cases x\inset ys }y\in\mathrm{ set ys rule: two-cases)
    case both
    from both(1) obtain as bs where ys =as@x#bs
        using in-set-conv-decomp[of x ys] by fast
    with assms Cons-snoc show False
        using min-maxsimpchainD-maxsimpchain[OF assms]
                    maxsimpchain-append-reduce2[of x#as]
                min-maxsimpchainD-min-betw[of x ys y]
        by fastforce
    next
    case one
    from one(1) obtain as bs where ys =as@x#bs
        using in-set-conv-decomp[of x ys] by fast
    with assms Cons-snoc show False
        using min-maxsimpchainD-maxsimpchain[OF assms]
                maxsimpchain-append-reduce2[of x#as]
                min-maxsimpchainD-min-betw[of x ys y]
        by fastforce
    next
    case other
    from other(2) obtain as bs where ys =as@y#bs
        using in-set-conv-decomp[of y ys] by fast
    with assms Cons-snoc show False
        using min-maxsimpchainD-maxsimpchain[OF assms]
            maxsimpchain-append-reduce1[of x#as@[y]]
                min-maxsimpchainD-min-betw[of x ys y]
        by fastforce
    next
    case neither
    moreover assume }\neg\mathrm{ distinct (x#ys @ [y])
    ultimately obtain as a bs cs where ys=as@[a]@bs@[a]@cs
        using assms Cons-snoc not-distinct-decomp[of ys] by auto
    with assms Cons-snoc show False
        using min-maxsimpchainD-maxsimpchain[OF assms]
                        maxsimpchain-append-reduce1[of x#as@[a]]
                        maxsimpchain-append-reduce2[of x#as@[a]@bs a#cs@[y]]
                        maxsimpchain-overlap-join[of x# as a cs@[y]]
                        min-maxsimpchainD-min-betw[of x ys y as@a#cs]
        by auto
```

```
    qed
    with Cons-snoc show distinct xs by fast
qed auto
lemma min-maxsimpchain-rev:
    assumes min-maxsimpchain xs
    shows min-maxsimpchain (rev xs)
proof (cases xs rule: list-cases-Cons-snoc)
    case Single with assms show ?thesis
        using min-maxsimpchainD-maxsimpchain maxsimpchainD-maxsimp by simp
next
    case (Cons-snoc x ys y)
    moreover have min-maxsimpchain (y # rev ys @ [x])
    proof (rule min-maxsimpchainI-betw)
        from Cons-snoc assms show }y\not=
            using min-maxsimpchain-pmaxsimpchain pmaxsimpchainD-distinct by auto
    from Cons-snoc show maxsimpchain (y # rev ys @ [x])
            using min-maxsimpchainD-maxsimpchain[OF assms] maxsimpchain-rev
            by fastforce
    from Cons-snoc assms
            show \\zs. maxsimpchain (y#zs@[x])\Longrightarrow length (rev ys) \leq length zs
            using maxsimpchain-rev min-maxsimpchainD-min-betw[of x ys y]
            by fastforce
    qed
    ultimately show ?thesis by simp
qed simp
lemma min-maxsimpchain-adj:
    \llbracket maxsimp x; maxsimp y;x~y;x\not=y\rrbracket\Longrightarrow min-maxsimpchain [x,y]
    using maxsimpchain-def min-maxsimpchainI-betw[of x y []] by simp
lemma min-maxsimpchain-betw-CCons-reduce:
    assumes min-maxsimpchain (w#x#ys@[z])
    shows min-maxsimpchain (x#ys@[z])
proof (rule min-maxsimpchainI-betw)
    from assms show }x\not=
        using min-maxsimpchain-pmaxsimpchain pmaxsimpchainD-distinct
        by fastforce
    show maxsimpchain (x#ys@[z])
        using min-maxsimpchainD-maxsimpchain[OF assms]
            maxsimpchain-Cons-reduce
        by fast
next
    fix zs assume maxsimpchain (x#zs@[z])
    hence maxsimpchain(w#x#zs@[z])
        using min-maxsimpchainD-maxsimpchain[OF assms] maxsimpchain-def
        by fastforce
    with assms show length ys \leq length zs
    using min-maxsimpchainD-min-betw[of wx#ys zx#zs] by simp
```


## qed

lemma min-maxsimpchain-betw-uniform-length:
assumes min-maxsimpchain ( $x \# x s @[y]$ ) min-maxsimpchain $(x \# y s @[y])$
shows length $x s=$ length ys
using min-maxsimpchainD-min-betw[OF assms(1)] min-maxsimpchainD-min-betw[OF assms(2)] min-maxsimpchainD-maxsimpchain $[$ OF assms (1)] min-maxsimpchainD-maxsimpchain[OF assms(2)]
by fastforce
lemma not-min-maxsimpchainI-betw:
$\llbracket$ maxsimpchain $(x \# y s @[y])$; length $y s<$ length $x s \rrbracket \Longrightarrow$ $\neg$ min-maxsimpchain $(x \# x s @[y])$
using min-maxsimpchainD-min-betw not-less by blast
lemma maxsimpchain-in-subcomplex:
$\llbracket$ Subcomplex $Y$; set ys $\subseteq Y$; maxsimpchain ys $\rrbracket$ SimplicialComplex.maxsimpchain $Y$ ys
using maxsimpchain-def max-in-subcomplex SimplicialComplex.maxsimpchain-def
by force
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 $f::{ }^{\prime} a \Rightarrow{ }^{\prime} b$
assumes $\operatorname{inj}: \operatorname{inj}$-on $f(\bigcup X)$
begin
lemmas morph $=$ map-is-simplicial-morph $[$ of $f]$
lemma iso-codim-map:
$x \in X \Longrightarrow y \in X \Longrightarrow \operatorname{card}\left(f^{\iota} x-f^{\iota} y\right)=\operatorname{card}(x-y)$
using inj inj-on-image-set-diff [of $f-x y]$ finite-simplex subset-inj-on[of $f-x-y$ ] inj-on-iff-eq-card[of $x-y$ ]
by fastforce
lemma maxsimp-im-max: maxsimp $x \Longrightarrow w \in X \Longrightarrow f^{\prime} x \subseteq f^{\prime} w \Longrightarrow f^{\prime} w=f^{\prime} x$
using maxsimpD-simplex inj-onD[OF inj] maxsimpD-maximal $[$ of $x w]$ by blast

```
lemma maxsimp-map:
    maxsimp x \Longrightarrow SimplicialComplex.maxsimp (f\vdashX) (f'x)
    using maxsimpD-simplex maxsimp-im-max morph
        SimplicialComplex.maxsimpI[of f\vdash
    by fastforce
lemma iso-adj-int-im:
    assumes maxsimp x maxsimp y }x~yx\not=
    shows (ff}x\cap\mp@subsup{f}{}{\prime}y)\triangleleft\mp@subsup{f}{}{\prime}
proof (rule facetrelI-card)
    from assms(1,\mathcal{Q}) have 1: f'}x\subseteq\mp@subsup{f}{}{\prime}y\Longrightarrow\mp@subsup{f}{}{\prime}y=\mp@subsup{f}{}{\prime}
        using maxsimp-map SimplicialComplex.maxsimpD-simplex[OF morph]
            SimplicialComplex.maxsimpD-maximal[OF morph]
    by simp
    thus f}\mp@subsup{f}{}{\prime}x\cap\mp@subsup{f}{}{\prime}y\subseteq\mp@subsup{f}{}{\prime}x\mathrm{ by fast
```



```
    using finite-maxsimp card-mono[of f}\mp@subsup{f}{}{\prime}x-\mp@subsup{f}{}{\prime}(x\capy)\mp@subsup{f}{}{`}x-\mp@subsup{f}{}{`}x\cap\mp@subsup{f}{}{\prime}y]\mathrm{ by fast
    moreover from assms(1,3,4) have card ( }\mp@subsup{f}{}{\prime}x-\mp@subsup{f}{}{\prime}(x\capy))=
    using maxsimpD-simplex faces[of x] maxsimpD-simplex
                iso-codim-map adjacent-int-facet1 [of x y] facetrel-card
    by fastforce
    ultimately have card (f`x - f'x\cap \capf`})\leq1\mathrm{ by simp
    moreover from assms(1,2,4) have card (f`x - f`}\\cap\mp@subsup{f}{}{`}\)\not=
    using 1 maxsimpD-simplex finite-maxsimp
        inj-onD[OF induced-pow-fun-inj-on, OF inj, of x y]
    by auto
    ultimately show card (f'x - f'x \cap f'y) =1 by simp
qed
lemma iso-adj-map:
    assumes maxsimp x maxsimp y }x~y x\not=
    shows f}\mp@subsup{f}{}{\prime}x~\mp@subsup{f}{}{\prime}
    using assms(3,4) iso-adj-int-im[OF assms] adjacent-sym
        iso-adj-int-im[OF assms(2) assms(1)]
    by (auto simp add: Int-commute intro: adjacentI)
lemma pmaxsimpchain-map:
    pmaxsimpchain xs \Longrightarrow SimplicialComplex.pmaxsimpchain }(f\vdashX)(f\modelsxs
proof (induct xs rule: list-induct-CCons)
    case Nil show ?case
        using map-is-simplicial-morph SimplicialComplex.pmaxsimpchain-def
        by fastforce
next
    case (Single x) thus ?case
    using map-is-simplicial-morph pmaxsimpchainD-maxsimp maxsimp-map
        SimplicialComplex.pmaxsimpchain-def
    by fastforce
```

```
next
    case (CCons x y xs)
    have SimplicialComplex.pmaxsimpchain (f\vdashX)( f`x # f`}y#f\modelsxs
    proof (
        rule SimplicialComplex.pmaxsimpchain-CConsI,
        rule map-is-simplicial-morph
    )
        from CCons(2) show SimplicialComplex.maxsimp (f\vdashX) (f`x)
            using pmaxsimpchainD-maxsimp maxsimp-map by simp
        from CCons show SimplicialComplex.pmaxsimpchain (f\vdashX) (ffy #f ==xs)
            using pmaxsimpchain-Cons-reduce by simp
        from CCons(2) show f'x ~ f'y
            using pmaxsimpchain-def iso-adj-map by simp
        from inj CCons(2) have distinct ( }f=(x#y#xs)
            using maxsimpD-simplex inj-on-distinct-setlistmapim
            unfolding pmaxsimpchain-def
            by blast
        thus f'x }\not\in\operatorname{set}(\mp@subsup{f}{}{\prime}y#f=xs) by sim
    qed
    thus ?case by simp
qed
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 $P \equiv(\bigcup x \in P .\{\{y$. pseudominimal-in $(P . \leq x) y\}\})$
lemma poset-is-SimplicialComplex:
assumes $\forall x \in P$. simplex-like $(P . \leq x)$
shows SimplicialComplex (PosetComplex P)
proof (rule SimplicialComplex.intro, rule ballI)
fix $a$ assume $a \in$ PosetComplex $P$
from this obtain $x$ where $x \in P a=\{y$. pseudominimal-in $(P . \leq x) y\}$
unfolding PosetComplex-def by fast
with assms show finite a
using pseudominimal-inD1 simplex-likeD-finite finite-subset[of a $P . \leq x]$ by fast
next
fix $a b$ assume $a b: a \in \operatorname{PosetComplex~} P b \subseteq a$
from $a b(1)$ obtain $x$ where $x: x \in P a=\{y$.pseudominimal-in $(P . \leq x) y\}$
unfolding PosetComplex-def by fast
from assms $x(1)$ obtain $f$ and $A::$ nat set
where fA: OrderingSetIso less-eq less $(\subseteq)(\subset)(P . \leq x) f$

$$
f^{\prime}(P . \leq x)=\text { Pow } A
$$

using simplex-likeD-iso[of P. $\leq x]$
by auto
define $x^{\prime}$ where $x^{\prime}: x^{\prime} \equiv$ the-inv-into $(P . \leq x) f\left(\bigcup\left(f^{\prime} b\right)\right)$
from $f A x$ (2) $a b(2) x^{\prime}$ have $x^{\prime}-P: x^{\prime} \in P$
using collect-pseudominimals-below-in-poset $[$ of $P x f]$ by simp
moreover from $x f A a b(2) x^{\prime}$ have $b=\left\{y\right.$. pseudominimal-in $\left.\left(P . \leq x^{\prime}\right) y\right\}$
using collect-pseudominimals-below-in-eq[of x P f] by simp
ultimately show $b \in$ PosetComplex $P$ unfolding PosetComplex-def by fast qed
definition poset-simplex-map :: 'a set $\Rightarrow{ }^{\prime} a \Rightarrow$ 'a set
where poset-simplex-map $P x=\{y$.pseudominimal-in $(P . \leq x) y\}$
lemma poset-to-PosetComplex-OrderingSetMap:
assumes $\bigwedge x . x \in P \Longrightarrow$ simplex-like $(P . \leq x)$
shows OrderingSetMap $(\leq)(<)(\subseteq)(\subset) P($ poset-simplex-map $P)$

## proof

from assms
show $\bigwedge a b . \llbracket a \in P ; b \in P ; a \leq b \rrbracket \Longrightarrow$
poset-simplex-map $P a \subseteq$ poset-simplex-map $P b$
using simplex-like-has-bottom pseudominimal-in-below-in
unfolding poset-simplex-map-def
by fast
qed
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 :: ' $a \Rightarrow^{\prime} a \Rightarrow b o o l($ infix $\leq 50)$
and less $\quad::{ }^{\prime} a \Rightarrow^{\prime} a \Rightarrow b o o l($ infix $<50)$

+ fixes $P$ :: 'a set
assumes below-in-P-simplex-like: $x \in P \Longrightarrow$ simplex-like $(P . \leq x)$
and $\quad P$-has-bottom : has-bottom $P$
and $\quad$-has-glbs $\quad: x \in P \Longrightarrow y \in P \Longrightarrow \exists b$. glbound-in-of $P$ x y b
begin
abbreviation smap $\equiv$ poset-simplex-map $P$
lemma smap-onto-PosetComplex: smap' $P=$ PosetComplex $P$
using poset-simplex-map-def PosetComplex-def by auto
lemma ordsetmap-smap: $\llbracket a \in P ; b \in P ; a \leq b \rrbracket \Longrightarrow$ smap $a \subseteq$ smap $b$

```
    using OrderingSetMap.ordsetmap[
    OF poset-to-PosetComplex-OrderingSetMap, OF below-in-P-simplex-like
        ]
    poset-simplex-map-def
by
    simp
lemma inj-on-smap: inj-on smap P
proof (rule inj-onI)
    fix }xy\mathrm{ assume xy: xGP y, Pmap x = smap y
    show }x=
    proof (cases smap x = {})
    case True with xy show ?thesis
        using poset-simplex-map-def below-in-P-simplex-like P-has-bottom
                simplex-like-no-pseudominimal-in-below-in-imp-singleton[of x P]
                simplex-like-no-pseudominimal-in-below-in-imp-singleton[of y P]
                below-in-singleton-is-bottom[of P x] below-in-singleton-is-bottom[of P y]
        by auto
    next
    case False
    from this obtain z}\mathrm{ where z smap x by fast
    with xy(3) have z1:z\inP.\leqxz\inP.\leqy
        using pseudominimal-inD1 poset-simplex-map-def by auto
    hence lbound-of x y z by (auto intro:lbound-ofI)
    with z1(1) obtain b where b: glbound-in-of P x y b
            using xy(1,2) P-has-glbs by fast
    moreover have b\inP.\leqxb b\inP.\leqy
        using glbound-in-ofD-in[OF b] glbound-in-of-less-eq1[OF b]
                glbound-in-of-less-eq2[OF b]
            by auto
    ultimately show ?thesis
            using xy below-in-P-simplex-like
                    pseudominimal-in-below-in-less-eq-glbound[of P x - y b]
                    simplex-like-below-in-above-pseudominimal-is-top[of x P]
                    simplex-like-below-in-above-pseudominimal-is-top[of y P]
        unfolding poset-simplex-map-def
        by force
    qed
qed
lemma OrderingSetIso-smap:
    OrderingSetIso (\leq) (<) (\subseteq)(\subset)P smap
proof (rule OrderingSetMap.isoI)
    show OrderingSetMap (\leq)(<) (\subseteq) (\subset) P smap
        using poset-simplex-map-def below-in-P-simplex-like
            poset-to-PosetComplex-OrderingSetMap
        by simp
next
    fix }xy\mathrm{ assume xy: xGP y, smap x}\subseteq\mathrm{ smap y
    from xy(2) have simplex-like (P.\leqy) using below-in-P-simplex-like by fast
```

```
    from this obtain g}\mathrm{ and A::nat set
        where OrderingSetIso (\leq) (<) (\subseteq) (\subset) (P.\leqy)g
            g}(P.\leqy)= Pow A
    using simplex-likeD-iso[of P.\leqy]
    by auto
with xy show }x\leq
    using poset-simplex-map-def collect-pseudominimals-below-in-eq[of y P g]
        collect-pseudominimals-below-in-poset[of P y g]
        inj-onD[OF inj-on-smap, of the-inv-into (P.\leqy)g(U(g'smap x)) x]
        collect-pseudominimals-below-in-less-eq-top[of P y g A smap x]
    by simp
qed (rule inj-on-smap)
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 : $y \in X \Longrightarrow \exists x$. maxsimp $x \wedge y \subseteq x$
and maxsimp-connect: $\llbracket x \neq y ;$ maxsimp $x ; \operatorname{maxsimp} y \rrbracket \Longrightarrow$ $\exists x s$. maxsimpchain $(x \# x s @[y])$
context ChamberComplex
begin
abbreviation chamber $\equiv$ maxsimp

```
abbreviation gallery \(\equiv\) maxsimpchain
abbreviation pgallery \(\equiv\) pmaxsimpchain
abbreviation min-gallery \(\equiv\) min-maxsimpchain
abbreviation supchamber \(v \equiv(S O M E C\). chamber \(C \wedge v \in C)\)
```

lemmas faces
lemmas singleton-simplex
lemmas chamberI
lemmas chamberD-simplex
lemmas chamberD-maximal

$$
=\text { maxsimpD-maximal }
$$

lemmas finite-chamber
lemmas chamber-vertices
lemmas gallery-def
lemmas gallery-append-reduce1 $=$ maxsimpchain-append-reduce1

$$
=\text { faces }
$$

$=$ singleton-simplex

$$
=\text { maxsimpI }
$$

$$
=\text { maxsimp } D \text {-simplex }
$$

$$
=\text { finite-maxsimp }
$$

lemmas chamber-nempty
lemmas gallery-snocI
lemmas galleryD-chamber
lemmas galleryD-adj
lemmas gallery-CConsI
lemmas gallery-Cons-reduce
lemmas gallery-append-reduce2 $=$ maxsimpchain-append-reduce2
lemmas gallery-remdup-adj $\quad=$ maxsimpchain-remdup-adj
lemmas gallery-obtain-pgallery $=$ maxsimpchain-obtain-pmaxsimpchain
lemmas pgallery-def $\quad=$ pmaxsimpchain-def
lemmas pgalleryI-gallery
lemmas pgalleryD-chamber
lemmas pgalleryD-adj
lemmas pgalleryD-distinct
lemmas pgallery-Cons-reduce
$=$ maxsimp-nempty
$=$ maxsimp-vertices
$=$ maxsimpchain-def
= maxsimpchain-snocI
$=$ maxsimpchainD-maxsimp
$=$ maxsimpchainD-adj
$=$ maxsimpchain-CConsI
$=$ maxsimpchain-Cons-reduce
$=$ pmaxsimpchainI-maxsimpchain
= pmaxsimpchainD-maxsimp
$=$ pmaxsimpchainD-adj
$=$ pmaxsimpchainD-distinct
$=$ pmaxsimpchain-Cons-reduce
lemmas pgallery-append-reduce1 = pmaxsimpchain-append-reduce1
lemmas pgallery $\quad=$ pmaxsimpchain
lemmas min-gallery-simps $\quad=$ min-maxsimpchain.simps
lemmas min-galleryI-betw $\quad=$ 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 $\quad=$ min-maxsimpchain-rev
lemmas min-gallery-adj $\quad=$ 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:
$\llbracket x \neq y ;$ chamber $x ;$ chamber $y \rrbracket \Longrightarrow \exists x s$. pgallery $(x \# x s @[y])$
using maxsimp-connect [of $x y$ ] gallery-obtain-pgallery[of $x y]$ by fast
lemma supchamberD:
assumes $v \in \bigcup X$
defines $C \equiv$ supchamber $v$
shows chamber $C v \in C$
using assms simplex-in-max someI[of $\lambda C$. chamber $C \wedge v \in C]$
by auto

## definition

ChamberSubcomplex $Y \equiv Y \subseteq X \wedge$ ChamberComplex $Y \wedge$
$(\forall C$. ChamberComplex.chamber $Y C \longrightarrow$ chamber $C)$
lemma ChamberSubcomplexI: assumes $Y \subseteq X$ ChamberComplex $Y$
$\bigwedge y$. ChamberComplex.chamber $Y y \Longrightarrow$ chamber $y$
shows ChamberSubcomplex $Y$
using assms ChamberSubcomplex-def
by fast
lemma ChamberSubcomplexD-sub: ChamberSubcomplex $Y \Longrightarrow Y \subseteq X$ using ChamberSubcomplex-def by fast
lemma ChamberSubcomplexD-complex: ChamberSubcomplex $Y \Longrightarrow$ ChamberComplex $Y$ unfolding ChamberSubcomplex-def by fast
lemma chambersub-imp-sub: ChamberSubcomplex $Y \Longrightarrow$ Subcomplex $Y$
using ChamberSubcomplex-def ChamberComplex.axioms(1) by fast
lemma chamber-in-subcomplex:
【 ChamberSubcomplex $Y ; C \in Y$; chamber $C \rrbracket \Longrightarrow$ ChamberComplex.chamber Y C
using chambersub-imp-sub max-in-subcomplex by simp
lemma subcomplex-chamber:
ChamberSubcomplex $Y \Longrightarrow$ ChamberComplex.chamber Y $C \Longrightarrow$ chamber $C$ unfolding ChamberSubcomplex-def by fast
lemma gallery-in-subcomplex:
$\llbracket$ ChamberSubcomplex $Y$; set ys $\subseteq Y$; gallery ys $\rrbracket \Longrightarrow$ ChamberComplex.gallery Y ys
using chambersub-imp-sub maxsimpchain-in-subcomplex by simp
lemma subcomplex-gallery:
ChamberSubcomplex $Y \Longrightarrow$ ChamberComplex.gallery $Y$ Cs $\Longrightarrow$ gallery Cs using ChamberSubcomplex-def gallery-def ChamberComplex.gallery-def
by fastforce

```
lemma subcomplex-pgallery
    ChamberSubcomplex Y ChamberComplex.pgallery Y Cs \Longrightarrow pgallery Cs
    using ChamberSubcomplex-def pgallery-def ChamberComplex.pgallery-def
    by fastforce
lemma min-gallery-in-subcomplex:
    assumes ChamberSubcomplex Y min-gallery Cs set Cs }\subseteq
    shows ChamberComplex.min-gallery Y Cs
proof (cases Cs rule: list-cases-Cons-snoc)
    case Nil with assms(1) show ?thesis
        using ChamberSubcomplexD-complex ChamberComplex.min-gallery-simps(1)
    by fast
next
    case Single with assms show ?thesis
        using min-galleryD-gallery galleryD-chamber chamber-in-subcomplex
                ChamberComplex.min-gallery-simps(2) ChamberSubcomplexD-complex
    by force
next
    case (Cons-snoc C Ds D)
    with assms show ?thesis
        using ChamberSubcomplexD-complex min-gallery-pgallery
                pgalleryD-distinct[of C#Ds@[D]] pgallery
                gallery-in-subcomplex[of Y] subcomplex-gallery
                min-galleryD-min-betw
                ChamberComplex.min-galleryI-betw[of Y]
    by force
qed
lemma chamber-card: chamber C chamber D card C = card D
    using maxsimp-connect[of C D] galleryD-adj adjacentchain-card
    by (cases C=D) auto
lemma chamber-facet-is-chamber-facet:
    \llbracketchamber C; chamber D;z\triangleleftC;z\subseteqD\rrbracket\Longrightarrowz\triangleleftD
    using finite-chamber finite-facetrel-card chamber-card[of C]
    by (fastforce intro: facetrelI-cardSuc)
lemma chamber-adj:
    assumes chamber C D\inX C~D
    shows chamber D
proof-
    from assms(2) obtain B where B: chamber B D\subseteqB
        using simplex-in-max by fast
    with assms(1,3) show ?thesis
        using chamber-card[of B] adjacent-card finite-chamber card-subset-eq[of B D]
        by force
qed
lemma chambers-share-facet:
```

```
    assumes chamber C chamber (insert vz) z\triangleleftC
    shows z\triangleleftinsert vz
proof (rule facetrelI)
    from assms show v\not\inz
    using finite-chamber[of C] finite-chamber[of insert v z] card-insert-if[of z v]
    by (auto simp add: finite-facetrel-card chamber-card)
qed simp
lemma adjacentset-chamber: chamber C \Longrightarrow D\inadjacentset C chamber D
    using adjacentset-def chamber-adj by fast
lemma chamber-shared-facet: \llbracket chamber C;z\triangleleftC;D\inX;z\triangleleftD\rrbracket\Longrightarrow chamber D
    by (fast intro: chamber-adj adjacentI)
lemma adjacentset-conv-facetchambersets:
    assumes X\not={{}} chamber C
    shows adjacentset C = (\bigcupv\inC.{D\inX.C-{v}\triangleleftD})
proof (rule seteqI)
    fix D assume D:D\inadjacentset C
    show }D\in(\bigcupv\inC.{D\inX.C-{v}\triangleleftD}
    proof (cases D=C)
        case True with assms
        have C}\not={}\mathrm{ and }C\in
            using chamber-nempty chamberD-simplex by auto
        with True assms show ?thesis
                using facetrel-diff-vertex by fastforce
    next
        case False
        from D have D':C~D using adjacentsetD-adj by fast
        with False obtain v}\mathrm{ where v: vॄDC= insert v (CคD)
            using adjacent-int-decomp by fast
        hence C-{v}=C\capD by auto
        with D' False have C-{v}\triangleleftD using adjacent-int-facet2 by auto
        with assms(2) D v(2) show ?thesis using adjacentset-def by fast
    qed
next
    from assms(2)
        show }\D.D\in(\bigcupv\inC.{E\inX.C-{v}\triangleleftE})
                D\in adjacentset C
    using facetrel-diff-vertex adjacentI
    unfolding adjacentset-def
    by fastforce
qed
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 chamber-system }\equiv{C\mathrm{ . chamber C}
abbreviation }\mathcal{C}\equiv\mathrm{ chamber-system
definition chamber-distance :: 'a set }=>\mathrm{ ''a set }=>\mathrm{ nat
    where 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 }=>\mathrm{ ' 'a set }=>\mp@subsup{|}{}{\prime}a\mathrm{ set
    where closest-supchamber F D=
            (ARG-MIN ( }\lambdaC\mathrm{ . chamber-distance C D) C.
                        chamber C ^ F\subseteqC)
definition face-distance F D \equiv chamber-distance (closest-supchamber F D) D
lemma chamber-system-simplices: }\mathcal{C}\subseteq
    using chamberD-simplex unfolding chamber-system-def by fast
lemma gallery-chamber-system: gallery Cs \Longrightarrow set Cs \subseteq\mathcal{C}
    using galleryD-chamber chamber-system-def by fast
lemmas pgallery-chamber-system = gallery-chamber-system[OF pgallery]
lemma chamber-distance-le:
    gallery (C#Cs@[D]) \Longrightarrow chamber-distance C D \leq Suc (length Cs)
    using chamber-distance-def
            arg-min-nat-le[of \lambdaCs.gallery (C#Cs@[D])-length]
    by auto
lemma min-gallery-betw-chamber-distance:
    min-gallery (C#Cs@[D])\Longrightarrow chamber-distance C D = Suc (length Cs)
    using chamber-distance-def[of C D] is-arg-min-size[of length - Cs] by auto
lemma min-galleryI-chamber-distance-betw:
    gallery (C#Cs@[D])\Longrightarrow Suc (length Cs) = chamber-distance C D \Longrightarrow
    min-gallery (C#Cs@[D])
    using chamber-distance-def chamber-distance-le min-galleryI-betw[of C D]
    by fastforce
lemma gallery-least-length:
    assumes chamber C chamber D C\not=D
```

```
defines Cs \equivARG-MIN length Cs.gallery (C#Cs@[D])
shows gallery (C#Cs@[D])
using assms maxsimp-connect[of C D] arg-min-natI
by fast
lemma min-gallery-least-length:
assumes chamber C chamber D C\not=D
defines Cs \equivARG-MIN length Cs. gallery (C#Cs@[D])
shows min-gallery(C#Cs@[D])
unfolding Cs-def
using assms gallery-least-length
by (blast intro: min-galleryI-betw arg-min-nat-le)
lemma pgallery-least-length:
assumes chamber C chamber D C\not=D
defines Cs \equivARG-MIN length Cs.gallery (C#Cs@[D])
shows pgallery (C#Cs@[D])
using assms min-gallery-least-length min-gallery-pgallery
by
fast
lemma closest-supchamberD:
assumes F\inX chamber D
shows chamber (closest-supchamber F D) F\subseteqclosest-supchamber F D
using assms arg-min-natI[of \lambdaC. chamber C}\wedgeF\subseteqC] simplex-in-max[of F
unfolding closest-supchamber-def
by auto
lemma closest-supchamber-closest:
    chamber C\LongrightarrowF\subseteqC\Longrightarrow
    chamber-distance (closest-supchamber F D) D\leqchamber-distance C D
using arg-min-nat-le[of \lambdaC. chamber C ^F\subseteqC C] closest-supchamber-def
by simp
lemma face-distance-le:
    chamber C\LongrightarrowF\subseteqC\Longrightarrow face-distance F D s chamber-distance C D
    unfolding face-distance-def closest-supchamber-def
    by (auto intro: arg-min-nat-le)
lemma face-distance-eq-0: chamber C \LongrightarrowF\subseteqC\Longrightarrow face-distance F C=0
    using chamber-distance-def closest-supchamber-def face-distance-def
        arg-min-equality[
            of \lambdaC. chamber C ^F\subseteqCC \lambdaD. chamber-distance D C
        ]
    by simp
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 => (' }a=>\mathrm{ ' b) # bool
    where label-wrt B f}\equiv(\forallC\in\mathcal{C}.bij-betw f C B
lemma label-wrtD: label-wrt B f C C\in\mathcal{C}\Longrightarrowbij-betw f C B
    using label-wrt-def by fast
lemma label-wrtD': label-wrt B f \Longrightarrow chamber C \Longrightarrowbij-betw f C B
    using label-wrt-def chamber-system-def by fast
lemma label-wrt-adjacent:
    assumes label-wrt B f chamber C chamber D C~D v\inC-D w\inD-C
    shows fv=fw
proof-
    from assms(5) have f}\mp@subsup{f}{}{\prime}D=\operatorname{insert (f v) (f`}(C\capD)
        using adjacent-conv-insert[OF assms(4), of v] label-wrtD'[OF assms(1,2)]
            label-wrtD'[OF assms(1,3)]
            bij-betw-imp-surj-on[of f]
        by force
    with assms(6) show ?thesis
        using adjacent-sym[OF assms(4)] adjacent-conv-insert[of D C]
            inj-on-insert[ of f w C\capD]
            bij-betw-imp-inj-on[OF label-wrtD', OF assms(1,3)]
        by (force simp add: Int-commute)
qed
lemma label-wrt-adjacent-shared-facet:
    \llbracketlabel-wrt B f; chamber (insert v z); chamber (insert wz);v\not\inz;w\not\inz\rrbracket\Longrightarrow
        fv=fw
    by (auto intro: label-wrt-adjacent adjacentI facetrelI)
lemma label-wrt-elt-image: label-wrt Bf\Longrightarrowv\in\X\Longrightarrowfv\inB
    using simplex-in-max label-wrtD' bij-betw-imp-surj-on by fast
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 $\quad X::$ 'a set set
and $\quad Y::{ }^{\prime} b$ set set

+ fixes $f::^{\prime} a \Rightarrow{ }^{\prime} b$
assumes chamber-map: domain.chamber $C \Longrightarrow$ codomain.chamber $\left(f^{\prime} C\right)$
and dim-map : domain.chamber $C \Longrightarrow \operatorname{card}\left(f^{\bullet} C\right)=\operatorname{card} C$
lemma (in ChamberComplex) trivial-morphism:
ChamberComplexMorphism X X id
by unfold-locales auto
lemma (in ChamberComplex) inclusion-morphism:
assumes ChamberSubcomplex Y
shows ChamberComplexMorphism Y X id
by (
rule ChamberComplexMorphism.intro, rule ChamberSubcomplexD-complex, rule assms, unfold-locales
)
(auto simp add: subcomplex-chamber[OF assms])
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 gf $(\bigcup X) \Longrightarrow$ ChamberComplexMorphism X Yg using chamber-map domain.chamber-vertices fun-eq-on-im $[$ of $g f]$ dim-map domain.chamber-vertices
by unfold-locales auto
lemma comp:
assumes ChamberComplexMorphism YZg
shows ChamberComplexMorphism X Z (gof)
proof
rule ChamberComplexMorphism.intro, rule domain-complex,
rule ChamberComplexMorphism.axioms(2), rule assms, unfold-locales
)
fix $C$ assume $C$ : domain.chamber $C$
from $C$ show SimplicialComplex.maxsimp $Z\left((g \circ f){ }^{‘} C\right)$
using chamber-map ChamberComplexMorphism.chamber-map[OF assms]

```
    by (force simp add: image-comp[THEN sym])
    from C show card ((g\circf)'C) = card C
    using chamber-map dim-map ChamberComplexMorphism.dim-map[OF assms]
    by (force simp add: image-comp[THEN sym])
qed
lemma restrict-domain:
    assumes domain.ChamberSubcomplex W
    shows ChamberComplexMorphism W Yf
proof (
    rule ChamberComplexMorphism.intro, rule domain.ChamberSubcomplexD-complex,
    rule assms, rule codomain-complex, unfold-locales
)
    fix C assume ChamberComplex.chamber W C
    with assms show codomain.chamber ( f}\mp@subsup{f}{}{`}C)\mathrm{ card ( }\mp@subsup{f}{}{`}C)=\mathrm{ card C
        using domain.subcomplex-chamber chamber-map dim-map by auto
qed
lemma restrict-codomain:
    assumes codomain.ChamberSubcomplex Z f\vdashX\subseteqZ
    shows ChamberComplexMorphism X Z f
proof (
    rule ChamberComplexMorphism.intro, rule domain-complex,
    rule codomain.ChamberSubcomplexD-complex,
    rule assms, unfold-locales
)
    fix C assume domain.chamber C
    with assms show SimplicialComplex.maxsimp Z (f`C) card (f ' C) = card C
        using domain.chamberD-simplex[of C] chamber-map
            codomain.chamber-in-subcomplex dim-map
    by auto
qed
lemma inj-on-chamber:domain.chamber C \Longrightarrow inj-on f C
    using domain.finite-chamber dim-map by (fast intro: eq-card-imp-inj-on)
lemma bij-betw-chambers: domain.chamber C \Longrightarrow bij-betw f C (f`C)
    using inj-on-chamber by (fast intro: bij-betw-imageI)
lemma card-map: x\inX \Longrightarrow card (ffx) = card x
    using domain.simplex-in-max subset-inj-on[OF inj-on-chamber]
        domain.finite-simplex inj-on-iff-eq-card
    by blast
lemma codim-map:
    assumes domain.chamber C y\subseteqC
    shows card (f`C - f'y) = card (C-y)
    using assms dim-map domain.chamberD-simplex domain.faces[of C y]
        domain.finite-simplex card-Diff-subset[of f'y f'C]
```

```
        card-map card-Diff-subset[of y C]
    by
    auto
lemma simplex-map: }x\inX\Longrightarrow\mp@subsup{f}{}{\prime}x\in
    using chamber-map domain.simplex-in-max codomain.chamberD-simplex
        codomain.faces[of - f'x]
    by force
lemma simplices-map: f\vdashX\subseteqY
    using simplex-map by fast
lemma vertex-map: }x\in\bigcupX\Longrightarrowfx\in\bigcup
    using simplex-map by fast
lemma facet-map: domain.chamber C \Longrightarrow z\triangleleftC\Longrightarrow \Longrightarrow f}z\triangleleft\mp@subsup{f}{}{`}
    using facetrel-subset facetrel-card codim-map[of C z]
    by (fastforce intro: facetrelI-card)
lemma adj-int-im:
    assumes domain.chamber C domain.chamber D C~D f}
    shows (f`C\cap f}D)\triangleleft\mp@subsup{f}{}{\prime}
proof (rule facetrelI-card)
    from assms(1,2) chamber-map have 1: f}C\subseteq\mp@subsup{f}{}{\prime}D\Longrightarrow\mp@subsup{f}{}{`}C=\mp@subsup{f}{}{\prime}
        using codomain.chamberD-simplex codomain.chamberD-maximal[of f}\mp@subsup{f}{}{`}C\mp@subsup{f}{}{`}D
    by simp
    thus f'C\capf'D\subseteqf'C by fast
    from assms(1) have card (f`C - f`C \cap f`D) \leq card (f`C - f
        using domain.finite-chamber
                card-mono[of f}\mp@subsup{f}{}{`}C-\mp@subsup{f}{}{\iota}(C\capD) f`C-\mp@subsup{f}{}{`}C\cap\mp@subsup{f}{}{`}D
    by fast
    moreover from assms(1,3,4) have card (f`}\\mathrm{ ( }C-\mp@subsup{f}{}{\prime}(C\capD))=
    using codim-map[of C C\capD] adjacent-int-facet1 facetrel-card
    by fastforce
    ultimately have card (f`C - f``}C\cap\mp@subsup{f}{}{`}D)\leq1 by sim
    moreover from 1 assms(1,4) have card ( }\mp@subsup{f}{}{`}C-\mp@subsup{f}{}{`}C\cap\mp@subsup{f}{}{`}D)\not=
    using domain.finite-chamber by auto
    ultimately show card (f}\mp@subsup{f}{}{`}C-\mp@subsup{f}{}{`}C\cap\mp@subsup{f}{}{`}D)=1 by sim
qed
lemma adj-map':
    assumes domain.chamber C domain.chamber D C~D f
    shows f`
    using assms(3,4) adj-int-im[OF assms] adjacent-sym
        adj-int-im[OF assms(2) assms(1)]
    by (auto simp add: Int-commute intro: adjacentI)
lemma adj-map:
    \llbracketdomain.chamber C; domain.chamber D; C ~ D\rrbracket\Longrightarrow f`C~ f}
```

using adjacent-refl[of $\left.f^{‘} C\right]$ adj-map' empty-not-adjacent $[$ of $D]$ by fastforce

```
lemma chamber-vertex-outside-facet-image:
    assumes v\not\inz domain.chamber (insert vz)
    shows fv}\not\in\mp@subsup{f}{}{\prime}
proof-
    from assms(1) have insert vz-z={v} by force
    with assms(2) show ?thesis using codim-map by fastforce
qed
lemma expand-codomain:
    assumes ChamberComplex Z ChamberComplex.ChamberSubcomplex Z Y
    shows ChamberComplexMorphism X Z f
proof (
    rule ChamberComplexMorphism.intro, rule domain-complex, rule assms(1),
    unfold-locales
)
    from assms show
    \ x . d o m a i n . c h a m b e r ~ x ~ C ~ S i m p l i c i a l C o m p l e x . m a x s i m p ~ Z ~ ( f ' x )
    using chamber-map ChamberComplex.subcomplex-chamber by fast
qed (auto simp add: dim-map)
end
```


### 4.4.2 Action on pregalleries and galleries

## context ChamberComplexMorphism

begin

```
lemma gallery-map: domain.gallery Cs \Longrightarrow codomain.gallery ( }f=Cs\mathrm{ )
proof (induct Cs rule: list-induct-CCons)
    case (Single C) thus ?case
        using domain.galleryD-chamber chamber-map codomain.gallery-def by auto
next
    case (CCons B C Cs)
    have codomain.gallery (f`B # f}C|#|=Cs
    proof (rule codomain.gallery-CConsI)
        from CCons(2) show codomain.chamber ( f` B)
            using domain.galleryD-chamber chamber-map by simp
        from CCons show codomain.gallery ( }\mp@subsup{f}{}{`}C#f=Cs
            using domain.gallery-Cons-reduce by auto
        from CCons(2) show f}\mp@subsup{f}{}{\prime}B~\mp@subsup{f}{}{\prime}
            using domain.gallery-Cons-reduce[of B C#Cs] domain.galleryD-adj
                domain.galleryD-chamber adj-map
            by fastforce
    qed
    thus ?case by simp
qed (simp add: codomain.maxsimpchain-def)
```


## lemma gallery－betw－map

domain．gallery $(C \# C s @[D]) \Longrightarrow$ codomain．gallery $\left(f^{`} C \# f=C s @\left[f^{‘} D\right]\right)$
using gallery－map by fastforce
end

## 4．4．3 Properties of the image

context ChamberComplexMorphism
begin
lemma subcomplex－image：codomain．Subcomplex $(f \vdash X)$
using simplicialcomplex－image simplex－map by fast
lemmas chamber－in－image $=$ codomain．max－in－subcomplex［OF subcomplex－image］
lemma maxsimp－map－into－image：
assumes domain．chamber $x$
shows SimplicialComplex．maxsimp $(f \vdash X)\left(f^{\prime} x\right)$
proof（
rule SimplicialComplex．maxsimpI，rule simplicialcomplex－image，rule imageI， rule domain．chamberD－simplex，rule assms
）
from assms show $\bigwedge z . z \in f \vdash X \Longrightarrow f^{\prime} x \subseteq z \Longrightarrow z=f^{\prime} x$
using chamber－map［of $x$ ］simplex－map codomain．chamberD－maximal［ of f＇x］
by blast
qed
lemma maxsimp－preimage：
assumes C $C \in$ SimplicialComplex．maxsimp $(f \vdash X)\left(f^{〔} C\right)$
shows domain．chamber $C$
proof－
from $\operatorname{assms}(1)$ obtain $D$ where $D$ ：domain．chamber $D C \subseteq D$
using domain．simplex－in－max by fast
have $C=D$
proof（rule card－subset－eq）
from $D(1)$ show finite $D$ using domain．finite－chamber by fast
with assms $D$ show card $C=\operatorname{card} D$
using domain．chamberD－simplex simplicialcomplex－image SimplicialComplex．maxsimpD－maximal［of $\left.f \vdash X f^{〔} C f^{〔} D\right]$
card－mono［of D C］domain．finite－simplex card－image－le［of C f］dim－map
by force
qed（rule $D(2)$ ）
with $D(1)$ show ？thesis by fast
qed
lemma chamber－preimage：
$C \in X \Longrightarrow$ codomain．chamber $\left(f^{‘} C\right) \Longrightarrow$ domain．chamber $C$
using chamber－in－image maxsimp－preimage by simp

```
lemma chambercomplex-image: ChamberComplex ( }f\vdash-X
proof (intro-locales, rule simplicialcomplex-image, unfold-locales)
    show }\y.y\inf\vdashX\Longrightarrow\existsx\mathrm{ . SimplicialComplex.maxsimp (f৮X) x ^ y }\subseteq
    using domain.simplex-in-max maxsimp-map-into-image by fast
next
    fix }x
    assume xy: x\not=y SimplicialComplex.maxsimp (f\vdash-X)x
                SimplicialComplex.maxsimp (f\vdashX) y
    from xy(2,3) obtain zx zy where zxy:zx\inX x = f`zx zy\inX y = f
    using SimplicialComplex.maxsimpD-simplex[OF simplicialcomplex-image, of x]
                SimplicialComplex.maxsimpD-simplex[OF simplicialcomplex-image, of y]
    by fast
    with xy obtain ws where ws: domain.gallery (zx#ws@[zy])
    using maxsimp-preimage domain.maxsimp-connect[of zx zy] by auto
    with ws zxy(2,4) have SimplicialComplex.maxsimpchain (f\vdashX) (x# (f\modelsws)@[y])
    using gallery-map[of zx#ws@[zy]] domain.galleryD-chamber
                domain.chamberD-simplex codomain.galleryD-chamber
                codomain.max-in-subcomplex[OF subcomplex-image]
                codomain.galleryD-adj
                SimplicialComplex.maxsimpchain-def[OF simplicialcomplex-image]
    by
                auto
    thus \exists xs. SimplicialComplex.maxsimpchain (f\vdashX)(x#xs@[y]) by fast
qed
lemma chambersubcomplex-image: codomain.ChamberSubcomplex ( }f\vdashX
    using simplices-map chambercomplex-image ChamberComplex.chamberD-simplex
    chambercomplex-image maxsimp-preimage chamber-map
    by (force intro: codomain.ChamberSubcomplexI)
lemma restrict-codomain-to-image: ChamberComplexMorphism X (f\vdashX)f
    using restrict-codomain chambersubcomplex-image by fast
end
```


### 4.4.4 Action on the chamber system

```
context ChamberComplexMorphism
```

context ChamberComplexMorphism
begin
begin
lemma chamber-system-into: f\vdash domain.C }\subseteq\mathrm{ codomain.C
using chamber-map domain.chamber-system-def codomain.chamber-system-def
by auto
lemma chamber-system-image: f\vdashdomain.C = codomain.C }\cap(f\vdashX
proof
show f\vdashdomain.C \subseteq codomain.C \cap (f\vdashX)
using chamber-system-into domain.chamber-system-simplices by fast
show f\vdashdomain.\mathcal{C}\supseteq codomain.\mathcal{C}\cap(f\vdashX)

```
```

    proof
    fix D assume D\in codomain.C \cap (f\vdashX)
    hence \existsC. domain.chamber C^ f}\\C=
        using codomain.chamber-system-def chamber-preimage by fast
    thus D\inf\vdashdomain.C using domain.chamber-system-def by auto
    qed
    qed
lemma image-chamber-system:ChamberComplex.C (f\vdashX)=f\vdash domain.C
using ChamberComplex.chamber-system-def codomain.subcomplex-chamber
ChamberComplex.chamberD-simplex chambercomplex-image
chambersubcomplex-image chamber-system-image
codomain.chamber-in-subcomplex codomain.chamber-system-def
by
auto
lemma image-chamber-system-image:
ChamberComplex.C (f\vdashX) = codomain.C }\cap(f\vdashX
using image-chamber-system chamber-system-image by simp
lemma face-distance-eq-chamber-distance-map:
assumes domain.chamber C domain.chamber D C\not=D z\subseteqC
codomain.face-distance (f`z) (f`D) = domain.face-distance z D
domain.face-distance z D=domain.chamber-distance C D
shows codomain.face-distance (f`z) (f`D) =
codomain.chamber-distance (f`C) (f`D)
using assms codomain.face-distance-le[of f'C ffz f'D] chamber-map
codomain.chamber-distance-le
gallery-betw-map[OF domain.gallery-least-length, of C D]
domain.chamber-distance-def
by force
lemma face-distance-eq-chamber-distance-min-gallery-betw-map:
assumes domain.chamber C domain.chamber D C\not=D z\subseteqC
codomain.face-distance ( }\mp@subsup{f}{}{`}z)(\mp@subsup{f}{}{`}D)=\mathrm{ domain.face-distance z D
domain.face-distance z D=domain.chamber-distance C D
domain.min-gallery (C\#Cs@[D])
shows codomain.min-gallery (f\models(C\#Cs@[D]))
using assms face-distance-eq-chamber-distance-map[of C D z]
gallery-map[OF domain.min-galleryD-gallery,OF assms(7)]
domain.min-gallery-betw-chamber-distance[OF assms(7)]
codomain.min-galleryI-chamber-distance-betw[of f`C f =Cs f`D]
by auto
end

```

\subsection*{4.4.5 Isomorphisms}
locale ChamberComplexIsomorphism \(=\) ChamberComplexMorphism X Yf for \(X\) :: 'a set set
and \(Y::\) ' \(b\) set set
and \(f::{ }^{\prime} a \Rightarrow{ }^{\prime} b\)
+ assumes bij-betw-vertices: bij-betw \(f(\bigcup X)(\bigcup Y)\)
and \(\quad\) surj-simplex-map : \(f \vdash X=Y\)
lemma (in ChamberComplexIsomorphism) inj: inj-on \(f(\bigcup X)\)
using bij-betw-vertices bij-betw-def by fast
sublocale ChamberComplexIsomorphism < SimplicialComplexIsomorphism using inj by (unfold-locales) fast
lemma (in ChamberComplex) trivial-isomorphism:
ChamberComplexIsomorphism X X id
using trivial-morphism bij-betw-id
by unfold-locales (auto intro: ChamberComplexIsomorphism.intro)
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 Yf
proof (rule ChamberComplexIsomorphism.intro)
show ChamberComplexMorphism X Yf..
show ChamberComplexIsomorphism-axioms XYf
proof
from assms show bij-betw \(f(\bigcup X)(\bigcup Y)\)
using vertex-map ChamberComplexMorphism.vertex-map
comps-fixpointwise-imp-bij-betw[of \(f \bigcup X \bigcup Y g]\)
by fast
show \(f \vdash X=Y\)
proof (rule order.antisym, rule simplices-map, rule subsetI)
fix \(y\) assume \(y \in Y\)
moreover hence \((f \circ g)\) ' \(y \in f \vdash X\)
using ChamberComplexMorphism.simplex-map[OF assms(1)]
by (simp add: image-comp[THEN sym])
ultimately show \(y \in f \vdash X\)
using fixespointwise-subset[OF assms(3), of y] fixespointwise-im by fastforce qed
qed
qed
context ChamberComplexIsomorphism
begin
lemmas domain-complex = domain-complex
lemmas chamber-map \(=\) chamber-map
lemmas dim-map \(\quad=\) dim-map
lemmas gallery-map \(=\) gallery-map
lemmas simplex-map = simplex-map
lemmas chamber-preimage \(=\) chamber-preimage
```

lemma chamber-morphism:ChamberComplexMorphism X Yf ..
lemma pgallery-map: domain.pgallery Cs \Longrightarrow codomain.pgallery (f =Cs)
using pmaxsimpchain-map surj-simplex-map by simp
lemma iso-cong:
assumes fun-eq-on g f (\bigcupX)
shows ChamberComplexIsomorphism X Yg
proof (
rule ChamberComplexIsomorphism.intro, rule cong, rule assms,
unfold-locales
)
from assms show bij-betw g (\bigcupX) (\bigcupY)
using bij-betw-vertices fun-eq-on-bij-betw by blast
show g\vdashX=Y using setsetmapim-cong[OF assms] surj-simplex-map by simp
qed
lemma iso-comp:
assumes ChamberComplexIsomorphism Y Z g
shows ChamberComplexIsomorphism X Z (g\circf)
by
rule ChamberComplexIsomorphism.intro, rule comp,
rule ChamberComplexIsomorphism.axioms(1),
rule assms, unfold-locales, rule bij-betw-trans,
rule bij-betw-vertices,
rule ChamberComplexIsomorphism.bij-betw-vertices,
rule assms
)
(simp add:
setsetmapim-comp surj-simplex-map assms
ChamberComplexIsomorphism.surj-simplex-map
)
lemma inj-on-chamber-system: inj-on ((`)f) domain.C
proof (rule inj-onI)
fix C D show \llbracketC C domain.C; D d domain.C; f}\mp@subsup{f}{}{\prime}C=\mp@subsup{f}{}{\prime}D\rrbracket\LongrightarrowC=
using domain.chamber-system-def domain.chamber-pconnect[of C D]
pgallery-map codomain.pgalleryD-distinct
by fastforce
qed
lemma inv:ChamberComplexIsomorphism Y X (the-inv-into (UX) f)
proof
show bij-betw (the-inv-into (UX) f) (UY) (UX)
using bij-betw-vertices bij-betw-the-inv-into by fast
show 4:(the-inv-into (\bigcupX)f)\vdashY=X
using bij-betw-imp-inj-on[OF bij-betw-vertices] surj-simplex-map
setsetmapim-the-inv-into

```
```

    by force
    next
fix C assume C: codomain.chamber C
hence }\mp@subsup{C}{}{\prime}:C\inf\vdashX\mathrm{ using codomain.chamberD-simplex surj-simplex-map by fast
show domain.chamber (the-inv-into (UX) f'C)
proof (rule domain.chamberI)
from C' obtain D where D\inX the-inv-into (UX) f'}C=
using the-inv-into-f-im-f-im[OF inj] by blast
thus the-inv-into ( }\bigcupX)f'C\inX by sim
fix z}\mathrm{ assume z:z<X the-inv-into ( }\bigcupX)f'C\subseteq
with C have f}\mp@subsup{f}{}{\prime}z=
using C' f-im-the-inv-into-f-im[OF inj, of C] surj-simplex-map
codomain.chamberD-maximal[of C f'z]
by blast
with z(1) show z = the-inv-into (UX) f'C
using the-inv-into-f-im-f-im[OF inj] by auto
qed
from C show card (the-inv-into ( }\bigcupX)f'C)=card
using C' codomain.finite-chamber
subset-inj-on[OF inj-on-the-inv-into, OF inj, of C]
by (fast intro: inj-on-iff-eq-card[THEN iffD1])
qed
lemma chamber-distance-map:
assumes domain.chamber C domain.chamber D
shows codomain.chamber-distance ( }\mp@subsup{f}{}{`}C)(\mp@subsup{f}{}{`}D)
domain.chamber-distance C D
proof (cases f`C=f`D)
case True
moreover with assms have C=D
using inj-onD[OF inj-on-chamber-system] domain.chamber-system-def
by simp
ultimately show ?thesis
using domain.chamber-distance-def codomain.chamber-distance-def by simp
next
case False
define Cs Ds where Cs = (ARG-MIN length Cs.domain.gallery (C\#Cs@[D]))
and Ds = (ARG-MIN length Ds.codomain.gallery (f`C # Ds @ [f`D]))
from assms False Cs-def have codomain.gallery (f`C # f=Cs @ [f`D])
using gallery-map domain.maxsimp-connect[of C D]
arg-min-natI[of \lambdaCs. domain.gallery (C\#Cs@[D])]
by fastforce
moreover from assms Cs-def
have \Es.codomain.gallery (f`C # Es @ [f`D])\Longrightarrow
length (f\modelsCs) \leq length Es
using ChamberComplexIsomorphism.gallery-map[OF inv]
the-inv-into-f-im-f-im[OF inj, of C] the-inv-into-f-im-f-im[OF inj, of D]
domain.chamberD-simplex[of C] domain.chamberD-simplex[of D]
domain.maxsimp-connect[of C D]

```
```

            arg-min-nat-le[of \lambdaCs.domain.gallery (C#Cs@[D]) - length]
    ```
    by force
    ultimately have length \(D s=\) length \((f \models C s)\)
    unfolding \(D s\)-def by (fast intro: arg-min-equality)
    with False Cs-def Ds-def show ?thesis
    using domain.chamber-distance-def codomain.chamber-distance-def by auto
qed
lemma face-distance-map:
    assumes domain.chamber \(C F \in X\)
    shows codomain.face-distance \(\left(f^{\prime} F\right)\left(f^{\prime} C\right)=\) domain.face-distance \(F C\)
proof -
    define \(D D^{\prime}\) invf where \(D=\) domain.closest-supchamber \(F C\)
        and \(D^{\prime}=\) codomain.closest-supchamber \(\left(f^{\iota} F\right)\left(f^{\iota} C\right)\)
        and invf \(=\) the-inv-into \((\bigcup X) f\)
    from assms \(D\)-def \(D^{\prime}\)-def invf-def have chambers:
        codomain.chamber \(\left(f^{\prime} C\right)\) domain.chamber \(D\) codomain.chamber \(D^{\prime}\)
        codomain.chamber ( \(f^{\prime} D\) ) domain.chamber (invf' \(D\) ')
        using domain.closest-supchamberD(1) simplex-map
            codomain.closest-supchamberD (1) chamber-map
            ChamberComplexIsomorphism.chamber-map[OF inv]
    by auto
    have codomain.chamber-distance \(D^{\prime}\left(f^{‘} C\right) \leq\) domain.chamber-distance \(D C\)
    proof-
    from assms \(D\)-def \(D^{\prime}\)-def
        have codomain.chamber-distance \(D^{\prime}\left(f^{\prime} C\right) \leq\)
                        codomain.chamber-distance \(\left(f^{\star} D\right)\left(f^{\bullet} C\right)\)
            using chambers(4) domain.closest-supchamberD(2)
                codomain.closest-supchamber-def
            by (fastforce intro: arg-min-nat-le)
        with assms \(D\)-def \(D^{\prime}\)-def show ?thesis
            using chambers(2) chamber-distance-map by simp
qed
moreover
    have domain.chamber-distance \(D C \leq\) codomain.chamber-distance \(D^{\prime}\left(f^{‘} C\right)\)
proof-
    from assms \(D^{\prime}\)-def have \(i n v f^{\prime} f^{\prime} F \subseteq i n v f^{\prime} D^{\prime}\)
        using chambers (1) simplex-map codomain.closest-supchamberD(2) by fast
    with \(\operatorname{assms}\) (2) invf-def have \(F \subseteq i n v f^{\prime} D\) '
            using the-inv-into-f-im-f-im[OF inj, of F] by fastforce
    with \(D\)-def
            have domain.chamber-distance \(D C \leq\)
                domain.chamber-distance (invf \({ }^{\prime} D{ }^{\prime}\) ) \(C\)
            using chambers(5) domain.closest-supchamber-def
            by (auto intro: arg-min-nat-le)
            with assms(1) invf-def show ?thesis
            using chambers \((3,5)\) surj-simplex-map codomain.chamberD-simplex
```

                f-im-the-inv-into-f-im[OF inj, of D']
                chamber-distance-map[of invf'D' C]
            by fastforce
    qed
    ultimately show ?thesis
    using D-def D'-def domain.face-distance-def codomain.face-distance-def
    by simp
    qed
end

```

\subsection*{4.4.6 Endomorphisms}
locale ChamberComplexEndomorphism \(=\) ChamberComplexMorphism X X f
    for \(X\) :: 'a set set
    and \(f::{ }^{\prime} a \Rightarrow^{\prime} a\)
+ assumes trivial-outside : \(v \notin \bigcup X \Longrightarrow f v=v\)
    - to facilitate uniqueness arguments
lemma (in ChamberComplex) trivial-endomorphism:
    ChamberComplexEndomorphism X id
    by (
        rule ChamberComplexEndomorphism.intro, rule trivial-morphism,
        unfold-locales
        )
        simp
context ChamberComplexEndomorphism
begin
abbreviation ChamberSubcomplex \(\equiv\) domain.ChamberSubcomplex
abbreviation Subcomplex \(\equiv\) domain.Subcomplex
abbreviation chamber \(\equiv\) domain.chamber
abbreviation gallery \(\equiv\) domain.gallery
abbreviation \(\mathcal{C} \equiv\) domain.chamber-system
abbreviation label-wrt \(\equiv\) domain.label-wrt
\begin{tabular}{ll} 
lemmas dim-map & \(=\) dim-map \\
lemmas simplex-map & \\
lemmas vertex-map & \(=\) simplex-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
\end{tabular}
```

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 facedist-chdist-mingal-btwmap =
face-distance-eq-chamber-distance-min-gallery-betw-map
lemmas trivial-endomorphism = domain.trivial-endomorphism
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\circf)
proof (rule ChamberComplexEndomorphism.intro)
from assms show ChamberComplexMorphism X X (g\circf)
using comp ChamberComplexEndomorphism.axioms by fast
from assms show ChamberComplexEndomorphism-axioms X (g\circf)
using trivial-outside ChamberComplexEndomorphism.trivial-outside
by unfold-locales auto
qed
lemma restrict-endo:
assumes ChamberSubcomplex Y f\vdashY\subseteqY
shows ChamberComplexEndomorphism Y (restrict1 f (UY))
proof (rule ChamberComplexEndomorphism.intro)
from assms show ChamberComplexMorphism Y Y (restrict1 f (UY))
using ChamberComplexMorphism.cong[of Y Y]
ChamberComplexMorphism.restrict-codomain
restrict-domain fun-eq-on-restrict1
by fast
show ChamberComplexEndomorphism-axioms Y (restrict1 f (\bigcupY))
by unfold-locales simp
qed
lemma funpower-endomorphism:

```
```

    ChamberComplexEndomorphism X (f^n}
    proof (induct n)
case 0 show ?case using trivial-endomorphism subst[of id] by fastforce
next
case (Suc m)
hence ChamberComplexEndomorphism X (f^~m}\circf
using endo-comp by auto
moreover have f^m}\circf=\mp@subsup{f}{}{\wedge`}(\mathrm{ Suc m)
by (simp add: funpow-Suc-right[THEN sym])
ultimately show ?case
using subst[of - - \lambdaf. ChamberComplexEndomorphism X f] by fast
qed
end
lemma (in ChamberComplex) fold-chamber-complex-endomorph-list:
\forallx\inset xs. ChamberComplexEndomorphism X (fx)\Longrightarrow
ChamberComplexEndomorphism X (fold f xs)
proof (induct xs)
case Nil show ?case using trivial-endomorphism subst[of id] by fastforce
next
case (Cons x xs)
hence ChamberComplexEndomorphism X (fold f xs \circf x)
using ChamberComplexEndomorphism.endo-comp by auto
moreover have fold f xs \circfx= fold f(x\#xs) by simp
ultimately show ?case
using subst[of - - \lambdaf. ChamberComplexEndomorphism X f] by fast
qed
context ChamberComplexEndomorphism
begin
lemma split-gallery:
\llbracketC\inf\vdash\mathcal{C};D\in\mathcal{C}-f\vdash\mathcal{C};\mathrm{ gallery (C\#Cs@[D])】 ב}
\existsAs A B Bs.A\inf\vdash\mathcal{C}\wedgeB\in\mathcal{C}-f\vdash\mathcal{C}\wedgeC\#Cs@[D]=As@A\#B\#Bs
proof (induct Cs arbitrary: C)
case Nil
define As :: 'a set list where As= []
hence C\#[]@[D]=As@C\#D\#As by simp
with Nil(1,2) show ?case by auto
next
case (Cons E Es)
show ?case
proof (cases E\inf\vdash\mathcal{C})
case True
from Cons(4) have gallery (E\#Es@[D])
using gallery-Cons-reduce by simp
with True obtain As A B Bs
where 1:A\inf\vdash\mathcal{C}B\in\mathcal{C}-f\vdash\mathcal{C}E\#Es@[D]=As@A\#B\#Bs

```
```

        using Cons(1)[of E] Cons(3)
        by blast
        from 1(3) have C#(E#Es)@[D]=(C#As)@A#B#Bs by simp
        with 1(1,2) show ?thesis by blast
    next
        case False
        hence E\in\mathcal{C}-f\vdash\mathcal{C}\mathrm{ using gallery-chamber-system[OF Cons(4)] by simp}\\mp@code{Col}
        moreover have C#(E#Es)@[D]=[]@C#E#(Es@[D]) by simp
        ultimately show ?thesis using Cons(2) by blast
    qed
    qed
lemma respects-labels-adjacent:
assumes label-wrt B \varphi chamber C chamber D C~D \forallv\inC.\varphi (fv)=\varphiv
shows }\forallv\inD.\varphi(fv)=\varphi
proof (cases C=D)
case False have CD:C\not=D by fact
with assms(4) obtain w where w: w\not\inDC=insert w (C\capD)
using adjacent-int-decomp by fast
with assms(2) have fC: fw\not\in f`}(C\capD) f`C= insert (fw) (f`(C\capD))     using chamber-vertex-outside-facet-image[of w C\capD] by auto     show ?thesis     proof     fix v}\mathrm{ assume v: veD     show }\varphi(fv)=\varphi     proof (cases v\inC)         case False         with assms(3,4) v have fD: fv\not\in f             using adjacent-sym[of C D] adjacent-conv-insert[of D C v}                 chamber-vertex-outside-facet-image[of v D\capC]         by auto     have }\varphi(fv)=\varphi(fw     proof (cases f}\mp@subsup{f}{}{`}C=\mp@subsup{f}{}{\prime}D
case True
with fC fD have fv=fw by (auto simp add: Int-commute)
thus ?thesis by simp
next
case False
from assms(2-4) have chamber (f`C) chamber (f`D) and fCfD: f}\mp@subsup{f}{}{\prime}C~\mp@subsup{f}{}{\prime}
using chamber-map adj-map by auto
moreover from assms(4) fC fCfD False have fw\inf`C-f`D
using adjacent-to-adjacent-int[of C D f] by auto
ultimately show ?thesis
using assms(4) fD fCfD False adjacent-sym
adjacent-to-adjacent-int[of D C f]
label-wrt-adjacent[OF assms(1),of f`C f}D\mp@code{f wfv,THEN sym]
by auto
qed
with False v w assms(5) show ?thesis

```
using label-wrt-adjacent[OF assms(1-4), of \(w v\), THEN sym] by fastforce qed (simp add: assms(5))
qed
qed (simp add: assms(5))
lemma respects-labels-gallery:
assumes label-wrt \(B \varphi \forall v \in C . \varphi(f v)=\varphi v\)
shows gallery \((C \# C s @[D]) \Longrightarrow \forall v \in D . \varphi(f v)=\varphi v\)
proof (induct Cs arbitrary: \(D\) rule: rev-induct)
case Nil with assms(2) show ?case
using galleryD-chamber galleryD-adj
respects-labels-adjacent \([O F \operatorname{assms}(1)\), of \(C D]\)
by force
next
case (snoc E Es)
with assms(2) show ?case
using gallery-append-reduce1[of C\#Es@[E]] galleryD-chamber galleryD-adj binrelchain-append-reduce2[of adjacent \(C \# E s[E, D]]\) respects-labels-adjacent \([O F \operatorname{assms}(1)\), of \(E D]\)
by force
qed
lemma respect-label-fix-chamber-imp-fun-eq-on:
assumes label : label-wrt \(B \varphi\)
and chamber: chamber \(C f^{\prime} C=g^{`} C\)
and respect: \(\forall v \in C . \varphi(f v)=\varphi v \forall v \in C . \varphi(g v)=\varphi v\)
shows fun-eq-on fg \(C\)
proof (rule fun-eq-onI)
fix \(v\) assume \(v \in C\)
moreover with respect have \(\varphi(f v)=\varphi(g v)\) by simp
ultimately show \(f v=g v\)
using label chamber chamber-map chamber-system-def label-wrtD[of B \(\varphi\) f \(\left.^{\prime} C\right]\) bij-betw-imp-inj-on [of \(\varphi\) ] inj-onD
by fastforce
qed
lemmas respects-label-fixes-chamber-imp-fixespointwise \(=\) respect-label-fix-chamber-imp-fun-eq-on[of --id, simplified]
end

\subsection*{4.4.7 Automorphisms}
locale ChamberComplexAutomorphism \(=\) ChamberComplexIsomorphism XXf for \(X::\) 'a set set and \(f::{ }^{\prime} a{ }^{\prime}{ }^{\prime} a\)
+ assumes trivial-outside : \(v \nsubseteq \bigcup X \Longrightarrow f v=v\)
- to facilitate uniqueness arguments
sublocale ChamberComplexAutomorphism < ChamberComplexEndomorphism using trivial-outside by unfold-locales fast
lemma (in ChamberComplex) trivial-automorphism:
ChamberComplexAutomorphism X id
using trivial-isomorphism
by unfold-locales (auto intro: ChamberComplexAutomorphism.intro)
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 (rule bijI)
show inj f
proof (rule injI)
fix }xy\mathrm{ assume f}x=fy\mathrm{ thus }x=
using bij-betw-imp-inj-on[OF bij-betw-vertices] inj-onD[of f UX x y]
vertex-map trivial-outside
by (cases }x\in\bigcupXy\in\bigcupX rule: two-cases) aut
qed
show surj f unfolding surj-def
proof
fix y show }\existsx.y=f
using bij-betw-imp-surj-on[OF bij-betw-vertices]
trivial-outside[THEN sym, of y]
by (cases }y\in\bigcupX)\mathrm{ auto
qed
qed
lemma comp:
assumes ChamberComplexAutomorphism X g
shows ChamberComplexAutomorphism X (g\circf)
proof (
rule ChamberComplexAutomorphism.intro,
rule ChamberComplexIsomorphism.intro,
rule ChamberComplexMorphism.comp
)
from assms show ChamberComplexMorphism X X g
using ChamberComplexAutomorphism.chamber-morphism by fast
show ChamberComplexIsomorphism-axioms X X (g\circf)
proof
from assms show bij-betw (g\circf) (\bigcupX) (\bigcupX)
using bij-betw-vertices ChamberComplexAutomorphism.bij-betw-vertices

```
```

                bij-betw-trans
            by fast
    from assms show (g\circf)\vdashX=X
            using surj-simplex-map ChamberComplexAutomorphism.surj-simplex-map
            by (force simp add: setsetmapim-comp)
    qed
show ChamberComplexAutomorphism-axioms X ( }g\circf
using trivial-outside ChamberComplexAutomorphism.trivial-outside[OF assms]
by unfold-locales auto
qed unfold-locales
lemma equality:
assumes ChamberComplexAutomorphism X g fun-eq-on fg(UX)
shows f=g
proof
fix }x\mathrm{ show f x = g x
using trivial-outside fun-eq-onD[OF assms(2)]
ChamberComplexAutomorphism.trivial-outside[OF assms(1)]
by force
qed
end

```

\subsection*{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 set set
and $f::{ }^{\prime} a \Rightarrow^{\prime} a$

+ assumes retraction: $v \in \bigcup X \Longrightarrow f(f v)=f v$
begin
lemmas simplex-map $=$ simplex-map
lemmas chamber-map $=$ chamber-map
lemmas gallery-map $=$ gallery-map
lemma vertex-retraction: $v \in f^{\iota}(\bigcup X) \Longrightarrow f v=v$
using retraction by fast
lemma simplex-retraction 1: $x \in f \vdash X \Longrightarrow$ fixespointwise $f x$
using retraction fixespointwiseI $[$ of $x f$ ] by auto
lemma simplex-retraction2: $x \in f \vdash X \Longrightarrow f^{\prime} x=x$
using retraction retraction $[$ THEN sym] by blast
lemma chamber-retraction1: $C \in f \vdash \mathcal{C} \Longrightarrow$ fixespointwise $f C$
using chamber-system-simplices simplex-retraction1 by auto

```
```

lemma chamber-retraction2: }C\inf\vdash\mathcal{C}\Longrightarrow\mp@subsup{f}{}{`}C=     using chamber-system-simplices simplex-retraction2[of C] by auto lemma respects-labels:     assumes label-wrt B \varphi v\in(\bigcupX)     shows }\varphi(fv)=\varphi proof-     from assms(2) obtain C where chamber C v\inC using simplex-in-max by fast     thus ?thesis         using chamber-retraction1[of C] chamber-system-def chamber-map                 maxsimp-connect[of f'C C] chamber-retraction1[of f}\mp@subsup{f}{}{\prime}C                 respects-labels-gallery[OF assms(1),THEN bspec, of f`C - C v]
by (force simp add: fixespointwiseD)
qed
end

```

\subsection*{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 $f::{ }^{\prime} a \Rightarrow{ }^{\prime} a$

+ assumes folding:
chamber $C \Longrightarrow C \in f \vdash X \Longrightarrow$
$\exists!D$. chamber $D \wedge D \notin f \vdash X \wedge f^{‘} D=C$
begin
lemmas folding-ex $\quad=$ ex1-implies-ex[OF folding]
lemmas chamber-system-into $=$ chamber-system-into
lemmas gallery-map $\quad=$ 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 }\equiv{x\inX.\existsC\in\mathcal{C}-f\vdash\mathcal{C}.x\subseteqC
abbreviation Y \equivopp-half-apartment
lemma opp-half-apartment-subset-complex: Y\subseteqX
using opp-half-apartment-def by fast
lemma simplicialcomplex-opp-half-apartment: SimplicialComplex Y
proof
show }\forallx\inY\mathrm{ . finite }
using opp-half-apartment-subset-complex finite-simplices by fast
next
fix x y assume }x\inYy\subseteqx\mathrm{ thus }y\in
using opp-half-apartment-subset-complex faces[of x y]
unfolding opp-half-apartment-def
by auto
qed
lemma subcomplex-opp-half-apartment: Subcomplex Y
using opp-half-apartment-subset-complex simplicialcomplex-opp-half-apartment
by fast
lemma opp-half-apartmentI: \llbracketx\inX;C\in\mathcal{C}-f\vdash\mathcal{C};x\subseteqC\rrbracket\Longrightarrowx\inY
using opp-half-apartment-def by auto
lemma opp-chambers-subset-opp-half-apartment: }\mathcal{C}-f\vdash\mathcal{C}\subseteq
proof
fix C assume C\in\mathcal{C}-f\vdash\mathcal{C}
thus C\inY using chamber-system-simplices opp-half-apartmentI by auto
qed
lemma maxsimp-in-opp-half-apartment:
assumes SimplicialComplex.maxsimp Y C
shows }C\in\mathcal{C}-f\vdash\mathcal{C
proof-
from assms obtain D where D: D\in\mathcal{C}-f\vdash\mathcal{C}C\subseteqD
using SimplicialComplex.maxsimpD-simplex[
OF simplicialcomplex-opp-half-apartment, of C
]
opp-half-apartment-def
by auto
with assms show ?thesis
using opp-chambers-subset-opp-half-apartment
SimplicialComplex.maxsimpD-maximal[
OF simplicialcomplex-opp-half-apartment
]

```
```

    by force
    qed
lemma chamber-in-opp-half-apartment:
SimplicialComplex.maxsimp Y C\Longrightarrow chamber C
using maxsimp-in-opp-half-apartment chamber-system-def by fast
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 \(C \equiv T H E D . D \in \mathcal{C}-f \vdash \mathcal{C} \wedge f^{\bullet} D=C\) abbreviation flop \(C \equiv\) if \(C \in f \vdash \mathcal{C}\) then opp-chamber \(C\) else \(f^{‘} C\)
lemma inj-on-opp-chambers':
assumes chamber \(C\) C \(\notin f \vdash X\) chamber \(D\) \(D \notin f \vdash X f^{\iota} C=f^{\iota} D\)
shows \(C=D\)
proof-
from \(\operatorname{assms}(1)\) folding have ex1: \(\exists!B\). chamber \(B \wedge B \notin f \vdash X \wedge f^{\prime} B=f^{‘} C\) using chamberD-simplex chamber-map by auto
from assms show ?thesis using ex1-unique[OF ex1, of CD] by blast qed
lemma inj-on-opp-chambers": \(\llbracket C \in \mathcal{C}-f \vdash \mathcal{C} ; D \in \mathcal{C}-f \vdash \mathcal{C} ; f^{\prime} C=f^{\prime} D \rrbracket \Longrightarrow C=D\)
using chamber-system-def chamber-system-image inj-on-opp-chambers' by auto
lemma inj-on-opp-chambers: inj-on (( \() f(\mathcal{C}-f \vdash \mathcal{C})\)

lemma opp-chambers-surj: \(f \vdash(\mathcal{C}-(f \vdash \mathcal{C}))=f \vdash \mathcal{C}\)
proof (rule seteqI)
fix \(D\) assume \(D: D \in f \vdash \mathcal{C}\)
from this obtain \(B\) where chamber \(B B \notin f \vdash X f^{\iota} B=D\)
using chamber-system-def chamber-map chamberD-simplex folding-ex[of D]
by auto
thus \(D \in f \vdash(\mathcal{C}-f \vdash \mathcal{C})\)
using chamber-system-image chamber-system-def by auto
qed fast
lemma opp-chambers-bij: bij-betw ((') f) \((\mathcal{C}-(f \vdash \mathcal{C}))(f \vdash \mathcal{C})\)
using inj-on-opp-chambers opp-chambers-surj bij-betw-def[of (') f] by auto
```

lemma folding':
assumes C\inf\vdash\mathcal{C}
shows }\exists!D\in\mathcal{C}-f\vdash\mathcal{C}.\mp@subsup{f}{}{`}D= proof (rule ex-ex1I)     from assms show \existsD.D\in\mathcal{C}-f\vdash\mathcal{C}\wedge f         using chamber-system-image chamber-system-def folding-ex[of C] by auto next     fix B D assume B \in\mathcal{C}-f\vdash\mathcal{C}\wedge\mp@subsup{f}{}{\prime}B=CD\in\mathcal{C}-f\vdash\mathcal{C}\wedge\mp@subsup{f}{}{\prime}D=C     with assms show B=D         using chamber-system-def chamber-system-image chamber-map                 chamberD-simplex ex1-unique[OF folding, of C B D]         by auto qed lemma opp-chambers-distinct-map:     set Cs\subseteq\mathcal{C}-f\vdash\mathcal{C}\Longrightarrowdistinct Cs \Longrightarrow distinct (f)=Cs)     using distinct-map subset-inj-on[OF inj-on-opp-chambers] by auto lemma opp-chamberD1: C\inf\vdash\mathcal{C}\Longrightarrowopp-chamber C }\in\mathcal{C}-f\vdash\mathcal{C     using theI'[OF folding'] by simp lemma opp-chamberD2: }C\inf\vdash\mathcal{C}\Longrightarrow\mp@subsup{f}{}{\prime}(\mathrm{ opp-chamber }C)=     using theI'[OF folding'] by simp lemma opp-chamber-reverse: }C\in\mathcal{C}-f\vdash\mathcal{C}\Longrightarrow\mathrm{ opp-chamber (f`C) =C
using the1-equality[OF folding] by simp
lemma f-opp-chamber-list:
set Cs \subseteqf\vdash\mathcal{C}\Longrightarrowf=(map opp-chamber Cs)=Cs
using opp-chamberD2 by (induct Cs) auto
lemma flop-chamber: chamber C \Longrightarrow chamber (flop C)
using chamber-map opp-chamberD1 chamber-system-def by auto
end

```

\subsection*{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.

\subsection*{4.5.1 Locales and basic facts}
```

locale ThinishChamberComplex = ChamberComplex X
for }X\mathrm{ :: 'a set set

+ assumes thinish:

```

```

    - being adjacent to a chamber, such a D would also be a chamber (see lemma
    chamber-adj)
begin
lemma facet-unique-other-chamber:
\llbracket chamber B; z\triangleleftB; chamber C;z\triangleleftC; chamber D; z\triangleleftD;C\not=B;D\not=B\rrbracket
C=D
using chamberD-simplex bex1-equality[OF thinish,OF - - bexI, of B z C C D]
by auto
lemma finite-adjacentset:
assumes chamber C
shows finite (adjacentset C)
proof (cases X = {{}})
case True thus ?thesis using adjacentset-def by simp
next
case False
moreover have finite ( }\bigcupv\inC.{D\inX.C-{v}\triangleleftD}
proof
from assms show finite C using finite-chamber by simp
next
fix v}\mathrm{ assume vGC
with assms have Cv:C-{v}\triangleleftC
using chamberD-simplex facetrel-diff-vertex by fast
with assms have C:C\in{D\inX.C-{v}\triangleleftD}
using chamberD-simplex by fast
show finite {D\inX.C-{v}\triangleleftD}
proof (cases {D\inX.C-{v}\triangleleftD}-{C}={})
case True
hence 1: {D\inX.C-{v}\triangleleftD}={C} using C by auto
show ?thesis using ssubst[OF 1, of finite] by simp
next
case False
from this obtain D where D: D\inX-{C} C-{v}\triangleleftD by fast
with assms have 2: {D\inX.C-{v}\triangleleftD}\subseteq{C,D}
using Cv chamber-shared-facet[of C] facet-unique-other-chamber [of C-D]
by fastforce
show ?thesis using finite-subset[OF 2] by simp
qed
qed
ultimately show ?thesis
using assms adjacentset-conv-facetchambersets by simp
qed

```
```

lemma label-wrt-eq-on-adjacent-vertex:
fixes v v':: 'a
and zz'::' 'a set

```

```

    and }\mp@subsup{D}{}{\prime}:\mp@subsup{D}{}{\prime}\equiv\mathrm{ insert v}\mp@subsup{v}{}{\prime}\mp@subsup{z}{}{\prime
    assumes label :label-wrt B ffv=f v'
    and chambers: chamber C chamber D chamber D' }z\triangleleftC\mp@subsup{z}{}{\prime}\triangleleftCD\not=C D'\not=
    shows }D=\mp@subsup{D}{}{\prime
    proof (
rule facet-unique-other-chamber, rule chambers(1), rule chambers(4),
rule chambers(2)
)
from D D' chambers(1-5) have z:z\triangleleftD and \mp@subsup{z}{}{\prime}:\mp@subsup{z}{}{\prime}\triangleleft\mp@subsup{D}{}{\prime}
using chambers-share-facet by auto
show z\triangleleftD by fact
from chambers(4,5) obtain w w'
where w:w\not\inz C= insert w z
and \quad w': w'}\not=\mp@subsup{z}{}{\prime}C=\mathrm{ insert }\mp@subsup{w}{}{\prime}\mp@subsup{z}{}{\prime
unfolding facetrel-def
by fastforce
from w' D' chambers(1,3) have f}\mp@subsup{f}{}{\prime}\mp@subsup{z}{}{\prime}=\mp@subsup{f}{}{\prime}C-{f\mp@subsup{v}{}{\prime}
using z' label-wrtD'[OF label(1), of C] bij-betw-imp-inj-on[of f C]
facetrel-complement-vertex[of z']
label-wrt-adjacent-shared-facet[OF label(1), of v']
by simp
moreover from wD chambers(1,2) have f`z = f`}C-{fv
using z label-wrtD'[OF label(1), of C] bij-betw-imp-inj-on[of f C]
facetrel-complement-vertex[of z]
label-wrt-adjacent-shared-facet[OF label(1), of v]
by simp
ultimately show z\triangleleft\mp@subsup{D}{}{\prime}
using z' chambers(1,4,5) label(2) facetrel-subset
label-wrtD'[OF label(1), of C]
bij-betw-imp-inj-on[of f] inj-on-eq-image[of fC z'z
by force
qed (rule chambers(3), rule chambers(6), rule chambers(7))
lemma face-distance-eq-chamber-distance-compare-other-chamber:
assumes chamber C chamber D z\triangleleftC z\triangleleftD C\not=D
chamber-distance C E < chamber-distance D E
shows face-distance z E= chamber-distance C E
unfolding face-distance-def closest-supchamber-def
proof (
rule arg-min-equality, rule conjI, rule assms(1), rule facetrel-subset,
rule assms(3)
)
from assms
show }\B.chamber B\wedgez\subseteqB

```
chamber-distance C \(E \leq\) chamber-distance \(B E\)
using chamber-facet-is-chamber-facet facet-unique-other-chamber by blast
qed
end
lemma (in ChamberComplexIsomorphism) thinish-image-shared-facet:
assumes dom: domain.chamber \(C\) domain.chamber \(D z \triangleleft C z \triangleleft D C \neq D\)
and cod: ThinishChamberComplex Y codomain.chamber \(D^{\prime} f^{\prime} z \triangleleft D^{\prime}\) \(D^{\prime} \neq f^{\prime} C\)
shows \(f^{\bullet} D=D^{\prime}\)
proof (rule ThinishChamberComplex.facet-unique-other-chamber, rule \(\operatorname{cod}(1)\) )
from \(\operatorname{dom}(1,2)\) show codomain.chamber \(\left(f^{‘} C\right)\) codomain.chamber \(\left(f^{‘} D\right)\)
using chamber-map by auto
from dom show \(f^{\prime} z \triangleleft f^{\prime} C f^{\prime} z \triangleleft f^{\prime} D\) using facet-map by auto
from dom have domain.pgallery \([C, D]\)
using domain.pgallery-def adjacentI by fastforce
hence codomain.pgallery \(\left[f^{\prime} C, f^{\prime} D\right]\) using pgallery-map \([o f[C, D]]\) by simp
thus \(f^{‘} D \neq f^{〔} C\) using codomain.pgallery \(D\)-distinct by fastforce
qed (rule \(\operatorname{cod}(2)\), rule \(\operatorname{cod}(3)\), rule \(\operatorname{cod}(4))\)
locale ThinChamberComplex \(=\) ChamberComplex \(X\)
for \(X\) :: 'a set set
+ assumes thin: chamber \(C \Longrightarrow z \triangleleft C \Longrightarrow \exists!D \in X-\{C\} . z \triangleleft D\)
sublocale ThinChamberComplex < ThinishChamberComplex
using thin by unfold-locales simp
context ThinChamberComplex
begin
lemma thinish: ThinishChamberComplex X ..
lemmas face-distance-eq-chamber-distance-compare-other-chamber \(=\)
face-distance-eq-chamber-distance-compare-other-chamber
abbreviation the-adj-chamber \(C z \equiv\) THE \(D . D \in X-\{C\} \wedge z \triangleleft D\)
lemma the-adj-chamber-simplex:
chamber \(C \Longrightarrow z \triangleleft C \Longrightarrow\) the-adj-chamber \(C z \in X\)
using theI' \([\) OF thin \(]\) by fast
lemma the-adj-chamber-facet: chamber \(C \Longrightarrow z \triangleleft C \Longrightarrow z \triangleleft\) the-adj-chamber \(C z\)
using theI' \([\) OF thin \(]\) by fast
lemma the-adj-chamber-is-adjacent:
chamber \(C \Longrightarrow z \triangleleft C \Longrightarrow C \sim\) the-adj-chamber \(C z\)
using the-adj-chamber-facet by (auto intro: adjacentI)
```

lemma the-adj-chamber:
chamber C\Longrightarrowz\triangleleftC\Longrightarrow chamber (the-adj-chamber C z)
using the-adj-chamber-simplex the-adj-chamber-is-adjacent
by (fast intro: chamber-adj)
lemma the-adj-chamber-neq:
chamber C\Longrightarrowz\triangleleftC\Longrightarrow the-adj-chamber C z}\not=
using theI'[OF thin] by fast
lemma the-adj-chamber-adjacentset:
chamber C \Longrightarrowz\triangleleftC\Longrightarrow the-adj-chamber C z \in adjacentset C
using adjacentset-def the-adj-chamber-simplex the-adj-chamber-is-adjacent
by fast
end
lemmas (in ChamberComplexIsomorphism) thin-image-shared-facet =
thinish-image-shared-facet[OF - - - ThinChamberComplex.thinish]

```

\subsection*{4.5.2 The standard uniqueness argument for chamber morphisms of thin chamber complexes}
```

context ThinishChamberComplex

```
context ThinishChamberComplex
begin
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~D f}D\not=\mp@subsup{f}{}{\prime}C\mp@subsup{g}{}{`}D\not=\mp@subsup{g}{}{`}C\mathrm{ chamber ( }\mp@subsup{g}{}{`}D
    and funeq : fun-eq-on fg}
    shows fun-eq-on f g D
proof (rule fun-eq-onI)
    fix v}\mathrm{ assume v: vGD
    show fv=gv
    proof (cases v\inC)
        case True with funeq show ?thesis using fun-eq-onD by fast
    next
    case False
    define FG where F= f}C\cap\cap\mp@subsup{f}{}{`}D\mathrm{ and }G=\mp@subsup{g}{}{`}C\cap\mp@subsup{g}{}{`}
    from morph(1) chambers(1-4) have 1: f}\mp@subsup{f}{}{\prime}C~\mp@subsup{f}{}{\prime}
            using ChamberComplexMorphism.adj-map' by fast
    with F-def chambers(4) have F-facet: F\triangleleftf'C F\triangleleftf'D
            using adjacent-int-facet1[of f`}C]\mathrm{ adjacent-int-facet2 [of f`}C]\mathrm{ by auto
    from F-def G-def chambers have G=F
```

using ChamberComplexMorphism.adj-map'[OF morph(2)] adjacent-to-adjacent-int[of C D g] 1 adjacent-to-adjacent-int[ of $C D f$ ] funeq fun-eq-on-im $[o f f g]$
by force
with $G$-def morph(2) chambers have $F$-facet': $F \triangleleft g^{\prime} D$
using ChamberComplexMorphism.adj-map' adjacent-int-facet2 by blast
with chambers $(1,2,4,5)$ have 2: $g^{`} D=f^{`} D$
using ChamberComplexMorphism.chamber-map[OF morph(1)] F-facet
ChamberComplexMorphism.chamber-map[OF morph(2)]
fun-eq-on-im[OF funeq]
facet-unique-other-chamber[of $\left.f^{\prime} C F g^{\prime} D f^{\prime} D\right]$
by auto
from chambers(3) v False have 3: $D=$ insert $v(D \cap C)$
using adjacent-sym adjacent-conv-insert by fast
from chambers(4) obtain $w$ where $w: w \notin f^{\prime} C w \in f^{\prime} D$
using adjacent-int-decomp[OF adjacent-sym, OF 1] by blast
with 3 have $w=f v$ by fast
moreover from $2 w(2)$ obtain $v^{\prime}$ where $v^{\prime} \in D w=g v^{\prime}$ by auto
ultimately show ?thesis
using $w(1) 3$ funeq by (fastforce simp add: fun-eq-on-im)
qed
qed
lemma standard-uniqueness-pgallery-betw:
assumes morph : ChamberComplexMorphism $W X f$
ChamberComplexMorphism W $\mathrm{X} g$
and chambers: fun-eq-on $f$ g ChamberComplex.gallery $W$ (C\#Cs@[D]) pgallery $(f \models(C \# C s @[D]))$ pgallery $(g \models(C \# C s @[D]))$
shows fun-eq-on fg $D$
proof-
from $\operatorname{morph}(1)$ have $W$ : ChamberComplex $W$
using ChamberComplexMorphism.domain-complex by fast
have 【fun-eq-on fg C ChamberComplex.gallery $W(C \# C s @[D])$; pgallery $(f \models(C \# C s @[D]))$; pgallery $(g \models(C \# C s @[D])) \rrbracket \Longrightarrow$ fun-eq-on $f g D$
proof (induct Cs arbitrary: C)
case Nil from assms Nil(1) show ?case
using ChamberComplex.galleryD-chamber[OF W Nil(2)]
ChamberComplex.galleryD-adj[OF W Nil(2)]
pgalleryD-distinct[OF Nil(3)] pgalleryD-distinct[OF Nil(4)]
pgalleryD-chamber $[O F \operatorname{Nil}(4)]$ standard-uniqueness-dbl[of $W$ f $g C D]$
by auto
next
case (Cons B Bs)
have fun-eq-on f $g B$
proof (rule standard-uniqueness-dbl, rule $\operatorname{morph}(1)$, rule $\operatorname{morph}(2))$
show ChamberComplex.chamber W ChamberComplex.chamber W B C~B
using ChamberComplex.galleryD-chamber[OF W Cons(3)]
ChamberComplex.galleryD-adj[OF W Cons(3)]

```
            by auto
            show f`B\not= f`C using pgalleryD-distinct[OF Cons(4)] by fastforce
            show g}\mp@subsup{g}{}{`}B\not=\mp@subsup{g}{}{`}C\mathrm{ using pgalleryD-distinct[OF Cons(5)] by fastforce
            show chamber (g`B) using pgalleryD-chamber[OF Cons(5)] by fastforce
    qed (rule Cons(2))
    with Cons(1,3-5) show ?case
    using ChamberComplex.gallery-Cons-reduce[OF W, of C B#Bs@[D]]
        pgallery-Cons-reduce[of f`}Cf\models(B#Bs@[D])
        pgallery-Cons-reduce[of g}\mp@subsup{g}{}{`}C|=(B#Bs@[D])
    by force
qed
    with chambers show ?thesis by simp
qed
lemma standard-uniqueness:
    assumes morph : ChamberComplexMorphism WXf
                            ChamberComplexMorphism W X g
    and chamber : ChamberComplex.chamber W C fun-eq-on fg C
    and map-gals:
    \Cs.ChamberComplex.min-gallery W (C#Cs)\Longrightarrow pgallery }(f=(C#Cs)
    \ C s . C h a m b e r C o m p l e x . m i n - g a l l e r y ~ W ~ ( C \# C s ) \Longrightarrow ~ p g a l l e r y ~ ( g \models ( C \# C s ) )
    shows fun-eq-on fg(\W)
proof (rule fun-eq-onI)
    from morph(1) have W: ChamberComplex W
    using ChamberComplexMorphism.axioms(1) by fast
    fix v assume v\in\bigcupW
    from this obtain D where ChamberComplex.chamber W D v\inD
    using ChamberComplex.simplex-in-max[OF W] by auto
    moreover define Cs where Cs = (ARG-MIN length Cs. ChamberComplex.gallery
W(C#Cs@[D]))
    ultimately show fv=gv
        using chamber map-gals[of Cs@[D]]
            ChamberComplex.gallery-least-length[OF W]
                            ChamberComplex.min-gallery-least-length[OF W]
                            standard-uniqueness-pgallery-betw[OF morph(1,2) chamber(2), of Cs]
            fun-eq-onD[of fg D]
    by (cases D=C) auto
qed
lemma standard-uniqueness-isomorphs:
    assumes ChamberComplexIsomorphism WX f
        ChamberComplexIsomorphism W X g
        ChamberComplex.chamber W C fun-eq-on f g C
    shows fun-eq-onfg(\W)
    using assms ChamberComplexIsomorphism.chamber-morphism
            ChamberComplexIsomorphism.domain-complex
            ChamberComplex.min-gallery-pgallery
            ChamberComplexIsomorphism.pgallery-map
    by (blast intro: standard-uniqueness)
```

```
lemma standard-uniqueness-automorphs:
    assumes ChamberComplexAutomorphism Xf
        ChamberComplexAutomorphism X g
        chamber C fun-eq-on f g C
    shows f=g
    using assms ChamberComplexAutomorphism.equality
        standard-uniqueness-isomorphs
        ChamberComplexAutomorphism.axioms(1)
    by blast
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 ThinishChamberComplexFolding $=$ ThinishChamberComplex $X+$ folding: ChamberComplexFolding $X f$ for $X$ :: ' $a$ set set and $f::{ }^{\prime} a \Rightarrow^{\prime} a$
begin
abbreviation opp-chamber $\equiv$ 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 $f^{\prime} D=C$
proof-
from adjacent obtain $z$ where $z: z \triangleleft C z \triangleleft D$ using adjacent-def by fast
moreover from $z(1)$ chambers(1) have $f z: f^{\prime} z=z$
using facetrel-subset [of $z C$ ] chamber-system-simplices
folding.simplicialcomplex-image
SimplicialComplex.faces[of $f \vdash X C$ z]
folding.simplex-retraction2[of z]
by auto
moreover from chambers have $f^{\prime} D \neq D C \neq D$ by auto
ultimately show ?thesis
using chambers chamber-system-def folding.chamber-map

```
folding.facet-map[of D z]
facet-unique-other-chamber[of D z f`D C]
    by force
qed
lemma adjacent-half-chamber-system-image-reverse:
    \llbracketC\inf\vdash\mathcal{C};D\in\mathcal{C}-f\vdash\mathcal{C};C~D\rrbracket\Longrightarrow opp-chamber C=D
    using adjacent-half-chamber-system-image[of C D]
        the1-equality[OF folding.folding']
    by fastforce
lemma chamber-image-closer:
    assumes }D\in\mathcal{C}-f\vdash\mathcal{C}B\inf\vdash\mathcal{C}B\not=\mp@subsup{f}{}{`}D\mathrm{ gallery(B#Ds@[D])
    shows \exists Cs.gallery (B#Cs@[f`}D])\wedge length Cs < length Ds
proof-
    from assms(1,2,4) obtain As A E Es
    where split: A\inf\vdash\mathcal{C}E\in\mathcal{C}-f\vdash\mathcal{C}B#Ds@[D]=As@A#E#Es
    using folding.split-gallery[of B D Ds]
    by blast
    from assms(4) split(3) have A~E
    using gallery-append-reduce2[of As A#E#Es] galleryD-adj[of A#E#Es]
    by simp
    with assms(2) split(1,2)
    have fB: f}\mp@subsup{f}{}{\prime}B=B\mathrm{ and }fA:\mp@subsup{f}{}{\prime}A=A\mathrm{ and }fE:\mp@subsup{f}{}{\prime}E=
    using folding.chamber-retraction2 adjacent-half-chamber-system-image[of A E]
    by auto
    show \existsCs. gallery (B#Cs@[f`D]) ^ length Cs < length Ds
    proof (cases As)
    case Nil have As:As=[] by fact
    show ?thesis
    proof (cases Es rule: rev-cases)
        case Nil with split(3) As assms(3) fE show ?thesis by simp
    next
        case (snoc Fs F)
        with assms(4) split(3) As fE
            have Ds=E#Fs gallery (B#f|=Fs @ [f`D])
            using fB folding.gallery-map[of B#E#Fs@[D]] gallery-Cons-reduce
            by auto
        thus ?thesis by auto
    qed
    next
    case (Cons H Hs)
    show ?thesis
    proof (cases Es rule: rev-cases)
            case Nil
            with assms(4) Cons split(3)
                have decomp: Ds=Hs@[A] D=E gallery (B#Hs@[A,D])
            by auto
            from decomp(2,3)fB fA fE have gallery (B#f\modelsHs @ [f`D])
```

```
            using folding.gallery-map gallery-append-reduce1[of B # f=Hs @ [f`D]]
            by force
        with decomp(1) show ?thesis by auto
    next
            case (snoc Fs F)
            with split(3) Cons assms(4) fB fA fE
            have decomp: Ds = Hs@A#E#Fs gallery (B#f=(Hs@A#Fs)@ [f`D])
            using folding.gallery-map[of B#Hs@A#E#Fs@[D]]
                gallery-remdup-adj[of B#f =Hs A f =Fs@[f`D]]
            by auto
            from decomp(1) have length (f =(Hs@A#Fs))< length Ds by simp
            with decomp(2) show ?thesis by blast
        qed
    qed
qed
lemma chamber-image-subset:
    assumes D: D\in\mathcal{C}-f\vdash\mathcal{C}
    defines C:C\equivf}\mp@subsup{|}{}{\prime}
    defines closerTo C \equiv{B\in\mathcal{C}.chamber-distance B C < chamber-distance B D}
    shows f\vdash\mathcal{C}\subseteqcloserToC
proof
    fix B assume B:B\inf\vdash\mathcal{C}
    hence }\mp@subsup{B}{}{\prime}:B\in\mathcal{C}\mathrm{ using folding.chamber-system-into by fast
    show B \in closerToC
    proof (cases B=C)
        case True with B D closerToC-def show ?thesis
            using B' chamber-distance-def by auto
    next
        case False
        define Ds where Ds = (ARG-MIN length Ds.gallery (B#Ds@[D]))
        with B C D False closerToC-def show ?thesis
            using chamber-system-def folding.chamber-map gallery-least-length[of B D]
                chamber-image-closer[of D B Ds]
                chamber-distance-le chamber-distance-def[of B D]
            by fastforce
    qed
qed
lemma gallery-double-cross-not-minimal-Cons1:
    \llbracketB\inf\vdash\mathcal{C};C\in\mathcal{C}-f\vdash\mathcal{C};D\inf\vdash\mathcal{C};\mathrm{ gallery (B#C#Cs@[D])】 ב}
        \neg min-gallery (B#C#Cs@[D])
using galleryD-adj[of B#C#Cs@[D]]
            adjacent-half-chamber-system-image[of B C]
            folding.gallery-map[of B#C#Cs@[D]]
            gallery-Cons-reduce[of B B #f =Cs @ [D]]
            is-arg-minD2[of length ( }\lambda\mathrm{ Ds. maxsimpchain (B#Ds@[D])) - fl=Cs]
            min-maxsimpchain.simps(3)[of B C#Cs D]
by(simp add: folding.chamber-retraction2)(meson impossible-Cons not-less)
```

```
lemma gallery-double-cross-not-minimal1:
    \llbracketB\inf\vdash\mathcal{C};C\in\mathcal{C}-f\vdash\mathcal{C};D\inf\vdash\mathcal{C};\mathrm{ gallery (B#Bs@C#Cs@[D])】 ב}
    \neg min-gallery(B#Bs@C#Cs@[D])
proof (induct Bs arbitrary: B)
    case Nil thus ?case using gallery-double-cross-not-minimal-Cons1 by simp
next
    case (Cons E Es)
    show ?case
    proof (cases E\inf\vdash\mathcal{C})
        case True
        with Cons(1,3-5) show ?thesis
            using gallery-Cons-reduce[of B E#Es@C#Cs@[D]]
                min-gallery-betw-CCons-reduce[of B E Es@C#Cs D]
            by auto
    next
        case False with Cons(2,4,5) show ?thesis
            using gallery-chamber-system
                gallery-double-cross-not-minimal-Cons1[of B E D Es@C#Cs]
            by force
    qed
qed
end
locale ThinChamberComplexFolding =
    ThinChamberComplex X + folding: ChamberComplexFolding X f
    for X :: 'a set set
    and f :: 'a>'a
sublocale ThinChamberComplexFolding < ThinishChamberComplexFolding ..
context ThinChamberComplexFolding
begin
abbreviation flop \equiv 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}\mp@subsup{f}{}{\prime}C~\mp@subsup{f}{}{\prime}
    shows C~D
proof (cases f`C= f`D)
    case True
    with chambers show C ~ D
```

using folding．inj－on－opp－chambers＇＂$o f$ C D］adjacent－refl［of C］by auto next
case False
from chambers have $C D$ ：chamber $C$ chamber $D$
using chamber－system－def by auto
hence ch－fCD：chamber $\left(f^{〔} C\right)$ chamber（ $\left.f^{〔} D\right)$
using chamber－system－def folding．chamber－map by auto
from adjacent obtain $z$ where $z: z \triangleleft f^{`} C z \triangleleft f^{`} D$
using adjacent－def by fast
from chambers（1）$z(1)$ obtain $y$ where $y: y \triangleleft C f^{\prime} y=z$
using chamber－system－def folding．inj－on－chamber［of C］
inj－on－pullback－facet［of f C z］
by auto
define $B$ where $B=$ the－adj－chamber $C y$
with $C D(1) y(1)$ have $B$ ：chamber $B y \triangleleft B \quad B \neq C$
using the－adj－chamber the－adj－chamber－facet the－adj－chamber－neq by auto
have $f^{\prime} B \neq f^{\prime} C$
proof（cases $B \in f \vdash \mathcal{C}$ ）
case False with chambers（1）show ？thesis
using $B(1,3)$ chamber－system－def folding．inj－on－opp－chambers＂${ }^{\prime}[$ of $B]$
by auto
next
case True show ？thesis
proof
assume $f B-f C$ ：$f^{\prime} B=f^{\iota} C$
with True have $B=f^{\prime} C$ using folding．chamber－retraction2 by auto with $z(1) y(2) B(2)$ chambers（1）have $y=z$
using facetrel－subset［of y B］chamber－system－def chamberD－simplex face－im folding．simplex－retraction2 $[$ of $y]$
by force
with chambers $y(1) z(2)$ have $f^{\prime} D=B$
using $C D(1)$ ch－fCD（2）B facet－unique－other－chamber $[$ of $C y]$ by auto
with $z(2)$ chambers $f B$－$f C$ False show False
using folding．chamber－retraction2 by force
qed
qed
with False z $y$（2）have $f B-f D$ ：$f^{‘} B=f^{〔} D$
using ch－fCD $B(1,2)$ folding．chamber－map folding．facet－map
facet－unique－other－chamber $\left[\right.$ of $\left.f^{\prime} C z\right]$
by force
have $B=D$
proof（cases $B \in f \vdash \mathcal{C}$ ）
case False
with $B(1)$ chambers（2）show ？thesis
using chamber－system－def fB－fD folding．inj－on－opp－chambers＂by simp

## next

case True
with $f B-f D$ have $B=f^{\prime} D$ using folding．chamber－retraction2 by auto moreover with $z(1) y(2) B(2) \operatorname{chambers}(2)$ have $y=z$
using facetrel-subset[of y B] chamber-system-def chamberD-simplex face-im folding.simplex-retraction2 [of $y$ ]
by force
ultimately show ?thesis
using $C D y(1) B$ ch-fCD(1) z(1) False chambers(1) facet-unique-other-chamber[of $B$ y $\left.C f^{`} C\right]$
by auto
qed
with $y(1) B(2)$ show ?thesis using adjacentI by fast
qed
lemma adjacent-opp-chamber:
$\llbracket C \in f \vdash \mathcal{C} ; D \in f \vdash \mathcal{C} ; C \sim D \rrbracket \Longrightarrow$ opp-chamber $C \sim$ opp-chamber $D$
using folding.opp-chamberD1 folding.opp-chamberD2 adjacent-preimage by simp
lemma adjacentchain-preimage:
set $C s \subseteq \mathcal{C}-f \vdash \mathcal{C} \Longrightarrow$ adjacentchain $(f=C s) \Longrightarrow$ adjacentchain $C s$
using adjacent-preimage by (induct Cs rule: list-induct-CCons) auto
lemma gallery-preimage: set $C s \subseteq \mathcal{C}-f \vdash \mathcal{C} \Longrightarrow$ gallery $(f \models C s) \Longrightarrow$ gallery $C s$ using galleryD-adj adjacentchain-preimage chamber-system-def gallery-def by fast
lemma chambercomplex-opp-half-apartment: ChamberComplex folding. $Y$
proof (intro-locales, rule folding.simplicialcomplex-opp-half-apartment, unfold-locales)
define $Y$ where $Y=$ folding. $Y$
fix $y$ assume $y \in Y$
with $Y$-def obtain $C$ where $C \in \mathcal{C}-f \vdash \mathcal{C} y \subseteq C$
using folding.opp-half-apartment-def by auto
with $Y$-def show $\exists x$. SimplicialComplex.maxsimp $Y x \wedge y \subseteq x$
using folding.subcomplex-opp-half-apartment folding.opp-chambers-subset-opp-half-apartment chamber-system-def max-in-subcomplex $[$ of $Y$ ]
by force
next
define $Y$ where $Y=$ folding. $Y$
fix $C D$
assume CD: SimplicialComplex.maxsimp Y C SimplicialComplex.maxsimp Y D
$C \neq D$
from $C D(1,2) Y$-def have $C D^{\prime}: C \in \mathcal{C}-f \vdash \mathcal{C} D \in \mathcal{C}-f \vdash \mathcal{C}$
using folding.maxsimp-in-opp-half-apartment by auto
with $C D(3)$ obtain $D s$
where Ds: ChamberComplex.gallery $(f \vdash X)\left(\left(f^{‘} C\right) \# D s @\left[f^{‘} D\right]\right)$
using folding.inj-on-opp-chambers'"[of C D] chamber-system-def
folding.maxsimp-map-into-image folding.chambercomplex-image ChamberComplex.maxsimp-connect $\left[o f f \vdash X f^{\prime} C f^{‘} D\right]$
by auto
define Cs where Cs $=$ map opp-chamber Ds
from $D s$ have $D s^{\prime}:$ gallery $\left(\left(f^{‘} C\right) \# D s @\left[f^{‘} D\right]\right)$
using folding.chambersubcomplex-image subcomplex-gallery by fast with $D s$ have $D s^{\prime \prime}:$ set $D s \subseteq f \vdash \mathcal{C}$
using folding.chambercomplex-image folding.chamber-system-image ChamberComplex.galleryD-chamber ChamberComplex.chamberD-simplex gallery-chamber-system
by fastforce
have $*$ : set $C s \subseteq \mathcal{C}-f \vdash \mathcal{C}$
proof
fix $B$ assume $B \in$ set $C s$
with Cs-def obtain $A$ where $A \in$ set $D s B=o p p$-chamber $A$ by auto
with $D s^{\prime \prime}$ show $B \in \mathcal{C}-f \vdash \mathcal{C}$ using folding.opp-chamberD1[of $A$ ] by auto
qed
moreover from $C s$ - def $C D^{\prime} D s^{\prime} D s^{\prime \prime} *$ have gallery $(C \# C s @[D])$
using folding.f-opp-chamber-list gallery-preimage $[$ of $C \# C s @[D]]$ by simp
ultimately show $\exists$ Cs. SimplicialComplex.maxsimpchain Y (C \# Cs @ $[D]$ )
using $Y$-def $C D^{\prime}$ folding.subcomplex-opp-half-apartment
folding.opp-chambers-subset-opp-half-apartment
maxsimpchain-in-subcomplex[of YC\#Cs@[D]]
by fastforce
qed
lemma flop-adj:
assumes chamber $C$ chamber $D C \sim D$
shows flop $C \sim$ flop $D$
proof (cases $C \in f \vdash \mathcal{C} \quad D \in f \vdash \mathcal{C}$ rule: two-cases)
case both
with assms(3) show ?thesis using adjacent-opp-chamber by simp
next
case one
with assms(2,3) show ?thesis
using chamber-system-def adjacent-half-chamber-system-image[of C]
adjacent-half-chamber-system-image-reverse adjacent-sym
by $\operatorname{simp}$
next
case other
with assms(1) show ?thesis
using chamber-system-def adjacent-sym[OF assms(3)]
adjacent-half-chamber-system-image[of D]
adjacent-half-chamber-system-image-reverse
by auto
qed (simp add: assms folding.adj-map)
lemma flop-gallery: gallery Cs $\Longrightarrow$ gallery (map flop Cs)
proof (induct Cs rule: list-induct-CCons)
case (CCons B C Cs)
have gallery (flop B \# (flop C) \# map flop Cs)
proof (rule gallery-CConsI)
from CCons(2) show chamber (flop B)
using galleryD-chamber folding.flop-chamber by simp

```
    from CCons(1) show gallery (flop C # map flop Cs)
        using gallery-Cons-reduce[OF CCons(2)] by simp
    from CCons(2) show flop B ~ flop C
    using galleryD-chamber galleryD-adj flop-adj[of B C] by fastforce
    qed
    thus ?case by simp
qed (auto simp add: galleryD-chamber folding.flop-chamber gallery-def)
lemma morphism-half-apartments: ChamberComplexMorphism folding.Y (f\vdashX) f
proof (
    rule ChamberComplexMorphism.intro, rule chambercomplex-opp-half-apartment,
    rule folding.chambercomplex-image, unfold-locales
)
    show
    \C. SimplicialComplex.maxsimp folding.Y C\Longrightarrow
        SimplicialComplex.maxsimp (f\vdashX) (f`C)
    \C. SimplicialComplex.maxsimp folding.Y C \Longrightarrow card (f`C) = card C
    using folding.chamber-in-opp-half-apartment folding.chamber-map
            folding.chambersubcomplex-image chamber-in-subcomplex
            chamberD-simplex folding.dim-map
    by
            auto
qed
lemma chamber-image-complement-closer:
    \llbracket D\in\mathcal{C}-f\vdash\mathcal{C};B\in\mathcal{C}-f\vdash\mathcal{C};B\not=D; gallery (B#Cs@[f`D])\rrbracket\Longrightarrow
            \existsDs.gallery (B#Ds@[D])^ length Ds < length Cs
    using flop-gallery chamber-image-closer[of D f`B map flop Cs]
        folding.opp-chamber-reverse folding.inj-on-opp-chambers"[of B D]
    by force
lemma chamber-image-complement-subset:
    assumes D: D\in\mathcal{C}-f\vdash\mathcal{C}
    defines C:C\equiv }\=\mp@subsup{f}{}{\prime}
    defines closerToD \equiv{B\in\mathcal{C}. chamber-distance B D < chamber-distance B C }
    shows }\quad\mathcal{C}-f\vdash\mathcal{C}\subseteq\mathrm{ closerToD
proof
    fix B assume B: B\in\mathcal{C}-f\vdash\mathcal{C}
    show B \in closerToD
    proof (cases B=D)
        case True with B C closerToD-def show ?thesis
        using chamber-distance-def by auto
    next
        case False
        define Cs where Cs=(ARG-MIN length Cs.gallery (B#Cs@[C]))
        with B C D False closerToD-def show ?thesis
            using chamber-system-def folding.chamber-map[of D]
                gallery-least-length[of B C] chamber-distance-le
                    chamber-image-complement-closer[of D B Cs]
                        chamber-distance-def[of B C]
```

```
        by fastforce
    qed
qed
lemma chamber-image-and-complement:
    assumes D: D\in\mathcal{C}-f\vdash\mathcal{C}
    defines C:C\equivf}\mp@subsup{|}{}{\prime}
    defines closerTo C \equiv{B\in\mathcal{C}.chamber-distance B C< chamber-distance B D}
    and closerToD \equiv{B\in\mathcal{C}.chamber-distance B D<chamber-distance B C}
    shows f\vdash\mathcal{C}=\mathrm{ closerToC }\mathcal{C}-f\vdash\mathcal{C}=\mathrm{ closerToD}
proof-
    from closerToC-def closerToD-def have closerToC \cap closerToD ={} by auto
    moreover from C D closerToC-def closerToD-def
        have}\mathcal{C}=f\vdash\mathcal{C}\cup(\mathcal{C}-f\vdash\mathcal{C}) closerToC\subseteq\mathcal{C}\mathrm{ closerToD}\subseteq\mathcal{C
        using folding.chamber-system-into
        by auto
    moreover from assms have f\vdash\mathcal{C}\subseteqcloserToC \mathcal{C}-f\vdash\mathcal{C}\subseteqcloserToD
            using chamber-image-subset chamber-image-complement-subset by auto
    ultimately show f\vdash\mathcal{C}=\mathrm{ closerToC C C-f言 = closerToD}
    using set-decomp-subset[of \mathcal{C f}\vdash\mathcal{C}] set-decomp-subset[of \mathcal{C C}-f\vdash\mathcal{C}] by auto
qed
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 $f::{ }^{\prime} a \Rightarrow{ }^{\prime} a$
and $g::{ }^{\prime} a \Rightarrow^{\prime} a$
+ fixes $C 0$ :: ' $a$ set
assumes chambers: chamber $C 0 C 0 \sim g^{`} C 0 C 0 \neq g^{`} C 0 f^{\prime} g{ }^{\prime} C 0=C 0$
begin
abbreviation $D 0 \equiv g^{`} C 0$
lemmas chamber-D0 $=$ folding-g.chamber-map $[$ OF $\operatorname{chambers}(1)]$
lemma ThinChamberComplexFolding-f: ThinChamberComplexFolding X f ..
lemma ThinChamberComplexFolding-g: ThinChamberComplexFolding X g ..
lemmas foldf $=$ ThinChamberComplexFolding- $f$
lemmas foldg $=$ ThinChamberComplexFolding-g

```
lemma fg-symmetric: OpposedThinChamberComplexFoldings X g f D0
    using chambers(2-4) chamber-D0 adjacent-sym by unfold-locales auto
lemma basechambers-half-chamber-systems: C0\inf\vdash\mathcal{C}D0\ing\vdash\mathcal{C}
    using chambers(1,4) chamber-D0 chamber-system-def by auto
lemmas basech-halfchsys =
    basechambers-half-chamber-systems
lemma f-trivial-C0:v\inC0\Longrightarrowfv=v
    using chambers(4) chamber-D0 chamberD-simplex[of D0]
        folding-f.vertex-retraction
    by fast
lemmas g-trivial-D0 =
    OpposedThinChamberComplexFoldings.f-trivial-C0[OF fg-symmetric]
lemma double-fold-D0:
    assumes v\inD0-C0
    shows g(fv)=v
proof -
    from assms chambers(2) have 1: D0 = insert v (C0\capD0)
        using adjacent-sym adjacent-conv-insert by fast
    hence f}\mp@subsup{f}{}{`}D0=\operatorname{insert ( }fv)(\mp@subsup{f}{}{`}(C0\capD0))\mathrm{ by fast
    moreover have f}\mp@subsup{f}{}{\prime}(C0\capD0)=C0\capD0 using f-trivial-C0 by forc
    ultimately have C0 = insert (fv) (C0\capDO) using chambers(4) by simp
    hence g'C0 = insert (g(fv)) (g`(C0\capD0)) by force
    moreover have g}\mp@subsup{g}{}{\prime}(C0\capD0)=C0\capD
        using g-trivial-D0 fixespointwise-im[of g D0 C0\capD0]
    by (fastforce intro: fixespointwiseI)
    ultimately have D0 = insert (g(fv)) (C0\capD0) by simp
    with assms show ?thesis using 1 by force
qed
lemmas double-fold-C0 =
    OpposedThinChamberComplexFoldings.double-fold-D0[OF fg-symmetric]
lemma flopped-half-chamber-systems-fg: }\mathcal{C}-f\vdash\mathcal{C}=g\vdash\mathcal{C
proof-
    from chambers(1,3,4) have D0\in\mathcal{C}-f\vdash\mathcal{C}C0\in\mathcal{C}-g\vdash\mathcal{C}
        using chamber-system-def chamber-D0 folding-f.chamber-retraction2[of D0]
                folding-g.chamber-retraction2[of C0]
    by auto
    with chambers(2,4) show ?thesis
    using ThinChamberComplexFolding.chamber-image-and-complement[
                OF ThinChamberComplexFolding-g, of C0
            ]
                ThinChamberComplexFolding.chamber-image-and-complement[
```

```
            OF ThinChamberComplexFolding-f, of D0
            ]
            adjacent-sym[of C0 DO]
    by
qed
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\vdashX
proof (rule seteqI)
    fix a assume a folding-f.Y
    from this obtain C where C\ing\vdash\mathcal{C}a\subseteqC
        using folding-f.opp-half-apartment-def flopped-half-chamber-systems-fg by auto
    thus }a\ing\vdash
        using chamber-system-simplices
            ChamberComplex.faces[OF folding-g.chambercomplex-image, of C]
        by auto
next
    fix b assume b: b\ing\vdashX
    from this obtain C where C: C\in\mathcal{C}}b\subseteq\mp@subsup{g}{}{\prime}
        using simplex-in-max chamber-system-def by fast
    from C(1) have g}\mp@subsup{g}{}{`}C\ing\vdash\mathcal{C}\mathrm{ by fast
    hence g'C\in\mathcal{C}-f\vdash\mathcal{C}\mathrm{ using flopped-half-chamber-systems-fg by simp}\\mp@code{l}
    with C(2) have \existsC\in\mathcal{C}-f\vdash\mathcal{C}.b\subseteqC by auto
    moreover from b have b\inX using folding-g.simplex-map by fast
    ultimately show b folding-f.Y
        unfolding folding-f.opp-half-apartment-def by simp
qed
lemmas flopped-half-apartments-gf =
    OpposedThinChamberComplexFoldings.flopped-half-apartments-fg[
        OF fg-symmetric
    ]
lemma vertex-set-split: \bigcupX=\mp@subsup{f}{}{`}(\bigcupX)\cup\mp@subsup{g}{}{`}(\bigcupX)
-f and g}\mathrm{ will both be the identity on the intersection
proof
    show \bigcupX\supseteq\mp@subsup{f}{}{`}(\bigcupX)\cup\mp@subsup{g}{}{`}(\bigcupX)
    using folding-f.simplex-map folding-g.simplex-map by auto
    show \bigcupX\subseteq\mp@subsup{f}{}{\prime}(\bigcupX)\cup\mp@subsup{g}{}{\prime}(\bigcupX)
    proof
        fix a assume }a\in\bigcup
        from this obtain C where C: chamber C a\inC
            using simplex-in-max by fast
        from C(1) have }C\inf\vdash\mathcal{C}\veeC\ing\vdash\mathcal{C
            using chamber-system-def flopped-half-chamber-systems-fg by auto
```

```
    with C(2) show }a\in(f\cupX)\cup(g\bigcupX
        using chamber-system-simplices by fast
    qed
qed
lemma half-chamber-system-disjoint-union:
    \mathcal{C}=f\vdash\mathcal{C}\cupg\vdash\mathcal{C}(f\vdash\mathcal{C})\cap(g\vdash\mathcal{C})={}
    using folding-f.chamber-system-into
        flopped-half-chamber-systems-fg[THEN sym]
    by
        auto
lemmas halfchsys-decomp =
    half-chamber-system-disjoint-union
lemma chamber-in-other-half-fg: chamber C\LongrightarrowC\not\existsf\vdash\mathcal{C}\LongrightarrowC\ing\vdash\mathcal{C}
    using chamber-system-def half-chamber-system-disjoint-union(1) by blast
lemma adjacent-half-chamber-system-image-fg:
    C\inf\vdash\mathcal{C}\LongrightarrowD\ing\vdash\mathcal{C}\LongrightarrowC~D\Longrightarrowf`D=C
    using ThinChamberComplexFolding.adjacent-half-chamber-system-image[
        OF ThinChamberComplexFolding-f
            ]
    by (simp add: flopped-half-chamber-systems-fg)
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\inf\vdash\mathcal{C}C~g`}
    shows OpposedThinChamberComplexFoldings XfgC
proof
    from assms(1) have C\in\mathcal{C}-g\vdash\mathcal{C}\mathrm{ using flopped-half-chamber-systems-gf by simp}\\mp@code{f}\mathrm{ (1)}
    with assms show chamber C C = g'C f f}\mp@subsup{g}{}{`}C=
        using chamber-system-def adjacent-half-chamber-system-image-fg[of C g'C]
        by auto
qed (rule assms(2))
lemma unique-half-chamber-system-f:
    assumes OpposedThinChamberComplexFoldings X f' g' C0 g'`C0 = D0
    shows f}\quad\mp@subsup{f}{}{\prime}\vdash\mathcal{C}=f\vdash\mathcal{C
proof-
    have 1: OpposedThinChamberComplexFoldings X f g' C0
    proof (rule OpposedThinChamberComplexFoldings.intro)
        show ChamberComplexFolding X f ThinChamberComplex X ..
```

```
    from assms(1) show ChamberComplexFolding X g
        using OpposedThinChamberComplexFoldings.axioms(3) by fastforce
    from assms(2) chambers
        show OpposedThinChamberComplexFoldings-axioms X f g' C0
        by unfold-locales auto
    qed
    define }ab\mathrm{ where }a=\mp@subsup{f}{}{\prime}\vdash\mathcal{C}\mathrm{ and b}=f\vdash\mathcal{C
    hence }a\subseteq\mathcal{C}b\subseteq\mathcal{C}\mathcal{C}-a=\mathcal{C}-
    using OpposedThinChamberComplexFoldings.axioms(2)[OF assms(1)]
        OpposedThinChamberComplexFoldings.axioms(2)[OF 1]
        ChamberComplexFolding.chamber-system-into[of X f]
        ChamberComplexFolding.chamber-system-into[of X f}
        OpposedThinChamberComplexFoldings.flopped-half-chamber-systems-fg[
        OF assms(1)
        ]
        OpposedThinChamberComplexFoldings.flopped-half-chamber-systems-fg[
                OF 1
            ]
        by auto
    hence }a=b\mathrm{ by fast
    with a-def b-def show ?thesis by simp
qed
lemma unique-half-chamber-system-g:
    OpposedThinChamberComplexFoldings X f' g' C0 \Longrightarrow g''C0=D0 \Longrightarrow
        g'\vdash\mathcal{C}=g\vdash\mathcal{C}
    using unique-half-chamber-system-f flopped-half-chamber-systems-fg
        OpposedThinChamberComplexFoldings.flopped-half-chamber-systems-fg[
            of X f' g
        ]
    by simp
lemma split-gallery-fg:
    \llbracketC\inf\vdash\mathcal{C};D\ing\vdash\mathcal{C}; gallery (C#Cs@[D])\rrbracket\Longrightarrow
    \existsAs A B Bs. }A\inf\vdash\mathcal{C}\wedgeB\ing\vdash\mathcal{C}\wedgeC#Cs@[D]=As@A#B#B
    using folding-f.split-gallery flopped-half-chamber-systems-fg by simp
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\mp@subsup{|}{}{\prime}
    where induced-automorphism v\equiv
        if v\inf`}(\cupX)\mathrm{ then }g\mathrm{ v else if vGg'(\X) then fv else v
-f and g}\mathrm{ will both be the identity on the intersection of their images
abbreviation s \equiv induced-automorphism
lemma induced-automorphism-fg-symmetric:
    s = OpposedThinChamberComplexFoldings.s X g f
    by (auto simp add:
        folding-f.vertex-retraction folding-g.vertex-retraction
        induced-automorphism-def
        OpposedThinChamberComplexFoldings.induced-automorphism-def[
            OF fg-symmetric
        ]
        )
```

lemma induced-automorphism-on-simplices-fg: $x \in f \vdash X \Longrightarrow v \in x \Longrightarrow$ s $v=g v$
using induced-automorphism-def by auto
lemma induced-automorphism-eq-foldings-on-chambers-fg:
$C \in f \vdash \mathcal{C} \Longrightarrow f u n-$ eq-on s $g C$
using chamber-system-simplices induced-automorphism-on-simplices-fg[of $C$ ]
by (fast intro: fun-eq-onI)
lemmas indaut-eq-foldch-fg $=$
induced-automorphism-eq-foldings-on-chambers-fg
lemma induced-automorphism-eq-foldings-on-chambers-gf:
$C \in g \vdash \mathcal{C} \Longrightarrow$ fun-eq-on s $f C$
by (auto simp add:
OpposedThinChamberComplexFoldings.indaut-eq-foldch-fg[
OF fg-symmetric
]
induced-automorphism-fg-symmetric
)
lemma induced-automorphism-on-chamber-vertices-f:
chamber $C \Longrightarrow v \in C \Longrightarrow \mathrm{~s} v=($ if $C \in f \vdash \mathcal{C}$ then $g$ v else $f v)$
using chamber-system-def induced-automorphism-eq-foldings-on-chambers-fg
induced-automorphism-eq-foldings-on-chambers-gf
flopped-half-chamber-systems-fg[THEN sym]
fun-eq-on $D[$ of s $g C]$ fun-eq-on $D[$ of s $f C]$
by auto

```
lemma induced-automorphism-simplex-image:
    C\inf\vdash\mathcal{C}\Longrightarrowx\subseteqC\Longrightarrow列}x=\mp@subsup{g}{}{`}xC\ing\vdash\mathcal{C}\Longrightarrowx\subseteqC\Longrightarrow\mp@subsup{\textrm{s}}{}{`}x=\mp@subsup{f}{}{`}
    using fun-eq-on-im[of s g C] fun-eq-on-im[of s f C]
        induced-automorphism-eq-foldings-on-chambers-fg
        induced-automorphism-eq-foldings-on-chambers-gf
    by auto
lemma induced-automorphism-chamber-list-image-fg:
    set Cs \subseteqf\vdash\mathcal{C}\Longrightarrow\textrm{s}=Cs=g\modelsCs
proof (induct Cs)
    case (Cons C Cs) thus ?case
    using induced-automorphism-simplex-image(1)[of C] by simp
qed simp
lemma induced-automorphism-chamber-image-fg:
    chamber C\Longrightarrow s}\mp@subsup{\textrm{s}}{}{`}C=(\mathrm{ if }C\inf\vdash\mathcal{C}\mathrm{ then g}\mp@subsup{g}{}{`}C\mathrm{ else f}\mp@subsup{f}{}{`}C
    using chamber-system-def induced-automorphism-simplex-image
        flopped-half-chamber-systems-fg[THEN sym]
    by auto
lemma induced-automorphism-C0: s'C0 = D0
    using chambers(1,4) basechambers-half-chamber-systems(1)
        induced-automorphism-chamber-image-fg
    by auto
lemma induced-automorphism-fixespointwise-C0-int-D0:
    fixespointwise s (C0\capD0)
    using fun-eq-on-trans[of s g] fun-eq-on-subset[of s g C0]
        fixespointwise-subset[of g DO]
        induced-automorphism-eq-foldings-on-chambers-fg
        basechambers-half-chamber-systems
        folding-g.chamber-retraction1
    by auto
lemmas indaut-fixes-fundfacet =
    induced-automorphism-fixespointwise-C0-int-D0
lemma induced-automorphism-adjacent-half-chamber-system-image-fg:
    \llbracketC\inf\vdash\mathcal{C};D\ing\vdash\mathcal{C};C~D\rrbracket\Longrightarrow \
    using adjacent-half-chamber-system-image-fg[of C D]
        induced-automorphism-simplex-image(2)
    by
        auto
lemmas indaut-adj-halfchsys-im-fg=
    induced-automorphism-adjacent-half-chamber-system-image-fg
lemma induced-automorphism-chamber-map: chamber C chamber (s 'C)
    using induced-automorphism-chamber-image-fg folding-f.chamber-map
```

folding-g.chamber-map
by auto
lemmas indaut-chmap $=$ induced-automorphism-chamber-map
lemma induced-automorphism-ntrivial: $\mathrm{s} \neq$ id
proof
assume $\mathrm{s}: \mathrm{s}=i d$
from chambers(2,3) obtain $v$ where $v: v \notin D 0 C 0=$ insert $v(C 0 \cap D 0)$ using adjacent-int-decomp[of C0 D0] by fast
from chambers(4) s $v(2)$ have $g v: g v=v$ using chamberD-simplex[OF chamber-D0] induced-automorphism-on-simplices-fg[of C0 $v$, THEN sym] by auto
have $g^{`} C 0=C 0$
proof (rule seteqI)
from $v(2) g v$ show $\wedge x . x \in C 0 \Longrightarrow x \in g^{`} C 0$ using $g$-trivial-D0 by force
next
fix $x$ assume $x \in g^{`} C 0$
from this obtain $y$ where $y: y \in C 0 x=g y$ by fast
moreover from $y(1) v(2) g v$ have $g y=y$
using $g$-trivial-D0 [of $y]$ by (cases $y=v$ ) auto
ultimately show $x \in C 0$ using $y$ by simp
qed
with chambers(3) show False by fast
qed
lemma induced-automorphism-bij-between-half-chamber-systems-f:
bij-betw (( $\left.{ }^{\prime}\right)$ s) $(\mathcal{C}-f \vdash \mathcal{C})(f \vdash \mathcal{C})$
using induced-automorphism-simplex-image(2)
flopped-half-chamber-systems-fg
folding-f.opp-chambers-bij bij-betw-cong[of $\left.\mathcal{C}-f \vdash \mathcal{C}\left({ }^{\prime}\right) \mathrm{s}\right]$
by auto
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 $\left(\left({ }^{\prime}\right) \mathrm{s}\right)(\mathcal{C}-g \vdash \mathcal{C})(g \vdash \mathcal{C})$
using induced-automorphism-fg-symmetric
OpposedThinChamberComplexFoldings.indaut-bij-btw-halfchsys-f[ OF fg-symmetric
]
by $\operatorname{simp}$
lemma induced-automorphism-halfmorphism-fopp-to-fimage:
ChamberComplexMorphism folding-f.opp-half-apartment $(f \vdash X) \mathrm{s}$
proof (
rule ChamberComplexMorphism.cong,

```
    rule ThinChamberComplexFolding.morphism-half-apartments,
    rule ThinChamberComplexFolding-f, rule fun-eq-onI
)
    show \v.v\in\bigcupfolding-f.Y\Longrightarrow s v = fv
    using folding-f.opp-half-apartment-def chamber-system-simplices
    by (force simp add:
        flopped-half-chamber-systems-fg
        induced-automorphism-fg-symmetric
        OpposedThinChamberComplexFoldings.induced-automorphism-def[
            OF fg-symmetric
        ]
        )
qed
lemmas indaut-halfmorph-fopp-fim=
    induced-automorphism-halfmorphism-fopp-to-fimage
lemma induced-automorphism-half-chamber-system-gallery-map-f:
    set Cs \subseteqf\vdash\mathcal{C}\Longrightarrowgallery Cs \Longrightarrow gallery ( }\textrm{s}=Cs
    using folding-g.gallery-map[of Cs]
        induced-automorphism-chamber-list-image-fg[THEN sym]
    by
        auto
lemma induced-automorphism-half-chamber-system-pgallery-map-f:
    set Cs \subseteqf\vdash\mathcal{C \Longrightarrow pgallery Cs \Longrightarrow pgallery ( }\textrm{s}==Cs\mathrm{ )}
    using induced-automorphism-half-chamber-system-gallery-map-f pgallery
        flopped-half-chamber-systems-gf pgalleryD-distinct
    folding-g.opp-chambers-distinct-map
    induced-automorphism-chamber-list-image-fg[THEN sym]
    by (auto intro: pgalleryI-gallery)
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 \subseteqg\vdash\mathcal{C}\Longrightarrow pgallery Cs \Longrightarrow pgallery ( }\textrm{s}=C=\mathrm{ )
    using induced-automorphism-fg-symmetric
        OpposedThinChamberComplexFoldings.indaut-halfchsys-pgal-map-f[
        OF fg-symmetric
    ]
    by simp
lemma induced-automorphism-halfmorphism-fimage-to-fopp:
ChamberComplexMorphism \((f \vdash X)\) folding-f.opp-half-apartment s
using OpposedThinChamberComplexFoldings.indaut-halfmorph-fopp-fim[ OF fg-symmetric
]
by (auto simp add:
flopped-half-apartments-gf flopped-half-apartments-fg
```

```
    induced-automorphism-fg-symmetric
    )
lemma induced-automorphism-selfcomp-halfmorphism-f:
    ChamberComplexMorphism (f\vdashX) (f\vdashX) (sos)
    using induced-automorphism-halfmorphism-fimage-to-fopp
    induced-automorphism-halfmorphism-fopp-to-fimage
    by (auto intro: ChamberComplexMorphism.comp)
lemma induced-automorphism-selfcomp-halftrivial-f: fixespointwise (sos) (\bigcup(f\vdashX))
proof (
    rule standard-uniqueness, rule ChamberComplexMorphism.expand-codomain,
    rule induced-automorphism-selfcomp-halfmorphism-f
)
    show ChamberComplexMorphism ( f\vdashX) X id
    using folding-f.chambersubcomplex-image inclusion-morphism by fast
    show SimplicialComplex.maxsimp ( }f\vdashX)\mathrm{ C0
    using chambers(1,4) chamberD-simplex[OF chamber-D0]
            chamber-in-subcomplex[OF folding-f.chambersubcomplex-image, of C0]
    by auto
    show fixespointwise (sos) C0
    proof (rule fixespointwiseI)
    fix v assume v: v\inC0
    with chambers(4) have v\inf`}(\\X
        using chamber-D0 chamberD-simplex by fast
    hence 1:s v=gv using induced-automorphism-def by simp
    show (sos) v=idv
    proof (cases v\inD0)
        case True with v show ?thesis using 1 g-trivial-D0 by simp
    next
        case False
        from v chambers (1,4) have s (gv)=f(gv)
            using chamberD-simplex induced-automorphism-fg-symmetric
                OpposedThinChamberComplexFoldings.induced-automorphism-def[
                        OF fg-symmetric, of g v
                    ]
            by force
        with v False chambers(4) show ?thesis using double-fold-C0 1 by simp
    qed
    qed
next
    fix Cs assume ChamberComplex.min-gallery (f\vdashX) (C0#Cs)
    hence Cs:ChamberComplex.pgallery (f\vdashX) (C0#Cs)
        using ChamberComplex.min-gallery-pgallery folding-f.chambercomplex-image
        by fast
    hence pCs: pgallery (C0#Cs)
    using folding-f.chambersubcomplex-image subcomplex-pgallery by auto
    thus pgallery (id =(C0#Cs)) by simp
    have set-Cs: set (C0#Cs)\subseteqf\vdash\mathcal{C}
```

using Cs pCs folding-f.chambersubcomplex-image
ChamberSubcomplexD-complex ChamberComplex.pgalleryD-chamber
ChamberComplex.chamberD-simplex pgallery-chamber-system
folding-f.chamber-system-image
by fastforce
hence pgallery $(\mathrm{s} \models(C 0 \# C s))$
using $p$ Cs induced-automorphism-half-chamber-system-pgallery-map-f $[$ of $C 0 \# C s]$
by auto
moreover have set $(\mathrm{s} \vDash(C 0 \# C s)) \subseteq g \vdash \mathcal{C}$
proof -
have set $(\mathrm{s} \mid=(C 0 \# C s)) \subseteq \mathrm{s} \vdash(\mathcal{C}-g \vdash \mathcal{C})$
using set-Cs flopped-half-chamber-systems-gf by auto
thus ?thesis
using bij-betw-imp-surj-on [
OF induced-automorphism-bij-between-half-chamber-systems-g
]
by simp
qed
ultimately have pgallery $(\mathrm{s} \vDash(\mathrm{s} \models(C 0 \# C s)))$
using induced-automorphism-half-chamber-system-pgallery-map-g[ of $\mathrm{s}=(C 0 \# C s)$
]
by auto
thus pgallery $((\mathrm{sos}) \mid=(C 0 \# C s))$
using ssubst $[$ OF setlistmapim-comp, of pgallery, of s s CO\#Cs] by fast
qed (unfold-locales, rule folding-f.chambersubcomplex-image)
lemmas indaut-selfcomp-halftriv- $f=$
induced-automorphism-selfcomp-halftrivial-f
lemma induced-automorphism-selfcomp-halftrivial-g: fixespointwise (sos) $(\bigcup(g \vdash X))$
using induced-automorphism-fg-symmetric
OpposedThinChamberComplexFoldings.indaut-selfcomp-halftriv- $f[$
OF fg-symmetric
]
by $\operatorname{simp}$
lemma induced-automorphism-trivial-outside:
assumes $v \notin \bigcup X$
shows $\mathrm{s} v=v$
proof-
from assms have $v \notin f^{\prime}(\bigcup X) \wedge v \notin g^{\prime}(\bigcup X)$ using vertex-set-split by fast
thus $\mathrm{s} v=v$ using induced-automorphism-def by simp
qed
lemma induced-automorphism-morphism: ChamberComplexEndomorphism X s
proof (unfold-locales, rule induced-automorphism-chamber-map, simp)
fix $C$ assume chamber $C$
thus card $\left(\mathrm{s}^{`} C\right)=\operatorname{card} C$

```
    using induced-automorphism-chamber-image-fg folding-f.dim-map
        folding-g.dim-map
        flopped-half-chamber-systems-fg[THEN sym]
    by (cases \(C \in f \vdash \mathcal{C})\) auto
qed (rule induced-automorphism-trivial-outside)
lemmas indaut-morph \(=\) induced-automorphism-morphism
lemma induced-automorphism-morphism-order2: \(\operatorname{sos}=i d\)
proof
    fix \(v\)
    show (sos) \(v=i d v\)
    proof (cases \(v \in f^{\prime}(\bigcup X) v \in g^{\prime}(\bigcup X)\) rule: two-cases)
        case both
        from both(1) show ?thesis
            using induced-automorphism-selfcomp-halftrivial-f fixespointwiseD[of sos]
            by auto
    next
        case one thus ?thesis
            using induced-automorphism-selfcomp-halftrivial-f fixespointwiseD[of sos]
            by fastforce
    next
        case other thus ?thesis
            using induced-automorphism-selfcomp-halftrivial-g fixespointwiseD[of sos]
            by fastforce
    qed (simp add: induced-automorphism-def)
qed
lemmas indaut-order2 \(=\) induced-automorphism-morphism-order2
lemmas induced-automorphism-bij =
    \(o-b i j[O F\)
        induced-automorphism-morphism-order2
        induced-automorphism-morphism-order2
    ]
lemma induced-automorphism-surj-on-vertexset: s' \((\bigcup X)=\bigcup X\)
proof
    show s' \((\bigcup X) \subseteq \bigcup X\)
        using induced-automorphism-morphism
            ChamberComplexEndomorphism.vertex-map
        by fast
    hence (sos) \({ }^{‘}(\bigcup X) \subseteq \mathrm{s}^{`}(\bigcup X)\) by fastforce
    thus \(\bigcup X \subseteq \mathrm{~s}^{`}(\bigcup X)\) using induced-automorphism-morphism-order2 by simp
qed
lemma induced-automorphism-bij-betw-vertexset: bij-betw s ( \(\bigcup X)(\bigcup X)\)
    using induced-automorphism-bij induced-automorphism-surj-on-vertexset
    by (auto intro: bij-betw-subset)
```

```
lemma induced-automorphism-surj-on-simplices: s}\vdashX=
proof
    show s\vdash
        using induced-automorphism-morphism
                ChamberComplexEndomorphism.simplex-map
    by fast
    hence s}\vdash(\textrm{s}\vdashX)\subseteq\textrm{s}\vdashXX\mathrm{ by auto
    thus X\subseteq s\vdashX
        by (simp add:
            setsetmapim-comp[THEN sym] induced-automorphism-morphism-order2
        )
qed
lemma induced-automorphism-automorphism:
    ChamberComplexAutomorphism X s
    using induced-automorphism-chamber-map
        ChamberComplexEndomorphism.dim-map
        induced-automorphism-morphism
        induced-automorphism-bij-betw-vertexset
        induced-automorphism-surj-on-simplices
        induced-automorphism-trivial-outside
    by (intro-locales, unfold-locales, fast)
lemmas indaut-aut = induced-automorphism-automorphism
lemma induced-automorphism-unique-automorphism':
    assumes ChamberComplexAutomorphism X s s\not=id fixespointwise s (C0\capD0)
    shows fun-eq-on s s C0
proof (rule fun-eq-on-subset-and-diff-imp-eq-on)
    from assms(3) show fun-eq-on s s (C0\capD0)
        using induced-automorphism-fixespointwise-C0-int-D0
            fixespointwise2-imp-eq-on
    by fast
show fun-eq-on s s (C0 - (C0\capD0))
proof (rule fun-eq-onI)
    fix v}\mathrm{ assume v:v }\inCO-C0\capD
    with chambers(2) have C0-insert: C0 = insert v (C0\capD0)
        using adjacent-conv-insert by fast
    hence s}\mp@subsup{s}{}{`}C0= insert (sv)(\mp@subsup{s}{}{`}(C0\capD0)) \mp@subsup{\textrm{s}}{}{`}C0= insert (s v) (s`(C0\capD0)
        by auto
    with assms(3)
        have insert: s`C0 = insert (s v) (C0\capD0) D0 = insert (s v) (C0\capD0)
        using basechambers-half-chamber-systems
                            induced-automorphism-fixespointwise-C0-int-D0
                    induced-automorphism-simplex-image(1)
        by (auto simp add: fixespointwise-im)
    from chambers(2,3) have C0D0-C0: (C0\capD0) }\triangleleftC
```

using adjacent-int-facet1 by fast
with assms(1) chambers (1) have $s^{`}(C 0 \cap D 0) \triangleleft s^{`} C 0$
using ChamberComplexAutomorphism.facet-map by fast
with $\operatorname{assms}(3)$ have $C 0 D 0-s C 0:(C 0 \cap D 0) \triangleleft s^{`} C 0$
by (simp add: fixespointwise-im)
hence sv-nin-COD0: s v $\neq C 0 \cap D 0$ using insert(1) facetrel-psubset by auto
from $\operatorname{assms}(1)$ chambers(1) have chamber ( $s^{`} \mathrm{C} 0$ )
using ChamberComplexAutomorphism.chamber-map by fast
moreover from chambers $(2,3)$ have C0D0-D0: $(C 0 \cap D 0) \triangleleft D 0$
using adjacent-sym adjacent-int-facet1 by (fastforce simp add: Int-commute)
ultimately have $s^{`} C O=C O \vee s^{`} C 0=D 0$
using chambers $(1,3)$ chamber-DO CODO-CO CODO-sCO
facet-unique-other-chamber[of s'C0 C0 $\cap D 0$ C0 D0]
by auto
moreover have $\neg s^{`} C 0=C 0$
proof
assume $s C 0: s^{`} C 0=C 0$
have $s=i d$
proof (
rule standard-uniqueness-automorphs, rule assms(1),
rule trivial-automorphism, rule chambers(1),
rule fixespointwise-subset-and-diff-imp-eq-on, rule Int-lower 1, rule assms(3), rule fixespointwiseI
)
fix $a$ assume $a \in C 0-(C 0 \cap D 0)$
with $v$ have $a=v$ using C0-insert by fast
with $s C 0$ show s $a=i d$ asing CO-insert sv-nin-CODO by auto
qed
with $\operatorname{assms}(1,2)$ show False by fast
qed
ultimately have $s C 0-D 0: s^{`} C 0=D 0$ by fast
have s $v \notin C 0 \cap D 0$ using insert(2) C0D0-D0 facetrel-psubset by force
thus $s v=\mathrm{s} v$ using insert sCO-D0 sv-nin-C0D0 by auto
qed
qed $\operatorname{simp}$
lemma induced-automorphism-unique-automorphism:
【 ChamberComplexAutomorphism $X$ s; sfid; fixespointwise $s(C 0 \cap D 0) \rrbracket$ $\Longrightarrow s=\mathrm{s}$
using chambers(1) induced-automorphism-unique-automorphism' standard-uniqueness-automorphs induced-automorphism-automorphism
by fastforce
lemmas indaut-uniq-aut $=$
induced-automorphism-unique-automorphism
lemma induced-automorphism-unique:

```
    OpposedThinChamberComplexFoldings X f' g}\mp@subsup{g}{}{\prime}\textrm{CO}\Longrightarrow\mp@subsup{g}{}{\prime}C0=\mp@subsup{g}{}{`}\textrm{C}0
    OpposedThinChamberComplexFoldings.induced-automorphism X f' g}\mp@subsup{g}{}{\prime}=\textrm{s
    using induced-automorphism-automorphism induced-automorphism-ntrivial
        induced-automorphism-fixespointwise-C0-int-D0
    by (auto intro:
            OpposedThinChamberComplexFoldings.indaut-uniq-aut[
                THEN sym
        ]
        )
lemma induced-automorphism-sym:
    OpposedThinChamberComplexFoldings.induced-automorphism X gf = s
    using OpposedThinChamberComplexFoldings.indaut-aut[
            OF fg-symmetric
        ]
        OpposedThinChamberComplexFoldings.induced-automorphism-ntrivial[
            OF fg-symmetric
        ]
        OpposedThinChamberComplexFoldings.indaut-fixes-fundfacet[
            OF fg-symmetric
        ]
        induced-automorphism-unique-automorphism
    by (simp add: chambers(4) Int-commute)
lemma induced-automorphism-respects-labels:
    assumes label-wrt B \varphi v\in(\bigcupX)
    shows \varphi(s v)=\varphiv
proof-
    from assms(2) obtain C where chamber C v\inC using simplex-in-max by fast
    with assms show ?thesis
        by (simp add:
                induced-automorphism-on-chamber-vertices-f folding-f.respects-labels
            folding-g.respects-labels
        )
qed
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 ob-
tain information about minimality of galleries by considering the walls they cross.
context ThinChamberComplex
begin
definition foldpairs :: $\left(\left(^{\prime} a \Rightarrow^{\prime} a\right) \times\left({ }^{\prime} a \Rightarrow^{\prime} a\right)\right)$ set
where foldpairs $\equiv\{(f, g) . \exists C$. OpposedThinChamberComplexFoldings XfgC\}
abbreviation walls $\equiv \bigcup(f, g) \in$ foldpairs. $\{\{f \vdash \mathcal{C}, g \vdash \mathcal{C}\}\}$
abbreviation the-wall-betw $C D \equiv$ THE-default $\}(\lambda H . H \in$ walls $\wedge$ separated-by $H C D)$
definition walls-betw :: ' $a$ set $\Rightarrow$ 'a set $\Rightarrow$ 'a set set set set where walls-betw $C D \equiv\{H \in$ 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 $(B \# C \# C s)=$ the-wall-betw $B C \#$ wall-crossings $(C \# C s)$
lemma foldpairs-sym: $(f, g) \in$ foldpairs $\Longrightarrow(g, f) \in$ foldpairs
using foldpairs-def OpposedThinChamberComplexFoldings.fg-symmetric by fastforce
lemma not-self-separated-by-wall: $H \in$ walls $\Longrightarrow \neg$ separated-by $H$ C C
using foldpairs-def OpposedThinChamberComplexFoldings.halfchsys-decomp(2) not-self-separated-by-disjoint
by force
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 -
from assms have 1: $\exists!H^{\prime} \in$ walls. separated-by $H^{\prime} C D$
using THE-default-none[of $\lambda H$. H walls $\wedge$ separated-by $H C D\}]$ by fast
show the-wall-betw $C D \in$ walls separated-by (the-wall-betw $C D$ ) $C D$
using THE-defaultI ${ }^{\prime}[$ OF 1] by auto
qed
lemma the-wall-betw-self-empty: the-wall-betw $C C=\{ \}$
proof -
\{
assume $*$ : the-wall-betw $C C \neq\{ \}$
then obtain $f g$
where $(f, g) \in$ foldpairs the-wall-betw $C C=\{f \vdash \mathcal{C}, g \vdash \mathcal{C}\}$
using the-wall-betw-nempty(1)[of C C]
by blast
with * have False
using the-wall-betw-nempty(2)[of C C] foldpairs-def

```
                OpposedThinChamberComplexFoldings.halfchsys-decomp(2)[
                of X
            ]
                not-self-separated-by-disjoint[of f\vdash\mathcal{C}g\vdash\mathcal{C}]
            by
                auto
}
thus ?thesis by fast
qed
lemma length-wall-crossings: length (wall-crossings Cs) = length Cs - 1
    by (induct Cs rule: list-induct-CCons) auto
lemma wall-crossings-snoc:
    wall-crossings (Cs@[D,E])=wall-crossings (Cs@[D])@ [the-wall-betw D E]
    by (induct Cs rule: list-induct-CCons) auto
lemma wall-crossings-are-walls:
    H\inset (wall-crossings Cs)\LongrightarrowH\not={}\LongrightarrowH\inwalls
proof (induct Cs arbitrary:H rule: list-induct-CCons)
    case (CCons B C Cs) thus ?case
        using the-wall-betw-nempty(1)
        by (cases H\inset (wall-crossings (C#Cs))) auto
qed auto
lemma in-set-wall-crossings-decomp:
    H\inset (wall-crossings Cs)\Longrightarrow
        \exists As A B Bs.Cs=As@[A,B]@Bs ^H= the-wall-betw A B
proof (induct Cs rule: list-induct-CCons)
    case (CCons C D Ds)
    show ?case
    proof (cases H set (wall-crossings (D#Ds)))
        case True
        with CCons(1) obtain As A B Bs
            where C#(D#Ds)=(C#As)@[A,B]@Bs H= the-wall-betw A B
            by fastforce
        thus ?thesis by fast
    next
        case False
        with CCons(2) have C#(D#Ds)=[]@[C,D]@Ds H = the-wall-betw C D
            by auto
    thus ?thesis by fast
    qed
qed auto
end
context OpposedThinChamberComplexFoldings
begin
```

```
lemma foldpair: (f,g)\infoldpairs
    unfolding foldpairs-def
proof
    have OpposedThinChamberComplexFoldings X f g C0 ..
    thus }(f,g)\in{(f,g)
        \exists}\mathrm{ . OpposedThinChamberComplexFoldings X f g C}
        by fast
qed
lemma separated-by-this-wall-fg:
    separated-by {f\vdash\mathcal{C},g\vdash\mathcal{C}}CD\LongrightarrowC\inf\vdash\mathcal{C}\LongrightarrowD\ing\vdash\mathcal{C}
    using separated-by-disjoint[
            OF - half-chamber-system-disjoint-union(2), of C D
        ]
    by fast
lemmas separated-by-this-wall-gf =
    OpposedThinChamberComplexFoldings.separated-by-this-wall-fg[
        OF fg-symmetric
    ]
lemma induced-automorphism-this-wall-vertex:
    assumes }C\inf\vdash\mathcal{C}\quadD\ing\vdash\mathcal{C}v\inC\cap
    shows s v=v
proof-
    from assms have s v=gv
        using chamber-system-simplices induced-automorphism-on-simplices-fg
        by auto
    with assms(2,3) show s v=v
            using chamber-system-simplices folding-g.retraction by auto
qed
lemmas indaut-wallvertex =
    induced-automorphism-this-wall-vertex
lemma unique-wall:
    assumes opp' : OpposedThinChamberComplexFoldings X f' g}\mp@subsup{g}{}{\prime}\mp@subsup{C}{}{\prime
    and chambers: }A\inf\vdash\mathcal{C}A\in\mp@subsup{f}{}{\prime}\vdash\mathcal{C}\quadB\ing\vdash\mathcal{C}B\in\mp@subsup{g}{}{\prime}\vdash\mathcal{C}\quadA~
    shows }{f\vdash\mathcal{C},g\vdash\mathcal{C}}={f\downarrow\vdash\mathcal{C},g\vdash\vdash\mathcal{C}
proof-
    from chambers have B: B=\mp@subsup{g}{}{\prime}A B=\mp@subsup{g}{}{\prime}}
            using adjacent-sym[of A B] adjacent-half-chamber-system-image-gf
                OpposedThinChamberComplexFoldings.adjhalfchsys-image-gf[
                        OF opp'
                ]
    by auto
with chambers(1,2,5)
    have A: OpposedThinChamberComplexFoldings X fg A
    and A': OpposedThinChamberComplexFoldings X f' g' A
```

using switch-basechamber[of A]
OpposedThinChamberComplexFoldings.switch-basechamber [ OF opp ${ }^{\prime}$, of $A$
]
by
auto
with $B$ show ?thesis
using OpposedThinChamberComplexFoldings.unique-half-chamber-system-f $[$ OF $A A^{\prime}$
] OpposedThinChamberComplexFoldings.unique-half-chamber-system-g[ OF $A A^{\prime}$
]
by auto
qed
end
context ThinChamberComplex
begin
lemma separated-by-wall-ex-foldpair:
assumes Hewalls 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-
from assms(1) obtain $f g$ where $f g:(f, g) \in$ foldpairs $H=\{f \vdash \mathcal{C}, g \vdash \mathcal{C}\}$ by auto show ?thesis
proof (cases $C \in f \vdash \mathcal{C}$ )
case True
moreover with $f g \operatorname{assms}(2)$ have $D \in g \vdash \mathcal{C}$
using foldpairs-def
OpposedThinChamberComplexFoldings.separated-by-this-wall-fg[ of $X f g-C D$
]
by auto
ultimately show ?thesis using $f g$ by auto
next
case False with $\operatorname{assms}$ (2) fg show ?thesis
using foldpairs-sym $[$ of $f g$ ] separated-by-in-other $[$ of $f \vdash \mathcal{C} g \vdash \mathcal{C} C D]$ by auto
qed
qed
lemma not-separated-by-wall-ex-foldpair:
assumes chambers: chamber $C$ chamber $D$
and wall : HEwalls $\neg$ 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-
from wall(1) obtain $f g$ where $f g:(f, g) \in$ foldpairs $H=\{f \vdash \mathcal{C}, g \vdash \mathcal{C}\}$ by auto
from $f g(1)$ obtain $A$ where A: OpposedThinChamberComplexFoldings Xfg A using foldpairs-def by fast

```
    from chambers have chambers': C\inf\vdash\mathcal{C}\vee \ C\ing\vdash\mathcal{C}D\inf\vdash\mathcal{C}\vee D\ing\vdash\mathcal{C}
    using chamber-system-def
        OpposedThinChamberComplexFoldings.halfchsys-decomp(1)[
        OF A
        ]
    by auto
show ?thesis
proof (cases C }\inf\vdash\mathcal{C}\mathrm{ )
    case True
    moreover with chambers'(2) fg(2) wall(2) have D\inf\vdash\mathcal{C}
        unfolding separated-by-def by auto
    ultimately show ?thesis using fg by auto
next
    case False
    with chambers'(1) have C\ing\vdash\mathcal{C}}\mathrm{ by simp
    moreover with chambers'(2) fg(2) wall(2) have D\ing\vdash\mathcal{C}
        using insert-commute[of f\vdash\mathcal{C}g\vdash\mathcal{C}{}] unfolding separated-by-def by auto
    ultimately show ?thesis using fg foldpairs-sym[of f g] by auto
    qed
qed
lemma adj-wall-imp-ex1-wall:
    assumes adj : C~D
    and wall: H0\inwalls separated-by H0 C D
    shows }\exists\mathrm{ !HEwalls. separated-by H C D
proof (rule ex1I, rule conjI, rule wall(1), rule wall(2))
    fix H}\mathrm{ assume H:HEwalls ^ separated-by H C D
    from this obtain fg
        where fg:(f,g)\infoldpairs }H={f\vdash\mathcal{C},g\vdash\mathcal{C}}\quadC\inf\vdash\mathcal{C}D\ing\vdash\mathcal{C
        using separated-by-wall-ex-foldpair[of H C D]
        by auto
    from wall obtain f0 g0
        where f0g0: (f0,g0)\infoldpairs H0={f0\vdash\mathcal{C},g0\vdash\mathcal{C}} C\inf0\vdash\mathcal{C}D\ing0\vdash\mathcal{C}
        using separated-by-wall-ex-foldpair[of H0 C D]
        by auto
    from fg(1) f0gO(1) obtain A AO
    where A: OpposedThinChamberComplexFoldings X f g A
    and A0: OpposedThinChamberComplexFoldings X f0 g0 A0
    using foldpairs-def
    by auto
    from fg(2-4) f0gO(2-4) adj show H=H0
    using OpposedThinChamberComplexFoldings.unique-wall[OF A0 A] by auto
qed
end
context OpposedThinChamberComplexFoldings
begin
```

```
lemma this-wall-betwI:
    assumes C\inf\vdash\mathcal{C}D\ing\vdash\mathcal{C}C~D
    shows the-wall-betw C D ={f\vdash\mathcal{C},g\vdash\mathcal{C}}
proof (rule THE-default1-equality, rule adj-wall-imp-ex1-wall)
    have OpposedThinChamberComplexFoldings X fg C0 ..
    thus {f\vdash\mathcal{C},g\vdash\mathcal{C}}\in\mathrm{ walls using foldpairs-def by auto}
    moreover from assms(1,\mathcal{L}) show separated-by {f\vdash\mathcal{C},g\vdash\mathcal{C}}CD
        by (auto intro: separated-byI)
    ultimately show {f\vdash\mathcal{C},g\vdash\mathcal{C}}\in\mathrm{ walls }\wedge\mathrm{ separated-by {ff-C},g\vdash\mathcal{C}}CD by simp
qed (rule assms(3))
lemma this-wall-betw-basechambers:
    the-wall-betw C0 D0 = {f\vdash\mathcal{C},g\vdash\mathcal{C}}
    using basechambers-half-chamber-systems chambers(2) this-wall-betwI by auto
lemma this-wall-in-crossingsI-fg:
    defines }H:H\equiv{f\vdash\mathcal{C},g\vdash\mathcal{C}
    assumes D: D\ing\vdash\mathcal{C}
    shows }C\inf\vdash\mathcal{C}\Longrightarrow\mathrm{ gallery (C#Cs@[D]) ఋH set (wall-crossings (C#Cs@[D]))
proof (induct Cs arbitrary: C)
    case Nil
    from Nil(1) assms have H\inwalls separated-by H C D
        using foldpair by (auto intro: separated-byI)
    thus ?case
        using galleryD-adj[OF Nil(2)]
                THE-default1-equality[OF adj-wall-imp-ex1-wall]
    by auto
next
    case (Cons B Bs)
    show ?case
    proof (cases B\inf\vdash\mathcal{C)}
        case True with Cons(1,3) show ?thesis using gallery-Cons-reduce by simp
    next
        case False
        with Cons(2,3) H have H\inwalls separated-by H C B
            using galleryD-chamber[OF Cons(3)] chamber-in-other-half-fg[of B] foldpair
            by (auto intro: separated-byI)
        thus ?thesis
            using galleryD-adj[OF Cons(3)]
                THE-default1-equality[OF adj-wall-imp-ex1-wall]
            by auto
    qed
qed
end
lemma (in ThinChamberComplex) walls-betw-subset-wall-crossings:
    assumes gallery (C#Cs@[D])
    shows walls-betw C D\subseteq set (wall-crossings (C#Cs@[D]))
```

```
proof
    fix H assume H\in walls-betw C D
    hence H:H\inwalls separated-by H C D using walls-betw-def by auto
    from this obtain fg
        where fg:(f,g)\infoldpairs }H={f\vdash\mathcal{C},g\vdash\mathcal{C}}\quadC\inf\vdash\mathcal{C}D\ing\vdash\mathcal{C
        using separated-by-wall-ex-foldpair[of H C D]
        by auto
    from fg(1) obtain Z where Z: OpposedThinChamberComplexFoldings Xfg Z
        using foldpairs-def by fast
    from assms H(2) fg(2-4) show H\in set (wall-crossings (C#Cs@[D]))
        using OpposedThinChamberComplexFoldings.this-wall-in-crossingsI-fg[OF Z]
        by auto
qed
context OpposedThinChamberComplexFoldings
begin
lemma same-side-this-wall-wall-crossings-not-distinct-f:
    gallery (C#Cs@[D])\LongrightarrowC\inf\vdash\mathcal{C}\LongrightarrowD\inf\vdash\mathcal{C}\Longrightarrow
        {f\vdash\mathcal{C},g\vdash\mathcal{C}}\in\mathrm{ set (wall-crossings (C#Cs@[D])) ב}
        \neg distinct (wall-crossings (C#Cs@[D]))
proof (induct Cs arbitrary: C)
    case Nil
```



```
    moreover hence the-wall-betw C D\not={} by fast
    ultimately show ?case
        using Nil(2,3) the-wall-betw-nempty(2) separated-by-this-wall-fg[of C D]
                half-chamber-system-disjoint-union(2)
        by auto
next
    case (Cons E Es)
    show ?case
    proof
        assume 1: distinct (wall-crossings (C # (E # Es)@ [D]))
        show False
        proof (
            cases E\inf\vdash\mathcal{C}{f\vdash\mathcal{C},g\vdash\mathcal{C}}\in\mathrm{ set (wall-crossings (E#Es@[D]))}
            rule: two-cases
        )
            case both with Cons(1,2,4) 1 show False
                using gallery-Cons-reduce by simp
        next
            case one
            from one(2) Cons(5) have {f\vdash\mathcal{C},g\vdash\mathcal{C}}= the-wall-betw C E by simp
            moreover hence the-wall-betw C E\not={} by fast
            ultimately show False
                using Cons(3) one(1) the-wall-betw-nempty(2)
                    separated-by-this-wall-fg[of C E]
                    half-chamber-system-disjoint-union(2)
```

```
        by auto
    next
        case other with Cons(3) show False
            using 1 galleryD-chamber[OF Cons(2)] galleryD-adj[OF Cons(2)]
                chamber-in-other-half-fg this-wall-betwI
            by force
    next
        case neither
        from Cons(2) neither(1) have E\ing\vdash\mathcal{C}
            using galleryD-chamber chamber-in-other-half-fg by auto
        with Cons(4) have separated-by {g\vdash\mathcal{C},f\vdash\mathcal{C}} ED
            by (blast intro: separated-byI)
```



```
            using foldpair walls-betw-def by (auto simp add: insert-commute)
        with neither(2) show False
            using gallery-Cons-reduce[OF Cons(2)] walls-betw-subset-wall-crossings
            by auto
    qed
    qed
qed
lemmas sside-wcrossings-ndistinct-f =
    same-side-this-wall-wall-crossings-not-distinct-f
lemma separated-by-this-wall-chain3-fg:
    assumes B\inf\vdash\mathcal{C}\mathrm{ chamber C chamber D}
        separated-by {f\vdash\mathcal{C},g\vdash\mathcal{C}} B C separated-by {f\vdash\mathcal{C},g\vdash\mathcal{C}} C D
    shows }\quadC\ing\vdash\mathcal{C}D\inf\vdash\mathcal{C
    using assms separated-by-this-wall-fg separated-by-this-wall-gf
    by (auto simp add: insert-commute)
lemmas sepwall-chain3-fg=
    separated-by-this-wall-chain3-fg
end
context ThinChamberComplex
begin
lemma wall-crossings-min-gallery-betwI:
    assumes gallery (C#Cs@[D])
        distinct (wall-crossings (C#Cs@[D]))
        \forallH\inset (wall-crossings (C#Cs@[D])). separated-by H C D
    shows min-gallery (C#Cs@[D])
proof (rule min-galleryI-betw)
    obtain B Bs where BBs:Cs@[D]=B#Bs using snoc-conv-cons by fast
    define }H\mathrm{ where }H=\mathrm{ the-wall-betw }C
    with BBs assms(3) have 1: separated-by H C D by simp
    show C\not=D
```

```
    proof (cases H={})
    case True thus ?thesis
        using 1 unfolding separated-by-def by simp
    next
    case False
    with H-def have H\in walls using the-wall-betw-nempty(1) by simp
    from this obtain fg
        where fg:(f,g)\infoldpairs }H={f\vdash\mathcal{C},g\vdash\mathcal{C}}\quadC\inf\vdash\mathcal{C}D\ing\vdash\mathcal{C
        using 1 separated-by-wall-ex-foldpair[of H C D]
        by auto
    thus ?thesis
        using foldpairs-def
                OpposedThinChamberComplexFoldings.halfchsys-decomp(2)[
                    of Xfg
                ]
        by auto
    qed
next
    fix Ds assume Ds: gallery (C # Ds @ [D])
    have Suc (length Cs) = card (walls-betw C D)
    proof-
        from assms(1,3) have set (wall-crossings (C#Cs@[D])) = walls-betw C D
            using separated-by-not-empty wall-crossings-are-walls[of - C#Cs@[D]]
                    walls-betw-def
                    walls-betw-subset-wall-crossings[OF assms(1)]
            unfolding separated-by-def
            by auto
        with assms(2) show ?thesis
            using distinct-card[THEN sym] length-wall-crossings by fastforce
    qed
    moreover have card (walls-betw C D) \leq Suc (length Ds)
    proof-
    from Ds have card (walls-betw C D) \leqcard (set (wall-crossings (C#Ds@[D])))
        using walls-betw-subset-wall-crossings finite-set card-mono by force
    also have .. . \leqlength (wall-crossings (C#Ds@[D]))
            using card-length by auto
    finally show ?thesis using length-wall-crossings by simp
qed
    ultimately show length Cs \leq length Ds by simp
qed (rule assms(1))
lemma ex-nonseparating-wall-imp-wall-crossings-not-distinct:
    assumes gal : gallery (C#Cs@[D])
    and wall: H\inset (wall-crossings (C#Cs@[D])) H\not={}
                                \neg separated-by H C D
    shows \neg distinct (wall-crossings (C#Cs@[D]))
proof-
    from assms obtain fg
        where fg:(f,g)\in\mathrm{ foldpairs }H={f\vdash\mathcal{C},g\vdash\mathcal{C}}\quadC\inf\vdash\mathcal{C}D\inf\vdash\mathcal{C}
```

```
    using wall-crossings-are-walls[of H]
        not-separated-by-wall-ex-foldpair[of C D H]
        galleryD-chamber
    by auto
    from fg(1) obtain Z where Z: OpposedThinChamberComplexFoldings X f g Z
    using foldpairs-def by fast
    from wall fg(2-4) show ?thesis
    using OpposedThinChamberComplexFoldings.sside-wcrossings-ndistinct-f [
                OF Z gal
            ]
    by blast
qed
lemma not-min-gallery-double-crosses-wall:
    assumes gallery Cs ᄀ min-gallery Cs {} & set (wall-crossings Cs)
    shows \neg distinct (wall-crossings Cs)
proof (cases Cs rule: list-cases-Cons-snoc)
    case Nil with assms(2) show ?thesis by simp
next
    case Single with assms(1,2) show ?thesis using galleryD-chamber by simp
next
    case (Cons-snoc B Bs C)
    show ?thesis
    proof (cases B=C)
        case True show ?thesis
        proof (cases Bs)
            case Nil with True Cons-snoc assms(3) show ?thesis
                using the-wall-betw-self-empty by simp
        next
            case (Cons E Es)
            define H where H}=\mathrm{ the-wall-betw }B
            with Cons have *:H\in set (wall-crossings (B#Bs@[C])) by simp
            moreover from assms(3) Cons-snoc * have H}\not={}\mathrm{ by fast
            ultimately show ?thesis
                using assms(1) Cons-snoc Cons True H-def
                    the-wall-betw-nempty(1)[of B E] not-self-separated-by-wall[of H B]
                    ex-nonseparating-wall-imp-wall-crossings-not-distinct[of B Bs C H]
                by fast
        qed
    next
        case False
        with assms Cons-snoc
            have 1: ᄀ distinct (wall-crossings Cs) V
                        \neg(\forallH\inset (wall-crossings Cs). separated-by H B C)
            using wall-crossings-min-gallery-betwI
            by force
        moreover {
            assume }\neg(\forallH\in\mathrm{ set (wall-crossings Cs). separated-by H B C)
            from this obtain H
```

```
            where H:H\inset (wall-crossings Cs) ᄀ separated-by H B C
            by auto
        moreover from H(1) assms(3) have H\not={} by fast
        ultimately have ?thesis
            using assms(1) Cons-snoc
                ex-nonseparating-wall-imp-wall-crossings-not-distinct
            by simp
    }
    ultimately show ?thesis by fast
    qed
qed
lemma not-distinct-crossings-split-gallery:
    \llbracketgallery Cs; {} & set (wall-crossings Cs); \neg distinct (wall-crossings Cs)\rrbracket\Longrightarrow
        \existsfg As A B Bs E F Fs.
            (f,g)\infoldpairs ^ A\inf\vdash\mathcal{C}\wedgeB\ing\vdash\mathcal{C}\wedgeE\ing\vdash\mathcal{C}\wedgeF\inf\vdash\mathcal{C}\wedge
            ( Cs=As@[A,B,F]@Fs\veeCs=As@[A,B]@Bs@[E,F]@Fs )
proof (induct Cs rule: list-induct-CCons)
    case (CCons C J Js)
    show ?case
    proof (cases distinct (wall-crossings (J#Js)))
        case False
        moreover from CCons(2) have gallery (J#Js)
            using gallery-Cons-reduce by simp
        moreover from CCons(3) have {} & set (wall-crossings (J#Js)) by simp
        ultimately obtain fg As A B Bs E F Fs where split:
            (f,g)\infoldpairs }A\inf\vdash\mathcal{C}B\ing\vdash\mathcal{C}\quadE\ing\vdash\mathcal{C}\quadF\inf\vdash\mathcal{C
            J#Js=As@[A,B,F]@Fs\vee J#Js=As@[A,B]@Bs@[E,F]@Fs
            using CCons(1)
            by blast
        from split(6)
            have C#J#Js = (C#As)@[A,B,F]@Fs \vee
                        C#J#Js=(C#As)@[A,B]@Bs@[E,F]@Fs
            by simp
        with split(1-5) show ?thesis by blast
    next
        case True
        define H}\mathrm{ where }H=\mathrm{ the-wall-betw C J
    with True CCons(4) have H\inset (wall-crossings (J#Js)) by simp
    from this obtain Bs E F Fs
            where split1: J#Js = Bs@[E,F]@Fs H= the-wall-betw E F
            using in-set-wall-crossings-decomp
            by fast
    from H-def split1(2) CCons(3)
            have Hwall:H \in walls separated-by H C J separated-by H E F
            using the-wall-betw-nempty[of C J] the-wall-betw-nempty[of E F]
            by auto
        from Hwall(1,2) obtain fg
            where fg:(f,g)\infoldpairs }H={f\vdash\mathcal{C},g\vdash\mathcal{C}}\quadC\inf\vdash\mathcal{C}\quadJ\ing\vdash\mathcal{C
```

```
using separated-by-wall-ex-foldpair[of H C J]
by auto
from fg(1) obtain Z
    where Z: OpposedThinChamberComplexFoldings X f g Z
    using foldpairs-def
    by fast
show ?thesis
proof (cases Bs)
    case Nil
    with CCons(2) Hwall(2,3) fg(2-4) split1(1)
    have F\inf\vdash\mathcal{C C#J#Js = []@[C,J,F]@Fs}\\mp@code{F}|
    using galleryD-chamber
                OpposedThinChamberComplexFoldings.sepwall-chain3-fg(2)[
                    OF Z, of C J F
        ]
    by auto
    with fg(1,3,4) show ?thesis by blast
next
    case (Cons K Ks) have Bs: Bs=K#Ks by fact
    show ?thesis
    proof (cases E\inf\vdash\mathcal{C})
    case True
    from CCons(2) split1(1) Bs have gallery (J#Ks@[E])
        using gallery-Cons-reduce[of C J#Ks@E#F#Fs]
                gallery-append-reduce1[of J#Ks@[E]F#Fs]
        by simp
    with fg(4) True obtain Ls L M Ms
        where LsLMMs: L\ing\vdash\mathcal{C}M\inf\vdash\mathcal{C}J#Ks@[E]=Ls@L#M#Ms
        using OpposedThinChamberComplexFoldings.split-gallery-gf[
                OF Z, of J E Ks
                ]
            by blast
    show ?thesis
    proof (cases Ls)
            case Nil
            with split1(1) Bs LsLMMs(3)
                have C#J#Js = []@[C,J,M]@(Ms@F#Fs)
            by simp
            with fg(1,3,4) LsLMMs(2) show ?thesis by blast
    next
        case (Cons N Ns)
        with split1(1) Bs LsLMMs(3)
            have C#J#Js = []@[C,J]@Ns@[L,M]@(Ms@F#Fs)
            by simp
            with fg(1,3,4) LsLMMs(1,2) show ?thesis by blast
    qed
next
    case False
    with Hwall(2,3) fg(2) split1(1) Cons
```

```
                have E\ing\vdash\mathcal{C}F\inf\vdash\mathcal{C}C#J#Js=[]@[C,J]@Ks@[E,F]@Fs
                    using OpposedThinChamberComplexFoldings.separated-by-this-wall-fg[
                    OF Z
                    ]
                        separated-by-in-other[of f\vdash\mathcal{C}g\vdash\mathcal{C}]
                by auto
                with fg(1,3,4) show ?thesis by blast
            qed
        qed
    qed
qed auto
lemma not-min-gallery-double-split:
    |gallery Cs; ᄀ min-gallery Cs; {} & set (wall-crossings Cs)\rrbracket\Longrightarrow
    \existsfg As A B Bs E F Fs.
            (f,g)\infoldpairs ^ A\inf\vdash\mathcal{C}\wedgeB\ing\vdash\mathcal{C}\wedgeE\ing\vdash\mathcal{C}\wedgeF\inf\vdash\mathcal{C}\wedge
            (Cs=As@[A,B,F]@Fs\veeCs=As@[A,B]@Bs@[E,F]@Fs )
    using not-min-gallery-double-crosses-wall not-distinct-crossings-split-gallery
    by simp
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 \(D ; C \sim D ; C \neq D \rrbracket \Longrightarrow\)
            \(\exists f g\). OpposedThinChamberComplexFoldings \(X f g C \wedge D=g^{\circ} C\)
```

lemma (in ThinChamberComplex) ThinChamberComplexManyFoldingsI:
assumes chamber C0
and $\quad \wedge C D . \llbracket$ chamber $C$; chamber $D ; C \sim D ; C \neq D \rrbracket \Longrightarrow$
$\exists f g$. OpposedThinChamberComplexFoldings $X f g C \wedge D=g^{\circ} C$
shows ThinChamberComplexManyFoldings X C0
using assms
by (intro-locales, unfold-locales, fast)
lemma (in ThinChamberComplexManyFoldings) wall-crossings-subset-walls-betw: assumes min-gallery (C\#Cs@[D])

```
    shows set (wall-crossings (C#Cs@[D]))\subseteq walls-betw C D
proof
    fix H assume H\inset (wall-crossings (C#Cs@[D]))
    from this obtain As A B Bs
    where H:C#Cs@[D]=As@[A,B]@Bs H=the-wall-betw A B
    using in-set-wall-crossings-decomp
    by blast
    from assms have pgal: pgallery (C#Cs@[D])
    using min-gallery-pgallery by fast
    with H(1) obtain fg
    where fg:OpposedThinChamberComplexFoldings X f g A B=g'A
    using pgalleryD-chamber pgalleryD-adj
        binrelchain-append-reduce2[of adjacent As [A,B]@Bs]
        pgalleryD-distinct[of As@[A,B]@Bs] ex-walls[of A B]
    by auto
from H(2) fg have H': A\inf\vdash\mathcal{C}B\ing\vdash\mathcal{C}H={f\vdash\mathcal{C},g\vdash\mathcal{C}} H\in\mathrm{ walls}
    using OpposedThinChamberComplexFoldings.basech-halfchsys[
                OF fg(1)
        ]
        OpposedThinChamberComplexFoldings.chambers(2)[OF fg(1)]
        OpposedThinChamberComplexFoldings.this-wall-betwI[OF fg(1)]
        foldpairs-def
    by auto
have CD:C\inf\vdash\mathcal{C}\cupg\vdash\mathcal{C}D\inf\vdash\mathcal{C}\cupg\vdash\mathcal{C}
    using pgal pgalleryD-chamber chamber-system-def
        OpposedThinChamberComplexFoldings.halfchsys-decomp(1)[
                OF fg(1)
        ]
    by auto
show H}\in\mathrm{ walls-betw C D
proof (cases Bs As rule: two-lists-cases-snoc-Cons')
    case both-Nil with H show ?thesis
        using H'(3) the-wall-betw-nempty[of A B] unfolding walls-betw-def by force
next
    case (Nil1 E Es)
    show ?thesis
    proof (cases C\inf\vdash\mathcal{C})
        case True
        with Nil1 H(1) have separated-by H C D
            using H'(2,3) by (auto intro: separated-byI)
        thus ?thesis using H'(4) unfolding walls-betw-def by simp
    next
        case False with assms Nil1 H(1) show ?thesis
                using OpposedThinChamberComplexFoldings.foldg[
                    OF fg(1)
                    ]
                    CD(1) H'}(1,2) pgal pgallery
                    OpposedThinChamberComplexFoldings.flopped-half-chamber-systems-gf[
                    OF fg(1)
```

```
        ]
        ThinChamberComplexFolding.gallery-double-cross-not-minimal1[
        of X g E A B Es []
        ]
        by force
    qed
next
    case (Nil2 Fs F)
    show ?thesis
    proof (cases D\inf\vdash\mathcal{C})
        case True
        with assms Nil2 H(1) show ?thesis
            using OpposedThinChamberComplexFoldings.foldf[
                    OF fg(1)
            ]
            H'(1,2) pgal pgallery
            OpposedThinChamberComplexFoldings.flopped-half-chamber-systems-fg[
                    OF fg(1)
                ]
                    ThinChamberComplexFolding.gallery-double-cross-not-minimal-Cons1[
                    of Xf
                ]
        by force
    next
        case False with Nil2 H(1) have separated-by H C D
            using CD(2) H'(1,3) by (auto intro: separated-byI)
        thus ?thesis using H'(4) unfolding walls-betw-def by simp
    qed
next
    case (snoc-Cons Fs F E Es) show ?thesis
    proof (cases C\inf\vdash\mathcal{C D\ing\vdash\mathcal{C rule: two-cases)}}\mathbf{~}\mathrm{ )}
        case both thus ?thesis
            using H'(3,4) walls-betw-def unfolding separated-by-def by auto
    next
        case one
        with snoc-Cons assms H(1) show ?thesis
            using OpposedThinChamberComplexFoldings.foldf[
                        OF fg(1)
                    ]
                            CD(2) H'(2) pgal pgallery
                            OpposedThinChamberComplexFoldings.flopped-half-chamber-systems-fg[
                                OF fg(1)
                        ]
                            ThinChamberComplexFolding.gallery-double-cross-not-minimal1[
                                of XfCBDEs@[A]
                    ]
            by fastforce
    next
        case other
```

```
        with snoc-Cons assms H(1) show ?thesis
        using OpposedThinChamberComplexFoldings.ThinChamberComplexFolding-g[
                OF fg(1)
            ]
            CD(1) H'(1) pgal pgallery
            OpposedThinChamberComplexFoldings.flopped-half-chamber-systems-gf[
                OF fg(1)
                ]
                        ThinChamberComplexFolding.gallery-double-cross-not-minimal1[
                        of X g E A F Es B#Fs
            ]
        by force
        next
        case neither
        hence separated-by {g\vdash\mathcal{C},f\vdash\mathcal{C}}CD using CD by (auto intro: separated-byI)
        thus ?thesis
            using H'(3,4) walls-betw-def by (auto simp add: insert-commute)
        qed
    qed
qed
```


### 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 ThinChamberComplexManyFoldings
```

begin
definition fundfoldpairs :: (('a>' $\left.a) \times\left({ }^{\prime} a \Rightarrow^{\prime} a\right)\right)$ set
where fundfoldpairs $\equiv\{(f, g)$. OpposedThinChamberComplexFoldings XfgC0 $\}$
abbreviation fundadjset $\equiv$ adjacentset $C 0-\{C 0\}$
abbreviation induced-automorph $::\left({ }^{\prime} a \Rightarrow{ }^{\prime} a\right) \Rightarrow\left({ }^{\prime} a \Rightarrow^{\prime} a\right) \Rightarrow\left({ }^{\prime} a \Rightarrow^{\prime} a\right)$
where induced-automorph $f g \equiv$
OpposedThinChamberComplexFoldings.induced-automorphism Xfg
abbreviation Abs-induced-automorph $::\left({ }^{\prime} a \Rightarrow^{\prime} a\right) \Rightarrow\left({ }^{\prime} a \Rightarrow^{\prime} a\right) \Rightarrow{ }^{\prime}$ 'a permutation
where Abs-induced-automorph $f g \equiv$ Abs-permutation (induced-automorph f $g$ )
abbreviation $S \equiv \bigcup(f, g) \in$ fundfoldpairs. $\{$ Abs-induced-automorph $f g\}$
abbreviation $W \equiv\langle S\rangle$

```
lemma fundfoldpairs-induced-autormorph-bij:
    (f,g) f fundfoldpairs \Longrightarrowbij (induced-automorph fg)
    using OpposedThinChamberComplexFoldings.induced-automorphism-bij
    unfolding fundfoldpairs-def
    by fast
lemmas permutation-conv-induced-automorph =
    Abs-permutation-inverse[OF CollectI, OF fundfoldpairs-induced-autormorph-bij]
lemma fundfoldpairs-induced-autormorph-order2:
    (f,g) \in fundfoldpairs \Longrightarrow induced-automorph fg\circ induced-automorph f g = id
    using OpposedThinChamberComplexFoldings.indaut-order2
    unfolding fundfoldpairs-def
    by fast
lemma fundfoldpairs-induced-autormorph-ntrivial:
    (f,g) \in fundfoldpairs \Longrightarrow induced-automorph fg}\not=i
    using OpposedThinChamberComplexFoldings.induced-automorphism-ntrivial
    unfolding fundfoldpairs-def
    by fast
lemma fundfoldpairs-fundchamber-image:
    (f,g)\infundfoldpairs \Longrightarrow Abs-induced-automorph fg '}->\textrm{C0}=\mp@subsup{g}{}{`}\textrm{C0
    using fundfoldpairs-def
    by (simp add:
        permutation-conv-induced-automorph
        OpposedThinChamberComplexFoldings.induced-automorphism-C0
    )
lemma fundfoldpair-fundchamber-in-half-chamber-system-f:
    (f,g)\infundfoldpairs \LongrightarrowC0\inf\vdash\mathcal{C}
    using fundfoldpairs-def
        OpposedThinChamberComplexFoldings.basech-halfchsys(1)
    by fast
lemma fundfoldpair-unique-half-chamber-system-f:
    assumes (f,g)\infundfoldpairs ( }\mp@subsup{f}{}{\prime},\mp@subsup{g}{}{\prime})\in\mathrm{ fundfoldpairs
    Abs-induced-automorph f' g' = Abs-induced-automorph f g
    shows }\quad\mp@subsup{f}{}{\prime}\vdash\mathcal{C}=f\vdash\mathcal{C
proof-
    from assms have g'`C0 = g'C0
    using fundfoldpairs-fundchamber-image[OF assms(1)
                fundfoldpairs-fundchamber-image[OF assms(2)]
    by simp
    with assms show f
        using fundfoldpairs-def
                OpposedThinChamberComplexFoldings.unique-half-chamber-system-f [
                        of XfgC0 f' g'
            ]
```

```
    by auto
qed
lemma fundfoldpair-unique-half-chamber-systems-chamber-ng-f:
    assumes ( }f,g)\infundfoldpairs ( f f',g')\infundfoldpairs
        Abs-induced-automorph f' }\mp@subsup{g}{}{\prime}=Abs-induced-automorph fg
        chamber C C }\not\ing\vdash\mathcal{C
    shows }C\inf{\mathcal{C
    using assms(1,3-5) fundfoldpairs-def chamber-system-def
        OpposedThinChamberComplexFoldings.flopped-half-chamber-systems-gf[
        THEN sym
    ]
    fundfoldpair-unique-half-chamber-system-f[OF assms(1,2)]
    by fastforce
lemma the-wall-betw-adj-fundchamber:
    (f,g)\infundfoldpairs \Longrightarrow
        the-wall-betw C0 (Abs-induced-automorph f g'->C0)={f\vdash\mathcal{C},g\vdash\mathcal{C}}
    using fundfoldpairs-def
        OpposedThinChamberComplexFoldings.this-wall-betw-basechambers
        OpposedThinChamberComplexFoldings.induced-automorphism-C0
    by (fastforce simp add: permutation-conv-induced-automorph)
lemma zero-notin-S: 0\not\inS
proof
    assume 0\inS
    from this obtain fg
        where (f,g)\infundfoldpairs 0 = Abs-induced-automorph fg
    by fast
    thus False
    using Abs-permutation-inject[of id induced-automorph fg]
        bij-id fundfoldpairs-induced-autormorph-bij
        fundfoldpairs-induced-autormorph-ntrivial
    by (force simp add:zero-permutation.abs-eq)
qed
lemma S-order2-add: s\inS\Longrightarrows+s=0
    using fundfoldpairs-induced-autormorph-bij zero-permutation.abs-eq
    by (fastforce simp add:
        plus-permutation-abs-eq fundfoldpairs-induced-autormorph-order2
    )
lemma S-add-order2:
    assumes }s\in
    shows add-order s = 2
proof (rule add-order-equality)
    from assms show s+^2 = 0 using S-order2-add by (simp add: nataction-2)
next
    fix m assume 0<m s+` m}=
```

with assms show $2 \leq m$ using zero-notin-S by (cases $m=1$ ) auto qed $\operatorname{simp}$
lemmas $S$-uminus $=$ minus-unique[OF S-order2-add]
lemma $S$-sym: uminus ' $S \subseteq S$
using $S$-uminus by auto
lemmas sum-list-S-in-W $=$ sum-list-lists-in-genby-sym[OF S-sym]
lemmas $W$-conv-sum-lists $=$ genby-sym-eq-sum-lists[OF S-sym]
lemma $S$-endomorphism:
$s \in S \Longrightarrow$ ChamberComplexEndomorphism $X$ (permutation s)
using fundfoldpairs-def
OpposedThinChamberComplexFoldings.induced-automorphism-morphism
by (fastforce simp add: permutation-conv-induced-automorph)
lemma $S$-list-endomorphism: ss $\in$ lists $S \Longrightarrow$ ChamberComplexEndomorphism X (permutation (sum-list ss)) by (induct ss)
(auto simp add:
zero-permutation.rep-eq trivial-endomorphism plus-permutation.rep-eq S-endomorphism ChamberComplexEndomorphism.endo-comp )
lemma $W$-endomorphism:
$w \in W \Longrightarrow$ ChamberComplexEndomorphism $X$ (permutation $w$ )
using $W$-conv-sum-lists $S$-list-endomorphism by auto
lemma $S$-automorphism:
$s \in S \Longrightarrow$ ChamberComplexAutomorphism $X$ (permutation s)
using fundfoldpairs-def
OpposedThinChamberComplexFoldings.induced-automorphism-automorphism
by (fastforce simp add: permutation-conv-induced-automorph)
lemma $S$-list-automorphism:
ss $\in$ lists $S \Longrightarrow$ ChamberComplexAutomorphism $X$ (permutation (sum-list ss))
by (induct ss)
(auto simp add:
zero-permutation.rep-eq trivial-automorphism plus-permutation.rep-eq S-automorphism ChamberComplexAutomorphism.comp
)
lemma $W$-automorphism:
$w \in W \Longrightarrow$ ChamberComplexAutomorphism $X$ (permutation $w$ )
using $W$-conv-sum-lists $S$-list-automorphism by auto
lemma $S$-respects-labels: $\llbracket$ label-wrt $B \varphi ; s \in S ; v \in(\bigcup X) \rrbracket \Longrightarrow \varphi(s \rightarrow v)=\varphi v$ using fundfoldpairs-def

```
        OpposedThinChamberComplexFoldings.indaut-resplabels[
        of X - CO B \varphiv
        ]
    by (auto simp add: permutation-conv-induced-automorph)
lemma S-list-respects-labels:
    \llbracketlabel-wrt B \varphi; ss\inlists S;v\in(\bigcupX)\rrbracket\Longrightarrow\varphi(sum-list ss }->v)=\varphi
    using S-endomorphism ChamberComplexEndomorphism.vertex-map[of X]
    by (induct ss arbitrary: v rule: rev-induct)
        (auto simp add:
        plus-permutation.rep-eq S-respects-labels zero-permutation.rep-eq
        )
lemma \(W\)-respects-labels:
【label-wrt \(B \varphi ; w \in W ; v \in(\bigcup X) \rrbracket \Longrightarrow \varphi(w \rightarrow v)=\varphi v\)
using \(W\)-conv-sum-lists S-list-respects-labels \([\) of \(B \varphi-v]\) by auto
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: $s \in S \Longrightarrow$ chamber $\left(s^{\iota} \rightarrow C 0\right)$
using fundfoldpairs-def
by (fastforce simp add:
fundfoldpairs-fundchamber-image OpposedThinChamberComplexFoldings.chamber-D0 )
lemma fundchamber- $W$-image-chamber:
$w \in W \Longrightarrow \operatorname{chamber}\left(w^{\prime} \rightarrow C 0\right)$
using fundchamber $W$-endomorphism ChamberComplexEndomorphism.chamber-map
by auto
lemma fundchamber-S-adjacent: $s \in S \Longrightarrow C 0 \sim\left(s^{\iota} \rightarrow C 0\right)$
using fundfoldpairs-def
by (auto simp add:
fundfoldpairs-fundchamber-image
OpposedThinChamberComplexFoldings.chambers(2)
)
lemma fundchamber-WS-image-adjacent: $w \in W \Longrightarrow s \in S \Longrightarrow\left(w^{\prime} \rightarrow C 0\right) \sim\left((w+s)^{\iota} \rightarrow C 0\right)$
using fundchamber fundchamber-S-adjacent fundchamber-S-chamber
$W$-endomorphism
ChamberComplexEndomorphism.adj-map[of $X$ permutation w C0 s $\left.s^{s} \rightarrow C 0\right]$
by (auto simp add: image-comp plus-permutation.rep-eq)
lemma fundchamber-S-image-neq-fundchamber: $s \in S \Longrightarrow s^{4} \rightarrow C 0 \neq C 0$
using fundfoldpairs-def OpposedThinChamberComplexFoldings.chambers(3)
by (fastforce simp add: fundfoldpairs-fundchamber-image)
lemma fundchamber-next-WS-image-neq:
assumes $s \in S$
shows $\quad(w+s)^{'} \rightarrow C 0 \neq w^{'} \rightarrow C 0$
proof
assume $(w+s){ }^{\prime} \rightarrow C 0=w^{'} \rightarrow C 0$
with assms show False
using fundchamber-S-image-neq-fundchamber[of s]
by (auto simp add: plus-permutation.rep-eq image-comp permutation-eq-image)
qed
lemma fundchamber-S-fundadjset: $s \in S \Longrightarrow s^{4} \rightarrow C 0 \in$ fundadjset using fundchamber-S-adjacent fundchamber-S-image-neq-fundchamber fundchamber-S-chamber chamberD-simplex adjacentset-def
by simp
lemma fundadjset-eq-S-image: $D \in$ fundadjset $\Longrightarrow \exists s \in S . D=s^{6} \rightarrow C 0$
using fundchamber adjacentsetD-adj adjacentset-chamber ex-walls[of C0 D] fundfoldpairs-def fundfoldpairs-fundchamber-image
by blast
lemma $S$-fixespointwise-fundchamber-image-int:
assumes $s \in S$
shows fixespointwise $((\rightarrow) s)\left(C 0 \cap s^{\star} \rightarrow C 0\right)$
proof-
from $\operatorname{assms}(1)$ obtain $f g$
where $f g:(f, g) \in$ fundfoldpairs $s=$ Abs-induced-automorph $f g$
by fast
show ?thesis
proof (rule fixespointwise-cong)
from fg show fun-eq-on $((\rightarrow)$ s) (induced-automorph $f g)\left(C 0 \cap s^{〔} \rightarrow C 0\right)$
using permutation-conv-induced-automorph fun-eq-onI by fastforce
from $f g$ show fixespointwise (induced-automorph $f g)\left(C 0 \cap s^{s} \rightarrow C 0\right)$
using fundfoldpairs-fundchamber-image fundfoldpairs-def OpposedThinChamberComplexFoldings.indaut-fixes-fundfacet
by auto
qed
qed
lemma $S$-fixes-fundchamber-image-int:
$s \in S \Longrightarrow s^{6} \rightarrow\left(C 0 \cap s^{6} \rightarrow C 0\right)=C 0 \cap s^{\star} \rightarrow C 0$
using fixespointwise-im[OF S-fixespointwise-fundchamber-image-int] by simp

```
lemma fundfacets:
    assumes s\inS
    shows C0\caps}\mp@subsup{}{}{\iota}->C0\triangleleftC0C0\cap\mp@subsup{s}{}{`}->C0\triangleleft\mp@subsup{s}{}{\iota}->C
    using assms fundchamber-S-adjacent[of s]
            fundchamber-S-image-neq-fundchamber[of s]
            adjacent-int-facet1[of C0] adjacent-int-facet2[of C0]
    by auto
lemma fundadjset-ex1-eq-S-image:
    assumes D\infundadjset
    shows }\exists!s\inS.D=\mp@subsup{s}{}{*}->C
proof (rule ex-ex1I)
    from assms show \existss.s\inS\wedgeD=s'->C0
        using fundadjset-eq-S-image by fast
next
    fix st assume s\inS ^D= s}->->C0t\inS\wedgeD= t'->C
    hence s:s\inS D= s}->C
        and t:t\inS D = t'->C0
        by auto
    from s(1) t(1) obtain fg\mp@subsup{f}{}{\prime}}\mp@subsup{g}{}{\prime
        where (f,g)\infundfoldpairs s=Abs-induced-automorph f g
        and }(\mp@subsup{f}{}{\prime},g')\in\mathrm{ fundfoldpairs }t=Abs-induced-automorph f' g'
        by auto
    with }s(2)t(2) show s=
        using fundfoldpairs-def fundfoldpairs-fundchamber-image
                OpposedThinChamberComplexFoldings.induced-automorphism-unique[
                of X (f' g'C0fg
                    ]
        by auto
qed
lemma fundchamber-S-image-inj-on: inj-on ( }\lambda\mathrm{ s. s' }->\mathrm{ C0) S
proof (rule inj-onI)
    fix st assume s\inS t\inS s}\mp@subsup{}{}{\iota}->C0=\mp@subsup{t}{}{6}->C0\mathrm{ thus }s=
        using fundchamber-S-fundadjset
                bex1-equality[OF fundadjset-ex1-eq-S-image, of s}\mp@subsup{s}{}{\prime}->C0 st
    by simp
qed
lemma S-list-image-gallery:
    ss\inlists S\Longrightarrow gallery (map (\lambdaw. w}->\mathrm{ CO) (sums ss))
proof (induct ss rule: list-induct-ssnoc)
    case (Single s) thus ?case
        using fundchamber fundchamber-S-chamber fundchamber-S-adjacent
                gallery-def
    by (fastforce simp add:zero-permutation.rep-eq)
next
```

case (ssnoc ss st)
define $C s D E$ where $C s=\operatorname{map}\left(\lambda w . w^{'} \rightarrow C 0\right)$ (sums ss)
and $D=$ sum-list $(s s @[s]){ }^{\prime} \rightarrow C 0$
and $E=$ sum-list $(s s @[s, t]){ }^{‘} \rightarrow C 0$
with ssnoc have gallery (Cs@[D,E])
using sum-list-S-in-W[of ss@ $[s, t]]$ sum-list-S-in-W $[$ of ss@ $[s]]$
fundchamber-W-image-chamber
fundchamber-WS-image-adjacent $[$ of sum-list $(s s @[s]) t]$
sum-list-append[of ss@[s][t]]
by (auto intro: gallery-snocI simp add: sums-snoc)
with Cs-def D-def E-def show ?case using sums-snoc $[o f ~ s s @[s] t]$ by (simp add:
sums-snoc)
qed (auto simp add: gallery-def fundchamber zero-permutation.rep-eq)
lemma pgallery-last-eq-W-image:
pgallery $(C 0 \# C s @[C]) \Longrightarrow \exists w \in W . C=w^{\iota} \rightarrow C 0$
proof (induct Cs arbitrary: $C$ rule: rev-induct)
case Nil
hence $C \in$ fundadjset
using pgallery-def chamberD-simplex adjacentset-def by fastforce
from this obtain $s$ where $s \in S C=s^{6} \rightarrow C 0$
using fundadjset-eq-S-image $[o f C]$ by auto
thus ?case using genby-genset-closed[of s S] by fast

## next

case (snoc D Ds)
have $D C$ : chamber $D$ chamber $C D \sim C D \neq C$
using pgallery-def snoc(2)
binrelchain-append-reduce2[of adjacent C0\#Ds [D,C]]
by auto
from snoc obtain $w$ where $w: w \in W D=w^{4} \rightarrow C 0$
using pgallery-append-reduce $1[$ of $C 0 \# D s @[D][C]]$ by force
from $w(2)$ have $(-w)^{\rightarrow} \rightarrow D=C 0$
by (simp add:
image-comp plus-permutation.rep-eq[THEN sym]
zero-permutation.rep-eq
)
with $D C w(1)$ have $C 0 \sim(-w)^{\iota} \rightarrow C C 0 \neq(-w)^{\iota} \rightarrow C(-w)^{\natural} \rightarrow C \in X$
using genby-uminus-closed $W$-endomorphism[of $-w]$
ChamberComplexEndomorphism.adj-map[of X - D C]
permutation-eq-image $[o f-w D$ ] chamberD-simplex $[$ of $C]$
ChamberComplexEndomorphism.simplex-map[of X permutation $(-w) C]$
by auto
hence $(-w)^{d} \rightarrow C \in$ fundadjset using adjacentset-def by fast
from this obtain $s$ where $s: s \in S(-w)^{6} \rightarrow C=s^{6} \rightarrow C 0$
using fundadjset-eq-S-image by force
from $s(2)$ have
(permutation $w \circ$ permutation $(-w))$ ' $C=($ permutation $w \circ$ permutation $s){ }^{\text {' }} \mathrm{C} 0$
by (simp add: image-comp[THEN sym])
hence $C=(w+s)^{4} \rightarrow C 0$

```
    by (simp add: plus-permutation.rep-eq[THEN sym] zero-permutation.rep-eq)
    with w(1) s(1) show ?case
    using genby-genset-closed[of s S] genby-add-closed by blast
qed
lemma chamber-eq-W-image:
    assumes chamber C
    shows }\existsw\inW.C=\mp@subsup{w}{}{\iota}->C
proof (cases C=C0)
    case True
    hence }0\inWC=\mp@subsup{0}{}{〔}->C
    using genby-0-closed by (auto simp add:zero-permutation.rep-eq)
    thus ?thesis by fast
next
    case False with assms show ?thesis
        using fundchamber chamber-pconnect pgallery-last-eq-W-image by blast
qed
lemma S-list-image-crosses-walls:
    ss}\in\mathrm{ lists S ב{}& set (wall-crossings (map ( }\lambdaw.\mp@subsup{w}{}{\prime}->C0)(sums ss)))
proof (induct ss rule: list-induct-ssnoc)
    case (Single s) thus ?case
        using fundchamber fundchamber-S-chamber fundchamber-S-adjacent
                fundchamber-S-image-neq-fundchamber[of s] ex-walls[of C0 s}
                OpposedThinChamberComplexFoldings.this-wall-betw-basechambers
    by (force simp add:zero-permutation.rep-eq)
next
    case (ssnoc ss st)
    moreover
    define A B where A=sum-list (ss@[s])'->C0 and B=sum-list (ss@[s,t])
    ->C0
    moreover from ssnoc(2) A-def B-def obtain fg
        where OpposedThinChamberComplexFoldings Xfg A B=g'A
        using sum-list-S-in-W[of ss@[s]] sum-list-S-in-W[of ss@[s,t]]
            fundchamber-W-image-chamber sum-list-append[of ss@[s][t]]
            fundchamber-next-WS-image-neq[of t sum-list (ss@[s])]
            fundchamber-WS-image-adjacent [of sum-list (ss@[s])t]
            ex-walls[of A B]
        by auto
        ultimately show ?case
        using OpposedThinChamberComplexFoldings.this-wall-betw-basechambers
            sums-snoc[of ss@[s]t]
    by (force simp add: sums-snoc wall-crossings-snoc)
qed (simp add: zero-permutation.rep-eq)
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 }=>(\mp@subsup{}{}{\prime}a\mp@subsup{|}{}{\prime}a)\times(\mp@subsup{'}{}{\prime}a\mp@subsup{|}{}{\prime}a
    where Spair s \equiv
        SOME fg.fg f fundfoldpairs }\wedges=\mathrm{ case-prod Abs-induced-automorph fg
lemma Spair-fundfoldpair: s\inS\Longrightarrow Spair s f fundfoldpairs
    using Spair-def
        someI-ex[of
            \lambdag. fg \in fundfoldpairs }
                s= case-prod Abs-induced-automorph fg
            ]
    by auto
```

lemma Spair-induced-automorph:
$s \in S \Longrightarrow s=$ case-prod Abs-induced-automorph (Spair s)
using Spair-def
someI-ex[of
$\lambda f g . f g \in$ fundfoldpairs $\wedge$
$s=$ case-prod Abs-induced-automorph fg
]
by auto
lemma S-list-pgallery-decomp1:
assumes ss: set $s s=S$ and gal: Cs $\neq[]$ pgallery $(C 0 \# C s)$
shows $\exists s \in$ set ss. $\exists C \in$ set Cs. $\forall(f, g) \in$ fundfoldpairs.
$s=$ Abs-induced-automorph $f g \longrightarrow C \in g \vdash \mathcal{C}$
proof (cases Cs)
case (Cons D Ds)
with $\operatorname{gal}(2)$ have $D \in$ fundadjset
using pgallery-def chamberD-simplex adjacentset-def by fastforce
from this obtain $s$ where $s: s \in S D=s^{4} \rightarrow C 0$
using fundadjset-eq-S-image by blast
from $s(2)$ have
$\forall(f, g) \in$ fundfoldpairs. $s=$ Abs-induced-automorph $f g \longrightarrow D \in g \vdash \mathcal{C}$
using fundfoldpairs-def fundfoldpairs-fundchamber-image
OpposedThinChamberComplexFoldings.basechambers-half-chamber-systems(2)
by auto
with $s(1)$ ss Cons show ?thesis by auto
qed (simp add: gal(1))

```
lemma S-list-pgallery-decomp2:
    assumes set ss=S Cs\not=[] pgallery (C0#Cs)
    shows
    \existsrs s ts.ss=rs@s#ts ^
        (\existsC\inset Cs.}\forall(f,g)\in\mathrm{ fundfoldpairs.
        s=Abs-induced-automorph f g\longrightarrowC\ing\vdash\mathcal{C})\wedge
        ( }\forallr\in\mathrm{ set rs. }\forallC\in\mathrm{ set Cs. }\forall(f,g)\infundfoldpairs
            r=Abs-induced-automorph fg\longrightarrowC\inf\vdash\mathcal{C})
proof-
    from assms obtain rs s ts where rs-s-ts:
        ss=rs@s#ts
        \existsC\inset Cs.}\forall(f,g)\infundfoldpairs
            s=Abs-induced-automorph f g\longrightarrowC\ing\vdash\mathcal{C}
        \forallr\inset rs. }\forallC\inset Cs
            \neg ( \forall ( f , g ) \in \text { fundfoldpairs. r = Abs-induced-automorph f g } \longrightarrow C \in g \vdash \mathcal { C } )
        using split-list-first-prop[OF S-list-pgallery-decomp1, of ss Cs]
        by auto
    have }\forallr\in\mathrm{ set rs. }\forallC\in\mathrm{ set Cs. }\forall(f,g)\infundfoldpairs
                r=Abs-induced-automorph f g\longrightarrowC\inf\vdash\mathcal{C}
    proof (rule ballI, rule ballI, rule prod-ballI, rule impI)
    fix r Cfg
    assume r set rs C \in set Cs (f,g)\infundfoldpairs
                r=Abs-induced-automorph fg
    with rs-s-ts(3) assms(3) show C\inf\vdash\mathcal{C}
            using pgalleryD-chamber
                fundfoldpair-unique-half-chamber-systems-chamber-ng-f[
                    of -- fg C
                ]
            by fastforce
    qed
    with rs-s-ts(1,2) show ?thesis by auto
qed
lemma S-list-pgallery-decomp3:
    assumes set ss =S Cs\not=[] pgallery (C0#Cs)
    shows
        \existsrs s ts As B Bs.ss=rs@s#ts ^Cs=As@B#Bs^
            (\forall(f,g)\infundfoldpairs.s=Abs-induced-automorph f g\longrightarrowB\ing\vdash\mathcal{C})^
            ( }\forallA\in\mathrm{ set As. }\forall(f,g)\infundfoldpairs
                s=Abs-induced-automorph f g\longrightarrowA\inf\vdash\mathcal{C})\wedge
            ( }\forallr\in\mathrm{ set rs. }\forallC\in\mathrm{ set Cs. }\forall(f,g)\infundfoldpairs
                r=Abs-induced-automorph fg\longrightarrowC\inf\vdash\mathcal{C})
proof-
    from assms obtain rs s ts where rs-s-ts:
        ss=rs@s#ts
        B\inset Cs.}\forall(f,g)\infundfoldpairs.s=Abs-induced-automorph fg\longrightarrowB\ing\vdash\mathcal{C
        \forallr\inset rs. }\forallB\in\mathrm{ set Cs. }\forall(f,g)\infundfoldpairs
            r=Abs-induced-automorph fg\longrightarrowB\inf\vdash\mathcal{C}
```

```
    using S-list-pgallery-decomp2[of ss Cs]
    by auto
    obtain As B Bs where As-B-Bs:
    Cs=As@B#Bs
    \forall(f,g)\infundfoldpairs.s=Abs-induced-automorph f g\longrightarrowB
    \forallA\inset As. \exists (f,g)\infundfoldpairs. s=Abs-induced-automorph f g ^A\not\ing\vdash\mathcal{C}
    using split-list-first-prop[OF rs-s-ts(2)]
    by fastforce
    from As-B-Bs(1,3) assms(3)
    have }\forallA\in\mathrm{ set As. }\forall(f,g)\infundfoldpairs
        s=Abs-induced-automorph f g\longrightarrowA\inf\vdash\mathcal{C}
        using pgalleryD-chamber
                fundfoldpair-unique-half-chamber-systems-chamber-ng-f
    by auto
    with rs-s-ts(1,3) As-B-Bs(1,2) show ?thesis by fast
qed
lemma fundfold-trivial-fC:
    r\inS\Longrightarrow\forall(f,g)\infundfoldpairs. }r=\mathrm{ Abs-induced-automorph f g}\longrightarrowC\inf\vdash\mathcal{C}
    fst (Spair r)' C=C
    using Spair-fundfoldpair[of r] Spair-induced-automorph[of r] fundfoldpairs-def
        OpposedThinChamberComplexFoldings.axioms(2)[
        of X fst (Spair r) snd (Spair r) C0
        ]
        ChamberComplexFolding.chamber-retraction2 [of X fst (Spair r) C]
    by fastforce
lemma fundfold-comp-trivial-fC:
    set rs \subseteqS\Longrightarrow
    \forallr\inset rs..}\forall(f,g)\infundfoldpairs
        r=Abs-induced-automorph fg\longrightarrowC\inf\vdash\mathcal{C}\Longrightarrow
    fold fst (map Spair rs) ' C = C
proof (induct rs)
    case (Cons r rs)
    have fold fst (map Spair (r#rs))' C=
            fold fst (map Spair rs)'fst (Spair r)'C
    by (simp add: image-comp)
    also from Cons have ... = C by (simp add: fundfold-trivial-f\mathcal{C})
    finally show ?case by fast
qed simp
lemma fundfold-trivial-f\mathcal{C-list:}
    r\inS\Longrightarrow
    \forallC\inset Cs. }\forall(f,g)\in\mathrm{ fundfoldpairs.
        r=Abs-induced-automorph fg\longrightarrowC\inf\vdash\mathcal{C}\Longrightarrow
    fst (Spair r) \modelsCs=Cs
    using fundfold-trivial-f\mathcal{C by (induct Cs) auto}
lemma fundfold-comp-trivial-f\mathcal{C-list:}
```

```
    set rs \subseteqS\Longrightarrow
    \forallr\inset rs. }\forallC\in\mathrm{ set Cs. }\forall(f,g)\infundfoldpairs
        r=Abs-induced-automorph f g\longrightarrowC\inf\vdash\mathcal{C}\Longrightarrow
    fold fst (map Spair rs) \modelsCs=Cs
proof (induct rs Cs rule: list-induct2')
    case (4 r rs C Cs)
    from 4(3)
        have r:\forallD\inset (C#Cs).}\forall(f,g)\infundfoldpairs
                r=Abs-induced-automorph f g\longrightarrowD\inf\vdash\mathcal{C}
        by simp
    from 4(2)
        have fold fst (map Spair (r#rs)) \models(C#Cs)=
                map ((`) (fold fst (map Spair rs))) (fst (Spair r) =(C#Cs))
    by (auto simp add: image-comp)
    also from 4 have ... = C#Cs
    using fundfold-trivial-fC-list[of r C#Cs]
    by (simp add: fundfold-comp-trivial-fC)
    finally show ?case by fast
qed auto
lemma fundfold-gallery-map:
    s\inS\Longrightarrow gallery Cs \Longrightarrow gallery (fst (Spair s)\modelsCs)
    using Spair-fundfoldpair fundfoldpairs-def
        OpposedThinChamberComplexFoldings.axioms(2)
        ChamberComplexFolding.gallery-map[of X fst (Spair s)]
    by fastforce
lemma fundfold-comp-gallery-map:
    assumes pregal: gallery Cs
    shows set ss}\subseteqS\Longrightarrow\mathrm{ gallery (fold fst (map Spair ss) }\modelsCs
proof (induct ss rule: rev-induct)
    case (snoc s ss)
    hence 1: gallery (fst (Spair s)\models(fold fst (map Spair ss) \modelsCs))
    using fundfold-gallery-map by fastforce
    have 2: fst (Spair s) \models(fold fst (map Spair ss) \modelsCs)=
            fold fst (map Spair (ss@[s])) \modelsCs
    by (simp add: image-comp)
    show ?case using 1 subst[OF 2, of gallery, OF 1] by fast
qed (simp add: pregal galleryD-adj)
lemma fundfold-comp-pgallery-ex-funpow:
    assumes ss: set ss=S
    shows pgallery(C0#Cs@[C])\Longrightarrow
    \existsn. (fold fst (map Spair ss) ^n n)'C=C0
proof (induct Cs arbitrary:C rule: length-induct)
    fix Cs C
    assume step : \forallys. length ys < length Cs }
                            (\forallx.pgallery (C0 # ys @ [x])\longrightarrow
                            (\existsn.(fold fst (map Spair ss) ^~n)`}x=C0)
```

and set-up: pgallery (C0\#Cs@[C])
from ss set-up obtain rs $s$ ts $A s B$ bsere decomps:
$s s=r s @ s \# t s C s @[C]=A s @ B \# B s$
$\forall(f, g) \in$ fundfoldpairs. $s=$ Abs-induced-automorph $f g \longrightarrow B \in g \vdash \mathcal{C}$
$\forall A \in$ set As. $\forall(f, g) \in$ fundfoldpairs. $s=$ Abs-induced-automorph $f g \longrightarrow A \in f \vdash \mathcal{C}$
$\forall r \in$ set $r s . \forall D \in$ set $(C s @[C]) . \forall(f, g) \in$ fundfoldpairs. $r=$ Abs-induced-automorph $f g \longrightarrow D \in f \vdash \mathcal{C}$
using $S$-list-pgallery-decomp3[of ss $C s @[C]$ ]
by fastforce
obtain Es $E$ where $E s E: C 0 \# A s=E s @[E]$ using cons-conv-snoc by fast
have $E s E-s-f \mathcal{C}$ :
$\forall A \in \operatorname{set}(E s @[E]) . \forall(f, g) \in$ fundfoldpairs. $s=A b s$-induced-automorph $f g \longrightarrow A \in f \vdash \mathcal{C}$
proof (rule ballI)
fix $A$ assume $A \in \operatorname{set}(E s @[E])$
with EsE decomps(4)
show $\forall(f, g) \in$ fundfoldpairs. $s=$ Abs-induced-automorph $f g \longrightarrow A \in f \vdash \mathcal{C}$
using fundfoldpair-fundchamber-in-half-chamber-system-f
set-ConsD[of A C0 As]
by auto
qed
moreover from $\operatorname{decomps}(2) E s E$
have decomp2: $C 0 \# C s @[C]=E s @ E \# B \# B s$
by $\operatorname{simp}$
moreover from $s s$ decomps (1) have $s \in S$ by auto
ultimately have $s B$ : $f s t$ (Spair s)' $B=E$
using set-up decomps(3) Spair-fundfoldpair[of s] Spair-induced-automorph[of s] fundfoldpairs-def pgalleryD-adj binrelchain-append-reduce2[of adjacent Es E\#B\#Bs]
OpposedThinChamberComplexFoldings.adjacent-half-chamber-system-image-fg[ of $X$ fst (Spair s) snd (Spair s) C0 E B
]
by auto
show $\exists n$. (fold fst (map Spair ss) ${ }^{\sim} n$ ) ' $C=C 0$
proof (cases Es=[] $\wedge B s=[])$
case True
from $\operatorname{decomps}(5)$ have
$\forall r \in$ set $r$ s. $\forall(f, g) \in$ fundfoldpairs. $r=$ Abs-induced-automorph $f g \longrightarrow C \in f \vdash \mathcal{C}$
by auto
with decomps(1) ss
have fold fst (map Spair ss) ' $C=$ fold fst (map Spair ts) 'fst (Spair s) ' $C$
using fundfold-comp-trivial-fC $[$ of rs $C$ ]
by (fastforce simp add: image-comp[THEN sym])
moreover
have $\forall r \in$ set ts. $\forall(f, g) \in$ fundfoldpairs.
$r=A b s$-induced-automorph $f g \longrightarrow C 0 \in f \vdash \mathcal{C}$

```
    using fundfoldpair-fundchamber-in-half-chamber-system-f
    by fast
    ultimately have (fold fst (map Spair ss)^^1) 'C=C0
    using True decomps(1,2) ss EsE sB fundfold-comp-trivial-f\mathcal{C}[of ts C0]
        fundfold-comp-trivial-fC[of ts C0]
    by fastforce
    thus?thesis by fast
next
    case False have EsBs: \neg(Es = []^ Bs=[]) by fact
    show ?thesis
    proof (cases fold fst (map Spair ss)'C=C0)
    case True
    hence (fold fst (map Spair ss)~ 1)'C=C0 by simp
    thus ?thesis by fast
    next
    case False
    from decomps(5) have C0CsC-rs-fC:
        \forallr\inset rs.}\forallD\inset (C0#Cs@[C]).\forall(f,g)\infundfoldpairs
        r=Abs-induced-automorph f g\longrightarrowD\inf\vdash\mathcal{C}
        using fundfoldpair-fundchamber-in-half-chamber-system-f
        by auto
    from decomps(1)
        have fold fst (map Spair (rs@[s]))\models(C0#Cs@[C])=
                fst (Spair s) \models(fold fst (map Spair rs) \models(C0#Cs@[C]))
        by (simp add: image-comp)
    also from ss decomps(1)
        have ...= fst (Spair s)\models(C0#Cs@[C])
```



```
        by fastforce
    also from decomp2 have ... = fst (Spair s) =(Es@E#B#Bs)
        by (simp add: image-comp)
    finally
        have fold fst (map Spair (rs@[s]))\models(C0#Cs@[C])=
            Es @ E # E # fst (Spair s)\modelsBs
```



```
    by fastforce
    with set-up ss decomps(1)
    have gal:gallery (Es @ E # fst (Spair s) \modelsBs)
    using pgallery fundfold-comp-gallery-map[of - rs@[s]]
                gallery-remdup-adj[of Es E fst (Spair s) \modelsBs]
    by fastforce
    from EsBs decomp2 EsE
    have \existsZs. length Zs < length Cs ^
                        Es @ E# fst (Spair s) \modelsBs=C0 # Zs@ @fst (Spair s)'C]
    using sB
    by (cases Bs Es rule: two-lists-cases-snoc-Cons') auto
    from this obtain Zs where Zs:
    length Zs < length Cs
```

```
            Es@E# fst (Spair s) = Bs=C0## Zs @ [fst (Spair s)'C]
            by fast
    define Ys where Ys= fold fst (map Spair ts) \modelsZs
    with }Zs(2) hav
    fold fst (map Spair ts)\models(Es @ E # fst (Spair s) = Bs)=
        fold fst (map Spair ts)'C0 # Ys @ [fold fst (map Spair (s#ts))'C]
        by (simp add: image-comp)
    moreover
        have }\forallr\in\mathrm{ set ts. }\forall(f,g)\in\mathrm{ fundfoldpairs.
            r=Abs-induced-automorph fg\longrightarrowC0\inf\vdash\mathcal{C}
        using fundfoldpair-fundchamber-in-half-chamber-system-f
        by fast
    ultimately have
    fold fst (map Spair ts)}\models(Es@ E# fst (Spair s) | Bs)
            C0 # Ys @ [fold fst (map Spair (s#ts))'fold fst (map Spair rs)'C]
        using decomps(1) ss C0CsC-rs-f\mathcal{C fundfold-comp-trivial-f\mathcal{C}[of ts C0]}
            fundfold-comp-trivial-f\mathcal{C}[of rs C]
        by fastforce
    with decomps(1) ss obtain Xs where Xs:
        length Xs \leq length Ys
        pgallery (C0 # Xs @ [fold fst (map Spair ss)'C])
        using gal fundfold-comp-gallery-map[of Es @ E # fst (Spair s) \models Bs ts]
                gallery-obtain-pgallery[OF False[THEN not-sym]]
            by (fastforce simp add: image-comp)
        from Ys-def Xs(1) Zs(1) have length Xs < length Cs by simp
        with Xs(2) obtain n where (fold fst (map Spair ss) ^ (Suc n))' C=C0
            using step by (force simp add: image-comp funpow-Suc-right[THEN sym])
            thus ?thesis by fast
        qed
    qed
qed
lemma fundfold-comp-chamber-ex-funpow:
    assumes ss: set ss =S and C: chamber C
    shows \exists n. (fold fst (map Spair ss) ^n)'C=C0
proof (cases C=C0)
    case True
    hence (fold fst (map Spair ss)~0 0)`C=C0 by simp
    thus ?thesis by fast
next
    case False with fundchamber assms show ?thesis
        using chamber-pconnect[of C0 C] fundfold-comp-pgallery-ex-funpow
        by fastforce
qed
lemma fundfold-comp-fixespointwise-C0:
    assumes set ss \subseteqS
    shows fixespointwise (fold fst (map Spair ss)) C0
```

```
proof (rule fold-fixespointwise, rule ballI)
    fix fg assume fg\in set (map Spair ss)
    from this obtain s}\mathrm{ where sєset ss fg=Spair s by auto
    with assms
        have fg': OpposedThinChamberComplexFoldings X (fst fg) (snd fg) C0
        using Spair-fundfoldpair fundfoldpairs-def
        by fastforce
    show fixespointwise (fst fg) C0
        using OpposedThinChamberComplexFoldings.axioms(2)[OF fg']
            OpposedThinChamberComplexFoldings.chamber-DO[OF fg']
            OpposedThinChamberComplexFoldings.chambers(4)[OF fg}
            chamber-system-def
            ChamberComplexFolding.chamber-retraction1[of X fst fg C0]
        by auto
qed
lemma fundfold-comp-endomorphism:
    assumes set ss \subseteqS
    shows ChamberComplexEndomorphism X (fold fst (map Spair ss))
proof (rule fold-chamber-complex-endomorph-list, rule ballI)
    fix fg assume fg: fg \inset (map Spair ss)
    from this obtain s}\mathrm{ where sєset ss fg=Spair s by auto
    with assms show ChamberComplexEndomorphism X (fst fg)
        using Spair-fundfoldpair
                    OpposedThinChamberComplexFoldings.axioms(2)[of X]
                        ChamberComplexFolding.axioms(1)[of X]
                            ChamberComplexRetraction.axioms(1)[of X]
    unfolding fundfoldpairs-def
    by fastforce
qed
lemma finite-S: finite S
    using fundchamber-S-fundadjset fundchamber finite-adjacentset
    by (blast intro: inj-on-finite fundchamber-S-image-inj-on)
lemma ex-label-retraction: \exists\varphi. label-wrt C0 \varphi^ fixespointwise \varphi C0
proof-
    obtain ss where ss: set ss =S using finite-S finite-list by fastforce
    define fgs where fgs = map Spair ss
    - for fg s set fgs, have fst fg' D=C0 for some D f fundajdset
    define }\psi\mathrm{ where }\psi=\mathrm{ fold fst fgs
    define vdist where vdist v=(LEAST n. (\psi^^n) v\inC0) for v
    define }\varphi\mathrm{ where }\varphiv=(\mp@subsup{\psi}{}{~^}(vdist v)) v for 
    have label-wrt CO \varphi
    unfolding label-wrt-def
proof
```

```
        fix C assume C: C\in\mathcal{C}
        show bij-betw \varphi C C0
        proof -
        from \psi-def fgs-def ss C obtain m where m: ( }\mp@subsup{\psi}{}{~}m)\mp@subsup{}{}{`}\C=C
            using chamber-system-def fundfold-comp-chamber-ex-funpow by fastforce
        have }\v.v\inC\Longrightarrow(\mp@subsup{\psi}{}{^^m})v=\varphi
        proof-
            fix v}\mathrm{ assume v:v,C
            define n where n=(LEAST n. ( }\psi~~n)v\inC0
            from vm \varphi-def vdist-def n-def have m}\geqn\varphiv\inC
            using Least-le[of \lambdan. (\psi~n})v\inC0m
                    LeastI-ex[of \lambdan. (\psi^^n) v\inC0]
            by auto
        then show ( }\mp@subsup{\psi}{}{~}m)v=\varphi
            using ss \psi-def fgs-def \varphi-def vdist-def n-def funpow-add[of m-n n \psi]
                fundfold-comp-fixespointwise-C0
                funpower-fixespointwise fixespointwiseD
            by fastforce
        qed
        with C m ss \psi-def fgs-def show ?thesis
        using chamber-system-def fundchamber fundfold-comp-endomorphism
            ChamberComplexEndomorphism.funpower-endomorphism[of X]
                    ChamberComplexEndomorphism.bij-betw-chambers[of X]
                    bij-betw-cong[of C \psi~~m}\varphiC0
        by fastforce
        qed
    qed
    moreover from vdist-def \varphi-def have fixespointwise \varphi C0
        using Least-eq-0 by (fastforce intro: fixespointwiseI)
    ultimately show ?thesis by fast
qed
lemma ex-label-map: \exists\varphi. label-wrt C0 \varphi
    using ex-label-retraction by fast
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 $w \in W w^{6} \rightarrow C 0=C 0$

```
    shows w}=
proof-
    obtain \varphi where \varphi: label-wrt C0 \varphi using ex-label-map by fast
    have fixespointwise (permutation w) C0
        using W-respects-labels[OF \varphi assms(1)] chamberD-simplex[OF fundchamber]
            ChamberComplexEndomorphism.respects-label-fixes-chamber-imp-fixespointwise[
                OF W-endomorphism, OF assms(1) \varphi fundchamber assms(2)
            ]
    by fast
    with assms(1) show ?thesis
    using fundchamber W-automorphism trivial-automorphism
                standard-uniqueness-automorphs
                permutation-inject[of w 0]
    by (auto simp add: zero-permutation.rep-eq)
qed
lemma fundchamber-W-image-inj-on:
    inj-on ( }\lambdaw.\mp@subsup{w}{}{\prime}->C0)
proof (rule inj-onI)
    fix w w' assume ww': w\inW w'\inW w'->C0= 和'->C0
    from ww'(3) have (-\mp@subsup{w}{}{\prime}\mp@subsup{)}{}{\prime}->\mp@subsup{w}{}{\prime}->C0=(-\mp@subsup{w}{}{\prime}\mp@subsup{)}{}{\prime}->\mp@subsup{w}{}{\prime}->CO by simp
    with ww'(1,2) show w= w'
            using fundchamber-W-image-ker[of -w'+w] add.assoc[of w' -w'w]
            by (simp add:
                image-comp plus-permutation.rep-eq[THEN sym]
                    zero-permutation.rep-eq genby-uminus-add-closed
            )
qed
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 $\Rightarrow{ }^{\prime} a$
where fundantivertex $s \equiv\left(\right.$ THE v. $\left.v \in C 0-s^{s} \rightarrow C 0\right)$
abbreviation fundantipermutation $\equiv$ the-inv-into $S$ fundantivertex
lemma fundantivertex: $s \in S \Longrightarrow$ fundantivertex $s \in C 0-s^{4} \rightarrow C 0$
using fundchamber-S-adjacent[of s]
fundchamber-S-image-neq-fundchamber [of $s]$
fundantivertex-def[of s] the $I^{\prime}[$ OF adj-antivertex]
by auto
lemma fundantivertex-fundchamber-decomp:
$s \in S \Longrightarrow C 0=$ insert (fundantivertex $s)\left(C 0 \cap s^{\iota} \rightarrow C 0\right)$
using fundchamber-S-adjacent [of s]
fundchamber-S-image-neq-fundchamber[of s]
fundantivertex[of s] adjacent-conv-insert[of C0]
by auto
lemma fundantivertex-unstable:
$s \in S \Longrightarrow s \rightarrow$ fundantivertex $s \neq$ fundantivertex $s$
using fundantivertex-fundchamber-decomp[of $s$ ] image-insert $\left[\right.$ of $(\rightarrow)$ s fundantivertex s $\left.C 0 \cap s^{〔} \rightarrow C 0\right]$ S-fixes-fundchamber-image-int fundchamber-S-image-neq-fundchamber
by fastforce
lemma fundantivertex-inj-on: inj-on fundantivertex $S$
proof (rule inj-onI)
fix $s t$ assume st: $s \in S$ t fundantivertex $s=$ fundantivertex $t$
hence insert (fundantivertex s) $\left(C 0 \cap s^{\natural} \rightarrow C 0\right)=$ insert (fundantivertex s) $\left(C 0 \cap t^{4} \rightarrow C 0\right)$
using fundantivertex-fundchamber-decomp [of $s]$ fundantivertex-fundchamber-decomp[of $t$ ]
by auto
moreover from st
have fundantivertex $s \notin C 0 \cap s^{\iota} \rightarrow C 0$ fundantivertex $s \notin C 0 \cap t^{\iota} \rightarrow C 0$
using fundantivertex[of $s$ ] fundantivertex[of $t$ ]
by auto
ultimately have $C 0 \cap s^{\iota} \rightarrow C 0=C 0 \cap t^{s} \rightarrow C 0$
using insert-subset-equality[of fundantivertex s] by simp
with $s t(1,2)$ show $s=t$
using fundchamber fundchamber-S-chamber[of s] fundchamber-S-chamber[of t]
fundfacets [of s] fundfacets(2)[of t]
fundchamber-S-image-neq-fundchamber[of s]
fundchamber-S-image-neq-fundchamber[of $t]$

genby-genset-closed $[$ of - S]
inj-onD[OF fundchamber-W-image-inj-on, of $s t]$
by auto
qed
lemma fundantivertex-surj-on: fundantivertex ' $S=C 0$
proof (rule seteqI)
show $\bigwedge v . v \in$ fundantivertex ' $S \Longrightarrow v \in C 0$ using fundantivertex by fast next
fix $v$ assume $v: v \in C 0$

```
    define D where D= the-adj-chamber C0 (C0-{v})
    with v have D\infundadjset
    using fundchamber facetrel-diff-vertex the-adj-chamber-adjacentset
        the-adj-chamber-neq
    by fastforce
    from this obtain s where s: s\inS D=s'->C0
    using fundadjset-eq-S-image by blast
    with vD-def [abs-def] have fundantivertex s=v
    using fundchamber fundchamber-S-adjacent
        fundchamber-S-image-neq-fundchamber[of s]
        facetrel-diff-vertex[of v C0]
        the-adj-chamber-facet facetrel-def[of C0-{v} D]
    unfolding fundantivertex-def
    by (force intro: the1-equality[OF adj-antivertex])
    with }s(1)\mathrm{ show }v\in\mathrm{ fundantivertex ' }S\mathrm{ by fast
qed
lemma fundantivertex-bij-betw: bij-betw fundantivertex S C0
    unfolding bij-betw-def
    using fundantivertex-inj-on fundantivertex-surj-on
    by fast
lemma card-S-fundchamber: card S = card C0
    using bij-betw-same-card[OF fundantivertex-bij-betw] by fast
lemma card-S-chamber:
    chamber C \Longrightarrow card C = card S
    using fundchamber chamber-card[of C0 C] card-S-fundchamber by auto
lemma fundantipermutation1:
    v \in C 0 \Longrightarrow \text { fundantipermutation v} \in S
    using fundantivertex-surj-on the-inv-into-into[OF fundantivertex-inj-on] by blast
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:
chamber $C \Longrightarrow z \triangleleft C \Longrightarrow$
$\exists D E . D \in X-\{C\} \wedge z \triangleleft D \wedge E \in X-\{C, D\} \wedge z \triangleleft E$
begin
definition some-third-chamber :: 'a set $\Rightarrow$ 'a set $\Rightarrow$ 'a set $\Rightarrow{ }^{\prime}$ 'a set where some-third-chamber $C D z \equiv S O M E E$. $E \in X-\{C, D\} \wedge z \triangleleft E$

```
lemma facet-ex-third-chamber: chamber \(C \Longrightarrow z \triangleleft C \Longrightarrow \exists E \in X-\{C, D\} . z \triangleleft E\)
    using thick[of \(C z]\) by auto
lemma some-third-chamberD-facet:
    chamber \(C \Longrightarrow z \triangleleft C \Longrightarrow z \triangleleft\) some-third-chamber C D z
    using facet-ex-third-chamber \([\) of \(C z D]\) someI-ex[of \(\lambda E . E \in X-\{C, D\} \wedge z \triangleleft E]\)
        some-third-chamber-def
    by auto
lemma some-third-chamberD-simplex:
    chamber \(C \Longrightarrow z \triangleleft C \Longrightarrow\) some-third-chamber \(C D z \in X\)
    using facet-ex-third-chamber[of \(C z D]\) someI-ex \([\) of \(\lambda E . E \in X-\{C, D\} \wedge z \triangleleft E]\)
        some-third-chamber-def
    by auto
lemma some-third-chamberD-adj:
    chamber \(C \Longrightarrow z \triangleleft C \Longrightarrow C \sim\) some-third-chamber \(C D z\)
    using some-third-chamberD-facet by (fast intro: adjacentI)
lemma chamber-some-third-chamber:
    chamber \(C \Longrightarrow z \triangleleft C \Longrightarrow\) chamber (some-third-chamber C D z)
    using chamber-adj some-third-chamberD-simplex some-third-chamberD-adj
    by fast
lemma some-third-chamberD-ne:
    assumes chamber \(C z \triangleleft C\)
    shows some-third-chamber C \(D z \neq C\) some-third-chamber C \(D z \neq D\)
    using assms facet-ex-third-chamber[of C z D]
        someI-ex \([\) of \(\lambda E . E \in X-\{C, D\} \wedge z \triangleleft E]\) some-third-chamber-def
    by auto
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\inS\Longrightarrow add-order s=2
begin
```

```
abbreviation W\equiv\langleS\rangle
```

abbreviation W\equiv\langleS\rangle
abbreviation S-length \equiv word-length S
abbreviation S-length \equiv word-length S
abbreviation S-reduced-for }\equiv\mathrm{ reduced-word-for S
abbreviation S-reduced-for }\equiv\mathrm{ reduced-word-for S
abbreviation S-reduced \equiv reduced-word S
abbreviation S-reduced \equiv reduced-word S
abbreviation relfun \equiv \st. add-order (s+t)
abbreviation relfun \equiv \st. add-order (s+t)
lemma no-zero-genset: }0\not\in
lemma no-zero-genset: }0\not\in
proof
proof
assume 0\inS
assume 0\inS
moreover have add-order ( }0:\mp@subsup{:}{}{\prime}w)=1\mathrm{ using add-order0 by fast
moreover have add-order ( }0:\mp@subsup{:}{}{\prime}w)=1\mathrm{ using add-order0 by fast
ultimately show False using genset-order2 by simp
ultimately show False using genset-order2 by simp
qed
qed
lemma genset-order2-add: s\inS\Longrightarrows+s=0
lemma genset-order2-add: s\inS\Longrightarrows+s=0
using add-order[of s] by (simp add: genset-order2 nataction-2)
using add-order[of s] by (simp add: genset-order2 nataction-2)
lemmas genset-uminus = minus-unique[OF genset-order2-add]
lemmas genset-uminus = minus-unique[OF genset-order2-add]
lemma relfun-S: s\inS\Longrightarrow relfun s s = 1
lemma relfun-S: s\inS\Longrightarrow relfun s s = 1
using add-order-relator[of s] by (auto simp add: genset-order2 nataction-2)
using add-order-relator[of s] by (auto simp add: genset-order2 nataction-2)
lemma relfun-eq1:\llbrackets\inS; relfun st=1\rrbracket\Longrightarrowt=s
lemma relfun-eq1:\llbrackets\inS; relfun st=1\rrbracket\Longrightarrowt=s
using add-order-add-eq1 genset-uminus by fastforce
using add-order-add-eq1 genset-uminus by fastforce
lemma S-relator-list: s\inS\Longrightarrow pair-relator-list s s = [s,s]
lemma S-relator-list: s\inS\Longrightarrow pair-relator-list s s = [s,s]
using relfun-S alternating-list2 by simp
using relfun-S alternating-list2 by simp
lemma S-sym:T\subseteqS\Longrightarrowuminus' }T\subseteq
lemma S-sym:T\subseteqS\Longrightarrowuminus' }T\subseteq
using genset-uminus by auto
using genset-uminus by auto
lemmas special-subgroup-eq-sum-list =
lemmas special-subgroup-eq-sum-list =
genby-sym-eq-sum-lists[OF S-sym]
genby-sym-eq-sum-lists[OF S-sym]
lemmas genby-S-reduced-word-for-arg-min =
lemmas genby-S-reduced-word-for-arg-min =
reduced-word-for-genby-sym-arg-min[OF S-sym]
reduced-word-for-genby-sym-arg-min[OF S-sym]
lemmas in-genby-S-reduced-letter-set =
lemmas in-genby-S-reduced-letter-set =
in-genby-sym-imp-in-reduced-letter-set[OF S-sym]
in-genby-sym-imp-in-reduced-letter-set[OF S-sym]
end

```
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(\bigcupT\inPow S. (\bigcupw\inW.{w+o\langleT\rangle}))
abbreviation \mathcal{P}\equivspecial-cosets
lemma special-cosetsI:T\inPow S\Longrightarroww\inW\Longrightarroww+o }\langleT\rangle\in\mathcal{P
    using special-cosets-def by auto
lemma special-coset-singleton: w\inW\Longrightarrow{w}\in\mathcal{P}
    using special-cosetsI genby-lcoset-empty[of w] by fastforce
lemma special-coset-nempty: X\in\mathcal{P}\LongrightarrowX\not={}
    using special-cosets-def genby-lcoset-refl by fastforce
lemma special-subgroup-special-coset: T\inPow S\Longrightarrow\langleT\rangle\in\mathcal{P}
    using genby-0-closed special-cosetsI[of T] by fastforce
lemma special-cosets-lcoset-closed: w\inW\LongrightarrowX\in\mathcal{P}\Longrightarroww+oX\in\mathcal{P}
    using genby-add-closed unfolding special-cosets-def
    by (fastforce simp add: set-plus-rearrange2)
lemma special-cosets-lcoset-shift: w\inW\Longrightarrow((+o)w)'\mathcal{P}=\mathcal{P}
    using special-cosets-lcoset-closed genby-uminus-closed
    by (force simp add: set-plus-rearrange2)
lemma special-cosets-has-bottom: supset-has-bottom \mathcal{P}
proof (rule ordering.has-bottomI, rule supset-poset)
    show W\in\mathcal{P}\mathrm{ using special-subgroup-special-coset by fast}\\mp@code{l}
next
    fix }X\mathrm{ assume X: X }\in\mathcal{P
    from this obtain wT where wT: w\inW T\inPow S X=w +o \langleT\rangle
        using special-cosets-def by auto
    thus }X\subseteqW\mathrm{ using genby-mono[of T] genby-lcoset-closed[of w] by auto
qed
```

```
lemma special-cosets-bottom: supset-bottom \mathcal{P}=W
proof (rule supset-bottomI[THEN sym])
    fix }X\mathrm{ assume }X\in\mathcal{P
    from this obtain w T where w\inW T\inPow S X =w +o \langleT\rangle
        using special-cosets-def by auto
    thus }X\subseteq
        using genby-mono[of T S] set-plus-mono[of \langleT\rangle W] genby-lcoset-el-reduce
        by force
qed (auto simp add: special-subgroup-special-coset)
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 Group WithGeneratorsRelators. 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 $R::$ ' $w$ list set where $R \equiv(\bigcup s \in S$. $\bigcup t \in S$. \{pair-relator-list st $t$ )
abbreviation $P \equiv$ map (charpair $S$ )' $R$
abbreviation $P^{\prime} \equiv$ GroupWithGeneratorsRelators. $P^{\prime} S R$
abbreviation $Q \equiv$ GroupWithGeneratorsRelators. $Q S R$
abbreviation $G \equiv$ GroupWithGeneratorsRelators.G S $R$
abbreviation relator-freeword $\equiv$
Group WithGeneratorsRelators.relator-freeword $S$
abbreviation pair-relator-freeword $:: ~ ' w \Rightarrow{ }^{\prime} w \Rightarrow{ }^{\prime} w$ freeword where pair-relator-freeword $s t \equiv$ Abs-freelist (pair-relator-list st)
abbreviation freeliftid $\equiv$ freeword-funlift id
abbreviation induced-id :: ' $w$ freeword set permutation $\Rightarrow$ ' $w$ where induced-id $\equiv$ GroupWithGeneratorsRelators.induced-id $S R$
lemma $S$-relator-freeword: $s \in S \Longrightarrow$ pair-relator-freeword $s s=s[+] s$ by (simp add: S-relator-list Abs-freeletter-add)
lemma map-charpair-map-pairtrue-R:
$s \in S \Longrightarrow t \in S \Longrightarrow$ map (charpair $S$ ) (pair-relator-list st) = map pairtrue (pair-relator-list st) using set-alternating-list map-charpair-uniform by fastforce
lemma relator-freeword:
$s \in S \Longrightarrow t \in S \Longrightarrow$
pair-relator-freeword st=relator-freeword (pair-relator-list st)
using set-alternating-list
arg-cong[OF map-charpair-map-pairtrue-R, of st Abs-freeword]
by fastforce
lemma relator-freewords: Abs-freelist' $R=P^{\prime}$
using relator-freeword by force
lemma GroupWithGeneratorsRelators-S-R: GroupWithGeneratorsRelators $S R$ proof
fix $r s$ assume $r s: r s \in R$
hence $r s^{\prime}: r s \in$ lists $S$ using set-alternating-list by fast
from $r s$ ' show $r s \in$ lists $(S \cup$ uminus ' $S$ ) by fast
from $r$ show sum-list rs $=0$ using sum-list-pair-relator-list by fast
from $r s^{\prime}$ show proper-signed-list (map (charpair $S$ ) rs)
using proper-signed-list-map-uniform-snd arg-cong[of map (charpair S) rs map pairtrue rs proper-signed-list]
by fastforce
qed
lemmas GroupByPresentation-S-P =
Group WithGeneratorsRelators.GroupByPresentation-S-P [
OF Group WithGeneratorsRelators-S-R
]
lemmas $Q$-Free $S=$ GroupByPresentation.Q-FreeS[OF GroupByPresentation-S-P]
lemma relator-freeword- $Q: s \in S \Longrightarrow t \in S \Longrightarrow$ pair-relator-freeword s $t \in Q$ using relator-freeword

GroupByPresentation.relators[OF GroupByPresentation-S-P]
by fastforce
lemmas $P^{\prime}$-Free $S=$
Group WithGeneratorsRelators. $P^{\prime}$-FreeS $[$
OF Group WithGeneratorsRelators-S-R
]
lemmas GroupByPresentationInducedFun-S-P-id $=$
Group WithGeneratorsRelators.GroupByPresentationInducedFun-S-P-id[ OF Group WithGeneratorsRelators-S-R
]
lemma rconj-relator-freeword:
$\llbracket s \in S ; t \in S$; proper-signed-list xs; fst'set xs $\subseteq S \rrbracket \Longrightarrow$ rconjby (Abs-freeword xs) (pair-relator-freeword st) $\in Q$
using GroupWithGeneratorsRelators.rconj-relator-freeword [
OF GroupWithGeneratorsRelators-S-R
]
relator-freeword
by force

```
lemma lconjby-Abs-freelist-relator-freeword:
    \llbracket s\inS; t\inS; xs\inlists S\rrbracket\Longrightarrow
    lconjby (Abs-freelist xs) (pair-relator-freeword st)\inQ
    using GroupWithGeneratorsRelators.lconjby-Abs-freelist-relator-freeword[
        OF GroupWithGeneratorsRelators-S-R
        ]
        relator-freeword
    by force
lemma Abs-freelist-rev-append-alternating-list-in-Q:
    assumes s\inS t\inS
    shows Abs-freelist (rev (alternating-list n st)@ alternating-list nst)\inQ
proof (induct n)
    case (Suc m)
    define }u\mathrm{ where }u=(if even m then s else t
    define x where x=Abs-freelist (rev (alternating-list m st)@ alternating-list
mst)
    from u-def x-def assms have
        Abs-freelist (rev (alternating-list (Suc m) st)@
            alternating-list (Suc m) st)=
            (pair-relator-freeword u u) + rconjby (Abs-freeletter u) x
    using Abs-freelist-append[of
                u # rev(alternating-list m s t)@ alternating-list m s t
                    [u]
            ]
            Abs-freelist-Cons[of
                u
                rev(alternating-list m s t)@ alternating-list m s t
            ]
    by (simp add: add.assoc[THEN sym] S-relator-freeword)
    moreover from Suc assms u-def x-def have rconjby (Abs-freeletter u)x\inQ
    using Abs-freeletter-in-FreeGroup-iff[of - S]
            FreeGroup-genby-set-lconjby-set-rconjby-closed
    by fastforce
    ultimately show ?case
    using u-def assms relator-freeword-Q genby-add-closed by fastforce
qed (simp add: zero-freeword.abs-eq[THEN sym] genby-0-closed)
lemma Abs-freeword-freelist-uminus-add-in-Q:
    proper-signed-list xs \Longrightarrowfst'set xs}\subseteqS
    - Abs-freelistfst xs + Abs-freeword xs }\in
proof (induct xs)
    case (Cons x xs)
    from Cons(2) have 1:
    - Abs-freelistfst (x#xs) + Abs-freeword (x#xs) =
    -Abs-freelistfst xs + -Abs-freeletter (fst x)
                        + Abs-freeword [x] + Abs-freeword xs
    using Abs-freelist-Cons[of fst x map fst xs]
    by (simp add: Abs-freeword-Cons[THEN sym] add.assoc minus-add)
```

```
show ?case
proof (cases snd x)
    case True
    with Cons show ?thesis
        using 1
        by (simp add:
                                Abs-freeletter-prod-conv-Abs-freeword
                        binrelchain-Cons-reduce
                )
next
    case False
    define s where s=fst x
    with Cons(3) have s-S: s\inS by simp
    define q}\mathrm{ where q= rconjby (Abs-freelistfst xs) (pair-relator-freeword s s)
    from s-def False Cons(3) have
        - Abs-freelistfst (x#xs) + Abs-freeword (x#xs) =
            -Abs-freelistfst xs + -pair-relator-freeword s s + Abs-freeword xs
        using 1 surjective-pairing[of x] S-relator-freeword[of s]
                uminus-Abs-freeword-singleton[of s False, THEN sym]
            by (simp add: add.assoc)
    with q-def have 2:
        - Abs-freelistfst (x#xs) + Abs-freeword (x#xs) =
            -q+(-Abs-freelistfst xs + Abs-freeword xs)
        by (simp add: rconjby-uminus[THEN sym] add.assoc[THEN sym])
    moreover from q-def s-def Cons(3) have - q\inQ
        using proper-signed-list-map-uniform-snd[of True map fst xs]
            rconj-relator-freeword genby-uminus-closed
        by fastforce
    moreover from Cons have -Abs-freelistfst xs + Abs-freeword xs }\in
        by (simp add: binrelchain-Cons-reduce)
    ultimately show ?thesis using genby-add-closed by simp
qed
qed (simp add: zero-freeword.abs-eq[THEN sym] genby-0-closed)
lemma Q-freelist-freeword':
    \llbracket proper-signed-list xs; fst'set xs \subseteqS; Abs-freelistfst xs }\inQ\rrbracket
    Abs-freeword xs }\in
    using Abs-freeword-freelist-uminus-add-in-Q genby-add-closed
by fastforce
lemma \(Q\)-freelist-freeword:
    c}\in\mathrm{ FreeGroup S ב Abs-freelist (map fst (freeword c)) }\inQ\Longrightarrowc\in
using freeword FreeGroupD Q-freelist-freeword' freeword-inverse[of c]
by fastforce
```

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 $x s \Longrightarrow f s t$ 'set $x s \subseteq S \Longrightarrow$

$$
\text { freeliftid }(\text { Abs-freeword } x s)=\text { sum-list }(\text { map fst xs })
$$

using freeword-funlift-Abs-freeword[of xs id] genset-uminus
sum-list-map-cong[of xs apply-sign id fst]
by fastforce
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:

$$
s \in S \Longrightarrow t \in S \Longrightarrow
$$

sum-list (pair-relator-halfist st) $=$ sum-list (pair-relator-halfist $t s)$
using add-order [of $s+t$ ] genset-order2-add alternating-order2-even-cancel-right[of st2*(relfun st)]
by (simp add: alternating-sum-list-conv-nataction add-order-add-sym)
definition flip-altsublist-adjacent $::$ ' $w$ list $\Rightarrow$ ' $w$ list $\Rightarrow$ bool
where flip-altsublist-adjacent ss ts

$$
\begin{aligned}
& \equiv \exists s t \text { as bs. ss }=\text { as @ (pair-relator-halfist } s t) @ b s \wedge \\
& t s=a s @(\text { pair-relator-halfist } t s) @ b s
\end{aligned}
$$

abbreviation flip-altsublist-chain $\equiv$ binrelchain flip-altsublist-adjacent
lemma fip-altsublist-adjacentI:
$s s=a s @($ pair-relator-halfist $s t) @ b s \Longrightarrow$ $t s=a s @($ pair-relator-halfist $t s) @ b s \Longrightarrow$ flip-altsublist-adjacent ss ts
using fip-altsublist-adjacent-def by fast
lemma flip-altsublist-adjacent-Cons-grow:
assumes flip-altsublist-adjacent ss ts
shows flip-altsublist-adjacent (a\#ss) (a\#ts)
proof-
from assms obtain $s t$ as bs where ssts: ss =as @ (pair-relator-halfist st) @bs $t s=a s @($ pair-relator-halfist $t s) @ b s$ using flip-altsublist-adjacent-def
by auto
from ssts have
$a \# s s=(a \# a s) @($ pair-relator-halflist $s t) @ b s$ $a \# t s=(a \# a s) @($ pair-relator-halflist $t s) @ b s$

```
    by auto
    thus ?thesis by (fast intro: flip-altsublist-adjacentI)
qed
lemma flip-altsublist-chain-map-Cons-grow:
    flip-altsublist-chain tss \Longrightarrow flip-altsublist-chain (map ((#) t) tss)
    by (induct tss rule: list-induct-CCons)
        (auto simp add:
        binrelchain-Cons-reduce[of flip-altsublist-adjacent]
        flip-altsublist-adjacent-Cons-grow
    )
lemma flip-altsublist-adjacent-refl:
    ss }\not=[]\Longrightarrow\mathrm{ ss lists S \ flip-altsublist-adjacent ss ss
proof (induct ss rule: list-nonempty-induct)
    case (single s)
    hence [s]= [] @ pair-relator-halflist s s @ []
            using relfun-S by simp
    thus ?case by (fast intro: flip-altsublist-adjacentI)
next
    case cons thus ?case using flip-altsublist-adjacent-Cons-grow by simp
qed
lemma flip-altsublist-adjacent-sym:
    flip-altsublist-adjacent ss ts \Longrightarrow flip-altsublist-adjacent ts ss
    using flip-altsublist-adjacent-def flip-altsublist-adjacentI by auto
lemma rev-flip-altsublist-chain:
    flip-altsublist-chain xss \Longrightarrow flip-altsublist-chain (rev xss)
    using flip-altsublist-adjacent-sym binrelchain-snoc[of flip-altsublist-adjacent]
    by (induct xss rule: list-induct-CCons) auto
lemma flip-altsublist-adjacent-set:
    assumes ss\inlists S flip-altsublist-adjacent ss ts
    shows set ts = set ss
proof-
    from assms obtain s t as bs where ssts:
        ss=as@ (pair-relator-halfist st)@bs
        ts=as@ (pair-relator-halflist t s)@ bs
        using flip-altsublist-adjacent-def
        by auto
    with assms(1) show ?thesis
        using set-alternating-list2[of relfun stst]
            set-alternating-list2[of relfun t sts]
            add-order-add-sym[of t s] relfun-eq1
    by (cases relfun st rule: nat-cases-2Suc) auto
qed
lemma flip-altsublist-adjacent-set-ball:
```

```
    \forallss\inlists S. }\forall\mathrm{ ts. flip-altsublist-adjacent ss ts }\longrightarrow\mathrm{ set ts = set ss
    using flip-altsublist-adjacent-set by fast
lemma flip-altsublist-adjacent-lists:
    ss }\in\mathrm{ lists S ב flip-altsublist-adjacent ss ts }\Longrightarrow\mathrm{ ts }\in\mathrm{ lists S
    using flip-altsublist-adjacent-set by fast
lemma flip-altsublist-adjacent-lists-ball:
    *sslists S. }\forall\mathrm{ ts. flip-altsublist-adjacent ss ts }\longrightarrow\mathrm{ ts }\in\mathrm{ lists S
    using flip-altsublist-adjacent-lists by fast
lemma flip-altsublist-chain-lists:
    ss }\in\mathrm{ lists S C flip-altsublist-chain (ss#xss@[ts]) ב ts }\in\mathrm{ lists S
    using flip-altsublist-adjacent-lists
        binrelchain-propcong-Cons-snoc[of
        \lambdass. ss\inlists S flip-altsublist-adjacent ss xss ts
            ]
    by fast
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 \Longrightarrow length ts = length ss
    unfolding flip-altsublist-adjacent-def
    by (auto simp add: add-order-add-sym length-alternating-list)
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-
    from assms(2) obtain st as bs where stasbs:
    ss=as@ (pair-relator-halflist st)@bs
    ts=as@ (pair-relator-halflist t s)@ bs
    using flip-altsublist-adjacent-def
    by auto
show ?thesis
proof (cases relfun st)
    case 0 thus ?thesis using stasbs by (simp add: add-order-add-sym)
```

```
    next
        case Suc
        with assms stasbs have s\inS t\inS
            using set-alternating-list1[of add-order (s+t) s t]
                    set-alternating-list1[of add-order (t+s)ts]
                    add-order-add-sym[of t]
                    flip-altsublist-adjacent-lists[of ss ts]
        by auto
    with stasbs show ?thesis
        using sum-list-pair-relator-halflist-flip by simp
    qed
qed
lemma flip-altsublist-adjacent-sum-list-ball:
    \forallss\inlists S. \forallts. flip-altsublist-adjacent ss ts \longrightarrow sum-list ts = sum-list ss
    using flip-altsublist-adjacent-sum-list by fast
lemma S-reduced-forI-flip-altsublist-adjacent:
    S-reduced-for w ss \Longrightarrow flip-altsublist-adjacent ss ts \LongrightarrowS-reduced-for w ts
    using reduced-word-for-lists[of S] reduced-word-for-sum-list
        flip-altsublist-adjacent-lists flip-altsublist-adjacent-sum-list
            flip-altsublist-adjacent-length
    by (fastforce intro: reduced-word-forI-compare)
lemma flip-altsublist-adjacent-in-Q':
    fixes as bs st
    defines xs:xs \equivas @ pair-relator-halfist s t @ bs
    and ys:ys \equivas @ pair-relator-halfist t s @ bs
    assumes Axs: Abs-freelist xs \inQ
    shows Abs-freelist ys }\in
proof-
    define X Y A B half-st half2-st half-ts
        where }X=Abs\mathrm{ -freelist xs
            and Y = Abs-freelist ys
            and A=Abs-freelist as
            and B=Abs-freelist bs
            and half-st = Abs-freelist (pair-relator-halflist s t)
            and half2-st =Abs-freelist (pair-relator-halfist2 st)
            and half-ts = Abs-freelist (pair-relator-halfist t s)
define }z\mathrm{ where z=-half2-st + B
define w1 w2 where w1 = rconjby z (pair-relator-freeword st)
    and w2 = Abs-freelist (rev (pair-relator-halfist t s)@ pair-relator-halfist t s)
    define w3 where w3 = rconjby B w2
    from w1-def z-def
        have w1':w1 = rconjby B (lconjby half2-st (pair-relator-freeword s t)}
        by (simp add: rconjby-add)
    hence -w1 = rconjby B (lconjby half2-st (-pair-relator-freeword s t))
        using lconjby-uminus[of half2-st] by (simp add: rconjby-uminus[THEN sym])
```

moreover from $X$-def xs $A$-def half-st-def $B$-def have $X=A+B+$ rconjby $B$ half-st
by (simp add:
Abs-freelist-append-append[THEN sym] add.assoc[THEN sym] )
ultimately have

$$
X+-w 1=A+B+
$$

( rconjby B (half-st $+($ half2-st + -pair-relator-freeword $s t-h a l f 2-s t))$ )
by (simp add: add.assoc add-rconjby)
moreover from w2-def half2-st-def half-ts-def have w2 $=$ half2-st + half-ts
by (simp add:
Abs-freelist-append [THEN sym]
pair-relator-halfist2-conv-rev-pair-relator-halfist )
ultimately have
$X+-w 1+w 3=A+B+($ rconjby $B(-$ half2-st $+($ half2-st + half-ts $)))$
using half-st-def halfo-st-def w3-def add-assoc 4 [
of half-st half2-st - pair-relator-freeword st-half2-st ]
by (simp add:
Abs-freelist-append[THEN sym] pair-relator-halfist-append add.assoc add-rconjby
)
hence $Y^{\prime}: Y=X-w 1+w 3$
using $A$-def half-ts-def $B$-def ys $Y$-def
by (simp add: add.assoc[THEN sym] Abs-freelist-append-append[THEN sym]
)
from Axs have $x s$-S: xs $\in$ lists $S$ using $Q$-FreeS FreeGroupD-transfer' by fast have $w 1 \in Q \wedge w 3 \in Q$
proof (cases relfun $s t$ )
case 0 with $w 1$-def w2-def w3-def show ?thesis using genby- 0 -closed
by (auto simp add:
zero-freeword.abs-eq[THEN sym]
add-order-add-sym
)
next
case (Suc m) have m: add-order $(s+t)=S u c m$ by fact
have st: $\{s, t\} \subseteq S$
proof (cases m)
case 0 with $m$ xs xs- $S$ show ?thesis
using set-alternating-list1 relfun-eq1 by force
next
case Suc with $m$ xs xs-S show ?thesis
using set-alternating-list2 $[$ of add-order $(s+t) s t]$ by fastforce
qed
from $x s$ xs-S $B$-def have $B$-S: $B \in$ FreeGroup $S$
using Abs-freelist-in-FreeGroup [of bs $S$ ] by simp
moreover from w2-def have $w 2 \in Q$
using st Abs-freelist-rev-append-alternating-list-in-Q[of $t$ s add-order $(t+s)]$
by fast
ultimately have $w 3 \in Q$
using w3-def FreeGroup-genby-set-lconjby-set-rconjby-closed by fast
moreover from half2-st-def have $w 1 \in Q$
using $w 1$ 'st $B$-S alternating-list-in-lists[of s $S$ ] alternating-list-in-lists[of $t S]$ lconjby-Abs-freelist-relator-freeword[ of st]
by (force intro: FreeGroup-genby-set-lconjby-set-rconjby-closed)
ultimately show ?thesis by fast
qed
with $X$-def $Y$-def Axs show ?thesis
using $Y^{\prime}$ genby-diff-closed[of $\left.X\right]$ genby-add-closed $[$ of $X-w 1-w 3]$ by simp

## qed

lemma flip-altsublist-adjacent-in- $Q$ :
Abs-freelist ss $\in Q \Longrightarrow$ flip-altsublist-adjacent ss $t s \Longrightarrow$ Abs-freelist ts $\in Q$
using flip-altsublist-adjacent-def flip-altsublist-adjacent-in- $Q^{\prime}$ by auto
lemma flip-altsublist-chain-G-in-Q:
$\llbracket$ Abs-freelist ss $\in Q$; flip-altsublist-chain $(s s \# x s s @[t s]) \rrbracket \Longrightarrow$ Abs-freelist $t s \in Q$
using flip-altsublist-adjacent-in- $Q$
binrelchain-propcong-Cons-snoc [of
$\lambda s s$. Abs-freelist ss $\in Q$
flip-altsublist-adjacent
]
by fast
lemma alternating-S-no-flip:
assumes $s \in S t \in S n>0 n<$ relfun $s t \vee$ relfun $s t=0$
shows sum-list (alternating-list $n s t) \neq$ sum-list (alternating-list $n t s$ )
proof
assume sum-list (alternating-list $n s t)=\operatorname{sum}-l i s t($ alternating-list $n t s)$
hence sum-list (alternating-list $n s t)+-$ sum-list (alternating-list $n t s)=0$
by $\operatorname{simp}$
with $\operatorname{assms}(1,2)$ have sum-list (alternating-list $(2 * n) s t)=0$
by (cases even $n$ )
(auto simp add:
genset-order2-add uminus-sum-list-alternating-order2
sum-list.append [THEN sym]
alternating-list-append mult-2
)
with $\operatorname{assms}(3,4)$ less-add-order-eq-0-contra add-order-eq0 show False
by (auto simp add: alternating-sum-list-conv-nataction)
qed
lemma exchange-alternating-not-in-alternating:

```
    assumes \(n \geq 2 n<\) relfun \(s t \vee\) relfun \(s t=0\)
    S-reduced-for \(w\) (alternating-list nst @ cs)
    alternating-list nst@cs=xs@[x]@ys S-reduced-for \(w(t \# x s @ y s)\)
    shows length \(x s \geq n\)
proof-
    from assms(1) obtain \(m k\) where \(n: n=S u c m\) and \(m: m=S u c k\)
        using gr0-implies-Suc by fastforce
    define altnst altnts altmts altkst
    where altnst \(=\) alternating-list \(n\) s \(t\)
    and altnts \(=\) alternating-list \(n t s\)
    and altmts \(=\) alternating-list \(m t s\)
    and altkst \(=\) alternating-list \(k s t\)
    from altnst-def altmts-def \(n\) have altnmst: altnst \(=s \#\) altmts
    using alternating-list-Suc-Cons[of m] by fastforce
    with assms(3) altnst-def have \(s\)-S: \(s \in S\) using reduced-word-for-lists by fastforce
    from assms(5) have \(t\)-S: \(t \in S\) using reduced-word-for-lists by fastforce
    from \(m\) altnmst altmts-def altkst-def have altnkst: altnst \(=s \# t \#\) altkst
        using alternating-list-Suc-Cons by fastforce
    have \(\neg\) length \(x s<n\)
    proof (cases Suc (length xs) \(=n\) )
    case True
    with assms \((4,5) n\) altnts-def have flip: S-reduced-for \(w(\) altnts @ cs)
        using length-alternating-list[of \(n \mathrm{~s} t]\)
            alternating-list-Suc-Cons[of mts]
        by auto
    from altnst-def have sum-list altnst \(=\) sum-list altnts
        using reduced-word-for-sum-list[OF assms(3)]
                reduced-word-for-sum-list[OF flip]
        by auto
    with \(n\) assms(2) altnst-def altnts-def show ?thesis
        using alternating-S-no-flip \([O F s-S t-S]\) by fast
next
    case False show ?thesis
    proof (cases xs ys rule: two-lists-cases-snoc-Cons)
        case Nil1
        from Nil1 (1) assms(4) altnkst altnst-def have ys \(=t \#\) altkst @ cs by auto
        with Nil1 (1) assms(5) show ?thesis
                using \(t\)-S genset-order2-add[of \(t\) ]
                    contains-order2-nreduced[of tS[]altkst@cs]
                    reduced-word-for-imp-reduced-word
        by force
    next
        case Nil2 with assms(4) altnst-def False show ?thesis
                using length-append[of altnst cs]
                by (fastforce simp add: length-alternating-list)
    next
        case (snoc-Cons us \(u z z s\) )
        with assms \((4,5)\) altnst-def
            have 1: altnst @ cs =us@[u,x,z]@zs S-reduced-for w (t\#us@ \([u, z] @ z s)\)
```

```
            by auto
            from 1(1) snoc-Cons(1) False altnst-def show ?thesis
            using take-append[of n altnst cs] take-append[of n us@[u,x,z]zs]
                set-alternating-list[of n s t]
                alternating-list-alternates[of n st us u]
                reduced-word-for-imp-reduced-word[OF 1(2)]
                s-S t-S genset-order2-add
                contains-order2-nreduced[of u St#us]
            by (force simp add: length-alternating-list)
        qed
    qed
    thus ?thesis by fastforce
qed
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 <br> 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. 【
$S$-length $y<S$-length $w ; y \neq 0 ; S$-reduced-for $y$ ss; $S$-reduced-for $y$ ts
$\rrbracket \Longrightarrow \exists$ xss. flip-altsublist-chain (ss \# xss @ [ts])
and set-up: $S$-reduced-for $w(a \# s s) S$-reduced-for $w(a \# t s)$
shows $\exists$ xss. flip-altsublist-chain ((a\#ss) \# xss @ [a\#ts])
proof (cases ss=ts)
case True
with set-up(1) have flip-altsublist-chain ((a\#ss) \# [] @ [a\#ts])
using reduced-word-for-lists fip-altsublist-adjacent-refl by fastforce
thus ?thesis by fast
next
case False
define $y$ where $y=$ sum-list ss
with set-up(1) have ss: $S$-reduced-for y ss
using reduced-word-for-imp-reduced-word reduced-word-Cons-reduce by fast

```
    moreover from y-def ss have ts: S-reduced-for y ts
    using reduced-word-for-sum-list[OF set-up(1)]
            reduced-word-for-sum-list[OF set-up(2)]
            reduced-word-for-eq-length[OF set-up(1) set-up(2)]
            reduced-word-for-lists[OF set-up(2)]
    by (auto intro: reduced-word-forI-compare)
moreover from ss set-up(1) have S-length y<S-length w
    using reduced-word-for-length reduced-word-for-length by fastforce
    moreover from False have }y\not=
        using ss ts reduced-word-for-0-imp-nil reduced-word-for-0-imp-nil by fastforce
    ultimately show ?thesis
    using step flip-altsublist-chain-map-Cons-grow by fastforce
qed
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 $\mathcal{H} \equiv\left(\bigcup w \in W\right.$. lconjby $\left.w^{\prime} S\right)$ - the set of reflections
abbreviation lift-signed-lconjperm $\equiv$ freeword-funlift signed-lconjpermutation
lemma lconjseq-reflections: ss $\in$ lists $S \Longrightarrow$ set (lconjseq ss) $\subseteq \mathcal{H}$ using special-subgroup-eq-sum-list[of S]
by (induct ss rule: rev-induct) (auto simp add: lconjseq-snoc)
lemma deletion':

$$
\text { ss } \in \text { lists } S \Longrightarrow \neg \text { distinct (lconjseq ss) } \Longrightarrow
$$

$\exists a b$ as bs cs.ss=as@[a]@bs@[b]@cs^
sum-list ss =sum-list $(a s @ b s @ c s)$
proof (induct ss)
case (Cons s ss)
show ?case
proof (cases distinct (lconjseq ss))
case True with Cons(2,3) show ?thesis
using subset-inj-on[OF lconjby-inj, of set (lconjseq ss) s] distinct-map[of lconjby s]
genset-order2-add order2-hd-in-lconjseq-deletion[of s ss]
by (force simp add: algebra-simps)

```
    next
        case False
    with Cons(1,2) obtain a b as bs cs where
        s#ss=(s#as)@ [a]@ bs @ [b]@ cs
        sum-list (s#ss)=sum-list ((s#as)@ bs@ cs)
        by auto
    thus ?thesis by fast
    qed
qed simp
lemma S-reduced-imp-distinct-lconjseq':
    assumes ss }\in\mathrm{ lists S ᄀdistinct (lconjseq ss)
    shows \negS-reduced ss
proof
    assume ss:S-reduced ss
    from assms obtain as a bs b cs
        where decomp:ss=as @ [a]@bs@ @b]@cs
                sum-list ss = sum-list (as@bs@cs)
    using deletion'[of ss]
    by fast
    from ss decomp assms(1) show False
    using reduced-word-for-minimal[of S - ss as@bs@cs] by auto
qed
lemma S-reduced-imp-distinct-lconjseq: S-reduced ss \Longrightarrow distinct (lconjseq ss)
    using reduced-word-for-lists S-reduced-imp-distinct-lconjseq' by fast
lemma permutation-lift-signed-lconjperm-eq-signed-list-lconjaction':
    proper-signed-list xs \Longrightarrowfst'set xs}\subseteqS
    permutation (lift-signed-lconjperm (Abs-freeword xs)) =
        signed-list-lconjaction (map fst xs)
proof (induct xs)
    case Nil
    have Abs-freeword ([]::'w signed list) = (0::'w freeword)
    using zero-freeword.abs-eq by simp
    thus ?case by (simp add: zero-permutation.rep-eq freeword-funlift-0)
next
    case (Cons x xs)
    obtain s b where x: x=( s,b) by fastforce
    with Cons show ?case
    using Abs-freeword-Cons[of x xs]
        binrelchain-Cons-reduce[of nflipped-signed x xs]
        bij-signed-lconjaction[of s] genset-order2-add[of s]
    by (cases b)
        (auto simp add:
                        plus-permutation.rep-eq freeword-funlift-add
                freeword-funlift-Abs-freeletter
                Abs-permutation-inverse uminus-permutation.rep-eq
                the-inv-signed-lconjaction-by-order2
```


## freeword-funlift-uminus-Abs-freeletter

)
qed
lemma permutation-lift-signed-lconjperm-eq-signed-list-lconjaction:
$x \in$ FreeGroup $S \Longrightarrow$
permutation (lift-signed-lconjperm $x$ ) $=$ signed-list-lconjaction (map fst (freeword $x$ ))
using freeword FreeGroup-def[of S] freeword-inverse[of $x$ ] permutation-lift-signed-lconjperm-eq-signed-list-lconjaction'
by force
lemma even-count-lconjseq-rev-relator:
$s \in S \Longrightarrow t \in S \Longrightarrow$ even (count-list (lconjseq (rev (pair-relator-list st))) x)
using even-count-lconjseq-alternating-order2[of $t$ ]
by (simp add: genset-order2-add add-order rev-pair-relator-list)
lemma GroupByPresentationInducedFun-S-R-signed-lconjaction: GroupByPresentationInducedFun S P signed-lconjpermutation
proof (intro-locales, rule GroupByPresentation-S-P, unfold-locales)
fix $p s$ assume $p s: p s \in P$
define $r$ where $r=A b s$-freeword $p s$
with $p s$ have $r: r \in P^{\prime}$ by fast
then obtain $s t$ where st: $s \in S t \in S r=$ pair-relator-freeword $s t$
using relator-freewords by fast
from $r$ st(3)
have 1: permutation (lift-signed-lconjperm r) =
signed-list-lconjaction (pair-relator-list $s t$ )
using $P^{\prime}$-FreeS
permutation-lift-signed-lconjperm-eq-signed-list-lconjaction
Abs-freelist-inverse[of pair-relator-list st]
map-fst-map-const-snd[of True pair-relator-list st]
by force
have permutation (lift-signed-lconjperm $r$ ) $=$ id
proof
fix $x$
show lift-signed-lconjperm $r \rightarrow x=i d x$
proof
show snd (freeword-funlift signed-lconjpermutation $r \rightarrow x$ ) $=$ snd (id $x$ )
using 1 st ( 1,2 ) even-count-lconjseq-rev-relator genset-order2-add set-alternating-list $[$ of $2 *$ relfun stst]
signed-list-lconjaction-snd[of pair-relator-list st $x$ ]
by fastforce
qed (simp add: 1 signed-list-lconjaction-fst sum-list-pair-relator-list)
qed
moreover
have permutation $\left(0:::^{\prime} w\right.$ signed permutation $)=\left(i d::^{\prime} w\right.$ signed $\Rightarrow{ }^{\prime} w$ signed $)$
using zero-permutation.rep-eq
by fast
ultimately show lift-signed-lconjperm $r=0$
using permutation-inject by fastforce
qed
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::group-add set
+ assumes deletion:
    ss \(\in\) lists \(S \Longrightarrow \neg\) reduced-word \(S\) ss \(\Longrightarrow\)
    \(\exists a b\) as bs cs. ss =as@ [a]@bs@ [b]@cs \(\wedge\)
        sum-list ss =sum-list (as@bs@cs)
```


### 5.2.2 Consequences of the deletion condition

```
context PreCoxeterSystem WithDeletion
begin
lemma deletion-reduce:
    ss \(\in\) lists \(S \Longrightarrow \exists\) ts. ts \(\in\) ssubseqs ss \(\cap\) reduced-words-for \(S\) (sum-list ss)
proof (cases \(S\)-reduced ss)
    case True
    thus ss \(\in\) lists \(S \Longrightarrow\)
        \(\exists\) ts. \(t s \in\) ssubseqs \(s s \cap\) reduced-words-for \(S\) (sum-list ss)
    by (force simp add: ssubseqs-refl)
next
    case False
    have ss \(\in\) lists \(S \Longrightarrow \neg S\)-reduced ss \(\Longrightarrow\)
        \(\exists\) ts. ts \(\in\) ssubseqs ss \(\cap\) reduced-words-for \(S\) (sum-list ss)
    proof (induct ss rule: length-induct)
    fix \(x s::^{\prime} w\) list
    assume \(x s\) :
            \(\forall y s\). length \(y s<l e n g t h ~ x s \longrightarrow y s \in\) lists \(S \longrightarrow \neg S\)-reduced ys
                \(\longrightarrow(\exists\) ts. ts \(\in\) ssubseqs ys \(\cap\) reduced-words-for \(S\) (sum-list ys))
                \(x s \in\) lists \(S \neg S\)-reduced \(x s\)
    from \(x s(2,3)\) obtain \(a s a b s b c s\)
                where asbscs: xs \(=a s @[a] @ b s @[b] @ c s\) sum-list xs \(=\) sum-list \((a s @ b s @ c s)\)
                using deletion[of \(x s\) ]
                by fast
    show \(\exists\) ts. ts \(\in\) ssubseqs \(x s \cap\) reduced-words-for \(S\) (sum-list xs)
    proof (cases S-reduced (as@bs@cs))
            case True with asbscs xs(2) show ?thesis
```

```
            using delete2-ssubseqs by fastforce
        next
            case False
            moreover from asbscs(1) xs(2)
                have length (as@bs@cs)< length xs as@bs@cs \in lists S
            by auto
            ultimately obtain ts
            where ts:ts \in ssubseqs(as@bs@cs)\cap
                    reduced-words-for S (sum-list (as@bs@cs))
            using xs(1,2) asbscs(1)
            by fast
            with asbscs show ?thesis
            using delete2-ssubseqs[of as bs cs a b] ssubseqs-subset by auto
        qed
qed
with False
    show ss }\in\mathrm{ lists S }
                \existsts.ts\in ssubseqs ss \cap reduced-words-for S (sum-list ss)
    by fast
qed
lemma deletion-reduce':
ss \(\in\) lists \(S \Longrightarrow \exists\) ts \(\in\) reduced-words-for \(S\) (sum-list ss). set ts \(\subseteq\) set ss using deletion-reduce[of ss] subseqs-powset[of ss] by auto
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 elemented represented.

```
context PreCoxeterSystemWithDeletion
begin
lemma exchange:
    assumes }s\inSS\mathrm{ S-reduced-for w ss }\negS\mathrm{ -reduced (s#ss)
    shows \existst as bs.ss=as@t#bs ^ reduced-word-for S w (s#as@bs)
proof-
    from assms(2) have ss-lists: ss \in lists S using reduced-word-for-lists by fast
    with assms(1) have s#ss\in lists S by simp
    with assms(3) obtain a b as bs cs
        where del: s#ss=as @ [a] @ bs @ [b] @ cs
            sum-list (s#ss) = sum-list (as@bs@cs)
        using deletion[of s#ss]
```

```
    by fastforce
    show ?thesis
    proof (cases as)
    case Nil with assms(1,2) del show ?thesis
        using reduced-word-for-sum-list add.assoc[of s s w] genset-order2-add ss-lists
        by (fastforce intro: reduced-word-forI-compare)
    next
    case (Cons d ds) with del assms(2) show ?thesis
        using ss-lists reduced-word-for-imp-reduced-word
                reduced-word-for-minimal[of S sum-list ss ss ds@bs@cs]
        by fastforce
    qed
qed
lemma reduced-head-imp-exchange:
    assumes reduced-word-for S w (s#as) reduced-word-for S w cs
    shows \existsa ds es.cs=ds@[a]@es ^ reduced-word-for Sw(s#ds@es)
proof-
    from assms(1) have s-S: s\inS using reduced-word-for-lists by fastforce
    moreover have }\negS\mathrm{ -reduced (s#cs)
    proof (rule not-reduced-word-for)
        show as \in lists S using reduced-word-for-lists[OF assms(1)] by simp
        from assms(1,2) show sum-list as = sum-list (s#cs)
            using s-S reduced-word-for-sum-list[of S w] add.assoc[of s s] genset-order2-add
            by fastforce
        from assms(1,2) show length as < length (s#cs)
            using reduced-word-for-length[of S w] by fastforce
    qed
    ultimately obtain a ds es
        where cs=ds@[a]@es reduced-word-for S w(s#ds@es)
        using assms(2) exchange[of s w cs]
    by auto
    thus ?thesis by fast
qed
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\not=t reduced-word-for Sw(t#bs)n< relfun s t V relfun st=0
        reduced-word-for Sw(alternating-list n st@ @s)
    shows \existsds.reduced-word-for Sw(alternating-list (Suc n) t s @ ds)
```

```
proof-
    define altnst where altnst = alternating-list n st
    with assms(2,4) obtain x xs ys
        where xxsys: altnst @ cs = xs@[x]@ys reduced-word-for S w (t#xs@ys)
        using reduced-head-imp-exchange
        by fast
    show ?thesis
    proof (cases n rule: nat-cases-2Suc)
        case 0 with xxsys(2) show ?thesis by auto
    next
        case 1 with assms(1,4) xxsys altnst-def show ?thesis
            using reduced-word-for-sum-list[of S w s#cs]
                reduced-word-for-sum-list[of S w t#cs]
            by (cases xs) auto
    next
        case (SucSuc k)
        with assms(3,4) xxsys altnst-def have length xs \geqn
            using exchange-alternating-not-in-alternating by simp
        moreover define ds where ds= take (length xs - n) cs
        ultimately have t#xs@ys=alternating-list (Suc n) ts @ ds @ ys
            using xxsys(1) altnst-def take-append[of length xs altnst cs]
                alternating-list-Suc-Cons[of n t]
            by (fastforce simp add:length-alternating-list)
        with xxsys(2) show ?thesis by auto
    qed
qed
lemma two-reduced-heads-imp-reduced-alt':
    assumes }s\not=t\mathrm{ reduced-word-for S w (s#as) reduced-word-for S w (t#bs)
    shows n\leq relfun s t\vee relfun s t=0\Longrightarrow(\exists cs.
        reduced-word-for Sw(alternating-list n s t @ cs) V
        reduced-word-for S w (alternating-list n t s @ cs)
        )
proof (induct n)
    case 0 from assms(2) show ?case by auto
next
    case (Suc m) thus ?case
        using add-order-add-sym[of s t]
                two-reduced-heads-imp-reduced-alt-step[
                    OF assms(1)[THEN not-sym] assms(2), of m
                ]
                two-reduced-heads-imp-reduced-alt-step[OF assms(1,3), of m]
        by fastforce
qed
lemma two-reduced-heads-imp-reduced-alt:
    assumes }s\not=t\mathrm{ reduced-word-for S w (s#as) reduced-word-for S w (t#bs)
    shows \existscs. reduced-word-for S w (pair-relator-halflist s t @ cs)
proof -
```

```
    define altst altts
    where altst = pair-relator-halflist s t
        and altts = pair-relator-halflist t s
    then obtain cs
    where cs: reduced-word-for Sw(altst @ cs) \vee
                reduced-word-for S w (altts @ cs)
    using add-order-add-sym[of t] two-reduced-heads-imp-reduced-alt'[OF assms]
    by auto
    moreover from altst-def altts-def
    have reduced-word-for Sw(altts @ cs) \Longrightarrow reduced-word-for Sw (altst @ cs)
    using reduced-word-for-lists[OF assms(2)] reduced-word-for-lists[OF assms(3)]
        flip-altsublist-adjacent-def
    by (force intro: S-reduced-forI-flip-altsublist-adjacent
        simp add: add-order-add-sym)
    ultimately show \exists cs. reduced-word-for S w (altst @ cs) by fast
qed
lemma two-reduced-heads-imp-nzero-relfun:
    assumes }s\not=t\mathrm{ reduced-word-for S w(s#as) reduced-word-for S w(t#bs)
    shows relfun s t}\not=
proof
    assume 1: relfun st=0
    define altst altts
        where altst = alternating-list (Suc (S-length w)) st
        and altts = alternating-list (Suc (S-length w)) ts
    with 1 obtain cs
        where reduced-word-for S w(altst @ cs)\vee
                reduced-word-for S w (altts @ cs)
    using two-reduced-heads-imp-reduced-alt'[OF assms]
    by fast
    moreover from altst-def altts-def
    have length (altst @ cs)>S-length w
        length (altts @ cs) > S-length w
    using length-alternating-list[of-s] length-alternating-list[of - t]
    by auto
    ultimately show False using reduced-word-for-length by fastforce
qed
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. $\llbracket S$-length $y<S$-length $w ; y \neq 0$; reduced-word-for $S$ y ss; reduced-word-for $S$ y ts $\rrbracket \Longrightarrow \exists$ xss. flip-altsublist-chain (ss \# xss @ $[t s]$ )

```
            reduced-word-for S w (a#as) reduced-word-for S w (b#bs) a\not=b
    shows \existsxss. flip-altsublist-chain ((a#as)#xss@[b#bs])
proof-
    from assms(2-4) obtain cs
        where cs: reduced-word-for S w (pair-relator-halfist a b @ cs)
        using two-reduced-heads-imp-reduced-alt
        by fast
    define rs us where rs= pair-relator-halflist a b@cs
    and us=pair-relator-halfist b a @ cs
    from assms(2,3) have a-S: a\inS and b-S: b\inS
        using reduced-word-for-lists[of S-a#as] reduced-word-for-lists[of S - b#bs]
        by auto
    with rs-def us-def have midlink: flip-altsublist-adjacent rs us
    using add-order-add-sym[of b a] flip-altsublist-adjacent-def by fastforce
    from assms(2-4) have relfun a b}=
    using two-reduced-heads-imp-nzero-relfun by fast
    from this obtain k where k: relfun a b=Suc k
    using not0-implies-Suc by auto
define qs vs
    where qs = alternating-list k b a @ cs
        and vs = alternating-list kab @ cs
    with k rs-def us-def have rs': rs =a # qs and us':us = b # vs
    using add-order-add-sym[of b a] alternating-list-Suc-Cons[of k] by auto
from assms(1,2) cs rs-def rs'
    have startlink: as \not=qs\Longrightarrow\existsxss. flip-altsublist-chain ((a#as) # xss @ [rs])
    using reduced-word-problem-eq-hd-step
    by fastforce
from assms(1,3) rs-def cs us'
    have endlink: bs }\not=vs\Longrightarrow\existsxss. flip-altsublist-chain (us # xss @ [b#bs]
    using midlink flip-altsublist-adjacent-sym
                S-reduced-forI-flip-altsublist-adjacent[of w rs]
                reduced-word-problem-eq-hd-step[of w]
    by auto
show ?thesis
proof (cases as=qs bs = vs rule: two-cases)
    case both
    with rs' us' have flip-altsublist-chain ((a#as) # [] @ [b#bs])
        using midlink by simp
    thus ?thesis by fast
next
    case one
    with rs' obtain xss
        where flip-altsublist-chain ((a#as) # (us # xss) @ [b#bs])
        using endlink midlink
        by auto
    thus ?thesis by fast
next
    case other
    from other(1) obtain xss where flip-altsublist-chain ((a#as) # xss @ [rs])
```

```
        using startlink by fast
    with other(2) us' startlink
        have flip-altsublist-chain ((a#as) # (xss@[rs]) @ [b#bs])
        using midlink binrelchain-snoc[of flip-altsublist-adjacent (a#as)#xss]
        by simp
    thus ?thesis by fast
    next
    case neither
    from neither(1) obtain xss
        where flip-altsublist-chain ((a#as) # xss @ [rs])
        using startlink
        by fast
    with neither(2) obtain yss
        where flip-altsublist-chain ((a#as) # (xss @ [rs,us] @ yss) @ [b#bs])
        using startlink midlink endlink
                binrelchain-join[of flip-altsublist-adjacent (a#as)#xss]
    by auto
    thus ?thesis by fast
    qed
qed
lemma reduced-word-problem:
    \llbracketw\not=0; reduced-word-for S w ss; reduced-word-for S wts\rrbracket\Longrightarrow
    \existsxss. flip-altsublist-chain (ss#xss@[ts])
proof (induct w arbitrary: ss ts rule: measure-induct-rule[of S-length])
    case (less w)
    show ?case
    proof (cases ss ts rule: two-lists-cases-Cons-Cons)
        case Nil1 from Nil1 (1) less(2,3) show ?thesis
        using reduced-word-for-sum-list by fastforce
    next
        case Nil2 from Nil2(2) less(2,4) show ?thesis
            using reduced-word-for-sum-list by fastforce
    next
        case (ConsCons a as b bs)
        show ?thesis
        proof (cases a=b)
            case True with less ConsCons show ?thesis
                using reduced-word-problem-eq-hd-step[of w] by auto
    next
            case False with less ConsCons show ?thesis
                using reduced-word-problem-ConsCons-step[of w] by simp
    qed
    qed
qed
lemma reduced-word-letter-set:
    assumes S-reduced-for w ss
    shows reduced-letter-set S w = set ss
```

```
proof (cases w=0)
    case True with assms show ?thesis
        using reduced-word-for-0-imp-nil[of S ss] reduced-letter-set-0 by simp
next
    case False
    show ?thesis
    proof
        from assms show set ss \subseteqreduced-letter-set S w by fast
        show reduced-letter-set S w\subseteq set ss
        proof
            fix x assume x reduced-letter-set S w
            from this obtain ts where reduced-word-for S w ts x set ts by fast
            with False assms show }x\in\mathrm{ set ss
            using reduced-word-for-lists[of S - ss] reduced-word-problem[of w ss]
                    flip-altsublist-chain-set
            by force
        qed
    qed
qed
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 }\langleT\rangle\capS=
proof
    show }\langleT\rangle\capS\subseteq
    proof
        fix t assume t: t\in\langleT\rangle\capS
        with assms obtain ts where ts: ts \in lists T t = sum-list ts
            using special-subgroup-eq-sum-list[of T] by fast
        with assms obtain us
            where us: reduced-word-for S (sum-list ts) us set us \subseteq set ts
            using deletion-reduce'[of ts]
            by auto
        with no-zero-genset ts(2) t have length us = 1
            using reduced-word-for-lists[of S - us] reduced-word-for-sum-list[of S-us]
```

```
                reduced-word-for-imp-reduced-word[of S-us] el-reduced[of S]
        by auto
    with us ts show t\inT
        using reduced-word-for-sum-list[of S - us] by (cases us) auto
    qed
    from assms show T\subseteq\langleT\rangle\capS using genby-genset-subset by fast
qed
lemma special-subgroup-inj: inj-on genby (Pow S)
    using special-subgroup-int-S inj-on-inverseI[of - \lambdaW.W\capS] by fastforce
lemma special-subgroup-genby-subset-ordering-iso:
    subset-ordering-iso (Pow S) genby
proof (unfold-locales, rule genby-mono, simp, rule special-subgroup-inj)
    fix X Y assume XY: X Genby 'Pow S Y G genby'Pow S X\subseteqY
    from XY(1,2) obtain TX TY
        where TX\inPow S X = \langleTX\rangle TY\inPow S Y = <TY\rangle
        by auto
    hence the-inv-into (Pow S) genby X = X\capS
                the-inv-into (Pow S) genby Y = Y\capS
        using the-inv-into-f-f[OF special-subgroup-inj] special-subgroup-int-S
        by auto
    with XY(3)
        show the-inv-into (Pow S) genby X \subseteq the-inv-into (Pow S) genby Y
        by auto
qed
lemmas special-subgroup-genby-rev-mono
    = OrderingSetIso.rev-ordsetmap[OF special-subgroup-genby-subset-ordering-iso]
lemma special-subgroup-word-length:
    assumes T\inPow S w\in\langleT\rangle
    shows word-length T w = S-length w
proof-
    from assms obtain ts where ts: ts lists T w= sum-list ts
        using special-subgroup-eq-sum-list by auto
    with assms(1) obtain us where us \in ssubseqs ts S-reduced-for w us
        using deletion-reduce[of ts] by fast
    with assms(1) ts(1) show ?thesis
        using ssubseqs-lists[of ts] reduced-word-for-sum-list
                        is-arg-min-size-subprop[of length word-for S w us word-for T w]
    unfolding reduced-word-for-def word-length-def
    by fast
qed
lemma S-subset-reduced-imp-S-reduced:
    T\inPow S \Longrightarrow reduced-word T ts \LongrightarrowS-reduced ts
    using reduced-word-for-lists reduced-word-for-lists[of T-ts]
        reduced-word-for-length[of T sum-list ts ts] special-subgroup-eq-sum-list[of T]
```

```
        special-subgroup-word-length[of T sum-list ts]
    by
        (fastforce intro: reduced-word-forI-length)
        lemma smallest-genby:T\inPow S\Longrightarroww\in\langleT\rangle\Longrightarrow reduced-letter-set S w\subseteqT
    using genby-S-reduced-word-for-arg-min[of T]
    reduced-word-for-imp-reduced-word[of T w]
    S-subset-reduced-imp-S-reduced[of T arg-min length (word-for T w)]
    reduced-word-for-sum-list[of T] reduced-word-for-lists reduced-word-letter-set
    by fastforce
lemma special-cosets-below-in:
    assumes w\inWT\in Pow S
    shows }\quad\mathcal{P}.\supseteq(w+o\langleT\rangle)=(\bigcupR\in(\mathrm{ Pow S ).`T. {w+o <R>})
proof (rule seteqI)
    fix }A\mathrm{ assume }A\in\mathcal{P}.\supseteq(w+o\langleT\rangle
    hence A: A\in\mathcal{P}A\supseteq(w+o\langleT\rangle) by auto
    from A(1) obtain R w' where R\inPow SA= w' +o \langleR\rangle
        using special-cosets-def by auto
    with A(2) assms(2) show }A\in(\bigcupR\in(Pow S).\supseteqT. {w+o\langleR\rangle}
        using genby-lcoset-subgroup-imp-eq-reps[of w T w' R]
                lcoset-eq-reps-subset[of w \langleT\rangle]
                special-subgroup-genby-rev-mono[of T R]
    by auto
next
    fix B assume B\in(\bigcupR\in(Pow S).\supseteqT.{w+o\langleR\rangle})
    from this obtain R where R: R (Pow S).\supseteqT B=w+o }\langleR\rangle\mathrm{ by auto
    moreover hence B\supseteqw+o \langleT\rangle
        using genby-mono elt-set-plus-def[of w] by auto
    ultimately show }B\in\mathrm{ special-cosets .ొ (w+o 
        using assms(1) special-cosetsI by auto
qed
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:
    \llbracketT1\inPow S;T2\inPow S;w1 +o \langleT1\rangle=w2 +o \langleT2\rangle\rrbracket\LongrightarrowT1=T2
    using set-plus-rearrange2[of -w1 w1 \langleT1\rangle]
        set-plus-rearrange2[of -w1 w2 <T2\rangle]
        genby-lcoset-subgroup-imp-eq-reps[of 0 T1 -w1+w2 T2]
        inj-onD[OF special-subgroup-inj]
    by force
lemma special-subgroup-special-coset-subset-ordering-iso:
subset-ordering-iso (genby'Pow S) ((+o) w)
proof
show \(\bigwedge a b . a \subseteq b \Longrightarrow w+o a \subseteq w+o b\) using elt-set-plus-def by auto
show 2: inj-on \(((+o) w)\) (genby ' Pow \(S\) )
using lcoset-inj-on inj-inj-on by fast
```

```
    show \ab.a\in(+o) w'genby 'Pow S\Longrightarrow
            b\in(+o)w'genby'Pow S\Longrightarrow
            a\subseteqb\Longrightarrow
            the-inv-into (genby 'Pow S) ((+o)w) a\subseteq
                the-inv-into (genby`Pow S) ((+o)w)b
    proof-
    fix ab
    assume ab:a\in(+o) w'genby 'Pow S b\in(+o)w'genby'Pow S
        and a-b: a\subseteqb
    from ab obtain Ta Tb
        where Ta\inPow S a =w +o \langleTa\rangle Tb\inPow S b =w +o \langleTb\rangle
        by auto
    with a-b
        show the-inv-into (genby'Pow S) ((+o)w)a\subseteq
                the-inv-into (genby 'Pow S) ((+o) w) b
        using the-inv-into-f-eq[OF 2] lcoset-eq-reps-subset[of w \langleTa\rangle\langleTb\rangle]
        by simp
    qed
qed
lemma special-coset-subset-ordering-iso:
    subset-ordering-iso (Pow S) ((+o) w ○ genby)
    using special-subgroup-genby-subset-ordering-iso
        special-subgroup-special-coset-subset-ordering-iso
    by (fast intro: OrderingSetIso.iso-comp)
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).\supseteqT) ((+o) w ○ genby)
    using special-coset-subset-ordering-iso by (auto intro: OrderingSetIso.iso-subset)
lemma special-coset-below-in-supset-ordering-iso:
    OrderingSetIso (\supseteq) (\supset) (\supseteq) (\supset) ((Pow S).\supseteqT) ((+o)w\circ genby)
    using special-coset-below-in-subset-ordering-iso OrderingSetIso.iso-dual by fast
lemma special-coset-pseudominimals:
    assumes supset-pseudominimal-in }\mathcal{P}
    shows }\exists\textrm{w}\mathrm{ s. wGW^s位^X=w+o \S-{s}>
proof-
    from assms have }X\in\mathcal{P}\mathrm{ using supset-pseudominimal-inD1 by fast
    from this obtain w T where wT: w\inW T\inPow S X =w +o \langleT\rangle
        using special-cosets-def by auto
    show ?thesis
    proof (cases T=S)
    case True with wT(1,3) assms show ?thesis
        using genby-lcoset-el-reduce supset-pseudominimal-ne-bottom
            special-cosets-bottom
```

```
    by fast
    next
    case False
    with wT(2) obtain s where s: s\inS T\subseteqS-{s} by fast
    from s(2) wT(1,3) assms have X\subseteqw+o \S-{s}\rangle
        using genby-mono by auto
    moreover from assms wT(1)s(1) have }\negX\subsetw+o\langleS-{s}
        using special-cosetsI[of-w]
                supset-pseudominimal-inD2[of \mathcal{P X w +o \langleS-{s}\rangle]}]
                lcoset-eq-reps[of w- <S\rangle]
                inj-onD[OF special-subgroup-inj, of S-{s} S]
            by (auto simp add: special-cosets-bottom genby-lcoset-el-reduce)
    ultimately show ?thesis using wT(1) s(1) by fast
    qed
qed
lemma special-coset-pseudominimal-in-below-in:
    assumes w\inW T\inPow S supset-pseudominimal-in (\mathcal{P}.\supseteq(w+o\langleT\rangle)) X
    shows }\existss\inS-T.X=w+o\langleS-{s}
proof-
    from assms obtain vs where vs: v\inW s\inS X =v+o \langleS-{s}\rangle
    using special-cosets-has-bottom special-cosetsI[of T w]
                supset-has-bottom-pseudominimal-in-below-in
                special-coset-pseudominimals
    by force
    from assms(3) have X:X\supseteqw+o \T\rangle
        using supset-pseudominimal-inD1 by fast
    with vs(3) have 1: X=w+o \langleS-{s}\rangle
    using genby-lcoset-subgroup-imp-eq-reps[of w TvS-{s}] by fast
    with X assms have T\subseteqS-{s}
    using special-cosetsI special-coset-subset-rev-mono[of T S-{s}]
    by fastforce
    with vs(2) show ?thesis using 1 by fast
qed
lemma exclude-one-is-pseudominimal:
    assumes w\inW t\inS
    shows supset-pseudominimal-in \mathcal{P}(w+o\langleS-{t}\rangle)
proof (rule supset-pseudominimal-inI, rule special-cosetsI)
    show w\inW by fact
    from assms have w+o }\langleS-{t}\rangle\not=
    using genby-lcoset-el-reduce[of w] lcoset-eq-reps[of w - W]
                inj-onD[OF special-subgroup-inj, of S-{t} S]
    by auto
    thus w+o }\langleS-{t}\rangle\not= supset-bottom \mathcal{P
    using special-cosets-bottom by fast
next
    fix X assume X:X\in\mathcal{P}w+o \langleS-{t}\rangle\subsetX
    with assms(1) have X \in(\bigcup R\in(Pow S).\supseteq(S-{t}). {w+o \langleR\rangle})
```

using subst $[$ OF special-cosets-below-in, of $w S-\{t\} \lambda A . X \in A]$ by fast
from this obtain $R$ where $R: R \in($ Pow $S) . \supseteq(S-\{t\}) X=w+o\langle R\rangle$ by auto from $R(2) X(2)$ have $R \neq S-\{t\}$ by fast
with $R(1)$ have $R=S$ by auto
with $\operatorname{assms}(1) R(2)$ show $X=$ supset-bottom $\mathcal{P}$
using genby-lcoset-el-reduce special-cosets-bottom by fast
qed fast
lemma exclude-one-is-pseudominimal-in-below-in:
$\llbracket w \in W ; T \in$ Pow $S ; s \in S-T \rrbracket \Longrightarrow$
supset-pseudominimal-in $(\mathcal{P} . \supseteq(w+o\langle T\rangle))(w+o\langle S-\{s\}\rangle)$
using special-cosets-has-bottom special-cosetsI
exclude-one-is-pseudominimal $[$ of $w s]$
genby-mono[of TS-\{s\}]
supset-has-bottom-pseudominimal-in-below-inI [
of $\mathcal{P} w+o\langle T\rangle w+o\langle S-\{s\}\rangle$
]
by auto
lemma glb-special-subset-coset:
assumes $\quad w T T^{\prime}: w \in W T \in$ Pow $S T^{\prime} \in$ Pow $S$
defines $\quad U: U \equiv T \cup T^{\prime} \cup$ reduced-letter-set $S w$
shows supset-glbound-in-of $\mathcal{P}\langle T\rangle(w+o\langle T\rangle)\langle U\rangle$
proof (rule supset-glbound-in-ofI)
from $w T T^{\prime}(2,3) U$ show $\langle U\rangle \in \mathcal{P}$
using reduced-letter-set-subset[of S] special-subgroup-special-coset by simp

```
show supset-lbound-of \(\langle T\rangle\left(w+o\left\langle T^{\prime}\right\rangle\right)\langle U\rangle\)
proof (rule supset-lbound-ofI)
    from \(U\) show \(\langle T\rangle \subseteq\langle U\rangle\) using genby-mono[of \(T U]\) by fast
    show \(w+o\langle T\rangle \subseteq\langle U\rangle\)
    proof
        fix \(x\) assume \(x \in w+o\langle T\rangle\)
        with \(w T T^{\prime}(3)\) obtain \(y\) where \(y: y \in\langle T\rangle x=w+y\)
            using elt-set-plus-def \([o f w]\) by auto
            with \(w T T^{\prime}(1) U\) show \(x \in\langle U\rangle\)
                using in-genby-S-reduced-letter-set genby-mono[of - U]
                    genby-mono[of \(\left.T^{\prime} U\right]\) genby-add-closed[of \(\left.w U y\right]\)
            by auto
        qed
    qed
```

next
fix $X$ assume $X: X \in \mathcal{P}$ supset-lbound-of $\langle T\rangle(w+o\langle T\rangle) X$
from $X(1)$ obtain $v R$ where $v R: R \in \operatorname{Pow} S X=v+o\langle R\rangle$
using special-cosets-def by auto
from $X(2)$ have $X^{\prime}: X \supseteq\langle T\rangle X \supseteq w+o\langle T\rangle$
using supset-lbound-of-def[of - X] by auto from $X^{\prime}(1) v R(2)$ have $R: X=\langle R\rangle$
using genby-0-closed genby-lcoset-el-reduce0 by fast
with $X^{\prime}(2)$ have $w: w \in\langle R\rangle$ using genby-0-closed lcoset-refl by fast
have $T^{\prime} \subseteq R$
proof (
rule special-subgroup-genby-rev-mono, rule $w T T^{\prime}(3)$, rule $v R(1)$, rule subsetI
)
fix $x$ assume $x \in\langle T\rangle$
with $X^{\prime}(2) R$ show $x \in\langle R\rangle$
using elt-set-plus-def[of $w\langle T\rangle$ ] w genby-uminus-add-closed[of $w R w+x]$
by auto
qed
with $X^{\prime}(1) w T T^{\prime}(2) v R(1)$ show $\langle U\rangle \subseteq X$
using special-subgroup-genby-rev-mono[of $T R]$ smallest-genby $U R$ genby-mono $[o f-R]$
by $\operatorname{simp}$
qed
lemma glb-special-subset-coset-ex:
assumes $\quad w \in W T \in$ Pow $S T^{\prime} \in$ Pow $S$
shows $\quad \exists$ B. supset-glbound-in-of $\mathcal{P}\langle T\rangle\left(w+o\left\langle T^{\prime}\right\rangle\right) B$
using glb-special-subset-coset[OF assms]
by fast
lemma special-cosets-have-glbs:
assumes $X \in \mathcal{P} \quad Y \in \mathcal{P}$
shows $\exists B$. supset-glbound-in-of $\mathcal{P} X Y B$
proof-
from assms obtain wx Tx wy Ty
where $X: w x \in W T x \in$ Pow $S X=w x+o\langle T x\rangle$
and $\quad Y: w y \in W T y \in P o w S Y=w y+o\langle T y\rangle$
using special-cosets-def
by auto
from $X(1,2) Y(1,2)$ obtain $A$
where $A$ : supset-glbound-in-of $\mathcal{P}\langle T x\rangle((-w x+w y)+o\langle T y\rangle) A$
using genby-uminus-add-closed[of wx] glb-special-subset-coset-ex by fastforce
from $X(1,3) Y(3)$ have supset-glbound-in-of $\mathcal{P} X Y(w x+o A)$
using supset-glbound-in-of-lcoset-shift[OF $A$, of wx]
by (auto simp add: set-plus-rearrange2 special-cosets-lcoset-shift)
thus ?thesis by fast
qed
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 \g. g\inG\Longrightarrow induced-id g=0\Longrightarrowg=0
    shows CoxeterSystem S
proof
    from assms have GroupIso G induced-id
        using Group WithGeneratorsRelators-S-R
                GroupWithGeneratorsRelators.induced-id-hom-surj(1)
        by (fast intro: GroupHom.isoI)
    thus inj-on induced-id G using GroupIso.inj-on by fast
qed
context CoxeterSystem
begin
abbreviation inv-induced-id \equivGroupPresentation.inv-induced-id S R
lemma GroupPresentation-S-R:GroupPresentation S R
    by (
        intro-locales, rule Group WithGeneratorsRelators-S-R,
        unfold-locales, rule induced-id-inj
        )
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 > 'w set
    where flipped-reflections w 三
        {t\in\mathcal{H. induced-signed-lconjperm (inv-induced-id (-w)) }->
                        (t,True ) = (rconjby wt,False)}
lemma induced-signed-lconjperm-inv-induced-id-sum-list:
    ss }\in\mathrm{ lists S C induced-signed-lconjperm (inv-induced-id (sum-list ss))}
        sum-list (map signed-lconjpermutation ss)
    by (simp add:
        inv-induced-id-sum-list Abs-freelist-in-FreeGroup
        GroupByPresentationInducedFun.induced-hom-Abs-freelist-conv-sum-list[
        OF GroupByPresentationInducedFun-S-R-signed-lconjaction
    ]
    )
lemma induced-signed-eq-lconjpermutation:
ss \(\in\) lists \(S \Longrightarrow\)
permutation \((\) induced-signed-lconjperm \((\) inv-induced-id \((\) sum-list ss \()))=\) signed-list-lconjaction ss
proof (induct ss)
case Nil
have permutation (induced-signed-lconjperm (inv-induced-id \((\) sum-list []\()))=\) id
using induced-signed-lconjperm-inv-induced-id-sum-list[of []]
zero-permutation.rep-eq
by simp
thus ?case by fastforce
next
case (Cons s ss)
from \(\operatorname{Cons}(2)\)
have induced-signed-lconjperm (inv-induced-id (sum-list (s\#ss))) = signed-lconjpermutation \(s+\) sum-list (map signed-lconjpermutation ss)
using induced-signed-lconjperm-inv-induced-id-sum-list[of \(s \#\) ss]
by \(\operatorname{simp}\)
with Cons(2) have
permutation \((\) induced-signed-lconjperm \((\) inv-induced-id \((\) sum-list \((s \# s s))))=\) permutation (signed-lconjpermutation s) \(\circ\) permutation (induced-signed-lconjperm (inv-induced-id (sum-list ss)))
using plus-permutation.rep-eq induced-signed-lconjperm-inv-induced-id-sum-list by \(\operatorname{simp}\)
with Cons show ?case
using bij-signed-lconjaction[of s] Abs-permutation-inverse by fastforce
qed
lemma flipped-reflections-odd-lconjseq:
assumes \(s s \in\) lists \(S\)
shows flipped-reflections (sum-list ss) \(=\{t \in \mathcal{H}\). odd (count-list (lconjseq ss) \(t)\}\)
```

```
proof (rule seteqI)
    fix t assume t\in flipped-reflections (sum-list ss)
    moreover with assms
        have snd (signed-list-lconjaction (rev ss) (t,True)) = False
        using flipped-reflections-def genset-order2-add uminus-sum-list-order2
                induced-signed-eq-lconjpermutation[of rev ss]
    by force
    ultimately show }t\in{t\in\mathcal{H}\mathrm{ . odd (count-list (lconjseq ss) t)}
    using assms flipped-reflections-def genset-order2-add
                signed-list-lconjaction-snd[of rev ss]
    by auto
next
    fix t assume t:t\in{t\in\mathcal{H. odd (count-list (lconjseq ss) t)}}
    with assms
        have signed-list-lconjaction (rev ss) (t,True) =
                (rconjby (sum-list ss) t, False)
        using genset-order2-add signed-list-lconjaction-snd[of rev ss]
                signed-list-lconjaction-fst[of rev ss]
                uminus-sum-list-order2[of ss, THEN sym]
    by (auto intro: prod-eqI)
    with t assms show t flipped-reflections (sum-list ss)
    using induced-signed-eq-lconjpermutation[of rev ss] genset-order2-add
                uminus-sum-list-order2 flipped-reflections-def
    by fastforce
qed
lemma flipped-reflections-in-lconjseq:
    ss\inlists S \Longrightarrow flipped-reflections (sum-list ss)\subseteq set (lconjseq ss)
    using flipped-reflections-odd-lconjseq odd-n0 count-notin[of - lconjseq ss]
    by fastforce
lemma flipped-reflections-distinct-lconjseq-eq-lconjseq:
    assumes ss\inlists S distinct (lconjseq ss)
    shows flipped-reflections (sum-list ss) = set (lconjseq ss)
proof
    from assms(1) show flipped-reflections (sum-list ss) \subseteq set (lconjseq ss)
    using flipped-reflections-in-lconjseq by fast
    show flipped-reflections (sum-list ss) \supseteq set (lconjseq ss)
    proof
    fix t assume t\in set (lconjseq ss)
    moreover with assms(2) have count-list (lconjseq ss) t=1
        by (simp add: distinct-count-list)
    ultimately show t\in flipped-reflections (sum-list ss)
            using assms(1) flipped-reflections-odd-lconjseq lconjseq-reflections
            by fastforce
    qed
qed
lemma flipped-reflections-reduced-eq-lconjseq:
```

```
    S-reduced ss \Longrightarrow flipped-reflections (sum-list ss) = set (lconjseq ss)
    using reduced-word-for-lists[of S] S-reduced-imp-distinct-lconjseq
    flipped-reflections-distinct-lconjseq-eq-lconjseq
    by fast
lemma card-flipped-reflections:
    assumes w\inW
    shows card (flipped-reflections w) =S-length w
proof-
    define ss where ss = arg-min length (word-for S w)
    with assms have S-reduced-for w ss
    using genby-S-reduced-word-for-arg-min by simp
    thus ?thesis
    using reduced-word-for-sum-list flipped-reflections-reduced-eq-lconjseq
                S-reduced-imp-distinct-lconjseq distinct-card length-lconjseq[of ss]
                reduced-word-for-length
    by fastforce
qed
end
sublocale CoxeterSystem < PreCoxeterSystemWithDeletion
proof
    fix ss assume ss: ss }\in\mathrm{ lists }S\negS\mathrm{ -reduced ss
    define w}\mathrm{ where w= sum-list ss
    with ss(1)
        have distinct (lconjseq ss) \Longrightarrow card (flipped-reflections w) = length ss
        by (simp add:
            flipped-reflections-distinct-lconjseq-eq-lconjseq distinct-card
            length-lconjseq)
    moreover from w-def ss have length ss > S-length w using word-length-lt by
fast
    moreover from w-def ss(1) have card (flipped-reflections w) =S-length w
        using special-subgroup-eq-sum-list card-flipped-reflections by fast
    ultimately have \neg distinct (lconjseq ss) by auto
    with w-def ss
        show \existsab as bs cs.ss=as @ [a] @ bs @ [b] @ cs ^
            sum-list ss=sum-list (as @ bs @ cs)
        using deletion'
        by fast
qed
```


### 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 PreCoxeterSystemWithDeletion
begin
```

```
lemma reducible-by-flipping:
    ss }\in\mathrm{ lists S }\Longrightarrow\negS\mathrm{ -reduced ss }
        \existsxss as t bs.flip-altsublist-chain (ss # xss @ [as@[t,t]@bs])
proof (induct ss)
    case (Cons s ss)
    show ?case
    proof (cases S-reduced ss)
        case True
        define w where w=sum-list ss
        with True have ss-red-w: reduced-word-for S w ss by fast
        moreover from Cons(2) have s\inS by simp
        ultimately obtain as bs where asbs: reduced-word-for S w (s#as@bs)
            using Cons(3) exchange by fast
        show ?thesis
        proof (cases w=0)
            case True with asbs show ?thesis
            using reduced-word-for-0-imp-nil by fast
        next
            case False
            from this obtain xss where flip-altsublist-chain (ss # xss @ [s#as@bs])
            using ss-red-w asbs reduced-word-problem by fast
        hence flip-altsublist-chain (
                        (s#ss) # map ((#) s) xss @ []@[s,s]@(as@bs)]
                    )
                using flip-altsublist-chain-map-Cons-grow by fastforce
            thus ?thesis by fast
        qed
    next
        case False
        with Cons(1,2) obtain xss as t bs
            where flip-altsublist-chain (
                (s#ss) # map ((#) s)xss@ [(s#as)@[t,t]@bs]
                )
            using flip-altsublist-chain-map-Cons-grow
            by fastforce
        thus ?thesis by fast
    qed
qed (simp add: nil-reduced-word-for-0)
lemma freeliftid-kernel':
    ss}\in\mathrm{ lists S 的m-list ss = 0 Abs-freelist ss }\in
proof (induct ss rule: length-induct)
    fix ss
    assume step: }\forall\mathrm{ ts.length ts < length ss }\longrightarrow\mathrm{ ts }\in\mathrm{ lists S }
                            sum-list ts = 0\longrightarrowAbs-freelist ts }\in
    and set-up: ss }\in\mathrm{ lists S sum-list ss =0
    show Abs-freelist ss }\in
    proof (cases ss=[])
```

```
    case True thus ?thesis
        using genby-0-closed[of \ w FreeGroup S.lconjby w'P'
    by (auto simp add: zero-freeword.abs-eq)
next
    case False
    with set-up obtain xss as t bs
    where xss: flip-altsublist-chain (ss # xss@ [as@[t,t]@bs])
    using sum-list-zero-nreduced reducible-by-flipping[of ss]
    by fast
    with set-up
    have astbs: length(as@[t,t]@bs)= length ss
                    as@[t,t]@bs\in lists S
                sum-list(as@ [t,t]@bs)=0
    using flip-altsublist-chain-length[of ss xss as@[t,t]@bs]
            flip-altsublist-chain-sum-list[of ss xss as@[t,t]@bs]
            flip-altsublist-chain-lists[of ss xss as@[t,t]@bs]
    by auto
    have listsS: as lists S t\inS bs\inlists S using astbs(2) by auto
    have sum-list as + (t+t+ sum-list bs)=0
    using astbs(3) by (simp add: add.assoc)
    hence sum-list (as@bs)=0
        using listsS(2) by (simp add: genset-order2-add)
    moreover have length (as@bs) < length ss using astbs(1) by simp
    moreover have as@bs \in lists S using listsS(1,3) by simp
    ultimately have Abs-freelist (as@bs) \inQ using step by fast
    hence Abs-freelist as + pair-relator-freeword tt+
        (- Abs-freelist as + (Abs-freelist as + Abs-freelist bs)) \inQ
        using listsS(1,2) lconjby-Abs-freelist-relator-freeword[of t t as]
                genby-add-closed
            by (simp add:Abs-freelist-append[THEN sym] add.assoc[THEN sym])
    hence Abs-freelist as + Abs-freelist [t,t] + Abs-freelist bs }\in
        using listsS(2) by (simp add: S-relator-freeword Abs-freeletter-add)
    thus ?thesis
    using Abs-freelist-append-append[of as [t,t] bs]
                rev-flip-altsublist-chain[OF xss]
                flip-altsublist-chain-G-in-Q[of as@[t,t]@bs rev xss ss]
    by simp
qed
qed
lemma freeliftid-kernel:
    assumes c\in FreeGroup S freeliftid c=0
    shows c\inQ
proof-
    from assms(2) have freeliftid (Abs-freeword (freeword c)) =0
        by (simp add: freeword-inverse)
    with assms(1) have sum-list (map fst (freeword c)) = 0
    using FreeGroup-def freeword freeliftid-Abs-freeword-conv-sum-list by fastforce
    with assms(1) show ?thesis
```

```
    using FreeGroup-def freeliftid-kernel'[of map fst (freeword c)]
        Q-freelist-freeword
    by fastforce
qed
lemma induced-id-kernel:
    c}\in\mathrm{ FreeGroup S > induced-id ( }\lceil\mathrm{ FreeGroup S|c|Q\)=0 cceQ
    by (simp add:
        freeliftid-kernel
        GroupByPresentationInducedFun.induced-hom-equality[
            OF GroupByPresentationInducedFun-S-P-id
        ]
        )
theorem CoxeterSystem: CoxeterSystem S
proof (rule CoxeterSystemI)
    fix }x\mathrm{ assume }x:x\inG\mathrm{ induced-id }x=
    from x(1) obtain c where c f FreeGroup S x = (\lceilFreeGroup S|c|Q\rceil)
        using Group.quotient-group-UN FreeGroup-Group by fast
    with x(2) show }x=
        using induced-id-kernel
            Group.quotient-identity-rule[OF FreeGroup-Group]
            GroupByPresentation.Q-subgroup-FreeS[OF GroupByPresentation-S-P]
            GroupByPresentation.normal-Q[OF GroupByPresentation-S-P]
        by auto
qed
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 $s s \in$ lists $S \neg$ reduced-word $S$ ss
shows $\neg$ min-gallery $\left(\operatorname{map}\left(\lambda w . w^{6} \rightarrow C 0\right)(\right.$ sums ss $\left.)\right)$
proof (cases ss rule: list-cases-Cons-snoc)
case Nil with assms(2) show ?thesis using nil-reduced-word-for-0 by auto next
case (Single s) with assms show ?thesis
using zero-notin-S reduced-word-singleton $[$ of $s S]$ by fastforce
next
case (Cons-snoc sts t) have ss: ss $=s \# t s @[t]$ by fact

```
define \(M s\) where \(M s=\operatorname{map}\left(\lambda w \cdot w^{6} \rightarrow C 0\right)(\operatorname{map}((+) s)(s u m s t s))\)
with \(s s\)
    have C0-ms-ss-C0: map \(\left(\lambda w . w^{4} \rightarrow C 0\right)(\) sums ss \()=\)
                                    C0 \# Ms @ [sum-list ss \(\left.{ }^{\rightarrow} \rightarrow C 0\right]\)
    by (simp add: sums-snoc zero-permutation.rep-eq)
define \(r s\) where \(r s=\) arg-min length (word-for \(S\) (sum-list ss))
with \(\operatorname{assms}(1)\) have rs: rs \(\in\) lists \(S\) sum-list rs \(=\) sum-list ss
    using arg-min-natI[of \(\lambda\) rs. word-for \(S\) (sum-list ss) rs ss length] by auto
show ?thesis
proof (cases rs rule: list-cases-Cons-snoc)
    case Nil
    hence sum-list ss ' \(\rightarrow C 0=C 0\)
        using \(r s(2)\) by (fastforce simp add: zero-permutation.rep-eq)
    with CO-ms-ss-C0 show?thesis by simp
next
    case (Single r)
    from Single have min-gallery \(\left[C 0, r^{6} \rightarrow C 0\right]\)
        using \(r s(1)\) fundchamber fundchamber-S-chamber fundchamber-S-adjacent
                fundchamber-S-image-neq-fundchamber
    by (fastforce intro: min-gallery-adj)
    with Single C0-ms-ss-C0 Ms-def show ?thesis
        using rs(2) min-galleryD-min-betw[of C0 Ms sum-list ss ' \(\rightarrow\) C0 []]
                min-galleryD-gallery
    by (fastforce simp add: length-sums)
next
    case (Cons-snoc p qs q)
    define \(N s\) where \(N s=\operatorname{map}\left(\lambda w . w^{6} \rightarrow C 0\right)(\operatorname{map}((+) p)(s u m s q s))\)
    from assms rs-def have length rs < length ss
        using word-length-lt[of ss \(S\) ]
                reduced-word-for-length reduced-word-for-arg-min \([\) of ss \(S\) ]
        by force
    with Cons-snoc ss Ms-def Ns-def have length Ns < length Ms
        by (simp add: length-sums)
    moreover from Ns -def Cons-snoc
        have gallery (C0 \# Ns @ [sum-list ss ' \(\rightarrow\) C0])
        using rs S-list-image-gallery[of rs]
        by (auto simp add: sums-snoc zero-permutation.rep-eq)
    ultimately show ?thesis using C0-ms-ss-C0 not-min-galleryI-betw by auto
qed
qed
lemma S-list-not-min-gallery-double-split:
    assumes \(s s \in\) lists \(S\) ss \(\neq[] \neg \min\)-gallery \(\left(\operatorname{map}\left(\lambda w . w^{6} \rightarrow C 0\right)(s u m s ~ s s)\right)\)
shows
    \(\exists f g\) as s bs tcs.
        \((f, g) \in\) foldpairs \(\wedge\)
        sum-list as \({ }^{\prime} \rightarrow C 0 \in f \vdash \mathcal{C} \wedge\)
        sum-list \((\) as \(@[s]){ }^{\prime} \rightarrow C 0 \in g \vdash \mathcal{C} \wedge\)
        sum-list (as@[s]@bs) ' \(\rightarrow C 0 \in g \vdash \mathcal{C} \wedge\)
```

```
        sum-list (as@[s]@bs@[t]) ' \(\rightarrow C 0 \in f \vdash \mathcal{C} \wedge\)
        \(s s=a s @[s] @ b s @[t] @ c s\)
proof -
    define \(C s\) where \(C s=\operatorname{map}\left(\lambda w . w^{4} \rightarrow C 0\right)(\) sums ss)
    moreover from assms(1) Cs-def have gallery Cs
        using S-list-image-gallery by fastforce
    moreover from assms(1) Cs-def have \(\} \notin\) set (wall-crossings Cs)
    using \(S\)-list-image-crosses-walls by fastforce
    ultimately obtain \(f g A s A B B s E F s\)
        where \(f g:(f, g) \in\) foldpairs
        and sep \(\quad: A \in f \vdash \mathcal{C} \quad B \in g \vdash \mathcal{C} \quad E \in g \vdash \mathcal{C} \quad F \in f \vdash \mathcal{C}\)
        and decomp-cases:
        \(C s=A s @[A, B, F] @ F s \vee C s=A s @[A, B] @ B s @[E, F] @ F s\)
    using assms(3) not-min-gallery-double-split[of Cs]
    by blast
    show ?thesis
    proof (cases \(C s=A s @[A, B, F] @ F s)\)
        case True
        define \(b s::\) ' \(a\) permutation list where \(b s=[]\)
        from True Cs-def obtain as st cs where
            ss \(=\) as@ \([s, t] @ c s A=\) sum-list as \({ }^{\prime} \rightarrow C 0 B=\) sum-list \((a s @[s]) ' \rightarrow C 0\)
            \(F=\) sum-list (as@ \([s, t]) \quad \rightarrow C 0\)
            using pullback-sums-map-middle3[of \(\lambda w . w^{\prime} \rightarrow C 0\) ss As A B F Fs]
            by auto
    with \(\operatorname{sep}(1,2,4)\) bs-def have
            sum-list as \({ }^{'} \rightarrow C 0 \in f \vdash \mathcal{C}\) sum-list \((\) as \(@[s])\) ' \(\rightarrow C 0 \in g \vdash \mathcal{C}\)
            sum-list (as@ \([s] @ b s\) ) ' \(\rightarrow C 0 \in g \vdash \mathcal{C}\) sum-list (as@ \([s] @ b s @[t]){ }^{\prime} \rightarrow C 0 \in f \vdash \mathcal{C}\)
            \(s s=a s @[s] @ b s @[t] @ c s\)
            by auto
    with \(f g\) show ?thesis by blast
    next
        case False
        with Cs-def decomp-cases obtain as sbst cs where
        \(s s=a s @[s] @ b s @[t] @ c s A=\) sum-list as \({ }^{'} \rightarrow C 0 B=\) sum-list \((\) as \(@[s]){ }^{\prime} \rightarrow C 0\)
        \(E=\) sum-list (as@[s]@bs) ' \(\rightarrow C 0 F=\) sum-list (as@ \([s] @ b s @[t])^{\prime} \rightarrow C 0\)
        using pullback-sums-map-double-middle2 [
                of \(\lambda w . w^{6} \rightarrow C 0\) ss As A B Bs E F Fs
            ]
        by auto
    with sep have
        sum-list as \({ }^{'} \rightarrow C 0 \in f \vdash \mathcal{C}\) sum-list \((\) as \(@[s])\) ' \(\rightarrow C 0 \in g \vdash \mathcal{C}\)
        sum-list (as@ \([s] @ b s\) ) ' \(\rightarrow C 0 \in g \vdash \mathcal{C}\) sum-list \((a s @[s] @ b s @[t]){ }^{\cdot} \rightarrow C 0 \in f \vdash \mathcal{C}\)
        \(s s=a s @[s] @ b s @[t] @ c s\)
        by auto
    with \(f g\) show ?thesis by blast
    qed
qed
lemma fold-end-sum-chain-fg:
```

```
    fixes fg:: ' }a>\mp@subsup{}{}{\prime}
    defines s : s \equiv induced-automorph f g
    assumes fg: (f,g) f foldpairs
    and as: as lists S
    and }s:s\in
    and sep: sum-list as '}->C0\inf\vdash\mathcal{C}\mathrm{ sum-list (as@[s])' }->C0\ing\vdash\mathcal{C
    shows bs \inlists S\Longrightarrow
        s'sum-list(as@[s]@bs)'->C0=sum-list (as@bs)'->C0
proof-
    from fg obtain C where C: OpposedThinChamberComplexFoldings X f g C
    using foldpairs-def by fast
    show bs \inlists S\Longrightarrows'sum-list (as@[s]@bs) '->C0= sum-list (as@bs) '->C0
    proof (induct bs rule: rev-induct)
    case Nil
    from s as s sep C show ?case
        using sum-list-S-in-W[of as] sum-list-append[of as [s]]
                            fundchamber-WS-image-adjacent
        by (auto simp add:
                                    OpposedThinChamberComplexFoldings.indaut-adj-halfchsys-im-fg
                )
    next
    case (snoc b bs)
    define bC0 B where bC0 = b'->C0 and B= sum-list (as@bs) '->C0
    define }y\mathrm{ where }y=CO\capbC
    define z z'
        where z= s'sum-list (as@[s]@bs)'->y
            and z' = sum-list (as@bs) '}->
    from snoc B-def have B': s' sum-list (as@ [s]@bs)'->C0=B by simp
    obtain \varphi where \varphi: label-wrt C0 \varphi using ex-label-map by fast
    from bC0-def y-def snoc(2) obtain u where u:bC0 = insert u y
        using fundchamber-S-adjacent[of b] adjacent-sym
                        fundchamber-S-image-neq-fundchamber
                    adjacent-int-decomp[of bC0 C0]
        by (auto simp add: Int-commute)
    define v v
        where v=s(sum-list (as@[s]@bs) ->u)
            and v}\mp@subsup{v}{}{\prime}=\mathrm{ sum-list (as@bs) }->
    from bC0-def u v-def z-def v'-def z'-def
        have ins-vz:s'sum-list (as@[s]@bs@[b])'->C0= insert v z
        and ins-vz': sum-list (as@bs@[b]) '->C0 = insert v' z'
        using image-insert[of permutation (sum-list (as@[s]@bs)) u y, THEN sym]
                image-insert[
                        of s sum-list(as@[s]@bs)->u sum-list (as@[s]@bs)'->y,
                        THEN sym]
                image-insert[of permutation (sum-list (as@bs)) u y,THEN sym]
            by (auto simp add: plus-permutation.rep-eq image-comp)
```

```
    from as s snoc(2) have sums:
    sum-list(as@[s]@bs)\inW sum-list (as@bs)\inW
    sum-list (as@[s]@bs@[b]) \in W sum-list (as@bs@[b]) \inW
    using sum-list-S-in-W[of as@[s]@bs] sum-list-S-in-W[of as@bs]
        sum-list-S-in-W[of as@[s]@bs@[b]] sum-list-S-in-W[of as@bs@[b]]
    by auto
    from ubC0-def snoc(2) have u:u\in\bigcupX
    using fundchamber-S-chamber[of b] chamberD-simplex[of bC0] by auto
    moreover from as s snoc(2) u have sum-list(as@[s]@bs)->u\in\bigcupX
    using sums(1)
        ChamberComplexEndomorphism.vertex-map[OF W-endomorphism]
    by fastforce
    ultimately have }\varphiv=\varphi\mp@subsup{v}{}{\prime
    using s v-def v'-def sums(1,2) W-respects-labels[OF \varphi, of sum-list (as@[s]@bs)
```

```
    W-respects-labels[OF \(\varphi\), of sum-list (as@bs) u]
```

    W-respects-labels[OF \(\varphi\), of sum-list (as@bs) u]
    OpposedThinChamberComplexFoldings.indaut-resplabels [
    OpposedThinChamberComplexFoldings.indaut-resplabels [
        OF \(C \varphi\)
        OF \(C \varphi\)
    ]
    ]
    by simp
    moreover from s have chamber (insert v z) chamber (insert v' z')
using sums(3,4)
fundchamber-W-image-chamber[of sum-list (as@[s]@bs@[b])]
OpposedThinChamberComplexFoldings.indaut-chmap[
OF C
]
fundchamber-W-image-chamber[of sum-list (as@bs@[b])]
by (auto simp add: ins-vz[THEN sym] ins-vz'[THEN sym])
moreover from y-def z-def z'-def bC0-def B-def snoc(2) s have z\triangleleftB z'\triangleleftB
using }\mp@subsup{B}{}{\prime}\operatorname{sums(1,2) fundchamber-S-adjacent[of b]
fundchamber-S-image-neq-fundchamber[of b]
adjacent-int-facet1[of C0]
W-endomorphism[of sum-list (as@bs)]
W-endomorphism[of sum-list (as@[s]@bs)]
fundchamber fundchamber-W-image-chamber[of sum-list (as@[s]@bs)]
ChamberComplexEndomorphism.facet-map[of X]
OpposedThinChamberComplexFoldings.indaut-morph[
OF C
]
ChamberComplexEndomorphism.facet-map[
of X s sum-list(as@[s]@bs)'->C0
]
by auto

```
\(u\) ]
moreover from \(\operatorname{snoc}(2) B\)-def s have insert \(v z \neq B\) insert \(v^{\prime} z^{\prime} \neq B\)
    using sum-list-append[of as@[s]@bs [b]] sum-list-append[of as@bs [b]]
```

            fundchamber-next-WS-image-neq[of b sum-list (as@[s]@bs)]
            fundchamber-next-WS-image-neq[of b sum-list (as@bs)]
            OpposedThinChamberComplexFoldings.indaut-aut[
                    OF C
                            ]
                    ChamberComplexAutomorphism.bij bij-is-inj B'
                    inj-eq-image[
                of s sum-list(as@[s]@bs@[b])'->C0 sum-list (as@[s]@bs)'->C0
            ]
            by (auto simp add: ins-vz[THEN sym] ins-vz'[THEN sym])
    ultimately show ?case
            using B-def sums(2) fundchamber-W-image-chamber[of sum-list (as@bs)]
                label-wrt-eq-on-adjacent-vertex[OF \varphi, of v v'B z z]
            by (auto simp add: ins-vz[THEN sym] ins-vz'[THEN sym])
    qed
    qed
lemma fold-end-sum-chain-gf:
fixes fg:: ' }a>>''
defines s \equiv induced-automorph fg
assumes fg: (f,g) f foldpairs
and as lists S s\inS bs \inlists S
sum-list as '}->C0\ing\vdash\mathcal{C
sum-list (as@[s])'->C0 \inf\vdash\mathcal{C}
shows s'sum-list (as@[s]@bs)'->C0=sum-list (as@bs)'->C0
proof-
from fg obtain C where C: OpposedThinChamberComplexFoldings X f g C
using foldpairs-def by fast
from assms show ?thesis
using foldpairs-sym fold-end-sum-chain-fg[of g f as s bs]
OpposedThinChamberComplexFoldings.induced-automorphism-sym[OF C]
by simp
qed
lemma fold-middle-sum-chain:
assumes fg: (f,g) f foldpairs
and }S: as\inlists S s\inS bs\inlists S t\inS cs \inlists
and sep: sum-list as '->C0 \inf\vdash\mathcal{C}
sum-list (as@[s]) '->C0 \in g\vdash\mathcal{C}
sum-list(as@[s]@bs)'->C0 \ing\vdash\mathcal{C sum-list (as@ [s]@bs@ [t])'->C0}
\inf\vdash\mathcal{C}
shows sum-list(as@[s]@bs@[t]@cs) '->C0 = sum-list (as@bs@cs)'->C0
proof-
define s where s = induced-automorph f g
from fg obtain C
where OpposedThinChamberComplexFoldings XfgC
using foldpairs-def
by fast

```
```

    then have id'sum-list (as@[s]@bs@[t]@cs)'->C0 = sum-list (as@bs@cs)'->
    C0
using s-def fg S sep fold-end-sum-chain-gf[of f g as@[s]@bstcs]
fold-end-sum-chain-fg[of fg as s bs@cs]
by (simp add:
image-comp[THEN sym]
OpposedThinChamberComplexFoldings.indaut-order2[
THEN sym]
)
thus ?thesis by simp
qed
lemma S-list-not-min-gallery-deletion:
fixes ss :: 'a permutation list
defines w:w\equiv sum-list ss
assumes ss: ss\inlists S ss\not=[] ᄀ min-gallery (map (\lambdaw. w'->C0) (sums ss))
shows \existsab as bs cs.ss=as@[a]@bs@[b]@cs ^w=sum-list (as@bs@cs)
proof-
from w ss(1) have w-W:w\inW using sum-list-S-in- W by fast
define Cs where Cs = map ( }\lambdaw.\mp@subsup{w}{}{4}->C0)(sums ss
from ss obtain fg as s bst cs
where fg :(f,g)\infoldpairs
and sep : sum-list as '->C0 \inf\vdash\mathcal{C}
sum-list(as@[s])'->C0 G g\vdash\mathcal{C}
sum-list(as@[s]@bs)'->C0 \ing\vdash\mathcal{C}
sum-list(as@[s]@bs@[t])'->C0 Gf\vdash\mathcal{C}
and decomp: ss = as@[s]@bs@[t]@cs
using S-list-not-min-gallery-double-split[of ss]
by blast
from fg sep decomp w ss(1)
have w'->C0 = sum-list (as@bs@cs) '->C0
using fold-middle-sum-chain
by auto
with ss(1) decomp have w= sum-list (as@bs@cs)
using w-W sum-list-S-in-W[of as@bs@cs]
by (auto intro: inj-onD fundchamber-W-image-inj-on)
with decomp show ?thesis by fast
qed
lemma deletion:
ss $\in$ lists $S \Longrightarrow \neg$ reduced-word $S$ ss $\Longrightarrow$
$\exists a b$ as $b s c s . s s=a s @[a] @ b s @[b] @ c s \wedge$ sum-list ss =sum-list $(a s @ b s @ c s)$
using nil-reduced-word-for- 0 [of S] not-reduced-word-not-min-gallery
S-list-not-min-gallery-deletion
by fastforce
lemma PreCoxeterSystemWithDeletion: PreCoxeterSystemWithDeletion S
using $S$-add-order2 deletion by unfold-locales simp

```
```

lemma CoxeterSystem: CoxeterSystem S
using PreCoxeterSystemWithDeletion
PreCoxeterSystemWithDeletion.CoxeterSystem
by fast

```
end

\subsection*{5.4 Coxeter complexes}

\subsection*{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 \equivordering.PosetComplex (\supseteq) (\supset) \mathcal{P}
abbreviation }\Sigma\equiv\mathrm{ TheComplex
end

```

\subsection*{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}.\supseteqX)
proof-
have image-eq-UN: \f A. f'A=(\bigcupx\inA.{fx}) by blast
from assms obtain wT where w\inW T\inPow S X=w+o \langleT\rangle
using special-cosets-def by auto
thus ?thesis
using image-eq-UN[where f=(+o) w\circ genby]
finite-genset simplex-like-pow-above-in
OrderingSetIso.simplex-like-map[
OF special-coset-below-in-supset-ordering-iso, of T w
]

```
```

        special-cosets-below-in
    by force
    qed
lemma SimplicialComplex-\Sigma: SimplicialComplex }
unfolding TheComplex-def
proof (rule ordering.poset-is-SimplicialComplex)
show ordering (\supseteq) (\supset) ..
show }\forallX\in\mathcal{P}\mathrm{ . supset-simplex-like ( P.\X)
using simplex-like-special-cosets by fast
qed
lemma ComplexLikePoset-special-cosets:ComplexLikePoset (\supseteq) (\supset)\mathcal{P}
using simplex-like-special-cosets special-cosets-has-bottom special-cosets-have-glbs
by unfold-locales
abbreviation smap \equivordering.poset-simplex-map (`) (\supset) \mathcal{P}
lemmas smap-def = ordering.poset-simplex-map-def[OF supset-poset, of \mathcal{P}]
lemma ordsetmap-smap: \llbracketX\in\mathcal{P};Y\in\mathcal{P}; X\supseteqY\rrbracket\Longrightarrow smap X\subseteq smap Y
using ComplexLikePoset.ordsetmap-smap[OF ComplexLikePoset-special-cosets]
smap-def
by simp
lemma rev-ordsetmap-smap: \llbracketX\in\mathcal{P}; Y\in\mathcal{P}; smap X \subseteqsmap Y\rrbracket\LongrightarrowX\supseteqY
using ComplexLikePoset.rev-ordsetmap-smap[
OF ComplexLikePoset-special-cosets
]
smap-def
by simp
lemma smap-onto-PosetComplex: smap'\mathcal{P}=\Sigma
using ComplexLikePoset.smap-onto-PosetComplex[
OF ComplexLikePoset-special-cosets
]
smap-def TheComplex-def
by
simp
lemmas simplices-conv-special-cosets = smap-onto-PosetComplex[THEN sym]
lemma smap-into-PosetComplex: X\in\mathcal{P}\Longrightarrowsmap X
using smap-onto-PosetComplex by fast
lemma smap-pseudominimal:
w\inW\Longrightarrows\inS\Longrightarrow\operatorname{smap}(w+o\langleS-{s}\rangle)={w+o\langleS-{s}\rangle}
using smap-def[of w+o \langleS-{s}\rangle]
special-coset-pseudominimal-in-below-in[of wS-{s}]
exclude-one-is-pseudominimal-in-below-in[of w S-{s}]

```
```

    by auto
    ```
lemma exclude-one-notin-smap-singleton:
    \(s \in S \Longrightarrow w+o\langle S-\{s\}\rangle \notin \operatorname{smap}(w+o\langle\{s\}\rangle)\)
    using smap-def[of \(w+o\langle\{s\}\rangle]\)
        supset-pseudominimal-inD1[of \(\mathcal{P} . \supseteq(w+o\langle\{s\}\rangle) w+o\langle S-\{s\}\rangle]\)
        special-coset-subset-rev-mono[of \(\{s\} S-\{s\}]\)
    by auto
lemma maxsimp-vertices: \(w \in W \Longrightarrow s \in S \Longrightarrow w+o\langle S-\{s\}\rangle \in \operatorname{smap}\{w\}\)
    using special-cosetsI[of \(S-\{s\}]\) special-coset-singleton
        ordsetmap-smap[of \(w+o\langle S-\{s\}\rangle]\) smap-pseudominimal
    by (simp add: genby-lcoset-refl)
lemma maxsimp-singleton:
    assumes \(w \in W\)
    shows SimplicialComplex.maxsimp \(\Sigma(\operatorname{smap}\{w\})\)
proof (rule SimplicialComplex.maxsimpI, rule SimplicialComplex- \(\Sigma\) )
    from assms show smap \(\{w\} \in \Sigma\)
        using special-coset-singleton smap-into-PosetComplex by fast
next
    fix \(z\) assume \(z: z \in \Sigma \operatorname{smap}\{w\} \subseteq z\)
    from \(z(1)\) obtain \(X\) where \(X: X \in \mathcal{P} z=\operatorname{smap} X\)
        using simplices-conv-special-cosets by auto
    with assms \(z(2)\) have \(X=\{w\}\)
        using special-coset-singleton rev-ordsetmap-smap special-coset-nempty by fast
    with \(X(2)\) show \(z=\operatorname{smap}\{w\}\) by fast
qed
lemma maxsimp-is-singleton:
    assumes SimplicialComplex.maxsimp \(\Sigma x\)
    shows \(\exists w \in W . \operatorname{smap}\{w\}=x\)
proof-
    from assms obtain \(X\) where \(X: X \in \mathcal{P}\) smap \(X=x\)
        using SimplicialComplex.maxsimpD-simplex \([\) OF SimplicialComplex- \(\Sigma]\)
            simplices-conv-special-cosets
        by auto
    from \(X(1)\) obtain \(w T\) where \(w T: w \in W T \in\) Pow \(S X=w+o\langle T\rangle\)
    using special-cosets-def by auto
    from \(w T(1)\) have \(\{w\} \in \mathcal{P}\) using special-coset-singleton by fast
    moreover with \(X w T(3)\) have \(x \subseteq \operatorname{smap}\{w\}\)
    using genby-lcoset-refl ordsetmap-smap by fast
    ultimately show ?thesis
    using assms wT(1) smap-into-PosetComplex
                SimplicialComplex.maxsimpD-maximal[OF SimplicialComplex- \(\Sigma\) ]
    by fast
qed
lemma maxsimp-vertex-conv-special-coset:
\[
w \in W \Longrightarrow X \in \operatorname{smap}\{w\} \Longrightarrow \exists s \in S . X=w+o\langle S-\{s\}\rangle
\]
using smap-def special-coset-pseudominimal-in-below-in[of \(w\}]\)
by (simp add: genby-lcoset-empty)
lemma vertices: \(w \in W \Longrightarrow s \in S \Longrightarrow w+o\langle S-\{s\}\rangle \in \bigcup \Sigma\)
using maxsimp-singleton SimplicialComplex.maxsimpD-simplex[OF Simplicial-
Complex- \(\Sigma\) ]
maxsimp-vertices
by fast
lemma smap0-conv-special-subgroups:
smap \(0=(\lambda s .\langle S-\{s\}\rangle)\) ' \(S\)
using genby-0-closed maxsimp-vertices maxsimp-vertex-conv-special-coset by force
lemma \(S\)-bij-betw-chamber0: bij-betw \((\lambda s .\langle S-\{s\}\rangle) S(\) smap 0)
unfolding bij-betw-def
proof
show inj-on \((\lambda s .\langle S-\{s\}\rangle) S\)
proof (rule inj-onI)
fix \(s t\) show 【 \(s \in S ; t \in S ;\langle S-\{s\}\rangle=\langle S-\{t\}\rangle \rrbracket \Longrightarrow s=t\)
using inj-onD[OF special-subgroup-inj, of \(S-\{s\} S-\{t\}]\) by fast
qed
qed (rule smap0-conv-special-subgroups[THEN sym])
lemma smap-singleton-conv-W-image:
\(w \in W \Longrightarrow \operatorname{smap}\{w\}=((+o) w)\) ' \((\operatorname{smap} 0)\)
using genby-0-closed[of S] maxsimp-vertices[of 0] maxsimp-vertices[of w] maxsimp-vertex-conv-special-coset
by force
lemma W-lcoset-bij-betw-singletons:
assumes \(w \in W\)
shows bij-betw \(((+o) w)(\operatorname{smap} 0)(\operatorname{smap}\{w\})\)
unfolding bij-betw-def
proof (rule conjI, rule inj-onI)
fix \(X Y\) assume \(X Y: X \in \operatorname{smap} 0 Y \in \operatorname{smap} 0 w+o X=w+o Y\)
from \(X Y(1,2)\) obtain \(s x s y\) where \(X=\langle S-\{s x\}\rangle Y=\langle S-\{s y\}\rangle\)
using maxsimp-vertex-conv-special-coset[of \(0 \quad X]\) maxsimp-vertex-conv-special-coset[of \(0 \quad Y]\) genby-0-closed \([\) of \(S]\)
by auto
with \(X Y(3)\) show \(X=Y\)
using inj-onD[OF special-coset-inj, of \(w S-\{s x\} S-\{s y\}]\) by force
qed (rule smap-singleton-conv-W-image[THEN sym], rule assms)

\section*{lemma facets:}
assumes \(w \in W s \in S\)
shows \(\operatorname{smap}(w+o\langle\{s\}\rangle) \triangleleft \operatorname{smap}\{w\}\)
proof (
```

    rule facetrelI, rule exclude-one-notin-smap-singleton, rule assms(2),
    rule order-antisym
    )
show smap {w}\subseteqinsert (w+o \langleS - {s}\rangle)(smap (w+o \langle{s}\rangle))
proof
fix X assume X \in smap {w}
with assms(1) obtain t where t\inS X=w+o \langleS-{t}\rangle
using maxsimp-vertex-conv-special-coset by fast
with assms show X\in insert (w+o \langleS - {s}\rangle) (smap (w+o \langle{s}\rangle))
using exclude-one-is-pseudominimal-in-below-in smap-def
by (cases t=s) auto
qed
from assms show smap {w} \supseteq insert (w+o \langleS - {s}\rangle) (smap (w+o \{s}\rangle))
using genby-lcoset-refl special-cosetsI[of {s}] special-coset-singleton
ordsetmap-smap maxsimp-vertices
by fast
qed
lemma facets':w\inW\Longrightarrows\inS\Longrightarrow smap {w,w+s}\triangleleft smap {w}
using facets by (simp add: genset-order2-add genby-lcoset-order2)
lemma adjacent: w\inW\Longrightarrows\inS\Longrightarrow smap {w+s} ~ smap {w}
using facets'[of w s] genby-genset-closed genby-add-closed[of w S]
facets'[of w+s s]
by (
auto intro: adjacentI
simp add: genset-order2-add add.assoc insert-commute
)
lemma singleton-adjacent-0: s\inS \Longrightarrow smap {s} ~ smap 0
using genby-genset-closed genby-0-closed facets'[of 0] facets'[of s]
by (fastforce intro: adjacentI simp add: genset-order2-add insert-commute)
end

```

\subsection*{5.4.3 As a chamber complex}

Now we verify that a Coxeter complex is a chamber complex.
```

context CoxeterComplex
begin

```
abbreviation chamber \(\equiv\) SimplicialComplex.maxsimp \(\Sigma\)
abbreviation gallery \(\equiv\) SimplicialComplex.maxsimpchain \(\Sigma\)
lemmas chamber-singleton \(\quad=\) maxsimp-singleton
lemmas chamber-vertex-conv-special-coset \(=\) maxsimp-vertex-conv-special-coset
```

lemmas chamber-vertices = maxsimp-vertices
lemmas chamber-is-singleton $\quad=$ maxsimp-is-singleton
lemmas faces $=$ SimplicialComplex.faces $\quad[$ OF SimplicialComplex- $\Sigma$ ]
lemmas gallery-def $=$ SimplicialComplex.maxsimpchain-def [OF SimplicialCom-
plex- $\Sigma$ ]
lemmas gallery-rev $=$ SimplicialComplex.maxsimpchain-rev $[$ OF SimplicialCom-
plex- $\Sigma$ ]
lemmas chamberD-simplex $=$
SimplicialComplex.maxsimpD-simplex[OF SimplicialComplex- $\Sigma$ ]
lemmas gallery-CConsI =
SimplicialComplex.maxsimpchain-CConsI[OF SimplicialComplex- $\Sigma$ ]
lemmas gallery-overlap-join $=$
SimplicialComplex.maxsimpchain-overlap-join $[$ OF SimplicialComplex- $\Sigma]$
lemma word-gallery-to- 0 :
ss $\neq[] \Longrightarrow$ ss $\in$ lists $S \Longrightarrow \exists$ xs. gallery (smap \{sum-list ss $\} \#$ xs @ [smap 0])
proof (induct ss rule: rev-nonempty-induct)
case (single s)
hence gallery (smap \{sum-list [s]\} \# [] @ [smap 0])
using genby-genset-closed genby-0-closed chamber-singleton
singleton-adjacent-0 gallery-def
by auto
thus ?case by fast
next
case (snoc s ss)
from $\operatorname{snoc}(2,3)$ obtain $x s$ where gallery (smap $\{$ sum-list ss $\} \#$ xs @ [smap 0])
by auto
moreover from $\operatorname{snoc}(3)$ have chamber (smap \{sum-list (ss@ $[s])\}$ )
using special-subgroup-eq-sum-list chamber-singleton by fast
ultimately
have gallery (smap \{sum-list (ss@[s])\} \#
(smap $\{$ sum-list ss $\}$ \# xs) @ [smap 0])
using snoc(3) special-subgroup-eq-sum-list adjacent[of sum-list ss s]
by (auto intro: gallery-CConsI)
thus ?case by fast
qed
lemma gallery-to-0:
assumes $w \in W \quad w \neq 0$
shows $\exists x s$. gallery (smap $\{w\} \#$ xs @ [smap 0])
proof-
from $\operatorname{assms}(1)$ obtain $s s$ where ss: ss $\in$ lists $S w=$ sum-list ss
using special-subgroup-eq-sum-list by auto
with assms(2) show ?thesis using word-gallery-to-0[of ss] by fastforce
qed

```
```

lemma ChamberComplex-\Sigma:ChamberComplex \Sigma
proof (intro-locales, rule SimplicialComplex-\Sigma, unfold-locales)
fix }y\mathrm{ assume }y\in
from this obtain }X\mathrm{ where }X:X\in\mathcal{P}y=\operatorname{smap}
using simplices-conv-special-cosets by auto
from X(1) obtain wT where w\inWX=w+o \langleT\rangle
using special-cosets-def by auto
with }X\mathrm{ show }\existsx\mathrm{ . chamber }x\wedgey\subseteq
using genby-lcoset-refl special-coset-singleton ordsetmap-smap
chamber-singleton
by fastforce
next
fix }x
assume xy: }x\not=y\mathrm{ chamber x chamber }
from xy(2,3) obtain w w'
where ww':w\inW x = smap {w} w'\inW y = smap {w'}
using chamber-is-singleton
by blast
show \existszs.gallery (x \# zs @ [y])
proof (cases w=0 w'=0 rule: two-cases)
case both with }xy(1)w\mp@subsup{w}{}{\prime}(2,4) show ?thesis by fas
next
case one with ww'(2-4) show ?thesis
using gallery-to-0 gallery-rev by fastforce
next
case other with ww'(1,2,4) show ?thesis using gallery-to-0 by auto
next
case neither
from this ww' obtain xs ys
where gallery (x \# xs @ [smap 0]) gallery (smap 0 \# ys @ [y])
using gallery-to-0 gallery-rev
by force
hence gallery (x \# (xs @ smap 0 \# ys) @ [y])
using gallery-overlap-join[of x\#xs] by simp
thus ?thesis by fast
qed
qed
lemma card-chamber: chamber x \Longrightarrow card x = card S
using bij-betw-same-card[OF S-bij-betw-chamber0] chamber-singleton
genby-0-closed[of S]
ChamberComplex.chamber-card[OF ChamberComplex-\Sigma, of smap 0]
by
simp
lemma vertex-conv-special-coset:
X\in\bigcup\Sigma\Longrightarrow\existsws.w\inW\wedges\inS\wedge X = w +o \langleS-{s}\rangle
using ChamberComplex.simplex-in-max[OF ChamberComplex-\Sigma] chamber-is-singleton
chamber-vertex-conv-special-coset
by fast

```

\subsection*{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
by (
rule CoxeterComplex.intro, rule CoxeterSystem, unfold-locales,
rule finite-S
)

```
abbreviation \(\Sigma \equiv\) CoxeterComplex.TheComplex \(S\)
lemma S-list-not-min-gallery-not-reduced:
    assumes \(s s \neq[] \neg\) min-gallery ( \(\operatorname{map}\left(\lambda w . w^{〔} \rightarrow C 0\right)(\) sums ss) \()\)
    shows \(\neg\) reduced-word \(S\) ss
proof (cases ss \(\in\) lists \(S\) )
    case True
    obtain \(a b\) as bs cs
        where ss=as@[a]@bs@[b]@cs sum-list ss=sum-list (as@bs@cs)
        using S-list-not-min-gallery-deletion [OF True assms]
        by blast
    with True show ?thesis using not-reduced-word-for \([\) of \(a s @ b s @ c s]\) by auto
next
    case False thus ?thesis using reduced-word-for-lists by fast
qed
lemma reduced-S-list-min-gallery:
    \(s s \neq[] \Longrightarrow\) reduced-word \(S\) ss \(\Longrightarrow\) min-gallery \(\left(\operatorname{map}\left(\lambda w . w^{\prime} \rightarrow C 0\right)(\right.\) sums ss) \()\)
    using S-list-not-min-gallery-not-reduced by fast
lemma fundchamber-vertex-stabilizer1:
    fixes \(t\)
    defines \(v: v \equiv\) fundantivertex \(t\)
    assumes \(t w: t \in S \quad w \in W w \rightarrow v=v\)
    shows \(w \in\langle S-\{t\}\rangle\)
proof-
    from \(v t w(1)\) have \(v-C 0: v \in C 0\) using fundantivertex by simp
    define \(s s\) where \(s s=\) arg-min length (word-for \(S w\) )
    moreover
```

    have reduced-word S ss \Longrightarrow sum-list ss }->v=v\Longrightarrow\mathrm{ sum-list ss }\in\langleS-{t}
    proof (induct ss)
case (Cons s ss)
from Cons(2) have s-S: s\inS using reduced-word-for-lists by fastforce
from this obtain fg
where fg:(f,g)\infundfoldpairs s}=Abs-induced-automorph fg
by auto
from fg(1) have opp-fg: OpposedThinChamberComplexFoldings X fgC0
using fundfoldpairs-def by auto
define Cs where Cs = map ( }\lambdaw.\mp@subsup{w}{}{6}->C0)(sums (s\#ss)
with Cons(2) have minCs: min-gallery Cs
using reduced-S-list-min-gallery by fast
have sv: }s->v=
proof (cases ss rule: rev-cases)
case Nil with Cons(3) show ?thesis by simp
next
case (snoc ts t)
define Ms Cn
where Ms = map (\lambdaw. w'->C0) (map ((+) s) (sums ts))
and Cn = sum-list (s\#ss) '->C0
with snoc Cs-def have Cs=C0 \# Ms @ [Cn]
by (simp add: sums-snoc zero-permutation.rep-eq)
with minCs Cs-def fg have C0\inf\vdash\mathcal{C}Cn\ing\vdash\mathcal{C}
using sums-Cons-conv-append-tl[THEN sym,of s ss]
wall-crossings-subset-walls-betw[of C0 Ms Cn] fundfoldpairs-def
the-wall-betw-adj-fundchamber walls-betw-def
OpposedThinChamberComplexFoldings.basech-halfchsys(1)[
OF opp-fg
]
OpposedThinChamberComplexFoldings.separated-by-this-wall-fg[
OF opp-fg, of C0 Cn
]
by (auto simp add: zero-permutation.rep-eq)
moreover from Cons(3) Cn-def have v\inCn using v-C0 by force
ultimately show }s->v=
using v-C0 fg
OpposedThinChamberComplexFoldings.indaut-wallvertex[
OF opp-fg
]
by (simp add: permutation-conv-induced-automorph)
qed
moreover from Cons(3) have 0 sum-list ss }->v=s->
using s-S
by (simp add: plus-permutation.rep-eq S-order2-add[THEN sym])
ultimately have sum-list ss }->v=v\mathrm{ by (simp add:zero-permutation.rep-eq)
with Cons(1,2) have sum-list ss }\in\langleS-{t}
using reduced-word-Cons-reduce by auto
moreover from tw(1)v have }s\in\langleS-{t}
using sv s-S genby-genset-closed[of s S-{t}] fundantivertex-unstable

```
```

            by fastforce
    ultimately show ?case using genby-add-closed by simp
    qed (simp add: genby-0-closed)
    ultimately show ?thesis
    using tw(2,3) reduced-word-for-genby-sym-arg-min[OF S-sym]
        reduced-word-for-sum-list
    by fastforce
    qed
lemma fundchamber-vertex-stabilizer2:
assumes s: s\inS
defines v:v\equiv fundantivertex s
shows }w\in\langleS-{s}\rangle\Longrightarroww->v=
proof (erule genby.induct)
show }0->v=v\mathrm{ by (simp add:zero-permutation.rep-eq)
next
fix t assume }t\inS-{s
moreover with sv}\mathrm{ have vौC0@t }->C
using inj-on-eq-iff[OF fundantivertex-inj-on] fundchamber-S-adjacent
fundchamber-S-image-neq-fundchamber[THEN not-sym]
not-the1[OF adj-antivertex, of C0 t}->\mathrm{ C0 v] fundantivertex
unfolding fundantivertex-def
by auto
ultimately show }t->v=
using S-fixespointwise-fundchamber-image-int fixespointwiseD by fastforce
next
fix w w' assume ww': w->v=v w'->v=v
from}w\mp@subsup{w}{}{\prime}(2) have (-\mp@subsup{w}{}{\prime})->v=id
using plus-permutation.rep-eq[of - w' w}
by (auto simp add: zero-permutation.rep-eq[THEN sym])
with }w\mp@subsup{w}{}{\prime}(1)\mathrm{ show }(w-\mp@subsup{w}{}{\prime})->v=
using plus-permutation.rep-eq[of w-w` by simp
qed
lemma label-wrt-special-coset1:
assumes label-wrt C0 \varphi fixespointwise \varphi C0 w0\inW s\inS
defines v\equiv fundantivertex s
shows}\quad{w\inW.w->\varphi(w0->v)=w0->v}=w0+o\langleS-{s}
proof-
from assms(4,5) have v-C0:v\inC0 using fundantivertex[of s] by simp
show ?thesis
proof (rule seteqI)
fix w assume w\in{w\inW.w->(\varphi(w0->v)) = w0->v}
hence w: w\inW w->(\varphi(w0->v))=w0->v by auto
from assms(2,3) have (-w0 +w) ->v=0->v
using w(2) v-C0 fundchamber chamberD-simplex
W-respects-labels[OF assms(1)] plus-permutation.rep-eq[of -w0 w0]
by (fastforce simp add: plus-permutation.rep-eq fixespointwiseD)
with assms(3-5) show w\inw0 +o \langleS-{s}\rangle

```
using \(w(1)\) genby-uminus-add-closed[of w0 \(S w]\) fundchamber-vertex-stabilizer 1
by (force simp add: zero-permutation.rep-eq elt-set-plus-def) next
fix \(w\) assume \(w: w \in w 0+o\langle S-\{s\}\rangle\)
from this obtain \(w 1\) where \(w 1: w 1 \in\langle S-\{s\}\rangle w=w 0+w 1\)
using elt-set-plus-def by blast
moreover with \(w \operatorname{assms}(3)\) have \(w-W: w \in W\)
using genby-mono[of \(S-\{s\} \quad S]\) genby-add-closed by fastforce
ultimately show \(w \in\{w \in W . w \rightarrow(\varphi(w 0 \rightarrow v))=w 0 \rightarrow v\}\)
using assms(2-5) v-C0 fundchamber chamberD-simplex
\(W\)-respects-labels [OF assms(1), of w0 v]
fundchamber-vertex-stabilizer2[of s w1]
by (fastforce simp add: fixespointwiseD plus-permutation.rep-eq) qed
qed
lemma label-wrt-special-coset1':
assumes label-wrt C0 fixespointwise \(\varphi C 0 w 0 \in W v \in C 0\)
defines \(s \equiv\) fundantipermutation \(v\)
shows \(\quad\{w \in W . w \rightarrow \varphi(w 0 \rightarrow v)=w 0 \rightarrow v\}=w 0+o\langle S-\{s\}\rangle\)
using assms fundantipermutation1 fundantivertex-bij-betw bij-betw-f-the-inv-into-f label-wrt-special-coset1 \(\left[\begin{array}{lll}\text { of } & \varphi & w 0\end{array}\right]\)
by fastforce
lemma label-wrt-special-coset2':
assumes label-wrt C0 fixespointwise \(\varphi C 0 w 0 \in W v \in w 0^{s} \rightarrow C 0\)
defines \(s \equiv\) fundantipermutation ( \(\varphi v\) )
shows \(\quad\{w \in W . w \rightarrow \varphi v=v\}=w 0+o\langle S-\{s\}\rangle\)
using assms fundchamber chamberD-simplex \(W\)-respects-labels label-wrt-special-coset1'[OF assms(1-3)]
by (fastforce simp add: fixespointwiseD)
lemma label-stab-map-W-fundchamber-image:
assumes label-wrt C0 \(\varphi\) fixespointwise \(\varphi C 0 w 0 \in W\)
defines \(\psi \equiv \lambda v .\{w \in W . w \rightarrow(\varphi v)=v\}\)
shows \(\psi^{\prime}\left(w 0^{\prime} \rightarrow C 0\right)=\) CoxeterComplex.smap \(S\{w 0\}\)
proof (rule seteqI)
from assms
show \(\bigwedge x . x \in\) CoxeterComplex.smap \(S\{w 0\} \Longrightarrow x \in \psi^{‘}\left(w 0^{‘} \rightarrow C 0\right)\)
using CoxeterComplex.chamber-vertex-conv-special-coset[
OF CoxeterComplex, of w0

\section*{]}
label-wrt-special-coset1 fundantivertex
by fastforce
next
fix \(x\) assume \(x \in \psi^{\prime}\left(w 0^{4} \rightarrow C 0\right)\)
from this obtain \(v\) where \(v: v \in w 0^{6} \rightarrow C 0 x=\psi v\) by fast
with assms have \(x=w 0+o\langle S-\{\) fundantipermutation \((\varphi v)\}\rangle\)
using label-wrt-special-coset2' by fast
moreover from \(v(1) \operatorname{assms}(3)\) have \(v \in \bigcup X\)
using fundchamber chamberD-simplex \(W\)-endomorphism ChamberComplexEndomorphism.vertex-map
by fastforce
ultimately show \(x \in\) CoxeterComplex.smap \(S\{w 0\}\)
using assms \((1,3)\) label-wrt-elt-image fundantipermutation1
CoxeterComplex.chamber-vertices[OF CoxeterComplex]
by fastforce
qed
lemma label-stab-map-chamber-map:
assumes \(\varphi\) : label-wrt \(C 0 \varphi\) fixespointwise \(\varphi C 0\)
and \(\quad C\) : chamber \(C\)
defines \(\psi: \psi \equiv \lambda v .\{w \in W . w \rightarrow(\varphi v)=v\}\)
shows CoxeterComplex.chamber \(S\left(\psi^{\prime} C\right)\)
proof-
from \(C\) obtain \(w\) where \(w: w \in W C=w^{\iota} \rightarrow C 0\)
using chamber-eq-W-image by fast
with \(\varphi \psi\) have \(\psi^{\prime} C=\) CoxeterComplex.smap \(S\{w\}\)
using label-stab-map-W-fundchamber-image by simp
with \(w(1)\) show ?thesis
using CoxeterComplex.chamber-singleton[OF CoxeterComplex] by simp qed
lemma label-stab-map-inj-on-vertices:
assumes \(\varphi\) : label-wrt C0 \(\varphi\) fixespointwise \(\varphi\) C0
defines \(\psi: \psi \equiv \lambda v .\{w \in W . w \rightarrow(\varphi v)=v\}\)
shows inj-on \(\psi(\bigcup X)\)
proof (rule inj-onI)
fix \(v 1\) v2 assume \(v: v 1 \in \bigcup X v 2 \in \bigcup X \psi v 1=\psi v 2\)
from \(v(1,2)\) have \(\varphi v: \varphi v 1 \in C 0 \varphi v 2 \in C 0\)
using label-wrt-elt-image \([O F \varphi(1)]\) by auto
define \(s 1\) s2 where \(s 1=\) fundantipermutation \((\varphi v 1)\) and \(s 2=\) fundantiper -
mutation ( \(\varphi\) v2)
from \(v(1,2)\) obtain \(w 1 w_{2}\) where \(w 1 \in W v 1 \in w 1^{\prime} \rightarrow C 0 \quad w 2 \in W \quad v 2 \in w 2^{\prime} \rightarrow C 0\)
using simplex-in-max chamber-eq-W-image by blast
with assms s1-def s2-def have \(\psi v: \psi v 1=w 1+o\langle S-\{s 1\}\rangle \psi v 2=w 2+o\)
\(\langle S-\{s 2\}\rangle\)
using label-wrt-special-coset2' by auto
with \(v(3)\) have \(w 1+o\langle S-\{s 1\}\rangle=w 2+o\langle S-\{s 2\}\rangle\)
using label-wrt-special-coset2' by auto
with \(s 1\)-def \(s 2\)-def have \(\varphi v 1=\varphi v 2\)
using PreCoxeterSystemWithDeletion.special-coset-eq-imp-eq-gensets[
OF PreCoxeterSystemWithDeletion, of \(S-\{s 1\} S-\{s 2\}\) w1 w2 ]
\(\varphi v\) fundantipermutation1[ of \(\varphi\) v1] fundantipermutation1[ of \(\varphi\) v2]
bij-betw-f-the-inv-into-f[OF fundantivertex-bij-betw, of \(\varphi\) v1]
bij-betw-f-the-inv-into-f[OF fundantivertex-bij-betw, of \(\varphi\) v2]
```

    by fastforce
    with v(3) \psi show v1=v2
    using \psiv(1) genby-0-closed[of S-{s1}] lcoset-refl[of \langleS-{s1}\ranglew1]
    by fastforce
    qed
lemma label-stab-map-surj-on-vertices:
assumes label-wrt C0 \varphi fixespointwise \varphi C0
defines }\psi\equiv\lambdav.{w\inW.w->(\varphiv)=v
shows }\mp@subsup{\psi}{}{`}(\bigcupX)=\bigcup proof (rule seteqI)     fix }u\mathrm{ assume }u\in\mp@subsup{\psi}{}{`}(\X
from this obtain v where v:v\in\bigcupX u=\psi v by fast
from v(1) obtain w where w\inW v\in\mp@subsup{w}{}{4}->C0
using simplex-in-max chamber-eq-W-image by blast
with assms v show }u\in\bigcup
using label-wrt-special-coset2' label-wrt-elt-image[OF assms(1)]
fundantipermutation1 CoxeterComplex.vertices[OF CoxeterComplex]
by auto
next
fix }u\mathrm{ assume }u\in\
from this obtain ws where w\inW s\inSu=w+o \langleS-{s}\rangle
using CoxeterComplex.vertex-conv-special-coset[OF CoxeterComplex] by blast
with assms show }u\in\mp@subsup{\psi}{}{\prime}(\bigcupX
using label-wrt-special-coset1 fundantivertex fundchamber chamberD-simplex
W-endomorphism ChamberComplexEndomorphism.vertex-map
by fast
qed
lemma label-stab-map-bij-betw-vertices:
assumes label-wrt C0 \varphi fixespointwise \varphi C0
defines }\psi\equiv\lambdav.{w\inW.w->(\varphiv)=v
shows bij-betw \psi (\bigcupX)(\bigcup\Sigma)
unfolding bij-betw-def
using assms label-stab-map-inj-on-vertices label-stab-map-surj-on-vertices
by auto
lemma label-stab-map-bij-betw-W-chambers:
assumes label-wrt C0 \varphi fixespointwise \varphi C0w0\inW
defines }\psi\equiv\lambdav.{w\inW.w->(\varphiv)=v
shows bij-betw \psi (w0'->C0) (CoxeterComplex.smap S {w0})
unfolding bij-betw-def
proof (rule conjI, rule inj-on-inverseI)
define f1 f2
where f1 = the-inv-into (CoxeterComplex.smap S 0) ((+o) w0)
and f2 = the-inv-into S (\lambdas. \langleS-{s}\rangle)
define g}\mathrm{ where g=(( }->\mathrm{ w0) ० fundantivertex ○f2 ○f1
from assms(3) have inj-opw0: inj-on ((+o) w0) (CoxeterComplex.smap S 0)

```
using bij-betw-imp-inj-on[OF CoxeterComplex. W-lcoset-bij-betw-singletons] CoxeterComplex
by fast
have inj-genby-minus-s: inj-on ( \(\lambda s .\langle S-\{s\}\rangle) S\)
using bij-betw-imp-inj-on[OF CoxeterComplex.S-bij-betw-chamber0]
CoxeterComplex
by fast
fix \(v\) assume \(v: v \in w 0^{〔} \rightarrow C 0\)
from this obtain \(v 0\) where \(v 0: v 0 \in C 0 v=w 0 \rightarrow v 0\) by fast
from \(v 0(1)\) have fap-v0: fundantipermutation \(v 0 \in S\)
using fundantipermutation 1 by auto
with \(\operatorname{assms}(3)\)
have \(v 0^{\prime}:\langle S-\{\) fundantipermutation \(v 0\}\rangle \in\) CoxeterComplex.smap \(S 0\)
using genby-0-closed[of S]
CoxeterComplex.chamber-vertices[OF CoxeterComplex, of 0]
by \(\operatorname{simp}\)
from \(v 0\) assms have \(\psi v=w 0+o\langle S-\{\) fundantipermutation v0 \(\}\rangle\)
using label-wrt-special-coset1' by simp
with f1-def assms(3) f2-def v0 g-def show \(g(\psi v)=v\)
using \(v 0^{\prime}\) fap-v0 the-inv-into-f-f[OF inj-opw0]
the-inv-into-f-f[OF inj-genby-minus-s]
bij-betw-f-the-inv-into-f[OF fundantivertex-bij-betw]
by \(\operatorname{simp}\)
next
from assms show \(\psi^{\prime}\left(w 0^{\prime} \rightarrow C 0\right)=\) CoxeterComplex.smap \(S\{w 0\}\)
using label-stab-map-W-fundchamber-image by simp
qed
lemma label-stab-map-surj-on-simplices:
assumes \(\varphi\) : label-wrt C0 \(\varphi\) fixespointwise \(\varphi\) C0
defines \(\psi: \psi \equiv \lambda v .\{w \in W . w \rightarrow(\varphi v)=v\}\)
shows \(\quad \psi \vdash X=\Sigma\)
proof (rule seteqI)
fix \(y\) assume \(y \in \psi \vdash X\)
from this obtain \(x\) where \(x: x \in X y=\psi^{\prime} x\) by fast
from \(x(1)\) obtain \(C\) where chamber \(C x \subseteq C\) using simplex-in-max by fast
with assms \(x\) (2) show \(y \in \Sigma\)
using label-stab-map-chamber-map
CoxeterComplex.chamberD-simplex[OF CoxeterComplex]
CoxeterComplex.faces[OF CoxeterComplex, of \(\left.\psi^{\prime} C y\right]\)
by auto
next
fix \(y\) assume \(y \in \Sigma\)
from this obtain \(z\) where \(z\) : CoxeterComplex.chamber \(S z y \subseteq z\)
using ChamberComplex.simplex-in-max [
OF CoxeterComplex.ChamberComplex- \(\Sigma\), OF CoxeterComplex
```

        ]
    by fast
    from z(1) obtain w where w: w\inWz=CoxeterComplex.smap S {w}
    using CoxeterComplex.chamber-is-singleton[OF CoxeterComplex] by fast
    with assms have bij-betw \psi ( w'->C0) z
    using label-stab-map-bij-betw-W-chambers by fast
    hence 1: bij-betw ((`) \psi) (Pow ( }\mp@subsup{w}{}{\prime}->C0))(\mathrm{ Pow z)
    using bij-betw-imp-bij-betw-Pow by fast
    define x where x: x \equiv the-inv-into (Pow ( w'->C0)) ((`)\psi) y
    with z(2) have }x\subseteq\mp@subsup{w}{}{6}->C0\mathrm{ using bij-betw-the-inv-into-onto[OF 1] by auto
    with w(1) have }x\in
        using faces fundchamber-W-image-chamber chamberD-simplex
        by fastforce
    moreover from xz(2) have y= \psi'}
    using bij-betw-f-the-inv-into-f[OF 1] by simp
    ultimately show }y\in\psi\vdashX by fas
    qed
lemma label-stab-map-iso-to-coxeter-complex:
assumes label-wrt C0 \varphi fixespointwise \varphi C0
defines }\psi\equiv\lambdav.{w\inW.w->(\varphiv)=v
shows ChamberComplexIsomorphism X \Sigma\psi
proof
rule ChamberComplexIsomorphism.intro,
rule ChamberComplexMorphism.intro
)
show ChamberComplex X ..
show ChamberComplex \Sigma
using CoxeterComplex CoxeterComplex.ChamberComplex- }\Sigma\mathrm{ by fast
from assms show ChamberComplexMorphism-axioms X \Sigma\psi
using label-stab-map-chamber-map
CoxeterComplex.card-chamber[OF CoxeterComplex]
card-S-chamber
by unfold-locales auto
from assms show ChamberComplexIsomorphism-axioms X \Sigma\psi
using label-stab-map-bij-betw-vertices label-stab-map-surj-on-simplices
by unfold-locales auto
qed
lemma ex-iso-to-coxeter-complex':
\exists}\mathrm{ . ChamberComplexIsomorphism X (CoxeterComplex.TheComplex S) }
using CoxeterComplex ex-label-retraction label-stab-map-iso-to-coxeter-complex
by force
lemma ex-iso-to-coxeter-complex:
\exists ::'a permutation set. CoxeterComplex S ^
( \exists\psi. ChamberComplexIsomorphism X (CoxeterComplex.TheComplex S) \psi)
using CoxeterComplex ex-iso-to-coxeter-complex' by fast

```
end

\section*{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.

\author{
theory Building \\ imports Coxeter
}
begin

\subsection*{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.

\subsection*{6.1.1 Locale and basic facts}
```

locale ChamberComplexWithApartmentSystem $=$ ChamberComplex X
for $X::$ ' $a$ set set

+ fixes $\mathcal{A}::$ 'a set set set
assumes subcomplexes $: A \in \mathcal{A} \Longrightarrow$ ChamberSubcomplex $A$
and thincomplexes $: A \in \mathcal{A} \Longrightarrow$ ThinChamberComplex $A$
and no-trivial-apartments: $\} \notin \mathcal{A}$
and containtwo :
chamber $C \Longrightarrow$ chamber $D \Longrightarrow \exists A \in \mathcal{A} . C \in A \wedge D \in A$
and intersecttwo :
$\llbracket A \in \mathcal{A} ; A^{\prime} \in \mathcal{A} ; x \in A \cap A^{\prime} ; C \in A \cap A^{\prime} ;$ chamber $C \rrbracket \Longrightarrow$
$\exists f$. ChamberComplexIsomorphism $A A^{\prime} f \wedge$ fixespointwise $f x \wedge$
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 $\quad$ [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}\wedgeC\inA ^D\inA)
lemma supapartmentD

```
```

    assumes CD: chamber C chamber D
    defines A:A \equiv supapartment C D
    shows }A\in\mathcal{A}C\inA D\in
    proof-
from CD have 1: \exists A. A\in\mathcal{A}\wedgeC\inA\wedge D\inA using containtwo by fast
from }A\mathrm{ show }A\in\mathcal{A}C\inA D\inA using someI-ex[OF 1] by aut
qed
lemma iso-fixespointwise-chamber-in-int-apartments:
assumes apartments: }A\in\mathcal{A}\mp@subsup{A}{}{\prime}\in\mathcal{A
and chamber : chamber C C }\inA\cap\mp@subsup{A}{}{\prime
and iso : ChamberComplexIsomorphism A A'f fixespointwise f C
shows fixespointwise f (U(A\capA})
proof (rule fixespointwiseI)
fix v}\mathrm{ assume }v\in\bigcup(A\cap\mp@subsup{A}{}{\prime}
from this obtain x where x: x\inA\capA' v\inx by fast
from apartments x(1) chamber intersecttwo[of A A] obtain g
where g: ChamberComplexIsomorphism A A'g
fixespointwise g x fixespointwise g C
by force
from assms g(1,3) have fun-eq-on fg}(\bigcupA
using chamber-in-apartment
by (auto intro:
apartment-standard-uniqueness-isomorphs
fixespointwise2-imp-eq-on
)
with x g(2) show fv=id v using fixespointwiseD fun-eq-onD by force
qed
lemma strong-intersecttwo:
A\in\mathcal{A; A'}\mathcal{A}; chamber C;C \in A\capA'\rrbracket\Longrightarrow
\existsf.ChamberComplexIsomorphism A A'f}^\wedge\mathrm{ fixespointwise f (U (A@A'))
using intersecttwo[of A A']
iso-fixespointwise-chamber-in-int-apartments[of A A' C]
by force
end

```

\subsection*{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 \(A \in \mathcal{A} A^{\prime} \in \mathcal{A}\) chamber \(C C \in A \cap A^{\prime}\)
shows \(\exists!f\). ChamberComplexIsomorphism \(A A^{\prime} f \wedge\)
fixespointwise \(f\left(\bigcup\left(A \cap A^{\prime}\right)\right) \wedge\) fixespointwise \(f(-\bigcup A)\)
- The third clause in the conjunction is to facilitate uniqueness. proof (rule ex-ex1I)
from assms obtain \(f\)
where \(f\) : ChamberComplexIsomorphism A A'f fixespointwise \(f\left(\bigcup\left(A \cap A^{\prime}\right)\right)\)
using strong-intersecttwo
by fast
define \(f^{\prime}\) where \(f^{\prime}=\) restrict1 \(f(\bigcup A)\)
from \(f(1) f^{\prime}\)-def have ChamberComplexIsomorphism \(A A^{\prime} f^{\prime}\)
by (fastforce intro: ChamberComplexIsomorphism.iso-cong fun-eq-onI)
moreover from \(f(2) f^{\prime}\)-def have fixespointwise \(f^{\prime}\left(\bigcup\left(A \cap A^{\prime}\right)\right)\)
using fun-eq-onI[ of \(\left.\bigcup\left(A \cap A^{\prime}\right) f^{\prime} f\right]\)
by (fastforce intro: fixespointwise-cong)
moreover from \(f^{\prime}\)-def have fixespointwise \(f^{\prime}(-\bigcup A)\)
by (auto intro: fixespointwiseI)
ultimately
show \(\exists f\). ChamberComplexIsomorphism \(A A^{\prime} f \wedge\) fixespointwise \(f\left(\bigcup\left(A \cap A^{\prime}\right)\right) \wedge\) fixespointwise \(f(-\bigcup A)\)
by fast
next
fix \(f g\)
assume ChamberComplexIsomorphism \(A A^{\prime} f \wedge\)
fixespointwise \(f\left(\bigcup\left(A \cap A^{\prime}\right)\right) \wedge\) fixespointwise \(f(-\bigcup A)\)
ChamberComplexIsomorphism \(A A^{\prime} g \wedge\)
fixespointwise \(g\left(\bigcup\left(A \cap A^{\prime}\right)\right) \wedge\) fixespointwise \(g(-\bigcup A)\)
with assms show \(f=g\)
using chamber-in-apartment fixespointwise2-imp-eq-on[of fCg] fun-eq-on-cong fixespointwise-subset of \(\left.f \bigcup\left(A \cap A^{\prime}\right) C\right]\)
fixespointwise-subset of \(\left.g \bigcup\left(A \cap A^{\prime}\right) C\right]\)
apartment-standard-uniqueness-isomorphs
by (blast intro: fun-eq-on-set-and-comp-imp-eq)
qed
definition the-apartment-iso \(::\) 'a set set \(\Rightarrow{ }^{\prime} a\) set set \(\Rightarrow\left({ }^{\prime} a \Rightarrow{ }^{\prime} a\right)\)
where the-apartment-iso \(A A^{\prime} \equiv\)
(THE f. ChamberComplexIsomorphism \(A A^{\prime} f \wedge\)
fixespointwise \(f\left(\bigcup\left(A \cap A^{\prime}\right)\right) \wedge\) fixespointwise \(\left.f(-\bigcup A)\right)\)
lemma the-apartment-isoD:
assumes \(\quad A \in \mathcal{A} A^{\prime} \in \mathcal{A}\) chamber \(C C \in A \cap A^{\prime}\)
defines \(f \equiv\) the-apartment-iso \(A A^{\prime}\)
shows ChamberComplexIsomorphism \(A A^{\prime} f\) fixespointwise \(f\left(\bigcup\left(A \cap A^{\prime}\right)\right)\)
fixespointwise \(f(-\bigcup A)\)
using assms theI '[OF ex1-apartment-iso]
unfolding the-apartment-iso-def
by auto
lemmas the-apartment-iso-apartment-chamber-map \(=\)
ChamberComplexIsomorphism.chamber-map [OF the-apartment-isoD(1)]
lemmas the-apartment-iso-apartment-simplex-map \(=\)
ChamberComplexIsomorphism.simplex-map [OF the-apartment-isoD (1)]
lemma the-apartment-iso-chamber-map:
\(\llbracket A \in \mathcal{A} ; B \in \mathcal{A}\); chamber \(C ; C \in A \cap B ;\) chamber \(D ; D \in A \rrbracket \Longrightarrow\) chamber (the-apartment-iso \(A B{ }^{\prime} D\) )
using chamber-in-apartment \([\) of \(A]\) apartment-chamber
the-apartment-iso-apartment-chamber-map
by auto
lemma the-apartment-iso-comp:
assumes apartments: \(A \in \mathcal{A} A^{\prime} \in \mathcal{A} A^{\prime \prime} \in \mathcal{A}\)
and chamber : chamber \(C C \in A \cap A^{\prime} \cap A^{\prime \prime}\)
defines \(f \equiv\) the-apartment-iso \(A A^{\prime}\)
and \(\quad g \equiv\) the-apartment-iso \(A^{\prime} A^{\prime \prime}\)
and \(\quad h \equiv\) the-apartment-iso \(A A^{\prime \prime}\)
defines \(g f \equiv\) restrict1 \((g \circ f)(\bigcup A)\)
shows \(h=g f\)
proof (
rule fun-eq-on-set-and-comp-imp-eq,
rule apartment-standard-uniqueness-isomorphs, rule apartments(3)
)
from \(g f\)-def have \(g f\)-cong1: fun-eq-on gf ( \(g \circ f\) ) ( \(\bigcup A)\)
by (fastforce intro: fun-eq-onI)
from \(g f\)-def have \(g f\)-cong2: fixespointwise \(g f(-\bigcup A)\)
by (auto intro: fixespointwiseI)
from \(\operatorname{apartments}(1,3)\) chamber \(h\)-def
show ChamberComplexIsomorphism \(A A^{\prime \prime} h\)
using the-apartment-iso \(D(1)\)
by fast
from apartments chamber \(f\)-def \(g\)-def
show ChamberComplexIsomorphism \(A A^{\prime \prime} g f\)
using ChamberComplexIsomorphism.iso-cong[OF - gf-cong1]
ChamberComplexIsomorphism.iso-comp the-apartment-isoD (1)
by blast
from apartments(1) chamber show ChamberComplex.chamber A C using chamber-in-apartment by fast
show fun-eq-on \(h\) gf \(C\)
proof (rule fixespointwise2-imp-eq-on)
from \(\operatorname{assms}(1,3)\) chamber \(h\)-def show fixespointwise \(h C\) using fixespointwise-subset the-apartment-isoD(2) by blast have fun-eq-on gf \((g \circ f)\left(\cup\left(A \cap A^{\prime} \cap A^{\prime \prime}\right)\right)\)
using fun-eq-on-subset[OF gf-cong1, of \(\left.\bigcup\left(A \cap A^{\prime} \cap A^{\prime \prime}\right)\right]\) by fast moreover from \(f\)-def \(g\)-def apartments chamber have fixespointwise \((g \circ f)\left(\bigcup\left(A \cap A^{\prime} \cap A^{\prime}\right)\right)\) using fixespointwise-comp[of \(\left.f \bigcup\left(A \cap A^{\prime} \cap A^{\prime \prime}\right) g\right]\)
fixespointwise-subset[
```

                OF the-apartment-isoD(2), of - - C \bigcup(A\capA'\capA')
                ]
        by auto
    ultimately have fixespointwise gf ( \bigcup (A\cap\mp@subsup{A}{}{\prime}\cap\mp@subsup{A}{}{\prime\prime}))
        using fixespointwise-cong[of gf g\circf] by fast
    with chamber(2) show fixespointwise gf C
        using fixespointwise-subset by auto
    qed
    from h-def apartments(1,3) chamber show fun-eq-on h gf (- \bigcupA)
    using the-apartment-isoD(3) gf-cong2 by (auto intro: fun-eq-on-cong)
    qed
lemma the-apartment-iso-int-im:
assumes }A\in\mathcal{A}\mp@subsup{A}{}{\prime}\in\mathcal{A}\mathrm{ chamber C C }\inA\cap\mp@subsup{A}{}{\prime}x\inA\cap\mp@subsup{A}{}{\prime
defines f}\equiv\mathrm{ the-apartment-iso }A\mp@subsup{A}{}{\prime
shows }\quad\mp@subsup{f}{}{\prime}x=
using assms the-apartment-isoD(2) fixespointwise-im[of f \bigcup(A\capA')x]
by
fast
end

```

\subsection*{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 set set }=>\mp@subsup{|}{}{\prime}a\mathrm{ set }=>('a>'a
where canonical-retraction A C=
restrict1 (\lambdav. the-apartment-iso (supapartment (supchamber v) C) A v)
(\bigcupX)

```
lemma canonical-retraction-retraction: assumes \(A \in \mathcal{A}\) chamber \(C C \in A \quad v \in \bigcup A\)
    shows canonical-retraction \(A C v=v\)
proof-
    define \(D\) where \(D=\) supchamber \(v\)
    define \(B\) where \(B=\) supapartment \(D C\)
    from \(D\)-def \(\operatorname{assms}(1,4)\) have \(D\)-facts: chamber \(D v \in D\)
using apartment-simplices supchamber \(D[o f v]\) by auto
from \(B\)-def \(\operatorname{assms}(2)\) have \(B\)-facts: \(B \in \mathcal{A} \quad D \in B \quad C \in B\) using \(D\)-facts(1) supapartment \(D[\) of \(D C]\) by auto
from \(\operatorname{assms}(1,4)\) have \(v \in \bigcup(B \cap A)\)
using \(D\)-facts(2) B-facts (1,2) apartment-vertex-set-int by fast
with \(\operatorname{assms}(1-3) D\)-def \(B\)-def show ?thesis
using canonical-retraction-def \(B\)-facts \((1,3)\) fixespointwise \(D[o f-\bigcup(B \cap A) v]\) the-apartment-isoD(2)[of BAC]
by \(\operatorname{simp}\)
qed
lemma canonical-retraction-simplex-retraction1:
\(\llbracket A \in \mathcal{A} ;\) chamber \(C ; C \in A ; a \in A \rrbracket \Longrightarrow\)
fixespointwise (canonical-retraction A C) a
using canonical-retraction-retraction by (force intro: fixespointwiseI)
lemma canonical-retraction-simplex-retraction2:
\(\llbracket A \in \mathcal{A}\); chamber \(C ; C \in A ; a \in A \rrbracket \Longrightarrow\) canonical-retraction \(A C\) ' \(a=a\)
using canonical-retraction-simplex-retraction1 fixespointwise-im \([\) of - a a] by simp
lemma canonical-retraction-uniform:
assumes apartments: \(A \in \mathcal{A} B \in \mathcal{A}\)
and chambers : chamber \(C C \in A \cap B\)
shows fun-eq-on (canonical-retraction \(A C\) ) (the-apartment-iso \(B A)(\cup B)\)
proof (rule fun-eq-onI)
fix \(v\) assume \(v: v \in \bigcup B\)
define \(D^{\prime} B^{\prime} g f h\)
where \(D^{\prime}=\) supchamber \(v\)
and \(B^{\prime}=\) supapartment \(D^{\prime} C\)
and \(g=\) the-apartment-iso \(B^{\prime} A\)
and \(f=\) the-apartment-iso \(B B^{\prime}\)
and \(h=\) the-apartment-iso \(B A\)
from \(D^{\prime}\)-def \(v\) apartments(2) have \(D^{\prime}\)-facts: chamber \(D^{\prime} v \in D^{\prime}\)
using apartment-simplices supchamber \(D[o f v]\) by auto
from \(B^{\prime}\)-def chambers (1) have \(B^{\prime}\)-facts: \(B^{\prime} \in \mathcal{A} \quad D^{\prime} \in B^{\prime} C \in B^{\prime}\)
using \(D^{\prime}\)-facts(1) supapartment \(D\left[\right.\) of \(\left.D^{\prime} C\right]\) by auto
from \(f\)-def apartments(2) chambers have fixespointwise \(f\left(\bigcup\left(B \cap B^{\prime}\right)\right)\)
using \(B^{\prime}\)-facts \((1,3)\) the-apartment-isoD(2)[of B \(\left.B^{\prime} C\right]\) by fast
moreover from \(v\) apartments(2) have \(v \in \bigcup\left(B \cap B^{\prime}\right)\)
using \(D^{\prime}\)-facts(2) \(B^{\prime}\)-facts \((1,2)\) apartment-vertex-set-int by fast
ultimately show canonical-retraction \(A C v=h v\)
using \(D^{\prime}\)-def \(B^{\prime}\)-def \(g\)-def \(f\)-def \(h\)-def \(v\) apartments chambers fixespointwise \(D[\) of \(\left.f \bigcup\left(B \cap B^{\prime}\right) v\right]\)
canonical-retraction-def apartment-simplices[of \(B] B^{\prime}\)-facts \((1,3)\) the-apartment-iso-comp[of \(\left.B B^{\prime} A C\right]\)
by auto
qed
lemma canonical-retraction-uniform-im:
```

    \llbracket A\in\mathcal{A; B\in\mathcal{A; chamber C; C\inA\capB; x\inB\rrbracket\Longrightarrow}}\\=|
        canonical-retraction A C'}x=\mathrm{ the-apartment-iso B A'x
    using canonical-retraction-uniform fun-eq-on-im[of-- - x] by fast
    lemma canonical-retraction-simplex-im:
assumes }A\in\mathcal{A}\mathrm{ chamber C C CA
shows canonical-retraction A C\vdashX=A
proof (rule seteqI)
fix y assume y canonical-retraction A C\vdashX
from this obtain x where x: x\inX y = canonical-retraction A C' }x\mathrm{ by fast
from x(1) obtain D where D: chamber D x\subseteqD using simplex-in-max by fast
from assms(2) D(1) obtain B where B\in\mathcal{A}D\inB C\inB
using containtwo by fast
with assms D(2) x(2) show }y\in
using the-apartment-isoD(1)[of B A]
ChamberComplexIsomorphism.surj-simplex-map
canonical-retraction-uniform-im apartment-faces[of B D x]
by fastforce
next
fix a assume }a\in
with assms show a\in canonical-retraction A C\vdashX
using canonical-retraction-simplex-retraction2[of A C a,THEN sym]
apartment-simplices
by fast
qed
lemma canonical-retraction-vertex-im:
\llbracket A\in\mathcal{A; chamber C;C\inA\rrbracket\Longrightarrow canonical-retraction A C` UX = \bigcupA}
using singleton-simplex ChamberComplex.singleton-simplex complexes
canonical-retraction-simplex-im[of A C]
by blast
lemma canonical-retraction:
assumes }A\in\mathcal{A}\mathrm{ chamber C C C A
shows ChamberComplexRetraction X (canonical-retraction A C)
proof
fix D assume chamber D
with assms
show chamber (canonical-retraction A C' D)
card (canonical-retraction A C' D) = card D
using containtwo[of C D] canonical-retraction-uniform-im
the-apartment-iso-chamber-map chamber-in-apartment
ChamberComplexIsomorphism.dim-map[OF the-apartment-isoD(1)]
by auto
next
fix v from assms
show v\in\bigcupX\Longrightarrow canonical-retraction A C (canonical-retraction A C v)=
canonical-retraction A C v
using canonical-retraction-retraction canonical-retraction-vertex-im

```
```

    by fast
    qed (simp add: canonical-retraction-def)
lemma canonical-retraction-comp-endomorphism:
\llbracket A\in\mathcal{A};B\in\mathcal{A}; chamber C; chamber D; C\inA;D\inB\rrbracket\Longrightarrow
ChamberComplexEndomorphism X
(canonical-retraction A C O canonical-retraction B D)
using canonical-retraction[of A C] canonical-retraction[of B D]
ChamberComplexRetraction.axioms(1)
ChamberComplexEndomorphism.endo-comp
by fast
lemma canonical-retraction-comp-simplex-im-subset:

```

```

        (canonical-retraction A C ○ canonical-retraction B D)\vdashX\subseteqA
    using canonical-retraction[of B D] ChamberComplexRetraction.simplex-map
        canonical-retraction-simplex-im[of A C]
    by (force simp add: image-comp[THEN sym])
    lemma canonical-retraction-comp-apartment-endomorphism:
\llbracketA\in\mathcal{A}; B\in\mathcal{A}; chamber C; chamber D; C\inA;D\inB\rrbracket\Longrightarrow
ChamberComplexEndomorphism A
(restrict1 (canonical-retraction A C ○ canonical-retraction B D) (UA))
using ChamberComplexEndomorphism.restrict-endo[of X - A]
canonical-retraction-comp-endomorphism[of A B C D] subcomplexes[of A]
canonical-retraction-comp-simplex-im-subset[of A B C D]
apartment-simplices[of A]
by auto
end

```

\subsection*{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}\mathrm{ chamber C chamber D C}\inA D\in
shows ChamberComplex.chamber-distance A C D = chamber-distance C D
proof (cases C=D)
case True with assms(1) show ?thesis
using apartment-chamber-distance-def chamber-distance-def by simp

```
```

next
case False
define Cs Dsf
where Cs =(ARG-MIN length Cs. ChamberComplex.gallery A (C\#Cs@[D]))
and Ds =(ARG-MIN length Ds.gallery (C\#Ds@[D]))
and f}=\mathrm{ canonical-retraction A C
from assms(2,3) False Ds-def have 1: gallery (C\#Ds@[D])
using gallery-least-length by fast
with assms(1,2,4,5) f-def have gallery (C \# f=Ds @ [D])
using canonical-retraction ChamberComplexRetraction.gallery-map[of X]
canonical-retraction-simplex-retraction2
by fastforce
moreover from f-def assms(1,2,4) have set (f\modelsDs)\subseteqA
using 1 galleryD-chamber chamberD-simplex
canonical-retraction-simplex-im[of A C]
by auto
ultimately have ChamberComplex.gallery A (C \# f\modelsDs @ [D])
using assms(1,4,5) gallery-in-apartment by simp
with assms(1) Ds-def False
have ChamberComplex.chamber-distance A C D \leq chamber-distance C D
using ChamberComplex.chamber-distance-le[OF complexes]
chamber-distance-def
by force
moreover from assms False Cs-def
have chamber-distance C D\leqChamberComplex.chamber-distance A C D
using chamber-in-apartment apartment-gallery-least-length
subcomplex-gallery[OF subcomplexes]
chamber-distance-le apartment-chamber-distance-def
by simp
ultimately show ?thesis by simp
qed
lemma apartment-min-gallery:
assumes A\in\mathcal{A ChamberComplex.min-gallery A Cs}
shows min-gallery Cs
proof (cases Cs rule: list-cases-Cons-snoc)
case Single with assms show ?thesis
using apartment-min-galleryD-gallery apartment-gallery galleryD-chamber
by fastforce
next
case (Cons-snoc C Ds D)
moreover with assms have min-gallery (C\#Ds@[D])
using apartment-min-galleryD-gallery[of A Cs] apartment-gallery[of A Cs]
apartment-galleryD-chamber apartment-chamberD-simplex
ChamberComplex.min-gallery-betw-chamber-distance[
OF complexes, of A C Ds D
]
galleryD-chamber apartment-chamber-distance

```
```

        min-galleryI-chamber-distance-betw
    by auto
    ultimately show ?thesis by fast
    qed simp
lemma apartment-face-distance:
assumes }A\in\mathcal{A}\mathrm{ chamber C C C A F}\in
shows ChamberComplex.face-distance A F C = face-distance F C
proof-
define D D'
where D = closest-supchamber F C
and D' = ChamberComplex.closest-supchamber A F C
from assms D'-def have chamber-D': ChamberComplex.chamber A D'
using chamber-in-apartment ChamberComplex.closest-supchamberD(1)
complexes
by fast
with assms(1,2,4) D-def have chambers: chamber D chamber D'
using closest-supchamberD(1)[of F C] apartment-chamber
apartment-simplices
by auto
from assms(1-3)
have 1:ChamberComplex.chamber-distance A D' C = chamber-distance D' C
using chamber-D' chambers(2) apartment-chamberD-simplex
apartment-chamber-distance
by fastforce
from assms D-def D'-def have F-DD':F\subseteqD F\subseteqD'
using apartment-simplices[of A] closest-supchamberD(2) chamber-in-apartment
ChamberComplex.closest-supchamberD(2)[OF complexes]
by auto
from assms(2) obtain B where B: B\in\mathcal{A}}C\inB D\in
using chambers(1) containtwo by fast
moreover from assms B have the-apartment-iso B A' F=F
using F-DD'(1) apartment-faces the-apartment-iso-int-im by force
moreover have the-apartment-iso B A' F\subseteqthe-apartment-iso B A'}
using F-DD'(1) by fast
ultimately have chamber-distance D C \geq chamber-distance D' C
using assms(1-3) D'-def 1 chambers(1) apartment-chamber-distance[of B]
chamber-in-apartment[of B D] chamber-in-apartment[of B C]
ChamberComplexIsomorphism.chamber-map[
OF the-apartment-isoD(1), of B A]
ChamberComplex.closest-supchamber-closest[
OF complexes, of A the-apartment-iso B A`D F C]
ChamberComplexIsomorphism.chamber-distance-map[
OF the-apartment-isoD (1), of B A C]
the-apartment-iso-int-im[of B A C C]
by force
moreover from assms D-def

```
```

    have chamber-distance D C \leq chamber-distance D' C
    using closest-supchamber-closest chambers(2) F-DD'(2)
    by simp
    ultimately show ?thesis
    using assms(1) D-def D'-def face-distance-def 1
        ChamberComplex.face-distance-def[OF complexes]
    by simp
    qed
lemma apartment-face-distance-eq-chamber-distance-compare-other-chamber:
assumes }A\in\mathcal{A}\mathrm{ chamber C chamber D chamber E C CA D }\inA\quadE\in
z\triangleleftC z\triangleleftD C\not=D chamber-distance C E \leq chamber-distance D E
shows face-distance z E= chamber-distance C E
using assms apartment-chamber-distance apartment-face-distance
facetrel-subset[of z C] apartment-faces[of A C z] chamber-in-apartment
ThinChamberComplex.face-distance-eq-chamber-distance-compare-other-chamber[
OF thincomplexes, of A CDzE
]
by
auto
lemma canonical-retraction-face-distance-map:
assumes }A\in\mathcal{A}\mathrm{ chamber C chamber D C C A F`C
shows face-distance F (canonical-retraction A C' D) = face-distance F D
proof-
from assms(2,3) obtain B where B: B\in\mathcal{A }C\inB D\inB
using containtwo by fast
with assms show ?thesis
using apartment-faces[of A C F] apartment-faces[of B C F]
apartment-face-distance chamber-in-apartment the-apartment-iso-int-im
the-apartment-iso-chamber-map the-apartment-iso-apartment-simplex-map
apartment-face-distance canonical-retraction-uniform-im
ChamberComplexIsomorphism.face-distance-map[
OF the-apartment-isoD(1), of B A C D F
]
by simp
qed
end

```

\subsection*{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 ChamberComplexApartmentSystemTriangle \(=\) ChamberComplexWithApartmentSystem X \(\mathcal{A}\)
for \(X\) :: 'a set set
and \(\mathcal{A}::\) ' \(a\) set set set
+ fixes \(A B B^{\prime}::\) ' \(a\) set set
and \(C D E z::\) 'a set
assumes apartments : \(A \in \mathcal{A} B \in \mathcal{A} \quad B^{\prime} \in \mathcal{A}\)
and chambers : chamber \(C\) chamber \(D\) chamber \(E\)
and facet \(: z \triangleleft C z \triangleleft D z \triangleleft E\)
and in-apartments: \(C \in A \cap B D \in A \cap B^{\prime} E \in B \cap B^{\prime}\)
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 \(B^{\prime} D\)
abbreviation res-opp-fold- \(A \equiv\) restrict1 opp-fold- \(A(\bigcup A)\)
lemma rotate: ChamberComplexApartmentSystemTriangle \(X \mathcal{A} B^{\prime} A B D E C z\) using apartments chambers facet in-apartments chambers-ne by unfold-locales auto
lemma reflect: ChamberComplexApartmentSystemTriangle \(X \mathcal{A} A B^{\prime} B D C E z\) using apartments chambers facet in-apartments chambers-ne by unfold-locales auto
lemma facet-in-chambers: \(z \subseteq C z \subseteq D z \subseteq E\)
using facet facetrel-subset by auto
lemma \(A\)-chambers:
ChamberComplex.chamber A ChamberComplex.chamber A D using apartments(1) chambers (1,2) in-apartments (1,2) chamber-in-apartment by auto
lemma res-fold- \(A\) - \(A\)-chamber-image:
ChamberComplex.chamber \(A F \Longrightarrow\) res-fold \(-A\) ' \(F=\) fold- \(A\) ' \(F\)
using apartments(1) apartment-chamberD-simplex restrict1-image
by fastforce
lemma the-apartment-iso-middle-im: the-apartment-iso \(A B\) ' \(D=E\) proof (rule ChamberComplexIsomorphism.thin-image-shared-facet)
from apartments \((1,2)\) chambers (1) in-apartments (1)
show ChamberComplexIsomorphism A B (the-apartment-iso AB)
using the-apartment-isoD (1)
by fast
from apartments(2) chambers(3) in-apartments(3)
show ChamberComplex.chamber B E ThinChamberComplex \(B\)
using chamber-in-apartment thincomplexes
by auto
```

    from apartments(1,2) in-apartments(1) have z\inA\capB
        using facet-in-chambers(1) apartment-faces by fastforce
    with apartments(1,2) chambers(1) in-apartments(1) chambers-ne(3) facet(3)
    show the-apartment-iso A B'z}\triangleleftEE\not=the-apartment-iso A B'C
    using the-apartment-iso-int-im
    by auto
    qed (
rule A-chambers(1), rule A-chambers(2), rule facet(1), rule facet(2),
rule chambers-ne(1)[THEN not-sym]
)
lemma inside-canonical-retraction-chamber-images:
canonical-retraction $B C{ }^{\prime} C=C$
canonical-retraction $B C^{\prime} D=E$
canonical-retraction $B C^{\prime} E=E$
using apartments $(1,2)$ chambers $(1,2)$ in-apartments
canonical-retraction-simplex-retraction2[of B C C]
canonical-retraction-uniform-im the-apartment-iso-middle-im canonical-retraction-simplex-retraction2
by auto
lemmas in-canretract-chimages $=$
inside-canonical-retraction-chamber-images
lemma outside-canonical-retraction-chamber-images:
canonical-retraction $A D{ }^{\prime} C=C$
canonical-retraction $A D{ }^{‘} D=D$
canonical-retraction $A D^{\prime} E=C$
using ChamberComplexApartmentSystemTriangle.in-canretract-chimages[ OF rotate
]
by auto
lemma fold- $A$-chamber-images:
fold $-A \cdot C=C$ fold $-A ' D=C$ fold $-A ' E=C$
using inside-canonical-retraction-chamber-images outside-canonical-retraction-chamber-images
image-comp[of canonical-retraction A $D$ canonical-retraction B C C] image-comp[of canonical-retraction A $D$ canonical-retraction B C D] image-comp [of canonical-retraction A $D$ canonical-retraction $B C E]$
by auto
lemmas opp-fold- A-chamber-images $=$
ChamberComplexApartmentSystemTriangle.fold-A-chamber-images[OF reflect]
lemma res-fold- $A$-chamber-images: res-fold- $A$ ' $C=C$ res-fold- $A$ ' $D=C$
using in-apartments $(1,2)$ fold-A-chamber-images $(1,2)$
res-fold- $A$ - $A$-chamber-image $A$-chambers $(1,2)$
by
auto

```
```

lemmas res-opp-fold-A-chamber-images }
ChamberComplexApartmentSystemTriangle.res-fold-A-chamber-images[OF reflect]
lemma fold-A-fixespointwise1: fixespointwise fold-A C
using apartments(1,2) chambers(1,2) in-apartments(1,2)
canonical-retraction-simplex-retraction1
by (auto intro: fixespointwise-comp)
lemmas opp-fold-A-fixespointwise2 =
ChamberComplexApartmentSystemTriangle.fold-A-fixespointwise1[OF reflect]
lemma fold-A-facet-im: fold-A' }z=
using facet-in-chambers(1) fixespointwise-im[OF fold-A-fixespointwise1] by simp
lemma fold-A-endo-X: ChamberComplexEndomorphism X fold-A
using apartments(1,2) chambers(1,2) in-apartments(1,2)
canonical-retraction-comp-endomorphism
by fast
lemma res-fold-A-endo-A: ChamberComplexEndomorphism A res-fold-A
using apartments(1,2) chambers(1,2) in-apartments(1,2)
canonical-retraction-comp-apartment-endomorphism
by fast
lemmas opp-res-fold-A-endo-A=
ChamberComplexApartmentSystemTriangle.res-fold-A-endo-A[OF reflect]
lemma fold-A-morph-A-A: ChamberComplexMorphism A A fold-A
using ChamberComplexEndomorphism.axioms(1)[OF res-fold-A-endo-A]
ChamberComplexMorphism.cong fun-eq-on-sym[OF fun-eq-on-restrict1]
by fast
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: res-fold- }A\vdashA=\mathrm{ fold- }A\vdash
using setsetmapim-restrict1[of A A fold-A] by simp
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- $A-\mathcal{C}$ - $A$-im-fold- $A-\mathcal{C}$ - $A$-im: res-fold- $A \vdash($ ChamberComplex.chamber-system $A)=$ fold-A $\vdash($ ChamberComplex.chamber-system A)
using setsetmapim-restrict1[of (ChamberComplex.chamber-system A) A] apartments(1) apartment-chamber-system-simplices

```
```

    by blast
    lemmas res-opp-fold-A-\mathcal{C}-A-im-opp-fold-A-\mathcal{C}-A-im =
ChamberComplexApartmentSystemTriangle.res-fold-A-\mathcal{C}-A-im-fold-A-\mathcal{C}-A-im[
OF reflect
]
lemma chambercomplex-fold-A-im: ChamberComplex (fold-A\vdashA)
using ChamberComplexMorphism.chambercomplex-image[OF fold-A-morph-A-A]
by simp
lemmas chambercomplex-opp-fold-A-im =
ChamberComplexApartmentSystemTriangle.chambercomplex-fold-A-im[
OF reflect
]
lemma chambersubcomplex-fold-A-im:
ChamberComplex.ChamberSubcomplex A (fold-A\vdashA)
using ChamberComplexMorphism.chambersubcomplex-image[OF fold-A-morph-A-A]
by simp
lemmas chambersubcomplex-opp-fold-A-im =
ChamberComplexApartmentSystemTriangle.chambersubcomplex-fold-A-im[
OF reflect
]
lemma fold-A-facet-distance-map:
chamber F\Longrightarrow face-distance z (fold-A`}F)=\mathrm{ face-distance z F
using apartments(1,2) chambers in-apartments(1,2) facet-in-chambers(1,2)
ChamberComplexRetraction.chamber-map[
OF canonical-retraction, of B C F
]
canonical-retraction-face-distance-map[of A D canonical-retraction B C'F]
canonical-retraction-face-distance-map
by (simp add: image-comp)
lemma fold-A-min-gallery-betw-map:
assumes chamber F chamber Gz\subseteqF
face-distance z G= chamber-distance F G min-gallery (F\#Fs@[G])
shows min-gallery (fold-A =(F\#Fs@[G]))
using assms fold-A-facet-im fold-A-facet-distance-map
ChamberComplexEndomorphism.facedist-chdist-mingal-btwmap[
OF fold-A-endo-X, of F Gz
]
by force
lemma fold- $A$-chamber-system-image-fixespointwise':
defines $\mathcal{C}-A: \mathcal{C}-A \equiv$ ChamberComplex. $\mathcal{C} A$
defines $f \mathcal{C}-A: f \mathcal{C}-A \equiv\{F \in \mathcal{C}-A$. face-distance z $F=$ chamber-distance $C F\}$

```
```

assumes F : F\infC-A
shows fixespointwise fold-A F
proof
show ?thesis
proof (cases F=C)
case True thus ?thesis
using fold-A-fixespointwise1 fixespointwise-restrict1 by fast
next
case False
from apartments(1) assms
have Achamber-F: ChamberComplex.chamber A F
using complexes ChamberComplex.chamber-system-def
by fast
define Fs where Fs = (ARG-MIN length Fs. ChamberComplex.gallery A
(C\#Fs@[F]))
show ?thesis
proof (rule apartment-standard-uniqueness-pgallery-betw, rule apartments(1))
show ChamberComplexMorphism A A fold-A
using fold-A-morph-A-A by fast
from apartments(1) show ChamberComplexMorphism A A id
using apartment-trivial-morphism by fast
show fixespointwise fold-A C
using fold-A-fixespointwise1 fixespointwise-restrict1 by fast
from apartments(1) False Fs-def
show 1:ChamberComplex.gallery A (C\#Fs@[F])
using A-chambers(1) Achamber-F apartment-gallery-least-length
by fast
from False Fs-def apartments(1) have mingal: min-gallery (C \# Fs @ [F])
using A-chambers(1) Achamber-F apartment-min-gallery
apartment-min-gallery-least-length
by fast
from apartments(1) have set-A: set (C\#Fs@[F])\subseteqA
using 1 apartment-galleryD-chamber apartment-chamberD-simplex
by fast
with apartments(1) have set (fold-A\models(C\#Fs@[F]))\subseteqA
using ChamberComplexMorphism.simplex-map[OF fold-A-morph-A-A]
by auto

```

```

            using chambers(1) apartments(1) apartment-chamber Achamber-F
                facet-in-chambers(1) mingal
                fold-A-min-gallery-betw-map[of C F] min-gallery-in-apartment
                apartment-min-gallery-pgallery
            by auto
        from apartments(1) False Fs-def
            show ChamberComplex.pgallery A (id \models(C#Fs@[F]))
            using A-chambers(1) Achamber-F
    ```
```

                ChamberComplex.pgallery-least-length[OF complexes]
            by auto
    qed
    qed
    qed
lemma fold-A-chamber-system-image:
defines }\mathcal{C}-A:\mathcal{C}-A\equiv\mathrm{ ChamberComplex.C A
defines f\mathcal{C}-A:f\mathcal{C}-A\equiv{F\in\mathcal{C}-A.face-distance z F=chamber-distance C F}
shows fold-A\vdash\mathcal{C}-A=f\mathcal{C}-A
proof (rule seteqI)
fix F assume F:F\infold-A\vdash\mathcal{C}-A
with }\mathcal{C}-A\mathrm{ have }F\in\mathcal{C}-
using ChamberComplexMorphism.chamber-system-into[OF fold-A-morph-A-A]
by fast
moreover have face-distance z F=chamber-distance C F
proof (cases F=C)
case False have F-ne-C: F\not=C by fact
from F obtain G where G: G\in\mathcal{C}-A F= fold-A' G by fast
with \mathcal{C}-A apartments(1) have G': chamber G G\inA
using apartment-chamber-system-def complexes apartment-chamber
apartment-chamberD-simplex
by auto
show ?thesis
proof (cases chamber-distance C G \leq chamber-distance D G)
case True thus face-distance z F= chamber-distance C F
using apartments(1) chambers(1,2) in-apartments(1,2) facet(1,2)
chambers-ne(1) F-ne-C G(2) G' fold-A-chamber-images(1)
facet-in-chambers(1) fold-A-facet-distance-map
fold-A-facet-im
apartment-face-distance-eq-chamber-distance-compare-other-chamber[
of ACD Gz
]
ChamberComplexEndomorphism.face-distance-eq-chamber-distance-map[
OF fold-A-endo-X, of C G z
]
by auto
next
case False thus face-distance z F= chamber-distance C F
using apartments(1) chambers(1,2) in-apartments(1,2) facet(1,2)
chambers-ne(1) F-ne-C G(2) G' fold-A-chamber-images(2)
facet-in-chambers(2) fold-A-facet-distance-map fold-A-facet-im
apartment-face-distance-eq-chamber-distance-compare-other-chamber[
of A D C Gz
]
ChamberComplexEndomorphism.face-distance-eq-chamber-distance-map[
OF fold-A-endo-X, of D G z
]
by auto

```
qed
qed (simp add: chambers(1) facet-in-chambers(1) face-distance-eq-0 chamber-distance-def)
ultimately show \(F \in f \mathcal{C}-A\) using \(f \mathcal{C}-A\) by fast
next
from \(\mathcal{C}-A f \mathcal{C}-A\) show \(\bigwedge F . F \in f \mathcal{C}-A \Longrightarrow F \in\) fold \(-A \vdash \mathcal{C}-A\)
using fold-A-chamber-system-image-fixespointwise' fixespointwise-im by blast
qed
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 \(A \Longrightarrow\) fixespointwise fold \(-A\left(\right.\) fold \(\left.-A^{‘} F\right)\)
using fold- \(A\)-chamber-system-image
fold- \(A\)-chamber-system-image-fixespointwise' \([\) of fold- \(A\) ' \(F]\)
by auto
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 \(F \Longrightarrow F \in\) fold \(-A \vdash A \Longrightarrow F \in\) fold- \(A \vdash\) ChamberComplex. \(\mathcal{C} A\)
using apartments(1)
ChamberComplexMorphism.chamber-system-image \([O F\) fold-A-morph- \(A\) - \(A\) ]
ChamberComplexMorphism.simplex-map[OF fold-A-morph-A-A]
chamber-in-apartment apartment-chamber-system-def
by fastforce
lemmas chamber-in-opp-fold-A-im \(=\)
ChamberComplexApartmentSystemTriangle.chamber-in-fold-A-im[OF reflect]
lemma simplex-in-fold- \(A\)-im-image:
assumes \(x \in\) fold- \(A \vdash A\)
shows fold- \(A\) ' \(x=x\)
proof-
from assms apartments(1) obtain \(C\)
where \(C \in\) ChamberComplex.C \(A x \subseteq\) fold- \(A\) ' \(C\)
using apartment-simplex-in-max apartment-chamber-system-def
by fast
thus ? thesis
using fold-A-chamber-system-image-fixespointwise fixespointwise-im
by blast
qed
```

lemma chamber1-notin-rfold-im: }C\not\in\mathrm{ opp-fold-A}\vdash
using chambers(1,2) facet(1,2) chambers-ne(1) facet-in-chambers(1)
min-gallery-adj adjacentI[of z] face-distance-eq-0
min-gallery-betw-chamber-distance[of D [] C]
chamber-in-opp-fold-A-im[of C] opp-fold-A-chamber-system-image
by auto
lemma fold-A-min-gallery-from1-map:
|chamber F; F\infold-A\vdashA; min-gallery (C\#Fs@[F])\rrbracket\Longrightarrow
min-gallery (C \# fold-A =Fs @ [F])
using chambers(1) chamber-in-fold-A-im fold-A-chamber-system-image
facet-in-chambers(1) fold-A-min-gallery-betw-map[of C F]
fold-A-chamber-images(1) simplex-in-fold-A-im-image
by simp
lemma fold-A-min-gallery-from2-map:
\llbracket chamber F;F\inopp-fold-A\vdashA; min-gallery (D\#Fs@ [F])\rrbracket\Longrightarrow
min-gallery ( C \# fold-A \models(Fs@[F]))
using chambers(2) facet-in-chambers(2) chamber-in-opp-fold-A-im
opp-fold-A-chamber-system-image fold-A-chamber-images(2)
fold-A-min-gallery-betw-map[of D F Fs]
by simp
lemma fold-A-min-gallery-to2-map:
assumes chamber F F \in opp-fold-A\vdashA min-gallery (F\#Fs@[D])
shows min-gallery (fold-A \models(F\#Fs) @ [C])
using assms(1,2) min-gallery-rev[of C \# fold-A \models(rev Fs @ [F])]
min-gallery-rev[OF assms(3)] fold-A-min-gallery-from2-map[of F rev Fs]
fold-A-chamber-images(2)
by (simp add: rev-map[THEN sym])
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 $\quad F \notin$ ChamberComplex.C (opp-fold-A $\vdash A$ )
proof assume $F \in$ ChamberComplex.C (opp-fold-A $\vdash A$ )
hence $F: F \in$ res-opp-fold- $A \vdash$ ChamberComplex.C $A$
using res-opp-fold- $A$ - $A$-im-opp-fold- $A$ - $A$-im

```
```

        ChamberComplexEndomorphism.image-chamber-system[
        OF opp-res-fold-A-endo-A
    ]
    by simp
    hence F':}FF\in\mathrm{ opp-fold-A }\vdash\mathrm{ ChamberComplex.C A
using res-opp-fold-A-C-A-im-opp-fold-A-\mathcal{C}-A-im by simp
from apartments(1) have Achamber-F: ChamberComplex.chamber A F
using F apartment-chamber-system-def[of A]
ChamberComplexEndomorphism.chamber-system-image[
OF opp-res-fold-A-endo-A
]
by auto
from apartments(1) have F-ne-C:F\not=C
using F' apartment-chamber-system-simplices[of A] chamber1-notin-rfold-im
by auto
have fixespointwise opp-fold-A C
proof (rule apartment-standard-uniqueness-pgallery-betw, rule apartments(1))
show ChamberComplexMorphism A A opp-fold-A
using opp-fold-A-morph-A-A by fast
from apartments(1) show ChamberComplexMorphism A A id
using apartment-trivial-morphism by fast
show fixespointwise opp-fold-A F
using F' opp-fold-A-chamber-system-image-fixespointwise by fast
define Fs where Fs = (ARG-MIN length Fs. ChamberComplex.gallery A
(F\#Fs@[C]))
with apartments(1)
have mingal:ChamberComplex.min-gallery A(F\#Fs@[C])
using A-chambers(1) Achamber-F F-ne-C
apartment-min-gallery-least-length[of A F C]
by fast
with apartments(1)
show 5: ChamberComplex.gallery A(F\#Fs@[C])
and ChamberComplex.pgallery A (id \models(F\#Fs@[C]))
using apartment-min-galleryD-gallery apartment-min-gallery-pgallery
by auto
have min-gallery (opp-fold-A =(F\#Fs)@ [D])
proof (rule opp-fold-A-min-gallery-to1-map)
from apartments(1) show chamber F
using Achamber-F apartment-chamber by fast
from assms have F\infold-A\vdashChamberComplex.C A
using apartments(1) chambers(1,2) in-apartments(1,2) facet(1,2)
chambers-ne(1) Achamber-F apartment-chamber
apartment-chamberD-simplex
apartment-face-distance-eq-chamber-distance-compare-other-chamber
apartment-chamber-system-def fold-A-chamber-system-image
apartment-chamber-system-simplices
by simp
with apartments(1) show }F\in\mathrm{ fold- }A\vdash
using apartment-chamber-system-simplices[of A] by auto

```
```

    from apartments(1) show min-gallery (F# Fs @ [C])
            using mingal apartment-min-gallery by fast
    qed
    hence min-gallery (opp-fold-A =(F#Fs@[C]))
    using opp-fold-A-chamber-images(2) by simp
    moreover from apartments(1) have set (opp-fold-A = (F#Fs@[C]))\subseteqA
    using 5 apartment-galleryD-chamber[of A]
                apartment-chamberD-simplex[of A]
                ChamberComplexMorphism.simplex-map[OF opp-fold-A-morph-A-A]
    by auto
    ultimately have ChamberComplex.min-gallery A (opp-fold-A =(F#Fs@[C]))
    using apartments(1) min-gallery-in-apartment by fast
    with apartments(1)
    show ChamberComplex.pgallery A (opp-fold-A =(F#Fs@[C]))
    using apartment-min-gallery-pgallery
    by fast
    qed
hence opp-fold-A' }C=C\mathrm{ using fixespointwise-im by fast
with chambers-ne(1) show False using opp-fold-A-chamber-images(2) by fast
qed
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 =
(ChamberComplex.\mathcal{C }(\mathrm{ fold-A }\vdashA))\cup(ChamberComplex.\mathcal{C }(\mathrm{ opp-fold-A }\vdashA))
(ChamberComplex.C (fold-A\vdashA)) \cap (ChamberComplex.C }(\mathrm{ opp-fold-A}\vdash\vdashA))
{}
proof (rule seteqI)
fix F assume F: F \in ChamberComplex.C A
with apartments(1) have F':ChamberComplex.chamber A F F\inA
using apartment-chamber-system-def apartment-chamber-system-simplices
apartment-chamber
by auto
from }\mp@subsup{F}{}{\prime}(1)\mathrm{ apartments(1) have F}\mp@subsup{F}{}{\prime\prime}:\mathrm{ chamber F
using apartment-chamber by auto
show }F\in(\mathrm{ ChamberComplex.C (fold-A}\vdash\vdashA))
(ChamberComplex.C (opp-fold-A\vdashA))
proof (cases chamber-distance C F\leqchamber-distance D F)
case True thus ?thesis
using apartments(1) chambers(1,2) in-apartments(1,2) facet(1,2)
chambers-ne(1) F F'(2) F'' fold-A-chamber-system-image
apartment-face-distance-eq-chamber-distance-compare-other-chamber

```
```

            ChamberComplexMorphism.image-chamber-system[OF fold-A-morph-A-A]
        by simp
    next
    case False thus ?thesis
        using apartments(1) chambers(1,2) in-apartments(1,2) facet(1,2)
            chambers-ne(1) F F'(2) F'\prime opp-fold-A-chamber-system-image
                    apartment-face-distance-eq-chamber-distance-compare-other-chamber
            ChamberComplexMorphism.image-chamber-system[OF opp-fold-A-morph-A-A]
        by simp
    qed
    next
fix }
assume F:F\in(ChamberComplex.C (fold-A\vdashA)) \cup
(ChamberComplex.C (opp-fold-A\vdashA))
thus F}\in\mathrm{ ChamberComplex.C A
using ChamberComplexMorphism.image-chamber-system-image[
OF fold-A-morph-A-A
]
ChamberComplexMorphism.image-chamber-system-image[
OF opp-fold-A-morph-A-A
]
by fast
next
show (ChamberComplex.C (fold-A\vdashA)) \cap
(ChamberComplex.C (opp-fold-A\vdashA))={}
using closer-to-chamber1-not-in-rfold-im-chamber-system
closer-to-chamber2-not-in-fold-im-chamber-system
by force
qed
lemma fold-A-im-min-gallery':
assumes ChamberComplex.min-gallery (fold-A\vdashA) (C\#Cs)
shows ChamberComplex.min-gallery A (C\#Cs)
proof (cases Cs rule: rev-cases)
case Nil with apartments(1) show ?thesis
using A-chambers(1) ChamberComplex.min-gallery-simps(2)[OF complexes]
by simp
next
case (snoc Fs F)
from assms snoc apartments(1)
have ch: }\forallH\inset(C\#Fs@[F]). ChamberComplex.chamber A H
using ChamberComplex.min-galleryD-gallery
ChamberComplex.galleryD-chamber
chambercomplex-fold-A-im
ChamberComplex.subcomplex-chamber[OF complexes]
chambersubcomplex-fold-A-im
by fastforce
with apartments(1) have ch-F: chamber F using apartment-chamber by simp
have ChamberComplex.min-gallery A(C\#Fs@[F])

```
proof (rule ChamberComplex.min-galleryI-betw-compare, rule complexes, rule apartments (1))
define Gs where Gs \(=(A R G-M I N\) length Gs. ChamberComplex.gallery A (C\#Gs@[F]))
from assms snoc show \(C \neq F\)
using ChamberComplex.min-gallery-pgallery
ChamberComplex.pgalleryD-distinct chambercomplex-fold- \(A\)-im
by fastforce
with chambers(1) apartments(1) assms snoc Gs-def
show 3: ChamberComplex.min-gallery A ( \(C \# G s @[F])\)
using ch apartment-min-gallery-least-length
by \(\operatorname{simp}\)
from assms snoc apartments(1)
show ChamberComplex.gallery \(A(C \# F s @[F])\)
using ch ChamberComplex.min-galleryD-gallery
ChamberComplex.galleryD-adj
chambercomplex-fold- \(A\)-im
ChamberComplex.gallery-def[OF complexes]
by fastforce
show length \(F s=\) length \(G s\)
proof -
from apartments(1) have set-gal: set \((C \# G s @[F]) \subseteq A\)
using 3 apartment-min-galleryD-gallery apartment-galleryD-chamber apartment-chamberD-simplex
by fast
from assms snoc have \(F\)-in: \(F \in\) fold- \(A \vdash A\)
using ChamberComplex.min-galleryD-gallery
ChamberComplex.galleryD-chamber
ChamberComplex.chamberD-simplex chambercomplex-fold-A-im
by fastforce
with apartments(1) have min-gallery ( \(C \#\) fold- \(A \models G s @[F]\) )
using ch-F 3 apartment-min-gallery fold-A-min-gallery-from1-map by fast
moreover have set \((\) fold- \(A \models(C \# G s @[F])) \subseteq A\)
using set-gal
ChamberComplexMorphism.simplex-map[OF fold-A-morph-A-A]
by auto
ultimately have ChamberComplex.min-gallery \(A(C \#\) fold- \(A \models G s\) @ \([F])\)
using apartments(1) F-in min-gallery-in-apartment
fold-A-chamber-images(1) fold-A-chamber-system-image-fixespointwise
simplex-in-fold-A-im-image
by simp
moreover have set \((\) fold \(-A \models(C \# G s @[F])) \subseteq\) fold- \(A \vdash A\)
using set-gal by auto
ultimately show ?thesis
using assms snoc apartments(1) F-in fold-A-chamber-images(1)
simplex-in-fold-A-im-image
ChamberComplex.min-gallery-in-subcomplex [
OF complexes, OF - chambersubcomplex-fold-A-im
```

                ]
                        ChamberComplex.min-gallery-betw-uniform-length[
                        OF chambercomplex-fold-A-im, of C fold-A \modelsGs F Fs
                ]
            by simp
        qed
    qed
with snoc show ?thesis by fast
qed
lemma fold-A-im-min-gallery:
ChamberComplex.min-gallery (fold-A\vdashA) (C\#Cs)\Longrightarrow min-gallery (C\#Cs)
using apartments(1) fold-A-im-min-gallery' apartment-min-gallery by fast
lemma fold-A-comp-fixespointwise:
fixespointwise (fold-A ○ opp-fold- $A$ ) $(\bigcup($ fold- $A \vdash A))$
proof (rule apartment-standard-uniqueness, rule apartments(1))
have fun-eq-on (fold-A ○ opp-fold-A) (res-fold-A ○ res-opp-fold-A) ( $\cup A$ )
using ChamberComplexEndomorphism.vertex-map[OF opp-res-fold-A-endo-A] fun-eq-onI[of $\bigcup A$ fold-A ○ opp-fold- $A]$
by auto
thus ChamberComplexMorphism $($ fold- $A \vdash A) A($ fold-A $\circ$ opp-fold- $A)$
using ChamberComplexEndomorphism.endo-comp[ OF opp-res-fold- $A$-endo- $A$ res-fold- $A$-endo- $A$

```

``` ChamberComplexEndomorphism.axioms(1) ChamberComplexMorphism.cong ChamberComplexMorphism.restrict-domain chambersubcomplex-fold-A-im
by fast
from apartments(1) show ChamberComplexMorphism (fold-A \(\vdash A\) ) A id using ChamberComplexMorphism.restrict-domain apartment-trivial-morphism chambersubcomplex-fold- \(A\)-im
by fast
from apartments(1) show ChamberComplex.chamber (fold-AトA) C
using \(A\)-chambers(1) apartment-chamberD-simplex fold-A-chamber-images(1) ChamberComplex.chamber-in-subcomplex \([\)
OF complexes, OF - chambersubcomplex-fold- \(A\)-im, of \(C\)
]
by fast
show fixespointwise (fold- \(A \circ\) opp-fold- \(A\) ) \(C\)
proof-
from facet(1) obtain \(v\) where \(v: v \notin z C=\) insert \(v z\)
using facetrel-def[of z C] by fast
have fixespointwise (fold- \(A \circ\) opp-fold- \(A\) ) (insert \(v z\) )
```

```
    proof (rule fixespointwise-insert, rule fixespointwise-comp)
    show fixespointwise opp-fold-A z
        using facet-in-chambers(2) fixespointwise-subset[of opp-fold-A D z]
                opp-fold-A-fixespointwise2
            by fast
    show fixespointwise fold-A z
        using facet-in-chambers(1) fixespointwise-subset[of fold-A C z}
                fold-A-fixespointwise1
            by fast
    have (fold-A \circ opp-fold-A)' C=C
        using fold-A-chamber-images(2) opp-fold-A-chamber-images(2)
        by (simp add: image-comp[THEN sym])
    with v(2) show (fold-A ○ opp-fold-A)'(insert vz)= insert vz by simp
qed
with v(2) show ?thesis by fast
qed
show \Cs. ChamberComplex.min-gallery (fold-A\vdashA) (C # Cs)\Longrightarrow
            ChamberComplex.pgallery A ((fold-A ○ opp-fold-A) \models (C#Cs))
proof
    fix Cs assume Cs: ChamberComplex.min-gallery (fold-A\vdashA) (C # Cs)
    show ChamberComplex.pgallery A ((fold-A \circ opp-fold-A) \vDash (C # Cs))
    proof (cases Cs rule: rev-cases)
        case Nil with apartments(1) show ?thesis
            using fold-A-chamber-images(2) opp-fold-A-chamber-images(2)
                A-chambers(1) ChamberComplex.pgallery-def[OF complexes]
            by (auto simp add: image-comp[THEN sym])
next
    case (snoc Fs F)
    from Cs snoc apartments(1)
            have F:F\infold-A\vdashA ChamberComplex.chamber A F
            using ChamberComplex.min-galleryD-gallery[
                OF chambercomplex-fold-A-im
                    ]
                        ChamberComplex.galleryD-chamber[
                            OF chambercomplex-fold-A-im,of C#Fs@[F]
                    ]
                        ChamberComplex.chamberD-simplex[OF chambercomplex-fold-A-im]
                ChamberComplex.subcomplex-chamber[
                    OF complexes, OF - chambersubcomplex-fold-A-im
                    ]
            by auto
        from F(2) apartments(1) have F': chamber F
            using apartment-chamber by fast
        with F(1) apartments(1)
            have zF-CF: face-distance z F= chamber-distance C F
            using chamber-in-fold-A-im[of F] fold-A-chamber-system-image
            by auto
    have min-gallery (C # fold-A \models(opp-fold-A \modelsFs @ [opp-fold-A`F]))
```

```
proof (rule fold-A-min-gallery-from2-map)
    from Cs snoc
        have Cs': ChamberComplex.gallery (fold-A\vdashA) (C#Fs@[F])
        using ChamberComplex.min-galleryD-gallery chambercomplex-fold-A-im
        by fastforce
    with apartments(1) have chF: ChamberComplex.chamber A F
        using ChamberComplex.galleryD-chamber chambercomplex-fold-A-im
            ChamberComplex.subcomplex-chamber[OF complexes]
            chambersubcomplex-fold-A-im
        by fastforce
    with apartments(1) show chamber (opp-fold-A`F)
        using ChamberComplexMorphism.chamber-map opp-fold-A-morph-A-A
                apartment-chamber
    by fast
    from apartments(1) show opp-fold-A` }F\in\mathrm{ opp-fold-A }\vdash
        using chF ChamberComplex.chamberD-simplex complexes by fast
    from Cs snoc apartments(1)
        show min-gallery (D # opp-fold-A =Fs @ [opp-fold-A`F])
        using chF Cs' opp-fold-A-min-gallery-from1-map apartment-chamber
        ChamberComplex.chamberD-simplex
        ChamberComplex.galleryD-chamber
        chambercomplex-fold-A-im fold-A-im-min-gallery
    by fastforce
qed
with snoc have min-gallery (fold-A \models(opp-fold-A\models(C#Cs)))
    using fold-A-chamber-images(2) opp-fold-A-chamber-images(2) by simp
with Cs apartments(1)
    have ChamberComplex.min-gallery A
        (fold-A }=(\mathrm{ opp-fold-A}\models(C#Cs))
    using ChamberComplex.min-galleryD-gallery[
        OF chambercomplex-fold-A-im, of C#Cs
        ]
        ChamberComplex.galleryD-chamber[
            OF chambercomplex-fold-A-im, of C#Cs
        ]
        ChamberComplex.subcomplex-chamber[
            OF complexes,OF - chambersubcomplex-fold-A-im
        ]
        apartment-chamberD-simplex
        ChamberComplexMorphism.simplex-map[OF opp-fold-A-morph-A-A]
        ChamberComplexMorphism.simplex-map[OF fold-A-morph-A-A]
    by (force intro: min-gallery-in-apartment)
with apartments(1)
    have ChamberComplex.pgallery A (fold-A \models(opp-fold-A \models(C#Cs)))
    using apartment-min-gallery-pgallery
    by fast
thus ?thesis
    using ssubst[
        OF setlistmapim-comp, of \lambdaCs. ChamberComplex.pgallery A Cs
```

```
                |
            by fast
    qed
qed
from apartments(1)
    show \bigwedgeCs. ChamberComplex.min-gallery (fold-A\vdashA) Cs\Longrightarrow
        ChamberComplex.pgallery A (id }\models=Cs
    using chambersubcomplex-fold-A-im
        ChamberComplex.min-gallery-pgallery[OF chambercomplex-fold-A-im]
        ChamberComplex.subcomplex-pgallery[OF complexes, of A fold-A\vdash A]
    by simp
qed
lemmas opp-fold-A-comp-fixespointwise =
    ChamberComplexApartmentSystemTriangle.fold-A-comp-fixespointwise[OF reflect]
lemma fold-A-fold:
    ChamberComplexIsomorphism (opp-fold-A\vdashA) (fold-A\vdashA) fold-A
proof (rule ChamberComplexMorphism.isoI-inverse)
    show ChamberComplexMorphism (opp-fold-A\vdashA) (fold-A\vdashA) fold-A
    using ChamberComplexMorphism.restrict-domain
                ChamberComplexMorphism.restrict-codomain-to-image
                ChamberComplexMorphism.cong fun-eq-on-sym[OF fun-eq-on-restrict1]
                ChamberComplexEndomorphism.axioms(1) res-fold-A-endo-A
                chambersubcomplex-opp-fold-A-im
    by fast
    show ChamberComplexMorphism (fold-A\vdashA) (opp-fold-A\vdashA) opp-fold-A
        using ChamberComplexMorphism.restrict-domain
                ChamberComplexMorphism.restrict-codomain-to-image
                ChamberComplexMorphism.cong fun-eq-on-sym[OF fun-eq-on-restrict1]
                ChamberComplexEndomorphism.axioms(1) opp-res-fold-A-endo-A
                chambersubcomplex-fold-A-im
    by fast
qed (rule opp-fold-A-comp-fixespointwise, rule fold-A-comp-fixespointwise)
lemma res-fold-A: ChamberComplexFolding A res-fold-A
proof (rule ChamberComplexFolding.intro)
    have ChamberComplexEndomorphism A (res-fold-A)
        using res-fold-A-endo-A by fast
    thus ChamberComplexRetraction A (res-fold-A)
    proof (rule ChamberComplexRetraction.intro, unfold-locales)
    fix v}\mathrm{ assume }v\in\bigcup
    moreover with apartments(1) obtain C
        where C \inChamberComplex.C A v\inC
        using apartment-simplex-in-max apartment-chamber-system-def
        by fast
```

```
    ultimately show res-fold-A (res-fold-A v) = res-fold-A v
    using fold-A-chamber-system-image-fixespointwise fixespointwiseD
    by fastforce
qed
show ChamberComplexFolding-axioms A res-fold-A
proof
    fix F assume F: ChamberComplex.chamber A F F \in res-fold-A\vdashA
    from F(2) have F': F\in fold-A\vdashA
    using setsetmapim-restrict1[of A A fold-A] by simp
hence F}F\in\mathrm{ fold-A }\vdash(opp-fold-A\vdashA
    using ChamberComplexIsomorphism.surj-simplex-map[OF fold-A-fold]
    by simp
from this obtain G where G:G\inopp-fold-A\vdashAF= fold-A`'G by auto
with F(1) F' apartments(1)
    have G':ChamberComplex.chamber A G
                G\inChamberComplex.C (opp-fold-A\vdashA)
    using ChamberComplex.chamber-in-subcomplex[OF complexes]
        chambersubcomplex-fold-A-im
        ChamberComplexIsomorphism.chamber-preimage[OF fold-A-fold, of G]
        ChamberComplex.subcomplex-chamber[
            OF complexes, OF apartments(1) chambersubcomplex-opp-fold-A-im
        ]
        ChamberComplex.chamber-system-def[
                        OF chambercomplex-opp-fold-A-im
        ]
    by
        auto
from apartments(1) G(2)
    have 1:\bigwedgeH. ChamberComplex.chamber A H}^\mp@code{H\not\infold-A\vdashA^
                fold-A' }H=F\LongrightarrowH=
    using G'(2) apartment-chamber-system-def[of A]
        fold-A-opp-fold-A-chamber-systems(1)
        chambercomplex-fold-A-im ChamberComplex.chamber-system-def
        ChamberComplex.chamberD-simplex
        inj-onD[
                        OF ChamberComplexIsomorphism.inj-on-chamber-system,
                OF fold-A-fold
            ]
    by blast
with apartments(1)
    have \H. ChamberComplex.chamber A H}\wedge H\not\inres-fold-A\vdashA
                res-fold-A'}H=F\LongrightarrowH=
    using 1 res-fold-A-A-chamber-image apartment-chamberD-simplex
        res-fold-A-A-im-fold-A-A-im
    by auto
moreover from apartments(1) have }G\not\in\mathrm{ res-fold-A}\vdash
    using G'
```

        ChamberComplex.chamber-system-def[OF chambercomplex-fold-A-im]
    ```
                ChamberComplex.chamber-in-subcomplex[
                    OF complexes,OF - chambersubcomplex-fold-A-im
                ]
                fold-A-opp-fold-A-chamber-systems(2) res-fold-A-A-im-fold-A-A-im
            by
                    auto
    ultimately
        show \exists!G. ChamberComplex.chamber }A\mathrm{ G ^G# res-fold-A }\vdashA
            res-fold-A' G = F
    using G'(1) G(2) res-fold-A-A-chamber-image ex1I[of-G]
    by force
qed
qed
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 $X \mathcal{A}$
for $X$ :: ' $a$ set set
and $\mathcal{A}::$ ' ${ }^{\prime}$ set set set

+ assumes thick: ThickChamberComplex X
begin
abbreviation some-third-chamber $\equiv$
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 \(A \in \mathcal{A}\) chamber \(C\) chamber \(D C \sim D C \neq D C \in A \quad D \in A\)
    defines \(E \equiv\) some-third-chamber \(C D(C \cap D)\)
    defines \(B \equiv\) supapartment \(C E\)
    and \(\quad B^{\prime} \equiv\) supapartment \(D E\)
    defines \(f \equiv\) restrict1 (canonical-retraction \(A D \circ\) canonical-retraction \(B C\) )
        \((\bigcup A)\)
    and \(\quad g \equiv\) restrict1 (canonical-retraction \(A C \circ\) canonical-retraction \(B^{\prime} D\) )
        \((\bigcup A)\)
    shows \(f^{\bullet} D=C\) ChamberComplexFolding \(A f\)
        \(g^{‘} C=D\) ChamberComplexFolding \(A g\)
proof-
    from assms have 1:
        ChamberComplexApartmentSystemTriangle \(X \mathcal{A} A B B^{\prime} C D E(C \cap D)\)
        using adjacent-int-facet1 [of C D] adjacent-int-facet2[of \(C D\) ]
                some-third-chamberD-facet chamber-some-third-chamber
                some-third-chamberD-ne[of \(C C \cap D D]\) supapartmentD
    by unfold-locales auto
    from \(f\)-def \(g\)-def
        show ChamberComplexFolding A ChamberComplexFolding Ag
            \(f^{\iota} D=C g^{`} C=D\)
    using ChamberComplexApartmentSystemTriangle.res-fold-A [OF 1]
                ChamberComplexApartmentSystemTriangle.opp-res-fold-A[OF 1]
                ChamberComplexApartmentSystemTriangle.res-fold-A-chamber-images(2)[
                    OF 1
                ]
        ChamberComplexApartmentSystemTriangle.res-opp-fold-A-chamber-images(2)[
                        OF 1
            ]
        by auto
qed
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 \(E \equiv\) some-third-chamber \(C D(C \cap D)\)
    defines \(B \equiv\) supapartment \(C E\)
    and \(\quad B^{\prime} \equiv\) supapartment \(D E\)
    defines \(f \equiv\) restrict1 (canonical-retraction \(A D \circ\) canonical-retraction \(B C\) )
                \((\bigcup A)\)
    and \(\quad g \equiv\) restrict1 (canonical-retraction \(A C \circ\) canonical-retraction \(B^{\prime} D\) )
        \((\bigcup A)\)
    shows OpposedThinChamberComplexFoldings AfgC
proof (rule OpposedThinChamberComplexFoldings.intro)
    from assms show ChamberComplexFolding A \(f\) ChamberComplexFolding A \(g\)
```

using apartments-have-many-foldings1 (2,4)[of A C D] by auto show OpposedThinChamberComplexFoldings-axioms AfgC proof (
unfold-locales, rule chamber-in-apartment, rule assms(1), rule assms(6), rule assms(2)
)
from $\operatorname{assms}(1-7) E$-def $B$-def $B^{\prime}-\operatorname{def} g$-def $f$-def
have $g C: g^{\prime} C=D$
and $f D: f^{\prime} D=C$
using apartments-have-many-foldings1(1)[of A C D]
apartments-have-many-foldings1 (3)[of A C D]
by auto
with $\operatorname{assms}(4,5)$ show $C \sim g^{`} C C \neq g^{`} C f^{\prime} g^{`} C=C$ by auto qed
qed (rule thincomplexes, rule assms(1))
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 $\exists f g$. OpposedThinChamberComplexFoldings $A f g C \wedge D=g^{`} C$
proof
define $E$ where $E=$ some-third-chamber $C D(C \cap D)$
define $B$ where $B=$ supapartment $C E$
define $f$ where $f=$ restrict1 (canonical-retraction A $D \circ$ canonical-retraction $B C)(\bigcup A)$
show $\exists g$. OpposedThinChamberComplexFoldings $A f g C \wedge D=g^{\prime} C$
proof
define $B^{\prime}$ where $B^{\prime}=$ supapartment $D E$
define $g$ where $g=$ restrict1 (canonical-retraction $A C$ canonical-retraction
$\left.B^{\prime} D\right)(\bigcup A)$
from assms $E$-def $B$-def $f$-def $B^{\prime}$-def $g$-def
show OpposedThinChamberComplexFoldings AfgC^D=g'C
using apartments-have-many-foldings1 (3)[of A C D]
apartments-have-many-foldings2
by auto
qed
qed
lemma apartments-have-many-foldings:
assumes $A \in \mathcal{A} C \in A$ chamber $C$
shows ThinChamberComplexManyFoldings A $C$

## proof (

rule ThinChamberComplex.ThinChamberComplexManyFoldingsI,
rule thincomplexes, rule assms(1), rule chamber-in-apartment,
rule assms(1), rule assms(2), rule assms(3)
)
from $\operatorname{assms}(1)$
show $\bigwedge C D$. ChamberComplex.chamber $A C \Longrightarrow$
ChamberComplex.chamber $A D \Longrightarrow C \sim D \Longrightarrow$ $C \neq D \Longrightarrow$

```
            \existsfg. OpposedThinChamberComplexFoldings A f gC\wedgeD=g'C
    using apartments-have-many-foldings3 apartment-chamber
        apartment-chamberD-simplex
    by simp
qed
theorem apartments-are-coxeter:
    A\in\mathcal{A \Longrightarrow\existsS::'a permutation set. (}
        CoxeterComplex S ^
        ( \exists\psi. ChamberComplexIsomorphism A (CoxeterComplex.TheComplex S) \psi)
    )
    using no-trivial-apartments apartment-simplex-in-max[of A]
        apartment-chamberD-simplex[of A] apartment-chamber[of A]
        apartments-have-many-foldings[of A]
        ThinChamberComplexManyFoldings.ex-iso-to-coxeter-complex[of A]
    by fastforce
corollary apartments-are-uniformly-coxeter:
    assumes }X\not={
    shows }\existsS::'a permutation set. CoxeterComplex S ^
        ( }\forallA\in\mathcal{A}.\exists\psi
                        ChamberComplexIsomorphism A (CoxeterComplex.TheComplex S) \psi
            )
proof-
    from assms obtain C where C: chamber C using simplex-in-max by fast
    from this obtain }A\mathrm{ where A: A}\mathcal{A}C\inA\mathrm{ using containtwo by fast
    from A(1) obtain S :: 'a permutation set and \psi
        where S: CoxeterComplex S
        and }\psi\mathrm{ : ChamberComplexIsomorphism A (CoxeterComplex.TheComplex S) }
        using apartments-are-coxeter
        by fast
    have }\forallB\in\mathcal{A}.\exists\varphi\mathrm{ .
            ChamberComplexIsomorphism B (CoxeterComplex.TheComplex S) \varphi
    proof
        fix }B\mathrm{ assume B: B}\mathcal{A
        hence }B\not={}\mathrm{ using no-trivial-apartments by fast
        with B obtain C' where C': chamber C' C'}\mp@subsup{C}{}{\prime}\in
            using apartment-simplex-in-max apartment-chamberD-simplex
                apartment-chamber[OF B]
            by force
        from C C'(1) obtain }\mp@subsup{B}{}{\prime}\mathrm{ where }\mp@subsup{B}{}{\prime}\in\mathcal{A}C\in\mp@subsup{B}{}{\prime}\mp@subsup{C}{}{\prime}\in\mp@subsup{B}{}{\prime
            using containtwo by fast
        with A B C C' \psi
            show \exists\varphi. ChamberComplexIsomorphism B
                    (CoxeterComplex.TheComplex S) \varphi
            using strong-intersecttwo
                ChamberComplexIsomorphism.iso-comp[of B'A - \psi ]
                ChamberComplexIsomorphism.iso-comp[of B B}
            by blast
```

qed
with $S$ show ?thesis by auto
qed
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.

